CS 2351 Data Structures

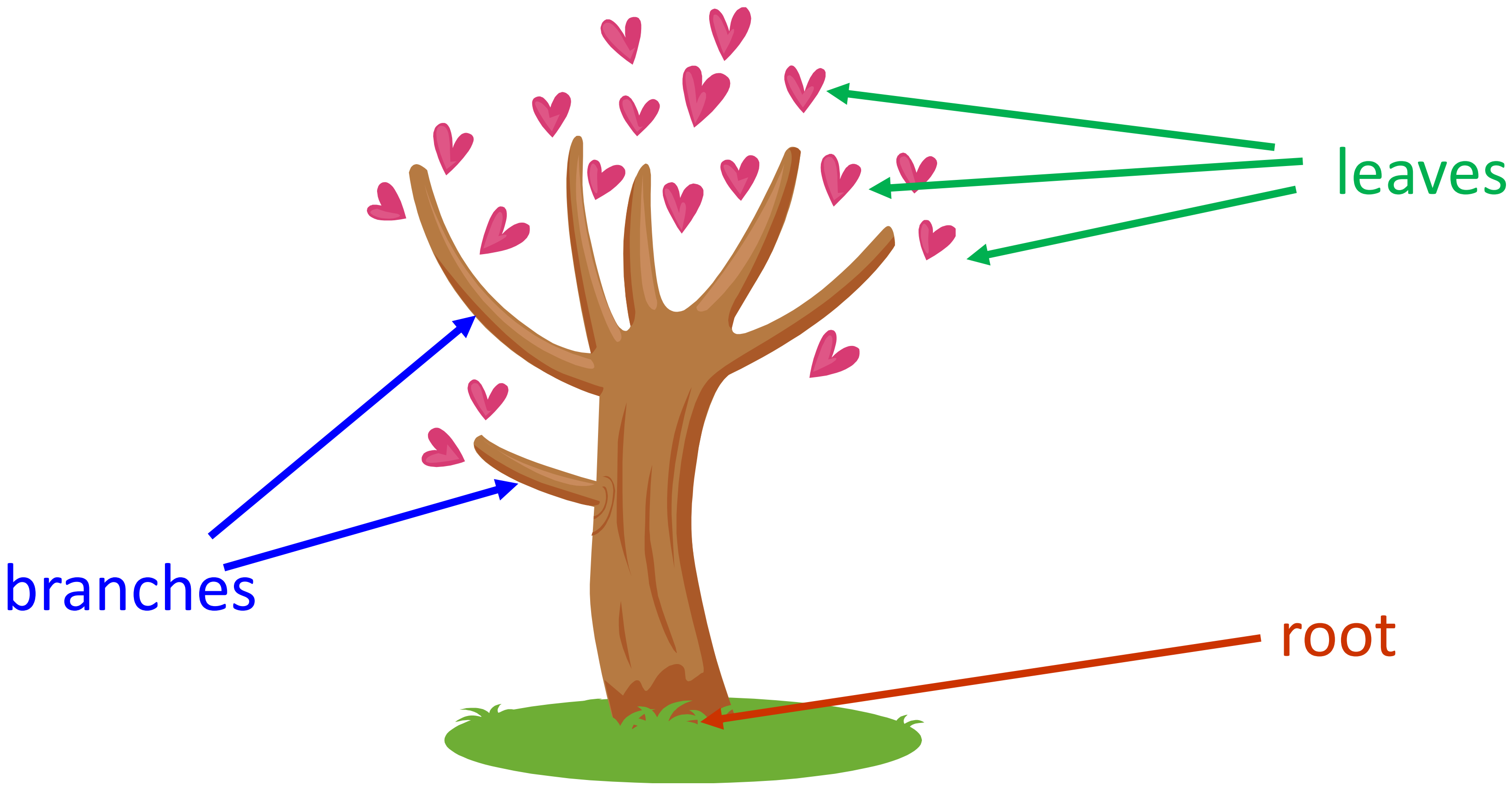
Trees (I)

Prof. Chung-Ta King
Department of Computer Science
National Tsing Hua University

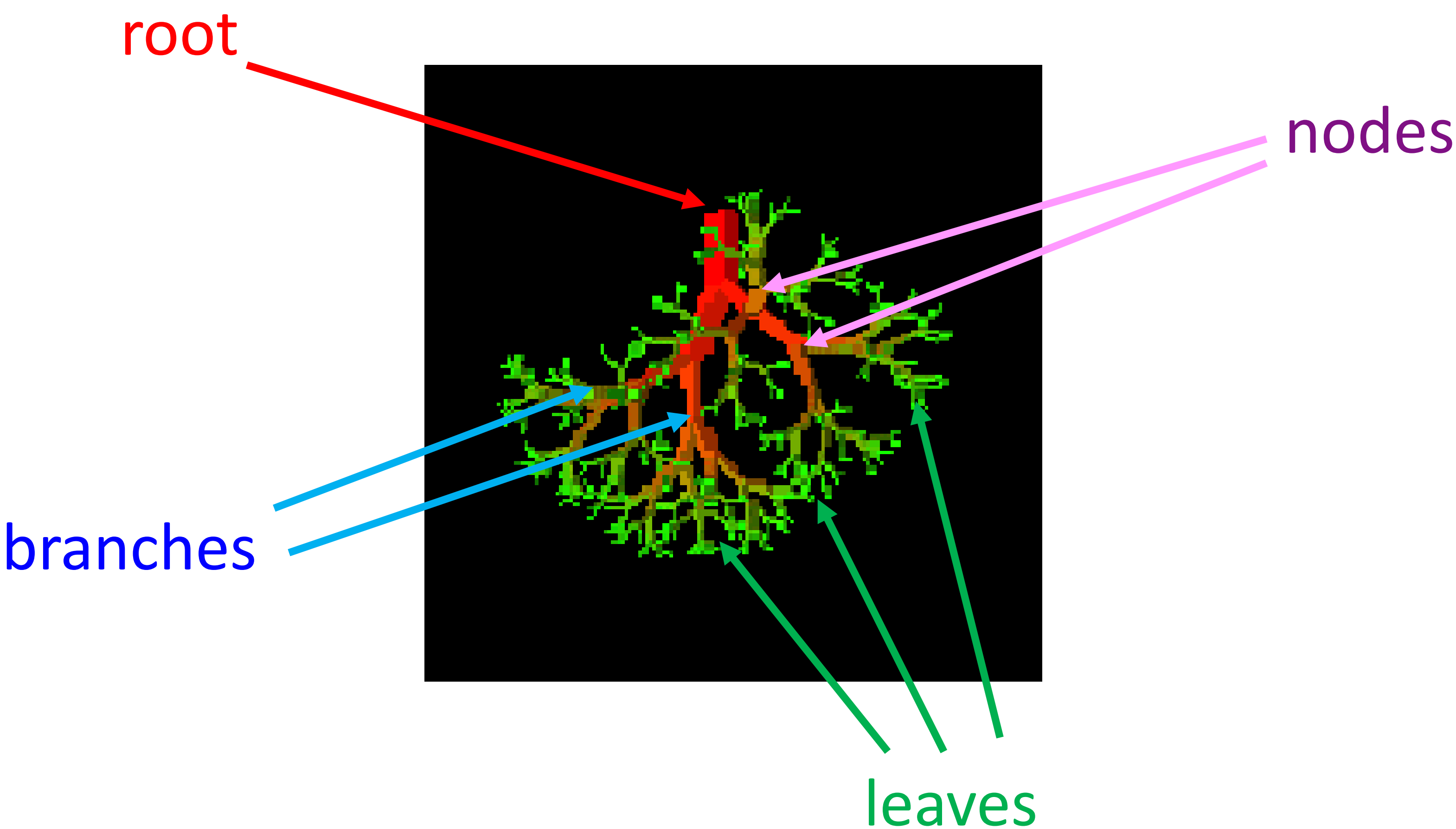
Outline

- Trees and their representation (Sec. 5.1)
- Binary trees (Sec. 5.2)
- Binary tree traversal and tree iterators (Sec. 5.3)

Nature Lover's View of a Tree

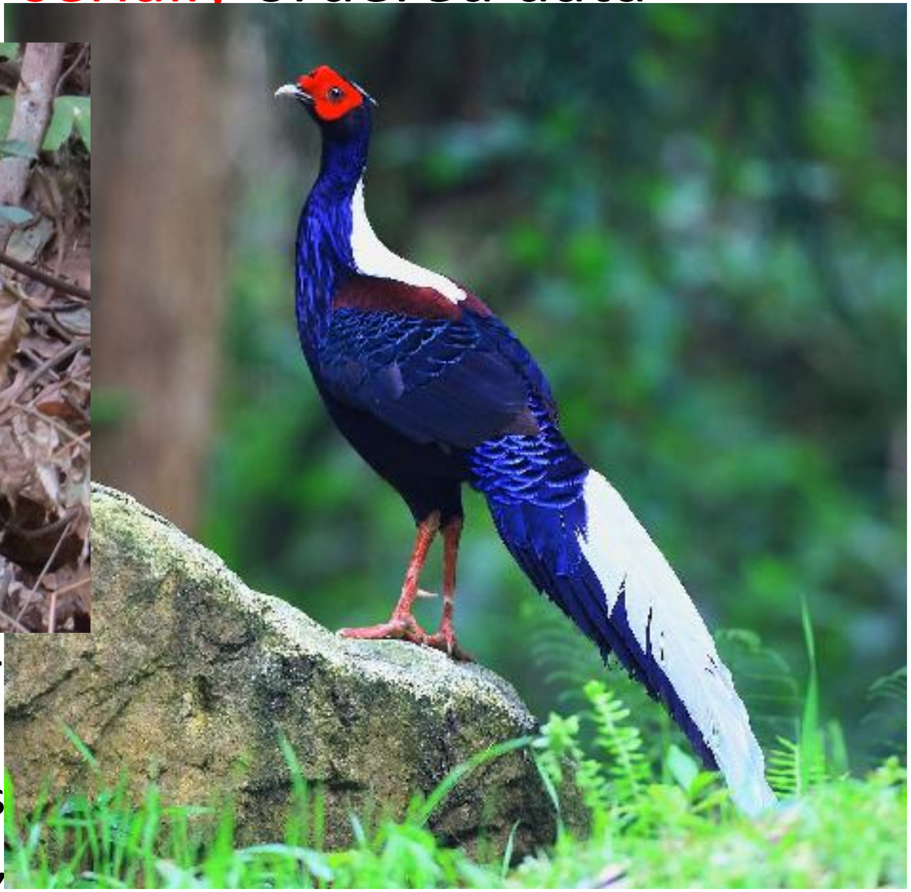


Computer Scientist's View



From Linear Lists to Trees

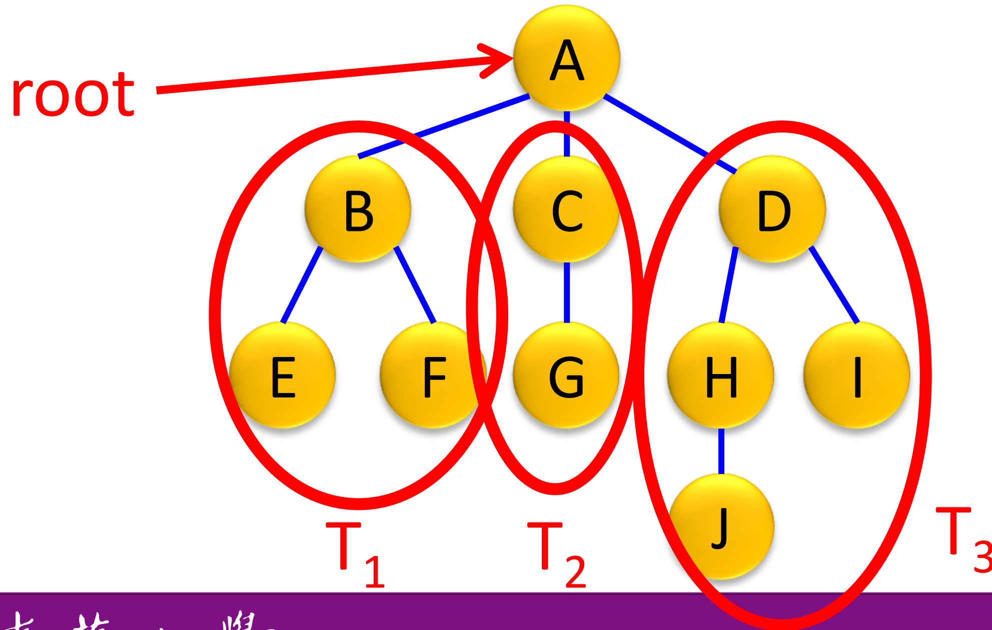
- Linear lists are useful for serially ordered data



脊索動物門 (Phylum Chor
鳥綱 (Class Aves),
雞形目 (Order Galliformes
雉科 (Family Phasianidae),
鷴屬 (Genus Lophura), **藍腹鷴** (Species L. swinhoii)

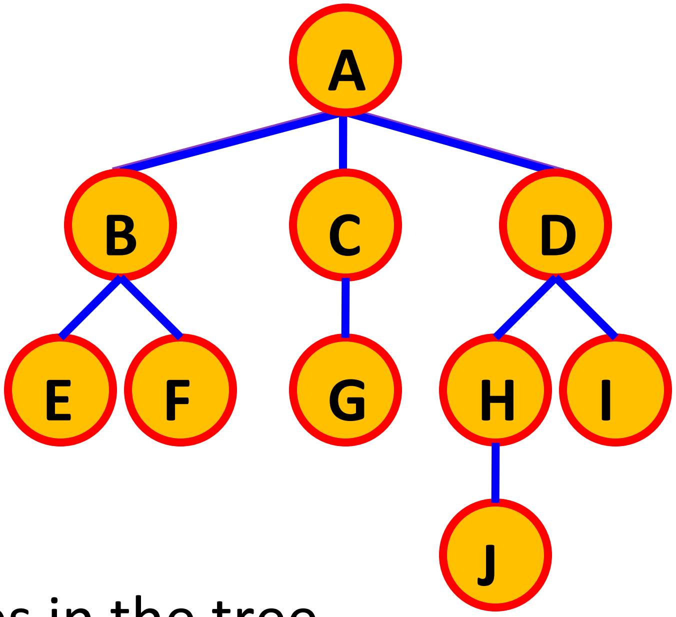
Tree Definition

- A **tree** is a finite set of one or more nodes
  - A specially designated node called ***root***
  - Remaining nodes are partitioned into n ≥ 0 disjointed sets T_1, T_2, … T_n, where each is a tree → recursive definition
  - T_1, T_2, … T_n are called ***subtrees*** of the root



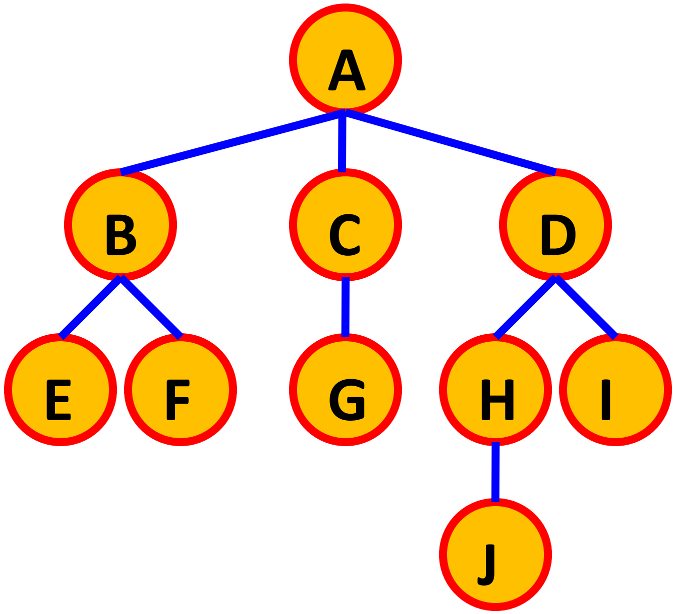
Terminology

- Degree of a node
  - The number of subtrees
  - e.g. deg(A) =3; deg(C) =1
- Leaf or terminal nodes
  - The node whose degree is 0
  - e.g. E, F, G, J, I
- Nonterminals
- Degree of a tree
  - The maximum degree of the nodes in the tree
  - e.g. degree of the tree = 3



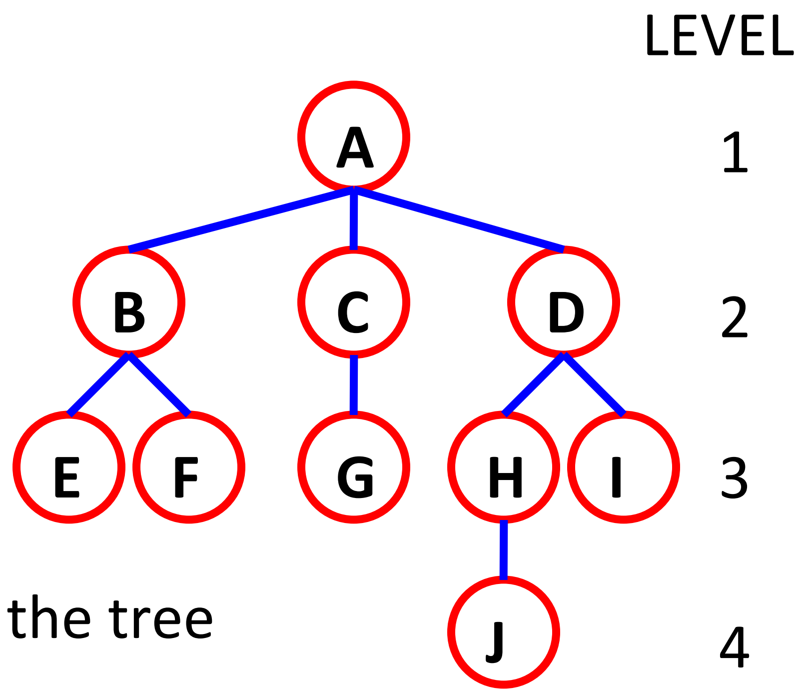
Terminology

- Parent/children
- Sibling
  - Children of the same parent
  - e.g. E and F are siblings
- Ancestors
  - All nodes along the path from the root to that node
  - e.g. ancestor of J → H, D, A
- Descendants
  - All nodes in the subtrees



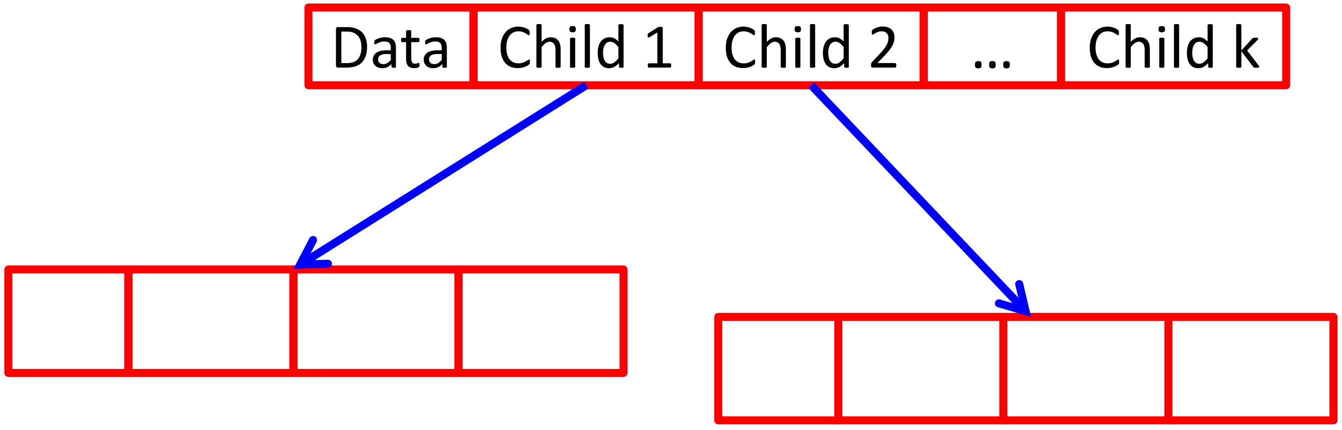
Terminology

- Level of a node
  - Level(root) = 1
  - Level(node) = l + 1
    if level of node's parent is l
  - e.g. level(G) = 3
- Height or depth of a tree
  - Maximum level of any node in the tree
  - e.g. height of the tree = 4



List Representation

- Each tree node holds **a data field** and **several link fields** pointing to subtrees
  - However, the degree of each node might vary
  - For a tree of **degree k** (k-ary tree), allocate k link fields for each node

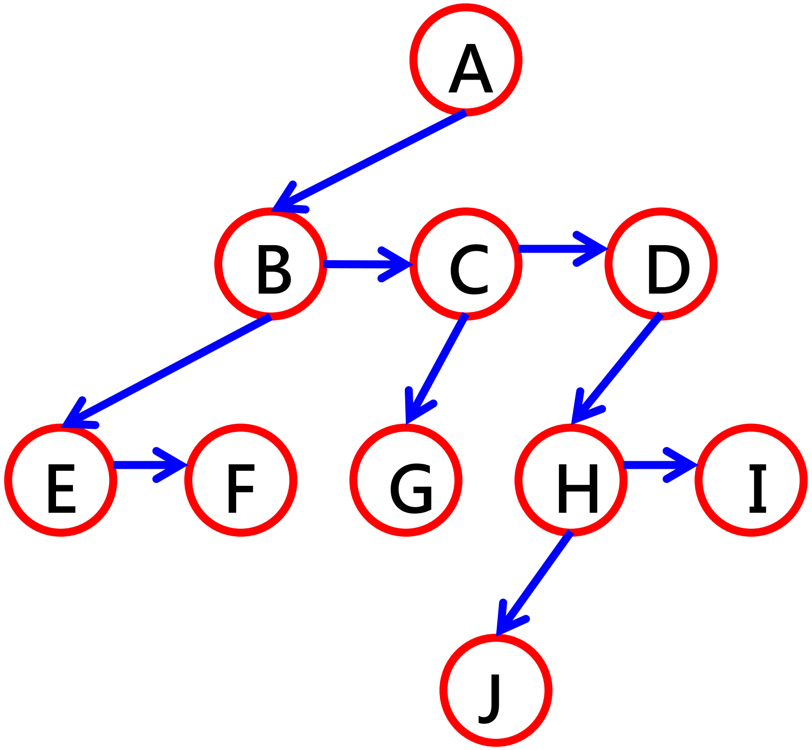


List Representation

- Disadvantage: waste memory!
  - If T is a tree of degree k with n nodes
  - The total # of link fields is $n \times k$
  - The total # of used link fields is n-1
    - For each node (except ***root***), there is one and only one pointer points to it
  - The # of zero link fields is $n \times k - (n - 1)$

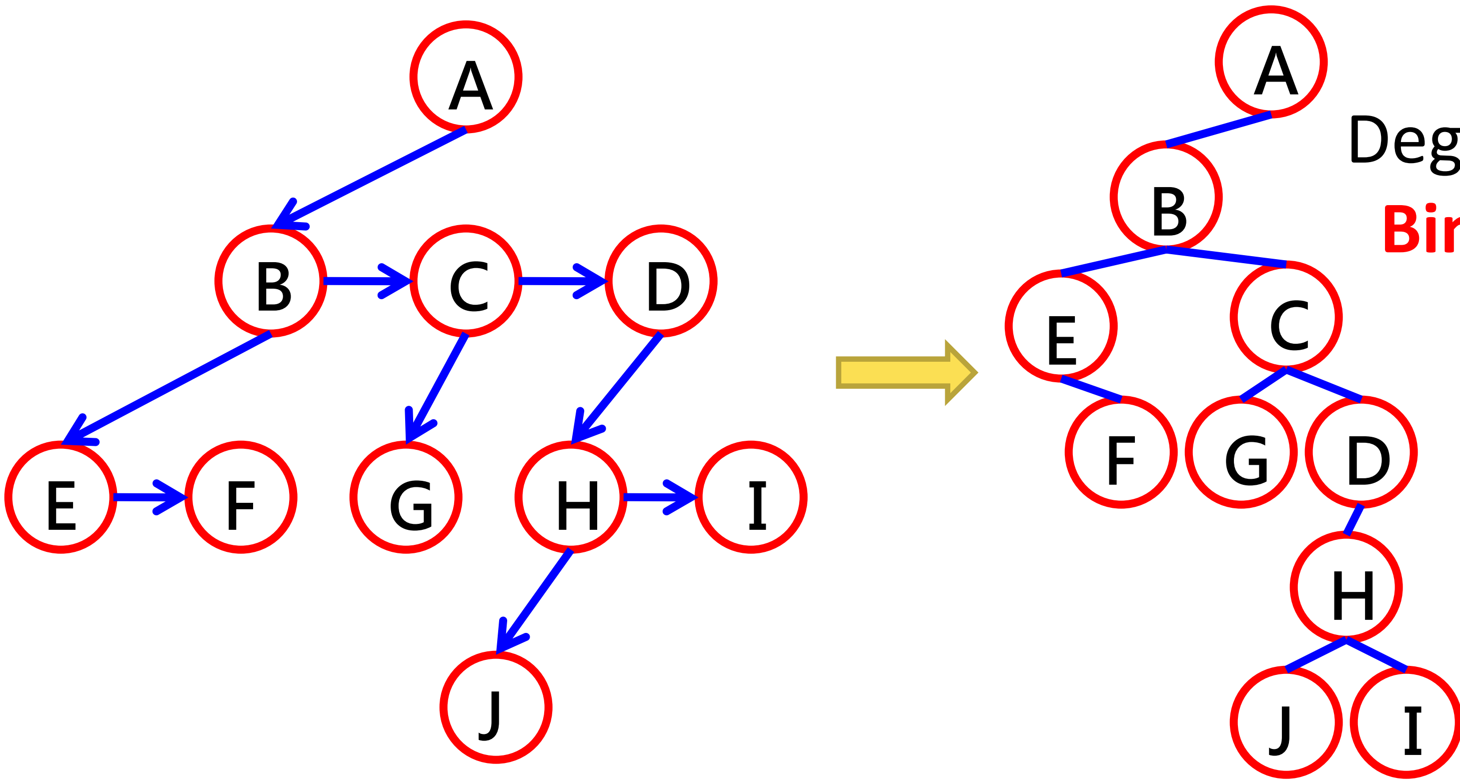
Left Child-Right Sibling Representation

- Each node has exactly **two link fields**
  - Left link (child): points to **leftmost child node**
  - Right link (sibling): points to **closest sibling node**



Left Child-Right Sibling Representation

- Rotate clockwise 45°



Degree-**2** tree

Binary Tree

Outline

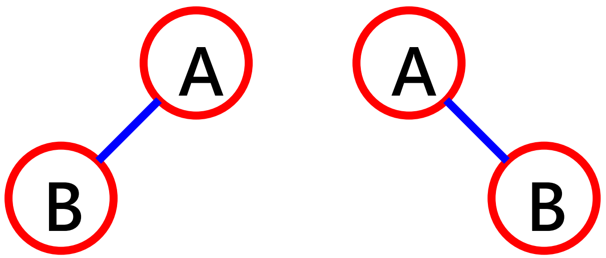
- Trees and their representation (Sec. 5.1)
- Binary trees (Sec. 5.2)
- Binary tree traversal and tree iterators (Sec. 5.3)

Binary Tree

- Definition: a ***binary tree*** is a finite set of nodes that either is ***empty*** or consists of a ***root*** and two disjoint binary trees called the ***left subtree*** and the ***right subtree***.
- Binary tree ≠ Regular tree

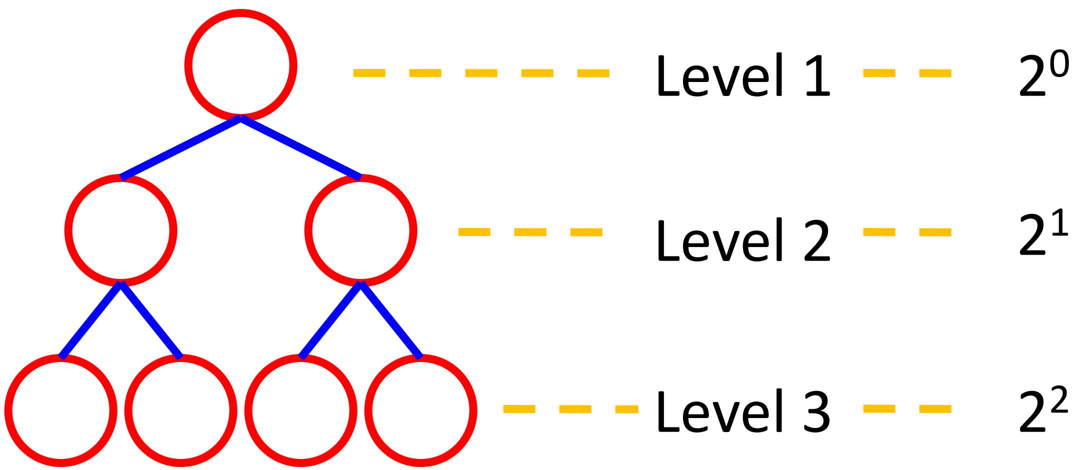
	Binary Tree	**Regular Tree**
Has zero nodes	YES	NO
Order of children	Important	Doesn't matter

Same trees but different binary trees



Properties of Binary Tree

- **[Maximum number of nodes]**
  - The max. # of nodes on level i is $2^{(i-1)}$
  - The max. # of nodes in a binary tree with depth k is $2^k - 1$



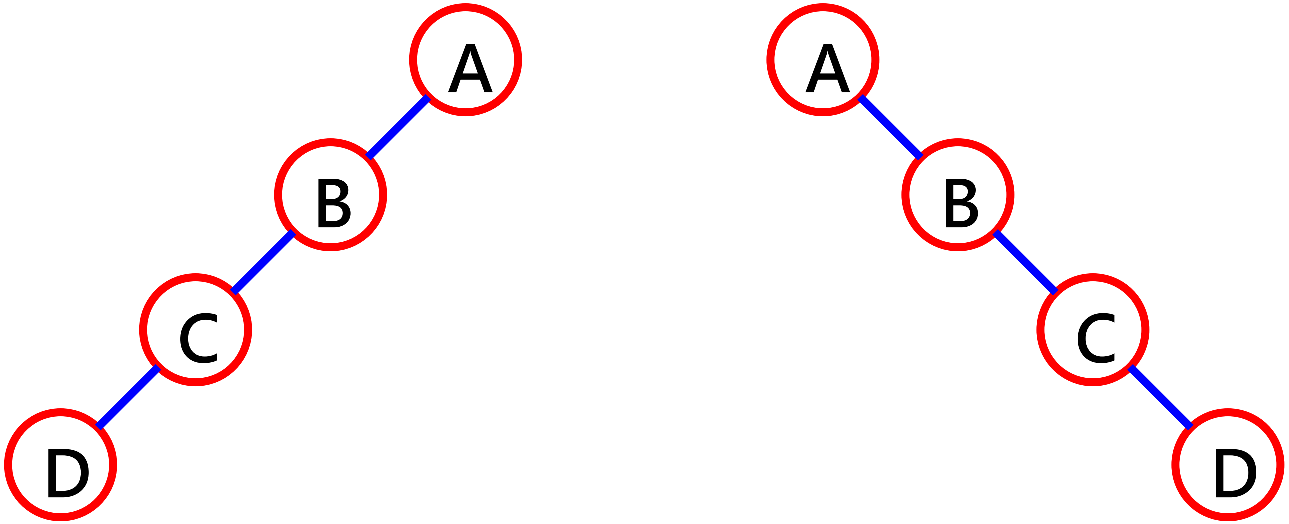
Total # of node is $1 + 2 + 2^2 + 2^3 + \ldots + 2^{(k-1)} = 2^k - 1$

Properties of Binary Tree

- **[Relation between # of leaf nodes and degree-2 nodes]**
  - if n_0 = number of leaf nodes and n_2 = number of degree-2 nodes, then $n_0 = n_2 + 1$
- Proof:
  - $n = n_0 + n_1 + n_2$, where n_1 is # of deg-1 nodes
  - $n = B + 1$, where B is # of branches
  - $B = n_1 + 2n_2$ (all branches stem from a node of degree 1 or 2)
  - $n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$
  - $n_0 = n_2 + 1$

Special Binary Tree

- **Skewed** tree

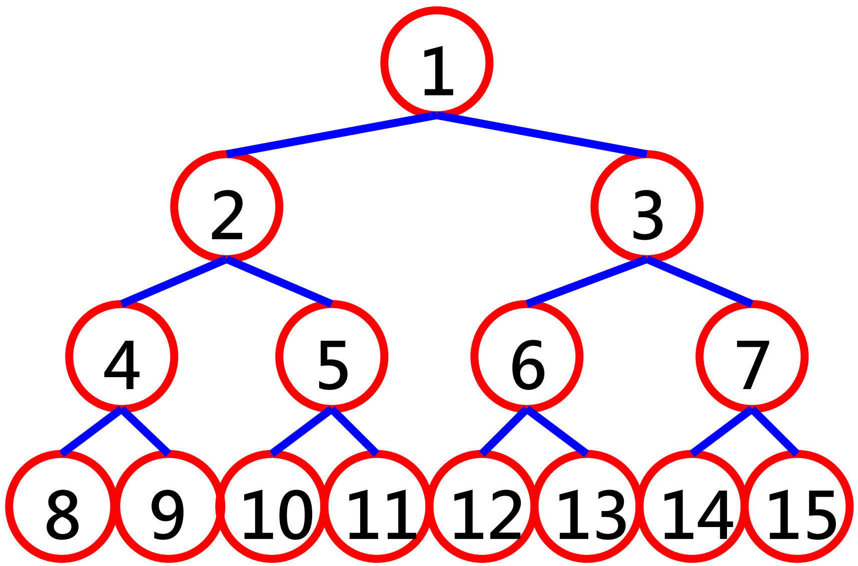


Skewed to the left

Skewed to the right

Special Binary Tree

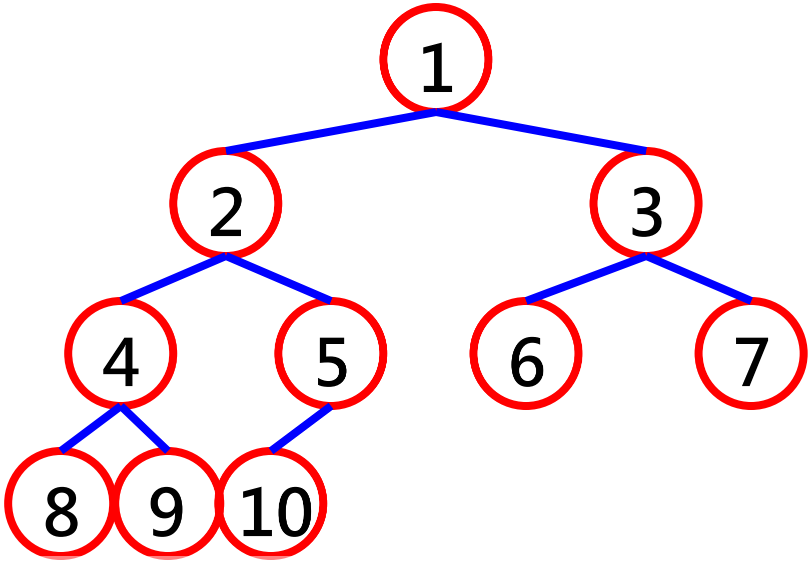
- **Full** binary tree
  - A binary tree of depth k which has $2^k - 1$ nodes



A full binary tree of depth 4

Special Binary Tree

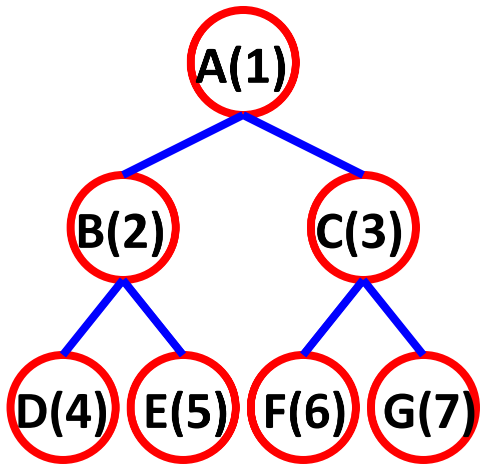
- **Complete** binary tree
  - A binary tree of depth k with n node is called **complete** iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree

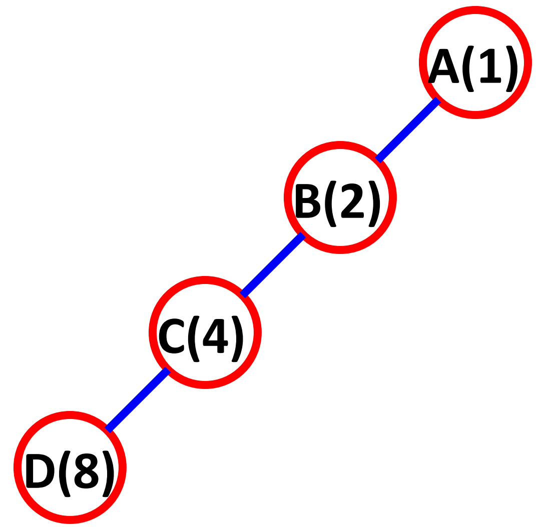


A complete binary tree of depth 4 with 10 nodes

Array Representation

- The numbering scheme suggests using a 1-D array to store the nodes





1	2	3	4	5	6	7
A	B	C	D	E	F	G

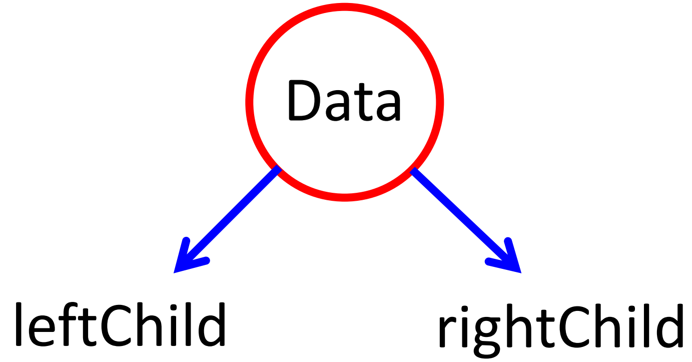
1	2	3	4	5	6	7	8
A	B		C				D

Array Representation

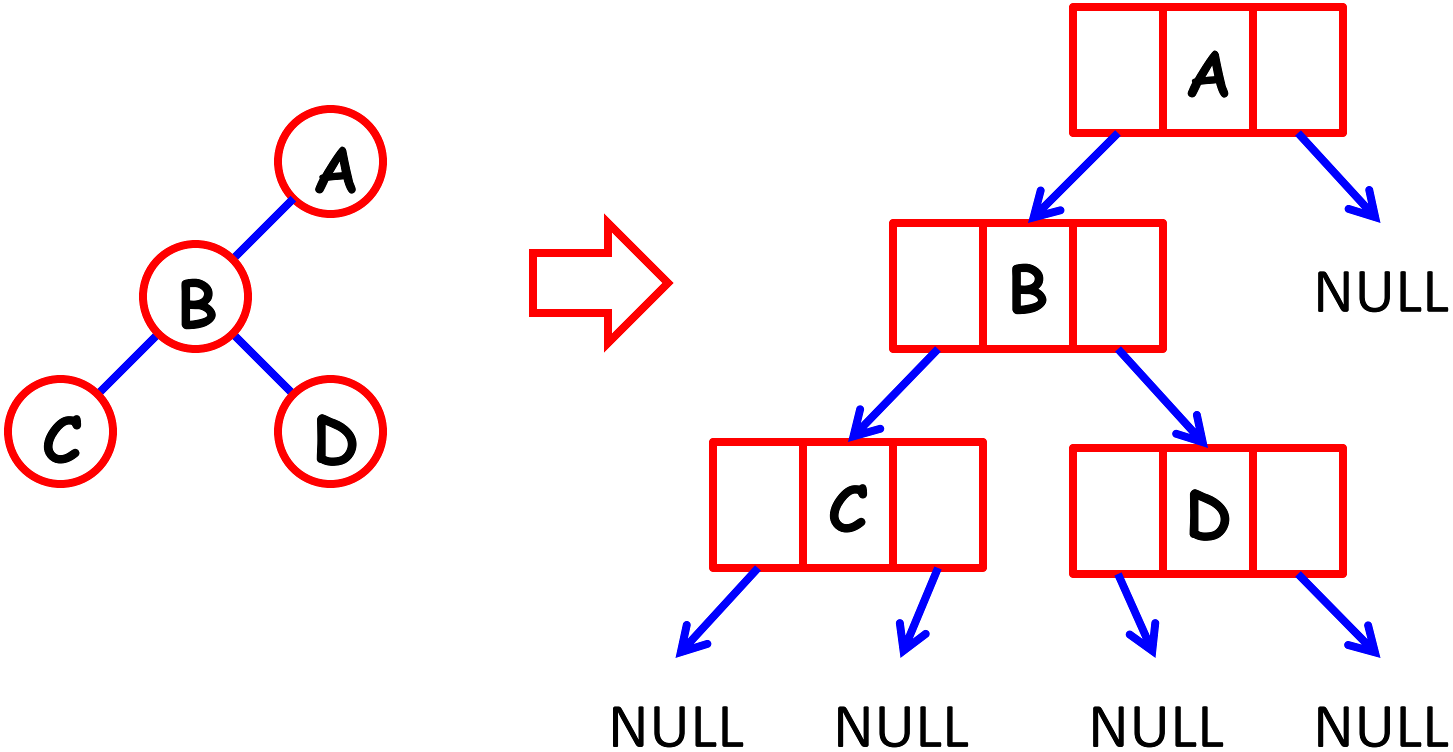
- Advantages: Easy to determine the locations of the parent, left child, and right child of any node
- Let node i be in position i (array[0] is empty)
  - **Parent(i) = $\lfloor i / 2 \rfloor$** if $i \neq 1$. If $i=1$, i is root and has no parent
  - **leftChild(i) = 2i** if $2i \le n$; if $2i > n$, i has no left child
  - **rightChild(i) = 2i+1** if $2i+1 \le n$; if $2i+1 > n$, i has no right child
- Disadvantages:
  - Waste space for a skewed tree
  - Insertion and deletion of nodes require moving a large parts of existing nodes

Linked Representation

- Similar to Chain structure in Chapter 4
- Each tree node consists of three fields
  - Data, leftChild, rightChild



Linked Representation



ADT of Binary Tree

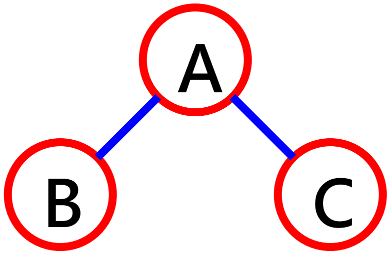
```
template <class T> class Tree;   // Forward decl.
template <class T>
Class TreeNode {
friend class Tree <T>;
private:
      T data;
      TreeNode<T>* leftChild;
      TreeNode<T>* rightChild;
};
template <class T>
Class Tree {
public:
      // Constructor
      Tree(void) {root=NULL;}
      // Tree operations here…
private:
      TreeNode<T> *root;
};
```

Outline

- Trees and their representation (Sec. 5.1)
- Binary trees (Sec. 5.2)
- Binary tree traversal and tree iterators (Sec. 5.3)

Binary Tree Traversal

- Visit each node in a tree exactly once
- Treat each node and its subtrees in the same fashion
  → recursive



Every time we visit a node A:

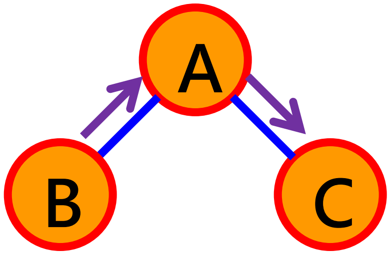
Inorder: visit left -> root -> right

Preorder: visit root -> left -> right

Postorder: visit left -> right -> root

Binary Tree Traversal

- Visit each node in a tree exactly once
- Treat each node and its subtrees in the same fashion



Every time we visit a node A:

Inorder: visit left -> root -> right

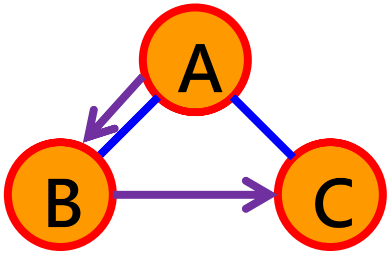
Preorder: visit root -> left -> right

Postorder: visit left -> right -> root

B A C

Binary Tree Traversal

- Visit each node in a tree exactly once
- Treat each node and its subtrees in the same fashion

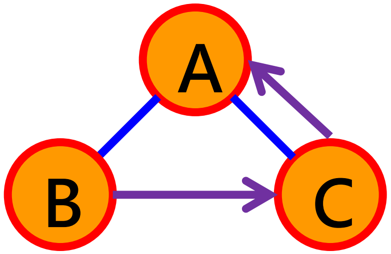


Every time we visit a node A:

Inorder:	visit left -> root -> right	BAC
Preorder:	visit root -> left -> right	ABC
Postorder:	visit left -> right -> root	

Binary Tree Traversal

- Visit each node in a tree exactly once
- Treat each node and its subtrees in the same fashion



Every time we visit a node A:

Inorder:	visit left -> root -> right	BAC
Preorder:	visit root -> left -> right	ABC
Postorder:	visit left -> right -> root	BCA

Inorder Traversal

- Steps of traversal:
  - Step1: Move down the tree toward the **left** until you can go no farther
  - Step2: **Visit** the node
  - Step3: Move one node to the **right** and continue step1
- Use recursion to describe this traversal

Inorder Traversal

```
template <class T>
void Tree<T>::Inorder()
{ // Start a recursive inorder traversal
  // a public member function of Tree
  Inorder(root);
}
template <class T>
void Tree<T>::Inorder(TreeNode<T>* currentNode)
{ // Recursive inorder traversal function
  // a private member function of Tree
  if(currentNode){
     Inorder(currentNode->leftChild);
     Visit(currentNode); // e.g., printout info.
     Inorder(currentNode->RightChild);
  }
}
```

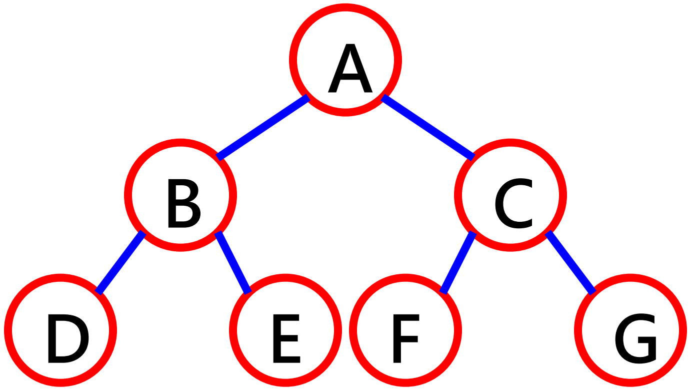
Preorder Traversal

```
template <class T>
void Tree<T>::Preorder()
{ // Start a recursive preorder traversal
  // a public member function of Tree
  Preorder(root);
}
template <class T>
void Tree<T>::Preorder(TreeNode<T>* currentNode)
{ // Recursive preorder traversal function
  // a private member function of Tree
  if(currentNode){
     Visit(currentNode); // e.g., printout info.
     Preorder(currentNode->leftChild);
     Preorder(currentNode->RightChild);
  }
}
```

Postorder Traversal

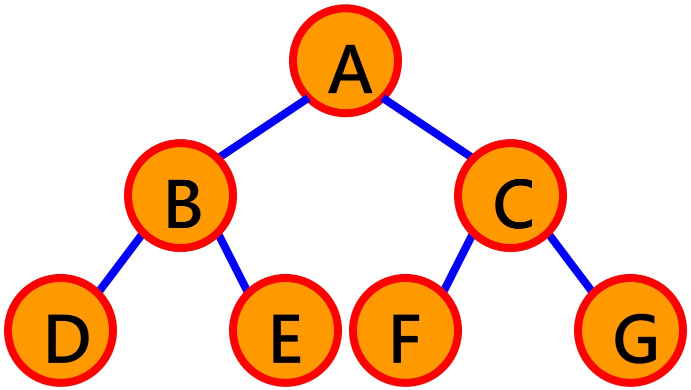
```
template <class T>
void Tree<T>::Postorder()
{ // Start a recursive postorder traversal
  // a public member function of Tree
  Postorder(root);
}
template <class T>
void Tree<T>::Postorder(TreeNode<T>* currentNode)
{ // Recursive postorder traversal function
  // a private member function of Tree
  if(currentNode){
     Postorder(currentNode->leftChild);
     Postorder(currentNode->RightChild);
     Visit(currentNode); // e.g., printout info.
  }
}
```

Running Example



Traversal	Output ordered list
Inorder	
Preorder	
Postorder	

Running Example



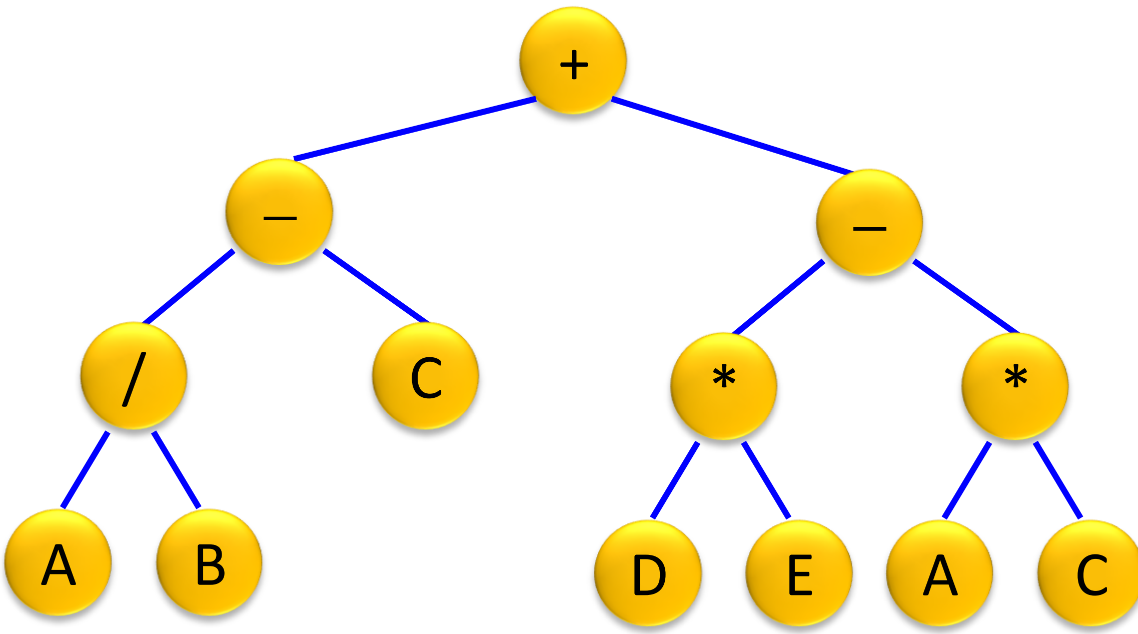
Traversal	Output ordered list
Inorder	D B E A F C G
Preorder	A B D E C F G
Postorder	D E B F G C A

Expression Evaluation and Tree Traversal

- Consider the example expression from Chapter 3

  A/B – C + D*E – A*C

- A possible tree representation is as follows:



Inorder traversal
→ infix rep.

Postorder traversal
→ postfix rep.

Tree Iterator

- We would like to visit nodes in a fashion like using **iterator** to visit elements in a container
- Recursive traversal is no long suitable
- We need an **iterative** version, but how?
  - Using stack to store non-visited nodes!

Non-Recursive Inorder Traversal

```
template <class T>
void Tree<T>::NonrecInorder()
{ // Non-recursive inorder traversal using stack
  Stack<TreeNode<T>*> s;
  TreeNode<T>* currentNode = root;
  while(1){
    while(currentNode){      // move down leftChild
       s.Push(currentNode); // add to stack
       currentNode = currentNode->leftChild; }
    if(s.IsEmpty()) return; // all nodes visited
    currentNode = s.Top();      s.Pop();
    Visit(currentNode);      // e.g. print out info.
    currentNode = currentNode->rightNode;
  }
}
```

We only need this part to develop tree iterator

Inorder Iterator

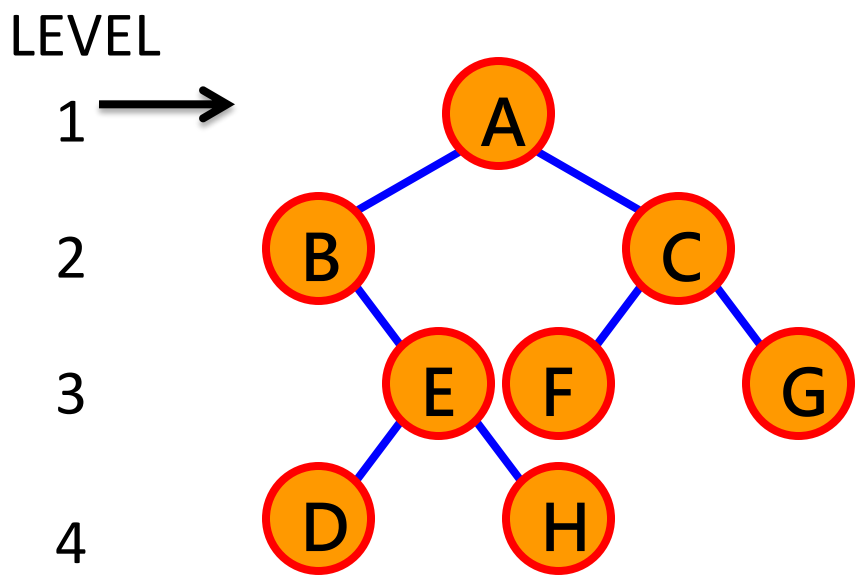
```
Template class<T> class Tree {
friend class InorderIterator;
private:
  TreeNode<T> *root;
public:
  class InorderIterator { // nested class
  public:
    InorderIterator(Tree<t> tree) :t(tree)
      {currentNode = t.root;}
    T* Next();
  private:
    Tree<T> t;
    Stack<TreeNode<T>*> s;
    TreeNode<T>* currentNode;
  };
  void inorder();
};
```

Inorder Iterator

```
template <class T> T* InorderIterator::Next() {
   while(currentNode){ // Move down leftChild
      s.Push(currentNode); // Add to stack
      currentNode = currentNode->leftChild; }
   if(s.IsEmpty()) return; // All nodes visited
   currentNode = s.Top();      s.Pop();
   T& temp = currentNode->data;
   currentNode = currentNode->rightNode;
   return &temp;
}
main() {
  Tree<char> t;
  Tree<char>::InorderIterator it(t);
  char *next = it.Next();
  while (next) { ...*next ...; next = it.Next();}
}
```

Level-Order Traversal

- Visit nodes in a top down, left to right manner



A B C E F G D H

Preorder	Inorder	Postorder	Level-
Stack	Stack	Stack	Queue

Level-Order Traversal

```
template <class T>
void Tree<T>::LevelOrder()
{ // Traverse the binary tree in level order
  Queue<TreeNode<T>*> q;
  TreeNode<T>* currentNode = root;
  while(currentNode){
    Visit(currentNode);
    if(currentNode->leftChild)
           q.Push(currentNode->leftChild);
    if(currentNode->rightChild)
           q.Push(currentNode->rightChild);
    if(q.IsEmpty()) return;
    currentNode = q.Front();    q.Pop();
  }
}
```

Summary

- Trees, terminologies of trees, tree representation
- Binary trees, properties of binary trees, array and linked representation
- Binary tree traversal: inorder, preorder, postorder
- Binary tree iterators
- Self-study topics
  - Binary tree operations: copying binary trees, testing equality