

1 Disjoint Sets

For each of the arrays below, write whether this could be the array representation of a weighted quick union with path compression and explain your reasoning. **Break ties by choosing the smaller integer to be the root.**

i:	0	1	2	3	4	5	6	7	8	9
A. a[i]:	1	2	3	0	1	1	1	4	4	5
B. a[i]:	9	0	0	0	0	0	9	9	9	-10
C. a[i]:	1	2	3	4	5	6	7	8	9	-10
D. a[i]:	-10	0	0	0	0	1	1	1	6	2
E. a[i]:	-10	0	0	0	0	1	1	1	6	8
F. a[i]:	-7	0	0	1	1	3	3	-3	7	7

$$\log_2 10 = 3-4$$

Solution:

There are three criteria here that invalidate a representation:

- If there is a cycle in the parent-link.
- For each parent-child link, the tree rooted at the parent is smaller than the tree rooted at the child before the link (you would have merged the other way around).
- The height of the tree is greater than $\log_2 n$, where n is the number of elements.

Therefore, we have the following verdicts.

- A. Impossible: has a cycle 0-1, 1-2, 2-3, and 3-0 in the parent-link representation.
- B. Impossible: the nodes 1, 2, 3, 4, and 5 must link to 0 when 0 is a root; hence, 0 would not link to 9 because 0 is the root of the larger tree.
- C. Impossible: tree rooted at 9 has height $9 > \log_2 10$.
- D. Possible: 8-6, 7-1, 6-1, 5-1, 9-2, 3-0, 4-0, 2-0, 1-0.
- E. Impossible: tree rooted at 0 has height $4 > \log_2 10$.
- F. Impossible: tree rooted at 0 has height $3 > \log_2 7$.

2 Asymptotics of Weighted Quick Unions

Note: for all big Ω and big O bounds, give the *tightest* bound possible.

- (a) Suppose we have a Weighted Quick Union (WQU) without path compression with N elements.

- What is the runtime, in big Ω and big O , of `isConnected`?

$\Omega(\underline{1})$, $O(\underline{\log N})$

- What is the runtime, in big Ω and big O , of `connect`?

$\Omega(\underline{1})$, $O(\underline{\log N})$

- (b) Suppose we add the method `addToWQU` to a WQU without path compression. The method takes in a list of `elements` and `connects` them in a random order, stopping when all elements are connected. Assume that all the `elements` are disconnected before the method call.

```
void addToWQU(int[] elements) {
    int[][] pairs = pairs(elements);
    for (int[] pair: pairs) {
        if (size() == elements.length) {
            return;
        }
        connect(pair[0], pair[1]);
    }
}
```

The `pairs` method takes in a list of `elements` and generates all possible pairs of elements in a random order. For example, `pairs([1, 2, 3])` might return `[[1, 3], [2, 3], [1, 2]]` or `[[1, 2], [1, 3], [2, 3]]`.

The `size` method calculates the size of the largest component in the WQU.

Assume that `pairs` and `size` run in constant time.

What is the runtime of `addToWQU` in big Ω and big O ?

$\Omega(\underline{N})$, $O(\underline{N \log N} \text{ } \underline{N^2 \log N})$

Hint: Consider the number of calls to `connect` in the best case and worst case. Then, consider the best/worst case time complexity for one call to `connect`.

- (c) Let us define a **matching size connection** as `connecting` two components in a WQU of equal size. For instance, suppose we have two trees, one with values 1 and 2, and another with the values 3 and 4. Calling `connect(1, 4)` is a matching size connection since both trees have 2 elements.

What is the **minimum** and **maximum** number of matching size connections that can occur after executing `addToWQU`? Assume N , i.e. `elements.length`, is a power of two. Your answers should be exact.

minimum: 1, maximum: $n-1$

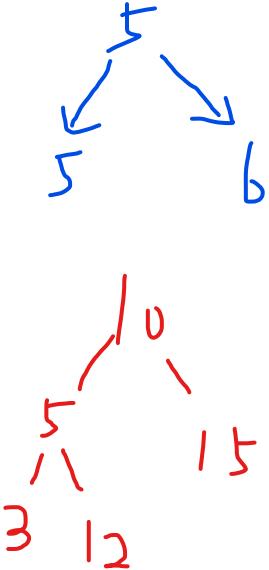
In general, there are $N/2$ connections of size 1, and so on, $N/2+N/4+N/8+\dots+2+1$, which simplifies to $N-1$.

3 Is This a BST?

In this setup, assume a **BST** (Binary Search Tree) has a **key** (the value of the tree root represented as an **int**) and pointers to two other child BSTs, **left** and **right**. Additionally, assume that **key** is between **Integer.MIN_VALUE** and **Integer.MAX_VALUE** non-inclusive.

- (a) The following code should check if a given binary tree is a BST. However, for some trees, it returns the wrong answer. Give an example of a binary tree for which **brokenIsBST** fails.

```
public static boolean brokenIsBST(BST tree) {
    if (tree == null) {
        return true;
    } else if (tree.left != null && tree.left.key >= tree.key) {
        return false;
    } else if (tree.right != null && tree.right.key <= tree.key) {
        return false;
    } else {
        return brokenIsBST(tree.left) && brokenIsBST(tree.right);
    }
}
```



- (b) Now, write **isBST** that fixes the error encountered in part (a).

Hint: You will find **Integer.MIN_VALUE** and **Integer.MAX_VALUE** helpful.

Hint 2: You want to somehow store information about the keys from previous layers, not just the direct parent and children. How do you use the parameters given to do this?

```
public static boolean isBST(BST T) {
    return isBSTHelper(T, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

public static boolean isBSTHelper(BST T, int min, int max) {
    if (T == null) {
        return true;
    } else if (T.key <= min || T.key >= max) {
        return false;
    } else {
        return isBSTHelper(T.left, min, T.key) && isBSTHelper(T.right, T.key, max);
    }
}
```