Recitation 15 – *Priority Queues, Hashing, and Leftist Heaps*

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2014)

April 22nd, 2014

1 Announcements

- *DPLab* is due on Wednesday.
- *CilkLab* is coming out on Wednesday! You will get some experience writing and analyzing **real** parallel code.

2 Meldable Priority Queues and Leftist Heaps

Q: What is a meldable priority queue?

A: An ADT for priority queues that supports meld, an operation that combines two priority queues into one.

Q: How are priority queues typically implemented?

A: With heaps: tree-based data structures that only maintain a partial ordering on their keys.

Q: What are the two major properties of a *binary* heap?

A: The *shape property* requires that the tree is a complete binary tree. The *heap property* enforces a partial ordering on the keys: in a *min*-heap, the key at each node must be less than both of its descendants. In a *max*-heap, it must be greater.

Q: What are the two major properties of a *leftist* heap?

A: The *heap property* is the same as before. The *leftist property* requires that for every node x with children L and R, $\mathsf{rank}(L) \geq \mathsf{rank}(R)$. We define $\mathsf{rank}(x)$ as the number of nodes in the right spine of x.

Q: Why do leftist heaps improve the cost of meld?

A: Lemma 20.3 from lecture states, "In a leftist heap with n entries, the rank of the root node is at most $\log_2(n+1)$."

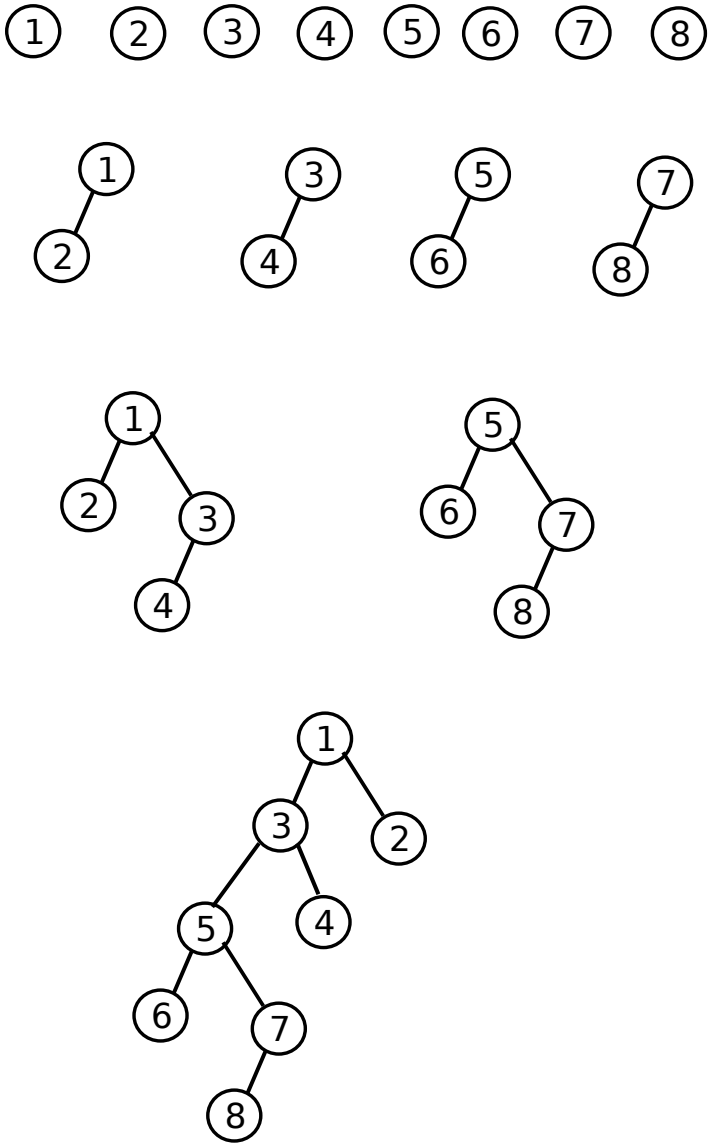
We know $\mathsf{meld}(A, B)$ only traverses the right spine of both A and B, so it follows that the work of meld is bounded by $O(\log|A| + \log|B|)$.

The code for meld on *min*-heaps from lecture is given below.

```
1  datatype PQ = Leaf | Node of (int × key × PQ × PQ)

2  fun rank Leaf = 0
3    | rank (Node(r, _, _, _)) = r

4  fun makeLeftistNode (v, L, R) =
5    if (rank(L) < rank(R))
6    then Node(1 + rank(L), v, R, L)
7    else Node(1 + rank(R), v, L, R)

8  fun meld (A, B) =
9    case (A, B) of
10     (_, Leaf) ⇒ A
11   | (Leaf, _) ⇒ B
12   | (Node(_, k_a, L_a, R_a), Node(_, k_b, L_b, R_b)) ⇒
13       case Key.compare(k_a, k_b) of
14         LESS ⇒ makeLeftistNode (k_a, L_a, meld(R_a, B))
15       | _ ⇒ makeLeftistNode (k_b, L_b, meld(A, R_b))
```

Given a sequence $S = \langle 1,2,3,4,5,6,7,8 \rangle$, what is the resulting leftist heap after applying the following code?

```
fun singleton v = Node(1, v, Leaf, Leaf)
Seq.reduce meld Leaf (Seq.map singleton S)
```


1 2 3 4 5 6 7 8
1 2
3 4
5 6
7 8
1 2 3 4
5 6 7 8
1 3 2 5 4 6 7 8

3 Hashing Review

We have a large space S_{keys} of keys, and a target range $\{0, \ldots, m-1\}$. We typically expect $|S_{keys}| \gg m$ (S_{keys} may even be infinite). A hash function is a mapping from S_{keys} to $\{0, \ldots, m-1\}$.

Q: What are the desired properties of a hash function?

A: It should be *deterministic,* and it should distribute keys uniformly across the target range.

Q: What is the "load factor" of a hash table, and what does it indicate?

A: The ratio n/m, where n is the number of keys that have been inserted, and m is the size of the table. This value indicates how full the hash table is, and consequently how often we should expect a collision.

Q: What are some possible techniques for handling collisions?

A: *Separate chaining* and *open addressing*.

Separate chaining forms lists ("chains") of keys that all map to the same hash value. Insertion, search, and deletion then all have running time proportional to the length of the chain.

Open addressing fits each key into one array slot. If a key k is not found at the initial location given by the hash function, then it might be stored at the next location in the *probe sequence*. The probe sequence is defined by a function, where $h(k,i)$ is the i^{th} alternative location for the key k.

Q: What are some examples of different probe sequences?

A: *Linear probing* is the simplest and most widely-used strategy, where $h(k,i) = [h(k) + i] \bmod m$. *Quadratic probing* defines $h(k,i) = [h(k) + i^2] \bmod m$. *Double hashing* utilizes a second hash function, g, and defines $h(k,i) = [h(k) + i \cdot g(k)] \bmod m$.

In theory, all of these techniques perform within constant factors of one another in expectation, and are ideal in different sets of conditions.

4 Parallel Hashing

Q: What do we mean by parallel hashing?

A: Instead of finding, inserting, or deleting one key at a time, each operation takes a set of keys and performs the operations on all of the keys in parallel. In this context, we have to be especially careful about collisions.

Q: How might we parallelize open addressing?

A: We perform multiple rounds. For `insert`, each round attempts to write the keys into the table at their appropriate positions in parallel. Any key that cannot be written because of a collision continues into the next round. We repeat until every key has been written into 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.

For example, if $S = \langle \texttt{SOME}\ 5, \texttt{NONE}, \texttt{NONE}, \texttt{SOME}\ 42, \texttt{NONE}, \texttt{SOME}\ 28 \rangle$, then $\texttt{injectCond}(\langle (3,43), (1,97), (4,8), (1,35) \rangle, S) = \langle \texttt{SOME}\ 5, \texttt{SOME}\ 97, \texttt{NONE}, \texttt{SOME}\ 42, \texttt{SOME}\ 8, \texttt{SOME}\ 28 \rangle$.

Let's write `insert`. Let T be our hash table and K be the set of keys we wish to 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)
10     end
11 in
12   insert'(T, K, 0)
13 end
```

For round i (starting at $i = 0$), `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 the 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 their next probe position $h(k, i+1)$. Rounds continue 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 $K = \langle E, C \rangle$ and $h(E, i)$ is 1 and $h(C, i)$ is 2, then $IV = \langle (1, E), (2, C) \rangle$ and `insert'` would fail to write E to index 1 but would succeed in writing C to index 2, resulting in the following table:

$T' =$

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

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

Note that if T is implemented using an `stseq`, 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 C 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	C	B	E		D	F

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

$T_S =$

0	1	2	3	4	5	6	7
	A	E	B	C		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, if the hash table is large enough, the expected size of K decreases by a constant fraction in each round, so the span is $O(\log |K|)$.

5 Example Hashing 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.

$$\langle \text{“}quux\text{”}, \text{“}foo\text{”}, \text{“}bar\text{”}, \text{“}foo\text{”}, \text{“}baz\text{”}, \text{“}bar\text{”} \rangle$$

$$\Downarrow$$

$$\langle \text{“}quux\text{”}, \text{“}foo\text{”}, \text{“}bar\text{”}, \text{“}baz\text{”} \rangle$$

A brute force solution would look at all pairs, but this is clearly inefficient. Which pairs do we really need to look at?

Q: What's the relationship between two keys *having the same hash value* and *being equal*?

A: Being equal $\Rightarrow$ having the same hash index, but having the same hash index $\not\Rightarrow$ being equal.

Already, we have reduced our solution space by only comparing those keys that have the same hash. This leads to the following algorithm: hash all the values in our sequence and compare only those that fall within the same bucket. We'll use open addressing, and will be able to achieve $O(n)$ work and $O(\log n)$ span!

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

A: Each element attempts to write its value in the hash table. If two keys are equal, only one of them gets successfully hashed.

Q: How do we proceed to the next round in order to eliminate duplicates?

A: When collecting the keys to retry inserting in the next round (by comparing each element to the one at its hashed position in the array), exclude duplicates of elements that already got hashed.

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

A: Instead of just writing elements into the table, we write a pair (v, i) where v is the value and i is its index in the original sequence. If two elements are duplicates, only one of them will get written with its index. The other has the same key as the element written to 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, so we repeat the process until there are no elements left. However, unlike hashing with open addressing, we can start the second round with an empty hash table after collecting the elements that successfully hashed, since these are definitely in our non-duplicate sequence.

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:

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

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, if we assume simple uniform hashing, 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 upper bounded 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, we can show that the number iterations is $O(\log n)$ in expectation.