

# CS213/293 Data Structure and Algorithms 2024

## Lecture 5: Tree

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-08-20

# Topic 5.1

## Tree

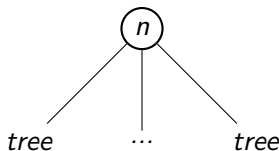
# Tree

## Definition 5.1

A *tree* is either a node



or the following structure consisting of a node and a set of children trees.



The above is **our first** recursive definition.

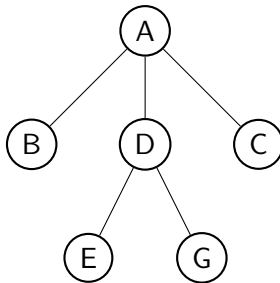
## Exercise 5.1

Does the above definition include *infinite* trees? How would you define an infinite tree?

## Example: tree

### Example 5.1

An instance of tree.



## Some tree terminology(2)

For nodes  $n_1$  and  $n_2$  in a tree  $T$ .

### Definition 5.2

$n_1$  is *child* of  $n_2$  if  $n_1$  is immediately below  $n_2$ . We write  $n_1 \in \text{children}(n_2)$ .

### Definition 5.3

We say  $n_2$  is *parent* of  $n_1$  if  $n_1 \in \text{children}(n_2)$  and write  $\text{parent}(n_1) = n_2$ .  
If there is no such  $n_2$ , we write  $\text{parent}(n_1) = \perp$ . **Root**

### Definition 5.4

$n_1$  is *ancestor* of  $n_2$  if  $n_1 \in \text{parent}^*(n_2)$ . We write  $n_1 \in \text{ancestors}(n_2)$ .  
 $n_2$  is *descendant* of  $n_1$  if  $n_1 \in \text{ancestor}(n_2)$ . We write  $n_1 \in \text{descendants}(n_2)$ .

**Commentary:** For a function  $f(x)$ , we define  $f^*(x) = y | y = f(\dots f(x))$ , i.e., the function is applied 0 or more times (informal definition). What would be a mathematically formal definition?

# Some tree terminology

## Definition 5.5

$n_1$  and  $n_2$  are *siblings* if  $\text{parent}(n_1) = \text{parent}(n_2)$ .

## Definition 5.6

$n_1$  is a *leaf* if  $\text{children}(n_1) = \emptyset$ .

$n_1$  is an *internal node* if  $\text{children}(n_1) \neq \emptyset$ .

## Definition 5.7

$n_1$  is a *root* if  $\text{parent}(n_1) = \text{Null}$ .

## Exercise 5.2

*Can the root be an internal node? Can the root be a leaf?*

Root with children is an internal node and without children i.e. single node is a leaf node.

## Example: Tree terminology

$B$ ,  $D$ , and  $C$  are children of  $A$ .

$D$  is the parent of  $G$ .

$A$  is an ancestor of  $G$  and  $E$  is a descendant of  $A$ .

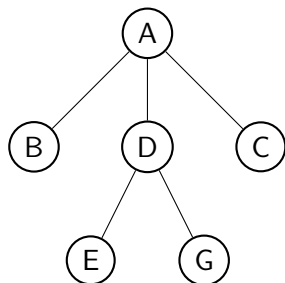
$A$  is an ancestor of  $A$ .

$G$  and  $E$  are siblings.

$B$ ,  $E$ ,  $G$ , and  $C$  are leaves.

$A$  and  $D$  are internal nodes.

$A$  is the root.



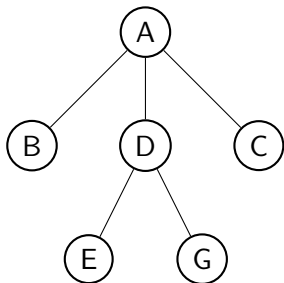
# Degree of nodes

## Definition 5.8

We define the degree of a node  $n$  as follows.

$$\text{degree}(n) = |\text{children}(n)|$$

## Example 5.3



$$\text{degree}(A) = 3$$

$$\text{degree}(B) = 0$$

$$\text{degree}(D) = 2$$



## Label of tree

Usually, we store data on the tree nodes.

We define the  $label(n)$  of a node  $n$  as the data stored on the node.

## Level/Depth and height of nodes

### Definition 5.9

*We define the level/depth of a node  $n$  as follows.*

$$\text{level}(n) = \begin{cases} 0 & \text{if } n \text{ is a root} \\ \text{level}(n') + 1 & n' \in \text{parent}(n) \end{cases}$$

### Definition 5.10

*We define the height of a node  $n$  as follows.*

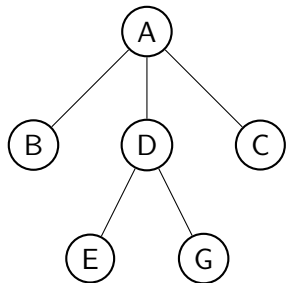
$$\text{height}(n) = \max(\{\text{height}(n') + 1 \mid n' \in \text{children}(n)\} \cup \{0\})$$

### Exercise 5.3

*Why do we need to take a union with 0 in the definition of height? leaf node's height is 0.*

## Example: Level(Depth) and height of nodes

### Example 5.4



$$\text{level}(A) = 0$$

$$\text{level}(B) = 1$$

$$\text{level}(E) = 2$$

$$\text{height}(E) = 0$$

$$\text{height}(D) = 1$$

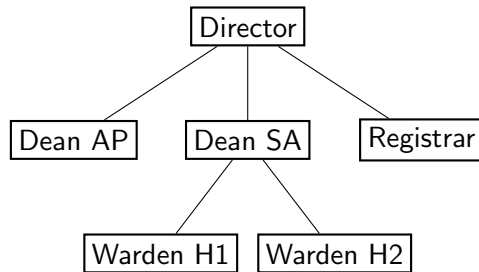
$$\begin{aligned}\text{height}(A) &= \max(\{\text{height}(B) + 1, \\ &\quad \text{height}(D) + 1, \\ &\quad \text{height}(C) + 1\} \cup \{0\}) \\ &= \max(\{1, 2, 1\} \cup \{0\}) = 2\end{aligned}$$

# Why do we need trees?

A tree represents a hierarchy.

## Example 5.5

- *Organization structure of an organization*

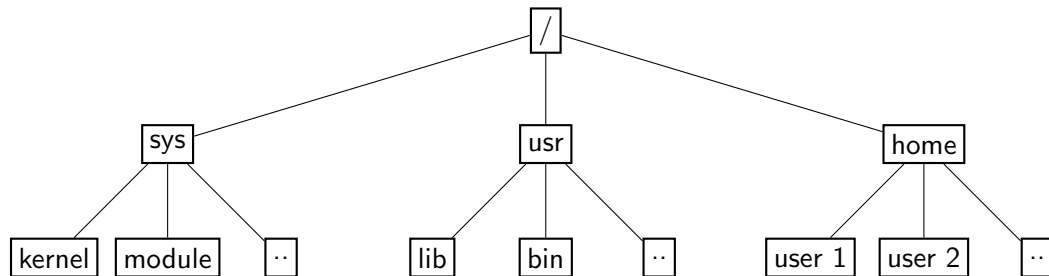


## Example: File system

Files are stored in Trees in Linux/Windows.

### Example 5.6

*Part of a Linux file system.*



## Topic 5.2

### Binary tree

# Ordered tree

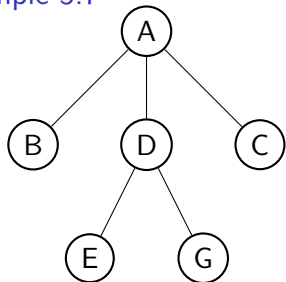
## Definition 5.11

A tree is an *ordered tree* if we assign an *order among children*.

## Definition 5.12

Let  $n$  be a node. In an ordered tree,  $\text{children}(n)$  is a *list* instead of a set.

## Example 5.7



In a tree, we define the children as follows.

$$\text{children}(A) = \{B, D, C\}$$

In an ordered tree, we define the children as follows.

$$\text{children}(A) = [B, D, C]$$

# Binary tree

## Definition 5.13

An ordered tree  $T$  is a *binary tree* if  $|\text{children}(n)| \leq 2$  for each  $n \in T$ .

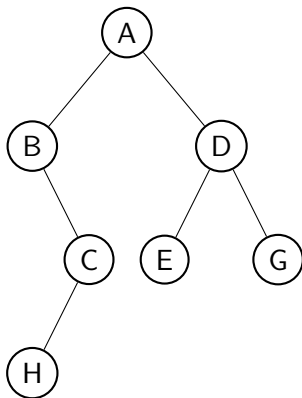
We define the left and right child of  $n$  as follows.

- ▶ if  $\text{children}(n) = [n_1, n_2]$ ,
  - ▶  $\text{left}(n) = n_1$  and  $\text{right}(n) = n_2$ .
- ▶ If  $\text{children}(n) = [n_1]$ ,  $n_1$  is either left or right child.
  - ▶  $\text{left}(n) = n_1$  and  $\text{right}(n) = \text{Null}$ , or
  - ▶  $\text{left}(n) = \text{Null}$  and  $\text{right}(n) = n_1$ .
- ▶ If  $\text{children}(n) = []$ ,
  - ▶  $\text{left}(n) = \text{Null}$  and  $\text{right}(n) = \text{Null}$ .



## Example: binary tree

### Example 5.8

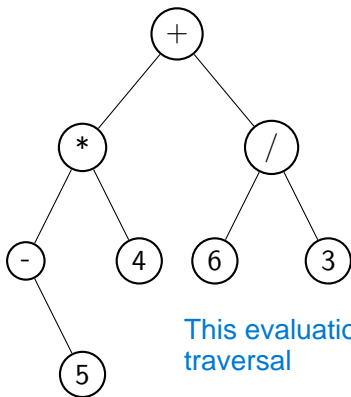


*E* is the left and *G* is the right child of *D*. *C* is the right child of *B*. *B* has no left child.

# Usage of binary tree: representing expressions

## Example 5.9

*Representing mathematical expressions*



This evaluation can be done using inorder traversal

## Exercise 5.4

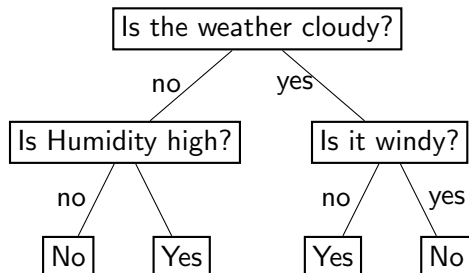
- Why do we need an ordered tree?*
- How would you evaluate a mathematical expression that is given as a binary tree?*

# Usage of binary tree: decision trees in AI

## Example 5.10

*Does one want to play given the weather?*

*Given the behavior, we may learn the following tree.*



# Complete binary tree

## Definition 5.14

A binary tree is **complete** if the height of the root is  $h$  and every level  $i \leq h$  has  $2^i$  nodes.

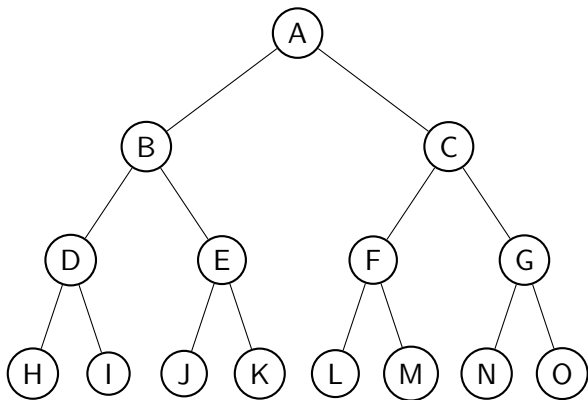
Leaves are only at level  $h$ .

The number of leaves =  $2^h$ .

Number of internal nodes =  $1 + 2 + \dots + 2^{h-1}$   
 $= 2^h - 1$ .

The total number of nodes is  $2^{h+1} - 1$ .

## Example 5.11



## Exercise 5.5

- Prove/Disprove: if no node in the binary tree has a single child, the binary tree is complete.
- What fraction of nodes are leaves in a complete binary tree?

# Maximum and minimum height of a binary tree

## Exercise 5.6

*Let us suppose there are  $n$  nodes in a binary tree.*

- ▶ *What is the minimum height of the tree?*
- ▶ *What is the maximum height of the tree?  $n-1$*

**Commentary:** For a given height  $h$ , a complete binary tree has  $2^{h+1} - 1$  nodes. All other binary trees with the height  $h$  have fewer nodes. Therefore,  $n \leq 2^{h+1} - 1$ . Therefore,  $\log_2 \frac{n+1}{2} \leq h$ . The maximum possible height for  $n$  nodes is  $n - 1$ . Therefore,  $\log_2 \frac{n+1}{2} \leq h \leq n - 1$ .

# Leaves of binary tree

## Theorem 5.1

For a binary tree,  $|leaves| \leq 1 + |internal\ nodes|$ .

### Proof.

We will prove the theorem by induction over the structure of a tree (Recall the recursive definition of a tree).

#### Base case:



We have a single node.

$|leaves| = 1$  and  $|internal\ nodes| = 0$ . Case holds.

...

Commentary:  $|A|$  indicates the size of set  $A$ .

## Leaves of binary tree(2)

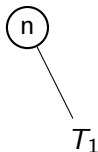
Proof(continued).

### Induction step:

We have two cases in the induction step: Root has one child or two children.

#### Case 1:

Let tree  $T$  be constructed as follows.



For  $T_1$ , let  $|leaves| = \ell_1$  and  $|internal\ nodes| = i_1$ .

$T$  has  $\ell_1$  leaves and  $i_1 + 1$  internal nodes.

By the induction hypothesis,  $\ell_1 \leq 1 + i_1$ .

Therefore,  $\ell_1 \leq 1 + i_1 + 1$ .

Therefore,  $\ell_1 \leq 1 + (i_1 + 1)$ . Case holds.

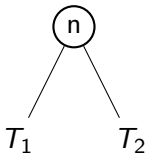
...

## Leaves of binary tree(3)

Proof(continued).

### Case 2:

Let tree  $T$  be constructed as follows.



For  $T_1$ , let  $|leaves| = \ell_1$  and  $|internal\ nodes| = i_1$ .

For  $T_2$ , let  $|leaves| = \ell_2$  and  $|internal\ nodes| = i_2$ .

$T$  has  $\ell_1 + \ell_2$  leaves and  $i_1 + i_2 + 1$  internal nodes.

By induction hypothesis,  $\ell_1 \leq 1 + i_1$  and  $\ell_2 \leq 1 + i_2$ .

Therefore, we have  $\ell_1 + \ell_2 \leq 2 + i_1 + i_2$ .

Therefore,  $\ell_1 + \ell_2 \leq 1 + (i_1 + i_2 + 1)$ . Case holds.

□



# Maximum and minimum number of leaves

Let  $n$  be the number of nodes in a binary tree  $T$ .

Due to the previous theorem, we know  $|\text{leaves}| \leq 1 + |\text{internal nodes}|$ .

Since  $|\text{leaves}| + |\text{internal nodes}| = n$ ,  $|\text{leaves}| \leq 1 + n - |\text{leaves}|$ .

$$|\text{leaves}| \leq \frac{(n+1)}{2}.$$

## Exercise 5.7

- When do  $|\text{leaves}|$  meet the inequality? **for CBT**
- When is the number of leaves minimum? **when tree is skew**

**Commentary:** If  $T$  is complete, the number of leaves is  $\frac{(n+1)}{2}$ .

## Topic 5.3

### Representing Tree

## Container for tree

Tree has to be implemented by programmer!

There is no C++ container for tree.

Trees are the backbone of many abstract data structures.

For some reason, it is not explicitly there.

### Exercise 5.8

*Why there is no tree container in C++ STL? (Let us ask ChatGPT)*

**Commentary:** I guess that we rarely explicitly need trees in our programming. We usually have higher goals such as stack, queue, set, and map, which may need a tree as an internal data structure, but users need not be exposed. However, there are applications where there is a clear need for trees. For example, the representation of arithmetic expressions. In my programming, whenever I needed a tree. I have implemented it myself.

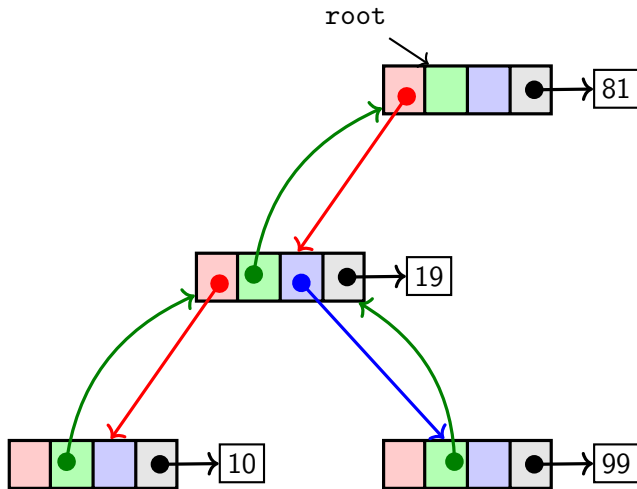
# Representation of a binary tree on a computer

## Definition 5.15

A binary tree consists of nodes containing four pointer fields.

- ▶ *left child*
- ▶ *parent*
- ▶ *right child*
- ▶ *label*

An additional root pointer points to the root of the tree.



The pointers that are not pointing anywhere are NULL.

Exercise 5.9 If don't wanna go back then omit it  
Do we need the parent pointer?

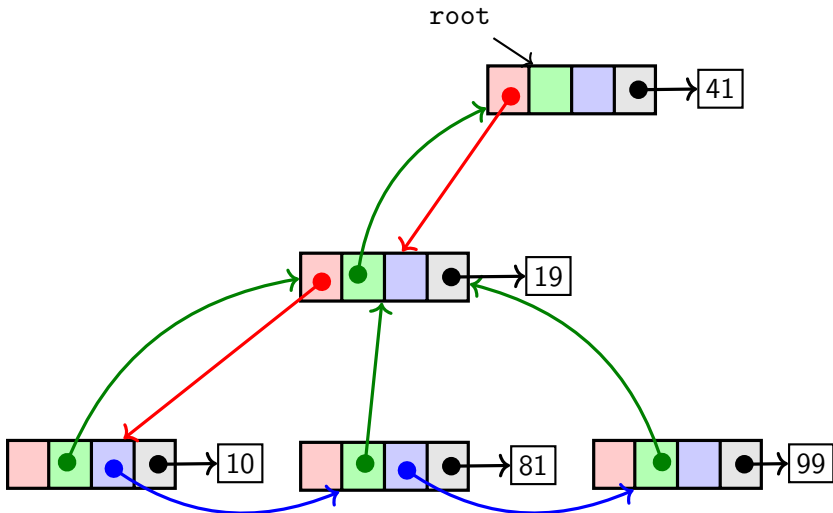
# Representation of a tree on a computer

## Definition 5.16

A tree consists of nodes containing four pointer fields.

- ▶ *first child*
- ▶ *parent*
- ▶ *next sibling*
- ▶ *label*

An additional root pointer points to the root of the tree.



Exercise 5.10 It looks unordered, but it should be ordered.

Are we representing an ordered tree or an unordered tree?

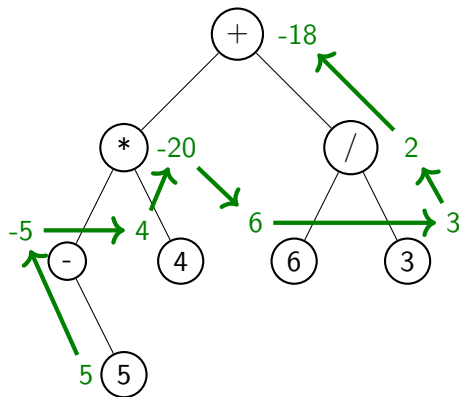
## Topic 5.4

### Tree walks

## Application : Evaluating an expression

### Example 5.12

If we want to evaluate an expression represented as a binary tree, we need to **visit** each node and evaluate the expression in a certain order.



In green, we have evaluated the value of the node. The path indicates the order of evaluation.

# Tree walks

Visiting nodes of a tree in a certain order are called **tree walks**.

There are two kinds of walks for trees.

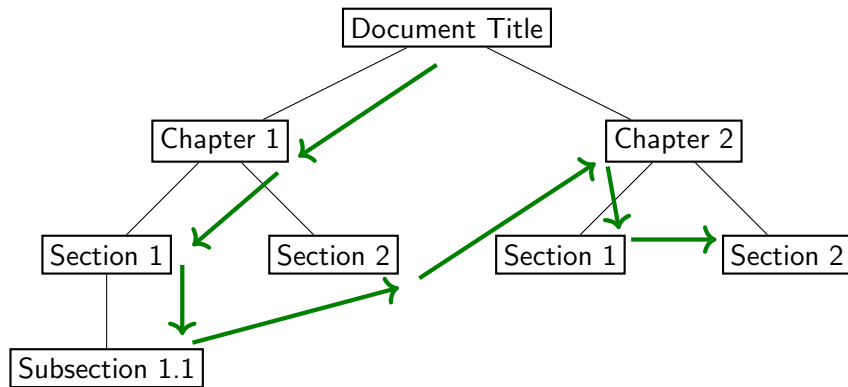
- ▶ preorder: visit **parent** first
- ▶ postorder: visit **children** first



## Example: preorder

### Example 5.13

*Let a document be stored as a tree. We read the document in preorder.*



# Preorder/Postorder walk

---

**Algorithm 5.1:** PreOrderWalk( $n$ )

---

```
1 visit( $n$ );  
2 for  $n' \in \text{children}(n)$  do  
3    $\lfloor$  PreOrderWalk( $n'$ );
```

---

---

**Algorithm 5.2:** PostOrderWalk( $n$ )

---

```
1 for  $n' \in \text{children}(n)$  do  
2    $\lfloor$  PostOrderWalk( $n'$ );  
3 visit( $n$ );
```

---

The first example of expression evaluation is postorder walk.

**Commentary:** visit( $v$ ) is some action taken during the walk.

# Walking on ordered tree

How do we walk on an ordered tree?

For an ordered tree, we may visit children in the given order among siblings.

We may have choices to change the order of visits among ordered siblings.

**Commentary:** Our description of the algorithm works for both ordered and unordered trees. Our algorithm does not specify the order of visits of siblings for unordered trees. Please pay attention to the subtle differences among trees, ordered trees, and binary trees.

## Topic 5.5

### Walking binary trees

## Preorder/Postorder walk over binary trees

We have more structure in binary trees. Let us write the algorithm for walks again.

---

**Algorithm 5.3:** PreOrderWalk(*n*)

---

```
1 if n == Null then
2   return
3 visit(n);
4 PreOrderWalk(left(n));
5 PreOrderWalk(right(n));
```

---

---

**Algorithm 5.4:** PostOrderWalk(*n*)

---

```
1 if n == Null then
2   return
3 PostOrderWalk(left(n));
4 PostOrderWalk(right(n));
5 visit(n);
```

---

# Inorder walk of binary trees

## Definition 5.17

*In an inorder walk of a binary tree, we visit the node after visiting the left subtree and before visiting right subtree.*

---

**Algorithm 5.5:** InOrderWalk( $n$ )

---

```
1 if  $n == Null$  then  
2   return  
3 InOrderWalk(left( $n$ ));  
4 visit( $n$ );  
5 InOrderWalk(right( $n$ ));
```

---

## Exercise 5.11

*Given a complete binary tree with 7 nodes, label the nodes so that the preorder, inorder, and postorder traversals produce the sequence 1,2,...,7.*

## Application : Printing an expression

To print an expression (without unary minus), we need to **visit** the nodes in inorder.

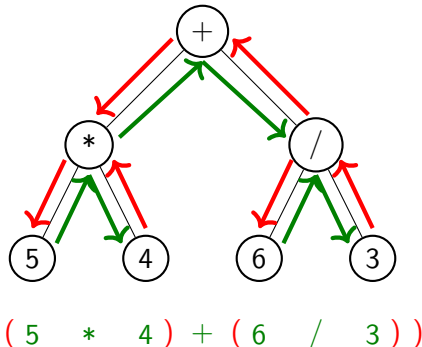
---

### Algorithm 5.6: PrintExpression(n)

---

```
1 if n is leaf then  
2   print(label(n));  
3   return  
4 print("(");  
5 PrintExpression(left(n));  
6 print(label(n));  
7 PrintExpression(right(n));  
8 print(")");
```

---



### Exercise 5.12

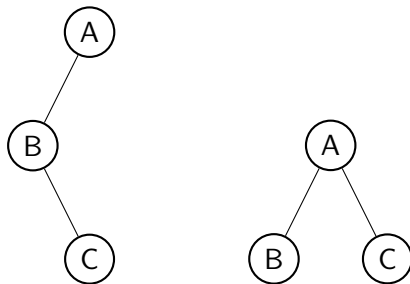
- Modify the above algorithm to support unary minus.
- What will happen if "if" at line 1 is replaced by "if  $n == \text{NULL}$  then return"?

then each label will have parentheses around it

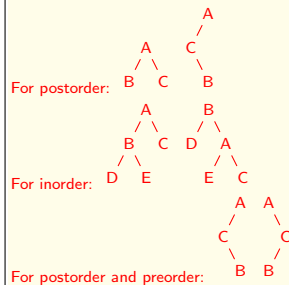
**Commentary:** The order of the walk is the pattern of recursive calls and actions on nodes. An application may need mixed action pattern. In the above printing example, we need to print parentheses before and after making recursive calls. The parentheses are printed pre/post-order. All three walks are present in the above algorithm.

## Many trees have the same walks

The following two ordered trees have the same preorder walks.



Commentary: Answer:



### Exercise 5.13

- Give two binary trees that have the same postorder walks.
- Give two binary trees that have the same inorder walks.
- Give two binary trees that have the same postorder and preorder walks.



## Topic 5.6

### Tutorial problems

## Exercise: paths in a tree

### Exercise 5.14

*Given a tree with a maximum number of children as  $k$ . We give a label between 0 and  $k-1$  to each node with the following simple rules. (i) the root is labeled 0. (ii) For any vertex  $v$ , suppose that it has  $r$  children, then arbitrarily label the children as  $0, \dots, r-1$ . This completes the labeling. For such a labeled tree  $T$ , and a vertex  $v$ , let  $\text{seq}(v)$  be the labels of the vertices of the path from the root to  $v$ . Let  $\text{Seq}(T) = \{\text{seq}(w) \mid w \in T\}$  be the set of label sequences. What properties does  $\text{Seq}(T)$  have? If a word  $w$  appears what words are guaranteed to appear in  $\text{Seq}(T)$ ? How many times does a word  $w$  appear as a prefix of some words in  $\text{Seq}(T)$ ?*

# Lowest common ancestor(LCA)

## Definition 5.18

For two nodes  $n_1$  and  $n_2$  in a tree  $T$ ,  $LCA(n_1, n_2, T)$  is a node in  $ancestors(n_1) \cap ancestors(n_2)$  that has the largest level. Two stack implementation is good one. Keep all the ancestor in stack and check the point till which they have common ancestors and done.  
Recursive approach is also good in which check if current root's descendent are  $n_1$  and  $n_2$  then return that root with highest level.

## Exercise 5.15

Write a function that returns  $lca(v, w, T)$ . What is the time complexity of the program?

## Exercise: paths in a tree

### Exercise 5.16

Given  $n \in T$ , Let  $f(n)$  be a vector, where  $f(n)[i]$  is the number of nodes at depth  $i$  from  $n$ .

- ▶ Give a recursive equation for  $f(n)$ .  
Initialize a vector with size  $V(n,0)$  and make the recursive call incrementing the level in parameter and on visiting make  $V[\text{level}]++$   
If at a level siblings are in form of list, then count them add that amount to  $V[\text{level}]$
- ▶ Give a pseudo code to compute the vector  $f(\text{root}(T))$ . How is the time complexity of the program? Time complexity would be  $O(n)$ , since gotcha meet all the nodes.

The uniqueness of walks if two walks are the same.

### Exercise 5.17

*Give an algorithm for reconstructing a binary tree if we have the preorder and inorder walks.*  
preorder visits root first hence make that root and check boundness in the inorder walk, as left of that key would form left subtree and right of the key would form right subtree.

### Exercise 5.18

*Let us suppose all internal nodes of a binary tree have two children. Give an algorithm for reconstructing the binary tree if we have the preorder and postorder walks.*

## Topic 5.7

### Problems

## Exercise: no single child (Quiz 2023)

### Exercise 5.19

*Prove/Disprove: If no node in the binary tree has single child,  $|leaves| = 1 + |internal\ nodes|$ .*

## Exercise: mean level

**Exercise 5.20** supposing the mean is for all the nodes in tree.

a. Suppose that you are given a binary tree, where, for any node  $v$ , the number of children is no more than 2. We want to compute the mean of  $ht(v)$ , i.e., the mean level of nodes in  $T$ . Write a program to compute the mean level. a map would help with recursive calls, keep the number of vertex at a level then just weighted average in the end would do it.

b. Suppose that we are given the level of all leaves in the tree. Can we compute the mean height? Given a sequence  $(n_1, n_2, \dots, n_k)$  of the levels of  $k$  leaves, is there a binary tree with exactly  $k$  leaves at the given levels?



# Reconstructing tree from preorder walks

## Exercise 5.21

*Let us suppose we can calculate the number of children of a node by looking at the label of a node of a binary tree, e.g., arithmetic expressions. Give an algorithm for reconstructing the binary tree if we have the preorder walk.* since current is always a root and we also know how much children it has (B tree), then we know where to halt when we'd make a left and a right i.e. if no children then we'd come back from that direction and make the other direction of tree.

## Exercise: previous print

### Exercise 5.22

*For a given binary tree, let  $\text{prevPrint}(T, a)$  give the node  $n'$  such that  $\text{label}(n')$  will appear just before  $\text{label}(n)$  in the inorder printing of  $T$ . Give a program that implements  $\text{prevPrint}$ .*

It is asking for predecessor in inorder walk of BT. There is a very efficient iterative  $\log(n)$  solution to this problem.  
Or even the inorder would do just fine just keep track of the previous node processed.

## Exercise: level-order walk

### Exercise 5.23

*Give an algorithm for walking a tree such that nodes are visited in the order of their level. Two nodes at the same level can visit in any order.*

BFS

### Exercise 5.24

*Give an algorithm for walking a tree such that nodes are visited in the order of their height.*

I guess there isn't possible just once.

End of Lecture 5