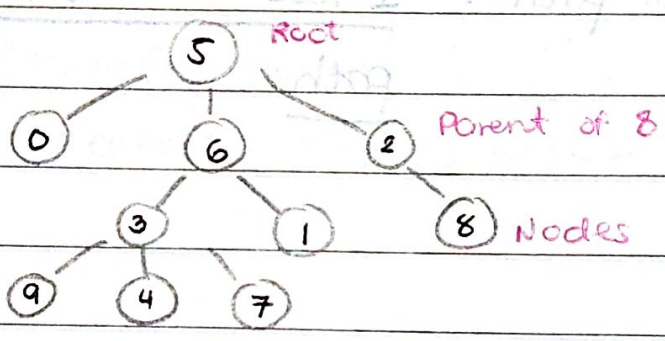
NO:
DATE:

- TREES -

*Trees

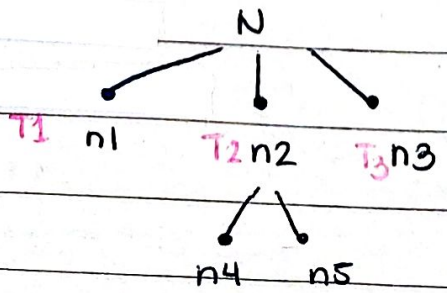
- a collection of elements called nodes
- start of tree is the root, only ONE root node
- relation: parenthood
  ↳ places a hierarchical structure on nodes



*Recursive Definition

(1.) Single node = tree
: also the root of the tree

(2.) A tree can be constructed using a single node n & subtrees $n_1, n_2, n_3, \ldots$ by making node n THE PARENT of the root nodes (children of n)



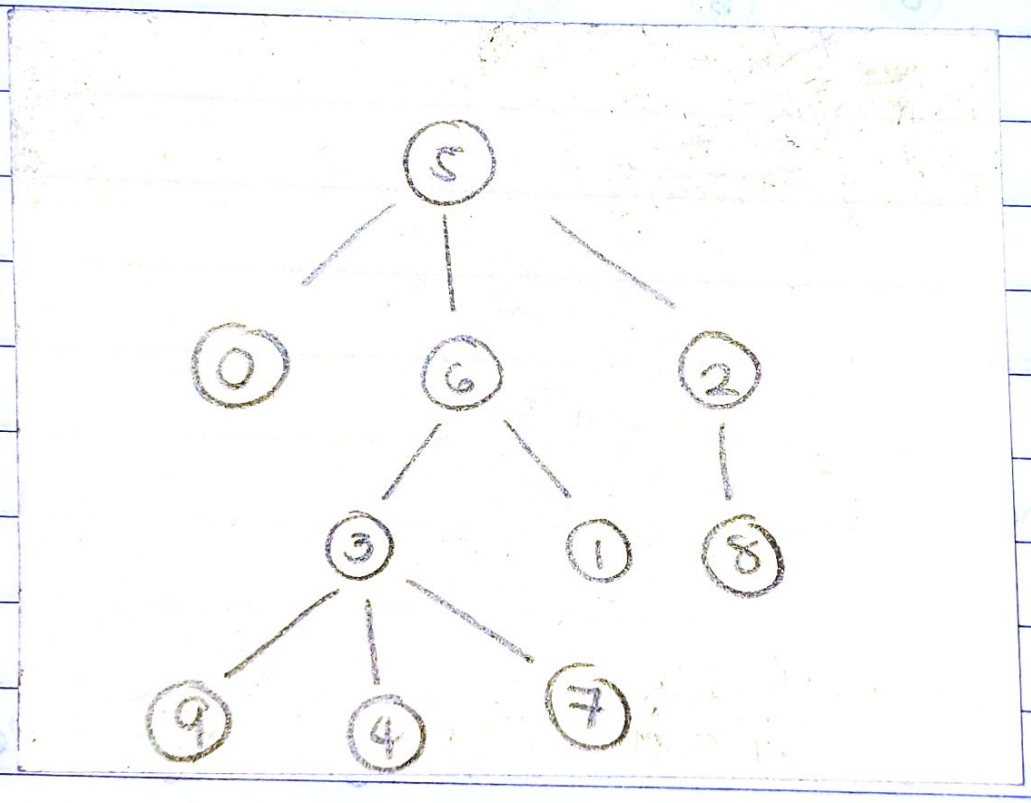
* Path

- path from n_1 to n_k is a sequence of nodes in a tree:

n_i is the parent of n_{i+1}

$1 \leq i < k$

- length of path : 1 less than number of nodes in path



Example: Path from n_1 (5) to n_k (3)

n_1 n_2 n_3

$5 \rightarrow 6 \rightarrow 3$ ✓

$6 \rightarrow 5 \rightarrow 2$ ✗

* Remember, nodes have to be SEQUENTIAL!

Length of path : 2

IDEAS COME FROM TIESONG

DATE:

* Ancestor and Descendant

- **Ancestor**
  - a is an ancestor of b, if a path from $a \rightarrow b$ exists
  - a node can be its own ancestor & descendant

  > **Proper Ancestor**
  - ancestor of a node other than itself

- **Descendant**
  - b is a descendant of a, if a path from $a \rightarrow b$ exists

  > **Proper Descendant**
  - descendant of a node other than itself

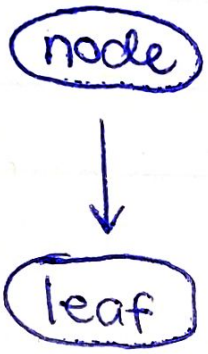
*The root DOES NOT have a proper ancestor.

* Leaf, Null & Subtree

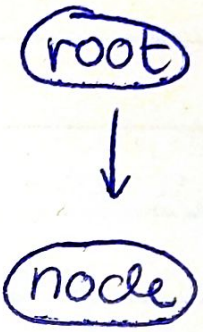
- **Leaf**
  - node with no proper descendants
- **Null Tree (Λ)**
  - tree with no nodes
- **Subtree**
  - a node, together with all its descendants

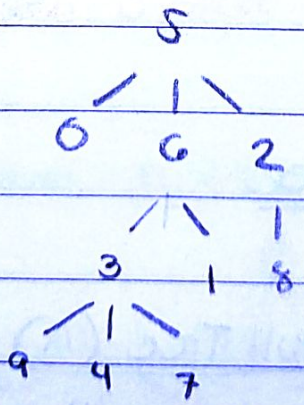
* Height & Depth

- **Height of a Node**
  - length of the longest path from the node to a leaf



- **Depth of a Node**
  - length of the unique path from the root to that node





Height & Depth of (6)

Example: (6) → (9) : 2 Length

(6) → (4) : 2

(6) → (7) : 2

(6) → (1) : 1

Longest path : 2

Height

(5) → (6) = 1 Length

Depth : 1

IDEAS COME FROM TIESONG

* Order of Nodes

- children of a node is ordered from: left-to-right

* Ordering of Sibling

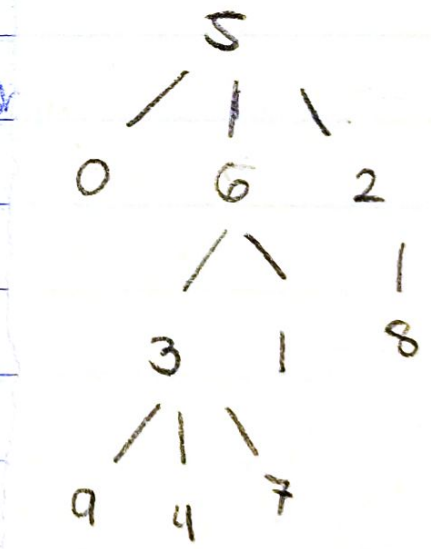
- nodes with the same parents
- can be used to compare 2 nodes that are not related by ancestor-descendent relationship

SIBLINGS

Example: 0, 6, 2 = children of (5)

9, 4, 7 = children of (3)

- 6 & 2 are NOT related by ancestor-descendant relationship
- 5 & 1 ARE related bcs 5 is an ancestor of 1 & vice versa.



* Relevant Rule

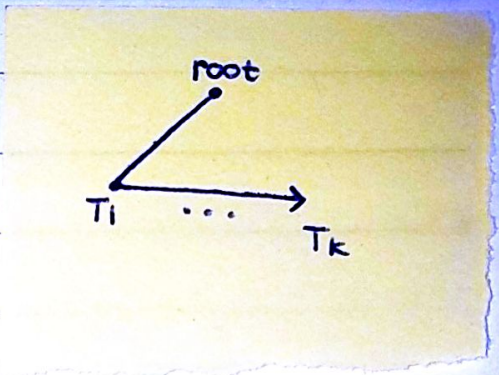
" If a and b are siblings, and a is TO THE LEFT of b, then ALL DESCENDATS of a are TO THE LEFT of ALL THE DESCENDATS of b. "

* If 6 is to the left of 2, the all the descendants of 6, including 6, is to the left of all of the descendants of 2.

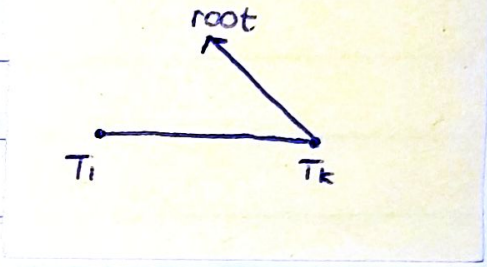
* Nodes: 0, 3, 9, 4, 7 is to the left of 1.

*Systematic Ordering of Nodes (Listings / Traversals)

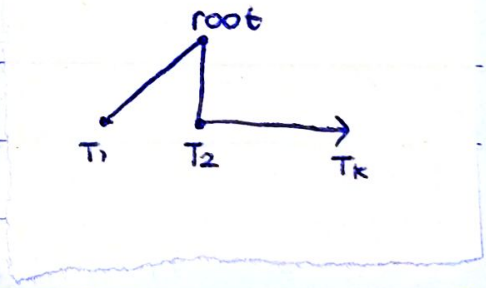
1. Preorder
   - root of T
   - nodes of T in preorder
   - etc.



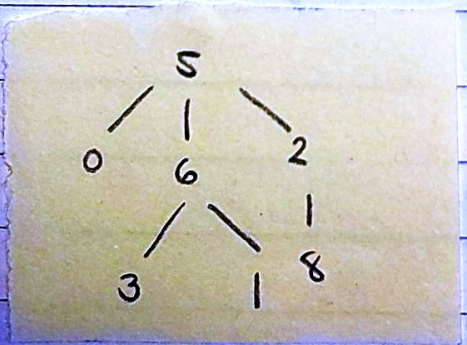
2. Postorder
   - nodes of T_1 in postorder
   - nodes of T_2
   - root of T



3. Inorder
   - nodes of T_1 in inorder
   - root of T
   - nodes of T_2 in inorder



Example:



1. Preorder = 5 0 6 3 1 2 8
2. Postorder = 0 3 1 6 8 2 5
3. Inorder = 0 5 3 6 1 8 2

* For Binary Trees, with OWE CHILD

- identify if child is on the LEFT / RIGHT

- draw broken line for easy tracing

DATE:

* Labeled and Expression Trees

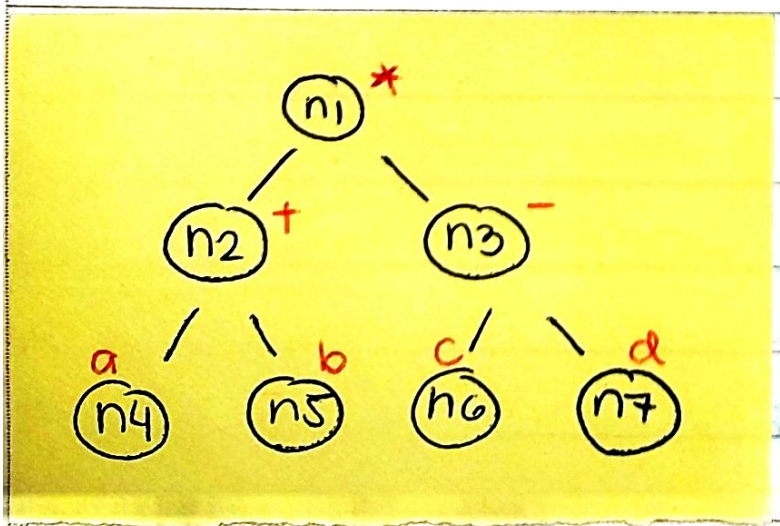
- **Label** - the "value" stored at the node

element : list label : tree

- **Labeled Tree**
  - tree whose nodes have labels

- **Expression Tree**
  - tree whose every leaf is labeled by an operand
  - every interior node is labeled by an operator
    ↳ any node that is NOT a le

Leaves: operands (A & B)
Inode: operators (+/-)



Expression:

(a+b) * (c-d)

- read in infix notation

root root root
n4 → n2 → n5 → n1 → n6 → n3 → n7

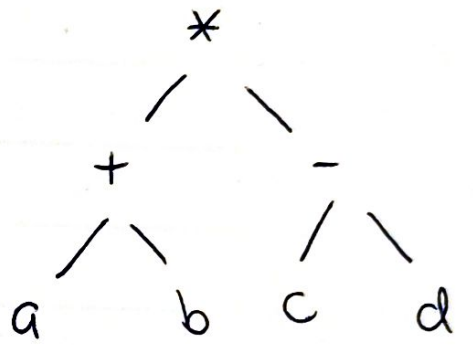
* Writing Expression

1. Prefix Notation - preorder
2. Infix Notation - inorder
3. Postfix Notation - postorder

Determine:

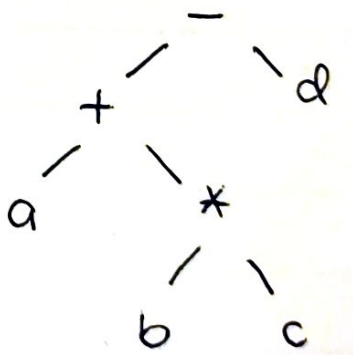
1. Prefix = * + a b - c d
2. Infix = a + b * c - d
3. Postfix = a b + c d - *

①



②

1. Infix = a + b * c - d
2. Postfix = a b c * + d -



* Polish Notation

- also known as prefix notation
- operator is placed BEFORE operands
- by Jan Lukasiewicz in 1924

* Reverse Polish

- the postfix notation
- operator is AFTER the operands

* No need for parenthesis bcs operators are arranged in their precedence associativity rule.

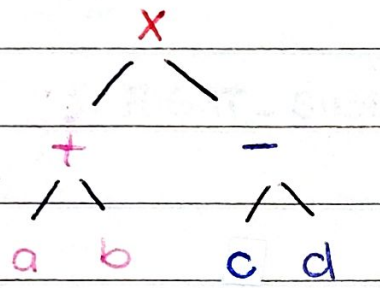
IDEAS COME FROM TJESONG

* Drawing Trees from Mathematical Expression

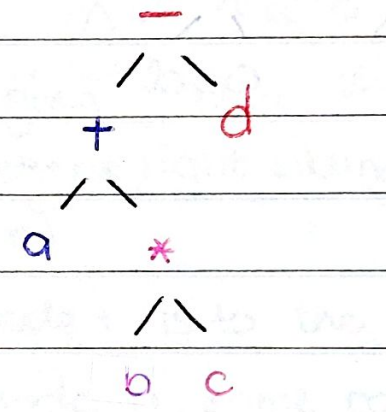
- a tree can be drawn given a notation

Example: Given in POSTFIX NOTATION, draw the tree.

① a b + c d - x



② a b c * + d -



* Postfix notation follows: "operand operand operator"

① a b + : OPRND OPRND OPRTR

once evaluated, result becomes OPERAND.

next, c d - : OPRND OPRND OPRTR

also becomes operand after.

now, (a b +) & (c d -) are OPERANDS followed by OPERATOR (x).

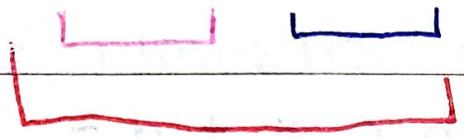
② b c * becomes OPERAND.

a (b c *) + : OPRND OPRND OPRTR

(a (b c *) +) d - : OPRND OPRND OPRTR

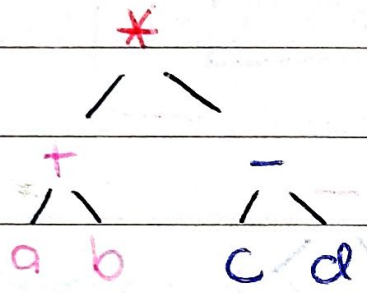
Example: Given in PREFIX NOTATION, draw the tree.

* + a b - c d



* Prefix notation follows:

"operator operand operand"



NO:

DATE:

* ADT Tree Operations

1. PARENT (n, T) - given node & tree; return parent of node n in tree T
   - if n is the root / the parent, return ∧ (null)
2. LEFTMOST_CHILD (n, T) - given node n & tree T
   - return leftmost child of n, ∧ if n is a leaf
3. RIGHT_SIBLING (n, T) - given node n & tree T,
   - returns right sibling of node n (r)
   - node r is to the right of node n, some parent p
4. LABEL (n, T) - returns label (value) of node n
5. CREATE (v, T_1, T_2, ...) - makes new root r with label v & gives it i children (roots of T_1, T_2, ... from left to right)
6. ROOT (T) - returns node (root) of tree T, ∧ if null tree
7. INITIALIZE (T) - prepares tree
8. MAKENULL (T)

* Ordering Functions

1. PREORDER (n, T) 2. POSTORDER (n, T)

DATE:

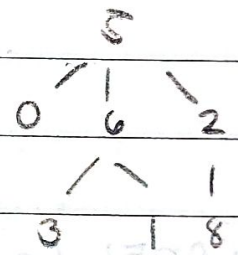
* Recursive Preorder Listing Function

- lists the labels of descendants n in preorder

```
void PREORDER (node n, Tree T) {
    node c;
    print (LABEL(n,T));

    c = LEFTMOST_CHILD(n,T);

    while (c != Λ) {
        PREORDER(c,T);
        c = RIGHT_SIBLING(c,T);
```

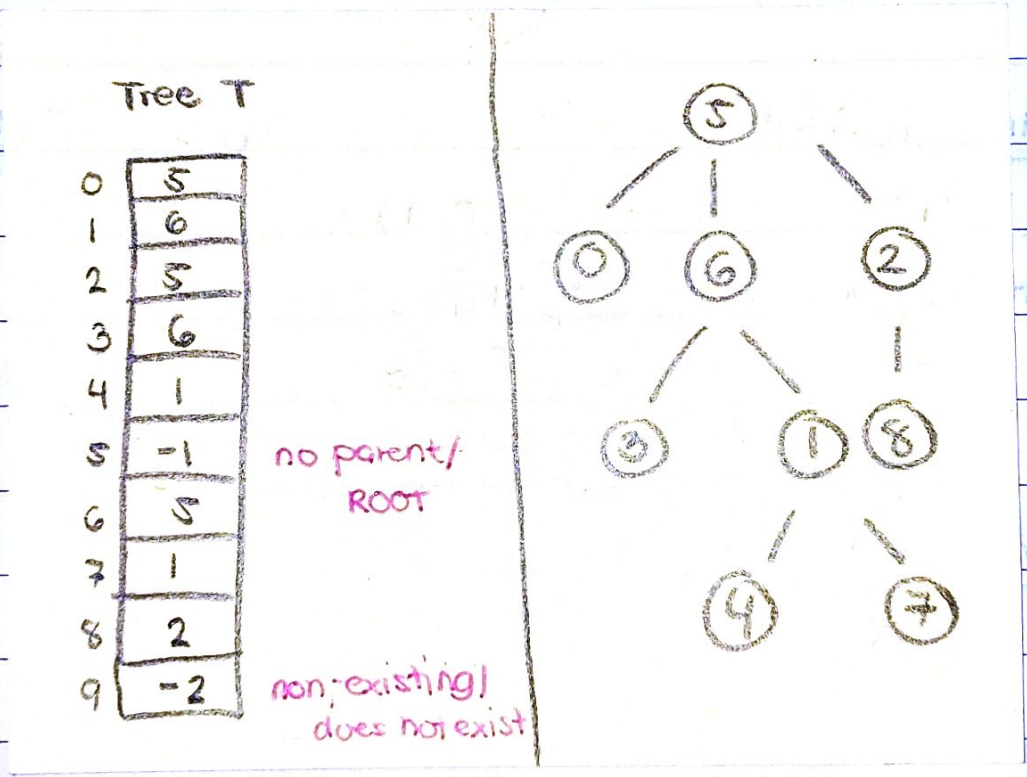


n = 6	
$n_1 = 6$, $c = \Lambda$	DISPLAY
$c_2 = 3$ X	LABEL - 6
$c_2 = 1$	LABEL - 3
$n_2 = 3$ $n_3 = 1$	LABEL - 1
$c_2 = \Lambda$ $c_3 = \Lambda$	
X X	
n = 2	
$n_1 = 2$	LABEL - 2
$c_1 = 8$	LABEL - 8
$c = \Lambda$ X	
$n_2 = 8$	
$c = \Lambda$	
X	

* Implementations

1.) Parent Pointer Representation

- uses an ARRAY
- index = node
- elem = parent
- runs at constant time O(1) when PARENT()
- looking for root is O(N), since has to start from index [0] & look for -1 (root)



* Disadvantages

- cannot give full info. of children (L/R)
- have to manually traverse thru array which has == parent node AND identify which is LM
- can be solved IF children are in INCREASING ORDER, so smallest child is leftmost, assuming an artificial order

NO:

DATE:

2.) Representation of Trees by List of Children

- linked list OR Cursor-based List
- array of linked list where index is element, connected/pointing to its descendants
- NULL pointer array means node has no children OR non-existing
  ↳ PROBLEM: can't distinguish

* Running Time

- **Finding Root**
  O(1)
  if add root variable

- **Finding Leftmost**
  O(1)
  always the first node

- **Finding Right**
  O(N)

- **Finding Parent**
  O(N)

```
#define SIZE 10
typedef struct node {
    int node;
    struct node *link;
} *List;

typedef struct {
    List Header[SIZE];
    int root;
} Tree;
```