B-tree Implementation: Design Document

Overview

This document outlines the design for implementing a B-tree, a self-balancing tree data structure optimized for systems that read and write large blocks of data like databases and file systems. The key architectural challenge is maintaining the B-tree invariants (node key/child counts, sorted order) during insertions and deletions while ensuring disk-friendly operations through node splitting, merging, and borrowing.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

1. Context and Problem Statement

> **Milestone(s):** This foundational section corresponds primarily to Milestone 1 (understanding B-tree node structure) and provides essential context for all subsequent milestones by explaining why B-trees exist and what problems they solve.

Imagine you're managing a library with 10 million books. When someone asks for a specific book, you could walk through every shelf checking each spine—a linear search that would take months. Instead, you use a catalog system: first you check the main catalog (organized by topic), then a section guide, then a specific shelf index, and finally locate the book. This multi-level indexing is exactly what B-trees provide for computer systems, but with a crucial optimization: each level of the index is designed to fit perfectly into the physical storage units (pages) of disk drives, minimizing expensive disk seeks.

The Disk Access Problem and B-tree Solution

Traditional in-memory balanced trees like AVL trees or Red-Black trees work beautifully when your entire dataset fits in RAM. Each node typically contains one key and two child pointers, creating tall, narrow trees. For memory operations where pointer dereferencing costs roughly the same regardless of location, this is optimal. However, when data exceeds available memory and must live on disk (as in databases or file systems), a fundamental hardware reality changes everything: **disk access is not uniform**.

> **Key Insight:** The primary cost in disk-based systems isn't CPU cycles for comparisons—it's the mechanical movement of disk heads to different positions (seeks) and the time waiting for the right sector to rotate under the head (rotational latency). Reading 1 byte from disk takes essentially the same time as reading an entire disk block (typically 4KB).

Consider this physical reality: if you store one key per disk block (node) as in a binary search tree, searching for a key among 1 million records might require ~20 disk reads ($\log_2(1{,}000{,}000) \approx 20$). Each read involves:

1. **Seek time**: Moving the read head to the correct track (3-10ms for mechanical drives)
2. **Rotational latency**: Waiting for the disk to rotate to the right sector (2-8ms)
3. **Transfer time**: Actually reading the data (negligible for small amounts)

Even with 20 reads at 10ms each, that's 200ms—unacceptably slow for database operations that might need thousands of lookups per second.

The B-tree breakthrough: Instead of storing one key per node, B-trees store **many keys per node**—exactly enough to fill a disk block. With a **minimum degree** `t` (a parameter you choose based on disk block size and key size), each node holds between `t-1` and `2t-1` keys. This "fat node" design has profound implications:

- **Height reduction**: A B-tree of degree `t` storing `n` keys has height at most $\log_t(n)$. With `t=100`, a tree of 1 million keys has height at most 3 (since $100^3 = 1{,}000{,}000$). That's 3 disk reads instead of 20!
- **Amortized disk I/O**: Each node read brings in `t` keys worth of comparison material. Even though binary searching within a node takes O(log t) comparisons, these happen in memory after the single disk read.
- **Sequential access advantage**: Keys within a node are stored in sorted arrays, allowing efficient binary search and taking advantage of CPU cache locality once the node is loaded.

> **Design Principle:** B-trees are designed around the unit of storage transfer (disk blocks/pages) rather than the unit of data comparison (individual keys). This is why they dominate disk-based indexing while binary trees dominate in-memory indexing.

The B-tree solution in technical terms: By carefully maintaining balance invariants (every node except root must be at least half-full, all leaves at same depth) and implementing split/merge operations during insert/delete, B-trees guarantee:

1. **Deterministic performance**: Always $O(\log_t n)$ disk accesses
2. **High storage utilization**: Typically 50-100% of each disk block used
3. **Stable performance**: No periodic rebalancing needed (unlike some hash tables)

Mental Model: The Library Catalog Analogy

Let's build intuition through a concrete analogy. Imagine our library with the Dewey Decimal system:

1. **The Root Node (Main Catalog Desk)**: Contains ranges of Dewey Decimal numbers and pointers to section catalogs. Example: "000-199 → Science Section Catalog," "200-399 → Literature Section Catalog," etc. This fits on one index card at the main desk.
2. **Internal Nodes (Section Catalogs)**: Each section (Science, Literature) has its own catalog with finer divisions. The Science catalog might have: "000-049 → Physics shelves," "050-099 → Chemistry

shelves," etc. Each catalog fits on one physical card.

3. **Leaf Nodes (Shelf Labels)**: At the shelves themselves, you find labels like "Physics 000-015: Thermodynamics books" listing actual books (data) on that shelf.
4. **The Search Process**: To find book "005.133" (a computer science book):
    - Start at main desk (root): Binary search shows "000-199 → Science"
    - Go to Science catalog (internal node): Binary search shows "000-049 → Physics"
    - Go to Physics shelf (leaf node): Binary search finds book "005.133" is actually here (or isn't if someone miscategorized it!)
5. **Insertion with Splitting**: When the Physics shelf becomes too full (exceeds capacity), we:
    - Take the middle book off the shelf
    - Create a new shelf
    - Put half the books on each shelf
    - Update the Science catalog to now have two entries: "000-024 → Old Physics Shelf" and "025-049 → New Physics Shelf"
6. **Deletion with Merging**: If the Chemistry shelf becomes too empty (below minimum), we either:
    - **Borrow**: Take a book from the adjacent Physics shelf (through the Science catalog)
    - **Merge**: If Physics shelf is also near minimum, combine both shelves into one and remove an entry from the Science catalog

This analogy maps directly to B-tree concepts:

- **Minimum degree `t`**: Minimum books per shelf (except possibly the first/last shelf in a section)
- **Node splitting**: Shelf getting too full
- **Median promotion**: Moving the middle book up to the section catalog
- **Borrowing/merging**: Shelf reorganization
- **Balanced depth**: All shelves are reachable through the same number of catalog lookups

> **Mental Model Summary:** Think of a B-tree as a multi-level catalog system where each catalog card (node) is sized to fit exactly one physical storage unit (disk page), and the system automatically reorganizes cards and shelves to maintain quick access while efficiently using space.

Comparison with Other Tree Structures

To understand when to choose B-trees over other tree structures, consider this comprehensive comparison:

Tree Type	Typical Keys per Node	Height for N keys	Primary Optimization Target	Disk-Friendly?	Common Use Cases	Balance Mechanism
Binary Search Tree (BST)	1	O(N) worst, O(log N) average	In-memory comparison speed	❌ No	In-memory algorithms, simple dictionaries	None (unbalanced)
AVL Tree	1	O(log N)	Guaranteed height balance for worst-case performance	❌ No	In-memory databases, real-time systems where worst-case matters	Rotations after insert/delete
Red-Black Tree	1	O(log N)	Simpler rebalancing with good amortized performance	❌ No	Standard library implementations (C++ STL map, Java TreeMap)	Color flips and rotations
B-tree	t-1 to 2t-1 (t typically 50-1000)	$O(\log_t N)$	Minimizing disk I/O, block utilization	✅ **Excellent**	Databases (MySQL, PostgreSQL), file systems (NTFS, HFS+, ext4), disk-based indices	Split/merge during operations
B+ tree	t-1 to 2t-1	$O(\log_t N)$	Range queries, sequential access	✅ **Excellent**	Database indices (especially with range scans), file systems	Split/merge, all data at leaves

Detailed Analysis:

Binary Search Trees (BSTs):

- **When to use**: Small datasets that fit entirely in memory, simple implementations, or when you need an easy-to-understand tree structure.
- **Limitations for disk**: Each node might reside in a different disk block, causing one disk read per level traversed. Worst-case height O(N) if keys are inserted in sorted order.
- **Storage overhead**: For each key, you store two child pointers (typically 8 bytes each on 64-bit systems).

AVL Trees:

- **When to use**: When guaranteed O(log N) worst-case time is critical, and the tree fits in memory. Used in real-time systems and in-memory databases.
- **Disk limitations**: Same as BSTs—each node potentially causes a disk read. Frequent rotations during insertion/deletion would cause excessive disk writes if stored on disk.
- **Balance factor**: Maintains height difference ≤ 1 between subtrees through rotations.

Red-Black Trees:

- **When to use**: General-purpose in-memory sorted containers where you want good amortized performance with simpler implementation than AVL trees. Most standard libraries use them.
- **Disk limitations**: Like AVL trees, not designed for disk storage. The approximately balanced nature (height ≤ $2\log_2(N+1)$) helps but doesn't solve the fundamental disk I/O problem.
- **Practical note**: While not disk-optimized, some systems use them for in-memory indices that occasionally flush to disk.

B-trees:

- **When to use**: Any dataset too large for memory, or when persistence to disk is required. The de facto standard for database indices and file systems.
- **Key advantages**:
  1. **Height reduction**: With t=100, a B-tree of 1 billion keys has height at most 5 ($100^5 = 10^{10}$), meaning 5 disk reads max.
  2. **Sequential reads within nodes**: Once a node is loaded, all its keys are in memory for efficient binary search.
  3. **High fanout**: Many children per internal node reduces height.
  4. **Predictable performance**: No periodic rebalancing needed.
- **Trade-offs**: More complex implementation than binary trees, overhead of maintaining minimum occupancy.

B+ trees (a common variant):

- **When to use**: Database systems where range queries are common. All actual data resides in leaf nodes, with internal nodes containing only keys for routing.
- **Advantages over B-trees**:
  1. **Better sequential access**: All leaf nodes are linked together, enabling efficient full-table scans.
  2. **More keys per internal node**: Since internal nodes don't store data, they can have higher fanout.
  3. **Consistent access time**: All data accesses require traversing to leaf level.
- **Our implementation focus**: While this document describes classic B-trees, the principles apply to B+ trees as well.

> **Architecture Decision Record: Why Implement Classic B-Trees Instead of B+ Trees?**

Decision: Implement Classic B-Trees for Educational Clarity

- **Context**: We need to choose which variant of B-tree to implement for this educational project. The core concepts of node splitting, merging, and disk-friendly design are shared across variants, but implementation details differ.
- **Options Considered**:
  1. **Classic B-trees**: Keys and associated data can be in any node. Simpler to implement for a basic key-value store.
  2. **B+ trees**: All data resides in leaf nodes; internal nodes contain only keys. Better for range queries and database indices.
  3. *B trees**: Nodes kept 2/3 full minimum (vs 1/2 for B-trees), reducing splits but adding complexity.
- **Decision**: Implement classic B-trees.
- **Rationale**:
  1. **Educational value**: Classic B-trees demonstrate all fundamental operations (search, insert, delete with rebalancing) in their most direct form.
  2. **Implementation simplicity**: No need to maintain linked lists of leaves or separate data storage.
  3. **Conceptual clarity**: Each node clearly demonstrates the "keys as separators" mental model without the additional abstraction of data-only leaves.
  4. **Foundation for variants**: Understanding classic B-trees makes it straightforward to later implement B+ or B* trees.
- **Consequences**:
  - **Positive**: Clear mapping from algorithm descriptions to implementation; easier to debug and visualize.
  - **Negative**: Less optimal for range scans compared to B+ trees; in practice, database systems usually implement B+ trees.

Option	Pros	Cons	Why Not Chosen
Classic B-tree	Direct implementation of textbook algorithm; clear mental model; all operations self-contained in each node	Less efficient for range queries; mixed data and routing keys in same nodes	**Chosen** - Best for learning fundamentals
B+ tree	Industry standard for databases; efficient range scans; higher fanout in internal nodes	More complex: need leaf node linked lists; data duplication between leaves and internal nodes	Better for production systems but adds learning overhead
*B tree**	Better space utilization (66% vs 50% minimum); fewer splits	More complex rebalancing (three-way splits); not commonly taught	Adds complexity without proportional educational benefit

Real-World Usage Context:

In practice, the choice between tree structures depends entirely on the storage medium and access patterns:

1. **In-memory data structures**: Redis uses skip lists and hash tables for its primary structures; most programming language standard libraries use Red-Black trees for their ordered map/set implementations.
2. **Database indices**: PostgreSQL, MySQL (InnoDB), Oracle, and SQL Server all use B+ trees for their table indices. The B+ tree variant provides optimal performance for both point lookups and range queries common in SQL.
3. **File systems**: NTFS (Windows), HFS+ (macOS), ext4 (Linux), and many others use B-trees or B+ trees for directory indexing and file metadata storage.
4. **Specialized systems**: Some modern databases use LSM-trees (Log-Structured Merge-trees) for write-intensive workloads, but they still often use B-trees for their in-memory components or for certain types of indices.

For this implementation project, we focus on classic B-trees because they capture the essential algorithmic insights while being approachable for implementation. The skills you learn—managing node capacities, implementing split/merge operations, maintaining invariants during modifications—directly transfer to understanding real-world database and file system implementations.

Implementation Guidance

> **Note:** Since this is the context section (not a component design section), we provide only minimal implementation guidance focused on setting up the project structure. Detailed code will appear in subsequent sections.

Technology Recommendations Table:

Component	Simple Option (Learning Focus)	Advanced Option (Production-Ready)
Language	C (standard library only)	C with platform-specific optimizations
Memory Management	Manual `malloc` / `free`	Arena allocator or slab allocator
Disk Persistence	In-memory only (no persistence)	Memory-mapped files with page caching
Key Type	Fixed-size integers (`int` or `long`)	Generic pointers with comparison function
Testing Framework	Simple test harness with assertions	Comprehensive unit test suite (Check, Unity)
Debugging Aids	Print statements and manual inspection	Graphviz output for tree visualization

Recommended File/Module Structure:

For a clean C implementation that separates concerns and makes testing easier:

```
btree-project/
├── include/                  # Public header files
│   └── btree.h              # Main B-tree API (public functions)
├── src/                      # Implementation source files
│   ├── btree.c              # B-tree operations (search, insert, delete)
│   ├── node.c               # Node manipulation functions
│   ├── disk.c               # Disk I/O simulation (if implementing persistence)
│   └── utils.c              # Utility functions (binary search, validation)
├── tests/                    # Test suite
│   ├── test_btree.c         # Comprehensive B-tree tests
│   ├── test_node.c          # Node-specific tests
│   └── test_runner.c        # Test runner main program
├── examples/                 # Example usage
│   └── basic_usage.c        # Simple demonstration program
├── docs/                     # Documentation
│   └── design.md            # This design document
└── Makefile                  # Build automation
```

Infrastructure Starter Code:

While the core B-tree algorithms are the learning focus, you'll need some basic infrastructure. Here's a complete, ready-to-use header file to start:

C

```c
/* include/btree.h - Main B-tree API */

#ifndef BTREE_H

#define BTREE_H


#include <stdbool.h>

#include <stddef.h>


/* Configuration constants */

#define DEFAULT_MIN_DEGREE 3  /* t=3 means nodes hold 2-5 keys */


/* Forward declaration of internal node structure */

typedef struct BTreeNode BTreeNode;


/* Main B-tree structure */

typedef struct {

    BTreeNode* root;          /* Pointer to root node */

    int t;                    /* Minimum degree (defines capacity bounds) */

    size_t key_count;         /* Total keys in tree (optional, for statistics) */

} BTree;


/* Search result structure */

typedef struct {

    bool found;               /* Whether the key was found */

    void* value;              /* Associated value (if any) */

    int comparisons;          /* Number of key comparisons made (for analysis) */

} SearchResult;


/* B-tree API functions */


/* Initialize a new empty B-tree with given minimum degree */
```

```c
BTree* btree_create(int min_degree);

/* Free all memory used by the B-tree */
void btree_destroy(BTree* tree);

/* Search for a key in the tree */
SearchResult btree_search(BTree* tree, int key);

/* Insert a key (and optional value) into the tree */
bool btree_insert(BTree* tree, int key, void* value);

/* Delete a key from the tree */
bool btree_delete(BTree* tree, int key);

/* Get the number of keys in the tree */
size_t btree_size(BTree* tree);

/* Get the height of the tree */
int btree_height(BTree* tree);

/* Validate tree invariants (for debugging) */
bool btree_validate(BTree* tree);

/* Print tree structure (for debugging) */
void btree_print(BTree* tree);

#endif /* BTREE_H */
```

Core Logic Skeleton Code:

For the node structure (Milestone 1), here's the skeleton you'll implement:

```c
/* src/node.c - Node implementation */

#include "btree.h"

#include <stdlib.h>

#include <stdio.h>


/* Internal node structure */

struct BTreeNode {

    int* keys;                /* Array of keys (size: 2*t-1 when full) */

    void** values;            /* Array of associated values (optional) */

    BTreeNode** children;     /* Array of child pointers (size: 2*t) */

    int num_keys;             /* Current number of keys in node */

    bool is_leaf;             /* True if leaf node (no children) */

    /* Note: For leaf nodes, children array may be NULL or all NULL */

};


/* Create a new empty node */

static BTreeNode* node_create(int t, bool is_leaf) {

    /* TODO 1: Allocate memory for the BTreeNode structure */

    /* TODO 2: Allocate arrays for keys (size 2*t-1) and values (if using) */

    /* TODO 3: If not leaf, allocate children array (size 2*t) */

    /* TODO 4: Initialize num_keys to 0 and is_leaf flag */

    /* TODO 5: For safety, initialize all child pointers to NULL */

    /* TODO 6: Return the new node */

}


/* Free a node and all its descendants recursively */

static void node_destroy(BTreeNode* node) {

    /* TODO 1: If node is NULL, return immediately */
```

```c
    /* TODO 2: If not leaf, recursively destroy all children */

    /* TODO 3: Free the keys, values, and children arrays */

    /* TODO 4: Free the node structure itself */

}


/* Find the index where key should be in this node (binary search) */

static int node_find_key_index(BTreeNode* node, int key, int* comparisons) {

    /* TODO 1: Initialize left = 0, right = node->num_keys - 1 */

    /* TODO 2: While left <= right, do binary search */

    /* TODO 3: Compare key with node->keys[mid], increment comparisons */

    /* TODO 4: If key found, return index (might be negative for child pointer logic) */

    /* TODO 5: If not found, return index where key should be inserted */

}


/* Check if node is full (has 2*t-1 keys) */

static bool node_is_full(BTreeNode* node, int t) {

    /* TODO: Return true if node->num_keys == 2*t - 1 */

}


/* Check if node is underfull (has fewer than t-1 keys) */

static bool node_is_underfull(BTreeNode* node, int t) {

    /* TODO: Return true if node->num_keys < t - 1 */

    /* Special case: root can have fewer keys */

}
```

Language-Specific Hints for C:

1. **Memory allocation**: Use `calloc` instead of `malloc` when you want zero-initialized memory (especially for child pointer arrays).
2. **Error handling**: Check every `malloc` / `calloc` return value for NULL and handle gracefully (free previous allocations and return error).

3. **Const correctness**: Use `const` for pointer parameters that shouldn't be modified (e.g., `const BTreeNode* node` in search functions).
4. **Static functions**: Mark helper functions as `static` (file scope) to avoid polluting the global namespace.
5. **Integer types**: Use `size_t` for counts and indices that can't be negative; use `int` for keys in this simple implementation.
6. **Debugging**: Compile with `-g -O0` for debugging and `-fsanitize=address` for memory error detection.

Milestone Checkpoint for Section 1:

After setting up your project structure and implementing the basic node infrastructure:

1. **Compile your code**: `gcc -c src/node.c -Iinclude -o node.o`
2. **Test node creation**: Write a simple test that creates nodes with different `t` values and checks:
   - Memory is correctly allocated
   - Arrays have the right sizes
   - Fields are properly initialized
3. **Verify with valgrind**: Run `valgrind ./your_test_program` to ensure no memory leaks.
4. **Expected behavior**: Nodes should be created and destroyed cleanly with no memory errors.
5. **Signs of trouble**: Segmentation faults indicate uninitialized pointers or array bounds errors; memory leaks mean you're not freeing all allocated memory.

---

2. Goals and Non-Goals

> **Milestone(s):** This foundational section corresponds to all milestones (1-4), establishing the functional and non-functional boundaries for the entire B-tree implementation.

This section precisely defines what the B-tree implementation will and will not do. Given the educational nature of the project, we deliberately constrain the scope to focus on core algorithmic concepts while establishing a clean foundation for potential future extensions. Clear boundaries prevent scope creep and help learners concentrate on mastering the fundamental mechanics of B-tree operations.

Functional Goals

The primary functional goal is to implement a **correct, in-memory B-tree** that adheres to all classic invariants and supports the three fundamental dictionary operations. This B-tree will serve as a pedagogical model for understanding how disk-optimized tree structures maintain balance.

> **Mental Model: A Self-Organizing Filing Cabinet** Think of the B-tree as a smart filing cabinet for index cards. Each drawer (node) can hold a fixed number of cards (keys). When a drawer becomes too full, it automatically splits into two, and a new divider card is sent to the drawer above. When a drawer becomes too empty, it either borrows a card from a neighbor or merges with one. The entire cabinet maintains a strict alphabetical order, and any card can be found by checking at most a few drawers. Our implementation builds this cabinet entirely in RAM, simulating the drawer operations that would normally occur on disk.

The following table enumerates the core operations the B-tree must support, their exact behavioral specifications, and how they map to the key invariants.

Operation	Signature (C)	Preconditions	Postconditions & Guarantees	Key Invariants Maintained
Create Tree	`BTree* btree_create(int min_degree)`	`min_degree` ≥ 2.	Returns a pointer to a new, empty `BTree` struct with `root` set to `NULL`, `t` set to `min_degree`, and `key_count` set to 0. Memory is allocated for the tree structure.	Establishes the minimum degree `t` for all future node capacity calculations.
Search Key	`SearchResult btree_search(BTree* tree, int key)`	`tree` is a valid pointer from `btree_create`. `key` is an integer.	Returns a `SearchResult` struct. If the key exists, `found` is `true` and `value` points to the associated data (or is `NULL` if no values are stored). If not found, `found` is `false`. The `comparisons` field records the total number of key comparisons performed during the search. The tree structure remains unchanged.	Demonstrates the `O(log n)` search path length. Relies on sorted keys within nodes and correct child pointer navigation.
Insert Key-Value	`bool btree_insert(BTree* tree, int key, void* value)`	`tree` is valid. `key` is an integer. `value` is an optional pointer to associated data (may be `NULL`).	If the key does **not** already exist in the tree, it is inserted along with its value. The tree is rebalanced via proactive splitting to	Guarantees all nodes (except root) have between `t-1` and `2t-1` keys. Ensures all keys in the tree remain in sorted order. Handles root split, increasing tree

Operation	Signature (C)	Preconditions	Postconditions & Guarantees	Key Invariants Maintained
			maintain all B-tree invariants. Returns `true` on successful insertion. If the key already exists, the operation is a **no-op**: the tree is unchanged, and the function returns `false`. The `key_count` in the tree is updated accordingly.	height by one when necessary.
Delete Key	`bool btree_delete(BTree* tree, int key)`	`tree` is valid. `key` is an integer.	If the key exists in the tree, it is removed. The tree is rebalanced via borrowing from siblings or merging nodes to maintain the minimum occupancy invariant. Returns `true` on successful deletion. If the key does not exist, the operation is a **no-op**: the tree is unchanged, and the function returns `false`. The `key_count` is updated.	Maintains the minimum key count invariant (all non-root nodes have at least `t-1` keys). Handles root deletion that may reduce tree height by one. Preserves sorted order.

Operation	Signature (C)	Preconditions	Postconditions & Guarantees	Key Invariants Maintained

Utility & Inspection	`size_t btree_size(BTree* tree)`
	`int btree_height(BTree* tree)`
	`bool btree_validate(BTree* tree)`
	`void btree_print(BTree* tree)`
		`tree` is valid (may be `NULL` for some functions).	Returns the total number of keys (`key_count`), the height (root to leaf, with leaves at height 0), a boolean indicating if all B-tree invariants hold, and a textual representation of the tree structure (for debugging), respectively. These functions do not modify the tree.	The validation function (`btree_validate`) is the ultimate correctness check, verifying every invariant listed in Section 4.

The sequence of these operations defines the primary data flow: creation → insertion (builds the tree) → search (queries the tree) → deletion (modifies the tree) → inspection (verifies the tree). Success is defined by the `btree_validate` function returning `true` after any sequence of these operations.

Non-Functional Goals & Non-Goals

This implementation prioritizes **clarity, correctness, and educational value** over performance optimizations or production-ready features. The non-functional goals define the quality attributes we will achieve, while the non-goals explicitly list features we are omitting, explaining why they are out of scope for this learning exercise.

Non-Functional Goals

1. **Algorithmic Complexity:** All operations must achieve their theoretical time complexities. Search, insertion, and deletion must run in `O(log n)` time, where the base of the logarithm is the minimum degree `t`. This requires efficient binary search within nodes (`O(log t)`) and a tree height of `O(log_t n)`.
2. **Memory Safety (C):** The implementation must not leak memory. Every `malloc` or `calloc` must have a corresponding `free`. The `btree_destroy` function must completely deallocate all tree nodes and the tree structure itself.
3. **Robustness to Edge Cases:** The implementation must correctly handle standard edge cases: inserting into an empty tree, deleting the only key in the tree, splitting the root, merging all the way to the root, and

attempting to delete a non-existent key.

4. **Debuggability:** The provided `btree_print` and `btree_validate` functions must offer sufficient introspection to diagnose structural issues during development. The `SearchResult.comparisons` field provides visibility into search efficiency.
5. **Configurable Node Size:** The B-tree's node capacity, determined by the minimum degree `t`, must be configurable at tree creation. This allows experimenting with different fanouts and observing their effect on tree height.

Non-Goals Explicitly excluding the following areas keeps the project focused and achievable, while the table below justifies each exclusion and suggests how they could be added later.

Feature	Status (Non-Goal)	Rationale for Exclusion	Potential Extension Path
Persistent Storage to Disk	Excluded	The core learning objective is the in-memory balancing algorithm. Disk I/O (page caching, serialization, write-ahead logging) introduces significant complexity that would distract from the core B-tree mechanics.	Add a `Pager` module that maps node IDs to disk offsets, and modify node splits/merges to write pages back to a file.
Concurrency Control (Thread Safety)	Excluded	Managing concurrent inserts, deletes, and searches with locks (e.g., B-link tree techniques) is an advanced topic beyond introductory B-tree concepts.	Protect the tree root with a mutex for coarse-grained locking, or implement fine-grained lock coupling on node traversal paths.
Variable-Length or Generic Keys	Excluded	Using integer keys simplifies comparison logic and memory management, letting learners focus on structure. Supporting strings or generic types requires custom comparators and more complex node memory layout.	Replace the `int* keys` array with a `void** keys` array and a function pointer `comparator`.
Associated Values (Full Key-Value Store)	Partially Supported	The `BTreeNode` struct includes a `values` array (`void**`), allowing storage of optional pointers. However, managing the lifecycle (allocation/freeing) of these values is **not** a goal. The implementation only stores and returns the pointer.	Make `btree_insert` accept a `value_size` and use `memcpy` to store values inline within the node or in a separate heap allocation.
Duplicate Keys	Excluded	Standard B-tree definitions prohibit duplicates. Supporting them requires modifying the search and insert logic (e.g., allowing multiple values per key or using a secondary index), which adds complexity.	Modify `node_find_key_index` to find the first occurrence, and adjust insertion to allow non-unique keys, potentially storing a list of values per key.
*B+ Tree or B Tree Variants**	Excluded	This is a classic B-tree where keys are stored in both internal nodes and leaves. B+ trees (keys only in leaves, linked leaves) are an important optimization for databases but represent a different structural variant.	Convert the B-tree to a B+ tree by removing keys from internal nodes (keeping only separators) and adding sibling pointers between leaf nodes.

Feature	Status (Non-Goal)	Rationale for Exclusion	Potential Extension Path
Advanced Optimizations (Bulk Loading, Compression)	Excluded	These are performance optimizations valuable in production systems but orthogonal to understanding the core insert/split/delete/merge algorithms.	Implement a `btree_bulk_load` function that sorts all keys and builds the tree from the bottom up for optimal packing.

> **Design Insight:** The most critical trade-off in this design is **abstraction vs. realism**. We abstract away disk persistence—the original raison d'être for B-trees—to isolate the balancing algorithm. This allows learners to build a correct, understandable mental model of node splitting and merging first. Once this core is solid, extending the design to include a pager module (for disk I/O) becomes a more manageable, separate challenge.

Common Pitfalls: Misinterpreting Scope

Learners often mistakenly believe they need to implement features from the non-goals list, or they overlook subtle aspects of the defined goals.

- ⚠️ **Pitfall: Implementing Reactive Instead of Proactive Splitting.** The design specifies *proactive* splitting during the insertion descent. A common mistake is to insert into a leaf first, find it's full, and then split (reactive). This works for leaves but fails for internal nodes if a child needs to be split during the descent. **Fix:** Always check if a child node is full *before* recursively descending into it during insertion.
- ⚠️ **Pitfall: Ignoring the `values` Array in Node Operations.** Even though value management is a non-goal, the `values` array pointer must be maintained correctly. When splitting a node or moving keys during borrowing/merging, the associated value pointer must be moved alongside its key. Forgetting this leads to search returning incorrect `value` pointers. **Fix:** Treat the key and its value as a bonded pair in all node-level operations (insert, delete, split, borrow).
- ⚠️ **Pitfall: Allowing `t=1`.** The minimum degree `t` must be at least 2. If `t=1`, the node capacity rules break down (a node would be allowed 1 to 1 key, making splitting nonsensical). **Fix:** In `btree_create`, validate that `min_degree >= 2` and return `NULL` or abort if not.

Implementation Guidance

This section provides concrete starter code and structure for the B-tree's foundational types and the goal-oriented functions. Since persistence and concurrency are non-goals, we focus on clean, memory-safe in-memory structures.

A. Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option (Post-Learning)
Memory Management	Standard `malloc` / `free` with careful manual tracking.	Use a custom arena allocator for nodes to improve locality and simplify freeing (destroy just frees the arena).
Key Comparison	Direct integer comparison (`<` , `==` , `>`).	Function pointer to a comparator for generic keys.
Tree Visualization	Recursive print to console with indentation (`btree_print`).	Generate Graphviz DOT format output to render as an image.
Validation	Recursive invariant checker (`btree_validate`).	Integrate the checker with a unit test framework (e.g., Google Test for C++).

B. Recommended File/Module Structure

Place the core B-tree implementation in two files: a header for declarations and a source file for definitions. This separates interface from implementation.

```
btree-project/
├── include/
│   └── btree.h          <-- Public API and struct definitions
├── src/
│   ├── btree.c          <-- Implementation of BTree operations
│   ├── node.c           <-- Implementation of Node-level helpers
│   └── main.c           <-- Example usage or test driver (optional)
└── tests/
    └── test_btree.c     <-- Unit tests
```

C. Infrastructure Starter Code

The following is **complete, compilable code** for the core data structures and basic tree lifecycle functions. A learner should place this in `btree.h` and start the implementation in `btree.c` .

File: `include/btree.h`

C

```c
#ifndef BTREE_H
#define BTREE_H

#include <stdbool.h>
#include <stddef.h>

// Constants
#define DEFAULT_MIN_DEGREE 3

// Core Types
typedef struct BTreeNode {
    int* keys;                  // Array of keys (size = 2*t - 1)
    void** values;              // Optional array of associated values
    struct BTreeNode** children;// Array of child pointers (size = 2*t)
    int num_keys;               // Current number of keys in node
    bool is_leaf;               // True if leaf, false if internal node
} BTreeNode;

typedef struct BTree {
    BTreeNode* root;            // Pointer to root node
    int t;                      // Minimum degree (defines capacity bounds)
    size_t key_count;           // Total keys in entire tree
} BTree;

typedef struct SearchResult {
    bool found;                 // Whether the key exists
    void* value;                // Associated value if found (may be NULL)
    int comparisons;            // Number of key comparisons performed
} SearchResult;
```

```c
// Tree Lifecycle & Public API

BTree* btree_create(int min_degree);

void btree_destroy(BTree* tree);

SearchResult btree_search(BTree* tree, int key);

bool btree_insert(BTree* tree, int key, void* value);

bool btree_delete(BTree* tree, int key);

size_t btree_size(BTree* tree);

int btree_height(BTree* tree);

bool btree_validate(BTree* tree);

void btree_print(BTree* tree);


// Node-level helper functions (internal use, but exposed for testing)

BTreeNode* node_create(int t, bool is_leaf);

void node_destroy(BTreeNode* node);

int node_find_key_index(BTreeNode* node, int key, int* comparisons);

bool node_is_full(BTreeNode* node, int t);

bool node_is_underfull(BTreeNode* node, int t);


#endif // BTREE_H
```

File: `src/btree.c` - Starter Skeleton

```c
#include "btree.h"

#include <stdio.h>

#include <stdlib.h>


// ============================================

// Tree Lifecycle Functions

// ============================================


BTree* btree_create(int min_degree) {

    // TODO 1: Validate min_degree >= 2. Return NULL if invalid.

    // TODO 2: Allocate memory for BTree struct.

    // TODO 3: Initialize fields: t = min_degree, root = NULL, key_count = 0.

    // TODO 4: Return the pointer to the new tree.

}


void btree_destroy(BTree* tree) {

    if (tree == NULL) return;

    // TODO 1: If tree->root exists, call node_destroy(tree->root) to free the entire node hierarchy.

    // TODO 2: Free the tree struct itself.

}


// ============================================

// Public API Functions (Skeletons)

// ============================================


SearchResult btree_search(BTree* tree, int key) {

    SearchResult result = {false, NULL, 0};

    if (tree == NULL || tree->root == NULL) return result;

    // TODO: Implement recursive search starting from tree->root.
```

```
    // Hint: Create a helper function `search_subtree(node, key, &comparisons)`.
    return result;
}

bool btree_insert(BTree* tree, int key, void* value) {
    if (tree == NULL) return false;
    // TODO: Implement proactive split insertion.
    // 1. If root is full, split it (creates new root, increases height).
    // 2. Call a recursive helper `insert_non_full(starting_node, key, value)`.
    // 3. If key already exists (found during descent), return false.
    // 4. On successful new insertion, increment tree->key_count and return true.
    return false;
}

bool btree_delete(BTree* tree, int key) {
    if (tree == NULL || tree->root == NULL) return false;
    // TODO: Implement deletion with borrowing/merging.
    // 1. Call a recursive helper `delete_key(node, key)`.
    // 2. If key not found, return false.
    // 3. If after deletion the root has 0 keys and is not a leaf, set its only child as
new root.
    // 4. On successful deletion, decrement tree->key_count and return true.
    return false;
}

size_t btree_size(BTree* tree) {
    return (tree == NULL) ? 0 : tree->key_count;
}
```

```c
int btree_height(BTree* tree) {
    if (tree == NULL || tree->root == NULL) return -1; // Empty tree height convention
    // TODO: Recursively compute height by following leftmost child until a leaf.
    // Height definition: leaf nodes are at height 0. Root alone (leaf) = 0.
    return 0;
}

bool btree_validate(BTree* tree) {
    if (tree == NULL) return false;
    // TODO: Implement comprehensive invariant checks.
    // 1. Check root key count: 0 <= num_keys <= 2t-1, and if root is leaf, children pointers should be NULL.
    // 2. Recursively check every node:
    //    a. Key count in [t-1, 2t-1] for non-root.
    //    b. Keys are strictly increasing within node.
    //    c. For internal nodes: num_keys + 1 == number of non-NULL children.
    //    d. Leaf check: is_leaf == (all children are NULL).
    // 3. Check that the tree's key_count matches the sum of keys in all nodes.
    return false;
}

void btree_print(BTree* tree) {
    if (tree == NULL) {
        printf("(null tree)\n");
        return;
    }
    // TODO: Implement a pretty-print or debug print.
    // Simple approach: Print each level with indentation.
```

```
    printf("B-Tree (t=%d, keys=%zu, height=%d)\n",
           tree->t, tree->key_count, btree_height(tree));
    // TODO: Call a recursive node printer.
}
```

File: `src/node.c` - Starter Skeleton

```c
#include "btree.h"

#include <stdlib.h>


BTreeNode* node_create(int t, bool is_leaf) {

    // TODO 1: Allocate memory for BTreeNode struct.

    // TODO 2: Allocate arrays: keys (size = 2*t - 1), values (same size), children (size = 2*t).

    // TODO 3: Initialize all fields: num_keys = 0, is_leaf = is_leaf.

    // TODO 4: Initialize children array to all NULLs (important for leaf nodes).

    // TODO 5: Return the new node.

    return NULL;

}


void node_destroy(BTreeNode* node) {

    if (node == NULL) return;

    // TODO 1: If node is not a leaf, recursively destroy all non-NULL children.

    // TODO 2: Free the keys, values, and children arrays.

    // TODO 3: Free the node struct itself.

}


int node_find_key_index(BTreeNode* node, int key, int* comparisons) {

    // TODO: Implement binary search within node->keys[0..num_keys-1].

    // 1. Initialize left=0, right=node->num_keys-1.

    // 2. While left <= right, compute mid, compare key with node->keys[mid].

    //    Increment (*comparisons) for each comparison.

    // 3. If key found, return the exact index.

    // 4. If not found, return the index of the first key greater than the target,

    //    or node->num_keys if all keys are smaller. This is the child index to descend into.
```

```
    return 0;
}

bool node_is_full(BTreeNode* node, int t) {
    return (node != NULL) && (node->num_keys == (2 * t - 1));
}

bool node_is_underfull(BTreeNode* node, int t) {
    return (node != NULL) && (node->num_keys < (t - 1));
}
```

D. Milestone Checkpoint: After Setting Up Foundations

After implementing just the skeleton and the `btree_create` and `btree_destroy` functions (with proper memory allocation/deallocation in `node_create` and `node_destroy`), you should be able to compile without errors and run a simple test.

Compilation Command (using GCC):

```bash
gcc -I./include -c src/btree.c src/node.c -o btree.o node.o
```

Expected Outcome: Successful compilation (no errors). Linking with a simple `main.c` that creates and destroys a tree should run without memory leaks (check with Valgrind).

Signs of Trouble:

- **Segmentation fault on `btree_destroy`:** Likely caused by not initializing the `children` array to `NULL` in `node_create`, leading to `free` on uninitialized pointers.
- **Memory leaks:** Use `valgrind ./your_program` to detect leaks. Any remaining allocations indicate missing `free` in `node_destroy` or `btree_destroy`.

E. Language-Specific Hints (C)

- **Memory Allocation:** Use `calloc` for arrays to ensure zero-initialization. For example, `node->children = calloc(2 * t, sizeof(BTreeNode*))` sets all pointers to `NULL`, which is required for leaf nodes and simplifies logic.
- **Integer Overflow:** When calculating array sizes (e.g., `2*t - 1`), ensure `t` is not so large that it causes integer overflow. For this educational project, assume `t` is small (< 1000).
- **Const Correctness:** For search functions that don't modify the tree, consider using `const` parameters (e.g., `const BTree* tree`) to signal intent, though the skeleton omits this for simplicity.

- **Header Guards:** Always use `#ifndef BTREE_H` / `#define BTREE_H` / `#endif` in headers to prevent double inclusion.

---

3. High-Level Architecture

> **Milestone(s):** This foundational section corresponds primarily to Milestone 1 (Node Structure), but also provides the overall architectural context for all subsequent milestones (2-4). Understanding how components interact is crucial before implementing search, insert, and delete operations.

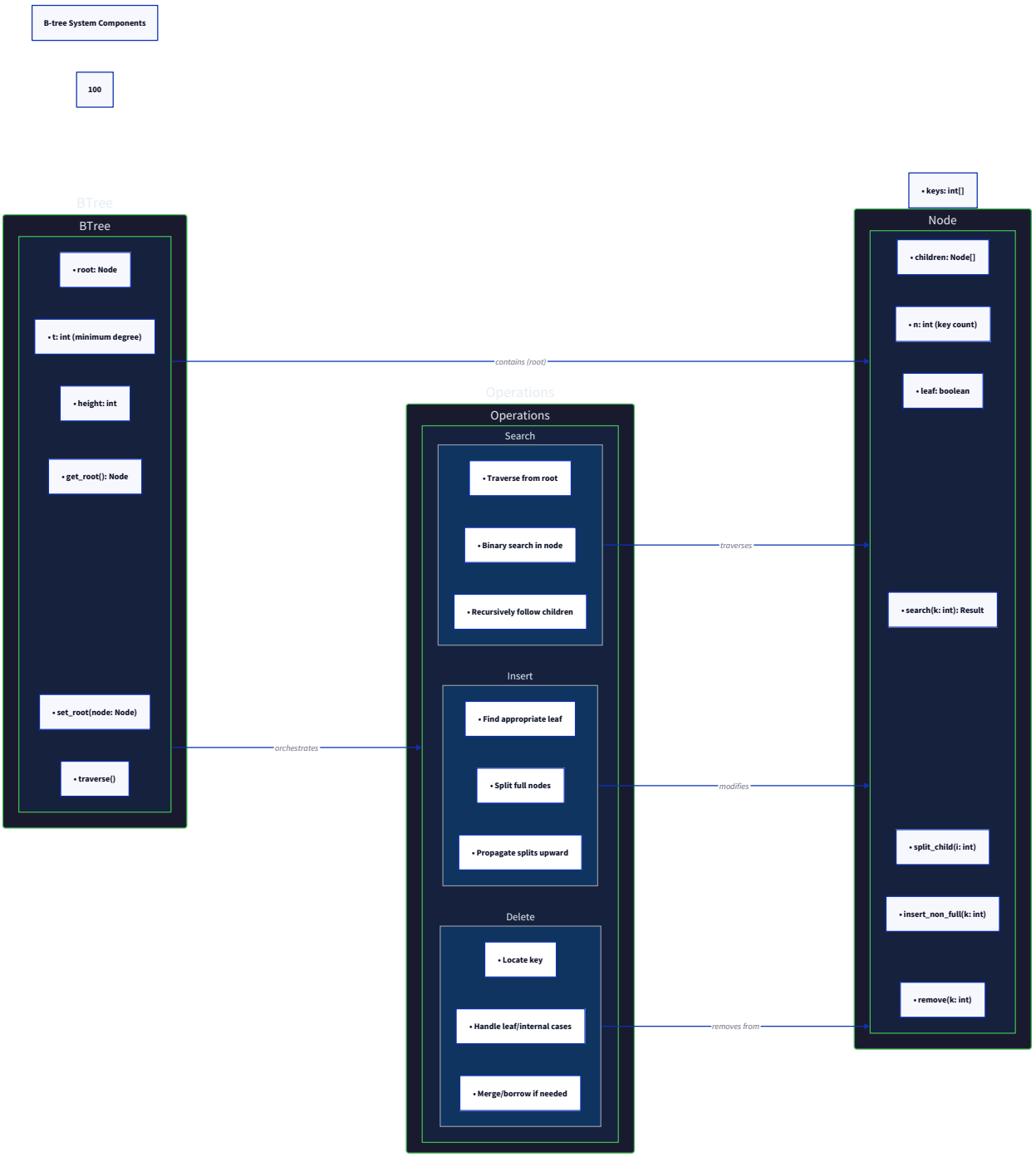
The B-tree implementation follows a layered, modular architecture designed to separate concerns between data storage (`Node`), tree management (`BTree`), and algorithmic operations (`Operations`). This separation allows each component to focus on a single responsibility, making the system easier to understand, test, and extend. Although our educational implementation is in-memory, the architecture mirrors patterns used in disk-based systems where each node corresponds to a disk page.

Component Overview

Think of the B-tree architecture as a **library building** with three distinct roles:

- **The `Node` component** is like an **individual bookshelf** within the library, holding a collection of books (keys) and possibly pointers to other shelves (child nodes). Each shelf has strict capacity rules.
- **The `BTree` component** is like the **library building's floor plan**, maintaining a reference to the main entrance (root node) and the building's structural rules (minimum degree `t`).
- **The `Operations` component** is like the **librarian's procedures**, defining the step-by-step processes for finding a book, adding a new book (which may require splitting a full shelf), or removing a book (which may require borrowing from neighbors or merging shelves).

The relationship between these components is visualized in the following diagram, which shows how `BTree` orchestrates operations that manipulate `Node` instances:



The table below details the responsibilities and key attributes of each logical component:

Component	Responsibility	Key Data it Holds	Key Behaviors it Manages
`BTree` (Orchestrator)	Serves as the main entry point and container for the entire tree structure. It maintains global properties and ensures the tree invariants are preserved after operations.	• Root node pointer • Minimum degree (`t`) • Total key count (optional)	• Creating/destroying the tree • Initiating search, insert, delete operations • Validating tree structure • Calculating tree height and size
`Node` (Data Container)	Represents a single node (or disk page) in the B-tree. Stores keys and child pointers, and knows whether it is a leaf. It is the fundamental unit of data organization and transfer.	• Array of integer keys • Array of child node pointers (for internal nodes) • Count of keys currently stored (`num_keys`) • Boolean flag indicating leaf status (`is_leaf`)	• Finding a key's position within its local array • Reporting if it is full or underfull • Splitting itself into two nodes (insert) • Borrowing a key from or merging with a sibling (delete)
`Operations` (Algorithms)	Encapsulates the core B-tree algorithms that traverse and modify the tree structure. These are implemented as functions that operate on the `BTree` and `Node` structures.	• No persistent state of its own	• Recursive tree traversal (search) • Proactive node splitting during insertion descent • Complex rebalancing (borrow/merge) during deletion

> **Key Design Insight:** The `BTree` struct acts as a thin facade. Its primary job is to hold the root pointer and configuration (`t`), delegating all complex logic to the operation functions which recursively navigate the node hierarchy. This keeps the tree's state simple and centralizes algorithm complexity in well-defined procedures.

Component Interactions and Data Flow

The components interact through a clear calling hierarchy:

1. **Client Code** calls a public function on the `BTree` component (e.g., `btree_insert`).
2. **`BTree` Function** performs initial validation, then calls a recursive internal operation function, passing the root node and other parameters.

3. **Internal Operation Function** traverses nodes, calling `Node` helper functions (like `node_find_key_index`) to navigate within each node. If structural changes are needed (split, borrow, merge), the operation function orchestrates these by creating/destroying nodes and updating pointers.
4. **`Node` Helper Functions** perform localized tasks, such as inserting a key into a node's sorted array or splitting a node's keys and children into two.

For example, during an insertion that causes a root split:

1. `btree_insert` is called on the `BTree`.
2. The internal insertion function discovers the root is full.
3. It creates a new root node and splits the old root, promoting the median key to the new root.
4. It updates the `BTree`'s root pointer to point to the new root.
5. Control returns to `btree_insert`, which updates the tree's key count.

This interaction pattern ensures that the `BTree` struct remains the single source of truth for the tree's identity (its root), while algorithms are free to manipulate the node graph.

Recommended File/Module Structure

A clean, modular code organization is essential for managing the complexity of the B-tree algorithms. The following structure separates interface from implementation, groups related functions, and provides a clear path for testing. This is particularly important in C, where module boundaries are enforced by file organization rather than language features.

The recommended directory and file structure for a C implementation is as follows:

```
btree-project/
├── include/                  # Public header files (API)
│   └── btree.h               # Main public interface: BTree, BTreeNode, SearchResult
types and public function declarations
├── src/                      # Private implementation source files
│   ├── btree.c               # Implementation of public API functions (btree_create,
btree_search, etc.)
│   ├── btree_private.h       *Private header: declarations of internal helper functions
and constants*
│   ├── node.c                # Implementation of node-level functions (node_create,
node_find_key_index, node_split, etc.)
│   └── operations.c          # Implementation of core recursive algorithms
(insert_recursive, delete_recursive, etc.)
├── tests/                    # Test suite
│   ├── test_btree.c          # Unit tests for the public API
│   ├── test_node.c           # Unit tests for node utilities
│   └── test_operations.c     # Unit tests for internal algorithms
├── examples/                 # Example programs
│   └── basic_usage.c         # Demonstrates creating a tree, inserting keys, searching,
and deleting
└── Makefile                  # Build automation
```

> *Note: `btree_private.h` is included only by `.c` files within the `src/` directory. It keeps internal implementation details (like recursive function prototypes) hidden from users of the public `btree.h` API.*

The table below justifies the purpose of each key file:

File	Purpose	Key Contents	Visibility
`include/btree.h`	Public API header. This is the only file a user of the B-tree library needs to include.	• `BTree`, `BTreeNode`, `SearchResult` type definitions (can be opaque pointers if desired) • All public function prototypes (e.g., `btree_create`, `btree_insert`) • Documentation comments for users	Public
`src/btree_private.h`	Private implementation header. Shared internal details between `.c` files in the `src/` module.	• `DEFAULT_MIN_DEGREE` constant • Prototypes for internal recursive functions (e.g., `insert_non_full`, `delete_from_subtree`) • Helper function prototypes used across operations	Private (internal to library)
`src/btree.c`	Implementation of the public API facade. Contains functions that initiate operations and manage the `BTree` struct.	• Definitions of `btree_create`, `btree_destroy`, `btree_size`, `btree_height` • `btree_search`, `btree_insert`, `btree_delete` implementations (which call internal functions in `operations.c`) • Public validation and print functions	Private implementation
`src/node.c`	Implementation of all node-level utilities. Functions here operate on a single `BTreeNode`.	• `node_create`, `node_destroy` • `node_find_key_index` (binary search) • `node_is_full`, `node_is_underfull` • `node_split_child` (critical for insertion) • `node_borrow_from_left`, `node_merge_with_right` (for deletion)	Private implementation
`src/operations.c`	Implementation of the core recursive algorithms that traverse and modify the tree.	• `search_recursive` (called by `btree_search`) • `insert_non_full` (recursive insertion helper) • `delete_from_subtree`	Private implementation

File	Purpose	Key Contents	Visibility
		(complex recursive deletion with borrow/merge logic)	

Architecture Decision: Modular Separation of Node Utilities and Operations

- **Context:** The B-tree algorithms require both low-level node manipulations (e.g., inserting a key into an array) and high-level recursive tree traversal logic. We needed to decide how to organize these related but distinct responsibilities.
- **Options Considered:**
  1. **Monolithic file:** Place all functions in one large `btree.c` file.
  2. **Separation by abstraction level:** Split node utilities (`node.c`) from tree operations (`operations.c`), with a public API layer (`btree.c`).
  3. **Separation by operation:** Have separate files for `search.c` , `insert.c` , `delete.c` .
- **Decision:** We chose option 2 (separation by abstraction level).
- **Rationale:** The node utilities (creating, destroying, searching within a node, splitting) are independent, reusable building blocks used by multiple operations. Separating them makes the code more modular and testable. It also mirrors the conceptual separation between the `Node` and `Operations` components. Option 1 leads to an unmanageably large file, while option 3 creates unnecessary fragmentation and duplication (e.g., both insert and delete need node-splitting logic).
- **Consequences:** Build times may be slightly longer due to more compilation units, but incremental compilation is improved. Developers must understand the dependency graph: `operations.c` depends on `node.c` , and `btree.c` depends on both.

This file structure provides a clear roadmap for implementation, aligning with the project milestones:

- **Milestone 1** focuses on `node.c` and the data structures in `btree.h` .
- **Milestone 2** focuses on `search_recursive` in `operations.c` and its exposure via `btree_search` in `btree.c` .
- **Milestone 3** focuses on `insert_non_full` and `node_split_child` across `operations.c` and `node.c` .
- **Milestone 4** focuses on `delete_from_subtree` and the borrow/merge helpers in `node.c` .

Implementation Guidance

A. Technology Recommendations Table

For this C implementation, we recommend a straightforward approach suitable for learning. The "Advanced Option" column hints at production-ready extensions.

Component	Simple Option (Learning)	Advanced Option (Production)
Memory Management	Standard `malloc`/`free` with careful manual tracking of all allocations.	Use a custom slab allocator or arena allocator to group node allocations, improving cache locality and reducing fragmentation.
Key Storage	Fixed-size array of `int` within each node. Simple and predictable.	Support generic keys via function pointers (`compare_fn`) and store keys as `void*` or variable-length bytes.
Value Storage	Store `void*` values in an array parallel to `keys`. Optional for Milestones 1-3.	Implement as a true key-value store with support for arbitrary value types and sizes.
Error Handling	Return boolean success/failure from public functions. Use `assert` for internal invariants during debugging.	Rich error codes (`BTREE_ERR_MEMORY`, `BTREE_ERR_KEY_NOT_FOUND`). Pluggable error logging callback.
Persistence	In-memory only for simplicity.	Add a `btree_save`/`btree_load` API that maps nodes to fixed-size disk pages using file I/O and `mmap`.

B. Recommended File/Module Structure (Code Layout)

Here is the minimal starter code to set up the recommended file structure. Create these files with the following initial content.

File: `include/btree.h` (Public API)

```c
#ifndef BTREE_H

#define BTREE_H


#include <stdbool.h>

#include <stddef.h>


// Forward declaration to make BTreeNode an opaque type.

// Users only handle pointers; internal structure is hidden.

typedef struct BTreeNode BTreeNode;


typedef struct {

    BTreeNode* root;

    int t;                   // Minimum degree

    size_t key_count;        // Total keys in the tree

} BTree;


typedef struct {

    bool found;

    void* value;

    int comparisons;         // For educational insight

} SearchResult;


// Tree lifecycle

BTree* btree_create(int min_degree);

void btree_destroy(BTree* tree);


// Core operations

SearchResult btree_search(BTree* tree, int key);

bool btree_insert(BTree* tree, int key, void* value);

bool btree_delete(BTree* tree, int key);
```

```
// Inspection
size_t btree_size(BTree* tree);
int btree_height(BTree* tree);
bool btree_validate(BTree* tree);
void btree_print(BTree* tree);

#endif // BTREE_H
```

File: `src/btree_private.h` (Private Implementation Details)

```c
#ifndef BTREE_PRIVATE_H

#define BTREE_PRIVATE_H


#include "btree.h"


// Default minimum degree if none specified

#define DEFAULT_MIN_DEGREE 3


// Internal node structure (visible only within src/)

struct BTreeNode {

    int* keys;

    void** values;           // Parallel array to keys

    BTreeNode** children;    // Array of child pointers

    int num_keys;            // Current number of keys

    bool is_leaf;

    // Note: No 't' stored per node; it's passed as parameter where needed.

};


// Node utilities (implemented in node.c)

BTreeNode* node_create(int t, bool is_leaf);

void node_destroy(BTreeNode* node);

int node_find_key_index(BTreeNode* node, int key, int* comparisons);

bool node_is_full(BTreeNode* node, int t);

bool node_is_underfull(BTreeNode* node, int t);


// Core operation helpers (implemented in operations.c)

SearchResult search_recursive(BTreeNode* node, int key);

bool insert_non_full(BTreeNode* node, int key, void* value, int t);

bool delete_from_subtree(BTreeNode* node, int key, int t);
```

```
#endif // BTREE_PRIVATE_H
```

File: `src/btree.c` (Public API Implementation)

```c
#include "btree_private.h"

#include <stdlib.h>

#include <stdio.h>

#include <assert.h>


BTree* btree_create(int min_degree) {

    // TODO 1: Validate min_degree >= 2 (or use DEFAULT_MIN_DEGREE if invalid)

    // TODO 2: Allocate memory for BTree struct

    // TODO 3: Initialize fields: t = min_degree, key_count = 0

    // TODO 4: Create an empty root node (leaf, with node_create)

    // TODO 5: Set tree->root to the new node

    // TODO 6: Return the tree pointer

    return NULL;

}


void btree_destroy(BTree* tree) {

    // TODO 1: If tree is NULL, return early

    // TODO 2: Call node_destroy on tree->root (recursively frees all nodes)

    // TODO 3: Free the tree struct itself

}


SearchResult btree_search(BTree* tree, int key) {

    SearchResult result = {false, NULL, 0};

    // TODO 1: If tree->root is NULL, return empty result

    // TODO 2: Call search_recursive(tree->root, key) and store its result

    // TODO 3: Return the result

    return result;

}
```

```
bool btree_insert(BTree* tree, int key, void* value) {
    // TODO 1: If tree or tree->root is NULL, return false
    // TODO 2: Check if key already exists? (Optional, but can prevent duplicates)
    // TODO 3: If root is full, handle root split:
    //   - Create new root node (internal)
    //   - Make old root the first child of new root
    //   - Call node_split_child on new root to split the old root
    //   - Update tree->root pointer
    // TODO 4: Call insert_non_full(tree->root, key, value, tree->t)
    // TODO 5: If insert successful, increment tree->key_count
    // TODO 6: Return success status
    return false;
}

bool btree_delete(BTree* tree, int key) {
    // TODO 1: If tree or tree->root is NULL, return false
    // TODO 2: Call delete_from_subtree(tree->root, key, tree->t)
    // TODO 3: If deletion successful:
    //   - Decrement tree->key_count
    //   - If root now has 0 keys and is not a leaf:
    //       * Set tree->root to its only child (root->children[0])
    //       * Free the old root
    // TODO 4: Return success status
    return false;
}

size_t btree_size(BTree* tree) {
    // TODO: Return tree->key_count
```

```c
    return 0;

}


int btree_height(BTree* tree) {

    // TODO 1: Start height = 0, current = tree->root

    // TODO 2: While current != NULL, increment height and move to first child (if not
leaf)

    // TODO 3: Return height

    return 0;

}


bool btree_validate(BTree* tree) {

    // TODO: (For Milestone 4) Implement recursive invariant checking

    //   - Check root key count: 0 <= num_keys < 2t (except root can have < t-1)

    //   - For internal nodes: num_keys + 1 == number of non-NULL children

    //   - Keys sorted within each node

    //   - All leaves at same depth

    //   - Child pointers' keys respect B-tree ordering

    return false;

}


void btree_print(BTree* tree) {

    // TODO: (Debug helper) Implement recursive printing with indentation

}
```

File: `src/node.c` **(Node Utilities - Starter Code)**

C

```c
#include "btree_private.h"

#include <stdlib.h>

#include <stdio.h>


BTreeNode* node_create(int t, bool is_leaf) {

    // TODO 1: Allocate memory for BTreeNode struct

    // TODO 2: Allocate arrays: keys (size 2*t-1), children (size 2*t), values (size 2*t-1)

    // TODO 3: Initialize fields: num_keys = 0, is_leaf = is_leaf

    // TODO 4: Initialize children array to NULL (important!)

    // TODO 5: Return the new node

    return NULL;

}


void node_destroy(BTreeNode* node) {

    // TODO 1: If node is NULL, return

    // TODO 2: If not leaf, recursively destroy all non-NULL children

    // TODO 3: Free the keys, values, children arrays

    // TODO 4: Free the node struct itself

}


int node_find_key_index(BTreeNode* node, int key, int* comparisons) {

    // TODO 1: Initialize left = 0, right = node->num_keys - 1

    // TODO 2: While left <= right, perform binary search:

    //   - mid = (left + right) / 2

    //   - Increment *comparisons

    //   - If node->keys[mid] == key, return mid (exact match)

    //   - If node->keys[mid] < key, left = mid + 1

    //   - Else right = mid - 1
```

```
    // TODO 3: Return left (the index where key should be inserted, or child index to
follow)

    return 0;

}


bool node_is_full(BTreeNode* node, int t) {

    // TODO: Return true if node->num_keys == 2*t - 1

    return false;

}


bool node_is_underfull(BTreeNode* node, int t) {

    // TODO: Return true if node->num_keys < t - 1 (except root case handled elsewhere)

    return false;

}
```

File: `src/operations.c` (Core Algorithms - Placeholder)

```c
#include "btree_private.h"

SearchResult search_recursive(BTreeNode* node, int key) {
    SearchResult result = {false, NULL, 0};
    // TODO: To be implemented in Milestone 2
    return result;
}

bool insert_non_full(BTreeNode* node, int key, void* value, int t) {
    // TODO: To be implemented in Milestone 3
    return false;
}

bool delete_from_subtree(BTreeNode* node, int key, int t) {
    // TODO: To be implemented in Milestone 4
    return false;
}
```

C. Language-Specific Hints (C)

1. **Memory Allocation:** Always check the return value of `malloc`. Use `calloc` for arrays you want zero-initialized (like `children` pointers). Remember to free in reverse order: first the internal arrays, then the struct.
2. **Array Management:** When inserting/deleting keys within a node, you'll need to shift elements manually. Use `memmove` for efficiency when shifting large portions of the arrays.
3. **Recursive Functions:** The recursive algorithms (`search_recursive`, `insert_non_full`, `delete_from_subtree`) will need the minimum degree `t` passed as a parameter at each call, since it's not stored in each node.
4. **Debugging:** Use `assert` liberally during development to catch invariant violations (e.g., `assert(node->num_keys <= 2*t-1)`). Implement the `btree_print` function early to visualize tree structure after operations.
5. **Build System:** A simple `Makefile` with separate compilation for each `.c` file will speed up development. Ensure `src/` files include `btree_private.h`, not `btree.h`, to maintain the abstraction

boundary.

---

4. Data Model

> **Milestone(s):** Milestone 1 (B-tree Node Structure), with foundational relevance to all subsequent operations (Milestones 2-4)

The data model defines the core building blocks and rules that give the B-tree its unique properties. Before diving into formal definitions, let's build an intuitive understanding of how these structures work together.

Core Types and Structures

Think of the B-tree data model as a **well-organized warehouse storage system**. The entire warehouse (`BTree`) has a master floor plan (`root`) and a strict rule about shelf capacity (`minimum degree t`). Each storage shelf (`BTreeNode`) has labeled compartments (`keys`) that either point to other shelves in the warehouse (`children`) for internal nodes, or directly to stored inventory (`values`) for leaf nodes. The warehouse manager ensures every shelf follows exact capacity rules and that labels are always in alphabetical order.

This mental model helps visualize why B-trees excel at disk operations: each shelf corresponds to a **disk page** (typically 4KB), and accessing one shelf (loading a page from disk) lets you examine many labels (keys) at once before deciding which next shelf to visit. The strict capacity rules guarantee that shelves are neither too empty (wasting disk space) nor too full (requiring frequent reorganization), optimizing both storage density and access patterns.

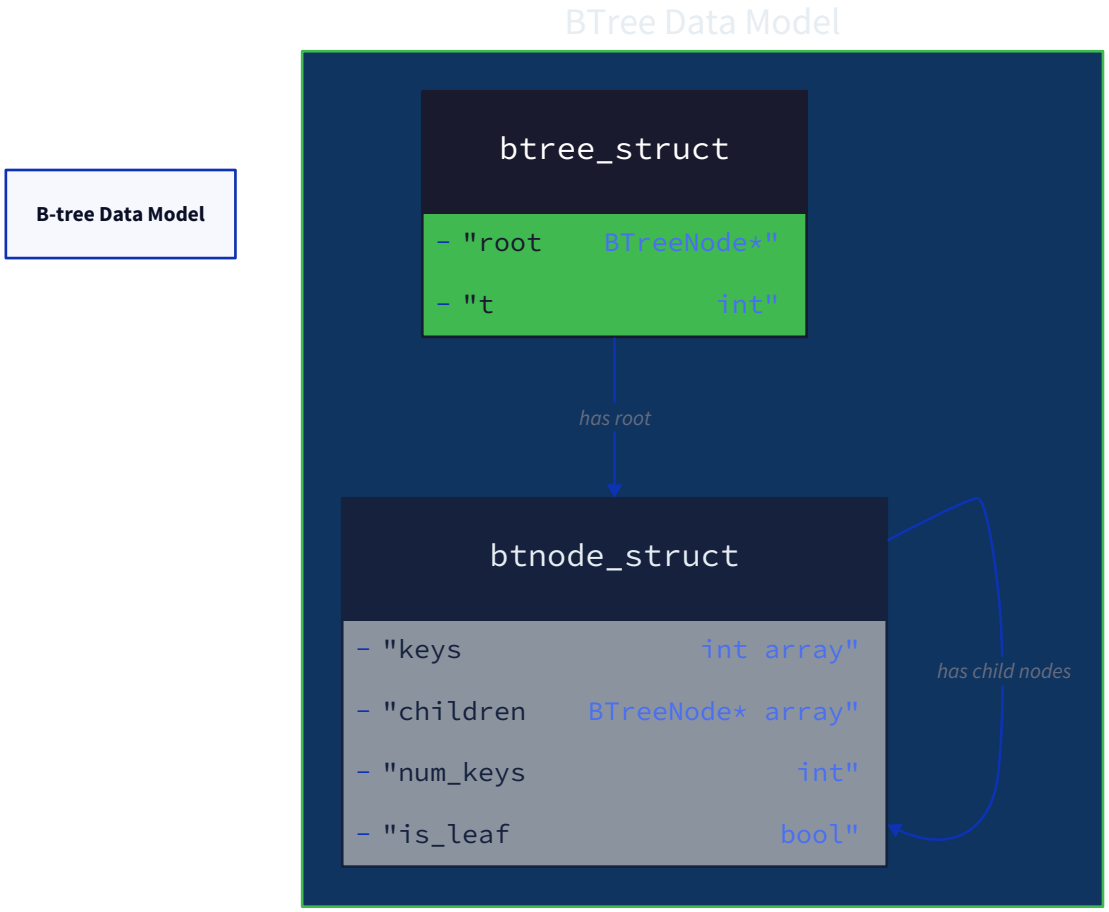
Now let's formalize these concepts into concrete data structures. The following table defines the two primary types that constitute our B-tree implementation:

Type Name	Field Name	Type	Description
`BTree` (The entire warehouse)	`root`	`BTreeNode*`	Pointer to the root node of the tree. The entry point for all operations. When the tree is empty, this is `NULL`.
	`t`	`int`	The **minimum degree** parameter that defines the capacity bounds for all nodes in the tree. This value is fixed when the tree is created and determines: • Maximum keys per node: `2*t - 1` • Minimum keys per node (except root): `t - 1` • Maximum children per internal node: `2*t` • Minimum children per internal node (except root): `t`
	`key_count`	`size_t`	Total number of keys stored in the entire tree. Maintained for O(1) size queries. Updated on every insertion and deletion.
`BTreeNode` (A single shelf)	`keys`	`int*`	Dynamic array of keys stored in this node. The array has capacity for `2*t - 1` integers. Keys within a node are always maintained in **ascending sorted order**. For internal nodes, keys act as separators between child subtrees.
	`values`	`void**`	Optional array of opaque pointers associated with each key. In a key-value B-tree, this stores the value corresponding to each key. For educational purposes focusing only on key structure, this may be `NULL`. If implemented, it parallels the `keys` array (same length).
	`children`	`BTreeNode**`	Dynamic array of child pointers. For internal nodes, this array has size `2*t` (one more than the number of keys). For leaf nodes, this is typically `NULL` or an array of `NULL` pointers (since leaves have no children). Each child pointer `children[i]` points to a subtree where all keys are greater than `keys[i-1]` and less than `keys[i]`.
	`num_keys`	`int`	Current number of keys actually stored in this node. Must satisfy: • For root: `0 ≤ num_keys ≤ 2*t - 1`

Type Name	Field Name	Type	Description
			• For other nodes: `t-1 ≤ num_keys ≤ 2*t - 1` Updated after every insertion, deletion, split, merge, or borrow operation.
	`is_leaf`	`bool`	Boolean flag indicating whether this node is a leaf (true) or an internal node (false). Determines whether `children` pointers are followed during traversal. All leaf nodes are at the same depth (height from root).
`SearchResult` (Search report)	`found`	`bool`	True if the search key was found in the tree, false otherwise.
	`value`	`void*`	If `found` is true and values are stored, contains the opaque pointer associated with the found key. Otherwise `NULL`.
	`comparisons`	`int`	Number of key comparisons performed during the search operation. Useful for performance analysis and debugging.

> **Design Insight:** The separation of `keys` and `children` into parallel arrays (rather than interleaved key-child pairs) simplifies index calculations during operations. The invariant `num_keys + 1 == num_children` for internal nodes means we can think of children as "slots between and around keys": child `i` contains keys less than `keys[i]` (if `i < num_keys`) and greater than `keys[i-1]` (if `i > 0`).

These structures relate to each other as shown in the following diagram:



The diagram illustrates the containment hierarchy: a `BTree` contains a `root` pointer to a `BTreeNode`, which in turn contains `children` pointers to other `BTreeNode` instances. Each node maintains its `keys` in sorted order, with internal nodes having one more child than keys (creating the "between and around" relationship).

B-tree Invariants

The power of B-trees comes from maintaining five **invariants** (mathematical rules) that guarantee balanced height and efficient operations. These invariants are like the warehouse's operating regulations—if any is violated, the entire system becomes inefficient or incorrect.

1. **Root Degree Invariant:**

> The root node may have as few as 1 key (or 0 if the tree is empty) and at most `2*t - 1` keys. Unlike other nodes, the root is exempt from the minimum key requirement (`t-1` keys) to allow the tree to grow and shrink gracefully.

2. **Internal Node Key Bounds Invariant:**

> Every non-root internal node must have between `t-1` and `2*t - 1` keys inclusive. This ensures nodes are sufficiently full to justify the disk read/write cost while leaving room for insertions without immediate splits.

3. **Leaf Node Key Bounds Invariant:**

> Every non-root leaf node must have between `t-1` and `2*t - 1` keys inclusive. Leaves follow the same capacity rules as internal nodes, maintaining uniform node sizes across the tree.

4. **Child Count Invariant:**

> For any internal node with `k` keys, it must have exactly `k + 1` children. This creates the fundamental B-tree structure where each key separates two child subtrees, and there are always one more children than keys.

5. **Key Ordering Invariant:**

> All keys within a node are stored in **strictly ascending order** (no duplicates). Furthermore, for any internal node, the keys in child `i` are all less than `keys[i]`, and the keys in child `i+1` are all greater than `keys[i]`. This recursive ordering property enables efficient binary search at every level.

These invariants work together to guarantee that a B-tree with `n` keys has height at most `O(log_t n)`. For typical disk-based systems with `t` around 100-200, this means even terabytes of data can be accessed with only 3-4 disk reads. The following table shows how operations maintain these invariants:

Operation	Which Invariants Might Be Threatened	How Operation Preserves Invariants
Insert	Internal/Leaf Key Bounds (node becomes overfull with `2*t` keys)	**Proactive splitting** during descent ensures no node ever receives a key when already full. Splitting moves the median key up to the parent, creating two nodes each with `t-1` keys.
Delete	Internal/Leaf Key Bounds (node becomes underfull with `t-2` keys)	**Borrowing** from a sibling (if possible) or **merging** with a sibling (otherwise) ensures every non-root node maintains at least `t-1` keys.
Root Split	Root Degree (root becomes overfull)	Create a new root with one key (the median from the old root) and two children (the split halves of the old root). The new root satisfies the root's relaxed bounds (1 key).
Root Merge	Root Degree (root becomes empty after merge)	When the root has 0 keys after a merge from its children, make the merged child the new root. Tree height decreases by 1.

Architecture Decision Record: Node Capacity Representation

Context: We need to represent a B-tree node's capacity constraints in code. The minimum degree `t` defines all bounds, but we must decide how to store and validate these bounds.

Options Considered:

1. **Store only `t` in `BTree`:** Each node calculates its bounds using the tree's `t` value.
2. **Store `max_keys` and `min_keys` in each node:** Precompute `2*t-1` and `t-1` when creating a node.
3. **Store `capacity` in each node:** Allocate arrays of size `2*t-1` for keys and `2*t` for children, and track `num_keys`.

Decision: Option 1 (store only `t` in `BTree`) with dynamic array sizes based on `t`.

Rationale:

- **Simplicity:** A single parameter `t` controls all bounds, reducing state to maintain.
- **Consistency:** All nodes in a tree share the same `t`, so storing it centrally avoids duplication.
- **Flexibility:** If we decide to support variable `t` per node (for advanced variants like B* trees), we can extend later.
- **Memory efficiency:** Not storing redundant bounds in every node saves memory, important for large trees.

Consequences:

- **Positive:** Clean abstraction where `t` is a tree-wide property.
- **Negative:** Functions manipulating nodes need access to `t`, requiring it to be passed as a parameter.
- **Mitigation:** We'll pass `t` to all node operations as an explicit parameter, making dependencies clear.

Option	Pros	Cons	Chosen?
Store only `t` in `BTree`	Simple, memory efficient, centralized control	Need to pass `t` to node functions	✅ Yes
Store bounds in each node	Node functions self-contained	Memory overhead, duplicate data	
Store capacity in each node	Clear array size tracking	Doesn't capture minimum bound, still needs `t` for splitting logic	

Common Pitfalls: Data Model Misunderstandings

⚠️ Pitfall: Off-by-one in child pointer counts

- **Description:** Allocating `2*t` child pointers for a leaf node (which needs none) or allocating `2*t-1` child pointers for an internal node (which needs `2*t`).
- **Why it's wrong:** Wastes memory for leaves, and causes buffer overflows for internal nodes when splitting (which requires temporarily storing `2*t` children before redistribution).
- **Fix:** Only allocate child pointer array for internal nodes. For leaves, set `children = NULL`. Check `is_leaf` flag before accessing children.

⚠️ Pitfall: Forgetting to update `num_keys` after operations

- **Description:** Modifying the `keys` array (inserting, deleting, splitting) without updating the `num_keys` counter.
- **Why it's wrong:** Binary search uses `num_keys` as the upper bound. If `num_keys` is too small, searches miss valid keys; if too large, searches access uninitialized memory.
- **Fix:** Treat `num_keys` as the sacred source of truth. Every function that modifies `keys` must update `num_keys` as its final step. Consider writing an invariant checker that validates `num_keys` equals actual occupied slots.

⚠️ Pitfall: Assuming all nodes have values

- **Description:** Accessing `values[i]` without checking if the tree stores values or if the implementation is key-only.
- **Why it's wrong:** In a key-only B-tree (common in set implementations), `values` may be `NULL`. Even in key-value trees, internal nodes often don't store values (only keys for routing).
- **Fix:** Design clearly: either implement full key-value storage (with `values` array parallel to `keys`), or omit values entirely for simplicity. Document the choice.

Implementation Guidance

Now let's bridge from design to implementation. For our C implementation, we'll create the foundational data structures with careful memory management.

Technology Recommendations Table

Component	Simple Option (Educational Focus)	Advanced Option (Production Ready)
Memory Allocation	Standard `malloc/free` with manual tracking	Memory pool allocator with slab allocation for nodes
Value Storage	Opaque `void*` pointers allowing any data type	Typed unions with serialization for disk persistence
Capacity Management	Dynamic arrays reallocated on split/merge	Fixed-size pages matching disk block size (e.g., 4096 bytes)
Debug Infrastructure	Simple print functions and invariant checker	Comprehensive logging, visualization, and property testing

Recommended File/Module Structure

For a clean C implementation, organize files as follows:

```
btree-project/
├── include/
│   └── btree.h          # Public interface declarations
├── src/
│   ├── btree.c          # BTree struct functions (create, destroy, public ops)
│   ├── node.c           # BTreeNode functions (create, destroy, helpers)
│   ├── search.c         # Search algorithm implementation
│   ├── insert.c         # Insertion with splitting
│   ├── delete.c         # Deletion with rebalancing
│   └── debug.c          # Validation, printing, debugging utilities
└── tests/
    ├── test_btree.c     # Comprehensive test suite
    └── test_utils.c     # Testing utilities
```

This separation follows the single responsibility principle: `btree.c` manages the tree-level operations, `node.c` handles node-level manipulations, and operation-specific files contain the algorithmic logic.

Infrastructure Starter Code

Here's complete, working code for the foundational data structures and basic utilities. Learners should copy this as their starting point.

File: `include/btree.h`

```c
#ifndef BTREE_H

#define BTREE_H


#include <stdbool.h>

#include <stddef.h>


// Forward declaration for opaque pointer pattern

typedef struct BTreeNode BTreeNode;


// Search result structure

typedef struct {

    bool found;

    void* value;

    int comparisons;

} SearchResult;


// Main B-tree structure

typedef struct {

    BTreeNode* root;

    int t;  // Minimum degree

    size_t key_count;

} BTree;


// Public API

BTree* btree_create(int min_degree);

void btree_destroy(BTree* tree);

SearchResult btree_search(BTree* tree, int key);

bool btree_insert(BTree* tree, int key, void* value);

bool btree_delete(BTree* tree, int key);
```

```c
size_t btree_size(BTree* tree);

int btree_height(BTree* tree);

bool btree_validate(BTree* tree);

void btree_print(BTree* tree);


// Default minimum degree if not specified

#define DEFAULT_MIN_DEGREE 3


#endif // BTREE_H
```

File: `src/node.c` - Foundation implementation

```c
#include "btree.h"

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>


// B-tree node structure definition

struct BTreeNode {

    int* keys;           // Array of keys

    void** values;       // Optional array of values (parallel to keys)

    BTreeNode** children; // Array of child pointers

    int num_keys;        // Current number of keys

    bool is_leaf;        // Is this a leaf node?

};


// Helper function: Create a new empty node

BTreeNode* node_create(int t, bool is_leaf) {

    BTreeNode* node = (BTreeNode*)malloc(sizeof(BTreeNode));

    if (!node) return NULL;



    // Allocate arrays based on B-tree properties

    // Maximum keys: 2*t - 1

    node->keys = (int*)malloc((2*t - 1) * sizeof(int));

    // Optional: allocate values array if implementing key-value store

    node->values = NULL; // Or malloc((2*t - 1) * sizeof(void*)) for key-value



    // For internal nodes, allocate child pointers (2*t capacity)

    // For leaves, children remain NULL

    if (!is_leaf) {
```

```
        node->children = (BTreeNode**)malloc(2*t * sizeof(BTreeNode*));
        // Initialize all children to NULL
        for (int i = 0; i < 2*t; i++) {
            node->children[i] = NULL;
        }
    } else {
        node->children = NULL;
    }

    node->num_keys = 0;
    node->is_leaf = is_leaf;

    return node;
}

// Helper function: Destroy a node and all its descendants recursively
void node_destroy(BTreeNode* node) {
    if (!node) return;

    // Recursively destroy all children
    if (!node->is_leaf) {
        for (int i = 0; i <= node->num_keys; i++) {
            if (node->children[i]) {
                node_destroy(node->children[i]);
            }
        }
        free(node->children);
```

```
    }


    free(node->keys);
    if (node->values) {
        free(node->values);
    }
    free(node);
}

// Helper function: Check if a node is full (has 2*t - 1 keys)
bool node_is_full(BTreeNode* node, int t) {
    return node->num_keys == (2*t - 1);
}

// Helper function: Check if a node is underfull (has fewer than t-1 keys)
bool node_is_underfull(BTreeNode* node, int t) {
    return node->num_keys < (t - 1);
}
```

File: `src/btree.c` - Tree-level infrastructure

```c
#include "btree.h"

#include <stdlib.h>


// Create a new empty B-tree with given minimum degree

BTree* btree_create(int min_degree) {

    if (min_degree < 2) {

        // B-tree requires t >= 2 for proper splitting

        min_degree = DEFAULT_MIN_DEGREE;

    }



    BTree* tree = (BTree*)malloc(sizeof(BTree));

    if (!tree) return NULL;



    tree->root = NULL;

    tree->t = min_degree;

    tree->key_count = 0;



    return tree;

}


// Destroy the entire B-tree and free all memory

void btree_destroy(BTree* tree) {

    if (!tree) return;



    if (tree->root) {

        node_destroy(tree->root);

    }
```

```
    free(tree);
}

// Get the number of keys in the tree (O(1) version)
size_t btree_size(BTree* tree) {
    return tree ? tree->key_count : 0;
}

// Simple placeholder implementations - to be completed in milestones
SearchResult btree_search(BTree* tree, int key) {
    SearchResult result = {false, NULL, 0};
    // TODO Milestone 2: Implement search
    (void)tree; (void)key; // Suppress unused parameter warnings
    return result;
}

bool btree_insert(BTree* tree, int key, void* value) {
    // TODO Milestone 3: Implement insertion
    (void)tree; (void)key; (void)value;
    return false;
}

bool btree_delete(BTree* tree, int key) {
    // TODO Milestone 4: Implement deletion
    (void)tree; (void)key;
    return false;
}

int btree_height(BTree* tree) {
```

```
    // TODO: Implement height calculation
    (void)tree;
    return 0;
}

bool btree_validate(BTree* tree) {
    // TODO: Implement invariant validation
    (void)tree;
    return true;
}

void btree_print(BTree* tree) {
    // TODO: Implement tree printing
    (void)tree;
}
```

Core Logic Skeleton Code

For the core operations that learners will implement themselves (starting with Milestone 1), here are skeleton functions with detailed TODOs:

File: `src/node.c` - Continued (core logic skeletons)

```c
// Helper function: Find the index where a key should be in this node
// Uses binary search for O(log t) performance within a node
// Returns:
//   - If key is found: index of the key (0 ≤ index < num_keys)
//   - If key is not found: -(insert_position + 1) where insert_position is
//     where the key would be inserted (0 ≤ insert_position ≤ num_keys)
// Also increments *comparisons if pointer is not NULL
int node_find_key_index(BTreeNode* node, int key, int* comparisons) {
    // TODO 1: Initialize binary search bounds: left = 0, right = num_keys - 1
    // TODO 2: While left <= right:
    //   TODO 2a: Calculate mid = left + (right - left) / 2
    //   TODO 2b: Compare key with node->keys[mid]
    //   TODO 2c: Increment *comparisons if comparisons pointer is not NULL
    //   TODO 2d: If key == node->keys[mid], return mid (found)
    //   TODO 2e: If key < node->keys[mid], search left half (adjust right)
    //   TODO 2f: If key > node->keys[mid], search right half (adjust left)
    // TODO 3: If key not found, return -(left + 1) where left is insert position
    // Hint: This follows standard binary search with careful handling of
    // the "not found" case using negative encoding
    return 0;
}
```

Language-Specific Hints for C

- **Memory Management:** Always check `malloc` return values for `NULL`. In production code, handle allocation failures gracefully. For educational purposes, you might use `assert` or simply exit with an error message.
- **Array Bounds:** C doesn't check array bounds. Be meticulous with indices, especially when accessing `children[i]` where `i` can be `num_keys` (valid for the rightmost child).

- **Integer Division:** When calculating the median during splitting (`t - 1`), remember C integer division truncates. With `t=3`, `(2*t - 1) = 5` keys, median index is `2` (0-based), which equals `t - 1`.
- **const Correctness:** For search functions that don't modify nodes, consider using `const BTreeNode*` to indicate read-only access and prevent accidental modifications.
- **Header Guards:** Always use `#ifndef/#define` guards in header files to prevent multiple inclusion, as shown in `btree.h`.

Milestone Checkpoint: Data Structure Validation

After implementing the data structures from Milestone 1 (before implementing search/insert/delete), verify your foundation:

1. **Compilation Test:**

```bash
gcc -c src/btree.c src/node.c -I./include
```

Should compile without errors or warnings.

2. **Memory Leak Test:**

```c
// In a simple test program:

#include "btree.h"

int main() {

    BTree* tree = btree_create(3);

    // Should create tree with t=3, root=NULL, key_count=0



    // Create a sample node

    BTreeNode* node = node_create(3, true);

    // Should allocate keys array of size 5, children=NULL, is_leaf=true



    // Clean up

    node_destroy(node);

    btree_destroy(tree);

    return 0;

}
```

Run with Valgrind to ensure no memory leaks:

```bash
valgrind --leak-check=full ./test_program
```

3. **Invariant Awareness:** Manually verify you understand the capacity rules:
    - For `t=3`, maximum keys = `2*3 - 1 = 5`, minimum keys (non-root) = `3 - 1 = 2`
    - For `t=3`, internal nodes have up to `2*3 = 6` children, at least `3` children (except root)
    - Root can have 1-5 keys (or 0 if tree empty)

If you encounter segmentation faults, likely causes are:

- **Accessing `children` array on a leaf node** → Check `is_leaf` before accessing `children`
- **Array index out of bounds** → Verify `num_keys` is correctly maintained and used as array bound
- **Double-free** → Ensure `node_destroy` handles `NULL` children pointers and `NULL` values array

With these data structures solidly implemented, you have the foundation for implementing B-tree operations in subsequent milestones.

5. Component Design: B-tree Node

> **Milestone(s):** Milestone 1 (B-tree Node Structure)

The B-tree node is the fundamental storage unit of the entire data structure—the "atom" from which the entire tree is built. Every operation (search, insert, delete) begins and ends with manipulations of nodes. This component defines how keys and child pointers are organized within a single node, establishes the boundary conditions for node capacity, and differentiates between leaf nodes (which store actual data) and internal nodes (which provide navigation). Getting this foundation right is critical because all subsequent operations build upon these basic building blocks.

Mental Model: A Bookshelf with Dividers

Imagine a bookshelf in a library that can hold a fixed maximum number of books. This bookshelf has dividers (vertical partitions) that separate sections, and each section contains either:

1. **Direct references to books** (if this is a leaf shelf)
2. **References to other shelves** (if this is an index shelf)

Keys as dividers: Each divider has a label (the "key") that indicates the range of books/shelves in the section to its left. For example, a divider labeled "M" means all books/shelves to the left have titles alphabetically before "M".

Child pointers as sections: The spaces between dividers (and the ends) are "slots" that hold either book references (at leaves) or pointers to other shelves (in internal nodes). A node with `k` keys has exactly `k+1` child slots.

Capacity constraints: Each shelf has a strict capacity: it can hold between `t-1` and `2t-1` dividers (except the topmost shelf, which can have as few as 1 divider). When a shelf becomes too full (exceeds `2t-1` dividers), it must be split. When a shelf becomes too empty (falls below `t-1` dividers), it must borrow from a neighbor or merge.

Leaf vs. internal distinction: Leaf shelves contain actual books (data values). Index shelves contain only dividers and references to other shelves—they never store books directly. The bottommost shelves in the library are always leaves.

This mental model helps visualize why B-trees are so disk-efficient: each "shelf" corresponds to one disk page (typically 4KB), and we maximize the number of keys per page to minimize the number of disk seeks during navigation. The strict capacity bounds ensure the tree remains balanced, guaranteeing predictable performance.

Interface and Operations

The node component provides the following core functions for creating, manipulating, and querying individual B-tree nodes. These functions are used internally by the B-tree operations (search, insert, delete) but are not exposed directly to external callers—they form the private API of the B-tree implementation.

Method Name	Parameters	Returns	Description
`node_create`	`t` (minimum degree, `int`), `is_leaf` (`bool`)	`BTreeNode*`	Allocates and initializes a new empty node. The node starts with 0 keys, and its child pointers are initialized to `NULL`. The `is_leaf` flag determines whether this node will store values directly (leaf) or child pointers (internal).
`node_destroy`	`node` (`BTreeNode*`)	`void`	Recursively deallocates a node and all its descendants. For leaf nodes, this may also free associated values if the B-tree owns them.
`node_find_key_index`	`node` (`BTreeNode*`), `key` (`int`), `comparisons` (`int*`)	`int`	Performs binary search within the node's sorted keys array to locate either (1) the exact position of `key` if present, or (2) the index of the child subtree where `key` would be located. The `comparisons` pointer allows counting search operations for performance analysis.
`node_is_full`	`node` (`BTreeNode*`), `t` (`int`)	`bool`	Checks whether the node has reached its maximum capacity (`2t-1` keys). Used during insertion to determine if a split is needed.
`node_is_underfull`	`node` (`BTreeNode*`), `t` (`int`)	`bool`	Checks whether the node has fallen below its minimum allowed capacity (`t-1` keys, except for the root). Used during deletion to trigger rebalancing operations.
`node_insert_key`	`node` (`BTreeNode*`), `index` (`int`), `key` (`int`), `value` (`void*`)	`void`	Inserts a new key (and optional associated value) at position `index` within the node's keys array, shifting existing keys and child pointers right as needed.

Method Name	Parameters	Returns	Description
			Assumes the node is not full and maintains sorted order.
`node_remove_key`	`node` (`BTreeNode*`), `index` (`int`)	`void`	Removes the key at position `index`, shifting subsequent keys and child pointers left to fill the gap. For leaf nodes, may also free the associated value.
`node_split_child`	`parent` (`BTreeNode*`), `child_index` (`int`), `t` (`int`)	`void`	Splits a full child node at `child_index` into two nodes, promoting the median key to the parent. The left child retains the first `t-1` keys, the right child gets the last `t-1` keys, and the median (t-th) key moves up to the parent at position `child_index`.
`node_borrow_from_left`	`node` (`BTreeNode*`), `parent` (`BTreeNode*`), `child_index` (`int`)	`void`	Borrows a key from the left sibling through the parent to fix an underflow condition. The parent's key at `child_index-1` moves down into `node`, and the left sibling's last key moves up to replace it. Corresponding child pointers are also transferred.
`node_borrow_from_right`	`node` (`BTreeNode*`), `parent` (`BTreeNode*`), `child_index` (`int`)	`void`	Borrows a key from the right sibling through the parent. The parent's key at `child_index` moves down into `node`, and the right sibling's first key moves up to replace it. Corresponding child pointers are transferred.
`node_merge_children`	`parent` (`BTreeNode*`), `child_index` (`int`), `t` (`int`)	`void`	Merges the child at `child_index` with its right sibling, using the parent's key at `child_index` as an additional key. All keys from both children and the parent key are combined into a single node, and the empty sibling is destroyed.

ADR: Node Memory Layout

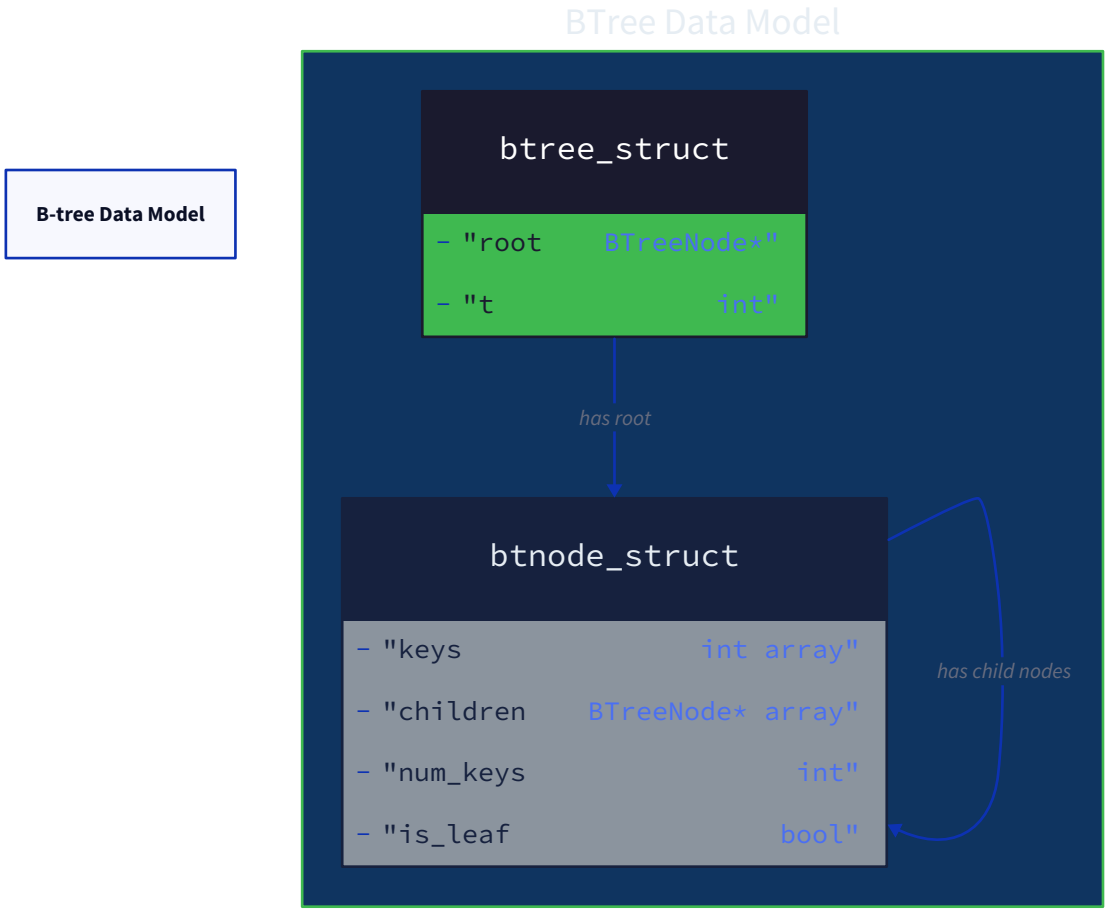
Decision: Separate Arrays for Keys and Children

- **Context**: We need to store two types of data in each B-tree node: (1) an array of integer keys, and (2) an array of child pointers (for internal nodes) or value pointers (for leaf nodes). The memory layout must support efficient binary search within keys, straightforward insertion/deletion with shifting, and clear distinction between leaf and internal node behavior.
- **Options Considered**:
  1. **Separate arrays**: One contiguous array for keys, another contiguous array for child/value pointers.
  2. **Unified array of structs**: An array where each element is a struct containing both a key and a child/value pointer.
  3. **Hybrid with union**: Separate key array, but child/value pointers stored in a union type that can represent either children or values based on leaf status.
- **Decision**: Use separate arrays for keys and child pointers, with leaf nodes storing values in the child pointer array (reinterpreted as `void*` values).
- **Rationale**:
  - **Binary search efficiency**: Searching within a contiguous array of keys (integers) is cache-friendly and allows standard binary search implementation without pointer dereferencing overhead.
  - **Insertion/deletion simplicity**: When inserting a new key, we shift only the keys array; child pointers shift independently at the same index. This maintains the invariant that child `i` contains keys less than key `i`, and child `i+1` contains keys greater than key `i`.
  - **Memory alignment**: Integer keys are typically 4 bytes, while pointers are 8 bytes on 64-bit systems. Keeping them separate avoids padding issues in a unified struct.
  - **Clear semantics**: The `is_leaf` flag determines how to interpret the pointer array: as `BTreeNode*` children for internal nodes, or as `void*` values for leaf nodes. This avoids the complexity of unions while maintaining type safety through careful casting.
  - **Educational clarity**: The separate-array approach matches most textbook descriptions and visualizations of B-trees, making the code easier to understand for learners.
- **Consequences**:
  - **Positive**: Simple, predictable memory access patterns; easy to implement binary search; clear correspondence with theoretical B-tree diagrams.
  - **Negative**: Requires maintaining two arrays with coordinated indices; leaf nodes waste the child pointer capacity (they allocate space for `2t` pointers but only use them for values). This waste is acceptable for an in-memory educational implementation.

Option	Pros	Cons	Chosen?
Separate arrays	- Efficient binary search on keys - Clear index correspondence between keys and children - Matches textbook descriptions - Simple insertion shifting logic	- Leaf nodes allocate unused child pointer capacity - Requires coordinated management of two arrays	**Yes**
Unified array of structs	- Keys and pointers stored together (potentially better locality) - Single array to manage	- Binary search must access keys through structs (slightly more overhead) - Insertion requires shifting larger structs - Less intuitive mapping to B-tree theory	No
Hybrid with union	- Explicit type differentiation via union tag - No wasted space for leaf nodes (values stored directly)	- More complex type handling - Requires careful casting and union access - Additional branching based on leaf status	No

The chosen layout directly corresponds to the `BTreeNode` struct definition from Section 4:

- `keys` : `int*` – Dynamic array of `2t-1` integers
- `children` : `BTreeNode**` – Dynamic array of `2t` pointers (interpreted as `void**` for leaf values)
- `num_keys` : `int` – Current number of keys (0 to `2t-1`)
- `is_leaf` : `bool` – Determines interpretation of `children` array



Common Pitfalls: Node Capacity

B-tree nodes have strict capacity bounds that must be maintained at all times. These invariants are easy to violate accidentally, especially when implementing split and merge operations. Below are the most common mistakes learners make when implementing the node component.

⚠️ Pitfall: Off-by-one errors in key counts

- **Description**: Incorrectly calculating the maximum (`2t-1`) or minimum (`t-1`) key counts, or mishandling the root's special case (can have as few as 1 key). For example, checking `if (node->num_keys == 2*t)` instead of `if (node->num_keys == 2*t - 1)` for fullness.
- **Why it's wrong**: Violates B-tree invariants, leading to potential tree corruption. A node with `2t` keys would overflow its array bounds (which is sized for `2t-1` keys). A node with `t-2` keys (when not the root) would be underfull, causing search paths to become imbalanced.
- **How to fix**: Define helper functions `node_is_full()` and `node_is_underfull()` that encapsulate these calculations. Use these functions consistently throughout the codebase instead of repeating the

logic. Remember the root exception: `node_is_underfull()` should return `false` for the root regardless of key count.

⚠️ Pitfall: Forgetting to initialize child pointers to NULL

- **Description**: After allocating the `children` array (size `2t`), failing to set all elements to `NULL`. This is especially critical for leaf nodes, where all child pointers should be `NULL`, and for internal nodes where unused slots beyond `num_keys+1` should be `NULL`.
- **Why it's wrong**: Uninitialized pointers cause undefined behavior when accessed (e.g., during recursive destruction or when checking if a child exists). This often manifests as segmentation faults during tree traversal or memory leaks during cleanup.
- **How to fix**: In `node_create()`, after allocating the `children` array, loop through all `2t` slots and explicitly set them to `NULL`. Similarly, when increasing `num_keys` (e.g., during insertion), ensure new child pointers are initialized to `NULL`.

⚠️ Pitfall: Mis-handling the leaf flag during node splitting

- **Description**: When splitting a child node, incorrectly setting the `is_leaf` flag on the new right sibling. For example, copying the flag from the original child without considering that leaves should remain leaves and internal nodes should remain internal.
- **Why it's wrong**: If an internal node's child is incorrectly marked as leaf, subsequent insertions will fail to recurse deeper, causing keys to be inserted at the wrong level. If a leaf node's child is incorrectly marked as internal, attempts to traverse further will fail (since child pointers are `NULL`).
- **How to fix**: When creating the right sibling during a split, explicitly copy the `is_leaf` flag from the original child being split. Both the left half (original child, now truncated) and right half (new sibling) should have the same leaf status as the original node before splitting.

⚠️ Pitfall: Incorrect child pointer indexing after key insertion/removal

- **Description**: When inserting or removing a key at index `i`, forgetting to shift child pointers at the same index. The invariant is that child `i` contains keys less than key `i`, and child `i+1` contains keys greater than key `i`. After inserting a new key at index `i`, child pointers from `i+1` onward must shift right by one position.
- **Why it's wrong**: Breaks the parent-child relationship, causing subtrees to become disconnected or assigned to wrong key ranges. This corrupts the tree structure and leads to missing keys during search.
- **How to fix**: Implement helper functions `node_insert_key()` and `node_remove_key()` that handle both key and child pointer shifting together. Always remember that a node with `k` keys has `k+1` child pointers (even for leaves, though they're unused).

⚠️ Pitfall: Not updating num_keys after mutations

- **Description**: Forgetting to increment `num_keys` after inserting a key or decrement it after removing a key, while still updating the keys array.

- **Why it's wrong**: The `num_keys` field becomes out of sync with the actual data in the arrays. This causes binary search to examine uninitialized key values and can lead to array bounds violations when accessing children.
- **How to fix**: Update `num_keys` as the last step in any mutation function, after all array shifts are complete. Better yet, encapsulate all mutations in small, well-tested functions that guarantee consistency.

Implementation Guidance

> **Technology Note**: This implementation uses plain C for direct memory control, which is appropriate for understanding the low-level details of B-tree node layout. The `stdlib.h` provides dynamic memory allocation, and `stdbool.h` gives us boolean types.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Memory Allocation	`malloc()` / `free()` with explicit NULL checks	Memory pool allocator for fixed-size node allocations
Array Management	Separate `keys` and `children` arrays with manual shifting	Generic array utility functions with `memmove()`
Debugging	`assert()` statements for invariants	Custom invariant checker with detailed error messages

B. Recommended File/Module Structure

```
btree-project/
├── include/
│   └── btree.h          ← Public interface (BTree, SearchResult, etc.)
├── src/
│   ├── btree.c          ← Main B-tree operations (search, insert, delete)
│   ├── node.c           ← Node-specific functions (this component)
│   └── btree_utils.c    ← Utility functions (validation, printing)
└── tests/
    ├── test_node.c      ← Unit tests for node functions
    └── test_btree.c     ← Integration tests
```

C. Infrastructure Starter Code

File: `include/btree.h` (partial - node-related declarations)

```c
#ifndef BTREE_H

#define BTREE_H


#include <stdbool.h>

#include <stddef.h>


// Forward declaration for opaque pointer

typedef struct BTreeNode BTreeNode;


// B-tree structure

typedef struct {

    BTreeNode* root;

    int t;  // Minimum degree

    size_t key_count;

} BTree;


// Search result structure

typedef struct {

    bool found;

    void* value;

    int comparisons;

} SearchResult;


// Public interface functions (declarations only)

BTree* btree_create(int min_degree);

void btree_destroy(BTree* tree);

SearchResult btree_search(BTree* tree, int key);

bool btree_insert(BTree* tree, int key, void* value);

bool btree_delete(BTree* tree, int key);
```

```c
size_t btree_size(BTree* tree);

int btree_height(BTree* tree);

bool btree_validate(BTree* tree);

void btree_print(BTree* tree);


// Node functions (internal, but declared here for testing)

#ifdef BTREE_INTERNAL

BTreeNode* node_create(int t, bool is_leaf);

void node_destroy(BTreeNode* node);

int node_find_key_index(BTreeNode* node, int key, int* comparisons);

bool node_is_full(BTreeNode* node, int t);

bool node_is_underfull(BTreeNode* node, int t);

#endif // BTREE_INTERNAL


#endif // BTREE_H
```

File: `src/node.c` (starter infrastructure)

```c
#include "btree.h"
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <string.h>

// B-tree node structure definition (hidden from public header)
struct BTreeNode {
    int* keys;           // Array of keys
    void** values;       // For leaves: array of value pointers
    BTreeNode** children; // For internal nodes: array of child pointers
    int num_keys;        // Current number of keys
    bool is_leaf;        // Leaf or internal node
    // Note: We use values for leaves and children for internal nodes,
    // but they share the same memory slot in the struct.
    // We'll manage this through the is_leaf flag.
};

// Helper function to allocate and initialize arrays
static void node_allocate_arrays(BTreeNode* node, int t) {
    // Keys array: maximum (2t-1) keys
    node->keys = (int*)malloc(sizeof(int) * (2 * t - 1));
    if (!node->keys) {
        perror("Failed to allocate keys array");
        exit(EXIT_FAILURE);
    }

    // Children/values array: maximum 2t children
```

```
    // We allocate as BTreeNode** but will interpret as void** for leaves
    node->children = (BTreeNode**)malloc(sizeof(BTreeNode*) * (2 * t));
    if (!node->children) {
        free(node->keys);
        perror("Failed to allocate children array");
        exit(EXIT_FAILURE);
    }

    // Initialize all child pointers to NULL
    for (int i = 0; i < 2 * t; i++) {
        node->children[i] = NULL;
    }
}

BTreeNode* node_create(int t, bool is_leaf) {
    // Validate minimum degree
    if (t < 2) {
        fprintf(stderr, "Minimum degree t must be at least 2\n");
        return NULL;
    }

    // Allocate node structure
    BTreeNode* node = (BTreeNode*)malloc(sizeof(BTreeNode));
    if (!node) {
        perror("Failed to allocate B-tree node");
        return NULL;
    }
```

```
    // Initialize fields
    node->num_keys = 0;
    node->is_leaf = is_leaf;

    // Allocate arrays
    node_allocate_arrays(node, t);

    return node;
}

void node_destroy(BTreeNode* node) {
    if (!node) return;

    // If internal node, recursively destroy children
    if (!node->is_leaf) {
        for (int i = 0; i <= node->num_keys; i++) {
            if (node->children[i]) {
                node_destroy(node->children[i]);
            }
        }
    } else {
        // For leaf nodes, we might need to free values if they're owned
        // In this basic implementation, we assume caller manages values
        // so we don't free them here
    }
```

```
    // Free arrays
    free(node->keys);
    free(node->children);


    // Free node itself
    free(node);
}
```

D. Core Logic Skeleton Code

File: `src/node.c` (continued - core functions with TODOs)

```c
// Finds the position of key within a node using binary search

// Returns:

// - If key is found: index of the key (0 <= index < node->num_keys)

// - If key is not found: index of the child where key should be (0 <= index <= node->num_keys)

int node_find_key_index(BTreeNode* node, int key, int* comparisons) {

    if (!node) {

        return 0;

    }



    int left = 0;

    int right = node->num_keys - 1;

    int mid = 0;



    // TODO 1: Handle empty node case (num_keys == 0) - return 0



    // TODO 2: Implement binary search loop:

    //   while (left <= right) {

    //     mid = left + (right - left) / 2;

    //     (*comparisons)++;

    //     if (node->keys[mid] == key) {

    //         return mid; // Key found at index mid

    //     } else if (node->keys[mid] < key) {

    //         left = mid + 1;

    //     } else {

    //         right = mid - 1;

    //     }
```

```
    //   }


    // TODO 3: Key not found in this node
    // Return the index of the child where the key would be
    // This should be the value of 'left' after the loop ends
}

bool node_is_full(BTreeNode* node, int t) {
    // TODO: Check if node has reached maximum capacity (2t-1 keys)
    // Remember: node can be NULL (handle gracefully)
}

bool node_is_underfull(BTreeNode* node, int t) {
    // TODO: Check if node has fallen below minimum capacity (t-1 keys)
    // Exception: Root node is allowed to be underfull (return false for root)
    // Hint: This function doesn't know if it's the root, so the caller
    // must handle the root case separately
}

// Inserts a new key into the node at the specified index
// Assumes: node is not full, index is valid (0 <= index <= node->num_keys)
void node_insert_key(BTreeNode* node, int index, int key, void* value) {
    // TODO 1: Shift existing keys from index to num_keys-1 one position right
    // Use memmove or a for loop


    // TODO 2: Shift child pointers from index+1 to num_keys+1 one position right
    // Important: A node with k keys has k+1 child pointers
```

```
    // TODO 3: Insert the new key at keys[index]


    // TODO 4: If leaf node, store value at children[index] (interpreted as void*)
    // If internal node, the new child pointer should already be NULL (initialized)


    // TODO 5: Increment num_keys
}

// Removes the key at the specified index from the node
void node_remove_key(BTreeNode* node, int index) {
    // TODO 1: Shift existing keys from index+1 to num_keys-1 one position left


    // TODO 2: Shift child pointers from index+1 to num_keys one position left


    // TODO 3: Decrement num_keys


    // TODO 4: For leaf nodes, we might want to clear the last child pointer
    // (set children[num_keys] to NULL since we now have one fewer key)
}
```

E. Language-Specific Hints

1. **Memory Management**: Always check `malloc()` return values. In production code, you'd handle allocation failures gracefully; for this educational implementation, we exit with an error for simplicity.
2. **Type Casting**: When dealing with the `children` array in leaf nodes (where it stores `void*` values), you'll need to cast: `(void*)node->children[i]`. Be consistent with these casts to avoid compiler warnings.
3. **Array Shifting**: Use `memmove()` for shifting elements in arrays rather than manual loops—it's safer and often optimized. Example: `memmove(&node->keys[index+1], &node->keys[index], (node->num_keys - index) * sizeof(int))`.

4. **Debugging**: Compile with `-g -DDEBUG` flags to include debug symbols and assertions. Use `assert(node->num_keys >= 0 && node->num_keys <= 2*t-1)` liberally in node functions.
5. **Const Correctness**: For read-only functions like `node_find_key_index`, mark the node parameter as `const`: `int node_find_key_index(const BTreeNode* node, ...)`.

F. Milestone Checkpoint

After implementing the node infrastructure:

1. **Compile and run unit tests**:

```bash
gcc -std=c11 -g -DBTREE_INTERNAL -I./include src/node.c tests/test_node.c -o test_node
./test_node
```

2. **Expected behavior**:
   - All tests should pass (create, destroy, basic insert/remove in node).
   - No memory leaks (verify with Valgrind: `valgrind --leak-check=full ./test_node`).
   - The `node_find_key_index` function should correctly locate keys using binary search.
3. **Manual verification**:
   - Create a node with `t=3` (max 5 keys).
   - Insert keys [10, 20, 30, 40] in order.
   - Verify `num_keys` updates correctly.
   - Search for key 20 should return index 1.
   - Search for key 25 should return child index 2 (between 20 and 30).
4. **Signs of trouble**:
   - **Segmentation fault**: Likely uninitialized pointers in `children` array.
   - **Incorrect search results**: Binary search bounds are wrong.
   - **Memory leak**: Forgot to free arrays in `node_destroy`.

6. Component Design: Search Operation

> **Milestone(s):** Milestone 2 (Search)

The search operation is the fundamental read operation in a B-tree and serves as the foundation upon which more complex operations like insertion and deletion are built. Unlike simpler binary search trees where each node contains a single decision point, a B-tree node contains multiple keys, allowing it to make multiple comparisons at each level before deciding which child subtree to explore. This design dramatically reduces

the tree's height and, consequently, the number of disk accesses required to locate a key—the core optimization that makes B-trees suitable for disk-based storage systems.

Mental Model: Finding a Topic in a Textbook

Imagine you're searching for information about "binary search" in a computer science textbook. The textbook has three levels of organization:

1. **The index at the back** (the **root node**): This contains high-level topic names with page ranges for each chapter.
2. **Chapter opening pages** (internal nodes): Each chapter has its own mini-index listing sections within that chapter.
3. **Individual pages** (leaf nodes): The actual content with detailed explanations.

Your search process mirrors the B-tree search algorithm:

- You start at the index (root). Using alphabetical order, you quickly determine that "binary search" falls between "binary trees" (page 150-180) and "B-trees" (page 200-250). This tells you to look in Chapter 7 (pages 150-180).
- You turn to Chapter 7's opening page (an internal node). It lists sections: "7.1 Basic Trees," "7.2 Binary Search Trees," "7.3 AVL Trees." You find that "binary search" belongs in section 7.2.
- You navigate to section 7.2 (a leaf node) and scan the pages until you find the exact discussion of binary search.

At each step, you're performing a **binary search within a sorted list** to narrow down your search space, then descending to the next appropriate level. The B-tree's structure ensures that even with millions of keys, you only need to examine a handful of "index pages" (nodes) to find your target.

Search Algorithm Steps

The search operation follows a recursive divide-and-conquer strategy that leverages both the sorted order within nodes and the hierarchical tree structure. The algorithm maintains the invariant that all keys in a subtree rooted at a child pointer `children[i]` are bounded by `keys[i-1]` and `keys[i]` (with special handling for the first and last child).

Search Algorithm (`search_recursive`):

1. **Initialize search at current node**: Begin with the current node (initially the root) and the target key.
2. **Perform binary search within node**: Use binary search on the node's sorted `keys` array to find the smallest index `i` such that `key ≤ keys[i]`. The binary search returns either:
   - The exact index where `key == keys[i]` (found case)
   - The index where `key` would be inserted (not found, indicates which child to descend into)
3. **Check for key match**:

- **If found** (`key == keys[i]`):
  - If this node is a leaf: Return a successful `SearchResult` with the key's associated value (if values are stored).
  - If this node is internal: The key acts as a separator between child subtrees but can also store a value (in some B-tree variants). For our implementation, we'll treat all keys as potentially having associated values, so return success with the value at `values[i]` .
- **If not found** (`key < keys[i]` or `i == num_keys`):
  - If this node is a leaf: The key doesn't exist in the tree. Return a "not found" `SearchResult` .
  - If this node is internal: Determine the appropriate child subtree to search:
    - If `key < keys[0]` : Descend into `children[0]` (all keys in this subtree are less than `keys[0]`).
    - Else: Descend into `children[i]` (all keys in this subtree are between `keys[i-1]` and `keys[i]`).

4. **Recurse into child**: Call `search_recursive` on the selected child node with the same target key.
5. **Return result**: Propagate the result (found/not found) back up the recursion chain.

The algorithm's time complexity is **O(log$_{t}$ n)** node accesses and **O(log t)** comparisons per node, where `t` is the minimum degree and `n` is the total number of keys. Each node access corresponds to a potential disk read in a real system.

ADR: Recursive vs. Iterative Search

Decision: Use Recursive Implementation for Search

- **Context**: We need to implement the search operation for an in-memory B-tree for educational purposes. The primary considerations are code clarity for learners versus potential performance optimizations. While real database systems might optimize for stack depth or cache locality, our implementation prioritizes readability and alignment with the recursive nature of tree algorithms.
- **Options Considered**:
  1. **Pure recursive implementation**: Clear mirroring of the algorithm description, easy to understand recursion flow, but uses call stack space proportional to tree height.
  2. **Iterative implementation with explicit stack**: Avoids recursion overhead, allows better control over memory, but adds complexity with stack management code.
  3. **Tail-recursive iterative implementation**: Uses a while loop to follow child pointers without a stack, but only works for simple search without additional bookkeeping for future operations.
- **Decision**: Use a pure recursive implementation for the core search operation.
- **Rationale**:
  - **Educational clarity**: The recursive implementation directly corresponds to the algorithm steps described in textbooks and documentation. Learners can trace the recursion to understand tree traversal.
  - **Consistency with other operations**: Insertion and deletion also use recursive approaches (particularly for proactive splitting during descent), so maintaining a consistent paradigm helps learners.
  - **Adequate performance**: For an in-memory educational implementation with typical tree heights (≤ 10 for millions of keys), recursion depth is not a practical concern. The $O(\log_t n)$ height means even with `t=2` (worst-case fanout) and `n=1,000,000`, height ≤ 20.
  - **Simpler error handling**: No need to manage an explicit stack data structure, reducing code complexity.
- **Consequences**:
  - **Positive**: Clean, readable code that clearly expresses the algorithm. Easy to extend for debugging (e.g., adding recursion depth tracking).
  - **Negative**: Theoretical risk of stack overflow for extremely large trees with small `t`, though practically improbable for educational use cases. Slight performance overhead of function calls.

Comparison of Search Implementation Approaches:

Option	Pros	Cons	Chosen?
Recursive	- Direct algorithm translation - Easy to understand and debug - Consistent with insert/delete operations	- Uses call stack memory - Theoretical stack overflow risk - Slight function call overhead	**Yes**
Iterative with explicit stack	- No recursion depth limits - Better control over memory usage - Potentially better performance	- More complex code - Requires stack management - Less intuitive for learners	No
Tail-recursive iterative	- Minimal memory usage - Simple loop structure - Good performance	- Only works for pure search - Cannot be extended for insertion/deletion bookkeeping - Requires different paradigm	No

Common Pitfalls: Search Logic

Implementing B-tree search seems straightforward but contains subtle traps that can lead to incorrect results, infinite recursion, or crashes.

⚠️ Pitfall: Off-by-one errors in binary search bounds

- **Description**: Using incorrect indices in the binary search loop, such as searching in range `[0, num_keys]` instead of `[0, num_keys-1]`, or incorrectly calculating the midpoint.
- **Why it's wrong**: May fail to find existing keys, return incorrect child indices, or access out-of-bounds memory.
- **Fix**: Implement a standard, verified binary search algorithm. Use inclusive lower bound and exclusive upper bound (`[low, high)`) consistently. Test edge cases: empty node, single key, full node.

⚠️ Pitfall: Forgetting to check leaf status before recursing

- **Description**: Attempting to access `children[i]` when `is_leaf` is true (leaf nodes have null child pointers).
- **Why it's wrong**: Dereferencing a null pointer or uninitialized memory causes segmentation faults.
- **Fix**: Always check `node->is_leaf` before accessing `node->children`. The check should occur after determining the key isn't in the current node but before attempting to descend.

⚠️ Pitfall: Incorrect child index selection when key not found

- **Description**: When `key` is not found in a node, choosing the wrong child index (e.g., using `i` when should use `i+1` or vice versa).

- **Why it's wrong**: The search descends into the wrong subtree and may miss the key even if it exists in the tree.
- **Fix**: Remember the invariant: all keys in subtree `children[i]` are less than `keys[i]` (for `i > 0`) and greater than `keys[i-1]`. If binary search returns index `i` where `key < keys[i]`, descend into `children[i]`. If `key > all keys`, descend into `children[num_keys]`.

⚠️ **Pitfall: Not handling duplicate keys (if allowed)**

- **Description**: The basic algorithm assumes unique keys. If duplicates are allowed, binary search may return any matching index.
- **Why it's wrong**: May return different instances of the same key unpredictably, or fail to find all duplicates.
- **Fix**: Define the semantics: return first/last occurrence, or return all values. Modify binary search to find leftmost/rightmost match. For simplicity, our implementation assumes unique keys.

⚠️ **Pitfall: Ignoring the return value of recursive calls**

- **Description**: Calling `search_recursive` on a child but not returning its result to the caller.
- **Why it's wrong**: The search always returns "not found" from the current node, even if a deeper node found the key.
- **Fix**: Always return the result of the recursive call: `return search_recursive(child, key);`.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Search Algorithm	Recursive implementation with binary search per node	Iterative with manual stack for predictable memory usage
Key Comparison	Simple integer comparison (`key < node->keys[i]`)	Generic comparator function pointer for any key type
Result Tracking	Basic `found` boolean with value pointer	Extended `SearchResult` with metadata (comparison count, node access count)

B. Recommended File/Module Structure

The search operation spans two files in the C implementation:

```
btree/
├── include/
│   └── btree.h          ← Public API declarations (btree_search, SearchResult)
├── src/
│   ├── btree.c          ← Public API implementation (btree_search wrapper)
│   ├── node.c           ← Node-level operations (node_find_key_index)
│   └── search.c         ← Internal recursive search (search_recursive)
└── tests/
    └── test_search.c    ← Search-specific unit tests
```

C. Infrastructure Starter Code

Complete `SearchResult` struct and public API wrapper (to be placed in `btree.h` and `btree.c`):

```c
/* btree.h */

#ifndef BTREE_H

#define BTREE_H


#include <stdbool.h>

#include <stddef.h>


/* Search result structure */

typedef struct {

    bool found;          /* Whether the key was found */

    void* value;         /* Associated value (NULL if not found or no value) */

    int comparisons;     /* Number of key comparisons performed (for analysis) */

} SearchResult;


/* B-tree structure (forward declaration) */

typedef struct BTree BTree;


/* Public search API */

SearchResult btree_search(BTree* tree, int key);


#endif /* BTREE_H ```
```

```c
/* btree.c */

#include "btree.h"

#include "node.h"

#include <stdlib.h>


/* B-tree structure definition */

struct BTree {

    struct BTreeNode* root;  /* Root node */

    int t;                   /* Minimum degree */

    size_t key_count;        /* Total keys in tree */

};


/* Public search function */

SearchResult btree_search(BTree* tree, int key) {

    SearchResult result = {false, NULL, 0};



    if (tree == NULL || tree->root == NULL) {

        return result;

    }



    /* Start recursive search from root */

    return search_recursive(tree->root, key);

}
```

D. Core Logic Skeleton Code

Binary search helper function (`node.c`):

```c
#include "node.h"
#include <assert.h>

/*
 * Performs binary search within a node's sorted keys array.
 * Returns the index where the key is found, or the index where
 * it should be inserted (first position where keys[i] >= key).
 *
 * Also increments the comparison count for performance tracking.
 */
int node_find_key_index(BTreeNode* node, int key, int* comparisons) {
    /* TODO 1: Handle empty node case - return index 0 */

    /* TODO 2: Initialize binary search bounds: low = 0, high = node->num_keys */

    /* TODO 3: While low < high:
        - Calculate mid = low + (high - low) / 2
        - Increment comparison count if comparisons pointer is not NULL
        - Compare key with node->keys[mid]
        - If key == node->keys[mid], return mid (exact match)
        - If key < node->keys[mid], set high = mid
        - Else set low = mid + 1
     */

    /* TODO 4: Return low (insertion position when key not found) */

    return 0; /* Placeholder */
```

```
}
```

Internal recursive search function (`search.c`):

```c
#include "btree.h"

#include "node.h"

#include <stdlib.h>


/*

 * Recursively searches for a key starting from the given node.

 * Returns SearchResult indicating whether key was found and its value.

 */

SearchResult search_recursive(BTreeNode* node, int key) {

    SearchResult result = {false, NULL, 0};



    if (node == NULL) {

        return result;

    }



    /* TODO 1: Perform binary search within current node using node_find_key_index */

    /*   - Pass &result.comparisons to track comparison count */

    /*   - Store the returned index in variable 'idx' */



    /* TODO 2: Check if key was found in current node (idx < node->num_keys && node->keys[idx] == key) */

    /*   - If found:

          - Set result.found = true

          - Set result.value = node->values[idx] (if values exist, else NULL)

          - Return result

     */



    /* TODO 3: Handle not-found case */
```

```
    /*   - If node is a leaf (node->is_leaf == true):
          - Key doesn't exist in tree, return result (found = false)
     */

    /* TODO 4: Determine correct child index for recursion */
    /*   - If key < node->keys[0] or node->num_keys == 0: child_idx = 0
       - Else: child_idx = idx (from binary search result)
       - Ensure child_idx is within [0, node->num_keys] range
     */

    /* TODO 5: Safety check - verify child pointer is not NULL */
    /*   - Assert: node->children[child_idx] != NULL
       - If NULL, return result (found = false) as error fallback
     */

    /* TODO 6: Recursively search in the appropriate child subtree */
    /*   - return search_recursive(node->children[child_idx], key);
     */

    return result;
}
```

E. Language-Specific Hints for C

- **Binary Search Implementation**: Use `int mid = low + (high - low) / 2;` to avoid integer overflow that could occur with `(low + high) / 2`.
- **Comparison Counting**: Increment `*comparisons` only when the pointer is not NULL to allow optional tracking.
- **Child Pointer Validation**: In debug builds, use `assert(node->children[child_idx] != NULL);` to catch invariant violations early.

- **Recursion Depth**: While recursion depth is limited, you can add a debug-only depth parameter to detect unexpectedly deep recursion.
- **Memory Access**: Always validate array bounds before accessing `node->keys[idx]` or `node->children[idx]`.

F. Milestone Checkpoint

After implementing the search operation:

1. **Compile and run the search tests**:

```
gcc -o test_search src/btree.c src/node.c src/search.c tests/test_search.c -I./include
./test_search
```

2. **Expected behavior**:
   - Empty tree search returns `found = false`
   - Search for existing key returns `found = true` with correct value
   - Search for non-existent key returns `found = false`
   - All tests pass without segmentation faults
3. **Manual verification**:
   - Create a small B-tree manually (insert a few keys)
   - Search for each inserted key - should succeed
   - Search for keys between inserted values - should fail
   - Search for key less than all keys - should fail
   - Search for key greater than all keys - should fail
4. **Signs of problems**:
   - **Segmentation fault**: Likely accessing NULL child pointer or out-of-bounds array index.
   - **Incorrect found status**: Binary search or child index logic is wrong.
   - **Infinite recursion**: Not checking leaf status or incorrect child selection.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Search always returns "not found" even for inserted keys	Binary search returning wrong index or not checking exact match	Add debug prints in `node_find_key_index` to trace comparisons	Verify binary search bounds and equality check
Segmentation fault when searching deep trees	Accessing `children[i]` when `is_leaf` is true	Add assert: `assert(!node->is_leaf	
Search returns wrong value for found key	Values array not properly aligned with keys array	Verify `node->values[idx]` corresponds to `node->keys[idx]`	Ensure insertion maintains key-value alignment
Recursion depth exceeds expected height	Child index calculation wrong, causing cycles	Print recursion depth and node addresses	Check child index logic for edge cases

---

7. Component Design: Insertion with Splitting

> **Milestone(s):** Milestone 3 (Insert with Split)

The insertion operation is where a B-tree demonstrates its self-balancing nature. Unlike binary search trees that grow vertically, B-trees grow horizontally (by filling nodes) and only increase height when absolutely necessary through a controlled root split. This design maintains the **disk-friendly property** of keeping tree height logarithmic while ensuring each node remains densely packed, minimizing disk accesses. The insertion algorithm's core challenge is maintaining all B-tree invariants—especially the key count bounds and sorted order—while adding new keys.

Mental Model: Adding a Book to a Full Shelf

Imagine organizing books in a library using a multi-level catalog system. Each catalog drawer (node) can hold a fixed number of index cards (keys). When a drawer becomes completely full, you can't simply add another card—you must split it:

1. **Identify the target drawer**: Start at the main catalog (root) and follow the alphabetical dividers to find which drawer should contain the new book's card.
2. **Prevent overflow during descent**: Before entering any drawer, if you see it's completely full, you split it proactively. You take out the middle card, create a new drawer, and place half the cards in each. You then

add the middle card to the parent drawer as a new divider, telling you which drawer contains which range.

3. **Insert the new card**: Once you reach the correct leaf drawer (which now has space because you split full drawers on the way down), you insert the new card in sorted order.
4. **Handle a full main catalog**: If the very first catalog (root) is full when you start, you split it too. This creates a new top-level catalog with just one divider card pointing to two lower drawers, increasing the total number of catalog levels by one.

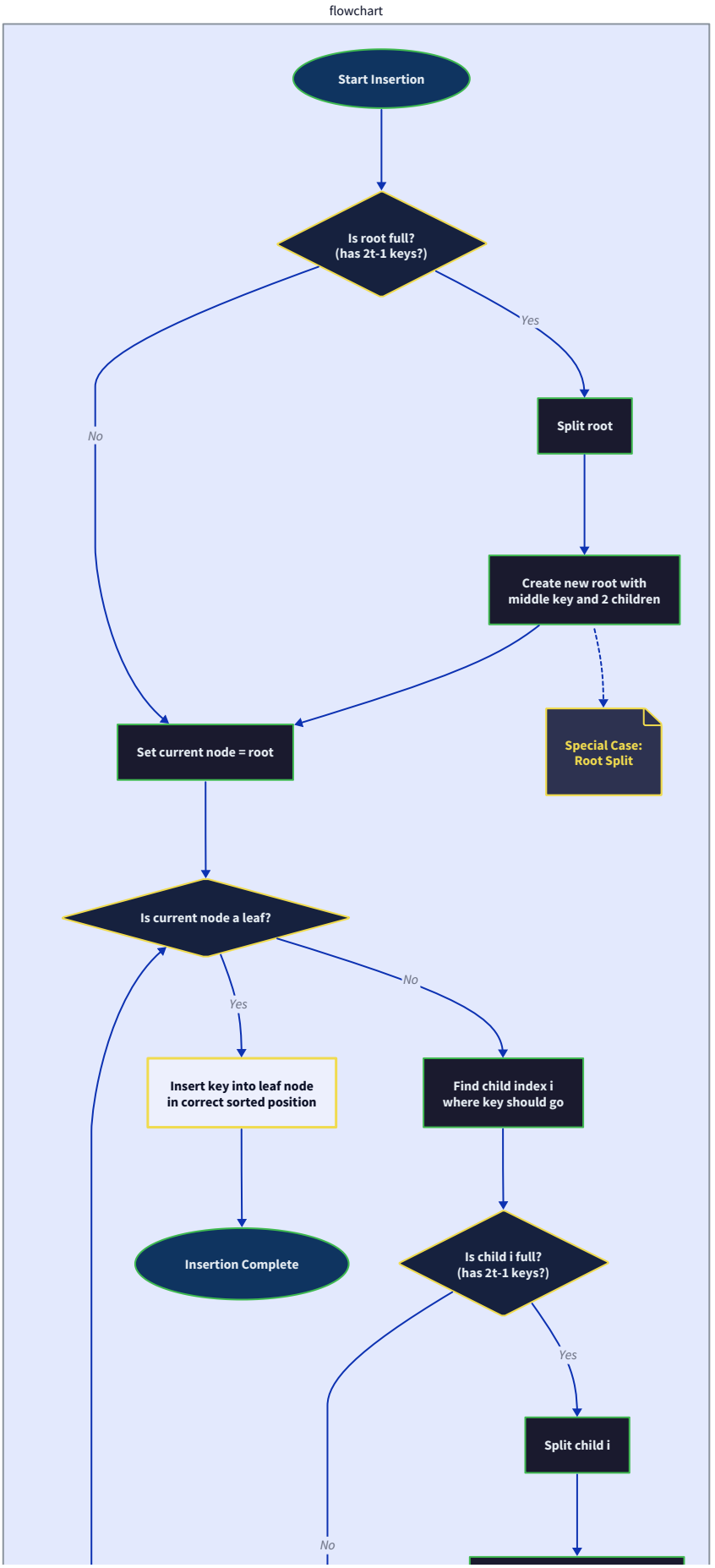
This proactive splitting during descent ensures you never attempt to insert into a full node, simplifying the algorithm. The middle key promotion resembles adding a new divider to a parent drawer, which may cause a cascade of splits upward if parents were also full.

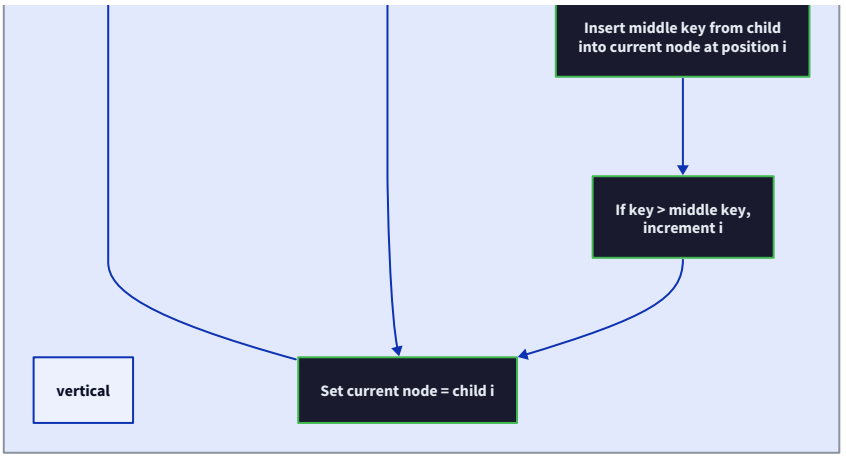
Insertion and Split Algorithm Steps

The insertion algorithm follows a **recursive, top-down approach** that maintains the invariant that any node we descend into is not full. This is achieved by proactively splitting full nodes before recursing into them. The algorithm consists of two main functions: the public `btree_insert` wrapper and the internal recursive `insert_non_full`.

Overall Insertion Flow

Insertion with Splitting Algorithm

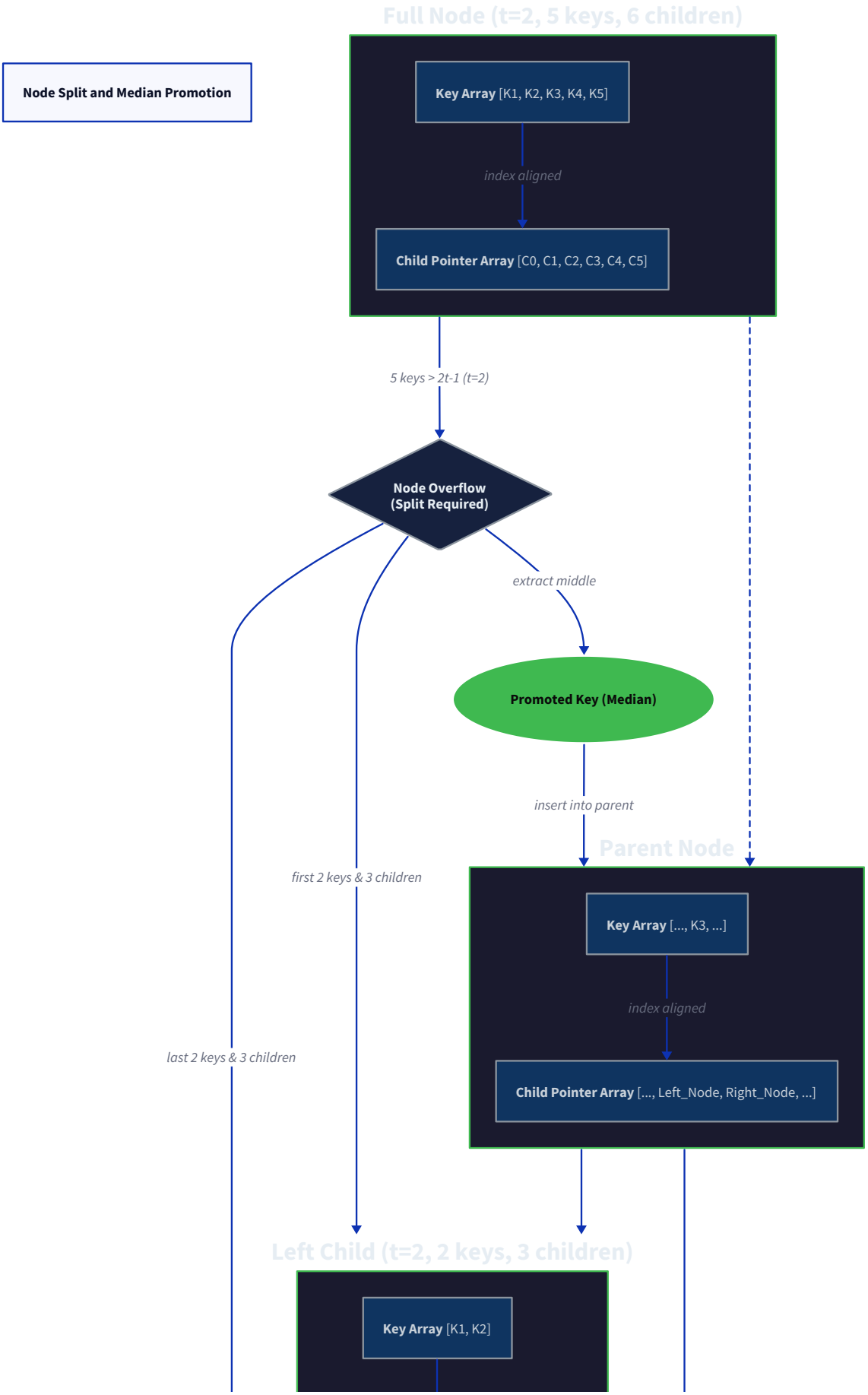


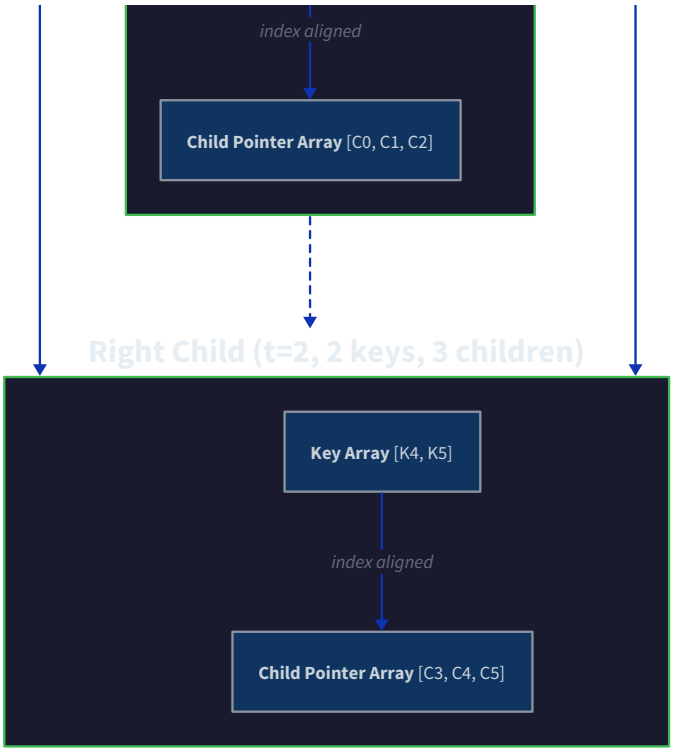


1. **Public Insertion Interface** (`btree_insert`):
   - **Step 1**: Validate inputs (tree pointer, key).
   - **Step 2**: If the root is full (`node_is_full(root, t)` returns true), create a new root node that will become the tree's new root. Split the old root by calling `node_split_child` on the new root at index 0. This increases tree height by one.
   - **Step 3**: Call `insert_non_full` on the (now guaranteed non-full) root to perform the actual insertion.
2. **Internal Recursive Insertion** (`insert_non_full`):
   - **Step 4**: If the current node is a leaf (`node->is_leaf == true`), perform direct insertion:
     - Use `node_find_key_index` to find the correct position `i` for the new key.
     - Call `node_insert_key` to insert the key (and associated value) at position `i` , shifting existing keys right.
     - Return success.
   - **Step 5**: If the current node is internal, find the child subtree where the key belongs:
     - Use `node_find_key_index` to get index `i` (position where key would be or child index).
     - **Step 6**: Check if the target child at `children[i]` is full using `node_is_full` .
     - **Step 7**: If full, split this child:
       - Call `node_split_child(current_node, i, t)` to split child `children[i]` .
       - After splitting, compare the promoted key (now at `keys[i]`) with the new key to decide which of the two new children to descend into (update `i` if needed).
     - **Step 8**: Recursively call `insert_non_full` on the appropriate child (now guaranteed non-full).

Node Splitting Subroutine (`node_split_child`)

Splitting a full child is a critical sub-operation that transforms one full node into two half-full nodes and promotes the median key to the parent.


Node Split and Median Promotion
Full Node (t=2, 5 keys, 6 children)
Key Array [K1, K2, K3, K4, K5]
index aligned
Child Pointer Array [C0, C1, C2, C3, C4, C5]
5 keys > 2t-1 (t=2)
Node Overflow
(Split Required)
extract middle
Promoted Key (Median)
insert into parent
Parent Node
first 2 keys & 3 children
Key Array [..., K3, ...]
index aligned
last 2 keys & 3 children
Child Pointer Array [..., Left_Node, Right_Node, ...]
Left Child (t=2, 2 keys, 3 children)
Key Array [K1, K2]



Given a parent node `P` and child index `i` where `child = P->children[i]` is full (has `2t-1` keys):

1. **Create a new sibling node** `new_child` with the same leaf status as the original child.
2. **Distribute keys**: The original child keeps the first `t-1` keys (indices `0` to `t-2`). The new sibling gets the last `t-1` keys (indices `t` to `2t-2`). The median key at index `t-1` will be promoted.
3. **Distribute children** (if internal node): Similarly, the first `t` children pointers stay with the original child; the remaining `t` children pointers go to the new sibling.
4. **Adjust key counts**: Set original child's `num_keys = t-1` and new sibling's `num_keys = t-1`.
5. **Promote the median key**:
   - Make space in parent `P` at index `i` by shifting parent keys and child pointers right.
   - Insert the median key into `P->keys[i]`.
   - Insert `new_child` as a new child pointer at `P->children[i+1]`.
   - Increment parent's `num_keys` by one.

This operation ensures both resulting nodes satisfy the minimum occupancy requirement (at least `t-1` keys) and the parent gains one key while maintaining the invariant that internal nodes have one more child than keys.

ADR: Proactive vs. Reactive Splitting

Decision: Proactive Splitting During Descent

- **Context**: The B-tree insertion algorithm must handle full nodes to maintain the key count bounds (maximum `2t-1` keys per node). Two approaches exist: proactively split full nodes during the downward traversal before insertion, or allow temporary overflow and fix it on the way back up (reactive).
- **Options Considered**:
  1. **Proactive splitting (top-down)**: Split any full node encountered during descent before recursing into it.
  2. **Reactive splitting (bottom-up)**: Allow insertion into full leaves, then split and propagate upward if needed.
  3. **Hybrid approach**: Allow leaves to become temporarily overfull, but split internal nodes proactively.
- **Decision**: Implement proactive splitting during the downward traversal for all nodes (both internal and leaf).
- **Rationale**:
  - **Simplicity for learners**: Proactive splitting ensures the recursive insertion function (`insert_non_full`) only ever operates on non-full nodes, reducing the number of cases to handle. The base case (leaf insertion) becomes trivial—just insert into a non-full array.
  - **Single-pass algorithm**: The algorithm makes only one pass from root to leaf, which is easier to reason about and debug compared to splitting on the way back up with potentially complex upward propagation logic.
  - **Consistency with textbook implementations**: Most academic references (e.g., CLRS) use the proactive approach, making it easier for learners to cross-reference.
  - **Predictable memory access**: By splitting during descent, we can allocate new nodes immediately rather than potentially allocating multiple nodes during upward propagation, which might be harder to manage in error conditions.
- **Consequences**:
  - **Slightly less efficient**: Some splits may occur that wouldn't be strictly necessary if the key ends up in a different subtree (though this is rare).
  - **Root split requires special handling**: Since the root has no parent, we must check and split it before starting descent, which is a special case in the public `btree_insert` function.
  - **Cleaner recursion**: The recursive helper `insert_non_full` has a simpler precondition (node is not full), making its implementation more straightforward.

Option	Pros	Cons	Chosen?
Proactive splitting	Single pass; simpler recursion; consistent with textbooks	May split nodes unnecessarily in rare cases	**Yes**
Reactive splitting	Only splits when absolutely needed; can be more efficient	Two-pass algorithm; more complex recursion with upward propagation	No
Hybrid approach	Balances efficiency and simplicity	Inconsistent logic between leaves and internal nodes; harder to teach	No

Common Pitfalls: Splitting and Promotion

Implementing insertion with splitting involves several subtle details that can easily lead to bugs. Below are the most common pitfalls, why they're problematic, and how to avoid them.

⚠️ Pitfall 1: Splitting Non-Full Nodes

- **Description**: Calling the split operation on a node that has fewer than `2t-1` keys, often due to incorrect `node_is_full` check or off-by-one in the minimum degree `t`.
- **Why it's wrong**: Splitting a non-full node violates B-tree invariants by creating nodes that may have fewer than `t-1` keys (underfull). This can cascade and break the tree structure during subsequent operations.
- **How to avoid**: Always verify `node_is_full(child, t)` returns `true` before splitting. Double-check that `node_is_full` correctly implements `num_keys == 2*t - 1`.

⚠️ Pitfall 2: Incorrect Median Index Calculation

- **Description**: Using the wrong index for the median key during splitting, such as using `t` instead of `t-1` (0-indexed) or miscalculating due to confusion between key counts and child counts.
- **Why it's wrong**: Promoting the wrong key breaks the sorted order invariant. If the median is off by one, the resulting nodes may have incorrect key ranges, causing search failures.
- **How to avoid**: Remember that for a node with `2t-1` keys (indices `0` to `2t-2`), the median key is at index `t-1`. Use concrete examples: for `t=3`, `2t-1=5` keys, indices 0-4, median at index 2.

⚠️ Pitfall 3: Forgetting to Update Child Pointers After Split

- **Description**: After creating a new sibling node, failing to update the parent's child pointer array to include the new sibling at the correct position, or incorrectly shifting existing child pointers.
- **Why it's wrong**: The parent will have an incomplete or incorrect list of children, potentially losing references to entire subtrees. This causes memory leaks and broken tree connectivity.
- **How to avoid**: When inserting the new sibling into the parent's child array at index `i+1`, ensure all child pointers from index `i+1` onward are shifted right by one position first. Diagram the pointer arrays before and after.

⚠️ Pitfall 4: Mishandling Root Split

- **Description**: Forgetting to handle the special case where the root is full, or incorrectly implementing the root split (e.g., not updating the tree's root pointer, not setting the new root's `is_leaf` to `false`).
- **Why it's wrong**: If the root remains full, insertion cannot proceed without violating capacity bounds. If the new root isn't properly initialized, the tree becomes malformed.
- **How to avoid**: In `btree_insert`, check if the root is full before any descent. If so: (1) create a new root node with `is_leaf = false`, (2) make the old root its first child, (3) split the old root via `node_split_child(new_root, 0, t)`, (4) update the tree's root pointer.

⚠️ Pitfall 5: Ignoring the Leaf Flag During Split

- **Description**: When creating the new sibling node during a split, copying the original node's `is_leaf` flag incorrectly (e.g., always setting to `true` or `false`).
- **Why it's wrong**: If an internal node is split but the new sibling is marked as a leaf, its child pointers will be invalid (null for leaves). Conversely, splitting a leaf but marking the new sibling as internal will cause null pointer dereferences when trying to access children.
- **How to avoid**: The new sibling must have the same `is_leaf` status as the node being split. Pass `child->is_leaf` to `node_create` when creating the sibling.

⚠️ Pitfall 6: Not Adjusting Search Index After Split

- **Description**: After splitting a child during descent, failing to re-evaluate which child to descend into based on the promoted key. The original index `i` might no longer be correct.
- **Why it's wrong**: The algorithm might descend into the wrong subtree, leading to insertion in an incorrect leaf or failed searches later.
- **How to avoid**: After `node_split_child(parent, i, t)`, compare the new key with the promoted key now at `parent->keys[i]`. If the new key is less, descend into `parent->children[i]`; if greater, descend into `parent->children[i+1]`.

Implementation Guidance

This section provides concrete implementation skeletons and guidance for the C programming language, following the naming conventions and structure outlined earlier.

Technology Recommendations Table

Component	Simple Option	Advanced Option
Memory Management	Manual `malloc` / `free` with careful ownership tracking	Reference-counted nodes or arena allocator for batch deallocation
Error Handling	Return boolean success/failure, print errors to stderr	Error codes enum with descriptive messages, optional error callback
Debugging Aids	`printf` statements at key decision points	Comprehensive logging macro with levels (DEBUG, INFO, ERROR)
Validation	Manual invariant checks in test code	Built-in `btree_validate` that traverses entire tree verifying all invariants

Recommended File/Module Structure

The insertion functionality extends the existing B-tree module structure:

```
btree/
├── include/
│   └── btree.h             ← Public interface declarations
├── src/
│   ├── btree.c             ← Public API implementations (btree_insert)
│   ├── node.c              ← Node operations (node_split_child, node_insert_key)
│   └── btree_private.h     ← Internal function declarations (insert_non_full)
└── tests/
    ├── test_insert.c       ← Insertion-specific tests
    └── test_operations.c   ← Integrated tests for all operations
```

Infrastructure Starter Code

The following helper functions for node manipulation should be implemented completely as they are prerequisites for the insertion logic but not the core learning goal:

```c
// In node.c - Complete implementation

#include <stdlib.h>

#include <string.h>

#include "btree_private.h"


/**

 * Inserts a key-value pair into a node at the specified index.

 * Shifts existing keys and values (if any) to the right.

 * For internal nodes, children pointers are NOT shifted by this function.

 */

void node_insert_key(BTreeNode* node, int index, int key, void* value) {

    // Shift keys right from index to num_keys-1

    for (int i = node->num_keys - 1; i >= index; i--) {

        node->keys[i + 1] = node->keys[i];

    }



    // Shift values right if values array exists

    if (node->values != NULL) {

        for (int i = node->num_keys - 1; i >= index; i--) {

            node->values[i + 1] = node->values[i];

        }

        node->values[index] = value;

    }



    // Insert new key

    node->keys[index] = key;

    node->num_keys++;
```

```c
}

/**
 * Creates a new empty node with the given minimum degree and leaf status.
 * Allocates arrays for keys, values (optional), and children (if internal).
 */
BTreeNode* node_create(int t, bool is_leaf) {
    BTreeNode* node = malloc(sizeof(BTreeNode));
    if (!node) return NULL;

    node->keys = malloc(sizeof(int) * (2 * t - 1));
    node->values = NULL;  // Optional: allocate if storing values
    node->children = is_leaf ? NULL : malloc(sizeof(BTreeNode*) * (2 * t));
    node->num_keys = 0;
    node->is_leaf = is_leaf;

    if (!node->keys || (!is_leaf && !node->children)) {
        free(node->keys);
        free(node->children);
        free(node);
        return NULL;
    }

    // Initialize children to NULL if internal node
    if (!is_leaf) {
        for (int i = 0; i < 2 * t; i++) {
            node->children[i] = NULL;
```

```
        }
    }

    return node;
}

/**
 * Checks if a node is full (contains 2t-1 keys).
 */
bool node_is_full(BTreeNode* node, int t) {
    return node->num_keys == (2 * t - 1);
}

/**
 * Binary search within a node's keys to find the position of a key.
 * Returns the index where the key is found or should be inserted.
 * Increments comparisons counter if provided.
 */
int node_find_key_index(BTreeNode* node, int key, int* comparisons) {
    int left = 0;
    int right = node->num_keys - 1;
    int mid;

    while (left <= right) {
        mid = left + (right - left) / 2;
        if (comparisons) (*comparisons)++;

        if (node->keys[mid] == key) {
```

```
            return mid;  // Key found
        } else if (node->keys[mid] < key) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    // Key not found, return where it should be inserted
    return left;
}
```

Core Logic Skeleton Code

The following skeletons show the insertion functions with TODO comments mapping to the algorithm steps described earlier. Learners should implement the missing logic.

```c
// In btree_private.h (internal declarations)
#ifndef BTREE_PRIVATE_H
#define BTREE_PRIVATE_H

#include "btree.h"

// Internal recursive insertion helper
bool insert_non_full(BTreeNode* node, int key, void* value, int t);

// Split child of parent at index i
void node_split_child(BTreeNode* parent, int i, int t);

#endif
```

```c
// In node.c - Split implementation skeleton

/**
 * Splits the i-th child of parent, which must be full.
 * Creates a new node, moves half of the child's keys and children
 * to the new node, and promotes the median key to the parent.
 */
void node_split_child(BTreeNode* parent, int i, int t) {
    // TODO 1: Get pointer to the full child to be split
    // BTreeNode* full_child = parent->children[i];

    // TODO 2: Create new sibling node with same leaf status as full_child
    // BTreeNode* new_sibling = node_create(t, full_child->is_leaf);

    // TODO 3: Copy the last (t-1) keys from full_child to new_sibling
    // for (int j = 0; j < t-1; j++) {
    //     new_sibling->keys[j] = full_child->keys[j + t];
    // }
    // new_sibling->num_keys = t-1;

    // TODO 4: If full_child is internal (not leaf), copy the last t children pointers
    // if (!full_child->is_leaf) {
    //     for (int j = 0; j < t; j++) {
    //         new_sibling->children[j] = full_child->children[j + t];
    //     }
    // }

    // TODO 5: Reduce key count of full_child to t-1
```

```
    // full_child->num_keys = t-1;


    // TODO 6: Make space in parent for new key and child pointer
    // Shift parent keys from index i to num_keys-1 right by one
    // Shift parent children from index i+1 to num_keys right by one


    // TODO 7: Insert median key (full_child->keys[t-1]) into parent at index i
    // parent->keys[i] = full_child->keys[t-1];


    // TODO 8: Insert new_sibling as parent's child at index i+1
    // parent->children[i+1] = new_sibling;


    // TODO 9: Increment parent's key count
    // parent->num_keys++;
}
```

```c
// In btree.c - Public insertion and recursive helper

#include "btree_private.h"

#include <stdlib.h>

#include <stdio.h>


/**

 * Public API: Insert a key-value pair into the B-tree.

 * Returns true on success, false on failure (e.g., memory allocation error).

 */

bool btree_insert(BTree* tree, int key, void* value) {

    if (!tree || !tree->root) return false;



    BTreeNode* root = tree->root;



    // TODO 1: Special case - if root is full, split it

    // if (node_is_full(root, tree->t)) {

    //     // Create new root

    //     BTreeNode* new_root = node_create(tree->t, false);

    //     if (!new_root) return false;

    //

    //     // Make old root the first child of new root

    //     new_root->children[0] = root;

    //

    //     // Split the old root

    //     node_split_child(new_root, 0, tree->t);

    //

    //     // Update tree root
```

```
    //     tree->root = new_root;
    //
    //     // Update root pointer for insertion
    //     root = new_root;
    // }


    // TODO 2: Now root is not full, call recursive helper
    // bool success = insert_non_full(root, key, value, tree->t);


    // TODO 3: Update tree->key_count if insertion successful
    // if (success) tree->key_count++;


    // return success;
    return false; // placeholder
}


/**
 * Internal recursive function to insert into a subtree rooted at a non-full node.
 * Precondition: node is not full (node_is_full(node, t) == false).
 */
bool insert_non_full(BTreeNode* node, int key, void* value, int t) {
    // TODO 1: Find the index where key belongs or exists
    // int i = node_find_key_index(node, key, NULL);


    // TODO 2: Handle case where key already exists (optional: update value)
    // if (i < node->num_keys && node->keys[i] == key) {
    //     if (node->values) node->values[i] = value;
```

```
    //     return true; // Key updated
    // }

    // TODO 3: If node is a leaf, insert key at position i
    // if (node->is_leaf) {
    //     node_insert_key(node, i, key, value);
    //     return true;
    // }

    // TODO 4: Node is internal - prepare to descend
    // BTreeNode* child = node->children[i];

    // TODO 5: If child is full, split it first
    // if (node_is_full(child, t)) {
    //     node_split_child(node, i, t);
    //
    //     // After split, check which child to descend into
    //     // Compare key with node->keys[i] (the promoted key)
    //     // if (key > node->keys[i]) i++;
    //     // Update child pointer
    //     // child = node->children[i];
    // }

    // TODO 6: Recursively insert into the appropriate child
    // return insert_non_full(child, key, value, t);

    return false; // placeholder
```

```
}
```

Language-Specific Hints for C

1. **Memory Management**: Always check `malloc` return values. When splitting nodes, ensure all allocated memory is properly freed in the destruction path (`node_destroy` should recursively free children).
2. **Array Indexing**: C uses 0-based indexing. The median key in a node with `2t-1` keys is at index `t-1` (0-indexed). Be careful with loop boundaries when shifting arrays.
3. **Pointer Arithmetic**: When shifting elements in arrays, iterate from the end backward to avoid overwriting elements: `for (int i = count-1; i >= index; i--)`.
4. **Boolean Values**: Include `<stdbool.h>` for `bool`, `true`, `false`. The `node->is_leaf` flag should be set correctly when creating nodes.
5. **Debugging**: Use `assert` from `<assert.h>` to validate invariants during development (e.g., `assert(node->num_keys <= 2*t-1)`).

Milestone Checkpoint

After implementing insertion with splitting:

1. **Compilation**: Run `gcc -Wall -Wextra -g src/*.c tests/test_insert.c -o test_insert` to ensure no warnings.
2. **Basic Test**: Create a simple test that inserts keys in increasing order and verifies tree structure:

```c
BTree* tree = btree_create(3);
for (int i = 1; i <= 20; i++) {
    btree_insert(tree, i, NULL);
}
btree_print(tree);  // Should show balanced tree with 3-4 levels
btree_destroy(tree);
```

3. **Validation**: Run the tree validator after each insertion in debug mode:

```c
assert(btree_validate(tree) == true);
```

4. **Expected Behavior**:
   - Inserting 1-10 with `t=3` should produce a root with 1 key after the 7th insertion (first root split).
   - All nodes (except root) should have between `t-1` and `2t-1` keys.
   - In-order traversal (via search) should find all inserted keys.

5. **Signs of Trouble**:
    - **Segmentation fault**: Likely uninitialized child pointers or incorrect array bounds.
    - **Missing keys**: Search fails for recently inserted keys - check split logic and key promotion.
    - **Too many nodes**: Each split should create exactly one new node. Count nodes to verify.
    - **Validation fails**: Implement `btree_validate` to check invariants and pinpoint issues.

8. Component Design: Deletion with Rebalancing

> **Milestone(s):** Milestone 4 (Delete with Rebalancing)

Deletion in a B-tree is the most complex operation, requiring careful maintenance of the **capacity bounds** invariants. Unlike binary search trees where deletion simply removes a node, B-trees must ensure every node (except the root) maintains at least `t-1` keys. When deletion would violate this minimum, the tree performs **borrowing** from a sibling or **merging** nodes to restore balance. This component handles the intricate logic of key removal, underflow detection, and structural rebalancing while preserving all B-tree invariants.

Mental Model: Removing a Book and Consolidating Shelves

Imagine a library with bookshelves organized by the B-tree system from our previous analogies. Each shelf (node) must contain between `t-1` and `2t-1` books (keys). When you remove a book from a shelf:

1. **Simple removal**: If the shelf still has enough books after removal (`≥ t-1`), you simply take the book out and shift the remaining books to fill the gap. The library index (parent node's keys) remains unchanged.
2. **Borrowing from a neighbor**: If removing the book would leave the shelf with too few books (`< t-1`), you first check if a neighboring shelf has extra books to spare. If the left neighbor has more than the minimum, you:
    - Take the rightmost book from the left neighbor
    - Move the dividing index card from the parent (which sits between these two shelves) down to your shelf
    - Move the neighbor's book up to replace that parent index card

    This rotates a key through the parent, maintaining alphabetical order while fixing the underflow.
3. **Merging shelves**: If neither neighbor has extra books, you must merge shelves. You combine your shelf with one neighbor, plus the parent's dividing index card, creating one shelf with exactly `2t-2` books (which is valid since `2t-2 ≤ 2t-1`). This reduces the number of shelves by one and may cause the parent shelf to now have too few index cards, potentially triggering borrowing or merging cascading upward.
4. **Empty root shelf**: If merging reaches the root and leaves it with no books (keys), the merged shelf becomes the new root, reducing the tree's height by one—like removing a now-unnecessary floor from the

library's catalog system.

This process ensures the library remains efficiently navigable: shelves never become too empty (wasting space) or too full (requiring frequent splitting), and the catalog hierarchy adjusts gracefully as books are removed.

Deletion and Rebalance Algorithm Steps

The deletion algorithm is implemented in `delete_from_subtree`, a recursive function that descends the tree, handles various deletion cases, and repairs underflows on the way back up. The public interface `btree_delete` simply calls this helper on the root.

Main Deletion Procedure

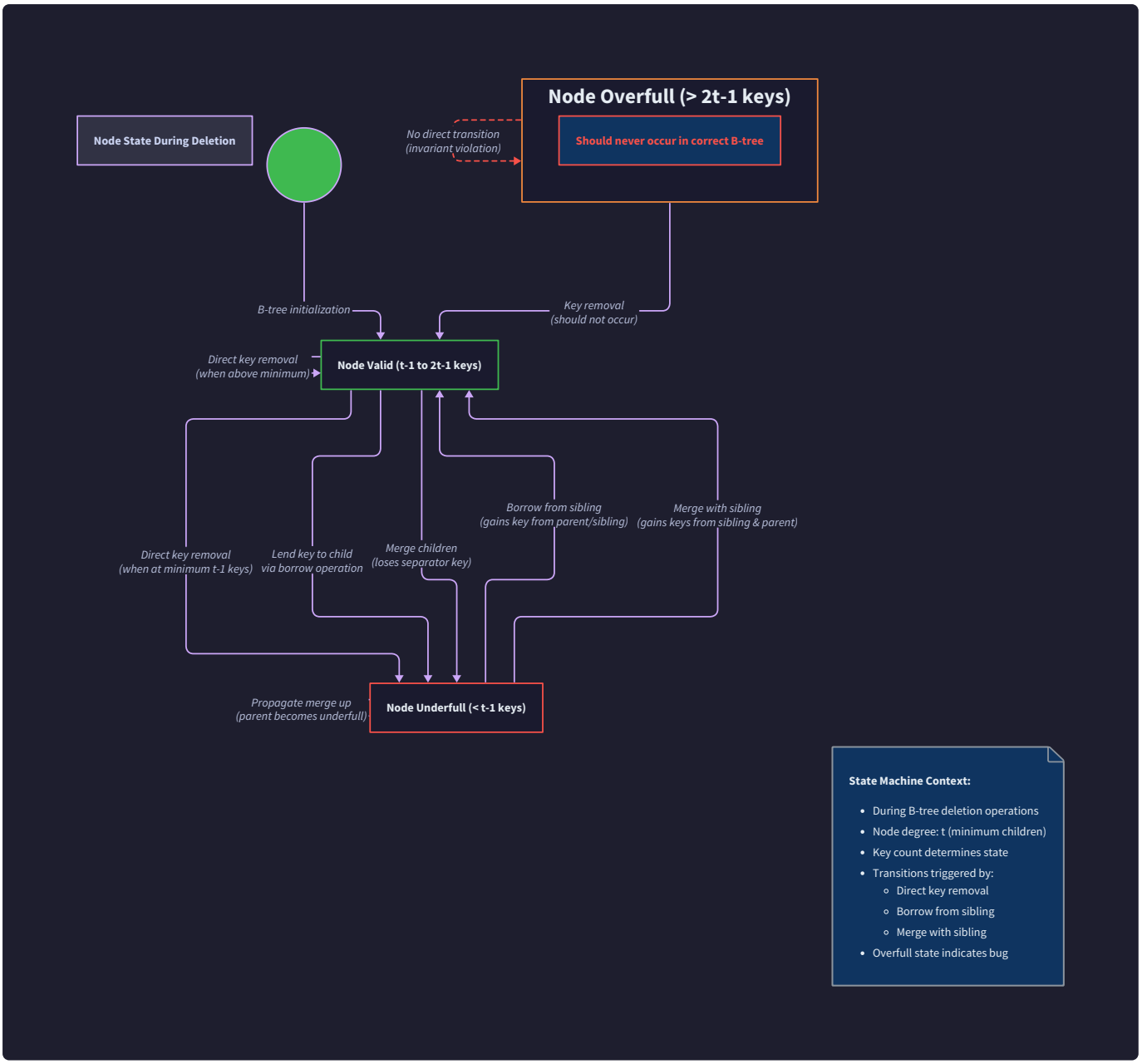
1. **Locate the key in the current node**: Use `node_find_key_index` to perform binary search within the current node's keys. This returns either the exact index where the key exists or the index of the child subtree where the key would be located.
2. **Case 1: Key found in current node (internal node)**:
   - **Subcase 1A: Current node is a leaf**: If `node->is_leaf` is true, simply call `node_remove_key` at the found index. This removes the key and shifts subsequent keys left.
   - **Subcase 1B: Current node is internal**: The key serves as a separator between two subtrees. We cannot simply remove it without disrupting the tree structure. Instead:
     1. **Check left child's capacity**: If the left child (at child index `key_index`) has at least `t` keys, find the **in-order predecessor** (the largest key in the left subtree) by traversing rightmost in that subtree.
     2. **Replace and recurse**: Replace the target key with this predecessor key, then recursively delete that predecessor key from the left child.
     3. **Alternative: right child**: If the left child has only `t-1` keys but the right child (at child index `key_index + 1`) has at least `t` keys, use the **in-order successor** (smallest key in right subtree) instead, found by traversing leftmost in the right subtree.
     4. **Fallback: merge children**: If both adjacent children have the minimum `t-1` keys, merge them along with the separator key into one node, then recursively delete the key from this merged node.
3. **Case 2: Key not found in current node**:
   - **Ensure child can lose a key**: Before descending to child at index `child_idx`, check if that child has only `t-1` keys (minimum). If so, we must **strengthen** it by either:
     - **Borrowing from left sibling**: If the immediate left sibling exists and has at least `t` keys, perform a right rotation: move a key from parent down to child, move a key from left sibling up to parent, and adjust child pointers.
     - **Borrowing from right sibling**: If no left sibling or it's at minimum, but right sibling exists with at least `t` keys, perform a left rotation (symmetric to above).

        - **Merging with sibling**: If neither sibling has extra keys, merge the child with a sibling (preferably left if exists) and the separating key from the parent.
    - **Recurse**: After ensuring the child has at least `t` keys (or merging made it larger), recursively call `delete_from_subtree` on the appropriate child.
4. **Underflow repair on return**: After the recursive deletion returns, check if the current node has become underfull (less than `t-1` keys, except if it's the root). If underfull and not root, this indicates a problem—but our proactive strengthening in step 3 should prevent this. However, some algorithms check and repair here as well for robustness.
5. **Root underflow special case**: If after deletion the root has 0 keys but still has a child (meaning the tree height decreased), make that child the new root and free the old root.

The algorithm's complexity is $O(\log_t n)$ with at most $O(\log_t n)$ node accesses, as each recursive call processes one level and operations within a node take O(t) time.

State Transitions During Deletion

The diagram



shows how a node's state changes during deletion operations. The key insight is that we proactively avoid underflow before descending, making the recursive deletion simpler.

Current State	Event	Next State	Actions Taken
Valid (≥ t keys)	Direct key removal (leaf)	Valid or Underfull	Remove key, check if still ≥ t-1
Valid	Key removed from child (recursive return)	Valid	No action needed
Minimum (t-1 keys)	Need to descend through this child	Strengthened to Valid	Borrow from sibling or merge before descending
Underfull (< t-1 keys)	Detected after child deletion (should not occur with proactive approach)	Valid	Borrow or merge with sibling via parent
Root with 1 key, 2 children	Merging of children	Root with 0 keys	Adopt merged child as new root, height decreases

ADR: Predecessor vs. Successor for Internal Delete

Decision: Prefer Predecessor from Left Child for Internal Node Deletion

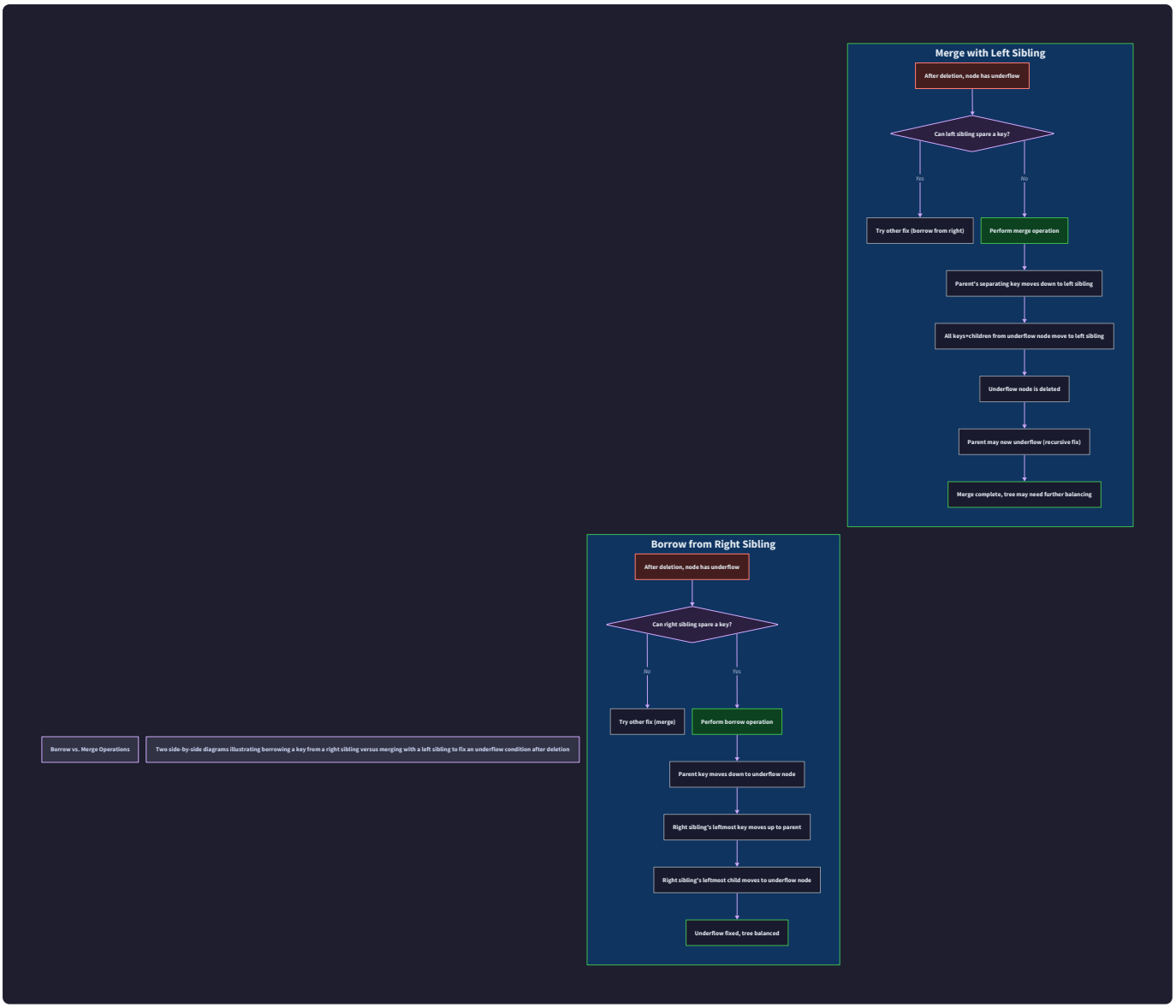
- **Context**: When deleting a key `k` from an internal node, we cannot simply remove it without leaving a "hole" in the tree structure that separates two subtrees. We must replace `k` with another key that maintains the ordering invariants, then delete that replacement key from its original location. The replacement must come from either the left subtree (predecessor) or right subtree (successor) of `k`.
- **Options Considered**:
  1. **Always use in-order predecessor**: Find the largest key in the left subtree by following rightmost pointers until reaching a leaf.
  2. **Always use in-order successor**: Find the smallest key in the right subtree by following leftmost pointers until reaching a leaf.
  3. **Choose based on child capacities**: Check which adjacent child (left or right) has more than the minimum keys, and use that subtree's predecessor/successor accordingly.
  4. **Random selection**: Arbitrarily choose predecessor or successor.
- **Decision**: Option 3—choose based on child capacities. If left child has ≥ t keys, use predecessor; else if right child has ≥ t keys, use successor; otherwise (both have minimum), proceed to merge.
- **Rationale**:
  - **Minimizes restructuring**: Using a child with extra keys avoids immediate merging/borrowing operations. The replacement key can be removed from that child without causing underflow (since it had ≥ t keys initially).
  - **Preserves balance**: This approach tends to keep subtree sizes more balanced than always choosing one side.
  - **Standard textbook approach**: Most B-tree implementations (Cormen et al.) use this capacity-based selection, making our implementation more conventional and easier to verify.
  - **Simplifies implementation**: The logic integrates naturally with the "strengthening" step—we already check child capacities before descending.
- **Consequences**:
  - Slightly more complex logic than always using predecessor.
  - Requires checking both child nodes' key counts before deciding.
  - May lead to more uniform tree degradation over many deletions compared to always favoring one side.

Option	Pros	Cons	Chosen?
Always predecessor	Simpler implementation, consistent behavior	May cause unnecessary merges if left child is at minimum, could lead to left-heavy tree	No
Always successor	Simpler implementation	May cause unnecessary merges if right child is at minimum, could lead to right-heavy tree	No
Capacity-based selection	**Minimizes restructuring, balances tree, standard approach**	**Slightly more logic, two checks needed**	**Yes**
Random selection	Potentially balanced over many operations	Unpredictable, harder to debug, non-deterministic	No

Common Pitfalls: Borrow vs. Merge Logic

⚠️ **Pitfall: Incorrect borrow/merge decision condition**

- **Description**: Checking whether a sibling has `> t-1` keys instead of `≥ t` keys when deciding to borrow. Since the minimum is `t-1`, a sibling with exactly `t-1` keys cannot spare one without becoming underfull.
- **Why it's wrong**: Borrowing from a sibling with only `t-1` keys would leave it with `t-2` keys, violating the minimum occupancy invariant. This corruption may not be detected immediately but will cause future operations to fail.
- **Fix**: Always check `sibling->num_keys ≥ t` before borrowing. For merging, check that both the child and chosen sibling have exactly `t-1` keys.



⚠️ Pitfall: Not handling left/right sibling symmetry

- **Description**: Implementing borrowing from only the left sibling (or only the right), or treating left and right cases with copy-pasted but not properly mirrored logic.
- **Why it's wrong**: When the left sibling doesn't exist or can't lend a key, the right sibling might be able to. Missing this case causes unnecessary merges, increasing tree height prematurely. Incorrect mirroring (e.g., wrong index adjustments) corrupts key ordering.
- **Fix**: Implement both left and right borrowing symmetrically. Use helper functions for rotation operations that take direction as a parameter, or carefully test both directions with symmetric test cases.

⚠️ Pitfall: Forgetting to update parent after borrowing/merging

- **Description**: When borrowing, moving a key between sibling and child but forgetting to update the parent's separating key. When merging, removing the parent's separating key but not shifting the parent's remaining keys and child pointers.
- **Why it's wrong**: The parent's keys no longer correctly separate the subtrees. Searches may fail to find existing keys or descend the wrong path. The tree's ordering invariants are broken.

- **Fix**: In borrowing, remember that the parent key at index `i` moves down to the child, and the sibling key moves up to replace it. In merging, after removing the parent key at index `i`, shift all keys and child pointers from `i+1` left by one position.

⚠️ Pitfall: Not recursing after borrowing/merging

- **Description**: After strengthening a child via borrow or merge, the algorithm proceeds to delete from the original child instead of the potentially changed child (in merge case, the child and sibling become one node).
- **Why it's wrong**: In a merge operation, the child node pointer may become invalid (the child is merged into the sibling). The deletion must continue on the merged node, not the original child.
- **Fix**: After merging child at index `i` with sibling, the merged node resides at the sibling's index. Update your recursion to target this merged node. Keep careful track of which node pointer to use for the recursive call.

⚠️ Pitfall: Ignoring root shrinkage

- **Description**: After deletion, if the root has 0 keys but still has a child, failing to make that child the new root and free the old root.
- **Why it's wrong**: The tree maintains an unnecessary empty root level, wasting memory and adding an extra traversal step for all operations. The height invariant becomes incorrect (height should decrease by 1).
- **Fix**: After the top-level `delete_from_subtree` call returns, check if the root has 0 keys and is not a leaf (has one child). If so, set `tree->root = root->children[0]` and free the old root.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Deletion Algorithm	Recursive implementation with explicit case handling	Iterative implementation with manual stack for potential optimization
Underflow Prevention	Proactive strengthening before descent (as described)	Reactive repair after underflow detected on return
Merge/Borrow Helpers	Separate functions for left/right cases	Unified rotation function with direction parameter

Recommended File/Module Structure:

```
btree/
├── include/
│   └── btree.h           ← Public interface declarations
├── src/
│   ├── btree.c           ← Public API implementations (btree_delete)
│   ├── node.c            ← Node operations (node_remove_key, etc.)
│   ├── delete.c          ← Internal deletion algorithms (delete_from_subtree, helpers)
│   └── validate.c        ← Invariant checking for debugging
└── tests/
    ├── test_delete.c     ← Comprehensive deletion tests
    └── test_invariants.c ← Post-deletion invariant validation
```

Core Logic Skeleton Code:

```c
/* FILE: src/delete.c */

#include <stdlib.h>

#include <stdio.h>

#include "btree.h"

/* Helper: Get the predecessor key from the left subtree */

static int get_predecessor(BTreeNode* node, int index) {

    // TODO 1: Start from the left child at children[index]

    // TODO 2: Traverse rightmost pointers until reaching a leaf

    // TODO 3: Return the last (rightmost) key in that leaf

}

/* Helper: Get the successor key from the right subtree */

static int get_successor(BTreeNode* node, int index) {

    // TODO 1: Start from the right child at children[index + 1]

    // TODO 2: Traverse leftmost pointers until reaching a leaf

    // TODO 3: Return the first (leftmost) key in that leaf

}

/* Helper: Borrow a key from left sibling */

static void borrow_from_left(BTreeNode* parent, int child_idx, int t) {

    BTreeNode* child = parent->children[child_idx];

    BTreeNode* left_sibling = parent->children[child_idx - 1];



    // TODO 1: Shift all keys in child right by one position

    // TODO 2: Shift all child pointers in child right by one (if not leaf)

    // TODO 3: Move parent key at index child_idx-1 down to child's first key position

    // TODO 4: Move left sibling's last key up to parent at index child_idx-1
```

```
    // TODO 5: If not leaves, move left sibling's last child to be child's first child
    // TODO 6: Update key counts: left_sibling->num_keys--, child->num_keys++
}

/* Helper: Borrow a key from right sibling */
static void borrow_from_right(BTreeNode* parent, int child_idx, int t) {
    BTreeNode* child = parent->children[child_idx];
    BTreeNode* right_sibling = parent->children[child_idx + 1];

    // TODO 1: Move parent key at index child_idx down to child's last key position
    // TODO 2: Move right sibling's first key up to parent at index child_idx
    // TODO 3: Shift all keys in right sibling left by one position
    // TODO 4: If not leaves, shift all child pointers in right sibling left by one
    // TODO 5: If not leaves, move right sibling's first child to be child's last child
    // TODO 6: Update key counts: right_sibling->num_keys--, child->num_keys++
}

/* Helper: Merge child at index child_idx with left sibling */
static void merge_with_left_sibling(BTreeNode* parent, int child_idx, int t) {
    BTreeNode* left_sibling = parent->children[child_idx - 1];
    BTreeNode* child = parent->children[child_idx];

    // TODO 1: Move parent key at index child_idx-1 down to left_sibling's keys at position
t-1
    // TODO 2: Copy all keys from child to left_sibling (starting at position t)
    // TODO 3: If not leaves, copy all child pointers from child to left_sibling
    // TODO 4: Update left_sibling->num_keys to 2*t-1
    // TODO 5: Remove parent key at index child_idx-1 by shifting parent keys left
```

```
    // TODO 6: Shift parent child pointers left starting from child_idx
    // TODO 7: Free child node
    // TODO 8: parent->num_keys--
}

/* Helper: Strengthen child at index child_idx (ensure it has at least t keys) */
static void strengthen_child(BTreeNode* parent, int child_idx, int t) {
    BTreeNode* child = parent->children[child_idx];

    // TODO 1: Check if left sibling exists and has >= t keys
    // TODO 2: If yes, borrow_from_left and return
    // TODO 3: Check if right sibling exists and has >= t keys
    // TODO 4: If yes, borrow_from_right and return
    // TODO 5: If left sibling exists, merge_with_left_sibling
    // TODO 6: Else (only right sibling exists), merge child with right sibling
    //         (Note: need to adjust indices for merging with right sibling)
}

/* Main recursive deletion function */
bool delete_from_subtree(BTreeNode* node, int key, int t) {
    int idx = node_find_key_index(node, key, NULL);

    // Case 1: Key is present in this node
    if (idx < node->num_keys && node->keys[idx] == key) {
        if (node->is_leaf) {
            // Subcase 1A: Leaf node
            node_remove_key(node, idx);
            return true;
```

```
        } else {
            // Subcase 1B: Internal node
            // TODO 1: Check if left child (children[idx]) has >= t keys
            // TODO 2: If yes:
            //   - Replace key at idx with predecessor from left subtree
            //   - Recursively delete predecessor from left child
            //   - Return true
            // TODO 3: Else check if right child (children[idx+1]) has >= t keys
            // TODO 4: If yes:
            //   - Replace key at idx with successor from right subtree
            //   - Recursively delete successor from right child
            //   - Return true
            // TODO 5: Else (both children have t-1 keys):
            //   - Merge left child, key, and right child into left child
            //   - Update node's keys and children (remove key at idx, shift)
            //   - Recursively delete key from the merged left child
            //   - Return true
        }
    } else {
        // Case 2: Key is not in this node
        if (node->is_leaf) {
            // Key not found in tree
            return false;
        }

        // Check if we're descending to last possible child
        bool last_child = (idx == node->num_keys);
```

```
        // TODO 1: If the child we're descending to has only t-1 keys:
        //   - Call strengthen_child(node, idx, t)
        //   - If we merged with left sibling, idx may need adjustment

        // TODO 2: Determine which child to descend to after potential merge
        // TODO 3: Recursively call delete_from_subtree on that child
        // TODO 4: Return the result
    }

    return false; // Should not reach here
}

/* Public deletion API */
bool btree_delete(BTree* tree, int key) {
    if (!tree || !tree->root) return false;

    bool deleted = delete_from_subtree(tree->root, key, tree->t);

    // Handle root underflow (root with 0 keys but has a child)
    if (tree->root->num_keys == 0 && !tree->root->is_leaf) {
        // TODO 1: Save old root pointer
        // TODO 2: Set tree->root to the root's only child (children[0])
        // TODO 3: Free the old root
        // TODO 4: Decrease tree height (optional tracking)
    }
```

```
    if (deleted) {
        tree->key_count--;
    }

    return deleted;
}
```

Language-Specific Hints (C):

- **Memory Management**: After merging nodes, remember to `free()` the child node that was merged into its sibling. Use `node_destroy()` if it recursively frees children, but be careful—in a merge, the children are transferred, not destroyed.
- **Array Shifting**: Implement careful `memmove()` operations for shifting keys and child pointers. For example:

```c
// Shift keys left from index i
memmove(&node->keys[i], &node->keys[i+1],
        (node->num_keys - i - 1) * sizeof(int));
// Shift child pointers if internal node
if (!node->is_leaf) {
    memmove(&node->children[i], &node->children[i+1],
            (node->num_keys - i) * sizeof(BTreeNode*));
}
```

- **Debugging**: Write a helper `print_tree(BTreeNode* node, int depth)` that indents based on depth to visualize tree structure before/after deletions.

Milestone Checkpoint:

After implementing deletion, verify correctness with these tests:

1. **Simple deletions**: Insert 10 keys, delete them one by one, verifying `btree_search` fails for deleted keys and `btree_validate` passes after each deletion.
2. **Borrowing scenarios**: Create a tree where a leaf has minimum keys (`t-1`) and its sibling has extra keys. Delete from the minimal leaf, ensuring it borrows correctly without merging.

3. **Merging cascade**: Build a tree where multiple levels are at minimum occupancy. Delete a key that triggers merging up to the root, potentially reducing tree height.
4. **Random sequence**: Generate random insert/delete sequences, validating invariants after each operation with `btree_validate`.

Expected output from `btree_validate` should always return `true` for a correct implementation. Use the visualization from `btree_print` to manually inspect tree structure after complex deletions.

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Segmentation fault after delete	Accessing freed memory or invalid child pointer	Run with Valgrind, add `assert(node->num_keys >= 0)` after deletions	Ensure borrow/merge updates all pointers correctly; don't access merged nodes after free
Search finds deleted keys	Key not actually removed or duplicate keys	Print tree after deletion, check all occurrences of the key	Ensure recursive deletion actually removes the key; handle all cases in `delete_from_subtree`
Tree height doesn't decrease when root merges	Not handling root underflow	Check root's key count after deletion; add debug print	Implement root reduction in `btree_delete` when root has 0 keys and a child
"Minimum occupancy violated" in validate	Borrow/merge logic incorrect	Run `btree_validate` after each operation to catch which node violates	Check borrow threshold (≥ t keys, not > t-1); ensure merge happens when both nodes have t-1 keys
Infinite recursion	Child index not adjusted after merge	Print node and child indices at each recursive call	When merging with left sibling, the target child index decreases by 1 for subsequent recursion

9. Interactions and Data Flow

> **Milestone(s):** Milestones 2, 3, and 4 (Search, Insert with Split, Delete with Rebalancing)

This section describes how the architectural components defined earlier collaborate to perform the core B-tree operations. While previous sections focused on individual components in isolation, here we examine their dynamic interactions—the function call sequences, data flow patterns, and state transitions that occur during search, insertion, and deletion. Understanding these interactions is crucial for debugging and for grasping how the B-tree maintains its invariants across complex operations.

At the highest level, the B-tree orchestrator (`BTree`) serves as the entry point for all operations, while the recursive operation functions (`search_recursive`, `insert_non_full`, `delete_from_subtree`) navigate the tree structure, and the node manipulation functions (`node_split_child`, `borrow_from_left`, etc.) perform localized transformations. The interactions follow a consistent pattern: start at the root, traverse downward while potentially transforming nodes, perform the target operation at the leaf or internal node, then propagate changes upward through the recursion stack.

Sequence of Operations

Each core operation follows a specific sequence of function calls that reflects the recursive nature of B-tree algorithms. The sequences below trace through concrete examples to illustrate typical interactions.

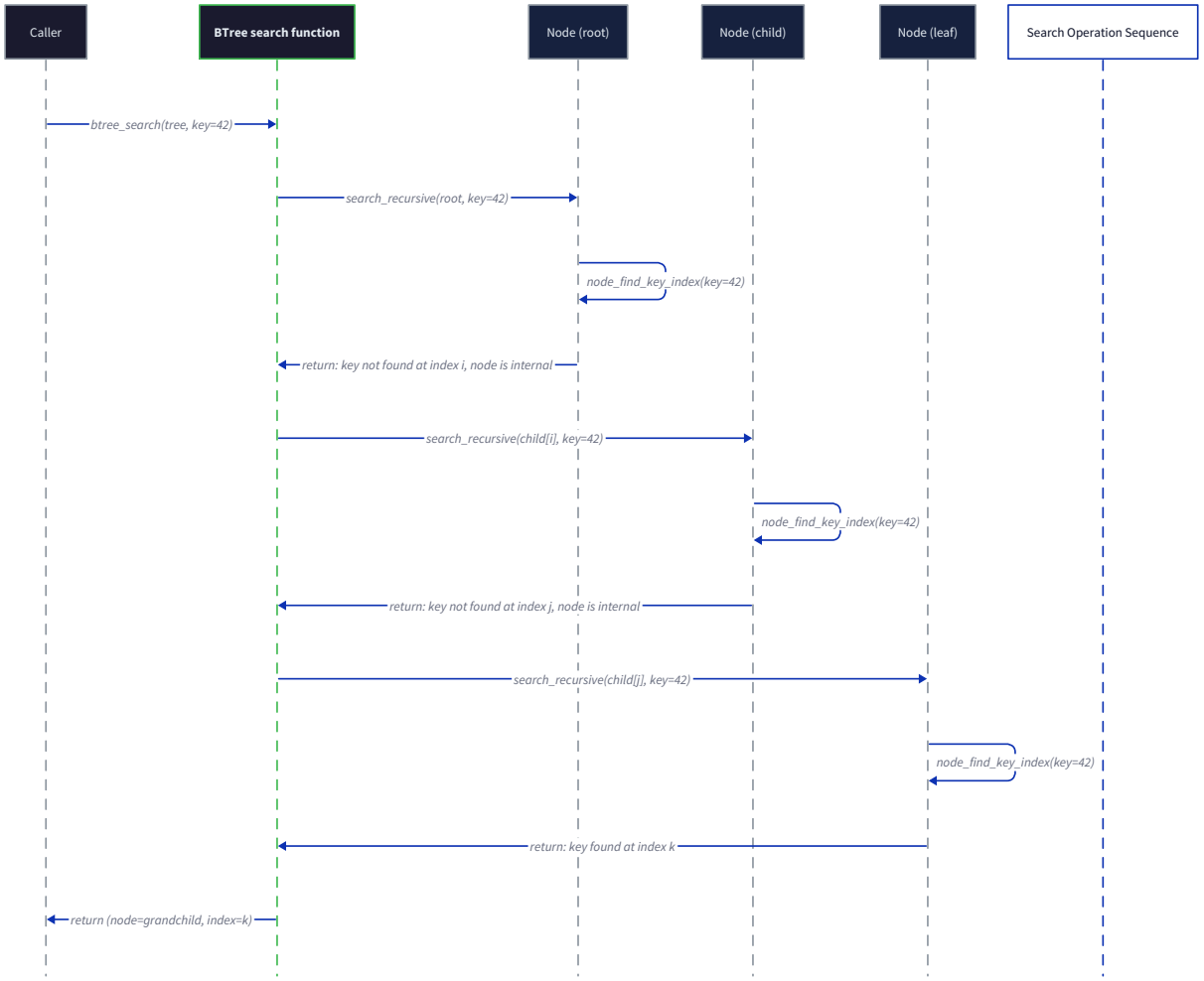
Search Operation Sequence

Search demonstrates the simplest interaction pattern—pure traversal without modification. Consider searching for key 42 in a B-tree with minimum degree `t = 3` and height 3.

The sequence begins with `btree_search(tree, 42)`, which immediately calls `search_recursive(tree->root, 42)`. The recursive search follows this pattern at each node:

1. **Binary Search Within Node**: At the current node, `node_find_key_index` performs binary search on the node's keys array, comparing the target key (42) against each key. This returns either the index where 42 was found or the index of the child subtree where 42 should be located.
2. **Key Found vs. Continue Search**: If the key is found in the current node, the recursion terminates and returns a `SearchResult` with `found = true` and the associated value. If not found and the current node is a leaf, the recursion terminates with `found = false`. If not found in an internal node, the function recurses on the appropriate child.
3. **Recursive Descent**: Before recursing, the function validates that the child pointer at the computed index is not `NULL` (for internal nodes). The recursion depth increases by one as control passes to the child node.

The following diagram illustrates this recursive descent through three levels of the tree:



Concrete Walkthrough Example: Suppose our B-tree has the following structure (keys shown, `t = 3`):

- Root: [30, 60] with 3 children
- Child 0 (keys < 30): [10, 20]
- Child 1 (keys between 30-60): [40, 50]
- Child 2 (keys > 60): [70, 80]

Searching for key 42 proceeds as:

1. `btree_search(tree, 42)` → `search_recursive(root, 42)`
2. At root node: `node_find_key_index` compares 42 with [30, 60], determines 42 is between 30 and 60, returns child index 1
3. Since key not found at root and root is internal, recurse: `search_recursive(root->children[1], 42)`
4. At child 1 node: `node_find_key_index` compares 42 with [40, 50], determines 42 < 40, returns child index 0
5. Recurse: `search_recursive(child1->children[0], 42)`

6. At leaf node: `node_find_key_index` finds no match in empty leaf (or in keys [35, 41] if present), returns index 0
7. Since leaf and key not found, return `SearchResult{found = false, value = NULL, comparisons = total}`

The recursion unwinds, propagating the not-found result back to the original caller.

Insertion with Splitting Sequence

Insertion involves more complex interactions because it may transform the tree structure. The sequence employs **proactive splitting** during the downward traversal to ensure that when we reach the target leaf, neither it nor any ancestor is full. Consider inserting key 25 into the same B-tree with `t = 3`, where the root is currently full (has 5 keys).

The interaction sequence follows:

1. **Public API Entry**: `btree_insert(tree, 25, value_ptr)` is called. The function first checks if the root is full using `node_is_full(root, t)`.
2. **Root Split (Special Case)**: If the root is full, the tree height must increase:
   - Create a new root node using `node_create(t, false)` (internal node)
   - Set the old root as the new root's first child (index 0)
   - Call `node_split_child(new_root, 0, t)` to split the old root
   - Update the tree's root pointer to the new root
   - Call `insert_non_full(new_root, 25, value_ptr, t)` to continue insertion
3. **Proactive Splitting During Descent**: `insert_non_full` implements the core recursive algorithm:
   - At each node, use `node_find_key_index` to determine the appropriate child index
   - Before descending to that child, check if the child is full using `node_is_full(child, t)`
   - If full, call `node_split_child(current_node, child_index, t)` to split the child
   - This splitting may change which child index we should descend to (if the promoted median key affects the comparison)
4. **Leaf Insertion**: When reaching a leaf node (identified by `node->is_leaf == true`), insert the key-value pair directly using `node_insert_key` at the position determined by `node_find_key_index`.
5. **Recursive Unwind**: As the recursion unwinds, no further action is needed because all splitting occurred proactively on the way down.

Concrete Walkthrough with Root Split: Starting with a full root [10, 20, 30, 40, 50] (t=3, max keys=5), inserting key 25:

1. `btree_insert` finds root full → creates new root `new_root`
2. `node_split_child(new_root, 0, t)` splits old root:

- Left child gets [10, 20], right child gets [40, 50]
- Median key 30 promoted to `new_root.keys[0]`
- Tree height increases from 1 to 2

3. `insert_non_full(new_root, 25, value_ptr, t)` begins:
   - At `new_root` (keys=[30]): `node_find_key_index` finds 25 < 30 → child index 0
   - Child 0 (keys=[10,20]) is not full → descend recursively
4. `insert_non_full(child0, 25, value_ptr, t)`:
   - Child0 is leaf → `node_find_key_index` finds insertion position between 20 and end
   - `node_insert_key` inserts 25, resulting in keys [10, 20, 25]
5. Final tree: root [30] with children [10,20,25] and [40,50]

Deletion with Rebalancing Sequence

Deletion has the most complex interaction pattern due to the need to handle underflow through borrowing or merging, which may cascade upward. Consider deleting key 30 from a B-tree where the target node and its siblings are at minimum capacity (`t-1` keys).

The deletion sequence involves multiple helper functions and potentially cascading rebalancing:

1. **Public API Entry**: `btree_delete(tree, 30)` calls `delete_from_subtree(root, 30, t)`.
2. **Traversal to Target Key**: `delete_from_subtree` recursively navigates to the node containing key 30, but with a crucial difference from insertion: it proactively **strengthens** nodes before descending to ensure we don't descend into an underfull node (except for the special case of the root).
3. **Strengthening Before Descent**: At each node before recursing to a child:
   - If the child has exactly `t-1` keys (minimum), call `strengthen_child(parent, child_index, t)`
   - `strengthen_child` attempts to:
     1. Borrow from left sibling if it has at least `t` keys (using `borrow_from_left`)
     2. Otherwise, borrow from right sibling if it has at least `t` keys (using `borrow_from_right`)
     3. Otherwise, merge with a sibling (using `merge_with_left_sibling` or equivalent for right)
4. **Key Location and Deletion Type**: Once the key is located in a node:
   - **Case A: Key in leaf**: Direct removal using `node_remove_key`
   - **Case B: Key in internal node**: Replace with predecessor (using `get_predecessor`) or successor, then recursively delete that predecessor/successor from the leaf
   - After deletion, check if the node is now underfull (but note: strengthening should prevent this except at root)
5. **Root Shrinking**: After deletion completes, check if the root has 0 keys but has a child (height > 1). If so, make that child the new root and free the old root.

6. **Cascading Effects**: Borrowing and merging operations modify parent nodes, which may themselves become underfull, potentially causing the underflow condition to propagate upward.

Concrete Walkthrough with Borrowing: Deleting key 30 from a node with keys [30, 40] (t=3, min=2) that has left sibling [10, 20, 25] and right sibling [50, 60]:

1. `delete_from_subtree` locates node containing 30
2. Since node has exactly `t-1` keys (2), parent calls `strengthen_child` before deletion
3. `strengthen_child` checks left sibling (3 keys ≥ t) → chooses `borrow_from_left`
4. `borrow_from_left`:
   - Moves largest key from left sibling (25) up to parent
   - Moves parent's separator key (say, 28) down to target node
   - Adjusts child pointers if nodes are internal
   - Result: target node now has [28, 30, 40] (3 keys)
5. Now safe to delete 30 from the strengthened node
6. After deletion: target node has [28, 40] (2 keys, still ≥ t-1)

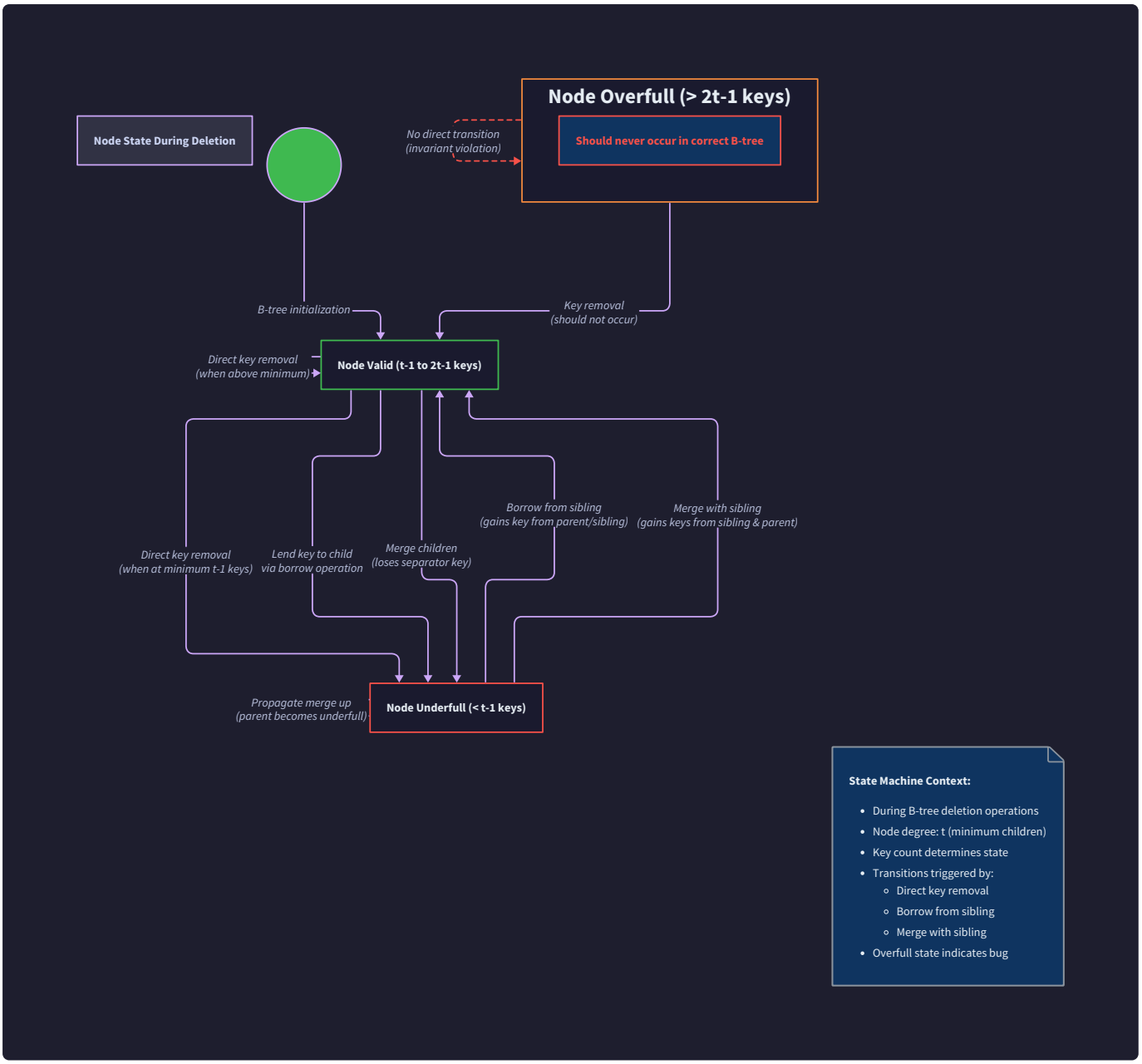
Concrete Walkthrough with Merging: Same scenario but left sibling has only 2 keys (minimum):

1. `strengthen_child` cannot borrow from left (only 2 keys)
2. Check right sibling: also has only 2 keys
3. Choose to merge with left sibling using `merge_with_left_sibling`
4. Merge operation:
   - Bring down parent's separator key (28) into left sibling
   - Move all keys from target node to left sibling
   - Adjust child pointers
   - Free the now-empty target node
   - Parent loses one key and one child pointer
5. Parent may now be underfull, requiring recursive strengthening upward

State Transitions During Operations

Each B-tree node transitions through different states defined by its key count relative to the capacity bounds. The table below captures how the four fundamental operations—direct insertion, direct deletion, borrowing, and merging—affect a node's key count, along with the conditions that trigger each transition.

The state machine for a node during deletion operations is visualized in the following diagram:


Node State During Deletion
Node Overfull (> 2t-1 keys)
Should never occur in correct B-tree
No direct transition
(invariant violation)
B-tree initialization
Key removal
(should not occur)
Direct key removal
(when above minimum)
Node Valid (t-1 to 2t-1 keys)
Borrow from sibling
(gains key from parent/sibling)
Merge with sibling
(gains keys from sibling & parent)
Direct key removal
(when at minimum t-1 keys)
Lend key to child
via borrow operation
Merge children
(loses separator key)
Propagate merge up
(parent becomes underfull)
Node Underfull (< t-1 keys)
State Machine Context:
During B-tree deletion operations
Node degree: t (minimum children)
Key count determines state
Transitions triggered by:
Direct key removal
Borrow from sibling
Merge with sibling
Overfull state indicates bug

Current State	Event	Next State	Action Taken	Conditions & Notes
Valid (t-1 to 2t-1 keys)	**Insert key** (no split)	**Valid** or **Overfull**	`node_insert_key` adds key at sorted position	If node had 2t-2 keys before insert → becomes Overfull (2t-1 keys). Overfull state is temporary and immediately triggers a split if detected during insertion descent.
Overfull (2t-1 keys)	**Split operation**	Two **Valid** nodes	`node_split_child` creates two nodes with t-1 keys each, promotes median to parent	Parent gains one key. This is the only way a node becomes Overfull—when a child is split and the parent receives the promoted key while already having 2t-2 keys.
Valid	**Delete key** (direct removal)	**Valid** or **Underfull**	`node_remove_key` removes key, shifts remaining keys left	If node had exactly t-1 keys before deletion → becomes Underfull (< t-1 keys). This should not happen in correctly implemented deletion due to proactive strengthening.
Underfull (< t-1 keys)	**Borrow from left sibling**	**Valid**	`borrow_from_left` takes largest key from left sibling via parent rotation	Left sibling must have ≥ t keys. Parent loses no keys (one moves down, another moves up). Right sibling borrowing is symmetric.
Underfull	**Borrow from right sibling**	**Valid**	`borrow_from_right` takes smallest key from right sibling via parent rotation	Right sibling must have ≥ t keys. Parent loses no keys (one moves down, another moves up).
Underfull	**Merge with left sibling**	**Valid** (merged node)	`merge_with_left_sibling` combines underfull node, left sibling, and parent separator	Both siblings have exactly t-1 keys. New merged node has 2t-2 keys (valid). Parent loses one key and one child pointer.
Underfull (root)	**Root shrinking**	**Valid** (new root) or	Root with 0 keys and 1 child becomes that child; old root freed	Only occurs when tree height decreases. New root

Current State	Event	Next State	Action Taken	Conditions & Notes
		Empty		may be Underfull if it has < t-1 keys but root has relaxed minimum.

Key Transition Insights:

> The most critical insight is that **nodes only become underfull during direct key removal**, and this should only happen at the leaf where the key is physically deleted or at internal nodes where a predecessor/successor is deleted. All other nodes are proactively strengthened before descent, ensuring that recursion never enters an underfull node (except the special case of the root).

State Transition Example Walkthrough: Consider a node with exactly `t-1` keys (the minimum for a non-root node):

- If we attempt to delete a key from this node, it would become underfull (violating the invariant).
- Therefore, before descending to delete from this node, its parent calls `strengthen_child`, which will either:
  1. **Borrow** from a sibling with ≥ t keys → node becomes valid (t keys)
  2. **Merge** with a sibling that also has t-1 keys → both combine into one valid node (2t-2 keys)
- After strengthening, the node has ≥ t keys, so deletion can proceed safely without causing underflow.

The parent node itself may become underfull after lending a key during borrowing or after losing a key during merging. This is why deletion may require recursive strengthening that propagates upward toward the root.

Implementation Guidance

> **Technology Note**: This implementation guidance focuses on the C programming language, which provides direct memory control essential for understanding B-tree's disk-oriented design. The guidance bridges the conceptual sequences described above with concrete function implementations.

A. Recommended Sequence Implementation Patterns

For each operation, follow these calling patterns in your implementation:

Search Pattern:

```c
SearchResult btree_search(BTree* tree, int key) {
    // Public API: simply call the recursive helper
    return search_recursive(tree->root, key);
}

SearchResult search_recursive(BTreeNode* node, int key) {
    // Base case: empty tree (root is NULL)
    if (node == NULL) {
        SearchResult result = {false, NULL, 0};
        return result;
    }

    // Find position within current node
    int comparisons = 0;
    int i = node_find_key_index(node, key, &comparisons);

    if (i < node->num_keys && node->keys[i] == key) {
        // Key found in this node
        SearchResult result = {true, node->values[i], comparisons};
        return result;
    } else if (node->is_leaf) {
        // Key not found and we're at a leaf
        SearchResult result = {false, NULL, comparisons};
        return result;
    } else {
        // Recurse to appropriate child
        return search_recursive(node->children[i], key);
```

```
    }

}
```

Insert Pattern (simplified structure showing call sequence):

```c
bool btree_insert(BTree* tree, int key, void* value) {
    // Handle root split first if needed
    if (node_is_full(tree->root, tree->t)) {
        BTreeNode* new_root = node_create(tree->t, false);
        new_root->children[0] = tree->root;
        node_split_child(new_root, 0, tree->t);
        tree->root = new_root;
    }

    // Now insert into non-full tree
    return insert_non_full(tree->root, key, value, tree->t);
}

bool insert_non_full(BTreeNode* node, int key, void* value, int t) {
    int i = node->num_keys - 1;

    if (node->is_leaf) {
        // Insert directly into leaf
        // TODO: Find insertion position and insert
        return true;
    } else {
        // Find child to descend to
        // TODO: Find child index i

        // Proactive split if child is full
        if (node_is_full(node->children[i], t)) {
            node_split_child(node, i, t);
```

```
            // After split, need to re-check which child to descend to
            // because the promoted key might affect the comparison
            // TODO: Update i if needed
        }

        // Recurse to child
        return insert_non_full(node->children[i], key, value, t);
    }
}
```

Delete Pattern (showing strengthening logic):

```c
bool delete_from_subtree(BTreeNode* node, int key, int t) {
    // This is a simplified version - actual implementation handles all cases

    // Find key position
    int idx = 0;
    while (idx < node->num_keys && key > node->keys[idx]) {
        idx++;
    }

    // Case 1: Key found in this node
    if (idx < node->num_keys && node->keys[idx] == key) {
        if (node->is_leaf) {
            // Case 1A: Leaf node
            node_remove_key(node, idx);
        } else {
            // Case 1B: Internal node
            // TODO: Replace with predecessor/successor and recurse
        }
        return true;
    } else {
        // Case 2: Key not in this node
        if (node->is_leaf) {
            return false; // Key not found
        }

        // Ensure child has at least t keys before descending
        if (node_is_underfull(node->children[idx], t)) {
```

```
            strengthen_child(node, idx, t);
        }

        // Recurse to child
        return delete_from_subtree(node->children[idx], key, t);
    }
}
```

B. Interaction Debugging Helpers

To visualize the interactions during development, add these debugging helpers to trace function calls:

```c
// Enable with compile flag -DDEBUG_INTERACTIONS
#ifdef DEBUG_INTERACTIONS
#define TRACE_ENTER(func, node, key) \
    printf("[TRACE] ENTER %s: node=%p, key=%d, is_leaf=%d, num_keys=%d\n", \
           func, (void*)node, key, node ? node->is_leaf : -1, node ? node->num_keys : -1)

#define TRACE_EXIT(func, result) \
    printf("[TRACE] EXIT  %s: result=%d\n", func, result)

#define TRACE_SPLIT(parent, child_idx) \
    printf("[TRACE] SPLIT: parent=%p, child_idx=%d\n", (void*)parent, child_idx)

#define TRACE_BORROW(parent, child_idx, direction) \
    printf("[TRACE] BORROW: parent=%p, child_idx=%d, from=%s\n", \
           (void*)parent, child_idx, direction)
#else
#define TRACE_ENTER(func, node, key)
#define TRACE_EXIT(func, result)
#define TRACE_SPLIT(parent, child_idx)
#define TRACE_BORROW(parent, child_idx, direction)
#endif
```

Use these in your functions:

```c
SearchResult search_recursive(BTreeNode* node, int key) {
    TRACE_ENTER("search_recursive", node, key);

    // ... implementation ...

    TRACE_EXIT("search_recursive", result.found);
    return result;
}
```

C. Common Interaction Bugs and Fixes

Symptom	Likely Cause	How to Diagnose	Fix
Infinite recursion during insertion	Forgetting to update child index after splitting in `insert_non_full`	Add trace logging before and after split; check if i needs adjustment when promoted key > insertion key	After `node_split_child`, compare key with promoted key; if key > promoted key, increment child index
Key disappears after deletion	Merging operation incorrectly handles child pointers	Use `btree_print` after each merge; verify all keys from both siblings and parent separator are present	In `merge_with_left_sibling`, ensure: 1) Copy parent separator, 2) Copy all keys from right node, 3) Copy all child pointers if internal
Segmentation fault during search	Child pointer is NULL when node claims it's internal	Add assertion: `assert(node->is_leaf	
Tree height grows unnecessarily	Splitting root when it's not full	Check `node_is_full` condition at root; might be off-by-one error	Verify `node_is_full` returns true only when `node->num_keys == 2*t - 1`
Borrowing makes sibling underfull	Not checking sibling has ≥ t keys before borrowing	Add precondition check in borrow functions	In `strengthen_child`, check sibling key count before attempting borrow

D. Testing Interactions

Create interaction tests that verify the function call sequences:

```c
// test_interactions.c

void test_insertion_sequence() {

    BTree* tree = btree_create(3);



    // Insert until root splits

    for (int i = 1; i <= 5; i++) {

        btree_insert(tree, i, NULL);

    }

    // At this point, root should have 1 key (median of 1..5 = 3)

    assert(tree->root->num_keys == 1);

    assert(tree->root->keys[0] == 3);



    // Insert more to cause child splits

    for (int i = 6; i <= 10; i++) {

        btree_insert(tree, i, NULL);

    }

    // Verify structure through search

    for (int i = 1; i <= 10; i++) {

        SearchResult r = btree_search(tree, i);

        assert(r.found);

    }



    btree_destroy(tree);

}


void test_deletion_cascade() {

    BTree* tree = btree_create(3);
```

```
    // Build a specific tree structure that will cascade on deletion
    // Insert keys in order that creates minimum-occupancy nodes
    int keys[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    for (int i = 0; i < 10; i++) {
        btree_insert(tree, keys[i], NULL);
    }

    // Delete a key that will trigger merge cascade
    bool deleted = btree_delete(tree, 10);
    assert(deleted);

    // Verify tree still valid
    assert(btree_validate(tree));

    btree_destroy(tree);
}
```

Run these tests frequently during development to ensure the component interactions remain correct as you implement each operation.

10. Error Handling and Edge Cases

> **Milestone(s):** Milestones 1-4 (all operations)

A robust B-tree implementation must handle edge cases gracefully and provide mechanisms for detecting invariant violations. Since this is an educational implementation focused on algorithmic correctness rather than production resilience, we adopt a pragmatic approach: fail loudly on unrecoverable errors (like memory exhaustion) while maintaining data structure invariants for all valid operations. This section catalogs the failure modes specific to B-tree operations and provides strategies for both runtime handling and development-time verification.

10.1 Failure Modes and Edge Cases

B-tree operations involve complex pointer manipulation, dynamic memory allocation, and intricate balancing logic. Understanding potential failure points is crucial for building a correct implementation. The table below enumerates common failure scenarios, their detection methods, and recommended handling strategies for this in-memory implementation.

Failure Mode	Detection Strategy	Recovery Approach	Rationale
Memory allocation failure (malloc returns NULL)	Check return value of all allocation calls (`malloc` , `calloc` , `realloc`)	Propagate failure upward by returning error indicator (NULL/false); clean up any partially allocated resources	Memory exhaustion is unrecoverable for this in-memory implementation; failing gracefully prevents undefined behavior from null pointer dereferences
Duplicate key insertion	Search operation during insertion path; binary search within each node	Treat as no-op (do nothing) or optionally update value if storing key-value pairs; return `false` to indicate no new key was added	B-trees typically support unique keys; silently ignoring duplicates maintains set semantics and simplifies deletion logic
Deletion of non-existent key	Search during deletion path fails to find key	Return `false` to indicate key was not present; tree remains unchanged	Idempotent operation: deleting a missing key should not alter the tree structure or cause errors
Invalid minimum degree (t < 2)	Validate parameter in `btree_create()`	Return NULL or use `DEFAULT_MIN_DEGREE` (3)	B-tree definition requires t ≥ 2 to ensure nodes can split (2t-1 ≥ 3 when t=2); lower values violate core invariants
Negative key values (if domain restricted)	Validate input in public API functions	Return error or ignore (implementation choice)	While B-trees can handle any comparable keys, some use cases may restrict domains; validation prevents logical errors

Failure Mode	Detection Strategy	Recovery Approach	Rationale
Node underflow during deletion	`node_is_underfull()` check after key removal	Trigger borrowing or merging via `strengthen_child()` before recursion	Proactive rebalancing maintains the t-1 minimum key invariant for all non-root nodes
Node overflow during insertion	`node_is_full()` check before insertion	Trigger splitting via `node_split_child()` during descent	Proactive splitting ensures we never insert into a full node, maintaining the 2t-1 maximum invariant
Invalid child index in split/borrow/merge	Range checks (0 ≤ i < parent->num_keys+1)	Assert during development; graceful failure with error return in production	These are internal functions; indices should always be valid if calling code maintains invariants
Corrupted node pointers (dangling/recycled)	Cannot detect at runtime without memory tagging	Use `btree_validate()` during testing; employ Valgrind for memory error detection	Pointer corruption indicates serious bugs in memory management; prevention via careful ownership discipline is essential

> **Design Insight:** For educational implementations, **failing fast with clear diagnostics** is preferable to complex recovery mechanisms. Use assertions (`assert()`) liberally during development to catch invariant violations immediately, then replace with graceful error returns if building for production use.

Beyond these failure modes, several edge case scenarios require special consideration in the algorithm logic:

Root Node Special Cases:

1. **Empty Tree Deletion:** Attempting to delete from an empty tree (root is NULL or root->num_keys == 0) should return `false` without error.
2. **Root Underflow:** The root is exempt from the minimum key requirement (can have as few as 1 key when not a leaf). However, if the root becomes empty after deletion (0 keys), its single child should become the new root, decreasing tree height.
3. **Root Split:** Splitting a full root is the only operation that increases tree height. The old root becomes a child of the new root.

Leaf Node Boundary Conditions:

1. **First/Last Key Deletion:** When removing the smallest or largest key from a leaf, binary search indices must be handled correctly to avoid off-by-one errors when shifting arrays.
2. **Leaf with Minimum Keys:** A leaf with exactly t-1 keys that loses a key becomes underfull and requires rebalancing, even though this violates the invariant only after the operation completes.

Internal Node Edge Cases:

1. **Predecessor/Successor Selection:** When deleting a key from an internal node, if the chosen subtree (predecessor's left or successor's right) has exactly t-1 keys, the `strengthen_child()` function must be called before recursion to ensure we don't descend into an underfull node.
2. **Leftmost/Rightmost Child:** When borrowing or merging with the leftmost or rightmost child of a parent, only one sibling exists, requiring asymmetric handling.

Algorithm-Specific Edge Cases:

For Insertion:

- Inserting into a tree where all nodes along the path are full requires splitting at every level from root to leaf.
- Inserting a key that would become the new minimum or maximum of the entire tree must maintain sorted order across all nodes.

For Deletion:

- The "borrow or merge" decision when a node becomes underfull: always attempt to borrow from an immediate sibling with > t-1 keys; only merge if both siblings have exactly t-1 keys.
- When merging reduces the parent's key count, the parent may become underfull, requiring recursive rebalancing upward.
- Deleting the only key in a root that is a leaf results in an empty tree.

For Search:

- Searching in an empty tree should return a `SearchResult` with `found = false` and `comparisons = 0`.
- Searching for a key smaller than all keys or larger than all keys should traverse to the leftmost or rightmost leaf respectively.

Memory Management Edge Cases:

1. **Destroying an Empty Tree:** `btree_destroy(NULL)` or `btree_destroy()` on an empty but allocated tree should be safe (no-op).
2. **Node Child Pointer Cleanup:** When merging nodes, the right node's children array must be copied before the node is freed to prevent memory leaks.
3. **Array Shifting Operations:** `node_insert_key()` and `node_remove_key()` must correctly handle the trailing elements when inserting at or removing from the beginning, middle, or end of the arrays.

> **Mental Model: The Building Inspector Analogy** Think of error handling as a building inspector visiting a B-tree "building." The inspector checks foundational invariants (minimum/maximum keys, sorted order, pointer consistency) much like checking a building's structural integrity. Some issues are critical (memory corruption = foundation cracks) requiring immediate shutdown, while others are maintenance items (underfull nodes = slightly uneven floors) that can be fixed on the spot with borrowing or merging tools. The `btree_validate()` function is the inspector's comprehensive checklist.

10.2 Invariant Verification for Debugging

Implementing a B-tree involves complex pointer manipulations and subtle boundary conditions that can easily introduce bugs. A comprehensive **invariant verification function** (`btree_validate()`) is an indispensable debugging tool that programmatically checks all B-tree rules, providing immediate feedback when an operation corrupts the data structure. This function should be called after each insert/delete during development and before returning from any public API in test builds.

Core Invariants to Verify:

1. **Structural Invariants:**
   - **Root Invariant:** The root has between 1 and 2t-1 keys (unless the tree is empty, where root may be NULL or have 0 keys).
   - **Node Capacity Invariant:** Every non-root node has between t-1 and 2t-1 keys.
   - **Leaf Child Invariant:** All leaf node child pointers are NULL.
   - **Internal Node Child Invariant:** Every internal node has exactly `num_keys + 1` non-NULL child pointers.
   - **Tree Height Invariant:** All leaf nodes are at the same depth (same distance from root).
2. **Ordering Invariants:**
   - **Intra-node Ordering:** Keys within each node are stored in strictly ascending order (no duplicates).
   - **Inter-node Ordering:** For any internal node with key `K_i` and children `C_i` and `C_{i+1}`:
     - All keys in subtree `C_i` are less than `K_i`
     - All keys in subtree `C_{i+1}` are greater than `K_i`
   - **Global Ordering:** An in-order traversal of the tree yields keys in strictly ascending order.
3. **Pointer Integrity Invariants:**
   - **No Cycles:** The tree is a directed acyclic graph (no node is its own ancestor).
   - **Parent Consistency:** If parent pointers were implemented (not required in our design), each child would reference its correct parent.
   - **Memory Bounds:** All array accesses are within allocated bounds (`0 ≤ index < num_keys` for keys, `0 ≤ index ≤ num_keys` for children).

Implementation Strategy for `btree_validate()`:

The validation function performs a recursive depth-first traversal, checking both local node properties and global relationships. It returns `true` if all invariants hold, `false` otherwise, and can optionally print detailed error messages to stderr.

```c
// Pseudo-code for validation logic (not actual implementation code)
bool validate_node(BTreeNode* node, int t, bool is_root,
                   int* min_key, int* max_key, int depth, int* leaf_depth) {
    // Check 1: Node pointer validity
    if (node == NULL) return is_root; // NULL only allowed for empty tree root

    // Check 2: Key count bounds
    if (is_root) {
        assert(0 <= node->num_keys && node->num_keys <= 2*t-1);
    } else {
        assert(t-1 <= node->num_keys && node->num_keys <= 2*t-1);
    }

    // Check 3: Key ordering within node
    for (int i = 0; i < node->num_keys - 1; i++) {
        assert(node->keys[i] < node->keys[i+1]); // Strictly ascending
    }

    // Check 4: Child pointer consistency
    if (node->is_leaf) {
        assert(all children are NULL);
        // Record leaf depth for height uniformity check
        if (*leaf_depth == -1) *leaf_depth = depth;
        else assert(depth == *leaf_depth);
    } else {
        assert(node->num_keys + 1 == number of non-NULL children);
        // Recursively validate children with key range checks
```

```
        for (int i = 0; i <= node->num_keys; i++) {
            int child_min, child_max;
            bool child_valid = validate_node(node->children[i], t, false,
                                             &child_min, &child_max,
                                             depth + 1, leaf_depth);
            assert(child_valid);
            // Check subtree key bounds relative to parent keys
            if (i > 0) assert(child_max < node->keys[i-1]);
            if (i < node->num_keys) assert(child_min > node->keys[i]);
        }
    }

    // Update min/max keys for parent checks
    *min_key = (node->num_keys > 0) ? node->keys[0] : INT_MAX;
    *max_key = (node->num_keys > 0) ? node->keys[node->num_keys-1] : INT_MIN;

    return true;
}
```

Common Validation Failures and Their Likely Causes:

Validation Failure	Likely Bug Location	Debugging Approach
Key count out of bounds	Insertion: forgot to split full nodes Deletion: incorrect borrow/merge logic	Add debug prints before/after each operation; check `node_is_full()` / `node_is_underfull()` logic
Keys not sorted within node	`node_insert_key()` or `node_remove_key()` with incorrect shift logic	Test array insertion/removal in isolation; verify binary search returns correct index
Leaf node has non-NULL children	`node_create()` initialization error or corruption during split	Ensure `is_leaf` flag set correctly; check `node_split_child()` child pointer allocation
Internal node has wrong child count	Split/merge operations miscount children	Verify that when promoting median key, we also adjust child pointer counts
Leaf depths unequal	Insertion: root split height increase incorrect Deletion: root shrinkage incorrect	Trace height changes; ensure all insertions start at same root level
Subtree key range violation	Binary search returns wrong child index; incorrect predecessor/successor selection	Add subtree range assertions in search path; verify in-order traversal
Memory access violation	Array index out of bounds in `keys[]` or `children[]`	Use `assert(index >= 0 && index < capacity)` before each array access

Integration with Testing:

The `btree_validate()` function should be integrated into the test harness:

1. **After every operation** in unit tests, call `btree_validate()` and assert it returns true.
2. **In stress tests** (thousands of random insertions/deletions), periodically validate the tree.
3. **As a debugging aid**, when a test fails, call `btree_print()` followed by `btree_validate()` with verbose error output to identify the first violated invariant.

> **ADR: Comprehensive Validation vs. Performance**
>
> **Context:** During development, we need to detect invariant violations immediately, but runtime validation has performance costs.
>
> **Options Considered:**
>
> 1. **Full validation after every public API call** (enabled only in debug builds via `#ifdef DEBUG`)
> 2. **No built-in validation** (rely entirely on external test suite)
> 3. **Selective validation** (check only the specific invariants relevant to the operation)
>
> **Decision:** Implement full validation gated by `#ifdef DEBUG` with optional verbose output.
>
> **Rationale:** B-tree bugs can manifest far from their source (e.g., a split error might only cause problems many operations later). Immediate detection with stack trace is invaluable for learning. The compile-time switch eliminates performance impact in release builds.
>
> **Consequences:** Developers must remember to test with debug builds; validation code must itself be bug-free.

Debugging Workflow:

When a test fails or invariant violation is detected:

1. **Isolate the failing operation** using a minimal reproducible test case.
2. **Add verbose logging** to trace the operation's execution path.
3. **Call `btree_print()`** to visualize the tree structure before and after the failing operation.
4. **Use `btree_validate()` with error messages** to identify the first violated invariant.
5. **Check the corresponding operation's logic** against the algorithm steps in Sections 6-8.

> **Key Insight:** The combination of `btree_print()` (visualization) and `btree_validate()` (invariant checking) forms a powerful debugging suite that can catch ~90% of B-tree implementation bugs without needing a debugger.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Error Detection	Return `false` / `NULL` with error messages to `stderr`	Error codes enum with descriptive strings
Invariant Checking	`assert()` macro from `<assert.h>`	Custom validation with detailed error logging
Memory Debugging	Valgrind for leak detection	AddressSanitizer (`-fsanitize=address`)

Recommended File Structure:

```
btree/
├── include/
│   └── btree.h           # Public API declarations
├── src/
│   ├── btree.c           # Main implementation
│   ├── btree_validate.c  # Invariant checking (debug only)
│   └── btree_debug.c     # Printing utilities
└── tests/
    ├── test_btree.c      # Unit tests
    └── stress_test.c     # Randomized operations
```

Infrastructure Starter Code (Complete Validation Function):

```c
// btree_validate.c - Complete implementation

#include "btree.h"

#include <assert.h>

#include <stdio.h>

#include <limits.h>


#ifdef DEBUG_VALIDATE


// Helper for recursive validation

static bool _validate_node(BTreeNode* node, int t, bool is_root,

                           int* min_key, int* max_key,

                           int depth, int* leaf_depth, int* error_count) {

    // Check node exists (NULL only allowed for empty root)

    if (node == NULL) {

        if (is_root) return true;

        fprintf(stderr, "ERROR: Non-root node is NULL\n");

        (*error_count)++;

        return false;

    }



    // Check key count bounds

    if (is_root) {

        if (node->num_keys < 0 || node->num_keys > 2*t-1) {

            fprintf(stderr, "ERROR: Root has %d keys, allowed [0, %d]\n",

                    node->num_keys, 2*t-1);

            (*error_count)++;

            return false;

        }
```

```
    } else {
        if (node->num_keys < t-1 || node->num_keys > 2*t-1) {
            fprintf(stderr, "ERROR: Non-root node has %d keys, allowed [%d, %d]\n",
                    node->num_keys, t-1, 2*t-1);
            (*error_count)++;
            return false;
        }
    }

    // Check key ordering within node
    for (int i = 0; i < node->num_keys - 1; i++) {
        if (node->keys[i] >= node->keys[i+1]) {
            fprintf(stderr, "ERROR: Keys not sorted at index %d: %d >= %d\n",
                    i, node->keys[i], node->keys[i+1]);
            (*error_count)++;
            return false;
        }
    }

    // Handle leaf nodes
    if (node->is_leaf) {
        // All child pointers must be NULL
        for (int i = 0; i <= 2*t; i++) {
            if (node->children[i] != NULL) {
                fprintf(stderr, "ERROR: Leaf node has non-NULL child at index %d\n", i);
                (*error_count)++;
                return false;
```

```
        }
    }

    // Record leaf depth
    if (*leaf_depth == -1) {
        *leaf_depth = depth;
    } else if (depth != *leaf_depth) {
        fprintf(stderr, "ERROR: Leaf depth mismatch: %d vs %d\n",
                depth, *leaf_depth);
        (*error_count)++;
        return false;
    }

    // Set min/max for this leaf
    *min_key = (node->num_keys > 0) ? node->keys[0] : INT_MAX;
    *max_key = (node->num_keys > 0) ? node->keys[node->num_keys-1] : INT_MIN;
    return true;
}

// Internal node validation
if (node->num_keys + 1 > 2*t) {
    fprintf(stderr, "ERROR: Internal node has too many children: %d\n",
            node->num_keys + 1);
    (*error_count)++;
    return false;
}
```

```
    // Recursively validate children with key range checks
    bool all_children_valid = true;
    for (int i = 0; i <= node->num_keys; i++) {
        if (node->children[i] == NULL) {
            fprintf(stderr, "ERROR: Internal node has NULL child at index %d\n", i);
            (*error_count)++;
            all_children_valid = false;
            continue;
        }

        int child_min, child_max;
        bool child_valid = _validate_node(node->children[i], t, false,
                                          &child_min, &child_max,
                                          depth + 1, leaf_depth, error_count);

        if (!child_valid) {
            all_children_valid = false;
        }

        // Check child's keys against parent's keys
        if (i > 0) {
            if (child_max >= node->keys[i-1]) {
                fprintf(stderr, "ERROR: Child %d max key %d >= parent key %d\n",
                        i, child_max, node->keys[i-1]);
                (*error_count)++;
                all_children_valid = false;
            }
```

```
        }

        if (i < node->num_keys) {
            if (child_min <= node->keys[i]) {
                fprintf(stderr, "ERROR: Child %d min key %d <= parent key %d\n",
                        i, child_min, node->keys[i]);
                (*error_count)++;
                all_children_valid = false;
            }
        }
    }

    // Set min/max for this internal node
    *min_key = (node->num_keys > 0) ? node->keys[0] : INT_MAX;
    *max_key = (node->num_keys > 0) ? node->keys[node->num_keys-1] : INT_MIN;

    return all_children_valid;
}

// Public validation function
bool btree_validate(BTree* tree) {
    if (tree == NULL) {
        fprintf(stderr, "ERROR: Tree is NULL\n");
        return false;
    }

    if (tree->t < 2) {
```

```
        fprintf(stderr, "ERROR: Minimum degree t=%d < 2\n", tree->t);
        return false;
    }

    int error_count = 0;
    int leaf_depth = -1;
    int dummy_min, dummy_max;

    bool valid = _validate_node(tree->root, tree->t, true,
                                &dummy_min, &dummy_max,
                                0, &leaf_depth, &error_count);

    if (error_count > 0) {
        fprintf(stderr, "Validation failed with %d error(s)\n", error_count);
    }

    return valid && (error_count == 0);
}

#else
// Stub for non-debug builds
bool btree_validate(BTree* tree) {
    (void)tree;  // Unused parameter
    return true; // Always passes in release builds
}
#endif // DEBUG_VALIDATE
```

Core Logic Skeleton for Error Handling:

```c
// btree.c - Error handling in public API functions

BTree* btree_create(int min_degree) {
    // TODO 1: Validate min_degree parameter (must be ≥ 2)
    // TODO 2: Allocate memory for BTree structure using malloc
    // TODO 3: Check if malloc returned NULL (memory allocation failure)
    // TODO 4: Initialize fields: root = NULL, t = min_degree, key_count = 0
    // TODO 5: Return the created BTree* or NULL on failure
}

bool btree_insert(BTree* tree, int key, void* value) {
    // TODO 1: Validate tree parameter (not NULL)
    // TODO 2: Check for duplicate key using btree_search()
    // TODO 3: If root is NULL (empty tree), create new root node
    // TODO 4: Check if root is full, if so:
    //   a) Create new root
    //   b) Split old root as child of new root
    //   c) Update tree->root pointer
    // TODO 5: Call insert_non_full on the (possibly new) root
    // TODO 6: Increment tree->key_count if insertion succeeded
    // TODO 7: Return true on success, false on failure (duplicate or memory error)
}

bool btree_delete(BTree* tree, int key) {
    // TODO 1: Validate tree parameter (not NULL)
    // TODO 2: Check if tree is empty (root == NULL or root->num_keys == 0)
    // TODO 3: Call delete_from_subtree on root
    // TODO 4: If root becomes empty after deletion:
```

```
    //   a) If root is leaf, set root = NULL (tree becomes empty)
    //   b) If root is internal, set root = root->children[0] (decrease height)
    // TODO 5: Decrement tree->key_count if deletion succeeded
    // TODO 6: Return true if key was found and deleted, false otherwise
}

// Example of memory allocation with error checking
BTreeNode* node_create(int t, bool is_leaf) {
    // TODO 1: Calculate array sizes based on t (max keys = 2t-1, max children = 2t)
    // TODO 2: Allocate memory for node structure using malloc
    // TODO 3: Check for allocation failure, return NULL if failed
    // TODO 4: Allocate keys array (size 2*t-1 * sizeof(int))
    // TODO 5: Check for allocation failure, free node and return NULL if failed
    // TODO 6: Allocate children array (size 2*t * sizeof(BTreeNode*))
    // TODO 7: Check for allocation failure, free keys and node, return NULL if failed
    // TODO 8: Initialize all child pointers to NULL
    // TODO 9: Set num_keys = 0, is_leaf flag
    // TODO 10: Return the allocated node
}
```

Language-Specific Hints for C:

1. **Memory Allocation Errors:** Always check `malloc` / `calloc` / `realloc` return values. Use `perror("malloc")` to print system error messages.
2. **Cleanup on Failure:** Implement a rollback pattern: if allocation fails mid-way through node creation, free previously allocated components in reverse order.
3. **Debug Builds:** Compile with `-DDEBUG_VALIDATE` to enable validation, `-g` for debug symbols, and `-fsanitize=address` for memory error detection.
4. **Assertions:** Use `assert()` liberally during development but note they're removed in release builds (`-DNDEBUG`).

Debugging Tips Table:

Symptom	Likely Cause	How to Diagnose	Fix
Segmentation fault after insert	Accessing child pointer beyond allocated array	Run with Valgrind; add array bounds checks before each access	Ensure `children` array size is 2*t, indices are 0 ≤ i ≤ num_keys
Tree height grows unexpectedly	Root split not increasing height properly	Print tree before/after each root split; check new root creation	When splitting root, create new root with old root as first child
Keys disappear after deletion	Incorrect merge operation losing keys	Use `btree_print()` after each deletion step; validate after each operation	Ensure all keys from both nodes and parent separator are preserved in merge
Infinite recursion during search	Child pointer points to ancestor (cycle)	Add cycle detection in validation; limit recursion depth	Check `node_split_child()` doesn't create circular references
Memory leak after many operations	Not freeing nodes during merge or tree destruction	Use Valgrind; track allocations/frees	Ensure `node_destroy()` recursively frees all children; call in `btree_destroy()`
Wrong search results	Keys not sorted within nodes	Validate intra-node ordering; check `node_insert_key()` shift logic	Binary search requires sorted keys; verify insertion maintains order
"Borrow from sibling" fails	Not checking sibling has sufficient keys	Print sibling key count before attempting borrow	Only borrow if sibling has > t-1 keys; otherwise merge

Milestone Checkpoint for Error Handling:

After implementing error handling and validation:

1. **Run the test suite with validation enabled:**

```bash
gcc -DDEBUG_VALIDATE -g src/btree.c src/btree_validate.c tests/test_btree.c -o test_btree

./test_btree
```

Expected: All tests pass with no validation errors printed.

2. **Test edge cases manually:**
   - Create tree with t=1 (should fail or use default)
   - Insert duplicate keys (should return false, tree unchanged)

- Delete non-existent key (should return false, tree unchanged)
- Insert until root splits multiple times (check height increases)
- Delete until root shrinks (check height decreases)

3. **Run memory checker:**

```bash
valgrind --leak-check=full ./test_btree
```

Expected: "All heap blocks were freed -- no leaks are possible"

4. **Stress test with validation:** Create a test that performs 10,000 random insertions/deletions, calling `btree_validate()` every 100 operations. The validation should never fail if implementation is correct.

11. Testing Strategy

> **Milestone(s):** Milestones 1-4 (all operations)

A comprehensive testing strategy is crucial for implementing a complex, self-balancing data structure like a B-tree. Unlike simpler structures where bugs might cause obvious crashes, B-tree bugs can be subtle—causing silent corruption of invariants that only manifests later during specific sequences of operations. This section outlines a systematic testing approach combining unit tests, property-based tests, and milestone-specific checkpoints to build confidence incrementally.

Testing Approach and Scenarios

Think of testing a B-tree like **testing a self-organizing library**. You need to verify not only that books can be found when requested (basic search), but also that the library's internal reorganization rules—splitting shelves when full, merging shelves when too empty, and updating the catalog index—work correctly under all scenarios. The testing strategy progresses from simple unit tests (verifying individual shelves work) to property-based tests (verifying the entire library stays organized after random activity) to edge case tests (verifying extreme scenarios like emptying and refilling the library).

The testing pyramid for our B-tree implementation has three layers:

1. **Unit Tests**: Test individual components (`BTreeNode` functions) in isolation.
2. **Integration/Operation Tests**: Test the public API (`btree_insert`, `btree_delete`, `btree_search`) and their interactions.
3. **Property-Based & Stress Tests**: Verify that the B-tree invariants hold after thousands of random operations.

Unit Testing Strategy

Unit tests focus on the smallest testable components: the node-level helper functions. These functions form the building blocks of the main algorithms, and their correctness is essential.

Component to Test	Test Scenarios	Expected Behavior	Validation Method
`node_create`	Create leaf and internal nodes with different `t` values	Node memory allocated, `num_keys` = 0, `is_leaf` set correctly, arrays allocated with correct capacities	Check struct field values, verify no memory leaks
`node_destroy`	Destroy a single node, destroy a subtree	All memory freed, no dangling pointers	Use Valgrind or address sanitizer to confirm clean deallocation
`node_find_key_index`	Binary search on empty node, node with keys, key less than all, key greater than all, key between keys, exact match	Returns correct index (0 for empty, correct position for non-existent, exact index for match)	Compare returned index against manual calculation, track comparison count
`node_is_full` / `node_is_underfull`	Nodes with `t-1`, `t`, `2t-1`, `2t` keys (for full), and `t-2`, `t-1`, `t` keys (for underfull)	Returns true/false according to B-tree capacity rules	Verify against mathematical definition: full = `num_keys == 2*t-1`, underfull = `num_keys < t-1`
`node_insert_key`	Insert at beginning, middle, end of key array	Key inserted at correct position, existing keys shifted right, `num_keys` incremented	Verify array contents and count
`node_remove_key`	Remove from beginning, middle, end	Key removed, remaining keys shifted left, `num_keys` decremented	Verify array contents and count
`node_split_child`	Split leaf child, split internal child, split when parent has capacity	Child split into two nodes, median key promoted to parent at correct index, parent's keys/children arrays adjusted	Verify all invariants: new nodes have `t-1` keys, parent gains one key, total keys preserved

> **Key Insight**: Unit testing `node_split_child` is particularly important because it's the core operation that maintains B-tree balance during insertion. A bug here will propagate through the entire tree.

Integration Testing: Public API Operations

These tests verify that the main operations (`btree_insert` , `btree_delete` , `btree_search`) work together correctly and maintain all B-tree invariants.

Operation Sequence	Test Scenario	Invariants to Verify	Expected Outcome
Search	Search in empty tree, search for existing key, search for non-existent key	None (search doesn't modify tree)	Correct `found` flag, correct `value` returned, `comparisons` count reasonable
Insert	Insert into empty tree, insert until root splits, insert keys in ascending/descending/random order	All five B-tree invariants (see Data Model section)	Tree height increases only when root splits, all keys findable after insertion
Delete	Delete from leaf with > `t-1` keys, delete causing borrow from left/right sibling, delete causing merge, delete from internal node	All five B-tree invariants, no underfull nodes except root	Key no longer findable, tree height decreases only when root has single child
Mixed Operations	Interleave insertions and deletions, repeating same keys, deleting non-existent keys	Invariants hold after each operation	Tree remains balanced, no memory corruption

A powerful technique is to write a **wrapper function** that calls the public API and then immediately validates all invariants using `btree_validate` . This can be used in every test:

```c
// Pseudocode for test helper
void checked_insert(BTree* tree, int key, void* value) {
    bool success = btree_insert(tree, key, value);
    assert(success); // or handle duplicates as needed
    assert(btree_validate(tree)); // invariants must hold
}
```

Property-Based Testing

Property-based testing generates random sequences of operations and verifies that certain properties always hold. This is excellent for uncovering edge cases that manual test design might miss.

Property to Verify	Testing Approach	How to Validate
Insertion maintains findability	Generate N random keys, insert all, then verify each key exists via `btree_search`	All searches return `found == true`
Deletion removes keys	Insert N random keys, delete a random subset, verify deleted keys are gone, others remain	Deleted keys return `found == false`, others `true`
Sorted order traversal	After any operation sequence, perform in-order traversal (depth-first, left-to-right)	Traversed keys are in strictly ascending order
Invariant preservation	After every single operation (insert/delete), call `btree_validate`	Validation passes for all intermediate states
Height bounds	After any operation sequence, verify tree height ≤ `log_t((n+1)/2) + 1` where n = key count	Height remains within theoretical B-tree bounds

> **Implementation Note**: For property-based tests in C, consider using a simple deterministic pseudo-random number generator with a fixed seed for reproducible tests. Run thousands of operations to stress the implementation.

Edge Case and Scenario Testing

Specific scenarios known to trigger bugs in B-tree implementations:

Scenario	Description	Why It's Tricky	Test Approach
Root split	Inserting into a full root (tree height increases)	Requires creating new root, special case in insertion algorithm	Insert exactly `2t-1` keys in ascending order, forcing root fills and splits
Cascading merges	Deletion causing underflow that propagates up multiple levels	Must correctly handle sequential merges up to root	Build tree of height 3, delete specific keys to trigger multiple merges
Borrow from left vs right	Underflow node with left sibling that has extra keys, vs right sibling	Symmetry bugs: implementation might work for one side but not the other	Create asymmetric sibling capacities, test deletions that require each borrow direction
Internal node deletion	Deleting a key from an internal node (requires predecessor/successor swap)	Must choose correct predecessor/successor, handle leaf deletion after swap	Delete keys that appear at various positions in internal nodes
Duplicate key insertion	Attempting to insert a key that already exists	Implementation may assume unique keys; must handle gracefully (return false or overwrite)	Depending on design choice, test that duplicates are rejected or values are updated
Minimum degree variations	Testing with different `t` values (e.g., t=2, t=3, t=10)	Different `t` affects node capacities and tree shape	Run same test suite with multiple `t` values
Empty tree operations	Deleting from empty tree, searching empty tree	Edge cases for root being `NULL`	Verify graceful handling (return false, not crash)
Single node tree	All operations when tree has only root (no children)	Special cases in deletion (root can become empty)	Insert then delete all keys, verify root can become empty

Milestone Checkpoints

Each milestone in the project has specific deliverables that should be testable. The following table provides concrete checkpoints to verify your implementation is on track after completing each milestone.

Milestone	What to Test	Expected Output	How to Verify Success
Milestone 1: Node Structure	Node creation with different `t` values, leaf vs internal flag, key array capacity, child pointer array for internal nodes	Nodes created with correct field values, arrays of appropriate size	Use debug prints or a test function to inspect node fields. Memory allocators should allocate `(2*t-1)*sizeof(int)` for keys and `(2*t)*sizeof(BTreeNode*)` for children.
	`node_insert_key` and `node_remove_key` on a node	Keys maintain sorted order after insert/remove, `num_keys` updated correctly	Manually verify key array contents after series of operations.
	`node_is_full` and `node_is_underfull`	Returns true precisely when `num_keys == 2t-1` (full) or `num_keys < t-1` (underfull)	Test with nodes having `t-2`, `t-1`, `t`, `2t-1`, `2t` keys (the last should not occur in practice).
Milestone 2: Search	Search in empty tree	Returns `found = false`, `value = NULL`	Basic sanity check.
	Search in single-node tree (leaf) for existing and non-existent keys	Correct `found` flag, correct value returned for existing key	Insert a few keys, search for each.
	Search in multi-level tree	Recursively descends through internal nodes, finds keys at all levels	Build a small tree manually (by calling node functions) with known structure, verify search finds all keys.
	Binary search within node (`node_find_key_index`)	Returns correct index for key position, comparison count increments appropriately	Test with keys less than all, greater than all, between, and exact matches.
Milestone 3: Insert with	Insert into non-full leaf	Key added in sorted position,	Insert up to `2t-2` keys into an empty tree (root is leaf), verify tree remains

Milestone	What to Test	Expected Output	How to Verify Success
Split		no splits occur	single node.
	Insert causing leaf split	Leaf splits, median promoted to parent (root), new leaf created	Insert `2t-1` keys in ascending order; after the `(2t-1)` th insertion, root should split (height becomes 2). Verify all keys findable.
	Insert causing internal node split	Split propagates upward, tree height increases only when root splits	Build tree of height 2, fill internal nodes until they split, verify invariants.
	Insertion with random key order	All invariants maintained, tree remains balanced	Insert 100 random keys, validate after each insertion.
Milestone 4: Delete with Rebalancing	Delete from leaf with sufficient keys (`num_keys > t-1`)	Key removed, no structural changes	Build tree, delete key from leaf that won't cause underflow, verify key gone, tree valid.
	Delete from leaf causing borrow from right sibling	Key borrowed from right sibling through parent, parent key updated	Create specific tree structure where leaf has `t-1` keys after deletion, right sibling has ≥ `t` keys. Verify borrow occurs.
	Delete from leaf causing borrow from left sibling	Symmetric case to above.	Test both directions to catch asymmetry bugs.
	Delete from leaf causing merge with sibling	Two siblings and parent separator merge into one node	Create scenario where both siblings have exactly `t-1` keys, deletion causes underflow, merge occurs.
	Delete from internal node (using predecessor)	Key replaced with predecessor from leaf, predecessor deleted from leaf	Delete key from internal node, verify tree valid and key replaced correctly.
	Delete from internal node (using successor)	Similar to predecessor case.	Ensure both predecessor and successor work if your implementation chooses one.
	Cascading merges up to root	Multiple merges cause root to	Build tree of height 3, delete specific sequence to cause merges up to root.

Milestone	What to Test	Expected Output	How to Verify Success
		have single child, height decreases	
	Delete all keys (tree emptying)	Tree can become empty (root = NULL) or root with 0 keys (depending on implementation choice)	Insert N keys, delete all, verify tree empty or root empty.

> **Critical Testing Principle**: After completing each milestone, re-run all previous milestone tests to ensure new code hasn't broken existing functionality. This is especially important when moving from insertion to deletion, as deletion builds upon insertion's tree structure.

Automated Test Harness Recommendation

For efficiency, implement a simple test harness that:

1. **Runs unit tests** for each component.
2. **Runs operation sequences** and validates invariants after each step.
3. **Runs property-based tests** with random seeds.
4. **Reports failures** with detailed information (tree state, operation that failed).

A sample test progression might look like:

```
$ ./test_btree --milestone 1  # Node structure tests
$ ./test_btree --milestone 2  # Search tests (also runs milestone 1 tests)
$ ./test_btree --milestone 3  # Insert tests (runs milestones 1-2)
$ ./test_btree --milestone 4  # Delete tests (runs all)
$ ./test_btree --property 10000  # Run 10,000 random operations
```

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Test Framework	Custom test harness with asserts and counters	Use a testing framework like **Check** (C unit testing framework) or **Google Test** (if using C++)
Randomness	`rand()` with fixed seed for reproducibility	Use a more robust PRNG like PCG or Mersenne Twister
Memory Checking	Manual tracking with `malloc`/`free` wrappers	Use **Valgrind** or **AddressSanitizer** for automatic detection
Invariant Validation	Implement `btree_validate` as described in Section 10	Extend validation with more detailed error messages and graph output for debugging

Recommended File/Module Structure

Add test files to your project structure:

```
btree-project/
├── include/
│   └── btree.h              # Public API declarations
├── src/
│   ├── btree.c              # B-tree implementation
│   ├── node.c               # Node-level functions
│   └── operations.c         # Search, insert, delete algorithms
└── tests/                   # Test directory
    ├── test_harness.c       # Main test runner
    ├── test_node.c          # Milestone 1: Node tests
    ├── test_search.c        # Milestone 2: Search tests
    ├── test_insert.c        # Milestone 3: Insert tests
    ├── test_delete.c        # Milestone 4: Delete tests
    ├── test_property.c      # Property-based tests
    └── test_utils.c         # Test utilities (tree printing, validation helpers)
```

Infrastructure Starter Code: Test Harness

Here's a complete, reusable test harness foundation:

```c
/* test_harness.c - Simple test framework for B-tree */

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <time.h>

#include "../include/btree.h"


#define TEST_START(name) printf("TEST: %s... ", name); fflush(stdout)

#define TEST_PASS() printf("PASS\n")

#define TEST_FAIL(reason) printf("FAIL: %s\n", reason); exit(1)


/* Global test counters */

static int tests_run = 0;

static int tests_passed = 0;


/* Wrapper for btree_insert that validates after each insert */

void checked_insert(BTree* tree, int key, void* value) {

    tests_run++;

    bool success = btree_insert(tree, key, value);

    if (!success) {

        TEST_FAIL("insert returned false (duplicate key?)");

    }

    if (!btree_validate(tree)) {

        TEST_FAIL("tree invariant violated after insert");

    }

    tests_passed++;

}


/* Wrapper for btree_delete */
```

```
void checked_delete(BTree* tree, int key) {
    tests_run++;
    bool success = btree_delete(tree, key);
    if (!success) {
        TEST_FAIL("delete returned false (key not found?)");
    }
    if (!btree_validate(tree)) {
        TEST_FAIL("tree invariant violated after delete");
    }
    tests_passed++;
}

/* Verify search returns expected result */
void verify_search(BTree* tree, int key, bool expected_found, void* expected_value) {
    tests_run++;
    SearchResult res = btree_search(tree, key);
    if (res.found != expected_found) {
        printf("FAIL: key %d: expected found=%d, got %d\n",
               key, expected_found, res.found);
        exit(1);
    }
    if (expected_found && res.value != expected_value) {
        printf("FAIL: key %d: value mismatch\n", key);
        exit(1);
    }
    tests_passed++;
}
```

```c
/* Run all milestone tests */
void run_milestone1_tests();
void run_milestone2_tests();
void run_milestone3_tests();
void run_milestone4_tests();
void run_property_tests(int num_ops);

int main(int argc, char** argv) {
    printf("=== B-tree Test Suite ===\n");


    /* Run specific milestone tests based on command line */
    if (argc < 2) {
        /* Run all tests */
        run_milestone1_tests();
        run_milestone2_tests();
        run_milestone3_tests();
        run_milestone4_tests();
        run_property_tests(1000);
    } else if (strcmp(argv[1], "--milestone1") == 0) {
        run_milestone1_tests();
    } else if (strcmp(argv[1], "--milestone2") == 0) {
        run_milestone1_tests();  /* Milestone 2 depends on 1 */
        run_milestone2_tests();
    } else if (strcmp(argv[1], "--milestone3") == 0) {
        run_milestone1_tests();
        run_milestone2_tests();
        run_milestone3_tests();
```

```
    } else if (strcmp(argv[1], "--milestone4") == 0) {
        run_milestone1_tests();
        run_milestone2_tests();
        run_milestone3_tests();
        run_milestone4_tests();
    } else if (strcmp(argv[1], "--property") == 0) {
        int num_ops = argc > 2 ? atoi(argv[2]) : 1000;
        run_property_tests(num_ops);
    }


    printf("\n=== Summary: %d/%d tests passed ===\n", tests_passed, tests_run);
    return tests_passed == tests_run ? 0 : 1;
}
```

Core Logic Skeleton for Test Utilities

For the test utilities that you'll need to implement (like tree validation and printing), here are skeletons with TODOs:

```c
/* test_utils.c - Utilities for testing */

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include "../include/btree.h"


/* Recursive helper for btree_validate */

bool validate_node(BTreeNode* node, int t, bool is_root,

                   int* min_key, int* max_key, int depth, int* leaf_depth) {

    // TODO 1: Check that node pointer is not NULL (unless empty tree handled by caller)

    // TODO 2: Verify key count:

    //   - If is_root: 0 <= num_keys <= 2t-1

    //   - Else: t-1 <= num_keys <= 2t-1

    // TODO 3: Verify keys are in strictly increasing order

    // TODO 4: If leaf node:

    //   - Verify children pointers are all NULL

    //   - Record leaf depth (first leaf found sets *leaf_depth, others must match)

    // TODO 5: If internal node:

    //   - Verify num_keys + 1 == number of non-NULL children

    //   - Verify each child's keys are in correct range relative to parent keys

    //   - Recursively validate each child

    // TODO 6: Update min_key and max_key for this subtree for parent validation

    return true;  // Placeholder

}


bool btree_validate(BTree* tree) {

    // TODO 1: Handle empty tree (root == NULL): return true

    // TODO 2: Initialize leaf_depth to -1
```

```
    // TODO 3: Call validate_node on root with is_root = true
    // TODO 4: Check that all leaves are at same depth (via leaf_depth)
    return false;  // Placeholder
}

void print_node(BTreeNode* node, int depth) {
    // TODO 1: Print indentation based on depth
    // TODO 2: Print node type (leaf/internal) and key count
    // TODO 3: Print keys in format like [k1, k2, ..., kn]
    // TODO 4: If internal node, recursively print children with depth+1
}

void btree_print(BTree* tree) {
    // TODO 1: Print tree header with t, height, size
    // TODO 2: If root is NULL, print "Empty tree"
    // TODO 3: Otherwise, call print_node(root, 0)
}
```

Language-Specific Hints: C Testing

- **Memory Leak Detection**: Always run tests with Valgrind: `valgrind --leak-check=full ./test_btree`
- **Random Number Generation**: Use `srand(42)` for reproducible tests during development, but also test with different seeds.
- **Assertions**: Use `assert()` from `<assert.h>` for invariants that should never fail in correct code.
- **Modular Compilation**: Compile tests separately from main implementation: `gcc -c src/btree.c -o build/btree.o` then link with test files.

Milestone Checkpoint Commands

After implementing each milestone, run these commands to verify:

```bash
# After Milestone 1 (Node Structure):

$ gcc -std=c99 -g -Iinclude src/node.c tests/test_node.c tests/test_harness.c -o test_node

$ ./test_node --milestone1

# Expected: "PASS" for all tests, no Valgrind errors


# After Milestone 2 (Search):

$ gcc -std=c99 -g -Iinclude src/*.c tests/test_search.c tests/test_harness.c
tests/test_utils.c -o test_search

$ ./test_search --milestone2

# Expected: Search finds all inserted keys, returns false for non-existent keys


# After Milestone 3 (Insert):

$ gcc -std=c99 -g -Iinclude src/*.c tests/test_insert.c tests/test_harness.c
tests/test_utils.c -o test_insert

$ ./test_insert --milestone3

# Expected: Tree validates after every insertion, root splits correctly


# After Milestone 4 (Delete):

$ gcc -std=c99 -g -Iinclude src/*.c tests/test_delete.c tests/test_harness.c
tests/test_utils.c -o test_delete

$ ./test_delete --milestone4

# Expected: All deletion cases handled, tree validates, no memory leaks


# Final comprehensive test:

$ gcc -std=c99 -g -Iinclude src/*.c tests/*.c -o test_all

$ valgrind --leak-check=full ./test_all

# Expected: All tests pass, "0 errors from Valgrind"
```

Debugging Tips for Tests

Symptom	Likely Cause	How to Diagnose	Fix
Segmentation fault in `node_create`	Memory allocation failure, or accessing uninitialized pointer	Check `malloc` return values, ensure all array indices are within bounds	Add NULL checks after malloc, verify `t > 1`
Tree validation fails after insert	Invariant violation (e.g., keys not sorted, wrong key count)	Use `btree_print` to visualize tree structure before/after failing operation	Check `node_insert_key` shifting logic, verify split median calculation
Search finds keys after insert but not after delete	Deletion incorrectly removes extra keys or corrupts pointers	Print tree before and after deletion, check borrow/merge logic	Verify sibling indices are correct during borrow/merge
Memory leak reported by Valgrind	`node_destroy` not recursive, or missing free for arrays	Run Valgrind with `--track-origins=yes`, check which allocation isn't freed	Ensure `node_destroy` frees keys, children, then the node itself
Infinite recursion during search/insert	Base case missing (leaf check), or child pointer incorrect	Add debug prints showing node address and depth	Ensure `is_leaf` is set correctly, recursion stops at leaves
Tree height grows too quickly	Splitting too aggressively (splitting non-full nodes)	Check `node_is_full` logic, verify split only called when `num_keys == 2*t-1`	Fix `node_is_full` condition

> **Pro Tip**: Implement `btree_print` early (even as a simple text dump) and use it liberally in tests. Visualizing the tree structure is the fastest way to spot structural bugs.

12. Debugging Guide

> **Milestone(s):** Milestones 1-4 (all operations)

Implementing a B-tree is a complex undertaking involving recursive algorithms, careful pointer management, and maintenance of multiple invariants. Even with a solid design, bugs are inevitable. This section provides a structured approach to identifying, diagnosing, and fixing the most common categories of bugs you'll

encounter during implementation. Think of this as your troubleshooting manual—a collection of known failure patterns and the tools to resolve them.

Debugging a B-tree is like being a building inspector for a multi-story library. You need to check each floor (node) to ensure it has the right number of books (keys), that the floor plans (child pointers) lead to the correct sub-floors, and that the entire structure follows building codes (B-tree invariants). When something collapses (segmentation fault) or books go missing (lost keys), you need systematic inspection techniques to find the root cause.

Common Bug Symptoms and Fixes

The following table catalogues the most frequent symptoms, their likely underlying causes, and specific fixes. These are drawn from common patterns observed when implementing B-trees in C.

Symptom	Likely Cause	Fix
Segmentation fault immediately after insertion or deletion	Accessing a `children` pointer at an index beyond the current `num_keys` (remember: internal nodes have `num_keys + 1` children).	Before accessing `node->children[i]`, verify `i <= node->num_keys` for internal nodes. For leaf nodes, never access children. Use `assert(node->is_leaf
Keys disappearing after a split operation	Incorrect calculation of the median index during `node_split_child`, leading to wrong key distribution between the old and new nodes. The median key should be at index `t-1` (0-based) in a node with `2t-1` keys.	Double-check the split logic: the original node keeps the first `t-1` keys (indices 0 to `t-2`), the new node gets the last `t-1` keys (indices `t` to `2t-2`), and the median key at index `t-1` is promoted.
Infinite recursion during search/insert/delete	Forgetting to check the `is_leaf` flag before attempting to recurse into child nodes. The recursive call is made even when `node->is_leaf` is true, leading to endless descent into invalid memory.	Ensure every recursive function has a base case that checks `if (node->is_leaf) return ...` before attempting to access `node->children`.
Tree height increases unnecessarily (more than one root split)	Failing to update the tree's root pointer after a root split. The old root remains, and a new root is created but not assigned to `tree->root`.	In `btree_insert`, after calling a helper that may split the root, check if the root was split (e.g., if the old root is full). If so, create a new root with the promoted key and two children.
"Key not found" for keys that were definitely inserted	Keys are not maintained in sorted order within nodes, breaking the binary search assumption. Insertion or split operations may place keys in wrong positions.	After any key insertion or movement (during insert, borrow, merge), ensure the keys array remains sorted. Write a helper `node_validate_order(node)` that asserts ascending order for debugging.
Memory leak (Valgrind reports lost blocks)	Not recursively freeing child nodes in `node_destroy`. Only freeing the node's own arrays but not its children's arrays for internal nodes.	In `node_destroy`, if `!node->is_leaf`, iterate through `i = 0 to node->num_keys` and call `node_destroy(node->children[i])` before freeing the node's own arrays.
Underflow (node has < t-1 keys) after deletion	The `strengthen_child` (or equivalent preemptive borrowing/merging during descent)	During deletion descent, before recursing into a child, ensure the child has at least `t` keys. If not, borrow from a sibling or merge with a

Symptom	Likely Cause	Fix
	was not performed correctly, or the borrow/merge logic itself is flawed.	sibling. Re-check the conditions for borrowing (sibling has ≥ t keys) vs. merging (sibling has exactly t-1 keys).
Child pointer becomes NULL after merge, causing segfault later	When merging two children, the right child node is freed but its pointer in the parent's `children` array is not set to NULL or removed. The array may later be accessed assuming all entries are valid.	After `merge_with_left_sibling`, the right child is absorbed. You must shift the parent's child pointers left to remove the now-empty slot. For `borrow` operations, child pointers are rearranged but not removed.
Duplicate keys appear in the tree	The insertion logic does not check for existing keys before inserting. B-trees typically allow duplicates? (Usually not; we assume unique keys for this implementation).	In `btree_insert`, before starting the insertion process, call `btree_search`. If found, decide whether to replace the value or reject the insertion. The design assumes unique keys, so you may choose to return false.
Binary search returns incorrect index (off-by-one)	Incorrect handling of the `high` index (should be `num_keys - 1` initially) or confusion between "index where key is found" and "index where key should be inserted".	The `node_find_key_index` should return the first index where `key <= node->keys[i]`. If `key` is greater than all keys, return `num_keys`. Test with a simple linear search as a reference implementation.
After many insertions and deletions, tree invariants break	Cumulative error from small mistakes in edge cases, e.g., root's minimum key count (can be 1), handling of the last child pointer during split, etc.	Implement `btree_validate` and call it after every operation in debug mode. This function recursively checks all invariants: key count bounds, ordering, leaf property consistency, and child pointer counts.

Debugging Techniques and Tools

Effective debugging requires more than just guessing. You need a systematic approach and the right tools. Below are techniques specifically tailored for B-tree implementations in C.

1. Verbose Logging with Function Tracing

Add logging macros that print the execution flow and key data structures. This lets you see exactly how the tree evolves.

> **Mental Model:** This is like adding security cameras at every staircase and bookshelf in the library. You can replay the footage to see exactly when and where a book was misplaced.

How to implement:

- Create a `DEBUG` macro that can be enabled/disabled at compile time.
- At the entrance and exit of each major function (`search_recursive`, `insert_non_full`, `delete_from_subtree`), log the function name, parameters (e.g., node address, key), and key state (e.g., `num_keys`).
- Before and after critical operations (split, borrow, merge), print the entire state of the affected nodes.

Example log output snippet:

```
[INSERT] insert_non_full node=0x1234, key=42, is_leaf=0, num_keys=3
[SPLIT] Splitting child at index 1 of parent 0x1234
  Before split: child keys=[10,20,30,40,50]
  After split: left keys=[10,20], promoted=30, right keys=[40,50]
```

This allows you to trace the exact sequence of operations leading to a bug.

2. Tree Visualization Function

A text-based tree printer is invaluable for seeing the overall structure. Implement `btree_print` that outputs the tree in a readable format.

Recommended format (indentation-based):

```
B-Tree (t=3, height=2, keys=10)
[Root] keys=30
  [Child 0] keys=10 20
    [Leaf] keys=5 7 9
    [Leaf] keys=12 15
    [Leaf] keys=22 25
  [Child 1] keys=40 50
    [Leaf] keys=32 35 38
    [Leaf] keys=42 45 48
    [Leaf] keys=52 55
```

This reveals structural problems like missing nodes, incorrect key distributions, or broken parent-child relationships.

Implementation Guidance: Use a recursive helper that takes a node and an indentation level. For each node, print its keys. If not a leaf, recursively print each child with increased indentation.

3. Invariant Verification Function

The single most powerful debugging tool is an automatic invariant checker. Implement `btree_validate` that traverses the entire tree and asserts every B-tree rule.

What to check:

1. **Key count bounds:** For root: `0 <= num_keys <= 2t-1`. For others: `t-1 <= num_keys <= 2t-1`.

2. **Key ordering:** Keys within each node are strictly increasing.
3. **Leaf consistency:** If `is_leaf` is true, all child pointers are NULL. If false, all child pointers are non-NULL (for indices `0..num_keys`).
4. **Child count:** Internal nodes have exactly `num_keys + 1` children.
5. **Recursive child validation:** For internal nodes, every key in child `i` is less than `node->keys[i]`, and every key in child `i+1` is greater than `node->keys[i]`. Also recursively validate each child.
6. **Tree height consistency:** All leaf nodes are at the same depth (same distance from root).

Call `btree_validate` after every insert and delete operation during development. When a violation occurs, print the node state and abort. This catches bugs immediately at their source rather than allowing corruption to propagate.

4. Memory Debuggers: Valgrind and AddressSanitizer

Memory errors are common in C. Use these tools to detect:

- Use of uninitialized memory
- Buffer overflows (writing beyond allocated arrays)
- Memory leaks
- Use-after-free errors

Valgrind usage:

```bash
valgrind --leak-check=full ./your_btree_program
```

AddressSanitizer (faster, compile-time instrumentation):

```bash
gcc -fsanitize=address -g your_code.c -o your_btree_program

./your_btree_program
```

These tools will pinpoint the exact line where illegal memory access occurs, which is especially helpful for segmentation faults.

5. Interactive Debugging with GDB

When you have a reproducible crash, use GDB to inspect program state at the moment of failure.

Essential GDB commands:

- `break function_name` – Set a breakpoint
- `run` – Start program
- `backtrace` or `bt` – Show call stack at crash
- `print node->num_keys` – Inspect variable
- `x/10wx node->keys` – Examine memory as array of words

- `step` and `next` – Single-step execution

Pro tip: Create a GDB script that sets breakpoints at key functions and prints node information automatically.

6. Unit Test with Known Sequences

Create small, focused test cases that exercise specific operations in isolation. For example:

1. **Test root split:** Insert exactly `2t-1` keys in ascending order, forcing a single root split.
2. **Test borrow from right sibling:** Create a specific tree configuration, then delete a key that triggers borrowing.
3. **Test merge cascade:** Build a tree where deletion causes a merge that propagates upward.

By isolating scenarios, you simplify the debugging surface. Compare your tree's state after each operation against a manually computed expected state.

7. Property-Based Testing

After basic operations work, use property-based testing to find edge cases. The key properties of a B-tree are:

- All keys are always present after insertion.
- No keys are present after deletion.
- The tree remains balanced (all leaves at same depth).
- Keys are always in sorted order when traversed in-order.

Generate random sequences of insertions and deletions, and after each operation, verify these properties using your `btree_validate` function. This can uncover subtle bugs that only appear with specific sequences.

Implementation Guidance

This section provides concrete code and tools to implement the debugging techniques described above.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Debug Logging	Custom `DEBUG` macro with `printf`	Logging library (e.g., `zlog` for C) with configurable levels
Memory Debugging	Valgrind	AddressSanitizer (ASan) + UndefinedBehaviorSanitizer (UBSan)
Interactive Debugging	GDB with basic commands	GDB with Python scripting for custom pretty-printers
Visualization	Simple text indentation	Graphviz DOT format generation for graphical output

B. Recommended File/Module Structure

Add debugging utilities to your project structure:

```
btree/
  include/
    btree.h          ← Main public interface
    debug.h          ← Debugging macros and utilities
  src/
    btree.c          ← Core B-tree implementation
    debug.c          ← Debugging implementations (validate, print)
  tests/
    test_btree.c     ← Unit and property tests
  examples/
    demo.c           ← Demonstration program
```

C. Infrastructure Starter Code

Here's complete, ready-to-use code for debugging macros and validation helpers:

`include/debug.h`:

```c
#ifndef DEBUG_H

#define DEBUG_H


#include <stdio.h>

#include <stdbool.h>

#include "btree.h"


// Enable/disable debug logging at compile time

// #define DEBUG 1


#ifdef DEBUG

#define LOG(fmt, ...) printf("[%s:%d] " fmt, __func__, __LINE__, ##__VA_ARGS__)

#else

#define LOG(fmt, ...)

#endif


// Assert with custom message

#define ASSERT(cond, msg) \

    do { \

        if (!(cond)) { \

            fprintf(stderr, "Assertion failed: %s (%s:%d)\n", msg, __FILE__, __LINE__); \

            abort(); \

        } \

    } while(0)


// Function prototypes for debugging utilities

bool btree_validate(const BTree* tree);

void btree_print(const BTree* tree);

void node_print(const BTreeNode* node, int indent);
```

```
#endif // DEBUG_H
```

`src/debug.c`:

```c
#include "debug.h"

#include <stdlib.h>

#include <assert.h>


// Helper to validate a subtree

static bool node_validate(const BTreeNode* node, int t, int depth,

                          int* leaf_depth, int min_key, int max_key,

                          bool is_root) {

    if (!node) return true;



    // Check key count bounds

    if (is_root) {

        if (node->num_keys < 0 || node->num_keys > 2*t-1) {

            LOG("Root key count violation: %d\n", node->num_keys);

            return false;

        }

    } else {

        if (node->num_keys < t-1 || node->num_keys > 2*t-1) {

            LOG("Node key count violation: %d\n", node->num_keys);

            return false;

        }

    }



    // Check keys are sorted

    for (int i = 0; i < node->num_keys - 1; i++) {

        if (node->keys[i] >= node->keys[i+1]) {

            LOG("Keys not sorted at index %d: %d >= %d\n",
```

```
            i, node->keys[i], node->keys[i+1]);
        return false;
    }
}

// Check key range (optional, for thorough validation)
for (int i = 0; i < node->num_keys; i++) {
    if (node->keys[i] <= min_key || node->keys[i] >= max_key) {
        LOG("Key %d out of range (%d, %d)\n", node->keys[i], min_key, max_key);
        return false;
    }
}

if (node->is_leaf) {
    // For leaves, check all children are NULL
    for (int i = 0; i <= node->num_keys; i++) {
        if (node->children[i] != NULL) {
            LOG("Leaf has non-NULL child at index %d\n", i);
            return false;
        }
    }

    // Check leaf depth consistency
    if (*leaf_depth == -1) {
        *leaf_depth = depth;
    } else if (*leaf_depth != depth) {
        LOG("Leaf depth mismatch: %d vs %d\n", *leaf_depth, depth);
```

```
            return false;
        }
        return true;
    } else {
        // Internal node: check child count and pointers
        if (node->num_keys + 1 != node->num_children) {
            LOG("Child count mismatch: keys=%d, children=%d\n",
                node->num_keys, node->num_children);
            return false;
        }

        // Validate each child recursively
        for (int i = 0; i <= node->num_keys; i++) {
            if (node->children[i] == NULL) {
                LOG("Internal node has NULL child at index %d\n", i);
                return false;
            }

            // Determine key bounds for child
            int child_min = (i == 0) ? min_key : node->keys[i-1];
            int child_max = (i == node->num_keys) ? max_key : node->keys[i];

            if (!node_validate(node->children[i], t, depth + 1,
                               leaf_depth, child_min, child_max, false)) {
                return false;
            }
        }
```

```
        return true;
    }
}

bool btree_validate(const BTree* tree) {
    if (!tree || !tree->root) return true;

    int leaf_depth = -1;
    return node_validate(tree->root, tree->t, 0, &leaf_depth,
                         INT_MIN, INT_MAX, true);
}

static void print_node(const BTreeNode* node, int indent) {
    for (int i = 0; i < indent; i++) printf("  ");

    printf("[%s] ", node->is_leaf ? "Leaf" : "Node");
    printf("keys=%d: ", node->num_keys);
    for (int i = 0; i < node->num_keys; i++) {
        printf("%d ", node->keys[i]);
    }
    printf("\n");
}

static void btree_print_recursive(const BTreeNode* node, int indent) {
    if (!node) return;

    print_node(node, indent);
```

```
    if (!node->is_leaf) {
        for (int i = 0; i <= node->num_keys; i++) {
            btree_print_recursive(node->children[i], indent + 1);
        }
    }
}

void btree_print(const BTree* tree) {
    if (!tree) {
        printf("BTree: NULL\n");
        return;
    }

    printf("B-Tree (t=%d, height=%d, total_keys=%zu)\n",
           tree->t, btree_height(tree), btree_size(tree));
    btree_print_recursive(tree->root, 0);
    printf("\n");
}
```

D. Core Logic Skeleton Code

For the main operations, add debugging hooks. Here's how to instrument `btree_insert`:

```c
bool btree_insert(BTree* tree, int key, void* value) {
    LOG("Inserting key=%d\n", key);

    // TODO 1: Check for duplicate key (if desired)
    // SearchResult res = btree_search(tree, key);
    // if (res.found) return false; // or replace value

    // TODO 2: If root is full, split it
    if (node_is_full(tree->root, tree->t)) {
        LOG("Root is full, splitting\n");
        // Create new root
        BTreeNode* new_root = node_create(tree->t, false);
        new_root->children[0] = tree->root;
        node_split_child(new_root, 0, tree->t);
        tree->root = new_root;
    }

    // TODO 3: Call insert_non_full on the root
    bool result = insert_non_full(tree->root, key, value, tree->t);

    // TODO 4: Validate tree in debug mode
    #ifdef DEBUG
    if (!btree_validate(tree)) {
        fprintf(stderr, "Invariant violated after insertion of key=%d\n", key);
        btree_print(tree);
        abort();
    }
```

```
    #endif


    LOG("Insertion %s\n", result ? "succeeded" : "failed");
    return result;
}
```

E. Language-Specific Hints (C)

- **Use `assert` liberally:** Include `<assert.h>` and use `assert(condition)` to catch logic errors early. Assertions can be disabled in release builds with `-DNDEBUG`.
- **Compile with debugging symbols:** Always use `-g` flag when compiling debug builds: `gcc -g -O0 -DDEBUG -o btree btree.c debug.c`
- **Enable all warnings:** Use `-Wall -Wextra -Werror` to catch potential issues at compile time.
- **Use `typedef` for function pointers:** If you need callback functions for tree traversal or value freeing, use typedefs for clarity.
- **Memory initialization:** Always initialize allocated memory with `calloc` or explicit loops. Uninitialized pointers cause non-deterministic bugs.

F. Debugging Tips Table

Symptom	Likely Cause	How to Diagnose	Fix
Valgrind reports "invalid read" at `node_find_key_index`	Reading beyond allocated `keys` array in binary search.	Add logging to print `num_keys` and the search bounds. Use GDB to break at that function and examine the array.	Ensure binary search limits are `[0, num_keys)` for key comparisons, and `num_keys` is correctly maintained.
Tree prints show duplicate keys at different levels	During split, the median key was not removed from the child node, or was inserted twice.	Print the state of both child nodes and the parent before and after split.	In `node_split_child`, after promoting the median key, reduce the original child's `num_keys` to `t-1` (removing keys from index `t` onward).
Deletion causes underflow even after borrow/merge	The `strengthen_child` function is not called during descent, or the wrong sibling is chosen for borrowing.	Add logging to show which child is strengthened, which sibling is chosen, and why.	Ensure you check left sibling first for borrowing, then right sibling. The sibling must have at least `t` keys to borrow.
Height increases but root has only one key	This is correct after a root split. If height increases multiple times unnecessarily, the root split logic may be called when root is not full.	Check the condition before splitting root: `if (node_is_full(tree->root, tree->t))`.	Only split the root when it's full (`num_keys == 2*t-1`).
Segmentation fault in `node_destroy`	Double-free or accessing freed memory. Child pointers might be invalid.	Run with Valgrind to see invalid access. Check that `node_destroy` only frees child pointers for internal nodes.	In `node_destroy`, check `if (!node->is_leaf)` before iterating through children. Set pointers to NULL after freeing.

G. Milestone Checkpoint for Debugging

After implementing each milestone, run these debugging checks:

Milestone 1 (Node Structure):

- Run: `./test_btree --test-node-creation`
- Expected: No memory leaks (Valgrind clean), nodes created with correct capacity.
- Verify: Use a debugger to inspect a node's memory layout. Check that `keys` and `children` arrays are properly allocated.

Milestone 2 (Search):

- Run: `./test_btree --test-search`
- Expected: Search finds all inserted keys, returns correct "not found" for absent keys.
- Debugging: Insert 10 keys, print tree, then search for each. Add logging to trace the search path.

Milestone 3 (Insert with Split):

- Run: `./test_btree --test-insert`
- Expected: Tree maintains all invariants after each insertion. Root splits when appropriate.
- Debugging: Insert exactly `2t-1` ascending keys and validate tree after each insertion. Check that height increases by 1 only when root splits.

Milestone 4 (Delete with Rebalancing):

- Run: `./test_btree --test-delete`
- Expected: All keys remain findable after random insert/delete sequences. Tree height may decrease.
- Debugging: Create a known tree structure, delete a specific key that triggers borrow, then merge. Print tree before and after each operation.

When something goes wrong, use this systematic approach:

1. **Reproduce minimally:** Create the smallest test case that triggers the bug.
2. **Add instrumentation:** Enable debug logging around the failing operation.
3. **Validate invariants:** Call `btree_validate` before and after the operation.
4. **Inspect state:** Use `btree_print` to visualize the tree structure.
5. **Debug step-by-step:** Use GDB to step through the failing function.
6. **Check memory:** Run Valgrind to catch memory corruption.

Remember, B-tree bugs often manifest far from their source—a missing key might result from a split that happened hours of operations earlier. The invariant checker is your best defense against these subtle, cumulative errors.

13. Future Extensions

> **Milestone(s):** This section builds upon all four milestones (Node Structure, Search, Insert with Split, Delete with Rebalancing) and explores how the foundational B-tree implementation can be extended for real-world use cases beyond the educational scope.

The current B-tree implementation provides a solid in-memory foundation that correctly maintains all structural invariants. However, production database systems and file systems extend this core design in several critical directions. This section outlines potential enhancements, explaining how each would modify the architecture, the technical challenges involved, and the benefits they would provide. These extensions transform the academic B-tree into a practical component suitable for building storage engines, indexing systems, and other disk-resident data structures.

13.1 Extension Ideas

The following extensions represent natural evolution paths for the B-tree implementation. Each addresses a specific limitation of the current design for production environments.

13.1.1 Persistence to Disk (Disk I/O Layer)

Mental Model: The Filing Cabinet System Think of the current in-memory B-tree as a temporary workspace on a desk—fast to access but lost when the power goes off. Adding disk persistence transforms it into a filing cabinet system: each node corresponds to a labeled folder (disk page) stored in a drawer (disk). To work with a document, you must first retrieve the entire folder from the cabinet (disk read), make changes, and then file it back (disk write). The cabinet's organization (the B-tree structure) survives power cycles, but accessing folders is slower than grabbing papers from your desk.

Current Limitation: The implementation stores all nodes in volatile RAM using `malloc`. A system crash or program termination loses all data.

Proposed Extension: Add a disk storage layer where each `BTreeNode` corresponds to a fixed-size disk page (e.g., 4KB). The `BTree` structure would manage a cache of recently accessed pages in memory and a mapping from node identifiers to disk locations.

Key Design Changes:

1. **Page-Oriented Node Allocation:** Replace pointer-based child references (`BTreeNode** children`) with disk page IDs (e.g., `uint64_t`). Each node would have a unique `page_id`.
2. **Buffer Pool Manager:** Introduce a fixed-size cache (buffer pool) of loaded pages. The `btree_search/insert/delete` operations would request nodes from the buffer manager, which handles loading from disk and writing dirty pages back.
3. **Serialization/Deserialization:** Add functions `node_serialize` and `node_deserialize` to convert between in-memory `BTreeNode` structures and raw byte arrays (disk pages).

4. **Free Space Management:** Track unused pages (e.g., with a free list or bitmap) to reuse space from deleted nodes.
5. **Root Pointer Persistence:** Store the `page_id` of the root node at a fixed location on disk (e.g., the first 8 bytes of the file) to bootstrap the tree.

> **ADR: Page-Oriented vs. Log-Structured Node Storage**
>
> - **Context:** Need to persist B-tree nodes to disk with efficient read/write patterns.
> - **Options Considered:**
>   1. **Page-Oriented (Traditional):** Each node occupies a fixed-size disk page at a stable location. Updates overwrite the page in-place.
>   2. **Log-Structured (Append-Only):** All node modifications are appended to a sequential log. A separate in-memory table maps node IDs to their latest log offsets.
> - **Decision:** Page-oriented storage for initial persistence extension.
> - **Rationale:** Page-oriented design is simpler to implement and matches the classic B-tree assumption of fixed-size disk blocks. It allows direct access to any node given its page ID without scanning a log. Overwrite-in-place is acceptable for a single-threaded implementation.
> - **Consequences:** Enables traditional database-style B-tree persistence. Requires careful handling of crash consistency (e.g., write-ahead logging for atomic operations) and fragmentation management over time.

Comparison of Disk Storage Approaches:

Approach	Pros	Cons	Best For
Page-Oriented (Fixed Location)	• Direct node access by page ID • Simple implementation • Matches textbook B-tree model	• Fragmentation from deletions • Crash consistency requires additional logging (WAL) • Random writes may be slow on HDDs	Traditional databases, educational clarity
Log-Structured (Append-Only)	• Sequential writes (faster on HDD/SSD) • Built-in versioning/history • Simplifies crash recovery (no torn pages)	• Requires garbage collection (compaction) • Need mapping table in memory (or itself logged) • Extra indirection for reads	Write-intensive workloads, SSDs, experimental storage engines

Implementation Considerations:

- **Page Size:** Typically 4KB (file system block size) or larger (8KB, 16KB). Must fit `(2t-1)` keys, `(2t)` child pointers, and metadata.
- **Cache Eviction Policy:** Least Recently Used (LRU) is a common choice for the buffer pool.
- **Concurrency:** Disk I/O operations should be non-blocking (asynchronous) to avoid stalling the main thread during page loads.

Common Challenges:

- ⚠️ **Pitfall: Ignoring Alignment** — Disk pages must be aligned to block boundaries. Serializing structures with `sizeof(BTreeNode)` may not produce the exact page size, leading to wasted space or read errors.
- ⚠️ **Pitfall: No Crash Consistency** — Writing a split node without ensuring the parent is updated can corrupt the tree after a crash. Requires a write-ahead log (WAL) for atomic multi-page operations.

13.1.2 Key-Value Storage with Associated Values

Mental Model: The Library Card Catalog The current B-tree acts like a simple index of book accession numbers (keys). Adding associated values transforms it into a full card catalog: each card (node entry) contains both the accession number (key) and detailed information about the book—title, author, location (value). The tree organizes the cards by accession number for quick lookup, but the valuable data is the book information attached to each key.

Current Limitation: The `BTreeNode` structure includes a `values void**` array, but the core algorithms ignore it. The `btree_insert` accepts a `value void*` but doesn't store or retrieve it.

Proposed Extension: Fully integrate the value pointers into all operations. Each key in a leaf node would have an associated value pointer. Internal nodes would continue to have `NULL` values for their keys (which are only separators).

Key Design Changes:

1. **Value Storage in Leaves:** Modify `node_insert_key` and `node_remove_key` to handle the parallel `values` array. When inserting a key at index `i`, also insert the value pointer at the same index in the `values` array.
2. **Search Returns Value:** Update `btree_search` and `search_recursive` to return the `value void*` in the `SearchResult` when a key is found at a leaf.
3. **Value Propagation During Splits:** When splitting a leaf node, copy value pointers along with keys to the new node. For internal node splits, values for the promoted key are irrelevant (can be `NULL`).
4. **Value Deletion:** Ensure `btree_delete` properly handles the value pointer—either freeing it if the tree owns the memory, or returning it to the caller for external management.

> **ADR: Inline Values vs. Indirect Storage**
>
> - **Context:** Need to store variable-sized or large values associated with keys.
> - **Options Considered:**
>   1. **Inline Storage (Current):** Store value pointers directly in the node's `values` array. The pointer points to external memory.
>   2. **Indirect Storage (Value Log):** Store values in a separate append-only log or heap file. The node stores only a small value handle (e.g., offset in log) instead of a pointer.
> - **Decision:** Keep inline storage with `void*` pointers for simplicity in the initial extension.
> - **Rationale:** Direct pointers are easiest to implement and understand. They allow the B-tree to store references to any data structure. The caller manages memory allocation for values, keeping the B-tree focused on structure.
> - **Consequences:** Values must fit in memory. The tree doesn't handle variable-sized values efficiently (all pointers are same size). For disk persistence, values would need separate storage strategy.

Comparison of Value Storage Strategies:

Strategy	Pros	Cons	Best For
Inline Pointers (Current)	• Simple implementation • Fast in-memory access • Can point to any data type	• Not disk-persistent directly • Variable-sized values waste space or require separate allocation • Pointers become invalid after program restart	In-memory caches, indices over existing objects
Separate Value Log	• Values can be variable-sized • Efficient disk persistence (sequential writes) • Can compress values independently	• Extra indirection (slower) • Requires garbage collection for deleted values • More complex to implement	Database storage engines, key-value stores (e.g., RocksDB)

Implementation Considerations:

- **Memory Management:** Who owns the value memory? The B-tree could adopt values (free on delete) or just store references (caller manages). A hybrid approach with a `free_value_fn` callback is flexible.
- **Duplicate Keys:** The B-tree typically assumes unique keys. If duplicates are allowed, values must be stored for each duplicate, possibly as a list per key.

Common Challenges:

- ⚠️ **Pitfall: Value Pointer Misalignment** — When splitting or merging nodes, forgetting to copy/move value pointers along with keys corrupts the value-key association.
- ⚠️ **Pitfall: Memory Leaks** — Deleting a key without freeing its associated value (if the tree owns it) leaks memory. Similarly, overwriting a key during insertion might leak the old value.

13.1.3 Concurrency Control (Multi-Threading)

Mental Model: The Library with Multiple Librarians Imagine a library where several librarians (threads) can simultaneously help patrons. Without coordination, two librarians might try to update the same catalog drawer (node) at once, leading to chaos. Concurrency control provides a system like "sign-out sheets" for drawers: a librarian must acquire a lock on a drawer before modifying it, and perhaps follow a protocol to prevent deadlock (e.g., always request locks top-down).

Current Limitation: The implementation is single-threaded. Concurrent `insert` and `search` operations from multiple threads would cause data races, leading to corruption or crashes.

Proposed Extension: Add fine-grained locking to allow safe concurrent operations. Common approaches include:

- **Readers-Writer Locks (Pessimistic):** Each node has a read-write lock. Searches acquire read locks; insertions/deletions acquire write locks on nodes along their path.
- **B-link Tree (Optimistic):** A variant where nodes have a "high key" and a link pointer to the right sibling. Searches can proceed without locks; insertions use careful atomic operations and retry mechanisms.
- **Copy-on-Write (Persistent):** Each modification creates new versions of nodes along the path, enabling snapshot isolation and lock-free reads.

Key Design Changes:

1. **Node-Level Locking:** Add a `pthread_rwlock_t` (or equivalent) field to `BTreeNode`. All operations must acquire appropriate locks as they traverse.
2. **Lock Coupling (Crabbing):** To prevent deadlock and ensure consistency, hold the lock on a parent while acquiring the lock on a child. Release parent lock after child is locked (for writes).
3. **Deadlock Avoidance:** Always acquire locks in a consistent order (e.g., top-down, left-to-right).
4. **Root Handling:** The root may change during splits. Need a special lock or version number for the root pointer.

ADR: Fine-Grained Locking vs. B-link Tree

- **Context:** Need to support concurrent operations without sacrificing correctness.
- **Options Considered:**
  1. **Fine-Grained Read-Write Locks:** Traditional approach using lock coupling (crabbing). Simple to understand but can lead to contention at high levels.
  2. **B-link Tree with Optimistic Concurrency:** A B-tree variant with sibling links that allows lock-free searches and non-blocking inserts using atomic compare-and-swap (CAS) operations.
- **Decision:** Start with fine-grained read-write locks for educational purposes.
- **Rationale:** Lock coupling directly extends the existing algorithms with clear critical sections. It's easier to implement and debug than B-link trees, which require significant algorithmic changes.
- **Consequences:** Provides thread safety but may limit scalability under high contention. Deadlock risk requires careful lock ordering.

Comparison of Concurrency Approaches:

Approach	Pros	Cons	Best For
Fine-Grained Read-Write Locks	• Conceptually straightforward • Minimal changes to existing algorithms • Well-understood deadlock avoidance protocols	• Lock overhead per node • Contention at root and hot nodes • Deadlock risk if ordering violated	Moderate concurrency, educational implementation
B-link Tree (Optimistic)	• Lock-free reads (scalable) • Non-blocking inserts • Avoids deadlock by design	• Complex to implement correctly • Requires atomic operations (CAS) • Algorithm differs from classic B-tree	High-concurrency databases (e.g., LMDB, WiredTiger)
Copy-on-Write (Persistent)	• Naturally provides snapshot isolation • Readers never block writers • Good for versioned data	• High write amplification • Requires garbage collection of old nodes • Memory/disk overhead	Multi-version systems, temporal databases, file systems (ZFS, Btrfs)

Implementation Considerations:

- **Lock Granularity:** Locking at the node level is fine-grained but adds memory overhead. Coarser granularity (e.g., locking entire tree) would destroy concurrency.
- **Recovery from Deadlocks:** Use timeout mechanisms or deadlock detection if not strictly ordering locks.
- **Performance:** Under high contention, the root node becomes a bottleneck. Techniques like root splitting or root versioning can help.

Common Challenges:

- ⚠️ **Pitfall: Forgotten Unlock** — In complex recursive functions with multiple return paths, failing to unlock a node before returning leads to deadlock.
- ⚠️ **Pitfall: Lock Ordering Violation** — Acquiring locks on two children in arbitrary order (e.g., during borrowing from sibling) can cause deadlock. Must always lock left-to-right or right-to-left.

13.1.4 B+ Tree and B* Tree Variants

Mental Model: Specialized Library Catalog Systems

- **B+ Tree:** Imagine a library catalog where the main index (internal nodes) contains only guide keys, and all actual book records (values) are stored in a separate, linked set of leaf pages. This is like having an index volume that points to specific shelves where books are stored contiguously.
- *B Tree:** Imagine a library that avoids creating new shelves until absolutely necessary. When a shelf becomes full, instead of immediately splitting it, librarians first try to redistribute books to neighboring shelves. Only when three adjacent shelves are all full do they split two of them into three. This maximizes space utilization.

Current Limitation: The implementation is a classic B-tree where values *could* be stored in internal nodes (though our design leaves them NULL). This is less efficient for range scans and has lower node occupancy on average than B* trees.

Proposed Extensions:

B+ Tree Modifications:

1. **Leaf Node Linking:** Add `BTreeNode* next` pointer to each leaf node, creating a linked list of all leaves in key order.
2. **Value Storage Only in Leaves:** Internal nodes store only keys and child pointers—no value pointers. All values reside in leaf nodes.
3. **Search Always Goes to Leaf:** Even if a key is found in an internal node during search, continue descending to the leaf to retrieve the associated value.
4. **Range Scans:** Implement `btree_range_scan(start_key, end_key)` that finds the leaf for `start_key` and traverses the leaf linked list until exceeding `end_key`.
5. **Split Differences:** When splitting a leaf, update the `next` pointers accordingly. The promoted key to the parent is the *first key of the right sibling* (not necessarily a key that exists in the left sibling).

*B Tree Modifications:**

1. **Higher Minimum Occupancy:** Instead of splitting a node when it reaches `2t-1` keys, try to redistribute keys to a sibling first. Only split when both the node and its immediate sibling are full.
2. **Three-Node Split:** When splitting is unavoidable, involve two full siblings and distribute keys among three nodes (two original plus one new), leading to better space utilization (nodes are at least 2/3 full).
3. **Modified Insertion Algorithm:** During descent, if a child is full, first check if an adjacent sibling has space. If yes, redistribute keys between the child, sibling, and parent. If no adjacent sibling has space, then perform a split that involves the child and one sibling.

> **ADR: B+ Tree vs. Classic B-tree for Range Queries**
>
> - **Context:** Need efficient range scans (e.g., "find all keys between 100 and 200") and sequential access.
> - **Options Considered:**
>   1. **Classic B-tree:** Values can be anywhere. Range scans require complex traversal jumping between internal and leaf nodes.
>   2. **B+ Tree:** All values in leaves, with leaves linked sequentially.
> - **Decision:** Implement B+ tree extension if range queries are important.
> - **Rationale:** B+ trees are the standard for database indices because they optimize for both random lookups and range scans. The leaf linkage makes scanning orders of magnitude faster (no need to backtrack up the tree).
> - **Consequences:** Slightly more complex insertion/deletion due to leaf linking. Internal nodes store only keys (smaller, higher fanout). Duplicate key handling is cleaner (all duplicates go to leaves).

Comparison of B-tree Variants:

Variant	Key Characteristics	Advantages	Disadvantages
Classic B-tree	Keys and values may be in internal nodes	• Simpler algorithms • Can store "hot" data higher in tree	• Less efficient for range scans • Lower fanout if values are large
B+ Tree	All values in leaves, leaves linked	• Excellent for range scans/sequential access • Higher fanout in internal nodes • Predictable access pattern (always to leaf)	• Extra pointer per leaf • Two different node types (internal vs leaf)
*B Tree**	Redistributes before splitting, nodes ≥ 2/3 full	• Higher space utilization (less wasted disk) • Fewer splits, potentially lower height	• More complex insertion logic • Redistribution overhead

Implementation Considerations:

- **Backward Compatibility:** Could maintain the same external API (`btree_insert` , `btree_search`) but change internal behavior. Range scan would be a new API.
- **Hybrid Approach:** Allow configuration at tree creation time (`btree_create(t, variant)`).
- **Testing:** Each variant requires specific test cases for redistribution and linking.

Common Challenges:

- ⚠️ **Pitfall: Broken Leaf Links** — During split or merge of leaves, incorrectly updating `next` pointers breaks the sequential scan.
- ⚠️ **Pitfall: Redistribution Complexity** — B* tree redistribution requires careful handling of keys and children across three nodes (two siblings and parent). Off-by-one errors are common.

13.2 Additional Extension Opportunities

Beyond the major extensions above, several other enhancements could improve utility or performance:

- **Bulk Loading:** Construct a B-tree efficiently from a pre-sorted set of keys (e.g., from a CSV file) by building the tree bottom-up rather than repeated insertion. This produces a perfectly balanced tree with higher leaf occupancy and is much faster.
- **Compression:** Store keys in nodes using prefix compression (especially useful for string keys) to increase effective fanout and reduce I/O.
- **Variable-Length Keys:** Support keys of arbitrary length (e.g., strings) by storing them within the node's byte array with offset pointers, rather than fixed-size `int` keys.

- **Transactional Support:** Group multiple insert/delete operations into atomic transactions with rollback capability, using an undo log.
- **Monitoring and Metrics:** Expose internal statistics (node counts, split/merge frequencies, average occupancy) for performance tuning.

13.3 Implementation Guidance

> **Note:** The following guidance provides starting points for the most practical extension: adding associated values and simple persistence. These are the most likely next steps for a learner wanting to build a usable key-value store.

Technology Recommendations:

Component	Simple Option	Advanced Option
Persistence Layer	Fixed-size binary file with direct I/O (`fread` / `fwrite`)	Memory-mapped files (`mmap`) with page fault handling
Concurrency Control	Single global mutex for entire tree (`pthread_mutex`)	Fine-grained per-node read-write locks (`pthread_rwlock_t`)
Value Storage	`void*` pointers to caller-managed memory	Separate value log file with offset-based handles

Recommended File Structure for Extended Project:

```
btree/
├── include/
│   └── btree.h                # Public API (unchanged)
├── src/
│   ├── btree.c                # Core B-tree algorithms (modified for values)
│   ├── node.c                 # Node operations
│   ├── disk_pager.c           # NEW: Buffer pool and disk I/O
│   ├── value_log.c            # NEW: Optional value storage
│   └── concurrent.c           # NEW: Locking wrappers (if adding concurrency)
├── tests/
│   └── test_extended.c        # Tests for extended features
└── tools/
    └── btree_dump.c           # Utility to inspect on-disk structure
```

Infrastructure Starter Code (Disk Pager Header):

```c
// disk_pager.h - Simple disk persistence layer

#ifndef DISK_PAGER_H

#define DISK_PAGER_H


#include <stdint.h>

#include <stdbool.h>


#define PAGE_SIZE 4096

#define INVALID_PAGE_ID UINT64_MAX


typedef uint64_t page_id_t;


// Disk page representation (matches node serialized size)

typedef struct {

    uint8_t bytes[PAGE_SIZE];

} disk_page_t;


// Buffer pool entry

typedef struct bp_entry {

    page_id_t page_id;

    disk_page_t page;

    bool is_dirty;

    struct bp_entry* next;   // For LRU list

    struct bp_entry* prev;

} bp_entry_t;


// Main pager structure

typedef struct {

    int fd;                    // File descriptor for database file

    uint64_t num_pages;        // Total pages in file
```

```
    bp_entry_t* lru_head;       // Most recently used
    bp_entry_t* lru_tail;       // Least recently used
    bp_entry_t* entries;        // Array of all buffer entries
    int num_entries;            // Size of buffer pool
} disk_pager_t;

// Public interface
disk_pager_t* pager_open(const char* filename, int cache_pages);
void pager_close(disk_pager_t* pager);
disk_page_t* pager_fetch(disk_pager_t* pager, page_id_t page_id);
void pager_mark_dirty(disk_pager_t* pager, page_id_t page_id);
page_id_t pager_allocate_page(disk_pager_t* pager);
void pager_deallocate_page(disk_pager_t* pager, page_id_t page_id);
page_id_t pager_get_root_page(disk_pager_t* pager);
void pager_set_root_page(disk_pager_t* pager, page_id_t root_id);

#endif // DISK_PAGER_H
```

Core Logic Skeleton (B-tree with Values and Disk Integration):

```c
// Modified BTree structure with disk pager
typedef struct {
    page_id_t root_page_id;   // Disk page ID of root (instead of pointer)
    int t;                    // Minimum degree
    size_t key_count;         // Total keys (cached for O(1) access)
    disk_pager_t* pager;      // Disk pager instance
} BTree;

// Public API with values (unchanged signatures, but now values are stored)
BTree* btree_create(const char* filename, int min_degree);
SearchResult btree_search(BTree* tree, int key);
bool btree_insert(BTree* tree, int key, void* value);
bool btree_delete(BTree* tree, int key);

// Internal function now works with page IDs
static BTreeNode* node_load(BTree* tree, page_id_t page_id) {
    // TODO 1: Call pager_fetch to get disk page for page_id
    // TODO 2: Deserialize disk page bytes into a BTreeNode structure
    // TODO 3: Return the in-memory node (caller must eventually release)
}

static void node_save(BTree* tree, BTreeNode* node, page_id_t page_id) {
    // TODO 1: Serialize BTreeNode into disk_page_t.bytes
    // TODO 2: Call pager_mark_dirty so page gets written back eventually
    // TODO 3: If this is a new node, assign it a page_id via pager_allocate_page
}

// Example modified search recursive function
```

```
static SearchResult search_recursive_disk(BTree* tree, page_id_t current_page_id, int key)
{

    // TODO 1: Load node using node_load(tree, current_page_id)

    // TODO 2: Perform binary search within node's keys

    // TODO 3: If key found and node is leaf, return value from node->values[index]

    // TODO 4: If key not found and node is leaf, return not found

    // TODO 5: If internal node, determine child index and recursively call
search_recursive_disk

    // TODO 6: Release node from buffer pool (implementation dependent)

}
```

Language-Specific Hints (C):

- Use `open()` with `O_DIRECT` flag for aligned direct I/O (bypasses kernel cache, more realistic for database engines).
- For serialization, consider using `memcpy()` between struct fields and byte arrays, but beware of padding bytes. Explicitly pack structures with `#pragma pack(1)` or manual byte-by-byte copying.
- Memory management for values: Provide a callback registration mechanism: `btree_set_free_value_callback(BTree*, void (*free_fn)(void*))` to allow custom cleanup.
- For concurrency, use `pthread_rwlock_init` in `node_create` and `pthread_rwlock_destroy` in `node_destroy`.

Milestone Checkpoint for Persistence Extension:

1. **Implement pager with LRU cache:** Test by creating a file, allocating pages, and reading them back.
2. **Add node serialization:** Write a test that creates a small tree in memory, serializes all nodes to disk, then reloads and verifies structure with `btree_validate`.
3. **Integrated operations:** After full integration, run a sequence of insertions, persist, close/reopen the file, and verify searches return correct values. Use a tool like `hexdump` to inspect the raw disk file.

Debugging Tips for Disk Extensions:

Symptom	Likely Cause	How to Diagnose	Fix
"Read wrong data after reload"	Serialization/deserialization mismatch	Add debug prints showing byte-by-byte comparison of original vs loaded node	Ensure all fields (especially child pointers converted to page IDs) are serialized
"Segfault in buffer pool"	LRU list corruption	Add assertions after each list operation; draw diagram of list before/after each fetch	Carefully maintain `next` / `prev` pointers in doubly-linked LRU
"File size grows uncontrollably"	Never deallocate pages on delete	Log each `pager_allocate_page` and `pager_deallocate_page` call	Implement free page list in pager header page
"Value pointer becomes invalid after reload"	Storing raw pointers instead of data	Pointers are memory addresses; they have no meaning after restart	For persistence, store values inline or in separate value log with offsets

14. Glossary

Milestone(s): Milestones 1-4 (all operations)
This section provides concise definitions for key terminology used throughout the B-tree design document, serving as a quick reference for readers encountering unfamiliar terms. Each entry references the primary section where the concept is first introduced or explained in detail.

Term Definitions

Term	Definition	Primary Reference
Binary search	An algorithm for finding a target value within a sorted array by repeatedly dividing the search interval in half. In B-trees, used within a node's key array to efficiently locate a target key or determine which child to traverse to.	Section 6: Component Design: Search Operation
Borrowing	An operation during deletion that transfers a key from a sibling node through the parent to fix an underflow condition in a node. This redistributes keys without merging nodes, preserving tree structure when possible.	Section 8: Component Design: Deletion with Rebalancing
B+ tree	A B-tree variant where all data values are stored only in leaf nodes, and internal nodes contain only keys for routing. Leaf nodes are typically linked together to support efficient range scans.	Section 13: Future Extensions
B* tree	A B-tree variant that delays splitting by first attempting to redistribute keys to neighboring nodes when a node becomes full, resulting in higher average node occupancy.	Section 13: Future Extensions
Buffer pool	A cache of recently accessed disk pages maintained in memory, used in persistent storage implementations to reduce disk I/O by keeping frequently accessed nodes available without reading from disk each time.	Section 13: Future Extensions (Future Extensions)
B-tree	A self-balancing tree data structure optimized for systems that read and write large blocks of data (like databases and file systems). Maintains sorted data and allows search, insertion, and deletion in logarithmic time while minimizing disk accesses through high fanout.	Section 1: Context and Problem Statement
Capacity bounds	The minimum and maximum number of keys allowed in a B-tree node based on the minimum degree t. Every node except the root must contain between t-1 and 2t-1 keys. The root may have as few as 1 key.	Section 5: Component Design: B-tree Node
Disk block/page	The unit of data transfer between disk and memory, typically 4KB. B-tree nodes are designed to align with these fixed-size blocks to optimize I/O efficiency.	Section 1: Context and Problem Statement
Fanout	The number of children per internal node in a tree. In B-trees, the fanout is typically high (dozens to hundreds) because each node holds many keys, which reduces tree height and disk seeks.	Section 4: Data Model

Term	Definition	Primary Reference
Height	The number of levels from the root node to the leaf nodes in a tree. B-trees maintain logarithmic height relative to the total number of keys, ensuring efficient operations.	Section 4: Data Model
In-order predecessor	The largest key in the left subtree of a given key. Used during deletion from internal nodes when replacing the target key before recursively deleting the replacement.	Section 8: Component Design: Deletion with Rebalancing
In-order successor	The smallest key in the right subtree of a given key. Alternative to the predecessor for internal node deletion.	Section 8: Component Design: Deletion with Rebalancing
Invariants	Mathematical rules that must always hold true for the data structure to be valid. B-tree invariants include key count bounds, sorted order of keys within nodes, and balanced height of all leaf nodes.	Section 4: Data Model
Lock coupling	A concurrency control technique where a thread holds a lock on a parent node while acquiring a lock on a child node, preventing deadlock in multi-threaded B-tree implementations.	Section 13: Future Extensions
Minimum degree (t)	The fundamental parameter defining capacity bounds in a B-tree. Each node (except root) must have at least t-1 keys and at most 2t-1 keys. The value t determines the tree's fanout and balance properties.	Section 5: Component Design: B-tree Node
Node access	The act of reading a node from disk or memory. A key performance metric for B-trees, as operations aim to minimize node accesses (especially disk I/O) by keeping tree height low.	Section 6: Component Design: Search Operation
Node merging	An operation during deletion that combines two underfull sibling nodes and the separating key from their parent into one node when borrowing is not possible. Reduces the number of nodes and may propagate up the tree.	Section 8: Component Design: Deletion with Rebalancing
Node splitting	An operation during insertion that divides a full node (with 2t-1 keys) into two nodes, each with t-1 keys, and promotes the middle key to the parent node. Maintains capacity bounds when inserting into a full node.	Section 7: Component Design: Insertion with Splitting
Opaque pointer	A pointer to an incomplete (forward-declared) type that hides implementation details from the user, promoting encapsulation. Used in the public API to abstract the internal `BTreeNode` structure.	Section 3: High-Level Architecture

Term	Definition	Primary Reference
Page-oriented storage	A persistence model where each B-tree node occupies a fixed-size disk block (page) at a stable location, enabling direct mapping between node pointers and disk addresses.	Section 5: Component Design: B-tree Node (Milestone 1 Deliverables)
Proactive splitting	A strategy where full nodes are split during the downward traversal for insertion, before the actual insertion occurs. This ensures that any node we descend into has room for an extra key if needed.	Section 7: Component Design: Insertion with Splitting
Range scan	An operation that retrieves all keys and their associated values within a specified interval [start_key, end_key]. Efficiently supported in B+ trees through linked leaf nodes.	Section 13: Future Extensions
Recursive traversal	A method of traversing tree structures where a function calls itself to process subtrees. Used in B-tree operations to navigate from the root to leaves.	Section 6: Component Design: Search Operation
Strengthening	The process of proactively ensuring a child node has at least *t* keys before descending into it during deletion. Achieved through borrowing from a sibling or merging with a sibling, preventing underflow during the recursive deletion process.	Section 8: Component Design: Deletion with Rebalancing
Underflow	A condition where a node has fewer than *t*-1 keys, violating the B-tree invariant. Occurs after deletion and must be resolved through borrowing or merging to restore the minimum occupancy requirement.	Section 8: Component Design: Deletion with Rebalancing