CS 61B Spring 2024 — ADTs, Asymptotics II, BSTs

Discussion 06: February 26, 2024

1 ADT Matchmaking

Match each task to the best Abstract Data Type for the job and justify your answer (ie. explain why other options would be less ideal). The options are `List, Map, Queue, Set, and Stack`. Each ADT will be used once.

1. You want to keep track of all the unique users who have logged on to your system.

   Solution: 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. You are creating a version control system and want to associate each file name with a Blob.

   Solution: 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. We are grading a pile of exams and want to grade starting from the top of the pile (*Hint: what order do we pile papers in?*).

   Solution: We should use a Stack. When papers are added to a pile, the top of the pile is the last paper added. Since we want to grade the top of the pile first, it makes sense for us to use a last-in-first-out (LIFO) approach in which we continually pop papers off the top of our Stack as we grade them.

4. We are running a server and want to service clients in the order they arrive.

   Solution: 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.

5. 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.

   Solution: 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.

Some geometric sums you may find helpful in the rest of the worksheet:

$$1 + 2 + 3 + 4 + 5 + \cdots + N \in \Theta(N^2)$$
$$1 + 2 + 4 + 8 + 16 + \cdots + N \in \Theta(N)$$

General case:

$$1 + 2 + 3 + 4 + 5 + \cdots + f(N) \in \Theta(f(N)^2)$$
$$1 + 2 + 4 + 8 + 16 + \cdots + f(N) \in \Theta(f(N))$$

2 I Am Speed

(a) For each code block below, fill in the blank(s) so that the function has the desired runtime. Do not use any commas. If the answer is impossible, just write "impossible" in the blank. Assume that `System.out.println` runs in constant time. You may use Java's `Math.pow(x, y)` to raise `x` to the power of `y`.

```
// Desired Runtime: Θ(N)
public static void f1(int N) {
    for (int i = 1; i < N; i += 1){
        System.out.println("hi Dom");
    }
}
```

Note the solution could be `i += C`, where `C` is some constant independent of `N`. This is because even if we did for example, `i += 10`, we would do $N/10$ work in total, which is still $\Theta(N)$.

```
// Desired Runtime: Θ(log N)
public static void f2(int N) {
    for (int i = 1; i < N; i *= 2) {
        System.out.println("howdy Ergun");
    }
}
```

Here, the solution could be `i *= C`, where `C` is some constant independent of `N`. This is because even if we did for example, `i *= 5`, we would do $\log_5(N)$ work in total, and in general $\log_i(N)$ work, which is still $O(\log n)$.

```
// Desired Runtime: Θ(1)
public static void f3(int N) {
    for (int i = 1; i < 1000; i += 1) {
        System.out.println("hello Anniyat");
    }
}
```

Again, the solution is actually just `i < C`, where `C` is some constant independent of the input `N`.

```
// Desired Runtime: Θ(2^N)
// This one is tricky! Hint: think about the dominating term in 1 + 2 + 4 + 8 + ... + f(N)
public static void f4(int N) {
    for (int i = 1; i < Math.pow(2, N); i *= 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("what's up Alyssa");
        }
    }
```

```
}
```

(b) *Extra* Give the worst case and best case running time in $\Theta(\cdot)$ notation in terms of M and N. Assume that `kachow()` runs in $\Theta(N^2)$ time and returns a `boolean`.

```
for (int i = 0; i < N; i += 1) {
    for (int j = 1; j <= M; ) {
        if (kachow()) {
            j += 1;
        } else {
            j *= 2;
        }
    }
}
```

Solution:

Worst case: $\Theta(N^3 M)$, Best case: $\Theta(N^3 \log M)$

To see this, note that in the best case `kachow()` always returns false. Regardless of best/worst case, `kachow()` runs with each iteration of the inner loop. Hence `j` multiplies by 2 each inner loop iteration, which means the inner loop would take $N^2 \log M$ time for each iteration of the outer loop. In the worst case, `kachow()` always returns true, thus the inner loop iterates `M` times, each time calling `kachow()`, so the inner loop would take $N^2 M$. 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!

To help visualize the solutions better, a video walkthrough of this problem is linked here!

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

```
public static int curse(int n) {
    if (n <= 0) {
        return 0;
    } else {
        return n + curse(n - 1);
    }
}

```

Solution: $\Theta(n)$. 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 $\Theta(n)$ work.

(b) 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.

```
public static void silly(int[] arr) {
    if (arr.length <= 1) {
        System.out.println("You won!");
        return;
    }

    int newLen = arr.length / 2;
    int[] firstHalf = new int[newLen];
    int[] secondHalf = new int[newLen];

    System.arraycopy(arr, 0, firstHalf, 0, newLen);
    System.arraycopy(arr, newLen, secondHalf, 0, newLen);

    silly(firstHalf);
    silly(secondHalf);
}


```

Solution:

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.

(c) Given that `exponentialWork` runs in $\Theta(3^N)$ time with respect to input N, what is the runtime of `ronnie`?

```
public void ronnie(int N) {
    if (N <= 1) {
        return;
    }
    ronnie(N - 2);
    ronnie(N - 2);
    ronnie(N - 2);
    exponentialWork(N); // Runs in Θ(3^N) time
}
```

Solution: $\Theta(3^N)$. Drawing out the recursive tree, the first level has $\Theta(3^N)$ work, the next level has $\Theta(3^{N-1})$ work, and so on until the last level which has approximately $\Theta(3^{N/2})$ work. This gives the sum $\Theta(3^N) + \Theta(3^{N-1}) + ... + \Theta(3^{N/2}) = \Theta(3^N)$.

4 BST Asymptotics

Below we define the `find` method of a BST (Binary Search Tree) as in lecture, which returns the BST rooted at the node with key `sk` in our overall BST. In this setup, assume a `BST` has a `key` (the value of the tree root) and then pointers to two other child BSTs, `left` and `right`.

```java
public static BST find(BST tree, Key sk) {
    if (tree == null) {
        return null;
    }
    if (sk.compareTo(tree.key) == 0) {
        return tree;
    } else if (sk.compareTo(tree.key) < 0) {
        return find(tree.left, sk);
    } else {
        return find(tree.right, sk);
    }
}
```

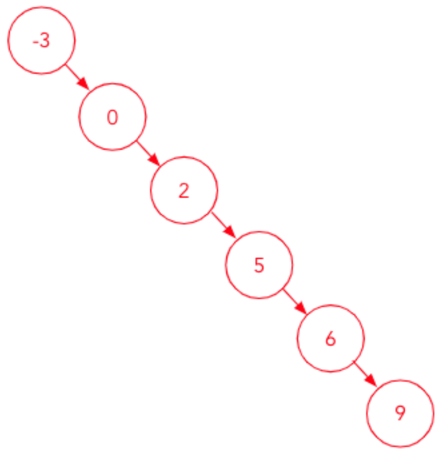
(a) Assume our BST is perfectly bushy. What is the runtime of a single `find` operation in terms of N, the number of nodes in the tree? Can we generalize the runtime of `find` to a tight bound?

Solution: Find operations on a perfectly bushy 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).

We cannot generalize the runtime of `find` to a tight bound because the lower and upper bounds are different. It is lower-bounded by $\Omega(1)$ (the case where the key we are looking for is at the root of the BST provided) and upper-bounded by $O(\log(N))$ (mentioned above as the path from root to leaf). Therefore, there is no tight bound for `find`.

(b) Say we have an empty BST and want to insert the keys `[6, 2, 5, 9, 0, -3]` (in some order). In what order should we insert the keys into the BST such that the runtime of a single `find` operation after all keys are inserted is $O(N)$? Draw out the resulting BST.

Solution: We should insert the keys in ascending sorted order: [-3, 0, 2, 5, 6, 9]. This results in a perfectly linear BST, which means that the longest path for `find` (from root to leaf) traverses every single node in the BST. The resulting BST looks like a direct chain of nodes:



Alternatively, we could insert the keys in descending sorted order, which also results in a perfectly linear BST (but the keys would chain left from $9 \rightarrow 6 \rightarrow 5 \rightarrow 2 \rightarrow 0 \rightarrow$ -3).