# CS 61B Spring 2018

# Midterm 2 Guerrilla Section

March 17, 2018

**Instructions**

Form a small group. Start on the first problem. Check off with a helper or discuss your *solution process* with another group once everyone understands *how to solve* the first problem and then repeat for the second problem ...

You may not move to the next problem until you check off or discuss with another group and *everyone understands why the solution is what it is*. You may use any course resources at your disposal: the purpose of this review session is to have everyone learning together as a group.

## 1 Asymptotic Approach

1.1 Give a tight asymptotic runtime bound on `foo(n)`.

```
1  int foo(int n) {
2      if (n == 0) {
3          return 0;
4      }
5      baz(n);
6      return foo(n / 3) + foo(n / 3) + foo(n / 3);
7  }
8
9  int baz(int n) {
10     for (int i = 0; i < n; i += 1) {
11         System.out.println("Help me! I'm trapped in a loop!");
12     }
13     return n;
14 }
```

$\Theta(n \log n)$. Each `baz` call will run for $n$ time and then each `foo` call will call 3 additional `foo` calls each with 1/3 of the initial run time. We can draw a tree to represent the runtime; each node will represent a `foo` call and will spawn three more calls / nodes each with a third of the parent's runtime. Each level of the tree will take $\Theta(n)$ time total and the tree will have height $\Theta(\log n)$.

1.2 For each of the following methods, give a tight asymptotic runtime bound with respect to $N$.

(a)
```
public void mystery1(int N) {
    for (int i = 0; i < N; i += 1) {
        for (int j = 0; j < N; j += 1) {
            i = i * 2;
            j = j * 2;
        }
    }
}
```

$\Theta(\log N)$. Both $i$ and $j$ are doubled in the inner loop so it takes $\log N$ time for both $i$ and $j$ to be incremented to $N$. The outer loop only runs once; the inner loop $\Theta(\log N)$ times.

(b)
```
public void mystery2(int N) {
    for (int i = N; i > 0; i = i / 2) {
        for (int j = 0; j < i * 2; j += 1) {
            System.out.println("Hello World");
        }
    }
}
```

$\Theta(N)$. Doubling the $i$ bound in the inner loop does not change the asymptotic nature of the runtime as opposed to if we did not double it (we can generally ignore coefficients). The number of print calls are then $2N + N + \frac{N}{2} + \cdots + 2 = \Theta(N)$.

(c)
```
public void mystery3(int N) {
    for (int i = N; i > 0; i = i / 2) {
        for (int j = 0; j < i * i; j += 1) {
            System.out.println("Hello World");
        }
    }
}
```

$\Theta(N^2)$ By squaring $i$ in the inner loop, the number of print calls will now be this sum: $N^2 + (N/2)^2 + (N/4)^2 \cdots + 1 = \Theta(N^2)$.

(d)
```
public void mystery4(int N) {
    int i = 1, s = 1;
    while (s <= N) {
        i += 1
        s = s + i;
        System.out.println(s);
     }
}
```

$\Theta(\sqrt{N})$. We know that each iteration of the for loop takes constant time, so we need to figure out how many iterations of the for loop occur; let this mount be $k$. At every step, $i$ increments by 1 so we are essentially accumulating in $s = 1 + 2 + 3 + \cdots + k$ until $s > N$. Therefore, we want to solve for $1 + 2 + 3 + \cdots + k = N$ which approximately simplifies to $\frac{k^2}{2} = N$, which we solve to obtain $k = \sqrt{2N} = \sqrt{2}\sqrt{N}$. Our runtime is determined by $k$ which is therefore $\Theta(\sqrt{N})$.

# 2 Iterator or Iterable?

2.1 Implement the `Filter` class such that its `main` method correctly prints out the even numbers in the given collection: `0 20 14 50 66`. Assume we have already imported `java.util.*` in each file.

```
public interface FilterCondition<T> {
    /** Returns true if the item meets a certain condition (is even, etc.). */
    boolean eval(T item);
}
```

```
public class EvenCondition implements FilterCondition<Integer> {
    public boolean eval(Integer i) {
        return i % 2 == 0;
    }
}
```

```java
public class Filter implements Iterable<Integer> {
    Iterable<Integer> iterable;
    FilterCondition<Integer> cond;

    public Filter(Iterable<Integer> it, FilterCondition<Integer> cond) {
        iterable = it;
        this.cond = cond;
    }

    public Iterator<Integer> iterator() {
        return new FilterIterator();
    }

    private class FilterIterator implements Iterator<Integer> {
        Iterator<Integer> integerIterator;
        Integer toReturn;

        private FilterIterator() {
            integerIterator = iterable.iterator();
            this.next();
        }

        public Integer next() {
            Integer processed = toReturn;
            toReturn = null;
            while (integerIterator.hasNext()) {
                int x = integerIterator.next();
                if (cond.eval(x)) {
                    toReturn = x;
                    break;
                }
            }
            return processed;
        }

        public boolean hasNext() {
            return toReturn != null;
        }
    }

    public static void main(String[] args) {
        List<Integer> arr = Arrays.asList(new Integer[]{0, 11, 20, 13, 14, 50, 66});
        for (int i : new Filter(arr, new EvenCondition())) {
            System.out.print(i);
        }
    }
}
```

# 3 Stacks of Fun

3.1 An SQueue is a queue implemented using two stacks.

```
1  public class SQueue {
2      private Stack in, out;
3
4      public SQueue() {
5          in = new Stack();
6          out = new Stack();
7      }
8
9      public void enqueue(int item) {
10         in.push(item);
11     }
12
13     public void dequeue() {
14         if (out.isEmpty()) {
15             while (!in.isEmpty()) {
16               out.push(in.pop());
17             }
18         }
19         return out.pop();
20     }
21 }
```

Now, suppose we construct an SQueue and enqueue 100 items.

(a) How many calls to push and pop result from the next call to dequeue?

201. 100 pops and 100 pushes in the while loop and one last pop that we return.

(b) How many calls to push and pop result from each of the next 99 calls to dequeue?

1 per call. out is not empty so we just call pop once in each call.

(c) How many calls to push and pop (total) were required to dequeue 100 elements? How many operations is this per element dequeued?

300 operations, or 3 per dequeue

(d) What is the worst-case time to dequeue an item from an SQueue containing $N$ elements? What is the runtime in the best case? Answer using $\Theta(\cdot)$ notation. You may assume that both push and pop run in $\Theta(1)$.

Worst: $\Theta(N)$, Best: $\Theta(1)$

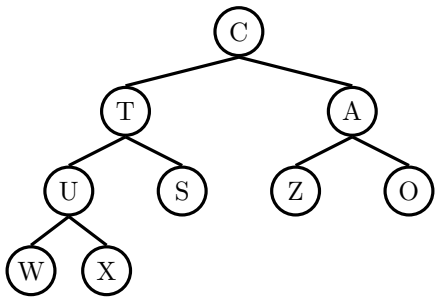
(e) What is the amortized time to dequeue an item from an SQueue containing $N$ elements? Again, answer using $\Theta(\cdot)$ notation.

$\Theta(1)$. In part (c) we establish that we average 3 constant time calls per `dequeue` which is $\Theta(1)$.

# 4 Greetings, Tree Traveler

4.1 Draw a full binary tree that has the following pre-order and post-order traversals. Each node should contain exactly one letter. A full binary tree is a tree such that all nodes except leaf nodes have exactly 2 children.

- Pre-order: C T U W X S A Z O
- Post-order: W X U S T Z O A C




(a) What is the in-order traversal of this tree?

W U X T S C Z A O

(b) Can a tree have the same in-order and post-order traversals? If so, what can you say about the tree?

Yes. The tree can only have left children. (In-order gives left-root-right, post-order gives left-right-root).

(c) What about a tree with the same pre-order and post-order traversals?

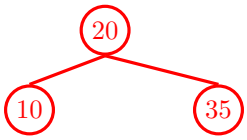
Yes, but only if the tree has one element or is empty. (Pre-order gives root-left-right, post-order gives left-right-root)

4.2 Insert the following list into a 2-3 tree: `20, 10, 35, 40, 50, 5, 25, 15, 30, 60`.

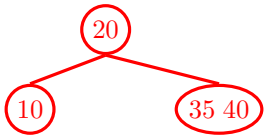
Inserting 20:


Inserting 10:

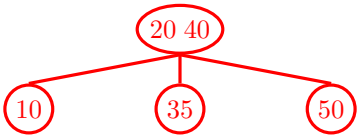

Inserting 35:



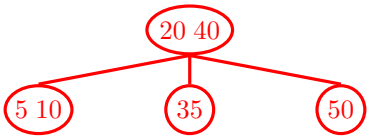
Inserting 40:



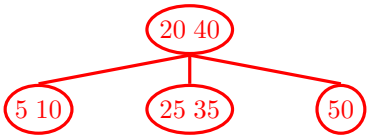

Inserting 50:



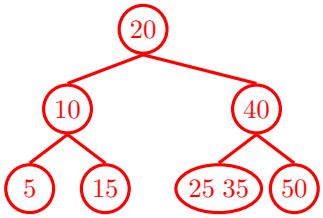

Inserting 5:



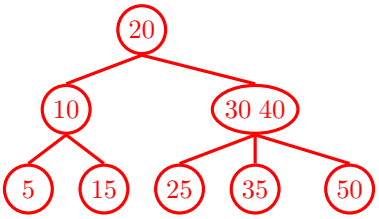

Inserting 25:



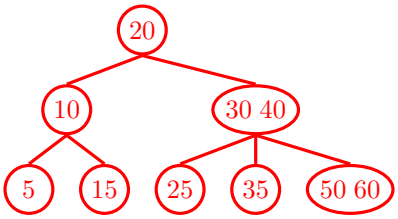

Inserting 15:



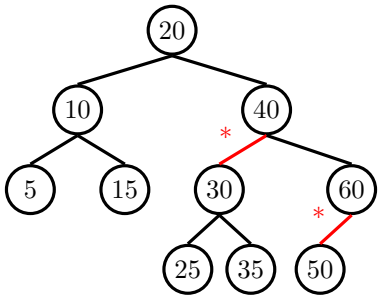

Inserting 30:




Inserting 60:



4.3 Draw the corresponding left-leaning red-black tree.




4.4 For each scenario below, indicate whether the node's edge to its parent is **red**, **black**, or **either** red or black. If it could be either color, explain.

(a) The largest value in a tree with more than one node.

The largest value is always a right child, so the edge must be black.

(b) The smallest value in a tree with more than one node.

Either. A two-node 2-3 tree has a red smallest child. A three-node 2-3 tree has a black smallest child.

(c) A node with a red *grandparent* edge.

Black, since we cannot have two consecutive red edges.

(d) A node whose children are the same color.

A node with two children of the same color (which must both be black) can have a parent edge of either red or black.

(e) The last node inserted, after any rotations and color flips.

When the insertion operation for a BST has completed, the node may have either red or black edge to its parent.

# 5 Quick Union

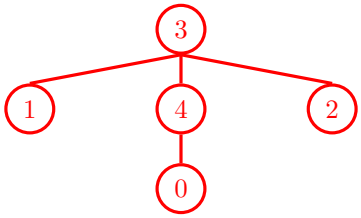
The quick union data structure is handles set union and membership operations.

- `connect(a, b)`: connects the set of `a` to the set of `b`.
- `isConnected(a, b)`: returns true if `a` and `b` are in the same set.

Internally, quick union sets are trees. Sets can be connected by adding one set's tree to the root of another set's tree. Weighted quick union data structures improve upon quick union data structures by always adding the shorter tree to the root of the taller tree during `connect` operations.

5.1 Draw the Weighted Quick Union object that results after the following method calls.

```
connect(1, 3)
connect(0, 4)
connect(0, 1)
connect(0, 2)
```




There are many possible solutions. When connecting sets, we only connect at the root.

5.2 (a) What is the worst way to connect the list `1, 2, 3, 4, 5`? Give an answer in the form of a series of calls to the `connect` method.

```
connect(1,2)
connect(2,3)
connect(3,4)
connect(4,5)
```

This will create a tree of height 4 with a branching factor of 1 (i.e. a linked list).

(b) Assuming a single node has a height of 0, what is the shortest and tallest height possible for a quick union object with $N$ elements?

Shortest: 1. For example, the tree created by the following sequence:

```
connect(1,2), connect(1,3), connect(1,4), ...,  connect(1,N)
```

Tallest: $N - 1$. See previous question.

(c) Give the best and worst case runtimes for `isConnected` and `connect`.

`isConnected`: Worst: $\Theta(N)$, Best: $\Theta(1)$.

`connect`: Worst: $\Theta(N)$, Best: $\Theta(1)$.

Worst case and best case in this case reflect the worst case and best case height of a quick union tree.

(d) What is the shortest and tallest height possible for a weighted quick union with $N$ elements? What does this mean for the best and worst-case runtimes for `isConnected` and `connect`?

Shortest: 1, Tallest: $\lfloor \log_2 N \rfloor$

`isConnected`: Worst: $\Theta(\log N)$, Best: $\Theta(1)$

`connect` Worst: $\Theta(\log N)$, Best: $\Theta(1)$

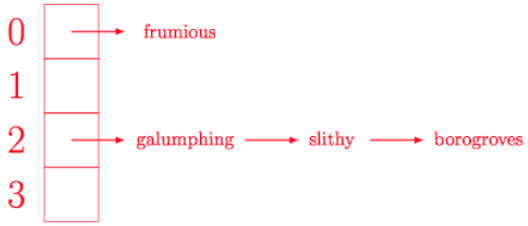
Worst case and best case of the functions reflect the worst case and best case height the tree.

# 6 Hashing

6.1 Illustrate the box-and-pointer diagram that results from inserting **galumphing, frumious, slithy, borogroves, mome, bandersnatch** into a new hash table, if `String::hashCode` just returns the length (this is *not* how strings actually work).

Use external chaining to resolve collisions. Assume 4 is the initial size of the hash table's internal array, and double the array's size when the load factor equals 1.

Below is the hash table after the first 4 insertions.




Before the next insertion, the array is resized.


(Note that the order of the terms in the external chain is not important and depends on the implementation.)

6.2 Describe a potential problem with each of the following `hashCodes`.

(a) `String::hashCode` simply returns the length of the string.

This will likely cause many collisions since many distinct strings have the same length. Since the efficiency of hash tables depends on a low number of collisions, this would result in slow insertion, removal, and search.

(b) `String::hashCode` simply returns a random number each time.

Hash functions must be deterministic, so this `hashCode` is not even valid. If random numbers are used, then `hashCode` may return different values for the same object when called repeatedly, and two `equals` objects may not necessar-

ily return the same `hashCode`.

(c) Overriding the `equals` method without overriding the `hashCode` method.

If `equals` is overridden and `hashCode` is not, then it is possible that two objects will be 'equal' but have different hash values. The `Object` class's `hashCode` returns different values for all objects, regardless of whether or not they're equal. This may cause a `HashSet` to report that an object is not present even if some equal object is in the `HashSet`.

(d) Overriding the `hashCode` method of a class without overriding the `equals` method.

This causes the same problem as above.

(e) Mutating an object after inserting it into a HashSet.

This may prevent us from being able to find the object again since the `HashSet` uses the object's `hashCode` to check if the object is present (and if the object is modified, its `hashCode` may also change).

6.3 (a) Give the worst-case runtime bound for inserting a single entry into a `HashSet` containing $N$ elements. Assume that hashing a key takes constant time. Then, describe a situation in which we would achieve the above runtime.

$\Theta(N)$

An example of a situation that would achieve this runtime is if all elements hashed into the same bucket, so that we essentially just have a linked list inside one bucket. This can occur when our hashing function is poorly designed.

(b) After finishing the hashing lab, Janice decides to create her own hash table implementation, `SmallMap`. In order to avoid costly resizing operations, `SmallMap` 's internal array does not resize: it has a fixed length of 1,024. What is the best-case and worst-case runtime for insertion into a `SmallMap` containing $N$ elements? Assume that the $N$ elements are distributed uniformly.

Best case: $\Theta(N)$. All elements are uniformly distributed, meaning that each bucket will have a linked list of approximately length $\frac{1}{1024}N$. Insertion into a linked list of length $\frac{1}{1024}N$ is $\Theta(N)$. (Note, this is a consequence of not resizing the underlying array).

Worst case: $\Theta(N)$. Insertion into a linked list of size $N$ is $\Theta(N)$.

# 7 Heaps of Fun

7.1 Describe a way to modify the usual max heap implementation so that finding the minimum element takes constant time without incurring more than a constant amount of additional time and space for the other operations.

Simply add a variable that keeps track of the minimum value in the heap. When inserting a new value, simply update this variable if the new value is smaller than it. Since the max heap only supports removing the largest element, rather than arbitrary elements, the minimum element will only be removed when the heap becomes empty, at which point we will need to reset the variable keeping track of the minimum value.

7.2 In class, we looked at one way of implementing a priority queue: the binary heap. Recall that a binary heap is a nearly complete binary tree such that any node is smaller than all of its children. There is a natural generalization of this idea called a $d$-ary heap. This is also a nearly complete tree where every node is smaller than all of its children. But instead of every node having two children, every node has $d$ children for some fixed constant $d$.

(a) Describe how to insert a new element into a $d$-ary heap. (This should be very similar to the binary heap.) What is the running time in terms of $d$ and $N$, the number of elements?

To insert, we add the new element on the last level of the tree and then bubble it up, much as in a binary heap. When bubbling up, we only need to compare the node to its parent. Since the tree has $\Theta(\log_d(n))$ levels and we have to do at most one comparison (compare the node to its parent) and one swap at each level, insertion takes $\Theta(\log_d(n))$.

(b) What is the running time of finding the minimum element in a min-$d$-ary heap with $N$ nodes in terms of $d$ and $N$?

The minimum element is simply the root, just as with a binary min-heap. So finding it takes $\Theta(1)$ time.

(c) Describe how to remove the minimum element from a min-$d$-ary heap. What is the running time in terms of $d$ and $N$?

To remove the minimum element, we first replace it with the last element on the last level of the heap and then bubble this element down, just as in a binary heap. When bubbling a node down, we have to compare the node to all of its children to determine if it is larger than any of them and to find the smallest one (since we must swap with the smallest child to preserve the heap property). So at each level we have to do at most $\Theta(d)$ comparisons and 1 swap. Since there are $\Theta(\log_d N)$ levels, removing the minimum takes $\Theta(d \log_d N)$ time.

7.3 Suppose a value in a max heap changed. To maintain heap invariants, would you bubble the value up, bubble it down, both, or neither? Explain.

If the value is now greater than its parent, bubble it up. If the value is now less than one of its children, bubble it down.

7.4 Suppose you are given an array `A` with $N$ elements such that no element is more than `k` slots away from its position in the sorted array. Assume that $k > 0$ and that $k$, while not constant, is much less than $N$ ($k \ll N$).

(a) Implement `zorkSort` such that the array `A` is sorted after execution. The important operations on a `PriorityQueue` are `add(x)`, `remove()` (remove smallest), and `isEmpty()`.

```
1  public static void zorkSort(int[] A, int k) {
2      int N = A.length, i = 0;
3      PriorityQueue<Integer> pq = new PriorityQueue<>();
4      while (i < k) {
5          pq.add(A[i]);
6          i += 1;
7      }
8      while (i < N) {
9          A[i - k] = pq.remove();
10         pq.add(A[i]);
11         i += 1;
12     }
13     while (!pq.isEmpty()) {
14         A[i - k] = pq.remove();
15         i += 1;
16     }
17 }
```

First we insert $k$ items into the priority queue. We iterate through the array starting at index $i = k$ and maintain a priority queue of the last $k$ elements we have seen. At each iteration, we remove the minimum item in the priority queue and place it in the array at index $i - k$, and add the item at index $i$ into the priority queue.

(b) What is the running time of this algorithm, as a function of $N$ and $k$?

$\Theta(n \log(k))$. The priority queue will have $k$ elements in it, so adding and removing from it will take $\Theta(\log k)$ time. For each of the $n$ elements in the array, we insert into the priority queue and remove from the priority queue once.

7.5 President Trump's speaking style has attracted a great deal of attention in recent years. We'd like to determine the top $k$ most common words found in the list of all $N$ words that the president has ever spoken. Describe how you would solve this problem efficiently and which data structure(s) you would use. The algorithm should be able to return the president's $k$ most frequently-used words in $O(N \log k)$. Assume $k \ll N$.

We'll find Trump's favorite words using a min-heap of $k$ words, along with a hash table of words to frequencies.

First, we will need to iterate over the president's words and build our hash table. When a new word is encountered, it is added to the hash table with an initial frequency of 1. When an existing word is encountered, increment its frequency.

From here, we can retrieve the top $k$ trending words using a min-heap. Iterate over the words in the hash table:

1. If the min-heap still has fewer than $k$ elements, add the word to the heap.
2. Else, peek at the min-heap. If the least frequent word in the heap is less frequently used than the current word, remove from the heap and add the current word.

Building the hash table takes $\Theta(N)$ time and retrieving the $k$ most frequent words takes $O(N \log k)$. Therefore, our total runtime is $O(N \log k)$.