Lecture 24 — Hash Tables

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

Lectured by Margaret Reid-Miller — 23 April 2013

Today:

- Hashing
- Hash Tables

1 Hashing

hash: transitive verb[1]

1. (a) to chop (as meat and potatoes) into small pieces
   (b) confuse, muddle
2. ...

This is the definition of hash from which the computer term was derived. The idea of hashing as originally conceived was to take values and to chop and mix them to the point that the original values are muddled. The term as used in computer science dates back to the early 1950's.

More formally the idea of hashing is to approximate a random function $h : \alpha \to \beta$ from a source (or universe) set U of type α to a destination set of type β. Most often the source set is significantly larger than the destination set, so the function not only chops and mixes but also reduces. In fact the source set might have infinite size, such as all character strings, while the destination set always has finite size. Also the source set might consist of complicated elements, such as the set of all directed graphs, while the destination are typically the integers in some fixed range. Hash functions are therefore many to one functions.

Using an actual randomly selected function from the set of all functions from α to β is typically not practical due to the number of such functions and hence the size (number of bits) needed to represent such a function. Therefore in practice one uses some pseudorandom function.

Exercise 1. *How many hash functions are there that map from a source set of size n to the integers from 1 to m? How many bits does it take to represent them? What if the source set consists of character strings up length up to 20. Assume there are 100 possible characters.*

Why is it useful to have random or pseudo random functions that map from some large set to a smaller set? Generally such functions might be used to hide data, to reorder data in a random order, or to spread data smoothly over a range. Here we consider some applications of each.

†Lecture notes by Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

[1] Merriam Websters

1. We saw how hashing can be used in treaps. In particular we suggested using a hash function to hash the keys to generate the "random" priorities. Here what was important is that the ordering of the priorities is somehow random with respect to the keys. Our analysis assumed the priorities were truly random, but it can be shown that a limited form of randomness that arise out of relatively simple hash functions is sufficient.

2. In cryptography hash functions can be used to hide information. One such applications is in digital signatures where a so-called secure hash is used to describe a large document with a small number of bits.

3. A one-way hash function is used to hide a string, for example for password protection. Instead of storing passwords in plain text, only the hash of the password is stored. To verify whether a password entered is correct, the hash of the password is compared to the stored value. These signatures can be use to *authenticate* the source of the document, ensures the *integrity* of the document as any change to the document invalidates the signature, and prevents *repudiation* where the sender denies signing the document.

4. String commitment protocols use hash functions to hide to what string a sender has committed so that the receiver gets no information. Later, the sender sends a key that allows the receiver to reveal the string. In this way, the sender cannot change the string once it is committed, and the receiver can verify the revealed string is the committed string. Such protocols might be used to flip a coin across the internet: The sender flips a coin and commits the result. In the mean time the receiver calls heads or tails, and the sender then sends the key so the receiver can reveal the coin flip.

5. Hashing can be used to approximately match documents, or even parts of documents. *Fuzzy matching* hashes overlapping parts of a document and if enough of the hashes match, then it conclude that two documents are approximately the same. Big search engines look for similar documents so that on search result pages they don't show the many slight variations of the same document (e.g., in different formats). It is also used in spam detection, as spammers make slight changes to email to bypass spam filters or to push up a document's content rank on a search results page. When looking for malware, fuzzy hashing can quickly check if code is similar to known malware. Geneticists use it to compare sequences of genes fragments with a known sequence of a related organism as a way to assemble the fragments into a reasonably accurate genome.

6. Hashing is used to implement hash tables. In hash tables one is given a set of keys $K \subset \alpha$ and needs to map them to a range of integers so they can stored at those locations in an array. The goal is to spread the keys out across the integers as well as possible to minimize the number of keys that collide in the array. You should not confuse the terms hash function and hash table. They are two separate ideas, the later that uses the prior.

There is a deep and interesting theory of hash functions. Depending on the application, the needs of the hash function are very different. We will not cover the theory here but you will likely see it in more advanced algorithms classes.

For hash table applications a hash function should have the following properties:

- It should distribute the keys evenly so there are not many collisions.

- It should be fast to compute.

So what are some reasonable hash functions. Here we consider some simple ones. For hashing integers we can use

$$h(x) = (ax + b) \bmod p$$

where $a \in [1, \ldots, p-1]$, $b \in [0, \ldots, p-1]$, and p is a prime. This is called a linear congruential hash function has some nice properties that you will likely learn about in 15-451.

For strings we can simply use a polynomial

$$h(S) = \left(\sum_{i=1}^{|S|} s_i a^i \right) \bmod p$$

Sequentially, Horner's method avoids computing a^i explicitly. In parallel, simply use scan with multiplication. This hash function tends to mix bits from earlier characters with bits in later characters.

In our analysis we will assume that we have hash functions with the following idealized property *simple uniform hashing*: The hash function uniformly distributes the n keys over the range $[0, \ldots, m-1]$ and the hash value for any key is independent of the hash value for any other key.

2 Hash Tables

Hash tables are used when an application needs to maintain a dynamic set where the main operations are `insert`, `find` and `delete`. Hash table can implement the abstract data types `Set` and `Table`. Unlike binary search trees, which require the universe of keys has a total ordering, hash tables do not. A total ordering enables the additional ordering operations provided by the Ordered Set abstract data type.

The main issue with hash table is collisions, where two keys hash to the same location. Is it possible to avoid collisions? Not if we don't know the set of keys in advance. Since the size of the table T is much smaller than the universe of keys U, $|T| << |U|$, there must exist at least two keys that map to the same index, by the *Pigeonhole Principle*: If you put more than m items into m bins, then at least one bin contains more than one item. For a particular hash function, the subset of keys $K \subset U$ that we want to hash may or may not cause a collision even when the number of keys is much smaller than the size of the table. Therefore, for general purpose dynamic hash tables we have to assume that collisions will occur.

How likely is there at least one collision? This is the same question as the birthday paradox: When there a n or more people in a room, what is the chance that two people have the same birthday. It turns out that for a table of size 365 you need only 23 keys for a 50% chance of a collision, and as little as 60 keys for a 99% chance. More interesting, when hashing to m locations, you can expect a collision after only $\sqrt{\frac{1}{2}\pi\, m}$ insertions, and can expect every location in the table has an element

after $\Theta(m \log m)$ insertions. The former is related to the *birthday paradox*, whereas the later is relate to the *coupon collector* problem.

There are several well-studied collision resolution strategies:

- **Separate chaining**: Store elements not in a table, but in linked lists (containers,bins) hanging off the table.
- **Open addressing**: Put everything into the table, but not necessarily into cell $h(k)$.
- **The perfect hash**: When you know the keys in advance, construct hash functions that avoids collisions entirely.
- **Multiple-choice hashing and Cuckoo hashing**: Consider exactly two locations $h_1(k)$ and $h_2(k)$ only.

We will consider the first two in this lecture.

In our discussion we will assume we have a set of n keys K that we want to store and a hash function $h : \texttt{key} \to [1, \ldots, m]$ for some m.

2.1 Separate Chaining

In 15-122 you covered hash tables using separate chaining. As you may recall, the idea is to maintain an array of linked lists. All keys that hashes to the same location in the sequence are maintained in a linked list. When inserting a key k, it inserts it at the beginning of the linked list at location $h(k)$. But if the application may attempt to insert the same key multiple times, separate chaining needs to search down the list to check whether the key is already in the list, in which case it might just as well add the key to the end of the linked list. To find a key, simply search for key in the linked list at location $h(k)$. To delete a key, remove it from the linked list.

The costs of these operations is related to the average length of a chain, which is n/m when there n keys in a table with m chains. We call $\lambda = n/m$ the *load factor* of the table.

We consider two cases when that can occur when applying these operations: an unsuccessful search when the key is not in the table, and a successful search when the key is in the table. We assume that we can compute the $h(k)$ in $O(1)$ work.

Claim 2.1. *Assuming simple uniform hashing, an* unsuccessful search *takes expected* $\Theta(1 + \lambda)$ *work.*

Proof. The average length of a list is λ. If we search for a key that is not in the table, we need to search the whole list to determine the key is not in the table. Including the cost of computing $h(k)$, the total work is $\Theta(1 + \lambda)$. □

Claim 2.2. *Assuming simple uniform hashing, a successful* successful search *takes expected* $\Theta(1 + \lambda)$ *work.*

Proof. To simplify the analysis, we assume that keys are added at the end of the chain[2]. The the cost of a successful search is the same as an unsuccessful search at the time the key was added to the table. That is, when we first insert a key, the cost to insert it is the same as the cost of an unsuccessful search. Suppose this cost is T_k. Subsequent searches for this key is also T_k. When the table has i keys, the cost of inserting a new key is expected to be $(1 + i/m)$. Averaging over all keys, the cost of a successful search is

$$\frac{1}{n}\sum_{i=0}^{n-1}(1 + i/m) = 1 + (n-1)/2m \leq 1 + \lambda/2 = \Theta(1 + \lambda)$$

□

That is, for successful searches we examine half the list on average and unsuccessful searches the full list. If $n = O(m)$ then with simple uniform hashing, all operations have expected $O(1)$ work and span. Even more important, some chains can be long, $O(n)$ in the worst case, but it is extremely unlikely they will more than a small constant factor of λ. The advantage of separate chaining is that it is not particularly sensitive to the size of the table. If the number keys is more than anticipated, the cost of search becomes only somewhat worse. If the number is less, then only a small amount of space in the table is wasted and cost of search is faster.

2.2 Open Address Hash Tables

The next technique does not need any linked lists but instead stores every key directly in the array. Open address hashing using so called linear probing has an important practical advantage over separate chaining: it causes fewer cache misses since typically all locations that are checked are on the same cache line.

The basic idea of open addressing is to maintain an array that is some constant factor larger than the number of keys and to store all keys directly in this array. Every cell in the array is either empty or contains a key.

To decide to which cells to assign a key, open addressing uses an ordered sequence of locations in which the key can be stored. In particular let's assume we have a function $h(k, i)$ that returns the i^{th} location for key k. We refer to the sequence $\langle h(k,0), h(k,1), h(k,2), \ldots \rangle$ as the *probe* sequence. We will get back to how the probe sequence might be assigned, but let's first go through how these sequences are used. When inserting a key the basic idea is to try each of the locations in order until it finds a cell that is empty, and then insert the key at that location. Sequentially insert would look like

```
1  function insert(T, k) =
2  let
3    function insert'(T, k, i) =
4      case nth T h(k, i) of
5        NONE ⇒ update(h(k, i), k) T
6      | _ ⇒ insert'(T, k, i + 1)
7  in
8    insert'(T, k, 1)
9  end
```

[2]The average successful search time is the same whether new keys are added to the front of the end of the chain

For example suppose the hash table has the following keys:

	0	1	2	3	4	5	6	7
T =		B			E	A		F

Now if for a key D we had the probe sequence $\langle 1, 5, 3, \cdots \rangle$, then we would find location 1 and 5 full (with B and E) and place D in location 3 giving:

	0	1	2	3	4	5	6	7
T =		B		D	E	A		F

Note that, in order for the update operation to be constant work and span, T must be a single threaded array. Also, the code will loop forever if all locations are full. Such an infinite loop can be prevented by ensuring that $h(k,i)$ tries every location as i is incremented, and checking when the table is full. Also, as given the code will insert the same key multiple times over. This problem is easily corrected by checking if the key is in the table and if so returning immediately.

To search we have the following code:

```
1  function find(T, k) =
2  let
3    function find'(T, k, i) =
4      case T[h(k, i)] of
5        NONE ⇒ false
6      | SOME(k') ⇒ if (eq(k, k')) then true
7                   else find'(T, k, i + 1)
8  in
9    find'(T, k, 1)
10 end
```

For example in the table above, if key E has the probe sequence $\langle 7, 4, 2, \cdots \rangle$, `find` would first search location 7, which is full, and then location 4 to find E.

What if we want to delete a key? Let's say we deleted A from the table above, and then searched for D. Will `find` locate it? No, it will stop looking once it finds the empty cell where A was. One solution might be to rehash everything after a delete. But that would be an extremely expensive operation for every delete. An alternative is to use what is called a *lazy delete*. Instead of deleting the key, simply replace the key with a special `HOLD` value. That is, introduce an entry data type:

```
1  datatype α entry = EMPTY | HOLD | FULL of α
```

Of course, we will need to change `find` and `insert` to handle `HOLD` entries. For `find`, simply skip over a `HOLD` entry and move to the next probe. Whereas, `insert`(v) can replace the first `HOLD` entry with `FULL`(v). But if `insert` needs to check for duplicate keys, it first needs to search for the key. If it finds the key it overwrites it with the new value. Otherwise it continues until it finds an empty cell, at which point it can replace the first `HOLD` in the probe sequence.

The main concern with lazy deletes is that they effectively increases the load factor, increasing the cost of the hash table operations. If the load factor becomes large and performance degrades, the

solution is to rehash everything to a new larger table. The table should be a constant fraction larger each time the table grow so as to maintain amortized constant costs.

Now let's consider some possible probe sequences we can use. Ideally, we would like a key to use any of the possible $m!$ probe sequences with equal probability. This ideal is called *uniform hashing* But uniform hashing is not practical. Common probe sequences, which we will consider next, are

- linear probing
- quadratic probing
- double hashing

2.2.1 Linear Probing

In linear probing, to insert a key k, it first checks $h(k)$ and then checks each following location in succession, wrapping around as necessary, until it finds an empty location. That is, the i^{th} probe is

$$h(k,i) = (h(k) + i) \mod m.$$

Each position determines a single probe sequence, so there are only m possible probe sequences.

The problem with linear probing is that keys tend to cluster. It suffers from *primary clustering*: Any key that hashes to any position in a cluster (not just collisions), must probe beyond the cluster and adds to the cluster size. Worse yet, primary clustering not only makes the probe sequence longer, it also makes it more likely that it will be lengthen further.

What is the impact of cluster for unsuccessful search? Let's consider two extreme examples when the table is half full, $\lambda = 1/2$. Clustering is minimized when every other location in the table is empty. In this case, the average number of probes needed to insert a new key k is $3/2$: One probe to check cell $h(k)$, and with probability $1/2$ that cell is full and it needs to look at the next location which, by construction, must be empty. In the worst case, all the keys are clustered, let's say at the end of the table. If k hashes to any of the first n locations, only one probe is needed. But hashing to the n^{th} location would require probing all n full locations before finally wrapping around to find an empty location. Similarly, hashing to the second full cell, requires probing $(n-1)$ full cells plus the first empty cell, and so forth. Thus, under uniform hashing the average number of probes needed to insert a key would be

$$1 + [n + (n-1) + (n-2) + ... + 1]/m = 1 + n(n+1)/2m \approx n/4$$

Even though the average cluster length is 2, the cost for an unsuccessful search is $n/4$. In general, each cluster j of length n_j contributes $n_j(n_j+1)/2$ towards the total number of probes for all keys. Its contribution to the average is proportional the *squares* of the length of the cluster, making long cluster costly.

We won't attempt to analyze the cost of successful and unsuccessful searches, as considering cluster formation during linear probing is quite difficult. We make the following claim:

Claim 2.3. *When using linear probing in a hash table of size m that contains $n = \lambda m$ keys, the average number of probes needed for an unsuccessful search or an insert is*

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

and for a successful search is

$$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right).$$

As you can see from the following table, which show the expected number of probes under uniform hashing, linear probing degrades significantly when the load factor increases:

λ	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.5	2.0	3.0	5.5
unsuccessful	1.4	2.5	5.0	8.5	55.5

Linear probing is quite competitive, though, when the load factors are in the range (30-70%) as clusters tend to stay small. In addition, a few extra probes is mitigated when sequential access is much faster than random access, as is the case of caching. Because of primary clustering, though, it is sensitive to quality of the hash function or the particular mix of keys that result in many collisions or clumping. Therefore, it may not be a good choice for a general purpose hash tables.

2.2.2 Quadratic Probing

Quadratic probe sequences causes probes to move away from cluster, by making increasing larger jumps. The i^{th} probe is

$$h(k,i) = (h(k) + i^2) \mod m.$$

Although, quadratic probing voids primary clustering, it still has *secondary clustering*: When two keys hash to the same location, they have the same probe sequence. Since there are only m locations in the table, there are only m possible probe sequences.

One problem with quadratic probing is that probe sequences do not probe all locations in the table. But since there are $(p+1)/2$ quadratic residues when p is prime, we can make the following guarantee.

Claim 2.4. *: If m is prime and the table is at least half empty, then quadratic probing will always find an empty location. Furthermore, no locations are checked twice.*

Proof. (by contradiction) Consider two probe locations $h(k) + i^2$ and $h(k) + j^2, 0 \le i, j < \lceil m/2 \rceil$. Suppose the locations are the same but $i \neq j$. Then

$$\begin{aligned} h(k) + i^2 &\equiv (h(k) + j^2) \mod m \\ i^2 &\equiv j^2 \mod m \\ i^2 - j^2 &\equiv 0 \mod m \\ (i-j)(i+j) &\equiv 0 \mod m \end{aligned}$$

Therefore, either $i - j$ or $i + j$ are divisible by m. But since both $i - j$ and $i + j$ are less than m and m is prime, they cannot be divisible by m. Contradiction.

Thus the first $\lceil m/2 \rceil$ probes are distinct and guaranteed to find an empty location. □

Computing the next probe is only slightly more expensive than linear probing as it can be computed without using multiplication:

$$\begin{aligned} h_i - h_{i-1} &\equiv (i^2 - (i-1)^2) \mod m \\ h_i &\equiv (h_{i-1} + 2i - 1) \mod m \end{aligned}$$

Unfortunately, requiring that the table remains less than half full makes quadratic probing space inefficient.

2.2.3 Double Hashing

Double hashing uses two hash functions, one to find the initial location to place the key and a second to determine the size of the jumps in the probe sequence. The i^{th} probe is

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m.$$

Keys that hash to the same location, are likely to hash to a different jump size, and so will have different probe sequences. Thus, double hashing avoids secondary clustering by providing as many as m^2 probe sequences.

How do we ensure every location is checked? Since each successive probe is offset by $h_2(k)$, every cell is probed if $h_2(k)$ is relatively prime to m. Two possible ways to ensure $h_2(k)$ is relatively prime to m are, either make $m = 2^k$ and design $h_2(k)$ so it is always odd, or make m prime and ensure $h_2(k) < m$. Of course, $h_2(k)$ cannot equal zero.

Double hashing behaves quite closely to uniform hashing for careful choices of h_1 and h_2. Under uniform hashing the average number of probes for an unsuccessful search or an insert is at most

$$1 + \lambda + \lambda^2 + \ldots = \left(\frac{1}{1-\lambda}\right)$$

and for a successful search is at most

$$\frac{1}{\lambda}\left(1 + \ln\left(\frac{1}{1-\lambda}\right)\right).$$

The former bound is because the probability of needing more than i probes is at most λ^i. A search always needs one probe, and with probability λ needs a second probe, and with probability λ^2 needs a third probe, and so on. The bound for a successful search for a key k follows the same probe sequences as when it was first inserted. So if k was the $(j+1)^{th}$ key inserted the cost for inserting it

is at most $1/(1-j/m)$. Therefore the average cost of a successful search is at most

$$\begin{aligned}\frac{1}{n}\sum_{j=0}^{n-1}\frac{1}{1-j/m} &= \frac{m}{n}\sum_{j=0}^{n-1}\frac{1}{m-j}\\ &= \tfrac{1}{\lambda}(H_m - H_{m-n})\\ &\leq \tfrac{1}{\lambda}(\ln m + 1 - \ln(m-n))\\ &= \tfrac{1}{\lambda}\left(1+\ln\left(\tfrac{1}{1-\lambda}\right)\right)\end{aligned}$$

The table below shows the expected number of probes under the assumption of uniform hashing and is the best one can expect by open addressing.

λ	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.4	1.6	1.8	2.6
unsuccessful	1.3	1.5	2.0	3.0	5.5

Comparing these numbers with the numbers in the table for linear probing, the linear probing numbers are remarkable close when the load factor is 50% or below.

The main advantage with double probing is that it allows for smaller tables (higher load factors) than linear or quadratic probing, but at the expense of higher costs to compute the next probe. The higher cost of computing the next probe may be preferable than longer probe sequences, especially when testing two keys equal is expensive.

2.3 Hash Table Summary

Hashing is a classic example of a space-time tradeoff: increase the space so table operations are faster, decrease the space but table operations are slower.

Separate chaining is simple to implement and is less sensitive to the quality of the hash function or load factors, so it is often the choice when it is unknown how many and how frequently keys may be inserted or deleted from the hash table. On the other hand open addressing can be more space efficient as there are no linked lists. Linear probing has the advantage that it has small constants and works well with caches since the locations checked are typically on the same cache line. But it suffers from primary clustering, which means its performance is sensitive to collisions and to high load factors. Quadratic probing, on the other hand, avoids primary clustering, but still suffers from secondary clustering and requires rehashing as soon as load factor reaches 50%. Although double hashing reduces clustering, so high load factors are possible, finding suitable pairs of hash functions is somewhat more difficult and increases the cost of a probe.

2.4 Parallel Hashing

How might we parallelize open addressing? In the parallel context, instead of inserting, finding or deleting one key at a time, each operation takes set of keys. Since a hash function distributes keys across slots in the table, we can expect many keys will be hashed to different locations. The idea is to

use open addressing in multiple rounds. For `insert`, each round attempts to write the keys into the table at their appropriate hash position. For any key that cannot be written because another key is already there, the key continues for another round using its next probe location. Rounds repeat until it writes all the keys to the table.

In order to prevent writing to a position already occupied in the table, we introduce a variant of the `inject` function. The function

$$\texttt{injectCond}(IV, S) : (\texttt{int} \times \alpha)\ \texttt{seq} \times (\alpha\ \texttt{option})\ \texttt{seq} \to (\alpha\ \texttt{option})\ \texttt{seq}$$

takes a sequence of index-value pairs $\langle (i_1, v_1), \ldots, (i_n, v_n) \rangle$ and a target sequence S and conditionally writes each value v_j into location i_j of S. In particular it writes the value only if the location is set to NONE and there is no previous equal index in IV. That is, it conditionally writes the value for the *first* occurrence of an index; recall `inject` uses the *last* occurrence of an index.

```
1  function insert(T, K) =
2  let
3    function insert'(T, K, i) =
4      if |K| = 0 then T
5      else let
6          T' = injectCond({(h(k, i), k) : k ∈ K}, T)
7          K' = {k : k ∈ K | T[h(k, i)] ≠ k}
8      in
9        insert'(T', K', i + 1)      end
10 in
11   insert'(T, k, 1)
12 end
```

For round i, `insert` uses each key's i^{th} probe in its probe sequence and attempts to write the key to the table. To see whether it successfully wrote a key to the table, it reads the value written to the table and checks if it is the same as the key. In this way it can filter out all keys that it successfully wrote to the table. It repeats the process on the keys it didn't manage to write, using the keys' $(i+1)$ probes.

For example, let's say that the table has the following entries before round i:

$T =$

0	1	2	3	4	5	6	7
	A		B			D	F

If $K = \langle E, F \rangle$ and $h(E, i)$ is 1 and $h(F, i)$ is 2, then $IV = \langle (1, E), (2, F) \rangle$ and `insert'` would fail to write E to index 1 but would succeed in writing F to index 2 resulting in the following table:

$T' =$

0	1	2	3	4	5	6	7
	A	F	B			D	F

It then repeats the process with $K' = \langle E \rangle$ and $i + 1$.

Note that if T is implemented using a single threaded array, then parallel `insert` basically does the same work as the sequential version adding the keys one by one. The only difference is that the parallel version may add keys to the table in a different order than the sequential. For example, with

linear probing, the parallel version adds F first using 1 probe and then adds E at index 4 using 4 probes:

$T_P =$

0	1	2	3	4	5	6	7
	A	F	B	E		D	F

Whereas, the sequential version might add E first using 2 probes, and then F using 3 probes:

$T_S =$

0	1	2	3	4	5	6	7
	A	E	B	F		D	F

Both make 5 probes in the table. Since we showed that, with suitable hash functions and load factors, the expected cost of insert is $O(1)$, the expected work for the parallel version is $O(|K|)$. In addition, in each round, the expected size of K decreases by a constant fraction, so span is $O(\log |K|)$.

How does a hash table implementation compare with a tree table implementation of Set and Table?

- Searches are faster in expectation
- Insertions are faster in expectation
- Map, reduce, and filter remain linear work
- Union/Merge can be slower.