CSM 61B Fall 2020

Heaps and Hashing

Mentoring 9: October 19-23, 2020

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.

Meta: Students often ask about the runtime of these methods. The specific runtime for heapification is hard to prove, but specifically level order, bubbling up will take $O(N \log(N))$ and reverse level order, bubbling down takes $O(N)$, so reverse level order + bubbling down is faster. Intuition on why it is faster: the majority of the nodes are near the bottom of the tree, so a lot of the nodes bubble down really quickly.

1.2 Are the values in an array-based min-heap sorted in ascending order?

Not necessarily. We can have higher or lower priority items loaded on one branch of the tree.

1.3 Is an array that is sorted in descending order also a max-oriented heap?

True, the heap invariant holds.

2 K Largest Items

2.1 The largest item in a heap must appear in position 1, and the second largest must appear in position 2 or 3. Give the list of positions in a heap where the kth largest can appear for $k \in \{2, 3, 4\}$. Assume values are distinct.

$k = 2$ can be in $\{2, 3\}$. $k = 3$ can be in $\{2 \dots 7\}$. $k = 4$ can be in $\{2 \dots 15\}$.

Consider complete binary trees with the largest values contained on one branch of the tree for a lower bound and consider how far the kth element can be from the root for an upper bound.

3 Ls for LinkedLists

3.1 (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?

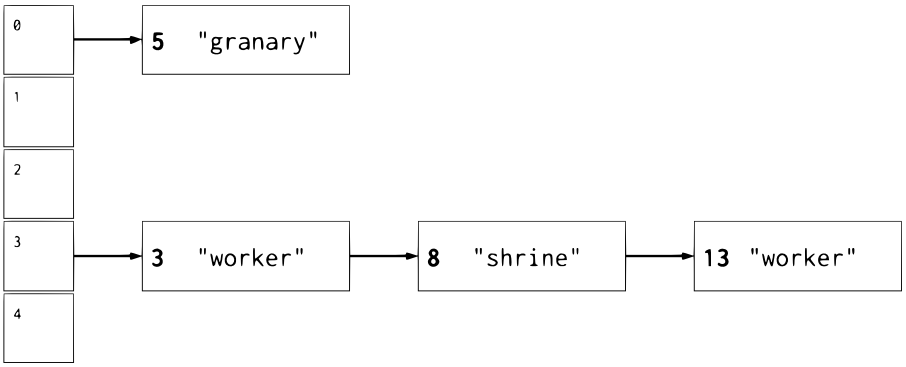
If you know in advance how large your data structure is, arrays are faster than linked lists in insertion, mutation, etc. However, if the array needs to expand frequently then things get expensive. But there are ways to amortize the cost of resizing with `ArrayLists`, for example.

- 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.

4 Hashing Practice

4.1 (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.

5 Hash Codes

There is a problem with each `hashCode()` method below (correctness, distribution, efficiency). Assume there are no problems with the correctness of `equals()`.

5.1
```
class Person {
    Long id;
    String name;
    Integer age;
    public int hashCode() {
        return id.hashCode() + name.hashCode() + age.hashCode();
    }
    public boolean equals(Object o) {
        Person p = (Person) o;
        return p.id == id;
    }
}
```

Incorrect: Persons that are `equals()` do not necessarily have the same `hashCode()`.

5.2
```
class Phonebook {
    List<Human> humans;
    public int hashCode() {
        int h = 0;
        for (Human human : humans) {
            // Assume Human::hashcode is correct
            h = (h + human.hashCode()) % 509;
        }
        return h;
    }
    public boolean equals(Object o) {
        Phonebook p = (Phonebook) o;
        return p.humans.equals(humans);
    }
}
```

Poor distribution: The `hashCode()` only distributes numbers between -508 and 508, which is an inefficient use of the full integer range and will cause more collisions than necessary.

5.3
```
class PokeTime {
    int startTime;
    int duration;
```

```
    public int getCurrentTime() {
        // Gets the current system clock time
    }
    public int hashCode() {
        return 1021 * (startTime + 1021 * duration + getCurrentTime());
    }
    public boolean equals(Object o) {
        PokeTime p = (PokeTime) o;
        return p.startTime == startTime && p.duration == duration;
    }
}
```

Incorrect: The `hashCode()` is non-deterministic.