A Supplementary Material – Correctness

In this section, we prove Oak's correctness. Since the rebalancer is orthogonal to our contribution, we omit it from the discussion of Oak's correctness. We only assume that RB1-3 hold. We note that a similar rebalance was fully proven in [17].

A.1 Preliminaries

We consider a shared memory system consisting of a collection of shared variables accessed by threads, which also have local variables. An *algorithm* defines the behaviors of threads as deterministic state machines, where state transitions are associated with either an instance of a shared variable primitive (read, write, CAS, etc.) or a local step affecting the thread's local variables. A *configuration* describes the current state of all local and shared variables. An *initial configuration* is one where all variables hold an initial value. A data structure implementation provides a set of operations, each with possible parameters. We say that operations are *invoked* and *return* or *respond*. The invocation of an operation leads to the execution of an algorithm by a thread. Both the invocation and the return are local steps of a thread. A *run* of algorithm $\mathcal{A}$ is an alternating sequence of configurations and steps, beginning with some initial configuration, such that configuration transitions occur according to $\mathcal{A}$. We say that two operations are *concurrent* in a run r if both are invoked in r before either returns. We use the notion of time t during a run r to refer to the configuration reached after the t^{th} step in r. An *interval* of a run r is a sub-sequence that starts with a step and ends with a configuration. The *interval of an operation op* starts with the invocation step of *op* and ends with the configuration following the return from *op* or the end of r, if there is no such return.

An implementation of concurrent data structure is *linearizable* [33] (a correctness condition for concurrent objects) if it provides the illusion that each invoked operation takes effect instantaneously at some point, called the *linearization point* (l.p.), inside its interval. A *linearization* of a run r ($lin(r)$) is the sequential run constructed by serially executing each operation at its l.p.

A.2 Linearizability proof

Definition 1. If there is an entry e in Oak that points to key k and handle h, (i.e., lookUp(k) returns e s.t. h = `handles[entries[e].hi]`) and h`.deleted = false`, we say that h is *associated with* k.

Claim 2. If an Oak operation searches for key k and finds a non-deleted handle h (`h.deleted = false`), then h is associated with k.

Proof. If an operation searches for k and finds h, then there is an entry e that points to k, since Oak ensures that there is at most one entry that points to k, and k is found only if there is such entry. This also means that e points to handle h (by handle index hi). Assume that e does not point to handle h, then the handle index is now $hi' \neq hi$. If $hi' = \perp$ then the handle index can be set only by a non-insertion operation using a CAS. According to Algorithm 3 this is only possible when h in `handles[`hi`]` is already deleted, but h is not deleted. Otherwise, $hi' \neq hi$ and $hi' \neq \perp$, then the handle index can be set only by an insertion operation using a CAS. According to Algorithm 2 this is only possible when h in `handles[`hi`]` is already deleted, which is not the case. Therefore, there is an entry e that points to k and h and `h.deleted = false`, so by Definition 1 h is associated with k. ◻

Claim 3. Assume handle h is associated with key k at time t in a run r. Then, h is associated with k at time $(t+1)$ in r if and only if the $(t+1)^{st}$ step in r is not the l.p. of a successful remove(k) operation.

Proof. Assume that h is not associated with k at time $(t+1)$.

If there is no handle associated with k at time $t+1$, then by Definition 1 either h`.deleted = true` or the entry's handle index (hi) is $\perp$. In the first case, the only possible step that marks a handle as deleted is the l.p. of a successful remove(k). In the second case, only non-insertion operations turn hi to $\perp$ by using CAS (lines 56 and 74), and according to Algorithm 3 this is only possible when the handle is deleted. However, at time t, h is still associated with k. Therefore, the entry's handle index (hi) is not $\perp$.

Otherwise, there is a different handle $h' \neq h$ that is associated with k at time $t+1$ ($h' \neq \perp$). This change can only be done by an insertion operation using CAS (line 38). According to Algorithm 2 an insertion operation reaches that CAS only if the handle (h) is already deleted (line 21). However, at time t, h is still associated with k, and so there is no different handle that is associated with k.

Therefore, as long as the $(t+1)^{st}$ step is not the l.p. of a successful remove(k), then h is still associated with k at time $t+1$ in r, and there is no handle associated with k at time $t+1$ if the $(t+1)^{st}$ step is a l.p. of a successful remove(k), as required. ◻

Claim 4. Assume no handle is associated with key k at time t in a run r. Then, no handle is associated with k at time $t+1$ in r if and only if the $(t+1)^{st}$ step in r is not the l.p. of a successful insertion operation of k.

Proof. If no handle is associated at time t, and at time $t+1$ there is an associated handle, then according to Definition 1 either a handle's deleted flag turned from false to true, or the entry's handle index turned from $\perp$ to a valid one. The former is not possible because the handles are initialized as not deleted and only become deleted by a remove; no operation turns a deleted handle to a non-deleted one. In the second case, this can only be done by a successful insertion operation, at its l.p. (line 38), as required. ◻

Look at the linearization $lin(r)$ of run r using l.p.s defined in Section 4.5. From Claims 3 and 4, by induction on the steps of a run, we get:

Corollary 5. At any point in a concurrent run r, the set of keys associated with handles is exactly the same as the set of inserted keys and not removed keys, associated with the same handles, in $lin(r)$ up to that point.

Claim 6 (Get)**.** In run r, if get(k) returns h then the corresponding get(k) in $lin(r)$ returns h, and if get(k) returns `null` then the corresponding get(k) in $lin(r)$ returns `null`.

Proof. There are three cases for get's l.p.:

1. Get(k) finds a non-deleted handle h (line 6), then get(k) returns h and by Claim 2 h is associated with k. By Corollary 5, in $lin(r)$ k is inserted and not removed (the map holds k) and since this is the l.p. of get then the corresponding get(k) in $lin(r)$ returns h as well.
2. LookUp(k) by get(k) (line 3) returns $\perp$ or if get(k) reads that the handle index is $\perp$ (line 4), then there is no handle associated with key k, and get(k) returns `null`. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of get then the corresponding get(k) in $lin(r)$ returns `null` as well.
3. Get(k) finds a deleted handle h at time t_2 (line 6) and returns `null`. Then its l.p. is the later between the read of handle index hi by get(k) at time $t_1 < t_2$ (line 4) and immediately after the set of `deleted` = `true` by remove(k) at some time $t < t_2$. Again there are two cases:
   a. If $t > t_1$ then the l.p. is immediately after the set of `deleted` = `true` then there is no handle associated with key k, and by Corollary 5, in $lin(r)$ the map does not hold k, and the corresponding get(k) in $lin(r)$ returns `null` as well.
   b. If $t_1 > t$ then the l.p. is the read of handle index hi by get(k) (line 4) at time t_1, after the set of `deleted` = `true` at time t. We need to show that at no time between t and t_1 the handle index changed to $hi' \neq hi$ and now it does not point to a deleted handle. Notice that only an insertion operation l.p. can change hi to hi'. Assume by contradiction that the l.p. of such an operation occurs between t and t_1. Then when get sees hi at time t_1, it is already hi' and not hi. A contradiction. Hence, at the l.p. of get(k), there is no handle associated with key k, and by Corollary 5, in $lin(r)$ the map does not hold k, so the corresponding get(k) in $lin(r)$ returns `null` as required.

□

Claim 7 (PutIfAbsent)**.** In run r, if putIfAbsent(k) returns `true` then the corresponding putIfAbsent(k) in $lin(r)$ returns `true`, and if putIfAbsent(k) returns `false` then in $lin(r)$ the corresponding putIfAbsent(k) returns `false`.

Proof. If putIfAbsent(k) finds a non-deleted handle h (line 21), then putIfAbsent(k) returns `false` and by Claim 2 h is associated with k. By Corollary 5, in $lin(r)$ k is inserted and not removed (the map holds k) and since this is the l.p. of putIfAbsent then the corresponding putIfAbsent(k) in $lin(r)$ returns `false` as well.

Otherwise, if putIfAbsent(k) performs a successful CAS of handle index from $\perp$ (line 38), then putIfAbsent(k) returns `true` and by Definition 1 there was no handle associated with k just before the CAS. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of putIfAbsent then the corresponding putIfAbsent(k) in $lin(r)$ returns `true` as required. □

Claim 8 (ComputeIfPresent)**.** In run r, if computeIfPresent(k) returns `true` then in $lin(r)$ the corresponding computeIfPresent(k) returns `true`, and if computeIfPresent(k) returns `false` then the corresponding computeIfPresent(k) in $lin(r)$ returns `false`.

Proof. If computeIfPresent(k) finds a non-deleted handle h and there is a successful nested call to handle compute (line 50), then computeIfPresent(k) returns `true` and by Claim 2 h is associated with k. By Corollary 5, in $lin(r)$ k is inserted and not removed (the map holds k) and since this is the l.p. of computeIfPresent then the corresponding computeIfPresent(k) in $lin(r)$ returns `true` as well.

If lookUp(k) by computeIfPresent(k) returns $\perp$, or if computeIfPresent(k) reads that the handle index is $\perp$ (line 47), then there is no handle associated with key k, and computeIfPresent(k) returns `false`. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of computeIfPresent then the corresponding computeIfPresent(k) in $lin(r)$ returns `false` as required.

Otherwise, a successful CAS of handle index to $\perp$ is performed by computeIfPresent(k) (line 56), from a handle index pointing to a deleted handle (line 49). Then computeIfPresent(k) returns `false` and by Definition 1 there is no handle associated with k just before the CAS and right after it. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of computeIfPresent then the corresponding computeIfPresent(k) in $lin(r)$ returns `false`. □

Claim 9 (Put)**.** In run r, if put(k) inserts k and returns then in $lin(r)$ the corresponding put(k) inserts k and returns, and if put(k) replaces k's value and returns then in $lin(r)$ the corresponding put(k) replaces k's value and returns.

Proof. If put(k) finds a non-deleted handle h and there is a successful nested call to handle put (line 23), then put(k) replaces k's value and returns, and by Claim 2 h is associated with k. By Corollary 5, in $lin(r)$ the map holds k and since this is the l.p. of put then the corresponding put(k) in $lin(r)$ replaces k's value and returns as well.

Otherwise, put(k) performs a successful CAS of handle index (line 38) from $\perp$, and inserts k and returns. By Definition 1 there is no handle associated with k just before the CAS, and there is one right after the CAS (the handle is initialized as non-deleted). Since this is the l.p. of put, and by Corollary 5 in $lin(r)$ the map does not hold k before the l.p. and does after. Therefore, the corresponding put(k) in $lin(r)$ inserts k and returns as required. □

Claim 10 (PutIfAbsentComputeIfPresent)**.** In run r, if putIfAbsentComputeIfPresent(k) inserts k and returns then in $lin(r)$ the corresponding putIfAbsentComputeIfPresent(k) inserts k and returns, and if putIfAbsentComputeIfPresent(k) updates k's value and returns then in $lin(r)$ the corresponding putIfAbsentComputeIfPresent(k) updates k's value and returns.

Proof. If putIfAbsentComputeIfPresent(k) performs a successful CAS of handle index (line 38) from $\perp$, then it inserts k and returns. By Definition 1 there is no handle associated with k just before the CAS, and there is one right after the CAS (the handle is initialized as non-deleted). Since this is the l.p. of putIfAbsentComputeIfPresent, and by Corollary 5 in $lin(r)$ the map does not hold k before the l.p. and does after. Therefore, the corresponding putIfAbsentComputeIfPresent(k) in $lin(r)$ inserts k and returns as required.

Otherwise, putIfAbsentComputeIfPresent(k) finds a non-deleted handle h and there is a successful nested call to handle compute (line 25), then putIfAbsentComputeIfPresent(k) updates k's value and returns, and by Claim 2 h is associated with k. By Corollary 5, in $lin(r)$ the map holds k and since this is the l.p. of putIfAbsentComputeIfPresent then the corresponding putIfAbsentComputeIfPresent(k) in $lin(r)$ updates k's value and returns as well. □

Claim 11 (Remove)**.** In run r, if remove(k) removes k and returns then in $lin(r)$ the corresponding remove(k) removes k and returns, and if remove(k) returns unsuccessfully (without removing any key) then in $lin(r)$ the corresponding remove(k) returns unsuccessfully.

Proof. If remove(k) finds a non-deleted handle h and a successful nested call to handle remove occurs, setting the handle to deleted (line 52), then remove(k) removes k and returns. By Claim 2 h is associated with k before and there is no handle associated with k right after (by Definition 1). Since this is the l.p. of remove, and by Corollary 5 in $lin(r)$ the map does hold k before the l.p. and does not after. Therefore, the corresponding remove(k) in $lin(r)$ removes k and returns as required.

If lookUp(k) by remove(k) returns $\perp$, or if remove(k) reads that the handle index is $\perp$ (line 47), then there is no handle associated with key k, and remove(k) returns unsuccessfully. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of remove then the corresponding remove(k) in $lin(r)$ returns unsuccessfully as required.

Otherwise, a successful CAS of handle index to $\perp$ is performed by remove(k) (line 56), from a handle index pointing to a deleted handle (line 49). Then remove(k) returns and by Definition 1 there is no handle associated with k just before the CAS and right after it. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of remove then the corresponding remove(k) in $lin(r)$ returns unsuccessfully. □

Having shown that all of OakâĂŹs operations behave the same way in a run r and its linearization $lin(r)$, we can conclude the following theorem:

Theorem 12. Oak is linearizable with the l.p.s defined in Section 4.5.