Dynamic Wait-Free Atomic Snapshot

by

Jamilush Talukder

A Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
of the University of Manitoba
in partial fulfillment of the requirements of the degree of

Master of Science

Department of Computer Science
University of Manitoba
Winnipeg

Copyright © 2025 by Jamilush Talukder

Abstract

Efficiently implementing wait-free atomic snapshot objects is one of the core problems in the shared memory model of the theory of distributed computing. An atomic snapshot object gives users access to a set of n shared registers and two types of operations to perform on them. One of the operations, called *update*, allows a process to update the data in one of the n registers. Another operation, called *snap*, allows a process to read the values of all n registers. Despite operating in an asynchronous environment, these operations should appear atomic, i.e., their outcomes should be consistent with an execution in which the operations run with no interruptions by other operations. We want the implementation to be wait-free, which means that the progress of any update or snap operation should not be paused due to any other operation that is being executed by any other process. For example, the implementation should not use any kind of locking mechanisms on variables. A dynamic version of this problem is where the n registers can be dynamically added or removed without needing to re-initialize the atomic snapshot object. In this thesis, we provide an efficient algorithm for dynamic atomic snapshot.

Contents

1 Introduction **7**
- 1.1 Model . . . 9
  - 1.1.1 Process . . . 9
  - 1.1.2 System-level Atomic Operations . . . 9
  - 1.1.3 Atomic Read-Write Register . . . 9
  - 1.1.4 Scheduler . . . 10
- 1.2 Problem Definition . . . 10
  - 1.2.1 Atomic Object . . . 10
  - 1.2.2 Atomic Snapshot Object . . . 11
  - 1.2.3 Linearizability . . . 13
- 1.3 Our Results . . . 15

2 Literature Review **16**
- 2.1 Atomic Object . . . 16
- 2.2 Atomic Snapshot . . . 17
- 2.3 Dynamic Atomic Snapshot . . . 21
- 2.4 Anonymous Atomic Snapshot Algorithms . . . 21

3 Preliminaries **22**
- 3.1 System Restrictions . . . 22
- 3.2 Definitions . . . 24
  - 3.2.1 Memory and Data Structures . . . 24
  - 3.2.2 Domination . . . 28
  - 3.2.3 Merge . . . 28
  - 3.2.4 SeqSum . . . 29
  - 3.2.5 Partial Ordering . . . 29
  - 3.2.6 Data Types . . . 29
- 3.3 Preliminary results and observations . . . 32
  - 3.3.1 Relating Domination and seqSum . . . 32
  - 3.3.2 Merging preserves domination . . . 34
  - 3.3.3 Partial ordering of Add operations . . . 36

3.4 Performance Metrics . . . 36

4 Algorithm code **37**
4.1 Algorithm: UPDATE . . . 37
4.2 Algorithm: ADD . . . 40
4.3 Algorithm: REMOVE . . . 43
4.4 Algorithm: SNAP . . . 46
4.5 Algorithm: CLASSIFIER . . . 49
4.6 Algorithm: ADDTOCLASSIFIERTREE . . . 52
4.7 Algorithm: GETUPDATEDACTIVELIST . . . 53
4.8 Algorithm: GETMERGERECENT . . . 57
4.9 Algorithm: GETSEQSUM . . . 59

5 Algorithm Observations **60**
5.1 Common anchor points in the Algorithms: 1, 2, 3, 4 . . . 60
5.2 Remove Operation Observations . . . 62
5.3 Add Operation Observations . . . 65

6 Linearization **66**
6.1 High-Level Idea . . . 67
6.2 Linearization Definition . . . 68
6.3 Total Ordering . . . 72
6.3.1 a, b and c are all snap operations . . . 72
6.3.2 a, b and c are modify operations within the same bucket 74
6.3.3 a and c are modify operations and b is snap operation 74

7 Proof of correctness **76**
7.1 Terminology . . . 76
7.2 Validity of Snap Operations . . . 76
7.2.1 Snap Validity 1: Contains the values written by the most recent update operation . . . 77
7.2.2 Snap Validity 2: Contains the added processes that have not been removed . . . 87

8 Analysis of Running Time **107**
8.1 Analysis of Add . . . 107
8.2 Analysis of Remove, Snap, and Update . . . 108
8.3 Analysis of AddToClassifierTree . . . 108
8.4 Analysis of Classifier . . . 109
8.5 Analysis of getSeqSum . . . 109
8.6 Analysis of GetMergeRecent . . . 109
8.7 Analysis of GetUpdatedActiveList . . . 109
8.8 Worst-Case Costs . . . 110

8.8.1 Add Operation 110
8.8.2 Remove, Snap and Update Operations 110

9 Supporting Technical Lemmas 110
9.1 New Lemmas 111
9.2 Lemmas from Attiya and Rachman [16] 138

10 Conclusion and Future Work 156
10.1 Future Work 157

Bibliography 159

Acknowledgements

I would like to start by expressing my sincere gratitude to my advisor, Dr. Avery Miller, for his continuous support, patience, and immense knowledge throughout the course of this research. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my thesis. I would also like to thank Dr. Stephane Durocher and Dr. Parimala Thulasiraman for serving on my thesis committee and providing valuable feedback. Last by not least, I would like to thank my wife Jereen for her endless support and encouragement throughout my studies, and my parents for always believing in me.

This thesis is dedicated to those who never gave up.

1 Introduction

Atomic snapshot primarily solves the difficult problem of taking an instantaneous picture of a collection of shared objects while simultaneous asynchronous updates are being made to the objects. To understand the significance of the problem, we can examine a simpler version where we have a simple array of data which is shared by multiple processes. There are two operations possible on this array. One of the operations is the *update* operation, where a process can update any element of the array with a value. Another operation is the *snap* operation, where a process can get the latest values of all the elements of the array in a single function call. In the simplest version of the problem (which we are familiar with in traditional/centralized computational models), the update and snap operations are synchronous which means only a single operation can be performed on the array at any given time. The opposite of synchronous operation is asynchronous operation, where multiple overlapping operations are allowed to be executed at any given time.

Atomic snapshot objects are widely used to greatly simplify the design and proof of correctness of other concurrent algorithms, such as wait-free implementation of data structures [12], approximate agreement [15], construction of atomic multi-writer multi-reader registers [47, 42], randomized consensus [2, 11, 13], concurrent time-stamp systems [21] and exclusion problems [20, 35, 36].

The problem of taking an instantaneous picture of a shared object is trivial when the update and snap operations are only allowed to be performed synchronously because it is possible to update any element of an array in $O(1)$ time and read all the values of the array in $O(n)$ time. However, as soon as the operations become asynchronous, it becomes harder to track the values. For example, during the execution of the snap operation by a process A, an update operation by another process B could also occur. An atomic snapshot object should be able to handle these cases in a consistent manner; more specifically, the output of the snap operation should be a version of the array that actually existed at some point in time during the execution of the snap operation. Consider a first (and incorrect) attempt at implementing the snap operation that simply scans the array (let the array be arr) locations in

sequential order. The problem with this approach is that if there is an update to $arr[0]$ after process A has read $arr[0]$, and then an update to $arr[2]$ before process A has read $arr[2]$, then the outcome of A's snap operation will be an array of values that never actually existed at any point in time. An equivalent way of stating the desired property of a snap is to require that the operation is atomic, which means from process A's point of view, it seems as though the snap operation was carried out without being interrupted by other operations (despite the fact that operations are in fact asynchronous/overlapping).

Another requirement is that the progress of any update or snap operation should not be paused due to any other operation that is being executed by any other process. This property is known as the *wait-free* property. In a synchronous shared memory all operations are trivially wait-free. Consider the atomicity issue that we talked about in the previous paragraph: one way to solve that issue would be to introduce a *lock* mechanism, where process A could set a lock bit at each array location in arr before it reads it, and then unsets all the lock bits after it is done reading the whole array. When an array location is locked other processes are not allowed to update it. While it is a simple enough solution, this is clearly not wait-free, since update operations might be paused while a snap is being executed. The wait-free property becomes important when a process is slow or crashes during the execution of an operation, and it is expected that the overall application will continue performing the rest of the operations. For this reason, using locks or busy-waiting must be avoided in our solutions.

The *dynamic atomic snapshot* supports addition and removal of processes. As a result, along with the update and snap operation, the dynamic atomic snapshot object has add and remove operations. Highly scalable distributed applications such as Hadoop [48], Spark [27] and distributed block chains [40, 18] are some of the example which can benefit from dynamic atomic snapshot algorithms. This will be the primary focus of our research. We will aim to come up with a wait-free algorithm that solves the dynamic atomic snapshot problem in distributed shared memory.

1.1 Model

1.1.1 Process

A *process* is an entity that can run code. This code consists of “local instructions”, which can be carried out completely by the process’ CPU and memory, as well as “non-local instructions”, which attempt to access remote shared memory, and these instructions must wait for such memory accesses to be carried out by the system. We consider systems that contain many processes executing their code simultaneously, and each process is identified using a unique integer number. We will usually use the notation p_i to identify process i.

1.1.2 System-level Atomic Operations

An *atomic operation* is a function that, from the caller’s perspective, is not interrupted during its execution. The only system-level atomic operations in our model are reads and writes to atomic registers, as described in the next section.

1.1.3 Atomic Read-Write Register

An *atomic read-write register* is a simple shared memory object that supports atomic read and atomic write operations. Other useful atomic shared memory objects can be implemented using atomic read-write registers, such as a shared array or a shared binary search tree. The read-write operations performed using read-write registers are atomic operations. A *single-writer multi-reader (SWMR)* register allows all processes to read from it, but only one designated process to write to it. On the other hand, using a *multi-writer multi-reader (MWMR)* register allows all process to read from it and write to it. In our system model, we assume that shared memory consists only of SWMR atomic read-write registers. These registers are among the simplest to implement in hardware, so algorithms designed for this model are very widely applicable to real shared memory systems.

1.1.4 Scheduler

To simulate real-world asynchronous behavior during the execution of the operations in our model, when a process p_i reaches a line where it tries to read/write to shared memory, it can immediately get access to the memory, or it can be frozen until later. This decision is made by the *Scheduler*. The Scheduler decides which process (if any) is allowed to carry-out their access to shared memory, and all other processes that are attempting to access shared memory are frozen. Since our shared memory consists of atomic read/write registers, once the Scheduler allows a process to access the shared memory, the process gets to complete its read or write to a single register before any other shared memory operation is allowed by the Scheduler. This is because the system level atomic operations in our model are reads and writes to atomic registers.

We consider the Scheduler to be *adversarial*, i.e., it knows our algorithm's code and can choose to schedule processes in a worst-case scenario manner in an attempt to make our algorithm perform incorrectly or inefficiently. However, the Scheduler must be *fair*, which means all attempted accesses to shared memory must be eventually scheduled (i.e., a process may be frozen for a long time, but cannot be frozen forever). For this reason, the performance of an algorithm is measured by the number of operations carried out by a process, not by the amount of time it takes for a process to carry them out.

1.2 Problem Definition

The goal of this thesis is to provide an implementation of a dynamic atomic snapshot object. In this section, we will define the dynamic snapshot object, define what it means for the object to be atomic, and define a correctness condition, i.e., linearizability.

1.2.1 Atomic Object

An *atomic object* is a simple form of any data on which one or more processes can perform atomic operations. More specifically, an atomic object consists of two

parts: a *data* part and a *code* part. The *data* part consists of some amount of shared storage in memory and the *code* part consists of one or more atomic operations that manipulates the *data* part. For example, the *data* part could be a shared register and the *code* could consist of an atomic read operation and an atomic write operation on the register. It is important to note that the operations supported by the atomic object aren't necessarily atomic, but they seem atomic from the perspective of the caller. When designing an atomic object, one has to prove that this atomic property holds. For example, atomic read-write registers are as an example of an atomic object, but no proof of atomicity is required since the registers are assumed to be atomic in the model.

1.2.2 Atomic Snapshot Object

The basic object in our shared memory model (the SWMR atomic register) allows a process to read from one memory location per shared memory access. We are interested in extending this capability by implementing a shared memory object that allows a process to capture a "snapshot" of the entire memory as it appears at one moment in time. This is called an *atomic snapshot object*, which is defined as follows. The *data* part consists of n *atomic read-write registers* and the *code* part consists of an implementation of an *update* operation and a *snap* operation. Each process p_i has a dedicated SWMR register i to which it is allowed to write. During an update operation, a process p_i writes an arbitrary value v to its dedicated register. During the snap operation, an array of size n is returned containing the updated values stored in the n registers. The implementations of the Update and Snap operations can involve many reads and writes to shared registers, e.g., the Snap operation likely involves at least n reads. Despite the Scheduler's ability to choose the ordering and delay of accesses to the shared registers, we want the Update and Snap operations to appear as atomic to the processes that call them, i.e., as if they had occurred without any interruptions (more about this in Section 6). We will implement a more general version of this object, called a *dynamic atomic snapshot object*, in which the value of n is not fixed in advance. In particular, processes may join or leave the system, and

while in the system, each process p_i is able to update values in its own designated register i, and can perform a snap operation to retrieve the register values belonging to all processes that have not left the system. They would also be able to add new processes or remove the processes that they added. We call a process "active" if it has been added to the system, but not yet removed. To implement this object, our solution must support 4 operations which can be invoked by any active processes:

1. Snap: Returns the values of all registers belonging to active processes in the system.
2. Update: Given a value as input, it writes the value to the register that belongs to the process that is executing the operation. Does not return any value.
3. Add: Given a process ID as input, it adds the process to the system. Does not return any value.
4. Remove: Given a process ID as input, it removes the process from the system. Does not return any value.

We highlight the fact that, since our shared memory model consists of SWMR registers, we are actually implementing the *single-writer multi-reader (SWMR) atomic snapshot object*, since each active process in the system is only allowed to write values to a single register dedicated to it, and so the number of registers is equal to the number of processes. A more general object, called the *MWMR atomic snapshot object*, would allow each process to write to any register, and so this allows situations where the number of registers is different from the number of processes. Interestingly, MWMR atomic registers can be simulated in SWMR systems once the SWMR atomic snapshot object has been implemented [8].

In this thesis, we consider the restricted *m-shot atomic snapshot object* that, for some known m, only allows a total of at most m of the above 4 operations to be called. A more general *∞-shot atomic snapshot object* would allow an unbounded number of operations, and we leave this generalization for future work.

1.2.3 Linearizability

In asynchronous systems, it is often difficult to define and prove the "correctness" of an implementation. Informally, we want "cause and effect" to be observed by any process calling the Update, Snap, Add, and Remove operations. This is simple to achieve if the Scheduler acts in a nice way, e.g., when processes p_i and p_j try to execute operations, the Scheduler allows p_i to perform all the shared memory accesses it needs, and then allows p_j to perform all the shared accesses it needs, and so p_j's entire operation occurs after p_i's entire operation, and anything done by p_i is observable by p_j. However, asynchronous systems do not execute operations in a nice way, and the adversarial behavior of our Scheduler will make sure of this. In particular, the adversarial Scheduler can freeze a particular process' operation partway through by preventing it from performing a shared memory access for a long time, and so there can be many simultaneous operations that overlap, and this means there is no clear way to define which operations happen "before" other operations. One method of proving that "cause and effect" is maintained by the implementation is to formally prove that the implementation satisfies a property called linearizability.

The *linearizability* property of a snapshot object implementation ensures that, in every execution of the algorithm regardless of the Scheduler's choices, all the operations can be ordered by assigning each operation a distinct point in time that occurs somewhere between the actual start and end of the operation, and when these points in time are placed on a line and we imagine a sequential execution of these operations from left to right along the line, the outcomes of the operations are the same as the concurrent execution of our object. This action of mapping an operation to a point in time on the line is called *linearizing*.

First, as a simple example, if the Scheduler ensures that no two operations overlap in time, then it is simple to prove that the linearizability property holds: for example, we could map each operation to its end time. However, things get more complicated when the Scheduler forces two or more operations to overlap in time. For example, if two operations a and b overlap each other where b is an Update operation and a is a Snap operation and the modifications made during b is found in the result

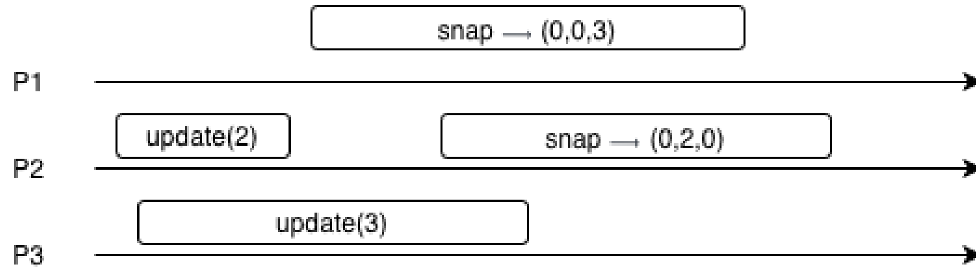

Figure 1: Example of a non-linearizable snapshot implementation. There are 4 operations here which are being executed by 3 processes. p_1 is executing a snap where the result is (0,0,3), p_2 is executing an update(2) on the second register and a snap where the result is (0,2,0), and, p_3 is executing an update(3) on the third register.

of a, then, the linearization mapping should ensure that a is linearized after b (for example, operation a could be linearized to a point in time t_a that is very late in a's execution, and it could linearize operation b to a point in time t_b that is very early in b's execution such that $t_b < t_a$).

To show why this is non-trivial, consider the three processes in the Figure 1. For each operation, we can try to pick an instant in time between the start and end of the operation and pretend that the operation occurs instantly at that point, but, no matter how we choose such points, we notice that at least one of the snap outputs is returning an invalid result. In particular, if update(2) is ordered before update(3), then the snap that reports a value of 3 in the 3rd register would also need to report a value of 2 in the 2nd register. Similarly, if update(3) is ordered before update(2), then the snap operation that reports a value of 2 in the 2nd register would also need to report a value of 3 in the 3rd register. Instead, if the implementation had P1's snap operation return (0,2,3) in this particular execution, then a valid ordering would be: update(2) by P2, snap by P2, update(3) by P3, snap by P1. Our implementation of the atomic snapshot object will avoid all possible errors like the one illustrated in Figure 1, and this is achieved by proving that the implementation is linearizable.

Ensuring that an implementation satisfies linearizability has the following three main components:

1. Designing the implementation so that it can be linearized (Section: 4)

2. Specifying the linearization mapping (Section: 6)

3. Proving that the output of our snap operation is correct in the sequential execution given by the linearization mapping (i.e., that the mapping we designed actually preserves "cause and effect") (Section: 7).

These three components make up the bulk of the remainder of this thesis.

1.3 Our Results

In asynchronous systems, there can be some processes that run slowly, or experience long delays, and the adversarial Scheduler in our model has the ability to cause such long freezes. This can have an undesirable effect for any implementations where processes use locking mechanisms, since a process can be stuck for a long time waiting for another process to release the lock. For example, an operation by process p_i could be in a loop where it repeatedly tries to access a shared memory location, sees that some other process is using it (it sees a lock bit has been set), so p_i keeps iterating the loop without making any progress. A *wait-free* implementation ensures this can never happen such that the implementation is designed in a way that, even if the Scheduler decides to freeze all processes except one process p_i, then p_i will make progress (and if this continues for long enough, p_i will eventually complete the operation it is performing). Wait-freedom is the strongest progress guarantee, and it is ideal/necessary for real-time systems.

The current state of the art for dynamic atomic snapshot implementations has a time complexity of $O(n^2)$, where n is the total number of active processes in the system at any time t. In this thesis, we present an implementation with time complexity $O(n \log m)$ for Snap, Update, and Remove, and time complexity $O(m + n \log m)$ for Add, where m denotes a fixed known upper bound on the number of operations that will be performed and n is the maximum number of active processes in the system. We will also make sure that the operations are wait-free and linearizable.

The remainder of the thesis is structured as follows. In Section 2, we review related work. Section 3 provides our model assumptions along with definitions, notation,

and some basic results about the definitions. Section 4 gives our dynamic atomic snapshot implementation, and Section 5 makes some useful observations about the implementation code. Section 6 defines our linearization mapping. Section 7 proves that the output of our snap operation is correct, i.e., it matches the output of a sequential execution after applying our linearization. Section 8 analyzes the worst-case cost of each operation in our implementation. Section 9 collects together a variety of technical lemmas that are used throughout the earlier sections. Section 10 concludes the thesis and provides several directions for future research.

2 Literature Review

In this section we summarize previous work related to atomic object and atomic snapshot algorithms.

2.1 Atomic Object

Lamport [33, 34] are among the very first papers to introduce the atomic shared memory object. In these papers, they referred to the atomic objects as atomic registers. They solved the problem of asynchronous communication among processes by having the processes write data to memory registers that could be red by all processes. This concept of atomicity was later utilized by Herlihy and Wing [26] when they came up with an important concept in distributed computing, currently known as *linearizability*. This powerful idea helps visualize concurrent processes as instantaneous processes by observing the points in time when the process begins emitting data and receives the response. The idea of linearizability was further discussed in the paper by Goldman and Telick [22]. They introduced the liveness properties of linearizable shared objects. They described a unified model which established a connection between shared atomic objects and message passing systems using the I/O automaton model [38, 39]. The I/O automaton model was initially used only for message passing models. In this paper, they extended the model and showed how it can be used for shared memory models. The actions performed by atomic objects are

known as atomic actions. Lamport and Schenider [37] discussed various properties of atomic actions and how they can be related to shared variables.

The *wait-free* property of a concurrent data object denotes that each action of the object is bound to terminate after a finite amount of time regardless of the states of the execution of the other operations. An important limitation of the atomic register was first proved by Herlihy [24]: although in a system where the shared memory hardware consists of non-atomic shared registers, it is possible to implement wait-free atomic read-write registers, it is impossible to use wait-free atomic read-write registers to create wait-free implementations of complex concurrent data structures such as stacks, queues, sets, priority queues or lists using only atomic registers.

2.2 Atomic Snapshot

The atomic snapshot object in shared memory systems was first introduced by Afek, Attiya, Dolev, Gafni, Merritt, and Shavit [3]. They described two fundamental and important algorithms to compute the atomic snapshot of an atomic object consisting of read-write atomic registers. One of the implementations used unbounded memory, which means there was no fixed upper bound on the integer fields that were used to maintain the states of this algorithm. Fundamentally, there were two operations in this algorithm. One of which was the *snap* operation and other one was the *update* operation. Each of these operations were atomic operations. Their implementation is commonly known as the "double-collect algorithm" since, during each snap operation, the algorithm repeatedly reads all register values until the array is the same in two consecutive passes. However, if the Update operation is implemented in the obvious way, i.e., just write the desired value to the register, then this can lead to a non-terminating Snap operation (e.g., two consecutive passes never look the same due to Update operations happening at the same time). To mitigate this issue, a nested Snap operation was performed as the first step during each Update operation. They proved this has the effect of ensuring that two consecutive passes through the array will look the same within n attempts, which leads to a $O(n^2)$ worst-case time complexity for Snap and Update operations. The other algorithm described in

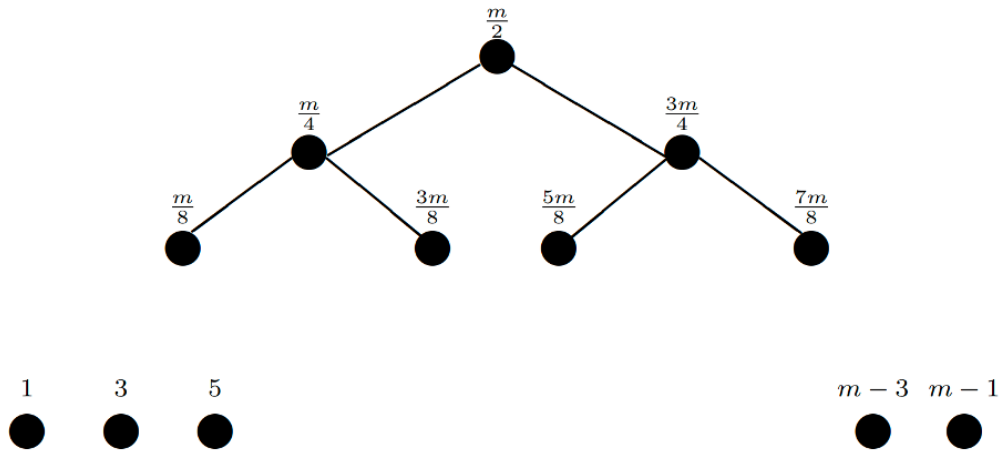

Figure 2: The Classifier binary tree. Picture taken from Attiya and Rachman [16]

this paper improved the implementation by ensuring that only bounded memory is needed. They used a finite number of "handshaking" bits to keep track of memory states, instead of incrementing integer counters. This handshaking-bit approach was inspired by Peterson [41]. They also extended the bounded-memory version of the algorithm to support multi-writer multi-reader registers (MWMR), where each of the processes could write to each of the registers, whereas the previous two algorithms only worked for SWMR systems of registers, i.e., each process was only allowed to write to a single register dedicated to it.

Anderson [8, 9] described a different atomic snapshot implementation using combinations of multiple atomic registers which he referred to as the composite register. This approach was highly inspired by the work of Chandy and Lamport [19] where they discussed the ways a process can query about the global state in a distributed environment.

Compared to the traditional double-collect approach, an asymptotically faster SWMR atomic snapshot implementation was introduced by Attiya and Rachman [16]. In this algorithm the running time of both update and snap operations were significantly improved from $O(n^2)$ to $O(n \log n)$. In this paper, initially an m-shot algorithm was proposed, which means that there is a fixed known upper bound m on the number of operations that will be performed. This algorithm introduced a

new way of thinking about the atomic snapshot implementation by using a binary tree to essentially "sort" the operations as they occur by placing them at the leaves of the binary tree, and then the linearization ordering can be obtained by reading the leaves of the tree from left to right. The height of the tree is bounded by $\log m$. More specifically, to traverse the tree, there is a Classifier procedure (Figure 2) that decides whether we should go to the left subtree or to the right subtree from a given node depending on how many other processes have previously visited the node. The Classifier operation took $O(n)$ time and it was repeated at most $\log m$ times. Thus, the overall complexity was reduced to $O(n \log n)$ (where m can be at most n) for both the snap and update operations. Our dynamic snapshot implementation in this thesis is based on this classification tree approach. There were two limitations with the m-shot algorithm. One of which was that it could only support up to m operations and another was that it needed unbounded memory. So, the authors described the implementation of a new ∞ - shot algorithm which supported an arbitrarily large number of operations and used bounded memory. This new algorithm was a clever combination of their m-shot algorithm and the handshaking-bit version of double-collect [3], and they proved that the running time of this new algorithm was still $O(n \log n)$ for both the Snap and Update operations where n is the number of process.

Since the work of Afek et al. [3] and Attiya et al. [16] there have been further implementations of atomic snapshots, but for systems with more sophisticated hardware primitives beyond just read-write registers. Jayanti [30] introduced the *f-array* which is a wait-free implementation of multi-writer snapshot registers. The author described a way to implement wait-free aggregate functions such as the minimum, the maximum, and the sum of an array of data. In the proposed algorithm, the complexity of the read operations is $O(1)$ and the complexity of the increment operations is $O(\min(k, \log n))$ where k is the number process accessing the algorithm at any given time. The core part of this algorithm uses a special variant of the priority queue data structure named *process priority queue* which is shared by n processes and supports *insert*, *findmin* and *delete* operations. This is why the increment operations have a complexity similar to the complexity of a heap data-structure. Riany, Shavit and Touitou [46] introduced another atomic snapshot algorithm called the

coordinated-collect algorithm which makes use of various hardware primitives such as CAS (compare-and-swap), fetch-and-increment and fetch-and-decrement. The update complexity of their algorithm was $O(1)$ and the snap complexity was $O(n)$. This algorithm was later improved by Jayanti [31] where it was proved that it was possible to run the algorithm using hardware that only supported the CAS (compare-and-swap) operations instead of all three operations. The time complexity of this new algorithm was similar to the previous one. Aspnes, Attiya, Censor-Hillel and Ellen [10] discussed a similar implementation of atomic snapshot that used $O(\log^3 n)$ time in each operation, which was a significantly large improvement. Interestingly, this algorithm was cost-effective since it used simple read/write registers whereas the previous algorithms required certain operations supported in hardware. But the drawback of this algorithm was that it was only possible to achieve the optimal time complexity when the number of completed update operations was polynomial in the number of registers n.

Another multi-writer atomic snapshot algorithm was proposed by Imbs and Raynal [28]. The key advantage of this algorithm was in its simplicity of the implementation. The algorithm suggested that during each update operation we should be writing the value to a helping register. The purpose of the helping register is to make the update operation more asynchronous and during the snap operation these helping registers are used to fetch the most updated value. The idea of this algorithm was highly inspired by the multicore architecture [25]. The time complexity of this algorithm in worst case was $O(n^2)$. Afek, Shavit and Tzafrir [4] implemented a variation of the double-collect atomic snapshot algorithm to solve the wait-free and lock-free problem in the Java Development Kit's size() method of concurrent data structures.

A partial snapshot [14, 29, 32] is another variant of general atomic snapshot where instead of getting a view of the whole object, in the snap operation it can be specified for which indices the snapshot will be generated. The F-snapshot problem [7] is another variant of the snapshot problem where the authors solved the problem for a range snap operation in an infinitely large domain using the existing solutions of the signaling problem [5, 6, 1].

2.3 Dynamic Atomic Snapshot

The notion of dynamic atomic snapshot was first introduced by Spiegelman and Keidar [49]. The algorithm described in this paper was inspired by the double-collect algorithm from Afek et al. [3]. Instead of using a normal shared memory, the authors in this paper considered the dynamic shared memory of the single-writer, multi-reader registers. This meant that instead of storing the data at a shared array they used a dynamic data structure. More specifically, it maintained a mapping from a dynamic set of *nodes* to their values. The nodes that they referred to in this paper, is a shared memory location which is similar to the "bag" data structure in our work. Each node had a unique identifier. The algorithm also kept track of all the nodes that had been added so far. Along with the snap and update operations of the traditional atomic snapshot algorithm, since this was a dynamic variant of the algorithm and nodes could be added and removed during the run-time, there were add and remove operations as well in order to add and remove the registers from the dynamic atomic object. This algorithm solves the problem in two different types of memory model. The first memory model is the *persistent memory* model where once a register is added, it is never fully de-allocated from the data structure after the remove operation. On the other hand, *ephemeral memory* allows the registers to be fully de-allocated from the data structure, so that it can be reclaimed by the system and reused elsewhere. This algorithm had a $O(n^2)$ worst-case running time for add, remove, update and snap operations, since it was based on the double-collect algorithm for static atomic snapshot.

2.4 Anonymous Atomic Snapshot Algorithms

Anonymous systems have a huge appeal in the area of the data security and privacy. For example, web servers [45] and peer to peer file sharing systems [17] are built up on the primary fundamentals of anonymous systems. It is imperative that multiple systems or applications run without revealing sensitive or identifying information to each other, to external eavesdroppers and make sure to conceal users' identity. There are two types of anonymous shared memory systems. One of the anonymous sys-

tems focuses on process anonymity and the other one focuses on memory anonymity. With the introduction of anonymous shared memory algorithms many practical applications are now possible to build using distributed shared memory. Distributed anonymous applications can be benefited from dynamic capabilities integrated into it.

In this thesis, our model assumes that processes have unique identifiers. However, atomic snapshot has also been studied in systems where processes are anonymous, starting with the work of Guerraoui and Ruppert [23]. The authors initially defined a timestamp algorithm in the anonymous shared memory model system using fetch-and-increment object. They called it the *weak counter object*. They implemented the wait-free version of the weak counter object using an intuitive binary search approach. This algorithm was later used as an integral part of the implementation of the anonymous atomic snapshot algorithm. It was important to use the multi-reader multi-writer register for the implementation of this atomic snapshot algorithm otherwise the anonymous property of the processes would not hold.

Recently there have been multiple papers [43, 23, 44] published on the fully anonymous versions of popular distributed shared memory algorithms such as the mutual exclusion problem, consensus problem and set agreement problem. An interesting approach was presented in Raynal and Taubenfeld [44] which uses co-prime numbers to solve the anonymity problem in a wait-free environment.

3 Preliminaries

3.1 System Restrictions

In this section, we recall some of the important assumptions made in our system model, and we also introduce additional assumptions that clarify some system behaviours that were not defined precisely in the model.

1. The shared memory consists entirely of single-writer multi-reader (SWMR) registers.

2. Each process that interacts with the system is identified by a unique integer id. For example, p_i is a process id. For ease of notation, since we are using SWMR registers, the process id is also used as memory address for the respective process. For example, process p_i can only write in memory address p_i.

3. When the system is initialized, i.e., at time 0, some non-empty set of processes is automatically added by the system. This set of processes cannot be removed and are called $procList_\alpha$.

4. When a process p_i is added to the system, a register is automatically allocated in shared memory with address p_i. This block of memory is automatically initialized as follows: `active` with the list of register addresses of the processes in $procList_\alpha$, `seq` with 0, `removeActive` with empty list and `val` with 0.

5. The total number of Update, Snap, Add, and Remove operations that can be performed is at most m, which is a fixed known value.

6. When a process p_i is added to the system, its identity p_i is automatically added to the activeList of the block at shared memory location p_i. In other words, when a process joins the system, its corresponding SWMR register already stores the existence of p_i.

7. The processes in $procList_\alpha$ are automatically added to the activeList of every block of every shared memory location that is ever initialized. In other words, whenever a process joins the system, its corresponding SWMR register already reflects the existence of all processes that were initially in the system at time 0, but not necessarily the existence of processes that have since joined the system.

8. When a process is executing an operation, it cannot start another operation unless the first operation has finished its execution. In other words, a process is considered as a single thread that runs its own code in a sequential manner.

9. If an Add operation is called to add a process p_i to the system, then the new process p_i cannot start running its own code until the Add operation finishes its execution.

10. A Remove operation to remove a process p_i from the system can only be called by the process that performed the original Add operation that added p_i to the system. Besides simplifying the implementation details, this assumption is of practical relevance, e.g., as a security feature.

11. When a process p_i is removed from the system, its associated shared register p_i is not erased or de-allocated, which means that the contents of register p_i can still be read by the remaining processes in the system. However, by the definition of the Snap operation, register p_i should not be contained in the output of the Snap operation after p_i is removed from the system.

3.2 Definitions

3.2.1 Memory and Data Structures

Suppose that each process has information that it wishes to be publicly available to all other processes in the system, and this information may change over time. There is a collection S of SWMR atomic read-write registers where each process will store its own information, and within the register the information will be stored in a field called *val*. To interact with a register in S, a process p_j calls S.WRITE(p_j,d) to write value d to its own register address p_j, and a process calls S.RETRIEVE(p_k) for any register address p_k that it knows about. (Recall that we are in a dynamic environment, so the process p_k might be added to the system without p_j knowing about it yet). Additionally, inside each register, processes will write additional fields that are used only for the purposes of our snapshot implementation. So, when reading and writing to shared registers, the processes will actually be reading and writing a *block* consisting of 4 fields, whose names are val, seq, active, and removeActive. The purpose of these additional 3 fields will become more apparent when we describe the snapshot implementation. When a new process joins the system and a new register is created for that process in S, a default block is written with `val` set to 0, `seq` set to 0, `active` set to be the list of register addresses of the processes in $procList_\alpha$, and `removeActive` is empty. Notationally, a block will usually be referred to using BL,

so, for example, accessing the val field within a block BL would be done using BL.val.

Some additional data types and variables will be needed to implement the snapshot object. At a very high level, whenever a process wants to perform an operation on the snapshot object, it must perform a "scan": it will read the contents of all shared registers in S from addresses it knows about. When it does this, it retrieves the block from each register, and it will store all these blocks in a local (i.e., unshared) structure called a *bag* (you can think of a bag as a dictionary of blocks, where the key associated with each block is the shared memory address). To interact with a bag, the process can call add(k,BL_k) to add a new address k to the bag and have it point to block BL_k, it can call write(k,BL_k) to overwrite the block at address k with the block BL_k, it can call retrieve(k) to get the block BL_k stored at address k, and it can call erase(k) to remove address k from the bag (and its associated block). Notationally, a bag will usually be referred to using BA, so, for example, retrieving the block at address k would be done by calling BA.retrieve(k). The list of addresses of the blocks in the bag BA is represented by $addr(BA)$. This list is only used while proving the correctness of the algorithms and is not available in actual algorithm itself.

Unfortunately, performing a "scan" of all the registers in S does not guarantee an accurate picture of what is stored in S, due to the adversarial Scheduler (for example, while a process is waiting to read one register from S, the Scheduler can ensure that all previously-read registers have been changed multiple times by other processes). To overcome this issue, each process will compare its bag (i.e., the result of the scan) to the bags of all other processes (i.e., the results of their scans). Doing this in a careful and efficient way is essentially what the rest of this thesis is about. At a high level, we adapt the Classifier Tree approach from Attiya and Rachman [16], which combines the "most recent" scans by all processes (this is called a *merge*, which will be defined in Section 3.2.3) to produce an "up-to-date" view of what is stored in all registers in S. To achieve this, the main ingredient is the "sequence number" seq that is stored in each block, which gets incremented each time that the block is modified, and this will help define what "most recent" means (we will use the notion of *domination*, which will be defined in Section 3.2.2). For now, we focus on defining the elements of the Classifier Tree structure.

The Classifier Tree is a binary tree structure that is implemented in shared memory using atomic registers. Using the fixed upper bound m on the number of operations that will be performed on the atomic snapshot object, the structure of a fixed perfect binary tree with m leaves is assumed to be known by all processes (i.e., where in memory the root node is stored, and where in memory each left/right child is stored). The Classifier Tree is a perfect binary tree with the height of $\lceil \log(m) \rceil$. If m is not a power of 2, we add dummy nodes in the Classifier Tree to make it a perfect binary tree. Each node is labeled with an integer value as shown in the Figure 2 (although the m leaf nodes are omitted from the diagram). The important part of the Classifier Tree is that each non-leaf node consists of a Container object C. A Container is a collection of shared registers, one dedicated to each process, where each process will write a bag (i.e., its current knowledge about the contents of registers in S). $addr(C)$ is the set of addresses in the Container C where a bag has been written. To interact with a Container object, a process can call C.WRITE(p_j,d) to overwrite the contents of the register at address p_j (where $p_j \in addr(C)$) with data d, it can call C.RETRIEVE(p_j) to retrieve the data stored at address p_j within the Container, and, in order to accommodate the fact that new processes may join the system, a process can call C.INITIATE(p_j) to allocate a new empty register with address p_j within the Container (so that p_j can later write its bag to the Container).

Notationally, a Container will usually be referred to using a variable C_j (where the subscript is used to distinguish between Containers at different nodes of the Classifier Tree). Since a Container stores many bags, the bag at address i within Container C_j will be referred to as $BA_{i,j}$. Since a bag stores many blocks, the block at address k within the bag at address i within Container C_j will be referred to as BL_{k,BA_i,C_j}. Other variants of this notation will be used depending on context. For example, a process p_i will traverse from root-to-leaf of the Classifier Tree each time it performs on operation op_i on the atomic snapshot object, and multiple processes could be doing this at the same time in parallel, i.e. one process performing op_i and another process performing op_j, so in that context we may use subscripts op_i and op_j to distinguish between values being encountered by each operation.

At a high level, the Classifier Tree is used as follows. After a process performs a

"scan" of the shared memory S to create its initial bag, it will start at the root node of the Classifier Tree, writes its bag into its dedicated spot within the Container, then do a special "merge" operation (to be discussed later) that "combines" all the bags that are in the Container to get a "merged view". Depending on the sequence numbers of blocks within this merged view, it will decide whether it needs to go to the node's left child (which essentially means "this merged view is relatively old") or to the node's right child (which essentially means "this merged view is relatively recent"), and the whole process is repeated at the Container of the child node that it visits, until it reaches a leaf of the tree, at which point it adopts the "merged view" as its snapshot of S. The benefit of this tree structure appears when linearizing and analyzing the algorithm: when considering the leaves of the Classifier Tree from left-to-right, we can get a causal ordering of the operations performed by the processes. The lowest common ancestor (LCA) node of two operations plays a vital part in our proofs in the later sections. LCA(op_u, op_v) is the node at the deepest level in the Classifier Tree which is visited during both op_u and op_v. In other words, LCA(op_u, op_v) node is the lowest common ancestor node during the operation op_u and op_v.

In the above, notice that some method calls are written in upper case while others are written in lower case. As a convention, we use upper case to denote operations on shared memory, and we use lower case to denote local operations that are performed on the process' local variables. This distinction is important because a process remains frozen at any line of code containing a shared memory operation until the Scheduler decides to let the operation execute, and no waiting occurs at all other lines of code. Also, note that, when we usually refer to an operation as op_i, it is implied that the operation is being performed by p_i. Additionally, as discussed later, the running-time complexity of the implementation is measured by the number of shared memory operations, and so distinguishing these operations in the code is helpful when carrying out the analysis.

3.2.2 Domination

Given two bags, each consisting of blocks, we will want to figure out which bag has the most recent versions of each block. So, we define a domination property: when the seq value in each block in a bag is at least as big as the seq value of the corresponding block in another bag, then we say that the former bag dominates the latter bag.

More formally, let two bags be BA_1 and BA_2. Let the set of addresses of the blocks in BA_1 and BA_2 be $addr_1$ and $addr_2$ respectively. BA_2 *dominates* BA_1 if the following conditions are met:

1. $addr_1 \subseteq addr_2$
2. For each address j that is both in $addr_1$ and $addr_2$, $BL_{j,1}$.seq $\leq BL_{j,2}$.seq where $BL_{j,1}$ is the block with address j in bag BA_1, and $BL_{j,2}$ is the block with address j in bag BA_2

3.2.3 Merge

The *merge* is the process of combining the information of multiple bags into one single bag. We only care about keeping the most recent information about each block, so, when merging, each block in the resultant bag is taken to be the most recent version of the block from all the bags we are merging together.

More formally, let *bagSet* be a set of any number of bags where the size of the set is at least 1. After performing the *merge* operation on *bagSet*, let the final bag be $mergedBA_{final}$. For each bag $BA_i \in bagSet$, let the set of addresses of the blocks be $addr_i$. When the *merge* operation is performed on the set of bags *bagSet* the following conditions are met:

1. The set of addresses of the blocks in $mergedBA_{final}$ is the union of all the $addr_i$.
2. The block at address j in $mergedBA_{final}$ is the block with maximum seq value among all blocks $BL_{j,i}$ (where $BL_{j,i}$ denotes the block at address j in bag BA_i) taken from all bags $BA_i \in bagSet$.

Note: If a bag BA_i has no block with address j, then the merge operation does not consider BA_i when determining the block at address j in $mergedBA_{final}$.

3.2.4 SeqSum

After a process does a scan of blocks in shared memory, this "view" is stored in a bag. To compare two views, we will use the domination property described above, but also, it will be helpful to assign an integer to each view so that the comparison is simpler and quicker. So, we associate with each bag its *seqSum*, which is the sum of the sequence numbers of all blocks in the bag. More formally: Let BA_1 be a bag. Let $addr_1$ be the set of addresses of the blocks in BA_1. The *SeqSum* of bag BA_1 is defined to be $\sum_{j \in addr_1} BL_{j,1}.seq$ where $BL_{j,1}$ is the block at address j in the bag BA_1.

3.2.5 Partial Ordering

For any two operations, there are two possibilities regarding their execution: either they overlap in time, or, one operation ends before the other begins. This means that the set of operations is a partially ordered set. The *partial order* is defined as follows: if an operation op_b starts its execution after the end of the execution of another operation op_a, then we define that op_a *occurs before* op_b, and this is denoted using $op_a \rightarrow op_b$.

3.2.6 Data Types

The following tables summarize the definitions and notation introduced so far, and they are provided as a useful reference when reading the rest of the thesis.

Variable Name	Memory Type	Operations	Description
	Register		SWMR atomic read-write registers. Each register is associated with a physical address in the shared memory. For notational convenience, the address of a SWMR register is assumed to be the same as the process ID p_j of the process that can write to the register.
S	Collection of Registers		This variable represents the primary shared memory in the system. It consists of SWMR atomic read-write registers. This variable is used inside the add, remove, update and snap procedures only.
		S.WRITE(p_j, d)	Writes data d in register at address p_j in the shared memory S and overwrites any existing values.
		S.RETRIEVE (p_j)	Retrieves the data located at register p_j in the shared memory S.
C	Collection of Registers		This variable represents an additional shared memory that we use in our solution. More specifically, these registers are used in the Classifier Tree data structure. In each node of the Classifier Tree there is a shared variable C. Each such C will be referred to as a *Container*.
		C.INITIATE(p_j)	System-provided call to allocate a new empty register with address p_j inside the Container C.
		C.WRITE(p_j, d)	Writes data d to the register at address p_j in the Container and overwrites any existing values.
		C.RETRIEVE (p_j)	Retrieves the data from the register located at the address p_j. If there is no such address then it returns null.

Table 3.1: Shared Memory Types

Type Name	Operations	Description
Block		Each block has four properties: val, seq, active, removeActive which hold the current value, current sequence number, current active set of register addresses and, the set of register addresses removed by the current process respectively. The initial values of a block are as follows- val: 0, seq: 0, active: list of processes initially present in the system ($procList_\alpha$), removeActive: empty list
Bag		A bag is a collection of blocks. Each block in a bag can only be accessed using its address/key k. Each bag supports the following operations:
	add(k, BL_k)	Writes block BL_k at address k in the bag for the first time
	write(k, BL_k)	Writes block BL_k at address k in the bag and overwrites any existing values
	retrieve(k)	Retrieves the block located at address k. If there is no such address then it returns null.
	erase(k)	Removes the block (including all the fields' values) from the memory of the bag located at address k. If there is no such address then it does not remove anything.
List	list()	Initializes the list. The list can only hold the integer addresses of the registers
	add(p)	Adds an address p to the list
	exists(p)	Returns true if the address p exists in the set, otherwise returns false

Table 3.2: Local Variable Data Types

Variable Name	Description
merge(X)	Input X is a set of bags. This notation denotes the merge result of the set of bags X.
merge(C)	Input C is a Container. This notation denotes the merge result of all the bags in the Container C.
$BA_{i,j}$	means the bag at address i of the Container j.
BL_{k,BA_i,C_j}	denotes the block k in bag i in Container j.
k $\in addr(BA_i)$	denotes any k from the list of processes that wrote a block in bag BA_i.
$i \in addr(C_j)$	denotes any i from the list of processes that wrote a bag in the Container C_j.
p_i	denotes the unique id the process p_i.
op_i	denotes an operation by a process.
BL_{p_i}	denotes the block at address p_i
$BL_{i,BA}$	denotes the block at address i of the bag BA
BL_{p_i,ss_ℓ}	denotes the block at address p_i in the bag ss_ℓ. This is a special case of $BL_{i,BA}$ where BA is referring to a bag being used at level ℓ of the Classifier algorithm.
BL_{p_i,ss_ℓ,op_k}	denotes a block at address p_i of the bag ss_ℓ being used at level ℓ of the Classifier algorithm while executing the operation op_k.
$procList_\alpha$	denotes the list of process that exist in the system at time 0. When a new process is added in the system, the processes in this list are added to the active list of that new process by the system.

Table 3.3: Notations

3.3 Preliminary results and observations

3.3.1 Relating Domination and seqSum

In this proposition, we show that if one bag dominates another bag, then the seqSum of the bag that dominates the other will be at least as big as the other bag's seqSum.

Proposition 3.1. *Let two bags be BA_1 and BA_2. Let the set of addresses of the blocks in BA_1 and BA_2 be $addr_1$ and $addr_2$ respectively. We prove that if BA_2* dominates

BA_1 *then:*

$$\left(\sum_{j \in addr_1} BL_{j,1}.seq\right) \leq \left(\sum_{k \in addr_2} BL_{k,2}.seq\right)$$

Proof. 1. From condition 1 of the definition of domination (Section: 3.2.2), each address of BA_1 exists in the address set of BA_2. And, from condition 2 of the definition of domination (Section: 3.2.2), each of the blocks in these common addresses has the property $BL_{j,1}$.seq $\leq BL_{j,2}$.seq. Using these two conditions, we can say the following:

$$\left(\sum_{j \in (addr_1 \cap addr_2)} BL_{j,1}.seq\right) \leq \left(\sum_{j \in (addr_1 \cap addr_2)} BL_{j,2}.seq\right)$$

2. There might be other addresses in BA_2 that are not included in the inequality in step 1 of this proof. Formally, these addresses are $addr_2 \setminus addr_1$, where $\setminus$ denotes set of elements that are in $addr_2$ but not in $addr_1$.

3. Since the initial value of seq field of all blocks is 0 and the seq values are never decreased in any of our algorithms, the following is true:

$$\left(\sum_{l \in (addr_2 \setminus addr_1)} BL_{l,2}.seq\right) \geq 0$$

4. Combining the inequalities of steps 1 and 3:

$$\left(\sum_{j \in (addr_1 \cap addr_2)} BL_{j,1}.seq\right) \leq \left(\sum_{j \in (addr_1 \cap addr_2)} BL_{j,2}.seq\right) + \left(\sum_{l \in (addr_2 \setminus addr_1)} BL_{l,2}.seq\right)$$

5. From the definition of set subtraction, $addr_2 = (addr_2 \setminus addr_1) \cup (addr_1 \cap addr_2)$

6. Combining the previous two steps we see that,

$$\left(\sum_{j \in addr_1} BL_{j,1}.seq\right) \leq \left(\sum_{k \in addr_2} BL_{k,2}.seq\right)$$

□

3.3.2 Merging preserves domination

In the following result, we show that if there are two Containers, and each bag in the first Container is dominated by its corresponding bag in the second Container, then the outcome of a merge on the first Container is dominated by the outcome of a merge on the second Container.

Proposition 3.2. *Let C_1 and C_2 be two Containers. Let $mergedBA_1 = merge(C_1)$ and $mergedBA_2 = merge(C_2)$. If the following conditions are satisfied:*

i. Any process $p_i \in addr(C_1)$ that writes a bag in C_1, also writes a bag in C_2.

ii. For each process $p_i \in addr(C_1)$ that writes a bag in C_1 and C_2, $BA_{i,1}$ is dominated by $BA_{i,2}$

then, we prove that $mergedBA_1$ is dominated by $mergedBA_2$.

Proof. To prove that $mergedBA_2$ dominates $mergedBA_1$, we prove the two conditions from the definition of domination:

1. $addr(mergedBA_1) \subseteq addr(mergedBA_2)$

2. For any address j $\in (addr(mergedBA_1) \cap addr(mergedBA_2))$, $BL_{j,mergedBA_1}$.seq $\leq BL_{j,mergedBA_2}$.seq

We prove these in the following steps:

1. From the definition of merge we know that $addr(mergedBA_1)$ is the union of all the addresses of all the blocks in all the bags in C_1. Similarly, it is true for $addr(mergedBA_2)$ with respect to C_2.

2. By the assumed conditions, for each $p_i \in addr(C_1)$, we know that $BA_{i,1}$ is dominated by $BA_{i,2}$, so by the definition of domination, we know that $addr(BA_{i,1}) \subseteq addr(BA_{i,2})$.

3. From step 1 of this proof, we know $addr(mergedBA_1) = \cup_{i \in addr(C_1)}(addr(BA_{i,1}))$ and $addr(mergedBA_2) = \cup_{j \in addr(C_2)}(addr(BA_{j,2}))$. From step 2 of this proof, we know each $addr(BA_{i,1}) \subseteq addr(BA_{i,2})$ for each $p_i \in addr(C_1)$. Since each $addr(BA_{i,1}) \subseteq addr(BA_{i,2})$, $\cup(addr(BA_{i,1})) \subseteq \cup(addr(BA_{i,2}))$ where the unions are taken over all $p_i \in addr(C_1)$. Since we assume that $addr(C_1) \subseteq addr(C_2)$, it follows that, $addr(mergedBA_1) \subseteq addr(mergedBA_2)$ (this proves condition 1).

4. Let k be any address where k $\in addr(mergedBA_1)$ and k $\in addr(mergedBA_2)$. From step 1, we know $addr(mergedBA_1) = \cup_{i \in addr(C_1)}(addr(BA_{i,1}))$ and $addr(mergedBA_2) = \cup_{j \in addr(C_2)}(addr(BA_{j,2}))$. This implies that there is at least one bag in C_1 that contains a block with address k, and, at least one bag in C_2 that contains a block with address k.

5. Since $mergedBA_1$ is the merge of all the bags in C_1, from the definition of merge, block $BL_{k,mergedBA_1}$ is the block with the maximal seq value at address k taken over all the bags in C_1.

6. By the definition of merge, there is at least one bag $BA_{i,1}$ in Container C_1 such that block $BL_{k,BA_i,C_1} = BL_{k,mergedBA_1}$. For each such bag, we know by assumptions i and ii that $BA_{i,1}$ is dominated by bag $BA_{i,2}$ from Container C_2. In other words, from the definition of domination, each BL_{k,BA_i,C_1}.seq $\leq$ BL_{k,BA_i,C_2}.seq. This means, there exists a block BL_{k,BA_i,C_2} with seq value at least as big as the seq value of the block BL_{k,BA_i,C_1}.

7. Step 6 proves the existence of at least one bag $BA_{i,2}$ in Container C_2 that contains a block BL_{k,BA_i,C_2} with seq value at least as large as $BL_{k,mergedBA_1}$. From the definition of merge, we conclude that $BL_{k,mergedBA_1}$.seq $\leq BL_{k,mergedBA_2}$.seq. (This proves condition 2)

□

3.3.3 Partial ordering of Add operations

Let p_x be any process that is not initially in the system, i.e., suppose that $p_x \notin procList_\alpha$. Informally, we observe that p_x must be added to the system as part of some sequence of Add operations, and that no two of these operations can overlap (in other words, each Add operation in the sequence is partially ordered before the next).

More formally, let AC be any sequence of add operations that lead to the addition of p_x in the system. Let the length of AC be len_{AC}. Let an add operation be $add_k(p_u, p_v)$ where it is the kth add operation in AC and this add operation is executed by p_u where p_v is added. More formally, $AC = add_1(p_0, p_1)$, $add_2(p_1, p_2)$, ... , $add_{len_{AC}}(p_{len_{AC}-1}, p_{len_{AC}})$ where $p_1 \in procList_\alpha$ and p_0 is the system. Let $AC(p_k)$ be the chain of add operations that lead to the addition of p_k in the system. Let $ancestor(p_k)$ be the list of processes involved in $AC(p_k)$ including p_k. Consider any two adjacent add operations in AC: $add_k(p_{k-1}, p_k)$, $add_{k+1}(p_k, p_{k+1})$. We know the system only allows a process to start executing any operation only after the add operation that added the process has ended its execution. Therefore, $add_{k+1}(p_k, p_{k+1})$ can only start after $add_k(p_{k-1}, p_k)$ has ended. So, $add_k(p_{k-1}, p_k) \rightarrow add_{k+1}(p_k, p_{k+1})$. Extending it to all operations in the chain of add operations AC, $add_1(p_0, p_1) \rightarrow add_2(p_1, p_2) \rightarrow .... \rightarrow add_{len_{AC}}(p_{len_{AC}-1}, p_{len_{AC}})$.

3.4 Performance Metrics

We measure the performance of the algorithm by doing a formal analysis of running time. The running time of each operation is defined as the worst-case number of reads and writes to shared memory, and will be expressed in terms n (the number of active processes in the system) and m (the number of operations performed). We are considering the number of shared memory operations, i.e., reads and writes to atomic registers, since in real systems these costs are orders of magnitude larger than local operations performed at each process. The local reads and writes are considered O(1).

4 Algorithm code

In this section, we provide a description and pseudocode of all the methods needed to implement our dynamic atomic snapshot object. First, in Sections 4.1-4.4, we describe our implementations of the required atomic snapshot operations that the processes in the system may execute: Update, Add, Remove, and Snap. Then, in Sections 4.5-4.9, we describe the private helper methods that are used in the descriptions of these operations.

4.1 Algorithm: Update

The purpose of Update is to update the *val* being stored at a process p_i's shared atomic register, using the value passed in as the input parameter. This is one of the 4 operations required in any dynamic atomic snapshot implementation. Algorithm 1 provides the pseudocode for Update, and Table 4.1 describes the variables that are used in the code. From line 2 to 5, process p_i (i.e., the process that is performing the Update operation) reads, updates, and writes its own block back to shared memory S. To update its block, it overwrites *val* with the new intended value, and increments the seq value of the block.

The rest of the code after line 5 relates to invoking the Classifier. From line 6 to line 11, process p_i creates its initial "view" of the system by scanning through the shared atomic registers of all other processes that it knows about, i.e., using the addresses that appear in its active list, and copying them into a bag called ss_0. From line 12 to 15, the Classifier Tree is traversed and updates its "view" of the system along the way by taking the bag returned to it by the Classifier algorithm. The first call to Classifier occurs at the root of the Classifier Tree, and in this first call, process p_i passes in its initial view ss_0. The details of the Classifier algorithm are provided in Section 4.5. Here, the loop is executed $\lceil \log(m) \rceil$ times because the height of the Classifier Tree is $\lceil \log(m) \rceil$ and Classifier is not executed at the leaves of the Classifier Tree (recalling that m is an upper bound on the number of operations that can be called on this dynamic atomic snapshot object). At line 14 inside the loop,

the Classifier algorithm returns a tuple where the first value is a Bag that represents p_i's updated view of the system, the second value is a list of integers representing an updated list of active processes in the system and, the third value is the address of the Classifier Tree node that process p_i will traverse to next (i.e., it is the left or right child of the node that it visited last).

Algorithm 1 The update operation

```
 1: procedure UPDATE(value)
 2:     b_i ← S.RETRIEVE(p_i)                                          ▷ SREAD1
 3:     b_i.seq ← b_i.seq + 1
 4:     b_i.val ← value
 5:     S.WRITE(p_i, b_i)                                              ▷ SWRITE
 6:     ss_0 ← {}
 7:     addr_0 ← b_i.active                                            ▷ ADDR0INIT
 8:     for all k in b_i.active do
 9:         b_k ← S.RETRIEVE(k)                                        ▷ SREAD2
10:         ss_0.add(k, b_k)
11:     end for
12:     v ← ClassifierTreeRoot                                         ▷ PRECFirst
13:     for all ℓ ← 1, ..., ⌈log(m)⌉ do                                ▷ PREC
14:         {ss_ℓ, addr_ℓ, v} ← Classifier(Label(v), ss_{ℓ−1}, addr_{ℓ−1})   ▷ POSTC
15:     end for
16: end procedure
```

Variable Name	Shared or Local	Data Type	Description
value	Local	Int	Passed in as the parameter of the update operation. During the update operation this value will be written to the *val* field of p_i's dedicated shared atomic register.
p_i	Local	Int	The process that invoked the update operation.
S	Shared	Registers	The primary shared memory.
b_i	Local	Block	Locally stores the block that is copied from address p_i in the shared memory S.
ss_ℓ	Local	Bag	Holds the system view returned by the Classifier algorithm at the ℓth level of the Classifier Tree. A special case is $\ell = 0$: the bag ss_0 holds the result of p_i scanning the blocks directly from shared memory S. The bag ss_0 is the initial view that is passed into the first call to Classifier, i.e., the call that occurs at the root of the Classifier Tree.
$addr_\ell$	Local	List	Holds a list of addresses of active processes, which is returned by the Classifier algorithm at the ℓth level of the Classifier Tree. A special case is $\ell = 0$: the list $addr_0$ holds the active list that was retrieved directly from p_i's block in shared memory S. The address list $addr_0$ is the initial list of active processes that is passed into the first call to Classifier, i.e., the call that occurs at the root of the Classifier Tree.
k	Local	Int	Loop variable; used to iterate through the active list of block b_i.
b_k	Local	Block	Holds the block copied from the address k of the shared memory S.
Classifier-TreeRoot	Shared	Node	The root node address of the Classifier Tree.
v	Local	Node	Holds the address of a Classifier Tree node.
m	Static	Int	Upper bound on the number of operation calls that this dynamic atomic snapshot object supports.
ℓ	Local	Int	Loop variable. Used to keep track of the current level of the Classifier Tree traversal.
Label(v)	Local	Int	A function that translates a Classifier Tree node address to the label of that node. In particular, the Classifier Tree is a complete binary tree whose nodes are labeled by consecutive integers in the order they would be visited by an inorder traversal.

Table 4.1: Variables used in Update Operation Algorithm 1

4.2 Algorithm: Add

The purpose of Add is to add a new process p_j to the system, which is one of the 4 operations required in any dynamic atomic snapshot implementation. Algorithm 2 provides the pseudocode for Add, and Table 4.2 describes the variables that are used in the code. The process p_j can only be added to the system by a process p_i that is currently active in the system, i.e., p_i was one of the processes that was initially active in the system, or, p_i was previously added via an add(p_i) operation. So in the remainder of this section, we assume that ADD is being performed by a process p_i. Before the first line of code of add(p_j) is executed, we assume that the underlying system automatically allocates a new atomic register in shared memory, and that this memory can be accessed using process p_j's unique ID. Moreover, we assume that this new atomic register stores a block whose variables are initialized as described in Section 3.1. At line 3 of the algorithm, the address of the new process p_j is used to initialize a new memory location in each Container of each Classifier Tree node (the details of how this is carried out are provided in Section 4.6). From line 4 to 7, process p_i (i.e., the process that is adding p_j to the system) reads, updates, and writes its own block back to shared memory S. To update its block, it adds p_j to the list of active processes and increments the seq value of the block.

The rest of the code after line 7 relates to invoking the Classifier. From line 8 to line 13, process p_i creates its initial "view" of the system by scanning through the shared atomic registers of all other processes that it knows about, i.e., using the addresses that appear in its active list, and copying them into a bag called ss_0. From line 14 to 17, the Classifier Tree is traversed and updates its "view" of the system along the way by taking the bag returned to it by the Classifier algorithm. The first call to Classifier occurs at the root of the Classifier Tree, and in this first call, process p_i passes in its initial view ss_0. The details of the Classifier algorithm are provided in Section 4.5. Here, the loop is executed $\lceil \log(m) \rceil$ times because the height of the Classifier Tree is $\lceil \log(m) \rceil$ and we don't execute Classifier at the leaves of the Classifier Tree (recalling that m is an upper bound on the number of operations that can be called on this dynamic atomic snapshot object). At line 16 inside the loop,

the Classifier algorithm returns a tuple where the first value is a Bag that represents p_i's updated view of the system, the second value is a list of integers representing an updated list of active processes in the system and, the third value is the address of the Classifier Tree node that process p_i will traverse to next (i.e., it is the left or right child of the node that it visited last).

Algorithm 2 The add operation

```
1: procedure ADD(p_j)
2:     %% When this method is invoked, but before line 3 is executed, the system
       automatically creates a new atomic register in shared memory S at address p_j
       and writes a new block at the new atomic register with initialized values
3:     AddToClassifierTree(ClassifierTreeRoot, p_j)
4:     b_i ← S.RETRIEVE(p_i)                                        ▷ SREAD1
5:     b_i.seq ← b_i.seq + 1
6:     b_i.active.add(p_j)
7:     S.WRITE(p_i, b_i)                                            ▷ SWRITE
8:     ss_0 ← {}
9:     addr_0 ← b_i.active                                          ▷ ADDR0INIT
10:    for all k in b_i.active do
11:        b_k ← S.RETRIEVE(k)                                      ▷ SREAD2
12:        ss_0.add(k, b_k)
13:    end for
14:    v ← ClassifierTreeRoot                                       ▷ PRECFirst
15:    for all ℓ ← 1, ..., ⌈log(m)⌉ do                               ▷ PREC
16:        {ss_ℓ, addr_ℓ, v} ← Classifier(Label(v), ss_{ℓ−1}, addr_{ℓ−1})   ▷ POSTC
17:    end for
18: end procedure
```

Variable Name	Shared or Local	Data Type	Description
p_j	Local	Int	The process being added to the system.
p_i	Local	Int	The process that initiated this Add operation.
S	Shared	Registers	The primary shared memory.
b_i	Local	Block	Locally stores the block that is copied from address p_i in the shared memory S.
ss_ℓ	Local	Bag	Holds the system view returned by the Classifier algorithm at the ℓth level of the Classifier Tree. A special case is $\ell = 0$: the bag ss_0 holds the result of p_i scanning the blocks directly from shared memory S. The bag ss_0 is the initial view that is passed into the first call to Classifier, i.e., the call that occurs at the root of the Classifier Tree.
$addr_\ell$	Local	List	Holds a list of addresses of active processes, which is returned by the Classifier algorithm at the ℓth level of the Classifier Tree. A special case is $\ell = 0$: the list $addr_0$ holds the active list that was retrieved directly from p_i's block in shared memory S. The address list $addr_0$ is the initial list of active processes that is passed into the first call to Classifier, i.e., the call that occurs at the root of the Classifier Tree.
k	Local	Int	Loop variable; used to iterate through the active list of block b_i.
b_k	Local	Block	Holds the block copied from the address k of the shared memory S.
Classifier-TreeRoot	Shared	Node	The root node address of the Classifier Tree.
v	Local	Node	Holds the address of a Classifier Tree node.
m	Static	Int	Upper bound on the number of operation calls that this dynamic atomic snapshot object supports.
ℓ	Local	Int	Loop variable. Used to keep track of the current level of the Classifier Tree traversal.
Label(v)	Local	Int	A function that translates a Classifier Tree node address to the label of that node. In particular, the Classifier Tree is a complete binary tree whose nodes are labeled by consecutive integers in the order they would be visited by an inorder traversal.

Table 4.2: Variables used in Add Operation Algorithm 2

4.3 Algorithm: Remove

The purpose of Remove is to remove a process p_j from the system, which is one of the 4 operations required in any dynamic atomic snapshot implementation. Algorithm 3 provides the pseudocode for Remove, and Table 4.3 describes the variables that are used in the code. To remove a process, the process must be active in the system and an existing process p_i must invoke remove(p_j). After the invocation of this method, but before any lines are executed, the system ensures that the process invoking remove(p_j) has permission to remove p_j (see Section 3.1). Recall that, for notational convenience, p_j is used to represent the process as well as its unique ID. Rather than deallocating the memory of p_j from the system, our approach is to have each process maintain a list of processes it removed, and then when a snap is performed, the set of active processes is determined by omitting those that appear in one of the lists of removed processes. From line 3 to 6, process p_i (i.e., the process that is removing p_j from the system) reads, updates, and writes its own block back to shared memory S. To update its block, it adds p_j to the list of removed processes and increments the seq value of the block.

The rest of the code after line 6 relates to invoking the Classifier. From line 7 to line 12, process p_i creates its initial "view" of the system by scanning through the shared atomic registers of all other processes that it knows about, i.e., using the addresses that appear in its active list, and copying them into a bag called ss_0. From line 13 to 16, the Classifier Tree is traversed and updates its "view" of the system along the way by taking the bag returned to it by the Classifier algorithm. The first call to Classifier occurs at the root of the Classifier Tree, and in this first call, process p_i passes in its initial view ss_0. The details of the Classifier algorithm are provided in Section 4.5. Here, the loop is executed $\lceil \log(m) \rceil$ times because the height of the Classifier Tree is $\lceil \log(m) \rceil$ and we don't execute Classifier at the leaves of the Classifier Tree (recalling that m is an upper bound on the number of operations that can be called on this dynamic atomic snapshot object). At line 15 inside the loop, the Classifier algorithm returns a tuple where the first value is a Bag that represents p_i's updated view of the system, the second value is a list of integers representing an

updated list of active processes in the system and, the third value is the address of the Classifier Tree node that process p_i will traverse to next (i.e., it is the left or right child of the node that it visited last).

Algorithm 3 The remove operation

```
1:  procedure REMOVE(p_j)
2:      %% Before the remove operation starts, the system ensures that the invoking
        process p_i is one which added the process p_j.
3:      b_i ← S.RETRIEVE(p_i)                                           ▷ SREAD1
4:      b_i.seq ← b_i.seq + 1
5:      b_i.removeActive.add(p_j)
6:      S.WRITE(p_i, b_i)                                               ▷ SWRITE
7:      ss_0 ← {}
8:      addr_0 ← b_i.active                                             ▷ ADDR0INIT
9:      for all k in b_i.active do
10:         b_k ← S.RETRIEVE(k)                                         ▷ SREAD2
11:         ss_0.add(k, b_k)
12:     end for
13:     v ← ClassifierTreeRoot                                          ▷ PRECFirst
14:     for all ℓ ← 1, ..., ⌈log(m)⌉ do                                 ▷ PREC
15:         {ss_ℓ, addr_ℓ, v} ← Classifier(Label(v), ss_{ℓ−1}, addr_{ℓ−1})  ▷ POSTC
16:     end for
17: end procedure
```

Variable Name	Shared or Local	Data Type	Description
p_j	Local	Int	The process that is being removed.
p_i	Local	Int	The process that invoked remove(p_j).
S	Shared	Registers	The primary shared memory.
b_i	Local	Block	Locally stores the block that is copied from address p_i in the shared memory S.
ss_ℓ	Local	Bag	Holds the system view returned by the Classifier algorithm at the ℓth level of the Classifier Tree. A special case is $\ell = 0$: the bag ss_0 holds the result of p_i scanning the blocks directly from shared memory S. The bag ss_0 is the initial view that is passed into the first call to Classifier, i.e., the call that occurs at the root of the Classifier Tree.
$addr_\ell$	Local	List	Holds a list of addresses of active processes, which is returned by the Classifier algorithm at the ℓth level of the Classifier Tree. A special case is $\ell = 0$: the list $addr_0$ holds the active list that was retrieved directly from p_i's block in shared memory S. The address list $addr_0$ is the initial list of active processes that is passed into the first call to Classifier, i.e., the call that occurs at the root of the Classifier Tree.
k	Local	Int	Loop variable; used to iterate through the active list of block b_i.
b_k	Local	Block	Holds the block copied from the address k of the shared memory S.
Classifier-TreeRoot	Shared	Node	The root node address of the Classifier Tree.
v	Local	Node	Holds the address of a Classifier Tree node.
m	Static	Int	Upper bound on the number of operation calls that this dynamic atomic snapshot object supports.
ℓ	Local	Int	Loop variable. Used to keep track of the current level of the Classifier Tree traversal.
Label(v)	Local	Int	A function that translates a Classifier Tree node address to the label of that node. In particular, the Classifier Tree is a complete binary tree whose nodes are labeled by consecutive integers in the order they would be visited by an inorder traversal.

Table 4.3: Variables used in Remove Operation Algorithm 3

4.4 Algorithm: Snap

The purpose of Snap is to create a snapshot of the shared atomic registers belonging to active processes in the system, which is one of the 4 operations required in any dynamic atomic snapshot implementation. Algorithm 4 provides the pseudocode for Snap, and Table 4.4 describes the variables that are used in the code. From line 2 to 4, process p_i (i.e., the process that is performing the Snap operation) reads, updates, and writes its own block back to shared memory S. To update its block, it increments the seq value of the block.

The rest of the code after line 4 relates to invoking the Classifier. From line 5 to line 10, process p_i creates its initial "view" of the system by scanning through the shared atomic registers of all other processes that it knows about, i.e., using the addresses that appear in its active list, and copying them into a bag called ss_0. From line 11 to 14, the Classifier Tree is traversed and updates its "view" of the system along the way by taking the bag returned to it by the Classifier algorithm. The first call to Classifier occurs at the root of the Classifier Tree, and in this first call, process p_i passes in its initial view ss_0. The details of the Classifier algorithm are provided in Section 4.5. Here, the loop is executed $\lceil \log(m) \rceil$ times because the height of the Classifier Tree is $\lceil \log(m) \rceil$ and we don't execute Classifier at the leaves of the Classifier Tree (recalling that m is an upper bound on the number of operations that can be called on this dynamic atomic snapshot object). At line 13 inside the loop, the Classifier algorithm returns a tuple where the first value is a Bag that represents p_i's updated view of the system, the second value is a list of integers representing an updated list of active processes in the system and, the third value is the address of the Classifier Tree node that process p_i will traverse to next (i.e., it is the left or right child of the node that it visited last).

From line 16 to 22, removeSet is created by iterating through all the blocks that exist in the system view returned by the final call to Classifier, and remembering all addresses that appear in any list of removed processes. From line 23 to 27, the processes in removeSet are removed from the final snapshot before it is returned.

Algorithm 4 The snap operation

```
1: procedure SNAP()
2:     b_i ← S.RETRIEVE(p_i)                                        ▷ SREAD1
3:     b_i.seq ← b_i.seq + 1
4:     S.WRITE(p_i, b_i)                                            ▷ SWRITE
5:     ss_0 ← {}
6:     addr_0 ← b_i.active                                          ▷ ADDR0INIT
7:     for all k in b_i.active do
8:         b_k ← S.RETRIEVE(k)                                      ▷ SREAD2
9:         ss_0.add(k, b_k)
10:    end for
11:    v ← ClassifierTreeRoot                                       ▷ PRECFirst
12:    for all ℓ ← 1, ..., ⌈log(m)⌉ do                              ▷ PREC
13:        {ss_ℓ, addr_ℓ, v} ← Classifier(Label(v), ss_{ℓ−1}, addr_{ℓ−1})   ▷ POSTC
14:    end for
15:    removeSet ← list()
16:    for all addr in addr_{ss_{log(m)+1}} do
17:        for all remAddr in ss_{log(m)+1}.retrieve(addr).removeActive do
18:            if removeSet.exists(remAddr) is False then
19:                removeSet.add(remAddr)
20:            end if
21:        end for
22:    end for
23:    for all addr in addr_{ss_{log(m)+1}} do
24:        if removeSet.exists(addr) is True then
25:            ss_{log(m)+1}.erase(addr)
26:        end if
27:    end for
28:    return ss_{log(m)+1}
29: end procedure
```

Variable Name	Shared or Local	Data Type	Description
p_i	Local	Int	The process that invoked the snap operation.
S	Shared	Registers	The primary shared memory.
b_i	Local	Block	Locally stores the block that is copied from address p_i in the shared memory S.
ss_ℓ	Local	Bag	Holds the system view returned by the Classifier algorithm at the ℓth level of the Classifier Tree. A special case is $\ell = 0$: the bag ss_0 holds the result of p_i scanning the blocks directly from shared memory S. The bag ss_0 is the initial view that is passed into the first call to Classifier, i.e., the call that occurs at the root of the Classifier Tree.
$addr_\ell$	Local	List	Holds a list of addresses of active processes, which is returned by the Classifier algorithm at the ℓth level of the Classifier Tree. A special case is $\ell = 0$: the list $addr_0$ holds the active list that was retrieved directly from p_i's block in shared memory S. The address list $addr_0$ is the initial list of active processes that is passed into the first call to Classifier, i.e., the call that occurs at the root of the Classifier Tree.
k	Local	Int	Loop variable; used to iterate through the active list of block b_i.
b_k	Local	Block	Holds the block copied from the address k of the shared memory S.
Classifier-TreeRoot	Shared	Node	The root node address of the Classifier Tree.
v	Local	Node	Holds the address of a Classifier Tree node.
m	Static	Int	Upper bound on the number of operation calls that this dynamic atomic snapshot object supports.
ℓ	Local	Int	Loop variable. Used to keep track of the current level of the Classifier Tree traversal.
Label(v)	Local	Int	A function that translates a Classifier Tree node address to the label of that node. In particular, the Classifier Tree is a complete binary tree whose nodes are labeled by consecutive integers in the order they would be visited by an inorder traversal.
removeSet	Local	Set	A list of integers that correspond to process IDs that have been removed from the system.
addr	Local	Int	Loop variable; used to iterate over the list of addresses returned by the final call to Classifier.
remAddr	Local	Int	Loop variable; used to iterate over the list of addresses in the removeActive list of each process.

Table 4.4: Variables used in Snap Operation Algorithm 4

4.5 Algorithm: Classifier

Each time that a process p_i performs an operation on the atomic snapshot object (i.e., add, remove, update, snap), the final part of the operation consists of the process p_i doing a traversal of the Classifier Tree, essentially to integrate its changes with the changes that the other processes have been making. This traversal starts at the root of the Classifier Tree, and goes downward one level at a time, and at each level the process calls the Classifier method, which we describe in the remainder of this subsection.

There are two main parts to the Classifier method. At the start, each process starts with a bag $ss_{\ell-1}$ that represents its current view of the system. In the first part, the process writes its own bag to the Container at the current node of the Classifier Tree, and then merges together all bags that are in the Container. This merges together the most recent observations of the system made by all other processes that have passed through this Classifier Tree node so far. The second part of the Classifier method determines whether the next step of the Classifier Tree traversal goes to the left child or the right child. At a high level, the traversal always goes left until all descendant leafs of the Classifier Tree are full, and then all future traversals go right. To actually implement this idea, each Classifier Tree node is labeled with an integer K, and after merging together all bags in the Container, the seqSum of this merged bag is compared with K (i.e., seqSum $\leq$ K means go left, and seqSum $>$ K means go right). These node labels are defined by assigning consecutive integers via the order that the nodes would be visited in an inorder traversal of the tree. Finally, if the process traverses to the left child, it keeps its own view of the system $ss_{\ell-1}$ as its new view ss_{ℓ}; but, if the process traverses to the right, it uses the bag it got from merging together all bags in the Container as its new view ss_{ℓ}.

Algorithm 5 provides the pseudocode for Classifier, and Table 4.5 describes the variables that are used in the code. At line 2, the current process P_i updates its value in the Container C (of the current node u) with the $ss_{\ell-1}$ bag that is passed via parameter. Then in lines 3 to 4, we compute an updated list of active processes in the system and a local copy of the Container consisting of all the bags belonging

to these active processes. The details of how this is carried out are described in Section 4.7. At line 5, mergeRecent is computed by merging together all bags in the Container belonging to active processes. The details of how this is carried out are described in Section 4.8. Then, we compute the seqSum of this mergeRecent result (as described in Section 4.9), and there can be two possible outcomes based on the relationship between the values of seqSum and K. If the seqSum is greater than K, then mergeRecent is updated by performing another merge on the Container, and then the new mergeResult along with the list of active processes and the address of the right child are returned (see lines 7 and 8). Otherwise, if seqSum $\leq$ K, then the old values that the process had at the start of the method are returned, i.e., $ss_{\ell-1}$ and $addr_{\ell-1}$, along with the address of the left child (see line 10). While returning the values in both cases, a tuple is returned where the first value is a Bag, the second value is a list of integer addresses and the last value is the address of a Classifier Tree node.

Algorithm 5 Classifier Process of node u in Classifier Tree.

1: **procedure** CLASSIFIER(K, $ss_{\ell-1}$, $addr_{\ell-1}$)
2: $\quad$ $\texttt{C.WRITE}(\texttt{p}_\texttt{i}, \texttt{ss}_{\ell-1})$
3: $\quad$ $\texttt{currentActive} \leftarrow \texttt{addr}_{\ell-1}$
4: $\quad$ $\{\texttt{updatedActiveList}, \texttt{localContainer}\} \leftarrow \texttt{GetUpdatedActiveList}(\texttt{currentActive})$
5: $\quad$ $\texttt{mergeRecent} \leftarrow \texttt{GetMergeRecent}(\texttt{updatedActiveList}, \texttt{localContainer})$
6: $\quad$ **if** $\texttt{getSeqSum}(\texttt{mergeRecent}, \texttt{updatedActiveList}) > \texttt{K}$ **then**
7: $\quad\quad$ $\{\texttt{updatedActiveList}, \texttt{localContainer}\} \leftarrow \texttt{GetUpdatedActiveList}(\texttt{currentActive})$
8: $\quad\quad$ $\texttt{newMergeRecent} \leftarrow \texttt{GetMergeRecent}(\texttt{updatedActiveList}, \texttt{localContainer})$
9: $\quad\quad$ **return** $\{\texttt{newMergeRecent}, \texttt{updatedActiveList}, \texttt{u.right}\}$
10: $\quad$ **else**
11: $\quad\quad$ **return** $\{\texttt{ss}_{\ell-1}, \texttt{addr}_{\ell-1}, \texttt{u.left}\}$
12: $\quad$ **end if**
13: **end procedure**

Variable Name	Shared or Local	Data Type	Description
K	Local	Int	Input variable. Holds the value of the label of the current node.
$ss_{\ell-1}$	Local	Bag	Input variable. Holds the snapshot from the $\ell - 1^{th}$ level of the Classifier Tree which is the process' current 'view' of the system.
$addr_{\ell-1}$	Local	Bag	Input variable. Holds the list of active addresses from the $\ell - 1^{th}$ level of the Classifier Tree which is the list of processes that p_i thinks are in the system.
p_i	Local	Int	This is the unique ID of the process that is currently executing the method.
C	Shared	Container	The Container stored at this node of the Classifier Tree. It consists of the bags that have been written to it by the processes that have visited this Classifier Tree node so far.
current-Active	Local	List	Holds the list of processes that p_i thinks are active at the start of this call to Classifier.
updated-ActiveList	Local	List	Holds an updated list of active processes that p_i finds by calling GetUpdatedActiveList.
local-Container	Local	Container	Holds the local copy of all the bags in the Container C of the current Classifier Tree node for each address in updatedActiveList.
merge-Recent	Local	Bag	Holds the bag that is produced by merging all the bags from Container C.
newMerge-Recent	Local	Bag	Holds the bag that is produced by merging all the bags from Container C for a second time. This is only performed at line 8, which occurs after deciding that the traversal of the Classifier Tree will continue down to the right child.
u.right	Shared	Node	u is the current node of the Classifier Tree on which the current call to Classifier is being executed. u.right is the address of the right child of the current node u.
u.left	Shared	Node	u.left is the address of the left child of the current node u of the Classifier Tree.

Table 4.5: Variables used in Algorithm 5

4.6 Algorithm: AddToClassifierTree

When calling Classifier at a particular node in the Classifier Tree, there are two actions involving the Container at the node: a process writes a bag to the Container at its own address, and, a process performs a merge on all bags in the Container at addresses belonging to processes it thinks are in the system. However, to perform these actions, the Container has to have shared registers where the bags can be written to or read from, and the system doesn't know in advance how many processes are going to be added to the system (and doesn't know their process IDs). So, we provide an AddToClassifierTree method that can be called whenever a process is added to the system, and the result is that each Container will have a shared register whose address corresponds to the process' ID.

The purpose of AddToClassifierTree is to allocate a new memory location in each Container in the Classifier Tree for a new process p_j. Algorithm 6 provides the pseudocode for AddToClassifierTree, and Table 4.6 describes the variables that are used in the code. The algorithm uses a depth-first approach to traverse the Classifier Tree. At each node of the Classifier Tree there is a Container C. At line 5, the unique ID of process p_j is used to create a new memory location in C at the current node u that points to a new shared atomic register. We assume that this shared memory allocation is possible using an underlying system operation called INITIATE.

Algorithm 6 Add a memory location p_j to Container C at node u

```
1: procedure ADDTOCLASSIFIERTREE(u, p_j)
2:     if u == null then
3:         return
4:     end if
5:     u.C.INITIATE(p_j)
6:     AddToClassifierTree(u.left, p_j)
7:     AddToClassifierTree(u.right, p_j)
8: end procedure
```

Variable Name	Shared or Local	Data Type	Description
u	Shared	Node	Holds the address of a Classifier Tree node.
P_j	Local	Address	The unique ID of process p_j, to be added to the Container at the Classifier Tree node u.
u.right	Shared	Node	u.right is the address of the right child of the current node u.
u.left	Shared	Node	u.left is the address of the left child of the current node u.

Table 4.6: Variables used in Algorithm 6

4.7 Algorithm: GetUpdatedActiveList

The main difference between a traditional Snapshot object and a Dynamic Snapshot object is that, traditionally, the entire list of active processes in the system is known at all times, whereas in the dynamic version, processes may be added and removed at arbitrary times. In our model, each process that joins the system is provided a list of "initial processes" that were active when the system started running, but this list can become out-of-date as other processes in the system call add and remove operations. So, each process will maintain a list of processes (called *active*) that it thinks are currently active in the system, but, we need to provide a mechanism that a process can use to update this list to improve its accuracy. At a high level, a process that performs an add or remove will publish this information to its dedicated SWMR register in shared memory, and then other processes can update their active lists by scanning the shared memory and seeing these published updates. There is an obvious difficulty with this approach: a process can only read the registers of processes that it knows about, so for example, if a process p_1 is not yet aware that process p_2 has been added to the system, and process p_2 adds process p_3 to the system, how will process p_1 find out about the existence of process p_3? To deal with this issue, we exploit the fact that each process must be added to the system via some unique chain of Add operations originating from a process that was initially in the system, and so our algorithm can be thought of as traversing through a directed tree consisting of

these chains.

GetUpdatedActiveList is performed inside every execution of Classifier, and each execution of Classifier occurs at some node in the Classifier Tree. The purpose of GetUpdatedActiveList is to search and find new addresses of the register of the active processes that it did not know about and also, during this process this function creates a local copy of the Container C that is located at this Classifier Tree node by retrieving the bags at addresses from the newly-updated active list.

Variable Name	Shared or Local	Data Type	Description
oldActive	Local	List	Input variable; It contains all the addresses that are already known by the process calling this method.
updated-Active-List	Local	List	Holds the current up-to-date list of active process' addresses
local-Container	Local	Container	A local Container that holds, for each address in updatedActiveList, a copy of the corresponding bag in the shared Container C of the current node of the Classifier Tree.
toScan	Local	List	Stores the addresses of the processes that need to be scanned in the current iteration.
newTo-Scan	Local	List	Stores new addresses that will be scanned in the next iteration of the algorithm.
C	Shared	Container	The Container C located at the current node of the Classifier Tree.
$localss_j$	Local	Bag	Holds a copy of the bag located at address j of the shared Container C
k	Local	Address	Loop variable. This is used to iterate through the addresses in updatedActiveList at line 9
r_k	Local	Block	Holds the block found at the address k of the bag $localss_j$
$address_{p_j}$	Local	Address	Loop variable. This is used to iterate through the list of addresses in r_k.active at line 12

Table 4.7: Variables used in Algorithm 7

Algorithm 7 provides the pseudocode for GetUpdatedActiveList, and Table 4.7 describes the variables that are used in the code. The GetUpdatedActiveList method

proceeds as follows. The process calling the method provides a list of register addresses of "known" active processes, which is passed in the *oldActive* argument. The algorithm starts by initializing the set of active processes using the input argument oldActive. Let j be the current address that is being scanned. For each j, the algorithm retrieves the bag located at address j of C (line 8) and copies it to the localContainer (line 9). In this copied bag $localss_j$, the algorithm uses its currently known set of active processes (updatedActiveList) to search through every block of the bag located at addresses it knows about. In each such block, it looks at the list of addresses written in the "active" variable, and searches for any addresses that are not already in updatedActiveList (line 10 to 20). At lines 15 - 17, the newly-found addresses are added to updatedActiveList, but also added to a newToScan list that will form the next list of addresses to be scanned (lines 24 and 5). This process continues until the algorithm cannot find any new addresses, i.e., the newToScan list is empty after scanning the entire toScan list (line 7 to 24).

Algorithm 7 Calculate active nodes inside a node in Classifier node u

Input *oldActive*: A list of register addresses

Output *updatedActiveList*: Updated list of register addresses found in the Container C of the current node based on the initial set of addresses in *oldActive*.

localContainer: Copy of the Container C from where the *updatedActiveList* is created.

```
1: procedure GetUpdatedActiveList(oldActive)
2:     updatedActiveList ← oldActive
3:     localContainer ← {}
4:     toScan ← oldActive
5:     while toScan is not empty do
6:         newToScan ← []
7:         for all j ∈ toScan do
8:             localss_j ← C.RETRIEVE(j)
9:             localContainer.write(j, localss_j)
10:            if localss_j is not null then
11:                for all k ∈ updatedActiveList do
12:                    r_k ← localss_j.retrieve(k)
13:                    if r_k is not null then
14:                        for all address_{p_j} ∈ r_k.active do
15:                            if updatedActiveList.exists(address_{p_j}) is False then
16:                                newToScan.add(address_{p_j})
17:                                updatedActiveList.add(address_{p_j})
18:                            end if
19:                        end for
20:                    end if
21:                end for
22:            end if
23:        end for
24:        toScan ← newToScan
25:    end while
26:    return {updatedActiveList, localContainer}
27: end procedure
```

4.8 Algorithm: GetMergeRecent

One of the purposes of the Classifier Tree is to combine information: each process has a certain “view” of the system, they will each come to the Classifier Tree to write down this view, and then collect all the views from other processes in an attempt to combine them into a coherent, up-to-date view. In particular, when visiting a node in the Classifier Tree, a process will arrive with a list of processes that it thinks are active, and will gather and combine the most recent information that was written to the node’s Container by those processes. The purpose of GetMergeRecent is to implement this action: it takes a Container and a list of addresses as input, gets the bags from the Container at those addresses, and performs a merge on the bags.

Algorithm 8 provides the pseudocode for GetMergeRecent, and Table 4.8 describes the variables that are used in the code. The input consists of activeList (a list of addresses) and localContainer (a Container of bags). The loop at line 4 retrieves the bag located at each j $\in$ activeList from the localContainer. The nested loop (from line 6 to 17) traverses the activeList, and for each k $\in$ activeList, retrieves block BL_k from bag j in localContainer, and updates the mergeResult bag as follows: if mergeResult doesn’t have a block at address k yet, then write BL_k there; otherwise, it writes BL_k to the block at address k if BL_k has a larger seq value than the existing block. In other words, after looping through all values of j and k, the block at address k in mergeResult should contain the block with largest seq value among all blocks with address k, taken over all bags in localContainer whose addresses appear in the activeList. The calculated mergeResult is returned at the end of the procedure.

Algorithm 8 Calculate the *mergeRecent* bag of a given node in Classifier Tree

Input *activeList*: A list of register addresses

localContainer: A copy of the Container at the current node

Output *mergeResult*: Returns a bag

```
1: procedure GetMergeRecent(activeList, localContainer)
2:     visited ← list()
3:     mergeResult ← {}
4:     for all j ∈ activeList do
5:         localss_j ← localContainer.retrieve(j)
6:         for all k ∈ activeList do
7:             BL_k ← localss_j.retrieve(k)
8:             if BL_k is not null then
9:                 if visited.exists(k) is False then
10:                    visited.add(k)
11:                    mergeResult.add(k, BL_k)
12:                end if
13:                if BL_k.seq ≥ mergeResult.retrieve(k).seq then
14:                    mergeResult.write(k, BL_k)
15:                end if
16:            end if
17:        end for
18:    end for
19:    return mergeResult
20: end procedure
```

Variable Name	Shared or Local	Data Type	Description
activeList	Local	List	Input variable. Holds the list of addresses of the processes that will be used to calculate the mergeRecent bag.
local-Container	Local	Container	Input variable. Holds multiple bags for each address in activeList.
visited	Local	Set	Keeps track of the processes' addresses that are already added to mergeRecent result bag.
merge-Result	Local	Bag	Holds the final result of this procedure which is the mergeRecent bag of the given set of processes.
j	Local	Address	Loop variable; used to iterate through the activeList at line 4.
$localss_j$	Local	Bag	Holds the snapshot (as a bag) located at address j in the Container C.
k	Local	Address	Loop variable; used to iterate through the activeList at line 6.
BL_k	Local	Block	Holds the block located at key k in the $localss_j$.

Table 4.8: Variables used in Algorithm 8

4.9 Algorithm: GetSeqSum

Recall that each block has a seq variable that is used to determine the freshness of information: the seq value gets incremented each time that the block is accessed, so a higher seq value corresponds to a more recent access. We extend this idea to an entire bag consisting of blocks: we get an integer called seqSum by adding up the seq values of all blocks in the bag, and for two different bags, we can compare their seqSum values to determine which one is more "recent".

The purpose of GetSeqSum is to calculate the sum of the seq values from a given bag. Algorithm 9 provides the pseudocode for GetSeqSum, and Table 4.9 describes the variables that are used in the code. The first input parameter is *bag*, which is a bag whose seqSum we want to compute, and the second input parameter is *addrs*, which is a list of addresses of blocks in the bag (which will allow us to iterate through the contents of the bag). In the loop (line 3 to 5), the algorithm traverses *addrs* and

for each j ∈ *addrs* retrieves the block from *bag* and updates the seqSum variable. The result of this procedure is the final value of seqSum.

Algorithm 9 Calculate the seq size of a given bag

```
1: procedure getSeqSum(bag, addrs)
2:     seqSum ← 0
3:     for all j ∈ addrs do
4:         seqSum ← seqSum + bag.retrieve(j).seq
5:     end for
6:     return seqSum
7: end procedure
```

Variable Name	Shared or Local	Data Type	Description
bag	Local	Bag	Input variable. Holds the bag that will be used to calculate the seq sum.
addrs	Local	List	Input variable. Holds the keys of the processes that are in the input bag.
seqSum	Local	Int	Holds the updated sum of the seq values. This is the final result of the procedure.
j	Local	Int	Loop variable; used to iterate through *addrs* at line 3.

Table 4.9: Variables used in Algorithm 9

5 Algorithm Observations

In this section, we will observe certain facts by directly looking at the algorithm code. These facts do not require proofs and will be used in the proofs of correctness in later sections.

5.1 Common anchor points in the Algorithms: 1, 2, 3, 4

In the later sections where we will show the proof of correctness of the algorithms, there will be situations where we want to prove the same fact about multiple al-

gorithms simultaneously instead of repeatedly writing out the argument but with modified line numbers. In particular, when proving facts about our Add, Remove, Update, and Snap algorithms, there are lines of code that are common to all four algorithms, but at different line numbers. In this subsection, we provide a table where we list out the lines that are common among these algorithms. We also associate keywords to these lines so that we can write out the keyword once instead of writing out all four line numbers. We call these keywords as "anchor" points in the algorithms. The anchor point keywords are also written as inline comments in the pseudocode.

Anchor Name	Line in Add [2]	Line in Remove [3]	Line in Snap [4]	Line in Update [1]	Description
SREAD1	4	3	2	2	This line refers to the line where the process (that is executing the current operation) read its own block from the shared memory S.
SWRITE	7	6	4	5	After reading its own block from the shared memory, the process updates the block's seq value and also depending on the type of operation it updates active, removeActive or value property of the block. Then, the process writes back the updated block into the shared memory S. This line refers to that line when the current process writes back the updated block to the shared memory S.
ADDR0INIT	9	8	6	7	This line refers to the line where $addr_0$ is initialized.
SREAD2	11	10	8	9	This line refers to the line where the process reads the blocks of the other processes which are found in its active list.
PRECFirst	14	13	11	12	If we want to refer to the point in time before the first call to the Classifier, then we can use this anchor point.
PREC	15	14	12	13	When a process is at level ℓ and just before starting the Classifier algorithm, if we want to refer to that point in time, then we can use this anchor point.
POSTC	16	15	13	14	This line is used to talk about the values after the Classifier algorithm ends.

Table 5.1: Explanation of common anchor points.

5.2 Remove Operation Observations

In this section, we make some observations about what can be concluded once any process p_s "sees" that a Remove(p_j) operation has been performed, i.e., once it reads a block where the removeActive list contains p_j. Let op_s be any operation executed by p_s. Let SS_{all} be a set of bags of all ss that are created during op_s. More formally, $SS_{all} = \{ss_0, ss_1, ..., ss_{\lceil \log(m) \rceil}\}$. Here, ss_0 is created after reading from the shared

memory S before the first call to the Classifier at line *PRECFirst* and rest of the ss are considered at line *POSTC*. Let any bag $ss_a \in SS_{all}$. Similarly let $ADDR_{all}$ $= \{addr_0, addr_1, ..., addr_{\lceil \log(m) \rceil}\}$ at the same lines as ss and $addr_a \in ADDR_{all}$. We also consider that there exists an address p_i where $p_j \in BL_{p_i,ss_a,op_s}$.removeActive. In other words, in the bag ss_a returned by Classifier in the loop iteration $\ell = a$ or before the first call to the Classifier for ss_0, at least one of the blocks has a removeActive variable containing p_j. We observe the following facts:

Fact 1. Before the execution of line *SWRITE* during the operation where p_j is removed, there exists no p_j in any removeActive list of any block. This is because:

i. The first time that the value p_j is added to any removeActive list in shared memory is during the operation where p_j is removed.

ii. Since there exists only one operation where p_j is removed and there exists no other lines of codes where removeActive is modified in shared memory, the fact holds.

Fact 2. The remove operation (let op_i) where p_j is removed must be executed by process p_i and this remove operation started before the time when $p_j \in BL_{p_i,ss_a,op_s}$.removeActive during op_s. This is because:

i. Using Fact 1, the line *SWRITE* of the remove operation where p_j is removed must have been executed before the line where $p_j \in BL_{p_i,ss_a,op_s}$.removeActive during op_s. Therefore, the remove operation op_i started before this time.

ii. At line *SWRITE* during op_i, the updated block is written back to the shared memory S after appending p_j in the removeActive list. Therefore, the block BL_{p_i,S,op_i} exists at this time. Since BL_{p_i,S,op_i} exists, BL_{p_i,S,op_i}.seq and $p_j \in BL_{p_i,S,op_i}$.removeActive also exists at this time.

iii. We know that during op_s, p_j is found in BL_{p_i,ss_a,op_s}.removeActive. Note that removeActive list for any block in any bag is never updated or manipulated during the Classifier algorithm and from Section 3.1, a process

can only be removed once. Therefore, the only way p_j can be found in the removeActive list of a block at address p_i is when p_i is the process that removed p_j. In other words, p_i must be process that executed op_i.

iv. In the remaining facts we will refer to p_i as the process that executed op_i where the p_j was removed.

Fact 3. BL_{p_i,ss_a,op_s} exists and $p_i \in addr_a$ during op_s. This is because:

i. Since $p_j \in BL_{p_i,ss_a,op_s}$.removeActive during op_s, the block BL_{p_i,ss_a,op_s} is not null. Therefore, this block exists and was found at address p_i in the bag ss_a during op_s.

ii. The fact that block p_i is written into ss_a during Classifier means that $p_i \in addr_a$.

Fact 4. The Add operation where p_j is added is partially ordered before op_i. This is because:

i. From the model definition (Section 3.1): any process p_j can only be removed once.

ii. From the model definition (Section 3.1): only the process that added p_j can remove the process p_j.

iii. Therefore, because of SWMR property, the add operation where p_j is added must be completed before the start of op_i.

Fact 5. At any time when $p_j \in BL_{p_i,ss_a,op_s}$.removeActive during op_s, we know that BL_{p_i,ss_a,op_s}.seq $\geq BL_{p_i,S,op_i}$.seq. This is because:

i. No items are ever removed from removeActive list of any block. Therefore, once p_j is added in the block p_i in shared memory S, it is never removed from that block.

ii. The process p_j is added exactly once to removeActive in block p_i in shared memory S. Therefore, any block where the removeActive list contains p_j

must be an exact copy or a more recently updated version of the block BL_{p_i,S,op_i}.

iii. If it is an exact copy of the block BL_{p_i,S,op_i}, then its seq value is equal to BL_{p_i,S,op_i}.seq. If it is an updated block than BL_{p_i,S,op_i}, then its seq value will be strictly larger than BL_{p_i,S,op_i}.seq.

5.3 Add Operation Observations

In this section, we make some observations about what can be concluded once any process p_s "sees" that an Add(p_j) operation has been performed. This occurs if p_j shows up in a list of active addresses but p_j wasn't one of the processes initially in the system at start up. More specifically, let op_s be any operation executed by p_s. Let SS_{all} be a set of bags of all ss that are created during op_s. More formally, $SS_{all} = \{ss_0, ss_1, ..., ss_{\lceil \log(m) \rceil}\}$. Here, ss_0 is created after reading from the shared memory S before the first call to the Classifier at line *PRECFirst* and rest of the ss are considered at line *POSTC*. Let any bag $ss_a \in SS_{all}$. Similarly let $ADDR_{all}$ $= \{addr_0, addr_1, ..., addr_{\lceil \log(m) \rceil}\}$ at the same lines as ss and $addr_a \in ADDR_{all}$. We consider such ss_a and $addr_a$ where during the execution of op_s, $p_j \in addr_a$ and $p_j \notin procList_\alpha$. We observe the following facts:

Fact 1. Since $p_j \notin procList_\alpha$ but $p_j \in addr_a$, it must be a process that was added in the system.

Fact 2. Since p_j was added in the system, there exists a process p_i which executed an Add(p_j) operation and from Section 3.1, there can be only one such as Add(p_j) operation. In the remaining observations below, we will refer to this operation as op_i.

Fact 3. Before the execution of line *SWRITE* during op_i, there exists no p_j in any active list or removeActive list in any block. This is because:

i. The first and only time that p_j is added to an active list is at the line before *SWRITE* in Add(p_j)=op_i, and so the first time it is written to a block is at line *SWRITE* during op_i.

Fact 4. At any time where $p_j \in BL_{p_i,ss_a,op_s}$.active during op_s, the block BL_{p_i,S,op_i} exists. This is because:

i. Using Fact: 3, the line *SWRITE* of op_i must have already been executed.

ii. Therefore, at this time, we know that line *SWRITE* of op_i has already been executed and the block BL_{p_i,S,op_i} exists.

Fact 5. At any time where $p_j \in BL_{p_i,ss_a,op_s}$.active during op_s, we know that BL_{p_i,ss_a,op_s}.seq $\geq BL_{p_i,S,op_i}$.seq. This is because:

i. From Fact: 4, we know that the line *SWRITE* of op_i has previously occurred. The first and only time where p_j is added to an active list is when p_i adds it to block p_i in S during op_i.

ii. No items are ever removed from active list of any block. Therefore, once p_j is added to the active list in the block in shared memory S, it is never removed from that block. Therefore, any block where the active list contains p_j must be an exact copy or a more updated version of the block BL_{p_i,S,op_i}.

iii. If it is an exact copy, then BL_{p_i,ss_a,op_s}.seq $= BL_{p_i,S,op_i}$.seq. If it is a more recently updated version of the block BL_{p_i,S,op_i}, then its seq value will be strictly larger than BL_{p_i,S,op_i}.seq.

6 Linearization

Recall that the linearizability property of a snapshot object implementation means that we can assign to each operation a distinct point in time (e.g., place them at distinct points on a timeline) that occurs somewhere between the start and end of the execution of the operation in such a way that, if we imagine a sequential execution of the operations at these points in time, the snap operations have the same outcome as they do in the concurrent execution of our snapshot implementation.

In this section, we start by defining a high-level idea of our linearization ordering. Then, based on that idea, we define a comparison algorithm that takes two operations

and tells us which one occurs earlier in our ordering. After that, we prove that this algorithm produces a total ordering of all the operations.

6.1 High-Level Idea

Two operations are said to follow *partial ordering* if one terminates before the other begins. In those cases, it is clear how they should be linearized. For operations that overlap, there are more complicated details required depending on the types of operations involved, as described below. In what follows, the add, remove, and update operations will be collectively referred to as *modify* operations.

1. At first, we will linearize all the snap operations.

   (a) While linearizing two snap operations, we will look whether they follow partial ordering. If they do, then the process that terminates first should be linearized first.

   (b) If the snap operations do not follow partial ordering, then we will linearize them by the domination property. We connect the domination property with the Classifier Tree leaf labels and then linearize the snap operations in order by increasing leaf labels.

   (c) The snap operations that end up at the same Classifier Tree leaf will be linearized by their process ids.

2. The second step is to consider each modify operation (i.e., update, add, remove) and insert them into the linearized list of snap operations.

   We will compare each modify operation against the linearized list of snap operations and insert the modify operation at the appropriate place so that the linearization order is preserved. To do this, we first define how to directly compare a modify and a snap: we look at the two bags that are output by the Classifier for the two operations, take from each bag the block belonging to the process performing the modify operation, and compare the sequence numbers of these blocks. If the modify operation's bag has the smaller of these sequence

numbers, then it is linearized before the snap. We can apply this comparison to each snap operation to figure out where the modify operation should be inserted within the linearized list of snap operations.

3. After the end of the second step, there will be buckets of un-linearized modify operations in between two consecutive linearized snap operations. In the third step, we will consider each bucket of un-linearized modify operations and linearize them. To do this, certain modify operations are linearized before the others, i.e., add before update before remove. This is because a process p_i should be added to the system before p_i performs an update to its shared register, and p_i should not be removed from the system until after it has been added and has performed its updates. Within operations of the same type, they are ordered by the ID of the process that performs them.

6.2 Linearization Definition

Algorithm 10 provides the pseudocode that implements the linearization idea that was described in Section 6.1. In particular, given two operations as input, the procedure will return true if and only if the first input is linearized before the second input. Table 6.1 describes the variables used in Algorithm 10. The values of $ModOrd_{op_1}$ and $ModOrd_{op_2}$ are assumed to have their value already set depending on the types of operations, i.e., they are set to 1,2,3 if op_i is add, update, or remove, respectively. These values are used to implement the priority ordering of the various modify operations, i.e., add should occur before update, and update should occur before remove.

Algorithm 10 Linearization comparison algorithm

Input op_1: An operation of add, remove, update or snap type.

Input op_2: An operation of add, remove, update or snap type.

Output Returns true if op_1 appears before op_2 in the linearization order. Otherwise, returns false.

```
1: procedure CheckLinDefOrder(op1, op2)
2:     if op1 and op2 follow partial ordering then
3:         return (op1 is partially ordered before op2)
4:     end if
5:     if Type_op1 is snap and Type_op2 is snap then
6:         if CLLop1 == CLLop2 then
7:             return (ProcId_op1 < ProcId_op2)
8:         else
9:             return (CLLop1 < CLLop2)
10:        end if
11:    end if
12:    if One of Type_op1 and Type_op2 is snap and other one is modify then
13:        if Type_op1 is modify then
14:            return (BL_{ProcId_op1,SS_op1}.seq ≤ BL_{ProcID_op1,SS_op2}.seq)
15:        else
16:            return (BL_{ProcId_op2,SS_op1}.seq < BL_{ProcID_op2,SS_op2}.seq)
17:        end if
18:    end if
19:    if Type_op1 is modify and Type_op2 is modify then
20:        rightSnap_op1 ← FindRightSnapOperation(op1)
21:        rightSnap_op2 ← FindRightSnapOperation(op2)
22:        if rightSnap_op1 == rightSnap_op2 then
23:            if Type_op1 == Type_op2 then
24:                return (ProcId_op1 < ProcId_op2)
25:            else
26:                return (ModOrd_op1 < ModOrd_op2)
27:            end if
28:        else
29:            return CheckLinDefOrder(rightSnap_op1, rightSnap_op2)
30:        end if
31:    end if
32: end procedure
```

Variable Name	Shared or Local	Data Type	Description
$\texttt{SS}_{\texttt{op}_1}$	Local	Bag	Final snapshot bag returned from Classifier for operation op_1.
$\texttt{SS}_{\texttt{op}_2}$	Local	Bag	Final snapshot bag returned from Classifier for operation op_2.
$\texttt{Type}_{\texttt{op}_1}$	Local	String	The operation type of op_1. There are 4 types of operations: “snap”, “update”, “add” and “remove”. The update, add and remove operations belong to the “modify” superset.
$\texttt{Type}_{\texttt{op}_2}$	Local	String	The operation type of op_2. There are 4 types of operations: “snap”, “update”, “add” and “remove”. The update, add and remove operations belong to the “modify” superset.
$\texttt{ProcId}_{\texttt{op}_1}$	Local	Int	The process ID of the process that performed op_1.
$\texttt{ProcId}_{\texttt{op}_2}$	Local	Int	The process ID of the process that performed op_2.
$\texttt{CLLop}_1$	Local	Int	The label of the leaf node in the Classifier Tree from where $\texttt{SS}_{\texttt{op}_1}$ is returned. CLL is short for *Classifier Leaf Label*.
$\texttt{CLLop}_2$	Local	Int	The label of the leaf node in the Classifier Tree from where $\texttt{SS}_{\texttt{op}_2}$ is returned
$\texttt{ModOrd}_{\texttt{op}_1}$	Local	Int	Defined to be 1,2 or 3 respectively if op_1 is add, update or remove operation.
$\texttt{ModOrd}_{\texttt{op}_2}$	Local	Int	Defined to be 1,2 or 3 respectively if op_2 is add, update or remove operation.

Table 6.1: Variables used in Algorithm 10

Algorithm 11 Finding the snap operation that is linearized to the immediate right of a modify operation

Input $modifyOp$: An operation of add, remove or update type.

Output Returns the snap operation that is on the immediate right of the given modify operation $modifyOp$. At the end of the mth operation, we assume that the system automatically performs a snap operation. Therefore, it is guaranteed that there will always exist a valid immediate right snap operation.

```
1: procedure FindRightSnapOperation(modifyOp)
2:     allOperations ← System generated list of all m+1 operations.
3:     snapOperations ← allOperations.filter(snap operations)
   %% Bubble Sort the list of snap operations using our linearization ordering.
4:     for i ← 0 to length(snapOperations) - 1 do
5:         for j ← 0 to length(snapOperations) - 2 do
6:             curr ← snapOperations[j]
7:             next ← snapOperations[j + 1]
8:             if CheckLinDefOrder(next, curr) then
9:                 snapOperations.SWAP(j, j + 1)
10:            end if
11:        end for
12:    end for
13:    sortedSnapOperations ← snapOperations
   %% Find the first snap that is linearized after modifyOp.
14:    for i ← 0 to length(sortedSnapOperations) - 1 do
15:        snapOp ← sortedSnapOperations[i]
16:        if CheckLinDefOrder(modifyOp, snapOp) then
17:            return snapOp
18:        end if
19:    end for
20: end procedure
```

6.3 Total Ordering

In this section, we verify that the algorithm described in Sections 6.1 and 6.2 produces a total ordering, i.e., that the operations can be mapped onto distinct points on a line. This does not yet prove that it produces a *correct* linearization, i.e., that "cause and effect" is preserved, and the proof of correctness is provided in Section 7. To show this, we focus on any three arbitrary operations a, b and c and prove that if a is linearized before b by CheckLinDefOrder and b is linearized before c by CheckLinDefOrder, then a is linearized before c by CheckLinDefOrder. We consider the following cases:

1. When a, b and c are all snap operations.
2. When a, b and c are modify operations within the same bucket.
3. When a and c are modify operations and b is snap operation.

In the following subsections, we show that transitivity holds in each of these three cases.

6.3.1 a, b and c are all snap operations

Suppose that CheckLinDefOrder(a,b) and CheckLinDefOrder(b,c) both return true. We consider all possible ways this could occur, and in all cases, we prove that CheckLinDefOrder(a,c) will also return true.

Case 1: CheckLinDefOrder(a,b) is returned from line 3 and CheckLinDefOrder(b,c) is returned from line 3: CheckLinDefOrder(a,c) will also return from 3 because of transitivity.

Case 2: CheckLinDefOrder(a,b) is returned from line 3 and CheckLinDefOrder(b,c) is returned from line 7: b and c end up at the same leaf node in the Classifier Tree. Therefore, from Lemma 9.28, their views are equal. However, a and b follow partial ordering, therefore from Lemma 9.18, a is to the left of b in the Classifier Tree leaf level. Since the views of b and c are equal and they end up

at the same leaf, a must be to the left of c. Therefore, CheckLinDefOrder(a,c) returns true from line 9.

Case 3: CheckLinDefOrder(a,b) is returned from line 3 and CheckLinDefOrder(b,c) is returned from line 9: b is to the left of c in the Classifier Tree leaf level. Since they are partially ordered, a is to the left of b in the Classifier Tree (from Lemma 9.18). Therefore, a is to the left of c in the Classifier Tree leaf level. Therefore, CheckLinDefOrder(a,c) returns true from line 9.

Case 4: CheckLinDefOrder(a,b) is returned from line 7 and CheckLinDefOrder(b,c) is returned from line 3: a and b end up at the same leaf. Therefore, from Lemma 9.28, their views are equal. Since b and c follow partial ordering, b must be to the left of c (from Lemma 9.18). Since a and b end up at the same leaf, a must be to left of c in the Classifier Tree leaf level. Therefore, CheckLinDefOrder(a,c) returns true from line 9.

Case 5: CheckLinDefOrder(a,b) is returned from line 7 and CheckLinDefOrder(b,c) is returned from line 7: CheckLinDefOrder(a,c) will also return from 7 because of transitivity.

Case 6: CheckLinDefOrder(a,b) is returned from line 7 and CheckLinDefOrder(b,c) is returned from line 9: a and b end up at the same leaf. Therefore, from Lemma 9.28, their views are equal. Operations b and c end up in different leaves and b is to the left of c in the leaves of the Classifier Tree. Since, a and b are at the same leaf, it must be the case that a is to the left of c. Therefore, CheckLinDefOrder(a,c) returns true from line 9.

Case 7: CheckLinDefOrder(a,b) is returned from line 9 and CheckLinDefOrder(b,c) is returned from line 3: a is to the left of b in the Classifier Tree leaf level. Operation b is to the left of c in the Classifier Tree leaf level because b is partially ordered before c. Therefore, a is to the left of c in the Classifier Tree and CheckLinDefOrder(a,c) returns true from line 9.

Case 8: CheckLinDefOrder(a,b) is returned from line 9 and CheckLinDefOrder(b,c) is returned from line 7: a is to the left of b in the Classifier Tree leaf level. Operation b and c are at the same leaf in the Classifier Tree. Therefore, it must be the case that a is to the left of c in the Classifier Tree leaf level and CheckLinDefOrder(a,c) returns true from line 9.

Case 9: CheckLinDefOrder(a,b) is returned from line 9 and CheckLinDefOrder(b,c) is returned from line 9: CheckLinDefOrder(a,c) will also return true from line 9 because of transitivity.

6.3.2 a, b and c are modify operations within the same bucket

We say that two modify operations are within the same bucket if they are linearized between the same two snap operations. Since the three operations are within the same bucket, the *rightSnap* variable at lines 20 and 21 will always indicate the same snap operations. Therefore, the condition at line 22 will always evaluate to true. Inside this if block, the return value is decided based on the lexicographical ordering using the pair ($ModOrd$, $ProcId$) which are integer values. Therefore, all comparisons will obey transitivity.

6.3.3 a and c are modify operations and b is snap operation

Suppose that CheckLinDefOrder(a,b) and CheckLinDefOrder(b,c) both return true. We prove that CheckLinDefOrder(a,c) will also return true.

Let the values in $rightSnap_a$ and $rightSnap_c$ variables be RS_a and RS_c. We claim that RS_a is linearized before or the same as the snap operation b and b is linearized before RS_c. Before we prove this claim, we need to understand how the FindRightSnapOperation algorithm works:

1. RS_a and RS_c are the values returned from the FindRightSnapOperation algorithm when respectively a and c are passed as the parameter modifyOp.

2. In the FindRightSnapOperation algorithm, at line 2, the allOperations variable is the list of all the operations that ever took place.

3. At line 3, only the snap operations are kept from the list of allOperations.

4. From line 4 to line 12, a bubble sort algorithm is run on the list of snap operations.

5. At line 13, the sortedSnapOperations variable is the list of linearized snap operations in ascending order.

6. In the loop starting at line 14, each snap operation is traversed one by one in the linearized order.

Now back to proving the two inequality claims:

1. RS_a is linearized before or the same snap operation as b:

   (a) Recall that we are assuming that a is linearized before b by CheckLinDefOrder. Therefore, during FindRightSnapOperation(a), line 16 will evaluate to true when snapOp = b.

   (b) Therefore, the return value at line 17 will be a snap operation that is either b or a snap operation that is linearized before b, as desired.

2. RS_c is linearized after b:

   (a) Let b_i be the index of b within the sortedSnapOperations list.

   (b) Recall that we are assuming that c is linearized after b. Therefore, during FindRightSnapOperation(c), line 16 cannot be true for any snap operation with index less than or equal to b_i.

   (c) We know RS_c exists. Therefore, during FindRightSnapOperation(c) line 16 was true for a snap operation RS_c with index greater than b_i. Since the index of RS_c is greater than b_i, RS_c in linearized after b, as desired.

We showed that RS_a is linearized before or the same as the snap operation b and b is linearized before RS_c. This means that RS_a and RS_c are not the same snap operations. Therefore, in CheckLinDefOrder, line 22 will evaluate to false and the code will reach line 29. At line 29, the CheckLinDefOrder algorithm will be called

again with the parameters as RS_a and RS_c. Since we already know that RS_a is strictly linearized before RS_c, the CheckLinDefOrder(RS_a, RS_c) will return true. Therefore, a will be linearized before c which concludes the proof that transitivity holds.

7 Proof of correctness

In this section, we show that by linearizing the operations as described in Section 6, the output of our snap implementation is consistent with what the user(s) of the system expect (regardless of any adversarial scheduler). In particular, there are two properties that must hold for a snap output to be considered valid:

1. For each process p_i, the block with address p_i in the snap output has the same value as what p_i most recently wrote to its shared register, and,
2. The blocks contained in the output correspond exactly to the processes that have previously been added to the system but have not been removed yet.

7.1 Terminology

Generally, in a shared memory system where concurrent operations are possible, the words *before* and *after* don't have a precise meaning. When we use the words *before* or *after* to compare two operations we strictly mean to say *linearized before* or *linearized after*, as defined by our linearization in Section 6. For example, if operation op_1 occurs *before* operation op_2, it means op_1 is linearized before op_2.

7.2 Validity of Snap Operations

The snap operations are the only type of operation that returns a result to the user. Therefore, we only need to focus on the validity of the result of the snap operations. Before we start with the proof of correctness, it might be better to understand the motivation behind the approach.

First, consider a simpler type of snapshot object whose definition does not allow operations to be executed asynchronously or in parallel. In this environment, we can

say the result of the snap operation is valid if the result contains the most recent modifications made by add, remove or update operations that were executed (and therefore terminated) prior to the start of snap operation. More formally:

1. The result of the snap operation should contain the values written by the most recent update operations.

2. The result of the snap operation should contain the processes that were added but not removed across all previous add and/or remove operations.

Now back to our snapshot object where the operations do not have the above-mentioned restriction, i.e., now assume that operations can overlap in time. The primary problem now is that there is no strict ordering can be enforced because of the wait-free nature of the operations, i.e., our algorithm cannot have operations wait until others have completed. However, in Section 6, we described a method for linearizing overlapping operations, i.e., introducing a strict ordering by mapping the operations to points on a line. Using this linearization we can define the validity of the result of the snap operations as follows:

1. The result of the snap operation should contain the values written by the most recent update operation where the *most recent* update operation is defined with respect to the linearization ordering.

2. The result of the snap operation should contain exactly the processes that were in the initial setup of the snapshot object and that were *previously* added but not removed. Here, the word previously is again, with respect to the linearization ordering.

In Sections 7.2.1 and 7.2.2, we separately prove each of these validity conditions about our snapshot implementation with respect to our provided linearization.

7.2.1 Snap Validity 1: Contains the values written by the most recent update operation

Recall that, during every operation (i.e., snap, update, add, remove), a process does a scan of each register in shared memory S. In the next lemma, we give conditions

under which we can guarantee that two processes performing this scan will read the same information from a particular register of shared memory S. The key insight here is that the sequence numbers uniquely identify a written block, i.e., if the sequence numbers match, the blocks are the same. This lemma will be used to consider a block in a snap operation's output and connect it back to the most recent write of that block to shared memory (in the case where the sequence numbers match).

Lemma 7.1. *Let op_j and op_k be any two operations performed by processes p_j and p_k. Let p_i be any process (where it can be p_j or p_k or any other process). If the following conditions are met:*

1. $BL_{p_i,S,op_j}.seq == BL_{p_i,S,op_k}.seq$ *where S is the shared memory S*
2. $p_i \in addr_0$ *during* op_j
3. $p_i \in addr_0$ *during* op_k

We prove that $BL_{p_i,S,op_j} == BL_{p_i,S,op_k}$.

Proof.

1. Because of SWMR property, only p_i can write to BL_{p_i} of the shared memory S.
2. No matter which operation p_i performs, after reading the BL_{p_i} block from the shared memory S, when p_i writes it back to the shared memory S, the seq value of that block is incremented. Therefore, each time a new block is written at address p_i of shared memory S, the seq value is updated.
3. It follows that, while reading the BL_{p_i} of shared memory S at line *SREAD2*, if p_j and p_k see the same seq value then, p_j and p_k are seeing the same block at address p_i of the shared memory S.

□

In the next lemma, each block in the bag that is returned from the final call to the Classifier during an operation is being connected back to a block in the shared memory S that was written during *some* previous operation. This connection essentially

establishes that each block that we see in the bag returned by snap operation was actually initially written to a register in the shared memory S and, eventually, that block was propagated down some path in the Classifier Tree to a leaf from which the snap result was taken. As with Lemma 7.1, this lemma will be used in the proof of snap validity to connect each block in a bag back to the most recent write operation to a shared register.

Lemma 7.2. *Let p_i and p_j be any two processes. Let op_j be any operation executed by p_j. Let op_i be any operation executed by p_i. Let op_i be linearized before op_j. Let ss_{final} be the bag returned by the final call to Classifier during op_j. We prove that there exists a process p_k for which $BL_{p_i,ss_{final},op_j} == BL_{p_i,S,op_k}$, and, during op_k, $p_i \in addr_0$.*

Proof. Let P(ℓ, p_j) be "if during op_j, $p_i \in addr_{\ell-1}$ then, at line *PREC* at level ℓ, there exists a process p_k for which $BL_{p_i,ss_{\ell-1},op_j} == BL_{p_i,S,op_k}$ and during op_k, $p_i \in addr_0$". We use a proof by induction on the level $\ell \geq 1$ and process p_j.

Base Case:

1. Consider $\ell = 1$. So, we are considering the block BL_{p_i,ss_0,op_j} at line *PREC*.
2. To reach line *PREC*, previous lines must be executed by p_j.
3. Since from induction assumption, $p_i \in addr_0$, BL_{p_i,ss_0,op_j} exists. This BL_{p_i,ss_0,op_j} is created after reading it from the shared memory S at line *SREAD2*. At line *SREAD2*, the block is found at address p_i.
4. Because of SWMR property, only process p_i can write to the block at address p_i in the shared memory S. Therefore, p_i is process for which $BL_{p_i,ss_0,op_j} == BL_{p_i,S,op_i}$ and during op_i, $p_i \in addr_0$. (base case proved)

Induction Hypothesis: Assume of an $\ell > 1$, $P(1, p_j) \wedge ... \wedge P(\ell - 1, p_j)$ is true.

Inductive Step:

1. Consider any arbitrary $\ell > 0$ and a process p_j. So, we are considering the block $BL_{p_i,ss_{\ell-1},op_j}$ at line *PREC* before the call to the Classifier.

2. Suppose that, $p_i \in addr_{\ell-1}$ during op_j.

3. From Lemma 9.4 (Part: 1), we know, for block p_i in $ss_{\ell-1}$, there exists a process p_k where $BL_{p_i,ss_{\ell-1},op_j} == BL_{p_i,ss_{\ell-2},op_k}$ and $p_i \in addr_{\ell-2}$ during op_k.

4. We apply the induction hypothesis to p_k at level $\ell-1$ and, it holds for $P(\ell-1, p_k)$.

5. Since $P(\ell-1, p_k)$ holds, we know there exists a process $p_{k'}$ for which $BL_{p_i,ss_{\ell-2},op_k}$ $== BL_{p_i,S,op_{k'}}$ and during $op_{k'}$, $p_i \in addr_0$.

6. From step 3 and step 5, we know:

   (a) $BL_{p_i,ss_{\ell-1},op_j} == BL_{p_i,ss_{\ell-2},op_k}$ and $p_i \in addr_{\ell-2}$ during op_k

   (b) $BL_{p_i,ss_{\ell-2},op_k} == BL_{p_i,S,op_{k'}}$ and during $op_{k'}$, $p_i \in addr_0$

7. Therefore, $BL_{p_i,ss_{\ell-1},op_j} == BL_{p_i,S,op_{k'}}$ and during $op_{k'}$, $p_i \in addr_0$.(induction step proved)

Concluding Statement:

1. Consider the end condition of the loop at line *PREC*. ss_{final} is the result at the final level $\ell = \lceil \log(m) \rceil$.

2. Since op_i is linearized before op_j, from Lemma 9.11, $p_i \in addr_{ss_{final}}$ during op_j.

3. From the induction statement, P($\lceil \log(m) \rceil$ + 1, p_j) be "there exists a process p_k for which $BL_{p_i,ss_{final},op_j} == BL_{p_i,S,op_k}$ and during op_k, $p_i \in addr_0$".

□

In the next lemma, we consider a block with address p_i in a snap operation's output bag, and compare this block to the most recent block written by p_i to its shared register during an update operation that terminated before the snap began

(i.e., there is no overlap between the update by p_i and the snap). We establish a relationship between the seq values in these blocks, i.e., that the seq value in the snap output is not smaller than the seq value that was written to shared memory. This connection between the seq values will help us prove that the snap operation outputs the most up-to-date version of the block.

Lemma 7.3. *Let p_i and p_s be any two processes. Let op_s be a snap operation executed by p_s. Let op_i be an update operation executed by p_i where op_i is the most recent update operation in the linearization order and $op_i \to op_s$. Let ss_{final} denote the bag returned by the final call to Classifier during an operation. We prove that $BL_{p_i,ss_{final},op_i}.seq \leq BL_{p_i,ss_{final},op_s}.seq$.*

Proof. From Lemma 9.9, we know, $BL_{p_i,ss_{final},op_i} == BL_{p_i,S,op_i}$. So, BL_{p_i,ss_{final},op_i}.seq $== BL_{p_i,S,op_i}$.seq. Let BL_{p_i,S,op_i} during op_i be seq_{p_i}.

Let P(ℓ) be "For any level ℓ, $seq_{p_i} \leq BL_{p_i,ss_\ell,op_s}$.seq and $p_i \in addr_{ss_\ell}$ during op_s at line *POSTC* of the snap algorithm".
We use a proof by induction on the level $\ell \geq$ a where a is the level where the LCA(op_i, op_s) is located in the Classifier Tree. This LCA node exists because:

1. Since op_i and op_s follow partial ordering, there will be at least one block where the seq value will be different in the final bag returned from the final calls to the Classifier. Therefore, from the contrapositive of Lemma 9.28, we know op_i and op_s will end up at different leaves.

2. Since the Classifier graph is a tree structure, the LCA node must exist.

Base Case: ($\ell = a$)

1. From Lemma 9.17, we know that during the execution of the Classifier algorithm at LCA(op_i, op_s) node, op_s goes right and op_i goes left.

2. Since p_i is executing op_i, $p_i \in updatedActiveList_{op_i}$. From 9.16 we know, $updatedActiveList_{op_i} \subseteq updatedActiveList_{op_s}$. Therefore, $p_i \in updatedActiveList_{op_s}$.

3. Since op_s is going right, $updatedActiveList_{op_s}$ will be returned and received in $addr_{ss_a}$ variable. So, $p_i \in addr_{ss_a}$ and therefore BL_{p_i} will exist in newMergeRecent bag at line 8. newMergeRecent is returned and received in ss_a variable outside of the Classifier algorithm therefore, BL_{p_i,ss_a} exists.

4. Since p_i exists is in both $updatedActiveList_{op_i}$ and $updatedActiveList_{op_s}$ and $op_i \rightarrow op_s$, BL_{p_i}.seq in newMergeRecent bag at line 8 will be at least as big as the seq value that was written to shared memory S during op_i.

5. Therefore, during op_s at line *POSTC*, $seq_{p_i} \leq BL_{p_i,ss_a,op_s}$.seq. (base case proved)

Induction Hypothesis: Assume of an $\ell > a$, $P(a) \wedge ... \wedge P(\ell - 1)$ is true.

Inductive Step:

1. Consider any arbitrary level $\ell > a$ at line *PREC* of the snap algorithm during op_s.

2. The input to the Classifier algorithm at level ℓ is $ss_{\ell-1}$ and the output of the Classifier algorithm is ss_ℓ

3. From assumption, we know, $P(\ell-1)$ is true, which means, $p_i \in addr_{ss_{\ell-1}}$ during op_s.

4. From Lemma 9.15, we know at line *POSTC*, $BL_{p_i,ss_{\ell-1},op_s}$.seq $\leq BL_{p_i,ss_\ell,op_s}$.seq.

5. From assumption, we know, $P(\ell - 1)$ is true, which means,
$seq_{p_i} \leq BL_{p_i,ss_{\ell-1},op_s}$.seq

6. Therefore, $seq_{p_i} \leq BL_{p_i,ss_{\ell-1},op_s}$.seq $\leq BL_{p_i,ss_\ell,op_s}$.seq (induction step proved)

Concluding Statement:

1. In the induction proof, we proved that for any level ℓ, $seq_{p_i} \leq BL_{p_i,ss_\ell,op_s}$.seq during op_s at line *POSTC*.

2. When $\ell = \log(m) + 1$, $ss_{final} == ss_{\log(m)+1}$.

3. Therefore, $seq_{p_i} \leq BL_{p_i,ss_{final},op_s}$.seq is true.

4. Since, $seq_{p_i} == BL_{p_i,S,op_i}$.seq $== BL_{p_i,ss_{final},op_i}$.seq, BL_{p_i,ss_{final},op_i}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq.

□

Finally, we are able to prove the main claim of this section, i.e., that the first condition of snap validity holds: each block p_i in the output bag of the snap operation contains the value written by the most recent update operation by p_i, with respect to the linearization ordering.

Lemma 7.4. *Let op_s be a snap operation executed by p_s. Let ss_{final} denote the bag returned by the final call to Classifier during an operation. Let op_i be the most recent update operation (executed by process p_i) that updated the $BL_{p_i,S}$ in shared memory S. Let $\beta\ell_{p_i}$ be the block written (at line SWRITE of the update operation) by p_i in shared memory S.*

We prove that $BL_{p_i,ss_{final},op_s}.val == \beta\ell_{p_i}.val$.

Proof. Since op_i is linearized before op_s, either line 14 or line 3 in the Linearization algorithm (Algorithm: 10) must be true.

1. If line 3 of the Linearization algorithm (Algorithm: 10) is true, we know $op_i \rightarrow op_s$. We know, if $op_i \rightarrow op_s$ is true, then from Lemma 7.3, BL_{p_i,ss_{final},op_i}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq.

2. Also, if line 14 of the Linearization algorithm (Algorithm: 10) is true, we know from the Linearization algorithm (Algorithm: 10) line 14 that BL_{p_i,ss_{final},op_i}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq.

3. So, we can proceed under the assumption that: BL_{p_i,ss_{final},op_i}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq

Case 1: BL_{p_i,ss_{final},op_i}.seq $< BL_{p_i,ss_{final},op_s}$.seq:

(a) From Lemma 9.9, we know, during op_i at any level ℓ, when p_i executes line *POSTC* of the Update algorithm, $BL_{p_i,ss_\ell,op_i} == \beta\ell_{p_i}$. So, at the end of the final call to Classifier when $\ell = \log(m) + 1$, $BL_{p_i,ss_{\log(m)+1},op_i} == \beta\ell_{p_i}$. Therefore, $BL_{p_i,ss_{final},op_i} == \beta\ell_{p_i}$.

(b) Since, $BL_{p_i,ss_{final},op_i} == \beta\ell_{p_i}$ and the case assumption is BL_{p_i,ss_{final},op_i}.seq $< BL_{p_i,ss_{final},op_s}$.seq, we can say, $\beta\ell_{p_i}$.seq $< BL_{p_i,ss_{final},op_s}$.seq.

(c) In the following steps we will use a term called *intermediate operation*. We define an operation op_j an intermediate operation if it is performed by p_i after op_i during which the line *SWRITE* took place, before the line where BL_{p_i,ss_{final},op_s} was calculated during op_s, and, $\beta\ell_{p_i}$.seq $<$ BL_{p_i,S,op_j}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq. Note that it is also possible for the op_s to satisfy the conditions for it to be an intermediate operation, but it is not possible for op_i. We now argue that there exists at least one such intermediate operation and, also, for at least one of the intermediate operations op_j, BL_{p_i,S,op_j}.seq $==$ BL_{p_i,ss_{final},op_s}.seq. We can say this because of the following facts:

    i. The seq value never decreases anywhere in the algorithms

    ii. Because of SWMR property, only p_i can write in a block with address p_i.

(d) Let op_j be an arbitrary intermediate operation. Since BL_{p_i,S,op_j}.seq $> \beta\ell_{p_i}$.seq and SWMR property, op_i is partially ordered before op_j. Therefore, from the Linearization algorithm (10) line 3, op_i is linearized before op_j.

(e) We argue that if op_j is a modify operation (add, update or remove operation) then op_j is linearized before op_s:

    i. From the assumption about op_j in Step: 1c, we know that BL_{p_i,S,op_j}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq.

    ii. Since p_i and p_j are the same processes, BL_{p_i,S,op_j}.seq can be written as BL_{p_j,S,op_j}.seq.

iii. From Lemma 9.9, we know, during op_j at any level ℓ, when p_j executes line *POSTC* of the algorithm, $BL_{p_j,ss_\ell,op_j} == BL_{p_j,S,op_j}$. So, at the end of the final call to Classifier when $\ell = \log(m) + 1$, $BL_{p_j,ss_{\log(m)+1},op_j} == BL_{p_j,S,op_j}$. Therefore, $BL_{p_j,ss_{final},op_j} == BL_{p_j,S,op_j}$.

iv. From step 1(e)i we know BL_{p_i,S,op_j}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq and from step 1(e)iii we know $BL_{p_j,ss_{final},op_j} == BL_{p_j,S,op_j}$. Therefore, BL_{p_j,ss_{final},op_j}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq.

v. From the Linearization algorithm (10) line 14, op_j is linearized before op_s.

(f) We argue that op_j cannot be an update operation. This is because from step 1d we know op_i is linearized before op_j and from step 1e we know op_j is linearized before op_s. However, this means that the assumption in the lemma statement where it states op_i is the most recent update operation linearized before op_s no longer holds. This is why op_j cannot be an update operation.

(g) Now that we have shown that every intermediate operation is not an update operation, in the following steps we will show that the val in block BL_{p_i,ss_{final},op_s} comes from $\beta\ell_{p_i}$.val. The val property of the block BL_{p_i} in the shared memory S is only updated during the update operation at line *SWRITE*. Since op_j is not an update operation, the val property of the block BL_{p_i} remain unchanged during op_j as well. Therefore, during op_j, BL_{p_i,S,op_i}.val $== BL_{p_i,S,op_j}$.val.

(h) In the following steps we consider another arbitrary intermediate operation op_j where BL_{p_i,S,op_j}.seq $== BL_{p_i,ss_{final},op_s}$.seq. (Recall that we have shown in step 1c that at least one such operation exists)

(i) From Lemma 7.2, we know there exists an op_k for which $BL_{p_i,ss_{final},op_s} == BL_{p_i,S,op_k}$ and, $p_i \in addr_0$ during op_k. Using this equality and BL_{p_i,ss_{final},op_s}.seq $== BL_{p_i,S,op_j}$.seq (from step 1h), BL_{p_i,S,op_k}.seq $== BL_{p_i,S,op_j}$.seq.

(j) We know the following:

i. The seq values are same at address p_i in the shared memory S during op_k and op_j. (from the previous step)

ii. $p_i \in addr_0$ during op_k (from the previous step)

iii. $p_i \in addr_0$ during op_j (from Lemma 9.5 since op_j is performed by p_i)

(k) Therefore, from Lemma 7.1, for op_j and op_k, $BL_{p_i,S,op_k} == BL_{p_i,S,op_j}$.

(l) From the previous step we know, $BL_{p_i,S,op_k} == BL_{p_i,S,op_j}$ and from step 1i we know, $BL_{p_i,ss_{final},op_s} == BL_{p_i,S,op_k}$, therefore, $BL_{p_i,ss_{final},op_s} == BL_{p_i,S,op_j}$. Therefore, BL_{p_i,ss_{final},op_s}.val $==$ BL_{p_i,S,op_j}.val

(m) From step 1l we know, BL_{p_i,ss_{final},op_s}.val $==$ BL_{p_i,S,op_j}.val. From step 1g we know, BL_{p_i,S,op_i}.val $==$ BL_{p_i,S,op_j}.val. Therefore, BL_{p_i,ss_{final},op_s}.val $==$ BL_{p_i,S,op_i}.val. In other words, BL_{p_i,ss_{final},op_s}.val $==$ $\beta\ell_{p_i}$.val (Case 1 proved)

Case 2: BL_{p_i,ss_{final},op_i}.seq $==$ BL_{p_i,ss_{final},op_s}.seq:

(a) From Lemma 7.2 $BL_{p_i,ss_{final},op_s} == BL_{p_i,S,op_k}$ and, $p_i \in addr_0$ during op_k.

(b) From Lemma 9.9, for level $\ell = \log(m) + 1$, $BL_{p_i,ss_{\log(m)+1},op_i} == \beta\ell_{p_i}$. Since, $ss_{\log(m)+1} == ss_{final}$, $BL_{p_i,ss_{final},op_i} == \beta\ell_{p_i}$

(c) We know: BL_{p_i,ss_{final},op_i}.seq $==$ BL_{p_i,ss_{final},op_s}.seq (from Case 2 assumption)

(d) Changing sides of this equation we get that:
BL_{p_i,ss_{final},op_s}.seq $==$ BL_{p_i,ss_{final},op_i}.seq

(e) From step (a) using $BL_{p_i,ss_{final},op_s} == BL_{p_i,S,op_k}$, it follows that BL_{p_i,S,op_k}.seq $==$ BL_{p_i,ss_{final},op_i}.seq

(f) From step (b) using $BL_{p_i,ss_{final},op_i} == \beta\ell_{p_i}$, it follows that BL_{p_i,S,op_k}.seq $==$ $\beta\ell_{p_i}$.seq ($==$ BL_{p_i,S,op_i} from Lemma statement of 9.9)

(g) We know the following:

i. The seq values are same at address p_i in the shared memory S during op_k and op_i.

ii. $p_i \in addr_0$ during op_k (from step (a) of this case)

iii. $p_i \in addr_0$ during op_i (from Lemma 9.5)

(h) Therefore, for op_i and op_k, from Lemma 7.1, $BL_{p_i,S,op_k} == BL_{p_i,S,op_i}$. We know from the lemma statement, $\beta\ell_{p_i} == BL_{p_i,S,op_i}$. So, $BL_{p_i,S,op_k} == \beta\ell_{p_i}$.

(i) Therefore, BL_{p_i,S,op_k}.val $== \beta\ell_{p_i}$.val

(j) From step (a), $BL_{p_i,S,op_k} == BL_{p_i,ss_{final},op_s}$ and so it follows that BL_{p_i,ss_{final},op_s}.val $== \beta\ell_{p_i}$.val (Case 2 proved)

□

7.2.2 Snap Validity 2: Contains the added processes that have not been removed

In the first two lemmas, we establish that the add and remove operations are linearized in the correct order relative to snap operations. In particular, if a snap result indicates that a certain process p is present or absent, then a corresponding add(p) or remove(p), respectively, occurs before that snap in the linearization order.

In the first lemma, we are proving that if a process p was not present during the initialization of the system but is found in the address list that is returned by the final call to Classifier during some snap operation op, then, then there must exist an add(p) operation before op.

Lemma 7.5. *Let op_s be a snap operation executed by p_s. Let p_j be an arbitrary process where $p_j \notin procList_\alpha$ and $p_j \in addr_{final}$ during op_s. We prove the following:*

1. *There exists an add operation op_i executed by p_i where p_j is added.*

2. *$p_i \in addr_{final}$ and op_i is linearized before op_s.*

Proof. From the Observation 5.3, there exists an add operation op_r executed by p_r. Since there can be only one such add operation where p_j is added, $p_i == p_r$ and $op_i == op_r$. (Part 1 of the statement is proved). We prove the part 2 of the statement in the following steps.

Case 1: Consider that $p_j \in addr_0$ during op_s at line *PREC*

1. At line *ADDR0INIT*, $addr_0$ is initialized with the list $BL_{p_s,S}$.active.
2. Since $p_j \in addr_0$ and $addr_0 == BL_{p_s,S}$.active, $p_j \in BL_{p_s,S}$.active
3. However in shared memory S, p_j only exists in $BL_{p_i,S}$.active (from Observation 5.3). Therefore, it must be the case that $p_i == p_s$.
4. Therefore, op_i and op_s are both executed by p_s. Because of SWMR property, op_i and op_s follow partial ordering.
5. From Observation 5.3, p_j first appears during op_i. Therefore, the execution of op_i was before op_s.
6. Since op_i and op_s follow partial ordering, op_i is partially ordered before op_s.
7. From the Linearization Definition Algorithm (Algorithm: 10) at line 3, this proves that op_i is linearized before op_s. (Case 1 proved)

Case 2: Consider that $p_j \notin addr_0$ during op_s at line *PREC*. Therefore, there exists a level a (where $1 \leq a \leq \log(m) + 1$), $p_j \notin addr_{a-1}$ at line *PREC* but $p_j \in addr_a$ at line *POSTC* during op_s.

1. During the call to the Classifier at level a, $p_j \notin addr_{a-1}$ and $p_j \in addr_a$. The only way $addr_{a-1}$!= $addr_a$ is possible if line 6 of the Classifier algorithm evaluates to true.
2. $addr_a$ is the list that is returned from the GetUpdatedActiveList. Since $p_j \in addr_a$, updatedActiveList variable in the GetUpdatedActiveList algorithm also contains p_j at the end of the GetUpdatedActiveList algorithm.

3. Since $p_j \notin addr_{a-1}$ and $addr_{a-1}$ is passed as oldActive and is assigned to updatedActiveList variable in the GetUpdatedActiveList algorithm at line 2, $p_j \notin$ updatedActiveList at 2 of the GetUpdatedActiveList algorithm.

4. Therefore, during the execution of GetUpdatedActiveList, p_j is found and added to the updatedActiveList list variable at line 17.

5. p_j is found in the active list of variable block r_k at line 14. This block is created at line 12 and the address k is found in the updatedActiveList at line 11.

6. We know p_j can only exist in the active list of BL_{p_i}. Therefore, p_i already exists in updatedActiveList when p_j is found.

7. Since no items are removed from the updatedActiveList and the value returned from the GetUpdatedActiveList is assigned to $addr_a$, $p_i \in addr_a$. From Lemma 9.2, $p_i \in addr_{final}$ (as required in the 2nd part of the lemma statement).

8. When p_j is found during GetUpdatedActiveList in BL_{p_i}.active at line 14, from Observation 5.3, BL_{p_i,ss_a,op_s}.seq $\geq BL_{p_i,S,op_i}$.seq.

9. From Lemma 9.9, $BL_{p_i,S,op_i} == BL_{p_i,ss_{final},op_i}$.

10. So, BL_{p_i,ss_a,op_s}.seq $\geq BL_{p_i,ss_{final},op_i}$.seq.

11. If level a is the final level then, BL_{p_i,ss_{final},op_s}.seq $\geq BL_{p_i,ss_{final},op_i}$.seq , and we are done.

12. So, in the following steps, we consider the case when level a is not the final level.

13. Consider the level $a+1$.

14. Since $p_i \in addr_a$, from Lemma 9.15, we know BL_{p_i,ss_a,op_s}.seq $\leq BL_{p_i,ss_{a+1},op_s}$.seq.

15. Also from Lemma 9.2, $p_i \in addr_{a+1}$. This argument can be extended down to the level $\log(m)+1$.

16. Therefore, BL_{p_i,ss_a,op_s}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq

17. We know:

    (a) From step (10) BL_{p_i,ss_a,op_s}.seq $\geq BL_{p_i,ss_{final},op_i}$.seq

    (b) From the previous step, BL_{p_i,ss_a,op_s}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq

    Therefore, BL_{p_i,ss_{final},op_s}.seq $\geq BL_{p_i,ss_a,op_s}$.seq $\geq BL_{p_i,ss_{final},op_i}$.seq.

18. Since BL_{p_i,ss_{final},op_s}.seq $\geq BL_{p_i,ss_{final},op_i}$.seq, from the Linearization Definition Algorithm (Algorithm: 10) at line 14, this proves that op_i is linearized before op_s.

□

In the next lemma, we prove that after the final call to Classifier during a snap operation *op*, if a process *p* is found in the removeActive list of any of the blocks in the final bag returned by Classifier, then there must exist a remove(p) operation before *op*.

Lemma 7.6. *Let op_s be a snap operation executed by p_s. Let p_j be an arbitrary process where $p_j \in \bigcup_{p_x \in addr_{final}} (BL_{p_x,ss_{final},op_s}.removeActive)$. We prove the following:*

1. *There exists a remove operation op_i executed by p_i where p_j is removed*

2. *op_i is linearized before op_s.*
   More specifically: $BL_{p_i,ss_{final},op_i}.seq \leq BL_{p_i,ss_{final},op_s}.seq$

Proof. Let $p_r \in addr_{final}$ be any arbitrary process such that $p_j \in BL_{p_r,ss_{final},op_s}$.removeActive. From Observation 5.2, there exists a remove operation op_r executed by p_r. Since there can be only one such remove operation where p_j is removed, $p_i == p_r$ and $op_i == op_r$. (Part 1 of the lemma statement is proved). We prove the part 2 of the lemma statement in the following steps.

1. We consider that $p_j \notin BL_{p_i,ss_{a-1},op_s}$.removeActive and $p_j \in BL_{p_i,ss_a,op_s}$.removeActive where $a \geq 0$. When $a = 0$, there exists no ss_{a-1} bag which means there exists no BL_{p_i,ss_{a-1},op_s} block and therefore, $p_j \notin BL_{p_i,ss_{a-1},op_s}$.removeActive holds.

2. From Observation 5.2, we know that BL_{p_i,ss_a,op_s}.seq $\geq BL_{p_i,S,op_i}$.seq
3. From Lemma 9.9, $BL_{p_i,S,op_i} == BL_{p_i,ss_{final},op_i}$.
4. So, BL_{p_i,ss_a,op_s}.seq $\geq BL_{p_i,ss_{final},op_i}$.seq.
5. If level a is the final level then, BL_{p_i,ss_{final},op_s}.seq $\geq BL_{p_i,ss_{final},op_i}$.seq , and we are done.
6. So, in the following steps, we consider the case when level a is not the final level.
7. Consider the level $a+1$.
8. Since $p_i \in addr_a$ (from Observation: 5.2), from Lemma 9.15, we know BL_{p_i,ss_a,op_s}.seq $\leq BL_{p_i,ss_{a+1},op_s}$.seq.
9. Also from Lemma 9.2, $p_i \in addr_{a+1}$. This argument can be extended down to the level $\log(m)+1$.
10. Therefore, BL_{p_i,ss_a,op_s}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq
11. We know:
    (a) From step (4) BL_{p_i,ss_a,op_s}.seq $\geq BL_{p_i,ss_{final},op_i}$.seq
    (b) From the previous step, BL_{p_i,ss_a,op_s}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq

    Therefore, BL_{p_i,ss_{final},op_s}.seq $\geq BL_{p_i,ss_a,op_s}$.seq $\geq BL_{p_i,ss_{final},op_i}$.seq.
12. Since BL_{p_i,ss_{final},op_s}.seq $\geq BL_{p_i,ss_{final},op_i}$.seq, from the Linearization Definition Algorithm (Algorithm: 10) at line 14, this proves that op_i is linearized before op_s.

□

In the next lemma, we consider what happens when two operations read different blocks from the same shared register: the more recently written version of the block

(as determined by sequence number seq) contains a list of active processes that is a superset of the list of active processes in the older version of the block. In other words, no information is lost when constructing a newer version of a block.

Lemma 7.7. *Let op_j and op_k be any two operations performed by processes p_j and p_k. Let p_i be any process. If the following conditions are met:*

1. $BL_{p_i,S,op_j}.seq < BL_{p_i,S,op_k}.seq$
2. $p_i \in addr(S)$ *during* op_j
3. $p_i \in addr(S)$ *during* op_k

then we prove that: $BL_{p_i,S,op_j}.active \subseteq BL_{p_i,S,op_k}.active.$

Proof.

1. Because of SWMR property, only p_i can write to BL_{p_i} of the shared memory S.

2. No matter which operation p_i performs, after reading the BL_{p_i} block from the shared memory S, when p_i writes it back to the shared memory S, the seq value of that block is incremented. Therefore, each time a new block is written at address p_i of shared memory S, the seq value is increased.

3. While reading the BL_{p_i} of shared memory S at line *SREAD2*, if p_j and p_k see different seq values, it must be the case that somewhere between op_j and op_k, p_i wrote to shared memory S during some operation.

4. More specifically, while reading from the shared memory S, if BL_{p_i,S,op_j}.seq < BL_{p_i,S,op_k}.seq, there must be a write operation by p_i after p_j reads BL_{p_i} and before p_k reads BL_{p_i} from shared memory S.

5. If op_i was an add operation, then when p_k reads BL_{p_i} from shared memory S, it will see an active list which will contain all the values in the active list that p_j read from shared memory S plus the newly added process during op_i. So, BL_{p_i,S,op_j}.active $\subset BL_{p_i,S,op_k}$.active.

6. Otherwise, if op_i was not an add operation then when p_k reads BL_{p_i} from shared memory S, it will see an active list which will contain all the values in the active list that p_j read from shared memory S. So, BL_{p_i,S,op_j}.active $==$ BL_{p_i,S,op_k}.active.

7. Therefore, BL_{p_i,S,op_j}.active $\subseteq$ BL_{p_i,S,op_k}.active.

□

In the next lemma, we show that Classifier is consistent when maintaining the list of active processes. More specifically, if one Classifier call returns a bag containing a block b and an active process list containing a process p, and a different Classifier call returns a bag containing the same block b, then the active process list returned by this call also contains p.

Lemma 7.8. *Let op_i and op_j be two operations executed by p_i and p_j respectively. We prove that if $BL_{p_i,ss_a,op_j} == BL_{p_i,ss_b,op_j}$ for any levels a,b greater than or equal to 0 and less than or equal to $\log(m) + 1$ and $p_i \in addr_a$ then, $p_i \in addr_b$.*

Proof. We use proof by contradiction. We assume that $p_i \notin addr_b$.

1. Since $p_i \notin addr_b$, BL_{p_i,ss_b,op_j} is null or empty.

2. However, since $p_i \in addr_a$, BL_{p_i,ss_a,op_j} is not null or empty.

3. This contradicts the condition where $BL_{p_i,ss_a,op_j} == BL_{p_i,ss_b,op_j}$.

□

In the next lemma, we consider a situation where a snap operation returns a bag that contains a block belonging to a "new" process, i.e., one that was not initially present in the system. We prove that during such a snap operation, there is at least one Classifier call in which the result is returned from the "winner case", or in other words, traverses to a right child somewhere in its path down the Classifier Tree. Another way to view this result is that such a snap operation cannot end up in the leftmost leaf of the Classifier Tree, since the view returned by this snap is necessarily more up-to-date than what is seen by the operation that added the new process.

Lemma 7.9. *Let op_i and op_s be two operations executed by p_i and p_s respectively. If the following conditions are met:*

1. *op_i is linearized before op_s*
2. *op_i is an add operation where p_j was added*
3. *$p_i \in addr_{final}$ during op_s*
4. *$p_j \in BL_{p_i,ss_{final},op_s}.active$*

then we prove that, during the execution of op_s, there exists at least one level where line 6 of the Classifier algorithm will evaluate to true.

Proof. We will use proof by contradiction. Assume that there exists no such level where line 6 of the Classifier algorithm evaluates to true during op_s.

1. Since there exists no such level where line 6 of the Classifier evaluates to true, at the final level op_s will reach the leftmost leaf node in the Classifier Tree. The label for this node in the Classifier Tree is 1. Therefore, at every level, starting at $\ell = 1$ to $\ell = \log(m) + 1$, the value of the getSeqSum(mergeRecent) ≤ 1 in the Classifier algorithm.

2. Since $p_j \in BL_{p_i,ss_{final},op_s}$.active, BL_{p_i,ss_{final},op_s} was found during op_s. Therefore, BL_{p_i,ss_{final},op_s}.seq is also used calculate the seqSum at line 6. This means $BL_{p_i,ss_{final},op_s}.\text{seq} \geq 1$.

3. Since p_s is executing the current operation op_s, $p_s \in addr_{final}$ and therefore BL_{p_s,ss_{final},op_s} exists. Here, $BL_{p_s,ss_{final},op_s}.\text{seq} \geq 1$ and this block is also used to calculate the seqSum at line 6.

4. Therefore, the seqSum at line 6, at the final level of the Classifier algorithm, is greater than or equal to 2. Therefore, line 6 will evaluate to true at the final level.

5. This contradicts with the assumption that there exists no level where line 6 evaluates to true.

□

In the next lemma, we prove that if an add(p) operation is linearized before a snap operation, then the address list returned after the final call to Classifier during the snap operation will include p. This fact is essentially the converse of Lemma 7.5

Lemma 7.10. *Let op_s be a snap operation executed by p_s. Let ss_{final} be the result after the final call to Classifier. Let op_i be an add operation executed by p_i where p_j was added. Let op_i be linearized before op_s.*

We prove that p_j appears in $addr_{final}$ during op_s.

Proof.

1. There can be only one operation that adds a process (in other words: A process can only be added once). So, op_i is the most recent add operation that added p_j which is linearized before op_s.

2. Since op_i is linearized before op_s, from the Linearization Definition algorithm (Algorithm: 10), we know either of the following cases will be true:

   (a) $BL_{p_i,ss_{final},op_i}.\text{seq} \leq BL_{p_i,ss_{final},op_s}.\text{seq}$

   (b) $op_i \rightarrow op_s$.

   If $op_i \rightarrow op_s$, we know from Lemma 7.3, $BL_{p_i,ss_{final},op_i}.\text{seq} \leq BL_{p_i,ss_{final},op_s}.\text{seq}$.

3. From Lemma 9.9, we know: $BL_{p_i,ss_\ell,op_i} == BL_{p_i,S,op_i}$ and $p_i \in addr_\ell$ for any level ℓ. When $\ell = \log(m) + 1$, $ss_\ell == ss_{final}$. Therefore, $BL_{p_i,ss_{final},op_i} == BL_{p_i,S,op_i}$ and $p_i \in addr_{final}$.

   From Lemma 7.2, we know there exists a process p_k such that: $BL_{p_i,ss_{final},op_s} == BL_{p_i,S,op_k}$ and during op_k, $p_i \in addr_0$.

   From the previous step, we know:
   $=> BL_{p_i,ss_{final},op_i}.\text{seq} \leq BL_{p_i,ss_{final},op_s}.\text{seq}$
   $=> BL_{p_i,S,op_i}.\text{seq} \leq BL_{p_i,S,op_k}.\text{seq}$ (since, $BL_{p_i,ss_{final},op_i} == BL_{p_i,S,op_i}$ and $BL_{p_i,ss_{final},op_s} == BL_{p_i,S,op_k}$)

4. Consider the case $BL_{p_i,S,op_i}.\text{seq} == BL_{p_i,S,op_k}.\text{seq}$, we know the following statements are also true:

(a) $p_i \in addr_0$ during op_i (by system design)

(b) $p_i \in addr_0$ during op_k (from step 3)

Therefore, from Lemma 7.1, we know:
=> $BL_{p_i,S,op_i} == BL_{p_i,S,op_k}$.
=> BL_{p_i,S,op_i}.active == BL_{p_i,S,op_k}.active

5. Consider the case BL_{p_i,S,op_i}.seq < BL_{p_i,S,op_k}.seq, we know the following statements are also true:

   (a) $p_i \in addr_0$ during op_i (from Section: 3.1)

   (b) $p_i \in addr_0$ during op_k (from step 3)

   Therefore, from Lemma 7.7, we know: BL_{p_i,S,op_i}.active $\subseteq$ BL_{p_i,S,op_k}.active

6. Combining the previous two steps: since BL_{p_i,S,op_i}.seq $\leq$ BL_{p_i,S,op_k}.seq, BL_{p_i,S,op_i}.active $\subseteq$ BL_{p_i,S,op_k}.active.

7. Since p_j was added during op_i, at line *SWRITE*, we know: $p_j \in BL_{p_i,S,op_i}$.active.

8. From step 3, we know:
   => $BL_{p_i,S,op_k} == BL_{p_i,ss_{final},op_s}$
   => BL_{p_i,S,op_k}.active == BL_{p_i,ss_{final},op_s}.active
   => BL_{p_i,S,op_i}.active $\subseteq$ BL_{p_i,ss_{final},op_s}.active (from step 6)
   Since $p_j \in BL_{p_i,S,op_i}$.active, $p_j \in BL_{p_i,ss_{final},op_s}$.active

9. From step 3 and the previous step, we know the following:

   (a) $p_i \in addr_{final}$ during op_s

   (b) $p_j \in BL_{p_i,ss_{final},op_s}$.active

   In the following steps, we show that $p_j \in addr_{final}$ during op_s. We will consider two cases.

10. Case 1: Consider the level a in the Classifier Tree during the execution of op_s where the following conditions are true:

(a) BL_{p_i,ss_{a-1},op_s} != BL_{p_i,ss_{final},op_s}

(b) $BL_{p_i,ss_a,op_s} == BL_{p_i,ss_{final},op_s}$

11. During the execution of the Classifier algorithm at level a, ss_{a-1} is part of the input and ss_a is the output of the Classifier algorithm. Since the block BL_{p_i} is not the same in the input bag and the output bag, it must be the case that Classifier algorithm returned from line 9.

12. In order to reach line 9, line 7 and 8 must be executed.

13. At line 8, GetMergeRecent algorithm is executed. The result of the GetMergeRecent is returned from the Classifier at line 9 as ss_a.

14. We assumed $BL_{p_i,ss_a,op_s} == BL_{p_i,ss_{final},op_s}$ and from step 9 we know $p_j \in BL_{p_i,ss_{final},op_s}$.active. Therefore, $p_j \in BL_{p_i,ss_a,op_s}$.active (from Lemma 7.8).

15. The block BL_{p_i,ss_a,op_s} is the result of the merge operation during GetMergeRecent algorithm based on the bags in $localContainer$. From the definition of the merge property (Section: 3.2.3), there exists a block BL_{p_i} in some bag in $localContainer$ from where BL_{p_i,ss_a,op_s} is copied over, and, the active list in that block contains p_j.

16. Since $BL_{p_i,ss_a,op_s} == BL_{p_i,ss_{final},op_s}$ and $p_i \in addr_{final}$, $p_i \in addr_a$ (from Lemma 7.8).

17. At line 7 of the Classifier algorithm, GetUpdatedActiveList algorithm is executed. The list returned from GetUpdatedActiveList is returned as $addr_a$. Since $p_i \in addr_a$, $p_i \in$ updatedActiveList variable at line 7.

18. Since the return result from GetUpdatedActiveList will contain p_i, during the execution of GetUpdatedActiveList, the variable k will be equal to p_i at line 11 eventually.

19. At line 12, BL_{p_i} is retrieved from the bag $localss_j$. This $localss_j$ is written down to the $localContainer$ variable and from step 15 above, we know there is a block BL_{p_i} in $localContainer$ where the active list of the block contains p_j.

20. Therefore, eventually at line 14, p_j will be traversed and added to updatedActiveList variable. Since no items are removed from updatedActiveList, the list returned from the GetUpdatedActiveList algorithm will contain p_j.

21. The list returned from the GetUpdatedActiveList is returned as $addr_a$ from the Classifier algorithm. Therefore, $p_j \in addr_a$. From Lemma 9.2, we know $p_j \in addr_{final}$. (Proved for Case 1)

22. Case 2: Consider the case where $BL_{p_i,ss_0,op_s} == BL_{p_i,ss_{final},op_s}$.

23. Since $p_i \in addr_{final}$, $p_i \in addr_0$ (from Lemma 7.8). And since $p_j \in BL_{p_i,ss_{final},op_s}$.active, $p_j \in BL_{p_i,ss_0,op_s}$.active.

24. When the first call to the Classifier starts at level $\ell = 1$, ss_0 is written down at address p_s in the Container.

25. From Lemma 9.5, $p_s \in addr_0$.

26. At line 4 of the Classifier algorithm, GetUpdatedActiveList algorithm is executed where the input is $addr_0$.

27. Since $addr_0$ is assigned to updatedActiveList at line 2 of the GetUpdatedActiveList algorithm and $p_s \in addr_0$, at the first iteration of the while loop at line 5, j = p_s. The bag that was just written down at line 2 of the Classifier algorithm is read here where the bag is ss_0 (from step 24 above). Since $p_i \in addr_0$ and $addr_0$ is assigned to updatedActiveList variable at line 2 of the GetUpdatedActiveList, when j = p_s, eventually at line 11, k = p_i.

28. Therefore, BL_{p_i,ss_0,op_s} will be read and assigned to r_k in the GetUpdatedActiveList algorithm.

29. From the active list of r_k, eventually p_j will be found (from step 23, $p_j \in BL_{p_i,ss_0,op_s}$.active) and added to the updatedActiveList variable. Since nothing is removed from the updatedActiveList variable, the list returned at line 4 of the Classifier algorithm will contain p_j.

30. If line 6 of the Classifier algorithm evaluates to true then, the arguments from steps 26 to 29 are applied again for the execution of lines line 7 and 8. Since no items are removed from the active list, p_j will still exist and returned from the Classifier algorithm as $p_j \in addr_1$. From Lemma 9.2, we know $p_j \in addr_{final}$. (Proved for Case 2)

31. If line 6 evaluates to false then, from Lemma 7.9, there is some deeper level in the Classifier Tree where line 6 of the Classifier algorithm evaluates to true. Consider the first such level, and apply the argument from steps 24 to 30 at that level instead of level 1. Then the previous step applies. (Proved for Case 2)

32. From Case 1 and Case 2, $p_j \in addr_{final}$.

□

In the next lemma, we consider the bag returned by the final call to Classifier during a snap operation. For each block p_i in that bag, we prove that the removeActive list is exactly the same as the removeActive list that was written by p_i in its shared memory register during its most recent remove operation. This proof is similar to the proof of Snap Validity 1 (Lemma 7.4), the only difference is we are considering the remove operation and the removeActive variable instead of the update operation and the val variable of the block.

Lemma 7.11. *Let op_s be a snap operation executed by p_s. Let ss_{final} be the bag returned by the final call to Classifier in an operation. Let op_i be the most recent remove operation executed by process p_i. Let $\beta\ell_{p_i}$ be the block written at line SWRITE of the remove operation op_i by p_i in shared memory S.*

We prove that $BL_{p_i,ss_{final},op_s}.removeActive == \beta\ell_{p_i}.removeActive$.

Proof. Since op_i is linearized before op_s, either line 14 or line 3 in the Linearization algorithm (Algorithm: 10) must be true.

1. If line 3 of the Linearization algorithm (Algorithm: 10) is true, we know $op_i \rightarrow op_s$. We know, if $op_i \rightarrow op_s$ is true, then from Lemma 7.3, $BL_{p_i,ss_{final},op_i}.\text{seq} \leq BL_{p_i,ss_{final},op_s}.\text{seq}$.

2. Also, if line 14 of the Linearization algorithm (Algorithm: 10) is true, we know from the Linearization algorithm (Algorithm: 10) line 14 that BL_{p_i,ss_{final},op_i}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq.

3. So, we prove for: BL_{p_i,ss_{final},op_i}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq

Case 1: BL_{p_i,ss_{final},op_i}.seq $< BL_{p_i,ss_{final},op_s}$.seq:

(a) From Lemma 9.9, we know, during op_i at any level ℓ, when p_i executes line *POSTC* of the Remove algorithm, $BL_{p_i,ss_\ell,op_i} == \beta\ell_{p_i}$. So, at the end of the final call to Classifier when $\ell = \log(m) + 1$, $BL_{p_i,ss_{\log(m)+1},op_i} == \beta\ell_{p_i}$. Therefore, $BL_{p_i,ss_{final},op_i} == \beta\ell_{p_i}$.

(b) Since, $BL_{p_i,ss_{final},op_i} == \beta\ell_{p_i}$ and the case assumption is BL_{p_i,ss_{final},op_i}.seq $< BL_{p_i,ss_{final},op_s}$.seq, we can say, $\beta\ell_{p_i}$.seq $< BL_{p_i,ss_{final},op_s}$.seq.

(c) In the following steps we will use a term called *intermediate operation*. We define an operation op_j an intermediate operation if it is performed by p_i after op_i during which the line *SWRITE* took place before the line where BL_{p_i,ss_{final},op_s} was calculated during op_s, and, $\beta\ell_{p_i}$.seq $<$ BL_{p_i,S,op_j}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq. Note that it is also possible for op_s to satisfy the conditions for it to be an intermediate operation, but it is not possible for op_i. We now argue that there exists at least one such intermediate operation and, also, for at least one of the intermediate operations op_j, BL_{p_i,S,op_j}.seq $== BL_{p_i,ss_{final},op_s}$.seq. We can say this because of the following facts:

   i. The seq value never decreases anywhere in the algorithms

   ii. Because of SWMR property, only p_i can write in any of the block BL_{p_i}

(d) Let op_j be an arbitrary intermediate operation. Since BL_{p_i,S,op_j}.seq $> \beta\ell_{p_i}$.seq and SWMR property, op_i is partially ordered before op_j. Therefore, from the Linearization algorithm (10) line 3, op_i is linearized before op_j.

(e) We argue that if op_j is a modify operation (add, update or remove operation) then op_j is linearized before op_s:

i. From the assumption about op_j in Step: 1c, we know that BL_{p_i,S,op_j}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq.

ii. Since p_i and p_j are the same processes, BL_{p_i,S,op_j}.seq can be written as BL_{p_j,S,op_j}.seq.

iii. From Lemma 9.9, we know, during op_j at any level ℓ, when p_j executes line *POSTC* of the algorithm, $BL_{p_j,ss_\ell,op_j} == BL_{p_j,S,op_j}$. So, at the end of the final call to Classifier when $\ell = \log(m) + 1$, $BL_{p_j,ss_{\log(m)+1},op_j} == BL_{p_j,S,op_j}$. Therefore, $BL_{p_j,ss_{final},op_j} == BL_{p_j,S,op_j}$.

iv. From step 1(e)i we know BL_{p_i,S,op_j}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq and from step 1(e)iii we know $BL_{p_j,ss_{final},op_j} == BL_{p_j,S,op_j}$. Therefore, BL_{p_j,ss_{final},op_j}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq.

v. From the Linearization algorithm (10) line 14, op_j is linearized before op_s.

(f) We argue that op_j cannot be a remove operation. This is because from step 1d we know op_i is linearized before op_j and from step 1e we know op_j is linearized before op_s. However, this means that the assumption in the lemma statement where it states op_i is the most recent remove operation linearized before op_s no longer holds.

(g) Now that we have shown that every intermediate operation is not a remove operation, in the following steps we will show that the removeActive in block BL_{p_i,ss_{final},op_s} comes from $\beta\ell_{p_i}$.removeActive. The removeActive property of the block BL_{p_i} in the shared memory S is only updated during the remove operation at line *SWRITE*. Since op_j is not a remove operation, the removeActive property of the block BL_{p_i} remain unchanged during op_j as well. Therefore, during op_j, BL_{p_i,S,op_i}.removeActive $== BL_{p_i,S,op_j}$.removeActive.

(h) In the following steps we consider another arbitrary intermediate operation op_j where BL_{p_i,S,op_j}.seq == BL_{p_i,ss_{final},op_s}.seq. (Recall that we have shown in step 1c that at least one such operation exists)

(i) From Lemma 7.2, we know there exists an op_k for which BL_{p_i,ss_{final},op_s} == BL_{p_i,S,op_k} and, $p_i \in addr_0$ during op_k. Using this equality and BL_{p_i,ss_{final},op_s}.seq == BL_{p_i,S,op_j}.seq (from step 1h), BL_{p_i,S,op_k}.seq == BL_{p_i,S,op_j}.seq.

(j) We know the following:

   i. The seq values are same at address p_i in the shared memory S during op_k and op_j. (from the previous step)

   ii. $p_i \in addr_0$ during op_k (from the previous step)

   iii. $p_i \in addr_0$ during op_j (from Lemma 9.5 since op_j is performed by p_i)

(k) Therefore, from Lemma 7.1, for op_j and op_k, BL_{p_i,S,op_k} == BL_{p_i,S,op_j}.

(l) From the previous step we know, BL_{p_i,S,op_k} == BL_{p_i,S,op_j} and from step 1i we know, BL_{p_i,ss_{final},op_s} == BL_{p_i,S,op_k}, therefore, BL_{p_i,ss_{final},op_s} == BL_{p_i,S,op_j}.
Therefore, BL_{p_i,ss_{final},op_s}.removeActive == BL_{p_i,S,op_j}.removeActive

(m) From step 1l we know,
BL_{p_i,ss_{final},op_s}.removeActive == BL_{p_i,S,op_j}.removeActive. From step 1g we know,
BL_{p_i,S,op_i}.removeActive == BL_{p_i,S,op_j}.removeActive.
Therefore, BL_{p_i,ss_{final},op_s}.removeActive == BL_{p_i,S,op_i}.removeActive.
In other words, BL_{p_i,ss_{final},op_s}.removeActive == $\beta\ell_{p_i}$.removeActive (Case 1 proved)

Case 2: BL_{p_i,ss_{final},op_i}.seq == BL_{p_i,ss_{final},op_s}.seq:

(a) From Lemma 7.2, there exists a process p_k such that BL_{p_i,ss_{final},op_s} == BL_{p_i,S,op_k} and, $p_i \in addr_0$ during op_k.

(b) From Lemma 9.9, for level $\ell = \log(m) + 1$, $BL_{p_i, ss_{\log(m)+1}, op_i} == \beta\ell_{p_i}$. Since, $ss_{\log(m)+1} == ss_{final}$, $BL_{p_i, ss_{final}, op_i} == \beta\ell_{p_i}$

(c) We know: $BL_{p_i, ss_{final}, op_i}$.seq $==$ $BL_{p_i, ss_{final}, op_s}$.seq (from Case 2 assumption)

(d) From step (a) using $BL_{p_i, ss_{final}, op_s} == BL_{p_i, S, op_k}$, it follows from the previous step that BL_{p_i, S, op_k}.seq $==$ $BL_{p_i, ss_{final}, op_i}$.seq

(e) From step (b) using $BL_{p_i, ss_{final}, op_i} == \beta\ell_{p_i}$, it follows from the previous step that BL_{p_i, S, op_k}.seq $==$ $\beta\ell_{p_i}$.seq ($==$ BL_{p_i, S, op_i} by Lemma 9.9)

(f) We know the following:

i. The seq values are same at address p_i in the shared memory S during op_k and op_i (from step (e) of this case).

ii. $p_i \in addr_0$ during op_k (from step (a) of this case)

iii. $p_i \in addr_0$ during op_i (from Lemma 9.5)

(g) Therefore, for op_i and op_k, from Lemma 7.1, $BL_{p_i, S, op_k} == BL_{p_i, S, op_i}$. We know from the lemma statement, $\beta\ell_{p_i} == BL_{p_i, S, op_i}$. So, BL_{p_i, S, op_k} $== \beta\ell_{p_i}$.

(h) Therefore, BL_{p_i, S, op_k}.removeActive $==$ $\beta\ell_{p_i}$.removeActive

(i) From step (a), we know $BL_{p_i, S, op_k} == BL_{p_i, ss_{final}, op_s}$ and so $BL_{p_i, ss_{final}, op_s}$.removeActive $==$ $\beta\ell_{p_i}$.removeActive.

□

In the next lemma, we prove that if a remove(p) operation is linearized before a snap operation, then p will appear in the removeActive list of at least one block in the bag returned by the final call to Classifier during the snap operation. This lemma will be used in the proof of Snap Validity 2 to ensure that all previous remove operations are reflected in the result of any snap operation.

Lemma 7.12. *Let op_s be a snap operation executed by p_s. Let ss_{final} be the bag returned by the final call to Classifier during an operation. Let op_i be a remove operation executed by p_i where p_j was removed. Let op_i be linearized before op_s.*

We prove that, p_j appears in $\bigcup_{p_x \in addr_{final}}$ $(BL_{p_x, ss_{final}, op_s}.removeActive)$ during op_s.

Proof.

1. There can be only one operation that removes a process (in other words: a process can only be removed once). So, op_i is the most recent remove operation that removed p_j which is linearized before op_s.
2. From Lemma 7.11, it follows that BL_{p_i,S,op_i}.removeActive == BL_{p_i,ss_{final},op_s}.removeActive.
3. Since p_i performed op_i, BL_{p_i,S,op_i} is written down into shared memory S at line *SWRITE* during op_i. Since BL_{p_i,S,op_i} is not empty or null, BL_{p_i,ss_{final},op_s} exists. In other words, $p_i \in addr_{final}$ during op_s.
4. At line *SWRITE* during op_i, $p_j \in BL_{p_i,S,op_i}$.removeActive.
5. Since the following conditions are true:
   (a) BL_{p_i,S,op_i}.removeActive == BL_{p_i,ss_{final},op_s}.removeActive
   (b) $p_j \in BL_{p_i,S,op_i}$.removeActive
   (c) $p_i \in addr_{final}$ during op_s

   it follows that $p_j \in BL_{p_i,ss_{final},op_s}$.removeActive

□

Finally, we are able to prove the main claim of this section, i.e., that the second condition of snap validity holds: the result of a snap operation contains exactly the processes that were initially in the system along with all processes that were previously added but not yet removed.

Lemma 7.13. *Let op_s be a snap operation executed by p_s. Let ss_{final} be the bag returned by the final call to Classifier during an operation. Let $activeAddr = addr_{final}$ $- \bigcup_{p_i \in addr_{final}}$ $(BL_{p_i,ss_{final},op_s}.removeActive)$.*

We prove that activeAddr exactly contains the processes in $procList_\alpha$ as well as each p_i that was previously added and not yet removed.

Proof. To show $activeAddr$ contains the $procList_\alpha$, we prove that $procList_\alpha \subseteq addr_{final}$ and $procList_\alpha \cap \bigcup_{p_i \in addr_{final}}(BL_{p_i,ss_{final},op_s}.\text{removeActive}) = \emptyset$ during op_s in the following steps:

1. Proof for: $procList_\alpha \subseteq addr_{final}$ during op_s:

   (a) Consider any process $p_a \in procList_\alpha$. From the model definition (Section: 3.1), $p_a \in BL_{p_s,S,op_s}$. Therefore, at line *ADDR0INIT*, $p_a \in addr_0$.

   (b) During op_s when $\ell = 1$ at line *PREC*, $p_a \in addr_0$. From Lemma 9.2, $p_a \in addr_{\log(m)+1}$. Since the final level is $\log(m)+1$, $p_a \in addr_{final}$. Therefore, $procList_\alpha \subseteq addr_{final}$.

2. Proof for: $procList_\alpha \cap \bigcup_{p_i \in addr_{final}}(BL_{p_i,ss_{final},op_s}.\text{removeActive}) = \emptyset$ during op_s:

   (a) A process is added to the removeActive list of a block at line 4 of the remove operation. Nowhere else in the code, a process is added to the removeActive list of any block. The process that is added to the removeActive list is the process that is being removed during the remove oepration. Therefore, $\bigcup_{p_i \in addr_{final}}(BL_{p_i,ss_{final},op_s}.\text{removeActive})$ contains only the processes that are to be removed or has already been removed.

   (b) From the model definition (Section 3.1), a process p_j can only be removed by a process that added p_j to the system. Therefore, p_j is a process that was added during an add operation. Since $procList_\alpha$ is a list of processes that is not added by any other processes (rather they were already present in the system when the system initialized, from model definition Section: 3.1), $p_j \notin procList_\alpha$.

   (c) A process that exists in removeActive list of any block cannot be part of the $procList_\alpha$. Therefore,
   $procList_\alpha \cap \bigcup_{p_i \in addr_{final}}(BL_{p_i,ss_{final},op_s}.\text{removeActive}) = \emptyset$ during op_s.

Consider any integer $p_j \notin procList_\alpha$. We consider the following 3 cases to prove the lemma statement:

Case 1: If there exist no add operations (linearized before op_s) where p_j was added, then, $activeAddr$ does not contain p_j.

(a) From Lemma 7.5, we know for any arbitrary process $p_j \in addr_{final}$ during op_s, there exists an add operation op_a (where p_j was added) and op_a is linearized before op_s.

(b) The contrapositive of this is that, if there exists no add operation op_a (where p_j was added) linearized before op_s then, $p_j \notin addr_{final}$ during op_s.

(c) From the definition of $activeAddr$ in the lemma statements, since $p_j \notin addr_{final}$ during op_s, $p_j \notin activeAddr$ during op_s.

Case 2: If there exists an add operation (linearized before op_s) where p_j was added and there exist no remove operations (linearized before op_s) where p_j was removed, then, $activeAddr$ contains p_j.

(a) From Lemma 7.10, we know if there exists an add operation op_a (where p_j was added) linearized before op_s then, $p_j \in addr_{final}$ during op_s.

(b) From Lemma 7.6, we know if any arbitrary process $p_j \in \bigcup_{p_x \in addr_{final}}$ $(BL_{p_x, ss_{final}, op_s}.\text{removeActive})$ during op_s, there exists an operation op_r (where p_j was removed) which is linearized before op_s. The contrapositive of this is that, if there exists no remove operation op_r (where p_j was removed) linearized before op_s then,
$p_j \notin \bigcup_{p_x \in addr_{final}} (BL_{p_x, ss_{final}, op_s}.\text{removeActive})$ during op_s.

(c) From the lemma definition, $activeAddr = addr(ss_{final}) - \bigcup_{p_i \in addr(ss_{final})}$ $(BL_{p_i, ss_{final}, op_s}.\text{removeActive})$. From the results of steps (2a) and (2b), we conclude that $p_j \in activeAddr$.

Case 3: If there exists an add operation (linearized before op_s) where p_j was added and there exists a remove operation (linearized before op_s and after the add operation) where p_j was removed, then, $activeAddr$ does not contain p_j.

(a) From Lemma 7.10, we know if there exists an add operation op_a (where p_j was added) linearized before op_s then, $p_j \in addr_{final}$ during op_s.

(b) From Lemma 7.12, we know if there exists a remove operation op_r (where p_j was removed) linearized before op_s then,
$p_j \in \bigcup_{p_x \in addr_{final}} (BL_{p_x, ss_{final}, op_s}.\text{removeActive})$ during op_s.

(c) Therefore, using the equation of $activeAddr$ in the lemma statement and values of the equation in step (3a) and (3b), $p_j \notin activeAddr$.

Concluding statement: From Case 2 above, we know that each process p_i that is previously added and not yet removed is in $activeAddr$, and, from Cases 1 and 3, we know that no other process is in $activeAddr$. □

8 Analysis of Running Time

In this section, we analyze the worst-case cost of the Add, Remove, Snap, and Update operations. It is important to recall that our definition of cost only considers the number of shared memory operations, i.e., reads and writes to atomic registers, since in real systems these costs are orders of magnitude larger than local operations performed at each process. The local reads and writes are considered O(1). Let h be the height of the Classifier Tree, which is a binary tree with height $\lceil \log_2(m) \rceil + 1$. Let n be the total number of processes that are ever active in the system. Let n_0 be the number of processes in the system when it is initialized. Let m denote an upper bound on the number of operations that must be supported by the snapshot implementation.

For any algorithm A, we will denote its worst-case cost using the notation T_A. For example, the cost of AddToClassifierTree will be represented by $T_{AddToClassifierTree}$.

8.1 Analysis of Add

1. At line 3, AddToClassifierTree algorithm is executed.

2. At line *SREAD1* and *SWRITE*, there is a read and a write from the shared memory respectively.

3. At line *SREAD2*, there is a read operation from the shared memory which can iterate at most n times.

4. At line *POSTC*, we have a call to Classifier inside a loop that repeats $\lceil \log_2(m) \rceil$ times.

So, overall, the worst-case cost of an Add operation is $1 + 1 + n + T_{AddToClassifierTree}$ $+ \lceil \log_2(m) \rceil * T_{Classifier}$

8.2 Analysis of Remove, Snap, and Update

1. At line *SREAD1* and *SWRITE*, there is a read and a write from the shared memory respectively.

2. At line *SREAD2*, there is a read operation from the shared memory which can iterate at most n times.

3. At line *POSTC*, we have a call to Classifier inside a loop that repeats $\lceil \log_2(m) \rceil$ times.

So, overall, the worst-case cost of a Remove, Snap, or Update operation is $1 + 1 + n$ $+ \lceil \log_2(m) \rceil * T_{Classifier}$

8.3 Analysis of AddToClassifierTree

The Classifier Tree is a perfect binary tree. The number of leaves in the Classifier Tree was chosen to be the smallest power of 2 that is greater than or equal to m, so the number of leaves is at most m. The number of nodes in a perfect binary tree is at most twice the number of leaves, so the total number of nodes is at most $2m$. In this algorithm, a write to the shared memory C is performed at each node of the Classifier Tree. Since it is a binary tree, every node will be visited once during the DFS traversal. So, the worst-case cost is bounded above by $\Theta(m)$.

8.4 Analysis of Classifier

1. At line 2, there is a write to the shared memory.
2. At line 4, the cost is $T_{GetUpdatedActiveList}$
3. At line 5, the cost is $T_{GetMergeRecent}$
4. At line 6, the cost is $T_{getSeqSum}$
5. At line 7, the cost is $T_{GetUpdatedActiveList}$
6. At line 8, the cost is $T_{GetMergeRecent}$

So, overall, the worst-case cost of a call to Classifier is $1 + 2{*}T_{GetUpdatedActiveList} + 2{*}T_{GetMergeRecent} + T_{getSeqSum}$

8.5 Analysis of getSeqSum

There is no read/write to the shared memory in this algorithm, so the worst-case cost is $\Theta(1)$.

8.6 Analysis of GetMergeRecent

There is no read/write to the shared memory in this algorithm, so the worst-case cost is $\Theta(1)$.

8.7 Analysis of GetUpdatedActiveList

The only shared memory operation in this algorithm is at line 8. We argue that this repeats for at most n times.

Line 8 is repeated for each process in toScan list. The if condition from line 15 to 18, ensures that once a process is visited, it is never added back to toScan list. By the definition of n, we know that there are at most n different processes ever in the system. Therefore, the disjoint union of all toScan lists at line 7 cannot have more than n processes. It follows that line 8 can repeat at most n times.

Therefore, the worst-case cost of this algorithm is $\Theta(n)$.

8.8 Worst-Case Costs

Combining the above worst-case analysis of the individual subroutines, we obtain worst-case cost bounds for Add, Remove, Snap, and Update in terms of the system parameters n and m.

8.8.1 Add Operation

Up to multiplicative constants, the worst-case cost is:

$$\begin{aligned}
&1 + 1 + n + T_{AddToClassifierTree} + \lceil \log_2(m) \rceil \cdot T_{classifier} \\
&= 2 + n + m + \lceil \log_2(m) \rceil \cdot (1 + 2 \cdot T_{GetUpdatedActiveList} + 2 \cdot T_{GetMergeRecent} + T_{getSeqSum}) \\
&= 1 + n + m + \lceil \log_2(m) \rceil \cdot (1 + 2 \cdot n + 2 \cdot 1 + 1)
\end{aligned}$$

Asymptotically, this bound is $O(m + n \log_2(m))$.

8.8.2 Remove, Snap and Update Operations

Up to multiplicative constants, the worst-case cost is:

$$\begin{aligned}
&1 + 1 + n + \lceil \log_2(m) \rceil \cdot T_{classifier} \\
&= 2 + n + \lceil \log_2(m) \rceil \cdot (1 + 2 \cdot T_{GetUpdatedActiveList} + 2 \cdot T_{GetMergeRecent} + T_{getSeqSum}) \\
&= 2 + n + \lceil \log_2(m) \rceil \cdot (1 + 2 \cdot n + 2 \cdot 1 + 1)
\end{aligned}$$

Asymptotically, this bound is $O(n \log_2(m))$.

9 Supporting Technical Lemmas

In this section, we have collected various technical lemmas that are used throughout the earlier parts of this thesis. Our new lemmas are provided in Section 9.1, while Section 9.2 restates results from Attiya and Rachman [16] using the terminology of our thesis and also provides proofs that were not included in Attiya and Rachman [16].

As a reminder, when we use the words "before" and "after" to compare two operations, we strictly mean to say *linearized before* or *linearized after* as defined by our linearization in Section 6.

9.1 New Lemmas

In the next lemma, we prove that if a process is present in the address list that is provided as input into the Classifier algorithm, then that process will be present in the output address list returned by the call to Classifier. This lemma shows that once a process is found, it never gets lost during the execution of any operation.

Lemma 9.1. *Consider an arbitrary process p_i, and let op_i be an operation executed by process p_i. During op_i, at any level ℓ, let a process $p_u \in addr_{\ell-1}$ at line 2 of the Classifier algorithm where $addr_{\ell-1}$ is the input parameter of the Classifier algorithm. We prove that, p_u will be present in the list $addr_\ell$ of process addresses returned by Classifier.*

Proof.

1. There are two lines in the Classifier algorithm from where the address list of processes is returned. If the Classifier algorithm's values are returned at line 11, then, p_u will be present in the address list of processes returned from the Classifier algorithm because $p_u \in addr_{\ell-1}$ and $addr_{\ell-1}$ is returned here. The other line from where the Classifier algorithm's values can be returned is at line 9. We will argue about this return statement from now on.

2. $addr_{\ell-1}$ is assigned to the currentActive variable at line 3. This variable is passed to the GetUpdatedActiveList algorithm where it is then assigned to the updatedActiveList variable. No items are removed from the updatedActiveList throughout the GetUpdatedActiveList algorithm. Therefore, at the end of the GetUpdatedActiveList when updatedActiveList is returned, it will contain p_u.

3. The returned list value from GetUpdatedActiveList is assigned to the updatedActiveList variable at line 4 of the Classifier algorithm. Therefore, at line 4 of the Classifier algorithm, the updatedActiveList variable will contain p_u.

4. Since in our current case, the Classifier algorithm's values will be returned from line 9, line 6 will be true.

5. At line 7 of the Classifier algorithm, the updatedActiveList variable will be reassigned with the updated list value from the GetUpdatedActiveList algorithm. Similar to the previous steps, since the input parameter of the GetUpdatedActiveList contains p_u, the return list will also contain p_u. Therefore, at line 7, updatedActiveList variable will contain p_u.

6. The updatedActiveList variable is returned at line 9. Therefore, p_u will be present in address list of processes returned from the Classifier algorithm.

□

In the next lemma, we prove that if a process is found in the input address list before the call to the Classifier algorithm, then that process will be present in the output address list of all the calls to the Classifier algorithm for the subsequent levels. This is an important property of the Classifier algorithm: once a process is found, it does not get lost while executing Classifier at the various levels of the Classifier Tree.

Lemma 9.2. *Consider an arbitrary process p_i, and let op_i be an operation executed by process p_i. During op_i, at any level $\ell \geq 1$, let a process $p_u \in addr_{\ell-1}$ at line PREC. We prove that for any level $\ell_d \geq \ell$, at line POSTC, $p_u \in addr_{\ell_d}$.*

Proof. Let P(ℓ_d) = "for any level ℓ_d, at line *POSTC*, $p_u \in addr_{\ell_d}$". We will use a proof by induction on $\ell_d \geq \ell$.

Base Case: $(\ell_d == \ell)$

1. We know from the lemma statement, $p_u \in addr_{\ell-1}$. $addr_{\ell-1}$ is the input to the Classifier algorithm at level ℓ.

2. We know from Lemma 9.1 that, if p_u is in the input of the Classifier algorithm, then the output of the Classifier algorithm will also contain p_u.

3. The return value of the Classifier is received at line *POSTC*. Therefore, at line *POSTC*, $p_u \in addr_{\ell_d}$.

Induction Hypothesis: Assume of an level ℓ_d where $\ell_d > \ell$, $P(\ell) \wedge ... \wedge P(\ell_d - 1)$ is true.

Induction Steps:

1. Since $P(\ell_d - 1)$ is true, at line *POSTC* of level $\ell_d - 1$, $p_u \in addr_{\ell_d - 1}$.
2. At level ℓ_d, the value of $addr_{\ell_d - 1}$ at line *PREC* is same as the value of $addr_{\ell_d - 1}$ at line *POSTC* of level $\ell_d - 1$.
3. Therefore, at line *PREC* in level ℓ_d, $p_u \in addr_{\ell_d - 1}$.
4. From Lemma 9.1, we know $p_u \in addr_{\ell_d}$ at line *POSTC*.

□

In the next lemma, we prove that when a process p_i writes a block into the shared memory S during an operation op_i, then this is the most up-to-date block that any process p_j can possibly read from address p_i in S during any read that happens before p_i starts another operation after op_i. This lemma is used when establishing the base case of induction arguments such as in Lemmas 9.6 and 9.9

Lemma 9.3. *Consider an arbitrary process p_i, and let op_i be any operation that is being executed by p_i. Let the seq value written by p_i in the shared memory be seq_{p_i} at line SWRITE.*

Let p_j be any process in procList other than p_i. Let op_j be any operation where the line PRECFirst is executed before the next operation by p_i.

We prove that, at line PRECFirst during op_j, $BL_{p_i,ss_0,op_j}.seq \leq seq_{p_i}$.

Proof. If $p_i \notin addr_0$ during op_j then, BL_{p_i,ss_0,op_j} does not exist. Therefore, BL_{p_i,ss_0,op_j}.seq is null or zero. Since seq_{p_i} is always greater than or equal to 1, BL_{p_i,ss_0,op_j}.seq $\leq seq_{p_i}$

is proved. From now on, we will consider the case where $p_i \in addr_0$ during op_j. $addr_0$ is initialized at line *ADDR0INIT* where the value is BL_{p_j,S,op_j}.active (which was read at line *SREAD1*). Therefore, $p_i \in BL_{p_j,S,op_j}$.active.

There can be two scenarios:

1. Scenario 1: op_i executes line *SWRITE* and then, op_j executes line *SREAD2*.

   (a) op_i executes line *SWRITE* where seq_{p_i} is written in the shared memory S at BL_{p_i,S,op_i}.

   (b) op_j executes line *SREAD2* where it reads BL_{p_i,S,op_j} from the shared memory S. And, this block exists here because $p_i \in BL_{p_j,S,op_j}$.active.

   (c) $BL_{p_i,S}$ can only be updated by an operation by p_i (because of SWMR property) at line *SWRITE*. From the assumption in the lemma statement, before this next operation by p_i is executed, op_j reaches line *PRECFirst* which is after the line *SREAD2* during op_j. Therefore, during op_j at line *SREAD2*, BL_{p_i,S,op_j} is the same block as the one BL_{p_i,S,op_i} wrote during p_i at line *SWRITE*.

   (d) $BL_{p_i,S,op_j} == BL_{p_i,S,op_i}$

   (e) BL_{p_i,S,op_j}.seq $==$ BL_{p_i,S,op_i}.seq

   (f) BL_{p_i,S,op_j}.seq $==$ seq_{p_i} (from the definition of seq_{p_i})

2. Scenario 2: op_j executes line *SREAD2* and then, op_i executes line *SWRITE*.

   (a) op_j executes line *SREAD2* where it reads BL_{p_i,S,op_j} from the shared memory S. And, this block exists here because $p_i \in BL_{p_j,S,op_j}$.active.

   (b) op_i executes line *SWRITE* after op_j executes line *SREAD2* and updates the block BL_{p_i,S,op_i} at line *SWRITE*. The seq value is incremented by 1 in the new block.

   (c) This implies that BL_{p_i,S,op_j}.seq $+ 1 ==$ BL_{p_i,S,op_i}.seq

   (d) BL_{p_i,S,op_j}.seq $<$ BL_{p_i,S,op_i}.seq

(e) BL_{p_i,S,op_j}.seq $< seq_{p_i}$ (from the definition of seq_{p_i})

The values read from the shared memory S at line *SREAD2* are stored in the local memory ss_0 at the next line. So, combining the two scenarios above, at line *PRECFirst*, BL_{p_i,ss_0,op_j}.seq $\leq seq_{p_i}$. □

In the next lemma, we prove that any level of the Classifier Tree, for any block in the output bag of Classifier, there must be a corresponding call to Classifier, at the same level, where the input bag had the same block at the same address. In other words, this establishes a connection between consecutive levels of the Classifier Tree by showing how information flows down the tree: each block in the output bag at a Classifier node must have previously come down in an output bag from the parent node.

Lemma 9.4. *Consider arbitrary processes p_i and p_j, and let op_i and op_j be respective operations performed by p_i and p_j. Let op_j be an operation that is executing the Classifier algorithm at level ℓ, and let ss_ℓ be the output bag of this execution of the Classifier algorithm. We assume that during op_j, $p_i \in addr_\ell$ at level ℓ.*

We prove that:

1. *for block p_i in ss_ℓ, there exists a process p_k where $BL_{p_i,ss_\ell,op_j} == BL_{p_i,ss_{\ell-1},op_k}$ and during op_k, $p_i \in addr_{\ell-1}$.*
2. *If p_j set BL_{p_i,ss_ℓ,op_j} before p_i starts performing any further operation after op_i then, p_k sets its $ss_{\ell-1}$ before p_i starts any further operation after op_i.*

Proof.

1. BL_{p_i,ss_ℓ,op_j} is set based on the return value of the Classifier algorithm executed by p_j. Here, this block exists because of assumption in proof statement: during op_j, $p_i \in addr_\ell$.
2. When p_j executes the Classifier algorithm, it can either return from line 9 or line 11 of the Classifier algorithm.
3. If it returns from line 11, then we argue that setting $p_k = p_j$ satisfies the conditions of part 1 of the lemma statement. The proof for this case is as follows:

(a) The input values are returned. The input values that are returned at level ℓ are $ss_{\ell-1}$ and $addr_{\ell-1}$.

(b) Therefore, at line *POSTC*, $BL_{p_i,ss_\ell,op_j} == BL_{p_i,ss_{\ell-1},op_j}$. So, we choose $p_k = p_j$.

(c) Since $p_i \in addr_\ell$ and $addr_\ell == addr_{\ell-1}$, $p_i \in addr_{\ell-1}$.

(d) Therefore, it is proved: $BL_{p_i,ss_\ell,op_j} == BL_{p_i,ss_{\ell-1},op_k}$ and during op_k, $p_i \in addr_{\ell-1}$. (This proves part 1 of the lemma statement)

4. If it returns from line 9, then it returns the value calculated from algorithm *GetMergeRecent* (Algorithm: 8).

5. In the algorithm *GetMergeRecent*, merge is calculated based on the given set of processes' addresses and returned from the algorithm. We know from the definition of merge (in 3.2.3) that the block at address j in $mergedBA_{final}$ is the block with *maximal* seq value among all $BL_{j,i}$. This means, $BL_{j,i}$ must already exist before the merge.

6. In our case, BL_{p_i,ss_ℓ,op_j} is the block with *maximal* seq value (since it is part of the merge result of the *GetMergeRecent* algorithm). In the algorithm *GetMergeRecent*, this block is created at line 7 where the algorithm reads the block BL_{p_i} from $localss_j$. This $localss_j$ is created at line 5, where this bag is read from the localContainer (which is a copy of Container C) located at j = p_k. And since, this bag exists, it must have been written by some process p_k at line 2 of the Classifier algorithm.

7. So, p_k wrote the bag $localss_j$ where BL_{p_i} exists.

8. When p_k writes at line 2 of the Classifier algorithm, it must be the input value of the Classifier algorithm for p_k which is the value of the bag $ss_{\ell-1}$. Since we know the bag p_k wrote contains BL_{p_i}, it must be the case that $p_i \in addr_{\ell-1}$.

9. So, BL_{p_i,ss_ℓ,op_j} is the block with *maximal* seq value which is written by p_k at line 2 of the Classifier algorithm. Therefore, $BL_{p_i,ss_\ell,op_j} == BL_{p_i,ss_{\ell-1},op_k}$ and $p_i \in addr_{\ell-1}$ during op_k. (This proves part 1 of the lemma statement)

10. Process p_j executing Classifier at level ℓ reads the value written by p_k at level $\ell - 1$. It must be the case that the write operation happens before the read operation. Also, after reading the value, process p_j set BL_{p_i,ss_ℓ,op_j}. So, p_k writes the value at BL_{p_i,ss_ℓ,op_j} before p_j set BL_{p_i,ss_ℓ,op_j}. From the lemma statement, we know that p_j set BL_{p_i,ss_ℓ,op_j} before p_i starts performing any further operation after op_i. Thus, p_k writes at line 2 of the Classifier algorithm, before p_i starts any further operation after op_i. Also, p_k sets its $ss_{\ell-1}$ before it writes at line 2 of the Classifier algorithm. Thus, p_k sets its $ss_{\ell-1}$ before p_i starts any further operation after op_i. (This proves part 2 of the lemma statement)

□

In the next lemma, we prove that the address of the process executing the current operation is present in the address list returned from the Classifier call at each level during the execution of the operation. In other words, we are confirming that any process that performs an operation in the system can see itself in the list of active processes reported by the Classifier. This is one of the fundamental properties that is common among all the operations and is used in various lemmas. For example, this fact will allow us to apply Lemma 7.1 in the proof of Lemma 7.4.

Lemma 9.5. *Consider an arbitrary process p_i, and let op_i be an operation executed by process p_i. We prove that during op_i, for any $\ell \geq 0$, at line PREC, $p_i \in addr_\ell$.*

Proof. Let P(ℓ) = "for any level ℓ, at line *PREC*, $p_i \in addr_{\ell-1}$". We will use a proof by induction on $\ell \geq 1$.

Base Case: ($\ell == 1$)

1. $addr_0$ is created at line *ADDR0INIT* where the value is BL_{p_i,S,op_i}.active.

2. From the system definition (from Section: 3.1), we know when a block for a process p_i is initialized in the shared memory S by the system for the first time, p_i is automatically added to the active list property of that block. Since no item is ever removed from active list, $p_i \in BL_{p_i,S,op_i}$.active.

3. Therefore, $p_i \in addr_0$. And, since $addr_0$ remains unchanged until *PREC* at level 1, $p_i \in addr_0$ at line *PREC* for level 1 (base case proved)

Induction Hypothesis: For some $\ell > 1$, assume that, $P(1) \wedge ... \wedge P(\ell - 1)$ is true.

Induction Steps:

1. Since $P(\ell - 1)$ is true, at level $\ell - 1$, at line *PREC*, $p_i \in addr_{\ell-2}$. We need to show that at level ℓ, at line *PREC*, $p_i \in addr_{\ell-1}$.
2. At level $\ell - 1$: at line *PREC*, $p_i \in addr_{\ell-2}$. The input to the Classifier algorithm is $addr_{\ell-2}$. Since $p_i \in addr_{\ell-2}$, from Lemma 9.1, we know, $p_i \in addr_{\ell-1}$ at line *POSTC*.
3. So, at the end of level $\ell - 1$, $p_i \in addr_{\ell-1}$ at line *POSTC*.
4. At the next iteration of the loop when level is ℓ, at line *PREC*, $addr_{\ell-1}$ remains unchanged. Therefore, $p_i \in addr_{\ell-1}$ at line *POSTC* at line *PREC* at level ℓ.

□

The next lemma can be viewed as a generalization of Lemma 9.3. Again, we are considering the situation where a process p_i writes a block into the shared memory S during an operation op_i, and considering what version of p_i's block can be seen by any process p_j. In Lemma 9.3, we considered what p_j could see when reading p_i's block directly from shared memory S, but in the next lemma, we instead consider what p_j could see at any time during the execution of op_j before p_i starts another operation after op_i. In particular, for each call to Classifier in op_j that occurs before p_i starts its next operation, Classifier will return a bag containing block p_i, and we show that this block is at least as old as what p_i wrote to shared memory S during op_i. This lemma is used when proving the inductive step of Lemma 9.9.

Lemma 9.6. *Consider an arbitrary process p_i, and let op_i be an operation executed by process p_i. Let the seq value written by p_i in the shared memory be seq_{p_i} at line SWRITE.*

Let p_j be any process in procList other than p_i. Let op_j be any operation where the line POSTC for level ℓ is executed before the next operation by p_i.

We prove that, at line POSTC in op_j, $BL_{p_i,ss_\ell,op_j}.seq \leq seq_{p_i}$.

Proof. If $p_i \notin addr_\ell$ during op_j then, BL_{p_i,ss_ℓ,op_j} does not exist. Therefore, BL_{p_i,ss_ℓ,op_j}.seq is null or zero. Since seq_{p_i} is greater than or equal to 1, BL_{p_i,ss_ℓ,op_j}.seq $\leq seq_{p_i}$ is proved. From now on, we will consider the case where $p_i \in addr_\ell$ during op_j.

Let $P(\ell, p_j)$ = "BL_{p_i,ss_ℓ,op_j}.seq $\leq seq_{p_i}$ at line *PREC* ". We use a proof by induction on the level $\ell \geq 0$.

Base Case:

1. Consider $\ell = 0$.

2. During any op_j, at line *PREC* (just before the first call to the Classifier), BL_{p_i,ss_0,op_j}.seq $\leq seq_{p_i}$ (Lemma 9.3) (Proved base case).

Induction Hypothesis: For some $\ell > 1$, assume that, $P(1, p_j) \wedge ... \wedge P(\ell - 1, p_j)$ is true for all p_j that set $BL_{p_i,ss_{\ell-1},op_j}$ before p_i starts performing any further operation after op_i.

Inductive Step:

1. Consider a process p_j that set BL_{p_i,ss_ℓ,op_j} before p_i starts performing any further operation after op_i.

2. From Lemma 9.4 (Part: 1), we know, for each block p_i in ss_ℓ, there exists a process p_k where BL_{p_i,ss_ℓ,op_j}.seq $==$ $BL_{p_i,ss_{\ell-1},op_k}$.seq (because of BL_{p_i,ss_ℓ,op_j} $==$ $BL_{p_i,ss_{\ell-1},op_k}$) and from Lemma 9.4 (Part: 2) that p_k sets its $ss_{\ell-1}$ before p_i starts any further operation after op_i.

3. We apply the induction hypothesis to p_k at level $\ell - 1$ – and so $P(\ell - 1, p_k)$ is true. By the definition of $P(\ell-1, p_k)$, this means that $BL_{p_i,ss_{\ell-1},op_k}.seq \leq seq_{p_i}$. From the previous step, we know that $BL_{p_i,ss_{\ell-1},op_k}.seq = BL_{p_i,ss_\ell,op_j}.seq$, and so $BL_{p_i,ss_\ell,op_j}.seq \leq seq_{p_i}$, which proves that $P(\ell, p_j)$ is true. (induction proved)

Concluding Statement:

1. Consider a p_j that executes line *POSTC* for any level ℓ before the start of the next operation p_i. This means that this p_j must have set BL_{p_i,ss_ℓ,op_j} (at line *POSTC*) before p_i starts performing any further operation after op_i.

2. From the induction proof, we know, for any $\ell > 1$, $P(\ell, p_j)$ is true for all p_j that set BL_{p_i,ss_ℓ,op_j} before p_i starts performing any further operation after op_i. Therefore, for any $\ell > 1$, $P(\ell, p_j)$ is true.

□

In the next lemma, we prove that if a process p_i passes a bag into Classifier such that its own version of block p_i has the most up-to-date seq value in the Container, then Classifier will output a bag containing that same version of block p_i. This lemma is used when proving Lemma 9.9.

Lemma 9.7. *Consider an arbitrary process p_i, and let op_i be an operation executed by process p_i. Let p_j be any other process in procList other than p_i. Let op_j be an operation executed by p_j.*

We prove that during op_i, at line PREC, for level ℓ, if $p_i \in addr_{\ell-1}$ and $BL_{p_i,ss_{\ell-1},op_i}.seq \geq BL_{p_i,ss_{\ell-1},op_j}.seq$ for all op_j then we prove that, at line POSTC, $p_i \in addr_\ell$ and $BL_{p_i,ss_\ell,op_i} == BL_{p_i,ss_{\ell-1},op_i}$

Proof. During op_i, since $p_i \in addr_{\ell-1}$, $p_i \in addr_\ell$ (from Lemma 9.2) (proved part 1 of the proof statement). In the following steps we will focus on proving: BL_{p_i,ss_ℓ,op_i} $== BL_{p_i,ss_{\ell-1},op_i}$

1. At line 2 of the Classifier algorithm, all processes p_j write their own $ss_{\ell-1}$ to shared memory Container C at their respective address p_j. Process p_i also writes its own $ss_{\ell-1}$ to shared memory Container C at address p_i. We know from the lemma statement, $BL_{p_i,ss_{\ell-1},op_i}$.seq $\geq BL_{p_i,ss_{\ell-1},op_j}$.seq. Therefore, when p_i finishes executing line 2 and goes to line 3, the BL_{p_i}.seq in all bags in Container C is less than or equal to $BL_{p_i,ss_{\ell-1},op_i}$.seq. We consider two cases based on whether Classifier returns at line 9 or line 11.

2. Winner Case (returned at line 9):

(a) For p_i to reach line 9, the previous lines must have been executed.

(b) In line 8, a bag is returned from the *GetMergeRecent* algorithm. Now, we take a look inside this algorithm.

i. The *GetMergeRecent* algorithm calculates the merge based on the localContainer variable which is a copy of the Container C at the current node and returns a merged bag. (Algorithm: 8)

ii. In the *GetMergeRecent* algorithm, the bag at each address j in the localContainer is retrieved. The merged bag is calculated based on these retrieved bags.

iii. When p_i executes line 5, – it reads the bag at each address j, written by process p_j. These bags were written by some process p_j. The only place where these bags are written to the Container C is at line 2 of the Classifier algorithm. Also, we know (from the first point of this proof above), when p_i finishes executing line 2 of the Classifier algorithm, the BL_{p_i}.seq in all bags in Container C is less than or equal to $BL_{p_i,ss_{\ell-1},op_i}$.seq. So, when p_i reads the bags from localContainer (which is a copy of Container C) at line 5 of the *GetMergeRecent* algorithm, for each bag, BL_{p_i}.seq is less than or equal to $BL_{p_i,ss_{\ell-1},op_i}$.seq.

iv. From the definition of merge (3.2.3), each block in the result of the merge operation is the block with maximal seq value. Since, at block BL_{p_i} the maximal seq value is $BL_{p_i,ss_{\ell-1},op_i}$.seq, in the merge result, BL_{p_i} will be the same as $BL_{p_i,ss_{\ell-1},op_i}$. This merge result is returned from the *GetMergeRecent* algorithm as the bag ss_ℓ.

(c) Therefore, in the result bag of the Classifier, the block BL_{p_i,ss_ℓ,op_i} is equal to the block $BL_{p_i,ss_{\ell-1},op_i}$.

3. Loser Case (returned at line 11): For op_i, Classifier will return the same bag as $ss_{\ell-1}$, thus, in the result bag of the Classifier, the block BL_{p_i,ss_ℓ,op_i} will equal to the block $BL_{p_i,ss_{\ell-1},op_i}$. (proved)

□

In the next lemma, we prove that the block that is written to the shared memory S during an operation is the same block read from the shared memory S at the same address. This lemma proves that after writing a block to the shared memory S during an operation, that block does not change when it is read again during that operation. This fact is later used in other lemmas to show that the block that is written in the shared memory is then later passed down to the Classifier tree algorithm after reading it from the shared memory.

Lemma 9.8. *Consider any process p_i, and let op_i be any operation that is being executed by p_i. Let $\beta\ell_{p_i}$ be the block written by p_i in the shared memory at line SWRITE.*

We prove, at line PREC, $BL_{p_i,ss_0,op_i} == \beta\ell_{p_i}$.

Proof.

1. From lemma statement, at line *SWRITE*, $BL_{p_i,S,op_i} == \beta\ell_{p_i}$.
2. $\beta\ell_{p_i}$ is read at line *SREAD1*. During op_i between line *SWRITE* and line *PRECFirst*, $\beta\ell_{p_i}$ does not change since it is a local variable.
3. BL_{p_i,ss_0,op_i} is a block in the bag ss_0 which is created during op_i after reading from the shared memory at line *SREAD2*. At line *SREAD2*, BL_{p_i,ss_0} is created from $BL_{p_i,S}$. Note that, here BL_{p_i,ss_0} exists because of Lemma 9.5.
4. So, for BL_{p_i,ss_0,op_i}.seq to have a different value than $\beta\ell_{p_i}$.seq, there must be an update in block $BL_{p_i,S}$.
5. Because of SWMR, only an operation by p_i can update $BL_{p_i,S}$ at line *SWRITE* and since, the operation op_i is already being executed by p_i, at line *SWRITE*, no other operation can update block $BL_{p_i,S}$.
6. Thus, $BL_{p_i,S}$ remains the same during the operation op_i. So, $BL_{p_i,ss_0,op_i} == \beta\ell_{p_i}$ at line *PREC*.

□

In the next lemma, an important connection is established between the block that is written in the shared memory S before the first call to Classifier and the block that

is returned from the call to Classifier at each level during the same operation. At a high level, when a process p_1 writes a block to shared memory S during some operation op_1, then this version of the block is the "freshest", and each call that p_1 makes to Classifier during op_1 will return a bag that contains this version. In addition, this lemma shows that the address of the process that is executing the current operation is present in the list of addresses that is returned from the Classifier call at each level. This lemma will ultimately be used to establish the connection between the block returned from the final call to the Classifier during the snap operation and the block that is written in the shared memory S at the beginning of the snap operation.

Lemma 9.9. *Consider any process p_1 and let op_1 be any operation that is being executed by p_1. Let the block value written at line SWRITE by p_1 in the shared memory be $\beta\ell_{p_1}$. We prove that, during op_1, for any level ℓ, when the execution of line POSTC completes, $BL_{p_1,ss_\ell,op_1} == \beta\ell_{p_1}$ and $p_1 \in addr_\ell$.*

Proof. Let $seq_{p_1} = \beta\ell_{p_1}$.seq. Let p_j be any process in procList other than p_1. Let op_j be any operation executed by p_j.

Let $P(\ell)$ be "$BL_{p_1,ss_\ell,op_1} == \beta\ell_{p_1}$ and $p_1 \in addr_\ell$". We prove that $P(\ell)$ is true for all $\ell \geq 1$ by induction on the level ℓ.

Base Case:

1. Consider $\ell = 1$.

2. During any op_j, at line *PREC* (just before the first call to the Classifier), BL_{p_1,ss_0,op_j}.seq $\leq seq_{p_1}$. This reason for this is as follows:

   (a) If op_j did not reach line *PREC* (just before the first call to the Classifier) during op_i, then BL_{p_1,ss_0,op_j} is null which means BL_{p_1,ss_0,op_j}.seq $= 0$. Since the seq value written in the shared memory S is always greater than 0, the inequality BL_{p_1,ss_0,op_j}.seq $\leq seq_{p_1}$ holds.

   (b) If op_j reached line *PREC* (just before the first call to the Classifier) during op_i, then it also means that op_j reached line *PREC* before the next operation by p_i. From Lemma 9.3, we know that BL_{p_1,ss_0,op_j}.seq $\leq seq_{p_1}$.

3. For op_1, at line *PREC*, $BL_{p_1,ss_0,op_1} == \beta\ell_{p_1}$ (Lemma 9.8). Here, BL_{p_1,ss_0,op_1} exists because of Lemma 9.5.

4. At line *PREC* for any op_j, BL_{p_1,ss_0,op_j}.seq $\leq seq_{p_1}$ (from Step 2). This means when the first call to the Classifier is made by p_1 at line *PREC* for level $\ell = 1$, BL_{p_1,ss_0,op_j}.seq $\leq seq_{p_1}$. For op_1, BL_{p_1,ss_0,op_1}.seq $== seq_{p_1}$. So, BL_{p_1,ss_0,op_j}.seq $\leq BL_{p_1,ss_0,op_1}$.seq for all op_j.

5. So, for p_1, at the start of the Classifier algorithm at level $\ell = 1$, BL_{p_1,ss_0,op_1}.seq $\geq BL_{p_1,ss_0,op_j}$.seq for all op_j. (Here, BL_{p_1,ss_0,op_1} exists because of Lemma 9.5) We know if this is the case, then for p_1, the block BL_{p_1} in the result of the Classifier will be equal to the block BL_{p_1,ss_0,op_1}.seq. (Lemma 9.7)

6. The result of the Classifier is received at line *POSTC*, at bag ss_1. So, $BL_{p_1,ss_1,op_1} == BL_{p_1,ss_0,op_1} == \beta\ell_{p_1}$ and from Lemma 9.8 we conclude that $p_1 \in addr_1$. (Proved base case)

Induction Hypothesis: For some $\ell > 1$, assume that $P(1) \wedge ... \wedge P(\ell - 1)$ is true.
Inductive Step: We will prove that $P(\ell)$ is true.

1. We know at level ℓ, for all op_j at line *POSTC*, BL_{p_1,ss_ℓ,op_j}.seq $\leq seq_{p_1}$. This reason for this is as follows:

   (a) If op_j did not reach line *POSTC* for level ℓ during op_i, then BL_{p_1,ss_ℓ,op_j} is null which means BL_{p_1,ss_ℓ,op_j}.seq $= 0$. Since the seq value written in the shared memory S is always greater than 0, the inequality BL_{p_1,ss_ℓ,op_j}.seq $\leq seq_{p_1}$ holds.

   (b) If op_j reached line *POSTC* during op_i, then it also means that op_j reached line *POSTC* before the next operation by p_i. From Lemma 9.6, we know that BL_{p_1,ss_ℓ,op_j}.seq $\leq seq_{p_1}$.

2. Since, $P(\ell - 1)$ is true, we know at level $\ell - 1$, for op_1, $BL_{p_1,ss_{\ell-1},op_1} == \beta\ell_{p_1}$.

3. For p_1, at the start of the Classifier algorithm at level ℓ, $BL_{p_1,ss_{\ell-1},op_1}$ is passed as the input bag to the Classifier algorithm.

4. So, for p_1, at the start of the Classifier algorithm at level ℓ, $BL_{p_1,ss_{\ell-1},op_1}$.seq $\geq$ $BL_{p_1,ss_{\ell-1},op_j}$.seq for all op_j (Here, $BL_{p_1,ss_{\ell-1},op_1}$ exists because of Lemma 9.5). We know if this is the case, then for p_1, the block BL_{p_1} in the result of the Classifier will be equal to the block $BL_{p_1,ss_{\ell-1},op_1}$. (Lemma 9.7)

5. The result of the Classifier is received at line *POSTC*, at bag ss_ℓ. So, BL_{p_1,ss_ℓ,op_1} $==$ $BL_{p_1,ss_{\ell-1},op_1}$ $==$ $\beta\ell_{p_1}$ and from Lemma 9.8 we conclude that $p_1 \in addr_\ell$.

□

In the next lemma, we prove that during the execution of the Classifier algorithm by any process, the updated active list at line 4 will always contain the initial processes present in the system. This finding regarding the updated active list is used to compare the results of Classifier algorithm between different operations.

Lemma 9.10. *Let $p_x \in procList_\alpha$. Let op_s be any operation executed by p_s. We prove that during op_s at any level ℓ, immediately after line 4 of the Classifier algorithm is executed, $p_x \in updatedActiveList_{op_s}$.*

Proof. 1. From the definition of $procList_\alpha$ (Table 3.3), we know that these processes are made available to each block's active list by the system in shared memory S. Therefore, during op_s, p_x will be available in BL_{p_s,S,op_s}.active.

2. We know from Lemma 9.9, BL_{p_s,S,op_s} $==$ BL_{p_s,ss_ℓ,op_s} and $p_s \in addr_\ell$ for any level ℓ.

3. Therefore, at level $\ell - 1$, BL_{p_s,S,op_s} $==$ $BL_{p_s,ss_{\ell-1},op_s}$ and $p_s \in addr_{\ell-1}$.

4. $ss_{\ell-1}$ and $addr_{\ell-1}$ are the inputs to the Classifier algorithm at level ℓ.

5. At level ℓ, in the Classifier algorithm, $ss_{\ell-1}$ is written in the Container C at address p_s.

6. $addr_{\ell-1}$ is passed to the GetUpdatedActiveList algorithm, and since $p_s \in addr_{\ell-1}$, eventually, at line 7, j $= p_s$ and from Container C, $ss_{\ell-1}$ will be read which was just written at the beginning of the Classifier algorithm.

7. Eventually, at line 11, k = p_s and $BL_{p_s,ss_{\ell-1}}$ will be read at line 12. From step (3), this block is equal to BL_{p_s,S,op_s}. From step (1), we know, p_x is available in BL_{p_s,S,op_s}.active.

8. Therefore, at line 14, $address_{p_j}$ will eventually be p_x and it will be added to the updatedActiveList variable.

9. Since, the items from updatedActiveList are never removed, the result of the GetUpdatedActiveList algorithm will contain p_x.

10. Therefore, $p_x \in updatedActiveList_{op_s}$.

□

In the next lemma, we prove that if a modify operation op_i is linearized before a snap operation op_s, then the process that performed op_i will appear in the address list returned by the final call to Classifier during op_s. In other words, a snap operation will always find out about the existence of any process that performs an operation before the snap.

Lemma 9.11. *Consider any process p_s and let op_s be a snap operation executed by p_s. Let ss_{final} be the bag returned by the final call to Classifier during op_s. Let op_i be any modify (add, remove or update) operation executed by p_i. If op_i is linearized before op_s, then $p_i \in addr(ss_{final})$ during op_s.*

Proof.

1. We use a proof by contradiction. Assume that $p_i \notin addr(ss_{final})$. This means BL_{p_i,ss_{final},op_s}.seq == 0.

2. When p_i executes op_i, because of SWMR property, BL_{p_i,S,op_i} always exists. Also, during op_i, the seq value is incremented by 1 at the beginning and it is written back to the shared memory S. The seq value is never decreased anywhere in the code. Therefore, BL_{p_i,S,op_i}.seq ≥ 1 (since the initial default value is 0).

3. We know from Lemma 9.9, $BL_{p_i,S,op_i} == BL_{p_i,ss_{final},op_i}$ during op_i.

4. Therefore, BL_{p_i,ss_{final},op_i}.seq ≥ 1.

5. From the previous step, it follows that BL_{p_i,ss_{final},op_i}.seq $> BL_{p_i,ss_{final},op_s}$.seq.

6. In the lemma statement, we assumed that op_i is linearized before op_s. If this is the case then, from line 14 of the Linearization Definition Algorithm (10), we know that BL_{p_i,ss_{final},op_i}.seq $\leq BL_{p_i,ss_{final},op_s}$.seq. This condition contradicts with the findings in the previous step.

□

In the next lemma, we prove that if an add operation is partially ordered before the current operation, then the process that executed the add operation will be present in the updated active list immediately after the execution of line 4 in the first call to the Classifier algorithm during the current operation.

Lemma 9.12. *Consider any process p_i and let op_i be any operation executed by p_i. Let op_k be an add operation executed by p_k where $op_k \rightarrow op_i$ and $p_k \notin procList_\alpha$. We prove that, during op_i, $p_k \in updatedActiveList_{op_i}$ immediately after the execution of line 4 of the Classifier algorithm when called at the root node of the Classifier Tree.*

Proof. 1. Since, $p_k \notin procList_\alpha$, from Observation 3.3.3, we know $AC(p_k)$ exists and also, $add_1(p_0, p_1) \rightarrow add_2(p_1, p_2) \rightarrow add_3(p_2, p_3) \rightarrow$ ...... $\rightarrow add_{k-1}(p_{k-2}, p_{k-1})$ $\rightarrow add_k(p_{k-1}, p_k)$.

2. Since the system only allows a process to perform an operation after the add operation that added the process is completed, $add_k(p_{k-1}, p_k) \rightarrow op_k$.

3. Therefore, the final partial ordering: $add_1(p_0, p_1) \rightarrow add_2(p_1, p_2) \rightarrow add_3(p_2, p_3)$ $\rightarrow$ ...... $\rightarrow add_{k-1}(p_{k-2}, p_{k-1}) \rightarrow add_k(p_{k-1}, p_k) \rightarrow op_k \rightarrow op_i$.

Consider any arbitrary $p_x \in ancestor(p_k)$. Let P(x) = "$p_x \in updatedActiveList_{op_i}$ at line 4 of the Classifier algorithm in the root node". We prove that P(x) is true by induction on x, where x > 0 and x $\leq len_{ancestor(p_k)}$ and p_x is the process that performed the xth operation in AC(p_k)

Base Case: x = 1

1. Consider the case when p_x is the first operation in the chain $AC(p_k)$. So, $p_x = p_1$. From Observation 3.3.3, we know, $p_1 \in procList_\alpha$.

2. From Lemma 9.10, we know, during op_i, if $p_1 \in procList_\alpha$ then at any level ℓ, $p_1 \in updatedActiveList_{op_i}$ at line 4 of the Classifier algorithm.(Base case proved)

Induction Hypothesis: For some x such that x > 1 and x < $len_{ancestor(p_k)}$, assume that $P(1) \wedge ... \wedge P(x-1)$ is true.

Inductive Step:

1. By the Induction Hypothesis, we know $p_{x-1} \in updatedActiveList_{op_i}$ at the end of execution of line 4 of the Classifier during op_i.

2. Since $p_{x-1} \in updatedActiveList_{op_i}$, eventually the active list of p_{x-1} will be traversed at line 14 of the GetUpdatedActiveList.

3. Since $add_x(p_{x-1}, p_x) \rightarrow op_i$, p_x will be found in the active list of p_{x-1} when it is traversed at line 14.

4. Since no items are removed from updatedActiveList in the GetUpdatedActiveList algorithm, the result returned from the GetUpdatedActiveList will contain p_x.

5. Therefore, $p_x \in updatedActiveList_{op_i}$. (Inductive step proved)

Concluding Statement: In the induction above, when x = $len_{ancestor(p_k)}$, $p_x = p_k$. Therefore, $p_k \in updatedActiveList_{op_i}$ at line 4 of the root node in the Classifier Tree. □

In the next lemma, we prove that at line 4 in the execution of Classifier, if the process that is executing the current operation is in the updated active list, then the process that added the current process in the system will also be in the updated active list.

Lemma 9.13. *Consider an arbitrary process p_i, and let op_i be an operation executed by process p_i. If $p_x \in updatedActiveList_{op_i}$ at any level ℓ, then $p_{x-1} \in updatedActiveList_{op_i}$ at level ℓ, where p_{x-1} added p_x.*

Proof. We will consider the following 2 cases.

Case 1: $p_x = p_i$

(a) Since p_x is executing the current operation op_i, the add operation (executed by p_{x-1}) that added p_x had already been completed before op_i started.

(b) Since the add operation (executed by p_{x-1}) is completed before op_i starts, the add operation that added p_{x-1} must already had ended before op_i started. From Lemma 9.12, we know $p_{x-1} \in updatedActiveList_{op_i}$ at line 4 in the root node of the Classifier during op_i.

Case 2: p_x != p_i: We consider the following two subcases:

Subcase a: During the call to the GetUpdatedActiveList, p_x is not in oldActive:

i. Since p_x is in the result of the GetUpdatedActiveList and not in oldActive, it must have been found in this call to the GetUpdatedActiveList.

ii. updatedActiveList is the result of the GetUpdatedActiveList. p_x is added to updatedActiveList at line 22.

iii. p_x is found in the active list of r_k at line 14. Since p_x can be found is in the active list of p_{x-1}, r_k is p_{x-1}.

iv. r_k is initialized at line 12 and, it is found from updatedActiveList at line 11. Therefore, p_{x-1} already exists in the updatedActiveList.

v. Since no processes are removed from updatedActiveList, the result of the GetUpdatedActiveList will contain p_{x-1}.

vi. At line 4 of the Classifier algorithm, $p_{x-1} \in updatedActiveList_{op_i}$.

Subcase b: During the call to the GetUpdatedActiveList, p_x is in oldActive:

i. oldActive is the value from the input $addr_\ell$ to the Classifier algorithm. Therefore, p_x must have been found at some level above ℓ where ℓ is the current level. We consider the following two cases:

Case b.1: when $p_i = p_{x-1}$

Case b.2: when p_i != p_{x-1}

ii. Case b.1: when $p_i = p_{x-1}$:

A. Since, p_{x-1} added p_x, p_x exists in the active list of $BL_{p_{x-1},S}$ (where S is the shared memory S). Since in this case op_i is being executed by p_{x-1}, p_x will be found before the first call to the Classifier.

B. So, p_x will be in the $addr_1$ in the root node. Since, p_{x-1} is executing the current operation, p_{x-1} will also be in the $addr_1$ in the root node.

C. We know (from Lemma 9.2) once a process $p_u \in addr_\ell$ in the Classifier algorithm at level ℓ, $p_u \in addr_{\ell_d}$ where $\ell_d >$ l. Therefore, $p_{x-1} \in addr_\ell$ at level ℓ.

D. Since p_{x-1} is in the input to the GetUpdatedActiveList and nothing is removed from updatedActiveList variable in GetUpdatedActiveList algorithm, at line 4 of the Classifier Tree, $p_{x-1} \in updatedActiveList_{op_i}$ during op_i.

iii. Case b.2: when p_i != p_{x-1}:

A. When the first call to the Classifier starts, since p_i != p_{x-1}, $p_x \notin addr_1$ at the root node.

B. However, p_x is discovered at some level above ℓ.

C. Therefore, $p_x \in updatedActiveList_{op_i}$ at some level $\ell_u < \ell$ where during the call to GetUpdatedActiveList, p_x was

not in oldActive.

D. We showed in Case 2(b)ii above that if "During the call to the GetUpdatedActiveList, p_x is not in oldActive" then, $p_{x-1} \in updatedActiveList_{op_i}$. $updatedActiveList_{op_i}$ is the input $addr_{\ell_u+1}$ at level $\ell_u + 1$. Therefore, $p_{x-1} \in addr_{\ell_u+1}$.

E. Since we know (from Lemma 9.2) once a process $p_u \in addr_\ell$ in the Classifier algorithm at level ℓ, $p_u \in addr_{\ell_d}$ where $\ell_d > \ell$, $p_{x-1} \in addr_\ell$ at level ℓ.

F. Since p_{x-1} is in the input to the GetUpdatedActiveList and nothing is removed from updatedActiveList variable in GetUpdatedActiveList algorithm, at line 4 of the Classifier Tree, $p_{x-1} \in updatedActiveList_{op_i}$ during op_i.

□

In the next lemma, we prove that for each process (that is not an initial process) at line 4 of the Classifier algorithm, the ancestor list of that process is present in the updated active list.

Lemma 9.14. *Let $updatedActiveList_{op_i}$ be the updatedActiveList variable at line 4 in the Classifier Tree during any operation op_i. Let an arbitrary process be p_k which exists in $updatedActiveList_{op_i}$ and $p_k \notin procList_\alpha$. Since $p_k \notin procList_\alpha$, from the Observation 3.3.3, we know $ancestor(p_k)$ exists.*

We prove that if $p_k \in updatedActiveList_{op_i}$, $ancestor(p_k) \subseteq updatedActiveList_{op_i}$.

Proof. Consider any arbitrary $p_x \in ancestor(p_k)$.
Let P(x) = "$p_x \in updatedActiveList_{op_i}$". We prove that P(x) is true by induction on x, where x > 0 and x $\leq len_{ancestor(p_k)}$

Base Case: x = $len_{ancestor(p_k)}$

1. The last process in ancestor(p_k) is the process p_k.
2. From Lemma statement, we know $p_k \in updatedActiveList_{op_i}$. (base case proved)

Induction Hypothesis: For some x such that x > 1 and x < $len_{ancestor(p_k)}$, assume that P(x+1) is true.

Inductive Step:

1. From the Induction Hypothesis, we know that P(x+1) is true, which means, $p_{x+1} \in updatedActiveList_{op_i}$. From Lemma 9.13, we know that if $p_{x+1} \in updatedActiveList_{op_i}$ then, $p_x \in updatedActiveList_{op_i}$.

Concluding statement: Since for any arbitrary $p_x \in ancestor(p_k)$, we showed that p_x is also $\in updatedActiveList_{op_i}$, $ancestor(p_k) \subseteq updatedActiveList_{op_i}$. □

In the next lemma, we show that if a process p_i exists in the input address list of the Classifier algorithm then, the seq value at block p_i in the output bag of the Classifier algorithm will be at least as big as it was in the input bag of the Classifier. Note that the inputs of the Classifier algorithm are the outputs of the execution of Classifier at the previous level. The information proved in this lemma creates a connection between the outputs of different levels in the Classifier algorithm which is then used to connect with seq values in the shared memory S.

Lemma 9.15. *Consider arbitrary processes p_i and p_s, and let op_s be a snap operation executed by process p_s. When p_s executes line POSTC of the snap algorithm at any level ℓ, we prove that, if $p_i \in addr_{\ell-1}$ during op_s, $BL_{p_i,ss_{\ell-1},op_s}.seq \leq BL_{p_i,ss_\ell,op_s}.seq$.*

Proof.

1. At level ℓ, $BL_{p_i,ss_{\ell-1},op_s}$.seq is part of the input to Classifier algorithm and BL_{p_i,ss_ℓ,op_s}.seq is part of the output of the Classifier algorithm. We know BL_{p_i} exists in $ss_{\ell-1}$ because $p_i \in addr_{\ell-1}$. Also, since $p_i \in addr_{\ell-1}$, $p_i \in addr_{\ell_d}$ where $\ell_d > \ell$. (From Lemma 9.2)
2. Therefore, BL_{p_i} exists in ss_ℓ as well.

3. At line 2 of the Classifier algorithm, the input of the Classifier is written into the shared memory Container C. Since p_s is executing the current operation, the input bag will be written at address p_s.

4. The GetUpdatedActiveList algorithm is called at line 4. Since $addr_{\ell-1}$ is passed as parameter, $p_i \in$ oldActive. Also, since p_s is executing the current operation, p_s will also be in oldActive.

5. In the GetUpdatedActiveList, oldActive is assigned to updatedActiveList and no item is removed from updatedActiveList, therefore, at the end of the GetUpdatedActiveList algorithm, p_i and p_s will both be in result list.

6. At the end of the execution at line 4 in Classifier Tree, *updatedActiveList* will include process p_s and p_i.

7. There are two lines from where the output is returned from the Classifier algorithm. If the Classifier algorithm returns from line 9:

   (a) The bag that is returned from the Classifier algorithm at line 9 is the bag calculated in the $GetMergeRecent$ algorithm at line 8.

   (b) In the $GetMergeRecent$ algorithm, the *updatedActiveList* is passed as input parameter which includes the process p_s and p_i.

   (c) Since the input variable in the GetMergeRecent algorithm activeList contains p_i and p_s and the localContainer is a copy of the Container at the current node, at line 5 of the GetMergeRecent algorithm, the bag that is written by p_s at line 2 of the Classifier algorithm will be read. Also, at line 7 of the GetMergeRecent algorithm, the block at address p_i will be read from the bag that is read at line 5. Since, p_i and p_s are both in *activeList*, eventually the block BL_{p_i} from the input of the Classifier algorithm will be read at line 7 and the corresponding seq value will be considered to calculate the merge result.

   (d) From the definition of merge (3.2.3), each block in the result of the merge operation is the block with maximal seq value. So, the seq value in the

block BL_{p_i} in the merge result will be at least as big as the seq value in the block BL_{p_i} in the input of the Classifier. This merge result is finally returned from the Classifier as the output.

(e) The input of the Classifier algorithm is $BL_{p_i,ss_{\ell-1},op_s}$ and the output of the Classifier algorithm is BL_{p_i,ss_ℓ,op_s}.

(f) Therefore, $BL_{p_i,ss_{\ell-1},op_s}$.seq $\leq BL_{p_i,ss_\ell,op_s}$.seq

8. If the Classifier algorithm returns from line 11, then the output of the Classifier is same as the input. Therefore, $BL_{p_i,ss_{\ell-1},op_s}$.seq $==$ BL_{p_i,ss_ℓ,op_s}.seq.

□

In the next lemma, we show that if two operations follow partial ordering then if we consider the two leaves where these operations ended up in the Classifier Tree, then at the least common ancestor (LCA) of these two leaves, the list of discovered processes during the operation that is partially ordered first will be a subset (or equal) of the list of discovered processes during the other operation. This subset relation is used to establish a connection between two operations' discovered processes.

Lemma 9.16. *Let op_i and op_s be any two operations performed by processes p_i and p_s where $op_i \rightarrow op_s$. We prove that at the LCA(op_i, op_s) node in the Classifier Tree, at line 4 of the Classifier algorithm, $updatedActiveList_{op_i} \subseteq updatedActiveList_{op_s}$.*

Proof. Consider any process $p_x \in updatedActiveList_{op_i}$. Consider two scenarios:

Scenario 1: $p_x \in procList_\alpha$

(a) Let the LCA(op_i, op_s) node in the Classifier Tree be at level a.

(b) From Lemma 9.10, we know, if $p_x \in procList_\alpha$ then at any level ℓ, $p_x \in updatedActiveList_{op_s}$ at line 4 of the Classifier algorithm.

(c) Therefore, $p_x \in updatedActiveList_{op_s}$ at level a.

(d) Since for any arbitrary $p_x \in updatedActiveList_{op_i}$, we showed that p_x is also $\in updatedActiveList_{op_s}$, $updatedActiveList_{op_i} \subseteq updatedActiveList_{op_s}$.

Scenario 2: $p_x \notin procList_\alpha$

(a) From the Observations in 3.3.3, we know the following exist:

i. $AC(p_x)$ is the chain of add operations that lead to the addition of p_x in the system.

ii. $ancestor(p_x)$ is the list of processes involved in $AC(p_x)$ including p_x.

(b) Let P(k) = "if $p_x \in updatedActiveList_{op_i}$, then $p_k \in updatedActiveList_{op_s}$". We will use a proof by induction to show that P(k) is true for each $p_k \in ancestor(p_x)$, i.e., for $0 <$ k $\leq len_{AC}$.

Base Case (k = 1):

i. From the definition of $AC(p_x)$, we know $p_1 \in procList_\alpha$.

ii. From Lemma 9.10, we know, if $p_1 \in procList_\alpha$ then at any level ℓ, $p_1 \in updatedActiveList_{op_s}$ at line 4 of the Classifier algorithm. (Base case proved)

Induction Hypothesis: Assume that, for some $1 <$ k $\leq len_{AC}$, $P(1) \wedge$ ... $\wedge P(k-1)$ is true.

Inductive Step:

i. During op_i, since $p_x \in updatedActiveList_{op_i}$ (from the definition of P(k)), we know from Lemma 9.14 that $ancestor(p_x)$ $\subseteq updatedActiveList_{op_i}$. Therefore, p_k was found at (if not before) line 4 of the Classifier algorithm during op_i. In other words, p_k was found before the end of op_i.

ii. Since $op_i \rightarrow op_s$, p_k was found before the start of op_s.

iii. p_{k-1} added p_k in the system, so p_{k-1} added p_k in the system before the start of op_s.

iv. From the Induction Hypothesis, we know $p_{k-1} \in updatedActiveList_{op_s}$.

v. Since, $p_{k-1} \in updatedActiveList_{op_s}$, eventually line 14 of GetUpdatedActiveList will be executed where the active list of p_{k-1} will be traversed, and, p_k will be found since p_{k-1} added p_k in the system before the start of op_s, and, once a process is added to active list it is never removed.

vi. p_k will be found in the active list of p_{k-1} at line 14 of the GetUpdatedActiveList algorithm and will be added to the updatedActiveList variable during op_s.

vii. Since, no items are removed from the updatedActiveList variable during GetUpdatedActiveList algorithm, p_k will be in the returned list from the GetUpdatedActiveList algorithm. Therefore, $p_k \in updatedActiveList_{op_s}$ at line 4 of the Classifier algorithm. (Inductive step proved)

Concluding Statement: When k = len_{AC}, $p_k = p_x$. From the proof above, we conclude that $p_x \in updatedActiveList_{op_s}$. Since for any arbitrary $p_x \in updatedActiveList_{op_i}$, we showed that p_x is also $\in updatedActiveList_{op_s}$, $updatedActiveList_{op_i} \subseteq updatedActiveList_{op_s}$.

□

In the next lemma, we show that if two operations follow partial ordering then if we consider the two leaves where these operations ended up in the Classifier Tree, then at the LCA node of these two leaves, the operation that is partially ordered first will go to the left child of the LCA node and the other operation will go to the right child of the LCA node. This relation between partial ordering and the direction of the operations is used in other lemmas to prove at which leaf the operations end up in the Classifier Tree.

Lemma 9.17. *Consider arbitrary processes p_i and p_s, and let op_s be an operation executed by process p_s. Let op_i be any operation executed by p_i where $op_i \rightarrow op_s$. We prove that during the execution of the Classifier algorithm at the LCA(op_i, op_s) node, for op_s and op_i, the code returns from line 9 and line 11 respectively.*

Proof.

1. Since $op_i \rightarrow op_s$, BL_{p_s,S,op_i}.seq $<$ BL_{p_s,S,op_s}.seq. Also, because of $op_i \rightarrow op_s$, during op_i, seq value in any BL_{p_s} will be less than BL_{p_s,S,op_s}.seq.

2. From Lemma 9.9, we know at any level ℓ during op_s, BL_{p_s,S,op_s}.seq == BL_{p_s,ss_ℓ,op_s}.seq. Therefore, during op_s, at any level ℓ, BL_{p_s,ss_ℓ,op_s}.seq is greater than the seq value at any BL_{p_s} read during op_i.

3. seqSum is calculated based on the updatedActiveList and the mergeRecent variables.

4. From Lemma 9.16 we know that $updatedActiveList_{op_i} \subseteq updatedActiveList_{op_s}$. Because of this and the fact that BL_{p_s,ss_ℓ,op_s}.seq is greater than the seq value at any BL_{p_s} read during op_i, $seqSum_{op_i} < seqSum_{op_s}$.

5. Since the LCA(op_i, op_s) is located at level a, from the definition of LCA we know that one of the operations from op_i, op_s goes right and the other one goes left. For the process that will go right, the seqSum will be greater than K. op_i cannot go right here because if $seqSum_{op_i}$ > K, $seqSum_{op_s}$ is also > K. Therefore, op_s goes right here and op_i goes left. In other words, at LCA(op_i, op_s), for op_s and op_i, the code returns from line 9 and line 11 respectively.

□

In the next lemma, we prove that if two operations follow partial ordering, then the operation that is partially ordered first ends up at a leaf node in the Classifier Tree which is to the left of the leaf node where the other operation ends up. This lemma is useful when proving results about our linearization ordering, as well as when providing the missing proofs from Attiya and Rachman [16] in the Section 9.2.

Lemma 9.18. *Consider arbitrary processes p_i and p_s. Let op_s be any operation executed by p_s. Let op_i be any operation executed by p_i where $op_i \rightarrow op_s$. Let op_i and*

op_s be two operations that terminate at Classifier Tree leaf nodes CLL_i and CLL_s respectively. We prove that CLL_i is to the left of CLL_s in the ordering of all leaf nodes from left to right.

Proof.

1. From Lemma 9.17, we know at LCA(op_i, op_s) node, for op_s and op_i, the code returns from line 9 and line 11 in the Classifier algorithm respectively.

2. From the Classifier Tree structure, it means at LCA(op_i, op_s) node, the next execution of the Classifier algorithm will be at the left child of LCA(op_i, op_s) node for op_i and at the right child of LCA(op_i, op_s) node for op_s.

3. Since the Classifier Tree is a binary tree structure, from the graph property, op_i will eventually end at a Classifier leaf node which is to the left of the leaf node where the op_s will end up at. Therefore, CLL_i is to the left of CLL_s in the ordering of all leaf nodes from left to right.

□

9.2 Lemmas from Attiya and Rachman [16]

In this section, we will primarily address the lemmas stated and proved in Attiya and Rachman [16]. We rewrite the most important lemma statements to match our terminology and notation, and we also provide the missing proofs.

First, we summarize the key similarities and differences between our thesis and the Attiya and Rachman [16] paper:

1. Our atomic snapshot object has the ability to add or remove processes while theirs does not.

2. Our "merge" operation has the same purpose as their "union" operation.

3. Our seqSum value has the same purpose as their "view size".

4. Due to the dynamic nature of our snapshot object, there is no fixed number of processes in the system, and our implementation does not have access to such a

value. Their implementation assumes that there is a fixed number of processes n, and this value is known to the algorithms.

5. Due to the static nature of their system, their implementation can loop through the registers using indices $1, \dots, n$. In our system, the set of processes and addresses can change over time, so we need to maintain a dynamic list of addresses that can be used when accessing blocks in a bag, or accessing bags in a Container.

In the next lemma, we consider any two operations performed by the same process that both reach the same node in the Classifier Tree during their executions, and we show that each bag in the Container during the later operation dominates the corresponding bag in the Container during the earlier operation.

Lemma 9.19. *Consider an arbitrary process p_i. Consider op_{curr} be the current operation that is being executed and op_{prev} be the previous operation executed by p_i and assume that they both reach the same node u in the Classifier Tree located at level ℓ. Let C_{prev} and C_{curr} be the states of Container at node u when the last $localContainer$ variable is initialized in the Classifier algorithm during the execution of op_{prev} and op_{curr} respectively.*

We prove that each bag at address p_x in C_{prev} is dominated by the bag at the same address p_x in C_{curr}.

Proof. We make a few observations that will be helpful for our proof:

1. Let $ss_{\ell-1,op_{prev}}$ and $ss_{\ell-1,op_{curr}}$ be the inputs to the Classifier algorithm at level ℓ during op_{prev} and op_{curr} respectively.

2. Since op_{prev} and op_{curr} are executed by the same process p_i, they follow partial ordering because of SWMR property. More specifically, op_{prev} is partially ordered before op_{curr}.

3. Since the Classifier graph is a binary tree, there exists exactly one path from the root to node u. Therefore, op_{curr} and op_{prev} must have visited the same set of nodes before reaching node u.

4. When a node is visited for the first time during the Classifier algorithm and writes a bag in the Container, that new bag dominates the existing bag. Since the node is being visited the first time, the blocks in that bag are all empty which means all the seq values of the block in that bag is zero. When a new bag is written by a process executing an operation, that bag is guaranteed to have at least one block (the block at the address of the current process) where the seq value is at least 1 because at the beginning of each operation the seq value is incremented. The seq values in all other blocks in the bag cannot be less than zero because the seq values never decrease. Therefore, from the definition of domination, the new bag dominates the existing empty bag.

We prove the lemma statement by induction on ℓ and t where the former is level of the node and latter is the number of operations that has visited node u. The value of t starts at 0 and each time an operation visits node u, t is incremented by 1. Let t_{curr} be the operation number when node u is visited during op_{curr}.

Let P(ℓ, t_{curr}) = "At any node u at level ℓ, when the number of visits to node u is t_{curr}, each bag at address p_x in C_{prev} is dominated by the bag at the same address p_x in C_{curr} at level ℓ at t_{curr}". We will use nested inductions to prove this predicate is true for all values of ℓ and t_{curr}. The outer induction will be the value of t_{curr} and the inner induction will be on the value of ℓ.

Outer Induction Base Case ($\ell \geq 1$ and $t_{curr} = 1$):

1. When $t_{curr} = 1$, the current node is being visited for the first time.

2. We know (from the observations above) when a node is visited for the first time, the input bag will dominate the existing bag in the same address.

Outer Induction Hypothesis: Assume $P(a, b)$ is true where the values of a and b are as follows:

1. $a = 1, ..., \lambda$ (where λ is the height of the Classifier Tree)

2. $b = 1, ..., t_{curr} - 1$ where $t_{curr} \geq 2$

Outer Induction Step: We prove $P(a, t_{curr})$ in the inner induction below.
Inner Induction Base Case ($\ell = 1$):

1. Since $\ell = 1$, node u is the root node. The input bag to the root node is ss_0 which is calculated at line *SREAD2* from the values of the shared memory S.

2. Since op_{prev} is partially ordered before op_{curr} (from the observations above), both operations are executed by the same process and no items are removed from the active list in the code, $addr_0$ of op_{prev} is a subset of $addr_0$ of op_{curr}.

3. Since op_{prev} is partially ordered before op_{curr} and seq values never decreases in the code, each seq value at address p_x ($\in addr_0$ in ss_0 during op_{prev}) is less than or equal to the seq value at address p_x ($\in addr_0$ in ss_0 during op_{curr}).

4. Combining the previous two steps, ss_0 of op_{curr} dominates ss_0 of op_{prev}.

5. ss_0 is the input to the root node. Since op_{curr} and op_{prev} are both executed by p_i, ss_0 is written down at address p_i in the Container C_{prev} and C_{curr} at root node during op_{prev} and op_{curr} respectively.

6. Since ss_0 of op_{curr} dominates ss_0 of op_{prev}, the bag at address p_i in C_{curr} dominates the bag at the same address in C_{prev}.

7. Now, we consider all other bags in the Container, i.e., consider any arbitrary address p_x in C_{curr}. We need to show the bag at address p_x in C_{curr} dominates the bag at same address in C_{prev}. Let these two bags be $BA_{p_x,C_{curr}}$ and $BA_{p_x,C_{prev}}$ respectively.

8. If $BA_{p_x,C_{curr}}$ and $BA_{p_x,C_{prev}}$ are equal or if $BA_{p_x,C_{curr}}$ is the first bag that is written at address p_x then, based on the edge cases below, $BA_{p_x,C_{curr}}$ dominates $BA_{p_x,C_{prev}}$.

   Edge cases:

(a) If during op_{curr}, no bag is written at p_x, then because of the partial ordering there exists no bag at p_x during op_{prev}. Therefore, the domination property holds since both seq values are 0.

(b) If during op_{curr}, there exists a bag written at p_x and there exists no bag at p_x during op_{prev} then, the domination property holds since the seq value is 0 during op_{prev} and the seq value is > 0 during op_{curr}.

9. The interesting case is when $BA_{p_x,C_{prev}}$ exists and is not equal to $BA_{p_x,C_{curr}}$.

10. The bag at address p_x in the root node is calculated in line *SREAD2* from Shared Memory S by the same process p_x. Also since, we are considering the bags at address p_x, because of SWMR, it must be case that these bags were written in the respective Containers during two different operations by the same process p_x. Let the two operations be op_{x1} and op_{x2} during which $BA_{p_x,C_{prev}}$ and $BA_{p_x,C_{curr}}$ were written in the Container respectively.

11. Because of SWMR, we know the two operations executed by the same process follow partial ordering. Therefore, op_{x1} and op_{x2} follow partial ordering.

12. From the previous steps, we know the following:

    (a) The bag written in the Container during op_{x1} was present during op_{prev}. Therefore, the start of op_{x1} must be before the end of the op_{prev}.

    (b) The bag written in the Container during op_{x2} was present during op_{curr}. Therefore, the start of op_{x2} must be before the end of the op_{curr}.

    (c) op_{x1} and op_{x2} follow partial ordering

    (d) Since op_{prev} is partially ordered before op_{curr}, it must be case that op_{x1} is partially ordered before op_{x2}. The following is a quick proof for this claim: We use proof by contradiction. Assume, op_{x2} is partially ordered before op_{x1}. That means, the end of op_{x2} is before the start of op_{x1}. We also know that the start of op_{x1} is before the end of the op_{prev}. Therefore, the end of op_{x2} is before the end of the op_{prev}. However, the start of op_{x2} is before the

end of the op_{curr}. Therefore, the start of op_{curr} must be before op_{prev} which contradicts the condition where op_{prev} is partially ordered before op_{curr}.

13. In other words, $BA_{p_x,C_{prev}}$ and $BA_{p_x,C_{curr}}$ are obtained from two separate reads from the shared memory S by the same process p_x where the bag $BA_{p_x,C_{prev}}$ was read during an operation which is partially ordered before the operation where the bag $BA_{p_x,C_{curr}}$ was read. Since in the code, no items are removed from the active list and the seq value never decrease in the shared memory S, the seq values in each block of the bag $BA_{p_x,C_{curr}}$ will be at least as big seq values in each block of the bag $BA_{p_x,C_{prev}}$ and $addr_{BA_{p_x,C_{prev}}} \subseteq addr_{BA_{p_x,C_{curr}}}$. Therefore, from the definition of domination the bag $BA_{p_x,C_{curr}}$ will dominate $BA_{p_x,C_{prev}}$. (Base case proved)

Inner Induction Hypothesis: Assume $P(a, b)$ is true where the values of a and b are as follows:

1. $a = 1....\ell - 1$ where $2 \leq \ell \leq \gamma$

2. $b = t_{curr}$

Inner Induction Step (we prove $P(\ell, t_{curr})$ is true):

1. Since the Classifier graph is a binary tree, there is only parent of the node u. It also follows that the node u is either a left child or a right child of the parent.

2. If it is a left child of the parent then all the inputs in this node are the same as the inputs to the parent node for any operation. Therefore, at the current node, the state of the Container is same as the parent node. By the Inner Induction Hypothesis, we know at level $\ell - 1$, each bag at address p_x in C_{prev} is dominated by the bag at the same address p_x in C_{curr} at level ℓ. Since op_{prev} is partially ordered before op_{curr}, the states of the Container C_{curr} is the same or more updated than state of the Container C_{prev} at level ℓ. Therefore, each bag at address p_x in C_{prev} is dominated by the bag at the same address p_x in C_{curr}

at level ℓ. This proves the induction when the current node is the left child of the parent node. In the following steps, we consider the current node to be the right child of the parent node.

3. Let the parent node of the node u located at level $\ell-1$ be par. For all operations that reach node u, the input to the Classifier algorithm at node u is the merge of the Container at node par. Therefore, $ss_{\ell-1,op_{prev}}$ and $ss_{\ell-1,op_{curr}}$ are the merge of C_{prev} and C_{curr} respectively at level $\ell-1$.

4. At the node par, LCA(op_{prev}, op_{curr}) is the node par itself and these operations follow partial ordering. Applying these conditions to the Lemma 9.16 we get: $updatedActiveList_{op_{prev}} \subseteq updatedActiveList_{op_{curr}}$.

5. Therefore, each process that wrote a bag at address $p_j \in updatedActiveList_{op_{prev}}$ in C_{prev}, also wrote a bag at address p_j in C_{curr} at node par. Applying this fact and the fact that the inner induction hypothesis states that the bags in C_{curr} dominates the bags in C_{prev} at the same address to Proposition 3.2, we know that merge(C_{curr}) dominates merge(C_{prev}).

6. The merge(C_{curr}) and merge(C_{prev}) from level $\ell-1$ are the input bags $ss_{\ell-1,op_{curr}}$ and $ss_{\ell-1,op_{prev}}$ respectively at level ℓ. Since $ss_{\ell-1,op_{curr}}$ dominates $ss_{\ell-1,op_{prev}}$ and these bags are written down at address p_i, we now have proved that the bag at address p_i in C_{prev} is dominated by the bag at the same address p_i in C_{curr} at level ℓ at t_{curr}. In the following steps, we will consider the rest of the addresses other than p_i in the same level ℓ (unless otherwise specified).

7. Consider any arbitrary bag at address $p_x \neq p_i$ in C_{curr} at t_{curr}. Since this bag exists in C_{curr}, it must have been written by p_x during some operation because of SWMR property. Let the bag at p_x in C_{curr} be written during an operation $op_{p_x,curr}$ at time $t_{p_x,curr}$.

8. Note that since the bag at address p_x exists in C_{curr} during op_{curr}, $t_{p_x,curr} < t_{curr}$.

9. Consider the same bag at address p_x in C_{prev} at t_{prev}. Since this bag exists in C_{prev}, it must have been written by p_x during some operation because of SWMR property. Let the bag at p_x in C_{prev} be written during an operation $op_{p_x,prev}$ at time $t_{p_x,prev}$.

10. Note that since the bag at address p_x exists in C_{prev} during op_{prev}, $t_{p_x,prev} < t_{prev}$.

11. From the previous 4 steps, we know the following:

    (a) For an arbitrary process p_x, there are two states of the Containers C_{curr} and C_{prev} at two different times $t_{p_x,curr}$ and $t_{p_x,prev}$ respectively

    (b) $t_{p_x,prev} < t_{prev}$

    (c) $t_{p_x,curr} < t_{curr}$

12. Applying the conditions from the previous step in the Outer Induction Hypothesis, we know the bag at p_x in C_{curr} dominates the bag at the same address in C_{prev} at $t_{p_x,curr}$.

13. However, at $t_{p_x,curr}$ the bag at p_x in C_{curr} is the same bag at p_x in C_{curr} at t_{curr}.

14. Therefore, each bag at address p_x in C_{prev} is dominated by the bag at the same address p_x in C_{curr} at level ℓ at t_{curr}.

□

In the next lemma, we prove that at any node in the Classifier Tree, when the input bag is written down in the Container of that node, the new bag dominates the existing bag in the Container at that address.

Lemma 9.20. *Consider an arbitrary process p_i, and let op_{curr} be the current operation executed by p_i. During the execution of op_{curr}, for any level $\ell \geq 1$, at line 2 of the Classifier algorithm, the bag $ss_{\ell-1}$ is written down at address p_i in the Container C. Before $ss_{\ell-1}$ is written down, let the bag at address p_i in the Container C be $BA_{prev,C}$. We prove that $ss_{\ell-1}$ dominates $BA_{prev,C}$.*

Proof. If this is the first time $ss_{\ell-1}$ is being written down at address p_i at line 2 of the Classifier algorithm then $BA_{prev,C}$ is dominated by $ss_{\ell-1}$. This is because:

1. Since it is the first time $ss_{\ell-1}$ is being written down at address p_i, $BA_{prev,C}$ is the bag that was initialized by the system which means the seq values are all zero.

2. The seq values in $ss_{\ell-1}$ cannot be less than zero since all blocks are initialized with the seq value as zero and the seq value never decreases in the code.

3. Since $BA_{prev,C}$ is the bag that was initialized by the system, it is $procList_\alpha$ which is a subset of the addresses in $ss_{\ell-1}$ (Table 3.3).

We consider the case where there already exists a bag before $ss_{\ell-1}$ is written down at line 2. From Lemma 9.19, we know that the bag at address p_i in C_{prev} is dominated by the bag at the same address p_i in C_{curr} at level ℓ. Therefore, it also stands that during op_{curr}, $ss_{\ell-1}$ dominates $BA_{prev,C}$ at any level ℓ. □

In the next lemma, we prove that at any node in the Classifier Tree, for any set of Classifier calls at the node, the merge of all bags passed as input into the Classifier calls dominates any of the output bags.

Lemma 9.21. *For arbitrary $a \geq 1$, let u be a node in the Classifier Tree located at level a. Let OP be a set of operations that execute the Classifier algorithm in node u.*

Let C_{inp} be a Container consisting of all the input bags ss_{a-1} that were passed into Classifier at node u during the operations in OP. Let $ADDR_{inp}$ be the union of all the input $addr_{a-1}$ lists that were passed into Classifier at node u during the operations in OP.

We prove that $merge(ADDR_{inp}, C_{inp})$ dominates ss_a, where ss_a is the output of Classifier at node u during any one operation $op_i \in OP$.

Proof. Case 1: ss_a is returned from line 11 (loser case) during op_i:

1. Since line 11 is reached, $addr_{a-1} = addr_a$ and $ss_{a-1} = ss_a$.

2. From the lemma definition, $addr_{a-1} \subseteq ADDR_{inp}$. Therefore, $addr_a \subseteq ADDR_{inp}$.

3. Since $ss_{a-1} = ss_a$ and $BA_{p_i,C_{inp}}$ dominates ss_{a-1} (By Lemma 9.20), $BA_{p_i,C_{inp}}$ dominates ss_a

4. From the definition of merge, for each address j that are in both $ADDR_{inp}$ and $addr_a$, $BL_{j,merge(ADDR_{inp},C_{inp})}$.seq $\geq BL_{j,ss_a}$.seq.

5. From the definition of domination (using facts from the previous two steps) $merge(ADDR_{inp}, C_{inp})$ dominates ss_a.

Case 2: ss_a is returned from line 9 (winner case) during op_i:

1. Let the Container be C'_{inp} during op_i at line 9. This means ss_{a-1} is written in C'_{inp}. This also means, any bag BA_{p_x} at address p_x that is written down to C'_{inp} is also written to C_{inp} and from Lemma 9.20, $BA_{p_x,C_{inp}}$ dominates $BA_{p_x,C'_{inp}}$.

2. From Proposition 3.2 we know, $merge(ADDR_{inp}, C_{inp})$ dominates $merge(addr_{a-1}, C'_{inp})$.

3. Since line 9 is reached, $merge(addr_{a-1}, C'_{inp}) = ss_a$. Therefore, $merge(ADDR_{inp}, C_{inp})$ dominates ss_a.

□

In the next lemma, we prove that at any node in the Classifier Tree, the seqSum of the merge of all input bags is greater than or equal to the seqSum of the merge of all output bags that are returned from the winner case (i.e., returned at line 9).

Lemma 9.22. *For arbitrary $a \geq 1$, let u be any node located at level a in the Classifier Tree. Let OP_w be the set of operations where line 9 was reached during the execution of the Classifier algorithm at node u. Let OP_{inp} be the set of all operations where the Classifier algorithm was executed at node u. Recall that, at any node u in the Classifier Tree, $addr_{a-1}$ and ss_{a-1} are the input addresses and input bags of the Classifier algorithm and $addr_a$ and ss_a are the output addresses and output bags of the Classifier algorithm.*

Let $ADDR_{inp}$ be the union of all $addr_{a-1}$ during the operations OP_{inp}. Let C_{inp} be a Container consisting of all the input bags ss_{a-1} that were passed into Classifier at node u during the operations in OP_{inp}.

Let $ADDR_w$ be the union of all $addr_a$ during the operations OP_w. Let C_w be a Container consisting of all the output bags ss_w that are returned by the operations in OP_w.

We prove that seqSum of $merge(ADDR_{inp}, C_{inp})$ is greater than or equal to the seqSum of $merge(ADDR_w, C_w)$.

Proof. Let $mergedBA_{inp} = merge(ADDR_{inp}, C_{inp})$ and $mergedBA_w = merge(ADDR_w, C_w)$. We first prove that $merge(ADDR_{inp}, C_{inp})$ dominates $merge(ADDR_w, C_w)$.

To prove $mergedBA_{inp}$ dominates $mergedBA_w$, we will be proving the following dominating conditions:

1. $ADDR_w \subseteq ADDR_{inp}$
2. For any address p_j, (where $p_j \in (ADDR_w \cap ADDR_{inp})$, $BL_{p_j,mergedBA_{inp}}.\text{seq} \geq BL_{p_j,mergedBA_w}.\text{seq}$

We prove these in the following steps:

1. From Lemma 9.21, we know $merge(ADDR_{inp}, C_{inp})$ dominates any single Classifier output bag ss_w in C_w. From the definition of domination, it follows that $addr_a \subseteq ADDR_{inp}$ for any single Classifier output $addr_a$ during any operation from OP_w. This implies that the union of all $addr_a$ outputs by Classifier during the operations in OP_w is a subset of $ADDR_{inp}$. However, the union is exactly $ADDR_w$. Therefore, $ADDR_w \subseteq ADDR_{inp}$ (and this proves the first dominating condition).

2. Since each $ss_w \in C_w$ is dominated by $merge(ADDR_{inp}, C_{inp})$, and we already showed that $ADDR_w \subseteq ADDR_{inp}$, it follows that $merge(ADDR_w, C_w)$ is dominated by $merge(ADDR_{inp}, C_{inp})$. (This proves the second domination condition)

Since $merge(ADDR_{inp}, C_{inp})$ dominates $merge(ADDR_w, C_w)$, it follows from Proposition 3.1 that the $seqSum$ of $merge(ADDR_{inp}, C_{inp})$ is greater than or equal to the $seqSum$ of $merge(ADDR_w, C_w)$. □

The next lemma is a rewrite of Lemma 3.1 in Attiya and Rachman [16]. There are five facts proven in this lemma. In the first two facts, we show that the Classifier Tree follows a binary search tree structure in the subtree rooted at the current node where everything that is going right is in between the label of the node and H and everything that is going left is in between L and the label of the node. Then, the next two facts show that the seqSum of the merge of all the winner operations is bounded above by H, and, the seqSum of the merge of all the loser operations is bounded above by the label of the current node. Finally, the fifth fact shows that the merge of all the loser operations is dominated by any winner operation at the current node.

Lemma 9.23. *For an arbitrary $\ell \geq 1$, let v be any node at level ℓ in the Classifier Tree.*

Let OP_{input} be the set of all operations that execute Classifier at the Classifier Tree node v. Let C_{input} be a Container consisting of the input bags that are passed to the Classifier algorithm during each operation in OP_{input}. Let $ADDR_{input}$ be the union of all the address lists that are passed to the Classifier algorithm during each operation in OP_{input}.

Let $ss_{\ell-1,op_i}$ and $addr_{\ell-1,op_i}$ be the input bag and input address list respectively to the Classifier algorithm at node v during $op_i \in OP_{input}$.

Let OP_{winner} be the set of all operations that execute Classifier at the Classifier Tree node v such that line 9 (winner case) is reached.

Let $ADDR_{winner}$ be the union of all the address lists that are returned during each operation in OP_{winner}.

Let OP_{loser} be the set of all operations that execute Classifier at the Classifier Tree node v such that line 11 (lower case) is reached.

Let $ADDR_{loser}$ be the union of all the address lists that are returned during each operation in OP_{loser}.

Suppose there are non-negative integers L, H, such that the following conditions are met:

1. $L < getSeqSum(ss_{\ell-1,op_i}, addr_{\ell-1,op_i}) \leq H$ *where* $op_i \in OP_{input}$

2. $getSeqSum(merge(ADDR_{input}, C_{input}), ADDR_{input}) \leq H$.

We prove the following:

1. *Label(v)* $< getSeqSum(ss_{\ell,op_i}, addr_{\ell,op_i}) \leq H$ *for each* $op_i \in OP_{winner}$

2. $L < getSeqSum(ss_{\ell,op_i}, addr_{\ell,op_i}) \leq$ *Label(v) for each* $op_i \in OP_{loser}$

3. *Let* C_{winner} *be a Container consisting of the bags that are returned from the Classifier algorithm during each operation in* OP_{winner}*. We prove:* $getSeqSum(merge(ADDR_{winner}, C_{winner}), ADDR_{winner}) \leq H$

4. *Let* C_{loser} *be a Container consisting of the bags that are returned from the Classifier algorithm during each operation in* OP_{loser}*. We prove:* $getSeqSum(merge(ADDR_{loser}, C_{loser}), ADDR_{loser}) \leq Label(v)$

5. *Let* C_{loser} *be a Container consisting of the bags that are returned from the Classifier algorithm during each operation in* OP_{loser}*. We prove:* ss_{ℓ,op_i} *dominates* $merge(ADDR_{loser}, C_{loser})$ *for each* $op_i \in OP_{winner}$

Proof. We prove the 5 parts of the lemma statement in the following steps:

1. Proof for Part 1: In the Classifier algorithm K = Label(v). Since here $op_i \in OP_{winner}$, line 9 is reached. In order to reach line 9, the condition at line 6 must be true. The condition at line 6 states Label(v) $< getSeqSum(ss_{\ell,op_i}, addr_{\ell,op_i})$. Therefore, the left side of the inequality is proved.

   Since here $op_i \in OP_{winner}$, ss_{ℓ,op_i} is the merge of a Container that is a subset of C_{input}, and, $addr_{\ell,op_i}$ is a subset of $ADDR_{input}$. Since we know that: $getSeqSum(merge(ADDR_{input}, C_{input}), ADDR_{input}) \leq H$,
   then due to the subset relationships stated in the previous sentence, it follows that $getSeqSum(ss_{\ell,op_i}, addr_{\ell,op_i}) \leq H$.

2. Proof for Part 2: In the Classifier algorithm K = Label(v). Since here $op_i \in OP_{loser}$, line 11 is reached. In order to reach line 11, the condition at line 6 must be false. The condition at line 6 states Label(v) $< getSeqSum(ss_{\ell,op_i}, addr_{\ell,op_i})$. The negation of this condition is Label(v) $\geq getSeqSum(ss_{\ell,op_i}, addr_{\ell,op_i})$. Since here $op_i \in OP_{loser}$, the input and output of the Classifier algorithm are the same. Therefore, the right side of the inequality is proved.

   We know from the lemma statement:
   $L < getSeqSum(ss_{\ell-1,op_i}, addr_{\ell-1,op_i})$.
   We also know that since here $op_i \in OP_{loser}$, the input and output of the Classifier algorithm are the same. Therefore,
   $L < getSeqSum(ss_{\ell,op_i}, addr_{\ell,op_i})$.

3. Proof for Part 3: From Lemma 9.22 we know, $seqSum$ of $merge(ADDR_{input}, C_{input})$ is greater than or equal to the $seqSum$ of $merge(ADDR_{winner}, C_{winner})$. However, from the lemma statement we know H is greater than or equal to $seqSum$ of $merge(ADDR_{input}, C_{input})$. Therefore, H is greater than or equal to $seqSum$ of $merge(ADDR_{winner}, C_{winner})$.

4. Proof for Part 4: The proof for this shown in Attiya and Rachman [16] in Lemma 3.1 (b4).

5. Proof for Part 5: The proof for this shown in Attiya and Rachman [16] in Lemma 3.1 (b5).

□

The next lemma is a rewrite of Lemma 3.2 in Attiya and Rachman [16]. Consider any node v in the Classifier Tree. For any operation op that reached the loser case of node v (i.e., returned at line 11 when Classifier was executed at v), consider the bag returned by the final call to Classifier during op (i.e., the bag returned when Classifier is eventually called at some leaf in the Classifier Tree). In this lemma, we prove that, this final result bag will be dominated by the merge of all the bags that are returned from the loser case of node v.

Lemma 9.24. *Let op_i be an operation that returns the bag ss_{final} from the Classifier call (at line POSTC) at the level* $\log(m) + 1$. *Let v be a node at any level ℓ in the Classifier Tree where ss_ℓ (the result of the Classifier algorithm at level ℓ) was returned from line 11 (loser case) of the Classifier algorithm during op_i.*

Let OP be the set of all operations that satisfy the following conditions:

1. *Reached node v in the Classifier Tree during its execution*
2. *During the execution of the Classifier algorithm at node v, line 11 was reached (loser case)*

Let C' be a Container consisting of the bags ss_ℓ returned by each operation in OP after executing the Classifier algorithm at node v. Let $ADDR$ be the union of the $addr_\ell$ lists returned by each operation in OP after executing the Classifier algorithm at node v.

We prove that ss_{final} is dominated by $merge(ADDR, C')$.

Proof. We prove this by induction on the path length from node v to the leaf node from where ss_{final} was returned. Let P(k) = "$merge(ADDR, C')$ dominates ss_a where ss_a is the output of the Classifier algorithm at level a during op_i at node u which is located at distance k in the path from node v to the leaf node from where ss_{final} is returned."

Base Case (k = 0):

1. Here, node a = node v.
2. Since current node is the node v, ss_a is one of the bags in the Container C' and $addr_a$ is a subset of ADDR.
3. From the definition of merge, for each address j that are in both $ADDR$ and $addr_a$, $BL_{j,merge(ADDR,C')}$.seq $\geq BL_{j,ss_a}$.seq.
4. From the definition of domination (using facts from the previous two steps) $merge(ADDR, C')$ dominates ss_a. (Base case proved)

Induction Hypothesis: Assume $P(0) \wedge ... \wedge P(k-1)$ is true for some k ≥ 1.

Induction Step:

1. Let C_{inp} be a Container consisting of the input bags ss_{a-1} during each operation in OP while executing the Classifier algorithm at node u. Let $ADDR_{inp}$ be the union of the input $addr_{a-1}$ lists during each operation in OP while executing the Classifier algorithm at node u.

2. We know $merge(ADDR_{inp}, C_{inp})$ dominates ss_a. (From Lemma 9.21)

3. Since the Classifier Tree is a tree structure, node u can have only one parent node. Let the parent node be par. Node par is distance k - 1 away from node v. From the Induction Hypothesis, we know the output of the node par is dominated by $merge(ADDR, C')$. The inputs to the node u are the outputs of the node par. In particular, this means that each bag in C_{inp} is dominated by $merge(ADDR, C')$, and $ADDR_{inp} \subseteq ADDR$. It follows that $merge(ADDR_{inp}, C_{inp})$ is dominated by $merge(ADDR, C')$.

4. From step 2, $merge(ADDR_{inp}, C_{inp})$ dominates ss_a. Therefore, from transitivity property, $merge(ADDR, C')$ dominates ss_a.

Concluding Statement: $merge(ADDR, C')$ dominates ss_a where ss_a is the output of the node u at distance k from the node v in the path to the leaf node. Therefore, when the leaf node is reached, $merge(ADDR, C')$ dominates ss_{final}. $\square$

The next lemma is a rewrite of Lemma 3.3 in Attiya and Rachman [16]. Consider any node v in the Classifier Tree. For any operation op that reached the winner case of node v (i.e., returned at line 9 when Classifier was executed at v), consider the bag returned by the final call to Classifier during op (i.e., the bag returned when Classifier is eventually called at some leaf in the Classifier Tree). In this lemma, we prove that this final result bag dominates any bag that is returned from the loser case (line 11) at node v.

Lemma 9.25. *Let op_i be an operation that returns the bag ss_{final} from the Classifier call (at line POSTC) at the level* $\log(m) + 1$. *Let u be a node at any level ℓ in the Classifier Tree where the result is returned from line 9 (winner case) of the Classifier algorithm during op_i.*

Let OP_{loser} be the set of all operations where each operation satisfies the following conditions:

1. *Reached node u in the Classifier Tree during its execution*

2. *During the execution of the Classifier algorithm at node u, line 11 is reached (loser case)*

Let ss_ℓ be the bag that is returned from the execution of the Classifier algorithm at node u during any arbitrary operation in OP_{loser}. We prove that ss_{final} dominates ss_ℓ.

Proof. 1. We argue that the output bag of every execution of the Classifier algorithm dominates the input bag:

(a) If the output is returned from the loser case at line 11 in the Classifier algorithm, then the output bag is equal to the input bag, and so the domination property holds and input is dominated by the output bag. The following arguments are when the output is returned from the winner case at line 9 in the Classifier algorithm.

(b) The input is written down at line 2 in the Container. Therefore, the input is considered when the mergeRecent is calculated at line 5. From the merge definition (Section: 3.2.3), we know that each bag in the mergeRecent bag has the maximal seq value. Since the input bag was considered during the calculation of the mergeRecent, each seq value in the mergeRecent bag is at least as big in the input bag. mergeRecent is the output of the Classifier algorithm and therefore, output dominates the input bag.

Let node v be the node at level $\ell + 1$ during op_i. Therefore, the input to node v is dominated by the output of the node v. We can extend this parent child

domination relationship all the way to the leaf node and therefore, the input of the node v is dominated by the output of the leaf node which is ss_{final}

2. Let C' be a Container consisting of the bags returned by each operation in OP_{loser} after executing the Classifier algorithm at node v. Let ADDR be the union of the address lists returned by each operation in OP_{loser} after executing the Classifier algorithm at node v. From 3.1(b5) of the paper Attiya and Rachman [16] (Lemma 9.23 Part: 5), we know the input to node v during op_i dominates $merge(ADDR, C')$.

3. However, from step (1), ss_{final} dominates the input to node v during op_i. Therefore, ss_{final} dominates $merge(ADDR, C')$.

4. From the definition of merge (Section 3.2.3), we know in the merged bag, each block has the maximal seq value. Since ss_{final} dominates $merge(ADDR, C')$, ss_{final} also dominates any individual bag ss_ℓ in C' (from the domination property in Section 3.2.2).

□

In what follows, when we say that two bags are comparable, we mean that one bag dominates the other. In the next lemma, we prove that if two operations end up at different leaves in the Classifier Tree, then the result returned from the final calls to the Classifier algorithm are comparable.

Lemma 9.26. *Let op_i and op_j be two operations that terminate at Classifier Tree leaf nodes CLL_i and CLL_j respectively where $CLL_i\ != CLL_j$. Let ss_{final,op_i} and ss_{final,op_j} be the bags returned from the final call to the Classifier algorithm during op_i and op_j respectively. We prove that ss_{final,op_i} and ss_{final,op_j} are comparable.*

Proof. The proof for this shown in Attiya and Rachman [16] in Lemma 3.4. □

The next lemma is a rewrite of Lemma 3.5 in Attiya and Rachman [16], and provides a relationship between the label of a node v in the Classifier Tree and the seqSum values of bags passed in as input when Classifier is executed at node v.

Lemma 9.27. *For an arbitrary $\ell \in \{1, \ldots, \log(m)\}$, let v be any node in the Classifier Tree at level ℓ. Let OP_{input} be the set of all operations that execute Classifier at the Classifier Tree node v. Let $ADDR_{input}$ be the union of all the address lists that are passed to the Classifier algorithm during each operation in OP_{input}. Let C_{input} be a Container consisting of the input bags that are passed to the Classifier algorithm during each operation in OP_{input}. We prove the following:*

1. *For every $op_i \in OP_{input}$, Label(v) - $(m/2^\ell) <$ seqSum$(ss_{\ell-1}, addr_{\ell-1}) \leq$ Label(v) + $(m/2^\ell)$*

2. *seqSum(merge($ADDR_{input}$, C_{input}), $ADDR_{input}$) $\leq$ Label(v) + $(m/2^\ell)$*

Proof. The proof for this shown in Attiya and Rachman [16] in Lemma 3.5. □

The next lemma is a rewrite of Lemma 3.6 in Attiya and Rachman [16]. It shows that if two operations end up at the same leaf in the Classifier Tree then the bags returned by their final calls to Classifier are the same. In particular, if the two operations are snap operations, then the output of the two snap operations are equal.

Lemma 9.28. *Let op_i and op_j be two operations that terminate at Classifier leaf nodes CLL_i and CLL_j respectively where $CLL_i == CLL_j$. Let ss_{final,op_i} and ss_{final,op_j} be the bags returned from the final call to Classifier algorithm during op_i and op_j respectively. We prove that ss_{final,op_i} and ss_{final,op_j} are equal.*

Proof. The proof for this shown in Attiya and Rachman [16] in Lemma 3.6.

□

10 Conclusion and Future Work

The dynamic atomic snapshot algorithm is an extension of a popular algorithm in distributed computing. The static atomic snapshot algorithm is primarily used as a building block in other shared memory algorithms for simple (often theoretical) systems. However, with the dynamic variant, it has become an interesting problem

not only from theoretical point of view but also for any large scale applications in real-world scenarios. We hope that with the implementation of the optimized dynamic snapshot algorithm, we will be able to unlock new possibilities in this area of distributed computing.

10.1 Future Work

In our current implementation, there is a weakness in the remove operation and that is if p_a adds p_b and p_b adds p_c, if p_b is later removed by p_a, then p_c can never be removed. One of the ways to solve this is to recursively delete all descendants of p_b when it is removed. Or, another solution can be to restore the right to remove to the immediate active ancestor. In both cases, the main challenge would be to carefully handle situations where a process is removed while it is in the middle of adding another process.

Our current implementation is restricted to an m-shot object, and we would like to extend this to an ∞-shot object (i.e., no fixed bound on the number of operations that will be performed). It would be interesting to leverage the $O(n \log n)$ idea described in Attiya and Rachman [16] and implement an ∞-shot dynamic atomic snapshot object. The challenge here would be to integrate the add and remove operations with their ∞-shot algorithm.

In our current implementation, when a process is removed from the system, the memory used by the process is not freed. The primary reason why we retain the information about the removed processes so that we can keep track of their seq values. The seq values of the processes are later used in calculating the seqSum which plays a key role in the Classifier algorithm. For example, if we forget about the removed processes and then new processes are added with an initialized seq value of 0, the seqSum calculated in later operations can suddenly become very small, and our linearization mapping (which essentially sorts by seqSum) would be “fooled” into thinking that these later operations happen very early in the execution. It would be interesting to implement a more memory efficient version of the atomic snapshot object where when a process is removed, the memory related to that process is freed up and can

later be assigned to other new processes. A similar idea for this implementation has been explored in Spiegelman and Keidar [49].

Our work follows the standard model in the literature, i.e., Afek, Attiya, Dolev, Gafni, Merritt, and Shavit [3] assumes that an unbounded amount of information can be read/written from a register in $O(1)$ time. This also applies in our algorithm where we can read and write a single Container from the shared memory in $O(1)$ time. Future work can consider the possibility of replacing such an assumption with a more realistic one, and this direction has been explored in Zhu and Ellen [50].

Recently there has been a growing interest in using anonymous shared memory in distributed computing. The anonymous atomic snapshot was first introduced in Guerraoui and Ruppert [23]. The goal of their work was to implement the process anonymous version of the atomic snapshot algorithm. Multi-reader multi-writer registers were used for the implementation of this atomic snapshot algorithm otherwise the anonymous property of the register would not hold. In contrast, our implementation uses process identities to distinguish the registers. So, it would be interesting to explore this variant of atomic snapshot, especially with regards to data security and privacy, and implement a dynamic anonymous atomic snapshot object.

Bibliography

[1] U. Abraham and G. Amram. On the mailbox problem. In *International Conference on Principles of Distributed Systems*, pages 453–468. Springer, 2014.

[2] K. Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 291–302, 1988.

[3] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.

[4] Y. Afek, N. Shavit, and M. Tzafrir. Interrupting snapshots and the javatm size method. *Journal of Parallel and Distributed Computing*, 72(7):880–888, 2012.

[5] M. K. Aguilera, E. Gafni, and L. Lamport. The mailbox problem. In *International Symposium on Distributed Computing*, pages 1–15. Springer, 2008.

[6] M. K. Aguilera, E. Gafni, and L. Lamport. The mailbox problem. *Distributed Computing*, 23(2):113–134, 2010.

[7] G. Amram. The f-snapshot problem. In *International Colloquium on Structural Information and Communication Complexity*, pages 159–176. Springer, 2016.

[8] J. H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.

[9] J. H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, 1994.

[10] J. Aspnes, H. Attiya, K. Censor-Hillel, and F. Ellen. Faster than optimal snapshots (for a while) preliminary version. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 375–384, 2012.

[11] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of algorithms*, 11(3):441–461, 1990.

[12] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous pram model. In *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, pages 340–349, 1990.

[13] H. Attiya, D. Dolev, and N. Shavit. Bounded polynomial randomized consensus. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 281–293, 1989.

[14] H. Attiya, R. Guerraoui, and E. Ruppert. Partial snapshot objects. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 336–343, 2008.

[15] H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? *Journal of the ACM (JACM)*, 41(4):725–763, 1994.

[16] H. Attiya and O. Rachman. Atomic snapshots in o (n log n) operations. *SIAM Journal on Computing*, 27(2):319–340, 1998.

[17] S. C. Bono, C. A. Soghoian, and F. Monrose. Mantis: A lightweight, server-anonymity preserving, searchable p2p network. *Information Security Institute Johns Hopkins University, USA, Tech. Rep. TR-2004-01-B-ISI-JHU*, 2004.

[18] C. Cachin et al. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310, pages 1–4. Chicago, IL, 2016.

[19] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[20] D. Dolev, E. Gafni, and N. Shavit. Toward a non-atomic era: l-exclusion as a test case. In *Proceedings of the twentieth Annual ACM Symposium on Theory of Computing*, pages 78–92, 1988.

[21] D. Dolev and N. Shavit. Bounded concurrrent time-stamp systems are constructible. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 454–466, 1989.

[22] K. Goldman and K. Yelick. A unified model for shared-memory and message-passing systems. 1993.

[23] R. Guerraoui and E. Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

[24] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[25] M. Herlihy and V. Luchangco. Distributed computing and the multicore revolution. *ACM SIGACT News*, 39(1):62–72, 2008.

[26] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[27] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[28] D. Imbs and M. Raynal. A simple snapshot algorithm for multicore systems. In *2011 5th Latin-American Symposium on Dependable Computing*, pages 17–24. IEEE, 2011.

[29] D. Imbs and M. Raynal. Help when needed, but no more: efficient read/write partial snapshot. *Journal of Parallel and Distributed Computing*, 72(1):1–12, 2012.

[30] P. Jayanti. f-arrays: Implementation and applications. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 270–279, 2002.

[31] P. Jayanti. An optimal multi-writer snapshot algorithm. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 723–732, 2005.

[32] Y. Kim, J. Nakamura, Y. Katayama, and T. Masuzawa. A cooperative partial snapshot algorithm for checkpoint-rollback recovery of large-scale and dynamic distributed systems. In *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 285–291. IEEE, 2018.

[33] L. Lamport. On interprocess communication, part 1: basic formalism, part ii: algorithms. *Distributed Computing. v1 i2*, pages 77–101.

[34] L. Lamport. On interprocess communication, part ii; algorithms. *Distributed Computing*, 1:86–101, 1986.

[35] L. Lamport. The mutual exclusion problem: part i—a theory of interprocess communication. In *Concurrency: the Works of Leslie Lamport*, pages 227–245. 2019.

[36] L. Lamport. The mutual exclusion problem: part ii—statement and solutions. In *Concurrency: the Works of Leslie Lamport*, pages 247–276. 2019.

[37] L. Lamport and F. B. Schneider. Pretending atomicity. *SRC Research Report 44*, 1989.

[38] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 137–151, 1987.

[39] N. A. Lynch and M. R. Tuttle. *An introduction to input/output automata*. Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[40] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[41] G. L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(1):46–55, 1983.

[42] G. L. Peterson and J. E. Burns. Concurrent reading while writing ii: the multi-writer case. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 383–392. IEEE, 1987.

[43] M. Raynal and J. Cao. Anonymity in distributed read/write systems: an introductory survey. In *International Conference on Networked Systems*, pages 122–140. Springer, 2018.

[44] M. Raynal and G. Taubenfeld. Fully anonymous shared memory algorithms. *arXiv preprint arXiv:1909.05576*, 2019.

[45] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM transactions on information and system security (TISSEC)*, 1(1):66–92, 1998.

[46] Y. Riany, N. Shavit, and D. Touitou. Towards a practical snapshot algorithm. *Theoretical Computer Science*, 269(1-2):163–201, 2001.

[47] R. Schaffer. On the correctness of atomic multi-writer registers. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1988.

[48] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[49] A. Spiegelman and I. Keidar. Dynamic atomic snapshots. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[50] L. Zhu and F. Ellen. Atomic snapshots from small registers. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, pages 17–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.