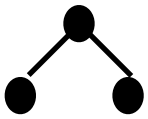
Isaac Delgado
4189182267

1. **A binary tree is a rooted tree in which each node has at most two children. Prove by induction that in any nonempty binary tree the number of nodes with two children is exactly one less than the number of leaves.**

We prove this by Induction.

Base Case:

We have a binary tree of size 3 (1 parent and 2 child nodes) as well as 2 edges connecting each child node to the parent.



# of parent nodes with two children = 1
# of leafs = 2

In this case we have 1 node and 2 children nodes hence 2 leafs. This satisfies the relationship for the number of nodes with two children is exactly one less than the number of leaves.

Inductive Hypothesis:

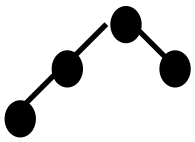
Assume this is true for a binary tree with N nodes and E edges.

Inductive Step:

Prove for a binary tree of size N+1 nodes.

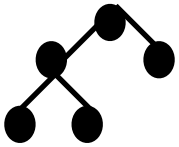
Consider a binary tree with N nodes, when we add an additional node there are two cases.

Case 1: Our added node is the first child to a parent node. In this case the number of leafs are unchanged and the number of parent nodes with two children are unchanged. Consequently are relationship is still satisfied.



# of parent nodes with two children = 1
# of leafs = 2

Case 2: Our added node is the second child to a parent node, thus completing the parent. In this case 1 new leaf is added and 1 new parent node with two children is added to our tree. Our relationship will continue to hold.



# of parent nodes with two children = 2
# of leafs = 3

Isaac Delgado
4189182267

**2. Suppose we perform a sequence of *n* operations on a data structure in which the $i^{th}$ operation costs *i* if *i* is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.**

Example operations

| Operation | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Cost | 1 | 2 | 1 | 4 | 1 | 1 | 1 |

The single cost operations will be linear ➔ O(n)
The operations that are power's of 2 is as as follows

$$S = \sum_{i=0}^{\log(n)} 2^i = 2^0 + 2^1 + 2^2 + 2^3 + \ldots.. + 2^{\log(n)}$$

$2S = 2 + 2^2 + 2^3 + \ldots.. + (2*2^{\log(n)})$
$S-2S = (2 + 2^2 + 2^3 + \ldots.. + (2*2^{\log(n)})) - (2^0 + 2^1 + 2^2 + 2^3 + \ldots.. + 2^{\log(n)})$
$S(1-2) = 1 + (2*2^{\log(n)})$
$-S = 1 + (2*2^{\log(n)})$
$S = (2*2^{\log(n)}) - 1$
**$S = 2n - 1$**

Putting it together we get the following, which we can say is less than a cn where c is some constant ≤ 3.

$n + (2n - 1) < 3n$

$$\frac{n + (2n - 1)}{n} < 3$$

Thus the amortized cost is simply constant. O(1).

Isaac Delgado
4189182267

3. **When we have two sorted lists of numbers in non-descending order, and we need to merge them into one sorted list, we can simply compare the first two elements of the lists, extract the smaller one and attach it to the end of the new list, and repeat until one of the two original lists become empty, then we attach the remaining numbers to the end of the new list and it's done. This takes linear time. Now, try to give an algorithm using O(*n* log *k*) time to merge *k* sorted lists (you can also assume that they contain numbers in non-descending order) into one sorted list, where *n* is the total number of elements in all the input lists. Use a heap for *k*-way merging.**

a.) We can build a heap with the first elements of each list. This take O(k).

b.) We delete_min from our created heap and also insert into our heap from the corresponding list of the element we just deleted. delete_min takes O(log(k)) and the insert will take O(log(k))

c.) Adding a & b we get O(k + 2(log(k))) = O(log(k))

d.) We a & b until there are no more elements available, this will take O(n)

Thus the algorithm takes O(n log(k))

4. **You are given a weighted graph G, two designated vertices *s* and *t*. Your goal is to find a path from *s* to *t* in which the minimum edge weight is maximized i.e. if there are two paths with weights 10→1→5 and 2→7→3 then the second path is considered better since the minimum weight (2) is greater than the minimum weight of the first (1). Describe an efficient algorithm to solve this problem and show its complexity.**

If I know the weights on the graph I can create a range of weights from lowest to highest and do a binary search on this range for my comparison weight, we can call this weight *w*. I run a BFS testing if I can find a path from *s* to *t* with only weight greater than my current weight *w* I got from my binary search. I will continue this until there I have gone through my entire range. At the end, my path should be the one counting the edge weight identical to *w*.

The binary search through my range of weights will be time complexity of O(log(n)) where n is the largest weight in our given graph.
We must travers our graph through BFS which will take linear time of O(V+E)
Thus, the time complexity will be O((V+E) + log(n))