Topic

Binary Trees

(Non-Linear Data Structures)

Linear Data Structures

- Arrays
- Linked lists
- Skip lists
- Self-organizing lists

Non-Linear Data Structures

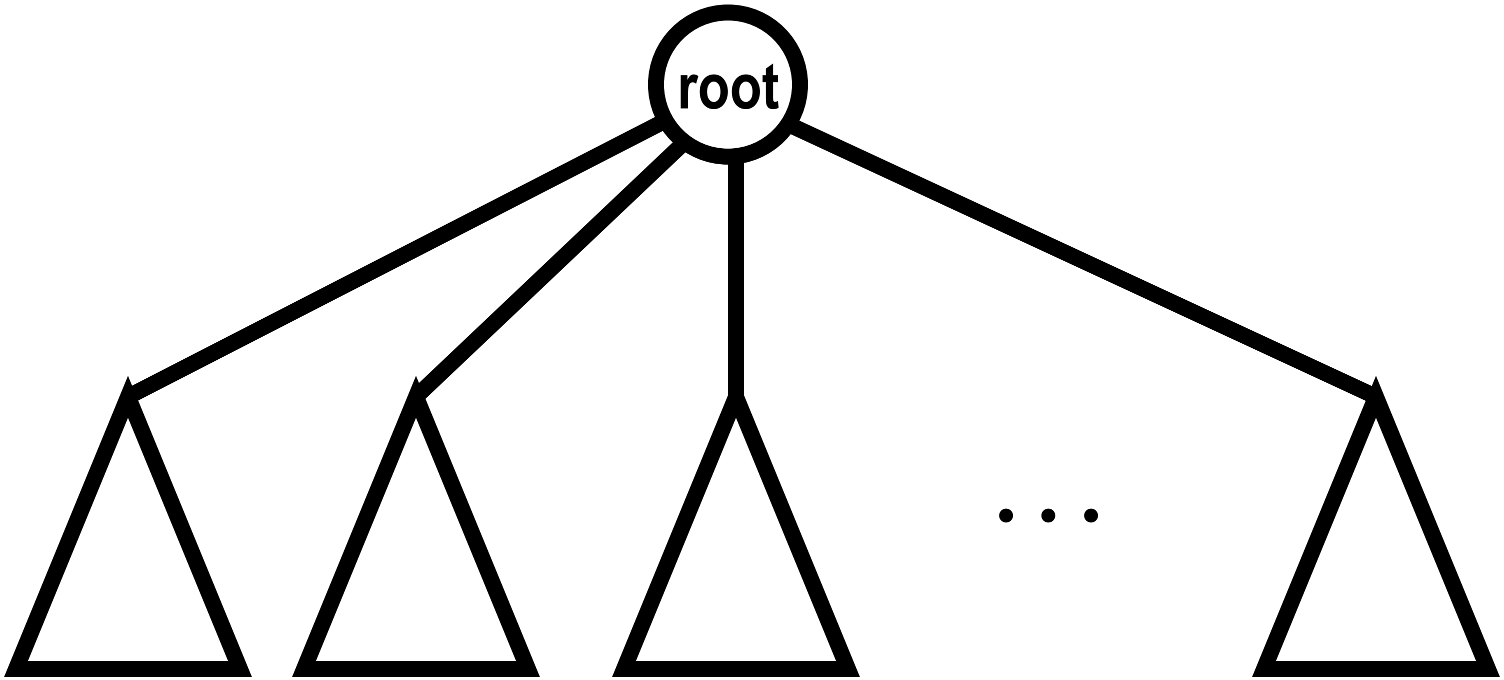
- Hierarchical representation?
  - Trees
  - General Trees
  - Binary Trees
  - Search Trees
  - Balanced Trees
  - Heaps
  - ..

General Trees

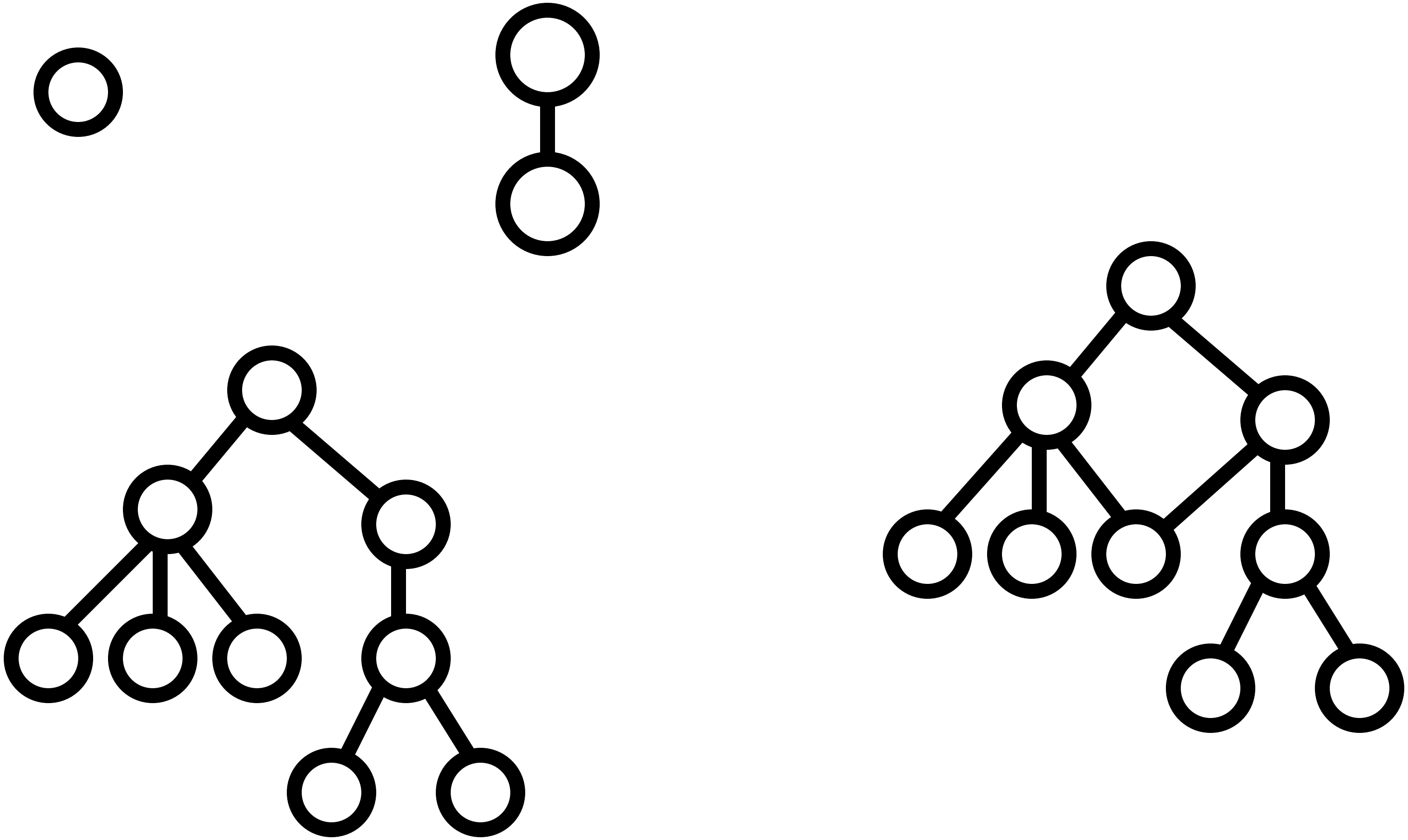
A (Rooted) Tree?

- A finite, nonempty set of nodes and edges s.t.
  - **One special node** (the root of the tree)
  - Each node may be associated (edge) with one or more different nodes (its children).
  - Each node except the root has exactly one parent. The root node has no parent (no incoming edge).
  - **There exists a unique path from the root to any other node!**

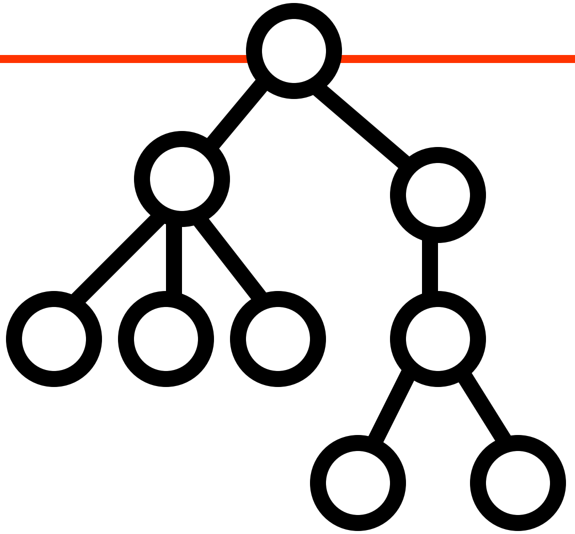
General Rooted Trees



Example: (General Rooted) Trees?



More Terminology

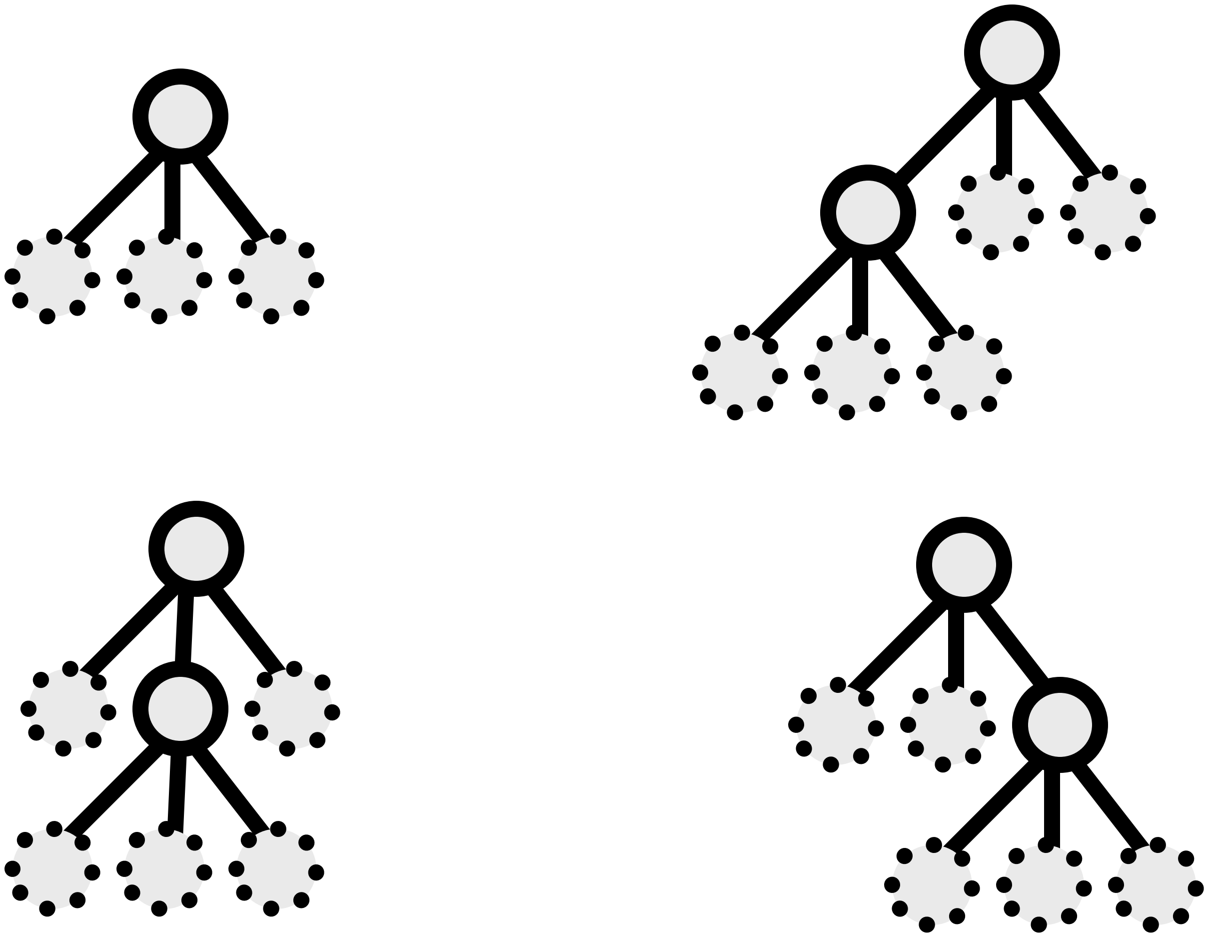


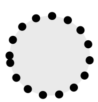
- **Path**
  - ➔A sequence of nodes.
- **Level** of a node
- **Height** of a node
  - ➔The number of nodes in the longest path from that node to a leaf.
- Height of a tree
  - ➔The height of the root node.

An N-ary Tree?

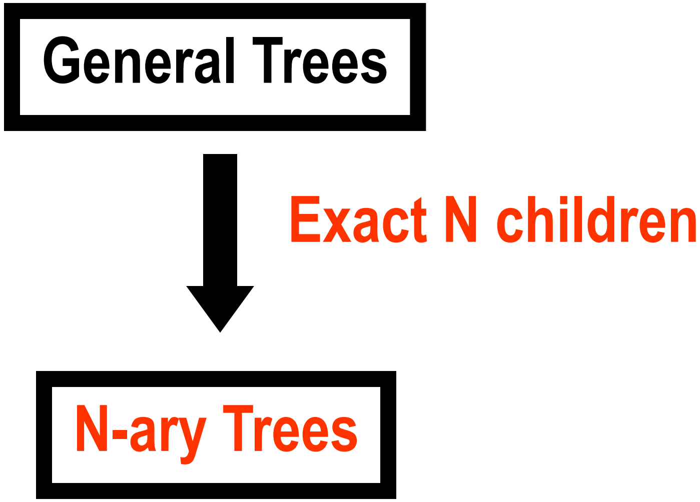
- Each node may be associated (edge) with **exactly N** different nodes (its children).
- If the set is empty (no node), then **an empty N-ary tree**.

Example: N-ary Trees (N=3)



* Note:  = Empty tree!

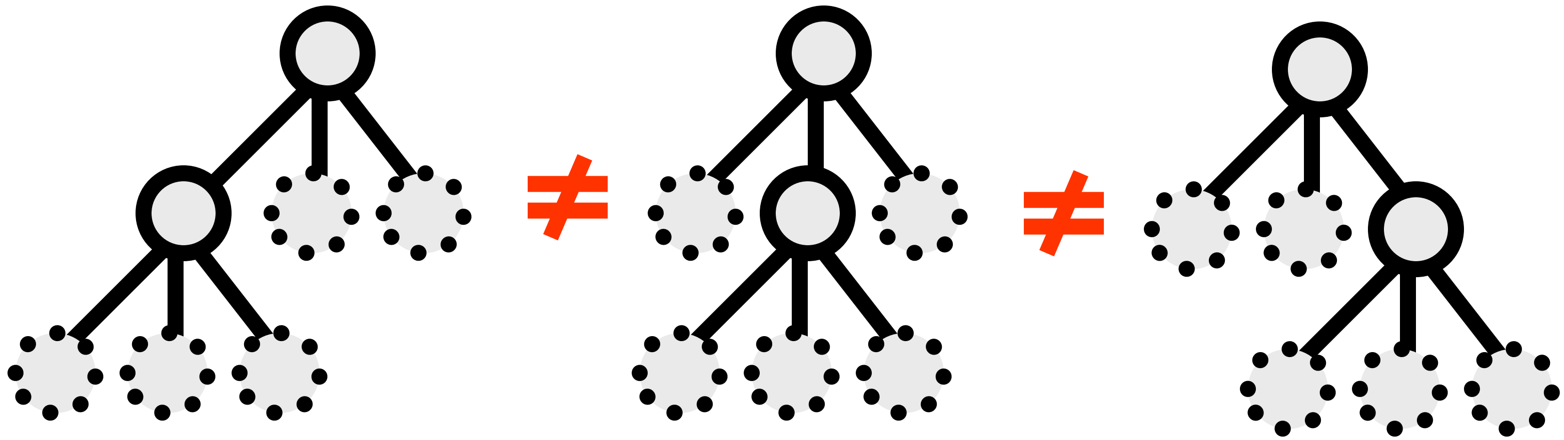
The World of Trees



An Ordered Tree?

- A rooted tree in which the **children** of each node **are ordered**.
  - ➔ first child, second child, third child, etc. …
- Most practical implementations of trees define an implicit ordering of the subtrees.

Example: Ordered Trees



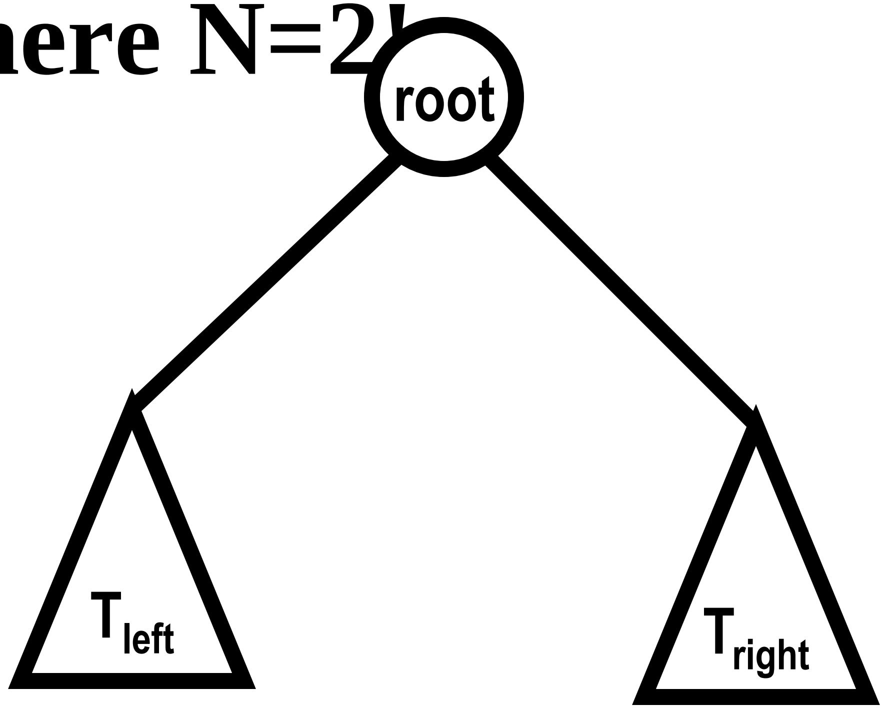
Different Views on Trees

- We may view trees as
  - A mathematical construct.
  - A data structure.
  - An abstract data type.

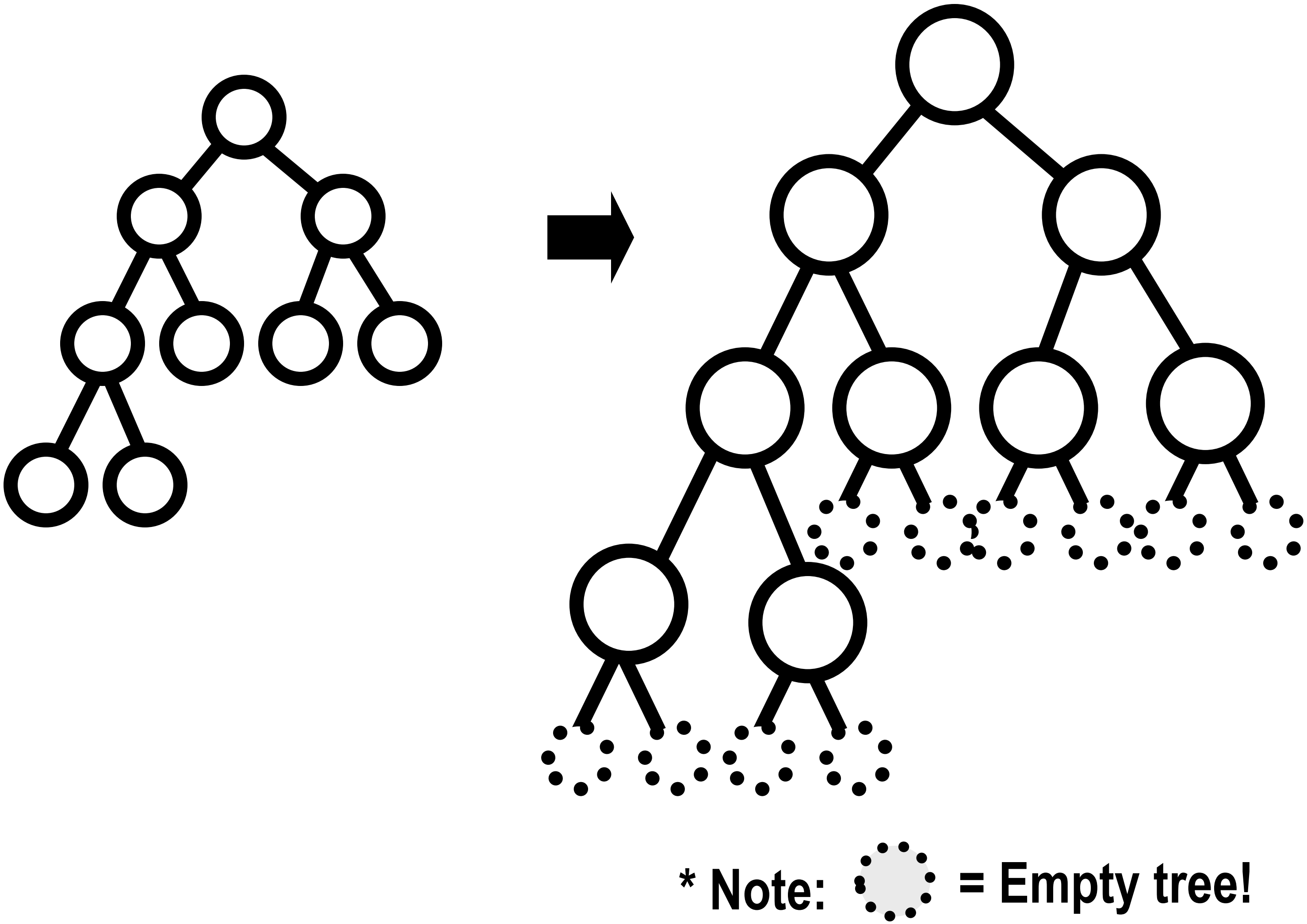
Binary Trees

Binary Trees

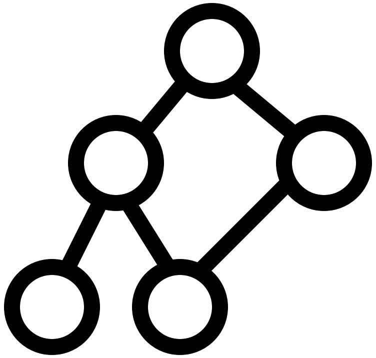
- **A binary tree is an ordered N-ary tree where N=2!**

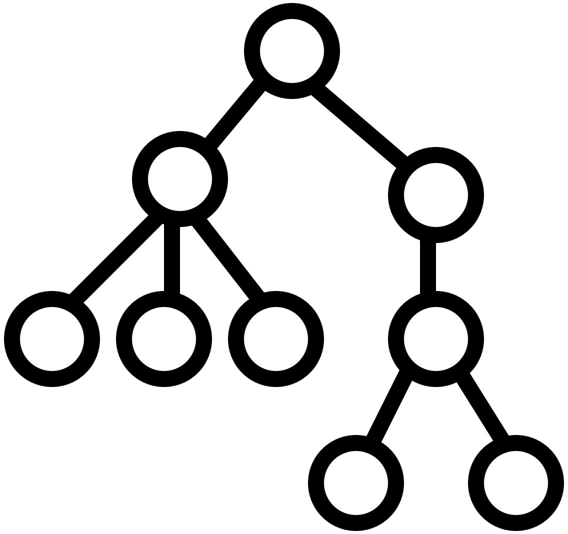


Example: Binary Trees

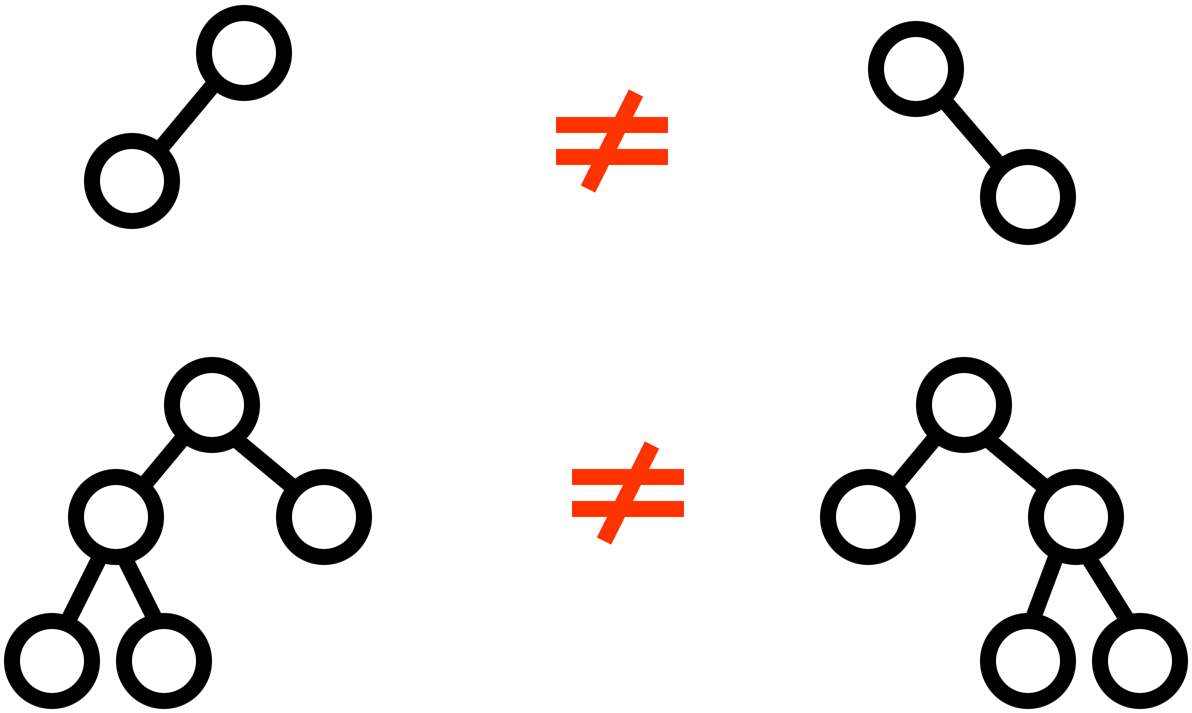


Example: Binary Trees?





Two Different Binary Trees



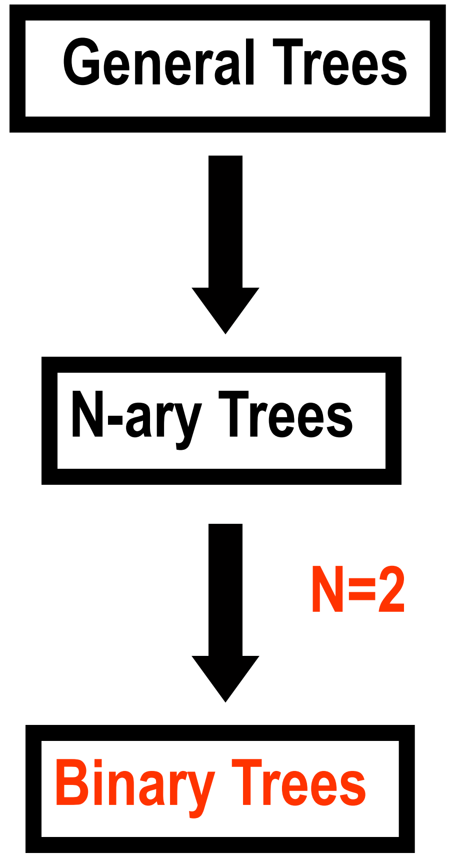
Quiz

- How many different binary trees with 3 nodes?
  - ➔5
- How many different binary trees with 4 nodes?
  - ➔14

Quiz

- What is the range of possible heights of a binary tree with 3 nodes?
  - ➔2 to 3
- What is the range of possible heights of a binary tree with 100 nodes?
  - ➔7 to 100

The World of Trees



Different Shapes of Binary Trees

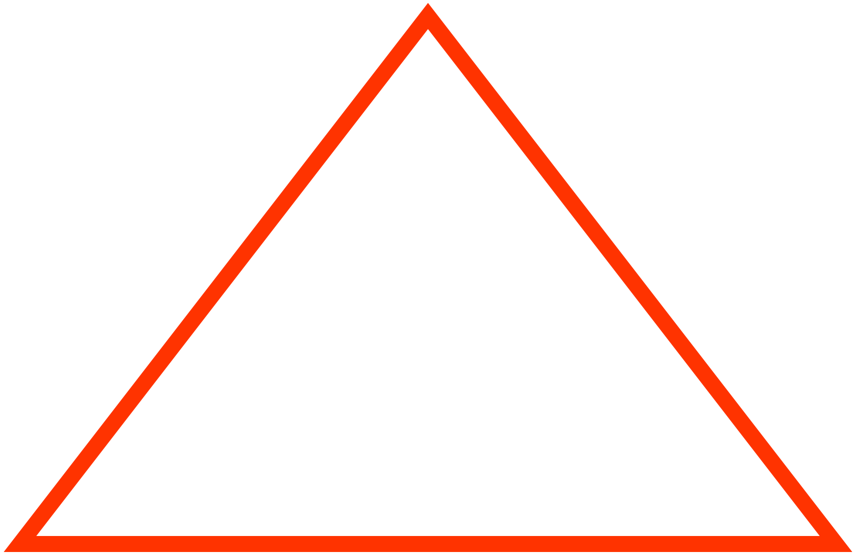
- **Short & Fat** binary trees
  - A full binary tree
  - A complete binary tree
  - A balanced binary tree
- **Tall & Skinny** binary trees
  - A skewed binary tree

A Full Binary Tree

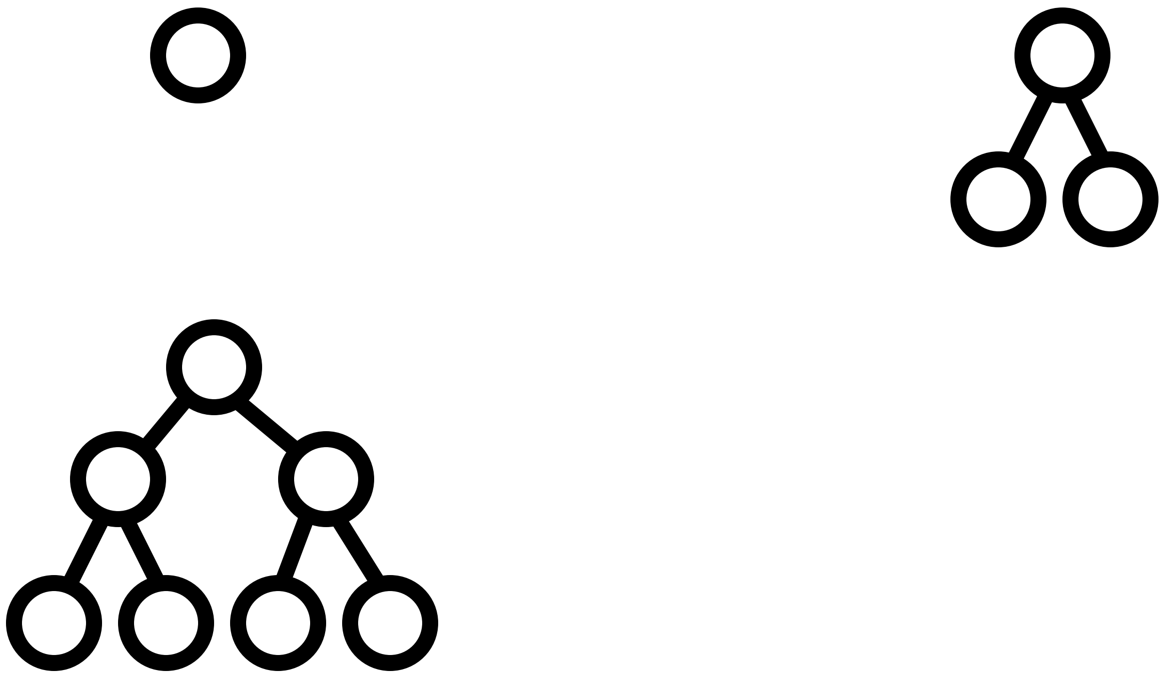
- A binary tree in which
  - ➔All of the leaves are on the same level. (at level h for the binary tree of height h.)
  - ➔Every nonleaf node has exactly two children.

Shape of A Full Binary Tree

- The basic shape of a full binary tree is triangular!



Example: Full Binary Trees

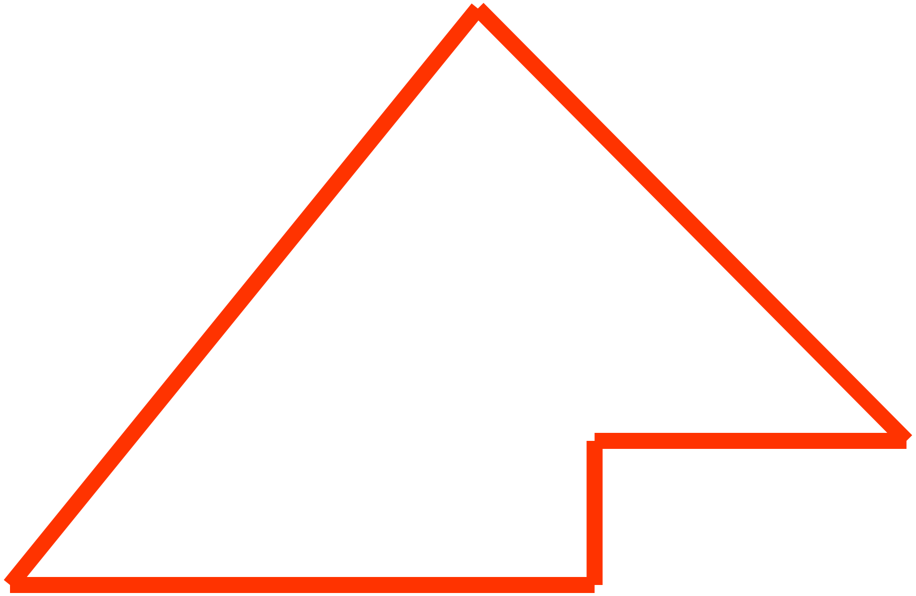


A Complete Binary Tree

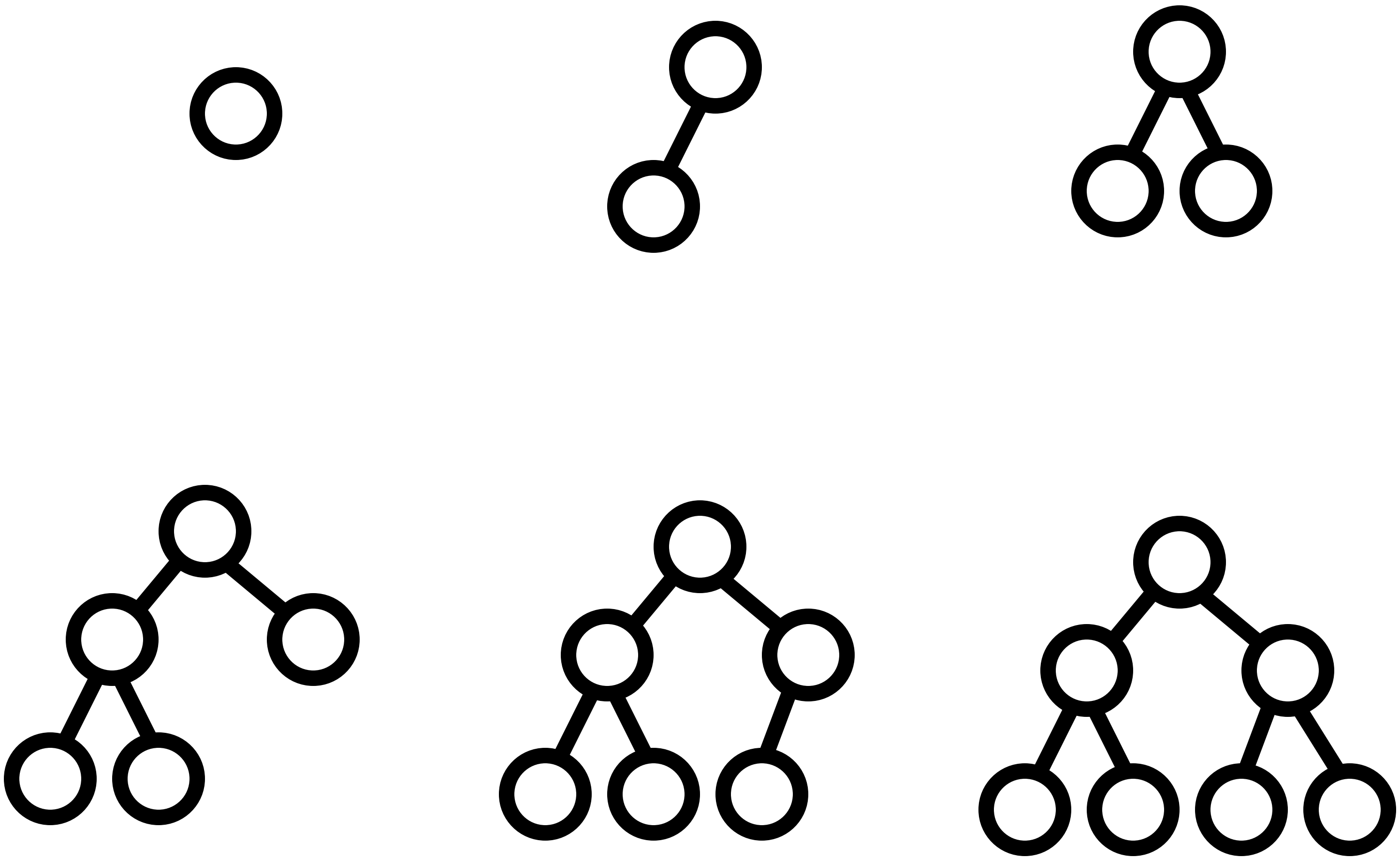
- A binary that is
  - Either full binary tree
  - Or full through the next-to-last level, with the leaves on the last level as far to the left as possible (filled in from left to right).
- A binary tree in which all leaf nodes are at level n or n-1, and all leaves at level n are towards the left.
- A binary tree of height h that is full to level h-1 and has level h filled from left to right.

Shape of A Complete Binary Tree

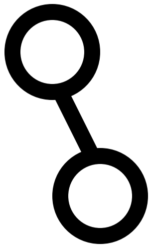
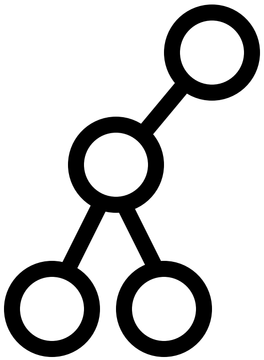
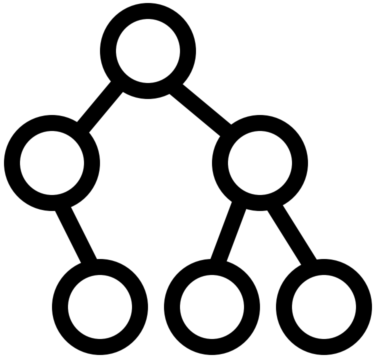
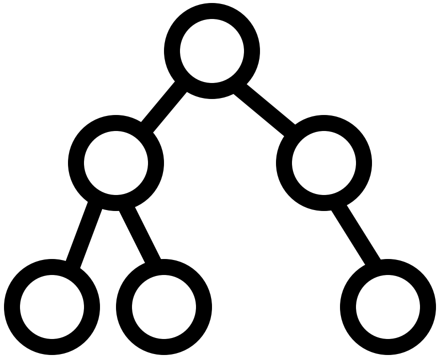
- The basic shape of a complete binary tree is like



Example: Complete Binary Trees?



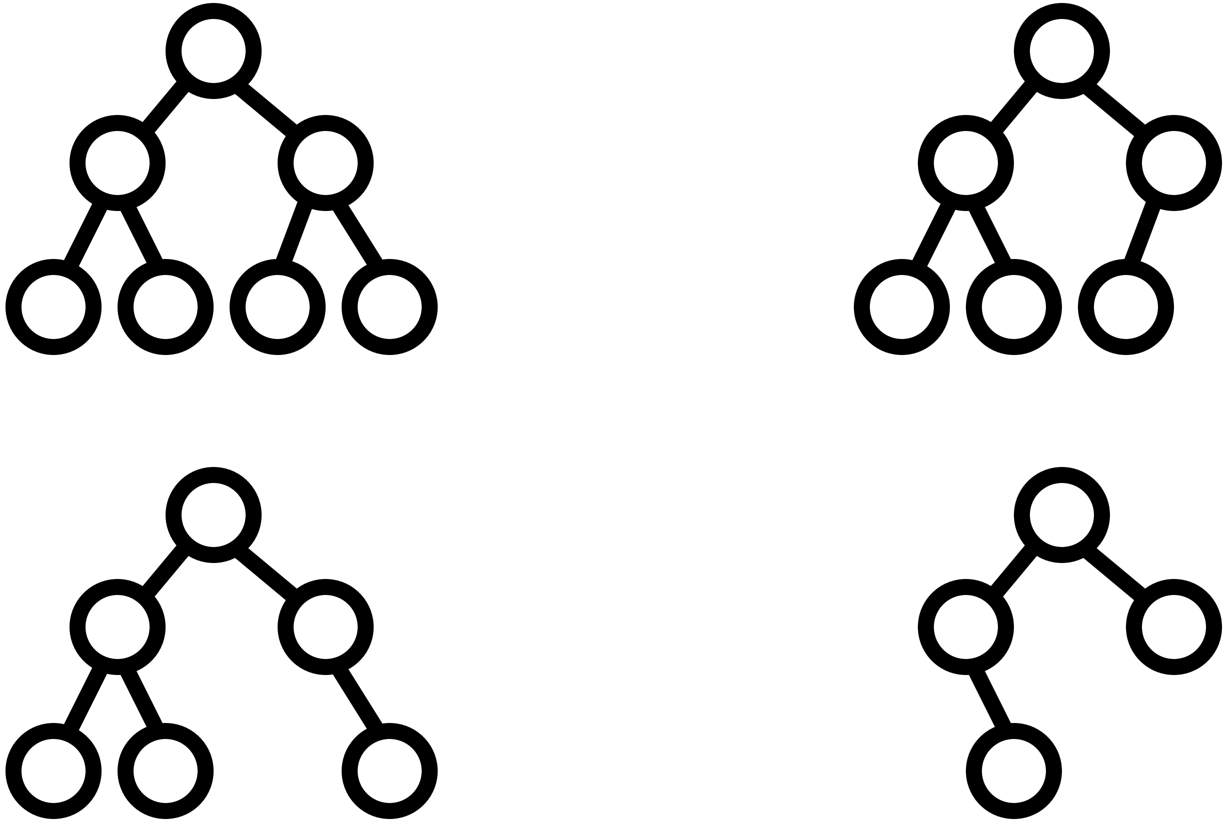
Example: Complete Binary Trees?

A (Height) Balanced Binary Tree

- A binary tree in which the left and right subtrees of any node have heights that differ **by at most 1**.

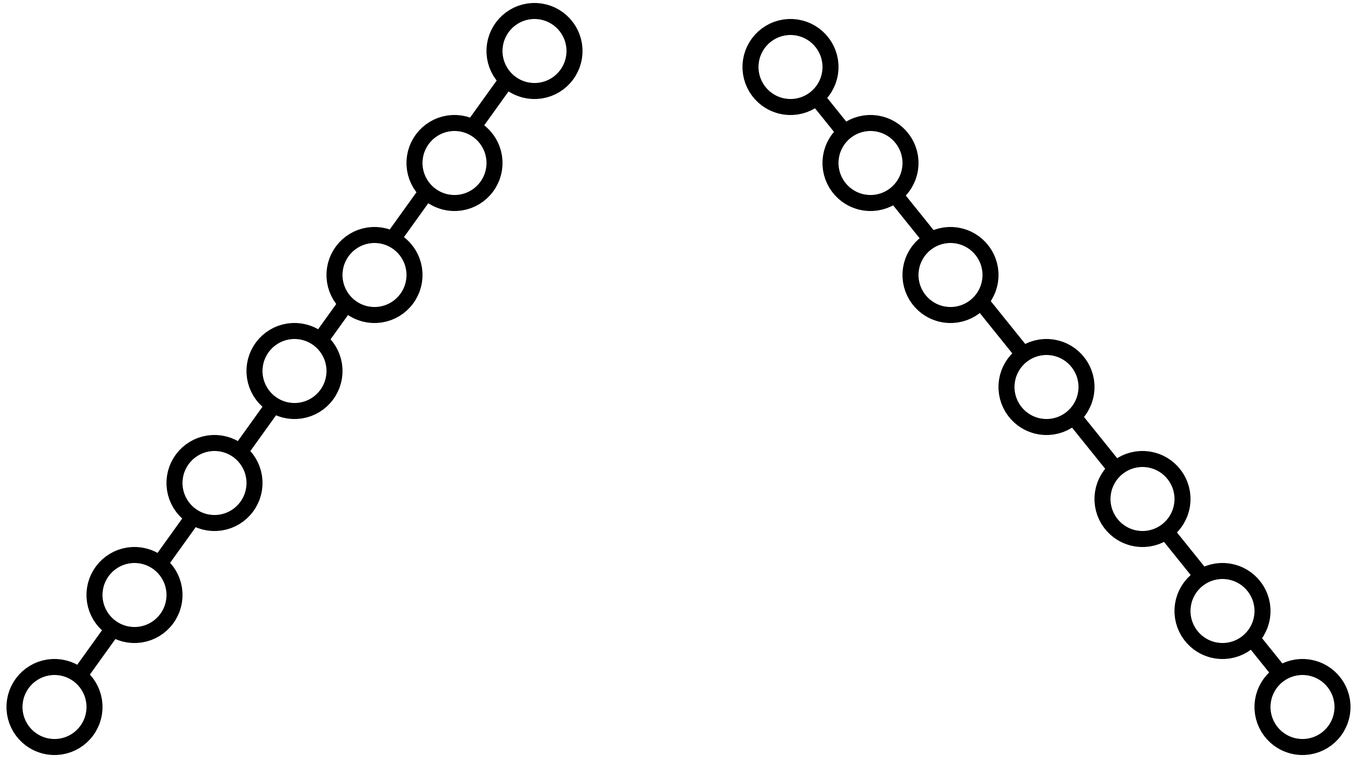
Example: (Height) Balanced Binary Trees



A Skewed Binary Tree

- A degenerate binary tree
- A skewed binary tree is expensive to process!

Example: Skewed Binary Trees



Quiz

- Is a full binary tree complete?
  - ➔ Yes!
- Is a complete binary tree full?
  - ➔ No!
- Is a full binary tree balanced?
  - ➔ Yes!
- Is a complete binary tree balanced?
  - ➔ Yes!

Properties of Binary Trees

- What is the maximum height of a binary tree with n nodes?
  - ➔ n

The maximum height of a binary tree with n nodes is n.

Properties of Binary Trees

- What is the minimum height of a binary tree with n nodes?
  - ➔ $\lceil \log(n+1) \rceil$ where the ceiling of log (n+1) = log (n+1) rounded up.

The minimum height of a binary tree with n nodes is $\lceil \log(n+1) \rceil$.
(The ceiling of log (n+1) = log (n+1) rounded up.)

Properties of Binary Trees

The height of a binary tree with n nodes is at least $\lceil \log(n+1) \rceil$ and at most n.

Properties of Binary Trees

- What is the minimum number of nodes that a binary tree of height h can have?
  - ➔h

The minimum number of nodes that a binary tree of height h can have is h.

Properties of Binary Trees

- What is the maximum number of nodes that a full binary tree of height h can have?
  - ➔ $2^h - 1$

The maximum number of nodes that a binary tree of height h can have is 2^h - 1.

Properties of Binary Trees

- **Full binary trees and complete binary trees have the minimum height!**
- **Skewed (degenerate) binary trees have the maxim m height!**

Representation of Binary Trees

How to Represent a Binary Tree?

- An array-based representation
- An array-based representation **for complete binary trees**
- A pointer-based representation

1. An Array-Based Representation

- Represent a node in the binary tree as a structure
  - A data
  - Two indexes (One for each child)
- Represent the binary tree as an array of structures.

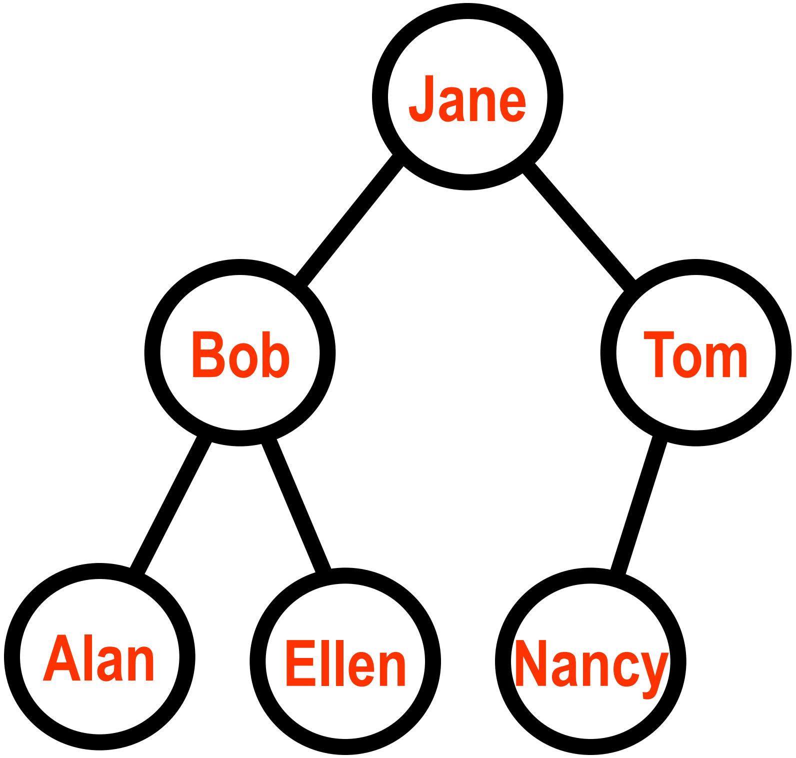
An Array-Based Representation

```
const int MAX_NODES = 100;

struct BTnode
{    typedef string DataType;
     DataType Data;
     int Lchild;
     int Rchild;
};

BTnode BT[MAX_NODES];
int Root;
int Free;
```

An Array-Based Representation



BT	Data	Lchild	Rchild
0	Jane	1	2
1	Bob	3	4
2	Tom	5	-1
3	Alan	-1	-1
4	Ellen	-1	-1
5	Nancy	-1	-1
6			
.			
.			
.			

Root

0

Free

6

2. An Array-Based Representation of a Complete Binary Tree

- A **better** array-based representation for a complete binary tree!
- Represent a node in the binary tree as
  - A data
- Represent the binary tree as an array.

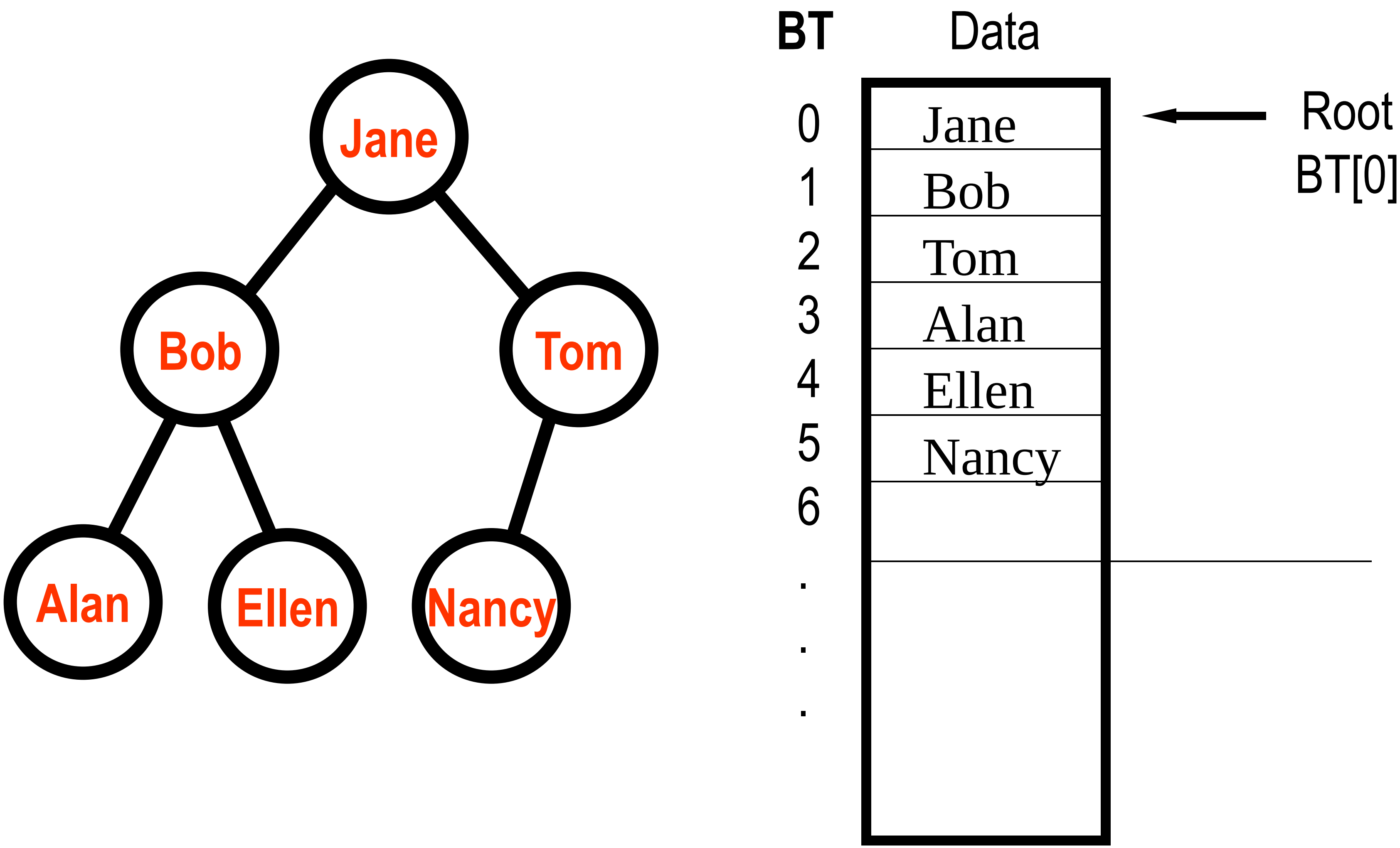
An Array-Based Representation of a Complete Binary Tree

```
const int MAX_NODES = 100;

typedef string DataType;

DataType BT[MAX_NODES];
int Root;
```

An Array-Based Representation of a Complete Binary Tree



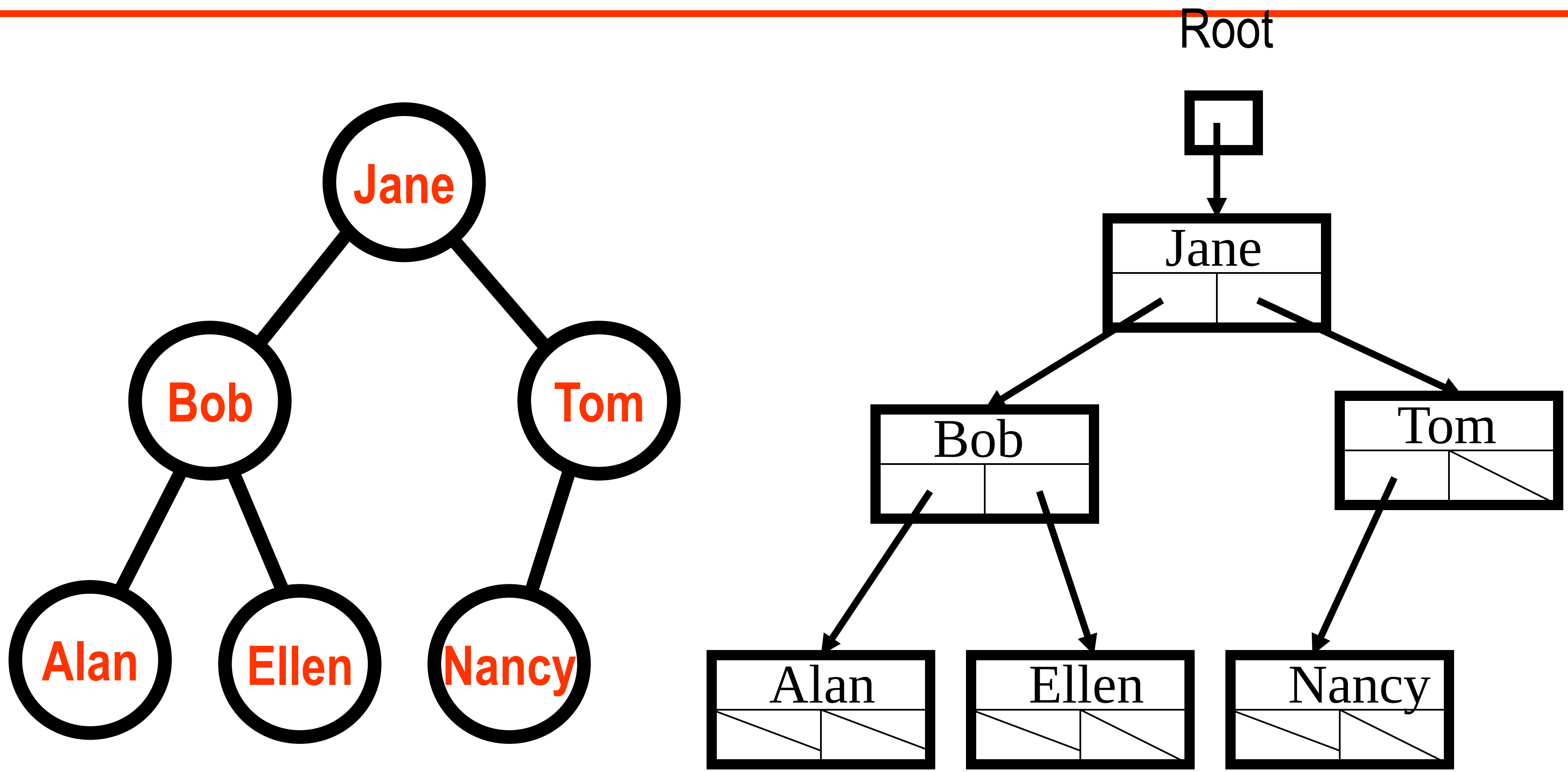
An Array-Based Representation of a Complete Binary Tree

- Any node BT[i]
  - Its left child =
    - BT [2 * i + 1]
  - Its left child =
    - BT [2 * i + 2]
  - Its parent =
    - BT [(i -1) / 2]
- Must maintain it as a complete binary tree!
- Limited **delete**!

3. A Pointer-Based Representation

- Represent a node in the binary tree as a structure
  - A data
  - Two pointers (One for each child)
- Represent the binary tree as a linked structure.

A Pointer-Based Representation



A Pointer-Based Representation

```
struct BTnode
{    typedef string DataType;
     DataType Data;
     BTnode* LchildPtr;
     BTnode* RchildPtr;
};

BTnode* rootBT;
```

A Pointer-Based Representation using Template

```
template<class DataType>
struct BTnode
{
     DataType Data;
     BTnode<DataType>* LchildPtr;
     BTnode<DataType>* RchildPtr;
};


BTnode<DataType>* rootBT;
```

A Pointer-Based Representation using Template Class

```
template<class DataType>
class BTnode
{
Public:
    BTnode();
    BTnode(DataType D, BTnode<DataType>* l,
             BTnode<DataType>* r)
             :data(D), LchildPtr(l),
RchildPtr(r) { }
    friend class BT<DataType>;
private:
    DataType data;
    BTnode<DataType>* LchildPtr;
    BTnode<DataType>* RchildPtr;
};
```

A Pointer-Based Representation using Template Class

```
template<class DataType>
class BT
{
Public:
    BT();
    …
private:
BTnode<DataType>* rootBT;
…
};
```

Binary Trees as ADTs

Operations on Binary Trees

- Create an empty binary tree.
- Create a one-node binary tree, given an item.
- Create a binary tree, given an item for its root and two binary trees for the root’s left and right subtrees.
- Attach a left or right child to the binary tree’s root.
- Attach a left or right subtree to the binary tree’s root.
- Detach a left or right subtree to the binary tree’s root.
- Destroy a binary tree.

More Operations on Binary Trees

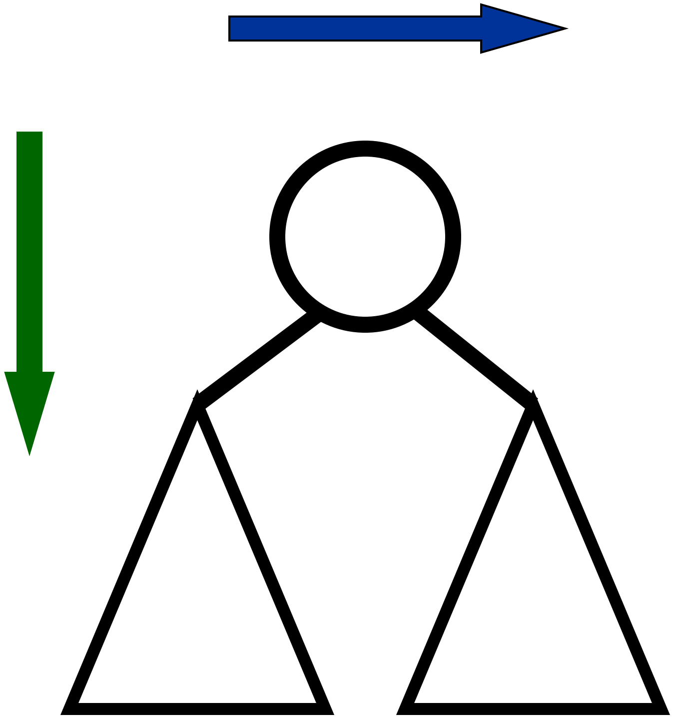
- Determine whether a binary tree empty?
- Determine or change the data in the binary tree’s root.
- Return a copy of the left or right subtree of the binary tree’s root.
- Traverse the nodes in a binary tree in preorder, inorder or postorder.
- ...

Traversal Operations on Binary Trees

- It is frequently necessary to examine every node exactly once in a binary tree.
- Binary tree traversal is the process of
  - ➔Visiting systematically all the nodes of a binary tree and
  - ➔Performing a task (calling a visit procedure like print).

Traversal Operations on Binary Trees

- Two essential approaches:
  - Depth-first traversal
  - Breadth-first traversal



Possible Depth-First Traversals

- Six possible ways to arrange those tasks:
  - 1. Process a node, then left-child subtree, then right-child subtree.
  - 2. Process left-child subtree, then a node, then right-child subtree.
  - 3. Process left-child subtree, then right-child subtree, then a node.
  - 4. Process a node, then right -child subtree, then left-child subtree.
  - 5. Process right -child subtree, then a node, then left-child subtree.
  - 6. Process right -child subtree, then left-child subtree, then a node.
- In almost all cases, the subtrees are analyzed left to right!

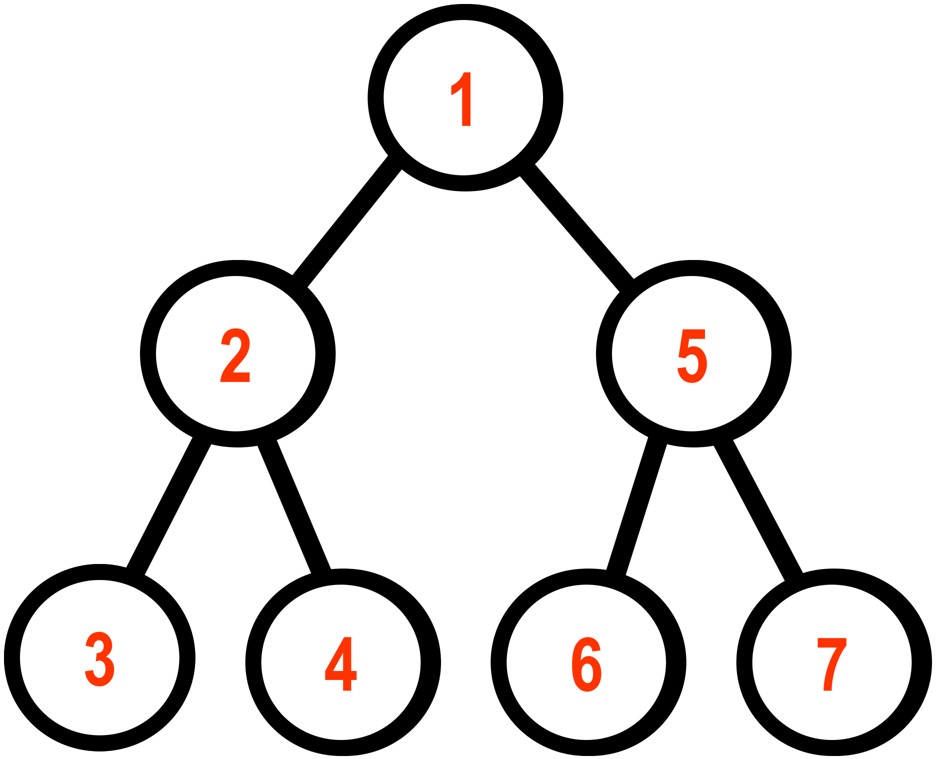
Common Binary Tree Traversals

- Three ways to arrange those tasks:
  - 1. Process a node, then its left-child subtree, then its right-child subtree.
    - **Preorder Traversal**
  - 2. Process its left-child subtree, then a node, then its right-child subtree.
    - **Inorder Traversal**
  - 3. Process its left-child subtree, then its right-child subtree, then a node.
    - **Postorder Traversal**

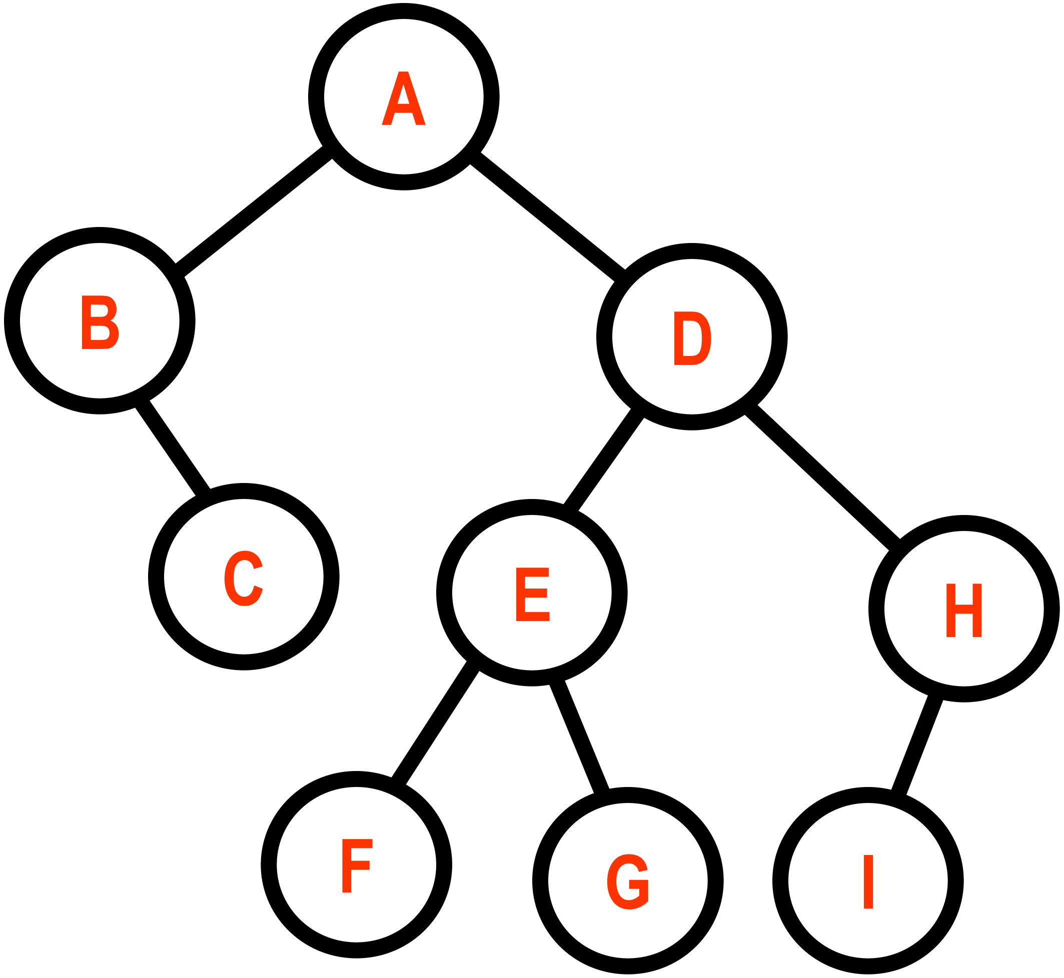
1. Pre-order Traversal

- If the tree is not empty then
  - Visit the root
  - Preorder traverse the left subtree recursively
  - Preorder traverse the right subtree recursively

Pre-order Traversal - Processing Order



Example: Preorder traversal



A B C D E F G H I

Preorder Traversal & Print

```
void preorder_print(BTnode* rootBT)
{
    if (rootBT != NULL)
    {
        cout << rootBT->Data << endl;
        preorder_print(rootBT-> LchildPtr);
        preorder_print(rootBT-> RchildPtr);
    }
}
```

Preorder Traversal and Operation

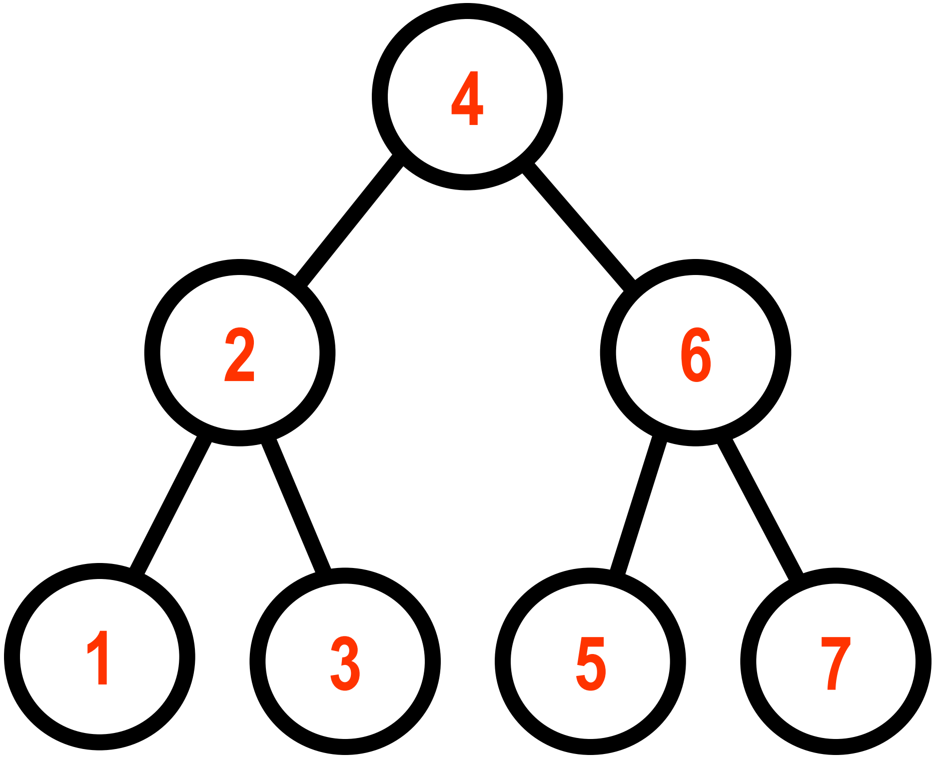
```
typedef void (*fType)(DataType& AnItem);

void preorder(BTnode* rootBT, fType Op)
{
    if (rootBT != NULL)
    {
        Op(rootBT->Data);
        preorder(rootBT-> LchildPtr);
        preorder(rootBT-> RchildPtr);
    }
}
```

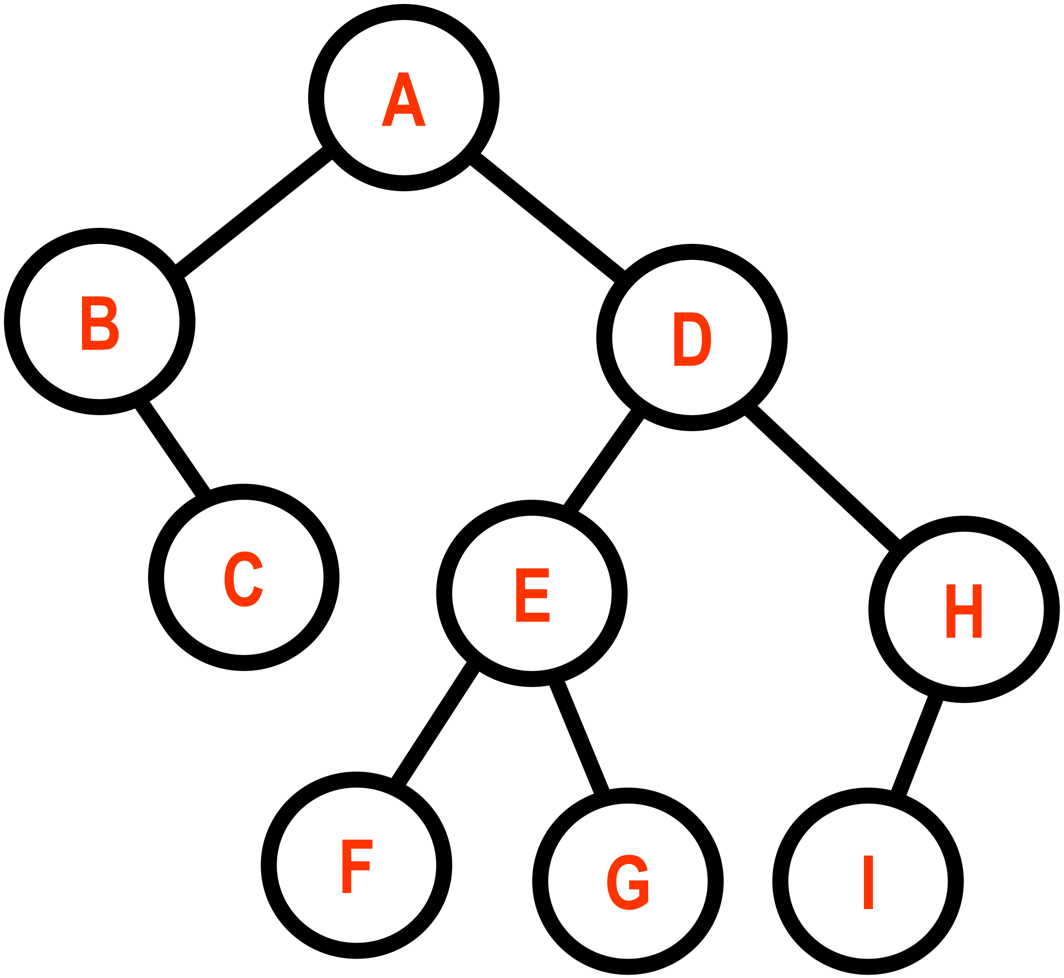
2. In-order Traversal

- If the tree is not empty then
  - ➔Inorder traverse the left subtree recursively
  - ➔Visit the root
  - ➔Inorder traverse the right subtree recursively

In-order Traversal - Processing Order



Example: Inorder traversal



B C A F E G D I H

Inorder Traversal & Print

```
void inorder_print(BTnode* rootBT)
{
    if (rootBT != NULL)
    {
        inorder_print(rootBT-> LchildPtr);
        cout << rootBT->Data << endl;
        inorder_print(rootBT-> RchildPtr);
    }
}
```

Inorder Traversal and Operation

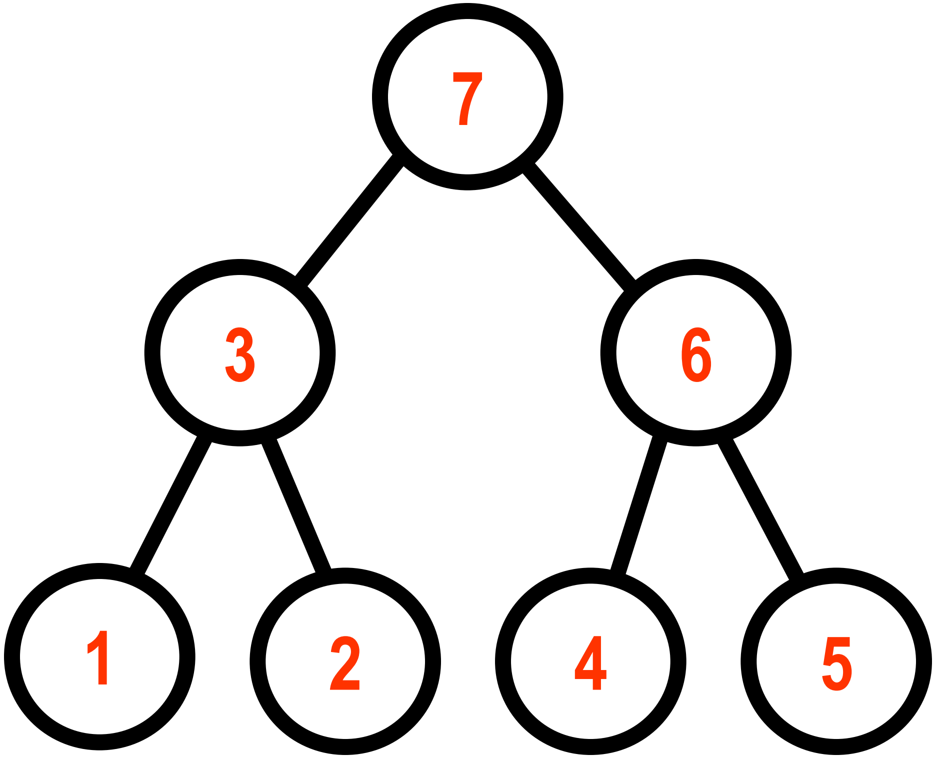
```
typedef void (*fType)(DataType& AnItem);

void inorder(BTnode* rootBT, fType Op)
{
    if (rootBT != NULL)
    {
        inorder(rootBT-> LchildPtr);
        Op(rootBT->Data);
        inorder(rootBT-> RchildPtr);
    }
}
```

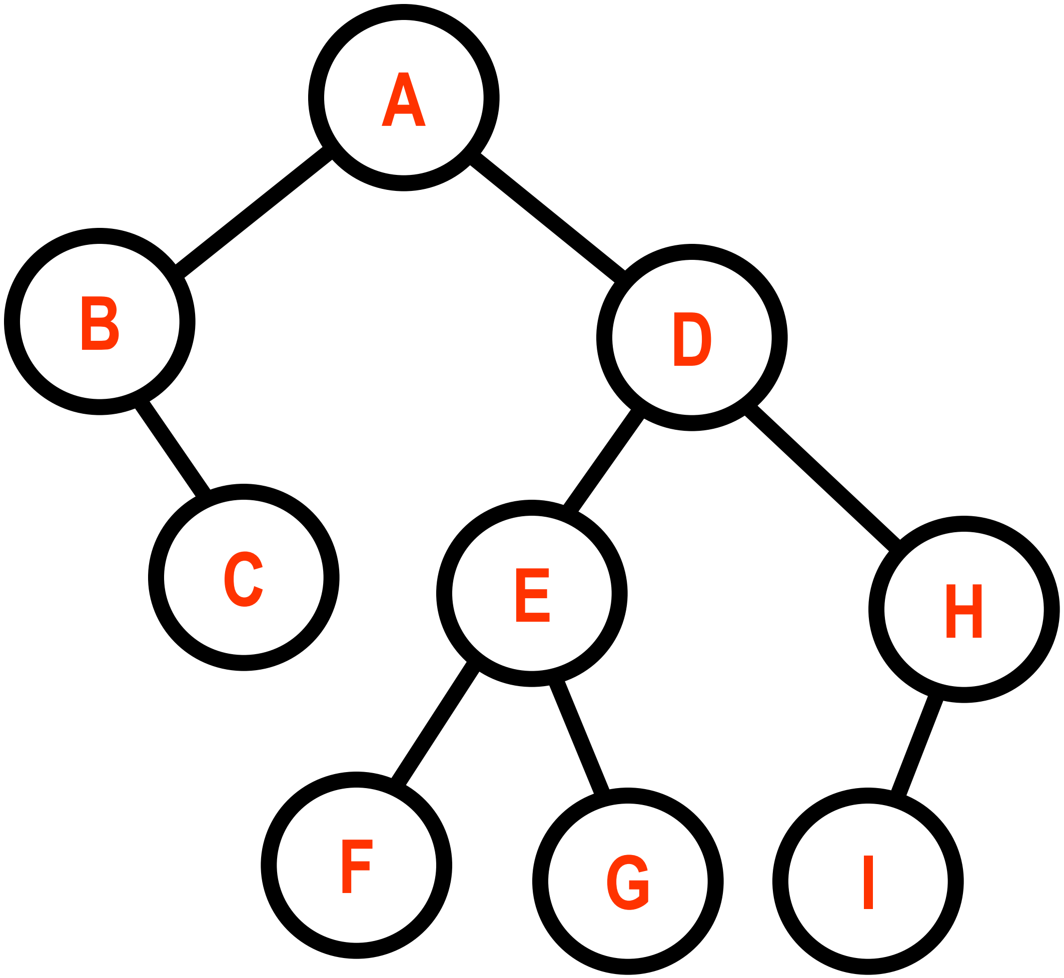
3. Post-order Traversal

- If the tree is not empty then
  - Postorder traverse the left subtree recursively
  - Postorder traverse the right subtree recursively
  - Visit the root

Post-order Traversal - Processing Order



Example: Postorder traversal



C B F G E I H D A

Postorder Traversal & Print

```
void postorder_print(BTnode* rootBT)
{
    if (rootBT != NULL)
    {
        postorder_print(rootBT-> LchildPtr);
        postorder_print(rootBT-> RchildPtr);
        cout << rootBT->Data << endl;
    }
}
```

Postorder Traversal and Operation

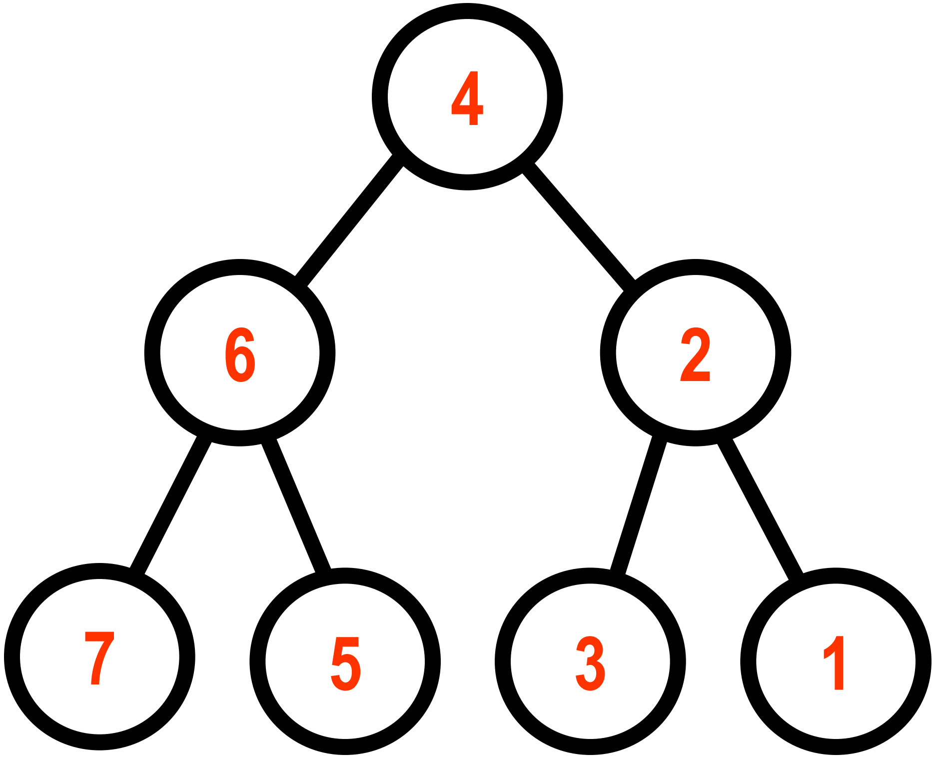
```
typedef void (*fType)(DataType& AnItem);

void postorder(BTnode* rootBT, fType Op)
{
    if (rootBT != NULL)
    {
        postorder(rootBT-> LchildPtr);
        postorder(rootBT-> RchildPtr);
        Op(rootBT->Data);
    }
}
```

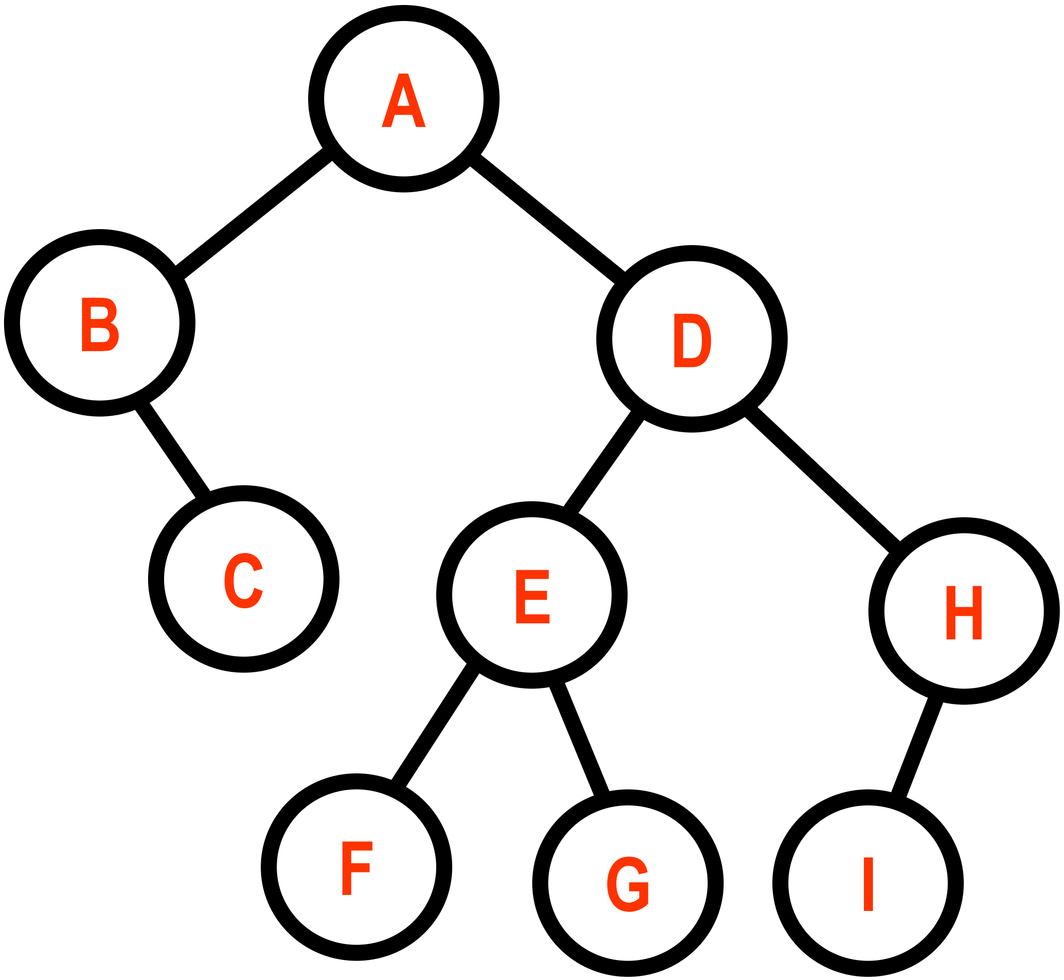
5. Backward In-order Traversal

- 5. Process right-child subtree, then a node, then left-child subtree.
- If the tree is not empty then
  - ➔Backward Inorder traverse the right subtree recursively
  - ➔Visit the root
  - ➔Backward Inorder traverse the left subtree recursively

Backward In-order Traversal - Processing Order

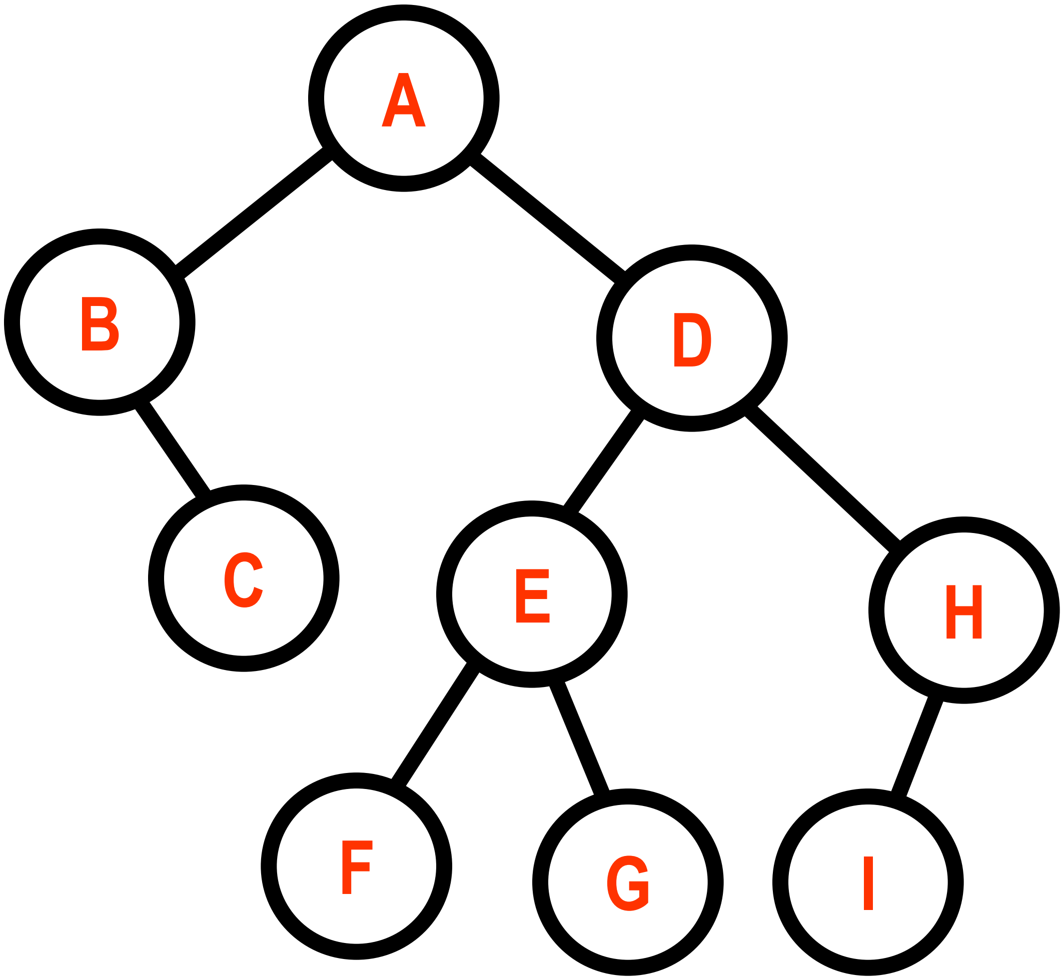


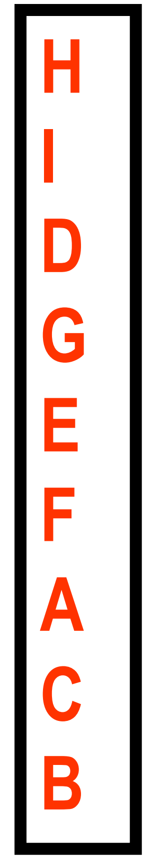
Example: Backward Inorder Traversal

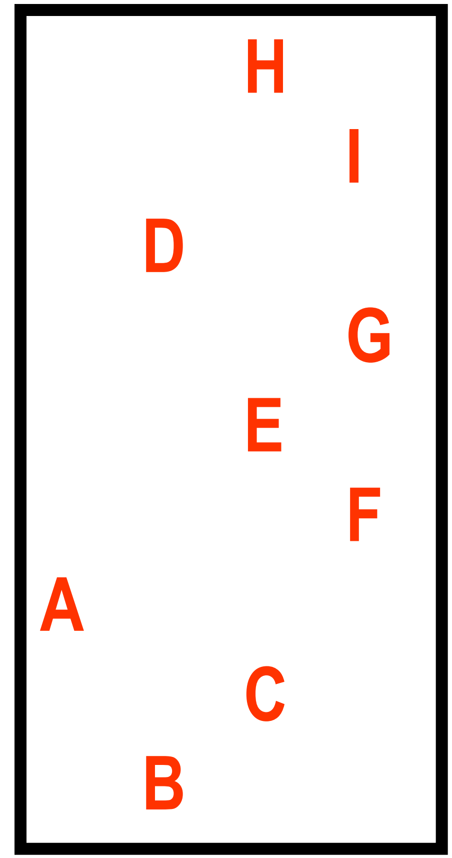


H I D G E F A C B

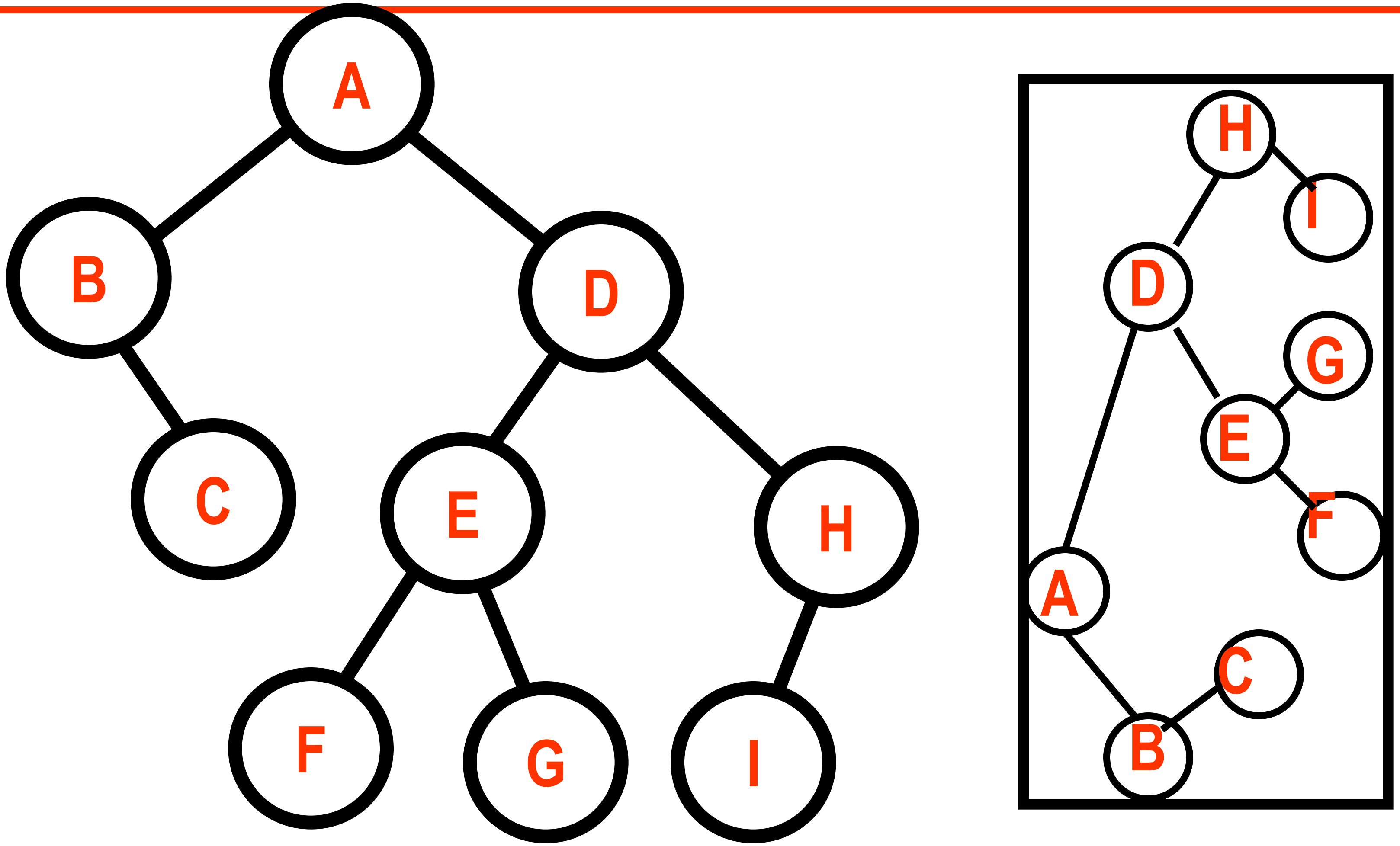
Example: Backward Inorder Traversal



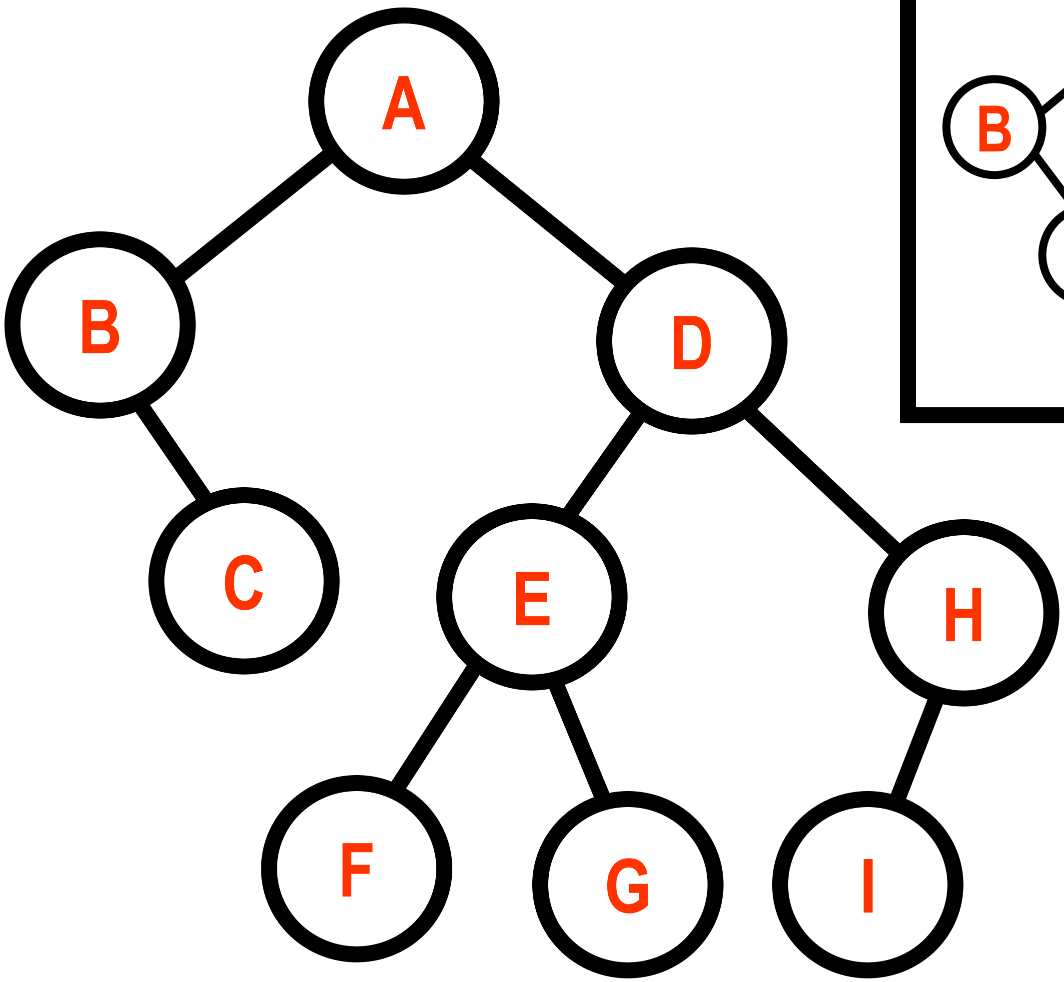


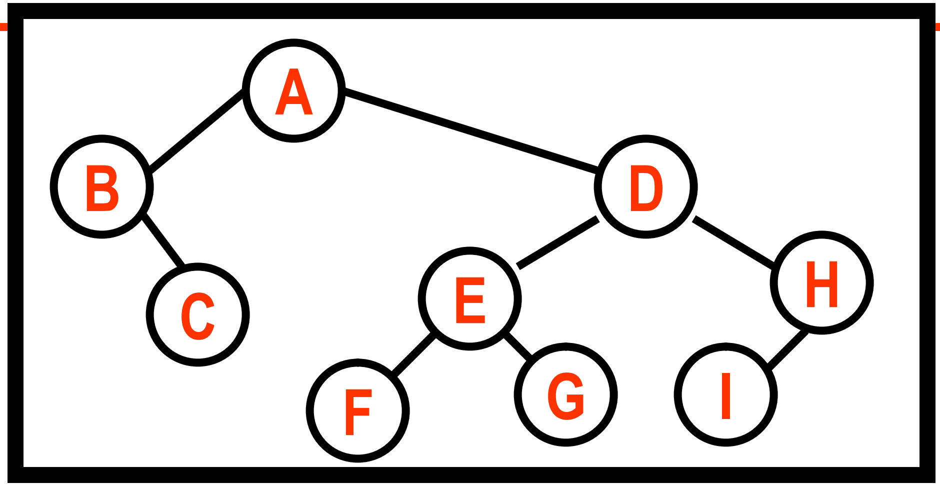


Example: Backward Inorder traversal

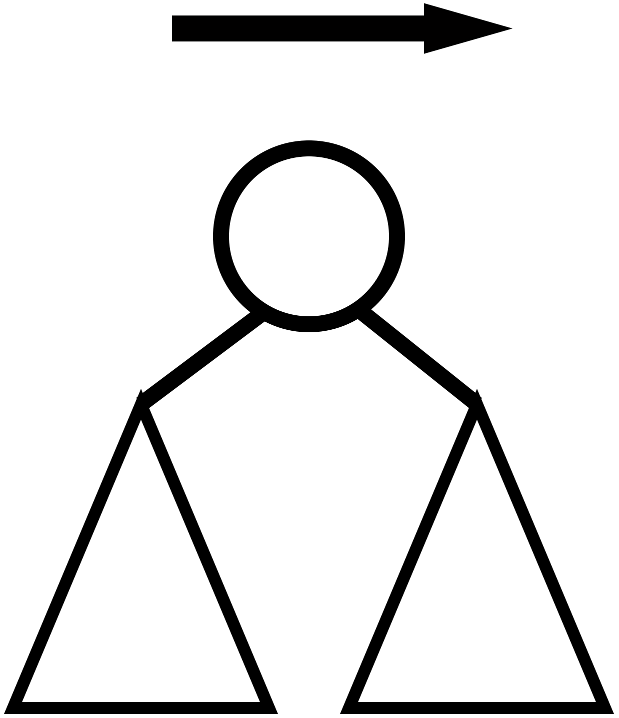


Example: Backward Inorder traversal





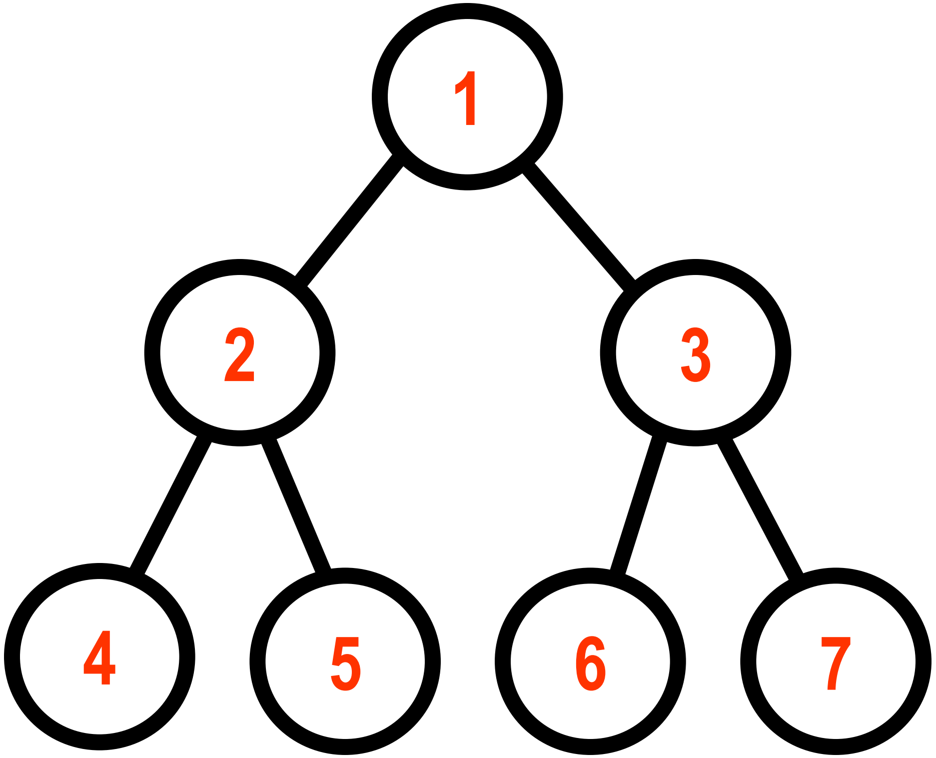
Breadth-First Traversal of Binary Trees



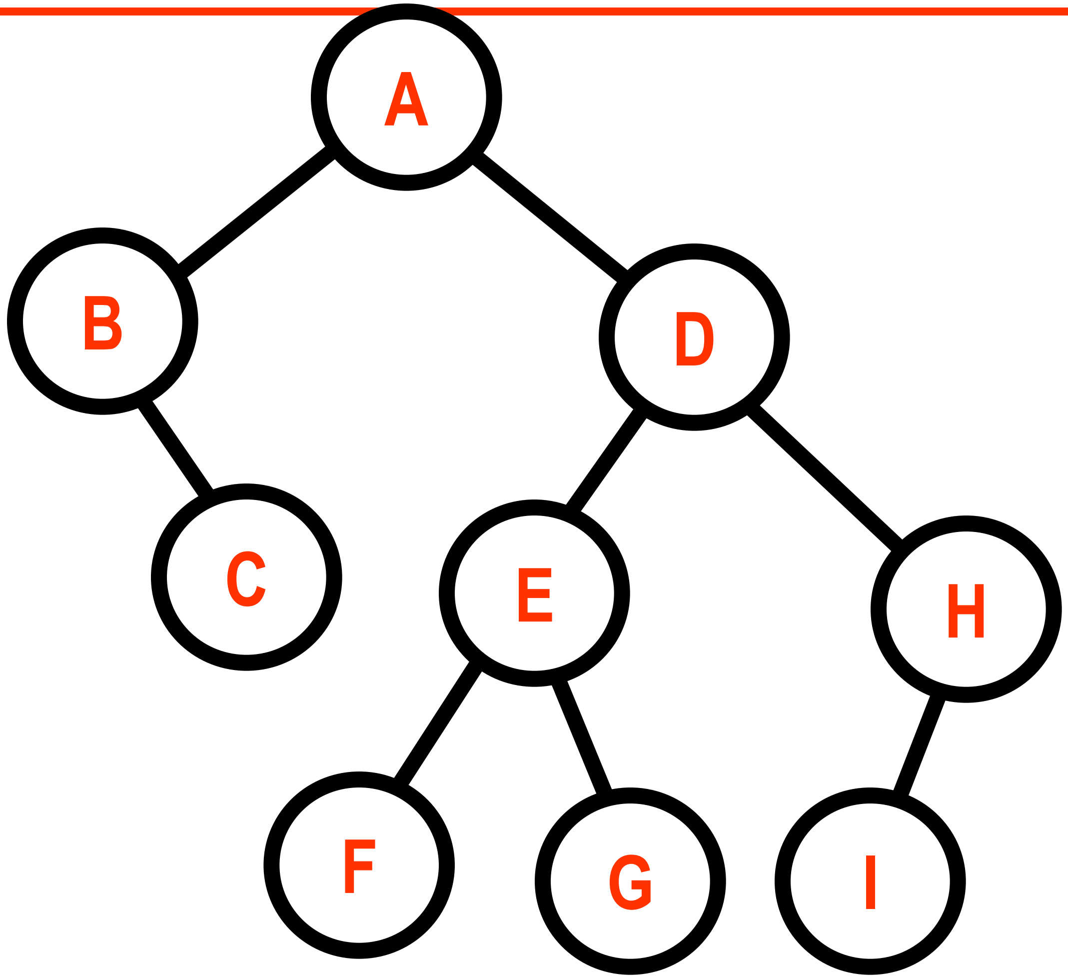
Breadth-First (Level-order)Traversal

- If the tree is not empty then then visit the nodes in the order of their level (depth) in the tree.
  - Visit all the nodes at depth zero (the root).
  - Then, all the nodes from left to right at depth one
  - Then, all the nodes from left to right at depth two
  - Then, all the nodes from left to right at depth three
  - And so on ...

Level-order Traversal - Processing Order



Example: Levelorder traversal



A B D C E H F G I

Expression Tree ADT

Algebraic Expressions: Notations

- Algebraic expressions
  - Fully parenthesized Infix notation
  - Not fully parenthesized Infix notation
  - Postfix notation
  - Prefix notation

1. Fully Parenthesized Infix Notation

- We need to place parentheses around each pair of operands together with their operator!
- Examples:
  - (1+2)
  - (1+(2 * 3))
  - ((1 + 2) * 3)
  - ((a / b) + ((c - d) * e))
- Inconvenient!

2. Not Fully Parenthesized Infix Notation

- We can omit unnecessary parentheses!
- Examples:
  - ➔(1+2) ➔ 1 + 2
  - ➔(1+ (2 * 3)) ➔ 1+ 2 * 3
  - ➔((1 + 2) * 3) ➔ (1 + 2) * 3
  - ➔((a / b) + ((c - d) * e)) ➔ a / b + (c - d) * e
- Convenient, BUT, we need rules to interpret correctly.

Not Fully Parenthesized Infix Notation

- Operator precedence rule
  - ➔ * / higher than + -
- Operator association rule
  - ➔ Associate from left to right
- Examples: (Interpretation using rules)
  - ➔ 1 + 2 ➔ (1+2)
  - ➔ 1+ 2 * 3 ➔ (1+ (2 * 3))
  - ➔ (1 + 2) * 3 ➔ ((1 + 2) * 3)
  - ➔ a / b + (c - d) * e ➔ ((a / b) + ((c - d) * e))

3. Postfix Notation

- Postfix Notation:
  - ➔<postfix> := <id> | (<postfix> <postfix> <op>)
  - ➔<op> := + | - | * | /
  - ➔<id> := <variable> | <number>
- Examples:
  - ➔(1+2) ➔ 1+ 2 ➔ 1 2 +
  - ➔(1+ (2 * 3)) ➔ 1+ 2 * 3 ➔ 1 2 3 * +
  - ➔((1 + 2) * 3) ➔ (1 + 2) * 3 ➔ 1 2 + 3 *
  - ➔((a / b) + ((c - d) * e)) ➔ a / b + (c - d) * e ➔ a b / c d - e * +

4. Prefix Notation

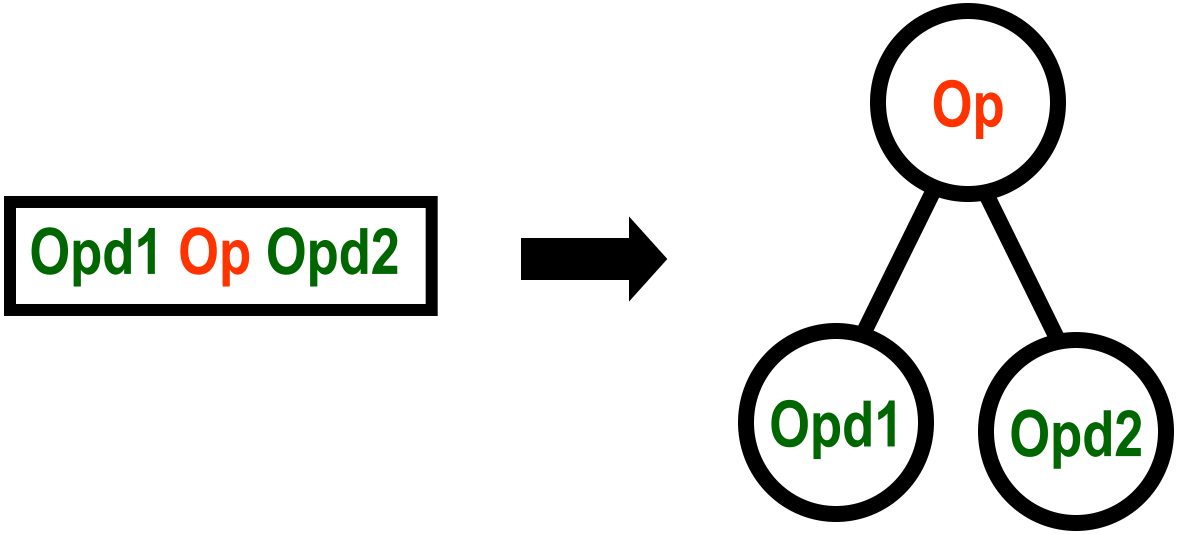
- Postfix Notation:
  - <prefix> := <id> | (<op> <prefix> <prefix>)
  - <op> := + | - | * | /
  - <id> := <variable> | <number>
- Examples:
  - (1+2) ➔ 1+ 2 ➔ + 1 2
  - (1+ (2 * 3)) ➔ 1+ 2 * 3 ➔ + 1 * 2 3
  - ((1 + 2) * 3) ➔ (1 + 2) * 3 ➔ * + 1 2 3
  - ((a / b) + ((c - d) * e)) ➔ a / b + (c - d) * e ➔ + / a b * - c d e

Postfix and Prefix Notations

- Postfix and prefix notations do not need!
  - Parentheses
  - Operator precedence rules
  - Operator association rules

Algebraic Expressions as Expression Trees

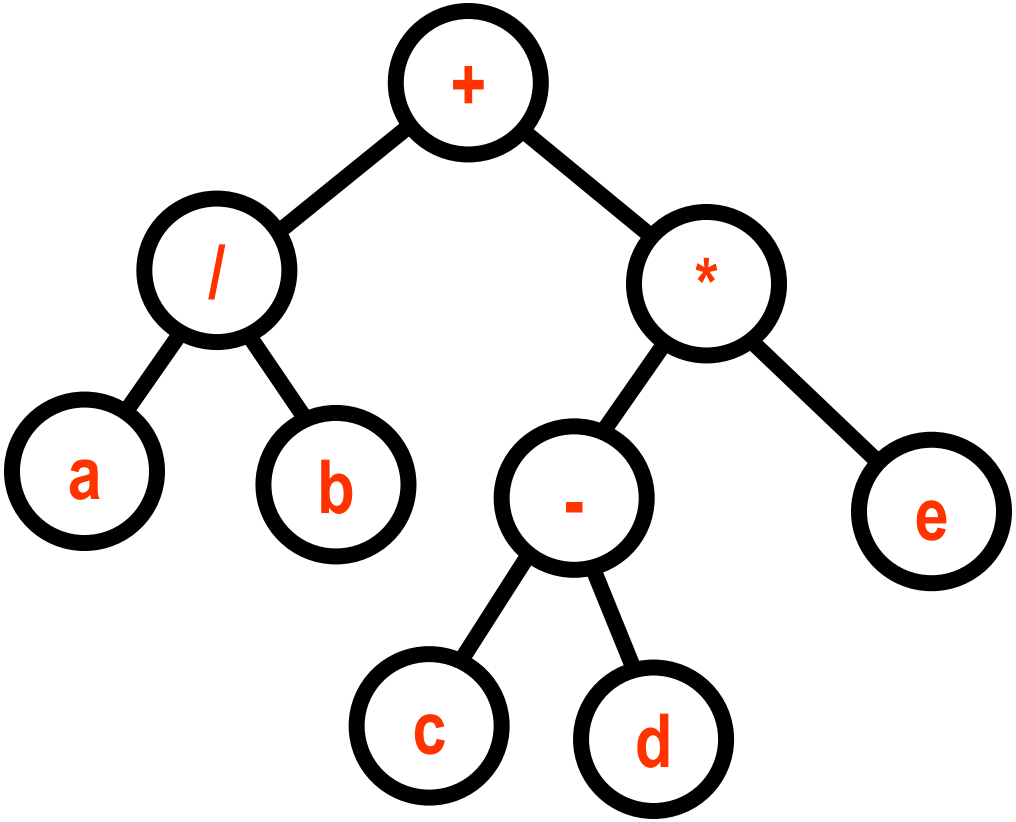
- Algebraic expressions involving binary operations can be represented by labeled binary trees.
- Expression trees represent algebraic expressions!



Algebraic Expression
a / b + (c - d) * e

- ((a / b) + ((c - d) * e))
- a b / c d - e * +
- + / a b * - c d e

Expression Tree for a / b + (c - d) * e



Expression Tree ADT

```
//  exprtree.h
//  Class declarations for the linked implementation of the
//  Expression Tree ADT
//------------------------------------------------------------------
class ExprTree;          // Forward declaration of the ExprTree class

class ExprTreeNode       // Facilitator class for the ExprTree class
{
 private:
   // Constructor
   ExprTreeNode ( char elem,
                  ExprTreeNode *leftPtr, ExprTreeNode *rightPtr );

   // Data members
   char dataItem;          // Expression tree data item
   ExprTreeNode *left,     // Pointer to the left child
                *right;    // Pointer to the right child
 friend class ExprTree;
};
```

Expression Tree ADT

```
class ExprTree
{
  public:
    // Constructor
    ExprTree ();
    // Destructor
    ~ExprTree ();
    // Expression tree manipulation operations
    void build ();                // Build tree from prefix expression
    void expression () const;   // Output expression in infix form
    float evaluate () const;     // Evaluate expression
    void clear ();                // Clear tree
```

Expression Tree ADT

```
// Output the tree structure -- used in testing/debugging
    void showStructure () const;

    ExprTree ( const ExprTree &valueTree );   // Copy constructor
```

Expression Tree ADT

```
private:
    // Recursive partners of the public member functions -- insert
    // prototypes of these functions here.
    void buildSub ( ExprTreeNode *&p );
    void exprSub ( ExprTreeNode *p ) const;
    float evaluateSub ( ExprTreeNode *p ) const;
    void clearSub ( ExprTreeNode *p );
    void showSub ( ExprTreeNode *p, int level ) const;
    // Data member
    ExprTreeNode *root;   // Pointer to the root node
};
```

Expression Tree ADT

```
//--------------------------------------------------------------------
//  exprtree.cpp
//--------------------------------------------------------------------
ExprTreeNode:: ExprTreeNode ( char nodeDataItem,
                              ExprTreeNode *leftPtr,
                              ExprTreeNode *rightPtr )
// Creates an expreesion tree node containing
//  data item nodeDataItem,
// left child pointer leftPtr, and right child pointer rightPtr.

  : dataItem(nodeDataItem),
    left(leftPtr),
    right(rightPtr)
{}
```

Expression Tree ADT

```
ExprTree:: ExprTree ()

// Creates an empty expression tree.

  : root(0)
{}
```

Expression Tree ADT

```
//--------------------------------------------------------------------
ExprTree:: ~ExprTree ()
// Frees the memory used by an expression tree.
{
   clear();
}
void ExprTree:: clear ()

// Removes all the nodes from an expression tree.

{
   clearSub(root);
   root = 0;
}
```

Expression Tree ADT

```
void ExprTree:: clearSub ( ExprTreeNode *p )

// Recursive partner of the clear() function. Clears the subtree
// pointed to by p.

{
   if ( p != 0 )
   {
      clearSub(p->left);
      clearSub(p->right);
      delete p;
   }
}
```

Expression Tree ADT

```
void ExprTree:: build ()
// Reads a prefix expression (consisting of single-digit, nonnegative
// integers and arithmetic operators) from the keyboard and
// builds the corresponding expression tree.
{
   buildSub(root);
}
```

Expression Tree ADT

```
void ExprTree:: buildSub ( ExprTreeNode *&p )
// Recursive partner of the build() function. Builds a subtree and
// sets p to point to its root.
{
    char ch;   // Input operator or number
    cin >> ch;
    p = new ExprTreeNode(ch,0,0);   // Link in node
    if ( !isdigit(ch) )             // Operator -- construct subtrees
    {
       buildSub(p->left);
       buildSub(p->right);
    }
}
```

Expression Tree ADT

```
void ExprTree:: expression () const
// Outputs the corresponding arithmetic expression in fully
// parenthesized infix form.
{
   exprSub(root);
}
```

Expression Tree ADT

```
void ExprTree:: exprSub ( ExprTreeNode *p ) const
// Recursive partner of the expression() function.
// Outputs the subtree pointed to by p.
{
   if ( p != 0 )
   {
      if ( !isdigit(p->dataItem) ) cout << '(';
      exprSub(p->left);
      cout << p->dataItem;
      exprSub(p->right);
      if ( !isdigit(p->dataItem) ) cout << ')';
   }
}
```

Expression Tree ADT

```
float ExprTree:: evaluate ()
// Returns the value of the corresponding arithmetic expression.

{
   // Requires that the tree is not empty
return evaluateSub(root);
}
```

Expression Tree ADT

```
float ExprTree:: evaluateSub ( ExprTreeNode *p ) const
// Recursive partner of the evaluate() function. Returns the value of
// subtree pointed to by p.
{
    float l, r,     // Intermediate results
          result;   // Result returned
    if ( isdigit(p->dataItem) )
      result = p->dataItem - '0';      // Convert from char to number
    else
    {
      l = evaluateSub(p->left);       // Evaluate subtrees
      r = evaluateSub(p->right);
      switch ( p->dataItem )          // Combine results
      {
        case '+' :  result = l + r;  break;
        case '-' :  result = l - r;  break;
        case '*' :  result = l * r;  break;
        case '/' :  result = l / r;
      }
    }
    return result;
}
```