CS 61B Trees Fall 2020

More Asymptotics, Search

Discussion 7: October 5, 2020

1 Binary Search Trees

(a) For a integer BST that contains N items, what is the asymptotic worst case of each of these functions?

i. `add(int x)` $\Theta(N)$. In the worst case, we add items in increasing or decreasing order, resulting in what is essentially a linked list which requires us to traverse the whole thing before adding x.

ii. `getMin()` $\Theta(N)$. Just like in part (i), if we add items in decreasing order we will have to traverse the whole list before we find our smallest item.

(b) Now assume that the BST from part (a) is certain to be bushy. What is our new asymptotic worst case?

i. `add(int x)` $\Theta(\log N)$. Since this tree is guaranteed to be busy, we know that the max height of the tree will be $\log N$ so we only need to visit $\log N$ items before we can add.

ii. `getMin()` $\Theta(\log N)$. Just like in part (i), since the tree is guaranteed to be bushy, the height is $\log N$ so the furthest away the minimum value can be is $\log N$.

(c) Let's implement the `find` function in our BSTMap. It should take in an integer and return the value associated with that key or null if the key is not in our BSTMap.

```
1  public class BSTMap {
2      private class Node {
3          int key;
4          int value;
5          Node left;
6          Node right;
7          Node (int key, int value) { ... }
8      }
9
10     Node head; // Contains the node at the head of the tree
11     ...
12     public Integer find(int key) {
13         return findHelper(key, head);
14     }
15
16     private Integer findHelper(int key, Node n) {
17         if (n == null) {
```

```
            return null;
        }
        if (n.key == key) {
            return n.value;
        } else if (n.key > key) {
            return findHelper(key, n.left);
        } else {
            return findHelper(key, n.right);
        }
    }
}
```

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.

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

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 Have You Ever Went Fast?

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 = $\Theta(1)$

4 CTCI *Extra*

(a) **Union** Write the code that returns an array that is the union between two given arrays. The union of two arrays is a list that includes everything that is in both arrays, with no duplicates. Assume the given arrays do not contain duplicates. For example, the union of `{2, 1, 3, 4}` and `{3, 4, 6, 5}` is `{1, 2, 3, 4, 5, 6}`. The method should run in $O(M + N)$ time where M and N is the size of each array. The returned list does not need to remain in the same order as the elements of the input lists.

Hint: Think about using ADTs other than arrays to make the code more efficient.

```
public static int[] union(int[] A, int[] B) {
    HashSet<Integer> set = new HashSet<Integer>();
    for (int num : A) {
        set.add(num);
    }
    for (int num : B) {
        set.add(num);
    }
    int[] unionArray = new int[set.size()];
    int index = 0;
    for (int num : set) {
        unionArray[index] = num;
        index += 1;
    }
    return unionArray;
```

```
}
```

(b) **Intersect** Now do the same as above, but find the intersection between both arrays. The intersection of two arrays is the list of all elements that are in both arrays. Again assume that neither array has duplicates. For example, the intersection of `{1, 2, 3, 4}` and `{3, 4, 5, 6}` is `{3, 4}`. The returned list does not need to remain in the same order as the elements of the input lists.

Hint: Like in part (a), think about using ADTs other than arrays to make the code more efficient.

```
public static int[] intersection(int[] A, int[] B) {
    HashSet<Integer> setOfA = new HashSet<Integer>();
    HashSet<Integer> intersectionSet = new HashSet<Integer>();
    for (int num : A) {
        setOfA.add(num);
    }
    for (int num : B) {
        if (setOfA.contains(num)) {
            intersectionSet.add(num);
        }
    }
    int[] intersectionArray = new int[intersectionSet.size()];
    int index = 0;
    for (int num : intersectionSet) {
        intersectionArray[index] = num;
        index += 1;
    }
    return intersectionArray;
}
```