INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Hopscotch Hashing

Maurice Herlihy, Nir Shavit, and Moran Tzafrir

DISC '08: Proceedings of the 22nd international symposium on Distributed Computing.
Arcachon, France: Springer-Verlag. pp. 350–364

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Hashing: What and Why?

- Hashing is the process of creating a unique identifier
- Hashing is the practice of taking an input key, a variable created for storing narrative data, and representing it with a hash value.
- Example: ID cards
- Using a hash table is faster than searching through an array

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Key terms

1. Hash table
2. Bucket
3. Key
4. Collision
5. Closed Addressing
6. Linear Probing
7. Cuckoo Hashing
8. Neighbourhood

Simple Hopscotch Hashing

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

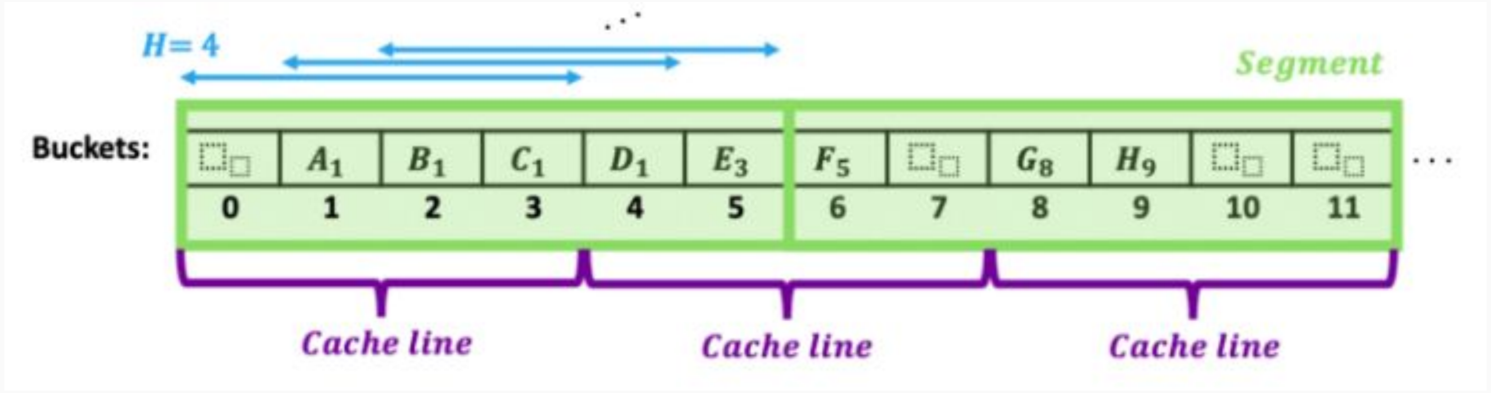
What is it?

Hopscotch hashing is a method of open addressing that builds on the prior work in chained hashing, linear probing and Cuckoo hashing in order to design a new methodology

The key idea of hopscotch hashing is as follows:

- Assign a global neighborhood size H
- keep the hashed key within this neighborhood

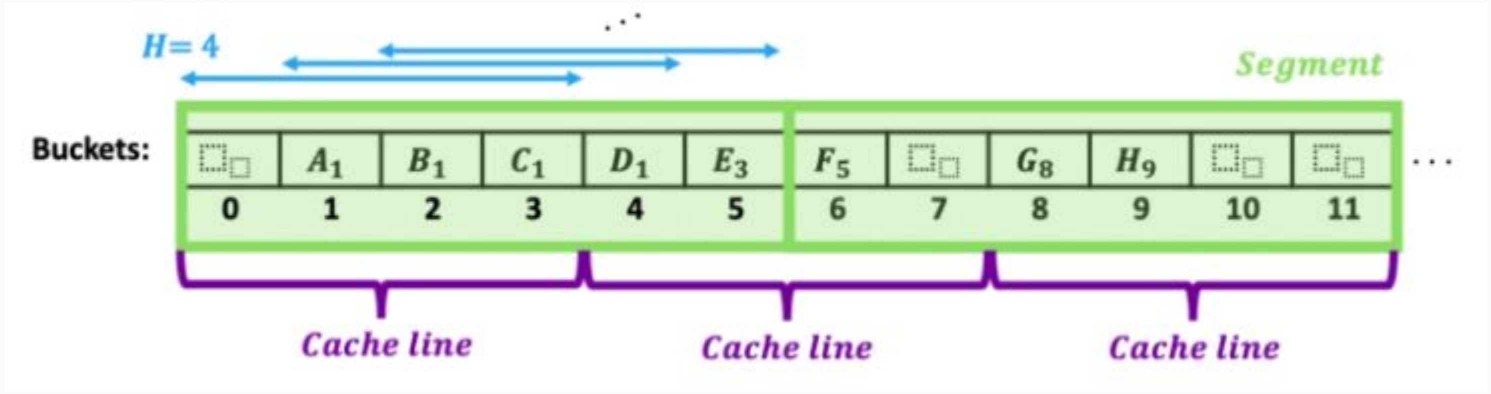
Basic Structure of Hopscotch table



- Each non-empty bucket contains a key, with the subscript denoting the key's home bucket. For D_1, key is D and home bucket index is 1.

- H is the size of the neighborhood, and here, it is 4. Every key whose home bucket is i is guaranteed to be in i's neighborhood.

Basic Structure of Hopscotch table



- For the concurrent version, each segment is protected by a lock, indicated by the green boxes of size 6

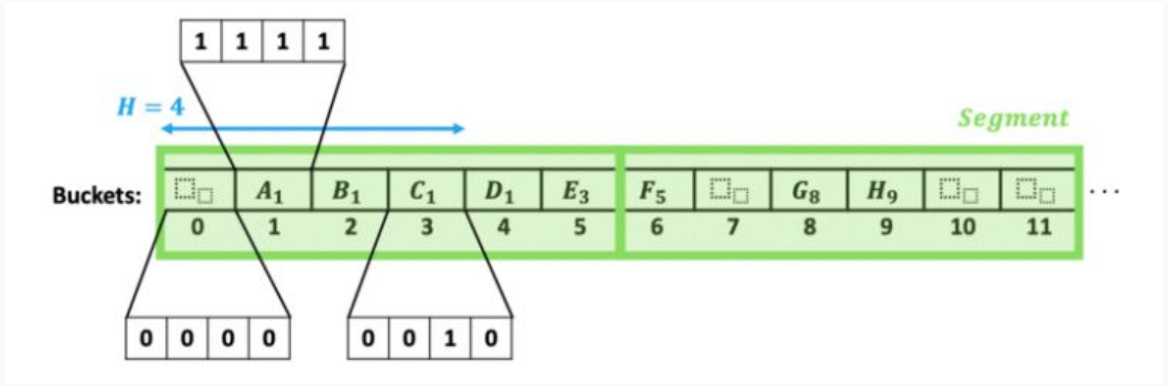
- For a cache-optimized variant of the algorithm, The associated cache lines, which are units of data that are fetched from main memory

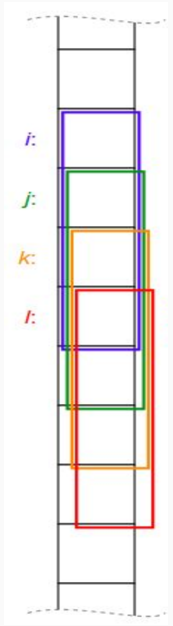
INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Algorithmic overview

1. There is a single hash function **h**
2. Hashed item will always be found either in that home bucket, or in one of the next H – 1 neighboring buckets
3. Each bucket includes some hop-information using a H-bit bitmap, it indicates which of the items in the bucket's neighborhood hashed to the current bucket

Example of hopscotch hashing





- In the above diagram, H=4, bitmap system is applied
- At index 3. The only entry that originally hashed to bucket 3 is E_3 (hence the subscript 3) at bucket 5, which is two buckets away from bucket 3.
- The bitmap associated with bucket 3, 0010 (from left to right), tells us that the neighbourhood binaries of original hash
- The map shows 3, 4, and 6 does not have original hashed values to 3 but 5 has it

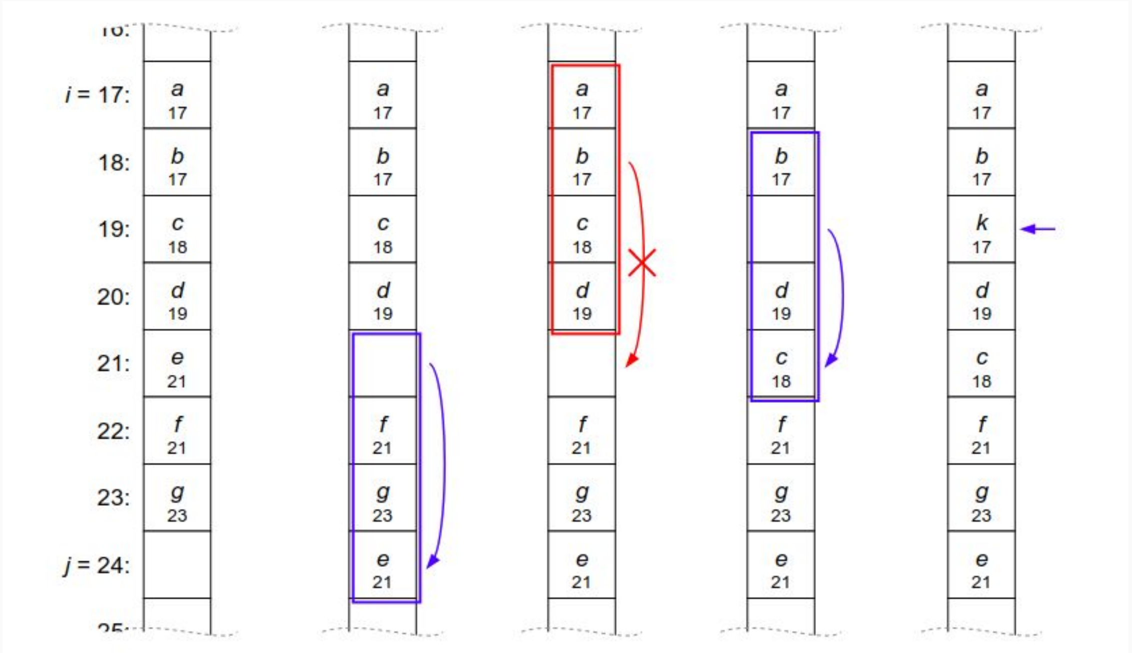
Function: add()

Suppose a key, k, is to be inserted, and that it hashes to bucket i. Bucket i is referred to as the home bucket of k.

Cases for home bucket:

- Is empty
- Is NOT empty

worst-case complexity is O(n)



INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY HYDERABAD

Function: contains(k), remove(k), resize(k)

For Lookup/contains:

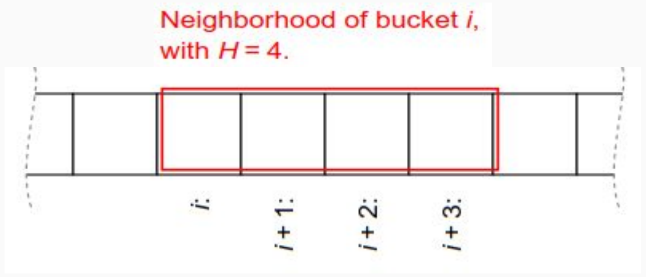
1. Every key is guaranteed to be found in the assumed H
2. Complexity is O(1)

For Remove:

1. Do Lookup
2. Update the bitmap (no need for lazy deletion)
3. Complexity is O(1)

For Resize:

1. Not specified
2. when the sequence of displacements cannot be performed
3. Can increase size according to memory constraints
4. Can resize the neighborhood size H, with a slight tradeoff in speed



INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Cache Locality

- Hopscotch hashing has super fast query times in practice is because of its excellent cache locality
- Complexity for find operation is O(1)
- Setting neighborhood size to be equal to the size of the cache line means there will be at most 2 cache misses before an object is found

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

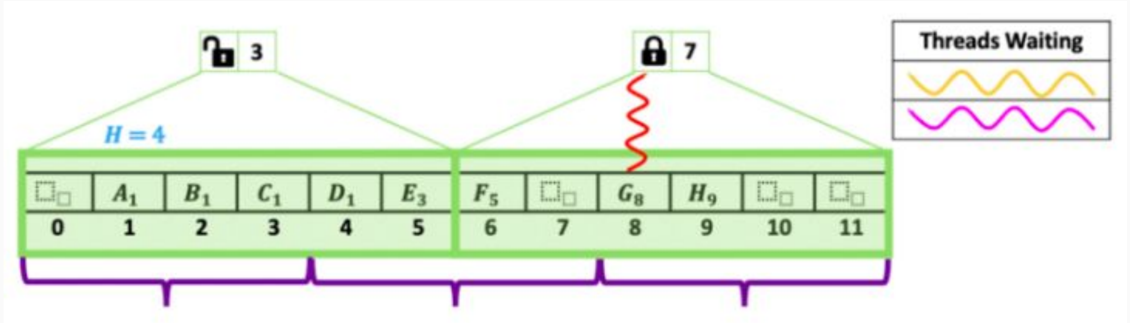
Concurrent Hopscotch Hashing

Concept

We will group buckets together into segments

Each segment will also contain its own lock and timestamp

Each time a segment is modified, that segment's timestamp is incremented by 1



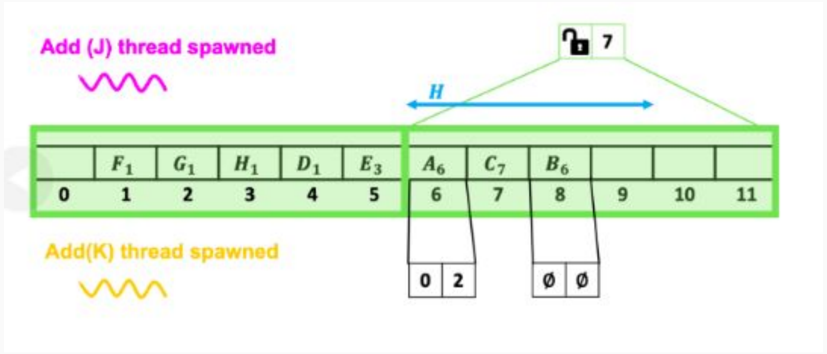
Concurrent hopscotch hash table outline

1. there cannot be any race conditions in the accessing or modification of the timestamp
2. it is possible for the contains operation to loop infinitely

Function: add()

The add operation is identical to the sequential add(k) except for two key differences:

- While traversing the hash table the relevant segment must be locked. This ensures that there are no race conditions during multiple concurrent add/remove calls to the hash table.
- Before returning, the add(k) function will increment the timestamp of the relevant segment and release the lock. The time stamp update is necessary to ensure that concurrent contains(k) re-check the updated hash table.
- worst-case complexity is O(n)



INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Function: contains(k), remove(k), resize(k)

For Lookup/contains:

1. Perform simple contain()
2. Check if timestamp has changed

For Remove:

1. Do Lookup
2. Acquire lock
3. when a key is found an *optimizeCacheline* operation is called

For Resize:

1. Not specified
2. when the sequence of displacements cannot be performed
3. Can increase size according to memory constraints
4. Can resize the neighborhood size H, with a slight tradeoff in speed

Comparison: Round Robin Hashing

Each index in a hopscotch hashing array tracks its neighborhood and its key’s home index, and each index in a Robin Hood hashing array tracks its key’s distance from home.

Hopscotch hashing swaps keys until each key is in the neighborhood of its home index, while Robin Hood hashing swaps keys when the key to insert is further from home than the current key.

Although Robin Hood hashing tends to have worse worst-case runtimes, in practice, variants of Robin Hood hashing are very competitive with hopscotch hashing.

Comparison: Round Robin Hashing

Hopscotch hashing is slightly more cache friendly than Robin Hood variants, but requires more complicated operations and implementations.

Hopscotch hashing outperforms Robin Hood hashing in workflows with high thread counts.

Robin Hood hashing outperforms Hopscotch hashing in update-heavy workflows or workflows with high load factors.

All in all, Robin Hood and hopscotch are two of the most effective hashing algorithms with their own merits depending on the workflow and memory limitations.

Future work

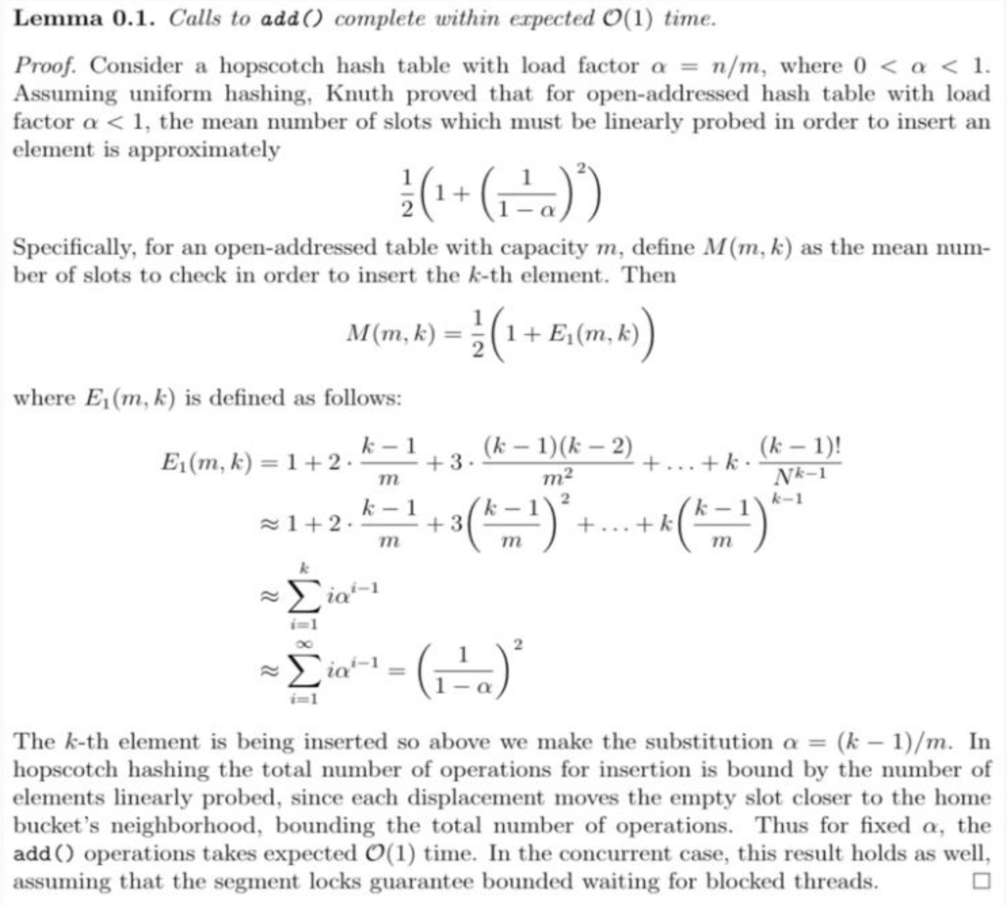
As discussed previously, hopscotch hashing has been shown to be efficient in theory and practice. Most of the ongoing research regarding hopscotch hashing focuses on the concurrent data structure. Specifically, the research is focused on answering how best to deal with contention.

As a reminder, contention is a common issue that arises in concurrent programs. It occurs when two threads attempt to access the same resource at the same time.

The implementation I will outline attempts to resolve this conflict via the use of locks.

Alternative strategies include lock-free implementations and the use of transactional memory.

Appendix: Runtime analysis



Lemma 0.1. *Calls to* **`add()`** *complete within expected $\mathcal{O}(1)$ time.*

Proof. Consider a hopscotch hash table with load factor $\alpha = n/m$, where $0 < \alpha < 1$. Assuming uniform hashing, Knuth proved that for open-addressed hash table with load factor $\alpha < 1$, the mean number of slots which must be linearly probed in order to insert an element is approximately

$$\frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$$

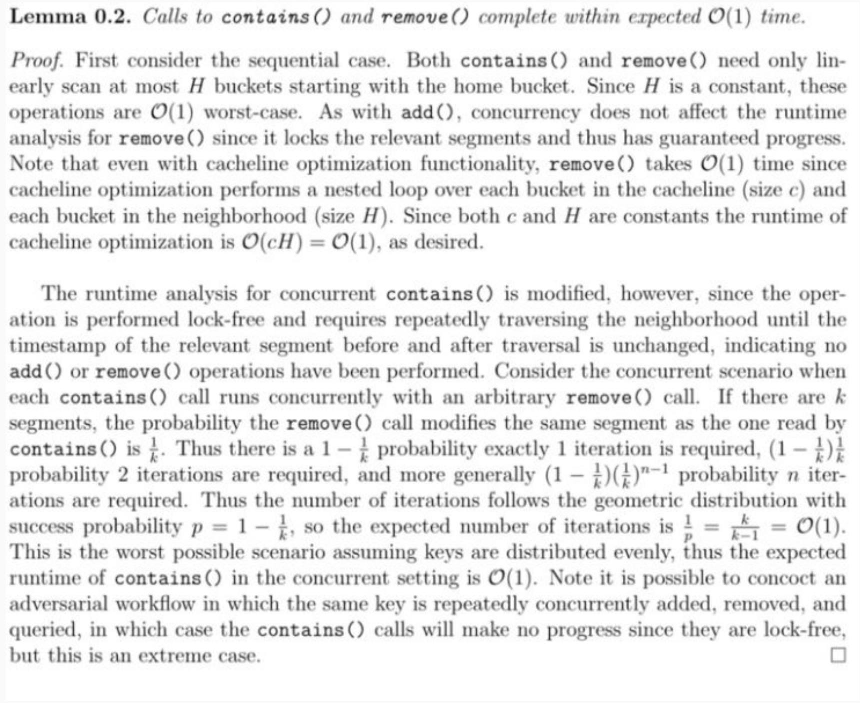
Specifically, for an open-addressed table with capacity m, define $M(m,k)$ as the mean number of slots to check in order to insert the k-th element. Then

$$M(m,k) = \frac{1}{2}\Big(1 + E_1(m,k)\Big)$$

where $E_1(m,k)$ is defined as follows:

$$\begin{aligned} E_1(m,k) &= 1 + 2\cdot\frac{k-1}{m} + 3\cdot\frac{(k-1)(k-2)}{m^2} + \ldots + k\cdot\frac{(k-1)!}{N^{k-1}} \\ &\approx 1 + 2\cdot\frac{k-1}{m} + 3\left(\frac{k-1}{m}\right)^2 + \ldots + k\left(\frac{k-1}{m}\right)^{k-1} \\ &\approx \sum_{i=1}^{k} i\alpha^{i-1} \\ &\approx \sum_{i=1}^{\infty} i\alpha^{i-1} = \left(\frac{1}{1-\alpha}\right)^2 \end{aligned}$$

The k-th element is being inserted so above we make the substitution $\alpha = (k-1)/m$. In hopscotch hashing the total number of operations for insertion is bound by the number of elements linearly probed, since each displacement moves the empty slot closer to the home bucket's neighborhood, bounding the total number of operations. Thus for fixed α, the `add()` operations takes expected $\mathcal{O}(1)$ time. In the concurrent case, this result holds as well, assuming that the segment locks guarantee bounded waiting for blocked threads. □

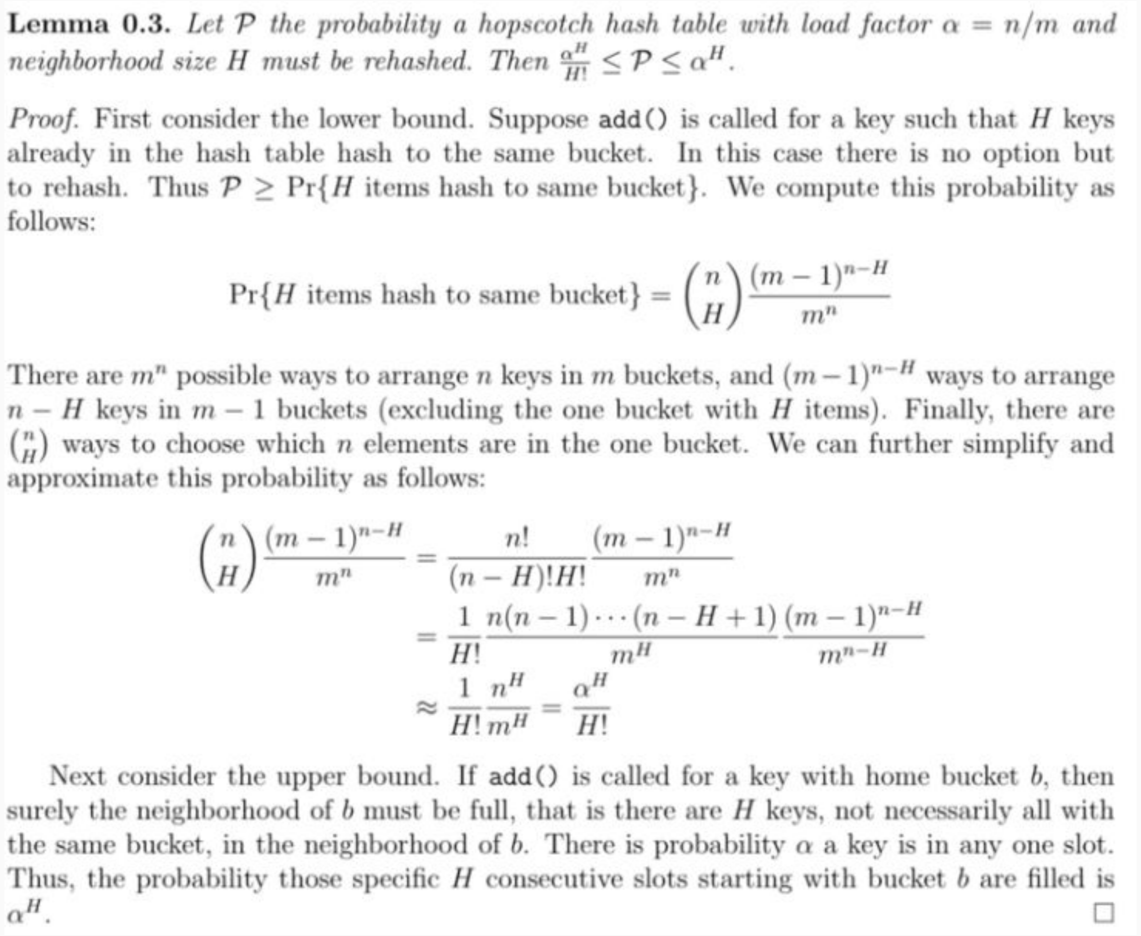


Lemma 0.2. *Calls to* **`contains()`** *and* **`remove()`** *complete within expected $\mathcal{O}(1)$ time.*

Proof. First consider the sequential case. Both `contains()` and `remove()` need only linearly scan at most H buckets starting with the home bucket. Since H is a constant, these operations are $\mathcal{O}(1)$ worst-case. As with `add()`, concurrency does not affect the runtime analysis for `remove()` since it locks the relevant segments and thus has guaranteed progress. Note that even with cacheline optimization functionality, `remove()` takes $\mathcal{O}(1)$ time since cacheline optimization performs a nested loop over each bucket in the cacheline (size c) and each bucket in the neighborhood (size H). Since both c and H are constants the runtime of cacheline optimization is $\mathcal{O}(cH) = \mathcal{O}(1)$, as desired.

The runtime analysis for concurrent `contains()` is modified, however, since the operation is performed lock-free and requires repeatedly traversing the neighborhood until the timestamp of the relevant segment before and after traversal is unchanged, indicating no `add()` or `remove()` operations have been performed. Consider the concurrent scenario when each `contains()` call runs concurrently with an arbitrary `remove()` call. If there are k segments, the probability the `remove()` call modifies the same segment as the one read by `contains()` is $\frac{1}{k}$. Thus there is a $1 - \frac{1}{k}$ probability exactly 1 iteration is required, $(1-\frac{1}{k})\frac{1}{k}$ probability 2 iterations are required, and more generally $(1-\frac{1}{k})(\frac{1}{k})^{n-1}$ probability n iterations are required. Thus the number of iterations follows the geometric distribution with success probability $p = 1 - \frac{1}{k}$, so the expected number of iterations is $\frac{1}{p} = \frac{k}{k-1} = \mathcal{O}(1)$. This is the worst possible scenario assuming keys are distributed evenly, thus the expected runtime of `contains()` in the concurrent setting is $\mathcal{O}(1)$. Note it is possible to concoct an adversarial workflow in which the same key is repeatedly concurrently added, removed, and queried, in which case the `contains()` calls will make no progress since they are lock-free, but this is an extreme case. □

Appendix: Probability of resize



Lemma 0.3. *Let $\mathcal{P}$ the probability a hopscotch hash table with load factor $\alpha = n/m$ and neighborhood size H must be rehashed. Then $\frac{\alpha^H}{H!} \le \mathcal{P} \le \alpha^H$.*

Proof. First consider the lower bound. Suppose `add()` is called for a key such that H keys already in the hash table hash to the same bucket. In this case there is no option but to rehash. Thus $\mathcal{P} \ge \Pr\{H \text{ items hash to same bucket}\}$. We compute this probability as follows:

$$\Pr\{H \text{ items hash to same bucket}\} = \binom{n}{H}\frac{(m-1)^{n-H}}{m^n}$$

There are m^n possible ways to arrange n keys in m buckets, and $(m-1)^{n-H}$ ways to arrange $n-H$ keys in $m-1$ buckets (excluding the one bucket with H items). Finally, there are $\binom{n}{H}$ ways to choose which n elements are in the one bucket. We can further simplify and approximate this probability as follows:

$$\begin{aligned}
\binom{n}{H}\frac{(m-1)^{n-H}}{m^n} &= \frac{n!}{(n-H)!H!}\frac{(m-1)^{n-H}}{m^n} \\
&= \frac{1}{H!}\frac{n(n-1)\cdots(n-H+1)}{m^H}\frac{(m-1)^{n-H}}{m^{n-H}} \\
&\approx \frac{1}{H!}\frac{n^H}{m^H} = \frac{\alpha^H}{H!}
\end{aligned}$$

Next consider the upper bound. If `add()` is called for a key with home bucket b, then surely the neighborhood of b must be full, that is there are H keys, not necessarily all with the same bucket, in the neighborhood of b. There is probability α a key is in any one slot. Thus, the probability those specific H consecutive slots starting with bucket b are filled is α^H. $\square$

Thanks!

Sridhar M
2018111021
sridhar.m@research.iiit.ac.in