What is a tree?

↳ A connected graph with no cycles

What is a binary tree?

↳ A tree of binary nodes

What is a binary node?

↳ A linked node container having a constant number of fields:

- A pointer to an item stored at the node.
- A pointer to a parent node (possibly None)
- A pointer to a left child node (possibly None).
- A pointer to a right child node (possibly None).

Code of Binary Node:

```
class Binary_Node:                          # O(1)
    def __init__(self, x):
        self.item = x
        self.left = x
        self.right = x
        self.parent = x
```

- A binary node can actually be connected to 3 other nodes (parent node, left node, right node)

- We call a node "binary" based on the number of children the node has (max 2).

What is the root of a tree?

↳ The only node in the tree lacking a parent

- All nodes in a tree that are not root nodes can reach the node traversing the parent pointers

- The nodes passed when traversing parent pointers from node <x> back to the root are called ancestors for <x> in the tree

- The depth of a node <x> in the sub-tree rooted at <R> is the length of the path from <x> back to <R> (number of edges).

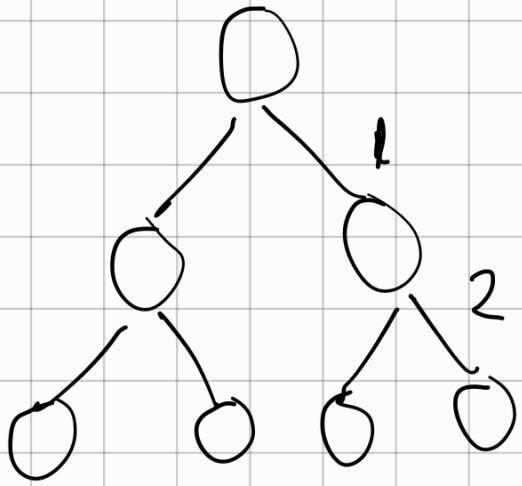
- The height of node <x> is the maximum depth of any node in the subtree rooted at <x>.

- A leaf is a node with no children

Linked Lists V.S. Trees

Why store items in a binary tree?

Many linked list nodes can be $O(n)$ pointers hops away from the head of the list, so it may take $O(n)$ time to reach them.



$\log_2 7 \approx 2.8$

$\log_2 \overset{\text{floor}}{=} 2$

It is possible to construct a binary tree with n nodes such that no node is more than $O(\log n)$ pointer hops away from the root → There exists binary

trees with logarithmic height.

Power → if we can keep the Binary Tree height low $O(\log n)$, and only perform operations on the tree that run in time on the order of the height of the tree, these operations will run $O(h) = O(\log n)$ time

$$O(1) < O(\log n) < O(n)$$

Constant ideal BT Linked List

Power:

- keep the height of the BT low → $O(\log n)$
- Only perform operations that run in time on the order of the height → $O(h) = O(\log n)$

Tree Traversals

Inorder traversal:

- Nodes in a binary tree have a natural order base on the fact that we distinguish one child to be left and one child to be right.

Rules for inorder traversal;

- every node in the left subtree of node <x> comes before <x> in the traversal order
- every node in the right subtree of node <x> comes after <x> in the traversal order

left → root → right

Algorithm:

Given a binary node <A>, we can list the nodes in <A>'s subtree by:

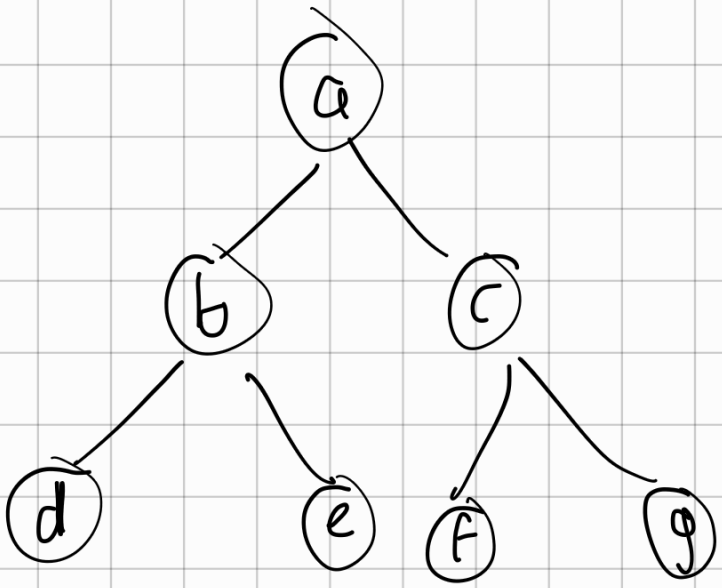
1. Recursively list the nodes in <A>'s left subtree

2. List <A> itself

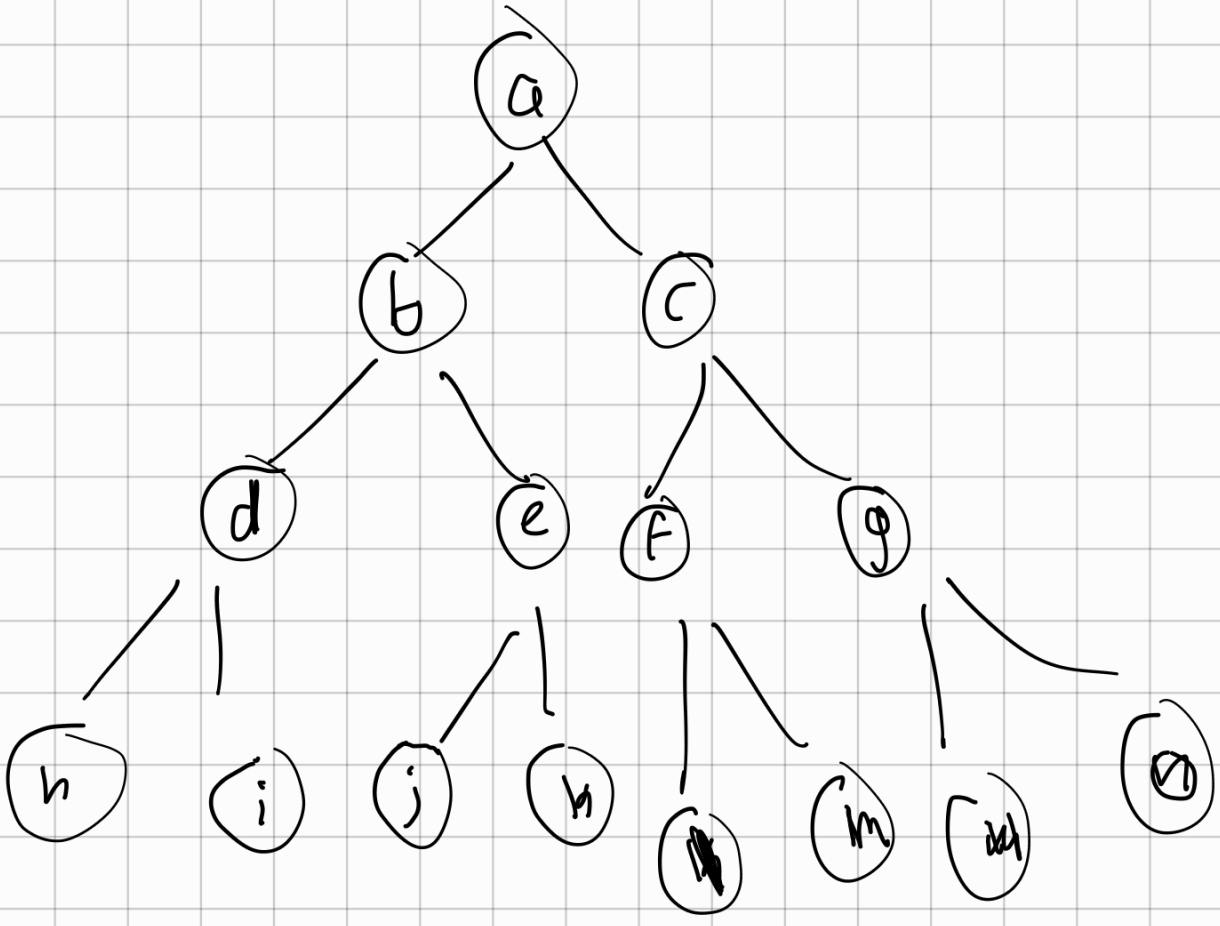
3. Recursively list the nodes in <A>'s right subtree

Code:

```
def subtree_iter(A):
    if A.left:
        yield from A.left.subtree_iter()
    yield A
    if A.right:
        yield from A.right.subtree_iter()
```



d → b → e → a → f → c → g



h → d → i → b → j → e → k → a → l → f → m → c → n → g → o

Tree Navigation

It will be useful to be able to navigate the nodes in their traversal order efficiently.

Most straightforward operation is to find the node in a given node's subtree that appears first or last in traversal order.

First node in inorder traversal order

↳ keep walking left if a left child exists

O(h) time: each step of the recursion moves down the tree

```
def subtree_first(A):
    if A.left:
        return A.left.subtree_first()
    else:
        return A
```

Second node in inorder traversal order

↳ keep walking right if a right child exists

O(h) time: each step of the recursion moves down the tree.

```
def subtree_last(A):
    if A.right:
        return A.right.subtree_last()
    else:
        return A
```

- Node's successor: The next node in the traversal order

To find the succesor of a node

<A>, if <A> has a right child, then <A>'s successor will be the first node in the right child's subtree.

O(h) → the algorithm only walks down the tree to find the successor

Otherwise, <A>'s successor cannot exist in <A>'s subtree, so we walk up the tree to find the lowest ancestor of <A> such that <A> is in the ancestor's left subtree.

O(h) → the algorithm only walks up the tree to find the succesor

Code:

```
def successor(A):
  if A.right:
     return A.right.subtree_first()

  while A.parent and (A is A.parent.right):
      A = A.parent
  return A.parent
```

- Predecessor of a node: The previous node in inorder traversal order

To find the predecessor of a node <A>:

If the node <A> has a left child, then <A>'s predecessor will be the last node in the left subtree.

#O(h): The algorithm walks down the tree to find the node's predecessor

Otherwise, <A>'s predecessor cannot exist in <A>'s right subtree, so we walk up the tree to find <A>'s lowest ancestor such that <A> is in the ancestor's right subtree.

$O(h)$ → The algorithm only walk up the tree to find the predecessor.