Wait-free Trees with Asymptotically-Efficient Range Queries

Ilya Kokorin ITMO University, VK Dan Alistarh IST Austria Vitaly Aksenov City, University of London

Abstract—Tree data structures, such as red-black trees, quad trees, treaps, or tries, are fundamental tools in computer science. A classical problem in concurrency is to obtain expressive, efficient, and scalable versions of practical tree data structures. We are interested in concurrent trees supporting range queries, i.e., queries that involve multiple consecutive data items. Existing implementations with this capability can list keys in a specific range, but do not support aggregate range queries: for instance, if we want to calculate the number of keys in a range, the only choice is to retrieve a whole list and return its size. This is suboptimal: in the sequential setting, one can augment a balanced search tree with counters and, consequently, perform these aggregate requests in logarithmic rather than linear time.

In this paper, we propose a generic approach to implement a broad class of range queries on concurrent trees in a way that is wait-free, asymptotically efficient, and practically scalable. The key idea is a new mechanism for maintaining metadata concurrently at tree nodes, which can be seen as a wait-free variant of hand-over-hand locking (which we call hand-over-hand helping). We implement, test, and benchmark a balanced binary search tree with wait-free insert, delete, contains, and count operations, returning the number of keys in a given range which validates the expected speedups because of our method in practice.

Index Terms—Data structures, Concurrent programming, Range queries

I. INTRODUCTION

Tree data structures are ubiquitous in computer science, due to their high expressive power and practical versatility. For instance, in databases, index trees allow searching for an indexed key faster than traversing through all the elements. Typically, such index is implemented as B-tree [10], [15], [18], although alternate implementations are possible, such as the red-black tree [19], or the splay tree [29]. Moreover, one could use quad trees [16] to store and retrieve a collection of points in a plane, or tries [11] for fast prefix matching in strings.

In this paper, we are interested in *concurrent* implementations of fundamental tree data structures that combine theoretical and practical efficiency, with *expressivity* in terms of the class of queries they support efficiently. Specifically, we are interested in trees supporting the following types of operations. We call a query, retrieving or modifying a single data item, a *scalar query*; and a query, involving multiple consecutive (by value) data items, a *range query*. For example, a search tree can provide the following scalar queries:

• insert (key) — if key does not exist in the tree, inserts it to the tree, otherwise, leaves the tree unmodified;

- remove (key) if key exists in the tree, removes it from the tree, otherwise, leaves the tree unmodified;
- contains (key) returns true if the tree contains key, false, otherwise.

Also, a search tree can provide the following range queries:

- collect (min, max) returns all the keys from the [min; max] interval from the set;
- count (min, max) returns the number of keys from the [min; max] interval from the set.

In addition, we would like to support aggregate range queries: for example, in a search tree storing key-value pairs, the range query range_add(min, max, delta) adds delta to all the values corresponding to the keys in a given range [min, max], whereas the range query range_sum(min, max) calculates the sum of all values corresponding to the keys in a given range [min, max].

In this work, in addition to extensively investigated collect query (see e.g., [8], [12]) we require the index to perform aggregate range queries (e.g., count) in an asymptotically optimal way. For example, we can use such aggregate range queries to find the number of requests to the system in the specified time range from the specified users.

Currently, all existing concurrent trees answer the aggregate range queries in time proportional to the number of elements in the range, i.e., for a count query it works as count(min, max) = collect(min, max).length(). This is clearly suboptimal: in the sequential setting, augmented search trees can perform such queries in O(height) (where height is the height of the tree) which can be exponentially faster for balanced trees.

Now, we overview how to sequentially perform count query in O(height) time for a binary search tree. Note that other aggregate range queries can be implemented similarly. For each node, we store the number of keys in its subtree. Then, we start traversing the tree from the root downwards. When we are in the node v we check three cases. If the range of keys in the subtree of v lies inside the required range — we add the stored size of the subtree of v to the answer. If it intersects with the required range — we go recursively to children, returning the sum of results for v.left and v.right. And, finally, if it does not intersect with the required range — we stop the call and return zero. (We unroll this recursion in our sequential and concurrent implementations we unroll the recursion.)

In this paper, we present a scalable approach that can make any tree data structure support wait-free operations, including asymptotically efficient aggregate range queries with logarithmic amortized time. The main idea is that the execution of an operation Op by a process P at node v begins by inserting the descriptor of Op into the root queue root. Queue, and obtaining a timestamp. Then, the process P helps to perform all pending operations in the queue, applies itself, and proceeds recursively at the children nodes, applying the same pattern. Thus, the process traverses the tree downwards, from the root to the appropriate lower nodes, at which the operation (e.g., an insertion of a new data item, or a removal of an existing one) should be performed. This method can be seen as a wait-free version of the classic hand-over-hand locking technique [23]. where instead of blocking we ask processors to perform work that is preceding them in the queue. We name this method as hand-over-hand helping. In the following, we describe this construction in detail, using a binary search tree as a running example. Finally, we provide a practical implementation of such a tree, supporting insert, delete, contains, and count operations. We validate the fact that our design permits the efficient implementation of various types of range queries while achieving non-trivial scalability.

A. Related work

Lock-based solutions. The easiest and the most obvious way to implement a concurrent data structure is to protect a sequential data structure with a *lock* to guarantee mutual exclusion [25]. Such construction is not lock-free (it is not even obstruction-free) and suffers from starvation. Moreover, since a lock allows only one process to work with the data structure at a time, the resulting construction does not scale and its throughput remains low.

Linear-time solutions. Several papers [8], [9], [12], [17], [32] address the issue of executing lock-free (and even wait-free, but with lock-free scalar queries) range queries on concurrent trees. However, the aforementioned papers address only the collect (min, max) query, returning the list of keys, located within a range [min; max]. All other range queries are proposed to be implemented on top of the collect query. For example, as we said before, the count query can be implemented as count (min, max) = collect (min, max).length().

This approach suffers from a major drawback: the collect query is executed in time proportional to the number of keys in the range. Thus, for wide ranges, such query takes O(N) time where N is the size of the tree: the number of keys in the range is almost equal to the size of the tree. This implementation is not asymptotically efficient: e.g., the count query can be executed in $O(\log N)$ time in a sequential environment using balanced search trees.

Therefore, despite being lock-free, these methods do not guarantee time efficiency, and thus cannot be used.

Persistent data structures. There exists a solution for efficient aggregate range queries based on persistent data structures [3]. Each read-only operation (e.g., contains

or count) takes the current version of the data structure and operates on it. Each update operation (e.g., insert or remove) creates a new version of the data structure without modifying the existing one and then tries to replace the old version with the new one using a Compare-And-Swap [1] (or CAS, for simplicity). If the CAS succeeds the operation finishes, otherwise the operation restarts from the very beginning. This approach is called Lock-free Universal Construction [23] and can be applied to any sequential persistent tree. As an interesting observation, this approach scales even on writeonly workloads [5]. However, there are at least two drawbacks: 1) we cannot provide strong fairness guarantees — one operation can restart infinitely often if we are not lucky enough; 2) for an update range query, the majority of computation time will be spent needlessly — since unsuccessful CAS makes us retry the whole operation from the very beginning. For more information about this approach, we point to [5].

Parallel augmented persistent trees. Sun, Ferizovic, and Belloch [31] presented a persistent augmented tree that can serve a batch of operations in parallel using *fork-join* parallelism. The paper does not propose a method of executing concurrent operations on augmented data structures. However, we can use various combining techniques [7], [21], [30] to form large batches of operations from individual concurrent updates. The main problem with this approach is that the combining techniques increase individual operation latency and, thus, are not acceptable in settings, where low operation latency is required.

II. OVERVIEW OF THE APPROACH

A. Timestamps invariant

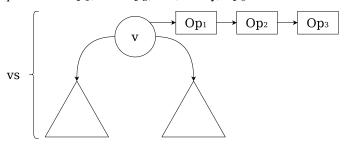
The main problem with the sequential algorithm for an aggregate range query presented in the introduction is that it will be incorrect if running as is in a concurrent environment. Indeed, each update operation (e.g. insert or remove) should modify not only the tree structure, but the augmentation values on the path (e.g., subtree sizes). By that, the augmentation values may become inconsistent with the tree structure.

Therefore, the main purpose of our concurrent solution is to get rid of such situations by ensuring that all operations are executed in a particular order. We enforce a particular execution order by maintaining an operation queue in each node.

Consider an arbitrary node v and its subtree vs. At v we maintain an *operations queue*, that contains descriptors of operations to be applied to vs (Fig. 1). These operations can, for example, insert a key to vs or remove a key from vs. We maintain the following invariant: operations should be applied to vs in the order, their descriptors were added to v queue.

Note that the aforementioned invariant can be applied to the root node too: indeed, since the whole tree is just the subtree of the root operations should be applied to the tree in the order their descriptors were added to the operations queue in the root. Thus, the order, in which operation descriptors are

Fig. 1: Node v has an operations queue with descriptors of three operations: Op_1 , Op_2 and Op_3 . These three operations should be applied to vs in the order of descriptors in the queue: first Op_1 , then Op_2 , and, finally, Op_3



added to the queue in the root, is exactly the *linearization* order.

Thus, we may use the operations queue at the root to allocate timestamps for operations. A timestamp allocation mechanism should provide the following guarantee: if a descriptor of operation A was added to the root queue before a descriptor of operation B, then timestamp(A) < timestamp(B) should hold. We explain how to achieve it in Section II-D. We store the timestamp of an operation in the corresponding descriptor, i.e., descriptor. Timestamp field.

As was stated before, operations should be applied to the tree in the order, their descriptors were added to the root descriptor queue. Therefore, one can wonder: how can we achieve parallelism, while linearizing all operations via the root queue? Note that there is no parallelism only in the queue in the root. Lower by the tree, two operations (even the modifying ones, e.g., two inserts) may be executed in parallel if they are executed on different subtrees, since on lower tree levels their descriptors will be placed to different operation queues (Fig. 2).

B. Operation execution: overview

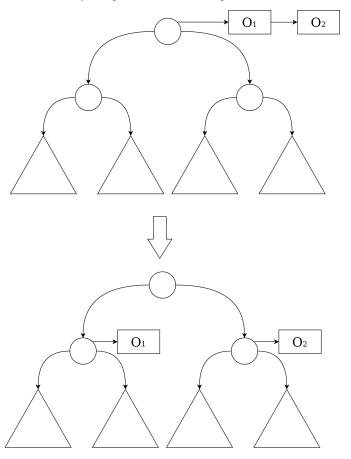
For simplicity of the overview, we consider only unbalanced trees for now. If we want to make our tree balanced, we can adapt the subtree rebuilding approach (we provide a detailed description in Section II-E). The study of other concurrent balancing strategies we leave for the future work.

The execution of an operation Op by a process P (we call such process P the *initiator* process) begins with inserting the descriptor of Op into the root queue and obtaining Op timestamp. In Section II-D, we describe, how the root queue with timestamp allocation may be implemented.

After that, the initiator process starts traversing the tree downwards, from the root to the appropriate lower nodes, at which the operation (e.g., an insertion of a new data item, or a removal of an existing one) should be performed.

In each visited node v some additional actions should be performed in order to execute Op properly. For example, during the count query the size of v's subtree can be added to the result, and during insert or remove operations pointers to v's children and v's subtree size can be changed. We call the

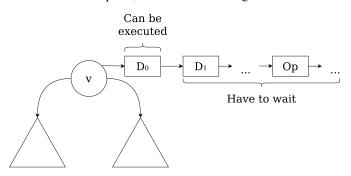
Fig. 2: Two operations can be executed in arbitrary order (even in parallel) if they operate on different subtrees, since on lover tree levels they are placed to different queues



process of performing these necessary actions — an execution of operation Op in node v.

As stated in the previous subsection, operations should be applied to v's subtree in the order their descriptors appear in v's operations queue. Thus, if the descriptor of Op is not located at the head of v's queue the initiator process P has to wait before executing Op in node v (Fig. 3). The execution of Op in node v cannot begin until execution of all the preceding operations in node v is finished.

Fig. 3: Process P has to wait before executing Op in node v, since only the operation D_0 , corresponding to the descriptor at the head of v queue, can be executed right now in v.



To make the algorithm wait-free we use the *helping* mechanism (e.g., [20], [27]): instead of merely waiting for the Op descriptor to move to the head of v queue, P helps executing in node v the operation from the head of the queue — D_0 in the example above. Thus, even if the initiator process of D_0 is suspended, the system still makes progress.

As discussed later, while helping to execute operations D_0, D_1, \ldots in node v the process P removes descriptors of these operations from the head of v's queue and inserts them to queues of appropriate v's children. Thus, while helping other processes execute their initiated operations in v, P moves Op descriptor closer to the head of v queue. Once P helped all preceding operations to finish their execution in node v, it can finally execute its operation Op in v (note that some other process may help executing Op in v, just like P previously helped executing D_0 in v).

The process of executing an operation Op in a node v consists of the following actions:

- 1. Determine the set of child nodes C, in which Op execution should continue.
 - For example, an execution of the count query on a binary search tree may continue in either single child or both children, as explained in Section I.
- 2. For each child c from the set C:
- 2.1. Modify the state of c (e.g., a size of c's subtree), if necessary;
- 2.2. Try to insert Op descriptor to the end of c's operations queue, thus allowing Op to continue its execution at lower levels of the tree.
- 3. Remove Op descriptor from the head of v's queue.

Note, that during the execution of operation Op in node v the said operation only modifies states of v's children, not v itself. Thus, no operation can ever modify the root state, since the root is not a child of some other node. We overcome that issue by the introduction of the *fictive root*. This fictive root does not contain any state and has only one child — the real tree root. The only purpose of the fictive root is to allow operations to modify the state of the real root. The state of the real root can be modified by operation Op while Op is being executed in the fictive root, since the real root is the child of the fictive root.

In Section II-C, we describe how an operation Op should be executed in a node v.

Since now we force processes to help each other, operation Op, initiated by process P, in any node v can be executed by some other helper process. Thus, we need to provide a mechanism for the process P by which it distinguishes between the two following situations:

- Operation Op has not yet been executed in node v. Thus, the descriptor of Op is still located somewhere in v queue.
 In that case, P needs to continue executing operations from the head of v queue in node v.
- Operation Op has already been executed in node v. In that case, P can proceed to execute Op in lower nodes of v's subtree.

We use timestamps to distinguish between these two situations. We describe that usage of timestamps with formulating and proving *timestamps increasing property*.

Theorem 1. In each queue, operation timestamps form a monotonically increasing sequence. More formally, if at any moment we traverse any queue Q from the head to the tail and obtain $t_1, t_2, \dots t_n$ — a sequence of timestamps of descriptors, located in Q, then $t_1 < t_2 < \dots < t_n$ will hold

We prove that theorem in Appendix D.

As follows from that property, the initiator process P can easily learn, whether its operation Op has been executed in node v by using the simple algorithm:

- if the queue is empty we conclude that Op has been executed in v;
- if the queue is not empty, we compare the timestamp of the descriptor in the head of v queue with the timestamp of Op: if v.Queue.Head.Timestamp > Op.Timestamp, we conclude that Op has been executed in v, otherwise, we conclude that Op has not been executed in v yet.

Therefore, we can implement the algorithm of executing all operations from v's queue up to Op. Timestamp (Listing 1):

```
ifun execute_until_timestamp(Op, v):
while true:

// obtains the first descriptor in FIFO order
head_descriptor := v.Queue.peek()
if head_descriptor = nil:
    return

if head_descriptor.Timestamp > Op.Timestamp:
    return

// execute_in_node changes states of v children
// pushes head_descriptor to child queues,
// removes head_descriptor from v queue
execute_in_node(head_descriptor, v)
```

Listing 1: The algorithm to execute all operations, up to the specified timestamp Op. Timestamp, from v queue

Suppose the initiator process P is traversing the tree to execute operation Op and P just finished executing Op in node v. How can P choose the next node in the traversal? It is not necessary to always continue the traversal in one of v's children, since Op can be now finished in v subtree by

other helper processes. To address this issue, in each operation descriptor we store a queue with nodes <code>Op.Traverse</code>—the queue of nodes that must be visited during the execution of <code>Op. The Traverse</code> queue is maintained and used in the following way:

- When any process (no difference initiator or helper) starts executing Op in node v, it adds to the tail of Op.Traverse all children of node v in which the execution should continue;
- When the initiator process finishes the procedure execute_until_timestamp(Op.Timestamp, v), it removes v from the head of Op.Traverse queue. Note, that only the initiator process can remove nodes from Op.Traverse queue;
- After the initiator process has removed the current node v from the head of Op.Traverse, it checks Op.Traverse: if it is empty, the operation is completed and the initiator returns the query result to the caller; otherwise (if Op.Traverse is not empty), the initiator continues the traverse by taking the next node from the head of Op.Traverse.

Note, that this queue maintenance scheme allows a node v to be inserted into Op.Traverse multiple times, since multiple helper processes may be executing Op in v parent in parallel. However, as will be explained in Section II-C v's state will still be modified exactly once, no matter how many times it is processed. The traverse algorithm can be implemented as in Listing 2.

```
1 fun execute_operation(op):
2     Tree.Root.Queue.push_acquire_timestamp(op)
3     op.Traverse = {Tree.Root}
4     while true:
5          v := op.Traverse.peek()
6          if v = nil: // op is finished
7          return
8          execute_until_timestamp(op.Timestamp, v)
9          op.Traverse.pop()
```

Listing 2: The algorithm for traversing the tree

Now we have to design a method, that will allow the initiator process to learn the operation result when the operation is completed. The problem here is that the operation result might consist of multiple parts (e.g., count result consists of a sum of multiple subtree sizes), and these parts (e.g., subtree sizes) may be computed by different processes, since force processes to help each other.

To allow operation result to be assembled from these parts, in each operation descriptor we store a concurrent map op.Processed, filled with nodes, in which the execution of op has been finished. The size of this map is expected to be small for aggregate range queries (e.g., $O(\log N)$), so, we can implement them in any way we want: a wait-free queue that stores all the required nodes (maybe multiple times, which we filter out at the end of the operation) or with a Wait-free Universal Construction [22], and, finally, we can use a wait-free map.

The Op.Processed uses tree nodes as its keys. To allow this, we augment each tree node v with an identifier, stored in

the v.Id field. Each node receives its identifier at the creation moment and the node identifier does not change throughout the node lifetime. The node identifiers must be unique. We can achieve that property using UUID [4] generation procedure or by incrementing fetch-and-add [2] counter.

Values of the Op.Processed map store parts of the result: for example, for the count query we store in the Op.Processed the node identifiers with the sizes of their subtrees that should be added to the result of the query.

Before removing Op descriptor from the head of v's queue we try to add v.Id along with a value x, corresponding to the part of the answer for the node v, into the Op.Processed map. If key v.Id already exists in the Processed map, we left the Op.Processed map unmodified, without changing the value, associated with v.Id.

We never modify the value, associated with node v, since stalled processes can calculate the value incorrectly. Indeed, consider the following scenario:

- 1. Descriptor D, corresponding to a count operation with timestamp 42, is located at the head of v's queue;
- 2. Process P reads D from the head of v's queue;
- 3. Process P is suspended by the OS;
- 4. Process R reads D from the head of v's queue;
- 5. Process R determines that the size of v's left subtree should be added to the result;
- Process R reads the size of v's left subtree (say, it equals to 5) and adds key-value pair (v.Id, 5) to the Processed map;
- 7. A new key is inserted to v left subtree by insert operation with timestamp 43, making v left subtree size equal to 6;
- 8. Process P is resumed by the OS;
- Process P reads the size of v's left subtree (now it equals to 6) and tries to add key-value pair (v.Id, 6) to the Processed map.

On step (9) we should not modify the value, corresponding to the node v, since the value 6 reflects the modification, performed by the insert operation with timestamp 43. The count operation has timestamp 42, thus, the count result should not include the key, inserted by insert operation with timestamp 43.

When the operation execution is finished (i.e., Op.Traverse is empty) we traverse the Processed map, forming the query result from partial results associated with visited nodes. Note, that it is safe to traverse the Processed map — indeed, now the Processed map cannot be modified concurrently, since the query execution is finished.

C. Detailed description of an execution in a node

In Section II-B, we explained how the execution of the operation works in general. Now, we go into details of the execution in the node.

The process of executing an operation Op in a node v consists of the following actions:

- Determine the set of child nodes C, in which Op execution should continue.
- \bullet For each child ${\tt c}$ from the set ${\tt C}:$
 - 1. Insert c into Op. Traverse queue;
 - 2. Modify the state of c (e.g., a size of c's subtree), if necessary;
 - 3. Insert Op descriptor to the operations queue of c, thus allowing Op to continue its execution at lower levels of the tree.
- Try to add v.Id along with a value x, corresponding to the part of the answer for the node v, into Op.Processed map.
- Try to remove Op descriptor from the head of v's queue if it is still there.

The removal of Op descriptor from the head of v's queue should be done after the insertion of Op descriptor to child queues and modification of child states are finished. Otherwise, the execution of later operations in v may start before the execution of Op in v is finished, which may break the main invariant (Section II-A). Inserting the descriptor to child queues, modifying child states, and removing the descriptor from the parent queue should happen exactly once, no matter how many processes are working on the descriptor concurrently.

Exactly-once insertion to and removal from queues is handled by our implementation of concurrent queues (see Section II-D). Queues provide two procedures:

- push_if inserts the descriptor to the tail of the queue only if it has not been inserted yet, otherwise, the queue is left unmodified.
- pop_if removes the descriptor from the head of the queue only if it has not been removed yet, otherwise, the queue is left unmodified.

The main problem in the execution of an operation Op in a node v is the proper work with the children states: we should be able to work with each state atomically and we should modify each state exactly once, no matter how many processes are executing Op in v.

The atomicity problem comes from the fact that the state may consist of multiple fields. To solve this problem, we do not store the state directly inside the node — instead we store the immutable state in the heap and the node stores the pointer S_Ptr to it.

The state, located in the heap, is considered immutable and is never modified. To modify the node state, we simply do the following:

- create the structure, corresponding to the modified state, with an arbitrary set of fields changed;
- 2. place the modified state somewhere in the heap;
- change the node.S_Ptr, so that it points to the new state.

To read the state atomically, we simply read the S_Ptr register. After that, we can safely access any field from the state structure, pointed at by the fetched pointer, without worrying that the state structure is being modified concurrently

by another process. Since the structure is immutable, it can never be modified by another process.

Now, we return to the second problem of modifying the state exactly once. In the state we store one additional field: Ts_Mod — timestamp of the operation, that was the last to modify the state. Thus, if the operation Op is willing to modify the state of node v, we should first read the current v's state and acquire the last modification timestamp.

- If Ts_Mod \geq Op. Timestamp we conclude that v's state has been already modified by Op. In that case, we simply do not try to modify v's state according to Op anymore.
- Otherwise, we create a new state (with Ts_Mod = Op.Timestamp) and try to change the state pointer using CAS(&v.S_Ptr, cur_state, new_state). We then go to the next step, no matter what was the CAS result. If the CAS returned true we have successfully modified the state, otherwise (if the CAS returned false), some other process has already modified the state according to Op.

Thus, the state is modified with each executed operation exactly once. Indeed, even if some stalled process will try to modify node v with an already applied operation Op the node state will not be changed, since the last modification timestamp is greater than or equal to Op. Timestamp. Therefore, the algorithm can be implemented in the following way (Listing 3):

```
i fun execute_in_node(op, v):
      C := /* set of v children in which
          execution of op should continue */
      for c in C:
          cur_state := v.State_Ptr
          op.Traverse.push(c)
          if cur_state.Ts_Mod < op.Timestamp:</pre>
               new_state := op.get_modified_state(cur_state)
               new_state.Ts_Mod := op.Timestamp
               CAS (&v.State_Ptr, cur_state, new_state)
          c.Queue.push_if(op)
11
      node_key := v.Id
      node_value := /* part of the result
13
14
          corresponding to v \star/
15
      op.try_insert(node_key, node_value)
      v.Queue.pop_if(op)
```

Listing 3: Algorithm for executing operation op in node v

D. Implementation of an operations queue

Queue structure. For our purpose, we can use any practical queue algorithms as a basis for our descriptors queues, e.g., fetch-and-add queue [33] or practical wait-free queue [24]: the final implementation remains almost the same. However, for simplicity of the presentation, we use Michael-Scott queue. This queue is lock-free which makes the whole algorithm lock-free. But if we make the root queue to be wait-free — all other queues based on Michael-Scott queue will automatically have the same progress guarantee due to the way how we work with the descriptors. For more information about the wait-freedom see Section II-F.

In each node of the queue we store the descriptor in field Data and the pointer to the next node in field Next. Also,

we have two pointers: Tail, that points to the last node of the queue, and Head, that points to the node *before* the first node of the queue. Note that the node at Head pointer does not store any data, residing in the queue. This node is considered dummy and only the node at Head.Next pointer contains the first real descriptor in the queue.

Queue in the root.

As discussed in Section II-A, the operation queue in the root node should provide timestamp allocation mechanism, with the following guarantees: if the descriptor of operation A was added to the root queue before the descriptor of the operation B, then timestamp(A) < timestamp(B) should hold.

As stated above, we can use a slight modification of Michael-Scott queue [27] to implement the timestamp allocation mechanism for the root queue. Each time we need to add a new descriptor to the root queue, we 1) create a new node with the descriptor; 2) take the timestamp of the tail; 3) set the new timestamp in our descriptor as the incremented timestamp of the tail; 4) try to move the queue tail to the new node using CAS; 5) if the CAS is successful we stop, otherwise, we repeat from step (2).

In Section II-F, we show how to implement such queue in a wait-free manner.

push_if implementation. As discussed in Section II-C, non-root queues should provide push_if operation that inserts a descriptor into the queue if it was not inserted yet (otherwise, the queue should be left unmodified). The procedure is based on the Michael-Scott queue insertion algorithm [27]: we check the timestamp of the tail, if it is higher then the descriptor has been inserted and we leave the queue unmodified, otherwise, we try to move the queue tail to the new node using CAS.

pop_if implementation. As discussed in Section II-C, the operation queue in any node should provide pop_if operation, that tries to remove descriptor with the specified timestamp TS from the head of the queue. If descriptor D with timestamp TS is still located at the head of the queue, it is removed. Otherwise, the queue is left unmodified — in this case, we assume that D was removed by some other process. We assume that at some moment D was located at the head of the queue (it may still be located at the head of the queue or it may be already removed), i.e., we never try to remove a descriptor from the middle of the queue. We can do this using Michael-Scott queue [27].

E. Balancing strategy

Until now, we considered unbalanced trees which may have $height \in \Omega(\log N)$. Since most of the queries (e.g., insert, remove, contains ,and count) are executed on a tree in $\Theta(height)$ time, using unbalanced trees may result in these queries being executed in non-optimal $\omega(\log N)$ time. Therefore, we must design an algorithm to keep the tree balanced. One possible balancing strategy is based on a subtree rebuilding and is similar to the balancing strategy proposed in [6], [14], [26], [28]. The idea of this approach can be formulated the following way: when the number of

modifications in a particular subtree exceeds a threshold, we rebuild that subtree making it perfectly balanced.

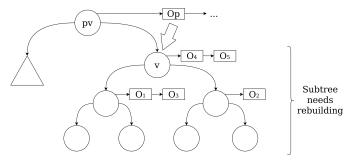
For each tree node we maintain Mod_Cnt in the node state — the number of modifications in the subtree of this node. Moreover, for each node we store an immutable number Init_Sz — the initial size of its subtree, i.e., the number of data items in that node subtree at the moment of node creation (node can be created when a new data item is inserted to the tree or when the subtree, where the node is located, is rebuilt). We rebuild the node subtree when Mod_Cnt $> K \cdot Init_Sz$, where K is a predefined constant. This approach makes the rebuilding to take O(1) amortized time and, thus, the rebuilding does not affect amortized total cost (according e.g., to [26]).

We check whether the subtree of v needs rebuilding (and perform the rebuilding itself) before inserting an operation descriptor to v's queue and changing v's state. Therefore, we can perform v's subtree rebuilding only during execution of some operation in v's parent.

Consider node v, its parent pv and operation Op, that is being executed in pv and that should continue its execution in v's subtree (and, therefore, its descriptor should be inserted to v's queue). Before inserting Op to v's queue and changing v state, we check whether Mod_Cnt in v will exceed the threshold after applying Op to v's subtree: if so, v subtree must be rebuilt.

Note, that the subtree of v can contain unfinished operations: their descriptors still reside in the queues in that subtree (Fig. 4).

Fig. 4: The subtree that needs rebuilding may contain descriptors of unfinished operations



As the first step, we should finish all these unfinished operations before rebuilding the subtree. To do so, we traverse the subtree and in each node $u \in subtree(v)$ execute all operations, residing in u queue. After that, we again traverse the subtree of v, that no longer contains unfinished operations, and collect all the stored data items (e.g., keys or key-value pairs). Then, we build an ideally balanced subtree, containing all these data items.

Each node of the new subtree should be initialized with $Mod_Cnt = 0$ and contain correct Init_Sz. We should set Ts_Mod of each node in the rebuilt subtree so that Op and all later operations (with timestamp \geq Op.Timestamp) can still modify the new subtree, but all the preceding opera-

tions (with timestamp < Op.Timestamp) cannot. Thus, we set Ts_Mod = Op.Timestamp - 1.

After that, we take nv — the root of the new subtree and try to modify the pointer that pointed at v, so that it starts to point at nv. For example, if v was the left child of pv, we execute CAS (&pv.Left, v, nv); if v was the right child of pv, we execute CAS (&pv.Right, v, nv). If the CAS returned true we conclude that we have successfully finished the rebuilding; if the CAS returned false we conclude that some other process has completed the rebuilding before us. In either case we resume the execution of Op in pv: we read nv — new root of the subtree, modify nv's state, insert Op descriptor to nv's queue (here we re-read root of the subtree because nv can be root of the subtree build not by our process, but by some another helder process) and remove Op descriptor from pv queue.

F. Wait-freedom

We now prove that our solution can be implemented efficiently with wait-free progress guarantee. We recall that *wait-freedom* [22] is a progress guarantee that requires all non-suspended processes to finish their execution within a bounded number of steps.

Theorem 2. Each operation Op in our solution finishes within a bounded number of steps.

To prove that theorem we recall that the execution of operation Op consists of:

- 1) Inserting Op descriptor into the root queue;
- 2) Propagating Op descriptor downwards, from the root to the appropriate lower nodes;
- 3) Executing Op in each node v on the target path.

Now, we prove that each of these stages finishes within a bounded number of steps.

Lemma 1. The insertion of a descriptor into the root queue finishes within a bounded number of steps.

Proof. Our queue implementation, described in Section II-D is lock-free, but not wait-free, since it is just a version of Michael and Scott queue [27].

The simplest approach is to implement the wait-free root queue using the well-known Wait-free Universal Construction [22], with no implementation caveats.

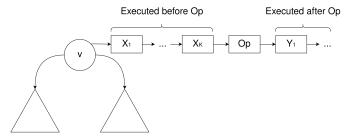
However, this approach has a very huge overhead. We hope that some practical wait-free queue (e.g., [24], [33]) can emulate our root queue and its timestamps distribution. Unfortunately, a wait-free queue from [33] can support the increasing timestamps using cell identifiers for that, but do not allow a simple wait-free peek function, that reads the head of the queue but does not remove it — this functionality is crucial for our queue in pop_if. Luckily for us the wait-free queue from [24] supports wait-free peek function and supports non-decreasing timestamps (or epochs in the paper). We can make them strongly increasing using a fetch-and-add register.

To distribute the timestamps, we need a version variable and an array of size P that contains the current descriptors. Each descriptor has an empty timestamp variable at the initialization. When performing an operation, process π creates a new descriptor and puts it into the corresponding cell. Then, it gets new version from the version variable using fetch-and-add and tries to CAS the current empty timestamp in its descriptor to the obtained version. Not depending on the result of CAS, the descriptor of π has a timestamp. Then, π traverses the array of descriptors and replaces empty timestamps by a newly fetched version. Also, π saves the descriptors with the timestamp smaller than the one in its descriptor. Finally, the process tries to enqueue into the root queue all these descriptors in the sorted order of their timestamps. Thus, the algorithm works in $O(P \log P)$ time.

Lemma 2. In each tree node v on the Op traversal path executing Op in v finishes in a finite number of steps.

Proof. Consider an operation queue at node v (Fig. 5). Here some operations $(X_1 ... X_K)$ should be executed before Op, while all other operations $(Y_1 ...)$ will be executed only after execution of Op in v is fully completed. Thus:

Fig. 5: Operation queue structure at node v



- We help to complete only a finite number of operations in a node v, since there cannot be more than |P| operations in the queue of v before Op (where P is the set of the processes executing operations);
- Each operations X_i takes a finite number of steps to complete its execution in a node v (see Section II-C for the list of those steps). Note, that in the process of execution operation Op in node v we never retry any operation (in contrast to lock-free algorithms, e.g., in [27]): for example if the insertion of Op descriptor to child node cv fails, we conclude that Op descriptor has been inserted to cv by another helper process and merely continue the execution of Op in v;

Therefore, executing Op in v finishes in a constant number of steps.

Lemma 3. Propagating the descriptor downwards, from the root to the appropriate lower nodes finishes within a bounded number of steps

Proof. Consider some operation Op_2 such that Op_2 . Timestamp > Op. Timestamp. If both Op_2

and $\bigcirc p$ are willing to change the very same tree node \lor , Op_2 under any conditions will do it after $\bigcirc p$, since the operations are executed in a strict timestamp order (see Section II-A for details). Thus, Op_2 cannot somehow change the structure of the tree to disrupt $\bigcirc p$'s traversal. Therefore, $\bigcirc p$ will finish its traversal in a constant amount of steps, since later operations cannot interfere in $\bigcirc p$ traversal. Since none of the later operations can overcome $\bigcirc p$, we note the following:

- At the moment when Op begins execution the size of tree is N and no more than |P| concurrent processes are inserting new nodes in the tree. Thus at the Op.Timestamp moment the size of the tree will no exceed O(N+|P|), which is definitely a finite number;
- By Lemma 2 operations takes a finite number of steps to execute in a node.

Thus, the operation takes a finite number of steps to finish its traversal. \Box

Note, that our rebuilding procedure does not fail the waitfreedom guarantee in the proof above since each rebuilding finishes in a bounded number of steps.

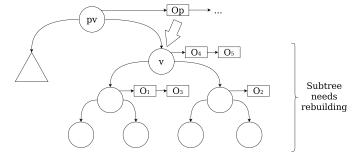
Lemma 4. The rebuilding procedure finishes in a bounded number of steps

Proof. Indeed, the rebuilding procedure of a subtree vs consists of the following steps:

- Traverse the subtree vs, collecting all unfinished operations;
- Help to complete all these unfinished operations;
- Collect all keys from vs;
- Build an ideal tree from collected keys.

Note, that only the operations that started before Op can be unfinished in vs (Fig 6), since we execute operations in the timestamp order.

Fig. 6: Unfinished operation $O_1, O_2, \dots O_5$ have timestamp lower than Op.Timestamp



Therefore: 1) there is a finite set of unfinished operations in vs; 2) a completing of each unfinished operations takes a finite number of steps by Lemma 3; 3) vs has a finite size, thus, the collecting all keys from vs and the construction of a new ideal subtree also takes a finite amount of steps. Thus, the rebuilding completes in a finite amount of steps.

G. Time cost analysis

We now estimate the time it takes to execute an operation in our solution.

Theorem 3. The amortized cost of insert, remove, contains or count operation on our concurrent binary search tree with rebuilding is $O((\log N + |P|) \cdot |P|)$

Proof. Suppose N is the size of the tree when the operation Op starts its execution. In a sequential setting each of these operations takes $O(\log N)$ time since it visits $O(\log N)$ nodes performing O(1) operations in each node. In concurrent setting, up to |P| other processes can be inserting their keys to the tree concurrently with Op, thus, at the moment of Op. Timestamp the size of the tree will not exceed N+|P|, therefore the amortized number of nodes Op will traverse is $O(\log N+|P|)$ (since the tree is balanced).

In each node v no more than |P| descriptors will be located closer to the head of v.Queue than the descriptor of our operation Op. Each operation takes O(1) amortized time to execute (the rebuilding takes O(1) amortized time as stated e.g., in [26]), thus, Op takes O(|P|) amortized time to finish its execution in each node.

Therefore, amortized Op execution cost is $O((\log N + |P|) \cdot |P|)$.

Theorem 4. When the workload is uniform (i.e., each data item is equally likely to be queried) insert and remove take $O(\log N + |P|)$ amortized time on our concurrent binary search tree with rebuilding.

Proof. Consider the size of the root operation queue. Since there exist up to |P| processes executing operations concurrently, the size of root operation queue is O(|P|).

Let us see, in which nodes these operations will continue their execution. Since each data item is equally likely to be queried, approximately half descriptors continues their execution in root.Left node, and the other half continues their execution in root.Right node. Therefore, the expected size of operation queue in each node of the second tree level is $O(\frac{|P|}{2})$.

Following the same reasoning, the expected size of operation queue in each node of the third tree level is $O\left(\frac{|P|}{2^2}\right) = O\left(\frac{|P|}{4}\right)$ and the expected size of operation queue in each node of the k-th level of the tree is $O\left(\frac{|P|}{2^{k-1}}\right)$.

Since the tree is balanced, the operation traverses $O(\log N + |P|)$ nodes. The expected amortized number of operations performed in i-th node is $O\left(\max\left(\frac{|P|}{2^{k-1}},1\right)\right)$ since the amortized cost of executing a single operation in a node is O(1) (of course, in each node we perform at least O(1) operations).

Therefore, the total expected amortized cost of performing an operation is $O\left(\sum_{k=1}^{\log N+|P|} \max\left(\frac{|P|}{2^{k-1}},1\right)\right) = O(\log N + |P|).$

III. EXPERIMENTS

According to the framework described in Section II, we implemented a concurrent balanced binary search tree that supports insert, remove, contains, and count queries. The code is written in Kotlin.

We decided to test our data structure only against the concurrent persistent tree presented in [5], since it is the only available data structure that supports asymptotically efficient range queries (e.g., can execute count queries in logarithmic time).

We test the implementations on the following workloads: 1) a read-heavy workload that runs contains operations; 2) an insert-delete workload with half insertions and half deletions on a random keys drawn from a range so that each operation is successful with a probability of approximately 0.5; 3) a successful-insert workload where we insert a random key from a very wide range (from -2^{63} to $2^{63}-1$) so that all insertions are successful with the very high probability. We consider these experiments more as preliminary rather than the full-detailed ones.

All our experiments are performed on Intel Gold 6240R with 24 cores. We decide to run on one socket due to the heavy load on the memory by our search tree. The plots show the throughput of the data structures, i.e., the number of operations in 10 seconds. Each point on the plots is obtained as an average of 5 separate runs. The blue lines are for our data structure, and the orange lines are for the persistent tree.

Contains Benchmark. We fix the key range as $[1, 2 \cdot 10^6]$. At first, we initialize a data structure — each element from the range is inserted with the probability 1/2. Then, we start T threads. Each thread for 10 seconds searches for a key taken uniformly at random from the range. As shown in Figure 7, our data structure does not have a large overhead for contains operations.

Insert-Delete Benchmark. We fix the key range as $[1,2\cdot 10^6]$. At first, we initialize a data structure — each element from the range is inserted with the probability 1/2. Then, we start T threads. Each thread for 10 seconds chooses the operation (insert/delete) uniformly at random and the argument uniformly at random from the range. As shown in Figure 8, our data structure starts worse due to the larger overhead, but it works under contention better than the persistent tree.

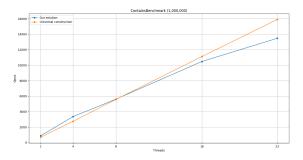


Fig. 7: Contains Benchmark.

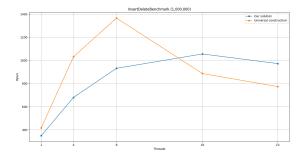


Fig. 8: Insert-Delete Benchmark.

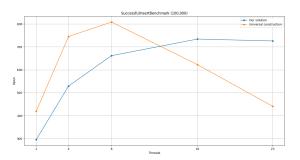


Fig. 9: Successful-Insert Benchmark.

Successful-Insert Benchmark. We initialize a data structure with 10^6 random integer elements. Then, we start T threads. Each thread for 10 seconds inserts random integers. With the very high probability each insertion is successful which affects the persistent tree very much. As shown in Figure 9, our data structure starts worse due to the larger overhead, but it works under contention better than persistent tree.

Outcome. Our experiments show that our data structure works better than the only existing solution with aggregate range queries on update-heavy workloads and has a small overhead on contains operations while supporting efficient aggregate range queries.

IV. CONCLUSION

We present an approach to obtain concurrent trees with efficient aggregate range queries in a wait-free manner. Our practical results validate our performance and scalability claims. We propose a number of avenues for future work. First, we can make the rebuilding collaborative [14], i.e., make different processes work together to rebuild a single subtree. Then, in order to achieve pure $O(\log n)$ complexity, instead of the amortized one, we can use another rebuilding strategy — the top-down rebuilding from the chromatic tree [13]. Another interesting question is how to decrease the number of allocations — now we use too many memory. Finally, it would be good to implement other tree data structures, e.g., quad-trees or tries.

REFERENCES

- [1] compare-and-swap, 2022. URL: https://en.wikipedia.org/wiki/ Compare-and-swap.
- [2] fetch-and-add, 2022. URL: https://en.wikipedia.org/wiki/Fetch-and-add.
- [3] Persistent data structures, 2022. URL: https://en.wikipedia.org/wiki/ Persistent_data_structure.
- [4] Universally unique identifier, 2022. URL: https://en.wikipedia.org/wiki/ Universally_unique_identifier.
- [5] Vitaly Aksenov, Trevor Brown, Alexander Fedorov, and Ilya Kokorin. Unexpected scaling in path copying trees. In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, pages 438–440, 2023.
- [6] Vitaly Aksenov, Ilya Kokorin, and Alena Martsenyuk. Parallel-batched interpolation search tree. In *International Conference on Parallel Computing Technologies*, pages 109–125. Springer, 2023.
- [7] Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. Parallel combining: Benefits of explicit synchronization. In 22nd International Conference on Principles of Distributed Systems, 2019.
- [8] Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. ACM SIGPLAN Notices, 53(1):14–27, 2018.
- [9] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 357–369, 2017.
- [10] Rudolf Bayer. R. bayer, e. mccreight organization and maintenance. Software Pioneers: Contributions to Software Engineering, 1:245, 2012.
- [11] Ferenc Bodon and Lajos Rónyai. Trie: an alternative data structure for data mining algorithms. *Mathematical and Computer Modelling*, 38(7-9):739–751, 2003.
- [12] Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *International Conference On Principles Of Distributed* Systems, pages 31–45. Springer, 2012.
- [13] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN* symposium on *Principles and practice of parallel programming*, pages 329–342, 2014.
- [14] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. Non-blocking interpolation search trees with doubly-logarithmic running time. In Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 276–291, 2020.
- [15] Douglas Comer. Ubiquitous b-tree. ACM Computing Surveys (CSUR), 11(2):121–137, 1979.
- [16] Mark Theodoor De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. Computational geometry: algorithms and applications. Springer Science & Business Media, 2000.
- [17] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In The 31st ACM Symposium on Parallelism in Algorithms and Architectures, pages 275–286, 2019.
- [18] Goetz Graefe et al. Modern b-tree techniques. Foundations and Trends® in Databases, 3(4):203–402, 2011.
- [19] Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In 19th Annual Symposium on Foundations of Computer Science (sfcs 1978), pages 8–21. IEEE, 1978.
- [20] Timothy L Harris, Keir Fraser, and Ian A Pratt. A practical multiword compare-and-swap operation. In *International Symposium on Distributed Computing*, pages 265–279. Springer, 2002.
- [21] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the* twenty-second annual ACM symposium on Parallelism in algorithms and architectures, pages 355–364, 2010.
- [22] Maurice Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS), 13(1):124–149, 1991.
- [23] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. The art of multiprocessor programming. Newnes. 2020.
- [24] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. ACM SIGPLAN Notices, 46(8):223–234, 2011.
- [25] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. In *Concurrency: the Works of Leslie Lamport*, pages 171–178. 2019.

- [26] Kurt Mehlhorn and Athanasios Tsakalidis. Dynamic interpolation search. *Journal of the ACM (JACM)*, 40(3):621–634, 1993.
- [27] Maged M Michael and Michael L Scott. Simple, fast, and practical nonblocking and blocking concurrent queue algorithms. In *Proceedings* of the fifteenth annual ACM symposium on Principles of distributed computing, pages 267–275, 1996.
- [28] Aleksandar Prokopec, Trevor Brown, and Dan Alistarh. Analysis and evaluation of non-blocking interpolation search trees. arXiv preprint arXiv:2001.00413, 2020.
- [29] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.
- [30] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multiversioned indexes. *Proceedings of the VLDB Endowment*, 13(2), 2019.
- [31] Yihan Sun, Daniel Ferizovic, and Guy E Belloch. Pam: parallel augmented maps. In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 290–304, 2018.
- [32] Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021.
- [33] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 1–13, 2016.

APPENDIX

Many range queries, especially the aggregating ones, can be executed in sub-linear (e.g. logarithmic) time. Consider, an example of such range query:

count (Set, min, max) = $|\{x \in Set : min \le x \le max\}|$ — the number of keys, located in the range [min; max]. It can be calculated in $O(\log N)$ time on binary search trees (where N is the number of keys in the set), using the following algorithm.

A. Tree structure

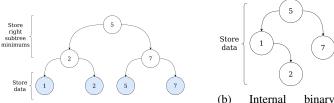
Let us begin with a couple of definitions:

Definition 1. A node is a leaf if it has no children.

Definition 2. A node is an internal node if it is not a leaf.

Definition 3. External binary search tree (Fig. 10a) is a binary search tree, in which keys are stored only in leaf nodes. In contrast, internal nodes store only auxiliary information, used for query routing (e.g., the minimal key, that might be located in the right subtree).

Definition 4. Internal binary search tree (Fig. 10b) is a binary search tree, in which keys are stored in both leaf nodes and in internal nodes.



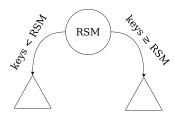
(a) External binary search tree

(b) Internal binary search tree

Fig. 10: Different types of search trees

To explain how to implement the count query, we consider external binary search trees. Each internal node will store Right_Subtree_Min — the minimal key, that might be located in the right subtree. All keys less than Right_Subtree_Min should be stored in the left subtree, and, thus, all scalar queries (insert, remove and contains) on such keys are redirected to the left subtree. Similarly, all keys greater than or equal to Right_Subtree_Min should be stored in the right subtree, and, thus, all scalar queries on such keys are redirected to the right subtree (Fig. 11).

Fig. 11: Using Right_Subtree_Min for query routing



Moreover, each internal node will store the size of that node's subtree — i.e., the number of keys in that node's subtree. Of course, that information should be properly maintained:

- When inserting new key k to the tree, increase by one subtree sizes of each node on the path from the root to the leaf, storing key k (Fig. 12).
- When removing key k from the tree, decrease by one subtree sizes of each node on the path from the root to the leaf, storing key k (Fig. 13).

Fig. 12: Maintaining subtree sizes on node insertion

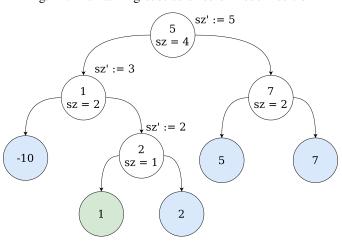
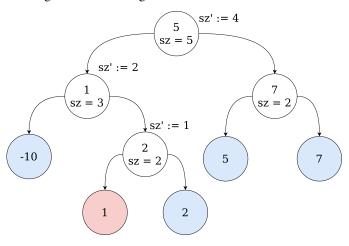


Fig. 13: Maintaining subtree sizes on node removal



Definition 5. We call additional information, stored in tree nodes and required for fast range queries execution, augmentation values.

For example, subtree sizes are augmentation values, required for asymptotically optimal execution of the count range query.

Note, that different range queries may require different augmentations in order to be executed asymptotically optimal. In Chapter ?? we shall describe augmentations, required for fast execution of different range queries.

B. Executing the count query asymptotically optimal

To implement the count query in an asymptotically optimal way, we present the following three functions:

- count_both_borders (node, min, max) returns the number of keys in node subtree, that are located in the range [min; max]
- count_left_border(node, min) returns the number of keys in node subtree, that are greater than or equal to min
- count_right_border(node, max) returns the number of keys in node subtree, that are less than or equal to max

Trivially, count(Set, min, max) =
count_both_borders(Set.Root, min, max).

Let us begin with defining count_both_borders(node, min, max) procedure recursively:

- If node is a leaf, we check whether $\min \le \text{node.Key} \le \max \text{ holds.}$ If so, we return 1, otherwise, we return 0.
- If min ≥ node.Right_Subtree_Min, then all keys from the left subtree are less than min (since for all such keys Key < node.Right_Subtree_Min holds, as guaranteed by the tree structure). Thus, all the required keys are located in the right subtree. Therefore, we return count_both_borders (node.Right, min, max).

- If max < node.Right_Subtree_Min, then all keys from the right subtree are greater than max. Thus, all the required keys are located in the left subtree. Therefore, we return count_both_borders(node.Left, min, max).
- Otherwise, min < node.Right_Subtree_Min</pre> < max. In that case, some satisfying keys may be located in the left subtree, and some of them may be located in the right subtree. Thus, we return count_both_borders (node.Left, min, node.Right_Subtree_Min) + count both borders (node. Right, node.Right Subtree Min, max). case, we call node with such a condition a fork node. Note, that the tree structure guarantees, that all keys in the left subtree are already less than node.Right_Subtree_Min and all keys in the right subtree are already greater than or equal to node.Right_Subtree_Min. Thus, we do not need to check, that keys in the left subtree are \leq node.Right_Subtree_Min and that keys in the right subtree are ≥ node.Right_Subtree_Min these inequations are guaranteed to be true by the tree structure itself. Thus, we return count_left_border(node.Left, min) + count_right_borders(node.Right, max).

Now, we shall define count_left_border(node, min):

- If node is a leaf, we check whether node. Key ≥ min holds. If so, we return 1, otherwise, we return 0.
- If min \geq node.Right_Subtree_Min, then all keys from the left subtree are less than min. Thus, all the required keys are located in the right subtree. Therefore, we return count_left_border(node.Right, min).
- Otherwise, min < node.Right_Subtree_Min. In that case, all the keys from the right subtree are greater than or equal to min. Thus, we should count all keys from the right subtree plus some keys from the left subtree. Therefore, the answer is get_size(node.Right) + count_left_border(node.Left, min).

Size of the right subtree can be calculated easily:

- If node. Right is a leaf, the size of the right subtree is 1:
- Otherwise, node.Right is an internal node —
 in that case the size of the right subtree is
 node.Right.Size;

We can define count_right_border(node, max) in the same manner:

- If node is a leaf, we check whether node. Key ≤ max holds. If so, we return 1, otherwise, we return 0.
- If max < node.Right_Subtree_Min, then all keys from the right subtree are greater than max. Thus, all the required keys are

- located in the left subtree. Therefore, we return
 count_right_border(node.Left, max).
- Otherwise, max ≥ node.Right_Subtree_Min. In that case, all keys from the left subtree are less than max. Thus, we should count all keys from the left subtree plus some keys from the right subtree. Therefore, the answer is get_size(node.Left) + count_right_border(node.Right, max). The size of the left subtree can be calculated similarly to the previous case.

We show how to implement the algorithm in Listing 4¹.

```
fun count_both_borders(node, min, max):
       case node of
       \mid EmptyNode \rightarrow
           /*
           EmptyNode is a dummy node that contains
           neither key nor children.
           We can use it to represent an empty set,
           for example
           return 0
10
       \mid LeafNode \rightarrow
           if min \leq node.Key \leq max:
12
13
                return 1
15
                return 0
       | InnerNode →
           if min > node.Right_Subtree_Min:
17
18
                return count_both_borders(
19
                          node.Right, min, max)
           elif max < node.Right_Subtree_Min:</pre>
20
                return count_both_borders(
                          node.Left, min, max)
22
23
           else:
                    count_left_border(node.Left, min) +
25
                    count_right_border(node.Right, max)
28 fun get_size(node):
29
       case node of
       | EmptyNode -
30
           return 0
         LeafNode \rightarrow
32
33
           return 1
       | InnerNode →
           return node.Size
35
36
37 fun count_left_border(node, min):
38
       case node of
       \mid EmptyNode \rightarrow
39
40
           return 0
       \mid LeafNode \rightarrow
41
           if node. Key \geq min:
42
43
               return 1
45
                return 0
46
       | InnerNode →
           if min > node.Right_Subtree_Min:
48
               return count_left_border(node.Right, min)
49
                return get_size(node.Right) +
50
                        count_left_border(node.Left, min)
```

¹In all subsequent pseudocode listings we denote shared objects (including names of fields, that may be accessed by multiple processes) in Upper_Snake_Case; class names in CamelCase; local variables in lower_snake_case; functions in lower_snake_case; Creation of a new variable is denoted by variable_name := initial_value syntax; Assigning a new value to the existing variable is denoted by variable_name ← new_value syntax;

```
52
  fun count_right_border(node, max):
53
        case node of
54
         EmptyNode 
ightarrow
55
             return 0
56
57
          LeafNode \rightarrow
58
             if node. Key \leq max:
                  return 1
59
             else:
60
61
                  return 0
62
          InnerNode \rightarrow
             if max < node.Right_Subtree_Min:</pre>
64
65
             else:
                  return get_size(node.Left) +
```

Listing 4: Implementation of the count range query

C. count query time complexity

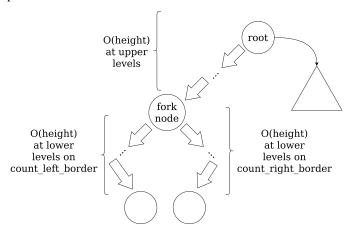
Theorem 5. The time complexity of the count query is O(height).

Proof. We state that both count_left_border and count_right_border work in O(height) time. Indeed, on each tree level both these procedures visit only one node per level, performing O(1) operations in each visited node.

Let us now switch to proving the time complexity of count_both_borders. At upper tree levels (higher than the fork node) it visits one node per level performing O(1)operations in each visited node, giving O(height) time at upper levels.

At one of the nodes (the *fork node*) the execution may fork: we shall call count left border on the left subtree and count_right_border on the right subtree. Note, that the execution can fork at most once and both called procedures have O(height) time complexity. Thus, at lower tree levels the procedure also has O(height) + O(height) = O(height)time complexity. Therefore, the total time complexity of the procedure is O(height) (Fig 14).

Fig. 14: Time complexity of the count_both_borders procedure



Suppose we use balanced binary search trees with $height \in$ $O(\log N)$ where N is the size of the tree. Thus, the count query is executed in $O(\log N)$ time.

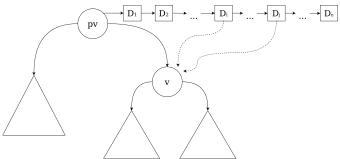
D. Proof of the descriptor timestamp monotony

Theorem 6. In each queue, operation timestamps form a monotonically increasing sequence. More formally, if at any moment we traverse any queue Q from the head to the tail and obtain $t_1, t_2, \dots t_n$ — a sequence of timestamps of return count_right_border(node.Left, maxdescriptors, located in Q, then $t_1 < t_2 < \ldots < t_n$ will hold.

count_right_border(node.Right, maRroof. We prove the theorem by the induction on the tree structure. As the induction basis, we will show that the statement holds for the tree root. As the induction step, we will prove that, given that the statement holds for some node pv, the statement holds for v — an arbitrary child of pv. Thus, the statement is guaranteed to hold for each tree node.

- · As requested in Section II-A and as explained in Section II-D, the root queue provides timestamp allocation mechanism with the following guarantees: if descriptor of operation A is inserted to the root queue before descriptor of operation B, then timestamp(A) < timestamp(B) holds. Thus, the induction base is proven.
- Consider non-root node v and its parent pv. According to the induction assumption, the statement holds for pv. Thus, at pv queue descriptor timestamps form a monotonically increasing sequence: $t_1 < t_2 < ... < t_n$. Consider descriptors D_i and D_j (Fig. 15), such that:
 - Both D_i and D_j should continue their execution at v;
 - timestamp(D_i) = t_i ;
 - timestamp(D_i) = t_i ;
 - D_i is located closer to the head of pv queue than D_i (therefore, D_i was inserted to pv queue prior to D_i) thus according to the induction assumption $t_i < t_j$.

Fig. 15: Descriptor D_i is located closed to the head of pv queue than D_i, both D_i and D_i will continue their execution in v subtree



In that case, according to the algorithm, the execution of D_i in pv cannot begin until the the execution of D_i in pv is finished. Since the execution of D_i in pv includes inserting D_i into v queue, the execution of D_i in pv cannot begin until D_i is inserted into v queue. Thus, D_i

is inserted into v queue prior to $D_{\, j},$ thus the timestamps increasing property holds for v.