A Lock-Free B$^+$tree *

Anastasia Braginsky† Erez Petrank‡

June 19, 2012

Abstract

Lock-free data structures provide a progress guarantee and are known for facilitating scalability, avoiding deadlocks and livelocks, and providing guaranteed system responsiveness. In this paper we present a design for a lock-free balanced tree, specifically, a B$^+$tree. The B$^+$tree data structure has an important practical applications, and is used in various storage-system products. As far as we know this is the first design of a lock-free, dynamic, and balanced tree, that employs standard compare-and-swap operations.

*Supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 283/10).

†Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel anastas@cs.technion.ac.il

‡Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel erez@cs.technion.ac.il

1 Introduction

The growing popularity of parallel computing is accompanied by an acute need for data structures that execute efficiently and provide guaranteed progress on parallel platforms. Lock-free data structures provide a progress guarantee: if the program threads are run sufficiently long, then at least one of them must make progress. This ensures that the program as a whole progresses and is never blocked. Although lock-free algorithms exist for various data structures, lock-free balanced trees have been considered difficult to construct and as far as we know a construction for a lock-free balanced tree is not known.

In recent decades, the B-tree has been the data structure of choice for maintaining searchable, ordered data on disk. Traditional B-trees are effective in large part because they minimize the number of disk blocks accessed during a search. When using a B-tree on the computer memory, a reasonable choice is to keep a node on a single cache line. However, some studies show that a block size that is a (small) factor of the processor's cache line can deliver better performance if cache pre-fetching is employed by the hardware [15, 4]. Further details about the B-Tree structure and the B^+tree variant appear in Subsection 2.1.

This paper presents the first lock-free, linearizable, dynamic B^+tree implementation supporting searches, insertions, and deletions. It is dynamic in the sense that there is no (static) limit to the number of nodes that can be allocated and put in the tree. The construction employs only reads, writes, and single-word CAS instructions. Searches are not delayed by rebalancing operations. The construction employs the lock-free chunk mechanism proposed in [3]. The chunk mechanism provides a lock-free linked list that resides on a consecutive chunk of memory and maintains a lower- and upper-bound on the number of elements. The chunks are split or joined with other chunks to maintain the bounds in the presence of insertions and deletions. This lock-free chunk mechanism fits naturally with a node of the B^+tree that is split and joined, keeping the number of elements within given bounds, and thus maintaining the balance of the tree.

Our construction follows some basic design decisions that reduce the complexity of the algorithm. First, a node marked by the need to join or split is frozen, and no more operations are allowed on it. It is never resurrected, and one or two nodes are allocated to replace it. This eliminates much of the difficulty with threads waking up after a long idle period and encountering an old node that has been split or joined. In general, a node begins its lifespan as an infant, proceeds to become a normal node, and remains so until frozen for a split or a join, after which it is eventually reclaimed. This monotonic progress, reflected in the node's state, simplifies the design. The replacement of old nodes with new ones is challenging as data may be held in both the old and the new nodes simultaneously. To allow lock-freedom, we let the search operation dive into old nodes as well as new ones. But to ensure linearizability, we only allow new nodes to be modified after the replacement procedure is completed. Additionally, we take special care in the selection of a neighboring node to join with, to ensure that it cooperates correctly. Finally, we enforce the invariant that two join nodes always have the same parent. Our construction follows important lock-free techniques that have been previously used. In particular, we mark pointers to signify deletion following Harris [8], we assign nodes with states similarly to Ellen et al. [6]. We also extend these techniques in ways that we hope will be useful for future work, e.g., we gradually move a node to the *frozen* state, by marking its fields one by one as frozen.

This design of the lock-free B^+tree is meant to show the feasibility of a lock-free balanced tree. It is quite complex and we have not added (even straightforward) optimizations. We implemented this design (as is) in C and ran it against an implementation of a standard lock-based B^+tree [16]. The results show that the lock-based version wins when no contention exists or the contention is very low. However, as contention kicks in, the lock-free B^+tree behaves much better than the lock-based version. The lock-free tree is highly scalable and allows good progress even when many threads are executing concurrently. Similarly to the lock-free algorithm of the linked-list, a wait-free variant of the search method (denoted *contains*) can be defined here as well and

in the same manner. Again, to keep things simple, we do not spell it out.

In addition to implementing and measuring the algorithm, we also present the full proof for the correctness of this design with respect to linearizability [10] and (bounded) lock-freedom [11, 14]. Note that a balanced tree has a better worst-case behavior compared to regular trees. Ignoring concurrency, each operation has a worst-case complexity of $O(\log n)$ in contrast to a worst-case complexity of $O(n)$ for an imbalanced tree. Furthermore, in the presence of concurrent threads, we prove that progress must be made at worst-case within $O(T \log n + T^2)$ computational steps, where T is number of the concurrent running threads and n is number of keys in the B$^+$tree. (This means bounded lock-freedom with bound $O(T \log n + T^2)$.) Such guarantee can only be achieved with balanced trees, as computing a similar bound on the worst-case time to make progress in a non-balanced tree would yield $O(Tn)$[1].

Previous work on lock-free trees include Fraser's construction [7] of a lock-free balanced tree that builds on a transactional memory system. Our work does not require any special underlying system support. Fraser also presents a construction of a lock-free tree that uses multiple-word CAS [7], but this construction offers no balancing and at worst may require a linear complexity for the tree operations. Recently, Ellen *et al.* [6] presented a lock-free tree using a single-word CAS, but their tree offers no balancing. Bender *et al.* [2] described a lock-free implementation of a cache-oblivious B-tree from LL/SC operations. Our construction uses single-word CAS operations. Moreover, a packed-memory cache-oblivious B-tree is not equivalent to the traditional B$^+$tree data structure. First, it only guarantees amortized time complexity (even with no contention), as the data is kept in an array that needs to be extended occasionally by copying the entire data structure. Second, it does not keep the shallow structure and is thus not suitable for use with file systems. Finally, a full version of [2] paper has not yet appeared and some details of lock-free implementation are not specified.

In Section 2 we set up some preliminaries and present the B$^+$tree representation in the memory together with the basic B$^+$tree algorithms. In Section 3 we describe the B$^+$tree node's states and recall the lock-free chunk functionality from [3]. Balancing functions are presented in brief in Section 4, and the implementation and results are described in Section 5. We conclude in Section 6. In Appendix A we describe the linearization points. Supporting B$^+$tree methods are presented in Appendix B. Balancing code and all relevant details are presented in Appendix C. In Appendix D the redirection and help methods are presented. Boundary conditions by which the root needs to be exchange are presented in Appendix E, and minor modifications required of the original chunk mechanism are presented in Appendix F. Finally, the full proof can be found in Appendix G.

2 Preliminaries and Data Structure

This section presents the data structures used to implement the lock-free B$^+$tree, starting with a review of the lock-free chunk mechanism presented in [3]. A *chunk* is a (consecutive) block of memory that contains *entries*. Each entry contains a key and a data field, and the entries are stored in the chunk as a key-ordered linked list. A chunk consumes a fixed amount of space and has two parameters, determining the minimum and maximum entries that may reside in it. The chunk supports set operations such as *search*, *insert* and *delete*. When an insert of a new entry increases the number of entries above the maximum, a *split* is executed and two chunks are created from the original chunk. Similarly, when a delete violates the minimum number of entries, the chunk mechanism *joins* this chunk and another chunk, obtained from the data structure using the chunks (in particular the B$^+$tree). Therefore, the B$^+$tree implements a method that the chunk can call to obtain a partner to join with. A different B$^+$tree method is called by the chunk mechanism when the split or join are completed to ask that the tree replaces the frozen nodes with new ones. The chunk also supports an additional *replace* operation that allows replacing the data of an entry with a new value atomically without modifying the entry's

[1] Actually, we do not know how to show a lock-free bound which is lower than $O(T^2 n)$ for non-balanced concurrent trees.

location in the list. This operation is useful for switching a descendant without modifying the key associated with it[2]. All operations are lock-free.

2.1 The B$^+$tree

A B$^+$tree [5] is a balanced tree used to maintain a set of *keys*, and a mapping from each key to its associated *data*. Each node of the tree holds entries, each entry has a key and an auxiliary data. In contrast to a B-tree, only the leaves in a B$^+$tree hold the keys and their associated data. The data of the keys in the internal nodes is used to allow navigating through the tree. Thus, data in an internal node of the tree contains pointers to descendants of the internal node. The B$^+$tree structure simplifies the tree insertions and deletions and is commonly used for concurrent access. In our variant of a B$^+$tree, key repetition is not allowed.

Each internal node consists of an ordered list of entries containing keys and their associated pointers. A tree search starts at the root and chooses a descendant according to the values of the keys, the convention being that the entry's key provides the upper bound on the set of keys in its subtree. Each node has a minimum and maximum number of possible entries in it. In our B$^+$tree the maximum is assumed to be even and is denoted d. The minimum is set to $d/2 - 3$. For $d \geq 10$ this ensures the balance of the tree, and specifically that the number of nodes to be read before reaching a leaf is bounded by a logarithm of the tree size. All insertions and deletions happen at leaves. When an insert violates the maximum allowed number of entries in the node, a split is performed on that node. When a delete violates the minimum allowed number of entries, the algorithm attempts to join two nodes, resulting in borrowing entries from a neighboring node or merging the two nodes, if moving entries is not possible.

Splitting and joining leaves may, in turn, imply an insert or a delete to the parent, and such an update may roll up until the root. We ignore the minimum number of entries on the root, in order not to enforce a minimal number of entries in the tree. Note that splits and joins always create nodes with a legitimate number of entries. In practice, the minimum value is sometimes set to be smaller than $d/2 - 3$ to avoid frequent splits and joins.

2.2 The structure of the proposed B$^+$tree

For simplicity, our construction assumes the key and the data fit into a single word. This is the assumption of the chunk mechanism and it makes the allocation of a new entry easier. In practice, this means a word of 64 bits, with a key of 32 bits and data of 32 bits.[3] An architecture that provides a double-word compare-and-swap would allow using a full word for each of the fields, removing the restrictions, and simplifying the construction. The key values are taken from a finite set, bounded from above by a value that we denote ∞. The tree is represented by a pointer to the root node, initially set to an empty root-leaf node.

Our B$^+$tree node is built using the chunk structure of [3]. The chunk's maximum and minimum number of entries are set to d and $d/2 - 3$ to satisfy the B$^+$tree node requirement (except for the zero minimum bound on the root). In addition to a chunk, the tree node contains two additional fields to support its management: a *height* field indicating the distance from the leaves and a *root* flag indicating whether the node is a root.

We briefly review the fields of a chunk (see Figure 1). A detailed discussion appears in [3]. The main part of the chunk is an array that contains all the entries. The *counter* field counts the number of entries in a chunk. It is accurate during sequential execution and is always guaranteed to hold a lower bound on the real count, even in the presence of concurrent executions. The pointers *new*, *joinBuddy*, *nextNew* and *creator* point to nodes

[2]The replace operation did not appear in the short conference version of [3] and is described in Section F.1.

[3]Since a data field cannot hold a full pointer, we assume a translation table, or some base pointer to which the 32-bit address is added to create the real memory address. In the first case, this limits the number of nodes to 2^{32} nodes, and in the second case, it limits the entire tree space to 4GB, which is not a harsh constraint.

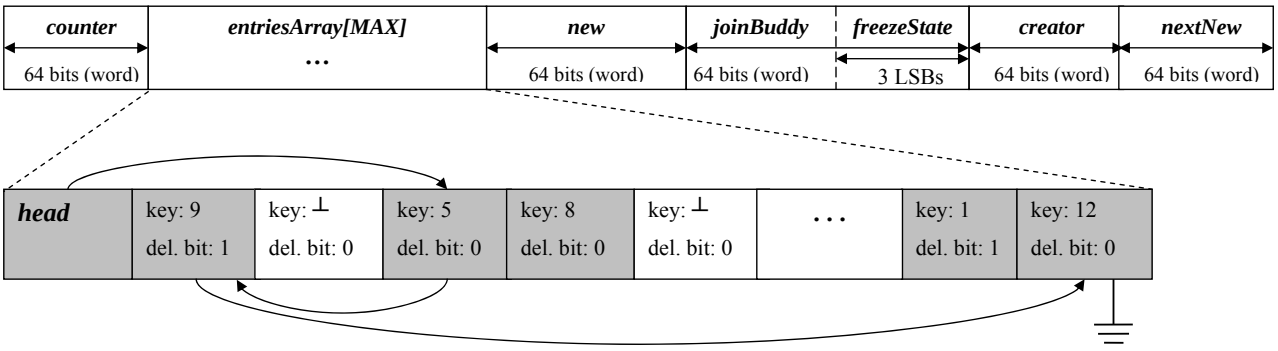

Figure 1: The structure of a chunk. The allocated grey entries present the ordered linked list.

involved in the rebalancing, to be described below in Section 4. The split and join of a chunk requires a *freeze* of all operations on it, which imposes the *freeze state* of a chunk to be declared using *freezeState* field. The freezing mechanism will be explained later, in Sections 3 and 4.

2.3 Memory Management

To avoid some of the ABA problems, lock-free algorithms typically rely on garbage collection or use the hazard pointer mechanism of Michael [13]. To simplify the current presentation, we assume the existence of garbage collection for the nodes. This means that nodes are never reused unless they become unreachable from all threads. An extension of the same scheme to a use of hazard pointers is possible.[4]

2.4 The Basic B$^+$tree Operations

The B$^+$tree interface methods: *SearchInBtree()*, *InsertToBtree()*, and *DeleteFromBtree()* are quite simple. The code of the basic B$^+$tree operations is presented in Algorithm 3 (relegated to Appendix C). An insert, delete, or search operation first finds the leaf with the relevant key range, after which the appropriate chunk operation is run on the leaf's chunk. It either simply succeeds or a more complicated action of a split or a join begins. Some care is needed when the suitable leaf is a new one (an infant), whose insertion into the B$^+$tree is not yet complete. In that case, we must help finish the insertion of the new node before continuing to perform the operation on it. Further explanations on the freezing of a node, on the infant state, etc. appear in Section 3.

3 Splits and Joins with Freezing

Before it is split or joined, a node's chunk must be frozen. The complete details appear in [3]. The freezing is executed by the chunk mechanism when its size limits are violated. This happens obliviously to the containing data structure, in this case, the B$^+$tree. Here we provide an overview on the chunk's freeze required to understand the B$^+$tree algorithm. To freeze a node, i.e., to freeze the chunk in it, all the chunk's entries are marked *frozen* (one by one) by setting a designated bit in each entry. After all the entries are marked frozen, no changes can occur on this node. A thread that discovers that a node needs to be frozen, or that a freeze has already begun, helps finish freezing the node. However, search operations do not need to help in freeze and can progress on the frozen nodes. Since changes may occur before all entries are marked frozen, the final state of the frozen node may not require a split or a join at the end of the freeze. Still a frozen node is never

[4] In the implementation we measured, we implemented hazard pointers inside the chunk and did not reclaim full nodes at all during the execution.

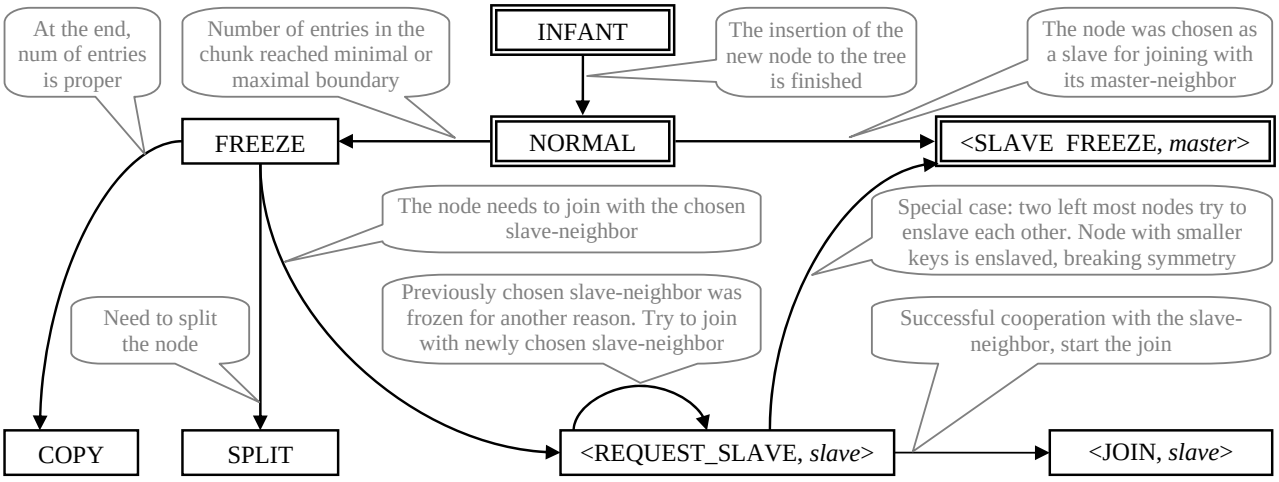

Figure 2: The state transitions of the freeze state of a node. The initial states are presented in the boxes with the double border.

resurrected. After the freeze has been marked and the node can no longer be modified, a decision is made on whether it should be split, or joined with a neighboring node, or just copied into a single new node. If a join is required, then a neighboring node is found by the B^+tree. This communication between the chunk and the B^+tree is implemented using a predetermined method *FindJoinSlave()* that the tree supplies and the chunk mechanism uses. Then the neighboring chunk is frozen too. To recover from the node freeze, one or two nodes are allocated, and the live entries in the frozen node (or nodes) are copied into the new node (or nodes). Thereafter, a B^+tree method *CallForUpdate()* is called to let the tree replace the frozen nodes with the new ones. We focus in what follows on issues specific to the B^+tree, i.e., finding a neighbor, replacing the frozen nodes with the new ones in the B^+tree, and maybe rolling up more splits or joins.

Each tree node has a *freezeState* field, holding one of eight possible freeze states. Three bits are used to store the state. The freeze state is also a communication link between the B^+tree and the chunk mechanism, and so it can be read and updated both by the B^+tree and by the chunk. When a new node is created to replace a frozen node, and until it is properly inserted into the B^+tree, its freeze state is marked as INFANT. No insertions or deletions are allowed on an infant node until the node's freeze state becomes NORMAL. Any thread that attempts an operation on such a node must first help move this node from the INFANT to the NORMAL state. A node that is properly inserted into the B^+tree and can be used with no restrictions has a NORMAL freeze state. When an insert or a delete operation violates the maximum or minimum number of entries, a freeze of that node is initiated and its freeze state becomes FREEZE. After the freezing process stabilizes and the node can no longer be modified, a decision is reached about which action should be taken with this node. This decision is then marked in its freeze state as explained below.

When neither split nor join is required (because concurrent modifications have resulted in a legitimate number of entries), the freeze state of the node becomes COPY, and the node is simply copied into a newly allocated node. By the end of the copy, the parent's pointer into the old node is replaced (using the chunk's replace operation) with the pointer to the new node, and the new node becomes NORMAL. When a split is required, the node's frozen state changes to SPLIT and all its live entries are copied into two new INFANT nodes. These nodes are then inserted into the tree in place of the frozen node, after which they can become NORMAL. A join is more complicated since a neighbor must be found and *enslaved* for the purpose of the join. Since only three

bits are required to store the freeze state, we can use the freeze state to also store a pointer to a join buddy and modify the state and the pointer together atomically.[5] The join process starts by looking for a neighbor that can be enslaved for the join and then the freeze state of the join initiator is changed into REQUEST_SLAVE together with a pointer to a potential join buddy in the *joinBuddy* word. Thus, the freeze state is actually modified into a pair <REQUEST_SLAVE, *slave*>. At the enslaved node, its state is then modified from NORMAL into the pair <SLAVE_FREEZE, *master*>, where *master* is a pointer to the node that initiated the join. (Upon failure, we try to resolve the contention and try again.) When the connection between the join initiator (the master) and the join slave is finalized, the freeze state of the master is modified into <JOIN, *slave*>, where *slave* points to the determined join buddy node. The node that is typically chosen for a join is the immediate left sibling of the current node, except for the leftmost node, which chooses its immediate right sibling for the join. A special boundary condition appears when the two leftmost children of a node try to enslave each other. In order to break the symmetry in this case, we take special care to identify this situation and then choose the leftmost sibling among the two to be the slave. Figure 2 presents the state transition diagram for the *freezeState* field.

4 Balancing the B$^+$tree

The basic methods for the tree operations have been discussed in Section 2.4. We now give a high-level description of how to balance the tree following a split or a join of a node, and discuss the interface between the chunk mechanism and the tree operations. As we said, upon a violation of the node size constraints, the chunk mechanism first freezes the node to prevent it from being modified and decides on the rebalancing action (split, join, or copy). In case of a join, the chunk mechanism invokes the B$^+$tree method *FindJoinSlave()*, which finds such a join buddy. Later, the chunk mechanism creates new node(s) and copies the relevant information into them. When this part is completed, the chunk calls B$^+$tree method *CallForUpdate()*. This method lets the B$^+$tree algorithm replace the frozen node (or nodes) with the newly created node (or nodes) in the tree. The *CallForUpdate()* method actually redirects the calls according to whether a split, a copy, or a join occurred. Let us examine each of these cases. The pseudo-code for *CallForUpdate()* is presented in Appendix D.

Before diving into the details, note that in general, upon creation of a node due to a split, a join, or a copy, the new node's freeze state is initiated to INFANT, its root flag is initiated to FALSE, its height value is copied from the original node's height value, and its counter is initiated to the exact number of entries copied into it. Also, the *creator* pointer of a new node is initiated to point to the old node, initiated the split, join, or copy operation.

4.1 Node Split

After the chunk mechanism executes a split, the original node N is frozen, and N's *new* field points to the new node N_1 holding the lower half of the keys from the old node N. The field N_1.*nextNew* points to the second new node N_2 holding the higher half of the keys from the old node N. The two new nodes' freeze states are initiated to INFANT so that no updates can occur on these nodes until they are inserted into the tree. Given that the chunk split already completed, the *CallForUpdate()* method invokes the *InsertSplitNodes()* method, algorithm of which we describe below. The code with all the details is provided in Appendix C (Algorithm 4).

Replacing split node N starts by searching for its parent P in the tree. If the parent cannot be found, then the input node is no longer in the tree. This happens if the new node N_1 was properly inserted by some other thread, and the node N was disconnected in the process. In this case, the splitting process continues with inserting N_2. Otherwise, and having found the parent, we modify it to point to the new node N_1. This is done by inserting a new entry to P. The new entry contains the maximal key from N_1 as key and the pointer to

[5] An 8-alignment of a node can be assumed in modern systems and the three redundant least-significant bits can hold the freeze state

N_1 as data. If the insert fails, it means that someone else has inserted this entry to the chunk and it is fine to continue. Therefore, we do not check if the insert succeeded. Note the possibility that the parent's chunk insert will create a split in the parent, which will recursively cause a split and roll it up the tree.

After the first new node is in place, we replace the pointer in the parent node, that points to the frozen node N, with the pointer to the second new node N_2. Again, this can only fail if another thread has done this earlier. In order to replace the pointer on the correct parent, we search for the parent (in the tree) of the split node once again. The second parent search may yield a different parent if the original parent was concurrently split or joined. After making the parent point to the two new nodes N_1 and N_2, it remains to set their state to NORMAL and return. The splitting process is completed.

If the original node N was determined to be the root, then a new root R with two new children N_1 and N_2 is created. Next, the B$^+$tree's root pointer is swapped from pointing to N to point to R. The details of the root's split code are relegated to Appendix E.1.

4.2 Nodes Join

Establishing master-slave relationship: We assume that the join is initiated by a sparse node N, denoted *master*. The chunk mechanism has frozen the node N and it has determined that this node has too few entries. To complete the join, the chunk lets the B$^+$tree find the *slave*. The B$^+$tree establishes a master-slave relationship and later the chunk mechanism joins the entries of both nodes. The B$^+$tree's *FindJoinSlave()* method is responsible for establishing master-slave relationship and returns the slave for the given master. Its code is presented in Algorithm 5 in Appendix C.2. The master-slave relationship establishing is described below.

The search for the neighboring node starts by finding the master's parent node P together with the pointers to the master's and its neighbor's entries. The parent node search fails only if the node N has already been deleted from the tree, in which case a slave has already been determined and this step is completed. Otherwise, the parent and a potential slave node M were found. The left-side neighbor is returned for all nodes except the left-most node, for which a right-side neighbor is returned. In order to establish the relationship, we first change N's freeze state from <FREEZE, NULL> to <REQUEST_SLAVE, M>. This CAS operation may fail if N's freeze state has been already promoted to JOIN, in which case N's final slave has already been set. This CAS operation may also fail if another slave was already chosen due to delay of this CAS command. In this case, we just use that slave.

After finding a potential slave, we attempt to set its freeze state to <SLAVE_FREEZE, N> and freeze it. For this purpose the *SetSlave()* method is invoked from the *FindJoinSlave()* method. The *SetSlave()* method's code and details are presented in Algorithm 6 in Appendix C.2. The *SetSlave()* method's algorithm is described in the next paragraph. After succeeding in setting the slave's freeze state, we change the master's state from <REQUEST_SLAVE, M> to <JOIN, M>, to enable the actual join attempts.

In order to enslave the slave M, we first attempt to CAS M's freeze state from <NORMAL, NULL> to <SLAVE_FREEZE, N>. After the CAS of the freeze state in the slave is successful, slave is frozen and we may proceed with the join. But M's freeze state isn't necessarily NORMAL: if it is not, then M is either still an infant or it is already frozen for some other reason. In the first case, we help M to become NORMAL and retry to set its freeze state. In the second case, we help to complete M's freeze. After the freeze on M is complete, M is frozen forever and is not suitable to serve as a slave. Therefore, M's enslaving has failed and the process of finding another slave must be repeated. A special case occurs when the potential slave M has a master freeze state as well and is concurrently attempting to enslave N for a join. This case can only happen with the two leftmost nodes and if we do not give it special care, an infinite run may result, in which each of the two nodes repeatedly tries to enslave the other. In order to break the symmetry, we check explicitly for this

case, and let the leftmost node among the two give up and become the slave, with a SLAVE_FREEZE state and a pointer to its master (which was originally meant to be enslaved for it). Finally, we finish M's enslaving by freezing M. After the freezing of the slave M is done, we continue with the join.

Merge: If the number of entries on the master and the slave is less than d, chunk mechanism creates a new single chunk to replace the master and the slave. We denote this operation *merge*. In this situation, the *InsertMergeNode()* method is called (via *CallForUpdate()*) by the chunk mechanism in order to insert the new node into the tree. At this point, a master-slave relationship has been established, both M and N have been frozen, and a new node N_1 has been created with the keys of both M and N merged. The code with all the details for the merge is presented in Algorithm 7 in Appendix C.2.

The merge starts by checking which of the original nodes (master N or slave M) has higher keys. We denote this node by N_{high} and the other node by N_{low}. Next, we look for N_{high}'s parent. If the parent is not found, then N_{high} must have already been deleted and we need to handle the node with the lower keys, N_{low}. Otherwise, we replace the parent's pointer to N_{high}, with a pointer to the new node N_1. Next, we handle the pointer to N_{low} at the parent by attempting to delete it. Finally, we turn the new node's freeze state to normal.

Special care is given to the root. We would like to avoid having a root with a single descendant, and that might happen when the two last descendants of a root are merged. In this case, we make the new merged node become the new root. (See the *MergeRoot()* method in Appendix E.2.)

Borrow: If the keys of two join nodes cannot fit a single node, they are copied into two new nodes. This operation is called *borrow*. The code and all the details for the borrow process is presented in Algorithm 8 in Appendix C.3. Recall that in the borrow case four nodes are involved: the master N, the slave M, the new node with the lower keys N_1 and the new node with the higher keys N_2. As in merge case, we start by finding the high and low keys' nodes, N_{high} and N_{low}, among the master and the slave.

We then take the following steps: (1) Insert a reference to N_1 to the parent node (with the maximal key on the N_1 as the key); (2) Change the parent entry pointing to N_{high} to point to N_2; (3) Delete the parent entry pointing to N_{low}.

4.3 Two Invariants

We mention two correctness invariants that may expose some of the correctness arguments behind the algorithm and help the reader understand the course of the algorithm.

Keys duplication. During the balancing operations described above, we sometimes make duplicates of keys appear in the tree, but at no time will a key appear to be absent. For example, after the first new node is inserted to the parent as part of the split, there are keys that reside simultaneously in two different nodes: all keys in the first new node are also still available in the old split node, which is still in the tree. Similarly, as part of the merge, when an old frozen node with higher keys is replaced with the new node, there are keys that appear twice: all keys in the old frozen node with lower keys now also appear in the new node. Recall that a search in our B$^+$tree is allowed to navigate through the tree and return the result, based on the data found on the frozen node.

This does not foil searches in the tree. Old searches may access keys in the old frozen node(s), but new searches can only access the new infant node(s). Furthermore, none of these nodes can be modified until the rebalance process terminates. The new node is an infant, which temporarily precludes modifications, and the old node is frozen, which precludes modifications permanently.

We should also note that the tree doesn't grow too big because of duplication. An invariant that we keep is

that there can only be two copies of a key in the tree. Thus, even though we may increase the size of the tree during the balancing, the increase will be at most by a factor of two. The factor-two increase is theoretical. In practice, the increase in the tree size is negligible. More about correctness and progress guaranties can be found in Appendix G.

Master-slave bond. We take special care to guarantee that the master and the slave keep the same parent up to the end of their join. Initially, the master and the slave are children of the same node P. However, there is always the chance that the parent node P is split so that these two children nodes do not have a single common parent anymore. This can subsequently lead the algorithm to make the tree structure inconsistent. Therefore, we enforce an invariant that the master and slave nodes must remain on the same parent. Namely, we do not allow the parent entries that point to a master and to its slave be separated into different nodes due to a parent's split or borrow, until new nodes replace the frozen master and slave. To this end, we take special care when the values of a frozen parent are divided into two new parents, and make sure that two such entries reside on a single new parent. Ensuring this variant is executed both on the parent splitting algorithm as well as on the children joining algorithm. First, at the parent side we check whether the descendants form a master and a slave and if they do, they are not placed on different new nodes. But the descendants may later enter a master and slave relationship, after this check was executed. Therefore, on the descendants' side, after declaring the intent of a master to enslave its neighbor (setting the master's state to REQUEST_SLAVE), we check that the master's parent is not in a frozen state. If it is, we help the parent to recover before continuing the descendants' join. This ensures that the parent split (or borrow) does not occur obliviously and concurrently with its descendants' join (or borrow).

4.4 Extensions to the Chunk Mechanism

The chunk interface requires some minor modifications over [3] to properly serve the B$^+$tree construction in this paper. Probably the most crucial modification arises from the need to deal with an ABA problem that arises during insertions and deletions of entries to the chunk of an internal node in the tree. The concern is that an insert or a delete may succeed twice due to a helper thread that remains idle for a while. Consider, for example, a merge and a subsequent delete of an entry at the parent node. Suppose that one thread executes the delete, but a second thread attempts this delete later, after the same key (with a different descendant) has been entered to the parent again. Thus, a delete should only succeed when the entry still points to the frozen node. As for inserts, we need to avoid reentering a pointer to a child node that has actually been frozen and deleted while the updating thread was stalled. To solve such problems, we add versioning to the *nextEntry* word in the chunk's linked-list. This eliminates the ABA problem, as a delayed CAS will fail and make us recheck the node that we attempt to insert or delete and discover that it has already been frozen. The extensions to the chunk mechanism are described and discussed in Appendix F.

5 Implementation and Results

We have implemented the lock-free B$^+$tree presented in this paper as well as the lock-based B$^+$tree of [16] in the C programming language. The lock-free design in this paper can be optimized in many ways. However, we have implemented it as is with no further optimizations. The operations of the lock-based B$^+$tree progress in a top-down direction. During the descent through the tree, *lock-coupling* [1] is employed, i.e., a child is locked before its parent is unlocked. Exclusive locks on the nodes are used for insert and delete operations, and shared locks are used for search operations. Deadlock-freedom is guaranteed by a proactive approach to rebalancing that splits full nodes or join sparse ones, while going down the path to the leaves.

We ran the experiments on the SUN FIRE machine with an UltraSPARC T1 8-core processor, each core running

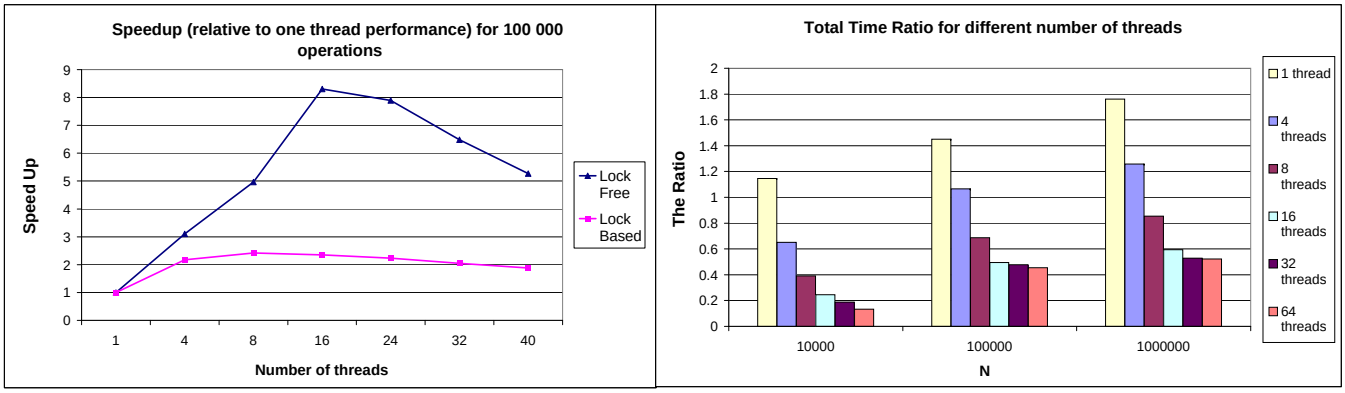

Figure 3: The empirical results.

4 hyper-threads, running Solaris 10. Overall, the eight cores, with quad hyper-threading simulates the concurrent execution of 32 threads. In both implementations the size of a B$^+$tree node was set to the machine's virtual page size, i.e., 8KB. In each test we start with a tree with N random keys in the range $[0, 2^{18}]$ already inserted to it, and during the test, we apply N operations on it. If the test runs T threads, then each executes N/T operations. The parameter N was varied among 10^4, 10^5 and 10^6. The operations consisted of insertions, deletions and searches in parallel, out of which 20% were insertions, 20% were deletions, and the remaining 60% were searches. All the threads start simultaneously at the beginning and we measure the time it takes to complete all operations by all threads.

The left graph of Figure 3 depicts the ratio between the time it took to complete the runs on the lock-free implementation as compared to the lock-based implementation. A result higher than 1 means that the lock-free implementation is slower. Clearly, the lock-free implementation outperforms the lock-based implementation when contention is not low. Note that contention increases as the tree gets smaller and as the number of threads increases. Also, the results show that the average cost of an operation increases as the tree gets larger, because rebalancing may ascend to higher levels. Such costs are heavier for the lock-free tree, but this overhead is offset by lock-freedom efficiency when contention kicks in. The right graph of Figure 3 depicts the speedup, which clearly shows that the lock-free algorithm is more scalable.

The weaker performance of the lock-free tree for low contention can be ameliorated by simple optimizations. For example, during the split, each thread helping the split copies the entries from the old node to a newly created private node and only one of these new nodes eventually replaces the old node and joins the tree. While threads can cooperate to perform copying, we decided to avoid it in this version because it complicates the design.

6 Conclusion

We presented a lock-free dynamic B$^+$tree, which builds on CAS synchronization. The construction is composed of a chunk mechanism that provides the low-level node implementation, including splitting and joining a node, and then a higher level mechanism which handles the operations at the tree level. The two mechanisms and their interface are lock-free. To the best of our knowledge, this is the first design of a lock-free balanced search tree for a general platform. Results indicate better handling of contention and higher scalability when compared to the lock-based version of the B$^+$tree. We have also proven the correctness (with respect to linearizability) of the algorithm and its and lock-freedom property.

References

[1] R. Bayer and M. Schkolnick, *Concurrency of Operations on B-trees*, Acta Informatica 9, 121. 1977.

[2] M. A. Bender, J. T. Fineman, S. Gilbert and B. C. Kuszmaul, *Concurrent Cache-Oblivious B-trees*, In Proceedings of the 17th ACM Symposium on Parallel Algorithms and Architectures, 2005

[3] A. Braginsky and E. Petrank, *Lock-Free Linked Lists with Improved Locality*, Proc. 12th Int. Conf. on Distributed Computing and Networking (ICDCN), 2011

[4] S. Chen, P. B. Gibbons, T. C. Mowry, G. Valentin, *Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance*, In Proceedings of the 2002 ACM SIGMOD International Conference (2002) 157-168

[5] D. Comer, *The Ubiquitous B-tree*, ACM Computing Surverys, 11(2), 1979

[6] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, *Non-blocking Binary Search Tree*, In Proceedings of the 29th Annual Symposium on Principles of Distributed Computing, 2010

[7] K. Fraser, *Practical Lock-Freedom* , Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004

[8] T. L. Harris, *A Pragmatic Implementation of Non-Blocking Linked-Lists*, DISC 2001

[9] M. Herlihy and N. Shavit, *"The Art of Multiprocessor Programming"* Morgan Kaufman 2008

[10] M. Herlihy, J. Wing, *Linearizability: a correctness condition for concurrent objects*, Trans. on Prog. Lang. and Syst. 1990

[11] M. Herlihy, *Wait-free synchronization*, ACM Transactions on Programming Languages and Systems, 13(1):124149, January 1991

[12] M. M. Michael, *Safe Memory Reclamation for Dynamic Lock-Free Objects using Atomic Reads and Writes*, In Proceedings of the 21st Annual Symposium on Principles of Distributed Computing, 2002

[13] M. M. Michael, *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*, IEEE Transactions on Parallel and Distributed Systems, 15(6):491504, June 2004.

[14] E. Petrank, M. Musuvathi, and B. Steensgaard, *Progress Guarantee for Parallel Programs via Bounded Lock-Freedom*, In PLDI'09, 144-154, 2009

[15] J. Rao and K. A. Ross, *Cache Conscious Indexing for Decision-Support in Main Memory*, In Proceedings of the 25th VLDB Conference, 1999.

[16] O. Rodeh, *B-trees, Shadowing, and Clones*, ACM Transactions on Storage Journal, 2008.

[17] IBM *System/370 Extended Architecture, Principles of Operation*, 1983. IBM Publication No. SA22-7085.

[18] R. K. Treiber, *Systems Programming: Coping with Parallelism*, Research report RJ 5118, IBM Almaden Research Center, 1986.

[19] J. D. Valois, *Lock-free Linked Lists using Compare-and-Swap*, In Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, pages 214-222, 1995.

A Linearization Points

When designing a concurrent data structure, it is important to spell out the linearization points for the different operations. This is done in this section. The B^+tree methods all have a similar pattern of operation: they traverse the B^+tree to find the relevant leaf node, and then call the appropriate chunking methods on the leaf's chunk. Thus the linearization points of the B^+tree are typically based on the linearization points defined for the chunk in [3].

Search linearization point: The linearization point of the search operation is exactly the linearization point of the leaf's chunk search, as in [3]. In particular, if the leaf is not frozen, then the linearization point follows that of the underlying linked-list in the leaf's chunk, and if the leaf is frozen then the linearization point is set to be the point in which the chunk became frozen. As the freezing mechanism is not instantaneous, we need to define a point in the freezing process more accurately for the linearization point. We follow [3] and set the linearization point to be the point in the freeze process by which all the frozen bits have been set and also the internal list of the freezing node has been stabilized. Define this point as the *freezing point*. The freezing process of a chunk is explained in Section F and more thoroughly in [3]. Formally, consider the linearization point of the search of the linked-list that is inside the chunk of the leaf (as defined by Harris [8]). If the chunk's linked-list search linearization point occurs before the freezing point, then that is also the linearization point of the overall tree search. If the chunk's linked-list linearization point happens after the freezing point, then we define the overall tree search linearization point to be the later point between the freezing point and the point in which the search started. The latter maximum makes sure that the linearization point happens during the execution of the search.

Justifying this choice for non-frozen node is straightforward. As for frozen nodes, we note that the frozen node may be replaced with a new node during the search execution and various actions may be applied on the new node. But at the freezing point, we know that the values of the frozen node exist only in the frozen node and are properly represented by the view of the frozen node.

The delicate case is when the search starts after the freezing point and still gets to the frozen leaf and completes the search there. In this case, since the search ends up in this leaf, we know that a new node that replaces this leaf (following the end of the freeze) has not yet been modified while the search traversed the tree, because the rebalancing operation has not yet terminated at that point. Therefore the new node has definitely not been modified when the search started, and the frozen values represent correctly the state of the tree at that point in time.

Insert and delete linearization points: Unlike the analysis of the search operation, frozen nodes are not hazardous for the insert's and delete's initial tree traversing. If an insert or delete arrive at a frozen leaf, than the *InsertToChunk()* or the *DeleteInChunk()* methods will redirect the operation (after helping the frozen node) to a non-frozen leaf node. Intuitively, the insert operation is assumed to be finished when a leaf including the new key is reachable from the root via data pointers. Similarly, the delete operation is assumed to be finished when a leaf excluding an old key is reachable from the root via data pointers. In a worst-case, this may require more than just handling a freeze.

There are three cases possible here. First, if the insert or delete operation doesn't cause a rebalancing activity (split, merge, borrow, or copy), than the linearization point is simply determined to be the leaf's chunk linearization point. Second, if a rebalancing (by freezing) occurs and if the thread performing the insert or delete operation has its operation executed in the node that replaces the frozen node, then the linearization point of the operation becomes the linearization point of the insert operation of the new node to the parent of the frozen node (replacing the frozen node with the new one). Note that this definition may be recursive if the parent requires rebalancing for the insertion. The third case is when the result of this operation is not reflected in the

Algorithm 1 Finding the relevant leaf node given a key.

```
Node* FindLeaf (key) {
 1: node = btree→root;
 2: while ( node→height != 0 ) {                                   // Current node is not leaf
 3:     Find(&(node→chunk), key); node = cur→data;                 // entry's data field is a pointer to the child node
 4: }
 5: return node;                                                   // current node is leaf
}
```

Algorithm 2 Find a parent of a node, given a key located on it and a pointer to the child

```
Node* FindParent (key, Node* childNode, Entry** prntEnt, Entry** slaveEnt) {
 1: node = btree→root;
 2: while (node→height != 0) {                                     // Current node is not leaf
 3:     Find(&(node→chunk), key);
 4:     if ( childNode == cur→data ) {          // Check whether we found exactly the entry that points to the child
 5:         *prntEnt = cur;
 6:         if( slaveEnt != NULL ) {            // Look for the child's neighbor, check if the current entry is the leftmost
 7:             if( prev==&(node→chunk→head) ) *slaveEnt=next; else *slaveEnt=EntPtr(prev);
 8:         } // end of if child neighbor was needed
 9:         if (node→freezeState == INFANT) helpInfant(node);      // Help infant parent node
10:         return node;
11:     } // end of if child was found
12:     node = cur→data;
13: } // end of while current node is not leaf
14: return NULL;                                                   // Current node is leaf, no parent found
}
```

node that replaces the frozen one. In this case, we again define the linearization point recursively, setting it to be the linearization point of the re-attempted operation on the new node that replaced the frozen one.

B B$^+$tree supporting methods

Before detailing the full B+ tree code in the following appendixes, we first describe in detail the supporting methods, *FindLeaf()* and *FindParent()*, which are used by other B+ tree methods. *FindLeaf()* finds the relevant leaf in which a given key may reside. This method is invoked by main B$^+$tree's interfaces: *SearchInBtree()*, *InsertToBtree()* and *DeleteFromBtree()*.

The *FindLeaf()* method is specified in Algorithm 1. Note that this search never fails, because each key may only belong to one leaf of the tree, depending on the keys and pointers in the internal nodes. An appropriate leaf can always be returned, even if the key does not exist in it. The procedure starts from the root and ends when a leaf is found. At each step, it uses the *Find()* operation of the chunk mechanism to locate the appropriate pointer for the next descent. The *Find()* method of the chunk mechanism sets a global pointer **cur* to the entry with the minimal key value that is larger than or equal to the input key. This is exactly the entry whose pointer should be followed when descending the tree.

The second supporting method is *FindParent()*. When a split or a join occurs, we may need to find the parent of the current node in order to modify its pointers. Furthermore, we may need to find an adjacent node as a partner for a join, when a node becomes too sparse. The *FindParent()* method is presented in Algorithm 2.

Algorithm 3 (a) Search, Insert, and Delete – High Level Methods.

```
(a) BOOL SearchInBtree (key, *data) {
 1: Node* node = FindLeaf(key);
 2: return SearchInChunk(&(node→chunk), key, data);
}
(b) BOOL InsertToBtree (key, data) {
 1: Node* node = FindLeaf(key);
 2: if (node→freezeState == INFANT) helpInfant(node);          // Help infant node
 3: return InsertToChunk(&(node→chunk), key, data);
}
(c) BOOL DeleteFromBtree (key, data) {
 1: Node* node = FindLeaf(key);
 2: if (node→freezeState == INFANT) helpInfant(node);          // Help infant node
 3: return DeleteInChunk(&(node→chunk), key);
}
```

It is given a pointer to the child node, but also a key that exists on the child node, which allows navigation towards the child on the tree. It either finds the parent node, or returns NULL if the child can no longer be found in the tree (i.e., it was removed from the tree before this search was completed). A leaf node cannot be a parent; therefore we return NULL if we reach a leaf. Otherwise, we stop when we find an entry in a node whose descendant is the input node. At that point we know we found the parent node. The discovered entry in the parent node is returned, using the parameter *prntEnt*.

FindParent() may also provide a neighbor to be enslaved for a join. *FindParent()* looks for this neighbor when the parameter *slaveEnt* is not NULL. It always returns a pointer to the left neighbor (of the child node), unless the child is the leftmost child of its parent and then it returns the right-side neighbor. Technically, due to the uses of this method, it doesn't simply return the pointer to the neighbor. Instead, it returns a pointer to the parent entry that points to the neighbor, using the parameter *slaveEnt*. To find this neighbor, we further exploit the method *Find()* of the chunk mechanism. The *Find()* method sets a global pointer **cur* to the entry with the minimal key value that is larger than or equal to the input key. But the *Find()* method also sets two more pointers: ***prev* and **next*. The global (indirect) pointer ***prev* points to the entry that precedes the entry pointed to by **cur* [6]. The entry that follows the one pointed to by **cur* is returned in a global pointer **next* (if such an entry exists). When *Find()* is used properly, the left neighbor of the child node will be point to from the global ***prev* pointer initiated by *Find()*, unless the child node is the leftmost child, in which case *prev* will point to the head entry. If it does point to the header, then we just return the pointer *next*.

If we find a parent, we also check whether its freeze state is INFANT; if it is, we help the infant parent to become a normal node before returning it. In most cases, the *FindParent()* is used to find a parent and then to apply insert, delete or replace operations on its chunk. Those operations are not allowed to be done on an infant parent.

C Code and Detailed Explanations for Split, Merge and Borrow

The code for the B^+tree interfaces is presented in Algorithm 3 and was described in Section 2.4. Here we describe the code and all the details for the balancing B^+tree operations.

[6] Eventually, ***prev* pointer points to the inner field *next* of the entry that precedes the entry pointed to by **cur*. We use *EntPtr()* to convert it to a normal pointer to the previous entry.

Algorithm 4 The split of a non-root node

```
void InsertSplitNodes (Node* node, sepKey) {      // sepKey is the highest key in the low-values new node
 1: Entry* nodeEnt;                                // Pointer to the parent's entry pointing to the node about to be split
 2: Node* n1 = node→new;                           // Pointer to the new node that holds the lower keys
 3: Node* n2 = node→new→nextNew;                   // Pointer to the new node that holds the higher keys
 4: maxKey = getMaxKey(node);                      // Get maximal key on the given frozen node
 5: if ((parent = FindParent(sepKey, node, &nodeEnt, NULL)) != NULL) {
 6:    InsertToChunk(parent→chunk, sepKey, n1);    // Can only fail if someone else completes it before we do
 7: }
 8: if ((parent = FindParent(maxKey, node, &nodeEnt, NULL)) != NULL) {
 9:    ReplaceInChunk(parent→chunk, nodeEnt→key,   // Can only fail if someone else completes it before we do
10:                   combine(nodeEnt→key, node), combine(nodeEnt→key, n2));
11: }
12: // Update the states of the new nodes from INFANT to NORMAL.
13: CAS(&(n1→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
14: CAS(&(n2→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
15: return;
}
```

C.1 Node Splits

After the chunk mechanism executes a split, the original node N is frozen, and N's *new* field points to the new node N_1, which holds the lower half of the keys from the old node N. The field N_1.*nextNew* points to the second new node N_2, which holds the higher half of the keys from the old node N. The two new nodes' freeze states are initiated to INFANT so that no updates can occur on these nodes until they are inserted into the tree.

The split of the root is relegated to Appendix E.1. The code for completing the split of a (non-root) node is presented in Algorithm 4. The *InsertSplitNodes()* method is invoked by the *CallForUpdate()* method. The *InsertSplitNodes()* method receives a pointer to the frozen node whose split needs to be completed, and the *sepKey* parameter. The *sepKey* parameter holds the middle key that "separates" the two new nodes. The *sepKey* key and all lower keys have been placed in the first new node (*n1*), and all keys higher than *sepKey* have been placed in the second new node (*n2*).

The split starts by searching for the parent (in the tree) of the split node, using *sepKey* for navigating in the tree. If the parent cannot be found, then the input node is no longer on the path for *sepKey* in the tree. This happens if the new node with the low level key was properly inserted by some other thread. Otherwise, and having found the parent P, we modify the parent to point to the new nodes. We want to insert a new link to the first new node (with the low level keys) into P. This is done by inserting a new entry to the parent node. If the insert fails, it means that someone else has inserted this entry to the chunk and it is fine to continue. Therefore, we do not check wether the insert succeeded.

Note that after the first new node is inserted to the parent, there are keys that reside simultaneously in two different nodes: all keys in the first new node are also still available in the old split node, which is still in the tree. This does not foil searches in the tree. Old searches may access keys in the old node, but new searches can only access the new node. Furthermore, none of these nodes can be modified until the split process terminates. The new node is infant, which temporarily precludes modifications, and the split node is frozen, which precludes modifications permanently.

After the first new node is in place, we replace the pointer in the parent node, which points to the frozen node, with the pointer to the second new node (Line 9). Again, this can only fail if another thread has done this

Algorithm 5 The code of finding a node partner for a join in the lock-free B$^+$tree.

```
Node* FindJoinSlave(Node* master) {
 1: Node* oldSlave = NULL;
 2: start: anyKey = master→chunk→head→next→key;                    // Obtain an arbitrary master key
 3: if ( (parent = FindParent(anyKey, master, &masterEnt, &slaveEnt)) == NULL)   // If master is not in the B+tree;
 4:     return master→<*, joinBuddy>;                  // thus its slave was found and is written in the joinBuddy
 5: slave=slaveEnt→data;                                          // Slave candidate found in the tree
 6: // Set master's freeze state to <REQUEST_SLAVE, slave>; oldSlave is not NULL if the code is repeated
 7: if ( oldSlave == NULL ) expState = <FREEZE, NULL>; else expState = <REQUEST_SLAVE, oldSlave>;
 8: if ( !CAS(&(master→<freezeState, joinBuddy>), expState, <REQUEST_SLAVE, slave>) ) {
 9:     // Master's freeze state can be only REQUEST_SLAVE, JOIN or SLAVE_FREEZE if the roles were swaped
10:     if ( master→<freezeState,*> == <JOIN,*> ) return master→<*, joinBuddy>;
11: }
12: slave = master→<*, joinBuddy>;                          // Current slave is the one pointed by joinBuddy
13: // Check that parent is not in a frozen state and help frozen parent if needed
14: if ( (parent→<freezeState,*> != <NORMAL,*>) && (oldSlave == NULL) ) {
15:     Freeze(parent, 0, 0, master, NONE, &result); oldSlave = slave; goto start;
16: }
17: // Set slave's freeze state from <NORMAL, NULL> to <SLAVE_FREEZE, master>
18: if ( !SetSlave(master, slave, anyKey, slave→chunk→head→next→key) ) {oldSlave = slave; goto start;}
19: // We succeed to get the slave update master
20: CAS(&(master→<freezeState, joinBuddy>), <REQUEST_SLAVE, slave>, <JOIN, slave>);
21: if (master→<freezeState,*> == <JOIN,*>) return slave; else return NULL;
}
```

earlier. The *ReplaceInChunk()* method finds the entry with key and data as in its third argument and replaces it with key and data as in its last argument. (The *combine()* method syntactically combines the key and the data values into a single word.) In order to invoke *ReplaceInChunk()* on the correct parent, we search for the parent (in the tree) of the split node, using the maximal key of that node for navigating in the tree. The second parent search may yield a different parent if the original parent was concurrently split or joined. After making the parent point to the two new nodes, it remains to set their state to NORMAL and return. The split is done.

Note that the chunk insert in Line 6 might create a split in the parent, which will recursively call *CallForUpdate()*, and the splits may roll up the tree.

C.2 Merges

Establishing master-slave relationship: We assume that the merge is initiated by a node N, denoted *master*. The chunk mechanism has frozen the node N and it has determined that this node has too few entries. But the merge can not be completed by the chunk mechanism because the *slave* is required. Thus, we start by establishing a master-slave relationship, in order to share the entries with this node. To this end, the chunk mechanism calls the *FindJoinSlave()* method, presented in Algorithm 5. This method returns the slave for the given master.

The *FindJoinSlave()* method starts by calling the *FindParent()* method, which returns a pointer to the master's parent node together with the pointers to the master's and its potential slave's entries. The parent node search fails only if the node N has already been deleted from the tree, in which case a slave has already been determined and can be retrieved from the *joinBuddy* field of N. Otherwise, the parent and a potential slave node M were returned by *FindParent()*. In order to establish the relationship we first change N's freeze state from

Algorithm 6 Setting the slave's freeze state for a join in the lock-free B$^+$tree.

```
Bool SetSlave(Node* master, Node* slave, masterKey, slaveKey) {
 1: // Set slave's freeze state from <NORMAL, NULL> to <SLAVE_FREEZE, master>
 2: while (!CAS(&(slave→<freezeState,joinBuddy>),<NORMAL,NULL>,<SLAVE_FREEZE,master>)){
 3:    // Help slave, different helps for frozen slave and infant slave
 4:    if (slave→<freezeState, *> == <INFANT, *>) { helpInfant(slave); return FALSE; }
 5:    elseif (slave→<freezeState, *> == <SLAVE_FREEZE,master>) break;          // Modification is done
 6:    else {                                  // The slave is under some kind of freeze, help and look for new slave
 7:       // Check for a special case: two leftmost nodes try to enslave each other, break the symmetry
 8:       if ( slave→<freezeState, *> == <REQUEST_SLAVE, master> ) {
 9:          if (masterKey < slaveKey) {          // Current master node is left sibling and should become a slave
10:             if ( CAS(&(master→<freezeState,joinBuddy>), <REQUEST_SLAVE,slave>,
11:                                 <SLAVE_FREEZE,slave>) ) return TRUE; else return FALSE;
12:          else                          // Current master node is right sibling and the other node should become a slave
13:             if ( CAS( &(slave→<freezeState,joinBuddy>), <REQUEST_SLAVE,master>,
14:                                 <SLAVE_FREEZE,master>) ) return TRUE; else return FALSE;
15:       } // end case of two leftmost nodes trying to enslave each other
16:       Freeze(slave, 0, 0, master, ENSLAVE, &result);                  // Help in different freeze activity
17:       return FALSE;
18:    } // end of investigating the enslaving failure
19: } // end of while
20: MarkChunkFrozen(slave→chunk); StabilizeChunk(slave→chunk); // Slave enslaved successfully. Freeze the slave
21: return TRUE;
}
```

<FREEZE, NULL> to <REQUEST_SLAVE, M>. (Recall that the *joinBuddy* field and the freeze state field are located in one word.) If this is not our first try, the field may hold a previous slave pointer (*old_slave*) that we could not enslave. In this case, we change the value of N's freeze state from <REQUEST_SLAVE, old_slave> to <REQUEST_SLAVE, M>, where M is the new potential slave. The CAS operation in Line 8 may fail if N's freeze state has already been promoted to JOIN or became SLAVE_FREEZE due to swapping the master-slave roles as it will be explained below. In these cases N's final slave has already been set in the *joinBuddy* field of N. The CAS operation in Line 8 may also fail if another slave was already chosen due to delay of this CAS command. In this case, we just use that slave (Line 12).

For correctness, we enforce the invariant that the parent entries that point to the master and the slave always reside on the same node. Namely, we do not allow entries that point to a master and to its slave be separated into different nodes due to a parent's split or borrow, until new nodes replace the frozen master and slave. To this end, we take special care when the values of frozen parent are divided into two new parents, and make sure that two such entries reside on a single new parent. A standard race occurs because after checking whether the descendants form a master and a slave and deciding that they are not, the descendants may later enter a master and slave relationship. A standard solution is to also check at the descendants' side. Namely, after declaring the intent of a master to enslave its neighbor (setting the state to REQUEST_SLAVE), we check that the master's parent is not in a frozen state. If it is, we help parent to recover before continuing the descendants' join (Lines 14, 15). This ensures that the parent split (or borrow) does not occur obliviously to and concurrently with its descendants' merge (or borrow).

After finding a potential slave, we attempt to set its freeze state to <SLAVE_FREEZE, N> and freeze it. This is done in the *SetSlave()* method presented in Algorithm 6 and explained in the next paragraph. If this action is not successful, we will try from scratch and look for another potential slave. After succeeding in setting the

Algorithm 7 The merge of two old nodes to one new node

```
void InsertMergeNode (Node* master) {
 1: Node* new = master→new;                                          // Pointer to the new node.
 2: Node* slave = master→<*, joinBuddy>;
 3: maxMasterKey = getMaxKey(master); maxSlaveKey = getMaxKey(slave);          // Both nodes are frozen
 4: if ( maxSlaveKey < maxMasterKey ) {                  // Find low and high keys among master and slave
 5:    highKey = maxMasterKey; highNode = master; lowKey = maxSlaveKey; lowNode = slave;
 6: } else { highKey = maxSlaveKey; highNode = slave; lowKey = maxMasterKey; lowNode = master; }
 7: if ((parent = FindParent(highKey, highNode, &highEnt, NULL)) != NULL) {
 8:    highEntKey = highEnt→key;                  // Change the highest key entry to point on new node
 9:    ReplaceInChunk(parent→chunk, highEntKey,   // If replacing fails, the parent chunk was updated by a helper
10:                   combine(highEntKey, highNode), combine(highEntKey, new));          // continue anyway
11: } // If high node cannot be found continue to the low
12: if ((parent = FindParent(lowKey, lowNode, &lowEnt, NULL)) != NULL) {
13:    if (parent→root) MergeRoot(parent, new, lowNode, lowEnt→key);
14:    else DeleteInChunk(&(parent→chunk), lowEnt→key, lowNode);          // lowNode is the expected data
15: } // If also low node can no longer be found on the tree, then the merge was completed (by someone else).
16: // Try to update the new node state from INFANT to NORMAL.
17: CAS(&(new→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
18: return;
}
```

slave's freeze state, we change the master's state from <REQUEST_SLAVE, M>, to <JOIN, M> to enable the actual join attempts.

The *SetSlave()* method attempts to CAS the freeze state of the slave M from <NORMAL, NULL> to <SLAVE_FREEZE, N>. If the CAS of the freeze state in the slave is successful, we may proceed with the join. But M's freeze state isn't necessarily NORMAL: if it is not, then M is either still an infant or it is already frozen for some other reason. In the first case, *SetSlave()* helps M to become NORMAL and retries to set M's freeze state. In the second case, it helps to complete M's freeze. After finishing the freeze on M, M is frozen forever and is not suitable to serve as a slave. Therefore, failure is returned by *SetSlave()* and another slave must be found. A special case occurs when the potential slave M has a master freeze-state as well and is concurrently attempting to enslave N for a join. This case can only happen with the two leftmost nodes and, if special care is not taken, an infinite run may result, in which each of the two nodes repeatedly tries to enslave the other. In order to break the symmetry, we check explicitly for this case, and let the leftmost node among the two give up and become the slave, with a SLAVE_FREEZE state and a pointer to its master (which was originally meant to be enslaved for it). The *FindJoinSlave()* checks for this case in its last line. If it is successful in turning the freeze state of the master into JOIN, then all is well. Otherwise, and given that *SetSlave()* completed successfully, then it must be the case that the master has become a slave. In this case, no slave is returned, and the returned NULL value tells the calling method (in the chunk mechanism) to treat the master as the slave.

Finally, we finish the *SetSlave()* by freezing the slave in Line 20, so that the join can continue. The method *MarkChunkFrozen()* marks all entries of a node frozen by setting a designated bit in each entry. After the entries are marked frozen, the *StabilizeChunk()* method ensures that no changes occur on this node. At this point the slave has been enslaved and frozen.

Merge: The *InsertMergeNode()* method is called (via *CallForUpdate()*) by the chunk mechanism in order to insert the new node into the tree. At this point, a master-slave relationship has been established, both M and N have been frozen, and a new node has been created with the keys of both M and N merged. (The case

Algorithm 8 The merge of two old nodes to two new nodes

```
void InsertBorrowNodes (Node* master, sepKey) { // sepKey is the highest key in the low-values new node
 1: Node* n1 = node→new;                                  // Pointer to the new node that holds the lower keys
 2: Node* n2 = node→new→nextNew;                          // Pointer to the new node that holds the higher keys
 3: Node* slave = master→<*, joinBuddy>;
 4: maxMasterKey = getMaxKey(master); maxSlaveKey = getMaxKey(slave);     // Both master and slave are frozen
 5: if ( maxSlaveKey < maxMasterKey ) {                   // Find nodes with low and high keys among master and slave
 6:    highKey = maxMasterKey; oldHigh = master; lowKey = maxSlaveKey; oldLow = slave;
 7: } else { highKey = maxSlaveKey; oldHigh = slave; lowKey = maxMasterKey; oldLow = master; }
 8: if ( lowKey < sepKey ) sepKeyNode = oldHigh;          // sepKey located on the higher old node
 9: else sepKeyNode = oldLow;                             // sepKey located on the lower old node
10: if ((insertParent = FindParent(sepKey,sepKeyNode, &ent, NULL)) != NULL) {
11:    InsertToChunk(insertParent→chunk, sepKey,n1);     // Insert reference to the new node with the lower keys
12: }
13: if ((highParent = FindParent(highKey, oldHigh,&highEnt, NULL)) != NULL) {   // Find the parent of the old node
14:    ReplaceInChunk(highParent→chunk, highEnt→key,     // with the higher keys and change it to point to the new
15:                   combine(highEnt→key,oldHigh), combine(highEnt→key,n2));   // node with the higher keys
16: }
17: if ((lowParent = FindParent(lowKey, oldLow, &lowEnt, NULL)) != NULL) {     // Delete, currently duplicated,
18:    DeleteInChunk(&(lowParent→chunk),lowEnt→key, oldLow);                   // reference to the old low node
19: }
20: // Try to update the new children states to NORMAL from INFANT
21: CAS(&(n1→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
22: CAS(&(n2→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
23: return;
}
```

in which there are two new nodes is handled by a similar method called *InsertBorrowNodes()*, described in Appendix C.3.) The code for completing of the merge is presented in Algorithm 7.

The *InsertMergeNode()* method's input parameter is a pointer to the master, this master's slave can be found in the *joinBuddy* field on the master. The *InsertMergeNode()* method starts by checking which of the original nodes (master and slave) has higher keys. Denote this node by *highNode*. Note that the master and the slave are frozen and thus immutable. Next, *FindParent()* is invoked on *highNode*. If the parent is not found, then *highNode* must have already been deleted and we need to handle the old node with the lower keys, *lowNode*. Otherwise, we adapt the parent's reference to *highNode*, to point to the new node. Next, we handle the pointer to *lowNode* at the parent by attempting to delete it. Finally, we turn the new node's freeze status to normal.

Special care is given to the root. We would like to avoid having a root with a single descendant, which can occur when the two descendants of a root are merged. In this case, we make the merged node become the new root. If merged node parent is found to be root, the *MergeRoot()* method is invoked from *InsertMergeNode()* instead of deleting the pointer to *lowNode* at the parent. This is so, because deleting an entry from the root may lead us to having a single root descendant. (See the *MergeRoot()* method in Appendix E.2.)

Note that after the *highNode* has been replaced with the new node, the keys in the two nodes are repeated (the old lower frozen node and the new node), but any search will find the keys at the same state in both places, as when the merge began. The concurrent insertions and deletions will help the two nodes due to freezing or infancy (similar to the split case).

Algorithm 9 Interface for updating the B$^+$tree on frozen node recovery

```
void CallForUpdate ( freezeState, Node* node, sepKey ) {
 1: Node* n1 = node→new;                                  // Pointer to the new node that holds the lower keys.
 2: Node* n2 = node→new→nextNew;                          // Pointer to the new node that holds the higher keys.
 3: switch ( freezeState ) {
 4:     case COPY:
 5:         if ( node→root ) { CAS(&(n1→root), 0, 1); CAS(&(btree→root), node, n1); }
 6:         else if ((parent=FindParent(node→chunk→head→next→key,node,&nodeEnt,NULL))!=NULL)
 7:             ReplaceInChunk(parent→chunk, combine(nodeEnt→key,node), combine(nodeEnt→key, n1));
 8:         CAS(&(n1→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
 9:         return;
10:     case SPLIT:
11:         if ( node→root ) SplitRoot(node, sepKey, n1, n2); else InsertSplitNodes(node, sepKey);
12:         return;
13:     case JOIN:
14:         if ( n2 == NULL) InsertMergeNode(node);          // If there is only one new node, then merge,
15:         else InsertBorrowNodes(node, sepKey);            // otherwise call the borrow operation.
16:         return;
17: }
}
```

C.3 Borrow

In this appendix we present the details of the *borrow* case, in which the keys in two join nodes are copied into two new nodes. The code for the *InsertBorrowNodes()* method responsible for the borrow is presented in Algorithm 8. As in merge case the *InsertBorrowNodes()* method is invoked from *CallForUpdate()* when a need for the borrow is encountered. The input parameters are a pointer to the master node and a separation key, which is the highest key on the new node that contains the lower keys. Recall that in the borrow case four nodes are involved: the master, the slave, the new node with the lower keys (in the code denoted *n1*) and the new node with the higher keys (in the code denoted *n2*). We start by finding the high and low keys' nodes, among master and slave (Lines 4-7), similar to the merge.

Next we are going to insert the new entry pointing to to the new node with the lower part of the keys. When we search for the parent for that, we carefully choose the child we supply for the search. The key of the new parent entry will be the separation key. This is the highest key on *n1* and therefore resides on the *n1* node. The data will be the pointer to *n1*. The separation key separates the two new nodes after the borrow. It may previously have been located either on the (old) high or the low keys' frozen node. It is important to supply to the *FindParent()* method the child on which the separation key was originally located. The check is done in Lines 8, 9.

We then follow these steps: (1) Insert a reference to *n1* to the parent node (2) Change the parent entry pointing to the high keys' frozen node to point to *n2* (3) Delete the parent entry pointing to the low keys' frozen node. During this execution, some entries will be duplicated. Namely, keys will appear twice in the tree, once in the old node and again in the new node (both times with same associated data). But none of the keys will appear to be absent, and therefore search correctness is maintained (similarly to the split and merge cases). Finally, we update the new nodes' freeze states.

Algorithm 10 Helping the infant node.

```
void helpInfant (Node* node) {
 1: creator = node→creator; creatorFrSt = creator→<freezeState, *>;
 2: Node* n1 = creator→new;                          // Pointer to the new node that holds lower keys from creator
 3: Node* n2 = creator→new→nextNew;                  // Pointer to the new node that holds higher keys from creator
 4: sepKey = getMaxKey(n1);                                                          // n1 is never NULL
 5: if ( (n1→<freezeState,*>) != INFANT ) {                                          // Check low is still infant
 6:    if (n2) CAS(&(n2→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
 7:    return;
 8: }
 9: if ( (creator→root) && (creatorFrSt == SPLIT) ) {    // If this is root split only children's state correction is needed
10:    CAS(&(n1→<freezeState,joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
11:    CAS(&(n2→<freezeState,joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
12:    return;
13: }
14: switch ( creatorFrSt ) {                                                        // Can be only COPY, SPLIT or JOIN
15:    case COPY:
16:       CAS(&(node→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>); return;
17:    case SPLIT:
18:       InsertSplitNodes(creator, sepKey); return;
19:    case JOIN:
20:       if (n2 == NULL) InsertMergeNode(creator); // If freeze state is JOIN and there is one new node, help in merge
21:       else InsertBorrowNodes(creator, sepKey);                                  // help in borrowing
22:       return;
23: } // end of switch
}
```

D Redirection of the call for an update

The B$^+$tree interfaces (discussed in Section 2.4) start by finding the relevant leaf of the tree and then executing the search, insert, or delete on the leaf's chunk . The insert or delete operations can cause the chunk to reach the minimal or maximal boundaries, after which the node will be marked frozen, stabilized, and a new node or nodes will be created according to the final (frozen) number of entries. Finally, the chunk mechanism invokes the *CallForUpdate()* function, which inserts the new nodes into the B$^+$tree instead of the old frozen node or nodes, to complete the rebalancing. The *CallforUpdate()* method is presented in Algorithm 9. It gets as input the pointer to the node that needs to be replaced, its freeze state and the separation key. Note that the freeze state was already determined by the chunk mechanism and is not going to be changed anymore. In addition, we assume the existence of a global pointer to the B$^+$tree named *btree*, which can be used by all the threads and provides the access to the shared B$^+$tree structure. The global *btree* pointer is used here and also in other methods later.

The *CallforUpdate()* method actually redirects the calls according to the freeze state that it gets. Notice that for the copy case, we do not have a special method because we only need to replace the pointer to the old node with pointer to new one (using the chunk replace operation).

When a node is in an INFANT freeze state, its insertion or its sibling insertion to the B$^+$tree is not yet complete. The insertion or deletion operations cannot be performed on an infant node, and it must be helped to become NORMAL before executing. Not only operations, but also balancing activities (split, join, copy), must be held until the insertion is completed. Thus, the *helpInfant()* method is called from various methods.

Algorithm 11 The code of the split of the root in the lock-free B$^+$tree.

```
void SplitRoot (Node* root, sepKey, Node* n1, Node* n2) {
 1: Node* newRoot = Allocate();                          // Allocate new root with freeze state set to INFANT
 2: newRoot→<freezeState, joinBuddy> = <NORMAL, NULL>; newRoot→root=1;
 3: newRoot→height = root→height+1;                                          // Update new root fields
 4: // Construct new root with old root's new nodes. Son with higher keys is pointed with ∞ key entry
 5: addRootSons(newRoot, sepKey, n1, ∞, n2);
 6: CAS(&(btree→root), root, newRoot);                  // Try to replace the old root pointer with the new
 7: // If CAS is unsuccessful, then old root's new nodes were inserted under other new root,
 8: CAS(&(n1→<freezeState,joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
 9: CAS(&(n1→<freezeState,joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
10: return;
}
```

Algorithm 12 Check if the merge of the root is needed and perform if needed

```
void MergeRoot (Node* root, Node* posibleNewRoot, Node* c1, c1Key) {
 1: rootEntNum=GetEntNum(root→chunk,&firstEnt,&secondEnt);  // Count the entries in the list (do not use counter)
 2: if( rootEntNum > 2 ) { DeleteInChunk(&(root→chunk), c1Key, c1); return; }
 3: // rootEntNum is 2 here, check that first entry points to the frozen low node second on infant new possible root
 4: if((firstEnt→data == c1) && (secondEnt→data == posibleNewRoot)) {
 5:     CAS(&(posibleNewRoot→root), 0, 1);                                   // Mark as root
 6:     CAS(&(btree→root), root, posibleNewRoot);          // Try to replace the old root pointer with the new
 7:     // If CAS is unsuccessful, then old root was changed by someone else
 8: }
 9: return;
}
```

The code of *helpInfant()* is presented in Algorithm 10. First, we find the creator of the given infant node. The creator initiated the freeze due to which this infant node was inserted into the B$^+$tree. The creator's freeze state reveals what the reason was for the freeze. To finish the freeze operation we often need the separation key, which is the highest key on the node with lower keys *in creation time*. But the highest key on the node with lower keys can be different now due to concurrent completion of the freeze activity. Thus we compute the key and then we check that the node with lower keys is still an infant. If it is not, we finish by changing the higher new node's freeze state to NORMAL (if the node exists). If the node with the lower keys is still an infant, than we have found the correct separation key.

If the infant has been inserted into the B$^+$tree due to a COPY, its insertion is almost done. It is completed by changing infant's freeze state to NORMAL. This operation can be unsuccessful only if it is concurrently completed by someone else.

In the cases of SPLIT and JOIN, we simply call the relevant method. Due to possible multiple invocations, the *InsertSplitNodes()*, *InsertMergeNode()* and *InsertBorrowNodes()* methods are idempotent. Making them such allows us to help the operation in its entirety, so that by its end, the insertion of the new nodes is complete.

E Root boundary conditions

E.1 Splitting the root

The code for splitting the root in method *SplitRoot()* is provided in Algorithm 11. The *SplitRoot()* method is called by *CallForUpdate()*, Line 11, when there is a request to split a node whose root bit is set. The *SplitRoot()* method's input parameters are a pointer to the old root, pointers to the new nodes that hold the lower and higher keys of the old root, and the highest key in the new low-value node. The *SplitRoot()* method starts by allocating a new root, making it point to the new nodes N_1 and N_2 . The nodes N_1 and N_2 are created by the chunk's split of the old root. The thread that succeeds in replacing the B^+tree root pointer from the old root to a new root (allocated by this thread), inserts the actual new root. Other allocated root's candidate are freed.

The actual root split occurs in a single CAS instruction. The split of the root doesn't prevent any other threads from going through the root (to perform operations on other nodes). In Algorithm 11 we use the method *addRootSons()*, which is local and sequential. It simply installs the new entries and we do not present its code. *SplitRoot()* uses this method to install two entries to the new root's chunk. The first entry holds the highest key in N_1 as key and a pointer to N_1 as data; the second entry holds the ∞ key and a pointer to node N_2.

E.2 Root Merge

Special care is required for handling the root merge. The root needs to be merged when the number of its children is reduced from two to one, that is, when its last two children are merged. To maintain tree balance, we do not allow a root node pointing to one single descendant. The *MergeRoot()* method (presented in Algorithm 12) is called on every merge of the root's children from the *InsertMergeNode()* method. A pictorial view of the root merge can be seen in Figure 4. We have an old root R_o having two children C_1 and C_2, while node C_2 was found as having too few entries and node C_1 was chosen as C_2's slave. Node R_n is created due to C_2's chunk merge and it needs to replace C_1 and C_2. In *MergeRoot()* method, as with every merge, the parent's entry pointer pointing to C_2 is first changed to point to R_n, after which C_2's parent is checked to determine whether it is the root.

Before an entry is deleted from the root node (due to a merge of the root's children), *MergeRoot()* is invoked and checks whether the root should be merged instead of just deleting one of its entries. If so, we recheck that root merge is needed and then set the root bit on new node R_n. Finally, we try to replace the pointer to the root. The *MergeRoot()* method's first input parameter is the current root; its second input parameter is the pointer to the new node, reference to which was already inserted into the root by replacing the old son pointer. This is the possible new root. The third input parameter is a pointer to the old low-value root child, which must be removed from the parent during a regular merge. We denote it *c1*. The key is the key located on the root's entry, whose data should point to the old low-value root child (*c1*).

We check whether the root has too few children by counting all reachable, non-deleted entries on the non-frozen root (the entries are pointing to the children on the root). This is done via the *GetEntNum()* method, which goes over the list and counts non-deleted entries. In addition, *GetEntNum()* copies the first and second entries that it finds to the given entry's pointers. If *GetEntNum()* finds more then two entries, the remaining reference to the frozen low node is deleted and we return to *InsertMergeNode()*. Otherwise we verify that the last two children are indeed those we received as input and try to replace the pointer to the root if necessary.

F Minor Modifications to the Chunk Interfaces

In this section we present some minor changes to the chunk list interfaces as they appeared in [3]. These changes are necessary to our B^+tree implementation.

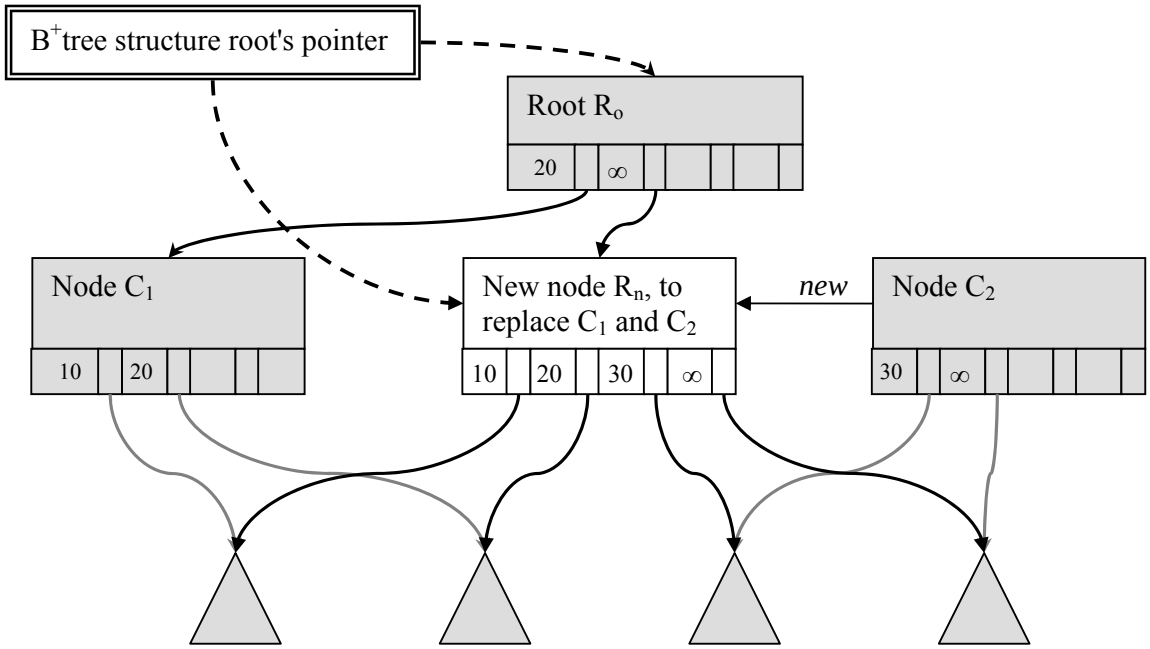

Figure 4: The diagram presenting the merging of the root. The initial B$^+$tree is marked in grey. Node R_o is an old root that initially had two children C_1 and C_2. Node C_2 is frozen as a master. Node C_1 is frozen as a slave for C_2.

F.1 The addition of replace interface to the list

Because the replace operation is not a standard interface for lists, it didn't appear in [3]. It is, however, required for the implementation of the B$^+$tree we present it here. It allows the value of data in the key-data word of an entry, to be changed (replaced), without the need to remove the entire entry from the list and insert it back again. It is used to replace the data only (i.e., the pointer to the descendant), making the entry point to another node in the B$^+$tree.

The *ReplaceInChunk()* method gets as input the key, the expected key and data (in a single word), and the new key and data values where only the data is different (also in a single word). The code appears in Algorithm 13. It starts by finding the entry e that holds the input key and then uses a CAS to atomically replace the key-data value, assuming e currently holds the expected key-data value, that is given in the input. If the *ReplaceInChunk()* method fails to find the key in the list or to find the expected data, it returns with a FALSE. If the entry is frozen, *ReplaceInChunk()* first needs to help finishing the freeze process. It is a caller's responsibility to assure the replace doesn't wrong the order of the list and to provide the expected value without a frozen bit set.

F.2 The insert and delete operations

Insert: For the insert operation we add a versioning mechanism in order to ensure that splits are executed one at a time. We avoid the ABA problem and ensure that an insert executed by the *InsertSplitNodes()* method is done exactly once. The change is relevant only to insertions of internal nodes. When an insert determines that it is working on a leaf chunk, it works as usual (as presented in [3]). The problematic case comes from the following scenario. Assume a process P is executing a split of a node A, replacing A with two new nodes B_1 and B_2 . When the split is almost done, P invokes the insertion of a new entry pointing to B_1 into A's

Algorithm 13 The replacing the key-data world value in the entry.

```
Bool ReplaceInChunk (chunk* chunk, key, exp, new) {
 1: while ( TRUE) {
 2:     if ( !Find(chunk, key) ) return FALSE;
 3:     if ( !CAS(&(cur→keyData), exp, new) ) {          // assume no freeze bit set in exp - no replace on frozen entry
 4:         if ( isFrozen(&(cur→keyData)) ) {              // if CAS failed due to freeze help in freeze and try again
 5:             chunk = Freeze(chunk, key, exp, new, REPLACE, &result);
 6:             if ( chunk == NULL ) return result;                    // Freeze completed the replace operation
 7:             continue;
 8:         }
 9:         return FALSE;                                // CAS failed due to unexpected key or data, return FALSE
10:     } // end of if CAS wasn't successful
11:     return TRUE;                                                   // CAS was successful, return TRUE
12: } // end of while
}
```

parent C. Assume P is delayed just before the insert operation and in the meantime B_1's entry is inserted by a helper and then even deleted. If P wakes up and continues with previous insert, the frozen node B_1 can be erroneously inserted again into the B$^+$tree.

In order to solve this problem and a similar delete problem we describe below, we include another field in the *nextEntry* word. The *nextEntry* is a pointer to the next entry in the chunk's list with its two LSB bits used for deleted and frozen bits. It is reasonable to assume that a chunk will not include more than 1024 entries. So we can use the entry indexes inside chunk's entries array, instead of the pointers. Thus, in the *nextEntry* word, we leave 10 bits for next entry index, 2 bits for deleted and frozen bits, and the remaining bits (52) can be used to present a version number, which will be updated each time entry's next pointer is updated.

The check is performed in *InsertEntry()* method of chunk. This methods gets the possible new entry location using *Find()* method and *prev* and *curr* global pointers as explained in Section B. After getting the possible new entry location (as part of it the version number of *prev*), we should check whether the data pointed by the new entry is still not a frozen node. Then we will be able to insert the new entry only if the version of *prev* wasn't changed. This solves the ABA problem because, after a node (referenced from the data field) is removed from the tree, it is always frozen.

Delete: The delete operation that we use in the given B$^+$tree is also slightly different from the one described in [3]. When delete concludes it is working on a leaf chunk, it works as usual (similarly to [3]). Otherwise, when delete is used on a chunk of an internal node, delete ensures that only the entry with given key *and* data is deleted. Usually a delete operation in a list is based only on a key. In our case delete should get not only the expected key, but also the expected data (i.e., a pointer to the descendant) that should be associated with this key. If the key is found, but the data is different, then the delete operation fails in the same way as when the key is not found. To avoid any kind of ABA problem and to ensure that delete (as part of a join) is applied only once, we use the versioning of the pointers. When an entry with an expected key and data is found, we also record the version of its *nextEntry* word where we are going to set the deleted bit. Next we check if data is pointing to the frozen node; if it is, we set the deleted bit if the version number is still the same.

This change is needed in order to ensure that a node is not removed twice from its parent, due to a delayed merge helper. Assume a process P is proceeding with a merge of nodes A and B. It invokes the deletion of A's entry on A's parent C. Let's assume that the key leading to node A is k. Assume P is delayed just before the delete operation, and in the meantime A's entry is deleted and key k is again inserted with an entry pointing

Algorithm 14 Freeze stabilization.

```
void StabilizeChunk(chunk* chunk) {
 1: maxKey = ∞;  Find(chunk, maxKey);                          // Implicitly remove deleted entries
 2: foreach entry e {
 3:     key = e→key; eNext = e→next;
 4:     if ( (key != ⊥) && (!isDeleted(eNext)) )               // This entry is allocated and not deleted
 5:         if ( !Find(chunk, key) ) InsertEntry(chunk, e, key);   // This key is not yet in the list
 6: } // end of foreach
 7: return;
}
```

Algorithm 15 Freezing all entries in a chunk

```
void MarkChunkFrozen(chunk* chunk) {
 1: foreach entry e {
 2:     savedWord = e→next;
 3:     while ( !isFrozen(savedWord) ) {                       // Loop till the next pointer is frozen
 4:         CAS(&(e→next), savedWord, markFrozen(savedWord));
 5:         savedWord = e→next;                                // Reread from shared memory
 6:     }
 7:     savedWord = e→keyData;
 8:     while ( !isFrozen(savedWord) ) {                       // Loop till the keyData word is frozen
 9:         CAS(&(e→keyData), savedWord, markFrozen(savedWord));
10:         savedWord = e→keyData;                            // Reread from shared memory
11:     }
12: } // end of foreach
13: return;
}
```

to some new node N. If P continues with an unchanged version of delete, node N can be erroneously deleted from the B$^+$tree. With the new delete version, node N cannot be deleted since the pointer to node N is not the expected data that should be the pointer to A.

F.3 Freeze Functionality Code

For self-containment, in this section we present the code of the freeze related functions initially presented in [3]. We avoid repeating the explanations of the code and only highlight the differences.

Initially, in [3], the *Freeze()* method for a chunk consisted of the following 5 steps: (1) change the freeze state to frozen; (2) mark all the entries as frozen and stabilize the chunk using *MarkChunkFrozen()* and *StabilizeChunk()*; and (3) decide whether the chunk should be split, joined, or copied, using the *FreezeDecision()* method. (If the chunk is to be merged, an additional step of finding the slave using the *FindJoinSlave()* method is required.) The two final steps are (4) creating new chunks according to the decision made and attaching them to the old chunk using *FreezeRecovery()*; and (5) calling the *CallForUpdate()* method to insert the new chunk(s) in place of the old one.

The *MarkChunkFrozen()*, *StabilizeChunk()* and *FreezeDecision()* methods used for the chunk lists and B$^+$tree are exactly the same as in [3]. The modifications to the *Freeze()* and *FreezeRecovery()* methods (relative to their origin in [3]) are mostly due to changes in the freeze states. For the chunk list we used fewer freeze states

Algorithm 16 Determining the freeze action.

```
recovType FreezeDecision (chunk* chunk) {
1: entry* e = chunk→head→next;    int cnt = 0;
2: while ( clearFrozen(e) != NULL) { cnt++; e = e→next; }          // Going over the chunk's list
3: if ( cnt == MIN) return MERGE; if ( cnt == MAX) return SPLIT; return COPY;
}
```

(only three), but for the B$^+$tree node we use eight. In this paper the parameters for *Freeze()* and *FreezeRecovery()* methods are nodes instead of chunks; both are very similar. Finally, in B$^+$tree we have a design point that requires master and slave parent entries to be located on the single parent node (explained in Section 4).

The *FreezeRecovery()* method presented in Algorithm 18 is similar to one presented in [3]. In [3] the *FreezeRecovery()* method has the following structure. First, the required new node or nodes are prepared (first switch statement). Then (second switch statement), the thread proceeding with the freeze recovery tries to promote its initial purpose. In other words, the thread carrying on the freeze recovery of frozen node N, does it in order to progress with the initial purpose (i.e., a delete, insert, replace, etc. on N). Finally, the threads proceeding with *FreezeRecovery()* compete to attach new node(s) to the old frozen one. The thread that succeed to attach also promotes its initial purpose.

In the B$^+$tree freeze recovery an additional check is carried between the first and second switch-statements. Recall that we keep an invariant (for simplicity and correctness), by which the parent entries that point to the master and the slave always reside on the same node. Namely, we do not allow entries that point to a master and its slave to be separated into different nodes due to a parent's split or borrow, until new nodes replace the frozen master and slave. It is this case that we test for between the first and second switch. This check is relevant only if we have two new non-leaf nodes. If the last entry on *newNode1* or the first entry on *newNode2* is pointing to a frozen node that has a merge buddy, then we move both of these entries into *newNode2*, so that they reside together. Apart from the previously mentioned alteration, the *FreezeRecovery()* method for the B$^+$tree differs slightly from the one in [3] in: updating the *creator* field, taking care to promoting the replace operation, and the way of determining which new node is going to replace the frozen one. All these changes are to ensure that the chunk will work properly as a node in the tree.

Algorithm 17 The main freeze method.

```
chunk* Freeze(Node* node, key, expected, data, triggerType tgr, Bool* res) {
 1: CAS(&(node→<freezeState,mergeBuddy>), <NORMAL, NULL>, <FREEZE, NULL>);
 2: // At this point, the freeze state is neither NORMAL nor INFANT
 3: switch ( node→<freezeState,*> ) {
 4:     case COPY: decision = COPY; break;                    // If the freeze state is already specifically set to split,
 5:     case SPLIT: decision = SPLIT; break;                  // copy or merge, only freeze recovery is needed
 6:     case MERGE: decision = MERGE; mergePartner=node→<*,mergeBuddy>; break; // mergePartner is already set
 7:     case REQUEST_SLAVE: decision = MERGE; mergePartner = FindJoinSlave(node);
 8:         if (mergePartner != NULL) break;          // If partner is NULL node was turned to SLAVE_FREEZE, continue
 9:     case SLAVE_FREEZE: decision = MERGE; mergePartner = node→<*,mergeBuddy>;
10:         // Swap between node and mergePartner, so node is always the master and mergePartner is the slave
11:         tmp = mergePartner; mergePartner = node; node = tmp;
12:         MarkChunkFrozen(mergePartner→chunk); StabilizeChunk(mergePartner→chunk);   // Verify slave is frozen
13:         CAS(&(node→<freezeState,mergeBuddy>),                 // Slave is set, verify master is in MERGE state
14:               <REQUEST_SLAVE, mergePartner>, <MERGE, mergePartner>);
15:         break;
16:     case FREEZE: MarkChunkFrozen(node→chunk); StabilizeChunk(node→chunk);
17:     decision = FreezeDecision(node→chunk);
18:         switch (decision) {
19:             case COPY: CAS(&(node→<freezeState,mergeBuddy>), <FREEZE, NULL>, <COPY, NULL>,); break;
20:             case SPLIT: CAS(&(node→<freezeState,mergeBuddy>), <FREEZE, NULL>, <SPLIT, NULL>,); break;
21:             case MERGE: mergePartner = FindJoinSlave(node);
22:                 if (mergePartner==NULL) {     // The node become slave, its merge body is master. The node need to be
23:                     mergePartner = node; node = node→<*,mergeBuddy>;     // the master and mergePartner - the slave
24:                     CAS(&(node→<freezeState,mergeBuddy>),
25:                           <REQUEST_SLAVE,mergePartner>,<MERGE,mergePartner>);
26:                 } break;
27:         } // end of switch on decision
28: } // end of switch on freeze state
29: return FreezeRecovery(node, key, expected, data, decision, mergePartner, trigger, res);
}
```

The *Freeze()* method presented in [3] works the same here, as presented in Algorithm 17. But because here we have the possibility to determine the freeze decision from the freeze state of a node, we use it for some optimization. If the reason for the freeze is already known, located in *freezeState* field, and we can figure out all needed data for the freeze recovery, we skip some steps and invoke *FreezeRecovery()* method. This is done in the switch statement. A thread that enters the FREEZE case of the switch statement behaves according to the *Freeze()* method in [3], with modification that inside the FREEZE case statement we set the freeze state according to its new possible values. The new freeze state values didn't exist in [3]. A final nuance is to check whether the *FindJoinSlave()* method returned a NULL value when the slave and master swapped their roles.

Algorithm 18 The freeze recovery.

Node* FreezeRecovery(Node* oldNode, key, expected, input, recovType, Node* mergePartner, triggerType trigger, Bool* result) {

```
1:  retNode = NULL; sepKey = ∞; newNode1 = Allocate(); newNode1→creator = oldNode; newNode2 = NULL;
2:  switch ( recovType ) {
3:      case COPY: copyToOneChunkNode(oldNode, newNode1); break;
4:      case MERGE:
5:          if ( (getEntrNum(oldNode)+getEntrNum(mergePartner))≥MAX ) {                    // Borrow, two new nodes
6:              newNode2 = Allocate(); newNode1→nextNew = newNode2;                        // Connect two new nodes together
7:              newNode2→creator = oldNode; sepKey=mergeToTwoNodes(oldNode,mergePartner,newNode1,newNode2);
8:          } else mergeToOneNode(oldNode,mergePartner,newNode1); break;                   // Merge into single new node
9:      case SPLIT:
10:         newNode2 = Allocate(); newNode1→nextNew = newNode2;                            // Connect two new nodes together
11:         newNode2→creator = oldNode; sepKey = splitIntoTwoNodes(oldNode, newNode1, newNode2); break;
12: } // end of switch
13: // If there are two new non-leaf nodes check and do not separate master and slave entries
14: if ( (newNode2 != NULL) && (newNode2→height != 0) ) {
15:     leftNode = (Node*)(getMaxEntry(newNode1))→data; leftState = leftNode→<freezeState,*>;
16:     rightNode = (Node*)newNode2→chunk→head→next→data; rightState = rightNode→<freezeState,*>;
17:     if ( (rightState == <REQUEST_SLAVE, *>) && ((leftState == NORMAL) || (leftState == INFANT) || (leftState == FREEZE) ||
18:                 (leftState == COPY) || (leftState == <SLAVE_FREEZE, rightNode>))) {
19:         moveEntryFromFirstToSecond(newNode1, newNode2); sepKey = (getMaxEntry(newNode1))→key;
20:     } else if (rightState == <MERGE, leftNode>) {
21:         moveEntryFromFirstToSecond(newNode1, newNode2); sepKey = (getMaxEntry(newNode1))→key;
22:     } else if ( (rightState == INFANT) && (leftState == <SLAVE_FREEZE, *>) ) {
23:         if (rightNode→creator == leftNode→<*,mergeBuddy>)
24:             { moveEntryFromFirstToSecond(newNode1, newNode2); sepKey = (getMaxEntry(newNode1))→key; }
25:     }
26: }
27: switch ( trigger ) {                                                                    // Perform the operation with which the freeze was initiated
28:     case DELETE:                                                                       // If key will be found, decrement counter has to succeed
29:         *result = DeleteInChunk(newNode1→chunk, key);
30:         if ( newNode2 != NULL ) *result = *result || DeleteInChunk(newNode2→chunk, key); break;
31:     case INSERT:                                                                       // input should be interpreted as data to insert with the key
32:         if ( (newNode2 != NULL) && (key<sepKey) ) *result = InsertToChunk(newNode2→chunk, key, input);
33:         else *result = InsertToChunk(newNode1→chunk, key, input); break;
34:     case REPLACE:
35:         if ((newNode2!=NULL)&&(key<sepKey)) *result=ReplaceInChunk(newNode2→chunk,key,expected,input);
36:         else *result = ReplaceInChunk(newNode1→chunk, key, expected, input); break;
37:     case ENSLAVE:                          // input should be interpreted as pointer to master trying to enslave, only in case of COPY
38:         if ( recovType == COPY ) newNode1→<freezeState,mergeBuddy> = <SLAVE_FREEZE, (Node*)input>;
39: } // end of switch
40: // Try to create a link to the first new node in the old node.
41: if ( !CAS(&(oldNode→new), NULL, newNode1) ) {
42:     // Determine in which of the new nodes the destination key is now located.
43:     if ( key<=sepKey ) retNode=oldNode→new; else retNode=oldNode→new→nextNew;
44: }
45: if (newNode1→<freezeState,*> == <SLAVE_FREEZE, *>) {
46:     m = newNode1→<*,mergeBuddy>;                                                       // If the new chunk is enslaved, correct its master to point on it
47:     CAS(&(m→<freezeState,mergeBuddy>),<REQUEST_SLAVE, oldNode>,<REQUEST_SLAVE, newNode1>);
48: }
49: CallForUpdate(recovType, oldNode, key);
50: return retNode;
}
```

Notation	Meaning
$N \Longrightarrow N_1 \Longrightarrow N_2$ or $N \Longrightarrow N_1$	N_1 and N_2 (or just N_1) are new nodes that are going to replace frozen node N in the B$^+$tree. N_1 is attached to N via *new* field and N_2 (if it exists) is attached to N_1 via *nextNew* field. If N_2 exists, then it holds higher-value keys from N and N_1 holds lower-value keys from N. N is the *creator* of N_1 and N_2, and they are *replacers* of N.
$N_1 \Longleftrightarrow N_2$	N_1 and N_2 are neighbors and are frozen as slave and master. N_1's state is <SLAVE_FREEZE, N_2> and N_2's state is <MERGE, N_1> or vice versa. N_1 has lower-value keys and N_2 has higher-value keys.
$P\langle C_1 \dashrightarrow C_2 \rangle$	The reference of the parent node P to the child node C_1 changed to point to C_2; the key remains the same. This is the effect of the successful *ReplaceInChunk()* invocation.
$N^{<x>}$	The node N is in freeze state x. It can be combined with denoting new node attachments such as $N^{<SPLIT>} \Longrightarrow N_1^{<INFANT>} \Longrightarrow N_2^{<INFANT>}$.
$< k : N >$	The entry key-data word that includes k as key and pointer to a node N as data.
k^N	Denotes the maximal key among N's entries keys.

Figure 5: The notations used in the correctness proofs.

G Correctness and Lock-Freedom

In this section we provide a full proof that B$^+$tree rebalancing operations preserve correctness, linearizability, and the lock-freedom property. We assume that the chunk mechanism of [3] is correct and lock-free. In particular, insertions, deletions, and searches of keys in the chunk's list are executed correctly and in a single computation step (which can be viewed as their linearization point). See [14] for a framework for composing lock-free algorithms.

First, we present the notations later used in the correctness proofs. For brevity, we usually say *state* instead of freeze state. The *parent* of node N is the node currently in the tree whose entry's data is pointing to N. For node N that appeared in the tree but was already disconnected from it, we define the *last parent* to be the parent it had just before the disconnection.

We also use terminus *N's parent-entry*, for a parent's entry whose data is a pointer to the node N. *N's parent-entry key* is a key located in N's parent's entry. N's parent-entry is defined for the period when node N is located in the tree and has a parent. The parent-entry for a root is also undefined. The remaining notations are presented the Figure 5.

In the following proofs we may refer to the specific invocations of some methods with specific parameters. Then we denote the input parameters to the methods as follows. We write *FindParent*(k,N,E_N,E_{nb}), where k is the search key, and N is the pointer to the child node, whose parent we are looking for on the path to the search key. E_N and E_{nb} are the pointers to the copies of the parent entries, pointing to N and N's neighbor respectively. When the values of some parameters are irrelevant to the proof, we denote this by *, e.g., *FindParent*(k,N,*,*). We write *InsertToChunk*(P,k,N), where P is the pointer to the node (usually a parent node) to whose chunk we want to insert an additional entry, having key k and data N. Data N is usually a pointer to a new child node N. Similarly, in *ReplaceInChunk*(P,k,$< k : N_o >$,$< k : N >$), P is the pointer

to the node (usually a parent node) in whose chunk we want to replace the data of an entry with key k from an old pointer to N_o to a new pointer to N. Last chunk interface is *DeleteInChunk(P, x, N)* , where P is the pointer to the node (usually a parent node) in whose chunk we want to delete the entry having key k and data N. Further, *CallForUpdate(rt, N, k)* denotes invocation of *CallForUpdate()* method on the frozen node N, in order replace node N with new nodes. The method is called after node N was split, merged or copied, which is noted by the recovery type rt. The key k is the highest key in the new node with lower keys; it is relevant only when two new nodes are going to replace N. Finally, *FreezeRecovery(N,*,...)* and *Freeze(N,*,...)* represent invocations of the *FreezeRecovery()* and *Freeze()* methods on the frozen node N, which must recover from the freeze. The remaining parameters for those two methods are not relevant to the discussion.

G.1 Node state transitions

Here we want to prove that the diagram of the node state transitions presented in Figure 2 is full and correct.

Observation 1.1: All CAS instructions that affect a node's state appear in the code presented in this paper.

Observation 1.2: The nodes are never reused unless they become unreachable from all threads. Every new node address is unique.

Observation 1.3: From the moment the node is frozen, it never changes and the decision about freeze recovery type (whether the node should be split, copied or merged) is constant and will be the same for any deciding thread. (Comes from the implementation of a chunk in [3].) Therefore, for any frozen node N, the method *CallForUpdate(rt, N,*)* is invoked with the same rt by any thread.

Observation 1.4: As stated in Section 4, a new node is always created in the INFANT state.

Definition 1.1: Let's denote by t^N_{update} the time when the method *CallForUpdate(*, N,*)* was first invoked among all the treads executing N's freeze recovery (Line 49 of *FreezeRecovery()*).

Definition 1.2: Let's denote by t^N_{attach} the time of attachment of the new node or nodes to the frozen node N. This happens on successful CAS at Line 41 of *FreezeRecovery()*.

Lemma 1.1: At t^N_{update} $N^{<x>} \Longrightarrow N_1^{<INFANT>} \Longrightarrow N_2^{<INFANT>}$ or $N^{<x>} \Longrightarrow N_1^{<INFANT>}$ or $N^{<x>} \Longrightarrow N_1^{<SLAVE\_FREEZE,M>}$. In addition from t^N_{update}, x is constant and can only be SPLIT, COPY or MERGE state.

Proof: New nodes are allocated only in order to replace an old frozen node and this is done only in the *FreezeRecovery()* method. The new nodes are in the INFANT state (Observation 1.4). The new node's state in the *FreezeRecovery()* method can only be changed in Line 38 to <SLAVE_FREEZE, *master*>. This happens when the recovery type is COPY and in this case only one new node is allocated according to the *FreezeRecovery()* method's code. Therefore, by time t^N_{update}, when the *CallForUpdate(*, N,*)* method is first invoked, any single new node (attached to N) can only be in the <SLAVE_FREEZE, *master*> or INFANT state. If there are two new nodes, they can be only in the INFANT state.

As for N's state, *FreezeRecovery(N,*,...)* is invoked only from *Freeze(N,*,...)*. According to the *FreezeRecovery(N,*,...)* method's code, new nodes can be attached only to N. In *Freeze(N,*,...)* code, there are several possibilities for changing N's state. (1) N's state is already COPY, SPLIT or MERGE (Lines 4, 5 or 6), just before *FreezeRecovery(N,*,...)* is called. (2) N's state is set to MERGE as a side effect of the *FindJoinSlave(N)* method (Lines 7 or 21), unless there was a swap in the master-slave roles, in which case N's state is set to MERGE in Lines 14 or 25. (3) The *Freeze(M,*,...)* input node was a slave with master N and N's state is set to

MERGE in Line 14, in which case *FreezeRecovery(N,*,...)* is invoked on N and not on M. (5) N's state is set to COPY or SPLIT directly in Lines 19 or 20 (*Freeze(N,*,...)*). ■

Lemma 1.2: N's *new* pointer (and N_1's *nextNew* pointer if relevant) is set once at t^N_{attach} and never changes after. $N \Longrightarrow N_1 \Longrightarrow N_2$ or $N \Longrightarrow N_1$.

Proof: The new node (or nodes) is attached to an old, frozen node N via N's *new* field. The only place where *new* is updated for any node, is in Line 41 of the *FreezeRecovery()* method. This CAS succeeds in changing the *new* field only once, because the *new* field initiated to NULL and can be updated from NULL only once.

If a connection to the second new node, N_2, is needed, it is done via N_1's *nextNew* pointer, which is only set before new nodes are attached to the old one. Then there is no concurrency and the pointer is set only once. There is no place in the code where N_1's *nextNew* pointer is changed. ■

Lemma 1.3: The initial state of a node in the B$^+$tree can be only: INFANT, NORMAL or <SLAVE_FREEZE, *master*> (as shown in Figure 2).

Proof: A node appears in the tree only as an initial root or as a result of recovering from the freeze of another node (that can be root or non-root). The initial root is created without concurrency and its state is set to NORMAL. From Observation 1.4, any node is created in the INFANT state.

After recovering from the freeze of an old root, a pointer to the new root is set only by following successful CAS instructions (every CAS instruction succeeds only once due to the uniqueness of the old root address):

1. Root exchange on split of the old root ($R_o^{<SPLIT>} \Longrightarrow N_1 \Longrightarrow N_2$) happens on success of the CAS instruction in Line 6 of the *SplitRoot()* method. The new root's state is set to NORMAL just before, in Line 2. So the new root appears in the tree in the NORMAL state. New root children, N_1 and N_2, are attached to the new root and they appear in the tree via the same CAS instruction. They are input parameters from the *CallForUpdate(*, R_o, *)* method, which is the only place from where *SplitRoot(R_o, *, N_1, N_2)* is invoked. According to Lemmas 1.1 and 1.2, N_1's and N_2's states initially are INFANT. All threads concurrently proceeding with the *SplitRoot(R_o, *, N_1, N_2)* method code perform Line 6 (inserting into the tree) before changing the new nodes' states. Thus, on successful CAS in Line 6 of the *SplitRoot()* method, N_1 and N_2 are also the in INFANT state, and so they appear in the tree.

2. Root exchange on copy of the old root ($R_o^{<COPY>} \Longrightarrow R_n$) happens on success of the second CAS instruction in Line 5 of the *CallForUpdate()* method. A new root R_n is the input parameter of *CallForUpdate(*, R_o, *)*, as R_n is attached to R_o via *new* field and by Lemma 1.2 this attachment never changes. According to Lemma 1.1, when the *CallForUpdate()* method is first invoked, R_n's state is INFANT or <SLAVE_FREEZE,M>. R_n's state is changed, in Line 8 of the *CallForUpdate()* method, only after insertion (Line 5). Thus, upon success of the second CAS instruction in Line 5 of the *CallForUpdate()*, the new root state is INFANT or <SLAVE_FREEZE,M>.

3. Root exchange on merge of the old root R_o happens on on success of the CAS instruction in Line 6 of the *MergeRoot()* method. In this case R_o is not frozen, but its last children, C_1 and C_2, are frozen, and $C_1 \Longleftrightarrow C_2$. Let's assume without loss of generality that C_1 is master and $C_1^{<MERGE,C_2>} \Longrightarrow R_n$. By Line 6 of the *MergeRoot(R_o, R_n, C_1, *)* method, $R_o\langle C_2 \dashrightarrow R_n\rangle$; otherwise we will not reach this line. So the new root R_n is already in the tree by the time the root exchange takes place. R_n is inserted into the tree before, by invoking *ReplaceInChunk()* in Line 9 of the *InsertMergeNode()* method, which is the only place from where *MergeRoot()* is invoked. The correctness of R_n's state (and the states of all the nodes inserted as a result of merge) is proven below.

Above we covered the cases when an old node is a root or a new one is going to be a root. Otherwise, a new node N_1 (or nodes N_1,N_2) can only be inserted as part of the freeze recovery from some old node N_o, $N_o \Longrightarrow N_1$ (or $N_o \Longrightarrow N_1 \Longrightarrow N_2$). According to Lemma 1.1, N_1's and N_2's states are initially set to INFANT or <SLAVE_FREEZE, M>. From the analysis of the code, we see that the <SLAVE_FREEZE,M> state never changes. Thus, if a node's state is initially set to <SLAVE_FREEZE, M>, this node is also inserted into the tree in this state. The INFANT state is changed to NORMAL only after the *InsertToChunk()* or *ReplaceInChunk()* methods were invoked. From Lemmas 3.2, 3.3, 3.6, 3.7 and 3.8, it follows that only the first invocation of *InsertToChunk(P,*,N$_1$)*, or *ReplaceInChunk(P,*,$< k : N_o >$,$< k : N_2 >$)*, or *ReplaceInChunk(P,*,$< k : N_o >$,$< k : N_1 >$)* methods inserts N_1 or N_2 into the tree successfully. The first invocation of the *InsertToChunk()* or *ReplaceInChunk()* methods always takes place before the first state change, and thus the new node is inserted when it is still in the INFANT state. ■

Lemma 1.4: There is no node state transition in the code that does not appear in Figure 2, and there is a node state transition for every arrow in Figure 2.

Proof: From analyzing the code and looking at the all CAS instructions that change the <*freezeState*, *mergeBuddy*> field on any node, we see that Lemma 1.4 is correct. ■

Theorem 1: The diagram of the node's state transitions presented in Figure 2 is full and correct.

Proof: Comes directly from Lemmas 1.3 and 1.4 ■

G.2 Supporting Lemmas

Here we provide some lemmas used for the later proofs.

Observation 2.1: A node in our B$^+$tree has at most d entries; d is assumed to be even for easier calculations. From a full node we should be able to create two nodes legal in their number of entries. Those nodes should still remain legal when (1) one entry is removed from them as part of helping delete operation during freeze recovery. In addition, (2) an entry can be removed from a newly created node when it is moved from one sibling to the other in order to keep master and slave together. Finally, (3) number of entries in the new node should allow one deletion to be done successfully on such a new node, in order to support lock-freedom. Therefore, the lower bound for the number of entries in the node should be $d/2 - 3$ nodes. Thus d has to be at least 8.

Lemma 2.1 : From the moment a new non-root node N is inserted into the tree, N's parent-entry key never changes.

Proof: Once inserted to the tree successfully, the node is constantly pointed to by its parent, always with the same key. This continues until the node is removed from the tree by replacing it with another node having the same parent-entry key (*ReplaceInChunk()*) or by deleting the entire entry from the parent (*DeleteInChunk()*). *ReplaceInChunk()* and *DeleteInChunk()* are the only methods that can change an existing entry. The parent key can never be changed because we do not have an interface in the code for changing the key of an entry. The *ReplaceInChunk()* method is used only for changing the data of an entry that is the reference to the new node. Once a node is removed from the tree, it is never pointed to again by that tree, unless reclaimed (Observation 1.2). ■

Lemma 2.2: A non-root node N is removed from the tree only after being frozen in the SPLIT, COPY or MERGE state.

Proof: A node N can be removed from a tree only when found sparse or full by some thread, and this after

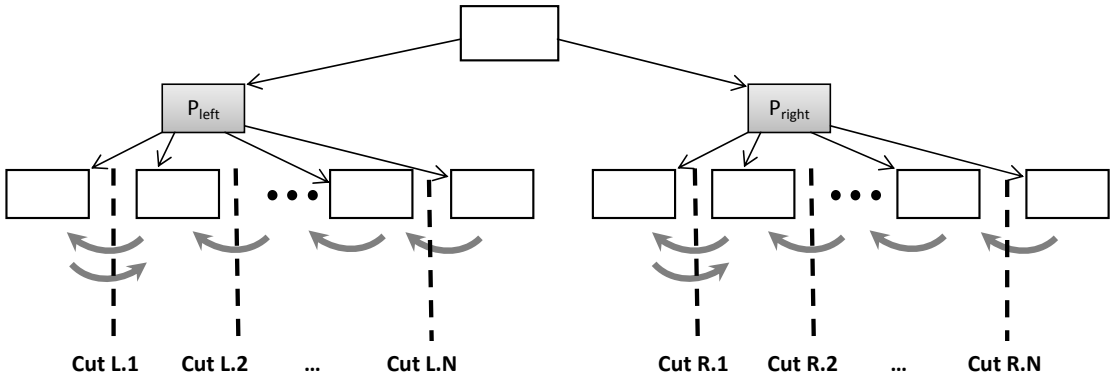

Figure 6: The separation cuts, that can be created when sons will be re-distributed upon old parents ($P_{left}\&P_{right}$) split or borrow. Grey nodes are frozen. Grey bold arrows show the possible ways how master can choose the slave that is always to the left unless master is the left-most son of its parent.

t^N_{update}, when *CallForUpdate(*,N,*)* was called. A node N can be removed by a successful CAS instruction in Line 5 of the *CallForUpdate()* method or via the *InsertSplitNodes()*, *InsertBorrowNodes()*, *InsertMergeNode()*, or *ReplaceInChunk()* methods, called directly from *CallForUpdate()*. The reference to N is supplied to those methods from *CallForUpdate(*,N,*)* as an input parameter. This parameter refers to an old node whose split, borrow, or merge need to be completed. According to Lemma 1.1, N's state is SPLIT, COPY or MERGE.

A node can also be removed from the tree, via the *helpInfant(N)* method, by calling *InsertSplitNodes()*, *InsertBorrowNodes()*, or *InsertMergeNode()* on N's creator, which we will denote C. An infant node N can be found in the tree only after t^C_{update}, when *CallForUpdate(*,C,*)* was called. This is because a new node is inserted into the tree only through the *CallForUpdate()* method or other methods that can be invoked only from *CallForUpdate()*. ■

Lemma 2.3: The separation key k, provided as input to *InsertSplitNodes(N,k)* and *InsertBorrowNodes(N,k)* methods when they are invoked from the *helpInfant(N_{infant})* method, is the same as the separation key calculated at t^N_{update} time, where N is N_{infant}'s creator.

Proof: Assume $N \Longrightarrow N_1 \Longrightarrow N_2$. The separation key k, provided as input to the *InsertSplitNodes(N,k)* and *InsertBorrowNodes(N,k)* methods (when they are invoked from *helpInfant(N_{infant})*), is the highest key on N_1. We argue that if Lines 18 or 21 (of the *helpInfant()* method) are reached (the *InsertSplitNodes(N,k)* or *InsertBorrowNodes(N,k)* methods are invoked), than node N_1 didn't change since t^N_{update} and until read at Line 4 of *helpInfant()* method. Thus the key is also the same as at t^N_{update} time.

We first need to prove that N_1's state is changed from INFANT to NORMAL always after full replacement of any frozen node. It can be seen in the code of the *InsertSplitNodes()*, *InsertMergeNode()*, *InsertBorrowNodes()*, *CallForUpdate()*, and *SplitRoot()* methods, which are the only methods to deal with frozen node replacement, that changing the new nodes' state to NORMAL is always the last operation in the code. In addition, no insertions, deletions or replacements can be done on the infant node.

In Line 4 the key is read from N_1. In the next line it is checked that N_1 is still in the INFANT state. If it is not, we do not invoke *InsertSplitNodes(N,k)* or *InsertBorrowNodes(N,k)* methods: the invocation is not needed since the node has already been fully replaced. Thus we can see that key that is used is always read from the infant node, which cannot be changed since its creation and until its state is changed ■

Definition 2.1: Two nodes N_1 and N_2 are *separated* if they were direct neighbors (as children of some parent) and they now have different new parents. This can happen as a result of a single parent split or when a parent was chosen to be a slave for its sparse neighbor, which borrows children from the slave neighbor.

Definition 2.2: Separation is done on a *cut*. A cut is a partition of the child nodes of a some frozen parent in the tree into two disjoint sets. The first set will have a different new parent than the second set.

In Figure 6 we can see an example of two parents, P_{left} and P_{right}, being frozen as master and slave, where one of them had to borrow child nodes from the other. Thus some child node will be separated according to some cut. From another point of view, we can think about two parents, P_{left} and P_{right}, as being frozen due to being full and needing to split. Thus again some child node both on P_{left} and on P_{right} will be separated. All possible cuts are presented in Figure 6. The following lemmas use the cuts' and nodes' names as presented in Figure 6.

Lemma 2.4: The separation cannot be on Cut L.1 or Cut L.2 (presented in Figure 6)

Proof: Let's assume by contradiction that separation on Cut L.1 or on Cut L.2 did happen. Separation happens only on split or borrow. Splitting P_{left} cannot cause separation on Cut L.1 or on Cut L.2, since the cuts are not medians, and the maximal number of children per node (d) has to be ≥ 8 (Observation 2.1). That is, there cannot be a node with less than 8 entries that is going to be split.

According to *FreezeRecovery()* method's code (Line 5), we decide to make a borrow and not a merge when:

(*number of sons on* P_{left}) + (*number of sons on* P_{right}) $\geq d \geq 8$,

so at least 4 child nodes have to remain on the new left node, which is going to replace P_{left}, or it is going to get nodes from P_{right}. Thus also the borrow cannot cause separation on Cut L.1 or on Cut L.2, in contradiction to the assumption. ■

Lemma 2.5: The separation cannot be on Cut R.1 and Cut R.2 (presented in Figure 6)

Proof: As in the proof of Lemma 2.4, splitting P_{right} cannot cause separation on Cut R.1 or on Cut R.2, since the cuts are not medians, and the maximal number of children per node (d) has to be ≥ 8 (Observation 2.1). In case of a merge or a borrow, one parent has exactly $d/2 - 3$ sons, without lost of generality let's assume it is P_{right}. In order to be decided about borrow, the following should hold:

(*number of children on* P_{left}) + (*number of children on* P_{right}) $\geq d$

(*number of children on* P_{left}) + $d/2 - 3 \geq d \Longrightarrow$ (*number of children on* P_{left}) $\geq d + 3 - d/2 = d/2 + 3$

(*number of children on* P_{right}) = $d/2 - 3$ and (*number of children on* P_{left}) $\geq d/2 + 3$

For two parents, we have at least d children, and each new parent will have half that amount. So each new parent has at least $d/2$ children. So at least 4 children have to be moved if their parents are separated! In cuts R.1 and R.2 we move only one or two sons from one parent to the other, thus preventing this situation. ■

Definition 2.3: Let's denote by *regular cut* the cut that is not the one of the two leftmost cuts for each parent node. In other words, cut that is not Cut L.1, Cut L.2, Cut R.1, or Cut R.2. For example Cut R.N, presented in Figure 6, is a regular cut.

We are now going to define the master-slave relationship more precisely.

Definition 2.4: The node N is defined as *master* from the time $t^N_{request}$, which is the time N's state was successfully changed from <FREEZE, NULL> to <REQUEST_SLAVE, $*$ >. This can happen only as a result of the first successful CAS in Line 8 of the *FindJoinSlave()* method. Note that N's state can later change again

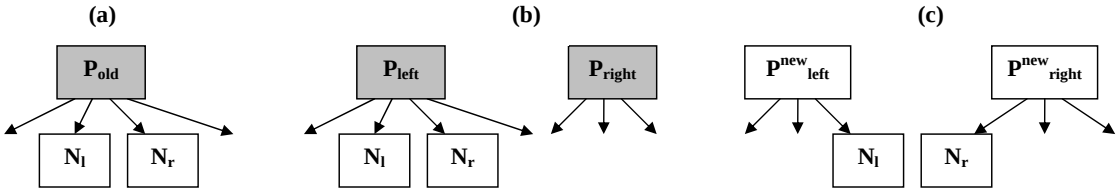

Figure 7: Part (c) shows the temporal location of children on (a) P_{old} split or (b) P_{left} and P_{right} borrow. New parents, P_{left}^{new} and P_{right}^{new}, are later going to replace the old parent (P_{old}) or old parents (P_{left} and P_{right}), who are frozen. Before P_{left}^{new} and P_{right}^{new} are attached to an old parent as its replacers, the distribution of the children among P_{left}^{new} and P_{right}^{new} can still be changed

from <REQUEST_SLAVE, X > to <REQUEST_SLAVE, Y > in same line discussed above or in Line 47 of the *FreezeRecovery()* method. But in this case the node is already a master. A master can also turn into a slave, as explained in Definition 2.5.

After a node becomes master, it needs to get a slave, for that a master first finds a slave candidate, which, if possible, becomes a true slave. The slave is always of some master.

Definition 2.5: The node M is defined as the *slave* of node N from the time $t_{enslave}^{M}$, which is the time M's state is successfully set to the <SLAVE_FREEZE, N> state or the time node M with an already-set state is inserted into the B^{+}tree. Node M can become a slave of node N can as result of successful CAS in Lines 2, or 11, or 14 of the *SetSlave()* method. Alternatively, when a slave candidate is frozen, node M becomes a slave of node N by slightly different means. The thread trying to find a slave for node N helps a previous slave candidate to recover from freeze, initially creating the new node as N's slave. After s successful CAS instruction in Line 38 of the *FreezeRecovery()* method, the enslaved node will be inserted into the tree and become a slave of node N as a result of invoking *CallForUpdate()*.

Lemma 2.6: At $t_{enslave}^{M}$ time, when M's state becomes <SLAVE_FREEZE, N>, N's state is always <REQUEST-_SLAVE, M >.

Proof: Lets look at all the possibilities of node M to become a slave, as stated in the Definition 2.5. The first possibility is that M becomes a slave after a successful CAS instruction in Line 2 of the *SetSlave(N, M, *, *)* method. This is the regular case, when slave candidate was found in a NORMAL state. The *SetSlave(N, M, *, *)* method can be only invoked only from Line 18 of the *FindJoinSlave(N)* method while N's state already is <REQUEST_SLAVE, M >, as verified in Line 12.

The second possibility is that M becomes a slave after a successful CAS instruction in Lines 11, or 14 of the *SetSlave()* method. In this case master and slave candidate are both trying to enslave each other. N's state is <REQUEST_SLAVE, M > and M's state is <REQUEST_SLAVE, N >. No matter which node's state is changed (N's state to <SLAVE_FREEZE, M > or M's state to <SLAVE_FREEZE, N >), the opposite neighbor state is already set to the correct REQUEST_SLAVE state.

Third, when M is created by enslaver thread as part of the M''s freeze recovery, M turns to be slave when it is inserted into the tree as M''s replacer. After a successful CAS instruction in Line 38 of the *FreezeRecovery()* method, M's state is set to <SLAVE_FREEZE, N>. M is inserted into the tree only as a result of invoking *CallForUpdate(*, M', *)* by any thread from Line 49 of the *FreezeRecovery()* method.

If the new node's state is SLAVE_FREEZE (and it is), the master's state is corrected to be <REQUEST_SLAVE, M >. This happens just before Line 49, in Line 47 of the *FreezeRecovery()* method. Therefore, it always

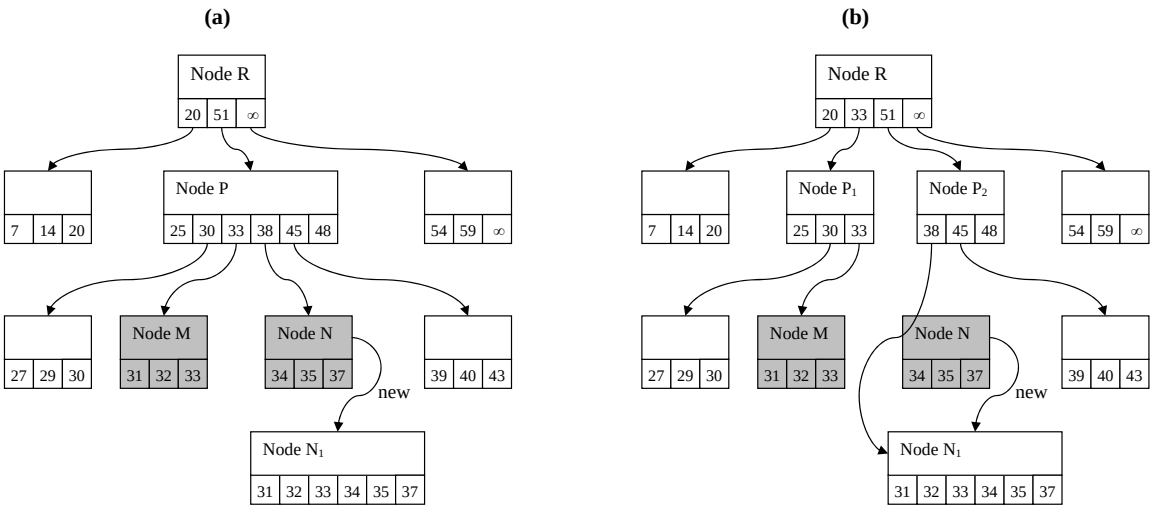

Figure 8: A possible scenario that could happen if nodes in master and slave relationship could be separated. In part (a) nodes M and N are merged into new node N_1. The merge algorithm's first step is to redirect the parent P pointer from N to N_1. If after this step M and N_1 are separated, we end up with the structure presented in part (b). Note that key 33 was taken to the node R. The merge algorithm's second step is to delete entry $< 33 : M >$ from node P_1. We then end up with the wrong B$^+$tree structure, where keys 31, 32, 33 are located in wrong sub-tree.

happens before the slave M is going to appear in the tree and thus before $t^M_{enslave}$ time. ■

Definition 2.6: The nodes N and M are defined to be in the *master-slave relationship* when N and M are direct neighbors in the B$^+$tree and (1) $N \Longleftrightarrow M$ or $M \Longleftrightarrow N$ or (2) N's state is $<$SLAVE_FREEZE, $R >$, $N \Longleftrightarrow R$, and M is the new infant node (already in the tree) that is replacing R.

Note that this holds even when master and slave are already not coupled as direct neighbors in the tree. When master and slave are merged, then either the master or the slave is disconnected from the tree, at the linearization point of *ReplaceInChunk()* method (invoked form *InsertMergeNode()*). In addition, when master and slave are borrowing entries, then an infant node is inserted into the tree and it separates master and slave. This insertion happens at the linearization point of *InsertToChunk()* method invoked form *InsertBorrowNodes()* method.

In a merge or borrow we have a parent-entry key of a node that is going to be deleted. This key is not allowed to be used in a possible concurrent split of the parent and taken to the upper levels as part of the split. This situation is presented in Figure 8. Therefore, nodes that are in a master-slave relationship always need to have the same parent or the last parent for the node that was already disconnected. When two new parent-nodes are created in order to replace some old parent or two old parents, we use the separation rules presented below, in order not to separate master and slave.

For the following lemmas and definitions, we assume the situation and node names presented in Figure 7. In this figure, node P_{old} is split, separating N_l and N_r (Figure 7(a)). Also in Figure 7(b) nodes are borrowed from P_{left} to P_{right}. In both cases two new parents P^{new}_{left} and P^{new}_{right} are created (Figure 7(c)) as replacers. In the following definition we refer to N_l and N_r as children of the new parents. Since the new parents are not yet attached to the old parent, their children (N_l and N_r) can still be moved. This is because new parents are created by each thread privately without concurrency. N_l and N_r are surely direct neighbors, because they are

read during recovery of a frozen node.

Definition 2.7: *The Separation Rule:*

1. If N_r is in state <REQUEST_SLAVE, $*$ >, then

   If N_l is in state: NORMAL, FREEZE, COPY, or <SLAVE_FREEZE, N_r > then move N_l to P_{right}^{new} and return

2. If N_r is in state $< merge, * >$, then move N_l to P_{right}^{new} and return

3. If N_r is in state INFANT and N_l is in state <SLAVE_FREEZE, N>, where N is N_r's creator, then move N_l to P_{right}^{new} and return

4. Otherwise, do nothing and return

Lemma 2.7: If the Separation Rule is followed, then no nodes in the master-slave relationship can be separated during the freeze recovery of their parent.

Proof: According to the Separation Rule, the only possible move is of N_l to P_{right}^{new}. First we want to prove that when it is decided to move N_l to P_{right}^{new} we do not separate N_l from some other master-slave relationship (not with N_r). In a regular cut, the only possible enslaving is from N_r (master) to N_l (potential slave), so the master is always from the right. In other words, in a regular cut it is impossible for a neighbor from the left of N_l to try to enslave N_l. Only N_l itself can initiate the enslaving to its left. Thus, when moving N_l to P_{right}^{new}, we do not break any master-slave relationship, if N_l is not in <REQUEST_SLAVE, N > or <MERGE, N > states, where $N \neq N_r$. If nodes in a master-slave relationship are no longer direct master and slave, then we do not break any master-slave relationship if N_l is not in the INFANT state.

In Separation Rules(1) and (3) it is clearly stated than N_l is not in <REQUEST_SLAVE, N >, <MERGE, N >, or INFANT state, for any N. In Separation Rule(2), N_l is in the correct state, because when N_r is in state <MERGE, N_l>, N_l has to be in <SLAVE_FREEZE, N_r > state. The last is justified because the change to MERGE state is done only after $t_{enslave}^{M}$, when according to Lemma 2.6 N_r is at least in <REQUEST_SLAVE, N_l>. Therefore we never separate N_l from some other master-slave relationship or spoil any other master-slave relationship establishment (not with N_r).

Now we'll look at the relationship between N_r (master) and N_l (potential or true slave). N_r as a master can only be in the <REQUEST_SLAVE, $*$ > or <MERGE, N_l> states. Another possibility is that N_r is the replacer of the original master and thus has to be in the INFANT state. Other possible N_r states are evidence of N_r not being in a master-slave relationship and are not relevant here.

If N_r is in the <REQUEST_SLAVE, $*$ > state, then it is trying to enslave N_l or a node that will replace N_l after N_l's freeze is finished. For Separation Rule(1), we prove below that if it is possible to establish a master-slave relationship between N_r and its neighbor on the left (which can be N_l or N_l's replacer), we will always move N_l to P_{right}^{new}. This is correct behavior since N_l's replacer is going to replace N_l wherever it will be.

So we assume N_r is in the <REQUEST_SLAVE, $*$ > state. Then if N_l is in NORMAL state, N_r can be successfully enslaved and N_l is not trying to enslave its left neighbor. If N_l is in the FREEZE or COPY state, N_r can be successfully enslaved (if the enslaving thread finishes the freeze recovery) and N_l is not trying to enslave its left neighbor. If N_l is in the <SLAVE_FREEZE, N_r > state, N_r can surely be successfully enslaved, and N_l is not trying to enslave its left neighbor.

Other possible N_l states are INFANT, SPLIT, <REQUEST_SLAVE, $*$ >, <MERGE, $*$ >, or <SLAVE_FREEZE, N >, where $N \neq N_r$. If N_l is in one of the states mentioned above except INFANT, the enslaving of N_r

is doomed to fail because N_l is frozen and N_l is trying to enslave its left neighbor or because N_l is going to be split. In those cases a new slave candidate will be found. If N_l's state is INFANT, it is first helped to become non-infant. But again, as in all the above cases, N_r is going to look for another potential slave, after an unsuccessful CAS attempt to change N_l's state from NORMAL to <SLAVE_FREEZE, N_r > and after helping N_l to recover from freeze or from infancy. So there is no need to move N_l to N_r in those cases. N_l cannot be in any other state not mentioned here, since we covered all the possibilities.

What if N_r is in the <MERGE, *> state? First note that if N_r is not in the <REQUEST_SLAVE, $*$ > state and we assume it is a master, it can be only in the <MERGE, N_l> state. This is so because N_r and N_l are direct neighbors, without possibility of any entry coming between them. This is so because they are read (in Lines 15 and 16 of *FreezeRecovery()*) from a frozen parent, and this parent cannot be changed (Observation 1.3). When a master-node is in the MERGE state, its slave is its direct neighbor from the left, which is N_l in our case. If so, a master-slave relationship between N_r and N_l is already established and we do not separate them due to Separation Rule(2).

Finally, when the true master has been removed from the tree, it has to be replaced by a replacer in the INFANT state. In a regular cut, the master is always removed from the tree before the slave. So if N_r, found when the parent nodes split, is in the INFANT state, it might be the master's replacer, if N_l's master is N_r's creator. If N_r is in INFANT state for some other reason, N_l cannot be found in the <SLAVE_FREEZE, N> state, where N is N_r's creator ■

Lemma 2.8: If the Separation Rule is followed, no nodes in the master-slave relationship can be separated

Proof: First, according to Lemmas 2.4 and 2.5, the cuts where master is located on the left of its slave are impossible. Thus $N_m^{<MERGE,N_s>} \Longleftrightarrow N_s^{<SLAVE\_FREEZE,N_m>}$ are never separated. From here we refer only to the $N_s^{<SLAVE\_FREEZE,N_m>} \Longleftrightarrow N_m^{<MERGE,N_s>}$ case. In other words, only regular cuts can be considered for separation.

Let's assume that at time $t_{request}^{N_m}$, N_m's state was set to <REQUEST_SLAVE, N_{ps}>, where N_{ps} is the node found as N_m's neighbor, by the *FindParent()* method invoked at Line 3 of the *FindJoinSlave()* method. N_{ps} can be N_s or its creator. Thus there was point in time, t_{init} (the *Find()*'s linearization point, explained in [3]), when N_{ps}'s and N_m's parent entries were consecutive and were located on the same parent node P.

According to the state transmission diagram in Figure 2, after obtaining the <REQUEST_SLAVE, N_{ps}> state at $t_{request}^{N_m}$, N_m remains with states: REQUEST_SLAVE, SLAVE_FREEZE, or MERGE. N_m cannot change to the SLAVE_FREEZE state in our case, since we are talking about regular cuts.

Let's also mark by time $t_{chk}^{N_m}$ the time at which the state of N_m's parent P is checked for being frozen, done in Line 14 of the *FindJoinSlave()* method. According to the code it is obvious that $t_{init} < t_{request}^{N_m} < t_{chk}^{N_m}$.

N_{ps}'s and N_m's parent entries can be separated to be located on two nodes only when P is replaced with two new nodes, upon P's split or borrow. Prior to this replacement, P must be frozen and its state must be changed from NORMAL to FREEZE or further to SPLIT, <REQUEST_SLAVE, *>, <SLAVE_FREEZE, *>, or <MERGE, *>. Either P's state is NORMAL at $t_{chk}^{N_m}$, or P's state is checked at $t_{chk}^{N_m}$ and found frozen (in any state); in the latter case we help to finish its freeze and we look for another slave. After helping an old parent recover from a freeze, N_m's (master's) state is surely <REQUEST_SLAVE, $*$ > when the parent is in the NORMAL state.

In Lemma 2.7 we showed that if N_m's state is at least <REQUEST_SLAVE, $*$ > and the Separation Rule is followed, then P's (N_m's parent's) freeze recovery causes N_m to stay on the same parent node with its potential slave, if this master-slave relationship can succeed or already did.

When parent P is frozen due to its split or borrow, P's state need to be set to SPLIT or MERGE, later P is

replaced with two new nodes. P is replaced with two new nodes P_{left}^{new} and P_{right}^{new}, which are created at Line 11 or Line 7 of the *FreezeRecovery()* method (P_{left}^{new} and P_{right}^{new} are related as on the Figure 7). In Lines 14-26 of *FreezeRecovery()* we implement the Separation Rule. There we check the states of the rightmost node pointed to from P_{left}^{new}, N_l, and the leftmost node pointed to from P_{right}^{new}, N_r, in order to check whether N_l and N_r are nodes in (potentially) master-slave relationship that could be separated. The check is done according to the Separation Rule and proved in Lemma 2.7.

We denote the time when the *mergeBuddy* fields of N_l and N_r are copied to local variables (that are later checked), in Lines 15, 16 of *FreezeRecovery(P,*,...)*, as t_{lr}^{P}. Either $t_{lr}^{P} < t_{chk}^{N_m}$ or $t_{lr}^{P} > t_{chk}^{N_m}$. In the first case, the master child will see that the parent is frozen and look for another slave. In the second case, the parent will find its child has a non-null *mergeBuddy* field and will keep master and slave on one new node. Thus, if two nodes are direct master and slave or infant and slave, their parent entries are located on the same parent node once the master's state has been set to <REQUEST_SLAVE, *>, on condition that the master's parent is not frozen. ■

G.3 Unique replacing of the frozen node

In this sub-section we would like to prove that the *InsertToChunk()*, *ReplaceInChunk()* and *DeleteInChunk()* methods used for inserting new nodes into the tree in place of the frozen ones, succeed once per new node and do not succeed when "replayed" with same parameters.

Lemma 3.1 : The *InsertToChunk(P,k,N)* method avoids the ABA problem of inserting the reference of node N to the parent P with parent-entry key k, when N was already deleted from parent P after existing in parent P with parent-entry key k.

Proof: Let's assume by contradiction that a reference to node N was inserted into the parent P with parent-entry key k, when N was already deleted from parent P after existing in parent P with parent-entry key k. According to the enhanced *InsertToChunk()* method, an entry with key k shouldn't exist in the linked list at the time when its potential location is found. Otherwise insert doesn't succeed and we assume it did. Thus, an entry pointing to N was not yet inserted or already deleted at the time new entry's, with key k, potential location is found. If an entry pointing to N was already deleted, N should be frozen (Lemma 2.2). In *InsertToChunk(P,k,N)* method, after potential location is found, it is checked that N is not frozen, if it is the new entry is not inserted. Therefore, an entry with key k and data N was not yet inserted when a frozen node appears as data to a new entry at the time its potential location is found. Same time we record the version v of the next pointer of the entry that potentially precedes the new entry. This version is located on next entry pointer and is increased any time this pointer changes. The version needs to be the same when the entry is later inserted after the frozen data check. If in the meanwhile entry with key k was inserted and deleted by some other thread, the insertion of this (already obsolete) entry is denied (because *nextEntry* pointer's version v had to be changed), in contradiction to the initial assumption. ■

In the following lemmas we are going to talk about different phases of recovery of an old frozen node referring to it as N_o for split and copy case and as N_m for merge and borrow cases. Also, for merge and borrow cases we will denote N_m's slave as N_s. The frozen node N_o or N_m has new nodes that are going to replace it. ${N_o}^{<COPY>}\Longrightarrow N_1$, or ${N_o}^{<SPLIT>}\Longrightarrow N_1\Longrightarrow N_2$, or ${N_m}^{<MERGE>}\Longrightarrow N_1$, or ${N_o}^{<MERGE>}\Longrightarrow N_1\Longrightarrow N_2$. We will always denote old frozen node's parent-entry key (N_o's or N_m's) as y. P will always mention old frozen node's (N_o's or N_m's) parent. In the following proofs, without lost of generality, we assume N_m (master node) has higher-value keys than N_s (slave node) $N_s^{<SLAVE\_FREEZE,N_m>}\Longleftrightarrow N_m^{<MERGE,N_s>}$. This is because both in *InsertMergeNode()* and *InsertBorrowNodes()* methods we work with higher-value and lower-

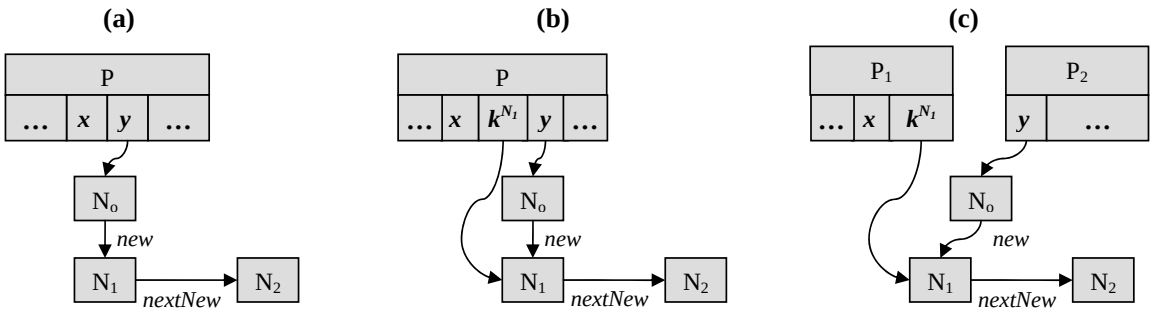

Figure 9: Insertion of the new node N_1.

value nodes and it doesn't matter who is a slave or a master. Finally, we assume correct structure of the tree, where *FindParent()* method can navigate correctly (according to Definition 4.4) through internal nodes. The correctness of the structure will be proven later in Section G.4.

Lemma 3.2: Assume $N_o \overset{<SPLIT>}{\Longrightarrow} N_1 \Longrightarrow N_2$ and *InsertToChunk(P,*k^{N_1}*,*N_1*)* is invoked as part of the recovery from node N_o's split (Line 6 of the *InsertSplitNodes()* method). Exactly one invocation of *InsertToChunk(P,*k^{N_1}*,*N_1*)* succeeds for every specific node N_1. Other invocations fail. The invocation that succeeds is of the thread that first succeeded to insert the new relevant entry to P.

Proof: Let's mark by time t_i the linearization point of successful insertion of an entry with key k^{N_1} and with data, which is a pointer to N_1, into P. The key k^{N_1} is constant per N_1 since it is provided as input to the *InsertSplitNodes(*N_o*,*k^{N_1}*)* method either from the *CallForUpdate(*,*N_o*,*k^{N_1}*)* method or from *helpInfant()* method. In first case, the k^{N_1} is captured during the time when N_1 was not yet in the tree. In second case, according to Lemma 2.3, k^{N_1} is the same as during the time when N_1 was not yet in the tree. The position before time t_i is presented in Figure 9(a). Key y is constant N_o's parent-entry key (according to Lemma 2.1). Because N_1 is a partial copy from N_o: $k^{N_1} \leq y$. In Line 5 of the *InsertSplitNodes(*N_o*,*k^{N_1}*)* method, *FindParent(*k^{N_1}*,*N_o*,*,*)* is invoked in order to find the parent of N_o. Before t_i, *FindParent(*k^{N_1}*,*N_o*,*,*)* has to find P. Because before t_i, N_o is the only node that holds key k^{N_1} in one of its entries (Definition 4.4). We are going to prove that among all the invocations of *InsertToChunk(P,*k^{N_1}*,*N_1*)* started before t_i, only one invocation succeeds, exactly at t_i.

From [3], the *InsertToChunk()* method succeeds if an entry was inserted into the list and fails if the key was already in the list. According to our additions to the original algorithm, explained in Section F.2, the *InsertToChunk()* method can also fail if the data it needs to insert, is a pointer to a frozen node. Let's look on all the invocations of *InsertToChunk(P,*k^{N_1}*,*N_1*)* started before t_i. One of them (given that there were enough progress steps) has to succeed, because originally k^{N_1} cannot be a key in the P's entries list ($k^{N_1} < y$) and because originally N_1 is not frozen and cannot freeze before being inserted into the tree. So *InsertToChunk(P,*k^{N_1}*,*N_1*)* method's invocation that caused successful insert of $< k^{N_1} : N_1 >$ the parent P has to be the first to try and succeed to insert $< k^{N_1} : N_1 >$ in. The remaining invocations of *InsertToChunk(P,*k^{N_1}*,*N_1*)* started before t_i will fail because key k^{N_1} appears in the list. Or, in the case of the ABA problem, when k^{N_1} was deleted from P and then inserted again with different data, they will fail by virtue of the fact that N_1 has to be frozen by then (proven in Lemma 3.1).

If parent P becomes frozen by the time *InsertToChunk(P,*k^{N_1}*,*N_1*)* is invoked, all the threads proceeding with *InsertToChunk(P,*k^{N_1}*,*N_1*)* will be redirected to the new N_o's parent, via P's *new* field. If P is replaced with two new nodes, P is first frozen, and threads that find P frozen act as explain above. The treads whose *FindParent()* was invoked after the freeze chose the new parent. In all those cases the same new N_o's parent

will be chosen by all threads, since they all are invoked with the same key (the redirection is explained in [3] and comes from Line 43 of *FreezeRecovery()*). The effect of *InsertToChunk()* on the new parent is the same as explained above.

Now let's see what happens after t_i. The position after t_i is presented in Figure 9(b) or (c). As we said, in Line 5 of the *InsertSplitNodes()* method, *FindParent*(k^{N_1},N_o,*,*) is invoked in order to find the parent of N_o. After t_i and before redirection to the new node N_2 is finished, key k^{N_1} is located both on N_1 and on N_o. According to Definition 4.4, because N_1's parent-entry key is k^{N_1} and it is less than N_o's parent-entry key, which is y, *FindParent()* will find N_1 and not N_o. So the result of *FindParent()* depends on whether the *Find()* method's linearization point happens before or after t_i. (The *Find()* method is invoked by *FindParent()* on P.) If before, than the discussion is the same as above. If after, then N_1 will be found on the path to the search key k^{N_1} instead of N_o. N_o will not be found in the tree by *FindParent()* and *InsertToChunk()* will not be invoked. So we see that among all the invocations of *InsertToChunk*(P,k^{N_1},N_1) started before or after t_i, only one invocation succeeds.

Notice that no entries can be added between $N_1{}^{<INFANT>}$ and $N_o{}^{<SPLIT>}$ parent-entries until split is finished, because entries can be added to a parent only on (another) split or borrow of N_1 and N_o, which cannot start since this split is not yet finished and there can not be any freeze on the infant node. But N_1's and N_o's parent-entries can be separated to be located on different nodes as the result of P's split (Figure 9(c)). In the following Lemma we will show that this causes also no problems with continuation of this split process. ■

Lemma 3.3: Assume $N_o{}^{<SPLIT>} \Longrightarrow N_1 \Longrightarrow N_2$ and *ReplaceInChunk*(P,y,$< y : N_o >$,$< y : N_2 >$) is invoked as part of the recovery from node N_o split (Line 9 of the *InsertSplitNodes()* method). Exactly one invocation of *ReplaceInChunk*(P,y,$< y : N_o >$,$< y : N_2 >$) succeeds for every specific node N_2. Other invocations fail. The invocation that succeeds is of the thread that first succeeded to replace the data of the relevant entry in P.

Proof: Let's mark by time t_r the linearization point of successful replacement of the data of an entry $< y : N_o >$ in P, when the new data is pointer to N_2. From Lemma 3.2 we know that insertion succeeds only once. Only the *InsertSplitNodes()* method is responsible for recovery of the frozen nodes in the SPLIT state, and the parent pointer is replaced after the insertion in the *InsertSplitNodes()* method's code. Therefore, the replacement can never start before a successful insertion. The final position for the insertion and thus the initial position for the replacement (before t_r) were explained in the proof of Lemma 3.2 and are presented in Figure 9(b) or (c). The keys with lower-values from N_o are duplicated on N_1. In Line 8 of the *InsertSplitNodes()* method, *FindParent*(k^{N_o}, N_o,e,*) is invoked, $k^{N_1} < k^{N_o} \leq y$. We have to look for the parent once again because parent found in previous find may have split, as shown at Figure 9(c). Every "replay" of N_o split uses the same key k^{N_o} because node N_o is frozen and unchangeable (Observation 1.3). Before t_r, *FindParent*(k^{N_o}, N_o,e,*) has to find P. This is because before t_r, N_o is the only node that holds the key k^{N_o} in one of its entries. When *FindParent()* succeeds, it also returns a copy of the parent's entry e pointing to N_o. We are going to prove that among all the invocations of *ReplaceInChunk*(P,y,$< y : N_o >$,$< y : N_2 >$) started before t_r, only one invocation succeeds exactly at t_r.

The *ReplaceInChunk*(P, y,$< y : N_o >$,$< y : N_2 >$) invocation succeeds if the entry with key y was found and the data of the entry was indeed N_o. If P is frozen the replacement is redirected to the entry on the new node that is going to replace P. Before t_r, one *ReplaceInChunk*(P,y,$< y : N_o >$,$< y : N_2 >$) invocation has to succeed, because e (with key y and data N_o) exists in P. *ReplaceInChunk*(P,y,$< y : N_o >$,$< y : N_2 >$) method's invocation that caused successful replace of the new data in the relevant entry in the parent P has to be the first to try and succeed to replace $< y : N_o >$ with $< y : N_2 >$. The remaining invocations fail on unexpected data, because after N_o is removed from the tree it never can reappear in that tree until it is

Figure 10: Merging two frozen nodes: N_s and N_m.

reclaimed and there are no threads referencing it (Observation 1.2).

Now let's see what happens after t_r. In Line 8 of the *InsertSplitNodes()* method *FindParent*(k^{N_o}, N_o,e,*) is invoked in order to find the parent of N_o. After $P\langle N_o \dashrightarrow N_2\rangle$. N_o is no longer in the tree. So the result of *FindParent()* depends on whether the *Find()* method linearization point happens before or after t_r. If before, then the discussion is the same as above. If after, than N_2 will be found on the path to the search key k^{N_o} instead of N_o; N_o will not be found in the tree by *FindParent()* and *ReplaceInChunk()* will not be invoked. So we see that among all the invocations of *ReplaceInChunk*(P,y,$< y : N_o >$,$< y : N_2 >$) started before or after t_r, only one invocation succeeds.

Lemma 3.4: Assume $N_s^{<SLAVE\_FREEZE,N_m>} \Longleftrightarrow N_m^{<MERGE,N_s>}$, $N_m \Longrightarrow N_1$ and *ReplaceInChunk*(P,y,$< y : N_m >$,$< y : N_1 >$) is invoked as part of the recovery from the nodes N_m and N_s merge (Line 9 of the *InsertMergeNode()* method). Exactly one invocation of *ReplaceInChunk*(P,y,$< y : N_m >$,$< y : N_1 >$) succeeds for every specific node N_1. Other invocations fail. The invocation that succeeds is of the thread that first succeeded to replace the data of the relevant entry in P.

Proof: Let's mark by time t_r the linearization point of successful replacement of the data of an entry $< y : N_m >$, in N_m's parent, where the new data is pointer to N_1. Position before t_r is presented in Figure 10(a) and after t_r - in Figure 10(b). Before t_r, N_s's parent-entry key is x and N_m's parent-entry key is y. The highest-value key on N_s is k^{N_s} and the highest-value key on N_m is k^{N_m}. Clear that $k^{N_s} \leq x < k^{N_m} \leq y$.

In Line 7 of the *InsertMergeNode()* method, *FindParent*(k^{N_m},N_m,e,*) is invoked. As it is proved in Lemma 2.8, master and slave can not be separated. Thus, before t_r, *FindParent*(k^{N_m},N_m,e,*) has to find P, which is both N_m's and N_s's parent. This is because before t_r, N_m is the only node that holds the key k^{N_m} in one of its entries. Every "replay" of N_m's parent find uses the same key k^{N_m} because node N_m is frozen and unchangeable (Observation 1.3). When *FindParent*(k^{N_m},N_m,e,*) succeeds, it returns a copy of the parent's entry e pointing to N_m. We are going to prove that among all the invocations of *ReplaceInChunk*(P,y,$< y : N_m >$,$< y : N_1 >$) started before t_r, only one invocation succeeds, exactly at t_r.

The *ReplaceInChunk*(P,y,$< y : N_m >$,$< y : N_1 >$) invocation succeeds if the entry with key y was found and the data of the entry was indeed N_m. If P is frozen the replacement is redirected to the entry of the new node that is going to replace P. Before t_r, one *ReplaceInChunk*(P,y,$< y : N_m >$,$< y : N_1 >$) invocation has to succeed, because e (with key y and data N_m) exists in P. *ReplaceInChunk*(P,y,$< y : N_m >$,$< y : N_1 >$) method's invocation that caused successful replace of the new data in the relevant entry in the parent P has to be the first to try and succeed to replace $< y : N_m >$ with $< y : N_1 >$. The remaining invocations fail on

unexpected data, because after N_m is removed from the tree it never can reappear that tree until it is reclaimed and there are no threads referencing it (Observation 1.2).

Now let's see what happens after t_r with *FindParent()* and *ReplaceInChunk()*. In Line 7 of the *InsertMergeNode()* method, *FindParent*(k^{N_m},N_m,e,*) is invoked in order to find the parent of N_m. After $P\langle N_m \dashrightarrow N_1\rangle$ at t_r, N_m is no longer in the tree. So the result of *FindParent()* depends on whether the *Find()* method's linearization point happens before or after t_r. If before, then the discussion is the same as above. If after, then N_1 will be found on the path to the search key k^{N_m} instead of N_m; N_m will not be found in the tree by *FindParent()* and *ReplaceInChunk()* will not be invoked.

Last possibility is that *FindParent()* of some thread happened before t_r, but additional invocation of *ReplaceInChunk()* is happening after. As explained above, this will fail on unexpected data, because after N_m is removed from the tree it never can reappear that tree. So we see that among all the invocations of *ReplaceInChunk*(P,y,$< y : N_m >$,$< y : N_1 >$) started before or after t_r, only one invocation succeeds.

Notice that no entries can be added between $N_1{}^{<INFANT>}$ and $N_s{}^{<SLAVE_FREEZE>}$ parent-entries until merge is finished, because entries can be added only on split or borrow of N_1 or N_s (which cannot start since this merge is not yet finished). In addition, N_1's and N_s's parent-entries still cannot be separated to be located on different nodes as the result of P's split (according to Lemma 2.8), because N_1 is infant and N_s is slave.. ■

Lemma 3.5: Assume $N_s^{<SLAVE\_FREEZE,N_m>} \Longleftrightarrow N_m^{<MERGE,N_s>}$, $N \Longrightarrow N_1$, and *DeleteInChunk*(P,x,N_s) is invoked as part of the recovery from nodes N_s and N_m merge (Line 14 of the *InsertMergeNode()* method). Here x is N_s parent-entry key. Exactly one invocation of *DeleteInChunk*(P,x,N_s) succeeds for every specific node N_s. Other invocations fail. The invocation that succeeds is of the thread that first succeeded to delete the relevant entry in the parent P.

Proof: Let's mark by time t_d the linearization point of successful deletion of the N_s parent-entry as part of the merge. From Lemma 3.4 we know that previous step in merge: $P\langle N_m \dashrightarrow N_1\rangle$ succeeds only once. Only the *InsertMergeNode()* method is responsible for recovery of the frozen nodes in the MERGE state, when number of the entries on N_s and N_m is less then d. From the *InsertMergeNode()* code the N_s parent-entry is deleted after the replacement. Therefore, the deletion can never start before a successful replacement. The final position for the replacement and thus the initial position for the deletion (before t_d) is presented in Figure 10(b). The keys with lower-values from N_1 are duplicated on N_s. In Line 12 of the *InsertMergeNode()* method, *FindParent*(k^{N_s}, N_s,e,*) is invoked. We have to look for the parent once again because it may be that N_m was not found in the tree, but N_s is still in the tree.

Every "replay" of N_s delete on merge uses the same key k^{N_s} for the parent finding, because node N_s is frozen and unchangeable (Observation 1.3). Recall that the keys that are lower or equal to x are duplicated on N_s and N_1. Before t_d, *FindParent*(k^{N_s}, N_s,e,*) has to find N_s's parent, let's call it P. This is because before t_d, N_s's parent-entry key x is lower than N_1's parent-entry key, which is y. N_s's parent can also be found in case N_s and N_1 were separated. In addition, when *FindParent*(k^{N_s}, N_s,e,*) succeeds, it returns a copy of the parent's entry e pointing to N_s. We are going to prove that among all the invocations of *DeleteInChunk*(P,x,N_s) started before t_d, only one invocation succeeds exactly at t_d, where x is the key of e.

The *DeleteInChunk*(P,x,N_s) invocation succeeds if the entry with key x was found and the data of the entry was indeed pointer to N_s. The simultaneity of both checks and deletion mark is achieved using the version numbers on the each entry's *nextEntry* pointers. If P is frozen the deletion is redirected on the entry of the new node that is going to replace P. Before t_d, one *DeleteInChunk*(P,x,N_s) invocation has to succeed once, because e (with key x and data N_s) exists in P. In addition e cannot be deleted for any other purpose, because for any node N, N's parent-entry is deleted only upon N's merge or borrow. N_s is currently merged and

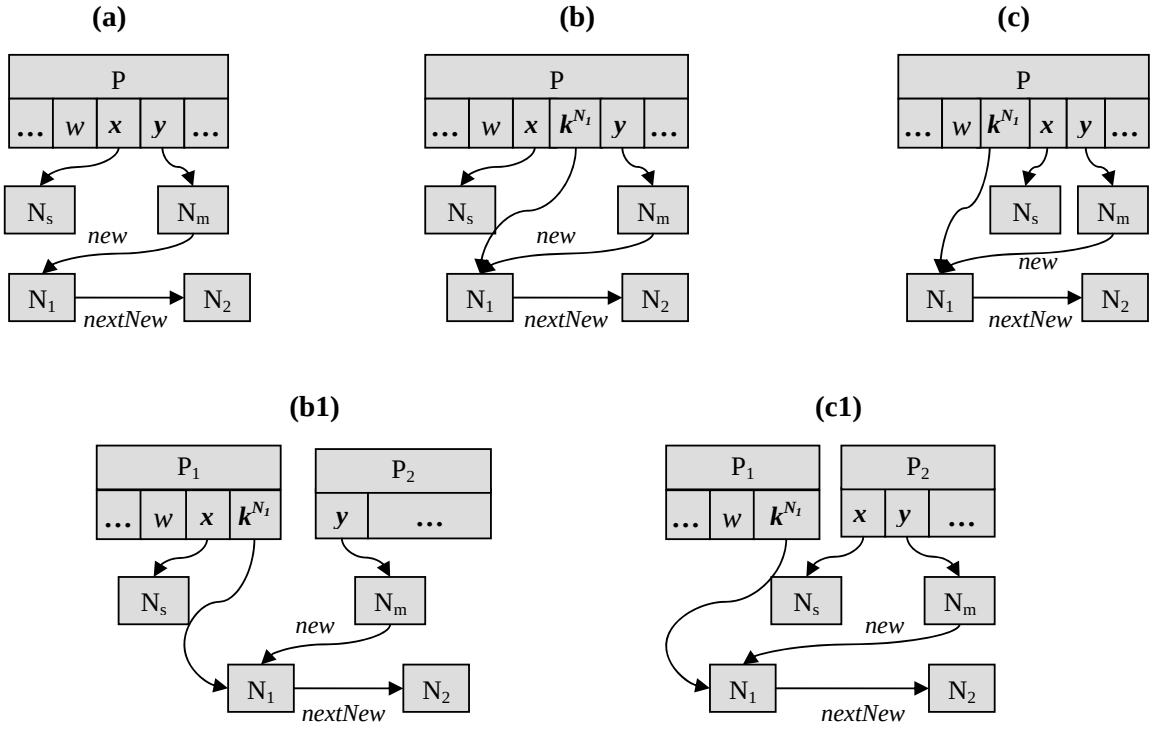

Figure 11: Borrowing from frozen N_s to frozen N_m or visa versa. This figure presents the start and the end point of the insertion of the lower-valued new node - N_1.

cannot be in (another) merge or borrow. The remaining invocations fail on unexpected data of the entry going to be deleted, because after N_s is removed from the tree it never can reappear that tree until it is reclaimed and there are no threads referencing it (Observation 1.2).

Now let's see what happens after t_d. In Line 12 of the *InsertMergeNode()* method *FindParent()* is invoked in order to find the parent of N_s. After t_d, N_s is no longer in the tree. So the result of *FindParent()* depends on whether the *Find()* method linearization point happens before or after t_d. If before, then the discussion is the same as above. If after, then N_1 will be found on the path to the search key x' instead of N_s; N_s will not be found in the tree by *FindParent()* and *DeleteInChunk()* will not be invoked. So we see that among all the invocations of *DeleteInChunk(P,x,N_s)* started before or after t_d, only one invocation succeeds. ■

Lemma 3.6: Assume $N_s^{<SLAVE\_FREEZE,N_m>} \Longleftrightarrow N_m^{<MERGE,N_s>}$, $N_m \Longrightarrow N_1 \Longrightarrow N_2$, and *InsertToChunk-(P,$k^{N_1}$,$N_1$)* is invoked as part of the recovery from nodes N_s and N_m borrow (Line 11 of the *InsertBorrowN-odes()* method). Exactly one invocation of *InsertToChunk(P,k^{N_1},N_1)* succeeds for every specific node N_1. Other invocations fail. The invocation that succeeds is of the thread that first succeeded to insert the relevant entry in the parent P.

Proof: Let's mark by time t_i the linearization point of successful insertion of an $< k^{N_1} : N_1 >$ into N_m's parent. The key k^{N_1} is constant per N_1 since it is provided as input to the *InsertBorrowNodes(N_m,k^{N_1})* method either from the *CallForUpdate(*, N_m,k^{N_1})* method, when the key is captured during the time when N_1 was not yet in the tree, or from *helpInfant()* method, when according to Lemma 2.3 key is the same as during the time when N_1 was not yet in the tree. The position before time t_i is presented in Figure 11(a) and after t_i - in Figure 11(b),(c), (b1), or (c1). Key y is constant N_m's parent-entry key (according to Lemma 2.1), and x is the N_s's parent-entry key, that directly precedes y in the parent's entries linked list.

We have two possibilities for borrowing the keys: (1) keys are borrowed from N_s to N_m and then k^{N_1} is initially located on N_s or (2) keys are borrowed from N_m to N_s and then k^{N_1} is initially located on N_m. This is verified in Lines 8 and 9 of the *InsertBorrowNodes()* method and we chose node N_{sep} to be N_s or N_m according to where k^{N_1} is initially located.

In Line 10 of the *InsertBorrowNodes(N_m,k^{N_1})* method, *FindParent(k^{N_1},N_{sep},*,*)* is invoked in order to find the parent of N_{sep}. Before t_i, *FindParent(k^{N_1},N_{sep},*,*)* has to find P as N_{sep}'s parent. Because before t_i, N_{sep} is the only node that holds key k^{N_1} in one of its entries. Also whether N_s or N_m is chosen to be N_{sep}, the same parent will be found because according to Lemma 2.8, before t_i, master and slave are located on same parent. We are going to prove that among all the invocations of *InsertToChunk(P,k^{N_1},N_1)* started before t_i, only one invocation succeeds, exactly at t_i.

First, we argue that k^{N_1} can not be a key in the P's entries linked list. In other words, if k^{N_1} is there, it means that there were no change in the N_s and N_m entries distribution. But one of them had to have minimal number of nodes - $d/2-3$, otherwise the borrow process wouldn't start. In addition N_s and N_m together have to have at list d entries. Therefore if there were no change in the N_s and N_m entries distribution we should finish with uneven entries distribution among N_s and N_m, which is impossible according to our algorithm of entries separation, where we divide all the entries exactly half half. In addition $k^{N_1} < k^{N_s ep}$, because if keys are borrowed from N_s to N_m, than at least maximal key on N_s (N_{sep}) is moved to N_2 and k^{N_1} (which have to be less) remains as highest-value key on N_1. If keys are borrowed from N_m to N_s, than the maximal key on N_m (N_{sep}) is the highest-value key on N_1 and N_2 and so also $k^{N_1} < k^{N_s ep}$.

From [3], the *InsertToChunk()* method succeeds if an entry was inserted into the list and fails if the key was already in the list. According to our additions to the original algorithm, explained in Section F.2, the *InsertToChunk()* method can also fail if the data it needs to insert, is a pointer to a frozen node. Let's look on all the invocations of *InsertToChunk(P,k^{N_1},N_1)* started before t_i. One of them (given that there were enough progress steps) has to succeed, because originally k^{N_1} cannot be a key in the P's entries list and because originally N_1 is not frozen and cannot freeze before being inserted into the tree. So *InsertToChunk(P,z,N_1)* method's invocation that caused successful insert of the new relevant entry to the parent P has to be the first to try and succeed to insert $< k^{N_1} : N_1 >$ in. The remaining invocations of *InsertToChunk(P,k^{N_1},N_1)* started before t_i will fail because key k^{N_1} appears in the list. Or, in the case of the ABA problem, when k^{N_1} was deleted from P and then inserted again with different data, they will fail by virtue of the fact that N_1 has to be frozen by then (proven in Lemma 3.1).

If parent P becomes frozen, all the threads proceeding with *InsertToChunk(P,k^{N_1},N_1)* will be redirected to the new N_m's and N_s's parent, via P's *new* field. If P is replaced with two new nodes, the same one will be chosen by all threads, since they all are invoked with the same key (the redirection is explained in [3] and comes from Line 43 of *FreezeRecovery()*). The effect of *InsertToChunk()* on the new parent is the same as explained above.

Now let's see what happens after t_i. The position after t_i is presented in Figure 11(b), (c), (b1), or (c1). As we said, in Line 10 of the *InsertBorrowNodes(N_m,k^{N_1})* method, *FindParent(k^{N_1},N_{sep},*,*)* is invoked in order to find the parent of N_{sep}. After t_i and before redirection to the new node N_2 will be finished, key k^{N_1} is located both on N_1 and on N_{sep}. But always N_1 is placed before N_{sep}, because $k^{N_1} < k^{N_s ep}$. So the result of *FindParent()* depends on whether the *Find()* method's linearization point happens before or after t_i. (The *Find()* method is invoked by *FindParent()* on P.) If before, than the discussion is the same as above. If after, then N_1 will be found on the path to the search key k^{N_1} instead of N_{sep}; N_{sep} will not be found in the tree by *FindParent()* and *InsertToChunk()* will not be invoked. So we see that among all the invocations of *InsertToChunk(P,k^{N_1},N_1)* started before or after t_i, only one invocation succeeds.

Notice that no entries can be added between $N_1^{<INFANT>}$ and $N_s^{<SLAVE_FREEZE>}$ and $N_m^{<MERGE>}$ parent-entries until borrow is finished, because entries can be added only on split or borrow of N_1, N_s or N_m (which cannot start since this borrow is not yet finished). But in some cases after the first insertion of borrow is done, N_1's, N_s's and N_m's parent-entries can be separated to be located on different nodes as the result of P's split, because master and slave are not considered in master-slave relationship. All possible separations are presented in Figure 11(b1) and (c1). Other separations are impossible because direct neighbors master and slave or because the leftmost node from the right is infant. We will show in following lemmas that this makes no problems also in further borrow algorithm's steps. ■

Lemma 3.7: Assume $N_s^{<SLAVE\_FREEZE,N_m>} \Longleftrightarrow N_m^{<MERGE,N_s>}$, $N_m \Longrightarrow N_1 \Longrightarrow N_2$ and *ReplaceInChunk-(P,y,$< y : N_m >$,$< y : N_2 >$)* is invoked as part of the recovery from the nodes N_m and N_s borrow (Line 14 of the *InsertBorrowNodes()* method). Exactly one invocation of *ReplaceInChunk(P,y,$< y : N_m >$,$< y : N_m >$)* succeeds for every specific node N_2. Other invocations fail. The invocation that succeeds is of the thread that first succeeded to replace the data of the relevant entry in the parent P.

Proof: Let's mark by time t_r the linearization point of successful replacement of the data of an entry $< y : N_m >$, in N_m's parent, where the new data is pointer to N_2. From Lemma 3.6 we know that previous insertion succeeds only once. Only the *InsertBorrowNodes()* method is responsible for recovery of the frozen master-nodes in the MERGE state, when master and slave have more than d entries together. The parent pointer is replaced after the insertion in the *InsertBorrowNodes()* code. Therefore, the replacement can never start before a successful insertion.

The final position for the insertion and thus the initial position for the replacement (before t_r) are usually as presented in Figure 11(b) or (c). But after insertion of an infant, master and infant can be separated, thus also positions as in Figure 11(b1) or (c1) can happen. The keys from N_s and (may be from) N_m are duplicated on N_1.

In Line 13 of the *InsertBorrowNodes()* method, *FindParent(k^{N_m}, N_m,e,*)* is invoked, if $k^{N_1} > x$ than $k^{N_1} < k^{N_m} \leq y$, otherwise $x < k^{N_m} \leq y$. We have to look for the parent once again because as we said it may have split. Every "replay" of N_s's and N_m's borrow uses the same key k^{N_m} because node N_m is frozen and unchangeable (Observation 1.3). Before t_r, *FindParent(k^{N_m},N_m,e,*)* has to find N_m's parent - P (even if P is already not N_s's or/and N_1 parent). This is because before t_r, N_m is the only node that holds the key k^{N_m} in one of its entries. When *FindParent(k^{N_m},N_m,e,*)* succeeds, it also returns a copy of the parent's entry e pointing to N_m. We are going to prove that among all the invocations of *ReplaceInChunk(P,y,$< y : N_m >$,$< y : N_2 >$)* started before t_r, only one invocation succeeds exactly at t_r.

The *ReplaceInChunk(P,y,$< y : N_m >$,$< y : N_2 >$)* invocation succeeds if the entry with key y was found and the data of the entry was indeed N_m. If P is frozen the replacement is redirected to the entry of the new node that is going to replace P. Before t_r, one *ReplaceInChunk(P,y,$< y : N_m >$,$< y : N_2 >$)* invocation has to succeed, because e (with key y and data N_m) exists in P. *ReplaceInChunk(P,y,$< y : N_m >$,$< y : N_2 >$)* method's invocation that caused successful replace of the new data in the relevant entry in the parent P has to be the first to try and succeed to replace $< y : N_m >$ with $< y : N_2 >$. The remaining invocations fail on unexpected data, because after N_m is removed from the tree it never can reappear that tree until it is reclaimed and there are no threads referencing it (Observation 1.2).

Now let's see what happens after t_r with *FindParent()* and *ReplaceInChunk()*. As we said, in Line 13 of the *InsertBorrowNodes()* method, *FindParent(k^{N_m}, N_m,e,*)* is invoked in order to find the parent of N_m. After $P\langle N_m \dashrightarrow N_2\rangle$ at t_r, N_m is no longer in the tree. So the result of *FindParent()* depends on whether the *Find()* method linearization point happens before or after t_r. If before, then the discussion is the same as above. If after, then N_2 will be found on the path to the search key k^{N_m} instead of N_m, N_m will not be found in the tree

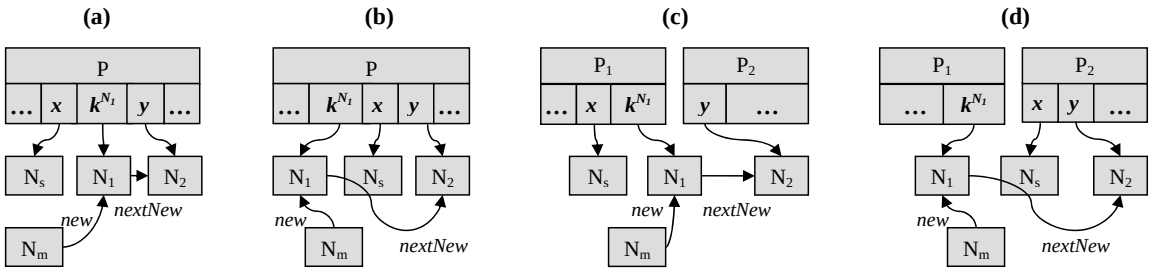

Figure 12: The intermediate position in borrowing from frozen N_s to frozen N_m or visa versa. The position presents all possible configurations when N_1 was inserted and N_2 was redirected to be pointed from the tree.

by *FindParent()* and *ReplaceInChunk()* will not be invoked.

Last possibility is that *FindParent()* of some thread happened before t_r, but additional invocation of *ReplaceInChunk()* is happening after. As explained above, this will fail on unexpected data, because after N_m is removed from the tree it never can reappear that tree. So we see that among all the invocations of *ReplaceInChunk(P,y,*$< y : N_m >$*,*$< y : N_2 >$*)* started before or after t_r, only one invocation succeeds.

Notice that still no entries can be added between $N_1{}^{<INFANT>}$, $N_s{}^{<SLAVE_FREEZE>}$ and $N_2{}^{<INFANT>}$ parent-entries until borrow is finished, because entries can be added only on split or borrow of N_2, N_1, or N_s (which cannot start since this borrow is not yet finished). But N_1's, N_2's and N_s's parent-entries can be again separated to be located on different nodes as the result of P's split or borrow. ■

Lemma 3.8: Assume $N_s^{<SLAVE\_FREEZE,N_m>} \Longleftrightarrow N_m^{<MERGE,N_s>}$, $N_m \Longrightarrow N_1 \Longrightarrow N_2$ and *DeleteInChunk(P, x,*N_s*)* is invoked as part of the recovery from nodes N_s and N_m borrow (Line 18 of the *InsertBorrowNodes()* method). Exactly one invocation of *DeleteInChunk(P,x,*N_s*)* succeeds for every specific node N_s. Other invocations fail. The invocation that succeeds is of the thread that first succeeded to delete the relevant entry in the parent P.

Proof: Let's mark by time t_d the linearization point of successful deletion of the N_s parent-entry as part of the borrow. From Lemmas 3.6 and 3.7 we know that previous steps in borrow are (1) insertion of the pointer to the N_1 and then (2) $P\langle N_m \dashrightarrow N_1 \rangle$, that both succeed in this order and only once. Only the *InsertBorrowNodes()* method is responsible for recovery of the frozen master-nodes in the MERGE state, when master and slave have more than d entries together. From the *InsertBorrowNodes()* code the N_s parent-entry is deleted after the insertion and the replacement. Therefore, the deletion can never start before a successful insertion and then replacement. The final positions for the replacement and thus the initial positions for the deletion (before t_d) are presented in Figure 12. All the keys from N_s are duplicated on N_1 and N_2, which are already in the tree.

In Line 17 of the *InsertBorrowNodes()* method, *FindParent(*k^{N_s}*,* N_s*,e,*)* is invoked. Every "replay" of N_s delete on borrow uses the same key k^{N_s} for the parent finding, because node N_s is frozen and unchangeable (Observation 1.3). Before t_d, *FindParent(*k^{N_s}*,*N_s*,e,*)* has to find N_s's parent, let's call it P. We will prove this for two cases: first if $x < k^{N_1}$ and second if $k^{N_1} < x$. In first case ($x < k^{N_1}$), before t_d, key k^{N_s} is located both on N_s and on N_1, but N_s's parent-entry key x is lower than N_1's parent entry key (k^{N_1}). So according to Definition 4.4 P will be found as N_s parent. In second case ($k^{N_1} < x$), so N_1 can not include k^{N_s} among it keys, since k^{N_1} is the maximal key on N_1 and $k^{N_1} < k^{N_s} \leq x$. So N_s is the only node that includes k^{N_1} key and according to Definition 4.4 P will be found as N_s parent. N_s's parent can be found also in case N_s and N_1 were separated, because the separations always comes after all the keys in the upper levels are corrected (will be proven later). In addition, when *FindParent(*k^{N_s}*,* N_s*,e,*)* succeeds, it returns a copy of the parent's entry e pointing to N_s. We are going to prove that among all the invocations of *DeleteInChunk(P,x,*N_s*)* started before

t_d, only one invocation succeeds exactly at t_d, where x is the key of e.

The *DeleteInChunk(P,x,N_s)* invocation succeeds if the entry with key x was found and the data of the entry was indeed pointer to N_s. The simultaneity of both checks and deletion mark is achieved using the version numbers on the each entry's *nextEntry* pointers. If P is frozen the deletion is redirected on the entry of the new node that is going to replace P. Before t_d, one *DeleteInChunk(P,x,N_s)* invocation has to succeed once, because e (with key x and data N_s) exists in P (otherwise it wouldn't be found by *Find()*). The remaining invocations fail on unexpected data of the entry going to be deleted, because after N_s is removed from the tree it never can reappear that tree until it is reclaimed and there are no threads referencing it (Observation 1.2).

Now let's see what happens after t_d. In Line 17 of the *InsertBorrowNodes()* method *FindParent()* is invoked in order to find the parent of N_s. After t_d, N_s is no longer in the tree. So the result of *FindParent()* depends on whether the *Find()* method linearization point happens before or after t_d. If before, then the discussion is the same as above. If after, then N_1 will be found on the path to the search key k^{N_s} instead of N_s, N_s will not be found in the tree by *FindParent()* and *DeleteInChunk()* will not be invoked. So we see that among all the invocations of *DeleteInChunk(P,x,N_s)* started before or after t_d, only one invocation succeeds. ■

Theorem 3: All the B$^+$tree structure changes are: (1)*InsertToChunk()*, *DeleteInChunk()*, and *ReplaceInChunk()* invocations; (2) replacing a node with its copy; (3) changing the root. All the B$^+$tree structure changes are used to replace the frozen nodes in the tree with new nodes, firstly succeed once per new node and do not change the B$^+$tree when "replayed" later with same parameters.

Proof: Regarding the first group of the structure changes, the correct usage of *InsertToChunk()*, *DeleteInChunk()*, and (*ReplaceInChunk()*) was proven in Lemmas 3.2, 3.3 (as part of their invocation in the *InsertSplitNodes()* method), in Lemmas 3.4, 3.5 (as part of their invocation in the *InsertMergeNode()* method), and in Lemmas 3.6, 3.7, 3.8 (as part of their invocation in the *InsertBorrowNodes()* method).

Second, the node is replaced with its copy, only by single CAS command in Line 5 of *CallForUpdate()* method (for root node) or by invocation of *ReplaceInChunk(P,*$< y : N_o >$*,*$< y : N_1 >$*)* in Line of *CallForUpdate()* method (for non-root node). For the root case, the CAS has to succeed once since for the very first try of this CAS the root has to be the same once read as old node. Every next CAS trial will fail on unexpected old root. For the non-root case, we are looking for the old node N_o's parent via *FindParent(x, N_o,*,*)* method's invocation, where x is a minimal key on N_o. Before linearization point of *ReplaceInChunk(P,*$< y : N_o >$*,*$< y : N_1 >$*)* invocation, the *FindParent(x, N_o,*,*)* has to find P, since N_o is still in the tree and is the only node that holds x as a key. Later invocation of *FindParent(x, N_o,*,*)* will find no parent, since N_o will be removed from the tree. In addition, first invocation of *ReplaceInChunk(P,*$< y : N_o >$*,*$< y : N_1 >$*)* succeeds since old node is still in the tree and after one such invocation succeeds all the rest will fail on unexpected data.

Finally, we are going to show that root's exchange on root's split (single CAS in Line 6 of *SplitRoot()* method) or delete (single CAS in Line 6 of *MergeRoot()* method) happens exactly once. In both cases, the CAS instruction has to succeed once since for the very first try of this CAS the root has to be the same once read as old node. Every next CAS trial will fail on unexpected old root. Full proof for uniqueness of root's merge and split is given in Lemma 4.2. ■

G.4 B$^+$tree structure correctness

In this section we would like to proof that our B$^+$tree always preserve the correct structure, so every operation on the tree can proceed correctly in $O(\log n)$ computational steps, without presence of concurrency. We assume that the B$^+$tree represents an abstract set of totally ordered keys. The only difference of our B$^+$tree

structure from the sequential B$^+$tree structure is in the fact that a key may be repeated in two consecutive nodes, in time when those two nodes are involved in rebalancing. In such cases the data of repeated keys is always the same.

Definition 4.0: A node *is reachable* if there is a path from root to this node. This path have to consists from the pointers from the non-deleted entries in the other reachable nodes. Root is reachable, by definition.

Definition 4.1: A key *is reachable*, if this key is reachable in the linked list from the *head* of a reachable node and it is not marked as deleted.

Definition 4.2: The key *is in the set* that B$^+$tree is representing, if this key is reachable and is located in the leaf node.

Definition 4.3: The B$^+$tree of order d *is correct* when it satisfies the following properties:

1. Every node has at most d entries
2. Every node (except root) has at least $d/2 - 3$ children
3. The root has at least two children if it is not a leaf node
4. All leaves appear in the same level, and carry information
5. For every entry with key k_1 and consecutive entry with key k_2 on every internal node N, all the keys that are in the set and are greater than k_1 and less or equal than k_2, are located in the k_2's sub-tree. But repetitions of those keys can also be located in the k_1's sub-tree or in the sub-tree, pointed from the further consecutive to the k_2, N's entry. This only in case there is a rebalancing operation on those keys' range.
6. The data of the repeated keys is always the same, as far as both keys are reachable from the root.

For brevity, we will denote the property number five from the Definition 4.3 as *the balance property*.

Definition 4.4: *FindParent(k,N,E_N,E_{nb})* returns pointer to N's parent P, if the following is TRUE: N is reachable and thus in linearization point of *Find()* method invoked on P, either N is the only node in the tree that holds the key k or N holds the key k and has the lower parent-entry key comparing with other node that holds the key k.

Lemma 4.1: Every node in our B$^+$tree, has at most d entries and every node (except root) has at least $d/2 - 3$ children.

Proof: Every node in our B$^+$tree is a chunk with maximal boundary set to d. In addition, the minimal boundary is set to $d/2 - 3$ in every node in our B$^+$tree (except root). The virtue of the fact that a chunk can not have more than maximal and less than minimal entries comes from the correctness of chunk's algorithm. ■.

Lemma 4.2: The root, in our B$^+$tree, has at least two children if it is not a leaf node.

Proof: The minimal boundary of the root's chunk is set to zero, therefore chunk's algorithm poses no limitations. Initial root is created as leaf node and can have from zero till d entries.

First, we prove that when a root becomes a non-leaf node, on the root-leaf split, then new root has exactly two children. The creation of the new root is done only in the *SplitRoot()* method. Old root R_o is frozen and has two new nodes to replace it. $R_o \Longrightarrow N_1 \Longrightarrow N_2$. New root R_n is created via *addRootSons()* method, which

creates R_n exactly with two sons N_1 and N_2. The new sons can not be deleted before new root is inserted to the tree. This is because a node can be deleted from the tree only as part of its merge or merge of one of its consecutive neighbors.

Second, we argue that a non-leaf root can not have less than two sons. Let's look on node R_o that has two sons C_1 and C_2 (C_1 has lower-valued keys), which are going to be merged. Without loss of generality we assume that C_2 is the master node, that was initially found sparse. Let us also divide the time of this process as following: before $t^{C_1}_{enslave}$, after $t^{C_1}_{enslave}$ and before the root pointer change (successful CAS in Line 6 of *MergeRoot()* method), and after root pointer change.

Before $t^{C_1}_{enslave}$, C_2 was found as sparse and frozen, then C_1 was enslaved and frozen. Until $t^{C_1}_{enslave}$ C_2 can not be deleted from the tree, since *MergeRoot(R_o,*,*,*)* method can not yet be invoked. Even if C_1 was also found as sparse, it still needs a partner for a merge. Therefore before $t^{C_2}_{request}$ root may have only two or more sons (in case C_1 split in the meanwhile). In order to continue with the proof, we assume C_1 wasn't split.

After $t^{C_1}_{enslave}$ and before the root pointer change, no entries can be added between C_1 and C_2. In this time, the new node (R_n), that is going to replace C_1 and C_2, is created and attached to C_2, $C_2 \Longrightarrow R_n$, then *InsertMergeNode(C_2)* is invoked. Note that node, which is created as part of the merge should have at least $d/2 - 3 + d/2 - 3 = d - 6$ entries, since we require $d \geq 8$ it should have at least two entries. As part of *InsertMergeNode(C_2)*, C_2's parent-entry is redirected to point to R_n (as presented in Figure 4). After that, it is checked whether C_1's parent is root. If C_1's parent is still R_o than it is also C_2's parent, otherwise R_o was split and C_1 and C_2 are no longer root's sons and this is not interesting. So we assume R_o was found as C_1's and C_2's parent and R_o is root. Further *MergeRoot(R_o,R_n,C_1,k)* is invoked, where k is C_1's parent-entry key. C_1 and C_2 are frozen and can not be removed by any thread that do not pass through *MergeRoot(R_o,R_n,C_1,k)*. They also can not be split. The only possibility that R_o has more than two sons is that initially there were other sons beside C_1 and C_2.

In *MergeRoot(R_o,R_n,C_1,k)* we count entries on the current root of the tree, notice root is not frozen and can concurrently change. We use special interface - *GetEntNum()*, which counts the reachable entries and returns the local copies of the first two entries that were found. If there are less than two entries, the output entries are just zeroed. Assuming we are after $t^{C_1}_{enslave}$ and before the root pointer change, the number of reachable keys on the root can be only two or more. If it is more we invoke the *DeleteInChunk()* method as in regular merge algorithm. Otherwise the two copies of the entries that we have from previous *GetEntNum()* invocation, have to point on C_1 and R_n. That is checked in Line 4 of the *MergeRoot()* method. If everything matches we set the new root's bit to 1 and switch the root. By this action also C_1 is removed from the tree.

Last, let's see what happens with the treads trying to merge root after root pointer change. All the threads that read the same information as the thread that changed the root will not be able to perform CAS, due to wrong expected data. Those are the threads that performed Line 1 of the *MergeRoot()* method, before the root change. Another thread group is those threads who performed Line 1 of the *MergeRoot()* method after the root change. In this lines we are counting the root's sons. If they got number more than two they will fail on unexpected data in the *DeleteInChunk()* method as in regular merge algorithm. If new root already got only two sons, than those threads fail to pass the condition in Line 4 of the *MergeRoot()* method. Finally if we are going to look on the old root that already got one or less sons we also will fail the conditions in Line 4. Threads that start the C_1 and C_2 merge even later then the previous group will not find C_1 in the tree and not its parent. Thus, the last group of threads will not even invoke *MergeRoot()* method. From here we see that we can never reach the situation where non-leaf root may have less then two sons. In addition the root merge is done only once per specific root and all other "replays" fail. ■

Lemma 4.3: All leaves appear in the same level, and carry information

Proof: This property follows from the sequential B$^+$tree structure which is the same in our B$^+$tree. The tree is structure can be changed only via nodes splits or merges and only the height of the root is updated. ■.

Lemma 4.4: If the balance property holds in the B$^+$tree before root's split or copy or merge, than it also holds after root's split or copy or merge. There are no repeated keys due to those actions.

Proof: First we refer to the split case. The root's split is presented in Figure **??** and its linearization point is the successful CAS in Line 6 of the *SplitRoot()* method. New root has two entries with keys x and ∞, where x is the maximal key among those located on the lower-values son and ∞ is greater than any possible key. The entries point to two lower- and high-values sons that in turn hold the old root's sons, that satisfy the balance property according to assumption of this Lemma.

The root is swapped in Line 5 of the *CallForUpdate()* method. The new root is exact copy of the previous root. Therefore if the balance property holds in the B$^+$tree before root's copy, than it also holds after root's copy.

Last, the root's merge is presented in Figure 4 and its linearization point is the successful CAS in Line 6 of the *MergeRoot()* method. In Lemma 4.2 we have seen that the root merge is done only once per specific root and all other "replays" fail. The node that becomes new root holds all the entries as they were on old root two last sons. This is so because the new root is the result of old root's two last sons merge. Therefore if the balance property holds for two last root's sons before root's merge, than it also holds for the new root after root's merge. ■

Lemma 4.5: Any node N can be simultaneously involved in only one rebalancing operation, which can be: split, copy, or merge.

Proof: According to chunk's algorithm correctness and according to the code any rebalancing operation starts by freeze. First, we need to show that freeze never starts on an infant node. Freezing of the node always starts by *Freeze()* method. Bellow we are going over all the cases where *Freeze()* method can be invoked and show that it can not be invoked on the infant node.

1. Line 15 of *FindJoinSlave()* method, when a son node in REQUEST_SLAVE state finds its parent is frozen and help parent to finish the freeze. The parent is found in Line 3 of *FindJoinSlave()* method via *FindParent()* method. The *FindParent()* method returns only parents not in infant state, this is checked in Line 9 of *FindParent()* method.

2. Line 16 of *SetSlave()* method, when a node that already get SLAVE_FREEZE state and it is going to be frozen.

3. Line 5 of *ReplaceInChunk()* method, when key-data field of an entry of the target chunk is found frozen. By going over all the places where *ReplaceInChunk()* is invoked in this code we see that it is never invoked on the reachable infant node.

4. Similarly to the previous case *Freeze()* method can be invoked form *InsertToChunk()* and *DeleteInChunk()* methods. By going over all the places where those methods are invoked in this code we see that they are never invoked on the reachable infant node.

Giving showing that freeze never starts on an infant node, please note that first instruction in *Freeze()* method is CAS that changes the target node's state to FREEZE if it was NORMAL. Therefore freeze is ensured to start on node, whose state is not INFANT and not NORMAL. Also according to *Freeze()* method's code the frozen node's entries are ensured to be marked with frozen bit and this node is ensured to be stabilized and thus to be unchangeable. Any thread that needs to proceed with any activity on the frozen node need first to finish current

freeze and thus the rebalancing operation. According to state transition diagram presented in Figure 2 (which was proven as correct in Theorem 1) it is impossible to a frozen node to turn back to be infant. ■

Lemma 4.6: If the balance property holds recursively in the B$^+$tree before insertion of the non-root node due to split, than it also holds recursively after. The data of the repeated keys, created by this action, is always the same, as far as both keys are reachable from the root.

Proof: The insertion of the non-root node due to split is presented in Figure 9. As presented there, the node N_o is split into the nodes N_1 and N_2, which hold all the entries from N_o. N_o's parent-entry key is y and the previous to y key, on the N_o's parent, is x. All the keys greater than x and less or equal than y were located on N_o before this insertion. This is correct due to this lemma assumption and due to the virtue of the fact that N_o is under rebalancing split operation, so all previous rebalancing operations are finished (Lemma 4.5). The insertion is of the node N_1 with the parent-entry key k^{N_1} which is the maximal key on N_1.

After insertion all keys greater than x and less or equal than k^{N_1} are on N_1. Those keys are also repeated on N_o, which is allowed according to balance property. In addition, all keys greater than k^{N_1} and less or equal than y are on N_o. So balance property is satisfied. The repeated keys on N_1 are copied from N_o and thus the data is the same. In addition until the finish of this split (as far as N_o is reachable) N_o is frozen and N_1 is infant and thus both are unchangeable.

The last thing that needs to be proved here, is that balance property remains correct recursively. In other words, if another split separates the infant and frozen nodes (N_1 and N_o) just after this insertions (and just inserted new key is used in the upper levels), than the balance property is not corrupted. We are arguing that if the separation, as presented in Figure 9(c), happens than the balance property holds recursively. Because the keys x, k^{N_1}, and y are ordered and the separation key (k^{N_1}) is already set into the tree. If the key k^{N_1} is now going to be taken as separation keys in more higher levels, the balance property will still be true. This is because the obsolete entries on N_o are frozen and they are exactly on the cut of new sub-trees separation created in the upper levels by k^{N_1}. Finally, all the threads searching through the tree will be redirected via the correct new path to new node N_1. ■

Lemma 4.7: If the balance property holds recursively in the B$^+$tree before replacement of the pointer to the non-root node due to split, than it also holds recursively after. The keys that were previously repeated due to this split are removed by this action.

Proof: The position before the replacement of the pointer to the non-root node due to split is presented in Figure 9(b) and (c). We replace the pointer with parent-entry key y to point on N_2 instead of N_o. After the replacement we remove the repetitions of the keys that are greater than x (previous to N_1's parent-entry key) and less or equal than k^{N_1} that were located on N_1 and N_o. N_2 includes only keys greater than k^{N_1} and less or equal than y. So after the replacement the balance property holds and there are no repetitions that were caused by previous insertion.

After the replacement action the split is actually done and we do not care about further splits of the parent and new nodes. Therefore this split can not any longer affect the structure of the tree in the upper levels (recursively). ■

Lemma 4.8: If the balance property holds recursively in the B$^+$tree before replacement of the pointer to the non-root node due to merge, than it also holds recursively after. The data of the repeated keys, created by this action, is always the same, as far as both keys are reachable from the root.

Proof: The replacement of the pointer to the non-root node due to merge is presented in Figure 10. There nodes N_s and N_m are frozen and are going to be merged. The key x is N_s's parent-entry key, y is N_m's parent-

entry key, and w is previous to N_s's parent-entry key. So before the replacement every key k on N_s satisfies $w < k \leq x$ and every key k on N_m satisfies $x < k \leq y$. This is correct due to this lemma assumption and due to the virtue of the fact that N_o is under rebalancing split operation, so all previous rebalancing operations are finished (Lemma 4.5). After the merge, new node N_1 includes all the keys from N_s and N_m and thus every key k on N_1 satisfies $w < k \leq y$.

Therefore after replacement, of the pointer of entry with key y from pointing to N_m to point on N_1, all keys greater than x and less or equal than y are on N_1. In addition the keys greater than w and less or equal than x are also located on N_1. Those keys are copied and repeated from N_s, so initial their data fields are equal. In addition, until the finish of this merge (as far as N_s is reachable) N_s is frozen and N_1 is infant and thus both are unchangeable.

As in Lemma 4.6, the last thing that needs to be proved here, is that balance property remains correct recursively. In other words, if some split separates the infant and frozen nodes (N_s and N_1) just after the redirection (and the key that is going to be deleted is used in the upper levels) the balance property is not corrupted. This is true because such separation can not happen according to the Lemma 2.8.■

Lemma 4.9: If the balance property holds recursively in the B$^+$tree before deletion of the non-root node due to merge, than it also holds recursively after. The keys that were previously repeated due to this merge are removed by this action.

Proof: The position before the deletion of the non-root node due to merge is presented in Figure 10(b). All the keys on N_s are repeated on N_1 and by deletion of N_s we remove all repeated keys from the tree. Because N_1 holds all the keys from N_s and N_m, all the keys greater than w and less or equal than y are located on N_1 and the balance property is satisfied.

After the deletion action the merge is actually done and we do not care about further splits or borrows of the parent and new nodes. Therefore this merge can not any longer affect the structure of the tree in the upper levels (recursively).■

Lemma 4.10: If the balance property holds recursively in the B$^+$tree before insertion of the non-root node due to borrow, than it also holds recursively after. The data of the repeated keys, created by this action, is always the same, as far as both keys are reachable from the root.

Proof: The insertion of the non-root node due to borrow is presented in Figure 11. As presented there, the entries of old, frozen nodes N_s and N_m are going to be redistributed among the new nodes N_1 and N_2. The key x is N_s's parent-entry key, y is N_m's parent-entry key, and w is previous to N_s's parent-entry key. So before the replacement every key k on N_s satisfies $w < k \leq x$ and every key k on N_m satisfies $x < k \leq y$. In borrow we have two cases of distribution. When k^{N_1}, that is the maximal key on N_1, was initially located on N_s and when k^{N_1} was initially located on N_m. We will look on those cases differently.

After insertion, if k^{N_1} was initially located on N_s, than $w < k^{N_1} \leq x$ and after the insertion keys greater than w and less or equal than k^{N_1} are located on N_1. In addition, keys greater than w (but also greater than k^{N_1}) and less or equal than x are located on N_s. In the second case, if k^{N_1} was initially located on N_m, than $x < k^{N_1} \leq y$ and after the insertion keys greater than w and less or equal than x are located on N_s. Copies of those keys and little more keys (up to k^{N_1}) are located on N_1. So in both cases, the balance property is satisfied. Also, in both cases, until this borrow is finished and node N_s is disconnected, N_s is frozen and N_1 is infant and thus both are unchangeable, thus the repeated keys' data is equal for both keys.

Again we need to prove, that balance property remains correct recursively. In other words, if another borrow or a split separates the infant and frozen nodes (N_s, N_m, or N_1) just after this insertions (and the key involved

in borrow operation is used), than the balance property is not corrupted. Only the separation, as presented in Figure 11(b1) and (c1) can happen. Because N_s with its parent-entry key x is as slave and according to Separation Rule it can not be separated from N_1, if N_1's parent-entry key k^{N_1} is greater than x. If the separation, as presented in Figure 11(b1) or (c1), happens, than the balance property holds recursively. Because the keys x, k^{N_1}, and y are ordered and the separation key is already set into the tree. If the key k^{N_1} is now going to be taken as separation keys in more higher levels, the balance property will still be true. This is because the obsolete entries on N_m or N_s are frozen and they are exactly on the cut of new sub-trees separation created in the upper levels by k^{N_1}. Finally, all the threads searching through the tree will be redirected via the correct new path to new node N_1. ■

Lemma 4.11: If the balance property holds recursively in the B$^+$tree before replacement of the pointer to the non-root node due to borrow, than it also holds recursively after. The data of the repeated keys, created by this action, is always the same, as far as both keys are reachable from the root.

Proof: The replacement of the pointer to the non-root node due to borrow is presented in Figure 11 (before) and in Figure 12 (after). Old nodes N_s and N_m are frozen and are redistributing their entries. New node N_1 is already in the tree and N_2 will be in the tree after this replacement. The key x is N_s's parent-entry key, y is N_m's parent-entry key, which will be redirected to N_2.

So before and after the replacement every key on N_s is less than x. Keys that are less than x also locate on N_1, but there we have also keys up to k^{N_1}, which is maximal key on N_1. This is the repetition of the keys that we have from the previous insertion and the similarity of the keys was explained in Lemma 4.10. After the replacement all the keys greater than k^{N_1} and less than y are located on N_2, that is according to the balance property. There are no repetitions of keys created by this replacement. Finally, the parent's separations possible in Lemma 4.10 can happen also here. It was explained in Lemma 4.10 that they cause no problems. Similarly to there, the separations can not cause the balance property to fail recursively. ■

Lemma 4.12: If the balance property holds recursively in the B$^+$tree before deletion of the non-root node due to borrow, than it also holds recursively after. The keys that were previously repeated due to this merge are removed by this action.

Proof: The position before the deletion of the non-root node due to borrow is presented in Figure 12. All the keys on N_s are repeated on N_1 and probably on N_2 and by N_s's deletion we remove all repeated keys from the tree. Because N_1 and N_2 hold all the keys from N_s and N_m, all the keys less or equal than k^{N_1} are located on N_1. In addition, all the keys greater than k^{N_1} and less or equal than y are located on N_2 and so balance property is satisfied.

After the deletion action the borrow is actually done and we do not care about further splits or borrows of the parent and new nodes. Therefore this merge can not any longer affect the structure of the tree in the upper levels (recursively). ■

Lemma 4.13: The balance property always holds in our B$^+$tree

Proof: First we argue that balance property holds for initial B$^+$tree structure. This is correct since initial B$^+$tree is the leaf-root, which can be empty or with ordered linked list of the entries. After, we need to prove that every operation that change the B$^+$tree structure doesn't corrupt the balance property. The operations that change the B$^+$tree structure are: (1) root's split; (2) root's copy; (3) root's merge (all three were proven in Lemma 4.4); (4) insertion of the non-root node due to split (proven in Lemma 4.6); (5) replacement of the pointer to the non-root node due to split (proven in Lemma 4.7); (6) replacement of the pointer to the non-root node due to merge (proven in Lemma 4.8); (7) deletion of the non-root node due to merge (proven in Lemma 4.9); (8) insertion of the non-root node due to borrow (proven in Lemma 4.10); (9) replacement of the pointer

to the non-root node due to borrow (proven in Lemma 4.11); (10) deletion of the non-root node due to borrow (proven in Lemma 4.12); (11) replacement of the pointer to the non-root node due to copy. Regarding the last operation, in one single atomic step we change the tree structure, so if the tree is balanced before the copy-CAS it is balanced after. ■

Lemma 4.14: In the B$^+$tree any key can be repeated only twice.

Proof: First, we proved in Lemma 4.5 that there can not be any new rebalancing operation (split, merge or borrow) on any node that is still involved in any old rebalancing operation. Having say that, we can see that a key can be repeated only as part of a single (for any specific node) rebalancing operation. The key's repetition is removed by the end of rebalancing (proven in Lemmas 4.12, 4.9 and 4.7). So any key appears once in a tree or appear twice in the tree if this key is related to the old and new node under rebalancing operation. ■

Lemma 4.15: If there exists two reachable entries with the same key in the B$^+$tree their data is same

Proof: The correctness of this lemma comes directly from Lemmas 4.14, 4.4, 4.6, 4.8, 4.10, 4.11. ■

Lemma 4.16: In the correct tree, the *FindParent(k,N,E_N,E_{nb})* method's implementation follows its definition 4.4

Proof: The *FindParent(k,N,E_N,E_{nb})* method's implementation is presented in Algorithm 2. According to correct B$^+$tree definition (Definition 4.3), on every internal node *FindParent(k,N,E_N,E_{nb})* method follows the entry's data pointer, where the entry's key is the greatest key in the linked list less or equal than k. Due to implementation of *Find()* method, when looking for such entry we are observing keys in the node's linked list in their growing order.

According to Lemma 4.15 any key that can be repeated, has exactly the same data as the other key, as far as both keys are reachable. Therefore as far as both keys are reachable we should get to the same location, following any of the repeated key. If in time when *FindParent()* method was investigating a node, this node got disconnected than the linearization point of this search is going to be set in the disconnection time and the result will be correct for this time. The correctness of this will be proved in Section TBD, talking about linearization points correctness. Therefore, if node N is reachable in the tree and it is the only node that includes key k, than one of $N's$ parents have to be found. We save "one of the parents", in order to cover the case where initial $N's$ parent was split and currently entry pointing to N is duplicated in two nodes.

In the other case, if node N is reachable in the tree, but it is not the only node that includes key k, than this key can be located on the one node before or after N. In this case N's parent will be found only if entry pointing to N will be the first in the growing order in N's parent-node linked list. In other words, N's parent will be found only in case N's parent-entry key is the lower among two nodes' parent'entry keys, so the other node appears after N. ■

Theorem 4: Lock-free B$^+$tree always preserve the correct structure, so every operation on the tree can proceed correctly in $O(\log n)$ computational steps, without presence of concurrency

Proof: According to Lemmas 4.1, 4.2, 4.3, 4.13 and 4.15 lock-free B$^+$tree satisfies the correctness definition (Definition 4.3). In addition, in the worst case it may include up to $2N$ keys where N is number of keys in the set. Therefore insertions, deletions, and searches in the tree can be done in $O(\log 2N)$ which is $O(\log N)$ computational steps, without presence of concurrency. ■

G.5 Bounded Lock-Freedom

Here we would like to prove that the B$^+$tree design retains bounded lock-freedom. In particular, we would like to prove that progress is made at least every $O(T \log n + T^2)$ computational steps, where n is the number of the keys in the set and T is number of the threads simultaneously proceeding with their operations on the B$^+$tree.

Observation 5.1: The chunk mechanism is lock-free as proven in [3]; i.e., the *InsertToChunk()*, *DeleteInChunk()*, or *ReplaceInChunk()* method's invocation restarts only if another entry in the same location was inserted or deleted. The *SearchInChunk()* method's invocation restarts only upon failure to disconnect a deleted entry from the list, which also happens only if another entry in the same location was inserted or deleted.

If chunk operation encounters a frozen chunk, the operation is delayed and will be applied on a new chunk, replacing the frozen one. Many threads may be involved in helping a node N to recover from a freeze. Here we prove that every freeze recovery results in at least one thread's progress.

Observation 5.2: There are only three reasons for a thread to start a node freeze. (1) An insert thread can not find an empty entry. (2) A delete thread is going to decrease the entry counter below the allowed minimum. (3) A node was selected as a slave for one of its neighbors. In rest of the cases, a thread does not start the freeze but helps the freeze activity initiated by some other thread, because recovery from the freeze is necessary for the progress of this thread.

Recall that every thread that helps N to recover from freeze creates a new node or nodes, which are going to replace N in the tree. The thread t has to help N in order to promote t's operation O, which is initially supposed to be applied on N. After the new node(s) are created, t applies operation O on the new node. Finally, t tries to make N point to the new node(s) (created by t) as to N's replacer(s). If the CAS instruction is successful, the thread's operation O is promoted as well.

Definition 5.1: The *freeze trigger* is the reason why a thread proceeds with the freeze of node N. A thread helps with the recovery from N's freeze ,in order to finish this thread's operation depending on N. The triggers can only be: INSERT, DELETE, REPLACE, ENSLAVE, or NONE. An insert, delete, or replace thread has the INSERT, DELETE, or REPLACE freeze trigger and will finish insertion, deleting, or replacing of an entry on the N's replacers, as part of the N's freeze recovery. An enslave thread has the ENSLAVE freeze trigger will enslave the N's replacer only if N is in COPY state. A thread having the NONE freeze trigger, just needs to finish the freeze and does not promote any operation.

Observation 5.3: According to Observation 2.1, after a split or a borrow, at least one deletion can succeed on the new half-full replacer. Two old nodes will be merged into one new node if both old nodes hold $d - 1$ entries. Therefore, at least one insert can succeed on the almost full new replacer after a merge as well.

Definition 5.2: We define the completing of the following actions as *intermediate progress*: (i) Inserting an entry into the chunk's linked list (*InsertToChunk()* method's result). (ii) Logical deletion of an entry in the chunk's linked list (marking the deleted bit). (iia) Physical deletion of an entry in the chunk's linked list (distinguished from ii). (iii) Replacing the data of an entry in the chunk's linked list (*ReplaceInChunk()* method's result). (iv) Verifying that master parent node is not frozen (progressing over the first part of the *FindJoinSlave()* method). (v) Enslaving of a node for the master node (*FindJoinSlave()* method's result).

Definition 5.3: The completion of the B$^+$tree interface (*SearchInBtree()*, *InsertToBtree()*, or *DeleteFromB-tree()*) is defined as *full progress*. According to the interface operation, we distinguish between the full progress of a search, an insert, and a delete. For a search, full progress means that it has reached the relevant leaf and determined whether the key exists in the leaf. For insert, full progress means that the key has been inserted

into the relevant leaf (reachable from the root) or that the key has been found to already exist. For delete, full progress means that the key has been deleted from the relevant leaf (reachable from the root) or that such a key has been found not to exist. Furthemore, a thread is considered to have made a full progress only when it returns to the user.

Definition 5.4: A thread has *completed the freeze recovery* of node N when it succeeds to attach a new node or nodes to N's *new* field. That means that the CAS instruction at Line 41 of *FreezeRecovery()* method has succeeded.

Definition 5.5: We denote all new[7] repeating loops that can occur in the B$^+$tree code as follows:

1. *Enslaving loop* is the repeating of the *FindJoinSlave(M)* method's code, because every chosen slave for master M appeared to be frozen in the trial to enslave it for M.
2. *Replacing loop* is the repeating of the *ReplaceInChunk()* method's code because the target chunk appeared to be frozen in the trial to replace an entry on it.

We show below that the above loops iterations number is bounded by a constant. Other new loops in the code, such as *while (node→height != 0)* in *FindParent()* and *FindLeaf()* may have only as many iterations as the height of the B$^+$tree. If the B$^+$tree grows concurrently, that can only be due to the successful insertions of new nodes. The rest of the loops in the code were already proven as lock-free in [3]. Other than the aforementioned loops, the code cannot repeat and will make progress as long as it gets computational steps.

Lemma 5.1: If a thread t, proceeding with the freeze recovery of parent node P and having the freeze trigger NONE, completes the freeze, then t makes intermediate progress by the end of P's freeze.

Proof: The freeze activity is invoked with NONE as a trigger only from Line 15 of the *FindJoinSlave()* method, presented in this paper, or from Line 7 of the *ClearEntry()* method, presented and proven in [3]. In the *FindJoinSlave(M)*, freeze is invoked after setting M's state to REQUEST_SLAVE if M's parent P was found frozen. This is done in order to ensure that P wasn't frozen before setting M's state to REQUEST_SLAVE. If it was, a parent might not be aware that its children have become master and slave and separate them. After setting M's state to REQUEST_SLAVE, it is enough to find a moment when the parent is not frozen. So either P is not frozen or its replacer node (M's new parent) is not frozen, just after P's freeze (invoked from Line 15 of the *FindJoinSlave()* method). Therefore, thread t, which invoked the freeze with NONE as trigger, from Line 15 of the *FindJoinSlave()* method, and completed this freeze, has progressed to the position from which t no longer needs to check whether the parent is frozen. This is intermediate progress (iv) from Definition 5.2. ■

Lemma 5.2: If a thread t, proceeding with the freeze recovery of node N and having the freeze trigger INSERT, DELETE, REPLACE, or NONE, completes the freeze, then t makes intermediate progress by finishing N's freeze.

Proof: If t's freeze trigger is NONE, then the intermediate progress was proven in Lemma 5.1, because trigger NONE cannot happen on a leaf node. For rest of the cases, by completing the freeze, t successfully ensured that the new node or nodes, which are going to replace N by finishing N's freeze, have the relevant operation (insert, delete, or replace) finished on them. This way intermediate progress: (i) insert; (ii) and (iia) delete; or (iii) replace is made by the end of N's freeze. ■

Lemma 5.3: A thread t that proceeds with the freeze recovery of node N and has the freeze trigger ENSLAVE, and which completes $N's$ freeze as part of establishing master-slave relationship. In such a case either t

[7]Loops are new in the sense that they do not appear in [3].

makes intermediate progress by the end of N's freeze or t (or another thread) makes intermediate progress in a constant number of computational steps after N's freeze end.

Proof: If N's state is COPY, then t, by completing the N's freeze, successfully ensures that the new node, which is going to replace N by the end of N's freeze, is properly enslaved. This ensures t's intermediate progress (v). Otherwise N's state isn't COPY. Let us examine the reasons for N's freeze, according to Observation 5.2. First, N can be full, so it will be split next. Second, it can be sparse or enslaved and in both cases N will be joined next.

If the first reason took place, N was split to N_1 and N_2. Let's assume without loss of generality that N_2 is the node suitable to be new slave candidate for thread t. We state that either t will immediately succeed to enslave N_2 or there will be some other thread's intermediate progress before that. Thread t may fail to enslave N_2 only if N_2 was already frozen for any other reason. Let's go over all three possibilities (from Observation 5.2) why the freeze of node N_2 was started.

If N_2 was full, then another thread's intermediate progress is ensured because, because at least one insert operation has to succeed on N_2 before there will be an insert thread that can't find an empty entry on N_2's chunk. if N_2 was sparse or enslaved, the cause is a delete thread that tried to decrease the N_1's or N_2's entry counter below the allowed minimum. According to Observation 2.1, it is possible to delete an entry from a node created after the split, ensuring another thread's intermediate progress. In both scenarios presented in this paragraph, the intermediate progress is made $O(1)$ computational steps after the end of the N's freeze.

A more complex scenario occurs if two or more threads are trying to simultaneously delete an entry from N_1 or N_2, thus decreasing the counter and causing the freeze without yet marking any entry as deleted. (If an entry is marked as deleted, the deletion will be finished by the freeze.) In this case N_2 is frozen and the freeze decision will be to copy a node, because actually there are more than the minimum entries on N_2. Any thread that completes the freeze recovery on N_2 will make progress. In this complex scenario, intermediate progress is ensured to be made $O(d)$ steps after the end of the N's freeze, where d is number of entries in a node and thus constant.

If the second reason took place, and N was joined with its neighbor to create a new node or nodes, we can prove that intermediate progress will soon take place using the same reasoning as for the first reason. In the first scenario, delete or insert threads have to make intermediate progress before a new slave candidate is frozen. Similarly to the first reason, in the complex scenario, the end of the freeze recovery of a new slave replacer ensures the intermediate progress of some thread.

In any case, we can see that the end of the N's freeze recovery ensures that some thread will make intermediate progress immediately or in a constant number of steps. There can not be more than a constant number threads taken before any intermediate progress. ■

Lemma 5.4: For any frozen node N, end of recovery from the freeze ensures the intermediate progress of thread t, which completed N's freeze. If the intermediate progress does not happen immediately with the end of recovery from the N's freeze, then the intermediate progress is made in a constant number of steps after N has recovered from the freeze.

Proof: If t's freeze trigger is INSERT, DELETE, REPLACE, or NONE, then according to Lemma 5.2, t makes intermediate progress by finishing N's freeze. If t's freeze trigger is ENSLAVE, then according to Lemma 5.3, t makes intermediate progress by finishing N's freeze or any thread makes intermediate progress in a constant number of steps after the completion of N's freeze recovery. ■

Another question is how many computational steps it takes for a thread t to complete the freeze of some frozen

node N. We state that it takes a constant number of computational steps and it does not depend on the number of concurrent executions.

Lemma 5.5: It takes $O(1)$ computational steps for a thread t to complete the freeze of some frozen node N, and it does not depend on the number of concurrent executions.

Proof: Frozen node N is immutable and in order to recover from its freeze N has to be copied, split or joined. To complete N's copy or split, thread t needs only to copy N's entries to a new node or nodes (individually) and try to swap N's *new* pointer to point to the new node using a CAS instruction. Regardless of whether the CAS is successful, t can continue with no need for a restart. This takes $O(d)$ computational steps and can not be influenced by any other concurrent executions.

The completion of N's join is more complex, because of the process of establishing a slave. When a master-slave relationship is established, it takes $O(d)$ computational steps in order to complete the freeze (as in the previous case). But how long does it take to establish a master-slave relationship? If N's slave candidate S is a normal node, it takes a single CAS instruction to enslave it. If S is frozen and doesn't need to be joined with another node, t needs to help with S's freeze recovery - spending $O(d)$ computational steps before enslaving S's replacement node. In the worst case S is frozen and needs to be joined with another node that also need to be enslaved. Theoretically the chain of the nodes trying to enslave their left neighbors can be of length d (maximal number of parent's children). In this rare case thread t may need to spend $O(d^2)$ steps helping other nodes in the chain. Note this also shows that the enslaving loop can repeat at most d times without any thread making progress. Since d is constant, we conclude that it takes $O(1)$ computational steps to a thread t to complete the freeze of some frozen node N, regardless of concurrent executions. ■

Lemma 5.6: The full progress of a search consists of $O(\log n)$ intermediate progresses.

Proof: The full progress of a search consists of reaching the relevant leaf (reachable from the root) and determine the key's existence. The leaf is found using the *FindLeaf()* method presented in Algorithm 1. The *FindLeaf()* method invokes only the *Find()* method on each node, going top down on the tree. The *Find()* method is not affected by frozen nodes, therefore it might need to restart on some node only in order to help physical deletion of an entry. If *Find()* method encounters no deleted entries, it proceeds in constant number of computational steps since the size of each chunk is constant. If there were deleted entries, the number of the steps required for completion of the *Find()* method on a specific chunk is equal to the constant number of the intermediate progresses (iia), which is physical deleting of an entry in the chunk's linked list. Once some node in the tree is transversed, the *FindLeaf()* method never returns back up the tree. The *SearchInBtree()* method, presented in Algorithm 3, consists of invoking the *FindLeaf()* method and then invoking the *SearchInChunk()*. Latter, in turn, consists of invoking the *Find()* method. Finally, the number of deleted entries in each node can be at most d, which is constant. From here the search's full progress is bounded by $O(\log n)$ intermediate progresses from the (iia) kind. ■

Please note that, unlike a full progress of a search, the full progress of an insert or a delete cannot be bounded with some constant number of intermediate progresses. This is because concurrent changes might result in the insert or delete thread going up and then down the tree. We assume a system, with only T insert and delete threads performing simultaneously. This is acceptable because search threads are not going to conflict with any delete or insert thread. We want to prove that after $O(T \log n + T^2)$ intermediate progresses, one of the insert or delete threads must make full progress and be returned to the user. We are going to prove it assuming harder conditions on tree balancing and thread dependency, and therefore the proof also holds for our actual B$^+$tree. In addition, we are going to refer to both merge and borrow rebalancing operations as join rebalancing operation that results in one new node. The merge and borrow operations are asymptotically equal in the number of computational steps and intermediate progress steps that need to be done.

Assumption 5.1: When the split of any node that has been replaced with two new nodes is complete, the first thread proceeding on one of the new nodes must make intermediate progress. The second thread proceeding on the same new node on which the first thread was successful, will make its intermediate progress as part of rebalancing (split or join) of the same new node.

Assumption 5.2: When the join of any two nodes, which has been replaced with new node is complete, the first thread proceeding on the new node must make intermediate progress. The second thread proceeding on the new node will make intermediate progress as part of rebalancing (split or join) of the new node.

In conclusion, we assume that the intermediate progress of every two threads causes a rebalancing of a node.

Let's justify the assumptions and compare them to the actual situation. After a split is done on our B$^+$tree node, resulting in two new nodes, one delete thread (and clearly one insert thread) has to succeed on any new node (Observation 5.3). The second delete thread (but not an insert thread) will succeed after a join of the new node, and if not this delete thread, then any other thread concurrently proceeding with the new node will make intermediate progress. Every freeze and thus every rebalancing operation end is causing intermediate progress (Lemma 5.4). In Assumptions 5.1 and 5.2, we bound by two the number of intermediate progresses between (any) two rebalancing operations, although, in our actual B$^+$tree it might be more than that.

Assumption 5.3: Finally, we assume dependence of all T threads on the same range of keys. This assumption leads to the worst case scenario, in which the computational steps in any rebalancing operation will be repeated by all the threads that are still running.

Lemma 5.7: In the worst case scenario, no more than $O(T \log n + T^2)$ intermediate progresses need to be done by T threads before full progress is made.

Proof: Consider the following scenario:

1. Every thread starts by a rebalancing operation that invokes splits or joins until reaching the root. This gives us $O(T \log n)$ intermediate progresses that need to be done from the beginning.

2. A thread that has made full progress is suspended before returning to the user; it is not given CPU before all the other threads have made full progress. (As we defined it, for full progress we also require a thread to return to the user.) Thus, such thread is "lost" whenever full progress is made, and we must allow a thread to return to the user if all T threads are lost.

3. All threads are focused on a single node's key range. for maximum dependence. That gives us that every rebalancing is repeated by all threads that are still running.

4. We allowed every thread to get up to the root and back (Clause 1), so after that at least one thread has made a full progress and is suspended. We remain with $T-1$ threads. In addition, it follows from Clause 1 that two threads will need to be suspended for the next rebalancing on any node in the threads' path (Observation 5.3).

5. In order to rebalance the tree to the height of i we will need to "lose" 2^i threads. Rebalancing the tree to the height of i involves iT' computational steps, where T' is number of the threads that are still running at that time. Therefore, the worst case scenario will be given by splitting and joining the same leaf all the time (height 1).

In conclusion, in the worst case we need to make: $O(T \log n + T - 3 + T - 5 + T - 7 + T - 9...)$ intermediate progresses before a thread that made full progress must be returned to the user. The number of intermediate

progresses is asymptotically bounded by $O(T \log n + T^2)$. Note that the linearization point of this thread happened before. ■

Definition 5.6: We can see each insert or delete thread having *a stack of intermediate progresses*, which need to be made in order to make full progress. Before the thread begins to run, a request for intermediate progress in the insertion or deletion of a key in the leaf resides at the bottom of the stack. When stack is empty, full progress has been made. As more intermediate progresses are needed in order to achieve full progress, they are added on top of the previous intermediate progress. More intermediate progresses might be necessary when the target node is frozen and the thread needs to finish this freeze for further progress. Although the intermediate progresses in the stack can be pushed and popped multiple times, the size of the stack is bounded by $O(\log n)$ intermediate progresses.

Lemma 5.8: The replacing loop can repeat at most twice before some thread makes an intermediate progress.

Proof: Let's assume thread t is executing the *ReplaceInChunk()* method on node N. The while-loop in the *ReplaceInChunk()* method can restart only if the entry whose data need to be replaced was found frozen and thread t didn't complete the freeze recovery. Let's look at thread t^{comp}, which did complete the freeze recovery. According to Lemma 5.4, either t^{comp} makes progress or some other thread will make progress on N's replacer in a constant number of steps. ■

Theorem 5: The B$^+$tree design retains bounded lock-freedom. In particular, there is a thread returned to the user at least every $O(T \log n + T^2)$ computational steps, where n is the number of the keys in the set and T is number of threads simultaneously proceeding with their operations on the B$^+$tree.

Proof: Our B$^+$tree's code repetitions can only happen as part of the enslaving or replacing loop. The constant number of those repetitions were proven to bring intermediate progress in Lemmas 5.5. and 5.8. Other new loops in the code, such as *while (node→height != 0)* in *FindParent()* and *FindLeaf()*, may have only as many iterations as the height of the B$^+$tree. If the B$^+$tree grows concurrently, this must be because of the intermediate progress of successful insertions of new nodes. The rest of the loops in the code were already proven as lock-free in [3] (Observation 5.1). The restarts there are due to the insertion, deletion, or freeze of a node. In Lemma 5.4 we saw that completing the freeze is always intermediate progress. Other than the aforementioned loops, the code can not repeat and will make progress as long as it gets computational steps.

So every repetition in our code is due to the success of some intermediate progresses. Therefore as threads progress with the computational steps, at least one thread is assured to make intermediate progress. In Definition 5.6 we see that sequence of intermediate progresses results in to the full progress. In Lemma 5.7, we proved that every $O(T \log n + T^2)$ intermediate progresses there is full progress in the tree, where n is the number of keys in the tree and T is the number of threads concurrently executing on the tree. Based on this, and by virtue of the fact that every repetition in our code is due to the success of some intermediate progresses, we conclude that there is full progress at least every $O(T \log n + T^2)$ computational steps. ■