# High-Level Design Document for Efficient Wait-Free Resizable Hash Table project

|  |  |
| --- | --- |
| **Author** | Alon Libling & Tal Gelbard |
| **Version** | V0.1 |
| **Created** | 15/04/20 |
| **Last Updated** | 15/04/20 |

# Introduction

## Assumptions and Design Constraints List

We assume the client isn’t malicious and will not try to find vulnerabilities in our data structure. We also assume the maximum thread count to be used in our DS is 128, but this could be easily expanded.

## Dependencies

Our DS will require very little dependencies in its basic form, which are from the *std* and *pthread* libraries. We also require that the core architecture will have an atomic CAS implemented, but this should be standard in today modern CPUs.

In the future we might add dependencies related to the hash function we are using and smart pointers (GC) for better overall performance.

## Issues List

Currently there are no open issues, we are only at the implementation stage.

## To-do List + Expected time-tables

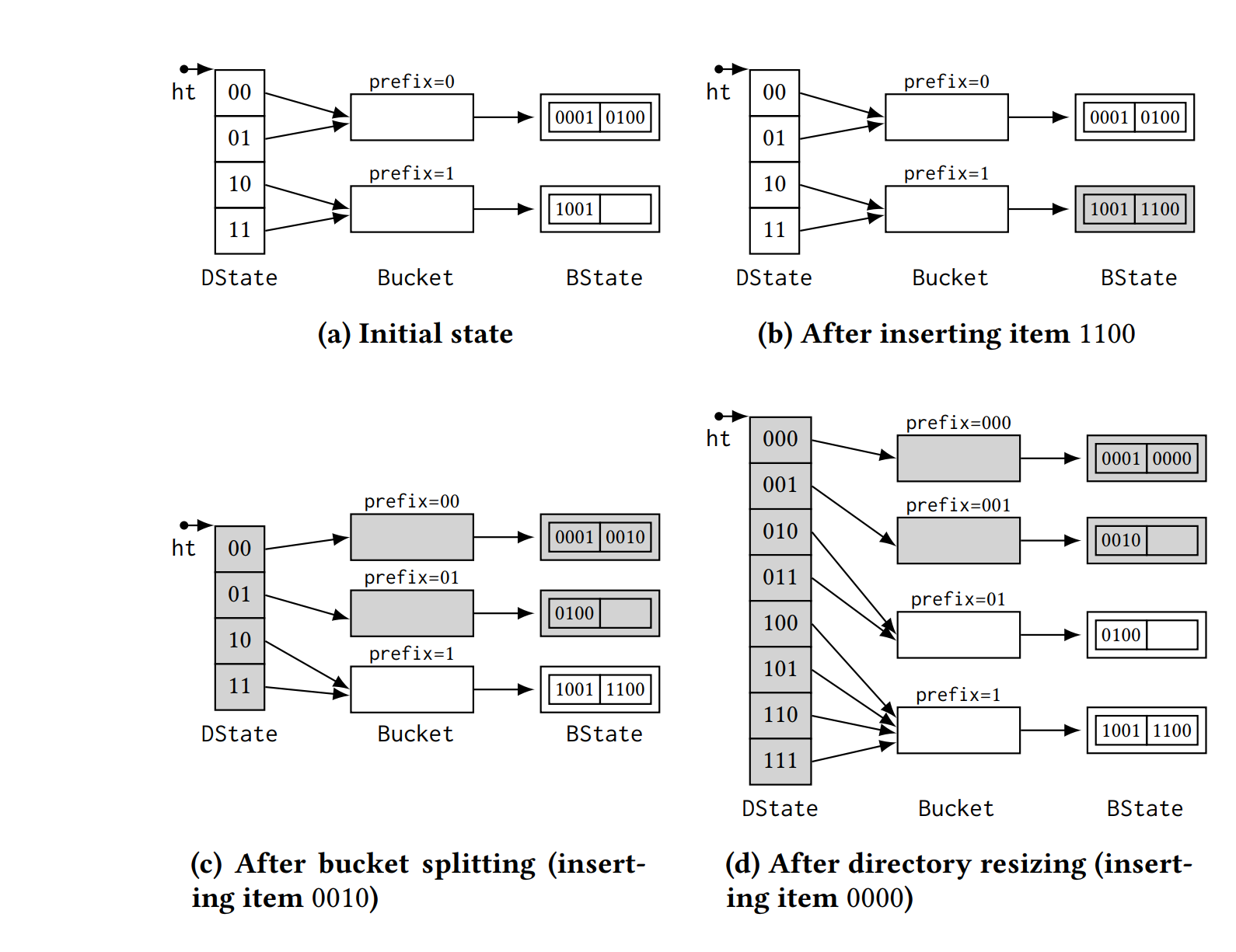
Implementation of the data structure with basic required features until the end of April.

Testing & debugging – end of June.

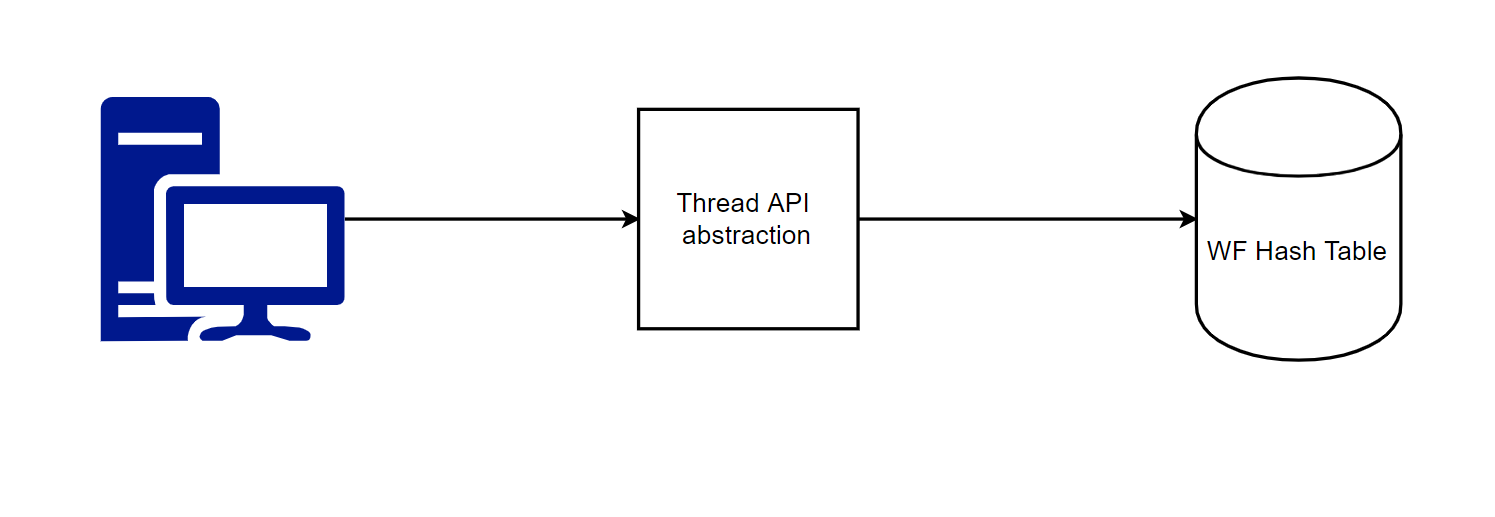
Extra features & benchmarks – end of July.

# Logical Architecture

The paper we base the DS on does a pretty good job at this abstraction:



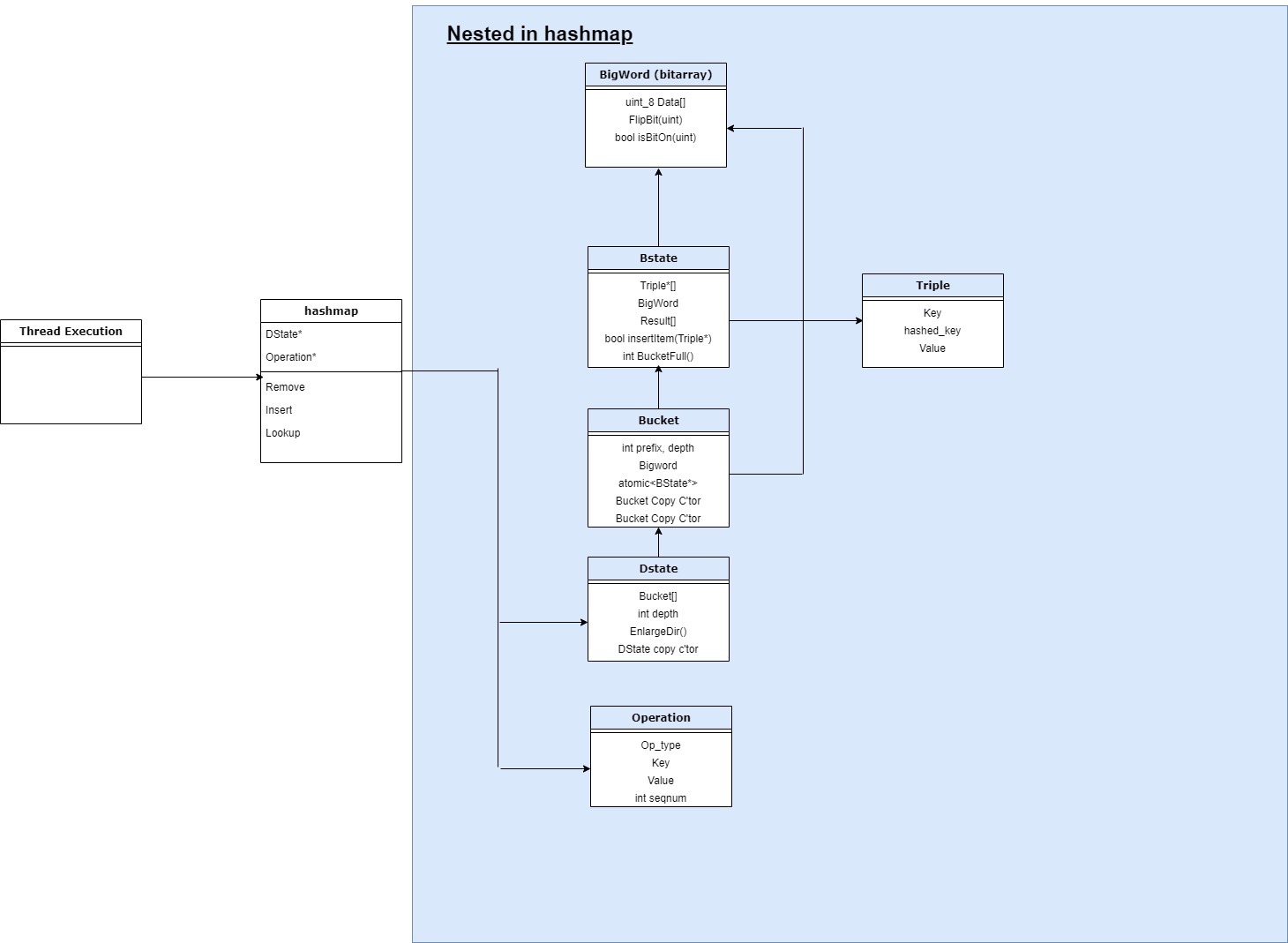
Logical abstraction of the implementation:



# Design

## Classes

### Class Diagram



### Class details

The elementary types (such as Operation and Triple) are used with default constructors. The other classes were implanted in order to make implementation more modular and clearer, and since an allocation operations were been made using new and delete at the places that require for the resizable attribute. The variables who needs to be replaced using atomic CAS operations (DState\* & BState\*) were declared using the std∷atomic library. In addition, as detailed within the paper, some of the arrays are at the size of the thread number, and each thread will use only it’s suitable cell.

hashmap:

* All the user operations are being made on the hashmap’s instance.
  + This is the only class that the user can access (and only to known public operations such as lookup, insert, remove).
* Stores a pointer to the DState (using std∷atomic) and two arrays at the fixed size of the thread number.
* Contains all the following neseted classes and structs within it.

DState:

* This class exists in addition to the hashmap mostly in order to make it possible to replace DStates within the hasmap using CAS while updating or changing the *dir* without delay or the risk of losing information.
* Stores a pointer to an allocated array *dir* of pointers to *Bucket*s. The array’s length can be deduced by powering 2 to the mutable primitive integer *depth* which is also in DState class.
* Notice that more than one cell within the array may point to the same Bucket.

Bucket:

* Stores mainly a pointer to uniquely BState (using std∷atomic), and other variables (Prefix, depth, toggle).
* The purpose of this class is to allow an atomic change within a BState without delaying operations or losing information as explained in the paper.

BState:

* Stores mainly a pointer to a fixed size array of Triple and another fixed size array of Results.
* Every Triple contains the hashed\_key. The highest binary *depth* (the depth is an integer within Bucket at listed above) is equal to the pointing Bucket Prefix variable (also integer within the Bucket).

BigWord:

* Since every BState and every Bucket uses an array of boolean, and an array of boolean type is heavy this class is literally a boolean array with much lighter implementation.
* The classes uses threads for marking T/F values, as described in the paper (toggle & applied).

## Setup

Setup should be easily done by including the header of the hashmap or the thread abstraction in the client’s program. Constructor is provided with the explained operations.

# Use cases

This implementation of a WF hash table surpass any other available implementation by quite a lot in cases that the resize operation doesn’t occur often () for multi-threaded programs. This characteristic is useful to many software applications like those who provide searching as a main feature, for example: address searching (Google maps), public transportation searching (Moovit) etc.

# References – to external papers/packages

[An Efficient Wait-free Resizable Hash Table](https://tropars.github.io/downloads/pdf/publications/spaa2018-FKR-WF_ext_hashing.pdf) – Our implementation is mainly based on the DS described in this paper.

[A Highly-Efficient Wait-Free Universal Construction](http://thalis.cs.uoi.gr/tech_reports/publications/TR2011-01.pdf) – The paper above uses the ideas in this paper to implement the hash-table, we use this for gaining deeper understanding of the ideas behind the DS.

[sim-universal-construction](https://github.com/nkallima/sim-universal-construction) – This repo contains many WF/Lockfree implementations of different data structure among WF DS that are based on the P-sim idea we implement in our DS, we use it for ideas on how to design our implementation. And at a later stage of development we will use those DS as benchmarks to compare to our implementation (Contains P-sim based Stack and Queue).