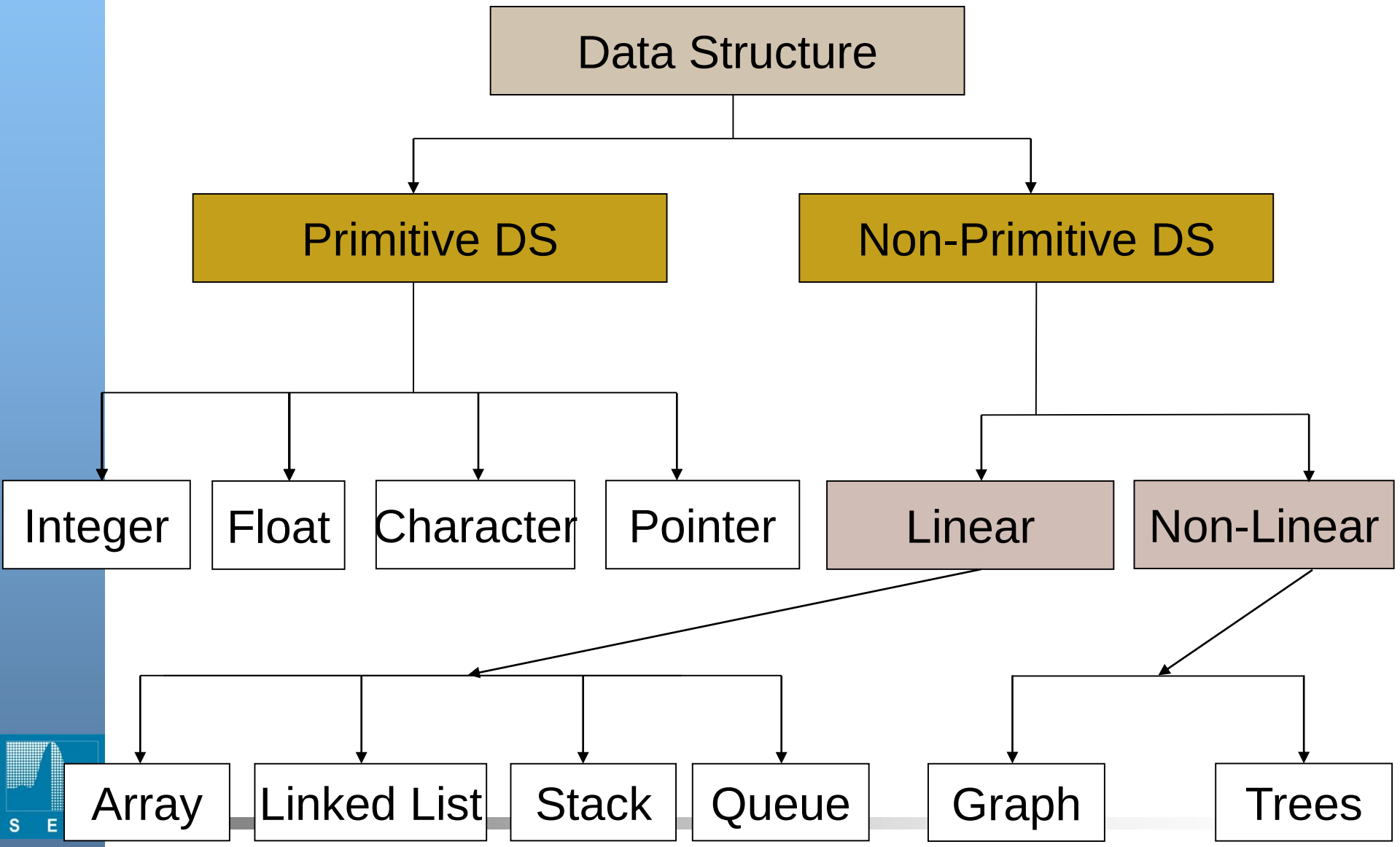
Data structure and Algorithms
Trees

Thanh-Hai Tran

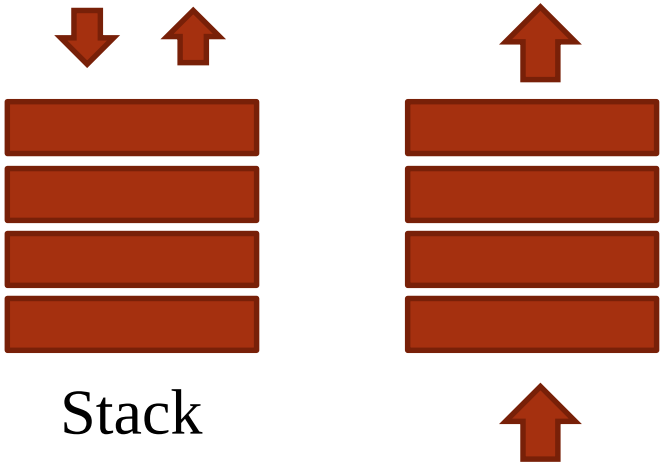
Electronics and Computer Engineering
School of Electronics and Telecommunications

Hanoi University of Science and Technology
1 Dai Co Viet - Hanoi - Vietnam

Data structures



Linear data structures



How to choose the best DS ?

What need to be stored ?
Cost of operations ?
Memory usage
Easy implementation

Introduction

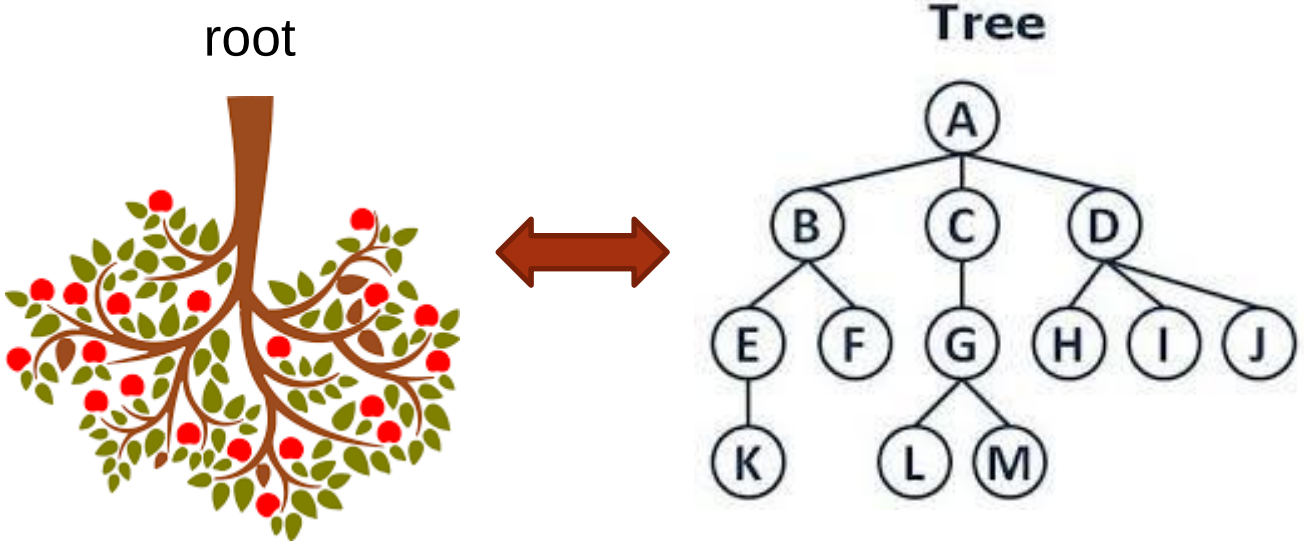
- Tree is one of the most important nonlinear data structures in computing, a breakthrough in data organization
- Trees also provide a natural organization for data
- They have become ubiquitous structures in file systems, graphical user interfaces, databases, Web sites, and other computer systems
- Non-linear relationship
  - Not “before” and “after”
  - But: **hierarchical** with “above” and “below”

Outline

- **General trees**
  - Tree Definitions and Properties
  - Tree Functions
- **Tree Traversal Algorithms**
  - Depth and Height
  - Preorder Traversal
  - Postorder Traversal
- **Binary Trees**
  - Introduction to Binary Tree
  - Properties of Binary Trees
  - A Linked Structure for Binary Trees
  - Traversals of a Binary Tree
  - Representing General Trees with Binary Trees

Tree definition

- A **tree** is an abstract data type that stores elements hierarchically
- Each element in a tree has
  - a parent element and
  - zero or more children elements
- The top element called the **root** of the tree

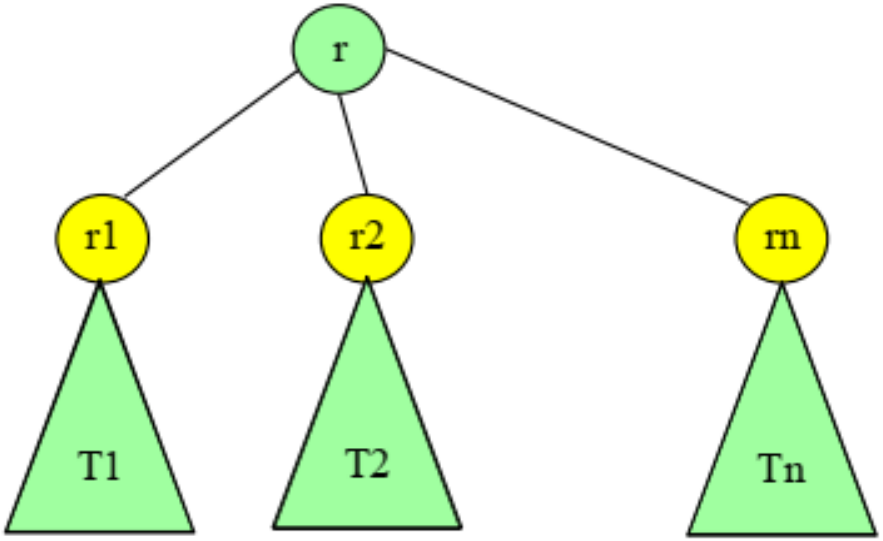


Formal definition of tree

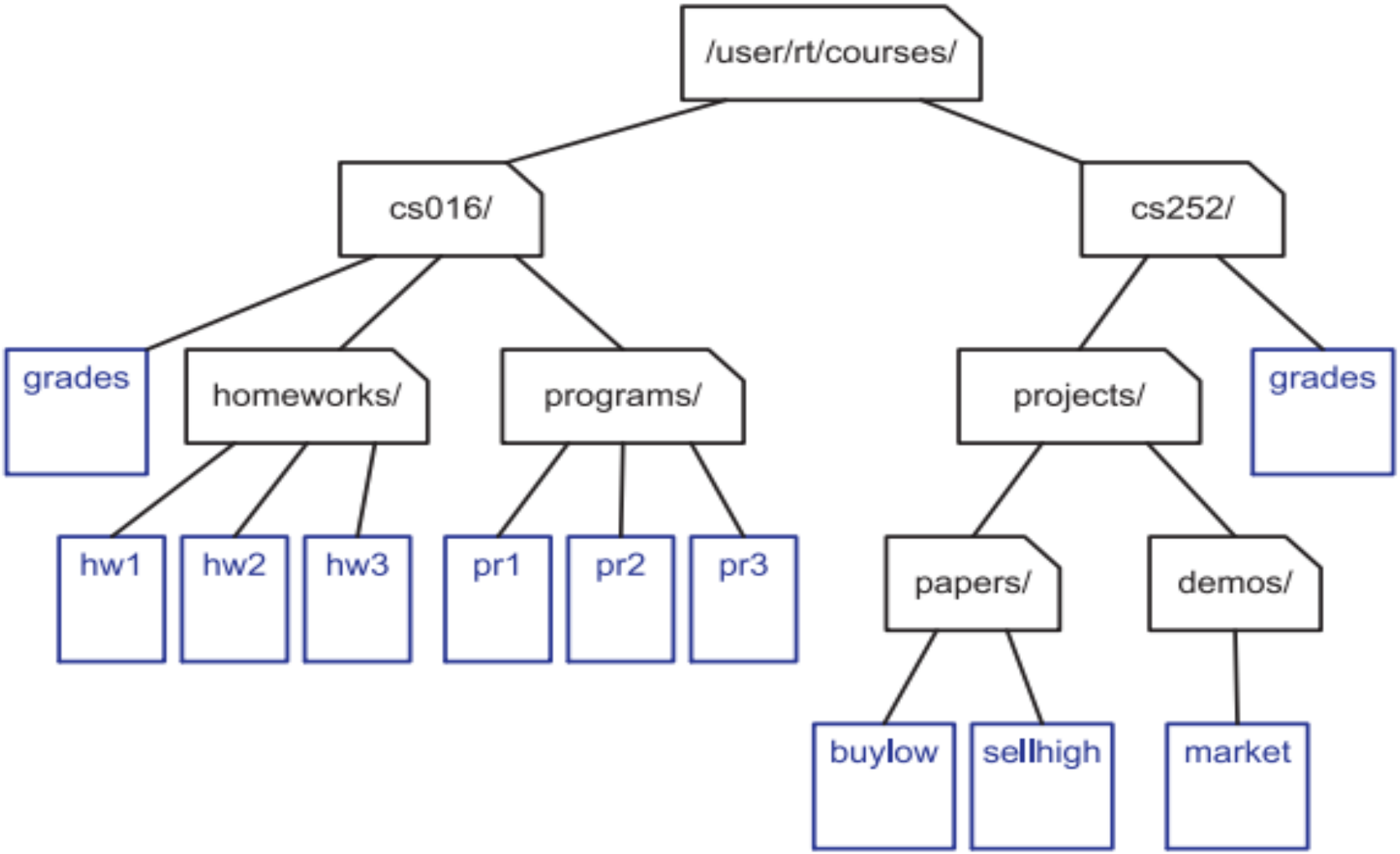
- **A tree T is a set of nodes storing elements in parent-child relationship with following properties:**
  - If T is nonempty, it has a special node, called the **root** of T, has no parent
  - Each node v of T different from the root has a unique **parent** node w; every node with parent w is a **child** of w

Recursive definition of tree

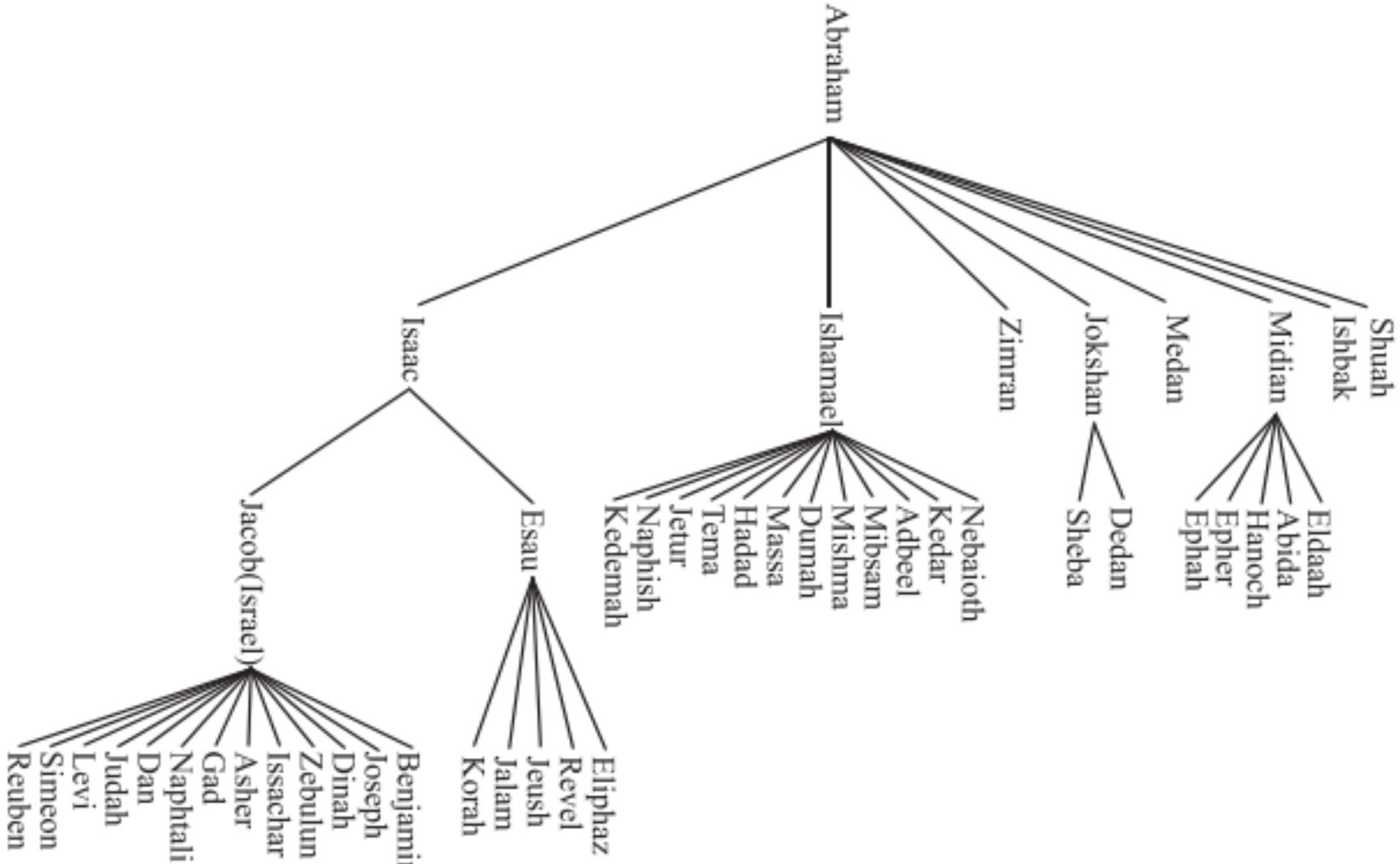
- **A recursive definition of tree**
  - A node creates a tree (root)
  - When there are n trees T_1, T_2, …, T_n
    - Each tree has corresponding roots: r_1, r_2, …, r_n
    - r has parent-child relationship with r_1, r_2, …, r_n
    - Then exist a tree T with r is a root



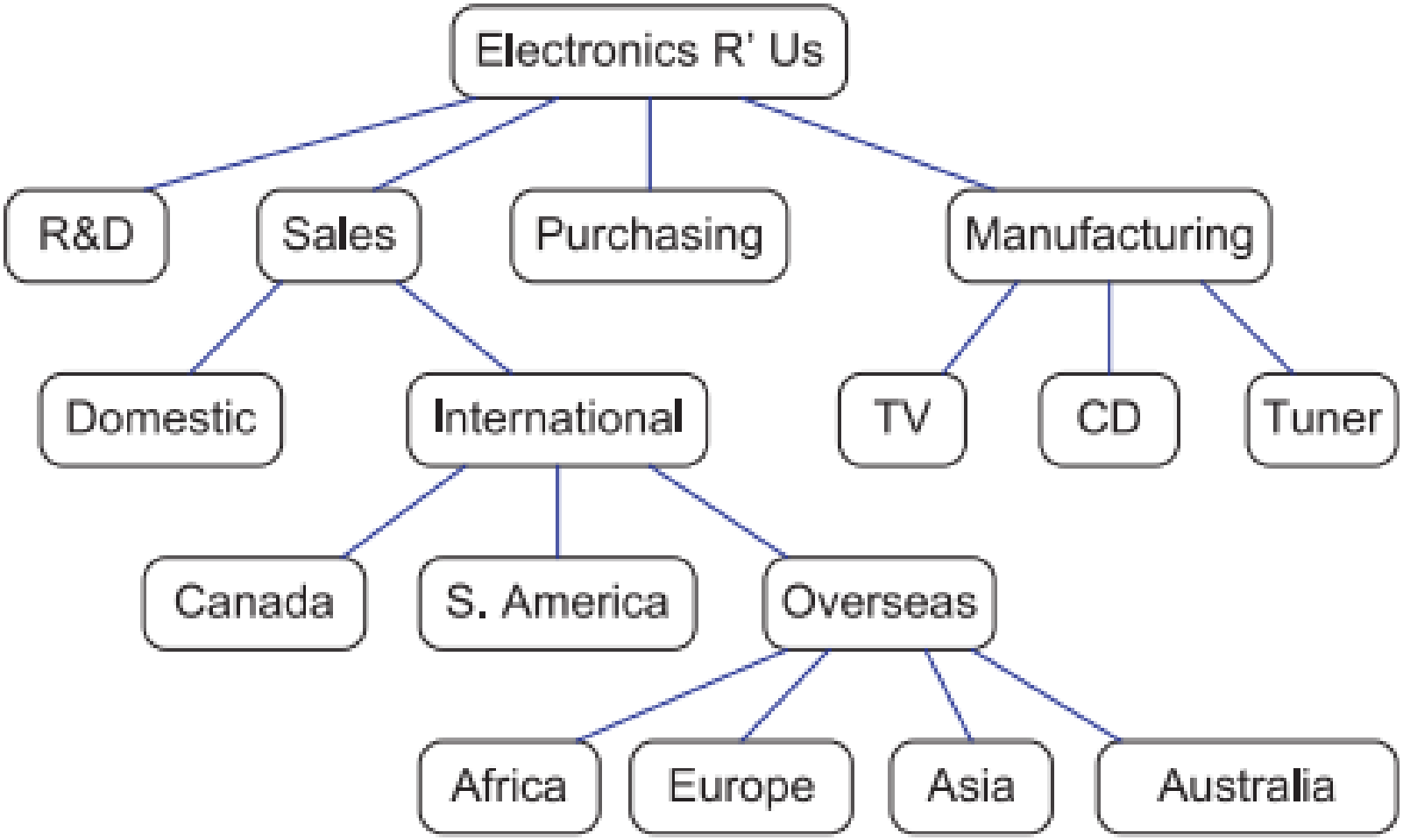
Examples of tree (1)



Examples of tree (2)

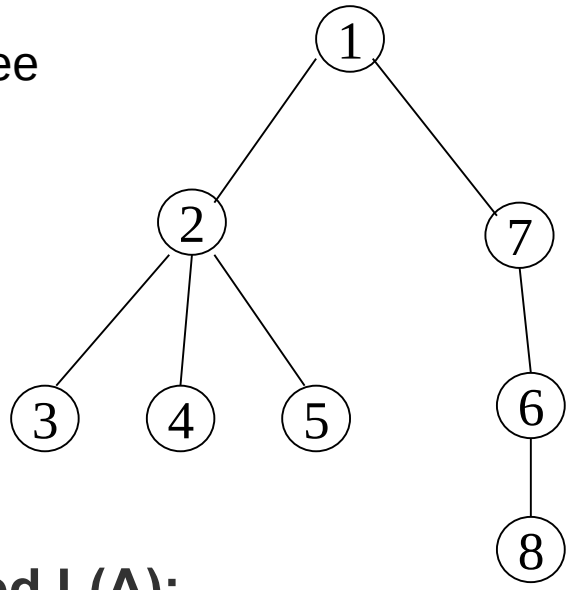


Examples of tree (3)



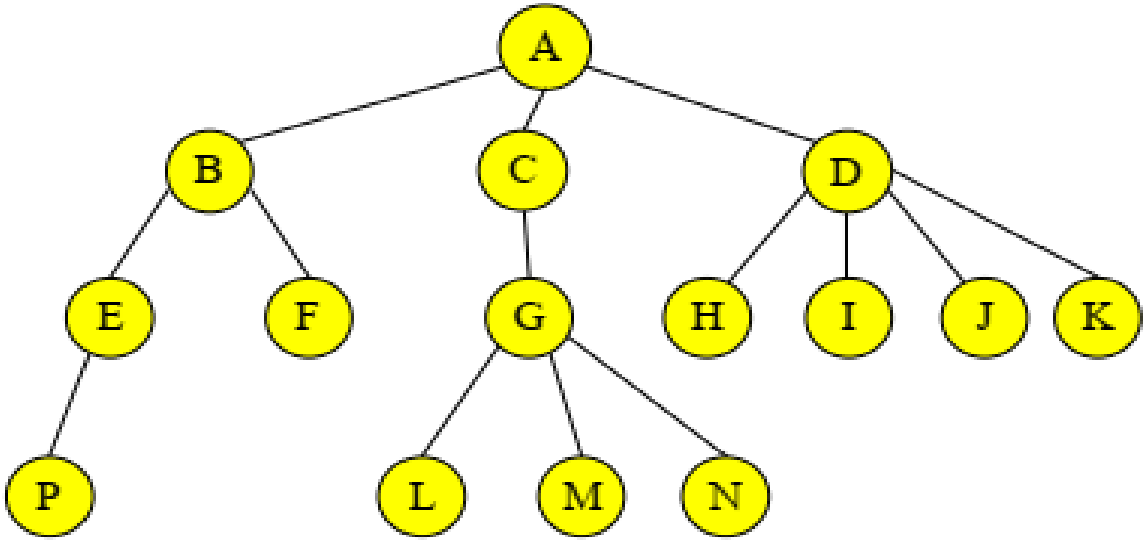
Basic concepts (1)

- **Size of tree (Kích thước cây): number of nodes in tree, noted S(T)**
- **Degree of node (Cấp của một nút): number of children the node has, noted d(A)**
- **Degree of tree (Cấp của cây): d(T)**
  - degree of the node having maximal degree
- **Branch node (Nút nhánh):**
  - the node having at least one child
  - also called internal node (nút trong), non-terminal (nút không tận cùng)
- **Leaf node (Nút lá):**
  - the node has no children
  - also called: external (nút ngoài), terminal
- **Level of node (Mức của một nút), noted L(A):**
  - If node A is a root then L(A) = 1
  - If node B is father of node A then L(A) = L(B) +1



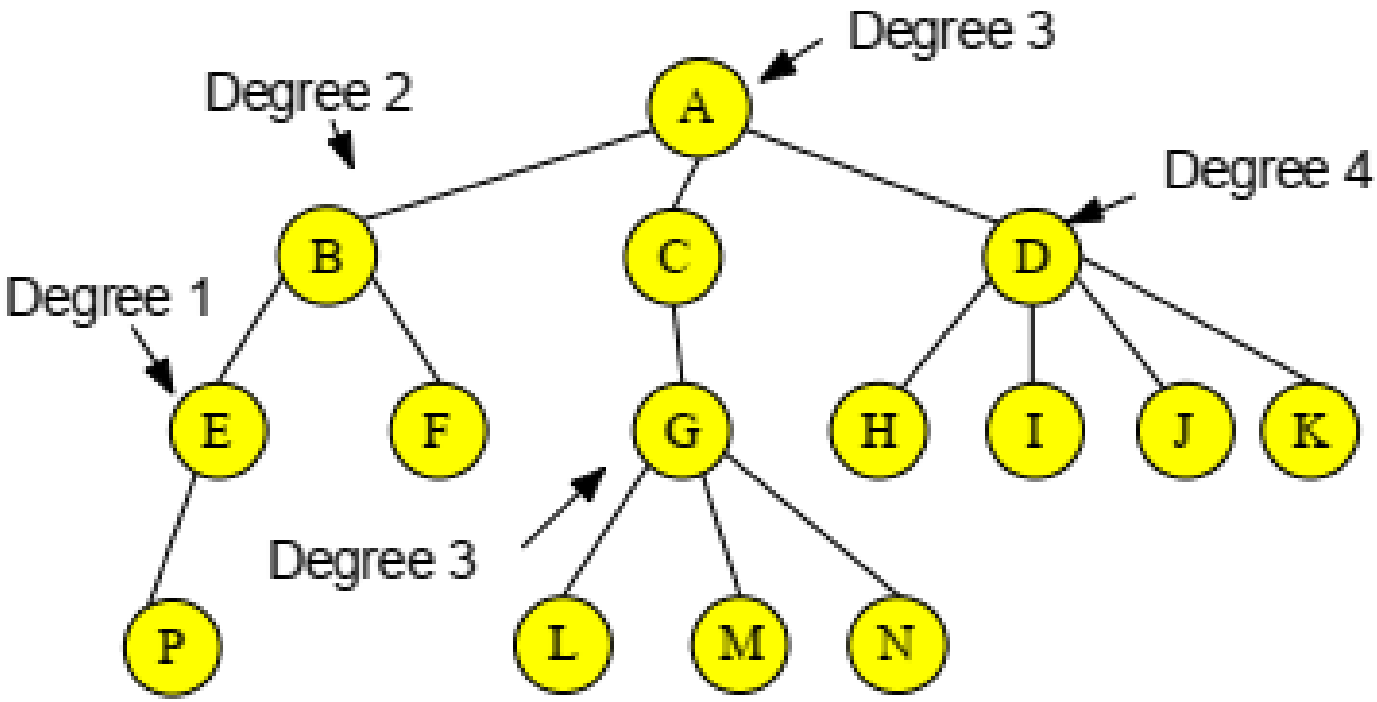
Basic concepts (2)

- **Siblings:** two nodes that are children of the same parent (B, C, D)
- **External node (leaf)**: has no children (P, F, L, M, N, H, I, J, K)
- **Internal node**: has one or more children (B, E, C, G, D)
- **Ancestor:** A, C G are ancestors of M
- **Descendent:** E, F, G, H, L,M …are descendents of A



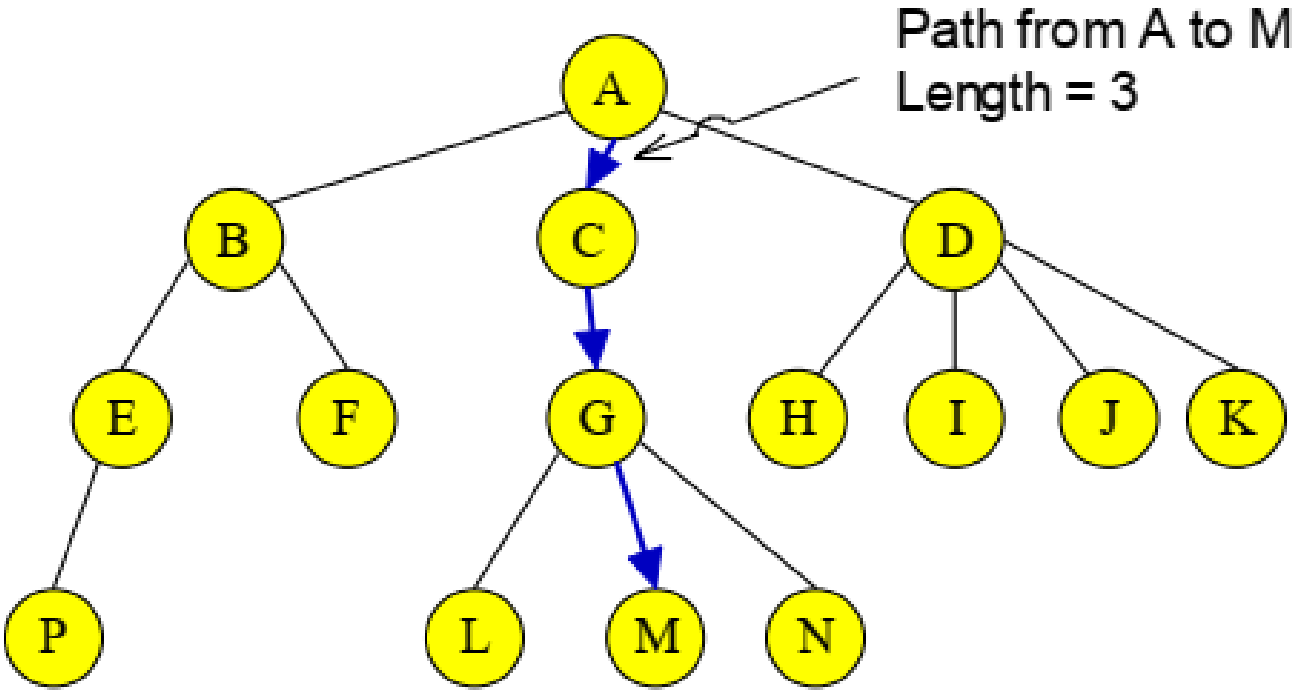
Basic concepts (3)

- **Degree of a node:** number of children of that node
- **Degree of the tree:** the maximal degree of a nodes in the tree (e.g. the following tree has degree 4)



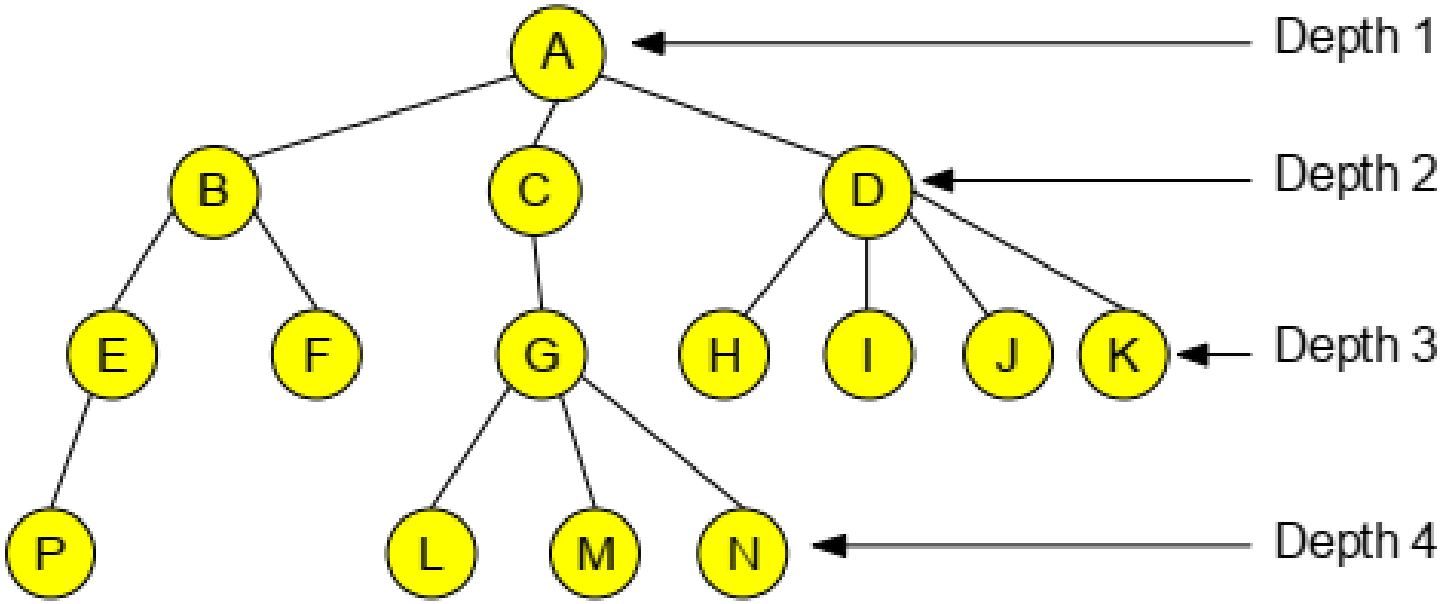
Basic concepts (4)

- **Path of tree:**
  - A sequence of nodes: n_1, n_2, …, n_k
  - n_i is parent of n_{i+1} (i = 1..k-1)



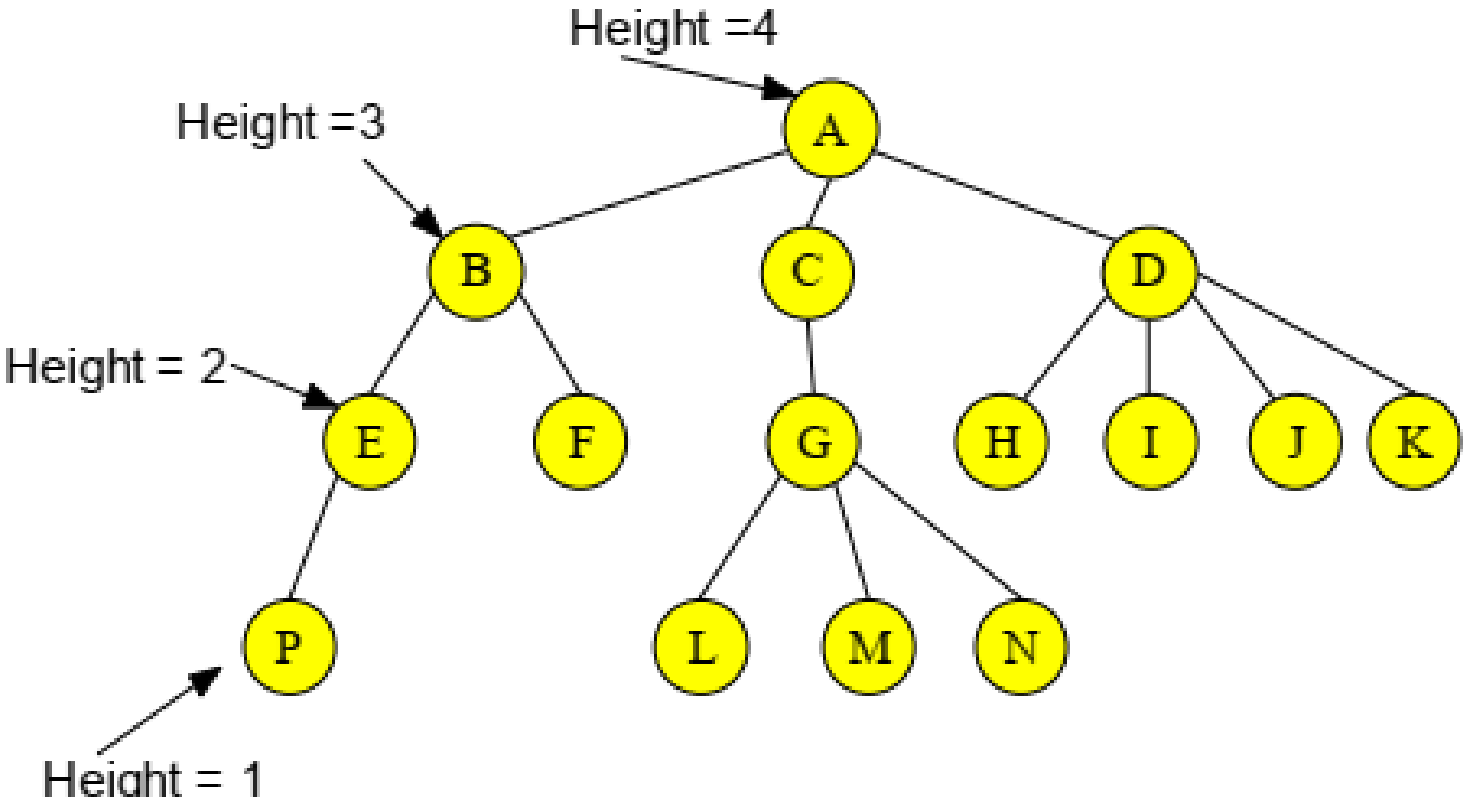
Basic concepts (5)

- **Depth of a node :**
  - Is the length of the path going from the root to that node + 1
  - Example: given a root of tree r, the node r_i: $d(r_i)$ = length (r, r_i) +1



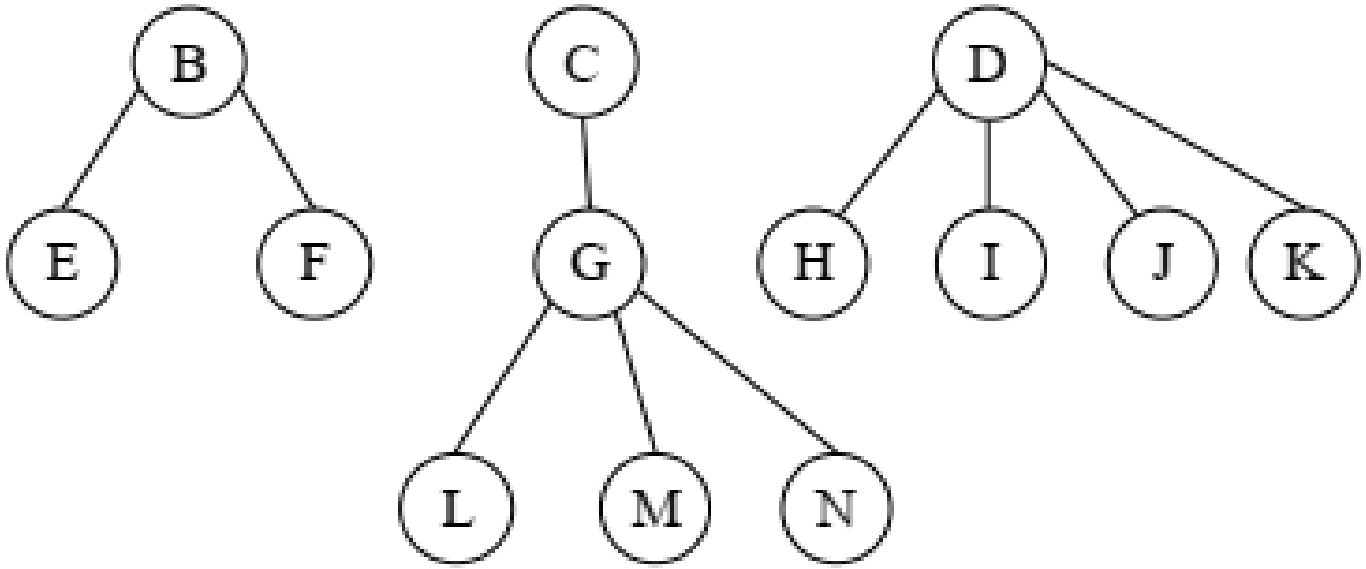
Basic concepts (6)

- **Height of a node:** The longest path from this node to a leaf in the tree + 1
- **Height of the tree:** the height of root node



Basic concepts (7)

- **Forest: the structure having one or more trees**



Illustrative example

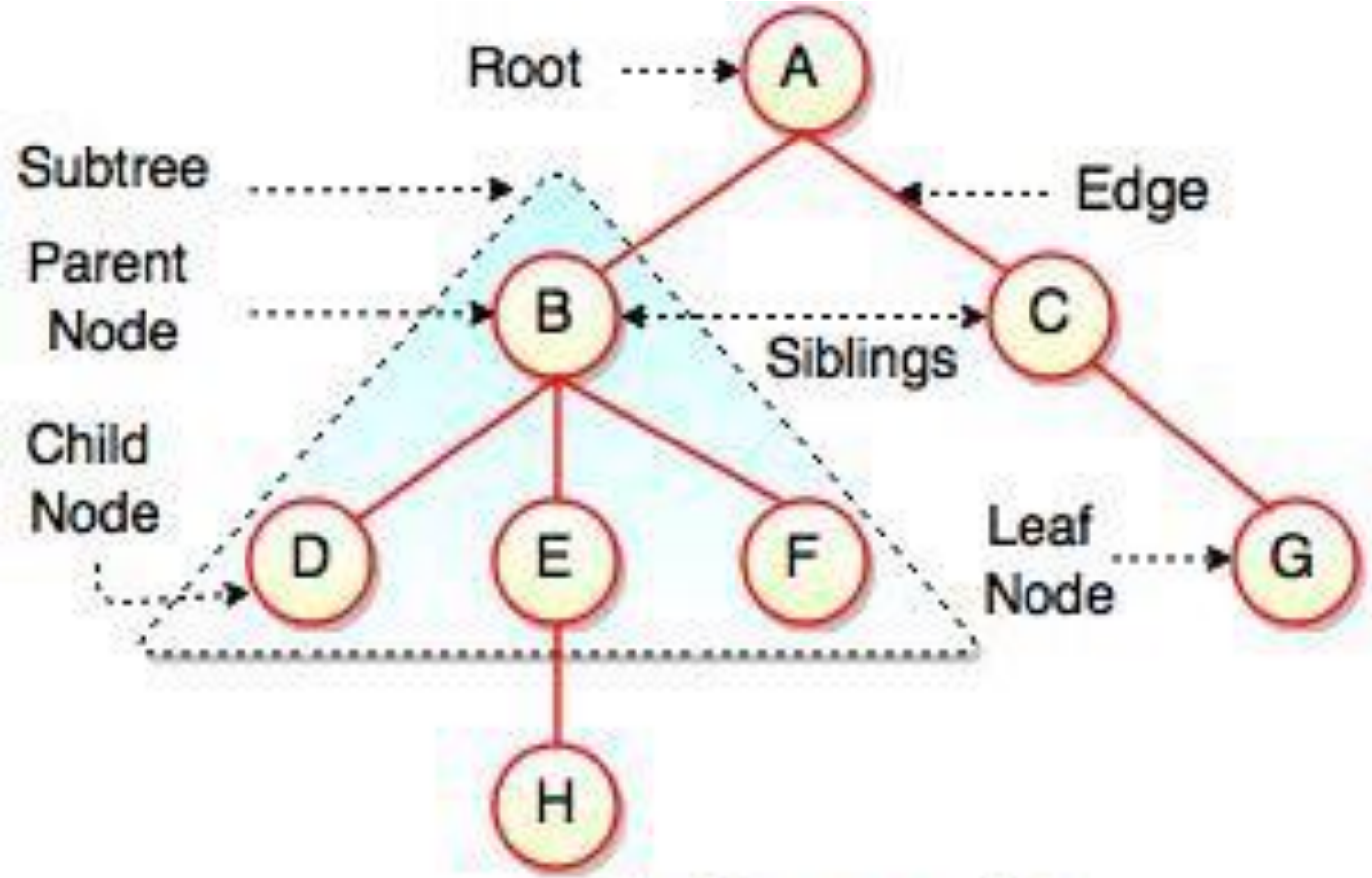


Fig. Structure of Tree

Tree properties

- **Size of the tree (Kích thước của cây): sum of branches + 1** (là tổng số nhánh + 1)
- **Recursion (Cây có tính chất đệ quy): a tree is made from multiple sub-trees** (một cây được tạo bởi nhiều cây con)
- **Dynamic structure (Cấu trúc dữ liệu động): the size of tree is variable** (Một cây có kích thước biến đổi)
- **Non-linear structure (Cấu trúc cây là cấu trúc phi tuyến): Hierachical** (phân cấp)
- **There exist an unique path from the root to other nodes in the tree** (Chỉ tồn tại duy nhất một đường đi từ nút gốc đến nút khác)

Basic tree operations

- **Initialize a tree**:
  - Declaration of a suitable data structure to store the tree
  - Initialize the tree as empty
- **Insert a new node into the tree:**
  - Determine the position to insert the new node into the tree
  - Determine the relationship between the new node and the existing node
- **Remove a node from tree:**
  - Determine the position of the node to be removed
  - Reorganize the tree

Basic tree operations

- **Access operations:**
  - **root()** : returns the root node
  - **parent(Tree T, Node p)**: returns the parent of node p in the tree T
  - **children(Tree T, Node p)**: returns the children of the node p in the tree T
  - **left_most_child(Tree T, Node p)** : returns the left most child of the node p in the tree T
  - **right_most_child(Tree T, Node p)** : returns the right most child of the node p in the tree T
  - **left_sibling (Tree T, Node p)** : returns the left sibling of a node p in the tree T
  - **right_sibling(Tree T, Node p)** : returns the right sibling of a node p in the tree T

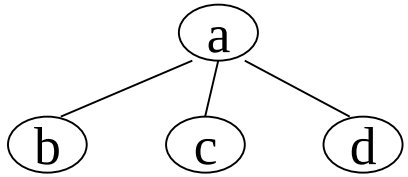
Basic tree operations

- **Other operations:**
  - **height (Tree T):** returns height of the tree T
  - **size(Tree T)**: return size of the tree
  - **isRoot (Tree T, Node p);** check if a node p is a root node
  - **isLeaf (Tree T, Node p);** check if a node p is a leaf
  - **isInternal (Tree T, Node p);** check if a node p is an internal node

Basic tree operations

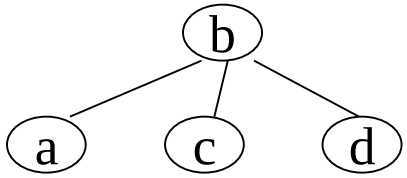
- **Traversal of tree (duyệt cây): visit all nodes of tree, only once for each node**

- **Preorder** (duyệt trước): 1, 2, 4, 3, 5, 7, 6, 8, 9



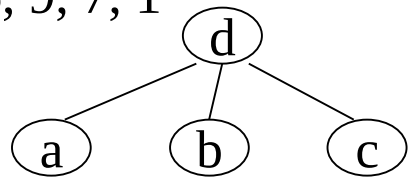
=> a, b, c, d

- **Inorder** (duyệt giữa): 4, 2, 3, 5, 1, 8, 6, 7, 9 : used for binary trees

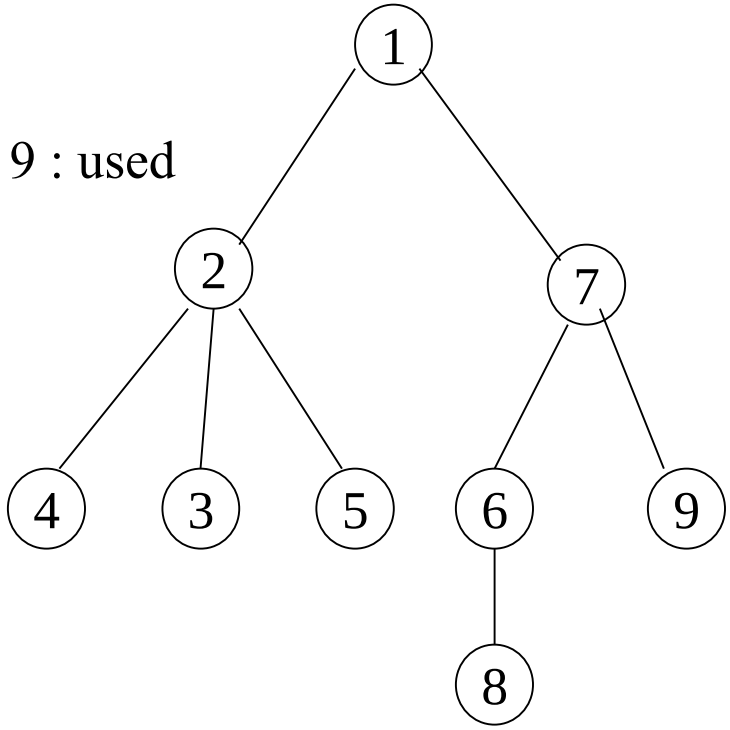


=> a, b, c, d

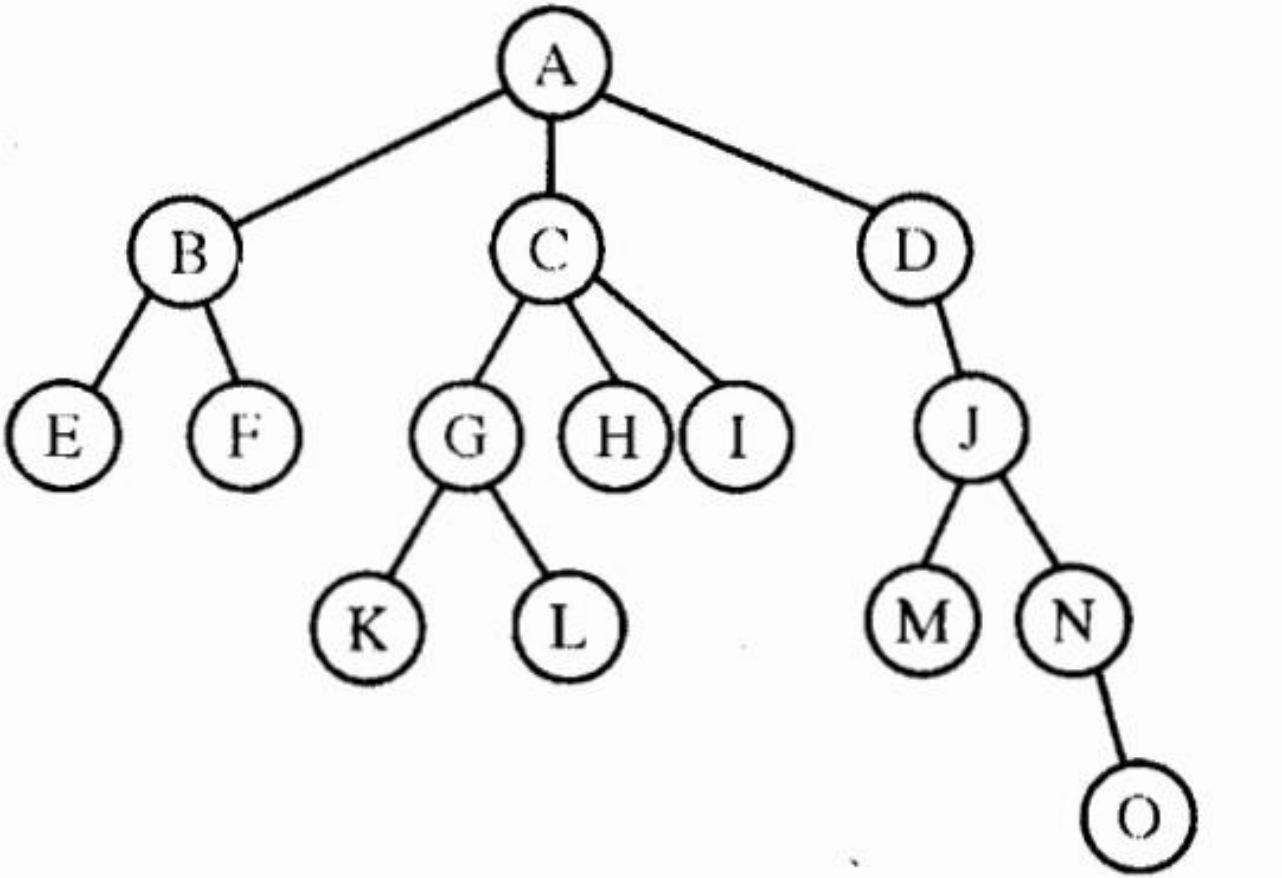
- **Postorder** (duyệt sau): 4, 3, 5, 2, 8, 6, 9, 7, 1



=> a, b, c, d

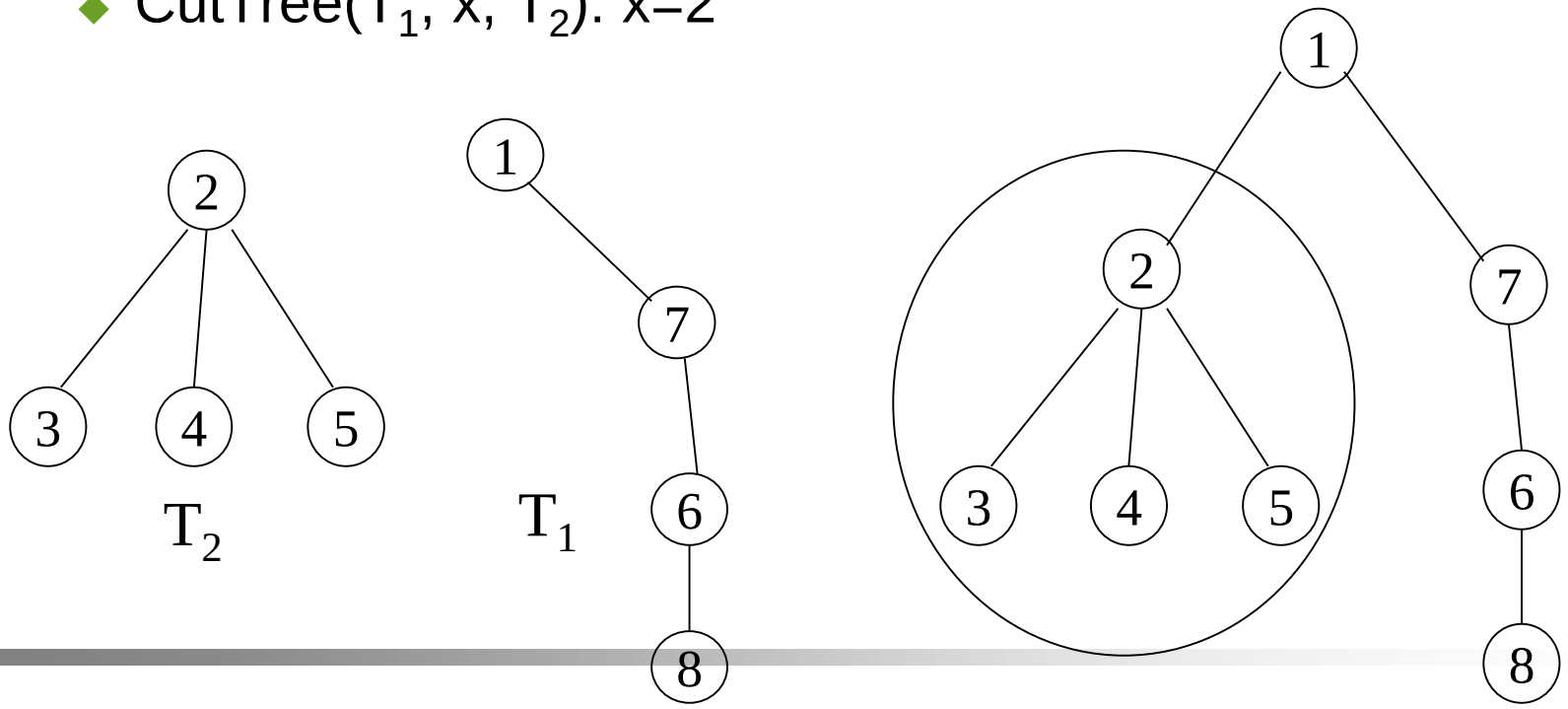


Tree traversal example



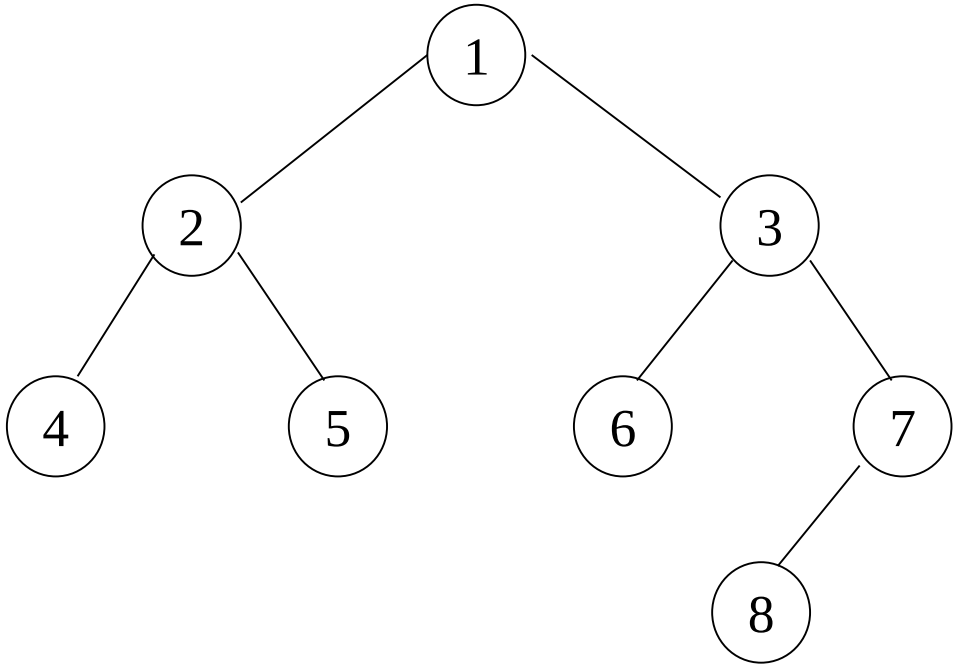
Basic tree operations

- **Tree Merge: merge two sub-trees to become a bigger one**
  - MergeTree(T_1,T_2, x): x=1
  - Role of x node: merged node, root of T_2 will be child of x
- **Tree Cut: cut a given tree into 2 sub-trees**
  - CutTree(T_1, x, T_2): x=2



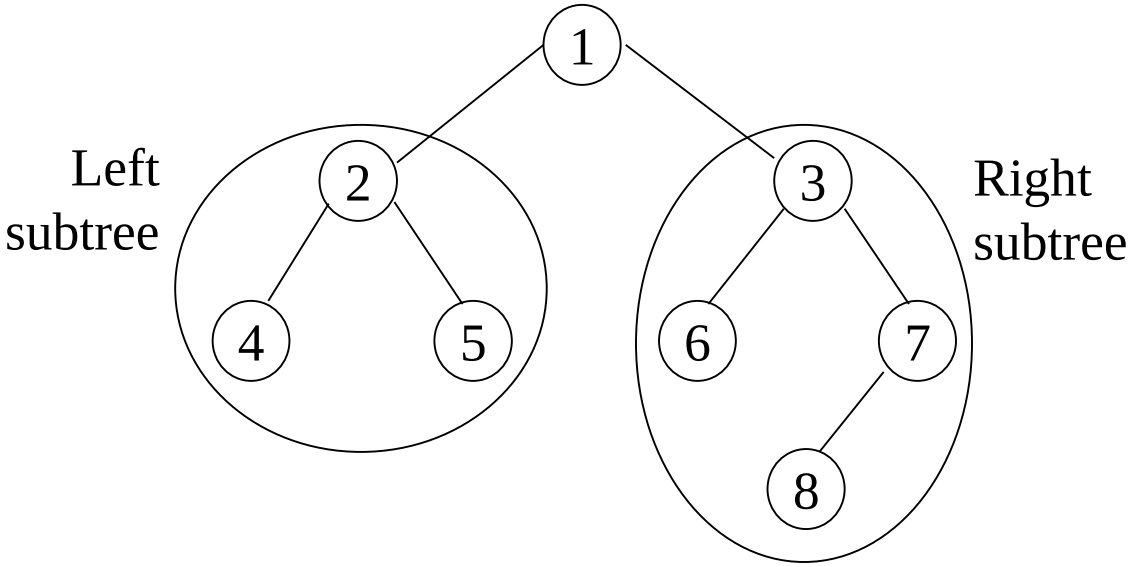
Binary trees (Cây nhị phân)

- **Description of binary trees:**
  - A binary tree is ordered and degree of 2 (each node has at most 2 children).
  - With 2 children of a node, one is *left child*, the other is *right child*, that should be shown differently.



Binary trees (Cây nhị phân)

- **Basic concepts**
  - Types of nodes (kiểu nút):
    - *Double node* (nút kép): having 2 children (1, 2, 3)
    - *Single node* (nút đơn): having only one child (7)
    - *Leaf* (nút lá): having no children (4, 5, 6, 8)
  - *Left/Right sub-trees* (cây con trái/phải):

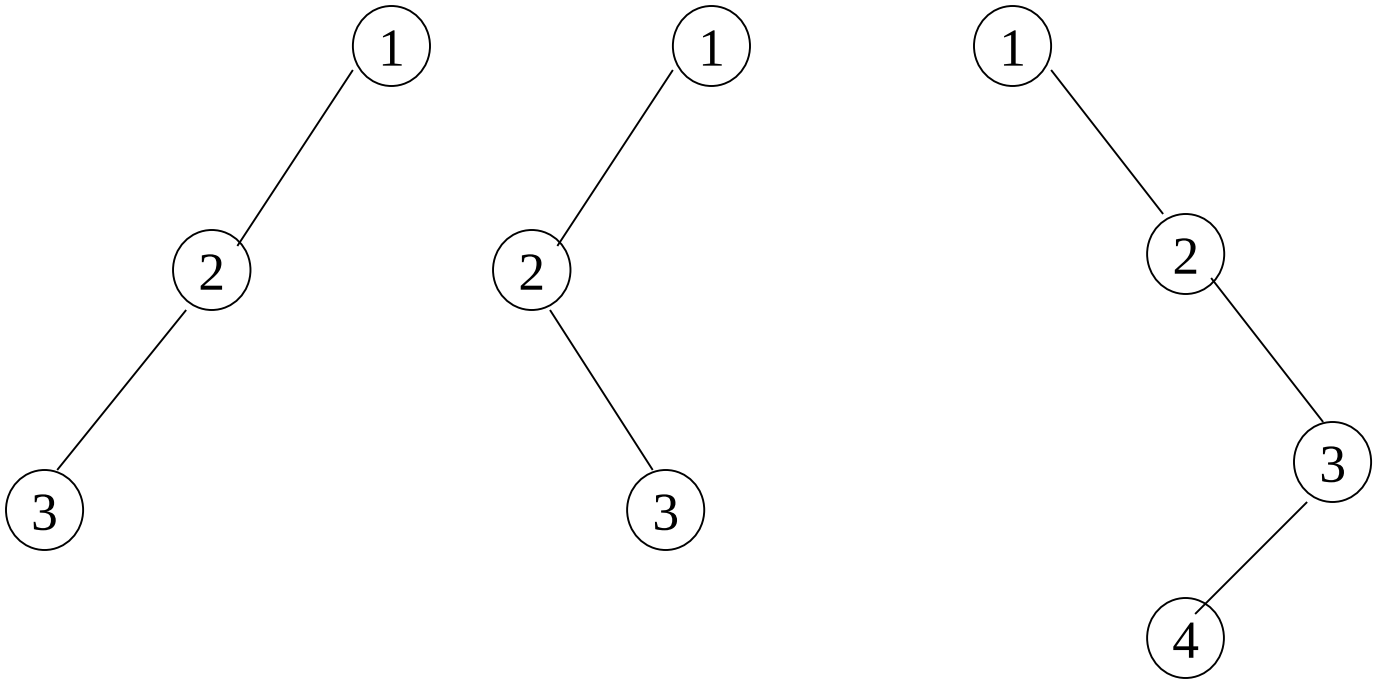


Binary trees (Cây nhị phân)

- **Basic operations:**
  - Similar to operations of generalized trees as:
    - Initialize (Khởi tạo):
    - Insert a new node (Bổ sung một nút mới vào cây)
    - Remove a node (Lấy ra một nút)
    - Find father of a given node (Tìm cha mỗi đỉnh): Parent(x)
    - Find left child (Tìm con bên trái của mỗi đỉnh): LeftChild(x)
    - Find right child (Tìm con bên phải của mỗi đỉnh): RightChild(x)
    - Tree merge (Ghép cây)
    - Tree cut (Cắt cây)
    - Traversal (Duyệt cây)

Special cases of binary trees

- **Degenerate binary tree (Cây suy biến)**
  - The binary tree that every node has at most one child → degenerated to list.
  - In other words, list is only a special case of tree.

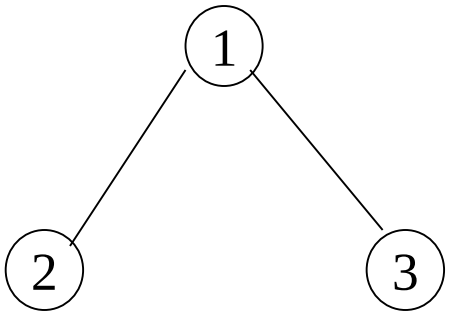


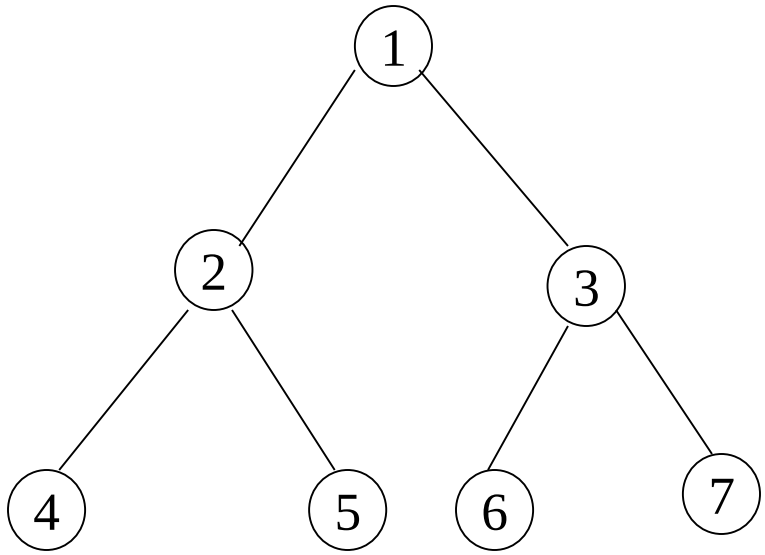
Special cases of binary trees

- **Full binary tree (Cây đầy đủ)**
  - The binary tree that has maximal number of nodes at all levels. It means that the tree has 2^{i-1} nodes at level i, or its size is:

$$S(T) = 2^0+2^1+\ldots+2^h = 2^{h+1}-1$$

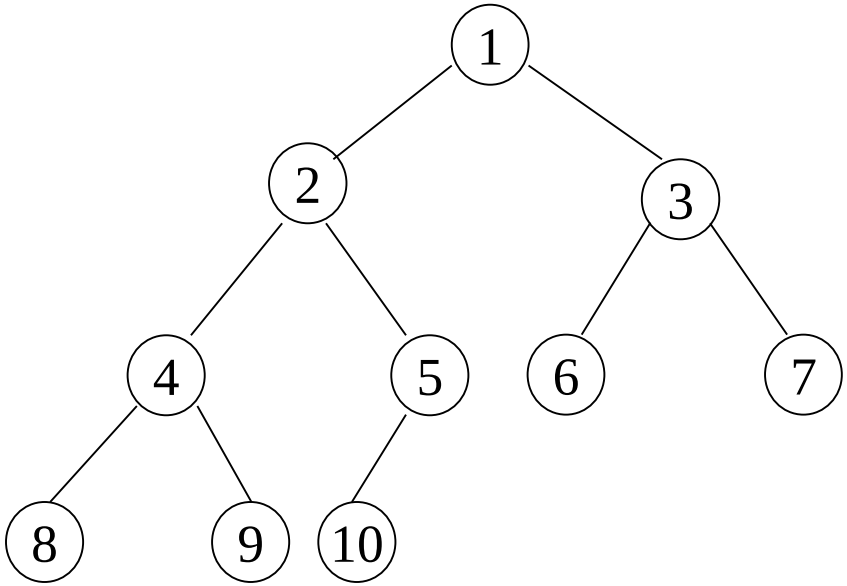
with h is its height





Special cases of binary trees

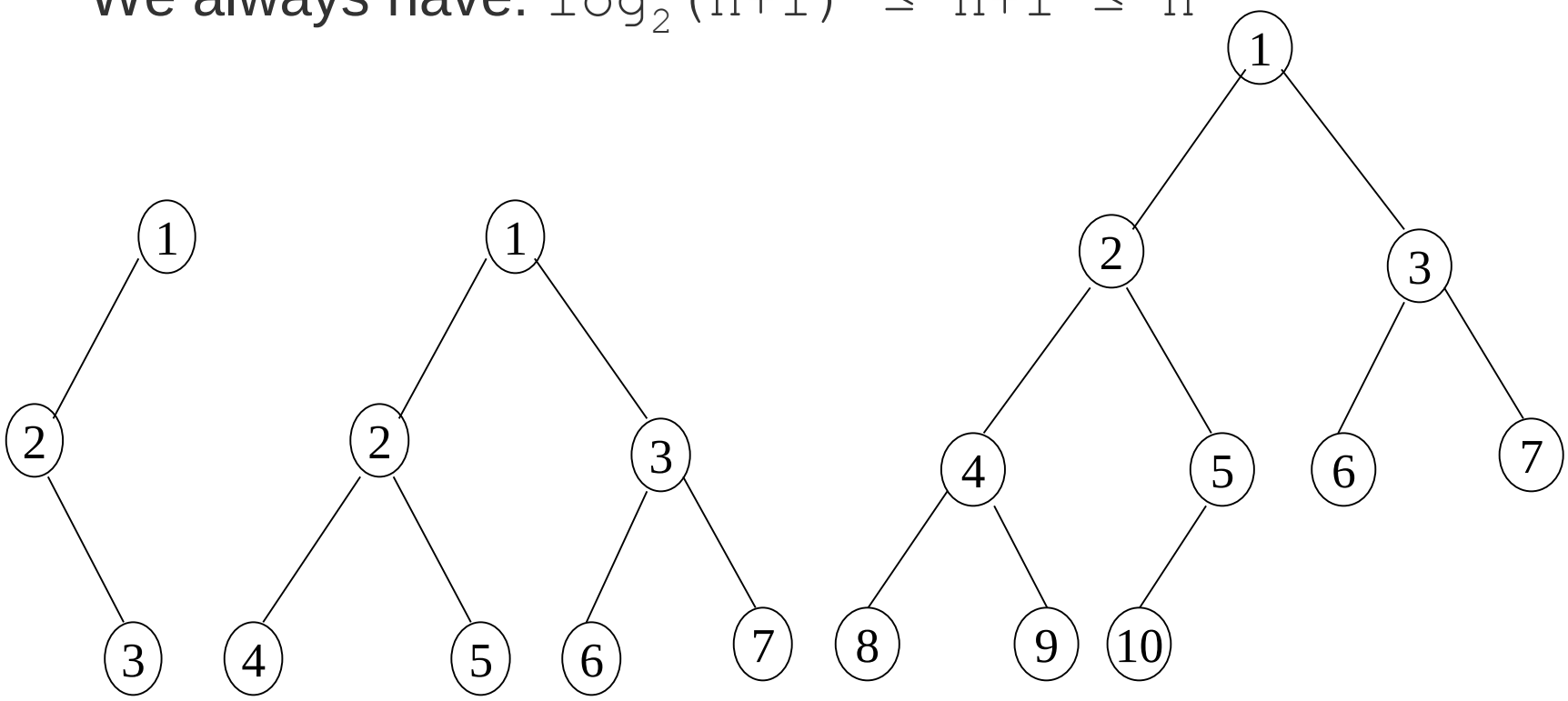
- **Complete binary tree (Cây hoàn chỉnh)**
  - The binary tree T, with height of h, is called *complete* if:
    - It is a full binary tree except the biggest level (till the height of h-1)
    - All nodes at level h are left most (Các nút có chiều cao h thì dồn hết về bên trái).



Properties of binary trees

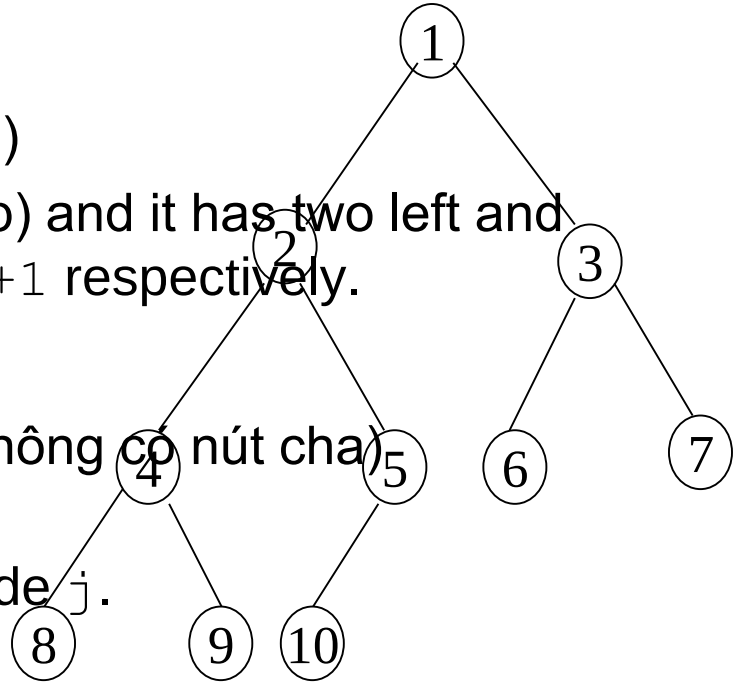
- **Recursive (Tính đệ quy):** Cutting at any branch of a binary tree will create 2 sub binary trees.
- **Given a binary tree with height of `h` and size of `n`.**

  We always have: $\log_2(n+1) \leq h+1 \leq n$



Properties of binary trees

- **For complete and full binary trees:** given a tree of size `N`. If all the nodes have been numbered as shown below, we have:
- **At node numbered `i`:**
  - if `2i > N`: node `i` is a leaf (nút lá)
  - if `2i = N`: node `i` is single (nút đơn)
  - if `2i < N`: node `i` is double (nút kép) and it has two left and right children are nodes `2i` and `2i+1` respectively.
- **At node numbered `j`:**
  - if `j = 1`: node `j` is root (nút gốc, không có nút cha)
  - if `j > 1`: node `j/2` (if `j` is even) or `(j-1)/2` (if `j` is odd) is father of node `j`.



Basic operations of binary tree

- **Traversal:**
  - Visit a node (Phép thăm một nút): access to a node of the tree.
  - Traversal (Phép duyệt): visit all nodes of the tree, each node once and only once (phép thăm một cách hệ thống tất cả các nút của cây, mỗi nút đúng một lần).

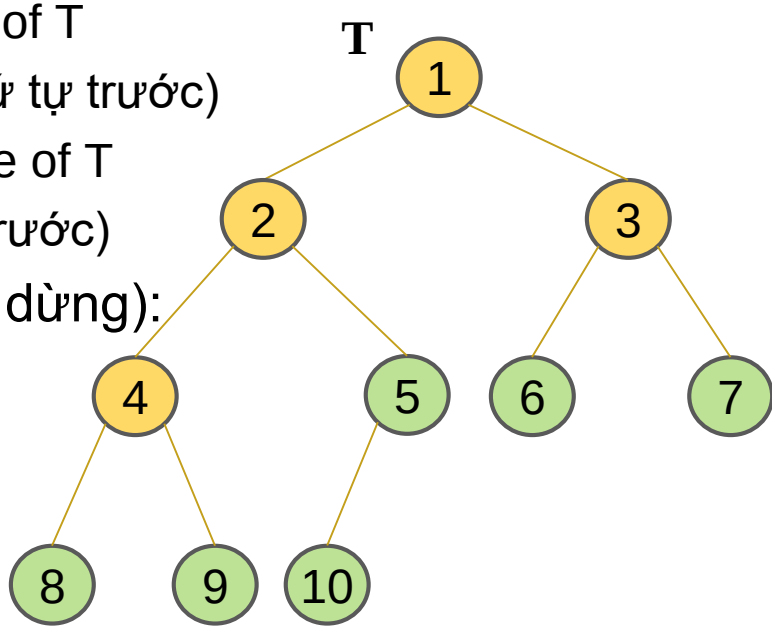
Basic operations of binary tree

- **Traversal methods: 2 approaches:**
  - First approach: based on recursive property of trees:
    - A binary tree consists of 3 parts: root, left sub-tree and right sub-tree. They are all binary trees. Following this property, there are 3 traversal methods:
      - Preorder traversal (Duyệt theo thứ tự trước)
      - Inorder traversal (Duyệt theo thứ tự giữa)
      - Postorder traversal (Duyệt theo thứ tự sau)
  - Second approach:
    - Transform a tree into a list before traversal. Then list traversal is applied.
    - Following this approach, a list will be used to store all nodes of tree.
    - Using different types of lists (queue, stack,…) create different traversal orders.

Traversal methods – 1st approach

- **Preorder traversal method (duyệt theo thứ tự trước): this is a recursive algorithm for traversal of binary tree T with two cases:**
  - *Recursive case* (Trường hợp đệ quy): following steps are used to traverse T:
    - Visit root of T (Thăm gốc)
    - Preorder traversal of left sub-tree of T (Duyệt cây con trái của T theo thứ tự trước)
    - Preorder traversal of right sub-tree of T (Duyệt cây con phải theo thứ tự trước)
  - *Stopping case* (Trường hợp điểm dừng):
    - When T is empty.

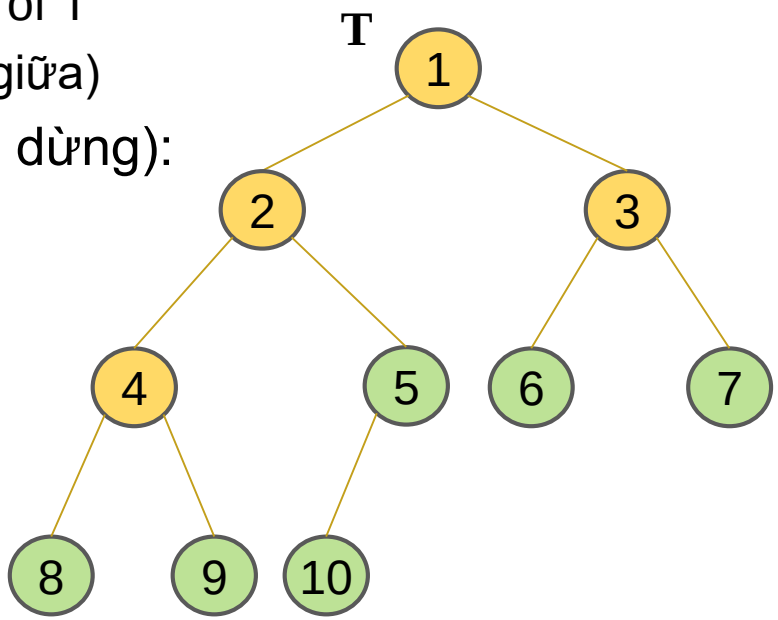
Exp: 1, 2, 4, 8, 9, 5, 10, 3, 6, 7



Traversal methods – 1st approach

- **Inorder travelsal method (duyệt theo thứ tự giữa):**
  - *Recursive case* (Trường hợp đệ quy): following steps are used to traverse T:
    - Inorder traversal of left sub-tree of T
      (Duyệt cây con trái của T theo thứ tự giữa)
    - Visit root of T (Thăm gốc)
    - Inorder traversal of right sub-tree of T
      (Duyệt cây con phải theo thứ tự giữa)
  - *Stopping case* (Trường hợp điểm dừng):
    - When T is empty.

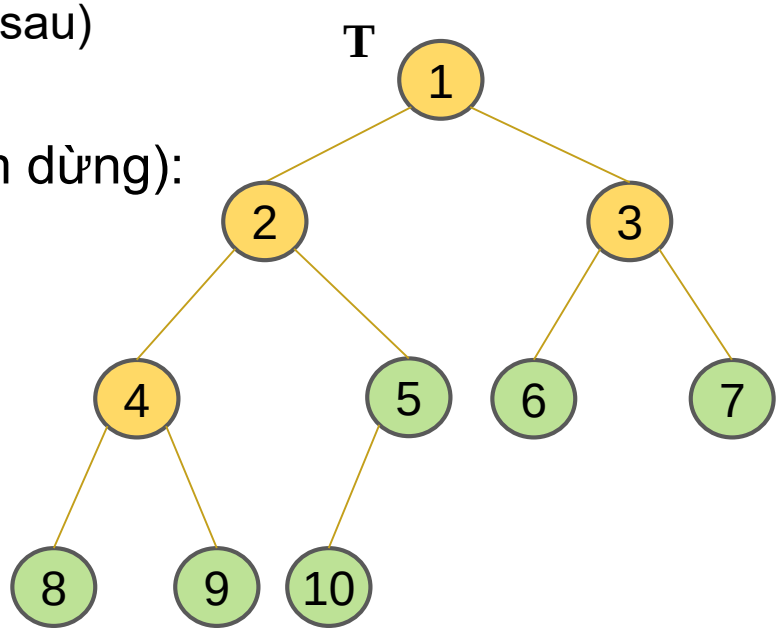
Exp: 8, 4, 9, 2, 10, 5, 1, 6, 3, 7



Traversal methods – 1st approach

- **Postorder travelsal method (duyệt theo thứ tự sau):**
  - *Recursive case* (Trường hợp đệ quy): following steps are used to traverse T:
    - Postorder traversal of left sub-tree of T
      (Duyệt cây con trái của T theo thứ tự sau)
    - Postorder traversal of right sub-tree of T
      (Duyệt cây con phải theo thứ tự sau)
    - Visit root of T (Thăm gốc)
  - *Stopping case* (Trường hợp điểm dừng):
    - When T is empty.

VD: 8, 9, 4, 10, 5, 2, 6, 7, 3, 1



Traversal methods – 2nd approach

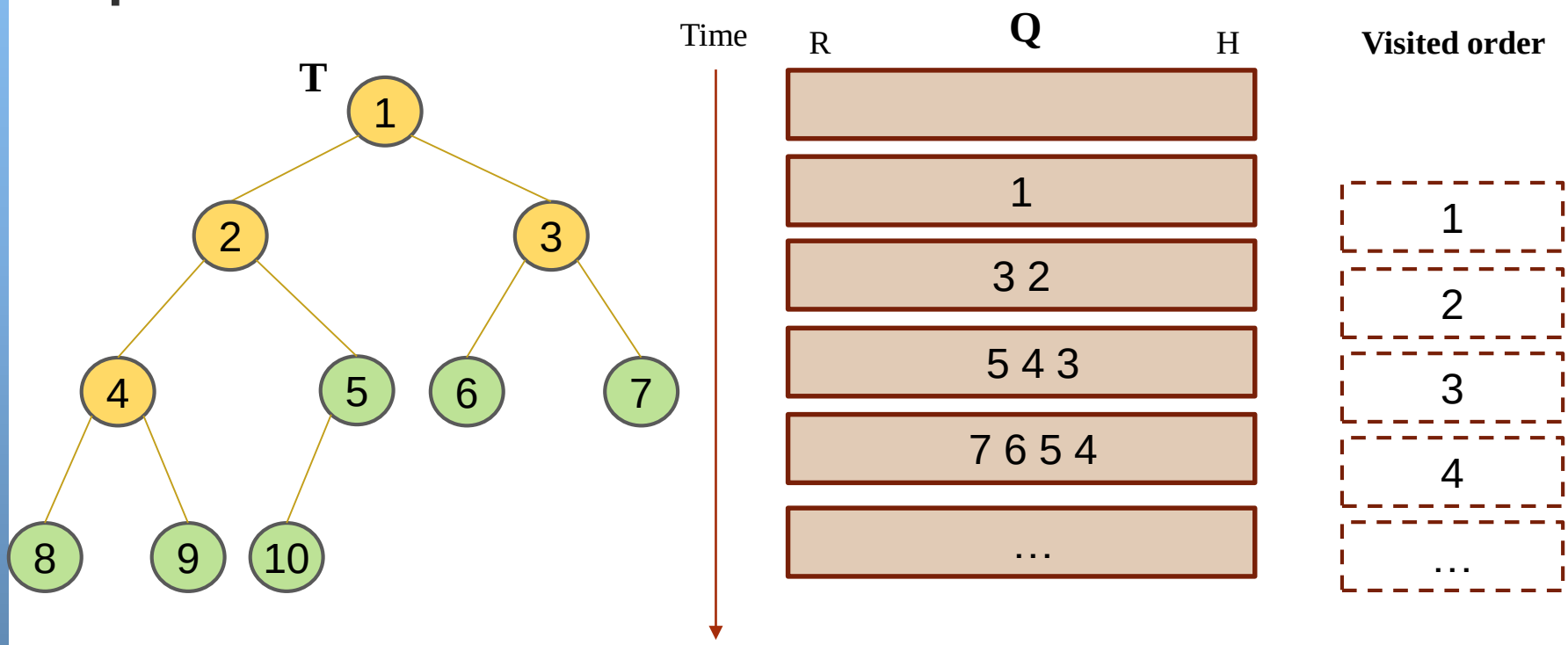
- **Traversal by levels method (breadth traversal):**
  - Using list type of queue to store all nodes of tree
  - Given a binary tree T, and a queue Q, traversal steps as follows:

```
Initialize (Q)          // Khởi tạo: Q=∅.
InsertQ (T, Q);         //Bổ sung nút gốc vào Q
while ( Q <> ∅ ) do
   P = DeleteQ (Q);     //Lấy một nút ra khỏi Q để chuẩn bị thăm
   Visit (P);           //Thăm P
   if <P has left child PL> then InsertQ (PL, Q);
   if <P has right child PR> then InsertQ (PR, Q);
end while ;
```

Traversal methods – 2nd approach

- **Traversal by levels method (breadth traversal):**

Exp:



Implementation of binary trees

- **There are 2 storage structures that can be used:**
  - Sequential storage structure(CTLT tuần tự):
    - Suitable only for special binary trees such as complete and full.
    - For normal binary trees, additional dummy nodes should be added to make them be special.
    - Therefore, this method is often not efficient, and applied when it is the only method in the chosen programming language.
  - Linked storage structures (CTLT móc nối):
    - Suitable for dynamic and non-linear structures like trees.
    - **Here we only care about this method!**

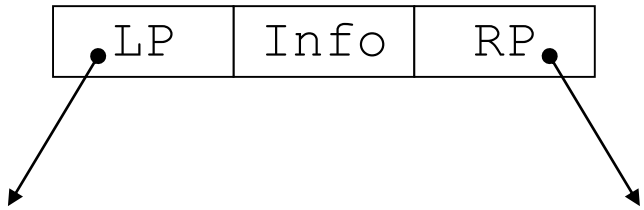
Implementation of binary trees

- **Principles: using linked storage structures, we need to store 2 parts of a binary tree:**
  - Elements (Nodes):
    - Stored by nodes of linked storage structures.
  - Branches:
    - Stored by pointers in the nodes.
    - Each node of linked storage structure has 2 pointers for at most 2 children.
- **Using linked storage structures**
  - Structure of a node (Cấu tạo mỗi nút):
    - Field Info: containing the information of an element
    - Fields LP and RP are pointers that points to left child and right child respectively (they are NULL if the node has no children)
    - Definition of structure:

Implementation of binary trees

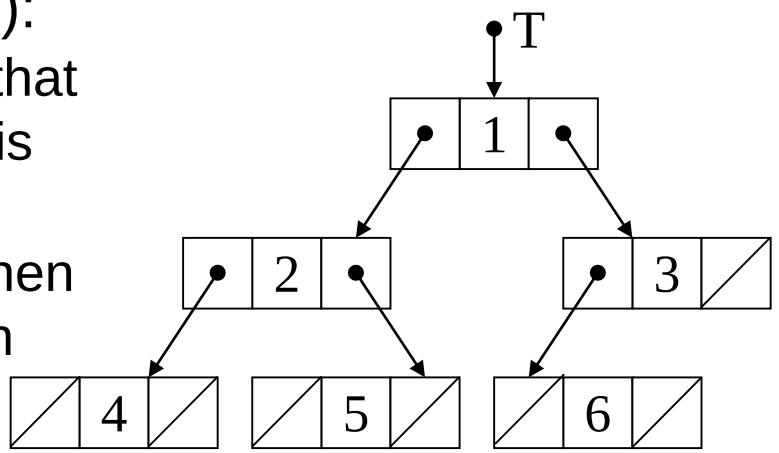
- **Using linked storage structures**
  - Structure of a node (Cấu tạo mỗi nút):
    - Field Info: containing the information of an element
    - Fields LP and RP are pointers that points to left child and right child respectively (they are NULL if the node has no children)
    - Definition of structure:

```
struct  Node  {
   type  Info;
   struct Node * LP, *RP;
};

typedef  Node*  PNode;
```



Implementation of binary trees

- **Using linked storage structures:**
  - Organization of tree (see figure):
    - There are at least one pointer that points to the root. This pointer is also an access point
    - Notes: if the size of tree is N, then there are N+1 NULL pointers in this structure..
    - Definition: there are some ways, and we use first way later.

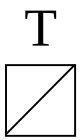


1st way:
```
typedef  PNode  BinaryTree;
```
2nd way:
```
struct  BinaryTree {
    PNode  T;    //con trỏ trỏ vào nút gốc
    int  n;      //kích thước cây
}
```

Implementation of binary trees by LSS

- **Basic operations:**
  - Initialize binary tree (khởi tạo): create an empty tree.

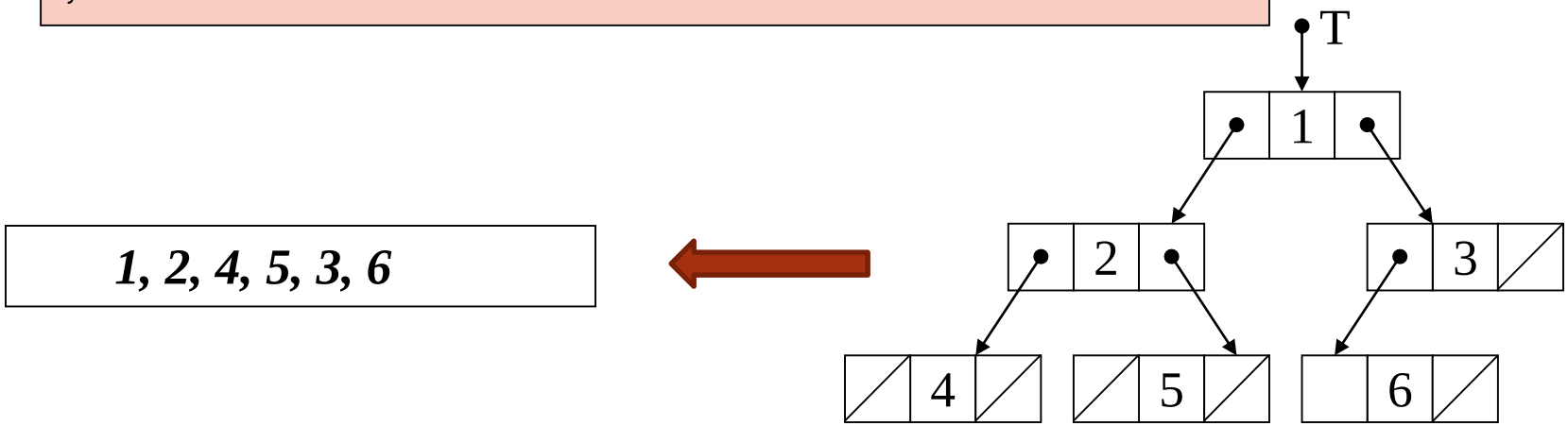
```
void  InitBT (BinaryTree  & T) {
   T = NULL;
}
```



Implementation of binary trees by LSS

- **Basic operations:**
  - Preorder traversal (Duyệt cây theo thứ tự trước)

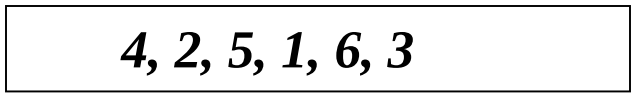
```
void  PreOrderTraversal (BinaryTree  T)
{
   if  (T==NULL) return;
   Visit(T);
   PreOrderTraversal(T->LP);
   PreOrderTraversal(T->RP);
}
```

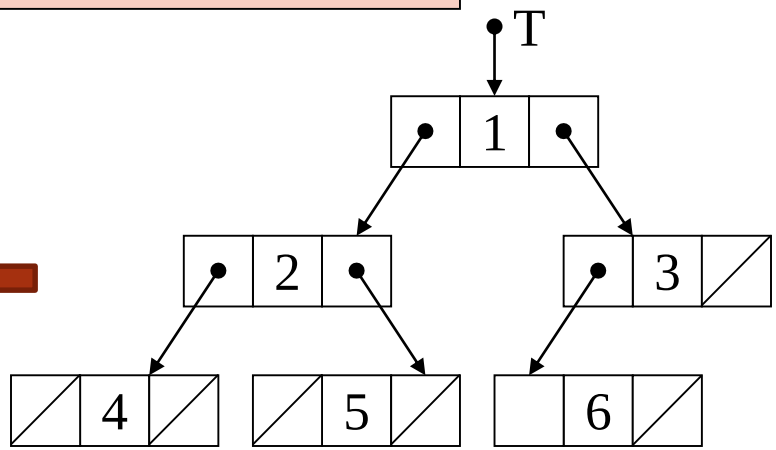


Implementation of binary trees by LSS

- **Basic operations:**
  - Inorder traversal (Duyệt cây theo thứ tự giữa)

```
void  InOrderTraversal (BinaryTree  T)
{
    if  (T==NULL) return;
    InOrderTraversal(T->LP);
    Visit(T);
    InOrderTraversal(T->RP);
}
```

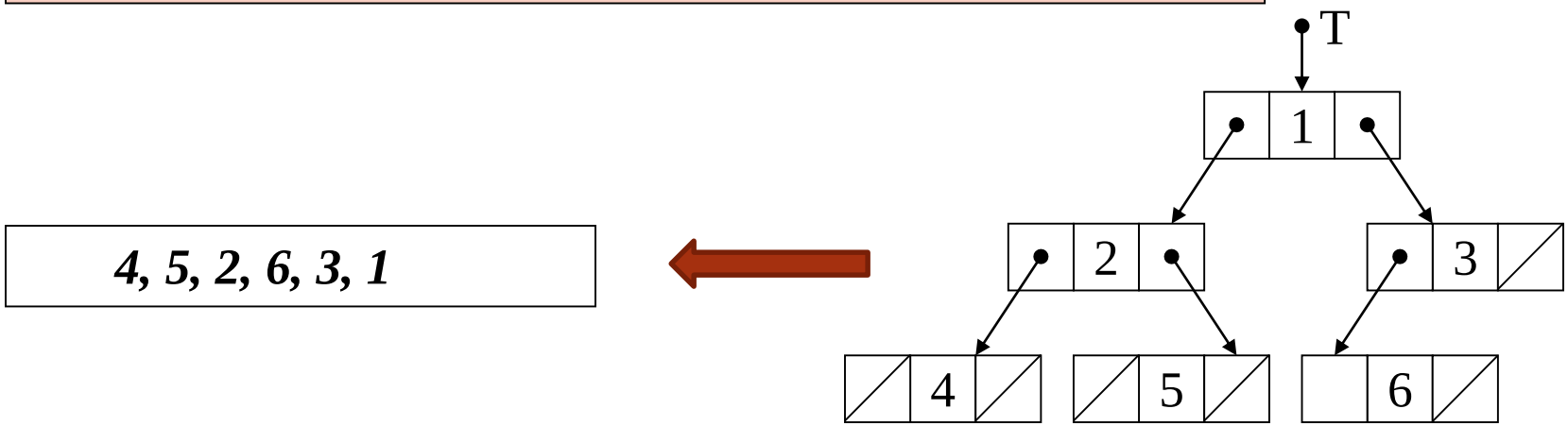




Implementation of binary trees by LSS

- **Basic operations:**
  - Postorder travelsal (Duyệt cây theo thứ tự sau)

```
void   PostOrderTraversal (BinaryTree  T)
{
   if  (T==NULL) return;
   PostOrderTraversal(T->LP);
   PostOrderTraversal(T->RP);
   Visit(T);
}
```

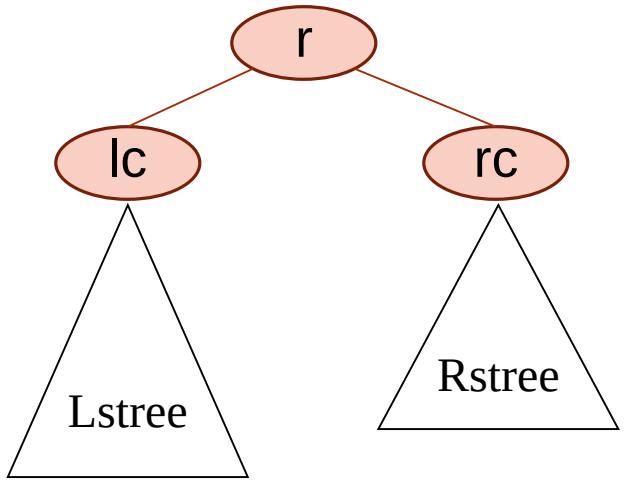


Binary Search Trees

- ***Problem*: Searching a node in binary trees**
  - Applying existing traversal methods seems to be slow, due to all nodes are involved in searching process.
  - We need a special binary tree that helps searching faster
- ***Solution*: Binary Search Trees (Cây nhị phân tìm kiếm)**

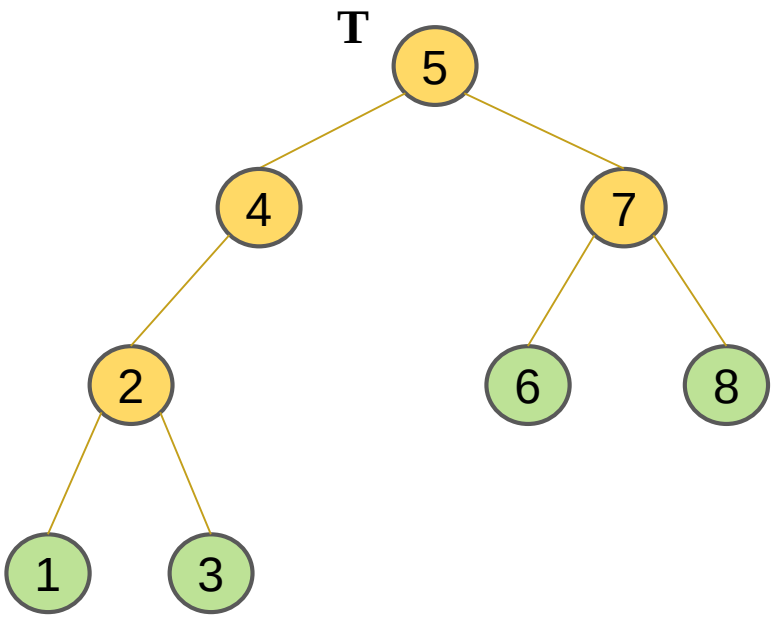
Binary Search Trees

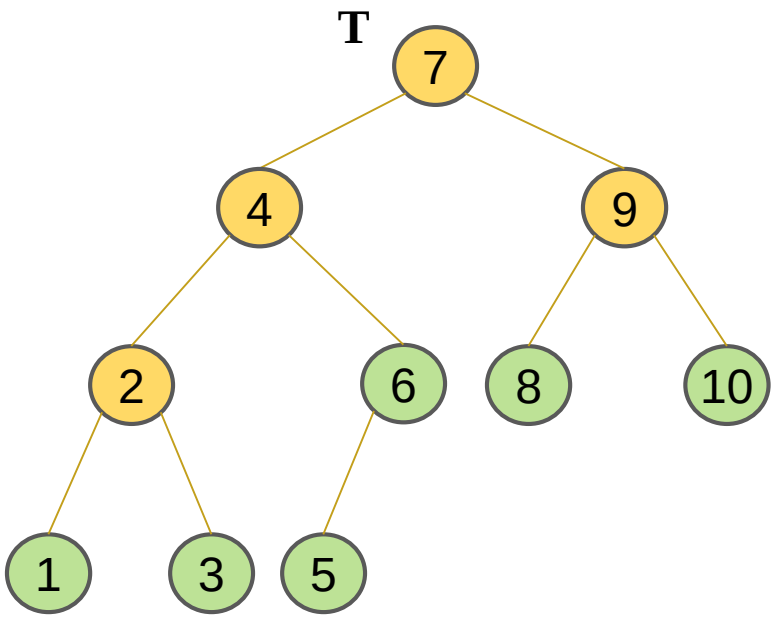
- **Suppose we have a binary tree `T`**
  - All nodes having a special field called *`key`* (we note *`key`* of a node `p` is `key(p)`).
  - We call the root of `T` is `r`, left child and right child of the root are `lc` and `rc` respectively, left sub tree and right sub tree of the root are `Lstree` and `Rstree` respectively;
- **T is binary search tree (BST) if it satisfies the following conditions:**
  - `key(lc) < key(r);`
  - `key(r) < key(rc);`
  - `Lstree` and `Rstree` are also BSTs



Binary Search Trees

Examples:





Inorder traversal: sorted list of nodes

1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8 9 10

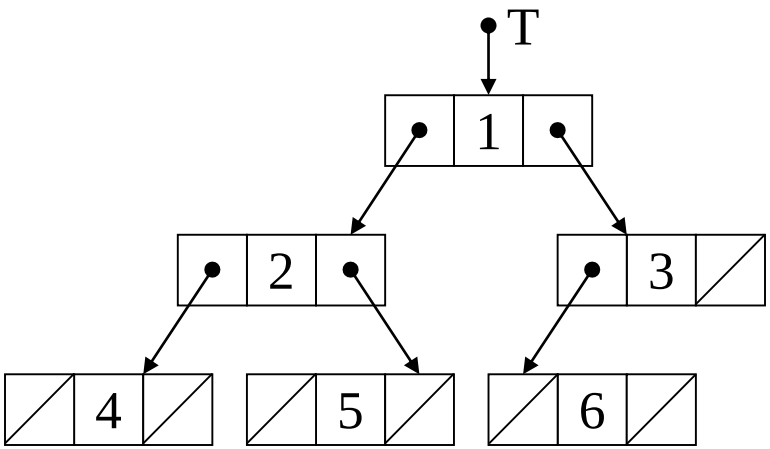
Binary Search Trees

- **Definition: similar to definition of binary tree**

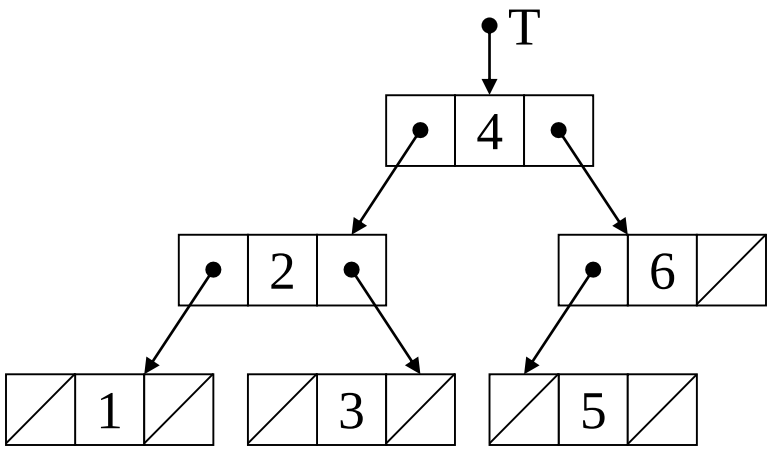
```
struct  Node  {
   keytype  key;
   Node *LP, *RP;
};

typedef  Node*  PNode;
typedef  PNode BinaryTree;
typedef  BinaryTree  BSearchTree;
```

Binary Search Trees

- **Organization: similar to binary tree**
  - Exp:



Binary tree



Binary search tree

Binary Search Trees

- **Basic operations:**
  - *Search for a node with given value* x *in BST* T: this operation returns a pointer that points to found node, otherwise it returns NULL. Applying the property of a BST, we have the following recursive algorithm:
    - Base case: if (T=NULL or key(T) = x) return T;
    - Recursive case:

      otherwise if (x<key(T)) search for a node in left sub-tree of T

      else search for a node in right sub-tree of T

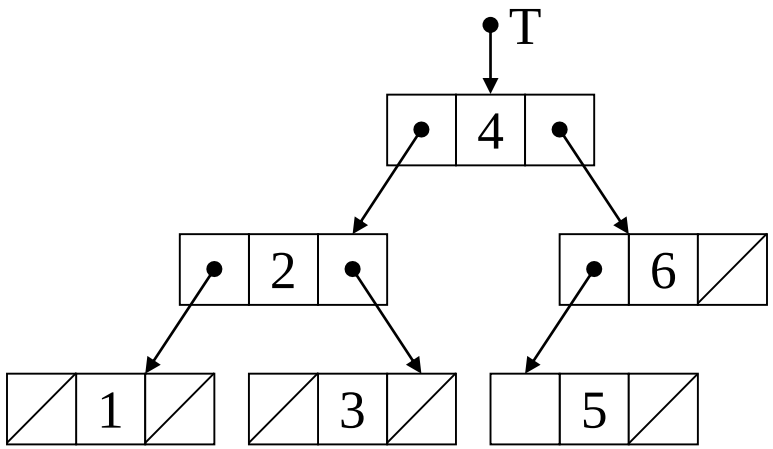
```
PNode Search (BSearchTree T, keytype  x){
   if  (T==NULL)  return  NULL;
   if  (x == T->Key) return  T;
   else
      if  (x < T->Key)  return  Search(T->LP, x);
      else  return  Search(T->RP, x);
}
```

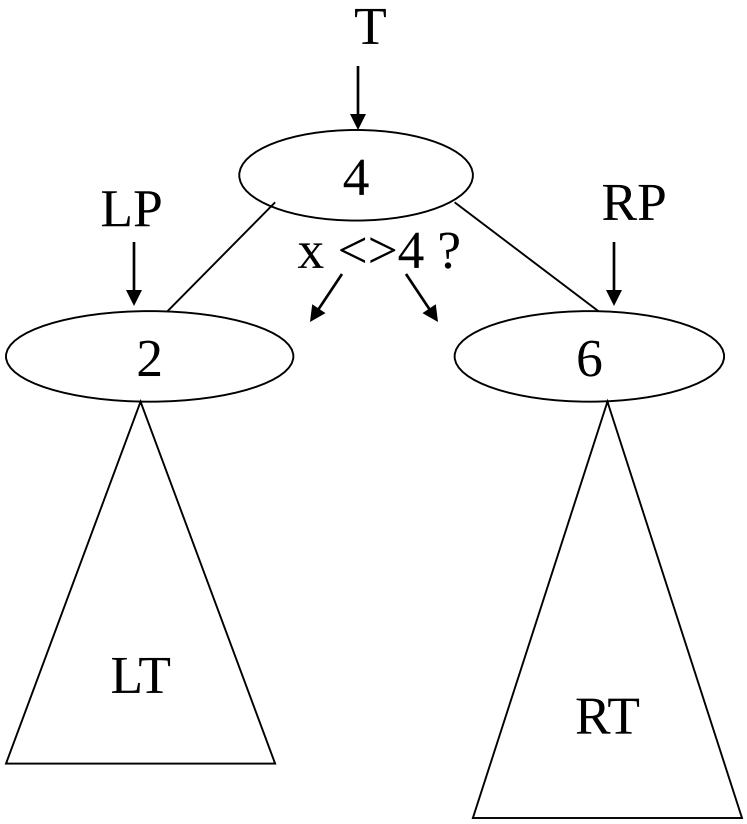
$$O(\text{Search}) = \log_2(N),\ N \text{ is size of T}$$

Binary Search Trees

- ***Add a new node***

  Exp: ideas of adding a new node x





Binary Search Trees

- ***Add a new node*: function to implement**

```
void InsertT(BSearchTree & Root, keytype x){
   PNode Q;
   if (Root==NULL) {          //tree is empty
        Q = new Node;
        Q->Key = x;
        Q->LP =       Q->RP = NULL;
        Root = Q;
   }
   else {
        if (x < Root->Key) InsertT(Root->LP, x);
        else if (x > Root->Key) InsertT(Root->RP, x);
   }
}
```

Binary Search Trees

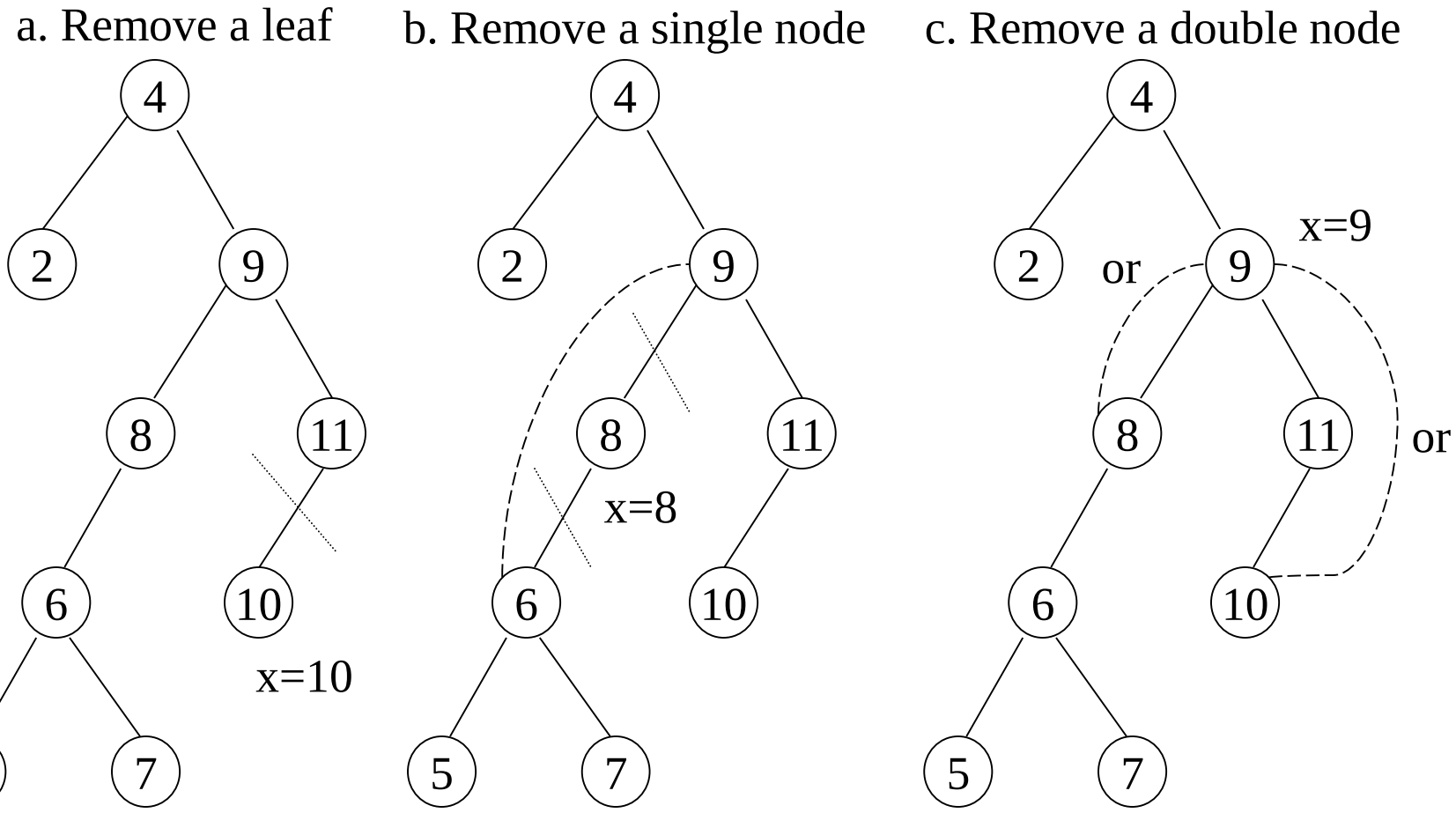
- ***Remove a node with a given key x from BST:***

  Algorithm:

  - Search for the node to be removed (with `key = x`)
  - Remove the node and re-arrange other nodes if necessary. 3 cases may happen:
    - If the remove node is a leaf => no re-arrangement
    - If the node is single => only child of the node will replace the node
    - If the node is double, means that it has two sub-trees `LTree`, `RTree` => replace the node by `Max(LTree)` (maximal node in `LTree`) or `Min(RTree)` (minimal node in `RTree`)
  - Notes:
    - `Max(LTree)` is the right most node of `LTree`.
    - `Min(RTree)` is the left most node of `RTree`.

Binary Search Trees

- ***Remove a node with a given key x from BST:***
  - Exp:



Binary Search Trees

- ***Remove a node with a given key x from BST:*** **functions to implement**

```
void  DeleteT(BSearchTree & Root, keytype  x){
   if  (Root != NULL) {
      if  (x < Root->Key)  DeleteT(Root->LP, x);
      else  if  (x > Root->key)  DeleteT(Root->RP, x);
      else  DelNode (Root);          //remove the root
   }
}
```

```
void  DelNode(PNode & P) {  //remove node P & re-arrange if necessary
   PNode Q, R;
   if (P->LP == NULL) {     //the node has only right child
      Q = P;
      P = P->RP;
   }
   else
      if (P->RP = NULL)     //the node has only left child
      {                     //continue in next page
```

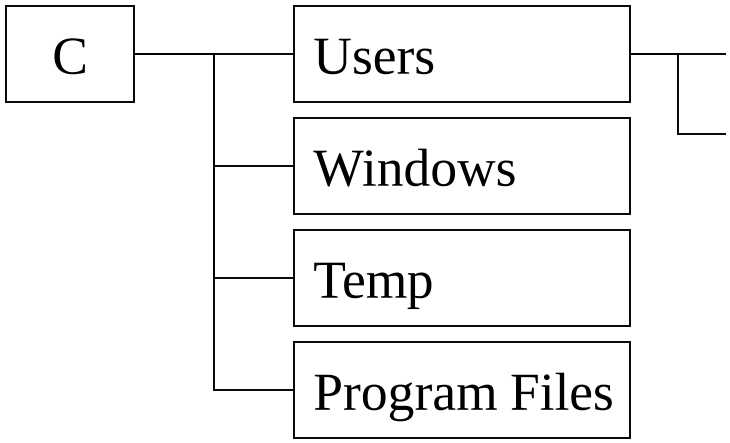
Binary Search Trees

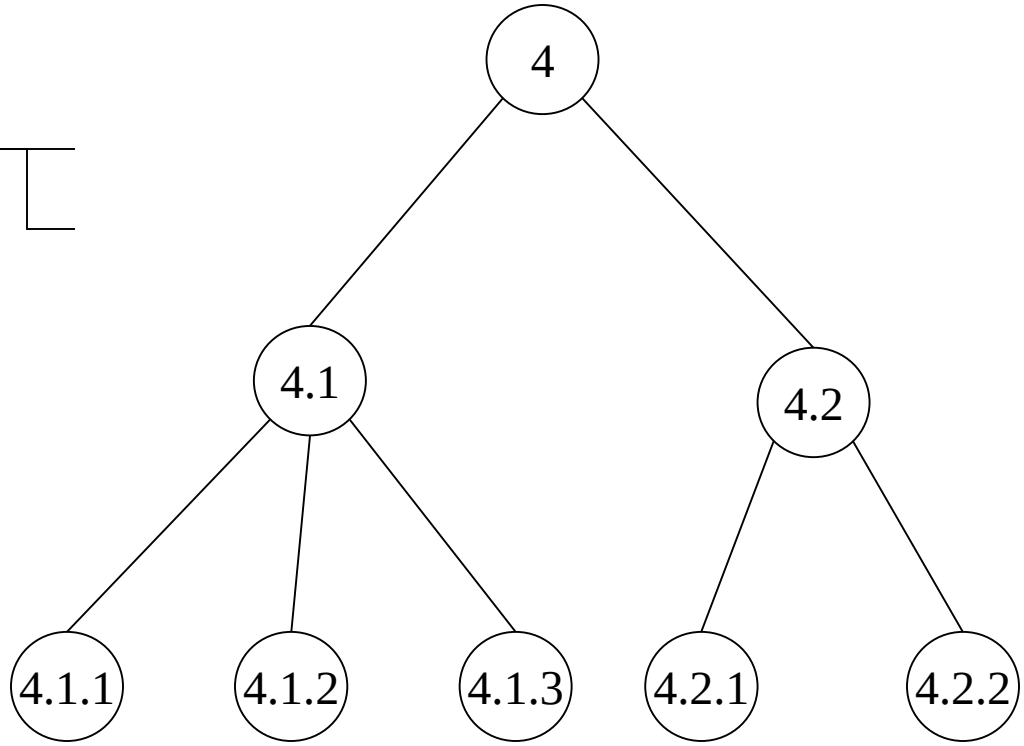
- ***Remove a node with a given key x from BST (cont')***

```
            Q = P;
            P = P->LP;
        }
        else {              //Remove a double node
            Q = P->LP;
            if  (Q->RP == NULL) {
                P->Key = Q->Key;
                P->LP = Q->LP;
            }
            else {
                do {      //R used to store parent of Q
                    R = Q;
                    Q = Q->RP;
                } while  (Q->RP != NULL);
                P->Key = Q->Key;
                R->RP = Q->LP;
            }
        }
    delete  Q;
}
```

Generalized Trees

- **Concept**
  - A generalized tree (cây tổng quát) is a tree with any degree (often more than 2)
  - Exp: family tree, file system tree, etc.





Generalized Trees

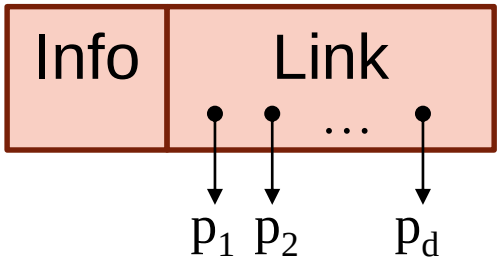
- **Implementation methods (Các phương pháp cài đặt): 2 approaches:**
  - 1st one: Directly using linked storage structures (Cài đặt trực tiếp sử dụng CTLT móc nối)
  - 2nd one: Indirect implementation by binary trees (Cài đặt gián tiếp qua cây nhị phân)

Generalized Trees

- **1st implementation method: Directly using linked storage structures**

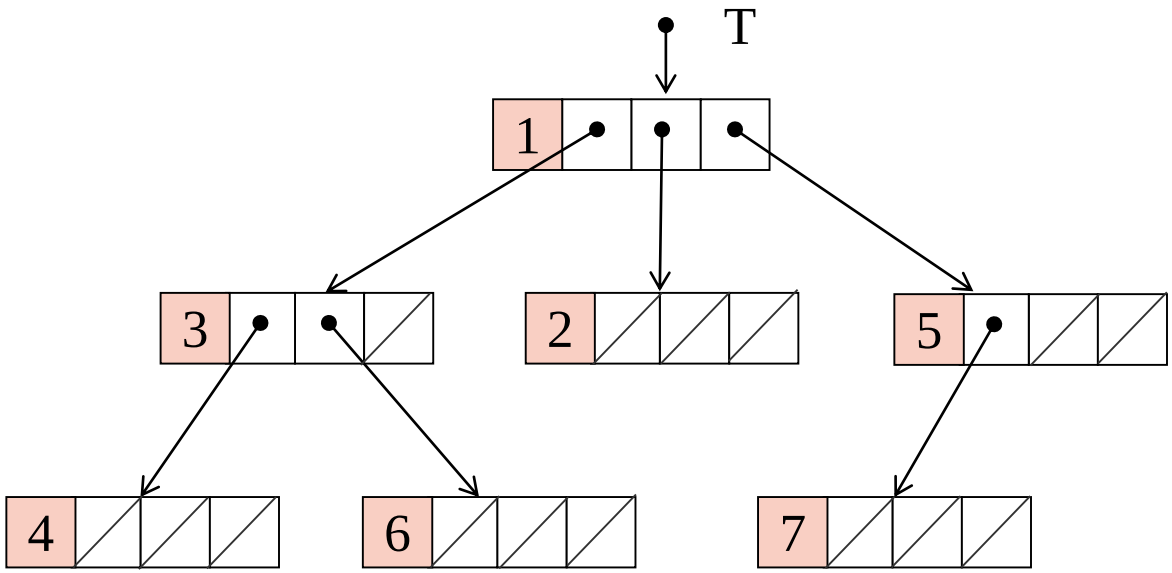
  Main ideas:

  - Similar method used in implementation of binary trees
  - Given a d-degree tree, the structure of a node consists of 2 parts:
    - *Info*: containing the information of an element
    - *Link*: including d pointers p_1, p_2,…p_d aiming to point to maximal number d children.



Generalized Trees

- **1st implementation method: Directly using linked storage structures**
  - Organization of GT
    - ★ One pointer acted as access point points to the root
    - ★ For each node, pointers in its link point to its children



Generalized Trees

- **Limitations of 1st implementation method:**
  - If the size and degree of implemented tree are `N` and `d` respectively => the number of NULL pointers are:

    `N(d-1) + 1`

  - When d ≥ 2, the number of NULL pointers is bigger than the size of the tree => wasting lots of memory space.
  - The degree of the tree must be known before implementation and fixed. This requirement is not always met in many real applications.

Generalized Trees

- **2nd implementation method: Indirect implementation by binary trees**

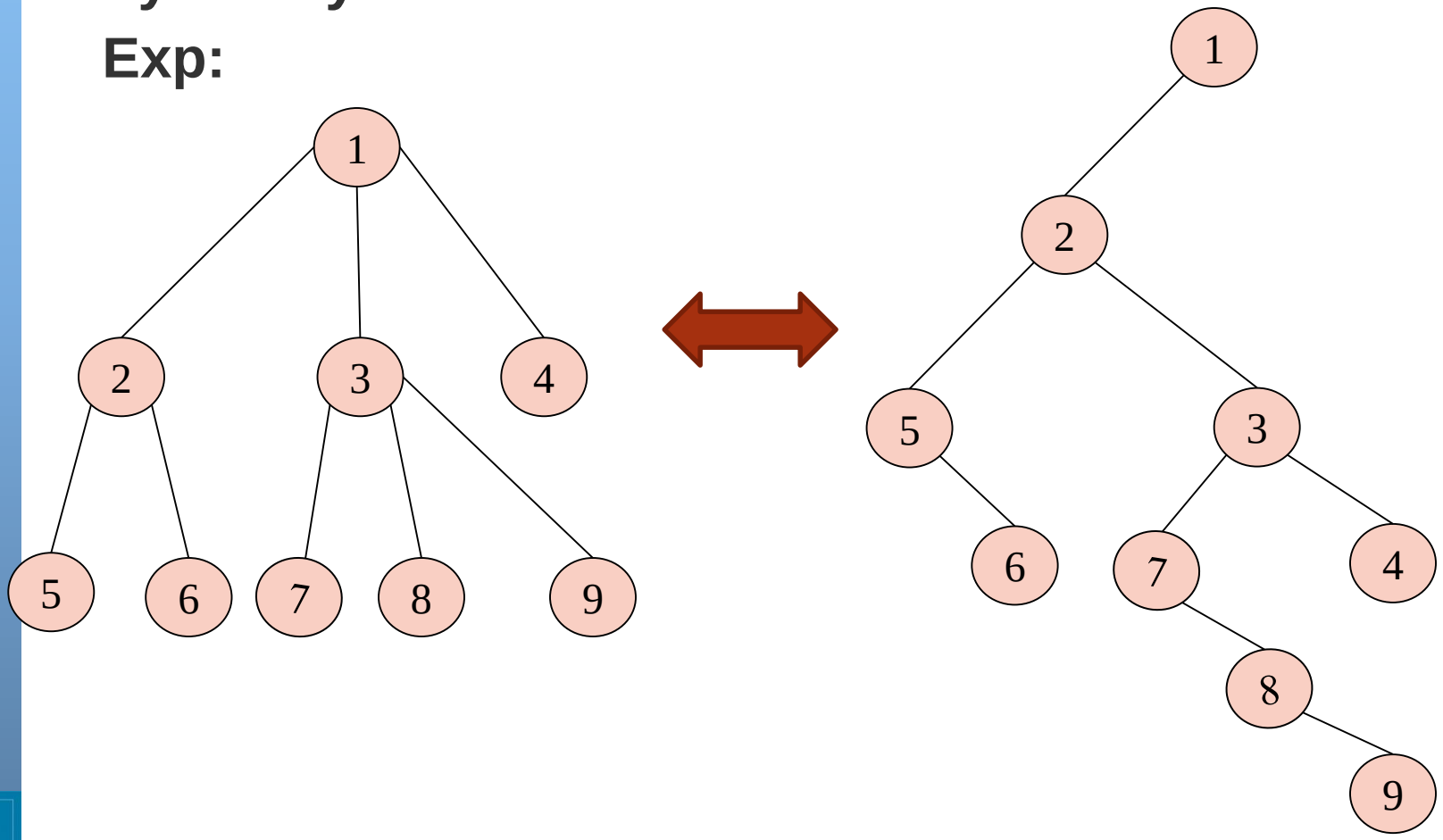
  Ideas:

  - Try to overcome the limitations of 1st implementation method.
  - Create a suitable mapping rule between a generalized tree and a binary tree as follows:
    - Make the generalized tree ordered if necessary.
    - For a given node:
      - Transform the left most child to left child of binary tree
      - Transform the next sibling node to right child of the node

Generalized Trees

- **2nd implementation method: Indirect implementation by binary trees**

  Exp:



Generalized Trees

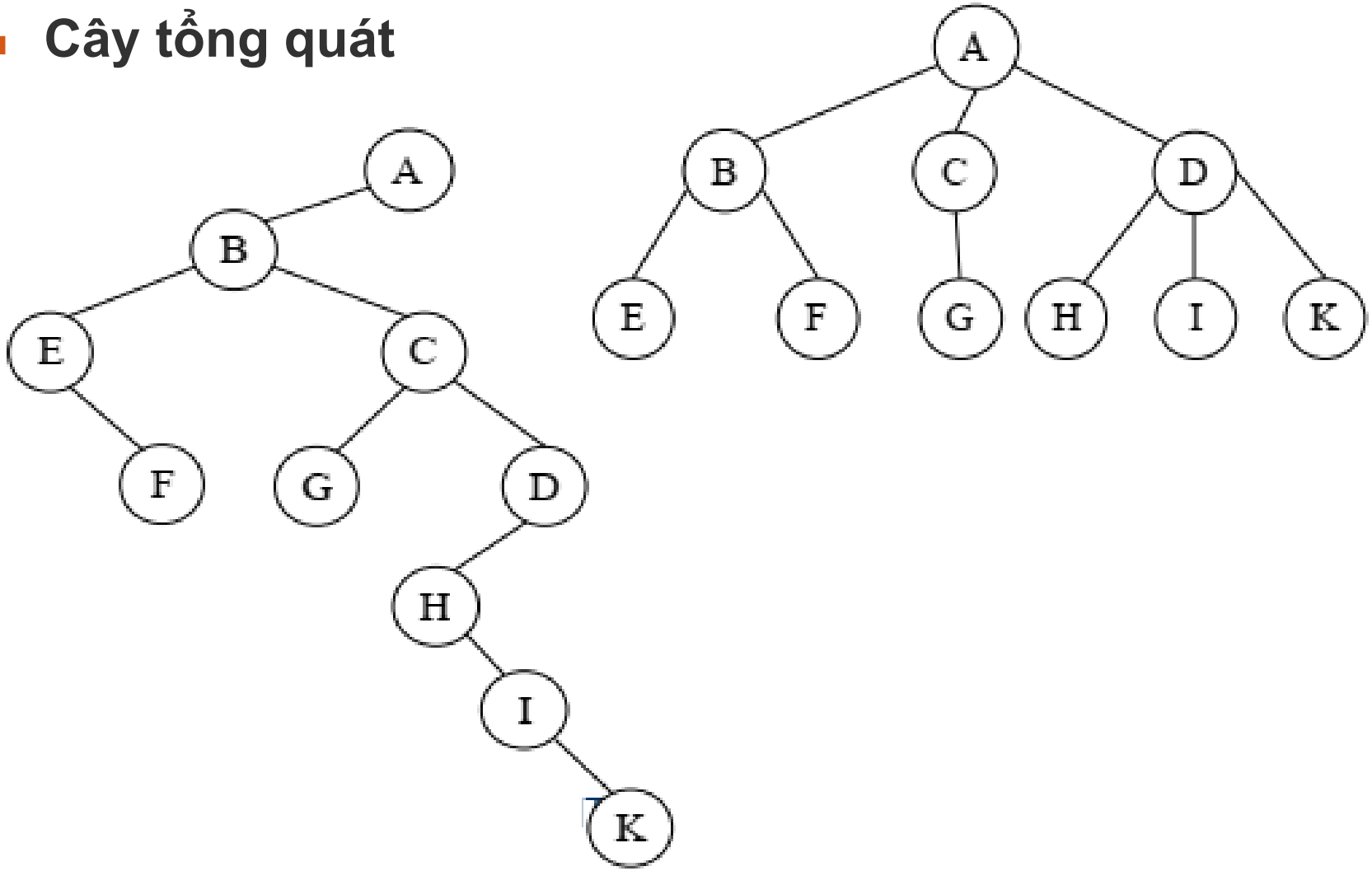
- **2nd implementation method: Indirect implementation by binary trees**

  Notes:

  - This method solved the limitations of 1st method, therefore can be effectively used (about memory space) for trees with dynamic degrees.
  - The additional mapping rule can make tree operations more complex.

Biểu diễn cây tương đương

- Cây tổng quát



- Cây nhị phân tương đương

Cây nhị phân tương đương

- **Mapping rule between a GT and BT :**
  - For each node:
    - Left most child becomes the left node of BT
    - Sibling becomes the right node of BT
- **This representation:**
  - Memory usage efficiency
  - However, the fact of using mapping rules make some operation more complex

Some applications of trees

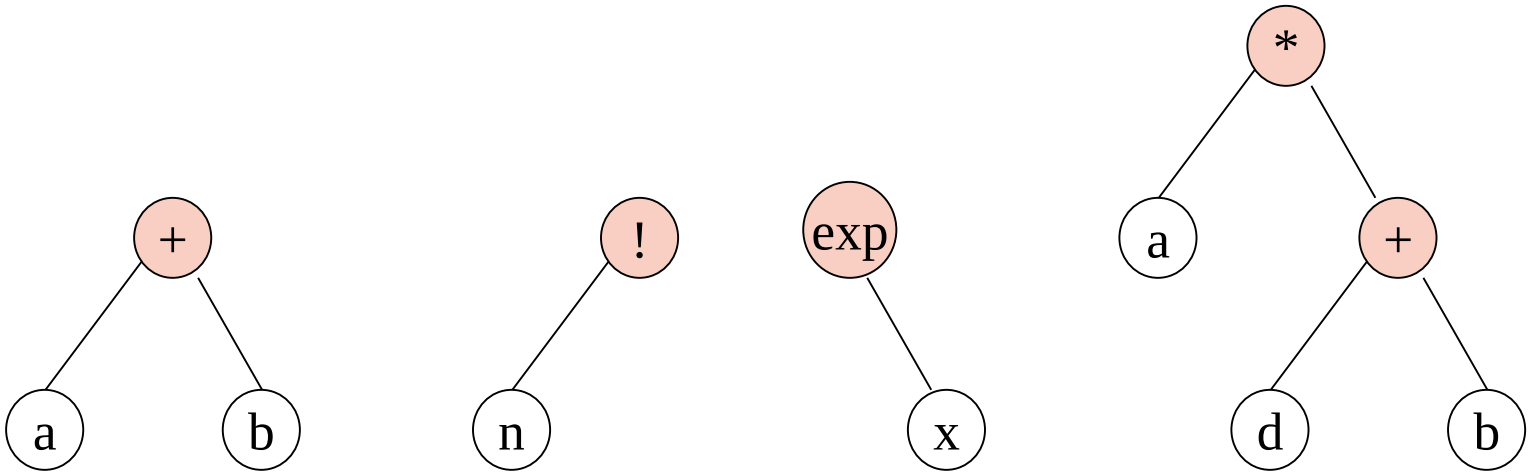
- **Applications of binary trees:**
  - Binary search trees
  - Trees of expressions (Cây biểu thức)
  - Balanced trees (Cây cân bằng)
- **Other applications:**
  - Decision tree (Cây hỗ trợ ra quyết định)

Some applications of trees

- **Trees of expressions**
  - Expression form (Biểu diễn biểu thức):
    - expression = expression <operator> expression
    - special case: expression = const
    - operator (Phép toán): +, / , *, -, exp, !, ...
  - Using binary trees:
    - Branch nodes for operators
    - Leafs for operands
  - Notes:
    - preorder traversal => prefix expression
    - postorder traversal => postfix expression
    - Inorder traversal => infix expression

Some applications of trees

- **Trees of expressions: examples**



a + b n! e^x a * (d + b)

Some applications of trees

- **Decision trees**
  - *Problem*: there are 5 golden coins. Find the only lightest coin among them?
  - *Solution*: suppose 5 coins A, B, C, D, E. We use a decision tree as follows: