Ministry of Science and Higher Education of the Russian Federation
ITMO UNIVERSITY

GRADUATION THESIS

DEVELOPMENT OF CONCURRENT HIERARCHICAL DATA STRUCTURES WITH EFFICIENT RANGE QUERIES

Author: Kokorin Ilya Vsevolodovich _____

Subject area: 01.04.02 Applied mathematics and informatics

Degree level: Master

Thesis supervisor: Aksenov V.E., PhD _____

Saint Petersburg, 2022

Student Kokorin Ilya Vsevolodovich
Group M42381c Faculty of IT&P

Subject area, program/major
Technologies of software engineering

Thesis received ‘‘_____’’ _____ 20___

Originality of thesis _____%

Thesis completed with grade _____

Date of defense ‘‘_____’’ _____ 20___

Secretary of State Exam Commission Khlopotov M.V. _____

Number of pages _____

Number of supplementary materials/Blueprints _____

CONTENTS

INTRODUCTION

The purpose of this work is to develop an algorithm for executing lock-free range queries on Hierarchical Data Structures (hereinafter, HDS) in an asymptotically optimal manner.

The following goals are to be achieved in order to complete the work:

1. Get familiar with existing methods of implementing concurrent range queries on HDS and outline the drawbacks of these methods.
2. Develop a general algorithm for executing asymptotically optimal lock-free range queries on HDS.
3. Apply the developed algorithm to implement binary search tree, supporting asymptotically optimal `count` range query.
4. Develop a method to test such HDS implementations for correctness (i.e., linearizability) in polynomial time and test our binary search tree implementation for correctness.
5. Show how to apply the developed algorithm to a broad class of HDS and a wide variety of range queries.

The importance of the topic is justified by the widespread applicability of range queries in modern Database Management Systems (DBMS) and other data storage and processing systems. However, existing methods that support range queries suffer from one or more of the following drawbacks:

— Lack of progress guarantees. Such implementations are usually lock-based, and do not satisfy lock-freedom or obstruction-freedom.

— Asymptotic sub-optimality. Many range queries (especially, the aggregating ones) can be executed in sub-linear (e.g. logarithmic) time in the sequential implementation. One example of such a query is `count(Set, min, max) = | { x ∈ Set : min ⩽ x ⩽ max } |` — the number of elements of the dataset, located in the range `[min; max]`. Using a relatively simple technique (that is described in our work later), the `count` query can be executed in $O(\log N)$ time on a binary search tree where N is the size of the tree. However, many concurrent range query algorithms can only execute such queries in $O(|ANS|)$ time where $|ANS|$ is the number of elements in the range. Thus, being $O(N)$ in the worst case.

— Lack of parallelism. Many data structures, despite allowing range queries, that are both lock-free and asymptotically efficient, allow only one modifying

(e.g. `insert` or `remove`,) operation at a time to be completed successfully, thus, effectively making the data structure sequential.

The innovative nature of this work is justified by the absence of an algorithm, that can execute concurrent range queries on HDS, while not suffering from any of the aforementioned drawbacks. This work presents such an algorithm.

The practical significance of this work is justified by the possibility to apply the developed algorithm in database management systems. Such DBMS will be able to execute range queries in a more optimal way (e.g., with higher throughput and lower latency), than without the proposed algorithm.

This work is structured the following way:

— The first chapter contains the review of the subject area. We present the notion of Hierarchical Data Structures, algorithms for sequential range queries on HDS, concurrent correctness criteria, progress guarantees in a concurrent environment, and possible applications of range queries in database management systems. Also, we consider known algorithms for concurrent range queries, and show that all of them suffer from significant drawbacks.

— The second chapter contains general description of the concurrent range query execution algorithm. In this chapter we describe the algorithm in general terms, without applying it to any specific data structure. In particular, this chapter contains the description of the heart of the algorithm — the queue propagation framework.

— The third chapter contains description of the algorithm in application to one particular data structure — binary search tree with `count(min, max)` operation.

— The forth chapter briefly describes the applicability of our algorithm to other HDS. In this chapter, we do not describe the application of our algorithm to each considered data structure in details. Instead, we briefly describe the HDS and the range query, the algorithm has to work with, and present an idea of how this range query can be efficiently implemented.

— The fifth chapter presents practical result and discusses future work. In particular, this chapter contains description of the method, that can be used to verify executions of the algorithm for correctness in polynomial time. We

apply this method to the binary search tree, described in Chapter 3, and show that our implementation passes all tests.

CHAPTER 1. REVIEW OF THE SUBJECT AREA

1.1. Hierarchical Data Structures

We start with a notion of a Hierarchical Data Structure.

Definition 1. *Hierarchical Data Structure* (henceforth HDS), or a *tree*, is a data structure, that consists of a set of *nodes*. Each node can be connected to many *child* nodes, but can have no more than one *parent* node. All nodes, except for the one, called the *root* node, have exactly one parent. Root node does not have a parent. Moreover, HDS must not contain loops.

Figure 1 presents an example of a tree. There are: 1) node `r` is the root node, 2) nodes `a` and `b` are children of node `r`, and 3) node `r` is a parent of nodes `a` and `b`.

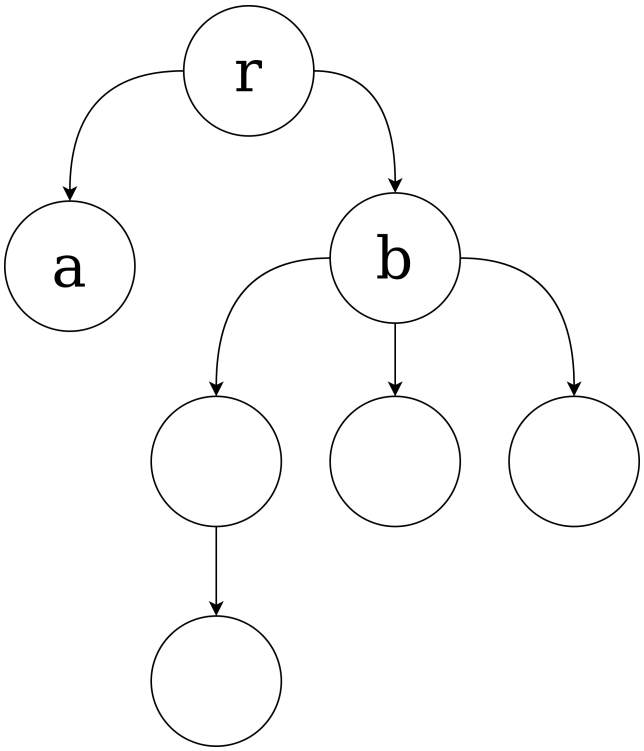
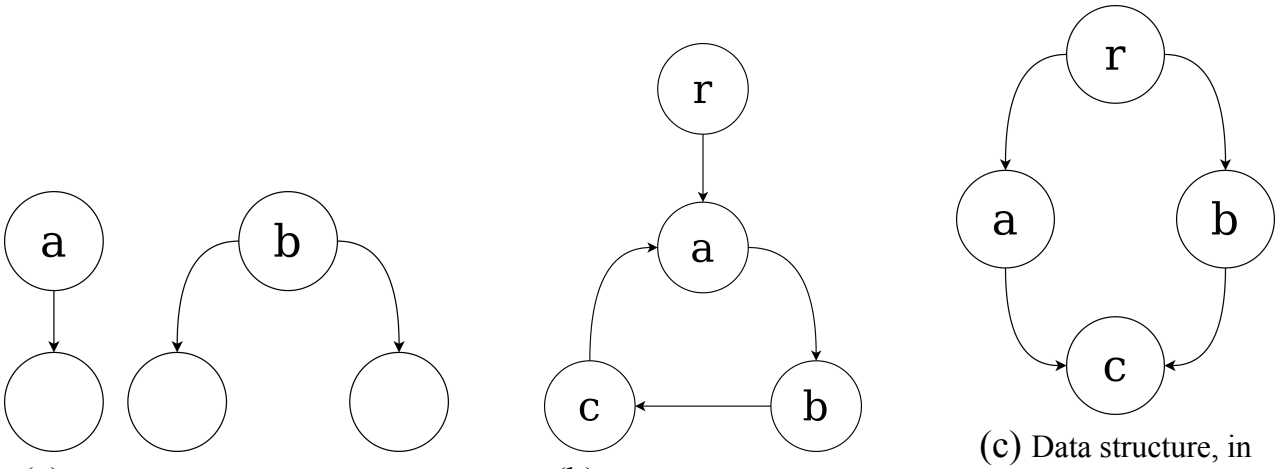


Figure 1 – An example of a tree

Figure 2 contains example of data structures, that are not hierarchical. Figure 2a presents a data structure, that has two nodes without a parent (nodes `a` and `b`), while a hierarchical data structure can contain only one such node. Figure 2b presents a data structure with a loop, consisting of nodes `a`, `b`, and `c`. Figure 2c presents data structure, in which node `c` has two parents simultaneously: nodes `a` and `b`.



(a) Data structure with two root nodes

(b) Data structure with a loop

(c) Data structure, in which a node has two parents

Figure 2 – Examples of data structures, that are not trees

Trees are defined in [43].

Multiple kind of trees have been studied in the literature. Amongst the most useful ones we may outline binary search trees [19] (Fig. 3), quad trees [9] (Fig. 4) and tries [4] (Fig 5). Algorithm, that we will design and implement in that work, will work with all of them, and with various other kind of trees.

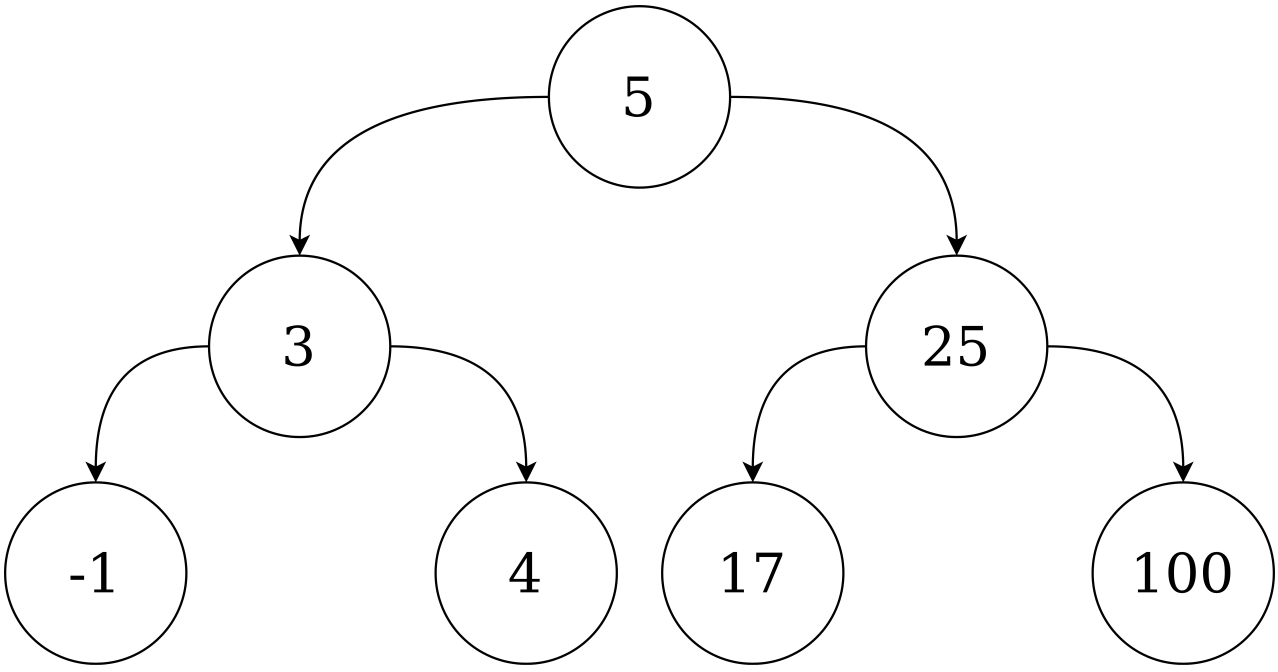


Figure 3 – Example of a binary search tree

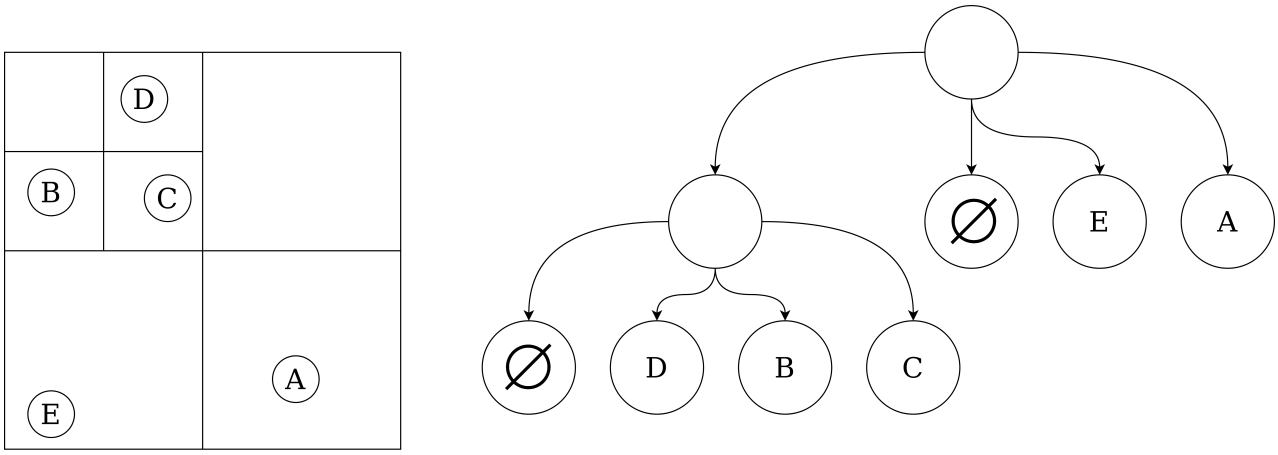


Figure 4 – Example of a quad tree

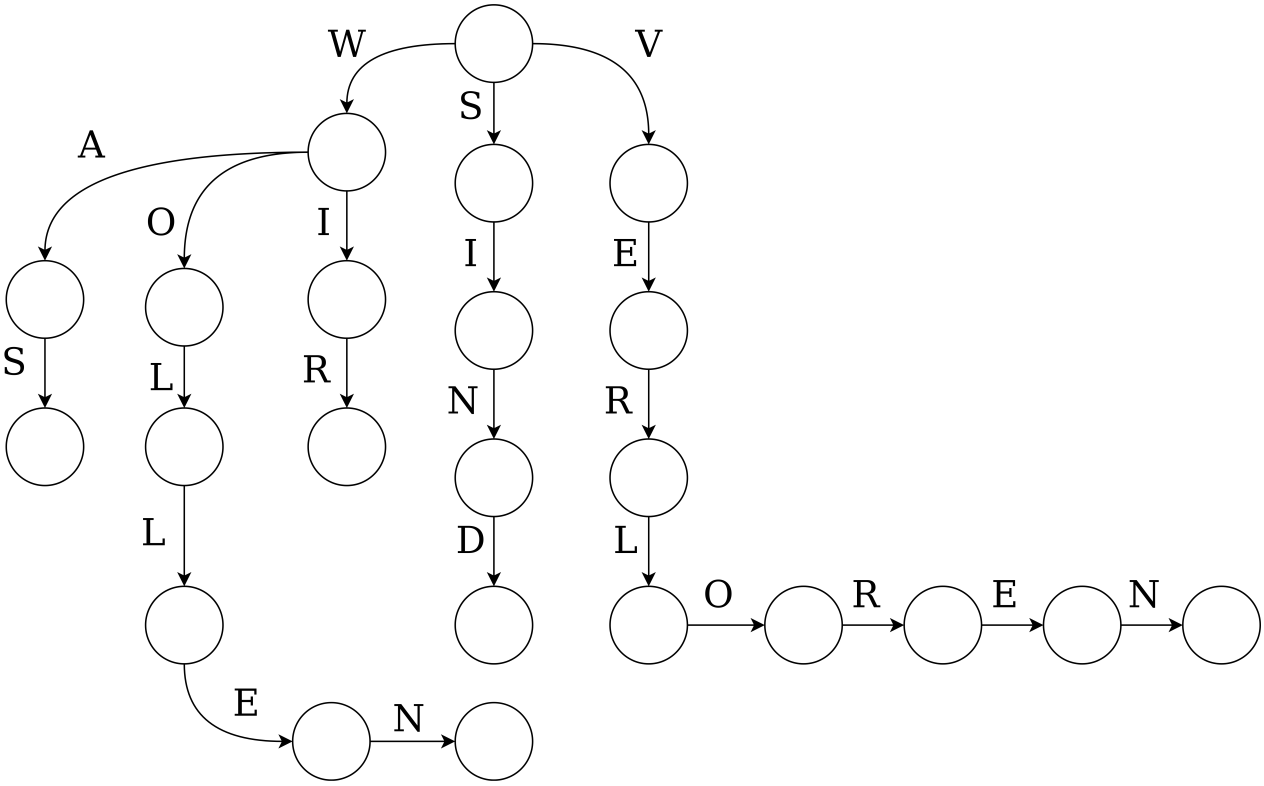


Figure 5 – Example of a trie

1.2. Range queries

Definition 2. Consider a tree, storing multiple data items. 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*.

Consider a sorted set, stored in a binary search tree. In this case, the following queries are scalar:

— `insert(key)` — if `key` exists in the data set, do not modify the set and return `false`. Otherwise, add `key` to the set and return `true`.

— `remove(key)` — if `key` does not exist in the set, do not modify the set and return `false`. Otherwise, remove `key` from the set and return `true`.
— `contains(key)` — return `true` if the set contains `key`, `false`, otherwise.

And the following queries are range queries:
— `count(min, max)` — returns the number of keys from the `[min; max]` interval.
— `collect(min, max)` — returns an array of set keys from the `[min; max]` interval.

1.3. Efficient sequential algorithm for range queries

Many range queries, especially the aggregating ones, can be executed in sublinear (e.g. logarithmic) time. Consider, an example of such range query: `count(Set, min, max) = | { x ∈ Set : min ⩽ x ⩽ 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.

1.3.1. Tree structure

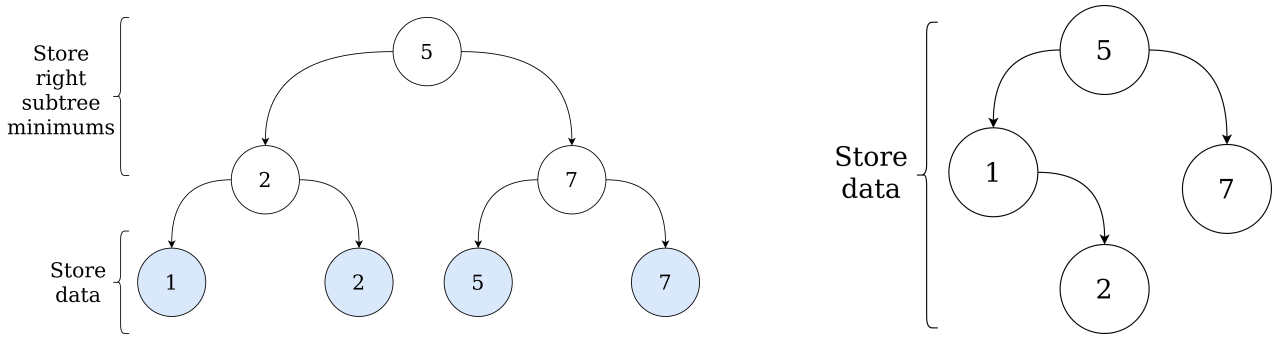
Let us begin with a couple of definitions:

Definition 3. A node is a *leaf* if it has no children.

Definition 4. A node is an *internal* node if it is not a leaf.

Definition 5. *External* binary search tree (Fig. 6a) 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 6. *Internal* binary search tree (Fig. 6b) 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

Figure 6 – 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. 7).

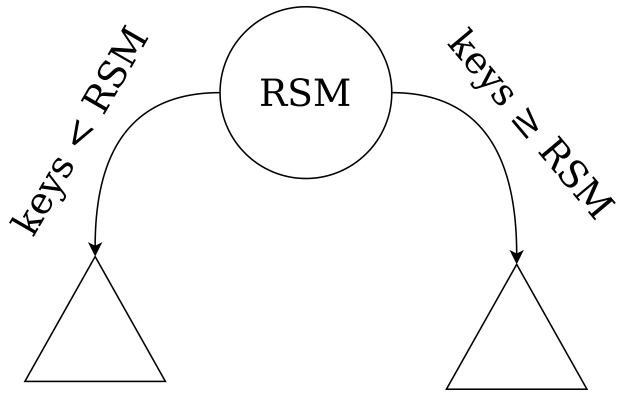


Figure 7 – Using `Right_Subtree_Min` for query routing

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. 8).

— 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. 9).

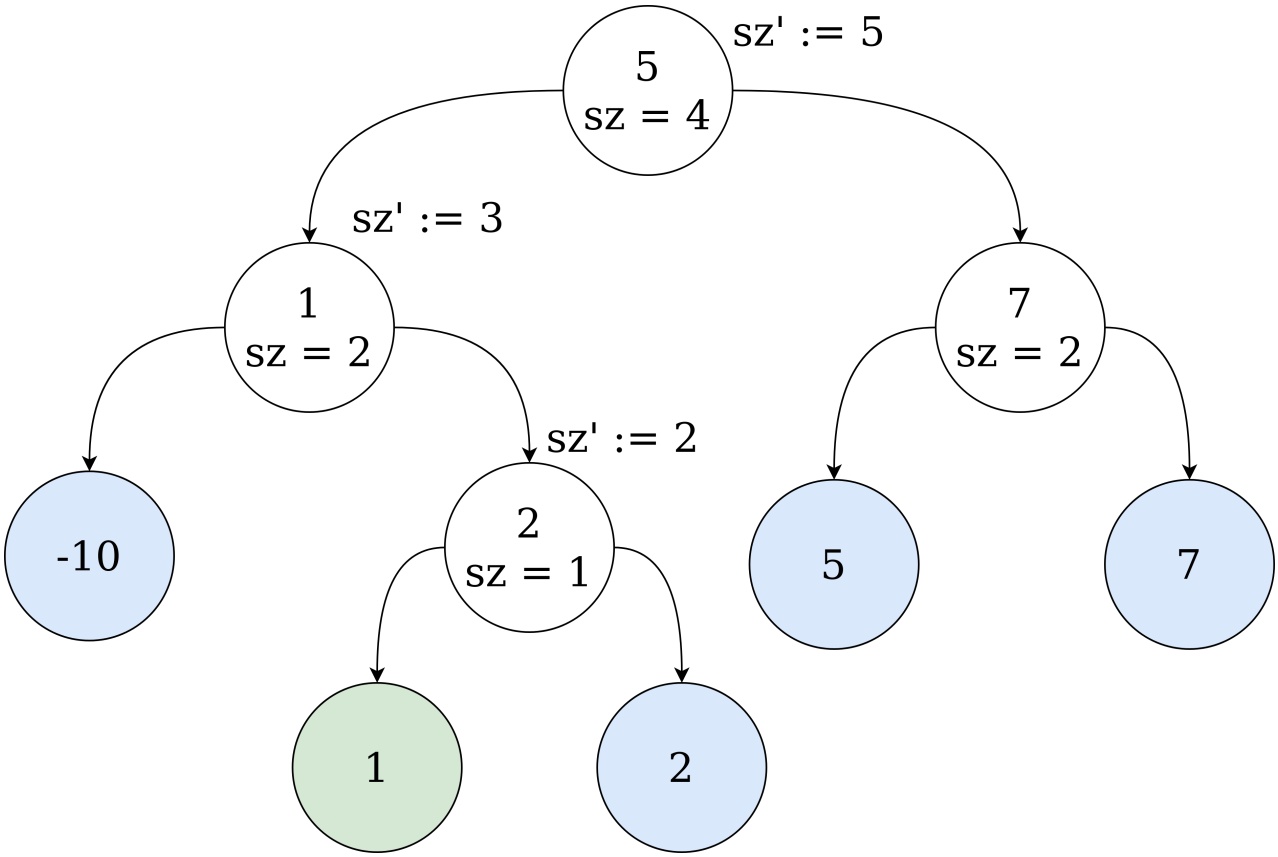


Figure 8 – Maintaining subtree sizes on node insertion

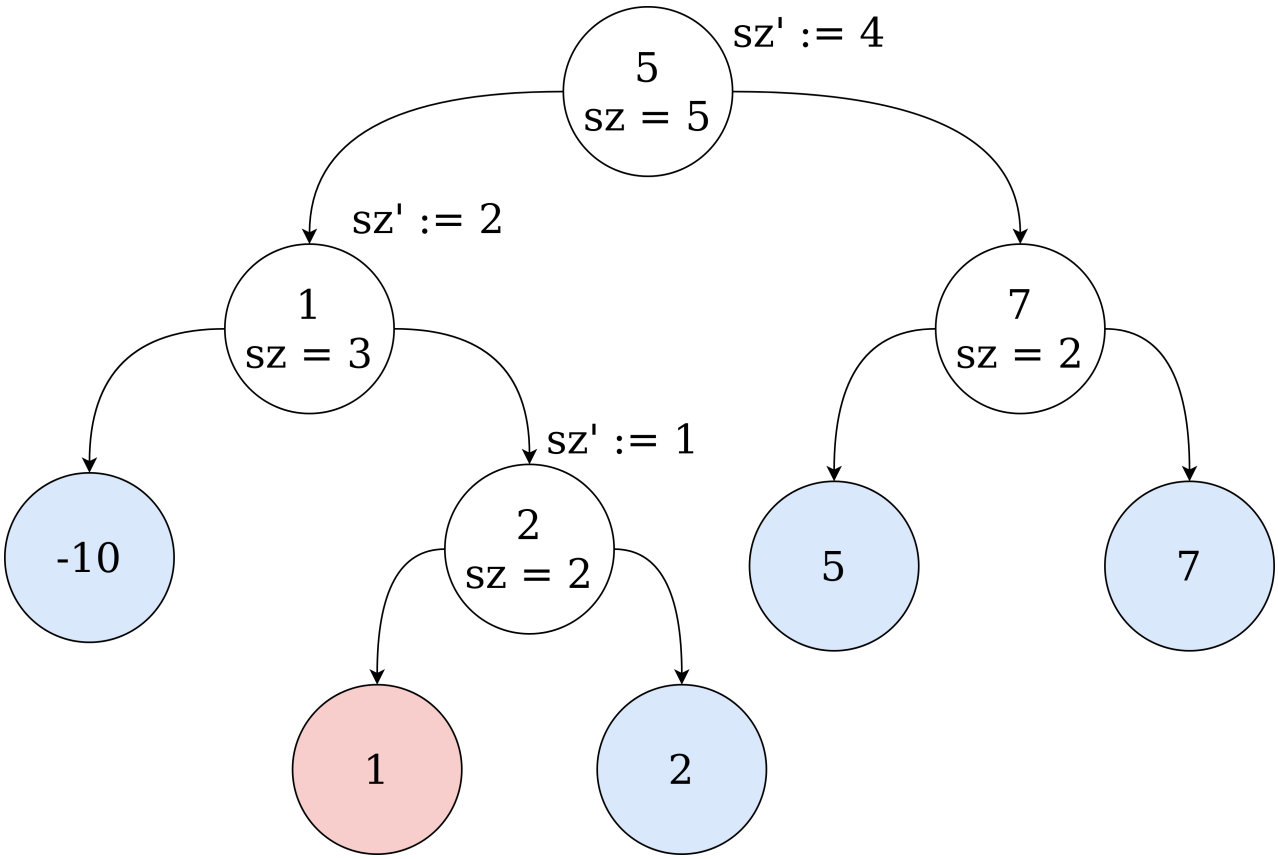


Figure 9 – Maintaining subtree sizes on node removal

Definition 7. 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 4 we shall describe augmentations, required for fast execution of different range queries.

1.3.2. 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 $\texttt{min} \leqslant \texttt{node.Key} \leqslant \texttt{max}$ holds. If so, we return `1`, otherwise, we return `0`.

— If $\texttt{min} \geqslant \texttt{node.Right\_Subtree\_Min}$, then all keys from the left subtree are less than `min` (since for all such keys $\texttt{Key} < \texttt{node.Right\_Subtree\_Min}$ 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 $\texttt{max} < \texttt{node.Right\_Subtree\_Min}$, 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, $\texttt{min} < \texttt{node.Right\_Subtree\_Min} \leqslant \texttt{max}$. 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)`. In that 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 $\leqslant$ `node.Right_Subtree_Min` and that keys in the right subtree are $\geqslant$ `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` $\geqslant$ `min` holds. If so, we return `1`, otherwise, we return `0`.

— If `min` $\geqslant$ `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` $\leqslant$ `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` $\geqslant$ `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 1 [1].

```
1  fun count_both_borders(node, min, max):
2      case node of
3      | EmptyNode →
4          /*
5          EmptyNode is a dummy node that contains neither key nor children.
6          We can use it to represent an empty set, for example
7          */
8          return 0
9      | LeafNode →
10         if min ⩽ node.Key ⩽ max:
11             return 1
12         else:
13             return 0
14     | InnerNode →
15         if min ⩾ node.Right_Subtree_Min:
16             return count_both_borders(mode.Right, min, max)
17         elif max < node.Right_Subtree_Min:
18             return count_both_borders(node.Left, min, max)
19         else:
20             return count_left_border(node.Left, min) +
21                    count_right_border(node.Right, max)
22
23 fun get_size(node):
24     case node of
25     | EmptyNode →
```

[1] In 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;

```
26          return 0
27      | LeafNode →
28          return 1
29      | InnerNode →
30          return node.Size
31
32  fun count_left_border(node, min):
33      case node of
34      | EmptyNode →
35          return 0
36      | LeafNode →
37          if node.Key ⩾ min:
38              return 1
39          else:
40              return 0
41      | InnerNode →
42          if min ⩾ node.Right_Subtree_Min:
43              return count_left_border(node.Right, min)
44          else:
45              return get_size(node.Right) +
46                     count_left_border(node.Left, min)
47
48  fun count_right_border(node, max):
49      case node of
50      | EmptyNode →
51          return 0
52      | LeafNode →
53          if node.Key ⩽ max:
54              return 1
55          else:
56              return 0
57      | InnerNode →
58          if max < node.Right_Subtree_Min:
59              return count_right_border(node.Left, max)
60          else:
61              return get_size(node.Left) +
62                     count_right_border(node.Right, max)
```

Listing 1 – Implementation of the `count` range query

1.3.3. Time complexity analysis

Theorem 8. 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 10).

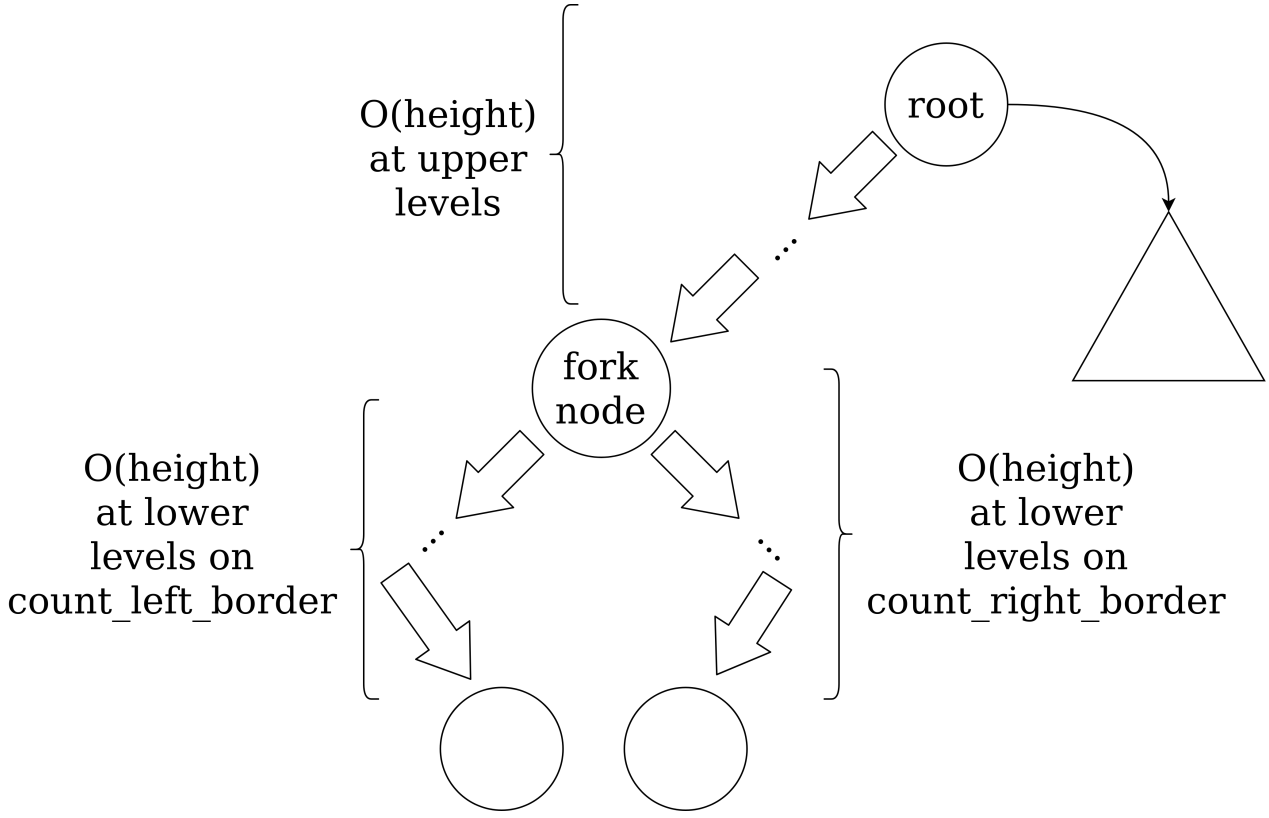


Figure 10 – Time complexity of the `count_both_borders` procedure

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.

1.4. Range queries applications

1.4.1. Spammers identification

Suppose we are developing a database for a messenger. In that case, SQL definition of a table, that will store sent messages might look like this (Listing 2):

```
CREATE TABLE Messages
(
    sender_id INT,
    receiver_id INT,
    send_timestamp TIMESTAMP,
    message_text VARCHAR
);
```

Listing 2 – SQL definition of the `Messages` table

Suppose also, that we want to identify spammers, given that database. One possible approach is to find users, that send a lot of messages during a short time period. To implement that approach, we should be able to answer a certain query: how many messages has some particular user sent during some particular time period? When written in SQL, that query might look like this (Listing 3):

```
SELECT COUNT(*)
FROM Messages
WHERE sender_id = :s_id AND
      send_timestamp BETWEEN :start_ts AND :start_ts + :time_delta
```

Listing 3 – SQL query for getting the number of messages, sent by a particular user during a particular time period

How can such queries be answered fast? We can build an ordered index on fields `(sender_id, send_timestamp)` (Fig. 11).

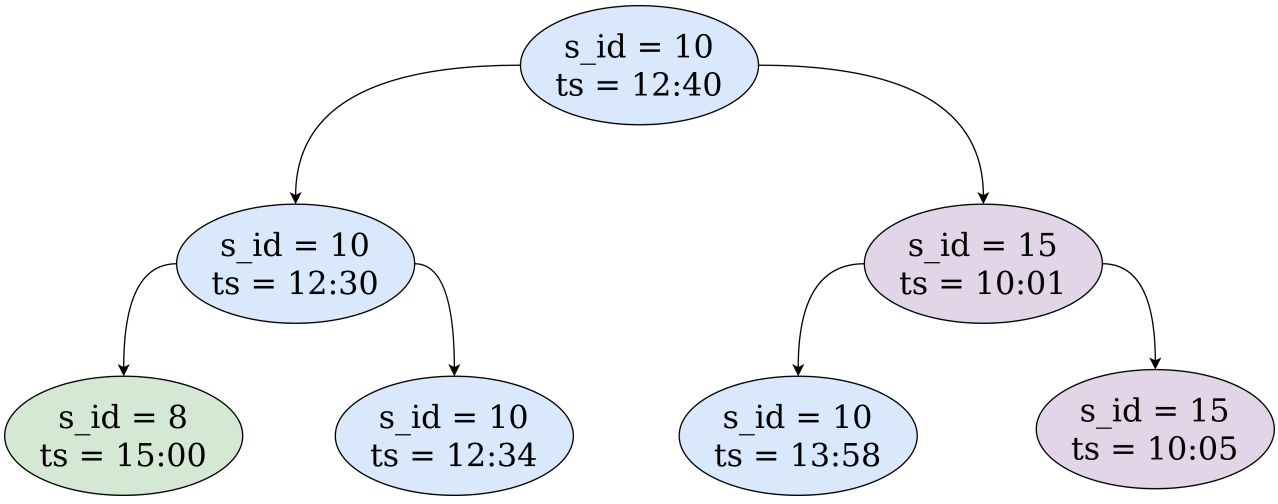


Figure 11 – Binary search tree (BST) as an ordered index on fields `(sender_id, send_timestamp)`

In such case, executing such SQL queries can be reduced to executing a range query `count(min = (:s_id, :start_ts), max = (:s_id, :start_ts + :time_delta))` on a binary search tree, serving as an ordered

index. In that case, the faster the BST can process such queries, the faster we can identify spammers. Thus, we need our index implementation to process such range queries in an asymptotically optimal manner.

1.4.2. Traffic jams identification

Suppose we are building an application, that should identify traffic jams based on a car location information and warn drivers to change their route, if it is expected to go through a traffic jam. In such case, a crucial part of our application would be an algorithm, that can identify traffic jams very fast.

But what is a traffic jam? To a first approximation it is a small area, that contains an enormous number of cars. Thus, to identify traffic jams, we must be able to answer a certain query: how many cars are located in the specified area? Assuming a car is a point on a plane and a search area is a rectangle (Fig. 12), we can solve this task using range queries on R-trees [14] or k-d trees [9]. Therefore, once again, we need our R-tree or k-d tree implementation to process such range queries in an asymptotically optimal manner.

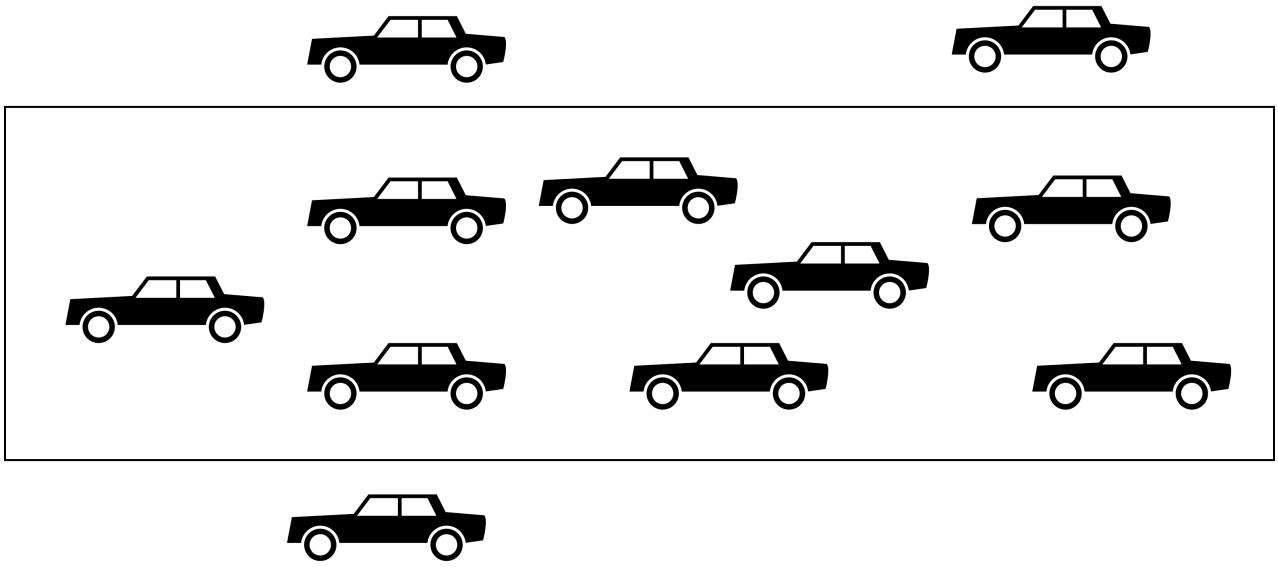


Figure 12 – Traffic jams identification problem

Thus, as can be seen, in both above-described cases the quality of our application directly depends on our ability to perform efficient range queries on trees.

1.5. Execution model

We consider the standard concurrent system model with a set of n processes [2] $\{p_i\}_{i=1}^{n}$ that work asynchronously and each of which executes its own sequence of operations.

Process communicate with each other by executing operations on objects, located in the shared memory. Basic shared objects (called *registers*) provide atomic read/write operations. Moreover, they can provide atomic read-modify-write operations, like compare-and-swap [38], fetch-and-add [40], test-and-set [42].

Amongst all read-modify-write operations, the most relevant for us is compare-and-swap (or compare-and-set, or CAS). This operation has three arguments: register, the expected value and the new value. CAS atomically checks, whether the value of the register equals to the expected value. If so, CAS sets the value of the register equal to the new value and returns true. Otherwise, it lefts the register value unmodified and returns false. The CAS operation can be specified in the following pseudocode (Listing 4):

```
1 fun cas(Register, expected_value, new_value):
2     atomically:
3         cur_value := Register
4         if cur_value = expected_value:
5             Register ← new_value
6             return true
7         else:
8             return false
```

Listing 4 – Pseudocode for the CAS operation

We can use basic shared objects to build more complex shared objects, representing different concurrent data structures. For example, we can use a collection of read/write/CAS registers to build a concurrent queue [27] or stack [35].

Moreover, each process p_i has access to an arbitrary set of local objects, on which only p_i can execute operations. If an object Obj is local to process p_i, only the process p_i has access to it and can execute operations on Obj (Fig. 13).

[2] Hereinafter terms ‘‘process’’ and ‘‘thread’’ are used interchangeably: in the context of this work they mean the same.

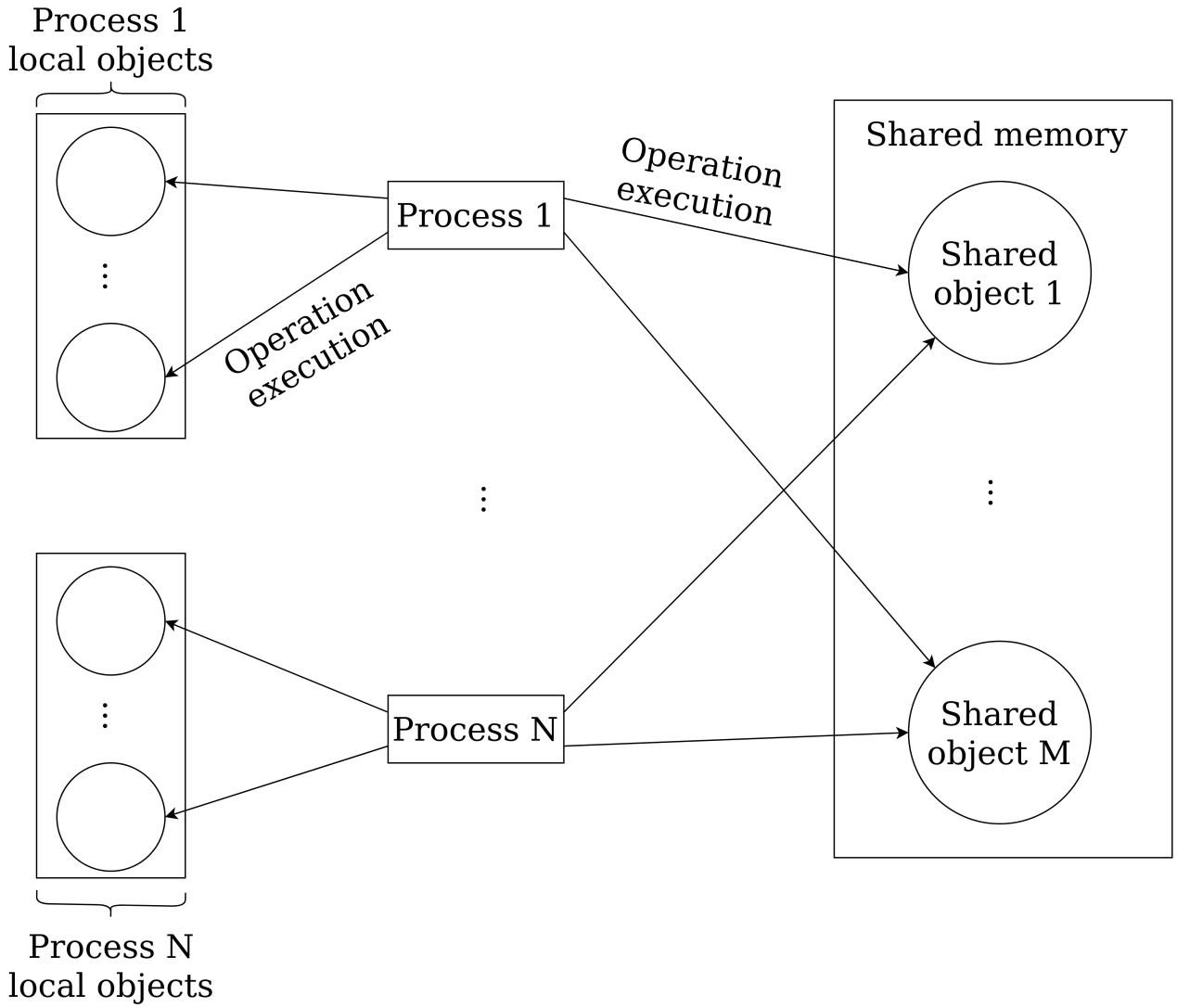


Figure 13 – Concurrent system model

We assume that the processes work asynchronously, each with its own speed, without synchronizing neither their pace nor the number of operations executed. Moreover, each process can be suspended indefinitely by the underlying (e.g., OS) scheduler.

1.6. Concurrent correctness criteria

Consider an execution of a program, where two operations: `Set.insert(5)` and `Set.contains(5)` are performed concurrently. Suppose the former request returns `true` (i.e., `5` was inserted to the set) and the latter request returns `false` (i.e. `5` did not exist in the set). How can one tell, whether the code works correctly or not?

More formally, consider a concurrent implementation of a data type T. We want to have a correctness criteria, that will tell us, whether this concurrent implementation is correct (in some sense) or not.

Before we can get to the formulation of correctness criteria, we should give additional definitions.

Definition 9. A *low-level history* (or an *execution*) is a finite or infinite sequence of primitive steps: invocations and responses of high-level operations, invocations and responses of primitives on the shared registers (reads, writes, etc.). We assume that executions are *well-formed*: no process invokes a new primitive, or high-level operation before the previous primitive, or a high-level operation, respectively, returns or takes steps outside its operation's interval.

Definition 10. A *high-level history* (or simply a *history*) of execution α on high-level object O is the subsequence of α consisting of all invocations and responses of operations on O.

Definition 11. Two high-level histories H and L are considered *equivalent* iff three conditions are met:

— They are defined on the same object O;
— H and L consists of the same set of operations;
— All operations in L have the same input and the same output as the corresponding operations in H;

Definition 12. High-level history L is said to be *sequential* if for any two operations $o_1, o_2 \in L$ either o_1 precedes o_2 or vice-versa, i.e. there are no concurrent operations in L.

Multiple correctness conditions for concurrent executions exist. We outline some of them, from the least to the most strict.

— *Serializability*, described in [31, 37]. History H is said to be serializable iff there exists sequential history L, equivalent to H. As can be seen from the definition, operations in H can be reordered arbitrary, we only care about the equivalence of operation results.

  A concurrent implementation of a data type is *serializable* iff each of its possible histories is *serializable*.

— *Sequential consistency*, described in [23]. To reason about the notion of sequential consistency, we should consider a set of processes $\{p_i\}_{i=1}^{n}$, executing operations from history H.

  History H is said to be sequentially consistent iff there exists a sequential history L, equivalent to the history H. Moreover, history L should satisfy

one additional requirement: if operations o_1 and o_2 were executed by the same process p_i in H and o_1 was executed before o_2 [3], o_1 should precede o_2 in L.
As follows from the definition, each sequentially consistent history is serializable, since serializability only requires that an equivalent sequential history L exists, while sequential consistency places additional requirements on L.
A concurrent implementation of a data type is *sequentially consistent* iff each of its possible histories is *sequentially consistent*.

— *Linearizability*, described in [18]. To reason about the notion of linearizability, we should consider $\rightarrow_H$ (pronounced *happens-before*): partial order on operations from a concurrent history H. We say, that $o_1 \rightarrow_H o_2 \mid o_1, o_2, \in H$ (pronounced o_1 *happens-before* o_2) if o_1 is completed before o_2 begins. Such operations can be causally related, i.e., o_1 can be a cause of o_2. If operations o_1 and o_2 are executed by the same process p_i and o_1 precedes o_2 in p_i execution, $o_1 \rightarrow_H o_2$. The *happens-before* relation is discussed more formally in [25].
History H is said to be linearizable iff there exists a sequential history L, equivalent to the history H. Moreover, execution L should satisfy one additional requirement: if $o_1 \rightarrow_H o_2$, o_1 should precede o_2 in L. Thus, $o_1 \rightarrow_H o_2 \Rightarrow o_1 \rightarrow_L o_2$ should hold.
As follows from the definition, each linearizable history is also sequentially consistent (and, thus, serializable), since sequential consistency only requires that ordering of operations from a single process should be preserved in L, while linearizability requires, that ordering of all causally related operations (including operations from a single process) should be preserved in L.
A concurrent implementation of a data type is *linearizable* iff each of its possible histories is *linearizable*.

1.7. Progress guarantees

Consider a simple lock-based concurrent algorithm (Listing 5):

```
1 fun do_something_concurrent():
2     Mutex.lock()
3     do_something()
4     Mutex.unlock()
```

Listing 5 – An example of a simple lock-based algorithm

Consider the following sequence of actions:

[3] Note, that for each pair (o_1, o_2) of operations, executed by a single process p_i, either o_1 precedes o_2 or vise-versa, since a single process always has sequential behaviour, even in a concurrent environment

1. Processes `P` and `R` are willing to execute the operation `do_something_concurrent` at the same time.
2. Process `P` manages to acquire the mutex first, process `R` has to wait for the mutex to be released.
3. Process `P` is suspended by the operating system.

In that case, neither `P` nor `R` is able to execute the operation: `P` is suspended by the OS and `R` has to wait until `P` finishes the operation execution and releases the mutex (which may take arbitrary long, given that `P` is suspended). Therefore, the whole system does not achieve any progress at all.

To prevent such situations from happening, we should design algorithms so that they guarantee progress even in the presence of scheduler-initiated suspends and arbitrary processes speed.

Multiple progress guarantees have been studied and described in the literature. We outline some of them from the most relaxed one to the most strict.

— *Obstruction-freedom*, described in [17]. This progress guarantee requires, given that all system processes $\{\mathtt{p_i}\}_{i=1,i\neq L}^{n}$, except for the one — $\mathtt{p_L}$, are suspended, $\mathtt{p_L}$ can finish its execution within a bounded number of steps.

Note, that the lock-based algorithm, described in the beginning of the section, is not obstruction-free. Indeed, even when all processes except `R` (thus, only the process `P`) are suspended, `R` cannot finish its execution within a bounded number of steps, because it is waiting for the mutex to be released.

— *Lock-freedom*, described, e.g. in [34]. This progress guarantee requires that at least one non-suspended process should finish its execution within a bounded number of steps.

Note that each lock-free algorithm is also obstruction-free. Indeed, suppose that all processes, except for $\mathtt{p_L}$ are suspended. Since the algorithm is lock-free, at least one non-suspended process should finish its execution within a bounded number of steps. After suspending all but one processes, we have only one non-suspended process — $\mathtt{p_L}$. Therefore, $\mathtt{p_L}$ will finish its execution within a bounded number of steps, therefore, the algorithm is also obstruction-free.

— *Wait-freedom*, described in [16]. This progress guarantee requires, that all non-suspended processes should finish their execution within a bounded number of steps.

Indeed, each wait-free algorithm is also lock-free, since lock-freedom requires at least one non-suspended process to finish its execution within a bounded number of steps, while wait-freedom guarantees, that all non-suspended processes will do so.

1.8. Existing solutions

1.8.1. 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*, or *mutex* (Fig. 14) to guarantee mutual exclusion [24] to the protected data structure. However, such construction is not lock-free (it is not even obstruction-free) and suffers from the stagnation, as described in Section 1.7. Moreover, since a lock allows only one process to work with the data structure at a time, the throughput of the resulting construction will be very low and the resulting construction will not scale.

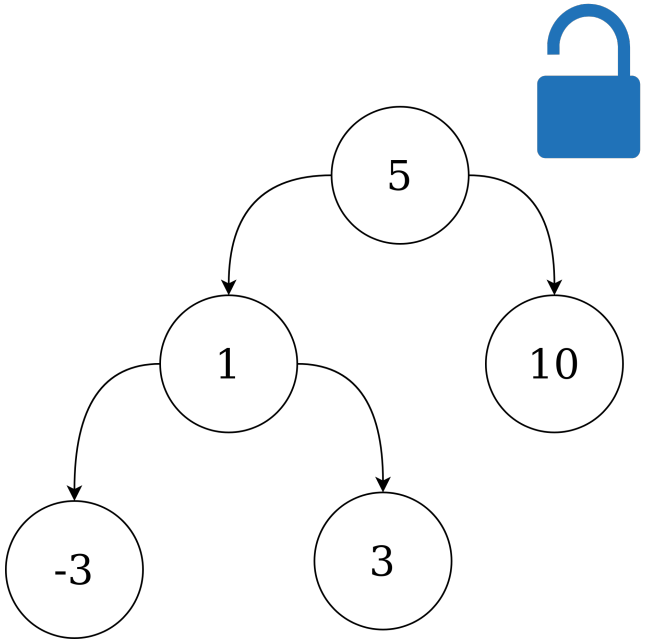


Figure 14 – Sequential data structure, protected with a lock

1.8.2. Linear-time solutions

A number of papers [2, 5] address the issue of executing lock-free 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, 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 length of the resulting list. Thus, for wide ranges, the query is executed in $O(N)$ time, since for wide ranges the result contains almost all the keys from the tree.

Thus, all range queries, implemented on top of the `collect` query, have $O(N)$ time complexity. However, this implementation is not asymptotically efficient: e,g, the `count` query can be executed in $O(\log N)$ time in a sequential environment instead of $O(N)$, as was shown in Section 1.3.

Therefore, despite being lock free, this method does not guarantee time efficiency, and thus cannot be used in environments, where low request latency is required.

1.8.3. Solutions based on the Universal Construction

These solutions are based on persistent data structures [41]. Each modifying operation (e.g. `insert` or `remove`) creates a new version of the data structure without modifying the existing one. In order to reduce time and memory consumption, for a lot of persistent data structures the new version shares most of its nodes with the old version.

For example, persistent trees can be implemented using *path-copying* method [26]. This method merely copies each node on the path from the modified (e.g., inserted or removed) node to the root (Fig. 15), achieving $O(\log N)$ copied nodes in balances trees.

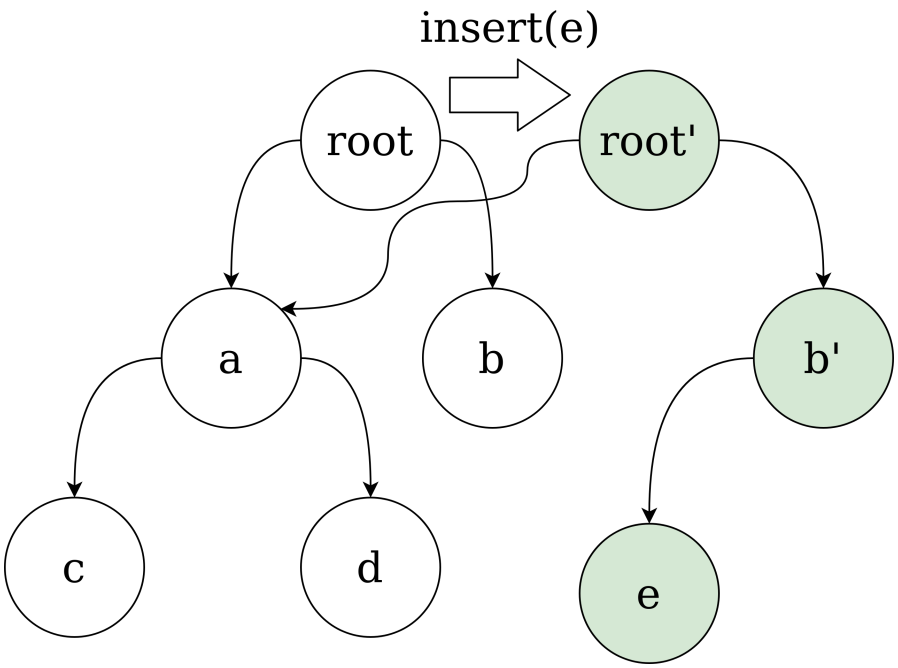


Figure 15 – An example of a persistent tree with path-copying. `insert` operation creates new version of the data structure instead of modifying the existing one, the new version shares most of its nodes with the old version

The basic idea of the Universal Construction, proposed first by Herlihy [16], is to store pointer to the root of the current version of the persistent data structure in the `read/CAS` register. Read-only queries (e.g. `contains` or `count`) on such data structures can be implemented very easily, no matter how complex are they: we just read the pointer to the latest version of the data structure and execute the query on it. Since the data structure is persistent, other processes cannot modify the fetched version, thus, it is completely safe to execute arbitrary read-only logic on the fetched version. For example, the `count` query can be implemented the following way (Listing 6):

```
1 fun count(Set, min, max):
2     /*
3     Other read-only operations (e.g. contains) can be
4     implemented the same way
5     */
6     cur_root := Set.Root_Pointer
7     result := sequential_count(cur_root, min, max)
8     return result
```

Listing 6 – Universal Construction-based implementation of a `count` query

Update queries (e.g. `insert` or `remove`) are a bit more complex and have to go through the following steps:

1. Fetch `cur_root` — the root of the current version of the data structure by reading the root pointer.
2. Obtain `new_root` — the root of the new version by executing the modification operation on the current version. Since the data is persistent, the current version is left unmodified.
3. Use `CAS(&Root_Pointer, cur_root, new_root)` to atomically change the current version to the new version. If the `CAS` returns `true`, it means that we have successfully applied the modifying operation. Otherwise (if the `CAS` returns `false`), we conclude that some other process has already changed the version, performing its modification operation. In that case, we retry the whole operation from the very beginning, i.e., from step (1).

For example, `insert` operation can be implemented the following way (Listing 7):

```
1  fun insert(Set, key):
2      /*
3      Other modification operations (e.g. remove) can be
4      implemented the same way
5      */
6      while true:
7          cur_root := Set.Root_Pointer
8          new_root := persistent_insert(cur_root, key)
9          if CAS(&Set.Root_Pointer, cur_root, new_root):
10             return
11         /* Otherwise, retry the whole operation from the very beginning */
```

Listing 7 – Universal Construction-based implementation of the `insert` operation

This solution is lock-free, since each unsuccessful `CAS` indicates that some other process has successfully executed its operation (and changed the root pointer using `CAS`). The Universal Construction can even be implemented with a wait-free progress guarantee, as shown in [16].

However, despite being lock-free (or even wait-free) this solution suffers from a major drawback. Suppose we are executing multiple concurrent modification operations. Only one of them can be finished successfully, while others have to retry (Fig 16). To better understand the drawback, consider the following sequence of actions:

1. Process `P` tries to execute the operation `insert(2)`, fetches the current root pointer (`RP`);

2. Process `Q` tries to execute the operation `remove(5)`, fetches the current root pointer (`RP`);
3. Process `P` obtains the new version of the data structure (root pointer is RP_P), with key `2` inserted to the set;
4. Process `Q` obtains the new version of the data structure (root pointer is RP_Q), with key `5` removed from the set;
5. Process `P` successfully executes `CAS(&Set.Root_Pointer, RP,` RP_P`);`
6. Process `Q` tries to execute `CAS(&Set.Root_Pointer, RP,` RP_Q`)` but fails to do so, because `Set.Root_Pointer` now contains RP_P. Process `Q` has to retry the whole `remove(5)` operation from the beginning;

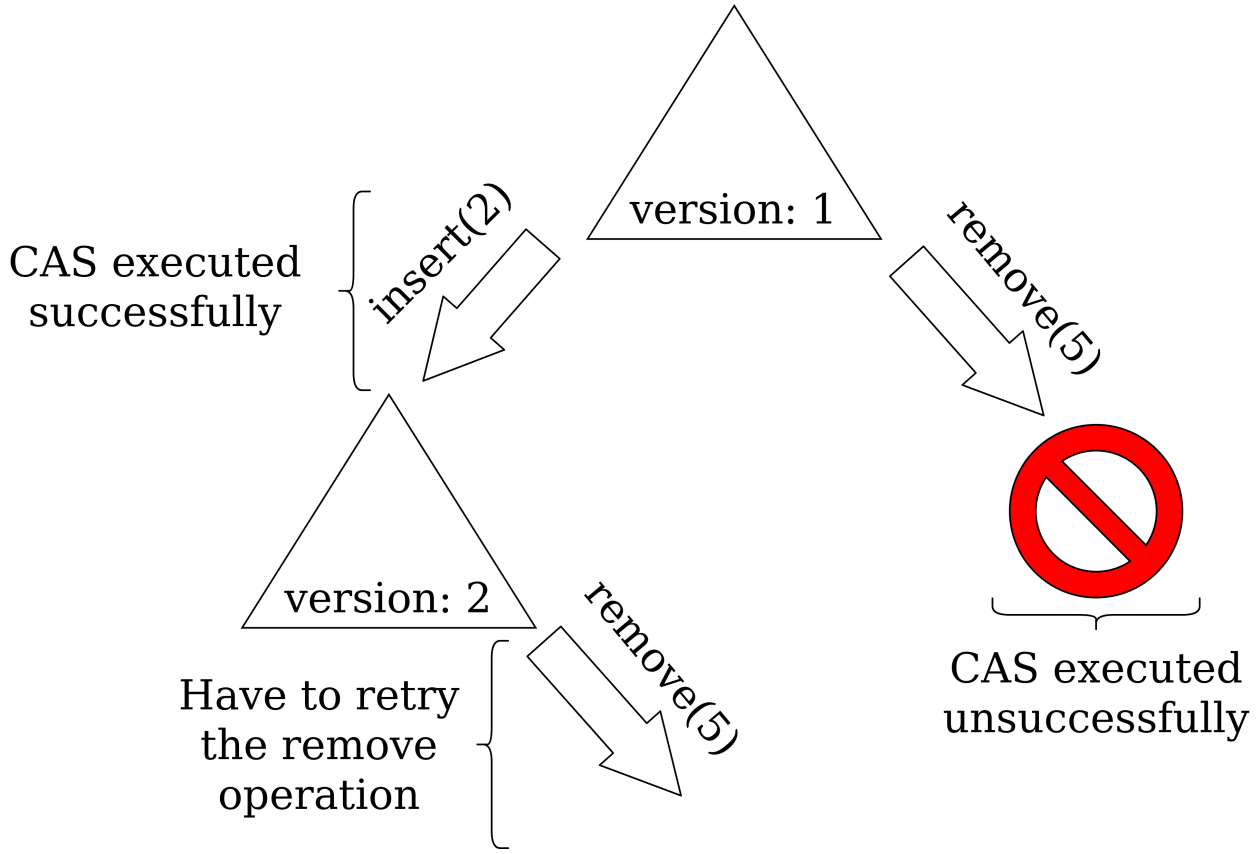


Figure 16 – Only one of multiple concurrent modifying operations can be executed successfully, the others have to retry

Therefore, all modification operation are applied sequentially, one after another, making the data structure effectively sequential for modifications. This greatly reduces the throughput of the overall data structure and dramatically limits the scalability. Thus, Universal Construction-based data structures cannot be used

in situations, when concurrent updates must be processed concurrently, instead of being processed one after another.

1.8.4. Solutions, based on augmented persistent trees

Sun, Ferizovic and Belloch [33] addressed the problem of executing range queries on persistent trees. They proposed *augmenting* each node of a persistent tree with an arbitrary value, that can be used to execute the range query faster (for example, the `count` query can be executed efficiently if each node of the tree is augmented with the size of its subtree).

However, the paper does not propose the method of executing concurrent operations on augmented data structures — only the method to execute a large batch of operations in parallel, e.g., an insertion of a batch of keys to the data structure (or a removal of a batch of keys from the data structure) using *fork-join* parallelism to speed up the execution.

Therefore, in some cases this method simply cannot be used — for example, when updates do not come in large batches. In cases, when there is a large number of concurrent single-value updates instead of a small number of batch updates, following sequentially, one after another, we cannot use augmented persistent trees as-is.

We can use various combining techniques [1, 11] to form large batches of updates from individual concurrent updates (Fig. 17). However, combining techniques increase individual operation latency and thus they are not acceptable in settings, where low operation latency is required.

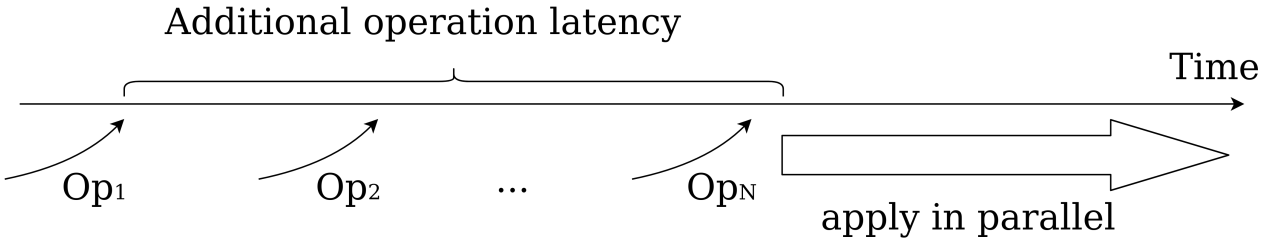


Figure 17 – Forming batches of updates from individual concurrent updates

Conclusions on Chapter 1

In this chapter, we have performed a review of the subject area. We have reviewed the notion of a tree, the notion of range queries, along with the sequential

time-efficient algorithm for execution of the `count` range query. We glanced over some examples of how efficient range queries can be used in modern databases. Also, we reminded the basics of concurrent computing: the notion of concurrency, along with concurrent correctness criteria, and concurrent progress guarantees. Finally, we studied modern solutions to the problem of executing range queries on concurrent trees and made sure, that all actual solutions suffer from various drawbacks, which we ought to overcome in this work.

CHAPTER 2. GENERAL DESCRIPTION OF THE ALGORITHM

2.1. Concurrent solution: the main invariant

The main problem with the sequential algorithm, described in Section 1.3, is that in a concurrent environment it will be incorrect. Indeed, each modifying operation (e.g. `insert` or `remove`) should modify not only the tree structure, but the augmentation values (e.g., subtree sizes) too. Thus, augmentation values may become inconsistent with the tree structure (Fig.18). In that case, the process executing a `count` query is not able to execute it correctly, given such an inconsistent view of the tree.

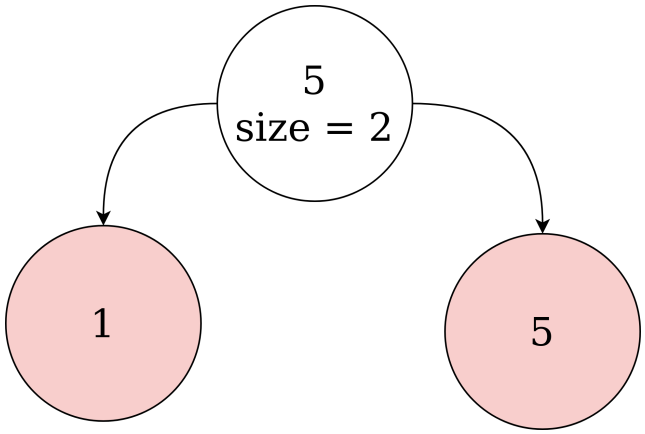


Figure 18 – Tree structure is inconsistent with the augmentation value: while both leaves have already been removed from the tree the subtree size is still two.

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 will enforce a particular execution order by maintaining operation queue in each node.

Consider an arbitrary node `v` and its subtree `vs`. At `v` we maintain *operations queue*, that contains descriptors of operations to be applied to `vs` (Fig. 19). 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.

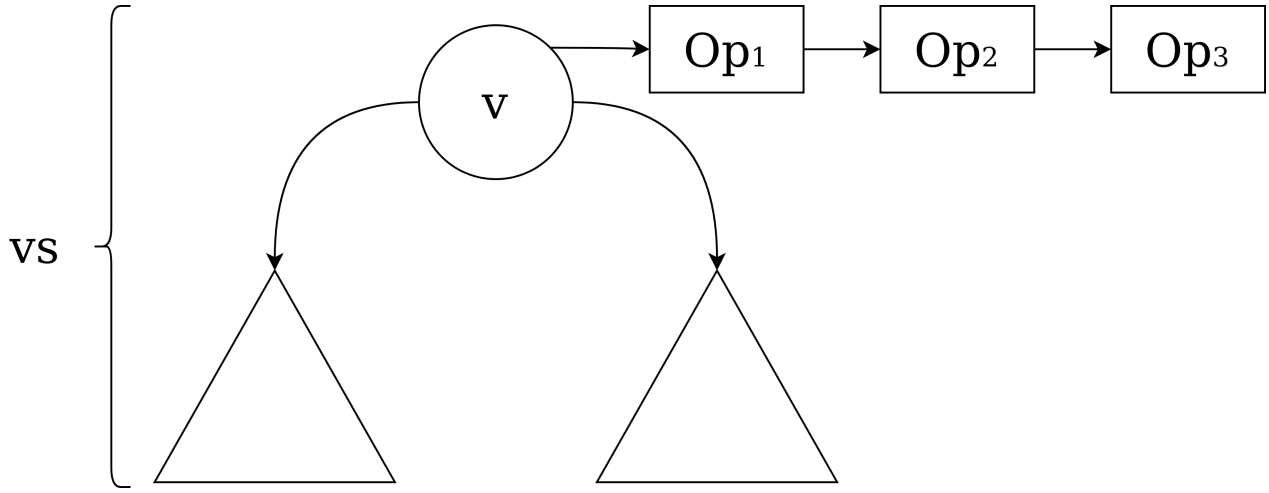


Figure 19 – Node `v` contains operations queue with descriptors of three operations: $\texttt{Op}_1$, $\texttt{Op}_2$ and $\texttt{Op}_3$. These three operations should be applied to `vs` in the order of descriptors in the queue: first $\texttt{Op}_1$, then $\texttt{Op}_2$, and, finally, $\texttt{Op}_3$

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

Thus, we may use the operation queue at the root node to allocate timestamps for operations. Timestamp allocation mechanism should provide the following guarantee: if descriptor of operation `A` was added to the root queue before descriptor of operation `B`, then `timestamp(A) < timestamp(B)` should hold. In Section 2.6.2, we will show how such timestamp allocation mechanism can be implemented. Since, the order, in which operation descriptors are added to the root queue, equals to the linearization order, operations should linearize in the order, determined by their timestamps. For example, operation `A` should precede operation `B` in the sequential execution $\mathbb{L}$ from the linearizability definition (that is described in Section 1.6). Therefore, the following three orders will be the same:

— The linearization order $\mathbb{L}$.
— The *timestamp order*: operation `A` precedes operation `B` in the timestamp order if `timestamp(A) < timestamp(B)`.
— The order, in which operation descriptors were added to the root queue.

As described in Section 2.2, system processes want to examine timestamps of different operations during the operation execution. To allow them do so, we

include the operation timestamp in the operation descriptor and store it in the `descriptor.Timestamp` field.

2.2. Operation execution: overview

At first, we start with unbalanced trees. One possible balancing strategy via subtree rebuilding is discussed in Section 2.7, while studying other concurrent balancing strategies we leave for the future work.

The execution of an operation `Op` by a process `P` (we call process `P` the *initiator* process) begins with inserting the descriptor of `Op` into the root queue and obtaining the operation timestamp. In Section 2.6.2, we describe, how the root queue with lock-free timestamp allocation may be implemented.

After that, the initiator process starts traversing the tree downwards, from the root to the appropriate lower node, i.e. to the node, at which the operation (e.g., insertion of a new data item, or removal/modification of an existing one) should be performed (Fig. 20).

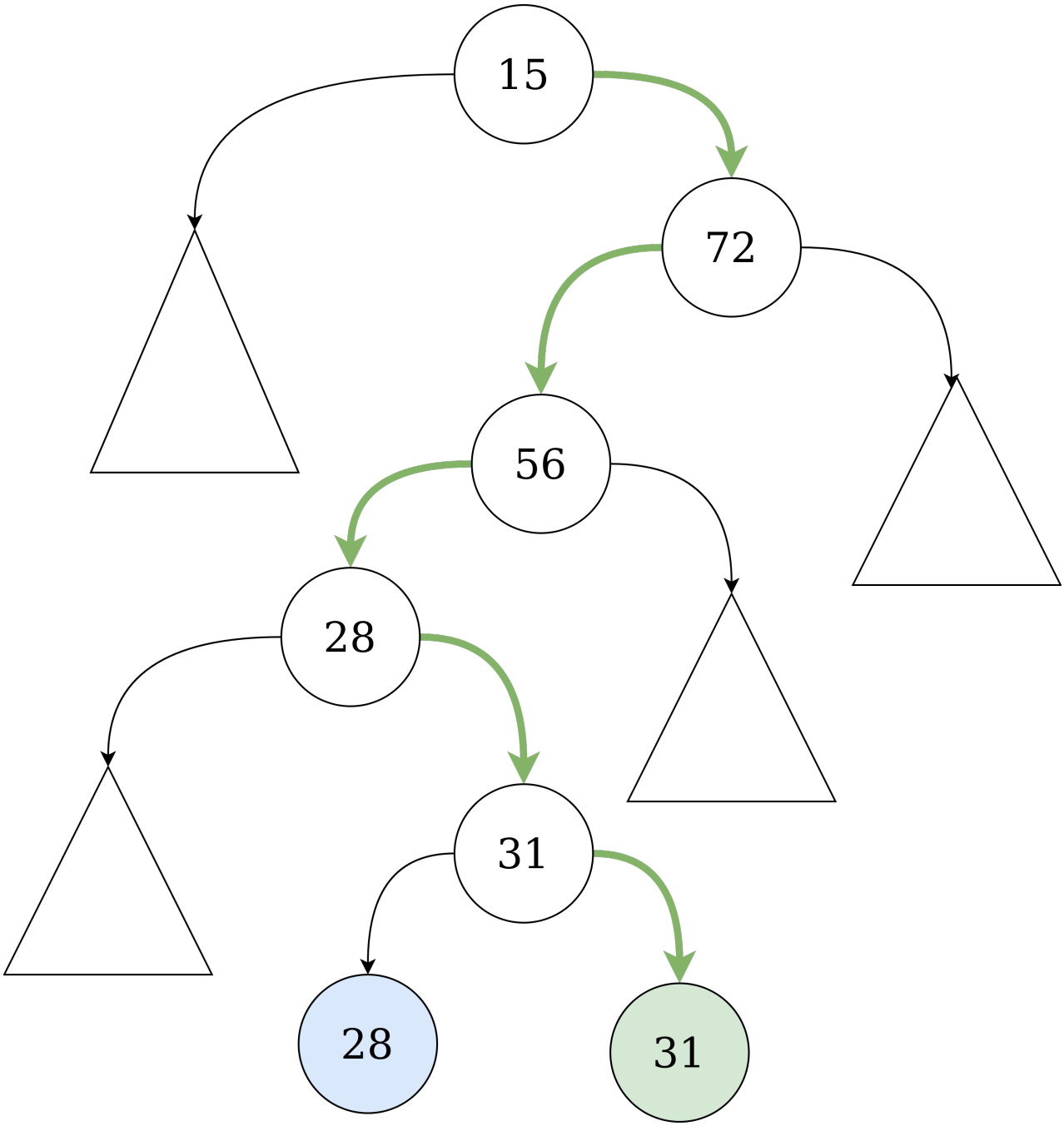


Figure 20 – Execution of operation `insert(31)` in an external binary search tree consists of traversing the tree from the root downwards to the leaf, where key `31` should be inserted.

Definition 13. In each visited node `v` some actions should be performed, in accordance with the meaning of the operation `Op` being executed. For example, size of `v` subtree or pointers to `v` children should be changed during `insert` or `remove` operation. We call the process of performing these actions *execution of operation `Op` in node `v`*.

As stated in Section 2.1, operations should be applied to `v` subtree in the order operation descriptors are inserted to `v` queue. Thus, if the descriptor of `Op` is not

located at the head of v queue the initiator process P will have to wait for the ability to begin executing Op in node v (Fig. 21). The execution of Op in node v cannot begin until execution of all preceding operations in node v is finished.

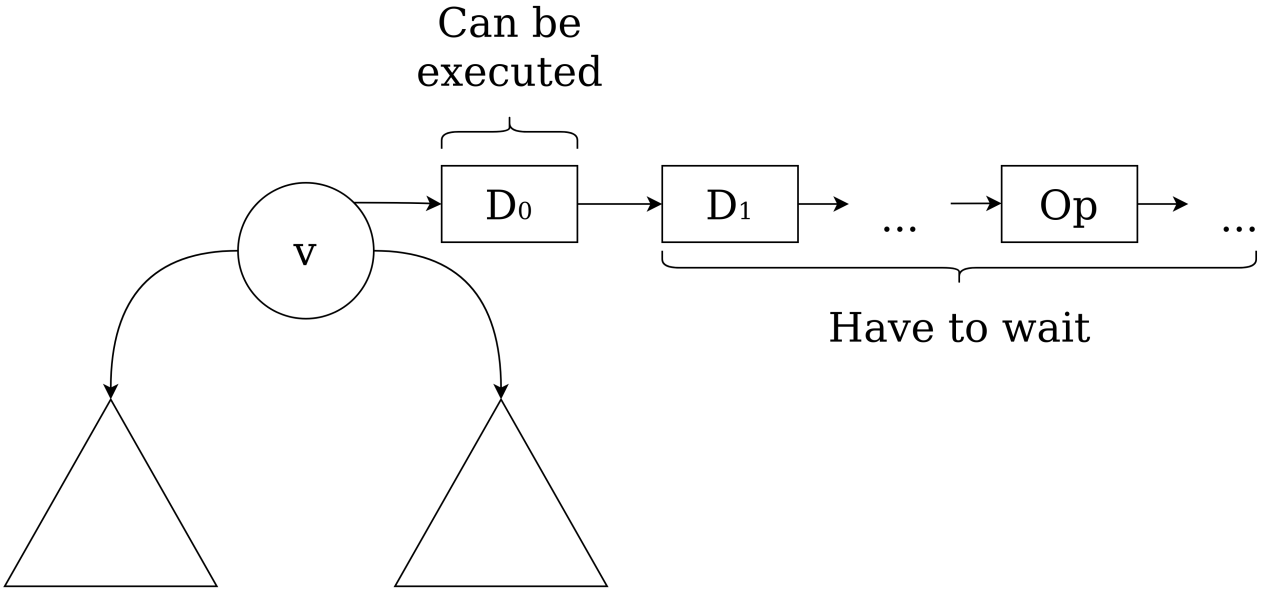


Figure 21 – Process P has to wait for ability to begin executing Op in node v, since only the operation D_0, corresponding to the descriptor at the head of v queue, can be executed in v.

The algorithms seems to be blocking, but that is where the *helping* mechanism comes to the rescue. Instead of simply waiting for the Op descriptor to move to the head of the queue, P helps executing in node v the operation from the head of v queue — D_0 in the example above. Thus, even if the initiator process of D_0 is suspended, the system still achieves progress.

As discussed later, while helping to execute operations $D_0, D_1, \ldots$ in node v the process P removes their descriptors from the head of v queue and inserts them to the appropriate child queues. Thus, while helping other processes execute their operations in v, P moves Op descriptor closer to the head of v queue. Once P helps all preceding operations to finish their execution in node v, it can finally begin executing Op in v.

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. For example, execution of the `count` query on an external binary search tree may continue in either single child or both children: consider the explanations in Section 1.3 — the execution continues in both children iff node v is a *fork*

node and in a single child (either left or right) otherwise. In contrast, the `insert` operation should always be continued in a single child, since any key should be stored in exactly one leaf of the tree.

— For each child `c` from the set `C`:
  1. Modify `c` state (e.g., `c` subtree size), if necessary;
  2. Insert `Op` descriptor to `c` operations queue, thus allowing `Op` to continue its execution at lower levels of the tree;
— Remove `Op` descriptor from the head of `v` queue.

Note, that in the process of executing operation `Op` in node `v` the said operation only modifies states of `v` 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 shall overcome that by introducing the *fictive root* node (Fig. 22). The fictive root does not contain any state and has only one child (no matter how many children each tree node should have according to the tree structure) — 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.

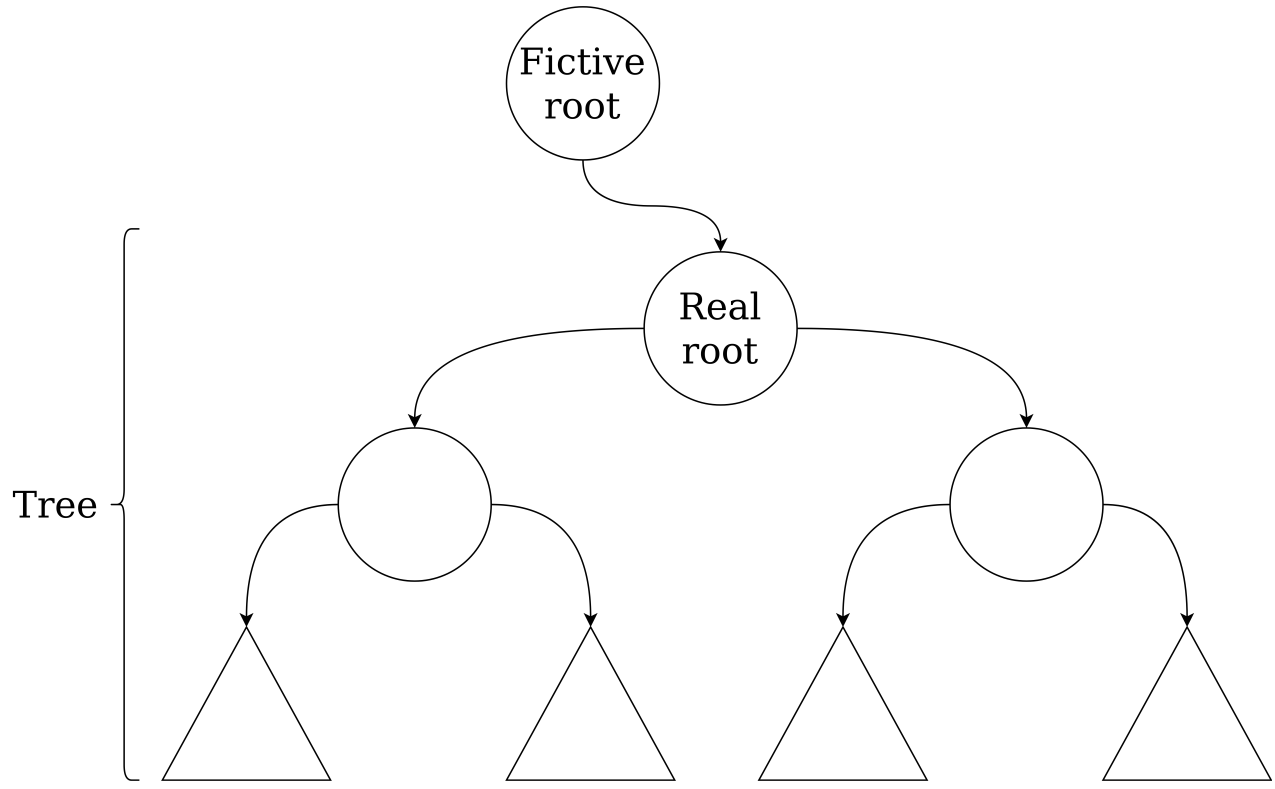


Figure 22 – The fictive root of the tree with no state and the only child: the real root

In the latter sections we will describe, how an operation Op can be executed in a node v: either by using CAS-N (Section 2.4) or without it (Section 2.5).

Since we allow processes 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 from the subtree of v.

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

Theorem 14. 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 $t_1, t_2, \ldots t_n$ — a sequence of timestamps of descriptors, located in Q, then $t_1 < t_2 < \ldots < t_n$ will hold.

Proof. 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 2.1 and as explained in Section 2.6.2, 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: $t_1 < t_2 < \ldots < t_n$. Consider descriptors D_i and D_j (Fig. 23), such that:
  – Both D_i and D_j should continue their execution at v;
  – timestamp(D_i) = t_i;

– timestamp(D_j) = t_j;
– D_i is located closer to the head of pv queue than D_j (therefore, D_i was inserted to pv queue prior to D_j) — thus according to the induction assumption $t_i < t_j$.

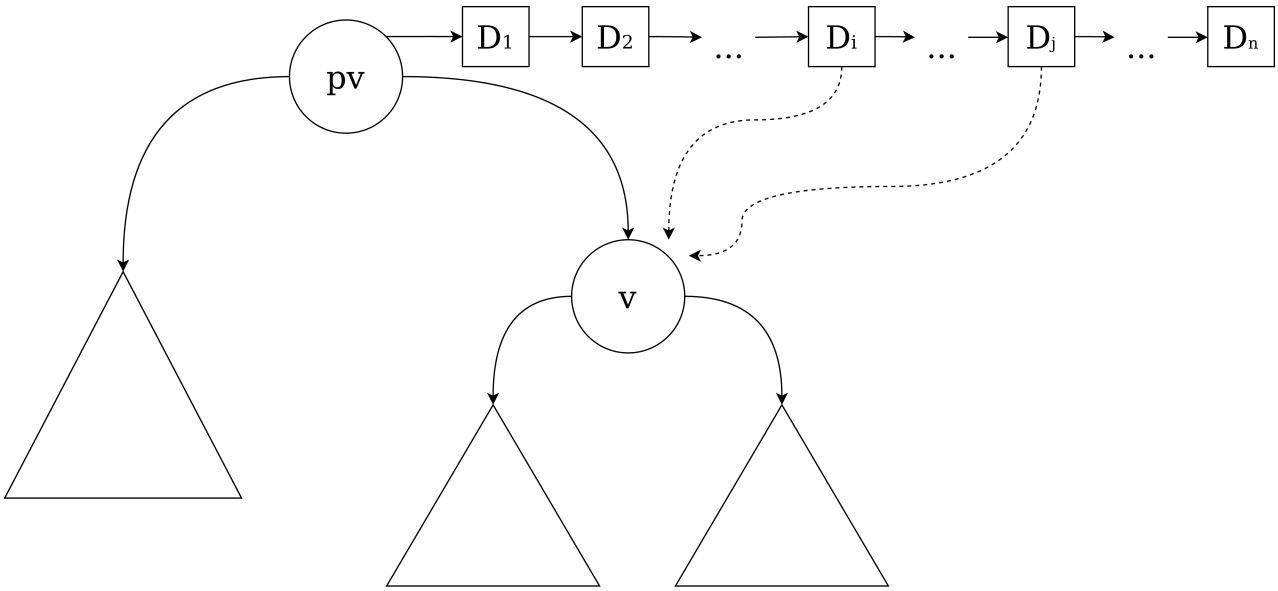


Figure 23 – Descriptor D_i is located closed to the head of pv queue than D_j, both D_i and D_j will continue their execution in v subtree

In that case, according to the algorithm, the execution of D_j 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_j 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.

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 following algorithm:

1. Try to read head_descriptor — the descriptor, located at the head of v queue.
2. If the queue is empty, we conclude that some other process has executed Op in node v. Thus, P continues traversing the tree, trying to execute Op at other nodes.
3. Otherwise, P examines the timestamp of the obtained descriptor. If head_descriptor.Timestamp > Op.Timestamp, P yet again concludes, that some other process has executed Op at node v.

4. Otherwise (`head_descriptor.Timestamp` $\leqslant$ `Op.Timestamp`) `Op` is still located in `v` queue: either at the head (if `head_descriptor.Timestamp` $=$ `Op.Timestamp`) or somewhere closer to the tail (if `head_descriptor.Timestamp` $<$ `Op.Timestamp`). In that case, `P` executes the operation, denoted by `head_descriptor` at node `v`.

Therefore, we can implement the algorithm of executing all operations, up to `Op.Timestamp`, from `v` queue the following way (Listing 8):

```
1  fun execute_until_timestamp(ts, v):
2      while true:
3          /*
4          Queue.peek() returns the first descriptor in FIFO order
5          */
6          head_descriptor := v.Queue.peek()
7          if head_descriptor = nil:
8              return
9          if head_descriptor.Timestamp > ts:
10             return
11         /*
12         execute_in_node changes states of v children
13         (in accordance with the operation, denoted by head_descriptor),
14         pushes head_descriptor to child queues,
15         removes head_descriptor from v queue
16         */
17         execute_in_node(head_descriptor, v)
```

Listing 8 – The algorithm to execute all operations, up to the specified timestamp `ts`, from `v` queue

Also, we should have a method to determine whether the operation execution has been finished or not. The motivation to introduce such a method is the following situation:

1. Process `P` starts executing operation `Op`.
2. `P` inserts descriptor of `Op` to the root queue.
3. `P` is suspended by the OS.
4. Other processes finish the execution of `Op`.
5. After being resumed by the OS, `P` should be able to learn, whether `Op` has already been executed or not.

We shall implement this capability by storing *result pointer* in each operation descriptor (Fig. 24). This pointer will point to a specific memory location that stores

either the operation result (if the operation execution has been already finished — in that case `P` can return that result to the caller) or `nil` (if the operation execution has not been finished yet — in that case `P` should continue traversing the tree and finish `Op` execution).

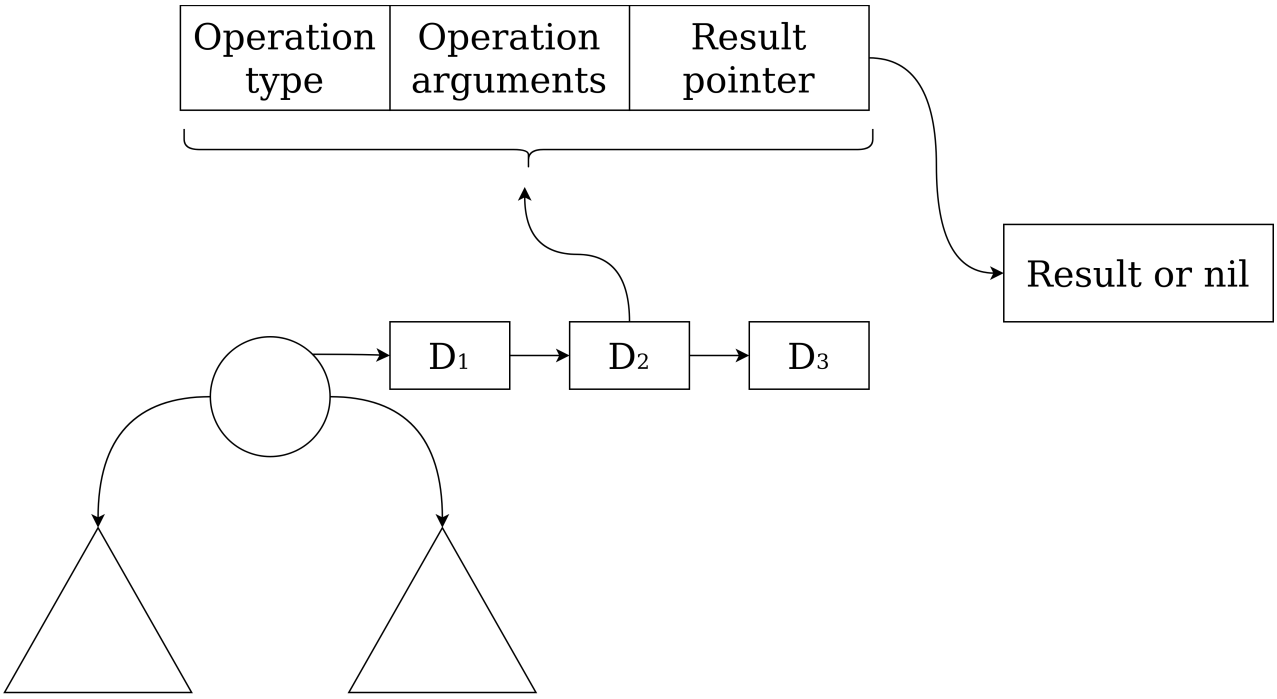


Figure 24 – A glance into a descriptor internals: each descriptor contains at least operation type (e.g. `insert`, `remove`, `count`), operation arguments (e.g., a key to insert or remove) and the result pointer

2.3. Ways to achieve parallelism

As was stated in Section 2.1, 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? It seems, that our proposed solution is not better than the solutions, based on the Universal Construction (see Section 1.8.3 for discussion on drawbacks of such solutions). Our scheme has one major advantage over the Universal Construction. As we remember, in solutions, based on the Universal Construction, execution of modifying operation O_2 could be started only after the execution of modifying operation O_1 has been finished — otherwise, one of these operations faces unsuccessful `CAS` and has to retry. In contrast, in our solution, two successful modifying operations may be executed in parallel if they are executed on different subtrees (Fig. 25).

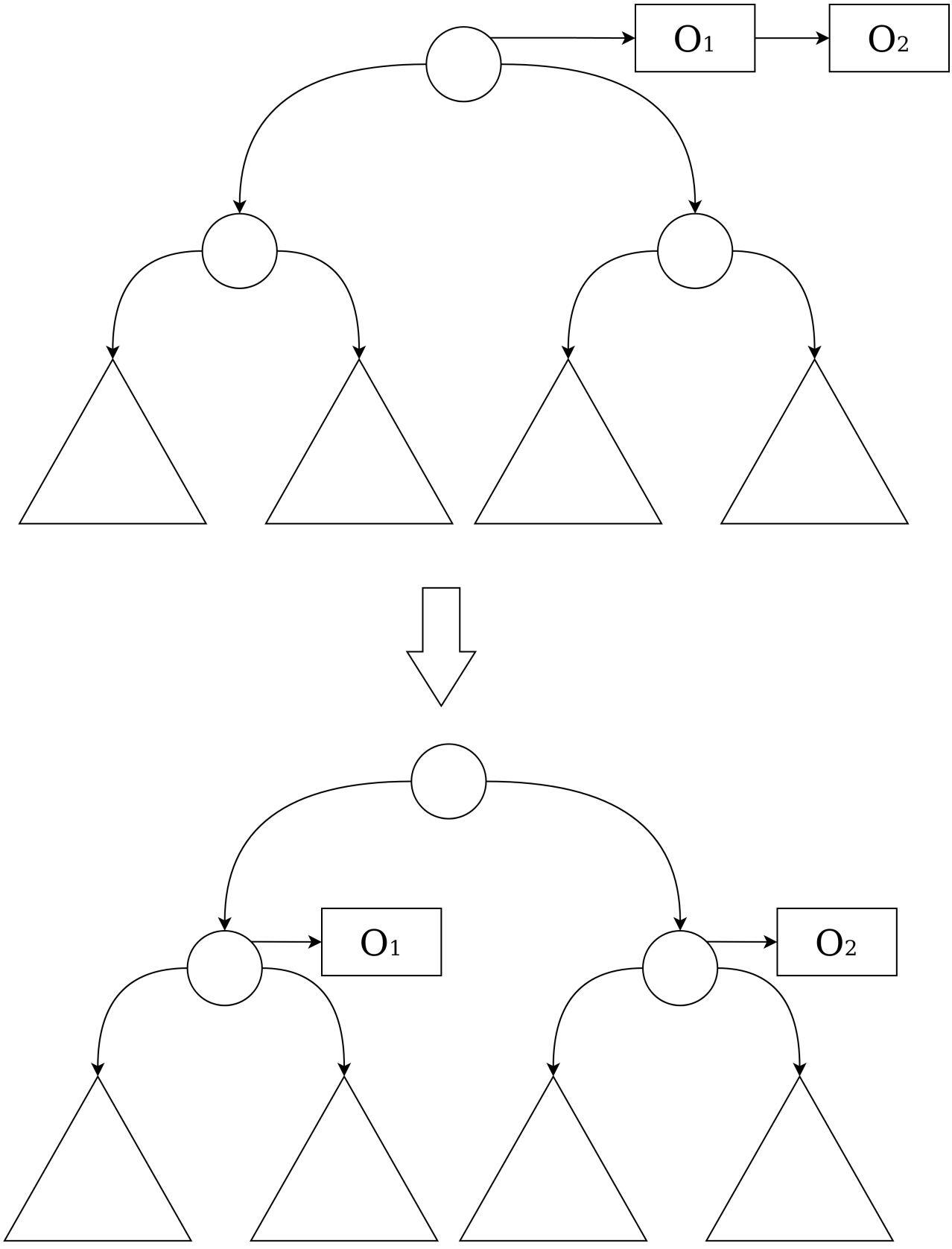


Figure 25 – After being routed to different subtrees, modifying operations O_1 and O_2 may be executed in parallel, in contrast to the Universal Construction-based solutions

Note that a particular execution order, determined by the queue, is enforced only in the root node — in the root node we execute O_1 before starting executing O_2. At lower tree levels, we do not enforce a particular execution order. This can be achieved because at lower tree levels O_1 and O_2 do not conflict with each other anymore, since they operate on different subtrees. Thus, our logical order (in which O_1 precedes O_2) does not require us to enforce a particular physical execution order on these operations, i.e., executing O_1 before starting executing O_2.

2.4. Executing an operation in a node via CAS-N

2.4.1. CAS-N definition and implementation

CAS-N is a powerful concurrent primitive that takes N registers, N expected values, and N new values as an input. After that, it atomically checks, whether $\forall i \in [1 \dots N]$: value of i-th register equals to the i-th expected value. If so, it modifies values of all registers, so that the value of the i-th register becomes equal to the i-th new value, and returns true. Otherwise — if $\exists j \in [1 \dots N]$: value of j-th register does not equal to j-th expected value — it leaves all registers unmodified and returns false. The pseudocode of CAS-N is presented on Listing 9:

```
1  fun multi_cas(n: int, Registers: [n]Register,
2                expected_values: [n]Value, new_values: [n]Value):
3      atomically:
4          for i ← 1 .. n:
5              v := Registers[i]
6              if v ≠ expected_values[i]:
7                  return false
8          for i ← 1 .. n:
9              Registers[i] ← new_values[i]
10         return true
```

Listing 9 – Pseudocode for CAS-N operation

We can use *Two-Phase Locking* protocol, described e.g., in [3], to implement CAS-N the following way (Listing 10):

```
1  fun multi_cas(n: int, Registers: [n]Register, Locks: [n]Mutex,
2                expected_values: [n]Value, new_values: [n]Value):
3      for i ← 1 .. n:
4          Locks[i].lock()
5          v := Registers[i]
6          if v ≠ expected_values[i]:
7              for j ← 1 .. i:
8                  Locks[j].unlock()
9              return false
10     for i ← 1 .. n:
```

```
11          Registers[i] ← new_values[i]
12          Locks[i].unlock()
13      return true
```

Listing 10 – Two-phase lock-based implementation of CAS-N

This implementation is lock-based, thus, it does not guarantee even the obstruction-freedom (see Section 1.7 for discussion on concurrent progress guarantees). Moreover, it is prone to deadlocks [3] — thus, the need for an implementation, not suffering from these drawbacks, arise. Harris et al. [15] described software lock-free implementation of CAS-N. Feldman et al. [10] showed that there exists a practical wait-free implementation of CAS-N. For efficiency reasons, it is possible to implement special cases of CAS-N (e.g., CAS-2 [39]) in a hardware, making them wait-free and extremely efficient (e.g., since the hardware implementation does dot require dynamic memory allocation).

2.4.2. Using CAS-N for operation execution

As was discussed in Section 2.2, execution of operation Op in node v consists of:

— Modifying states of v children;
— Modifying v child queues — inserting Op descriptor into some of them;
— Modifying v queue — removing Op descriptor from its head;

We can do it atomically using CAS-N. To allow atomic modification of queues with CAS-N, we shall employ persistent queues (Fig. 26). We can simply store a pointer to the current version of node persistent queue in node.Q_Ptr register. After that, we can use CAS(&node.Q_Ptr, cur_queue, new_queue) or CAS-N to try to atomically modify the queue (see Section 1.8.3 for explanation on how CAS can be used to modify persistent data structures).

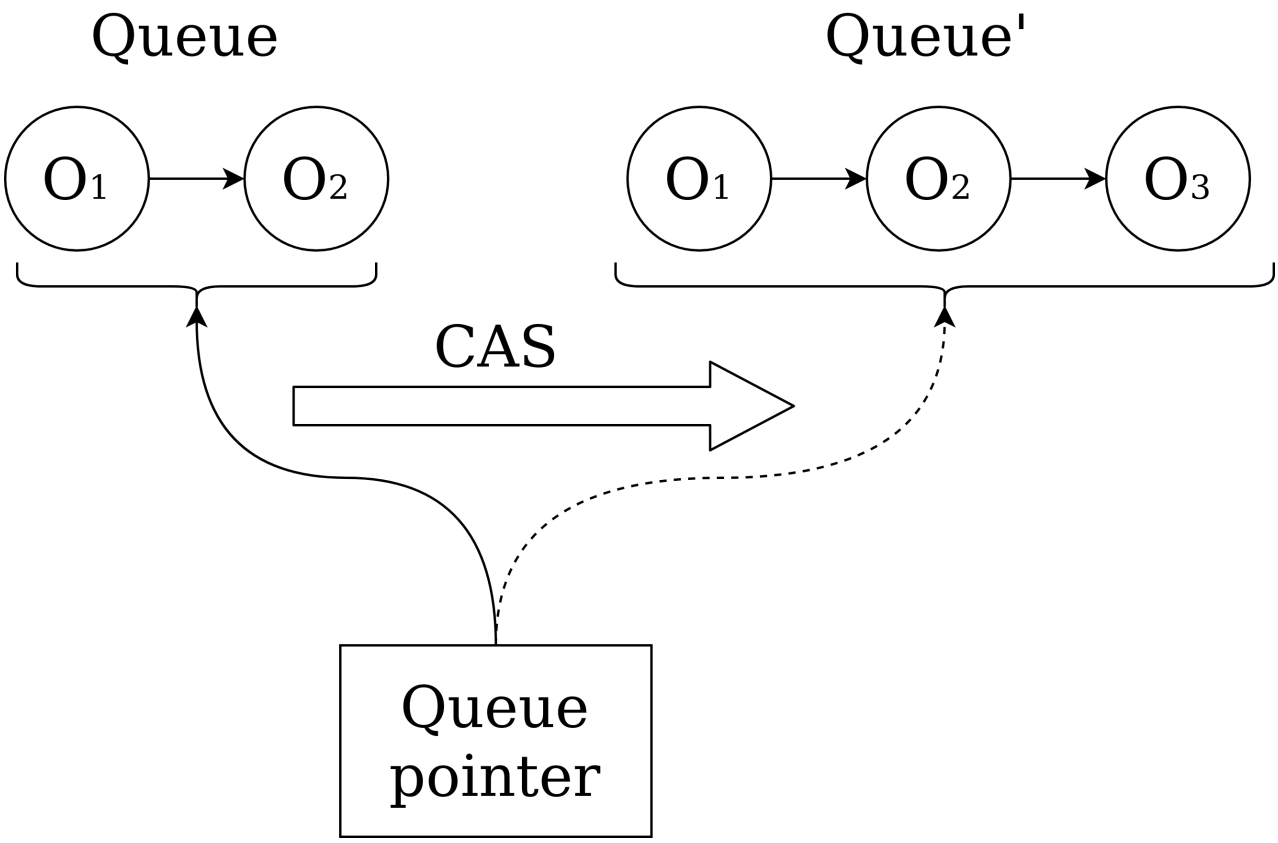


Figure 26 – Using `CAS` to atomically modify persistent queues

Multiple persistent queues have been studied and described in the literature. Bankers queue [30] is considered one of the fastest and the most memory-optimal.

Node state can be modified either: 1) the same way as queues — via storing the current state in the heap and modifying the pointer to it located in the tree node (this method is described in more details in Section 2.5); 2) directly in the tree node — each component of the state (e.g., subtree size) is located directly in the tree node and modified inplace. In the last case, `CAS-N` should be applied to each component of the state.

Therefore, we can use `CAS-N` to atomically modify all the necessary registers: child queue pointers, `v` queue pointer, and child states (Fig. 27).

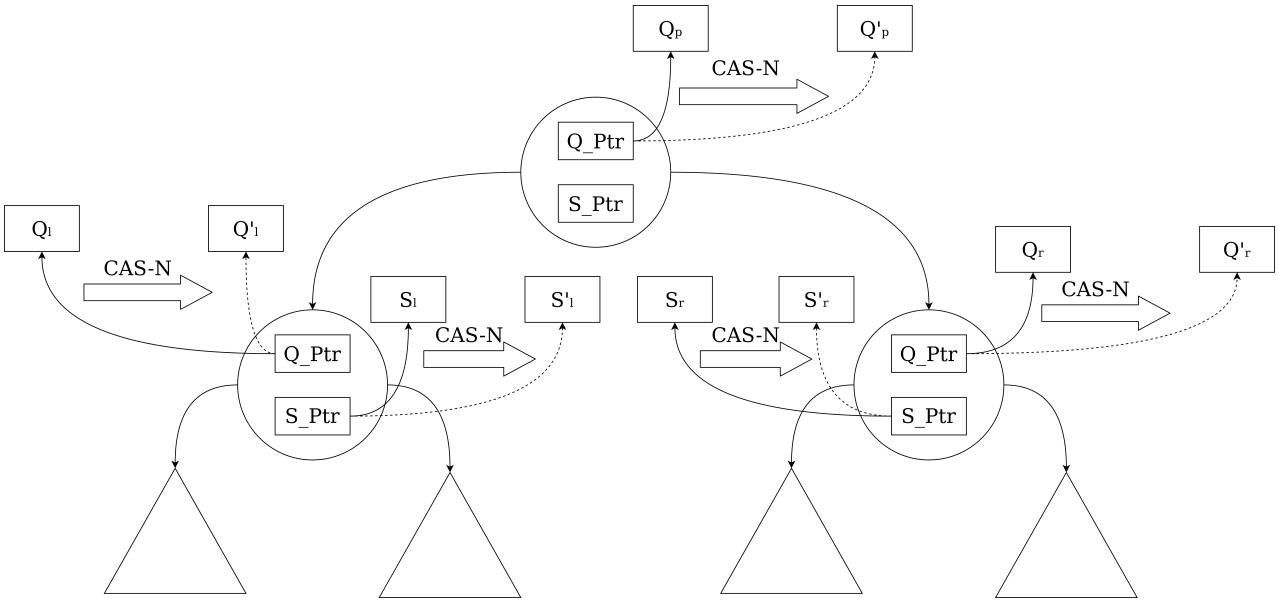
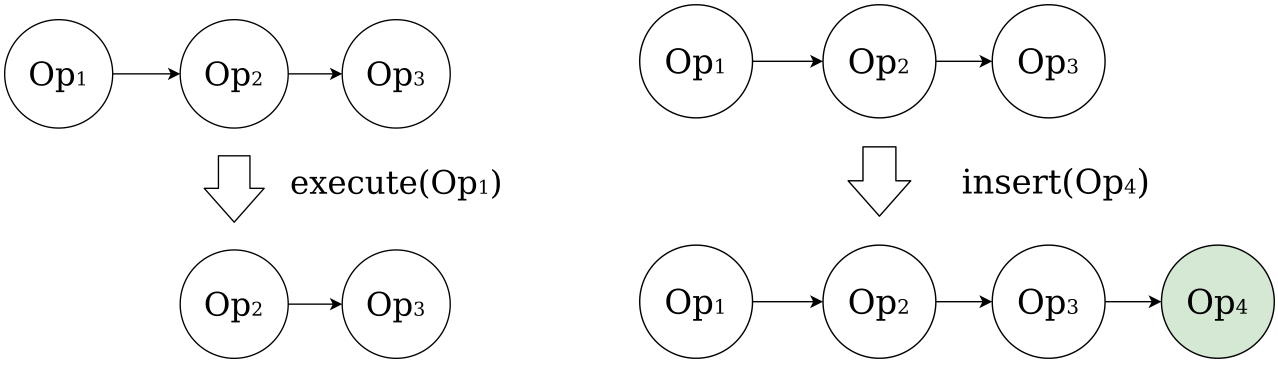


Figure 27 – Using `CAS-N` to atomically modify child queues, parent queue, and child states in a binary tree. Here we store the node state in the heap and store `S_Ptr` — a pointer to the node state — in the node. In that case, `CAS-5` is sufficient to modify all the required registers: both child queues, both child states and the parent queue

If `CAS-N` returns `true` we conclude that we have successfully executed `Op` in node `v`. Otherwise, the `CAS-N` may return `false` because of any of the following two reasons (Fig 28):

1. Other process has executed in node `v` the operation we are trying to execute, including removal of its descriptor from the head of `v` queue (Fig. 28a).
2. Other process has inserted new operation descriptor to `v` queue, without executing in node `v` the operation we are trying to execute (Fig. 28b).



(a) `CAS-N` fails due to other process executing Op_1 in `v`

(b) `CAS-N` fails due to other process inserting new descriptor into `v` queue

Figure 28 – Reasons for `CAS-N` to return `false`

We can distinguish between these two situations by peeking descriptor from the head of v queue. If that descriptor still denotes the operation, we are trying to execute — we conclude, that situation (2) happened and try to execute that operation in v one more time. Otherwise, we can conclude, that some other process executed the operation, we are trying to execute, in node v, i.e., situation (1) happened. In that case, we may proceed to execute the next operation from v queue or to traverse the tree, depending on whether our initiated operation has been executed in v or not (see Section 2.2 for details).

Thus, the algorithm executing operation op in node v using CAS-N can be implemented the following way (Listing 11):

```
1  fun execute_in_node(op, v):
2      while true:
3          cur_v_queue := v.Queue_Ptr
4          if cur_v_queue.peek() ≠ op:
5              /* another process has executed op in v */
6              return
7          new_v_queue := cur_v_queue.pop_persistent()
8
9          cas_registers := [&v.Queue_Ptr]
10         cas_expected_values := [cur_v_queue]
11         cas_new_values := [new_v_queue]
12
13         C ← /*
14             set of v children in which the execution of op should continue
15         */
16         for c in C:
17             cur_child_state := c.State_Ptr
18             new_child_state := op.get_modified_state(cur_child_state)
19             cas_registers.append(&c.State_Ptr)
20             cas_expected_values.append(cur_child_state)
21             cas_new_values.append(new_child_state)
22
23             cur_child_queue = c.Queue_Ptr
24             new_child_queue := cur_child_queue.push_persistent(op)
25             cas_registers.append(&c.Queue_Ptr)
26             cas_expected_values.append(cur_child_queue)
27             cas_new_values.append(new_child_queue)
28
29         cas_res := multi_cas(
30             n = cas_registers.length(),
31             Registers = cas_registers,
32             expected_values = cas_expected_values,
33             new_values = cas_new_values
34         )
```

```
35          if cas_res:
36              return
37          /*
38          Otherwise, try to execute op in v
39          from the very beginning one more time
40          */
```

Listing 11 – Algorithm for executing operation `op` in node `v` using `CAS-N`

The algorithm is lock-free since each retry means that some other descriptor was inserted into `v` queue, i.e., another process executed some operation in `v` parent.

2.5. Execution of an operation in a node without `CAS-N`

Despite the fact that `CAS-N` provides the ability to execute an operation in a node atomically with lock-freedom progress guarantees, this concurrent primitive remains very inefficient due to indirections and a dynamic memory allocation. Thus, we want to design a method to execute operations in a node without using `CAS-N`.

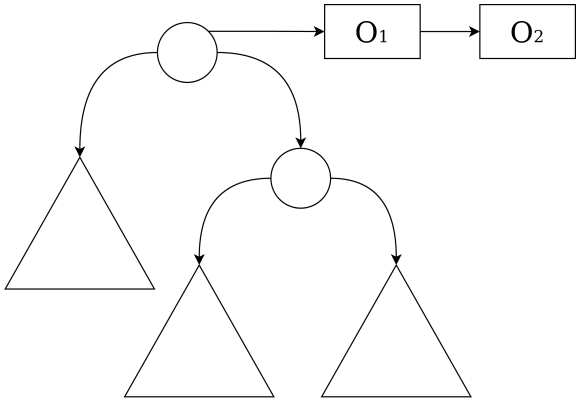
The algorithm to execute operation `Op` in node `v` consists of the following steps:

1. Determine the set of children `C`, in which execution of `Op` should continue.
2. Traverse the set `C`. For each child `c` from `C`:
   2.1. Atomically read `c` state.
   2.2. If `c` state has not been modified by `Op` yet, modify it. We explain how to do it below.
   2.3. Insert `Op` descriptor to `c` queue if it has not been yet inserted. We explain in Section 2.6.3 how to do it.
3. If `Op` descriptor has not been yet removed from `v` queue, remove it. We explain in Section 2.6.4 how to do it.

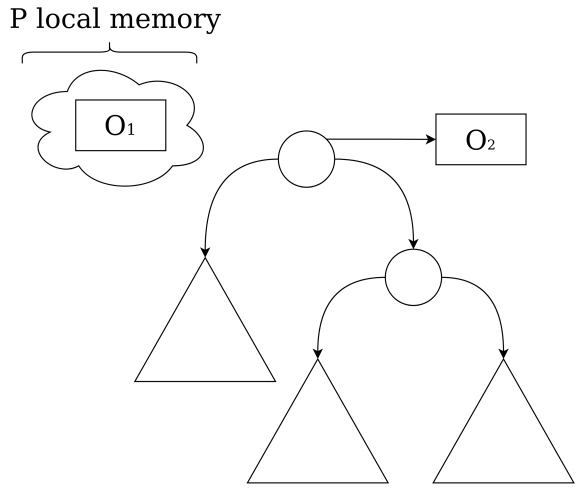
Note, that the removal of `Op` descriptor from the head of `v` queue should be done after the insertion of `Op` descriptor to child queues and modification of child states are finished. Otherwise, the execution of later operations in `v` may start before the execution of `Op` in `v` is finished, which may break the main invariant (Section 2.1). Consider, for example, the following scenario:

1. Descriptors of O_1 and O_2 are located in `v` queue and execution of both O_1 and O_2 should be continued in `v.Right` (Fig. 29a).
2. Process `P` reads O_1 descriptor from the head of `v` queue and starts executing O_1 in `v`: removes O_1 descriptor from the head of `v` queue before inserting it to `v.Right` queue (Fig. 29b).

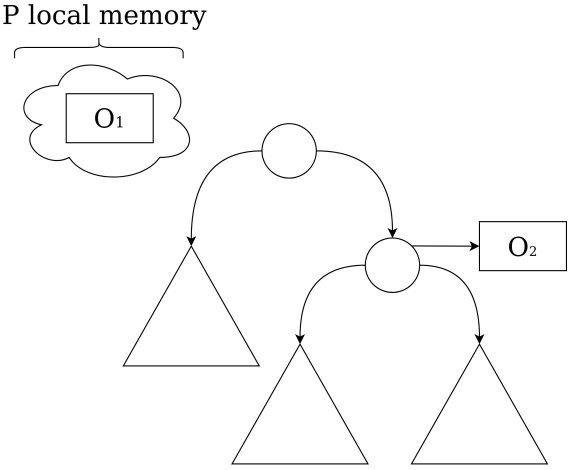
3. `P` is suspended by the OS.
4. Process `R` reads O_2 descriptor from the head of `v` queue and executes O_2 in `v`: removes it from the head of `v` queue and inserts it to `v.Right` queue (Fig. 29c).
5. Process `P` finishes executing O_1 in `v`: inserts O_1 descriptor to `v.Right` queue (Fig. 29d). Thus, the descriptors are placed in `v.Right` queue in the wrong order and the main invariant is broken: O_2 will be applied to `v.Right` subtree before O_1, despite O_1 descriptor was inserted to `v` queue before O_2 descriptor.



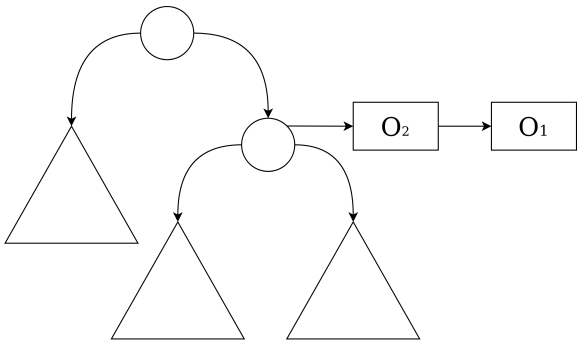
(a) The initial tree structure



(b) Process `P` removes O_1 descriptor from the head of the parent queue before inserting it to the child queue



(c) Process `R` executes O_2 in `v`



(d) Process `P` finishes executing O_1 in node `v`

Figure 29 – The main invariant is broken if a descriptor is removed from the head of the parent queue before being inserted to child queues

Note also, that actions (2) and (3) may be executed by multiple processes concurrently. Consider the following scenario:

1. Descriptor of operation `Op` is located at the head of `v` queue, execution of `Op` should continue at `v.Left`;
2. Processes `P` and `Q` both read `Op` descriptor from the head of `v` queue;
3. `P` and `Q` both try to modify `v.Left` state;
4. `P` and `Q` both try to push `Op` descriptor to `v.Left` queue;
5. `P` and `Q` both try to remove `Op` descriptor from the head of `v` queue;

Thus, 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.

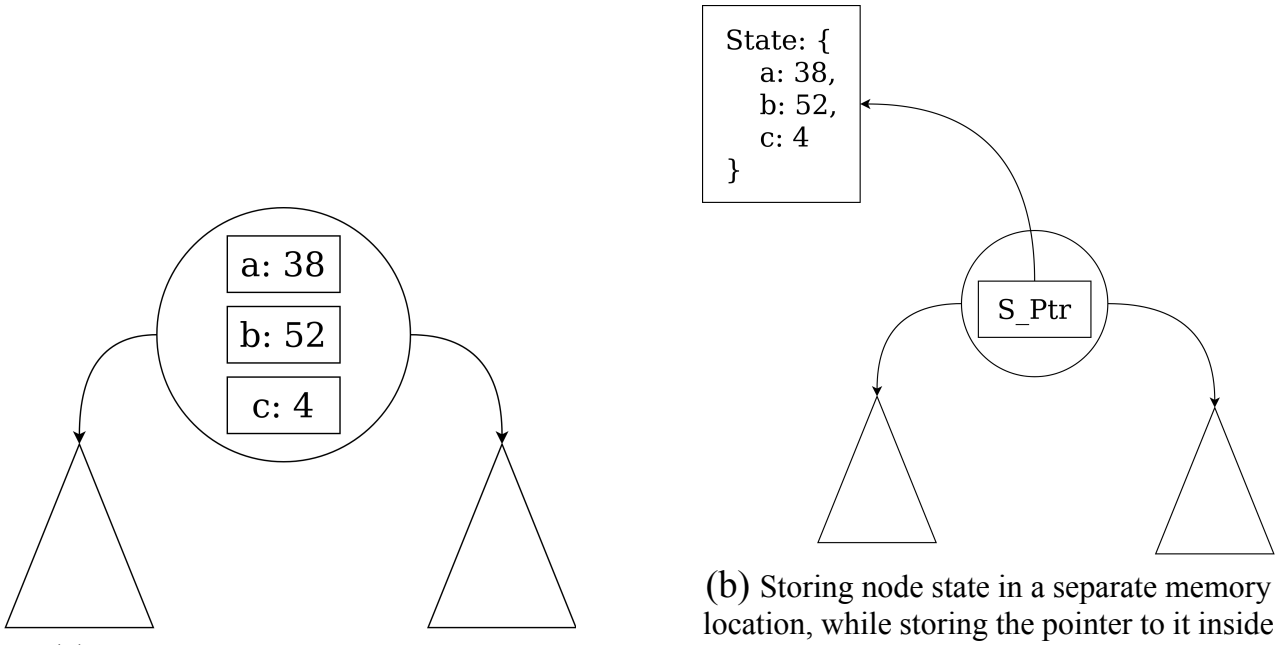
Exactly-once insertion to and removal from queues is handled by our implementation of concurrent queues. Queues provide two procedures:

— `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. The implementation of this procedure is discussed in Section 2.6.3.

— `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 implementation of this procedure is discussed in Section 2.6.4.

Therefore, in this chapter we explain on how to change the node state exactly once and how to read it atomically.

The main problem with reading the state atomically is that it may consist of multiple fields. To solve this problem, we do not store the state directly inside the node (Fig. 30a) — instead, the immutable state is located somewhere in the heap and the node will contain only one field `S_Ptr` — the pointer to the heap location of the state (Fig. 30b).



(a) Storing node state inside the node

(b) Storing node state in a separate memory location, while storing the pointer to it inside the node

Figure 30 – Different methods of storing the node state

The state, located in the heap, is considered immutable and is never modified. To modify the node state, we simply do the following (Fig. 31):

1. Create the structure, corresponding to the modified state, with an arbitrary set of fields changed.
2. Place the modified state somewhere in the heap.
3. Change the `node.S_Ptr` so than it points to the modified state.

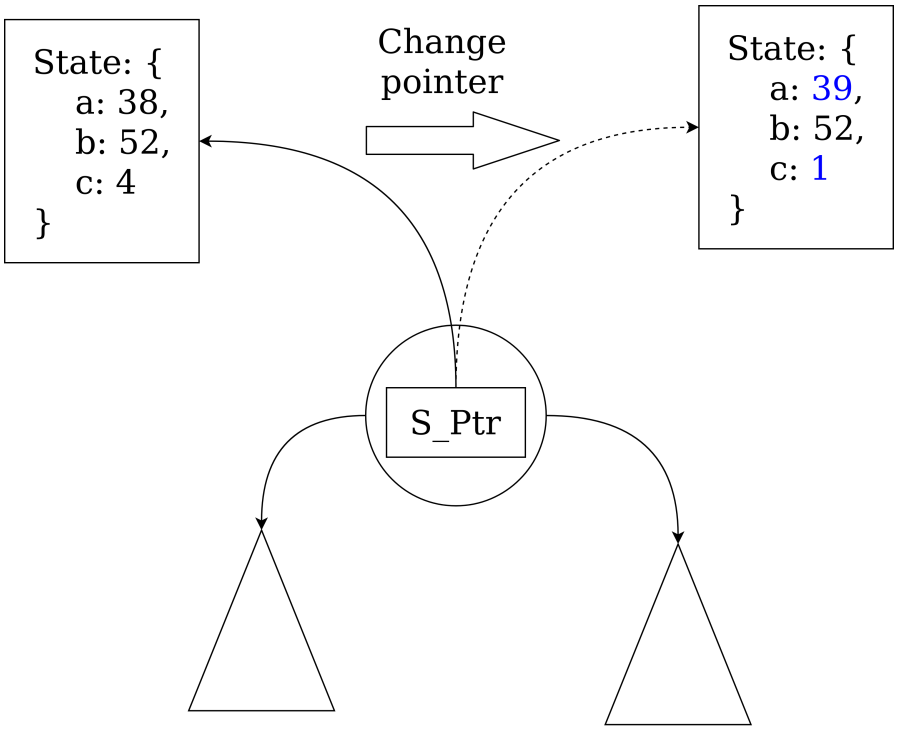


Figure 31 – The modification of the node state via the creation of a new state structure and change of `S_Ptr`

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 problem of modifying the state exactly once. In the state we shall 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 node `v` state, we should first read the current `v` state and acquire the last modification timestamp.

— If `Ts_Mod` $\geqslant$ `Op.Timestamp` then `v` state has been already modified by `Op`. In that case, we simply do not try to modify `v` 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 in accordance with each executed operation exactly once. Therefore, the algorithm can be implemented the following way (Listing 12):

```
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
        if cur_state.Ts_Mod < op.Timestamp:
            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)
    v.Queue.pop_if(op)
```

Listing 12 – Algorithm for executing operation `op` in node `v` without using `CAS-N`

2.6. Operation queue implementation

2.6.1. Queue structure

We implement all the necessary operations on a slightly modified version of the Michael-Scott queue [27].

We maintain the descriptor queue as a linked list of nodes. Each node contains two fields (Listing 13, Fig. 32):

— `Data`, that stores the operation descriptor.

— `Next`, that stores a pointer to the next node in the queue, or `nil` if that node is the last in the queue.

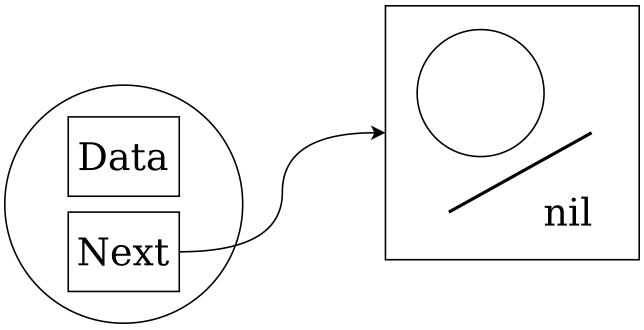


Figure 32 – Queue node structure

```
type QueueNode = struct {
    Data: OperationDescriptor,
    Next: QueueNode*
}
```

Listing 13 – Queue node structure

For each queue we maintain 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 (Fig. 33). 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.

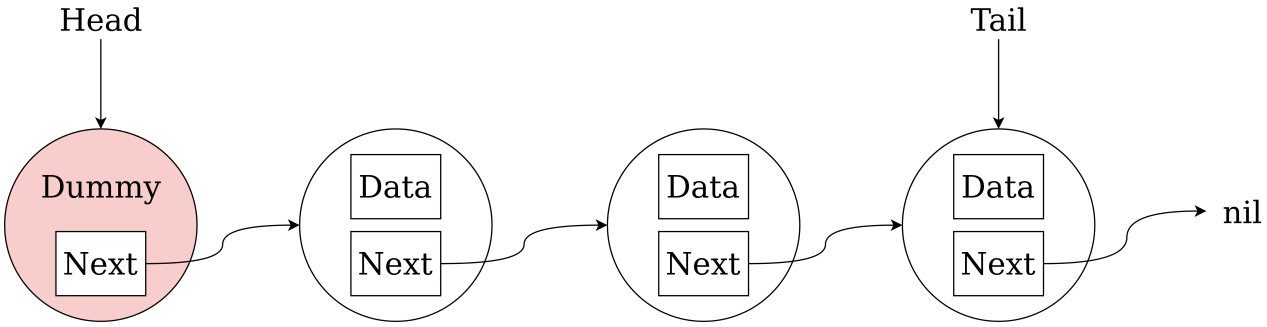


Figure 33 – Queue structure

An empty queue consists of a single dummy node, pointed at by both `Head` and `Tail` pointers (Fig. 34).

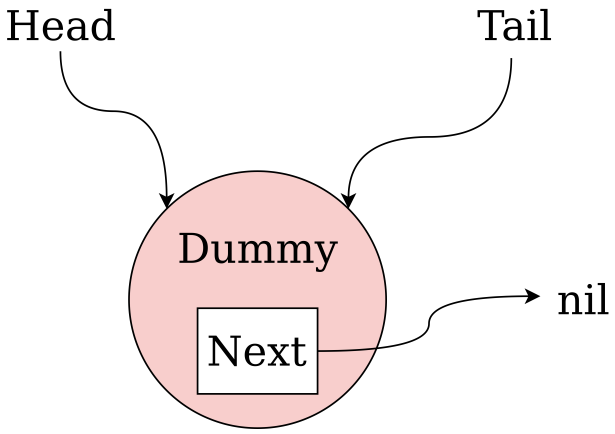


Figure 34 – Empty queue structure

2.6.2. `push` with acquiring operation timestamp

As discussed in Section 2.1, 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.

Note, that the descriptor becomes visible to all the system processes at the moment it is added to the root queue, and, as described in Section 2.2, the system processes examine timestamps of all descriptors in order to execute their operations.

Thus, the timestamp should be written to the `descriptor.Timestamp` field before the descriptor is added to the root queue.

As was stated in Section 2.6.1, we can use a slight modification of Michael-Scott queue [27] to implement the timestanp alocation mechanism for the root queue.

The algorithm can be structured the following way:

1. Read `cur_tail := Queue.Tail` — the current queue tail to learn the maximal allocated timestamp and start inserting the new descriptor to the tail of the queue. At this moment, multiple possible situations can happen:
   — The queue is not empty and the tail points to the latest added descriptor (Fig. 35).

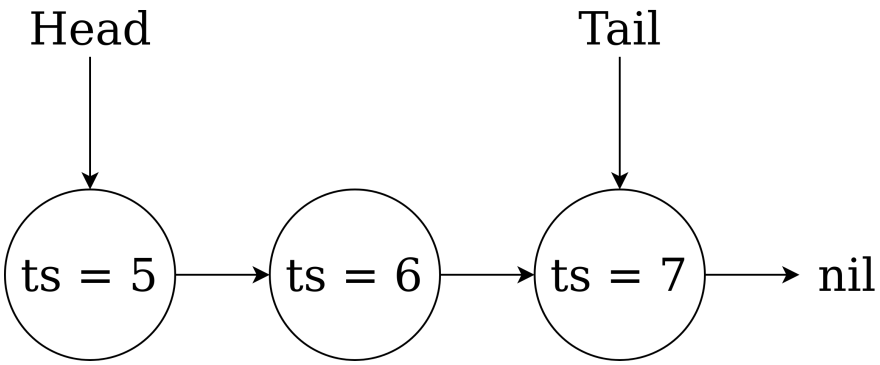


Figure 35 – The queue is not empty and the tail points to the latest added descriptor

   Thus, that descriptor contains the maximal timestamp, allocated by now and we can learn that timestamp by simply reading `cur_tail.Data.Timestamp`
   — The queue is empty, but at least one node has been added to it since the beginning of the execution (Fig. 36). In that case, `cur_tail` points to the node that was the last removed from the queue, as guaranteed by the Michael-Scott queue structure [27].

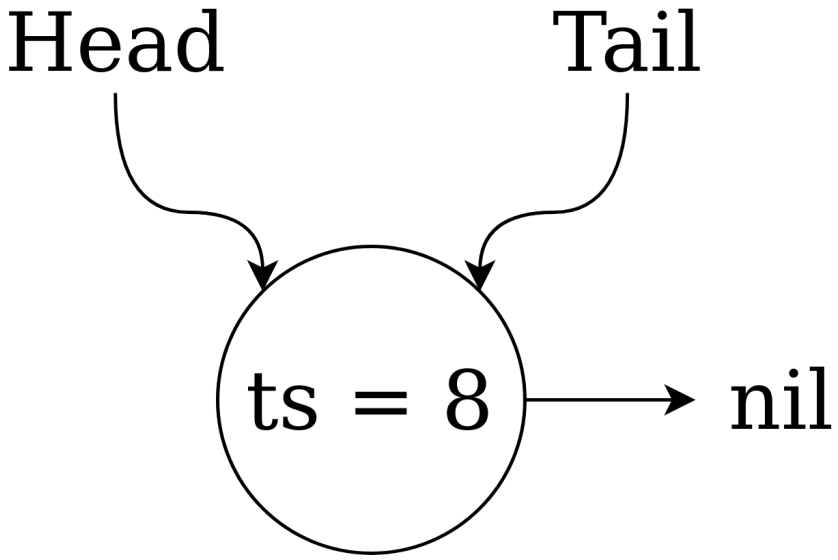


Figure 36 – The queue is empty but it was non-empty at least once

Therefore, the node, pointed at by `cur_tail`, is the last node added to the queue, thus it contains the maximal allocated timestamp. Therefore, as in the previous case, we can learn the maximal allocated timestamp by reading `cur_tail.Data.Timestamp`.

— The queue is empty and not a single node has been inserted to it since the beginning of the execution. In that case, the `cur_tail` points at the dummy node, as guaranteed by the Michael-Scott queue structure [27] and stated in Section 2.6.1 (Fig 37).

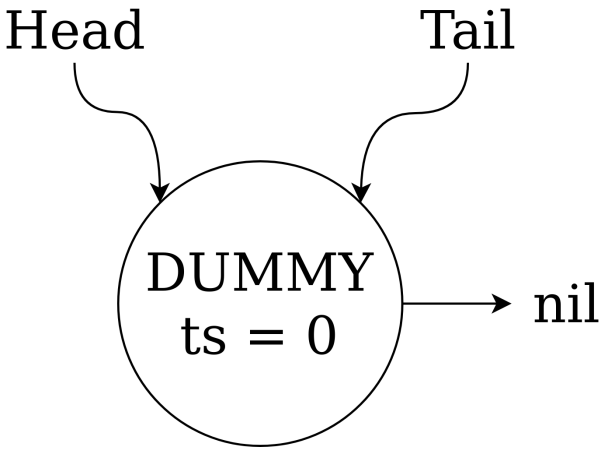


Figure 37 – The queue has always been empty

We may consider zero to be the maximal timestamp, allocated at the beginning of the execution. Thus, we construct a dummy node so that it contains zero as its timestamp (i.e. `Dummy_Node.Data.Timestamp = 0`). Thus, yet again

we we can learn the maximal allocated timestamp by reading `cur_tail.Data.Timestamp`

— The queue is not empty and the tail does not point to the latest added descriptor (Fig. 38). This can happen if another descriptor is being inserted to the queue concurrently.

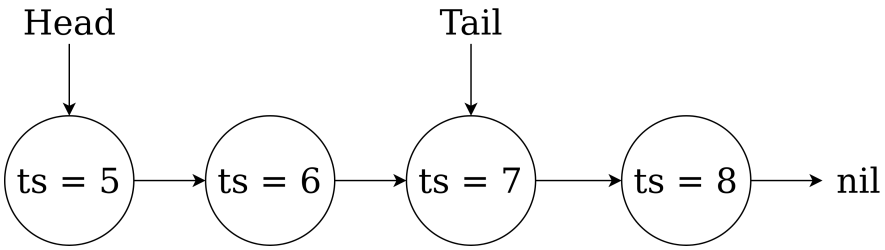


Figure 38 – The queue is not empty and the tail does not point to the latest added descriptor

In that case, as described later, we retry the whole procedure from the very beginning, i.e., from step (1). Thus, it does not matter, which timestamp we learn — for example, we may choose `cur_tail.Data.Timestamp` to be the learned timestamp.

Therefore, in all possible cases we can learn the maximal allocated timestamp by reading `cur_tail.Data.Timestamp`.

2. After learning the maximal allocated timestamp, we set `new_descriptor.Timestamp` equal to the the learned maximal timestamp incremented by one. Note, that this `write` operation is not concurrent with any `read` or `write` operation on `new_descriptor`. Indeed, writing `new_descriptor.Timestamp` is performed only by the initiator process on the `new_descriptor` before it is inserted to the queue, and thus before the `new_descriptor` becomes visible to other processes.
3. We try to add the new descriptor to the tail of the queue the same way we insert elements to the Michael-Scott queue: we simply try to perform `CAS(&cur_tail.Next, nil, new_node)`. We can have two possible outcomes of that `CAS`:
   — If the `CAS` returns `false`, then some other process successfully inserted its descriptor to the tail of the queue thus modifying `cur_tail.Next`. In that case, we help the successful process finish its insertion. We be-

gin with reading `other_process_tail := cur_tail.Next`, after that we try to move the queue tail forward by executing `CAS(&Queue.Tail, cur_tail, other_process_tail)` (Fig. 39).

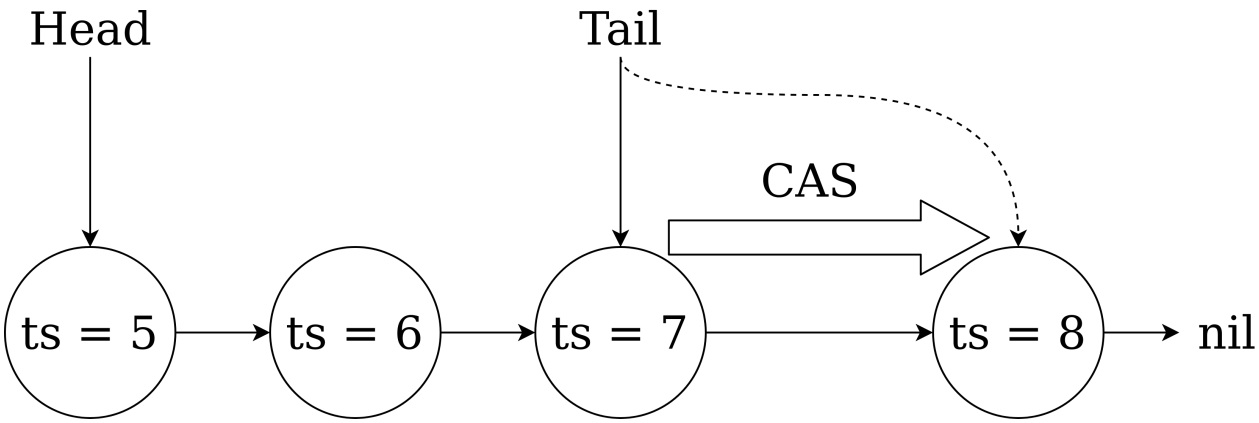


Figure 39 – Moving the queue tail forward

As any `CAS`, that `CAS` can be either 1) successful — in that case we have helped the other process and moved the queue tail forward; 2) unsuccessful — in that case, some other process helped before us. In either case, we simply retry the whole procedure from the from step (1).

— If `CAS` returns `true`, we added the descriptor to the tail of the queue. In that case, as in the previous one, we should move the queue tail forward by `CAS(&Queue.Tail, cur_tail, new_node)`. After that, we simply finish the insertion, no matter is the second `CAS` successful (if so, we moved `Queue.Tail` forward) or not (in that case, some other process moved it to help us, as described above).

The algorithm can be implemented the following way (Listing 14):

```
1  /*
2  Executed by the initiator process at the
3  beginning of the operation execution
4  */
5  fun push_acquire_timestamp(Root_Queue, descriptor):
6      new_node := new QueueNode(Data = descriptor, Next = nil)
7      while true:
8          cur_tail := Root_Queue.Tail
9          max_timestamp := cur_tail.Data.Timestamp
10         descriptor.Timestamp ← max_timestamp + 1
11         if CAS(&cur_tail.Next, nil, new_node):
12             CAS(&Root_Queue.Tail, cur_tail, new_node)
13             return
```

```
14          else:
15              other_process_tail := cur_tail.Next
16              CAS(&Root_Queue.Tail, cur_tail, other_process_tail)
17              /* Retry the whole operation from the very beginning */
```

Listing 14 – Implementation of the `push` procedure with acquiring operation timestamp

2.6.3. `push_if` implementation

As discussed in Section 2.5, 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). Just like in the previous case, the procedure is based on the Michael-Scott queue insertion algorithm [27] and can be implemented the following way:

1. We learn the current queue tail by reading `cur_tail := Queue.Tail;`
2. We learn the maximal operation timestamp, that has ever been inserted to the queue. We do it the same way as in Section 2.6.2 — by reading `cur_queue.Data.Timestamp` value.
3. If that timestamp is greater than or equal to `descriptor.Timestamp`, we can conclude that the descriptor was already inserted to the queue. Thus, we can simply finish the operation, leaving the queue unmodified.

   As was stated in Section 2.6.2, we can learn the timestamp not from the last node, but from the penultimate node (Fig. 40).

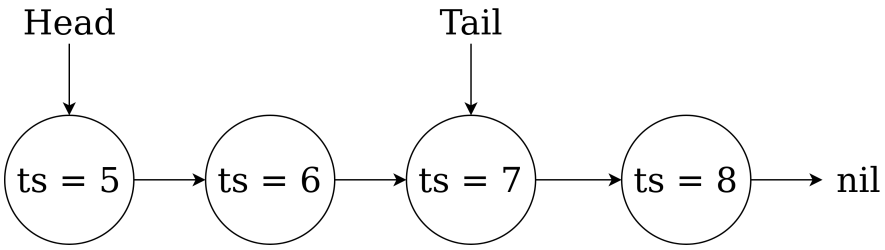


Figure 40 – The queue tail does not point to the latest added descriptor

Nevertheless, our conclusion remains the same: if at least one descriptor in the queue has `Timestamp` $\geqslant$ `descriptor.Timestamp` (even if the said descriptor is located in the penultimate queue node), it means that `descriptor` has already been inserted to the queue and we can simply finish the insertion.

4. Otherwise, we try to insert the descriptor to the tail of the queue the same way, we did in Section 2.6.2. Note, that if the acquired `cur_tail` pointer was pointing to the penultimate node, we shall simply retry the insertion from the very beginning.

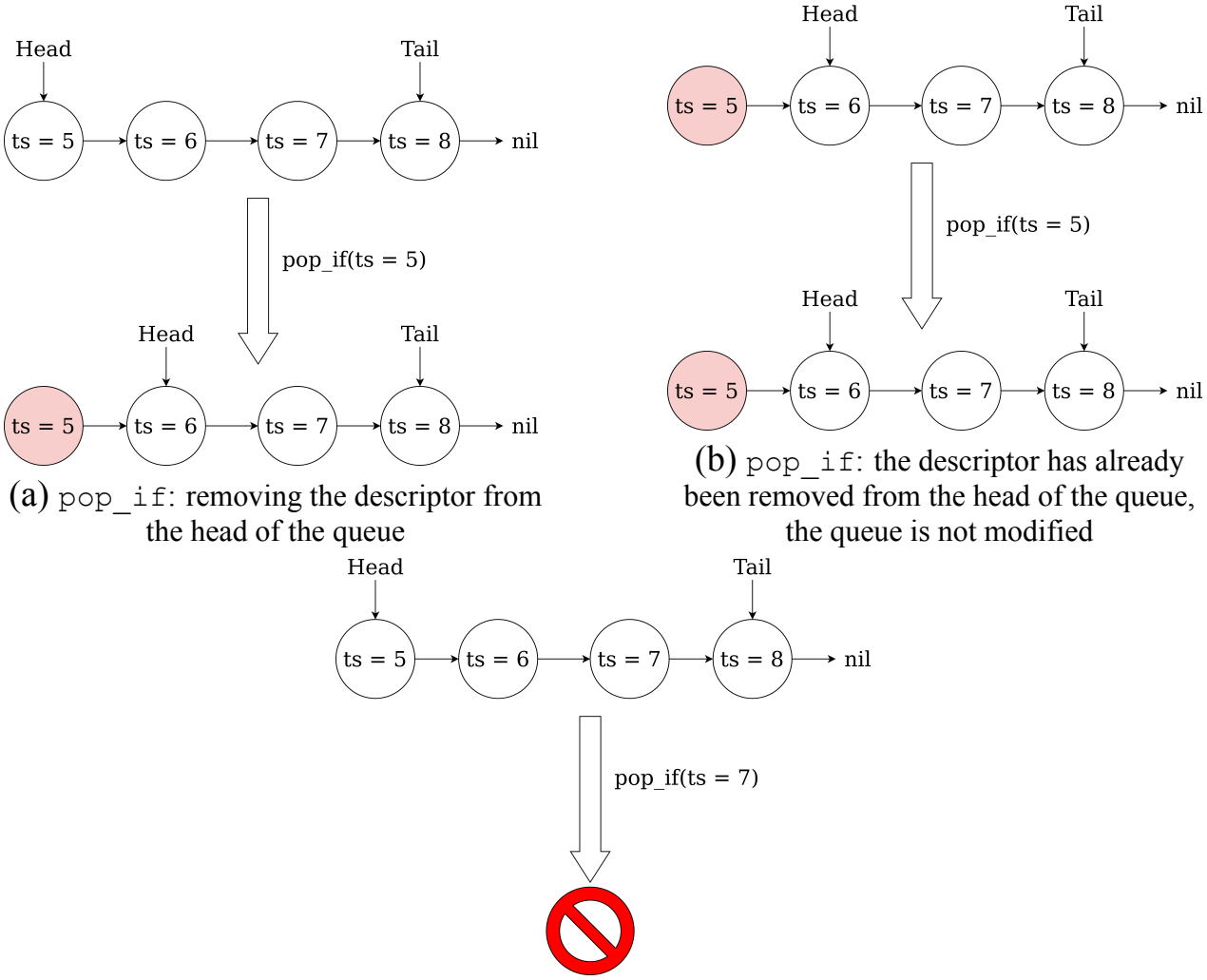
Therefore, the algorithm can be implemented the following way (Listing 15):

```
1  fun push_if(Non_Root_Queue, descriptor):
2      new_node := new QueueNode(Data = descriptor, Next = nil)
3      while true:
4          cur_tail := Non_Root_Queue.Tail
5          if cur_tail.Data.Timestamp ⩾ descriptor.Timestamp:
6              return
7          elif CAS(&cur_tail.Next, nil, new_node):
8              CAS(&Non_Root_Queue.Tail, cur_tail, new_node)
9              return
10         else:
11             other_process_tail := cur_tail.Next
12             CAS(&Non_Root_Queue.Tail, cur_tail, other_process_tail)
13             /* Retry the whole operation from the very beginning */
```

Listing 15 – Implementation of the `push_if` procedure

2.6.4. `pop_if` implementation

As discussed in Section 2.5, 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 (Fig. 41a). Otherwise, the queue is left unmodified (Fig. 41b) — 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 (Fig. 41c).



(a) `pop_if`: removing the descriptor from the head of the queue

(b) `pop_if`: the descriptor has already been removed from the head of the queue, the queue is not modified

(c) `pop_if`: cannot try to remove a descriptor that was never located at the head of the queue

Figure 41 – Execution of `pop_if` procedure for different queues

Yet again, we can use slightly modified version of Michael-Scott queue [27] to implement such a procedure in the following way:

1. Acquire head and tail nodes by reading `cur_head := Queue.Head` and `cur_tail := Queue.Tail`, respectively. According to the Michael-Scott queue structure [27] the head node is a dummy node, that does not contain any data. Instead, the first descriptor is located at the node, pointed at by `cur_head.Next` (Fig. 42).

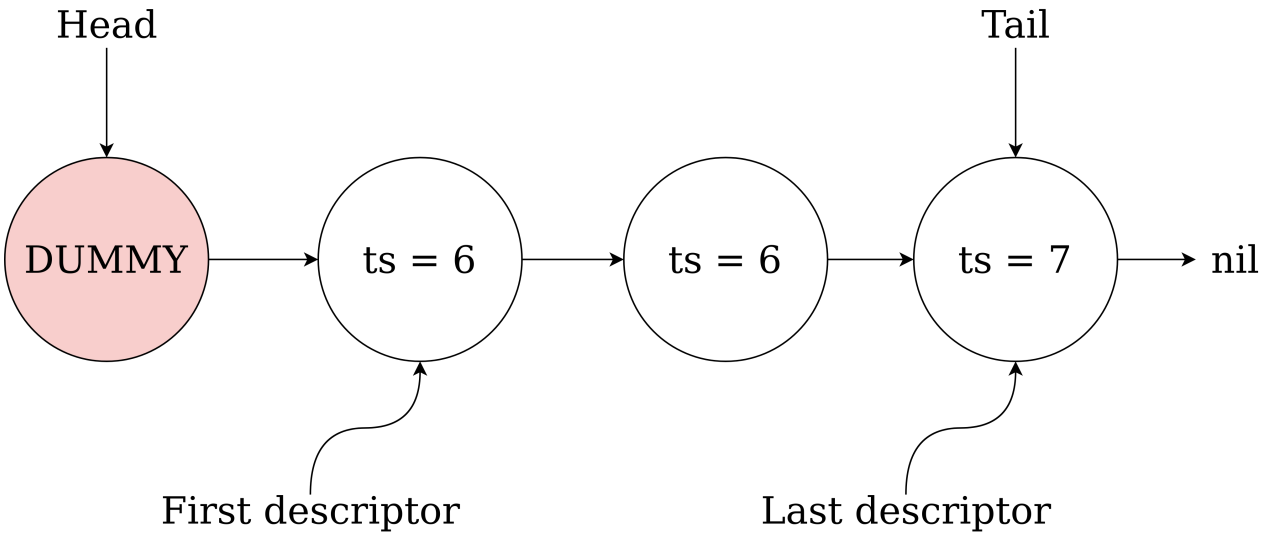


Figure 42 – The head node is a dummy node that does not store any data

2. If `cur_head` and `cur_tail` point to a single node, queue may be empty. In that case, to distinguish empty queue from non-empty, we read `cur_tail.Next`. Two possible situations can happen:
   — `cur_tail.Next = nil` (Fig. 43). In that case, the queue is empty and we finish the procedure, leaving the queue unmodified — we cannot remove the node from an empty queue. We can conclude that the descriptor was already removed by another process.

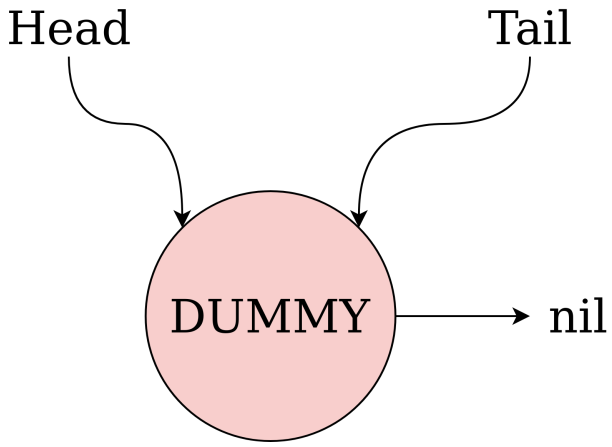


Figure 43 – The queue is empty

   — `cur_tail.Next` $\neq$ `nil` (Fig. 44). In that case,the queue is not empty since new descriptor is being concurrently inserted to the queue. We should help finish the insertion by trying to move `Queue.Tail` forward the same way that was described in Section 2.6.2.

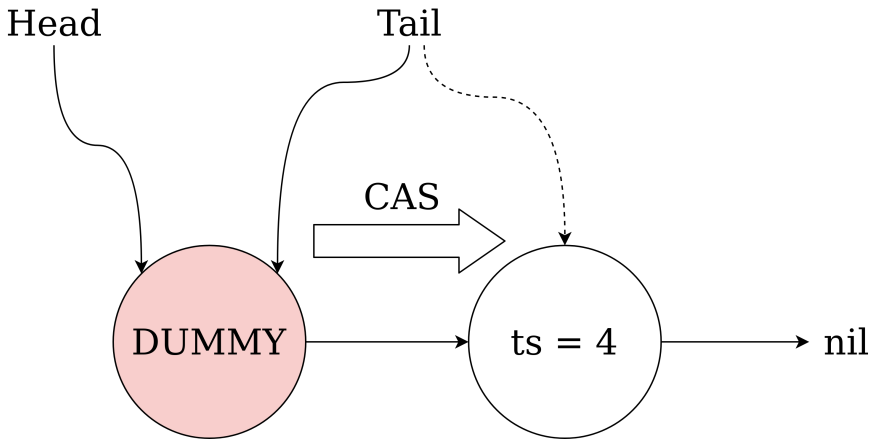


Figure 44 – The queue is not empty, we move the queue tail forward

After that, we simply retry the the `pop_if` procedure from the very beginning, i.e., from step (1).

3. If `cur_head` and `cur_tail` point to different nodes, we continue executing the operation. We begin with reading `first_timestamp` — timestamp of the first descriptor of the queue. The said timestamp can be acquired in `next_head := cur_head.Next` node (Fig. 45).

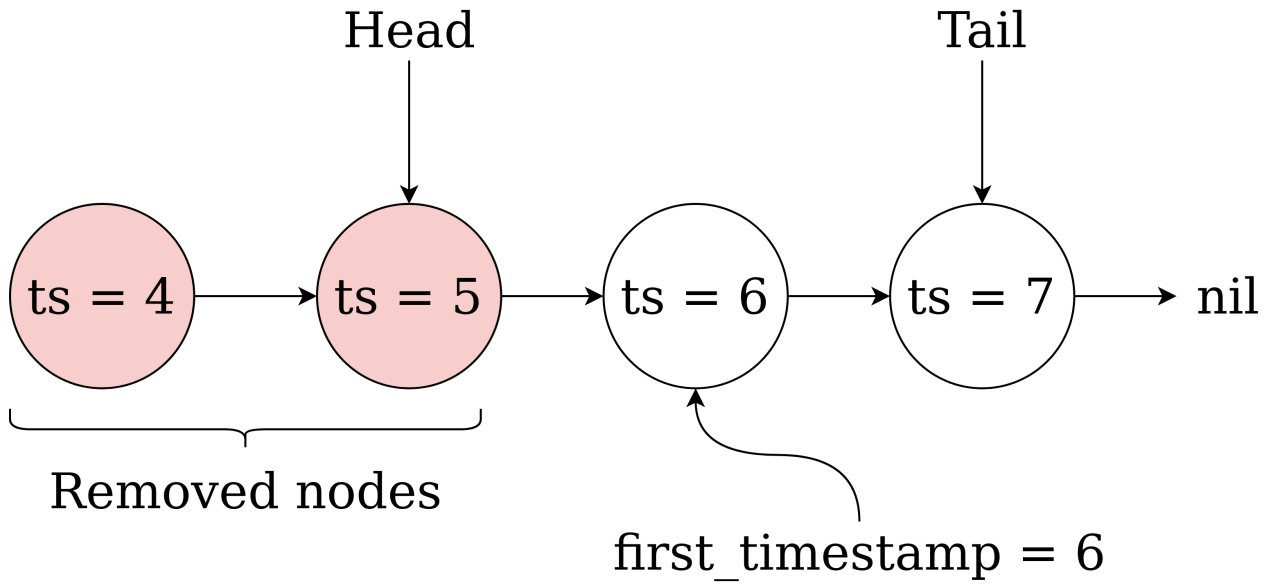


Figure 45 – Acquiring `first_timestamp`

— If `first_timestamp > TS` (e.g., is `TS = 4`), we can conclude that the descriptor with timestamp `TS` has already been removed by another process. Thus, we simply finish the `pop_if` procedure leaving the queue unmodified.

— Otherwise, `first_timestamp = TS`. Note that `first_timestamp` cannot be less than `TS` since, as was stated

above, we never try to remove a node, that has never been located at the head of the queue.

In that case, we try to move the queue head forward by executing `CAS(&Queue.Head, cur_head, next_head)`. If the `CAS` succeeds, we conclude that we have removed the requested descriptor from the queue and finish the operation. Otherwise, we conclude that some other process removed the requested descriptor and modified `Queue.Head`. In that case, we yet again simply finish the execution of the procedure.

Therefore, `pop_if` can be implemented in the following way (Listing 16):

```
1  fun pop_if(Queue, timestamp):
2      while true:
3          cur_head := Queue.Head
4          cur_tail := Queue.Tail
5          if cur_head = cur_tail:
6              next_tail := cur_tail.Next
7              if next_tail = nil:
8                  return
9              else:
10                 CAS(&Queue.Tail, cur_tail, next_tail)
11                 /* Retry the operation from the very beginning */
12         else:
13             next_head := cur_head.Next
14             first_timestamp := next_head.Data.Timestamp
15             if first_timestamp = timestamp:
16                 CAS(&Queue.Head, cur_head, next_head)
17             return
```

Listing 16 – Implementation of the `pop_if` procedure

2.6.5. Queues progress guarantees and implementation details

Note that all the queue operations described above are lock-free, just like in the original queue by Michael and Scott [27]. Indeed, the repetition of each procedure from the very beginning indicates that other process successfully executed its procedure, thus modifying the queue.

Of course, we can take other queue algorithms as a basis for our solution, not only the one proposed by Michael and Scott. For example, we can use `fetch-and-add` queue, proposed by Yang et al. [36] or practical wait-free queue, proposed by Kogan and Petrank [22]: `pop_if`, `push_if`, and

`push_acquire_timestamp` implementation principles remain the same. We use Michael-Scott queue only due to the simplicity of its implementation.

2.7. One possible tree balancing strategy

Until now, we considered only unbalanced trees. However, using unbalanced trees may result in $height \in \omega(\log N)$. Since most of the queries (e.g., `insert`, `remove` or `contains`) 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 subtree rebuilding and is similar to the balancing strategy proposed in [6]. The idea of this approach can be formulated the following way: when the number of modifications, applied to a particular subtree, exceeds a threshold, we completely rebuild that subtree, making it perfectly balanced (Fig 46).

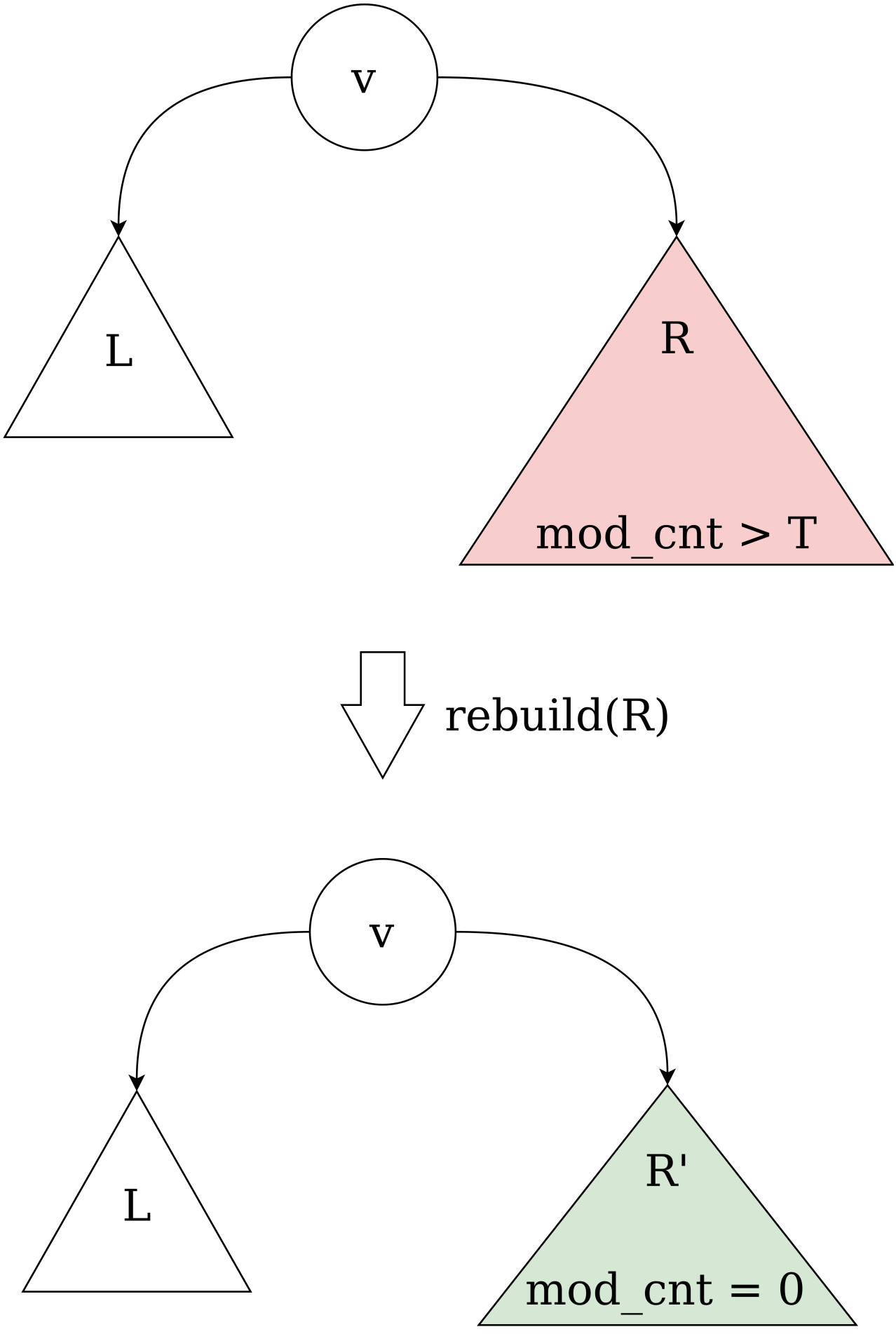


Figure 46 – Tree balancing via subtree rebuilding: when the number of modifications, applied to a subtree, exceeds a threshold, we rebuild the whole subtree

For each tree node we maintain `Mod_Cnt` — the number of modifications, applied to the subtree of this node. We store `Mod_Cnt` in the node state. Moreover, for each node we store an immutable number `Init_Sz` — 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 · Init_Sz`, where `K` is a predefined constant.

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

Consider node `v`, its parent `pv` and operation `Op`, that is being executed in `pv` and that should continue its execution in `v` subtree (and, therefore, its descriptor should be inserted to `v` queue). Before inserting `Op` to `v` queue and changing `v` state, we check whether `Mod_Cnt` in `v` will exceed the threshold after applying `Op` to `v` subtree (Fig. 47).

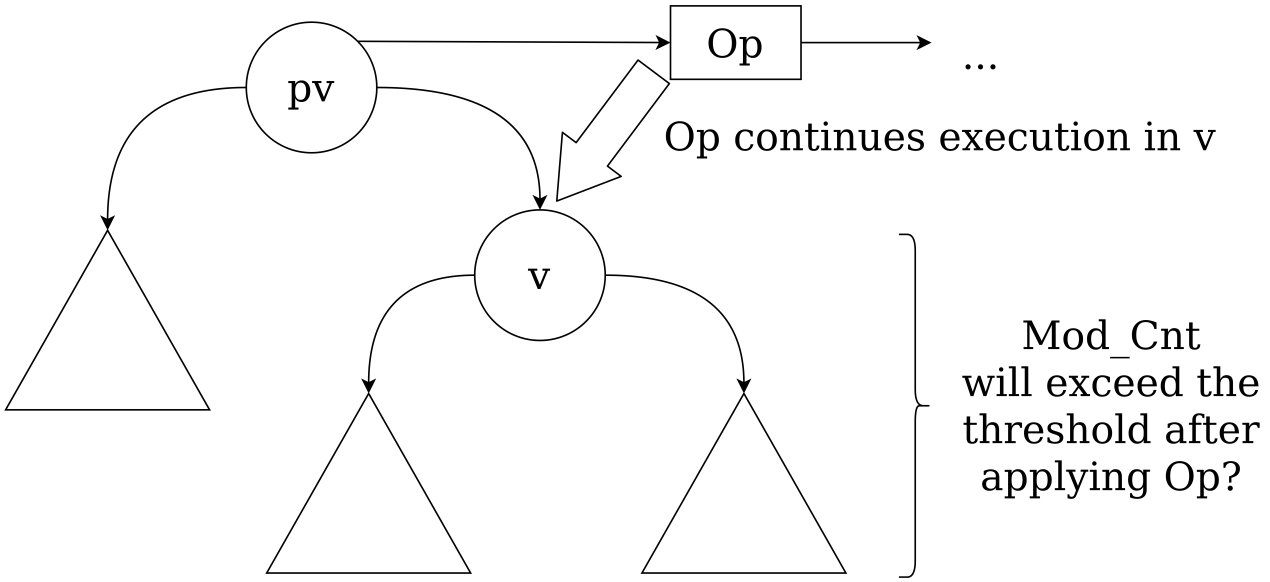


Figure 47 – We check whether the subtree of `v` should be rebuilt or not

The algorithm for the check whether the subtree of `v` should be rebuilt or not, is the following:

1. If `Op` is not a modifying operation (i.e. `Op` is a read-only operation), e.g., `contains` or `count`, we should not rebuild `v` subtree, since `Op` does not increase the number of modifications after being applied to `v` subtree.
2. Otherwise, atomically read `v` state.

3. If `Ts_Mod` $\geqslant$ `Op.Timestamp`, then `v` state has already been modified by `Op`. Thus, `Op` has already been counted in the number of modifications, applied to `v` subtree and we do not need to rebuild `v` subtree.
4. Otherwise, check `Mod_Cnt`. If `Mod_Cnt + 1` $<$ `K` $\cdot$ `init_sz`, then the application of `Op` does not lead to `Mod_Cnt` exceeding the threshold (and, thus, to `v` subtree being rebuilt). Therefore, we should only increment `Mod_Cnt`. The increment of `Mod_Cnt` should be performed atomically with all other state changes and should be performed as discussed in Section 2.5.
5. Otherwise, `v` subtree should be rebuilt.

Note, that `v` subtree can contain unfinished operations: their descriptors still reside in queues in `v` subtree (Fig. 48).

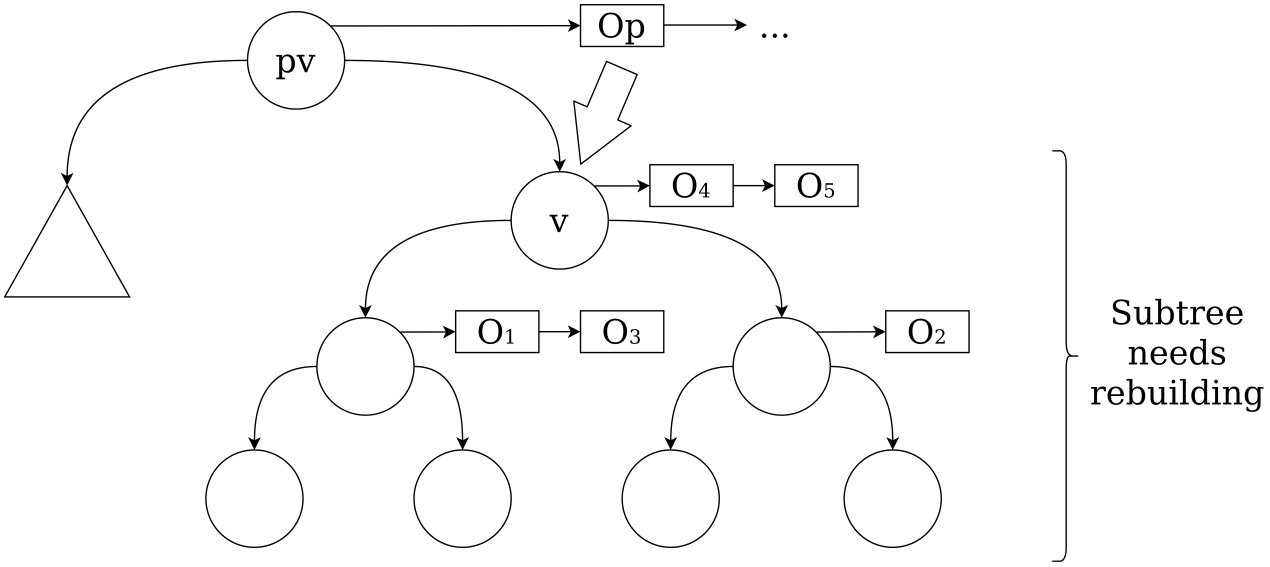


Figure 48 – Unfinished operations in the subtree, that should be rebuilt

We should finish all such unfinished operations before rebuilding the subtree. To do so, we traverse `v` subtree and in each node `c` $\in$ `subtree(v)` execute all operations, residing in `c` queue.

After that, we traverse `v` subtree, that contains no more unfinished operations, and collect all the data items (e.g. keys or key-value pairs) stored in it. After collecting them, we build a balanced subtree, containing all these data items (Fig. 49).

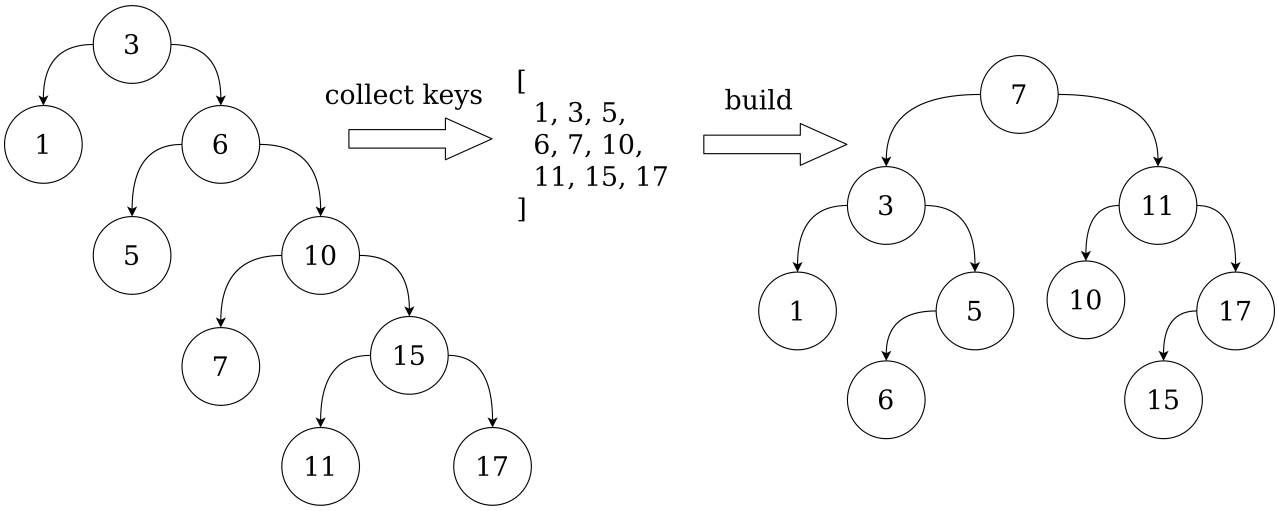


Figure 49 – Subtree rebuilding consists of collecting all data items (keys, in this example) in the old unbalanced subtree and building a new, balanced one

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 ⩾ Op.Timestamp`) can still modify the new subtree, but all the preceding operations (with `timestamp < Op.Timestamp`) cannot. Thus, we set `Ts_Mod = Op.Timestamp - 1`.

After that, we take `v'` — the root of the new subtree and try to modify the pointer that pointed at `v`, so that it starts to point at `v'`. For example, if `v` was the left child of `pv`, we execute `CAS(&pv.Left, v, v')`; if `v` was the right child of `pv`, we execute `CAS(&pv.Right, v, v')`.

Regardless of the return value of that `CAS`, we resume the execution of `Op` in `pv`: we modify `v'` state, insert `Op` descriptor to `v'` queue (here `v'` is the root of the rebuilt subtree) and remove `Op` descriptor from `pv` queue. Indeed, if `CAS` returned `true`, we conclude that we have successfully completed the rebuilding and can proceed with the execution of `Op` in `pv`. Otherwise, if `CAS` returned `false`, we conclude that the rebuilding was completed by some other process and it already modified the necessary child pointer. Yet again, in that case we proceed with the execution of `Op` in `pv`.

Note that new descriptors cannot appear in `v` subtree until the rebuild is completed, since new descriptors cannot be inserted to `v` queue until the execution of `Op` in `pv` is finished, according to the main invariant (Fig. 50). Since the execution of `Op` in `pv` includes rebuilding `v` subtree, all the later descriptors are inserted to the rebuilt subtree.

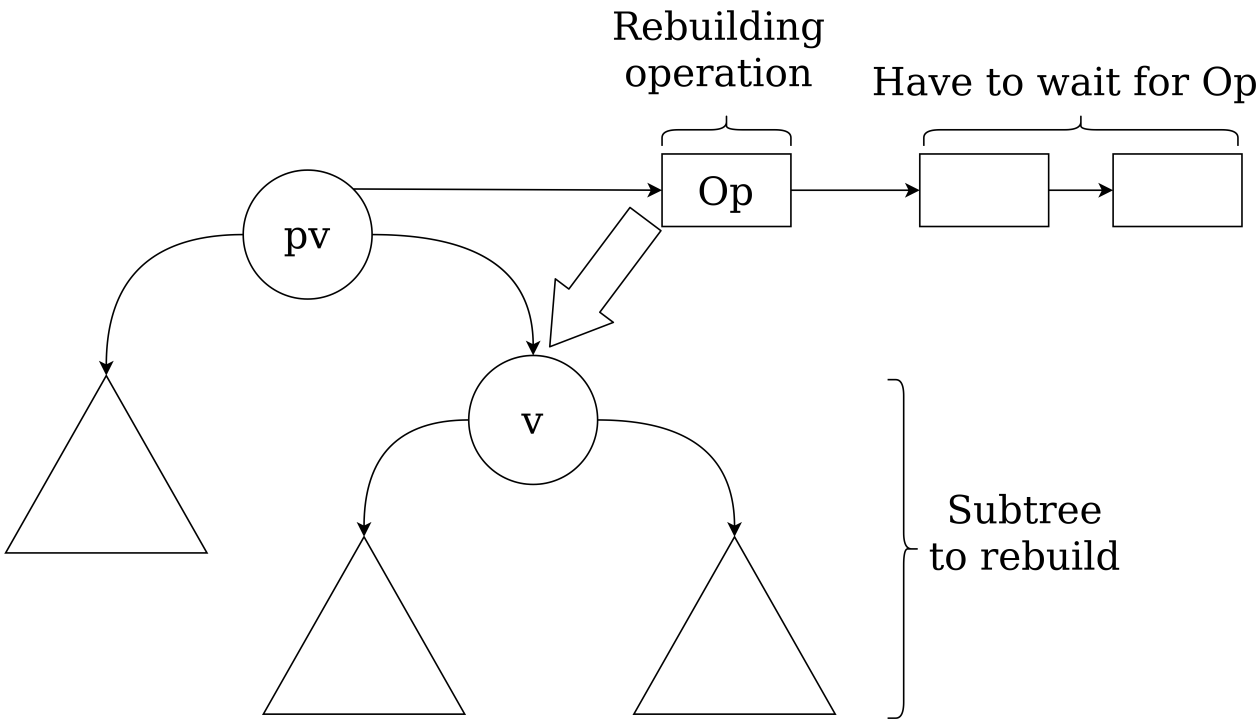


Figure 50 – New descriptors cannot appear in `v` subtree during the rebuilding procedure

Conclusions on Chapter 2

In this chapter we described the algorithm in general terms, without applying it to any particular data structure. We outlined the problems that we may face trying to apply the sequential algorithm for asymptotically efficient range queries execution in a concurrent setting. We introduced our solution to the aforementioned problems in queue ordering mechanism, providing operations linearization. We described the overall process of executing an operation on a tree, while also explaining the details of executing an operation in a single node. We studied both the `CAS-N`-based solution and the solution without using `CAS-N`, that relies on an explained concurrent queue implementation. Finally, we studied one possible tree balancing strategy (the one, based on subtree rebuilding), while leaving other balancing strategies for the future work.

CHAPTER 3. BINARY SEARCH TREE, SUPPORTING THE COUNT RANGE QUERY

3.1. Tree structure

As stated in Section 1.3, we implement the `count` query on an external binary search tree. We have three types of nodes in the tree (Listing. 17):

— `KeyNode` is a leaf node containing exactly one key.

— `EmptyNode` a leaf node with no key.

— `InternalNode`. Each internal node has exactly two children (left and right). Internal nodes do not store keys directly, they store only the information required for the query routing. In our case, each internal node stores `Right_Subtree_Min` value, as discussed in Section 1.3. All keys less than `Right_Subtree_Min` are stored in the left subtree, while all keys greater than or equal to `Right_Subtree_Min` are stored in the right subtree. Each internal node maintains the state: 1) `Ts_Mod`, and 2) current subtree size (see Section 2.5 for discussion on state maintenance algorithm). Moreover, each internal node stores `Mod_Cnt` and immutable value `Init_Sz` (see Section 2.7 for details on the rebuilding algorithm).

```
type KeyNode = struct {
    Key: KeyType
}

type EmptyNode = struct {}

type InternalNodeState = struct {
    Ts_Mod: Timestamp,
    Mod_Cnt: uint,
    Size: uint
}

type InternalNode = struct {
    S_Ptr: InternalNodeState*,
    Left: Node*,
    Right: Node*,
    Right_Subtree_Min: KeyType
}
```

Listing 17 – Tree nodes definition

3.2. `insert` and `remove` operations

Execution of a scalar (`insert`, `remove`, `contains`) operation on a key `k` begins with traversing the tree downwards, from the root to the appro-

priate leaf, where key `k` should exist. In each visited internal node if `k < node.Right_Subtree_Min`, we continue the traversal in `node.Left`, otherwise — in `node.Right`.

As stated in Section 1.3, while executing `insert` operation we increase by one the subtree size of each visited node on the path from the root to the appropriate leaf. Similarly, while executing `remove` operation we decrease by one the subtree size of each visited node on the path from the root to the appropriate leaf. See Section 2.5 for an explanation of how the state of the node can be changed.

Consider an external binary search tree that contains key `7` (Fig. 51).

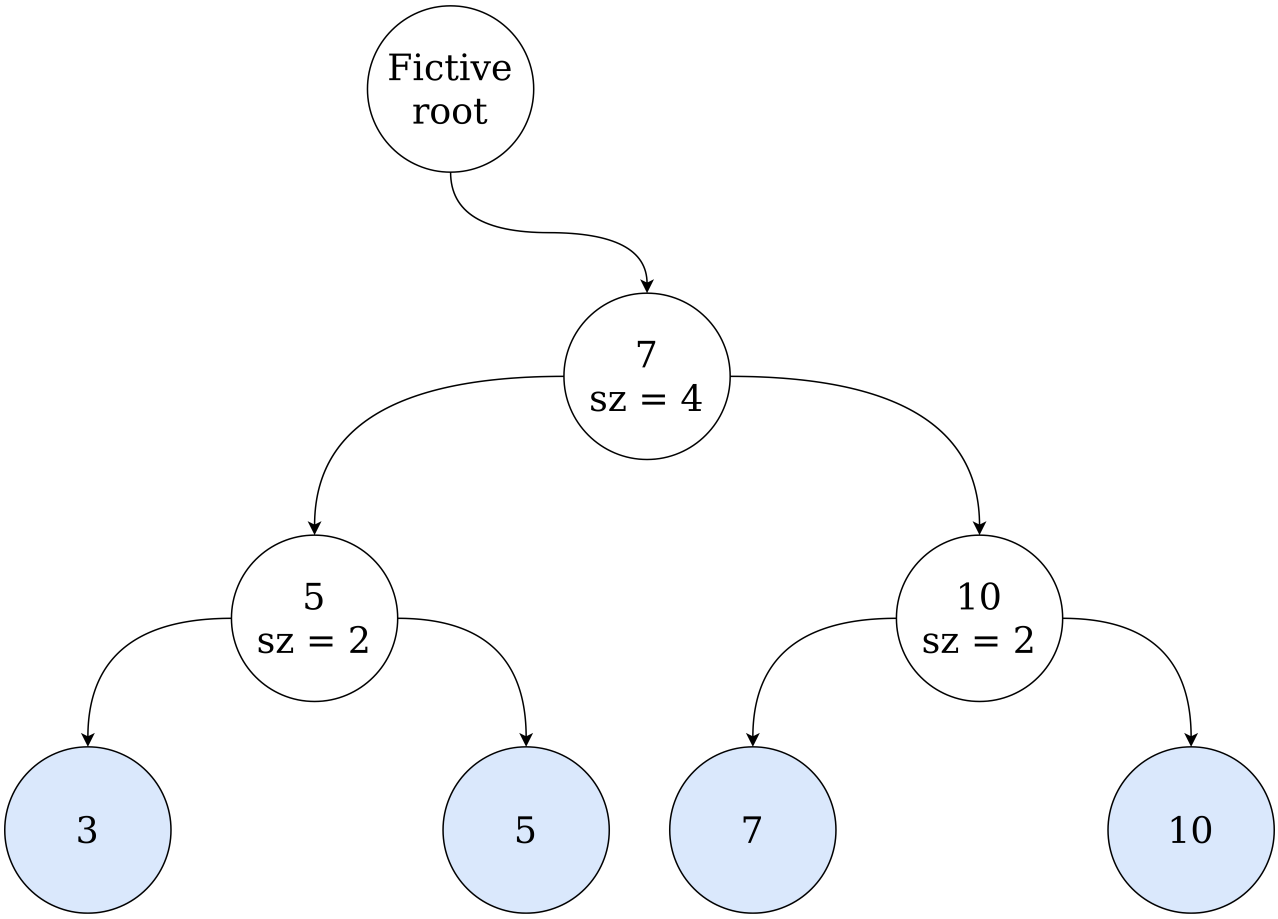


Figure 51 – External binary search tree that contains key `7`

Suppose we are going to insert key `7` at this tree. If we increase by one the subtree size of each traversed inner node, we end up with incorrect subtree sizes: subtree sizes of visited inner nodes become greater than they should be, since `7` already existed in the tree (Fig. 52).

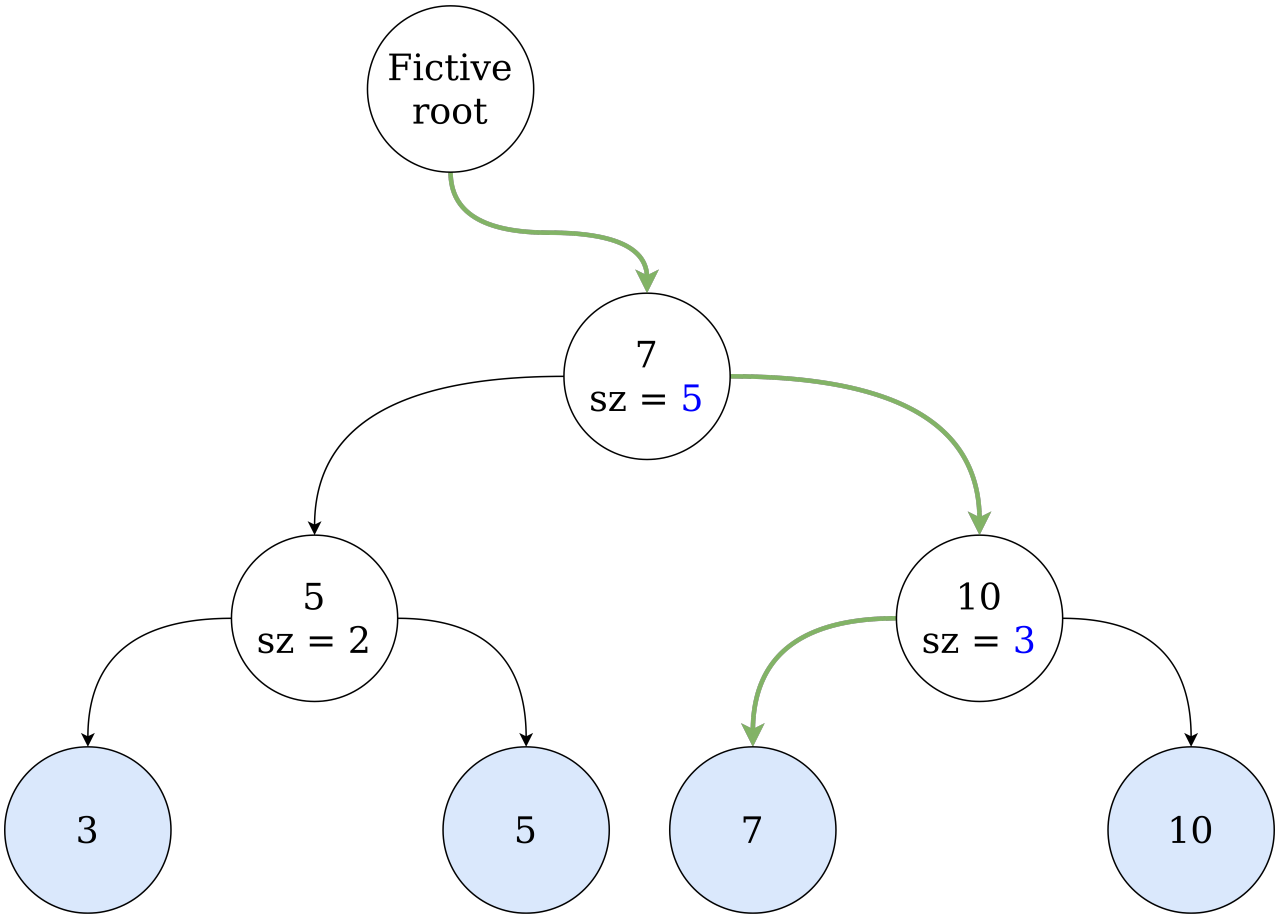


Figure 52 – `insert(7)` to the external binary search tree containing key `7`

We face similar problem when someone removing a key `k`, that does not exist in the tree: subtree sizes of visited inner nodes become less than they should be.

Thus, we should not execute `insert(k)` operation if key `k` already exists in the tree. Also, we should not execute `remove(k)` operation if key `k` does not exist in the tree.

The initiator process, after inserting the descriptor of `insert(k)` or `remove(k)`, should check whether key `k` exist in the tree (Fig. 53). We describe how to implement this check in Section 3.3.

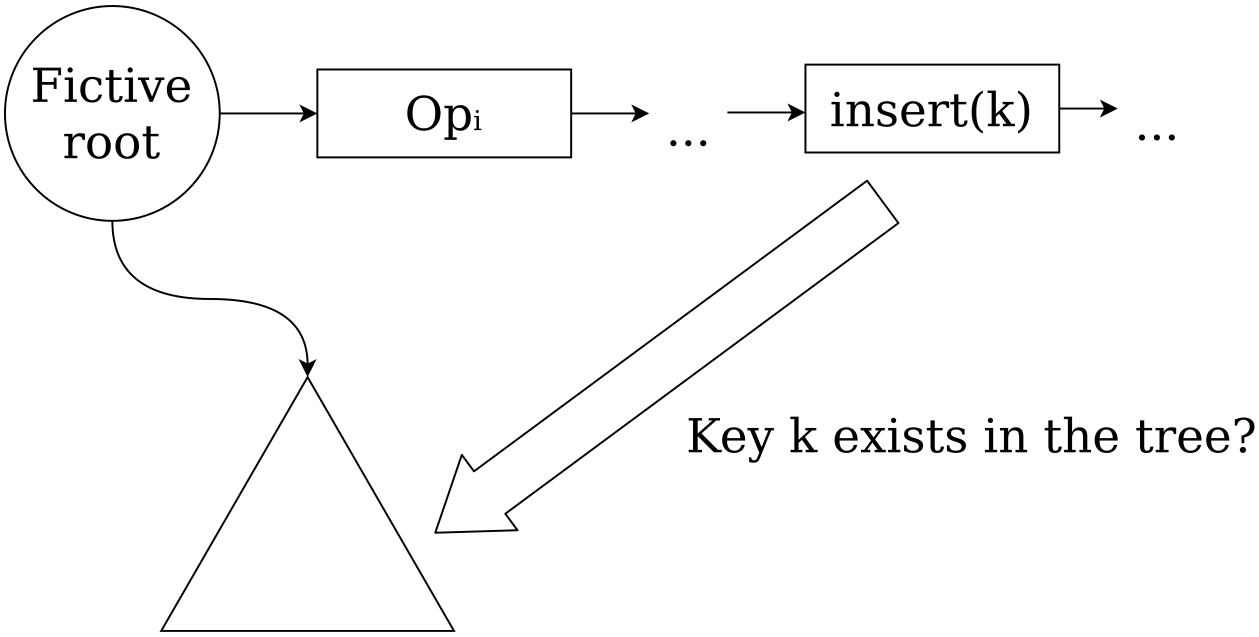


Figure 53 – Check whether key `k` exist in the tree

After determining the key existence, the initiator process makes a decision whether its initiated operation should be executed or not:

— `insert(k)` operation should be executed if key `k` does not exist in the set.

— `remove(k)` operation should be executed if key `k` exists in the set.

After making a decision, the initiator process stores it in `descriptor.Status` field. `descriptor.Status` should be set to `true` if the operation should be executed and `false`, otherwise.

Note that for each `insert` or `remove` descriptor from the fictive root queue the check is done in parallel by the process that initiated that operation.

Suppose `Op` is the operation, corresponding to the descriptor from the head of the fictive root queue. The process of executing `Op` in the fictive root node differs a bit from an ordinary execution process in a node:

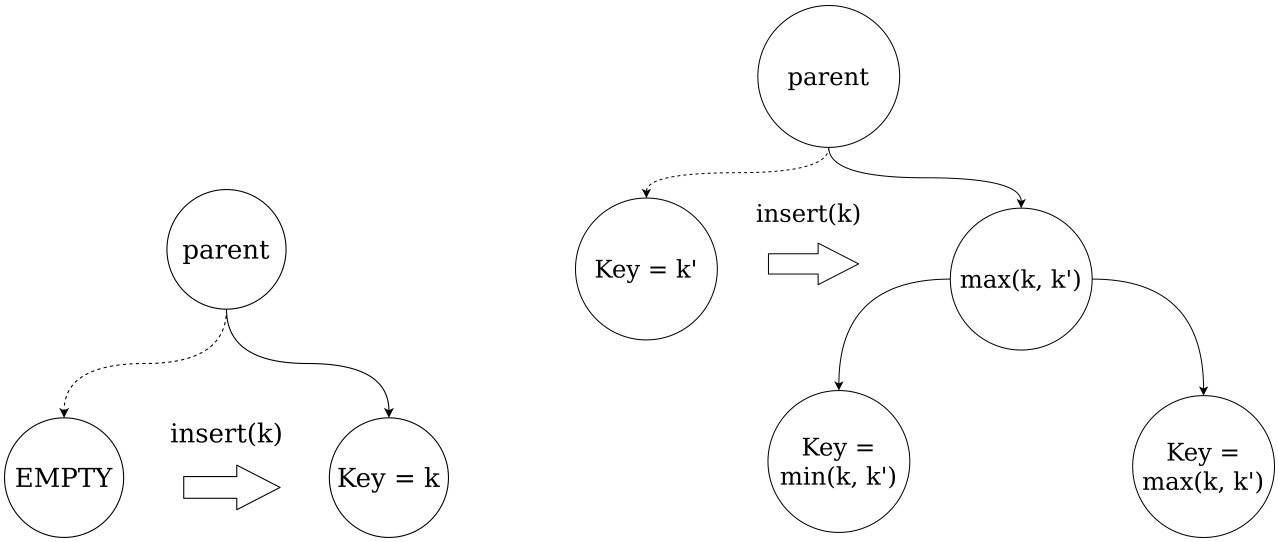
1. If `Op` is `contains` or `count` operation, it should be propagated directly to the real root queue without extra checks;
2. Otherwise (if `Op` is `insert` or `remove`), we check `Op.Status` field;
3. If `Op.Status` has not been set yet (i.e. it equals to `nil`), we conclude that the initiator process has not yet checked whether the key exists in the tree. Thus, it is not known yet whether `Op` should be executed or not. In that case, we help the initiator process make the decision: we determine whether `Op.Key` exist in the tree and try to set `Op.Status` ourselves, if

`Op.Status` is not set yet (other helper processes or the initiator process may set it before us).

4. After the `Op.Status` is set (either by us, or by other helper process or by the initiator process) we can finish the execution of `Op` in the fictive root node:
   — If `Op.Status = true`, we should execute `Op` in the fictive root node in an ordinary way: rebuild the tree if necessary (see Section 2.7 for details), modify the state of the real root, insert `Op` descriptor to the real root queue and remove `Op` from the fictive root queue.
   — Otherwise, `Op` should not be executed at all, since it does not affect the tree (because an insertion of an existing key and a removal of a non-existing one do not modify the tree). Thus, `Op` descriptor should simply be removed from the fictive root queue.

When executing `insert(k)` operation, we may end up in one of the two possible situations (Fig. 54):

— After traversing the tree, we encounter an `EmptyNode` leaf. In that case, we simply replace that leaf with new `KeyNode`, storing key `k` (Fig. 54a).
— After traversing the tree, we encounter a `KeyNode` leaf, storing key `k′` $\neq$ `k`. In that case, we simply replace that leaf with a new subtree, consisting of three nodes (Fig. 54b). The root of this subtree is an `InternalNode`. Its left child is a `KeyNode` that stores the least of two keys, `k` and `k′`. Thus, the left child stores a key `min(k, k′)`. Similarly, the right child is a `KeyNode` that stores the largest of two keys, `k` and `k′`. Thus, the right child stores a key `max(k, k′)`. Since the right subtree of the `InnerNode` contains only `max(k, k′)` key, we set `Right_Subtree_Min = max(k, k′)` in the root of the new subtree.



(a) Execution of `insert(k)` operation on an `EmptyNode` leaf.

(b) Execution of `insert(k)` operation on a `KeyNode` leaf.

Figure 54 – Execution of `insert(k)` operation on different leaves

When executing `remove(k)` operation, we find the `KeyNode` leaf, storing key `k`. After that, we replace that `KeyNode` with an `EmptyNode` (Fig. 55).

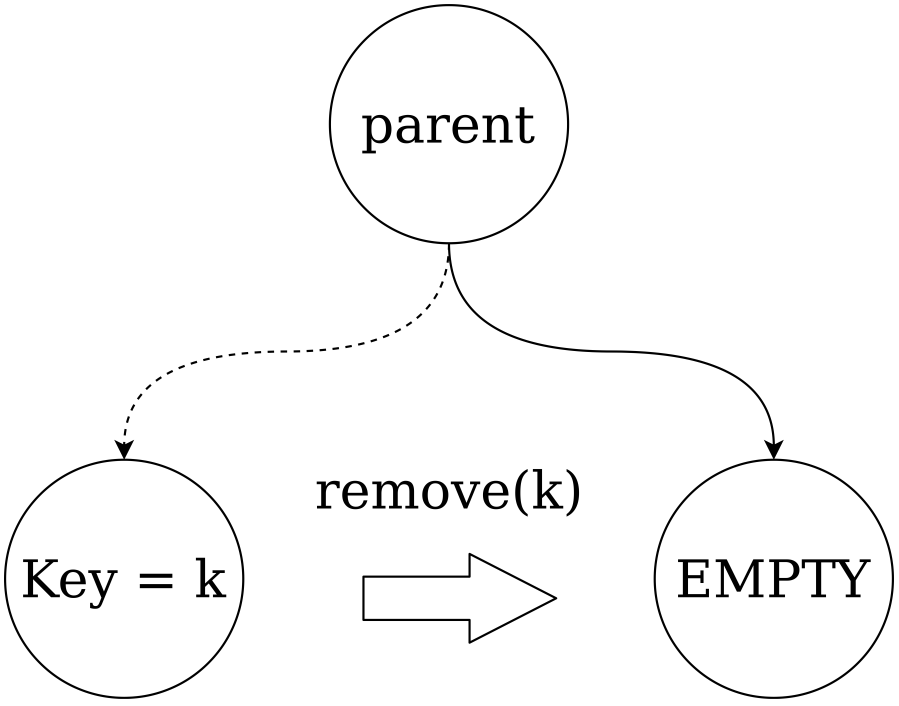


Figure 55 – Execution of `remove(k)` operation

Our `remove` implementation can create some amount of empty nodes in the tree: these nodes do not store any data and just waste space. However, the rebuilding procedure will get rid of them and the rebuilt subtree will consist only of `KeyNode` leaves and internal nodes.

If these tree transformations are implemented improperly, stalled processes may break the tree structure. Consider the tree, consisting of an inner node `v` and two its leaf children, storing keys `3` and `5`. Suppose descriptor of operation `remove(5)` is located at the head of `v` queue. The second descriptor in `v` queue corresponds to the operation `insert(5)` (Fig 56a). Trivially, after executing `remove(5)` and `insert(5)` after that the tree should remain unmodified.

Consider the following sequence of actions:

1. Process `P` reads `remove(5)` descriptor from the head of `v` queue (Fig 56b).
2. Process `P` is suspended by the OS.
3. Process `R` reads `remove(5)` descriptor from the head of `v` queue and executes the operation, replacing `v` right child with `EmptyNode` (Fig 56c).
4. Process `R` reads `insert(5)` descriptor from the head of `v` queue and executes the operation, replacing `v` right child with `KeyNode{ Key = 5 }` (Fig 56d).
5. Process `P` is resumed by the OS.
6. Process `P` executes `remove(5)` in `v`, replacing `v` right child with `EmptyNode` (Fig 56e).

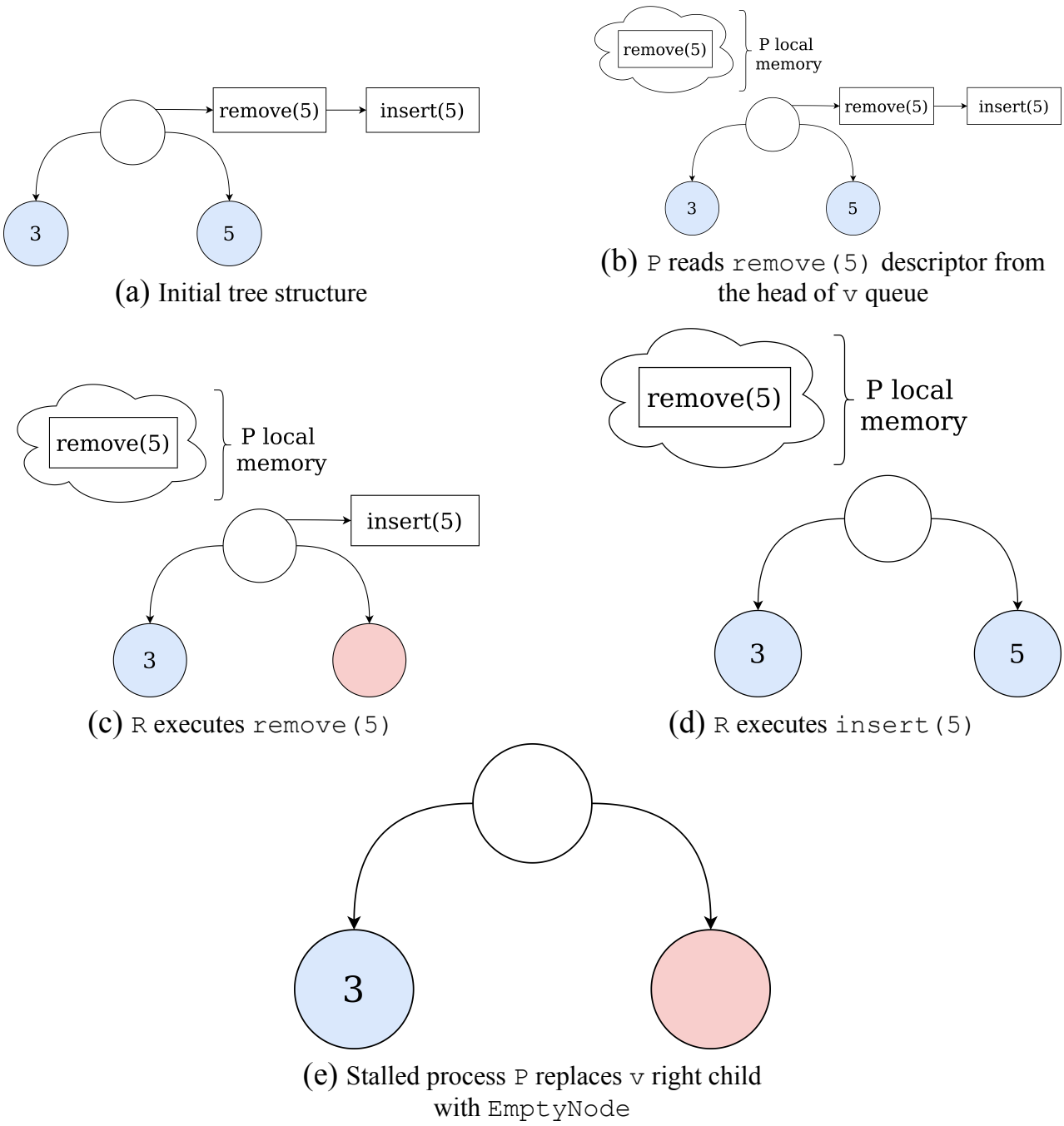


Figure 56 – Stalled processes can break the tree structure

Therefore, after the execution of `remove(5)` and `insert(5)` the tree is modified, instead of remaining unmodified.

We can solve this problem by augmenting each tree node with `Creation_Ts` — timestamp of the operation, that created that node. If operation `Op` wants to replace node `v` with some other node (e.g., replace `EmptyNode` with `KeyNode`), `Op` first checks whether `v.Creation_Ts` $\geqslant$ `Op.Timestamp` holds. If so, `v` was created by `Op` or some later operation and `Op` should not replace `v`. Otherwise, we try to replace `v` using `CAS(&child_ptr, v, new_node)`.

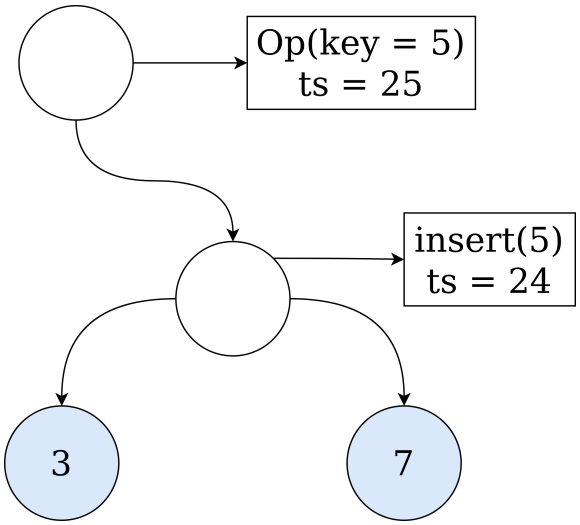
No matter what was the result of the `CAS`, we may finish the execution of `Op`. Indeed, if the `CAS` returns `true`, we conclude that we have replaced the node successfully. Otherwise, we conclude that some other process has replaced the node while helping `Op`.

3.3. Determining the existence of a key

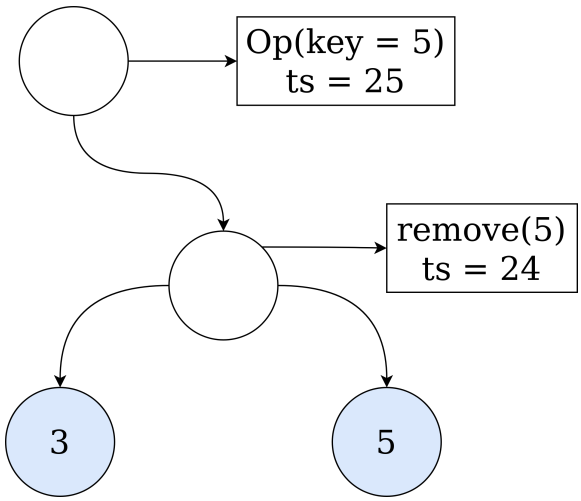
Suppose descriptor of operation `Op`, that is either `insert(k)` or `remove(k)`, is located in the fictive root queue and we need to check, whether key `k` exist in the tree, to determine, whether `Op` should be executed or not.

Simply checking whether the tree contains `KeyNode { Key = k }` is not sufficient (Fig. 57). Indeed, consider a tree, where key `k` does not exist physically: there is no node, storing key `k` in the tree. However, descriptor of `insert(k)` exists in the queue of some tree node. Moreover, timestamp of `insert(k)` is less than `Op.Timestamp` (Fig. 57a). Thus, `Op` should consider key `k` as existing in the tree, since that `insert(k)` operation should linearize before `Op`, according to its timestamp.

Similarly, even if key `k` is physically stored in some tree leaf, it may not exist in the tree logically, if descriptor of `remove(k)` with timestamp less than `Op.Timestamp` exist somewhere in the tree (Fig. 57b).



(a) Key `5` does not exist in the tree physically. However, from `Op` point of view, it exists in the tree logically.



(b) Key `5` exists in the tree physically. However, from `Op` point of view, it does not exist in the tree logically.

Figure 57 – A simple check whether the tree contains `KeyNode { Key = k }` is insufficient

We may formulate the algorithm to check whether key `k` logically exists in the tree at timestamp `Op.Timestamp` the following way:

1. Collect all descriptors, corresponding to operations `insert(k)` and `remove(k)` with timestamps less than `Op.Timestamp`.
2. If we have not collected anything, traverse the tree, looking for key `k`.
   — If we have reached `EmptyNode`, then key `k` does not exist in the tree.
   — If we have reached `KeyNode`, storing key `k'` $\neq$ `k`, then key `k` does not exist in the tree.
   — Otherwise (we have reached `KeyNode`, storing key `k`), then key `k` exist in the tree.
3. Otherwise we consider descriptor `D` — the collected descriptor with the maximal timestamp.
   — If `D` corresponds to `insert(k)` operation, we conclude that the key `k` exists in the tree. Indeed, operation `D` may either be: 1) executed successfully — thus, key `k` is inserted to the tree by that operation; 2) discarded, because key `k` already exists in the tree. In either case, key `k` exists in the tree after applying the operation with timestamp `D.Timestamp` and there is no `remove(k)` operation with timestamp from the semi-interval `[D.Timestamp; Op.Timestamp)`. Thus, from `Op` point of view the key `k` still exist in the tree.
   — If `D` corresponds to `remove(k)` operation, we conclude that the key `k` does not exist in the tree (can be proven similarly to the previous case).

The main question is: where we can find descriptors of operations `insert(k)` and `remove(k)`? The answer is quite simple: remember, that the process, executing such an operation on key `k`, visit only nodes on the path from the fictive root to the leaf, that would contain key `k`, if it existed in the tree. Thus, we should look for descriptors of `insert(k)` and `remove(k)` only in queues of such nodes.

Suppose in the queue of a node `v` we have found a descriptor `D`, denoting either `insert(k)` or `remove(k)` operation. In that case, there is no need to look for `insert(k)` and `remove(k)` descriptors in `v` subtree. Indeed, descriptors are propagated downwards in the order, determined by their timestamps. Thus, timestamps of descriptors, found in `v` subtree, will be less than `D.Timestamp`. Since we need to consider only the highest-timestamped descriptor in order to determine

the existence of key `k`, we need not to consider descriptors, timestamps of which are guaranteed to be less than `D.Timestamp`.

Note, that each considered queue should be traversed till the very tail. Indeed, suppose we find descriptor `D`, denoting either `insert(k)` or `remove(k)` operation in `v` queue. The very same `v` queue could contain higher-timestamped descriptor of `remove(k)` or `insert(k)`, that would change our opinion on whether key `k` exists in the set. Such higher-timestamped descriptor may be located closer to the tail of `v` queue, thus we should keep traversing `v` queue even after we have found first `insert(k)` or `remove(k)` descriptor.

However, suppose in `v` queue we find descriptor `D` with `D.Timestamp` $\geqslant$ `Op.Timestamp`. All further (located closer to the tail) descriptors from `v` queue will have even greater timestamp. Since we should consider only descriptors, timestamps of which are less than `Op.Timestamp` we do not need to consider any further descriptor from `v` queue. Thus, we may finish traversing `v` queue as soon as we encounter descriptor `D` with `D.Timestamp` $\geqslant$ `Op.Timestamp`.

Note, that if in the traversal process we encounter either:

— Node with `Creation_Ts` $\geqslant$ `Op.Timestamp`;

— Inner node with `Mod_Ts` $\geqslant$ `Op.Timestamp`;

— Descriptor `D`, such that `D.Timestamp` $\geqslant$ `Op.Timestamp`, if `D` is located not in the fictive root queue;

we conclude that the decision, whether `Op` should be executed or not, was already done, since at least one descriptor with `Timestamp` $\geqslant$ `Op.Timestamp` was propagated downwards from the fictive root node. We can learn the decision, whether `Op` should be executed or not, by reading `Op.Status` field.

The algorithm to check the existence is shown on the following pseudocode (Listing 18):

```
1  /*
2  Searches for descriptors of insert(k) and remove(k)
3  in the fictive root queue
4  */
5  fun process_topmost_queue(Root_Queue, ts, key):
6      result := nil
7      for descriptor ← Root_Queue:
8          if descriptor.Timestamp ⩾ ts:
9              break
10         case descriptor of
11         | InsertDescriptor { Key = key } →
```

```
            result ← true
        | RemoveDescriptor { Key = key } →
            result ← false
        | _ →
            continue
    return result


/*
Searches for descriptors of insert(k) and remove(k)
in a queue, not located in the fictive root node
*/
fun process_queue(Non_Root_Queue, ts, key):
    result := nil
    for descriptor ← Non_Root_Queue:
        if descriptor.Timestamp ⩾ ts:
            /*
            Decision whether the operation Op should
            be executed or not has already been made
            by another process. We use special value
            ANSWER_NOT_NEEDED to tell the caller
            about this situation.
            */
            return ANSWER_NOT_NEEDED
        case descriptor of
        | InsertDescriptor { Key = key } →
            result ← true
        | RemoveDescriptor { Key = key } →
            result ← false
        | _ →
            continue
    return result

fun key_exists(Set, ts, key):
    res := process_topmost_queue(Set.Root.Queue, ts, key)
    if res ≠ nil:
        return res
    cur_node := Set.Root.Child /* the real tree root */
    while true:
        case cur_node of
        | EmptyNode →
            if cur_node.Creation_Ts ⩾ ts:
                return ANSWER_NOT_NEEDED
            else:
                return false
        | KeyNode →
            if cur_node.Creation_Ts ⩾ ts:
                return ANSWER_NOT_NEEDED
```

```
60              else:
61                  return cur_node.Key = key
62          | InnerNode →
63              cur_state := cur_node.S_Ptr
64              if cur_state.Mod_Ts ⩾ ts:
65                  return ANSWER_NOT_NEEDED
66              res := process_queue(cur_node.Queue, ts, key)
67              if res ≠ nil:
68                  /* res ∈ {ANSWER_NOT_NEEDED, true, false} */
69                  return res
70              elif key < cur_node.Right_Subtree_Min:
71                  cur_node ← cur_node.Left
72              else:
73                  cur_node ← cur_node.Right
```

Listing 18 – Learning, whether key `k` exists in the tree

3.4. Executing the `count` query

3.4.1. Query execution algorithm

Now we explain how to implement the `count` query according to the algorithm, described in Section 1.3.

The result of the `count` query is an integer. In a node `v` we do the following:

— Add some value to the result;

— Continue the execution in some of `v` children;

`count_both_borders(node, min, max)` is executed in the following way:

— If `node` is a `KeyNode`, we check whether `min` $\leqslant$ `node.Key` $\leqslant$ `max` holds. If so, we add `1` to the result, otherwise, we add `0` to the result.

— If `node` is an `EmptyNode`, we add `0` to the result.

— If `min` $\geqslant$ `node.Right_Subtree_Min`, we add `0` to the result and continue the execution in `node.Right`.

— If `max` $<$ `node.Right_Subtree_Min`, we add `0` to the result and continue the execution in `node.Left`.

— Otherwise, `min` $<$ `node.Right_Subtree_Min` $\leqslant$ `max`. In that case, we add `0` to the result, execute `count_left_border` in `node.Left` and execute `count_right_border` in `node.Right`.

`count_left_border(node, min)` is executed in the following way:

— If `node` is a `KeyNode`, we check whether `node.Key` $\geqslant$ `min` holds. If so, we add `1` to the result, otherwise, we add `0` to the result.
— If `node` is an `EmptyNode`, we add `0` to the result.
— If `min` $\geqslant$ `node.Right_Subtree_Min`, we add `0` to the result and continue the execution in `node.Right`.
— Otherwise, `min` $<$ `node.Right_Subtree_Min`. In that case, we get the size of the right subtree:
  – If `node.Right` is an `EmptyNode`, its size is `0`;
  – If `node.Right` is a `KeyNode`, its size is `1`;
  – If `node.Right` is an inner node we can read its size from the current state (`node.Right.S_Ptr.Size`);

  After that, we add right subtree size to the result and continue the execution in `node.Left`.

`count_right_border(node, max)` is executed the following way:
— If `node` is a `KeyNode`, we check whether `node.Key` $\leqslant$ `max` holds. If so, we add `1` to the result, otherwise, we add `0` to the result.
— If `node` is an `EmptyNode`, we add `0` to the result;
— If `max` $<$ `node.Right_Subtree_Min`, we add `0` to the result and continue the execution in `node.Left`.
— Otherwise, `max` $\geqslant$ `node.Right_Subtree_Min`. In that case, we add the size of the left subtree (this size can be calculated similarly to the previous case) to the result and continue the execution in `node.Right`.

The main question is: how can we add some value to the result, given that multiple processes can try to do it concurrently? The main problem is yet again adding the value to the result exactly once. Suppose processes `P` and `R` are both executing the `count` query in node `v`:
— `P` determines, that `x` should be added to the result.
— `R` determines, that `x` should be added to the result.
— `P` adds `x` to the result.
— `R` adds `x` to the result.

Therefore, the value `x` is added to the result twice, therefore the result is incorrect.

We studied two solutions for that problem: the one based on CAS-2 operation [39] (see Section 3.4.2) and the one without using CAS-2 (see Section 3.4.3).

3.4.2. Using CAS-2

We augment each count descriptor with Used boolean field. Suppose count descriptor D is located in node v. In that case, D.Used may store either:

— true, if the value from node v was already added to the result.

— false, if the value from node v was not added to the result yet.

We add the value x, corresponding to the node v, to the result the following way:

1. Read cur_res — the current value of the result.
2. Read D.Used field. If it contains true, we conclude that some other process has already added x to the result, thus, we may finish. Otherwise, we proceed to the next step.
3. We try to atomically change D.Used to true and increment the result using CAS-2([&D.Used, &Result], [false, cur_res], [true, cur_res + x]).
4. If the CAS-2 returns true, we successfully incremented the result and may finish.
5. Otherwise, we retry the addition from step (1). Note, that the CAS-2 may return false in the two possible situations:
   — Another process has added the value x to the result. In this case, D.Used has been set to true and we simply finish on step (2) after the retry.
   — Another process incremented the result, but the value x has not been added to the result yet. In this case, we try to perform the addition one more time.

When propagating descriptor D downwards, we do not insert D itself to child queues: instead, we insert copy of D, with the Used field set to false, to child queues. Indeed, the Used field should be set to false to allow adding values, corresponding to child nodes, to the result.

Since we always create a copy of a count descriptor when propagating it downwards, multiple descriptors may correspond to a single count operation. In all such descriptors the result pointer must point to the same memory location

(Fig. 58) to allow calculation of the result value based on multiple descriptors, corresponding to the same `count` operation.

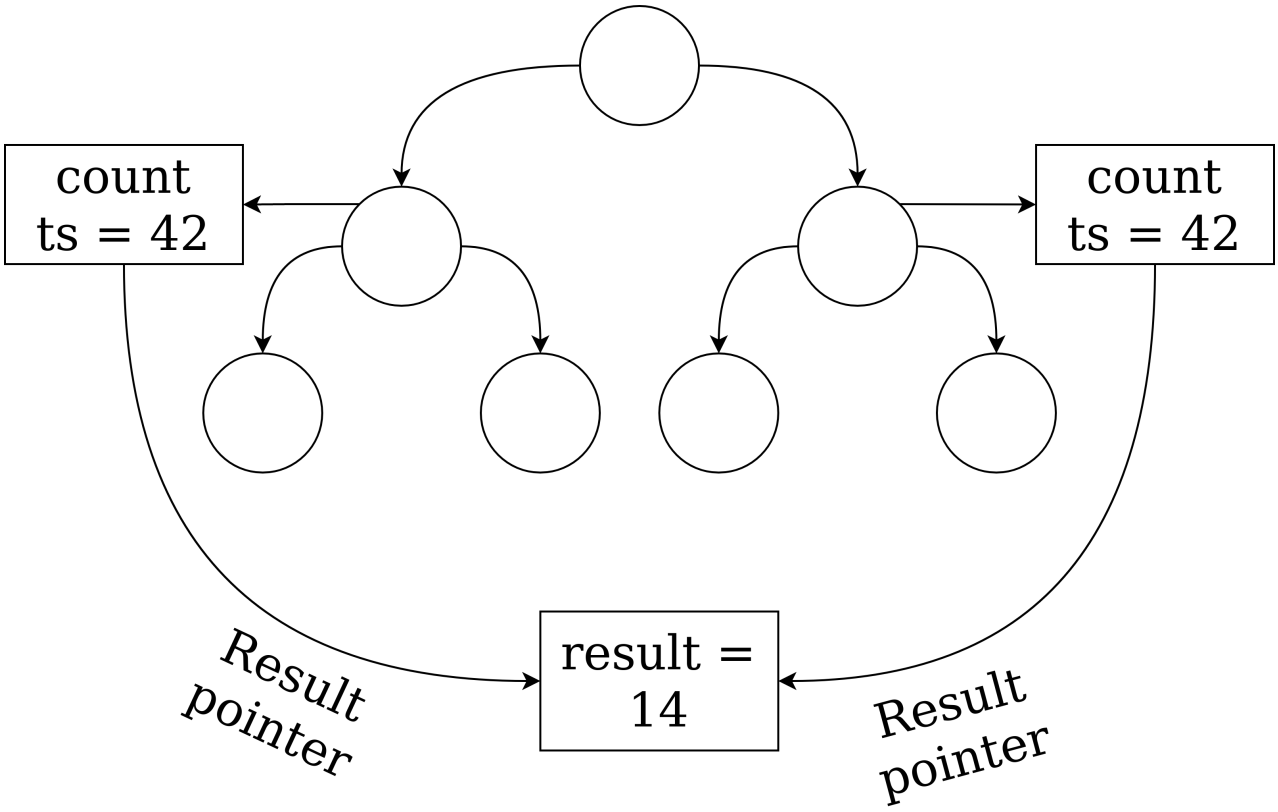


Figure 58 – Multiple descriptors, corresponding to a single `count` operation, should point to the same memory location of the result.

3.4.3. Without using `CAS-2`

As stated in Section 2.5, despite `CAS-2` is a powerful concurrent primitive, almost no CPUs supports it in the hardware, while software implementations of `CAS-2` usually suffer from poor performance due to heap allocations and additional indirection. Thus, we found another way to execute `count`.

We augment each tree node with an identifier, stored in the `Id` field. Node receives its identifier at the creation moment and the node identifier does not change throughout the node lifetime.

Distinct nodes must have unequal identifiers, i.e., node identifiers must be unique. We can achieve that property in multiple different ways, including:

— Use GUID or UUID [44] generation algorithm to generate node identifiers.

— Use global register `Max_Id`, supporting fetch-and-add [40] operation. Node identifier can be acquired using `new_node.Id := FAA(&Max_Id, 1)` operation.

Both these approaches guarantee, that node identifiers are unique.

We may use node identifiers to execute `count` operations without using `CAS-2`.

Instead of a single integer value, the `count` result will consist of two data structures:

— Set `Visited`, filled with identifiers of nodes that we visited during the execution;

— Map `Counted`, filled with identifiers of nodes, for which the value to add to the result is known, along with the values itself.

Before inserting a `count` descriptor to a node `v` queue, we add `v.Id` to the `Visited` set.

Before removing a `count` descriptor from a node `v` queue we try to add `v.Id` along with a value `x`, corresponding to the node `v`, to the `Counted` map. If key `v.Id` already exists in the `Counted` map, we left the `Counted` map unmodified, without changing the value, associated with `v`.

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` queue;
2. Process `P` reads `D` from the head of `v` queue;
3. Process `P` is suspended by the OS;
4. Process `R` reads `D` from the head of `v` queue;
5. Process `R` determines that the size of `v` left subtree should be added to the result;
6. Process `R` reads `v.Left.S_Ptr.Size` value (say, it equals to 5) and adds key-value pair ⟨ `v.Id, 5` ⟩ to the `Counted` 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;
9. Process `P` reads `v.Left.S_Ptr.Size` value (now it equals to 6) and tries to add key-value pair ⟨ `v.Id, 6` ⟩ to the `Counted` 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 that `insert`.

The `Visited` set and the `Counted` map must be concurrent since multiple processes may modify them concurrently. However, they need to support only single-key operations: `insert` for the `Visited` set and `insert_if_not_exists` for the `Counted` map. Therefore, we can use almost any concurrent key-value data structure, e.g., hash tables [12, 32] or trees [7, 28].

We can learn the `count` result the following way:

1. Read the size of the `Counted` map.
2. Read the size of the `Visited` set.
3. If the latter size is greater than the former size, the answer for at least one node is not known — thus, the answer for the `count` is not known yet.
4. Otherwise, the answer for all the required nodes is known. In that case, we traverse the `Counted` map, summing all the values and obtaining the answer. Note, that it is safe to traverse the `Counted` map now — indeed, the `Counted` map cannot be modified concurrently, since the `count` query is finished.

To allow reading sizes of `Counted` and `Visited` efficiently, the size of a data structure (set or map) should be stored directly in the data structure.

Note, that we should read size of the `Counted` map before reading the size of the `Visited` set. Otherwise, we may think that the `count` query is finished, when in fact it is still being executed. Consider the following scenario (`Visited` and `Counted` are initially empty):

1. Process `R` adds node `v` to the `Visited` set, `Visited = {v.Id}`;
2. Process `P` reads the `Visited` set size and it equals to `1`;
3. Process `R` adds node `u` to the `Visited` set, `Visited = {v.Id, u.Id}`;
4. Process `R` adds key-value pair `⟨ v.id, 5 ⟩` to the `Counted` map, `Counted = {v.Id: 5}`;
5. Process `P` reads the `Counted` map size and it equals to `1`;

Since the `Visited` set size equals to the `Counted` map size, process `P` concludes that the `count` query is finished. However, this conclusion is false, since the result for the node `u` is not known yet.

Conclusions on Chapter 3

In this Chapter, we have explained in details, how the general algorithm, described in Chapter 2, can be applied to a particular data structure: external binary search tree, supporting `insert(k)`, `remove(k)`, `contains(k)` and `count(min, max)` operations. We have provided detailed descriptions of how modifying operations are executed, including an explanation of how we determine, whether a particular key exists in the tree or not. Also we provided an explanation of how the `count` range query is executed: both with `CAS-2` and without it.

CHAPTER 4. DIFFERENT APPLICATIONS OF THE ALGORITHM

4.1. Binary search tree with `collect` range query

Range query `collect(Set, min, max) = {x ∈ Set : min ⩽ x ⩽ max}` returns all keys from the range `[min; max]`. We can implement `collect` range query on an external binary search tree.

`collect(node, min, max)` is executed the following way:

— If `node` is a `KeyNode` we check, whether `min ⩽ node.Key ⩽ max` holds. If so, we add `node.Key` to the result.

— If `node` is an `EmptyNode`, we do nothing.

— If `max < node.Right_Subtree_Min`, then all the required keys are located in the left subtree. In this case, we continue the execution in `node.Left`.

— If `min ⩾ node.Right_Subtree_Min`, then all the required keys are located in the right subtree. In this case, we continue the execution in `node.Right`.

— Otherwise, we continue the execution both in `v.Left` and `v.Right`.

We may calculate the result in a similar way to learning the `count` result (see Section 3.4.3 for detailed explanation). The result yet again consists of two data structures: the `Visited` set (it contains identifiers of all nodes to which queues we inserted the `collect` descriptor) and the `Processed` map.

The `Processed` map contains node identifiers as keys. If node `v` is a `KeyNode` and `min ⩽ v.Key ⩽ max` (i.e., if key `v.Key` should be added to the result) `v.Key` corresponds to `v.Id` in the `Processed` map. Otherwise, `nil` corresponds to `v.Id` in the `Processed` map.

The `Visited` set and the `Processed` map are filled the same way as in Section 3.4.3: 1) we add `v.Id` to the `Visited` set before inserting the `collect` descriptor to `v` queue; 2) we try to add `⟨ v.Id, v.Key ⟩` (or `⟨ v.Id, nil ⟩`) pair to the `Processed` map before removing the `collect` descriptor from `v` queue.

After the `collect` operation has been executed, we obtain the result in the following way:

1. Initialize an empty array `answer`;
2. Iterate through key-value pairs from the `Processed` map. Suppose `⟨ node_id, node_key ⟩` is the current key-value pair;

— If `node_key = nil`, we skip the current pair and proceed to the next key-value pair;
— Otherwise, we add `node_key` to the `answer`;

3. Return `answer` to the caller after the iteration is finished;

4.2. Number of points in a rectangle

Consider a set of points on a plane $\{(x_i, y_i)\}_{i=1}^{n}$. We can formulate a 2D `count` range query the following way: $\texttt{count}(\texttt{Set}, x_{min}, x_{max}, y_{min}, y_{max}) = |\{(x, y) \in \texttt{Set} : x_{min} \leqslant x \leqslant x_{max}, y_{min} \leqslant y \leqslant y_{max}\}|$ — the number of points located in the rectangle bounded by lines $x = x_{min}$, $y = y_{min}$, $x = x_{max}$, $y = y_{max}$ (Fig. 59).

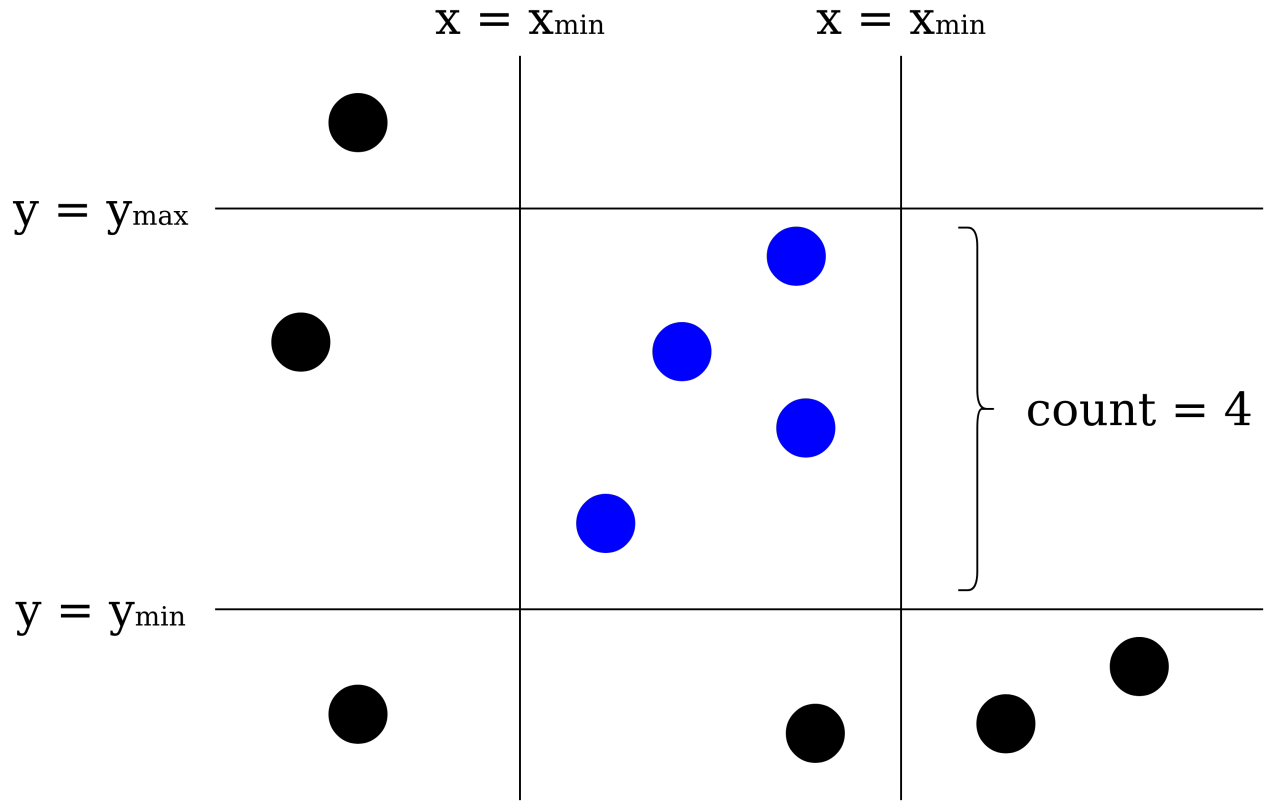


Figure 59 – 2D `count` query

We can employ quad trees [9] to execute 2D `count` queries in an asymptotically optimal way. Our quad tree implementation is an external tree, consisting of three types of nodes:

— `PointNode` is a leaf node that stores a single `(x, y)` point.
— `EmptyNode` is a leaf node that does not store any point.

— `InnerNode` that does not store any points directly: instead, points are stored in the subtree of an internal node, while the internal node itself is used for query routing.

  Consider an internal node `v`, subtree of which stores point set $\mathtt{PS_v} = \{(\mathtt{x_i}, \mathtt{y_i})\}_{i=1}^{n_v}$. Node `v` has two parameters: $\mathtt{x_m}$ and $\mathtt{y_m}$. `v` has four children:

  – Subtree of the first child stores a subset of points $\{(\mathtt{x},\ \mathtt{y}) \in \mathtt{PS_v} : \mathtt{x} < \mathtt{x_m},\ \mathtt{y} < \mathtt{y_m}\}$;
  – Subtree of the second child stores a subset of points $\{(\mathtt{x},\ \mathtt{y}) \in \mathtt{PS_v} : \mathtt{x} < \mathtt{x_m},\ \mathtt{y} \geqslant \mathtt{y_m}\}$;
  – Subtree of the third child stores a subset of points $\{(\mathtt{x},\ \mathtt{y}) \in \mathtt{PS_v} : \mathtt{x} \geqslant \mathtt{x_m},\ \mathtt{y} < \mathtt{y_m}\}$;
  – Subtree of the forth child stores a subset of points $\{(\mathtt{x},\ \mathtt{y}) \in \mathtt{PS_v} : \mathtt{x} \geqslant \mathtt{x_m},\ \mathtt{y} \geqslant \mathtt{y_m}\}$;

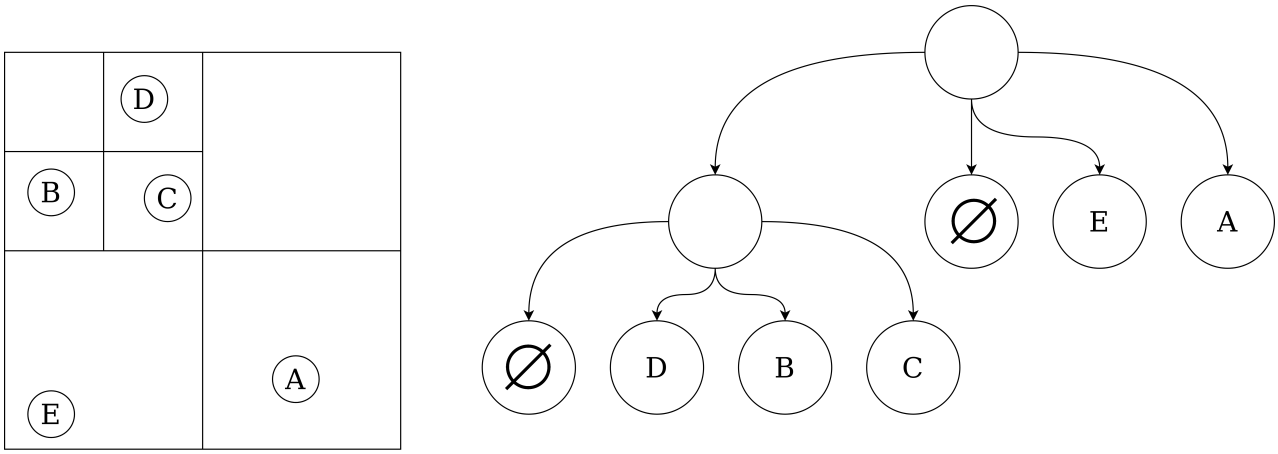
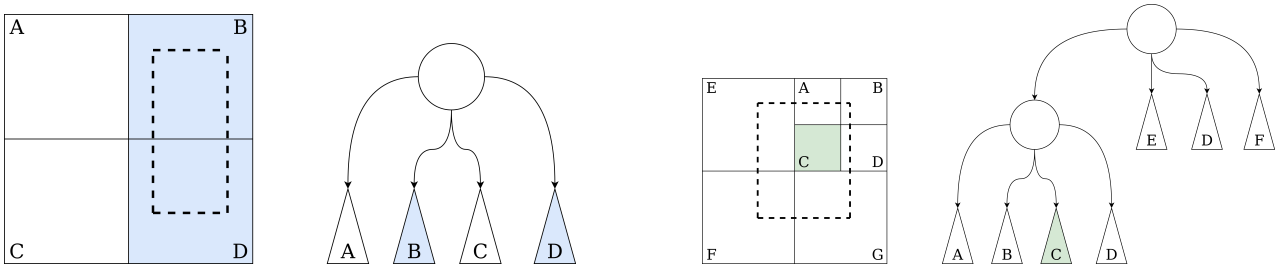


Figure 60 – Example of a quad tree

`count(node,` $\mathtt{x_{min}}, \mathtt{x_{max}}, \mathtt{y_{min}}, \mathtt{y_{max}}$`)` is executed the following way:

— If `node` is a `PointNode`, we check whether $\mathtt{x_{min}} \leqslant \mathtt{node.X} \leqslant \mathtt{x_{max}},\ \mathtt{y_{min}} \leqslant \mathtt{node.Y} \leqslant \mathtt{y_{max}}$ holds. If so, we add `1` to the result, otherwise we add `0` to the result.
— If `node` is an `EmptyNode`, we add `0` to the result.
— Otherwise, `node` is an `InternalNode`. In that case, we execute the query similarly to the algorithm described in Section 3.4. We determine:
  – The set of children, in which the execution should continue (Fig. 61a).
  – The set of children, subtree sizes of which may be added to the result without executing the query in them (Fig. 61b).

and execute the query in an asymptotically optimal way. Detailed explanation of the query execution algorithm goes beyond the scope of this work: it looks very similar to the algorithm, described in Section 3.4, except that more cases should be handled.



(a) The query execution should continue in two children: `B` and `D`

(b) Size of `C` subtree may be added to the result without executing the query in `C`

Figure 61 – Execution of 2D `count` range query in an internal node

Therefore, to execute 2D `count` queries in an asymptotically optimal way, we should augment each inner node with its subtree size. The subtree size is maintained the same way as it was maintained in the binary search tree (see Section 3.2 for details).

4.3. Sorted key-value map with range add and range sum operations

Consider a sorted key-value map, storing a set of key-value entries $\{\langle \texttt{Key}_i, \texttt{Value}_i \rangle\}_{i=1}^{n}$. In addition to ordinary scalar operations (get a value by a key, insert a new entry to the map, modify the value associated with a key, delete an entry by a key, etc.) consider two range queries:

— `range_sum(min, max)` $= \sum_{i=1}^{n} \texttt{Value}_i \cdot \mathbb{I}_{\texttt{min} \leqslant \texttt{Key}_i \leqslant \texttt{max}}$ — calculates the sum of all values whose keys are located in the range `[min; max]`. An $O(N)$ algorithm for executing this operation can be specified the following way (Listing 19):

```
1
2       fun range_sum(Map, min, max):
3           result := 0
4           for ⟨ key, value ⟩ ← Map:
5               if min ⩽ key ⩽ max:
6                   result ← result + value
7           return result
8
```

Listing 19 – An $O(N)$ algorithm for executing `range_sum` operation

— `range_add(min, max, delta)` increments by `delta` all values, whose keys are located in the range `[min; max]`. An $O(N)$ algorithm for executing this operation can be specified the following way (Listing 20):

```
1
2     fun range_add(Map, min, max, delta):
3         for ⟨ key, value ⟩ ← Map:
4             if min ⩽ key ⩽ max:
5                 value ← value + delta
6
```

Listing 20 – An $O(N)$ algorithm for executing `range_add` operation

Both the `range_sum` and the `range_add` operations can be executed in $O(\log N)$ time on an external binary search tree. Let us begin with the `range_sum` operation.

We augment each `InnerNode` with a sum of all values in its subtree. The rules of updating this augmentation are quite simple and remind the rules for updating the subtree size from Section 1.3:

— When inserting new entry $\langle$`k, v`$\rangle$ to the tree, increase by `v` subtree sums of each node on the path from the root to the leaf, storing the new entry.

— When removing an entry $\langle$`k, v`$\rangle$ from the tree, decrease by `v` subtree sums of each node on the path from the root to the leaf, storing that entry.

After that, we can execute `range_sum` query the same way, we executed the `count` query (see Section 3.4 for details), only instead of calculating the sum of subtree sizes we calculate the sum of subtree sums.

Now, we can explain how to execute the `range_add` operation. For that, we augment each `InnerNode` with two more values:

— `Sum_Deltas` — the sum of all the `delta` values, that should be applied to each key-value entry in that node subtree;

— `Size` — the number of key-value pairs, located in the subtree (the rules for updating this augmentation are described in Section 1.3);

We traverse the tree the same way we did it for `count` and `range_sum` queries. Suppose we execute the `range_add(min, max, delta)` operation in a node `v`.

— If `v` is an `EmptyNode`, we do nothing.

— If v is a KeyNode, we check, whether min ⩽ v.Key ⩽ max holds. If so, we increase v.Value by delta (see Section 2.5 for detailed description on how we change node state).
— Otherwise, v is an InnerNode. If we can conclude, that all the keys in v subtree are located in the range [min; max] (see Section 1.3 for details), we increase v.Sum_Deltas by delta (See Section 2.5 for the explanation on how a node state can be changed). Otherwise, we continue the execution in either one or both children.

Sum_Deltas augmentation should be taken into account when executing range_sum queries. If we traverse an inner node v while executing the range_sum query we should take into account that each value in v subtree should be increased by v.Sum_Deltas value. Since v subtree contains v.Size values, the sum of all values in v subtree equals to v.Sum + v.Size · v.Sum_Deltas. If we are executing range_sum(min, max) query in a node v and can conclude, that all keys in v subtree are located in the range [min; max] (see Section 1.3 for details), we increase the result by v.Sum + v.Size · v.Sum_Deltas.

Conclusions on Chapter 4

In this chapter we have shown, that the general algorithm, described in Chapter 2, can be applied to a wide variety of range queries and a broad class of concurrent trees. We have given a number of examples of how our algorithm can be applied to efficiently implement different range queries on different concurrent trees. We have glanced at the collect query on an external binary search tree; the 2D count query on a quad tree; range_sum and range_add queries on an an external binary search tree. We have provided a brief description of how our algorithm can be used in the aforementioned cases, letting us execute such range queries asymptotically optimally.

CHAPTER 5. PRACTICAL RESULTS AND FUTURE WORK

5.1. Linearizability checking in polynomial time

The linearizability verification of a concurrent execution is known to be a very difficult task. It was even shown by Gibbons et al. [13] that the checking, whether a concurrent execution is linearizable or not, is an NP-complete task. However, concurrent trees, implemented with our algorithm, allows for the polynomial-time linearizability check.

Consider $\mathtt{T}$ — an instance of a concurrent tree implemented according to the algorithm from this work. Consider a concurrent execution H in which a set of processes $\{P_i\}_{i=1}^{n}$ execute a set of operations on $\mathtt{T}$, $\{O_j\}_{j=1}^{m}$. For each operation O its result, $R_H(O)$, in the execution H is known. The task is to determine: whether the execution H is linearizable or not. We use the following algorithm:

1. For each operation O_k obtain its timestamp.
2. Sort the operations by timestamps $\{O_{t_j}\}_{j=1}^{m}$, such that $\forall a < b : \mathtt{timestamp}(O_{t_a}) < \mathtt{timestamp}(O_{t_b})$.
3. According to the main invariant (see Section 2.1 for details) the operations should be applied to $\mathtt{T}$ in the order determined by their timestamps — thus, the operations should seem to be applied to $\mathtt{T}$ in the $\{O_{t_j}\}_{j=1}^{m}$ order. Thus, the only possible sequential execution L, equivalent to H (see Section 1.6 for details), must consists of operations in the $\{O_{t_j}\}_{j=1}^{m}$ order.
4. Sequentially apply operations in the timestamps order to $\mathtt{T}$. For each operation O record $R_L(O)$ — the result of O in the sequential execution L.
5. For each operation, compare its result in the concurrent and in the sequential executions. If $\forall j \mid 1 \leqslant j \leqslant m \ : \ R_H(O_{t_j}) = R_L(O_{t_j})$, then the execution H is linearizable. Otherwise — if $\exists j \mid 1 \leqslant j \leqslant m \ : \ R_H(O_{t_j}) \neq R_L(O_{t_j})$ — the execution H is not linearizable.

Therefore, the linearizability checking of the execution of concurrent operations on tree $\mathtt{T}$, implemented according to the algorithm from this work, can be done in the polynomial time.

The algorithm for linearizability checking in polynomial time may be implemented the following way: (Listing 21):

```
1
2 fun lincheck(n_procs, ops_per_process, ops_generator):
3     Tree := new ConcurrentTree()
4
```

```
    /*
    Worker processes will use Operations_Chan to report
    executed operations along with their results
    */
    Operations_Chan := new Channel()

    for i ← 1..n_procs:
        process {
            cur_proc_operations := []
            for j ← 1..ops_per_process:
                /*
                gen_random_operation generates a random
                operation to be executed on the tree instance
                */
                op := ops_generator.gen_random_operation()
                res := Tree.execute(op)
                cur_proc_operations.append(⟨op, res⟩)

            /*
            Report all executed operations along with
            their results
            */
            Operations_Chan.send(cur_proc_operations)
        }.start()

    /*
    Wait for all worker processes to finish their execution
    and collect all executed operations along with their results
    */
    all_operations := []
    for i ← 1..N_PROCS:
        for ⟨op, res⟩ ← Operations_Chan.receive():
            all_operations.append(⟨op, res⟩)

    /* Sort all executed operations by their timestamp */
    all_operations.sort_by { ⟨op, _⟩ → op.Timestamp }

    /*
    Do the linearizability checking: check each operation return the same
    result in both sequential and concurrent executions
    */
    seq_tree := new ConcurrentTree()
    for ⟨op, res⟩ ← all_operations:
        seq_res := seq_tree.execute(op)
        if seq_res ≠ res:
            /* The execution is not linearizable */
            return false
```

```
53      /* The execution is linearizable */
54      return true
```

Listing 21 – The algorithm for linearizability checking of the execution of concurrent operations on tree `T`, implemented according to the algorithm from this work

After we implemented a binary search tree, supporting the `count` query (see Chapter 3 for details), we tested it for linearizability. For that, we ran more than ten millions of concurrent executions, each consisting of a number of operations on an instance of the tree. Each execution was reported to be linearizable.

5.2. Benchmark results

We ran benchmarks of the concurrent tree, supporting `count` range query, to verify, that our solution scales better than the lock-based one and the one, based on the Universal Construction. Benchmark results are presented on Fig. 62. As can be seen from the results, our solution scales better and outperforms both lock-based and Universal Construction-based solutions on `8` threads.

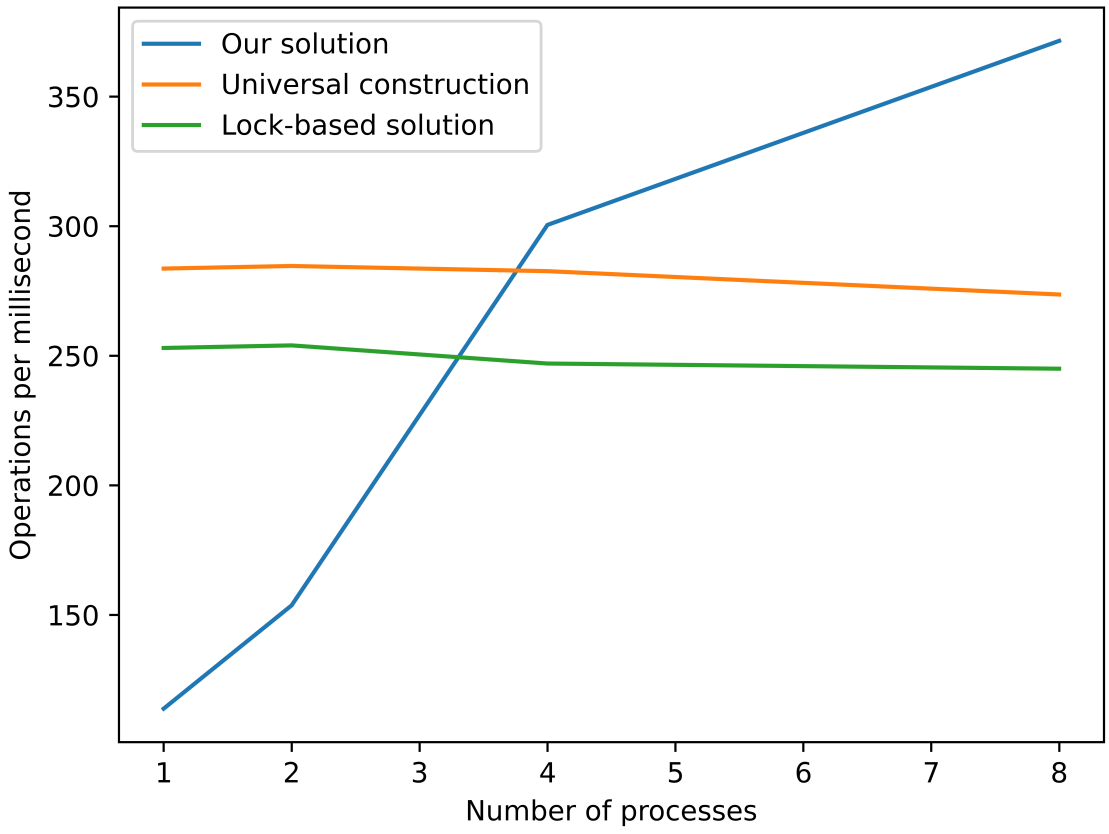


Figure 62 – The results of the benchmark of our solution against the lock-based and Universal Construction-based ones

5.3. Future work

5.3.1. Collaborative rebuilding

Suppose

— Descriptor of a modifying operation `Op` is located at the head of `pv` queue;
— `v` is `pv` child;
— `Op` should continue its execution in `v`;
— `v` subtree should be rebuilt by `Op`, since the number of modification, applied to `v` subtree, will exceed the threshold after applying `Op`.

Suppose processes `P` and `R` at the same time:

1. Read `Op` descriptor from the head of `pv` queue;
2. Determine, that `v` subtree should be rebuilt after applying `Op`;
3. Traverse `v` subtree collecting `IS` — data items from it;
4. Build `v'` — an ideal subtree, consisting of data items `IS`;
5. Try to replace `v` with `v'` using `CAS(&child_ptr, v, v');`

Only a single process (either `P` or `R`, say `P` for convenience) performs a successful modification at step (5) — another process (`R`) faces unsuccessful `CAS`, therefore the CPU time spent by process `R` is wasted, since only the process `P` performed the rebuilding.

To get rid of such CPU wasting, we can employ *collaborative rebuilding*. If multiple processes see that `v` subtree should be rebuilt, these processes may rebuild `v` subtree in parallel, so that each participating process does a fraction of the overall work. We may adapt parallel tree traversal algorithm and parallel tree building algorithm, described in [20], for this purpose. We may use one of the multitude existing *fork-join* frameworks, such as Cilk [8] to implement these parallel algorithms in the *fork-join* paradigm.

5.3.2. $O(\log N)$ tree balancing strategies

If the number of modifications, applied to the whole tree, exceeds a threshold, we rebuilt the whole tree in $O(N)$ time. Thus, some operations take an abnormally long time to execute, since they should spend $O(N)$ time to rebuild the whole tree (in contrast to $O(\log N)$ time required to execute the operation without doing the rebuild).

We should study other tree balancing strategies in an effort to find a strategy that guarantees, that each operation is executed in $O(\log N)$ time. We may study concurrent balancing strategies starting with *chromatic balancing*, described in [29].

5.3.3. Executing `insert(k)` and `remove(k)` operation without checking, whether key `k` exists in the tree

Suppose we execute `insert(k)` (or `remove(k)`) operation without knowing, whether key `k` exists in the tree. Thus, we cannot increase (or decrease) subtree size of each node on the execution path — indeed, we do not know, whether the operation actually inserts new key to the tree (or removes an existing one from the tree).

We can split `v` subtree size on two components: the integer number `Base_Size` and a set of result pointers (see Section 2.2 for details on result pointers) of `insert` and `remove` operation, that are being executing in `v` subtree (Fig. 63).

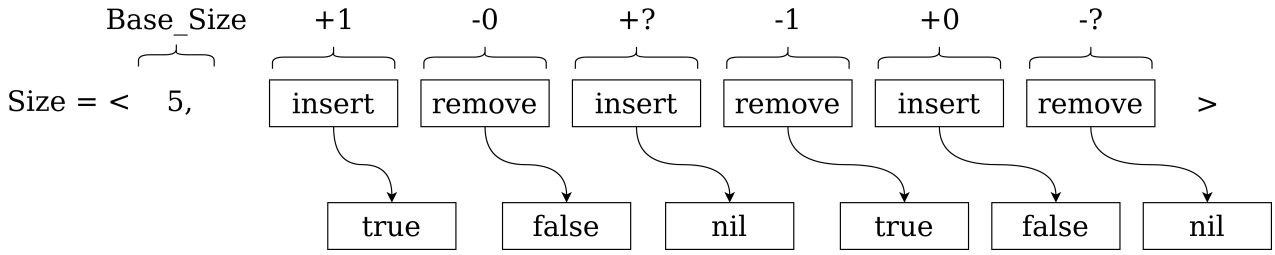


Figure 63 – Splitting node subtree size on `Base_Size` and a set of result pointers

Each `insert` operation, being executed in `v` subtree, may be completed either

— Successfully (when a new key is inserted to the tree). In this case, the result pointer for this `insert` operation points to `true`. In that case, `v` subtree size should be increased by one.

— Unsuccessfully (when the key already exists in the tree, and, thus, no new key is inserted to the tree). In this case, the result pointer for this `insert` operation points to `false`. In that case, `v` subtree size should not be modified.

Similarly, each `remove` operation, being executed in `v` subtree, may be completed either

— Successfully (when an existing key is removed from the tree). In this case, the result pointer for this `remove` operation points to `true`. In that case, `v` subtree size should be decreased by one.

— Unsuccessfully (when the key does not exist in the tree, and thus no key is removed from the tree). In this case, the result pointer for this `remove` operation will point to `false`. In that case, `v` subtree size should not be modified.

To get the current subtree size of a node `v` we perform the following:

1. Read `v` state, obtaining `Base_Size` and `DS` — the set of descriptors of `insert` and `remove` operation, being executing in `v` subtree.
2. If there is at least one uncompleted operation in `DS` — help complete it.
3. After there are no more uncompleted operations in `DS`, calculate `inc_val` — the number of successful `insert` operations from `DS`.
4. And `dec_val` — the number of successful `remove` operations from `DS`.
5. The resulting size of `v` subtree equals to `Base_Size` + `inc_val` − `dec_val`.

We may reduce the size of the set `DS` by

— Removing result pointers of unsuccessful `insert` and `remove` operations from `DS` (indeed, unsuccessful `insert` and `remove` operations do not change `v` subtree size);
— Removing result pointers of successful `insert` operations from `DS`, meanwhile increasing `Base_Size` by one for each such removed result pointer;
— Removing result pointers of successful `remove` operations from `DS`, meanwhile decreasing `Base_Size` by one for each such removed result pointer.

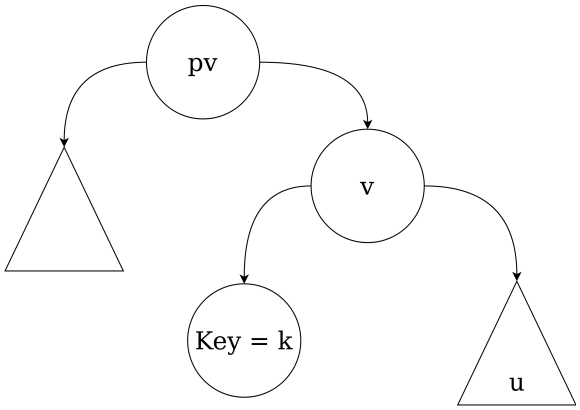
This reduction procedure reminds the *garbage collection* [21] and can be done either by helper processes, executing operations on a tree, or by dedicated background processes (garbage collectors).

In our current implementations `insert(k)` and `remove(K)` operations require two tree traversals: the first one to former whether key `k` exists in the tree and the latter one to execute the operation, if it is to be be executed successfully (see Section 3.2 for details). After implementing this optimization, `insert` and `remove` operations will require only one tree traversal to be executed, thus reducing the operation execution time.

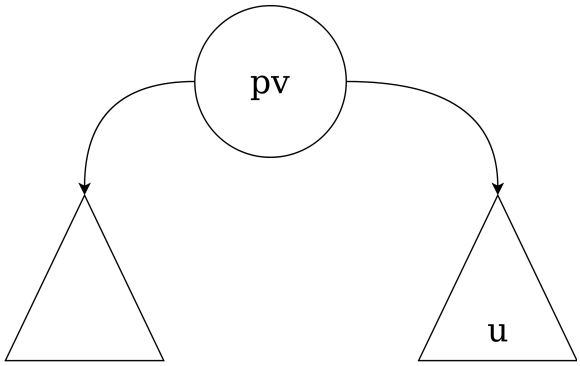
5.3.4. Getting rid of `EmptyNode` structure

Empty node do not store keys, wasting memory for nothing. We may try to modify the `remove` operation so that it does not replace key nodes with empty nodes.

One possible `remove(k)` execution strategy is the following: suppose node `v` is a parent of `KeyNode { Key = k }`, `pv` is a parent of `v` and `u` is a sibling of `KeyNode { Key = k }` (Fig. 64a). Instead of replacing `KeyNode { Key = k }` with and `EmptyNode`, we may replace `v` with `u`: after that, `u` will become a child of `pv` (Fig. 64b). Key `k` will thus be removed from the tree, and no `EmptyNode` will be created.



(a) Initial tree structure



(b) Tree structure after the key is removed from the tree

Figure 64 – Execution of `remove(k)` operation without creating an `EmptyNode`

CONCLUSION

We have managed to design an algorithm for executing range queries on concurrent trees. Our solution satisfies all the desired properties:

— Lock-free progress guarantees;
— Asymptotic optimality of range queries;
— The ability to execute multiple concurrent modifying operation successfully;

while neither of the existing solutions satisfy all these properties.

Our solution is generic and can be applied to a wide variety of range queries and to a broad class of trees. We shown the validity of the developed algorithm by designing and implementing a concurrent external binary search tree supporting the `count` range query. Also, we provided a brief glance on how our solution can be used to implement other range queries on other types of trees. On top of that, we described the algorithm to check the linearizability of the developed concurrent data structures in polynomial time.

REFERENCES

1 *Aksenov V.*, *Kuznetsov P.*, *Shalyto A.* Parallel combining: Benefits of explicit synchronization // arXiv preprint arXiv:1710.07588. — 2017.

2 *Arbel-Raviv M.*, *Brown T.* Harnessing epoch-based reclamation for efficient range queries // ACM SIGPLAN Notices. — 2018. — Vol. 53, no. 1. — P. 14–27.

3 *Bernstein P. A.*, *Hadzilacos V.*, *Goodman N.* Concurrency control and recovery in database systems. Vol. 370. — Addison-wesley Reading, 1987.

4 *Bodon F.*, *Rónyai L.* Trie: an alternative data structure for data mining algorithms // Mathematical and Computer Modelling. — 2003. — Vol. 38, no. 7–9. — P. 739–751.

5 *Brown T.*, *Avni H.* Range queries in non-blocking k-ary search trees // International Conference On Principles Of Distributed Systems. — Springer. 2012. — P. 31–45.

6 *Brown T.*, *Prokopec A.*, *Alistarh D.* Non-blocking interpolation search trees with doubly-logarithmic running time // Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2020. — P. 276–291.

7 *Chatterjee B.*, *Nguyen N.*, *Tsigas P.* Efficient lock-free binary search trees // Proceedings of the 2014 ACM symposium on Principles of distributed computing. — 2014. — P. 322–331.

8 Cilk: An efficient multithreaded runtime system / R. D. Blumofe [et al.] // Journal of parallel and distributed computing. — 1996. — Vol. 37, no. 1. — P. 55–69.

9 Computational geometry: algorithms and applications / M. T. De Berg [et al.]. — Springer Science & Business Media, 2000.

10 *Feldman S.*, *LaBorde P.*, *Dechev D.* A wait-free multi-word compare-and-swap operation // International Journal of Parallel Programming. — 2015. — Vol. 43, no. 4. — P. 572–596.

11 Flat combining and the synchronization-parallelism tradeoff / D. Hendler [et al.] // Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures. — 2010. — P. 355–364.

12 *Gao H.*, *Groote J. F.*, *Hesselink W. H.* Lock-free dynamic hash tables with open addressing // Distributed Computing. — 2005. — Vol. 18, no. 1. — P. 21–42.

13 *Gibbons P. B.*, *Korach E.* Testing shared memories // SIAM Journal on Computing. — 1997. — Vol. 26, no. 4. — P. 1208–1244.

14 *Guttman A.* R-trees: A dynamic index structure for spatial searching // Proceedings of the 1984 ACM SIGMOD international conference on Management of data. — 1984. — P. 47–57.

15 *Harris T. L.*, *Fraser K.*, *Pratt I. A.* A practical multi-word compare-and-swap operation // International Symposium on Distributed Computing. — Springer. 2002. — P. 265–279.

16 *Herlihy M.* Wait-free synchronization // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1991. — Vol. 13, no. 1. — P. 124–149.

17 *Herlihy M.*, *Luchangco V.*, *Moir M.* Obstruction-free synchronization: Double-ended queues as an example // 23rd International Conference on Distributed Computing Systems, 2003. Proceedings. — IEEE. 2003. — P. 522–529.

18 *Herlihy M. P.*, *Wing J. M.* Linearizability: A correctness condition for concurrent objects // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1990. — Vol. 12, no. 3. — P. 463–492.

19 Introduction to algorithms / T. H. Cormen [et al.]. — MIT press, 2022.

20 *JéJé J.* An introduction to parallel algorithms // Reading, MA: Addison-Wesley. — 1992. — Vol. 10. — P. 133889.

21 *Jones R.*, *Hosking A.*, *Moss E.* The garbage collection handbook: the art of automatic memory management. — CRC Press, 2016.

22 *Kogan A.*, *Petrank E.* Wait-free queues with multiple enqueuers and dequeuers // ACM SIGPLAN Notices. — 2011. — Vol. 46, no. 8. — P. 223–234.

23 *Lamport L.* How to make a multiprocessor computer that correctly executes multiprocess programs // IEEE Transactions on Computers c-28. — 1979. — Vol. 9. — P. 690–691.

24 *Lamport L.* A new solution of Dijkstra's concurrent programming problem // Concurrency: the Works of Leslie Lamport. — 2019. — P. 171–178.

25 *Lamport L.* Time, clocks, and the ordering of events in a distributed system // Concurrency: the Works of Leslie Lamport. — 2019. — P. 179–196.

26 Making data structures persistent / J. R. Driscoll [et al.] // Journal of computer and system sciences. — 1989. — Vol. 38, no. 1. — P. 86–124.

27 *Michael M. M.*, *Scott M. L.* Simple, fast, and practical non-blocking and blocking concurrent queue algorithms // Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. — 1996. — P. 267–275.

28 *Natarajan A.*, *Mittal N.* Fast concurrent lock-free binary search trees // Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming. — 2014. — P. 317–328.

29 *Nurmi O.*, *Soisalon-Soininen E.* Chromatic binary search trees // Acta informatica. — 1996. — Vol. 33, no. 6. — P. 547–557.

30 *Okasaki C.* Purely functional data structures. — Cambridge University Press, 1999.

31 *Papadimitriou C. H.* The serializability of concurrent database updates // Journal of the ACM (JACM). — 1979. — Vol. 26, no. 4. — P. 631–653.

32 *Purcell C.*, *Harris T.* Non-blocking hashtables with open addressing // International Symposium on Distributed Computing. — Springer. 2005. — P. 108–121.

33 *Sun Y.*, *Ferizovic D.*, *Belloch G. E.* PAM: parallel augmented maps // Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2018. — P. 290–304.

34 The art of multiprocessor programming / M. Herlihy [et al.]. — Newnes, 2020.

35 *Treiber R. K.* Systems programming: Coping with parallelism. — International Business Machines Incorporated, Thomas J. Watson Research …, 1986.

36 *Yang C.*, *Mellor-Crummey J.* A wait-free queue as fast as fetch-and-add // Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2016. — P. 1–13.

37 Cloud Spanner: TrueTime and external consistency [Электронный ресурс]. — 2022. — URL: `https://cloud.google.com/spanner/docs/true-time-external-consistency`.

38 compare-and-swap [Электронный ресурс]. — 2022. — URL: https://en.wikipedia.org/wiki/Compare-and-swap.

39 Double compare-and-swap [Электронный ресурс]. — 2022. — URL: https://en.wikipedia.org/wiki/Double_compare-and-swap.

40 fetch-and-add [Электронный ресурс]. — 2022. — URL: https://en.wikipedia.org/wiki/Fetch-and-add.

41 Persistent data structures [Электронный ресурс]. — 2022. — URL: https://en.wikipedia.org/wiki/Persistent_data_structure.

42 test-and-set [Электронный ресурс]. — 2022. — URL: https://en.wikipedia.org/wiki/Test-and-set.

43 Tree (data structure) [Электронный ресурс]. — 2022. — URL: https://en.wikipedia.org/wiki/Tree_(data_structure).

44 Universally unique identifier [Электронный ресурс]. — 2022. — URL: https://en.wikipedia.org/wiki/Universally_unique_identifier.