UNLV | University Libraries
University of Nevada, Las Vegas

UNLV Theses, Dissertations, Professional Papers, and Capstones

5-1-2015

Concurrent Non-blocking Skip List Using Multi-word Compare and Swap Operation

Anish Ratna Tuladhar
University of Nevada, Las Vegas, anishrtuladhar@gmail.com

Follow this and additional works at: https://digitalscholarship.unlv.edu/thesesdissertations

 Part of the Computer Sciences Commons

Repository Citation

Tuladhar, Anish Ratna, "Concurrent Non-blocking Skip List Using Multi-word Compare and Swap Operation" (2015). *UNLV Theses, Dissertations, Professional Papers, and Capstones*. 2438.
https://digitalscholarship.unlv.edu/thesesdissertations/2438

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Theses, Dissertations, Professional Papers, and Capstones by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

CONCURRENT NON-BLOCKING SKIP LIST USING MULTI-WORD COMPARE AND SWAP OPERATION

by

Anish Ratna Tuladhar

Bachelor of Computer Engineering
Tribhuvan University
Kantipur Engineering College, Nepal
2009

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science Degree in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2015

Copyright by Anish Ratna Tuladhar, 2015
All Rights Reserved

UNLV | GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Anish Ratna Tuladhar

entitled

Concurrent Non-blocking Skip List Using Multi-word Compare and Swap Operation

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Department of Computer Science

Ajoy K Datta, Ph.D., Committee Chair

John Minor, Ph.D., Committee Member

Lawrence L. Larmore, Ph.D., Committee Member

Venkatesan Muthukumar, Ph.D., Graduate College Representative

Kathryn Hausbeck Korgan, Ph.D., Interim Dean of the Graduate College

May 2015

Abstract

We present a non-blocking lock-free implementation of skip list data structure using multi word compare and swap (CASN) operation. This operation is designed to work on arbitrary number of memory locations as a single atomic step. We discuss the implementation details of CASN operation which only utilizes the single word compare and swap atomic primitive found in most of the contemporary multiprocessor systems. Using this operation, we first design lock-free algorithms to implement various operations on linked list data structure, then extend it to design skip lists. Skip list is a probabilistic data structure composed of linked lists stacked together forming different levels. It provides expected logarithmic time search like balanced search trees, but without requiring rebalancing. The fundamental operations on a skip list data structure require traversing and updating a number of memory locations. Due to this nature of the data structure, using a powerful atomic primitive like CASN in its implementation simplifies the design and makes the concurrent reasoning easier. In addition to fundamental operations, we present a variety of other operations on linked list and skip list data structures and provide examples to support the correctness of the proposed algorithms.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Dr. *Ajoy K Datta* for his continuous support on my thesis. I thank him for his humble behavior, patience, motivation and enthusiasm. His guidance and immense knowledge helped me during the period of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my thesis.

I would like to thank *Dr. Lawrence L. Larmore*, *Dr. John Minor* and *Dr. Venkatesan Muthukumar*, for being part of the committee and providing their insightful comments and encouragement. My sincere gratitude goes to my parents, my sister *Pritama Tuladhar* and my beloved girlfriend *Brizika Rai* as they have always inspired and supported me on each and every aspect of my life.

At last, I would like to thank all my friends, seniors and juniors who have made my stay at UNLV a memorable one. I thank you for your wonderful company.

Anish Ratna Tuladhar

University of Nevada, Las Vegas
May 2015

Contents

Abstract iii

Acknowledgements iv

Contents v

List of Figures vii

List of Algorithms viii

1 Introduction 1

1.1 Motivation . . . 3

1.2 Objective . . . 4

1.3 Outline . . . 5

2 Background 7

2.1 Shared Memory and Synchronization . . . 7

2.1.1 Multi processors and Multi threaded systems . . . 8

2.1.2 Atomic primitives and ABA Problem . . . 9

2.2 Behavior of Concurrent Objects . . . 11

2.2.1 Designing Concurrent data structures . . . 12

3 Literature Review 14

3.1 Universal Transformations . . . 14

3.2 Atomic Operations . . . 15

3.2.1 Single word compare and swap . . . 15

3.2.2 Double word compare and swap . . . 16

3.2.3 Multi-word compare and swap . . . 16

3.3 Concurrent Data Structures . . . 17

3.3.1 Linked Lists . . . 18
3.3.2 Balanced search trees . . . 20
3.3.3 Skip Lists . . . 21

4 Understanding CASN Operation **25**
4.1 Implementing SubtractX Operation . . . 26
4.2 Illustration of CASN Operation . . . 29
4.2.1 Construction of Restricted Double Compare and Single Swap Operation . . . 30
4.2.2 Construction of CASN Operation . . . 33

5 Proposed Linked List and Skip List Implementation **37**
5.1 Proposed Linked List Algorithms . . . 38
5.1.1 Adding a Node in a Linked List . . . 39
5.1.2 Deleting a Node in a Linked List . . . 42
5.1.3 Searching a Node in a Linked List . . . 44
5.1.4 Prepending a Node in a Linked List . . . 45
5.1.5 Appeding a Node in a Linked List . . . 46
5.1.6 Proof of Concurrency . . . 48
5.2 Proposed Skip List Algorithms . . . 53
5.2.1 Adding a Node in a Skip List . . . 54
5.2.2 Deleting a Node in a Skip List . . . 58
5.2.3 Searching a Node in a Skip List . . . 61
5.2.4 Prepending a Node in a Skip List . . . 62
5.2.5 Appending a Node in a Skip List . . . 63
5.2.6 Proof of Concurrency . . . 65

6 Conclusion and Future Work **68**

Bibliography **70**

Curriculum Vitae **72**

List of Figures

1.1 A sorted singly linked list with three nodes. 3
1.2 A simple skip list with five nodes. Each linked list serve as a level for skip list. . . . 3

2.1 UMA and NUMA multiprocessor architectures. 9

3.1 A binary search tree and its worst case scenario. 21

4.1 A RDCSS Descriptor. 32
4.2 A CASN Descriptor. 33

5.1 Adding a node in a linked list. 40
5.2 Deleting a node in a linked list. 43
5.3 Two concurrent threads A and B operating together. Thread A is trying to delete an item from the list and thread B is trying to add an item. 49
5.4 Two concurrent threads A and B operating together. Thread A and B trying to delete adjacent items at the same time. 50
5.5 Adding a node in skip list. 55
5.6 Deleting a node in skip list. 59
5.7 Searching a node in skip list. The search starts from the maxlevel and descends down each level until the sought after key is found. 61
5.8 Two concurrent threads operating together simultaneously in skip list. Both threads are trying to add an item in the skip list simultaneously. 66
5.9 Two concurrent threads A and B operating together simultaneously. Thread A is trying to delete an item from the list and thread B is trying to add an item at the same time. 67

List of Algorithms

3.1 A basic syntax of CAS Operation 16
3.2 A basic syntax of DCAS Operation 16
4.1 A basic syntax of CASN Operation 25
4.2 Pseudo code to implement subtractX and readX sequentially 27
4.3 Pseudo code to implement substractX and readX with descriptors 28
4.4 Pseudo code of RDCSS operation 31
4.5 Pseudo code of CASN Operation 34
4.6 Pseudo code of CASNRead operation 35
5.1 Pseudo code to implement 'add' operation in a linked list. 41
5.2 The linked list 'find' operation: a helper function used by add and delete operations. . 42
5.3 Pseudo code to implement 'delete' operation in a linked list. 44
5.4 Pseudo code to implement 'search' operation in a linked list. 45
5.5 Pseudo code to implement 'prepend' operation in a linked list. 46
5.6 Pseudo code to implement 'append' operation in a linked list. 47
5.7 Two designs to generate a random level. The level thus generated can range from minimum value 1 through maxlevel 54
5.8 Pseudo code to implement 'add' operation in a skip list. 57
5.9 The skip list 'find' operation: a helper function used by add and delete operations. . . 58
5.10 Pseudo code to implement 'delete' operation in a skip list. 60
5.11 Pseudo code to implement 'search' operation in a skip list. 62
5.12 Pseudo code to implement 'prepend' operation in a skip list. 63
5.13 Pseudo code to implement 'append' operation in a skip list. 64

Chapter 1

Introduction

With the advancement of technology, single core processors are fading away. Modern technology is getting faster and faster every day. Processors were initially developed with only one core. Manufacturers are now concerned on the multicore architectures which allow high degree of parallelism. Multicore processor is a single component with two or more independent processing units that reads and executes program instructions. Multicore processors being able to carry out multiple instructions at the same time, they work together in parallel to execute a single task, thus making computing faster.

For the processors to become faster, as predicted by Moore, the number of transistors in a dense integrated circuit doubles approximately every two years. But the speed up is not achieved without overheating of the transistors. Although there is an exponential increase in transistor count, the clock speed is flattening sharply. So, the manufacturers are now more concerned on building efficiency on software rather than making the most of the hardware. Multiprocessor chips makes computing more effective by enforcing a high degree of parallelism between the number of processes working on the same shared memory.

With this major switch to enhance the software, existing algorithms on the sequential data structures needs to be revisited for them to work on multi-threaded environment. To make use of the multi-core processors and achieve highest degree of parallelism, the data structures are in of improvement in the implementation of the algorithms they operate on. The simplest and traditional way to maintain concurrency between a number of threads is by using locks. Data structures implemented using locks are also called blocking implementation of that data structure. But, they have prominent drawbacks and are discussed in section 2.2.1.

An alternative to blocking implementations are non-blocking implementations where the failure or suspension of any thread cannot cause failure or suspension of another thread. They are guaranteed to make progress system-wide or per-thread. In this research, we will concentrate on designing lock-free implementations of data structures and operations needed to implement them. Another strongest non-blocking guarantee of progress is the property of wait-freedom. We discuss all these properties in section 2.2.1. Much of the research are done on concurrent data structures. Few of them being binary search trees, AVL-trees, hash, stack and queues.

With few exceptions, non-blocking algorithms use atomic read-modify-write primitives. These primitives are anticipated to be made available by the hardware. The most distinguished of which is compare and swap (CAS) operation. Using these primitives, standard interfaces are designed to implement critical sections. CAS however operates on single memory location. We are concerned with the design of multi-word compare and swap operation which operates on arbitrary number of memory locations in a non-blocking manner. We explore various non-blocking implementations of this operation and present algorithms to implement it elaborating the operation with various examples and details. The atomic primitive thus designed will be used in implementing various non-blocking algorithms for linked list and skip list data structures.

Linked list is one of the basic data structures which consists of a group of nodes aligned together representing a sequence. Each node has a key field to store the element and a next pointer which will store the reference to the next item on the linked list. Each node in a linked list has two neighbor nodes, one preceding it and one succeeding it. The first and last nodes are called head and tail and are sentinel nodes. First non-blocking lock free implementation of linked list was presented by Valois [11]. The design used traditional compare and swap operation. To implement it as a concurrent non-blocking data structure, we will use CASN operation. Along with fundamental operations, lock-free append and prepend operations will be introduced. The linked list thus designed when stacked together into different levels, form a data structure called skip lists which was invented by Pugh [16, 20, 21]. An example of linked list and skip list data structure is given in figure 1.1 and 1.2. Skip lists designed with this structure can serve as an alternative to balanced search trees. Unlike balanced search trees, they use probabilistic approach to maintain the balance which makes the implementation and concurrent reasoning easier for skip lists. This offers a path to carry out experiments and implementations on this data structure. Neglecting the worst case scenarios, skip lists offers logarithmic time complexities for the fundamental operations like balanced search trees.

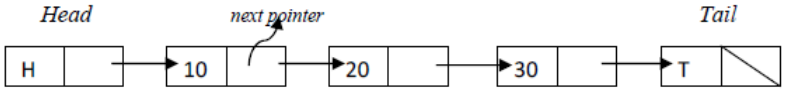

Figure 1.1: A sorted singly linked list with three nodes.

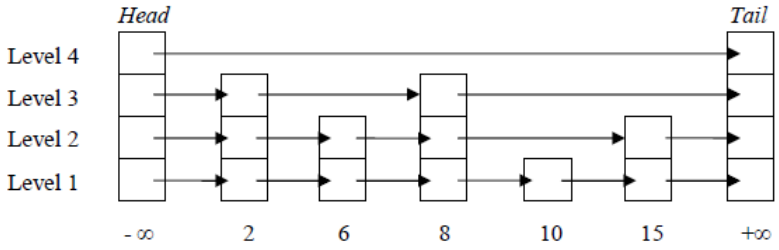

Figure 1.2: A simple skip list with five nodes. Each linked list serve as a level for skip list.

1.1 Motivation

To take the advantage of the improvement on the performance gain by using the multi-core processors, a software compatible to multiprocessor hardware needs to be designed. Exploiting the hardware and obtaining gain by increasing the clock speed is not the solution in modern computer science. Technology is now concerned on increasing parallelism in multi-core environments and not on increasing clock speed. We need to design programs that exploit multi-core processors which results increase in performance. Concurrent programs on the other hand needs to be supported by concurrent data structures. Concurrent data structures can be accessed by multiple threads which may access the data simultaneously because they run on different processors that communicate with one another via a shared memory. It is usually easier to design algorithms for a data structure in a sequential manner. Designing similar data structure to work in a concurrent environment requires a lot of other invariants to be preserved - the safety and liveness properties.

Balanced search trees are designed for efficient search, insertion and deletion provided with logarithmic performance. The operations on these tree structures require rebalancing which is very complex to implement and makes optimization difficult. An alternative to such a data structure would be skip lists which is simpler than balanced search trees both in concept and implementation

[20]. Skip lists do not require rebalancing and use probabilistic approach to maintain the balance. Under the assumption that the input sequence will not consistently produce worst-case performance, implementing skip lists can be easier and faster than balanced trees. The expected average cost of search, insertion and deletion is O(log n). Although skip lists have bad worst case performance than balanced trees, the probability that the input sequence consistently produces a worst case performance is very minimal.

In the construction of skip list data structure, a much simpler and most common data structure, a linked list is used. It is a dynamic data structure. The main advantage of using a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal. These linked lists serve as each level of the skip list data structure. Designing concurrent algorithms for linked lists and skip lists may be challenging. But with dominant CASN operation in hand, we can design a simpler non-blocking implementation of linked lists and skip lists.

1.2 Objective

In concurrent programming, an operation is said to be atomic if it appears to occur instantaneously to rest of the system. The property of a process to work independently without the interference of other concurrent operations is called mutual exclusion. Mutual exclusion refers to the requirement of ensuring that no two concurrent processes are in their critical section at the same time. To ensure the atomicity of the operation, the standard way to approach mutual exclusion is using locks. The main disadvantage of using locks for mutual exclusion is they cause blocking. Some processes might have to wait until a lock is released. If one of the threads holding a lock dies, blocks or goes into any sort of infinite loop, other threads waiting for that lock may wait forever. Problems like lock contention, priority inversion and deadlock can occur. Alternatives to locking include non-blocking synchronization methods, like lock-free and wait-free programming techniques and transactional memory [25]. Designing generalized lock-free algorithms is a hard task. So, we will focus on understanding and designing lock-free data structures instead.

Lock-free data structures are often implemented in terms of simpler atomic primitives like compare and swap (CAS). With the existence of universal primitives such as compare and swap, Herlihy

[26] showed this primitive is necessary and sufficient to construct any non-blocking object. With this generalization, we will use compare and swap to construct a powerful atomic primitive multi word compare and swap (CASN). CAS works on a single memory location. Various researches have been conducted to find the efficient algorithm and propose the lock free and wait free designs for implementing multi word compare and swap operation (CASN) [1, 5, 6, 7, 14]. Our designs are highly influenced by the ones described in [2, 14]. We present the design of this operation using simpler examples and will use it to implement lock-free linked lists and skip lists. The aim of this thesis is to simplify the understanding of non-blocking linked list and skip list data structures in concurrent environment with the help of CASN operation. In addition to that, we will design some novel operations in the skip list data structure and implement optimized algorithms which will enhance the searching of nodes.

To begin with understanding of the implementation, comprehensive knowledge of working mechanism of CASN operation is crucial. The working mechanism of this operation is not as simple as single word compare and swap or double word compare and swap. The algorithm we will go after is complex and the design is intricate. The key to the implementation is the use of descriptor data structure. We will begin with the understanding the basics of CASN operation, its requirements and data structures involved in constructing this powerful operation. Various fundamental and other operations on linked lists will be created using CASN. Skip lists, although not a tree, are a popular alternative which share the same performance characteristics like balanced search trees while being easier to implement and understand. Skip lists have their own unique behavior and are linked lists stacked together. We will present a detailed implementation of skip lists using the linked lists.

1.3 Outline

In chapter 1, we provided the brief introduction to the area of research. We discussed the need and motive to choose the particular area as the topic of research.

In chapter 2, we will discuss the importance of synchronization and various techniques available to enforce it in shared memory multiprocessor systems. We will discuss the power of atomic primitives offered by the multi-core systems and their role in synchronization along with the problems associated with it. We will provide details on the correctness and progress conditions required for concurrent objects. Also, we will discuss on the various techniques to design concurrent data structures - blocking and non-blocking.

In chapter 3, we will go briefly through all the ideas people have came forward for implementing concurrent data structures. We will discuss atomic primitives which are essential for the research area and discuss various approaches that has been implemented for designing these atomic primitives. We will outline the research performed on linked lists, balanced search trees and skip list concurrent data structures which have highly influenced this research topic itself.

In chapter 4, we will discuss the non blocking implementation of multi word compare and swap atomic operation which we will need to implement lock-free implementation of linked lists and skip lists, referencing the design from Harris, Fraser and Pratt [8]. We will provide detailed examples to aid the understanding of this powerful operation. We will discuss the algorithms on how a restricted form of DCAS will be used to implement generalized CASN operation.

In chapter 5, we discuss on the various operations we will be implementing for linked lists and skip lists data structures. We will first propose lock-free algorithms to implement the fundamental operations along with some novel operations like append and prepend. We will discuss how CASN operation will help on gaining the non-blocking behavior. We will provide few examples to prove the correctness of the algorithms. We will then extend the implementation to design skip lists and provide the proof of correctness in some sample concurrent execution scenarios.

In chapter 6, we conclude our research topic and provide ideas and suggestions to extend the area as a future work.

Chapter 2

Background

The computer industry is undergoing a paradigm transfer. Chip manufacturers are changing development resources away from single processor chips to a new generation of multi-processor chips known as multi-cores. At the beginning, a computer system came up with a single processing unit that was used to execute computer tasks. All the operations were carried out sequentially on this unit. This is called a uniprocessor system. With the advancement of technology, the uniprocessor systems are vanishing away and multiprocessor systems have occupied the area. Multiprocessor systems are able to support more than one process at the same time. With this evolution in parallel hardware, the problem of allocation of resources to the multiple processes in operation and making effective use of the hardware have came up.

Also, increasing clock speed to improved performance is no longer an option. Moores law has shifted to mean that each generation of processors will provide more cores, leaving clock speed essentially flat [2]. This has led technology to obtain parallelism in multi-core environment rather than exploiting hardware. This fundamental change in the computing architecture require fundamental alteration on the way a computer program is written with multiple processors in hand. Concurrency and synchronization are fundamental issues that are critical for the design of concurrent programs. The future of multiprocessor computers will rely on how well programmers can take advantage of concurrency offered by such hardware.

2.1 Shared Memory and Synchronization

Shared memory is a method by which processes can exchange data more quickly than by reading and writing using the regular operating system services. Shared memory is an efficient means of

passing data between programs. Concurrent processes in the multiprocessor architecture work on the globally shared memory. Processes operate concurrently and share a common resource; a critical section. The problem with concurrent processes operating together in a shared memory is that they may interleave in arbitrary ways which may lead to incorrect results.

Synchronization is the art of managing those interleavings such that they are impossible to occur [9]. It is a mechanism which ensures that the concurrent processes or threads do not simultaneously use the same shared memory. This is also known as mutual exclusion [2]. Failure to apply proper synchronization techniques might result in race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads. In context of concurrent algorithm designing, they require the programmer to handle situations that simply don't occur when developing single-threaded, sequential algorithms. For instance, when two or more threads try to access and modify a shared resource (race conditions), the design should not leave the system in an inconsistent or deadlock state. Synchronization is needed in all systems and environments where several processors can be active at the same time. Without proper synchronization, the integrity of the data may be destroyed.

2.1.1 Multi processors and Multi threaded systems

Both multi-threaded and multi-processor (multi-core) systems exploit concurrency. Multi-threaded system is an execution model that allows a single process to have multiple threads to run concurrently within the perspective of that process. To facilitate multi-threading, processors have hardware support to efficiently execute multiple processes or threads. Whereas multiprocessor systems refers to executing multiple processes at the same time and share the resources of a single core: the computing units, the CPU caches. It refers to the hardware (i.e., the CPU units) rather than the software (i.e., running processes). They are independent (but complementary) design decisions.

The correctness and performance of synchronization algorithms depend crucially on architectural details of multi-core and multiprocessor machines. In a shared memory architecture, there are many possible configurations of processors, cores, caches and memories [9]. It refers to a multiprocessing design where several processors access globally shared memory. Two of the possible architectures are uniform memory access (UMA) and non-uniform memory access (NUMA) shown in figure 2.1.

In UMA architecture, all processors share the physical memory uniformly. From processor p0, the latency time to memory m0 and m2 is same. If placement of data is unimportant, this architecture is

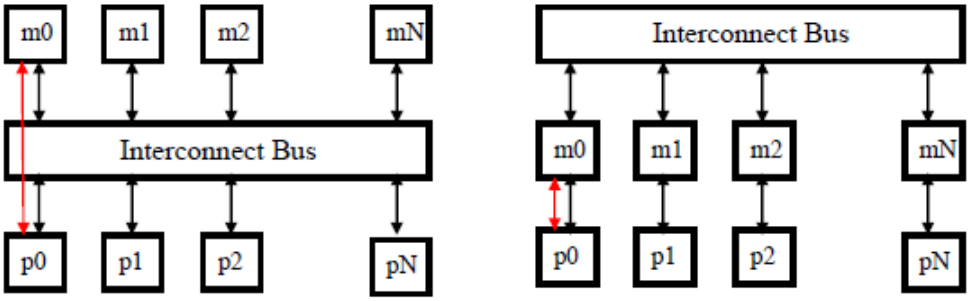

Figure 2.1: UMA and NUMA multiprocessor architectures. Latency is long from processor to memory in UMA and short in NUMA.

easier to implement. But as the system grows, the latency time will get longer. So, this architecture is typically is suited for small machines for the sake of simplicity [9]. The physical distances in larger machines hence motivate to switch to NUMA architecture where the latency time to local memory is shorter and helps in performance. In NUMA architecture, memory access time depends on the memory location relative to a processor. From the processor p0, the latency time to local memory m0 is very short but is long for m2. This architecture is typically used in larger multiprocessors.

2.1.2 Atomic primitives and ABA Problem

To facilitate the construction of synchronization algorithms and concurrent data structures, most of the modern multiprocessor architectures provide us with at least one of the atomic primitives which is capable of reading and writing a memory location as a single atomic operation [9]. The simplest idea to achieve synchronization between the threads is using locks to ensure the mutual exclusion. Synchronization primitives such as mutexes, semaphores, and critical sections are few techniques of achieving it. A thread trying to access the critical section will acquire the lock and release it after it is done. Locks make sure that that no other threads will intervene in the critical section of the thread acquiring the lock. This is also called blocking synchronization. Using locks to ensure synchronization between processes may look straightforward but care must be taken to avoid unexpected effects caused by multiple processes trying to acquire exclusive access to shared resources. One obvious disadvantage of using locks is that a thread acquiring lock will block the shared memory for all other threads until it releases the lock. It may cause other problems like deadlock, priority inversion and long delays [2, 27].

The solution to avoid locking is to make use of the atomic primitives offered by the hardware itself to ensure synchronization. Using these atomic primitives, an execution of a concurrent process can be made linearizable. Atomic registers are supported by most of the contemporary multiprocessor systems which support atomic reads and writes operations. A weaker idea of safe registers [2] where a guarantee that a read not concurrent with writes obtain a correct value is implemented too. However, a read that is concurrent with write may return arbitrary value in safe registers. In addition to reads and writes, most modern architectures support some strong form of atomic primitives like compare and swap, test and set, fetch and increment, fetch and add, load linked/store conditional, etc. [9]. If not, we can create atomic instruction like this easily using the atomic instructions offered by the system. Detail discussion on the operations with conditional atomic modification of memory location - CAS, DCAS and CASN is provided in section 3.2.

Atomic operation like compare and swap is sufficient to avoid blocking behavior in algorithms and avoid the use of locks. Herlihy [3, 26] proved that the compare and swap operation is universal and can be implemented to design lock-free algorithms. He proved that CAS can implement more of these lock-free algorithms than instructions like atomic read, write, or fetch-and-add, and assuming a fairly large amount of memory, that it can implement all of them. He proved that is impossible to design non-blocking or wait-free implementations of many simple data structures using other well known synchronization primitives i.e. read, write, test and set, fetch and add and swap [3].

However, the compare and swap operation comes inherently with a package of difficulty which needs to be addressed in designing concurrent non-blocking algorithms - the ABA problem. It is possible that between the time the old value is read and the time CAS is attempted, some other processes or threads change the memory location two or more times such that it acquires a bit pattern which matches the old value. The value in the memory location seems to be as expected but actually it has changed. Below is an example of the possible sequence of events that will result in the ABA problem:

- Process *P1* reads value A from shared memory.
- *P1* is preempted, allowing process *P2* to run.
- *P2* modifies the shared memory value A to value B and back to A before preemption.
- *P1* begins execution again, sees that the shared memory value has not changed and continues.

This may cause undesirable situations [4, 9]. It is a fundamental problem in CAS based designs and there are many solutions. The problem is often solved by adding a time-stamp to the variable

that is increased every time it is changed. This lowers the probability that the ABA-problem will occur, but it also lowers the amount of information that can be stored in the variable. Using LL/SC pair of atomic primitives, if available will also avoid ABA problem [4]. Another solution is using version tags [28] which also requires system to support DCAS primitive.

2.2 Behavior of Concurrent Objects

A concurrent object is a data structure shared by concurrent processes. The behavior of concurrent objects are best described through their safety and liveness properties, often referred as correctness and progress [2, 9]. A concurrent object provides a finite set of operations which are the only means to modify the concurrent data structure. Since a concurrent operation can be invoked by many processes simultaneously, the question of how to provide the perception of correctness and progress for a system arises.

To begin the reasoning and understanding of the correctness properties, let us review some formal terms first. An execution of a concurrent system is modeled by a history, a finite sequence of method invocation and response events. A sequential history is a set of events where method calls do not overlap. The first event is an invocation, and for each invocations, except possibly the last, there is a matching response. An object or thread sub-history is the subsequence of events in history relating to that thread or object. A sequential specification for an object is a set of sequential histories for that object. A sequential history is legal if each object sub-history is legal for that object. Now, let us discuss some ideas to develop the intuition regarding the specification of correctness in a system - quiescent consistency, sequential consistency and linerizability.

- *Lineraizability*: An execution in a concurrent system is linearizable if it is equivalent to the legal sequential execution. Every concurrent history is equivalent to some sequential history. An object is linearizable if it appears to the rest of the system to occur instantaneously. It is done by identifying a linearization point where the operation takes effect. Simply, a straightforward approach in lock based system to define linearization point is to use the critical section.

- *Sequential consistency*: An execution in a concurrent system is sequentially consistent if each method calls in the history appear as they occurred in a sequential order consistent with program order. In any concurrent executions, there is a way to order the method calls sequentially. In most of modern multiprocessor architectures, memory reads and writes are not sequentially consistent. So, to ensure that reads and writes interact correctly, special instruction like memory barriers or fences are provided.

- *Quiescent Consistency*: An execution in a concurrent system is quiescent consistent if at any time a concurrent object becomes quiescent (a period of time where no method is being called by any thread), then the execution so far is equivalent to some sequential execution of the completed method calls. This consistency condition is stronger than sequential consistency, but is still not strong enough that we usually expect from a multiprocessor system.

Safety properties defined above ensure that bad things never happen and liveness properties ensure that good things will eventually happen. In addition to ensuring the correctness of an algorithm, we will also want the algorithm to make forward progress because in multiprocessor systems, unexpected thread delays are common. An implementation of an algorithm is called blocking if there is some reachable state in the system in which the thread that has called the method of the algorithm blocks another thread. Lock-based algorithms are therefore inherently blocking. Liveness properties for lock-based systems require the implementation to be deadlock free, critical sections to be free of infinite loops and all threads will continue to execute. An algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread. Some progress conditions are:

- *Deadlock freedom*: An implementation is deadlock free if some thread trying to acquire the lock eventually succeeds.
- *Starvation-freedom*: An implementation is starvation free if every thread trying to acquire the lock eventually succeeds.
- *Lock-freedom*: An implementation is lock free if some threads are executed sufficiently for long time, at least one of the threads makes progress.
- *Wait-freedom*: An implementation is lock free if every thread executing the algorithm operations succeeds.
- *Obstruction-freedom*: An implementation is obstruction-free if at any point, a single thread executed in isolation for a bounded number of steps will complete its operation. All lock-free algorithms are obstruction-free.

2.2.1 Designing Concurrent data structures

A concurrent data structure is a particular way of storing and organizing data for access by multiple computing threads on a multiprocessor system. Shared memory multiprocessor systems allow multiple processes to work on same memory locations concurrently. So, designing concurrent data

structures is significantly more difficult than designing the sequential implementation of the same data structure [2, 29]. Designing concurrent data structures for multiprocessor systems also provides numerous challenges with respect to performance and scalability. There are blocking and non-blocking alternatives to implement concurrent data structures - non-blocking being harder to implement and reason about the proof of correctness. We will outline few blocking and non-blocking implementation strategies here:

- *Coarse-grained locking*: In this technique, on the sequential implementation of the data structure, we add a lock field and ensure that each operation acquires and releases the lock. It is like adding a huge single lock on the whole data. But, coarse grain locks are slow. We should use them cautiously and only when necessary. It also increases the chance of deadlocking if implemented incorrectly.

- *Fine-grained locking*: In this technique, instead of using single lock, we divide it into multiple locks. Each fine grained lock is responsible to protect the portion of data the operation is working on. This will ensure that the operations interfere with each other only when trying to access the same memory location at the same time. Fine-grained locking can improve the overall throughput of a concurrent system. However, we must consider to avoid deadlock, livelock, starvation, preemption, priority inversion, convoying, etc.

- *Optimistic locking*: Some data structures like trees, lists, skip lists consists of multiple components linked together by references. Some operations only search for a particular component. The idea here is to search without locking the whole component. The lock will only be placed upon the finding of the required data. This will reduce the cost of fine-grained locking. This technique is beneficial only if it succeeds more, so it is called optimistic method.

- *Non-blocking*: This technique avoids using locks and use atomic primitives like CAS to make algorithms lock-free or wait-free. Non-blocking methods do not involve mutual exclusion, and therefore do not suffer from the problems that blocking can cause.

- *Transactional memory*: Transactional Memory is a synchronization primitive which is a general form of the LL/SC operation. This technique allows a group of load and store instructions to be executed in an atomic way. Either the entire sequence of operations appear to occur atomically, or else the system state is unchanged. It is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing.

Chapter 3

Literature Review

In modern computing era, there has been a tremendous improvement in the hardware - building multi-core systems from the computer system with a single central processing unit. To achieve a gain in performance, there has been a lot of work in increasing the clock frequency. But recently it has become much harder to increase performance just by increasing the clock speed. As predicted by Moore, memory speed has not increased at the same rate and an already high clock frequency has led to much greater power requirements with associated excessive processor heating.

So, researchers are now more concerned in the software side rather than exploiting hardware. There is a need to translate the sequential algorithms to concurrent algorithms which are capable to work on the multi-core hardware provided by the manufacturers to obtain the ultimate performance gain. For these algorithms to execute correctly, efficient synchronization between processes are needed to ensure that the safety and liveness properties are preserved in the implementation. Currently, majority of the researchers are concerned in the non-blocking implementations of concurrent data structures. They have proposed various algorithms for shared data objects.

Modern computer architectures have support for atomic primitive instructions like compare and swap, test and set and fetch and add. Using these atomic primitives, non-blocking implementation of concurrent data structures can be achieved.

3.1 Universal Transformations

A universal construction [26] is a mechanical translation protocol that takes as input a sequential specification or algorithm (a specification of the desired behavior in the absence of concurrency),

transforms it, and outputs a provably equivalent non-blocking or wait-free concurrent algorithm. Since sequential algorithms are well understood and relatively easy to reason about, the conceptual burden on the programmer is lightened. At first this transformation does the copying of the entire object, making necessary changes to it and trying to replace the old object by CAS operation. With the support of atomic primitives like CAS and LL/SC, he showed that it is necessary and sufficient to construct any non-blocking object using these primitives [14, 26].

Herlihy [26] proposed a new transformation, the Large Object Protocol. This protocol takes a data structure made up of blocks connected by pointers. Only blocks which are modified, or contain pointers to modified blocks, need be copied. A parallel structure is constructed with pointers to both modified and unmodified blocks. When the root is updated by CAS, the new structure contains new copies of each modified block. Referencing his work, there are many implementations done to translate a sequential object into concurrent objects. The later approaches are more sophisticated than simply copying the data structure. They reduce the amount of copying, increase the level of parallelism, and lessen contention that arises due to multiple processes hammering on the same data-structure. We will discuss the implementation of multi word compare and swap operation which was the work done by Harris, Fraser and Pratt [8] using just compare and swap atomic primitive in chapter 4.

3.2 Atomic Operations

An operation acting on shared memory is atomic if it completes in a single step relative to other processes. Without this guarantees, lock-free programming would be impossible, since we can never let different processes manipulate a shared variable at the same time. We will now discuss few atomic operations which are used in the design of lock-free algorithms.

3.2.1 Single word compare and swap

Single word Compare and swap is an atomic operation that takes two arguments an expected value and an update value. It writes a value to a memory location only if its current register value is equal to a given expected value else the value is left unchanged. It has been justified that CAS hardware primitive is universal and thus can be used to implement any shared data structure in a non-blocking manner [3], in the sense that it can be used to implement other similar atomic instructions including multi word compare and swap. This hardware primitive will be used to design other operations described in other sections of this report. The basic syntax of the operation is given as:

Algorithm 3.1: A basic syntax of CAS Operation

```
int CAS(int &address, int old, int new){
    //atomic execution begin
    oldval_temp = &address;
    if(oldval_temp == oldval){
        &address = newval;
    }
    return oldval_temp;
    //atomic execution end
}
```

3.2.2 Double word compare and swap

We can outline the double word compare and swap operation as an extension to the CAS operation. It is also an atomic operation that takes two contiguous memory locations (not necessarily) and writes new values into them only if they match pre-supplied expected values.

Algorithm 3.2: A basic syntax of DCAS Operation

```
bool DCAS(int&address1, int &address2, int oldval1, int oldval2, int newval1, int newval2){
    //atomic execution begin
    if((&address1 == oldval1) && (&address2 == oldval2)) {
        &address1 = newval1;
        &address2 = newval2;
        return TRUE;
    }
    else{
        return FALSE;
    }
    //atomic execution end
}
```

3.2.3 Multi-word compare and swap

Multi word compare and swap operation (CASN) is an operation that takes a number of arguments: a series of memory locations and if all the memory locations the operation is trying to update contains the expected values, the operation updates the memory locations with a new set of values.

Otherwise, the values are not updated and old values are restored in all the memory locations. It is the generalization of CAS and DCAS operations defined above. This operation will be used in implementing linked lists and skip lists in chapter 5. Details on the procedure to implement this operation is provided in chapter 4. There have been various approaches proposed and implemented to integrate this operation and many research papers implementing the CASN operation have appeared in the literature [1, 5, 6, 7, 8, 17].

H. Sundell [5] have designed to work on the implementation of CASN operation which is wait-free and use the technique of greedy helping and grabbing. In the first phase of this algorithm, the thread attempts to place a reference to its descriptor object at as many of the addresses in its operation as it can. In the next phase, if another CASN operation holds some of these addresses needed for this operation, then one of the two operations will help the other process. Other atomic primitives like Fetch and add, Compare and swap and Swap are used to construct the algorithm for CASN in his design. To meet the algorithm wait free requirements, descriptors are used which keeps all necessary information about a CASN operation in progress, and rather than locking the whole word, it is performed by replacing the value with a pointer to the appropriate descriptor. As the descriptor keeps a single status variable that indicates the standing of the whole CASN operation in progress, it is possible to update the memory locations atomically using the primitives[5].

We will discuss in detail regarding the descriptor data structures in chapter 4. Recursive helping technique is used in [1, 7, 8] to implement the CASN operation. Also, there is a different approach introduced [6] where the CASN operation has the possibility to adapt its helping policy depending on the level of contention in a dynamic fashion. Our designs are highly influenced by the ones implemented in [16, 20, 21, 8, 17] which only utilizes the primitive CAS which can be found in most of the contemporary multiprocessor systems and analyze it in detail. Remember that this is the lock free non-blocking implementation of CASN operation. We will understand and describe in detail the CASN operation implemented by Harris, Fraser and Pratt [8] and use the operation to implement linked lists. Various researches have been conducted to find the efficient algorithm and propose the lock free and wait free designs [5, 6].

3.3 Concurrent Data Structures

Multiprocessor machines are replacing single core architectures which implies the need to design concurrent data structures. These are the data structures that can be accessed by multiple processes which may actually access the data simultaneously because they run on different processors that

communicate with one another. These data structures are designed to operate on shared memory systems. There are various data structures implemented to work in concurrent environment. Some of them are:

3.3.1 Linked Lists

A linked list is a dynamic data structure. Each element of a list is comprising of two items - the data and a reference to the next node. Details on linked lists are discussed in chapter 5. Linked lists are important data structures. They are not only a foundation concept, they are very important in implementing other data structures like skip lists. Various blocking and non-blocking techniques of implementing linked lists can be found in the literature [2, 11, 12, 13, 14, 17]. All the operations made on the linked list should be linearizable [2]. One way of simply ensuring this property is to use locks. Use of locks will make sure that no two processes will interfere in between because only one process will have the lock and will only release it after it is done. This clearly ensures that no conflicting additions or removal of nodes will meddle in between. Various techniques of implementing lock based linked list are provided in [2]. However, using locks to ensure linerizability, it is impossible for one process to make progress while the list is locked by another process. So, as a liveness property, it is essential that the operations are non-blocking. We will discuss and implement a lock free approach to ensure that deadlock problems introduced by mutual exclusion locks are solved. Various atomic primitives like compare and swap can be utilized to solve the problem. We will focus with the non-blocking implementation of linked lists using multi word compare and swap operation.

First lock free design of linked list was presented by Valois [11]. He used compare and swap atomic primitive to implement linked lists. He presented the use of auxiliary nodes to implement his design. Auxiliary nodes are the nodes in the linked list which only contained the next field. These nodes were to link the ordinary nodes and found between them. These nodes were used to connect the nodes together for the purpose of deletion and addition of nodes. According to his design, each ordinary node had this node as predecessor and successor nodes. The process of deletion left one extra auxiliary node behind so these extra nodes needed to be cleaned up (keep trying to) by the same process. He also presented the use of cursor pointers in his design. The use of back link field to traverse in the reverse direction was used for the deletion operation. This field was used to track back the node that has not been deleted from the list. The use of cursors was to identify the cell position while traversing the linked list. By the use of this cursor, nodes addition and deletion process would advance further. However, this sophisticated technique of design made this implementation highly intricate. Moreover, there are no proofs of linerizability provided for the basic operations.

Greenwald [31] suggested a different approach to design non-blocking implementation of linked lists. He used another atomic primitive DCAS to fulfill the requirements linearizable and non-blocking. He implemented a non-blocking priority queue as a linked list. The algorithm is simple and linearizable but uses DCAS. He also presented a non-blocking implementation of CASN operation that supports multi objects with the DCAS primitive. The main idea he used was using DCAS operation is to swing the references atomically while deleting a node from the link list. The DCAS operation will atomically update the next pointer of the deleted node and that of its predecessor using one instruction. DCAS however is not provided in most of the multiprocessor architectures. However, he does provide us with a simple and linearizable linked list algorithm.

Harris and Fraser [12, 14] presented various non-blocking approaches to implement abstract data structures. They presented three approaches for designing non-blocking implementations of arbitrary data structures. They discuss the MWCAS operation which we are going to use to implement our data structure. They constructed this operation using conditional compare and swap operation and simulated this operation in software as no hardware is able to support them. They compared the performance of several concurrent tree implementations. They found that the skip lists with locks are very fast and work well with concurrent accesses than other data structures. Lock-free skip lists are hardly reliably faster than lock ones. They also discuss the performance of other tree structures like red black trees. Various other abstractions are implemented and performance evaluations were done to justify the operations [14]. Harris [12] used compare and swap primitive to implement a lock free linked list. To delete a node, he used the concept of marked nodes to logically delete a node and then swing pointers to physically delete it from the list. If in between the predecessor and successor nodes, marked nodes are found during search, they are deleted at once. To justify the use of marked nodes, he used the concept of "bit stealing".

The difference in designs of [12] and [13] is that while searching for a desired node, all the marked nodes in the path are deleted in [13]. But, only the marked nodes in between the left and right nodes are deleted in the implementation of Harris [12].

Harris suggested various alternatives to operate the delete operation. When an address is aligned to a power of 2 then a number of low-order bits in its binary representation are zero. This unused bit of address (might not be) can be exploited and used to implement delete operation in linked lists [12, 13]. Both [12] and [13] use this unused bit to implement the delete operation. But, the bit

does not necessarily always remains unused. Harris [12] implemented his algorithm utilizing this bit found in the next field of a node. But, he suggested an alternative to this because of the realization that this bit can be used by the system for various other purposes. For example, the bit might not be available in the implementation environment because the addresses might not always remain ordered [12]. As an alternative to this, we can use an extra level of indirection in which marked references are accessed through dummy nodes. A node is marked or deleted logically if and only if it has a dummy node as its left node.

3.3.2 Balanced search trees

Balanced search trees are used for storing data in a way that supports fast retrieval, insertion and deletion operations. Each item in the data structure is represented by a node of a tree. The main problem with search trees is that they require balancing so that the searching of a node in the tree structure becomes faster. The height of the balanced search trees is logarithmic and the fundamental operations on the data structures requires O(log n) time. Various versions of balanced search tree data structures (hundreds of them) are present and each of them with their unique properties. Some of the popular tree structures are binary search trees, red black trees and 2-3 trees.

A binary tree [30] is a tree with exactly two sub-trees for each node, called the left and right sub-trees. For each node, the left sub-tree only has nodes with keys smaller than (according to some total order) the key, while the right sub-tree only has nodes with keys larger than that key. The height of a tree is the number of nodes on its longest branch. Operations like insertion and deleting item from binary search tree will violate the tree's balancing property which is to be maintained as invariant. So, balancing of the binary search tree is needed to ensure uniform distribution of data which will aid on efficient retrieval. Also, these re-balancing transformations should also take O(log n) time, so that the effort is worth it.

A binary search tree is a data structure designed for efficient search, insertion, and deletion in the presence of a large number of items. But if balancing is needed, these operations will break the balancing invariant of the binary search tree and the tree needs to be restructured. Also, balanced trees are very complex when it comes to optimization. An alternative to balanced trees is to use randomization to guarantee, with high probability, that a tree will not be so unbalanced as to lose its O(log n) performance properties. These randomized trees are much simpler than their balanced siblings and far safer than basic binary search trees. If an absolute guarantee of good performance is not necessary, a recursive implementation and a slim chance of poor performance are acceptable,

randomized trees such as skip lists will provide a good solution. Binary search trees (Figure 3.1) work well for many applications but they are limiting because of their bad worst-case performance O(n). A binary search tree with this worst-case structure is no more efficient than a regular linked list.

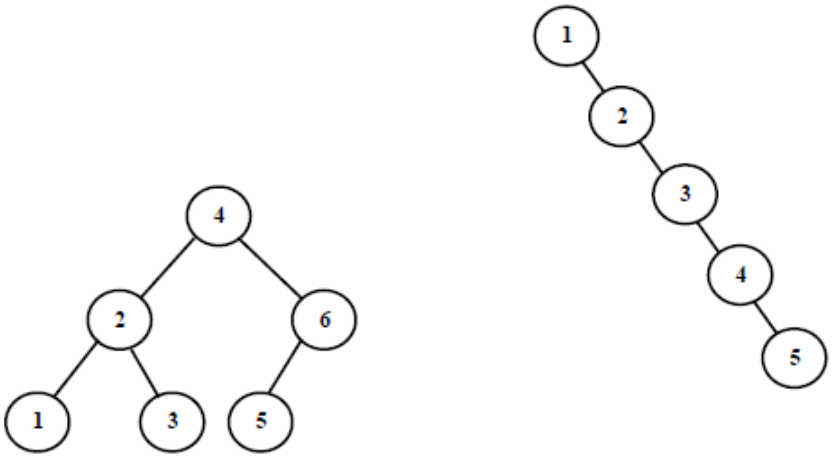

Figure 3.1: A binary search tree and its worst case scenario.

3.3.3 Skip Lists

Skip list is a probabilistic data structure which maintains a set of linked lists which are in sorted order in multiple levels.. The maximum level of the skip list will be fixed and is calculated in advance. Details on skip lists are discussed in chapter 5. Skip lists data structures were invented by Pugh [20, 21, 16] in early 1990s. It does not require any rebalancing like other popular search structures. In concurrent applications, rebalancing can be a bottleneck and hence lead to high contention between threads. As skip list do not need rebalancing, they can be a great alternative to balanced search trees [20, 22]. Since levels are probabilistically calculated, they do not need rebalancing and provide us with expected logarithmic search time [2, 13, 21, 23].

Pugh [16, 20, 21] was the first to give a notion of skip list data structure. He provided concurrent algorithms to implement fundamental operations in a skip list data structure [16] and used probabilistic balancing in skip lists to represent balanced trees. Skip lists are probabilistic data structure that can be used as a replacement for balanced trees [20]. Since skip lists are built of many levels of linked lists, searching for an item will be complicated. He discussed various solutions for the

problems encountered during the searching an element in skip list. As one of his solutions, in our algorithm, we will start searching for an item in skip list from the highest level. Pugh compared the performance of skip lists with other search trees like AVL trees and 2-3 trees and demonstrated that insertion and deletion times of skip list implementation is far more faster than that of other tree based structures. Although skip lists have bad worst case performance than these balanced trees, the probability that the input sequence consistently produces a worst case performance is very minimal. The expected cost of search, insertion and deletion is O(log n) [20, 21]. He also presented algorithms for inserting an item after Kth element in a list as linear list operations. Implementing these algorithms for skip list data structure, Pugh stated that skip lists may behave rivals to balanced search trees in case of versatility and are in most cases faster and simpler.

Fraser and Harris [17] discussed various practical non-blocking programming abstractions. They implemented multi word compare and swap using conditional compare and swap operation. They also designed a higher level of programming abstraction - software transactional memory and object based transactional memory called FSTM. They presented various implementation details on data structures like binary search trees, red-black trees and skip lists using CAS, multi word CAS and FSTM programming abstractions. In our implementation, we will use CASN operation discussed in chapter 4 to implement skip lists in a lock-free manner. We will use the similar concept as used in the linked lists to implement skip lists. We will follow the concept of Pugh [16, 21], the concept of back pointing to avoid the overhead of maintaining 'marked' nodes [2, 10, 18] which is used as bits of next field of the list node. We will refer to the work performed by [2, 17] to implement various operations on lock-free skip list.

Pugh [16] described the highly concurrent implementation of skip lists using locks. He presented concurrent implementation for sorted linked list, proved the correctness of the algorithm and then implemented skip lists using the same approach. We will use the same analogy to implement skip lists. Because skip lists are an alternative to balanced trees [20], implementing concurrent update algorithms for skip lists are much simpler than implementing the same for balanced trees. Also, the degree of concurrency is higher. Pugh presented concurrent algorithm using pointers pointing reverse direction - deleted nodes point their predecessors after being deleted to ensure that a search concurrently backtracks if it traverses in a path of deleted nodes. An important point to consider, he provided alternatives to make searching efficient. Because all the other operations like insert, delete, update rely on searching, the algorithm would be efficient if searching is at best cost. He proposed a solution that instead of searching for a key starting from maximum level, the number of

comparisons can be reduced if we search from the current maximum level, the maximum level of a node in the current instance of skip list. He used levelHint variable in his algorithms to achieve this [16]. We will use the same technique in our skip list implementation. We will not start from the maximum level always but from the current maximum level which will be stored in a shared variable 'curr_maxlevel'(details on section 5.2). Conducting various experiments and analysis, Pugh affirmed that concurrent skip list algorithms he described are at least as efficient as any possible concurrent balanced tree implementation.

Herlihy et al. [10, 18] presented a blocking implementation of skip list which was based on optimistic synchronization. In this technique, search operation does not acquire locks while traversing the list. Only after the target node is found, it is verified and locked. Clearly, this method will be efficient if there are large numbers of searches than other operations. It is very straightforward to reason about the correctness and understand every concurrent execution. Various techniques to make the algorithm efficient are proposed too. Moreover, this algorithm always maintains the skip list property: higher level lists are always contained in lower level lists [2]. The algorithm uses lazy synchronization technique to delete the marked node. This is done using the marked bit from the next pointers of the nodes. The add and delete operations use optimistic fine grained locking and contains operation is implemented wait-free. This implementation is very simple to understand and can be used as an alternative to lock free implementation of skip lists 'ConcurrentSkip listMap' in java platform, which was implemented by Doug Lea [19], based on the work by Fraser and Harris [17]. Experiments have been conducted implementing the algorithm in java and shown that this algorithm works as well as of Lea's [19] under most of the conditions. The most important advantage of this algorithm is it is very simple, yet effective.

Herlihy et al. [13] also provided a simple and effective approach to design non-blocking implementation of skip lists. This design is lock-free and is very simple and effective. Similar to lock based implementations [10, 18], this algorithm also provides logarithmic search [20] without the need to rebalance. However, this design might not preserve the skip list property because the design is lock-free and locks cannot be used [2]. The nodes at higher level lists only serve as a path for lower level list. We can understand this implementation as multi-level list implementation, which consequently is a skip list. An item is said to be present in the set if it is contained in the list at the lowest level. Also, similar to lock based optimistic implementation [18], the nodes will retain 'marked' flag. By using 'AtomicMarkableReference' object and its methods (compareAndSet, set, attemptMark, get, etc.) from the java concurrency package, they provided a lock-free implementa-

tion of skip lists. The need of this object is to make sure two instructions (checking the marked field and updating pointers) are executed atomically without any other concurrent threads interfering in between. The add and delete operations are designed lock-free and the contains operation as wait-free. So, the authors have presented two beautiful and very effective lock based and lock-free approaches to build skip lists [10, 13, 18]. We will use the similar idea introduced by the authors in implementing non-blocking skip lists using CASN operation.

Brian and Zachary [22] provided details on transforming skip lists to balanced search tree representation. This was done by converting the skip lists into equivalent multi-way branching search tree. They presented methods of transforming the fundamental skip lists operations into equivalent BST operations. They described the analogy between left and right rotation of a BST to raising and lowering of a section in a skip list [22].

Chapter 4

Understanding CASN Operation

Multi-word compare and swap operation (CASN) also popular as N-word compare and swap is an operation that takes a number of arguments: a series of memory locations and if all the memory locations the operation is trying to update contains the expected values, the operation updates the memory locations with a new set of values. Otherwise, the values are not updated and old values are restored in all the memory locations. The basic syntax of CASN operation is:

Algorithm 4.1: A basic syntax of CASN Operation

```
bool CASN(int &address1, int &address2, ....., addressN, int oldval1, int oldval2, ....., int
oldvalN, int newval1, int newval2, ....., int newvalN){
    //atomic execution begin
    if((&address1 == oldval1) && (&address2 == oldval2) && .....
    (&addressN == oldvalN)){
        &address1 = newval1;
        &address2 = newval2;
        .
        &addressN = newvalN;
        return TRUE;
    }
    else{
        return FALSE;
    }
    //atomic execution end
}
```

The CASN operation conditionally updates a set of memory locations to a new set of values given that the words currently match a given set of values [5]. A CASN descriptor will be used that will contain all the information to describe the CASN operation. A CASN operation will need to execute atomically. But if the conditions do not match, i.e., if any of the expected values do not match the old values, the CASN operation is said to be failed and must re-write the previous values in the corresponding memory addresses. The addresses to be updated needs to be sorted to some total order which all threads agree to implement the non-blocking behavior [8].

The algorithm to implement CASN instruction begins by defining 'descriptors' and describing descriptor object's use. Descriptors are data structures in which the process initiating the operation stores all the information necessary to perform the operation. As a first step, a descriptor is installed or made active using a CAS operation by the concerning thread. When another concurrent thread attempting an operation referring the same memory cell finds a descriptor instead of an ordinary value, it can choose between various options - it can either back off or wait for the former thread to finish, or complete the operation on its own. First option, the technique of helping other threads to complete the operation ensures the progress of the overall system, which is a non-blocking implementation. For concurrent objects, linearizability is an important correctness condition [2]. If a concurrent operation appears to execute instantaneously in a given point of time 'τlin' between the time of invocation 'τinv' and the time of its completion 'τend', the operation is linearizable [2]. The literature often refers to 'τlin' as a linearization point. A common programming technique applied to guarantee the linearizability requirements for such operations is the use of a descriptor object [14]. CASN descriptors are described in detail in section 4.2. Now let us examine one simple example to describe the use of descriptors which allow an interrupting thread to help the interrupted thread complete an operation rather than wait for its completion.

4.1 Implementing SubtractX Operation

Let us consider an operation 'subtractX' that decrements a shared variable 'x' by a value of 'a'. Also, let us consider a trivial operation 'readX' on that shared variable which reads the contents of the memory location 'x'. The sequential behavior of the operation 'subtractX' is described by the pseudo-code as:

Algorithm 4.2: Pseudo code to implement subtractX and readX sequentially

```
int subtractX(int &x, int a){
    result = &x;
    &x = x - a;
    return result;
}
int readX(int &x){
    result = &x;
    return result;
}
```

The code provided above is trivial where the operation will subtract a value 'a' from the value in the memory location found in 'x'. Similarly, 'readX' is used to read the contents on that location. Now, the pseudo-code presented in 4.3 describes the operation defined above with a simple use of descriptors. It is however just an example and non-atomic implementation of 'subtractX' and 'readX'. The implementations of both operations described consider the use of descriptors which are made active by the thread itself installing it(or trying to) or other threads which might have installed it already in the memory location the prior thread is referring to. The thread might help perform the required operation on behalf of the other process that installed the descriptor(helping technique). Pseudo-code to implement this operation with descriptor objects is given as:

Algorithm 4.3: Pseudo code to implement substractX and readX with descriptors

```
object{
    int &x, &old;
    int a;
}SubDescriptor;
int subtractX(int &x, int a){
    //initialize a new descriptor
    SubDescriptor &desc = newSubDescriptor;
    do{
        desc.old = &desc.x;
        //install descriptor or help other thread
        S1: result = CAS(desc.x, desc.old, desc);
        if(isDescriptor(result))
            C1: CAS(result.x, result, result.old - result.a);
    }while(isDescriptor(result));
    //subtraction is done and descriptors made inactive
    C2: CAS(desc.x, desc, desc.old - desc.a);
    return desc.old;
}
int readX(int &x){
    do{
        result = &x;
        if(isDescriptor(result))
            C3: CAS(result.x, result, result.old - result.a);
    }while(isDescriptor(result));
    return result;
}
```

In this illustration implementation, a descriptor object is created first. The process invoking subtractX will initialize the descriptor object. At the start (S1), the invoking thread installs the descriptor (or will try to if the descriptor is already installed by prior threads in the same memory location). It uses CAS to install the descriptor. If the descriptor installed by other threads is already present in the memory location, it will help the thread which installed the descriptor to complete the operation and thus is unsuccessful in installing the existing descriptor (C1). If there are multiple

threads trying to subtract the value from the same location, the one which have already placed the descriptor pointers on the memory location will be able to complete the invocation first, the later threads will help complete it. After the helping is done, the thread will finally be able to install its descriptor. The invoking thread will now complete the subtract operation and delete its descriptor (C2) on the memory location where the descriptor pointer is installed. The pseudo-code for the readX operation is trivial. If a thread trying to read the location from the memory already has a descriptor reference, it will help it try to complete the operation (C3), make the descriptor pointer inactive and finally return the value from the location and the read operation is completed.

4.2 Illustration of CASN Operation

In multi-threaded programming, being able to update a number of shared variables at a same time atomically might be very useful. During the design of dynamic data structures, several pointers needs to be updated at the same time simultaneously. Because locks inherently limit the achievable parallelism, non-blocking implementations are essential. The hardware of most of the contemporary shared memory systems support at least one atomic primitives (like CAS). If not, there will be other instructions through which CAS can be easily implemented with a small effort. This hardware primitive is universal and can be used to implement any non-blocking shared data structure [7]. CAS is sufficient to construct CASN atomic primitive in a non-blocking manner. The CASN operation is implemented with the use of descriptor objects. It has a CASN descriptor which holds the number of information required to complete the operation. Basically, the fields it will hold are:

- A status field
- The addresses of the locations to be updated
- The expected values to be found there
- The new values to be updated
- A counter 'count'

The descriptor is active by placing the pointer to it in the location in shared memory. This will hence avoid locking the location in the memory and provide us with the efficient non-blocking alternative. If the location already refers to the descriptor pointer, later threads seeing the descriptor pointer will use the information in it to help the prior thread complete its operation and then release the location after the task is completed. A CASN operation carries on by placing pointers to its descriptor in each location being updated. It checks if they contain the values which are expected

to be there. If this succeeds for every location then all the locations are released, replacing the descriptor-pointers with the new values. If any location does not hold the expected value then the CASN is said to have failed and each location is restored to its old value.

Hence, for simplicity, CASN can be divided into two sections [8], using descriptor pointers so that other threads accessing the shared memory locations do not block (non-blocking method). The CASN operation can be divided into two parts:

1. Constructing Restricted Double Compare and Single Swap (RDCSS) operation
2. Implementing CASN operation from RDCSS

Fraser [17] designed the CASN operation using CCAS or the conditional compare and swap. CCAS uses a second memory location with the condition to control the execution of a normal CAS operation. The main purpose of using the CCAS operation in the algorithm is to avoid the ABA problem. But we will move ahead with the design of the RDCSS operation which uses only traditional CAS [8].

RDCSS can be understood as a restricted version of double DCAS operation described in section 3.2.3. This operation is implemented by using two CAS operations. The main purpose of this operation is to extend its use to derive a generic multi CAS operation. We can also use other atomic primitives like strong LL/SC primitives, conditional compare and swap [17] but the method we are following here [8] to use restricted form of DCAS which looks promising because it utilizes only the current primitive operation CAS which can be found on most of the current multiprocessor systems.

4.2.1 Construction of Restricted Double Compare and Single Swap Operation

RDCSS can be understood as a restricted version of double DCAS operation described in section 3.2.3. This operation is implemented by using two CAS operations. The main purpose of this operation is to extend its use to derive a generic multi word CAS operation. We can also use other atomic primitives like strong LL/SC primitives,conditional compare and swap [17] but the method we are following here [8] to use restricted form of DCAS which looks promising because it utilizes only the current primitive operation CAS which can be found on most of the current multiprocessor systems. RDCSS is also an atomic operation that takes two memory locations and writes new value only in memory location address2. Below is the pseudo code to implement this:

Algorithm 4.4: Pseudo code of RDCSS operation

```
object{
    int &address1, &oldvalue1, int &address2, int &oldvalue2, int &newvalue2;
}RDCSSDescriptor;
int RDCSS(RDCSSDescriptor &rdcssdesc){
    RDCSSDescriptor &rdcssdesc = new RDCSSDescriptor;
    do{
        result = CAS(rdcssdesc.address2, rdcssdesc.oldval2, rdcssdesc); //step1
        if(IsDescriptor(result)) //step2
            Complete(result);
        //IsDescriptor checks if the passed parameter points to a descriptor.
    }while(IsDescriptor(result)); //step3
    //If descriptor is not already installed, install the descriptor and complete the operation
    if(result == rdcssdesc.oldval2)
        Complete(rdcssdesc); //step4
    return result;
}
void Complete(RDCSSDescriptor &rdcssdesc){
    value = &rdcssdesc.address1;
    if(value == rdcssdesc.oldval1)
        CAS(rdcssdesc.address2, rdcssdesc, rdcssdesc.newval2); //update with new value
    else
        CAS(rdcssdesc.address2, rdcssdesc, rdcssdesc.oldval2); //restore old value
}
int RDCSSRead(int &address)
    do{
        result = &address;
        if(IsDescriptor(result))
            Complete(result);
    }while(IsDescriptor(result));
    return result;
}
```

RDCSSDescriptor data structure is a structure with the following fields:

RDCSSDescriptor	
address1	*address of the first condition*
oldval1	*value expected at the first address*
address2	*address of the second condition*
oldval2	*value expected at the second address*
newval2	*the new value to be written at the second address*

Figure 4.1: A RDCSS Descriptor.

RDCSS descriptor is created and initialized when a process which starts the RDCSS operation invokes it. Other threads will have no connection to it until the first CAS operation in the function RDCSS succeeds, making the descriptor active [8, 17]. The main notion of the algorithm is to replace the second memory location with a descriptor saying what you want to do. Then, given that the descriptor is present, check the first memory location to see if its value has changed. If it hasn't, replace the descriptor at the second memory location with the new value. Otherwise, write the second memory location back to the old value. So, to describe the above pseudo code for RDCSS in 4.4, in step1, first we try to change the value of address2 to our (own) descriptor rdcssdesc. If CAS operation succeeds then result will contain rdcssdesc.oldval2 (i.e. result is NOT a descriptor) else it will contain the pointer to descriptor installed by other processes invoking RDCSS operation. We will now check in step2 if result is a reference to descriptor i.e., whether it was already referenced by a descriptor of prior threads and step1 failed (another thread also might have changed address2). In this case, we will help another thread to complete its operation and release the descriptor. In step3, the loop will iterate again if descriptor not installed. In step4, if we succeed in fetching the expected value from address2, then we succeeded in installing our descriptor pointer in address2 and we can now move forward to complete our task of updating newval2 to address2 satisfying the condition that address1 still contains oldval1.

RDCSSRead operation simply returns the value from the memory location of the address passed to it. If the memory location is a reference to a descriptor, the operation will help the thread complete the operation which installed the descriptor pointer on that memory location. The proof of correctness of the RDCSS and RDCSSRead operations [8] explained above provides us with linearizable and non-blocking implementations. A very useful modeling and refining of non-blocking

algorithms like the one implemented in RDCSS operation [15] provide a method and tool support to linearize the implementation of operations like RDCSS. Various RDCSS operations performing on the same memory location can be serialized and we can consider them individually.

4.2.2 Construction of CASN Operation

It is assumed that memory locations can store both descriptor pointers and ordinary values and they can be separated from each other. The implementation states that the memory locations which are subjected to change with RDCSS operation will require reading those locations with RDCSSRead operation [8, 17]. Now, using the above RDCSS operation defined in algorithm 4.4, a multi word CAS operation can be constructed. A CASN descriptor will be used that will keep all the necessary information about the CASN operation in progress an d locking is done by replacing the value to be updated with a pointer to the appropriate CASN descriptor. CASNDescriptor is a structure with the following fields:

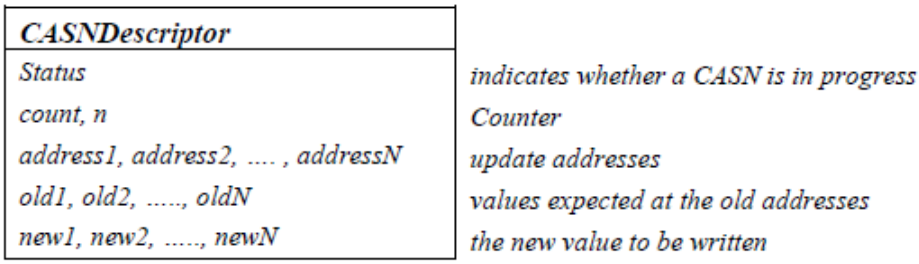

Figure 4.2: A CASN Descriptor.

The pseudo code of CASN operation is given as:

Algorithm 4.5: Pseudo code of CASN Operation

```
object{
    char status;
    int count;
    int &address[ ], int &oldvalue[ ], int &newvalue[ ];
}CASNDescriptor;
bool CASN(int count, int &address[ ], int oldvalue[ ], int newvalue[ ]){
    CASNDescriptor &casndesc = new CASNDescriptor; //descriptor object is created
    (casndesc.count, casndesc.address, casndesc.oldvalue[], casndesc.newvalue[],
casndesc.status) = (count, address, oldvalue[], newvalue[], UNDECIDED); //tuples of item
    Sortbyaddress(casndesc); //memory locations sorted in order of address
    if(casndesc.status == UNDECIDED) {
        //begin of phase 1:
        status = SUCCEEDED;
        for(int i = 0; (i < casndesc.count) && (status == SUCCEEDED) ; i++){
            insert_item: item = casndesc.item[i];
            result = RDCSS(new RDCSSDescriptor(&casndesc.status,
UNDECIDED,item.address, item.oldvalue, casndesc));
            if(IsCASNDescriptor(result)) {
                if(result != casndesc) { CASN(result); goto insert_item; }
            }
            else if(result != item.oldvalue) status = FAILED;
        }
        CAS(&casndesc.status, UNDECIDED, status); //operation is successful
    }
    //begin of phase 2:
    succeeded = (casndesc.status == SUCCEEDED); //true or false depending on status
    for(i = 0; i < casndesc.count; i++){
        if(succeeded == TRUE)
            CAS(casndesc.item[i].address, casndesc, casndesc.item[i].newvalue);
        else
            CAS(casndesc.item[i].address, casndesc, casndesc.item[i].oldvalue);
    }
    return succeeded;
}
```

Algorithm 4.6: Pseudo code of CASNRead operation

```
int CASNRead(int &address) {
    do{
        S1: result = RDCSSRead(address); //complete the incomplete RDCSS operation
        if(IsCASNDescriptor(result))
        CASN(result);
    }while(IsCASNDescriptor(result));
    return result;
}
```

Similar to the RDCSS descriptor, a CASN descriptor is created and initialized when a process which starts the CASN operation invokes it. The status field held by the descriptor may have values UNDECIDED, FAILED or SUCCEEDED depending upon the progress of the CASN operation. In CASN algorithm, the RDCSS operation will operate conditional on this status field.

The pseudo-code for CASN is divided into two phases. The first phase will install (try to) the CASN descriptors in the memory locations. At the end of this phase, pointers from each address to be updated are introduced to the descriptor. Each CASN invocation will create a descriptor which fully describes the updates to be made (let us say a set of (address, oldvalue, newvalue) tuples) and the current status of the operation which can be one of the following: UNDECIDED, FAILED or SUCCEEDED. For each of the entry tuples, the algorithm tries to reference it with the associated CASN descriptor by validating the old values with the expected values. For each item to be referenced, it will check if the update address already has the reference to the descriptor. This is done using the function *IsCASNDescriptor* which validates if the passed parameter points to a descriptor. If it is a reference to a descriptor, it will help the owning thread complete the operation. Thus, it permits recursive helping for the incomplete operations. The CASN operation is called recursively until all the values to be updated contains the reference to the descriptor. The coherent value of an owned location (by a descriptor) is found by asking the CASN descriptor that is active on that location. At the end of first phase, if the status of descriptor is UNDECIDED or FAILED, the coherent value is then expected value. If the status is SUCCESSFUL, the value is the new value. Based on the status of the descriptor, second phase of the algorithm advances to update the old value with the new value. If any of the memory locations failed to reference the descriptor pointer, we will assume that CASN failed and will update each of the memory locations with the old values. And the descriptor pointers are made inactive or freed from the memory locations recently updated.

CASNRead defined in algorithm 4.6 is similar to RDCSSRead. Both employ the helping functionality in the sense that if the memory location a CASNRead it trying to refer to already contains reference to the descriptor pointer, it will invoke the CASN operation and will let the thread complete its operation. Or if the location contains reference to a RDCSS descriptor, it will help it complete it first and then return an ordinary value or CASN descriptor pointer (S1). The algorithm will advance further to complete the CASN operation (if in case a CASN descriptor returned) and finally return the ordinary value in the memory location.

Chapter 5

Proposed Linked List and Skip List Implementation

Linked list is a data structure consisting of a group of nodes aligned together representing a sequence. This data structure comprise of cells containing two fields: a key field which is used to store the element and a next pointer which will store the reference to the next item on the linked list data structure. Each node in a linked list has two neighbor nodes, one preceding it and one succeeding it. The first node in the list is called the head and the last is called the tail of the linked list (Figure 1.1). These are the sentinel nodes of the list.

A linked list is a general data structure which can be used to implement various different abstract data types. We will use this data structure to build non-blocking implementation of skip lists later. We here consider a sorted linked list with head and tail sentinel nodes at the ends of the linked list. There are various operations that can be done on the linked list data structure. Let us demonstrate the execution of each of the operations. Later we will describe and discuss the non-blocking implementations of the operations on this data structure. The operations supported by linked list are:

- *Adding a node*:

  This operation introduces a new node with the desired key in between the two nodes strictly less than and greater than it. If the linked list already contains a node with the key to be inserted, the add operation should fail. In case of successful addition of a node, the desired node's next pointer is pointed to the node having key just greater than it. We will call this node a successor node. The node with the key just below the desired key's node will be called

a predecessor node. This predecessor node's next pointer will be then updated to point the new node. If both the steps are successful, the new node has been added to the linked list.

- *Deleting a node*:

  This operation will delete the node with the desired key from the linked list. If the linked list does not contain a node with the key to be deleted, the delete operation should fail. In case of a successful deletion of a node, the next pointer of predecessor's desired node is updated to point to its successor.

- *Searching a node*: This is a trivial operation where the node with the desired key is searched in the entire list. If the linked list contains or does not contain the node with the desired key, the operation will move ahead with the decision accordingly. The decision can be acknowledged with true/false values simply.

- *Prepending a node*:

  The prepend operation will prependa node at the start of the linked list. To prependa node, the node to be prepended must have a key strictly less than any other keys in the list. If the key to be prepended is the least among all, a node is created and added just after the head of the linked list.

- *Appending a node*:

  The append operation will add a node at the end of the linked list. To append a node, the node to be appended (added at the end of the linked list) must have a key strictly greater than any other keys in the list. If the key to be added at the end is greatest of all, a node is created and added just before the tail of the linked list.

5.1 Proposed Linked List Algorithms

There are various approaches to perform addition, deletion, appending, prepending and other operations in a linked list, both blocking and non-blocking alternatives. The case will be trickier if we want to implement this data structure in a non-blocking manner. The major problems occur during concurrent updates to the same part of the list. These include deletion of adjacent nodes, concurrent insertion of nodes between the same pair of nodes, and concurrent insertion and deletion of adjacent nodes. For the deletion of the node, we will follow the technique implemented by Pugh [16], using the concept of back pointers. Briefly, the concept is, when deleting a node, we will update the predecessor of the node to point to its successor. In addition to it, we will also update the

node's next pointer to point back to its predecessor. Other threads that were passing through the node at the time of its deletion can follow this reversed pointer to get back into the current list, at the correct place to continue the work [16]. We will exercise this technique in our implementation of lock free linked list (using CASN atomic primitive) and then design skip lists using this linked list.

We will design our linked list implementation with the help of CASN operation described in chapter 4. Since memory locations can be concurrently updated, this implementation will ease the overhead of maintaining the marked node as implemented in designs [12, 13]. The primary use of marked node is to mark the node in the process of deletion. This marked node is said to be logically deleted from the list and the next step will be to delete the node physically by swinging its predecessors next pointer to its successor. This two-step deletion process will require the concept of marking nodes before removing it physically to maintain concurrency in the data structure. Here in our algorithm, we will instead incorporate the approach suggested by Pugh [16] which is using the concept of back pointers along with CASN atomic primitive to implement the data structure. Each node in a linked list is connected to each other using the next pointer reference. The linked list is empty first, only having the head and tail sentinel nodes. In concurrent data structures, addition and deletion of nodes occur simultaneously. The lock free implementation should guarantee the progress of the operations even if the other processes are on their paths. Now let us describe in detail the implementation of various linked list operations in concurrent non-blocking manner using the CASN primitive.

5.1.1 Adding a Node in a Linked List

Adding a node in a linked list will be successful if and only if the node is not previously present in the list. The key to be added in the linked list is inserted between two nodes - the predecessor and successor nodes which are strictly less than and greater than the key to be inserted. During the addition of node, a new node is created and inserted between the predecessor and successor nodes. The add operation starts with the invocation of find method call which will supply the predecessor and successor for the key attempted to be inserted. Based on the find values, the pointer references of the node to be added and the predecessor nodes are updated atomically using the CASN operation. This primitive help us ensure the lock-free implementation of the add operation. The execution of the add operation is illustrated in figure 5.1.

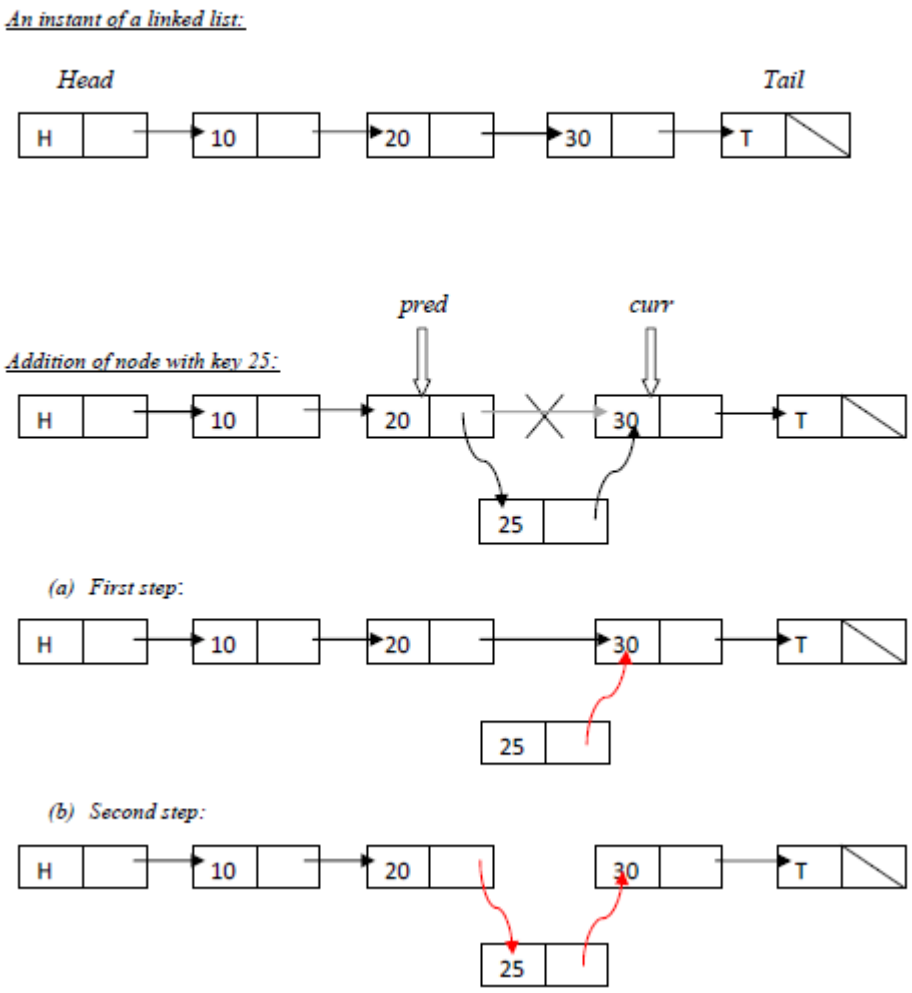

Figure 5.1: Adding a node in a linked list. A linked list with three nodes: 10, 20 and 30. Addition of a node with key 25 between pred and curr nodes. In fig (a), in the first step, the new node updates its next pointer with the address of curr node. In fig (b), in second step, the preds next pointer is updated to point the new node. The addition of node is successful.

A successful addition of nodes will require new nodes next pointer and pred nodes next pointer to be updated atomically without any interference from other concurrent processes. We ensure this not by using traditional CAS operation [11, 12, 2] but by using CASN. The add operation (Algorithm 5.1) proceeds with a find call (Algorithm 5.2). The find operation is also used by delete operation. The find operation takes a key and returns the reference to two nodes. These nodes are called predecessor and successor nodes of the key provided, which are denoted by 'pred' and 'curr'. The operation always starts traversing the list from the head. A CASNRead operation is invoked

which will ensure that any other concurrent CASN operations working on that memory location will continue to complete its work first and then release the location. The nodes are traversed one after another until the key with the searched value greater than or equal to it is encountered. After this comes the validation step, where we need to ensure that the predecessor still refers to the current node. This is because other concurrent operations might add or delete the nodes in between. After the validation is successful, references to pred and curr nodes are returned. If it fails, the pred and curr values must have changed, so the operation will start over again. After the find call returns the valid pred and curr values, addition will proceed checking if the curr node is itself the node the operation is trying to add. If the node already exists as curr node, the addition should fail because the node is already present in the list. If not, a new node with the key to be inserted is created.

We will now use the powerful CASN operation to update the next pointers of new node and the predecessor - both at one single step, atomically. The CASN operation will try to modify these two memory locations atomically by installing the CASN descriptor pointers to the locations first. There will be no problem to install the descriptor reference to a newly created node. But, if any other operation will update the preds next reference, the CASN operation will fail to install the descriptor on that location and the CASN operation fails. This will guide the code to execute find, to identify the new set of pred and curr nodes. If CASN operation succeeded, both the memory locations are confirmed to be updated atomically without any intervening processes. This will ensure that our node with the desired key is inserted between the pred and curr nodes returned by the find method. The pseudo code to implement add operation along with find method is given as:

Algorithm 5.1: Pseudo code to implement 'add' operation in a linked list.

```
bool add(int key){
    Node pred, curr = new Node();
    int cas1 = 0, cas2 = 1;
    do{
        (pred, curr) = find(head, key);
        if(curr.key == key){
            return FALSE;
        }
        else{
            Node node = new Node(key);
            (address[cas1], oldvalue[cas1], newvalue[cas1]) = (&node.next, null, curr);
            (address[cas2], oldvalue[cas2], newvalue[cas2]) = (&pred.next, curr, node);
        }
    }while(CASN(2, address[], oldvalue[], newvalue[]) == FALSE); //if CASN fails, retry
adding.
    return TRUE;
}
```

Algorithm 5.2: The linked list 'find' operation: a helper function used by add and delete operations.

```
(Node, Node) find(Node head, int key){
    start: Node pred, curr = NULL;
    pred = CASNRead(&head);
    while(TRUE){
        curr = CASNRead(&pred.next);
        //CASNRead will consequently return a value rather than descriptor reference.
        if(curr.key >= key){
            break;
        }
        pred = curr;
    }
    if(CAS(pred.next, curr, curr)){
        return (pred, curr);
    }
    else{
        goto start;
    }
}
```

5.1.2 Deleting a Node in a Linked List

A successful deletion of a node will require target nodes preds next reference to be updated with its successor nodes memory location. Similar to add operation, find operation will distinguish the predecessor and current nodes of the target key. If the curr nodes reference is not equal to the target key, the node with the key to be deleted is not said to be present on the linked list and will return false (Figure 5.2). If the key is present in the curr reference, the deletion of the node will proceed. The execution of deletion operation is given in Figure 5.2.

The deletion of the node will advance with the find operation (Algorithm 5.2). The operation will return the pred and curr node references for the key passed to it. For deletion, if the find call does not return the curr node with the reference of node having key searched for, there will be no node to delete and deletion fails. If there is a key with the target value in the list, the deletion operation will move forward with the operation. First, the successor node for the curr node is identified using CASNRead operation. As usual, if there is another operation working on that memory location, CASNRead will find the descriptor reference already present there and will help that operation complete it first. Ultimately, it will return the curr nodes successor reference. For deletion of the node, we will need to update two memory locations atomically. The reason we are updating the pointer of the node to be deleted to point to its predecessor is that other concurrent operations that

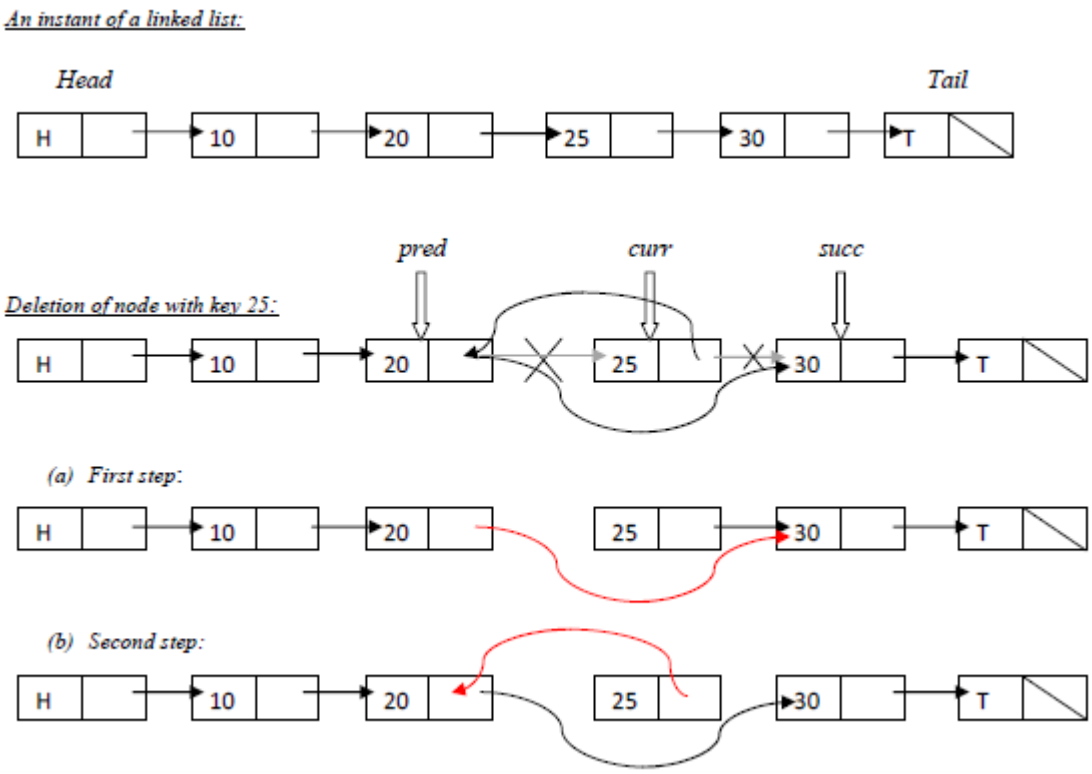

Figure 5.2: A linked list with four nodes: 10, 20, 25 and 30. Deletion of a node with key 25'. In fig (a), in the first step, the pred node updates its next pointer with the address of succ node. In fig (b), in second step, the currs next pointer is updated to point the previous node. The deletion of node is successful.

were traversing through the deleted node at the time of its deletion can follow this reversed pointer path pointed back to its predecessor to get back into the current list, at the correct place and to continue their work [16]. Also, this will prevent other concurrent threads adding other nodes after the deleted nodes. Changing the next reference of the deleted nodes will ensure the failing of CASN for other add operations trying to add nodes after the deleted node (preds next will no longer point to the curr node). This verification while adding nodes is done in the add operations CASN when updating the second memory location (Algorithm 5.1). So, deletion will proceed only after both the currs next and preds next references are correctly updated. If at any point between updating these nodes next field, the references are updated by other concurrent threads, than CASN will fail and the deletion process retries. If both the memory locations are updated correctly, the node will no longer be reachable in the linked list from the head and the deletion process returns with a true value. The pseudo code to implement delete operation is given as:

Algorithm 5.3: Pseudo code to implement 'delete' operation in a linked list.

```
bool delete(int key){
    Node pred , curr = new Node();
    int cas1 = 0, cas2 = 1;
    do{
        (pred, curr) = find(head, key);
        if(curr.key != key){
            return FALSE;
        }
        else{
            Node succ = CASNRead(&curr. next);
            (address[cas1], oldvalue[cas1], newvalue[cas1]) = (&pred.next, curr, succ);
            (address[cas2], oldvalue[cas2], newvalue[cas2]) = (&curr.next, succ, pred);
        }
    }while(CASN(2, address[], oldvalue[], newvalue[]) == FALSE);
    return TRUE;
}
```

5.1.3 Searching a Node in a Linked List

Another operation we define is the search operation which returns boolean result 'true' or 'false' depending upon if the node with the target key is present or absent from the linked list. There are various wait-free implementations of this operation. Herlihy *et. al* [2] implemented lock-free linked list using atomicmarkable reference object from java concurrent library. He presented a wait-free implementation of search operation by the use of concept of marking nodes logically deleting nodes before physically deleting it. Our search operation will not be wait-free but will be non-blocking implementation, a lock-free operation. The search operation will advance starting from the head of the list, traversing each node one by one. CASNRead operation will be used to read the memory location because other processes might have already acquired the memory location and will need to help them complete it. This behavior will make this operation lock-free and limit from wait-freedom property [2, 10, 12, 18]. However, if the node yet traversed reaches the value greater than or equal to the lookup key, a decision is made returning true or false depending upon if the key is present in the set. The pseudo code to implement search in linked list is given as:

Algorithm 5.4: Pseudo code to implement 'search' operation in a linked list.

```
bool search(int key){
    Node pred = new Node();
    Node curr = new Node();
    pred = CASNRead(&head);
    while(TRUE)
    {
        curr = CASNRead(&pred.next);
        if(curr.key >= key)
        {
            break;
        }
        pred = curr;
    }
    return (curr.key == key);
}
```

5.1.4 Prepending a Node in a Linked List

Prepending a node in linked list refers to adding a new item at the beginning of the linked list. This operation inserts a new node at the beginning of the linked list only if the condition that the key to be inserted is the least among all keys of the list is satisfied. It is similar to add operation described above but careful observations are required to prepend a node at the beginning of the list. The linked list data structure we are referring to is in sorted order. So, the new node which will be inserted will be the first item in the linked list. This key of this item must strictly be less than any of the other keys in the linked list. If it is not the least of all, this operation should fail and rather the add operation should be called to insert the node.

The operation starts by finding the next reference of the head in the linked list (Algorithm 5.5). The node to be prepended must have the key smaller than first item in the list (referred to as firstnode in the code). If the condition is not satisfied, the key should not be inserted and the operation terminates. If the key to be inserted is the least among all, the next pointers of the head of the list and that of new node is updated using atomic CASN primitive. Two CAS operations are executed atomically to ensure that the next reference of new node points to the first item in the linked list and the next reference of the head points to the new node itself. If during the placement of descriptor pointers by the CASN operation, anything changes (new node added or another operation working on the same memory location), the CASN operation will fail or help the other pending operations. The CASN operation used to update the head and new nodes pointers will start (try to) by placing the descriptor pointers on the memory locations to be updated. If the

operation succeeds to install the descriptor pointers, no other threads will interfere in between but rather help the operation to complete. The implementation of prepend operation (Algorithm 5.5) is lock-free. The pseudo code to implement prepend operation is given as:

Algorithm 5.5: Pseudo code to implement 'prepend' operation in a linked list.

```
bool prepend(int key){
    int cas1 = 0;
    int cas2 = 1;
    Node firstnode = new Node();
    do{
        firstnode = CASNRead(&head.next);
        if(firstnode.key <= key)
        {
            return FALSE;
        }
        else
        {
            Node node = new Node(key);
            (address[cas1], oldvalue[cas1], newvalue[cas1]) = (&node.next, null, firstnode);
            (address[cas2], oldvalue[cas2], newvalue[cas2]) = (&head.next, firstnode, node);
        }
    }while(CASN(2, address[], oldvalue[], newvalue[]));
    return TRUE;
}
```

The prepend operation presented here adds an element strictly at the beginning of the list. This unique behavior of the operation can lead us with other useful implementations. The prepending and deletion of an item from the beginning of the list only requires O(1) time. This behavior can be used to implement a prominent data structure stack. The item inserted just after the head of the list can also be seen as push operation in a stack. An operation which will ensure the removal of first item in the list can be introduced easily and will act like a pop operation. This resulting behavior can provide us with a stack with Last In First Out (LIFO)nature.

5.1.5 Appeding a Node in a Linked List

This operation inserts a new node at the end of the linked list, just before the tail sentinel node only if the condition that the key to be inserted is the greatest among all keys of the list is satisfied. Appending a node at the end of the linked list is similar to inserting a node with the greatest key in the linked list. The key must satisfy the condition that it must strictly be greater than any other keys in the linked list. If the key to be inserted is not greatest, or equals the already present key in the linked list, the operation must fail and the operation terminates.

The operation starts the traversal from the head of the linked list (Algorithm 5.6). Each of the nodes in the list is traversed until the tail sentinel node is reached. The node just before the tail node is recorded and is the last element on the linked list. This node is now compared with the intended key to be appended in the linked list. If the recorded keys value is greater than or equal to the key to be appended, the operation will fail. Otherwise, with the help of CASN atomic primitive, the next pointers of the new node to be appended and the recorded last node are updated atomically. As we can see, in case of the empty list (C1), with head directly pointing to tail, traversing the node is unnecessary and may cause undesirable effects. So, we limit this in the code by visualizing the head of the empty list as the last item. Other updates of the pointers will go similarly. This implementation of append operation(Algorithm 5.6) is lock-free. All the memory locations are updated in a non-blocking manner using the CASN operation. The pseudo code to implement append operation is given as:

Algorithm 5.6: Pseudo code to implement 'append' operation in a linked list.

```
bool append(int key){
    int cas1 = 0, cas2 = 1;
    Node pred, curr, lastnode = new Node();
    pred = CASNRead(&head);
    do{
        while(TRUE){
            curr = CASNRead(&pred.next);
            C1:if(CASNRead(&head.next) == CASNRead(&tail)){
            curr = pred; //if the list is empty
            break;
            }
            if(CASNRead(&curr.next) == CASNRead(&tail)){
                break;
            }
            pred = curr;
        }
        lastnode = curr;
        if(lastnode.key >= key){
            return FALSE;
        }
        else{
            Node node = new Node(key);
            (address[cas1], oldvalue[cas1], newvalue[cas1]) = (&node.next, null,
CASNRead(&tail));
            (address[cas2], oldvalue[cas2], newvalue[cas2]) = (&lastnode.next,
CASNRead(&tail), node);
        }
    }while(CASN(2, address[], oldvalue[], newvalue[]) == FALSE);
    return TRUE;
}
```

The append operation presented here takes O(n) time. The operation can be optimized to O(1) if we can maintain a certain level of bookkeeping. If we can track the last node in the list, the prepending the operation can be done easily in constant time. For making this happen, all the changes to the last item in the list (removal of the item, other append operations, insertion of the item, clearing of the list, etc.) should be tracked. If we are able to store this extra bit of information every time for the list, we will be able to prepend an item without traversing the list from the head. This will behave similar to enqueuing an item in the queue data structure. With an additional operation which ensures the removal of item from the head, the nature of these operations will provide us with a queue with First In First Out (FIFO) behavior.

To summarize the linked list operations presented above which can also be viewed as an implementation of set using linked list,

- The add(x) method adds x to the linked list based set, returning true if and only if x was not present in the list.
- The delete(x) method deletes x from the list based set, returning true if and only if x was present in the list.
- The search(x) returns true if and only if the list based set contains x, else returns false.
- The prepend(x) method adds a node at the beginning of the linked list, just after the head sentinel node.
- The append(x) method adds a node at the end of the linked list, just before the tail sentinel node.

All these operations are implemented to work in a lock free manner.

5.1.6 Proof of Concurrency

To prove the correctness of the algorithms of the operations presented above, let us demonstrate few sample executions of the various operations and analyze each of the invocations and responses of the concurrent operations in detail. It will be easier to reason about the operations because of the use of the powerful CASN operation in the implementation. Let us consider a scenario described Figure 5.3.

A thread A is about to delete item 20 and another concurrent thread B is about to add item 25 in the list. As we can see, both the threads will operate on the same memory locations to make their

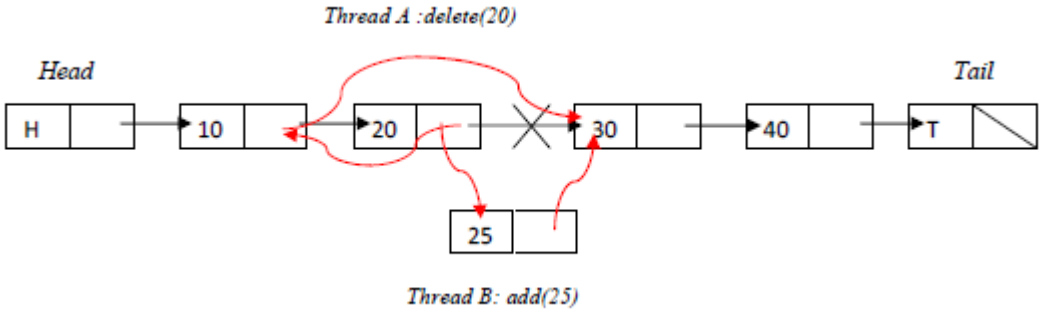

Figure 5.3: Two concurrent threads A and B operating together. Thread A is trying to delete an item from the list and thread B is trying to add an item.

respective changes to the pred and curr node's next pointers. If thread B would add 25 just after thread A completed its update of its pred's next field (node 10's next field point to node 30) but thread A yet to update the deleted node's back pointer, the net effect would be that 20 would be deleted successfully but 25 would not be added correctly. Harris [12] and Herlihy [2] solved this using both CAS together with the 'marked' bit of the node's next field. In our implementation, we use CASN to update both the pointers atomically, solving this issue. Hence, this will not be a problem in the implementation using CASN because if deletion by thread A preceded before addition by thread B, thread A will install its descriptor pointers prior to thread B and complete its operation. In addition to it, if thread B finds A's descriptor pointer in the memory location, it will help thread A to complete the operation. Thread B will again call find() to get a new set of pred and curr nodes and proceed with addition. Also, if we consider a case where thread A has installed its descriptor pointer on pred's next field but yet to install on curr's next. Thread B at the same time adds the item in the list. In this case, thread A's CASN will fail because the curr will no longer be pointing to the expected succ node. So, the find() method of delete operation will be called again to get a new set of pred and curr nodes and delete operation will proceed.

Now, let us consider another scenario described in figure 5.4. Thread A is trying to delete node 20 from the linked list and thread B is trying to delete node 30, both operating concurrently.

In Figure 5.4, if compare and set operations used were disjoint and executed individually, and if thread A after performing its updating of pointer from its pred's next (node 10) to its succ node (node 30), if another concurrent thread B tries to perform deletion of node 30 swinging its pred's (node 20) next pointer to its succ (node 40), the result would be that node 30 would not be deleted

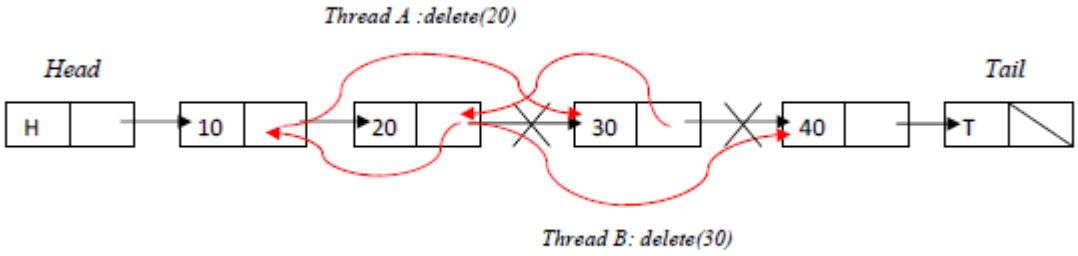

Figure 5.4: Two concurrent threads A and B operating together. Thread A is trying to delete an item from the list and thread B is trying to delete another adjacent item at the same time. Operations of both operations are shown for clarity only.

correctly because node 20 is not reachable from previous deletion. In our implementation, this will not be problem because we ensure that the concurrent deletion operations do not interfere with each other. Let us assume if thread B preceded thread A. In this case thread B will execute CASN on its pred's next field to point to its successor and the target node's (node 30) next field itself to its predecessor. Another concurrent thread (thread A) trying to delete node 20 (thread B has placed its descriptor pointer) while trying to update node 20's next pointer to point its predecessor (node 10) will find updated values and CASN for thread A will fail and thread A restarts to get new set of pred and curr values to continue deletion. So, thread A will only proceed after CASN of thread B is completed. This ensures that concurrent deletions do not interfere with each other. Because of the powerful CASN operation, we are able to modify all of the locations needed to be modified atomically without any other thread intervening in between the thread's invocation and response period.

Now we will move ahead with describing the skip list and designing lock free operations for this data structure. Basically, we will extend the implementation of linked lists and implement it on skip lists.

Skip lists are a data structure that can be used in place of balanced trees. Skip lists use probabilistic balancing rather than strictly enforced balancing and as a result the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.

-William Pugh

Skip lists are basically composed of linked lists stacked together forming levels. The maximum level of the skip list will be fixed and is calculated in advance. The leftmost and rightmost nodes are the head and tail of the data structure (Figure 1.2) with the minimum and maximum possible keys. The lowest level of the skip list contains every item in the list, and the levels above it probabilistically contain that item up to some maximum level that is chosen independently and randomly for each node using some random level generator function. The level of a node to be added in skip list is determined using a random level generator [21].The number of nodes in each level of skip list decreases exponentially with the level, implying that a key can be quickly found by searching first at higher levels, skipping over large numbers of nodes, and progressively working downward until a node with the desired key is found, or the bottom level is reached. Thus, the expected time complexity of skip list operations is logarithmic in the length of the list [2, 13, 21, 23].

Each node in a skip list is part of at least one linked list (the lowest level), and at maximum it could be part of all the lists in all levels. It can also be visualized as a single linked list with each node containing more than one next pointer. This set of next pointers will allow us to traverse the list of nodes, skipping over them. We can see that by maintaining a sorted order in the list, it will help us traverse the list to find the desired node. As the higher level lists are sparser, we can use them to reach the key we are looking for in fewer steps.

Balanced search trees need to re-arrange its height accordingly as the fundamental operations (insert, delete) are performed to maintain good performance [20]. Skip lists can be used as a replacement to balanced search trees. Skip lists do not require re-balancing and takes a probabilistic approach to maintain the balance. The probability of the level of a node in a skip list is generated using a random generator. The probability that a node is at level 1 is 1, at level 2 is 0.5, at level 3 is 0.25, at level 4 is 0.125 and so on [20, 21]. Due to this probability distribution, each higher level list in skip list roughly skips about 2^i nodes in the list just below it [2]. For example, maximum level of 8 is appropriate for skip list data structure of 2^8 elements. Between any two nodes at a given level, the number of nodes in the level below it is constant. So, the height of the skip list is logarithmic to the number of nodes. Balanced trees can do everything that a skip list can do. The only reason skip lists can replace balanced search trees is its probabilistic approach used to insert nodes in the data structure, which makes it very simple to implement, modify and scale. Under the assumption that the input sequence will not consistently produce worst-case performance, implementing skip lists can be easier and faster than balanced trees.

A skip list is built in layers. The bottom layer is an ordered linked list with all the items present. Each higher layer acts as an "express lane" for the lists below, where an element in level 'i' appears in level 'i+1' with some fixed probability p (two commonly used values for p are 1/2 or 1/4) [23, 21]. On average, each element appears in 1/(1-p) lists. Searching of an item in a skip list begins at the head. Each layer of the skip list above is the lock-free linked described in section 5.1. Each node in the skip list contains a next reference that is the array of list level references, one for each of the lock-free skip list's list-levels to which the node can be linked. The position of a node during searching is allocated by finding the set of predecessors and successors for that node. The set refers to the each of the predecessor and successor for that node in each level of the skip list. All the operations performed in the linked lists can be done with skip lists. Let us demonstrate the execution of each of the operations. The operations supported by skip list we discuss here are:

- Adding a node:

  This operation introduces a new node with the desired key in between the two nodes of the skip list, strictly less than and greater than it. A random top level is generated for the new node and is spliced in between the calculated set of predecessor and successor nodes at each level. At each level, the new node's next pointer will point its successor for that level (up to the top level). Starting from the bottom level, the predecessor node's next pointer for that level will be then updated to point the new node. If both the steps are successful, the new node has been added to the skip list.

- Deleting a node:

  This operation will delete the node with the desired key from the skip list. The item will be in the skip list if it is in the bottom level list. This is because all the higher level lists are subsets of lower level list and the list at the bottom level contains all the items in the skip list. If the skip list does not contain a node with the key to be deleted, the delete operation should fail. At each level, predecessor of the node to be deleted will point its successor and the node's next pointer will point its predecessor (using the concept of back pointers) [16, 21].

- Searching a node:

  This operation will make a decision whether a node is present in the skip list. A search starts from the current maximum level of the list of the head and descends down to the bottom level list maintaining list-level references to a predecessor node and a current node. The search will skip over nodes at each level if the key of desired node is greater than the nodes in path. The process of skipping nodes continues until the node with key greater than or equal to the

desired key is found. The execution stops at the bottom level and acknowledgement is made with yes/no decision.

- Prepending a node:

  The prepend operation will append a node at the start of the skip list. To prepend a node, the node to be prepended must have a key strictly less than any other keys in the skip list. If the key to be prepended is the least among all, a node is created and added just after the head of the skip list.

- Appending a node:

  The append operation will add a node at the end of the skip list. To appenda node, the node to be appended (added at the end of the skip list) must have a key strictly greater than any other keys in the skip list. If the key to be added at the end is greatest of all, a node is created and added just before the tail of the skip list.

5.2 Proposed Skip List Algorithms

All these operations described above will be implemented using CASN operation in a non-blocking manner. We will design our skip list implementation with the help of CASN operation described in chapter 4. With the above described powerful CASN primitive, deletions, additions, appending and prepending of nodes will be atomic. All the respective pointers to be updates at each of the levels of the skip list will be carried out as a single atomic step. This will ease the implementation in the highest degree and make the concurrent reasoning of the operations easier. During the implementation of various operations, we make an assumption that some random level generator function provides us with the level of a node in a skip list. Let us discuss on this function first.

Finding a random level is not as easy as just picking any random between some of the possible levels. An important point to note in a skip list is that the maximum level should be fixed during its design. If a lot of time is spent moving down to the next level from an unreasonably high column compared to the rest, any gain from the skip list could be lost. So there should be a bound to the upper limit to the maximum level in a skip list [24]. A skip list with a maximum level of 16, for example, could hold approximately 2^{16} nodes [21, 23]. So, level can be logarithmic and still support a large number of nodes. For finding a proper random level, using random number generators is not a good idea. Random number generators return a random number with uniform distribution. This will make every level to be equally probable. This will not make an efficient data structure that we

are creating using randomization. So, we will need a function that will return a weighted random number based on a probability equation. To get the binary tree-like structure in a skip list, each level must be about half as probable as the level below it, or in more precise terms, the random height should be chosen with a probability of 1/2. To find such a probabilistic number, the easiest way to achieve is to "flip coins". Starting from the lowest level, we will flip the coin until we will find the "head". Few examples of random level generating functions are:

Algorithm 5.7: Two designs to generate a random level. The level thus generated can range from minimum value 1 through maxlevel

```
int randomlevel(){// returns a random level for a node to be added [21]
    int level = 1; //should never return 0
    while(random() < 0.5 and level <maxlevel) //random function returns value 01
        level = level + 1;
    return level;
}
int randomlevel(int maxlevel){
    int a = 0, b = 0;
    for(int level = 0; b = 0; level++){
        if(a == 0) {
            a = random(); //calculate a random number
        }
        b = a % 2; //if the remainder is 1, the loop will terminate
        a = a / 2;
     }
    if(level > maxlevel){
        level = maxlevel - 1;
    }
    return level;
}
```

Above are the two designs to calculate the random level. The first one is expensive because it calls the random() function in every iteration of the loop which might cause bottlenecks. So, in the second design [24], the random function is called only once and using the bits of the returned data, the level is calculated. Now let us describe the implementation of various skip list operations designed to work in concurrent non-blocking environment.

5.2.1 Adding a Node in a Skip List

The add operation adds a node between the two nodes of a skip list with the keys strictly less than and greater than the node to be added (Figure 5.5). If the key intended to be added is already present in the skip list, the operation fails. For a successful add operation, first the set of predecessor and successor nodes are calculated for each of the levels of the skip list using the find

operation (Algorithm 5.9). The new node to be added is to be spliced between these levels of the skip list. A helper function randomlevel (Algorithm 5.7) generates a random value 'toplevel' for the new node. The new node will then be linked on all the levels, starting from bottom level to top level between the predecessor and successor nodes. The linking of the pointers require the new node's next pointer to be updated with successors from bottom to top level and the predecessor nodes next pointer updated with the address of the new node. We use CASN primitive to update all these set of pointers, all at once atomically in a lock-free manner. Let us consider the example below to illustrate the add operation:

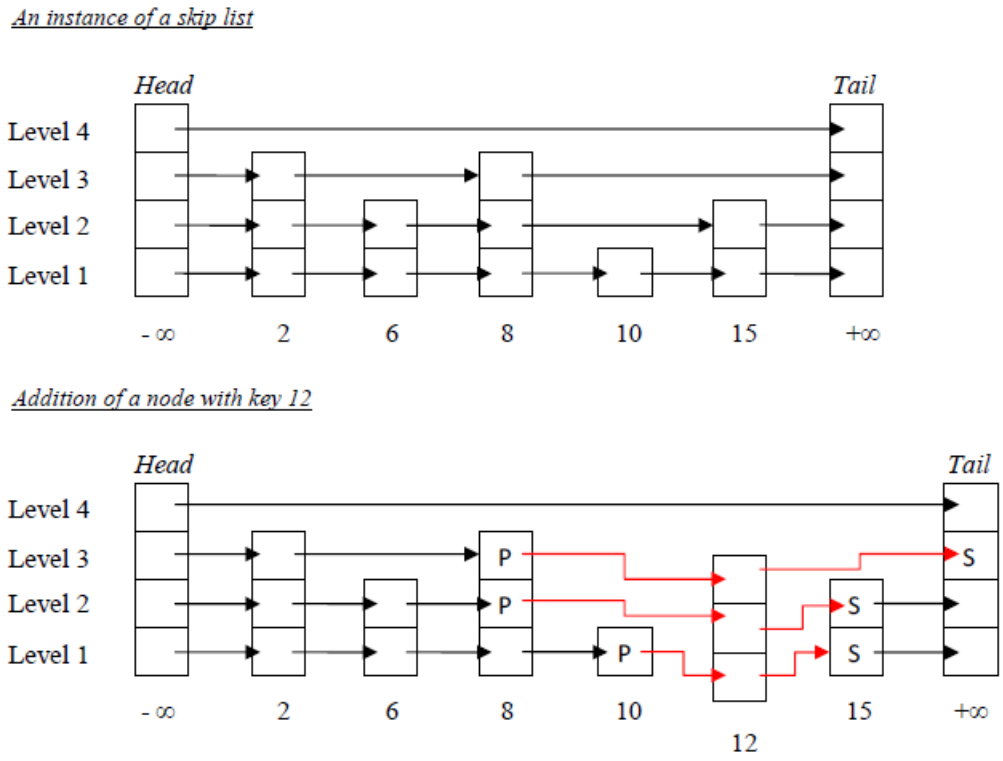

Figure 5.5: A skip list with five nodes and four levels. Addition of a node with key 12 in the skip list.find() operation is executed first to find the set of predecessors and successors(stored in preds[] and currs[] array). For key 12, the set of nodes of preds[] and currs[] in each level are denoted by P and S respectively.

The add operation (Algorithm 5.8) advances with the find call (Algorithm 5.9). The find call provides the predecessor and successor nodes at each levels for the key supplied with the results stored in preds and currs arrays. The value of each preds in each level is strictly greater and the value of currs is strictly greater than or equal to the supplied key. The find call starts from the

head of the maximum level(here 4). As it moves down the levels up to the lowest level, it stores the predecessor and successor for the supplied key in the preds and currs array for that level.

The current maximum level (curr_maxlevel) is a shared variable maintained to store the current maximum level of any node in the skip list. This will help us to start from the midst of the skip list levels rather than always starting from the maximum level. We will use it in search operation and not in find because we will need all set of predecessors and successors for a node at all levels in find method (new node to be added can have any toplevel upto maxlevel allowed).

After the set of apparent predecessors and successors are calculated, we verify if the key to be added is already present in the list. Verifying the successor node at bottom level of the skip list is enough because each of the higher level is the subset of the lower levels and the bottom level contains all keys. If the key is already present, the add operation fails. If no such key is found, a new node is created with a randomly generated top level. This node will be spliced between the predecessor and successor nodes from the random top level to the bottom level. The new node's next pointer is updated with the successor nodes for each of its levels. To update the predecessor node's pointers, add operation uses CASN operation to ensure all the updates are atomic. For this, the CASN operation begins by installing the descriptor pointers at each of the predecessor's next fields at each level. This operation provides us with the lock-free implementation of the add operation. The descriptor pointers are correctly installed only if none of the concurrent threads have interfered in between the add operation's invocation and response. If any other operation had changed the predecessor's next field, or may be deleted it, the CASN will fail because the predecessor no longer point to the successor. In such a case, all the memory locations with descriptors already placed are replaced with old values and the add operation retries. If no any interference is caused by other processes, the CASN operation will update the entire predecessor's next field with the address of new node and the add operation terminates with a success.

Algorithm 5.8: Pseudo code to implement 'add' operation in a skip list.

```
object{
    int key;
    int toplevel;
    Node next[toplevel];
}Node;
bool add(int key){
    Node[] preds, currs = (Node[]) new Node[maxlevel];
    int bottomlevel = 1;
    int cas1 = 0;
    int toplevel = randomlevel(); //assign a random level to new node
    do{
        (preds[], currs[]) = find(head, key);
        if(currs[bottomlevel].key == key){
            return FALSE;
        }
        else{
            Node node = new Node(key, toplevel);
            for(int level = bottomlevel; level <= toplevel; level++){
                node.next[level] = curr[level];
                (address[level], oldvalue[level], newvalue[level]) =
(&preds[level].next[level], currs[level], node);
            }
        }
    }while(CASN(toplevel, address[], oldvalue[], newvalue[]) == FALSE);
//update the current max level.
    do{
        result = CASNRead(&curr_maxlevel);
        if(result < maxlevel && head.next[result+1] != tail){
            temp_maxlevel = result;
            while(temp_maxlevel < maxlevel && head.next[temp_maxlevel+1] != tail){
                temp_maxlevel = temp_maxlevel + 1;
            }
        }
        else{
            break;
        }
        (address[cas1], oldvalue[cas1], newvalue[cas1]) = (&curr_maxlevel, result,
temp_maxlevel);
    }while(CASN(1, address[], oldvalue[], newvalue[]) == FALSE);
    return TRUE;
}
```

Algorithm 5.9: The skip list 'find' operation: a helper function used by add and delete operations.

```
(Node [], Node [])find(Node head, int key){
    int bottomlevel = 1;
    Node[] preds, currs = (Node[]) new Node[maxlevel]; //maximum level in the skip list
permitted.
    Node pred = null, curr = null;
    pred = CASNRead(&head); //start from the head of the skip list
    for(int level = maxlevel; level >= bottomlevel; level–){ //start from maximum level.
        while(TRUE){
            curr = CASNRead(&pred.next[level]);
            if(curr.key >= key){
                break;
            }
            pred = curr;
        }
        (preds[level], currs[level]) = (pred, curr);
    }
    return (preds, currs);
}
```

5.2.2 Deleting a Node in a Skip List

The delete operation will remove a key supplied to it, if the node is already present in the skip list (Figure 5.6). A successful delete operation requires the predecessor of target nodes next reference updated with the successor of target node in each of the levels starting from the top level. Similar to add, delete will invoke find() operation to retrieve the set of probable predecessors and successors (stored in preds[] and currs[] arrays) at each level of the skip list. For deletion to move ahead, the target key must be present in the skip list and recorded in currs[] array. The predecessors next pointers at each level are updated at once using CASN operation. Also, the deleted nodes next pointers are pointed backwards to point to its predecessor [16]. The reason to do this is to ensure that no other nodes are added after this deleted node and search operations traversing through this node can follow the back path to return to the original list. Using CASN as a primitive will ensure that the delete operation is non-blocking. Let us consider the example to illustrate the delete operation described in figure 5.6.

The deletion of a node starts with the invocation of find operation which returns arrays of predecessors and successors for the supplied key. If the key is present in the skip list, all the currs[] array values contains the key to be deleted. If the find call does not return the currs values with the key searched for, the node is not present in the skip list and deletion fails. After the set of preds[] and

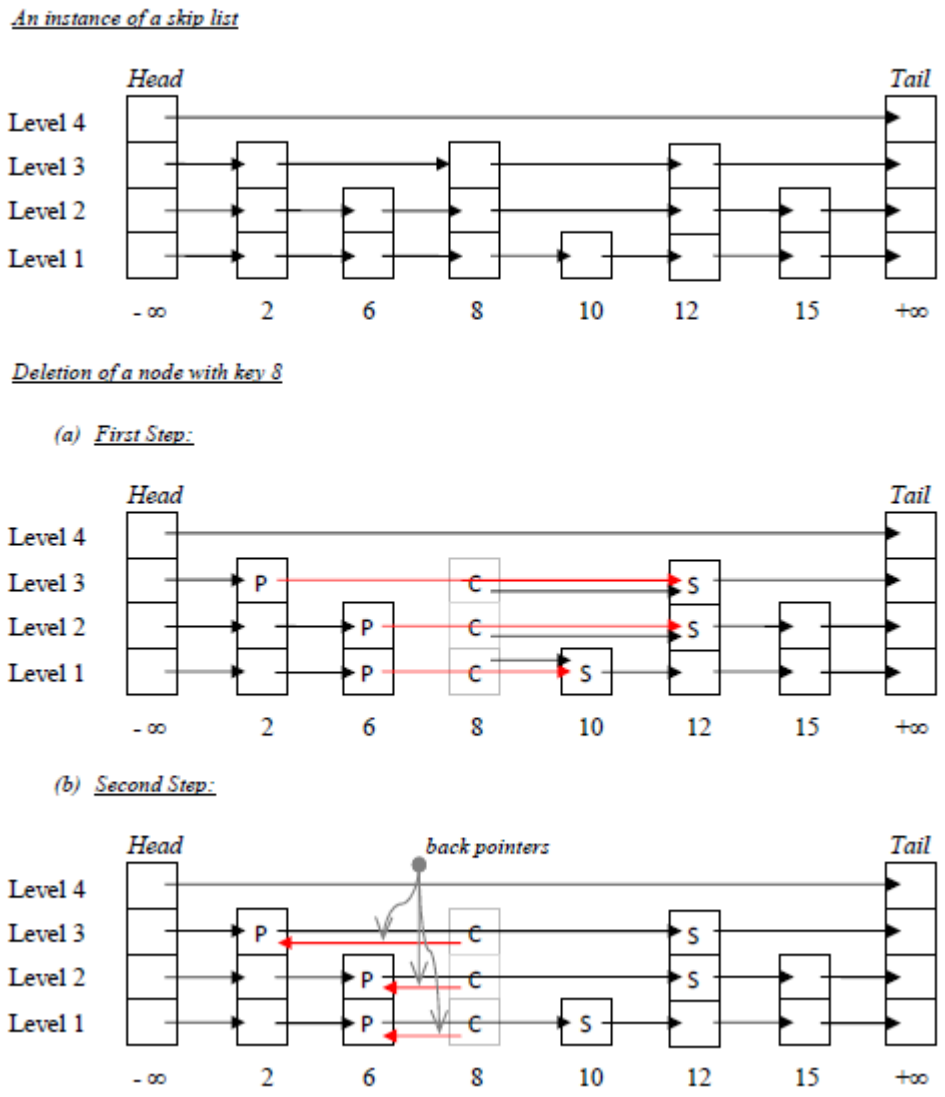

Figure 5.6: An instance of a skip list with delete(8) invocation. In fig (a), find fetches the set of predecessors and successors (current node in this case) stored in preds[] and currs[] array. Also, an arrysuccs[] is used to store the successors with keys just greater than the current node. For key 8, the set of nodes of preds[], currs[] and succs[] in each level are denoted by P, C and S respectively. The next pointers of the predecessor point to successors. In fig (b), in second step, the currs next pointer is updated to point backwards at each level. The deletion of node is completed.

currs[] arrays are calculated and if the key is present in the skip list, we use the CASN operation to update all the set of pointers. The pseudo code to implement delete operation is given in algorithm 5.10.

Algorithm 5.10: Pseudo code to implement 'delete' operation in a skip list.

```
bool delete(int key){
    int bottomlevel = 1;
    int newlevel = 0;
    int cas1 = 0;
    Node[] preds, currs, succs = (Node[]) new Node[maxlevel];
    do{
        (preds[], currs[]) = find(head, key);
        if(currs[bottomlevel].key != key){
            return FALSE;
        }
        else{
            Node node = currs[bottomlevel];
            for(int level = bottomlevel; level <= node.toplevel; level++){
                Node succs[level] = CASNRead(&currs[level].next[level]);
                (address[level], oldvalue[level], newvalue[level]) =
(&preds[level].next[level], currs[level], succs[level]);
                (address[newlevel+1], oldvalue[newlevel+1], newvalue[newlevel+1]) =
(&currs[level].next[level], succs[level], preds[level]); //back pointing
                newlevel = newlevel + 2;
            }
        }
    }while(CASN(2*node.toplevel, address[], oldvalue[], newvalue[]) == FALSE);
//update the current max level.
    do{
        result = CASNRead(&curr_maxlevel);
        if(result > 1 && head.next[result] == tail){
            temp_maxlevel = result;
            while(temp_maxlevel > 1 && head.next[temp_maxlevel] == tail){
                temp_maxlevel = temp_maxlevel - 1;
            }
        }
        else{
            break;
        }
        (address[cas1], oldvalue[cas1], newvalue[cas1]) = (&curr_maxlevel, result,
temp_maxlevel);
    }while(CASN(1, address[], oldvalue[], newvalue[]) == FALSE);
    return TRUE;
}
```

The pointers update starts with installing the CASN descriptors in all the memory locations to be updated such that no intervening threads will muddle with the invoked delete process. If all the pointers are successfully installed, CASN will move ahead with updating the pointer, all at once. If CASN is unable to install descriptor in any of the memory locations due to many possible reasons (concurrent threads deleting the curr node, new node added in between, etc.), all the memory locations where the descriptors are installed are updated with old values and the delete operation

retries. In figure 5.6, in case of successful installation of descriptor pointers, the update of set of pointers in part a and b both are executed all at once using CASN. The operation is lock-free because of the use of this atomic primitive. If all the memory locations are updated correctly, the node will no longer be reachable in the skip list and the deletion process returns with a true value.

5.2.3 Searching a Node in a Skip List

The search operation returns boolean result depending upon if the node with the target key present in the skip list (Algorithm 5.11). Searching of a key value in the skip list starts from the head of the current maximum level (Figure 5.7). Since we are not searching for a node from the maximum level of the list and only from the current maximum level which depends on the data stored in the skip list, a lot of unnecessary comparisons are avoided. This idea was suggested by Pugh [16] and we used it in our skip list design. Since higher level lists are sparser than the lower level lists, a large amount of keys will be skipped as we move down towards the bottom level. Because the number of nodes in each higher level of skip list decreases exponentially with the level decreasing , a key can be quickly found by searching first at higher levels, skipping over large numbers of nodes, and progressively working downward until a node with the desired key is found, or the bottom level is reached. We have discussed this already that the expected time complexity of search operation is logarithmic in the length of the list [2, 13, 21, 23]. An example of searching a node in a skip list is given below:

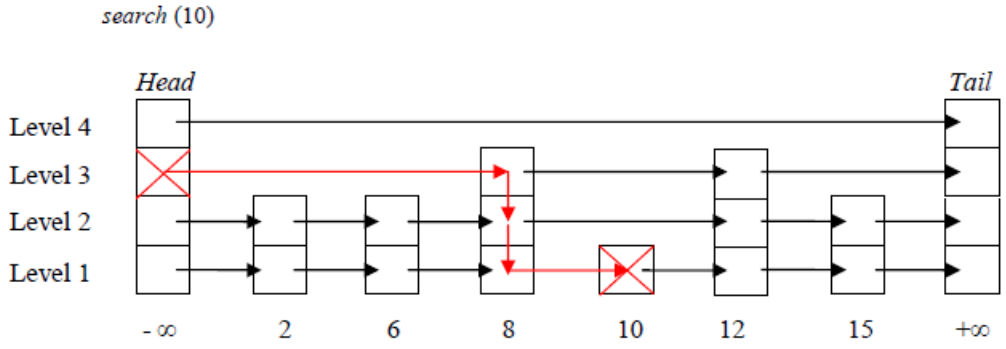

Figure 5.7: Searching a node in skip list. The search starts from the maxlevel and descends down each level until the sought after key is found.

However, the search operation we use here is not wait-free as implemented in [2, 10, 18]. But this operation is non-blocking, lock-free operation. The reason this operation is lock-free and not wait-free is both because of the use of atomic primitive CASN. As the operation advances from the

head to all other nodes across and down the levels, reading of the memory locations needs to be done with CASNRead operation. This is because other concurrent thread is already operating on that memory location. So, we will help that thread complete first and then place our CASN descriptor to complete the search. Hence, this operation is lock-free but not wait-free. The pseudo code to implement search operation is given as:

Algorithm 5.11: Pseudo code to implement 'search' operation in a skip list.

```
bool search(int key){
    int bottomlevel = 1;
    Node[] pred, curr, succ = (Node[]) new Node[maxlevel];
    pred = CASNRead(&head);
    curr_maxlevel = CASNRead(&curr_maxlevel);
    for(int level = curr_maxlevel; level >= bottomlevel; level–){
        curr = CASNRead(&pred.next[level]);
        while(TRUE){
            succ = CASNRead(&curr.next[level]);
            if(curr.key < key){
                pred = curr;
                curr = succ;
            }
            else{
                break;
            }
        }
    }
    return (curr.key == key); //checking only in bottomlevel list is sufficient
}
```

5.2.4 Prepending a Node in a Skip List

This operation inserts a new node at the beginning of the skip list satisfying the condition that the key to be inserted must be the least among all keys in the skip list (Algorithm 5.12. Since the skip list has to be in sorted order, the key to be added must be least of all the keys. Otherwise, this operation will fail. The operation starts with storing all the next references of the head in all levels in an array. The bottom level list contains all the items. So, if the first node of the list at the bottom level is greater than the supplied key to be added, the operation advances. A new node is created with a randomly generated level. The new node's next pointers are updated with the head's next references which are already stored in an array. We don't need to ensure atomic updates here because the node is not yet added to the skip list and if any other concurrent operations changed something in between; we will track it in CASN update and retry prepending. All the heads next field update is carried out using CASN operation. The CASN operation moves forward with

installing the descriptor pointers in each of the memory locations to be updated. If nothing has changed, i.e. if no other nodes has been added between the head and first node or the first node is not deleted, than installing descriptors at all levels (only upto top level needed) will succeed and CASN will proceed with atomic update of all the memory locations. If other concurrent threads changed something in between, then CASN will fail and prepend operation retries. We can see that this operation is also lock-free. If other processes are working on the memory locations it is trying to place its descriptor, it will help them complete first and then move ahead with own process. The use of CASN primitive allows the lock-free behavior of this operation. We will also need to update the current maximum level variable for this operation similar to add operation. The pseudo code to implement prepend operation is given below:

Algorithm 5.12: Pseudo code to implement 'prepend' operation in a skip list.

```
bool prepend(int key){
    int toplevel = randomlevel(); //assign a random level to new node
    int bottomlevel = 1;
    Node[] firstnode = (Node[]) new Node[maxlevel];
    do{
        for(int level = maxlevel; level >= bottomlevel; level–){
            firstnode[level] = CASNRead(&head.next[level]);
        }
        if(firstnode[bottomlevel].key <= key){
            return FALSE;
        }
        else{
            Node node = new Node(key, toplevel);
            for(int level = bottomlevel; level <= toplevel; level++){
                node.next[level] = firstnode[level];
                (address[level], oldvalue[level], newvalue[level]) = (&head.next[level],
firstnode[level], node);
            }
        }
    }while(CASN(toplevel, address[], oldvalue[], newvalue[]));
    return TRUE;
}
```

5.2.5 Appending a Node in a Skip List

This operation inserts a new node at the end of the skip list satisfying the condition that the key to be inserted must be the greatest among all keys in the skip list (Algorithm 5.13). Since the skip list has to be in sorted order, the key to be added must be greatest of all the keys. Otherwise, this operation will fail. Analogous to searching the last element in the skip list, this operation needs to go all the way to the end of the list to verify if the last element of the skip list is greater than the

intended key to be added. If the last element of the skip list is greater than the key to be added, this operation will fail. Pseudo code to implement append operation is given as:

Algorithm 5.13: Pseudo code to implement 'append' operation in a skip list.

```
bool append(int key){
    int toplevel = randomlevel();
    int bottomlevel = 1;
    Node[] lastnode, pred, curr = (Node[]) new Node[maxlevel];
    pred = CASNRead(&head);
    do{
        for(int level = maxlevel; level >= bottomlevel; level–){
            while(TRUE){
                curr = CASNRead(&pred.next[level]);
                if(CASNRead(&head.next[level] == CASNRead(&tail))){
                    curr = pred;
                    break;
                }
                if(CASNRead(&curr.next) == CASNRead(&tail)){
                    break;
                }
                pred = curr;
            }
            lastnode[level] = curr;
        }
        if(lastnode[bottomlevel].key >= key){ return FALSE; }
        else{
            Node node = new Node(key, toplevel);
            for(int level = bottomlevel; level <= toplevel; level++){
                node.next[level] = CASNRead(&tail);
                (address[level], oldvalue[level], newvalue[level]) = (lastnode[level].next,
CASNRead(&tail), node);
            }
        }
    }while(CASN(toplevel, address[], oldvalue[], newvalue[]) == FALSE);
    return TRUE;
}
```

An array will store the entire last nodes (node just before the tail pointer) across all levels of the skip list. If the list is empty, the head of the skip list will act as the last node of the skip list. This is because the new key to be added needs to be spliced in between the last node and the tail sentinel node. If the last node at the bottom level list is less than the supplied key, appending proceeds with the creation of a new node with a random level. The new nodes next references are updated will the tail up to the top level of new node. To update the next reference of the last nodes at each level, we will use CASN operation. The operation begins with installing the descriptors of the invoked

CASN operation in next fields of the last nodes. If anything changed after the invocation of the CASN or any other previously was working on the same location, CASN will help that operation to complete first and then install its own descriptor pointers. If the memory locations were updated in between, CASN will fail and the operation retries from the start. If nothing bad had happened and after all the references to the descriptor pointers are installed, CASN will execute all the update at once, verifying nothing has been changed in between. This will complete the appending of new node with the highest key at the end and the operation terminates. The use of CASN primitive allows the lock-free behavior of this operation. We will also need to update the current maximum level variable for this operation similar to add operation.
To summarize the operations presented above on skip list which can also be viewed as an implementation of linked list stacked at multiple levels,

- The add(x) method adds x to the skip list, returning true if and only if x was not present in the list.
- The delete(x) method deletes x from the skip list, returning true if and only ifx was present in the list.
- The search(x) returns true if and only if the skip list contains x, else returns false.
- The prepend(x) method adds a node at the beginning of the skip list, just after the head sentinel node.
- The append(x) method adds a node at the end of the skip list, just before the tail sentinel node.

All these operations are implemented to work in a lock free manner.

5.2.6 Proof of Concurrency

To prove the correctness of the algorithms of the operations presented above, let us demonstrate few sample executions of the various operations and analyze each of the invocations and responses of the operations in detail. It will be easier to reason about the execution of operations because of the use of the powerful CASN operation in their implementation. Let us consider a scenario described in figure 5.8.

Two threads A and B are trying to add items to the skip list simultaneously. To add an item, a process will invoke CASN operation to update the next pointers of its predecessors to point to itself.

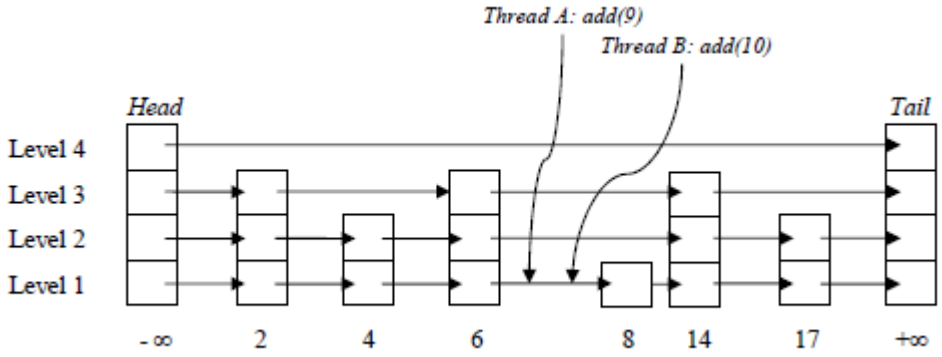

Figure 5.8: Two concurrent threads operating together simultaneously in skip list. Both threads are trying to add an item in the skip list simultaneously.

This is done by installing the thread's CASN descriptor in that location. In the scenario above, let us assume both threads started simultaneously. Thread A and B will invoke add operation and create new nodes and point its next references to its successors. There is no problem up to here. But, both these operations are trying to work on the same memory locations i.e., at their predecessor's next fields (node 8 at level 1,2, 3 and head at level 4). But, we will be safe from these two simultaneous additions because only one of them will be able to install its CASN descriptor at the predecessor's next fields. If thread A was able to install its descriptor, thread B will see that another thread already has a reference to descriptor in that location and help it complete first. After thread A is done adding item 9, it will terminate its operation with a success. Thread B will then start to place its descriptor values in its predecessor's next field (node 8). Thread B while executing CASN will find that the its predecessor's next reference is no longer pointing to the successor (set of preds[] and currs[] returned by find() call for thread B) because a new node 9 has already been added and node 8's next references are pointing to 9. So, CASN for thread B will fail and will retry the add operation by finding a new set of apparent predecessors and successors. So, both the simultaneous operations are safe from intervention of each other. This is due to the powerful CASN operation we are using which makes the operations atomic without using locks.

Now, let us consider another scenario where two threads are invoking a delete and an add operation exploiting same memory addresses as described in figure 5.9.

Two threads are operating concurrently. Thread A is trying to remove key 8 from the skip list and thread B is trying to add an item 9 in the skip list. With the lock-free implementation

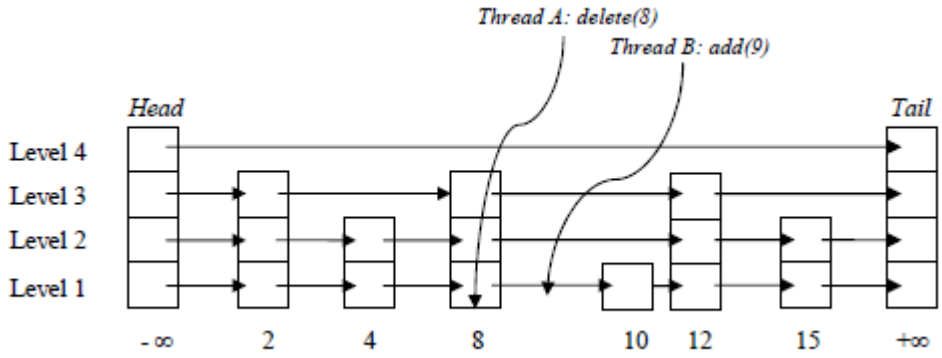

Figure 5.9: Two concurrent threads A and B operating together simultaneously. Thread A is trying to delete an item from the list and thread B is trying to add an item at the same time.

using CASN primitive described above, this will not be a problem. Let us assume add(9) operation started first. And just before updating the predecessors next reference to itself (8s next reference to itself), a delete(8) operation is invoked. After thread A has installed all its descriptor pointers in node 8s next references, deleting that node will fail because it requires updating the deleted nodes next pointer to point to its predecessor. If delete operation finds descriptor pointers already in the memory location it is trying to update, it will help the operation complete first (help thread B). So, add will win in this situation. If the delete operation which began already and succeeded in installing descriptors in its predecessor nodes will redo the installing because a new node has just been added and the next reference of the deleted node has changed. Similarly, if delete(8) was able to install its descriptor pointers first in all the memory locations to be updated, add(9) will find that the node 8s references have changed to point backwards, i.e. the node is deleted and add operation retries again. Hence, this simple example shows that in any case none of the operations invoked by any number of processes will interfere with each other because all the operations use powerful CASN primitive and so add, delete, prepend and append operation are all lock free operations.

Chapter 6

Conclusion and Future Work

In this report, we have discussed the implementation of various atomic primitives like CAS, DCAS and CASN. We explored various alternatives for implementation of these atomic primitives and explored different ideas to implement a non-blocking CASN operation. CASN being the most powerful atomic primitive can update a number of memory locations atomically. We presented simple examples to understand and implement the CASN atomic operation using traditional CAS operation which are found many of the current multi-processor systems.

We proposed the non-blocking, lock-free implementation of various operations on linked list using CASN and demonstrated the proof of correctness of the algorithms for various operationsconsidering different scenarios and presenting various examples. The lock-free linked lists thus implemented served as each levels of skip list. Using the implementation of linked list, various operations on skip lists were implemented. In a skip list, to insert, delete, append and prepend a node, a number of memory locations needs to be worked on to ensure the operation worked correctly. For a data structure like this, the powerful CASN operation fits best. The nature of CASN operation complements the data structure like skip lists. We explored the various techniques to implement skip lists and borrowed ideas to complete our design. We implemented our own operations and proposed various algorithms.

The linked list and skip lists algorithms we presented are designed to be lock-free and work with multiple concurrent processes. Because of the nature how CASN works, we were unable to make the search operation wait-free, but it is non-blocking nonetheless. However, because most of the modern multi-processor systems only support the traditional compare and swap operation, the implementation of CASN operation as a machine instruction in operating system is yet to be implemented.

The append and prepend operations have their unique significance. We are considering sorted lists in our implementation. With a small modification, we can implement these operations to insert items at the first and last positions of the list (not considering the key values). Prepending a node ensures that an item is inserted only at the beginning of the list. Removing this node from the list requires O(1) time. So, this behavior will provide us with a stack of Last In First Out (LIFO)nature. Similarly, appending an item will serve as a queue with First In First Out (FIFO) behavior.

Skip lists are logarithmic with high probability and are designed for fast searching. However, skip lists are not to be considered as a complete replacement of balanced trees. Skip lists require more comparisons than binary search trees. Due to the extra comparisons, skip lists are not always faster than binary search trees on average. Skip lists use multiple pointers and multiple copies of the data for the same item. So it requires a lot more space than balanced search trees. But because simpler data structures like linked lists and arrays are used to construct skip lists, they are easier to implement, optimize and experiment with than their other tree based cousins.

There are various areas to broaden the work presented in this report. Using the position of nodes in the skip list, a variety of useful operations can be implemented. Some of them could be search a node by position, insert a node by position or delete a node by position of a node in the skip list data structure. A very careful thought is necessary to implement operations like these. Because the environment is concurrent and threads can work independently on disjoint memory locations [1], determining the position might be hard task. Coarse grained locking might be one blocking solution for this. Non-blocking techniques are to be determined.

The similar idea can be used to implement various other data structures. The CASN operation used can offer very simple designs for complex structures like binary trees and other balanced tree data structures.

Another area to extend this work will be to implement this algorithm in a programming language, mathematically derive the proof of correctness of the algorithms proposed, perform experimental estimations and compare the performance of this algorithm with other algorithms.

Bibliography

[1] A. Israeli and L. Rappoport, "Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives". *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pp. 151-160, 1994.

[2] M. Herlihy and N. Shavit, "The Art of Multiprocessor Programming". *Morgan Kaufmann Publishers Inc.*, 2008.

[3] M. Herlihy, "Wait-free synchronization". *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 124-149, 1991.

[4] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs". *Proceedings of the 13th IEEE Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pp. 185-192, 2010.

[5] H. Sundell, "Wait-free multi-word compare-and-swap using greedy helping and grabbing". *International Journal of Parallel Programming*, vol. 39, no. 6, pp. 694-716, 2011.

[6] P. H. Ha and P. Tsigas, "Reactive multi-word synchronization for multiprocessors". *Journal of Instruction-Level Parallelism*, vol. 6, 2004.

[7] J. H. Anderson and M. Moir, "Universal Constructions for Multi-Object Operations". *Proceedings of the 14th Annual ACM Symposium on the Principles of Distributed Computing*, pp. 184-193, 1995.

[8] T. Harris, K. Fraser, and I. Pratt, "A practical multi-word compare-and-swap operation". *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.

[9] M. L. Scott, "Shared-Memory Synchronization". *Morgan & Claypool*, 2013.

[10] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A simple optimistic skiplist algorithm". *Proceedings of the 14th international conference on Structural information and communication complexity*, pp. 124-138, 2007.

[11] J. D. Valois, "Lock-free linked lists using compare-and-swap". *Proceedings of the 14th Annual ACM Symposium on the Principles of Distributed Computing*, pp. 214-222, 1995.

[12] T. Harris, "A pragmatic implementation of non-blocking linked-lists". *Proceedings of the 15th International Symposium on Distributed Computing*, pp. 300-314, 2001. Available as www.google.com/patents/US7117502.

[13] M. Herlihy, Y. Lev, and N. Shavit, "A lock-free concurrent skiplist with wait-free search". *Unpublished Manuscript, Sun Microsystems Laboratories*, 2007. Available as www.google.com/patents/US7937378.

[14] K. Fraser and T. Harris, "Concurrent programming without locks". *ACM Transactions on Computer Systems*, vol. 25, no. 2, 2007.

[15] D. Deharbe, L. Fejoz, P. Fontaine, and S. Merz., "Verified Incremental Development of Lock-Free Algorithms".

[16] W. Pugh, "Concurrent Maintenance of Skip Lists". *University of Maryland at College Park*, 1990.

[17] K. Fraser, "Practical lock freedom". *Ph.D. thesis, University of Cambridge*, 2003.

[18] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A provably correct scalable concurrent skip list". *Proceedings of the 10th International Conference On Principles Of Distributed Systems*, 2006.

[19] Lea, D, ConcurrentSkipListMap. In java.util.concurrent.

[20] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees". *Communications of the ACM*, vol. 33, no. 6, pp. 668-676, 1990.

[21] W. Pugh, "A skip list cookbook". *University of Maryland at College Park*, 1990.

[22] B. C. Dean and Z. H. Jones, "Exploring the duality between skip lists and binary search trees". *Proceedings of the 45th annual ACM southeast regional conference*, 2007.

[23] Wikipedia contributors, "Skip list". Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Skip_list.

[24] Eternally Confuzzled, Retrieved from http://eternallyconfuzzled.com/tuts/datastructures/jsw_tut_skip.aspx.

[25] P. E. McKenney, M. M. Michael, J. Triplett, and J. Walpole, "Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory". *Proceedings of the 4th workshop on Programming languages and operating systems*, pp. 1-5, 2007.

[26] M. Herlihy, "A methodology for implementing highly concurrent data objects". *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 5, pp. 745-770, 1993.

[27] H. Sundell, "Efficient and Practical Non-Blocking Data Structures". *PhD thesis, Chalmers University of Technology*, 2004.

[28] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele Jr., "Lock-free reference counting". *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pp. 190-199, 2001.

[29] M. Moir and N. Shavit, "Concurrent data structures". *Handbook of Data Structures and Applications, D. Metha and S. Sahni, eds. Chapman and Hall/CRC Press*, 2007.

[30] Eternally Confuzzled, Retrieved from http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_bst1.aspx

[31] M.B. Greenwald, "Non-Blocking Synchronization and System Design". *Ph.D. thesis, Stanford University*, 1999.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Anish Ratna Tuladhar

Degrees:

Bachelor of Computer Engineering 2009
Tribhuvan University
Kantipur Engineering College, Nepal

Thesis Title: Concurrent Non-blocking Skip List Using Multi-word Compare and Swap Operation

Thesis Examination Committee:

Chairperson, Dr. Ajoy K. Datta, Ph.D.
Committee Member, Dr. Lawrence L. Larmore, Ph.D.
Committee Member, Dr. John Minor, Ph.D.
Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.