12-1-2016

Novel Non-Blocking Approach for a Concurrent Heap

Rashmi Niyolia
University of Nevada, Las Vegas, rashmi.niyolia@gmail.com

NOVEL NON-BLOCKING APPROACH FOR A CONCURRENT HEAP

By

Rashmi Niyolia

Bachelor of Science (Hons) - Computer Science
University of Delhi
2008

Master of Science - Software Engineering
Birla Institute of Technology and Science
2012

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

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

University of Nevada, Las Vegas
December 2016

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

November 15, 2016

This thesis prepared by

Rashmi Niyolia

entitled

Novel Non-Blocking Approach for a Concurrent Heap

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.
Examination Committee Chair

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

John Minor, Ph.D.
Examination Committee Member

Ju-Yeon Jo, Ph.D.
Examination Committee Member

Emma Regentova, Ph.D.
Graduate College Faculty Representative

Abstract

We present a non-blocking algorithm for a concurrent Heap in asynchronous shared memory multiprocessors. Processors on these machines often execute instructions at varying speeds and are subject to arbitrarily long delays. Our implementation supports Non-blocking FindMin, Delete, and Insert operations on heap. Non-blocking techniques avoid the drawbacks associated with mutual exclusion and admit improved parallelism. Insert and Delete operations in Heap take more than one atomic instruction to complete, thus, it is possible that a new operation may be started before the previous one completes, leaving heap inconsistent between operations. We have represented Heap as an array of pointers. Any modifications to the heap are done by using single-word Compare&Swap instructions. If all update operations modify different parts of the heap, they run completely concurrently. Our approach guarantees lock freedom for all concurrent threads and wait freedom if threads execute operations on different nodes of heap.

Acknowledgement

I would like to express my deepest sincere gratitude to my supervisor, Dr. Ajoy K. Datta, for his tremendous guidance, motivation, support and for understanding my complicated schedule. His encouragement has been instrumental behind me successfully completing my thesis in such an important topic in the field of multi-core computing. During the course of working with him, I learnt valuable skills like diligence, time management, problem solving which are sure to be helpful throughout the life.

I would also like to thank Dr. Emma E. Regentova, Dr. John Minor and Dr. Ju-Yeon Jo for investing time in reviewing my report and their willingness to serve on my committee.

I am extremely thankful to my family for their unconditional support. My husband, Arjun, has been a cornerstone in this journey of pursuing further education. He has been supportive in any way he could, so I could spend more time on conducting research and writing the thesis. Both set of parents, who constantly motivate me to pursue my dreams. Besides, I thank my friends who made life easier here in Vegas and those back home who always had my back when I needed the motivation to go on.

Table of Contents

List of Figures

Chapter 1

Introduction

We are living in a world of concurrency revolution where Multi-core architectures and multi-threading are a need of the hour rather than a 'good-to-have' factor. Clock frequency has hardly advanced in recent years, and focus has shifted to packing more execution cores into a single chip.

What is multicore technology? In the early years of the first decade of 21st century, desktop PCs hit a performance wall. Intel thought that it could keep raising the clock speeds and address the cooling issues, but 'Heat wall' sabotaged its plans on the way to 4GHz. The heat wall required a heat sink and fan almost as big as a power supply to keep the CPU cool. Even experiments with liquid nitrogen and antifreeze coolants did not prove to be of much help [28]. This lead to the idea of going multicore, and to speed up PCs by dividing the tasks among cores.

Multi-core means having several cores on one chip, each running several threads in parallel. When there is more than one core available, a scheduler can give processes slices of time on different CPUs. This way a dual-core processor allows two things to happen at once. The more the number of cores, the more processes can run simultaneously.

As per Wikipedia report [18], until 2005 single-core processors outnumbered multi-core processors. In the second half of 2006 the best processors were dual-core processors. Since 2006 the development has gone on, so that the new processors get four or more independent

microprocessors. Today, single-core processors are popular only in embedded systems [18]. Multi cores are commonplace these days in desktop computers, laptops and mobile phones.

Although driven by hardware changes, it calls for concurrency in software to fully exploit the throughput gains of multi-core processors. Modern software relies on high-level data structures. A data structure is a set of rules on how to organize data units in memory in a way such that specific operations on that data can be executed more efficiently. With the advent of multicore processors, concurrent data structures are required to coordinate access to shared resources. These data structures allow multiple threads to access same data simultaneously [12]. This necessitates contention management between simultaneously active threads for operations to produce correct results and data structure to stay valid. Designing such data structures is challenging compared to sequential ones as concurrent threads may have interleaving steps leading to unexpected and potentially incorrect outcomes. Adding concurrency to a sequential data structure is bound to make it more complex and less efficient. However, the ubiquity of multicore systems makes concurrent data structures inevitable.

The simplest step towards making a sequential data structure concurrent is by means of locking [11]. Locking provides the lock owner an exclusive access to data structure [4]. A more sophisticated version of locking is fine-grained locking which locks only some part of the data structure. These approaches are called 'blocking' and have several issues as explained in Section 2.3. These issues are abated by 'non-blocking' implementation of data structure. In a non-blocking algorithm, failure or delay of one thread does not suspend progress of another thread [11].

The subject of this thesis is one such fundamental data structure - HEAP. It is a tree based data structure widely used in heapsort, selection algorithms, graph algorithms and priority queues [37]. It can be classified as max heap or min heap based on the ordering of nodes. If parent nodes have higher key-values than child nodes, the heap is called a max-heap. The top-most node (root) of a max-heap contains the highest value. If parent nodes contain lower values than child nodes, it is known as a min-heap. The root node of a min heap contains the lowest value in the heap. Throughout this thesis, we will use min-heap for our illustrations and examples. Hence, whenever we use the word 'heap', it refers to a min-heap. There are three main operations in heap: insert, delete and find-min. Insert adds a new node to the heap. The delete operation removes root node from heap and returns its value. Find-min operation returns the value of root node without actually removing it from heap.

An interesting feature of insert and delete operations on heap is that they are a sequence of many sub-operations, called bubble-up and bubble-down. After insertion/ deletion, nodes are moved up/down the heap. It means that a single operation can potentially access multiple nodes on the heap. In a concurrent heap, where multiple threads are working concurrently, this poses an impediment to maintaining a consistent state of the heap. We present a non-blocking implementation of a concurrent heap which is linearizable and immune to conflicts to the best of our knowledge.

1.1 Contribution

We present a non-blocking implementation of a heap whose operations make changes in heap atomically in an asynchronous shared memory system. Our implementation offers *lock-free*

insert, delete and *wait-free* find-min operations. The Insert operation adds a new node to the heap. The Delete operation removes the root node from the heap and returns its value. Find-min operation returns the value of the root node without actually removing it from the heap.

We have represented the heap as an array of pointers as opposed to traditional representation as an array. Any updates to shared data structures are made atomically by changing one of the pointers using *CAS*. Potential conflicts in operations when multiple threads are trying to access the same node are confronted by usage of *flagging* and *yielding*. The concept of flagging enables threads to help other delayed operations or crashed threads. This further ensures that as long as there is an active thread in the system, pending operations on its way would be completed and the system would make progress. We have provided a high level algorithm and detailed pseudo-code followed by proofs for atomicity, conflict resolution and linearizability.

1.2 Outline

In Chapter 2 we will provide a background in multi-core systems. We will give an overview of shared memory systems and key concepts of multithreading, atomicity, execution history and linearizability. We will discuss various synchronization techniques in threads including pros and cons of different techniques. It will also include a brief explanation of blocking and non-blocking systems in terms of lock-free, wait-free and obstruction-free implementations.

The next step is to apply the lessons of Chapter 2 into building concurrent data structures. In Chapter 3 we will study Maurice Herlihy's universal constructions for transforming a sequential

algorithm into non-blocking concurrent one. We will study concurrent implementations of various data structures like queues, lists, doubly-linked lists and binary trees. We will also briefly discuss the work done by various researchers on heaps over the years.

Chapter 4 will give an insight into our work. It will discuss our data representation, tools and techniques followed by the high-level algorithm. We will explain the functions in our heap algorithm in detail. Viable conflicting scenarios will be illustrated with expected outcomes. Chapter will conclude with detailed the pseudocode for our algorithm.

Chapter 5 consists of invariants and proofs for atomicity, conflict resolution and linearizability of our proposed approach.

We will conclude the thesis with Chapter 6 which will give our final words on the work and scope of improvement or future recommendations for work in this direction.

Chapter 2

Background

The definition of modern systems that support concurrent programming is pretty intricate. The basic one being the number of processors in the system, making it uni(single)-processor or multiprocessor. Multiple systems regarded as computation nodes can be connected despite physical separation to form a distributed system. These computation nodes can be connected via a message passing system or a shared memory system.

2.1 Shared Memory System

In message passing system, communication between various nodes is done by exchanging information packages over the inter-connection network [11]. A shared memory system presents a higher level of abstraction, and gives the impression of existence of a global memory for all nodes, with shared access possibilities. In this thesis, we will concentrate on shared memory systems.

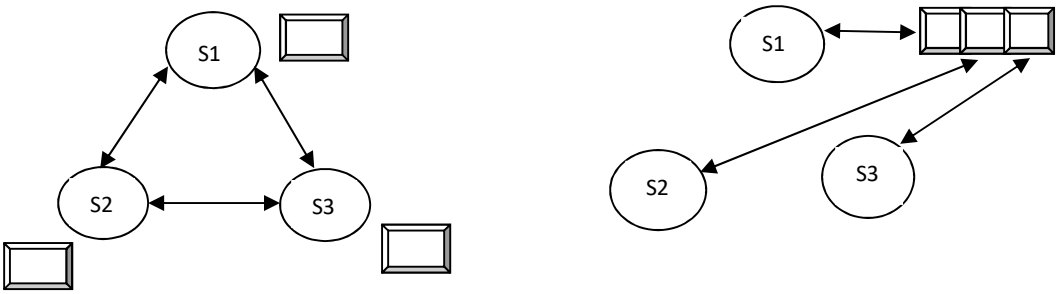

Figure 2.1 Message passing vs Shared memory (for three processes S1, S2, and S3)

A shared memory system can be Uniform Memory Access (UMA) architecture or Non-Uniform memory Access (NUMA) architecture. In UMA, shared memory appears to be equidistant from each processor, thus, response time for memory access is the same all over the system. In NUMA, memory access response time varies in the system depending on the actual distance between a processor and real memory.

2.2 Multicore Systems

A multicore processor is a single computing component with more than one independent processor, referred to as cores. They can execute several instructions simultaneously, thereby, increasing overall speed of the system. Adding more cores alone does not guarantee superior functionality; as asserted by Amdahl's law, gain in performance is largely driven by how much of the code can be parallelized [3]. As these concurrent processes work on a shared memory, they pose more problems like race conditions and cache coherence.

Definition 2.1 Race condition: The situation when multiple processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition [4].

Definition 2.2 Cache Coherence: A consistent view of individual cache lines containing data from a shared resource. An update to value of a variable in one cache must be immediately reflected in all other caches where the variable resides [4].

It is vital that processes are well synchronized to avoid race conditions. We will learn more about this in next section.

2.3 Concurrent data structures

Concurrent data structures allow multiple processes to access and update data simultaneously on shared memory systems [12]. Asynchronous processes and interleaving executions increase complexity while designing such data structures. Well-founded conflict resolution rules must be set to avoid unexpected outcomes. In this section, we will discuss various aspects of designing concurrent data structures, like synchronization techniques and correctness verification.

2.1.1 Synchronization techniques

Synchronization techniques for a concurrent system are categorized into two types – blocking and non-blocking [11].

2.3.1.1 Blocking

The traditional approach to regulate access to shared data is by means of locking. Locking is a traditional synchronization mechanism which provides the lock owner an exclusive access of a shared resource [4]. Software constructs like Semaphores, Monitors, Mutex are examples of low-level locking primitives. Using a single lock for the entire shared object is called Coarse grained locking [11]. It is easier to implement, with no room for unintended interference, but it blocks system parallelism. Fine grained locking supports parallelism as it provides a process exclusive access only to a part of the shared object, although the same feature makes it more difficult to implement. Lock based synchronization is poorly suited for asynchronous, fault-tolerant systems. One faulty process can block or slow down the entire system. Locking can lead to error conditions such as deadlock, livelock, convoying and priority inversion.

Definition 2.3 Deadlock*: A s*tate where two or more processes are waiting indefinitely for an event (resource acquisition or release) that can be caused only by one of the waiting processes. When such a state is reached, these processes are said to be deadlocked [4].

Definition 2.4 Livelock: A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, with none progressing [5].

Definition 2.5 Starvation: A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen [18].

A lock-based implementation is called blocking because an unexpected delay by one thread can prevent others from making process. Disadvantages of blocking are curbed by non-blocking mechanisms.

2.3.1.2 Non-Blocking

A concurrent object implementation is non-blocking if it guarantees the system as a whole will make progress despite individual halting failures or delays. It can be classified into three categories.

Lock free – An implementation is lock-free if it satisfies that when the program threads are run sufficiently long at least one of the threads makes progress. It allows individual threads to starve but guarantees system-wide throughput [11].

Obstruction free – In an obstruction-free implementation, if at a certain point, a process executes in isolation and finishes in a finite number of steps from that point [11].

Wait free – A wait free implementation guarantees that every process will complete its execution within a finite number of steps [13].

Wait-freedom is the strongest non-blocking guarantee of progress [11]. A wait-free implementation is necessarily lock-free, but not vice-versa, since a lock free implementation may permit individual processes to starve.

How can we prove the correctness of a concurrent data structure? Is there a standard verification technique using which programmers can specify and reason about concurrent objects? The answer to these questions is Linearizability.

2.4 Linearizability

Linearizability is the cardinal correctness condition for concurrent systems. Maurice Herlihy and Jeanette Wing [2] introduced the notion of Linearizability in 1990. It is the standard technique for demonstrating that a non-blocking implementation of an object is correct.

In a sequential system, where an object's operations are invoked one at a time by a single process, the meaning of the operations can be given by pre and post conditions. In a concurrent system, simultaneously executing processes make this task complex [12]. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can also be given by pre- and post-conditions [11]. Operations that do not overlap take effect in their "real-time" order.

A concurrent system is linearizable if and only if each individual object is linearizable [14]. Linearizability facilitates certain kinds of formal (and informal) reasoning by transforming assertions about complex concurrent behavior into assertions about simpler sequential behavior.

It is a safety property; it states that certain interleaving cannot occur, but makes no guarantees about what must occur [2].

2.5 Compare and Swap (CAS)

Modern multiprocessors provide hardware support of atomic read-modify-write operations in order to facilitate inter-thread coordination and synchronization. A C*ompare-and-swap* (CAS) operation is often the synchronization primitive of choice for implementing concurrent data structures – both lock-based and non-blocking [1] – and is supported by hardware in most contemporary multiprocessor architectures [6]. The CAS operation takes three arguments: a memory address, an old value, and a new value. If the address stores the old value, it is replaced with the new value; otherwise, it is unchanged [34]. The success or failure of the operation is then reported back to the calling thread. CAS is widely available and used because its atomic semantics allows any number of concurrent threads to update shared data without any loss of data consistency.

Literature includes a wide range of atomic instructions like getAndDecrement(), swap(), and many others. Identifying the right one to be used based on synchronization requirements of an algorithm is important. The mechanism to evaluate the power of various synchronization primitives is Consensus. Consensus number is the maximum number of threads for which the consensus problem can be solved.

Consensus

A consensus protocol is a system of n processes that communicates through a set of shared objects [21]. The processes each start with an input value from some domain; they communicate with one another by applying operations to the shared objects; and they eventually agree on a common input value and halt [2].

A consensus protocol is required to be

i. consistent: distinct processes never decide on distinct values
ii. wait-free: each process decides after a finite number of steps
iii. valid: the common decision value is the input to some process

As proven in Herlihy's seminal paper [1], a CAS register has an infinite consensus number. It can implement, together with reads and writes, any object in a wait-free manner.

The correctness of algorithms using CAS often depends on the fact that, if the CAS succeeds, the old value has not been changed since the preceding read [11]. Suppose the value of a variable was A when CAS operation began. Meanwhile, the value was changed by another thread to B and then back to A. In this scenario, CAS can succeed. However, it should not; since value was not A throughout the operation. This is called the ABA problem. A good implementation must have strategies to avoid the ABA problem. Some of the conventional solutions are usage of counters and timestamps. Counter solution does not guarantee that the ABA problem will not occur, but it makes it extremely unlikely [18]. Valois's reference counting technique guarantees preventing the ABA problem without the need for modification counters or the double-word CAS [18].

Chapter 3

Literature review

The increasing popularity of multi-core systems has lead to an opulent amount of research being done in various areas like locking, atomicity, and linearizability. One of the prominent areas is designing concurrent data structures. This field is focused on developing concurrent versions of the fundamental data structures like stacks, queues, search trees, linked lists, heaps, etc., due to their ubiquitous usage. In this chapter, we will look at some of the ground-breaking work in this field along with different flavors of implementations for various data structures. Section 3.1 will present the concept of universal transformations. In Section 3.2, we will have a closer look at various implementations of linked lists. Our thesis has been greatly influenced by N. Shaifei's work [24] in this area. Section 3.3 will showcase existing research on heaps. In Section 3.4, we will provide some references to highlight research on some basic data structures like stacks, queues, search trees, etc.

3.1 Universal Transformation

In one of his pioneering papers [12], Herlihy introduced the notion of universality. He presented a translation protocol that produces equivalent non-blocking concurrent algorithms for given sequential algorithms. For making any changes to the shared object, a copy is made, updates are applied to the copy and then the shared object is replaced with its updated copy using a CAS

operation. This is done as a single atomic operation, ensuring that the new value is computed based on the latest information. This protocol could not steer clear of two issues – the ABA problem and the overhead of copying large objects.

Herlihy then proposed another transformation protocol based on the fact that each data structure is made up of blocks connected by pointers [11]. Only the blocks, which are to be updated or contain pointers to the blocks which are to be modified, need to be copied. This approach also consisted of a few problem areas like lots of copying, the programmer had to incorporate logic for breaking the structure into proper blocks, and it could not be used for efficient transformation of data structures like priority queues implemented as linked lists.

The idea of universal helping was another milestone in wait-free implementation of concurrent data structures. It suggests adding extra code so that the active processes 'help' the blocked processes terminate, so that the resulting implementation is wait-free. an implementation has helping if in some situations, a process makes an undecided operation of another process becomes decided on some value. Attiya, Castaneda, and Hendler presented two approaches to formalize helping in wait-free implementations of shared objects [7]. They proposed that objects for which there are wait-free implementations can be separated from those with only non-blocking implementations, using primitives with a finite consensus number. An implementation using test-and-set showed that any wait-free queue must have nontrivial helping while this is not true for stacks.

3.2 Linked list

A list stores a sequence of items. It is one of the elementary data structures and has many applications in distributed systems including processor scheduling [35], memory management [37] and sparse matrix computations [36]. It is also used as a building block for more complicated data structures such as deques, skip lists and Fibonacci heaps.

The first non-blocking implementation of singly-linked lists using CAS was given by Valois [32]. This implementation made use of 'cursors'. A single cursor points to three consecutive nodes in the list. If the part of the list that the cursor is associated with is changed, the cursor becomes invalidated. To restore the validity of its own cursor, a process may have to perform CAS steps to help complete other processes' updates. One of the drawbacks is that it inserts auxiliary nodes between adjacent nodes, so the list might be longer than it is supposed to be.

Fomitchev and Ruppert [16] also gave a non-blocking singly-linked list implementation using CAS steps. Three CAS steps are performed to remove a node from the list. Similar to [22], each operation searches the list for a node on which to operate. Each node also has a back-link pointer. When a node is deleted, a back-link pointer is set to its last predecessor. If an operation's CAS fails, it uses the back-link pointers to resume its operation. However, these back-link pointers cannot be used to move a cursor to the left since the back-link pointers of only removed nodes are set. They provided a correctness proof and an amortized analysis for their implementation. Timnat [31] used the cooperative technique to present a wait-free singly-linked list using CAS steps. They extended existing implementations by using a wait-free helping mechanism.

Only a few non-blocking implementations have been proposed for doubly-linked-lists. Greenwald [20] presented an implementation using 2-word CAS. In his approach, all processes cooperate to execute a single piece of sequential code. An operation executes a step of the sequential code and increments a shared counter that is equal to the line number of the running code simultaneously using 2-word CAS. When one operation is completed, each process can try to update the program counter to the beginning of the operation it wants to perform next. This implementation does not support any concurrency.

Attiya and Hillel [15] also proposed a 3-coloring doubly-linked list implementation using 2-word CAS, but it only supports update operations. Three ordered colors are assigned to nodes in the list such that any two adjacent nodes have distinct colors. In their implementation, each update operation needs to acquire three consecutive nodes by using CAS steps to write its id into the nodes before performing its update. If two of the nodes have the same color, a 2-word CAS is used to acquire those two nodes. If a process does not acquire a node because it is acquired by another operation, the process first helps the other operation and then continues its own operation. The operation acquires nodes in the order of their colors. One of the main highlights of this implementation is that concurrent operations can interfere with one another only if they are changing nodes close to each other. The ABA problem is avoided by storing both a pointer and a counter in a single word. They also give a restricted implementation using a single-word CAS, in which deletions can be performed only at the ends of the list.

Sundell and Tsigas [30] gave the first non-blocking doubly-linked list using single-word CAS. It made use of cursors to access the list. Shaifei [24] came up with a new approach to design non-blocking, linearizable implementations of shared data structures using the concept of

cursors and info objects. Her approach has been applied to two data structures - doubly-linked lists and Patricia tries. Both implementations are linearizable. Every update operation calls one fairly simple routine to perform the real work. This is the first amortized analysis for a non-blocking doubly-linked list with detailed correctness proofs. Our approach is largely influenced by Shaifei's work in terms of modularity of code and the using concept of info objects for update operations.

3.3 Heap

The common usage of a heap is as a priority queue data structure which maintains a collection (multiset) of items ordered according to a priority associated with each item. Priority queues are used in graph algorithms, discrete event simulation, schedulers and modern SAT solvers. General operations provided by heap are: insert (d, p), which adds a data item d with priority p, and extractMin (), which removes and returns the highest priority data item. Binary heaps are represented as arrays: the root is located at position 1, and the left and right children of the node at location i are located at positions 2i and 2i+1, respectively. Position 0 is not used.

Tamir, Morrison, and Rinetzky [8] proposed Champ, a heap-based concurrent priority queue which allows one to change the priority of an element after its insertion. This allows algorithms like Dijkstra's single source shortest path algorithm to be more efficient. It uses an array based binary heap with the use of tags to identify elements in the queue, instead of their data items. This allows storing multiple elements with the same data item in the queue. First, every operation grabs the locks it requires and inspects, adds, removes, or changes the shared state. Then, it invokes bubbleUp(e) or bubbleDown(e) to locally restore the heap property. With

increased parallelism, Champ's changeKey () operation improves the client overall performance by saving it from doing wasted work. This occurs despite the fact that Champ's extractMin () and insert () operations do not scale as well as in prior designs. Skiplist outperforms Champ since skiplist's insert () and extractMin () are more scalable and efficient than Champ's.

Afek, Barr and Schiff [10] presented the Power priority queue, a recursive and fast construction of an n-element priority queue from exponentially smaller priority queues. The main construction idea is to sort n elements by partitioning into square root n elements into square root n fixed size lists, followed by sorting and merging the lists. It uses a simple in-memory(RAM) FIFO queue called Rlist implemented by linked lists with push and pop operations.

The insert operation on a heap is a bottom-up procedure, while a delete operation is a top-down procedure. Concurrent inserts and deletes can thus create a conflicting or deadlock situation. An algorithm proposed by Rao and Kumar [9] changes the conventional bottom-up insert operation to a top-down procedure which presents a valuable solution to avoid this problem. Each node must be locked before accessing it and unlocked after that. Very often the consecutive insert procedures traverse the same path and this may lead to a race condition when accessing the common nodes.

Ayani [8] proposed the LA-algorithm for concurrent insertions on priority queues. Conventional heap implementations allow insertion of a new node at the next available location in the heap, going from left to right. The LA-algorithm directs any two consecutive insertion requests to two different subtrees and thus provides a possibility to perform multiple insertions in parallel. Performance evaluation on a Sequent Symmetry shared memory multiprocessor

indicated that the algorithm is a promising approach. However, it is a fine-grained locking mechanism and is plagued with the drawbacks of locking.

The first wait-free algorithm to manipulate heaps was a result of Herlihy's work on his methodology for implementing concurrent objects [12]. This has been mentioned in Section 3.1. It requires a thread to check out a pointer to the object, make and change a copy of the object, and place the copy in place of the old object. Note that Herlihy and Shavit also gave a fine-grained locking based implementation of a heap in [11].

Greg Barnes [9] examined heap algorithms and came up with an abstract algorithm for a wait-free implementation. The key feature of his algorithm is his unique representation of a heap - as an array of pointers to nodes. Each node is a record consisting of the key, some flags, and a few auxiliary variables. By using Load Linked and Store Conditional on the pointer to a record, the algorithm atomically checks the entire record for updates. We have borrowed the heap representation aspect of this algorithm for our approach.

Israeli and Rappoport [17] gave a non-blocking implementation which supports deleteMin and insert operations using transactional memory-based atomic primitives. Their approach uses some bits from the node value to store information about the current operation on the node. Thus, even if the process dies, another thread which comes across this node can help it with the operation. This supports the non-blocking property, but also lowers the range of values that can be stored in a node. We have avoided this constraint by storing information in a separate structure.

3.4 Other Data Structures

There has been a lot of interesting work on various data structures. We will mention a few interesting papers here for the reader's reference.

Michael and Scott [36] presented simple lock-based concurrent implementations for both stack and queue. Treiber [35] proposed a lock-free implementation of a stack. It represents a stack using a singly-linked list with top pointer. Modifications to top pointer are done atomically using the CAS atomic instruction. Michael and Scott [33] also gave a two-lock queue algorithm in which one enqueue and one dequeue can proceed concurrently. The two-lock concurrent queue is livelock free and outperforms a single lock when several processes are competing simultaneously for access - good for busy queues on machines with non-universal atomic primitives like test&set. Scherer, Lea, and Scott [29] proposed two new non-blocking and contention-free implementations of synchronous queues. Adam, Morrison, Afek [17] introduced LCRQ - a linearizable non-blocking FIFO queue for x86 processors. They used a combining-based algorithm with CAS. The idea behind combining is that the synchronization cost of a contended CAS hot spot is so large that performing work serially is more efficient. They proved that combining based queues scale better than CAS-based list queues. Ellen, Fatourou, Ruppert, and Breugel [26] proposed the first non-blocking, linearizable Binary Search Tree implementation using CAS operations. Natarajan, Savoie and Mittal [27] gave a wait-free implementation of a concurrent red-black tree using CAS instructions. Afek, Kaplan, Korenfeld, Morrison and Tarjan [23] presented a practical concurrent self-adjusting search tree called a CBTree (Counting Based Tree) that scales with the amount of concurrency. Crain, Gramoli and Raynal [25] proposed a

non-blocking implementation of a skip list. Shaifei [24] applied her approach for concurrent data structures to Patricia Tries.

Chapter 4

Our Implementation of Heap

A heap is a specialized tree-based data structure. Placement of nodes in various levels of the heap in increasing or decreasing order makes the heap a min-heap or max-heap. A heap must satisfy this property at all times. Heap is an inherently sequential data structure usually implemented in an array. After any modification to the heap (insert/delete), the heap property is restored using internal operations.

In a single-core system, only one operation is performed at a time and the heap is guaranteed to be in a stable state in-between operations. In multi-core systems, multiple operations can be active concurrently. Being a shared data structure in this case, heap operations must be ordered in some way to guarantee results identical to a sequential execution. One way of doing this is to provide an operation exclusive access to the heap using locking. The heap remains inaccessible to other active operations, and they must wait till the lock-owning operation finishes. Another way can be fine-grained locking, where an operation holds exclusive access only to certain nodes on the heap at one time. Both of these ideas fall under the category of 'blocking' techniques as threads waiting for access to locked nodes cannot make any progress. This makes the system prone to delays, deadlocks and resource under-utilization.

Our algorithm is an endeavor to parallelize operations on heaps in a non-blocking manner. It evades the cons of blocking techniques and also checks the risks associated with concurrent operations on shared data.

In this chapter, we describe our approach to implementing a non-blocking linearizable Heap. In our illustrations and examples, we will talk about a min-heap wherein the root node is the lowest valued node in the heap and values increase as we move down the heap levels. This chapter is divided into six sections.

Section 4.1 presents three crucial tools used in this implementation. The three tools are heap representation as an array of pointers, the concept of flagging and CAS usage.

In Section 4.2 we present an overview of our algorithm. We provide an overall idea of INSERT, DELETE and FIND-MIN operations. This section highlights how the key concepts of 'info structures' and 'flagging a node' work together to make this algorithm non-blocking.

In Section 4.3 we describe the data structures in detail. Our implementation consists of two main data structures – An array of pointers and a dictionary. The heap is represented as an array of pointers to nodes and info objects. A dictionary is used to record operation ids and the location where corresponding operation is currently active.

Section 4.4 consists of our algorithm.

In Section 4.5 we will talk about the heap operations. There are 3 main operations on heaps – insert, delete and find-min. There are 6 sub-operations – BubbleUp (BU), BubbleDown (BD), CopyToRoot(CTR), UPdateRoot(UPR), GetParent, and GetChild.

Section 4.6 is composed of special cases of various heap operations. We see the results of conflicting scenarios between concurrent BubbleUp (BU) and BubbleDown (BD) operations using examples.

Section 4.7 comprises the pseudocode.

4.1 Tools and Techniques

In a multi-core system, allowing concurrent processes to access a shared data structure in parallel is a very complex task. To establish the non-blocking property, we must ensure that concurrent threads make progress regardless of other threads. It is likely that different threads will execute at different speeds and more than one thread might access a node on the heap at the same time. Multiple threads trying to update the same node simultaneously would further lead to race conditions. To avoid data inconsistency and race conditions, it is important that heap nodes are always updated atomically.

We know that in a sequential execution, the heap property is restored after every insert/delete operation. Restoring the heap property requires traversing the heap and swapping nodes. Thus, one insert or delete operation in the Heap affects multiple nodes, thereby taking more than one atomic instruction to complete. Also, it is possible that a new operation may be started before the previous one completes, leaving the heap inconsistent between operations. We have attended to these complications by three means in our proposed algorithm – the choice of data structure to implement the heap, the concept of flagging and the single word Compare-and-Swap (CAS) operation.

4.1.1 Heap implementation as Array of Pointers

Generally, the heap is implemented in an array. For a parent node at location x, left and right child nodes are stored at locations 2x and 2x+1 in the heap. It does not require pointers between elements. This implementation is simple and easy; however in a shared memory concurrent system, it poses the risk of data inconsistency.

Suppose a heap node has three fields – value(v), priority(p) and operation id (o). There are three operations A, B, C trying to access/update the same node simultaneously. Operation A is updating the node while B and C are reading. A has successfully updated v to v', when B starts reading the node. B finishes reading and then A updated p to p'. At this time C reads the node. After C is done, A updated o to o'. At the end of operations A, B and C, they have the following values for the same node.

A: (v', p', o') B: (v',p,o) C: (v', p' o)

These values were indeed true at some instant in time; however we want an update operation on a node to appear as if it was atomic. i.e. v, p, o were changed to v', p', o' at the same instant in time. To achieve this result, we have represented the heap as an array of pointers (**Heap Array**). Starting from index 1, each index denotes a location on the heap and contains a field called **node object pointer**.

Definition 4.6. Node object represents a node on the heap. It contains the value, priority etc information that is intended to be stored on the heap.

Definition 4.7 Node object pointer is the second component of an array index. It points to the node object for the corresponding heap location.

Using this representation, the node at any location can be changed by changing the value of the node object pointer. Let's assume the same example as above (node with values v, p, o). Now, to update any of the values, thread A will create a new node with values v', p', o'. Then, it will change the node object pointer to the address of the new node. This can be done atomically in one step.

With multiple operations working concurrently on various nodes in the heap, we need some mechanism to locate the node where an operation is currently active. This can be done by traversing HeapArray. However, in a large heap, it would be a time consuming process. To serve this purpose efficiently, another data structure used by this algorithm is a dictionary (Opdict). It records the **operation id** and the location on heap where the operation is currently active.

Definition 4.8 Operation Id is a unique number assigned to every operation on the heap.

These data structures will be discussed in detail in Section 4.2. Now, we have the ability to replace multiple node values in one step in the heap. Next, we will discuss how to make the algorithm non-blocking and ensure progress even if some thread is delayed/crashed.

4.1.2 Concept of FLAGGING

To maintain the consistency of a concurrent shared data structure, we need a mechanism to ensure that there is some ordering in operations executing on the same location. If insert and delete operations are active on the same node of the heap, they must be prioritized to avoid race conditions. What if the higher priority operation crashed while working on the node? Will the lower priority operation waiting for the node starve forever? If we allow a lower priority operation to access the node after waiting for a fixed time, how will it identify the updates done by the

previously crashed thread and fix/overwrite them? Answers to all these questions lie in the concept of 'flagging'.

Earlier, we mentioned that the array contains a node object pointer. There is another pointer in each array component, known as, the **info object pointer**. The value of this field is null by default. 'Flagging' is the process of changing this value from null to a valid pointer to an **info object**.

Definition 4.9. Info object is an operation descriptor object. It contains all necessary details to complete an operation successfully.

Definition 4.10. Info object pointer is the first component of an array index. It points to the Info object for that location on the heap.

If the info object pointer for a location is null, it means that no operation is currently active on that node. Like locking, the main purpose of flagging a location is to give an operation exclusive permission to change the fields of that location. The advantage of flagging over locking is that failure of one process does not block other processes from making progress, since info objects store enough information for an operation to be completed. It also helps faster threads to avoid being blocked by slow threads working on shared locations on the heap.

Any update operation on HeapArray flags the location to be updated, makes the required changes and unflags it on completion. Other operations trying to access this location meanwhile, read the info object and help the active operation to complete. We learned about the crucial role that flagging plays in this algorithm. But in a multi-core system, how can we ensure that flagging is done atomically? The answer to this lies in the next section.

4.1.3 Compare-and-Swap operation

The Compare-and-Swap (CAS) instruction when called with three parameters CAS (x, old, new) returns false if the value of variable x is not equal to expected old value [34]. Otherwise, it changes the value of variable x from the expected value old to some value new and returns true. For more details on CAS, refer the Section 2.3.

In our algorithm, values of the info object pointer and node object pointer are always changed using CAS. We know that an update operation flags a location, makes required changes and then unflags. This can be done as follows:

i. `CAS(info_object_pointer, NULL, new_info_object)`

ii. `CAS(info_object_pointer, current_info_object, NULL)`

iii. `CAS(node_object_pointer, current_node, new_node)`

The correctness of algorithms using CAS often depends on the fact that, if the CAS succeeds, the old value has not been changed since the preceding read. Suppose the value of an info_object_pointer was A when the CAS operation began. Meanwhile, the value was changed by another thread to B and then back to A. In this scenario, CAS can succeed. However, it should not since the value was not A throughout the operation. This is called the ABA problem [34, 11].

Flagging checks if the value of the info object pointer is null. Suppose the value is changed to something else and then back to null (ABA problem with A=null and B=some value). If CAS returns true in this scenario, we are still ok. The purpose of flagging is to make sure that no one else has flagged the location (info_object_pointer is null). While unflagging, we update the value of info_object_pointer to null for the current operation. Once the value of an info_object_pointer has been set to non-null, it cannot be changed. If any other thread attempts

to change the value, equation (i) will fail. Hence, we are sure that there is no ABA problem in unflagging (eq ii). Nodes are referred to by their address. We are assuming that the algorithm will be implemented in a language that supports garbage collection or memory management is taken care of to avoid the ABA problem.

4.2 Overview of Algorithm

Before discussing the functionality of heap operations, let's summarize the working of update operations in general. When an update operation is called, it first determines which heap locations would be affected by the update. A location is said to be affected by an update if at least one of its fields would be changed during the operation. Before applying an update to HeapArray, the operation must flag the location that it can possibly alter. Each operation has a unique info object that is used to flag the required locations on the heap. After creating an info object, the operation checks if the location is already flagged by another concurrent update. If yes, it tries to help the concurrent update to complete and unflag the location. Unflagging is done by setting the info object pointer of the location to null. After helping, the operation retries flagging the location. Setting the info pointer field to the descriptor is done using CAS. Once a location has been flagged for an operation, we can be sure that as long as there is an active thread in the system trying to access this location, the operation in the info object would be completed.

Our heap supports three operations: INSERT, DELETE and FIND-MIN. INSERT operation is called with a value. It inserts that value at the next available location in heap. DELETE operation removes one node from heap and returns the value of the root node. FIND-MIN operation

returns the value of root node without actually removing the node from the heap. We will give a brief overview of these function here followed by a detailed explanation in Section 4.4.

INSERT(value) adds an entry into Opdict with operation Id and index = 0. It creates an info object I and flags the size of the heap. It records the current value of size in I and calculates the new value by adding 1 to current value. It creates a new size node with the new value and replaces the old size with the new node. Next, it creates a new node with the value passed as input to the function and places it in 'new size' location on heap. After successfully adding the new node to the heap, size is unflagged and the index in Opdict is set to 'new size'. The new node is then bubbled up the heap. The new node value is compared with the parent node value and swapped if the parent value is greater than its own value. This bubbling up continues until the parent node value is less than the current value.

DELETE starts with adding an entry into Opdict with operation Id and index = 0. It creates an info object I for delete and flags the size of the heap. It records the current value of size in I and calculates the new value by subtracting 1 from current value. It creates a new size node with the new value and replaces the old size with the new node. Next, it creates a new info object with operation = 'CTR' and flags the node at location 'old size' on the heap. On successful flagging, size is unflagged and the index value in Opdict is set to the new location. If this node is in status 'BU', it is set to done and a new info object for operation = 'UPR' is created to copy this node value to the root. Next, the operation tries to flag the root node with the info object for UPR. Once root has been flagged, the last node is unflagged. The current value of root is stored in the info object. A new node is created using information in the info object and placed at the root location. Root node is unflagged. Status of new root node is 'BD'. It is bubbled down the

heap by comparing values with its child nodes. If it is greater than any of the child nodes, it is swapped with the smaller node. Bubbling down continues until it finds its right location on the heap.

The FIND-MIN operation creates an info object for the find-min operation and flags the root node. Initially the value of root in the info object is null. If the status of root node is 'done', and the value of root is null in its info object, the current value is stored using the CAS operation. FIND-MIN returns the value stored in info object.

At any point in all the operations, if an operation is unable to flag a location, or it finds that a location is already flagged by another operation, it helps the other operation to complete successfully. By doing so, the heap is made tolerant to slow or crashed threads. Other operations are helped using values stored in the info objects. At each point during an execution, if the node corresponding to an operation id is moved up or down the heap, the new location is recorded in opdict. When a node is placed in it's correct position in the heap, the index in opdict is updated to -1.

Now that we are familiar with an overview of the algorithm, we will discuss data structures, functions and special scenarios in detail in the next few sections.

4.3 Data Structures

Our heap implementation comprises of two data structures: an array of pointers and a dictionary.

4.3.1 Array of pointers

Traditionally, the heap is implemented as an array without any pointers. To make changes to the heap atomic, we have represented the heap as an array of pointers.

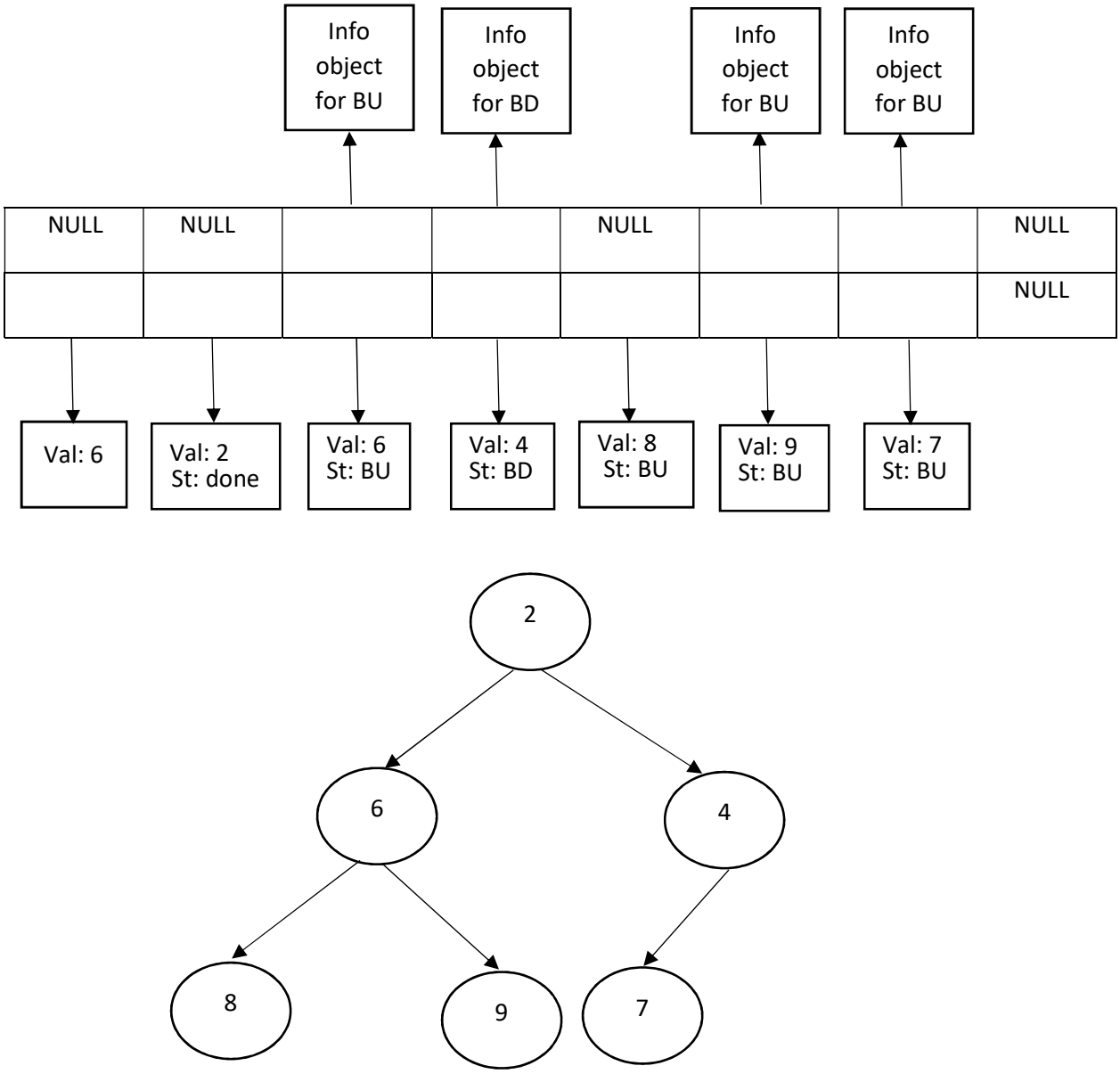

Figure 4.2 Heap representation as array of pointers

Each array index comprises two pointers: the info object pointer and node object pointer. Each index in the array starting from 1 represents a node in the heap. Array index 1 is the root node of the heap. Array index 0 has a special purpose. It is used to record the size of the heap. Figure 4.1 shows an array and corresponding heap.

4.3.1.1 Info object

The Info object pointer points to the info object for a location. An Info object is an operation descriptor object. It contains all necessary details to complete an operation successfully. Each operation creates a unique info object for each step during execution. A general descriptor class contains 5 values: the first three are integer placeholders for values and indices, an operation field to store the type of operation and another integer field to store the operation id.

```
type HeapArray
     Nodeptr   node
     Infoptr    info

type Descriptor
     Integer          value/index
     Integer          value/index
     Integer          value/index
     Operation        {INS/DEL/CTR/UPR/BU/BD}
     Operation_id     OpId
```

The general Descriptor class is customized in four ways for various operations.

```
type sizeDesc
     Value            Val in Insert, NULL in Delete
     Size             Size of heap
     Operation        INS/DEL/CTR/MIN
     OpId             operation id

type delDesc
     Value            Value of last node
     Old_root_value
     Operation        UPR
     OpId             Operation Id
```

```
type BUDesc
      Pindex          Parent node index
      Cindex          Child node index
      Operation       BU
      OpId            Operation Id

type BDDesc
      Pindex          Parent node index
      Lcindex         Left Child node index
      Rcindex         Right Child node index
      Operation       BD
      OpId            Operation Id
```

4.3.1.2 Node object

The Node object pointer is the second component of an array index. It points to the node object for the corresponding heap location. Each node object represents a node on the heap. It contains value, status, and operation id. Any information that is intended to be stored on the heap can be added to the node object. The Node at 0^{th} index in the array serves a special purpose. It is used to store the size of the heap.

```
type Node
      Value   value
      Status {BU, BD, done}
      Operation_Id OpId

HeapArray[0] contains heap size

      Value : Size of heap
      status: done
      OpId  : Null
```

Another data structure used in our implementation is a dictionary.

4.3.2 Dictionary

A dictionary or map is an associative container that stores elements formed by a combination of a *key value* and a *mapped value*, in a specific order. Any mapped value in a dictionary can be searched by passing a key value. In a heap, where one operation might work on multiple nodes,

finding out the exact node where an operation is currently active can be done easily using a dictionary.

In our implementation, the key to the dictionary is operation id. An operation id is a unique number assigned to each operation. The mapped value for each key is the index/location on the heap where the operation corresponding to the 'key' operation id is currently functional.

Operation id	Index
1234	3
1248	-1
2367	2
2563	0

Figure 4.2 Operations Dictionary

An INSERT or DELETE operation starts with adding the operation id and index as 0 in the dictionary (Example: Operation Id 2563 in Figure 4.2). Every time a node is moved up or down the heap, the new location is updated in the dictionary. When a node finds it's right place in the heap, its index is changed to -1 to indicate that the operation has completed (Example: Operation Id 1248 in Figure 4.2). Note that space for the completed operations can be reclaimed from the dictionary by removing Operation ids with index value equal to -1.

Having learned the key features of our approach, a brief overview of the algorithm and data structures, we are ready to discuss the algorithm and update functions in detail in the upcoming sections.

4.4 Algorithm

```
1   UPDATEINDEX(opid: Operation Id, index): {Boolean}
2         Update index of given opid in Opdict

3   GETPARENT(idx: Node Index): {Index, invalidposition}
4         parent <- floor(idx/2)
5         if parent < 1 then
6               return invalidposition
7         return parent

8   GETCHILD(idx: Node Index, side: Left/Right): {Index, invalidposition}
9         if side is left then
10              child <- 2*idx
11        else
12              child <- 2*idx + 1
13        if child > current_size_of_heap then
14              return invalidposition
15        return child

16  INSERT(val: Value, opid: Operation Id):{Boolean}
17        add opid to Opdict
18        create new Info object I with Op = INS
19        keep trying CAS until Size is flagged with I
20        HELP(Size)
21        while (new node is not bubbled up to right position in heap)
22              HELP node
23        return true

24  DELETE(opid: Operation Id):{value, emptyheap}
25        add opid to Opdict
26        create new Info object I with Op = DEL
27        keep trying CAS until Size is flagged with I
28        retvalue = HELP(Size)
29        while (node is not bubbled down to right position in heap)
30              HELP node
31        return retvalue

32  FINDMIN(Opid : Operation Id) : {Value}
33        If root node exists
34              create new info object I to flag root for operation 'MIN'
35        else return 'heap empty'
36        while (status of root node != 'done')
37              HELP(root)
38        CAS to flag root with I
39        HELP(root)
40        if root value in I is not null then
41              return root value in I
42        return false


43  HELP(cnode: node): {Boolean, value : incase of delete}
44
45        if node does not need help
46              return false
```

```
47
48          copid = opid value of node being helped
49          cindex = index of node being helped
50
51          Evaluate(true)
52          Case 1:  node is flagged for Op = INS
53
54                CAS to update size value in Info
55                newsize = size value in Info object of size + 1
56                update size pointer to new node(newsize, done, opid)
57                a) if newsize = 1 then                    //first node in
58    heap
59                      add new node at root location <- (val,done,opid)
60                b) if newsize > 1
61                      i) node at location newsize is unflagged
62                            Create new node <- new node(val, BU, opid)
63                            CAS to add new node to location newsize
64                            Update index of opid in Opdict to newsize
65                            Unflag size node
66
67                      ii) node is flagged
68                            HELP(node at location newsize)
69
70          Case 2: status of node = BU
71
72                a) if node is unflagged then
73                      Create new Info object I <- BUDesc(NULL, cindex, BU,
74    opid)
75                      CAS to flag node with object I
76                      if unable to flag then HELP(cnode)
77
78                b) if node is flagged then
79                      pnode = GETPARENT(cindex)
80                      i)  status of parent = 'BD' and parent has been
81    flagged
82                            unflag itself and HELP(pnode)
83                            return
84                      ii) pnode is flagged or status of parent is != done
85                            HELP(pnode) until unflagged and done
86                            return
87                      iii) pnode is unflagged and status = 'done'
88                            CAS to flag parent for 'BU' operation
89                            HELP(pnode)
90                            return
91
92          Case 3: status of node = 'done' and node is flagged for 'BU'
93
94                a) if Info object does not have parent index
95                      CAS to update parent index in Info object
96                b) Parent index is present in Info
97                      i) if childval >= parval
98                            update child node status <- 'done'
99                            update child node's opid index in Opdict <- -1
100
101                     ii) if childval < parval
102                           create new nodes P and C nodes with swapped
103   values
```

```
                              a) parent was root node (index 1)
                                    status of P = 'done'
                              b) parent not root node
                                    status of P = 'BU'
                              c) parent status was 'BD' then
                                    status of C = 'BD'
                        Update parent node P and child node C
                        Update the opids of both nodes in Opdict

                  Unflag parent and child nodes
                  return true


      Case 4: node is flagged for op 'DEL'  // compute size and last
node

                  if old size is not updated in Info object
                        CAS to update old size in info object
                  if oldsize = 0 then     return emptyheap
                  newsize = oldsize - 1
                  update Size to new node (newsize, done, NULL)
                  a) if last node in Heap Array is unflagged then
                        if last node is root
                              CAS to flag root for op 'UPR' with value = 0
                        else
                              CAS to flag last node for op 'CTR'
                  b) if flagged for current operation then
                        Update the opid of node in Opdict to new location
                        Unflag Size
                        HELP(last node)
                  return


      Case 5: node is flagged for op 'CTR'  //copytoroot

                  if status of node!= done then
                        update status to 'done' and update index = -1 in
opdict
                  if root node is already flagged
                        HELP(root) until unflagged
                        return
                  CAS to flag root with new Info update (node.val, NULL, UPR,
opid)
                  unflag last node
                  update opid index in Opdict to 1
                  HELP(root) and return old root value
                  if HELP returns numeric value then return the value


      Case 6: node is flagged for op 'UPR'    //update root HeapArr[1]

                  if old value in Info object has not been updated then
                        CAS to update old value of root in Info Object
                  if info.node.val = 0 then
                        update nodeptr to NULL
                  else
```

```
                  create new node R(cnode.info.value, BD,
cnode.info.opid)
                  update root to R
            unflag root
            return old value of root


      Case 7: status of node = 'BD' and left node index in Info is
missing

            if node is unflagged
                  Create new Info object I to flag node for 'BD'
                  CAS to flag node for I
                  if unable to flag then HELP(node)
                  return

            while (left child index in Info object is missing)
                  lcnode = GETCHILD(cindex, left)
                  a)  if lcnode > heap size
                        CAS to Update left child index to 0 in I
                        CAS to Update right child index to 0 in I
                        return
                  b)  if left child is flagged for another operation
                        HELP(left child)
                        return
                  c)  if left child is unflagged
                              CAS to flag child with I
                        CAS to update left child index in I
                        HELP(left child)
                        return

      Case 8: node flagged for 'BD' and right node index in Info is
missing

            while (right child index in Info object is missing)
                  rcnode = GETCHILD(cindex, right)
                  a)  rcnode > current heap size
                        CAS to update left child index to 0 in I
                        return
                  b)  if right child is flagged for another operation
                        HELP(right child)
                        return
                  c)  if right child is unflagged
                              CAS to flag child with I
                        CAS to update right child index in I
                        HELP(right child)
                        return

      Case 9: node is flagged for 'BD'

            a)  parent value is < left child and right child value
                  update all three nodes status to done
                  update all node's opid index in Opdict to -1
            b)  left child has the smallest value
                  create new node P with left child node attributes
                  create new node C with parent node attributes
                  if child status is 'BU', P node status will be 'BU'
```

```
                    update parent index pointer to new node P
                    update child index pointer to new node P
                    update opids for both nodes in Opdict to new location
              c)  right child has the smallest value
                    same as above
              Unflag right child, left child and parent node
              return true


       Case 10: status of node is 'done' and node is flagged for op 'MIN'
                    CAS to update value of root in Info Object
                    unflag root

     End-evaluate
```

4.5 Functions

Our algorithm consists of three main functions – INSERT, DELETE, FIND-MIN and six sub-operations – BU, BD, CTR, UPR, GETPARENT and GETCHILD. As evident from Section 4.3, a major part of each operation is performed in the HELP function. This enables a helping operation to execute some part of the code for the operation it is trying to help. In this section, we will look at all the functions in detail. Various cases of the HELP function will be explained as part of the execution flow of the main functions.

4.5.1 INSERT

The insert function takes the value to be inserted into the heap and the operation id of the current operation as input parameters. It will add an entry for the operation id in OpDict and create a new Info object (line 18). It will keep trying CAS on heap size (HeapArray[0].infoptr) to flag it. After successfully flagging size, it will call the HELP function in line 20. At this point, size has been flagged for op=INS, hence Case 1 of HELP function will be executed. It will perform CAS to update the old value of size in I. Since we are using CAS, it will ensure that the value is updated atomically. The new size will be calculated by adding one to the existing heap size. The new node for size will be created using information in I and CASed to HeapArray[0].nodeptr (line 57). The new value of size is the heap location where insertion has to be done. If new size = 1, it means that the new node to be added will be the root node. For root nodes, status of node will be 'done'. For non-root nodes, new node will be created with status 'BU'. If the heap location is flagged for another operation, HELP function will be called to complete the ongoing operation (line 68) and retry flagging. If the location is unflagged, the new node will be added to

HeapArray[new size(nodeptr)] using CAS. The index for operation id in OpDict will be updated to the new location and the size node will be unflagged.

At this point, we have successfully incremented heap size and added the new node to heap. Next, we have to place this node at its right place in the heap, where it is less than the parent node and greater than both the child nodes (since it is a min-heap). This is accomplished by calling a series of Bubble Up (BU) operations (line 21).

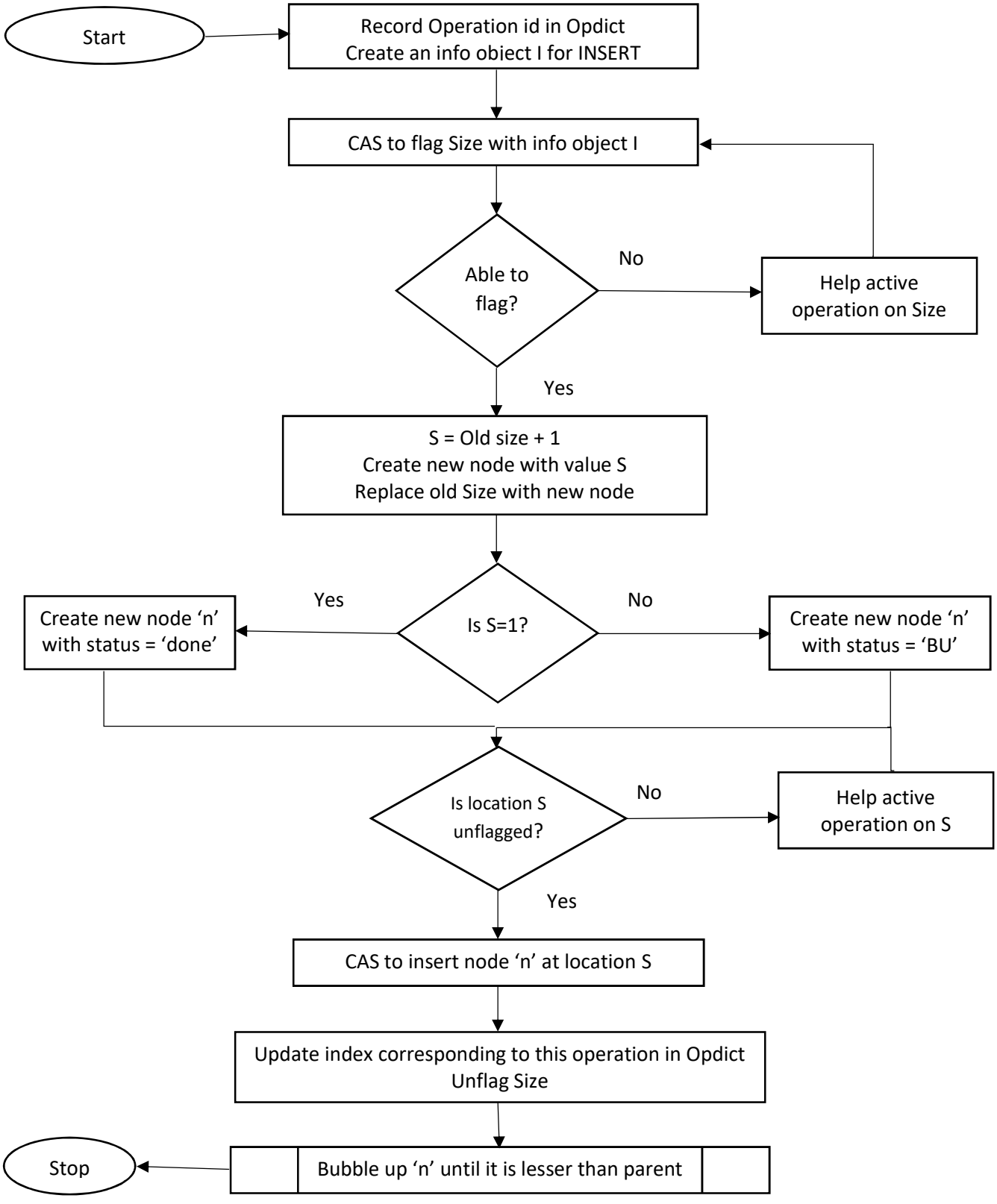

Figure 4.3 INSERT flow chart

4.5.1.1 Bubble Up (BU)

The Bubble up operation moves a node upwards in the heap until it satisfies the heap property. A node that requires bubbling up is identified with status = 'BU'. A newly added node always has status 'BU' unless it is the root node. Once the node is placed in the correct position, status is changed to 'done'. Corresponding index in OpDict is also updated as the node is moved to higher levels. Code for BU comprises two cases in the HELP function.

Case 2 (line 70): node is in status BU. If the node is unflagged, create info object I for BU operation and try to flag the node. If flagging CAS is unsuccessful, it means some helping concurrently active helping thread has flagged it for either BU or some other operation. Call HELP function to work on the info object. If the node is already flagged or CAS is successful, execution continues from line 77. The index of parent node is obtained by calling GETPARENT function. If status of parent node is BD it means that it is not yet in its right position and is moving down the heap (line 79). If the parent node is flagged for BD, BU will yield by unflagging itself and calling HELP for the parent node. If parent node in status BD is unflagged, it will be flagged for current BU. If parent is already flagged for some other operation, HELP the parent node to bring it to 'done' status. Once parent node is unflagged and in 'done' status it means that it is placed correctly in the heap and is available for an operation (line 85). Perform CAS to flag parent node with I (Info object for BU) and call the HELP function.

Case 3 (line 90): It is the next step after Case 2. After parent node has been successfully flagged for BU, the code in Case 3 is executed. Update the parent index field in I. Obtain the child and parent node values using indices stored in I. We are sure that the values would not be changed by concurrent threads as they are updated in I after flagging the nodes. If the parent

value is less than the child value, mark status of both nodes as ‘done’ since they are satisfying the heap property. If child is greater than parent, the nodes need to be swapped. We will make copies of parent and child nodes and update node pointers. If old parent was a root, the new parent status will be ‘done’ else new parent status will be ‘BU’. If old parent was ‘BD’, the new child status will be ‘BD’ else it will be ‘done’. Update indices in OpDict for both the nodes.

Satisfying of the heap property by the new node marks completion of INSERT operation. The point in the execution when the status of the new node is updated to ‘done’ marks the linearization point of INSERT. Adding a new node to the heap only makes it a part of an intermediate heap. A node becomes part of the actual heap only after reaching the linearization point.

4.5.2 DELETE

The Delete function removes the root node from the heap and returns its value. The root node is replaced by the last node in the heap. The location of the last node is obtained from the heap size. The DELETE operation starts with adding an entry for the operation id in OpDict and creating the new Info object (line 26). The process of updating heap size is the same as in INSERT. It will keep trying CAS on the heap size (HeapArray[0].infoptr) to flag it. After successfully flagging size, it will call HELP function in line 28. At this point, size has been flagged for op=DEL, hence Case 4 of the HELP function will be executed. It will perform CAS to update the old value of size in I. The new size will be calculated by subtracting one from the existing heap size. A new node for size will be created and CASed to HeapArray[0].nodeptr (line 120). The last node of heap, identified by the new value of size, is to be placed at the root location. If the last node is flagged for another operation, it will call HELP to have the node unflagged. If the last node is also the root node (line

122), it will be flagged with the info object for op ‘UPR’ with new root value = 0. If it is an unflagged non-root node, it will be flagged for op = ‘CTR’. After successfully flagging the last node, the index for operation id in OpDict will be updated to the new location and size node will be unflagged.

At this point, we have successfully decremented the heap size and flagged the last node of the heap. Next, we have to copy it to the root node.

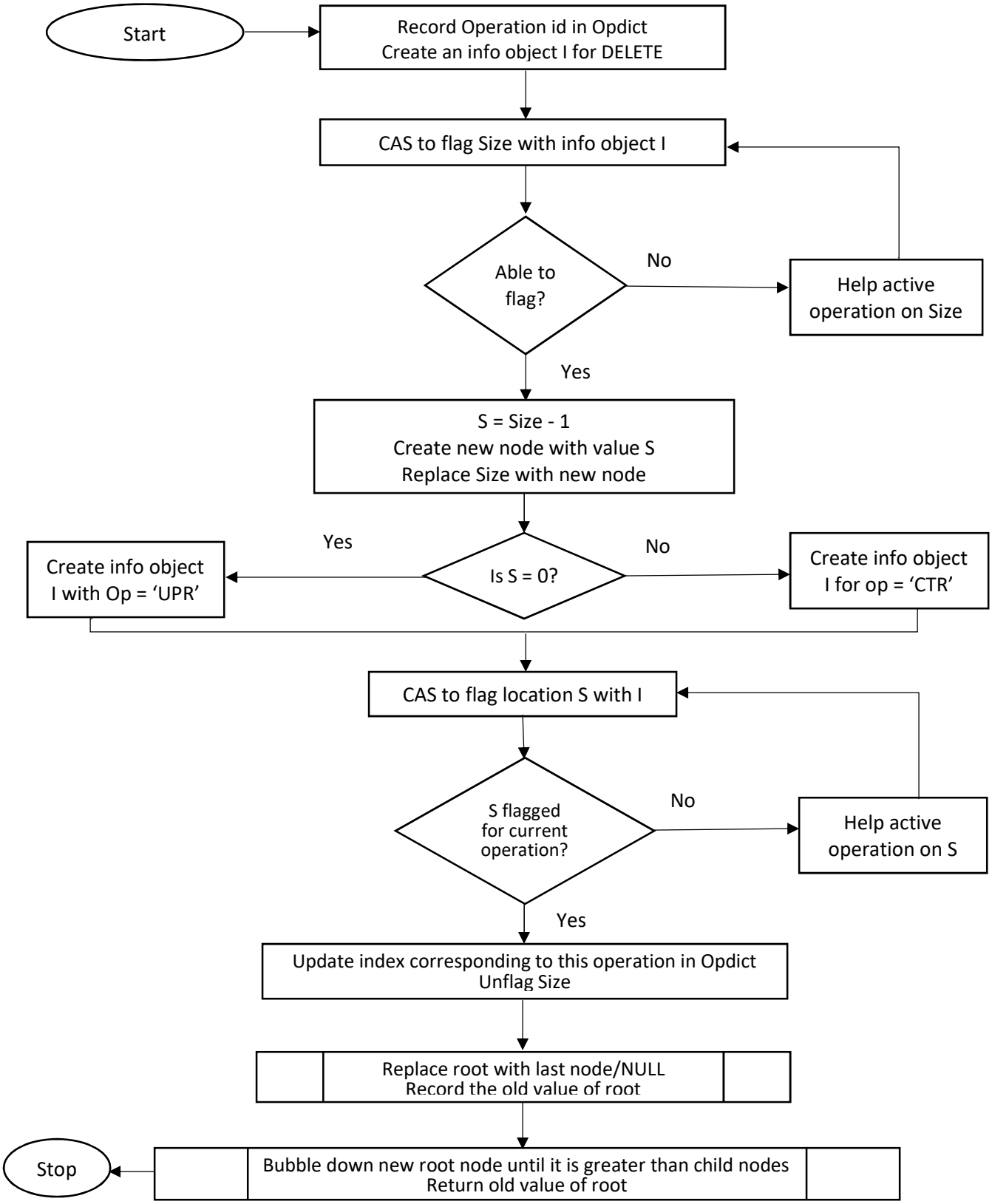

Figure 4.4 DELETE flow chart

4.5.2.1 Copy To Root (CTR)

This is a sub-operation of DELETE to copy the last node of the heap to HeapArray[0] location. Code for CTR is Case 5 of HELP function. If status of the last node is not 'done', it means that it is not yet a part of the actual heap. CTR will update the status to 'done' and the index in OpDict to -1 to indicate that the node is placed in its correct location in the heap (line 136). Note that this is not necessarily true. The last node might be in BU status; however, it will still be marked as done to make it a part of the actual heap. It will eventually satisfy the heap property when it is bubbled down the heap. Check if the current root node is already flagged. If yes, call HELP for unflagging the root node by completing the active operation. Create a new info object for operation 'UPR' with details of the last node. The root value field in the info object will be NULL this time. CAS to update HeapArray[1].infoptr with this info object. After successfully flagging the root node, unflag the last node. Call HELP to execute the next step UPR.

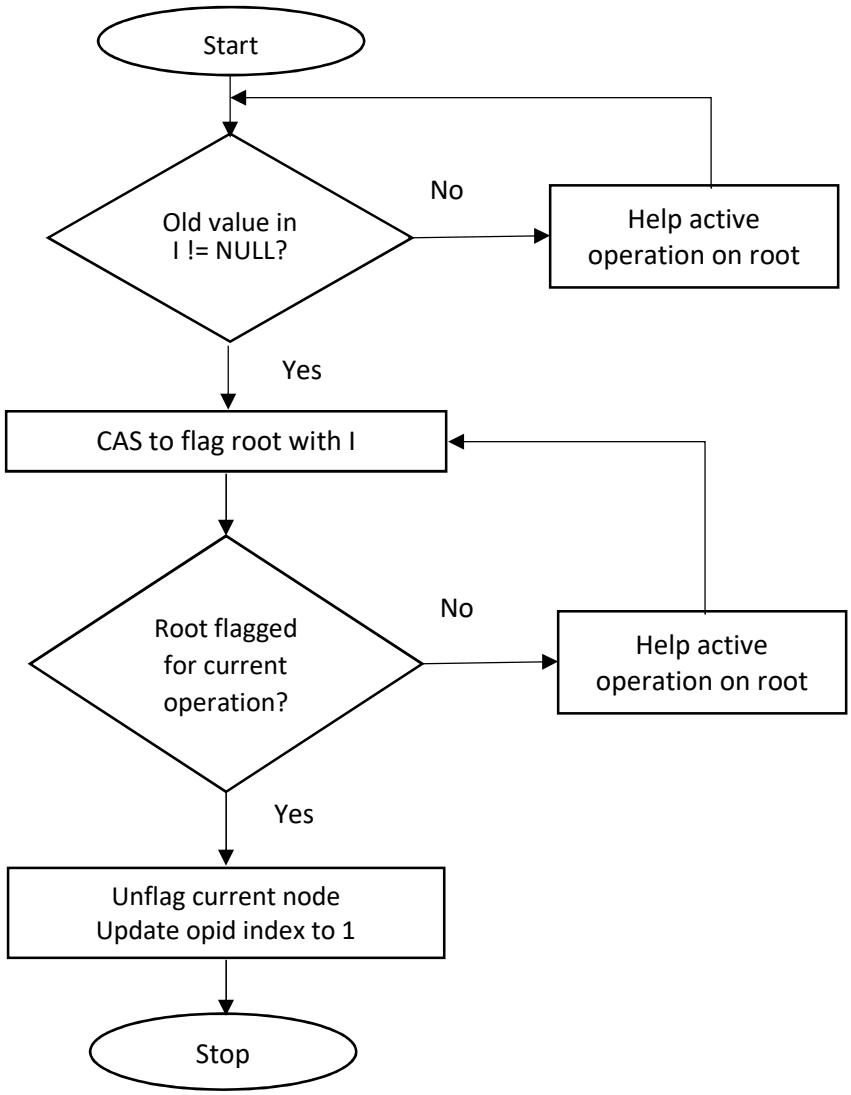

Figure 4.5 CTR Flow chart

4.5.2.2 UPdate Root (UPR)

The update root function replaces the root with a new node created using information in the info object. It is Case 6 of HELP function. While creating the UPR info object we left root size field as NULL. Now, after flagging the root node we are sure that the value of root cannot be changed by any concurrent threads, hence it will be recorded in the info object (line 150). If UPR is being

executed for the root node (root is the only node in heap and is being deleted), update HeapArray[1].nodeptr to NULL (line 152). If last node is different than the root node, create a new node using values from the info object and status = 'BD'. Use CAS to place this new node at HeapArray[1].nodeptr (line 155). Unflag root node and return the value of the old root stored in the info object.

So far we have successfully removed the root node from the heap and replaced it with another heap node. This new root must be moved downwards in the heap until it satisfies the heap property. This is done by the Bubble Down operation.

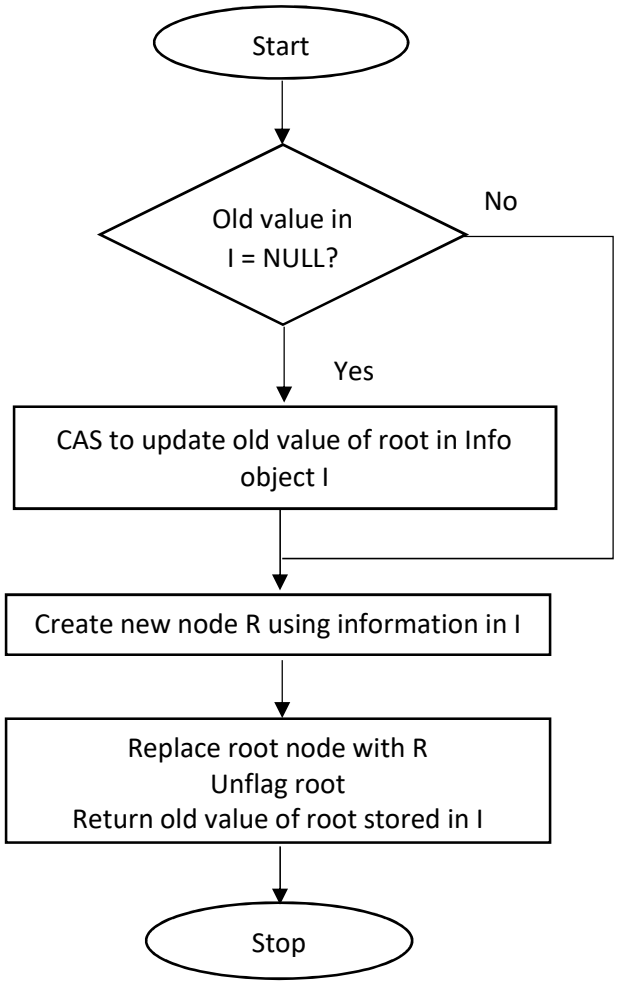

Figure 4.6 UPR Flow chart

4.5.2.3 Bubble Down (BD)

A node that requires bubbling down is identified with status = 'BD'. A new root node placed as part of DELETE operation always has status 'BD'. Once the node is placed in the correct position, status is changed to 'done'. The corresponding index in OpDict is also updated as the node is moved to lower levels. The code for BD comprises three cases in HELP function.

Case 7 (line 160): node is in status BD and infoptr.lindex is NULL. This scenario will occur when either the node has not been flagged for BD or if the node is flagged but left child is being worked on. If current node (parent node) is flagged for some other operation, call HELP. Create info object I for 'BD'. Flag the current node using CAS (line 164). If left child is not yet flagged, obtain left child index by calling GETCHILD with parameters current index and left (line 169). If left child index is greater than size of heap (left child does not exist), update left child index in info object to 0. A binary heap is filled in left to right order, so if left child does not exist, we can assume that right child is also absent, hence we will update right child index as well to 0 in info object. If left child exists and is flagged for another operation, call HELP. If left child is available for an operation (it is unflagged), flag it with info object for BD.

Case 8 (line 183): node is in status BD and infoptr.rindex is NULL. This scenario occurs when parent node and left child node have been flagged for BD. Obtain the index for right child. If right child does not exist, update right child index to 0 in info object (line 188). If right child is flagged for some other operation, call HELP. If it is unflagged, flag for BD info object (line 193). Update the right child index in info object.

Case 9 (line 199): Node is flagged for 'BD'. This code will be executed when all three nodes (parent node and two child nodes) have been flagged for BD operation. If parent value is less

than both child values, update all three node status to ‘done’. Else find the smallest child and compare with parent. Create copies of parent and smallest child nodes. If child status is ‘BU’, new parent status will be ‘BU’. Update the nodeptr of parent and child nodes to point to new nodes. Update indices in Opdict. Unflag all three nodes.

Satisfying the heap property by the new node marks the completion of DELETE operation. The point in execution when the root node is replaced by the last node is the linearization point of DELETE. The value of root node obtained by the operation just before this CAS is the return value of the function.

4.5.3 FIND-MIN

FINDMIN operation returns the highest priority/ minimum value (in case of min-heap) in the heap. This is usually the value of the root node. It does not remove any node from the heap. A FINDMIN operation executed on an empty heap returns ‘empty heap’ message (line 36). If root node exists, but status of root is not ‘done’, FINDMIN calls HELP function to bring the root to ‘done’ status. It creates an Info object I for operation = ‘MIN’ (line 35). It will repeatedly try to flag the root with I. On successful flagging, call HELP(root).

At this point, the status of root is ‘done’ and it has been flagged for ‘MIN’, hence Case 10 of HELP() is executed (line 217). The value of the current root node is stored in info object I using CAS and the root is unflagged. FINDMIN returns the value stored in the info object.

The advantage of flagging for FINDMIN is that even if the owner thread crashes or is delayed, some operation trying to access the root will perform Case 10 of HELP() and free the root node for the next operation. It will also ensure that FINDMIN has a valid root value to return. The linearization point of this operation is the CAS to the store root value in info object (line 218).

4.5.4 GETPARENT

This function returns the index of the parent node. It takes in an integer (index of current node) as parameter. Left child and right child of a node are stored in 2*index and 2*index + 1 location on the heap. This function obtains the parent index by dividing input integer by 2 (line 4). If the index obtained is less than one, it returns the message 'Invalid position/ Parent does not exist' (line 6) else it returns the parent index (line 7).

4.5.5 GETCHILD

This function returns the index of left/right child node. It takes in an integer (index of current node) and left or right as parameters. As mentioned earlier, left child and right child of a node are stored in 2*index and 2*index + 1 location on heaps. For the left child index, it multiplies the input integer by 2 (line 10). For the right child index, it multiplies by 2 and adds 1 (line 12). If the index obtained is greater than heap size, it returns message 'Invalid position/ child does not exist' (line 14) else it returns the child index (line 15).

4.6 Different Cases of BU/BD using examples

Bubble-Up(BU) and Bubble-down(BD) are sub-operations of Insert and Delete. These operations help in traversing the heap and placing a node in its right place. BD works in a top to bottom direction whereas BU works in a bottom-up way. Also, concurrent bubble-up operations might try to access the same nodes on the heap. That makes these operations vulnerable to conflicts. We will look at some scenarios of BD and BU including conflicting cases.

i. Parent status is BD and both child nodes are in status ‘done’. Parent node is lesser than both child nodes.

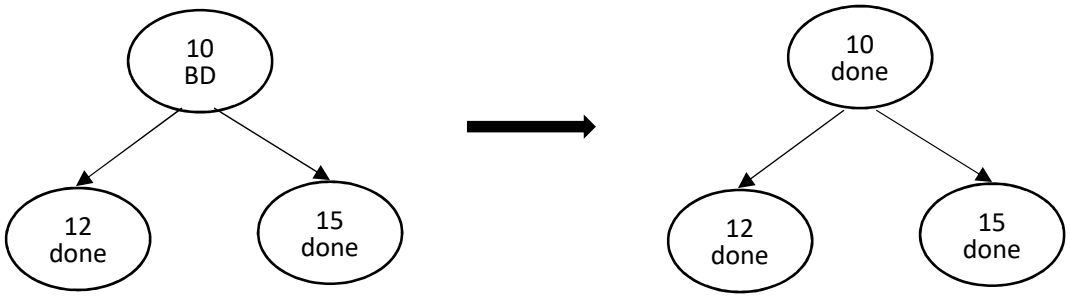

Figure 4.7 BD scenario (i)

ii. Parent status is BD and both child nodes are in status ‘done’. Parent node is greater than one child node.

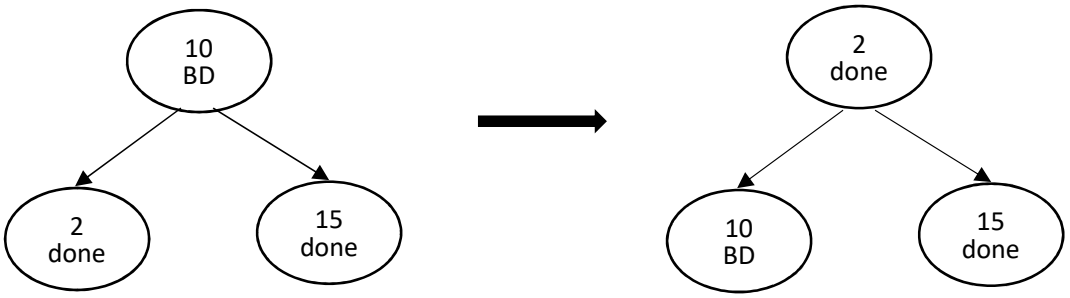

Figure 4.8 BD scenario (ii)

iii. Parent status is BD and both child nodes are in status ‘done’. Parent node is greater than both the child nodes.

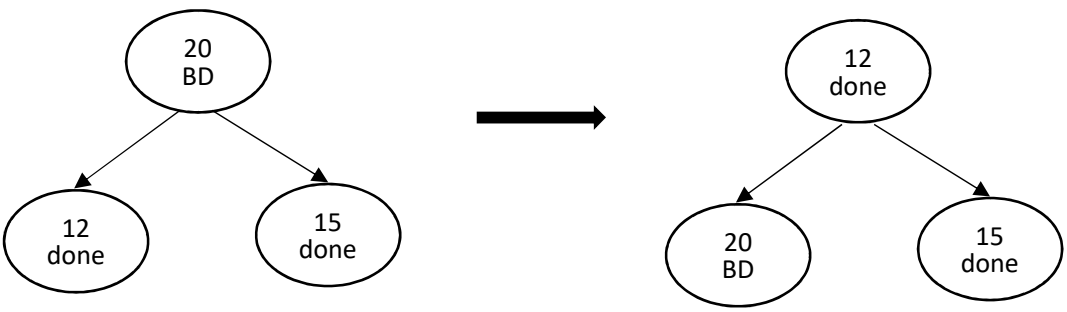

Figure 4.9 BD scenario (iii)

iv. Parent status is BD and one child node is in status BU while the other is 'done'. Parent node is lesser than both child nodes. In this case, both BD and BU operations will end up with the same configuration of nodes.

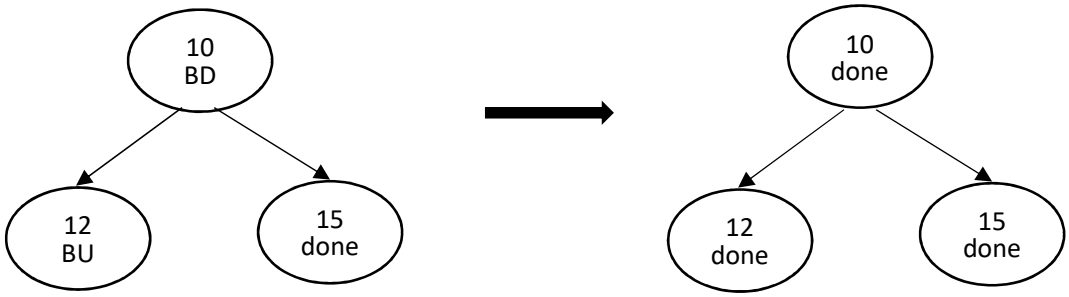

Figure 4.10 BD scenario (iv)

v. Parent status is BD and one child node is in status BU while the other is 'done'. Parent node is greater than the child node in status 'BU' but lesser than the one in status 'done'. For BD and BU, nodes status will be different, depending on which operation flagged the parent node first. If BD takes place first:

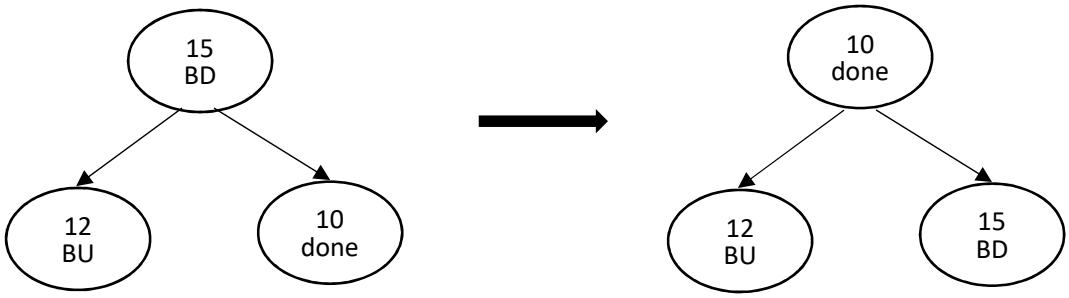

Figure 4.11 BD scenario (v)

If BU takes place first:

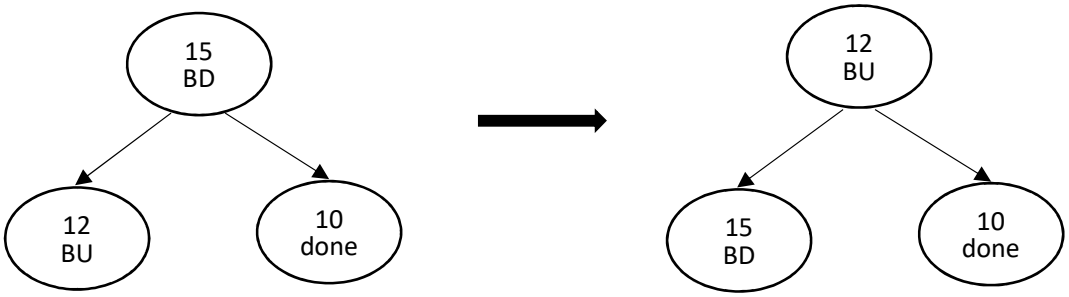

Figure 4.12 BD scenario (vi)

vi. Parent status is BD and one child node is in status BU while the other is 'done'. Parent node is greater than the child node in status 'BU' but lesser than the one in status 'done'. For BD and BU, nodes status will be the same.

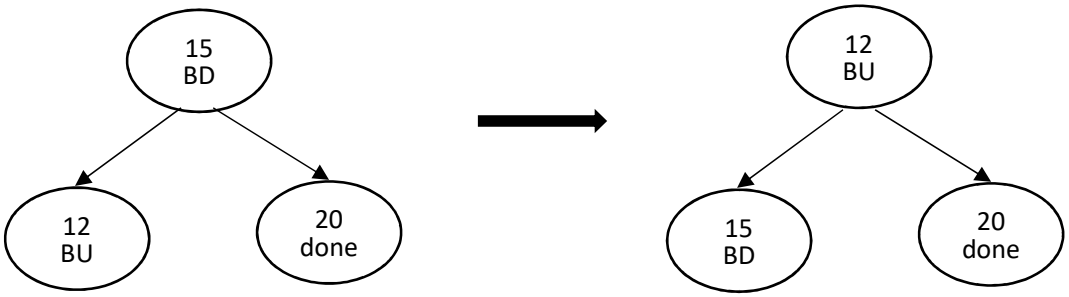

Figure 4.13 BD scenario (vii)

vii. Parent status is BD and both child status is BU. Parent is lesser than both child nodes.

If BD executes first:

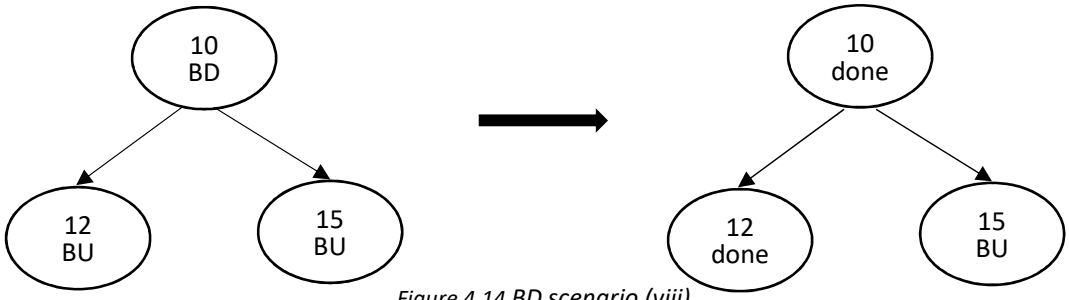

Figure 4.14 BD scenario (viii)

If left BU executes first:

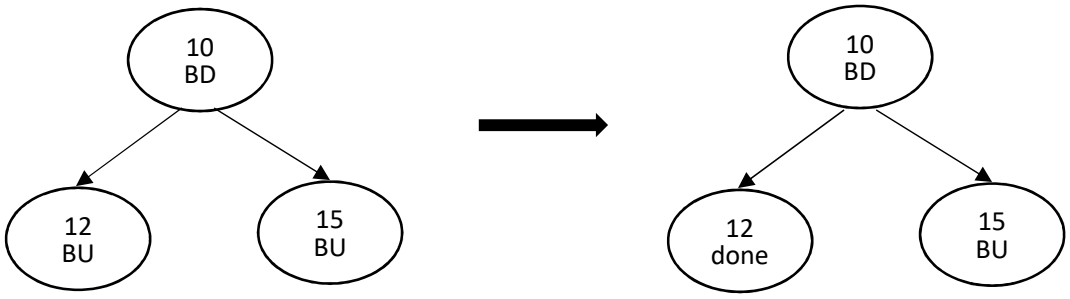

Figure 4.15 BD scenario (ix)

If right child BU executes first:

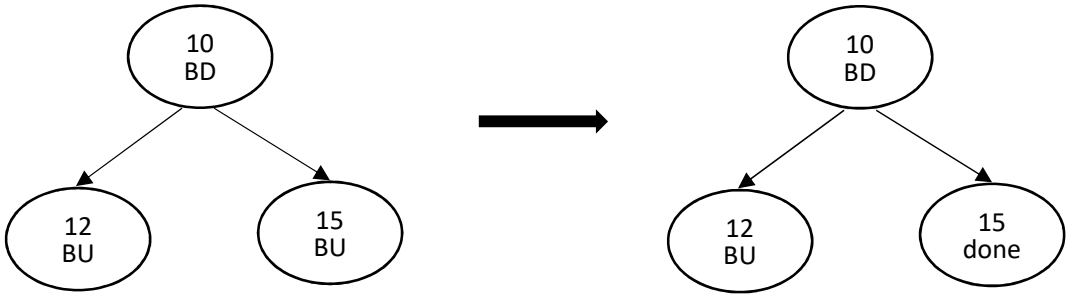

Figure 4.16 BD scenario (x)

viii. Parent status is BD and both child status is BU. Parent is lesser than left child node.

If BD or left child BU executes first:

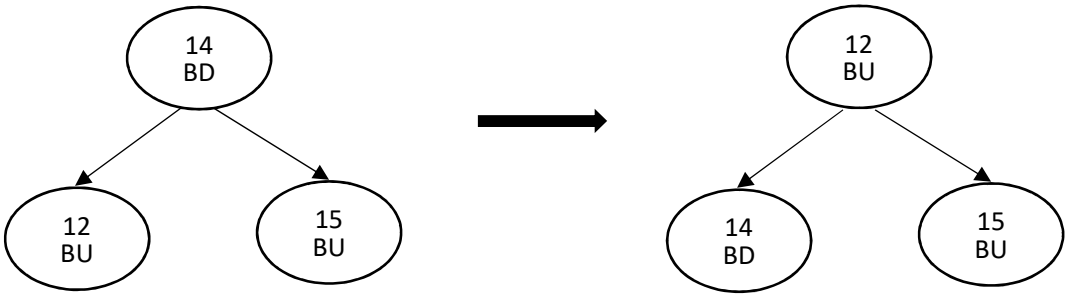

Figure 4.17 BD scenario (xi)

If right child BU executes first:

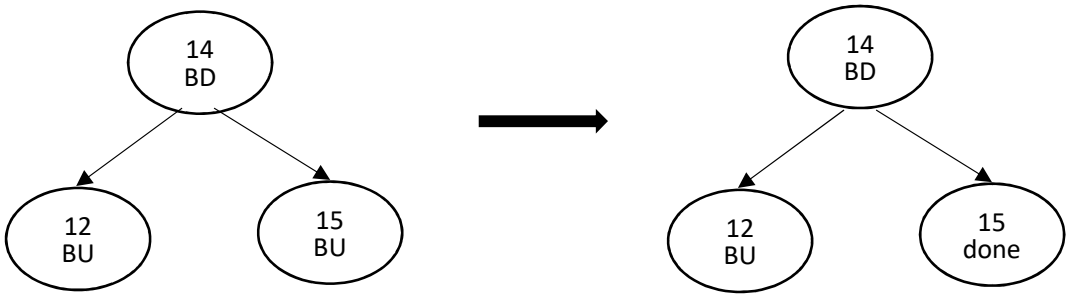

Figure 4.18 BD scenario (xii)

ix. Parent node is in status 'done' and child nodes are in status BU. Parent node is greater than child nodes. If left BU executes first:

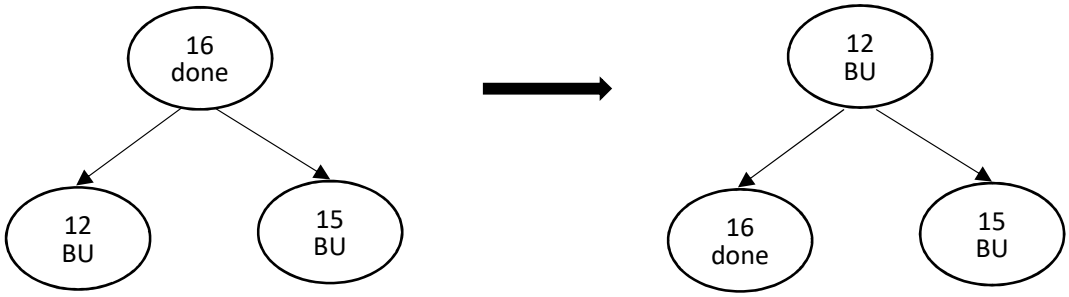

Figure 4.19 BD scenario (xiii)

If right BU executes first:

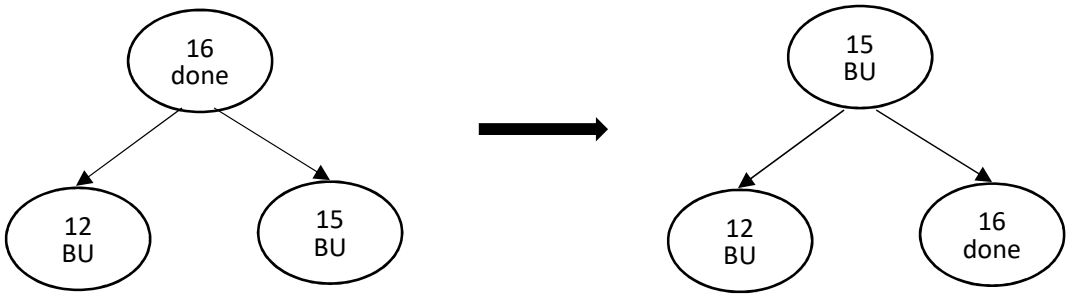

Figure 4.20 BD scenario (xiv)

4.7 Pseudocode

```
1   UPDATEINDEX(opid: Operation Id, index): {Boolean}
2         for opid in Opdict
3               Index = index

4   GETPARENT(idx: Node Index): {Index, invalidpos}
5         parent <- floor(idx/2)
6         if parent < 1 then
7               return invalidpos
8         return parent

9   GETCHILD(idx: Node Index, side: Left/Right): {Index, invalidpos}
10        if side = left then
11              child <- 2*idx
12        else
13              child <- 2*idx + 1
14        if  child > HeapArr[0].node.val then
15              return invalidpos
16        return child

17  INSERT(val: Value, opid: Operation Id):{true}
18        if opid not in Opdict then
19              Opdict.add(opid, I, val, 0)
20        while (Opdict.opid.index = 0)
21              if HeapArr[0].infoptr = NULL then
22                    I <- new SizeDesc(val, NULL, INS, opid)
23                    CAS(HeapArr[0].info, NULL, I)
24                    HELP(HeapArr[0])
25              else
26                    HELP(HeapArr[0])
27
28        while (Opdict.opid.index != -1)
29              HELP(HeapArr[Opdict.opid.index])
30
31        return true

32  DELETE(opid: Operation Id):{value, emptyheap}
33        if opid not in Opdict then
34              Opdict.add(opid, D, 0, 0)
35        while (Opdict.opid.index = 0)
36              if HeapArr[0].infoptr = NULL then
37                    I <- new sizeDesc(NULL, NULL, DEL, opid)
38                    CAS(HeapArr[0].info, NULL, I)
39                    retvalue = HELP(HeapArr[0])
40                    if retvalue is numeric
41                          value = retvalue
42              else
43                    HELP(HeapArr[0])
44
45        while (Opdict.opid.index != -1)
46              HELP(HeapArr[Opdict.opid.index])
47
48        return value
```

```
49  FINDMIN(Opid : Operation Id) : {Value}
50
51        I <- new SizeDesc(NULL, NULL, MIN, opid)
52        while (HeapArr[1].status != done)
53              HELP(HeapArr[1])
54        if CAS(HeapArr[1].info, NULL, I)
55              HELP(HeapArr[1])
56              if HeapArr[1].info.val != NULL
57                    return HeapArr[1].info.val
58        return false

59  HELP(cnode: node): {Boolean, value : incase of delete}
60
61        copid = cnode.node.opid
62        if Opdict.copid.index != cnode.index(HeapArr)
63              return false
64
65        Evaluate(true)
66        Case 1: cnode.info.op = INS
67
68              if HeapArr[0].info.size = NULL then
69                   CAS(HeapArr[0].info.size, NULL, HeapArr[0].node.value)
70              newsize = HeapArr[0].info.size + 1
71              HeapArr[0].node <- new node(newsize, done, opid)
72              if newsize = 1 then           //first node in heap
73                    HeapArr[newsize].node <- new node(val,done,opid)
74              else
75                    if HeapArr[newsize].infoptr = NULL then
76                          N <- new node(val, BU, opid)
77                          if CAS(HeapArr[newsize].node, NULL, N) then
78                                Opdict.opid.index = newsize
79                                if HeapArr[0].info.opid =
80                                      HeapArr[0].node.opid then
81                                      HeapArr[0].infoptr <- NULL
82                    else
83                          HELP(HeapArr[newsize])
84              return true
85
86        Case 2: cnode.node.status = BU  //till flagging parent
87
88              if cnode.infoptr = NULL then
89                    I <-new BUDesc(NULL, cnode.index(HeapArr), BU,
90                          cnode.node.opid)
91                    if !CAS(cnode.info, NULL, I) then
92                          HELP(cnode)
93              else
94                    pnode = GETPARENT(cindex)
95                    if pnode.status = 'BD' and pnode.infoptr != NULL then
96                          unflag itself and HELP(pnode)
97                          return true
98                    if pnode.infoptr != NULL or pnode.status != done then
99                          HELP(pnode)
100                   CAS(pnode.info, NULL, cnode.info)
101                   HELP(pnode)
102             return true
103
104       Case 3: cnode.node.status = 'done' and cnode.info.op = 'BU'
```

```
105
106               if cnode.info.pindex = NULL
107                     CAS(cnode.info.pindex, NULL, cnode.index(HeapArr))
108               childval = HeapArr[cnode.info.cindex].node.val
109               parval   = HeapArr[cnode.info.pindex].node.val
110               if ( childval >= parval ) then
111                     HeapArr[cnode.info.cindex].node.status <- 'done'
112                     Opdict.(Cnode.info.opid).index <- -1
113
114               else
115                     if cnode.info.pindex = 1 then //root node
116                           P <- new node(childval, done, cnode.info.opid)
117                     else
118                           P <- new node(childval, BU, cnode.info.opid)
119                     if HeapArr[cnode.info.pindex].node.status = 'BD' then
120                           C <- new node(parval, BD,
121                                 HeapArr[cnode.info.pindex].node.opid)
122                     else
123                           C <- new node(parval, done,
124                                 HeapArr[cnode.info.pindex].node.opid)
125                     HeapArr[cnode.info.pindex] <- P
126                     HeapArr[cnode.info.cindex] <- C
127                     if cnode.info.pindex = 1 then
128                           Opdict.(Cnode.info.opid).index <- -1
129                     else
130                           Opdict.(Cnode.info.opid).index <-
131                                                cnode.info.pindex
132
133               if HeapArr[cnode.info.cindex].info.opid =
134                     HeapArr[cnode.info.cindex].node.opid then
135                     HeapArr[cnode.info.cindex].infoptr <- NULL
136               if cnode.info.opid = cnode.node.opid then
137                     cnode.infoptr <- NULL
138
139               return true
140
141         Case 4: cnode.info.op = 'DEL'      // compute size and last node
142
143               if cnode.info.oldsize = NULL then
144                     CAS(cnode.info.oldsize, NULL, cnode.node.value)
145               oldsize <- cnode.info.oldsize
146               if oldsize = 0 then
147                     return emptyheap
148               newsize = oldsize - 1
149               S <- new node(newsize, done, NULL)
150               HeapArr[0].node <- S
151               if HeapArr[oldsize].infoptr = NULL then
152               I <- new sizeDesc(NULL, oldsize, CTR, cnode.info.opid)
153               if CAS(HeapArr[oldsize].info, NULL, I) then
154                     Opdict.opid.index = oldsize
155                     if HeapArr[0].info.opid = cnode.node.opid then
156                           HeapArr[0].infoptr <- NULL
157               HELP(HeapArr[oldsize])
158               return true
159
160         Case 5: cnode.info.status = 'CTR'   //copytoroot
161
```

```
162               if cnode.node.status!= done then
163                     cnode.node.status <- done
164                     oldopid <- cnode.node.opid
165                     Opdict[oldopid].index = -1
166               val = cnode.node.value
167               I <- new delDesc(val, NULL, UPR, opid)
168               if HeapArr[1].infoptr = NULL  then //root flag
169                     if CAS(HeapArr[1].info, NULL, I) then
170                           if cnode.info.opid = cnode.node.opid then
171                           HeapArr[oldsize].infoptr <- NULL
172                           Opdict[cnode.node.opid].index = 1
173
174               if rootval = HELP(HeapArr[1]) is numeric //help anyway
175                     return rootval
176               return true
177
178         Case 6: cnode.info.status = 'UPR'    //update root HeapArr[1]
179
180               if cnode.info.oldval = NULL then
181                     CAS(cnode.info.oldval, NULL, cnode.node.value)
182               R <- new node(cnode.info.value, BD, cnode.info.opid)
183               HeapArr[1].node <- R
184               rootval = HeapArr[1].info.oldval
185               if cnode.info.opid = cnode.opid then
186                     cnode.infoptr <- NULL
187               return rootval
188
189         Case 7: cnode.node.status = 'BD' and cnode.info.lcnode = NULL
190
191               if cnode.infoptr = NULL then
192                     I <- new BDDesc(cnode.index(HeapArr), NULL, NULL, BD,
193                                 cnode.node.opid)
194                     if !CAS(cnode.info, NULL, I) then
195                           HELP(cnode)
196               else
197                     while (cnode.info.lcnode = NULL)
198                           lcnode = GETCHILD(cindex, left)
199                           if lcnode < HeapArr[0].node.val then
200                                 if HeapArr[lcnode].infoptr = NULL then
201                                       CAS(HeapArr[lcnode].info, NULL,
202   node.info)
203                                 if HeapArr[lcnode].infoptr =
204   cnode.infoptr then
205                                       CAS(cnode.info.lcnode, NULL,
206   lcnode)
207
208                                 HELP(HeapArr[lcnode])
209                           else
210                                 CAS(cnode.info.lcnode, NULL, 0)
211
212               return true
213
214         Case 8: cnode.info.status = 'BD' and cnode.info.rcnode = NULL
215
216               while (cnode.info.rcnode = NULL)
217                     rcnode = GETCHILD(cindex, right)
218                     if rcnode < HeapArr[0].node.val then
```

```
                        if HeapArr[rcnode].infoptr = NULL then
                              CAS(HeapArr[rcnode].info, NULL,
cnode.info)
                              if HeapArr[rcnode].infoptr =
cnode.infoptr then
                                    CAS(cnode.info.rcnode, NULL,
rcnode)
                        HELP(HeapArr[rcnode])
                  else
                        CAS(cnode.info.rcnode, NULL, 0)
            return true

      Case 9: cnode.info.status = 'BD'

            lchildval = HeapArr[cnode.info.lcindex].node.val
            rchildval = HeapArr[cnode.info.rcindex].node.val
            parval    = HeapArr[cnode.info.pindex].node.val
            if (lchildval >= parval ) and (rchildval >= parval) then
                  HeapArr[cnode.info.pindex].node.status  <- 'done'
                  Opdict.(Cnode.info.opid).index <- -1
                  HeapArr[cnode.info.lcindex].node.status <- 'done'
                  Opdict.(Cnode.info.opid).index <- -1
                  HeapArr[cnode.info.rcindex].node.status <- 'done'
                  Opdict.(Cnode.info.opid).index <- -1
            else
                  if (lchildval < rchildval)
                        smalval  = lchildval
                        smalidx  = cnode.info.lcindex
                        smalstat =
HeapArr[cnode.info.lcindex].node.status
                  else
                        smalval = rchildval
                        smalidx = cnode.info.rcindex

                  P <- new node(smalval,
                        HeapArr[cnode.info.smalidx].node.status,
                        HeapArr[cnode.info.smalidx].node.opid)
                  C <- new node(parval, BD, cnode.info.opid)

                  HeapArr[cnode.info.pindex] <- P
                  HeapArr[cnode.info.smalidx] <- C
                  Opdict.(Cnode.info.opid).index <- smalidx

            if HeapArr[cnode.info.rcindex].info.opid =
                  HeapArr[cnode.info.rcindex].node.opid then
                  HeapArr[cnode.info.rcindex].infoptr <- NULL
            if HeapArr[cnode.info.lcindex].info.opid =
                  HeapArr[cnode.info.lcindex].node.opid then
                  HeapArr[cnode.info.lcindex].infoptr <- NULL
            if HeapArr[cnode.info.pindex].info.opid =
                  HeapArr[cnode.info.pindex].node.opid then
                  HeapArr[cnode.info.pindex].infoptr <- NULL

            return true

     Case 10: if cnode.node.status = 'done' and cnode.info.op = 'MIN'
                  CAS(cnode.info.val, NULL, cnode.node.val)
```

```
276                     if cnode.info.opid = cnode.opid then
277                           cnode.infoptr <- NULL
278                     return true
279
280         End-evaluate
281   return true
```

Chapter 5

Correctness Proofs

In this chapter, we present a detailed correctness proof of our Heap algorithm. It includes four sections. We will refer to a heap in progress as intermediate heap and a stable heap, for which no update operation is in progress, as actual heap.

Section 5.1 will show some basic properties and observations for our Heap implementation. This includes showing properties that must always hold for a stable heap like parent-child relationship and validity of a node. It will also examine rules for state transitions of a node and how states indicate the overall status of heap.

Section 5.2 will highlight the progress conditions. We will show how different operations make progress concurrently and the role of flagging to ensure there is no ABA problem. We will also prove that the proposed operations are non-blocking.

Section 5.3 will examine how conflicts are handled during an execution on Heap. It will address possible conflicts between INSERT, DELETE, FIND-MIN, BU, BD, CTR and UPR operations. We will look at the possibilities of concurrent operations trying to flag same node and how these situations are handled to maintain non-blocking property of the approach.

Section 5.4 will discuss linearizability of the approach. We will define a linearization point for each operation that terminates.

5.1 Basic Properties

In this section, we will discuss the observations from pseudocode and basic properties that must hold true for the actual heap.

Property 5.1. *Let x and y be node objects. If index(y) = 2 * index(x) or 2* index(x) +1, then x.yalue <= y.value in a Min-Heap. Here, x is the parent node and y is one of the child nodes of x. x is also <= the child nodes of y and so on.*

Property 5.2. *For three nodes x, y, z where index(y) = 2 * index(x) and index(z) = 2 * index(x) + 1, y and z are the child nodes and x.*

x.value <= y.value and x.value <= z.value

Direct relationship between y.value and z.value is not established.

Observation 5.3. *A node x is reachable in actual heap iff*

i. For some index i, Heap[i].nodeptr = x. x is pointed to by an index in Heap.

ii. Index i < Heap[0].nodeptr.value. Index i is less than size of Heap.

iii. x.status = 'done' or 'BD'

Observation 5.4. *For two nodes x and y, Node x < Node y iff x.nodeptr.value is less than y.nodeptr.value.*

Observation 5.5. *Node at location i is flagged if Heap[i].infoptr != NULL. Any update at a location can be done only after successfully flagging the location.*

Property 5.6. *A node is said to be placed in correct position in min-heap if it is greater than its parent and lesser than its child nodes.*

Lemma 5.7. *A node is placed in correct position in heap in our implementation, if*

i. node.status = done

ii. node.infoptr.op != CTR

Proof. Node status is changed to 'done' when it reaches its correct positon in heap. An exception to this case is sub-operation 'Copy- to-Root'. This is performed as a part of DELETE operation. It changes the status of last node of heap to 'done' if it was 'BU' and flags it for copying to root. Hence, even if the node was not in correct position in heap, it is marked as done. On being copied to root, it is 'Bubbled down' to its correct position. □

Observation 5.8. *The contents of a Node object are never changed.*

At any point during an operation, contents of node object are never changed. If node value needs to be changed at any location, new node object is created and node pointer is updated using CAS to point to new node.

Observation 5.9. *No field of an info object is changed except once (if it was previously null).*

Lemma 5.10. *Status of a newly inserted node at the last location in a heap is initially BU. It can only be updated to status 'done'.*

Proof. A new node is created at line 76. Initial status is set to 'BU'. There are four possible operations on this node:

i. Swap due to 'BU' operation initiated by the node itself – update to done or BU

ii. Swap due to 'BU' operation initiated by a child node – update to done or BU

iii. Swap due to 'BD' operation initiated by the parent node – update to done or BU

iv. 'Copy to root' operation initiated by a Delete request – update to done

In all four scenarios, status is either unchanged from 'BU' or updated to 'done'. □

Lemma 5.11. *If a node is in status 'done', it can only be changed to 'BD'.*

Proof. Once a node has attained status 'done', there can be three possible operations on this node:

i. Swap due to 'BU' operation initiated by the child – no change in status

ii. Swap due to 'BD' operation initiated by parent node – no change in status

iii. 'Copy to root' operation initiated by a Delete request – no change in status

iv. 'Update root' operation initiated by a Delete request – update status to 'BD'

In all scenarios, only possible update in status is 'BD'. □

Observation 5.12. *A node can belong to two operation ids.*

Op id 1- The INSERT operation which added node to heap.

Op id 2- DELETE operation that move the node to root location.

Property 5.13. *If node at index 0 has value = 0, it indicates empty heap.*

Observation 5.14. *Node at location 0 (Size of heap) does not give the actual size of heap. It rather gives the number of nodes in heap + number of ongoing insert operations.*

A node is considered part of actual heap only if the status has been changed to 'done' from 'BU'. Nodes with status 'BD' are part of actual heap (using observation that BD status is reached only after done status). Value of 'size' is incremented at the beginning of an insert operation. Thus, even if node is not yet part of actual heap, it is counted towards size of heap.

Actual size of heap = Number of nodes with status 'done' or 'BD'

Observation 5.15. *If size.value = '1', node is inserted with status 'done'. In all other cases, status of a newly inserted node is 'BU'.*

Observation 5.16. *For any index i in Heap, Heap[i].infoptr = null inbetween two sub-operations.*

When a sub-operation completes, it unflags the node by setting Heap[i].infoptr = null. Next sub-operation begins with flagging Heap[i] with new Info object. Example: line 91 flags a node and line 100 unflags it.

Lemma 5.17. *A location i in heap, where I is less than heap size, is empty only if Heap[i].infoptr = null and Heap[i].nodeptr = null.*

Proof.

i. if Heap[i].infoptr points to the Info object with details of operation to be performed at location i in Heap. Heap[i].infoptr is not null if some operation is being executed on the node.

ii. Heap[i].nodeptr points to the actual node at location i in Heap.

iii. It is possible to have Heap[i].infoptr = Not null and Heap[i].nodeptr is null if an Insert operation has been initiated to add node at location i in heap. Location i has been flagged, but node is yet to be added.

iv. It is possible to have Heap[i].infoptr = null and Heap[i].nodeptr not null inbetween sub operations (using Observation 5.16).

Thus, only if both Heap[i].infoptr and Heap[i].nodeptr = null, location i is empty. □

Lemma 5.18. *At most one operation can be performed at a given location in Heap at one time.*

Proof. In order to execute any operation on a location i in heap, it must be flagged by setting Heap[i].infoptr to an Info object. This setting of flag is done using CAS operation.

```
CAS(HeapArr[i].info, Null, Info)
```

This ensures that if multiple threads are trying to flag same location on heap, only one succeeds. The winning thread's operation is executed at that location (using details in Info object). □

Lemma 5.19. *For two reachable nodes x and y, operation id of x != operation id of y.*

Proof. A unique id is assigned to each operation.

i. Initially, Operation id of a node is the id of INSERT operation which added node to the Heap. Since one insert operation adds only one node to heap, id will be unique for every node.

ii. If a node is moved to root location, as part of a DELETE operation then the operation id of node is changed to id of the DELETE operation. Since a delete operation moves only one node to root, id will be unique for every node. □

Property 5.20. *Opid.index in Opdict indicates the Heap location where operation Opid needs to be executed. i.e. the location of the node for which node.operation id = Opid.*

(Location of actual node of Opid, it does not indicate the node Opid might be helping)

Observation 5.21. *There can be at most one BD operation active at root at a given time.*

This follows from Lemma 5.18.

Lemma 5.22. *Node in status 'BD' is a part of the actual heap.*

Proof. DELETE operation on heap changes the status of last node of heap to 'done' if it was 'BU' and flags it for CTR operation. On being copied to root, status is updated to ‘BD’ to move it down the heap to it's correct position. Thus, it is evident that a node is marked as ‘done’ before it’s status is changed to ‘BD’. □

Observation 5.23. *Node in status ‘done’ is not necessarily available to be worked upon by an operation. It can be flagged only if node.infoptr != NULL.*

Status ‘done’ of a node indicates that it is in the right position in heap. It can still be a part of a BU or BD initiated by its child or parent node.

5.2 Progress Conditions

Observation 5.24. *Successful completion of a 'BD' operation on three nodes (parent and two children) ensures that at most one node out of the three is in status 'BD'.*

BD operation is initiated by the parent node. It flags both child nodes in status 'done'/'BU' for operation. If any of the child nodes' status is 'BD', it is worked upon first. This ensures that when operation ends, there can be at most one node in status "BD'. This is evident by referring to images in Section 4.6 for various scenarios of BD.

Lemma 5.25. *Successful completion of a 'BU' operation on two nodes (parent and child) ensures that at most one node out of the two is in status 'BU'.*

Proof. Child node initiates a BU operation. After successful flagging of child, parent node is flagged by setting parent node's infoptr to info object for BU operation. Values are compared for swap. If child is lesser than parent, nodes are not swapped and child is marked as 'done'. If parent is larger, child node moves up the node with status 'BU'. If both parent and child were in status 'BU' as the beginning of operation, child BU first bring parent node to 'done' status and then continues with its own operation. This ensures that a completed 'BU' operation leaves at most one node in status 'BU'. □

Observation 5.26. One BD operation on three nodes (one parent and two children) consists of all nodes in status 'BD' only if the parent node is root of the heap.

Observation 5.27. *There cannot be a chain of continuous nodes in status 'BD'. Two nodes in status 'BD' will always be interleaved with at least one node in status 'done'/'BU' except for root node.*

This follows from Observations 5.24 and 5.26.

Corollary 5.28. *There can be a chain of continuous BU nodes in heap.*

If a continuous stream of insert operation is performed on the heap, new nodes will be added to end of the heap in status 'BU'. This can lead to a chain of nodes where node status is 'BU' in all levels of heap except root node.

Lemma 5.29. *Chain of BU nodes in Heap is deadlock-free.*

Proof. A Bubble Up operation is initiated by child node. It brings the parent node to 'done' status (if previously not 'done') and then makes comparison between the parent and initiating-child node.

Observation 5.30 stated that root node is inserted with status 'done'. In all other cases, status of a newly inserted node is 'BU'. Thus,

i. Topmost parent node in a chain of BU nodes is guaranteed to be in status 'done'.
ii. One of the child nodes of the topmost parent node will succeed in flagging the parent for 'BU' operation and make progress for 'BU' operation.
iii. As supported by previous lemma, now there would be 2 nodes in status 'done'.
iv. By induction, subsequent BU operations will resolve the chain of any number of nodes in status 'BU'.

□

Lemma 5.30. *FIND-MIN operation always returns a valid value of root of heap.*

Proof. FIND-MIN operation flags the root node with Info.value = NULL. This value is updated only if status of root node = 'done'. Line 267 in code `CAS(cnode.info.val, NULL, cnode.node.val)` ensures that the value is updated only once in info structure. Hence, FIND-MIN always returns a valid value.

□

Observation 5.31. *All info flags and nodes are changed using CAS to avoid ABA problem.*

5.3 Conflict Resolution

This section discusses all possible conflicts during an execution on heap.

Observation 5.32. *An INSERT operation cannot start until it is able to flag 'Size' of heap.*

An INSERT operation tries to access the size of heap to locate next available location in heap for inserting new node. Once the location is identified and flagged, new node is inserted with status 'BU' and moved to correct location in Heap by series of BU operations.

Observation 5.33. *A DELETE operation starts only when 'Size' of heap is successfully flagged.*

A DELETE operation tries to access the size of heap to locate last location in heap for swapping with root node. Once the location is identified and flagged, node is moved to root and then Bubbled down to correct location in Heap by series of BD operations.

Observation 5.34. *A FIND-MIN operation does not flag 'Size' of heap.*

FIND-MIN operation works only on root of the heap. It returns a valid value of the root node by directly accessing Heap[1].infoptr and Heap[1].nodeptr.

Lemma 5.35. *'Size' of heap can be modified by only one operation at a time.*

Proof. Size of the Heap is accessed by two operations INSERT and DELETE to locate last used/next available location on heap. Both the operations modify the 'Size' by incrementing/decrementing the existing value. It is necessary that 'Size' of heap is altered only one by operation at a time to avoid accidentally overwriting nodes in heap.

From Observation 5.5, we can conclude that 'Size' can be modified only by obtaining a flag.

Line 23 and 38: `CAS(HeapArr[0].info, NULL, I)` in INSERT and DELETE code ensures that only one amongst any number of concurrent operations will be successful in flagging the size of heap at a time.

□

5.3.1 INSERT-INSERT Conflict

Observation 5.36. *Concurrent INSERT operations are serialized while accessing the 'Size' of heap.*

Observation 5.32 and Lemma 5.35 establish that an INSERT operation starts by trying to flag 'Size', only one will succeed if many compete. Thus, it will serialize concurrent operations at the first step.

Lemma 5.37. *Concurrent INSERT operations never try to insert a new node at same location in Heap.*

Proof. Only one of concurrent INSERT operations can access 'Size' at a given time (Using Observation 5.36). Winning operation increments the 'Size' and then unflags it. Next INSERT operation would work on the incremented value of size. This ensures that two INSERTS are never performed on same location in Heap. □

Observation 5.38. *Multiple INSERT operations make progress concurrently after securing a location in heap.*

Once a location has been identified to insert node in heap, INSERT operations continue to make progress using multiple 'BU' sub-operations until node is placed in right location in heap.

5.3.2 DELETE-DELETE Conflict

Observation 5.39. *Concurrent DELETE operations are serialized while accessing the 'Size' of heap.*

Using Observation 5.33 and Lemma 5.35 we can infer that a DELETE operation starts with attempt to flag 'size'. Use of CAS for flagging selects one as winner among multiple competing threads. Thus, it will serialize concurrent operations at the first step.

Lemma 5.40. *Concurrent DELETE operations never try to replace root of Heap with same node.*

Proof. Only one of concurrent DELETE operations can access 'Size' at a given time (from Observation 5.39). Winning operation decrements the 'Size' (Line 149 `S <- new node(newsize, done, NULL)`) and then unflags it. Next DELETE operation would work on the decremented value of size. This ensures that two DELETES never try to swap root with same node in Heap. □

Lemma 5.41. *Concurrent DELETE operations never return same root node value.*

Proof. We know from Observation 5.5 that root must be flagged before performing any operation on it. Concurrent DELETE operations, will try to flag root node for sub-operation 'UPR – Update root'. Line 167 and 169 in code perform CAS to flag root with new Info object.

```
I <- new delDesc(val, NULL, UPR, opid)
CAS(HeapArr[1].info, NULL, I)
```

CAS will select one winner amongst competing operations. Winning operation will update the root node and return old root value. Root is unflagged after successful completion of one 'UPR' operation and is available to be flagged by another operation. Next winner will work on the updated root node.

Hence, concurrent DELETE operations will not return same root node value. □

Observation 5.42. *Multiple DELETE operations make progress concurrently after updating root of heap.*

Once old root value has been obtained and root of the heap is updated, DELETE operations continue to make progress using multiple 'BD' sub-operations until node is placed in right location in heap.

5.3.3 INSERT-DELETE Conflict

Observation 5.43. *Concurrent INSERT and DELETE operations are serialized while accessing the 'Size' of heap.*

Observation 5.32, Observation 3.33 and Lemma 5.35 establish that both INSERT and DELETE operations starts by trying to flag 'Size', only one will succeeds among many competitors. Thus, it will serialize concurrent operations at the first step.

Property 5.44. *If an INSERT operation is followed by a concurrent DELETE operation, INSERT gets the priority.*

- If an INSERT operation is not yet finished adding node to heap, DELETE will help the INSERT.
- If there is only one node in heap, DELETE will return the value.

  Else, it will update the status of node to 'done' and flag it for operation 'CTR'.
- By updating status to 'done', DELETE completes the INSERT operation and node is added to the actual heap.

Property 5.45. *If a DELETE operation is followed by a concurrent INSERT operation, DELETE gets the priority.*

- If a DELETE operation is not yet finished moving last node to root, INSERT will help DELETE.
- If there is only one node in heap, INSERT will add root to the Heap.

  Else, it will help DELETE until the last location is unflagged.

5.3.4 BU-BU Conflict

Property 5.46. *Multiple BU operations make progress concurrently if they do not work on same nodes.*

Property 5.47. *Concurrent BU operations on same nodes are prioritized in top-down order.*

BU operations are initiated by child nodes.

- If parent is in status 'BU', Child-BU will help parent-BU to complete and bring parent to 'done' status
- If both children of a node are trying to perform BU operation, only one child-BU will succeed in flagging the parent. Line 100 `CAS(pnode.info, NULL, cnode.info)` decides the winning operation. Winner continues to make progress while other child-BU helps the winner.

5.3.5 BD-BD Conflict

Property 5.48. *Multiple BD operations make progress concurrently if they are not being executed on same nodes.*

Property 5.49. *Concurrent BD operations on same nodes are prioritized in bottom-up order.*

BU operations are initiated by parent nodes. If child node is in status 'BD', parent-BD will help child-BD to complete and flag for own operation.

5.3.6 BD-BU Conflict

Property 5.50. *BD and BU operations currently active on mutually exclusive nodes make progress concurrently.*

If the operations are not active on same nodes, they can continue the operation independently. Only if they are working on the same node, their execution steps would conflict.

Lemma 5.51. *If Parent status is BU and child status is BD, there is no conflict.*

Proof. BU operation is initiated by child node and works in upper direction of heap (current level +1). BD operation is initiated by parent node and works with (current level -1) nodes. Thus, if parent node status is BU and child node is BD, both operations will be executed in opposite directions in Heap and will not conflict each other. □

Observation 5.52. *If Parent status is BD and child status is BU, there is conflict.*

This is the opposite scenario of Lemma 5.51. In this case, both the operations will work on same nodes leading to conflict.

Property 5.53. *If Parent status is BD and child status is BU, BD gets the priority.*

Case 1: If parent has been flagged for BD, child-BU operation will unflag the child node, yielding to BD. It will help to complete the parent-BD operation. Line 95 in code accomplishes this task.

Case 2: If parent is unflagged, i.e. BD operation has not started yet, child-BU will make progress and flag the parent for BU operation. Line 100 in code performs this step.

Case 2 ensures that if BU is not yielding to BD forever.

Lemma 5.54. *In case of conflict, BU is starvation-free. It does not yield to BD forever.*

Proof. Case 2 above, shows that BU yields to BD only if BD has been initiated by the parent node. If child-BU finds that parent node's infoptr = NULL, it will perform `CAS(pnode.info, NULL, cnode.info)` in line 100 to flag parent node for BU operation and make progress. This ensures that BU will not starve yielding to BD forever. □

5.3.7 INSERT- FINDMIN Conflict

Observation 5.55. *Concurrent INSERT and FIND-MIN operations conflict only at the root node.*

From Observation 5.34 we know that FIND-MIN does not access size of heap. It works only on root of the heap, hence that is the only point of conflict with an INSERT operation.

Lemma 5.56. *Conflict between Concurrent INSERT and FIND-MIN operations is resolved by flagging.*

Proof. Both the operations will try to flag the root node using CAS(Heap[0].infoptr, NULL, new info) , only one will succeed.

- If INSERT operation wins, FIND-MIN will help INSERT and retry flagging.
- If FIND-MIN operation wins, it will find that Heap[1].node is empty. It will return message that Heap is empty. INSERT will help FIND-MIN.

Hence, conflict is resolved by flagging. □

5.3.8 DELETE- FINDMIN Conflict

Observation 5.57. *Concurrent DELETE and FIND-MIN operations conflict only at the root node.*

Observation 5.34 stated that a FIND-MIN operation does not flag 'Size' of heap. It works only on root of the heap, hence that is the only point of conflict with a DELETE operation.

Observation 5.58. *Conflict between Concurrent DELETE and FIND-MIN operations is resolved by flagging.*

Both the operations will try to flag the root node using CAS(root.infoptr, NULL, new info) , only one will succeed.

- If DELETE operation wins, FIND-MIN will help DELETE and retry flagging.

- If FIND-MIN operation wins, it will return the value of the root in status 'done'. DELETE will help FIND-MIN.

5.3.9 BU - FINDMIN Conflict

Observation 5.59. *Concurrent BU and FIND-MIN operations conflict only at the root node.*

FIND-MIN works only on the root node; hence it can be the only point of conflict with BU operations.

Observation 5.60. *At most two Concurrent BU can conflict with one FIND-MIN operation at a given time.*

Root node has two child nodes who are the possible contenders to initiate a BU operation on root at a given time. BU initiated by these two child nodes will compete with FIND-MIN to flag root node. Since the maximum number of child nodes that root can have is two, we can conclude that at a given time, at most two concurrent BU operations will conflict with one FIND-MIN.

Observation 5.61. *Any number of FIND-MIN operations can conflict with one BU operation at a given time.*

Any number of FIND-MIN operations can be active on the heap. All these would try to flag the root node at same time and would conflict with the BU operations initiated by the children of root. Thus, one BU might compete with any number of FIND-MIN operations.

Observation 5.62. *Conflict between Concurrent BU and FIND-MIN operations is resolved by flagging.*

From Observation 5.59 it is evident that there can be multiple concurrent operations trying to work on root node. All the operations will try to flag the root node using

`CAS(HeapArr[1].infoptr, NULL, new info)` , only one will succeed. Winner will continue its operation on the root node, while others will help the winner complete and retry flagging.

5.3.10 BD- FINDMIN Conflict

Observation 5.63. *Concurrent BD and FIND-MIN operations conflict only at the root node.*

Since FINDMIN works only on root of the heap, that is the only point of conflict with a BD operation.

Observation 5.64. *At most one Concurrent BD can conflict with one FIND-MIN operation at a given time.*

BD operation on root is a sub operation of DELETE. In Observation 5.21 we said that there can be at most one BD operation active on root at a given time. This BD will compete with FIND-MIN to flag root node.

Observation 5.65. *Any number of FIND-MIN operations can conflict with one BU operation at a given time.*

Any number of FIND-MIN operations can be active on the heap at same time. All these would try to flag the root node simultaneously and would conflict with the BD operation on root.

Observation 5.66. *Conflict between Concurrent BD and FIND-MIN operations is resolved by flagging.*

Both the operations will try to flag the root node using `CAS(HeapArr[1].infoptr, NULL, new info)` , only one will succeed. Winner will continue its operation on the root node, while others will help the winner complete and retry flagging.

5.3.11 FINDMIN - FINDMIN Conflict

Observation 5.67. *Conflict between Concurrent FIND-MIN operations is resolved by flagging.*

Any number of FIND-MIN operations can be active on the heap at same time. All these would try to flag the root node simultaneously. Of all threads executing (`CAS(HeapArr[1].infoptr, NULL, new info)`, only one will succeed and continue its operation on the root node. Other threads will retry flagging after helping the winner complete.

5.3.12 CTR - CTR Conflict

Lemma 5.68. *Concurrent CTR operations conflict only at the root node.*

Proof. CTR is a sub-operation of DELETE that operation brings the 'last heap node' to 'done' status and then tries to flag root for 'UPR' operation. In Lemma 5.40 we proved that two DELETE operations cannot work on same 'last heap node'. Hence, we can guarantee that a CTR operation also works on an exclusive node. Next, CTR tries to swap its current node with root of heap. Thus, root is the only point of conflict in concurrent CTR operations. □

Observation 5.69. *Conflict between Concurrent CTR operations is resolved by flagging.*

In continuation to Lemma 5.68, concurrent CTR operations active on more than one nodes attempt to simultaneously flag the root node. Line 458 in code performs this operation: `CAS(HeapArr[1].infoptr, NULL, new info).` Winning thread successfully flags root for UPR and continues operation.

5.3.12 CTR - BU Conflict

Observation 5.70. *Conflict between concurrent BU and CTR operations is resolved by flagging.*

Root node is the point of conflict between BU and CTR operations. BU operation on child nodes of root try to flag root for BU while CTR operation tries to flag root for UPR. All these operations execute CAS on root, hence there is only one winner.

5.3.13 CTR - BD Conflict

Observation 5.71. *Conflict between Concurrent BD and CTR operations is resolved by flagging.*

Concurrent BD and CTR operations try to flag the root node using `CAS(HeapArr[1].infoptr, NULL, new info)` , only one will succeed. Winner will continue it's operation on the root node, while others will help the winner complete and retry flagging.

5.3.14 CTR – FINDMIN Conflict

Observation 5.72. *Conflict between Concurrent FIND-MIN and CTR operations is resolved by flagging.*

We know that FINDMIN works only on root of the node, hence it is the only possible point of conflict. When concurrent FINDMIN and CTR operations try to flag the root node, only one will succeed. CAS will select one winner among competing threads who will continue its operation on the root node.

5.3.15 UPR - UPR Conflict

Lemma 5.73. *UPR operation does not flag any node on heap.*

Proof. CTR and UPR both are sub-operations of DELETE. CTR creates an info object with information required for UPR and flags the root node with that info object. UPR creates a new node using information in info object and updates the old value of root in info object. Then, it

replaces old root with the new node. New node created by UPR has status 'BD'. On successful replacing of root, UPR unflags the root node. In this entire process, no node was flagged as part of UPR operation.

Observation 5.74. *There can be only one active UPR operation at a given time.*

It is clear from Lemma 5.73 that UPR operation works only on root of the heap. It replaces root of the heap with a new node in status 'BD' and returns the old value. UPR sub operation starts only after CTR has flagged the root. Since, there is only one root, there can be only one active UPR operation at a given time.

5.4 Linearization Points

In this section, we will show that the proposed approach is linearizable and each operation has a well-defined linearization point.

Theorem 5.75. *An INSERT operation is linearized when status of the node is changed to 'done'.*

Proof. An INSERT operation adds a new node to next available location in heap. If this is the first node in heap (root node), it is inserted with status 'done'. Else, the new node is inserted with status 'BU'. It is moved up the heap by a series of BU/BD operations until it is smaller than its parent node. Once it finds the correct location, status is changed to 'done'. Execution of line 111 `HeapArr[cnode.info.cindex].node.status <- 'done'` or line 163 `cnode.node.status <- done` is the linearization point of INSERT operation.

When a node is in status 'BU' it is only a part of the intermediate heap. Only at the linearization point, it becomes a part of the actual heap.

If a DELETE operation attempts to replace root with another node, it makes sure that status of the node is 'done'. This is because root node cannot be replaced with a value that is not a part of the actual heap. If node status is 'BU', INSERT changes the status to 'done' in line 162: `if cnode.node.status!= done then`

`cnode.node.status <- done.` □

Theorem 5.76. *A DELETE operation is linearized when root is successfully replaced with new value.*

Proof. A DELETE operation returns the root value and replaces root with last node in heap. Status of the new root node is set to 'BD' and it is moved down the heap by a series of BD/BU operations till it is smaller than it's child nodes.

Linearization point of a DELETE operation is the point when UPR sub-operation successfully replaces root with new node (last node of heap). Line 183 `HeapArr[1].node <- R` where `R <- new node(cnode.info.value, BD, cnode.info.opid).`

At this point, we also have the last value of root node stored in the info structure which is returned by the UPR function. Line 180 in code.

```
if cnode.info.oldval = NULL then
        CAS(cnode.info.oldval, NULL, cnode.node.value)
```
□

Theorem 5.77. A FIND-MIN operation is linearized when value of root is successfully saved in info structure.

Proof. A FIND-MIN operation flags root node in status 'done' with an info structure for find-min. This info structure has an empty placeholder for value of root. After successful flagging of root

in line 54 `CAS(HeapArr[1].info, NULL, I)`, where `I <- new SizeDesc(NULL, NULL, MIN, opid)` we are sure that no more changes can be made to root until FIND-MIN unflags the root□ Next, in line 267 `CAS(cnode.info.val, NULL, cnode.node.val)` value of root is stored in info structure. This CAS will be successful only for the first time when placeholder is empty. This will ensure that a valid value of root in status 'done' is returned by the FIND-MIN operation.

Note: If several concurrent operations are linearized at exactly the same point during an execution, the order of their linearization points is arbitrary.

Chapter 6

Conclusion and Future Work

We proposed a non-blocking implementation of a heap in which all updates to shared nodes of the heap are done atomically in an asynchronous shared memory system. Our implementation offers *lock-free* insert, delete and *wait-free* find-min operations. This implementation is free of the drawbacks of blocking algorithms and ensures that if there is an active thread in the system, operations will complete. Concurrent movement of nodes in the same or opposite directions in the heap does not block the way of threads trying to insert or delete nodes for the heap. Our approach is an amalgamation of ideas from various existing ideas and implementations, with a few heap-specific tricks. We have represented the heap as an array of pointers as opposed to the traditional representation as an array. Potential conflicts in operations when multiple threads are trying to access the same node are confronted by the usage of *flagging* and *yielding*. We established the linearizability of the algorithm which proves it is theoretically correct.

We have identified two improvement areas in the algorithm. Currently, heap size is a bottleneck. Any insert or delete operation must update the single shared variable size at the first step. If this can be avoided in some way, it would make the algorithm more efficient. Insert and delete operations might be improved by selecting random locations for insertion and deletion to reduce contention on the same nodes.

A thread helps other operations if it encounters them on its path in the heap. If a thread crashes leaving an operation incomplete after the linearization point, and no other active thread works on the same node, we cannot guarantee wait-freedom in such a scenario. Adding more details to the help mechanism such that a thread is guaranteed to be helped within a fixed time, would strengthen progress condition of the algorithm.

Implementing this algorithm and conducting experimental evaluation for performance would be an interesting future work. It would give a better insight into possible conflicting scenarios and the ABA problem. A similar technique can be applied to other data structures and different types of heaps.

Bibliography

[1] M. Herlihy, *Wait-free synchronization*, ACM Transactions on Programming Languages and Systems (TOPLAS), 1991.

[2] M. Herlihy and J. M. Wing, *Linearizability: A correctness condition for concurrent objects*, ACM Transactions on Programming Languages and Systems (TOPLAS), 1990.

[3] Mark D. Hill and Michael R. Marty, *Amdahl's Law in the Multicore Era*, University of Wisconsin-Madison, Google, July 2008.

[4] A. Silberschatz, P. Galvin and G. Gagne, Operating system concepts, John Wiley and Sons Inc, 2005

[5] A. F. *Babich, Proving Total Correctness of Parallel Programs,* IEEE Transactions on Software Engineering, *1979*.

[6] D. Dice, D. Hendler, I. Mirsky, *Software-based contention management for efficient compare-and-swap operations,* Wiley Online Library (wileyonlinelibrary.com), 2014.

[7] H. Attiya, A. Castañeda, D. Hendler, *Nontrivial and Universal Helping for Wait-Free Queues and Stacks*, Leibniz International proceedings in Informatics Schloss Dagstuhl, Germany.

[8] R. Ayani, *LR-algorithm: Concurrent Operations on Priority Queues*, In proceedings of 2nd IEEE Symposium on Parallel and Distributed Processing, 1991.

[9] V. N. Rao and V. Kumar, *Concurrent Access of Priority Queues*, IEEE Transactions on Computers, Dec 1988.

[10] G. Barnes, *Wait-Free Algorithms for Heaps*, University of Washington Seattle, 1992.

[11] M. Herlihy and N. Shavit*, The Art of Multiprocessor Programming*, Revised Reprint. Elsevier, 2012.

[12] M. Herlihy, *A methodology for implementing highly concurrent data objects*, ACM Transactions on Programming Languages and Systems (TOPLAS), 1993.

[13] M. Herlihy, *Impossibihty and universality results for wait-free synchronization*, In proceedings of 7th ACM Symposium on Principles of Distributed Computing, Aug 1988.

[14] M. Herlihy and J. Wing, *Axioms for concurrent objects*, In proceedings of ACM Symposium on Principles of Programming Languages, Jan 1987.

[15] H. Attiya and E. Hillel, *Built-in coloring for highly-concurrent doubly-linked lists*, Theory of Computing Systems, 2013.

[16] M. Fomitchev and E. Ruppert, *Lock-free linked lists and skip lists*, In proceedings of 23rd Annual ACM Symposium on Principles of Distributed Computing, 2004.

[17] A. Israeli and L. Rappoport. *Efficient Wait-Free Implementation of a Concurrent Priority Queue,* 7th International Workshop on Distributed Algorithms, Springer-Verlag, Sep 1993.

[18] Wikipedia, Multi-Core Processor, https://simple.wikipedia.org/wiki/Multi-core_processor

[19] L. Lamport, *Specifying concurrent program modules*, ACM Transactions on Programming Languages Systems, April 1983.

[20] M. Greenwald. *Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS*, 21st Symposium on Principles of Distributed Computing, 2002.

[21] S. Plotkin, *Sticky bits and universality of consensus*, In proceedings of 8th ACM Symposium on Principles of Distributed Computing, 1989.

[22] T. Harris, *A pragmatic implementation of non-blocking linked-lists*, International Symposium on Distributed Computing, 2001.

[23] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan, *Cbtree: A practical concurrent self-adjusting search tree*, Distributed Computing, Springer, 2012.

[24] N. Shaifei, Non-Blocking Doubly Linked Lists with good Amortized Complexity, Leibniz International proceedings in Informatics Schloss Dagstuhl, Germany, 2013.

[25] T. Crain, V. Gramoli, and M. Raynal, *No hot spot non-blocking skip list*, IEEE 33rd International Conference on Distributed Computing Systems, 2013.

[26] F. Ellen, P. Fatourou, E. Ruppert, and F. Breugel, *Non-blocking binary search trees*, In proceedings of 29th ACM symposium on Principles of distributed computing, 2010.

[27] A. Natarajan, L. H. Savoie, and N. Mittal, *Concurrent wait-free red black trees*, Stabilization, Safety, and Security of Distributed Systems, Springer, 2013.

[28] A. Patrizio*, In Smartphones and Tablets, Multicore is Not Necessarily the Way to Go,* Smartbear Blog, July 2013.

[29] W. Scherer, D. Lea, M. Scott, *Scalable Synchronous Queues, Communications of the ACM 2009.*

[30] H. Sundell and P. Tsigas, *Lock-free deques and doubly linked lists*, Journal of Parallel and Distributed Computing, 2008.

[31] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. *Wait-free linked-lists*. In proceedings of 16th Annual Symposium on Principles of Distributed Systems, 2012.

[32] J. Valois, *Lock-free linked lists using compare-and-swap*, In proceedings of 14th ACM Symposium on Principles of Distributed Computing, 1995.

[33] M. Michael and M. Scott, *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms, PODC, Colorado 2006*

[34] Wikipedia, *Compare and swap*, http://en.wikipedia.org/wiki/Compare-and-swap.

[35] R. K. Treiber, *Systems programming: Coping with parallelism*. IBM, Thomas J. Watson Research Center, 1986.

[36] M. Michael and M. Scott, *Nonblocking algorithms and preemption-safe locking on multi-programmed shared memory multiprocessors*, Journal of Parallel and Distributed Computing, 1998.

[37] Wikipedia, Heap, https://en.wikipedia.org/wiki/Heap_(data_structure)

[38] M. Moir and N. Shavit, *Concurrent data structures*, Handbook of Data Structures and Applications, 2007.

Curriculum Vitae

Rashmi Niyolia, M.S.

Degree:

Master of Science in Computer Science 2016
University of Nevada Las Vegas

Thesis Title: Novel Non-Blocking Approach for a Concurrent Heap

Thesis Examination Committee:

Chairperson, Dr. Ajoy K. Datta, Ph.D.
Committee Member, Dr. John Minor, Ph.D.
Committee Member, Dr. Ju Yeon Jo, Ph.D.
Graduate Faculty Representative, Dr. Emma E. Regentova, Ph.D.