Assignment 1: The Companion

A Walkthrough for Building Your First Red-Black Tree with Claude Code

How to read, plan, prompt, test, commit, and finish with tokens to spare

Version 1.0 August 25, 2026

Version history appears in Section 17.

This document sits alongside Assignment 1. It replaces nothing. The assignment specification tells you *what* to build and *what* will be graded; this companion tells you *how* to spend your hours and your tokens while building it. If you'd rather read only one of the two? Read the assignment spec. If you read both? Read the spec first and then this document. Everything here assumes you have. Where the two seem to disagree, the spec wins. Two more things, since this assignment hands most of you two new tools at once: nothing here requires prior experience with Claude Code, and nothing here requires prior experience with C. The walkthrough starts from an empty folder and a fresh install, and it moves at the pace of someone meeting both for the first time. That is the whole point of a first assignment with a small, friendly number attached to it. Meet the tools now, while mistakes are cheap.

Contents

1 What This Document Is and Is Not . . . 2
2 Glossary: The Words I Keep Throwing Around . . . 2
3 Please Read the Assignment . . . 7
4 A Word for the Formerly Garbage-Collected . . . 9
5 Insertion, Deletion, and the Invariants Watching You . . . 10
6 Plan the Modules Before the Code . . . 12
7 The Two Weeks: No Software Is Developed in a Day . . . 15
8 Commit Wisely. Commit Often . . . 17
9 Test in Increments, Not in Ceremonies . . . 18
10 Fixing Bugs Without Burning Down the House . . . 19
11 Tokens, Limits, and the Meter You Cannot See . . . 20
12 Tokens: Spending Like It Is Your Money, Because It Is . . . 26
13 Write It Down While It Is Warm: The Devlog . . . 27
14 Your Prompt Log Is a Professional Artifact . . . 29
15 How AI was Used in Developing This Companion . . . 29
16 A Last Word . . . 30
17 Version History . . . 30

1 What This Document Is and Is Not

The spec defines the destination. The description here is one way of walking there without blisters. You could ignore this companion entirely and still earn a full score on the assignment; the tools that grade you don't care how you got to a clean valgrind report. But the students who land at the top of the transcript rubric mostly work the way this document describes, whether or not anyone told them to.

So treat what follows as a strong recommendation, not a decree. It is organized the way the work for the assignment is: you read, then you plan, then you build across two weeks, and while you build you commit, you test, you debug, and you write things down. Each section leans on the ones before it. The token discipline in Section 12 is mostly a summary of habits you will have already picked up by then for other reasons. That is not an accident. Almost everything that makes you a good engineer here also makes you a frugal one.

2 Glossary: The Words I Keep Throwing Around

Some of what follows is *jargon* you have met before. Some of it is jargon this document invented for its own convenience. And some of it is the kind of shorthand that everybody in the computer science field uses but nobody ever defines out loud, which is a genuinely annoying way to learn a field.

So here is the dictionary (I have put them in gray boxes). Skim it now. Come back when a phrase in a later section makes you squint.

Words about tests and tools

green: Everything passed. No failures, no leaks, no sanitizer complaints. The word comes from test runners that print failures in red and successes in green. So "commit at green" means: commit when nothing is broken. It is a state you can check, not a feeling you can have at 1 am.

red: The other one. Something failed. Stop and read what it said before you touch anything.

the battery: The full set of checks, run back to back: `make test`, then `make asan`, then `make memcheck`. Borrowed from "a battery of tests," which is old testing jargon rather than electricity. Nothing is charging. "Battery green" means all of it came back clean.

table-driven test: One array of cases and one loop that runs them. Each row is an input plus what should happen. Adding a case means adding a row, not writing a whole new function. "The table is green" means every row passed, including the mean ones you wrote on purpose.

repro: Short for reproduction. The exact recipe that makes the bug appear on demand: same seed, same operations, same failure, every single run. Without one, you do not have a bug report. You have a rumor.

shrink (or minimize): Cutting a repro down until there is almost nothing left of it. Ten thousand operations become twelve. The bug survives, the noise does not. This is probably the highest-value four minutes in all of debugging.

seed: The number you hand the random generator so it produces the same "random" sequence twice. Fixed seeds are what turn a fuzzer from a slot machine into a test.

fuzzer: A driver that hammers your tree with piles of random operations and checks that the answers are still right.

reference model: A second, dumber implementation that you trust completely. A sorted array is perfect for this. You perform the same operations on both and compare. Sometimes called an *oracle*, because it knows the truth and your tree is the one on trial.

harness: The scaffolding around the thing being tested. It sets up the scenario, runs it, checks the outcome, cleans up after itself. Your fuzzer runs inside a harness you write.

Words about the code

slice: One small piece of work you can finish, test, and commit in a single sitting. Not a feature. Not a milestone. "`rb_insert` without the fixup" is a slice.

diff: The lines that changed. VS Code's Source Control panel shows it to you, colored and staged, before you commit. Read it. Every time. Test files especially.

seam: A boundary you can swap something in behind without the code above ever noticing. `rb_malloc` is the seam the spec tips you to install in Part-1. The tree calls it and asks no questions. Plain `malloc` sits underneath today; next month, a fault injector takes its turn.

stub: A function that exists and compiles and does nothing useful yet. An honest placeholder, not a lie.

invariant: Something that is true every time the loop reaches the top. Not usually true. Always. State it and you can usually show the loop must end. Fail to state it and you have learned something uncomfortable about your own code.

fixup: The repair walk after an insert or a delete: recolor where you can, rotate where you must, climb until the damage is gone. Insert fixup never needs more than two rotations; delete fixup never needs more than three. Your 2am count of eleven cases is an illusion.

doubly black: The spec's accounting fiction for a deleted black node: the empty slot owes one black, and delete fixup is the business of settling that debt. It lives in your reasoning, not in your color enum. Do not add a third color. Please.

mirror case: The same fixup case with `left` and `right` traded. Same logic, fields swapped, and the half people forget to write. Or test.

ownership: Who is responsible for eventually freeing an allocation. Several pointers can reach an object; exactly one owner frees it. Confuse reachability with ownership and you get a double free, which is the allocator's way of suggesting you reread the spec's field guide.

unwind: Undo the partial work when something fails halfway through. The second allocation failed, so the first one has to go back where it came from. Get this wrong and you leak on exactly the path nobody tests.

teardown: Destroying the whole structure and handing every byte back. Different from deleting one key, and much easier to get subtly wrong.

spelunking: What the agent does when you do not tell it where to look. It wanders off into your repo with a headlamp, opening files, reading tests, billing you for the tour.

Words about git

commit: A saved snapshot of your repo with a message attached. Cheap, fast, and boring. Do it constantly.

commit at green: Only save snapshots that actually work. Your history then becomes a list of states you can safely return to, which is the entire reason for having a history in the first place.

`git restore`: Throw away your uncommitted changes and snap back to the last commit. Instant and free. Vastly better than paying the agent to un-write code it wrote forty minutes ago.

`git bisect`: Binary search through your history for the commit that broke things. Magnificent when your commits are small. Completely useless when your history is one commit called "final."

clean clone: Cloning your own repo into an empty directory and building from that. It answers one question, and it is a question worth asking before you submit: is the project I am handing in the same project I have been building, or does it only compile because of some untracked file sitting in my working directory?

Words about the agent and the meter

token: The chunk of text the model actually reads. Somewhere between a syllable and a short word. Call it 4 characters of English prose, and closer to 3 for C source, because your punctuation counts.

context: Everything the model can see on this turn: your prompt, your `CLAUDE.md`, every file it opened, the output of every command it ran, and all of its own earlier replies. It gets re-sent and re-read on every single message. So context is not a purchase. It is a subscription.

context window: The size of the container. How much context fits at once.

usage limit: Your budget, which is a completely different thing. You can be nowhere near filling the window and still be out of allowance for the week.

session: One continuous conversation, from your first message until you `/clear` it.

`/clear`: Wipes the conversation and starts clean. Your `CLAUDE.md` survives, so your project rules survive with it.

`/compact`: Summarizes what came before and carries on. You can steer what lives through it: `/compact keep the failing test output and the current diff`.

plan mode: Shift+Tab until the status line says so. The agent can read and analyze but cannot touch a file. This is where arguments are cheap and mistakes cost nothing.

`CLAUDE.md`: Your project's standing law, loaded at the start of every session. Which makes every line in it a recurring charge, so keep it lean and keep it about the project.

slash command: A saved prompt you fire with one keystroke, like the `/battery` loop. It lives as a file in `.claude/commands/`.

adversarial review: Asking a fresh session to attack code it did not write. A reviewer who shares the author's assumptions also inherits the author's blind spots, and the author here is sometimes you.

devlog: `DEVLOG.md`. Five minutes at the end of each session. It is memory that never once sends you a bill.

3 Please Read the Assignment

The spec is long. Read it the way you would read your housing lease. You are about to live in the spec for two weeks.

3.1 Pass one: read for shape, hands off the keyboard

Sit somewhere without a terminal and read the whole thing. Start to finish in one sitting. Budget 45 minutes. Do not open VS Code. Do not open Claude. The urge to start typing somewhere around the workflow section will be strong, and resisting it is the first test of the semester.

You are reading for shape on this pass: what the milestones are, what order they come in, what gets graded by machine and what gets graded by a human looking you in the eye. You are also reading for tone, because the spec telegraphs where the danger is. When a document spends half a dozen figures walking single inserts and deletes through every recolor, panel by panel, and then detours for an entire section to un-teach your Java reflexes, that is not padding. Those are flares over a minefield.

3.2 Pass two: read with a pen, extract into notes

Now read it again (slower) with `NOTES.md` open in VS Code (create it in your repo; it is yours, not a deliverable). This pass is extraction. You are pulling four kinds of things out of the prose, and every one of them gets the spec's section name written next to it so you can jump back later:

Extract	What it looks like	Where it ends up
Contracts	The header is frozen. Keys are copied and owned. `rb_insert` returns -1 and the caller still owns the value. Overwriting frees the old value through `value_free`, and nobody else does.	Lines in `CLAUDE.md`; the rules the agent must never break.
Thresholds	$\geq 10^5$ fuzz ops. `rb_validate` every 100 ops. The four required deletion cases, plus the fifth the spec tells you to add. Zero leaks, not few.	Assertions in your tests. Every threshold is a test waiting to be written.
Choices left to you	`NULL`-as-black or a shared sentinel? Parent pointers or not? Sorted array or linked list for the model? Recursive `rb_destroy` now, or the Reach's spine walk behind a second function?	Plan-mode discussions with Claude, and one-line decision records in your devlog (Section 13).
Confusions	"Why is a black uncle always NIL on the first fixup pass?" "Who frees the old value on an overwrite, and exactly when?"	Your first Claude session: exploration questions, before any code exists.

Notice what just happened? Your notes *are* the plan. The contracts become your `CLAUDE.md`. The thresholds become your test list. The choices become the agenda for plan mode. The confusions become your opening conversation. When Section 7 tells you to "plan Milestone 1," you will not be staring at a blank prompt box wondering what to say; you will be reading from a page you already wrote.

3.3 Revisit; don't rely on memory

Nobody holds a spec this size in their head for two weeks, and nobody should try. The spec is a reference manual, and reference manuals are for referencing. Build the habit now: before each milestone, reread the matching part of the spec even if you are sure you remember it. Especially if you are sure.

Before you start...	...reread, in the spec
M0 (setup)	Part-0 Setup, and both appendices (the Makefile and the sample `CLAUDE.md`).
M1 (insert/find/validate)	"What is a Red-Black Tree" through the harder-insertion figures, the "From Java to C" field guide, Part-1, and the frozen header in Appendix A. Read the header comments twice; the ownership rules in them are exam material in every sense.
M2 (delete + fuzzer)	The deletion walkthrough, twice: the recolor-only pair, the one-child splice, and the targeted-deletion-tests paragraph in Part-1, which names the exact cases your table must spring.
M3 (hardening)	The memory rules in Part-1 and the autograder preview. Milestone 3 is short precisely because everything it checks was supposed to be true already.
Every session	Skim your own notes from pass two. Thirty seconds. Cheaper than rediscovery.

And when something surprises you mid-implementation, the reflex is: spec first, notes second, Claude third, and a test to settle whatever survives all three. The order matters. Claude will cheerfully answer a question the spec already settled and you would have paid tokens for an answer that you owned in print.

4 A Word for the Formerly Garbage-Collected

Most of you are meeting C for the first time this month, and the spec knows it: the section called "From Java to C: A Field Guide for the Formerly Garbage-Collected" exists because your Java reflexes are about to write bugs on your behalf. Read it before you write a line of C. Then, as the spec itself suggests, reread it the first time valgrind says something you do not believe. That moment is coming. It is practically on the syllabus.

Two questions from that field guide deserve a permanent spot in your head, because every pointer in this assignment answers to both: how long does the thing being pointed at stay alive, and who frees it. Java answered these for you, silently, for years. Nobody answers them now except you. So when you finish any function that touches memory, run the primer's little audit: what was allocated, who owns each allocation now, what code will eventually free it. If any answer is "I think," keep looking. That sentence is stolen straight from the spec because it is the best sentence in it.

And here is where the agent fits, because Claude Code is a genuinely good C tutor if you make it teach rather than type. Ask it why `malloc(sizeof *n)` is the idiom. Ask it to quiz you on the Java-to-C table, one row at a time, with you answering first. Ask it what `n->left` expands to and why the arrow exists at all. What you should not do is let it quietly install habits you cannot defend, because "the agent knows C" and "I know C" are different claims, and the walkthrough only

interviews one of you.

5 Insertion, Deletion, and the Invariants Watching You

Here is the unusual thing the spec does, and it deserves to be used rather than admired. Most textbooks hand you the fixup case table and wish you luck. Your assignment instead *works the cases in front of you*, panel by panel: the red-uncle recolor that fixes everything, the harder insertion where the same recolor fixes nothing and two rotations have to finish the job, the deletion that repairs itself with two paint strokes, and the one-child deletion that is almost insulting once you see why the child had no choice about being red. All of it lives in the spec's "What is a Red-Black Tree?" section. This companion section is about extracting the value from those figures, because a figure read passively teaches nothing. It just looks like studying.

5.1 Predict, then scroll

The drill is simple and slightly embarrassing, which is how you know it works. Put your hand over the next panel. Say out loud what happens: which nodes change color, which pointers move, where the violation goes. Then look. When you are wrong, and you will be wrong, write one line about why in your notes.

Do it alone first. Then do it again with Claude Code in plan mode, which is the spec's own Step 2 wearing a lab coat: describe the starting configuration, have the agent predict each move while you grade it, then swap roles. The spec's advice on this is blunt and correct: get it wrong out loud, because it is a great deal cheaper here than at the walkthrough. Run the drill on the four worked sequences, in order. The red-uncle recolor. The two-rotation zig-zag insert. The recolor-only delete. The one-child splice. An hour, total, and it may be the best hour you spend on this assignment.

5.2 Insertion breaks a rule you can see

Every new key enters the tree as a red node, and the spec explains the strategy: red leaves every path's black count exactly where it was, at the price of a possible red-red edge. So insertion's damage is polite. One violation, one edge, an address you can point at.

But do not mistake the red-uncle recolor for a cure. It is a loop iteration. The spec says it tidies your desk by putting the mess on the desk upstairs, and the harder-insertion figures show that happening in real time: the violation you fixed at the bottom reappears two levels up, now wearing an interior node and a genuinely black uncle, and recoloring has nothing left to offer it.

While you are there, own the counting argument the spec makes about first-pass uncles: on the first iteration, a black uncle is *always* NIL. No exceptions, no clever counterexample waiting for you at 2am. Be able to reproduce that argument cold, because it is exactly the flavor of question the walkthrough loves, and it also explains something practical: hand-testing insertion with five keys and a hopeful expression will never manufacture the rotation cases. The loop has to climb first. Which is an argument for the fuzzer, and for trusting targeted tests over vibes.

And then the stale root. Panel (d) of the harder insertion is a bug with a biography: rotate at the top, forget to tell `t->root`, and `rb_find` starts searching a subtree while `rb_validate` cheerfully certifies the wreckage. The spec calls it a great bug to have once, briefly, and never again. Arrange the "once" on your own terms: write the stale-root regression test the same evening you write the rotations. A few lines. Done.

5.3 Deletion breaks a rule you can only count

Deletion is not so courteous, and the spec says so in exactly those words. Remove a black node and no red sits on a red anywhere; every node, examined by itself, is perfectly legal. The tree is just quietly one black short down a single line, and you cannot point at the problem because the problem is a count, not a place. The spec's accounting fiction, the *doubly black* slot that owes one black, is how you reason about it. Mind the red warning beside that paragraph: doubly black lives in your head, not in your enum. Your code needs to know where the missing black is and who its parent and sibling are. It does not need a third color.

The mercy is the funnel. Every deletion, however it starts, reduces: a two-child node hoists its successor's payload and deletes the successor instead, and the successor has at most one child by construction. A red leaf comes out for free. A black node with one child has a *red* child, and not by luck; the spec walks the arithmetic, and you should be able to walk it too, because "the child had no say in the matter" is a proof, not a shrug. Splice, paint it black, go home. The only genuinely hard case is the black leaf, where the debt is real, and even then the spec's worked example settles it with two recolors and zero pointer motion. When the parent is black instead, the debt climbs, the analysis reruns a level up, and if it ever reaches the root it simply dissolves there, because a black subtracted from every path is no violation at all.

One tripwire the spec flags in red, so I will too: if your deletion lands in the one-red-child shape and your code launches into the sibling case analysis anyway, something has gone sideways. Painting the child black settles everything. Test for that on purpose.

And the word "hoist" is doing real work. You are transferring a payload between nodes, which means transferring *ownership* of a key allocation. End up with two nodes that both believe they own the same key and you will discover, during `rb_destroy`, how strongly both of them hold that belief. That is the spec's own phrasing. It stays funny right up until valgrind agrees with it.

5.4 Keep the two repairs straight

	Insertion	Deletion
What breaks	Rule 4: a red-red edge. Visible, local, has an address.	Rule 5: one path's black count. Invisible, global, has to be added up.
First tool	Recolor while the uncle is red; each pass moves the problem two levels up.	Recolor while the sibling is black with no red child; each pass moves the debt one level up.
When rotations enter	The uncle is black: straighten the zig-zag, then rotate at the grandparent.	The sibling is red, or owns a red child: rotate that red across to the starving side.
Rotation ceiling	Two.	Three.
How it ends	A black parent stops the climb, or the root gets painted black.	A red node absorbs the debt, or the root dissolves it.

If you can reproduce that table from memory at the walkthrough, with the reasons attached, the fixup questions will feel like a formality. If you cannot, you have just found your study list.

5.5 The invariants are your tripwires

`rb_validate` is the five rules compiled into executable form, and the spec's testing bar makes it your constant companion: at least every 100 operations in the fuzzer, and after every mutation in your unit tests. Write it so a failure names the rule it broke. "Black-height mismatch under node 12" is an answer. A bare nonzero return is a mood.

The memory subplot rides along with every one of these operations, and the deletion figures keep pointing at the same trap: fixup must never dereference memory that has already been freed. Save the child pointer *before* you free the parent. A pointer kept "just for a second" does not receive a grace period from C, and ASan, as the spec promises, will explain this to you with characteristic warmth.

Last, the mirrors. Every case in both fixups has a twin with `left` and `right` traded, and the twin is the half people forget. The spec even tells you which extra deletion case to add to the required four. Tables make mirrors nearly free: one more row, not one more function. Decline the rite of passage.

6 Plan the Modules Before the Code

Here is a truth that will follow you into every job you ever hold: humans maintain what they can hold in their head, and now, agents bill you for what *they* have to read. Those are the same problem

at two different scales. Modularity is the answer to both. A 1,200-line `rbtree.c` is expensive for you to reason about and expensive for Claude to ingest, and you will be doing both, repeatedly, for two weeks.

6.1 The file-level map is given; the function-level map is yours

The spec fixes the file layout, so the coarse modularity is already decided for you. What is yours to design is the decomposition *inside* those files, and it is worth twenty minutes of thought before any code exists.

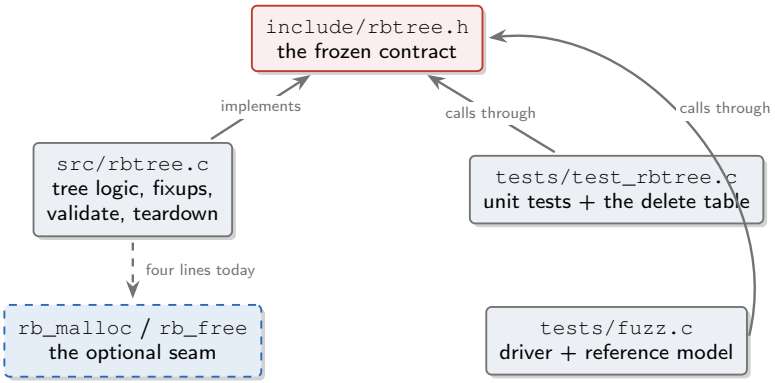

Figure 1: The module map. Every box is something you can point Claude at *by name*, and every arrow is a boundary that keeps a prompt small. The header is red because it is frozen. The seam is blue and dashed because the spec only tips it, in Part-1: today it forwards to `malloc` and `free` and asks no questions, and next month a fault injector sits down underneath without the tree ever hearing about it. Install the pipes while the house is small.

Inside `rbtree.c`, aim for static helpers with one duty each. Something like this:

Internal module	Its one job
`rotate_left`, `rotate_right`	The pointer surgery from the rotation figure. Nothing else. No colors. And they own updating `t->root` when the rotation happens at the top; the spec's harder-insertion figure shows, in loving detail, what forgetting that costs.
`node_alloc`, `node_release`	The *entire* allocation dance (node, then key copy) and its cleanup, in one place. If ownership is ever confusing, it is confusing in exactly one function.
`insert_fixup`	The recolor-and-rotate climb after insertion: the red-uncle loop, then the rotation cases with their mirrors.
`transplant`, `tree_minimum`, `delete_fixup`	The deletion machinery, kept apart from insertion's, because you will be pointing Claude at these and only these when deletion misbehaves. And it will.
`rb_validate` helpers	Black-height counting and order checking, written so a failure names the invariant it broke.
`rb_destroy`, `rb_foreach`	Recursion is legal this time, and the spec tells you to savor it. If you attempt the Reach, the spine-walk teardown lives behind a second function, checked against the same fuzzer, touched by nothing else.

Why does this granularity matter so much here? Because a prompt like "`delete_fixup` in `src/rbtree.c` mishandles the mirrored case where the sibling is red; here is the failing table entry" costs Claude one function's worth of reading. "Something is wrong with delete" costs it the file, and "fix my tree" costs it your repository, your dignity, your Tuesday, and a measurable slice of the weekly cap.

Put the map in writing. A short "Code map" block in your `CLAUDE.md` (five or six lines: what lives where) means the agent stops spelunking through your tests to find your rotation code. You pay for that block once per session. You save it back the first time the agent *doesn't* read `fuzz.c` on its way to a rotation bug.

6.2 You don't need all of it on day one

Look again at Figure 1. `delete_fixup` does not exist in week 1, and this is not a gap to feel guilty about; it is a schedule working as intended. `rb_delete` can spend Milestone 1 as an honest stub that returns -1, because the milestone boundaries exist precisely so you are never debugging deletion on top of an insertion you don't yet trust. The spec puts it memorably: that is less a milestone than an archaeological dig.

The seam is the other day-one kindness. Four lines of `rb_malloc` and `rb_free` now, forwarding to `malloc` and `free` and asking no questions, and next month's fault injector slots in underneath without the tree ever noticing. And the Reach? A stub, or nothing at all. It is ungraded, which is naturally the strongest argument the spec makes for it.

Build the smallest thing that lets the next test pass. Then the next. Remember the module map is a destination, not a birth certificate!

6.3 The design will wiggle. That is fine.

Your node struct will probably not survive the assignment unchanged. The textbook fixup algorithms may talk you into parent pointers halfway through Milestone 1; the spec calls that your first real design decision, and it is allowed to go either way. The Reach would swap your comfortable recursive teardown for a spine walk. So hold your design decisions firmly but not permanently, and notice that small modules are exactly what make revision cheap: when the struct grows a field, `node_alloc` changes, the rotations change, and almost nothing else does. A design that wiggles inside clean boundaries is called iteration. A design that wiggles without them is called a rewrite, and rewrites are billed at full price: in tokens and in evenings.

7 The Two Weeks: No Software Is Developed in a Day

The spec estimates 8–12 hours. That number is honest. But it hides the part that matters: those hours behave completely differently depending on *how* they are distributed. Twelve hours across six unhurried evenings is a comfortable project. Twelve hours starting the night before is a disaster with a countdown timer, and the weekly token cap makes it arithmetic rather than opinion: two calendar weeks means two weekly allowances. The night before the deadline means one, shared with your panic.

So here is what two weeks actually looks like, evening by evening. Each evening is 90 minutes to two hours. Each one ends with a green state, a commit, five minutes of devlog (Section 13), and a closed laptop. None of them ends with "I'll just get this one thing working," because that sentence is how 11 pm becomes 2 am.

7.1 Week 1: Setup, the C on-ramp, and insertion (M0 + M1)

Evening 1: the empty folder, the first session. Make the project directory and make it dedicated (the spec's transcript-submission rules assume this, and so does your sanity). In VS Code: open the folder, `git init`, add a `.gitignore` with `build/` in it, drop in the frozen header and the starter Makefile from the appendices, commit. That is commit #1 and you have written no code. Good.

Now install Claude Code per the spec's setup section, open the VS Code integrated terminal, and run `claude`. This is your first session, so let it be a gentle one: type `/init` to have it generate a starter `CLAUDE.md`, then *edit that file yourself* until it contains your project law, using the sample in the spec's appendix and the contracts column from your reading notes. Commit it. Then spend the rest of the evening on the spec's Step 2: exploration, zero edits. Ask the questions from your confusions list.

> *Read include/rbtree.h and CLAUDE.md. Walk me through the ownership rules in the header comments: on a failed rb_insert, who owns the value? Who owns the key copy at each point inside a successful insert? What happens to the old value when an existing key is overwritten?*

> *Then quiz me on the Java-to-C table from the spec, one row at a time: ask, wait for my answer, then correct me. Don't write or edit anything.*

Take notes on the answers. Argue with anything that smells off. End the session with `/clear`. You have now used plan-free exploration, `CLAUDE.md`, and context hygiene, on day one, at nearly zero cost.

Evening 2: insert and find, tests leading. Reread Part-1 of the spec first (thirty seconds of your notes, then the real thing). New session. **Shift+Tab** into plan mode and plan only the first slice: the node struct, `rb_create`, `rb_insert` without fixup, `rb_find`, `rb_validate`'s ordering check, and the comfortable recursive `rb_destroy` the spec permits this time. The parent-pointer question gets settled here, out loud, with the reason written into your devlog, because the spec calls it your first real design decision and "why did you choose this" is a walkthrough question. Argue with the plan until you could defend it. Approve. Then out of plan mode: tests first, smallest slice, run `make test` and `make asan`, show output, commit at green. 2–3 commits tonight, each one boring. Boring is the goal.

Evening 3: the rotations, the fixup, and the finish line for M1. Run the predict-then-scroll drill from Section 5 on both insertion sequences before you prompt anything. Then plan mode, then slices: rotations first, with the `t->root` update living *inside* them, then the fixup loop with its mirrors. Write the stale-root regression test the same evening you write the rotations; it is a few lines, and it springs the exact trap the spec spends a full panel warning you about. Finish with `rb_foreach`, an early fuzzer running inserts and finds against your reference model, and the full battery: `make test`, `make asan`, `make memcheck`. When all three pass, commit "M1: insertion complete, battery green." And before you close the laptop, make Claude do the thing you will later do live:

> *State the invariant the insert fixup loop maintains at the top of each iteration, and show why the climb toward the root must terminate. If you can't, the code is wrong; say so.*

If the answer does not convince you, the walkthrough version of you, weeks from now, will not convince anyone either. Last chore: copy this week's `.jsonl` transcripts out of `~/.claude/projects/` into a safe place. The 30-day purge is real (the spec warned you) and "the tool ate my homework" has never once worked on a grader.

7.2 Week 2: Deletion, hardening, and the paperwork (M2 + M3)

Evening 4: deletion, tests leading, and your first real argument. Reread the deletion walkthrough in the spec. Twice. Deletion is where both students and agents go subtly wrong; the spec says so more than once and then draws you pictures. So don't let Claude write `rb_delete` until the table-driven tests exist. Steal the spec's own Step-4 prompt for them, nearly verbatim:

> *Write the failing table-driven tests for rb_delete first: red leaf, black leaf with red sibling, node with two children, root deletion, a black node with exactly one (red) child, and the mirrored variants. Each case asserts rb_validate and rb_size afterward. Don't touch rbtree.c yet. Run make test and show me the failures.*

Then plan mode, and **argue with the plan**. Ask what happens when the node has two children and its successor is its own right child. Ask exactly where the key copy and the value get freed on every path, overwrite included. If the plan hand-waves, say so and make it revise. A first plan accepted unmodified is a red flag the graders look for, but more to the point, first plans are usually wrong somewhere, and finding where is the entire skill. Record the revision in `PROMPTLOG.md` tonight, while you still remember why you pushed back. Then implement in slices: the two-children reduction, then the one-child splice, committing at each green rung.

Evening 5: the fixup loop and the hostile reviewer. The doubly-black case analysis, one case per slice, mirrors included, until the table is green. Then turn the fuzzer loose with deletes enabled: 10^5 operations, fixed seed, `rb_validate` every 100 ops, under both `asan` and `memcheck`. When it fails at operation 6,410, and something like that will happen, shrink before you prompt (Section 10). End the evening with the spec's Step 7: `/clear` into a fresh context and run the adversarial review, hunting the exact bug families the spec names: a use-after-free in the successor splice, a key copy leaked on the overwrite path, an allocation whose `NULL` return nobody checks. Triage the findings yourself. One real finding and one false positive go into `PROMPTLOG.md` with your reasoning, and doing it tonight takes five minutes; doing it Sunday from memory takes an hour and reads like fiction.

Evening 6: hardening, the paperwork, and the buffer. Milestone 3 is the empty tree, the single node, the overwrite of the only key, absurdly long keys, and whatever your devlog's OPEN list has been quietly accumulating. Then finish annotating `PROMPTLOG.md` (4–6 episodes; your devlog makes this transcription, not archaeology), and check the required four are present: the plan you revised, the diff you rejected, the debugging loop with real tool output, the review finding you triaged. Write `REFLECTION.md`; half of it already exists as devlog entries, and the spec's closing question, the thing about C that most surprised the Java programmer you were two weeks ago, deserves an honest answer rather than a shrug. Copy the remaining `.jsonl` transcripts in as-is. Check the git log: comfortably past 8 commits, each one telling a small true story. Build from a clean clone of your own repo to make sure the tar you submit is the project you think it is. Then stop. If the evening ends early and you are still having fun? The Reach is sitting right there, and it is a postcard from next month. But the buffer comes first. The buffer is load-bearing.

Add it up: six evenings, none heroic, and the total lands inside the spec's 8–12 hours with slack to spare. That slack is not free time you failed to use. It is the difference between a schedule and a gamble.

8 Commit Wisely. Commit Often

The spec requires 8 meaningful commits and promises a process-grade of zero for a single "final submission" commit. Treat that as a floor with a trapdoor under it, not a target. Working the way this document describes produces 20 commits without trying, because a commit is not a ceremony. It is a checkpoint of a green state. And green states happen constantly when your slices are small.

What "wisely" means, concretely:

- One logical change per commit. "M2: delete fixup cases + table tests" is one story. "M2: various fixes" is a shrug with a hash.

- Commit only from green. The relevant battery for the slice passes. Your history then becomes a sequence of known-good states, which is the entire point of having one.
- Commit *before* anything risky. About to let Claude restructure the delete fixup? Commit first. If the restructuring goes sideways, `git restore` puts it back in two seconds, and it is the only undo button in this workflow that does not bill you. Paying tokens to have Claude un-write code it wrote an hour ago is the saddest line item in this course.
- Read the diff before you bless it. VS Code's Source Control panel shows you exactly what changed, staged and colored. Reading it there, before committing, is where you catch the agent's "helpful" edit to a test. Every diff. Especially test diffs.
- Let Claude drive git, conversationally, after you have read. "Run the full battery; if clean, commit as 'M2: delete survives the case tests'." That is the spec's Step 8, and it is a fine habit, with one non-negotiable clause: the diff passed your eyes first.

A healthy history for this assignment reads like a lab notebook that happens to compile:

```
M0: repo skeleton, frozen header, Makefile, .gitignore
M0: CLAUDE.md project law (from /init, then edited)
M1: node struct, create/insert(no fixup)/find + unit tests
M1: rotations own t->root; stale-root regression test
M1: insert_fixup + mirrors, validate checks black-height
M1: rb_foreach + in-order test; fuzzer (inserts/finds) vs model
M2: table-driven delete tests (failing, on purpose)
M2: two-children reduction + one-child splice, half the table green
M2: delete_fixup all cases + mirrors, table green
M2: fuzzer with deletes, 1e5 ops, asan+memcheck clean
M3: edge cases: empty tree, single node, overwrite-only-key
M3: post-review fixes from adversarial pass (1 real, 1 rejected)
docs: PROMPTLOG episodes 1-5 annotated, REFLECTION drafted
```

Notice what this buys you beyond the grade. When Thursday's "fix" regresses the fuzzer, `git bisect` finds the guilty commit in minutes *because the commits are small.* Your history is also your best answer at the walkthrough: asked how a piece of code came to be, you can literally show its biography! And it is corroboration for the transcript rubric, which explicitly scores claims against the repository. The grader is checking whether your story and your history agree. Make them agree by having them be the same thing.

9 Test in Increments, Not in Ceremonies

Testing on this assignment is a ladder, and you climb it one rung per slice. Never all at once at the end. The end is where testing goes to become debugging.

1. Unit tests, born alongside each operation. `rb_insert` gets tests the evening it exists, not the evening the tree is "done." Small and deterministic, each named for what it checks.

2. `rb_validate` after every mutation inside your tests. It is the cheapest oracle you own. A structural bug caught one operation after it happens is a five-minute fix; the same bug caught 4,000 operations later is an afternoon.
3. Table-driven cases *before* the hard code exists. Deletion especially. A first draft of `delete_fixup` will often pass random fuzzing yet fail the targeted cases; that is precisely the trap the tables exist to spring early, while the failure is cheap. The spec hands you the case list and tells you to add a fifth; the mirrors ride along as extra rows, which is the whole charm of tables.
4. The fuzzer against a model. Fixed seeds, so failures reproduce. When it fails at operation 6,410, your job is to shrink before you prompt: find the shortest operation sequence that still fails. Claude debugs a 12-op repro brilliantly and a 100,000-op haystack expensively, and only one of those bills is worth paying.
5. The edge cases nobody fuzzes into on purpose: the empty tree, the single-node tree, the overwrite of the only key. The autograder preview says it out loud: empty tree included, because you tested the empty tree. Milestone 3 exists because these are cheap to write and mortifying to fail.

Cadence matters as much as coverage. `make test` runs in seconds; run it constantly, after every slice. Like breathing. `make asan` after each meaningful change. `make memcheck` once per evening, as the closing ritual; valgrind is slow, and that is fine. Let it be slow at the end of the evening, like dessert!

Two standing rules, both borrowed from the spec because they are that important. First: never accept "the tests should pass now." The agent runs the commands and shows output (every time) and you paste the interesting lines back when they are not clean. Second: watch for tests getting quietly weaker. An assertion that loosens, a case that vanishes, a threshold that drifts from 10^5 to 10^3: these are real failure modes of real agents, your `CLAUDE.md` forbids them in writing. Your eyes on every test diff are the enforcement mechanism. The tests are the contract you hold the code to. Guard them like one.

10 Fixing Bugs Without Burning Down the House

There is a prompt you will be tempted to write around week two, at roughly 10:40 pm. It goes: "here is my whole codebase, something leaks, fix it." Don't write it!!! It fails on both axes at once. It is expensive, because the agent must read everything you have ever written to answer anything at all, and then you carry that entire reading forward in context for the rest of the session. And it is ineffective because an agent handed everything and told nothing will guess! And its guesses arrive as a confident multi-file diff that you now have to review, which was the work you were trying to avoid, except larger.

Debugging with an agent is the same loop debugging has always been; the agent just makes each step faster *if you do the steps in order*:

1. Reproduce deterministically. Same seed, same scenario, same failure, every run. If it will not reproduce, that fact *is* the bug report; in this assignment, nondeterminism almost always means

a read of uninitialized memory or a pointer used after free, which is to say: your code.

2. Minimize. Shrink the failing input until it is embarrassing. Twelve operations. Four. The smaller the repro, the sharper everything downstream.
3. Localize with the tools you already run. ASan and valgrind hand you file and line and an allocation backtrace. That is not decoration; that is the address of the crime scene.
4. Prompt with the artifact, scoped to the module. Name the file. Name the function. Paste the twenty lines that matter, not the two thousand that say `PASS`. `grep -A6 'definitely lost'` is free; context is not.
5. Smallest diff, verify with the same tool, commit, log it. The episode goes in `PROMPTLOG.md` while it is warm, because tool-output debugging loops are a required episode and tonight's version of you writes it in five honest minutes.

   Which turns the 10:40 pm prompt into this instead:

   > *make memcheck on the delete table reports: "40 bytes definitely lost, allocated at rbtree.c:212 in node_alloc". That allocation is the key copy, and the loss shows up only in the overwrite-existing-key case. Read only node_alloc and rb_insert in src/rbtree.c. Trace ownership of that allocation through the overwrite path, propose the smallest fix to that path only, then re-run make memcheck and show me a clean report.*

Same bug. A fraction of the reading. And an answer you can actually review, because it is the size of the question.

Two more habits for the bad nights. Press Escape the moment a response heads somewhere you did not intend. It is the cheapest key on your keyboard and it depreciates by the second. And when a long debugging thread finally lands but has left three dead theories in its wake, `/compact keep the failing test output and the current diff`, so the session stops paying rent on furniture it no longer owns.

11 Tokens, Limits, and the Meter You Cannot See

Nobody has told you what you are actually buying when you press Enter. You probably haven't taken an advanced machine learning course (CS445 or CS555) and the tool does not print a price tag. Furthermore, the whole thing feels like a website, which is to say free. It is not free. There is a meter, it runs on every message, and the difference between a student who finishes this assignment comfortably and one who spends the last weekend locked out is mostly a matter of understanding *what the meter counts*.

So here is the short version, and then the version with numbers.

11.1 What a token actually is

A language model does not read letters. And it doesn't quite read words either. It reads *tokens*: chunks of text carved up by a tokenizer, usually somewhere between a syllable and a short word.

"Tree" is one token. "Rebalancing" is probably three. A space, a newline, and a semicolon each cost you something!

For ordinary English prose, the rule of thumb is about 4 characters per token, so a thousand tokens is roughly 750 words. Call it a page and a half double-spaced.

And code is worse. And this surprises people. C source is dense with punctuation and with identifiers that no tokenizer has ever seen, so `rb_insert_fixup` lands as 4-5 tokens rather than one, and every level of indentation is billed by the character. Reckon on 3 characters per token for source, which is a polite way of saying your files are *bigger* than they look.

Thing	Rough size	In tokens, about
`include/rbtree.h` with its comments	130 lines	1,300
A finished `src/rbtree.c`	900 lines	12,000
The assignment PDF, figures and all	30+ pages	30,000+
A full `make test` run, unfiltered	1,800 lines of `PASS`	25,000
The one valgrind line you cared about	1 line	30

Look at the last two rows for a second. That ratio is the entire lesson of this section. Everything below is just elaboration with a plot.

11.2 The bill has two sides, and one of them repeats

Input tokens are everything the model reads: your prompt, your `CLAUDE.md`, every file it opened, the output of every command it ran, and its own earlier replies. **Output tokens** are what it writes back, including the reasoning it does before answering. Output is *pricier* per token. Input is where the volume lives, and volume wins by a mile!

Now the part that catches everyone. I mean everyone. The model has no memory between turns. None. Each time you send a message, the whole conversation is shipped back over and read again from the top, because that is the only way the thing knows what you were talking about. Your context is not paid for once when it arrives. It is paid for on every single turn it survives.

Which means a bloated session does not cost you extra once. It taxes you on message 4, and on message 19, and on message 40, quietly, at the same rate, *forever*, until you **clear** it.

One big file read on turn 3 is a purchase. The same file still sitting in context on turn 30 is a subscription.

Two things people confuse. It's worth ten seconds to separate them. The **context window** is a container: how much text the model can hold at once. Your **usage limit** is a budget: how much reading and writing your plan lets you do this week. You can be nowhere near filling the window and still be broke. Filling the window just means the session starts forgetting things, usually the thing you needed.

11.3 Two ways to spend a fortune before dinner

Scenario one: you attach the spec. It seems reasonable. The agent should know what the assignment says, so you drag the PDF into the chat and start working. That is roughly 30,000 tokens for the text version, and considerably more if the PDF went in as page images. Fine, once.

But it is not once. You then have a normal working evening, thirty messages of back and forth about `delete_fixup`, and every one of those thirty messages carries the full spec along for the ride. Half a million tokens, spent on a document that is sitting open on your second monitor, that you have already read twice with a pen, and that you could have quoted the relevant paragraph from for about 200 tokens!

And here is the insult on top of the injury: it did not even help. The model now has twenty pages of fixup panels and grading rubrics competing for attention with your actual question. You paid a fortune to make the answer slightly worse. This is what Section 3 was quietly protecting you from. Your notes and your `CLAUDE.md` are the compressed version of the spec, written by the one person who knows which parts matter to tonight.

Scenario two: the 10:40 pm shrug. You type "I have a leak somewhere, fix it." Here is what that actually buys.

	"I have a leak, fix it"	The scoped version (Section 10)
Agent reads	`rbtree.c`, both test files, the Makefile, the header. Call it 35k.	`node_alloc` and `rb_insert`. Under 2k.
Tool output	Full `make test` plus full `make memcheck`, unfiltered. Another 45k.	The four lines you grepped. Call it 100.
What comes back	A confident 300-line diff across four files, one of which is a test.	A 15-line change to one path.
What you do next	Read 300 lines at 11 pm, or bless them unread and fib about it in `PROMPTLOG.md`.	Read 15 lines. Run `make memcheck`. Commit.
Turn cost	∼80k, and every later turn in the session pays it again.	∼2k, and the session stays light.
After ten turns	Comfortably past three-quarters of a million tokens. Leak still there.	Bug fixed on turn two. You went to bed.

Same bug. Same tools. Roughly two orders of magnitude between them, and the cheap one is also the one you can actually review, which was the real goal the whole time. Frugality here is not a virtue you practice at the expense of quality. It is the same behavior, seen from the billing department.

11.4 Sessions, caps, and a clock that has never heard of your deadline

Your Pro plan runs on two timers at once, and you should know both before week one rather than during week two.

The first is a **session window** of about 5 hours. It starts with your first message, not at some tidy hour, and when you exhaust it you wait for it to roll over. The second is a **weekly cap** that sits across everything you do. Anthropic assigns your reset day and time to your account, so it is the same slot every week *no matter when* you happen to start working. `/usage` in Claude Code tells you where you stand, as does Settings → Usage on the web. Anthropic does not publish an exact token number for these, partly because the honest answer is "it depends on what you are doing," which is unsatisfying but true.

A few consequences worth writing on your hand:

- It is one allowance, not several. Chatting on claude.ai, working in the desktop app, and running Claude Code in your VS Code terminal all draw from the same pot. The pot does not care which window you were in.
- Attachments and conversation length are the variables. Message count is a terrible proxy. Forty short scoped prompts can cost less than five messages in a session dragging a PDF and a repository behind it.
- Model choice moves the needle hard. The heavyweight model burns the allowance several times faster than Sonnet for the same work. Routine implementation does not need it. `/model` is one deliberate decision per task, not a setting you pick once in week one out of vanity.
- The clock is wall-clock. Nothing about "this is due at 11:59 pm" persuades the meter to refill at 11:58. It is a thermostat, not a professor, and thermostats don't grant extensions.

Which brings us to the arithmetic that Section 7 was really about. Two calendar weeks of work means two weekly allowances, spent by a rested person asking narrow questions. The same work compressed into the final night means one allowance, shared with panic, spent by someone who has stopped reading diffs. Those are not the same project with different start dates. They are different projects.

And when you hit the cap, the failure mode is specific and awful: your threads go read-only, you can look but not ask, and there is no appeal. Support cannot reset it. Your TA cannot reset it. You sit there with a broken `delete_fixup` and a countdown, which is a genuinely miserable way to spend a Sunday. I say that with sympathy not smugness, because everyone learns this exactly once.

11.5 Why brute force always loses

There is a specific pattern I want you to recognize while it is happening to you, because it feels productive right up until the lights go out.

The bug persists. You say "that didn't work, try again." It tries again. Still broken. "Still broken, are you sure?" It apologizes, tries a third thing. Now it is 12:30, you have four rejected diffs

in your history, your working tree is a collage of half-reverted edits, and the leak is exactly where it was at ten o'clock.

That loop cannot converge, and the reason is structural rather than a matter of luck:

1. Every failed attempt stays in the context. So turn six is more expensive than turn five, which was more expensive than turn four. Cost climbs monotonically while progress does not.
2. You added no information. "Try again" is not a new fact. It is the same question, asked louder, and the model is drawing from exactly the same well it drew from the first time.
3. The failures are now evidence, and they are misleading evidence! Your transcript contains four confident wrong theories, written in the model's own voice, and models condition on their own text. You have been steadily building a case for the wrong answer.
4. The code degrades underneath you. By attempt five you may well be debugging bugs that did not exist at attempt one, which is a special kind of hell to be paying by the token for.

So brute force does not merely fail. It fails while accelerating! Its terminal condition is your weekly cap. Guessing is a fine strategy against a four-digit padlock, where the space is small and enumerable. A bug is not a padlock. A bug is a wrong belief you currently hold about your own code, and you cannot enumerate your way out of a wrong belief. You have to go and look.

The exit is unglamorous and takes about four minutes. Press Escape. `git restore` back to the last green commit, which costs nothing, unlike asking the agent to un-write its own work. Get a deterministic repro, shrink it until it is embarrassing, and come back with a smaller question and a file name. Section 10 spells out the loop. It is worth noticing that the disciplined version is not slower. It only feels slower, because typing furiously feels like progress and thinking looks like sitting still.

11.6 A brief and unflattering history of token maxing

A little history, because the habit you are about to be tempted by has a lineage.

In the early days the context windows were small. A couple of thousand tokens and prompting was mostly the art of deciding what to leave out. Then the windows grew. Eight thousand, thirty-two, a hundred, a million! Vendors advertised them like horsepower. "Paste your entire codebase" became a feature bullet rather than a warning.

People did what people do. They pasted the entire codebase. They pasted the novel, the transcript, the forty-file retrieval dump, all of it hauled in "just in case,". On the theory that context is information and information is good. Tooling grew up to help. Which is to say tooling grew up to make it effortless to be profligate! By the time agentic tools could read files on their own initiative, in a loop, unattended, a genuine subculture had formed around leaving one running overnight to see how much it could chew through. A small number of enthusiasts were burning thousands of dollars of compute on twenty dollar subscriptions and sharing logins to do it. Which is roughly why the five-hour window and the weekly cap exist. The sport acquired a scoreboard, and then it acquired a referee.

The punchline arrived from the research side, and it was not kind. More context frequently makes answers *worse*. Models lose material buried in the middle of a long window, and irrelevant

filler dilutes *attention* away from the thing you actually asked about. Feed a model your whole repository to fix one rotation bug and you have paid a premium for a diluted answer, and then paid again to review the extra code it wrote because it noticed some unrelated thing on the way past.

It is the all-you-can-eat buffet strategy of loading one plate with everything so you never have to walk back! The price is the same either way, sure. But the shrimp now tastes of gravy, the ice cream has gone into the soup, and you cannot actually find the shrimp. The model is standing at that plate. You built it for them.

Or if you prefer here's the version I see in the library every semester: the textbook where every single sentence has been highlighted in yellow. Enormous effort. Real money spent on markers. A page where everything is highlighted is a page with no highlighting on it at all.

Token maxing is what you do when you have confused thoroughness with volume. Thoroughness is knowing which forty lines matter. Anyone can paste four thousand.

11.7 Odds, ends, and cheap wins

Things that cost more than students expect:

- Screenshots of your terminal. Images tokenize expensively, and you had the text right there, selectable, greppable. Paste the text. Paste the twenty lines of it that matter.
- Letting the agent explore instead of telling it where to look. Every `grep` and speculative file read it does on your behalf is input you are buying. Name the file. Name the function. That "Code map" block in `CLAUDE.md` from Section 6 exists precisely to stop the spelunking.
- Build artifacts. Keep `build/` in `.gitignore` and out of the agent's way. Nobody needs to read an object file, least of all at these prices!
- Sessions that never end. `/clear` between milestones, `/compact` after a long thread lands. A session you keep open across three evenings is carrying two evenings of dead theories on every message.
- A `CLAUDE.md` that has grown a personality. It loads every session. Keep it to project law. It is a standing charge, and standing charges deserve scrutiny.

Things that are free, or near enough:

- Reading the spec yourself. Rereading your own `DEVLOG.md`. Both of these replace questions you would otherwise pay to ask.
- `grep`, `head`, `tail`, and a moment's thought about which lines of output are the interesting ones.
- `git restore`, `git diff`, `git bisect`. Your history is the undo button that does not bill you.
- The Escape key, the instant a response starts heading somewhere you did not ask for. It depreciates by the second.

- `/usage` at the start of every work session. Ten seconds. The point is to never be surprised.

One last thing, and it is the one I would like you to carry past this assignment. Everyone in this class has the same cap. Nobody is issued extra. So when you notice, in week two, that some of your classmates are finishing calmly while others are rationing messages and staring at a reset timer, understand that you are not looking at a difference in resources. You are looking at a difference in question quality, compounded over six evenings. That gap is the whole skill, and it is being graded whether or not anyone is watching.

12 Tokens: Spending Like It Is Your Money, Because It Is

You have read this far, which means the token lecture is mostly over; you just heard it wearing other hats. That is the quiet trick of this assignment, and the spec says it outright: cost scales with context. So nearly every habit that makes you a disciplined engineer also makes you a frugal one. Here is the ledger, assembled in one place:

Habit	Why it saves tokens	You were doing it anyway because...
Small modules, named in prompts	The agent reads one function, not a repository.	Section 6: maintainability.
Plan mode before code	Reading and thinking are cheap; un-writing 400 confident wrong lines is not.	The spec's Step 3, and your grade.
Tests-first, smallest slice	Small diffs are reviewed once, not three times.	Section 9.
`/clear` per milestone, `/compact` after long threads	Stale context is billed on every message, forever, like a storage unit.	Section 7: fresh sessions keep the agent sharp.
Filtered tool output	Twenty relevant lines beat two thousand `PASS`es. `grep` is free.	Section 10.
Commit before risk; `git restore` over regeneration	Reverting is free. Regenerating is the same wrong answer, purchased twice.	Section 8.
A devlog instead of asking Claude what you did	Your notebook is free memory. The context window is the expensive kind.	Section 13, and the quiz.
Lean `CLAUDE.md`	It loads every session; every line is a standing charge.	You dislike relitigating brace style at breakfast.

To that ledger, add the meter itself. `/usage` at the start of each work session, ten seconds, so the weekly cap is never a surprise. `/context` whenever a session feels sluggish, because a ballooning context is something you want to notice before it forces a compaction, not autopsy afterwards. And `/model` deserves one deliberate choice per task: routine implementation does not need the heavyweight model. Save it for the genuinely hard reasoning in this assignment, which is roughly one thing: arguing through the deletion-fixup cases. Everything else, Sonnet handles happily at a lower burn.

But the single biggest lever is not a command at all. It is the calendar. Spread across two weeks, this assignment fits inside the allowance with room to argue about tabs versus spaces. Compressed into a single night, it meets both clocks at once. The spec has already told you which of those clocks has no email address.

13 Write It Down While It Is Warm: The Devlog

You will forget what you built. Not might. Will. The quiz at the walkthrough arrives weeks after you wrote the code it asks about, the next assignment will drag this exact tree through kernel-style constraints next month, a near-twin of it will eventually sit inside the Assignment 4 scheduler, and if life intervenes and you have to set the project down and pick it up later, future-you inherits whatever present-you bothered to write down. Memory is not a plan. A devlog is.

So: `DEVLOG.md`, in the repo, five minutes at the end of every session, before the laptop closes. Not prose. A dated entry with the same small skeleton every time:

```
## 2026-09-01 (evening 4, ~2h)
STATE: delete table half green; fuzz fails at op 6410, seed 1337.
DID: two-children reduction + one-child splice; transplant helper.
DECIDED: single shared NIL sentinel (not per-leaf) -- validate
  counts it once at the bottom of every path. See spec, invariants.
LEARNED: my "case 3" tested the sibling's wrong child in the
  mirrored branch; rb_validate caught it via black-height, not
  ordering. Validate-every-100-ops earns its keep.
NEXT FIRST STEP: shrink the seed-1337 failure to <15 ops,
  then reread the deletion cases before prompting.
OPEN: on overwrite, old value freed before or after new key
  alloc succeeds? (Header says value not consumed on failure;
  so: after. Confirm in tests.)
```

The `NEXT FIRST STEP` line is the one that pays compound interest. Every session opens by reading it, which means no session opens with twenty minutes of "where was I". No resumed project, after a week or a month away, opens with an evening of rediscovery. It is the difference between a five-minute restart and an archaeology dig through your own git log.

Alongside the log, keep decision records: one-liners, written the moment you choose. "Parent pointers: yes. The textbook fixup climbs toward the root, and I would rather keep one more pointer honest through every rotation than re-descend from the top to find my ancestors." One sentence. That sentence is your walkthrough answer, pre-written by the only person who ever knew the real reason. And the invariant comments your `CLAUDE.md` style rules already require are the same idea

living inside the code, sitting exactly where the quiz will point.

Because here is the compounding part: the devlog is not extra paperwork on top of the deliverables. It is what the deliverables are smelted from. `PROMPTLOG.md` annotation becomes transcription instead of archaeology. `REFLECTION.md` is half-written before you start it, and its closing question about what surprised the Java programmer in you has been answering itself one LEARNED line at a time. The live modification at the walkthrough starts from your notes instead of from zero. Five minutes a night, and every downstream document gets cheaper. That is the best exchange rate in this course.

14 Your Prompt Log Is a Professional Artifact

One last reframe, and it is the one to carry out of this course. The transcript you are producing here is not homework exhaust. It is a genre of document your career is going to be full of.

In teams that work with coding agents, and that is rapidly becoming most teams, session transcripts get shared, reviewed, inherited, and mined the way pull requests are. A tech lead reads them to understand how a problem actually got solved. A teammate inherits your session to continue your work, and inherits your reasoning with it. Token spend shows up as a line item that somebody with a budget looks at. And when a bug ships, the transcript is the accountability trail: it shows what was reviewed, what was merged on faith, who decided, and when. "The agent wrote it" is not a defense in any code review on Earth, which is exactly the spec's you-own-the-bugs rule, wearing a badge.

Which means your prompts are legible evidence of how you think. A transcript full of "fix it" and "make it work" reads one way. A transcript where every prompt names a file, cites a failing test, states the constraint, and pushes back on a hand-wavy plan reads another way entirely, and the person reading it is forming an opinion of you either way. The habits are the same ones that get sanded into you privately; the stakes just grow teeth once the audience does. Because habits are precisely the things that follow you into rooms where they matter.

So use this small, five-point assignment as the rehearsal it is. Write every prompt as if a senior engineer you respect will read it later, because the TA literally will, and someday the senior engineer literally will too. Annotate `PROMPTLOG.md` the way you would annotate a design doc for a teammate: here is what I asked, here is what came back, here is my judgment and my reasons. Look at the transcript rubric's level 5 and notice that it is not describing a grading gimmick. It is describing what a good engineering session looks like anywhere. The rubric is a job description in disguise.

15 How AI was Used in Developing This Companion

This companion document was developed with assistance from AI tools. Its structure, recommendations, examples, sequencing, and pedagogical choices are mine. I used Claude and Claude Code primarily to stress-test explanations, check technical details, identify ambiguities, and see what happened when the guidance was followed literally. I then revised. The document then went through several additional rounds of revision, including review by the TAs, who caught issues and edge cases that I had overlooked.

That process is worth mentioning because it mirrors what I am asking you to do. Good work rarely arrives fully formed. You build something, test it, let other eyes find what you missed, revise it, and then do the whole thing again. This companion got better through exactly that process. Yours should too.

16 A Last Word

The spec grades two things: the tree and you. This companion has been almost entirely about the second one, because the second one is the part that transfers. Trees are forgotten. The shape of *how* you work is not.

So: read the whole thing before you type. Turn your reading into notes, and your notes into plans. Predict the fixup panels before you scroll past them, and make the counting arguments yours. Build in modules, and only the modules the current milestone needs. Let tests lead and keep the diffs small; make the agent show its work, every time. Commit every green state like the checkpoint it is. Debug with a repro and a file name, never with a shrug and a repository. Watch the meter without worshipping it. And spend five minutes each night telling the future-you what happened, because that future-you is the one taking the quiz.

None of what I have laid out here is heroic. That is the point! Six calm evenings beat one "legendary" weekend every single time. Only one of those makes a good story at the walkthrough. Go start your repo!

17 Version History

This section reflects the change history for this companion document. Changes clarify the walkthrough; the spirit of it will not change, and nothing in it ever overrides the assignment specification.

Version	Date	Comments
1.0	8/25/2026	First public release of the companion walkthrough for Assignment 1.