CS 1332 - Week 06

Monday

Topics

I. HashMaps review
   A. key-value pairs are stored as objects with key and value fields
   B. n - the number of entries in the map
   C. N - the capacity of the backing array, collisions are reduced when N is prime
   D. Load Factor = n / N
      1. max load factor - some threshold for the load factor to trigger resizing, typically between 50% and 75%
      2. collisions are reduced when resizing to 2N + 1 (keep the capacity odd)
   E. H(k) - hashcode of key k, converts the key to some integer (i)
   F. i % N - compression function, fits the hashcode into the bounds of the array
   G. External Chaining
      1. Closed addressing - once we compress the hashcode to find the index, this is definitely the index at which the entry with that key will be

II. HashMaps vs. HashTables

HashMap	HashTable
● Asynchronous ● Simple ● Fast, less space ● Allows null keys/values ● Iterator to see all values	● Synchronous ● Complex ● More time and memory ● Does not allow null keys or values ● Enumerator to see all values

III. HashMap ADT - implementation specifics depend on collision resolution strategy
   A. v put(<k, v>) → takes in an entry and puts it in the HashMap, returns null or a value if one was replaced due to a duplicate key being passed in
      1. finds the hashcode of the key
      2. compresses the hashcode to get an index
      3. O(1) with external chaining → just add to the head
   B. v get(k) → takes in a key, returns the value associated with it
      1. O(n) with external chaining → search through the linked list
   C. <k,v> remove(k) → takes in a key, removes and returns the entry with that key
      1. O(n) with external chaining → perform a LL remove
   D. int size() → returns the size, O(1) whenever the size is stored as a field

IV. Collision Strategy - Probing
   A. Open addressing - once we calculate the index, the actual index the entry ends up at can be different
   B. Linear probing - if a collision occurs, we'll just increase the index by one and check again (wrap around back to index 0 if necessary); in other words, we add the number of times we've probed to the original index

1. Example - put 8, 1, 15, 5, 2, 22, 50 in a backing array of length 7

key	h(k)	original index: h(k) % N	probe count	index: orig index + probe
8	8	8 % 7 = 1	0	1 + 0 = 1
1	1	1 % 7 = 1	0	1 + 0 = 1 (filled)
			1	1 + 1 = 2
15	15	15 % 7 = 1	0	1 + 0 = 1 (filled)
			1	1 + 1 = 2 (filled)
			2	1 + 2 = 3
5	5	5 % 7 = 5	0	5 + 0 = 5
2	2	2 % 7 = 2	0	2 + 0 = 2 (filled)
			1	2 + 1 = 3 (filled)
			2	2 + 2 = 4
22	22	22 % 7 = 1	0	1 + 0 = 1 (filled)
			1	1 + 1 = 2 (filled)
			2	1 + 2 = 3 (filled)
			3	1 + 3 = 4 (filled)
			4	1 + 4 = 5 (filled)
			5	1 + 5 = 6
50	50	50 % 7 = 1	0 to 5	(filled)
			6	1 + 6 = 7 % 7 = 0

   a) Remove(1) - original index = 1 % 7 = 1
      (1) arr[1] is not null but 8 != 1, so we keep probing
      (2) arr[2] is not null and 1 == 1
      (3) Instead of setting arr[2] to null, we'll add a "delete" marker
         (a) The key does not exist if we probe to a null index.
         (b) When we added 15, we probed passed 1, so we need some indication to keep probing even after we remove the entry with 1 as the key

b) Get(1): original index = 1 % 7 = 1

0	1	2	3	4	5	6
	8	DEL	15	DEL		22

(1) arr[1] is not null but 8 != 1
(2) arr[2] is deleted, so we keep probing
(3) arr[3] is not null but 15 != 3
(4) arr[4] is deleted, so we keep probing
(5) arr[5] is null → 1 is not in the HashMap

No Content - Wednesday

Friday

Topics

I. Collision Strategy - Quadratic Probing
   A. Quadratic probing - Similar to linear probing in that you use the number of probes (0,1,2,3…) to find where you go from the original index. Instead of just adding the probe number, you add the probe number SQUARED to the original index
      1. Quadratic probing aims to break up clusters created by linear probing
   B. Example:
      1. Backing array size: N = 7
      2. Hash function: h(k) = k % 7
      3. Data: 8, 1, 15, 2, 5, 22
      4. Number of probes: p = 0, 1, 2, 3, 4 ...
         a) h(8) = 1, probes: $1 + 0^2$
         b) h(1) = 1, probes: $1 + 0^2 \rightarrow 1 + 1^2$
         c) h(15) = 1, probes: $1 + 0^2 \rightarrow 1 + 1^2 \rightarrow 1 + 2^2$
         d) h(2) = 2, probes: $2 + 0^2 \rightarrow 2 + 1^2$
         e) h(5) = 5, probes: $5 + 0^2 \rightarrow 5 + 1^2$
         f) h(22) = 1; probes: $1 + 0^2 \rightarrow 1 + 1^2 \rightarrow 1 + 2^2 \rightarrow 1 + 3^2 \rightarrow 1 + 4^2 \rightarrow 1 + 5^2 \rightarrow 1 + 6^2 \rightarrow$ stop probing after N times, otherwise you can run into infinite loops when no spaces can be found with this collision strategy (if you probe N times and cannot place the data, resize the HashMap to make more clear spaces)

II. Collision Strategy - Double Hashing
   A. Double hashing - more similar to linear probing because you add a factor of the probe count to the original index; the factor is determined by a second hash
   B. You have two hash functions:
      1. h(k) = k % N (just like before)
      2. d(k) = q - k % q (where q is prime and less than the capacity)
         a) eg. d(k) = 5 - k % 5
   C. Double hash calculations (in a backing array of length 7)

key	first hash: h(k)	second hash: d(k)	probe	index: h(k) + d(k)*p
8	1	5 - 8%5 = 2	0	1 + 2(0) = 1 % 7 = 1
1	1	5 - 1%5 = 4	0	1 + 4(0) = 1 % 7 = 1
			1	1 + 4(1) = 5 % 7 = 5
15	1	5 - 15%5 = 5	0	1 + 5(0) = 1 % 7 = 1
			1	1 + 5(1) = 6 % 7 = 6
2	2	5 - 2%5 = 3	0	2 + 3(0) = 2 % 7 = 2
5	5	5 - 5%5 = 5	0	5 + 5(0) = 5 % 7 = 5
			1	5 + 5(1) = 10 % 7 = 3
22	1	5 - 22%5 = 3	0	1 + 3(0) = 1 % 7 = 1
			1	1 + 3(1) = 4 % 7 = 4