# Wait-free Linearizable Queue Implementations

by

Matei David

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

# Abstract

Wait-free Linearizable Queue Implementations

Matei David
Master of Science
Graduate Department of Computer Science
University of Toronto
2004

We study wait-free linearizable Queue implementations in asynchronous shared-memory systems from other consensus number 2 types, such as Fetch&Add and Swap. It was known that such implementations exist when at most two processes perform dequeue operations. We provide a new implementation, when only one process performs enqueue operations and any number of processes perfrom dequeue operations. We introduce the BH&AR type, which has two important characteristics: only one BH&AR object simulates countably infinitely many Fetch&Add objects and Registers; and we can exactly quantify the flow of information in a system providing only one BH&AR object. The latter might prove useful in developing an adversarial argument showing that there is no implementation of a Queue from Fetch&Add objects when three processes can fully access the Queue. The B-History and Basic-Id-Queue types are restricted versions of the BH&AR and Queue types, respectively. We show that implementing a Queue from one BH&AR object is equivalent to implementing a Basic-Id-Queue from one B-History object. Preliminary results show that this is impossible for two restricted classes of implementations.

# Acknowledgements

First of all, I would like to thank Andreea, my family and my old friends Lali, Bogdan, Alex and many others, for shaping me into the person that I am today.

I would like to express my gratitude towards Canada as a country, for accepting me as a landed immigrant and for financially supporting me during my undergraduate studies here. I would like to thank NSERC for partly supporting my graduate studies with a PGS A scholarship.

The University of Toronto in general, and its department of Computer Science in particular, are great places for learning and living. I am very grateful to my fellow graduate students, not only for their helpful discussions on academic topics, but also for their moral support and extra-curricular activities. The list of first names would include Travis, Cristiana, Daniela, Leoni, Alanoman, Josh, Steve, Vlad, Natasha, Antonina, Panayiotis, Sebastian, Stavros and probably many others.

I would like to thank Nir Shavit, for his prompt reading of my thesis.

Last but not least, I would like to thank my supervisor, Faith Ellen Fich, for her helpful advice, her constructive criticism of my sometimes hurried writing, her patience with me, and in general for the invaluable support she provided me with throughout my studies here.

# Contents

**Abstract** **ii**

**Acknowledgements** **iii**

**1 Introduction** **1**

1.1 Related Work . . . 3

1.2 Mathematical Notation . . . 5

1.3 System Model Definitions . . . 5

1.4 Shared Object Types . . . 10

**2 A $(1, n)$-Queue implementation** **13**

2.1 The algorithm . . . 13

2.2 Proof of Correctness . . . 16

**3 Simulating a System with One Object** **28**

3.1 The B-History and BH&AR Object Types . . . 29

3.1.1 An Informal Description of the B-History and BH&AR Types . . . 29

3.1.2 Notion of Indistinguishability . . . 30

3.1.3 Formal Object Type Definitions . . . 33

3.2 Implementing a B-History Object . . . 34

3.3 Implementing a BH&AR Object . . . 40

3.4 Implementing a Collection of Fetch&Add Objects and MRSW Registers . . . 41

**4 A Problem Equivalent to Implementing a $(m, n)$-Queue** **43**

4.1 The Basic-Id-Queue Object Type . . . 44

4.2 Basic-Id-Queue vs. Queue . . . 44

4.3 BH&AR Object vs. B-History Object . . . 45

4.4 Preliminary Negative Results . . . 48

4.4.1 No Implementation when Dequeue Takes Only One Step . . . 49

4.4.2 No Implementation when Enqueue Takes Only One Step . . . 51

**5 Conclusions** **52**

**Bibliography** **54**

**Index** **54**

# Chapter 1

# Introduction

A contemporary multi-processor system consists of a number of processors, a shared memory, and a set of primitive operations by which processors can access the shared memory. This system can be formally modeled by an asynchronous shared-memory distributed system, in which a number of processes (or processors) run at independent speeds and communicate using a collection of shared objects. The asynchronism takes into account the fact that processes may encounter arbitrary delays in executing instructions, such as cache misses or page faults, or they may even crash. A shared object can be viewed as an automaton which moves from state to state as operations are applied on it, according to some sequential specification.

Different systems provide the processes with different collections of objects. Hence, an algorithm for system $\mathcal{S}_1$, using the objects in the collection $\mathcal{O}_1$ provided by $\mathcal{S}_1$, may have to be translated to work in system $\mathcal{S}_2$, with the objects in the collection $\mathcal{O}_2$ provided by $\mathcal{S}_2$. A generic approach which allows us to avoid translating every single algorithm from $\mathcal{S}_1$ to $\mathcal{S}_2$ is to implement the objects in $\mathcal{O}_1$ using those in $\mathcal{O}_2$. In this thesis, we focus on two correctness conditions for an implementation: *linearizability* [HW90] and *wait-freedom* [Her91]. Exact definitions for these notions will be provided later in this chapter. Informally, the former is an intuitive way of saying that the results produced by an implementation must appear as if the simulated operations occurred atomically. The latter is a strong progress condition, saying that no process can be infinitely delayed while executing a simulated operation. In this thesis, every time we discuss (the existence of) an implementation, the attributes wait-free and linearizable will be implicitly assumed, unless explicitly stated otherwise.

Given a target object $O$, and a set of base shared objects in $\mathcal{B}$, there arises the question whether or not $O$ can be implemented from $\mathcal{B}$ in a wait-free and linearizable manner. A general way to answer this question positively is to provide such an implementation. However, negative results require more elaborate techniques [FR03].

An important approach for proving such negative results was developed by Herlihy in [Her91]. To explain it, let $T$ be a target object type and let $\mathcal{T}'$ be a set of object types. If every object $O$ of type $T$ can be implemented from a set of objects with types in $\mathcal{T}'$, we say that the type $T$ can be implemented by the types $\mathcal{T}'$. In that case, every set of objects of type $T$ can be simulated by some set of objects with types in $\mathcal{T}'$. Furthermore, by transitivity of wait-free linearizable implementations, if some other object $O'$ can be implemented from a set of objects of type $T$ and Registers, then $O'$ can be implemented from a set of objects with types in $\mathcal{T}' \cup \{$ Register $\}$.

In his paper [Her91], Herlihy considers the following Consensus problem. Each process starts with an input value and outputs some value, satisfying:

**Agreement** No two processes output different values.

**Validity** Any value output by a process is the input of some process.

**Termination** No process can be infinitely delayed from outputting a value by the actions of other processes.

This problem can be cast into a shared object, $n$-Consensus, in such a way that the Consensus problem can be solved in a system of $n$ processes using objects in $\mathcal{B}$ if and only if there exists an implementation of an $n$-Consensus object from $\mathcal{B}$. Suppose there exists an implementation of $n$-Consensus from a set of objects of type $T$ and Registers. If type $T$ can be implemented from a set of types $\mathcal{T}'$ containing Register, then there exists an implementation of $n$-Consensus from objects with types in $\mathcal{T}'$. Hence, to show that there exists no implementation of some object $O$ of type $T$ from objects with types in $\mathcal{T}'$, it is enough to show that there exists no implementation of $n$-Consensus from objects with types in $\mathcal{T}'$.

Given an object type $T$, we define the *consensus number* of $T$ to be the maximum $n$ such that the Consensus problem among $n$ processes can be solved using objects of type $T$ and Registers. If no such maximum exists, the consensus number of type $T$ is $\infty$. This definition yields the $h_m^r$ hierarchy, which was originally introduced by Jayanti in [Jay93], based on the work of Herlihy [Her91]. We say that a type $T$ is *$n$-universal* if every object $O'$ can be implemented from objects of type $T$ and Registers in a system of $n$ processes. Herlihy proves that a type $T$ is $n$-universal if and only if $T$ has consensus number at least $n$.

Now let $T$ and $T'$ be two object types, such that $n$ is the consensus number of $T$ and $n'$ is the consensus number of type $T'$. If type $T$ can be implemented by the types $T'$ and Register, then the Consensus problem can be solved using objects of type $T'$ and Registers in a system of $n$ processes, hence $n' \geq n$. Equivalently, if $n' < n$, then there exists some object $O$ of type $T$ such that there is no implementation of $O$ from objects of type $T'$ and Registers in a system of more than $n'$ processes.

Let $O_1$ and $O_2$ be two objects of type $T$, which differ only in their initial states, $q_1$ and $q_2$. Let $\mathcal{T}'$ be a set of types. Suppose that there is a finite sequence $\sigma$ of operations on type $T$ that drives $O_1$ from state $q_1$ to state $q_2$. If there exists an implementation of $O_1$ from objects with types in $\mathcal{T}'$, then there exists an implementation of $O_2$ from objects with types in $\mathcal{T}'$, simply because we can directly initialize the objects used in the implementation of $O_1$ to the respective states they reach after the operations in $\sigma$ are simulated by the implementation. An object type is *strongly connected* if every state can be reached from every other state by some finite sequence of operations. All common object types such as Register, Test&Set (with Reset operation), Fetch&Add, Swap, Queue, Stack and Compare&Swap are strongly connected. An object is *oblivious* if every process is allowed to perform on it every operation provided by the object's type. Let $n$-$T$ denote an oblivious object of type $T$ in a system of $n$ processes.

With the above definitions, the following two facts are consequences of Herlihy's results. Let $T$ and $T'$ be two types with consensus numbers $n$ and $n'$, respectively. On the one hand, if $T$ is strongly connected and $n' < n$, then there exists no implementation of an oblivious $T$ object from objects of type $T'$ and Registers in a system of more than $n'$ processes. On the other hand, if $n' \geq n$, then for every object $O$ of type $T$, there exists an implementation of $O$ from objects of type $T'$ and Registers in a system with at most $n'$ processors.

However, if two types $T$ and $T'$ have the same consensus number $n$, it is not known, in general, whether an object $O$ of type $T$ has an implementation from objects of type $T'$ and Registers in a system of more than $n$ processes. In fact, Herlihy leaves as an open problem in [Her91] the question whether objects of the Fetch&Add type and Registers can implement any other object whose type has consensus number 2 in a system of 3 or more processes.

We know from Herlihy's previous work [Her91] that the Register type has consensus number 1, and that Test&Set, Fetch&Add, Swap and Queue types all have consensus number 2. In this thesis, we study the problem of implementing a Queue object from objects with other types which have consensus number 2 and are more commonly provided by real systems, such as Test&Set, Fetch&Add and Swap.

A summary of related work is presented in section 1.1 of this chapter. Some useful mathematical notation is introduced in section 1.2, followed by the formal definitions for our model of computation in section 1.3. Section 1.4 contains the definitions for the shared object types relevant to this thesis.

In Chapter 2, we refute a conjecture made by Li [Li01], and prove that there exists a wait-free linearizable implementation of a Queue object from Fetch&Add, Swap and Register objects, as long as only one process performs enqueue operations.

The work presented in Chapters 3 and 4 is aimed at proving that there exists no wait-free linearizable implementation of an oblivious Queue object from Fetch&Add objects and Registers in a system of at least 3 processes. We cannot use Herlihy's approach, since both Queue and Fetch&Add object types have the same consensus number, 2. Another way of proving this impossibility result is to use an adversarial argument [FR03]. This consists of assuming that such an implementation exists, and eventually building a "bad" execution of this implementation, where either wait-freedom or linearizability is violated. One difficulty with this approach is quantifying the exact information that a process gets when it takes a step in a system of Fetch&Add objects and Registers.

In Chapter 3, we give a new characterization of a system providing Common2 types and Registers. By the results in [AWW93], this system is equivalent with one providing only Fetch&Add objects and Registers. We introduce two new object types, B-History and BH&AR, which have two important properties. On the one hand, a system with only one BH&AR object is as powerful as a system providing countably infinitely many Fetch&Add objects and Registers. Hence the problem of implementing a Queue object from Fetch&Add objects and Registers is equivalent to that of implementing a Queue object from one BH&AR object. On the other hand, in a system with only one object of either the BH&AR or the B-History type, we can exactly quantify the information a process receives as a result of taking a step, and this might facilitate the development of an adversary argument in that system.

In Chapter 4, we introduce a restricted version of the Queue object type, called the Basic-Id-Queue. We then consider the problem of implementing a Basic-Id-Queue object from a B-History object. Although this problem appears to be intuitively simpler than implementing a Queue object from a BH&AR object, we show that they are equivalent when each process accessing the Queue can perform either only Enqueue or only Dequeue operations. In section 4.4, we present two results saying that in some restricted cases, there is no wait-free linearizable implementation of a Basic-Id-Queue object from a B-History object. Several conclusions are drawn in Chapter 5.

## 1.1 Related Work

In [AWW93], Afek *et al.* study implementations of objects with types that have consensus number 2 from other types with consensus number 2. To explain their results, we say that an object type $T$ is a *read-modify-write* (RMW) type if the result of every operation provided by $T$ is the state of the object to which it is applied prior to the occurrence of that operation. The RMW type $T$ is *commutative* if for every two operations $op, op'$ of $T$ and every state $q$ of $T$, the state resulting from the application of $op$ starting from $q$ followed by $op'$ is the same as the state resulting from the application of $op'$ starting from $q$ followed by $op$. The RMW type $T$ is *overwriting* if for every two operations $op, op'$ of $T$ and every state $q$ of $T$, the state resulting from the application of $op$ starting from $q$ followed by $op'$ the same as the state resulting from the application of $op'$ starting from $q$. For example, the Fetch&Add and Fetch&Increment types are commutative RMW types, the Test&Set and Swap types are overwriting RMW types, and the Compare&Swap type is a RMW type which is neither commutative nor overwriting. Section 1.4 contains formal definitions of these common object types.

Let *Common2* denote the set of all commutative or overwriting RMW types with consensus number more than 1. We know from [Her91] that all types in Common2 have consensus number exactly 2. For example, Test&Set, Fetch&Add, Fetch&Increment and Swap are all in Common2. We say that an object $O$ of type $T$ is 2-ported if only two processes are allowed to apply operations to it. In practice, distributed systems might provide only 2-ported objects of some particular type.

Afek *et al.* showed in [AWW93] that for every object $O$ of some type $T$ in Common2 and every type $T'$ of consensus number 2, there exists an implementation of $O$ from 2-ported objects of type $T'$ and Registers. Hence, every type $T$ in Common2 can be implemented from any type $T'$ of consensus number 2 and Register.

Let *Basic2* denote the set of types of consensus number 2 which can be implemented from some type in Common2 and Registers. Hence, all types in Common2 are also in Basic2. It is not known if Basic2 consists of all types of consensus number 2. In this thesis, we investigate the question whether the Queue object type is in Basic2. Since the Queue object type is strongly connected, and given the results in [AWW93], the Queue type is in Basic2 if and only if there exists a wait-free linearizable implementation of an initially empty oblivious $n$-Queue object from Fetch&Add objects and Registers, for all $n$.

Jayanti and Toueg showed in [JT92] that for any $n$, there exist types which have consensus number $n$. They introduce an object type, $n$-Bounded Peek-Queue, which is a queue with at most $n$ values supporting a Peek operation instead of a Dequeue operation, and this type is shown to have consensus number $n$. McCrickard proves in [McC94] that the 2-Bounded Peek-Queue, which is not a read-modify-write type, can be implemented from the Test&Set and Register types, and therefore, that 2-Bounded Peek-Queue is in Basic2. Hence, Common2 is strictly contained in Basic2.

In [HW90], Herlihy and Wing presented an implementation for an oblivious $n$-Limited-Queue object from Fetch&Add and Swap objects. The Limited-Queue object type is similar to the Queue object type with the exception that Dequeue operations are not defined when the queue is in the empty state. However, Herlihy and Wing use a weaker progress condition instead of wait-freedom: an implementation is *non-blocking* if as long as a process is infinitely delayed, other processes complete infinitely many simulated operations. The implementation in [HW90] is linearizable and non-blocking, but not wait-free.

In [Li01], Li modifies the implementation in [HW90] and obtains a non-blocking linearizable implementation of an oblivious $n$-Queue object from Common2 objects and Registers. Furthermore, Li observes that the original implementation in [HW90] is wait-free if only one process can perform Dequeue operations. In [Her91], Herlihy showed that in a system of $n$ processes, any object can be implemented from objects of type $n$-Consensus and Registers. Using ideas from Herlihy's construction, Li modifies the implementation in [HW90] and obtains a wait-free linearizable implementation of a Queue object from Common2 objects and Registers, as long as at most two processes can perform Dequeue operations. The bound on the number of steps performed during a dequeue operation in the latter implementation varies with the number of enqueue processes.

The following two Conjectures made by Li are relevant to this thesis:

**Conjecture 1.1** ([Li01], Conjecture 5.1)**.** *There is no wait-free linearizable implementation of a Queue object from Common2 objects and Registers if more than two processes can perform dequeue operations.*

In this thesis, we refute the above conjecture, by providing, in Chapter 2, a wait-free linearizable implementation of a Queue object from Common2 objects and Registers, when only one process can perform enqueue operations, and any number of processes can perform dequeue operations.

**Conjecture 1.2** ([Li01], Conjecture 5.3)**.** *There is no wait-free linearizable implementation of an oblivious $n$-Queue object from Common2 objects and Registers, for any $n > 2$.*

The work in Chapters 3 and 4 of this thesis is directed toward a proof of this Conjecture. Although our objective was not achieved, we hope that the results in Chapters 3 and 4 will prove useful in completing the proof of this Conjecture.

## 1.2 Mathematical Notation

For any set $E$, let $E^*$ denote the set of all finite sequences of elements in $E$. Throughout the entire thesis, $\lambda$ will denote the empty sequence. For $\sigma \in E^*$, let $\sigma[i]$ denote the $i$-th element in the sequence $\sigma$, and let $|\sigma|$ denote the number of elements in $\sigma$. For $\sigma \in E^*$ and $x \in E$, let $\sigma \cdot x$ denote the finite sequence which is equal to $\sigma$ but has one more element, $x$, at its end. Let $count(\sigma, x)$ denote the number of occurrences of $x$ in $\sigma$. For $k \leq |\sigma|$, let $prefix(\sigma, k)$ denote the prefix of $\sigma$ of length $k$. In this thesis, whenever $\sigma$ is a sequence and we mention $\sigma[i]$, we implicitly assume that $\sigma$ contains an $i$-th element. Any sequence $\sigma$ begins with index 1, $\sigma[1]$, unless we explicitly say it begins at index 0. We write $X = (y_1, y_2, \ldots)$ when $X$ is the sequence defined by $X[i] = y_i$.

Let $\mathbb{N}$ denote the set of positive integers $\{1, 2, \ldots\}$, and let $\mathbb{Z}$ denote the set of all integers. For any $a, b \in \mathbb{Z}$ let $[a, b]$ denote the set $\{a, \ldots, b\}$ when $a \leq b$, and $\emptyset$ otherwise. For any set $E$, any $\sigma \in E^*$ and any $A \subseteq [1, |\sigma|]$, define $\sigma|_A$ to be the subsequence of $\sigma$ determined by the elements at positions in $A$. Let $S_n$ denote the set of permutations of $n$ elements. For a permutation $\pi \in S_{|\sigma|}$, let $\pi(\sigma)$ be the sequence obtained by permuting the elements in $\sigma$ according to $\pi$. Formally, for any $i \in [1, |\sigma|]$, $\pi(\sigma)[i] \overset{def}{=} \sigma(\pi[i])$. For two sequences $\sigma, \sigma' \in E^*$, we say that $\sigma'$ is a permutation of $\sigma$ if there exists $\pi \in S_{|\sigma|}$ such that $\sigma' = \pi(\sigma)$.

## 1.3 System Model Definitions

The system we consider is an asynchronous shared-memory distributed system. It consists of $N$ concurrent processes, and a collection of shared objects. Processes execute sequential programs and communicate by accessing shared objects. During a step, a process performs an operation on a certain shared object and receives a response from that object.

**Object Types** The type of an object specifies how that object reacts to the operations applied on it by the processes in the system. Formally, an *object type* $T$ is a tuple $(Q_T, OP_T, RES_T, \delta_T)$ where $Q_T$ is a set of states, $OP_T$ is a set of operations, $RES_T$ is a set of responses, and $\delta_T : Q_T \times OP_T \rightarrow Q_T \times RES_T$ is a state transition function. If $\delta_T(q, op) = (q', res)$, then whenever an object $O$ of type $T$ is in state $q$ and operation $op$ is applied to it, the object moves to state $q'$ and $res$ will be returned as response to the operation. In this thesis, we are only considering *deterministic* object types and that is why, in contrast to definitions elsewhere, we have chosen $\delta$ to be a function. We sometimes parametrize the definition of an object type $T$ using a given set of values $V$, and write $T\langle V\rangle$.

To illustrate the definition, we give here the specification of the Fetch&Add and Register object types. The object type Fetch&Add is defined by $Q_{F\&A} = \mathbb{Z}$, $OP_{F\&A} = \{Fetch\&Add(x) : x \in \mathbb{Z}\}$, $RES_{F\&A} = \mathbb{Z}$ and for all $x, y \in \mathbb{Z}$, $\delta_{F\&A}(x, Fetch\&Add(y)) = (x + y, x)$. The Register$\langle V\rangle$ type has $Q_{REG\langle V\rangle} = V$, $OP_{REG\langle V\rangle} = \{Read\} \cup \{Write(x) : x \in V\}$, $RES_{REG\langle V\rangle} = V \cup \{OK\}$, for all $x \in V$, $\delta_{REG\langle V\rangle}(x, Read) = (x, x)$ and for all $x, y \in V$, $\delta_{REG\langle V\rangle}(x, Write(y)) = (y, OK)$.

**Objects** An *object* is an automaton which moves from state to state according to a transition function as operations are applied to it. Each shared object has a name, e.g.. $O$, and the definition of an object $(O, T, q_O, b_O)$ includes its type $T$, its initial state $q_O$ (which is an element of the set of states of its object type) and its binding function $b_O$. The latter specifies what operations each process in the system is allowed to perform on this object. Formally, for an object $O$ of type $T$, the function $b_O$ maps each process $P_i$ in the system, to a set of operations $b_O(P_i) \subseteq OP_T$ which $P_i$ is allowed to perform on $O$.

An object is *oblivious* if any process may perform on this object any operation that is permitted by its type. In other words, object $O$ of type $T$ is oblivious if for all processes $P_i$, $b_O(P_i) = OP_T$. An object which is not oblivious is *non-oblivious.*

We illustrate this terminology by defining two objects. Consider a system of two processes, $P_1$ and $P_2$, and a given set of values $V$. An oblivious Fetch&Add object $O$ for this system has the Fetch&Add object type defined above, an initial state $q_O \in \mathbb{Z}$ and bindings $b_O(P_1) = b_O(P_2) = OP_{F\&A}$. Now consider a system of five processes, $P_1, \ldots, P_5$, and a given set of values $V$. For this system, a (non-oblivious) Single-Reader Single-Writer (SRSW) Register$\langle V \rangle$ object $O'$ which allows $P_1$ to read it and $P_4$ to write it has the Register$\langle V \rangle$ object type defined above, an initial state $q_{O'} \in Q_{REG\langle V \rangle}$ and bindings $b_{O'}(P_1) = \{Read\}$, $b_{O'}(P_4) = \{Write(x) : x \in V\}$ and $b_{O'}(P_2) = b_{O'}(P_3) = b_{O'}(P_5) = \emptyset$.

In general, the distributed system provides a collection $\mathcal{B}$ of shared objects. In this thesis, we always assume that a system provides at most *countably infinitely* many shared objects, so that we can index their names by positive integers, $\mathcal{B} = \{O_1, O_2, \ldots\}$. Whenever we say that a system provides objects of certain object types, we mean that the system provides a finite or countably infinite collection of shared objects, each of which has one of the listed object types.

Whenever we say that a system provides objects of object type $T$, with no explicit reference to the bindings of those objects, we assume that any object of type $T$ will have a default set of bindings. This default set of bindings is specific to the object type $T$ and, in most cases, these bindings are as unrestrictive as possible (i.e. the objects of that type are oblivious). However, some of the object types defined in this thesis have a more restrictive set of default bindings, and these will be introduced along with the definitions of those object types.

Whenever we say that a system provides objects of type $T$ with no explicit reference to the set of values parametrizing $T$, we mean that an algorithm for this system can use objects of type $T\langle V \rangle$, for any finite or countably infinite set $V$.

To illustrate this terminology, when we say that a system provides Fetch&Add objects and Multi-Reader Single-Writer (MRSW) Registers, we mean that the system provides a (finite or) countably infinite collection of shared objects, each of those objects has either the Fetch&Add or the Register$\langle V \rangle$ object type for some finite or countably infinite set $V$, each Fetch&Add object is oblivious, and each Register object has bindings which allow one process to perform $Write$ operations and any process to perform *Read* operations.

**Processes** A system contains finitely many processes. Throughout this thesis, $N$ will denote the number of processes in our distributed system. We might talk about distributed systems providing different collections of shared objects, but the number of processes in any of these systems will always be $N$. The processes in our system have names, $P_1, \ldots, P_N$.

A *process* $P_i$ is a deterministic sequential thread of control which starts from a fixed initial state and interacts with other processes by means of the shared memory. Processes perform local computation and access the shared objects. An atomic operation, called a *step*, is a tuple $(P_i, O, op, res)$ and signifies that process $P_i$ atomically applies operation $op$ to a shared object $O$ and receives a response $res$ to that operation. The process then updates its internal state based on $res$. Processes can be more formally modeled as I/O automata which interact with objects via input and output events [Lyn96]. However, we feel that extensive formalism is not necessary for the purposes of this thesis.

Processes take steps one at a time, according to the order specified by a scheduler. In this thesis, we impose no fairness or other restrictions on the scheduler. Within this setting, the crash of some process can be modeled by having the scheduler not allocate any more steps to that process. Between steps, a process can perform an arbitrary amount of local computation. This assumption captures the fact that process $P_i$ cannot get any information about the computation of process $P_{i'}$ except for what is conveyed by the operations $P_{i'}$ performs on shared objects.

**Configurations, Steps, System History** A *configuration* of the system consists of the state of each process and the state of each shared object. The *initial configuration* of the system has every process and every object in its initial state. If the system is in configuration $C$ and process $P_i$ is scheduled to take a step, then a step of the form $s = (P_i, O, op, res)$ occurs. Let the type of object $O$ be $T$, let the state of object $O$ in $C$ be $q$, and let $\delta_T(q, op) = (q', res')$. We say the *step $s$ is legal from $C$* if $op \in b_O(P_i)$ and $res = res'$, that is, $P_i$ is allowed to perform operation $op$ on $O$ and the response $P_i$ gets to the operation is consistent with the state transition function of type $T$. Let $C' = C \cdot s$ denote the unique configuration the system enters after step $s$ is applied from $C$. $C'$ can only differ from $C$ in the internal state of process $P_i$ and in the state of object $O$, which is now $q'$.

A *system history* is a finite sequence of steps. We inductively define when *system history $S$ is legal from configuration $C$*, as well as the meaning of $C \cdot S$. A system history $S$ of length 1 is legal from $C$ if the unique step $S[1]$ in $S$ is legal from $C$. In this case, we define $C \cdot S$ to be $C \cdot S[1]$. For a system history $S$ of length $i > 1$, let $S'$ be $S$ without its last step $s$. So $S'$ has length $i - 1 \geq 1$. Now $S$ is legal from $C$ if $S'$ is legal from $C$ and $s$ is legal from $C \cdot S'$. We also define $C \cdot S$ to be $(C \cdot S') \cdot s$. A system history is *legal* if it is legal from the initial configuration of the system.

**Object Histories** A *history* of an object $O$ of type $T$ is a finite sequence of pairs $H \in (OP_T \times RES_T)^*$. The history $H = ((op_1, res_1), (op_2, res_2), \ldots)$ of object $O$ is *legal* if there exists a sequence $Q = (q_0, q_1, \ldots) \in Q_T^*$ of states of $O$, such that $q_0 = q_O$ (the initial state of $O$) and for $i \geq 1, \delta_T(q_{i-1}, op_i) = (q_i, res_i)$. Notice that a sequence of states $Q$ starts with $Q[0]$ instead of $Q[1]$. Any system history $S$ determines a history $H(S, O)$ for every shared object $O$ available in the system. From the definitions, an easy inductive argument shows that if $S$ is legal then $H(S, O)$ is legal.

**Implementations** Suppose that our system provides a certain collection $\mathcal{B}$ of shared objects, referred to as *base objects*. Furthermore, suppose that we have an algorithm which was written for some other distributed system, using some object $O \notin \mathcal{B}$ not available in our system. Notice that our system might provide objects of the same object *type* as $O$, but with incompatible bindings. As an example, we might want to use a Multi-Reader Multi-Writer Register when our system only provides Single-Reader Single-Writer Registers. In order to be able to use the original algorithm, we will need to implement the object $O$ in our system. Informally, an implementation is a way for our system to simulate the object $O$ using only available objects from $\mathcal{B}$.

Let $(O, T, q_O, b_O)$ be an object of type $T = (Q_T, OP_T, RES_T, \delta_T)$, with initial state $q_O \in Q_T$ and bindings $b_O$. An *implementation* of $O$ from base objects in $\mathcal{B}$ is specified by giving, for each process $P_i$ in the system and each operation $op \in b_O(P_i)$, an *access procedure* $P_i\!:\!op$ consisting of local computation and accesses to shared objects in $\mathcal{B}$, respecting the bindings of those objects, and returning a value $res \in RES_T$.

Consider the situation when $P_i$ needs to apply $op \in b_O(P_i)$ on $O$. In a system which provides the object $O$, $P_i$ will apply it atomically in a step of the form $(P_i, O, op, res)$, obtaining the result $res$. In a system which does not directly provide $O$ but in which an implementation of $O$ is available, $P_i$ will execute the access procedure $P_i\!:\!op$, which returns $res \in RES_T$ and is taken by $P_i$ to be the response of $O$ to its operation.

Access procedures on the implemented object consist of local computation and accesses to shared objects in $\mathcal{B}$, that is, steps. One access procedure may require the process performing it to take two or more steps before returning. Since our system only provides for the atomicity of steps, executions of access procedures for $O$ may overlap with one another, because their steps may be interleaved. We say that a step is part of the execution of an access procedure $P_i : op$ when $P_i$ performs that step while executing $P_i\!:\!op$.

**Runs** A system history can only capture the correctness of individual steps taken by processes in the system. However, in order for an implementation to be useful, we would like to argue about what happens when processes take their steps while executing access procedures. To do that, we define a *run* $R$ of the implementation of $O$ to be a pair $(\psi, S)$ where:

- $\psi$ maps each process $P_i$ to a finite sequence of operations $\psi(P_i) \in (b_O(P_i))^*$ applied by $P_i$ on the implemented object $O$ in run $R$.

- $S = (s_1, s_2, \ldots)$ is a legal system history, in which each process $P_i$ starts from its initial state and sequentially executes access procedures of the form $P_i : op$ in the same order the operations $op$ appear in $\psi(P_i)$. In particular, $P_i$ will complete the execution of one access procedure before starting the next access procedure.

- Every process $P_i$ completes the execution of every access procedure corresponding to every operation in $\psi(P_i)$ except possibly the last one and, furthermore, $P_i$ executes at least one step from the access procedure corresponding to the last operation in $\psi(P_i)$. Notice that $P_i$ may or may not complete the execution of the access procedure corresponding to the last operation in $\psi(P_i)$.

**Procedure Instances** Given a run $R = (\psi, S)$, one can partition the subsequence of steps in $S$ taken by any process $P_i$ into contiguous blocks, such that each block contains the steps that $P_i$ is taking while executing some access procedure $P_i : op$. We define an *instance of the access procedure $P_i\!:\!op$ in $R$* to be the set of indices in $S$ of the steps in one such block. Although formally defined as a set of indices of steps, we prefer to think of a procedure instance in terms of the steps it contains. We say that a procedure instance $\alpha$ contains the $i$-th step $s_i$ in $S$ if $i \in \alpha$. The first step of instance $\alpha$ is $s_{\min(\alpha)}$ and its last step is $s_{\max(\alpha)}$. Let $Instances(R)$ denote the set of all procedure instances in the run $R$. Notice that there is one procedure instance in $Instances(R)$ for every occurrence of an operation in any of the sequences $\psi(P_i)$. Let $process(\alpha)$ denote the process executing the procedure instance $\alpha$ and let $operation(\alpha)$ denote the operation being applied by the access procedure instance $\alpha$. With this notation, whenever $\alpha$ is an instance of the access procedure $P_i\!:\!op$, we have $process(\alpha) = P_i$ and $operation(\alpha) = op$.

Notice that two different instances of the same access procedure need not contain the same number of steps. We say that an instance $\alpha$ of the access procedure $P_i\!:\!op$ is *complete* if, after $P_i$ executes the last step $s_{\max(\alpha)}$ of this instance, the access procedure contains only local computation before returning a result $res$ to operation $op$. Otherwise, the procedure instance is *incomplete*. Notice that if $\alpha$ is incomplete, $s_{\max(\alpha)}$ is *not* the last step that access procedure would execute if it were complete, but simply the last step of $\alpha$ appearing in the system history $S$. In a run, only the last instance of an access procedure executed by $P_i$ in $R$ can be incomplete. For any complete procedure instance, $R$ determines what result it returns. Let $Complete(R)$ denote the set of procedure instances which are complete in $R$. For $\alpha \in Complete(R)$, let $result(\alpha)$ denote the result returned by $\alpha$ in $R$.

The run $R$ induces a partial order $\leq_R$ on the procedure instances it contains: for two procedure instances $\alpha, \alpha' \in Instances(R)$, we write $\alpha <_R \alpha'$ whenever $\alpha$ is complete and $\max(\alpha) < \min(\alpha')$, that is, the last step of $\alpha$ precedes the first step of $\alpha'$. We say that two different procedure instances are *concurrent* in $R$ if neither $\alpha <_R \alpha'$ nor $\alpha' <_R \alpha$. Since processes act sequentially, two procedure instances can only be concurrent if they are performed by different processes.

Let $step(\alpha, i)$ denote the index in $S$ of the $i$-th step of instance $\alpha$. We will only use this notation when $\alpha$ contains at least $i$ steps. As an example, we always have $step(\alpha, 1) = \min(\alpha)$.

**Linearizability** In order for an implementation to be useful, we need to impose some restrictions on what responses the access procedures may return. The only correctness condition we consider in this thesis is *linearizability* [HW90]. Informally, this condition states that whenever $P_i$ executes the access procedure $P_i\!:\!op$, no matter how the steps in the execution of $P_i\!:\!op$ are interleaved with those in the executions of other access procedures by other processes, the execution of $P_i : op$ has to appear to be atomic, occurring at some moment between its first and last steps. We formalize this notion below.

**Definition 1.1.** A run $R$ of the implementation of $O$ is *linearizable* if there exist:

- a set $Occur(R)$ of procedure instances such that $Complete(R) \subseteq Occur(R) \subseteq Instances(R)$;
- a linear order $\prec$ on the procedure instances in $Occur(R)$, so that $\alpha_1 \prec \alpha_2 \prec \ldots$ and in general, $\alpha_i$ is the $i$-th procedure instance in $Occur(R)$ with respect to $\preceq$; and
- a history $H = ((op_1, res_1), (op_2, res_2), \ldots)$ of object $O$

satisfying the following requirements, for all $i$ and $j$:

1. $op_i = operation(\alpha_i)$;
2. if $\alpha_i \in Complete(R)$, then $res_i = result(\alpha_i)$;
3. if $\alpha_i <_R \alpha_j$ (that is, $\max(\alpha_i) < \min(\alpha_j)$ and $\alpha_i \in Complete(R)$), then $i < j$; and
4. $H$ is legal.

The implementation of an object is *linearizable* if all its runs are linearizable.

**Wait-freedom** In addition to requiring the access procedures to return consistent values, we would also like to know that there is some *progress* in the system: under certain assumptions, some access procedures are eventually completed. The only progress condition we consider in this thesis is *wait-freedom* [Her91].

An implementation is said to be *wait-free* if, for every process $P_i$, every operation $op \in b_O(P_i)$ and every run $R$ in which $P_i$ completed all its operations, if $P_i$ starts executing the access procedure $P_i : op$ from the system configuration at the end of run $R$, $P_i$ will eventually complete the execution of this access procedure within finitely many steps, regardless of the steps performed by other processes in the system. In our formal definitions we only allow for finite runs. Hence, we will consider the following equivalent condition:

**Proposition 1.1.** *An implementation of an object $O$ is* not wait-free *if and only if there exists a process $P_i$ and an infinite family $R_1, R_2, \ldots$ of runs of this implementation, where $R_k = (\psi_k, S_k)$, satisfying the following conditions, for all $k \geq 1$:*

- *$R_{k+1}$ is an extension of $R_k$, that is, $S_k$ is a prefix of $S_{k+1}$ and for all processes $P_j$, $\psi_k(P_j)$ is a prefix of $\psi_{k+1}(P_j)$;*
- *$S_{k+1}$ contains more steps by process $P_i$ than $S_k$; and*
- *$\psi_{k+1}(P_i) = \psi_k(P_i)$.*

We know that in any run, the steps that $P_i$ is taking have to be part of the execution of an access procedure on the implemented object. Hence, the condition in the Proposition implies that there is no bound on the number of steps that $P_i$ is taking in order to complete the execution of its last access procedure.

An implementation is *b-bounded wait-free* if, in any run, no instance of an access procedure can include more than $b$ steps. Notice that bounded wait-freedom is a stronger condition than wait-freedom.

**Implementation of a Collection of Objects** Let $\mathcal{A} = \{A_1, A_2, \ldots\}$ be a (finite or) countably infinite collection of shared objects. For all $A_i \in \mathcal{A}$, let $T_i$ be the object type of $A_i$, $q_{A_i}$ be the initial state of $A_i$ and $b_{A_i}$ be the function giving the bindings of $A_i$. We define the shared object type $\mathit{Type}(\mathcal{A})$ to be the Cartesian product of the object types $T_1, T_2, \ldots$:

- $Q_{\mathit{Type}(\mathcal{A})} \overset{def}{=} Q_{T_1} \times Q_{T_2} \times \ldots$;
- $OP_{\mathit{Type}(\mathcal{A})} \overset{def}{=} \{\langle A_i, op\rangle : A_i \in \mathcal{A} \text{ and } op \in OP_{T_i}\}$;
- $RES_{\mathit{Type}(\mathcal{A})} \overset{def}{=} RES_{T_1} \cup RES_{T_2} \cup \ldots$; and
- for all $q = (q_1, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots) \in Q_{\mathit{Type}(\mathcal{A})}$, and $\langle A_i, op\rangle \in OP_{\mathit{Type}(\mathcal{A})}$, we have

  $$\delta_{\mathit{Type}(\mathcal{A})}(q, \langle A_i, op\rangle) \overset{def}{=} (q', res)$$

  where $q' = (q_1, \ldots, q_{i-1}, q'_i, q_{i+1}, \ldots)$ and $\delta_{T_i}(q_i, op) = (q'_i, res)$.

We then define $\mathit{Object}(\mathcal{A})$ to be the shared object with:

- object type $\mathit{Type}(\mathcal{A})$;
- initial state $q_{\mathit{Object}(\mathcal{A})} \overset{def}{=} (q_{A_1}, q_{A_2}, \ldots)$; and
- bindings $b_{\mathit{Object}(\mathcal{A})}(P_j) \overset{def}{=} \{\langle A_i, op\rangle \in OP_{\mathit{Type}(\mathcal{A})} : A_i \in \mathcal{A} \text{ and } op \in b_{A_i}(P_j)\}$, for all processes $P_j$.

Let $\mathcal{A}$ be a collection of shared objects, let $\mathcal{S}_1$ be a distributed system of $N$ processes providing the objects in $\mathcal{A}$ and let $\mathcal{S}_2$ be a distributed system of $N$ processes providing only one object, $\mathit{Object}(\mathcal{A})$. Let $P_j$ be a process, let $A_i \in \mathcal{A}$ be a shared object, and let $op \in OP_{T_i}$ be an operation allowed by the type of $A_i$. Whenever $P_j$ applies $op$ on $A_i$ in $\mathcal{S}_1$, the same result can be achieved in $\mathcal{S}_2$ by having $P_j$ apply $\langle A_i, op\rangle$ on $\mathit{Object}(\mathcal{A})$. From the definition of $\mathit{Object}(\mathcal{A})$, it is easy to see that any legal system history in $\mathcal{S}_1$ has a corresponding legal system history in $\mathcal{S}_2$, where every step of the form $(P_j, A_i, op, res)$ is replaced by $(P_j, \mathit{Object}(\mathcal{A}), \langle A_i, op\rangle, res)$. Therefore, there exists an algorithm for some problem in system $\mathcal{S}_1$ if and only if there exists an algorithm for the same problem in system $\mathcal{S}_2$. In particular, for any shared object $O$, there exists a wait-free linearizable implementation of $O$ from $\mathcal{A}$ if and only if there exists a wait-free linearizable implementation of $O$ from $\mathit{Object}(\mathcal{A})$.

We say that there exists an implementation of a set of objects $\mathcal{A}$ from a set of base objects $\mathcal{B}$ if and only if there exists an implementation of $\mathit{Object}(\mathcal{A})$ from $\mathcal{B}$.

**Equally Powerful Collections of Objects** We say that two collections of shared objects, $\mathcal{A}_1$ and $\mathcal{A}_2$ are *equally powerful* if and only if they can implement each other. That is, there exists a wait-free linearizable implementation of $\mathit{Object}(\mathcal{A}_1)$ from $\mathcal{A}_2$, and there exists a wait-free linearizable implementation of $\mathit{Object}(\mathcal{A}_2)$ form $\mathcal{A}_1$.

## 1.4 Shared Object Types

The following object types will be used in this thesis:

**Register** For any set of values $V$, the Register$\langle V\rangle$ object type is defined by:

- $Q_{REG\langle V\rangle} = V$;
- $OP_{REG\langle V\rangle} = \{\text{Read}\} \cup \{\text{Write}(x) : x \in V\}$;
- $RES_{REG\langle V\rangle} = V \cup \{OK\}$ (where $OK \notin V$);
- $\delta_{REG\langle V\rangle}(x,\text{Read}) = (x, x)$, for all $x \in V$; and
- $\delta_{REG\langle V\rangle}(x,\text{Write}(y)) = (y, OK)$, for all $x, y \in V$.

The possible bindings for objects of this type are SRSW (Single-Reader Single-Writer), MRSW (Multi-Reader Single-Writer) and MRMW (Multi-Reader Multi-Writer). The attribute Single means that only one process can apply that kind of operation(s), while the attribute Multi means that all processes can apply that kind of operation(s). Unless specified otherwise, any object of this type is oblivious, that is, MRMW.

**Append-Register** For any set of values $V$, the Append-Register$\langle V\rangle$ object type is defined by:

- $Q_{AP-REG\langle V\rangle} = V^*$;
- $OP_{AP-REG\langle V\rangle} = \{\text{Read}\} \cup \{\text{Append}(x) : x \in V\}$;
- $RES_{AP-REG\langle V\rangle} = V^* \cup \{OK\}$;
- $\delta_{AP-REG\langle V\rangle}(\sigma,\text{Read}) = (\sigma, \sigma)$, for all $\sigma \in V^*$; and
- $\delta_{AP-REG\langle V\rangle}(\sigma,\text{Append}(x)) = (\sigma \cdot x, OK)$, for all $\sigma \in V^*$ and $x \in V$.

In this thesis, any object of this type will have initial state $\lambda$ (the empty sequence) and MRSW bindings.

**Fetch&Add** The Fetch&Add object type is defined by:

- $Q_{F\&A} = \mathbb{Z}$;
- $OP_{F\&A} = \{\text{Fetch\&Add}(x) : x \in \mathbb{Z}\}$;
- $RES_{F\&A} = \mathbb{Z}$; and
- $\delta_{F\&A}(x,\text{Fetch\&Add}(y)) = (x + y, x)$, for all $x, y \in \mathbb{Z}$.

Unless explicitly stated otherwise, every object of this type will have initial state 0 and will be oblivious.

**Fetch&Increment** The Fetch&Increment object type is defined by:

- $Q_{F\&I} = \mathbb{Z}$;
- $OP_{F\&I} = \{\text{Fetch\&Increment}\}$;
- $RES_{F\&I} = \mathbb{Z}$; and
- $\delta_{F\&I}(x,\text{Fetch\&Increment}) = (x + 1, x)$, for all $x \in \mathbb{Z}$.

Unless explicitly stated otherwise, every object of this type will have initial state 0 and will be oblivious.

**Swap** For any set of values $V$, the Swap$\langle V\rangle$ object type is defined by:

- $Q_{SWAP\langle V\rangle} = V$;
- $OP_{SWAP\langle V\rangle} = \{\text{Swap}(x) : x \in V\}$;
- $RES_{SWAP\langle V\rangle} = V$; and
- $\delta_{SWAP\langle V\rangle}(x,\text{Swap}(y)) = (y, x)$, for all $x, y \in V$.

Unless explicitly stated otherwise, every object of this type will be oblivious.

**Compare&Swap** For any set of values $V$, the Compare&Swap$\langle V\rangle$ object type is defined by:

- $Q_{C\&S\langle V\rangle} = V$;
- $OP_{C\&S\langle V\rangle} = \{\text{Compare\&Swap}(x, y) : x, y \in V\}$;
- $RES_{C\&S\langle V\rangle} = V$;
- $\delta_{C\&S\langle V\rangle}(x,\text{Compare\&Swap}(x, y)) = (y, x)$, for all $x, y \in V$; and
- $\delta_{C\&S\langle V\rangle}(v,\text{Compare\&Swap}(x, y)) = (v, v)$, for all $x, y, v \in V$ with $x \neq v$.

This type is provided for reference purposes only.

**Test&Set** The Test&Set object type is defined by:

- $Q_{T\&S} = \{0, 1\}$;
- $OP_{T\&S} = \{\text{Test\&Set, Reset}\}$;
- $RES_{T\&S} = \{0, 1, OK\}$;
- $\delta_{T\&S}(x,\text{Test\&Set}) = (1, x)$, for all $x \in \{0, 1\}$; and
- $\delta_{T\&S}(x,\text{Reset}) = (0, OK)$, for all $x \in \{0, 1\}$.

Unless explicitly stated otherwise, any object of this type will be oblivious.

**Queue** For any set of values $V$, the Queue$\langle V\rangle$ object type is defined by:

- $Q_{QUEUE} = V^*$;
- $OP_{QUEUE} = \{\text{Enqueue}(x) : x \in V\} \cup \{\text{Dequeue}\}$;
- $RES_{QUEUE} = V \cup \{\varepsilon, OK\}$ (where $OK, \varepsilon \notin V$);
- $\delta_{QUEUE}(\sigma,\text{Enqueue}(x)) = (\sigma \cdot x, OK)$, for all $x \in V$ and $\sigma \in V^*$;
- $\delta_{QUEUE}(\lambda,\text{Dequeue}) = (\lambda, \varepsilon)$; and
- $\delta_{QUEUE}(x \cdot \sigma,\text{Dequeue}) = (\sigma, x)$, for all $x \in V$ and $\sigma \in V^*$.

In this thesis, any object of the Queue type will have initial state $\lambda$, corresponding to an empty queue, unless explicitly stated otherwise. For any non-negative integers $m, n, p$, an $(m, n, p)$-Queue object for a system of $N = m + n + p$ processes has bindings which allow $m$ processes to perform only enqueue operations, $n$ other processes to perform only dequeue operations and the remaining $p$ processes to perform both kinds of operations. We write $N$-Queue for a $(0, 0, N)$-Queue object, which is, in fact, an oblivious Queue object in a system of $N$ processes. We write $(m, n)$-Queue for an $(m, n, 0)$-Queue object.

# Chapter 2

# A $(1, n)$-Queue implementation

Li conjectures in his thesis that there is no wait-free linearizable implementation of a $n$-Queue object from Common2 objects and Registers whenever $n \geq 3$ [Li01, Conjecture 5.3]. In an attempt to narrow down the difficulty of the problem, Li proposes another conjecture, stating that there is no wait-free linearizable implementation of a (1,3)-Queue from Common2 objects [Li01, Conjecture 5.1]. This would imply that three concurrent dequeue operations are impossible to synchronize in a wait-free manner using Common2 objects, and the former conjecture would follow. As we will see below, the latter conjecture does not hold.

In this chapter, we prove the following:

**Theorem 2.1.** *Let $V$ be a finite or countably infinite set. For any $n$, there exists a wait-free linearizable implementation of a $(1, n)$-Queue$\langle V \rangle$ object from Common2 objects and Registers.*

We achieve this by presenting Algorithm 1 in section 2.1, which is a 3-bounded wait-free linearizable implementation of a $(1, n)$-Queue$\langle V \rangle$ object from Common2 objects and Registers. Section 2.2 contains the proof of correctness of this algorithm.

## 2.1 The algorithm

Algorithm 1 implements a $(1, n)$-Queue$\langle V \rangle$ from Basic2 objects and Registers. Let us denote the process performing enqueue operations by $E$ and the $n$ processes performing dequeue operations by $D_1, \ldots, D_n$. We are using a one dimensional array $HEAD$ of Fetch&Increment objects and a two dimensional array $ITEMS$ of Swap objects together with one Multi-Reader Single-Writer Register $ROW$. Both arrays, $ROW$ and $ITEMS$, are infinite.

Informally, the algorithm works as follows. The cells in the two dimensional array $ITEMS$ are initialized to a default value, $\perp \notin V$. Whenever they are accessed during an enqueue procedure, their value is updated to contain the respective element to be enqueued. Whenever they are accessed by a dequeue procedure, their value is updated to contain $\top \notin V$. By design, each cell in the array $ITEMS$ will be used at most once by an enqueue operation, and at most once by a dequeue operation.

In order to perform a dequeue operation, process $D_i$ reads from $ROW$ the value of the active row in the two-dimensional array $ITEMS$. This is the row which was last used to enqueue a value by an enqueue procedure which has already finished. Having obtained the value of this row in its local variable *deq_row*, process $D_i$ selects the column *head* of a cell to query on this row using the Fetch&Increment object $HEAD[\mathit{deq\_row}]$. It then proceeds to query the Swap object $ITEMS[\mathit{deq\_row}, \mathit{head}]$ and update its value to $\top$. If the value retrieved is not $\perp$, then some value to be enqueued was written in this location and process $D_i$ dequeues that value. Otherwise, this location was never used by an enqueue operation, and in this case $D_i$ finds an empty queue.

**Algorithm 1.** A 3-bounded wait-free linearizable implementation of a $(1,n)$-Queue$\langle V\rangle$ object from Registers, Fetch&Add and Swap objects. Process $E$ performs enqueue operations and processes $D_1, \ldots D_n$ perform dequeue operations.

Shared objects:

$ROW$ is an integer Register, initialized to 0.

$HEAD$ $[0 \ldots \infty]$ is an array of Fetch&Increment objects, each initialized to 0.

$ITEMS$ $[0 \ldots \infty, 0 \ldots \infty]$ is a two-dimensional array of Swap objects, each initialized to $\perp$. The set of values allowed in any of these Swap objects is $V \cup \{\perp, \top\}$, where the latter two symbols are not in $V$.

Process $E$:

- Persistent local variables:

  $tail$, $enq\_row$ are integers initialized to 0.

- Access procedure $E\!:\!Enq(x)$, for all $x \in V$:

```
1. (step 1) val ⟵ Swap( ITEMS[ enq_row, tail ], x )
2.          if val = $\top$
            then
3.               increment( enq_row )
4.               tail ⟵ 0
5. (step 2)      Swap( ITEMS[ enq_row, tail ], x )
6. (step 3)      Write( ROW, enq_row )
            end if
7.          increment( tail )
8.          return OK
```

Process $D_i$, for $1 \leq i \leq n$:

- Access procedure $D_i\!:\!Deq$:

```
1. (step 1) deq_row ⟵ Read( ROW )
2. (step 2) head ⟵ Fetch&Increment( HEAD[ deq_row ] )
3. (step 3) val ⟵ Swap( ITEMS[ deq_row, head ], $\top$ )
4.          if val = $\perp$
            then
5.               return $\varepsilon$
            else
6.               return val
            end if
```

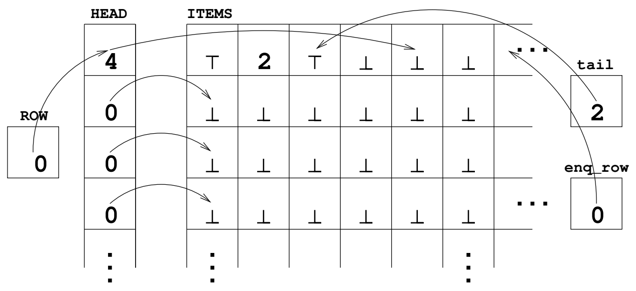

Figure 2.1: A possible state of the shared variables in this implementation.

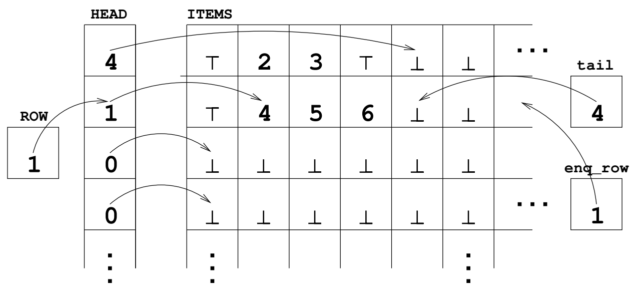

Figure 2.2: Another possible state, extending the previous one.

The process $E$ performing enqueue operations has two local *persistent* variables, *enq_row* and *tail*. They are persistent in the sense that their values are not lost from one enqueue procedure to the next. The value of the variable *enq_row* mirrors the value of the shared register $ROW$, while *tail* contains the smallest index of a Swap object not already used by an enqueue procedure on row *row_enq* of the array $ITEMS$.

In order to perform an enqueue operation, process $E$ writes the value to be enqueued in the array location $ITEMS[enq\_row, tail]$ and retrieves the latter's value. If this value was $\top$, then some dequeue operation has already accessed this cell before $E$ had a chance to write to it. In this case the enqueue procedure will abandon the current row and start using the next row for storing the values in the queue.

The access procedures above consist of local computation and accesses to shared objects, that is, steps. A complete execution of the enqueue procedure can consist of at most three steps, in lines 1, 5 and 6. A complete execution of the dequeue procedure always consists of three steps, in its first three lines.

For example, figure 2.1 presents a possible state of the shared variables in this implementation. In it, exactly 2 enqueue procedures with arguments 1 and 2 were started, both were completed, and neither of them executed the contents of the `if` statement in lines 3 through 6. Exactly 4 dequeue procedures were started and executed at least their first two steps. All 4 of them obtained the result 0 in their first step, and they obtained the results $0, 1, 2, 3$ in their second steps, respectively. The dequeue procedures with $(deq\_row = 0, head = 0)$ and $(deq\_row = 0, head = 2)$ were completed and output the values 1 and $\lambda$, respectively. The dequeue procedures with $(deq\_row = 0, head = 1)$ and $(deq\_row = 0, head = 3)$ only executed their first two steps, and if either was allowed to take another step, they would output 2 and $\lambda$, respectively. Notice that if a new enqueue procedure with argument 3 were started at this point, it would apply a Swap operation with argument 3 to the cell $ITEMS[0, 2]$, and it would obtain the result $\top$ to that step. In this situation, a dequeue procedure accessed that cell before the enqueue procedure, so the latter would then execute the contents of the `if` statement in lines 3 through 6.

The state in figure 2.2 is an extension of the one in figure 2.1. In here, 4 more enqueue procedures with arguments $3,4,5,6$ were started, and all of them were completed. The only enqueue procedure to execute the contents of the `if` statement in lines 3 through 6 was the one with argument 3. One more dequeue procedure was started and executed its first two steps, obtaining $(deq\_row = 1, head = 0)$. This dequeue procedure was completed, and it output 3. Furthermore, one of the two incomplete dequeue procedures from the state in figure 2.1 was completed, the one with $(deq\_row = 0, head = 3)$, and it output $\lambda$.

## 2.2 Proof of Correctness

We now prove that the implementation in Algorithm 1 is 3-bounded wait-free and linearizable. Since in any run, any enqueue or dequeue instance has at most three steps, the implementation is clearly 3-bounded wait-free. In the reminder of this section, we prove the remaining part of our claim:

**Theorem 2.2.** *Algorithm 1 is linearizable.*

To this end, we begin by fixing an arbitrary run $R$ of the implementation and we show that $R$ is linearizable. We first build the set $Occur(R)$ of procedure instances which will be linearized, so that $Complete(R) \subseteq Occur(R) \subseteq Instances(R)$. Next, we specify a linear order on the set $Occur(R)$. We then construct a Queue history $H$ which satisfies the first three requirements for the linearizability of $R$ in Definition 1.1. Finally, we show that $H$ is legal and we conclude that $R$ is linearizable.

**Notation and Definitions** Before we tackle the proof, we need to introduce some notation.

For $\pi$ an enqueue instance in $R$, let $enq\_row_\pi$ and $tail_\pi$ denote the values of the local variables $enq\_row$ and $tail$, respectively, at the beginning of the execution of $\pi$. Let $val_\pi$ denote the result of the first step of $\pi$ (line 1).

For $\phi$ a dequeue instance in $R$, let $deq\_row_\phi$ denote the result of the first step of $\phi$ (line 1). If $\phi$ has at least two steps, let $head_\phi$ denote the result of its second step (line 2). If $\phi$ is complete, that is, if it contains three steps, let $val_\phi$ denote the result of its third step (line 3).

Let $C_0$ denote the initial configuration of the system, and for $i \geq 1$, let $C_i$ denote the configuration of the system after the steps $s_1, \ldots, s_i$ in the system history $S$ of run $R$ have been performed. For $i \geq 0$, let $ITEMS_i$, $HEAD_i$ and $ROW_i$ denote the values of the respective shared variables in the system configuration $C_i$, that is, immediately after step $s_i$ is performed.

For example, for $\phi$ a dequeue instance containing at least two steps, we have

$$head_\phi = HEAD_{step(\phi,2)}[deq\_row_\phi] - 1 = HEAD_{step(\phi,2)-1}[deq\_row_\phi]$$

Let $t(i)$ be the first column of a cell on row $ROW_i$ of $ITEMS_i$ that hasn't been accessed in steps up to and including $s_i$ during an enqueue procedure instance. If $t(i) \leq HEAD_i[ROW_i]$, let $state(i)$ be the empty queue state, $state(i) \overset{def}{=} \lambda$. If, on the other hand, $HEAD_i[ROW_i] < t(i)$, let $state(i)$ denote the queue state which has the elements in the queue ordered from head to tail in columns $HEAD_i[ROW_i], \ldots, t(i)-1$ on row $ROW_i$ of array $ITEMS_i$. Formally,

$$state(i) \overset{def}{=} (ITEMS_i[ROW_i, HEAD_i[ROW_i]], \ldots, ITEMS_i[ROW_i, t(i)-1])$$

We prove in Lemma 2.8 that none of those cells can contain the values $\perp$ or $\top$. Notice that from the definitions above, $t(0) = 0$ and $state(0) = \lambda$.

**Enqueue Instances** All enqueue instances are of one of these two types:

**regular** We say that an enqueue instance $\pi$ is a *regular* enqueue instance if $val_\pi \neq \top$, so $E$ does not execute the body of the *if* statement during $\pi$. A complete regular enqueue instance consists of only one step.

**jump** We say that $\pi$ is a *jump* enqueue instance if $val_\pi = \top$, referring to the fact that it "jumps" to the next row of the array $ITEMS$. A complete jump enqueue instance consists of three steps.

Since all enqueue instances in $R$ are executed sequentially by the same process $E$, no two enqueue instances are concurrent. Furthermore, only the last enqueue instance in $R$ can be incomplete, because in a run $R$, a process must finish the execution of an access procedure before starting the next one. We give several consequences of these definitions, which will be useful in our proof.

**Lemma 2.3.** *Let $\pi$ and $\pi'$ be two enqueue instances such that $\pi$ is a jump enqueue instance and $\pi <_R \pi'$. Then $enq\_row_{\pi'} > enq\_row_\pi$.*

*Proof.* Since $\pi <_R \pi'$ and enqueue instances are executed sequentially by $E$, $\pi$ is complete. Therefore $E$ executes line 3 in $\pi$, incrementing the local variable *enq_row* from $enq\_row_\pi$ to $enq\_row_\pi + 1$. That variable is never decremented, so $enq\_row_{\pi'} \geq enq\_row_\pi + 1 > enq\_row_\pi$. □

**Lemma 2.4.** *At the beginning of the execution of any enqueue instance $\pi$ in $R$, the value of the shared variable ROW is equal to the value of the local variable enq_row. Formally, $ROW_{step(\pi,1)-1} = enq\_row_\pi$.*

*Proof.* Notice that the shared variable $ROW$ is only modified by line 6 (step 3) of the enqueue procedure, and the local variable *enq_row* is only modified by line 3 of the enqueue procedure. In the initial configuration of the system, both variables are 0.

The proof is by induction on the number of enqueue instances in $R$. If $\pi$ is the first enqueue instance in $R$, then $enq\_row_\pi = HEAD_{step(\pi,1)-1} = 0$ and our claim holds. Otherwise let $\pi'$ be the enqueue instance preceding $\pi$ in $R$. We know $\pi'$ is complete. If $\pi'$ is a regular instance, then neither $ROW$ nor *enq_row* are modified in it, so they are still equal at the beginning of $\pi$. Otherwise $\pi'$ is a complete jump instance, and $E$ executes line 6 (step 3) in $\pi'$, which sets $ROW$ to equal *enq_row*. Then they are still equal at the beginning of $\pi$. □

**Corollary 2.5.** *Each time line 6 (step 3) of some enqueue instance is executed, the value of the shared variable ROW is incremented.*

*Proof.* Let $\pi$ be some enqueue instance which has three steps in $R$. By Lemma 2.4, the values of *enq_row* and $ROW$ are equal at the beginning of $\pi$. During $\pi$, $E$ executes line 3, incrementing *enq_row*, before it gets to line 6 (step 3). Therefore, when it executes the latter, it increments the value of $ROW$. □

**Lemma 2.6.** *The cells on a given row of ITEMS are accessed in increasing order of columns by enqueue instances in R, starting at column 0. Formally, if $ITEMS[r,c]$ is accessed in step $s_{i'}$, where $i' \in \pi'$ for some enqueue instance $\pi'$ and $c > 0$, then $\pi'$ is not the first enqueue instance in R and there exists $i < i'$ such that $ITEMS[r,c-1]$ is accessed in step $s_i$ and $i \in \pi$, where $\pi$ is the enqueue instance preceding $\pi'$ in R.*

*Proof.* Let $\pi'$ be an enqueue instance accessing the cell $ITEMS[r,c]$ in step $s_{i'}$ of $S$, with $c > 0$. Since the access in line 5 (step 2) always involves the cell at column 0 of some row, we conclude that $s_{i'}$ is the first step of $\pi'$ and that $c = tail_{\pi'}$. Furthermore, $\pi'$ cannot be the first enqueue instance in $R$, because in that case $tail_{\pi'} = 0$, so $c = 0$. Since $\pi$ is the enqueue instance preceding

$\pi'$ in $R$, $\pi$ is complete. Therefore, $E$ executes line 7 during $\pi$, incrementing *tail* from $c-1$ to $c$, before line 1 (step 1) in $\pi'$. Then $\pi$ accesses the cell $ITEMS[r, c-1]$, either in line 1 (step 1) or in line 5 (step 2). ☐

**Lemma 2.7.** *The cells in $ITEMS$ are accessed by enqueue instances in increasing lexicographical order. Formally, if $\pi$ and $\pi'$ are two, not necessarily distinct, enqueue instances in $R$, $i \in \pi$, $i' \in \pi'$, $i < i'$, $s_i$ contains an access to $ITEMS[r, c]$ and $s_{i'}$ contains an access to $ITEMS[r', c']$, then $(r, c)$ is lexicographically smaller than $(r', c')$, that is, either $r < r'$ or $r = r'$ and $c < c'$.*

*Proof.* Let $\pi$ be an enqueue instance in $R$. Let $i \in \pi$ such that $s_i$ contains an access to $ITEMS[r, c]$. Let $i'$ be the index of the *first step following* $s_i$ in $S$ in which some enqueue instance $\pi'$ accesses some cell $ITEMS[r', c']$. Then either $\pi = \pi'$, or $\pi <_R \pi'$. In the former case, $(r', c') = (r+1, 0)$. In the latter, $\pi <_R \pi'$ so $s_i$ is the last step of $\pi$. Then $s_i$ is either step 1 of a regular enqueue instance, or step 3 of a jump enqueue instance. In any case, between steps $s_i$ and $s_{i'}$, neither $ROW$ nor *enq_row* are modified. Furthermore, line 7 is executed in $\pi$ before the first step of $\pi'$. Hence, $(r', c') = (r, c+1)$. The claim in the Lemma follows by transitivity. ☐

We are now able to prove the following Lemma, which is required by the definition of $state(i)$:

**Lemma 2.8.** *For any $i \geq 0$, if $HEAD_i[ROW_i] < t(i)$, then no cell on row $ROW_i$ of $ITEMS_i$ at a column in $\{HEAD_i[ROW_i], \ldots, t(i)-1\}$ can contain the values $\perp$ or $\top$.*

*Proof.* Fix $i \geq 0$ and assume that $HEAD_i[ROW_i] < t(i)$.

First suppose that $\top$ appears on $ROW_i$ of $ITEMS_i$. The value $\top$ can only appear there as a result of step 3 of some dequeue instance $\phi$, for which $deq\_row_\phi = ROW_i$ and $step(\phi, 3) \leq i$. Then $step(\phi, 2) < i$. Notice that $HEAD[ROW]$ is incremented on each access, hence $head_\phi < HEAD_i[ROW_i]$ and $\phi$ accesses a cell on a column less than $HEAD_i[ROW_i]$. Therefore $\top$ cannot appear in a column greater than or equal to $HEAD_i[ROW_i]$ on row $ROW_i$ of $ITEMS_i$.

Now suppose that $ITEMS_i[ROW_i, k] = \perp$ for some $k$ such that $HEAD_i[ROW_i] \leq k < t(i)$. Since $t(i) > HEAD_i[ROW_i] \geq 0$, $t(i) - 1 \geq 0$. A cell in the array $ITEMS$ can only have the value $\perp$ if it was never accessed by either enqueue or dequeue instances. By definition of $t(i)$, the cell at column $t(i) - 1$ was accessed during an enqueue instance, so $ITEMS_i[ROW_i, t(i)-1] \neq \perp$ and $k < t(i) - 1$. However, by Lemma 2.6, cells on a given row of $ITEMS$ are accessed consecutively in increasing order of columns by enqueue instances, so the cell at column $k$ of row $ROW_i$ must have been accessed before the one at column $t(i) - 1 > k$ on the same row. Hence $ITEMS_i[ROW_i, k]$ cannot contain $\perp$. ☐

**Association between Enqueue Instances and Dequeue Instances** For a dequeue instance $\phi$ with at least two steps, we say that $\phi$ *reserves* the cell at row $deq\_row_\phi$ and column $head_\phi$ of $ITEMS$. We establish a relation between dequeue instances and enqueue instances as follows. Let $\phi$ be a dequeue instance with at least two steps. If there exists an enqueue instance $\pi$ such that:

- $\pi$ accesses the cell in $ITEMS$ reserved by $\phi$, and
- if $\phi$ has three steps, then $\pi$ accesses that cell before the third step of $\phi$,

then we define $\rho(\phi) \overset{def}{=} \pi$. By Lemma 2.7, if $\pi$ exists, then $\pi$ is unique, so this definition is sound. If no such enqueue instance exists, we leave $\rho(\phi)$ undefined. In Lemma 2.11, we will establish the fact that $\rho$ is injective where it is defined, and that it is a correspondence between the subset of all dequeue instances containing at least two steps and a subset of all complete enqueue instances.

In the following Lemma we would like to say that any cell of the array $ITEMS$ is accessed at most once in instances of the dequeue procedure. However, it turns out that in the proof of linearizability we will need a stronger statement, in order to deal with incomplete dequeue instances.

**Lemma 2.9.** *Any cell in the array $ITEMS$ is reserved by at most one dequeue instance in $R$.*

*Proof.* Suppose two dequeue instances $\phi_1$ and $\phi_2$ reserve cells on the same row $r$ of $ITEMS$. Then $deq\_row_{\phi_1} = deq\_row_{\phi_2} = r$, so both $\phi_1$ and $\phi_2$ contain, in their respective steps 2, Fetch&Increment operations on the same shared object, $HEAD[r]$. However, $HEAD[r]$ is incremented each time it is accessed, so $head_{\phi_1} \neq head_{\phi_2}$. Then $\phi_1$ and $\phi_2$ cannot reserve the same cell of $ITEMS$. □

**Lemma 2.10.** *Let $\phi$ be a dequeue instance such that $\rho(\phi)$ is defined, and suppose that $\pi = \rho(\phi)$ is a jump enqueue instance. Then the second step of $\pi$ (and not its first) contains the access to the cell that $\phi$ reserves. Furthermore, $\pi$ is complete.*

*Proof.* Suppose that $\phi$ reserves the cell which is accessed in line 1 (step 1) of $\pi$. The initial value of that cell is $\perp$. Since $\pi$ is a jump enqueue instance, in step 1 of $\pi$ the value $\top$ is read from that cell, and that value can only appear there as a result of an access by a dequeue instance. By Lemma 2.9, no dequeue instance other than $\phi$ accesses that cell, so $\phi$ must access the cell before $\pi$ does. But then, the third step of $\phi$ occurs before the first step of $\pi$, which contradicts the second part of the condition that $\pi = \rho(\phi)$. Therefore, if $\pi = \rho(\phi)$, then $\pi$ has at least two steps, and its second step contains the access to the cell that $\phi$ reserves.

We further claim that $\pi$ is complete in $R$, containing three steps. If it were not complete, then $\pi$ would be the last enqueue instance in $R$, and the maximum value of $ROW$ during the entire execution would be $enq\_row_\pi$. In particular, $deq\_row_\phi \leq enq\_row_\pi$. However, the cell that $\pi$ accesses in step 2 is on row $enq\_row_\pi + 1$ hence $\phi$ would not reserve that cell. □

**Corollary 2.11.** *Let $\phi$ and $\phi'$ be two distinct dequeue instances such that $\rho(\phi)$ and $\rho(\phi')$ are defined. Then $\rho(\phi) \neq \rho(\phi')$.*

*Proof.* Assume that $\rho(\phi) = \rho(\phi') = \pi$. By Lemma 2.9, $\phi$ and $\phi'$ reserve different cells. Only a jump enqueue can access two different cells in $ITEMS$, so $\pi$ is a jump enqueue instance. However, by Lemma 2.10, both $\phi$ and $\phi'$ reserve the cell that $\pi$ accesses in its second step. This is a contradiction. □

The following Lemma shows that the result returned by a dequeue instance $\phi$ is related to the enqueue instance $\rho(\phi)$.

**Lemma 2.12.** *Let $\phi$ be a complete dequeue instance in $R$. If $result(\phi) = \varepsilon$, then $\rho(\phi)$ is not defined. If $result(\phi) \neq \varepsilon$, then $\rho(\phi) = \pi$ is defined and $\pi$ is enqueuing $result(\phi)$.*

*Proof.* Let $\phi$ be a complete dequeue instance with $result(\phi) = \varepsilon$. Since $\varepsilon$ is not a value that can appear in a cell of the array $ITEMS$, it must be that $\phi$ executes line 5, so $\phi$ retrieves $val_\phi = \perp$ in its step 3 from the cell it reserves. Then no procedure instance, in particular no enqueue instance, contains an access to that cell before $step(\phi, 3)$. In order for $\rho(\phi)$ to be defined, some enqueue instance $\pi$ would have to access the cell reserved by $\phi$ before $step(\phi, 3)$, hence $\rho(\phi)$ is not defined.

Now let $\phi$ be a complete dequeue instance with $result(\phi) \neq \varepsilon$. Then in its step 3, $\phi$ retrieves a value other than $\perp$ from the cell it reserves. By Lemma 2.9, that value is the result of an access to that cell by some enqueue instance $\pi$. By Lemma 2.7, $\pi$ is unique. By definition, $\pi = \rho(\phi)$. Notice that whenever $\pi$ accesses a cell from $ITEMS$, the value that $\pi$ is enqueuing is written to that cell. Hence, the value retrieved and output by $\phi$ is the value enqueued by $\pi$. □

**Dequeue Instances** We define three types of dequeue instances:

**type I** A dequeue instance $\phi$ consisting of at least two steps is a *type I* dequeue instance if $\rho(\phi) = \pi$ is defined, and the step in which $\pi$ accesses the cell reserved by $\phi$ occurs *after* step 2 of $\phi$. By definition of $\rho$, the step in which $\pi$ accesses that cell has to precede the third step of $\phi$, should the latter exist in $R$. We prove in Lemma 2.13 that $\pi$ is a regular enqueue instance.

Informally, a complete type I dequeue instance $\phi$ will return a value other than $\varepsilon$, but when $\phi$ reserves a cell (in step 2), the value is not yet in the cell.

**type II** A dequeue instance $\phi$ consisting of at least two steps is a *type II* dequeue instance if there exists a complete jump enqueue instance $\pi'$ such that $\mathit{enq\_row}_{\pi'} = \mathit{deq\_row}_{\phi}$ and $\mathit{step}(\pi', 3) < \mathit{step}(\phi, 2)$. We prove in Lemma 2.14 that $\pi'$ is unique, and in Lemma 2.15 that $\rho(\phi)$ is not defined, hence $\phi$ cannot also be a type I dequeue instance. Informally, between step 1 and step 2 of a type II dequeue instance, a jump enqueue instance has incremented $ROW$. If complete, $\phi$ will return $\varepsilon$.

**type III** A dequeue instance $\phi$ consisting of at least two steps is a *type III* dequeue instance if it is neither type I nor type II. Informally, we cannot say anything about the return value of a type III dequeue instance: it may return $\varepsilon$ or not.

**Lemma 2.13.** *Let $\phi$ be a type I dequeue instance and let $\pi = \rho(\phi)$. Then $\pi$ is a regular enqueue instance.*

*Proof.* Suppose $\pi$ were a jump enqueue instance. By Lemma 2.10, $\pi$ is complete and the access of $\pi$ to the cell reserved by $\phi$ occurs in step 2 of $\pi$, so this cell is on row $\mathit{enq\_row}_{\pi} + 1$ of $ITEMS$. Furthermore, by definition of type I dequeue instance, $\mathit{step}(\phi, 2) < \mathit{step}(\pi, 2)$.

By Corollary 2.5, the value of $ROW$ prior to step 3 of $\pi$ is strictly less than $\mathit{enq\_row}_{\pi} + 1$. $\phi$ reserves a cell on the row $\mathit{deq\_row}_{\phi}$ which is the value of $ROW$ at step 1 of $\phi$. So, in order for $\phi$ to reserve a cell on row $\mathit{enq\_row}_{\pi} + 1$, it must be the case that $\mathit{step}(\pi, 3) < \mathit{step}(\phi, 1)$. But then $\mathit{step}(\pi, 2) < \mathit{step}(\phi, 2)$. This is a contradiction. □

**Lemma 2.14.** *The complete jump enqueue instance mentioned in the definition of a type II dequeue instance is unique.*

*Proof.* Let $\phi$ be a type II dequeue instance in $R$, and let $\pi'$ and $\pi''$ be two complete jump enqueue instances satisfying the requirements in the definition of a type II dequeue instance. In particular, $\mathit{enq\_row}_{\pi'} = \mathit{enq\_row}_{\pi''} = \mathit{deq\_row}_{\phi}$.

If $\pi' \neq \pi''$, then, without loss of generality, suppose that $\pi' <_R \pi''$. But then, by Lemma 2.3, $\mathit{enq\_row}_{\pi''} \geq \mathit{enq\_row}_{\pi'} + 1$. This is a contradiction. □

**Lemma 2.15.** *Let $\phi$ be a type II dequeue instance. Then $\rho(\phi)$ is not defined.*

*Proof.* Let $\pi'$ be a complete jump enqueue instance with $\mathit{enq\_row}_{\pi'} = \mathit{deq\_row}_{\phi}$ and $\mathit{step}(\pi', 3) <_R \mathit{step}(\phi, 2)$. Suppose $\pi = \rho(\phi)$ for some enqueue instance $\pi$. Enqueue instances are completely ordered in $R$, so either $\pi = \pi'$ or $\pi' <_R \pi$ or $\pi <_R \pi'$.

If $\pi = \pi'$, then, by Lemma 2.10, $\mathit{deq\_row}_{\phi} = \mathit{enq\_row}_{\pi'} + 1$. This is a contradiction.

If $\pi' <_R \pi$, by Lemma 2.3, $\mathit{enq\_row}_{\pi} \geq \mathit{enq\_row}_{\pi'} + 1$. Since $\pi = \rho(\phi)$, $\mathit{deq\_row}_{\phi} \geq \mathit{enq\_row}_{\pi} \geq \mathit{enq\_row}_{\pi'} + 1$. This is also a contradiction

Finally, it might be the case that $\pi <_R \pi'$. In its step 1, $\pi'$ reads the value $\top$ from the cell at $(\mathit{enq\_row}_{\pi'}, \mathit{tail}_{\pi'})$. This value cannot occur there unless some dequeue instance $\phi'$ accesses that cell before $\pi'$. By Lemma 2.9, there is a unique dequeue instance $\phi'$ such that $(\mathit{enq\_row}_{\pi'}, \mathit{tail}_{\pi'}) = (\mathit{deq\_row}_{\phi'}, \mathit{head}_{\phi'})$. Since dequeue instance only access the array $ITEMS$ in their third steps, $\mathit{step}(\phi', 3) < \mathit{step}(\pi', 1)$. We have

$$\mathit{step}(\phi', 2) < \mathit{step}(\phi', 3) < \mathit{step}(\pi', 1) < \mathit{step}(\pi', 3) < \mathit{step}(\phi, 2).$$

By definition of $\pi'$, $\mathit{deq\_row}_{\phi} = \mathit{enq\_row}_{\pi'}$. By definition of $\phi'$, $\mathit{deq\_row}_{\phi'} = \mathit{enq\_row}_{\pi'}$. Hence both $\phi$ and $\phi'$ perform a Fetch&Increment operation on $HEAD[\mathit{enq\_row}_{\pi'}]$ in their second steps, and that shared variable is incremented at each access, so $\mathit{head}_{\phi'} < \mathit{head}_{\phi}$.

Suppose $\pi$ is a regular enqueue instance. Since $\pi = \rho(\phi)$, $enq\_row_\pi = deq\_row_\phi = enq\_row_{\pi'}$, and furthermore, $tail_\pi = head_\phi$. By Lemma 2.7, $tail_\pi < tail_{\pi'}$. By definition of $\phi'$, $head_{\phi'} = tail_{\pi'}$. Hence $head_\phi < head_{\phi'}$. This is a contradiction.

Now suppose $\pi$ is a jump enqueue instance. By Lemma 2.10, $head_\phi = 0$. This is impossible because $head_{\phi'} < head_\phi$ and $head_{\phi'}$ is a non-negative integer.

This completes the case analysis and shows that $\rho(\phi)$ is not defined. □

**Lemma 2.16.** *Let $\phi$ be a dequeue instance containing at least two steps, such that $ROW_{step(\phi,2)} \neq deq\_row_\phi$. Then $\phi$ is a type II dequeue instance.*

*Proof.* Note that $ROW_{step(\phi,2)} = ROW_{step(\phi,2)-1}$ as the second step of $\phi$ does not modify $ROW$. By the algorithm, $deq\_row_\phi = ROW_{step(\phi,1)}$. Let $i$ be the index of the first step following $step(\phi,1)$ such that $ROW_i \neq deq\_row_\phi$. So $step(\phi,1) < i < step(\phi,2)$.

The only line modifying the value of $ROW$ is line 6 (step 3) of an enqueue instance. By Corollary 2.5, $ROW_i = deq\_row_\phi + 1$ and there exists an enqueue instance $\pi'$ such that $enq\_row_{\pi'} = deq\_row_\phi$ and $i = step(\pi',3)$. But then, by definition, $\phi$ is a type II dequeue instance. □

**Construction of** $Occur(R)$ We can now construct $Occur(R)$ so that $Complete(R) \subseteq Occur(R) \subseteq Instances(R)$. Let $Occur(R)$ be the set of all complete enqueue instances in $R$, together with all dequeue instances containing at least two steps in $R$. We will define a linear order $\prec$ on the procedure instances in $Occur(R)$. To do that, we first define an *occurrence point* for each of these instances.

**Occurrence Points** For $\alpha \in Occur(R)$, the occurrence point $occ(\alpha)$ is defined to be the index of some step in $S$, at or after the first step of $\alpha$, and also at or before the last step of $\alpha$, in case $\alpha$ is complete. Formally, $step(\alpha,1) = \min(\alpha) \leq occ(\alpha)$, and if $\alpha$ is complete, $occ(\alpha) \leq \max(\alpha)$. The *occurrence step* of procedure instance $\alpha$ is $s_{occ(\alpha)}$. Notice that the occurrence step of a procedure instance need not be part of that instance. Intuitively, a procedure instance seems to be performed atomically at its occurrence step.

Occurrence points are defined as follows:

- For a complete regular enqueue instance $\pi \in Occur(R)$, let $occ(\pi) = step(\pi,1)$. Since $\max(\pi) = step(\pi,1) = \min(\pi)$, we have $\min(\pi) \leq occ(\pi) \leq \max(\pi)$.

- For a complete jump enqueue instance $\pi \in Occur(R)$, let $occ(\pi) = step(\pi,3)$. Since $\max(\pi) = step(\pi,3)$, we have $\min(\pi) \leq occ(\pi) \leq \max(\pi)$.

- For a type I dequeue instance $\phi \in Occur(R)$ with $\pi = \rho(\phi)$, let $occ(\phi) = step(\pi,1)$. By Lemma 2.13, $\pi$ is a regular enqueue instance. By definition of type I, $step(\phi,2) < step(\pi,1)$. Hence $min(\phi) \leq occ(\phi)$. If $\phi$ is complete, then by the second condition in the definition of $\rho$, $step(\pi,1) < step(\phi,3) = max(\phi)$. Hence $occ(\phi) \leq max(\phi)$.

- For a type II dequeue instance $\phi \in Occur(R)$, let $\pi'$ be a complete jump enqueue instance such that $enq\_row_{\pi'} = deq\_row_\phi$ and $step(\pi',3) < step(\phi,2)$. By Lemma 2.14, there is a unique such $\pi'$. In this case, let $occ(\phi) = step(\pi',3)$. By Lemma 2.5, the value of $ROW$ after step 3 of $\pi'$ is at least as large as $enq\_row_{\pi'}+1$. So in order for $\phi$ to read $enq\_row_{\pi'}$ from $ROW$ in its first step, we must have $step(\phi,1) < step(\pi',3)$. Hence $step(\phi,1) = min(\phi) \leq occ(\phi)$. Furthermore,
$$occ(\phi) \leq step(\phi,2) \leq max(\phi).$$

- For a type III dequeue instance $\phi \in Occur(R)$, let $occ(\phi) = step(\phi,2)$. Clearly $min(\phi) \leq occ(\phi) \leq max(\phi)$.

**Linear Order on** $\mathit{Occur}(R)$ Occurrence points are positive integers, and they give us a partial order on the set of procedure instances in $\mathit{Occur}(R)$. We will complete this order to get a linear order $\prec$ on $\mathit{Occur}(R)$. To do this, we linearly order the procedure instances with the same occurrence step. There are only three types of occurrence steps:

- Step 1 of a regular enqueue instance $\pi$. Apart from $\pi$ itself, only type I dequeue instances can have this as their occurrence step. However, a dequeue instance $\phi$ has step 1 of $\pi$ as its occurrence step only if $\phi$ is type I and $\pi = \rho(\phi)$. By Corollary 2.11, at most one such $\phi$ exists, so at most one dequeue instance can have step 1 of $\pi$ as its occurrence step. If there is indeed one such type I dequeue instance $\phi$, we put $\pi$ *before* $\phi$ in the total order on $\mathit{Occur}(R)$.

- Step 3 of a jump enqueue instance $\pi$. Apart from $\pi$ itself, only type II dequeue instances can have this as their occurrence step. Let $\mathcal{D}_\pi$ denote the set of those type II dequeue instances having step 3 of $\pi$ as their occurrence step. In contrast to the case above, the cardinality of $\mathcal{D}_\pi$ can be greater than 1. In the total order on $\mathit{Occur}(R)$, we put $\pi$ *after* all type II dequeue instances in $\mathcal{D}_\pi$, and we order the dequeue instances in $\mathcal{D}_\pi$ arbitrarily.

- Step 2 of a type III dequeue instance $\phi$. Only $\phi$ can have this step as its occurrence step.

We have now obtained a linear order $\prec$ on $\mathit{Occur}(R)$. Let us denote with $\alpha_i$ the $i$-th instance in $\mathit{Occur}(R)$ with respect to $\prec$, so that $\alpha_1 \prec \alpha_2 \prec \ldots$. Notice that, when $\alpha_i <_R \alpha_j$ (that is, $\max(\alpha_i) < \min(\alpha_j)$ and $\alpha_i$ is complete), we have

$$\mathit{occ}(\alpha_i) \leq \max(\alpha_i) < \min(\alpha_j) \leq \mathit{occ}(\alpha_j).$$

Hence $\mathit{occ}(\alpha_i) < \mathit{occ}(\alpha_j)$, so $i < j$.

**Object History** $H$ We now build a Queue history $H = ((op_1, res_1), (op_2, res_2), \ldots)$ as follows: for all $i$, we let $op_i \overset{def}{=} \mathit{operation}(\alpha_i)$. If $\alpha_i$ is complete, $res_i \overset{def}{=} \mathit{result}(\alpha_i)$. This covers case when $\alpha_i$ is an enqueue instance or a complete dequeue instance. Now let $\phi = \alpha_i$ be an incomplete dequeue instance. Then $\phi$ contains exactly two steps out of three. If $\rho(\phi)$ is defined and $\pi = \rho(\phi)$ is enqueuing $x$, let $res_i = x$. If $\rho(\phi)$ is not defined, let $res_i = \varepsilon$.

Notice that, with this definitions, the first three conditions for the linearizability of the run $R$ are satisfied, and all that it remains to prove is that $H$ is legal.

**State Sequence** In order to prove that $H$ is legal, we construct a state sequence $Q = (q_0, q_1, \ldots)$ and we prove that for all $i \geq 1$, $\delta_{QUEUE}(q_{i-1}, op_i) = (q_i, res_i)$. Let $q_0 \overset{def}{=} \lambda$ denote the state where the queue is empty. For $i \geq 1$, we define $q_i$ as follows:

- If among all procedure instances with the same occurrence step $s_j$ as $\alpha_i$, $\alpha_i$ is the last in the total order on $\mathit{Occur}(R)$, let $q_i \overset{def}{=} \mathit{state}(j)$. This covers the cases where $\alpha_i$ is a regular enqueue instance and there is no type I dequeue instance $\phi$ with $\rho(\phi) = \alpha_i$, a jump enqueue instance, a type III dequeue instance, or a type I dequeue instance.

- If $\alpha_i$ is a regular enqueue instance and there exists a type I dequeue instance $\phi$ such that $\rho(\phi) = \alpha_i$, we define $q_i$ to be the state in which the queue consists of only one element, the one being enqueued by $\alpha_i$.

- If $\alpha_i$ is a type II dequeue instance, we define $q_i \overset{def}{=} \lambda$.

Before we tackle the proof that the sequence of states $Q$ we have constructed is consistent with the operations in the history $H$, we need several Lemmas.

**Lemma 2.17.** *Let $i \geq 1$ such that no procedure instance in $Occur(R)$ has $i$ as its occurrence point. Then $state(i) = state(i-1)$.*

*Proof.* The step $s_i$ is not the occurrence step of any procedure instance if and only if $s_i$ is one of: step 1 of some dequeue instance, step 2 of some type I or type II dequeue instance, step 3 of some dequeue instance, and step 1 or step 2 of some jump enqueue instance. Below, we analyze all these cases:

- If $s_i$ is step 1 of some dequeue instance (executing a Read operation on $ROW$), no shared variable is modified by $s_i$. Hence $state(i) = state(i-1)$.

- If $s_i$ is step 2 of some type I dequeue instance $\phi$ (executing a Fetch&Increment operation on $HEAD[deq\_row]$), $ROW_i = ROW_{i-1}$ and $ITEMS_i = ITEMS_{i-1}$. Since $t(i)$ is defined only as function of $ROW_i$, $ITEMS_i$ and steps of enqueue instances in $R$, $t(i) = t(i-1)$.

  By Lemma 2.16, $deq\_row_\phi = ROW_i$. Hence $\phi$ performs a Fetch&Increment operation on $HEAD[ROW_i]$ in step $s_i$, so
  $$head_\phi = HEAD_i[ROW_i] - 1 = HEAD_{i-1}[ROW_{i-1}]$$
  Let $\pi = \rho(\phi)$. By Lemma 2.13, $\pi$ is a regular enqueue instance, so $tail_\pi = head_\phi$. By definition of type I dequeue instances, $i < step(\pi, 1)$. By Lemma 2.6, no cell in a column greater than or equal to $tail_\pi$ on row $ROW_i$ could have been accessed by an enqueue instance before step $step(\pi, 1)$, hence before step $i$. Then
  $$t(i) \leq tail_\pi = head_\phi < HEAD_i[ROW_i],$$
  so $state(i) = \lambda$. Furthermore,
  $$t(i-1) = t(i) \leq head_\phi = HEAD_{i-1}[ROW_{i-1}],$$
  so $state(i-1) = state(i) = \lambda$.

- If $s_i$ is step 2 of some type II dequeue instance $\phi$, let $\pi'$ be the complete jump enqueue instance mentioned in the definition of a type II dequeue instance and in Lemma 2.14. Then $occ(\phi)$ is defined to be $step(\pi', 3)$, and we have seen that $min(\phi) = step(\phi, 1) \leq occ(\phi) = step(\pi', 3)$. By definition of type II dequeue instances, we have $step(\pi', 3) < step(\phi, 2)$. So we have
  $$step(\phi, 1) < step(\pi', 3) < step(\phi, 2) = i.$$
  By Corollary 2.5, the value of $ROW$ is incremented in $step(\pi', 3)$ and never decreases, so the value that $\phi$ reads in its step 1 must be strictly less than $ROW_i$: $deq\_row_\phi < ROW_i$. Therefore $\phi$ does not modify $HEAD[ROW_i]$ in $s_i$. Then $ROW_i = ROW_{i-1}$, $ITEMS_i = ITEMS_{i-1}$, $t(i) = t(i-1)$ and $HEAD_i[ROW_i] = HEAD_{i-1}[ROW_{i-1}]$, hence $state(i) = state(i-1)$.

- If $s_i$ is step 3 of some dequeue instance $\phi$ (executing a Swap operation on the variable on row $deq\_row_\phi$ and column $head_\phi$ of $ITEMS$, with argument $\top$), then $ROW_i = ROW_{i-1}$ and $HEAD_i = HEAD_{i-1}$. Since $t(i)$ is only defined in terms of the cells in $ITEMS$ accessed during enqueue instances, we get $t(i) = t(i-1)$, therefore $state(i) = state(i-1)$.

- If $s_i$ is step 1 of some jump enqueue instance $\pi$, $enq\_row_\pi = ROW_i = ROW_{i-1}$ (by Lemma 2.4), $HEAD_i = HEAD_{i-1}$, $t(i-1) = tail_\pi$ (by Lemma 2.6) and $t(i) = tail_\pi + 1$. In $s_i$, $\pi$ retrieves the value $\top$ from $ITEMS[ROW_i, tail_\pi]$ which means a dequeue instance reserved and accessed that cell before $s_i$. The shared variable $HEAD[ROW_i]$ is incremented at each access, so
  $$HEAD_i[ROW_i] = HEAD_{i-1}[ROW_{i-1}] \geq tail_\pi + 1 = t(i) > tail_\pi = t(i-1),$$
  hence $state(i) = state(i-1) = \lambda$.

- If $s_i$ is step 2 of some jump enqueue instance $\pi$ (executing a Swap operation on the variable at row $enq\_row_\pi+1$ and column 0 of $ITEMS$, with argument $x$), $enq\_row_\pi = ROW_i = ROW_{i-1}$ and $HEAD_i = HEAD_{i-1}$. Note that $\pi$ writes to row $ROW_i+1$ of $ITEMS$ in $s_i$, not to row $ROW_i$. So $t(i) = t(i-1)$ and therefore $state(i) = state(i-1)$.

This completes the case analysis, showing that whenever no instance has $i$ as its occurrence point, $state(i) = state(i-1)$. □

**Corollary 2.18.** *Let $i \geq 1$ be such that $\alpha_i$ is the first instance in the total order on $Occur(R)$ to have $j$ as its occurrence point. Then $q_{i-1} = state(j-1)$.*

*Proof.* When $i > 1$, $\alpha_{i-1}$ is the last instance in the total order on $Occur(R)$ to have $k < j$ as its occurrence point, so by definition, $q_{i-1} = state(k)$. When $i = 1$, let $k = 0$ to get $q_{i-1} = state(k) = \lambda$. In either case, no instance has its occurrence point between $k+1$ and $j-1$, inclusively, so an inductive argument using the Lemma 2.17 yields the desired result. □

**Lemma 2.19.** *Let $i \geq 1$ be such that $s_i$ is step 1 of a regular enqueue instance $\pi$ and suppose that there exists a type I dequeue instance $\phi$ such that $\rho(\phi) = \pi$. Then $state(i) = state(i-1) = \lambda$.*

*Proof.* By the algorithm, $ROW_i = ROW_{i-1}$ and $HEAD_i = HEAD_{i-1}$. Since $\pi = \rho(\phi)$, $enq\_row_\pi = deq\_row_\phi$ and $head_\phi = tail_\pi$. By Lemma 2.4, $ROW_{i-1} = enq\_row_\pi$. So $s_i$ contains an access by $\pi$ to the cell in column $tail_\pi$ of row $ROW_i$, and by Lemma 2.6, we have $t(i-1) = tail_\pi$ and $t(i) = tail_\pi + 1$.

By definition of type I dequeue instances, $\phi$ performs its second step reserving the cell at column $head_\phi$ before $s_i$. Since $HEAD[ROW_i]$ is incremented at each access,

$$HEAD_i[ROW_i] = HEAD_{i-1}[ROW_{i-1}] \geq head_\phi + 1 = tail_\pi + 1 = t(i) > tail_\pi = t(i-1),$$

hence $state(i) = state(i-1) = \lambda$. □

**Lemma 2.20.** *Let $i \geq 1$ be such that $s_i$ is step 3 of some jump enqueue instance $\pi$. Then $state(i-1) = \lambda$ and $state(i)$ is the state in which the queue contains only the element enqueued by $\pi$.*

*Proof.* The third step of a jump enqueue instance increments the shared variable $ROW$ from $ROW_{i-1}$ to $ROW_i$. No dequeue instance read the value $ROW_i$ in its first step prior to this increment, so $HEAD_i[ROW_i] = 0$, the initial value of that Fetch&Increment shared object. The only access by an enqueue instance to a cell on row $ROW_i$ of $ITEMS$ is the one which occurs in $s_i$. Hence, $t(i) = 1$ and $ITEMS_i[ROW_i, 0]$ contains the value enqueued by $\pi$. Therefore, in $state(i)$, the queue contains only the element enqueued by $\pi$.

Now let us prove that $state(i-1) = \lambda$. By Corollary 2.5, $enq\_row_\pi = ROW_{i-1} = ROW_i - 1$. In its step 1, $\pi$ accesses the cell $ITEMS[ROW_{i-1}, tail_\pi]$ so by Lemma 2.6, $t(i-1) = tail_\pi + 1$. However, $\pi$ retrieves the value $\top$ in its step 1, so that cell must have been reserved and accessed by a dequeue instance before step 1 of $\pi$. The variable $HEAD[ROW_{i-1}]$ is incremented at each access, therefore $HEAD_{i-1}[ROW_{i-1}] \geq tail_\pi + 1 = t(i-1)$. Hence $state(i-1) = \lambda$. □

We are finally ready to prove:

**Theorem 2.21.** *The queue history $H$ is legal.*

*Proof.* We show that for all $i \geq 1$, $\delta_{QUEUE}(q_{i-1}, op_i) = (q_i, res_i)$. To this end, fix $i$ and consider all possibilities for $\alpha_i$:

- $\pi = \alpha_i$ is a regular enqueue instance and there is no type I dequeue instance $\phi$ such that $\rho(\phi) = \pi$. We have $op_i = Enq(x)$ for some $x$ and $res_i = OK$. Let $j = step(\pi, 1)$ be the occurrence point of $\pi$. We know that

  $$\delta_{QUEUE}(q_{i-1}, Enq(x)) = (q_{i-1} \cdot x, OK),$$

  so all we have to show is that $q_i = q_{i-1} \cdot x$.

  Since $\pi$ is the only instance with occurrence point $j$, by definition, $q_i = state(j)$, and by Corollary 2.18, $q_{i-1} = state(j-1)$. Clearly $ROW_j = ROW_{j-1}$, $HEAD_j = HEAD_{j-1}$. By Lemma 2.4, $ROW_{j-1} = enq\_row_\pi$. By Lemma 2.6, $t(j-1) = tail_\pi$ and $t(j) = tail_\pi + 1$. Notice that $t(j) \geq 1$. The value $x$ is put in the array $ITEMS$ at $(ROW_j, t(j) - 1)$. Since $HEAD_j[ROW_j] = HEAD_{j-1}[ROW_{j-1}]$ and $t(j) = t(j-1) + 1$, in order to show that $q_i = q_{i-1} \cdot x$, it is enough to prove that $q_i \neq \lambda$, that is, $HEAD_j[ROW_j] < t(j)$.

  Suppose that $1 \leq t(j) \leq HEAD_j[ROW_j]$. Since $HEAD[ROW_j]$ is only changed by step 2 of a dequeue instance which increments it, and its initial value is 0, there exists a dequeue instance $\phi'$ such that $deq\_row_{\phi'} = ROW_j$, $head_{\phi'} = t(j) - 1 \geq 0$ and $step(\phi', 2) < j$. Then $head_{\phi'} = tail_\pi$, hence $\phi'$ reserves the cell that $\pi$ accesses. Given that $\pi$ is a regular instance, the value retrieved from that cell by $\pi$ in its step 1 is not $\top$, therefore, if $\phi'$ is complete, $j < step(\phi', 3)$. But then $\rho(\phi') = \pi$, and furthermore, $\phi'$ is a type I dequeue instance. We assumed no such dequeue instance exists. This contradiction shows that $HEAD_j[ROW_j] < t(j)$.

- $\pi = \alpha_i$ is a regular enqueue instance and there exists a type I dequeue instance $\phi$ such that $\rho(\phi) = \pi$. We have $op_i = Enq(x)$ for some $x$ and $res_i = OK$. Let $j = step(\pi, 1)$ be the occurrence point of $\pi$. The only procedure instances to have their occurrence point at $j$ are $\pi$ and $\phi$. By definition, $\pi \prec \phi$, so by Corollary 2.18, $q_{i-1} = state(j-1)$. By Lemma 2.19, $state(j-1) = \lambda$. By definition, $q_i = (x)$. Our conclusion follows from the fact that

  $$\delta_{QUEUE}(\lambda, Enq(x)) = ((x), OK)$$

- $\pi = \alpha_i$ is a jump enqueue instance. We have $op_i = Enq(x)$ for some $x$ and $res_i = OK$. Let $j = step(\pi, 3)$ be the occurrence point of $\pi$. The only procedure instances to have $j$ as their occurrence point are $\pi$ and possibly some type II dequeue instances collected in the set $\mathcal{D}_\pi$. By definition, $\pi$ comes after any dequeue instances in $\mathcal{D}_\pi$ in $\prec$, so, by definition, $q_i = state(j)$. By Lemma 2.20, $state(j-1) = \lambda$ and in $state(j)$, the queue contains only the element $x$.

  If there are no dequeue instances with occurrence point $j$, then $\pi$ is also the first instance occurring at $j$, and by Corollary 2.18, $q_{i-1} = state(j-1) = \lambda$. Otherwise, there are type II dequeue instances with occurrence point $j$. Then $\alpha_{i-1}$ is one of them, and by definition, $q_{i-1} = \lambda$. Hence, in either case, $q_{i-1} = \lambda$. Our conclusion again follows from the fact that

  $$\delta_{QUEUE}(\lambda, Enq(x)) = ((x), OK)$$

- $\phi = \alpha_i$ is a type I dequeue instance, and $\rho(\phi) = \pi$ is defined. So $op_i = Deq$ and by Lemma 2.13, $\pi$ is a regular enqueue instance. Let $x$ be the element enqueued by $\pi$ and let $j = step(\pi, 1)$ be the occurrence of point of both $\pi$ and $\phi$. Only these two instances have occurrence point $j$, and $\pi \prec \phi$. Then $\alpha_{i-1} = \pi$, and by definition, $q_{i-1} = (x)$. Since $\phi$ is the last instance with occurrence point $j$, by definition, $q_i = state(j)$. By Lemma 2.19, $state(j) = \lambda$. If $\phi$ is complete, by Lemma 2.12, $result(\phi) = x$, and by definition, $res_i = result(\phi)$. If $\phi$ is incomplete, by definition, $res_i = x$. Our conclusion follows from the fact that

  $$\delta_{QUEUE}((x), Deq) = (\lambda, x)$$

- $\phi = \alpha_i$ is a type II dequeue instance. So $op_i = Deq$. Let $\pi'$ be the complete jump enqueue mentioned in the definition of a type II dequeue instance. By Lemma 2.14, $\pi'$ is unique. Let $j = step(\pi', 3)$ be the occurrence point of both $\pi'$ and $\phi$. The procedure instances which have $j$ as their occurrence point are $\pi'$ and some type II dequeue instances (including $\phi$) collected in the set $\mathcal{D}_{\pi'}$. By definition, $q_i = \lambda$.

  If $i > 1$ and $\alpha_{i-1} \in D_{\pi'}$, then $\alpha_{i-1}$ is a type II dequeue instance and, by definition, $q_{i-1} = \lambda$. If $i > 1$ and $\alpha_{i-1} \notin D_{\pi'}$, by Corollary 2.18, $q_{i-1} = state(j-1)$, and by Lemma 2.20, $state(j-1) = \lambda$. If $i = 1$, by definition, $q_{i-1} = q_0 = \lambda$. Hence, in any case, $q_{i-1} = \lambda$.

  By Lemma 2.15, $\rho(\phi)$ is not defined. If $\phi$ is complete, by definition $res_i = result(\phi)$, and by Lemma 2.12, $result(\phi) = \varepsilon$. If $\phi$ is incomplete, by definition $res_i = \varepsilon$. Hence, in any case, $res_i = \varepsilon$. Our conclusion follows from the fact that

  $$\delta_{QUEUE}(\lambda, Deq) = (\lambda, \varepsilon)$$

- $\phi = \alpha_i$ is a type III dequeue instance, $\rho(\phi)$ is defined and $\pi = \rho(\phi)$ is enqueuing $x$. So $op_i = Deq$. Let $j$ be the occurrence point of $\phi$, which is, by definition, $step(\phi, 2)$. No other procedure instance has its occurrence point there, hence, by definition, $q_i = state(j)$, and by Corollary 2.18, $q_{i-1} = state(j-1)$.

  If $\phi$ is complete, by definition, $res_i = result(\phi)$, and by Lemma 2.12, $result(\phi) = x$. If $\phi$ is incomplete, by definition, $res_i = x$. Hence, in any case, $res_i = x$. So in order to show that

  $$\delta_{QUEUE}(q_{i-1}, Deq) = (q_i, x),$$

  it is enough to show that $q_{i-1} = x \cdot q_i$.

  Since $s_j$ contains a Fetch&Increment operation, $ROW_j = ROW_{j-1}$, $ITEMS_j = ITEMS_{j-1}$ and $t(j) = t(j-1)$. By Lemma 2.16, $ROW_j = deq\_row_\phi$, so

  $$head_\phi = HEAD_{j-1}[ROW_{j-1}] = HEAD_j[ROW_j] - 1.$$

  Since $\phi$ is not a type I dequeue instance, $\pi$ has accessed the cell reserved by $\phi$ before step $s_j$, so $ITEMS_{j-1}[ROW_{j-1}, head_\phi] = x$ and by Lemma 2.6, $t(j-1) \geq head_\phi + 1$. Since $head_\phi = HEAD_{j-1}[ROW_{j-1}]$, $state(j-1) \neq \lambda$, and in $state(j-1)$, the element at the head of the queue is

  $$ITEMS_{j-1}[ROW_{j-1}, HEAD_{j-1}[ROW_{j-1}]] = x.$$

  In $state(j)$, the queue contains the same elements as in $state(j-1)$, except that the head of the queue is now $HEAD_j[ROW_j] = HEAD_{j-1}[ROW_{j-1}] + 1$, so $state(j-1) = q_{i-1} = x \cdot state(j) = x \cdot q_i$.

- $\phi = \alpha_i$ is a type III dequeue instance and $\rho(\phi)$ is not defined. So $op_i = Deq$. Let $j$ be the occurrence point of $\phi$, which is, by definition, $step(\phi, 2)$. No other procedure instance has its occurrence point there, hence, by definition, $q_i = state(j)$, and by Corollary 2.18, $q_{i-1} = state(j-1)$.

  If $\phi$ is complete, by definition, $res_i = result(\phi)$, and by Lemma 2.12, $result(\phi) = \varepsilon$. If $\phi$ is incomplete, by definition, $res_i = \varepsilon$. Hence, in any case, $res_i = \varepsilon$. So in order to show that

  $$\delta_{QUEUE}(q_{i-1}, Deq) = (q_i, \varepsilon),$$

  it is enough to show that $q_{i-1} = q_i = \lambda$.

  Since $s_j$ contains a Fetch&Increment operation, $ROW_j = ROW_{j-1}$, $ITEMS_j = ITEMS_{j-1}$ and $t(j) = t(j-1)$. By Lemma 2.16, $ROW_j = deq\_row_\phi$, so

  $$head_\phi = HEAD_{j-1}[ROW_{j-1}] = HEAD_j[ROW_j] - 1.$$

Suppose that $head_{\phi} < t(j-1)$. By Lemma 2.6, there exists an enqueue instance $\pi$ which accesses the cell at $(ROW_{j-1}, head_{\phi})$ before $step(\phi, 2) = j$. But then $\pi = \rho(\phi)$, and this contradicts the assumption that $\rho(\phi)$ is not defined. Therefore $t(j-1) \leq head_{\phi} = HEAD_{j-1}[ROW_{j-1}] < HEAD_j[ROW_j]$, and thus $state(j-1) = q_{i-1} = state(j) = q_i = \lambda$.

This completes the case analysis, and show that for all $i \geq 1$, $\delta_{QUEUE}(q_{i-1}, op_i) = (q_i, res_i)$. Therefore the queue history $H$ is legal. □

# Chapter 3

# Simulating a System with One Object

One of the established ways to attempt proving Conjecture 1.2 is to consider any wait-free implementation of a 3-Queue$\langle \mathbb{Z} \rangle$ object and use an adversarial argument to construct either a "bad" run violating the linearizability assumption, or a family of "bad" runs violating the wait-freedom assumption. Adversarial arguments often consider simpler models, even if somewhat more powerful, in order to simplify the description of that argument. A main difficulty in analyzing the interactions between the various processes in a system of Fetch&Add objects and Registers comes from the multitude of shared objects we have to deal with — in general, an algorithm for this system (in particular a Queue object implementation) can use countably infinitely many shared objects. In this framework, there arises an issue about the flow of information between processes: if in some configuration of the system, $P_1$ and $P_2$ are about to apply two operations on two different shared objects, how long will it take, or under what circumstances, will either of them become aware of the other's action? Ideally, we would like to both simplify and quantify the flow of information in the system. To do this, we introduce in this chapter a new shared object type, BH&AR, which has the following two important characteristics:

- On the one hand, a system with one BH&AR object is simple: we can exactly quantify the information that a process gets each time it applies an operation on $O$.
- On the other hand, a system with one BH&AR object is as powerful as one providing a countably infinite collection of Fetch&Add objects and Registers. Therefore, there exists a wait-free linearizable Queue implementation from Fetch&Add objects and Registers if and only if there exists a similar implementation from only one BH&AR object.

We begin by formally defining the BH&AR and B-History object types in section 3.1, and describing the information each process gets as a result of an operation on one such object. The results in this chapter are summarized by the following theorem:

**Theorem 3.1.** *The following collections of shared objects are equally powerful:*

*(i) a countably infinite collection of Fetch&Add objects and Registers;*

*(ii) one B-History object and $N$ MRSW Registers, one written by each process in the system;*

*(iii) one BH&AR object.*

In section 3.2, we prove that a B-History object can be implemented from one Fetch&Add object, implying that (ii) can be implemented from (i). In section 3.3, we argue that (iii) can be implemented from (ii). In section 3.4, we prove that a countably infinite collection of Fetch&Add objects and MRSW Registers can be implemented from (iii). We know from [Jay95] that a MRMW Register can be implemented from MRSW Registers, therefore by transitivity, (i) can be implemented from (iii).

## 3.1 The B-History and BH&AR Object Types

We begin this section by informally describing, in subsection 1, the B-History and BH&AR object types. In subsection 2, we introduce a notion of indistinguishability over the set of possible states of a BH&AR object. Subsection 3 contains the formal object type definitions.

### 3.1.1 An Informal Description of the B-History and BH&AR Types

Consider a distributed system of $N$ processes, providing Registers and Fetch&Add objects. One of the difficulties in trying to prove Conjecture 1.2 is to quantify what "information" a process $P$ can have about the steps taken by other processes, as $P$ is executing an access procedure of the implemented Queue object. Our goal is to show that even if every process has the "maximum possible information allowed by this system", there still exists a run of this implementation which is not linearizable. The B-History object type is introduced in an attempt to capture the intuition behind the phrase "maximum possible information". It is primarily intended to be of theoretical interest, in proving an impossibility result.

We would like to design an object which functions as a signature log. Each process $P_i$ can apply only one type of operation on this object, $Sign(i)$, which consists of signing the log. The object keeps, by means of its internal state, a complete ordered list of the signatures in the log. As process $P_i$ signs the log, $i$ is added at the end of this ordered list, and $P_i$ retrieves the list of signatures up to, but not including, the one being applied by its current operation.

Consider the *History* object type defined by: $Q_{HISTORY} = RES_{HISTORY} \overset{def}{=} \mathbb{N}^*$, $OP_{HISTORY} \overset{def}{=} \{Sign(i) : i \in \mathbb{N}\}$, and transition function $\delta_{HISTORY}(\sigma, Sign(i)) \overset{def}{=} (\sigma \cdot i, \sigma)$. Let $O$ be an object of this type with initial state $\lambda$ and bindings which allow each process $P_i$ to apply operation $Sign(i)$ on $O$. It is not difficult to see that $O$ could be used to solve the Consensus problem among the $N$ processes in the system, no matter how large $N$ actually is. The consensus number of $O$ is thus $\infty$, and results by Herlihy [Her91] show that one could not implement $O$ from Registers and Fetch&Add objects in a system of 3 or more processes.

Informally, a *B-History* (short for *Blurred History*) object works as much like a History object as possible, under the restriction that a B-History object (with the same binding as above) can be implemented from Registers and Fetch&Add objects. The B-History object type has the same states and the same operations as the History object type, and it only differs from the latter in the results that are returned to the signing operations.

In order to restrict the power of the History object type, we will introduce a notion of *indistinguishability with respect to i* ($\sim_i$) between two finite sequences of signatures. As $P_i$ signs the log in a B-History object, it retrieves not the exact sequence $\sigma$ of signing operations applied thus far, but instead, the equivalence class $[\sigma]_{\sim_i}$ of sequences indistinguishable from $\sigma$ with respect to $i$. Before we give its formal definition, let us provide the intuition behind this indistinguishability notion. Given a finite sequence of integers, label each element with a superscript that indicates the number of elements with the same value occurring at or before that element. For example, the labeled version of the sequence $(1, 3, 2, 3, 1, 4)$ is $(1^1, 3^1, 2^1, 3^2, 1^2, 4^1)$. We say that two finite sequences $\sigma$ and $\sigma'$ are indistinguishable with respect to $i$ when the set of elements of both labeled sequences is the same, and every labeled element of the form $j^k$ has the same position in the two labeled sequences, except possibly when $i \neq j$ and this is the last occurrence of $j$ (i.e., $j$ occurs exactly $k$ times in $\sigma$ and $\sigma'$).

The *BH&AR* object type is a generalization of the B-History object type. The acronym stands for *B-History & Append-Registers*, as an object $O$ of this type encapsulates one B-History object $B$ and a set of MRSW Append-Registers $A_i$, one for every process $P_i$ in the system. The process $P_i$ is allowed to perform operations of the form $Sign\&Append(i, x)$ on $O$. Each atomic operation $Sign\&Append(i, x)$ performed by $P_i$ on $O$ consists of three parts: the value $x$ is appended to

$A_i$, a $Sign(i)$ operation is performed on $B$, and finally, the result of the $Sign(i)$ operation and the values of all Append-Registers $A_1, \ldots, A_N$ are returned as the result of the current ($Sign\&Append$) operation.

### 3.1.2 Notion of Indistinguishability

In the formal definitions which follow, $\mathbb{N}$ is the set of possible process indices and $V$ is a finite or countably infinite set of possible values to be used in the Append-Registers inside a BH&AR object. The notion of indistinguishability with respect to $i$ that we introduce is between two sequences over $\mathbb{N} \times V$, rather than over $\mathbb{N}$, in order to capture the generality of the BH&AR object type.

Let $\sigma \in (\mathbb{N} \times V)^*$ and $i \in \mathbb{N}$. We define $last(\sigma) \subset \mathbb{N}$ to be the set of indices in $\sigma$ of the last occurrences of any first coordinate in $\sigma$, and we define $last(\sigma, i) \subset \mathbb{N}$ to be the set $last(\sigma)$ excluding the index of the last occurrence of first coordinate $i$ in $\sigma$, if any. In the definitions below, notice that $\sigma[k]$ is a pair, and $(\sigma[k])[1]$ is the first coordinate of $\sigma[k]$.

$$last(\sigma) \overset{def}{=} \{k \in [1, |\sigma|] : \forall k', k < k' \leq |\sigma| \Rightarrow (\sigma[k])[1] \neq (\sigma[k'])[1]\}$$

$$last(\sigma, i) \overset{def}{=} \{k \in last(\sigma) : (\sigma[k])[1] \neq i\}$$

To exemplify these definitions, let $\overline{\sigma} = ((1,a),(2,b),(4,b),(3,a),(1,b),(3,a))$. We then have $last(\overline{\sigma}) = \{2,3,5,6\}$, $last(\overline{\sigma},1) = \{2,3,6\}$, $last(\overline{\sigma},2) = \{3,5,6\}$ and $last(\overline{\sigma},i) = \{2,3,5,6\}$ for any $i > 4$.

For any finite set $U \subset \mathbb{N}$, we define $MaxConsec(U)$ to be the set of maximal subsets of consecutive elements in $U$. Formally,

$$MaxConsec(U) \overset{def}{=} \{[a,b] : [a,b] \subseteq U, a-1 \notin U, \text{ and } b+1 \notin U\}.$$

As an example, for $\overline{\sigma}$ as above, we have $MaxConsec(last(\overline{\sigma},1)) = \{\{2,3\},\{6\}\}$ and $MaxConsec(last(\overline{\sigma},i)) = \{\{2,3\},\{5,6\}\}$ for any $i > 4$.

**Definition 3.1.** Let $\sigma, \sigma' \in (\mathbb{N} \times V)^*$ and let $i \in \mathbb{N}$. We say $\sigma$ and $\sigma'$ are *indistinguishable with respect to* $i$ and we write $\sigma \sim_i \sigma'$ if and only if the following three conditions hold:

1. $|\sigma'| = |\sigma|$; and
2. for all $k \in [1, |\sigma|]$, if $k \notin last(\sigma, i)$, then $\sigma'[k] = \sigma[k]$; and
3. for all $A \in MaxConsec(last(\sigma, i))$, $\sigma'|_A$ is a permutation of $\sigma|_A$.

To illustrate the definitions, with $\overline{\sigma}$ as before, we have

$$\overline{\sigma} \sim_i ((1,a),(2,b),(4,b),(3,a),(3,a),(1,b)), \text{ for all } i \notin \{1,3\},$$

$$\overline{\sigma} \sim_i ((1,a),(4,b),(2,b),(3,a),(1,b),(3,a)), \text{ for all } i \notin \{2,4\},$$

$$\overline{\sigma} \sim_i ((1,a),(4,b),(2,b),(3,a),(3,a),(1,b)), \text{ for all } i > 4.$$

We prove in Lemma 3.5 that $\sim_i$ is an equivalence relation. Before we do that, Lemma 3.2 establishes the fact that two sequences which are indistinguishable with respect to some $i$ must be a permutation of one another. Lemmas 3.3 and 3.4 contain the results needed to establish Lemma 3.5.

**Lemma 3.2.** *Let $\sigma, \sigma' \in (\mathbb{N} \times V)^*$. If there exists $i \in \mathbb{N}$ such that $\sigma \sim_i \sigma'$, then $\sigma'$ is a permutation of $\sigma$.*

*Proof.* A permutation which transforms $\sigma$ in $\sigma'$ can be obtained by composing the individual permutations mentioned by condition 3 of Definition 3.1 with the identity permutation on the set $[1, |\sigma|] \setminus last(\sigma, i)$. $\square$

**Lemma 3.3.** *Let $\sigma, \sigma' \in (\mathbb{N}\times V)^*$ and $i \in \mathbb{N}$ and suppose that $\sigma \sim_i \sigma'$. Then for any $k$, $(\sigma[k])[1] = i$ if and only if $(\sigma'[k])[1] = i$. Informally, the value $i$ occurs as first coordinate in the same positions in $\sigma$ and in $\sigma'$.*

*Proof.* If $(\sigma[k])[1] = i$, then $k \notin last(\sigma, i)$, so by condition 2 in Definition 3.1, we have $\sigma'[k] = \sigma[k]$, so $(\sigma'[k])[1] = i$.

Now suppose that $(\sigma'[k])[1] = i$ and $(\sigma[k])[1] \neq i$. By condition 2 in Definition 3.1, if $\sigma[k] \neq \sigma'[k]$, then $k \in last(\sigma, i)$. By condition 3 in Definition 3.1, there exist $A \in MaxConsec(last(\sigma, i))$ and $j \in A$ such that $k \in A$ and $\sigma'[k] = \sigma[j]$. But then $(\sigma[j])[1] = i$, so $j \notin last(\sigma, i)$. This is a contradiction, because $j \in A \subseteq last(\sigma, i)$. Therefore, if $(\sigma'[k])[1] = i$, then $(\sigma[k])[1] = i$. □

**Lemma 3.4.** *Let $\sigma, \sigma' \in (\mathbb{N} \times V)^*$ and $i \in \mathbb{N}$ and suppose that $\sigma \sim_i \sigma'$. Then $last(\sigma) = last(\sigma')$.*

*Proof.* Let $k \in last(\sigma)$.

If $(\sigma[k])[1] = i$, then $k$ is the index of the last occurrence of first coordinate $i$ in $\sigma$. By Lemma 3.3, $i$ occurs as first coordinate in the same positions in $\sigma$ and $\sigma'$, so $k$ is also the index of the last occurrence of first coordinate $i$ in $\sigma'$. Therefore $k \in last(\sigma')$.

Now assume that $(\sigma[k])[1] \neq i$, so $k \in last(\sigma, i)$. We want to prove that $k \in last(\sigma')$.

By condition 3 in Definition 3.1, there exist $A \in MaxConsec(last(\sigma, i))$ and $j \in A$ such that $k \in A$ and $\sigma[j] = \sigma'[k]$. Let $p = (\sigma[j])[1] \neq i$. Since $j \in A$, we know that $j$ is the index of the last occurrence of first coordinate $p$ in $\sigma$. Let $B$ be the set of indices of occurrences of first coordinate $p$ in $\sigma$ other than its last (at index $j$). Then $B \cap last(\sigma, i) = \emptyset$. Since $A$ is a maximal subset of consecutive indices in $last(\sigma, i)$, $j \in A$ and $j$ is greater than the elements in $B$, any element of $B$ is less than any element of $A$. By condition 2 in Definition 3.1, $p$ occurs as first coordinate in $\sigma'$ at every index in $B$. Furthermore, $p$ occurs as first coordinate in $\sigma'$ at index $k \in A$, and $k$ is greater than any element in $B$. By Lemma 3.2, $\sigma$ and $\sigma'$ contain the same number of occurrences of first coordinate $p$, therefore the last occurrence of first coordinate $p$ in $\sigma'$ is at index $k$. Hence, $k \in last(\sigma')$.

This proves that $last(\sigma) \subseteq last(\sigma')$. Using a similar argument, $last(\sigma') \subseteq last(\sigma)$. □

The next result follows easily from Lemma 3.4:

**Lemma 3.5.** *For any $i$, $\sim_i$ is an equivalence relation on $(\mathbb{N} \times V)^*$.*

*Proof.* Reflexivity is trivial, using the identity permutation in the third condition of Definition 3.1.

For symmetry, suppose that $\sigma \sim_i \sigma'$. Condition 1 for $\sigma' \sim_i \sigma$ is the same as for $\sigma \sim_i \sigma'$. By Lemma 3.4, condition 2 in the definition of $\sigma' \sim_i \sigma$ is the same as condition 2 in the definition of $\sigma \sim_i \sigma'$. By using the inverse permutation from condition 3 for $\sigma \sim_i \sigma'$, we obtain condition 3 for $\sigma' \sim_i \sigma$.

For transitivity, suppose that $\sigma \sim_i \sigma'$ and $\sigma' \sim_i \sigma''$. Condition 1 for $\sigma \sim_i \sigma''$ is satisfied. We use Lemma 3.4 to obtain condition 2 for $\sigma \sim_i \sigma''$. To get condition 3 for $\sigma \sim_i \sigma''$, we compose the permutations from condition 3 for the two hypotheses. □

For any $\sigma \in (\mathbb{N} \times V)^*$ and $a \in \mathbb{N}$, let $SubseqVal(\sigma, a) \in V^*$ be the sequence of values which occur in $\sigma$ in pairs whose first coordinate is $a$. The following Lemma will also be useful later:

**Lemma 3.6.** *Let $\sigma, \sigma' \in (\mathbb{N} \times V)^*$ and let $i \in \mathbb{N}$. Suppose that $\sigma \sim_i \sigma'$. Then for any $a \in \mathbb{N}$, $SubseqVal(\sigma, a) = SubseqVal(\sigma', a)$.*

*Proof.* Let $B$ be the set of indices of pairs in $\sigma$ whose first coordinate is $a$, and $B'$ be the similar set of indices in $\sigma'$. So $SubseqVal(\sigma, a)$ is the sequence of second coordinates of pairs in $\sigma|_B$. To prove this Lemma, we show that $\sigma|_B = \sigma'|_{B'}$. By Lemma 3.2, $|B| = |B'|$.

If $|B| = 0$, then no pair in either $\sigma$ or $\sigma'$ has first coordinate $a$, so $SubseqVal(\sigma, a) = SubseqVal(\sigma', a) = \lambda$.

Now assume that $|B| \geq 1$. Let $k = \max(B)$ be the index in $\sigma$ of the last occurrence of a pair with first coordinate $a$. We know $|B'| = |B| \geq 1$, so we can let $k' = \max(B')$ be the similar index in $\sigma'$. By Lemmas 3.3 and 3.4, $last(\sigma, i) = last(\sigma', i)$. For any $j \in B \setminus \{k\}$, $j \notin last(\sigma, i)$, so by condition 2 in Definition 3.1, $\sigma'[j] = \sigma[j]$. For any $j' \in B' \setminus \{k'\}$, $j' \notin last(\sigma', i) = last(\sigma, i)$, so by the same condition, $\sigma'[j'] = \sigma[j']$. Therefore $\sigma|_{B\setminus\{k\}} = \sigma'|_{B'\setminus\{k'\}}$.

By condition 3 in Definition 3.1, there exists $A \in MaxConsec(last(\sigma, i))$ and $j \in A$ such that $\sigma'[j] = \sigma[k]$. But then $j \in last(\sigma, i) = last(\sigma', i)$ and $\sigma'[j]$ has first coordinate $a$, therefore $j = k'$. So $\sigma'[k'] = \sigma[k]$, and hence, $\sigma|_B = \sigma'|_{B'}$. □

The idea behind the next technical Lemma is that whenever $P_i$ applies a Sign&Write operation to a BH&AR object which is in state $\sigma$, it retrieves enough information to compute the result obtained by some (possibly other) process $P_{i'}$ in its $l$-th step preceding that of $P_i$, provided that either $i = i'$ or $l$ is not the last step by $P_{i'}$ in $\sigma$.

**Lemma 3.7.** *Let $V$ be a finite or countably infinite set. Let $\sigma, \sigma' \in (\mathbb{N} \times V)^*$ and $i \in \mathbb{N}$ be such that $\sigma \sim_i \sigma'$. Let $j \in [1, |\sigma'|] \setminus last(\sigma', i)$. Let $i'$ be the first coordinate of $\sigma'[j]$. Then the prefixes of length $j-1$ of $\sigma$ and $\sigma'$ are indistinguishable with respect to $i'$.*

*Proof.* Notice that by condition 1 in Definition 3.1, $|\sigma'| = |\sigma|$, and by Lemma 3.4, $last(\sigma', i) = last(\sigma, i)$. Therefore, $j \in [1, |\sigma|] \setminus last(\sigma, i)$, and by condition 2 from the same definition, we get $\sigma'[j] = \sigma[j]$. Let $\tau$ and $\tau'$ denote the prefixes of length $j-1$ of $\sigma$ and $\sigma'$, respectively. Notice that $|\tau'| = |\tau| = j-1$, so condition 1 in Definition 3.1 of $\tau \sim_{i'} \tau'$ holds.

We now prove that condition 2 in the definition of $\tau \sim_{i'} \tau'$ holds. Let $k \in [1, |\tau|] \setminus last(\tau, i')$ and let $a = (\tau[k])[1]$. We claim that $k \notin last(\sigma)$. To see why, consider two possible cases. If $a \neq i'$ then $k \notin last(\tau)$, and since $\tau$ is a prefix of $\sigma$, $k \notin last(\sigma)$. Otherwise, $a = i'$, and now $a$ occurs as first coordinate in $\sigma$ at indices $k$ and $j$, with $k < j$, therefore $k \notin last(\sigma)$. Hence, in either case, $k \notin last(\sigma)$, so $k \notin last(\sigma, i)$. Then, by condition 2 in the definition of $\sigma \sim_i \sigma'$, we have $\sigma'[k] = \sigma[k]$ and therefore, $\tau'[k] = \tau[k]$. This shows that condition 2 in the definition of $\tau \sim_{i'} \tau'$ holds.

To complete the proof of this Lemma, it is enough to show that condition 3 in the definition of $\tau \sim_{i'} \tau'$ holds. To do this, we first claim that $last(\sigma, i) \cap [1, j-1] \subseteq last(\tau, i')$. Let $k \in last(\sigma, i) \cap [1, j-1]$ and let $a = (\sigma[k])[1]$. So $k$ is the index of the last occurrence of first coordinate $a \neq i$ in $\sigma$. Since $\tau$ is a prefix of $\sigma$, $k$ is also the index of the last occurrence of first coordinate $a$ in $\tau$. Therefore, $k \in last(\tau)$. Furthermore, since $i'$ occurs as first coordinate at index $j > k$ in $\sigma$, we have $a \neq i'$. Hence, $k \in last(\tau, i')$ as claimed.

Next, we claim that for all $A \in MaxConsec(last(\tau, i'))$, the family of subsets $\{B \in MaxConsec(last(\sigma, i)) : B \cap A \neq \emptyset\}$ partitions $A \cap last(\sigma, i)$. To prove this, first notice that the sets in $MaxConsec(last(\sigma, i))$ are mutually disjoint. Next, for all $k \in A \cap last(\sigma, i)$, there exists a $B \in MaxConsec(last(\sigma, i))$ such that $k \in B$ and therefore, $B \cap A \neq \emptyset$. Hence,

$$A \subseteq \cup\{B \in MaxConsec(last(\sigma, i)) : B \cap A \neq \emptyset\}.$$

In order to prove the other direction of this set inclusion, let $B \in MaxConsec(last(\sigma, i))$ such that $B \cap A \neq \emptyset$. We want to show that $B \subseteq A$. Let $k \in B \cap A$ (we know $k$ exists). Since $k \in A$ and $A \subseteq last(\tau, i') \subseteq [1, j-1]$, we have $k < j$. But, by definition, $j \notin last(\sigma', i) = last(\sigma, i)$. Since $B$ is a maximal subset of consecutive elements in $last(\sigma, i)$, $B$ contains the element $k < j$ and $j \notin last(\sigma, i)$, we can conclude that $B \subseteq [1, j-1]$. Now, for any element $k' \in B$, $k' \in last(\sigma, i) \cap [1, j-1]$, so by the previous claim, $k' \in last(\tau, i')$. Therefore $B$ is a subset of consecutive indices in $last(\tau, i')$, but not necessarily maximal. But $A$ itself is a maximal subset of consecutive indices in $last(\tau, i')$. Hence, if $B \cap A \neq \emptyset$, then $B \subseteq A$.

Finally, we are able to prove condition 3 from the definition of $\tau \sim_{i'} \tau'$. Let $A \in MaxConsec(last(\tau, i'))$. Since $\{A \setminus last(\sigma, i), A \cap last(\sigma, i)\}$ is a partition of A, by the previous claim, the following family of subsets also partitions $A$:

$$\{\{k\} : k \in A \setminus last(\sigma, i)\} \cup \{B \in MaxConsec(last(\sigma, i)) : B \cap A \neq \emptyset\}.$$

By condition 2 in the definition of $\sigma \sim_i \sigma'$, $\sigma'[k] = \sigma[k]$ for all $k \notin last(\sigma, i)$, in particular for all $k \in A \setminus last(\sigma, i)$. By condition 3 in the same definition, $\sigma'|_B$ is a permutation of $\sigma|_B$ for all $B \in MaxConsec(last(\sigma, i))$, in particular for those $B$ such that $B \cap A \neq \emptyset$. Hence, a permutation which transforms $\sigma|_A = \tau|_A$ into $\sigma'|_A = \tau'|_A$ can be obtained by composing the identity permutation on the singleton sets $\{k\}$ for $k \in A \setminus last(\sigma, i)$ with the permutation given by condition 3 of $\sigma \sim_i \sigma'$ on the sets $B \in MaxConsec(last(\sigma, i))$ such that $B \cap A \neq \emptyset$. $\square$

### 3.1.3 Formal Object Type Definitions

For $\sigma \in (\mathbb{N} \times V)^*$ and $i \in \mathbb{N}$, let $[\sigma]_{\sim_i}$ denote the equivalence class of $\sigma$ with respect to the equivalence relation $\sim_i$. We are now ready to define the BH&AR$\langle V \rangle$ object type:

- $Q_{BH\&AR\langle V \rangle} \overset{def}{=} (\mathbb{N} \times V)^*$;
- $OP_{BH\&AR\langle V \rangle} \overset{def}{=} \{Sign\&Append(i, x) : i \in \mathbb{N} \text{ and } x \in V\}$;
- $RES_{BH\&AR\langle V \rangle} \overset{def}{=} \{[\sigma]_{\sim_i} : \sigma \in (\mathbb{N} \times V)^* \text{ and } i \in \mathbb{N}\}$; and
- for all $\sigma \in Q_{BH\&AR\langle V \rangle}$ and for all $Sign\&Append(i, x) \in OP_{BH\&AR\langle V \rangle}$, we have

$$\delta_{BH\&AR\langle V \rangle}(\sigma, Sign\&Append(i, x)) \overset{def}{=} (\sigma \cdot (i, x), [\sigma]_{\sim_i})$$

In this thesis, any object $O$ of the BH&AR$\langle V \rangle$ type has initial state $q_O = \lambda$ and bindings $b_O(P_i) = \{Sign\&Append(i, x) : x \in V\}$. Notice that with these bindings, in a distributed system with $N$ processes, an object of the BH&AR$\langle V \rangle$ type can only reach states $q$ such that $q \in ([1, N] \times V)^*$.

For any $\sigma \in \mathbb{N}^*$, let $\sigma \times \bot$ be the sequence in $(\mathbb{N} \times \{\bot\})^*$ defined by $(\sigma \times \bot)[i] = (\sigma[i], \bot)$, for all $i \in [1, |\sigma|]$. We can now extend the definition of $\sim_i$ over sequences of positive integers: for any $\sigma, \sigma' \in \mathbb{N}^*$, $\sigma \sim_i \sigma'$ if and only if $(\sigma \times \bot) \sim_i (\sigma' \times \bot)$. It should be clear that $\sim_i$ is also an equivalence relation on $\mathbb{N}^*$, so that we can write $[\sigma]_{\sim_i}$ for the equivalence class of $\sigma \in \mathbb{N}^*$ with respect to the equivalence relation $\sim_i$.

The formal definition of the B-History object type is as follows:

- $Q_{B-HISTORY} \overset{def}{=} \mathbb{N}^*$;
- $OP_{B-HISTORY} \overset{def}{=} \{Sign(i) : i \in \mathbb{N}\}$;
- $RES_{B-HISTORY} \overset{def}{=} \{[\sigma]_{\sim_i} : \sigma \in \mathbb{N}^*, i \in \mathbb{N}\}$; and
- For all $\sigma \in Q_{B-HISTORY}$ and $i \in \mathbb{N}$, $\delta_{B-HISTORY}(\sigma, Sign(i)) \overset{def}{=} (\sigma \cdot i, [\sigma]_{\sim_i})$.

In this thesis, any object $O$ of the B-History type will have initial state $q_O = \lambda$ and bindings $r_O(P_i) = \{Sign(i)\}$. Notice that with these bindings, in a distributed system with $N$ processes, an object of the B-History type can only reach states $q$ such that $q \in [1, N]^*$.

An explanation is in order with respect to the relationship between the B-History and BH&AR object types. Notice that, formally, the B-History object type is almost identical to the BH&AR$\langle V \rangle$ object type, for any $V$ containing a single element. The acronym BH&AR stands for B-History & Append-Registers, because any object $O$ of the BH&AR object type can be seen as encapsulating one B-History object $B$, and several MRSW Append-Registers $A_i$, one for each process $P_i$ in the system. With these formal definitions, the B-History object $B$ encapsulated in state $\sigma$ of $O$ is represented by the sequence of first coordinates of pairs in $\sigma$. The value of the MRSW Append-Register object $A_i$ corresponding to $P_i$, encapsulated in the state $\sigma$ of $O$, is the sequence $SubseqVal(\sigma, i)$. The newest value appended to that register is the last value in the latter sequence.

## 3.2 Implementing a B-History Object

In this section, we prove:

**Theorem 3.8.** *There exists a 1-bounded wait-free linearizable implementation of a B-History object from one Fetch&Add object.*

In order to present the implementation, we will need some notation.

**Mapping of Tuples to Non-Negative Integers** Let $\mathbb{Z}$ be the set of integers and let $\mathbb{N}$ be the set of positive integers. Throughout this section, let $\gamma$ denote a bijective mapping from $\mathbb{Z}^{N+1}$ into $\mathbb{N} \cup \{0\}$ which exists, since $\mathbb{Z}^{N+1}$ is countably infinite (recall that $N$ is the number of processes in our distributed system). Let $\gamma^{-1}$ be the inverse of $\gamma$. The function $\gamma$ determines a natural bijection $\overline{\gamma}$ between finite subsets of $\mathbb{Z}^{N+1}$ and non-negative integers. For any finite $A \subset \mathbb{Z}^{N+1}$, the sequence of bits in the binary representation of the non-negative integer $\overline{\gamma}(A)$ (starting from its least significant bit) represents the characteristic function of $A$. Formally,

$$\overline{\gamma}(A) \overset{def}{=} \sum_{x \in A} 2^{\gamma(x)}$$

and in particular, $\overline{\gamma}(\emptyset) = 0$. Let $\overline{\gamma}^{-1}$ denote the inverse of $\overline{\gamma}$.

As an example, let $a_0, a_1, \ldots$ be an enumeration of $\mathbb{Z}^{N+1}$ (that is, $a_i = \gamma^{-1}(i)$). Then the positive integer $11 = 2^3 + 2^1 + 2^0$ encodes the finite subset $\overline{\gamma}^{-1}(11) = \{a_0, a_1, a_3\} \subset \mathbb{Z}^{N+1}$.

Notice that for any finite $A, B \subset \mathbb{Z}^{N+1}$,

$$A \cap B = \emptyset \Longrightarrow \overline{\gamma}(A \cup B) = \overline{\gamma}(A) + \overline{\gamma}(B)$$

**Tuples** In the following definitions, let $\sigma \in \mathbb{N}^*$. We will associate a series of tuples with $\sigma$, whose purpose is to describe $\sigma$. The relation between the tuples associated with $\sigma$ and the equivalence class of $\sigma$ will become apparent in Lemma 3.12.

For $k \in [1, |\sigma|]$, we define $tup(\sigma, k)$ to be the $(N+1)$-tuple $(\sigma[k], t_1, \ldots, t_N)$ such that for all $j \in [1, N]$, $t_j \overset{def}{=} count(\sigma|_{[1,k-1]}, j)$ is the number of occurrences of $j$ strictly before index $k$ in $\sigma$. Notice that $k = 1 + \sum_{j=1}^{N} t_j$.

For $k \in [1, |\sigma|]$, we define $pre\_tup(\sigma, k)$ as follows. If $k$ is the index of the first occurrence of $\sigma[k]$ in $\sigma$, then we give $pre\_tup(\sigma, k)$ a dummy value, $pre\_tup(\sigma, k) \overset{def}{=} (\sigma[k], -1, \ldots, -1)$. Otherwise, let $k' < k$ be the index of the previous occurrence of $\sigma[k]$ in $\sigma$, and let $pre\_tup(\sigma, k) \overset{def}{=} tup(\sigma, k')$.

We define $Tuples(\sigma)$ and $Tuples(\sigma, i)$ to be the following finite subsets of $\mathbb{Z}^{N+1}$:

$$Tuples(\sigma) \overset{def}{=} \{pre\_tup(\sigma, k) : 1 \le k \le |\sigma|\}$$

and

$$Tuples(\sigma, i) \overset{def}{=} Tuples(\sigma) \cup \{tup(\sigma, k) : \sigma[k] = i \text{ and } k \in last(\sigma)\}$$

Notice that there is at most one $k$ such that $\sigma[k] = i$ and $k \in last(\sigma)$, so $Tuples(\sigma, i)$ has at most one more element than $Tuples(\sigma)$. With these definitions, $Tuples(\lambda) = Tuples(\lambda, i) = \emptyset$, for any $i$.

The following Lemma establishes some basic facts about the tuples in $Tuples(\sigma)$:

**Lemma 3.9.** *Let $\sigma \in [1, N]^*$ and let $i \in [1, N]$. Then:*

- *for any $k \neq k'$, $tup(\sigma, k) \neq tup(\sigma, k')$;*
- *for any $k \neq k'$, $pre\_tup(\sigma, k) \neq pre\_tup(\sigma, k')$;*

- *for any a*, $|\{\tau \in Tuples(\sigma) : \tau[1] = x\}| = count(\sigma, x)$;
- $|Tuples(\sigma)| = |\sigma|$.

*Proof.* Fix $\sigma$ and $i$. Let $k, k' \in [1, len(\sigma)]$ be such that $k \neq k'$.

Let $tup(\sigma, k) = (a, t_1, \ldots, t_N)$ and let $tup(\sigma, k') = (a', t'_1, \ldots, t'_N)$. Since $k = 1 + \sum_{j=1}^{N} t_j$, $k' = 1 + \sum_{j=1}^{N} t'_j$ and $k \neq k'$, we have $tup(\sigma, k) \neq tup(\sigma, k')$.

We now want to prove that $pre\_tup(\sigma, k) \neq pre\_tup(\sigma, k')$. If $\sigma[k] \neq \sigma[k']$, the two tuples differ in their first coordinate. Now assume that $\sigma[k] = \sigma[k']$. Without loss of generality, assume that $k < k'$. Let $j'$ be the index of the last occurrence of $\sigma[k']$ before index $k'$. Since $\sigma[k'] = \sigma[k]$ and $k < k'$, we know that $k \leq j' < k'$. By definition, $pre\_tup(\sigma, k') = tup(\sigma, j')$.

If $\sigma[k]$ occurs in $\sigma$ before index $k$, let $j$ be its last occurrence before index $k$. Then $j < k \leq j'$, so $j \neq j'$. By definition, $pre\_tup(\sigma, k) = tup(\sigma, j)$, and by the previous assertion in the Lemma, $tup(\sigma, j) \neq tup(\sigma, j')$, therefore $pre\_tup(\sigma, k) \neq pre\_tup(\sigma, k')$.

If, on the other hand, $\sigma[k]$ does not occur in $\sigma$ before index $k$, then $pre\_tup(\sigma, k)$ has the dummy value with negative coordinates, while $pre\_tup(\sigma, k') = tup(\sigma, j')$ has non-negative coordinates.

Hence, in all cases, $pre\_tup(\sigma, k) \neq pre\_tup(\sigma, k')$, as required.

For the third assertion, notice that, for any $j$, the first coordinate of $pre\_tup(\sigma, j)$ is $\sigma[j]$. By the second part of this Lemma, $pre\_tup(\sigma, j) \neq pre\_tup(\sigma, j')$ whenever $j \neq j'$. Then, by definition of $Tuples(\sigma)$, the number of tuples in $Tuples(\sigma)$ which have their first coordinate $x$ is equal to the number of times $x$ occurs in $\sigma$, $count(\sigma, x)$, for any $x$.

Now for the last assertion, $Tuples(\sigma)$ is defined as $\{pre\_tup(\sigma, k) : 1 \leq k \leq |\sigma|\}$. By the second assertion, $pre\_tup(\sigma, k) \neq pre\_tup(\sigma, k')$ when $k \neq k'$. Therefore $|Tuples(\sigma)| = |\sigma|$. ☐

The following Lemma provides a connection between the two sets $Tuples(\sigma, i)$ and $last(\sigma, i)$, and its Corollary will be essential in proving Lemma 3.12.

**Lemma 3.10.** *Let* $\sigma \in [1, N]^*$ *and let* $i \in [1, N]$. *Let* $t_1, \ldots, t_N$ *be* $N$ *non-negative integers and let* $k = 1 + \sum_{b=1}^{N} t_b$. *Let* $a$ *be a positive integer, and let* $\tau = (a, t_1, \ldots, t_N)$. *Then* $\tau \in Tuples(\sigma, i)$ *if and only if* $\tau = tup(\sigma, k)$ *and* $k \notin last(\sigma, i)$.

*Proof.* $\tau \in Tuples(\sigma, i)$ if and only if either $\tau \in Tuples(\sigma)$ or there exists $j$ such that $\tau = tup(\sigma, j)$ and $j$ is the index of the last occurrence of $i$ in $\sigma$.

Furthermore, $\tau \in Tuples(\sigma)$ if and only if there exists $j'$ such that $\tau = pre\_tup(\sigma, j')$. Since $\tau$ has non-negative coordinates, $\tau = pre\_tup(\sigma, j')$ if and only if there exists $j < j'$ such that $\tau = tup(\sigma, j)$ and $\sigma[j'] = \sigma[j]$. Hence $\tau \in Tuples(\sigma)$ if and only if there exists $j$ such that $\tau = tup(\sigma, j)$ and $j \notin last(\sigma)$.

Notice that $j \notin last(\sigma, i)$ if and only if either $j \notin last(\sigma)$ or $j$ is the last occurrence of $i$ in $\sigma$.

Combining the equivalences, $\tau \in Tuples(\sigma, i)$ if and only if there exists $j$ such that $\tau = tup(\sigma, j)$ and either $j \notin last(\sigma)$ or $j$ is the index of the last occurrence of $i$ in $\sigma$, if and only if there exists $j$ such that $\tau = tup(\sigma, j)$ and $j \notin last(\sigma, i)$.

But if $\tau = tup(\sigma, j)$, then $j = k = 1 + \sum_{b=1}^{N} t_b$. So there exists $j$ such that $\tau = tup(\sigma, j)$ and $j \notin last(\sigma, i)$ if and only if $\tau = tup(\sigma, k)$ and $k \notin last(\sigma, i)$. ☐

**Corollary 3.11.** *Let* $\sigma, \sigma' \in \mathbb{N}^*$, *let* $i$ *be a positive integer and let* $k \notin last(\sigma, i)$. *If* $tup(\sigma, k) \in Tuples(\sigma', i)$, *then* $\sigma'[k] = \sigma[k]$ *and* $\sigma'|_{[1,k-1]}$ *is a permutation of* $\sigma|_{[1,k-1]}$.

*Proof.* Let $\tau = (a, t_1, \ldots, t_N) = tup(\sigma, k)$. By definition of $tup(\sigma, k)$, we have $k = 1 + \sum_{b=1}^{N} t_i$, $a = \sigma[k]$ and $t_1, \ldots, t_N$ are non-negative integers. Since $\tau \in Tuples(\sigma', i)$ and all variables in this Corollary fit the description in Lemma 3.10, by this Lemma, $\tau = tup(\sigma', k)$. Hence $\sigma'[k] = a = \sigma[k]$. Furthermore, for any $j \in [1, N]$, the number of occurrences of $j$ strictly before index $k$ is exactly $t_j$ in both $\sigma$ and $\sigma'$. Therefore, the prefix of $\sigma'$ of length $k - 1$ is a permutation of the similar prefix of $\sigma$. ☐

The next Lemma will be used to compute the equivalence class $[\sigma]_{\sim_i}$ when only $\mathit{Tuples}(\sigma, i)$ is provided, and not $\sigma$ itself.

**Lemma 3.12.** *Let $\sigma \in [1, N]^*$ and let $i \in [1, N]$. For any $\sigma' \in [1, N]^*$, if $\mathit{Tuples}(\sigma', i) = \mathit{Tuples}(\sigma, i)$, then $\sigma \sim_i \sigma'$.*

*Proof.* We prove the contrapositive. Fix $\sigma, \sigma' \in [1, N]^*$ and $i \in [1, N]$ such that $\sigma \not\sim_i \sigma'$. We want to prove that $\mathit{Tuples}(\sigma', i) \neq \mathit{Tuples}(\sigma, i)$.

If $i$ occurs in $\sigma$, but not in $\sigma'$, then $\mathit{Tuples}(\sigma)$ contains at least one tuple with first coordinate $i$, while $\mathit{Tuples}(\sigma')$ contains none. Furthermore, $\mathit{Tuples}(\sigma, i)$ contains at least two tuples with first coordinate $i$, and $\mathit{Tuples}(\sigma', i)$ still contains none. Then $\mathit{Tuples}(\sigma, i) \neq \mathit{Tuples}(\sigma', i)$. From now on, assume that $i$ occurs in either both or none of $\sigma$ and $\sigma'$.

Assume that $|\sigma| \neq |\sigma'|$. If $i$ occurs in both, $|\mathit{Tuples}(\sigma, i)| = |\mathit{Tuples}(\sigma)| + 1$, by Lemma 3.9, $|\mathit{Tuples}(\sigma)| = |\sigma|$, so $|\mathit{Tuples}(\sigma, i)| = |\sigma| + 1$ and similarly $|\mathit{Tuples}(\sigma', i)| = |\sigma'| + 1$. Since $|\sigma| \neq |\sigma'|$, we get $\mathit{Tuples}(\sigma, i) \neq \mathit{Tuples}(\sigma', i)$. If $i$ occurs in neither, $|\mathit{Tuples}(\sigma, i)| = |\mathit{Tuples}(\sigma)| = |\sigma|$ and $|\mathit{Tuples}(\sigma', i)| = |\sigma'|$. Since $|\sigma| \neq |\sigma'|$, we get $\mathit{Tuples}(\sigma, i) \neq \mathit{Tuples}(\sigma', i)$. From now on, assume that $|\sigma| = |\sigma'|$, so the first condition in the definition of $\sigma \sim_i \sigma'$ holds.

Now assume that the second condition in the definition of $\sigma \sim_i \sigma'$ does not hold. Specifically, assume that there exists $k$ such that $k \notin \mathit{last}(\sigma, i)$ and $\sigma'[k] \neq \sigma[k]$. By Lemma 3.10, $\mathit{tup}(\sigma, k) \in \mathit{Tuples}(\sigma, i)$. Since $\sigma'[k] \neq \sigma[k]$, by the contrapositive of Corollary 3.11, $\mathit{tup}(\sigma, k) \notin \mathit{Tuples}(\sigma', i)$ and hence $\mathit{Tuples}(\sigma, i) \neq \mathit{Tuples}(\sigma', i)$. From now on, assume that the second condition in the definition of $\sigma \sim_i \sigma'$ holds.

Now suppose that $\mathit{last}(\sigma, i) \neq \mathit{last}(\sigma', i)$. Without loss of generality, there exists $k \notin \mathit{last}(\sigma, i)$ such that $k \in \mathit{last}(\sigma', i)$. Let $\tau = (a, t_1, \ldots, t_N) = \mathit{tup}(\sigma, k)$. By Lemma 3.10, $\tau \in \mathit{Tuples}(\sigma, i)$. To obtain a contradiction, suppose that $\tau \in \mathit{Tuples}(\sigma', i)$. By the same Lemma, $1 + \sum_{b=1}^{N} t_b \notin \mathit{last}(\sigma', i)$. But $k = 1 + \sum_{b=1}^{N} t_b$, so $k \notin \mathit{last}(\sigma', i)$. This is a contradiction, hence $\mathit{Tuples}(\sigma, i) \neq \mathit{Tuples}(\sigma', i)$. From now on, assume that $\mathit{last}(\sigma, i) = \mathit{last}(\sigma', i)$.

Since $\sigma \not\sim_i \sigma'$, the only scenario we still have to analyze is as follows: the first two conditions of $\sigma \sim_i \sigma'$ hold, $\mathit{last}(\sigma, i) = \mathit{last}(\sigma', i)$, and the third (and last) condition of $\sigma \sim_i \sigma'$ does not hold. Specifically, without loss of generality, there exists $A \in \mathit{MaxConsec}(\mathit{last}(\sigma, i))$ such that $\sigma|_A$ is not a permutation of $\sigma'|_A$. Since $A$ contains indices of the last occurrences in $\sigma$ of various positive integers, $\sigma|_A$ contains distinct integers. Since $\mathit{last}(\sigma, i) = \mathit{last}(\sigma', i)$, $\sigma'|_A$ also contains distinct integers. Then $\sigma'|_A$ is not a permutation of $\sigma|_A$ if and only if there exists $k \in A$ such that $\sigma[k] \notin \{\sigma'[j] : j \in A\}$. Since $k \in A \subseteq \mathit{last}(\sigma, i)$, we have $\sigma[k] \neq i$. Let $a = \sigma[k]$.

By Lemma 3.9, the number of tuples in $\mathit{Tuples}(\sigma)$ which have their first coordinate $x$ is equal to the number of times $x$ occurs in $\sigma$, for any $x$. Furthermore, $\mathit{Tuples}(\sigma, i)$ may contain only one additional tuple, whose first coordinate is $i$. Therefore, the number of tuples in $\mathit{Tuples}(\sigma, i)$ whose first coordinate is $a \neq i$ is equal to $\mathit{count}(\sigma, a)$. In what follows, we prove that $\mathit{count}(\sigma', a) \neq \mathit{count}(\sigma, a)$. Then, as a consequence, $\mathit{Tuples}(\sigma, i) \neq \mathit{Tuples}(\sigma', i)$.

We know $k$ is the index of the last occurrence of $a$ in $\sigma$. Let $k'$ be the index of the last occurrence of $a$ in $\sigma'$. Then $k' \in \mathit{last}(\sigma', i) = \mathit{last}(\sigma, i)$, and there exists $A' \in \mathit{MaxConsec}(\mathit{last}(\sigma, i))$ such that $k' \in A'$. By choice of $k$, we have $k \in A$ and $\sigma[k] \notin \{\sigma'[j] : j \in A\}$. Since $\sigma'[k'] = \sigma[k] = a$, we have $k' \notin A$. The sets in $\mathit{MaxConsec}(\mathit{last}(\sigma, i))$ partition the set $\mathit{last}(\sigma, i)$, so $A \neq A'$ and in fact $A \cap A' = \emptyset$. Both $A$ and $A'$ are maximal subsets of consecutive indices in $\mathit{last}(\sigma, i)$, so without loss generality, there exists $j \notin \mathit{last}(\sigma, i)$ such that all elements in $A$ are less than $j$ and all elements in $A'$ are greater than $j$. In particular, $k < j < k'$.

By Lemma 3.10, $\mathit{tup}(\sigma, j) \in \mathit{Tuples}(\sigma, i)$. If $\mathit{tup}(\sigma, j) \notin \mathit{Tuples}(\sigma', i)$, $\mathit{Tuples}(\sigma, i) \neq \mathit{Tuples}(\sigma', i)$. So assume that $\mathit{tup}(\sigma, j) \in \mathit{Tuples}(\sigma', i)$. By Corollary 3.11, $\sigma'|_{[1,j-1]}$ is a permutation of $\sigma|_{[1,j-1]}$, therefore $\mathit{count}(\sigma'|_{[1,j-1]}, a) = \mathit{count}(\sigma|_{[1,j-1]}, a)$. By definition of $k$, all occurrences of $a$ in $\sigma$ are strictly before index $j > k$, therefore $\mathit{count}(\sigma, a) = \mathit{count}(\sigma_{[1,j-1]}, a)$. By definition of $k'$, $a$ occurs in $\sigma'$ at index $k' > j$, so $\mathit{count}(\sigma', a) > \mathit{count}(\sigma'|_{[1,j-1]}, a)$. Therefore $\mathit{count}(\sigma', a) \neq \mathit{count}(\sigma, a)$, and by the argument above, $\mathit{Tuples}(\sigma, i) \neq \mathit{Tuples}(\sigma', i)$. ☐

**The Implementation** Informally, the implementation works as follows. Let $V$ be the integer value of the shared Fetch&Add object. Throughout the execution of the implementation, we maintain the invariant that after the sequence $\sigma$ of operations have been performed, $\overline{\gamma}^{-1}(V) = \mathit{Tuples}(\sigma)$. At the beginning of the execution of each access procedure $P_i : \mathit{Sign}(i)$, the local variable $\tau$ is a new $(N+1)$-tuple to be added to the set $\overline{\gamma}^{-1}(V)$. $P_i$ adds $\tau$ to this set by performing one Fetch&Add operation on $V$. The value retrieved in the Fetch&Add operation encodes $\mathit{Tuples}(\sigma)$, and $P_i$ can now compute $\mathit{Tuples}(\sigma, i)$. By Lemma 3.12, any $\sigma'$ such that $\mathit{Tuples}(\sigma', i) = \mathit{Tuples}(\sigma, i)$ satisfies $\sigma' \sim_i \sigma$, so $[\sigma']_{\sim_i} = [\sigma]_{\sim_i}$.

The implementation of a B-History object $O$ from one Fetch&Add object is presented in Algorithm 2. To analyze it, we introduce some notation.

**Notation** Fix $R = (\psi, S)$ to be an arbitrary run of this implementation. Since each process $P_i$ is only bound to the operation $\mathit{Sign}(i)$, $\psi(P_i)$ is actually a sequence containing only $\mathit{Sign}(i)$ operations. Procedure instances in $R$ consist of only one step, hence any procedure instance in $R$ is complete. So $\mathit{Instances}(R) = \mathit{Complete}(R)$ and $\mathit{result}(\alpha)$ is defined for any operation $\alpha \in \mathit{Instances}(R)$. Furthermore, the partial order $<_R$ on procedure instances in $R$ (defined by $\alpha <_R \alpha'$ if and only if $\alpha$ is complete and $\max(\alpha) < \min(\alpha')$) is now a total order. Let $\alpha_k$ be the $k$-th procedure instance in $\mathit{Occur}(R)$ with respect to the linear order $<_R$. Since each instance has one step, notice that for any $k$, $\min(\alpha_k) = \max(\alpha_k) = k$. Let $\alpha(i,j)$ be the $j$-th instance of the access procedure $P_i : \mathit{Sign}(i)$ in $\mathit{Occur}(R)$ with respect to $<_R$.

We define a sequence of B-History states $q_k \in [1,N]^*$ as follows: let $q_0 \overset{def}{=} \lambda$, and for $k \geq 1$, let $q_k \overset{def}{=} q_{k-1} \cdot i$, with $i$ such that $\mathit{operation}(\alpha_k) = \mathit{Sign}(i)$. Notice that $|q_k| = k$. Using this state sequence, we prove in Lemma 3.14 that the run $R$ is linearizable.

Let $\tau_\alpha$ be the value of the persistent local variable $\tau$ at the beginning of the execution of $\alpha$. Let $v_\alpha$ be the value retrieved in step 1 (line 1) of $\alpha$. Let $T_\alpha$ be the value of the local variable $T$ when line 5 is executed during $\alpha$. Let $\sigma_\alpha$ be the value of the local variable $\sigma$ computed in line 5 of $\alpha$. We prove below, in Lemma 3.13, that the process $P_i$ executing $\alpha$ will always be able to find some $\sigma_\alpha$ satisfying the condition in line 5 of $\alpha$, namely that $\mathit{Tuples}(\sigma_\alpha, i) = S_\alpha$.

Let $C_0$ be the initial configuration of the system, and for $k \geq 1$, let $C_k \overset{def}{=} C_{k-1} \cdot S[k]$, that is, the system configuration immediately after the $k$-th step occurs in $S$. For $k \geq 0$, let $V_k$ be the value of the shared Fetch&Add variable $V$ in the system configuration $C_k$. Notice that for any procedure instance $\alpha_k$ in $R$, $v_\alpha = V_{\min(\alpha_k)-1} = V_{k-1}$.

The following Lemma establishes the invariant for our algorithm.

**Lemma 3.13.** *For any $k \geq 0$, we claim that:*

- $\mathit{Tuples}(q_k) = \overline{\gamma}^{-1}(V_k)$*;*
- *if $k \geq 1$, then $\tau_{\alpha_k} = \mathit{pre\_tup}(q_k, k)$;*
- *if $k \geq 1$, then $T_{\alpha_k} = \mathit{Tuples}(q_{k-1}, i)$, where $\alpha_k = \alpha(i,j)$;*
- *if $k \geq 1$, then the computation in line 5 of $\alpha_k$ is possible, and $\sigma_{\alpha_k} \sim_i q_{k-1}$, where $\alpha_k = \alpha(i,j)$.*

*Proof.* The invariant has this form because the sequence of queue states begins at index 0, while the sequences of procedure instances begins at index 1. To prove it, we use induction on $k$.

For $k = 0$, only the first assertion is non-trivial. We have $V_0 = 0$, $q_0 = \lambda$, and

$$\mathit{Tuples}(q_0) = \mathit{Tuples}(\lambda) = \emptyset = \overline{\gamma}^{-1}(0) = \overline{\gamma}^{-1}(V_0).$$

Now let $k \geq 1$, and assume that for any $k' < k$, our invariant holds. In particular, $\mathit{Tuples}(q_{k'}) = \overline{\gamma}^{-1}(V_{k'})$. Let $i, j$ be such that $\alpha_k = \alpha(i,j)$.

**Algorithm 2.** A 1-bounded wait free, linearizable implementation of a B-History object from one Fetch&Add object.

Shared object:

$V$ is a Fetch&Add object, initialized to 0.

Process $P_i$, for $1 \leq i \leq N$:

- Persistent local variable:

  $\tau$ is an $(N+1)$-tuple, initialized to $(i, -1, \ldots, -1)$.

- Access Procedure $P_i\!:\!Sign(i)$:

```
1. (step 1) v ⟵ Fetch&Add( V, $2^{\gamma(\tau)}$ )
2.          if $\tau$ = ( i, -1, ..., -1 ) then
3.             T ⟵ $\overline{\gamma}^{-1}$( v )
            else
4.             T ⟵ $\overline{\gamma}^{-1}$( v ) ∪ { $\tau$ }
            end if
5.          find any $\sigma$ such that Tuples( $\sigma$, i ) = T
6.          for k ⟵ 1 :  N do
7.             $\tau$[ k + 1 ] ⟵ |{ t ∈ $\overline{\gamma}^{-1}$( v ) :  t[ 1 ] = k }|
            end for
8.          return $[\sigma]_{\sim_i}$
```

We first show that $\tau_{\alpha_k} = \mathit{pre\_tup}(q_k, k)$. If $j = 1$, then $\alpha_k = \alpha(i, j)$ is the first instance of the access procedure $P_i : \mathit{Sign}(i)$ in $R$, so $\tau_{\alpha_k}$ has its default value, $(i, -1, \ldots, -1)$. Furthermore, $i$ does not occur in $q_{k-1}$ and it occurs in $q_k$ at its end. So $k$ is the index of the first occurrence of $i$ in $q_k$, therefore by definition, $\mathit{pre\_tup}(q_k, k) = (i, -1, \ldots, -1) = \tau_{\alpha_k}$.

Now suppose that $j > 1$. Let $k' \geq 1$ be such that $\alpha(i, j-1) = \alpha(k')$. Since $\alpha(i, j-1) <_R \alpha(i, j)$, we have $k' < k$. Instances of the access procedure $P_i : \mathit{Sign}(i)$ are executed sequentially by $P_i$, so $P_i$ completed $\alpha(i, j-1)$ before starting $\alpha(i, j)$. The only modifications to the variable $\tau$ local to $P_i$ come from the `for` loop in lines 6 and 7 of the access procedure $P_i : \mathit{Sign}(i)$, therefore $\tau_{\alpha_k} = \tau_{\alpha(i,j)}$ is the value computed in the `for` loop during $\alpha(i, j-1)$. Specifically, we have $\tau_{\alpha(i,j)}[1] = i$, and for all $1 \leq b \leq N$,

$$\tau_{\alpha(i,j)}[b+1] = |\{t \in \overline{\gamma}^{-1}(v_{\alpha(i,j-1)}) : t[1] = b\}|$$

But

$$v_{\alpha(i,j-1)} = V_{\min(\alpha(i,j-1))-1} = V_{\min(\alpha_{k'})-1} = V_{k'-1} = \overline{\gamma}(\mathit{Tuples}(q_{k'-1}))$$

by the induction hypothesis, as $1 \leq k' < k$. So $\overline{\gamma}^{-1}(v_{\alpha(i,j-1)}) = \mathit{Tuples}(q_{k'-1})$, and $\tau_{\alpha(i,j)}[b+1]$ is equal to the number of tuples in $\mathit{Tuples}(q_{k'-1})$ whose first coordinate is $b$. By Lemma 3.9, this is exactly the number of times $b$ appears in $q_{k'-1}$. Notice that, for any $l < k$, $q_l = q_k|_{[1,l]}$, so $\tau_{\alpha(i,j)} = \mathit{tup}(q_k, k')$. But $k'$ is the index of the last occurrence of $i$ strictly before index $k$ in $q_k$, hence, by definition, $\mathit{pre\_tup}(q_k, k) = \mathit{tup}(q_k, k')$. Therefore, $\tau_{\alpha(i,j)} = \tau_{\alpha_k} = \mathit{pre\_tup}(q_k, k)$, as required.

Secondly, we want show that $\mathit{Tuples}(q_k) = \overline{\gamma}^{-1}(V_k)$. By the induction hypothesis, $\mathit{Tuples}(q_{k-1}) = \overline{\gamma}^{-1}(V_{k-1})$. The only modification in the value of the shared variable $V$ between configurations $C_{k-1}$ and $C_k$ comes from the Fetch&Add operation performed in step 1 (line 1) of $\alpha_k$. Specifically, we have $V_k = V_{k-1} + 2^{\gamma(\tau_{\alpha_k})}$. We also know that $q_k = q_{k-1} \cdot i$ and that $|q_k| = k$, so $\mathit{Tuples}(q_k) = \mathit{Tuples}(q_{k-1}) \cup \mathit{pre\_tup}(q_k, k)$. We have already seen that $\tau_{\alpha_k} = \mathit{pre\_tup}(q_k, k)$. By Lemma 3.9, $\tau_{\alpha_k} = \mathit{pre\_tup}(q_k, k) \notin \mathit{Tuples}(q_{k-1})$. Therefore,

$$V_k = V_{k-1} + 2^{\gamma(\tau_{\alpha_k})} = \overline{\gamma}(\mathit{Tuples}(q_{k-1})) + \overline{\gamma}(\{\tau_{\alpha_k}\}) = \overline{\gamma}(\mathit{Tuples}(q_k)).$$

Thirdly, we show that $T_{\alpha_k} = \mathit{Tuples}(q_{k-1}, i)$. We know that $v_{\alpha_k} = V_{k-1}$, and by the induction hypothesis, $V_{k-1} = \overline{\gamma}(\mathit{Tuples}(q_{k-1}))$. We have already showed that $\tau_{\alpha_k} = \mathit{pre\_tup}(q_k, k)$.

If $j = 1$, then $\tau_{\alpha_k} = (i, -1, \ldots, -1)$ and line 3 is executed, so $T_{\alpha_k} = \overline{\gamma}^{-1}(V_{k-1}) = \mathit{Tuples}(q_{k-1})$. In this case $i$ does not occur in $q_{k-1}$, hence

$$\mathit{Tuples}(q_{k-1}, i) = \mathit{Tuples}(q_{k-1}) = T_{\alpha_k}.$$

If $j > 1$, then $\tau_{\alpha_k} = \mathit{pre\_tup}(q_k, k)$ has non-negative coordinates, so line 4 is executed, and

$$T_{\alpha_k} = \overline{\gamma}^{-1}(v_{\alpha_k}) \cup \{\tau_{\alpha_k}\} = \overline{\gamma}^{-1}(V_{k-1}) \cup \{\tau_{\alpha_k}\} = \mathit{Tuples}(q_{k-1}) \cup \{\tau_{\alpha_k}\}$$

Let $k' \geq 1$ be such that $\alpha(i, j-1) = \alpha_{k'}$. Then $k'$ is the index of the last occurrence of $i$ in $q_k$ before $k$. So $\tau_{\alpha_k} = \mathit{pre\_tup}(q_k, k) = \mathit{tup}(q_k, k')$. Since $q_k = q_{k-1} \cdot i$ and $k' \leq k-1$, $\mathit{tup}(q_k, k') = \mathit{tup}(q_{k-1}, k')$. But $k'$ is the last occurrence of $i$ in $q_{k-1}$, so, by definition, $\mathit{Tuples}(q_{k-1}, i) = \mathit{Tuples}(q_{k-1}) \cup \{\mathit{tup}(q_{k-1}, k')\}$. Therefore,

$$T_{\alpha_k} = \mathit{Tuples}(q_{k-1}) \cup \{\tau_{\alpha_k}\} = \mathit{Tuples}(q_{k-1}) \cup \{\mathit{tup}(q_{k-1}, k')\} = \mathit{Tuples}(q_{k-1}, i)$$

Finally, let us explain why the computation in line 5 of $\alpha_k$ is possible. We know that $|q_{k-1}| \leq |\mathit{Tuples}(q_{k-1}, i)| = |T_{\alpha_k}|$, so in order to find some $\sigma_{\alpha_k}$ such that $\mathit{Tuples}(\sigma_{\alpha_k}, i) = T_{\alpha_k}$, we only need to inspect sequences over $[1, N]$ of length at most $|T_{\alpha_k}|$. Since $q_{k-1}$ fits this description, $P_i$ will always be able to find one such sequence $\sigma_{\alpha_k}$. Now $\mathit{Tuples}(\sigma_{\alpha_k}, i) = T_{\alpha_k} = \mathit{Tuples}(q_{k-1}, i)$, so by Lemma 3.12, we have $\sigma_{\alpha_k} \sim_i q_{k-1}$. ☐

Algorithm 2 is clearly 1-bounded wait-free, as any procedure execution consists of only one step, in line 1. To complete the proof of Theorem 3.8, we prove below, in Lemma 3.14, that Algorithm 2 is linearizable:

**Lemma 3.14.** *Algorithm 2 is linearizable.*

*Proof.* We have chosen $R$ to be an arbitrary run of this implementation, so it is enough to show that $R$ is linearizable. For this, we follow Definition 1.1.

Let $Occur(R) = Instances(R)$ and let the linear order on $Occur(R)$ be $<_R$, so that $\alpha_k$ is the $k$-th procedure instance in $Occur(R)$ with respect to $<_R$. Let $H$ be the object history with $(op_k, res_k) \overset{def}{=} (operation(\alpha_k), result(\alpha_k))$ for all $k \geq 1$. By definition, $H$ satisfies the first three requirements for the linearizability of $R$ in Definition 1.1, and all that is left to prove is that $H$ is legal. To show that $H$ is legal, we prove that for all $k \geq 1$, $\delta_{B-HISTORY}(q_{k-1}, op_k) = (q_k, res_k)$.

Fix $k \geq 1$ and let $i$ be such that $op_k = Sign(i)$. We know that

$$\delta_{B-HISTORY}(q_{k-1}, Sign(i)) = (q_{k-1} \cdot i, [q_{k-1}]_{\sim_i}).$$

By definition, $q_k = q_{k-1} \cdot i$. We also know that $res_k = result(\alpha_k) = [\sigma_{\alpha_k}]_{\sim_i}$. By Lemma 3.13, the computation in line 5 of $\alpha_k$ is possible, and $\sigma_{\alpha_k} \sim_i q_{k-1}$. Therefore, $res_k = [\sigma_{\alpha_k}]_{\sim_i} = [q_{k-1}]_{\sim_i}$. □

## 3.3 Implementing a BH&AR Object

**Theorem 3.15.** *For any finite or countably infinite set of values $V$, there exists a $(N+1)$-bounded wait-free linearizable implementation of a BH&AR$\langle V \rangle$ object from one B-History object and $N$ MRSW Registers.*

*Proof sketch.* A formal proof of this Theorem involves giving an implementation and proving it is wait-free and linearizable. However, we feel that this result is not complicated enough to justify that kind of formalism. Rather, we will informally explain how the implementation works.

Let us denote the shared B-History object by $BH$, and the shared Register associated with $P_i$ by $R_i$. The set of values that can be held by $R_i$ is $V^*$, which is countably infinite as long as $V$ is finite or countably infinite. The initial value of $R_i$ is $\lambda$. The process $P_i$ has a local persistent variable, $r_i$, which mirrors the value of $R_i$. The access procedure $P_i\!:\!Sign\&Write(i, x)$ works as follows. First, $P_i$ appends the value $x$ to $R_i$ by writing $r_i \cdot x$ to both $R_i$ and $r_i$. Next, $P_i$ performs a $Sign(i)$ operation on $BH$ and sets $\sigma \in \mathbb{N}^*$ to be any representative of the equivalence class retrieved as a result of that operation. Now $P_i$ performs $Read$ operations on all shared Registers $R_j$ with $j \neq i$. A new sequence $\sigma' \in (\mathbb{N} \times V)^*$ is built from $\sigma$ by attaching the $k$-th element in the sequence retrieved from $R_j$ to the $k$-th occurrence of $j$ in $\sigma$, for every $k$ and $j$. The result of the access procedure $P_i\!:\!Sign\&Write(i, x)$ is the equivalence class $[\sigma']_{\sim_i}$.

Let $R$ be any run of this implementation. Following Definition 1.1, let $Occur(R)$ be the set containing those procedure instances which perform the $Sign(i)$ operation on $BH$ in $R$, and let the linear order $\preceq$ on $Occur(R)$ be defined by $\alpha \prec \alpha'$ if and only if the step in which $\alpha$ accesses $BH$ occurs before the similar step of $\alpha'$ in the system history of the run $R$. From here on, it is easy to build a BH&AR object history which satisfies all the conditions in Definition 1.1. The essential ingredient is that, for any instance $\alpha$ of the access procedure $P_i : Sign\&Write(i, x)$, if $\alpha \in Occur(R)$, then $x$ was appended to $R_i$ before the $Sign(i)$ operation in $\alpha$. Hence, for any $\alpha'$ following $\alpha$ in $Occur(R)$, the sequence held by $R_i$ contains the value $x$ corresponding to $\alpha$ at the moment when $R_i$ is read in the final part of $\alpha'$. □

## 3.4 Implementing a Collection of Fetch&Add Objects and MRSW Registers

In this subsection, we prove:

**Theorem 3.16.** *Let $V$ be a finite or countably infinite set of values, and let $\mathcal{A}$ be a finite or countably infinite collection of Fetch&Add and MRSW Register$\langle V \rangle$ objects. There exists a 1-bounded wait-free linearizable implementation of Object$(\mathcal{A})$ from one BH&AR object.*

Let $\mathcal{A} = \{A_1, A_2, \ldots\}$, and for any $A_j \in \mathcal{A}$, let $T_j$ be the object type of $A_j$, let $q_{A_j}$ be the initial state of $A_j$ and let $b_{A_j}$ be the function giving the bindings of $A_j$. So $T_j$ is either the Fetch&Add object type, or the Register$\langle V \rangle$ object type. We know that $OP_{F\&A} = \{Fetch\&Add(x) : x \in \mathbb{Z}\}$ and $OP_{REG\langle V \rangle} = \{Read\} \cup \{Write(x) : x \in V\}$. Since both $\mathbb{Z}$ and $V$ are at most countably infinite, we derive that for any $j$, $OP_{T_j}$ is at most countably infinite. Therefore, $OP_{Type(\mathcal{A})} = \{\langle A_j, op \rangle : A_j \in \mathcal{A} \text{ and } op \in OP_{T_j}\}$ is at most countably infinite.

Let $B$ denote the shared BH&AR object used by the implementation. Its object type will be parametrized by the countably infinite set $OP_{Type(\mathcal{A})}$. For every process $P_i$ and every operation $\langle A_j, op \rangle$ that $P_i$ is allowed to apply on $Object(\mathcal{A})$, the access procedure $P_i : \langle A_j, op \rangle$ starts with its only step, which consists of applying a $Sign\&Write(i, \langle A_j, op \rangle)$ operation on $B$. Let $\sigma \in (\mathbb{N} \times OP_{Type(\mathcal{A})})^*$ be a representative of the equivalence class obtained as a result of this operation. There are three possibilities for *op*:

- $op = Write(x)$ for some $x \in V$. In this case, the access procedure simply returns the result $OK$.

- $op = Read$. If for any $y \in V$, $\langle A_j, Write(y) \rangle$ does not appear in $\sigma$ as the second coordinate of a pair in $\sigma$, the access procedure returns $q_{A_j}$, the initial state of object $A_j$. Otherwise, the access procedure returns the last value $y \in V$ to appear in a second coordinate of the form $\langle A_j, Write(y) \rangle$.

- $op = Fetch\&Add(x)$ for some $x \in \mathbb{Z}$. In this case, let $K$ be the set of indices in $\sigma$ of pairs whose second coordinate is of the form $\langle A_j, Fetch\&Add(y) \rangle$, for any $y \in \mathbb{Z}$. The access procedure computes and returns the value

$$q_{A_j} + \sum_{k \in K} \begin{cases} y & \text{if } (\sigma[k])[2] = \langle A_j, Fetch\&Add(y) \rangle, \\ 0 & \text{otherwise.} \end{cases}$$

This implementation is clearly 1-bounded wait-free, as any access procedure consists of only one step, the $Sign\&Write$ operation on $B$. The proof that this implementation is linearizable is based on two simple ideas: on the one hand, every Register is written to by at most one process and the subsequence of operations by any process is preserved under indistinguishability (by Lemma 3.6); on the other, the sequence of operations applied on a Fetch&Add object is commutative.

More formally, let $R$ be any run of this implementation. Since any access procedure consists of only one step, any procedure instance appearing in $R$ is complete. Hence, $result(\alpha)$ is defined for any $\alpha$ in $R$ and furthermore, the partial order $\leq_R$ on procedure instances is now a total order. Following Definition 1.1, let $Occur(R) = Instances(R)$ and for $k \geq 1$, let $\alpha_k$ be the $k$-th procedure instance in $Occur(R)$ with respect to the linear order $\leq_R$. Let $H$ be the $Object(\mathcal{A})$ history defined by $(op_k, res_k) \overset{def}{=} (operation(\alpha_k), result(\alpha_k))$, for all $k \geq 1$. By definition, $H$ satisfies the first three requirements for linearizability in Definition 1.1, and all that is left to prove is that $H$ is legal. We define a sequence of $Type(\mathcal{A})$ states as follows: $q_0 \overset{def}{=} q_{Object(\mathcal{A})}$ and for all $k \geq 1$, $q_k$ is the first coordinate of $\delta_{Type(\mathcal{A})}(q_{k-1}, operation(\alpha_k))$. To show that $H$ is legal (and therefore, that $R$ is linearizable) we still have to prove that for any $k \geq 1$, the second coordinate of $\delta_{Type(\mathcal{A})}(q_{k-1}, operation(\alpha_k))$ is $result(\alpha_k)$.

For $k \geq 1$, let $\sigma_{\alpha_k}$ denote the representative of the equivalence class obtained as a result to the unique step in the procedure instance $\alpha_k$. For $k \geq 0$, let $B_k \in (\mathbb{N} \times OP_{Type(\mathcal{A})})^*$ denote the state of the shared BH&AR object $B$ in the system configuration right before the occurrence of the unique step of $\alpha_{k+1}$ (so $B_0 = \lambda$, the initial state of $B$). Note that $(i, \langle A_j, op\rangle)$ appears in $B_k$ if and only if there is some $1 \leq k' \leq k$ such that $\alpha_{k'}$ is an instance of $P_i : \langle A_j, op\rangle$. By the algorithm, for any $k \geq 1$, $\sigma_{\alpha_k} \sim_i B_{k-1}$, where $P_i = process(\alpha_k)$.

Now fix $k \geq 1$ and let $process(\alpha_k) = P_i$. We distinguish three possibilities for $operation(\alpha_k)$:

- $operation(\alpha_k) = \langle A_j, Write(x)\rangle$ for some $x \in V$ and some $A_j \in \mathcal{A}$. In this case, the second coordinate of $\delta_{Type(\mathcal{A})}(q_{k-1}, operation(\alpha_k))$ is $OK$, which is the same as the result returned by the procedure instance $\alpha_k$.

- $operation(\alpha_k) = \langle A_j, Read\rangle$ for some $A_j \in \mathcal{A}$. In this case, the second coordinate of $\delta_{Type(\mathcal{A})}(q_{k-1}, operation(\alpha_k))$ is the value of the Register $A_j$ in state $q_{k-1}$.

  If no procedure instances prior to $\alpha_k$ wrote to $A_j$, the latter has its initial value, $q_{A_j}$. Furthermore, for any $y \in V$, $\langle A_j, Write(y)\rangle$ does not appear as a second coordinate in $B_{k-1}$. Since $\sigma_{\alpha_k} \sim_i B_{k-1}$, by Lemma 3.2, $\langle A_j, Write(y)\rangle$ does not appear in $\sigma_{\alpha_k}$, for any $y$. Hence, the procedure instance returns $result(\alpha_k) = q_{A_j}$, as required.

  If $A_j$ was written to in some procedure instances prior to $\alpha_k$, only one process $P_{i'}$ could have written to $A_j$ (recall that $A_j$ is a MRSW Register). The value of $A_j$ in $q_{k-1}$ is $y \in V$, where $\langle A_j, Write(y)\rangle$ is the last element of the form $\langle A_j, Write(y')\rangle$ in $SubseqVal(B_{k-1}, i')$, for any $y' \in V$. Since $\sigma_{\alpha_k} \sim_i B_{k-1}$, by Lemma 3.2, the pair $(i', \langle A_j, Write(y)\rangle)$ occurs in $\sigma_{\alpha_k}$. Let $z = result(\alpha_k)$. By definition, $z$ is the last value occurring in a second coordinate of the form $\langle A_j, Write(z')\rangle$, for any $z' \in V$. This type of second coordinate only occurs with first coordinate $i'$ in $B_{k-1}$, so by the same Lemma 3.2, it will only occur with first coordinate $i'$ in $\sigma_{\alpha_k}$. Therefore, $z$ is the last value occurring in an element of the form $\langle A_j, Write(z')\rangle$ in $SubseqVal(\sigma_{\alpha_k}, i')$, for any $z' \in V$. But now, by Lemma 3.6, $SubseqVal(\sigma_{\alpha_k}, i') = SubseqVal(B_{k-1}, i')$. Therefore $z = y$, as required.

- $operation(\alpha_k) = \langle A_j, Fetch\&Add(x)\rangle$ for some integer $x$ and some $A_j \in \mathcal{A}$. In this case, the second coordinate of $\delta_{Type(\mathcal{A})}(q_{k-1}, operation(\alpha_k))$ is the value of the Fetch&Add object $A_j$ in state $q_{k-1}$, namely $q_{A_j}$ plus the sum of those integers $y$ which occur in pairs of the form $(i', \langle A_j, Fetch\&Add(y)\rangle)$ in $B_{k-1}$, for any $i'$ (counting repetitions). The access procedure performs a similar computation using $\sigma_{\alpha_k}$ instead of $B_{k-1}$, and since $\sigma_{\alpha_k} \sim_i B_{k-1}$, Lemma 3.2 guarantees that the two sums yield the same result.

# Chapter 4

# A Problem Equivalent to Implementing a $(m, n)$-Queue

The work presented in this chapter is aimed toward proving Conjecture 1.2, namely that there is no wait-free linearizable implementation of an 3-Queue from Fetch&Add objects and Registers. We will be formulating a new problem, which will be shown to be equivalent to implementing a Queue object when every process in the system can perform either enqueue or dequeue operations. Although our objective was not achieved, we believe that the results in this chapter are interesting on their own and that they might prove useful in completing the proof of the Conjecture. Alternatively, solving this new problem would refute the Conjecture.

Using the results in the previous chapter, we have seen that a BH&AR object and collection of Fetch&Add objects and Registers are equally powerful. Therefore, there exists an implementation of a Queue from Fetch&Add objects and Registers if and only if there exists an implementation of a Queue from one BH&AR object. In this chapter, we simplify this problem in two ways. On the one hand, rather than considering a general Queue object, we will focus our attention on a restricted version of this object type, the Basic-Id-Queue. Informally, a Basic-Id-Queue object works very much like a Queue object, except that process $P_i$ is enqueuing the sequence $(i, 1), (i, 2), (i, 3), \ldots$ rather than arbitrary elements. On the other hand, we will argue that when implementing a $(m, n)$-Basic-Id-Queue from a BH&AR object, we can dispose of the Append Registers inside that object, and therefore, use only one B-History object.

This chapter is organized as follows. In section 4.1, we give the formal definition of the Basic-Id-Queue object type. In section 4.2, we show that implementing a Queue from Fetch&Add objects and Registers is equivalent to implementing a Basic-Id-Queue from Fetch&Add objects and Registers. In section 4.3, we show that for any object $O$ whose bindings allow each process in the system to apply at most one type of operation on $O$ (as will be the case with a $(m, n)$-Basic-Id-Queue object), implementing $O$ from one BH&AR object is equivalent to implementing $O$ from one B-History object. The results in these sections are summarized in the following theorem.

**Theorem 4.1.** *Let $m, n$ be two positive integers and let $V$ be a finite or countably infinite set of values. There exists a wait-free linearizable implementation of a $(m, n)$-Queue$\langle V \rangle$ object from Fetch&Add objects and Registers if and only if there exists a wait-free linearizable implementation of a $(m, n)$-Basic-Id-Queue object from one B-History object.*

By analyzing the roles played by various processes in an implementation, the following two corollaries to Theorem 4.1 should be clear.

**Corollary 4.2.** *Let $m, n$ be two positive integers and let $V$ be a finite or countably infinite set of values. If there exists an implementation of an oblivious $(m+n)$-Queue$\langle V \rangle$ object from Fetch&Add objects and Registers, then there exists an implementation of a $(m, n)$-Basic-Id-Queue object from a B-History object.*

**Corollary 4.3.** *Let $m, n$ be two positive integers and let $V$ be a finite or countably infinite set of values. If there exists an implementation of a $(m, n)$-Basic-Id-Queue object from one B-History object, then there exists an implementation of a* $\min(m, n)$*-Queue$\langle V\rangle$ object from Fetch&Add objects and Registers.*

In particular, proving that there is no (3,3)-Basic-Id-Queue implementation from a B-History object implies that there is no $n$-Queue implementation from Fetch&Add objects and Registers, for any $n \geq 6$. Alternatively, proving that there exists an implementation of a (3,3)-Basic-Id-Queue from a B-History object implies that there exists a 3-Queue implementation from Fetch&Add objects and Registers, refuting Conjecture 1.2.

In section 4.4, we provide two preliminary results, saying that there exists no (3,3)-Basic-Id-Queue implementation from a B-History object when either the enqueue procedures or the dequeue procedures are restricted to taking only one step.

## 4.1 The Basic-Id-Queue Object Type

A Basic-Id-Queue object works like a Queue object, but each Enqueue process $P_i$ is restricted to enqueuing the sequence $(i, 1), (i, 2), \ldots$. For a sequence $\sigma \in (\mathbb{N} \times \mathbb{N})^*$, let $countFirst(\sigma, i)$ denote the number of occurrences of $i$ as first coordinate in $\sigma$. The Basic-Id-Queue object type is defined as follows:

- $Q_{BIdQ} \overset{def}{=} (\mathbb{N} \times \mathbb{N})^*$;
- $OP_{BIdQ} \overset{def}{=} \{EnqId(i) : i \in \mathbb{N}\} \cup \{Deq\}$;
- $RES_{BIdQ} \overset{def}{=} (\mathbb{N} \times \mathbb{N}) \cup \{OK, \varepsilon\}$;
- for all $\sigma \in Q_{BIdQ}$ and $i \in \mathbb{N}$,
$$\delta_{BIdQ}(\sigma, EnqId(i)) \overset{def}{=} (\sigma \cdot (i, countFirst(\sigma, i) + 1), OK);$$
- $\delta_{BIdQ}(\lambda, Deq) \overset{def}{=} (\lambda, \varepsilon)$; and
- for all $\sigma \in Q_{BIdQ}$ with $\sigma = (i, j) \cdot \sigma'$, $\delta_{BIdQ}(\sigma, Deq) \overset{def}{=} (\sigma', (i, j))$.

Any Basic-Id-Queue object has the Basic-Id-Queue object type and initial state $\lambda$. For any non-negative integers $m, n, p$, a $(m, n, p)$-Basic-Id-Queue object for a system of $N = m+n+p$ processes has bindings which allow $m$ processes to perform enqueue operations, $n$ other processes to perform dequeue operations and the remaining $p$ processes to perform both kinds of operations. We write $N$-Basic-Id-Queue for a $(0, 0, N)$-Basic-Id-Queue object, which is in fact an oblivious Basic-Id-Queue object in a system of $N$ processes. We write $(m, n)$-Basic-Id-Queue for a $(m, n, 0)$-Basic-Id-Queue object.

## 4.2 Basic-Id-Queue vs. Queue

In this section we establish a relationship between these two object types.

**Theorem 4.4.** *Let $m, n, p$ be three non-negative integers. For any finite or countably infinite set of values $V$, there exists a wait-free linearizable implementation of a $(m, n, p)$-Queue$\langle V\rangle$ object from Fetch&Add objects and Registers if and only if there exists a wait-free linearizable implementation of a $(m, n, p)$-Basic-Id-Queue object from Fetch&Add objects and Registers.*

*Proof.* Let $O$ be a $(m, n, p)$-Queue$\langle \mathbb{N} \times \mathbb{N} \rangle$ object, and let $O'$ be a $(m, n, p)$-Basic-Id-Queue object with bindings similar to those of $O$: $P_i$ is allowed to perform enqueue (respectively, dequeue) operations on $O$ if and only if $P_i$ is allowed to perform enqueue (respectively, dequeue) operations on $O'$.

Suppose there exists a wait-free linearizable implementation of $O$ from Fetch&Add objects and Registers. For all processes $P_i$ and operations $op \in b_O(P_i)$, let $P_i : (O, op)$ denote the access procedure by which $P_i$ applies $op$ on $O$. The following is a wait-free linearizable implementation of $O'$ from Fetch&Add objects and Registers, using the same shared objects as the implementation of $O$:

- For all processes $P_i$ allowed to perform enqueue operations, $P_i$ has a local integer variable $c_i$ initialized to 0. The access procedure $P_i : (O', EnqId(i))$ by which $P_i$ applies $EnqId(i)$ on $O'$ consists of incrementing $c_i$ and then executing the access procedure $P_i : (O, Enq((i, c_i)))$.
- For all processes $P_j$ allowed to perform dequeue operations, the access procedure $P_j : (O', Deq)$ by which $P_j$ applies $Deq$ on $O'$ is identical to $P_j : (O, Deq)$.

This implementation is wait-free and linearizable, essentially because the enqueue procedure works as required by the Basic-Id-Queue object type, that is, process $P_i$ is enqueuing the sequence $(i, 1), (i, 2), \ldots$.

Now suppose that there exists a wait-free linearizable implementation of $O'$ from Fetch&Add objects and Registers. For all processes $P_i$ and operations $op \in b_{O'}(P_i)$, let $P_i : (O', op)$ denote the access procedure by which process $P_i$ applies operation $op$ on $O'$. We want to show that there exists a wait-free linearizable implementation of $O$ from Fetch&Add objects and Registers.

Since a MRSW Append-Register object can be easily implemented from a MRSW Register object, we will assume that we can use Append-Register objects in our implementation of $O$. To implement $O$, we use the same shared objects as in the implementation of $O'$, plus $N$ MRSW Append-Register$\langle V \rangle$ objects, $R_1, \ldots R_N$, where for all $i$, $R_i$ is written by process $P_i$:

- For all processes $P_i$ allowed to perform enqueue operations and all $x \in V$, the access procedure $P_i : (O, Enq(x))$ by which $P_i$ applies $Enq(x)$ on $O$ consists of two parts. First, $P_i$ atomically appends the value $x$ to $R_i$. Afterward, $P_i$ executes the access procedure $P_i : (O', EnqId(i))$, which may not be atomic, and returns $OK$.
- For all processes $P_j$ allowed to perform dequeue operations, the access procedure $P_j : (O, Deq)$ by which $P_j$ applies $Deq$ on $O$ also consists of two parts. First, $P_j$ executes the access procedure $P_j : (O', Deq)$ and saves the result of that operation in some local variable $res$. If $res$ is $\varepsilon$, the access procedure itself returns $\varepsilon$. Otherwise, $res = (j', k)$ for some process id $j'$ and some positive integer $k$. In this case, the access procedure atomically reads the shared Append-Register $R_{j'}$. It then returns the $k$-th element in the sequence stored in that object.

The reason this implementation is wait-free and linearizable is that by the time some dequeue instance obtains the value $(j', k)$ from its $Deq$ operation on $O'$, the process $P_{j'}$ has already written the $k$-th enqueued value in the shared Append-Register object $R_{j'}$. ☐

## 4.3 BH&AR Object vs. B-History Object

Throughout this section, let $V$ be a finite or countably infinite set of values, let $B$ be a BH&AR$\langle V \rangle$ object and let $B'$ be a B-History object. In this section, we prove that whenever a shared object $O$ satisfies certain conditions, $O$ can be implemented from $B$ if and only if $O$ can be implemented from $B'$. The easier direction of this implication is captured by the following Lemma.

**Lemma 4.5.** *Let $O$ be any shared object. If there exists a wait-free linearizable implementation of $O$ from one B-History object, then there exists a wait-free linearizable implementation of $O$ from one BH&AR object.*

*Proof.* Observe that by Theorem 3.16, Theorem 3.8 and transitivity of implementation, a B-History object can be implemented from a BH&AR object. Therefore, if $O$ can be implemented from a B-History object, it can also be implemented from a BH&AR object. □

In the reminder of this section, we prove a converse of this fact for a class of objects which includes the $(m, n)$-Basic-Id-Queue. Specifically, this is the class of objects which allow each process to apply only one type of operation, and which further satisfy the non-triviality condition that every access procedure execution contains at least one step.

**Theorem 4.6.** *Let $O$ be a shared object which satisfies the following two conditions:*

*(i) for all processes $P_i$ in the system, $|b_O(P_i)| = 1$; and*

*(ii) for any implementation of $O$ from any collection of shared objects $\mathcal{B}$, any execution of any access procedure on $O$ contains at least one step.*

*If $O$ can be implemented from one BH&AR object, then $O$ can be implemented from a B-History object.*

Notice that, in particular, a $(m, n)$-Basic-Id-Queue object satisfies this conditions, in any non-trivial case when $m \geq 1$ and $n \geq 1$. We prove this theorem in a constructive manner: we begin by assuming that a wait-free linearizable implementation of $O$ from $B$ exists, we construct an implementation of $O$ from $B'$, and we eventually show that the latter implementation is wait-free and linearizable.

The same result can be obtained even if we were to relax the restrictions imposed on $O$ and only require that $|b_O(P_i)| \leq 1$ for all processes $P_i$. However, the extra formalism involved in proving this slightly more general result is not necessary for the rest of this thesis. The main motivation behind conditions (i) and (ii) is that, if $O$ satisfies both, and if an implementation of $O$ from $B$ exists, then at any point in a run of this implementation, any process can be scheduled to take a step. More formally, any sequence $\gamma \in [1, N]^*$ is a possible schedule of processes in a run $R$ of this implementation.

So let us assume that, for all processes $P_i$ and operations $op \in b_O(P_i)$, the access procedures $P_i :_B\colon op$ specify a wait-free linearizable implementation of $O$ from $B$. Recall that, since $B$ is a BH&AR object, $P_i$ can only apply to $B$ operations of the form $Sign\&Write(i, x)$, for all $x \in V$. Hence, the execution of the access procedure $P_i :_B\colon op$ consists, apart from local computation, of a number of steps and, in each of those, $P_i$ applies one $Sign\&Write(i, x)$ operation to $B$ and obtains a result of the form $[\sigma]_{\sim_i}$, the class of sequences indistinguishable from $\sigma \in ([1, N] \times V)^*$ with respect to $i$. When implementing $O$ from the B-History object $B'$, the only operation that $P_i$ is allowed to apply to $B'$ is $Sign(i)$, and the result of this operation is of the form $[\gamma]_{\approx_i}$, the class of sequences indistinguishable from $\gamma \in [1, N]^*$ with respect to $i$ (in this section we use $\sim_i$ for indistinguishability on sequences of pairs and $\approx_i$ for indistinguishability on sequences of integers).

Before we get into the formalism required to prove this result, here are the main steps of our proof. In order to show that there exists a wait-free linearizable implementation of $O$ from $B'$, we will first define a translation function $\rho$ which, given a state of $B'$ in the sequence $\gamma \in [1, N]^*$, computes a state of $B$ in $\sigma = \rho(\gamma) \in ([1, N] \times V)^*$. Secondly, we will show how to extend $\rho$ to be defined on subsets of $[1, N]^*$ in such a way that $\rho$ preserves equivalence classes of the indistinguishability relation. Thirdly, to implement $O$ from $B'$, we will define access procedures of the form $P_i :_{B'}\colon op$. Having obtained an implementation of $O$ from $B'$, we will finally prove that it is wait-free and linearizable by showing that, for any run $R'$ of this implementation, there exists a run $R$ of the implementation of $O$ from $B$, such that every process $P_i$ behaves exactly the same in $R$ and $R'$. Our conclusion follows from the fact that the latter implementation is, by assumption, wait-free and linearizable.

**Definition of the Translation Function $\rho$** Let $\gamma$ be any sequence in $[1, N]^*$. Since $O$ satisfies conditions (i) and (ii), there exists a run $R$ of the implementation of $O$ from $B$ in which processes are scheduled to take steps in the same order as their indices appear in $\gamma$. Since all access procedures are deterministic, in any run, all processes start from their initial states, and any process $P_i$ has exactly one access procedure available to execute on $O$, the run $R$ is unique. We define $\rho(\gamma) \in ([1, N] \times V)^*$ to be the state of the shared object $B$ at the end of this run $R$. Notice that, for any occurrence of $i$ in $\gamma$, $P_i$ takes a step of the form $Sign\&Write(i, x)$. This has the effect of adding the pair $(i, x)$ to the state of $B$. Therefore, $|\rho(\gamma)| = |\gamma|$, $\rho(prefix(\gamma, k)) = prefix(\rho(\gamma), k)$ for all $k$, and $\gamma$ is the sequence of first coordinates in $\rho(\gamma)$.

**Defining $\rho$ on Subsets** We now want to extend the definition of $\rho$ to subsets of $[1, N]^*$ in such a way that equivalence classes of the indistinguishability relation are preserved. We accomplish this is as follows: for all positive integers $i$ and every set $A \subset [1, N]^*$, we define $\rho_i(A) = \cup_{\gamma \in A}[\rho(\gamma)]_{\sim_i}$. We prove below that $\rho(\gamma) \sim_i \rho(\gamma')$ whenever $\gamma \approx_i \gamma'$, implying that $\rho_i([\gamma]_{\approx_i}) = [\rho(\gamma)]_{\sim_i}$.

Informally, the main idea behind Lemma 4.7 is the following. In the implementation of $O$ from $B'$, whenever $B'$ is in state $\gamma$ and $P_i$ applies $Sign(i)$ to $B'$, $P_i$ can compute the second argument to every step by every process in run $R$, where $R$ is the run of the implementation of $O$ from $B$ in which processes are scheduled as in $\gamma$. The reason is that the second argument of the $l$-th step by $P_{i'}$ in $R$ is completely determined by the results $P_{i'}$ got in $R$ to its first $l - 1$ steps, and by Lemma 3.7, $P_i$ can compute those results.

**Lemma 4.7.** *Let $\gamma, \gamma' \in [1, N]^*$ and let $i \in [1, N]$. If $\gamma \sim_i \gamma'$, then $\rho(\gamma) \sim_i \rho(\gamma')$.*

*Proof.* Let $\sigma = \rho(\gamma)$ and $\sigma' = \rho(\gamma')$. Clearly, $|\sigma| = |\gamma| = |\gamma'| = |\sigma'|$. Let $R$ and $R'$ be the two runs of the implementation of $O$ from the BH&AR object $B$ in which processes are scheduled in the order their indices appear in $\gamma$ and $\gamma'$, respectively.

Since $\gamma$ is the projection of $\sigma$ on the first coordinate and since $last(\sigma, i)$ is only defined as function of the first coordinates in $\sigma$, we get that $last(\sigma, i) = last(\gamma, i)$. We know from the definition of $\gamma \sim_i \gamma'$ that for all $A \in MaxConsec(last(\gamma, i))$, there exists a permutation $\pi_A$ such that $\gamma'|_A = \pi_A(\gamma|_A)$. Let $\sigma'' \in ([1, N] \times V)^*$ be the unique sequence of the same length as $\sigma$, satisfying:

- for any $k \notin last(\sigma, i)$, $\sigma''[k] = \sigma[k]$; and
- for any $A \in MaxConsec(last(\sigma, i)) = MaxConsec(last(\gamma, i))$, $\sigma''|_A = \pi_A(\sigma|_A)$.

By construction, $\sigma'' \sim_i \sigma$ and $\gamma'$ is the projection of $\sigma''$ on the first coordinate. In order to prove that $\sigma' \sim_i \sigma$, we inductively show that $\sigma'$ and $\sigma''$ have the same prefixes and are, therefore, equal.

Trivially, the zero-length prefixes of both $\sigma'$ and $\sigma''$ are $\lambda$. Now let $0 \leq k < |\sigma|$ and assume that $prefix(\sigma', k) = prefix(\sigma'', k)$. In order to show that the prefixes of length $k + 1$ are the same, it is enough to show that $\sigma'[k + 1] = \sigma''[k + 1]$. Since $\gamma'$ is the projection on the first coordinate of both $\sigma'$ and $\sigma''$, we get that $(\sigma'[k+1])[1] = (\sigma''[k+1])[1]$. Let $\sigma'[k+1] = (a, x)$ and let $\sigma''[k+1] = (a, y)$. To complete the proof of the Lemma, all that is left to show is that $x = y$. We consider two cases.

First, suppose that $k + 1$ is the index of the first occurrence of $a$ as first coordinate in $\sigma'$. So the first step by process $P_a$ in run $R'$ is $Sign\&Write(a, x)$. By the inductive hypothesis, $k + 1$ is also the index of the first occurrence of $a$ as first coordinate in $\sigma''$. Since $\sigma'' \sim_i \sigma$, by Lemma 3.6, $SubseqVal(\sigma, a) = SubseqVal(\sigma'', a)$, so $(a, y)$ is the first pair having first coordinate $a$ in $\sigma$, therefore the first step taken by process $P_a$ in $R$ is $Sign\&Write(a, y)$. All access procedures are deterministic, all processes start from the same state in $R$ and $R'$ and each has only one access procedure available to execute, so the first step by $P_a$ has to be the same in both runs. Hence, $x = y$.

Now suppose that $a$ occurs as first coordinate in $\sigma'$ and $\sigma''$ at $b \geq 1$ indices $j_1, \ldots, j_b$ before index $k+1$. Let $J = \{j_1, \ldots, j_b\}$. The $b$ results obtained by $P_a$ in run $R'$ to its first $b$ steps are $[prefix(\sigma', j_1 - 1)]_{\sim_i}, \ldots, [prefix(\sigma', j_b - 1)]_{\sim_i}$. By definition, $a$ appears as first coordinate at index $k+1$ in $\sigma''$, hence $J \cap last(\sigma'', i) = \emptyset$. Since $\sigma'' \sim_i \sigma$, by Lemma 3.4, $last(\sigma, i) = last(\sigma'', i)$, so $J \cap last(\sigma, i) = \emptyset$ and $J$ also contains the indices of the first $b$ occurrences of $a$ as first coordinate in $\sigma$. So the $b$ results obtained by $P_a$ in $R$ to its first $b$ steps are $[prefix(\sigma, j_1 - 1)]_{\sim_i}, \ldots, [prefix(\sigma, j_b - 1)]_{\sim_i}$. Now for any $j \in J$, since $j \notin last(\sigma, i)$, by Lemma 3.7, $prefix(\sigma, j-1) \sim_i prefix(\sigma'', j-1)$. By induction hypothesis, $prefix(\sigma'', j-1) = prefix(\sigma', j-1)$. Therefore, all $b$ results obtained by $P_a$ to its first $b$ steps in $R$ and $R'$ are the same. All access procedures are deterministic, all processes start from the same state in $R$ and $R'$, each has only one access procedure available to execute, and $P_a$ obtains the same results to its first $b$ steps in both runs, so the $(b+1)$-st step by $P_a$ has to be the same in $R$ and $R'$, hence $x = y$. □

**Access Procedures for the New Implementation** We are now ready to define an implementation of $O$ from $B'$. For all processes $P_i$ and operations $op \in b_O(P_i)$, the access procedure $P_i\!:_{B'}\!:\!op$ is syntactically identical to $P_i\!:_B\!:\!op$, except that each step of the form

$$res \longleftarrow Sign\&Write(i, x)$$

is replaced by

$$res \longleftarrow \rho_i(Sign(i)).$$

**Linearizability and Wait-Freedom** Let $R'$ be any run of the implementation of $O$ from $B'$ that we have just defined. Let $\gamma$ be the state of the shared B-History object $B'$ at the end of $R'$. Let $\sigma = \rho(\gamma)$ and let $R$ be the run of the implementation of $O$ from $B$ in which processes are scheduled in the same order as in $R'$ (that is, in the order their indices appear in $\gamma$). By definition, $\sigma$ is the state of $B$ at the end of $R$.

We now claim that all procedure instances are identical in $R$ and $R'$. Furthermore, either they are complete in both runs or in neither run, and if they are complete, they return the same values in both runs. Formally, we are claiming that $Instances(R) = Instances(R')$, $Complete(R) = Complete(R')$ and for all $\alpha \in Complete(R)$, $result(\alpha)$ is the same in both runs.

To see this, inductively assume that all processes behaved the same in $R$ and $R'$ in steps up to and including the $k$-th. In the $(k+1)$-st step of $R'$, some process $P_i$ applies $Sign(i)$ on $B'$ and receives the result $[prefix(\gamma, k)]_{\approx_i}$. It then applies $\rho_i$ to this result, obtaining

$$\rho_i([prefix(\gamma, k)]_{\approx_i}) = [\rho(prefix(\gamma, k))]_{\sim_i} = [prefix(\sigma, k)]_{\sim_i},$$

which is exactly the result of the $(k+1)$-st step in $R$. So by induction, all processes behave exactly the same in $R$ and $R'$. Since the linearizability and wait-freedom conditions are defined in terms of the procedure instances in runs of an implementation, and since the implementation of $O$ from $B$ is by assumption linearizable and wait-free, the implementation of $O$ from $B'$ is also linearizable and wait-free.

## 4.4 Preliminary Negative Results

In the following two subsections we present some preliminary negative results regarding the possibility of implementing a (3,3)-Basic-Id-Queue from one B-History object. In subsection 1, we prove that there is no such implementation when every dequeue access procedure is restricted to taking only one step. In subsection 2, we prove that there is no such implementation when every enqueue access procedure is restricted to taking only one step.

Before we present any results, let us make an observation. Since a $(m, n)$-Basic-Id-Queue object $O$ satisfies conditions (i) and (ii) from Lemma 4.6, all access procedures are deterministic and process start from the same initial state in any run, there is a unique run of the implementation of $O$ from a B-History object $BH$ in which processes take steps in the same order their indices appear in $\sigma$. In the following subsections, we will denote this run by $run(\sigma)$. Since each time $P_i$ takes a step in this implementation, it applies the operation $Sign(i)$ on $BH$, it should be clear that the state of $BH$ at the end of $run(\sigma)$ is $\sigma$ and, furthermore, the system history $S$ of $run(\sigma)$ is of the form $S[k] = (P_i, BH, Sign(i), [prefix(\sigma, k-1)]_{\sim_i})$ for all $k$.

## 4.4.1 No Implementation when Dequeue Takes Only One Step

In this subsection we prove:

**Theorem 4.8.** *There is no wait-free linearizable implementation of a (1, 2)-Basic-Id-Queue object from one B-History object in which all dequeue access procedures are restricted to taking only one step.*

Let $O$ be a (1, 2)-Basic-Id-Queue object. Let us assume, without loss of generality, that process $P_1$ is the enqueuer and processes $P_2$ and $P_3$ are the two dequeuers. Let $BH$ be a B-History object. To prove the theorem, we assume that there exists such a restricted implementation of $O$ from $BH$ and eventually derive a contradiction. Specifically, we build an infinite family of sequences $\sigma_1, \sigma_2, \ldots$ in $\{1, 2, 3\}^*$, such that the associated infinite family of runs $run(\sigma_1), run(\sigma_2), \ldots$, together with the enqueue process, satisfy the conditions in Proposition 1.1, contradicting the assumption that the implementation is wait-free. For all $k$, let $d_k = 2$ when $k$ is odd, and $d_k = 3$ when $k$ is even. In general, the sequence $\sigma_k$ will be of the form

$$\sigma_k = (1^{j_1} \cdot 2 \cdot 1) \cdot (1^{j_2} \cdot 3 \cdot 1) \cdot \ldots (1^{j_k} \cdot d_k \cdot 1)$$

for some non-negative integers $j_1, \ldots, j_k$.

**Lemma 4.9.** *Let $\sigma \in 1, 2, 3^*$ and let $d \in \{2, 3\}$. If every dequeue instance in $run(\sigma \cdot d)$ consists of only one step and outputs $\varepsilon$, then no enqueue instance is complete in $run(\sigma)$.*

*Proof.* Assume that there were some complete enqueue instance $\pi$ in $run(\sigma)$ and that all dequeue instances in $run(\sigma \cdot d)$ output $\varepsilon$. The implementation is linearizable and a queue cannot be empty immediately after an enqueue operation. Hence, if there exists a dequeue instance $\phi$ immediately following $\pi$ in the linear order on procedure instances, $\phi$ must output a value other than $\varepsilon$. Since the dequeue instance corresponding to the last step by $P_d$ in $run(\sigma \cdot d)$ begins after $\pi$ is completed, we derive that there indeed exists a dequeue instance $\phi$ immediately following $\pi$. However, by assumption, $\phi$ outputs $\varepsilon$. This is a contradiction. □

We inductively build the family $\sigma_0, \sigma_1, \ldots$ of sequences in $[1, 3]^*$ satisfying the following predicate $I(k)$, for all $k \geq 0$:

- any dequeue instance in $run(\sigma_k \cdot d_{k+1})$ outputs $\varepsilon$ (notice that any dequeue instance is complete, as it contains only one step); and
- if $k \geq 1$, then $\sigma_{k-1}$ is a prefix of $\sigma_k$, and $\sigma_k$ contains at least one more occurrence of 1 than $\sigma_{k-1}$.

We define $\sigma_0 = \lambda$. In this case, $I(0)$ is trivially true: in $run(\sigma_0 \cdot d_1) = run(2)$, the only dequeue instance, by $P_2$, has to output $\epsilon$ because the queue starts from an empty state.

Now fix $k \geq 0$ and assume that we have built $\sigma_k$ in such a way that $I(k)$ holds. We build $\sigma_{k+1}$ such that $I(k+1)$ holds.

Consider the following experiment. Starting from the configuration at the end of $run(\sigma_k)$, we let $P_1$ takes steps over and over again, until $P_1$ completes some enqueue procedure. By the first condition in $I(k)$ and Lemma 4.9, there is no complete enqueue instance in $run(\sigma_k \cdot d_{k+1})$. Notice that $P_1$ might have an incomplete enqueue procedure in $run(\sigma_k)$, and in that case we let it take steps until it completes that access procedure. By wait-freedom, there exists some $max\_z \geq 1$ such that $run(\sigma_k \cdot 1^{max\_z})$ contains only one enqueue instance, and this instance is complete. By $I(k)$, no dequeue instance in $run(\sigma_k \cdot d_{k+1})$ outputs anything other than $\varepsilon$. The same will be true for $run(\sigma_k \cdot 1^{max\_z})$, as the latter contains the same dequeue instances as the former, except for the last dequeue instance by $P_{d_{k+1}}$. Then, by linearizability of our implementation, if $P_{d_{k+1}}$ is scheduled to take a step in the configuration at the end of $run(\sigma_k \cdot 1^{max\_z})$, $P_{d_{k+1}}$ has to output $(1, 1)$, the first element in the queue. Therefore, there exists $z$, $0 \leq z \leq max\_z$, such that the last dequeue instance by $P_{d_{k+1}}$ in $run(\sigma_k \cdot 1^z \cdot d_{k+1})$ outputs a value other than $\varepsilon$. Let $min\_z$ be the minimum such $z$. By the first part of $I(k)$, all dequeue instances in $run(\sigma_k \cdot d_{k+1})$ output $\varepsilon$, therefore $1 \leq min\_z \leq max\_z$.

We define $\sigma_{k+1} = \sigma_k \cdot 1^{min\_z-1} \cdot d_{k+1} \cdot 1$. Notice that, by construction, $\sigma_{k+1}$ satisfies the second part of $I(k+1)$. To complete the proof of $I(k+1)$, it is enough to prove its first part.

On the one hand, $run(\sigma_k)$ contains the same dequeue instances as $run(\sigma_k \cdot d_{k+1})$ except for the last one, so they all output $\varepsilon$. Furthermore, $run(\sigma_{k+1}) = run(\sigma_k \cdot 1^{min\_z-1} \cdot d_{k+1} \cdot 1)$ contains the same dequeue instances as $run(\sigma_k)$ plus one more instance, the one by $P_{d_{k+1}}$ which takes its step second to last. By definition of $min\_z$, that dequeue instance outputs $\varepsilon$, hence all dequeue instances in $run(\sigma_{k+1})$ output $\varepsilon$.

On the other hand, by the construction of $min\_z$, $run(\sigma_k \cdot 1^{min\_z} \cdot d_{k+1})$ contains at most one enqueue instance, and the last dequeue instance $\phi$ by $P_{d_{k+1}}$ outputs a value other than $\varepsilon$. By the first part of $I(k)$, all other dequeue instances output $\varepsilon$, so $\phi$ must output the first value available in the queue, namely (1,1). Therefore, if $P_{d_{k+2}}$ is scheduled to take a step after that run, it has to output $\varepsilon$. So in $run(\sigma_k \cdot 1^{min\_z} \cdot d_{k+1} \cdot d_{k+2})$, the last dequeue instance by $P_{d_{k+2}}$ outputs $\varepsilon$. Notice that, in the run above, the result obtained by $P_{d_{k+2}}$ to its only step in the last dequeue procedure is $[\sigma_k \cdot 1^{min\_z} \cdot d_{k+1}]_{\sim_{d_{k+2}}}$. Furthermore, in $run(\sigma_k \cdot 1^{min\_z-1} \cdot d_{k+1} \cdot 1 \cdot d_{k+2}) = run(\sigma_{k+1} \cdot d_{k+2})$, the result obtained by $P_{d_{k+2}}$ to its only step in the last dequeue procedure is $[\sigma_{k+1}]_{\sim_{d_{k+2}}}$. But now, $d_{k+2} \notin \{1, d_{k+1}\}$, so

$$\sigma_k \cdot 1^{min\_z} \cdot d_{k+1} \sim_{d_{k+2}} \sigma_k \cdot 1^{min\_z-1} \cdot d_{k+1} \cdot 1 = \sigma_{k+1}.$$

Since $P_{d_{k+2}}$ obtains the same results to all its steps in both of these runs, it must output the same value, hence the last dequeue instance by $P_{d_{k+2}}$ in $run(\sigma_{k+1} \cdot d_{k+2})$ outputs $\varepsilon$. This completes the proof that $I(k+1)$ holds.

Finally, we claim that $P_1$ together with the infinite family $run(\sigma_1), run(\sigma_2), \ldots$ satisfy the conditions in Lemma 1.1. Notice that we do not consider $\sigma_0$, because there are no incomplete enqueue instances in $run(\sigma_0)$.

Let $run(\sigma_k) = (\psi_k, S_k)$. By the second part of the invariant, $\sigma_k$ is a prefix of $\sigma_{k+1}$, hence $S_k$ is a prefix of $S_{k+1}$. In general, whenever $S_k$ is a prefix of $S_{k+1}$ and every process is only bound to one operation of the implemented object, as is the case with a Basic-Id-Queue object, $\psi_k(P_i)$ is a prefix of $\psi_{k+1}(P_i)$ for any process $P_i$, essentially because there is no choice in assigning the steps of $P_i$ to access procedures. Furthermore, $\sigma_{k+1}$ contains at least one more occurrence of 1 than $\sigma_k$, so $S_{k+1}$ contains at least one more step by $P_1$ than $S_k$. This completes the proof that the first two conditions in Lemma 1.1 are satisfied. For the third and last one, we claim that $\psi_k(P_1) = (EnqId(1))$ for all $k \geq 1$. By the first part of $I(k)$ and Lemma 4.9, no enqueue instance is complete in $\sigma_k$. Since $P_1$ is taking steps in $run(\sigma_k)$ and only the last procedure instance by a process in a run can be incomplete, all the steps of $P_1$ in $run(\sigma_k)$ are part of the execution of its first access procedure. Thus $\psi_k(P_1) = (EnqId(1))$.

This shows that the implementation of $O$ from $BH$ is not wait-free, which is a contradiction. Hence no such implementation exists.

The following is an obvious Corollary to Theorem 4.8:

**Corollary 4.10.** *There is no wait-free linearizable implementation of a 3-Basic-Id-Queue from one B-History object in which all dequeue access procedures are restricted to taking only one step.*

### 4.4.2 No Implementation when Enqueue Takes Only One Step

In this subsection we prove the following:

**Theorem 4.11.** *There is no wait-free linearizable implementation of a* $(2, 1)$*-Basic-Id-Queue object from one B-History object in which all enqueue access procedures are restricted to taking only one step.*

Let $O$ be a (2, 1)-Basic-Id-Queue object. Without loss of generality, let $P_1$ and $P_2$ be the enqueue processes and $P_3$ be the dequeue process. Let $BH$ be a B-History object. To prove the theorem, we assume that there exists a wait-free linearizable implementation of $O$ from $BH$ and we eventually derive a contradiction.

Consider a run in which $P_1$ and $P_2$ each take a step, each of them completing an enqueue procedure. We now schedule $P_3$ to take steps until it completes its first dequeue procedure. By wait-freedom, there exists some $z \geq 1$, such that $run(1 \cdot 2 \cdot 3^z)$ contains only one dequeue instance and that instance is complete. Since the enqueue instance by $P_1$ is completed before the one by $P_2$ begins, the first element enqueued will be (1, 1). The dequeue instance begins after both enqueue instances are completed, so it must output the first element available in the queue, namely (1, 1). However, since $3 \notin \{1, 2\}$, $1 \cdot 2 \cdot 3^z \sim_3 2 \cdot 1 \cdot 3^z$. But then, all the results that $P_3$ gets to its steps in $run(2 \cdot 1 \cdot 3^z)$ are the same as in $run(1 \cdot 2 \cdot 3^z)$. Hence $P_3$ has to behave the same in both of these runs, so in $run(2 \cdot 1 \cdot 3^z)$, there will be only one dequeue instance, it will be complete, and it will output (1, 1). However, in the latter run, the first element enqueued is (2, 1) and the first element dequeued is (1, 1). This contradicts the linearizability of our implementation, therefore no such implementation exists.

The following is an obvious Corollary to Theorem 4.11:

**Corollary 4.12.** *There is no wait-free linearizable implementation of a 3-Basic-Id-Queue from one B-History object in which all enqueue access procedures are restricted to taking only one step.*

# Chapter 5

# Conclusions

The main result in this thesis is a wait-free linearizable implementation of a $(1, n)$-Queue object from Common2 objects and Registers, for any $n$. In [Li01], Li provided an $(m, 2)$-Queue implementation from Common2 objects and Registers, for any $m$. The existence of a wait-free linearizable $(m, n)$-Queue implementation from Common2 objects and Registers remains open whenever $m \geq 2$ and $n \geq 3$. Attempts by both Li and the author to implement a 3-Queue$\langle V \rangle$ object from Common2 objects and Registers have failed, so this problem also remains open, even when $V$ contains only one element (in this case, the Queue is just a non-negative integer counter supporting Increment and Decrement-If-Positive operations). Li conjectured in [Li01] the impossibility of such an implementation, a conjecture which we reiterate in this thesis, in Conjecture 1.2.

The work presented in Chapters 3 and 4 of this thesis is aimed at proving Conjecture 1.2. Although our objective was not achieved, we believe that the results obtained are interesting on their own and that they might prove useful in completing the proof of the Conjecture.

In Chapter 3, we develop a new shared object type, BH&AR, which has two important characteristics. On the one hand, only one BH&AR object can simulate countably infinitely many Common2 objects and Registers. On the other hand, we can precisely describe all the information that a process gets about other processes as a result of applying an operation to a BH&AR object. These properties might facilitate the development of an adversarial argument for proving the impossibility of an $N$-Queue implementation for some $N$.

In Chapter 4, we study implementations of non-oblivious Queue objects and show that implementing an $(m, n)$-Queue object from Common2 objects and Registers is equivalent to implementing an $(m, n)$-Basic-Id-Queue object from one B-History object, a simplified version of a BH&AR object. Furthermore, we show that in any implementation of a (3, 3)-Basic-Id-Queue object from a B-History object, neither enqueue procedures nor dequeue procedures can be restricted to one step. In general, the existence of a (3, 3)-Basic-Id-Queue object from one B-History object remains open, even when all access procedures are restricted to two steps.

In [Li01], Li obtains an $(m, 2)$-Queue implementation by modifying an $(m, 1)$-Queue implementation using ideas from Herlihy's universal construction in [Her91]. Specifically, he develops a mechanism by which the two dequeue processes in his implementation agree on a total order on the dequeue operations to be performed, and subsequently perform those operations much like they would in the original $(m, 1)$-Queue implementation. We have attempted to apply a similar mechanism in order to obtain a $(2, n)$-Queue implementation from our $(1, n)$-Queue implementation, but without success. Informally, the problem appears to lie with the interaction between the enqueue and dequeue processes: in the $(m, 1)$-Queue implementation considered in [Li01], the communication between enqueue processes and dequeue processes is achieved exclusively through Register objects; while in our $(1, n)$-Queue implementation in Chapter 2, this communication is achieved through both the Register object $ROW$, and the Swap objects in the array $ITEMS$. Li obtains a $(m, 2)$-Queue implementation by (i) having the two dequeue processes agree on a total order on the dequeue operations; (ii) having each dequeue process execute the steps of each of the dequeue operations, including those initiated by the other dequeue process; and (iii) having the two dequeue processes agree on the result of each dequeue operation. When trying to extend our $(1, n)$-Queue implementation to a $(2, n)$-Queue implementation, part (iii) is irrelevant (for all enqueue operations produce the same result, $OK$) and part (i) can be achieved by having the two enqueue processes agree on a sequence of enqueue operations. The problem lies with part (ii), specifically with the fact that we were not able to synchronize two processes both executing the same steps to (together) perform a single enqueue operation. In Li's extended implementation, each of the Register objects used for communication between enqueue and dequeue processes only influences the steps taken by the two dequeue processes. If we were to apply the same method to our implementation, accesses to the shared Swap objects in the array $ITEMS$ would influence not only the steps of the two enqueue processes, but also of one dequeue process. For example, consider the situation in which enqueue processes $E_1$ and $E_2$ are working together to perform some enqueue operation. Suppose $E_1$ first applies its Swap operation to the cell $ITEMS[r, c]$, and now the $E_2$ applies its own Swap operation to the same cell. At that moment, $E_2$ cannot tell if some dequeue process accessed that cell before $E_1$, so $E_2$ cannot tell if $E_1$ has to jump to the next row of $ITEMS$ or not. This is merely an informal argument of why Li's method cannot be straightforwardly applied. In general, the existence of a $(2, n)$-Queue implementation from Common2 objects and Registers remains open when $n \geq 3$.

# Bibliography

[AWW93] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 159–170, 1993.

[FR03] Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2–3):121–163, 2003.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HW90] Maurice Herlihy and Jeanette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):495–504, January 1990.

[Jay93] Prasad Jayanti. On the robustness of Herlihy's hierarchy. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 145–158, 1993.

[Jay95] Prasad Jayanti. Wait-free computing. In *Proceedings of the 9th Workshop on Distributed Algorithms*, 1995.

[JT92] Prasad Jayanti and Sam Toueg. Some results on the impossibility, universality and decidability of consensus. In *Proceedings of the 6th Workshop on Distributed Algorithms*, 1992.

[Li01] Zongpeng Li. Non-blocking implementation of queues in asynchronous distributed shared-memory systems. Master's thesis, University of Toronto, 2001.

[Lyn96] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.

[McC94] Scott D. McCrickard. A study of wait-free hierarchies in concurrent systems. Technical Report GIT-CC-94-04, Georgia Institute of Technology, 1994.

# Index

$(m, n)$-Queue, 12
$(m, n, p)$-Queue, 12
$\lambda$, 5
$\leq_R$, 8
$run(\sigma)$, 49

access procedure, 7

B-History type, 29
Basic-Id-Queue type, 44
Basic2, 4
BH&AR type, 29
binding function, 5

Common2, 3
complete procedure instance, 8
configuration, 6
consensus number, 2
Consensus problem, 1

implementation, 7
initial configuration, 6

jump enqueue instance, 17

legal object history, 7
legal step, 7
legal system history, 7
linearizability, 1, 8

non-blocking, 4

object, 5
object history, 7
object type, 5
oblivious, 2
oblivious object, 5
occurrence point, 21

procedure instance, 8
process, 6

read-modify-write, 3
regular enqueue instance, 17
run, 7

strongly connected, 2
system history, 7

type I dequeue instance, 19
type II dequeue instance, 20
type III dequeue instance, 20

universal type, 2

wait-freedom, 1, 9