CSM 61B Fall 2017

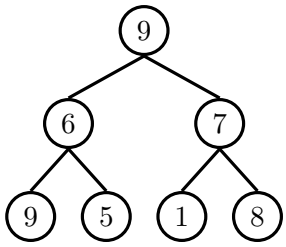
Heaps & Hashing

Mentoring 9: October 23, 2017

1 Min-Heapify This

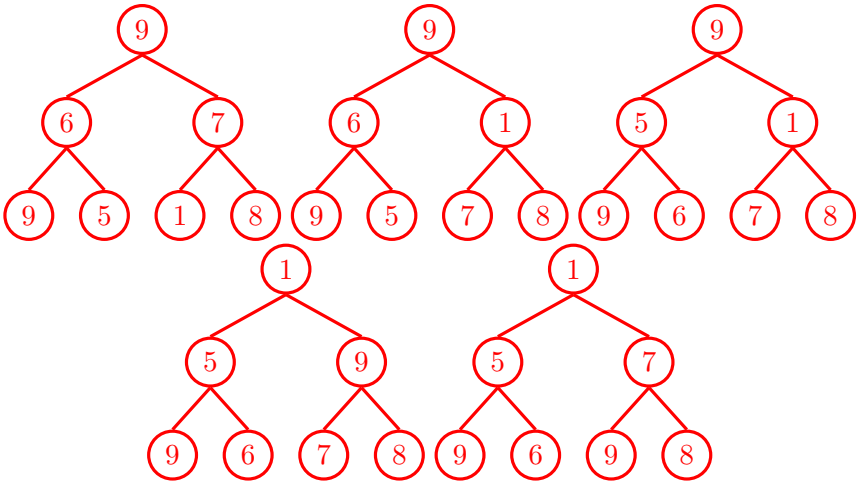
1.1 In general, there are 4 ways to heapify. Which 2 ways actually work?

- Level order, bubbling up
- Level order, bubbling down
- Reverse level order, bubbling up
- Reverse level order, bubbling down

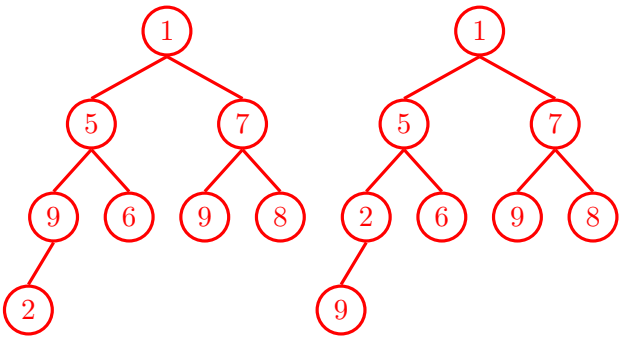


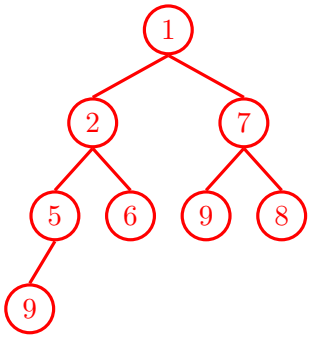
Only level order, bubbling up and reverse level order, bubbling down work as they maintain heap invariant. Namely, that every node is either larger (in a max heap) or smaller (in a min heap) than all of its children.

1.2 (a) Show the heapification of the tree.

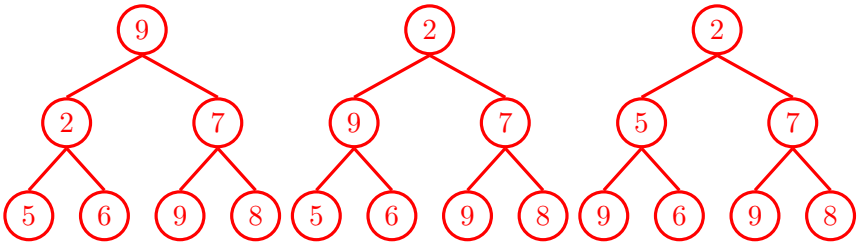


(b) Now, insert the value 2.





(c) Finally, remove the value 1.



2 Motivation

2.1 (a) In the worst case, how long does it take to index into a linked list?

$\Theta(N)$

(b) In the worst case, how long does it take to index into an array?

$\Theta(1)$

(c) In the worst case, how long does it take to insert into a linked list?

$\Theta(N)$, where N is the length of the linked list.

(d) Assuming there's space, how long does it take to put a element in an array?

$\Theta(1)$

(e) What if we assume there is no more space in the array?

$\Theta(N)$ to copy over N elements into the new array.

(f) Given what we know about linked lists and arrays, how could we build a data structure with efficient access and efficient insertion?

If you know in advance how large your data structure is, arrays are faster than linked lists in insertion, mutation, etc. However, if the array needs to expand frequently then things get expensive. But there are ways to amortize the cost of resizing with `ArrayLists`, for example.

- An array of linked lists will offer constant look up to a certain linked list, and adding to the front of that linked list will also be constant. This is a `HashMap`.
- Objects with rows and columns (like a chessboard) where we wish to randomly index into exact position and where the board is of a fixed size
- Arguments to a java program: `String[] args`. Using a resizing `List` in this scenario doesn't necessarily make things better since the arguments to a program don't change once we start the program.

3 Hash Table Basics

3.1
```
public class BadHashMap<K, V> implements Map<K, V> {
    private V[] array;
    public void put(K key, V value) {
        this.array[key.hashCode() % this.array.length] = value;
    }
}
```

```
interface Map<K, V> {
    boolean containsKey(K key);
    V get(K key);
    void put(K key, V value);
    int size();
}
```

(a) Why do we use the `%` (modulo) operator?

Allows us to take a large **`int`** and turn it into an index into the `array`.

(b) What are collisions? What data structure can we use to address them?

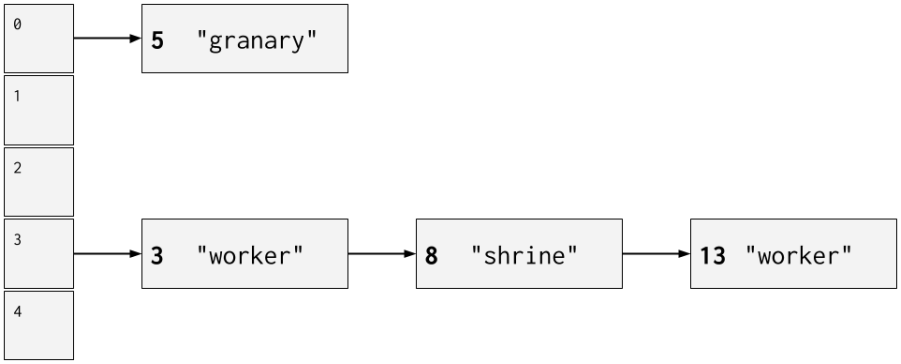
Collisions occur when two keys map to the same bucket, or index, in the `array`. We can use external chaining with a linked list, for example, to resolve the conflicts. Each bucket can be associated with more than one key.

(c) Why is this a bad `HashMap`?

`BadHashMap` doesn't handle collisions. If a given key produces a collision, the value will be completely overwritten rather than stored alongside the value currently there.

3.2 (a) Draw the diagram that results from the following operations on a Java `HashMap`. `Integer::hashCode` returns the integer's value.

```
put(3, "monument");
put(8, "shrine");
put(3, "worker");
put(5, "granary");
put(13, "worker");
```



`"worker"` replaces `"monument"` as their keys are the same. Each `put` must iterate through the entire external chain to ensure that a key-update is not necessary.

(b) Suppose a resize occurs, doubling the array to size 10. What changes?

The value of `"shrine"` and `"granary"` will move. Specifically, the new length of the array is 10. A key of 8 will force `"shrine"` to be placed in the 8th index. A key of 5 will move `"granary"` to the 5th index. Everything else will remain the same.

4 Hash Codes *Extra Practice*

4.1 What does it mean for a hashcode to be valid?

- Consistency: Whenever it is invoked on the same object more than once, the `hashCode` method must consistently returns the same integer if nothing about the object changes.
- Equality constraint: If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

4.2 Which of the following hashcodes are valid? Good?

```
class Point {
    private int x, y;
    private static int count = 0;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
        count += 1;
    }
}
```

(a)
```
public void hashCode() {
    System.out.print(this.x + this.y);
}
```

Invalid. Return type should be **int**. The method should override Java's `hashCode` method, but since it has the same signature with different return types, this would be a compile-time error

(b)
```
public int hashCode() {
    Random random = new Random();
    return random.nextInt();
}
```

Invalid. Not consistent. This method would compile because it does return an **int**. However, this does not obey the principle of consistency about returning the same integer every time, and so the `HashMap` would

not work as expected. If we store an object at key 1, the next time around the object might have key 2, and we would be unable to find the object.

(c)
```
public int hashCode() {
    return this.x + this.y;
}
```

Valid, but certain inputs may cause a significant number of collisions. This method would work, since it obeys all bullets under the contract. However, this will cause many collisions. All points that sum up to the same number will be mapped to the same `hashCode` such as $(5, 3), (4, 4), (2, 6), (6, 2), (1, 7), (7, 1), \ldots$

(d)
```
public int hashCode() {
    return count;
}
```

Invalid. Not consistent. This method would not error, because it does return an **`int`**, but it does not obey the principle of consistency. `count` is a static variable. Consider what happens when we create a **`new`** `Point(3, 4)` and call the `hashCode` method on it the first time. We would get 1. If we then create a **`new`** `Point(1, 2)`, calling the `hashCode` method on the earlier `Point(3, 4)` would now return 2, even though nothing about the original `Point` object changed.

(e)
```
public int hashCode() {
    return 4;
}
```

Valid, but causes collisions on all inputs. While this method is valid, it collides on any input.