CS 61B Heaps and Hashing Spring 2020

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	$\Theta(logN)$	$\Theta(N)$
Find Min	$\Theta(1)$	$\Theta(1)$
Remove Min	$\Theta(logN)$	$\Theta(logN)$

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 $\Theta(N)$.

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

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

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.

(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.

Insert all elements from the array into the min-heap. Remove all items from the min-heap one-by-one and place them back into the array at index 0, 1, 2, ..., etc. The resulting array should now be sorted in ascending order.

Runtime analysis:

- **Best case**: All items are equivalent. Because there is no need to bubble up or bubble down when inserting each item into the heap, each insertion takes $\Theta(1)$ time, giving us a total of $\Theta(N)$ for inserting all N items into the heap. If all items are equivalent, there is no need to bubble up or bubble down when removing an item. Therefore, removing a single item will take $\Theta(1)$ time and removing all N items from the heap takes $\Theta(N)$ time. Therefore the best case runtime is (insertion time) + (removal time), which is $\Theta(N+N) \in \Theta(2N) \in \Theta(N)$.
- **Worst case**: In the worst case, all items are different and for each insertion we must bubble the inserted item all the way up to the root of the heap. There are two ways to derive the runtime: finding an explicit tight bound on the runtime, and using lower and upper bounds to converge on a tight bound.

  1. **Finding an explicit tight bound**: Inserting an item into the heap takes $log_2(x)$ time, with x being the number of items currently in the heap. This means that when we insert item i into the heap, there will be $i-1$ items already in the heap. The table below shows the number of swaps performed when we insert item i into the heap.

ith item to be inserted	Number of swaps
1	0
2	$log_2(1)$
3	$log_2(2)$
4	$log_2(3)$
...	...
N	$log_2(N-1)$

The total runtime for inserting N items into the heap is simply the result of summing the total number of swaps performed, which is as follows:

$$\begin{aligned}
log(1)+log(2)+...+log(N-1) &= log((N-1)(N-2)(N-3)...(3)(2)(1)) \\
&= log((N-1)!) \\
&= log(\frac{N!}{N}) \\
&= log(N!)-log(N) \\
&\in \Theta(log(N!)) \\
&\in \Theta(NlogN)
\end{aligned}$$

Removing an item will also take $log_2(x)$ time, so by the same logic removing N items will also take $\Theta(NlogN)$ time. Therefore, the worst case runtime is (insertion time) + (removal time), which is $\Theta(NlogN+NlogN) \in \Theta(NlogN)$.

2. **Using lower and upper bounds**: We'll break down this into two steps: finding an upper bound, and finding a lower bound. We'll first look at the time it takes to insert all the items into the heap.
   (a) **Upper bound**: The worst case insertion time of any single element happens when we insert the last (Nth) item into the heap, resulting in a total of $log_2(N-1)$ swaps and taking $\Theta(logN)$ time. Therefore, the time it takes to insert N items will take no more than N $\cdot$ (worst case insertion time), which is $O(NlogN)$ time.
   (b) **Lower bound**: Finding the lower bound is a little trickier. We'll calculate the time it takes to perform a subset of insertion operations to our heap in order to give us a valid lower bound. Given items 1, 2, 3, ..., N, we'll only consider the insertion of the last $\frac{N}{2}$ items. Of the last $\frac{N}{2}$ items, the quickest insertion time is the $\frac{N}{2}$th item, which performs $log_2(\frac{N}{2}-1)$ swaps in $\Theta(log\frac{N}{2}) \in \Theta(logN)$ time. Performing the last $\frac{N}{2}$ insertions should therefore take at least $\frac{N}{2}$ $\cdot$ (quickest insertion time), which is $\Omega(\frac{N}{2}logN) \in \Omega(NlogN)$. Because the time to insert the last $\frac{N}{2}$ items is guaranteed to be smaller than the time it takes to insert *all* N items, this is also a valid lower bound for the time it takes to insert all N items.

   Since the upper and bounds are equivalent, we can safely conclude that the runtime to insert all N items into a heap is $\Theta(NlogN)$.

   Like Explanation 1, the same proof holds true for the removal of all N items from the heap. Therefore the worst case runtime for inserting and removing all N items from the heap is (insertion time) + (removal time), which is $\Theta(NlogN+NlogN) \in \Theta(NlogN)$.

If you want to implement a more efficient version of this sorting algorithm, check out the **heapsort** algorithm in next week's discussion worksheet.

2 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

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

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.

3 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.

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

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

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

Valid. Similar to (a), this hash function returns the same hash code for `Integers` that are `.equals()`. However, `Integers` 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.

```
(3) 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.

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

Invalid. This hash function returns some integer corresponding to the `Integer` object's location in memory. Different `Integer` objects will exist in different locations in memory, so even if they represent the same value they will return different hash codes.

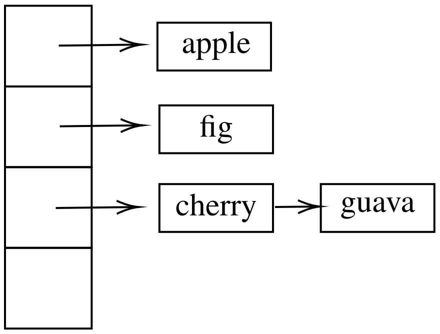
4 Hashing Practice

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.

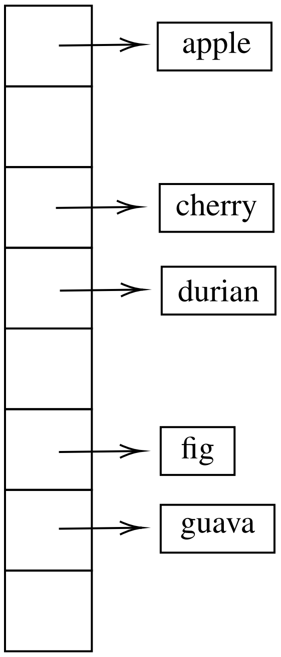
```java
/** Returns 0 if word begins with 'a', 1 if it begins with 'b', etc. */
public int hashCode() {
    return word.charAt(0) - 'a';
}
```

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

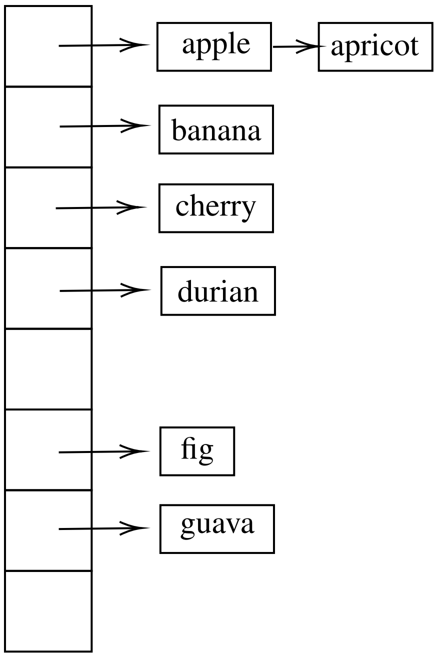
Here is what the hash table should look like after inserting `guava`:



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.