CS 61B Spring 2022

Heaps & Hashing

Discussion 10: March 28, 2022

1 Heaps of Fun

(a) Consider an array-based min-heap with N elements. What is the worst case asymptotic runtime of each of the following operations if we ignore resizing? What is the worst case asymptotic runtime if we take resizing into account?

	Without Resizing	With Resizing
Insert		
Find Min		
Remove Min		

(b) What are the advantages of using an array-based heap over a pointer-based heap?

Without Resizing:

- Insert: When we insert an item into the min-heap, we place it in the uppermost leftmost available spot in the tree. In the worst case, we need to bubble up the item all the way to the root. The height of a heap is $log_2 N$ (this is *always* true becuase heaps are complete binary trees), so bubbling the item to the top of the tree requires $log_2(N-1) \in \Theta(logN)$ swaps. Therefore, the worst case runtime for inserting an item into a min-heap (without resizing) is $\Theta(logN)$.
- Find Min: By definition, the smallest item will always be at the root of a min-heap. The root of a heap will always be at index 1 of the array in which the items are stored. Indexing into an array takes $\Theta(1)$ time, so therefore finding the minimum item in a min-heap takes $\Theta(1)$ time.
- Remove Min: When we remove an item from the min-heap, we make the rightmost leaf element the new root. In the worst case, this new root value needs to be bubbled all the way back down to the lowest level of the heap. If there are N nodes in the heap then there are $log_2 N$ levels, resulting in $log_2(N-1) \in \Theta(logN)$ swaps. Therefore, the worst case runtime for Remove Min (without resizing) is $\Theta(logN)$.

With Resizing:

- Insert: In the worst case, the array we are trying to insert into is already full. We need to make a new array to store the items of the min-heap, and then copy over all N items into the new array. Copying over a single array element takes constant time, so copying N items will take a total of $\Theta(N)$ time.
- Find Min: Same explanation as Find Min without resizing. We do not need to do any resizing operations when we are finding the minimum

element of the min-heap.

- Remove Min: In general, Java data structures do not size down. Therefore, removing an item from a resizing heap has the same runtime as removing an item from a heap that does not resize, giving us a worst case runtime of $\Theta(logN)$.

  Note: If we decided to use a data structure that *does* resize down, then after reaching some minimum occupancy we would need to create a new smaller array. All the items would then need to be copied over into the new array, resulting in a runtime of $\Theta(N)$.

(c) How can you implement a max-heap of integers if you only have access to a min-heap?

Using a pointer-based representation is not as space-efficient. For an array-based heap, you simply need to keep a cell for each element. For a pointer-based heap you need to maintain pointers to each element's child in adddition to keeping a field to store the element itself.

(d) Given an array and a min-heap, describe an algorithm that would allow you to sort the elements of the array in ascending order. Give the best and worst case runtime of your algorithm.

For every `insert` operation, negate the number and add it to the min-heap. To perform a `removeMax` operation, call `removeMin` on the min-heap and negate the number returned.

2 Assorted Heap Questions

(a) Describe a way to modify the usual max heap implementation so that finding the minimum element takes constant time without incurring more than a constant amount of additional time and space for the other operations.

Simply add a variable that keeps track of the minimum value in the heap. When inserting a new value, simply update this variable if the new value is smaller than it. Since the max heap only supports removing the largest element, rather than arbitrary elements, the minimum element will only be removed when the heap becomes empty, at which point we will need to reset the variable keeping track of the minimum value.

(b) In class, we looked at one way of implementing a priority queue: the binary heap. Recall that a binary heap is a nearly complete binary tree such that any node is smaller than all of its children. There is a natural generalization of this idea called a d-ary heap. This is also a nearly complete tree where every node is smaller than all of its children. But instead of every node having two children, every node has d children for some fixed constant d.

(i) Describe how to insert a new element into a d-ary heap (this should be very similar to the binary heap case). What is the running time in terms of d and n (the number of elements)?

To insert, we add the new element on the last level of the tree and then bubble it up, much as in a binary heap. When bubbling up, we only need to compare the node to its parent. Since the tree has $\Theta(\log_d(n))$ levels and we have to do at most one comparison (compare the node to its parent) and one swap at each level, insertion takes $\Theta(\log_d(n))$ time.

(ii) What is the running time of finding the minimum element in a d-ary heap with n nodes in terms of d and n?

The minimum element is simply the root, just as with a binary min-heap. So finding it takes $\Theta(1)$ time.

(iii) Describe how to remove the minimum element from a d-ary heap (this should be very similar to the binary heap case). What is the running time in terms of d and n?

To remove the minimum element, we first replace it with the last element on the last level of the heap and then bubble this element dowm, just as in a binary heap. When bubbling a node down, we have to compare the node to all of its children to determine if it is larger than any of them and to find the smallest one (since we must swap with the smallest child to preserve the heap property). So at each level we have to do at most $\Theta(d)$ comparisons and 1 swap. Since there are $\Theta(\log_d(n))$ levels, removing the minimum takes $\Theta(d \log_d(n))$ time.

3 HashMap Modification (61BL Summer 2010, MT2)

(a) If you modify a **key** that has been inserted into a `HashMap`, can you retrieve that entry again? Explain.

☐ Always ☐ Sometimes ☐ Never

Sometimes. It is possible that the new key will end up colliding with the old key. Only in this rare situation will we be able to retrieve the value. Otherwise, the new key will hash to a different hash code, causing us to look in the wrong bucket inside our `HashMap` for our entry. It is very bad to modify the key in a map because we cannot guarantee that the data structure will be able to find the object for us if we change the key.

(b) If you modify a **value** that has been inserted into a `HashMap`, can you retrieve that entry again? Explain.

☐ Always ☐ Sometimes ☐ Never

Always. You can safely modify the value without any trouble. When you retrieve the value from the map, the changes made to the value will be reflected. We use the key to determine where to look for our value inside our `HashMap`, and because the key hasn't been changed, we are still able to find the entry we are looking for.

4 Hash Code

In order for a hash code to be valid, objects that are equivalent to each other (i.e. `.equals()` returns true) must return equivalent hash codes. If an object does not explicitly override the `hashCode()` method, it will inherit the `hashCode()` method defined in the `Object` class, which returns the object's address in memory.

Here are four potential implementations of `Integer`'s `hashCode()` function. Assume that `intValue()` returns the value represented by the `Integer` object. Categorize each `hashCode()` implementation as either a valid or an invalid hash function. If it is invalid, explain why. If it is valid, point out a flaw or disadvantage.

(a)
```
public int hashCode() {
    return -1;
}
```

Valid. As required, this hash function returns the same hash code for `Integer`s that are `.equals()` to each other. However, this is a terrible hash code because collisions are extremely frequent and occur 100% of the time.

(b)
```
public int hashCode() {
    return intValue() * intValue();
}
```

Valid. Similar to (a), this hash function returns the same hash code for `Integer`s that are `.equals()`. However, `Integer`s that share the same absolute values will collide (for example, `x = 5` and `x = -5` will both return the same hash code). A better hash function would be to just return `intValue()` itself.

(c)
```
public int hashCode() {
    Random rand = new Random();
    return rand.nextInt();
}
```

Invalid. If we call `hashCode()` multiple times on the same `Integer` object, we will get different hash codes returned each time.

(d)
```
public int hashCode() {
    return super.hashCode();
}
```

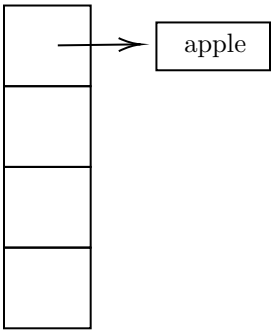
Invalid. This hash function returns some integer corresponding to the `Integer` object's location in memory. When a new object is instaniated, it is created in a new spot in memory even when it is functionally identical to an existing object. This is so modifying the object does not result in the other object being modified. In order to check whether or not an object exists within a hashtable, we find the hashcode of an equivalent object (in this case an Integer with the same int value), and check that bucket. Different `Integer` objects will exist in different locations in memory, so even if they represent the same value they will return different hash codes.

5 Hashing Practice

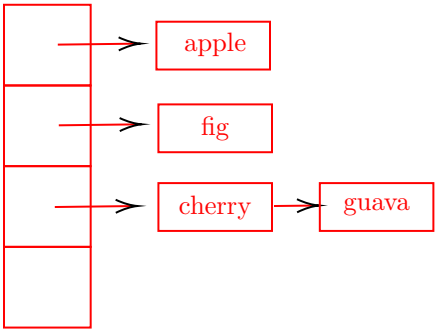
Given the provided `hashCode()` implementation, hash the items listed below with external chaining (the first item is already inserted for you). Assume the load factor is 1. Use geometric resizing with a resize factor of 2. You may draw more boxes to extend the array when you need to resize.

```
/** Returns 0 if word begins with 'a', 1 if it begins with 'b', etc. */
public int hashCode() {
    return word.charAt(0) - 'a';
}
```

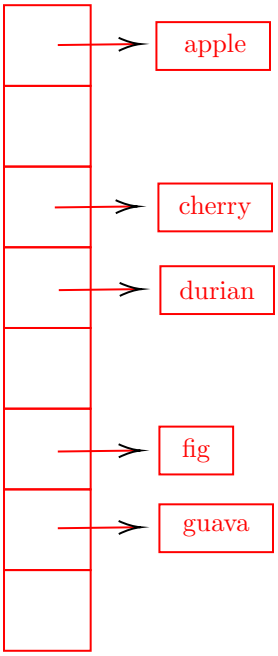
```
["apple", "cherry", "fig", "guava", "durian", "apricot", "banana"]
```



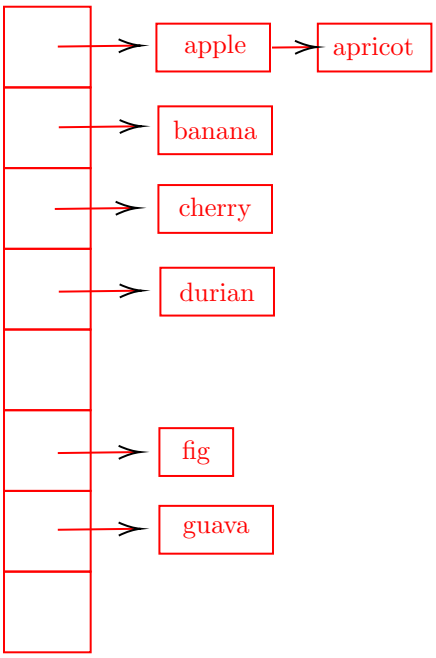
Here is what the hash table should look like after inserting `guava`:



Here is what the hash table should look like after inserting `durian`:



Here is what the hash table should look like after all insertions have been completed:



Extra Suppose that we represent Tic-Tac-Toe boards as 3×3 arrays of integers (with each integer in the range [0, 2] to represent blank, 'X', and 'O', respectively). Describe a hash function for Tic-Tac-Toe boards that are represented in this way such that boards that are not equal will never have the same hash code.

We can interpret the Tic-Tac-Toe board as a nine-digit base 3 number, and use this as the hash code. More concretely, if the array used to store the Tic-Tac-Toe board was called board, then we could compute the hash code as follows:

$$\text{board}[0][0] + 3 \cdot \text{board}[0][1] + 3^2 \cdot \text{board}[0][2] + 3^3 \cdot \text{board}[1][0] + \ldots + 3^8 \cdot \text{board}[2][2]$$

This hash code actually guarantees that any two distinct Tic-Tac-Toe boards will always have distinct hash codes (in most situations this property is not feasible). Note that if we used this same idea on boards of size $N \times N$, it would take $\Theta(N^2)$ time to compute the hash.