**By: Roee Moyal &**

**Assaf Zaltsman**

**Supervisor: Kfir Lev-Ari**

**Non-Blocking SWMR Data Structures**

Table of Content

[Abstract 4](#_Toc397697849)

[Introduction 5](#_Toc397697850)

[The Problem 6](#_Toc397697851)

[SWMR Problem 6](#_Toc397697852)

[SWMR existing trivial solution: 6](#_Toc397697853)

[SWMR non-blocking solution: 7](#_Toc397697854)

[Project Objectives 8](#_Toc397697855)

[The Development Process 9](#_Toc397697856)

[C++11 9](#_Toc397697857)

[B+ Tree 10](#_Toc397697858)

[Uses 10](#_Toc397697859)

[B+ Tree properties 10](#_Toc397697860)

[Insert 11](#_Toc397697861)

[Remove 13](#_Toc397697862)

[Non-Blocking Implementation 16](#_Toc397697863)

[Node Implementation: 16](#_Toc397697864)

[Linked List Implementation: 16](#_Toc397697865)

[Array Implementation: 17](#_Toc397697866)

[Node Implementation Comparison: 17](#_Toc397697867)

[Insert adjustments: 19](#_Toc397697868)

[Remove adjustments: 22](#_Toc397697869)

[B+ Tree Comparisons: 26](#_Toc397697870)

[Choosing the Optimal Configuration: 26](#_Toc397697871)

[Non-Blocking vs Blocking DS comparison: 27](#_Toc397697872)

[Partitioned Framework 29](#_Toc397697873)

[Load Balancing - Chunk Support 30](#_Toc397697874)

[Add Chunk: 31](#_Toc397697875)

[Freeze & Remove Chunk: 31](#_Toc397697876)

[Garbage Collection 33](#_Toc397697877)

[Boehm Garbage Collector 34](#_Toc397697878)

[Our Garbage Collector 34](#_Toc397697879)

[The Collector’s Basic Flow: 34](#_Toc397697880)

[The Collector’s Implementation: 35](#_Toc397697881)

[Handling Wrap Around: 36](#_Toc397697882)

[Time Complexity: 37](#_Toc397697883)

[GC Comparison: 37](#_Toc397697884)

[Conclusions 38](#_Toc397697885)

[Open Issues 39](#_Toc397697886)

[Future Work 40](#_Toc397697887)

[Bibliography 41](#_Toc397697888)

# Abstract

Today, there's an ever-growing need for parallel access to data structures, as multi-core processors and distributed computing goes ever more prevalent.  
In this project, we explore the feasibility of converting an existing data structure (B+ tree) for use in a parallel environment of multi-core processors.  
We discuss the various parts of the resulting parallel data structure and algorithm, and the problems / considerations that led to their final shape.  
Another main discussion is the need of garbage collection and various ways of doing it. Finally, we compare our algorithm to existing solutions (such as C++ STL set) and show an improvement over several aspects.

# Introduction

Initially, algorithms were constructed as a series of instructions to solve a problem. These instructions were then executed in a serial fashion on a single CPU on a single computer. Much in the same way, the initial data structures that were used on these single-processor machines (also called sequential data structures) supported a single access at a time. Today, as multiprocessor architectures become more prevalent and provide ever-increasing levels of parallelism, the need arises for data structures that can accommodate multiple simultaneous accesses by different threads that run on different processors. Such data structures are referred to as concurrent data structures.   
Concurrent data structures usually reside in a shared-memory environment, where simultaneous accesses such as described above can occur. Such memory can be physically implemented on a single local storage module or a distributed collection of storage modules.  
This project covers a modification to a popular traditional (sequential) data structure, the B+ tree, to allow it to operate in a parallel environment.

# The Problem

## SWMR Problem

While allowing for multiple simultaneous readers alone is pretty straightforward, allowing a simultaneous writer and readers introduces concurrency problems. For instance: Suppose we have a key-value pair, (k,v) in a container and a ‘read k’ instruction is being executed by a thread, and the path to k in the container is P. If a write operation changes P, while the reader from the ‘read k’ operation is travelling on it, the writer could cause the reader to fail.  
The coexistence of several readers and one writer shouldn’t affect the consistency of the valid keys in the container and shouldn’t cause readers to fail.

## SWMR existing trivial solution:

The most straight-forward method of dealing with concurrency problems is using locks. A lock is a protection mechanism for enforcing limits on access to a resource in a shared environment. A lock is designed to enforce mutual exclusion (only one thread may acquire a lock and enter the critical section) While locks may seem a good solution at first, it quickly becomes apparent that they have many drawbacks. For instance, having too few locks in a data structure means that each lock spans a big portion of the data structure. As a result, much of the data structure is locked to several threads access, and only a few concurrent operations may be performed at a given time. In the base case of a single lock on the entire data structure for example, said data structure devolves into a sequential data structure. While increasing lock granularity (by adding more locks and assigning each a smaller portion of the data structure) may improve performance by allowing the data structure to accommodate more simultaneous operations, the amount of shared memory and the inter-processor communication required to maintain cache coherency over many locks, increase memory usage and reduce performance.

## SWMR non-blocking solution:

A more delicate method of dealing with concurrency problem is using multi-versioning of the data. Multi-versioning is a mechanism where the writer doesn’t change the existing elements, because they could be accessed by a reader, but creates a new version of the element with the data relevant after the write operation. Multi-versioning enforces a version of the container that is unique for each reader, at each point of its traversal. As a result, readers won’t fail because the version they see is unique and won’t change while the valid data in the container is reachable. This technique allows full utilization of the system’s concurrency while it requires a garbage collector in order to collect old versions.

# Project Objectives

* Designing a concurrent non-blocking, lock free, SWMR data structure (B+ Tree) in C++11.
* Design a designated GC for lock free SWMR data structures.
* Adapt a serial key value data structure into a concurrent SWMR data structure (Skip list).
* Compare the new concurrent SWMR data structures to existing concurrent and serialized data structures.

# The Development Process

The data structure is designed in C++11, using Microsoft Visual Studio IDE.

The design is test driven, using Google Test testing environment:

For each new entity we firstly write tests to prove the entity’s API correctness. Secondly we implement the entity, and finally we check it and debug it.

A test driven design will promise that in any stage of the design the functionality of all the designed entities is correct.

## C++11

The project was designed in C++11, using its multi-threading features. C++11 introduced a multi-threading library which includes Atomic types, Threads, Mutexes and more. Atomic types where used in our GC, as shared variables, we used CAS and Fetch & Add operations for our atomic operations. Our GC uses Threads as well and so does our test environment which uses Mutexes too.

# B+ Tree

## Uses

A B+ tree is a key value data storage system that provides low search complexity. As such, it is an ideal data structure for read-dominated workloads, and in today's world, is extremely useful in block- oriented data storages like file systems. B+ Trees have a large fan-out comparing to a regular B tree, and thus make them very flexible and allows the optimization to reduce I/O operations. Examples of uses in file-systems: NTFS, ReiserFS, NSS, XFS, JFS, and ReFS.

## B+ Tree properties

A B+ Tree is a graph that keeps the keys and values in a sorted order to allow an efficient insertion, deletion and retrieval of the value, given a key. The Tree starts at the top, with a single root. The order of the tree (b) is defined as the capacity of a single node, which is the maximum number of children it can have. In a regular Binary tree b=2. In a B+ tree b can be any number, but for convenience it is preferred to use multiples of 2. B+ Tree is an external tree, which means that traversal (or internal) nodes contain only keys and pointer/references to their children. The values are stored only at the leaves level. The tree itself is inherently a balanced tree – meaning the path from the root to any leaf is of the same length. This balance is achieved through strict rules that the tree needs to follow at any time: Let ‘m’ be the number of children a node can have, so for each node but the root:  and for the root .  
Any operation that modifies the tree, such as an insertion or a deletion, that might violate this property at any level of the tree, has to fix it. Insert operation uses split method to fix the tree while remove operation uses borrow & merge.

Insert  
Insert performs a search to determine which node the new key-value pair should go into.

1. Create a leaf node that contains the (key, value) pair and add it as a son of the appropriate node.
2. If the father has b sons or less then finish, otherwise.
3. Split the current node into 2 nodes, so one node has  sons and the other has  sons. Now if the current node was the root, then add a new root that will have 2 sons, otherwise the father has one more son. Go back to step 2.

Example:   
Tree with rank 3.

Insert key 6:

Split node (5,7):

Split root (9,18):

Remove

This action performs a search to determine whether a key-value pair exists in a tree.

1. If the pair was found, delete it’s node from the tree.
2. If the father has at least  sons, finish. Otherwise.
3. Try to re-distribute, borrowing from sibling (adjacent node with same parent as L), if the sibling has at least  sons.
4. If re-distribution fails, merge with one of the siblings. Now the father has one son less, go back to 2.

* Merge could propagate upwards to the root, potentially decreasing the tree's height (if the root has only one son).

Examples:

Borrow Example (remove 9):

Merge Example (remove 18):

# Non-Blocking Implementation

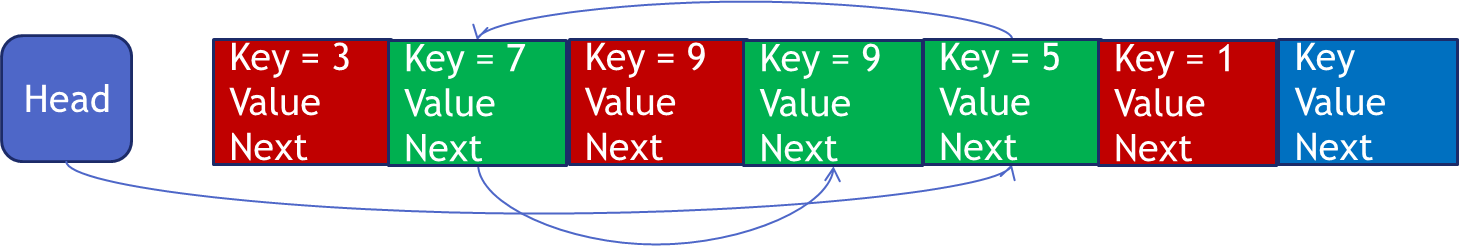
The B+ Tree was designed and implemented to be non-blocking. This means that tree supports the multi-versioning mechanism. Thus it requires a new and thorough design of the tree’s basic element - the node, so modification of keys in the node, which happens very often in this technique, won’t cause readers to fail.   
In addition, write operations such as insert & remove where redesigned to support multi-versioning.

## Node Implementation:

The node is the basic element in the B+ Tree, which contains keys and values. There are two types of nodes in our implementation – InnerNode which values are pointers to node and LeafNode which values are the values stored in the B+ Tree. Our node is implemented above an array for cache locality reasons.  
In the multi-versioning technique the node keeps the values of the invalidated keys, which causes the array to be sparse, hence compaction procedures are needed.

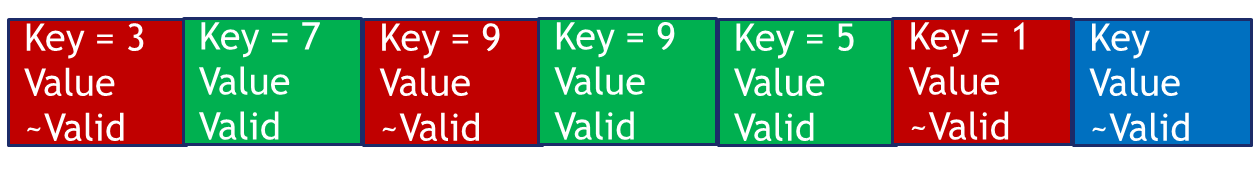
We tested two different designs - Linked list & valid bit array, in order to find which implementation is best suitable for our tree.

### Linked List Implementation:

The ‘linked list node’ is sorted, while the node is implemented over an array. Travelling on the node is done via the ‘next’ pointers, and therefore finding a key in the node requires scanning only the active keys. 

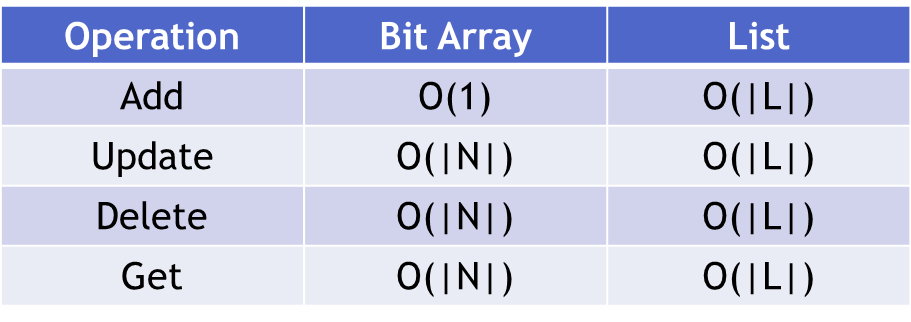
### Array Implementation:

The ‘array node’ is using a valid bit array in order to determine which entries are valid. In this implementation insertion of new keys is immediate, but keys aren’t sorted and the node has to scan all the keys that were used in order to find a key.



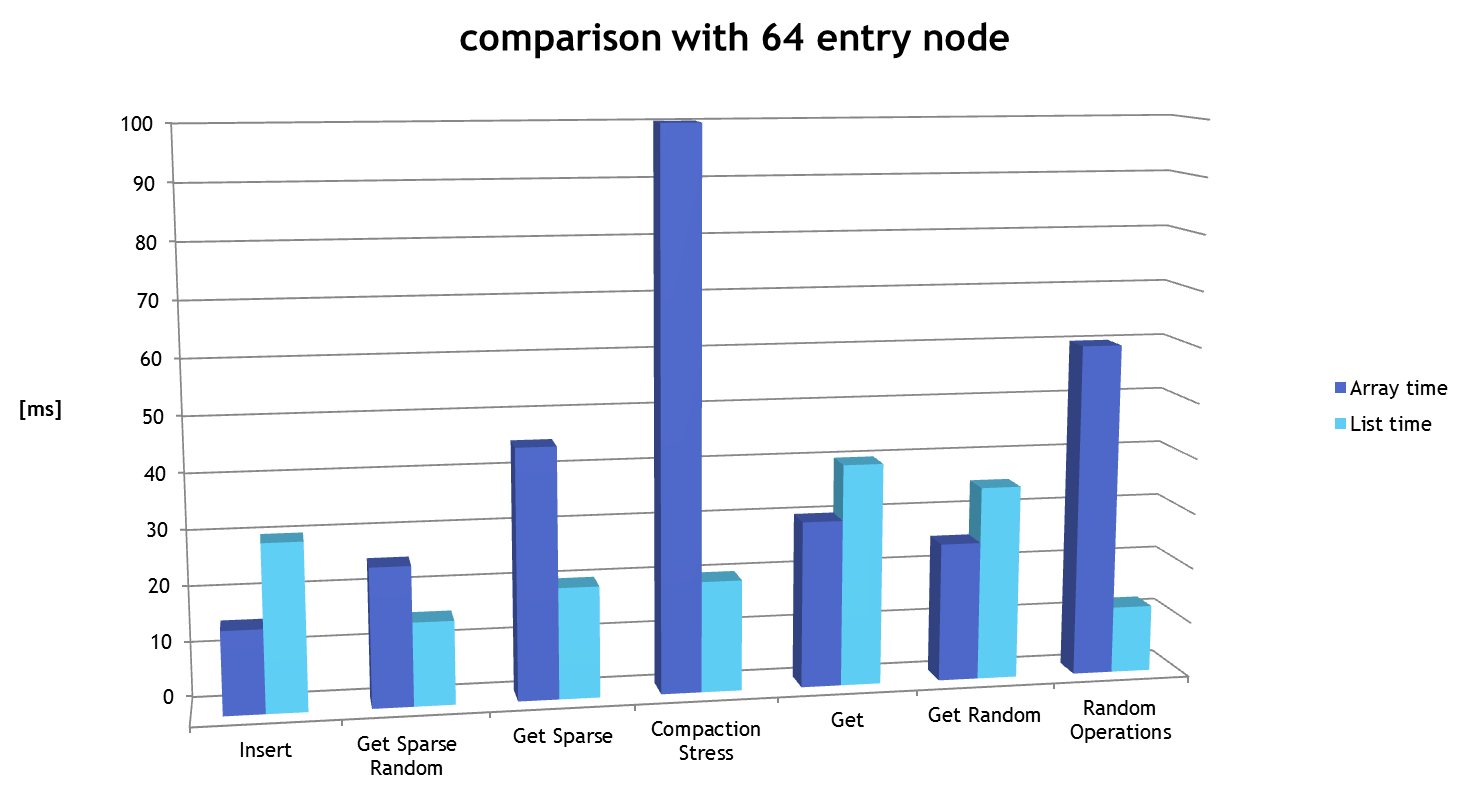
### Node Implementation Comparison:

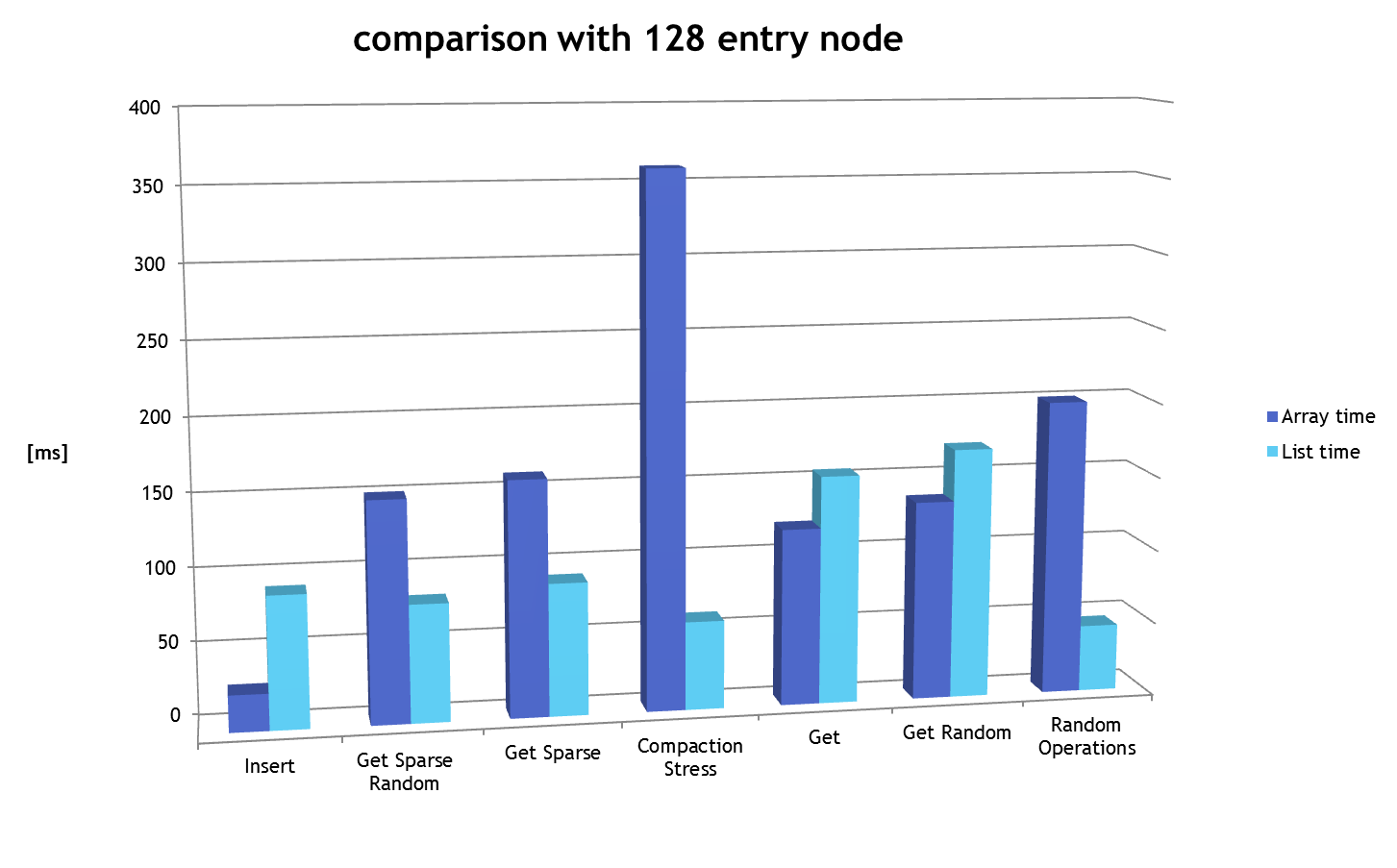
Time complexity of the node operations:



N – Number of elements inserted to the node.  
L – Number of active elements.

The tests were executed on Intel i5 Ivy Bridge processor with 4 threads.





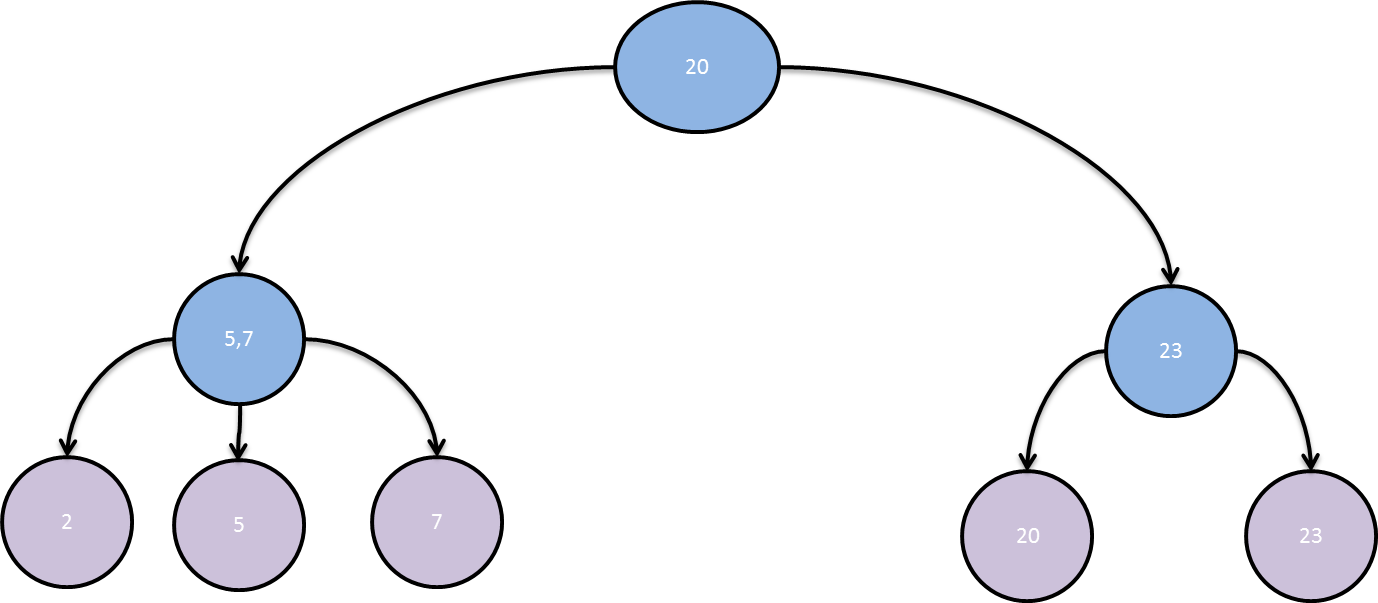
We chose the ‘linked list node’.

## Insert adjustments:

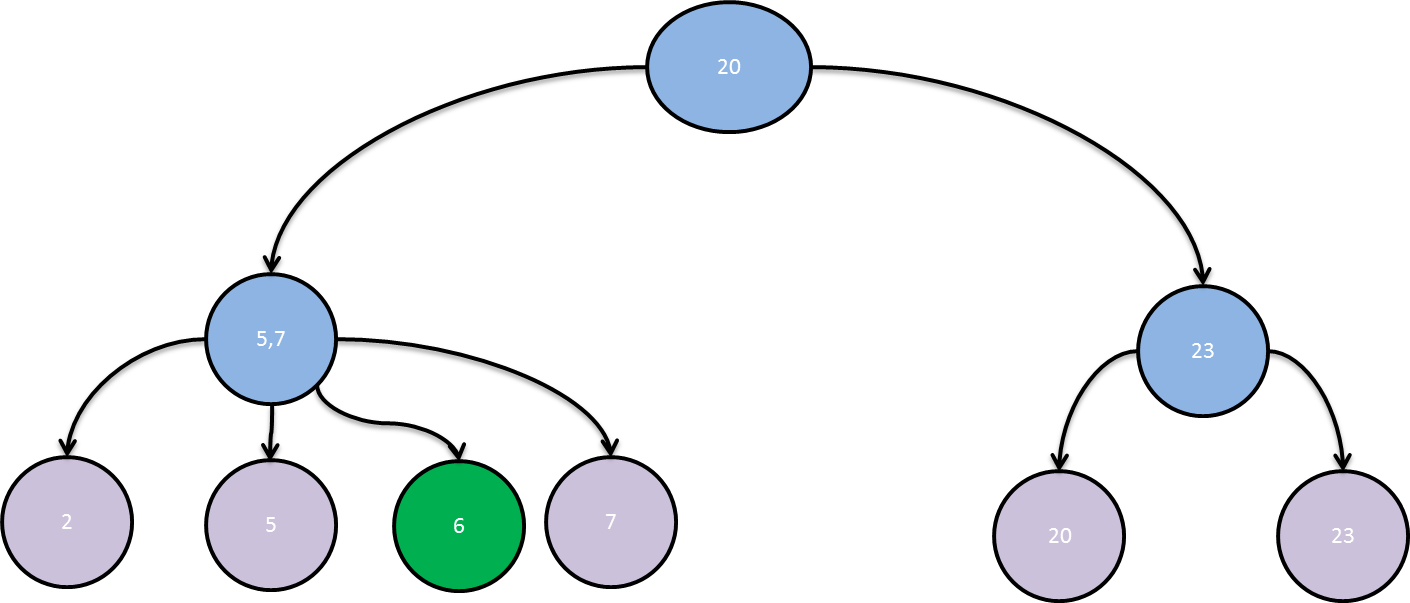
The insert operation was adjusted in order to support the multi-versioning technique. The adjustment was made in the split operation which is responsible for fixing the tree so the B+ Tree properties are kept.

When splitting a node, we first create two new nodes, where each node has half of the keys. After the two new nodes are ready, we replace the old node with the two new nodes in the father node atomically.

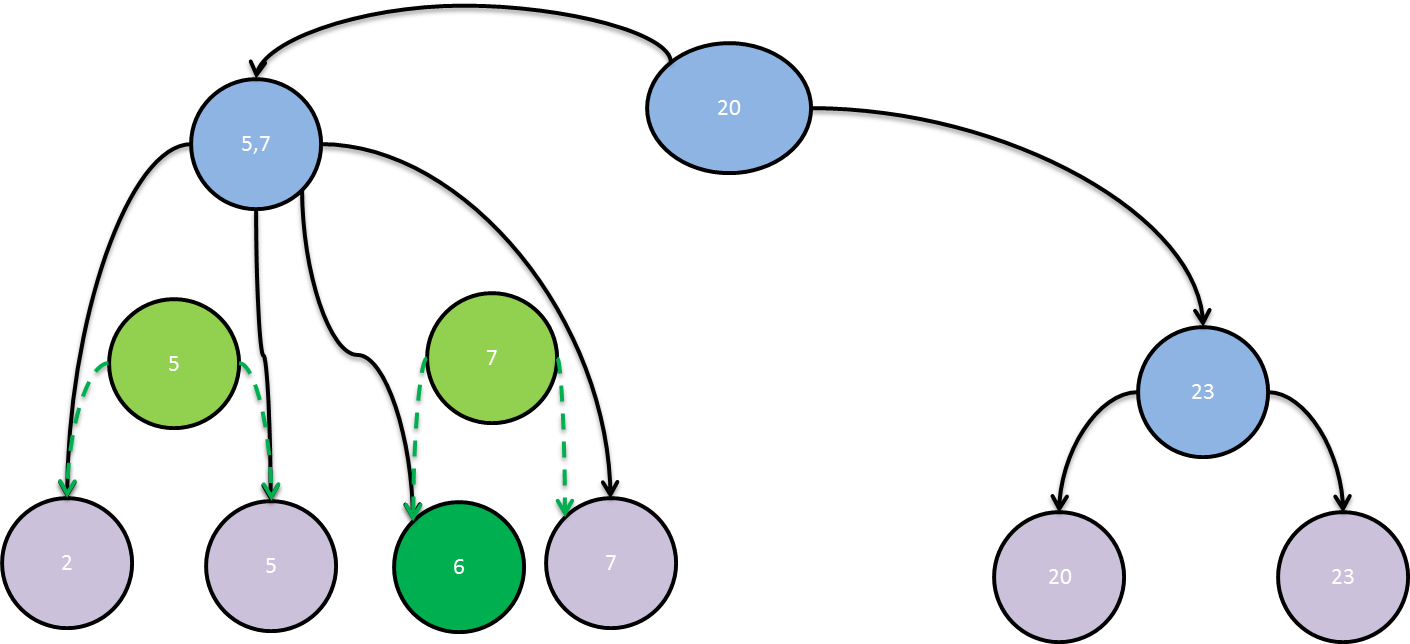
Example:



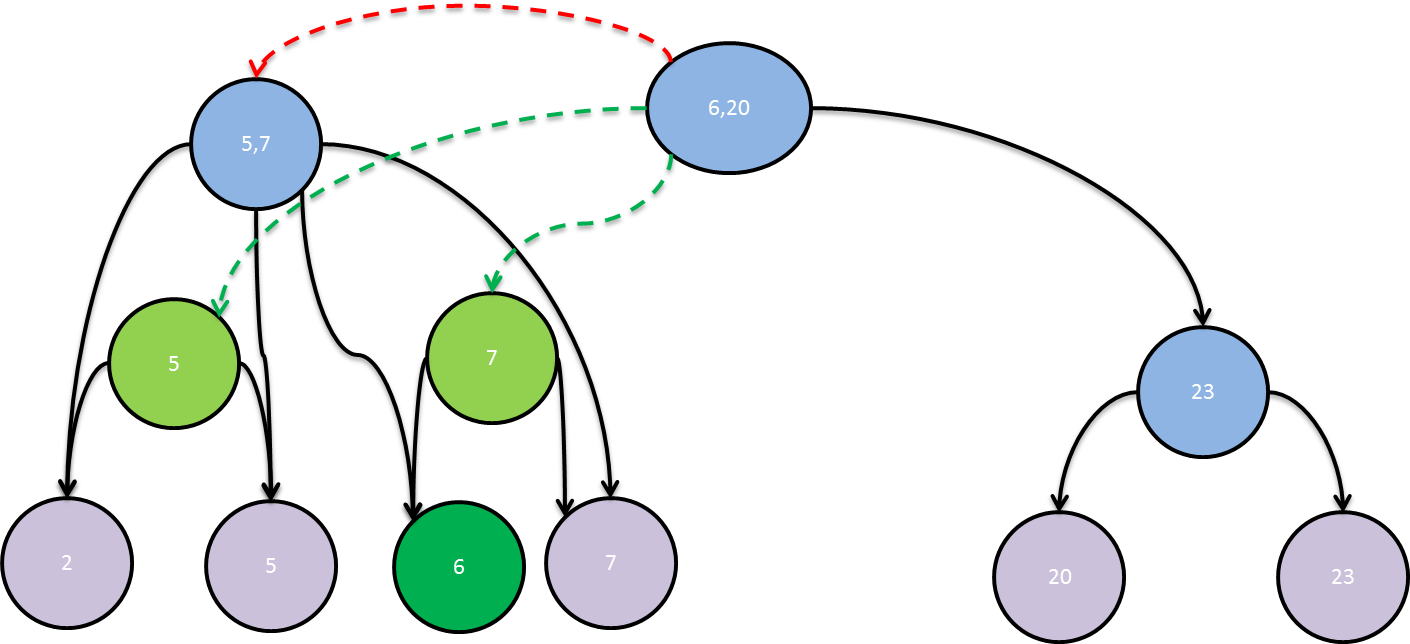
Adding a new key 6:



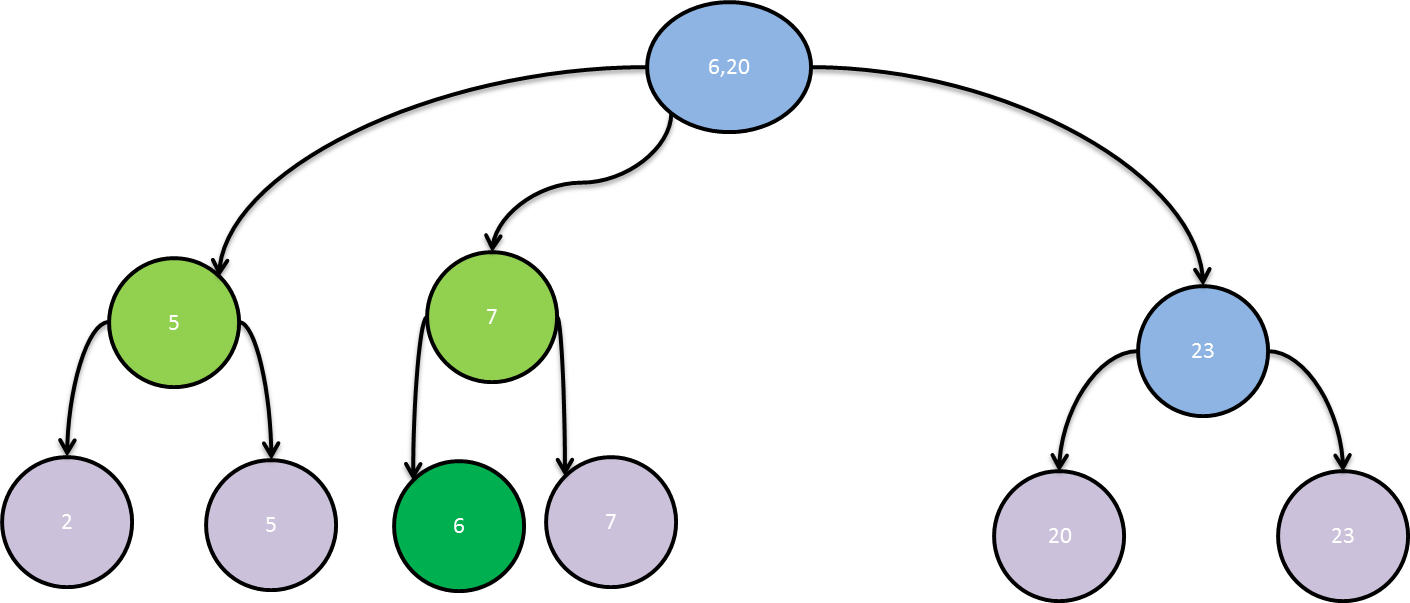
Split node (5,7):



Connect the two new nodes and disconnect the old one atomically:



After a while the detached node is collected:



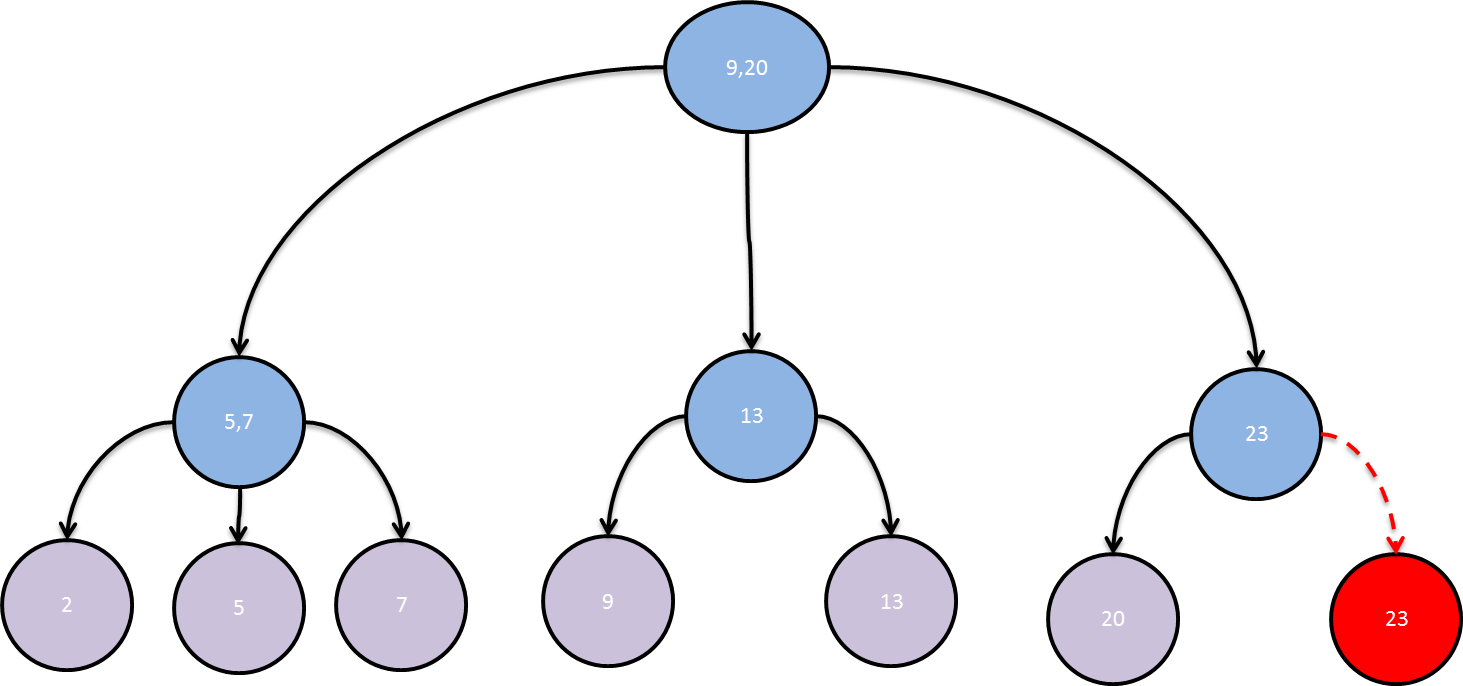
## Remove adjustments:

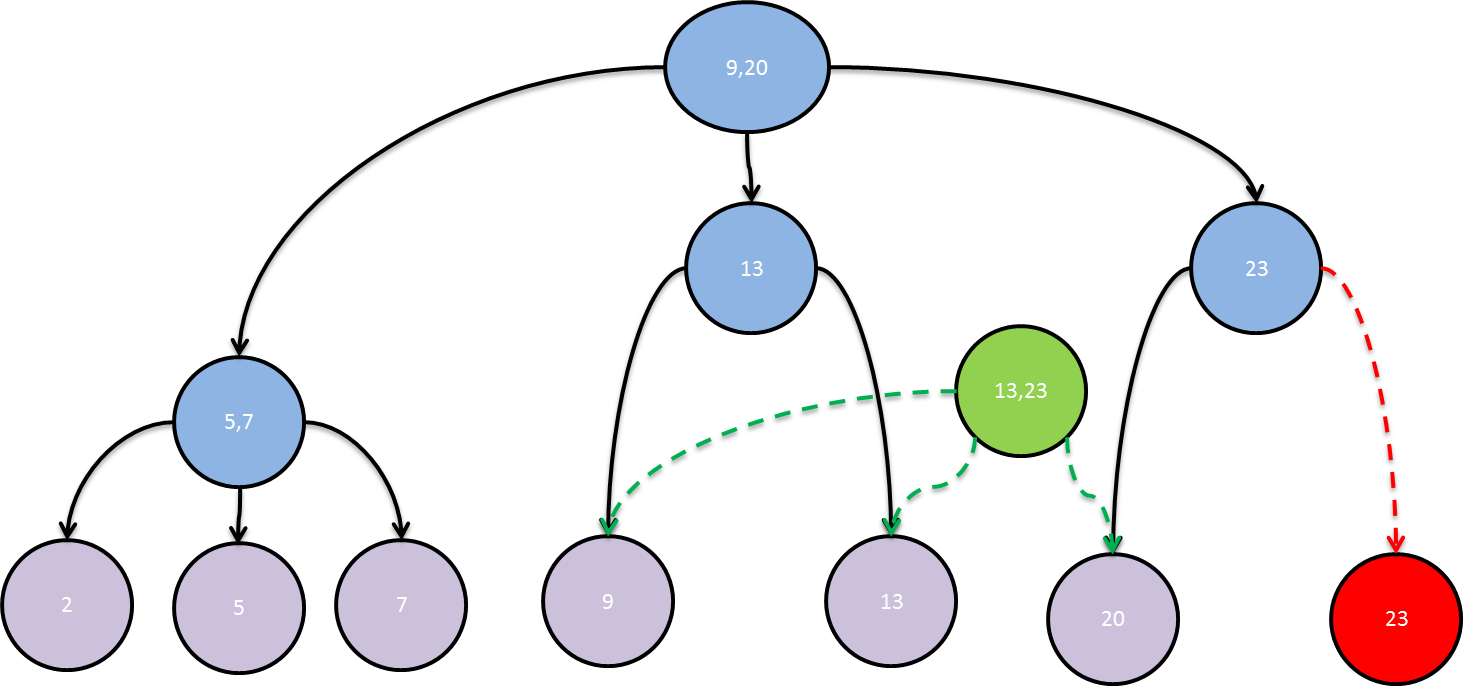
The remove operation was adjusted in order to support the multi-versioning technique. The adjustments were made in the merge & borrow operations which are responsible for fixing the tree so the B+ Tree properties are kept.

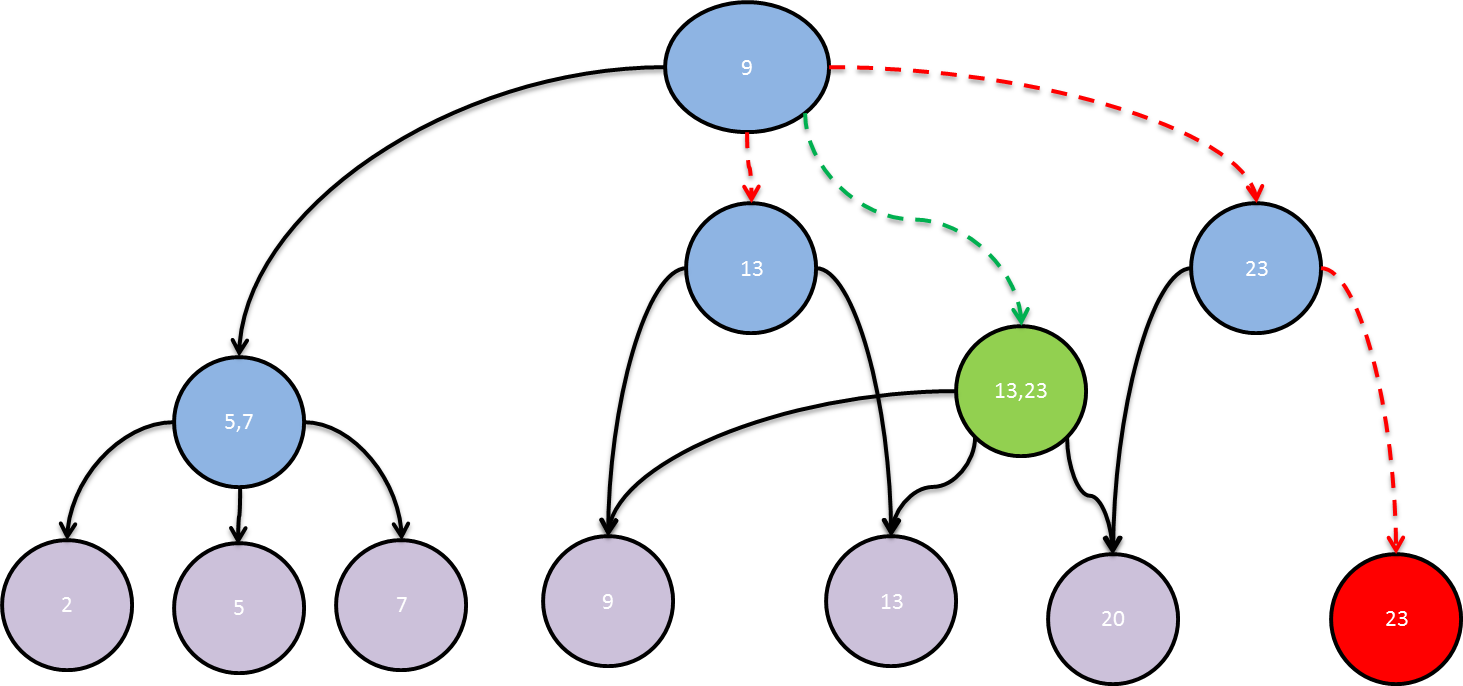
When merging two nodes, we first create a new node, which contains all the keys from the old nodes. After the new node is ready, we replace the old nodes with the new node in the father node atomically.

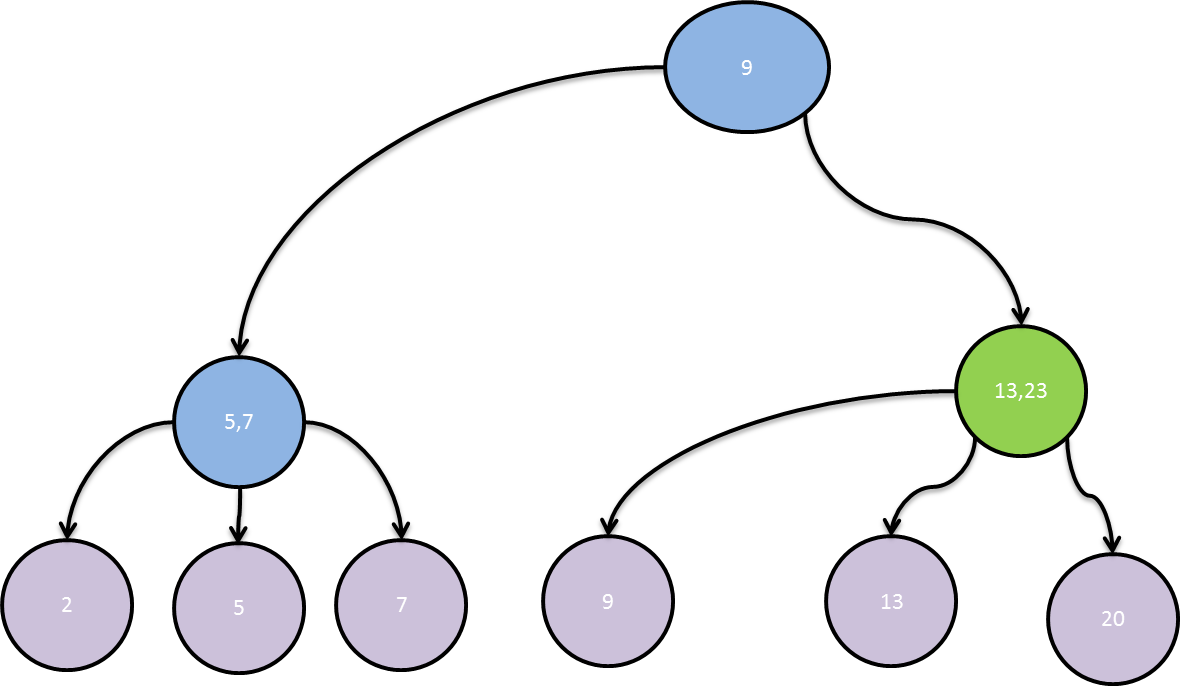
Example:

Removing key 23:



Creating a new merged node: 

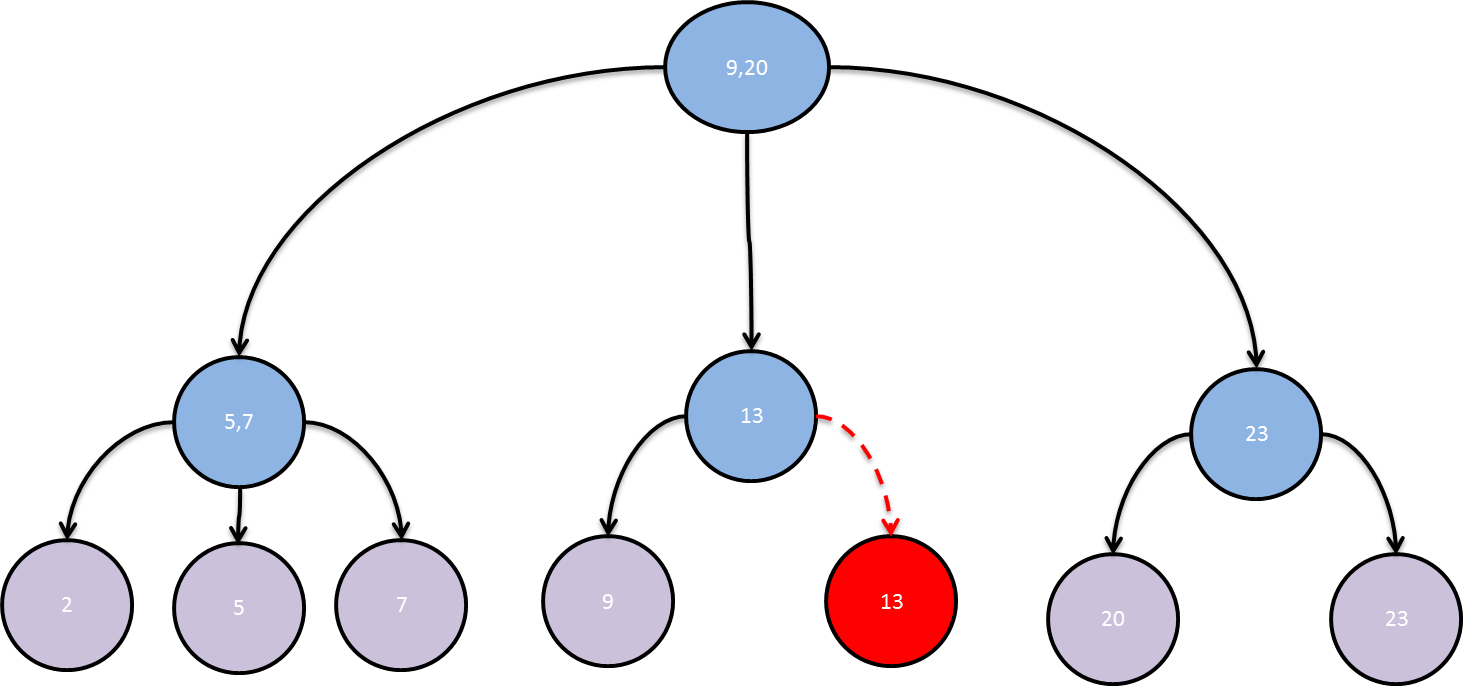
Change the father’s sons atomically:

After a while the detached nodes are collected:

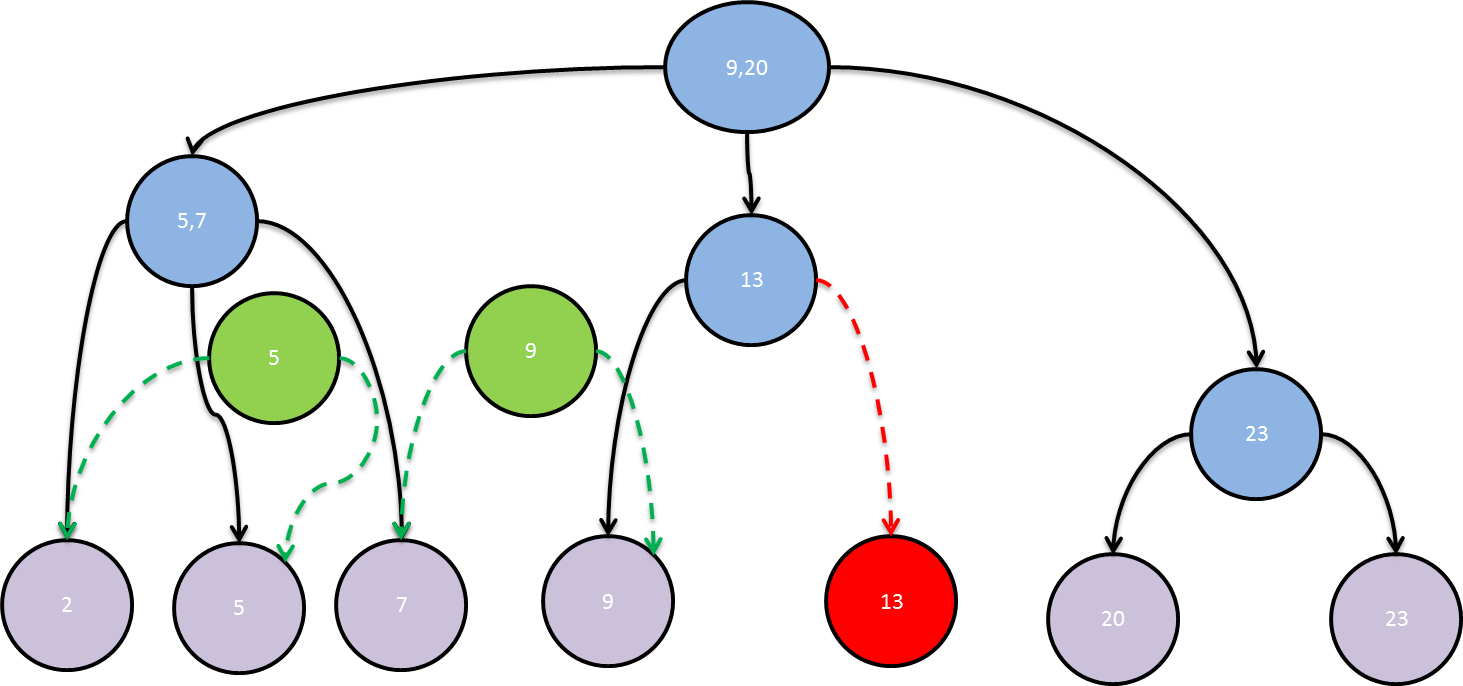
When borrowing a node from a brother, we first create two new nodes. The first one contains all the keys from the brother apart from the borrowed key. The second one contains all the keys from the borrowing node and the borrowed key. After both nodes are ready, we atomically replace the old nodes with the new nodes in the father node atomically.

Example:

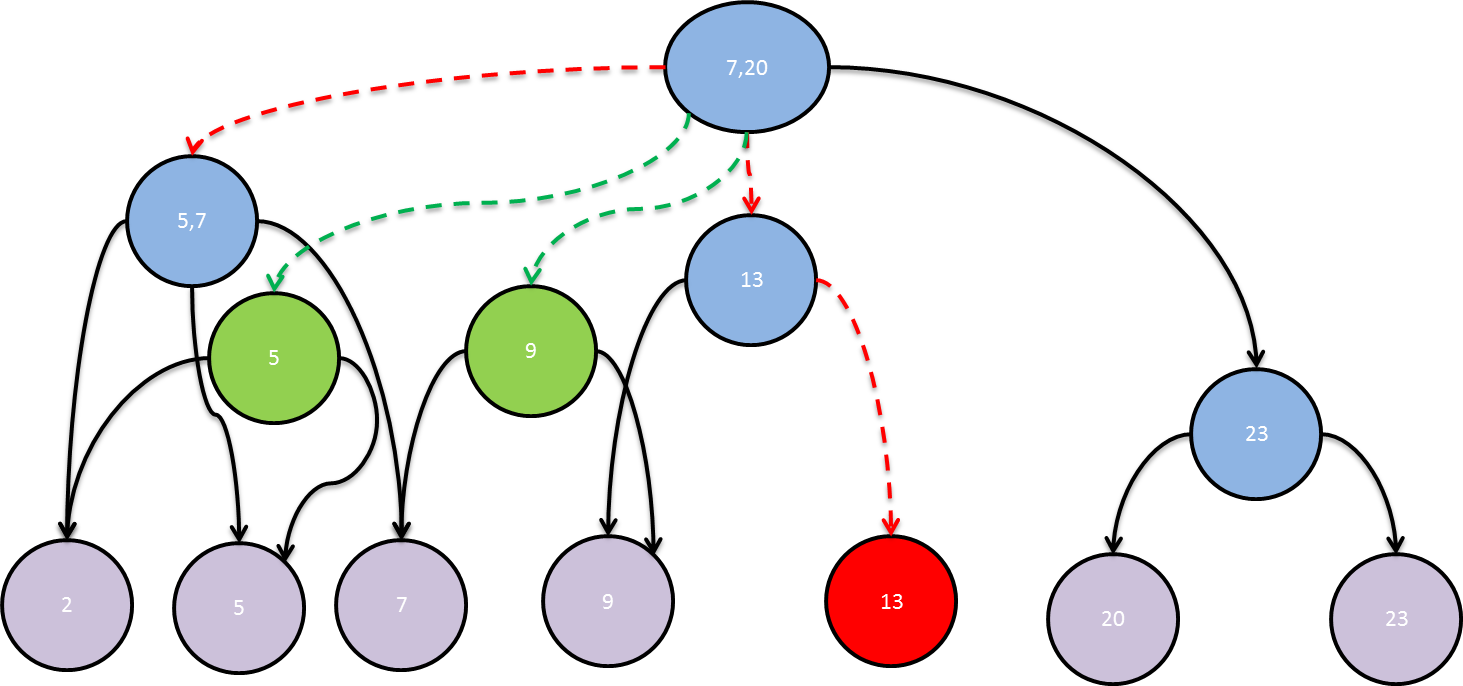
Removing key 13:



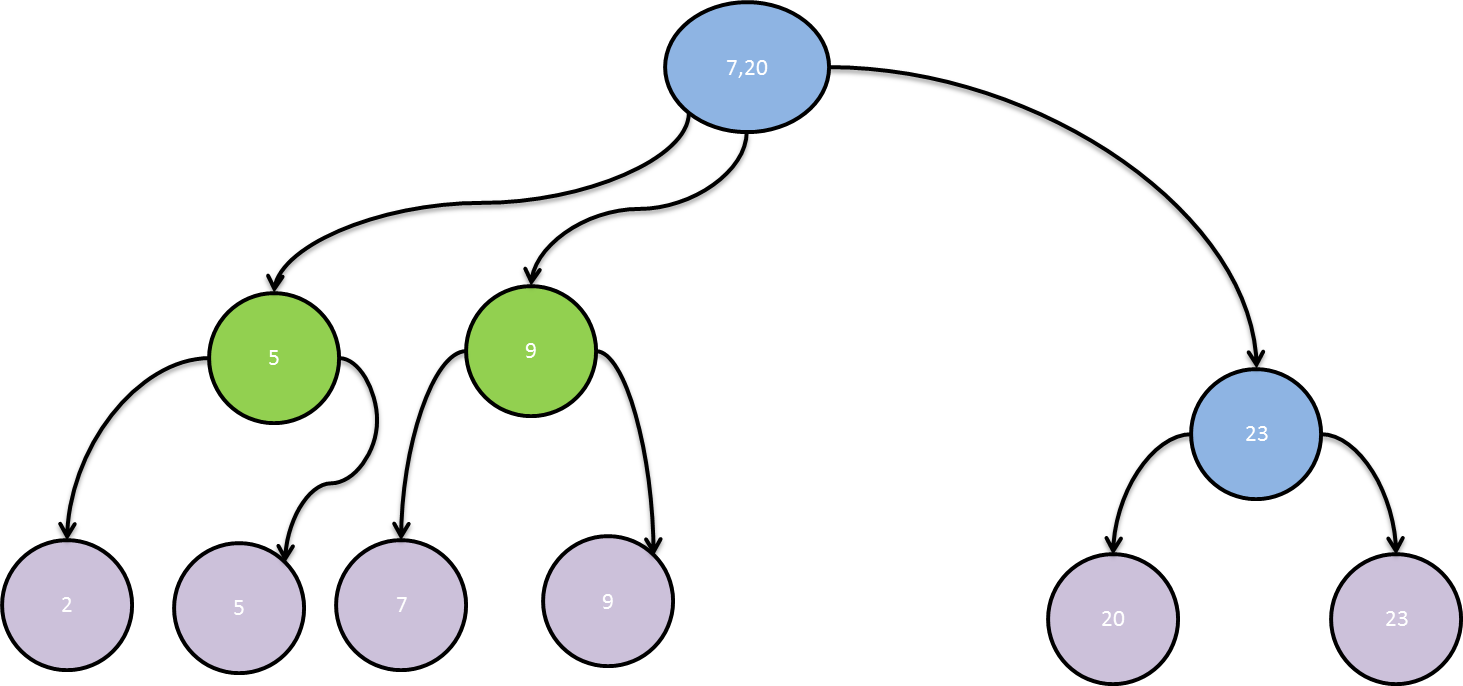
Creating two new nodes:



Change the father’s sons atomically:



After a while the detached nodes are collected:



# B+ Tree Comparisons:

Google Test environment was used in our design to check the correctness of the implementation and to compare the performance. Several typical workloads were tested for comparing the performance, using randomization and multi-threading.   
The tests ran on windows 7 OS, with Intel i5 Ivy-bridge core with 4 HW threads.

## Choosing the Optimal Configuration:

We tested our throughput on different array size & tree rank. As we explained earlier the array size is bigger than the rank because of the sparse array and we wanted to choose the optimal configuration for our workload.

We used two reading threads, using find operation and one writing thread that uses insert and remove operations.

As we see, the optimal configuration for our workload was rank = 17 and array size = 32. We used this configuration in all the further tests.

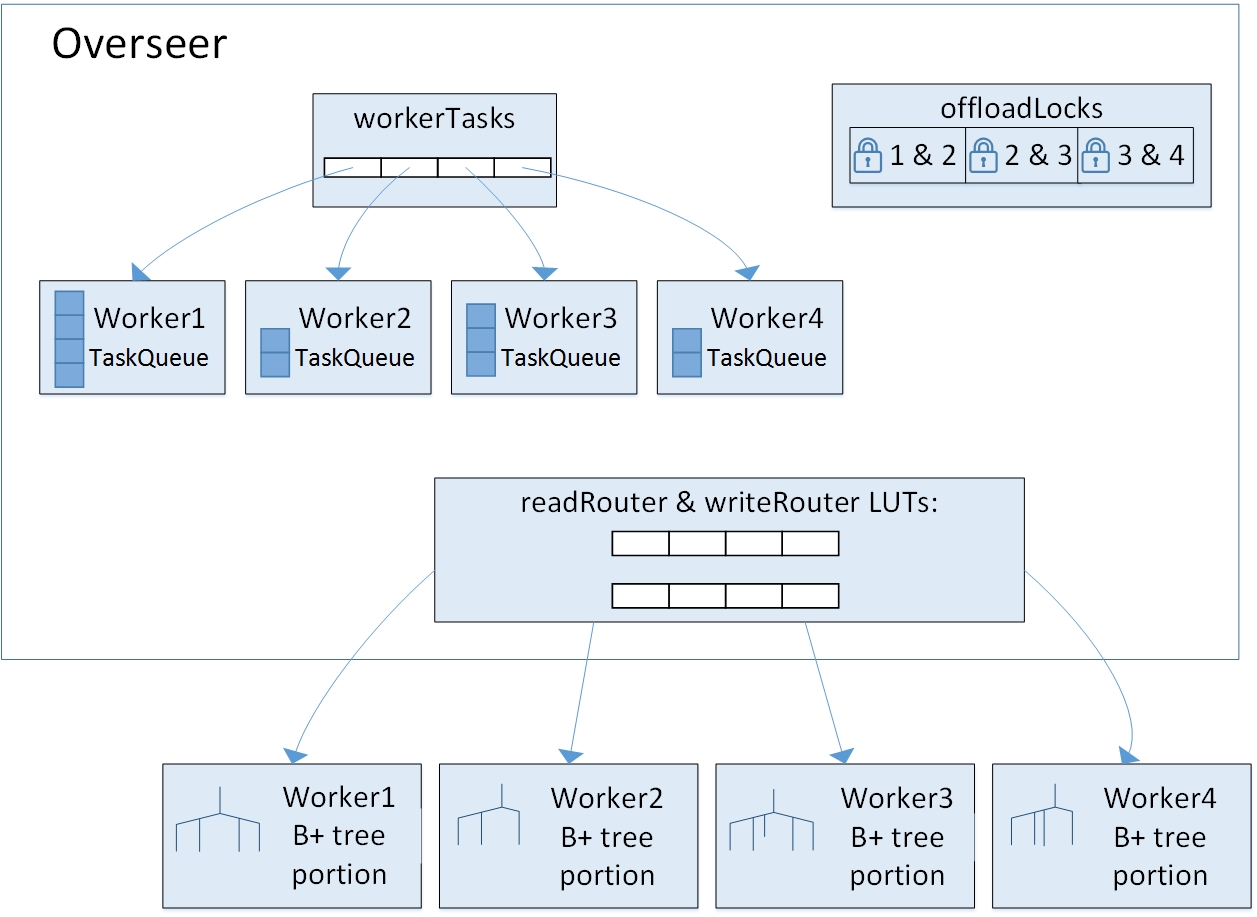
## Non-Blocking vs Blocking DS comparison:

We tested our B+ Tree implementation with our GC, timestamp based, which is a non-blocking design vs. an STL set blocking container. The set is a serial container that was converted to a concurrent container using the strait read-write lock.

In our comparison we see that even though the set container is faster on a single thread, using multi-threading, increases the number of  a thread can run. The number of threads running doesn’t affect the non-blocking container, but we can see it blocks the container that uses locks, which causes it for a longer response time.

# Partitioned Framework

The B+ Tree we designed utilizes the system’s concurrency between readers and a writer, but still, it allows only one writer. In order to allow more than one writing threads in the system a partitioned framework is needed. Such a framework, called The Overseer, was designed in our faculty. This framework consists of multiple trees which work in parallel and a worker’s queue which dispatches tasks to the trees. It is designed in way that when one sub tree becomes much busier than other sub trees, load balancing is done. Otherwise we might get into a worst-case situation where one working thread will have to perform all or most of the operations, while the rest of the threads will be idle. In addition the height of this tree might be very large so the read will become slower. Its queue can be full or very long so it'll take more time for the tasks to finish.

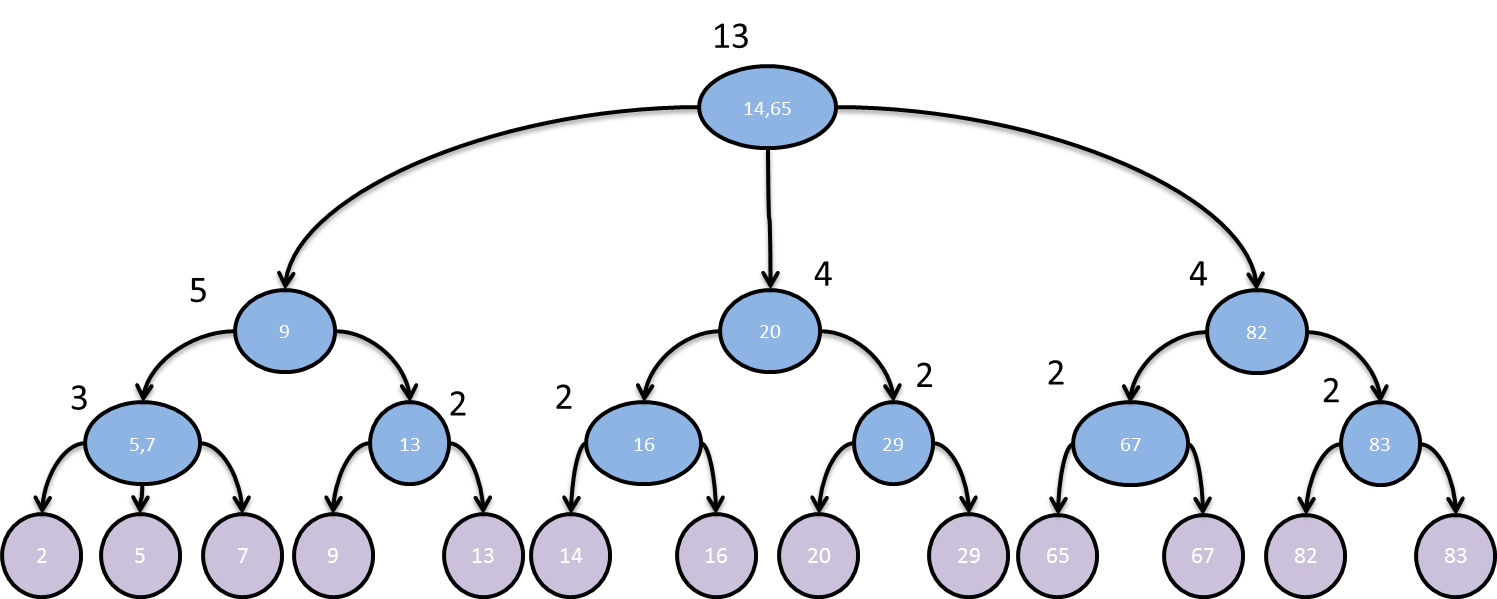


# Load Balancing - Chunk Support

Chunk support is a technique to manage load balancing between trees in the Overseer. The most important thing in load balancing is to keep the data validity consistency. The main problem with load balancing is that transferring a chunk of elements from one tree to another isn’t atomic and if not done correctly can harm the consistency of the valid data. In order to atomize the operation of transferring a chunk from one tree to its neighbor, we designed two complement operations –freeze & remove. When a chunk is to be transferred to a neighbor tree it must be first freezed, then inserted to the neighbor and at last removed from the current tree.

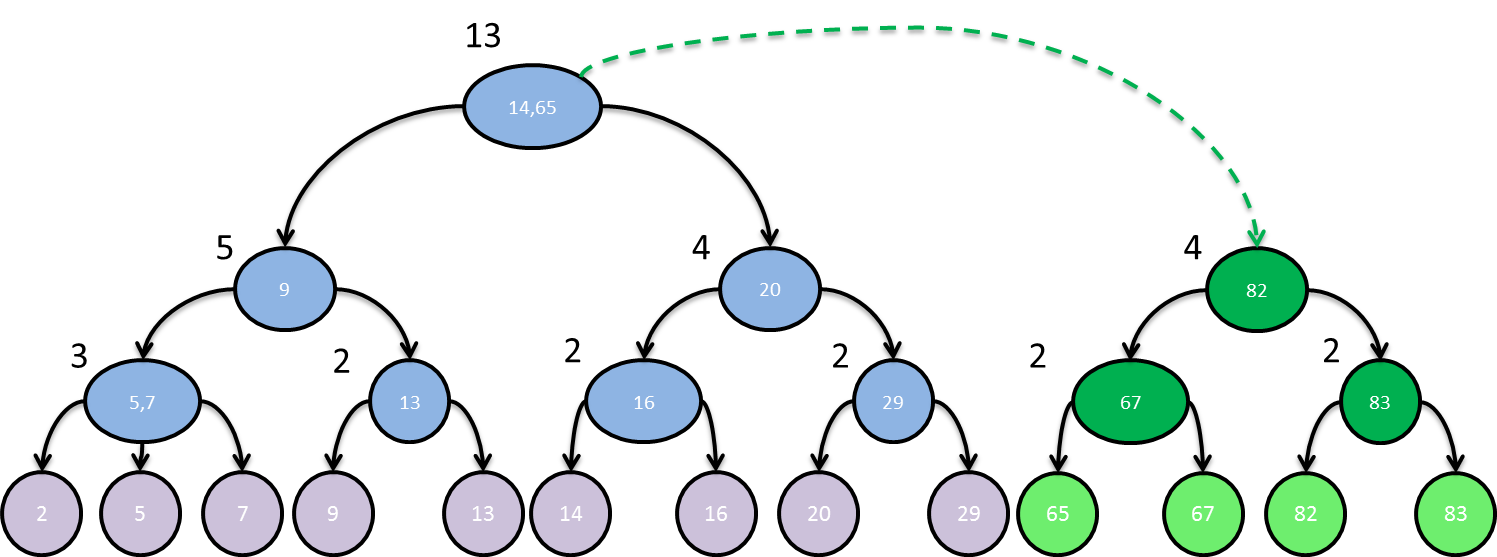
In our B+ tree we defined a chunk in the following way: a left (right) chunk of size X is the largest leftmost (rightmost) sub tree which includes at most X values.

In order to support chunks as defined properly we implemented a rank tree over our B+ tree.



## Add Chunk:

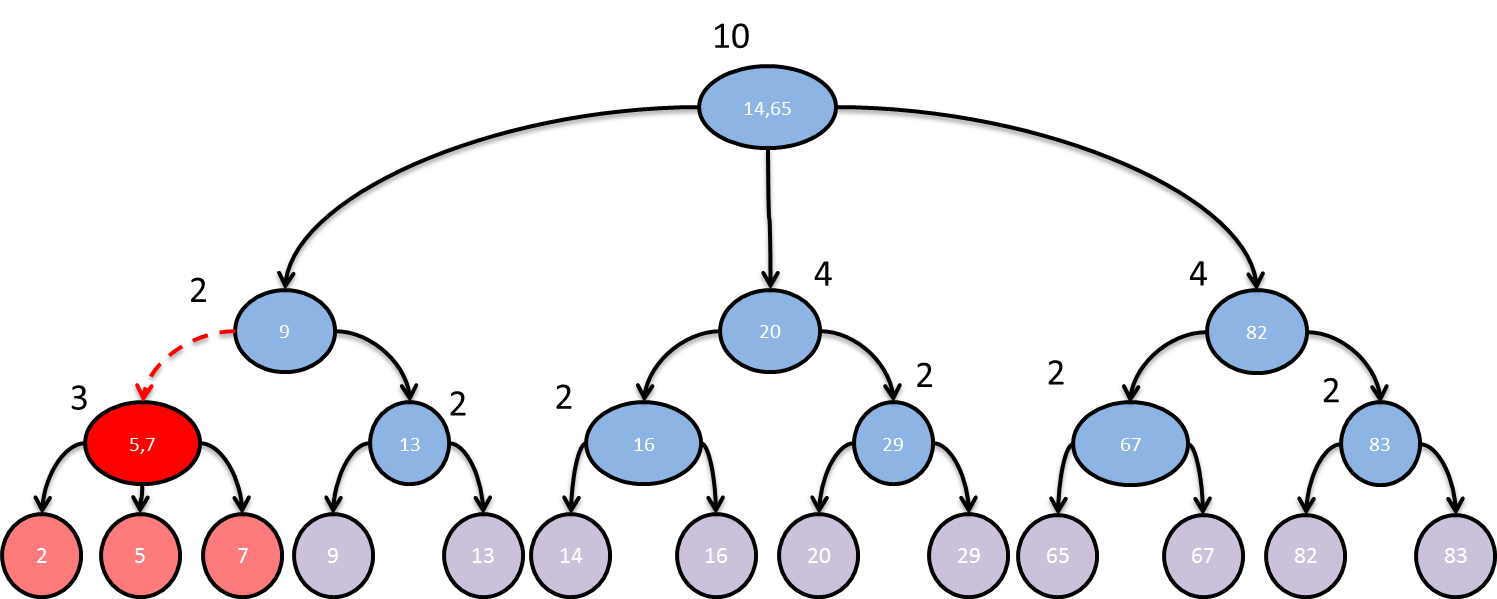
Using rank tree, a chunk can be simply added to the leftmost or the rightmost side of the tree. After adding the chunk, the tree is fixed from the father the regular way up to the root.



## Freeze & Remove Chunk:

Using rank tree, a chunk can be simply removed from the leftmost or the rightmost side of the tree. After removing a chunk, the tree is fixed from the father the regular way up to the root.

Removing the chunk is done in two steps, first we freeze the chunk – remove the chunk from the tree, save it in a designated place and allow it to be visible only for readers, meaning a reader can see the values in the chunk but a writer can’t modify them. After the chunk was added to the neighbor tree, the remove operation detaches the freezed chunk so it could be collected.



# Garbage Collection

The multi versioning mechanism, as explained in the previous chapter, as a side effect, generates detached parts of the tree. These detached parts aren’t reachable from the tree’s root once they are disconnected and hence cannot be manually collected by the tree’s interface. On any reasonable workload it is mandatory to collect the detached parts and de-allocate their used memory, therefore a garbage collector must be integrated into the container.

The C++ language inherently doesn’t support garbage collection, so we had to integrate an outer “daemon” which will collect the garbage. We tested two approaches for garbage collection:

* A generic conservative well implemented garbage collector called Boehm-Demers-Weiser garbage collector which we integrated into the runtime environment.
* A designated timestamp based garbage collector we designed and implemented to be suitable for any SWMR container. We integrated it into to tree’s interface.

Both garbage collectors are invisible to the user after setting up the runtime environment.

## Boehm Garbage Collector

The ‘Boehm-Demers-Weiser garbage collector’ is a generic garbage collector based on the ‘Mark & Sweep’ algorithm. The C++ language doesn’t distinguish between pointer types and regular types at run time and hence the Boehm collector is a conservative garbage collector. As a result, in the Mark phase of the algorithm the collector treats every 32-bit word in the heap as if it were a pointer.

The Boehm collector is integrated into the runtime environment, meaning it has to be enabled before making any operations on the tree. When integrated, any type of a variable in the program which is allocated using ‘new’ must inherit the type ‘gc’.

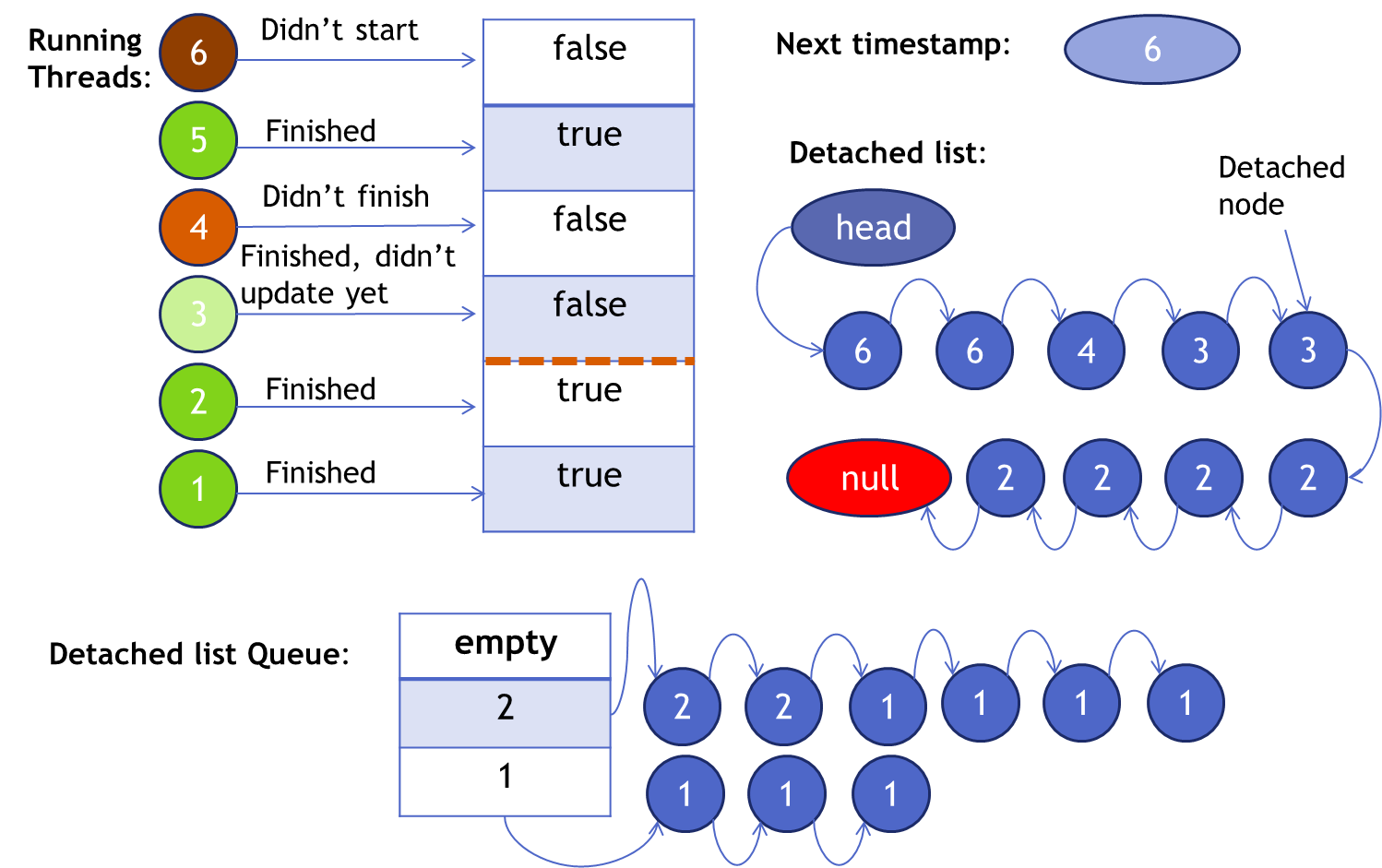
## Our Garbage Collector

We designed and implemented a designated Garbage Collector which is based on timestamp ordering. The main goal was to design a designated GC that will work on SWMR containers (not only on our tree), which will have minimum interference with the heap and will exploit the qualities of the SWMR design.

### The Collector’s Basic Flow:

* + Each thread that begins operation on the container receives a new timestamp.
  + Each node that is detached receives the next timestamp to enter the system.
  + GC daemon: checks which detached nodes can be released, and releases them. A node can be released if the oldest timestamp valid in the system is at least the timestamp given to the detached node.

### The Collector’s Implementation:



Threads which enter the system receives a new timestamp, by using Fetch & Add operation on the atomic ‘next timestamp’, and when they leave the system they publish their timestamp in the ‘boolean array’, by setting the appropriate entry to be true. In addition, when the writer thread detaches nodes from the collector, it gives them the value of the atomic ‘next timestamp’ and inserts them to the atomic ‘detached list’ using CAS operation.

‘GC Daemon’ wakes up at a configurable time, when so it updates the oldest time stamp in the system. In the example, the next time the daemon wakes it would updates the oldest timestamp to be 3. Next, it inserts the ‘detached list’ into the ‘detached list queue’ using CAS operation. Finally, the daemon frees each list in the queue which oldest timestamp is at most the oldest active timestamp.

Handling Wrap Around:

In long term workloads the ‘finished thread array’ and the ‘next timestamp’ may become saturated and reach their maximum values. In order to deal with this problem we used two ‘finished arrays’ instead of one while the daemon works only on one array at each time.

The LSB of the ‘next timestamp’ indicates the relevant array for the timestamp, meaning every finished thread accesses the array indicated by this LSB. The other bits of the timestamp serve as the timestamp value and the index in the array.

The daemon switches the LSB of the ‘next timestamp’ when the ‘finished array’ reaches a configurable threshold and saves the maximum value of the timestamp. After switching the LSB, the new detached parts and the new threads receive a timestamp which is addressing the new array. After the thread, which timestamp’s value equals the saved maximum value finishes, the daemon releases all the detached nodes from the old array and begins to work on the new finished array. Finally, the switching is done and all the entities are working on the same array. Switching the arrays can be done only when all the entities are working on the same array and the threshold was reached. If the new array is fully saturated and the daemon releases elements from the other array, then all new operations are blocked until the daemon finished cleaning the old array.

### Time Complexity:

* Tree interface operations:
  + Receives a timestamp in the beginning - added F&A.
  + Publishes it finished working on the tree, O(1).
* Add & Remove operations:
  + Publishes detached nodes - added CAS.
* GC Daemon:
  + Updates which readers left the system -.
  + De-allocates unreachable nodes - .

### GC Comparison:

* Both GC have similar results.
* Our GC has a little advantage in average.

# Conclusions

The project goal of modifying the B+ tree from a sequential to a concurrent SWMR non-blocking data structure was achieved using C++11 and its new multi-threading features.

We successfully designed a designated GC for our design, which showed better results than a generic conservative GC.

We adapted a java serial ‘Skip List’ implementation into a SWMR data structure and found that although the effort isn’t too big, in order to achieve a remarkable improvement, it requires a re-design of the container as an SWMR non-blocking implementation that supports the partitioned framework.

We compared the concurrent non-blocking SWMR vs. blocking Set. The results showed that adding new threads to the system doesn’t harm the performance of our non-blocking implantation, while the blocking implementation’s performance is damaged.

# Open Issues

* Our garbage collector was designed for a single container. The framework requires supporting several containers, which raises an issue about the need of timestamp correlation between the containers. We thought of two solutions to the issue:
  + Deep copy the chunks that are moved for load balancing so the container that owns them would free them
  + The Timestamp GC will be a shared resource of the partitioned framework.

There is a tradeoff in the two solutions, and a future work could check which one is better under the Overseer framework workload.

* When the system is under an instance workload, it might start blocking I/F operations because of the GC implementation (Wrap Around). We implemented this blocking mechanism as an initial solution, but there may be better solution and the solution depends on the Overseer framework workload.

# Future Work

* Implement the Overseer in C++ as a partitioned framework to maximize our B+ Tree abilities.
* Our GC performance has a linear dependency with the number of readers and the number of elements that were detached. Depending on the Overseer framework workload, this dependency could change, for example, every timestamp value will have of threads with the same value. This for example could be one of the solutions, so even with intense workload our GC won’t block new operations.

# Bibliography

* Wikipedia Articles in: Parallel data structures, parallel computing, B+ tree, Locks, Boehm GC.
* B+ Trees – by Susan Anderson-Freed, 1998, Clark university