Chinese Remainder Theorem based Hash Table

(Layered Hash Map)

The hash table data structure wiki:
https://en.wikipedia.org/wiki/Hash_table

The Chinese remainder theorem wiki:
https://en.wikipedia.org/wiki/Chinese_remainder_theorem

Short Explanation:

My goal working on this project was to make a data structure that can be faster and use less space than Hash table that uses separate chaining with linked lists. Hash Tables are useful for quick search and retrieval of objects, given that no collisions have taken place.
But collisions are inevitable. To quote wikipedia:

„...if 2450 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday [paradox] there is approximately a 95% chance of at least two of the keys being hashed to the same slot“

So what should we do if, let's say, we have 10 million keys and 11 million buckets?
We find that the number of keys in collision (overflow) will be:

$$(\frac{11000000-1}{11000000})^{10000000} * 11000000 - (11000000 - 10000000) \approx 3431793$$

So the numbers of keys directly linked to the hash table are 6,568,206.
Doing a similar calculation we get that the number of keys at position 2 of the linked list are 2,169,306
At the position 3 are 798,042
At the position 4 are 293,583
At the position 5 are 108,003
At the position 6 are 39,732 and so on ..

This gives us an average search length of at least 1.53 as well as the maximum of at least 15.
Notice that the maximum search length does not depend on the number of buckets as much as the avg search length does.

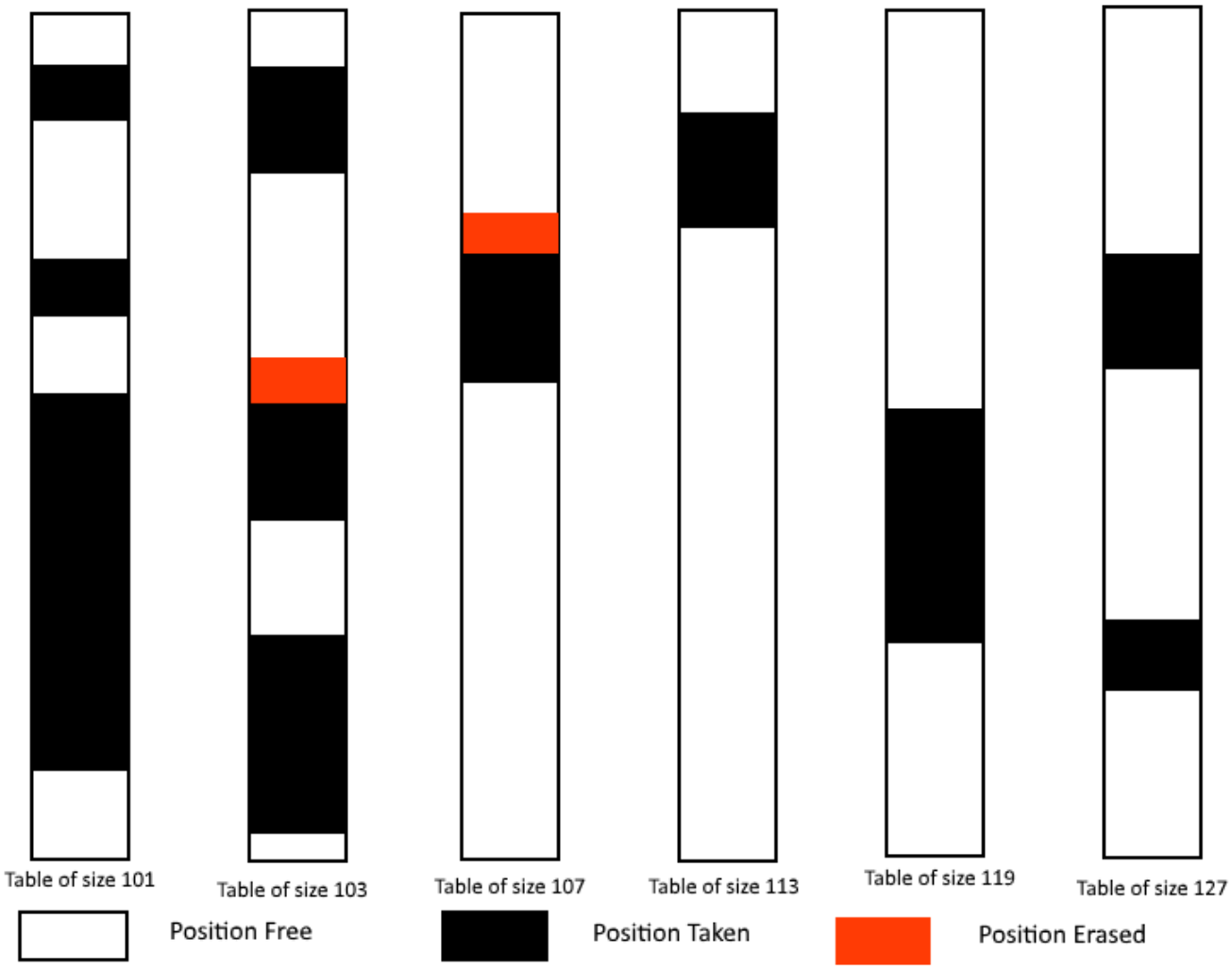
Using the here proposed algorithm we can reduce the speed of the maximum and average hash table at the cost of memory.
The main idea of the algorithm is to use hash tables linked to each other with different number of blocks, each a different prime number.

The structure methods are virtually the same as before, but instead of adding the collisioned key in the linked list we add it to a new Hash Table.

It's also clear that this type of table layering can be used in conjuction with parallelism to get the search length of log(max_search_length) but I'll leave that for some other day.

The CRT::HashMap consists of **n** unique tables layered upon each other in a certain way.

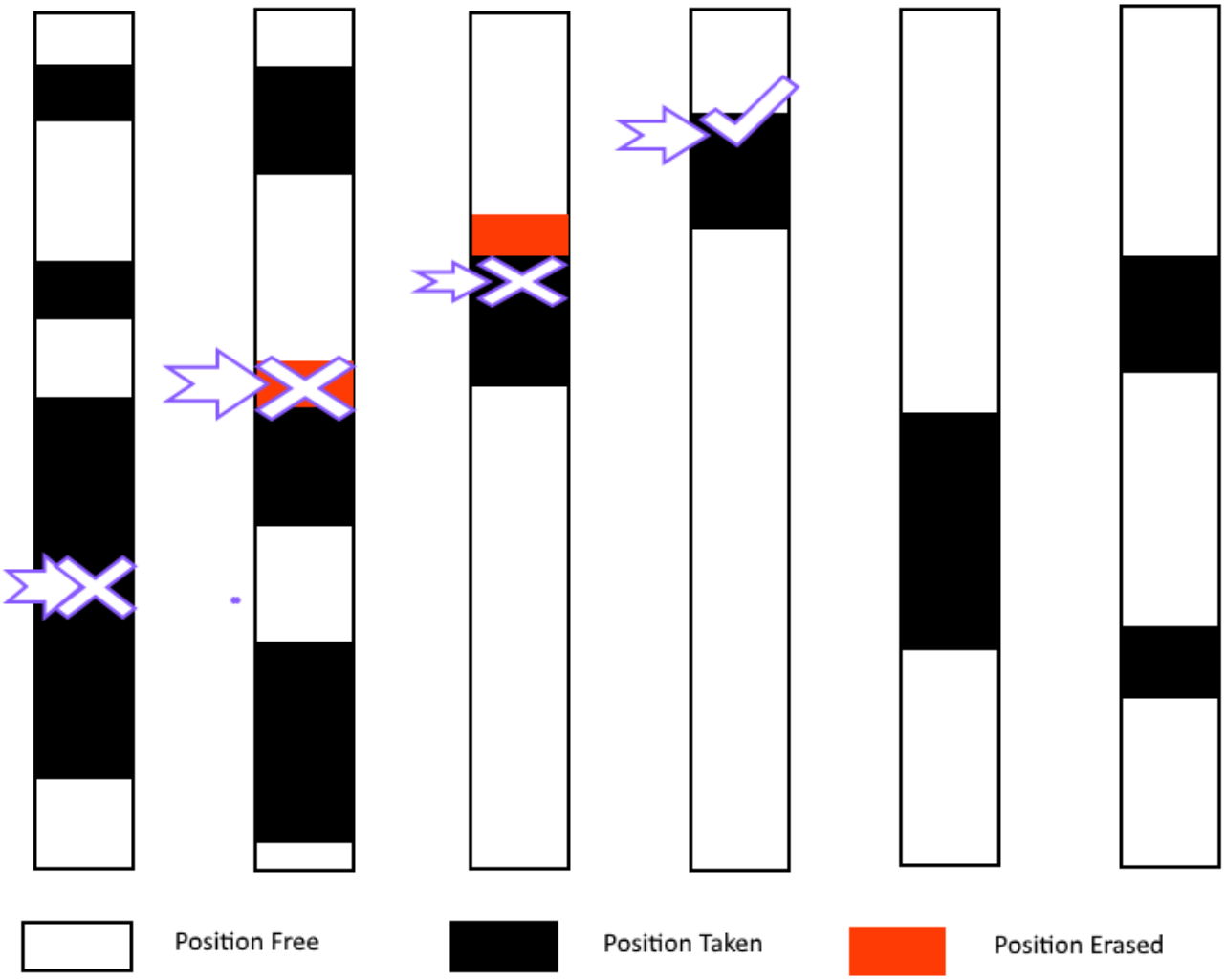
This way we can equalize the number of elements in all the tables greatly reducing the maximum time to reach the certain element as well as reducing memory footprint, as well as the average speed to get and insert a key-value mapping.



Example HashMap with 6 layers of unique tables

The class has 4 methods described below.
Get(Key, &Value)
Put(Key, Value)
Remove(Key)
Clear()

Get(*Key, &Value*)



A simple sketch of algorithm for get method

The method checks if the key is in found the next block, returns true and a value if the key is found. Returns false if key not found.

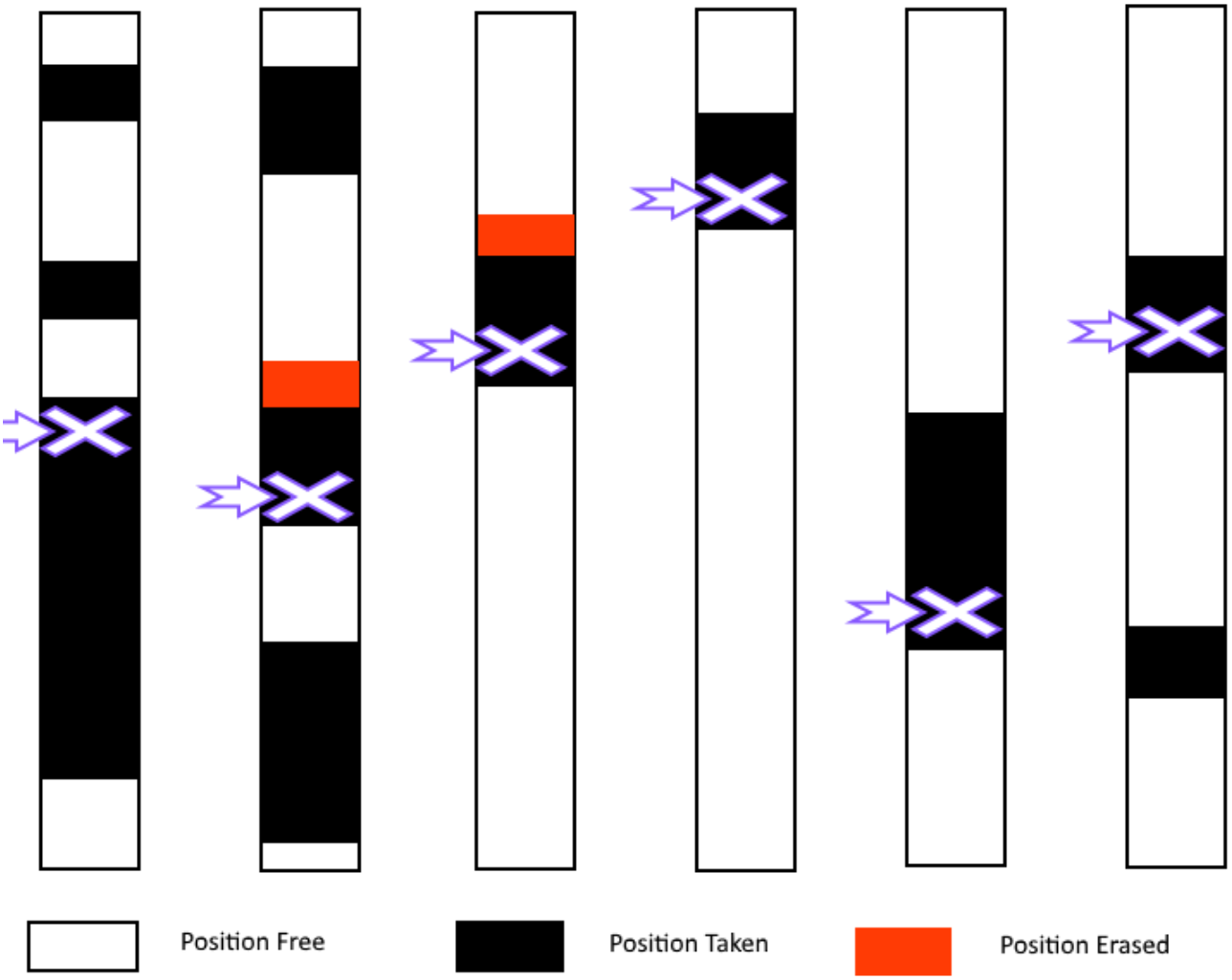
If it encounters the Empty flag returns with false.

Should it encounter flag Erased on it's way, returns the value and moves it to block where it first encountered flag Erased.

Put(*Key, Value*)

Put algorithm is similar to get in it's first run. The algorithm tries all blocks to insert the Key-Value pair.

If any of the position flags are Empty, deleted or have the same key it overwrites the mapping.



Unsuccessful first run of put()

Should the first run be unsuccessful it tries to put all same hash elements of block **{0, 1, ..., i-1}** to block **i** for **1≤i≤n** as second run.

In case second run fails we create a new block with a new unique prime number of elements.

Remove(Key) & Clear()

Remove(Key) is similar to get() but instead of inserting the mapping it deletes it.
Clear removes all block tables and creates a new one of default size.

Birthday Paradox?

Given 10 million keys and a starting block size of 3 million we get a 5 block HashMap.

Similar to having 10 million keys in 15 million buckets with no collision.