Assignment 1

Building Your First Red-Black Tree in C

Pointers, ownership, memory-safety verification, and disciplined AI-assisted development

Version 1.0 August 25, 2026

Version history appears in Section 22.

Contents

Nota Bene: This is the first assignment of the course, and the gentler half of a pair. Here you build a red-black tree in C and learn to keep every byte of it accounted for. In the next assignment, the same tree gets dragged through kernel-style constraints: allocators you write yourself, allocation that fails on purpose, stacks too small to recurse on, and trees that quietly share nodes behind your back. So nothing you build here is disposable. The tree you write this week is the tree that gets mutated next month, and a near-twin of it will eventually sit at the heart of the CPU scheduler you build in Assignment 4. Understand it now and you will lean on it then. Borrow it now and you will have to explain it then, to a version of yourself who has forgotten where it came from. Two things about this assignment are genuinely new for a CS course here. You will be writing C, most of you for the first time, and Section 4 exists to break your Java habits before they break your tree. And 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 tool many of you already reach for, with varying degrees of permission, is now the one I am openly asking you to master. That turns out to be a very different assignment.

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 will build the tree once, and for this assignment that is the whole job: make it grow, make it answer, make it shrink, and make it give every byte back when it is done. The kernel-style torments (custom allocators, injected allocation failures, stacks too small to recurse on) are waiting for you in the next assignment, where this exact tree is the patient. The structure matters beyond our classroom, 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 *ownership*. Writing a red-black tree that returns the right answer is the familiar part; you did harder things in your data structures course. But Java cleaned up after you, silently, every single time, and that particular kindness ends now. In C, every node you allocate is yours until you free it. Every key you copy is yours until you free it. Nobody else is keeping track, because there is nobody else. A tree that looks up the right key but leaks, double-frees, or reads memory it already gave back 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 **5 points**, and those points answer two different questions, calling for two different kinds of evidence.

4 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 your test suite, your fuzzer, and AddressSanitizer and valgrind.

The remaining **1 point** answers a harder question: *do you understand what you built?* Half of it comes 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. The other half comes from a short quiz generated from your own code, administered at a live walkthrough: your own function names, your own error-path structure, what frees a given allocation on a given path, why a particular node gets recolored. 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

- Read and write enough C to build a pointer-linked data structure, and know at every moment which piece of code owns which allocation.
- Implement a red-black tree in C and reason precisely about node ownership, lifetimes, and the constant-time pointer surgery in rotation and fixup.
- 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
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 8–12 hours across three milestones, including required use of Claude Code
Tooling	Claude Code is required for this assignment (see Section 11)

3 What is a Red-Black Tree?

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*.

One small convention before we go any farther. NIL is the black "nothing is here" at the end of a branch. Your implementation may represent it with an actual shared sentinel node, or with `NULL` that you consistently treat as black. Either representation is fine unless the frozen interface says otherwise. What is not fine is sometimes treating `NULL` as black and sometimes forgetting that you did. Red-black trees are remarkably intolerant of philosophical inconsistency.

Textbooks also differ slightly on whether the starting node itself is included when they report a numerical black-height. In the figures here, I count the black nodes you actually cross on the displayed path, including NIL. Do not get hung up on the convention. The invariant that matters is that all paths descending from the same node have the *same* black count.

Rules 4 and 5 together squeeze the tree's shape. Rule 5 says that competing paths must contain the same number of black nodes. Rule 4 says that between two black nodes you can slip in at most one red node, because two reds may never sit consecutively. So the longest possible root-to-NIL path alternates red and black, while the shortest uses only black nodes. The long path can therefore be at most twice the length of the short one. 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. Figure 1 shows a valid tree. 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.

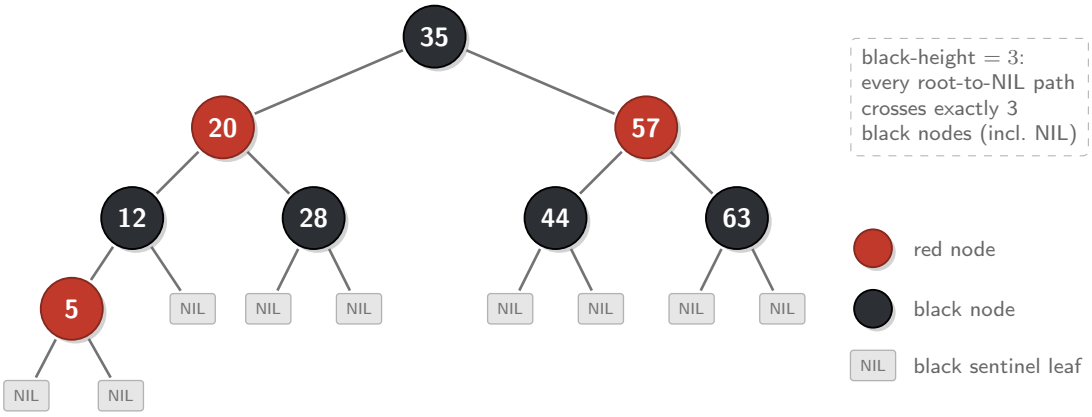


Figure 1: A valid red-black tree over integer keys. No red node has a red child, every root-to-NIL path crosses the same number of black nodes, and in-order traversal yields the keys in sorted order. The path through 5 is the longest and the path to 28's children is among the shortest, but the two differ 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. Notice that these are different kinds of damage. Insertion usually gives you a bad *pair of nodes*; deletion gives you a bad *count along a set of paths*. That difference is why deletion feels so much less friendly. 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. Deletion is famously the fiddly one, and its case analysis is where both students and AI agents most often go subtly wrong, which is precisely why this assignment requires all of it.

One repair (worked in full) before this gets any more abstract. Figure 3 shows the gentlest of the insertion cases. A new key always enters the tree as a red node at a leaf position. The reason is strategic: making it red leaves the black-height of every existing path unchanged. The price is that, if its parent is also red, you may violate rule 4 at exactly one new edge. If the new node arrived black instead, the path containing it would immediately gain one black that its neighboring paths did not. When the new node's parent and uncle are both red, the repair moves no pointers at all: recolor the parent and uncle black and the grandparent red, then look again two levels up. When the uncle is black, recoloring cannot help, and one or two of the rotations from Figure 2 finish the job instead. That is the entire toolkit. Deletion uses the same two tools with a meaner case analysis, and you will meet all of it in Milestone 2. And if the case analysis reads like a ritual at first, that is normal; asking Claude Code to walk you through it, case by case, with you predicting each step before it answers, is Step 2 of the workflow and among the best hours you will spend on this assignment.

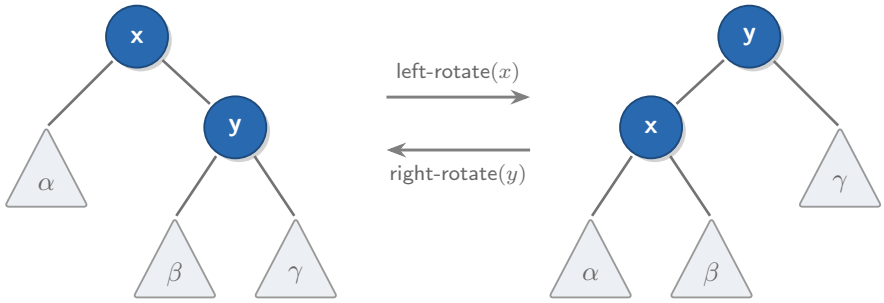


Figure 2: Rotation, the balancing primitive. the level of the tree pictured here, three child relationships change: y rises, x sinks, and subtree β changes parents. Your C implementation may require several additional pointer assignments to repair `parent` links and, when the rotation occurs at the top, the tree's root pointer. Do not confuse 'constant time" with 'three lines of C." 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.

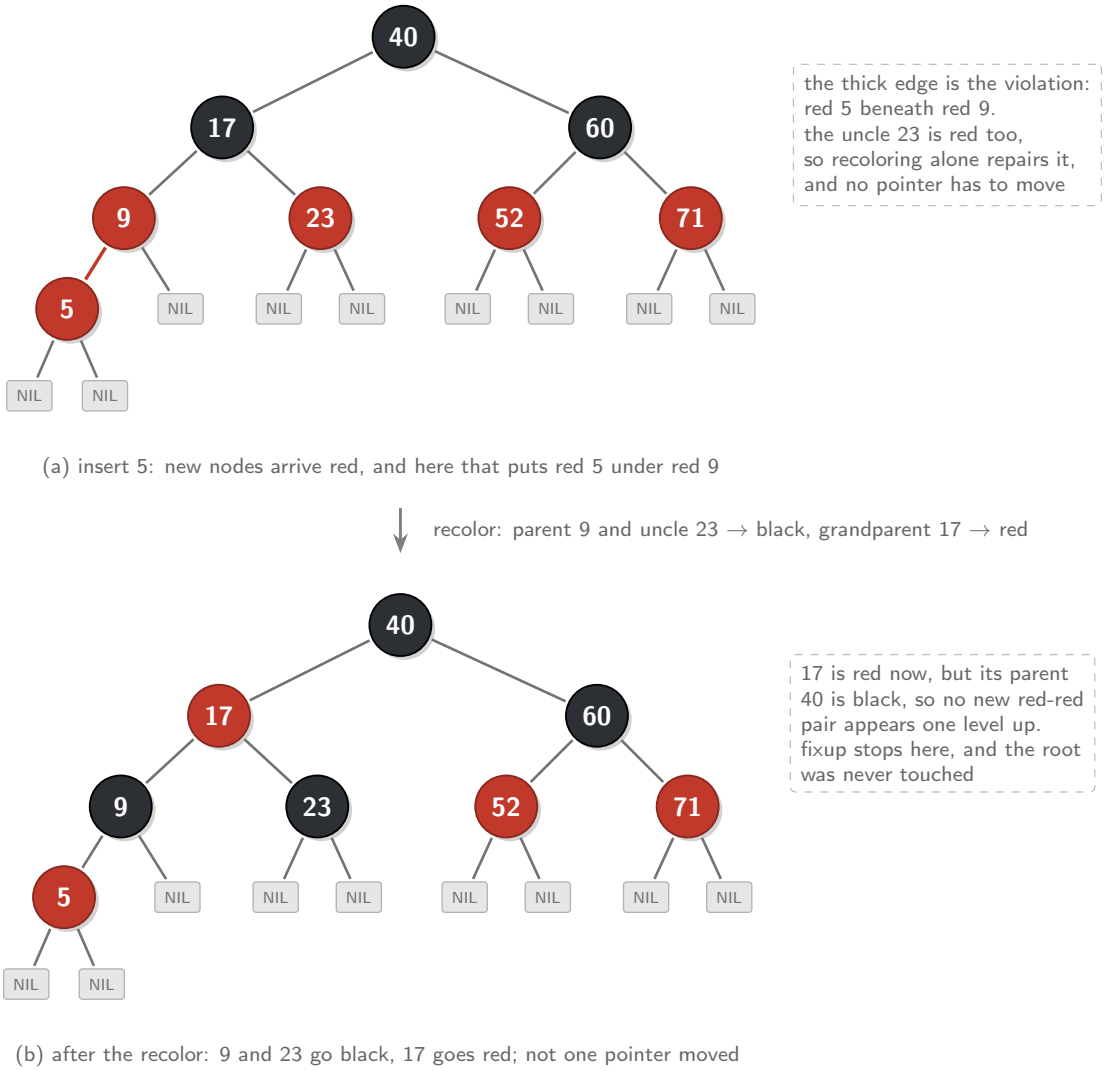
A harder insertion, in which recoloring only moves the problem

Figure 3 was the friendly one. One recolor, done, and the root never found out. Real inserts are not always that agreeable, and the interesting ones all share a feature that is easy to skim past: insert fixup is a *loop*. The red-uncle case repairs the violation you are looking at, but it does not necessarily finish the job. It can create the same red-red problem two levels higher. In other words, it tidies your desk by putting the mess on the desk upstairs.

Before the example, a small counting fact that explains why the rotation cases feel so weirdly rare when you test insertion by hand. You have just put a red node at a leaf and its parent is red. What color is the uncle? Count blacks. From the grandparent, the path down through the parent crosses a red parent, a red new node, then a NIL, so exactly one black. Every path through the uncle has to cross one black too. But a real black uncle contributes one all by itself and its NIL descendants contribute another, which is two, and that means the tree was already broken before you laid a finger on it. It wasn't. So on the first pass, a black uncle is *always* NIL. If your implementation represents missing children with `NULL`, read 'NIL" here as 'that `NULL` child, treated as black." The tree-theoretic argument does not care how C happens to spell the missing child. No exceptions, no clever counterexample waiting for you at 2am. The rotation cases with a substantial black uncle and actual subtrees hanging off it cannot show up until the loop has already climbed at least once. Which is a fine argument for testing insertion with more than five keys and a hopeful expression.

Figures 4 and 5 run one insert all the way through, and it uses every tool insert fixup owns. We start from a legal tree and add key 25, which lands as a red left child of red 30. Uncle 40 is red, so panel (b) is the move you already know: parent and uncle go black, grandparent 35 goes red, no pointer moves. And now look where the trouble is. It is not gone. It is at 35, whose parent 20 is also red, two full levels above where you were a moment ago.

So the loop goes around again, and this time everything is different. The node in question is 35, an interior node with real children. Its uncle is 70, black and genuinely there. Recoloring has



(a) insert 5: new nodes arrive red, and here that puts red 5 under red 9

(b) after the recolor: 9 and 23 go black, 17 goes red; not one pointer moved

Figure 3: Insertion fixup, the red-uncle case, drawn on a complete tree so that no node's position is in doubt: 40 is the root throughout, and the repair happens well below it. Every path from 17 downward still crosses the same number of black nodes after the recolor, so rule 5 survives untouched, while the red-red violation moves two levels up, to the grandparent. Here 17's parent 40 is black, so the climb ends immediately; had 40 been red, the same test would run again one grandparent higher. The loop therefore climbs toward the root and cannot run forever, and if it ever arrives at the root, the last step simply paints it black (rule 2). When the uncle is black instead, recoloring cannot help and the rotations of Figure 2 take over; insert fixup never needs more than two of them.

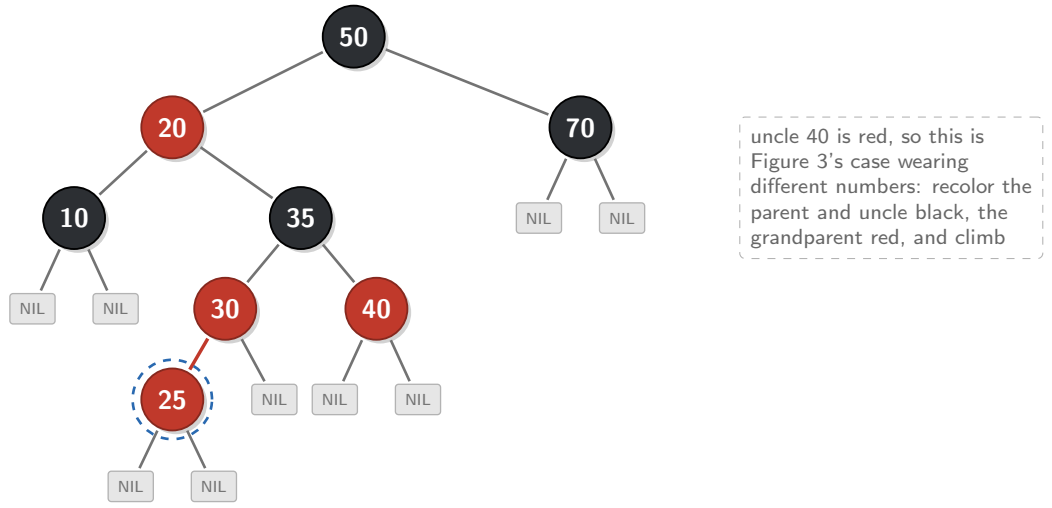
nothing left to offer. If you simply paint 20 black, the left branch through 20 gains one black. The right branch through 70 gains nothing, because 70 was black already. You would fix the red-red violation and immediately create a black-height violation. This is not an upgrade.

What you need is a rotation, and here the tree makes you work for it. 35 hangs off the right of 20, while 20 hangs off the left of 50. That kink has a name in every textbook and none of the names agree: zig-zag, triangle, inside case, left-right. Whatever you call it, a single rotation at 50 will not fix it, and if you try it you will get the mirror image of the same problem and a growing suspicion that the algorithm is trolling you. The answer is to spend one rotation buying a shape you already know how to handle. Left-rotate at 20 (panel c) and the three nodes line up on one diagonal. The violation is still there. It is just tidy now.

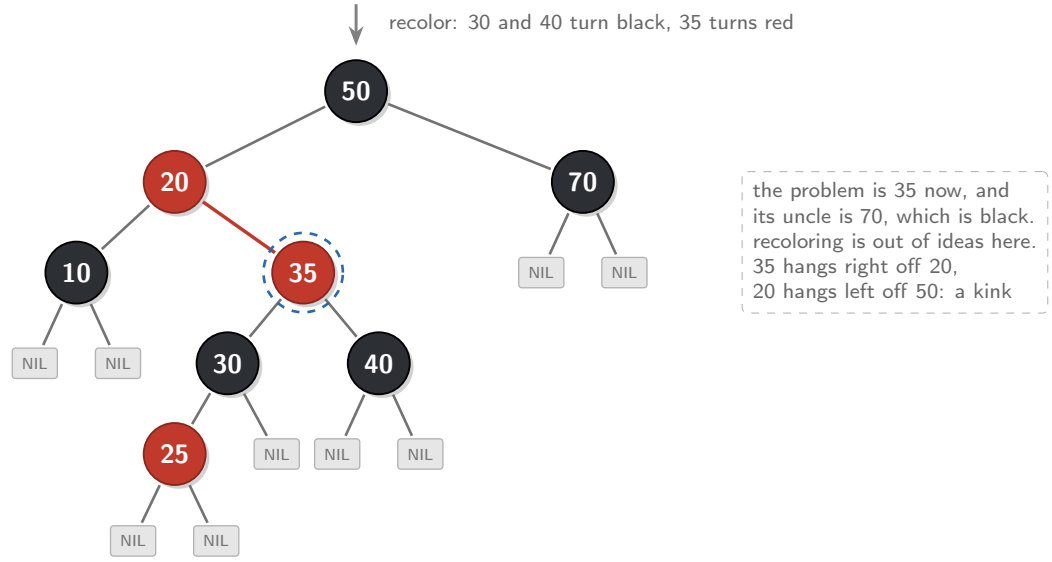
Then panel (d) collects. Right-rotate at 50, swap the colors of 35 and 50, and the red-red pair is gone with the black count on every path exactly where it started. Two rotations, which is the most insert fixup will ever need, so you have now seen the worst insertion can do to you.

One more thing. 50 was the root when this started, and it isn't anymore. If your fixup only ever talks to parents and children, the tree structure is perfect and your `rbtree_t` is still pointing at 50, which is now a mid-level node with a red hat on. Every subsequent `rb_find` then searches a subtree instead of a tree, and half your keys politely vanish. A correctly written `rb_validate` should save you here, because the object recorded as the root is red, violating rule 2. If you maintain parent pointers, it will still have a parent as well. If your validator cheerfully accepts this state, congratulations: you have found two bugs for the price of one. Still, do not make the validator responsible for keeping `t->root` correct. The rotation owns that job. The validator merely gets to complain afterward. This is a great bug. You should have it once, briefly, and never again.

The mirror image (a right child of a right child, needing a left rotation at the grandparent) is the same code with `left` and `right` swapped, and it is the half people forget. Do not forget it. And it is worth handing this exact sequence to Claude Code in plan mode, asking it to predict each panel before you scroll, because arguing about 35 for ten minutes now is considerably cheaper than discovering at the walkthrough that you learned deletion fixup and merely memorized this.

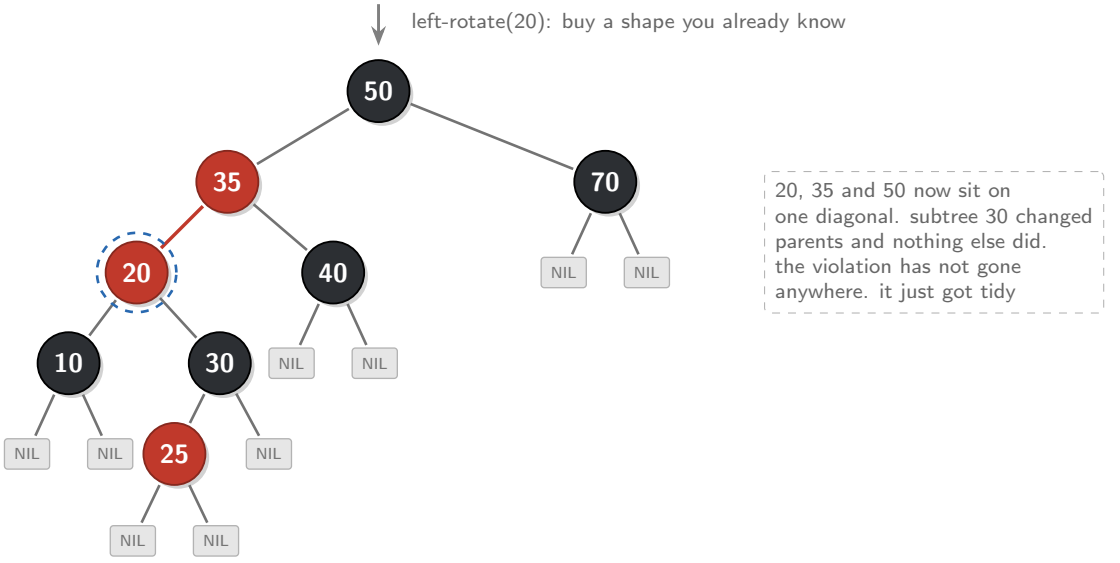


(a) insert 25: red beneath red 30, and the uncle 40 happens to be red

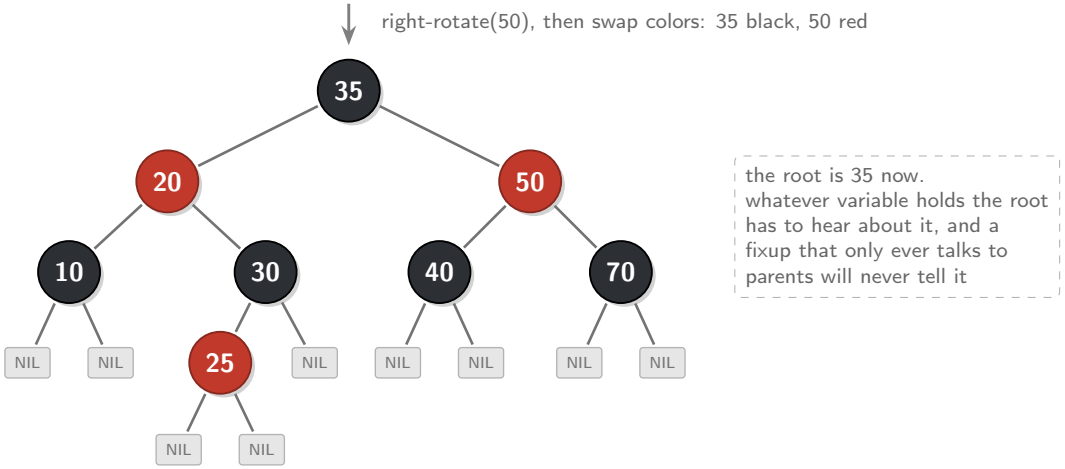


(b) two levels up, the violation reappears, and this time the uncle is black

Figure 4: A harder insertion, part-one: the recolor that solves nothing. Panel (a) is the familiar red-uncle move, and it is genuinely correct; every path below 35 still crosses the same number of black nodes afterward. But the loop does not exit. Panel (b) shows what it actually did, which was to relocate the red-red pair from the edge 30–25 to the edge 20–35, two levels closer to the root. Now the node under inspection is an interior node with real children, and its uncle 70 is a black node rather than a NIL, which is a configuration that cannot occur on the first iteration at all. Recoloring 20 and 70 black would fix the red-red pair and immediately break rule 5, since the right branch would gain a black node the left branch never gets. Rotation is the only tool left, and the shape has to be straightened before it can be applied. That is Figure 5.



(c) after left-rotate(20): same red-red pair, better geometry



(d) after right-rotate(50) and the color swap: legal again, and the root has moved

Figure 5: A harder insertion, part-two: the two rotations, picking up from panel (b) of Figure 4. The kink comes out first. Left-rotating at 20 in panel (c) puts 20, 35 and 50 on a single diagonal and moves subtree 30 across to 20, which changes nobody's black count because 35 was red and contributed nothing to begin with. The red-red pair survives that rotation on purpose. Panel (d) then does the real repair: rotate right at 50, paint 35 black and 50 red, and the black that used to sit at 50 now sits at 35, one level higher and on every path rather than just some of them. Count them yourself; three from the root to any NIL, by any route. Two rotations is the ceiling for insert fixup, so this is as bad as insertion gets. Note what panel (d) costs you in bookkeeping, though. 50 entered as the root and left as a child, and if your `rbtree_t` is still pointing at it, you now own a beautifully balanced tree that `rb_find` can only see two thirds of. `rb_validate` will happily certify the wreckage, because starting from the stale root it is looking at a perfectly legal red-black tree. It is simply not yours.

Deletions in the red-black tree

Now the unpleasant direction. Insertion breaks a rule you can *see*: a red node lands on top of a red node, at one edge, and the violation has an address. Deletion is not so courteous about it.

But some of it is free, and knowing which parts are free is worth doing before you panic. Removing a red leaf costs nothing as far as the red-black invariants are concerned: unhook it, free it, and get on with your day. The physical path becomes one edge shorter, of course, but no path loses a *black* node. So, black-height does not change. A node with two children isn't really a deletion either. You find its in-order successor, hoist that successor's key and value into the doomed node's slot, then delete the successor instead, which by construction has at most one child. In C, "hoist" is doing some work in that sentence. You are transferring the logical payload while preserving your ownership rules. Do not accidentally create two nodes that both believe they own the same key allocation and then discover, during destruction, how strongly both of them hold that belief. So every deletion, however it starts, funnels down to a single question: what do you do when the node you actually unlink is black?

What you have then is a counting error. One path lost a black node, every other path kept theirs, and there is nothing local to look at. No red sits on a red. Each node, examined by itself, is perfectly legal. The tree is just quietly one black short down a single line, and the standard way to reason about that is to pretend the empty slot carries a debt: it is *doubly black*, holding an extra unit of blackness with nowhere to put it. Fixup is the business of moving that debt somewhere it can be paid off.

Doubly black is a reasoning device, not a third value you need to add to your color enum. Your code needs to know *where* the missing black is and who its parent and sibling are. The "double black" exists in your accounting, not necessarily in a field inside a node.

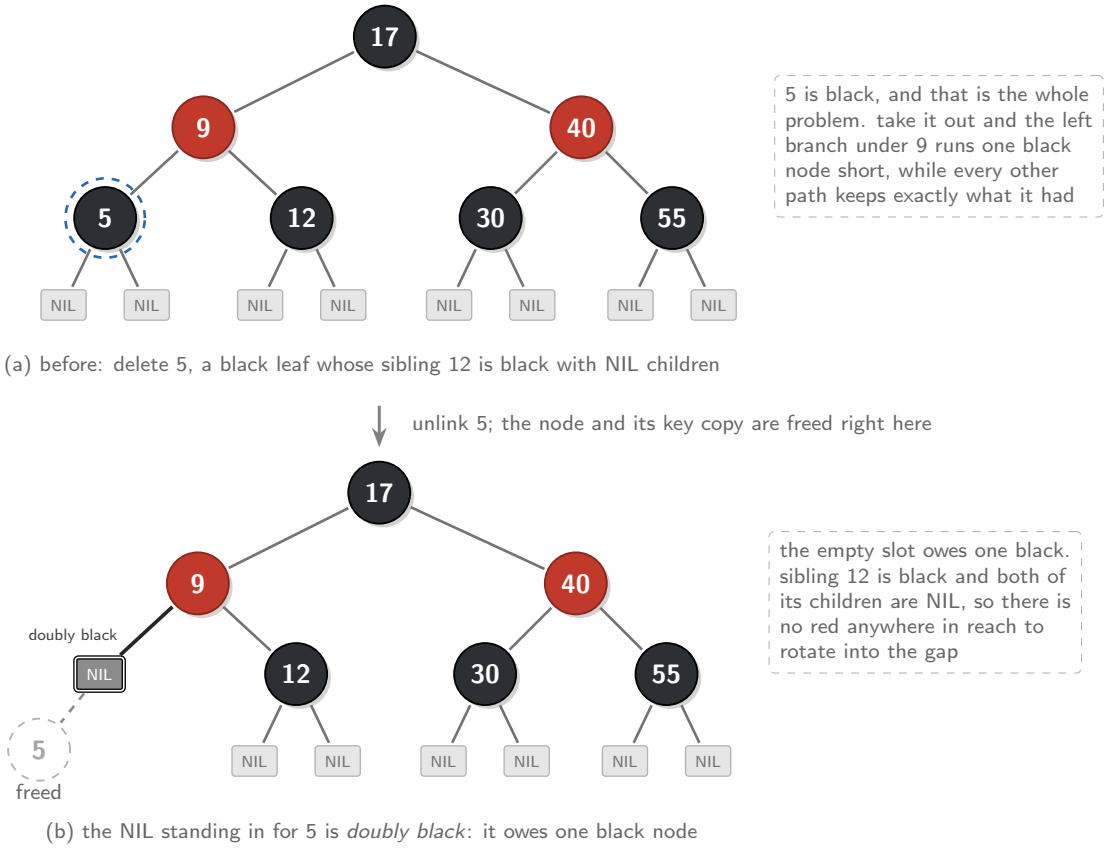
Figures 6 and 7 pay it the cheap way, and together they are the deletion twin of Figure 3 in that not one pointer moves. Black leaf 5 comes out, leaving a doubly-black hole under red 9. Look at the sibling, 12, because that is where the decision lives. It is black and both of its children are NIL, so there is no red anywhere in reach to rotate into the deficient side. What's left is subtraction. Paint 12 red, which strips a black off the right branch too, and now the two branches under 9 agree with each other while the subtree as a whole sits one black short of where it started. The debt has moved up to 9. And 9 is red, so it turns black and swallows the debt whole. Two recolorings, no pointer surgery, and the root never found out.

Where does it get mean? Everywhere else. If the sibling has a red child, recoloring can't save you, and one of the rotations from Figure 2 has to drag that red across to the side that's short. If the sibling itself is red, you rotate first purely to reach a configuration you already know how to think about, which feels like cheating and isn't. And if the parent is black rather than red, nothing absorbs the debt at all: it climbs, and the whole case analysis runs again one level up, possibly all the way to the root. If the debt reaches the root, it disappears there. Another way to say the same thing is that every root-to-NIL path in the affected tree has now lost the same one black, so equality has been restored. The absolute black-height may have dropped by one. The invariant only demands that the competing paths agree.

One detail in panel (b) deserves a longer look, because it will bite somebody in this class. In the sequence drawn here, node 5 has already been freed while the tree is still temporarily illegal, and fixup proceeds using the surviving slot, parent, and sibling information. That ordering is legal

if you saved everything you need before the free. You may instead structure your implementation so that the removed node is freed after fixup. Either way, one rule is absolute: fixup must never dereference memory that has already been freed. A pointer kept "just for a second" does not receive a grace period from C. ASan will explain this to you with characteristic warmth.

Count the deletion cases however you like. CLRS says four (The book *Introduction to Algorithms* is widely known as CLRS after its authors Cormen, Leiserson, Rivest, and Stein) . Wikipedia says six. Your own code at 2am will appear to have eleven. They are the same small handful of shapes wearing different hats, and the fastest route to believing that is to have Claude Code walk you through them one at a time while you predict each move before it answers. Get it wrong out loud. That's Step 2, and it is a great deal cheaper here than at the walkthrough.



(a) before: delete 5, a black leaf whose sibling 12 is black with NIL children

(b) the NIL standing in for 5 is *doubly black*: it owes one black node

Figure 6: Deletion fixup, part-one: the damage. Drawn on a full tree so that no node's position is in doubt, with 17 as the root throughout; it never changes color. Removing black 5 leaves the left branch under 9 one black node short of the right, and unlike a red-red violation there is no offending pair to point at. The damage is a counting error spread across paths, which is why the accounting fiction of a *doubly black* slot earns its keep. Sibling 12 is black with no red child, so nothing can be rotated into the gap and recoloring is the only tool left on the bench. Notice also what panel (b) quietly admits about memory: node 5 and its key copy are already freed while the tree is still, for a few instructions, not a red-black tree. That gap is normal. It is also exactly where a pointer held one moment too long turns into a use-after-free. The repair itself is in Figure 7.

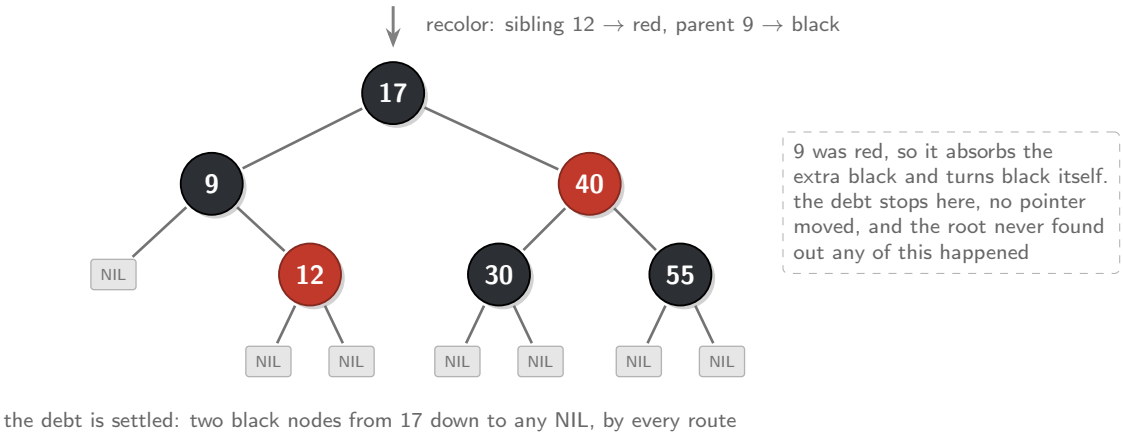


Figure 7: Deletion fixup, part-two: the repair, picking up from panel (b) of Figure 6. Painting 12 red strips a black off the right branch as well, which levels the two sides under 9 at the price of leaving the whole subtree one black short of where it started, and that hands the debt upward. 9 is red, so it turns black and settles the debt on the spot. Not one pointer moved, which is what makes this the deletion twin of Figure 3. Had 9 been black instead, nothing would have absorbed the debt and the same case analysis would run again one level higher, at 17; had 12 or either of its children been red, this would have been one of the rotation cases and the picture would have arrows in it.

Deleting a black node with exactly one child

There is one corner case hiding inside what I just told you was the easy part: a black node with exactly one child. Not a leaf. Not a two-child node. Just the awkward middle child of the deletion taxonomy. Start by asking what color that only child can be. Suppose it were black. Then the path down through it crosses at least two black nodes on the way out of the parent, while the other side, which is nothing but NIL, crosses one, and the tree would have been broken before you laid a finger on it. It wasn't broken. You have been maintaining it correctly this whole time. So the child is red, and it had no say in the matter. A red node's children are NIL, which means the entire subtree hanging beneath our condemned black node is one red node and then the ground.

Which makes the repair almost insulting. Unhook the black node, move the red child into the vacancy, and paint it black. The child now carries exactly the black color that the parent was carrying, so the count along that path is untouched, and no path elsewhere in the tree ever learns that anything happened. No debt. Nothing to climb. Notice what never entered the argument: the sibling's color, and the parent's, neither of which mattered here.

Figure 8 does it on the tree you already know. And do save the child pointer *before* you free the parent, not after, because for one instruction that pointer is the only thing standing between you and a subtree nobody can reach anymore. ASan is very good at this particular disappointment.

Now the part that makes this worth a figure rather than a footnote. The case is not exotic. It is the ordinary outcome of the two-children reduction from a paragraph ago: you promote the successor's key and value, then delete the successor, which by construction has at most one child, and when it does have one, that child is red. You are standing right back here. So the shape you might file under curiosity is the shape your delete path walks through on a regular Tuesday, arriving

by the side door.

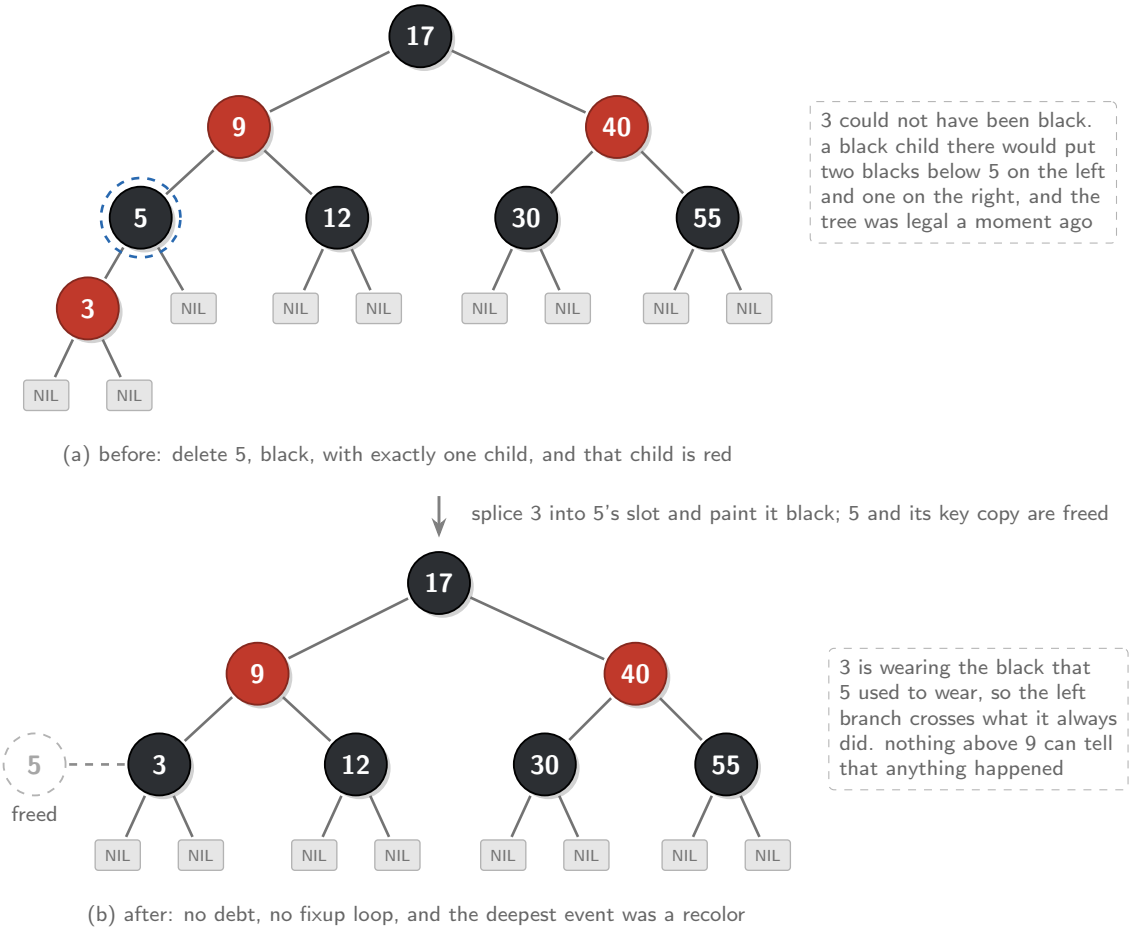


Figure 8: Deleting a black node with exactly one child, drawn on the same tree as Figures 6 and 7 with one red node hung beneath 5. The child's color is forced rather than chosen: a black child there would already have broken rule 5 before the deletion began, so red is the only possibility, and a red node's children are NIL, which is why the whole subtree under 5 is one node deep. The repair is a splice and a recolor. 3 takes the vacant slot, turns black, and contributes precisely the black-height 5 was contributing, so no ancestor needs adjusting and no debt is ever created. Contrast Figure 6, where the unlinked black node had no red child within reach and the deficit had to be carried upward instead. Panel (b) is also where pointer discipline earns its keep: 5 is freed here, and the address of 3 had better have been saved before that happened.

Which is a good argument for testing it on purpose. Milestone 2 asks for a red leaf, a black leaf with a red sibling, a node with two children, and the root; add a black node with a single red child to that list. A textbook-style implementation may still call its deletion-fixup routine here, and that is perfectly fine. The important part is what happens next: because the replacement child is red, the usual doubly-black loop should not begin. Painting that child black settles the missing black

immediately. If your code launches into the sibling case analysis anyway, *then* something has gone sideways. Better to learn that now than in front of us. The mirror image, red child on the right, is the same move with the fields traded, and skipping it is a rite of passage you are allowed to decline.

That is the whole idea. 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 next section is about learning to see the circles that way.

4 From Java to C: A Field Guide for the Formerly Garbage-Collected

You know Java. That is genuinely useful here: you already think in objects linked by references, and a red-black tree is exactly such a thing. But Java also spent years telling you a comforting story about memory, and this section exists to sort out which parts of the story were true and which parts were the runtime quietly doing your chores. The C syntax is not the important leap. The important leap is that every reference-like thing now comes with two questions Java mostly answered for you: (1) *how long* does the thing being referenced remain alive, and (2) *who* is responsible for eventually giving its memory back? Read it before you write a line of C. Then reread it the first time valgrind says something you do not believe.

4.1 The variable is not the object

In Java, a variable of type `Node` was never a node. It was a reference: an opaque handle, managed for you, pointing at an object living somewhere you were never allowed to look. C has the same idea with the chaperone removed. A *pointer* is a variable whose value is a memory address, and the language will happily show that address to you, and just as happily let you aim it at complete nonsense. Keep the pointer and the thing it points to separate in your head. Copying the pointer copies an address. It does not copy the object at that address.

Two operators do most of the daily work. `&x` produces the address of `x`; `*p` follows the pointer `p` to whatever lives at that address. The asterisk does double duty in C, which is mildly rude to beginners. In a declaration such as `struct rb_node *p`, it says that `p` is a pointer. In an expression such as `*p`, it means "follow that pointer." Context tells you which job it is doing. Because nearly everything in this assignment is "follow a pointer to a struct, then touch a field," C supplies an abbreviation: `n->left` means `(*n).left`. You will type that arrow several hundred times before the semester ends, and you will stop noticing it by Thursday.

Here is the comforting part. Aliasing behaves the way you already expect: assigning one pointer to another copies the address, not the node, exactly as assigning Java references does. Two names, one object, and a change through either is visible through both. What is new is the bookkeeping around that alias. Two pointers may both reach the same object, but that does not mean both are allowed to free it. Reachability and ownership are different ideas. The rest of this section is mostly about learning not to confuse them.

4.2 Memory you must give back

Java's deal was simple: say `new`, use the object, walk away. A garbage collector watched what was still reachable and swept up behind you. C's deal is simpler still, and much colder. `malloc(n)` finds you `n` bytes on the heap and hands back their address. Allocation and initialization are separate acts here. `malloc` gives you storage; it does not give you a constructed node, sensible field values, or an owner who will clean up after you. The bytes are not zeroed and not constructed; they are whatever the previous tenant left behind. When you are done, you call `free(p)`, exactly once, and never again touch anything that pointer reached. Nobody reminds you.

Because there is no collector, the failure modes have names, and this course will make you intimate with every one of them. Forget to free and you have a *leak*: memory still allocated, no longer reachable, dead weight until the process exits. Free twice and you have a *double free*, which corrupts the allocator's own bookkeeping. Free and then keep using the pointer anyway and you have a *use-after-free*, the most treacherous of the family, because the program frequently keeps working. For a while. In the wrong place.

One more Java habit to unlearn: ordinary local objects have an automatic lifetime. When the function or block that created them ends, those objects are gone. On the systems we use they are typically stored on the *stack*, but the lifetime is the part that matters. So a node built as an ordinary local variable is a node that evaporates mid-handshake. Anything that must outlive the call, which is to say every node in your tree and every private copy of a key, needs storage whose lifetime extends beyond that call. In this assignment, that means heap storage obtained with `malloc`.

And `malloc` can say no. When memory is unavailable it returns `NULL`, and dereferencing `NULL` is not a tidy exception with a stack trace; it is undefined behavior. Check every allocation. Also remember that allocation failures can happen halfway through a larger operation. If you have already allocated a node and the subsequent key copy fails, the first allocation is still yours to clean up. "The second `malloc` failed" is an explanation, not an exemption. The header in Appendix A tells you exactly what each function must do when a check fails, and the grading harness checks that you did it.

4.3 Strings are just bytes with a promise

Java's `String` was an object that knew its own length and compared itself politely with `.equals`. A C string is a bare array of `char` whose end is marked by a zero byte, written `'\0'`, and every civilized property it appears to have is a promise that you personally keep. `strlen` walks the bytes, counting until it meets the zero; lose the zero and it keeps walking into whatever lies beyond. Comparing two strings with == compares their addresses, so two identical keys stored at different addresses come out "not equal," which is a bug you will write exactly once and remember forever. `strcmp(a, b)` compares contents and returns negative, zero, or positive, and that signed answer is precisely the ordering your tree needs.

Copying a string means allocating `strlen(s) + 1` bytes and copying the characters across. Never forget the `+ 1`; the terminator is a byte too, and the bug you create by dropping it will surface three functions away from where you planted it. Your tree performs this copy on every insert, because the contract in Appendix A says the tree owns its own copy of each key. It has to. The caller's string might be a stack buffer that ceases to exist the instant `rb_insert` returns, and

a tree full of pointers into dead stack frames is a crime scene, not a data structure. This also means that one logical tree node already involves at least two separate lifetimes: the node allocation and the key allocation. They are related, but they are not the same block of memory, and both must eventually be accounted for.

4.4 Structs, and the anatomy of a node

A C `struct` is a class with the interesting parts removed: named fields laid out in order, and nothing else. No methods, no inheritance, no constructors, and certainly no `private`. When you `malloc` one, you get raw bytes, and initializing every field is your job; the compiler will not remind you about the one you forgot, though valgrind might, at an unhelpful distance from the scene. A reasonable node for this assignment looks like this:

```
typedef enum { RED, BLACK } rb_color_t;

struct rb_node {
    char           *key;    /* heap copy; the tree owns it       */
    void           *value;  /* ownership per the header contract */
    struct rb_node *left;
    struct rb_node *right;
    rb_color_t      color;
};
```

That layout is a suggestion, not the contract. One small piece of C syntax is worth noticing before we move on. If `n` has type `struct rb_node *`, then `sizeof *n` means "the size of the thing `n` would point to." It does not dereference `n` at runtime. That is why `malloc(sizeof *n)` is such a common C idiom: the allocation automatically stays matched to the pointer's type. The public header never shows these fields at all: it declares `typedef struct rbtree rbtree_t;` and stops, which makes `rbtree_t` an *opaque type*. Callers can hold a pointer to one, pass it around, and hand it back to your functions, but they cannot see inside, because the compiler itself has never been told what is inside. That is C's version of `private`, enforced not by keyword but by ignorance, and it is why the header can stay frozen while your implementation remains entirely your business. Whether to add a `parent` pointer is your first real design decision; the classic textbook fixup algorithms assume one, and every extra pointer is one more thing to keep correct through every rotation. Figure 9 shows what one node actually is once the abstractions are peeled off: a block of bytes holding, among other things, the addresses of other blocks.

4.5 Ownership is the contract

Before reading the contract, separate one more pair of ideas. A pointer tells you how to reach an object. Ownership tells you who is responsible for eventually releasing that object's resources. You can have several pointers to the same object and still have exactly one owner. In this assignment, that distinction is not philosophy. It is how you avoid double frees.

Read the comments in Appendix A the way a lawyer reads a lease, because that is what they are. The discipline underneath them is simple to state: at every moment, every allocation has

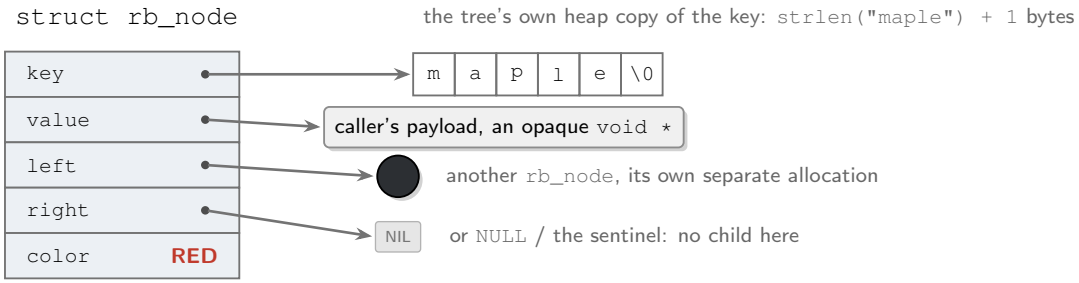


Figure 9: The anatomy of one node, which is to say one circle from Figure 1, as C actually sees it: a block of bytes on the heap whose pointer fields hold the addresses of other blocks. The key points at a second, separate allocation, the tree's private copy of the caller's string. Nothing in this picture frees itself. When this node dies, your code must free the key copy and (if a destructor was supplied) the value, then free the node itself, in an order that never touches a pointer after the memory behind it is gone. A `parent` pointer is a common sixth field; it makes the fixup loops easier to write and is one more pointer to get wrong.

exactly one owner, and the owner is the code responsible for eventually freeing it. Functions move ownership around, and the documentation says exactly when. `rb_insert` is the instructive case. On success, the tree takes ownership of `value`; the caller must not free it, ever, because the tree will, through the `value_free` function supplied at creation. On failure, ownership never moved: the caller still holds `value` and remains responsible for it. And on an overwrite of an existing key, the tree frees the *old* value before installing the new one, because at that moment the tree is the old value's only owner and nobody else will ever see it again. Get any of these wrong in either direction and the result is a leak or a double free, depending on which side of the lease you violated.

For this assignment, it helps to track three things separately: the node, the tree's private key copy, and the opaque `value`. The tree owns every node allocation and every key copy. Ownership of `value` follows the public API. The `left`, `right`, and optional `parent` fields are merely links between nodes; changing those links during a rotation does not create a new owner. A rotation changes topology. It does not change who must eventually free the node, its key, or its value.

The table below is the pocket translation guide. It will not make you a C programmer. But it will keep you from being surprised in the specific ways Java has trained you to be.

In Java	In C
`new Node()`	`malloc(sizeof *n)`, which returns raw, uninitialized bytes; there is no constructor, so you set every field yourself
the garbage collector	`free(p)`: called by you, exactly once, at the right moment; skip it and you have written a leak
`a.equals(b)`	`strcmp(a, b) == 0`; the `==` operator compares addresses, not contents
`node.left`	`n->left`, where `n` is a pointer and the arrow follows it
`Node b = a;`	`struct rb_node *b = a;`: the address is copied, not the node; `a` and `b` now alias the same object
`NullPointerException`	undefined behavior: perhaps a crash, perhaps silent corruption an hour later; the program is not required to tell you
`ArrayIndexOutOf BoundsException`	nothing at all; C cheerfully reads whatever bytes are there, which is a large part of why ASan exists

None of this is hard the way deletion fixup is hard. But it is unforgiving in a way nothing in your Java years was. The habit I want you to build is very small and very concrete: after every operation, be able to answer what was allocated, who owns each allocation now, and what code will eventually free it. If any one of those answers is "I think," keep looking. The tools in Section 8.3 exist precisely because human beings cannot reliably spot these mistakes by reading. Let the machines look. They enjoy it.

5 Assignment Structure

The assignment proceeds through three milestones plus a setup step and one optional, ungraded reach. Do them in order. Each stands on the one before it, and the hour estimates below assume you do. The boundaries are deliberate: each milestone limits how many kinds of failure you are debugging at once. In particular, do not start teaching deletion new tricks while insertion or `rb_validate` is still behaving suspiciously. Debugging a broken delete on top of a broken insert is less a milestone than an archaeological dig.

Milestone	Title	Est. hours
M0	Setup: repository, `CLAUDE.md`, toolchain	1
M1	Skeleton, insert, find, foreach, validate	3–4
M2	Delete (all the fixup cases) + fuzzer, green under asan and memcheck	3–4
M3	Hardening, edge cases, final clean sweep	1–2
The Reach	Teardown without recursion; ungraded, see Section 10	1–2
–	Process artifacts (`CLAUDE.md`, `PROMPTLOG.md`, `REFLECTION.md`), written throughout	1

You may notice what is missing: no custom allocator, no injected allocation failures, no stack budget measured in kilobytes. Those are all waiting in the next assignment, where this tree is the patient and the operating table is already booked. So enjoy plain `malloc` while nobody is deliberately making it fail on schedule. It can still return `NULL`, as Section 4 has just warned you, but this assignment is not yet trying to make that happen for sport. The one discipline that starts now and never relaxes is the memory one: everything you allocate, you free, and the tools verify it.

Milestone	Title	Est. hours
M0	Setup: repository, `CLAUDE.md`, toolchain	1
M1	Skeleton, insert, find, foreach, validate	3–4
M2	Delete (all the fixup cases) + fuzzer, green under asan and memcheck	3–4
M3	Hardening, edge cases, final clean sweep	1–2
The Reach	Teardown without recursion; ungraded, see Section 10	1–2
–	Process artifacts (`CLAUDE.md`, `PROMPTLOG.md`, `REFLECTION.md`), written throughout	1

You may notice what is missing: no custom allocator, no injected allocation failures, no stack budget measured in kilobytes. Those are all waiting in the next assignment, where this tree is the patient and the operating table is already booked. So enjoy `malloc` while it is still unconditionally on your side. The one discipline that starts now and never relaxes is the memory one: everything you allocate, you free, and the tools verify it.

6 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:** Build a red-black tree in C and own every byte of it: correct rebalancing above, airtight allocation and teardown below. A correct tree that leaks, double-frees, or reads freed memory **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 targeted deletion tests and the grading harness punish code nobody reviewed, (b) part 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.

7 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 15 (`CLAUDE.md`, `PROMPTLOG.md`, `REFLECTION.md`), the raw session transcripts, and a git history that shows incremental work (≥ 8 meaningful commits; a single "final submission" commit is an automatic process-grade of zero).
- **Forbidden:** submitting code you cannot explain. The live walkthrough (Section 14) exists to check exactly this. "Claude wrote it" is not an explanation; "this branch frees the old value before installing the new one because the tree is its only remaining owner" is.
- **You own the bugs:** If the agent introduces a use-after-free and you commit it, it is *your* use-after-free.

8 Part-0: Setup (Milestone 0)

8.1 Repository layout

Create (or clone the provided) skeleton:

```
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
|-- tests/
|   |-- test_rbtree.c      # your unit tests
|   `-- fuzz.c             # randomized stress driver
`-- PROMPTLOG.md           # annotated AI-interaction log (see Deliverables)
```

8.2 Install Claude Code

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`.

8.3 Toolchain rules

Everything must compile with `-std=c23 -Wall -Wextra -Werror`. Two verification builds are required and graded, and both lean on tools you may not have met before.

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.

9 Part-1: The Tree (Milestones 1–3)

Implement a classic red-black tree behind the fixed API in `include/rbtree.h` (see Appendix A). Keys are C strings; the tree **copies** keys on insert and **owns** the copies. Values are opaque `void *` pointers; the tree takes ownership of a value on successful insert and releases it (via the destructor supplied to `rb_create`) on delete/destroy/overwrite. If those two sentences read like fine print, go back to Section 4; they are the whole assignment wearing a suit.

Required operations: `rb_create`, `rb_insert` (with rebalancing), `rb_find`, `rb_delete` (with the full deletion-fixup cases; yes, all of them), `rb_size`, `rb_foreach` (in-order), `rb_validate`, `rb_destroy`.

`rb_validate` must check, and your tests must exercise, all invariants (Section 3, Figure 1):

1. The root is black.
2. No red node has a red child.
3. Every root-to-NIL path contains the same number b of black nodes (the tree's *black-height*); so the tree height is $O(\log n)$.
4. In-order traversal yields strictly increasing keys: $k_1 < k_2 < \cdots < k_n$ under `strcmp`.
5. `rb_size` matches the actual node count n.

Testing bar: your fuzzer (`tests/fuzz.c`) performs $\geq 10^5$ random insert/find/delete operations against a reference model (a sorted array or a simple linked list is fine), calling `rb_validate` at least every 100 operations, under both `asan` and `memcheck`.

Targeted deletion tests. The fuzzer is necessary and nowhere near sufficient. Random operations are surprisingly bad at manufacturing the rare fixup configurations on purpose, and an implementation can survive 10^5 random operations while still botching a case it simply never met. So you must also write table-driven tests for `rb_delete` covering, at minimum: a red leaf, a black leaf with a red sibling, a node with two children, and deletion of the root. Each test asserts `rb_validate` and `rb_size` afterward. "Passed the fuzzer" and "handles every case" are different claims, and the grading harness tests the second one.

Memory rules. `rb_destroy` must free every node together with its key copy and, when a destructor was supplied, every value as well; both verification builds must come back with zero findings across your entire suite, empty tree included, because you tested the empty tree. Recursion is permitted this time, in `rb_destroy` and `rb_foreach` alike. Savor that. The next assignment revokes it, and Section 10 lets you find out early what the revocation costs.

A tip you may thank yourself for. Nothing here requires it, but consider routing all allocation in `src/rbtree.c` through two four-line wrappers, `rb_malloc` and `rb_free`, that simply forward to `malloc` and `free`. The next assignment requires exactly that plumbing, so a test harness can make allocation fail on demand and watch what your error paths do about it. Installing the pipes now, while the house is small, is pleasant. Retrofitting them mid-semester is not.

10 The Reach: Teardown Without a Stack

This part carries no points. Do it anyway, or at least read it, because it is a postcard from the next assignment.

Your recursive `rb_destroy` keeps one stack frame alive per level of recursion, and the stack is a fixed-size region of memory that nobody grows for you on demand. On your laptop that region is generous, some megabytes, and a balanced tree of a hundred thousand nodes recurses maybe forty levels deep. No drama. But a kernel thread lives on roughly 8–16 KB of stack, total, shared with everything else it is doing, and code that recurses to a depth it cannot bound is code the kernel community simply does not accept. In the next assignment you will be required to tear down a million-node tree in $O(1)$ auxiliary space: no recursion, no heap-allocated stack, a couple of local variables and nothing more.

The classic trick is lovely. As you free, rotate the tree into a right spine, so there is never a left subtree left to remember; the tree's own pointer fields become the bookkeeping. If you want a head start on next month, implement it now behind a second function and check it against your fuzzer's teardown. Nobody is checking whether you did. Which is, as usual, exactly what makes it worth doing.

11 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. In your project root, run `claude`, then type `/init` to have it generate a starter file from your repo, then **edit it** to contain your project law. 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;
- 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 could a failed allocation inside `rb_insert` leak the key copy?" 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 rb_delete. Read src/rbtree.c and include/rbtree.h first. I want: (1) the transplant and successor strategy you propose, (2) every deletion-fixup case enumerated, with the invariant each one restores, (3) exactly where the key copy and the value get freed, on every path including overwrite, (4) which table-driven tests you'd write first. List risks. Do not write code yet.*

Then **argue with the plan**. Ask "what happens when the node has two children and its successor is its own right child?" 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 table-driven tests for rb_delete first: red leaf, black leaf with red sibling, node with two children, root deletion. Each asserts rb_validate and rb_size afterward. Don't touch rbtree.c yet. Run make test and show me the failures.*

Then: "Now implement only the two-children case. 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. One case 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 Milestone 2 as a hostile kernel reviewer. Hunt specifically for: a use-after-free in the successor splice, a key copy leaked on the overwrite path, allocations whose NULL return is never checked, and any path where rb_ destroy can miss a subtree. 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: delete survives the case tests'." Small commits are your undo button when a later "fix" regresses the suite, 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`.

11.1 Prompt patterns: good vs. useless

Useless	Effective
"solve the assignment"	"Plan rb_delete per CLAUDE.md; enumerate every fixup case and where each path frees memory. No code yet."
"fix the bug"	"The fuzzer fails at operation 4,213: `rb_validate` reports a black-height violation after deleting key 'm'. Reproduce with the seed in `tests/fuzz.c` line 14, shrink it to the smallest failing sequence, 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.

12 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 11, 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.

13 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 your fuzzer debugging. 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 14 and 17 exist specifically to catch. It is the only prompt in this course that manages to be expensive twice.

That is also why Section 11 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 the deletion fixup for a tree whose keys are integers rather than the C strings in the frozen header. You then pay a second time to have it undone. Specific prompts are cheap because "fix the leak on the overwrite path in `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 insertion to deletion, or from a green fuzzer to the write-ups. Stale context is charged on every subsequent message, and the deletion work 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 one thing: arguing through the deletion-fixup cases. It is not required for `rb_size`, which returns a number the tree already knows. Switch with `/model`.
- **Filter tool output before the agent sees it:** This assignment generates spectacular quantities of log. Your fuzzer will print one hundred thousand 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 two 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 bug is in your delete fixup, sixty lines of `rbtree.c`. There is no reason for the agent to read the fuzzer, the Makefile, and four 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 `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`.

14 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, or handling a key-length range you did not originally test. 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 key copy on the overwrite path?", "what happens here if `malloc` returns `NULL`?") and one episode from your `PROMPTLOG.md` and ask you to defend your judgment call. This is the concrete form of the quiz half of the process point described in the grading breakdown.

Why this exists. Every algorithm in here is specified precisely enough to autograde by test battery and sanitizer, 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.

15 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` and your tests; the *unmodified* frozen `include/rbtree.h` (Appendix A).
- `CLAUDE.md`, checked in and evolving across the milestones.

- `PROMPTLOG.md`: 4–6 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` (about a page): where the agent was most and least reliable; one bug it introduced that you caught, and how; and the thing about C that most surprised the Java programmer you were two weeks ago.
- 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 ≥ 8 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.

16 Grading

This assignment is worth **5 points**: 4 points for program correctness, and 1 point split evenly between an analysis of your Claude Code transcripts and the quiz on your own code described in Sections 11 and 14.

Within the 4 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, or reads freed memory 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 fuzzer agrees with a reference model across $\geq 10^5$ operations; the table-driven deletion tests pass; `rb_validate` genuinely exercises every invariant; `rb_destroy` leaves nothing behind. 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
Core tree: insert, find, foreach, validate	Unit tests + fuzzer ($\geq 10^5$ ops) agree with the reference model under asan + memcheck; `rb_validate` exercises all invariants	1.5
Deletion, all fixup cases	Table-driven case tests (red leaf, black leaf with red sibling, two children, root) pass; fuzzer stays green with deletes enabled	1.5
Code quality & memory cleanliness	Compiles clean under `-Wall -Wextra -Werror`; zero ASan/UBSan/valgrind findings; `rb_destroy` leak-free; sane structure	1
Program correctness subtotal		*4*
Prompt analysis	Manual TA review of your raw Claude Code transcripts (Section 17) for genuine, iterative use rather than wholesale generation, corroborated against your repository	0.5
Quiz	Short quiz generated from your own submitted code, administered at the live walkthrough (Sections 14 and 11)	0.5
Total		**5**
The Reach	Teardown without recursion, checked against your own fuzzer: attempted for the craft, not for points	–

Autograder preview. Before you submit, conformance looks like: your fuzzer runs green against the reference model under both sanitizer targets; the table-driven deletion tests pass; and `rb_destroy` leaves a valgrind report with nothing in it, on trees of every size your tests build, empty tree included. Full public test cases will be posted and made available later.

17 How We Grade Your Claude Code Transcripts

Half a point of the 5 comes from your Claude Code transcripts. It is worth being precise about what that half point measures, because it is easy to misread as a participation check that any submitted log passes. It is not. It measures 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 fixup 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.

17.1 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 8.3 and the live walkthrough in Section 14, 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.

17.2 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 0.1: level 5 earns the full half point, and each level down subtracts one tenth.

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.	0.5
4	**Substantive, partially corroborated**	The same quality of reasoning, but one or two episodes cannot be tied back to repository evidence.	0.4
3	**Concrete but shallow**	Real artifacts are cited, but the judgments are acceptance without technical reasoning, of the "looked good, kept it" variety.	0.3
2	**Generic**	Plausible transcripts with no verifiable anchors; the log could describe any student's project as easily as yours.	0.2
1	**Performative or contradicted**	The narrative conflicts with git history, or episodes appear fabricated.	0.1

17.3 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 11 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 this half point exists to catch.

18 Suggested Schedule

- **Week 1:** Milestone 0 + 1: setup, `CLAUDE.md`, the C primer put to work, insert/find/validate, first unit tests green under asan.
- **Week 2:** Milestones 2 + 3: deletion with its case tests, fuzzer to 10^5 ops, memcheck clean, `PROMPTLOG.md`/`REFLECTION.md`, and the Reach (Section 10) if you have time. Walkthroughs in section.

19 Late Submission

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

20 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 tree 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.

21 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.

22 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	8/25/2026	First public release of the assignment.

A `include/rbtree.h` (frozen)

This header is a strict subset of the one you will meet in the next assignment, which is by design: the implementation you write behind it carries forward untouched.

```
#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);

/* 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 traversal. */
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. NULL-safe. */
void rb_destroy(rbtree_t *t);

#endif
```

B Starter `Makefile`

```
CC      := gcc
CFLAGS  := -std=c23 -Wall -Wextra -Werror -g -O1 -Iinclude
SRC     := src/rbtree.c
TSRC    := tests/test_rbtree.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 $@

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

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.
- Check every allocation. malloc can return NULL; a NULL return must
  leave the tree unchanged and return the 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>".
```