COL106 - Data Structures and Algorithms

COMPUTING THE DEPTH OF A NODE

DEPTH of a node is the number of ancestors until root.

1) If node p is the root of the tree then its depth = 0.

2) Depth of a node is 1 more than the depth of its parent.

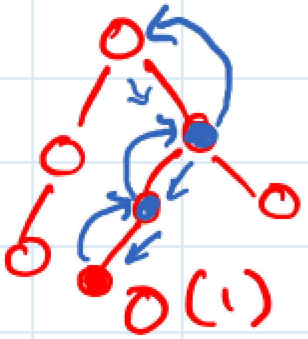
Recursion

```
public int depth (TreeNode<E> p){
    if (isRoot(p) == true)
        return 0;
    return depth(parent(p)) + 1;
}
```

COMPUTING THE DEPTH OF A NODE

DEPTH of a node is the number of ancestors until root.

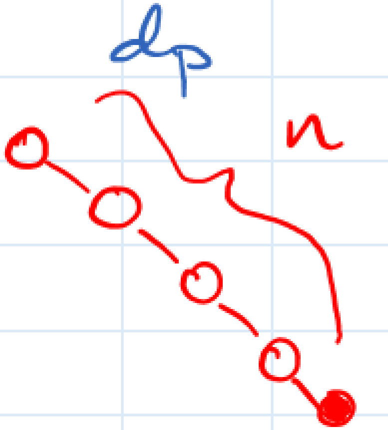
1) If node p is the root of the tree then its depth = 0.
2) Depth of a node is 1 more than the depth of its parent.



$O(d_p)$

```java
public int depth (TreeNode<E> p){
    if (isRoot(p) == true)
        return 0;
    return depth(parent(p)) + 1;
}
```

non recursive



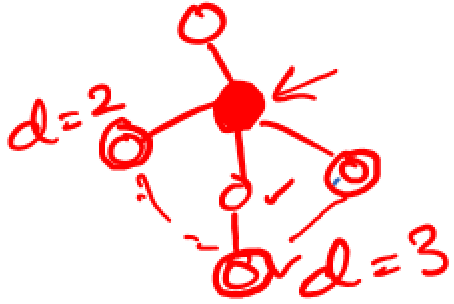
If a node at depth d_p, then above program takes $O(d_p+1)$

What is the worst-case running time? $O(n)$

Height of a Tree

Height = max. of all depths.

```
int ComputeHeight (){
    int height = 0;
    for (TreeNode<E> p : positions()) {
        if (isExternal(p)) { //Consider only leaf nodes
            height = Math.max(height, depth(p));
        }
    }
    return height;
}
```



$O(n)$

$\underline{n * O(d_p + 1)}$

$O(n)$

$O(n^2)$

Height of a Tree

Height = max. of all depths.

```
int ComputeHeight (){  1 usage
    int height = 0;
    for (TreeNode<E> p : positions()) {
        if (isExternal(p)) { //Consider only leaf nodes
            height = Math.max(height, depth(p));
        }
    }
    return height;
}
```

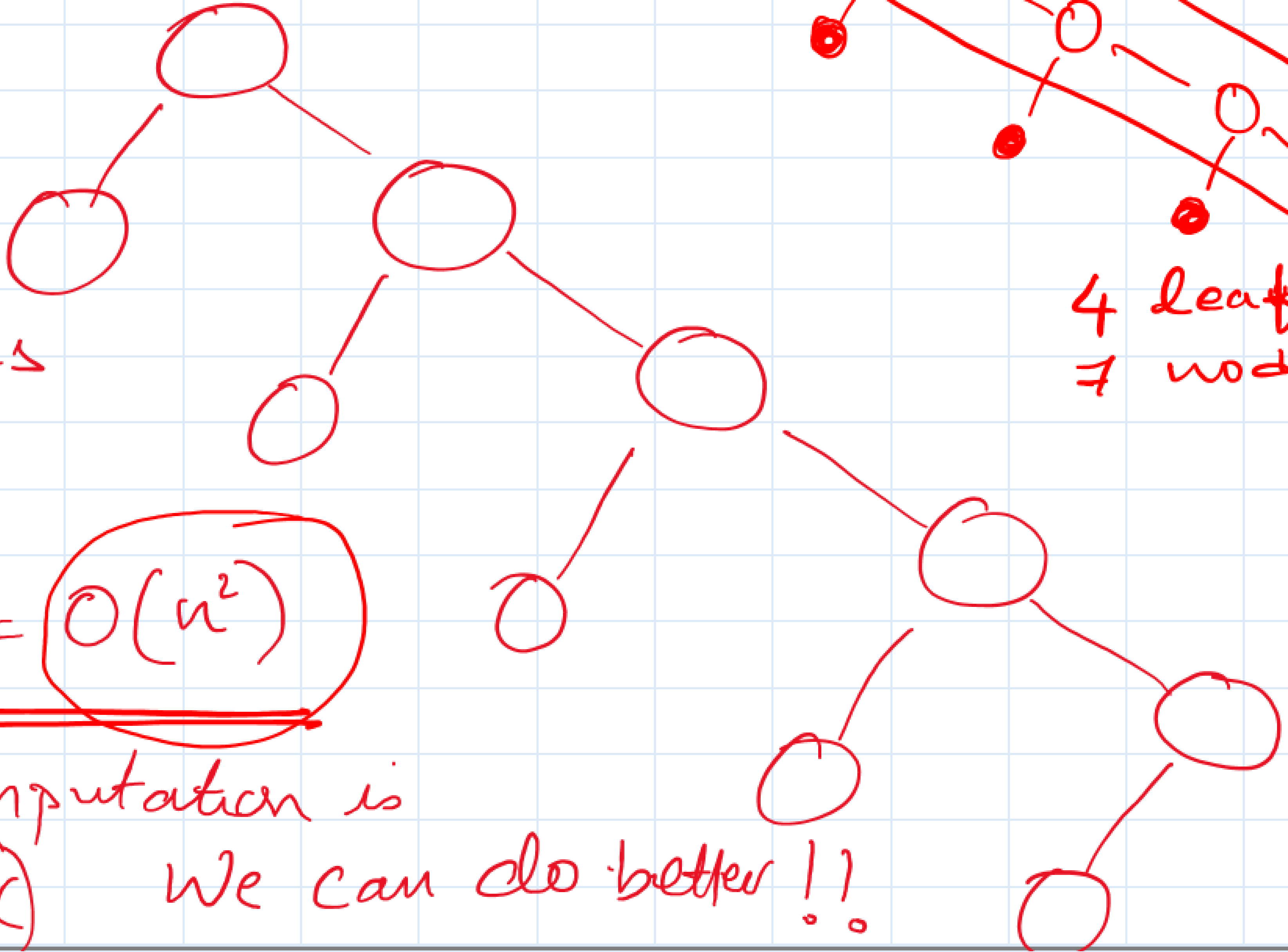
positions () can be run in $O(n)$

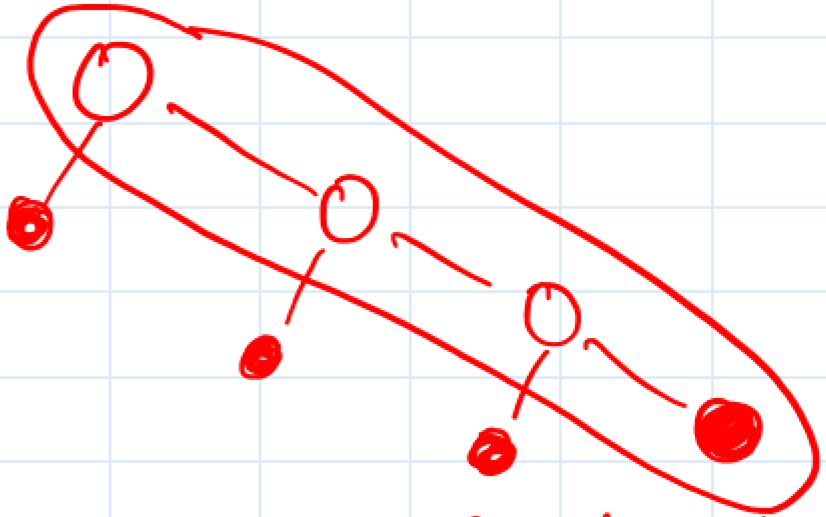
depth() on each leaf.

so, if there are L leaf nodes then $O(n + \sum_{L} (d_p + 1))$

* What is the overall complexity?

Height of a tree





4 leaf nodes
7 nodes

\# leaf nodes

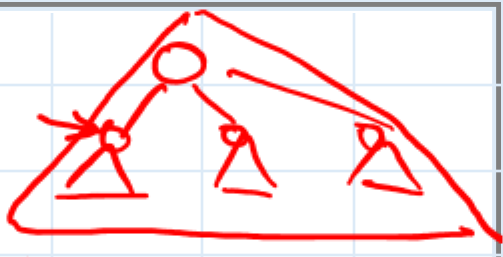
$$= \frac{n}{2} + 1$$

$$\Rightarrow \sum_{L} (d_p + 1) = O(n^2)$$

$\Rightarrow$ Height computation is

$O(n^2)$:-(We can do better !!

Alternative definition of height



- For a leaf node p, its height is 0.
- Height of p is one more than max of heights its children.

```java
int fasterHeight (TreeNode<E> p) {  1 usage
    int h = 0;
    for (TreeNode<E> c : children (p)) {
        h = Math.max(h, 1 + fasterHeight (c));
    }
    return h;
}
```

$O(C_p + 1)$

children(p) can be run in $O(C_p + 1)$

$\Rightarrow$ For each TreeNode, the algorithm does $O(C_p + 1)$ work

```java
int fasterHeight (TreeNode<E> p) {   1 usage
    int h = 0;
    for (TreeNode<E> c : children (p)) {
        h = Math.max(h, 1 + fasterHeight (c));
    }
    return h;
}
```

Height of a Tree

children(p) can be run in $O(c_p + 1)$

$\Rightarrow$ For each Tree Node, the algorithm does $O(c_p + 1)$ work

$\Rightarrow$ Overall running time $O(\Sigma (c_p + 1))$

$= O(n + \Sigma c_p)$

What is Σc_p ?

$n-1$

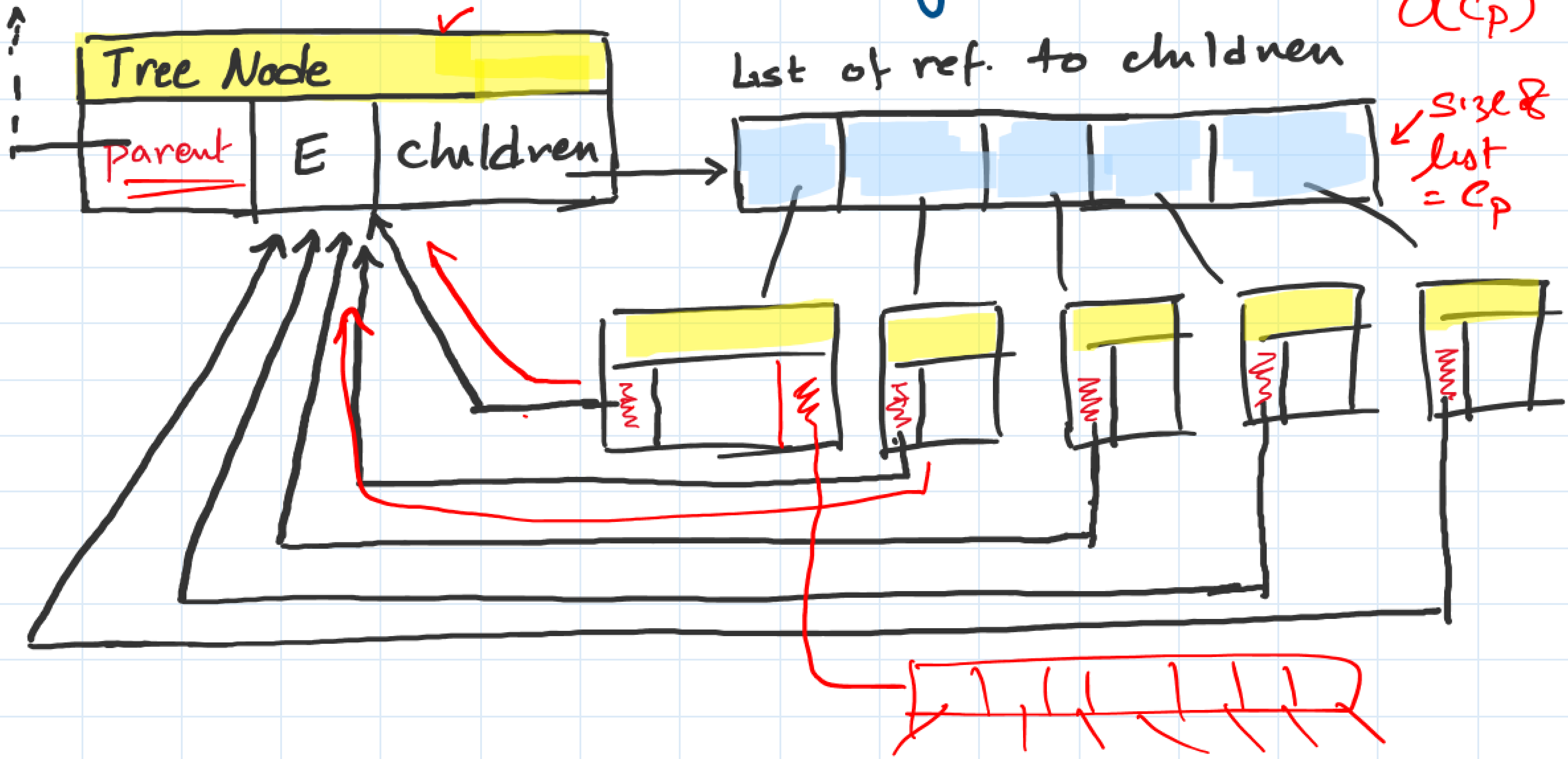
$O(n + n - 1)$

$= O(2n - 1)$

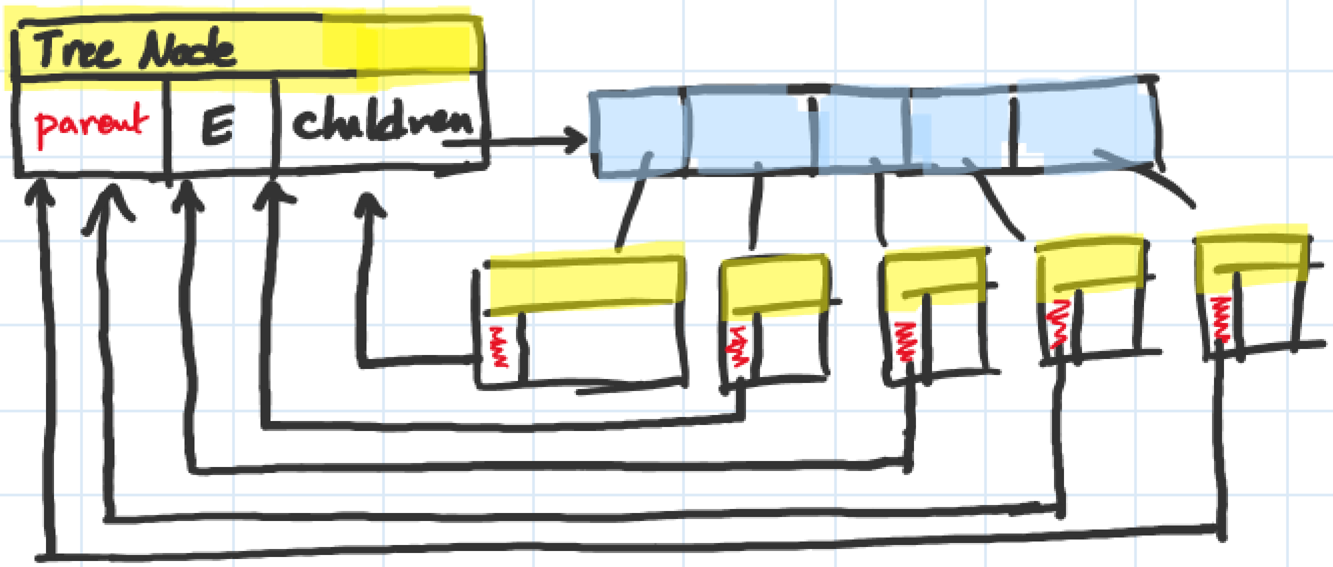
$= O(n)$

Implementing Trees.

An internal node can have many children.



Implementing Trees.




```
class TreeNode<E> {
    E element;
    List<TreeNode<E>>
        children;
    TreeNode<E> parent;
    :
    :
}
```

parent(p) — $O(1)$ ✓

children(p) — $O(c_p + 1)$

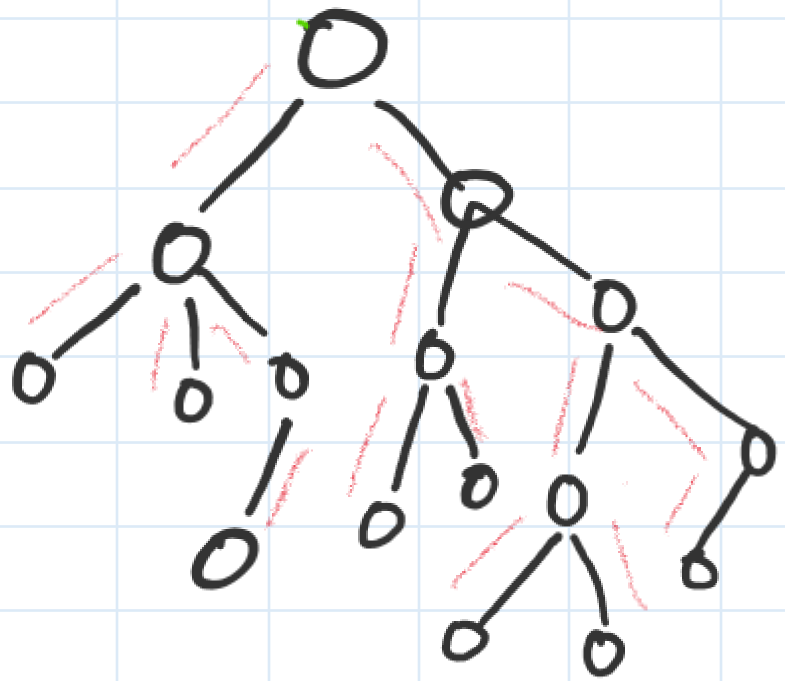
positions() — ?

TREE TRAVERSALS

TRAVERSAL IS A SYSTEMATIC WAY OF "VISITING" ALL NODES

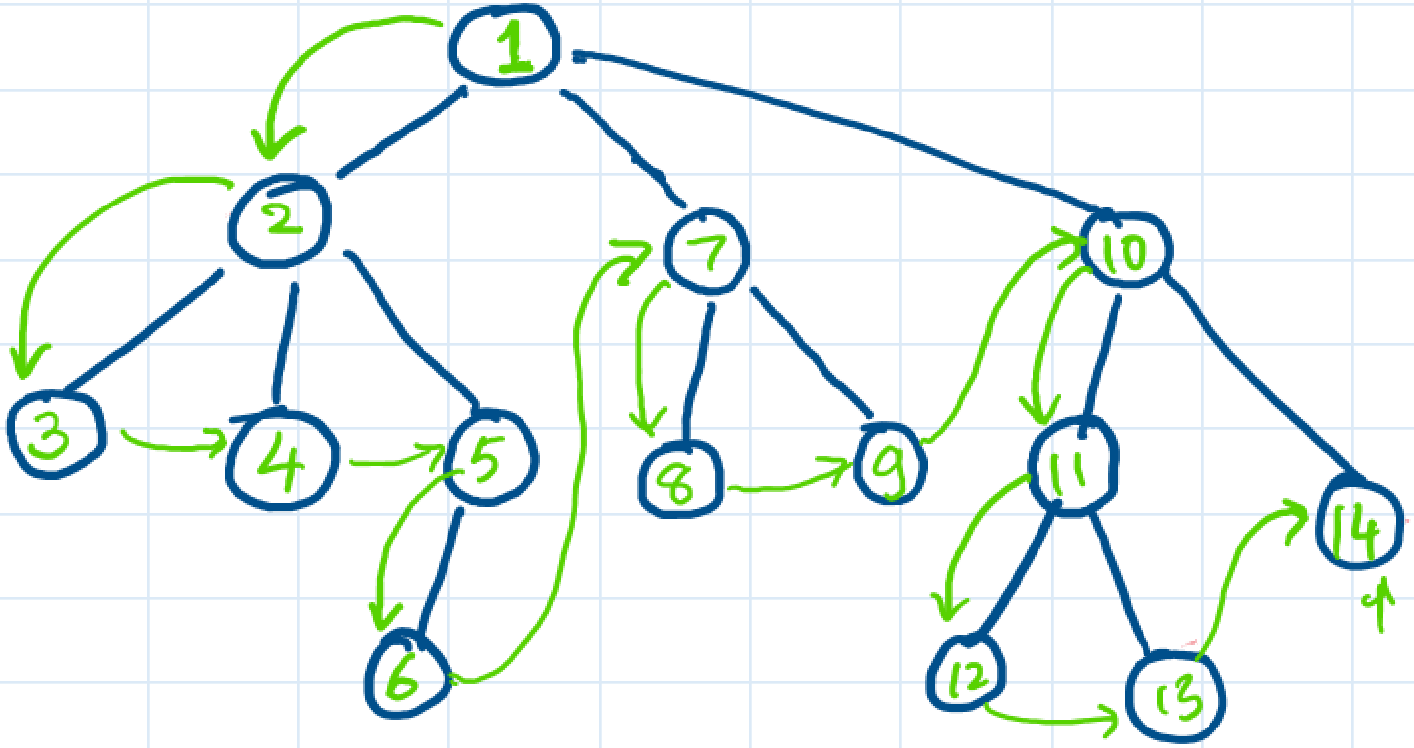
"VISIT" MAY INVOLVE SOME WORK DONE AT A NODE

WE ASSUME THE WORK IS $\boxed{O(1)}$ AT EACH VISIT.



PREORDER TRAVERSAL OF TREE T

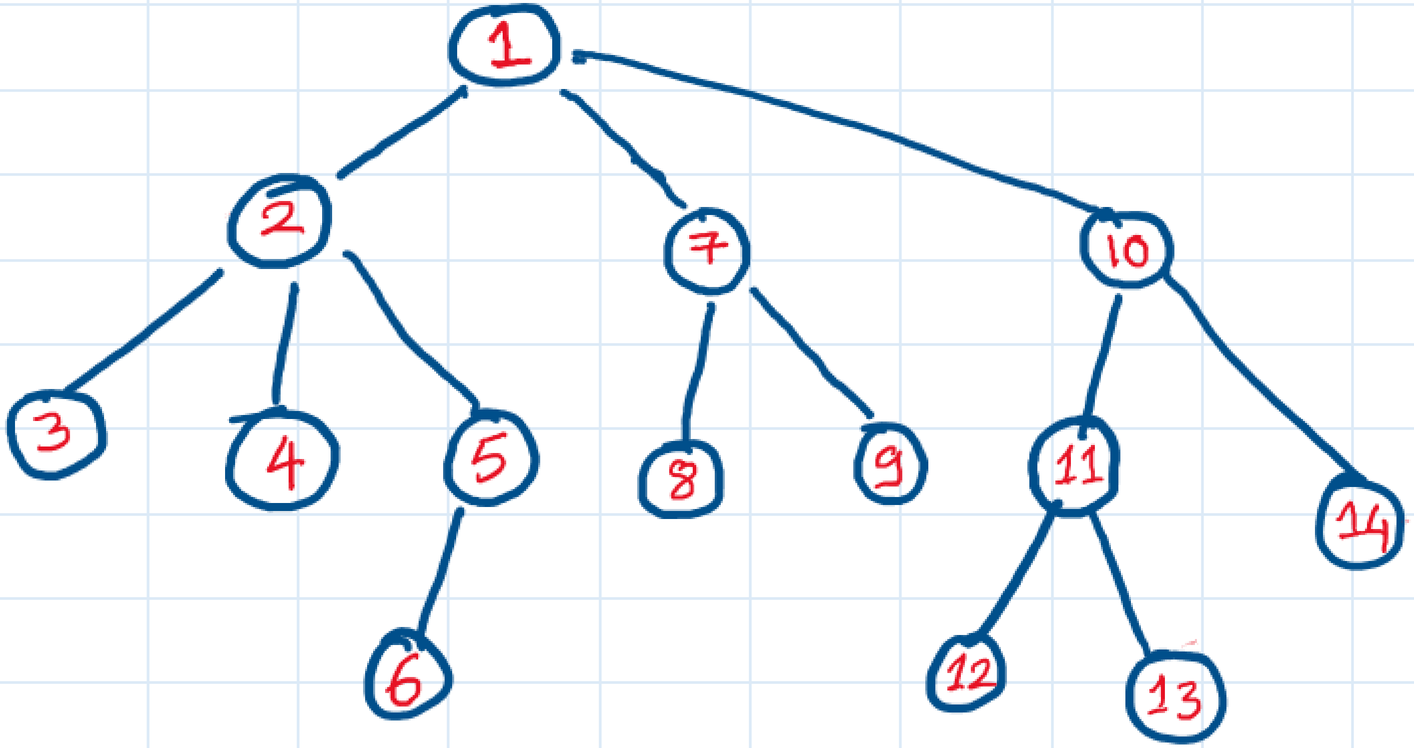
- VISIT ROOT OF T
- TRAVERSE SUBTREES ROOTED AT ITS CHILDREN
  - In ORDERED TREES MAINTAIN ORDER.




```
preorder(p):
  visit(p);
  for each c in children(p)
    preorder(c)
```

PREORDER TRAVERSAL OF TREE T

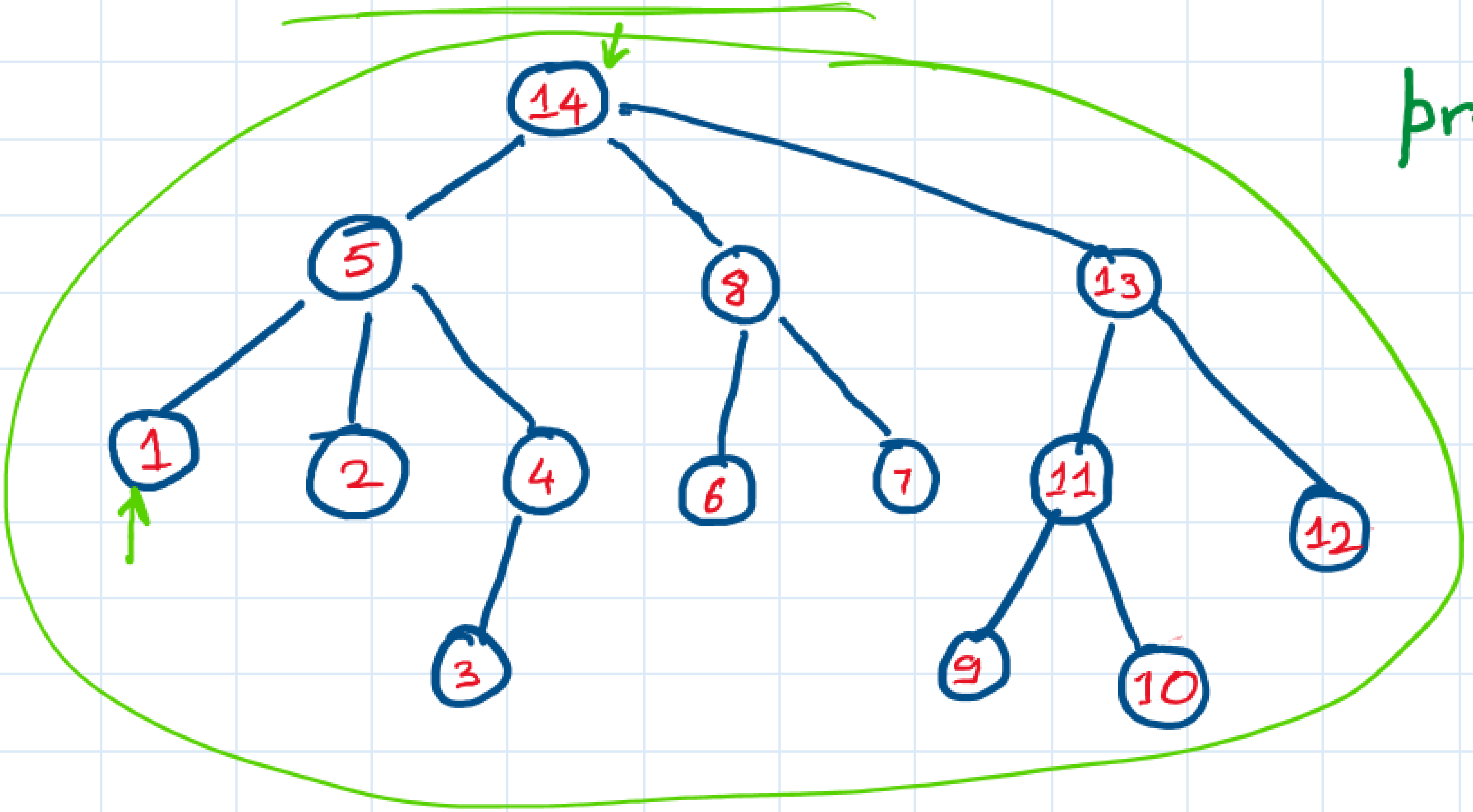
- VISIT ROOT OF T
- TRAVERSE SUBTREES ROOTED AT ITS CHILDREN
    - In ORDERED TREES MAINTAIN ORDER.



preorder(p):
  visit(p);
  for each c in children(p)
    preorder(c)

POST ORDER ~~PREORDER~~ TRAVERSAL OF TREE T

- TRAVERSE SUBTREES ROOTED AT ITS CHILDREN
  - In ORDERED TREES MAINTAIN ORDER.
- VISIT ROOT OF T



preorder(p) :
  visit(p);
  for each c in children(p)
    preorder(c)