Assignment 2

Mutating Your Red-Black Tree Under Kernel-Style Constraints

Custom allocators, allocation that fails on purpose, $O(1)$-space teardown, and disciplined AI-assisted development

Version 1.0 September 16, 2026

Version history appears in Section 20.

Contents

Nota Bene: This is the second half of a pair, and the half that does the damage. In Assignment 1 you built a red-black tree in C and learned to account for every byte of it. Nothing about that tree changes here: same API, same invariants, same answers. What changes is the ground it stands on. Three *mutations* follow, and each one takes away something the baseline quietly assumed: that allocation always succeeds, that nodes may come from `malloc`, and that there is a stack deep enough to recurse on. Your Assignment 1 test suite comes with you and is now a regression suite; every mutation has to leave it green. The effort compounds, too. And, as in Assignment 1, an AI coding agent (Claude Code) is not merely tolerated but *required*: working with it is woven into both the workflow you will follow and the grade you will earn. The assignments are commensurately harder from here on.

1 Overview

Everything in this assignment orbits a single data structure. A red-black tree is a small thing to describe and a demanding thing to keep alive. Every node is a separately allocated heap object stitched to its neighbors by pointers. Every insertion may allocate several times and then rebalance by rewiring pointers far from the insertion point. Every deletion must repair global invariants while releasing memory in exactly the right order. That combination makes the red-black tree an ideal specimen for studying memory management, which is the part this course actually cares about.

You built the tree once, in Assignment 1. Here you keep it correct while the assignment repeatedly changes the rules about *where* its bytes live and *how* they may be reclaimed. By the end, the logical tree is unchanged: same API, same invariants, same answers. Everything underneath it has been replaced. If you finish with the sense that the balancing algorithm was the easy part, the assignment has worked. The structure matters beyond the assignment's perimeters, too. The Linux kernel ships its own red-black tree library and leans on it wherever predictable $O(\log n)$ behavior and tight memory control must coexist: most famously in the CFS scheduler's runqueue (one of the crown jewels of the CPU-scheduling module, where we will study it in depth) and, for several years, in tracking the memory regions of every process's address space.

The core theme here is *discipline under a moving target*. Writing a red-black tree that returns the right answer is fairly straightforward, and you have already done it. Keeping that tree correct when allocation can fail at an arbitrary point, when nodes must come from your own allocator, when there is no stack to recurse on, and when two logical trees quietly share the same bytes: that is the harder skill, and the one this assignment is actually grading. A tree that looks up the right key but leaks, double-frees, or recurses unboundedly fails. Every correct answer it returns is beside the point.

Grading methodology and the Quiz that you can't Google

This assignment is graded out of **15 points**, and those points answer two different questions, calling for two different kinds of evidence.
12 points answer the question the tools already ask: *does the program work, and does it work without corrupting memory*? That question is settled for you, mechanically and without opinion, by the test batteries, the fault-injection sweep, and AddressSanitizer and valgrind.
The remaining **3 points** answer a harder question: *do you understand what you built?* **2 points** come from an analysis of your Claude Code session transcripts: whether they show the short, specific, iterative exchanges of someone building and debugging their own program, or the single large paste of someone outsourcing it. **1 point** comes from a short quiz generated from your own code, administered at a live walkthrough: your own function names, your own error-path structure, why one of your teardown loops cannot run forever, what frees a given allocation on a given failure path. Two students can submit code that behaves identically on every test case and still walk away from that quiz with very different scores, because the quiz was never testing the program. It was testing you.
There is a reason for the split. A working red-black tree is a nice thing to have. A student who can explain, unprompted, why a failed insert must leave the tree byte-for-byte as it was is a nice thing to become. Most of the hours you put into this assignment build the former. The prompt analysis and the quiz at the end are where we find out whether you also built the latter.

2 Learning Objectives

- Separate the logical data structure from its memory model, and hold the former invariant while the latter is replaced: fault-tolerant allocation, a slab pool, $O(1)$-space traversal, and copy-on-write sharing.
- Write allocation-failure-safe code that unwinds cleanly, mirroring kernel paths that must cope with `kmalloc` returning `NULL`.
- Build a slab-style pool allocator with an intrusive free list, and explain what the kernel's version of the same idea adds and why.
- Replace recursion with a walk that borrows the data structure's own pointers for bookkeeping, and state the invariant that makes it terminate.
- Verify memory-management correctness with sanitizers and valgrind, treating a single leaked byte or invalid access as a failure, not a warning.
- Design and defend a C module against a frozen public API: struct layout, ownership contract, and the error-code discipline the grading harness depends on.

- Direct an AI coding agent (specifically, *Claude Code*) the way a senior engineer directs a capable but overconfident junior: give it context, make it plan, keep changes small, verify everything with tools, and merge nothing you cannot explain.

Item	Detail
Prerequisite	Your Assignment 1 tree, passing its own battery. That suite comes with you and becomes the regression suite for everything below
Language	C (C23), compiled with `gcc` and `make`, must build cleanly with `-Wall -Wextra -Werror`
Verification	AddressSanitizer/UBSan (`make asan`) and valgrind (`make memcheck`); a single leak or UB report caps that milestone's correctness at 50%
Expected effort	Approximately 12–16 hours across three milestones, including required use of Claude Code
Tooling	Claude Code is required for this assignment (see Section 9)

3 Where We Left Off

Nothing in this section is new. It is here so that the invariants your mutations must preserve sit on the same desk as the mutations themselves, and so that Figures 1 and 2 are one page-flip away when Mutation 3 asks you to tear a tree down by rotating it flat.

A red-black tree is a binary search tree that keeps itself approximately balanced by attaching one bit of bookkeeping, a *color*, to every node. Five rules govern the colors:

1. Every node is red or black.
2. The root is black.
3. Conceptually, every missing child ends at a black sentinel leaf, written NIL.
4. A red node has only black children, so reds never stack.
5. Every path from a given node down to any NIL leaf crosses the same number of black nodes. This count is called the *black-height*.

Rules 4 and 5 together squeeze the tree's shape. The longest possible root-to-NIL path alternates red and black, while the shortest is all black, so no path can be more than twice as long as any other. From this it follows that the height satisfies $h \leq 2\log_2(n+1)$, and search, insert, and delete all run in $O(\log n)$ worst-case time. The figures here use small integer keys, which make the ordering easy to see at a glance; your implementation stores C-string keys compared with `strcmp`, but the structure and the invariants are identical either way.

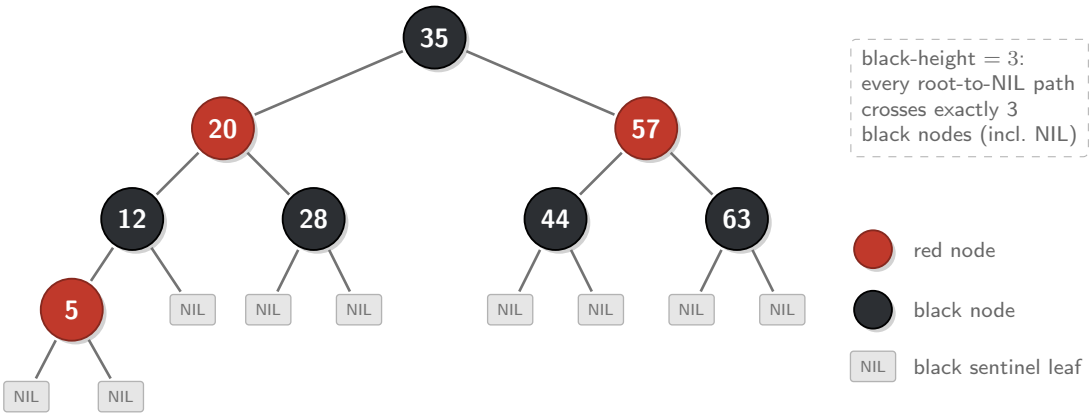


Figure 1: A valid red-black tree over integer keys. No red node has a red child, every root-to-NIL path crosses the same number of black nodes, and in-order traversal yields the keys in sorted order. The path through 5 is the longest and the path to 28's children is among the shortest, but the two differ in length by less than a factor of two.

Balance is maintained, not assumed. An insertion places a new red node at a leaf position, which may create a red-red violation; a deletion may remove a black node, which breaks the black-height rule. Both are repaired by walking toward the root with two local tools. *Recoloring* flips colors and

moves no pointers. *Rotation* (Figure 2) is a constant-time pointer surgery that lifts a child over its parent while preserving the in-order sequence. Insert fixup needs at most two rotations; delete fixup needs at most three. Keep the rotation picture in mind: Mutation 3 uses the same primitive for a purpose it was never designed for, which is flattening a tree you are in the middle of destroying.

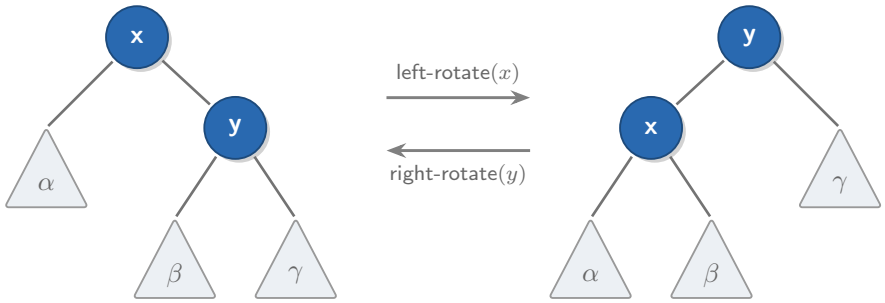


Figure 2: Rotation, the balancing primitive. Three pointers are rewritten in $O(1)$: y rises, x sinks, and subtree β changes parents. The in-order sequence $\alpha, x, \beta, y, \gamma$ is identical on both sides, so search order is preserved. Rotations move no colors; recoloring is a separate step in the fixup procedures.

That is the whole idea, and it is the part that does not move. What the definition hides, and what this assignment is built to expose, is that each of those circles is a heap allocation with an owner, a lifetime, and neighbors holding pointers to it. The remainder of this specification is about getting that part right when the allocator stops cooperating.

4 Assignment Structure

The assignment proceeds through three milestones plus a setup step and an optional, ungraded reach. Do them in order; each mutation builds on the last. The hour estimates for each element are included in the table below.

Milestone	Title	Est. hours
M0	Setup: carry the Assignment 1 tree forward, new frozen header, `rb_malloc`/`rb_free` plumbing, `CLAUDE.md`	1
M1	Mutation 1: allocation-failure injection and the fault sweep	4–5
M2	Mutation 2: slab pool, then the whole battery again on the pooled build	4–5
M3	Mutation 3 ($O(1)$-space teardown and traversal), hardening, process artifacts	3–4
The Reach	Mutation 4 (copy-on-write snapshots); ungraded, see Section 8.4	2–4
–	Process artifacts (`CLAUDE.md`, `PROMPTLOG.md`, `REFLECTION.md`), written throughout	1–2

Each mutation targets a property the tree already has:

Property of the tree	Mutation	OS analog
`rb_insert` performs several dependent allocations (node, key copy), any of which can fail	M1: fault-injection sweep	unwinding cleanly when `kmalloc` returns `NULL`
nodes are small, uniform, and allocated in bulk	M2: slab pool	the kernel slab allocator (`kmem_cache`)
height is $O(\log n)$, but teardown still visits all n nodes	M3: $O(1)$-space walks	8–16 KB kernel stacks
an update touches only the $O(\log n)$ path back to the root	M4: COW snapshots	copy-on-write pages after `fork()`

5 Why This Assignment Looks Different

You are *required* to use Claude Code for this assignment, and that requirement shaped its entire design. Modern AI coding agents can produce a working red-black tree in one shot. If that were the whole assignment, you would learn nothing. This assignment has two deliberately intertwined goals:

1. **Systems goal:** Take a working red-black tree and survive a series of *mutations* that stress memory management the way an OS kernel stresses it: custom allocators, tiny stacks, allocation failure at arbitrary points, and copy-on-write structural sharing. A correct tree that leaks, double-frees, or recurses unboundedly **fails**, even if every lookup returns the right answer.
2. **Process-Engineering-with-AI goal:** Learn to direct an AI agent the way a senior engineer directs a capable but overconfident junior: give it context, make it plan, keep changes small, verify everything with tools, and never merge code you can't explain.

The anti-goal: pasting this assignment into Claude and typing "solve this." That approach will actually *hurt* you here, for three reasons: (a) the fault-injection and grading harness punishes code nobody reviewed, (b) a large fraction of your grade comes from your process artifacts and a live walkthrough, and (c) you will be asked to explain arbitrary lines of your submission (*live*), and to extend it on parameters you have never seen.

6 Rules of Engagement for AI Use

- **Allowed and expected:** using Claude Code to explore, plan, implement, test, debug, and review; asking it to explain any concept; having it write test scaffolding and Makefiles.
- **Required:** the process deliverables in Section 13 (`CLAUDE.md`, `PROMPTLOG.md`, `REFLECTION.md`), the raw session transcripts, and a git history that shows incremental work for this assignment

($\geq$ 10 meaningful commits; a single "final submission" commit is an automatic process-grade of zero).

- **Forbidden:** submitting code you cannot explain. The live walkthrough (Section 12) exists to check exactly this. "Claude wrote it" is not an explanation; "this loop restructures the tree into a right-spine so teardown needs $O(1)$ space, and here's why that can't loop forever" is.
- **You own the bugs:** If the agent introduces a use-after-free and you commit it, it is *your* use-after-free.

7 Part-0: Setup (Milestone 0)

7.1 Carrying Assignment 1 forward

Start from the tree you already wrote. Continue in the same repository or clone it into a fresh one; either is fine, but the commit count in Section 13 counts commits made *for this assignment*, so a fresh repository that opens with one enormous import commit and then ten real ones is acceptable, while ten real ones and no import is better. Four things change on day one:

- **Swap in the new frozen header.** Appendix A is a strict superset of the one you have: two new declarations (`rb_create_pooled` and `rb_snapshot`) and two amended comments, on `rb_foreach` and `rb_destroy`. Everything you already implemented still compiles against it, unmodified. It is still frozen, and the agent must still never edit it.
- **Add the allocation plumbing.** All allocation in `src/` now goes through `rb_malloc`/`rb_free` (Section 8.1). If you took the tip at the end of Assignment 1, this is already done, and you may skip ahead feeling justified.
- **Keep your tests.** Your unit tests, your table-driven deletion tests, and your fuzzer come with you and are now a *regression suite*. Every milestone below must leave all of it green. If a mutation breaks a test you wrote three weeks ago, the mutation is wrong; the test is not negotiable.
- **Add the new files:** `tests/fault_alloc.c`, `tests/fault_alloc.h`, and `src/pool.c`.

7.2 Repository layout

```
rbtree-lab/
|-- CLAUDE.md              # you will write this in the workflow section
|-- Makefile               # Appendix B
|-- include/
|   `-- rbtree.h           # fixed public API (Appendix A). DO NOT MODIFY
|-- src/
|   |-- rbtree.c           # your implementation
|   `-- pool.c             # Mutation 2
|-- tests/
|   |-- test_rbtree.c      # your unit tests
|   |-- fault_alloc.c      # fault-injecting allocator (Mutation 1)
```

```
|   |-- fault_alloc.h
|   `-- fuzz.c              # randomized stress driver
`-- PROMPTLOG.md            # annotated AI-interaction log (see Deliverables)
```

7.3 Install Claude Code

If it is still installed from Assignment 1, skip this. Otherwise, macOS / Linux / WSL:

```
curl -fsSL https://claude.ai/install.sh | bash
```

Windows PowerShell:

```
irm https://claude.ai/install.ps1 | iex
```

(Homebrew users: `brew install --cask claude-code`.) Then, from inside `rbtree-lab/`, run `claude` and follow the login prompt. Full instructions: `https://code.claude.com/docs/en/quickstart`.

7.4 Toolchain rules

Everything must compile with `-std=c23 -Wall -Wextra -Werror`. Two verification builds are required and graded. You met both in Assignment 1 and you will not escape them here.

AddressSanitizer (ASan, switched on with `-fsanitize=address`) is instrumentation the compiler weaves directly into your binary. It watches every memory access as the program runs and halts the instant one goes wrong: a read past the end of a heap block, a use of memory after it was freed, a double free. It then prints the offending source line together with a backtrace of where that memory was originally allocated. Its companion *UBSan* (UndefinedBehaviorSanitizer, `-fsanitize=undefined`) catches the C standard's undefined behavior in the act: signed overflow, out-of-range shifts, misaligned or null pointer dereferences. Because both are compiled into the program, they slow it down. In exchange, they pin each bug to the exact instruction that commits it.

valgrind takes the opposite approach. It is a separate program that runs your *unmodified* binary on a synthetic CPU, interposing on every allocation and memory access from the outside, so it needs no special compile flags and no recompilation. Its `memcheck` tool finds the same families of fault (leaks, invalid reads and writes, and reads of uninitialized memory) at the cost of running many times slower than native. The two tools overlap but are not redundant: ASan is faster and sharper about the access that goes wrong, while valgrind needs no rebuild and catches some uninitialized-value reads that ASan lets through. Requiring both to come back clean is deliberate belt-and-suspenders.

- `make asan`: AddressSanitizer + UBSan build, runs the test suite.
- `make memcheck`: valgrind `--leak-check=full --error-exitcode=1` over the test suite.

A single leaked byte, invalid read/write, or UB report in either target caps the correctness portion of that milestone at 50%. This is the assignment's central discipline: memory bugs are not warnings. They are failures.

One practical note that matters more this time than last. Mutation 3 asks for a tree of a million nodes, and valgrind runs many times slower than native. Put the million-node case behind a flag or its own target so `make memcheck` still finishes inside a coffee break: run the large case under `asan`, which is fast enough, and a smaller case under valgrind. What you must not do is quietly shrink the large case because a timing problem was inconvenient. If your harness ends up running the "million-node" teardown at ten thousand nodes, say so in `REFLECTION.md`, and expect the walkthrough to ask about it.

8 Part-1: The Mutations (Milestones 1–3)

Up to this point the tree has lived in the most forgiving world imaginable: every node it asked for, it received, and every node it finished with, it was free to forget. That forgetting is the luxury this part takes away. Three mutations follow, and not one of them changes what the tree computes; each changes only the ground it stands on. To keep your footing you will need three ideas the baseline never forced you to meet.

A useful way to read the next three milestones is as a sequence of increasingly strict contracts. Mutation 1 says: memory allocation may fail. Mutation 2 says: nodes may no longer come from the general-purpose allocator at all. Mutation 3 says: the call stack is no longer available as scratch space for walking the tree. At each stage, the abstract data structure remains exactly the same. Search still searches. Insert still inserts. The red-black invariants still hold. What keeps changing is which implementation conveniences you are allowed to depend on.

That separation is an important systems idea in its own right. An abstraction tells callers *what* a component does. Its implementation decides *how* resources are acquired, managed, and eventually returned. Much of operating-systems work consists of changing the latter without allowing the former to flinch.

The first is the *memory leak*. When you ask the allocator for a block, it hands you the bytes and remembers nothing else about them; keeping track of that block is now entirely your problem. Free it and it returns to circulation. Lose the last pointer to it without freeing, and the block is stranded: still allocated, forever unreachable, dead weight until the process exits. In a program that runs for a second, such a leak is a tree falling in an empty forest. In one that runs for months, which is to say a kernel, a database, or a web server, leaks accumulate, and accumulation is fatal. The cruelty is that they gather on precisely the paths you are least likely to test: the error paths, where one allocation failed halfway through a chain of them, you returned early, and you quietly forgot to release what you had already grabbed. Mutation 1 (Section 8.1) exists to walk down every one of those paths and check. There is a second lesson hiding inside that one. Correctness under failure is not merely "remember to free things." Ideally, an operation should either complete or leave the structure as though it had never started. You will encounter this idea again under other names: failure atomicity, transactional behavior, commit points. Here you are learning it with two malloc calls and a tree, which is a pleasantly small place to meet a very large idea.

The second idea is that asking for memory one node at a time is not free. Every call to `malloc` must find a suitable block, record its size, align it, and stitch its own bookkeeping around it. Do that a million times for a million identical little nodes and the bookkeeping starts to rival the payload! While the nodes themselves scatter across the heap and your cache spends the day chasing pointers.

When every object is the same size, you can decline to play that game. Grab memory wholesale in big page-sized chunks, slice each chunk into node-sized slots, and hand them out yourself; when a node dies, thread its corpse onto a free list so the next request can reuse it without troubling `malloc` at all. That is a *pool allocator*, and it turns both allocation and freeing into an $O(1)$ pointer shuffle. An analogy might be useful here. General-purpose `malloc` is a hotel desk that handles guests needing rooms of every imaginable size and duration. Your pool has a much easier job: every guest wants exactly the same room. Once you know that, you can build a corridor of identical rooms and keep a list of which doors are available. There is much less deciding to do on every arrival.

A related term you need for Mutation 2 is a *slab*. A slab is simply one of those wholesale chunks from which many same-sized objects are carved. The allocator is the policy and machinery; the slab is the chunk of storage it manages. The pattern has a distinguished pedigree, because the Linux kernel allocates staggering numbers of identical small objects, one structure per open file, per running process, per network packet, and it manages them with exactly this machinery, the slab allocator that lives behind `kmem_cache`. Mutation 2 (Section 8.2) asks you to build a small, honest version of the same thing.

Historically, the slab-allocation idea came from Solaris and was later adopted and evolved in Linux. Modern Linux has changed quite a bit under the hood, but the object-cache abstraction remains useful for exactly the reason it is useful here: kernels repeatedly create and destroy enormous numbers of small objects with known sizes. Your allocator is tiny by comparison, but the design pressure is recognizably the same.

The third idea is different. A recursive algorithm appears not to allocate anything because there is no malloc in sight. But it is still consuming memory. Every recursive call allocates another stack frame implicitly. Mutation 3 asks you to notice that hidden resource and then stop using it. You will walk and destroy the same tree with only a fixed handful of pointers, regardless of how many nodes it contains.

Do all three in order; each builds on the last. The ordering matters more than it may first appear. Mutation 2 does not replace Mutation 1; your pool must still behave correctly when its slab allocation fails. Mutation 3 does not replace either one; its teardown has to work for both ordinary and pooled trees. Think of the milestones as constraints accumulating, not assignments being discarded.

And notice, at the end, that the tree never noticed: same keys, same invariants, same answers, and an entirely different story underneath about where its nodes come from and where they go to die. That story is, not coincidentally, the whole job description of a memory manager.

8.1 Mutation 1: "Every path leaks nothing" (allocation-failure injection)

Route **all** allocation in `src/rbtree.c` through `rb_malloc`/`rb_free` wrappers (declared in `tests/fault_alloc.h`, trivially forwarding to `malloc`/`free` in production). The test build interposes a fault-injecting version with this contract:

```
#include <stddef.h>

/* Allocation wrappers. In production these forward to malloc/free;
```

```
 * the test build swaps in the fault injector below. ALL allocation
 * in src/rbtree.c (and src/pool.c) must go through these. */
void *rb_malloc(size_t n);
void  rb_free(void *p);

/* the n-th allocation from now returns NULL */
void  fault_alloc_arm(long n);
void  fault_alloc_disarm(void);
/* allocations observed so far */
long  fault_alloc_total(void);
```

Your harness must run the **fault sweep**: for $n = 1, 2, 3, \ldots$ re-run a fixed scenario (e.g., insert 200 keys, delete 100, overwrite 50) with the n-th allocation failing, until a run completes without hitting the fault. After *every* run, whether it failed midway or not:

- `rb_validate` must still pass: a failed insert must leave the tree exactly as it was;
- `rb_destroy` must free everything; ASan/valgrind must report **zero** leaks across the entire sweep;
- every API call that hit the failure must have returned its documented error code.

Why this is hard: `rb_insert` allocates up to three times (node, key copy, and possibly pool metadata later). Failure *between* allocations is where real systems leak. This is the same discipline as kernel code paths that must unwind cleanly when `kmalloc` returns `NULL`.

The important phrase there is *between* allocations. Failure before you have acquired anything is easy. Failure after everything has succeeded is easy. The interesting state is the middle, when you own some resources but not yet enough resources to complete the operation. At that instant, every allocation you have successfully acquired has become an obligation. Either ownership eventually transfers into the tree, or you must give it back. There is no third destination.

This is also why fault injection is more than an artificial classroom trick. Real systems use it precisely because genuine allocation failures are too rare and too inconvenient to wait for. Linux has fault-injection facilities that can deliberately fail page and slab allocations so developers can exercise error paths on demand. A failure that occurs once every six months in production is a terrible test plan. Here, we make it occur at some allocation number.

A guarantee about the sweep. A fixed scenario performs a bounded number of allocations, so the fault sweep is guaranteed to terminate: some finite n is larger than every allocation the scenario makes, and the run at that n completes without ever hitting the injected fault. The injected failures are deterministic for a given scenario and n. If your sweep appears not to terminate, or if the same n gives different results on different runs, treat that as a bug in your own code, most often a hidden dependence on uninitialized memory or on allocation addresses, not as a sign that the harness is unfair.

"Exactly as it was" is a testable claim, so test it. That bullet is easy to read as a slogan and hard to read as an assertion, so pin it down. After a failed `rb_insert` of key k, at minimum:

`rb_validate` still passes, `rb_size` returns what it returned before the call, `rb_find(t, k)` returns what it returned before the call, every other key in your reference model is still findable with its original value, and the caller's `value` has not been freed, because ownership never moved. That last one is invisible to `rb_validate` and is exactly the sort of thing our scenario checks: allocate the value with a known destructor, fail the insert, and confirm the destructor never ran.

There are actually two promises bundled into "exactly as it was." One is *structural*: the tree still contains the same keys, values, size, and red-black relationships. The other is *ownership*: nothing that belonged to the caller before the failed operation has quietly become the tree's problem, and nothing already owned by the tree has been lost. Your tests need evidence for both.

Where the design work actually is. The cheapest way to survive the sweep is to make most of the failure paths impossible rather than correct. Figure 3 is the shape to aim for: perform every allocation the operation needs *before* you touch a single pointer in the tree, so that by the time you start modifying the structure, nothing that remains to be done can fail. Rebalancing allocates nothing. Rotations allocate nothing. Recoloring allocates nothing. If all of the allocation sits in front of the point of no return, then "leave the tree exactly as it was" costs you nothing at all, because you never changed it.

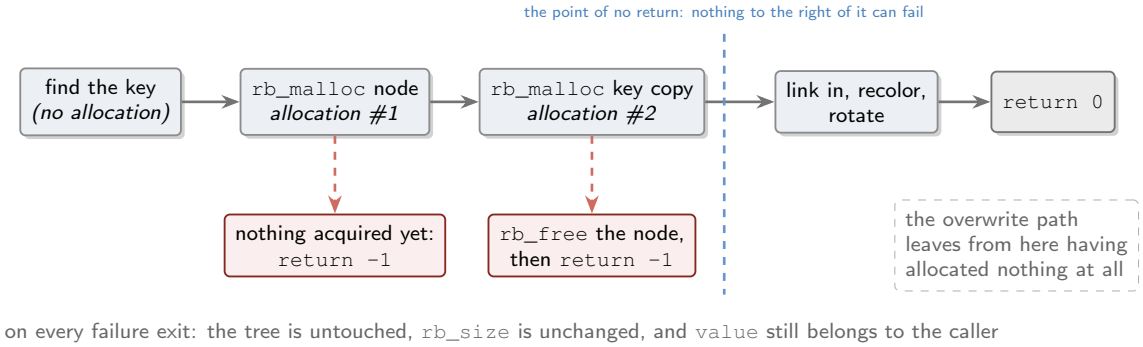


Figure 3: The shape of an insert that survives the sweep. Every allocation the operation will ever need is taken before the first pointer in the tree is rewritten, so each failure exit has only its own predecessors to undo and the structure is never observed half-built. The dashed line is the point of no return: to its right, fixup does pointer surgery and recoloring, neither of which can fail, which is why "the tree is left exactly as it was" is free rather than laborious. The mirror-image design, in which the node is linked in first and the key copied afterwards, is also implementable, and is where partial-unwind leaks come from. Note that the arrow into `return -1` is not an apology: returning failure *is* correct behavior here, and the harness checks that you return it rather than press on.

This is the miniature transaction hiding inside `rb_insert`. Before the dashed line in Figure 3, you are preparing. After it, you are committing. A good commit point has one particularly useful property: once you cross it, every remaining operation is guaranteed to complete. That is why moving allocations to the left of the line simplifies the program so dramatically.

Notice what happened here. We did not become better at cleaning up complicated failure states.

We redesigned the operation so that most of those states cannot exist. In systems code, eliminating a failure path is usually better than becoming heroic at recovering from it.

Three consequences worth thinking through before you write code, each of which is a plausible walkthrough question:

- **The overwrite path should not allocate.** Overwriting an existing key means freeing the old value and installing the new one. The tree already owns a perfectly good copy of that key. If your implementation allocates a fresh key copy on overwrite and then has to unwind when it fails, you have manufactured a failure path that did not need to exist. Deleting it is a better fix than handling it.
- **`rb_delete` should not allocate either.** The frozen header gives `rb_delete` exactly one error code, and it means "absent." There is no way to report an allocation failure through it. If your delete allocates a temporary, you have an unreportable failure mode, and the fix is to stop allocating rather than to invent a return value.
- **Partial success is not success.** If the node allocation succeeds and the key copy fails, you are holding one live allocation that nobody else will ever hear about. That is the classic two-step leak, and it is the single most common thing the sweep catches.

All three bullets are instances of the same design principle: do not create fallible work unless the operation actually requires it. Every additional allocation is another place where the operation can fail, another ownership transition you must reason about, another branch for your tests, and another opportunity for Claude to cheerfully get 95% of it right.

What the harness looks like. The design is yours, but the shape is not mysterious: a function that runs the fixed scenario once and reports what happened, wrapped in a loop over n that arms the injector, runs the scenario, tears the tree down, and asserts the three bullets above. Keep the scenario deterministic and keep it in one place, because you will be re-running it on the pooled build in Mutation 2, and a scenario that quietly depends on `rand()` without a fixed seed will cost you an evening. Print one line per n, not one line per operation, and see Section 11 on what to paste back to the agent.

Once Mutation 1 passes, you have established a useful contract for the rest of the assignment: no operation is allowed to depend on allocation succeeding merely because it usually does. Keep that contract. Mutation 2 is about to replace the allocator underneath you, but it does not repeal any of the rules you just learned.

8.2 Mutation 2: "Roll your own slab" (pool allocator)

Nodes may no longer come from per-node `malloc`. Implement a slab-style pool in `src/pool.c`:

```
typedef struct rb_pool rb_pool_t;
/* slabs are 4096 bytes */
rb_pool_t *pool_create(size_t obj_size);
/* O(1), from a free list */
void      *pool_alloc(rb_pool_t *p);
/* O(1), returns to the free list */
void       pool_free(rb_pool_t *p, void *obj);
void       pool_stats(const rb_pool_t *p,
                      size_t *slabs, size_t *live, size_t *free_objs);
/* releases all slabs */
void       pool_destroy(rb_pool_t *p);
```

Constraints: slabs are page-sized ($4096 = 2^{12}$ bytes) chunks obtained with `rb_malloc`; freed objects are chained into an intrusive free list *inside the dead objects themselves* (no side table); `pool_alloc`/`pool_free` are $O(1)$; `pool_destroy` frees every slab regardless of how many objects are live. The tree gains `rb_create_pooled(...)` and must pass the *entire* Assignment 1 + Mutation 1 test battery (including the fault sweep, since slab allocation can fail too) on the pooled build.

The conceptual change is worth making explicit. In Mutation 1, you changed the allocator's behavior: it could refuse a request. In Mutation 2, you change the allocator's granularity. Instead of obtaining one node from the system for every node the tree needs, you obtain a large block occasionally and satisfy many later requests yourself. That means one `rb_malloc(4096)` can support dozens of future `pool_alloc()` calls. Those calls no longer need to ask a general-purpose allocator to search its data structures, perform bookkeeping, or decide where another tiny allocation should live. You paid the general-purpose cost once and amortized it across many objects.

Figure 4 is worth reading in two layers. Horizontally, each slab is just an array of potential objects. Vertically, the free list ignores that physical layout completely and links together whichever objects happen currently to be unused. The slab answers "where is the storage?" The free list answers "which slot can I hand out next?" Keeping those questions separate makes the implementation much easier to reason about.

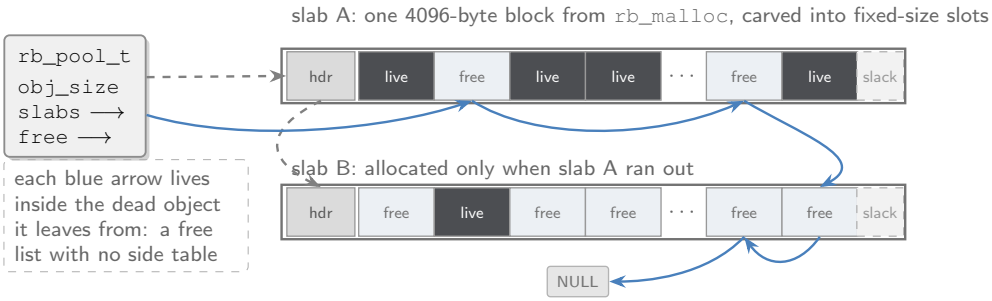


Figure 4: The pool, in one picture. Two page-sized slabs, obtained with `rb_malloc` and chained so that `pool_destroy` can release every one of them; within each slab, fixed-size slots handed out to the tree. The blue chain is the free list, and its whole trick is that the "next" pointer lives in the first bytes of an object no longer in use, which is why `pool_free` costs one store and `pool_alloc` costs one load. Note the two places where the arithmetic goes wrong for almost everybody: the slab header, which eats bytes the naive division forgets, and the `slack` at the tail, the remainder too small to hold another object, which must never be handed out. Note also that the free list wanders between slabs in whatever order objects happened to die. That is fine, and it is also why the objects a long-running tree gets back arrive in no pleasing spatial order.

Pinning down the parts the API leaves vague. Three decisions are yours, and all three are fair game at the walkthrough.

- **Alignment.** A slot must be aligned strictly enough for the object you are going to put in it, and also for the free-list pointer you are going to put in it when that object dies. Say out loud what alignment you are guaranteeing and how the slot stride enforces it. `-fsanitize=undefined` has opinions about misaligned pointers and will share them. Alignment is one of those requirements that feels ceremonial until it is wrong. A CPU does not see "an `rb_node_t`." It sees loads and stores at addresses. C types come with alignment requirements that constrain which addresses are legal for those operations. Your slot stride thus has to satisfy a mathematical condition before the first node is ever placed in it.
- **Where the slab header lives.** In-band, at the front of the slab, costs you slots and makes "which slab is this object in?" answerable by masking. Out-of-band, in a separate small allocation, keeps the arithmetic clean and adds a second allocation that can fail. Either is defensible. Pick one and be able to say why. This is a nice example of a recurring systems tradeoff: metadata can live *with* the thing it describes or *beside* it. In-band metadata improves locality and avoids a second allocation, but steals payload space. Out-of-band metadata keeps the slab geometrically clean but introduces another object whose allocation, lifetime, and failure path you now own. Neither choice is universally right. What matters here is that yours is deliberate.
- **What `pool_stats` counts.** Define `free_objs` as the number of objects `pool_alloc` could hand out *without* calling `rb_malloc` again: slots on the free list, plus any slots in an existing slab you have not carved yet. Your three numbers must then satisfy `live + free_objs == slabs * objs_per_slab` at all times, which is a one-line assertion your tests should

make after every scenario, and a quantity the walkthrough may ask you to tighten. Treat `live + free_objs == slabs * objs_per_slab` as an invariant, not merely as a convenient check at the end. If an invariant should always be true, assert it repeatedly while the allocator is under stress. The earliest assertion failure is usually much more informative than the corpse you discover ten thousand operations later.

What still comes from `rb_malloc`. The pool holds fixed-size objects, so it holds nodes. Key copies are variable-length and keep coming from `rb_malloc` exactly as before. This matters at teardown: destroying a pooled tree can return all the node storage wholesale, but the key copies and (when owned) the values still have to be released one at a time, which means teardown still has to visit every node. Mutation 3 is about how you visit them. That distinction is important because "pooled tree" does not mean "the tree performs no ordinary allocations." You have changed the storage policy for one class of object, the fixed-size nodes. Variable-sized keys remain a different allocation problem. Real allocators make exactly this kind of distinction because different object populations reward different strategies.

OS connection: this is a miniature of the kernel slab allocator (`kmem_cache_alloc`). Write one paragraph in `REFLECTION.md` comparing your design to it: what does the kernel's version add, and why? When you write that reflection paragraph, do not merely list features that Linux has and yours does not. Ask why those features become useful at kernel scale. Per-CPU caches, for example, are not decorative sophistication. They reduce contention when many cores are allocating the same kind of object simultaneously. Once the workload changes, the allocator design changes with it.

Watch out: reusing a freed node's memory hides use-after-free bugs from ASan (the memory is still "valid" to the sanitizer). Add **poisoning** (`memset` freed objects to `0xDD` in debug builds) and explain in your reflection what class of bug this catches that ASan now cannot. Think about the interaction with the free list while you are there: the poison and the "next" pointer want the same bytes, and only one of them can have them.

The reason is subtle and worth understanding. ASan knows that the 4096-byte slab is still a live allocation. From its point of view, a stale pointer into that slab still points into allocated memory. Your pool knows that a particular slot is dead, but ASan does not know about your sub-allocation boundaries. You have created a memory manager below the sanitizer's level of abstraction. Poisoning gives you a crude way to make that otherwise invisible state visible. In a debug build, fill the dead slot first, then write the free-list link into the bytes that must remain meaningful. If stale code later reads the rest of that object, it sees conspicuously impossible data rather than the old node contents. This does not make every use-after-free magically detectable, but it makes an important class of stale accesses much harder to pass unnoticed. ASan isn't asleep. You have simply started subletting rooms it believes you still occupy.

At this point you have changed both the failure model and the source of node storage. One resource remains suspiciously convenient: the call stack. Mutation 3 takes that one too.

8.3 Mutation 3: "No stack to spare" ($O(1)$-space teardown and traversal)

Before the requirement, a word about the thing you are about to run out of.

Every running function needs somewhere to keep its private business: its parameters, its local variables, the address it should jump back to when it's done, and whatever registers its caller is trusting it not to clobber. That bundle is a *stack frame*. It lives on the *stack*, a region of memory handed to your thread when the thread is created, which grows as calls nest and shrinks as they return.

The nesting is the whole point. Say `x()` calls `y()`, and `y()` calls `z()`. At the instant `z` begins executing, all three frames are on the stack at once, piled in call order: `x` at the bottom, `y` on top of it, `z` on top of that. Nothing of `x` has gone anywhere. Its local variables (or locals, for short) are just sitting there, buried but alive, waiting patiently for the frames above them to clear out. When `z` returns, its frame pops and `y` resumes with everything exactly where it left it; when `y` returns, same again for `x`. Frames come off in exactly the reverse of the order they went on, which is the entire reason this thing is called a stack and not something more imaginative.

There is one conceptual connection I want you to make here. Recursion is not free bookkeeping. A recursive tree walk works because the runtime is quietly remembering, on your behalf, which node you came from and where execution should resume. The algorithm looks as though it uses only a node pointer because the call stack is carrying the rest of the state out of sight. Big-O analysis counts that memory too.

Now here's the catch, and it's a big one. That region is finite. Its size gets fixed the moment the thread is born, and nobody grows it for you on demand, not even a little. Each nested call eats another few dozen bytes of it, and a function that calls *itself* is really just an especially enthusiastic version of `x` calling `y` calling `z`: recursion of depth d means d frames stacked up and alive at once, all of them leaning on the innermost one to finish first. A recursive tree teardown recurses once per level, so its peak stack usage tracks the height of the tree, a number you don't control and, mid-teardown, can't easily bound.

Do the arithmetic and it gets uncomfortable fast. Give a frame a modest 48 bytes, which is optimistic for anything doing real work, and a 64 KB stack buys you somewhere north of a thousand nested calls before you are out of room. A kernel thread, which is what you should be imagining throughout this course, gets roughly 8–16 KB in total. That is a couple of hundred frames for *everything*, your recursion included, and you are sharing it with whatever called you.

What happens when you run off the end is worth knowing, because the two answers are very different. In userspace on Linux, a guard page sits below the stack and the write into it kills your process with a **segfault**. That is the *good* outcome. You get a core dump, a backtrace, and an afternoon. Inside a kernel, and on older kernels without guard pages especially, you get no such courtesy: the stack simply keeps growing into whatever memory was unlucky enough to be adjacent, silently corrupting it, and the resulting bug surfaces somewhere else entirely, in a different subsystem, an hour later, wearing a disguise. It's a genuinely miserable class of bug to chase and a large part of why kernel code avoids unbounded recursion as a matter of policy rather than taste.

The obvious dodge: allocating your own explicit stack on the heap and pushing node pointers onto it, is *not* available to you here either. It relocates the problem without solving it. It makes teardown cost $O(n)$ extra memory at exactly the moment you are trying to give memory back, and (per Mutation 1) that allocation can fail, which leaves you needing to free a tree with no way to walk it. Destruction paths that can fail are their own special kind of unpleasant.

That last sentence is the deeper reason for the restriction. `rb_destroy` is the operation you

call because you are finished with memory. Requiring it to acquire more memory before it can release the memory it already owns creates an awkward dependency: cleanup itself can now fail because the system is short of exactly the resource you are trying to return.

So: no recursion, no heap, no borrowing. Reimplement:

- `rb_destroy`: no recursion, no heap-allocated stack/queue, $O(1)$ auxiliary space. (Classic technique: rotate the tree into a right spine as you free it, or use pointer reversal.)
- `rb_foreach`: same constraints. Morris traversal or an explicit-parent-pointer walk both work; if you add parent pointers, account for the memory cost in `pool_stats` and your reflection.

Both techniques lean on one idea worth appreciating before you write a line of code: the tree already has plenty of pointers lying around, and most of them are pointing at nodes you're finished with. Instead of recording where you've been in some separate structure, you borrow the tree's own pointer fields to remember it, then put them back (or free them) on the way out. Your bookkeeping lives inside the data itself, which is exactly why the auxiliary space is a couple of local variables and nothing more. It's a lovely trick. It's also precisely the sort of code whose loop invariant you'll be asked to justify at the walkthrough, out loud, without hedging, without reaching for your notes. Know what it maintains. Know why it can't spin forever.

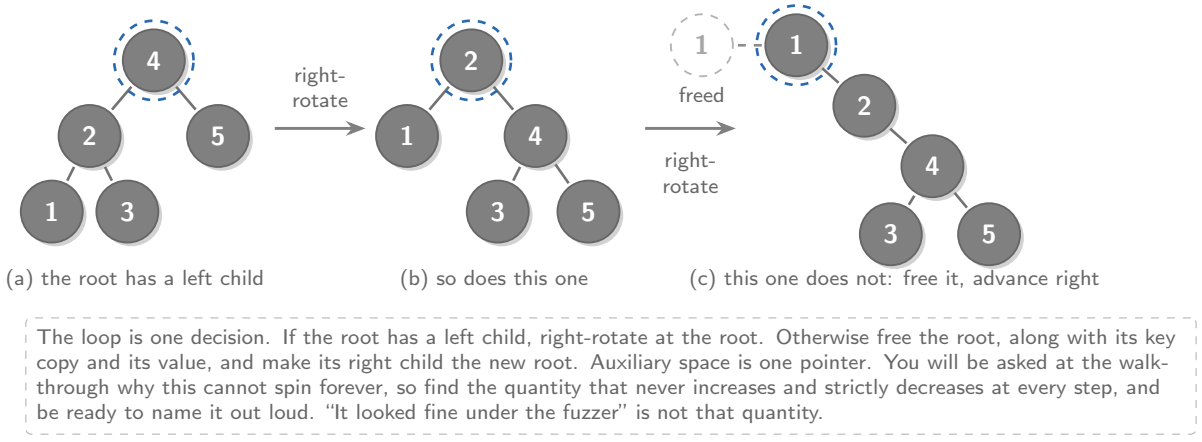


Figure 5: Teardown by rotating the tree flat, which is the technique the parenthesis in the requirement is pointing at. Nothing here is allocated and nothing is remembered: each iteration either lifts the root's left child over it, strictly shrinking what still hangs to the left of the root, or finds no left child at all and reclaims the root outright. Colors are drawn uniformly gray because teardown does not care about them; a tree being destroyed is allowed to stop being a legal red-black tree the instant you start, which is precisely why `rb_validate` is no help to you in the middle of this loop. Note the ordering obligation in panel (c): the right child's address has to be saved before the node holding it is freed. Panel (c) is also where a pooled build differs from a plain one, since the node goes back to the pool while the key copy goes back to `rb_free`.

Figure 5 also contains a useful algorithmic lesson: termination should have a *measure*. "The loop seems to make progress" is intuition. "This nonnegative quantity strictly decreases" is an argument.

When you can identify that quantity, you have converted a visual hunch into something much closer to a proof. That habit travels well beyond trees.

A wrinkle in `rb_foreach` worth noticing before ASan notices it for you. The frozen header declares the traversal as taking a `const rbtree_t *`. Morris traversal works by temporarily rewiring a node's right pointer so that its in-order predecessor points back at it, then rewiring it again on the way past. So for the duration of the walk, the structure is deformed and then restored. That is legal C here, because `const` qualifies the handle rather than the nodes reached through it, but it has consequences you should be able to state: during the callback, the tree is *not* in its resting shape, so a callback that calls `rb_find`, or worse `rb_insert`, is asking a question of a structure that is temporarily lying. Decide what your contract is, write it in a comment, and test whatever you decided. Also decide what happens if the callback never returns, though for this assignment "don't do that" is an acceptable answer as long as you know you chose it.

This is a nice place to separate logical constness from physical constness. To the caller, `rb_foreach` behaves like a read-only operation: the keys, values, ordering, and final tree are unchanged. Internally, Morris traversal may temporarily mutate pointers and then restore them exactly. The externally visible abstraction is const even though the machinery briefly is not. Systems code contains many variations on this idea, and it is one reason API contracts matter more than a casual glance at individual assignments. The restoration requirement is absolute. If the callback returns normally, the tree after `rb_foreach` must have precisely the same structure it had before. Temporary mutation is an implementation technique, not permission to leave fingerprints.

The test. Your test must build a tree of $\geq 10^6$ nodes (the pooled build makes this fast) and then destroy it under a deliberately small pthread stack (64 KB).

Run `rb_foreach` there as well. This remains a useful stress test and should finish with zero leaks and no sanitizer failures. But because red-black height is logarithmic, passing the small-stack test is not by itself proof that you eliminated recursion. We will also inspect the implementation and may ask you at the walkthrough to explain exactly where traversal state lives and why its storage remains $O(1)$. The million-node case is testing something broader as well: that your $O(1)$-space algorithm still behaves correctly when "works on my 20-node unit test" has stopped being a persuasive argument. 20 nodes are wonderfully cooperative. A million tends to ask whether you meant what you wrote.

8.4 Mutation 4: "`fork()` your tree" (The Reach: COW snapshots)

This mutation carries no points. Do it anyway. It is the part of the assignment that most resembles real kernel work, and the one you will be proudest to have gotten right. Nobody is checking whether you finished it, which is exactly what makes finishing it mean something. If the first three mutations taught you to keep a tree alive under pressure, this one asks whether you can make two trees share a life without either one corrupting the other.

Copying is expensive, and most copies are never even scribbled on. That single observation is the whole idea behind *copy-on-write*, one of the great lazy triumphs of systems design. When you

ask for a snapshot of the tree, the honest but foolish reply is to duplicate every node; the clever reply is to hand back a second doorway into the very same nodes and to promise the real copying for later, if and only if someone actually tries to change something. As long as the original and the snapshot only ever read, that promise never comes due, and one set of nodes quietly serves two masters.

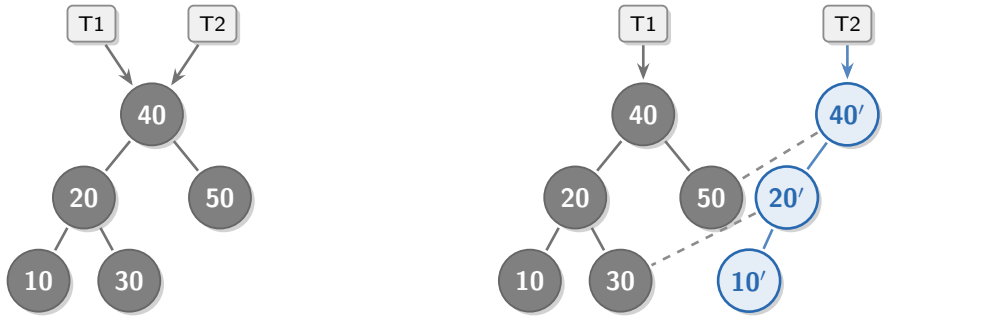
The bill arrives on the first write. You cannot scribble on a shared node without the other handle seeing it, so before changing a node you copy it, and because its parent pointed at the old version you must copy the parent too, and its parent, all the way up to the root. A write clones exactly the $O(\log n)$ nodes on the path back to the root, which is why the technique is called *path copying*; every subtree hanging off that path stays shared, now claimed by two parents at once (Figure 6). That shared claiming is what makes teardown interesting, because a node no longer belongs to one tree and cannot simply be freed when one handle goes away. Enter the per-node *reference count*: raise it when a new handle comes to share a node, lower it when a handle lets go, and free the node only at the instant its count reaches zero. Get the order of those decrements wrong and you have produced either a leak or a use-after-free, which is precisely the bug class that has kept kernel security researchers gainfully employed for decades.

If all of this is starting to sound like `fork()`, that is because it is `fork()`. When a process forks, the kernel does not copy its address space; it marks the pages read-only, shares them between parent and child, and waits. The first write to a shared page faults, the kernel copies that one page, and both processes carry on as though the copy had been there all along, with the reference count living in `struct page`. Mutation 4 asks you to rebuild that mechanism in miniature, on a tree instead of an address space, which is exactly why the reflection below asks you what plays the role of the page fault.

Add:

```
/* O(1): new handle sharing all nodes */
rbtree_t *rb_snapshot(rbtree_t *t);
```

After a snapshot, both handles are fully usable. Writes through either handle must not be visible through the other: copy the modified node's path to the root ("path copying", Figure 6), leaving shared subtrees intact. Manage node lifetime with per-node reference counts; `rb_destroy` on one handle must free exactly the nodes no longer reachable from any live handle. (If you do not attempt the Reach, simply leave `rb_snapshot` undefined; the declaration in the frozen header costs you nothing as long as nothing calls it.)



(a) after `T2 = rb_snapshot(T1)`: every node shared, refcounts bumped

(b) after a write to key 10 through T2: only the path is copied (blue); 50, 30 stay shared (dashed)

Figure 6: Copy-on-write snapshots by path copying (Mutation 4). The snapshot itself is $O(1)$: the new handle shares every node, and per-node reference counts record the sharing. A later write through T2 copies only the $O(\log n)$ path it touches ($40'$, $20'$, $10'$); subtrees rooted at 50 and 30 remain shared and are now reachable from two parents each. Destroying either handle must free exactly the nodes whose reference counts drop to zero, in any destruction order. Node colors are omitted here; the copied path carries its red-black colors with it.

Required tests: interleaved writes to 3+ snapshots with model checking; the fault sweep over a snapshot-heavy scenario; a leak-free teardown in *every* snapshot destruction order. In `REFLECTION.md`, connect this to copy-on-write pages after `fork()`: what plays the role of the page fault? Of the reference count in `struct page`?

This mutation is where refcount bugs (the classic kernel CVE generator) live. Expect Claude to get it *almost* right on the first try. "Almost" is the lesson. Notice too that it collides head-on with Mutation 3: a teardown that must decrement a refcount and stop descending wherever the count stays positive is a considerably more delicate loop than the one in Figure 5, because it can no longer assume the nodes it is walking are its own to rearrange.

9 Part-2: Required Use of Claude Code (the workflow)

You are required to use Claude Code for this assignment and to log that usage, but the goal is for you to leave the assignment understanding every line you submit, not just to produce working code by any means available. This section is graded via your transcripts and your `PROMPTLOG.md`. Work through it step by step; each step names the failure it prevents.

Step 1: Give the agent persistent context in `CLAUDE.md`

Claude Code reads a file named `CLAUDE.md` at the start of every session; it is the one place where your rules survive between conversations. You wrote one for Assignment 1: carry it forward and amend it rather than starting over, because the diff itself is evidence of how your constraints evolved. A working example is in Appendix C; yours must at minimum pin down:

- build/test commands (`make test`, `make asan`, `make memcheck`) and the rule that *nothing is "done" until all three pass*;
- `include/rbtree.h` is frozen and the agent must never edit it;
- all allocation goes through `rb_malloc`/`rb_free`;
- style rules (C23, no VLAs, no `goto` except cleanup-label unwinding, every allocation checked);
- "make the smallest change that passes; do not refactor unrelated code."

Failure this prevents: re-explaining your constraints every session, and the agent "helpfully" rewriting your header.

Step 2: Explore before anything is written

Start a fresh session and ask questions with no edits: "Walk me through the deletion fixup cases and why each rotation preserves black-height." "Where in this codebase would an allocation failure between the node alloc and the key copy leak memory?" You are calibrating two things: your own understanding, and whether the agent's picture of the problem matches yours.

Failure this prevents: discovering at the walkthrough that you never understood deletion fixup.

Step 3: Plan first, in plan mode

For every milestone, press **Shift+Tab** to cycle permission modes until the status line shows **plan mode** (or use `/plan`). In plan mode the agent can read and analyze but not modify anything. Prompt like this:

> *Plan Mutation 1. Read src/rbtree.c and tests/fault_alloc.h first. I want: (1) every allocation site in rb_insert identified, (2) for each, what must be unwound if it fails, (3) the exact error-path code structure you propose (I prefer goto-cleanup), (4) the fault-sweep harness design, (5) which tests you'd write first. List risks. Do not write code yet.*

Then **argue with the plan**. Ask "what breaks if the key allocation succeeds but the node allocation fails?" If the plan hand-waves, say so and make it revise. Only approve when *you* could defend every step. Record at least one plan revision in `PROMPTLOG.md`; a plan you accepted unmodified on the first try is a red flag we will look for.

Failure this prevents: 400 lines of confident code solving the wrong problem.

Step 4: Tests first, then the smallest slice

Out of plan mode, direct implementation in slices, tests leading:

> *Write the failing tests for the fault sweep first: arm fault_alloc at $n = 1 \ldots N$ over the standard scenario, asserting rb_validate passes and error codes are correct after each run. Don't touch rbtree.c yet. Run make test and show me the failures.*

Then: "Now make only the insert path survive the sweep. Smallest diff. Run `make asan` and show me the output." One slice per prompt. If a diff comes back touching five files, reject it: "Too big. Insert path only."

Failure this prevents: the un-reviewable mega-diff, and the agent quietly weakening tests to make them pass (read every test diff, because this is a real failure mode).

Step 5: Close the loop with tools, demand evidence

Never accept "the tests should pass now." Make the agent run the commands and show output. Feed tool output back verbatim; the agent reads a valgrind report better than a vague description of one:

> *make memcheck reports "40 bytes definitely lost, allocated at rbtree.c:212 in rb_insert". Trace the ownership of that allocation through every path out of rb_insert and fix the leak. Then re-run make memcheck and show me a clean report.*

Failure this prevents: the agent asserting success; you graduating without ever reading a valgrind report.

Step 6: Manage the context window

The context is the agent's working memory; a session full of dead ends makes it dumber. Use `/clear` between milestones and after finished sub-tasks; use `/compact` mid-task if a long debugging session gets noisy but you need the thread. Rule of thumb: new milestone, new session. `CLAUDE.md` carries the constraints forward, which is why Step 1 matters.

Failure this prevents: the "it keeps forgetting my instructions" spiral late in a long session.

Step 7: Adversarial review in a fresh context

Before each milestone commit, get a review from a context that did not write the code, because a reviewer that shares the author's assumptions inherits its blind spots. Either run the bundled `/code-review` skill, or `/clear` and prompt:

> *Review the diff for Mutation 2 as a hostile kernel reviewer. Hunt specifically for: use-after-free enabled by pool reuse, off-by-one in slab carving, alignment bugs for the intrusive free-list pointer, paths where pool_destroy is called with live objects. Report findings with file:line; do not fix anything.*

Triage the findings yourself and decide which are real. Log one real finding and one false positive in `PROMPTLOG.md` with your reasoning.

Failure this prevents: the author grading its own homework.

Step 8: Git discipline, conversationally

Commit at every green state: "run the full battery; if clean, commit as 'M2: pool allocator survives fault sweep'." Small commits are your undo button when a later "fix" regresses the sweep, and they are also your process grade.

Step 9 (optional): Automate your loop

Create a project slash command, a file at `.claude/commands/battery.md`, so the whole battery is one keystroke:

```
Run make test, make asan, and make memcheck in sequence.
If any fail, stop and diagnose the first failure: $ARGUMENTS
Show all output. Do not modify tests to make them pass.
```

Then `/battery` runs it in any session. Docs for going further (hooks that block edits to `rbtree.h`, subagents): `https://code.claude.com/docs/en/common-workflows`.

9.1 Prompt patterns: good vs. useless

Useless	Effective
"solve the assignment"	"Plan Mutation 1 per CLAUDE.md; identify every allocation site in `rb_insert` and what must unwind on failure. No code yet."
"fix the bug"	"The fault sweep fails at $n = 7$: `rb_validate` reports a black-height violation after a failed insert of key 'm'. Reproduce with the seed in `tests/fuzz.c` line 14, find the state the failed insert leaves behind, and propose a fix."
"make it not leak"	"Here is the full valgrind output: [paste]. Explain the ownership of the 40 bytes at `rbtree.c:212` on the early-return path at line 230, then fix only that path."
"write tests"	"Write table-driven tests for `rb_delete` covering: red leaf, black leaf with red sibling, root deletion, both children present. Each asserts `rb_validate` and `rb_size` afterward."
"is this right?"	"State the invariant this loop maintains at the top of each iteration, and show why termination follows. If you can't, the code is wrong; say so."

Transcript submission. You must submit the raw session transcript(s) Claude Code already saves for you, not a hand-written or `/export`'d summary. By default these live locally as JSONL files at `~/.claude/projects/<project> /<session-id>.jsonl`, one file per session, timestamped and recorded tool call by tool call. Copy the `.jsonl` file(s) for this assignment's project directory into your submission as-is; do not edit, reformat, retype, or hand-transcribe them. Work on this assignment in its own dedicated project directory, both so this copy step is a single `cp`, and so the file doesn't contain unrelated conversations from other work. Claude Code purges local session transcripts automatically after 30 days by default; if you start early, copy them out periodically or raise `cleanupPeriodDays` in your settings. Losing early transcripts to this default is not a valid excuse for an incomplete submission, so plan around it.

10 Helpful Claude Code Resources

You will not need most of Claude Code's documentation for this assignment, and chasing all of it is its own way of not writing the tree. What follows is the short path, ordered the way you will actually meet each idea. Read the first two before you write a line; keep the middle three open while you work; reach for the last two when the basics have gone automatic. Each maps to a step in the workflow of Section 9, so if a step there feels underspecified, the matching link is where it is spelled out in full.

- **Quickstart**: install Claude Code, log in, and run your first session. Start here, once.
  `https://code.claude.com/docs/en/quickstart`
- **How Claude remembers your project**: what `CLAUDE.md` is and how it carries your project's rules across sessions. This is Step 1, and it is the difference between correcting the agent once and correcting it every morning.
  `https://code.claude.com/docs/en/memory`
- **Choose a permission mode**: what **plan mode** is, and how **Shift+Tab** cycles between reading, planning, and editing. This is the single habit (Step 3) that separates directing the agent from being surprised by it.
  `https://code.claude.com/docs/en/permission-modes`
- **Best practices**: the house rules for getting good work out of the agent: specific prompts, small diffs, evidence over assertion. Read once early, skim again when a session starts going in circles.
  `https://code.claude.com/docs/en/best-practices`
- **Common workflows**: worked recipes for exploring a codebase, fixing a bug, and writing tests, which is most of what Steps 4 and 5 ask of you. Also your pointer to going further with hooks and subagents.
  `https://code.claude.com/docs/en/common-workflows`
- **Commands**: the reference for the session commands this spec uses: `/clear` and `/compact` for managing context (Step 6), `/plan`, and defining your own slash commands like the `/battery` loop in Step 9.
  `https://code.claude.com/docs/en/commands`
- **Code Review**: how the `/code-review` skill runs an adversarial pass over your diff from a fresh context, which is exactly the hostile-reviewer move in Step 7.
  `https://code.claude.com/docs/en/code-review`

Two standing tips that outlast any link: inside a session, `/help` lists everything available right now, and asking Claude Code "how do I . . . " in plain language is often faster than finding the page. The documentation moves quickly, so if a link has drifted, the index at `https://code.claude.com/docs` will have its new home.

11 Using Your Claude Code Allowance Well

This course requires a paid Claude subscription, currently $20/month for the Pro plan; the free tier will not carry you through this assignment or the ones that follow it this semester. A subscription is not bottomless, though. Your allowance is metered on two clocks at once: (1) a session window that resets 5 hours after your first message, and (2) a weekly cap that resets at a fixed day and time assigned to your account. Both are shared across everything you do with Claude, so the evening you spend having it adjudicate an argument about tabs versus spaces is drawn from the same purse as the fault sweep. Neither clock is generous enough to be ignored. Neither is tight enough to be a problem if you work deliberately. The difference between those two outcomes is almost entirely a matter of habit, which is the good news, because the habits that conserve your allowance are the same ones this assignment already demands of you for entirely unrelated reasons.

Here is the mechanism behind that happy coincidence. Every message you send carries your whole conversation with it, because the agent has no memory between turns and must be handed the *context* afresh each time. So cost scales with context, not with the number of questions you ask, and a long session grows more expensive per message the longer it runs, whether or not the accumulated history is still relevant. A conversation that has wandered through three dead ends is paying rent on all three ... indefinitely, like a storage unit full of furniture you no longer own.

Consider the anti-goal named earlier in this assignment: pasting the whole specification into a session and typing "solve this." This document is not short. You pay for it once when you paste it, and then again on every subsequent message for the remainder of that session, which is a remarkable sum to spend on an answer that Sections 12 and 15 exist specifically to catch. It is the only prompt in this course that manages to be expensive twice.

That is also why Section 9 tells you to `/clear` between milestones and `/compact` when a debugging thread gets noisy: those instructions were written to keep the agent sharp, and they happen to be the single most effective thing you can do to keep your allowance intact. Nearly every other rule there pays the same double dividend. Plan mode is cheap precisely because it prevents the expensive outcome: four hundred lines of confident, beautifully formatted code implementing a pool allocator for a node struct the agent guessed at rather than the one in your source. You then pay a second time to have it undone. Specific prompts are cheap because "add the goto-cleanup unwind to `rb_insert`" reads one function, whereas "improve this code" reads `src/`, then `tests/`, then the Makefile, and eventually `PROMPTLOG.md`, where it will encounter your earlier remarks about it. Small diffs are cheap because you review them once instead of three times. You were going to do all of this anyway; you may as well know that it pays twice.

Habits that matter most here

- **One milestone, one session:** Start fresh with `/clear` when you move from the fault sweep to the pool, or from the pool to the teardown. Stale context is charged on every subsequent message, and the pool allocator gains nothing from remembering the afternoon you spent certain that `rb_validate` was miscounting black-height when in fact you were counting the NIL sentinel twice. Use `/rename` before clearing and `/resume` to return to a session you may want again.

- **Compact rather than continue:** When a long debugging thread is finished but the task is not, `/compact` summarizes what came before. You can steer what survives: `/compact keep the failing test output and the current diff`, not `/compact keep everything`, which is the command for people who have made peace with their weekly cap.
- **Match the model to the job:** Sonnet handles ordinary implementation work well and costs less. Save the heavier model for the genuinely hard reasoning in this assignment, which is roughly two things: the unwind analysis in Mutation 1, and the refcount decrement order in Mutation 4. It is not required for a `pool_stats` function that returns three numbers through three pointers. Switch with `/model`.
- **Filter tool output before the agent sees it:** This assignment generates spectacular quantities of log. The fault sweep prints a result per run. A 10^6-node teardown will print one million cheerful lines if you let it, and it will let you. A valgrind report can run to hundreds of lines of which four matter. Step 5 asks you to paste tool output verbatim, and you should, but paste the twenty lines containing the error, not the two thousand containing the word `PASS`. `grep`, `head`, and `tail` are free; context is not. `make memcheck 2>&1 | grep -A6 'definitely lost'` costs you nothing and tells the agent everything.
- **Stop early when it goes wrong:** Press Escape the moment the agent heads somewhere you did not intend, rather than letting a bad edit finish and paying a second time to unwind it. Escape is the cheapest key on your keyboard, and it gets cheaper the sooner you press it.
- **Keep `CLAUDE.md` lean:** It loads at the start of every session, so every line is a standing charge. The example in Appendix C is about the right size. If yours has grown a subsection on your preferred brace style, you are paying a small fee to relitigate brace style every single morning for three weeks.

Watch the meter

You can't manage what you can't see, and both of the commands that show you are free.

Command	What it tells you
`/usage`	How much of your plan's session and weekly allowance you have spent, with a breakdown of what consumed it. Press `d` or `w` to switch between the last day and the last week.
`/context`	What is occupying your context window right now, which is the thing you are paying for on every message.

Check `/usage` once at the start of a work session and once when something feels slow. If you would rather not think about it at all, configure the status line to display context usage continuously; the goal is to notice a ballooning context before it forces a compaction, not to perform an autopsy afterwards.

Three ways students will burn a week's allowance in an afternoon

1. **The session that never closes:** Opened Friday for Milestone 1, still open Sunday for Milestone 2. Every question you ask on Sunday arrives escorted by Friday's confident and entirely wrong theory that the leak was inside `strcmp`, and you are billed for the escort.
2. **Pointing the agent at everything:** The off-by-one is in your slab-carving loop, 40 lines of `pool.c`. There is no reason for the agent to read the fuzzer, the Makefile, and eleven hundred lines of tests on its way there. Name the file. Name the function.
3. **Convening a committee:** Agent teams spawn multiple instances, each with a context window of its own, and can consume several times the tokens of an ordinary session. Four agents hunting one missing `rb_free` is not a productivity strategy; it is a way of buying the same wrong answer four times, concurrently!! They are off by default. For this assignment, leave them off.

A last word, and it is the least technical of all: start early. The 5-hour window is a coffee break, but the weekly cap is a week, and it does not care that your deadline is Thursday. A student who works three unhurried evenings will finish comfortably inside the allowance. A student who begins the night before will meet both clocks at once, and only one of them can be waited out. The weekly cap is also the only entity associated with this course that has no discretion, no sympathy, and no email address.

Figures and mechanics here were current when this assignment was written and do change; `/usage` in your own session is always the final authority. For details, see `https://code.claude.com/docs/en/costs` *and* `https://support.claude.com/en/articles/8325606-what-is-the-pro-plan`.

12 Personalized Verification

Passing every test battery does not end your obligation on this assignment. Every student, not a flagged subset, sits for a short mandatory live walkthrough after the deadline. It is not a bonus check triggered by a mismatch between your log and your code; it happens for everyone.

At the walkthrough you will be given a small set of parameters specific to you and one small, previously unseen modification to make to your own already-submitted code, for example adding a new command-line flag to the test driver, handling a key-length range you did not originally test, or tightening a `pool_stats` figure. You will make this change live, starting from your own submitted source and nothing else. We will also pick lines of *your* code ("why is this node recolored?", "what frees this on the error path?", "why can't this teardown loop forever?") and one episode from your `PROMPTLOG.md` and ask you to defend your judgment call. This is the concrete form of the 1-point quiz described in the grading breakdown.

Why this exists. Every algorithm in here is specified precisely enough to autograde by test battery, sanitizer, and fault sweep, which is exactly what makes rigorous grading possible. It is also, unavoidably, what makes this document alone reproducible from a single request to an AI coding tool. Personalized Verification exists because a static submission, however complete, can only ever get you to a plausible-looking result. Only your ability to navigate, defend, and extend

your own code after the fact, on parameters and modifications neither you nor any AI assistant could have seen in advance, can confirm the work behind it was actually yours.

13 Deliverables

Submit, via Canvas, a tar file that contains the following.

- **Source code:** a `Makefile` (Appendix B) that builds the test and fuzz binaries with `gcc -Wall -Wextra -Werror -std=c23` and no warnings; `src/rbtree.c`, `src/pool.c`, and your tests; the *unmodified* frozen `include/rbtree.h` (Appendix A).
- `CLAUDE.md`, checked in and evolving across the milestones.
- `PROMPTLOG.md`: 6–10 annotated episodes, each with the prompt, what came back, and *your* judgment (accepted, rejected, or pushed back on, and why). Must include one plan you revised (Step 3), one rejected/oversized diff (Step 4), one tool-output debugging loop (Step 5), and one review finding you triaged (Step 7). Raw transcript dumps here score nothing; annotation is the deliverable.
- `REFLECTION.md` (1–2 pages): the slab and (if attempted) COW comparisons from Section 8; your `pool_stats` definitions and your slab-header and alignment decisions; where the agent was most and least reliable; one bug it introduced that you caught, and how; your poisoning rationale.
- The raw Claude Code session transcript(s) (`.jsonl` files) for this project's directory, copied in as-is. Not a summary, not a `/export`, not retyped.
- A git history with ≥ 10 meaningful commits tracking the milestones. A single "final submission" commit is an automatic process-grade of zero.

`PROMPTLOG.md` and `REFLECTION.md` carry no separate point weight, but they are reviewed by the TA alongside your transcripts and are used to seed quiz questions; a missing or perfunctory one will cost you where it hurts, on the quiz.

14 Grading

This assignment is worth **15 points**: 12 points for program correctness, 2 points for an analysis of your Claude Code transcripts, and 1 point for the quiz on your own code described in Sections 9 and 12.

Within the 12 correctness points, the governing discipline is not a grading technicality; it reflects what this course is actually about. A red-black tree that returns the right answer but leaks memory, double-frees, reads freed memory, or recurses without bound has not solved the problem this assignment poses; it has solved an easier one. So correctness is graded in two layers, both settled by tools rather than by opinion.

- **Layer 1: functional correctness.** Your Assignment 1 battery is still green; failed allocations unwind to the prior tree state with the documented error code; the pooled build passes that entire battery including the fault sweep; a 10^6-node tree tears down under a 64 KB stack. This is checked by running your suite, and by running our scenarios against your tree.
- **Layer 2: the memory-bug cap.** A single leaked byte, invalid read/write, or UB report under `make asan` or `make memcheck` caps the correctness portion of that milestone at 50%. The measurement is mechanical and reproducible; there is no room for disagreement, which is exactly why we grade the hardest part of the assignment this way.

Component	Method	Points
Mutation 1: fault sweep	All allocation in `src/` routed through the wrappers; full sweep leak-free; every failed operation unwinds to the prior state, with the correct error code and without consuming the caller's value	4
Mutation 2: pool allocator	$O(1)$ slab pool with intrusive free list passes the entire Assignment 1 + Mutation-1 battery on the pooled build; `pool_stats` internally consistent; `pool_destroy` safe with live objects; debug poisoning present	4
Mutation 3: $O(1)$-space teardown	$\geq 10^6$ nodes destroyed under a 64 KB stack, zero leaks, no recursion and no heap auxiliary space; `rb_foreach` under the same constraints	2.5
Code quality & memory cleanliness	Compiles clean under `-Wall -Wextra -Werror`; zero ASan/UBSan/valgrind findings; frozen header unmodified; sane structure	1.5
Program correctness subtotal		*12*
Prompt analysis	Manual TA review of your raw Claude Code transcripts (Section 15) for genuine, iterative use rather than wholesale generation, corroborated against your repository	2
Quiz	Short quiz generated from your own submitted code, administered at the live walkthrough (Sections 12 and 9)	1
Total		**15**
The Reach	Mutation 4 (COW snapshots) complete with tests and reflection: attempted for the craft, not for points	–

Autograder preview. Before you submit, conformance looks like: your Assignment 1 suite and fuzzer run green against the reference model under both sanitizer targets; the fault sweep completes leak-free with correct error codes after every run; the pooled build passes the same sweep; a 10^6-node tree tears down under a 64 KB `pthread` stack with no crash and no leaks; and, if attempted, interleaved writes across three or more snapshots match the model with a leak-free teardown in every destruction order. Full public test cases will be posted and made available later.

15 How We Grade Your Claude Code Transcripts

Two of the 15 points come from your Claude Code transcripts. It is worth being precise about what those points measure, because it is easy to misread them as a participation check that any submitted log passes. They are not. They measure whether your transcripts show the fingerprints of someone building and debugging their own program, and that is a claim we can check against your repository rather than take on faith.

The governing principle is simple, and everything below follows from it: **the grade is based on what can be corroborated against the repository, never on the log's own narrative**. A transcript can say anything. It can claim you caught a subtle refcount bug, demanded a test, or rejected a bad design. What earns credit is not the claim but the evidence for it sitting in your git history, your source, and your tool output. A confident story with nothing behind it scores below a modest one that checks out.

15.1 Mechanism

The grader runs headless (`claude -p`) or as a slash command, with your repository checked out alongside the log, not the log alone. It reads your transcripts as a set of claims and then verifies those claims by inspection against the repo. It asks questions of this shape: does the commit a prompt refers to actually exist; does the fix for a diff you rejected in conversation show up later in history; does the table-driven `rb_delete` test you told Claude you wanted actually appear in your test files; does the `valgrind` leak you debugged at, say, `rbtree.c:212` correspond to code that in fact changed around that line? Episodes that cross-check against a commit, a test, or a diff count. Episodes that float free of any repository artifact do not, no matter how well written they are.

This is the same philosophy as the memory-bug cap in Section 7.4 and the live walkthrough in Section 12, applied to your process instead of your output. A plausible-looking artifact that does not actually connect to real work is easy to produce and, deliberately, easy to catch.

15.2 The scale

Your transcripts are scored on the five-level scale below for technical substantiveness. The level determines the point value directly, in even steps of 0.4: level 5 earns the full two points, and each level down subtracts four tenths.

Level	Name	What it looks like	Points
5	**Verified engineering dialogue**	Prompts reference concrete artifacts (file and line, invariants, tool output); your pushback demonstrably changed outcomes; every required episode cross-checks against a commit, test, or diff.	2.0
4	**Substantive, partially corroborated**	The same quality of reasoning, but one or two episodes cannot be tied back to repository evidence.	1.6
3	**Concrete but shallow**	Real artifacts are cited, but the judgments are acceptance without technical reasoning, of the "looked good, kept it" variety.	1.2
2	**Generic**	Plausible transcripts with no verifiable anchors; the log could describe any student's project as easily as yours.	0.8
1	**Performative or contradicted**	The narrative conflicts with git history, or episodes appear fabricated.	0.4

15.3 What this means for how you work

You do not earn a high score by narrating impressively. You earn it by working in a way that leaves a verifiable trail: small commits, tests you actually asked for and actually wrote, real error messages pasted from real runs, and design decisions you can point to in the code. If you use Claude Code the way Section 9 describes, as a collaborator you argue with one function and one bug at a time, the corroboration falls out of your normal workflow and you will land at level 4 or 5 without effort. The only way to score low here is to submit a log that has little to do with the code beside it, and that is precisely the case these points exist to catch.

16 Suggested Schedule

- **Week 1:** Milestone 0 + 1: carry the tree forward, swap in the frozen header, allocation wrappers, `CLAUDE.md` amended, fault sweep green under asan/memcheck.
- **Week 2:** Milestone 2: the pool allocator, then the entire battery again on the pooled build, sweep included.
- **Week 3:** Milestone 3: the $O(1)$-space teardown and traversal, then hardening. Finish the write-ups, `PROMPTLOG.md` and `REFLECTION.md`, and take the Reach (Mutation 4) if you have time. Walkthroughs in section.

17 Late Submission

The late submission policy for CS370 is available from the course website at `https://www.cs.colostate.edu/~cs370`.

18 The Spirit of This Assignment

You are required to submit your own work, and you are required to use Claude Code to help you produce it. Those two requirements are not in conflict. Your own work in the context of this assignment has never meant work built with no help; it means work that you understand well enough to defend and extend, regardless of how much help you had along the way.

There is no reason to submit someone else's work on this assignment. Not as a shortcut. Not under deadline pressure. Whatever a classmate's pool allocator might save you tonight, it costs you at the walkthrough, where a quiz built from code you did not write, and a live modification to code you do not understand, is not a shortcut but a trap you built for yourself. If the agent introduces a use-after-free and you commit it without understanding it, it is still your use-after-free to explain.

Do the assignment the way it is designed to be done. Use Claude Code as a real collaborator, not a copy-paste machine: plan first, keep the diffs small, verify everything with tools, and merge nothing you cannot explain. There is nothing left to gain by doing it any other way.

19 How AI Was Used in Developing This Assignment

This assignment has been developed with assistance from AI tools. The assignment itself, including its structure, requirements, learning objectives, and overall design, is my work. I used AI as a second set of eyes. A *tireless* second set of eyes. It helped me interrogate the specification and verify technical details. Notably, it helped me identify places where what I intended and what I had actually written were not quite the same thing.

In particular, I iterated with Claude and Claude Code to check the correctness of the specification and prompts, identify inconsistencies, explore corner cases that were missing from earlier versions, and test whether the instructions would lead to the behavior that I actually intended. That last part mattered more than you might think. This assignment has several moving parts: a small ambiguity in one place can have large, downstream consequences several steps later. Claude Code was useful for finding those places before you did.

These tools also identified places where my exposition could be clearer. I used those observations selectively in revising the text. The examples, explanations, constraints, and progression of the assignment went through substantial human iteration as well. Quite a lot of it. Intermediate drafts of this assignment were vetted by the GTAs, who would also point places where things were underspecified.

Another thing that I tested was what happens when a long assignment specification gets summarized by an LLM. I deliberately tested what happens when a long, detailed specification is handed to a model with a prompt along the lines of, "Give me a bulleted list of what he's going on about." This turned out to be useful. It exposed places where an important constraint, qualification, or

dependency could disappear in the summarization process, or where a concise summary could be technically correct but still lead you down the wrong path. I revised the specification to reduce those unintended consequences without trying to make the full document reducible to a small set of convenient bullets. Please read the entire assignment.

There is a certain appropriateness to all of this. I am asking you to use AI in this assignment as an engineering tool, not as a substitute for understanding. That is largely how I used it in creating the assignment too: to question, check, probe, and refine work that I was already doing. The goal was to make the assignment better before asking you to wrestle with it.

20 Version History

This section will reflect the change history for the assignment. It will list the version number, the date it was released, and the changes that were made to the preceding version. Changes to the first public release are made to clarify the assignment; the spirit or the crux of the assignment will not change.

Version	Date	Comments
1.0	9/16/2026	First public release of the assignment.

A `include/rbtree.h` (frozen)

This header is a strict superset of the one you implemented against in Assignment 1: two new declarations and two amended comments. Replace your copy with this one and do not touch it again.

```
#ifndef RBTREE_H
#define RBTREE_H
#include <stddef.h>

typedef struct rbtree rbtree_t;
typedef void (*rb_value_free_fn)(void *value);

/* Returns NULL on allocation failure.
 * value_free may be NULL (values not owned). */
rbtree_t *rb_create(rb_value_free_fn value_free);

/* Mutation 2: node storage drawn from an internal slab pool. */
rbtree_t *rb_create_pooled(rb_value_free_fn value_free);

/* Copies key (tree owns the copy).
 * Takes ownership of value ONLY on success.
 * Overwriting an existing key frees the old value via value_free.
 * Returns 0 on success, -1 on allocation failure (tree unchanged, value NOT
 * consumed; caller still owns it). */
int rb_insert(rbtree_t *t, const char *key, void *value);

/* Returns the value, or NULL if absent. Tree retains ownership. */
void *rb_find(const rbtree_t *t, const char *key);

/* Removes key; frees the key copy and the value.
 * Returns 0, or -1 if absent. */
int rb_delete(rbtree_t *t, const char *key);

size_t rb_size(const rbtree_t *t);

/* In-order. Mutation 3: O(1) auxiliary space, no recursion. */
void rb_foreach(const rbtree_t *t,
                void (*fn)(const char *key, void *value, void *ctx),
                void *ctx);

/* Returns 0 iff all red-black invariants hold
 * (see the baseline section). */
int rb_validate(const rbtree_t *t);

/* Frees all nodes, key copies, and (if owned) values.
 * Mutation 3: O(1) auxiliary space, no recursion. NULL-safe. */
void rb_destroy(rbtree_t *t);
```

```
/* Mutation 4 (The Reach): O(1) copy-on-write snapshot.
 * NULL on alloc failure. */
rbtree_t *rb_snapshot(rbtree_t *t);

#endif
```

B Starter `Makefile`

```
CC      := gcc
CFLAGS  := -std=c23 -Wall -Wextra -Werror -g -O1 -Iinclude
SRC     := src/rbtree.c src/pool.c
TSRC    := tests/test_rbtree.c tests/fault_alloc.c
BIN     := build/test_rbtree
FUZZBIN := build/fuzz

all: $(BIN) $(FUZZBIN)

$(BIN): $(SRC) $(TSRC) include/rbtree.h
    @mkdir -p build
    $(CC) $(CFLAGS) $(SRC) $(TSRC) -o $@ -lpthread

$(FUZZBIN): $(SRC) tests/fuzz.c tests/fault_alloc.c include/rbtree.h
    @mkdir -p build
    $(CC) $(CFLAGS) $(SRC) tests/fuzz.c tests/fault_alloc.c -o $@ -lpthread

test: $(BIN) $(FUZZBIN)
    ./$(BIN) && ./$(FUZZBIN) 100000

asan: CFLAGS += -fsanitize=address,undefined -fno-omit-frame-pointer
asan: clean test

memcheck: all
    valgrind --leak-check=full --show-leak-kinds=all \
        --error-exitcode=1 ./$(BIN)
    valgrind --leak-check=full --show-leak-kinds=all \
        --error-exitcode=1 ./$(FUZZBIN) 20000

clean:
    rm -rf build

.PHONY: all test asan memcheck clean
```

(Note: `asan` and `memcheck` are separate targets on purpose, because valgrind and ASan do not mix in one binary.)

C Sample **CLAUDE.md**

```
# rbtree-lab: project rules

## Commands
- Build & unit tests: `make test`
- Sanitizers: `make asan`   Valgrind: `make memcheck`
- A change is DONE only when all three pass. Always run them; show output.

## Hard constraints
- NEVER modify include/rbtree.h. It is the graded contract.
- All heap allocation in src/ goes through rb_malloc/rb_free
  (tests/fault_alloc.h). Direct malloc/free in src/ is a defect.
- Any allocation may fail. Every failure path must unwind completely:
  no leaks, tree left exactly as before the call, documented error code.
- NEVER weaken, skip, or delete a test to make the suite pass. If a test
  looks wrong, stop and explain why instead.

## Style
- C23. -Wall -Wextra -Werror must stay clean. No VLAs.
- Error handling: goto-cleanup pattern for multi-allocation functions.
- Prefer the smallest diff that passes. Do not refactor unrelated code.
- Every non-obvious loop gets a one-line invariant comment.

## Workflow
- For any multi-file or algorithmic change: propose a plan and wait for
  approval before editing.
- Commit only from a green state; message format "M<n>: <what>".
```