15-418/618 Project Final Report
Concurrent Lock-free BST

Names: Swapnil Pimpale, Romit Kudtarkar
AndrewID: spimpale, rkudtark

1.0 SUMMARY

We implemented two concurrent binary search trees (BSTs): a fine-grained-locking BST (FG_BST) and a lock-free BST (LF_BST). The FG_BST is our own original implementation, and the LF_BST is a BST presented by Shane V. Howley and Jeremy Jones. We compared the performance of both BSTs on 4 different types of traces: read intensive low contention traces, read intensive high contention traces, write intensive low contention traces and write intensive high contention traces. We also attempted to augment the LF_BST with hazard pointers to handle memory reclamation, and compared the performance of our augmented LF_BST against the original LF_BST presented by Shane V. Howley and Jeremy Jones.

2.0 BST DATA STRUCTURE

Both the FG_BST and LF_BST implement the set interface by storing unique integers at each node within the tree. Both BSTs support 3 operations: *insert, search* and *delete*. The insert operation adds a new node to the tree with the specified value, provided the value is not already present in the tree. The search operation traverses the tree looking for the node with the specified value, and reports whether or not this value has been found. The delete operation removes the node with the specified value from the tree, and replaces the removed node with an appropriate successor/predecessor if necessary. Naturally, none of these operations are allowed to alter the BST invariant that all values in the left subtree of a node are smaller than the value stored at the node and that all values in the right subtree of a node are larger than the value stored at the node.

3.0 FINE-GRAINED-LOCKING BST

We will begin our analysis of the FG_BST by providing an overview of the concept of fine-grained-locking, and how it relates to our BST. We will then discuss the specific implementation details of each of the 3 operations supported by our FG_BST.

3.1 Overview of fine-grained-locking and BST

Fine-grained-locking enables multiples threads to read and modify different parts of a data structure at the same time, as opposed to coarse-grained-locking, which only permits a single thread to read modify the data structure at once. In the case of our BST, fine-grained-locking is implemented by having each node in the tree contain a single lock. The implementation details are shown below:

```
typedef struct Fine_Grained_BST_Node {

      int value;
      struct Fine_Grained_BST_Node *left;
      struct Fine_Grained_BST_Node *right;
      struct Fine_Grained_BST_Node *parent;
      pthread_mutex_t lock;

}FG_BST_Node;
```

As can be seen above, each node in the tree is represented by a structure containing a single integer value, a single mutex lock, and pointers to a left child, a right child and a parent node. The mutex is used to perform the fine-grained-locking over each node in the tree. Any thread attempting to modify any field of a particular node must first acquire exclusive access to the node by locking the mutex. This is demonstrated in the sections below, where we describe each operation in detail.

3.2 Search operation

The search operation takes in an integer value as input, and traverses the BST appropriately, looking for the node with the specified value. The search operation is implemented via hand-over-hand locking, where the parent node is locked first, followed by the appropriate left or right child node (depending on whether the search traverses the left subtree or the right subtree).

Once the child node's lock is obtained, the parent node's lock is released. This process of hand-over-hand locking repeats itself while traversing the BST, until the node with the specified input integer value is found or we reach a null pointer. In either case, the thread then proceeds to release any remaining locks before reporting whether or not the input integer value was found. A visual depiction of the search algorithm can be found in the appendix.

3.3 Insert operation

The insert operation works in the same manner as the search operation; it takes in an integer value as input, and traverses the BST appropriately, looking for the node with the specified value, using hand-over-hand locking once again. Once again, the process of hand-over-hand locking repeats itself while traversing the BST, until the node with the specified input integer value is found or we reach a null pointer. In the case where we find the node with the specified input integer value, the insert operation fails since the BST does not allow duplicate elements, so we release all held locks and return. In the case where we reach a null pointer, we allocate a new node with the specified input integer value, and set the single node we currently have a lock on to have either it's left or right child pointer to point to this new node as specified by the BST invariant. We also have the newly allocated node's parent pointer to point to the node we currently have a lock on. Once this is done, we release the lock on this node and return. A visual depiction of the insert algorithm can be found in the appendix.

3.4 Delete operation

The delete operation has two phases; the first phase requires a hand-over-hand-locking style traversal of the tree to find the node to be deleted (TBD node), and the second phase requires another hand-over-hand-locking style traversal of the right or left subtree of the TBD node to find a successor (smallest value in right subtree) or predecessor (largest value in left subtree) to replace the node to be deleted. Note that the second phase will not get executed if the node to be deleted is a leaf node.

3.4.1 *Delete - First phase*

During the first phase, the BST is traversed appropriately via hand-over-hand locking to find the TBD node with the specified input integer value. The main difference during this traversal is that after we acquire the lock on the TBD node, we return from the traversal without releasing the lock on the TBD node's parent (i.e. we return from the traversal holding both the locks on the TBD node and the TBD node's parent). If the TBD node is not found, the delete has failed, so we release our remaining held lock and return from the delete function.

Assuming we find the TBD node, we know that we now have a lock on the TBD node and its parent node. If the TBD node is a leaf node, we set the TBD's parent node's appropriate child pointer to NULL, unlock the TBD node, free it, unlock the TBD's parent node, and successfully return from the delete function.

3.4.2 *Delete - Second phase*

If the TBD node is not a leaf node, we release the lock on TBD's parent node, and move on to phase 2, where we traverse either the right or left subtree of the TBD node looking for an appropriate successor or predecessor (respectively) to replace the TBD node. This traversal is once again done in a hand-over-hand style, with the additional requirement that we return from the traversal holding locks on both the successor/predecessor and their respective parent nodes.

(Note that at this stage, a successor/predecessor is guaranteed to be found, since we hold a lock on the TBD node and know for a fact that it has at least one child. If the child is a leaf, then it cannot be deleted by another thread, since we hold a lock on the child's immediate parent; TBD. If the child is not a leaf, it may get deleted, but it will not be freed, but rather have it's value replaced by it's appropriate successor or predecessor. So either way, TBD will still have a valid child to pick as a successor/predecessor.)

Once we return from this second traversal, we now have locks on the TBD node, the successor/predecessor node, and the successor/predecessor's parent node. Note that the successor

node is the smallest node in the right subtree rooted at the TBD node, and that the predecessor node is the largest node in the left subtree rooted at the TBD node. This means that the successor and predecessor node can have 1 most one child; a right child for the successor node or a left child for the predecessor node.

If a child does not exist, then the successor/predecessor node is a leaf node, so we have all the necessary information and locks to complete our deletion. We copy over the value of the successor/predecessor node to the TBD node, we unlock the successor/predecessor node and free it, we set the successor/predecessor's parent node to have it's appropriate child pointer point to NULL, and we release the locks on the TBD node and the successor/predecessor's parent node.

If a child does exist, then we obtain a lock on that child node. We now have 4 locks on 4 nodes: the TBD node, the successor/predecessor's parent node, the successor/predecessor node, and the appropriate child of the successor/predecessor node. We copy over the value of the successor/predecessor node to the TBD node, we set the successor/predecessor's parent node to have it's appropriate child pointer point to the child of the successor/predecessor node, we set the child of the successor/predecessor node to have it's parent pointer point to the successor/predecessor's parent node, we unlock the successor/predecessor node and free it, and then release the remaining three locks on the successor/predecessor's child and parent nodes and the TBD node.

A visual depiction of the delete algorithm with a successor replacement can be found in the appendix.

3.5 Special case: locking the root

Locking the root is a special case, since it is possible to free the root (and consequently the lock stored in the root) while another thread may be waiting on the lock for the root. To avoid this, we require that any thread attempting to lock the root must first lock a global 'tree lock' that is used

to gain access to the root. Once the tree lock is obtained, the root lock can be obtained, and the tree lock can be released for other threads waiting to begin their operations.

4.0 LF_BST ALGORITHM

We will begin our analysis of the LF_BST by providing an overview of the concept of lock-freedom in data structures, and how it relates to our BST. We will then discuss the specific implementation details of each of the 3 operations supported by our LF_BST.

4.1 Overview of lock-freedom and BST

Lock-freedom in data structures refers to a concept whereby concurrent access to a data structure can be performed in a non-blocking manner that guarantees system-wide progress. The traditional approach to concurrent data structures involves the use of locks to perform synchronization, which results in a blocking algorithm due to the fact that any thread holding a lock may be swapped out of an execution context by the operating system, leaving other threads unable to make progress on the system.

Lock-free data structures attempt to avoid the blocked nature of locked data structures by using hardware synchronization primitives such as *compare-and-swap* to atomically read-modify-write shared memory values. However, these hardware synchronization primitives are often restricted to atomically read-modify-write one or two memory locations at most, meaning they do not provide the same level of exclusivity or isolation that locked data structures do. For this reason, lock-free data structures that have several points of contention often employ algorithms that are cooperative in nature to ensure correctness alongside system-wide progress.

The implementation details of the LF_BST (shown below) allude to the cooperative nature of the algorithm:

```
//Enum constants
enum flag_type {
      NONE = 0,
```

```
	MARK,
	CHILDCAS,
	RELOCATE
};

//Structures
typedef struct Lock_Free_BST_Node {
	int volatile key;
	void *op;
	struct Lock_Free_BST_Node *left;
	struct Lock_Free_BST_Node *right;
}LF_BST_Node;

typedef struct Child_Compare_And_Swap_Operation {
	bool is_left;
	LF_BST_Node *expected;
	LF_BST_Node *update;
}Child_CAS_OP;

typedef struct Relocate_Operation {
	int state;
	LF_BST_Node *dest;
	void *dest_op;
	int remove_key;
	int replace_key;
}Relocate_OP;

//pointer manipulation functions
void *SET_FLAG(void *ptr, int state);
int GET_FLAG(void *ptr);
void *UNFLAG(void *ptr);
void *SET_NULL(void *ptr);
bool IS_NULL(void *ptr);
```

```
//helper functions
void help(LF_BST_Node *pred, void *pred_op, LF_BST_Node *curr, void
*curr_op);
void helpChildCAS(Child_CAS_OP *op, LF_BST_Node *dest);
void helpMarked(LF_BST_Node *pred, void *pred_op, LF_BST_Node *curr);
bool helpRelocate(Relocate_OP *op, LF_BST_Node *pred, void *pred_op,
LF_BST_Node *curr);
```

4.1.1 *Structures*

As can be seen above, each node in the tree is represented by a structure containing a single integer value, pointers to a left child and a right child, and a void pointer to a single operation structure (*op)*. The operation structure is used to indicate one of the two types of operations that can be performed on nodes in the BST: a Child-Compare-And-Swap-Operation (*Child_CAS_OP*) or a Relocate-Operation (*Relocate_OP*).

A *Child_CAS_OP* structure contains all the necessary information needed for any thread to complete an operation which modifies a given node's child pointers. The *is_left* variable indicates whether the child pointer to be modified is the left or the right child of the node, the *expected* variable is the pre-modified value of the soon-to-be-modified child pointer variable, and the *update* variable is the new value of the soon-to-be modified pointer variable.

Similarly, the *Relocate_OP* structure contains all the necessary information needed for any thread to complete a delete operation for a node with 2 children. The *dest* variable is a pointer to the *LF_BST_Node* who is to be removed, the *dest_op* variable is a pointer to the operation structure (*op)* contained in the *dest* node, the *remove_key* variable is the integer value contained at *dest*, and *replace_key* is the successor's key that will replace the *remove_key* value in the *dest* node.

The specifics of how each of these operation structures are used will be discussed when we go over how each of the BSTs 3 operations are implemented.

4.1.2 *Pointer manipulation functions: SET_FLAG, GET_FLAG and UNFLAG*

The first 3 pointer manipulation functions: SET_FLAG, GET_FLAG and UNFLAG are used on the operation pointer (*void *op*) in the *LF_BST_Node* structure to embed and retrieve information about the state of the current operation being performed on the node. Since memory addresses on 32-bit and 64-bit machines are 4-byte and 8-byte aligned respectively, auxiliary information about the state of the node can be embedded in the last 2 bits of the address of the operation pointer (*void *op*) by performing simple bitwise operations on the pointer. For the purposes of our BST, each node can be in one of 4 states:

NONE - There is no operation being performed on this node
MARK - This node has been marked as logically removed from the tree, so it needs to be compare-and-swapped out of the tree
CHILDCAS - This node is currently undergoing a *Child_CAS_OP*, i.e. one of its child pointers is being changed
RELOCATE - This node is currently being affected by a delete operation

We can embed this state information in the operation pointer of any node by calling the SET_FLAG function, which simply performs a logical 'OR' of the operation pointer and any of the values in the *enum flag_type* table. We use similar bitwise operations in the GET_FLAG function to retrieve state information from the operation pointer, and the UNFLAG function to retrieve the original pointer value.

4.1.3 *More pointer manipulation functions: SET_NULL and IS_NULL*

Similar to the previous functions, the SET_NULL and IS_NULL functions also perform some bitwise operations to encode and decode auxiliary information about what state the node is in. However, unlike the previous functions, the SET_NULL and IS_NULL functions are used on

any pointer that points to a *LF_BST_Node*, rather than the operation field (*void *op)* of a *LF_BST_Node*. The SET_NULL function sets the last bit of any *LF_BST_Node* pointer, and the IS_NULL function returns whether or not a given *LF_BST_Node* pointer has it's last bit set or not. These functions effectively allow us to have unique null pointers for every *LF_BST_Node*, which helps deal with the ABA problem. Specifically, if a child pointer for a node is originally NULL (x1, because last bit is set)), and then is set to a valid node (x8e3f0000), and then set back to NULL (x8e3f0001), the change is reflected since x8e3f0001 is not equal to x1.

4.1.4 *Helper functions*

The *help* function makes calls to 3 helper functions: *helpChildCAS, helpMarked, and helpRelocate*.

The *helpChildCAS* function does the actual work of performing a compare and swap to alter the child pointers of a given node. It takes in a pointer to a *Child_CAS_OP* structure and a pointer to a *dest* node whose child pointer is to be modified, and uses a compare-and-swap to atomically read and modify the appropriate child pointer of the *dest* node. It then uses another compare-and-swap to atomically read and reset the operation field of the *dest* node from CHILDCAS to NONE (indicating that no operation is being performed on the node anymore).

The *helpMarked* function does the work of removing a marked node from the tree by making a call to the *helpChildCAS* function. The *helpMarked* function takes in a parent node *pred*, the operation associated with the parent *pred_op*, and *curr*, the marked child node of *pred* that is to be removed. It constructs a *Child_CAS_OP* structure with the appropriate parameters, and uses a compare-and-swap to set the operation on the *pred* node to be a CHILDCAS. If the compare-and-swap succeeds, it calls *helpChildCAS* with the *Child_CAS_OP* structure and the *pred* node. Note that this function is only called when the marked child *curr* has 1 or fewer children.

The *helpRelocate* function is used to perform deletes on nodes in the tree that have two children. It takes in a *Relocate_OP op*, a node *curr,* which is either the successor node or the node to be deleted (which can happen if *helpRelocate* is called from the find function), and the parent node *pred* of the *curr* node and the operation associated with the *pred* node. Note that the *helpRelocate* function is called with the successor nodes' operation field already set to RELOCATE. The first thing the *helpRelocate* function tries to do is to set the operation field of the *op->dest* node (the node that is getting deleted i.e. TBD node) to RELOCATE via a compare-and-swap.

If it successfully manages to do this, or if the operation field of the TBD node is already set to RELOCATE (because another thread performed the compare-and-swap first), it attempts to set the state of the *Relocate_OP* to be successful via a compare-and-swap. If this compare-and-swap succeeds, the function attempts to use two more compare-and-swaps to replace value in the TBD node (which is equal to *op->remove_key*) with the value of *op->replace_key* and to reset the operation field of the TBD node from RELOCATE to NONE (the value has been copied over to the TBD node, so it is free of any operations at this point).

The second part of *helpRelocate()* performs cleanup on the *successor* node by either marking and excising it, if the relocation was successful, or clearing its *op* field if the relocation failed. If the relocate operation was successful and the *successor* was marked, it is excised using *helpMarked()*.

4.2 Search operation

The search operation calls the *find()* function with the key value to be searched and the root of the tree. The *find()* function embodies the *cooperative* nature of the lock-free BST. The *find()* function traverses the tree searching for the appropriate node. For every node in its path, it checks to see if there is an on-going operation at that node by calling the GET_FLAG function on the *op* pointer of that node. If there is, it "temporarily" suspends the *find()* and helps complete the on-going operation at the node, before retrying the *find()* it was previously trying to do. This

is how system-wide progress is achieved through *cooperation* among concurrent operations contending for the same node.

If a node's key is replaced as a result of a *remove* operation, this affects the range of keys that can possibly be contained in each of its subtrees. Replacing a node's key with the next larger value expands the range of keys contained in the left subtree and shrinks the range of keys contained in the right subtree. Expansion of the range of keys does not affect ongoing searches in that subtree because the key to be searched will still be present in the left subtree but shrinking of the range of keys may require any ongoing searches in that subtree to restart. To detect such a change in the range of keys, the *find()* function keeps track of the last node for which the right path was taken *(last_right)*.

find() retries the operation in the following two cases:

1. If it couldn't find the key and the operation pointer of *last_right* changed.
2. If the operation pointer of the current node changed - done to ensure that the current node was not modified between reading its *op* and reading its *key*.

find() function returns one of the following four values:

1. FOUND: if the key is found
2. NOTFOUND_L: if the key is not found but would be positioned at the left child of *curr* if it were inserted
3. NOTFOUND_R: if the key is not found but would be positioned at the right child of *curr* if it were inserted
4. ABORT: if the search of a subtree cannot return a usable result

4.3 Insert operation

The *add()* function implements the insert operation. Since duplicates are not allowed, it first checks if the key to be added is already present in the set. Once the key is verified not to be in the tree and an insertion point found, a new BST node is created. This is followed by the creation

of a new *Child_CAS_Op* object. The *Child_CAS_Op* contains the information necessary to complete the insert operation, and is inserted into *curr's op* field using CAS. If the CAS succeeds, the node can be considered to be logically inserted into the tree. It then calls *helpChildCAS(),* which does the actual work of physically adding the node to the tree and resetting *curr's* operation pointer to from *CHILDCAS* to *NONE.* If the above CAS fails, the insert operation is restarted.

4.4 Delete operation

The *remove()* function implements the delete operation. It first calls *find()* to find the node of interest. Once found, it checks the number of child nodes the node to be deleted (TBD) has.

If TBD has one or less child nodes, its removal is simple. TBD's operation pointer *op* is marked using a compare-and-swap. At which point, TBD can be considered to be logically removed from the tree. *helpMarked()* is then called to perform the actual removal. TBD's predecessor's left or right child is set to NULL if TBD is a leaf node, otherwise it is set to TBD's only child.

If TBD has two children, then a second *find()* is done to find the node with the next largest key - TBD's *successor.* A *Relocate_Op* is then allocated and filled-in with information necessary for the completion of the remove operation. This *Relocate_Op* is then inserted into the *successor's* operation field using CAS. This is done to ensure that the successor doesn't get deleted during this operation is going on. Once this CAS is successful, the actual swapping of key values is done by calling *helpRelocate().*

5.0 HAZARD POINTERS

Dynamic memory management of nodes in a lock-free data structure pose a problem in environments where automatic garbage collection is not supported. This is because in a lock-free data structure each thread is guaranteed to have unrestricted access to any node at any time. This makes it tricky to decide when a particular node can actually be "freed". When a thread frees a node, it is possible that some other contending thread could hold a reference to the freed node. If the memory associated with the node is freed to the OS before the contending thread tries to access it, it may result in fatal errors.

For this reason, the lock-free design in the reference paper does not implement a memory manager or a garbage collector. The paper's design holds on to all the memory allocated during an entire run. This makes it impractical to be used for any real-world BST implementation.

We attempted to solve this problem by implementing hazard pointers. Hazard pointers are single-writer, multiple-readers pointers. Each thread maintains its own list of hazard pointers indicating which nodes the thread is currently accessing. Whenever a thread detaches a node from the tree it does not immediately free up the node. Instead, it stores that node in a *"retired nodes"* list *(rlist)*. The *rlist* is also maintained on a per-thread basis, and each thread is only permitted to modify its own *rlist*. When the number of nodes in the *rlist* goes above an arbitrarily set threshold, the thread performs a scan of its *rlist*. During the scan, every node in the thread's *rlist* is checked against the hazard pointers list of all the other threads. If any thread's hazard pointer list contains this node, it is unsafe to free the node and hence freeing up of the node is deferred until the next scan. If no thread's hazard pointer list contains the node then it is safe to free the node.

Our hazard pointers implementation is not 100% complete at this point. We have narrowed down the problem to a bug in our implementation for a particular mixed workload case wherein we insert a few elements in the BST, delete all of them and then reinsert elements. It works perfectly fine for workloads with inserts only, searches only, deletes only, combination of searches and deletes, combination of all inserts followed by all searches followed by all deletes.

6.0 TESTING

We built a test suite around our project early on, since we believed benchmarking would be a crucial aspect of this project. On a high level, the test suite could be divided into 3 parts: work creation, work distribution and test cases.

We built a trace generator which is capable of generating sequential, random/mixed and low contention workloads. A sequential workload is one where the inserts, deletes and searches are performed on nodes with increasing or decreasing key values. We believe this type of a workload would be a high contention workload since all the threads would be working on the same part of the tree. A low contention workload is one where the operations would be performed on different parts of the tree. This is done by accessing high and low key values alternately. Finally, a mixed workload has a random mix of insert, delete and search operations.

We used the synchronized work queue from one of the class assignments to store all the work from the trace file. Our test framework creates multiple pthreads, each of which grabs work from the queue and executes it. For the low contention workloads, where we want to start with a particular tree structure, the tree is created using a single thread initially to ensure we get the exact tree structure required.

Correctness of both the fine-grained and lock-free versions is tested in the following two ways:

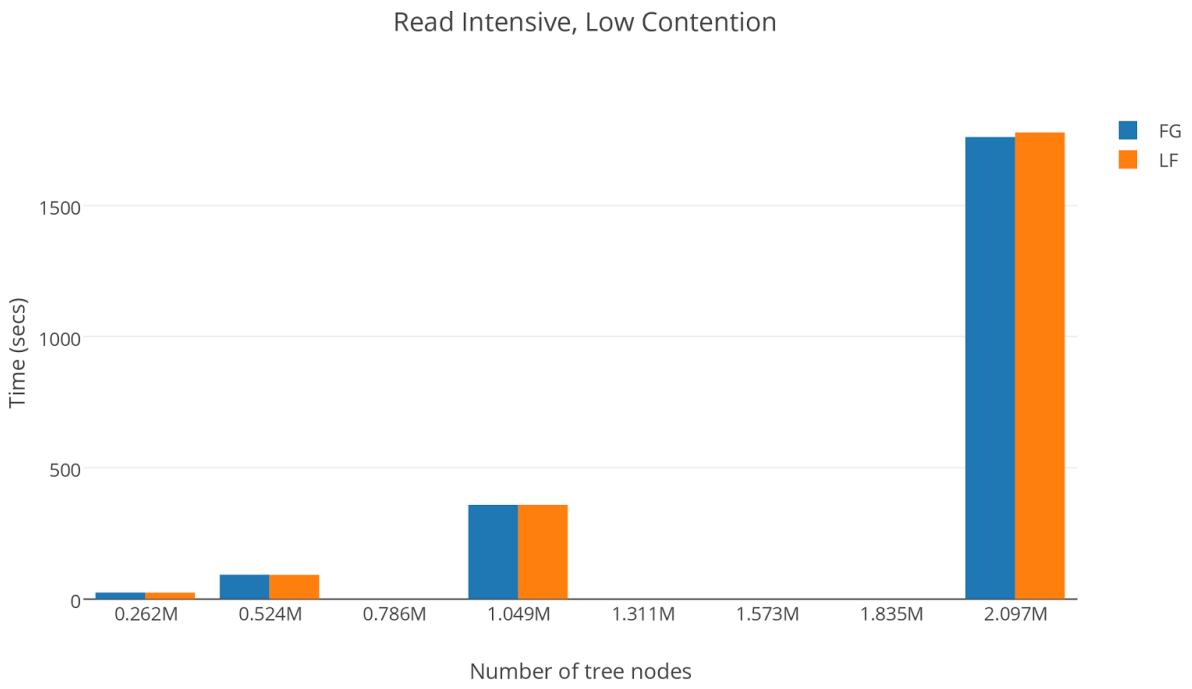
- Since there are multiple threads performing operations on the tree simultaneously the final BST may be different across different runs of the same trace file. We test whether the final BST satisfies the BST invariant (every value in the left subtree < node's value and every value in the right subtree > node's value) by performing an inorder traversal on the final BST.
- We start with a particular tree and perform a fixed number of delete operations on it. At the end of the run we check if the number of nodes remaining and their values are as expected.

7.0 RESULTS

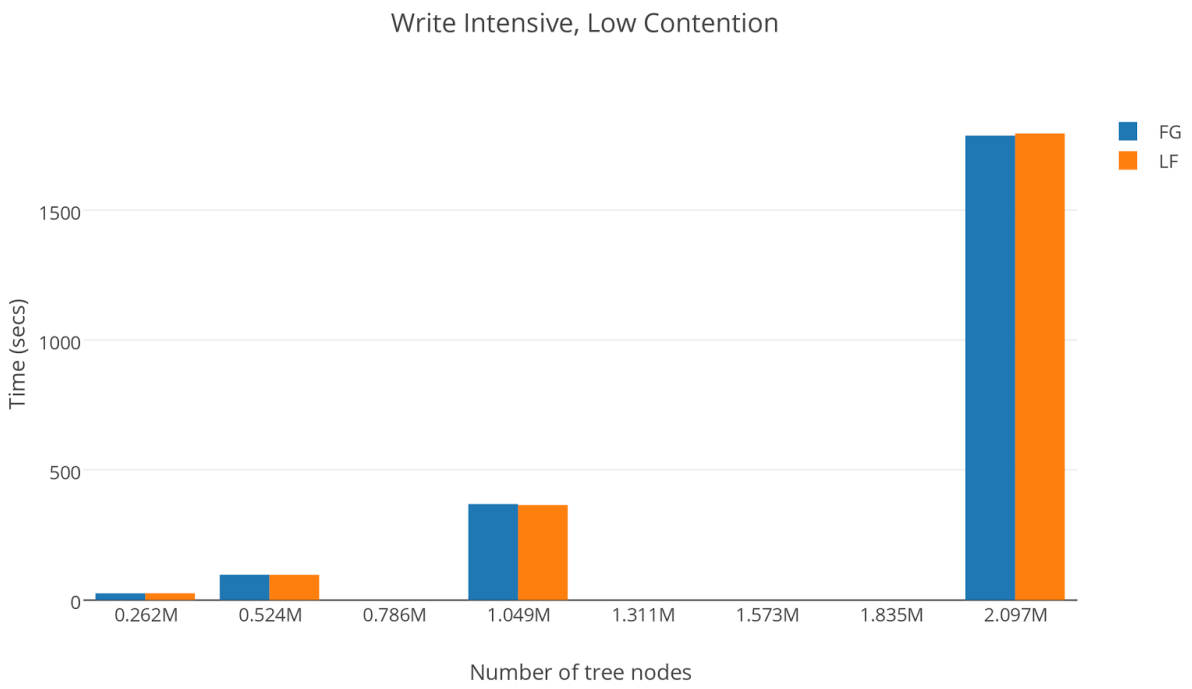
7.1 Platform used:

We used GHC 27-46 machines for our testing. These machines have six-core, hyper-threaded, 3.2 GHz CPUs (so a total of 12 virtual cores)

7.2 Workload: Read Intensive, Low Contention

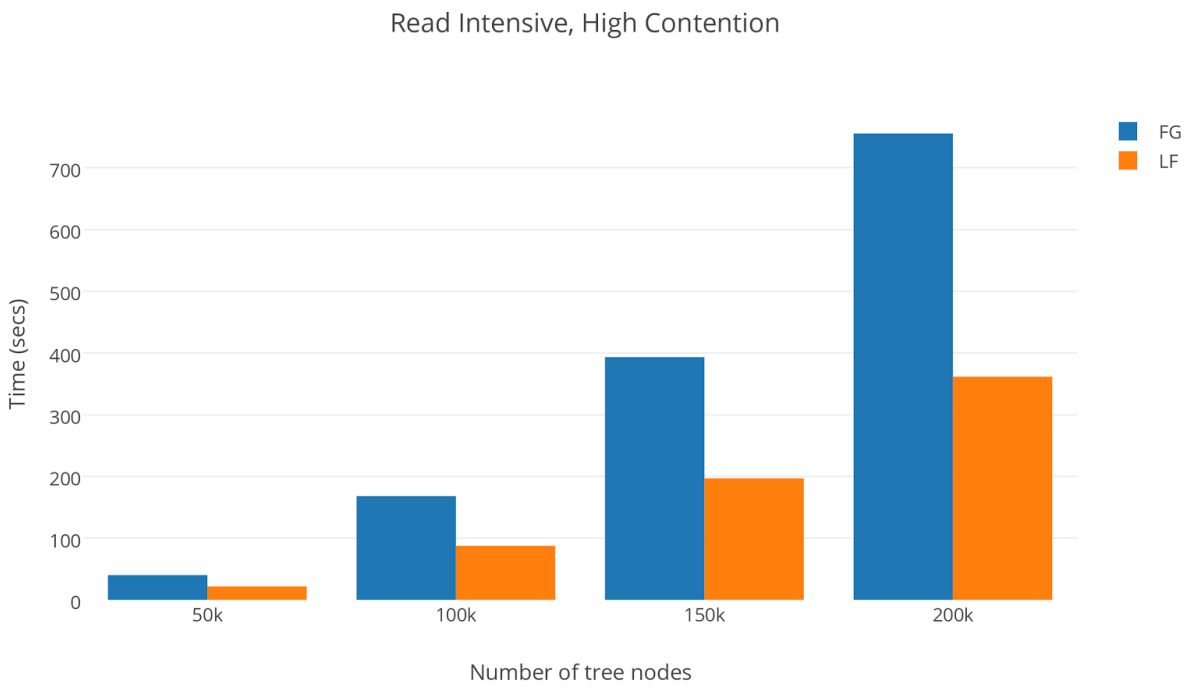


7.3 Workload: Write Intensive, Low Contention

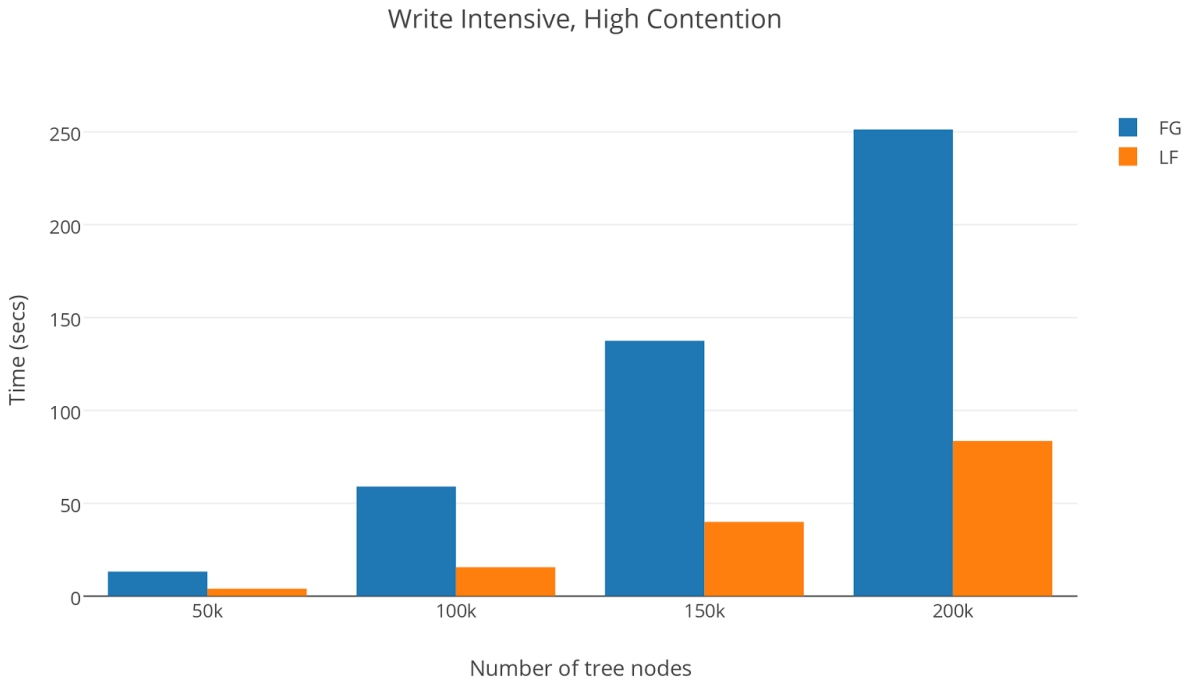


As described above, the low contention workload generates accesses to different parts of the tree. This allows for a lot of operations to be done parallely without much interference from other threads. The read intensive workload had all search operations and the write intensive workload had all insert operations. Both the read and write intensive workloads were run with 262144, 524288, 1048576 and 2097152 nodes. It can be seen that the fine-grained version works almost as good as the lock-free version. Thus, we concluded that our fine-grained version holds only the absolutely necessary locks and allows for maximal parallelism whenever possible.

7.4 Workload: Read Intensive, High Contention

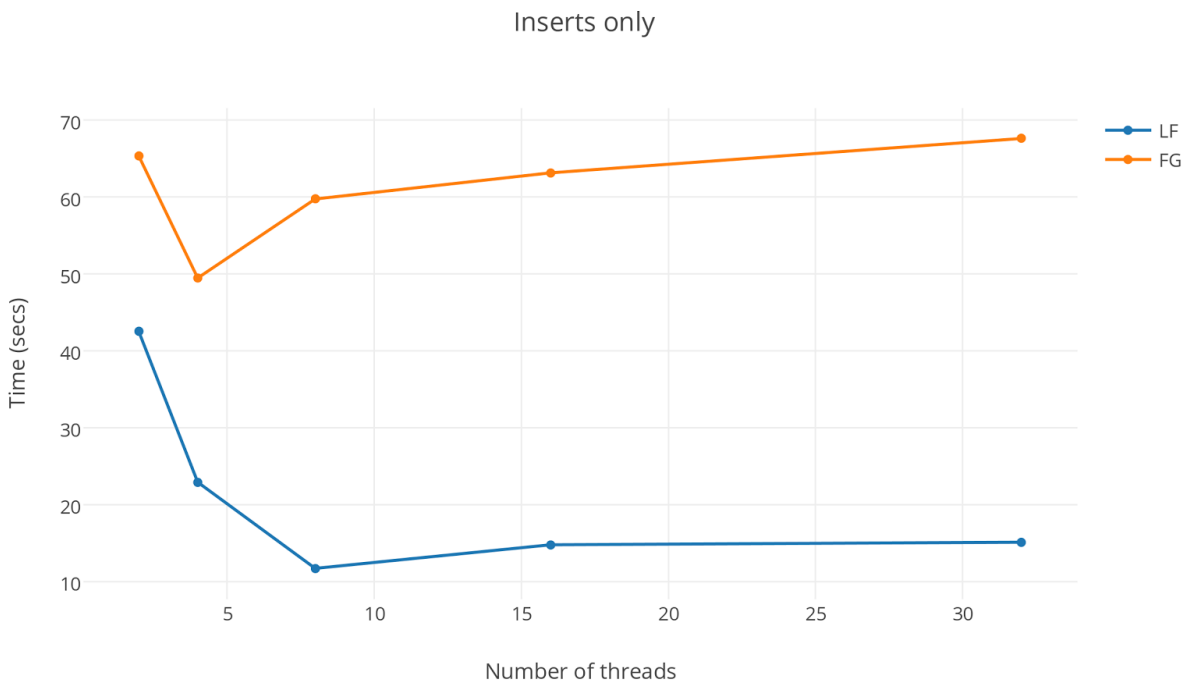


7.5 Workload: Write Intensive, High Contention

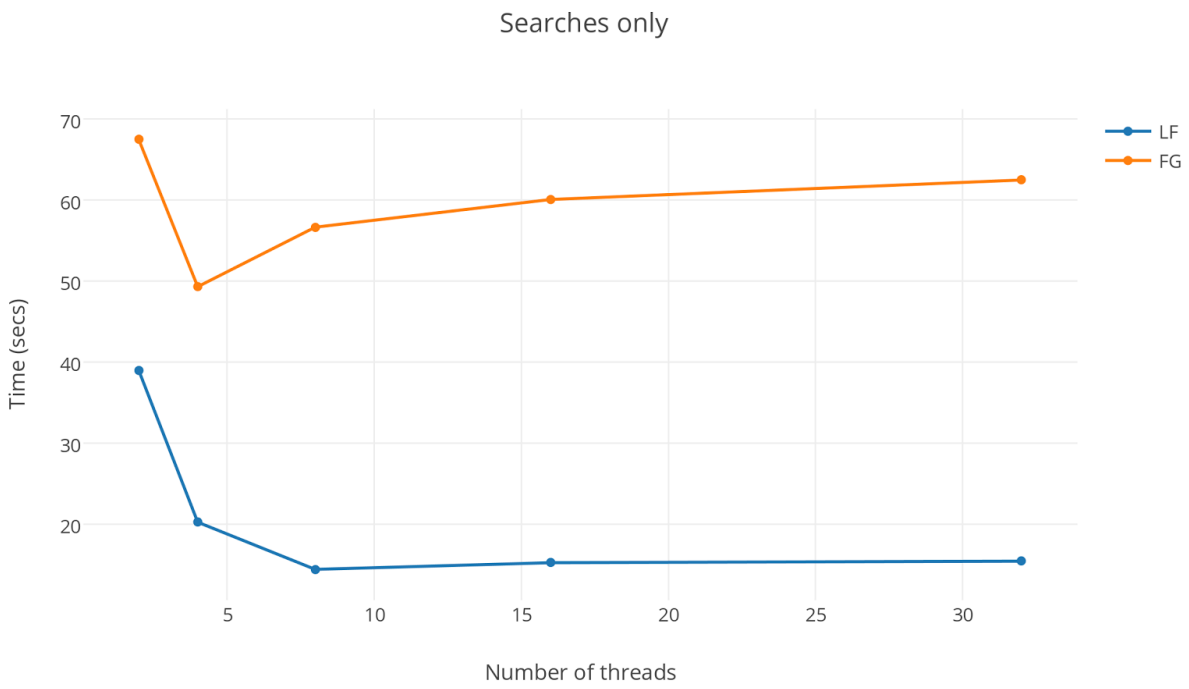


As described above, the high contention workload generates accesses to the same parts of the tree leading to contention for nodes between multiple threads. It can be seen that fine-grained locking performs very poorly in comparison to lock-free access, which is about **1.96x** (on average) faster for the read-intensive workload and about **3.4x** (on average) faster for the write-intensive workload. This is primarily because fine-grained locking is still a blocking algorithm, in that, a thread holding a lock on a node prevents any other node from going past it, restricting parallelism severely. On the other hand, the lock-free version performs much better under a high contention workload. This is made possible because a thread, on detecting an ongoing operation on a node, tries to help complete that operation (rather than waiting for the operation to complete) before retrying its own operation. This way system-wide progress is guaranteed even if some threads might be starved because of the retries.

7.6 Workload: Inserts only, 100000 nodes, plot as a function of number of threads



7.7 Workload: Searches only, 100000 nodes, plot as a function of number of threads



Both the above workloads (inserts only and searches only) were run with a tree of 100000 nodes. The time it took to run the workload was plotted as a function of the number threads. The tests were performed with number of threads = 2, 4, 8, 16 and 32. The U-shaped curve for fine-grained locking shows that the test completion time decreases initially and then increases as the number of threads is increased. This is because as the number of threads increases, the contention in the system increases and the threads spend a lot of time waiting to acquire locks held by some other thread. For the lock-free version the test completion time decreases when the total number of threads are 2, 4 and 8 and then stays about constant for higher thread counts. This is because the GHC machines have 12 hyper-threaded cores and so having a thread count greater than 12 will not provide much significant advantage - any advantage due to increased latency hiding might get overshadowed by the cost of thread context switching.

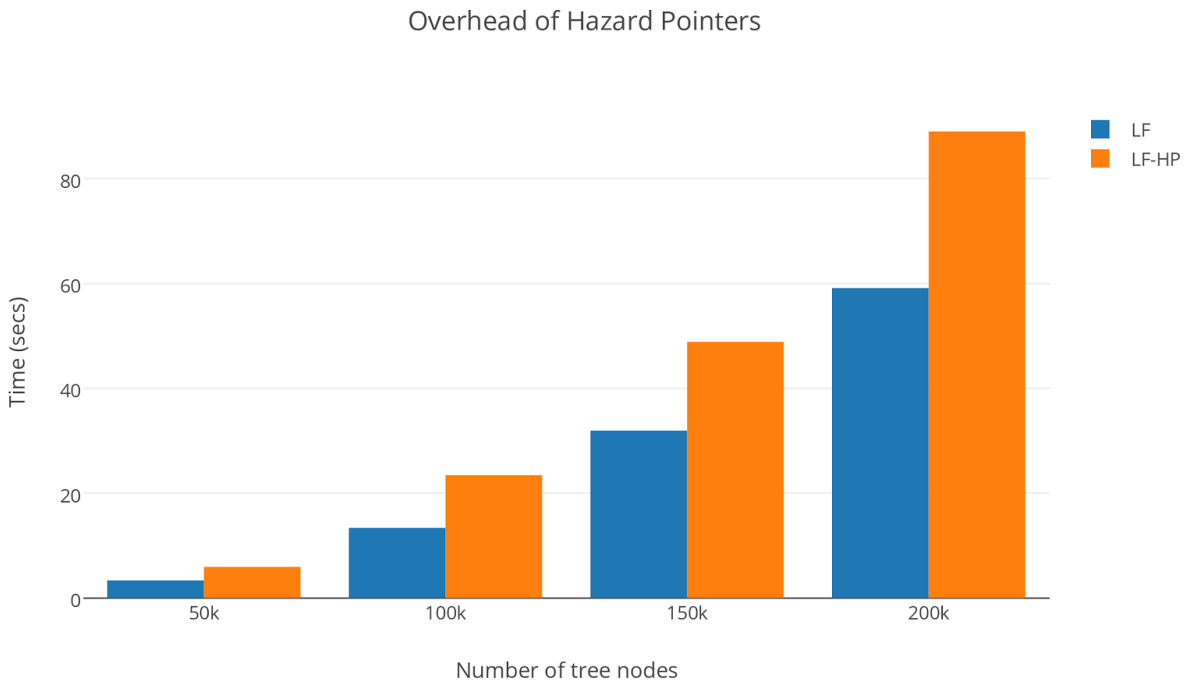
Memory reclamation using hazard pointers technique

Initial Tree Size	Operation	Nodes Freed	% Memory Reclaimed
50000	Delete all 50000 nodes	49915	99.83
100000	Delete all 100000 nodes	99913	99.91
150000	Delete all 150000 nodes	149926	99.95
200000	Delete all 200000 nodes	199908	99.95

In the above test, we started off with a tree of varying sizes and deleted all the nodes in the tree. We kept track of the number of nodes we freed during this process. As can be seen from the table above, using hazard pointers technique we are able to free almost all of the memory allocated by the program.

The threshold size for hazard pointers list was set to 10. The scanning of the retired nodes list is done only when this threshold is exceeded. We believe that some of the nodes were not freed at the end of the run because every thread had a few of them remaining in their rlist and the size of their rlist was below the threshold. The remaining nodes could be cleaned up running a garbage collector at the end of the run.

Overhead of hazard pointers technique



There's no free lunch. The hazard pointers technique introduced overhead in the run time of the lock-free version as can be seen from the graph above. The overhead is about 64% on average, which is to say that the hazard pointer version is **1.64x** slower than the non-hazard pointer version. We believe that the overhead is due to traversing the whole *rlist* checking for a matching address in the hazard pointer lists of all the other threads. Every thread does the scan every time its *rlist* size exceeds the threshold and that's the reason for the increased test completion time when using hazard pointers.

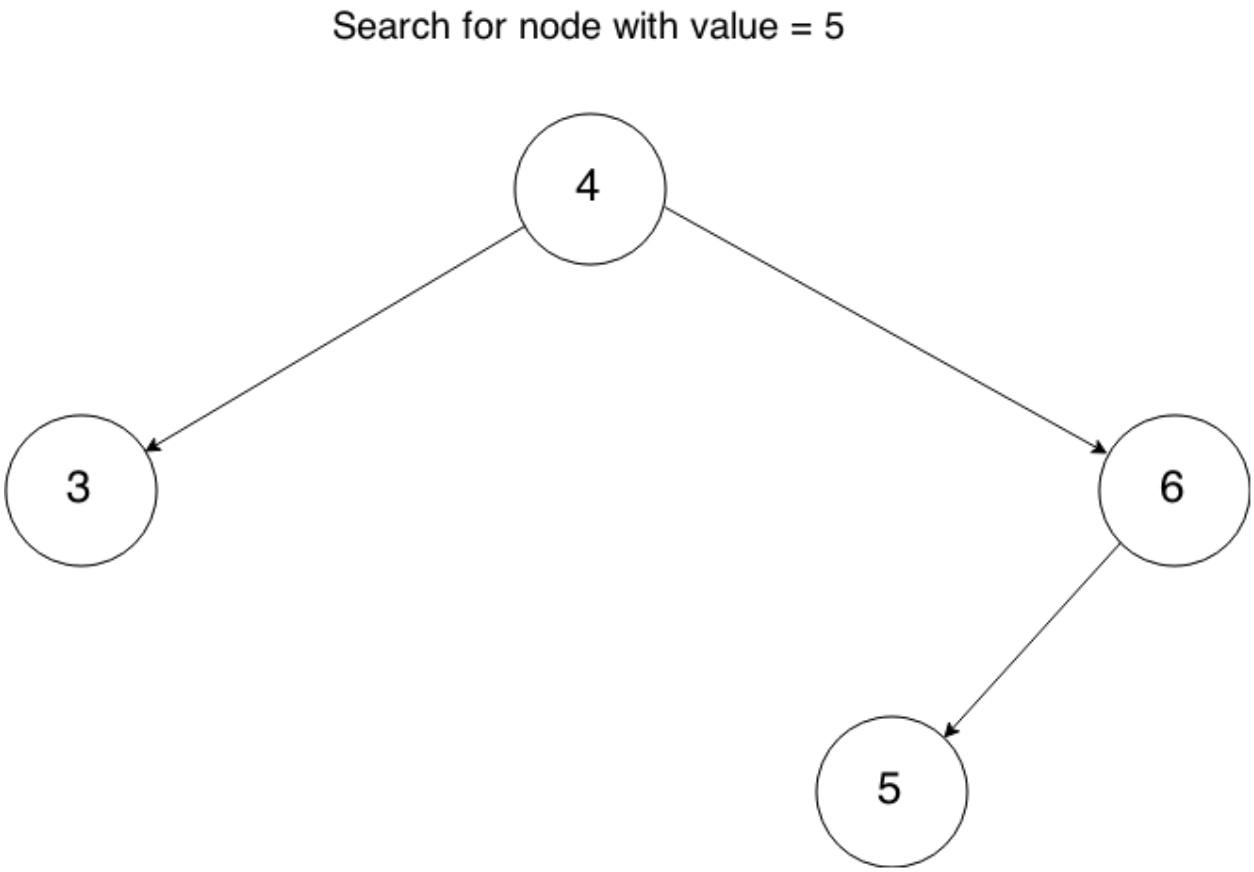
8.0 REFERENCES

[1] Shane V. Howley, Jeremy Jones, *A non-blocking internal binary search tree* (http://dl.acm.org/citation.cfm?id=2312036)

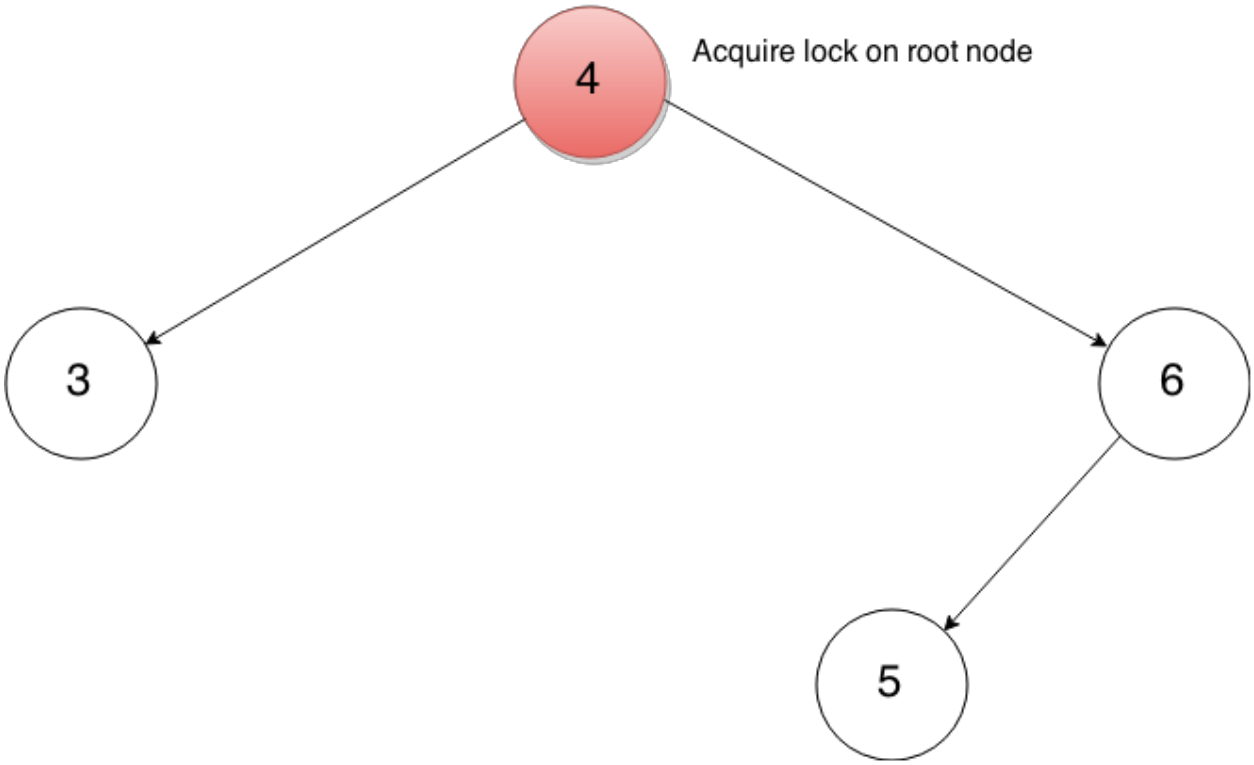
[2] Maged M. Michael, *Hazard Pointers: Safe memory reclamation for lock-free objects* (http://web.cecs.pdx.edu/~walpole/class/cs510/papers/11.pdf)

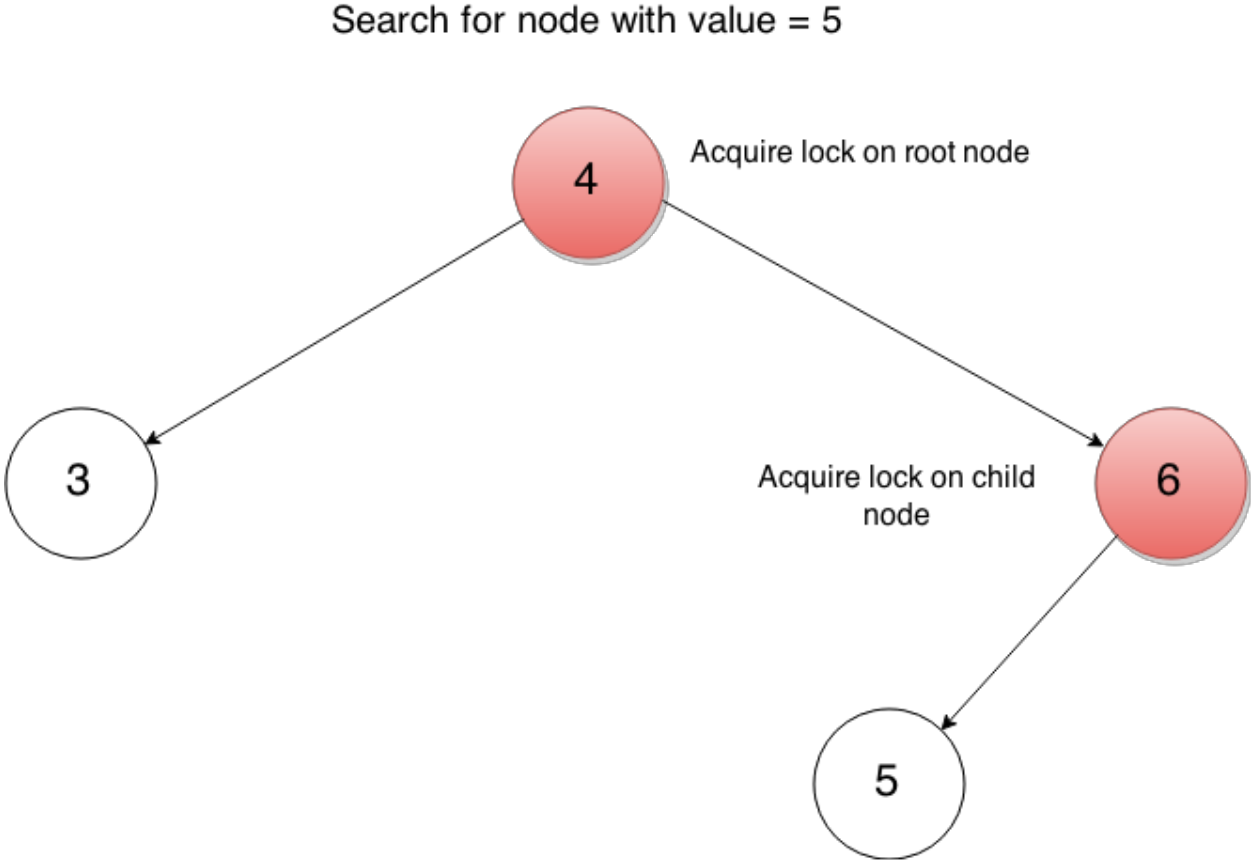
APPENDIX A

Visual depiction of search for FG_BST:

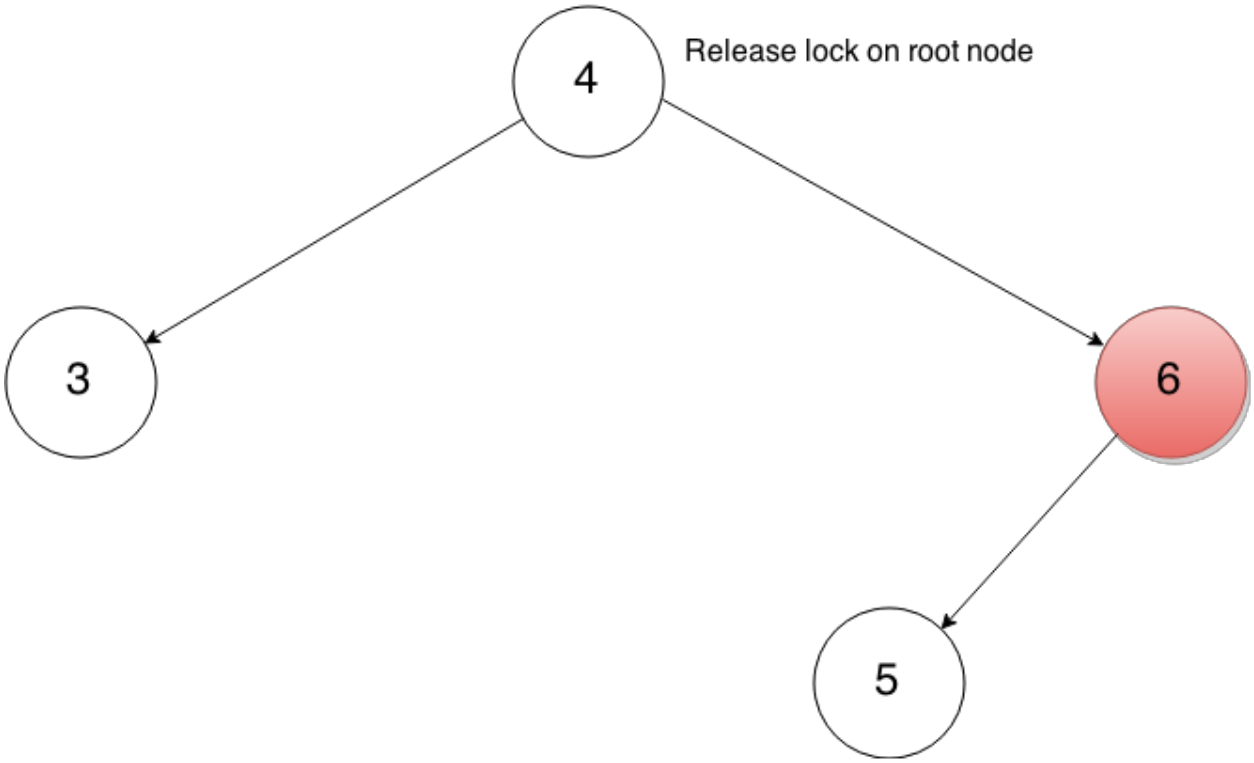


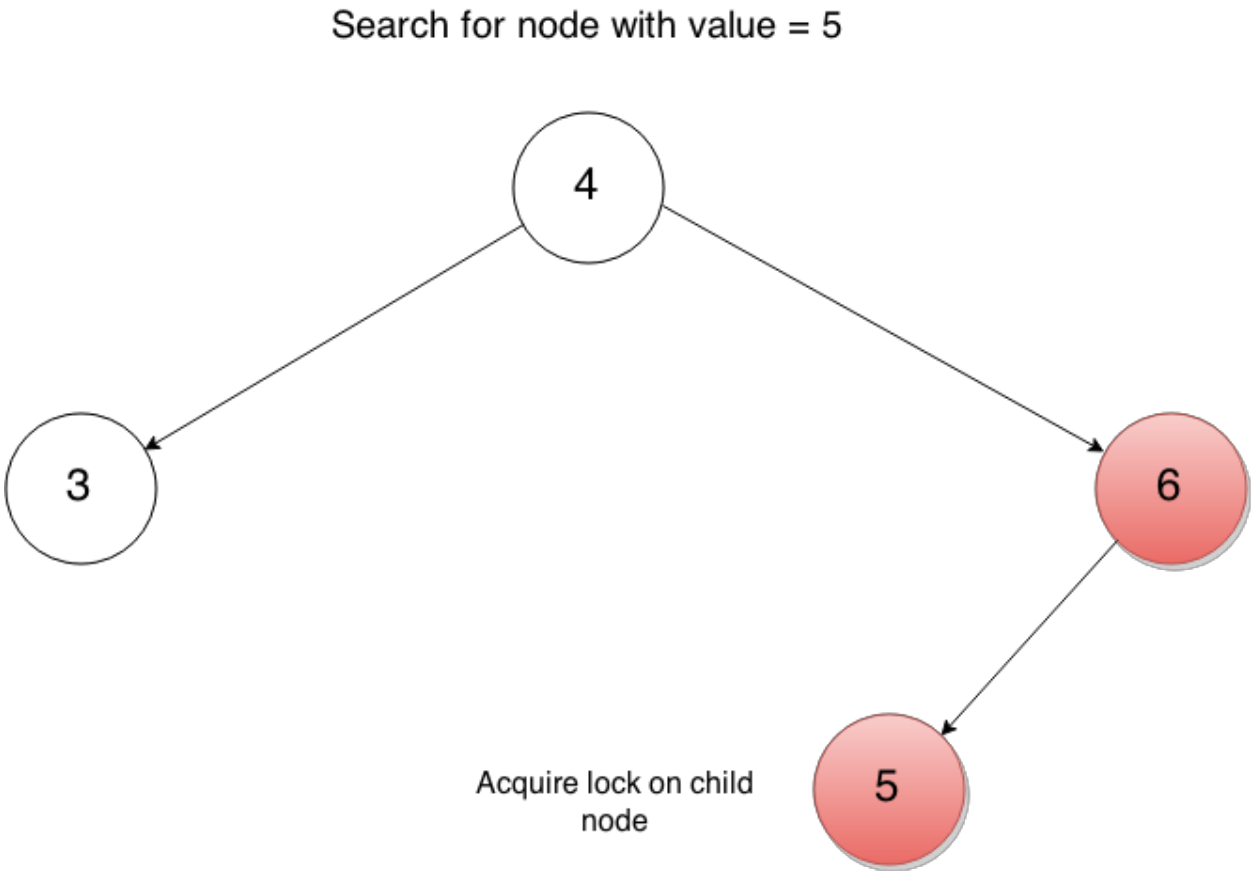
Search for node with value = 5

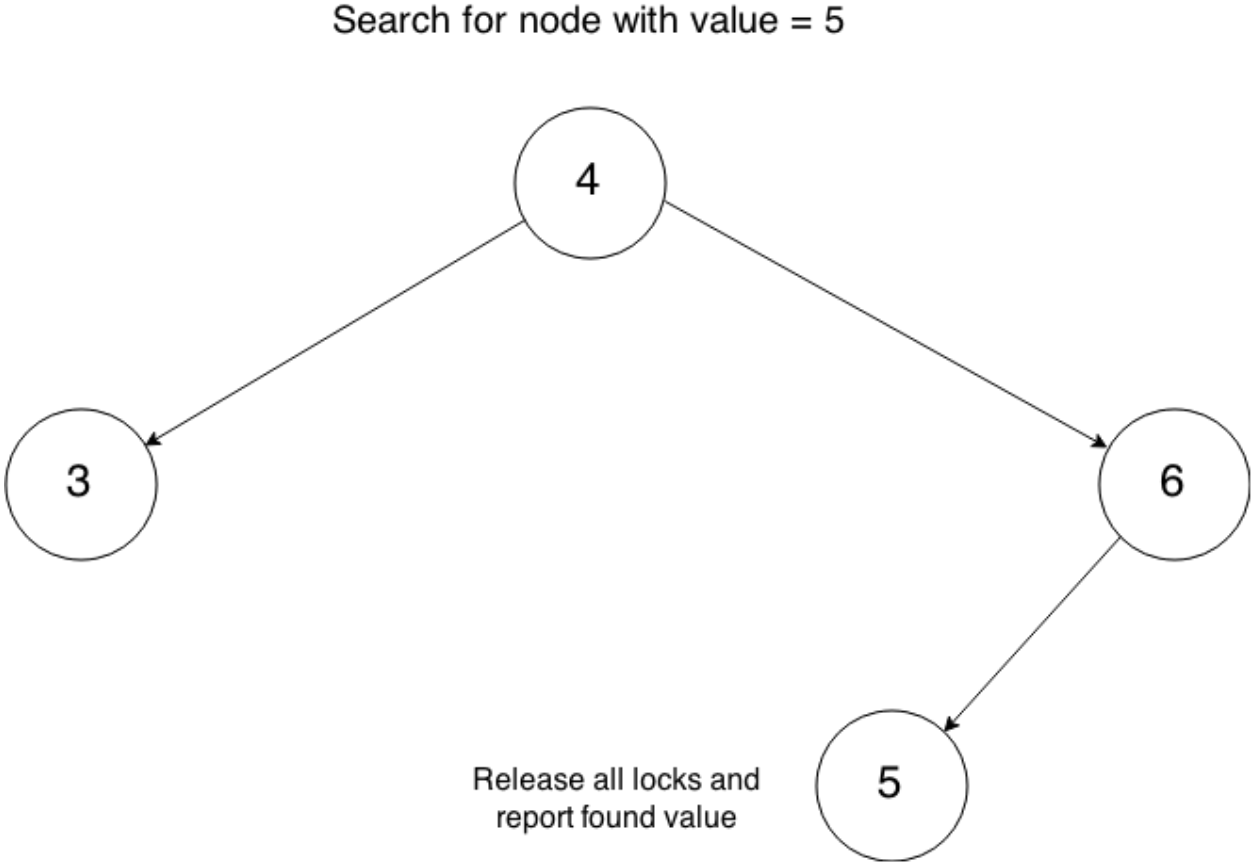



Search for node with value = 5
4
Acquire lock on root node
3
Acquire lock on child node
6
5

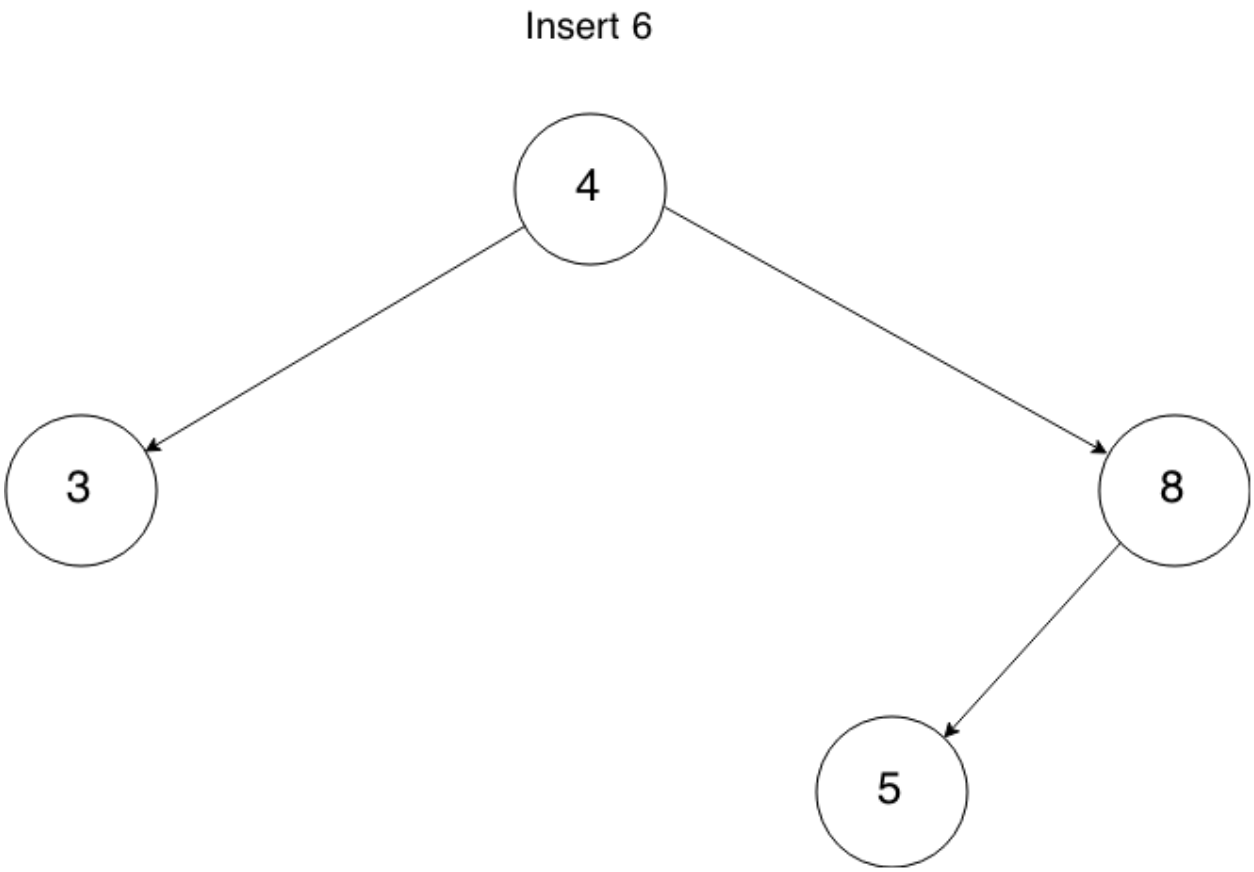
Search for node with value = 5



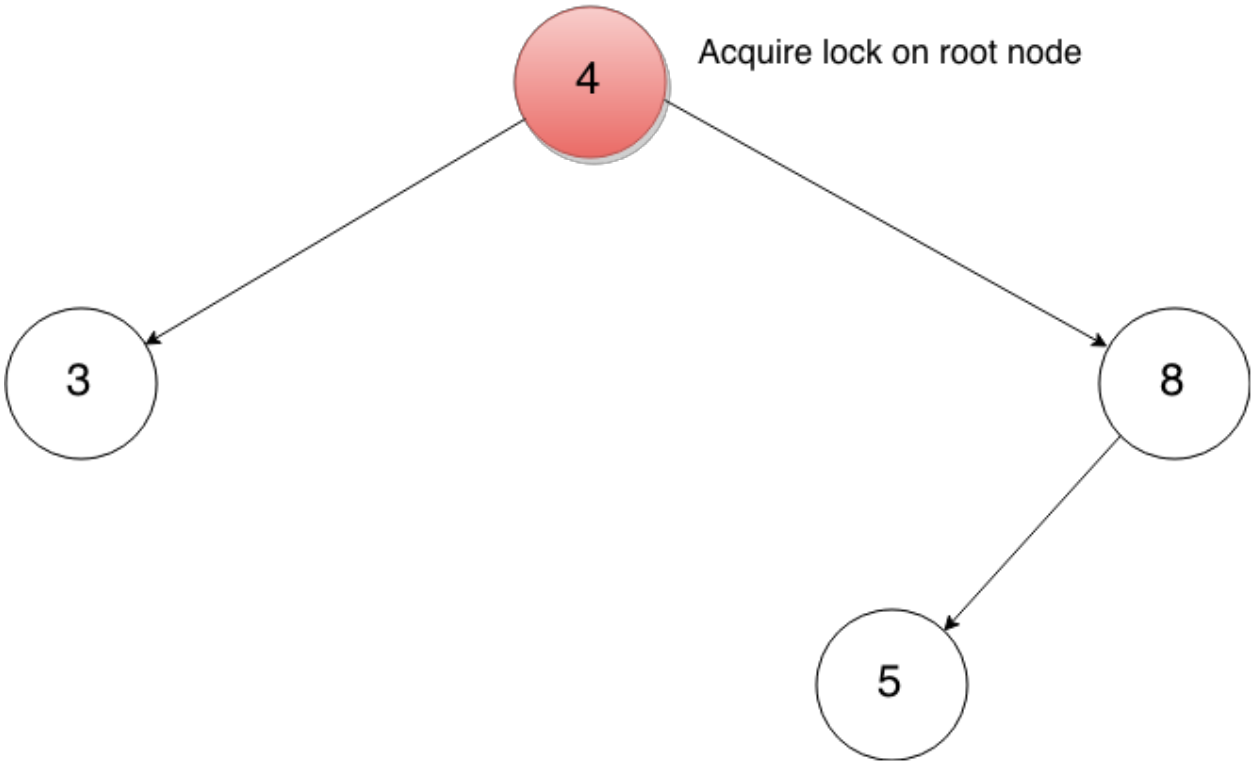

Search for node with value = 5
4
3
6
Acquire lock on child node
5

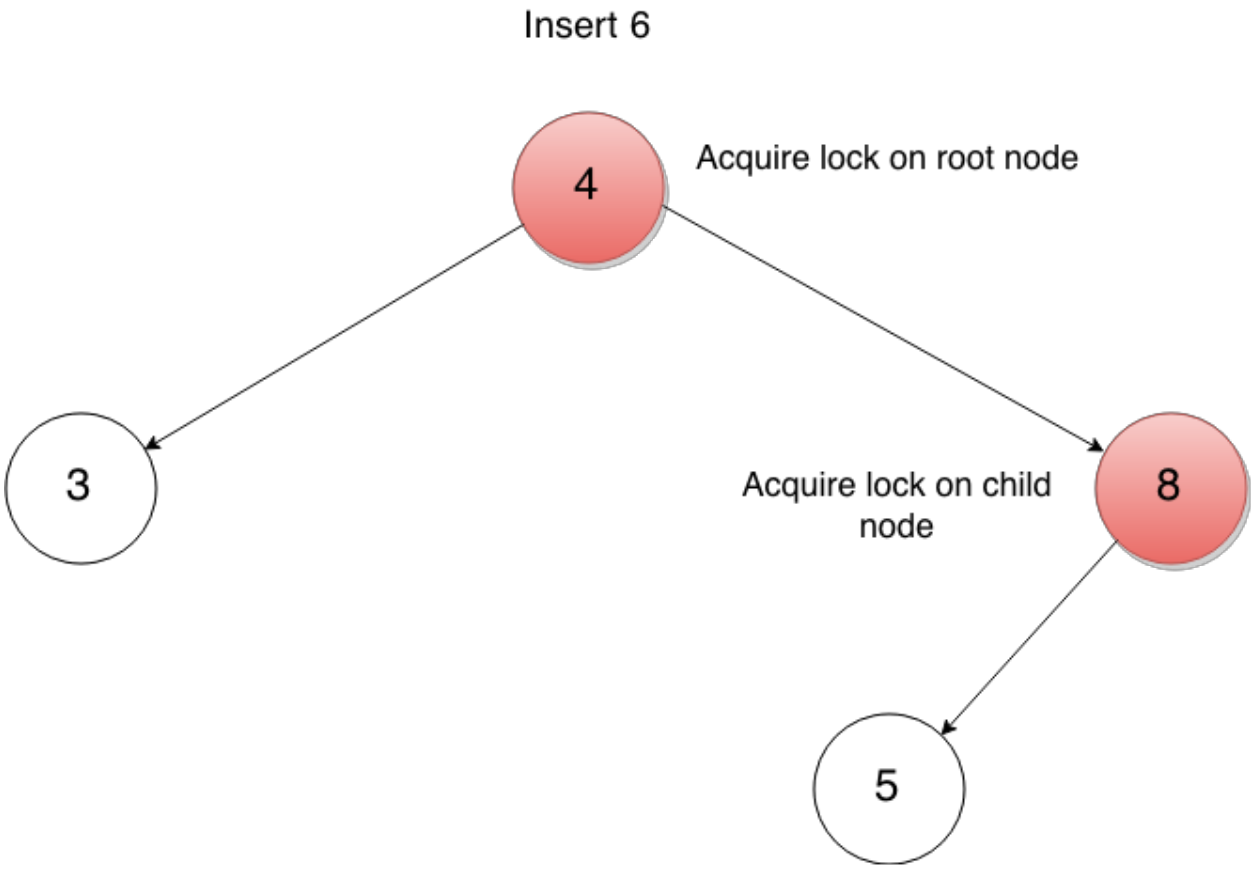

Search for node with value = 5
4
3
6
5
Release all locks and
report found value

Visual depiction of insert for FG_BST:

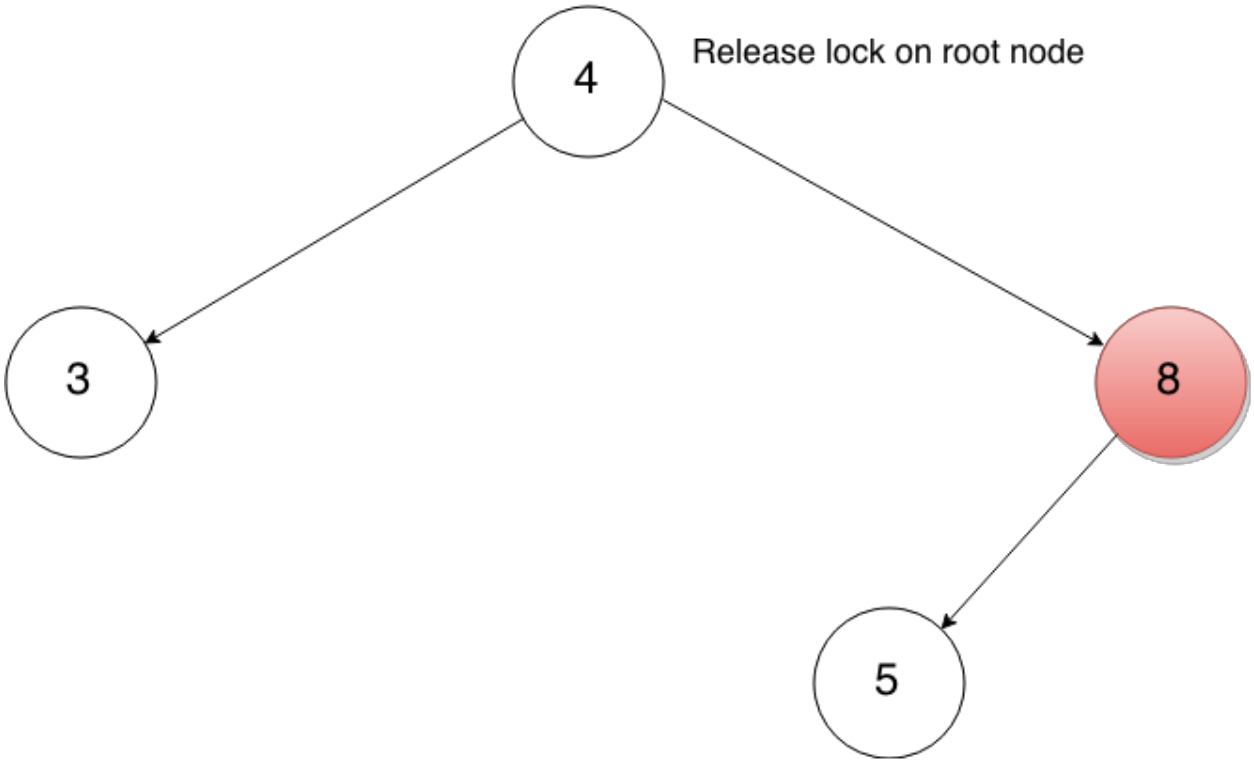


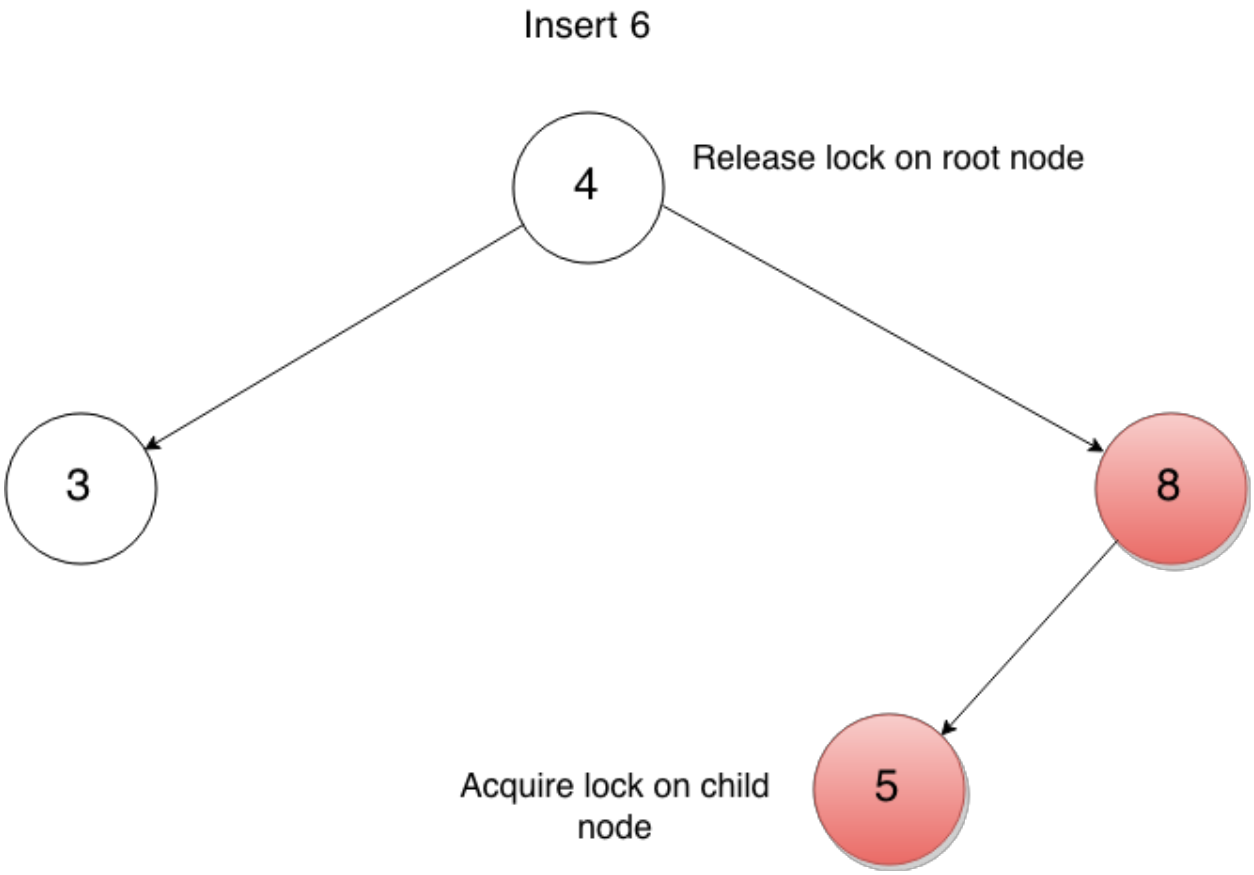
Insert 6

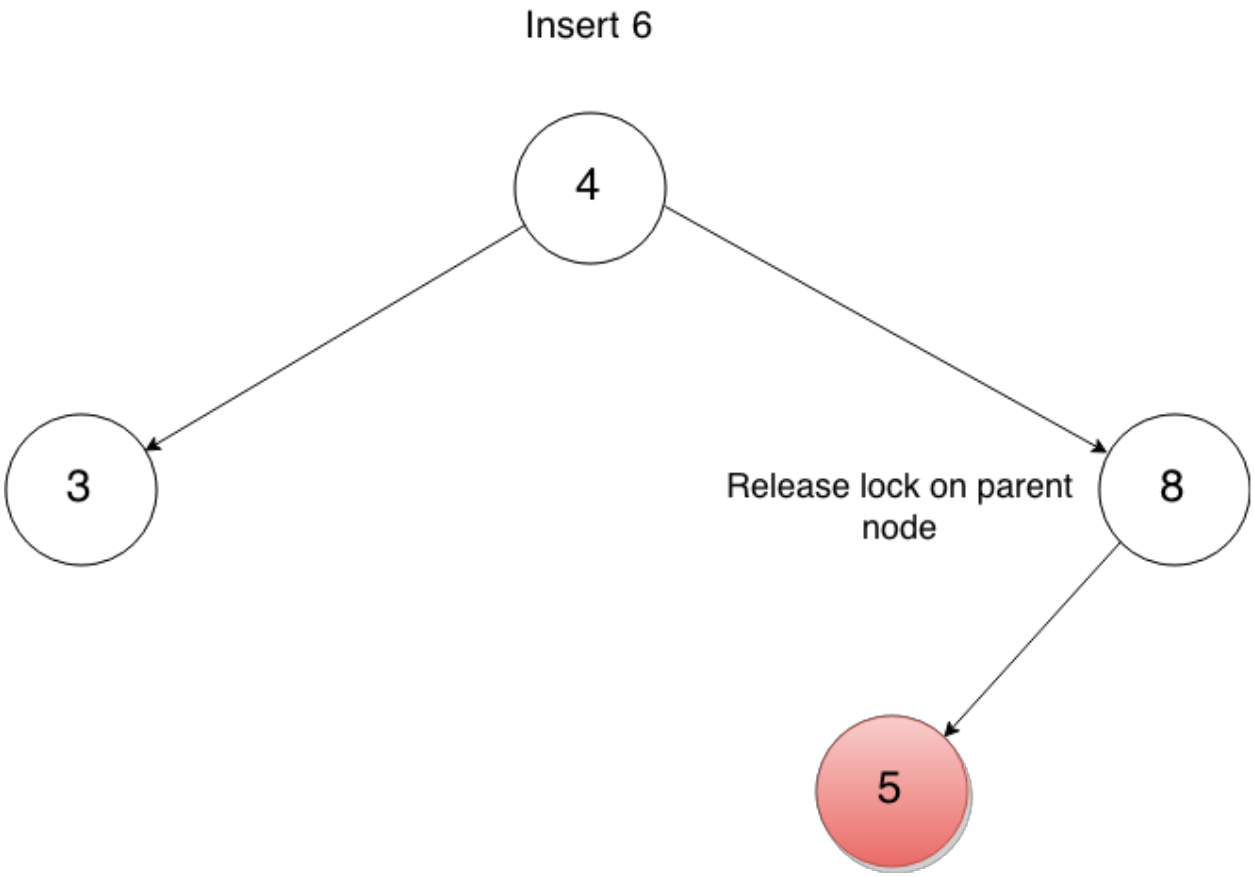


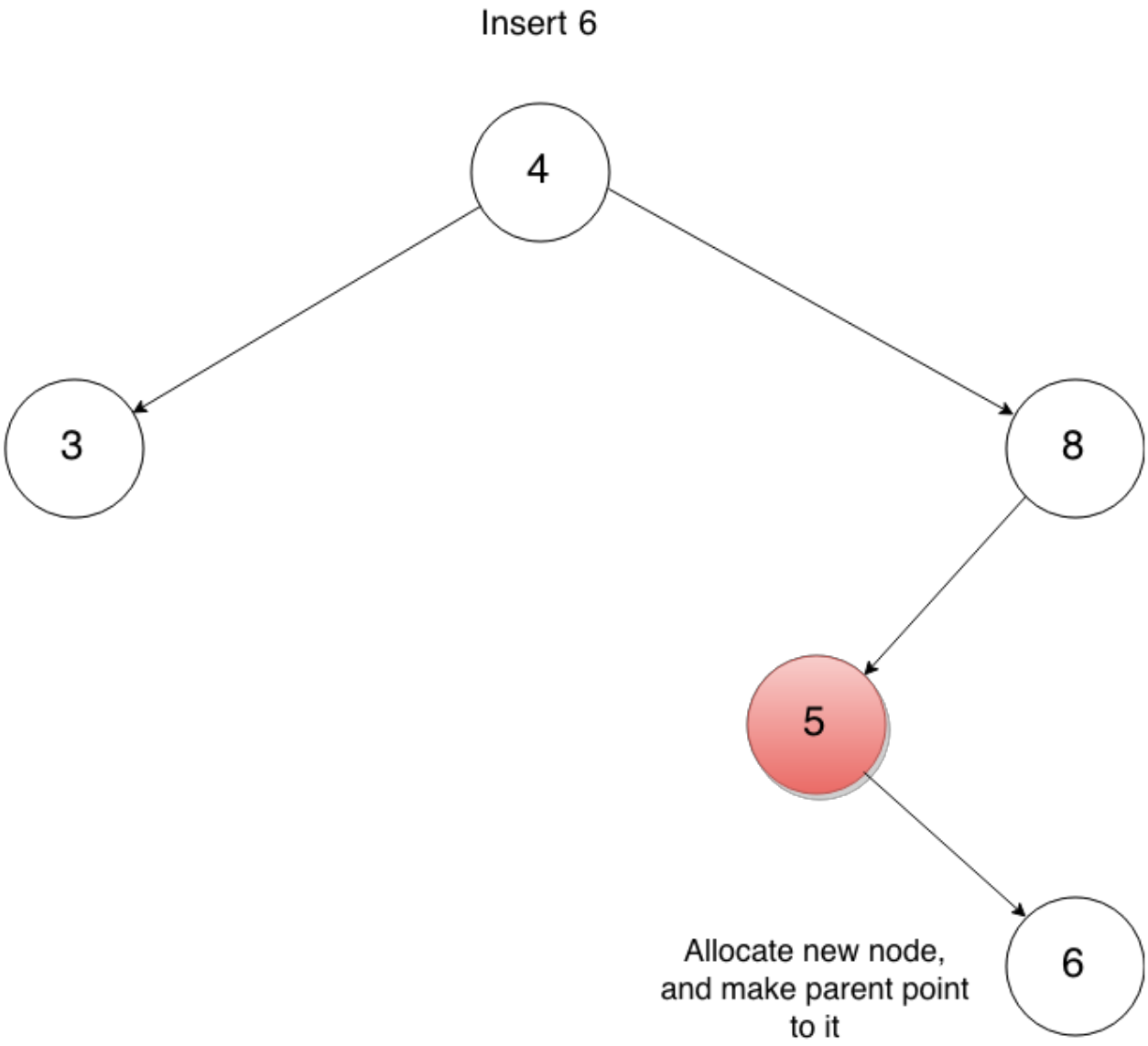

Insert 6
4
Acquire lock on root node
3
Acquire lock on child node
8
5

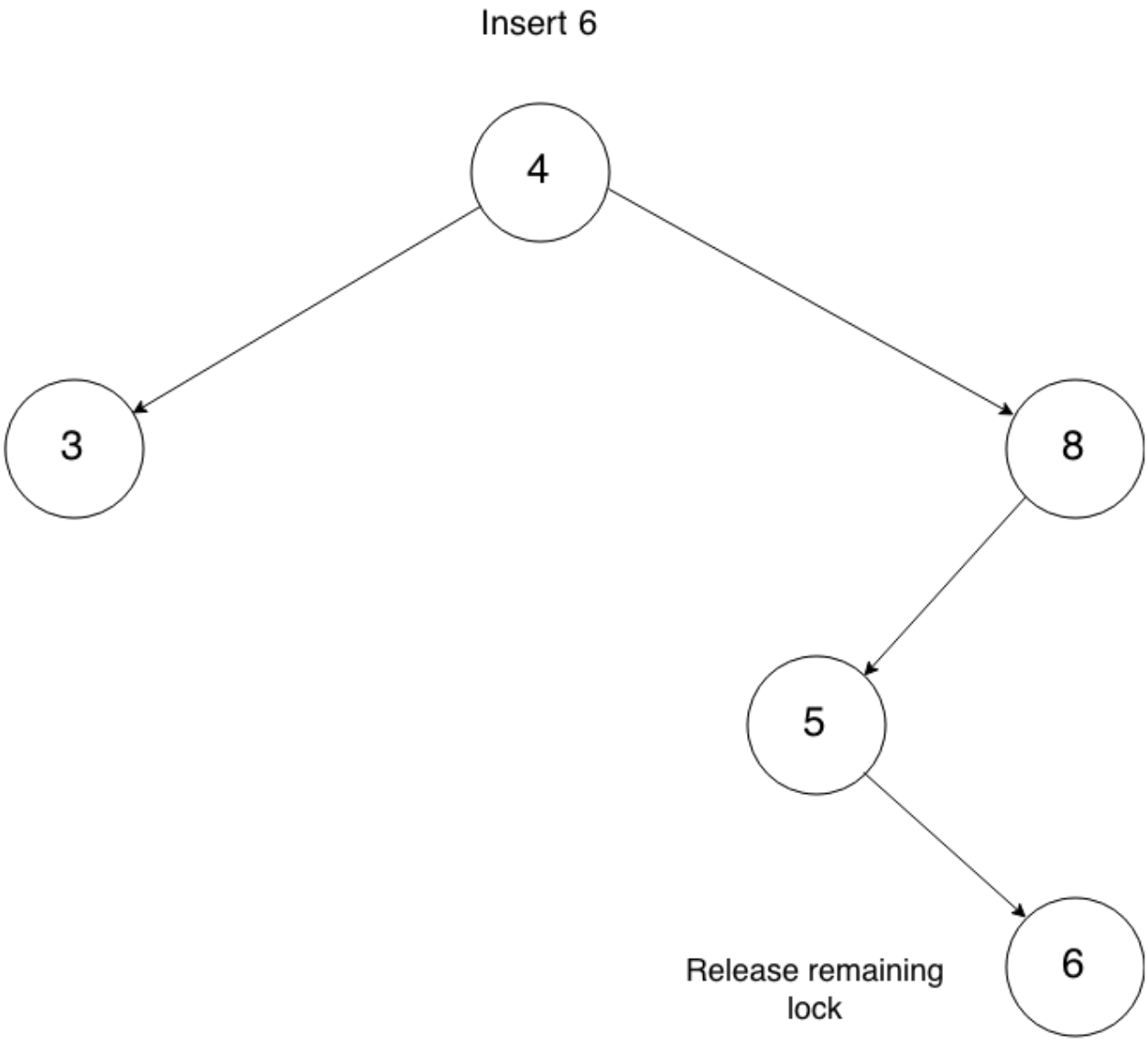
Insert 6



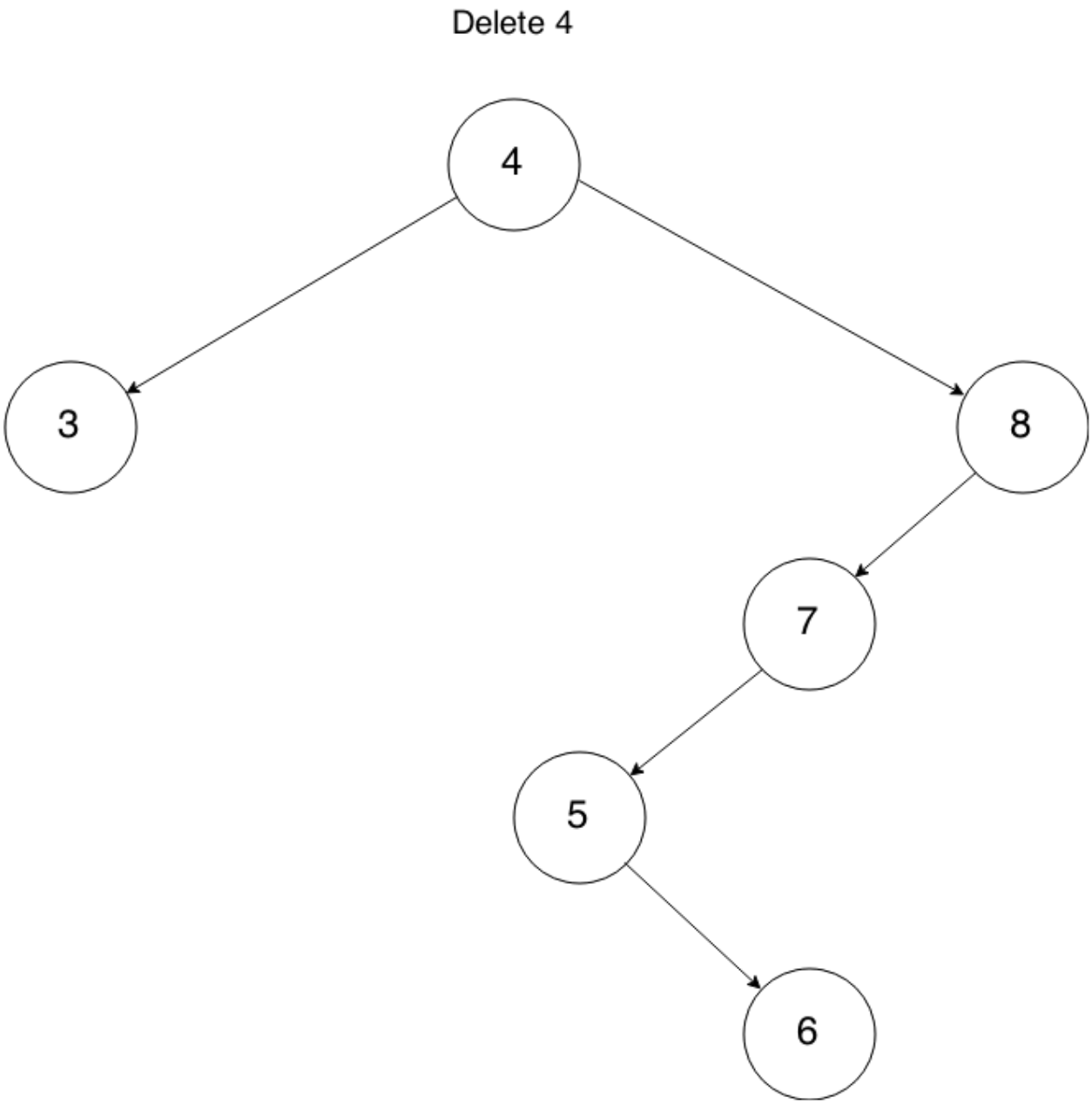

Insert 6
4
Release lock on root node
3
8
Acquire lock on child node
5


Insert 6
4
3
Release lock on parent node
8
5

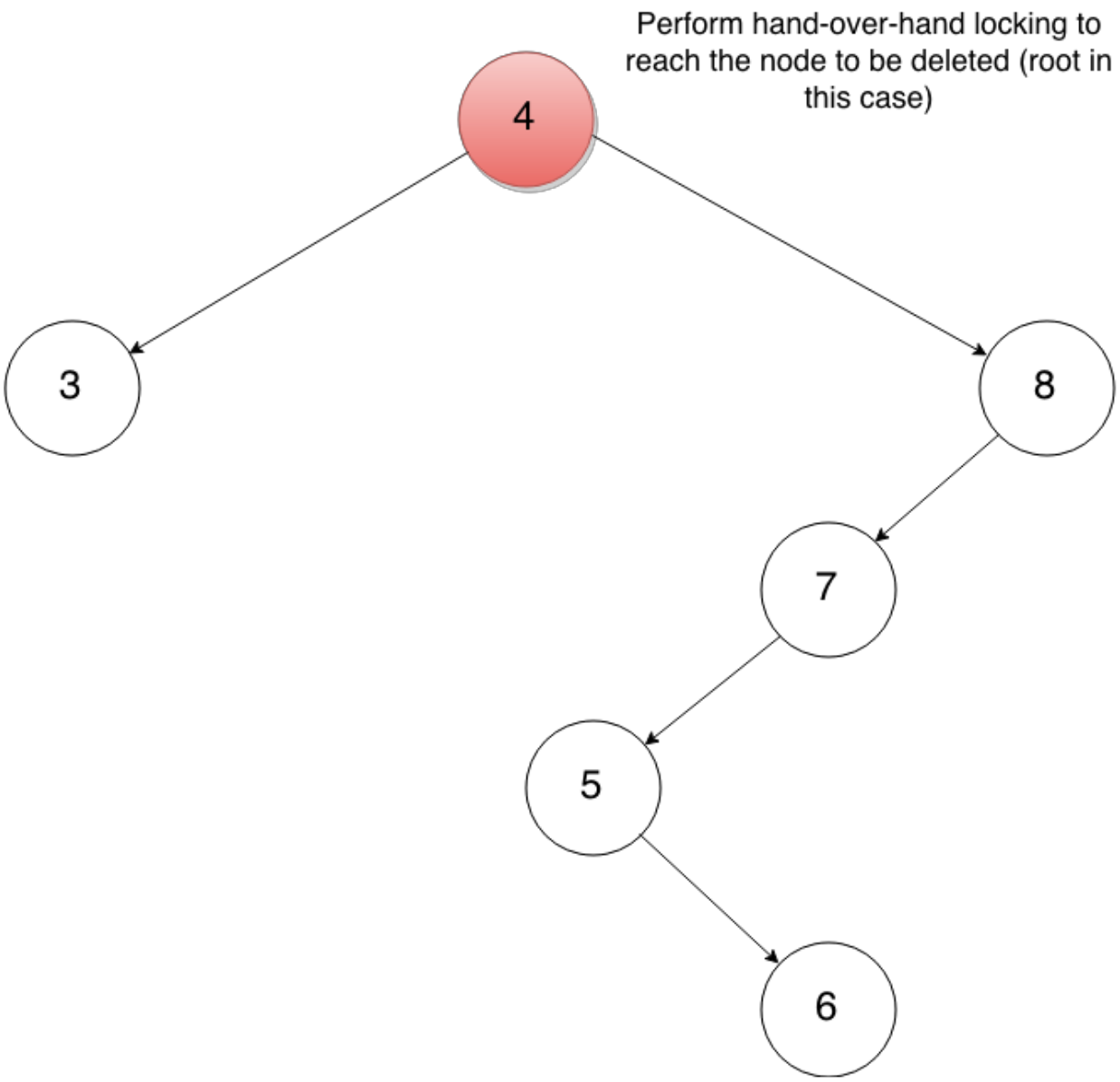

Insert 6
4
3
8
5
6
Allocate new node,
and make parent point
to it


Insert 6
4
3
8
5
6
Release remaining
lock

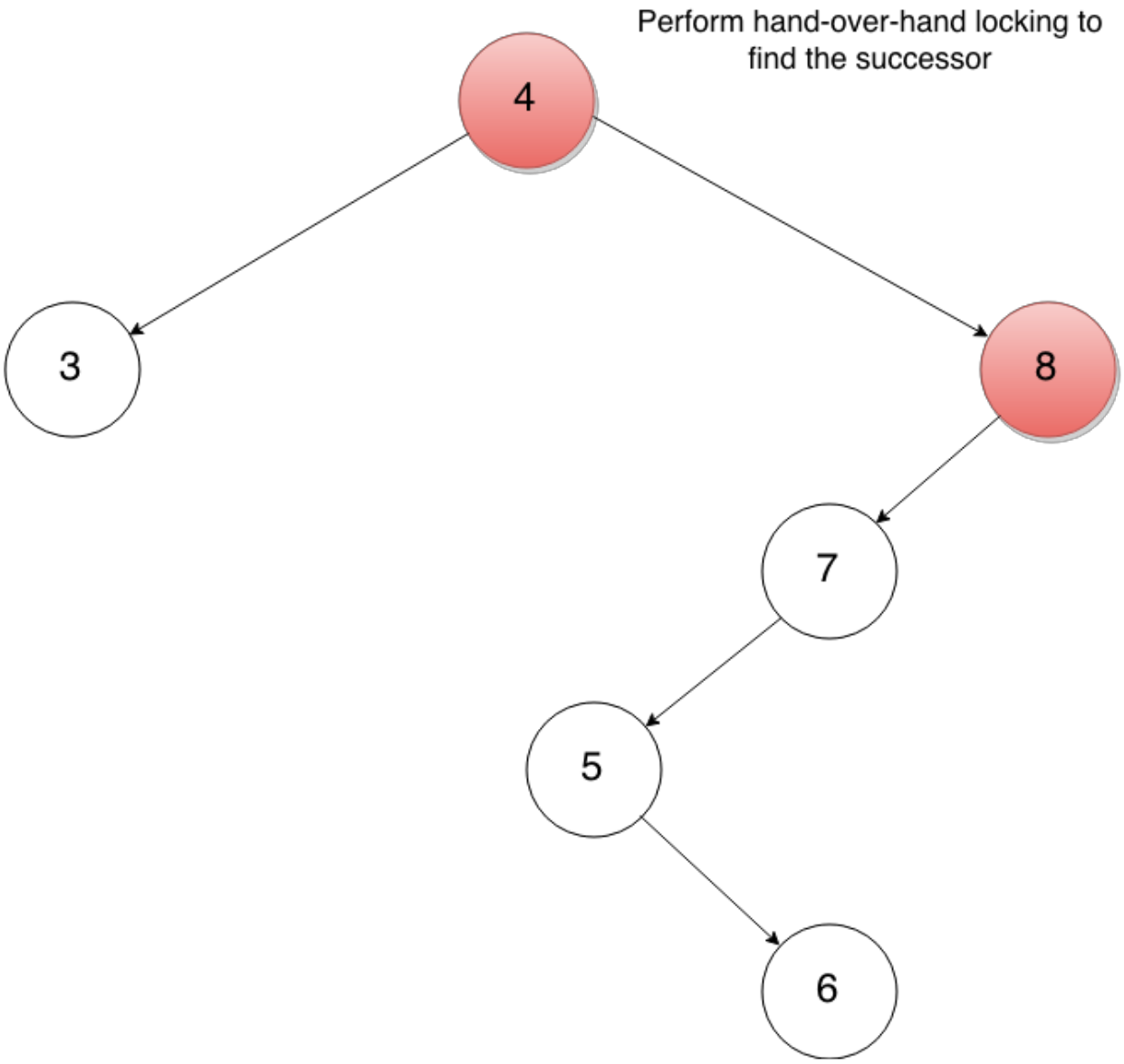
Visual depiction of deletion with a successor replacement:



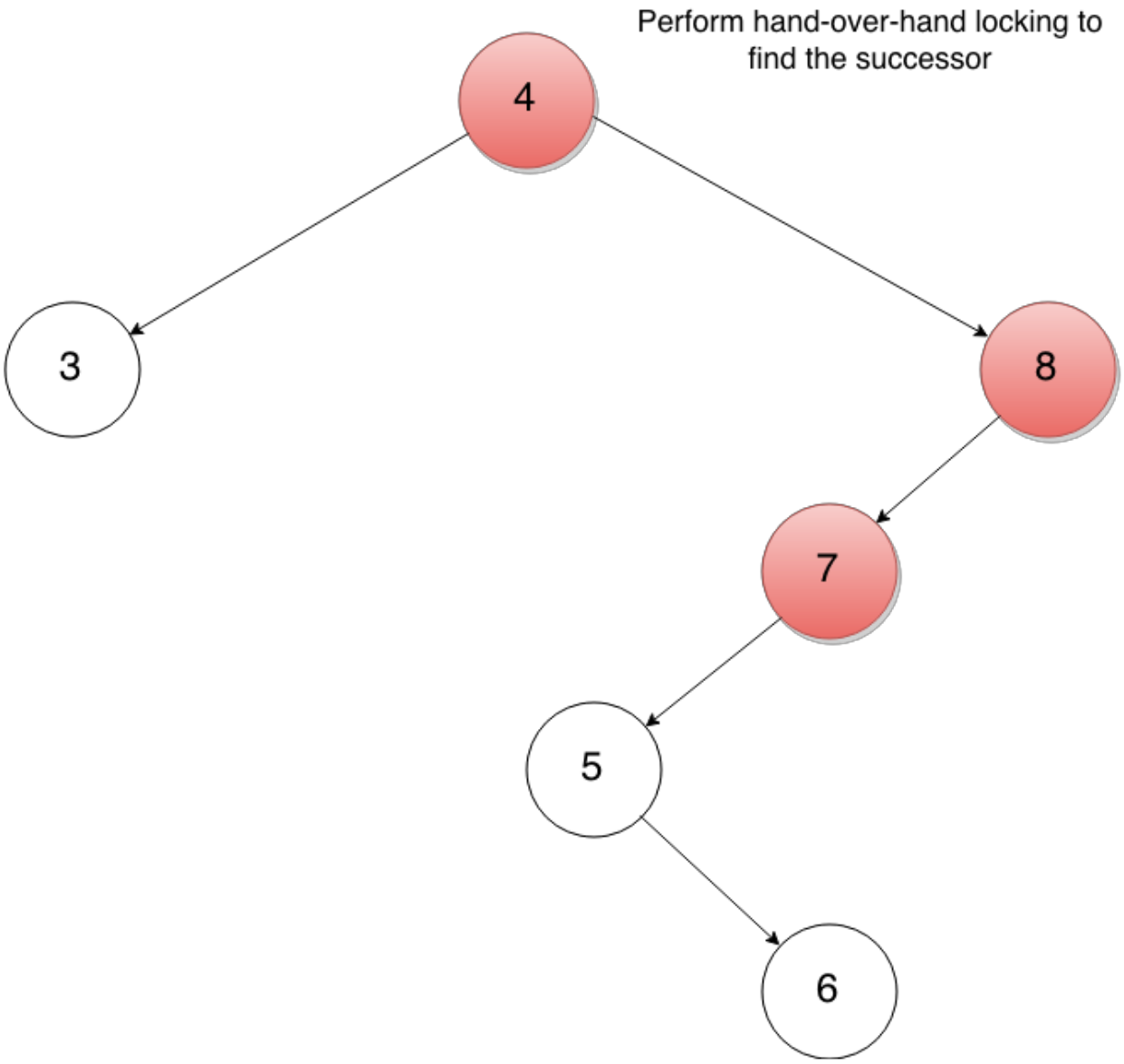
Delete 4

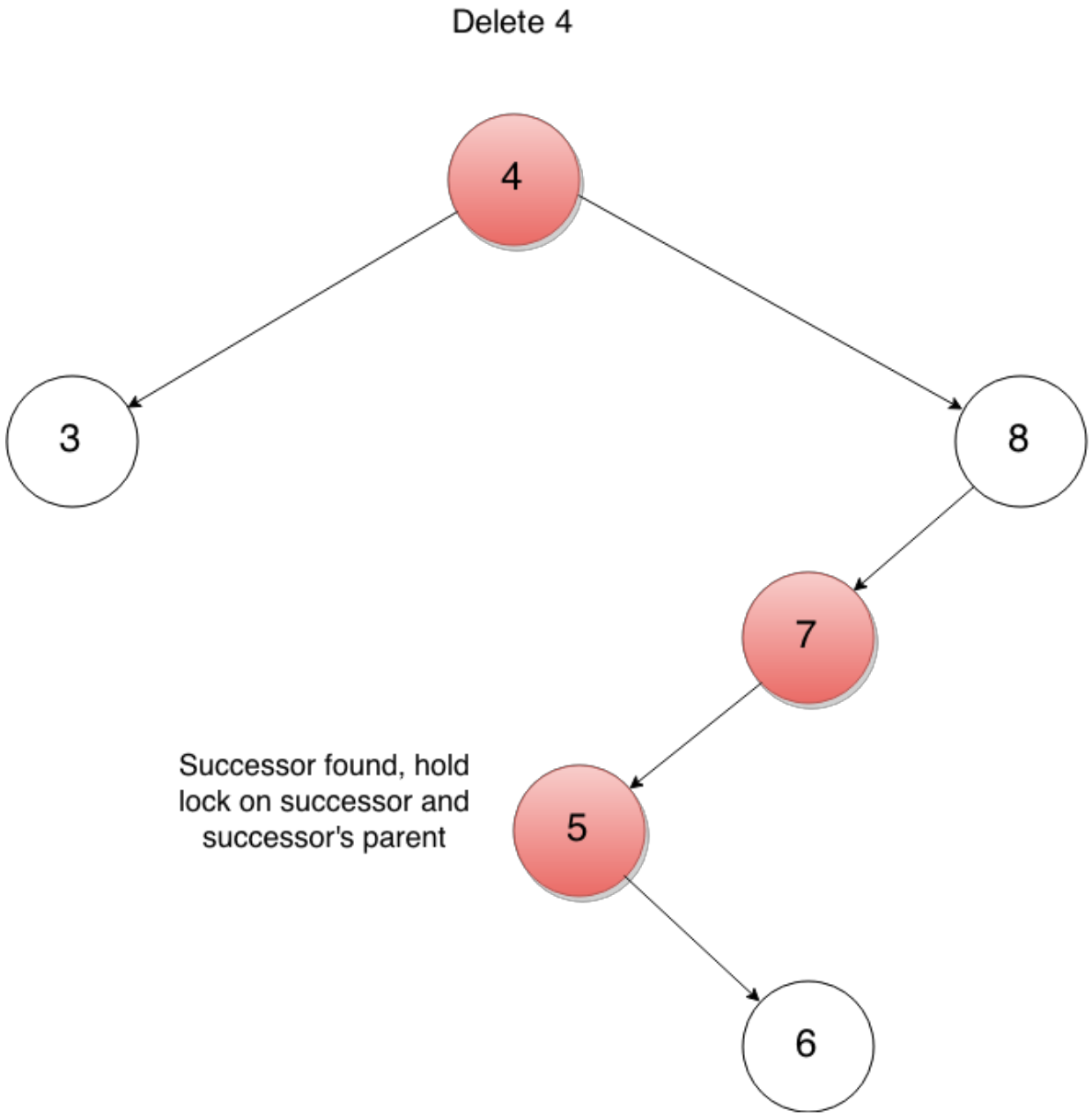

Perform hand-over-hand locking to reach the node to be deleted (root in this case)
4
3
8
7
5
6

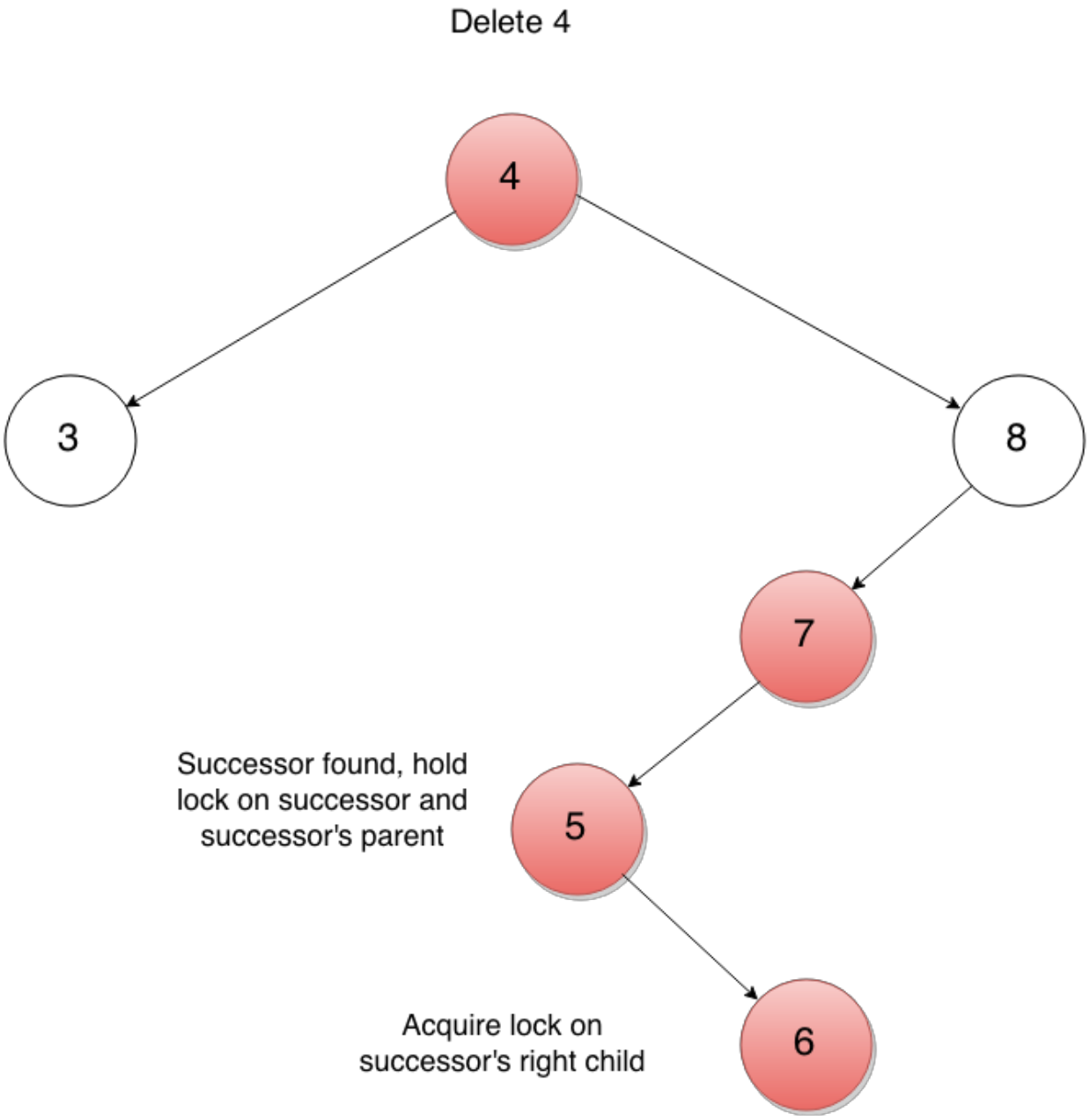
Delete 4


Perform hand-over-hand locking to find the successor
4
3
8
7
5
6

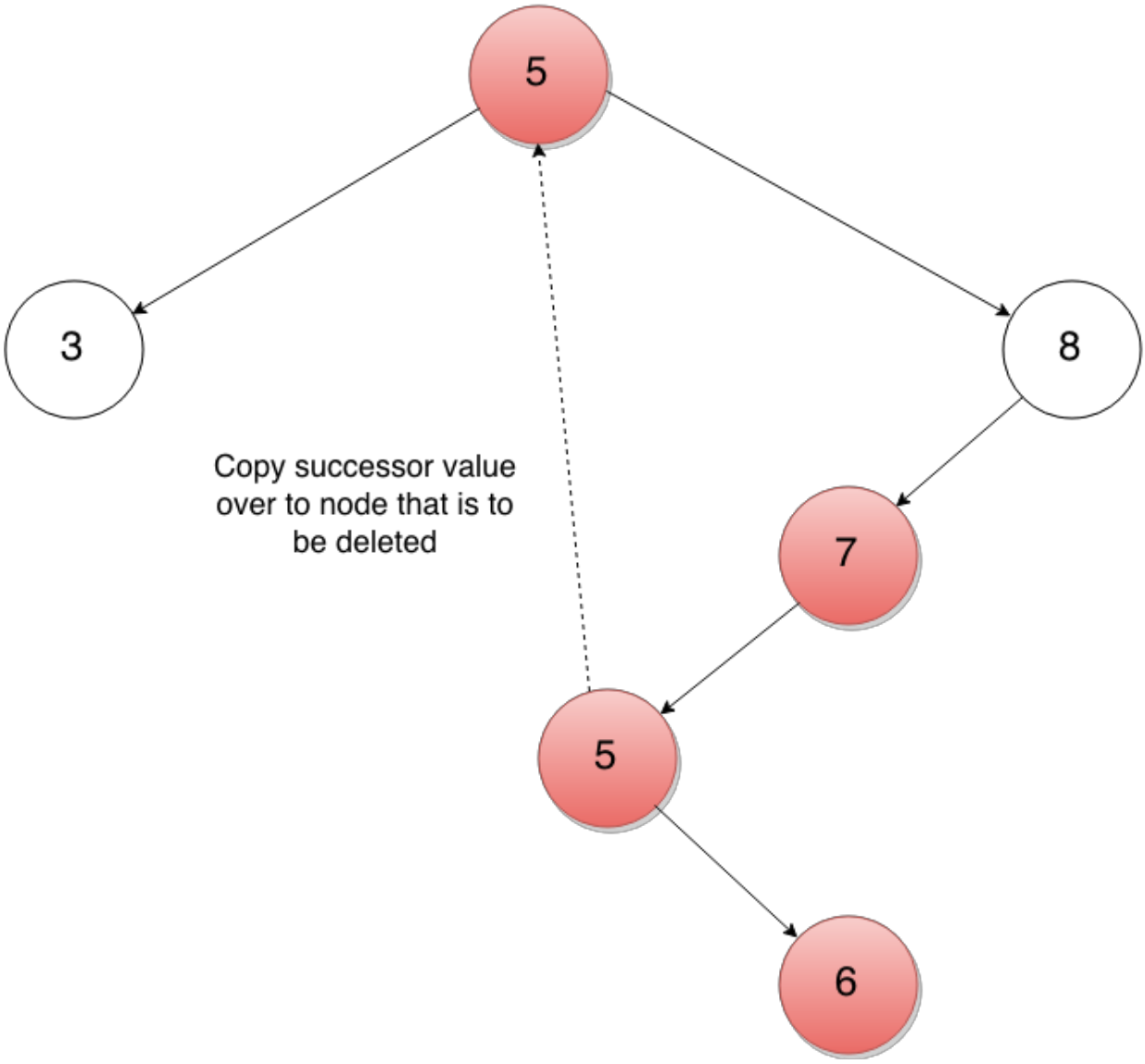
Delete 4

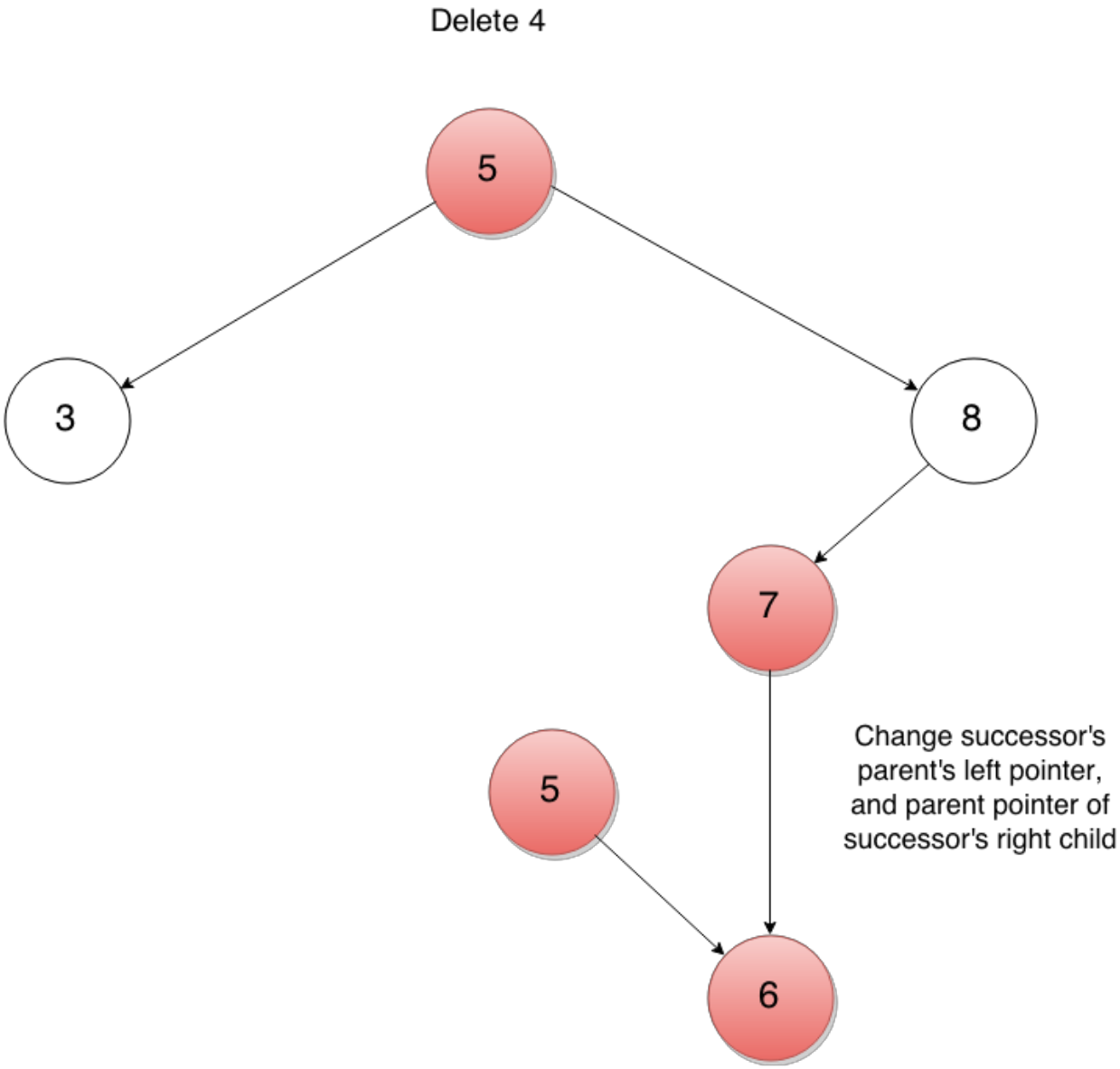

Perform hand-over-hand locking to find the successor
4
3
8
7
5
6


Delete 4
4
3
8
7
Successor found, hold lock on successor and successor's parent
5
6

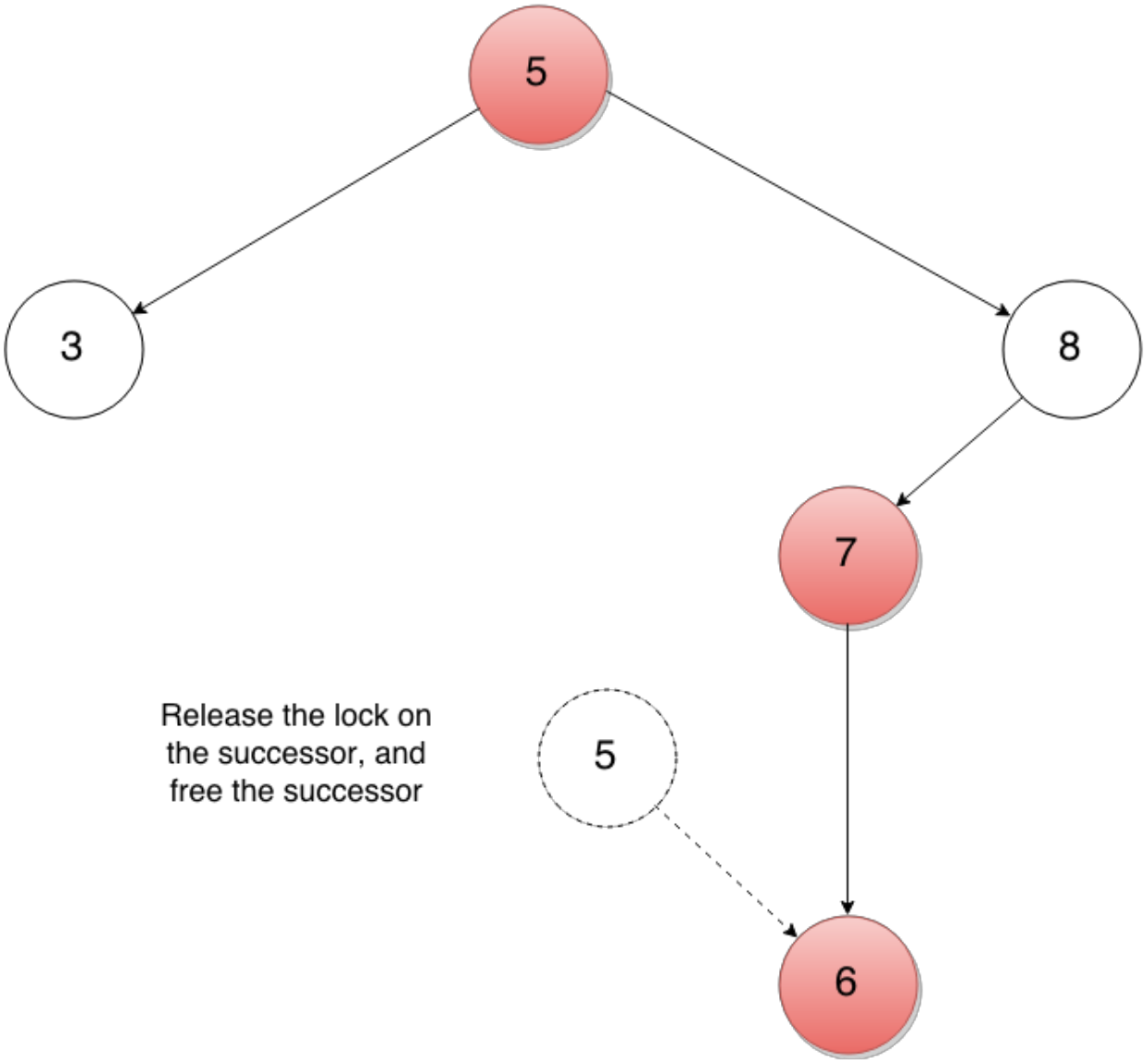

Delete 4
4
3
8
7
Successor found, hold lock on successor and successor's parent
5
Acquire lock on successor's right child
6

Delete 4

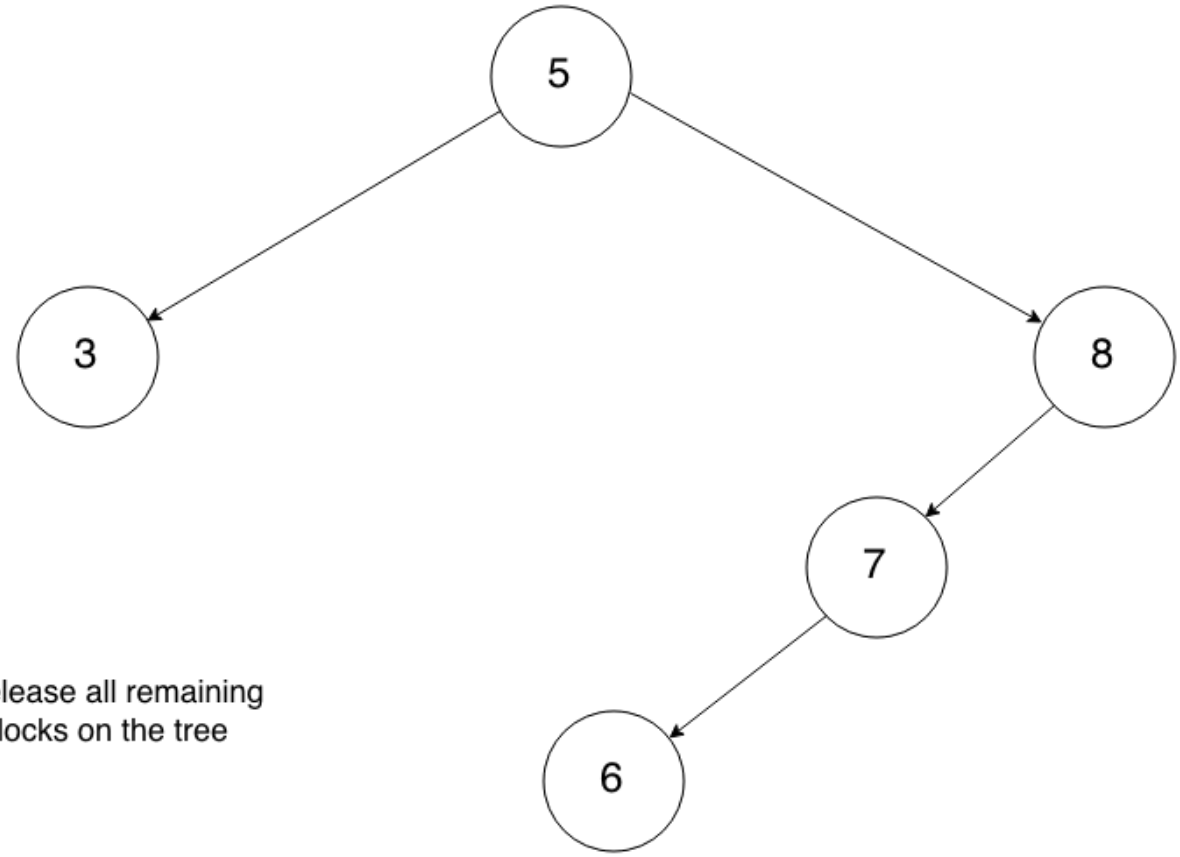

5
3
8
Copy successor value over to node that is to be deleted
7
5
6


Delete 4
5
3
8
7
5
Change successor's parent's left pointer, and parent pointer of successor's right child
6

Delete 4


5
3
8
7
Release the lock on the successor, and free the successor
5
6

Delete 4



Release all remaining
locks on the tree