# Heaps & Hashing

Discussion 10

# Announcements

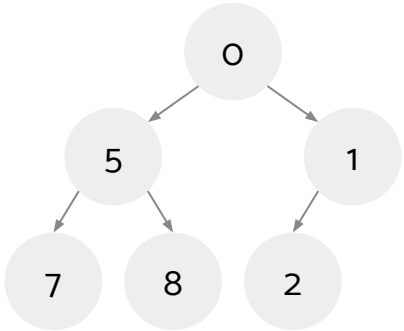
- Homework 6 due Tuesday 03/29
- Week 9 Survey due Tuesday 03/29
- Project 2 due Friday April 04/01
- Test 2 Review Sessions
  - Wednesday 03/30
  - Friday 04/01
- Test 2 on Wednesday 04/06

# Review

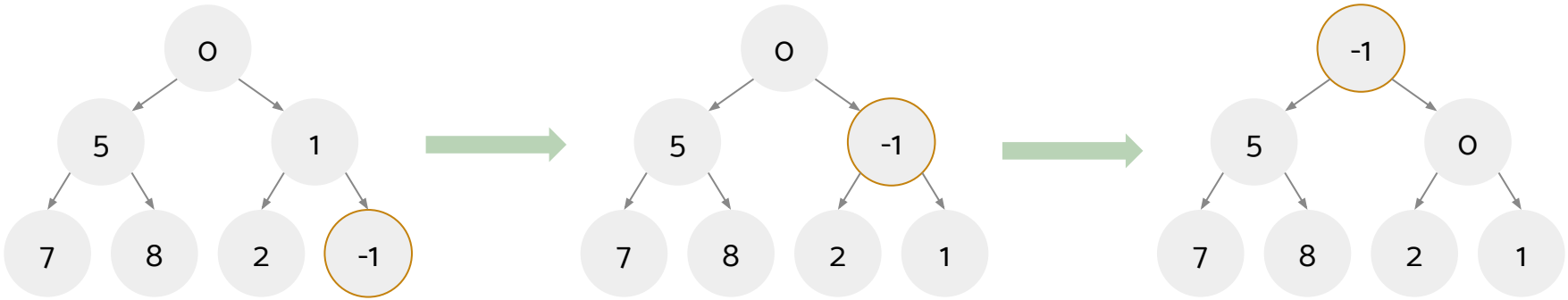
# Heaps

**Heaps** are special trees that follow a few basic rules:

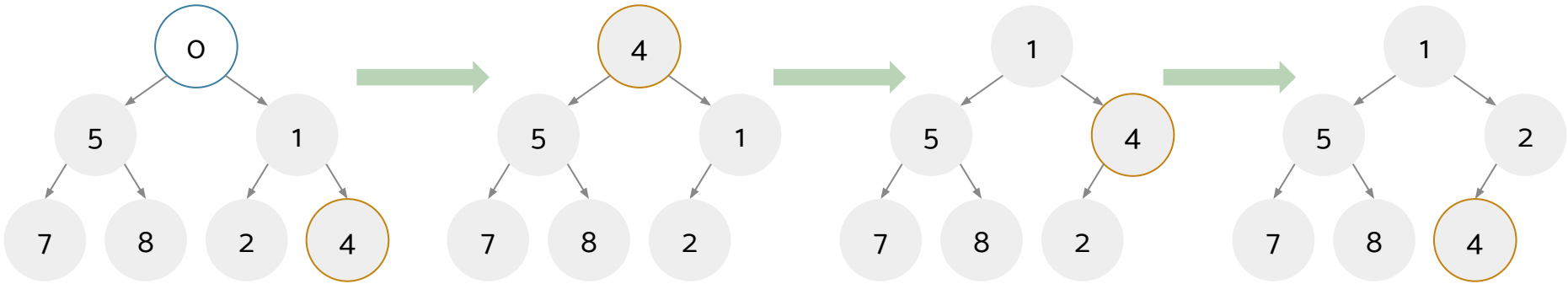
1. Heaps are complete - the only empty parts of a heap are in the bottom row, to the right
2. In a min-heap, each node must be *smaller* than all of its child nodes. The opposite is true for max-heaps.



# Insertion into Heaps



# Deletion from Heaps

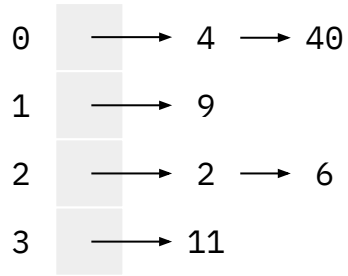


# Hashing

**Hash functions** are functions that represent an object using an integer. We use them to figure out which bucket of our hashset the item should go in.

Once we have a hash for our object we use mod to find out which bucket it goes into.

In each bucket, we deal with having lots of items by chaining the items and using .equals to find what we are looking for.




**It is important that your `.equals()` function matches the result of comparing hashcodes - if two items are equal, they must also have the same hashcode**

# Open Addressing

An alternative to externally chained hashmaps. When there is a collision in bucket `h(k)`, use another box using the formula `h(k) + f(m)` for some function `f`.

**Linear Probing** → `h(k) + m, h(k) + 2m, h(k) + 3m, ...`
**Quadratic Probing** → `h(k) + 1 * m, h(k) + 4 * m, h(k) + 9 * m, ...`
**Double Hashing** → `h(k) + h'(k), h(k) + 2h'(k), h(k) + 3h'(k), ...`

# CS 61B Spring 2022

# Heaps & Hashing

Exam Prep Discussion 10: March 21, 2022

## 1 Fill in the Blanks

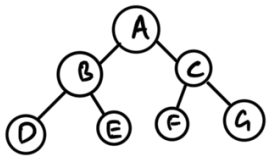
Fill in the following blanks related to min-heaps. Let `N` is the number of elements in the min-heap. For the entirety of this question, assume the elements in the min-heap are **distinct**.

1. `removeMin` has a best case runtime of $\Theta(1)$ and a worst case runtime of $\Theta(\log N)$.
2. `insert` has a best case runtime of $\Theta(1)$ and a worst case runtime of $\Theta(\log N)$.
3. A preorder or level order traversal on a min-heap *may* output the elements in sorted order. Assume there are at least 3 elements in the min-heap.
   ↳ must print root first
4. The fourth smallest element in a min-heap with 1000 elements can appear in 14 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 $2^{N-1} - 2$ element(s) and greater than 1 element(s). → other subtree, root, itself → root
   - to be on the bottommost level it must be less than 0 element(s) and greater than $N - 1$ element(s). → no req → branch

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

# 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.

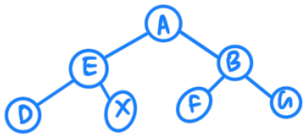


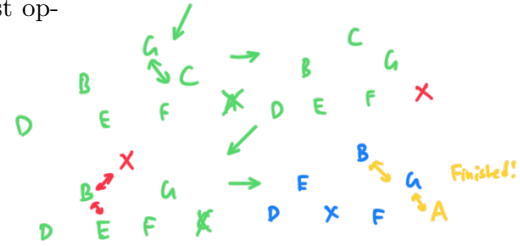

Array: [A, B, C, D, E, F, G]

**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.

have to remove, C at not have to insert, also insert

Array: [A, E, B, D, X, F, G] Note C is missing, X has been added both A (from first removeMin)




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

1. `removeMin()`
2. insert(X)
3. removeMin() → after, X gets to the root, then bubbles
4. insert(A)

(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

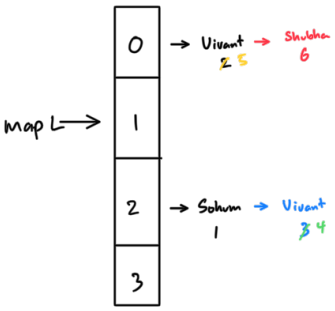
Why C has to be removed after inserting X:

Want to shift B to the right side, so promote it over G

F and G are on wrong sides!!

# 3 Hashing Gone Crazy

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




```
1   public class TA {
2       int charisma;
3       String name;
4       TA(String name, int charisma) {
5           this.name = name;
6           this.charisma = charisma;
7       }
8       @Override
9       public boolean equals(Object o) {
10          TA other = (TA) o;
11          return other.name.charAt(0) == this.name.charAt(0);
12      }
13      @Override
14      public int hashCode() {
15          return charisma;
16      }
17  }
```

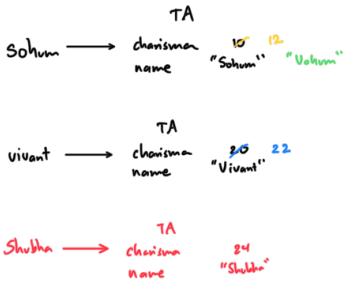
Does first letter match?

Bad! Being equal should imply hashcodes match up.

Charisma is hashcode

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:




```
1   ECHashMap<TA, Integer> map = new ECHashMap<>();
2   TA sohum = new TA("Sohum", 10);
3   TA vivant = new TA("Vivant", 20);
4   map.put(sohum, 1);
5   map.put(vivant, 2);
6
7   vivant.charisma += 2;
8   map.put(vivant, 3);
9
10  sohum.name = "Vohum";
11  map.put(vivant, 4);
12
13  sohum.charisma += 2;
14  map.put(sohum, 5);
15
16  sohum.name = "Sohum";
17  TA shubha = new TA("Shubha", 24);
18  map.put(shubha, 6);
```

hashcode: 10 % 4 = 2

hashcode: 20 % 4 = 0

hashcode: 22 % 4 = 2

hashcode: 12 % 4 = 0 Remember it is now "vohum" which equals vivant

# 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  }
```

Not deterministic

```
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  }
```

does not imply

Equals ! → Hashcode matching

Overriding Equals? Override hashcode!