CS 61B Fall 2021

Heaps and Hashing

Exam Prep Discussion 10: October 25, 2021

1 Fill in the Blanks

Fill in the following blanks related to min-heaps. Let `N` is the number of elements in the min-heap. For the entirety of this question, assume the elements in the min-heap are **distinct**.

1. `removeMin` has a best case runtime of _____ and a worst case runtime of _____.
2. `insert` has a best case runtime of _____ and a worst case runtime of _____.
3. A _____ or _____ traversal on a min-heap *may* output the elements in sorted order. Assume there are at least 3 elements in the min-heap.
4. The fourth smallest element in a min-heap with 1000 elements can appear in _____ places in the heap.
5. Given a min-heap with $2^N - 1$ distinct elements, for an element
   - to be on the second level it must be less than _____ element(s) and greater than _____ element(s).
   - to be on the bottommost level it must be less than _____ element(s) and greater than _____ element(s).

   Hint: A complete binary tree (with a full last-level) has $2^N - 1$ elements, with N being of levels.

Solution:

1. `removeMin` has a best case runtime of $\Theta(1)$ and a worst case runtime of $\Theta(logN)$.
2. `insert` has a best case runtime of $\Theta(1)$ and a worst case runtime of $\Theta(logN)$.
3. A pre order or level order traversal on a min-heap can output the elements in sorted order.
   Explanation: The smallest item of a min heap is at the top, so whatever traversal we choose must output the top element first in a complete binary tree. Only preorder and level-order have this property.
4. The fourth smallest element in a min-heap with 1000 distinct elements can appear in 14 places in the heap.
   Explanation: The 4th smallest item can be on the 2nd, 3rd, or 4th level of the heap.
5. Given a min-heap with $2^N - 1$ distinct elements, for an element -

- to be on the second level it must be less than $2^{(N-1)} - 2$ element(s) and greater than 1 element(s).
- to be on the bottommost level it must be less than 0 element(s) and greater than N - 1 element(s). (must be greater than the elements on its branch)

Explanation: An element on the second level must be larger than the root and less than the elements in its subtree. There are $2^{(N-1)} - 2$ elements in the subtree of an element on the second level: half the elements in the tree minus the root, then subtracting off the node itself.
An element on the bottom level must be greater than all elements on the path from itself to the root. A min heap with $2^N - 1$ elements has N levels, so there are N - 1 items above it on a path to the root.

2 Heap Mystery

We are given the following array representing a min-heap where each letter represents a **unique** number. Assume the root of the min-heap is at index zero, i.e. `A` is the root. Note that there is **no** significance of the alphabetical ordering, i.e. just because B precedes C in the alphabet, we do not know if B is less than or greater than C.

Array: [A, B, C, D, E, F, G]

Four unknown operations are then executed on the min-heap. An operation is either a `removeMin` or an `insert`. The resulting state of the min-heap is shown below.

Array: [A, E, B, D, X, F, G]

(a) Determine the operations executed and their appropriate order. The first operation has already been filled in for you!

1. `removeMin()`
2. _____
3. _____
4. _____

Solution:

1. `removeMin()`
2. `insert(X)`
3. `removeMin()`
4. `insert(A)`

Explanation: We know immediately that A was removed. Then, after looking at the final state of the min-heap, we see that C was removed. Then, for A to remain in the min-heap, we see that A must have been inserted afterwards. And, after seeing a new value X in the min-heap, we see that X must have been inserted as well. We just need to determine the relative ordering of the `insert(X)` in between the operations `removeMin()` and `insert(A)`, and we see that the `insert(X)` must go before both.

(b) Fill in the following comparisons with either $>$, $<$, or ? if unknown. We recommend considering which elements were compared to reach the final array.

1. X _____ D
2. X _____ C
3. B _____ C
4. G _____ X

Solution:

1. X ? D

2. X > C
3. B > C
4. G < X

Reasoning:

1. X is never compared to D
2. X must be greater than C since C is removed after X's insertion.
3. B must also be greater than C otherwise the second call to `removeMin` would have removed B
4. X must be greater than G so that it can be "promoted" to the top after the removal of C. It needs to be promoted to the top to land in its new position.

3 Hashing Gone Crazy

For this question, use the following `TA` class for reference.

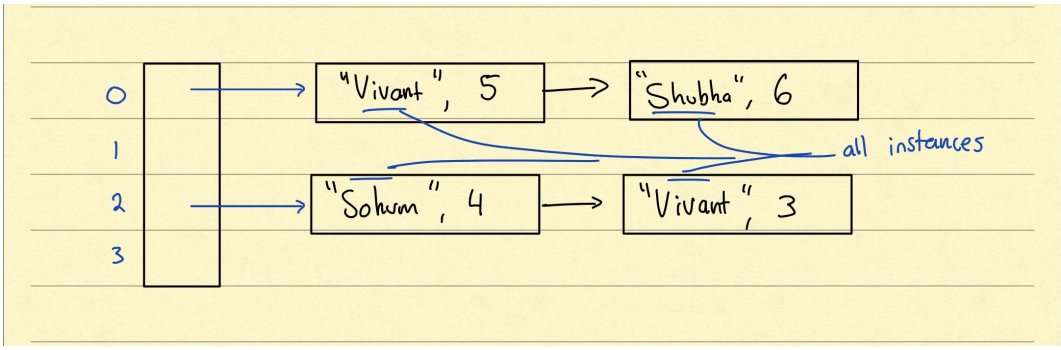
```
public class TA {
    int charisma;
    String name;
    TA(String name, int charisma) {
        this.name = name;
        this.charisma = charisma;
    }
    @Override
    public boolean equals(Object o) {
        TA other = (TA) o;
        return other.name.charAt(0) == this.name.charAt(0);
    }
    @Override
    public int hashCode() {
        return charisma;
    }
}
```

Assume that the `hashCode` of a `TA` object returns `charisma`, and the `equals` method returns `true` if and only if two `TA` objects have the same first letter in their `name`.

Assume that the `ECHashMap` is a `HashMap` implemented with external chaining as depicted in lecture. The `ECHashMap` instance begins at size 4 and, for simplicity, does not resize. Draw the contents of `map` after the executing the insertions below:

```
ECHashMap<TA, Integer> map = new ECHashMap<>();
TA sohum = new TA("Sohum", 10);
TA vivant = new TA("Vivant", 20);
map.put(sohum, 1);
map.put(vivant, 2);

vivant.charisma += 2;
map.put(vivant, 3);

sohum.name = "Vohum";
map.put(vivant, 4);

sohum.charisma += 2;
map.put(sohum, 5);

sohum.name = "Sohum";
TA shubha = new TA("Shubha", 24);
map.put(shubha, 6);
```

Solution:



Explanation:

Line 4: `sohum` has `charisma` value `10`. `10 % 4 = 2`, so `sohum` is placed in bucket 2 with value 1.

`0: [], 1: [], 2: [(sohum, 1)], 3: []`

Line 5: `vivant` is placed in bucket 0 with value 2.

`0: [(vivant, 2)], 1: [], 2: [(sohum, 1)], 3: []`

Line 7: Increasing the `charisma` value of `vivant` does *not* cause it to be rehashed! (This is why modifying objects in a Hashmap is dangerous–it can change the hash-code of your object and make it impossible to find which bucket it belongs to).

Line 8: `vivant` now has `charisma` 4, so bucket 2 also has a node pointing to `vivant`, with value 3. (Note that the two `vivants` refer to the same object).

`0: [(vivant, 2)], 1: [], 2: [(sohum, 1), (vivant, 3)], 3: []`

Line 11, 12: `vivant` with `charisma` 22 hashes to bucket 2. However, since we have changed `sohum`'s name to be `"Vohum"`, `vivant.equals(sohum)` returns true. Since we are hashing a key that is already present in the dictionary according to `.equals`, we replace `sohum`'s old value with the new value, 4.

`0: [(vivant, 2)], 1: [], 2: [(sohum, 4), (vivant, 3)], 3: []`

Line 13, 14: `sohum` with `charisma` 12 hashes to bucket 0. However, since we have changed `sohum`'s name to be `"Vohum"`, `sohum.equals(vivant)` returns true. Since we are hashing a key that is already present in the dictionary according to `.equals`, we replace `vivant`'s old value with the new value, 5.

`0: [(vivant, 5)], 1: [], 2: [(sohum, 4), (vivant, 3)], 3: []`

Line 16, 17, 18: `shuba` hashes to bucket 0. `shuba.equals(vivant)` returns `false`, so we add a new node after `vivant` with value 6.

`0: [(vivant, 5), (shuba, 6)], 1: [], 2: [(sohum, 4), (vivant, 3)], 3: []`

4 Buggy Hash

The following classes may contain a bug in one of its methods. Identify those errors and briefly explain why they are incorrect and in which situations would the bug cause problems.

```
1   class Timezone {
2       String timeZone; // "PST", "EST" etc.
3       boolean dayLight;
4       String location;
5       ...
6       public  int currentTime() {
7           // return the current time in that time zone
8       }
9       public int hashCode() {
10          return currentTime();
11      }
12      public boolean equals(Object o) {
13          Timezone tz = (Timezone) o;
14          return tz.timeZone.equals(timeZone);
15      }
16  }
```

Solution:

Although equal objects will have the same hashcode, but the problem here is that `hashCode()` is not deterministic. This may result in weird behaviors (e.g. the element getting lost) when we try to put or access elements.

```
1   class Course {
2       int courseCode;
3       int yearOffered;
4       String[] staff;
5       ...
6       public int hashCode() {
7           return yearOffered + courseCode;
8       }
9       public boolean equals(Object o) {
10          Course c = (Course) o;
11          return c.courseCode == courseCode;
12      }
13  }
```

Solution: The problem with this `hashCode()` is that not all equal objects have the same hashcode. This may produce unexpected behavior, e.g. multiple "equal" objects may be exist in different buckets in the `HashMap`, the `containsKey` operation may return false, etc. One key thing to remember is that when we override the `equals()` method, we have to also override the `hashCode()` method to ensure equal objects have the same `hashCode`.