

# **GCList: Garbage Collection in Concurrent Sets**

Jonathan Marbaniang

A Thesis Submitted to  
Indian Institute of Technology Hyderabad  
In Partial Fulfillment of the Requirements for  
The Degree of Master of Technology

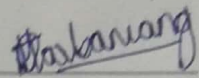


Department of Computer Science & Engineering

June 2018

## Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.



(Signature)

---

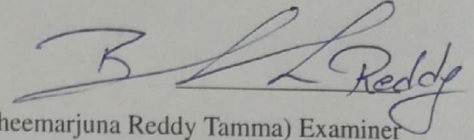
(Jonathan Marbaniang)

CS16MTECH11006

(Roll No.)

## Approval Sheet

This Thesis entitled GCList: Garbage Collection in Concurrent Sets by Jonathan Marbaniang is approved for the degree of Master of Technology from IIT Hyderabad



(Dr. Bheemarjuna Reddy Tamma) Examiner

Dept. of CSE

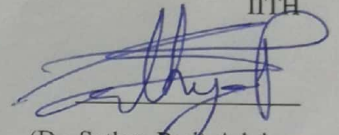
IITH



(Dr. Sobhan Babu) Examiner

Dept. of CSE

IITH



(Dr. Sathya Peri) Advisor

Dept. of CSE

IITH



(Dr. Sparsh Mittal) Chairman

Dept. of CSE

IITH

## **Acknowledgements**

I would first like to thank my Thesis Advisor, Prof. Sathya Peri, who was always available whenever I ran into any problems in my research or writing. I am extremely grateful for his valuable guidance and sincere advice to steer me in the right direction, whenever he thought I needed it. This work wouldn't have been achievable without his constant guidance and encouragement.

I would like to thank my parents who molded me into the person that I am today. For their unconditional love and support for my whole life. They worked hard to support me and set me on the path I am on today. Without their blessings and encouragement, I wouldn't have been where I am at right now. I would also like to thank my other family members for always believing in me, no matter the circumstances.

I am also extremely grateful to my colleagues at IIT Hyderabad, with whom I could have a lot of helpful technical discussions, regarding any problems I was working on. Being a part of such a great group helped me a lot in both my academic and personal life.

Last but not the least, I feel blessed to have been a part of IIT Hyderabad, and it's amazing people, for the past two years. I learned a lot over here and I'd like to thank everyone who gave me any sort of help, advice and support during this time.

# Dedication

To my family, friends and teachers.

## Abstract

Garbage Collection in concurrent data structures, especially lock-free ones, pose multiple design and consistency challenges. In this instance, we consider the case of concurrent sets. A set is a collection of elements, where the elements are ordered and distinct. These two invariants are always maintained at every point in time.

Sets are usually represented as a linked list of nodes, with each node denoting an element in the Set. Operations on the set include adding elements to the set, removing elements from it and searching for elements in it. Currently, multiple implementations of concurrent sets already exist. LazyList[1], Hand-over-hand List[2] and Harris' List[3] are some of the well-known implementations. However none of these implementations employ, or are concerned with garbage collection of deleted nodes. Instead each implementation ignores deleted nodes or depends on the language's garbage collector to handle them.

Additionally, Garbage collection in concurrent lists, that use optimistic traversals or that are lock-free, is not trivial.

For example, in Lazy List and Harris' List, they allow a thread to traverse a node or a sequence of nodes after these nodes have already been removed from the list, and hence possibly deleted. If deleted nodes are to be reused, this will potentially lead to the ABA problem.[4]

Moreover, some languages like C++ do not have an in-built garbage collector. Some constructs like Shared Pointers[5] provide a limited garbage collection facility, but it degrades performance by a large scale. Integrating Shared Pointers into a concurrent code is also not a trivial task.

In this thesis, we propose a new representation of a concurrent set, GCList, which employs in-built garbage collection. We propose a novel garbage collection scheme that implements in-built memory reclamation whereby it reuses deleted nodes from the list. We propose both lock-based and lock-free implementations of GCList. The garbage collection scheme works in parallel with the Set operations.

In our experiments with varying workloads and randomised Set operations, GCList shows comparable performance to LazyList[1] & Harris' List[3] while outperforming Shared Pointers[5], Hazard Pointers[6] and Hand-over-hand List[2]. GCList also consumed 3-4 times less memory as compared to LazyList[1] and Harris' List[3] and is comparable to Shared Pointers[5] and Hazard Pointers[6].

# Contents

Declaration . . . . .	ii
Approval Sheet . . . . .	iii
Acknowledgements . . . . .	iv
Abstract . . . . .	vi
<b>Nomenclature</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Introduction to Concurrent Sets . . . . .	1
<b>2 System Model and Preliminaries</b>	<b>2</b>
2.1 System Model & Preliminaries . . . . .	2
<b>3 Literature Review</b>	<b>3</b>
3.1 Hand-Over-Hand List . . . . .	3
3.2 LazyList . . . . .	3
3.3 LockFreeList . . . . .	4
3.4 Reference Counting . . . . .	5
3.5 Pointer-based techniques . . . . .	6
<b>4 Our Proposal: GCList</b>	<b>7</b>
4.1 Our Algorithm: GCList . . . . .	7
4.1.1 GCLBList . . . . .	7
4.1.2 GCLFList . . . . .	17
<b>5 The Pool</b>	<b>26</b>
5.1 The Pool . . . . .	26
5.1.1 The blocking unbounded total queue . . . . .	26
5.1.2 The unbounded Lock-free queue . . . . .	26
<b>6 Results</b>	<b>28</b>
6.1 Setup . . . . .	28
6.2 Results . . . . .	30
6.3 Analysis of Results . . . . .	36
<b>7 Conclusion</b>	<b>37</b>

7.1 Conclusion and Future Work . . . . .	37
<b>References</b>	<b>38</b>



# Chapter 1

## Introduction

### 1.1 Introduction to Concurrent Sets

Sets are a collection of items where the set items are ordered and distinct. These two properties of the set, commonly called as invariants, have to be maintained at every point of time. Sets are represented as a linked-list of nodes where each node in the set denotes a distinct item. Operations include adding items to the set, removing items from the set and searching for an item from the set.

Currently, multiple list-based implementation of concurrent sets are available. LazyList [1], Hand-over-Hand List [2] and Harris's LockFreeList [3] are some common examples. However none of these implementations address the issue of garbage collection of nodes deleted from the list. Either the algorithm ignores the issue or it relies on the language's garbage collector to handle it for them.

There are several reasons to implement our own memory management scheme. Languages such as C and C++ do not provide garbage collection and often it is more efficient to do our own memory management. C++ has some constructs like Shared Pointers [5] that offer limited garbage collection facility. Other garbage collection techniques like Stop-the-World and Hazard Pointers[6] are also available. Even though Shared Pointers, Hazard Pointers [6] and these other garbage collection schemes are very generic techniques, since they can be applied to almost all concurrent data structures, they are expensive and cost a lot in terms of performance and the extra data structures required to implement them.

Integrating Shared Pointers, Hazard Pointers and these other garbage collection schemes into a concurrent data structure is also not a trivial task. And more often than not, they are not very optimized for performance. They become even more complicated in case of lock-free data structures employing lock-free methods. Garbage collection, in these cases, is byzantine [4].

In this thesis, we concentrate on the garbage collection scheme for a concurrent set. We introduce a new representation of a concurrent set, **GCList**, with in-built garbage collection. Nodes that are removed from the set are collected in a "Pool" of deleted nodes, to be reused for later add operations. We introduce both lock-based and lock-free versions of GCList. We use the terms node, key and value interchangeably in this thesis.

# Chapter 2

## System Model and Preliminaries

### 2.1 System Model & Preliminaries

In this thesis, we assume that our system consists of finite set of  $p$  processors, accessed by a finite set of  $n$  threads that run in a completely asynchronous manner and communicate using shared objects. The threads communicate with each other by invoking higher-level methods on the shared objects and getting corresponding responses. Consequently, we make no assumption about the relative speeds of the threads. We also assume that none of these processors and threads fail.

**Safety:** To prove a concurrent data structure to be correct, *linearizability* proposed by Herlihy & Wing [7] is the standard correctness criterion in the concurrent world. They consider a history generated by a data structure which is collection of method invocation and response events. Each invocation of a method call has a subsequent response. A history is linearizable if it is possible to assign an atomic event as a *linearization point* inside the execution interval of each method such that the result of each of these methods is the same as it would be in a sequential history in which the methods are ordered by their linearization points [7].

**Progress:** The *progress* properties specifies when a thread invoking methods on shared objects completes in presence of other concurrent threads. Some progress conditions used in this thesis are mentioned here which are based on the definitions in Herlihy & Shavit. The progress condition of a method in concurrent object is defined as: (1) Blocking: In this, an unexpected delay by any thread (say, one holding a lock) can prevent other threads from making progress. (2) Deadlock-Free: This is a **blocking** condition which ensures that **some** thread (among other threads in the system) waiting to get a response to a method invocation will eventually receive it. (3) Wait-Free: This is a **non-blocking** condition which ensures that **every** thread trying to get a response to a method, eventually receives it[8].

# Chapter 3

## Literature Review

We discuss some of the list-based set algorithms in this section and some existing garbage collection techniques that can be used in concurrent sets.

### 3.1 Hand-Over-Hand List

In this list-based representation of a set, also called **lock-coupling** [2], each thread traverses the list from the head of the list, while acquiring fine-grained locks in a hand-over-hand manner. Each thread acquires the lock for the next node and then releases the lock for the current node.

All operations require the usage of locks which may affect the overall performance of the list, even though garbage collection in this list is a fairly trivial task. A guarantee exists that only one thread can have a reference to a node at any particular time. Any deleting thread can free a deleted node, without compromising the Safety property of the list.

### 3.2 LazyList

An improvement over the Hand-over-Hand list is the LazyList [1]. Threads traverse the list **optimistically**, without using any locks. Nodes are locked only when the required pair are found. An additional boolean field called “marked” field is associated with every node. The “marked” field is used to identify nodes that have been deleted but are still reachable from the head of the list.

In LazyList, nodes are deleted in two steps:

- **Logical deletion:** The marked field is set to true.
- **Physical deletion:** The node’s predecessor’s next reference is swung to the node’s successor.

The contains method is completely wait-free. It traverses the list without using any locks. It’s easy to see that garbage collection, in this case, is not so trivial. It may lead to an issue known as the “**ABA Problem**” [4]. Figures 3.1 to 3.3 depict the ABA problem[4] in LazyList [1].

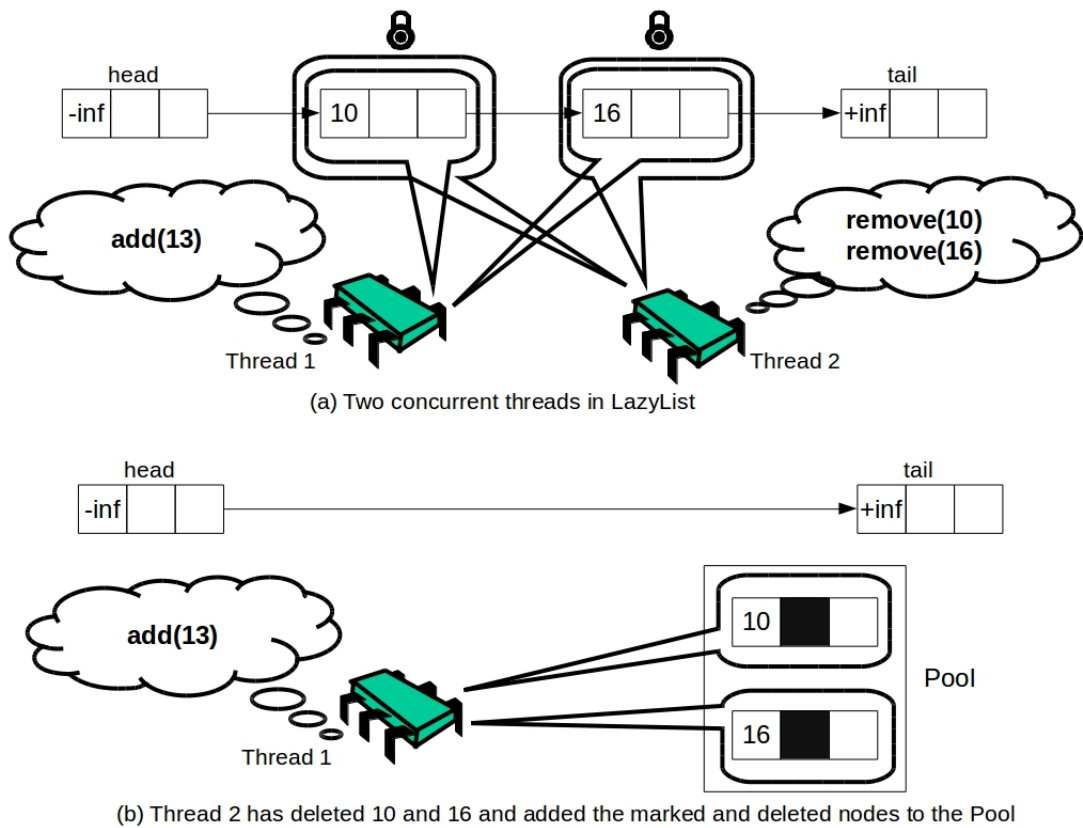


Figure 3.1: The ABA Problem in LazyList (Part 1)

### 3.3 LockFreeList

The LockFreeList [9] is an extension of the LazyList [1], where locks are eliminated altogether from the list operations and all the methods are non-blocking [8].

The list uses an AtomicMarkableReference [10] object as a part of its structure, which allows a thread to atomically read and update both the boolean mark and the next reference of a node. The list also uses compareAndSet or CAS calls for its operations.

The remove method is similar to LazyList [1], in that deletion is done in two steps.

- A CAS call is used to set the marked field of a node.
- Another CAS call is used on the node's predecessor to physically delete the node from the list.

An important difference between LockFreeList [9] and LazyList [1] is that LockFreeList never traverses logically marked nodes. Instead the encountered marked nodes are physically deleted from the list. Essentially, threads "help" out other slower threads that have completed the first CAS call but not the second.

It can also be seen that similar to LazyList [1], LockFreeList [9] is also vulnerable to the ABA problem [4].

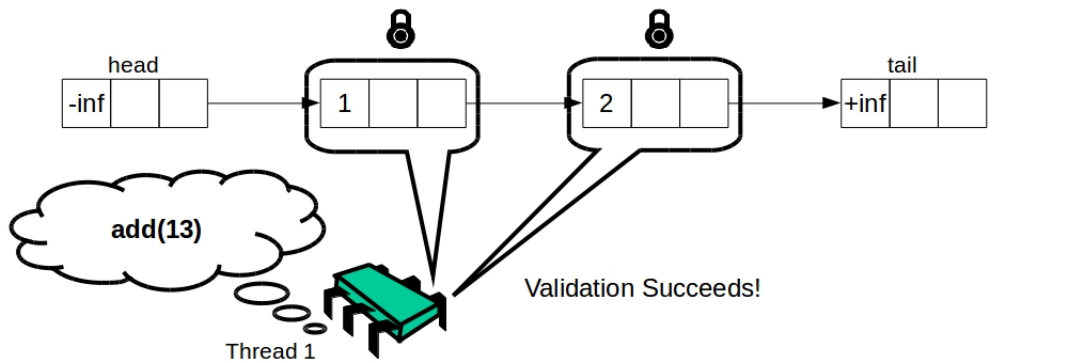
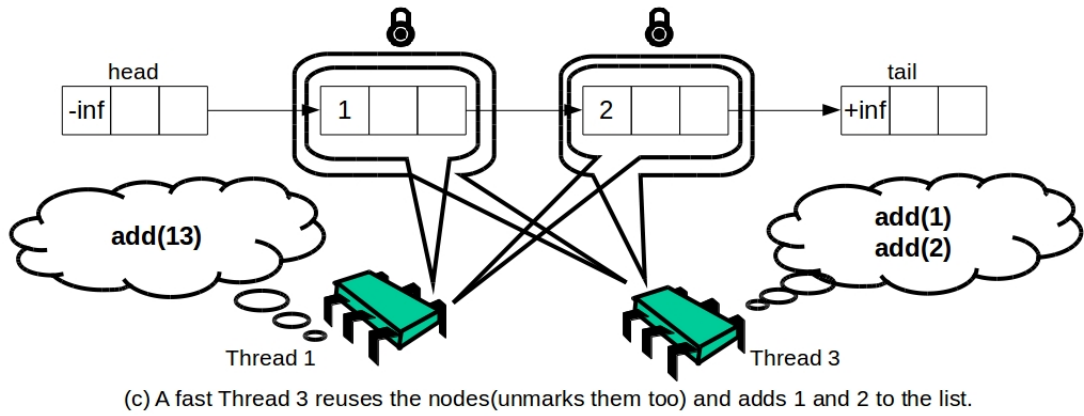


Figure 3.2: The ABA Problem in LazyList (Part 2)

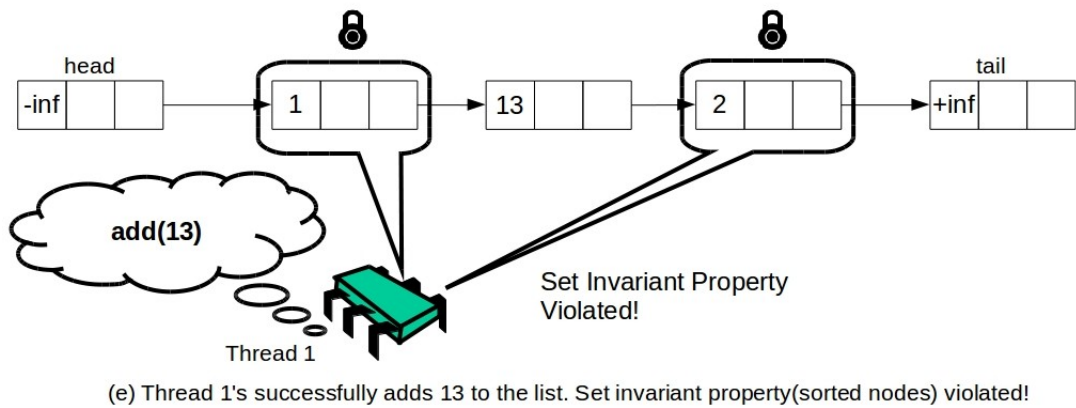


Figure 3.3: The ABA Problem in LazyList (Part 3)

### 3.4 Reference Counting

In a dynamic and concurrent data structure, arbitrary objects can continuously and concurrently be added or removed from the data structure. And multiple owners may have a reference to the shared objects. Unsafe

freeing of a node may lead to safety issues and possible crashes [11].

So, before freeing a shared object, it should be checked that there are no remaining references to it. This should also include possible local references to the shared object that any thread might have, as a read or write access to the memory of a reclaimed object might be fatal to the correctness of the data structure and/or to the whole system [11].

In the “Reference-Counting” category of garbage collection techniques, shared counters are assigned to objects and they are used to count the number of references to any object at any given time [11]. In other words, a group of owners share the ownership for an object. This group is responsible for deleting that object when the last one among them releases that ownership. The shared object can be freed if and only if the counter becomes zero [12].

This method, however is expensive. A shared atomic counter has to be associated with every object [11] [12]. Getting a reference to an object and incrementing the shared counter has to be an atomic operation. Same thing applies when losing the reference to the object and decrementing the shared counter. Even a simple read operation from the shared object has to increment the shared counter. Essentially, the memory read becomes a read-modify-write operation [13].

In C++, Shared Pointers [5] comes under this category of garbage collection techniques. However they are susceptible to data races when the shared object is accessed without proper synchronization. To prevent this, the Shared Pointer atomic operations have to be used for every read and write, from and to, the shared object [5]. This heavily affects the performance of the data structure.

### 3.5 Pointer-based techniques

Pointer-based techniques such as Hazard Pointers [6] explicitly mark live objects (objects that threads can access) which are not de-allocated. Pointer-based schemes suffer from two limitations: they must be customized to the data structure at hand, which makes them difficult to deploy; they publish each pointer that is used in a shared memory location, which is expensive in terms of synchronization.

Hazard Pointers (HP) and other pointer-based techniques will typically publish the pointer to each object they use, and then check that the pointer has not changed in the meantime. Such approach guarantees that an object which has been deleted will not be later dereferenced, at the cost of each reader doing synchronization on a per-object basis.

Because it requires validation of the pointer that will be accessed next, Hazard Pointers are lock-free for readers, although in some situations they can be made wait-free for readers. HP is wait-free bounded for reclamation, with the bound being proportional to the number of threads times the number of hazard pointers, because each reclaimer has to scan all the hazard pointers of all the other threads before deleting a node. In HP the retired nodes are placed in a retired list which is scanned once its size reaches an R threshold. In terms of memory usage, when the R factor is set to the lowest setting of 1, each reclaimer can have at most a list of retired nodes with a size equal to the number of threads minus 1, times the number of hazard pointers. If each thread has one such list of nodes pending to be deleted, at any given point in time there are at most  $O(N_{threads}^2)$  nodes to be deleted.

## Chapter 4

# Our Proposal: GCList

### 4.1 Our Algorithm: GCList

We introduce a new list-based set algorithm, **GCList**, which has an in-built garbage collection scheme. Nodes that have been deleted from the list are added to a “Pool” of deleted nodes. These nodes are reused for later add operations to the list. The set is represented as a linked-list of nodes, supporting the following operations:

- **add(key)**, adds key to the set, and returns true if and only if key was not already present in the set.
- **remove(key)**, removes key from the set, and returns true if and only if key was present in the set.
- **contains(key)**, searches for key in the set, and returns true if and only if key is present in the set.

We introduce two versions of GCList, a blocking version or **GCLBList** and a non-blocking version or **GCLFList**.

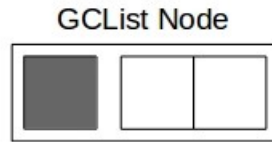
#### 4.1.1 GCLBList

Each node in the list consists of three fields: the key field, an **AtomicStampedReference** [14] object called as infoNext and a lock associated with the node. We have implemented our own AtomicStampedReference [14] in C++. The list is ordered according to the keys of each node. infoNext contains a reference to the next node in the list and an integer stamp associated with the node. Both the stamp and the reference can be read and updated atomically [14]. The lock field is a lock used for synchronization. Figure 4.1 denotes the structure and components of a GCList node.

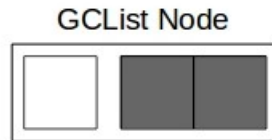
As mentioned earlier, we consider three operations on the list i.e. add, remove and contains. However we

```
class Node
{
    int key;
    AtomicStampedReference<Node> infoNext;
    mutex nodeLock;
};
```

Listing 4.1: GCLBList Node



(a) Shaded area represents the key value



(b) Shaded area represents the **AtomicStampedReference** Object, called infoNext, consisting of an integer stamp and a next reference. The stamp and the reference can be read and updated atomically.

Figure 4.1: GCList Node and its components

factor out functionality common to the add and remove methods by creating an inner Window class to help navigation. The common functionality is used to optimistically traverse the list and “find” the required pair of nodes required for each operation. The find method then returns the references to the nodes and their respective stamps in a Window object to the calling method.

### The find method

The find method is used by the add and remove methods to optimistically traverse the list. The thread gets a reference to the “head” node and keeps traversing the list in an optimistic hand-over-hand fashion. At every step of the traversal, the infoNext’s reference and stamp fields of a node are read atomically [14]. The thread keeps traversing the list until it finds the relevant pair of nodes, pred and curr. curr holds a reference to the first node with a key greater than or equal to the key that is being searched, in the list, with pred being curr’s predecessor. The find method returns a window object, containing references to pred and curr along with their respective stamps, to the calling method.

An important observation to be made here is the use of stamps during traversal. Stamps are used to detect synchronization conflicts by a traversing thread. This can be inferred from the working of the remove method later. If at any time during a thread’s traversal, the stamp of the pred node changes, a synchronization conflict with another “removing” thread is detected. The current thread “retries” its traversal from the head node.



---

**Algorithm 1** The find method

---

```
1: function Window find(Node head, int key)    ▷ Traverse from head and find node with key-value 'key'
2:   if head.infoNext.getReference() == tail then    ▷ head & tail are the only nodes in the list
3:     return Window (head, tail, head.infoNext.getStamp() , tail.infoNext.getStamp())
4:   end if
5:   while true do
6:     pred ← head    ▷ Start from the head
7:     curr ← pred.infoNext.get(predSt)    ▷ Read pred's infoNext's reference & stamp atomically
8:     while true do
9:       breakTest ← key ≤ curr.key    ▷ Break when key-value greater than or equal to re-
10:        succ ← curr.infoNext.get(currSt)    ▷ Read curr's infoNext's reference & stamp
11:        nPredSt ← pred.infoNext.getStamp()    ▷ Read pred's stamp again before advancing for-
12:        ward. This is the safety check to ensure we are
13:        traversing the list correctly, in increasing order
14:        of keys
15:       if predSt ≠ nPredSt then
16:         go to 5    ▷ If pred's new stamp is different from the one read previously,
17:         a synchronization conflict is detected. curr may have been
18:         deleted by another thread from the list. The thread restarts it's
19:         traversal to ensure correctness. If pred's stamp is still the same,
20:         then everything is fine.
21:       end if
22:       if breakTest then
23:         go to 22    ▷ If pred's stamp has not changed, everything is fine. Check if required
24:         pair of nodes has been found. If yes, break. Else, continue.
25:       end if
26:       pred ← curr    ▷ Keep advancing pred and curr in the list
27:       curr ← succ
28:       predSt ← currSt    ▷ Keep track of new pred's old stamp to be used later, to detect syn-
29:       chronization conflicts
30:     end while
31:     return Window(pred, curr, predSt, currSt)    ▷ Return pred and curr, along with their
32:     stamps, encapsulated in a window object.
33:   end while
34: end function
```

---

**The validate method**

The validate method is used to ensure that the calling method has locked the correct pair of nodes. It uses the stamps and references returned by the find method to ensure that both pred and curr are still present in the list and pred is still pointing to curr. If the stamps of either node has changed or pred is no longer pointing

to curr, then it signifies a synchronization conflict with another thread. The current thread then restarts its execution.

---

**Algorithm 2** The validate method

---

```

1: function bool validate(Node pred, int predSt, Node curr, int currSt)
    ▷ Checks consistency of locked nodes 'pred' & 'curr', using
    their stamps, predSt & currSt
2:   nCurr ← pred.infoNext.get(predSt)  ▷ Re-read pred's infoNext's reference and stamp atomically
3:   nCurrSt ← curr.infoNext.getStamp()  ▷ Re-read curr's infoNext's stamp atomically
4:   return predSt == nPredSt && currSt == nCurrSt && curr == nCurr
    ▷ Checks if pred is still pointing to curr. And if any of their stamps have changed from
    their old values. If yes, a conflict is detected. Returns true or false to calling method.
5: end function

```

---

**The remove method**

The remove method is used to remove key from the set, returning true if and only if key was in the set. It calls the “find” method to determine the correct pair of nodes for the remove operation. The nodes are locked and then validation is performed using the “validate” method. If validation fails, the nodes are unlocked and the thread retries, otherwise it continues its operation.

Deletion is performed in two steps:

- **Step 1:** pred's infoNext's reference is swung to curr's infoNext's reference and pred's infoNext's stamp is incremented by one. This operation to update pred's infoNext's reference and stamp fields is atomic.
- **Step 2:** curr's infoNext's stamp is incremented by 1. This marks the successful deletion of curr from the list.

Figure 4.2 shows the deletion steps of GCLBList. Figures 4.3 - 4.4 shows the case of two concurrent deleting threads in the list.

After curr has been successfully deleted, it is added to the “Pool”. A **Pool** is a concurrent data structure which is used to hold the deleted nodes. These deleted nodes can now be reused for later add operations.

Now, an important thing to discuss in this section is why does a thread traversing the list, in the find or contains method, has to retry if the pred's stamp changes. Based on the working of the “find” method, we can see that if any thread has a reference to curr, it should also have read pred's old stamp. This is because reads from an AtomicStampedReference [14] object is atomic. At this point, if curr were to be deleted from the list, pred's stamp would have been incremented, in Step 1. Again, this updation of pred's infoNext fields is atomic.

If the current thread were to continue its traversal, it may instead traverse the Pool or some other part of the list, since we have no guarantees about curr's position after its deletion. Instead, before advancing pred and curr in the list, we check pred's stamp again. If it has changed, it implies that curr may have been deleted and the current thread is in a synchronization conflict with a removing thread. The current thread then restarts its traversal from the list's head again. If pred's stamp is unchanged though, it implies that curr is still a part of the list and the thread can advance pred and curr.

Conversely, we can say that if a thread has read pred's updated stamp at the first read, then it cannot have a reference to curr. Again, this is because the updation of pred's infoNext fields is atomic [14].

Step 2 is the **linearization point** for a successful remove method. Step 2 ensures that a thread which has a reference to *curr* and is waiting to lock it, will fail in its validation later. This is because it will have previously read the old value of *curr*'s stamp. The updated stamp of *curr* will cause the new thread to fail in its validation and restart. An unsuccessful remove would be **linearized** when a node with a key-value immediately greater than the required 'key' is found in the list.

---

**Algorithm 3** The remove method

---

```

1: function bool remove(Node head, int key)      ▷ Remove a node with key-value 'key' from the list
2:   while true do
3:     window ← find(head, key)
4:     pred ← window.pred, curr ← window.curr
5:     predSt ← window.predSt, currSt ← window.currSt
                                     ▷ Retrieve pred and curr, and their stamps, from the window object
6:     pred.lock()
7:     if !curr.tryLock() then
8:       pred.unlock()
9:       go to 6      ▷ Lock both the nodes. tryLock() is to prevent deadlocks, since there is no
                                     guarantee, that keys are being locked in increasing order
10:    end if
11:    if validate(pred, predSt, curr, currSt) then  ▷ Use validate to ensure the consistency of pred
                                     and curr
12:      if curr.key ≠ key then
13:        curr.unlock()
14:        pred.unlock()
15:        return false      ▷ If key is not present, unlock both nodes. And return false
16:      else
17:        stamp ← pred.infoNext.getStamp()
18:        temp ← curr.infoNext.getReference()
19:        pred.infoNext.set(temp, ++stamp)      ▷ Deletion Step 1: atomically swing pred's
                                     infoNext's reference to curr's infoNext's
                                     reference and increment pred's infoNext's
                                     stamp by 1
20:        temp ← curr.infoNext.get(stamp)
21:        curr.infoNext.set(temp, ++stamp)      ▷ Deletion Step 2: atomically increment
                                     curr's infoNext's stamp by 1
22:        Pool.set(curr)      ▷ Add deleted node 'curr' to the Pool. curr can be reused for later
                                     add operations
23:        curr.unlock()
24:        pred.unlock()
25:        return true      ▷ Unlock pred and curr. Return true
26:      end if
27:    end if
28:    curr.unlock()
29:    pred.unlock()
30:  end while
31: end function

```

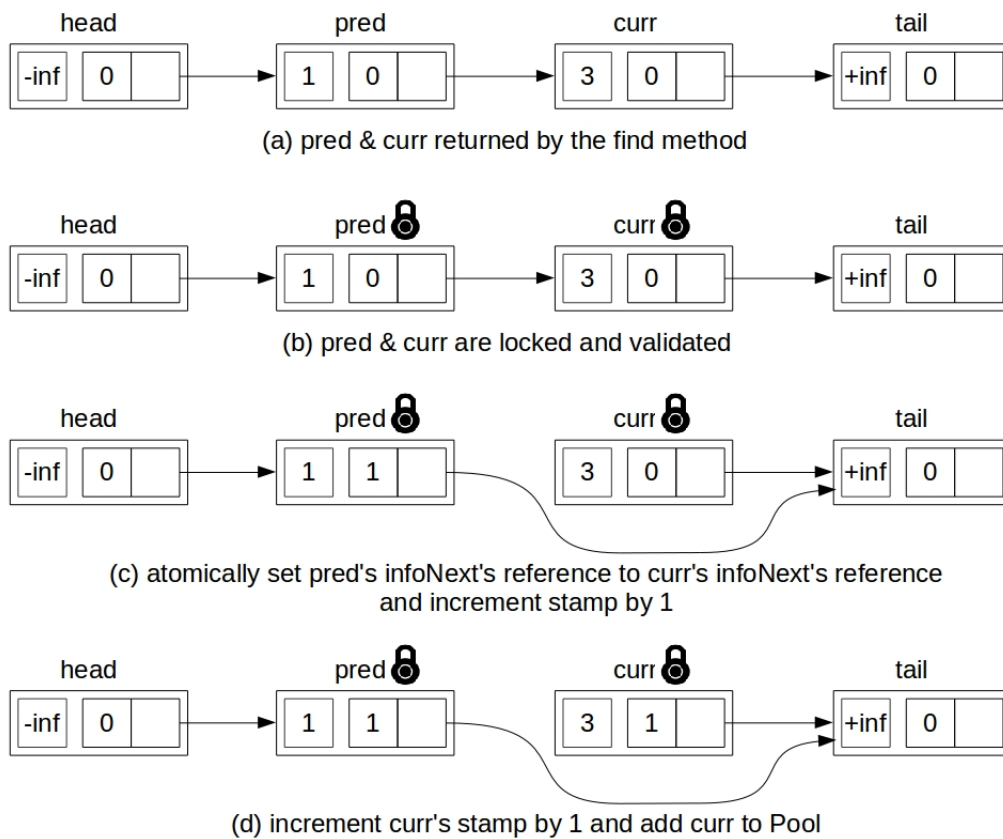


Figure 4.2: GCLBList: Remove Steps

### The add method

The add method is used to add a key to the list if and only if the key is not already present in the list. It calls the “find” method to determine the correct pair of nodes for the add operation. The nodes are locked and then validation is performed using the “validate” method. If validation fails, the nodes are unlocked and the thread retries, otherwise it continues its operation. The thread then queries the Pool (a data structure containing deleted nodes) for a node. If the Pool is not empty, a node is returned to be reused. Else, the thread creates a new node. It then inserts the new node, unlocks pred and curr and returns true.

The step in which pred’s infoNext’s reference is set to the new node is the **linearization point** for the add method. An unsuccessful add would be **linearized** when a node with a key-value equal to the required ‘key’ is found in the list.

---

**Algorithm 4** The add method

---

```
1: function bool add(Node head, int key) ▷ Add a node with key-value 'key' from the list
2:   while true do
3:     window ← find(head, key)
4:     pred ← window.pred, curr ← window.curr
5:     predSt ← window.predSt, currSt ← window.currSt
▷ Retrieve pred and curr, and their stamps, from the window object
6:     pred.lock()
7:     if !curr.tryLock() then
8:       pred.unlock()
9:       go to 6 ▷ Lock both the nodes. tryLock() is to prevent deadlocks, since there is no
guarantee, that keys are being locked in increasing order
10:    end if
11:    if validate(pred, predSt, curr, currSt) then ▷ Use validate to ensure the consistency of pred
and curr
12:      if curr.key == key then
13:        curr.unlock()
14:        pred.unlock()
15:        return false ▷ If key is already present, unlock both nodes. And return false
16:      else
17:        node ← Pool.get() ▷ Query the Pool for a node
18:        if node ≠ nullptr then
19:          node.key ← key ▷ node has been retrieved from pool. Reuse for new add operation
20:        else
21:          node ← newNode(key) ▷ Pool is empty. Create new node.
22:        end if
23:        stamp ← node.infoNext.getStamp()
24:        node.infoNext.set(curr, stamp) ▷ Set new node's reference to curr. No need
to change new node's stamp
25:        stamp ← pred.infoNext.getStamp()
26:        pred.infoNext.set(node, stamp) ▷ Atomically set pred's infoNext's refer-
ence to new node. No need to change
pred's stamp
27:        curr.unlock()
28:        pred.unlock()
29:        return true ▷ Unlock pred and curr. Return true
30:      end if
31:    end if
32:    curr.unlock()
33:    pred.unlock()
34:  end while
35: end function
```

---

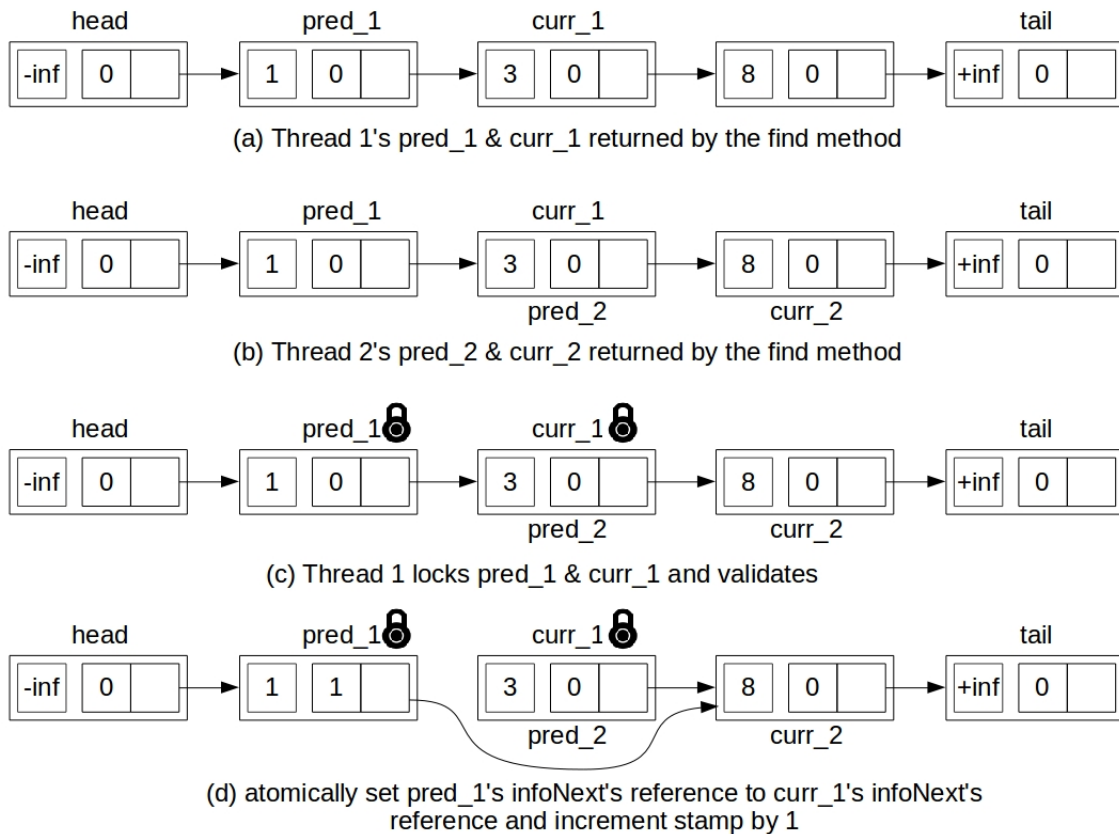


Figure 4.3: GCLBList: Two concurrent removing Threads (Part 1)

### The contains method

The contains method is similar to the find method. It starts from the “head” node and keeps traversing the list in an optimistic hand-over-hand fashion. At every step of the traversal, the infoNext’s reference and stamp fields of a node are read atomically [14]. The thread keeps traversing the list until it finds the first node with a key greater than or equal to the key that is being searched.

Similar to the find method, stamps are used to detect synchronization conflicts during traversal. If at any time during a thread’s traversal, the stamp of the pred node changes, a synchronization conflict with another “removing” thread is detected. The current thread “retries” its traversal from the head node.

The method returns true if and only if the key is present in the list. A successful contains is **linearized** when a matching key is found and the stamp of the predecessor hasn’t changed from its previous value. This shows that when curr was read by the thread, it was still a part of the list. The unchanged stamp of pred denotes that no other concurrent thread has deleted curr, while the current thread was obtaining its reference and reading its key-value. An unsuccessful contains would be when a node with a key-value immediately greater than the required key is found by the thread.

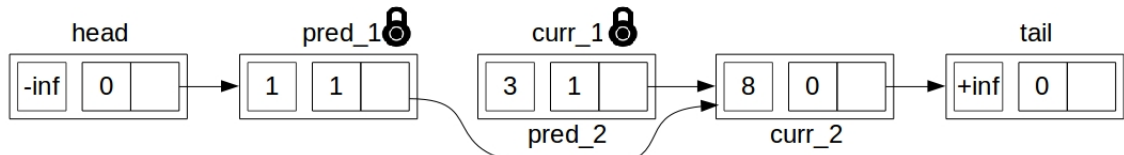
---

**Algorithm 5** The contains method

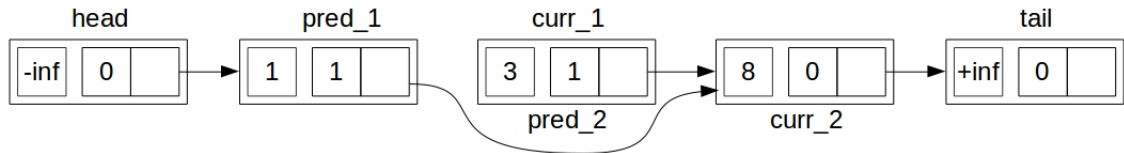
---

```
1: function bool contains(Node head, int key) ▷ Traverse from head and find node with key-value 'key'
2:   breakTest ← false
3:   while true do
4:     pred ← head ▷ Start from the head
5:     curr ← pred.infoNext.get(predSt) ▷ Read pred's infoNext's reference & stamp atomically
6:     currKey ← curr.key ▷ Read curr's key-value
7:     while true do
8:       breakTest ← key ≤ currKey ▷ Break when key-value greater than or equal to
9:       succ ← curr.infoNext.get(currSt) ▷ Read curr's infoNext's reference & stamp
10:      nPredSt ← pred.infoNext.getStamp() ▷ Read pred's stamp again before advancing forward. This is the safety check to ensure we are
11:      if predSt ≠ nPredSt then ▷ Read pred's stamp again before advancing forward. This is the safety check to ensure we are
12:        go to 3 ▷ traversing the list correctly, in increasing order of keys
13:      end if
14:      if breakTest then
15:        go to 22 ▷ If pred's new stamp is different from the one read previously, a synchronization conflict is detected. curr may have been
16:      end if ▷ deleted by another thread from the list. The thread restarts its traversal to ensure correctness. If pred's stamp is still the same,
17:      if breakTest then ▷ then everything is fine.
18:        go to 22 ▷ If pred's stamp has not changed, everything is fine. Check if
19:      end if ▷ required pair of nodes has been found. If yes, break. Else,
20:      pred ← curr ▷ continue.
21:      curr ← succ ▷ Keep advancing pred and curr in the list
22:      predSt ← currSt ▷ Keep track of new pred's old stamp to be used later, to detect
23:      currKey ← curr.key ▷ synchronization conflicts
24:      end while ▷ Read curr's key-value
25:      return currKey == key ▷ Return true if 'key' has been found. Else, return false.
26:    end while
27:  end function
```

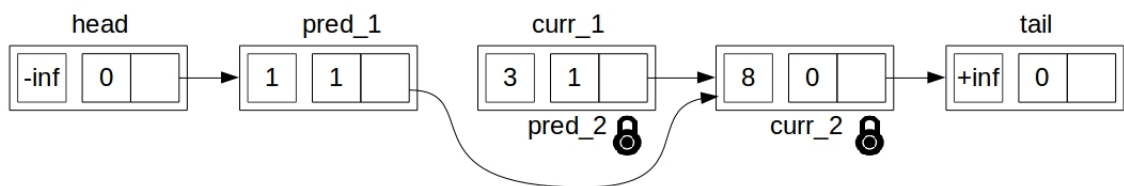
---



(e) increment curr\_1's infoNext's stamp by 1

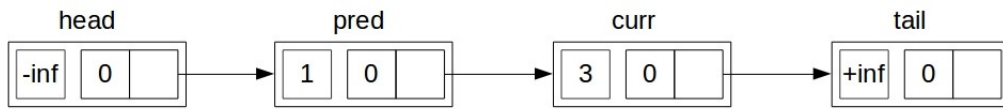


(f) Thread 1 unlocks pred\_1 and curr\_1

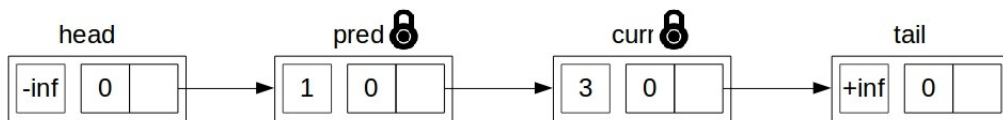


(g) Thread 2 locks pred\_2 and curr\_2 but validation fails since pred\_2's stamp has changed. Thread 2 restarts!

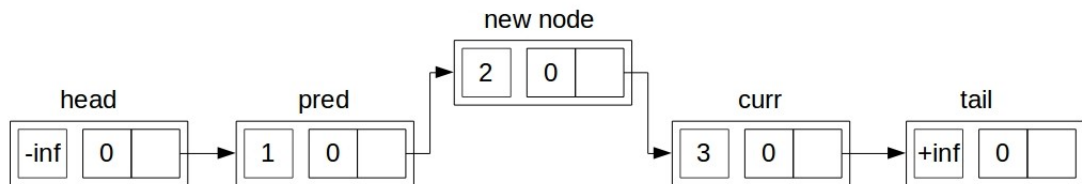
Figure 4.4: GCLBList: Two concurrent removing Threads(Part 2)



(a) pred & curr returned by the find method



(b) pred & curr are locked and validated



(c) add new node between pred & curr

Figure 4.5: GCLBList: Add Steps



```

class Node
{
public:
    int key;
    AtomicStampedReference<Node> infoNext;
}

```

Listing 4.2: GCLFList Node

## 4.1.2 GCLFList

GCLFList is the non-blocking version of our list-based set algorithm.

Each node in the list now consists of two fields, the key field and an AtomicStampedReference [14] object called as infoNext. The list is ordered according to the keys of each node. infoNext contains a reference to the next node in the list and an integer stamp associated with the node. Both the stamp and the reference can be read and updated atomically [14]. There is no lock field associated with the node anymore.

We instead use atomic functions like compareAndSet [14] or CAS to perform our operations on the list. Atomic operations [14] are used to atomically read and update the AtomicStampedReference object associated with each node. However, this also leads to complications. For example, if we follow the deletion steps of GCLBList, what happens in the case of two adjacent concurrent remove operations, using CAS? We can see that one of the nodes won't be removed from the list.

To solve this problem, we need a way to identify a marked node in the list, even though it may still be present in the list i.e. a logically deleted node. We differentiate between a logically deleted node and a node that is a part of the list by using parity of stamp.

- A node with an **even stamp** is a part of the list.
- A node with an **odd stamp** denotes a node that has been deleted from the list.

The deletion operation is also divided into two steps

- **Logical Deletion:** Increment curr's stamp by 1 using CAS [14] i.e marking curr. This step is the **linearization point** of the remove method.
- **Physical Deletion:** Swing pred's infoNext's reference to curr's infoNext's reference and increment pred's infoNext's stamp by 2, atomically using CAS [14].

We also adopt the concept of **Helping** i.e. if a traversing thread encounters a logically deleted or marked node, it attempts to first remove the node from the list, before advancing forward.

The methods are similar to GCLBList with a few modifications to incorporate the above changes.

### The find method

The find method is used by the add and remove methods to optimistically traverse the list. The thread gets a reference to the "head" node and keeps traversing the list in an optimistic hand-over-hand fashion. At every step of the traversal, the next reference and stamp of a node is read atomically [14]. The thread keeps traversing the list until it finds the relevant pair of nodes, pred and curr. It returns a window object, containing references to pred and curr along with their respective stamps, to the calling method.

As mentioned above, each time the thread encounters a marked node i.e. a node with an odd stamp, it attempts to physically delete the node first before advancing. If the CAS operation for the physical deletion succeeds, the node advances forward. Else it retries. Threads never traverse marked nodes because they lead to consistency issues.

For example, find may return a marked pred and an unmarked curr to the remove method trying to add a new node between pred and curr. If pred is physically removed by another thread before the new node could be added, the new node would end up being not added to the list. This difficulty arises because the current thread is not holding locks on pred and curr.

Similar to the previous find method, stamps are also used to detect synchronization conflicts by a traversing thread. If at any time during a thread's traversal, the stamp of the pred node changes, a synchronization conflict with another "removing" thread is detected. The current thread "retries" it's traversal from the head node.

---

**Algorithm 6** The find method

---

```

1: function Window find(Node head, intkey, Node prevCurr)
                                     ▷ Traverse from head and find node with
                                     key-value 'key'
2:   breakTest ← false, snip ← false
3:   while true do
4:     pred ← head                                     ▷ Start from the head
5:     curr ← pred.infoNext.get(predSt)             ▷ Read curr's infoNext's reference & stamp atomically
6:     while true do
7:       currKey ← curr.key                           ▷ Read curr's key value
8:       succ ← curr.infoNext.get(currSt)           ▷ Atomically read curr's infoNext's reference
                                               and stamp. Successor may be null if curr has
                                               been deleted
9:       if currSt mod 2 == 1 then
10:        snip ← pred.infoNext.compareAndSet(curr, succ, predSt, predSt + 2)
                                               ▷ This is the "helping" step. If curr is marked(stamp is odd), attempt
                                               to physically remove from the list. Done by calling an atomic CAS
                                               operation on pred, to atomically set pred's infoNext's reference to suc-
                                               cessor and increment stamp by 2
11:        if !snip then
12:          go to 3                                     ▷ If the CAS operation fails, restart the traversal
13:        end if
14:        Pool.set(curr)                               ▷ Else, add curr to the Pool.
15:        predSt += 2                                   ▷ And keep track of updated pred's stamp
16:      end if
17:      breakTest ← key ≤ currKey                     ▷ Break when key greater than or equal to re-
                                               quired key is found
18:      nPredSt ← pred.infoNext.getStamp()
                                               ▷ Read pred's stamp again before advancing for-
                                               ward. This is the safety check to ensure we are
                                               traversing the list correctly, in increasing order
                                               of keys

```

```

19:         if predSt  $\neq$  nPredSt then
20:             go to 3          ▷ If pred’s new stamp is different from the one read previously,
                                a synchronization conflict is detected. curr may have been
                                deleted by another thread from the list. The thread restarts it’s
                                traversal to ensure correctness. If pred’s stamp is still the same,
                                then everything is fine
21:         end if
22:         if breakTest then
23:             go to 34       ▷ If pred’s stamp has not changed, everything is fine. Check if required
                                pair of nodes has been found. If yes, break. Else, continue
24:         end if
25:         if !snip then
26:             pred  $\leftarrow$  curr      ▷ If no helping was done i.e. no marked node was found,
27:             curr  $\leftarrow$  succ      ▷ Keep advancing pred and curr in the list
28:             predSt  $\leftarrow$  currSt  ▷ Keep track of new pred’s old stamp to be used later, to detect syn-
                                chronization conflicts
29:         else
30:             curr  $\leftarrow$  succ      ▷ If helping was done to remove an encountered marked done, It
                                implies pred is still the same. Advance only curr
31:             snip  $\leftarrow$  false
32:         end if
33:         end while
34:         return Window(pred, curr, predSt, currSt)
                                ▷ Return pred and curr, along with their stamps, encapsulated in a Window object
35:     end while
36: end function

```

---

### The remove method

The remove method is used to remove key from the set, returning true if and only if key was in the set. It calls the “find” method to determine the correct pair of nodes for the remove operation.

Deletion of “curr” is performed in two steps as mentioned earlier. The step for logical deletion of “curr” is the linearization point for the remove method.

After curr has been successfully deleted, it is added to the “Pool”. These deleted nodes can now be reused for later add operations.

Now, what happens if any of the two CAS operations fail.

**Case 1:** CAS for logical deletion of curr fails. It implies that some other thread has performed a concurrent operation on curr and a synchronization conflict is detected. The current thread has to restart it’s operation.

**Case 2:** CAS for physical deletion fails. It implies that some other thread has performed a concurrent operation on pred. The current thread has two choices: it can depend on other traversing threads to “help” physically delete curr or it can traverse the list once more time to ensure curr’s deletion.

An important note is that incrementing the pred’s stamp by 2 during physical deletion prevents the ABA problem.

Figure 4.1.2 shows the deletion steps in GCLFList. Figures 4.7 - 4.8 shows the case of two concurrent

removing threads in the list.

A successful remove is **linearized** when the Logical deletion step succeeds. This ensures that even if the thread fails in the Physical deletion step, another concurrent traversing thread will **help** with the physical removal of the node from the list. The thread also has the option to retrace the list, find the marked node and physically delete it. An unsuccessful remove is **linearized** when an unmarked node with a key-value immediately greater than the required key is found in the list.

---

**Algorithm 7** The remove method

---

```

1: function bool remove(Node head, int key)           ▷ Remove a node with key-value 'key' from the
                                                    list
2:   while true do
3:     window ← find(head, key, nullptr)
4:     pred ← window.pred, curr ← window.curr
5:     predSt ← window.predSt, currSt ← window.currSt
                                                    ▷ Retrieve pred and curr, and their stamps, from the window object
6:     if curr.key ≠ key then
7:       return false                               ▷ If key is not present, return false
8:     else
9:       succ ← curr.infoNext.getReference()         ▷ Read curr's infoNext's reference
10:      snip ← curr.infoNext.compareAndSet(succ, succ, currSt, currSt + 1)
                                                    ▷ Deletion Step 1: Atomically increment curr's
                                                    infoNext's stamp by 1 using CAS i.e. Logical Deletion
11:      if !snip then
12:        go to 2                                   ▷ If CAS fails, restart the operation
13:      end if
14:      if pred.infoNext.compareAndSet(curr, succ, predSt, predSt + 2) then
                                                    ▷ Deletion Step 2: Atomically swing pred's infoNext's
                                                    reference to successor. And increment pred's
                                                    infoNext's stamp by 2 i.e. Physical Deletion
15:        Pool.set(curr)                             ▷ If physical deletion is successful, add curr to the Pool
16:      else
17:        find(head, key, nullptr)                   ▷ This step is optional. If physical deletion is unsuccessful,
                                                    retrace the list to remove it. Or depend on some
                                                    other thread to "help out"
18:      end if
19:      return true                               ▷ Return true on successful deletion. Note: Will return true even if only
                                                    Logical deletion is successful
20:    end if
21:  end while
22: end function

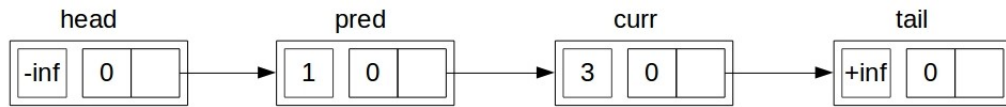
```

---

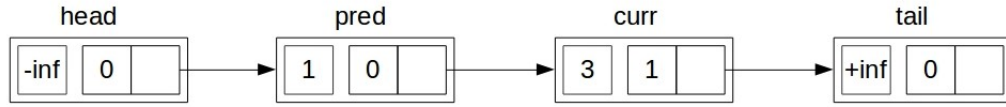
### The add method

The add method is used to add a key to the list if and only if the key is not already present in the list. It calls the "find" method to determine the correct pair of nodes for the add operation. The thread then queries the Pool for a node. If the Pool is not empty, a node is returned to be reused. Else, the thread creates a new node. It then inserts the new node, unlocks pred and curr and returns true. If the node is obtained from the Pool, it's stamp is incremented by 1 before inserting it into the list.

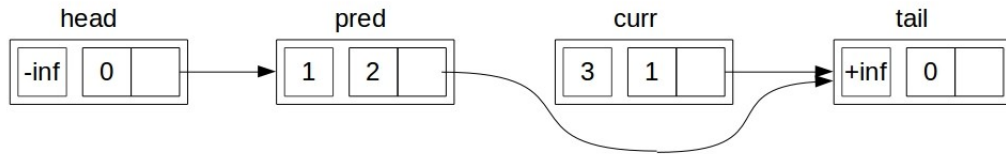
An important observation to be made here is if the adding thread's CAS call on pred to insert the new node



(a) pred & curr returned by the find method

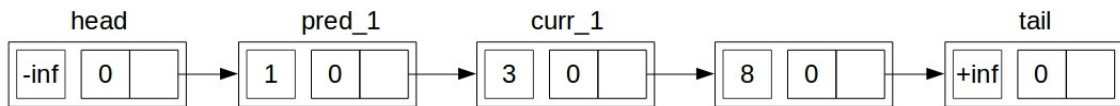


(b) atomically increment curr's infoNext's stamp by 1 i.e. Logical Deletion

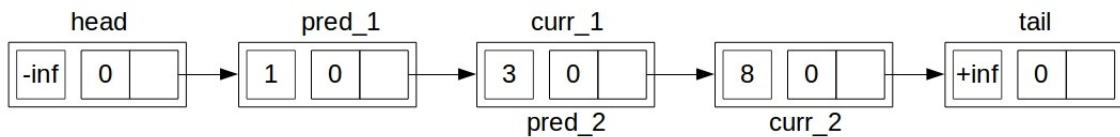


(c) swing pred's infoNext's reference to curr's infoNext's reference and increment stamp by 2 i.e. Physical Deletion

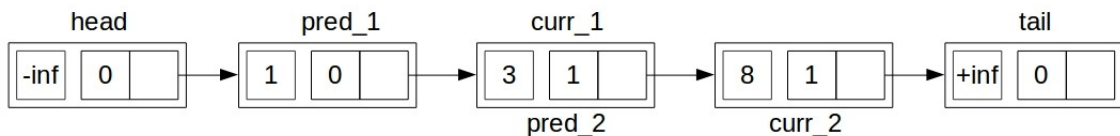
Figure 4.6: GCLFList: Remove Steps



(a) Thread 1's pred\_1 & curr\_1 returned by the find method



(b) Thread 2's pred\_2 & curr\_2 returned by the find method



(c) Thread 1 and 2 mark their respective curr nodes using CAS

Figure 4.7: GCLFList: Two concurrent removing Threads(Part 1)

to the list fails, it calls the find method again, resulting in a new pair of pred and curr. However, another concurrent adding thread may have meanwhile added the same key to the list. The current thread now cannot add the same key anymore and has to return false. Before doing that, if the node was retrieved from the Pool, it is added back again to it. Else, if it was a newly created node, we can delete it since we have a guarantee

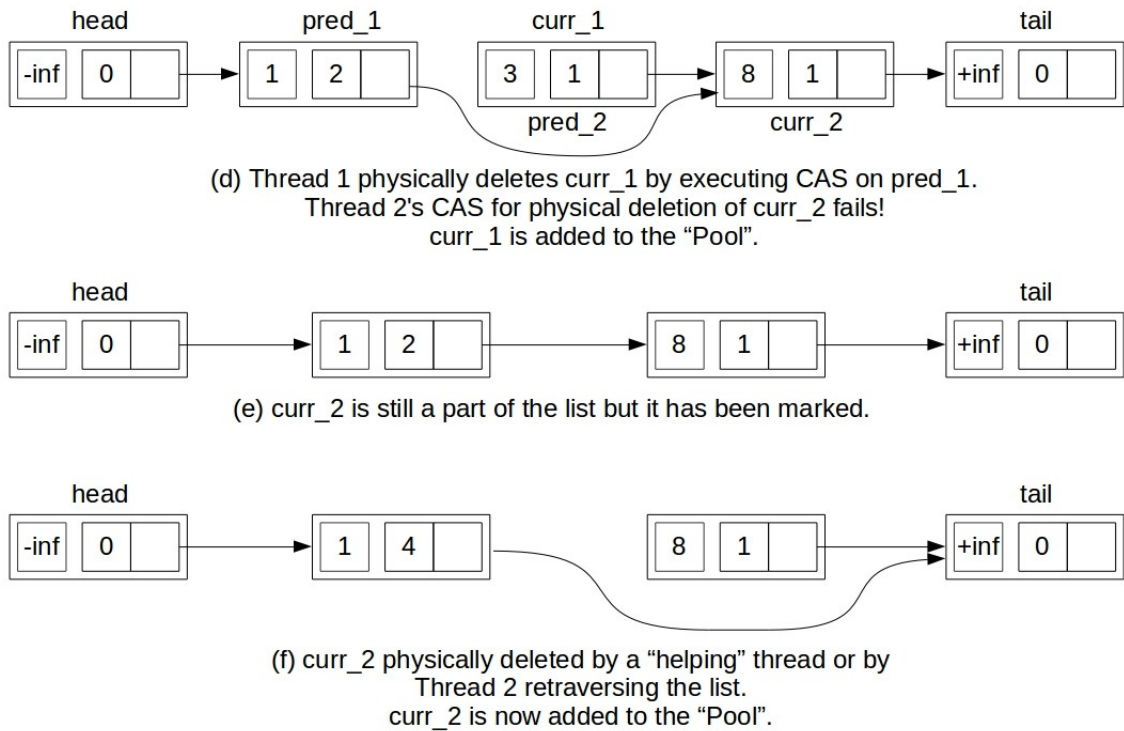


Figure 4.8: GCLFList: Two concurrent removing Threads(Part 2)

that no other thread has a reference to it.

Figure 4.9 shows the steps of adding a node to the list. Two concurrent threads, Thread 1 and Thread 2, are simultaneously trying to add a node with key-value '2' to the list. Thread 1 is trying to reuse a node that has been retrieved from the Pool. Thread 2 is trying to insert a newly created node. Both use a CAS call to try to insert their respective nodes, but only one CAS call will succeed. In this case, Thread 1's CAS call succeeds. Thread 2 retraverses the list but now finds that the key-value is already present. It now safely frees its newly created node, since no other thread holds a reference to it and returns false.

This scenario never occurred in GCLBList since once pred and curr were locked and validated and the key was previously absent from the list, there is a guarantee that the current thread would be able to add the key to the list successfully. Provided it doesn't crash midway before that.

The CAS call to set pred's infoNext's reference to the new node is the **linearization point** for this method. An unsuccessful add is **linearized** when an unmarked node with a matching key is found to be already present in the list.

---

**Algorithm 8** The add method

---

```
1: function bool add(Node head, int key) ▷ add a node with key-value 'key' from the list
2:   fromPool ← false
3:   node ← Pool.get() ▷ Query the Pool for a node
4:   if node == nullptr then
5:     node ← newNode(key) ▷ If Pool is empty, create a new node
6:     fromPool ← false
7:   else
8:     node.key ← key ▷ Else, node successfully retrieved from the Pool.
9:     nodeSt ← node.infoNext.getStamp()
10:    node.infoNext.set(nullptr, nodeSt + 1) ▷ Increment new node's stamp by 1, to make the stamp even
11:    fromPool ← true
12:  end if
13:  while true do
14:    window ← find(head, key, nullptr)
15:    pred ← window.pred, curr ← window.curr
16:    predSt ← window.predSt, currSt ← window.currSt ▷ Retrieve pred and curr, and their stamps, from the window object
17:    if curr.key ≠ key then
18:      nodeSt ← node.infoNext.getStamp()
19:      node.infoNext.set(curr, nodeSt) ▷ If 'key' is not already present in the list, set new node's infoNext's reference to curr
20:      if pred.infoNext.compareAndSet(curr, node, predSt, predSt) then ▷ Attempt to atomically CAS pred's infoNext's reference to new node. If CAS succeeds, return true
21:        return true
22:      else
23:        go to 13 ▷ Else, restart the operation. Note: Next iteration, some other thread may have added the new key instead. If so, then this thread will return false
24:      end if
25:    else ▷ Key is already present in the list
26:      if !fromPool then
27:        delete node ▷ new node was newly created by this thread. It can be safely freed, since no other thread has a reference to this node
28:      else
29:        nodeSt ← node.infoNext.getStamp()
30:        node.infoNext.set(nullptr, nodeSt - 1)
31:        Pool.set(node) ▷ node was retrieved from the Pool. Decrement node's stamp to make it odd again and add the node back to the Pool
32:      end if
33:      return false ▷ Return false since 'key' already present
34:    end if
35:  end while
36: end function
```

---

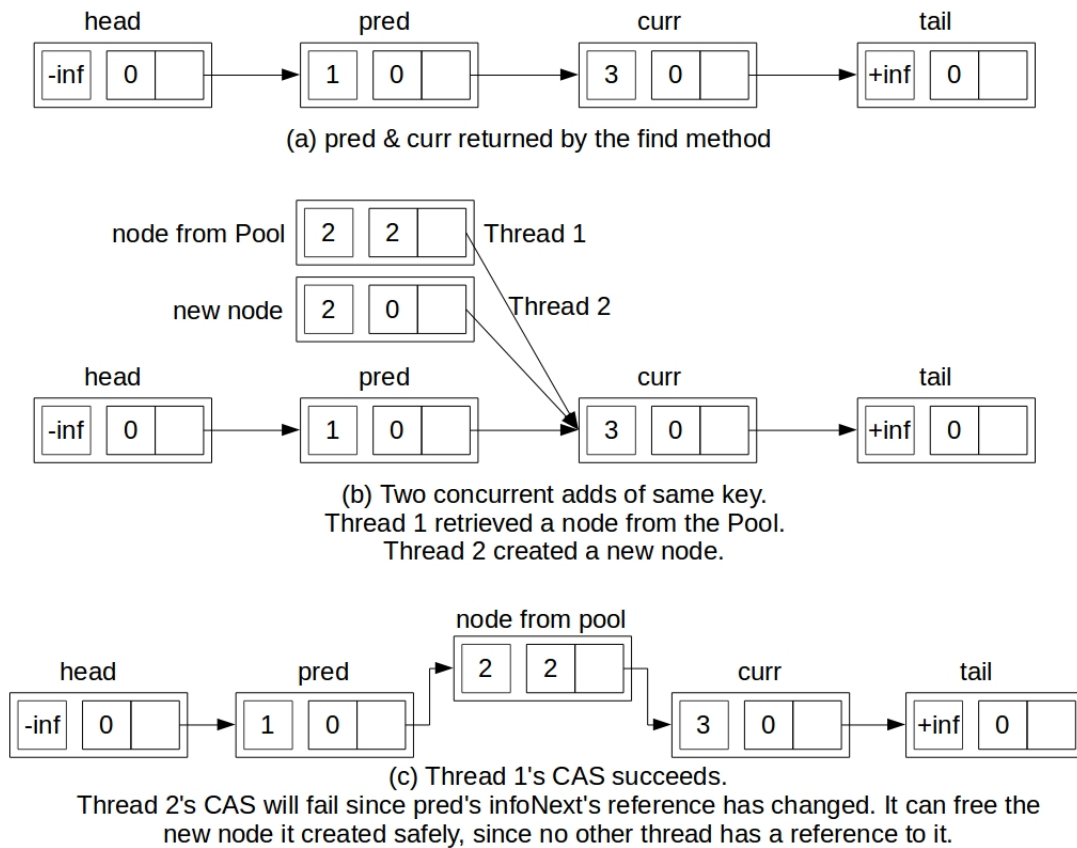


Figure 4.9: GCLFList: Add Steps

### The contains method

The contains method starts from the “head” node and keeps traversing the list in an optimistic hand-over-hand fashion. At every step of the traversal, the next reference and stamp of a node is read atomically [14]. The thread keeps traversing the list until it finds the first node with a key greater than or equal to the key that is being searched.

Again, stamps are used to detect synchronization conflicts during traversal. If at any time during a thread's traversal, the stamp of the pred node changes, a synchronization conflict with another “removing” thread is detected. The current thread “retries” its traversal from the head node.

The method returns true if and only if the key is present in the list and its infoNext's stamp is even. A successful contains is **linearized** when a node with a matching key-value is found and its stamp is even i.e. unmarked. An unsuccessful contains is **linearized** when an unmarked node with a key-value immediately greater than the required key is found.



---

**Algorithm 9** The contains method

---

```
1: function bool contains(Node head, int key)  ▷ Traverse from head and find node with key-value 'key'
2:   breakTest ← false
3:   while true do
4:     pred ← head  ▷ Start from the head
5:     while true do
6:       curr ← pred.infoNext.get(predSt)  ▷ Read curr's infoNext's reference & stamp
                                           atomically
7:       currKey ← curr.key  ▷ Read curr's key value
8:       succ ← curr.infoNext.get(currSt)  ▷ Atomically read curr's infoNext's reference
                                           and stamp. Successor may be null if curr has
                                           been deleted
9:       breakTest ← key ≤ currKey  ▷ Break when key greater than or equal to re-
                                           quired key is found
10:      nPredSt ← pred.infoNext.getStamp()  ▷ Read pred's stamp again before advancing for-
                                           ward. This is the safety check to ensure we are
                                           traversing the list correctly, in increasing order
                                           of keys
11:      if predSt ≠ nPredSt then
12:        go to 3  ▷ If pred's new stamp is different from the one read previously,
                                           a synchronization conflict is detected. curr may have been
                                           deleted by another thread from the list. The thread restarts it's
                                           traversal to ensure correctness. If pred's stamp is still the same,
                                           then everything is fine
13:      end if
14:      if breakTest then
15:        go to 20  ▷ If pred's stamp has not changed, everything is fine. Check if required
                                           node has been found. If yes, break. Else, continue
16:      end if
17:      pred ← curr  ▷ Keep advancing pred in the list
18:      predSt ← currSt  ▷ Keep track of new pred's old stamp to be used later, to detect syn-
                                           chronization conflicts
19:    end while
20:    marked ← currSt mod 2 == 1  ▷ Check if curr is marked i.e. odd stamp
21:    return currKey == key && !marked  ▷ Return true if and only if key is found and node
                                           is unmarked. Else, return false
22:  end while
23: end function
```

---

# Chapter 5

## The Pool

### 5.1 The Pool

The pool is a concurrent data structure that is used to hold the deleted nodes that have been reclaimed from the list. Ideally, any data structure that treats the node object as a “payload” can be used as the pool. In our experiments we used two different queue implementations to act as the pool. The code for both the queues has been kept in the appendix.

#### 5.1.1 The blocking unbounded total queue

This lock-based concurrent queue [9] uses two separate locks for each queue operation i.e. an `enqLock` to enqueue a deleted node to the queue and a `deqLock` to dequeue a node from the queue, respectively.

Before a thread performs an enqueue or a dequeue operation, it acquires the corresponding lock on the queue. After acquiring the lock, the thread performs its operation and releases the lock upon completion. The lock ensures that, at a particular time, only one thread is able to perform an enqueue or a dequeue operation on the queue.

#### 5.1.2 The unbounded Lock-free queue

This lock-free concurrent queue [9] uses atomic `compareAndSet` or `CAS` calls instead of locks for the queue operations. The `CAS` calls are used to enqueue a node into the queue and also to dequeue a node from the queue.

This lock-free implementation helps to prevent faster threads from starving, with the removal of coarse-grained locks. This queue implementation also uses the concept of helping, where faster thread help the slower threads to finish their queue operations.

The enqueue operation is done in two steps:

- The thread locates the last node in the queue and uses a `CAS` call to append the new node to the queue.
- It then uses another `CAS` call to change the queue’s tail from the previous last node to the current last node.

Since the above two CAS calls are not a single atomic operation, threads help each other to complete the second CAS, if a half finished enqueue operation is encountered.

An important attribute to be noted about the queue is that it also uses the AtomicStampedReference [14] object, in its' head and tail, to prevent the ABA problem [4] problem from occurring in the queue.

# Chapter 6

## Results

### 6.1 Setup

We tested both versions of GCList against existing implementations of a concurrent set namely: LazyList [1], Hand-over-Hand List [2], Harris’s LockFreeList [3], a Shared Pointer [5] version of LazyList and a LockFreeList using Hazard Pointers.

We used both the lock-based queue and the lock-free queue, as a Pool, in combination with the two versions of GCList. The resultant set representation is named by using the list’s name as prefix and pool’s name as suffix. For example, the GCLBList using the lock-based queue would be named GCLBListLBQueue and the GCLFList using the lock-free queue would be named GCLFListLFQueue.

The LazyList based on Shared Pointers has been named LazyList.SP. The LockFreeList using Hazard Pointers for Memory Reclamation has been named LockFreeList.HP.

Table 6.1: Table showing all the evaluated algorithms

<b>Algorithms</b>	<b>Description</b>
GCLBListLBQueue	The Lock-based variant of GCList using a Lock-based Queue as the Pool
GCLFListLFQueue	The Lock-free variant of GCList using a Lock-free Queue as the Pool
LazyList	The original LazyList without any garbage collection
LockFreeList	The original LockFreeList without any garbage collection
LazyList.SP	The LazyList using Shared Pointers for Garbage Collection
LockFreeList.HP	The LockFreeList using Hazard Pointers for Garbage Collection

Table 6.1 shows all the algorithms used for the evaluation and their respective descriptions. We tested the above mentioned algorithms versus our algorithms for both performance and memory consumption, with varying workloads and randomized Set operations.

For performance, we fix the total number of operations that each thread can perform, divided in varying ratios between adds, removes and contains. We allowed each algorithm to run for 10 seconds and measure the number of operations completed during said time period. The higher the number of operations completed by an algorithm in said time period, the better is it’s throughput.

For memory consumption, we fix the total number of operations that each thread can perform, divided in varying ratios between adds, removes and contains. We keep track of the number of times each thread allocates and de-allocates memory. Whenever the thread allocates new memory, a thread-local variable is incremented and whenever the memory is released, the variable is decremented.

At the end of all thread operations, the main thread consolidates the sum of all the thread-local variables. We take the ratio of a List's node count versus the Hand-Over-Hand List. This is because a thread can immediately free a node, after it's deletion, in the case of Hand-over-Hand List. We use this ratio to compare the memory consumed by an algorithm during it's entire execution. The lower the ratio of a List versus Hand-over-Hand List, the lower is it's memory consumption.

Based on the above setup and criteria, we ran the tests on different lookup-intensive and update-intensive environments and obtained the following graphs.

## 6.2 Results

Figures 6.1 to 6.3 show the performance graphs of the different algorithms with varying workloads. Figure 6.1 shows the throughput of each algorithm in a lookup-intensive environment. Figures 6.2 to 6.3 shows the throughput of each algorithm in an update-intensive environment.

Figures 6.4 to 6.6 show the graphs for the memory consumption ratio with respect to the Hand-Over-Hand List for the various algorithms. Figure 6.4 shows the memory consumption of each algorithm in a lookup-intensive environment. Figures 6.5 to 6.6 shows the throughput of each algorithm in an update-intensive environment. The plot for the Hand-Over-Hand list is a straight line and denotes our baseline.

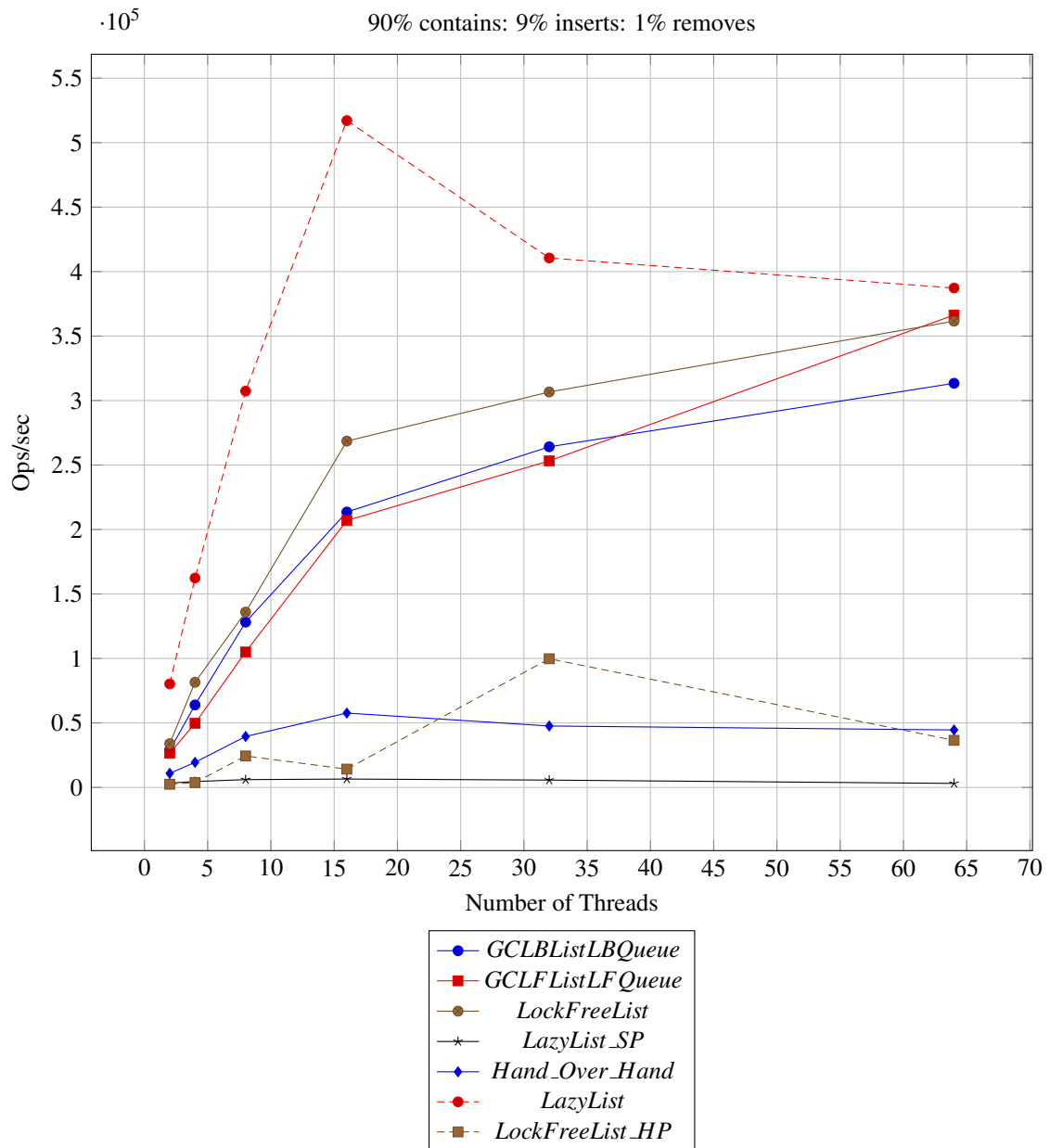


Figure 6.1: Performance Analysis with 10% writes

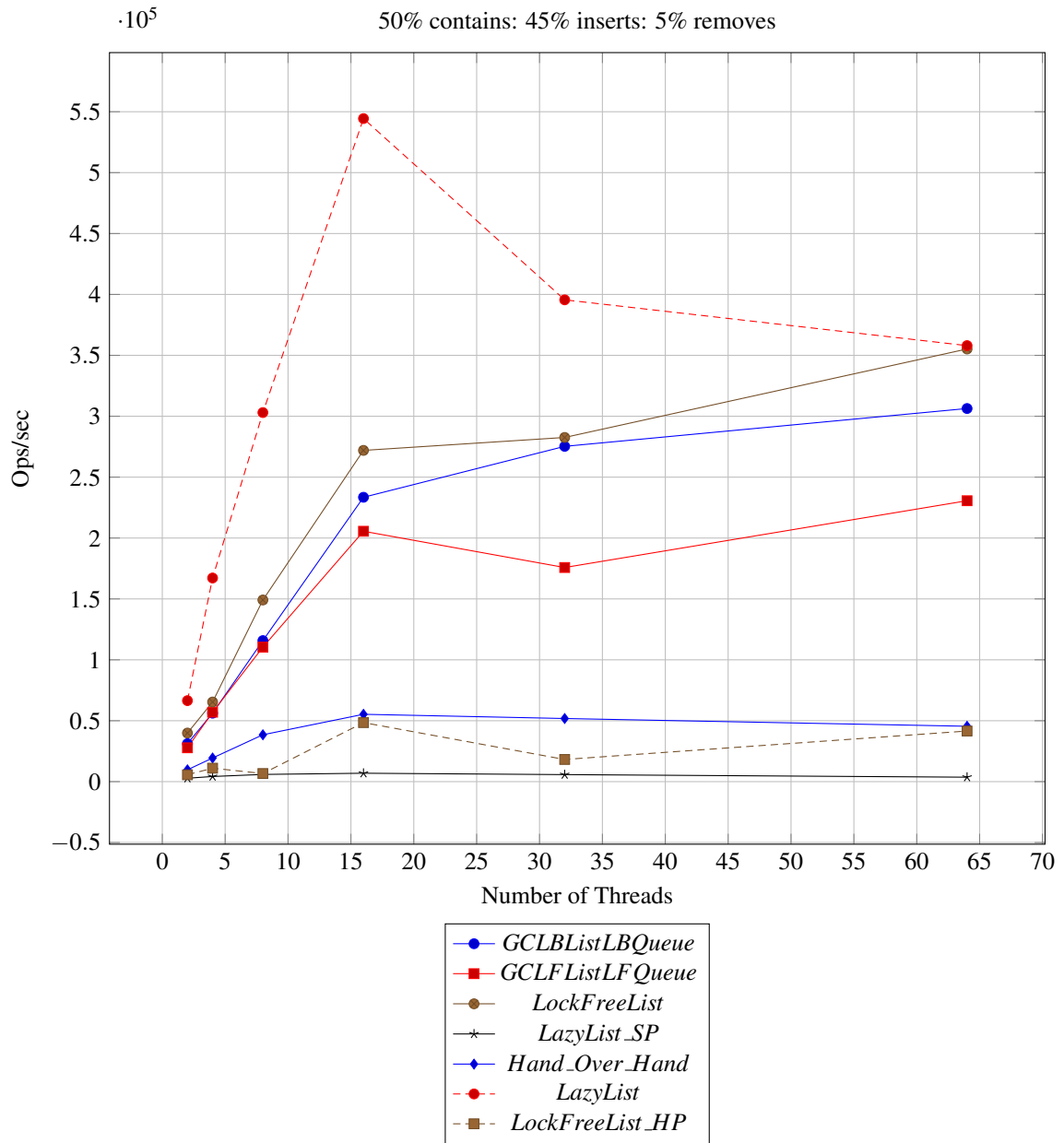


Figure 6.2: Performance Analysis with 50% writes

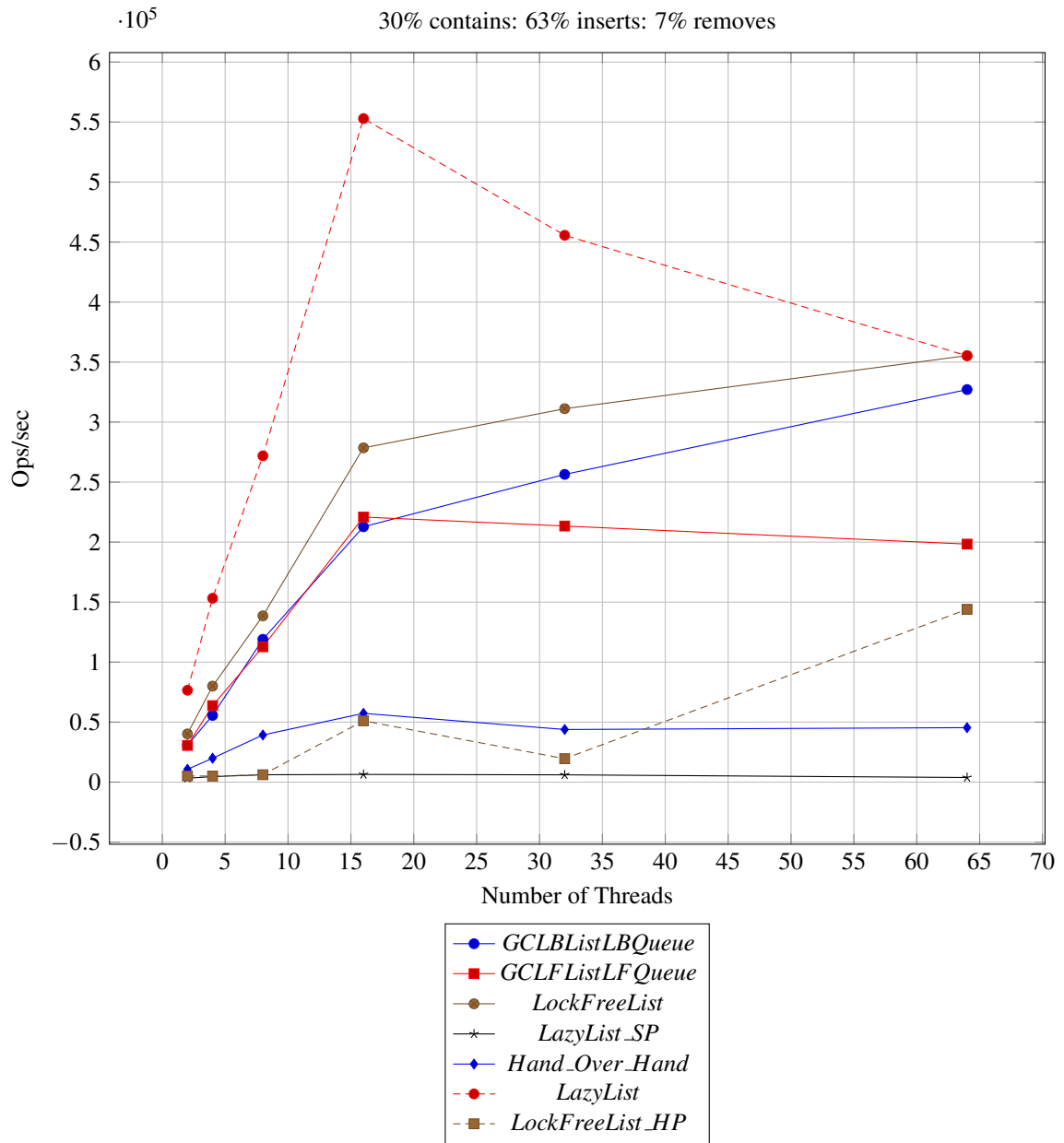


Figure 6.3: Performance Analysis with 70% writes



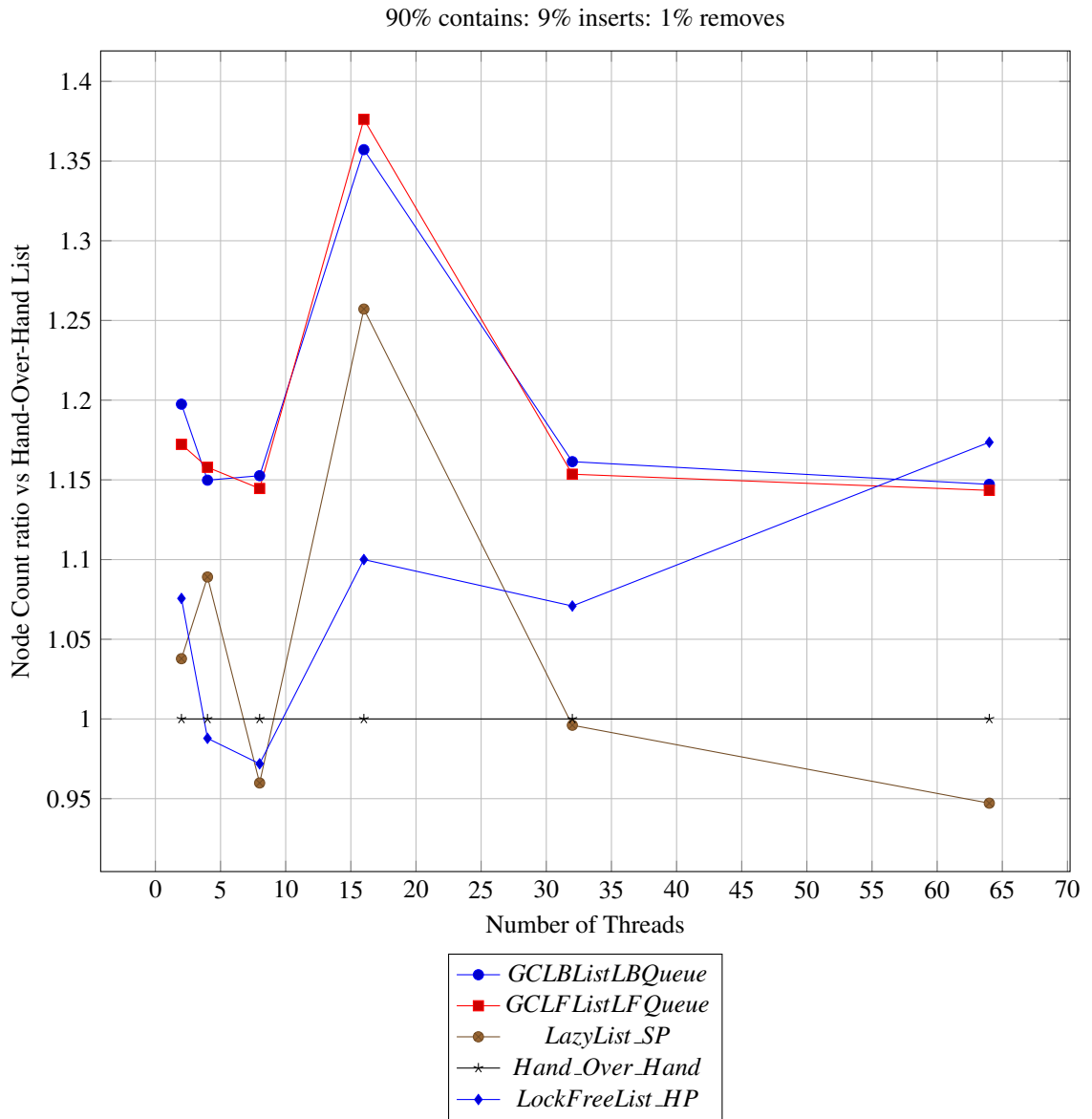


Figure 6.4: Memory Consumption Analysis with 10% writes

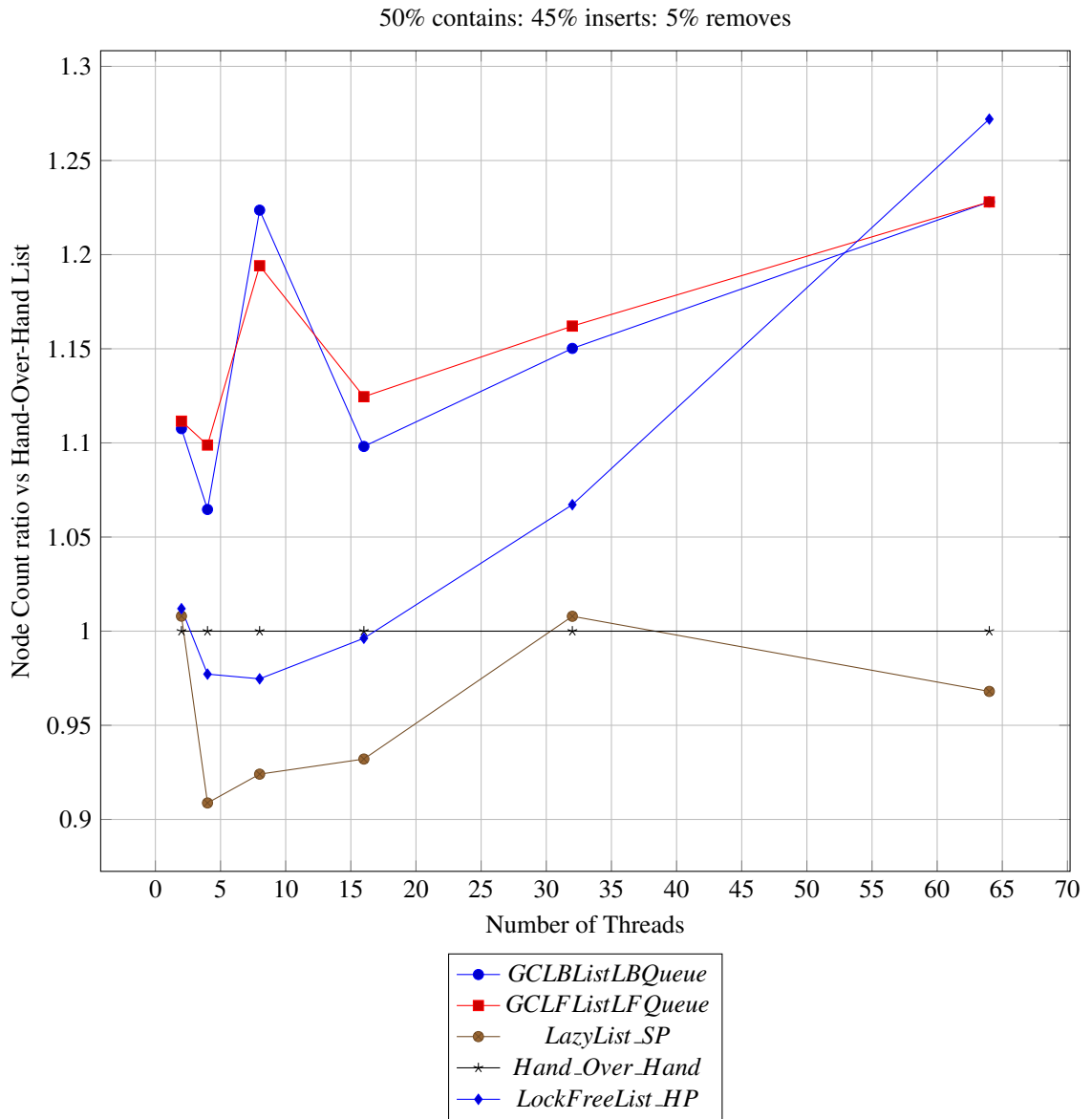


Figure 6.5: Memory Consumption Analysis with 50% writes

30% contains: 63% inserts: 7% removes

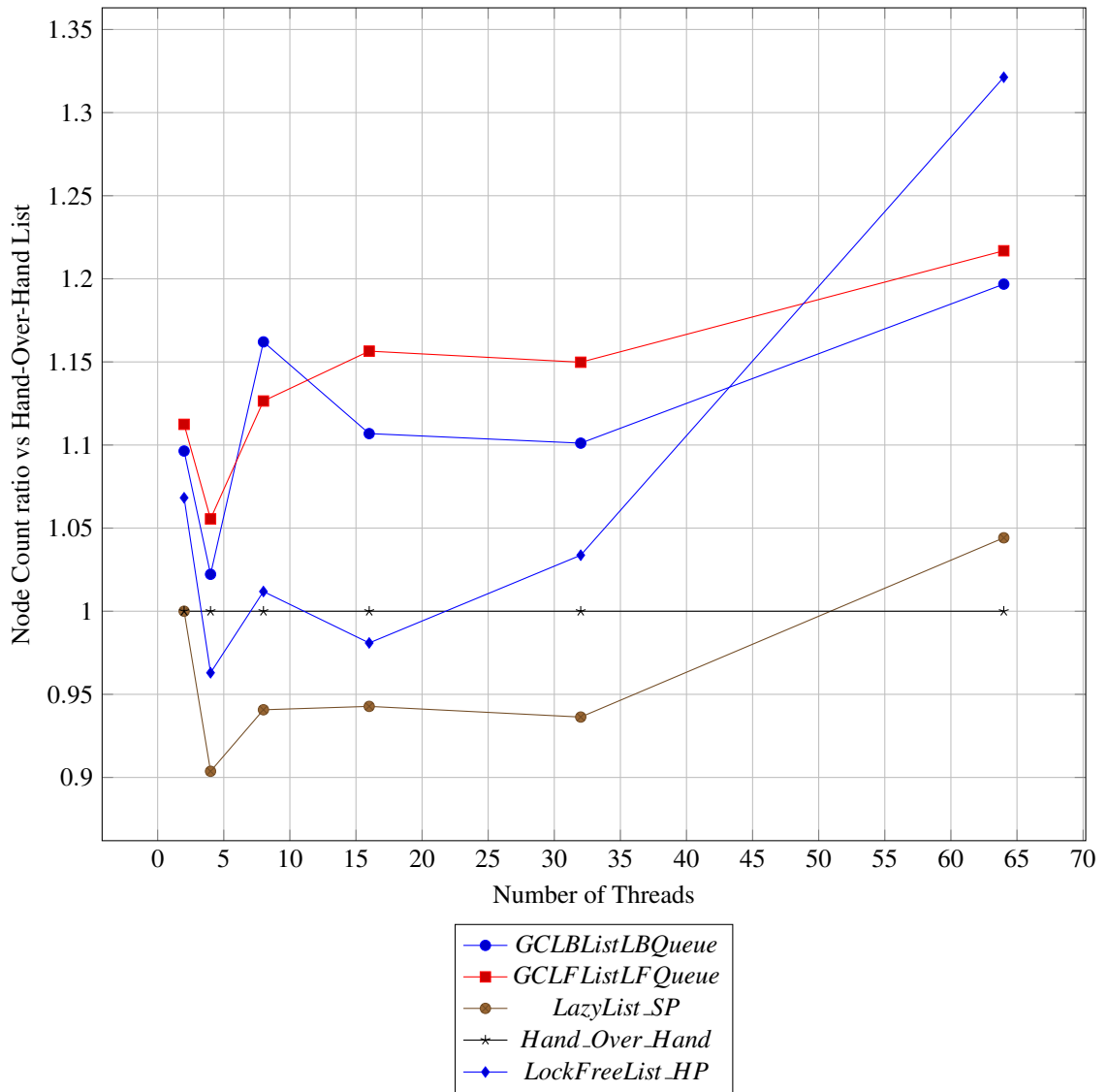


Figure 6.6: Memory Consumption Analysis with 70% writes

## 6.3 Analysis of Results

### Performance Analysis

From the graphs, we can see that the performance of both versions of GCList i.e GCLBList and GCLFList, is at par or even better than Harris's LockFreeList. Both outperform the Hand-over-Hand List, the LazyList based on Shared Pointers and the LockFreeList using Hazard Pointers for memory reclamation by multiple folds. The GCList versions are only outperformed by the original LazyList.

### Memory Consumption Analysis

However, in terms of Memory consumption, both versions of GCList consume a lot less memory than the original LazyList. It also needs less memory than Harris's LockFreeList and the Hand-over-Hand List. In comparison with generic techniques like Shared Pointers and Hazard Pointers, memory consumption of GCList is still comparable to both.

The plots for LazyList and LockFreeList have not been shown in the graphs. This is because they consume way too much memory compared to the other lists. Adding the plots for LazyList and LockFreeList reduces the other plots to straight lines similar to the Hand-over-Hand plot. This is due to the fact that LazyList and LockFreeList are unable to either free deleted nodes or reuse them. For each insert operation, new memory has to be allocated for the node.

# Chapter 7

## Conclusion

### 7.1 Conclusion and Future Work

In this thesis, we have presented **GCList**, a linked-list representation of a concurrent set, with in-built garbage collection. Both the lock-based and lock-free versions of **GCList**, i.e. **GCLBList** and **GCLFList**, are introduced.

Our results show that **GCList** matches or outperforms most of the existing representations of a concurrent set, while consuming a lot lesser memory than the higher-performing algorithms like LazyList. Memory consumption was at par with generic garbage collection facilities like Shared Pointers and Hazard Pointers, while outperforming them many folds.

In future work, we plan to investigate whether we can extend it to other data structures similar to a concurrent list or using it as a part of it's structure e.g. SkipList, Hash Tables etc.

# References

- [1] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. III, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. *OPODIS'05 Proceedings of the 9th international conference on Principles of Distributed Systems* 3–16.
- [2] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica* 9, (1977) 1–21.
- [3] H. T.L. A Pragmatic Implementation of Non-blocking Linked-lists. In Proceedings of the Symposium on Distributed Computing (DISC). Berlin, Heidelberg, 2001 .
- [4] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *Proc. of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* 267–275.
- [5] Shared Pointers. 1. [http://www.cplusplus.com/reference/memory/shared\\_ptr/](http://www.cplusplus.com/reference/memory/shared_ptr/).
- [6] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on* 15 491–504.
- [7] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, (1990) 463–492.
- [8] M. Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, (1991) 124–149.
- [9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. 1st edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.
- [10] AtomicMarkableReference Java SE. <https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/atomic/AtomicMarkableReference.html>.
- [11] A. Gidenstam, M. Papatriantafyllou, H. Sundell, and P. Tsigas. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 20, (2009) 1173–1187.
- [12] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele. Lock-free reference counting. In Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC). New York, 2001 190–199.
- [13] D. Alistarh, W. Mitchell Leiserson, A. Matveev, and N. N. Shavit. ThreadScan: Automatic and Scalable Memory Reclamation .
- [14] AtomicStampedReference Java SE. <https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/atomic/AtomicStampedReference.html>.