An Efficient Adaptive Partial Snapshot Implementation

Bashari, Benyamin

http://hdl.handle.net/1880/114065

Downloaded from PRISM Repository, University of Calgary
Abstract

In an asynchronous shared-memory system with $n$ processes, the single-writer snapshot type provides consistent views of an array $A$ of $n$ components, each of which can be updated by one of the processes. Formally, a process $p$ can call $\text{Update}(v)$ to write $v$ into $A[p]$, and call $\text{Scan}()$ to obtain a view of $A$. Inherently, under realistic assumptions on the base objects' size, the specification of this type leads to a lower bound of $\Omega(n)$ steps for method $\text{Scan}()$. In some cases, it suffices for a process to take a snapshot of $k$ ($1 \leq k \leq n$) components of $A$, and thus, taking $\Omega(n)$ steps to obtain that snapshot is inefficient when $k$ is small. In this thesis, we provide an implementation of the single-writer adaptive partial snapshot type, which allows a process to take a partial snapshot of $k$ components in $O(k \log n)$ steps. We define a new version of $\text{Scan}()$ that does not return anything and its behavior is defined in terms of another operation $\text{Observe}()$. An $\text{Observe}(i)$ operation by process $p$ returns the value that $A[i]$ had at the point in time of $p$’s preceding $\text{Scan}()$. We implement a single-writer adaptive partial snapshot object from read-write registers, fetch-and-increment objects, and compare-and-swap objects, such that the step complexity of method $\text{Scan}()$ is $O(1)$ and that of methods $\text{Update}()$ and $\text{Observe}()$ is $O(\log n)$. 
Acknowledgements

First and foremost, I thank my wonderful supervisor, Philipp Woelfel, for his help, guidance, and encouragement over the past couple of years. He has a true passion for research, and I am glad that I was able to work under his supervision. I also like to thank the distributed algorithm group of the University of Calgary for the fun activities and the enjoyable group lunches we had. Furthermore, I would like to thank Trevor Brown and Wojciech Golab for their insightful discussions. Last but not least, I like to thank my parents, Tahereh and Farshid, my brother, Danial, and my friends, Fatemeh, Bijan, Kiarash, Ehsan, Ali, Konstantinos, Mehrdad, Mohammad, Morteza, and Navid, who supported me throughout these two years.
# Contents

Abstract ................................................................. ii

Acknowledgements ....................................................... iii

Contents ................................................................. iv

List of Figures and Tables ............................................ v

1 Introduction ........................................................... 1
   1.1 Asynchronous Shared-Memory System .......................... 3
   1.2 Problem Statement and Results ................................... 7
   1.3 Building Blocks ................................................... 9
   1.4 Outline .......................................................... 10

2 Literature Review .................................................... 11

3 Single-Writer Predecessor Object .................................. 14
   3.1 The Basic Algorithm .............................................. 14
   3.2 The Memory Reclamation Algorithm ............................ 16
   3.3 Correctness Proof ................................................. 24
   3.4 Analysis ........................................................ 36

4 Single-Writer Adaptive Partial Snapshot Algorithm ................. 39
   4.1 The Basic Algorithm .............................................. 39
   4.2 The Memory Reclamation Algorithm ............................ 44
   4.3 Correctness Proof ................................................. 49
   4.4 Analysis ........................................................ 59

5 Conclusion ................................................................ 64
List of Tables and Figures

Figure 1  A sample history on a stack. . . . . . . . . . . . . . . . . . 5
Table 2   Comparison of partial snapshot objects  . . . . . . . . . . . 13
Figure 3  A BST that applies the node copying technique. . . . . . . 15
Figure 4  A non-linearizable execution of Algorithm 3. . . . . . . 41
Chapter 1

Introduction

In asynchronous shared-memory systems, processes communicate with one another by performing operations on shared objects. A well-studied object in such systems is of type snapshot. It provides consistent views of the shared memory. Formally, the snapshot type maintains an array $A[0 \ldots m - 1]$ and provides two operations $\text{Update}(i, v)$, which changes the value of $A[i]$ to $v$, and $\text{Scan}()$, which returns a view of $A$ (i.e., the vector $A[0], \ldots, A[m - 1]$). The single-writer snapshot type is a restricted version, in which the number of components of $A$ is equal to the number of processes, $n$, and process $p$ can only update $A[p]$.

Aspnes and Herlihy [9], and Anderson [4] showed that it is possible to implement a single-writer snapshot object from read-write registers. Much research followed [9] and obtained efficient implementations of snapshots and single-writer snapshots. Also, to obtain theoretical results about the power of read-write registers, some work (e.g., [7, 8, 12]) assumes that a base object (i.e., an object that is provided by the shared-memory system, such as a read-write register) can store an entire view of $A$. Under realistic assumptions on the base objects’ size, storing a view of $A$ needs $m$ base objects. Thus, it is not hard to see that method $\text{Scan}()$ has an inherent lower bound of $\Omega(m)$ atomic operations, when using base objects with realistic sizes. This is inefficient for the applications in which a process only requires reading a consistent view of $k$ ($1 \leq k \leq m$) components. As an example, in a stock portfolio management system, when we want to compute the total assets of a portfolio consisting of $k$ stocks, we only need to take a snapshot of the price of those $k$ stocks, not all of them. To solve such problems, Attiya, Guerraoui, and Ruppert [10] introduced the partial snapshot type, in which a $\text{Scan}()$ operation takes $k$ ($1 \leq k \leq m$)
arguments $i_1, \ldots, i_k$ and returns a consistent view of $A[i_1], \ldots, A[i_k]$. The value of $k$ can vary with each $\text{Scan}()$ operation, and thus a process can take partial snapshots of array $A$. Wei, Ben-David, Blelloch, Fatourou, Ruppert, and Sun [27], extended this definition to obtain the constant-time snapshot type: A $\text{Scan}()$ operation returns a handle $h$. A process can call $\text{ReadSnapshot}(h, i)$ to read the value that $A[i]$ had at the point in time of that $\text{Scan}()$ operation. Motivated by [27], we defined the adaptive partial snapshot type. A $\text{Scan}()$ operation does not return anything and its behavior is defined in terms of another operation $\text{Observe}()$. A process is allowed to call an $\text{Observe}()$ operation after it has performed at least one $\text{Scan}()$ call. When process $p$ calls an $\text{Observe}(i)$ operation, it returns the value that $A[i]$ had at the point in time of $p$’s preceding $\text{Scan}()$ operation. Following a $\text{Scan}()$, a process can adaptively decide which components of the snapshot object to observe, based on the results of previous $\text{Observe}()$ operations. We therefore call this type adaptive partial snapshot. We implement a single-writer adaptive partial snapshot object, where $\text{Scan}()$ has step complexity $O(1)$ and $\text{Update}()$ and $\text{Observe}()$ have step complexity $O(\log n)$. The advantage over [10] and [27] is that all methods in our algorithm have bounded step complexity.

We use read-write registers, compare-and-swap, and fetch-and-increment as base objects, all of which are provided by common hardware. We assume that the base objects’ size, $W$, is at least max{$3 \log n, \log |D_{snp}|$} + $O(1)$ bits, where $D_{snp}$ is the domain of values that can be stored in a component of a snapshot object. This is a practical assumption because we need at least $\log n$ bits to store a process ID and at least $\log |D_{snp}|$ bits to store the value of $A[i]$ ($0 \leq i < n$). Also, we store a sequence number that is incremented at most 5 times with each $\text{Update}(), \text{Scan}()$, and $\text{Observe}()$. Hence, only up to $2^W/5 - 1$ operations can be executed in our object. This is not a restriction that has any practical impact on current 64-, or even 32-bit architectures.

As discussed earlier, under realistic assumptions on the base objects’ size, the most efficient single-writer snapshot object has step complexity $O(1)$ for $\text{Update}()$ and $O(n)$ for $\text{Scan}()$. Our object can simulate a single-writer snapshot object with $O(\log n)$ overhead over the most efficient implementation but is more efficient when a snapshot of $o(n/ \log n)$ components is required. Moreover, using our object, a process can take a snapshot and then adaptively decide what components of that snapshot to read next, based on the values it read earlier. For example, if a snapshot represents a data structure, the search
path through the data structure can depend on the actual values found. Also, in an error recovery system, a process takes many snapshots throughout the execution. When there is an error in the system, that process usually needs to roll back to the most recent backup. Since the step complexity of \texttt{Scan()} is $O(1)$, taking snapshots of an error recovery system can be done efficiently.

### 1.1 Asynchronous Shared-Memory System

In this section, we provide the definitions used throughout this thesis. These definitions are based on the work of Herlihy and Wing [19]. In some cases, we deviate from the original definitions to facilitate our proofs.

We consider the standard asynchronous shared-memory system with $n$ processes, each of which has a unique ID in \{0, $\cdots$, $n - 1$\}. Each process’s ID is stored in a local variable $myID$. Processes communicate by performing operations on shared objects. Each object is an instance of a type, and a type defines some methods (a.k.a. operations), and has a sequential or a concurrent specification that dictates the behaviour of an object of that type in sequential or concurrent executions, respectively.

In concurrent systems, the objects can be divided into two groups: base objects (a.k.a. atomic objects) and implemented objects. Base objects are provided by the system (e.g., read-write registers), and implemented objects are constructed from base objects. Each operation on a base object is called a step and occurs atomically. On the other hand, each operation on an implemented object may consist of multiple steps.

We assume that the base objects, provided by the system, have size $W$ bits. A word means $W$ bits, and thus $k$-word means $kW$ bits.

**Execution and History** An execution is a sequence of invocation and response events of operations. Each event $e$ comprises a process $e$.proc, an object $e.obj$, a method $e.met$ (including its arguments), and a point in time $e.time$. If $e$ is a response event of an operation, then it also includes the return value (if any) of that operation in $e.ret$. The invocation and response events of an operation $op$ are denoted by $invE(op)$ and $rspE(op)$, respectively. In an execution $\mathcal{E}$, for each operation $op$, if $rspE(op)$ is in $\mathcal{E}$, then:

(a) $invE(op)$ appears before $rspE(op)$ in $\mathcal{E}$,
(b) $invE(op).proc = rspE(op).proc,$
(c) \(\text{invE}(op).\text{obj} = \text{rspE}(op).\text{obj}\), and
(d) \(\text{invE}(op).\text{met} = \text{rspE}(op).\text{met}\).

Let \(E\) be an execution. Let \(e_1\) and \(e_2\) be two events in \(E\), such that \(e_1\) is followed immediately by \(e_2\). Then \(e_1.\text{time} \leq e_2.\text{time}\). If \(e_1\) is the invocation of an operation \(op\) and \(e_2 = \text{rspE}(op)\), then \(e_1.\text{time}\) can be equal to \(e_2.\text{time}\). Otherwise, \(e_1.\text{time} < e_2.\text{time}\). Suppose \(p\) is a process and \(O\) is an object. We define \(E|p\) and \(E|O\) as the subsequence of all events \(e\) in \(E\), such that \(e.\text{proc} = p\) and \(e.\text{obj} = O\), respectively. We say \(E|p\) is \textit{well-formed} if for each operation \(op\), where \(\text{invE}(op)\) is an event in \(E|p\), one of the following holds.

(a) \(\text{invE}(op)\) is immediately followed by \(\text{rspE}(op)\),
(b) \(\text{invE}(op)\) is followed by the invocation of another operation \(op'\), and if \(\text{rspE}(op) \in E|p\), then \(\text{rspE}(op')\) appears before \(\text{rspE}(op)\),
(c) \(\text{invE}(op)\) is the last event in \(E|p\).

Execution \(E\) is \textit{well-formed} if for every process \(p\), \(E|p\) is well-formed. For the rest of this thesis, we only consider well-formed executions.

Let \(E\) be an execution and \(op\) an operation, where \(\text{invE}(op) \in E\). We say \(op\) is \textit{pending} if \(\text{rspE}(op) \not\in E\). We define \(\text{inv}(op) = \text{invE}(op).\text{time}\). If \(op\) is not pending, we define \(\text{rsp}(op) = \text{rspE}(op).\text{time}\), otherwise we let \(\text{rsp}(op) = \infty\).

Execution \(E\) is sequential if for any two operations \(op_1\) and \(op_2\) performed by different processes, intervals \([\text{inv}(op_1),\text{rsp}(op_1)]\) and \([\text{inv}(op_2),\text{rsp}(op_2)]\) do not intersect. An execution is \textit{complete} if it does not have any pending operation.

We can construct a \textit{completion} of an incomplete execution \(E\), by doing one of the following for each pending operation \(op\):

(a) Removing \(\text{invE}(op)\) from \(E\), or
(b) appending a response of \(op\) to \(E\).

For any two operations \(op_1\) and \(op_2\) in \(E\), we say \(op_1\) \textit{happens before} \(op_2\) if \(\text{rsp}(op_1) < \text{inv}(op_2)\).

A \textit{history} is an execution \(E\), such that for any two distinct operations \(op_1\) and \(op_2\) performed by the same process, either \(op_1\) happens before \(op_2\) or \(op_2\) happens before \(op_1\). In a complete sequential history, for every operation \(op\), \(\text{invE}(op)\) is followed immediately by \(\text{rspE}(op)\). Hence, we can replace the pair of consecutive events \(\text{invE}(op), \text{rspE}(op)\) by \(op\) to obtain a canonical sequence of operations. Figure 1 shows a sample history on a stack object \(O\). First,
The specification of a type is a set of valid histories. A history $H$ on objects $O_1, \ldots, O_k$ is valid if for every $i$ ($1 \leq i \leq k$), $H | O_i$ is valid.

**Correctness Condition: Linearizability** To define the behavior of an object that has a sequential specification (e.g., a stack) in concurrent histories, we need a correctness conditions. The gold standard of correctness conditions is linearizability. A complete history $H$ is linearizable if there exists a sequential history $S$ containing exactly the same operations as $H$, such that:

(a) $S$ is valid, and

(b) $S$ preserves the happens before order. (I.e., for any two operations $op_1$ and $op_2$ in $H$, such that $op_1$ happens before $op_2$, $op_1$ appears before $op_2$ in $S$)

An incomplete history $H$ is linearizable if there exists is a completion of $H$ that is linearizable. An implemented object $O$ is linearizable if every (finite and infinite) history obtained from executions of $O$ is linearizable. If $O$ has a deterministic sequential specification (i.e., in each sequential history $H$ of $O$, the return value of each operation $op$ is uniquely determined by the operations
that appear before \( op \) in \( H \)), then to prove the linearizability of \( O \), it suffices to show that any finite execution of \( O \) is linearizable [17]. In this thesis, we only consider deterministic sequential specifications.

A popular method to prove linearizability of a history is using linearization points. Let \( H \) be a complete history. Let \( \text{lin} \) be a mapping that maps each operation \( op \) in \( H \) to a point in time, such that \( \text{inv}(op) \leq \text{lin}(op) \leq \text{rsp}(op) \). Suppose \( S \) is a sequential history containing all the operations in \( H \), ordered by mapping \( \text{lin} \). Then \( S \) preserves happen before order. Therefore, if \( S \) is also valid, then \( H \) is linearizable. So to prove the linearizability of \( H \), one can define \( \text{lin} \) and afterward, prove that \( S \) is valid. We call \( \text{lin}(op) \) the linearization point of \( op \).

Linearizability is compositional. That is, if a history \( H \) is linearizable on each of its objects, then \( H \) is also linearizable. This allows us to use a modular approach when implementing concurrent algorithms. Also, a linearizable object can be considered as an atomic object in deterministic algorithms. Therefore, when implementing a deterministic concurrent algorithm, we can use atomic and linearizable objects interchangeably.

**Atomic Base Objects** In this thesis, we assume that the system provides three atomic base objects: compare-and-swap (CAS) objects, fetch-and-increment (FAI) objects, and read-write registers. These base objects are commonly supported by today’s hardware architectures.

A read-write register stores a value and supports two operations, \( \text{Write}(v) \), which changes its value to \( v \) and returns nothing, and \( \text{Read()} \), which returns the value of the register. A FAI object stores an integer, initially 0, and provides an operation \( \text{FAI()} \), which increments the object’s value by 1 and returns the value before the increment. A CAS object stores a value and provides an operation \( \text{CAS}(old, new) \). If the value of the object is \( old \), this operation updates the value to \( new \) and returns true, otherwise, the object remains unchanged and the operation returns false.

**Progress Condition: Wait-Freedom** In concurrent executions, some operations of an implemented object may not terminate. Hence, there are various progress conditions that characterize the progress of the system. Wait-freedom is the one that we focus on in this thesis. An operation is wait-free if every time a process invokes that operation and continues taking steps, it receives a
response after taking a finite number of steps. An object is wait-free if all of its operations are wait-free.

**Step Complexity** The *step complexity* of a wait-free operation is the maximum number of shared-memory steps a process needs to take to finish that operation. The *time complexity* of a wait-free operation accounts for the local computation time (i.e., the computation in which the process does not access any shared objects) in addition to the shared memory steps.

### 1.2 Problem Statement and Results

In this thesis, we present a single-writer adaptive partial snapshot object and a single-writer predecessor object. We use the latter as a building block in the implementation of the former. However, the single-writer predecessor object might be of independent interest as well. The following are the sequential specifications of these types along with our main results.

The single-writer adaptive partial snapshot type maintains an $n$-component array and supports three operations \texttt{Update}(v), \texttt{Scan}(), and \texttt{Observe}(k). An \texttt{Update}(v) operation called by process $p$ changes the value of the $p$-th component to $v$, and returns nothing. Method \texttt{Scan}() does not return anything, and its behavior is only defined in terms of method \texttt{Observe}(k). A process is only allowed to call an \texttt{Observe}(k) operation after it has performed at least one \texttt{Scan}() call, and an \texttt{Observe}(k) call by process $p$ returns the value that the $k$-th component of the object had at the point of $p$'s latest preceding \texttt{Scan}() operation.

Let $D_{\text{sn}}$ be the domain of values that can be stored in a component of a single-writer adaptive partial snapshot object.

**Theorem 1 (A Single-Writer Adaptive Partial Snapshot Algorithm).** Let $W$ be an integer greater than or equal to $\max\{\log |D_{\text{sn}}|, 3\log n\} + O(1)$. There exists a wait-free linearizable single-writer adaptive partial snapshot implementation that supports up to $2^W/5 - 1$ operations and uses $O(n^3 \log n)$ CAS objects, FAI objects and registers, each of size $W$ bits, such that \texttt{Scan}() has constant time (and step) complexity, and \texttt{Update}() and \texttt{Observe}() have $O(\log n)$ time (and step) complexity.

A predecessor object maintains a set of pairs, each comprising a *key* and a *value*. The keys are distinct elements of a totally ordered set $D_{\text{pred,key}}$ and the
values belong to a set $D_{\text{pred,val}}$. The predecessor type provides four operations, $\text{Insert}(k,v)$, $\text{Remove}(k)$, $\text{Pred}(k)$, and $\text{Succ}(k)$, where $k \in D_{\text{pred,\text{key}}}$ and $v \in D_{\text{pred,val}}$. The specification of each operation is as follows:

- **Insert**($k,v$): If the data structure does not contain a pair with key $k$, then the $\text{Insert}(k,v)$ operation inserts a pair with key $k$ and value $v$, and returns true. Otherwise, it does not change the data structure and returns false.

- **Remove**($k$): If the data structure contains a pair with key $k$, then the $\text{Remove}(k)$ operation removes the pair with key $k$, and returns true. Otherwise, it does not change the data structure and returns false.

- **Pred**($k$): If the data structure contains a pair with key less than or equal to $k$, then the $\text{Pred}(k)$ operation returns the pair $(x, \text{true})$, where $x$ is the pair with greatest key smaller than or equal to $k$. Otherwise, it returns the pair $(y, \text{false})$, where $y$ is an arbitrary pair.

- **Succ**($k$): If the data structure contains a pair with key greater than $k$, then the $\text{Succ}(k)$ operation returns the pair $(x, \text{true})$, where $x$ is the pair with smallest key larger than $k$. Otherwise, it returns the pair $(y, \text{false})$, where $y$ is an arbitrary pair.

We call operations $\text{Insert}()$ and $\text{Remove}()$ *update* operations, and all other operations *query* operations.

A predecessor object is *single-writer* if there is only one dedicated process that is allowed to perform updates. We will consider single-writer predecessor objects with *bounded capacity* $\Delta$, which informally means that at most $\Delta$ pairs can be stored in the data structure at any point in time. Since the object is single-writer, it is uniquely determined at the point of invocation of an update operation, whether that operation will be successful or not. (We call an incomplete update operation successful if it must be successful in any extension of the execution in which it completes.) Formally, a predecessor object with bounded capacity $\Delta$ only guarantees correctness if at any point in time the number of invocations of successful $\text{Insert}()$ operations minus the number of responses of successful $\text{Remove}()$ operations is at most $\Delta$.

**Theorem 2 (A Single-Writer Predecessor Algorithm).** Let $W$ be an integer greater than or equal to \(\max\{\log |D_{\text{pred,\text{key}}}|, \log |D_{\text{pred,val}}|, \log n + \)
There exists a wait-free linearizable single-writer predecessor implementation with bounded capacity $\Delta$ that uses $O(n \Delta \log \Delta + n^2)$ CAS objects and registers, each of size $W$ bits, such that each update and query operation has $O(\log \Delta)$ time (and step) complexity.

1.3 Building Blocks

As building blocks for our algorithms, we use two linearizable objects: A destination array and a multi-word LL/SC. For both of these objects, we use the implementations of Blelloch and Wei [14] from CAS objects and registers. The sequential specifications of these two types are as follows:

A destination array stores an array of $n$ entries and supports the operations $\text{Read}()$ and $\text{SWCopy}()$. Operation $\text{Read}(i)$ takes an integer $i \in \{0, \ldots, n-1\}$ as an argument, and returns the value of the $i$-th component of the array. Operation $\text{SWCopy}(R)$ takes a reference $R$ to a register as an argument, and if process $p$ calls that operation, it changes the value of the $p$-th component of the array to the value of register $R$.

Let $D_{\text{dst}}$ be the domain of values that can be written in an entry of a destination array.

Blelloch and Wei [14] show that a linearizable and wait-free destination array can be implemented from $O(n^2)$ CAS objects and registers, each of size $W$ bits, such that each $\text{Read}()$ and each $\text{SWCopy}()$ operation can be executed in a constant number of steps, given that $W \geq \max\{\log |D_{\text{dst}}|, \log n\} + O(1)$.

An LL/SC type stores a values and provides two operations, $\text{LL}()$ and $\text{SC}(v)$. An $\text{LL}()$ operation returns the value, and an $\text{SC}(v)$ call by process $p$ changes the value to $v$ if $p$ has previously called $\text{LL}()$ and no successful $\text{SC}()$ operation has occurred since then. An $\text{SC}()$ operation returns true if it succeeds to update the objects value, and returns false otherwise.

Let $D_{\text{llsc}}$ be the domain of values that can be stored in an LL/SC object. For a constant $k$, Blelloch and Wei [14] implemented a wait-free linearizable array of $n$ $k$-word LL/SC objects (i.e, each LL/SC object can store $kW$ bits) from $O(n^3)$ CAS objects and registers, each of size $W$ bits, such that each $\text{LL}()$ and each $\text{SC}()$ operation can be executed in constant number of steps, given that $W \geq \max\{\left(\log D_{\text{llsc}}\right)/k, 2 \log n\} + O(1)$. 
1.4 Outline

The rest of this thesis is organized as follows: First, we investigate the related work on standard and partial snapshot objects in Chapter 2. Then we present our single-writer predecessor object in Chapter 3, and our single-writer adaptive partial snapshot in Chapter 4. Finally, in Chapter 5, we provide a conclusion and a discussion about future work.
Chapter 2

Literature Review

Afek, Attiya, Dolev, Gafni, Merritt, and Shavit [1] provided a formal specification of the snapshot type and implemented a single-writer snapshot object from read-write registers with worst-case step complexity of $O(n^2)$ for \texttt{Scan()} and \texttt{Update()}. Following [1], most research focused on implementing single-writer snapshot objects from registers. Attiya, Herlihy, and Rachman [11] introduced a decision problem \textit{lattice agreement} and showed that any algorithm for solving this problem can be used to implement a single-writer snapshot object. Using this transformation, they implemented a single-writer snapshot object from registers with expected step complexity of $O(n \log^2 n)$ for \texttt{Update()} and \texttt{Scan()}. Later, Inoue, Chen, Masuzawa, and Tokura [20] constructed a more efficient algorithm for solving lattice agreement, which led to a deterministic single-writer snapshot object with worst-case linear step complexity. It follows from a proof technique by Jayanti, Tan, and Toueg [22] that this is optimal. They proved that any deterministic implementation of a single-writer snapshot object from read-write registers has $\Omega(n)$ worst-case step complexity, even when using unbounded registers. To obtain a snapshot object with sublinear step complexity, Aspnes, Attiya, Censor-Hillel, and Ellen [7] implemented a limited-use (i.e., the total number \texttt{Update()} and \texttt{Scan()} operations is a polynomial function of $n$) single-writer snapshot object with poly-logarithmic worst-case step complexity. Later, Aspnes and Censor-Hillel [8] devised a randomized algorithm with poly-logarithmic expected step complexity, and Mirza Baig, Hendler, Milani, and Travers [12] implemented a deterministic algorithm with poly-logarithmic amortized step complexity.

All these implementations assume that they can store an entire snapshot of size $n$ in a register. Assuming that a single-word register can store at
most one entry of a snapshot, they need $n$-word registers in their implementation. Although we can construct $n$-word registers from single-word ones, this inherently increases the step complexity of Read and Write operations by a factor of $\Omega(n)$. Riyani, Shavit, and Touitou [25] used single-word CAS objects and FAI objects to implement a single-writer snapshot object with constant step complexity for Update and $O(n)$ step complexity for Scan. Jayanti [21] generalized this result to the multi-writer case with $m$ components, and achieved a step complexity of $O(m)$ for Scan, even without relying on FAI.

Anderson [5] showed that given a single-writer snapshot object from registers with step complexity $S(n)$ for Scan and $U(n)$ for Update, one can construct a multi-writer snapshot object from registers with step complexity $O(S(n))$ for Scan and $O(U(n) + S(n))$ for Update.

Attiya, Guerraoui, and Ruppert [10] were the first to devise a snapshot object that allows a process to capture partial views of the snapshot array. They implemented a wait-free multi-writer partial snapshot algorithm from registers, CAS objects, and FAI objects, in which scanning $r$ array components takes $O(r^2)$ steps in the worst-case. There is no bound on the worst-case step complexity of an Update operation. However, the amortized step complexity per Update operation is bounded by the maximum interval contention, as well as the maximum number of components accessed by Scan operations.

The constant-time snapshot of Wei, Ben-David, Blelloch, Fatourou, Ruppert, and Sun [27] is a generalized version of the partial snapshot type. Their type maintains an array $A$ of $m$ CAS objects and provides three operations: CAS($i$, old, new), Scan(), and ReadSnapshot($h$, $i$). A Scan() operation takes a snapshot of the array and returns a handle $h$ to that snapshot. Later, a process can perform ReadSnapshot($h$, $i$) to determine the value that the $i$-th CAS object had at the point in time at which the snapshot was taken. They gave an implementation of this type from CAS objects and registers, where each CAS() and each Scan() takes a constant number of steps. However, the step complexity of performing ReadSnapshot($h$, $i$) is linear in the number of updates on $A[i]$ that occurred after the Scan() operation which returned $h$. Hence, their algorithm is unbounded (but not bounded) wait-free.

The authors use a version list for each CAS object that stores the complete history of updates performed on the object. Each update is associated with a global sequence number, which is also stored in the version list. The sequence
We write “unbounded” in column “Step complexity”, if the algorithm in the corresponding row is unbounded (but not bounded) wait-free. All algorithms use sequence numbers are incremented with Update() or Scan() operations, and store them in base objects. Therefore, these algorithms implicitly assume that the number of Scan() or Update() operations is bounded by $2^W$. (But We believe that in [10] unbounded sequence numbers can be avoided by using LL/SC instead of CAS.) It is safe to assume that this assumption will never be violated on current 64-bit architectures.

Table 2: Comparison of partial snapshot objects

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Step complexity</th>
<th>Base objects</th>
<th>Number of obj.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reading r loc.</td>
<td>Update</td>
<td></td>
</tr>
<tr>
<td>[10]</td>
<td>$O(r^2)$</td>
<td>unbounded</td>
<td>n-word CAS and FAI</td>
</tr>
<tr>
<td>[27]</td>
<td>unbounded</td>
<td>$O(1)$</td>
<td>single-word CAS</td>
</tr>
<tr>
<td>This work</td>
<td>$O(r \log n)$</td>
<td>$O(\log n)$</td>
<td>single-word CAS and FAI</td>
</tr>
</tbody>
</table>

number is incremented with each Scan() operation and is returned as the handle. It can then be used to identify the latest update that was applied to a CAS object, using the version list.

Our algorithm is motivated by [27]. In our object, we replace the version list with a single-writer concurrent red-black tree. Using a pruning method, we maintain the invariant that the number of elements in a red-black tree is at most $3n$. This allows a process to find the latest update before a Scan() operation in $O(\log n)$ time. A comparison of our implementation with the partial snapshot objects of [10, 27] is shown in Table 2.

The idea of associating data structure modifications with timestamps has also been used in algorithms for software transactional memory [26] and multi-version databases [28]. These algorithms are either not wait-free or allow operations to fail. Moreover, using an FAI object as a global clock to linearize operations at the time of FAI() operations is used in [6] to obtain a general method to add range queries to data structures.
Chapter 3

Single-Writer Predecessor Object

In this chapter, we present our linearizable wait-free single-writer predecessor object with bounded capacity $\Delta$. First, in Section 3.1, we describe the sequential persistent red-black tree implementation of [15]. We show that this algorithm can be directly used as a linearizable single-writer predecessor object. In this algorithm, during each update operation (Insert() or Remove()), the number of base objects grows by $O(\log \Delta)$ and thus, in an infinite execution we need an unbounded number of base objects. In Section 3.2, we implement a memory reclamation algorithm to bound the number of base objects by a function of $n$ and $\Delta$. Afterward, in Sections 3.3 and 3.4, we provide the correctness proof and the analysis of our algorithm, respectively.

3.1 The Basic Algorithm

Driscoll, Sarnak, Sleator, and Tarjan [15], present a technique called node-copying to make linked data structures persistent. (A version of a data structure is the state of the entire data structure after performing multiple (maybe zero) update operations. “A data structure is persistent if it allows access to multiple versions of that data structure. [15]”) We first describe a basic version of this technique that can be applied to any binary search tree (BST) implementation, where each node store only pointers to its children (i.e., there are no parent pointers):

1Section 3.1 is taken from our publication [13].
A dedicated variable $R$ stores a pointer to the root $r$ of the tree. An update operation does not modify any nodes of the data structure. Instead, it adds copies of all nodes that need to be modified, as well as a new root $r'$, and finally changes the pointer $R$ so that it points to $r'$ instead of $r$. To be more precise, suppose the set of nodes reachable from $r$ form a conventional BST $T_1$. Let $T_2$ be the BST obtained by applying a conventional update operation to $T_1$ (e.g., an insertion). Let $S$ be the set of nodes in $T_2$, that are either added or modified by this update operation, and $S'$ the parent-closure of $S$ (i.e., if $v$ is in $S'$, then the parent of $v$ is also in $S'$). Instead of modifying the nodes in $S$, we create a copy $v'$ of each node $v$ in $S'$. Each field of $v'$ has the same value as the corresponding field in $v$, except that a pointer to a node $u$ in $S'$ is replaced with a pointer to the copy $u'$ of that node. Since $S'$ is parent-closed, the root $r$ of $S'$ is also copied into a node $r'$. It is easy to see that the nodes reachable from $r'$ now form a BST that is equivalent to $T_2$. Hence, to complete the update operation, it suffices to replace the value of $R$ with a pointer to $r'$. Figure 3 shows an illustration of inserting a node with value 7 to a binary search tree that applies the node copying technique.

To perform a query operation on the persistent data structure, a process simply reads the pointer $R$ to obtain a root $r$ and then performs the same operations as it would in the conventional BST algorithm, using $r$ as a root.

For the purpose of concreteness, we will now consider a red-black tree [18]. Driscoll, Sarnak, Sleator, and Tarjan have applied the node copying technique described above to that data structure to obtain a persistent red-black tree, where each of the operations Insert(), Remove() and Find() take $O(\log m)$
steps [15], where $m$ is the number of elements stored in the tree. It is straightforward to augment the data structure with query operations $\text{Pred()}$ and $\text{Succ()}$ so that all operations have time complexity $O(\log m)$. Thus, we obtain an implementation of a persistent sequential predecessor type with the same asymptotic time complexity. Recall that our object has bounded capacity $\Delta$. Hence, at any point in time, at most $\Delta$ elements are stored in the tree and thus, each update and query operation takes $O(\log \Delta)$ steps.

We can now use that persistent predecessor implementation in a shared-memory system, by storing $R$ and each node of the data structure in an atomic register. We will allow only one process, $p_w$, to perform update operations, but all processes are allowed to execute query operations. Observe that if at some point $t$, pointer $R$ points to a root $r$ in the data structure, all nodes reachable from $r$ form the BST that was obtained as a result of the update operation that wrote the pointer to $r$ into $R$. None of these nodes can change after point $t$. Hence, if a query operation reads the pointer to $r$ from $R$, then it will visit exactly the same nodes that would be visited in the sequential case. Similarly, an update operation by process $p_w$ does initially not make changes to any reachable nodes, and all tree modifications will become visible to other processes only when $p_w$ changes the root pointer, $R$, to point to the new root copy $p_w$ created. Hence, it is easy to see that each update operation can linearize with the write to $R$, and each query operation can linearize with the read of $R$. It follows that this object is linearizable, provided that only one process can perform update operations. (In fact, it is linearizable as long as no two update operations are concurrent.)

### 3.2 The Memory Reclamation Algorithm

During each update operation, process $p_w$ creates $O(\log \Delta)$ nodes. Thus, in infinite executions, we need an unbounded number of nodes. We implement a $\text{Recycle()}$ method, in which $p_w$ detects the nodes that cannot be accessed by any process anymore. Therefore, when $p_w$ wants to create a node, it can use one of the nodes it created earlier, which cannot be accessed anymore, instead.

To accommodate our description and proofs, we present Algorithm 1, which incorporates the $\text{Recycle()}$ method but uses an unbounded number of nodes. Then we slightly modify Algorithm 1 to obtain Algorithm 2, which uses $O(n\Delta \log \Delta)$ nodes.
Node Structure  Let $\lambda$ be a sufficiently large integer in $O(n\Delta \log \Delta)$. (We later show the exact value of $\lambda$.) For each integer $i$ and each integer $j$, where $0 \leq i < \lambda$ and $j \geq 0$, $w_{i,j}$ is a node. (Recall that we use the red-black tree implementation of [15] in our algorithm.) Each node is an instance of a structure that has 6 fields: A color, a key, a value, an index, and two pointers that point to the left child and the right child. These 6 fields are denoted by $col$, $key$, $val$, $ind$, $lc$, and $rc$, respectively. We can store a field of a node in a register. Let $w_{i,j}$ be a node and $x$ a field in \{$col$, $key$, $val$, $ind$, $lc$, $rc$\}. The field $x$ of $w_{i,j}$ is denoted by $w_{i,j}.x$. Also, $w_{i,j}.ind = i$.

High-Level Node Management  Process $p_w$ uses a local queue $free$. Initially, $free$ consists of $\lambda = O(n\Delta \log \Delta)$ shared nodes $\{w_{0,0}, \ldots, w_{\lambda-1,0}\}$. When $p_w$ wants to create a node, it dequeues a node $w_{i,j}$ from $free$ and then fills the fields of $w_{i,j}$, instead. (Recall that $p_w$ creates a node only in an update operation.) Later, in a $Recycle()$ operation, when $p_w$ detects that $w_{i,j}$ cannot be accessed by any process anymore (a node $v$ can be accessed after a point in time $t$ if a process $p$ may visit $v$ after $t$ in a query operation. We will later describe how $p_w$ can detect the nodes that cannot be accessed anymore.), it enqueues $w_{i,j+1}$ into $free$.

In Algorithm 1, $p_w$ does not recycle any node, and it uses method $Recycle()$ to detect the nodes that cannot be accessed by any process anymore. Later, in Algorithm 2, $p_w$ uses the same scheme as Algorithm 1 with $\lambda$ shared nodes $\{w_0, w_1, \ldots, w_{\lambda-1}\}$. Whenever $p_w$ detects that a node $w_i$ cannot be accessed anymore, it enqueues $w_i$ into $free$ again. This results in using at most $\lambda$ nodes throughout any execution.

Let $i$ be an integer in $\{0, \ldots, \lambda - 1\}$. Initially the only node with index $i$ in $free$ is $w_{i,0}$. Also, for an integer $j \geq 0$, $w_{i,j+1}$ is enqueued to $free$, when $p_w$ detects that $w_{i,j}$ cannot be accessed anymore. Therefore, $w_{i,j}$ is dequeued at most once from $free$. Let $up$ be the update operation, in which $p_w$ dequeues $w_{i,j}$ from $free$. Suppose $p_w$ changes the tree root from $r$ to $r'$ in $up$. Clearly, after $p_w$ writes $r'$ to $R$ in $up$, $p_w$ does not change $w_{i,j}$ anymore.

Let $t$ be a point in time and $i$ an integer in $\{0, \ldots, \lambda - 1\}$. Suppose $j$ is the greatest integer, such that $w_{i,j}$ is enqueued into $free$ before $t$. If for every $j' \geq 0$, $w_{i,j'}$ is not enqueued into $free$ before $t$, we let $j = 0$. We define $curnode(i,t) = w_{i,j}$. If $j = 0$, then for each integer $j' > 0$, $w_{i,j'}$ has not been enqueued to $free$ before $t$. Therefore, $p_w$ has not filled the fields of $w_{i,j'}$ before
t. Otherwise, before \( t \), \( p_w \) enqueued \( w_{i,j} \) into \textit{free}. Hence, for every integer
\( 0 \leq j' < j \), \( p_w \) has already detected before \( t \) that \( w_{i,j'} \) cannot be accessed by any process anymore. Also, for every integer \( j' > j \), \( p_w \) has not filled the fields of \( w_{i,j'} \) before \( t \). Therefore, to implement a \texttt{Recycle()} method, for every integer \( i^* \in \{0, \ldots, \lambda - 1\} \) and a point in time \( t^* \), \( p_w \) only needs to detect if \( \texttt{curnode}(i^*, t^*) \) can be accessed after \( t^* \).

**Main Idea**  Consider an execution. Let \( r_0 \) be the initial root of the tree and \( r_i \) the root that is written to \( R \) in the \( i \)-th update operation by process \( p_w \). Let \( t_i \) be the point in time at which \( p_w \) writes \( r_i \) to \( R \). We define \textit{reachable}(\( r_i \)) as the set of reachable nodes from \( r_i \) at \( t_i \). Before \( t_i \), \( p_w \) has already filled the fields of every node in \textit{reachable}(\( r_i \)). Therefore, after \( t_i \), \( p_w \) does not change any node in \textit{reachable}(\( r_i \)) and thus, the set of nodes reachable from \( r_i \) remains unchanged after \( t_i \). Let \( w_{i,j} \) be a node, and \( t \) a point in time. Let \( r_k \) be the root of the tree pointed to by \( R \) at \( t \). We say a node \( w_{i,j} \) at \( t \) is outdated if

\[
\exists j' \in \{0, \ldots, k - 1\}, w_{i,j} \in \textit{reachable}(r_j) \setminus \textit{reachable}(r_k)
\]  

(3.1)

Suppose \( \texttt{curnode}(i, t) \) is outdated at \( t \). Thus, for some \( j \), where \( 0 \leq j \leq k - 1 \), \( \texttt{curnode}(i, t) \in \textit{reachable}(r_j) \). Although \( \texttt{curnode}(i, t) \) is outdated at \( t \), it might be still visited by a process in a query operation after \( t \): Let \( oq \) be a query operation by a process \( p \). Suppose \( t_1 < t \) is the point in time at which \( p \) reads a tree root from \( R \) in \( oq \). Let \( r_j \) be that tree root. During \([t_1, \texttt{rsp}(oq)]\), process \( p \) may visit any node in \textit{reachable}(\( r_j \)), including \( \texttt{curnode}(i, t) \). Therefore, if \( \texttt{rsp}(oq) > t \), then \( \texttt{curnode}(i, t) \) may be visited by \( p \) after \( t \). To detect if \( \texttt{curnode}(i, t) \) will be accessed after \( t \) by a process in a query operation, processes use an announcement array \( M[0 \ldots n - 1] \) that has an entry for each process. When process \( p \) reads a tree root \( r \) from \( R \) in a query operation, \( p \) protects the nodes in \textit{reachable}(\( r \)) by writing \( r \) to \( M[p] \). Hence, if \( \texttt{curnode}(i, t) \) is outdated and not protected at \( t \), then it cannot be accessed by any process after \( t \).

**The \texttt{Query()} method**  Suppose \( M \) is an array of registers. In a query operation \( oq \) by a process \( p \), \( p \) may read a tree root \( r \) from \( R \) and then fall asleep before writing \( r \) into \( M[p] \). Meanwhile, \( p_w \) cannot detect that the nodes in \textit{reachable}(\( r \)) are protected. To avoid this, we replace \( M \) with the destination array (see Section 1.3) of Blelloch and Wei [14] and let \( p \) per-
Shared:
- Register $R = \perp$
- Destination Array $M$
- Node $w_{i,j}$, for each $i \in \{0, \ldots, \lambda - 1\}$, $j \geq 0$

Local for Writer:
- Queue $\text{free} = < w_{0,0}, \ldots, w_{\lambda-1,0} >$
- Array $\text{outdated}[0, \ldots, \lambda - 1] = [(\text{false}, 0), \ldots, (\text{false}, 0)]$
- Array $\text{last}[0, \ldots, \lambda - 1] = [0, 0, \ldots, 0]$
- Integer $sw = 0$

1. **Function Update**
   - Let $r$ be the root pointed to by $R$.
   - Starting from the root $r$, perform the corresponding update operation by taking exactly the same steps as in the sequential implementation in [15].
   - When $p_w$ wants to create a node, dequeue a node $w_{i,j}$ from free and use $w_{i,j}$ instead.
   - Let $r'$ be the new root pointed to by $R$.
   - For each node $v \in \text{reachable}(r) \setminus \text{reachable}(r')$, set $\text{outdated}[v.\text{ind}] \leftarrow (\text{true}, 1 - sw)$.
   - Every $n\Delta \text{Update}()$ calls, set $sw \leftarrow 1 - sw$ and start a new Recycle() method.
   - Contribute $\kappa \log \Delta$ steps towards an ongoing Recycle() call, where $\kappa$ is a sufficiently large constant.

2. **Function Query**
   - $\text{M.SWCopy}(R)$
   - $r \leftarrow \text{M.Read}(\text{myID})$
   - Starting from the root $r$, perform the corresponding query operation by taking exactly the same steps as in the sequential implementation in [15].

3. **Function Recycle**
   - Let $\text{protected}$ be a Boolean array of size $\lambda$ with initial value of false for each entry.
   - for $p \in \{0, \ldots, n - 1\}$ do
     - $r \leftarrow \text{M.Read}(p)$
     - if $r \neq \perp$ then
       - for $v \in \text{reachable}(r)$ do
         - $\text{protected}[v.\text{ind}] \leftarrow \text{true}$
     - for $i \in \{0, \ldots, \lambda - 1\}$ do
       - if $\text{outdated}[i] = (\text{true}, sw)$ then
         - if $\text{protected}[i] = \text{false}$ then
           - $\text{outdated}[i] = (\text{false}, sw)$
           - $\text{last}[i] \leftarrow \text{last}[i] + 1$
           - $\text{free}.\text{Enq}(w_{i,\text{last}[i]})$
         - else $\text{outdated}[i] = (\text{true}, 1 - sw)$

**Algorithm 1:** Modified Single-Writer Predecessor Algorithm
form $M.$\textsc{SWCopy}(R) at the beginning of $oq$. (The main motivation for the definition of the destination array has been to solve the same type of “protection” problem. Other sophisticated techniques to deal with that in constant time and using only registers have been described in [2, 3].) By performing $M.$\textsc{SWCopy}(R), $p$ copies the tree root $r$ that is pointed to by $R$ into $M[p]$ and thus, protects the nodes in $\text{reachable}(r)$. We linearize $oq$ at the point in time at which $p$ performs $M.$\textsc{SWCopy}(R) in line 10. After this operation, $p$ reads $r$ by performing $M.$\textsc{Read}(p) and it then performs exactly the same steps as in the sequential implementation in [15], starting from $r$.

Next, we describe in detail the implementation of update and \textsc{Recycle()} methods. As we will later explain, the time complexity of each \textsc{Recycle()} call is $O(n\Delta \log \Delta)$. Hence, to obtain an efficient implementation, we distribute the work of a \textsc{Recycle()} call over $n\Delta$ update operations. That is, every $n\Delta$ update operations, process $p_w$ calls a new \textsc{Recycle()} method, and at the end of every update operation, $p_w$ contributes $\kappa \log \Delta$ steps towards an ongoing \textsc{Recycle()} call, where $\kappa$ is a sufficiently large constant.

\textbf{Variables} First, we discuss the local variables that process $p_w$ uses for method \textsc{Recycle()}. We store a local integer $sw$ and two local arrays $\text{outdated}[0 \ldots \lambda - 1]$ and $\text{last}[0 \ldots \lambda - 1]$ for process $p_w$. Let $i$ be an integer in $\{0, \ldots, \lambda - 1\}$ and $t$ a point in time. Initially, $\text{last}[i] = 0$, and whenever, $p_w$ detects that $\text{curnode}(i,t)$ cannot be accessed anymore, it increases $\text{last}[i]$ by 1 and then enqueues $w_i,\text{last}[i]$ into $\text{free}$. The initial value of $sw$ is 0, and its value is changed to $1 - sw$ whenever process $p_w$ calls a new \textsc{Recycle()} method. The value stored in $\text{outdated}$ is a tuple $(x,y)$, where $x \in \{\text{false, true}\}$ and $y \in \{0,1\}$. Let $t$ be a point in time. At $t$, the first component of $\text{outdated}[i]$ indicates whether $\text{curnode}(i,t)$ is outdated at $t$ or not. Suppose the value of $\text{outdated}[i]$ is $(\text{true}, y)$ at $t$. Process $p_w$ can compare $y$ to the value of $sw$ at $t$ to detect whether $\text{curnode}(i,t)$ became outdated during or before the ongoing \textsc{Recycle()} operation.

\textbf{The Update() Method} Suppose $up$ is an update operation, in which $p_w$ changes the root pointer from $r$ to $r'$. We linearize $up$ with the write of $r'$ to $R$. After $\text{lin}(up)$, for each node $v \in \text{reachable}(r) \setminus \text{reachable}(r')$, $p_w$ sets $\text{outdated}[v,\text{ind}]$ to $(\text{true}, 1 - sw)$ in line 32 because $v$ is no longer reachable by the current tree root. Note that each node $v$ is in the parent closure of
the nodes that are modified by a standard red-black tree implementation that applies the node copying technique. Therefore, considering that our algorithm has bounded capacity \( \Delta \),

\[
|\text{reachable}(r) \setminus \text{reachable}(r')| \leq c_{rb} \log \Delta
\]  

(3.2)

where \( c_{rb} \) is a constant. Moreover, we can compute this set in \( O(\log \Delta) \) steps.

Afterward, if \( up \) is the \( kn\Delta \)-th update operations by \( p_w \), for some integer \( k \geq 1 \), \( p_w \) changes \( sw \) to \( 1 - sw \) and starts a new \texttt{Recycle()} method. Finally, \( p_w \) contributes \( O(\log \Delta) \) steps towards an ongoing \texttt{Recycle()} call.

**The \texttt{Recycle()} Method**  
Let \( rc \) be a \texttt{Recycle()} operation. Assume \( r_j \) is the root pointer that \( p_w \) writes to \( R \) in the \( j \)-th update operation. Suppose \( r_k \) is the tree root at \( inv(rc) \), \( S_{out} \) the set of outdated nodes at \( inv(rc) \), and \( x \) the value of \( sw \) at \( inv(rc) \). (Notice that for every \( j \geq k \) and for each node \( v \in S_{out}, v \notin \text{reachable}(r_j) \).) In \( rc \), \( p_w \) first creates a Boolean array \texttt{protected} of size \( \lambda \) with initial value of false for each entry in line 40. Then \( p_w \) sets \( \texttt{protected}[v.ind] \) to true for each protected node \( v \) as follows: For every process \( p \in \{0, \ldots, n-1\} \), \( p_w \) performs \texttt{M.Read}(\( p \)) and assigns the return value of that operation to \( r \) in line 42. Afterward, \( p_w \) iterates over every \( v \in \text{reachable}(r) \) and sets \( \texttt{protected}[v.ind] \) to true in line 45. (If \( r = \bot \), then \( r \) points to a tree without any node so \( p_w \) does not need to compute \( \text{reachable}(r) \). Otherwise, \( p_w \) can compute \( \text{reachable}(r) \) using a tree traversal algorithm such as depth-first-search.) Notice that if \( p \) changes \( M[p] \) after \( p_w \) performs \texttt{M.Read}(\( p \)) in line 42 of \( rc \), then \( p \) changes \( M[p] \) to a tree root \( r_j \), where \( j \geq k \) and thus, \( p \) does not protect any node in \( S_{out} \). After the for-loop in lines 15-19, \( p_w \) iterates over every integer \( i \in \{0, \ldots, \lambda - 1\} \) in the for-loop in lines 20-26. First, \( p_w \) observes the value of \( \text{outdated}[i] \) in line 21 at a point in time \( t_i \). If \( \text{outdated}[i] \) is \( (\text{true}, x) \) at \( t_i \), then \( \text{curnode}(i, t_i) \) became outdated before \( inv(rc) \). (Recall that \( p_w \) does not change the value of \( sw \) during \( [inv(rc), rsp(rc)] \). Therefore, if \( \text{curnode}(i, t_i) \) becomes outdated during \( rc \), then \( p_w \) sets the second component of \( \text{outdated}[i] \) to \( 1 - x \) in line 6 of an update operation.) Hence, \( \text{curnode}(i, t_i) \in S_{out} \). Afterward, \( p_w \) observes the value of \( \text{protected}[i] \) in line 22. If \( \text{protected}[i] = \text{false} \), then \( \text{curnode}(i, t_i) \) is outdated and not protected at \( t_i \). Thus, \( \text{curnode}(i, t_i) \) will not be accessed by any process after \( t_i \). Therefore, \( p_w \) sets \( \text{outdated}[i] \) to \( (\text{false}, x) \) in line 23,
increases \( last[i] \) by 1 in line 24, and enqueues \( w_i, last[i] \) into \( free \) in line 25. If \( outdated[i] \) is \((true, x)\) at \( t_i \) and \( protected[i] = true\), then \( p_w \) changes \( outdated[i] \) to \((true, 1 - x)\) in line 51. Suppose \( rc' \) is the next \texttt{Recycle()} operation after \( rc \). The value of \( sw \) is \( 1 - x \) during \( rc' \). Hence, by setting the second component of \( outdated[i] \) to \( 1 - x \), \( p_w \) indicates that \( curnode(i, t_i) \) became outdated before \( inv(rc') \).

**Analysis**  Now we discuss that if \( \lambda = 4c_{rb}n\Delta \log \Delta \), then at any point in time, the number of elements in \( free \) is greater than 0. This suffices for the correctness of our memory reclamation technique because whenever \( p_w \) wants to create a node, it can always dequeue a new node from \( free \).

Consider the \texttt{Recycle()} operation \( rc \), as described above. Let \( rc_0 \) be the first \texttt{Recycle()} call by \( p_w \), and \( rc' \) the next \texttt{Recycle()} operation after \( rc \). It is clear that the number of elements in \( free \) is greater than \( c_{rb}n\Delta \log \Delta \) at \( inv(rc_0) \). Let \( x \) and \( y \) be the number of elements in \( free \) at \( inv(rc) \) and at \( inv(rc') \), respectively. We show that if \( x \geq c_{rb}n\Delta \log \Delta \), then \( y \geq c_{rb}n\Delta \log \Delta \). Thus, using induction, we can conclude that the number of elements in \( free \) is greater than or equal to \( c_{rb}n\Delta \log \Delta \) at the invocation of every \texttt{Recycle()} call.

Let \( i \) be an integer in \( \{0, \ldots, \lambda - 1\} \) and \( r \) the tree root at \( inv(rc) \). One can observe that at \( inv(rc) \), either \( curnode(i, inv(rc)) \) is in \( free \), \( curnode(i, inv(rc)) \in reachable(r) \), or \( outdated[i] = (true, z) \), where \( z \) is the value of \( sw \) at \( inv(rc) \). (This is formally proved in Claims 17 and 18.) Let \( U \) be the set of indices \( j \in \{0, \ldots, \lambda - 1\} \), such that \( outdated[j] = (true, z) \) at \( inv(rc) \). Since our algorithm has bounded capacity \( \Delta \), \( |reachable(r)| \leq \Delta \). Hence, \( |U| \geq \lambda - (x + \Delta) \). During \( rc \), for every node in \( v \in U \), if \( protected[v.ind] = false \), then \( p_w \) enqueues a node into \( free \). In the for-loop in lines 15-19, \( p_w \) reads \( n \) tree roots, and if a node \( v \) is reachable by one of those roots, then \( p_w \) sets \( protected[v.ind] \) to true. Hence, at most \( n\Delta \) entries in \( protected \) are true. Therefore, at least \( |U| - n\Delta \) nodes will be enqueued into \( free \) during \( rc \). Also, \( rc \) is distributed over \( n\Delta \) update operations, during each of which \( p_w \) dequeues at most \( c_{rb} \log \Delta \) nodes from \( free \). (Recall that \( c_{rb} \) is the constant from Equation 3.2.) Thus,
\[ y \geq x + (|U| - n\Delta) - (n\Delta c_{rb} \log \Delta) \]
\[ \geq x + (\lambda - (x + \Delta) - n\Delta) - (n\Delta c_{rb} \log \Delta) \]
\[ = \lambda - c_{rb} n\Delta \log \Delta - n\Delta - \Delta \]
\[ = 4c_{rb} n\Delta \log \Delta - c_{rb} n\Delta \log \Delta - n\Delta - \Delta \]
\[ \geq c_{rb} n\Delta \log \Delta. \] (3.3)

Since \( p_w \) performs a new \texttt{Recycle()} call every \( n\Delta \) update operations, and during each update operation, \( p_w \) dequeues at most \( c_{rb} \log \Delta \) nodes from \texttt{free}, at any point in time, the number of elements in \texttt{free} is greater than 0.

As discussed above, for a tree root \( r \), \( p_w \) can compute \texttt{reachable(r)} in \( O(\Delta) \) steps. Also, performing each operation on \( M \) takes constant time. Hence, the time complexity of a \texttt{Recycle()} call is \( O(n\Delta + \lambda) = O(n\Delta \log \Delta) \). Each \texttt{Recycle()} operation is distributed over \( n\Delta \) update operations. Thus, the extra work in each update call is \( O(\log \Delta) \). In an update operation, in which \( p_w \) changes the tree root from \( r \) to \( r' \), \texttt{reachable(r)}\textbackslash\texttt{reachable(r')} can be computed in \( O(\log \Delta) \) steps. Hence, the time complexity of an update operation is \( O(\log \Delta) \). As explained earlier, in a query operation by process \( p \), \( p \) performs \texttt{M.SWCopy(R)}, then it reads \( r \) by performing \texttt{M.Read(p)}, and afterward it performs exactly the same steps as in as in the sequential implementation in [15], starting from \( r \). Thus, the time complexity of a query operation is also \( O(\log \Delta) \).

Using a Bounded Number of Nodes  Now we describe how to modify Algorithm 1 to obtain Algorithm 2, which uses \( \lambda \) nodes. In Algorithm 1, when \( p_w \) enqueues a node \( w_{i,j} \) into \texttt{free} at \( t \), \( w_{i,j-1} \) cannot be accessed by any process after \( t \). Therefore, if at \( t \), \( p_w \) enqueues \( w_{i,j-1} \) instead of \( w_{i,j} \) into \texttt{free}, then our algorithm is still linearizable. Hence, we can obtain Algorithm 2 as follows: Initially, \texttt{free} consists of \( \lambda \) nodes \( w_0, \ldots, w_{\lambda-1} \). When \( p_w \) wants to create a node, it dequeues a node \( w_i \) from \texttt{free} and fills the fields of \( w_i \), instead. Afterward, when \( p_w \) detects that \( w_i \) cannot be accessed by any process anymore, \( p_w \) enqueues \( w_i \) into \texttt{free} again.

In Algorithm 2, the set of nodes reachable from a tree root \( w_i \) may change throughout the execution because \( w_i \) may be recycled multiple times. Thus, process \( p_w \) cannot call \texttt{reachable(w_i)} in that algorithm. (In Algorithm 1,
$p_w$ is the only process that calls $reachable(w_i)$. In Algorithm 2, we replace $reachable(w_i)$ by a function $\text{Traverse}(w_i)$ that simply performs a depth-first-search from $w_i$ and returns the nodes that it visits. Since $p_w$ is the only process that performs the $\text{Recycle()}$ method, the set of nodes reachable from $w_i$ remains unchanged while $p_w$ computes $\text{Traverse}(w_i)$.

### 3.3 Correctness Proof

Let $\mathcal{E}$ be an arbitrary finite execution of Algorithm 1. Since our implementation is wait-free, we can assume without loss of generality that all operations in $\mathcal{E}$ complete. If some operations do not complete, we can construct a completion of the execution, by letting processes with pending operations take steps in an arbitrary order until their operations have completed. Note that this is possible due to the progress property of our algorithm (i.e wait-freedom), see Section 1.1.

Whenever process $p_w$ dequeues a node from $\text{free}$, then that node is dequeued for the first time. (See Observation 3.) We will also show in Lemma 4 that the number of elements in $\text{free}$ at any point in time is greater than 0. Therefore, whenever $p_w$ performs a $\text{free.Deq()}$ operation, it always dequeues a new node. Thus, similar to our basic algorithm, we can linearize an update operation with the write to $R$ and a query operation with $\text{M.SWCopy(R)}$. Considering Observation 3 and Lemma 4, the linearizability proof of Algorithm 1 is simple. Thus, we will not provide a linearizability proof of this algorithm.

To prove the correctness of Algorithm 2, we also need to show that whenever $p_w$ enqueues a node $w_{i,j+1}$ into $\text{free}$ at a point in time $t$, then $w_{i,j}$ will not be accessed by any process after $t$. This is proved in Lemma 5.

In the following, the number of elements in $\text{free}$ is denoted by $|\text{free}|$, and the first and the second component of $\text{outdated}[i]$ by $\text{outdated}[i].val$ and $\text{outdated}[i].sw$, respectively. Also, $x_t$ denotes the value of a shared or a local variable $x$ at a point in time $t$. If $x$ is modified at $t$, we let $x_t$ be the value of $x$ just after $t$. Suppose $up$ is an update operation, in which $p_w$ changes the tree root from a root $r$ to another root $r'$. We let $r_{\text{old}}(up) = r$ and $r_{\text{new}}(up) = r'$.

**Observation 3.** Let $w_{i,j}$ be a node. Then $w_{i,j}$ is dequeued at most once from $\text{free}$.

**Proof.** The only line in the pseudocode, in which $p_w$ enqueues a node into
Shared:
  Register \( R = \perp \)
  Destination Array \( M \)
  Node \( w_i \), for each \( i \in \{0, \ldots, \lambda - 1\} \)

Local for Writer:
  Queue \( free = < w_0, \ldots, w_{\lambda-1} > \)
  Array \( outdated[0 \ldots \lambda - 1] = [(false, 0), \ldots, (false, 0)] \)
  Integer \( sw = 0 \)

Function Update()
  Let \( r \) be the root pointed to by \( R \).
  Starting from the root \( r \), perform the corresponding update
    operation by taking exactly the same steps as in the sequential
    implementation in [15].
  When \( p_w \) wants to create a node, dequeue a node \( w_i \) from \( free \) and
    use \( w_i \) instead.
  Let \( r' \) be the new root pointed to by \( R \).
  For each node \( v \in \text{Traverse}(r) \setminus \text{Traverse}(r') \), set
    \( outdated[v.ind] \leftarrow (true, 1 - sw) \).
  Every \( n\Delta \) Update() calls, set \( sw \leftarrow 1 - sw \) and start a new
    Recycle() method
  Contribute \( \kappa \log \Delta \) steps towards an ongoing Recycle() call, where
    \( \kappa \) is a sufficiently large constant.

Function Query()
  \( M_.SWCopy(R) \)
  \( r \leftarrow M_.Read(myID) \)
  Starting from the root \( r \), perform the corresponding query operation
    by taking exactly the same steps as in the sequential
    implementation in [15].

Function Recycle()
  Let \( protected \) be a Boolean array of size \( \lambda \) with initial value of false
  for each entry.
  for \( p \in \{0, \ldots, n - 1\} \) do
    \( r \leftarrow M_.Read(p) \)
    if \( r \neq \perp \) then
      for \( v \in \text{Traverse}(r) \) do
        \( protected[v.ind] \leftarrow true \)
  for \( i \in \{0, \ldots, \lambda - 1\} \) do
    if \( outdated[i] = (true, sw) \) then
      if \( protected[i] = false \) then
        \( outdated[i] = (false, sw) \)
      \( free_.Enq(w_i) \)
    else \( outdated[i] = (true, 1 - sw) \)

Algorithm 2: Single-Writer Predecessor Algorithm
free is line 25 of a Recycle() operation. In a Recycle() operation, before \(p_w\) enqueues \(w_{i,last[i]}\) into free in line 25, \(p_w\) increases \(last[i]\) by 1. Also, \(p_w\) does not change \(last[i]\) anywhere else in the pseudocode. Hence, by the fact that the initial value of \(last[i]\) is 0, each node \(w_{i,j'}\), where \(j' \geq 1\), is enqueued at most once in free. Initially, free consists \(\{w_{0,0}, \ldots, w_{\lambda-1,0}\}\). Therefore, each node \(w_{i,j'}\), where \(j' \geq 0\), is dequeued at most once from free.

For a node \(w_{i,j}\), \(start(w_{i,j})\) denotes the point in time at which \(w_{i,j}\) is dequeued from free and \(end(w_{i,j})\) is the first point in time after \(start(w_{i,j})\) at which \(p_w\) performs free.Enq\((w_{i,last[i]}\) in line 25 of a Recycle() operation.

The proofs of Lemmas 4 and 5 are provided in Section 3.3.1 and 3.3.2, respectively. For the rest of this section, we provide the necessary observations and claims that lead to the correctness of Algorithm 2.

**Lemma 4.** At any point in time, |free| is at least 1.

**Lemma 5.** Let \(p\) be a process and \(oq\) a query operation by \(p\). If \(p\) visits a node \(w_{i,j}\) during \(oq\) at a point in time \(t\), then \(t < end(w_{i,j})\).

**Observation 6.** Let \(up\) and \(up'\) be two consecutive update operations. The set of reachable nodes from \(rnew(up')\) consists of the new nodes that \(p_w\) dequeues from free during \(up'\) and a subset of nodes from \(reachable(rnew(up))\).

**Observation 7.** Let \(i\) be an integer in \(\{0, \ldots, \lambda - 1\}\) and \(t_1\) a point in time at which \(outdated[i].val = true\). Suppose \(t_2\) is the first point in time after \(t_1\), such that \(outdated[i].val_{t_2} = false\). Then at \(t_2\), \(p_w\) sets \(outdated[i].val\) to false in line 23 of a Recycle() operation.

**Claim 8.** Let \(i\) be an integer in \(\{0, \ldots, \lambda - 1\}\) and \(t_1\) a point in time at which \(outdated[i].val = false\). Suppose \(t_2\) is the first point in time after \(t_1\), such that \(outdated[i].val_{t_2} = true\). Then at \(t_2\), \(p_w\) sets \(outdated[i].val\) to true in line 6 of an update operation.

**Proof.** The only lines in the pseudocode in which \(p_w\) sets \(outdated[i].val\) to true are line 6 of an update operation and line 26 of a Recycle() operation. For the purpose of contradiction, assume at \(t_2\), \(p_w\) sets \(outdated[i].val\) to true in line 26 of a Recycle() operation \(rc\). During \(rc\), before \(t_2\), \(p_w\) performs line 21 of \(rc\) and observes that the value of \(outdated[i].val\) is true. Let \(t^* < t_2\) be that point in time. Since \(t_2\) is the first point in time after \(t_1\), such that \(outdated[i].val_{t_2} = true\), \(outdated[i].val = false\) throughout \([t_1, t_2)\). By the
fact that $t^* < t_2$ and outdated.$i$.val$^*$ = true, $t^* < t_1$. Suppose $t_3$ is the first point in time after $t^*$, such that outdated.$i$.val$^*_3 = false$. Since $t^* < t_1$ and outdated.$i$.val$^*_1 = true$, $t_3 ≤ t_1 < t_2$. By Observation 7, at $t_3$, $p_w$ sets outdated.$i$.val to false in line 23 of a Recycle() operation. Also, $t^* < t_3 < t_2$.

So at $t_3$, $p_w$ performs line 23 of rc. This is a contradiction because $p_w$ does not perform line 23 of rc during $(t^*, t_2)$. □

Claim 9. Let up be an update operation, and $w_{i,j}$ a node in reachable(rold(up)) \ reachable(rnew(up)). Suppose $t$ is a point in time after lin(up) and $r$ the tree root at $t$. Then $w_{i,j} ∉ reachable(r)$.

Proof. First we show that start($w_{i,j}$) < lin(up).

Let up$^*$ be the update operation, in which $w_{i,j}$ is dequeued from free. (By Observation 3, there is only one such update operation.) Suppose r$^*$ is the tree root that $p_w$ writes to R at lin(up$^*$). Then $w_{i,j} ∈ reachable(r^*)$. Let $t_1$ be a point in time before lin(up$^*$) and $r_1$ the tree root at $t_1$. Since $w_{i,j}$ is dequeued from free during up$^*$, $w_{i,j} ∉ reachable(r_1)$. Therefore, by the fact that $w_{i,j} ∈ reachable(rold(up))$ and rold(up) is the root of the tree just before lin(up), lin(up$^*$) < lin(up). Since start($w_{i,j}$) < lin(up$^*$), start($w_{i,j}$) < lin(up).

By Observation 3, each node is only dequeued once from free. Since start($w_{i,j}$) < lin(up), $w_{i,j}$ is not dequeued from free after lin(up). Also, $w_{i,j} ∉ reachable(rnew(up))$. Therefore, by Observation 6, for any point in time $t'$ after lin(up), $w_{i,j}$ is not reachable from $r'$, where $r'$ is the tree root at $t'$. Hence, $w_{i,j} ∉ reachable(r)$. □

Claim 10. Let up and up$'$ be two update operations, such that $w_{i,j} ∈ reachable(rold(up)) \ reachable(rnew(up))$ and $w_{i,j} ∈ reachable(rold(up')) \ reachable(rnew(up'))$. Then up = up$'$.

Proof. For the purpose of contradiction, assume up $≠$ up$'$. Without loss of generality assume up happens before up$'$. Since $w_{i,j} ∈ reachable(rold(up')) \ reachable(rnew(up'))$, $w_{i,j} ∈ reachable(rold(up'))$. Also, rold(up$'$) is the root of the tree just before lin(up$'$) > lin(up). By Claim 9, $w_{i,j}$ is not reachable rold(up$'$). This is a contradiction. □

We define the following predicate $A(i, j)$ for each $i \in \{0, \ldots, \lambda - 1\}$ and each integer $j ≥ 0$:

Predicate $A(i, j)$: For every integer $j' \in \{0, \ldots, j\}$, while start($w_{i,j'}$) is in free, outdated.$i$.val = false.
Observation 11. Let \( i \in \{0, \ldots, \lambda - 1\} \) and \( j \) be an integer greater than or equal to 0. Then \( \text{start}(w_{i,j}) < \text{start}(w_{i,j+1}) \).

Proof. If \( j > 0 \), it follows from the proof of Observation 3 that \( p_w \) has enqueued \( w_{i,j} \) into \textit{free} before it enqueues \( w_{i,j+1} \) into \textit{free}. Also, \( w_{i,0} \) is in \textit{free} from the start of the execution and thus is dequeued before any other \( w_{i,j'} \), for \( j' > 0 \). Therefore, \( \text{start}(w_{i,j}) < \text{start}(w_{i,j+1}) \).

By definition and Observation 11, \( \text{end}(w_{i,j+1}) > \text{start}(w_{i,j+1}) > \text{start}(w_{i,j}) \). Now we prove that \( \text{end}(w_{i,j}) < \text{start}(w_{i,j+1}) \).

Claim 12. Let \( i \in \{0, \ldots, \lambda - 1\} \), \( j \) be an integer greater than or equal to 0 and \( j' \) an integer greater than \( j \). If \( A(i,j) \) is true, then \( \text{end}(w_{i,j}) < \text{start}(w_{i,j'}) \).

Proof. By Observation 11, \( \text{start}(w_{i,j+1}) \leq \text{start}(w_{i,j'}) \). Thus we only need to prove that \( \text{end}(w_{i,j}) < \text{start}(w_{i,j+1}) \).

To derive a contradiction, assume there is some smallest integer \( k \), such that \( \text{end}(w_{i,k}) \geq \text{start}(w_{i,k+1}) \). If \( k > j \), then \( \text{end}(w_{i,j}) < \text{start}(w_{i,j+1}) \).

Therefore, we only need to consider the case that \( k \leq j \).

Recall that \( p_w \) only enqueues a node into \textit{free} in line 25 of a \texttt{Recycle()} operation. Before \( p_w \) enqueues \( w_{i,\text{last}[i]} \) into \textit{free} in line 25, it first observes that the value of \textit{outdated}[i].\textit{val} is true in line 21, and then increases \textit{last}[i] by one in line 24. The initial value of \textit{last}[i] is 0, and \( w_{i,k+1} \) is in \textit{free} just before \( \text{start}(w_{i,k+1}) \). Therefore, before \( \text{start}(w_{i,k+1}) \), \( p_w \) has performed \texttt{free.Enq}(w_{i,\text{last}[i]}) at least \( k \) times. By our assumption about \( k \), \( \text{start}(w_{i,0}) < \text{end}(w_{i,0}) < \ldots < \text{start}(w_{i,k-1}) < \text{end}(w_{i,k-1}) < \text{start}(w_{i,k}) < \text{start}(w_{i,k+1}) \).

First we prove that \( p_w \) does not perform \texttt{free.Enq}(w_{i,\text{last}[i]}) during \((\text{end}(w_{i,k'}), \text{start}(w_{i,k'+1}))\), for any \( k' \), where \( 0 \leq k' < k \). Process \( p_w \) has performed \texttt{free.Enq}(w_{i,\text{last}[i]}) at least \( k' + 1 \) times by \( \text{end}(w_{i,k'}) \). Therefore, \( w_{i,k'+1} \) is in \textit{free} after \( \text{end}(w_{i,k'}) \) and before \( \text{start}(w_{i,k'+1}) \). Hence, by Predicate \( A(i,j) \), \textit{outdated}[i].\textit{val} = false throughout \((\text{end}(w_{i,k'}), \text{start}(w_{i,k'+1}))\). Let \( rc \) be the \texttt{Recycle()} operation, such that \( \text{inv}(rc) < \text{end}(w_{i,k'}) < \text{rsp}(rc) \). For the purpose of contradiction, assume \( p_w \) enqueues a node \( w_{i,\ell} \) into \textit{free} at a point in time \( t \) during \((\text{end}(w_{i,k'}), \text{start}(w_{i,k'+1}))\) in a \texttt{Recycle()} operation \( rc' \). In a \texttt{Recycle()} operation, for each integer \( i' \in \{0, \ldots, \lambda\} \), \( p_w \) performs \texttt{free.Enq}(w_{i',\text{last}[i']}) at most once. During \( rc \), \( p_w \) performs \texttt{free.Enq}(w_{i,\text{last}[i]}) at \( \text{end}(w_{i,k'}) \). Hence, \( \text{rsp}(rc) < \text{inv}(rc') \). After \( \text{inv}(rc') \) and before \( t \), \( p_w \) performs line 21 of \( rc' \) and observes that the value of \textit{outdated}[i].\textit{val} is true.
However, $\text{outdated}[i].\text{val}$ is false throughout $(\text{end}(w_{i,k'}), \text{start}(w_{i,k'+1}))$. Since $\text{end}(w_{i,k'}) < \text{rsp}(rc) < \text{inv}(rc')$ and $t < \text{start}(w_{i,k'+1})$, $\text{outdated}[i].\text{val}$ is false throughout $[\text{inv}(rc'), t]$. This is a contradiction.

As shown above, $p_w$ does not perform $\text{free}.\text{Enq}(w_{i,\text{last}[i]})$ during $(\text{end}(w_{i,k'}), \text{start}(w_{i,k'+1}))$, for any $k$, where $0 \leq k' < k$. Also, by Predicate $A(i,j)$, $\text{outdated}[i].\text{val} = false$ before $\text{start}(w_{i,0})$. Therefore, $p_w$ does not perform $\text{free}.\text{Enq}(w_{i,\text{last}[i]})$ before $\text{start}(w_{i,0})$. (Recall that before $p_w$ enqueues $w_{i,\text{last}[i]}$ into free, it observes that the value of $\text{outdated}[i].\text{val}$ is true.) In addition, by definition of end and our assumption about $k$, $p_w$ does not perform $\text{free}.\text{Enq}(w_{i,\text{last}[i]})$ throughout $[\text{start}(w_{i,k}), \text{start}(w_{i,k+1})]$. Moreover, by definition of end, $p_w$ does not enqueue $w_{i,\text{last}[i]}$ into free throughout $[\text{start}(w_{i,k'}), \text{end}(w_{i,k'})]$. Therefore, before $\text{start}(w_{i,k'+1})$, $p_w$ has performed $\text{free}.\text{Enq}(w_{i,\text{last}[i]})$, $k-1$ times, once at each $\text{end}(w_{i,k'})$. This is a contradiction.

It follows from the proof of Claim 12 that at $\text{end}(w_{i,j'})$, where $j' \leq j$, $p_w$ enqueues $w_{i,j'+1}$ into free.

**Observation 13.** Let $i \in \{0, \ldots, \lambda - 1\}$ and $j$ be an integer greater than or equal to 0. If $A(i,j)$ is true, then at $\text{end}(w_{i,j})$, $p_w$ enqueues $w_{i,j+1}$ into free.

**Claim 14.** Let $i \in \{0, \ldots, \lambda - 1\}$ and $j$ be an integer greater than or equal to 0. Suppose $rc$ is the $\text{Recycle()}$ operation such that $\text{inv}(rc) < \text{end}(w_{i,j}) < \text{rsp}(rc)$. Suppose $t$ is the first point in time after $\text{start}(w_{i,j})$, such that $\text{outdated}[i].\text{val}_t = true$. If $A(i,j)$ is true, at $t$, $p_w$ sets $\text{outdated}[i].\text{val}$ to true in an update operation $up$, such that $w_{i,j} \in \text{reachable}(\text{rold}(up)) \setminus \text{reachable}(\text{rnew}(up))$ and $\text{lin}(up) < \text{inv}(rc)$.

**Proof.** Let $k$ be an integer in $\{0, \ldots, j\}$. Let $rc_k$ be the $\text{Recycle()}$ operation, such that $\text{inv}(rc) < \text{end}(w_{i,k}) < \text{rsp}(rc)$. Suppose $t^*_k$ is the point in time at which $p_w$ performs line 21 of $rc_k$ and observes that the value of $\text{outdated}[i]$ is $y_k$. Let $t_k$ be the first point in time after $\text{start}(w_{i,k})$, such that $\text{outdated}[i].\text{val}_{t_k} = true$.

**$\text{start}(w_{i,k}) < t^*_k$:** For the purpose of contradiction, assume $\text{start}(w_{i,k}) \geq t^*_k$. We show that $w_{i,k}$ is in free at some point before $t^*_k$ and then arrive at a contradiction.

If $k = 0$, then initially $w_{i,0}$ is in free and thus, $w_{i,0}$ is in free before $t^*_k$. Otherwise, $k > 0$. Before $\text{start}(w_{i,k})$, $p_w$ performs $\text{free}.\text{Enq}(w_{i,k})$ in line 25
of a \texttt{Recycle()} operation \(rc'\). (Recall that \(p_w\) only enqueues a node into \texttt{free} in line 25 of a \texttt{Recycle()} operation.) In a \texttt{Recycle()} operation, for each integer \(i' \in \{0, \ldots, \lambda\}\), \(p_w\) performs \texttt{free.Enq}(\(w_{i',\text{last}[i']}\)) at most once. During \(rc_k\), \(p_w\) performs \texttt{free.Enq}(\(w_{i,\text{last}[i]}\)) at most once. Since \(\text{start}(w_{i,k}) < \text{end}(w_{i,k})\), \(\text{rsp}(rc') < \text{inv}(rc_k)\). Thus, \(w_{i,k}\) is enqueued into \texttt{free} before \(t^*_k\).

Since \(\text{start}(w_{i,k})\) is after \(t^*_k\), \(w_{i,k}\) is in \texttt{free} at \(t^*_k\). Therefore, by Predicate \(A(i,j), \text{outdated}[i].\text{val}_{t^*_k} = \text{false}\). Thus, \(p_w\) does not perform \texttt{free.Enq}(\(w_{i,\text{last}[i]}\)) in line 25 of \(rc_k\). This is a contradiction.

Next, we prove by induction that at \(t_k\), \(p_w\) sets \(\text{outdated}[i].\text{val}\) to true in an update operation \(up_k\), such that \(w_{i,k} \in \text{reachable}(\text{rold}(up_k)) \setminus \text{reachable}(\text{rnew}(up_k))\) and \(\text{lin}(up_k) < t^*_k\).

First, we show this for \(k = 0\). By Predicate \(A(i,j)\), at \(\text{start}(w_{i,0})\), \(\text{outdated}[i].\text{val} = \text{false}\). Also, since \(\text{inv}(rc_0) < \text{end}(w_{i,0}) < \text{rsp}(rc_0)\), \(y_0.\text{val} = \text{true}\). As discussed above \(\text{start}(w_{i,0}) < t^*_0\). Hence, \(t_0 < t^*_0\).

By Claim 8, at \(t_0\), \(p_w\) sets \(\text{outdated}[i].\text{val}\) to true in line 6 of an update operation \(up_0\). Hence, there exists an integer \(k' \geq 0\), such that \(w_{i,k'} \in \text{reachable}(\text{rold}(up_0)) \setminus \text{reachable}(\text{rnew}(up_0))\). Since \(t_0 < t^*_0\) and \(\text{lin}(up_0) < t_0\), \(\text{lin}(up_0) < t^*_0\). By Claim 12, for every integer \(j' \geq 1\), \(\text{start}(w_{i,j'}) > \text{end}(w_{i,0})\).

Therefore, before \(\text{end}(w_{i,0})\), only \(w_{i,0}\) is the only node with index \(i\) that is dequeued from \texttt{free}, and thus only \(w_{i,0}\) can be reachable from a tree root. Hence, \(w_{i,0} \in \text{reachable}(\text{rold}(up_0)) \setminus \text{reachable}(\text{rnew}(up_0))\).

Now we assume that for every integer \(k' < k\), at \(t_{k'}\), \(p_w\) sets \(\text{outdated}[i].\text{val}\) to true in an update operation \(up_{k'}\), such that \(w_{i,k'} \in \text{reachable}(\text{rold}(up_{k'})) \setminus \text{reachable}(\text{rnew}(up_{k'}))\) and \(\text{lin}(up_{k'}) < t^*_{k'}\). Then we prove at \(t_k\), \(p_w\) sets \(\text{outdated}[i].\text{val}\) to true in an update operation \(up_k\), such that \(w_{i,k} \in \text{reachable}(\text{rold}(up_k)) \setminus \text{reachable}(\text{rnew}(up_k))\) and \(\text{lin}(up_k) < t^*_k\).

By Predicate \(A(i,j)\), at \(\text{start}(w_{i,k})\), \(\text{outdated}[i].\text{val} = \text{false}\). Also, since \(\text{inv}(rc_k) < \text{end}(w_{i,k}) < \text{rsp}(rc_k)\), \(y_k.\text{val} = \text{true}\). As discussed above \(\text{start}(w_{i,k}) < t^*_k\). Hence, \(t_k < t^*_k\).

By Claim 8, at \(t_k\), \(p_w\) sets \(\text{outdated}[i].\text{val}\) to true in line 6 of an update operation \(up_k\). Hence, there exists an integer \(j' \geq 0\), such that \(w_{i,j'} \in \text{reachable}(\text{rold}(up_k)) \setminus \text{reachable}(\text{rnew}(up_k))\). Since \(t_k < t^*_k\) and \(\text{lin}(up_k) < t_k\), \(\text{lin}(up_k) < t^*_k\). By Claim 12, for every integer \(\ell > k\), \(\text{start}(w_{i,\ell}) > \text{end}(w_{i,k})\).

Therefore, before \(\text{end}(w_{i,k})\), \(w_{i,\ell}\) is not dequeued from \texttt{free}, and thus \(w_{i,\ell} \notin \text{reachable}(\text{rold}(up_k))\). By Claim 12, for every integer \(\ell < k\), \(\text{end}(w_{i,\ell}) < \text{start}(w_{i,k})\).

Since \(\text{start}(w_{i,k}) < t_k\) and throughout \([\text{lin}(up_k), t_k]\), \(p_w\) does not dequeue a node from \texttt{free}, \(\text{start}(w_{i,k}) < \text{lin}(up_k)\).
Therefore, \( \text{end}(w_{i,\ell}) < \text{lin}(up_k) \). By the inductive hypothesis, \( \text{lin}(up) < t^*_w < \text{end}(w_{i,\ell}) < \text{lin}(up_k) \). Hence, by Claim 9, \( w_{i,\ell} \not\in \text{reachable}(\text{rold}(up_k)) \). Thus, \( w_{i,k} \in \text{reachable}(\text{rold}(up_k)) \setminus \text{reachable}(\text{rnew}(up_k)) \).

Finally, we prove that \( \text{lin}(up) < \text{inv}(rc) \).

\textbf{lin}(up) < \text{inv}(rc) \ Let \( t_1 \) be the point in time at which \( p_w \) sets \( \text{oustdated}[i] \) to \((\text{true}, 1 - \text{sw})\) in line 6 of \( up \). During \( up \), \( p_w \) contributes to an ongoing \text{Recycle()}\ call after \( t_1 \). Since \( \text{lin}(up) < t_1 \) and \( \text{lin}(up) < t^*, t_1 < t^* \). We will prove that \( t_1 < \text{inv}(rc) \). Then since \( \text{lin}(up) < t_1, \text{lin}(up) < \text{inv}(rc) \).

For the purpose of contradiction, assume \( \text{inv}(rc) \leq t_1 \). Let \( x \) be the value of \( \text{sw} \) at \( \text{inv}(rc) \). Process \( p_w \) changes \( \text{sw} \) only when there is not any pending \text{Recycle()}\ call. Therefore, \( \text{sw} = x \) throughout \([\text{inv}(rc), \text{rsp}(rc)]\). Since \( \text{inv}(rc) < \text{end}(w_{i,j}) < \text{rsp}(rc) \), \( p_w \) observes that the value of \( \text{oustdated}[i].\text{val} \) is \((\text{true}, x)\) at \( t^* \). In \( rc \), process \( p_w \) does not change \( \text{oustdated}[i].\text{val} \) during \([\text{inv}(rc), t^*] \). Also, if \( p_w \) sets \( \text{oustdated}[i].\text{val} \) to true in line 6 of an update operation during \([\text{inv}(rc), t^*] \), then \( p_w \) sets \( \text{oustdated}[i].\text{sw} \) to \( 1 - x \). Since \( \text{inv}(rc) < t_1 < t^* \), the value of \( \text{oustdated}[i] \) is \((\text{true}, 1 - x)\) at \( t^* \). This is a contradiction. \( \square \)

\textbf{Claim 15.} Let \( i \in \{0, \ldots, \lambda - 1\} \) and \( j \) be an integer greater than or equal to 0. Then \( A(i, j) \) is true.

\textbf{Proof.} We show by induction that for each integer \( k \geq 0 \), \( A(i, k) \) is true.

Initially, \( w_{i,0} \) is in \text{free} and \( \text{oustdated}[i].\text{val} = \text{false} \). Before \( \text{start}(w_{i,0}) \), for each integer \( j' \geq 0 \), \( w_{i,j'} \) is not dequeued from \text{free}. Therefore, for each tree root \( r \) that is initialized before \( \text{start}(w_{i,0}) \), \( w_{i,j'} \not\in \text{reachable}(r) \). Hence, before \( \text{start}(w_{i,0}) \), \( p_w \) does not change \( \text{oustdated}[i].\text{val} \) in line 6 of an update operation. Therefore, by Claim 8, \( p_w \) does not change \( \text{oustdated}[i].\text{val} \) before \( \text{start}(w_{i,0}) \). Hence, while \( w_{i,0} \) is in \text{free}, \( \text{oustdated}[i].\text{val} = \text{false} \). Thus \( A(i, 0) \) is true.

Now we assume that \( A(i, k - 1) \) is true and then prove \( A(i, k) \) is also true. By definition of Predicate \( A(i, k - 1) \), we only need to show that while \( w_{i,k} \) is in \text{free}, \( \text{oustdated}[i].\text{val} = \text{false} \).

Suppose \( rc \) is the \text{Recycle()}\ operation, such that \( \text{inv}(rc) < \text{end}(w_{i,k-1}) < \text{rsp}(rc) \). We know that for each integer \( k' < k \), while \( w_{i,k'} \) is in \text{free}, \( \text{oustdated}[i].\text{val} = \text{false} \). Therefore, by Claim 14, there exists an update operation \( up \), such that \( w_{i,k-1} \in \text{reachable}(\text{rold}(up)) \setminus \text{reachable}(\text{rnew}(up)) \) and \( \text{lin}(up) < \text{inv}(rc) \). Let \( t^* < \text{end}(w_{i,k-1}) \) be the point in time at which
$p_w$ performs line 23 of $rc$ and sets $\text{outdated}[i].\text{val}$ to false. By Claim 12, $t^* < \text{end}(w_{i,k-1}) < \text{start}(w_{i,k})$. Also by Observation 13, $p_w$ enqueues $w_{i,k}$ into $\text{free}$ at $\text{end}(w_{i,k-1})$. We will prove that $\text{outdated}[i].\text{val}$ remains false throughout $[t^*, \text{start}(w_{i,k})]$. Therefore, while $w_{i,k}$ is in $\text{free}$, $\text{outdated}[i].\text{val} = \text{false}$.

For the purpose of contradiction, assume $p_w$ changes $\text{outdated}[i].\text{val}$ for the first time after $t^*$ at a point in time $t_2$, such that $t_2 < \text{start}(w_{i,k})$. By Claim 8, at $t_2$, $p_w$ changes $\text{outdated}[i].\text{val}$ in an update operation $up'$. Therefore, there exists an integer $j' \geq 0$, such that $w_{i,j'} \in \text{reachable}(r\text{old}(up')) \setminus \text{reachable}(r\text{new}(up'))$. Since $t_2 < \text{start}(w_{i,k})$, for each integer $\ell \geq k$, $w_{i,\ell}$ is not dequeued from $\text{free}$ before $t_2$ and thus, $w_{i,\ell} \notin \text{reachable}(r\text{old}(up'))$. Also, by Claims 12 and 14, for each integer $\ell < k$, there exists an update operation $up_{\ell}$, such that $w_{i,\ell} \in \text{reachable}(r\text{old}(up_{\ell})) \setminus \text{reachable}(r\text{new}(up_{\ell}))$ and $\text{lin}(up_{\ell}) < t^*$. Hence, by Claim 9, $w_{i,\ell} \notin \text{reachable}(r\text{old}(up'))$. Hence, such index $j'$ does not exist. This is a contradiction. \hfill \Box

### 3.3.1 Proof of Lemma 4

**Claim 16.** Let $up$ be an update operation, and $w_{i,j}$ a node in $\text{reachable}(r\text{old}(up)) \setminus \text{reachable}(r\text{new}(up))$. Suppose $t$ is the point in time at which $p_w$ sets $\text{outdated}[i].\text{val}$ to true in line 6 of $up$. The value of $\text{outdated}[i].\text{val}$ is false just before $t$.

**Proof.** By Claim 15, $\text{outdated}[i].\text{val} = \text{false}$ at $\text{start}(w_{i,j})$. Suppose $t_1$ is the first point in time after $\text{start}(w_{i,j})$, such that $\text{outdated}[i].\text{val}_{t_1} = \text{true}$. By Claims 14 and 15, at $t_1$, $p_w$ sets $\text{outdated}[i].\text{val}$ to true in an update operation $up'$ such that $w_{i,j} \in \text{reachable}(r\text{old}(up')) \setminus \text{reachable}(r\text{new}(up'))$. By Claim 10, $up = up'$. Process $p_w$ sets $\text{outdated}[i].\text{val}$ to true for the first time after $\text{start}(w_{i,j})$ at $t$. Hence, just before $t$, $\text{outdated}[i].\text{val} = \text{false}$.

**Claim 17.** Let $rc$ be a $\text{Recycle()}$ operation and $i$ an integer in $\{0, \ldots, \lambda-1\}$. Suppose the value of $sw$ is $x$ at $\text{inv}(rc)$. At $\text{inv}(rc)$, if $\text{outdated}[i].\text{val} = \text{true}$, then $\text{outdated}[i].sw = x$.

**Proof.** Process $p_w$ changes $sw$ just before each $\text{Recycle()}$ call. Also, the only lines in the pseudocode in which $p_w$ sets $\text{outdated}[i].\text{val}$ to true are line 6 of an update operation and line 26 of a $\text{Recycle()}$ operation. We consider two cases:

**Case 1 (rc is the first Recycle() call by $p_w$):**
Initially, $\text{outdated}[i].\text{val} = \text{false}$. Since $rc$ is the first $\text{Recycle()}$ call, if
outdated[i].val = true at inv(rc), then \( p_w \) has set outdated[i].val to true in line 6 of an update operation before inv(rc). The initial value of sw is 0 and \( p_w \) changes sw for the first time to \( 1 = x \) just before inv(rc). Therefore, if \( p_w \) sets outdated[i].val to true before inv(rc), then outdated[i].sw = 1 − sw = 1 = x.

**Case 2** (rc is not the first Recycle() call by \( p_w \)):
Let rc' be the previous Recycle() call by \( p_w \) before rc. Let \( t^* \) be the point in time at which \( p_w \) changes sw to \( x \) just before inv(rc). Throughout \([inv(rc'), t^*]\), sw is 1 − \( x \). Let \( t_1 \) be the point in time at which \( p_w \) performs line 21 of rc' and observes that the value of outdated[i] is \( y \).

For the purpose of contradiction, assume outdated[i]/inv(rc) = (true, 1 − \( x \)). Let \( t_2 \) be the last point in time before inv(rc), such that at \( t_2 \), \( p_w \) sets outdated[i] to (true, 1 − \( x \)). Therefore, throughout \((t_2, inv(rc)]\), outdated[i] = (true, 1 − \( x \)). During \([inv(rc'), t^*]\), \( p_w \) does not set outdated[i].sw to 1 − \( x \).

(See lines 6 and 26.) Also, \( p_w \) does not perform any operation throughout \((t^*, inv(rc))\). Therefore, \( t_2 < inv(rc') \). Hence, since outdated[i] = (true, 1 − \( x \)) throughout \((t_2, inv(rc)]\), \( y = (true, 1 − x) \). Thus, \( p_w \) changes either outdated[i] to (false, 1 − \( x \)) in line 23 of rc' or to (true, \( x \)) in line 26 of rc'. This is a contradiction because outdated[i] = (true, 1 − \( x \)) throughout \((t_2, rsp(rc'))\).

**Claim 18.** Let rc be a Recycle() operation and \( i \) an integer in \( \{0, \ldots , \lambda − 1\} \). Suppose \( r \) is the tree root at inv(rc). If for each integer \( j \geq 0 \), \( w_{i,j} \) is not in free_{inv(rc)} and \( w_{i,j} \not\in \text{reachable}(r) \), then outdated[i].val_{inv(rc)} = true.

**Proof.** Initially \( w_{i,0} \) is in free. Since \( w_{i,0} \) is not in free_{inv(rc)}, \( w_{i,0} \) is dequeued from free before inv(rc). Let integer \( k \geq 0 \) be the greatest integer, such that \( w_{i,k} \) is dequeued from free before inv(rc). The only line in the pseudocode in which \( p_w \) dequeues a node from free is line 4 of an update operation. Suppose up is the update operation, that dequeues \( w_{i,k} \) from free. During up, after start(\( w_{i,k} \)), \( p_w \) writes \( rnew(up) \) to R and then may start a new Recycle() method. Therefore, since start(\( w_{i,k} \)) < inv(rc), lin(up) < inv(rc).

By Claim 15, outdated[i].val = false at start(\( w_{i,k} \)). Let \( t^* \) be the first point in time after start(\( w_{i,k} \)), such that outdated[i].val_{t*} = true. If \( t^* \) does not exist, by Observation 6 and Observation 3, \( w_{i,k} \in \text{reachable}(r) \).

This is a contradiction. Thus, \( t^* \) exists. By Claims 14 and 15, at \( t^* \), \( p_w \) sets outdated[i].val to true in an update operation up', such that \( w_{i,k} \in \text{reachable}(rold(up'))\) \( \setminus \text{reachable}(rnew(up')) \). Since \( w_{i,k} \) is dequeued from free in up, \( w_{i,k} \in \text{reachable}(rnew(up)) \). Let \( t_1 \) be a point in time in
We prove that outdated[i].val = true throughout (t∗, inv(rc)]. For the purpose of contradiction, assume an earliest point in time t2, where t∗ < t2 ≤ inv(rc), such that outdated[i].valt2 = false. By Observation 7, at t2, pw sets outdated[i].val to false in line 23 of a Recycle() operation rc′. Since t2 ≤ inv(rc), rsp(rc′) < inv(rc). After t2 and before rsp(rc′), pw enqueues w_{i, last[i]} into free at a point in time t3 < rsp(rc′) < inv(rc). Since start(w_{i,k}) < t3, pw enqueues w_{i,k′} at t3, where k′ > k. Hence, w_{i,k′} either is in free_{inv(rc)}, or after t3 and before inv(rc) is dequeued from free. The former contradicts the fact that for each integer j ≥ 0, w_{i,j} is not in free_{inv(rc)}, and the latter contradicts the fact that k is the greatest integer, such that w_{i,k} is dequeued from free before inv(rc). Hence, outdated[i].val_{inv(rc)} = true.

\[ \text{Proof of Lemma 4.} \]\] Let rc_i be the i-th Recycle() operation by pw. First, we prove by induction that for each i, |free| at inv(rc_i) is greater than c_{rb}n\Delta \log \Delta.

Before inv(rc_0), pw performs n\Delta update operations, during each of which pw dequeues at most c_{rb}\Delta nodes from free. (Recall that c_{rb} is the constant from Inequality 3.2.) Initially, there are 4c_{rb}n\Delta \log \Delta elements in free. Therefore, at inv(rc_0), |free| is greater than c_{rb}n\Delta \log \Delta.

Now we prove that if |free_{inv(rc_i)}| > c_{rb}n\Delta \log \Delta, then |free_{inv(rc_{i+1})}| > c_{rb}n\Delta \log \Delta.

Suppose r is the tree root at inv(rc_i). Let U be the set of indices j, such that outdated[j] = (true, x) at inv(rc_i), where x is sw_{inv(rc_i)}. Since our algorithm has bounded capacity \Delta, |reachable(r)| ≤ \Delta. Therefore, by Claim 17 and Claim 18, at inv(rc_i), |U| ≥ \lambda - \Delta - |free_{inv(rc_i)}|.

The only lines in the pseudocode in which pw may change outdated[j] (0 ≤ j < \lambda) are line 6 of an Update() operation, and lines 23 and 26 of a Recycle() operation. By Claim 16, after inv(rc_i), for each j ∈ U, pw does not change outdated[j] until it performs line 23 or 26 of a Recycle() operation.

During rc_i, in each iteration of the for-loop in lines 15-19, pw reads r from M[p] (0 ≤ p < n), and then sets protected[mem(v)] to true, for each v ∈ reachable(r). Since |reachable(r)| ≤ \Delta, at most n\Delta entries in protected are true. Next, pw iterates over every j, 0 ≤ j < \lambda, in the for-loop in lines 20-26. If outdated[j] = (true, x) and protected[j] = false, pw enqueues a node
into free in line 25. At most \( n\Delta \) entries in protected are true and \(|U| \geq \lambda - \Delta - |\text{free}_{\text{inv}(rc_i)}|\). Hence, at least \((\lambda - \Delta - |\text{free}_{\text{inv}(rc_i)}|) - n\Delta\) nodes are enqueued into free during \( rc_i \). Also, \( p_w \) calls a new \texttt{Recycle()} method every \( n\Delta \) update operations. During each update, \( p_w \) dequeues at most \( c_{rb} \log \Delta \) nodes from free. Therefore, at most \( c_{rb} n \Delta \log \Delta \) nodes are dequeued from free during \([\text{inv}(rc_i), \text{inv}(rc_{i+1})]\). Hence,

\[
|\text{free}_{\text{inv}(rc_{i+1})}| \geq |\text{free}_{\text{inv}(rc_i)}| - c_{rb} n \Delta \log \Delta + (\lambda - \Delta - |\text{free}_{\text{inv}(rc_i)}| - n\Delta)
\]

\[
= \lambda - \Delta - n\Delta - c_{rb} n \Delta \log \Delta
\]

\[
= 4c_{rb} n \Delta \log \Delta - \Delta - n\Delta - c_{rb} n \Delta \log \Delta
\]

\[
> c_{rb} n \Delta \log \Delta.
\]

Let \( t \) be a point in time. As discussed above, if \( t < \text{inv}(rc_0) \), then \(|\text{free}_t| > 0\). Otherwise \( t > \text{inv}(rc_0) \). Suppose \( rc \) is the last \texttt{Recycle()} call such that \( \text{inv}(rc) < t \). Process \( p_w \) calls a new \texttt{Recycle()} method every \( n\Delta \) update operations. Therefore, during \([\text{inv}(rc), t]\), \( p_w \) calls at most \( n\Delta \) update operations. Since at most \( c_{rb} \log \Delta \) nodes are dequeued from free in an update operation, at most \( c_{rb} n \Delta \log \Delta \) nodes are dequeued from free during \([\text{inv}(rc), t]\). Hence, by the fact that \( |\text{free}_{\text{inv}(rc)}| > c_{rb} n \Delta \log \Delta, |\text{free}_t| > 0 \).

3.3.2 Proof of Lemma 5

Proof. For the purpose of contradiction, assume \( \text{end}(w_{i,j}) \leq t \). Let \( rc \) be the \texttt{Recycle()} operation, such that \( \text{inv}(rc) < \text{end}(w_{i,j}) < \text{rsp}(rc) \). Therefore, \( p_w \) performs \texttt{free.Enq(w_{i,\text{last}[i]})} in line 25 of \( rc \).

By Claims 14 and 15, there exists an update operation \( up \), such that \( w_{i,j} \in \text{reachable}(\text{rold}(up)) \setminus \text{reachable}(\text{rneu}(up)) \) and \( \text{lin}(up) < \text{inv}(rc) \). Let \( r_p \) be the tree root that \( p \) copies from \( R \) to \( M[p] \) at \( \text{lin}(oq) \). Since \( p \) visits \( w_{i,j} \) in \( oq \), \( w_{i,j} \in \text{reachable}(r_p) \). Thus, by Claim 9, \( \text{lin}(oq) < \text{lin}(up) < \text{inv}(rc) \). The value of \( M[p] \) is \( r_p \) throughout \( (\text{lin}(oq), \text{rsp}(oq)) \). Since \( t < \text{rsp}(oq) \), \( M[p] = r_p \) throughout \( (\text{lin}(oq), t] \). In addition, \( \text{lin}(oq) < \text{inv}(rc) \) and \( \text{end}(w_{i,j}) \leq t \). Therefore, \( M[p] = r_p \) throughout \([\text{inv}(rc), \text{end}(w_{i,j})]\).

During \( rc \), before \( \text{end}(w_{i,j}) \), \( p_w \) reads \( r_p \) from \( M[p] \) in line 16 and for each \( u \in \text{reachable}(r_p) \) sets \( \text{protected}[\text{ind}(u)] \) to true in line 19. Since \( w_{i,j} \in \text{reachable}(r_p) \), \( p_w \) sets \( \text{protected}[i] \) to true at a point in time \( t_{prot} \). Clearly, throughout \( (t_{prot}, \text{end}(w_{i,j})], \text{protected}[i] = \text{true} \). Therefore, \( p_w \) reads true from
protected[i] in line 22 of rc, and thus does not perform free.Enq(w_i, last[i]) in line 25 of rc. This is a contradiction. \qed

3.4 Analysis

In this section, first, we analyze the time complexity of method Recycle() of Algorithm 1 and then prove Theorem 2.

Lemma 19. The time complexity of a Recycle() operation is $O(n \Delta \log \Delta)$.

Proof. Suppose rc is a Recycle() operation. First, $p_w$ creates a Boolean array protected of size $\lambda$ in line 14 of rc. The time complexity of this operation is $O(\lambda)$. Then for each process $p \in \{0, \ldots, n-1\}$, $p_w$ reads $r_p$ by performing M.Read($p$), and then traverses through every node $v \in \text{reachable}(r_p)$. As discussed in Section 1.3, performing a Read() operation on a destination array takes $O(1)$ time. Since our object has bounded capacity $\Delta$, $|\text{reachable}(r_p)| \leq \Delta$. Hence, $p_w$ can traverse through all the reachable nodes from $r_p$ by performing a depth-first-search algorithm in $O(\Delta)$ time. Therefore, the time complexity of the for-loop in lines 15-19 is $O(n \Delta)$. The for-loop in lines 20-26 iterates $\lambda$ times. In each iteration, $p_w$ performs a constant number of local steps. Hence, the time complexity of this for-loop is $O(\lambda)$. Thus, the total time complexity of rc is $O(\lambda + n \Delta)$. Since $\lambda = 4c r b n \Delta \log \Delta$, the total time complexity of a Recycle() operation is $O(n \Delta \log \Delta)$. \qed

Proof of Theorem 2. We proved in Lemma 4 that $|\text{free}|$ at any point in time is greater than 0. Therefore, as discussed in Section 3.3, Algorithm 1 is linearizable. We also showed in Lemma 5, if $p_w$ enqueues $w_{i,j+1}$ into free at $t$, then $w_{i,j}$ cannot be accessed by any process after $t$. Thus, Algorithm 2 is also linearizable. In the following, we analyze the time complexity of Algorithm 2 and we show that this algorithm uses a bounded number of base objects.

Time complexity. As discussed in Section 1.3, each operation on a destination array takes constant time. Also, since at any point in time the size of the tree is at most $\Delta$, performing a query operation on a tree root by taking the exact same steps as in the sequential implementation in [15] takes $O(\log \Delta)$ time. Hence, the time complexity of a query operation is $O(\log \Delta)$. Let up be an update operation. In up, when $p_w$ wants to create a node, it dequeues a node $w_i$ from free and fills the fields of $w_i$, instead. Since $p_w$ creates $O(\log \Delta)$
nodes in each update operation, line 30 of \textit{up} takes $O(\log \Delta)$ time. Suppose $S = \text{reachable}(\text{old}(\text{up})) \setminus \text{reachable}(\text{new}(\text{up}))$. During \textit{up}, for each $v \in S$, $p_w$ changes the value of \texttt{outdated}[v.ind] in line 32. The set $S$ is the parent closure of the nodes that get modified during \textit{up}. Therefore $|S| \in O(\log \Delta)$, and thus, the time complexity of computing $S$ is $O(\log \Delta)$. Also, as discussed above, $p_w$ contributes $O(\log \Delta)$ steps to an ongoing \texttt{Recycle()} call during each update operation. Therefore, the time complexity of each update is also $O(\log \Delta)$.

**Number of base objects and bit complexity** In our implementation, we store $\lambda = O(n \Delta \log \Delta)$ nodes, a register $R$, and a destination array $M$.

First, we determine what we store in each object. Suppose $D_{nl} = \{w_0, \ldots, w_{\lambda-1}\}$ and $D_{ind} = \{0, \ldots, \lambda - 1\}$. Then

(a) $R \in D_{nl} \cup \{\bot\}$,
(b) $\forall p \in \{0, \ldots, n - 1\} \ M[p] \in D_{nl} \cup \{\bot\}$,
(c) $\forall i \in \{0, \ldots, \lambda - 1\}$:
   (i) $w_i.col \in \{\text{red, black}\}$,
   (ii) $w_i.key \in D_{\text{pred,key}}$,
   (iii) $w_i.val \in D_{\text{pred,val}}$,
   (iv) $w_i.ind \in D_{\text{ind}}$,
   (v) $w_i.lc, w_i.rc \in D_{nl}$.

Hence, $D_{dst} = D_{nl} \cup \{\bot\}$.

As discussed in Section 1.3, using the implementation of Blelloch and Wei [14], a destination array can be constructed from $O(n^2)$ CAS objects and registers, each of size $W$ bits, where $W \geq \max\{\log |D_{dst}|, \log n\} + O(1)$. Also, the size of each register must be greater than $\max\{\log |D_{nl}|, \log |D_{\text{pred,key}}|, \log |D_{\text{pred,val}}|, \log |D_{\text{ind}}|\} + O(1)$. Therefore, we can construct our single-writer predecessor object from $O(n^2 + \lambda)$ CAS objects and registers, each of size $W$ bits, if $W \geq \max\{\log |D_{nl}|, \log |D_{\text{pred,key}}|, \log |D_{\text{pred,val}}|, \log |D_{\text{ind}}|\} + O(1)$. Notice that $|D_{nl}| = |D_{ind}|$. Hence, it suffices that $W \geq$
max\{\log |D_{nl}|, \log |D_{pred,\text{key}}|, \log |D_{pred,\text{val}}|\} + O(1)$. Considering that

\[
\max\{\log |D_{nl}|, \log |D_{pred,\text{key}}|, \log |D_{pred,\text{val}}|\} + O(1) = \\
\max\{\log (n\Delta \log \Delta), \log |D_{pred,\text{key}}|, \log |D_{pred,\text{val}}|\} + O(1) = \\
\max\{\log n + \log \Delta + \log \log \Delta, \log |D_{pred,\text{key}}|, \log |D_{pred,\text{val}}|\} + O(1) \leq \\
\max\{\log n + 2 \log \Delta, \log |D_{pred,\text{key}}|, \log |D_{pred,\text{val}}|\} + O(1),
\]

we can construct our object from $O(n^2 + n\Delta \log \Delta)$ base objects of size $W$ bits, if $W \geq \max\{\log n + 2 \log \Delta, \log |D_{pred,\text{key}}|, \log |D_{pred,\text{val}}|\} + O(1)$.
Chapter 4

Single-Writer Adaptive Partial Snapshot Algorithm

In this chapter, we present a linearizable wait-free single-writer adaptive partial snapshot object. We start by describing an unbounded wait-free algorithm in Section 4.1. Then we extend that algorithm in Section 4.2 to reach a bounded wait-free implementation. Finally, we provide the linearizability proof and the analysis of our object in Sections 4.3 and 4.4, respectively.

4.1 The Basic Algorithm

We begin by describing a simple non-linearizable algorithm to convey our key ideas, see Algorithm 3. We assign each operation a point in time, that would seem to be a natural linearization point. Even though, as we will show later, operations cannot always linearize at these points, this assignment motivates the choice of linearization points in our final algorithm.

Algorithm 3 uses a shared FAI object $C$ and an array of single-writer predecessor objects $S[0 \ldots n-1]$, each with capacity $\Delta = \infty$. Process $p$ is the only process that can perform update operations on $S[p]$. In an Update() or a Scan() operation $op$ by process $p$, $p$ increments $C$ from a positive integer $x$ to $x+1$, see lines 53 and 56. We associate $op$ with $x$ and linearize $op$ at the point in time at which $p$ increments $C$ from $x$ to $x+1$. Thus, $x$ can be used to determine the relative position of $op$ in the linearization order of Update() and Scan() operations. In other words, if the associated value of an Update() or a Scan() operation is less than $x$, then that operation linearizes before $op$, and if the associated value is greater than $x$, then that operation linearizes.
Shared:
Fetch-And-Increment $C$, initially 0
Predecessor $S[0 \ldots n - 1]$, $S[i]$, $i \in \{0, \ldots, n - 1\}$, is empty

Local for each Process $myID$:

```plaintext
ls_{myID} ← C.FAI()
```

Function $Update(u)$

```plaintext
a ← C.FAI()
S[myID].Insert(a, u)
```

Function $Scan()$

```plaintext
ls_{myID} ← C.FAI()
```

Function $Observe(pID)$

```plaintext
((a, u), f) ← S[pID].Pred(ls_{myID})
```

return $u$

Algorithm 3: Non-linearizable adaptive partial snapshot algorithm.

after $op$. In the case that $op$ is an $Update()$ operation with argument $v$, $p$ inserts the pair $(x, v)$ into $S[p]$ in line 54, and in the case that $op$ is a $Scan()$ operation, $p$ stores $x$ in a local variables $ls_p$. If $op$ is a $Scan()$ operation, then $p$ can later call $Observe(q)$, where $q$ is a process, and perform $S[q].Pred(ls_p)$ in line 58 to determine the argument of the last $Update()$ by process $q$ that linearizes before the linearization point of $op$. The $S[q].Pred(ls_p)$ call by $p$ returns $((x', v'), f)$, where $x'$ is the greatest key less than or equal to $ls_p$ in $S[q]$. (Recall that $f$ is the flag returned by the $Pred()$ operation.) The pair $(x', v')$ shows that the value of the $q$-th component of the snapshot object at the linearization point of $op$ is $v'$.

To avoid having to deal with edge cases, we assume w.l.o.g. that every execution of our algorithm begins with $n$ sequential $Update(\bot)$ operations, one for each process in $\{0, 1, \ldots, n - 1\}$. First, process 0 performs an $Update(\bot)$, followed by an $Update(\bot)$ call by process 1 and eventually, process $n - 1$ executes an $Update(\bot)$.

Now we provide a non-linearizable execution of Algorithm 3. Figure 4 is an illustration of this execution after the initialization described above. Process $p$ first performs a complete $Update(v)$ operation $up$, during which $p$ fetches $x$ by performing $C.FAI()$ in line 53 and inserts $(x, v)$ into $S[p]$ in line 54. Note that $x \geq n$ because of the initialization procedure. After the response of $up$, $p$ invokes another $Update(v')$ operation $up'$. In $up'$, $p$ fetches $x' = x + 1$ by performing $C.FAI()$ in line 53 and falls asleep. While process $p$ is idle, another
process \( q \) performs a complete \( \text{Scan()} \) operation \( \text{sc} \), during which \( q \) performs \( C.FAI() \) in line 56 and fetches \( y = x' + 1 \) from \( C \). After the response of \( \text{sc} \) and before \( p \) performs \( S[p].\text{Insert}(x', v') \) in line 54 of \( \text{up}' \), \( q \) performs a complete \( \text{Observe}(p) \) operation \( \text{ob} \). Throughout the execution of \( \text{ob} \), the only pairs in \( S[p] \) are \((x, v)\) and \((p, \perp)\), where \( p < x < y \). Hence, when \( q \) performs \( S[p].\text{Pred}(y) \) in line 58 of \( \text{ob} \), that operation returns \( v \). So far the return value of \( \text{ob} \) does not match our linearization points. However, this execution is still linearizable because \( \text{up}' \) can be linearized after \( \text{sc} \). But if process \( p \) completes \( \text{up}' \) after the response of \( \text{ob} \) and then process \( q \) performs another complete \( \text{Observe}(p) \) operation \( \text{ob}' \), we obtain a non-linearizable execution: Since \( x < x' < y \) and \( p \) inserts \((x', v')\) into \( S[p] \) before \( q \) invokes \( \text{ob}' \), process \( q \) returns \( v' \) instead of \( v \) in \( \text{ob}' \). So we have two \( \text{Observe}(p) \) operations by process \( q \) that have the same preceding \( \text{Scan()} \) but return different values, which is incorrect.

We extend Algorithm 3 to obtain a linearizable adaptive partial snapshot object. For this purpose, we use an additional shared double-word LL/SC object for each process \( p \), \( LU[p] \), and a method \text{HelpUpdate()} \), see Algorithm 4. The following is a high-level overview of the techniques used in this algorithm:

- **Announcement**: During an \( \text{Update}(v) \) operation \( op \) by process \( p \), \( p \) announces the start of \( op \) by writing \((\perp, v)\) in \( LU[p] \).

- **Helping**: During an \( \text{Observe}(q) \) call by process \( p \), first \( p \) executes \text{HelpUpdate}(p) \) to help a possible pending \( \text{Update()} \) operation by \( q \) to linearize after \( p \)'s preceding \( \text{Scan()} \).

Next, we describe these techniques in more detail. The value of \( LU[p] \) is a pair \((x, v)\), such that \( x \) is either \( \perp \) or a positive integer fetched from \( C \). If \( x = \perp \), then \( p \) has a pending \( \text{Update}(v) \) operation. Otherwise, \( LU[p] \)
Shared:
  Fetch-And-Increment \( C \)
  LL/SC \( LU[0 \ldots n - 1] \)
  Predecessor \( S[0 \ldots n - 1] \)

Local for each Process \( myID \):
  \( ls_{myID} \)

Function Update \( (u) \)
  \( LU[myID].LL() \)
  \( LU[myID].SC(\perp, u) \)
  HelpUpdate \( (myID) \)
  \( (a, u') \leftarrow LU[myID].LL() \)
  \( S[myID].Insert(a, u) \)

Function HelpUpdate \( (pID) \)
  \( (a, u) \leftarrow LU[pID].LL() \)
  if \( a = \perp \) then
    \( a' \leftarrow C.FAI() \)
    \( LU[pID].SC(a', u) \)

Function Scan \( () \)
  \( ls_{myID} \leftarrow C.FAI() \)

Function Observe \( (pID) \)
  HelpUpdate \( (pID) \)
  \( (a, u) \leftarrow LU[pID].LL() \)
  if \( a \neq \perp \) and \( a < ls_{myID} \) then
    return \( u \)
  else
    \( \left( (a', u'), f \right) \leftarrow S[pID].Pred(ls_{myID}) \)
    return \( u' \)

Algorithm 4: Basic adaptive partial snapshot algorithm.
indicates that the last Update() by process $p$ had parameter $v$ and its relative position in the linearization order of Update() and Scan() operations is $x$. In a HelpUpdate($q$) call by process $p$, $p$ reads $(x', v')$ from $LU[p]$ in line 67. If $q$ has a pending Update() operation ($x' = \bot$), then $p$ performs C.FAI() in line 69 to fetch a positive integer $y$, and performs $LU[q].SC(y, v')$ in line 70. If that operation succeeds, then the pending Update() operation by $q$ linearizes at the point in time at which $p$ fetches $y$ from $C$. Otherwise, some other process (possibly $q$ itself) has already helped the pending Update() by $q$ to linearize.

The announcement and helping techniques affect Update() and Observe() methods. At the beginning of an Update($v$) operation $up$ by process $p$, $p$ announces the start of $up$ by writing $(\bot, v)$ in $LU[p]$. (This is done by performing $LU[p].LL()$ and $LU[p].SC(\bot, v)$, consecutively in lines 61 and 62. Later, we show in Claim 37 that the SC() operation always succeeds.) Then $p$ executes a HelpUpdate($p$) operation $hu$ in line 63 of $up$. In $up$, after $p$ has written $(\bot, v)$ in $LU[p]$, either process $p$ or another process changes $LU[p]$ from $(\bot, v)$ to $(x, v)$ in a HelpUpdate($p$) operation (this HelpUpdate($p$) operation might be $hu$), where $x$ is a positive integer fetched from $C$. We associate $x$ with $up$ and linearize this operation at the point in time at which $C$ is incremented from $x$ to $x + 1$. Because of the helping technique, $C$ might be incremented multiple times during an Update() operation. However, we still associate $up$ with one such increment. At the end of $up$, $p$ inserts $(x, v)$ into $S[p]$. The first entry of the pair stored in $LU[p]$ is denoted by $LU[p].key$ and the second entry is denoted by $LU[p].val$.

During an Observe($q$) operation $ob$ by process $p$, first $p$ calls a HelpUpdate($q$) operation $hu$ in line 74. If there is a pending Update() operation by $q$, $p$ helps that operation in $hu$ to linearize. Then $p$ reads $(x', v')$ from $LU[p]$ in line 75. Since $p$ has already performed $hu$, if $x' = \bot$, then process $q$ performed another Update() operation after the invocation of $ob$. Clearly, this Update() operation happens after the preceding Scan() by $p$ and does not affect the return value of $ob$. In addition, if $x' > ls_p$, then the last Update() by $q$ linearizes after the linearization point of $p$’s preceding Scan() and also does not affect the return value of $ob$. (Recall that $ls_p$ is the value associated with $p$’s preceding Scan().) Therefore, in these two cases ($x' = \bot$ or $x' > ls_p$), $p$ performs $S[q].Pred(ls_p)$ in line 80 to determine the value of the $q$-th component of the snapshot at the linearization point of $p$’s preceding Scan().
Otherwise, \( x' \neq \perp \) and \( x' < l_s p \) which indicates that the last \texttt{Update()} by \( q \) linearizes before the linearization point of \( p \)'s preceding \texttt{Scan}(). Hence, the \( q \)-th entry of the snapshot array has value \( v' \) at the linearization point of \( p \)'s preceding \texttt{Scan}(). Therefore, \( p \) returns \( v' \) in \textit{ob} in line 77.

In Algorithm 4, the number of pairs stored in the predecessor objects increases with the number of \texttt{Update()} operations. Thus, predecessor objects with capacity \( \Delta = \infty \) are needed. The time complexity of each operation on these predecessor objects is unbounded. Therefore, although this algorithm is linearizable and wait-free, the time complexity of \texttt{Update()} and \texttt{Observe()} methods, which are dominated by the time complexity of an operation on a predecessor object, is unbounded. In the next section, we remove unnecessary elements from the predecessor objects to bound the time complexity.

4.2 The Memory Reclamation Algorithm

First, we specify the pairs that can be removed from the predecessor objects of Algorithm 4. Then we design Algorithm 5, which incorporates a \texttt{Prune()} method that prunes the unnecessary elements from \( S[0 \ldots n-1] \). This leads to an upper bound of \( 3n \) for the number of pairs in each of \( S[0 \ldots n-1] \). Hence, for a process \( p \), we can change the capacity bound (\( \Delta \)) of \( S[p] \) from \( \infty \) to \( 3n \).

In the following, \( x_t \) denotes the value of a shared or a local variable \( x \) at a point in time \( t \). If \( x \) is modified at \( t \), we let \( x_{t^*} \) be the value of \( x \) just after \( t \).

Let \( t \) be an arbitrary point in time. Recall that the pairs stored in \( S[p] \) can be later accessed by a process \( q \) in an \texttt{Observe}(\( p \)) operation. In such an \texttt{Observe}(\( p \)) operation, \( q \) performs \( S[p].\texttt{Pred}(l_s q) \) in line 79 to retrieve a pair from \( S[p] \). (Recall that during a \texttt{Scan()} operation by process \( q \), \( q \) performs a \texttt{C.FAI()} and stores the fetched value into \( l_s q \). The value of \( l_s q \) at a point in time \( t \) is denoted by \( l_{s,q,t} \).) Hence, if we guarantee that a pair will never be returned by an \( S[p].\texttt{Pred}(l_s q) \) operation after \( t \), for \( 0 \leq q < n \), then we can remove that pair from \( S[p] \) at any point in time after \( t \). In the following, we explain in detail which pairs will not be returned by any \( S[p].\texttt{Pred}(l_s q) \) operations after \( t \).

Let \( \textit{last} \) be the pair with the largest key in \( S[p]_t \) and \( U = S[p]_t \setminus \textit{last} \). Then for any process \( q \) and any point in time \( t^* > t \),

\[
ls_{q,t^*} = ls_{q,t} \quad \text{or} \quad ls_{q,t^*} > \textit{last.key} \tag{4.1}
\]
because if $ls_q$ is not modified during $[t, t^*]$, then $ls_{q,t^*} = ls_{q,t}$. Otherwise, $ls_q$ is changed at a point in time $t_1$, such that $t \leq t_1 \leq t^*$. Since $ls_q$ can only be modified in line 72 of a $\text{Scan}(\cdot)$ operation by $q$, at $t_1$, $q$ changes the value of $ls_q$ to $C_{t_1} - 1$. Also, since $last$ is already in $S[p]$ at $t$, at a point in time before $t$, $p$ has completed the $\text{Update}(\cdot)$ operation in which $last$ was inserted into $S[p]$. Therefore, $last.key$ is the value of $C_{t_2} - 1$, where $t_2 < t \leq t_1$. So by the fact that the value of $C$ is increasing and the fact that $C$’s value changes at $t_1$ and $t_2$, $ls_{q,t^*} > last.key$.

Assume that the return value of $S[p].\text{Pred}(ls_{q,t})$ is $\ell_q$ (To avoid writing $t$ as the subscript of both $S[p]$ and $ls_q$, instead of writing $S[p]_t.\text{Pred}(ls_{q,t})$, we write $S[p].\text{Pred}(ls_{q,t})$). By (4.1), for any process $q$ and any point in time $t^* > t$, the return value of $S[p].\text{Pred}(ls_{q,t^*})$ is

$$
\begin{cases} 
\ell_q & \text{if } ls_{q,t^*} = ls_{q,t} \\
(k, v) \text{ s.t } k \geq last.key & \text{if } ls_{q,t^*} > last.key 
\end{cases}
$$

(4.2)

Recall that all the pairs in $U$ have keys less than $last.key$. Hence, a pair $(k, v)$, such that $k \geq last.key$ does not belong to $U$. Therefore, at any point in time after $t$, if process $q$ calls $S[p].\text{Pred}(ls_q)$, then that operation either returns $\ell_q$ or a pair that does not belong to $U$. Hence, all the pairs in $U \setminus E$ will not be returned by $S[p].\text{Pred}(ls_q)$ operations after $t$, where

$$E = \{ \ell_q \mid 0 \leq q < n \}$$

(4.3)

So we can remove all the pairs in $U \setminus E$ from $S[p]$ after $t$.

Since our predecessor objects are single-writer, only process $p$ can remove elements from $S[p]$. However, process $p$ cannot compute $U \setminus E$ because computing $E$ requires reading the local variables $ls_0, \ldots, ls_{n-1}$. Therefore, we replace these local variables with shared single-word LL/SC objects $LS[0 \ldots n - 1]$, and use similar announcement and helping techniques that are used in Algorithm 4. To implement these techniques, we use a helping method $\text{HelpScan}(\cdot)$ (see Algorithm 5). During a $\text{Scan}(\cdot)$ operation $sc$ by process $q$, $q$ announces the start of $sc$ by writing ⊥ in $LS[q]$. Also, whenever process $q$ wants to read the value stored in $LS[p]$, it first executes a $\text{HelpScan}(p)$ to help a possible pending $\text{Scan}(\cdot)$ by $p$ to complete.

First, we explain the announcement and helping mechanisms and then describe how we can implement a $\text{Prune}(\cdot)$ operation using these techniques.
Algorithm 5: Adaptive partial snapshot algorithm with memory reclamation
In a \texttt{HelpScan}(p) call by process \(q\), first \(q\) reads \(x\) from \(LS[p]\) in line 108. If \(x = \bot\), then \(p\) has a pending \texttt{Scan()} operation \(psc\). In such a case, \(q\) helps \(psc\) complete: \(q\) fetches a positive integer \(y\) by performing \(C.FAI()\) in line 110 and executes \(LS[p].SC(y)\) in line 111. If that operation succeeds, then \(psc\) linearizes at the point in time at which \(q\) fetches \(y\) from \(C\). Otherwise, some other process (possibly \(p\) itself) has already helped \(psc\) complete. In a \texttt{Scan()} operation \(sc\) by process \(q\), first \(q\) writes \(\bot\) to \(LS[q]\) by performing \(LS[q].LL()\) and \(LS[q].SC(\bot)\), consecutively, in lines 104 and 105. (We later prove in Section 4.3 that the \texttt{SC()} operation at the beginning of each \texttt{Scan()} call succeeds.) Then \(q\) calls a \texttt{HelpScan}(q) operation \(hs\). In \(sc\), after \(q\) has written \(\bot\) to \(LS[q]\), either process \(q\) or another process changes \(LS[q]\) from \(\bot\) to \(x\) in a \texttt{HelpScan}(q) operation (this \texttt{HelpScan}(q) operation might be \(hs\)), where \(x\) is a positive integer fetched from \(C\). We associate \(x\) with \(sc\) and linearize this \texttt{Scan()} operation at the point in time at which \(C\) is incremented from \(x\) to \(x+1\).

Next, we describe the \texttt{Prune()} method of Algorithm 5. At the beginning of a \texttt{Prune()} call by \(p\), \(p\) assigns the pair with the largest key in \(S[p]\) to \(last\) by performing \(S[p].Pred(\infty)\) in line 114. Let \(t\) be the point in time at which \(S[p].Pred(\infty)\) occurs and \(U = S[p]_{t} \setminus \{last\}\). (Notice that the value of \(C\) at any point in time after \(t\) is greater than \(last.key\).) Hence, if process \(p\) inserts a pair into \(S[p]\) after \(t\), then the key of that pair is greater than \(last.key\).)

Afterward, \(p\) computes a set \(E\) of protected pairs as follows: For each process \(q\), \(p\) first executes a \texttt{HelpScan}(q) \(hs_{q}\) in line 116, and then reads a value \(a_{q}\) from \(LS[q]\) in line 117. Finally, if \(a_{q} \neq \bot\), then \(p\) inserts the return value of \(S[p].Pred(a_{q})\) into \(E\) in line 119. Suppose the return value of \(S[p].Pred(a_{q})\) is \(\ell_{q}\) and \(t_{q}\) is the point in time at which \(p\) reads \(a_{q}\) from \(LS[q]\). Since \(p\) performs a \texttt{HelpScan}(q) call after \(t\), at any point in time after \(t_{q}\), \(LS[q]\) is either \(a_{q}, \bot\), or \(C_{t^\ast}\), where \(t^\ast > t\). (Recall that \(C_{t^\ast} > last.key\).) Therefore, the only pair that might belong to \(U\) and might be returned by an \(S[p].Pred(\ell_{q})\) operation after \(t_{q}\) is \(\ell_{q}\). Hence, all the pairs in \(U \setminus E\) can be removed from \(S[p]\). So \(p\) traverses through all the pairs in \(U\) and removes the ones that do not belong to \(E\) in lines 121-125. To this end, process \(p\) first assigns the return value of \(S[p].Succ(-\infty)\) to \(cur\). Notice that \(cur\) is the pair with the smallest key in \(S[p]\). Then process \(p\) performs a while-loop in lines 121-125. In each iteration of that while-loop, first \(p\) finds the pair with smallest key greater than \(cur.key\) by performing \(S[p].Succ(cur)\) and assigning its return value to a
local variable \( n_{xt} \) in line 122. Afterward, if \( \text{cur} \notin E \), \( p \) removes \( \text{cur} \) from \( S[p] \) in line 124. Finally, \( p \) replaces \( \text{cur} \) with \( n_{xt} \) in line 125. The while-loop stops iterating if \( \text{cur}.\text{key} \geq \text{last}.\text{key} \). Hence, in this while-loop, process \( p \) traverses through all the pairs in \( U \).

To obtain an efficient implementation, we distribute the work of a \( \text{Prune()} \) call by \( p \) over \( n \) \( \text{Update()} \) operations by \( p \). We maintain the invariant that \( p \)'s predecessor object contains at most \( 3n \) pairs at any point in time, and thus we can use a predecessor object with bounded capacity \( \Delta = 3n \). In the following, we use the same notations as the paragraph above. Notice that in each iteration of the for-loop in lines 115-119, at most one pair is inserted into \( E \). Since that for-loop iterates \( n \) times, \( |E| \leq n \). In the while-loop in lines 121-125, at most \( n \) pairs in \( U \setminus E \) are not removed from \( S[p] \). Also, since a \( \text{Prune()} \) operation is distributed over \( n \) \( \text{Update()} \) operations, and during each \( \text{Update()} \) operation one pair is inserted into \( S[p] \), \( n \) pairs are inserted into \( S[p] \) throughout a \( \text{Prune()} \) call. Therefore, the total number of pairs stored in \( S[p] \) at the response of each \( \text{Prune()} \) operation is at most \( 2n \). Since \( p \) performs a new \( \text{Prune()} \) call in every \( n \) \( \text{Update()} \) operations, the number of pairs stored in \( S[p] \) at any point in time is at most \( 3n \). Therefore, using our bounded capacity predecessor object, implemented in Chapter 3, the time complexity of each operation on \( S[p] \) is \( O(\log n) \).

Now we can analyze the time complexity of each method in Algorithm 5. Recall that each \( \text{LL()} \) and \( \text{SC()} \) operation takes constant time. Therefore, methods \( \text{HelpScan()} \) and \( \text{HelpUpdate()} \) have constant time complexity. In a \( \text{Prune()} \) call, we can implement the local set \( E \) using a balanced binary tree. As discussed above, \( |E| \leq n \) and the number of pairs in a predecessor object is at most \( 3n \). Hence, each operation on \( E \) or \( S[p] \) takes \( O(\log n) \) time, and the while-loop in lines 121-125 iterates at most \( 3n \) times. Since the for-loop and the while-loop both iterates \( O(n) \) times, each of those loops take \( O(n \log n) \) time. So the total time complexity of a \( \text{Prune()} \) operation is \( O(n \log n) \). Since we distribute the work of a \( \text{Prune()} \) call over \( n \) \( \text{Update()} \) operations, the extra work in each \( \text{Update()} \) operation is \( O(\log n) \). Therefore, the time complexity of \( \text{Update()} \) and \( \text{Observe()} \) is \( O(\log n) \), and \( \text{Scan()} \) is \( O(1) \). A detailed analysis of this algorithm is provided in Section 4.4.
4.3 Correctness Proof

In this section, we prove the correctness (i.e., linearizability) of Algorithm 5. Let $\mathcal{E}$ be an arbitrary finite execution of our adaptive partial snapshot object. Since our implementation is wait-free, we can assume without loss of generality that all operations in $\mathcal{E}$ complete. If some operations do not complete, we can construct a completion of the execution, by letting processes with pending operations take steps in an arbitrary order until their operations have completed. Note that this is possible due to the progress property of our algorithm (i.e. wait-freedom), as described in Section 1.1. Also, recall that every execution of our object starts with $n$ sequential $\text{Update()}$ operations. First, process 0 performs a $\text{Update}(\perp)$ operation and then each process $p$ ($1 \leq p < n$) after the response of process $p - 1$ $\text{Update}(\perp)$ call, executes an $\text{Update}(\perp)$. As discussed in Section 1.1, to prove the correctness of our algorithm, it suffices to show that $\mathcal{E}$ is linearizable.

In Section 4.3.1, we define a mapping $\text{lin}$ that maps each $\text{Update()}$, $\text{Scan()}$ and $\text{Observe()}$ operation $op$ in $\mathcal{E}$ to a point in time, such that $\text{inv}(op) \leq \text{lin}(op) \leq \text{rsp}(op)$. Then in Sections 4.3.2 - 4.3.4 we prove that if we order all operations by $\text{lin}$, the resulting sequential history is valid (with respect to the specification of the adaptive partial snapshot type).

To improve the readability of the proofs, for each $\text{Update()}$ and $\text{Scan()}$ operation $op$, we define $t(op)$ and $t'(op)$ as follows: Let $p$ be a process, $up$ an $\text{Update()}$ operation by $p$ and $sc$ a $\text{Scan()}$ operation by $p$. We define $t(up)$ as the point in time at which process $p$ performs $LU[p].\text{SC()}$ in line 83 of $up$, and $t(sc)$ as the point in time at which process $p$ performs $LS[p].\text{SC()}$ in line 105 of $sc$. If $LU[p]$ is modified in $(t(up), \infty)$, then we let $t'(up)$ be the first point when this happens in that interval, and otherwise we let $t'(up) = \infty$. Similarly, if $LS[p]$ is changed in $(t(sc), \infty)$, $t'(sc)$ denotes the first point when this happens in that interval, and otherwise $t'(sc) = \infty$.

4.3.1 Linearization Points

Below we will show two lemmas that facilitate the definition of mapping $\text{lin}$. Lemma 20 shows that during each $\text{Update()}$ operation by process $p$, $LU[p].\text{key}$ is changed exactly once from $\perp$ to a positive integer fetched from $C$. (It is possible that $p$ changes $LU[p]$ when it calls $\text{HelpUpdate()}$ during its own $\text{Update()}$, or some other process may do that during a $\text{HelpUpdate()}$ call.)
Lemma 21 proves a symmetric statement for $LS[p]$. The proofs of these two lemmas are presented in Section 4.3.4.

**Lemma 20.** Let $p$ be a process that executes first an $Update(v)$ operation $up$ and later an $Update(v')$ operation $up'$, and no $Update()$ operation in-between $up$ and $up'$. Then

(a) at point $t(up)$ process $p$ changes $LU[p].key$ to $\perp$,
(b) $t'(up)$ is before the point in time at which $p$ reads $LU[p]$ in line 85 of $up$,
(c) at point $t'(up)$ some process performs a successful $LU[p].SC(x,v)$ operation in line 93 of a $HelpUpdate(p)$ call, for some $x \in \mathbb{N}$, and
(d) $LU[p]$ remains unchanged throughout $\left(t'(up), t(up')\right)$.

**Lemma 21.** Let $p$ be a process that executes first a $Scan()$ operation $sc$ and later a $Scan()$ operation $sc'$, and no $Scan()$ operation between $sc$ and $sc'$. Then

(a) at point $t(sc)$ process $p$ changes $LS[p]$ to $\perp$,
(b) $t'(sc) < rsp(sc)$,
(c) at point $t'(sc)$ some process performs a successful $LS[p].SC(x)$ operation in line 111 of a $HelpScan(p)$ call, where $x \in \mathbb{N}$, and
(d) $LS[p]$ remains unchanged throughout $\left(t'(sc), t(sc')\right)$.

First, for each $Update()$ and $Scan()$ operation $op$, we define a positive integer $asn(op)$ and an operation $help(op)$, using Lemmas 20 and 21, and then we define the mapping $lin$. Suppose $p$ is the process that calls $op$. In the case that $op$ is an $Update()$ operation, by Lemma 20 (c), $LU[p].key$ is changed to a positive integer $x$ at $t'(op)$ during a $HelpUpdate(p)$ operation. The value $x$ is denoted by $asn(op)$ and that $HelpUpdate(p)$ operation by $help(op)$. Similarly, in the case that $op$ is a $Scan()$ operation, based on Lemma 21 (c), $LS[p]$ is changed to an integer $y$ at $t'(op)$ during a $HelpScan(p)$ operation. Let $asn(op) = y$ and $help(op)$ be that $HelpScan(p)$ operation.

Let $op$ be an $Observe()$, $Scan()$ or $Update()$ operation. Recall that we consider only complete executions. If $op$ is an $Observe()$ operation, then we define $lin(op) =rsp(op)$. If it is an $Update()$ or $Scan()$, then $lin(op)$ is the point at which $C.FAI()$ is executed in $help(op)$ (in line 92 or line 110). We say that operation $op$ linearizes at point $lin(op)$. Now, we show that $lin(op)$ is always in $[inv(op), rsp(op)]$. 50
Correctness of Mapping lin

Claim 22. Let $op$ be an $\text{Update()}$ or $\text{Scan()}$ operation. Then $t(op) < \text{lin}(op) < t'(op)$.

Proof. We prove this claim for the case that $op$ is an $\text{Update()}$ operation. A symmetric proof applies when $op$ is a $\text{Scan()}$.

Let $q$ be the process calling $\text{help}(op)$. By definition of $\text{lin}$, $op$ linearizes when $q$ performs $C\text{.FAI()}$ in line 92 of $\text{help}(op)$. By Lemma 20 (c), at point $t'(op)$ process $q$ executes line 93 of $\text{help}(op)$. So $\text{lin}(op) < t'(op)$. Thus, it remains to show that $t(op) < \text{lin}(op)$.

Let $t_1$ be the point at which $q$ performs $LU[p].\text{LL()}$ in line 90 of $\text{help}(op)$. Since $q$’s $LU[p].\text{SC()}$ at $t'(op)$ in line 93 of $\text{help}(op)$ succeeds, there is no successful $LU[p].\text{SC()}$ throughout $[t_1, t'(op))$. Also, based on Lemma 20 (a), there is a successful $LU[p].\text{SC()}$ at $t(op)$, and by definition $t(op) < t'(op)$. Therefore, $t(op) < t_1$ and since $t_1 < \text{lin}(op)$, it follows that $t(op) < \text{lin}(op)$. □

Claim 23. Let $op$ be an $\text{Update()}$, $\text{Observe()}$, or $\text{Scan()}$ operation. Then $\text{inv}(op) \leq \text{lin}(op) \leq \text{rsp}(op)$.

Proof. If $op$ is an $\text{Observe()}$ operation, then $\text{lin}(op) = \text{rsp}(op)$. If it is an $\text{Update()}$ or $\text{Scan()}$ operation, then $t(op) < \text{lin}(op) < t'(op)$ by Claim 22. Moreover, $\text{inv}(op) < t(op)$ by definition, and $t'(op) < \text{rsp}(op)$ by Lemmas 20 (b) and 21 (b), respectively. Therefore, $\text{inv}(op) \leq \text{lin}(op) \leq \text{rsp}(op)$. □

Assume that process $p$ executes a $\text{Scan()}$ $sc$ and later completes an $\text{Observe}(q)$ operation that returns a value $v$. (Recall that $p$ is not allowed to call $\text{Observe()}$ until it has completed a $\text{Scan()}$.) Then the sequential specification dictates that the latest $\text{Update()}$ by process $q$ that linearizes before $sc$ uses parameter $v$. (Such an $\text{Update()}$ operation that linearizes before $sc$ always exists because $E$ starts with $n$ sequential $\text{Update()}$ operations, one called by each process in $\{0, \cdots, n-1\}$.) In fact, since neither $\text{Update()}$ nor $\text{Scan()}$ returns anything, execution $E$ is linearizable, if this is true for every $\text{Observe()}$ operation. We will show in Section 4.3.3 that this is the case. To facilitate that proof, we will first show in Section 4.3.2 that if a process $q$ executes an $\text{Update}(u)$ operation, then once it has added $u$ to $S[q]$, it will not remove $u$ from $S[q]$ until all $\text{Observe()}$ operations that might have to return $u$ have responded.
4.3.2 Correctness of Memory Reclamation

We prove the correctness of our memory reclamation method in Lemma 28. To facilitate that proof, first we show a relation between \( \text{asn}(op) \) and \( \text{lin}(op) \), for an \texttt{Update()} or a \texttt{Scan()} operation \( op \). (See Claim 24.) Then we prove that during an \texttt{Update}(v) operation \( op \) by process \( p \), the pair \( (\text{asn}(op), v) \) is inserted into \( S[p] \). (See Claim 25.) And finally, in Claim 26, we show an interesting property of the \texttt{HelpScan()} method that helps us to prove Lemma 28.

Claim 24. Let \( op_1 \) and \( op_2 \) each be an \texttt{Update()} or \texttt{Scan()} operation. Then \( \text{lin}(op_1) < \text{lin}(op_2) \) if and only if \( \text{asn}(op_1) < \text{asn}(op_2) \).

Proof. The definition of \( \text{lin} \) together with Lemmas 20 (c) and 21 (c), imply that \( \text{asn}(op_1) \) is the value of \( C \) just before \( \text{lin}(op_1) \) and \( \text{asn}(op_2) \) is the value of \( C \) just before \( \text{lin}(op_2) \). There is a \texttt{C.FAI()} operation at \( \text{lin}(op_1) \) and \( \text{lin}(op_2) \). Therefore, by the fact that the value of \( C \) is increasing, \( \text{lin}(op_1) < \text{lin}(op_2) \) if and only if \( \text{asn}(op_1) < \text{asn}(op_2) \). \( \square \)

Claim 25. If process \( p \) completes an \texttt{Update}(v) operation \( up \), then in line 86 of \( up \) it inserts \( (\text{asn}(up), v) \) into \( S[p] \).

Proof. Let \( t_1 \) be the point in time at which process \( p \) reads \( LU[p] \) in line 85 of \( up \). By Lemma 20 (b), \( t'(up) < t_1 \). By definition of \( t'(up) \) and Lemma 20 (c), at \( t'(up) \), \( (\text{asn}(up), v) \) is written to \( LU[p] \). Also, by Lemma 20 (d), \( LU[p].\text{key} \) is not changed throughout \( (t'(op), rsp(op)) \). Therefore, process \( p \) reads \( (\text{asn}(up), v) \) in line 85 of \( up \), and then inserts that pair into \( S[p] \) in line 86. \( \square \)

The next claim shows that if a \texttt{Scan()} operation \( sc \) by \( p \) linearizes before the invocation of a \texttt{HelpScan}(p) operation \( hs \) by \( q \), then \( \text{asn}(sc) \) is written to \( LS[p] \) before \( rsp(hs) \). Therefore, if \( q \) reads \( LS[p] \) after it performed \( hs \), then it either reads \( \text{asn}(sc) \) or a value that is written to \( LS[p] \) by \( p \) during a \texttt{Scan()} operation that linearizes after \( \text{inv}(hs) \). This shows that the \texttt{HelpScan()} method prevents processes from reading outdated values from \( LS[0 \cdots n-1] \).

Claim 26. Suppose process \( p \) executes a \texttt{Scan()} operation \( sc \), and process \( q \) executes a \texttt{HelpScan}(p) operation \( hs \), such that \( \text{lin}(sc) < \text{inv}(hs) \). Then \( t'(sc) < rsp(hs) \).

Proof. For the purpose of a contradiction, assume \( rsp(hs) \leq t'(sc) \). By definition and Lemma 21 (a), \( LS[p] = \bot \) throughout \( (t(sc), t'(sc)) \) and it is not
modified in this interval. Also, by Claim 22, \( t(sc) < \text{lin}(sc) \). Hence, since \( \text{lin}(sc) < \text{inv}(hs) \) and \( \text{rsp}(hs) \leq t'(sc) \), \( LS[p] = \bot \) and it is not modified throughout the execution of \( hs \). So process \( p \) reads \( \bot \) from \( LS[p] \) in line 108 of \( hs \), and changes \( LS[p] \) in line 111 with an \( \text{SC()} \) operation to a positive integer fetched from \( C \). This contradicts the fact that \( LS[p] = \bot \) throughout the execution of \( hs \). 

Finally, we use the above claim to prove that elements added to \( S[0 \cdots n-1] \) are not removed prematurely, i.e., our memory reclamation works correctly. Consider an \( \text{Update}(v) \) operation by process \( p \). In this operation, \( p \) inserts \( (\text{asn}(op), v) \) into \( S[p] \). The memory reclamation scheme ensures that this pair remains in the predecessor object until every \( \text{Observe()} \) operation that may need to return \( v \) has completed. Note that an \( \text{Observe()} \) operation by process \( q \) needs to return \( v \) if \( q \)'s latest preceding \( \text{Scan()} \) linearizes after \( p \)'s \( \text{Update}(v) \) and no \( \text{Update()} \) by \( p \) linearizes in between the \( \text{Update}(v) \) and that \( \text{Scan()} \).

**Claim 27.** Suppose process \( p \) executes an \( \text{Update}(v) \) operation up during which it inserts a pair \( (k, v) \) into \( S[p] \). Let \( sc \) be a \( \text{Scan()} \) operation by some process \( q \), such that \( \text{lin}(up) < \text{lin}(sc) \), and no \( \text{Update()} \) operation by \( p \) linearizes between \( \text{lin}(up) \) and \( \text{lin}(sc) \). If at point \( t^* > \text{rsp}(up) \) the pair \( (k, v) \) is in \( S[p] \), and some process executes \( S[p].\text{Pred}(\text{asn}(sc)) \), then that operation returns \( (k, v) \).

**Proof.** By Claim 25, \( k = \text{asn}(up) \). Therefore, by the assumption of this observation, \( (\text{asn}(up), v) \) is in \( S[p] \) at \( t^* \). Also, by Claim 24, since \( \text{lin}(up) < \text{lin}(sc) \), we have \( \text{asn}(up) < \text{asn}(sc) \). Hence, if there is no pair \( E \) in \( S[p] \) at \( t^* \), such that \( \text{asn}(up) \leq E.key \leq \text{asn}(sc) \), then \( S[p].\text{Pred}(\text{asn}(sc)) \) operation returns \( (k = \text{asn}(up), v) \). We will prove by contradiction that no such pair \( E \) exists.

Suppose at point \( t^* \) there is a pair \( E \) in \( S[p] \), such that \( \text{asn}(up) \leq E.key \leq \text{asn}(sc) \). The pair \( E \) is inserted into \( S[p] \) during an \( \text{Update()} \) operation by process \( p \). Let \( up' \) be that \( \text{Update()} \) operation. By Claim 25, \( E.key = \text{asn}(up') \). We do not consider the cases that \( E.key = \text{asn}(sc) \) or \( \text{asn}(up) \) for an \( \text{Update()} \) or a \( \text{Scan()} \) operation \( op \) is a distinct value fetched from the fetch-and-increment object \( C \). Therefore, \( \text{asn}(up) < \text{asn}(up') \leq \text{asn}(sc) \). Hence, by Claim 24, \( \text{lin}(up) \leq \text{lin}(up') \leq \text{lin}(sc) \). This contradicts the fact that no \( \text{Update()} \) operation by \( p \) linearizes between \( \text{lin}(up) \) and \( \text{lin}(sc) \). 

**Lemma 28.** Suppose process \( p \) executes an \( \text{Update()} \) operation \( up \) and inserts a pair \( (k, v) \) into \( S[p] \) in line 86. Let \( q \) be a process and \( sc \) a \( \text{Scan()} \) operation
by q, such that \(\text{lin}(up) < \text{lin}(sc)\), and no \texttt{Update()} operation by p linearizes between \(\text{lin}(up)\) and \(\text{lin}(sc)\). Let \(t^* > \max\{\text{lin}(sc), \text{rsp}(up)\}\) be a point in time, such that q invokes no \texttt{Scan()} in \([\text{rsp}(sc), t^*]\). Then \((k, v)\) is in \(S[p]\) at \(t^*\).

**Proof.** Let \(t_1\) be the point in time at which \(p\) inserts \((k, v)\) into \(S[p]\). We need to prove that this pair is not removed from \(S[p]\) throughout \((t_1, t^*)\). For the purpose of contradiction, assume \(p\) removes \((k, v)\) from \(S[p]\) at \(t_{\text{rmv}}\) in \((t_1, t^*)\).

The only line in the pseudocode, in which process \(p\) can remove a pair from \(S[p]\) is line 124 of a \texttt{Prune()} operation. Suppose \(pr\) is the \texttt{Prune()} operation, in which \(p\) removes \((k, v)\) from \(S[p]\).

Let \(t_3\) be the point when \(p\) executes \(S[p].\texttt{Pred}(\infty)\) in line 114 of \(pr\), and \(\ell\) the pair that this operation returns. Let \(k' = \ell.\text{key}\). Process \(p\) assigns \(\ell\) to \texttt{last} in line 114 of \(pr\) and later removes pair \((k, v)\) from \(S[p]\) in line 124. Thus, the condition of the while-loop in line 121 of \(pr\) implies that \(k < k'\).

Let \(t_2 < t_3\) be the point when \(p\) inserts the pair with key \(k'\) into \(S[p]\). Then \(p\) executes an \(S[p].\texttt{Insert}\) operation in line 86 at point \(t_2\), during an \texttt{Update()} operation \(up'\). By Claim 25, \(k = \text{asn}(up)\) and \(k' = \text{asn}(up')\). Since \(\text{asn}(up) = k < k' = \text{asn}(up')\), by Claim 24, \(\text{lin}(up) < \text{lin}(up')\). Hence \(p\) executes \(up\) before \(up'\). By Claim 22 and Lemma 20 (b), \(\text{lin}(up') < t'(up) < t_2 < t_3\). Since \(up\) is \(p\)'s last update that linearizes before \(sc\), we conclude \(\text{lin}(sc) < \text{lin}(up') < t_3\).

Let \(hs\) be the \texttt{HelpScan}(q) operation that \(p\) executes in line 116 of \(pr\). Recall that \(t_3\) is the point when \(p\) executes line 114 of \(pr\), and thus \(t_3 < \text{inv}(hs)\). Since \(\text{lin}(sc) < t_3\), we obtain \(\text{lin}(sc) < \text{inv}(hs)\). Therefore, by Claim 26, \(t'(sc) \leq \text{rsp}(hs)\).

By definition, \(LS[q] = \text{asn}(sc)\) at point \(t'(sc)\). Since there is no \texttt{Scan()} by \(q\) throughout \((\text{rsp}(sc), t^*)\), by Lemma 21 (d), \(LS[q] = \text{asn}(sc)\) throughout \((t'(sc), t^*)\). By the assumption that \(t_{\text{rmv}} \leq t^*\), and, as shown above, \(t'(sc) \leq \text{rsp}(hs)\), \(LS[q] = \text{asn}(sc)\) throughout \((\text{rsp}(hs), t_{\text{rmv}})\). Note that \(p\) calls \(hs\) in line 116 of \(pr\), and at point \(t_{\text{rmv}}\) it executes line 124 of \(pr\) (where it removes pair \((k, v)\) from \(S[p]\)). Hence, when \(p\) executes line 117 during \((\text{rsp}(hs), t_{\text{rmv}})\), it reads \(\text{asn}(sc)\) from \(LS[q]\), and in line 119 it determines a pair \((y, u) = S[p].\texttt{Pred}(\text{asn}(sc))\), which it adds to \(E\). Since \((k, v)\) is stored in \(S[p]\) throughout \((t_1, t_{\text{rmv}})\), and \(up\) is \(p\)'s latest \texttt{Update()} preceding \(sc\), by Claim 27, \((y, u) = (k, v)\). Thus, when \(p\) executes line 124 of \(pr\), \((k, v) \in E\), and so \(p\) does not remove \((k, v)\) from \(S[p]\) in that line. This is a contradiction. \(\Box\)
4.3.3 Validity

Consider the sequential execution obtained by ordering all operations in \( \mathcal{E} \) by \( \text{lin} \). We prove in Lemma 32 that this sequential execution is valid. To aid that proof, we use Observation 29 and Claim 30 and 31. The observation is a direct implication of Lemma 21(c) and (d) together with definition of asn.

Observation 29. Let \( \text{ob} \) be an \( \text{Observe}() \) operation executed by process \( p \), and let \( \text{sc} \) be \( p \)'s latest \( \text{Scan}() \) operation preceding \( \text{ob} \). Then \( \text{LS}[p] = \text{asn}(\text{sc}) \) throughout \([\text{inv}(\text{ob}), \text{rsp}(\text{ob})]\).

Claim 30. Suppose process \( p \) executes a \( \text{Scan}() \) operation \( \text{sc} \) and later an \( \text{Observe}(q) \) operation \( \text{ob} \), and no other \( \text{Scan}() \) between \( \text{sc} \) and \( \text{ob} \). Let \( t_1 \) be the point at which \( p \) performs \( \text{LU}[q].\text{LL}() \) in line 96 during \( \text{ob} \). Let \( \text{up} \) be the last \( \text{Update}() \) operation by \( q \), such that \( t(\text{up}) < t_1 \). If \( \text{lin}(\text{sc}) < \text{lin}(\text{up}) \), then \( p \) does not execute line 99 throughout \( \text{ob} \).

Proof. In line 96 of \( \text{ob} \), process \( p \) reads a value \( x \) from \( \text{LU}[q].\text{key} \) at \( t_1 \), and in line 97 it reads a value \( y \) from \( \text{LS}[p] \). For the purpose of contradiction, assume \( p \) executes line 99 of \( \text{ob} \). Therefore, the if-statement in line 98 evaluates to true. Therefore, \( \bot \neq x < y \). By Observation 29, \( y = \text{asn}(\text{sc}) \). We will now show that \( x = \text{asn}(\text{up}) \), and then arrive at a contradiction.

By Lemma 20, the value of \( \text{LU}[q].\text{key} \) is \( \bot \) throughout \((t(\text{up}), t'(\text{up}))\), and is not modified to \( \text{asn}(\text{up}) \) at \( t'(\text{up}) \). Since \( x \neq \bot \), \( t_1 \notin (t(\text{up}), t'(\text{up})) \). Because \( t(\text{up}) < t_1 \), it follows that \( t'(\text{up}) < t_1 \). Also, by Lemma 20(d), \( \text{LU}[q].\text{key} \) remains \( \text{asn}(\text{up}) \) until process \( q \) performs \( \text{LU}[q].\text{SC}(\bot) \) in line 83 of a later \( \text{Update}() \) operation \( \text{up}' \) at point \( t(\text{up}') \). Hence, because \( \text{up} \) is the last \( \text{Update}() \) by \( q \) satisfying \( t(\text{up}) < t_1 \), we obtain \( x = \text{asn}(\text{up}) \). Since \( \text{lin}(\text{sc}) < \text{lin}(\text{up}) \), it follows from Claim 24 that \( y = \text{asn}(\text{sc}) < \text{asn}(\text{up}) = x \). This is a contradiction. \( \square \)

Claim 31. Suppose process \( p \) executes a \( \text{Scan}() \) operation \( \text{sc} \) and later an \( \text{Observe}(q) \) operation \( \text{ob} \), and no other \( \text{Scan}() \) between \( \text{sc} \) and \( \text{ob} \). Let \( t_1 \) be the point at which \( p \) performs \( \text{LU}[q].\text{LL}() \) in line 96 during \( \text{ob} \). Let \( \text{up} \) be the latest \( \text{Update}() \) operation by \( q \), such that \( \text{lin}(\text{up}) < \text{lin}(\text{sc}) \). Then \( t'(\text{up}) < t_1 \).

Proof. For the purpose of contradiction, assume \( t_1 \leq t'(\text{up}) \). By Claim 22, \( t(\text{up}) < \text{lin}(\text{up}) \). We also know that \( \text{lin}(\text{up}) < \text{lin}(\text{sc}) \) and \( \text{lin}(\text{sc}) < \text{rsp}(\text{sc}) < \text{inv}(\text{ob}) \). So \( t(\text{up}) < \text{inv}(\text{ob}) \). By Lemma 20(a) and the fact that \( t_1 \leq t'(\text{up}) \),
$LU[q],key = \perp$ throughout $[inv(ob),t_1)$. Process $p$ performs a $\text{HelpUpdate}(q)$ operation $hu$ after $inv(ob)$ and before it reads $LU[q]$ at $t_1$. Hence, $LU[q],key = \perp$ throughout the execution of $hu$. Therefore, process $p$ reads $\perp$ from $LU[q],key$ in line 90 of $hu$, and changes $LU[q]$ in line 93 to an integer fetched from $C$. This contradicts the fact that $LU[q],key = \perp$ throughout the execution of $hu$.

Lemma 32. Suppose process $p$ performs a $\text{Scan}()$ operation $sc$, and later an $\text{Observe}(q)$ operation $ob$, and no other $\text{Scan}()$ between $sc$ and $ob$. If process $q$’s latest $\text{Update}()$ that linearizes before $lin(sc)$ uses argument $v$, then $ob$ returns $v$.

Proof. Let $up$ be the process $q$’s latest $\text{Update}()$, such that $lin(up) < lin(sc)$. Let $t_1$ be the point in time at which process $p$ performs $LU[q],\text{LL}()$ in line 96 of $ob$. Process $p$ reads a value $(x,v')$ from $LU[q]$ at $t_1$ in line 96, and then reads a value $y$ from $LS[p]$ in line 97. By Claim 31, $t'(up) < t_1$. Also, by Observation 29, $y = \text{asn}(sc)$. We consider two cases.

Case 1: $LU[q]$ is not modified throughout $(t'(up),t_1]$.

By definition of $t'(up)$ and Lemma 20 (c), at $t'(up)$, $(\text{asn}(up),v)$ is written to $LU[q]$. Since $LU[q]$ is not changed throughout $(t'(up),t_1]$, $(x,v') = (\text{asn}(up),v)$ throughout this interval. Therefore, by the fact that $lin(up) < lin(sc)$ it follows from Claim 24 that $\text{asn}(up) = x < y = \text{asn}(sc)$. So the if-statement in line 99 of $ob$ evaluates to true and process $p$ returns $v$.

Case 2: $LU[q]$ is modified during $(t'(up),t_1]$.

Let $up^*$ be the last $\text{Update}()$ operation by $q$, such that $t(up^*) < t_1$. By Lemma 20 (d), $LU[q]$ is not changed after $t'(up)$ until process $q$ performs $LU[q],\text{SC}(\perp)$ in line 83 of another $\text{Update}()$ operation. Hence, $up$ happens before $up^*$. Since $up$ is the last $\text{Update}()$ operation by $q$, such that $lin(up) < lin(sc)$, it follows that $lin(sc) < lin(up^*)$. So by Lemma 30, process $p$ does not execute line 99 of $ob$, and hence executes line 101. Since process $q$ performs $LU[q],\text{SC}()$ in line 83 of $up^*$ before $t_1$, we have $rsp(up) < inv(up^*) < t_1$. Therefore, by Claim 27 and Lemma 28, process $p$ returns $v$.

Assuming the correctness of Lemmas 20 and 21, Lemma 32 completes the linearizability proof of Algorithm 5. Next, we prove Lemmas 20 and 21.
4.3.4 Proof of Lemma 20 and 21

The Proof of Lemmas 20 and 21 are symmetric to each other. Here, we only show the correctness of Lemma 20.

First, assuming the correctness of Lemma 20 (a), we prove part (b) to (d) of this lemma in Claims 33, 35, and 36. Then for each process \( p \), we consider all \texttt{Update()} operations by \( p \), and show by induction that each of these operations satisfies Lemma 20 (a).

**Claim 33.** Suppose process \( p \) performs an \texttt{Update}(\( v \)) operation \( up \). Let \( t^* \) be the point in time at which \( p \) reads \( LU[p] \) in line 85 of \( up \). If \( LU[p].\texttt{SC}(\bot, v) \) in line 83 of \( up \) succeeds, then \( t'(up) < t^* \).

*Proof.* For the purpose of contradiction, assume \( t^* \leq t'(up) \). Let \( hu \) be the \texttt{HelpUpdate}(\( p \)) operation, performed by process \( p \) in line 84 of \( up \). We show that \( LU[p].\text{key} \) is \( \bot \) and is not modified throughout the execution of \( hu \) and then arrive at a contradiction.

By the assumption of this claim, \( p \) changes \( LU[p].\text{key} \) to \( \bot \) at \( t(up) \). By definition, \( LU[p] \) is not modified throughout \((t(up), t'(up))\). Therefore, since \( t(up) < t^* \leq t'(up) \), \( LU[p] = \bot \) and is not modified throughout \((t(up), t^*)\). Note that the entire execution of \( hu \) is in \((t(up), t^*)\).

During the execution of \( hu \), \( p \) reads \( \bot \) from \( LU[p].\text{key} \) in line 90, and later changes \( LU[p].\text{key} \) in line 93 with an \texttt{SC()} operation. This contradicts the fact that \( LU[p] \) remains unchanged throughout the execution of \( hu \).

**Observation 34.** Suppose process \( p \) first performs an \texttt{Update}(\( v \)) operation \( up \) and then another \texttt{Update()} operation \( up' \) and no \texttt{Update()} operation in-between. Then throughout \((t(up), t(up'))\), the only way, in which a process can change \( LU[p] \) is the \( LU[p].\texttt{SC()} \) operation in line 93 of a \texttt{HelpUpdate}(\( p \)) operation.

*Proof.* The value of \( LU[p] \) can be modified with an \( LU[p].\texttt{SC()} \) operation either in line 93 of a \texttt{HelpUpdate}(\( p \)) operation or line 83 of an \texttt{Update()} operation by process \( p \). However, by definition, process \( p \) does not perform line 83 of an \texttt{Update()} operation throughout \((t(up), t(up'))\). Hence, this observation is correct.

\( \square \)
Claim 35. Suppose process $p$ performs an $\text{Update}(v)$ operation $up$. If $LU[p].\text{SC}(\bot, v)$ in line 83 of $up$ succeeds, then at $t'(up)$ some process performs a successful $LU[p].\text{SC}(x, v)$ operation in line 93 of a $\text{HelpUpdate}(p)$ call, where $x \in \mathbb{N}$.

Proof. By Claim 33 and definition, $t'(up)$ occurs in $(t(up), rsu(up))$. Therefore, by Observation 34, at $t'(up)$ some process performs a successful $LU[p].\text{SC}(x', v')$ operation in line 93 of a $\text{HelpUpdate}(p)$ call. Suppose $hu$ is that $\text{HelpUpdate}(p)$ operation and $q$ is the process calling $hu$. Process $q$ fetches $x'$ by performing $C.FAI()$ in line 92. Therefore, $x' \in \mathbb{N}$. Thus, it remains to prove that $v' = v$.

For the purpose of contradiction, assume $v' \neq v$. By the assumption of this claim, at point $t(up)$, $LU[p].val$ is changed to $v$. Therefore, by the fact that $LU[p]$ is not modified throughout $(t(up), t'(up))$, $LU[p].val = v$ in this interval. Let $t^*$ be the point in time at which $q$ reads $v'$ from $LU[p].val$ in line 90 of $hu$. By the fact that $v' \neq v$, $t^* < t'(up)$, and $LU[p] = v$ throughout $(t(up), t'(up))$, $t^* < t(up)$. So process $q$ performs $LU[p].LL()$ at $t^* < t(up)$ and afterward a successful $LU[p].\text{SC()}$ at $t'(up) > t(up)$. However, at $t(up)$ a successful $LU[p].\text{SC()}$ occurs. This contradicts the fact that the $LU[p].\text{SC()}$ at $t'(up)$ is successful.

Claim 36. Suppose process $p$ first performs an $\text{Update}(v)$ operation $up$ and then another $\text{Update}(v')$ operation $up'$ and no $\text{Update()}$ operation in-between. If $LU[p].\text{SC}(\bot, v)$ in line 83 of $up$ succeeds, then $LU[p]$ is not modified throughout $(t'(up), t(up'))$.

Proof. For the purpose of contradiction, assume $LU[p]$ is modified during $(t'(up), t(up'))$. Let $q$ be the first process that changes $LU[p]$ after $t'(up)$. By Claim 33 and definition, $t'(up)$ occurs after $t(up)$ and before $rsu(up)$. So by Observation 34, $q$ changes $LU[p]$ in a $\text{HelpUpdate}(p)$ operation. Let $hu$ be that $\text{HelpUpdate}(p)$ operation, $t_1$ the point in time at which $q$ reads $x$ from $LU[p].\text{key}$ in line 90 of $hu$ and $t_2$ the point in time at which $q$ performs a successful $LU[p].\text{SC()}$ in line 93. By the fact that $q$ is the first process that changes $LU[p]$ after $t'(up)$, $t_2 > t'(up)$. Hence, by Claim 35, $LU[p].\text{key} \neq \bot$ throughout $(t'(up), t_2)$. Since process $q$ performs $LU[p].\text{SC()}$ in line 93 of $hu$, the if-statement in line 91 evaluates to true ($x = \bot$). Therefore, considering that $LU[p] \neq \bot$ throughout $(t'(up), t_2)$ and $t_1 < t_2$, $t_1 < t'(up)$. So process $q$ performs $LU[p].LL()$ at $t_1 < t'(up)$ and afterward a successful $LU[p].\text{SC()}$ at
Claim 37. Let \( p \) be a process. Then the \( LU[p].SC() \) operation in line 83 of every \( \text{Update()} \) operation by \( p \) succeeds.

Proof. Let \( up_1, up_2, \ldots \) be the \( \text{Update()} \) operations by \( p \), ordered by invocation time. Let \( t^*_i \) be the point in time at which process \( p \) performs \( LU[p].LL() \) in line 82 of \( up_i \). In its next step after \( t^*_i \), process \( p \) performs \( LU[p].SC() \) in line 83 of \( up_i \) at \( t(up_i) \). Therefore, if \( LU[p] \) is not modified throughout \( [t^*_i, t(up_i)) \), then \( LU[p].SC() \) of \( up_i \) succeeds. We prove by induction that \( LU[p] \) remains unchanged throughout \( [t^*_i, t(up_i)) \), for \( i \geq 1 \).

Base Case: We show that \( LU[p] \) is not modified throughout \( [t^*_1, t(up_1)) \).

Recall that \( E \) starts with \( n \) sequential \( \text{Update()} \) operations, one called by each process in \( \{0, \ldots, n-1\} \). Therefore, \( p \) performs \( up_1 \) solo and hence, \( LU[p] \) is not modified throughout \( [t^*_1, t(up_1)) \).

Let \( k \) be a positive integer greater or equal than 1.

Inductive Hypothesis: \( LU[p] \) is not modified throughout \( [t^*_k, t(up_k)) \).

Inductive Step: We show that \( LU[p] \) is not modified throughout \( [t^*_k+1, t(up_{k+1})) \).

Since \( LU[p] \) is not modified throughout \( [t^*_k, t(up_k)) \), \( LU[p].SC() \) in line 83 of \( up_k \) succeeds. Therefore, by Claim 36, \( LU[p] \) remains unchanged throughout \( (t'(up_k), t(up_{k+1})) \). Based on Claim 33, \( t'(up_k) < rsp(up_k) < t^*_k+1 \). Hence, \( LU[p] \) is not modified throughout \( [t^*_k+1, t(up_{k+1})) \). \( \square \)

4.4 Analysis

First, we show in Lemma 38 that a capacity bound of \( \Delta = 3n \) is sufficient for our single-writer predecessor objects. Then we analyze the time complexity of \( \text{Prune()} \) operations in Lemma 39, and afterwards, we prove our main theorem.

Lemma 38. For any process \( p \), at any point in time, the number of pairs stored in \( S[p] \) does not exceed \( 3n \).

Proof. First, we show that at the response of each \( \text{Prune()} \) operation by process \( p \), at most \( 2n \) pairs are stored in \( S[p] \). Then we prove the \( 3n \) bound for any point in time.
Let \( pr \) be a Prune() operation by \( p \). Consider the local set \( E \) that is initialized to \( \emptyset \) in line 113 of \( pr \). In each iteration of the for-loop starting in line 115, at most one element is inserted into \( E \). Since there are \( n \) iterations, \(|E| \leq n\). Let \( t^* \) be the point in time at which \( p \) reads last in line 114 of \( pr \). By Claim 24 and Claim 25, the pairs that are inserted into \( S[p] \) after \( t^* \) have keys greater than last. Let \( U \) be the set of pairs in \( S[p] \) at \( t^* \). Then in line 124, all pairs in \( U \setminus E \), except for the one with the largest key (last), are removed from \( S[p] \). Since \(|E| \leq n\), at most \( n+1 \) elements from \( U \) are not removed.

Moreover, process \( p \) distributes the work of \( pr \) over \( n \) Update() operations \( up_1, up_2, \ldots, up_n \). The invocation of \( pr \) is after the point in time at which \( p \) inserts a pair into \( S[p] \) in \( up_1 \) and \( rsp(pr) \) is before \( rsp(up_n) \). Therefore, during \( pr \), \( p \) inserts \( n-1 \) pairs into \( S[p] \). Hence, the number of elements in \( S[p] \) at \( rsp(pr) \) is at most \( 2n \).

Suppose \( up^* \) is the 2\( n \)-th Update() operation by process \( p \). By the fact that during each Update() operation by \( p \), one pair is inserted into \( S[p] \), at any point in time before \( rsp(up^*) \) there are at most 2\( n \) pairs in \( S[p] \). Thus, it remains to prove that for any point in time after \( rsp(up^*) \) the number of elements in \( S[p] \) is at most 3\( n \).

Let \( t \) be a point in time after \( rsp(up^*) \). Suppose \( pr \) is the last Prune() call by \( p \), such that \( rsp(pr) \leq t \). Such a Prune() operation exists because the response of the first Prune() call by \( p \) is before \( rsp(up^*) \). By the fact that \( p \) calls a new Prune() operation every \( n \) Update() operations, at most \( n \) pairs are inserted into \( S[p] \) during \([rsp(pr), t]\). Therefore, since the number of elements in \( S[p] \) at \( rsp(pr) \) is at most 2\( n \), there are at most 3\( n \) pairs in \( S[p] \) at \( t \).

It follows that we can use our single-writer predecessor object from Section 3 with bounded capacity \( \Delta = 3n \).

To analyze the time complexity of a Prune() operation, we need to determine the time complexity of each operation on our building blocks. Lemma 38 together with Theorem 2 yield an \( O(\log n) \) time complexity for each operation on a predecessor object. For the LL/SC objects LU and LS, as discussed in Section 1.3, we use the implementation of Blelloch and Wei [14]. Hence, each operation on \( LU[p] \) and \( LS[p] \), for a process \( p \), takes constant time.

**Lemma 39.** The time complexity of a Prune() operation is \( O(n \log n) \).

**Proof.** Let \( p \) be a process and \( pr \) a Prune() operation by \( p \). We use a sequential
set, implemented by a balanced binary search tree for the local set \( E \) that is initialized in line 113 of \( pr \). In each iteration of the for-loop starting in line 115, at most one element is inserted into \( E \), so \(|E| \leq n\). Therefore, inserting and finding a pair in \( E \) takes \( O(\log n) \) time.

Each \( \text{HelpScan()} \) operation in line 116 and each \( \text{LL()} \) operation in line 117 takes constant time. The \( \text{Pred()} \) operation on \( S[p] \) in line 119 takes \( O(\log n) \) time, and so does adding the predecessor to \( E \). Hence, the for-loop comprising lines 115-119, takes \( O(n \log n) \) time.

By Claim 24 and Claim 25, all pairs that are inserted into \( S[p] \) after \( p \) reads \( \text{last} \) in line 114 of \( pr \) have keys greater than \( \text{last} \). Let \( U \) be the set of pairs in \( S[p] \) with key less than \( \text{last} \). By Lemma 38, \(|U| \in O(n)\). In the while-loop from line 121 to 125, \( p \) traverses through the pairs in \( U \) by calling successor of elements starting from the pair with the smallest key (\( S[p].\text{Succ}(\text{−∞}) \)). Performing a \( \text{Succ()} \) in line 122, finding a pair in \( E \) in line 123, and removing a pair from \( S[p] \) in line 124 each takes \( O(\log n) \) time. Therefore, since \(|U| \in O(n)\), the total amount of work for the while-loop is \( O(n \log n) \). So the time complexity of \( pr \) is also \( O(n \log n) \).

Claim 40. If the total number of \( \text{Update()} \), \( \text{Observe()} \), and \( \text{Scan()} \) operations is less than \( 2^W/5 \), then the value of \( C \) is less than \( 2^W - 4 \) throughout the execution.

Proof. The initial value of \( C \) is 0. Also, the only lines in the pseudocode in which \( C \) is incremented are line 92 of a \( \text{HelpUpdate()} \) operation, and line 110 of a \( \text{HelpScan()} \) operation. In the following, we show that when a process \( p \) performs an \( \text{Update()} \), an \( \text{Observe()} \), or a \( \text{Scan()} \) operation, \( p \) calls \( \text{HelpUpdate()} \) and \( \text{HelpScan()} \) at most 5 times. Therefore, if the number of \( \text{Update()} \), \( \text{Observe()} \), and \( \text{Scan()} \) operations is less than \( 2^W/5 \), then \( C \) is incremented at most \( 2^W - 5 \) times and thus, has a value less than \( 2^W - 4 \).

Let \( p \) be a process, \( sc \) a \( \text{Scan()} \) operation by \( p \), \( ob \) an \( \text{Observe()} \) operation by \( p \), and \( up \) an \( \text{Update()} \) operation by \( p \). During each of \( sc \) and \( ob \), \( p \) calls \( \text{HelpScan()} \) or \( \text{HelpUpdate()} \) exactly once (in line 106 or 95). In \( up \), \( p \) calls \( \text{HelpUpdate()} \) in line 84, and then contributes \( O(\log n) \) steps towards an ongoing \( \text{Prune()} \) call. As discussed above, the contribution of \( p \) in the ongoing \( \text{Prune()} \) call consists of performing 4 iterations of the for-loop and/or while-loop. In each of those iterations, process \( p \) may perform a \( \text{HelpScan()} \) call (in line 116). Therefore, \( p \) may call \( \text{HelpScan()} \) and \( \text{HelpUpdate()} \) at most 5 times in \( up \).
Proof of Theorem 1. Linearizability follows immediately from Lemma 32.

**Time complexity**  As discussed earlier in this section, each operation on an LL/SC object or a predecessor object takes $O(1)$ or $O(\log n)$ time, respectively. Also, since $C$ is a base object, each operation on it takes constant time. Therefore, the time complexity of methods `HelpScan()` and `HelpUpdate()` is $O(1)$. Hence, the time complexity of `Scan()` is $O(1)$, and the time complexity of `Observe()`, which is dominated by an operation on a predecessor object, is $O(\log n)$. The time complexity of an `Update()` operation, disregarding the contribution to an ongoing `Prune()` call, is $O(\log n)$. By Lemma 39, the time complexity of a `Prune()` operation is $O(n \log n)$. Since we distribute the work of every `Prune()` call over $n$ `Update()` operations, the additional work in each `Update()` operation is $O(\log n)$. Thus, the time complexity of `Update()` is $O(\log n)$.

**Number of base objects and bit complexity**  Without loss of generality we assume that, for a process $p$, $LS[p]$ is a double-word LL/SC object.

Theorem 1 assumes that only up to $2W/5 - 1$ `Update()`, `Scan()` and `Observe()` operations can be executed on our object. Therefore, in the following, we assume that the number of operations that are executed on our object is less than $2W/5$.

In our implementation, we use a shared FAI object, $2n$ double-word LL/SC objects, and $n$ predecessor objects with bounded capacity $\Delta = 3n$. First, we investigate what we store in each of those objects. Let $D_c = \{0, \ldots, 2^W - 4\} \cup \{\perp\}$. By Claim 40, the value of $C$ is in $D_c \setminus \{\perp\}$ and thus, the following holds for each process $p$.

(a) $\forall p \in \{0, \ldots, n - 1\}$ $LS[p] \in D_c$ (see lines 105 and 111.),
(b) $\forall p \in \{0, \ldots, n - 1\}$ $LU[p] \in D_c \times D_{snp}$ (see lines 83 and 93), and
(c) for every pair $x$ in $S[p]$, $x.key \in D_c$ and $x.val \in D_{snp}$ (see line 86).

Therefore, $D_{pred,key} = D_c$, $D_{pred,val} = D_{snp}$, and $D_{llsc} = (D_c \times D_{snp}) \cup D_c$.

By Theorem 2, each predecessor object with bounded capacity $\Delta = 3n$ can be constructed from $O(n^2 \log n)$ CAS objects and registers, each of size $W$ bits, where $W \geq \max\{\log |D_{pred,key}|, \log |D_{pred,val}|, 3 \log n\} + O(1)$. Also, as discussed in Section 1.3, using the implementation of Blelloch and Wei [14], $2n$ double-word LL/SC objects can be constructed from $O(n^3)$ CAS objects.
and registers, each of size $W$ bits, where $W \geq \max\{\log|D_{llsc}|/2, 2\log n\} + O(1)$. Hence, we can construct our adaptive partial snapshot object from $O(n^3 \log n)$ base objects (FAI, CAS, and registers) of size $W$ bits, if $W \geq \max\{\log |D_c|, \log |D_{snp}|, 3 \log n\} + O(1)$. Since $|D_c| < 2^W$, $W \geq \log |D_c|$. Therefore, it suffices that $W \geq \max\{\log |D_{snp}|, 3 \log n\} + O(1)$.

☐
Chapter 5

Conclusion

We implemented a single-writer adaptive partial snapshot object, which allows a process to adaptively read a consistent view of $k \in \{1, \ldots, n\}$ memory components in $O(k \log n)$ time. The advantage of our algorithm over poly-logarithmic snapshot implementations is the realistic assumptions about the base objects’ size. Moreover, as a building block to our snapshot object, we implemented a single-writer predecessor object with bounded capacity $\Delta$, which may be of independent interest.

We showed in Chapter 3 that we can use the sequential persistent red-black tree of [15] as a single-writer predecessor object. However, in infinite executions, this implementation requires an unbounded number of base objects. Although many general frameworks are provided for memory reclamation of concurrent data structures (such as [23, 24]), none of them can be applied in our predecessor object: In our predecessor object, when a process performs an update operation, it creates a duplicate of a path of the tree, as opposed to the entire tree. Hence, the nodes that are created during the $i$-th update operation may have pointers to the nodes that are created during the $j$-th update operation, where $j < i$. Because of this property of our algorithm, we cannot use the general memory reclamation frameworks. Thus, we devised a novel memory reclamation technique that accounts for the pointers between the nodes that are created during different update operations. We believe that our technique can be generalized to obtain an efficient framework for the memory management of concurrent data structures.

A weakness of our snapshot algorithm is using unbounded sequence numbers. We increment the value of a FAI object at most 5 times with each $\text{Update}()$, $\text{Scan}()$, and $\text{Observe}()$. Hence, using base objects of size $W$
bits (single-word base objects), our snapshot object supports up to \(2^W/5 - 1\) \texttt{Update()}, \texttt{Scan()} and \texttt{Observe()} operations. The other base objects that we use in our algorithm are registers and CAS objects, each of which has efficient multi-word implementations. That is, we can implement a CAS object/register of size \(kW\) bits (\(k\)-word) from single-word CAS objects/registers, such that every operation takes \(O(k)\) times. Therefore, if we can construct an efficient \(k\)-word FAI object from single-word base objects (CAS, FAI, and register), then our algorithm can support up to \(2^{kW}/5 - 1\) operations, where \(k\) can be arbitrarily large. Ellen and Woelfel [16] implemented a single-word FAI object from single-word LL/SC objects and registers, where each operation takes \(O(\log n)\) steps. Their work can be combined with the \(k\)-word LL/SC implementation of [14] to obtain a \(k\)-word FAI object with step complexity of \(O(k \log n)\) for each operation. Using their object, we can obtain a single-writer adaptive partial snapshot object that supports up to \(2^{kw}/5 - 1\) operations, where the step complexity of each operation is \(O(k \log n)\). We conjecture that using single-word FAI objects, CAS objects, and registers, one can implement a \(k\)-word FAI object with step complexity of \(O(k)\) for each operation. An interesting open problem is to prove or disprove this conjecture.

Even when using FAI objects of size \(kW\) bits, our algorithm is not satisfying from a theoretical point of view because there is an upper bound on the number of operations that our object can support. We believe that the FAI object in our implementation can be replaced with a bounded timestamp system. However, no practical bounded timestamp systems are known that could replace the unbounded FAI in our algorithm without significantly reducing efficiency.

An important open problem is to extend our algorithm to the multi-writer version, where each process can update any component of the snapshot array. The next step is to generalize the multi-writer version to support snapshots of other base objects such as CAS (similar to [27]).
Bibliography


68