Problem Set 4 Solutions

Exercise 4-1. Do Exercise 19.3-1 on p. 522 in CLRS.

Exercise 4-2. Do Exercise 19.4-1 on p. 526 in CLRS.

Exercise 4-3. Do Exercise 21.2-3 on p. 568 in CLRS.

Exercise 4-4. Do Exercise 21.3-3 on p. 572 in CLRS.

Exercise 4-5. Do Exercise 23.1-7 on p. 630 in CLRS.

Exercise 4-6. Do Exercise 23.2-4 on p. 637 in CLRS.
Problem 4-1. Order-Maintenance Data Structures

An order-maintenance (OM) data structure maintains a total order of a dynamic set of items and supports the following operations:

- **INSERT** \((x, y)\): Insert element \(y\) immediately after element \(x\) in the total order. The order of all other elements remains the same.
- **PRECEDES** \((x, y)\): Return \(\text{TRUE}\) if \(x\) precedes \(y\) in the total order and \(\text{FALSE}\) otherwise.

For example, after the sequence **INSERT** \((x, y)\), **INSERT** \((x, z)\), **INSERT** \((z, t)\), the total order is \(x < z < t < y\), and hence query **PRECEDES** \((x, t)\) should return \(\text{TRUE}\), while the query **PRECEDES** \((y, z)\) should return \(\text{FALSE}\).

Consider a doubly linked-list implementation of an OM data structure which assigns numeric tags to elements so that the order of tags matches the maintained order of elements. Thus, **PRECEDES** \((x, y)\) returns \(x.\text{tag} < y.\text{tag}\). To perform **INSERT** \((x, y)\) we first insert the element \(y\) after \(x\) within the linked list (in \(O(1)\) time), and then assign a new tag to \(y\) to reflect the new order.

**(a)** Suppose **INSERT** \((x, y)\) assigns the tags as follows:

- if the list was empty, set \(x.\text{tag} = 0\) and \(y.\text{tag} = 2^{k-1}\)
- if \(y\) was inserted between \(x\) and \(z\), set \(y.\text{tag} = (x.\text{tag} + z.\text{tag})/2\)
- if \(y\) was inserted at the tail of the list, set \(y.\text{tag} = (x.\text{tag} + 2^k)/2\)

Show that the tag assigned by **INSERT** is always integer if and only if \(k < n - 1\), where \(n\) is the number of items maintained in the total order. (Note: the initial **INSERT** puts two new items in the list.)

**Solution:** For the forward direction, we prove the contrapositive that \(k < n - 1\) could result in a non-integer tag. Suppose that \(k < n - 1\). Then, we can construct a sequence of inserts **INSERT** \((x, z)\), **INSERT** \((x, y_1)\), \ldots, **INSERT** \((x, y_{n-2})\) that results in the total order \(x < y_{n-2} < \ldots < y_1 < z\) with \(n\) elements. The first insert **INSERT** \((x, z)\) results in the tag assignments \(x.\text{tag} = 0\) and \(z.\text{tag} = 2^{k-1}\). **INSERT** \((x, y_i)\) for \(i \geq 1\) results in the tag assignment \(y_i.\text{tag} = 2^{k-i-1}\). Thus, the final insert **INSERT** \((x, y_{n-2})\) results in \(y_{n-2}.\text{tag} = 2^{k-n+1}\) which is not an integer for \(k < n - 1\).

For the reverse direction, we prove a stronger statement in the lemma below. Refer to \(2^k\) as the sentinel tag. While the sentinel tag is never assigned to an element, it serves as an upper bound on the possible tags that can be assigned.

**Lemma 1** After inserting \(i\) elements, the tag difference \(z.\text{tag} - x.\text{tag}\) between any two consecutive elements \(x < z\) in the total order or the difference between the last element’s tag and the sentinel tag is equal to \(2^j\) for some integer \(j \geq k - i + 1\).

**PROOF.** For the base case \(i = 2\), the tags are \(2^{k-1}\) apart. For the inductive case, suppose we’ve inserted \(i - 1\) elements, and assume the lemma to be true for \(1, \ldots, i - 1\). Any pair of consecutive tags, including the tag of the last element and the sentinel tag,
is $2^j$ apart for some $j \geq k - i + 1$. When we insert the $i$-th element $y$, it will be assigned a midpoint tag, i.e. $(x.tag + z.tag)/2$ or $(x.tag + 2^k)/2$, so that the tags are now $2^j/2 = 2^j$ apart for some $j' = j - 1 \geq k - (i + 1) - 1$.

If $k \geq n - 1$, then $j \geq 0$ as proven above, and it follows that the tags are always integers.

(b) When $n > k + 1$, we update each tag $x.tag' = 2^k x.tag$ and $k' = 2k$. Assume that comparisons and arithmetic on two tags can be computed in $O(1)$ time. Show that this implementation of INSERT runs in $O(1)$ amortized time and uses $O(n)$ bits to express each tag.

Solution: Every event instance of $n > k + 1$ requires updating $k + 1$ tags and doubling $k$. Let $k_0$ refer to the original value of $k$ before it is ever doubled. By an aggregate analysis, inserting $n$ elements requires doing $n$ initial tag assignments and $(k_0 + 1) + (2k_0 + 1) + (4k_0 + 1) + \ldots + (2^\lceil \log (n/k_0) \rceil k_0 + 1) = O(n)$ tag updates, i.e. $O(1)$ amortized time for each INSERT. This argument is similar to the analysis for table doubling shown in lecture. Every event instance of $n > k + 1$ also increases the max tag size by $2^k$, i.e. $k$ more bits are needed to express each tag. The initial number bits needed is $k_0$, so the final number of bits needed is $k_0 + (k_0 + 2k_0 + 4k_0 + 8k_0 + \ldots + 2^\lceil \log (n/k_0) \rceil k_0) = O(nk_0) = O(n)$ bits.

In practice, for $O(1)$ time arithmetic and comparisons, we must ensure that tags fit in a constant number of machine words. Since a machine word is large enough to express any address in memory, if the maximum number of elements is $n$, we can certainly afford $\Theta(\log n)$-bit tags. We must revise our tag assignment strategy.

Suppose that the tags are integers in the range $[0, u)$, where $u \in \mathbb{N}$. When we insert an element $y$ between elements $x$ and $z$, where $z.tag - x.tag \geq 2$, we are free to choose for $y.tag$ any integer value in the open interval $(x.tag, z.tag)$. If $z.tag - x.tag = 1$, however, then there are no integer values for $y.tag$. In this case, we retag a (contiguous) sublist of the linked list by renumbering all the tags in the sublist to make room for the inserted element.

To find a suitable sublist, consider a “virtual” complete binary tree of $\lceil \log u \rceil$ levels and $u$ leaves where each element of the list occupies a leaf determined by its tag (the left-most leaf holds tag 0, and the right-most leaf holds $u - 1$). The leaves corresponding to unassigned tags are unoccupied. Each node of the tree defines a tag range containing all the elements in the list that occupy leaves in the subtree below the node. Hence each leaf has $\lceil \log u \rceil$ such enclosing tag ranges. The density of a tag range is the fraction of its descendant leaves that are occupied.

If we insert $y$ after $x$ and need to retag to make room, we examine enclosing tag ranges of $x$ until we find the smallest enclosing tag range with low-enough density. We find the next tag range by walking up the implicit tree, but we need to traverse the elements in the range to calculate its
density. Let \( T \) be a constant such that \( 1 < T < 2 \). We consider the density of a range of size \( 2^k \) to be low enough for retagging if it is less or equal \( T^{-k} \). Otherwise, we say that the range is in overflow.

(c) Assume \( n \) items in the order. Show that the structure needs only \( O(\lg n) \) bits per tag to ensure that a suitable (not in overflow) range for retagging is always found.

Solution: First, let us grab some intuition on this problem. If \( \lg u \), the number of bits per tag, is too small, then the number of possible tags \( u \) could be so small that no matter how we assign tags to the \( n \) elements, there will not exist any tag range that is not in overflow. In other words, the \( n \) elements will be so densely packed, in the sense of which leaves their tags occupy in the tree, that all tag ranges are inescapably in overflow.

Now for the proof. Since we walk up the virtual tree until a tag range of low-enough density is found, if we can ensure that the tag range of the root node of the tree is never in overflow, then we can ensure that we will always find some tag range that is not in overflow. The density of the root is always \( n/u \) regardless of which leaves are occupied, and we must ensure that it is never in overflow. Thus,

\[
\frac{n}{u} \leq T^{-\lg u} = u^{-\lg T}
\]

This yields

\[
\lg u \geq \lg n/(1 - \lg T) = O(\lg n)
\]

(d) The value of \( u \) must be adjusted to \( n \). When \( n \) gets too large for \( u \) to guarantee the above, we pick a new \( u \) that is sufficient for up to \( 2n \) elements and retag the whole list. Show that this introduces only a constant amortized overhead.

Solution: When \( u \) is too small and the whole list is in overflow, we set \( u \) to be large enough for \( 2n \). I.e., when \( n > u^{1-\lg T} \), set \( u = \lceil (2n)^{1/(1-\lg T)} \rceil \). The next time we need to do it is when \( n' > u'^{1-\lg T} \geq 2n \). We need \( \Omega(n) \) inserts before we perform \( O(n) \) work, hence amortized work is \( O(1) \).

(e) Assume that when we retag a range, we assign the tags uniformly from that range. Show that once we retag a range of size \( 2^k \), it will not be retagged again until we make at least \( \Omega((2/T)^k) \) inserts (for sufficiently large \( k \)).

Solution: If the size of a range is \( 2^k \), then the density of the range before retagging is no more than \( T^{-k} \), because we only retag ranges that are not overflowing. Before we retag this range again, one of the two child subranges (of size \( 2^{k-1} \)) must overflow.
After uniform retagging, there are at most \((2/T)^k/2 + 1\) elements in either child subrange, and to overflow their density must exceed \(T^{-k+1}\). Hence, the minimum number of elements that need to be inserted is at least
\[
(2/T)^{k-1} - (2/T)^k/2 - 1 = (2/T)^k(T - 1)/2 - 1 = \Omega((2/T)^k)
\]
for sufficiently large \(k\).

(f) Consider the actual work performed to retag a range, and conclude that this implementation of \textsc{Insert} takes amortized \(O(\lg n)\) time.

\textbf{Solution:} The actual work of retagging is \(O(1)\) per element in the range. The number of elements stored in a range of size \(2^k\) at the moment it is retagged is at most \((2/T)^k\). It takes \(\Omega((2/T)^k)\) inserts to induce \(O((2/T)^k)\) work on an enclosing range so the amortized cost of \textsc{Insert} per enclosing range is \(O(1)\). Since each insert has \(O(\lg u) = O(\lg n)\) enclosing subranges, the total amortized cost is \(O(\lg n)\).

Consider a two-level OM structure in which each OM structure at the bottom stores \(\Theta(\lg n)\) elements and the top-level OM structure stores \(\Theta(n/\lg n)\) such bottom-level structures. Specifically, each element \(x\) in the bottom-level structure has attribute \(x\.\text{sublist}\) which refers to an element in the top-level structure. We implement the OM operations as follows:

\begin{verbatim}
TWOLEVELPRECEDES(x, y)
1   if x\.sublist == y\.sublist
2      return PRECEDES(x, y)
3   else return PRECEDES(x\.sublist, y\.sublist)

TWOLEVELINSERT(x, y)
1   FASTINSERT(x, y)
2   if size of x\.sublist \geq 1 + 2\lg n
3       split x\.sublist into 2 lists: x\.sublist and newlist of size at least \lg n each
4       SLOWINSERT(x\.sublist, newlist)
\end{verbatim}

Here, \textsc{FastInsert} uses the simple tag assignment strategy analyzed in (a)-(b), while \textsc{SlowInsert} uses the strategy analyzed in (c)-(f).

(g) Show that the amortized time of \textsc{TwoLevelInsert} is \(O(1)\) and that it needs no more than \(O(\lg n)\) bits to express each tag, for \(n\) items in the maintained order.

\textbf{Solution:} From part (b), we know that the amortized cost of \textsc{FastInsert} is \(O(1)\). \textsc{FastInsert} is called the same number of times as \textsc{TwoLevelInsert}, so \textsc{FastInsert} contributes \(nO(1) = O(n)\) to the total cost in running \textsc{TwoLevelInsert} \(n\)
times. From part (f), we know that the amortized cost of SLOWINSERT is \( O(\lg n) \). SLOWINSERT inserts an element into the top-level OM in each call, and so it is called the same number of times as the size of the top-level OM, i.e. \( \Theta(n / \lg n) \). Thus, SLOWINSERT contributes \( O(\lg n) \Theta(n / \lg n) = O(n) \) to the total cost in running TWOLEVELINSERT \( n \) times. The total time for \( n \) calls to TWOLEVELINSERTS is therefore \( O(n) \), and the amortized cost per call is \( O(1) \).

Each bottom-level OM stores \( m = \Theta(\lg n) \) elements and according to part (b), the required bits to express a tag is \( O(m) = O(\lg n) \). Each top-level OM stores \( m = \Theta(n / \lg n) \) elements and according to part (f), the required bits to express a tag is \( O(\lg m) = O(\lg n) \). Thus, in both the bottom-level OM’s and the top-level OM, only \( O(\lg n) \) are needed to express each tag.
Problem 4-2. Race Detection in Multithreaded Programs

Multithreaded programs, though intended to be deterministic, may exhibit nondeterministic behavior due to bugs called *determinacy races*. These bugs are typically difficult to detect through normal debugging techniques, such as breakpoints or print statements, as these bugs are dependent on a specific scheduling and timing.

The execution of a multithreaded program can be viewed as a directed acyclic graph (dag), or *computation dag*, where nodes are either *forks* or *joins* and edges are *strands*. A fork node has a single incoming edge and multiple outgoing edges. A join node has multiple incoming edges and a single outgoing edge. A strand (edge) represents a block of serial execution.

In the fork-join programming model, every fork has a corresponding join that unites the forked strands. In this model, the structure of the computation dag can be represented efficiently by a series-parallel (SP) parse tree. In the parse tree each internal node is either an S-node or a P-node and each leaf is a strand in the dag. If two subtrees are children of the same S-node, then the parse tree indicates that the left subtree executes before the right subtree. If two subtrees are children of the same P-node, then the parse tree indicates that the two subtrees execute logically in parallel. We assume that the SP tree is binary and each internal node has two children. An example SP tree is shown below:

(a) The *least common ancestor* of two nodes $u$ and $v$ is the node that is an ancestor of both $u$ and $v$ that has the greatest depth in the tree.

Show that two leaves $u_i$ and $u_j$ can execute in parallel if and only if their least common ancestor is a P-node.

**Solution:** $\text{LCA}(u_i, u_j)$ is the only node $v$ such that $u_i$ and $u_j$ are in the opposite subtrees of $v$. For the other ancestors $u_i$ and $u_j$ will be in the same subtree, the one containing $v$. If $v$ is a P-node, then the subtrees containing $u_i$ and $u_j$ can execute concurrently. If $v$ is an S-node, then the subtrees will execute in series and there can be no race.

Assume a simplified model where each leaf is a write to a memory location. We shall design a race detection algorithm which will determine whether there is a determinacy race, i.e., any two strands could be performing a conflicting write operation on the same memory location at the same time.
Specifically, in each node $v$, we store $v.type \in \{\text{SERIAL}, \text{PARALLEL}, \text{LEAF}\}$, as well as pointers to children $v.left$ and $v.right$. If $v.type = \text{LEAF}$, the accessed memory address is stored in $v.address$. The race detector executes $\text{FINDRACES}(v)$ on the root of the SP tree and returns $\text{TRUE}$ if there is a race and $\text{FALSE}$ otherwise. Additionally, the race detector can keep for each address $a$ a reference $W[a]$ to a particular leaf node accessing that address.

Consider the following recursive implementation of $\text{FINDRACES}$ which employs a disjoint-set data structure:

\begin{verbatim}
FINDRACES(v)
1 MAKE-SET(v)
2 FIND-SET(v).ancestor = v.type
3 if v.type == LEAF
4    if W[v.address] ≠ NIL and FIND-SET(W[v.address]).ancestor == PARALLEL
5       return TRUE
6    W[v.address] = v
7 else
8    for each child u of v
9        if FINDRACES(u)
10           return TRUE
11          UNION(u, v)
12    FIND-SET(v).ancestor = v.type
13 return FALSE
\end{verbatim}

(b) Argue that at the time of the call $\text{FINDRACES}(v)$ the number of sets in the disjoint-set data structure equals the depth of $v$ in the SP tree.

Solution: Intuitively, $\text{FINDRACES}(v)$ recurses in a depth-first fashion on the SP tree, constructing a set when the execution moves down the tree and merging two sets when it backtracks up the tree so that the number of sets always equals the depth in the recursion. We prove this formally below.

Lemma 2 By the time of completion $\text{FINDRACES}(v)$ always adds exactly one more set to the disjoint-set data structure. This set contains $v$.

Proof. We do induction on the height of node $v$’s subtree. For the base case of height 1, $v$ is a leaf node and $\text{FINDRACES}(v)$ adds a singleton set containing only $v$. For the inductive case, assume the lemma to be true for height $i - 1 \geq 1$. The execution of $\text{FINDRACES}(v)$ on a node $v$ of height $i \geq 2$ starts by constructing a singleton set, denoted as $S$, from line 1. The execution proceeds to the for-loop in lines 8-12. Every call to $\text{FINDRACES}(u)$ on a child of $v$ in line 9 adds exactly one new set by the inductive hypothesis, but this set is immediately merged with $S$ in line 11. Thus, by the time $\text{FINDRACES}(v)$ completes, there will only be one new set added. \qed
**Lemma 3** At the time of the call FINDRACES \((v)\) the number of sets in the disjoint-set data structure equals the depth of \(v\) in the SP tree.

**Proof.** We do induction on the depth of the node. For the base case of the root node with depth 0, there are no sets when FINDRACES is called. For the inductive case, assume the statement to be true for depth \(i - 1 \geq 0\). Let \(u\) be a node with depth \(\geq 1\) and \(v\) be the parent of \(u\). When FINDRACES \((v)\) is called, there are \(i - 1\) sets. With one more set added in line 1, there will be \(i\) sets at the time of the first subroutine call in line 12 to FINDRACES \((u)\) for some child \(u\) of \(v\). FINDRACES \((u)\) returns one new set containing \(u\) according to Lemma 2, but this is merged with the set containing \(v\). Thus at the time of subsequent calls in line 12 to FINDRACES \((u)\) for other children of \(v\), the number of sets is always \(i\). □

**(c)** Prove that FINDRACES returns TRUE if and only if there is a determinacy race for some address.

**Solution:** The following lemma is needed to show that the ancestor attribute is actually storing relevant information about least common ancestors. Intuitively we keep a finger in \(W[a]\) on the last leaf \(u\) we have seen such that \(u.\text{address} = a\).

**Lemma 4** Let \(v\) be a leaf node. During the execution of line 4 of FINDRACES \((v)\), if \(W[v.\text{address}] \neq \text{NIL}\), then \(W[v.\text{address}]\) is a leaf node \(u\) that writes to the same address as \(v\) and FIND-SET \((W[v.\text{address}])\).ancestor is the node type of the least common ancestor of \(u\) and \(v\).

**Proof.** If \(W[v.\text{address}] \neq \text{NIL}\), then for some different leaf node \(u\) that writes to the same address, FINDRACES \((u)\) must have been called prior to FINDRACES \((v)\) in order that \(W[v.\text{address}]\) could have been set in line 6. Let \(lca\) denote the least common ancestor of \(u\) and \(v\). Because of the depth-first order of recursion as seen in part (b), at any time point the set containing \(lca\) also contains all of \(lca\)’s descendants for which FINDRACES has thus far been run to completion. Since \(W[v.\text{address}] \neq \text{NIL}\), FINDRACES \((u)\) was already completed, so \(u\) is in this set. Furthermore, line 13 always runs after merging sets, so the ancestor attribute of this set is the node type of \(lca\). Thus, FINDSET \((W[v.\text{address}])\).ancestor returns the node type of the least common ancestor of \(u\) and \(v\). □

Note that the only way that FINDRACES can return TRUE is if the statement in line 4 is satisfied for some recursive call. Also, if any recursive call returns TRUE, then the entire program returns TRUE.

For the reverse direction, if a determinacy race exists between two nodes \(u\) and \(v\), then from part (a) the node type of their least common ancestor is PARALLEL. Without loss of generality, assume that FINDRACES \((v)\) is called after FINDRACES \((u)\). According to the above lemma, FINDRACES \((v)\) returns TRUE from line 5.
For the forward direction, suppose there doesn’t exist a determinacy race. Then, either every leaf node writes to a different address or the least common ancestor of any two leaf nodes that write to the same address is of node type SERIAL. In the former case, $W[v, address]$ always evaluates to NIL. In the latter case, $\text{FINDSET}(W[v, address]).\text{ancestor}$ always evaluates to SERIAL. Thus, $\text{FINDRACES}$ can never return $\text{TRUE}$, so it must return $\text{FALSE}$.

(d) Analyze the worst-case time of $\text{FINDRACES}$ for an SP tree with $n$ leaves. Assume the disjoint-set data structure uses union by rank and path compression heuristics.

Solution: Recursive $\text{FINDRACES}$ calls $\text{MAKE-SET}$ for each node in the tree. It calls $\text{UNION}$ once for each child node in the current subtree, so also once for each node in the whole tree (except root). $\text{FIND-SET}$ is called as many times as $\text{UNION}$ and in addition twice for each leaf. (Note: in a savvy implementation $\text{UNION}$ and $\text{MAKE-SET}$ would return the newly created set and this would let us save a bunch of calls to $\text{FIND-SET}$.)

In a tree of $n$ leaves, there is at most $2n$ nodes, so we perform up to $6n$ operations on up to $2n$ sets. The given disjoint-set data structure will need in the worst-case $O(6n\alpha(2n)) = O(n\alpha(n))$ time, where $\alpha$ is the inverse of the Ackermann function.

We can improve the running time of the algorithm. Consider two kinds of walks of the SP tree:

- An English walk visits the left subtree of each node before the right. In our example SP tree, the English walk visits: P1, S1, u1, S2, P2, u2, u3, S5, u5, S4, P3, u6, u7, u8
- A Hebrew walk does the same for S-nodes, but it visits the right subtree before the left for P-nodes. In our example SP tree, the Hebrew walk visits: P1, S3, u5, S4, P3, u7, u6, u8, S1, u1, S2, P2, u3, u2, u4

(e) Show that $u_i$ serially precedes $u_j$ if and only if $u_i$ precedes $u_j$ both in the English walk and in the Hebrew walk.

Solution: From part (a), if $u_i$ serially precedes $u_j$, then their least common ancestor $\text{LCA}(u_i, u_j)$ is an S-node, $u_i$ is in its left subtree, and $u_j$ is in its right subtree. Because English and Hebrew walks visit the left subtree and then the right subtree of an S-node, $u_i$ precedes $u_j$ both in the English and Hebrew walks.

On the other hand if $u_i$ does not serially precede $u_j$, i.e. they can execute in parallel, then $\text{LCA}(u_i, u_j)$ is a P-node (again from part (a)). Because English and Hebrew walks have opposite visit rules for a P-node, the order of $u_i$ and $u_j$ are reversed in the English and Hebrew walks.
(f) Design a race detector that uses an order-maintenance data structure (see Problem 4-1) and needs worst-case \(O(n)\) time for an SP tree with \(n\) leaves.

**Solution:** At the high level, we will use two OM data structures to maintain the order of English and Hebrew walks separately. We walk the SP tree depth-first search and insert nodes in the OM structures in pre-order fashion. However, the order of subtrees in the Hebrew walk is inverted if the current node type is PARALLEL.

(Side note: many students tried to finish the walk on the tree and then check the order for all pairs of candidate leaves. This is not a good solution as it would take \(O(n^2)\) to detect all races.)

As we walk the tree, we keep a finger in \(W[a]\) on the last leaf \(u\) we have seen such that \(u.address = a\). When we encounter a leaf \(u\), we first check if this finger is set for \(u.address\), i.e., if we have seen another leaf \(v\) such that \(v.address = u.address\). Then we query both OM structures to determine if \(u\) serially precedes \(v\) or vice versa. If the order is different in both structures, then \(u\) and \(v\) could execute in parallel and we have a race. Below is the pseudo-code for the algorithm, in which we use suffixes -E and -H to distinguish the two OM structures.

```plaintext
 FINDRACES(v)
  1  if v.type == LEAF
  2     if W[v.address] ≠ NIL
  3         if PRECEDES-E(W[v.address], v) ≠ PRECEDES-H(W[v.address], v)
  4         return TRUE
  5   W[v.address] = v
  6   else
  7     INSERT-E(v, v.left)
  8     INSERT-E(v.left, v.right)
  9     if v.type == SERIAL
 10        INSERT-H(v, v.left)
 11        INSERT-H(v.left, v.right)
 12   else
 13     INSERT-H(v, v.right)
 14     INSERT-H(v.right, v.left)
 15     if FINDRACES(v.left)
 16         return TRUE
 17     if FINDRACES(v.right)
 18         return TRUE
 19   return FALSE
```

We have shown above that if FINDRACES returns TRUE then there is a race. Now we need to show that if there is a race in the SP tree, then this algorithm will detect it. Let \(u\) and \(v\) be two nodes that can execute in parallel and write to the same address.
Without loss of generality, assume that \( \text{FINDRACES}(v) \) is called after \( \text{FINDRACES}(u) \). Thus, line 2 in the execution of \( \text{FINDRACES}(v) \) will be satisfied. Note that at the time of the call to \( \text{FINDRACES} \) for any node, that node will have already been inserted into both OM’s during the execution of \( \text{FINDRACES} \) for the node’s parent. Thus, line 4 in the execution of \( \text{FINDRACES}(v) \) can be evaluated because both \( u \) and \( v \) will be in both OM’s. In particular the IF statement is satisfied, and \( \text{FINDRACES} \) returns TRUE.