For the containment property, let us consider two processes $p_i$ and $p_j$ that stop at stairs $k_i$, and $k_j$, respectively. Without loss of generality, let $k_i \leq k_j$. Due to Lemma 20, there are exactly $k_i$ processes on the stairs 1 to $k_i$, and $k_j$ processes on stairs 1 to $k_j \leq k_i$. As no process backtracks on the stairway (a process proceeds downwards or stops), the set of $k_j$ processes returned by $p_j$ includes the set of $k_i$ processes returned by $p_i$.

Let us finally consider the immediacy property. Let us first observe that a process deposits its value before starting its descent of the stairway (line 1), from which it follows that, if $j \in \text{set}_i$, $\text{REG}[j]$ contains the value $v_j$ deposited by $p_j$. Moreover, it follows from lines 4 and 5 that, if a a process $p_j$ stops at a stair $k_j$ and then $i \in \text{set}_j$, then $p_i$ stopped at a stair $k_i \leq k_j$. It then follows from Lemma 20 that the set $\text{set}_j$ returned by $p_j$ includes the set $\text{set}_i$ returned by $p_i$, from which follows the immediacy property.

8.5.4 A Recursive Implementation of a One-Shot Immediate Snapshot Object

This section describes a recursive implementation of a one-shot immediate snapshot object due to E. Gafni and S. Rajsbaum (2010). This construction can be seen as a recursive formulation of the previous iterative algorithm.

Underlying data structure As we are about to see, when considering distributed computing, an important point that distinguishes distributed recursion from sequential recursion on data structures lies in the fact that the recursion parameter is usually the number $n$ of processes involved in the computation. The recursion parameter is used by a process to compute a view of the concurrency degree among the participating processes.

The underlying data structure representing the immediate snapshot object consists of a shared array $\text{REG}[1..n]$ such that each $\text{REG}[x]$ is an array of $n$ SWMR atomic registers. The aim of $\text{REG}[x]$, which is initialized to $[\bot, \ldots, \bot]$, is to contain the view obtained by the processes that see exactly $x$ other processes in the system. For any $x$, $\text{REG}[x]$ is accessed only by the processes that attain recursion level $x$ and the atomic register $\text{REG}[x][i]$ can be read by all these processes but can be written only by $p_i$.

The recursive algorithm implementing the operation $\text{update\_snapshot}()$ The algorithm is described in Fig. 8.16. Its main call is an invocation of $\text{rec\_update\_snapshot}(n, v_i)$, where $n$ is the initial value of the recursion parameter and $v_i$ the value that $p_i$ wants to deposit into the immediate snapshot object (line 1). This call is said to occur at recursion level $n$. More generally, an invocation $\text{rec\_update\_snapshot}(x, \_)$ is said to occur at recursion level $x$. Hence, the recursion levels are decreasing from level $n$ to level $n - 1$, then to level $n - 2$, etc. (Actually, a recursion level corresponds to what was called a “level” in Sect. 8.5.3.)
8.5 Immediate Snapshot Objects

When it invokes \texttt{rec_update_snapshot}(x, v), \( p_i \) first writes \( v \) into \( \text{REG}[x][i] \) and reads asynchronously the content of \( \text{REG}[x][1..n] \) (lines 3–4, let us notice that these lines implement a store-collect). Hence, the array \( \text{REG}[x][1..n] \) is devoted to the \( x \)th recursion level.

Then, \( p_i \) computes the view \( \text{view}_i \) obtained from \( \text{REG}[x][1..n] \) (line 5). Let us remark that, as the recursion levels are decreasing and there are at most \( n \) participating processes, the set \( \text{view}_i \) contains the values deposited by \( n' = |\text{view}_i| \) processes, where \( n' \) is the number of processes that, from \( p_i \)'s point of view, have attained recursion level \( x \).

If \( p_i \) sees that exactly \( x \) processes have attained the recursion level \( x \) (i.e., \( n' = x \)), it returns \( \text{view}_i \) as the result of its invocation of the immediate snapshot object (lines 6 and 9). Otherwise, fewer than \( x \) processes have attained recursion level \( x \) and consequently \( p_i \) invokes recursively \texttt{rec_update_snapshot}(x − 1, v) (line 7) in order to attain a recursion level \( x' < x \) accessed by exactly \( x' \) processes. It will stop its recursive invocations when it attains such a recursion level (in the worst case, \( x' = 1 \)).

\textbf{Theorem 38} The algorithm described in Fig. 8.16 is a wait-free construction of an immediate snapshot object. Its step complexity (number of shared memory accesses) is \( O(n(n - |\text{res}| + 1)) \), where \( \text{res} \) is the set returned by \texttt{update_snapshot}(v).

\textbf{Proof} Claim C. If at most \( x \) processes invoke \texttt{rec_update_snapshot}(x, −) then (a) at most \( (x - 1) \) processes invoke \texttt{rec_update_snapshot}(x − 1, −) and (b) at least one process stops at line 6 of its invocation \texttt{rec_update_snapshot}(x, −).

Proof of claim C. Assuming that at most \( x \) processes invoke \texttt{update_snapshot}(x, −), let \( p_k \) be the last process that writes into \( \text{REG}[x][1..n] \). We necessarily have \( |\text{view}_k| \leq x \). If \( p_k \) finds \( |\text{view}_k| = x \), it stops at line 6. Otherwise, we have \( |\text{view}_k| < x \) and \( p_k \) invokes \texttt{rec_update_snapshot}(x − 1, −) at line 7. But in that
case, as $p_k$ is the last process that wrote into the array $\text{REG}[x][1..n]$, it follows from $|\text{view}_j| < x$ that fewer than $x$ processes have written into $\text{REG}[x][1..n]$, and consequently, at most $(x - 1)$ processes invoke $\text{rec_update_snapshot}(x - 1, -)$. End of the proof of claim C.

To prove the termination property, let us consider a correct process $p_i$ that invokes $\text{update_snapshot}(v_i)$. Hence, it invokes $\text{rec_update_snapshot}(n, -)$. It follows from Claim C and the fact that at most $n$ processes invoke $\text{rec_update_snapshot}(n, -)$ that either $p_i$ stops at that invocation or belongs to the set of at most $n - 1$ processes that invoke $\text{rec_update_snapshot}(n - 1, -)$. It then follows by induction from the claim that if $p_i$ has not stopped during a previous invocation, it is the only process that invokes $\text{rec_update_snapshot}(1)$. It then follows from the text of the algorithm that it stops at that invocation.

The proof of the self-inclusion property is trivial. Before stopping at recursion level $x$ (line 6), a process $p_i$ has written $v_i$ into $\text{REG}[x][i]$ (line 3), and consequently we have then $(i, v_i) \in \text{view}_i$, which concludes the proof of the self-inclusion property.

To prove the self-containment and immediacy properties, let us first consider the case of two processes that return at the same recursion level $x$. If a process $p_i$ returns at line 6 of recursion level $x$, let $\text{view}_i[x]$ denote the corresponding value of $\text{view}_i$. Among the processes that stop at recursion level $x$, let $p_i$ be the last process which writes into $\text{REG}[x][1..n]$. As $p_i$ stops, this means that $\text{REG}[x][1..n]$ has exactly $x$ entries different from $\bot$ and (due to Claim C) no more of its entries will be set to a non-$\bot$ value. It follows that, as any other process $p_j$ that stops at recursion level $x$ reads $x$ non-$\bot$ entries from $\text{REG}[x][1..n]$, we have $\text{view}_i[x] = \text{view}_j[x]$ which proves the properties.

Let us now consider the case of two processes $p_i$ and $p_j$ that return at line 6 of recursion level $x$ and $y$, respectively, with $x > y$; i.e., $p_i$ returns $\text{view}_i[x]$ while $p_j$ returns $\text{view}_j[y]$. The self-containment follows then from $x > y$ and the fact that $p_j$ has written into all the arrays $\text{REG}[z][1..n]$ with $n \geq z \geq y$, from which we conclude that $\text{view}_j[y] \subseteq \text{view}_i[x]$. Moreover, as $x > y$, $p_i$ has not written into $\text{REG}[y][1..n]$ while $p_j$ has written into $\text{REG}[x][1..n]$, and consequently $(j, v_j) \in \text{view}_i[x]$ while $(i, v_i) \notin \text{view}_j[y]$, from which the containment and immediacy properties follow.

As far as the number of shared memory accesses is concerned we have the following. Let $\text{res}$ be the set returned by an invocation of $\text{rec_update_snapshot}(n, -)$. Each recursive invocation costs $n + 1$ shared memory accesses (lines 3–4). Moreover, the sequence of invocations, namely $\text{rec_update_snapshot}(n, -)$, $\text{rec_update_snapshot}(n - 1, -)$, etc., until $\text{rec_update_snapshot}(|\text{res}|, -)$ (where $x = |\text{res}|$ is the recursion level at which the recursion stops) contains $n - |\text{res}| + 1$ invocations. It follows that the cost is $O(n(n - |\text{res}| + 1))$ shared memory accesses. \qed