CS 61B Spring 2025

Iterators, Asymptotics

Discussion 04: February 17, 2025

1 OHQueue

Meshan is designing the new 61B Office Hours Queue. The code below for **`OHRequest`** represents a single request. It has a reference to the **`next`** request. **`description`** and **`name`** contain the description of the bug and name of the person on the queue, and **`isSetup`** marks the ticket as being a setup issue or not.

```
public class OHRequest {
    public String description;
    public String name;
    public boolean isSetup;
    public OHRequest next;

    public OHRequest(String description, String name, boolean isSetup, OHRequest next) {
        this.description = description;
        this.name = name;
        this.isSetup = isSetup;
        this.next = next;
    }
}
```

(a) Create a class **OHIterator** that implements an **Iterator** over **OHRequest**s and only returns requests with good descriptions (using the **isGood** function). Our **OHIterator**'s constructor takes in an **OHRequest** that represents the first **OHRequest** on the queue. If we run out of office hour requests, we should throw a **NoSuchElementException** when our iterator tries to get another request, like so:

```
  throw new NoSuchElementException();
public class OHIterator ____________________________________________________ {
    private OHRequest curr;

    public OHIterator(OHRequest request) {

        ________________________________________;

    }

    public static boolean isGood(String description) { return description.length() >= 5; }

    @Override
    _________________ _________________ ______________________________ {

        while (_____________________________________________________________) {

            ___________________________________;

        }

        ___________________________________;

    }

    @Override
    _________________ _________________ ______________________________ {

        if (___________________________________) {

            throw __________ ______________________________________________;

        }

        ___________________________________;

        ___________________________________;

        ___________________________________;

    }
}
```

Solution:

```
public class OHIterator implements Iterator<OHRequest> {
    private OHRequest curr;

    public OHIterator(OHRequest request) {
        curr = request;
    }

    public static boolean isGood(String description) { return description.length() >= 5; }

    @Override
    public boolean hasNext() {
        while (curr != null && !isGood(curr.description)) {
            curr = curr.next;
        }
        return curr != null;
    }

    @Override
    public OHRequest next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        OHRequest temp = curr;
        curr = curr.next;
        return temp;
    }
}
```

Explanation: The `OHRequest` object `queue` passed into `OHIterator`'s constructor represents the first `OHRequest` on the queue. Initializing `curr` to `queue` in the constructor allows our `OHIterator` to start at this first request. Since `OHIterator` implements an Iterator over `OHRequests`, we must provide implementations for the interface methods `hasNext()` and `next()`. The `hasNext()` method handles checking whether there are more `OHRequests`. However, we only want requests with good (as defined by `isGood`) descriptions, so we must check the descriptions of each `OHRequest` and skip over the ones with bad descriptions before determining whether there are `OHRequests` left.

(b) Define a class `OHQueue` below: we want our `OHQueue` to be `Iterable` so that we can process `OHRequest` objects with good descriptions. Our constructor takes in the first `OHRequest` object on the queue.

```
public class OHQueue ____________________________________________________ {
    private OHRequest request;
    public OHQueue(OHRequest request) {

        ________________________________________;
    }

    @Override
```

```
    ________________ ________________ ______________________________ {

        ________________________________________;
    }
}
```

Solution:

```
public class OHQueue implements Iterable<OHRequest> {
    private OHRequest request;

    public OHQueue(OHRequest request) {
        this.request = request;
    }

    @Override
    public Iterator<OHRequest> iterator() {
        return new OHIterator(request);
    }
}
```

Explanation: If we want our `OHQueue` to be `Iterable`, `OHQueue` has to implement the interface `Iterable`. A condition of this is implementing the methods of the interface (which in the case of `Iterable`, is the `iterator()` method). As our `OHQueue` processes `OHRequest` objects, `iterator()` in `OHQueue` should return an `OHIterator` over `OHRequest` objects.

(c) Suppose we notice a bug in our office hours system: if a ticket's description contains the words "thank u", it is put on the queue twice. To combat this, we'd like to adjust our implementation of `OHIterator`'s `next()`.

If the current item's description contains the words "thank u", it should skip the next item on the queue, because we know the next item is an accidental duplicate from our buggy system. As an example, if there were 4 `OHRequest` objects on the queue with descriptions `["thank u", "thank u", "im bored", "help me"]`, calls to `next()` should return the 0th, 2nd, and 3rd `OHRequest` objects on the queue.

To check if a `String s` contains the substring "thank u", you can use: `s.contains("thank u")`

```
@Override
_____________ ___________ _____________ {

    if (___________________________________) {

            throw _________ ________________________________________________;

    }
    ___________________________________;

    ___________________________________;

    if (_________________________________________) {

        ____________________;
```

```
    }
    return ______________________;
}
```

Solution:

```
@Override
public OHRequest next() {
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    OHRequest temp = curr;
    curr = curr.next;
    if (temp.description.contains("thank u")) {
        curr = curr.next;
    }
    return temp;
}
```

(d) Now assume the **OHQueue** uses the modified **OHIterator** as its iterator. Fill in the blanks to print only the names of tickets from the queue beginning at **s1** with good descriptions, skipping over duplicate descriptions that contain "thank u". What would be printed after we run the **main** method?

```
public static void main(String[] args) {
    OHRequest s5 = new OHRequest("I deleted all of my files, thank u", "Elana", true, null);
    OHRequest s4 = new OHRequest("conceptual: what is Java", "Mihir", false, s5);
    OHRequest s3 = new OHRequest("git: I never did lab 1", "Kevin", true, s4);
    OHRequest s2 = new OHRequest("help", "Ethan", false, s3);
    OHRequest s1 = new OHRequest("no I haven't tried the debugger", "Ashley", false, s2);

    OHQueue q = ____________________________________;

    for (_____________________________________________) {

        ___________________________________________;
    }
}
```

Solution:

```
public static void main(String[] args) {
    OHRequest s5 = new OHRequest("I deleted all of my files, thank u", "Elana", true, null);
    OHRequest s4 = new OHRequest("conceptual: what is Java", "Mihir", false, s5);
    OHRequest s3 = new OHRequest("git: I never did lab 1", "Kevin", true, s4);
    OHRequest s2 = new OHRequest("help", "Ethan", false, s3);
    OHRequest s1 = new OHRequest("no I haven't tried the debugger", "Ashley", false, s2);

    OHQueue q = new OHQueue(s1);
    for (OHRequest r: q) {
        System.out.println(r.name);
    }
}
```

Overall, we print:

```
Ashley
Kevin
Mihir
Elana
```

2 Asymptotics

(a) Say we have a function **findMax** that iterates through an unsorted int array one time and returns the maximum element found in that array. Give the tightest lower and upper bounds ($\Omega(\cdot)$ and $O(\cdot)$) of **findMax** in terms of **N**, the length of the array. Is it possible to define a $\Theta(\cdot)$ bound for **findMax**?

Solution: Lower bound: $\Omega(N)$, Upper bound: $O(N)$

Because the array is unsorted, we don't know where the max will be, so we have to iterate through the entire array to ensure that we find the true max. Therefore, we know that we can never go faster than linear time with respect to the length of the array. Since the function is both lower and upper bounded by N, we can say that the function is theta-bounded by N as well ($\Theta(N)$).

(b) Give the worst case and best case runtime in terms of M and N. Assume **ping** runs in $\Theta(1)$ and returns an **int**.

```
for (int i = N; i > 0; i--) {
    for (int j = 0; j <= M; j++) {
        if (ping(i, j) > 64) { break; }
    }
}
```

Solution: Worst: $\Theta(MN)$, Best: $\Theta(N)$ We repeat the outer loop N times, no matter what. For the inner loop, the amount of times we repeat it depends on the result of **ping**. In the best case, it returns **true** immediately, such that we'll only ever look at the inner loop once and then break the inner loop. In the worst case, **ping** is always **false** and we complete the inner loop **M** times for every value of **N** in the outer loop.

(c) Below we have a function that returns **true** if every **int** has a duplicate in the array, and **false** if there is any unique int in the array. Assume **sort(array)** is in $\Theta(N \log N)$ and returns **array** sorted.

```
public static boolean noUniques(int[] array) {
    array = sort(array);
    int N = array.length;
    for (int i = 0; i < N; i += 1) {
        boolean hasDuplicate = false;
        for (int j = 0; j < N; j += 1) {
            if (i != j && array[i] == array[j]) {
                hasDuplicate = true;
            }
        }
        if (!hasDuplicate) return false;
    }
    return true;
}
```

Give the worst case and best case runtime where $N =$ **array.length**.

Notice that we call `sort` at the beginning of the function, which we are told runs in $\Theta(N \log N)$.

First, we consider the best case. We notice that if `hasDuplicate` is false after the inner loop (i.e. `!hasDuplicate` has truth value `true`) we can exit the `for` loop early via the return statement on line 11. Thus, the best case is when we never set `hasDuplicate` to be `true` during the first time we run the inner loop. In this case, we can return after only looping through the array once, giving us $\Theta(N \log N + N) = \Theta(N \log N)$.

For the worst case, we notice that if `hasDuplicate` is always set to `true` by the inner loop, we never return on line 11. Thus, we consider the worst case where `hasDuplicate` is always set to `true` in every loop, forcing us to have to loop fully through both the inner and outer loop. One such input is an array of all the same integer! Since we have to fully loop through both loops, our worst-case runtime is $\Theta(N \log N + N^2) = \Theta(N^2)$.