CIS 2101

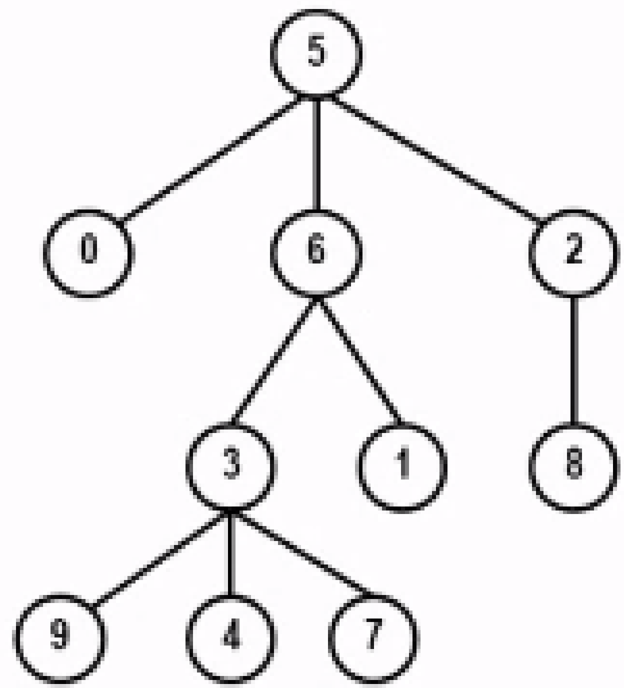
Trees

Tree

(Illustration and Definition)

- What is a Tree ?
  - A ***tree*** is a collection of elements called ***nodes***, one of which is distinguished as a ***root***, along with the relation ("***parenthood***") that places a hierarchical structure on the nodes

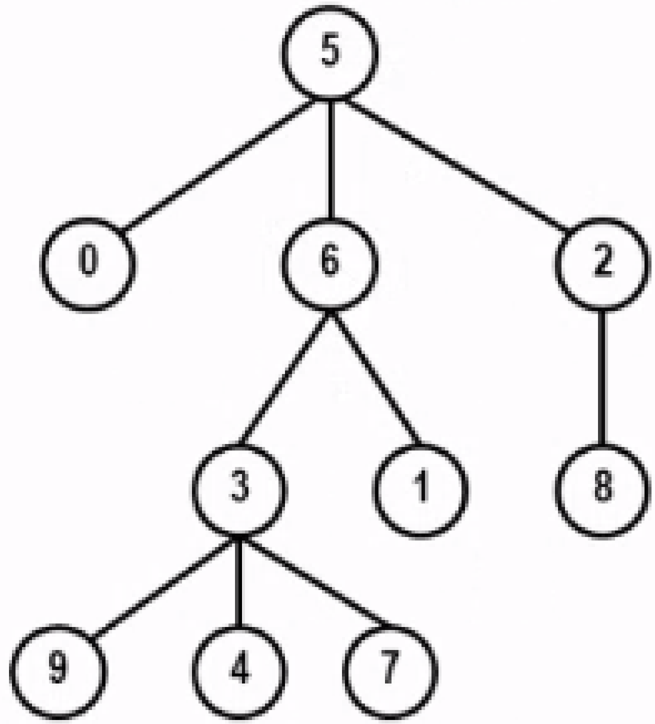
Illustration



Exercise 1

1. What are the nodes of the tree?
2. What is the root node ?
3. Parenthood:
    a) Parent of 6 ?
    b) Parent of 7 ?

c) Parent of 1 ?



Recursive Definition of Tree

1. A single node by itself is a tree. This node is also the root of the tree.
2. Suppose ***n*** is a node and $T_1, T_2, \ldots, T_k$ are trees with roots $n_1, n_2, \ldots, n_k$, respectively.

   We can construct a new tree by making ***n*** the parent of nodes $n_1, n_2, \ldots, n_k$.

   In this tree, ***n*** is the root and $T_1, T_2, \ldots, T_k$ are the ***subtrees of the root***.

   Nodes $n_1, n_2, \ldots, n_k$ are ***called the children of node n.***

Recursive Definition of Tree

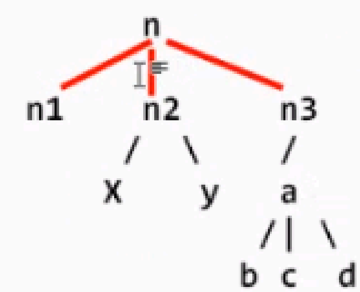
1. A single node by itself is a tree. This node is also the root of the tree.
2. Suppose ***n*** is a node and $T_1, T_2, \ldots, T_k$ are trees with roots $n_1, n_2, \ldots, n_k$, respectively.

   We can construct a new tree by making ***n*** the parent of nodes $n_1, n_2, \ldots, n_k$.

   In this tree, ***n*** is the root and $T_1, T_2, \ldots, T_k$ are the ***subtrees of the root***.

   Nodes $n_1, n_2, \ldots, n_k$ are ***called the children of node n.***

Prepared by chrisp 5

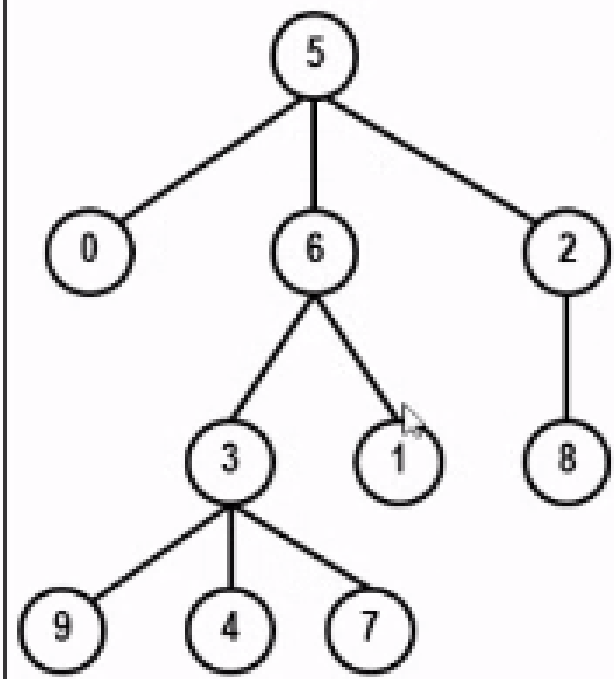


Path

The ***path from n_1 to node n_k*** is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ in a tree such that $\boldsymbol{n_i}$ is the parent of $\boldsymbol{n_{i+1}}$ for $1 \leq i < k$.

The ***length of a path*** is one less than the number of nodes in the path.

Illustration



Ancestor and Descendant

If there is a path from ***node a*** to ***node b***, then ***a is an ancestor of b*** and ***b is a descendant of a***.

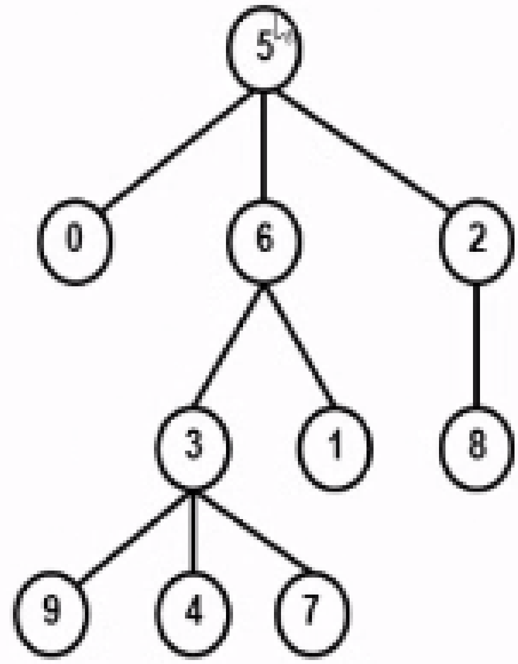
Any node is both an ancestor and a descendant of itself.

Proper Ancestor is an ancestor of a node other than itself.

Proper Descendant is a descendant of a node other than itself.

Note: The root is the only node without a proper ancestor.

Illustration



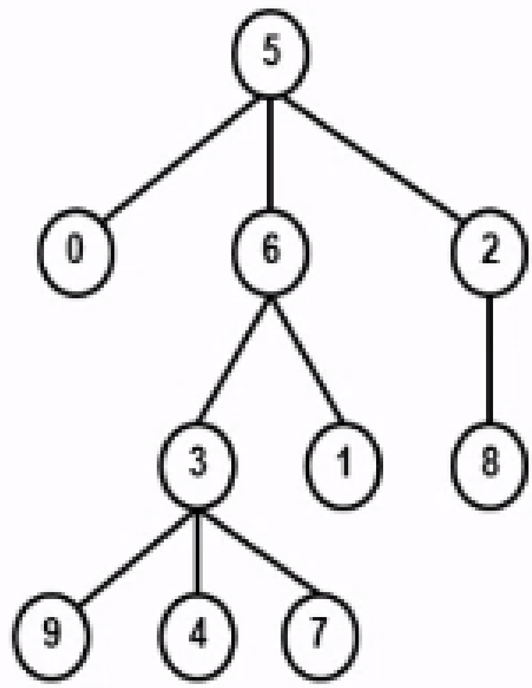
Leaf, Null & Subtree

A **leaf** is a node with no proper descendants.

A **null tree** is a tree with no nodes. It is represented by the symbol ($\wedge$).

A **subtree of a tree** is a node, together with all its descendants.

Illustration

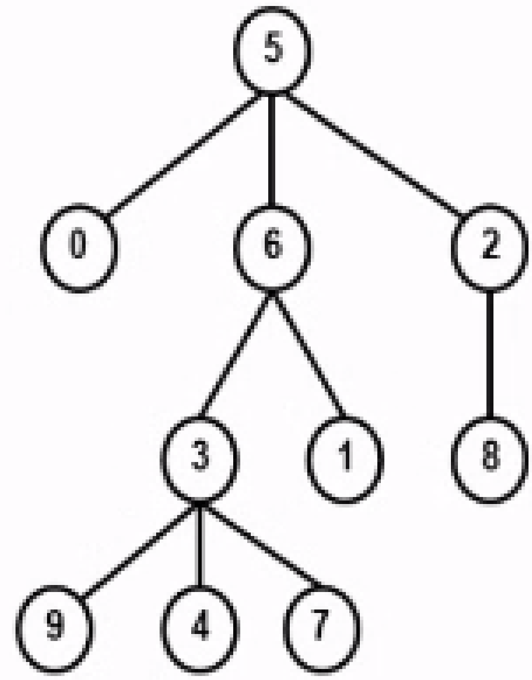


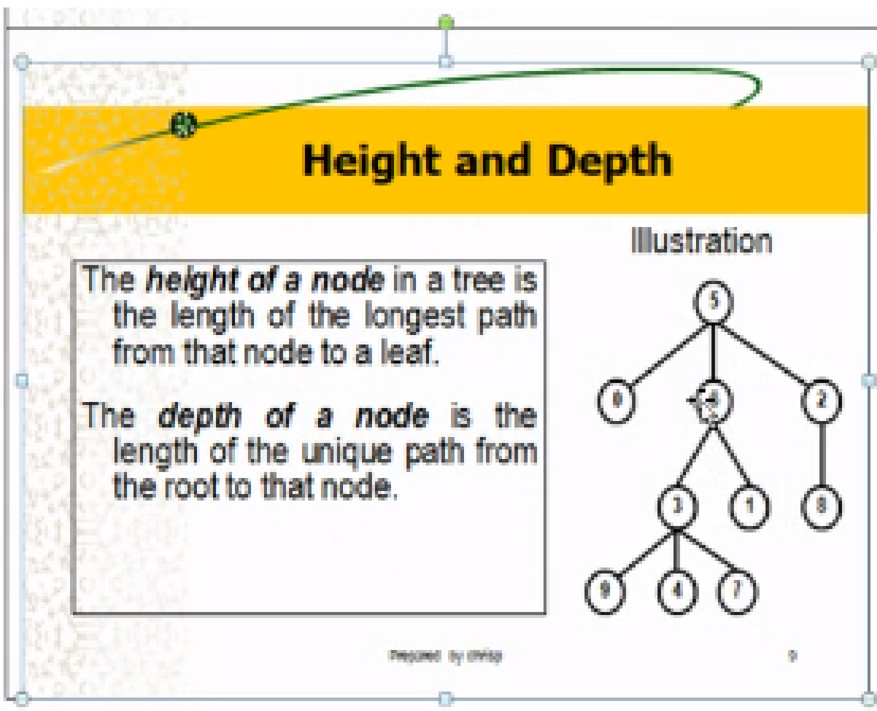
Height and Depth

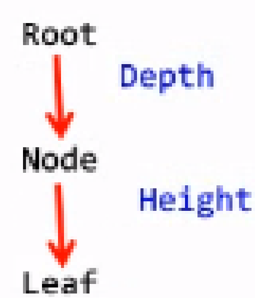
The ***height of a node*** in a tree is the length of the longest path from that node to a leaf.

The ***depth of a node*** is the length of the unique path from the root to that node.

Illustration





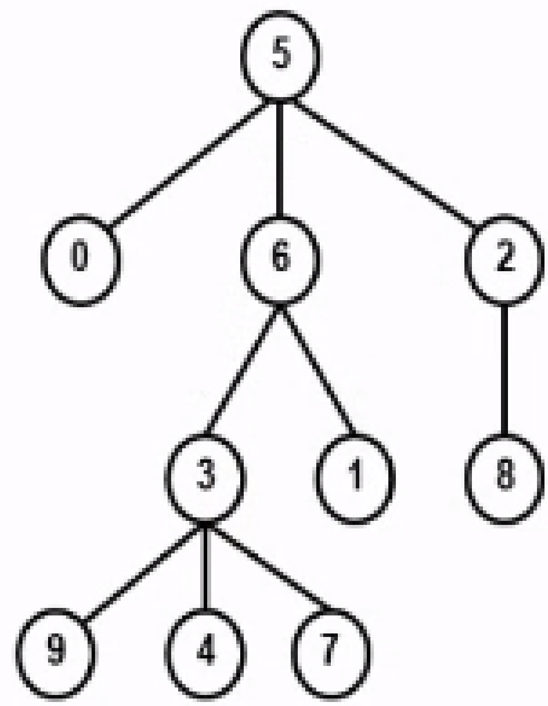


Ordering of Sibling

The *left-to-right* ordering of **siblings** (children of the same node) can be extended to compare any two nodes that are not related by ancestor-descendant relationship.

The relevant rule is that:
if a and b are siblings, and
a is to the left of b, then
all the descendants of a are to the left of all the descendants of b.

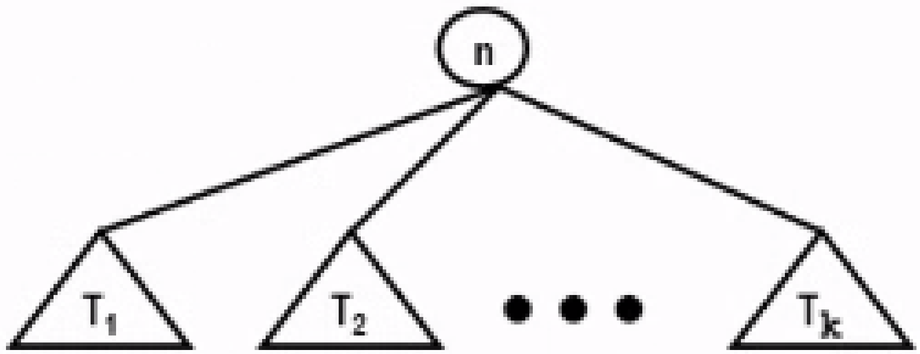
Illustration



Systematic Ordering of Nodes
(Listings or Traversals)

1. Preorder
2. Postorder
3. Inorder

1. Preorder: A Recursive Definition



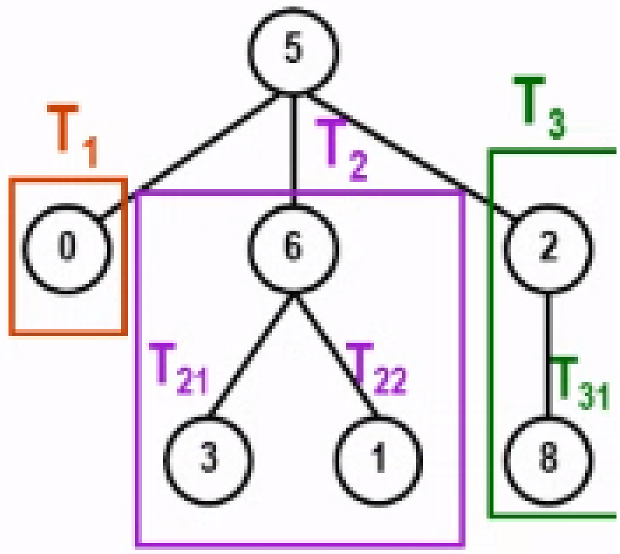
The ***preorder listing*** (or ***preorder traversal***) of the nodes of tree T is:

1. the root n of T
2. followed by the nodes of T_1 in preorder,
3. then the nodes of T_2 in preorder, and
4. so on, up to the nodes of T_k in preorder.

Preorder Listing or Traversal

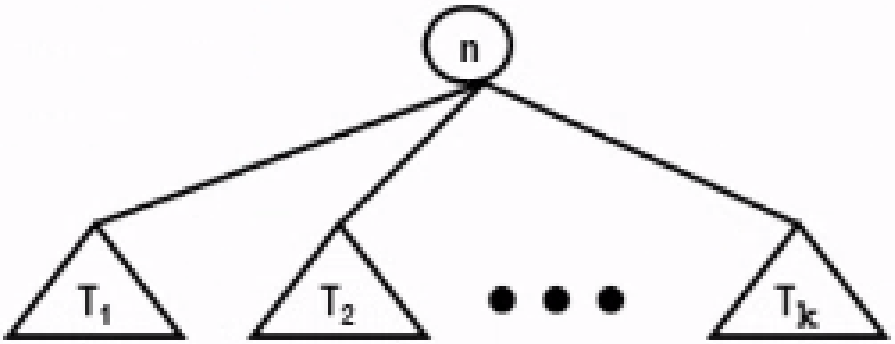
1. the root n of T — **5**
2. followed by the nodes of T_1 in preorder, — **0**
3. then the nodes of T_2 in preorder, and — **6, 3, 1**
4. Up to the nodes of T_3 in preorder. — **2, 8**

Tree T



Preorder Listing : 5, 0, 6, 3, 1, 2, 8

2. Postorder: A Recursive Definition



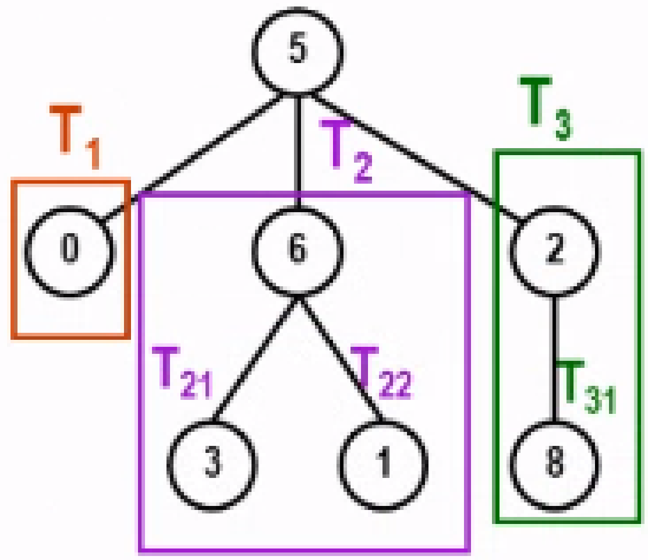
The ***postorder listing*** (or ***postorder traversal***) of the nodes of tree T is:

1. the nodes of T_1 in postorder,
2. then the nodes of T_2 in postorder, and
3. so on, up to the nodes of T_k in postorder,
4. all followed by root n of T

Postorder Listing or Traversal

1. the nodes of T_1 in postorder, **0**
2. then the nodes of T_2 in postorder, and **3, 1, 6**
3. Up to the nodes of T_3 in postorder. **8, 2**
4. All followed by the root n of T **5**

Tree T



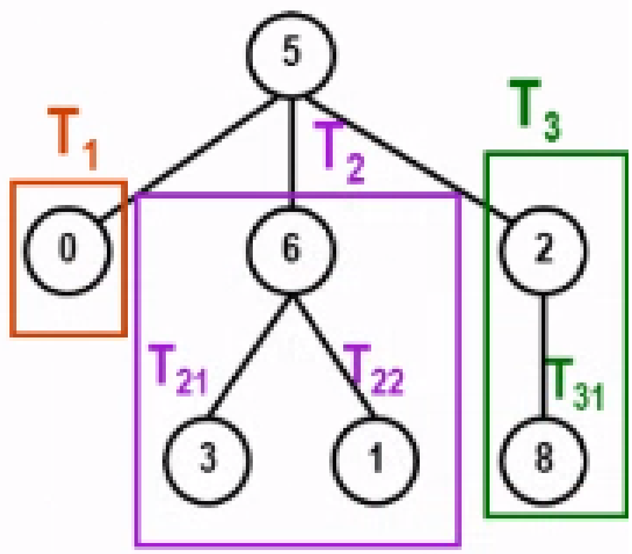
Postorder Listing : **0, 3, 1, 6, 8, 2, 5**

Inorder Listing or Traversal

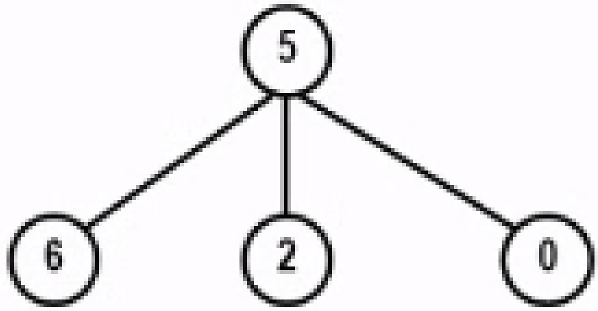
1. the nodes of T_1 in inorder, **0**
2. followed by the root n of T **5**
3. then the nodes of T_2 in inorder, and **3, 6, 1**
4. Up to the nodes of T_3 in preorder. **8, 2**

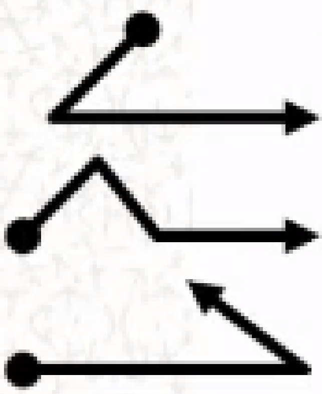
Inorder Listing : **0, 5, 3, 6, 1, 8, 2**

Tree T



Traversal Summary





Preorder Listing : 5, 6, 2, 0

Inorder Listing : 6, 5, 2, 0

Postorder Listing : 6, 2, 0, 5

Labeled and Expression Trees

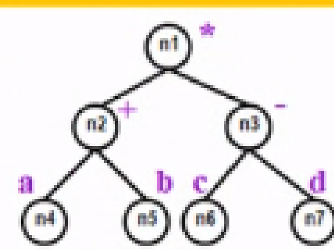
The ***label of a node*** is the value "stored" at the node and not the name of the node.

"An ELEMENT is to a LIST as a LABEL is to a TREE."

- A **Labeled Tree** is a tree whose nodes have labels.
- An **Expression Tree** is a tree where every leaf is labeled by an operand and consists of that operand alone and every interior node is labeled by an operator.

A + B
Leaves: Operands (ex. A and B)
Inode : Operators (ex. +)

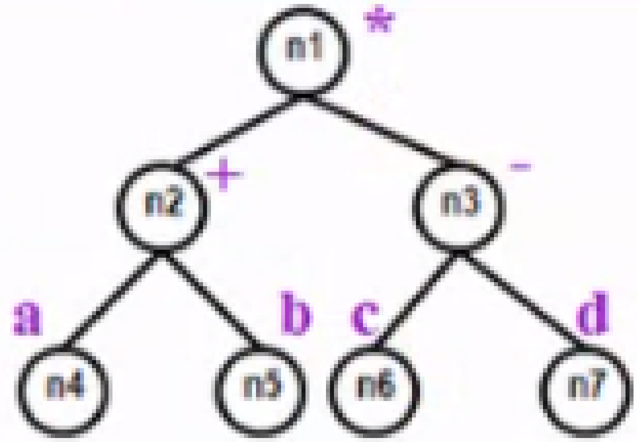
Expression Tree



The above expression tree represents the arithmetic expression (a + b)*(c - d), where n_1, n_2, . . ., n_7 are the names of the nodes, and the operands & operators are th labels.

Forms of writing Expression

1. Prefix Notation – is the listing of labels in preorder
2. Infix Notation – is the listing of labels in inorder
3. Postfix Notation – is the listing of labels in postorder

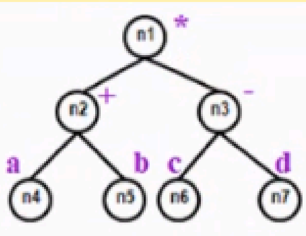


Determine the following:

1. **Prefix Notation**
2. **Infix Notation**
3. **Postfix Notation**

Forms of writing Expression

1. Prefix Notation – is the listing of labels in preorder
2. Infix Notation – is the listing of labels in inorder
3. Postfix Notation – is the listing of labels in postorder



Determine the following:

1. Prefix Notation
2. Infix Notation
3. Postfix Notation

Prepared by chrisp 22

```
Determine the following:
   1) Prefix
   2) Infix
   3) Postfix

//Research:

Polish Notation
Reverse Polish Notation
```

Given the mathematical Expression, draw the expression tree

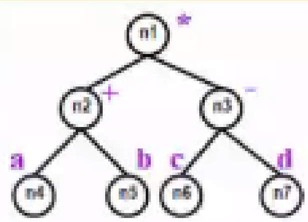
Mathematical Expression #1: (a+b)*(c-d)	Mathematical Expression #2: a + b * c - d

```
      *
    /   \
   +      -
  / \    / \
 a   b  c   d
```

Mathematical Expression #2:
a + b * c - d

Forms of writing Expression

1. Prefix Notation – is the listing of labels in preorder
2. Infix Notation – is the listing of labels in inorder
3. Postfix Notation – is the listing of labels in postorder



Determine the following:

1. **Prefix Notation**
2. **Infix Notation**
3. **Postfix Notation**

Mathematical Expression: (a+b)*(c-d)

Determine the following:

1) Prefix: * + a b - c d
2) Infix : a + b * c - d
3) Postfix: a b + c d - *

//Research:

Polish Notation

Reverse Polish Notation

ADT Tree Operations

1. **PARENT(n, T).** This function returns the parent of node **n** in tree T. If **n** is the root, which has no parent, $\wedge$ is returned.
2. **LEFTMOST_CHILD(n,T).** This returns the leftmost child of node **n** in tree T and returns $\wedge$ if n is a leaf.
3. **RIGHT_SIBLING(n,T).** This returns the right sibling of node **n** in tree T. Right sibling is defined to be the node **r** having the same parent **p** as node **n** and node **r** lies immediately to the right of node **n** in the ordering of the children of node **p**.

ADT Tree Operations

4. **LABEL(n,T).** This returns the label of node **n** in tree T.
5. **CREATEi(v, T_1,T_2,...,T_i).** Makes a new root **r** with label v and gives it **i** children, which are the roots of T_1,T_2,...,T_i , in order from left to right
6. **ROOT(T).** This returns the node that is the root of tree T, or $\wedge$ if T is a null tree
7. **INITIALIZE(T).** This prepares the tree so that it will be used for the first time.
8. **MAKENULL(T).** This makes tree T be an empty tree.

A Recursive Preorder listing function

```
void PREORDER(node n, Tree T )
{  /* List the labels of the descendants of n in preorder */
     node c;
     print(LABEL(n,T));
     c = LEFTMOST_CHILD(n,T);
     while (c <>  ^)
     {
         PREORDER(c, T);
         c = RIGHT_SIBLING(c, T);
     }
 } /* function PREORDER */
```

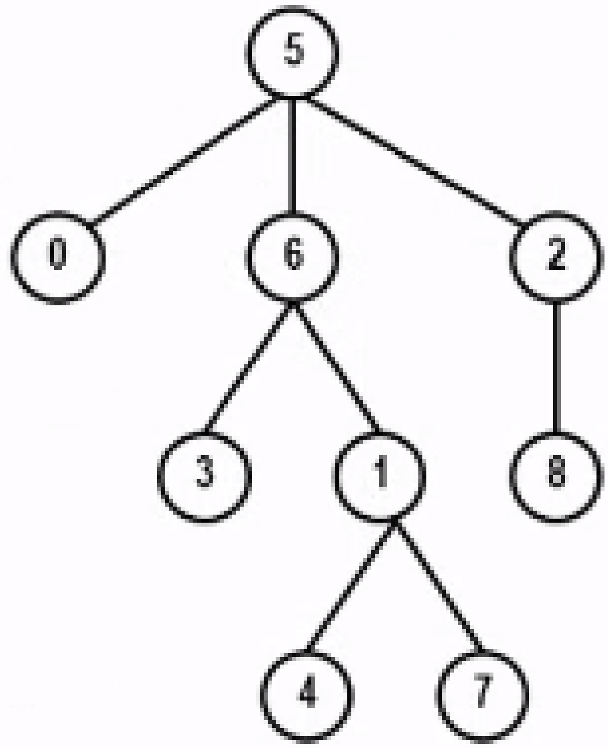

Implementations of Trees

1. **Parent Pointer Representation**
2. **Representation of Trees by LISTS of CHILDREN**

1. Parent Pointer Representation

(Illustration)

	Tree T
0	5
1	6
2	5
3	6
4	1
5	-1
6	5
7	1
8	2
9	-2



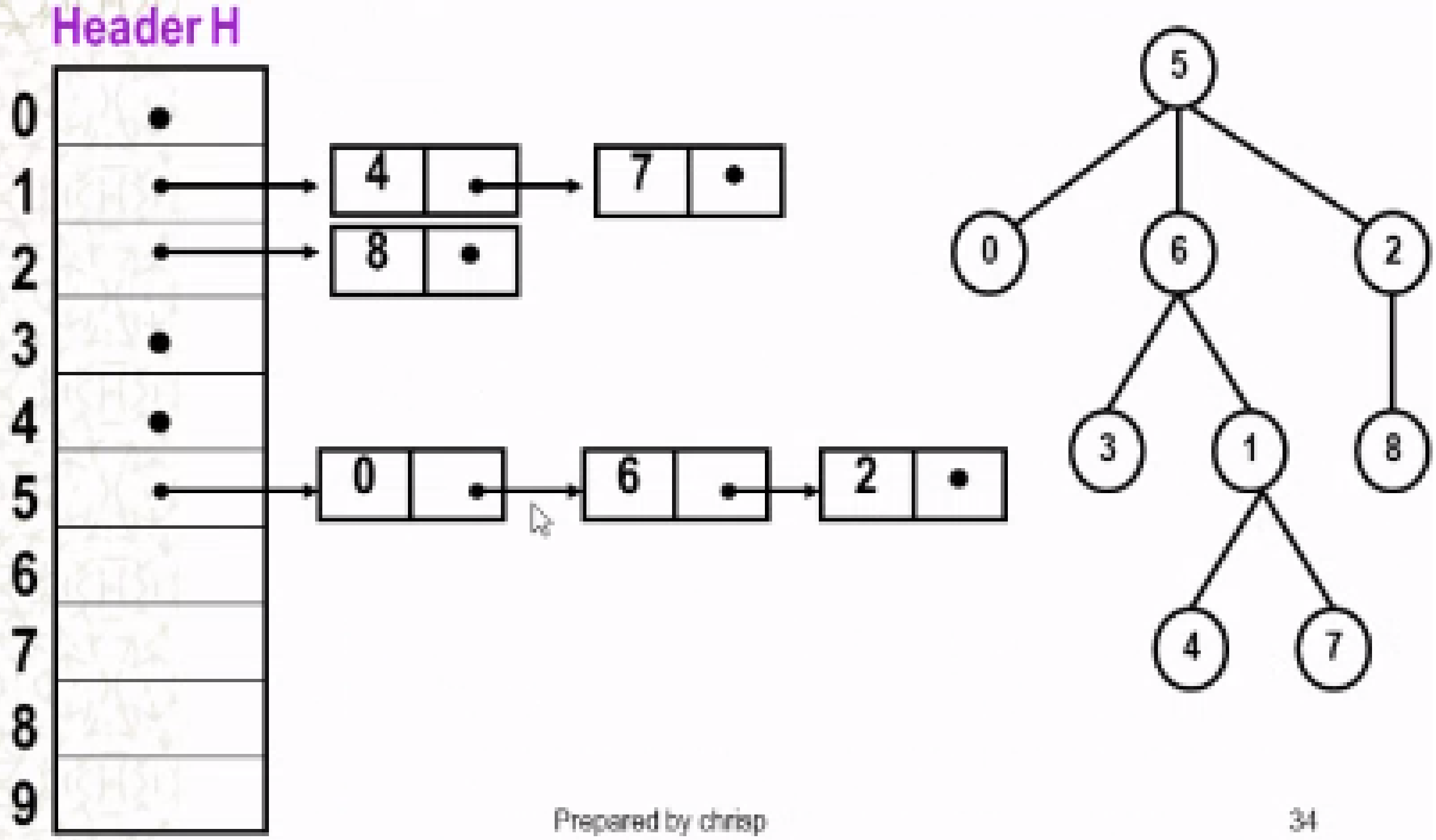
1. Parent Pointer Representation (Description)

- An Array Representation of Trees
- This representation is the simplest representation of tree T that supports the PARENT operation
- T[x] = **y**, if node **y** is the parent of node **x** ;

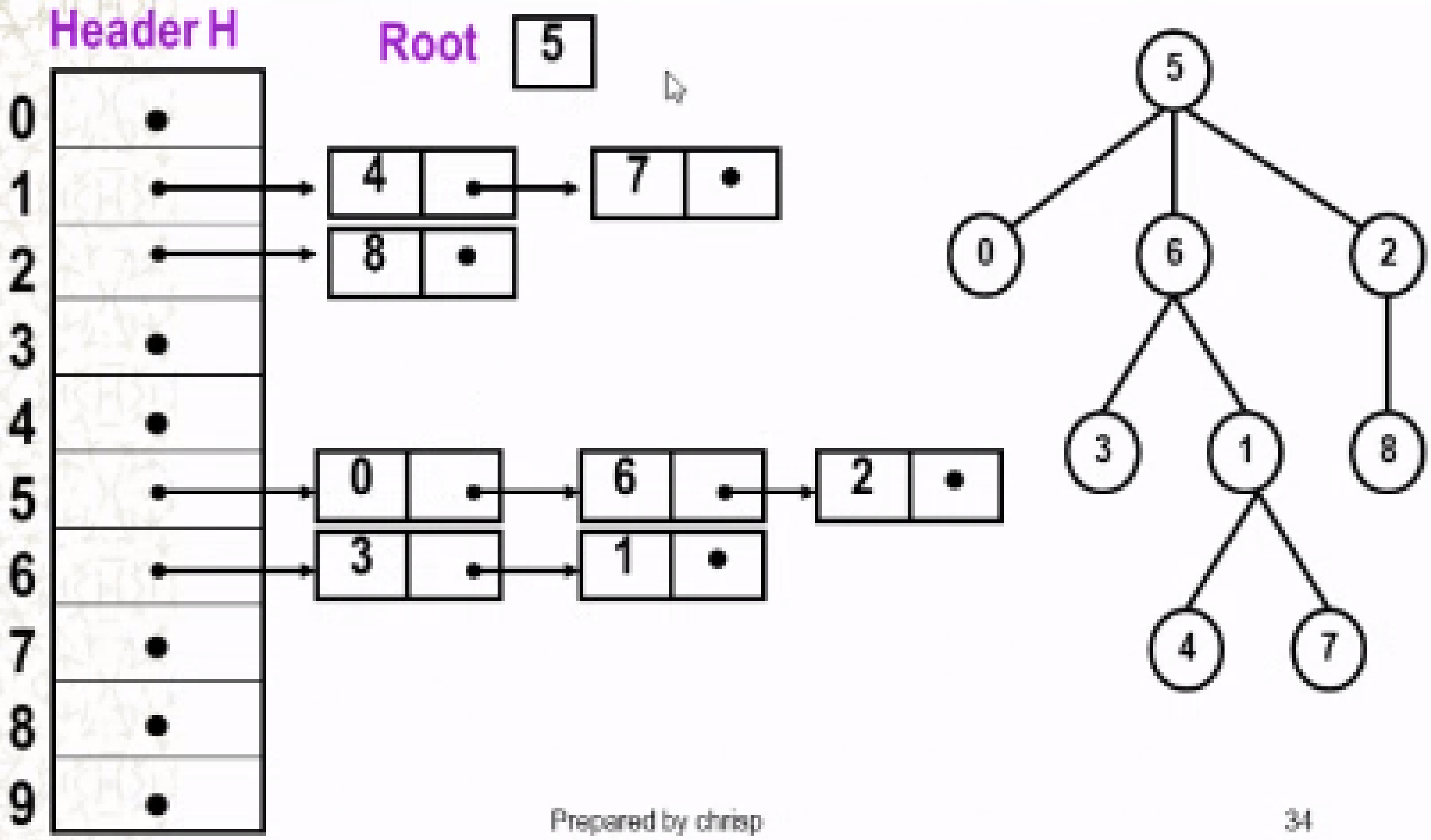
  T[x] = **-1**, if node **x** is the root node; and

  T[x] = **-2**, if node **x** is not a node in the tree

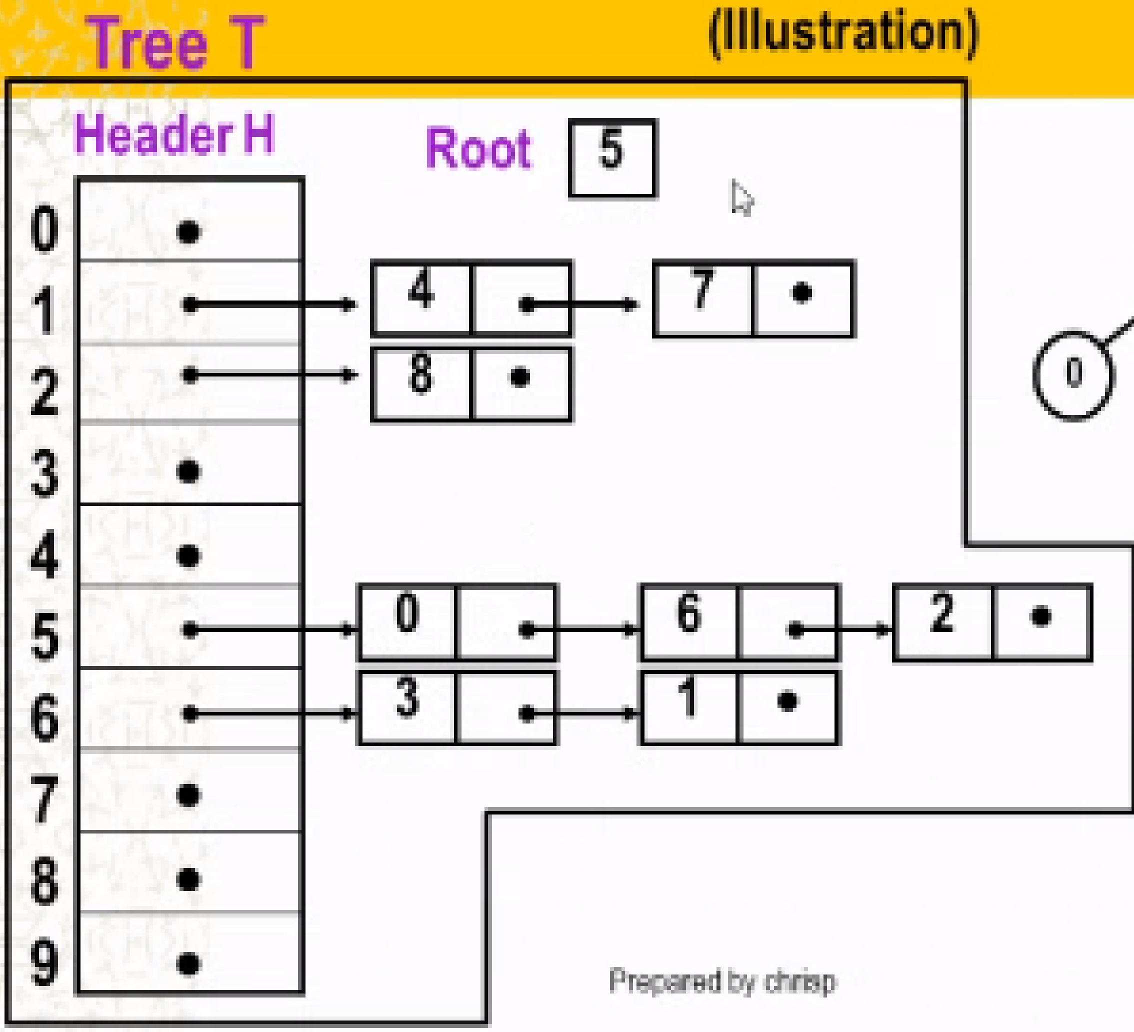
2. Representation of Trees by List of Children
(Illustration)

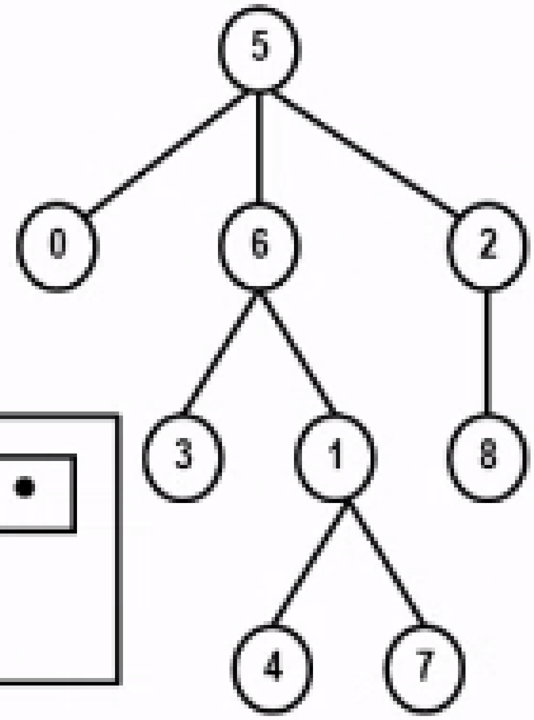


2. Representation of Trees by List of Children
(Illustration)

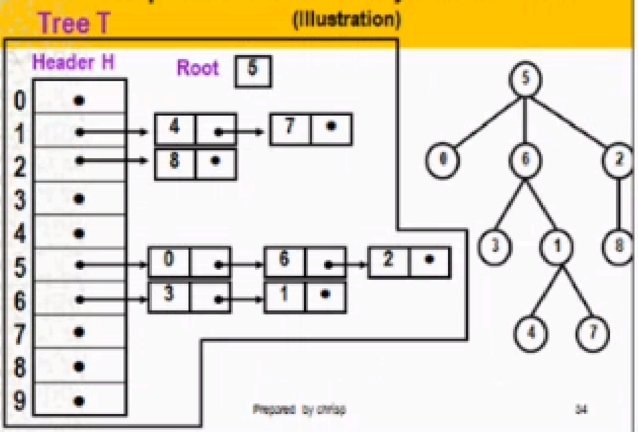


2. Representation of Trees by List of Children
(Illustration)





2. Representation of Trees by List of Children
(Illustration)




```
#define SIZE 10

typedef struct node {
  ___ node;
  ___ link;

}*List;


typedef struct {
   _____ Header[SIZE];
   _____ Root;
}Tree;
```

Exercise 5

1. Write an appropriate definition of the datatypes Tree and node using the representation of trees by List of children. Include macro definition for the size of the array.
2. Write the code for each of the following:
   a) **node LEFTMOST_CHILD(node n, Tree T)**
   b) **node PARENT(node n, Tree T)**