CS 61B Spring 2021

ADTs and Asymptotic Analysis

Discussion 7: March 1, 2021

1 ADT Matchmaking

Match each task to the correct Abstract Data Type for the job by drawing a line connecting matching pairs.

1. You want to keep track of all the unique users who have logged on to your system.
2. You are creating a version control system and want to associate each file name with a Blob.
3. We are running a server and want to service clients in the order they arrive.
4. We have a lot of books at our library and we want our website to display them in some sorted order. We have multiple copies of some books and we want each listing to be separate.

a) List

b) Map

c) Set

d) Queue

1) c) You should use a set because we only want to keep track of unique users (i.e. if a user logs on twice, they shouldn't show up in our data structure twice). Additionally, our task doesn't seem to require that the structure is ordered.
2) b) You should use a map. Maps naturally let you pair a key and value, and here we could have the file name be the key, and the blob be the value.
3) d) We should use a queue. We can push clients to the front of the queue as they arrive, and pop them off the queue as we service them.
4) a) We should use a list because a list is an ordered collection of items. Additionally, we need to allow for duplicate items because we have multiple copies of some books.

2 I Am Speed

Give the worst case and best case running time in $\Theta(\cdot)$ notation in terms of M and N. Assume that `comeOn()` is in $\Theta(1)$ and returns a boolean.

```
1 for (int i = 0; i < N; i += 1) {
2     for (int j = 1; j <= M; ) {
3         if (comeOn()) {
4             j += 1;
5         } else {
6             j *= 2;
7         }
8     }
9 }
```

For `comeOn()` the worst case is $\Theta(NM)$ and the best case is $\Theta(N \log M)$. To see this, note that in the best case `comeOn()` always returns false. Hence `j` multiplies by 2 each iteration, which means the inner loop would take $\log M$ time each time. In the worst case, `comeOn()` always returns false, thus the inner

loop iterates `M` times. Since the outer loop always iterates N times, we get the worst and best case by multiplying the time the inner loop takes by N.

3 Re-cursed with Asymptotics!

(a) What is the runtime of the code below in terms of n?

```
1  public static int[] curse(int n) {
2      if (n <= 0) {
3          return 0;
4      } else {
5          return n + curse(n - 1);
6      }
7  }
```

This takes $\Theta(n)$ time. On each recursive call, we do a constant amount of work. We make n recursive calls, because we go from n to 1. Then n recursive layers with 1 work at each layer is overall $\Theta(n)$ much work.

(b) Assume our BST (Binary Search Tree) below is perfectly bushy. What is the runtime of a single `find` operation in terms of N, the number of nodes in the tree? In this setup, assume a `Tree` has a `Key` (the value of the tree) and then pointers to two other trees, `Left` and `Right`.

```
1  public static BST find(BST T, Key sk) {
2     if (T == null)
3        return null;
4     if (sk.compareTo(T.key) == 0))
5        return T;
6     else if (sk.compareTo(T.key) < 0)
7        return find(T.left, sk);
8     else
9        return find(T.right, sk);
10 }
```

Find operations on a BST take $O(\log(N))$ time, as the height of a perfectly bushy BST is log(N). In the worst case scenario, the key we're looking at is all the way at a leaf, so we have to traverse a path from root to leaf of length log(N).

(c) Can you find a runtime bound for the code below? We can assume the System.arraycopy method takes $\Theta(N)$ time, where N is the number of elements copied. The official signature is `System.arrayCopy(Object sourceArr, int srcPos, Object dest, int destPos, int length)`. Here, `srcPos` and `destPos` are the starting points in the source and destination arrays to start copying and pasting in, respectively, and `length` is the number of elements copied.

```
1  public static void silly(int[] arr) {
2      if (arr.length <= 1) {
3          System.out.println("You won!");
4          return;
5      }
6      int newLen = arr.length / 2
7      int[] firstHalf = new int[newLen];
8      int[] secondHalf = new int[newLen];
9      System.arraycopy(arr, 0, firstHalf, 0, newLen);
10     System.arraycopy(arr, newLen, secondHalf, 0, newLen);
```

```
11      silly(firstHalf);
12      silly(secondHalf);
13  }
```

At each level, we do N work, because the call to System.arraycopy. You can see that at the top level, this is N work. At the next level, we make two calls that each operate on arrays of length N/2, but that total work sums up to N. On the level after that, in four separate recursive function frames we'll call System.arraycopy on arrays of length N/4, which again sums up to N for that whole layer of recursive calls.

Now we look for the height of our recursive tree. Each time, we halve the length of N, which means that the length of the array N on recursive level k is roughly $N * \frac{1}{2}^k$. Then we will finally reach our base case $N <= 1$ when we have $N * \frac{1}{2}^k = 1$. Doing some math, we see this can be transformed into $N = 2^k$, which means $k = log_2(N)$. In other words, the number of layers in our recursive tree is $log_2(N)$. If we have $log_2(N)$ layers with $\Theta(N)$ work on each layer, we must have $\Theta(Nlog(N))$ runtime.

4 Have You Ever Went Fast? *Extra*

Given an `int x` and a *sorted* array `A` of N distinct integers, design an algorithm to find if there exists indices `i` and `j` such that `A[i] + A[j] == x`.

Let's start with the naive solution.

```
public static boolean findSum(int[] A, int x) {
    for (int i = 0; i < A.length; i++){
        for (int j = 0; j < A.length; j++) {
            if (A[i] + A[j] == x) return true;
        }
    }
    return false;
}
```

(a) How can we improve this solution? *Hint*: Does order matter here?

```
public static boolean findSumFaster(int[] A, int x){
    int left = 0;
    int right = A.length - 1;
    while (left <= right) {
        if (A[left] + A[right] == x) {
            return true;
        } else if (A[left] + A[right] < x) {
            left++;
        } else {
            right--;
        }
    }
    return false;
}
```

(b) What is the runtime of both the original and improved algorithm?

Naive: Worst = $\Theta(N^2)$, Best = $\Theta(1)$. Optimized: Worst = $\Theta(N)$, Best = $\Theta(1)$