A Study on the FEAST Algorithm

Natalia Colmenares
Department of Computer Science
University of Central Florida
Orlando, Florida USA
natcolchu@knights.ucf.edu

Aedan Deyto
Department of Computer Science
University of Central Florida
Orlando, Florida USA
aedandeyto22@gmail.com

Eduardo Bourget
Department of Computer Science
University of Central Florida
Orlando, Florida USA
eduardobourget@gmail.com

Joel Joy
Department of Computer Science
University of Central Florida
Orlando, Florida USA
joeljoy1912@gmail.com

***Abstract*—Concurrent binary search trees are a relevant research topic. We investigate the proposed implementation of the FEAST algorithm, a fast concurrent lock free concurrent binary search tree, to determine the feasibility of implementation on an Intel Core i7 8th generation processor with four cores and eight logical cores. Additionally, implementation in other languages, such as C++, Java, and Rust were tested to determine the portability of the proposed algorithm. We conclude the paper with results of concurrent testing for each of the implementations we completed.**

I. Introduction

There have been remarkable developments in the field of multiprocessor programming as well as multi-core systems in recent years. With the rise of such technologies, it is important to properly utilize the power of the multiple processors to operate on overlapping regions of the data structures at the same time. Concurrency has often been achieved by way of locks and lock-based algorithms being implemented in various data structures, such as Queues, Linked Lists, Hash Tables, Skip Lists, and Tries. When locks are being utilized in these data structures, a certain process can restrict access to a particular portion of the data structure for an extended period of time. Because of this, lock-based algorithms that implement these data structures are prone to deadlocks, and performance issues.

One of the fundamental data structures in computer science are Binary Search Trees (BSTs). They are used to store and organize ordered data and they support the insert, delete, and search operations. As it happens, generalized attempts to derive a concurrent non-blocking BST from its sequential version tends to be quite inefficient, but the FEAST (Fast Concurrent Algorithm for Binary Search Tree) implementation guarantees a lock-free manipulation of a concurrent BST in an asynchronous memory system that supports all of the essential BST functions: insert, search and delete. In stark contrast to most of the existing algorithms that implement a concurrent BST, the FEAST algorithm is edge-based rather than node-based.

The proposed FEAST algorithm by Natarajan et al. explores how to implement the lightweight lock-free binary search tree, but there is not a study on the FEAST algorithm implementation in other languages. In fact, the experimental evaluation for the FEAST algorithm was created with the Intel C++ compiler (version 18.0.2). Thus, the purpose of this paper aims to answer the following questions:

1. What kind of modifications to the program will be necessary to implement this algorithm in different programming languages?
2. Is the efficiency of the program affected when it is implemented in different languages? If so, is there a particular programming language that is substantially more efficient than others when implementing this algorithm?

II. Background and Related Work

A. Motivation

Binary Search Trees are data structures that support dynamic-set operations such as search, insert, delete, minimum, maximum, predecessor and successor, and are represented with key fields and node objects [3]. The node objects contain left and right pointers to their respective node objects. The support of such operations allows for an organized representation of data, where keys, the field in which the organization revolves around, satisfy the binary-search-property: Child nodes that have keys larger than the parent node's are expected to station themselves in the left subtree of the parent node, while nodes with keys smaller than the current parent node's are positioned to be in the right subtree of the parent. The efficiency of the dynamic-set operations, O(log n) time on average, where n is the number of nodes in a binary search tree [3], is one of the main reasons a concurrent representation of the data structure has been studied.

Due to the spread of multiprocessor architectures, studies on popular sequential data structures and how to represent them concurrently have already been implemented in program libraries; thus, research on them continues to progress. For example, the Harris-Michael algorithm as a representation of a concurrent linked-list is one used in the Java Concurrency Package [6]. There are various methods to implement a concurrent data structure by using a blocking or a non-blocking strategy. BSTs that have used locks have been tested and proven correct [5], but due to the use of locks, other operations, such as search, can be blocked. Blocking can prevent other threads from making progress if there is an unexpected delay in one thread [6]. Although a blocking implementation may prove easier to implement and test, a non-blocking property is attractive since a delay in one thread may not result in preventing other threads from making progress. The wait-free property and the lock-free property are examples of non-blocking progress conditions. In a wait-free property, every thread that takes steps is guaranteed to make progress. In contrast, a lock-free property states infinitely often, at least one of the operations eventually completes. Although a wait-free property sounds generous, wait-free algorithms may prove to be slower than lock-free algorithms. Thus, "A fast lock-free algorithm may be more attractive than a slower wait-free algorithm" [6]. Current research on blocking BSTs include CASTLE [9], BST-TK [4], and CITRUS [1]. In addition, Non-blocking BSTs include RM-BST[8], Non-blocking Binary Search Trees by Ellen et al. [5], and FEAST [7].

FEAST is described as a lightweight lock-free concurrent binary search tree. Due to the non-blocking progress condition with a lock-free property, this algorithm was of the options studied to implement a concurrent BST. The work on FEAST presents an experimental evaluation with a few of the mentioned concurrent BST implementations above: CASTLE, BST-TK, CITRUS, and RM-BST. Using system throughput as an evaluation metric, the performances of the mentioned algorithms were compared with FEAST on a multi-core machine. Results of the experiment described FEAST as having the best performance out of the five BST algorithms with smaller key spaces when contention was high [7]. Thus, due to the superior performance of FEAST for small and medium key spaces, we decided to study the FEAST algorithm as a concurrent BST.

B. Overview of the FEAST Algorithm

The FEAST algorithm is a lock-free algorithm used for concurrent manipulation of a binary search tree in an asynchronous shared memory system that supports search, insert, and delete operations. In contrast to other concurrent algorithms, this algorithm is edge-based rather than node-based. This means that when compared to other algorithms for a binary search tree, the insert and delete operations will work on a smaller tree section, execute fewer instructions and/or use fewer dynamically allocated objects.

This algorithm uses an external representation instead of an internal representation. The main comparison between these two is that in the internal representation, the data is stored in every node: both internal and leaf nodes, while In an external representation, the data is stored only in leaves [2, 7]. The internal nodes are used for routing purposes only. The main advantage of using the external representation over the other is that it dramatically simplifies the tree's search, insert, and delete operations in a concurrent environment [7]. Both the insert and delete operations use a constant-size window and do not cause keys to move from one location in the tree to another. As a result, when looking for a key, the traversal does not need to start if the key is not found. It is better to choose the internal representation when looking for a smaller memory footprint and a better cache performance since it contains about half the number of nodes than the external representation.

Every operation of the algorithm will start with a seek phase, in which the operation traverses the search tree starting from the root node until it reaches a leaf node. The path traversed by the operation in this phase as the access-path and the last three nodes on that access-path as the grandparent, parent and terminal nodes, respectively. The operation then compares the key stored in the terminal node, on the access-path, and according to that result of the comparison and the type of operation, it will either terminate or move to the next phase.

The modifying operations like insert and delete obtain ownership of the edges it needs to work on. The delete operation will obtain ownership of an edge by marking it. By marking, it indicated that either both its tail and head nodes, or only its tail node would be removed from the tree. The first case is referred to as flagging and tagging as the second type. To enable this tagging or flagging of an edge, two bits from each child field stored at a node are stolen, denoting the two bits by flag and tag. A value of 1 in the respective field means that the corresponding edge has been flagged (or tagged). Contrastingly, a value of 0 exhibits a case where the edge is unflagged (or untagged).

To demonstrate the correctness of the algorithm, FEAST uses linearizability as the safety property and lock freedom as the liveness property. This means that the linearizability property requires that an operation should appear to take effect instantaneously at some point during its execution [6]. Having lock-freedom requires that if a process takes an infinite number of steps, then at least one operation eventually completes.

C. Details of the Algorithm

$\boldsymbol{Seek}$ **:** Most of the dynamic-set operations make use of the seek function to return the access-path from a given key. The access-path is described as a simple path from the root node to the leaf node. The ancestor, successor, and parent variables in the function are first initialized with the sentinel nodes in the tree for traversal. Traversal begins when the address of the next node (the current node's address field) on the access-path exists. With every iteration of the while loop, we attempt to extend the access-path by examining the state of the nodes and continuously updating the ancestor, successor, parent, and current nodes. Once the terminal node points to a leaf node, the loop breaks and the final values stored in the local ancestor, successor, parent, and terminal variables are returned in the form of a SeekRecord object. The SeekRecord class holds the ancestor node (the grand-parent of the terminal node), successor, parent, current, and terminal nodes. Functions make use of this class to be able to traverse and modify the tree effectively.

$\boldsymbol{Insert}$ **:** The insert operation runs inside a while loop and only returns when a successful insert occurs or through the identification of an existing object with the same key value. The function starts execution by calling the seek function to find parent and terminal nodes in the tree. Once the address of the child field is obtained through the parent node, two new nodes are created: an internal node (newInternal) and a leaf node. The newIntermal node will contain the larger variable between the key to insert and the key of the terminal node, while the new leaf node will hold the key to insert. A subtree is then created to insert into the tree, where newInternal acts as a root and newLeaf and the terminal nodes as the child nodes. A CAS operation is then performed to attempt to insert the subtree into the tree. The CAS operation is only successful if the references did not change in the time between initializing new nodes and getting the address of the next child field, and if the stamp value is not tagged nor flagged. In other cases, if insertion fails due to changing states of a node being flagged or tagged, cleanup is called to delete the flagged nodes before attempting to insert again.

$\boldsymbol{Delete}$ **:** The delete function will mark the node that needs to be deleted and try to delete it. We do so by first setting the mode to injection: Mark the leaf node. Within the injection mode, a search for the key in the tree occurs by using the SeekRecord object called by the delete function. If the key is not present in the tree, the function immediately returns. Otherwise, we attempt to flag the edge of the leaf node and then try to physically remove the leaf node from the tree by calling the cleanup function. At this point, the mode is transferred to cleanup. The delete operation is persistent with physically deleting the flagged node during cleanup mode and will not return until successful.

$\boldsymbol{Cleanup}$ **:** Cleanup will physically remove leaf nodes and its parent from the tree when marked for deletion. Compared to all the other main functions in the algorithm it will need to be sent a SeekRecord object instead of a key. Using the SeekRecord object, it will fill in its local nodes of ancestor, successor, parent, and terminal nodes. Then it will create the variables addressOfSuccessorField, addressOfChildField, and addressOfSiblingField to hold an edge. The addressOfSuccessorField will contain the edge between the ancestor and successor. The addressOfChildField will get the edge between parent and terminal, and the addressOfSiblingField will get the other edge that parent holds. After, we get the stamp of addressOfChildField, if the stamp is not flagged, the sibling field needs to be deleted, so set the addressOfSiblingField to addressOfChildField. Then, we check if the stamp of addressOfChildField is tagged. If tagged, we try to tag the addressOfSiblingField. Subsequently, we attempt to make the sibling node a direct child of the ancestor node with a CAS operation. The result of the CAS operation is returned as the return value of cleanup.

III. Methodology

The algorithm was first implemented in the Java language since knowledge about pointers and bit manipulation were not necessary. Once the implementation was done, tests were created to run the methods sequentially in order to verify correctness of the program. This included testing insert, search, delete, and delete and insert simultaneously. After sequential testing, concurrent testing was completed with 4 threads that tested on different iteration sizes. Iterations chosen were 4,000, 40,000, and 400,000. This would allow us to conclude whether the increase in execution time proportionally increased with the iterations. Additionally, the iteration sizes would not require us to utilize a more powerful computer. Concurrent tests included calls on insert, search, delete, and insert and delete simultaneously. Average time consumption in milliseconds was used as a metric to evaluate the performance of the algorithm since we expect a different performance between our processor, and the processor used to test in the paper when retrieving results for the FEAST algorithm. To retrieve the average execution time, each test was run five times. Running the tests five times would allow us to view if any anomalies occurred. If all tests remained in the same range, then it was concluded that the average could correctly describe the execution time. Other languages tested were C++ and Rust. If the implementation was successful the steps described above were replicated. A modified version of Java and C++ meant to speed up execution time based on a proposed modification in the FEAST algorithm was also tested to verify if it resulted in faster execution time than the original implementations.

A. Java Implementation

The supported dynamic-set operations in our lock free BST algorithm are seek, input, and delete. The components of the tree are a node which contains the key, and Java's AtomicStampedReference class for the left child and right child of the node. The stamped value will represent if the edge between the node and its individual child have been tagged or flagged. In our code, we agreed to use a two digit system based on 1's and 0's. The first digit describes the flagged state and the second digit represents the tagged state. The number 00 represents a stamp that is neither tagged nor flagged. In contrast, 10 is used to mark that it is flagged, but not tagged. The number 01 illustrates that it is tagged and not flagged, and 11 is used to represent a stamp that is both tagged and flagged. The decision to use a digit representation for the states was chosen because AtomicMarkableReference allowed only one markable bit. Additionally, the algorithm always compares a clean state (00) in a CAS operation; thus, using a digit representation does not provide any setbacks. In addition, three sentinel keys are present and never removed from the tree. The sentinel keys must be greater than any other keys.

B. C++ Implementation

Differing from the Java implementation, C++ utilized a SeekRecord struct and a node struct to represent the classes we used. Additionally, due to the lack of a class that produces the same results as an AtomicStampedReference, we had to directly manipulate the unused bits by "stealing" bits from the pointer values. Thus, learning bit manipulation and pointers in C++ was needed in order to produce correct results. The first attempt to steal the bits from the pointer values included utilizing a uintptr_t to turn the pointer into an integer. Specifically, reinterpret_cast<std::uintptr_t>(pointer) was used. Therefore, in order to return the value into a pointer, the opposite, reinterpret_cast<Node*>(x), would occur. One problem that arose was the inability to tag and flag simultaneously. Therefore, another suggestion was to utilize an unsigned long instead (reinterpret_cast<unsigned long>(x)). This returned the correct results and allowed the required bit manipulation to extract flagged and tagged states, and a node address.

Defined functions, such as GET_ADDR(x), GET_TAG(x), GET_FLAG(x) were used to retrieve the flag, tag, and address as proposed by the paper. Thus, in order to properly retrieve a key from a stored pointer value, GET_ADDR(x)->key was needed. Without getting the proper node address, the key value would return 0 since at that point it would return the value in a single word, not the node reference.

Atomic functions such as __sync__bool__compare__and__swap() were investigated due to its ability to return a boolean to depict the results of the CAS operation and its ability to support a state change with a node reference. For example, FLAGGED(UNTAGGED(x)) would return a node reference with a flagged state and an untagged state. This could be used in the CAS operation to detail the new value that should be used if the expected value is found. More importantly, the compiler can directly translate the provided builtin atomic into machine code, which we expect would speed up execution time. Additionally, because we were testing on an Intel processor, we knew this operation would be compatible.

C. Rust Implementation

The Rust implementation proved difficult to complete. In the end, we were unable to finish the implementation because of a lack of time and knowledge to properly implement the FEAST algorithm within Rust. One of the biggest issues we faced was trying to share information between a function and a struct. Additionally, when trying to implement the FEAST algorithm, we were primarily having issues trying to figure out a way to create the AtomicStampedReference object from Java into Rust since one did not previously exist. The algorithm requires us to have an object that can hold both the reference of the node, and the edge it is pointing towards as a way to show if the edge is either marked or flagged. When trying to implement the AtomicStampedReference into Rust we first tried using a tuple that could hold an atomic integer and an atomic pointer to the child nodes. However, we were having trouble with updating the contents inside the tuple safely. As a result, we then attempted to have a marked atomic integer in the node itself, but that implementation was unsuccessful as well.

When we were trying to figure out a way to implement the FEAST algorithm we first tried to create a basic non concurrent BST. We used Option<Box<Node>> so that we could tell if the child nodes were empty. This would allow us to have an indefinite amount of nodes in our BST. Node references were then made atomic in order to make it thread safe. Subsequently, we tried to create another struct to hold our SeekRecord. When trying to implement the seek function, sending references of the nodes failed because atomic references could not be cloned. At the very end, we felt that because we were not progressing with the Rust implementation, we decided to focus on implementing the modified version of the FEAST algorithm with local recovery in Java and C++.

D. Modified FEAST Algorithm

When it comes to the modified FEAST algorithm, they add in a stack that allows the algorithm to achieve local recovery. This means that the algorithm would not have to traverse all over again from the top, and instead just start at the node wherever it failed. Previously, if an insert or delete failed it would call seek again and seek would start traversing the BST from the very top, at the root. But, with the given stack, it contains the full traversal path from the initial seek; therefore, when calling seek again we could just backtrack to the proper position we would need to be at during the insert or delete functions. Also we do not have to worry about making the stack accessible to everyone since each stack is local; therefore, the stack does not need to be a concurrent data structure.

When implementing the stack, we had to create our own struct and its own method class, or add to the given methods, since some of the important methods we needed were not in the java or C++ stack library. For example, getting the second from the head elements, and initializing the stack to make sure it has the first two sentinel nodes it will encounter in its traversal are not a normal stack methods and had to be implemented. When it comes to editing our actual class to add in the stack, most of the functions all stay the same but we just add in the stack object and initialize the stack on add and delete. Seek was the only class that had the biggest difference, where instead of starting at the root, we start at the node at the head of the stack and from there we would either traverse down the BST and pop in our new traversed nodes or backtrack by popping the stack.

***Java* :** The stack class in Java was implemented through scratch, rather than using the Stack class in Java. Due to Java running comparatively slower than C++ owing to the Java language being interpreted, another object initialization was unnecessary if all methods were not going to be used. In fact, every call to insert(), and delete() would have had to initialize another Stack object. Additionally, other methods would have to be added onto the Stack class to complete the requirements of the modified algorithm. Therefore, creating the class would prove to be more efficient. Methods included were common ones such as push(), pop(), and peek(). But other methods added were getSecondToTop(), and initializeTraversalStack().

One of the biggest challenges we had to deal with was during the delete method. We found that delete would sometimes receive a null pointer because of how seek would remove all the sentinel nodes in the bottom of the stack. Delete or Cleanup up would receive a sentinel node as a terminal node, which causes the sentinel node to be marked which is not appropriate behavior. However, after a bit of testing and tweaking, we found that the issue was caused during the seek operation. When the seek operation traversed through the BST, if it encountered a flagged node, it would remove the node by creating a SeekRecord object and sending it to clean up. We found that the SeekRecord object it created was incorrect; therefore, after fixing that issue we were able to make it properly delete nodes.

$C++$: The stack was implemented through the std::stack class. Various functions such as push(), pop(), and top() were used to recreate the modified algorithm. As mentioned previously, a function that gets the second element from the top of the stack does not exist in the C++ stack library. Therefore, a simple implementation utilized a temporary Node to hold the Node at the top of the stack. Once the top of the stack was pushed, the top() function could then retrieve the second element that was on the stack. After storing that value in a variable, the temporary Node pushed the original head back to the top of the stack. In C++, we chose to pass the root of the BST in every function to avoid declaring a global root variable. Therefore, when initializing the stack with the sentinel nodes, push(root) and push(root->left) was utilized.

IV. Testing

We tested all of our code on a Lenovo Yoga Laptop that has an Intel core i7 8th gen processor with 4 cores and 8 logical cores. We tested our original Java FEAST algorithm first to ensure that we could implement it in other languages. Our first set of testing was to make sure that it could work sequentially for correctness. After seeing that the algorithm worked sequentially, we started testing our code concurrently. This would allow us to view how fast the algorithm would take to execute and finish with four threads.

A. Java

In our concurrent testing, we had four different tests for insert, search, delete, and a mixture of insert and delete. In the testing code, we had each test have a total of four threads and each thread would perform their task a certain amount of iterations. As a result, we had testing for a total of 4000, 40000, and 400000 iterations for the Java implementation.

TABLE I
Java Results in milliseconds

Iterations	***Insert***	***Delete***	***Search***	***Insert and Delete***
4,000	17.2ms	9ms	4ms	18.4ms
40,000	42.6ms	21.8ms	18.2ms	44.6ms
400,000	200ms	91.6ms	120.6ms	205.6ms

B. C++

When it came to testing the C++ implementation, we mimicked the Java concurrent testing framework described above on the same computer. Thus, our tests consisted of insert, delete, search, and a mix of both insert and delete. In each of these tests, we tested with 4 threads total and each thread doing a total of either 1000, 10000, or 100000 iterations. Therefore, we made each test do 4000, 40000, and 400000 iterations. Mimicking the Java testing framework would allow us to compare the C++ and Java implementations equally.

TABLE II
C++ Results in milliseconds

Iterations	***Insert***	***Delete***	***Search***	***Insert and Delete***
4,000	1.9ms	1.8ms	1.8ms	2.1ms
40,000	14.2ms	13.3ms	12.8ms	18.4ms
400,000	127.4ms	115.3ms	123.1ms	180.2ms

C. Java (Modified)

After finishing up the testing for our original Java program that implemented the FEAST algorithm, we then started to test the modified Java algorithm. When testing the modified algorithm we were able to successfully test insert, search, and a mixture of insert and delete. Results below show that in most cases, the modified version of the Java implementation ran slower than the unmodified version.

TABLE III
Java (Modified) Results in milliseconds

Iterations	***Insert***	***Delete***	***Search***	***Insert and Delete***
4,000	27.4ms	6.4	4.6ms	23.6
40,000	77.6ms	41.6	35.4ms	68.8
400,000	320.8ms	142.2	145.2ms	297.2

D. C++ (Modified)

For testing the C++ modified implementation, we used the exact same test we used in our original C++ test since we wanted to see how the C++ modified implementation compared to the original C++ implementation.

TABLE IV
C++(Modified) Results in milliseconds

Iterations	***Insert***	***Delete***	***Search***	***Insert and Delete***
4,000	3.5ms	4.2ms	1.9ms	3.8ms
40,000	41.2ms	36.8ms	12.9ms	37.9ms
400,000	386.2ms	333.9ms	123.6ms	385.1ms

V. Evaluation

We have implemented the FEAST algorithm in Java and C++ as well as their modified versions. The Java implementation of FEAST does not seem to perform as well as the C++ implementation in the execution of a mix of Insert and Delete operations together. However, the Java implementation does perform slightly better than the C++ implementation in the 400,000 iteration test in the execution of the delete and search operations. In the 40,000 and 4,000 iteration tests, however, the Java implementation does not outperform the C++ implementation in the execution of any of the operations.

We also implemented the modified versions of the Java and C++ implementations that were suggested by the research paper that we were referencing. Contrary to our assumptions, we found that the modified versions of the Java and C++ implementations did not perform better than their original counterparts. For the modified Java implementation, the insert operation in all iterations took almost twice the amount of time to complete the task than its unmodified counterpart. On the other hand, the search operations for the modified Java implement took approximately the same amount of time as its unmodified counterpart. In the case of the modified C++ implementation, both the insert and delete operations for all iterations took approximately twice the amount of time taken by its unmodified counterpart. The search operation for the modified C++ implementation was relatively similar in performance to its unmodified counterpart.

In conclusion, we have observed that the Java implementation of the FEAST algorithm performs worse overall than the C++ implementation of the algorithm, despite its better performance in search operations. We have also found that the modified versions of the Java and C++ implementations are demonstrably worse in performance than both of the original Java and C++ implementations. A reason as to why the modified version of the algorithms did not perform as well could be due to the number of threads. When the researchers tested the FEAST algorithm, the algorithm was tested with 6 to 96 threads. Contrastingly, we tested the algorithm with 4 threads due to limitations on the compute power.

VI. Discussion

This work evaluated the performance and implementation of the FEAST algorithm in various languages to determine the portability of the algorithm, and the difference in overall performance. Through testing, we were able to determine that the best overall performance with 4 threads was in the unmodified version of C++.

The implementation in Java proved to be more convenient to create due to its builtin memory management (i.e. Java's garbage collection). Additionally, developers do not have to worry about the portability of the program with the Java Virtual Machine, which provides a portable execution environment. Pointer manipulation was also unnecessary in the Java implementation with existing classes such as the AtomicStampedReference; but, caution must be taken when implementing the flagged and tagged states when mimicking two bits. Fortunately, the FEAST algorithm never required a CAS operation on two unclean states.

C++ allows for greater control over other topics such as memory, packing sub-fields into a single word for less space, and compiler optimization with builtin atomics. Additionally, options such as pass by reference and pass by value are available to the developer, allowing for greater control over addresses and copies of values. Another challenge that comes with C++ is ensuring that the complete program is portable, since C++ does not have a virtual machine. Thus, knowledge of the architecture is needed when diving into more complex topics, such as word alignment. As a result, one can recommend the FEAST algorithm implementation in Java when efficiency, optimization and manual memory management is not a concern. Otherwise, C++'s implementation proves to be more complex but valuable when considering speed.

VII. Future Work

In the future, several ideas can be implemented which can drastically change the experimental results and further develop this study implementation beyond Binary Search Trees. The first that can be done is to expand the hardware possibilities for testing. This would enable the addition of threads and iterations that can be done on the insert, delete, search, and mix of insert and delete. Another idea that can be implemented is the use of the framework of testing sequentially, and concurrently, on different sized iterations, to test additional concurrent data structures beyond the primary binary trees we have currently. One exciting data structure is the Trie. A Trie is an ordered tree structure used mostly for compactly storing strings. This can have a maximum of 26 nodes where you can store data; this makes it a fast retrieval data structure that facilitates faster access to an element. Moreover, this opens the idea of a mass implementation to various tree data structures that can have enormous variances in their test results, which could impact critical elements in different research worldwide.

References

[1] Arbel, M., AND Attiya, H. 2014. Concurrent updates with RCU: Search tree as an example. In Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC'14). ACM, New York, NY, 196–205.

[2] Arbel-Raviv, M., Brown, T., AND Morrison, A. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. Proceedings of the 2018 USENIX Annual Technical Conference.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L.,, AND Stein, C. (2001). Introduction to Algorithms. The MIT Press. ISBN: 0262032937

[4] David, T., Guerraoui, R., AND Trigonakis, V. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. In Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15). ACM, New York, NY, 631–644.

[5] Ellen, Fatourou, P., Ruppert, E., AND van Breugel, F. (2010). Non-blocking binary search trees. Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 131–140. https://doi.org/10.1145/1835698.1835736

[6] Herlihy, M., AND Shavit, N. 2008. The Art of Multiprocessor Programming. Morgan Kaufmann. ISBN: 0123705916

[7] Natarajan, A., Ramachandran, A., AND Mittal, N. 2020. FEAST: A Lightweight Lock-free Concurrent Binary Search Tree. ACM Trans. Parallel Comput. 7, 2, Article 10 (June 2020), 64 pages. DOI:https://doi.org/10.1145/3391438

[8] Ramachandran, A., AND Mittal, N. 2015. A fast lock-free internal binary search tree. In Proceedings of the 16th International Conference on Distributed Computing and Networking (ICDCN'15). ACM, New York, NY, Article 37, 10 pages.

[9] Ramachandran, A., AND Mittal, N. 2015. CASTLE: Fast concurrent internal binary search tree using edge-based locking. In Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'15). ACM, New York, NY, 281–282.