Recitation 14 — Hashing and Leaftist Heaps

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

5 December 2012

Today's Agenda:

- Parallel Hashing
- Removing Duplicates
- Leafist Heaps Lemma

Announcements:

- Assignment 10 due this Friday. No code.
- Final exam Thursday, December 15, 5:30pm-8:30pm, GHC 4401

Questions from class?

1 Hashing Review

We have a large space of keys K (this might be infinite as in the case of the set of all possible strings, or simply large like all binary strings of length 1024) and a target range $\{0, \ldots, m-1\}$. We normally expect $|K| \gg m$. A hash function h is a mapping from K to $\{0, \ldots, m-1\}$.

One important application of hash functions is in implementing a dictionary data structure (hash table).

What are some properties we want out of a hash function?

- It should be a deterministic function—if we ask for the hash value of the same key twice, we should get the same value.
- It should distribute keys across buckets uniformly.

In general, there's a tension between wanting "random-like" behavior and wanting determinism. For now, we'll suppose we have a "good" hash function (without going too deep into what that means). For concreteness, though, we will make the following assumption:

> **The Simple Uniform Hashing Assumption.** Any given key $k \in K$ is equally likely to be hashed to one of the m possible values, independently of what values other keys might hash to.

Let's now address the other half of the problem, collision resolution.

Q: What does "load factor" refer to and why does it matter?

A: The ratio n/m. Literally, it is how full the hash table is. It is also an indicator of how often we should expect a collision.

Q: What are some techniques for dealing with a collision?

A: Separate chaining and open addressing. In separate chaining, we build a so-called "chain"—a sequence or a list containing all the keys that hash to the same value; therefore, with this method, insertion, search, and deletion operations require traversing such a chain and has running time proportional to the length of the chain. In the other strategy, open addressing employs a technique that fits each key to a slot in a single array. For example, when a key k is inserted, we first consider the hash location $h(k)$ and if that is occupied, we proceed to a different location according to some *probe sequence* until an empty slot is found. Searching is done in the same manner.

Q: Can you suggest some probe sequence patterns?

A: Formally, a probe sequence is a function $h(k,i)$ which decides the i-th alternative location for key k; thus, for a key k, we'd consider locations $h(k,1), h(k,2), \ldots$ in this order.

Linear probing is the simplest and most widely used probe sequence strategy, whereby $h(k,i) = [h(k) + i)] \bmod m$. That is, first it tries $h(k)$, then $h(k)+1$, then $h(k)+2$, so on so forth. This sequence wraps around if we reach the end of the array. This strategy has very good cache locality and performs really well in practice. In fact, it also has many nice theoretical performance guarantees, which we won't be able to cover here. Because of its simplicity, linear probing is often the method of choice in generic hash-table implementations these days.

Alternatively, we have *quadratic probing* where $h(k,i) = [h(k)+i^2] \bmod m$, or *double hashing* where $h(k,i) = [h(k) + i \cdot g(k)] \bmod m$ utilizing a second hash function g. In theory, all these techniques perform within constant factors of each other in expectation and are ideal in different sets of conditions.

2 Parallel Hashing

Q: What do we mean by hashing in parallel? A: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.

Q: How might we parallelize open addressing? A: 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 in parallel. 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.

Let's write `insert`

```
1  fun insert(T,K) =
2  let
3    fun insert'(T,K,i) =
4      if |K| = 0 then T
5      else let
6        val T' = injectCond({(h(k,i),k) : k ∈ K}, T)
7        val 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` attempts to put each key k into the hash table at position $h(k,i)$, but only if the position is empty. To see whether it successfully wrote a key to the table, it reads the values written to the table and checks if they are the same as the keys. In this way it can filter out all keys that it successfully wrote to the table. It repeats the process on any keys that did not get hashed on the next round using the their next probe position $h(k,i+1)$. Rounds continued until every element was put in the hash table.

For example, suppose 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 which adds the keys one by one. The 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 $O(\log |K|)$.

2.1 Example application: removing duplicates

In class, one of the examples that showed good speedup was removing duplicates. That is, suppose we have a sequence of n elements possibly with some duplicate entries, and we want to remove the duplicates.

```
{ "quux", "foo", "bar", "foo", "baz", "bar" }
```

Brute force: look at all pairs. Q: Which pairs do we really need to look at? What's the relationship between two keys *having the same hash value* and *being equal*?

A: Being equal implies having the same hash index, but not the other way around.

Already, we have reduced our solution space by only comparing those keys that have the same hash.

Q: If we use separate chaining, how long do we expect a chain to be (at least intuitively)?

A: With the simple uniform hashing assumption, the probability that a key k hashes to a value t is $1/m$; therefore, the length of a chain is simply the number of keys, out of n keys, that hash to the same value t. We know by linearity of expectation that this is n/m.

This leads to the following algorithm: We're going to hash all the values in our sequence, keep all the elements in a bucket by themselves, and compare only those values within buckets.

Aside (ignore if you wish): Note that if the chains have lengths $B_1, B_2, \ldots, B_m$. Comparing entries within chain i will take $O(B_i^2)$ work; therefore, *even if* $\mathbf{E}\left[B_i\right] = n/m$ may be constant, the expected work $\mathbf{E}\left[B_i^2\right]$ can be much larger. Analyzing this requires looking at the "second moment" of B_i.

On our example above, let's suppose "bar" and "baz" hash to the same index. We'd get

1	2	3
foo, foo	bar, baz, bar	quux

Q: How can we implement this in parallel?

A: Hmm... separate chaining is non-trivial to implement in parallel.

Fortunately, there is a way use open addressing to give $O(n)$ work and $O(\log n)$ span.

Q: Recall, how did we hashed in parallel using open addressing?

Q: How might we use parallel open addressing to check for duplicates?

A: Each element attempts to write its value in the hash table. Let's say the initial input consists of n elements and we're using a table of size m. As will be apparent, we'll want m to be larger than n to guarantee efficiency.

Q: What do we know about the values that get hashed?

A: They must be unique, since only one of the equal values can write to the hash table. Notice, though, that m is larger than n, so in this case, to extract out these unique elements, we're generally better off filtering on the input elements (cost: about n) instead of looking at all of the table (cost: about m).

Q: How does an element know if it was the one that wrote to the hash table?

A: It writes not only its value, but also its index in the sequence. That is, element S_i writes (S_i, i) into location $h(S_i)$.

Q: How does an element know if it is a duplicate?

A: It has the same key as the element in the hash table but not the same index. That is, element S_j is a duplicate if what got written in $h(S_j)$ is (x, ℓ) where $x = S_j$ but $\ell \neq j$.

Q: Do all unique elements get written to the hash table?

A: No. Some may collide with elements in the hash table. This situation is similar to what happened in open addressing; however, we'll resolve the conflict differently.

Q: What can we do with elements that are not duplicates and have not been added to the hash table?

A: Repeat the process until there are no elements left.

In summary, using contraction, we proceed in rounds, where each round does the following:

1. For $i = 0, \ldots, |S| - 1$, each element S_i attempts to "write" the value (S_i, i) into location $h(S_i)$ in an array using `injectCond`.
2. We will divide S into unique elements (ACCEPT) and potentially unique elements (RETRY) as follows:

$$\mathsf{ACCEPT} = \langle S_i \in S | T[h(S_i)] = (S_i, i) \rangle$$
$$\mathsf{RETRY} = \langle S_i \in S | \#1(T[h(S_i)]) \neq S_i \rangle$$

3. Recurse on RETRY, appending together all the ACCEPT's.

The ACCEPT elements are those that successfully wrote to the hash table, and the RETRY elements are those that attempted to but did not write to the hash table and are not a duplicate of an element in ACCEPT. *It is crucial that* RETRY *does not contain a duplicate of an element in* ACCEPT. Further, note that implicitly, there is the other group REJECT which is thrown away: this group is made up of the elements that are duplicates of what we already have in ACCEPT.

Why is this algorithm correct? It is easy to see that if a key k is present in S, we'll never throw it away until we include it in ACCEPT, so we only need to argue that we never put two copies of the same key in ACCEPT. Let's consider a round of this algorithm. If S_i and S_j are the same key, only one of them will be in ACCEPT, and furthermore, none of the entries of this key can be in RETRY.

We'll now analyze this algorithm: To bound work, we're interested in knowing how large RETRY is on an input sequence S of length n. Although the worst case might be bad, we're happy with expected-case behaviors. For this, it suffices to compute the probability that an entry S_i is included in RETRY. This happens *only if* the key S_i hashes to the same value as some other key S_j where $S_i \neq S_j$. Therefore, with the simple uniform hashing assumption, we have that for any i,

$$\begin{aligned}
\mathbf{Pr}\left[S_i \in \mathsf{RETRY}\right] &\leq \mathbf{Pr}\left[\exists j \text{ s.t. } S_i \neq S_j \wedge h(S_i) = h(S_j)\right] \\
&\leq \sum_{j:S_j \neq S_i} \mathbf{Pr}\left[h(S_i) = h(S_j)\right] \\
&\leq n/m,
\end{aligned}$$

where we have upperbounded the probability with a union bound.

If $m = 3n/2$, then $n/m = 2/3$, and by linearity of expectation, we have $|\mathsf{RETRY}| \leq 2n/3$. We have seen this recurrence pattern before; this gives that the total work is expected $O(n)$ because in expectation, this forms a geometrically decreasing sequence. Furthermore, applying KUW, we know that the number iterations is expected $O(\log n)$.

2.2 Leftist Heaps

It turns out there is a relatively easy fix to this imbalance problem. The idea is to keep the trees so that the trees are always deeper on the left than the right. In particular, we define the *rank* of a node x as

and more formally:

$$\mathsf{rank}(\mathsf{leaf}) = 0$$
$$\mathsf{rank}(\mathsf{node}(\_, \_, R) = 1 + \mathsf{rank}(R)$$

Recall that all nodes of a leftist heap have the "leftist property". That is, if $L(x)$ and $R(x)$ are the left and child children of x, then we have:

Leftist Property: For all node x in a leftist heap, $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x))$

where

$$\mathsf{rank}(x) = \#\text{ of nodes on the right spine of the subtree rooted at } x,$$

This is why the tree is called leftist: for each node in the heap, the rank of the left child must be at least the rank of the right child. That way, the right spine of a leftist heap is kept short as most of the entries will amass on the left.

The following lemma was stated in lecture, but not proved.

Lemma 2.1. *In a leftist heap with n entries, the rank of the root node is at most* $\log_2(n+1)$.

In words, this lemma says *leftist heaps have a short right spine,* about $\log n$ in length.

Before proving Lemma **??** we will first prove a claim that relates the number of nodes in a leftist heap to the rank of the heap.

Claim: If a heap has rank r, it contains at least $2^r - 1$ entries.

To prove this claim, let $n(r)$ denote the number of nodes in the smallest leftist heap with rank r. It is not hard to convince ourselves that $n(r)$ is a monotone function; that is, if $r' \geq r$, then $n(r') \geq n(r)$. With that, we'll establish a recurrence for $n(r)$. By definition, a rank-0 heap has 0 nodes. We can establish a recurrence for $n(r)$ as follows: Consider the heap with root note x that has rank r. It must be the case that the right child of x has rank $r-1$, by the definition of rank. Moreover, by the leftist property, the rank of the left child of x must be at least the rank of the right child of x, which in turn means that $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x)) = r-1$. As the size of the tree rooted x is $1 + |L(x)| + |R(x)|$, the smallest size this tree can be is

$$\begin{aligned} n(r) &= 1 + n(\mathsf{rank}(L(x))) + n(\mathsf{rank}(R(x))) \\ &\geq 1 + n(r-1) + n(r-1) = 1 + 2 \cdot n(r-1). \end{aligned}$$

Unfolding the recurrence, we get $n(r) \geq 2^r - 1$, which proves the claim.

Proof of Lemma **??**. To prove that the rank of the leftist heap with n nodes is at most $\log(n+1)$, we simply apply the claim: Consider a leftist heap with n nodes and suppose it has rank r. By the claim it must be the case that $n \geq n(r)$, because $n(r)$ is the fewest possible number of nodes in a heap with rank r. But then, by the claim above, we know that $n(r) \geq 2^r - 1$, so

$$n \geq n(r) \geq 2^r - 1 \implies 2^r \leq n+1 \implies r \leq \log_2(n+1).$$

This concludes the proof that the rank of a leftist heap is $r \leq \log_2(n+1)$. □