HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

DATA STRUCTURES AND ALGORITHMS

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Chapter 6- Tree

Lesson 1. Definitions and general concepts, binary trees

ONE LOVE. ONE FUTURE.

OBJECTIVES

After the lesson, students can

1. Understand the concept of tree data structures and related concepts.

2. Implement the tree data structure.

CONTENTS

1. Definitions and general concepts

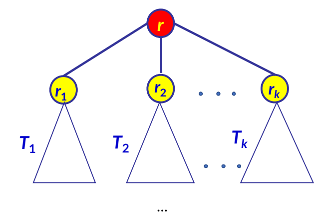
2. Binary tree

1. DEFINITIONS AND GENERAL CONCEPTS

- 1.1. Tree definition
  - A tree consists of nodes, has a special node called the **root**, and edges connect the nodes

Tree definition:

- *Basic step: A node **r** is a tree and **r** is called the root of this tree*
- *Recursive step : Suppose $T_1, T_2, \dots, T_k$ are trees with roots $r_1, r_2, \dots, r_k$:*
  - *Build a new tree by making r the parent of nodes $r_1, r_2, \dots, r_k$*
  - *In this tree r is the root and $T_1, T_2, \dots, T_k$ are subtrees of root r. Nodes $r_1, r_2, \dots, r_k$ is called a child of node r*

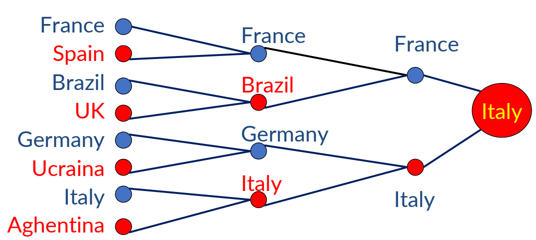


Tree

Note: *Null tree is the tree without node*

1. DEFINITIONS AND GENERAL CONCEPTS

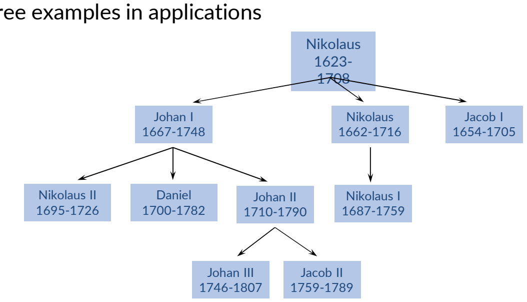
- 1.1. Tree definition
  - Realistic tree examples in applications



Describe the schedule of sports tournaments in a knockout format, such as the second round of the World Cup

1. DEFINITIONS AND GENERAL CONCEPTS

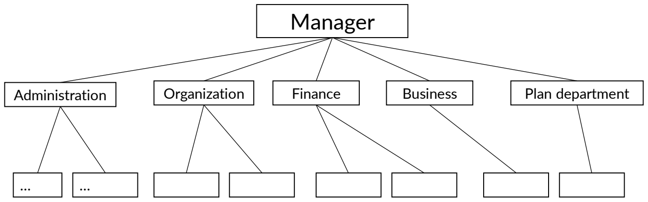
- 1.1. Tree definition
  - Realistic tree examples in applications



Family tree of the Bernoulli family of mathematicians

1. DEFINITIONS AND GENERAL CONCEPTS

- 1.1. Tree definition
  - Realistic tree examples in applications



Cây phân cấp quản lý hành chính

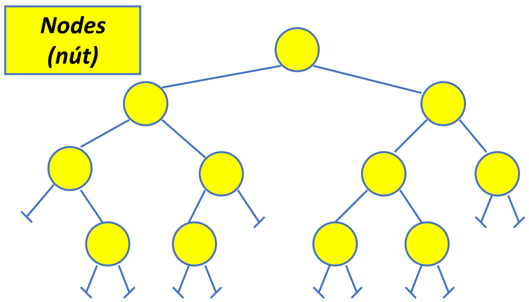
1. DEFINITIONS AND GENERAL CONCEPTS

- 1.1. Tree definition
  - Realistic tree examples in applications

Directory tree

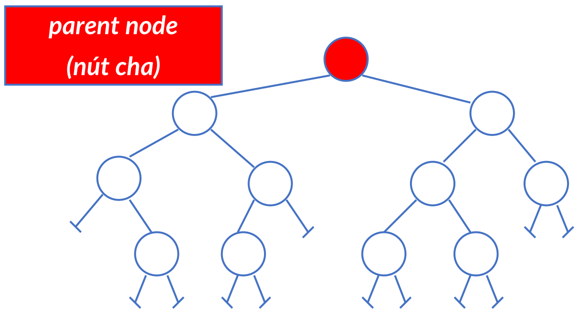
1. DEFINITIONS AND GENERAL CONCEPTS

- 1.2. Terminology



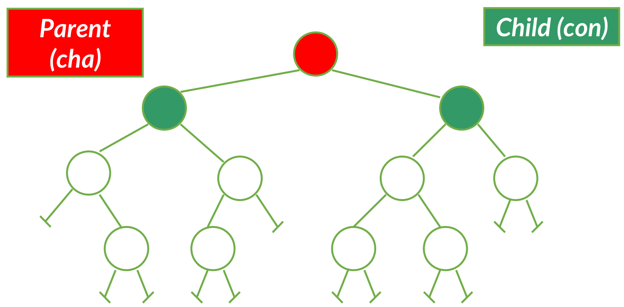
1. DEFINITIONS AND GENERAL CONCEPTS

- 1.2. Terminology



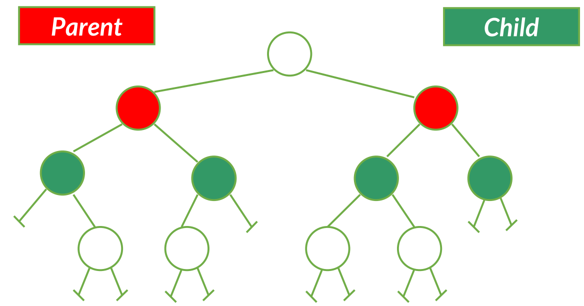
1. DEFINITIONS AND GENERAL CONCEPTS

- 1.2. Terminology



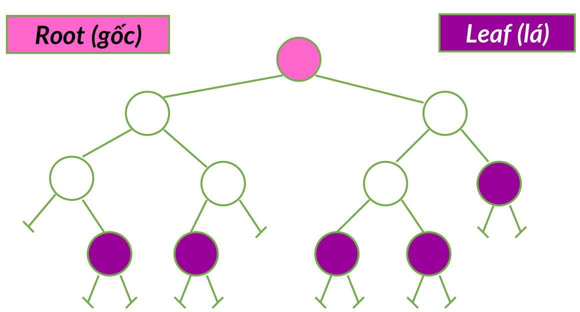
1. DEFINITIONS AND GENERAL CONCEPTS

- 1.2. Terminology



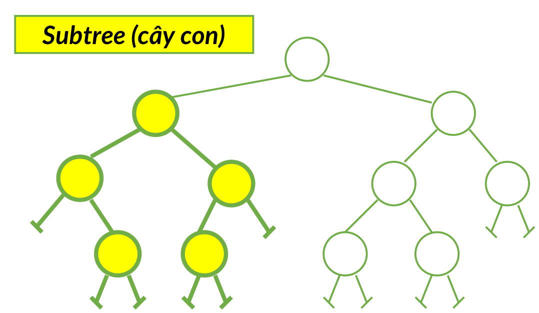
1. DEFINITIONS AND GENERAL CONCEPTS

- 1.2. Terminology



1. DEFINITIONS AND GENERAL CONCEPTS

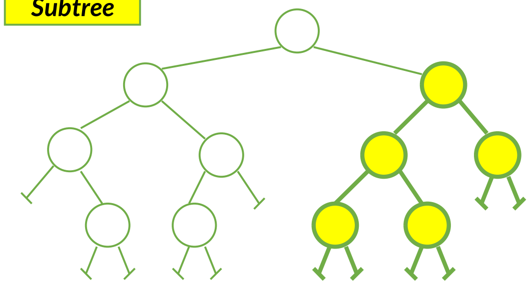
- 1.2. Terminology



1. DEFINITIONS AND GENERAL CONCEPTS

- 1.2. Terminology

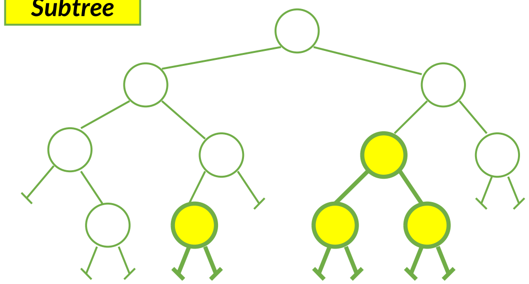
Subtree



1. DEFINITIONS AND GENERAL CONCEPTS

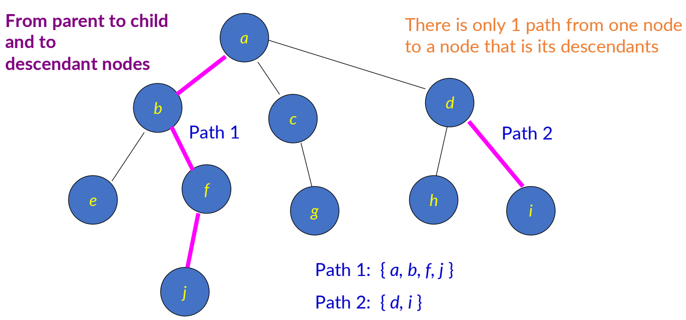
- 1.2. Terminology

Subtree



1. DEFINITIONS AND GENERAL CONCEPTS

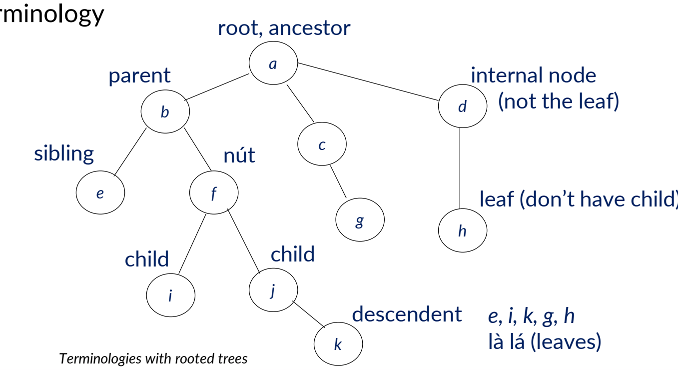
- 1.2. Terminology



Path on the tree

1. DEFINITIONS AND GENERAL CONCEPTS

- 1.2. Terminology



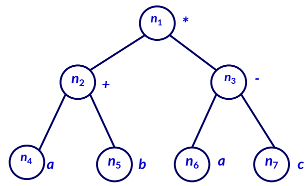
Terminologies with rooted trees

1. DEFINITIONS AND GENERAL CONCEPTS

- 1.2. Terminology

Labeled Tree (Cây có nhãn)

- Each node of the tree has a label or value
- The node's label is not the name of the node but the value stored in it
- For example: Consider a tree with 7 nodes n1, ..., n7. Label the buttons:
  - Node n1 is labeled *;
  - Node n2 is labeled +;
  - Node n3 has label -;
  - Node n4 has label a;
  - Node n5 has label b;
  - Node n6 has label a;
  - Node n7 is labeled c.



Expression tree (a+b)(a-c)*

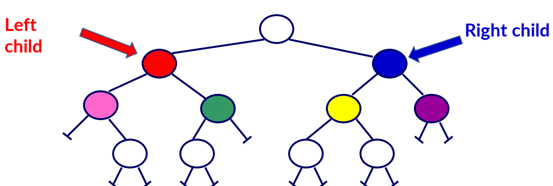
CONTENTS

1. Definitions and general concepts

2. Binary tree

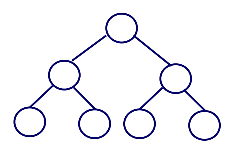
2. BINARY TREE

- **2.1.** Binary tree
  - **Binary Tree:** A tree in which each node has at most 2 child
    - ***Left child*** and ***right child***
    - Each node either:
      - Not have child
      - Has only left child
      - Has only right child
      - Has both left and right child



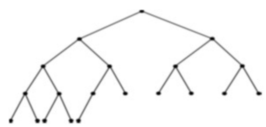
2. BINARY TREE

- **2.2.** Full binary tree
  - **Full Binary Tree (Cây nhị phân đầy đủ):** is binary tree that satisfies
    - Each leaf node has the same depth
    - The internal nodes have exactly 2 children



2. BINARY TREE

- **2.3.** Complete binary tree
  - **Complete Binary Tree (Cây nhị phân hoàn chỉnh):** A binary tree of depth *n* satisfies:
    - Is a complete binary tree if nodes at depth n are not taken into account, and
    - All nodes at depth n are as far left as possible.
  - A complete binary tree of depth *n* has a number of nodes ranging from 2^{n-1} to $2^n - 1$

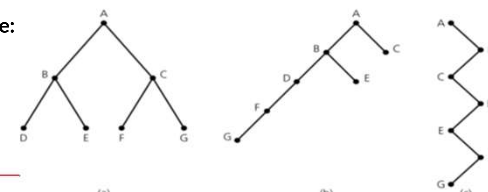


Complete binary tree

2. BINARY TREE

- **2.4.** Balanced binary tree
  - **A binary tree is called balanced** if the height of the left subtree and the height of the right subtree differ by no more than 1 unit.
  - Comment:
    - If the binary tree is complete then it is complete
    - If the binary tree is complete then it is balanced

Example:



(a) (b) (c)

1. Which tree is complete?
2. Which tree is balance?
3. Which tree is full?

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Chapter 6- Tree

Lesson 2. Tree representation data structure, tree traversal

ONE LOVE. ONE FUTURE.

OBJECTIVES

After this lesson, students can

1. Understand the concept of traversing a tree: inorder, preorder, postorder
2. Implement the tree representation data structure

CONTENTS

1. The data structure represents a tree

1.1. Represent trees using arrays

1.2. Represent a tree as a list of children

1.3. Represent the tree with the left child and the right neighbor

2. Traverse tree

1. THE DATA STRUCTURE REPRESENTS THE TREE

- 1.1. Represent the tree by array
  - Suppose T is a tree with nodes named 1, 2, . . . , *n*.
  - Represent *T* :
    - linear list A where each element A[i] contains a pointer to node i's parent
    - The root of T can be distinguished by a null pointer.
    - Assign *A*[*i*] = *j* if node *j* is parent of node *i*,

      A[*i*] = 0 if node *i* is root.
    - the **parent** operation returns the parent of a node
  - This representation is based on the fact that each node of the tree (except the root) has only one parent.

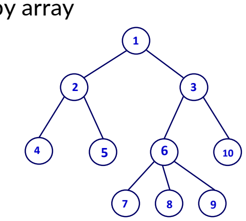
1. THE DATA STRUCTURE REPRESENTS THE TREE

- 1.1. Represent the tree by array
  - With this representation: the parent of a node can be determined in constant time.
  - Path from a node to an ancestor (including the root):

    n ← **parent**(n) ← **parent**(**parent**(n)) ← ...
  - You can also use the array L[i] to support recording labels for nodes,
  - or use each element A[i] as a record with 2 fields:
    - integer variable records parent
    - label.

1. THE DATA STRUCTURE REPRESENTS THE TREE

- 1.1. Represent the tree by array
  - **Example**



A	0	1	1	2	2	3	6	6	6	3

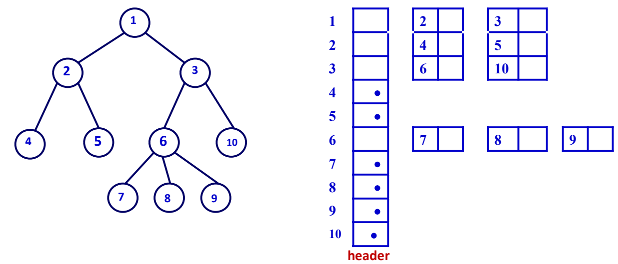
- ***Limitation:*** The use of parent pointers is not suitable for operations with children.
- Given node n, it takes a long time to determine n's children and n's height.
- Do not give us the information about the order of the child nodes → **leftmost_child** and **right_sibling** be undefined → this representation is only used in certain cases.

1. THE DATA STRUCTURE REPRESENTS THE TREE

- 1.2. Represent the tree by list of children
  - Each node of the tree stores a list of its children.
  - List of children can be represented by one of the list representations presented in the previous chapter.
  - The number of children of nodes varies widely → linked lists are often the most appropriate choice.

1. THE DATA STRUCTURE REPRESENTS THE TREE

- 1.2. Represent the tree by list of children



  - There is an array of pointers to the beginning of the child list of nodes 1, 2, . . . , 10: *header*[*i*] points to the child list of node *i*.

1. THE DATA STRUCTURE REPRESENTS THE TREE

- 1.2. Represent the tree by list of children
  - **Example:** The following description can be used to represent trees

```
typedef  ?  NodeT;  /* sign ? needs to be replaced by a suitable type definition  */
typedef  ?  ListT; /*  sign ? needs to be replaced by a suitable list type definition */
typedef  ?  position;
typedef struct
{
    ListT  header[maxNodes];
     labeltype  labels[maxNodes];
    NodeT  root;
 } TreeT;
```

  - Assume that the root of the tree is stored in the ***root*** field and 0 to represent an empty node.

1. THE DATA STRUCTURE REPRESENTS THE TREE

- 1.2. Represent the tree by list of children
  - Below is an illustration of the **leftmost_child** operation settings. The implementation of the remaining operations is considered an exercise.

```
NodeT leftmost_child (NodeT n, TreeT  T)
/*  return the leftmost child of node n in tree T */
{
  ListT L;  /* list of child of node n */
  L = T.header[n];
  if (empty(L))   /*  n is leaf */
     return(0);
  else return(retrive ( first(L), L));
}
```

1. THE DATA STRUCTURES REPRESENTS THE TREE

- 1.3. Represent the tree with **the leftmost child and the right sibling**

Comment:

- Each node of the tree is either
  - Childless
  - has exactly 1 leftmost child node
  - don't have right sibling
  - has exactly one right sibling

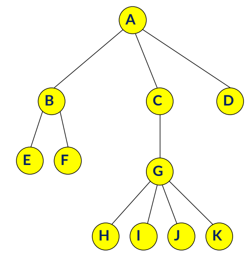
→ To represent a tree: store information about the leftmost child and right sibling of each node.

Use the following description:

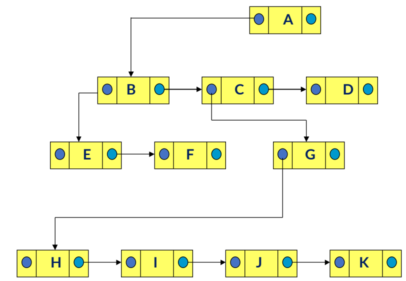
```
struct Tnode
{
    char word[20]; // Data stored in node
    struct Tnode *leftmost_child;
    struct Tnode *right_sibling;
};
typedef struct Tnode treeNode;
treeNode Root;
```


1. THE DATA STRUCTURE REPRESENTS THE TREE

- 1.3. Represent the tree with **the leftmost child and the right sibling**



General tree



Tree representation

1. THE DATA STRUCTURE REPRESENTS THE TREE

- 1.3. Represent the tree with **the leftmost child and the right sibling**

Comment:

- With this representation, basic operations are easy to implement.
- Only the **parent** operation requires traversing the list, so it is inefficient.
  - In cases where this operation must be used frequently, add another field to the record to store the node's parent.

CONTENTS

1. **The data structure represents a tree**

2. Tree traversal

2.1. Preorder

2.2. Postorder

2.3. Inorder

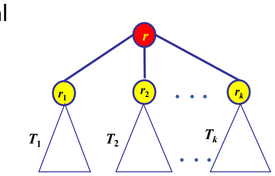
3. Binary tree

2. TREE TRAVERSAL

- Order the nodes
- **Preorder, Postorder and Inorder**
- These orders are defined recursively as follows:
  - If tree T is empty, then the empty list is the preorder, postorder, and middle order list of tree *T*.
  - If tree T has 1 node, then then that node is the pre-ordered, post-ordered, and middle-ordered list of the tree *T*.
  - Otherwise, suppose T is a tree with root r with subtrees T_1, T_2,..., T_k.

2. TREE TRAVERSAL

- **2.1.** Preorder traversal



- ***Preorder traversal*** of tree *T* is:
  - Root *r* of *T*,
  - Next are the nodes of T_1 in the preOrder,
  - Next are the nodes of T_2 in the preOrder,
  - ...
  - And finally, the nodes of T_k in the preOrder.

2. TREE TRAVERSAL

- **2.1.** Preorder traversal

Algorithm:

void PREORDER (*nodeT r*)

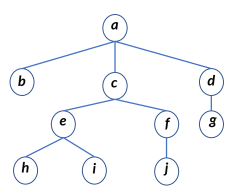
{

(1) Print out *r*;

(2) **for** (each child *c* of *r*, if any, in the order from the left) **do**

PREORDER(*c*);

}

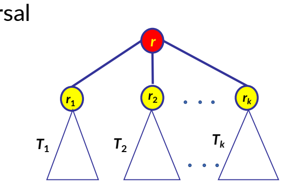


Example: Preorder of nodes on the tree of the figure:

a, b, c, e, h, i, f, j, d, g

2. TREE TRAVERSAL

- **2.2.** PostOrder traversal



- **PostOrder** of nodes on tree *T* :
  - The nodes of T_1 in postOrder,
  - Next are the nodes of T_2 in postOrder,
  - ...
  - Nodes of T_k in postOrder,
  - Finally is the root *r*.

2. TREE TRAVERSAL

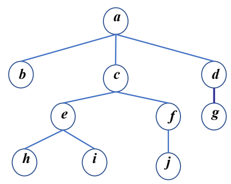
- **2.2.** PostOrder traversal

  Algorithm:

  void POSTORDER (*nodeT r*)
  {
  for (each child *c* of *r*, if any, in the order from the left) **do**
  POSTORDER(*c*)
  Print out *r*;
  }

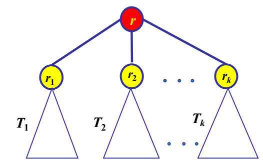
  - **Example:** PostOrder of nodes on the tree of the figure:

    b, h, i, e, j, f, c, g, d, a



2. TREE TRAVERSAL

- **2.3.** InOrder traversal

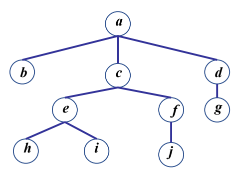


  - ***Inorder*** of nodes on the tree *T*:
    - Nodes of T_1 in InOder,
    - Next is the root *r*,
    - The following are nodes of $T_2, \ldots, T_k$, each group is in InOrder.

2. TREE TRAVERSAL

- **2.3.** InOrder traversal

  void INORDER (*nodeT r*)
  {
  if (*r* is leaf) Print out *r*;
  else
  {
  INORDER(leftmost child of *r*);
  Print out *r*;
  for (each child *c* of *r*, except the leftmost child, in order from the left) **do**
  INORDER(*c*);
  }
  }



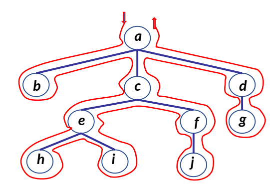
  - **Example:** InOrder of nodes on the tree of the figure:

    b, a, h, e, i, c, j, f, g, d

2. TREE TRAVERSAL

- Order the nodes

  Walk around the tree starting at the root, counterclockwise and following the tree closest



- PreOrder: print out the node every time it is visited.
- PostOrder: print out the node when it was last passed before returning to its parent.
- InOrder: print out *the leaf as soon as it is visited, while internal nodes are printed out the second time it is visited.*
- **Note:** the leaves are arranged in the same order from left to right in all three orders.

CONTENTS

1. The data structure represents a tree

2. Tree traversal

3. Binary tree

3.1. Representation of binary tree

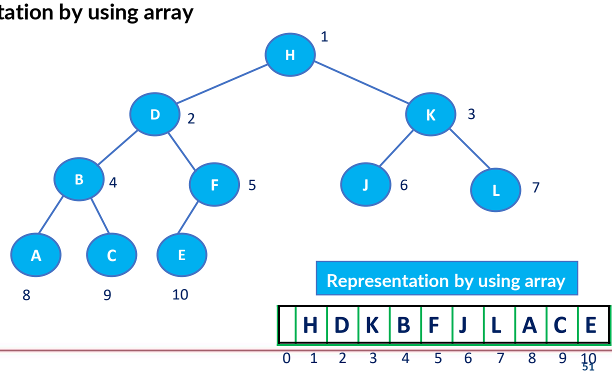
3.2. Tree traversal

3. BINARY TREE

- **3.1.** Representation of binary tree
  - **Representation by using array**
    - Similar as in general tree representation.
    - In the case of a complete binary tree, many operations can be implemented with the tree very efficiently.
      - Consider a complete binary tree *T* with *n* nodes, where each node contains a value.
      - Assign lables to the nodes of the complete binary tree *T* from top to bottom and from left to right using the numbers 1, 2,..., *n*.
      - Let tree *T* correspond to array *A* in which the i^{th} element of A is the value stored in the i^{th} node of tree *T*, *i* = 1, 2, ..., *n*.

3. BINARY TREE

- **3.1.** Representation of binary tree
  - **Representation by using array**



3. BINARY TREE

- **3.1.** Representation of binary tree
  - **Representation by using array**

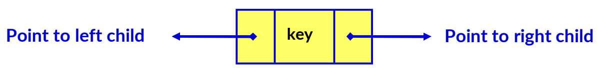
	H	D	K	B	F	J	L	A	C	E
0	1	2	3	4	5	6	7	8	9	10

Find	Use	Limit
Left child of A[i]	A[2*i]	2*i <= n
Right child of A[i]	A[2*i + 1]	2*i + 1 <= n
Parent of A[i]	A[i/2]	i > 1
Root	A[1]	A is not empty
Check A[i] is leaf?	True	2*i > n

3. BINARY TREE

- **3.1.** Representation of binary tree
  - **Representation by using pointer**

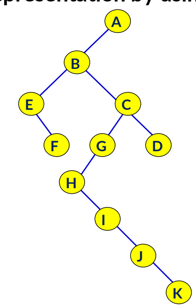
Each node of tree has two pointers that point to left child and right child



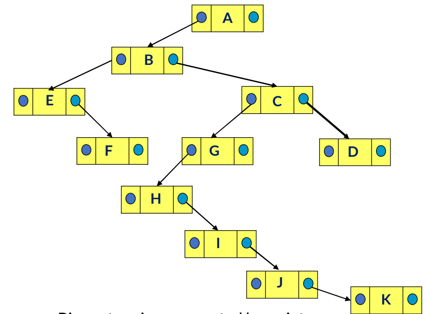

```
struct Tnode {
        DataType Item; // DataType –data type of node on the tree

        struct Tnode *left;
        struct Tnode *right;
};
typedef struct Tnode treeNode;
```


3. BINARY TREE

- **3.1.** Representation of binary tree
  - **Representation by using pointer**



Binary tree



Binary tree is represented by pointers

3. BINARY TREE

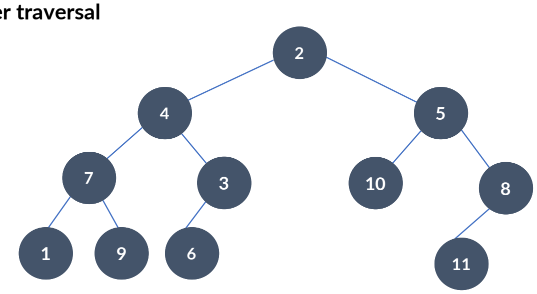
- **3.2. Tree traversal**
  - **PreOrder traversal**

- Visit the node
- Traverse the left subtree
- Traverse the right subtree

```
void printPreorder(treeNode *tree)
{
        if( tree != NULL )
   {
          printf("%s\n", tree->word);
          printPreorder(tree->left);
          printPreorder(tree->right);
   }
}
```


3. BINARY TREE

- **3.2. Tree traversal**
  - **PreOrder traversal**



2, 4, 7, 1, 9, 3, 6, 5, 10, 8, 11

3. BINARY TREE

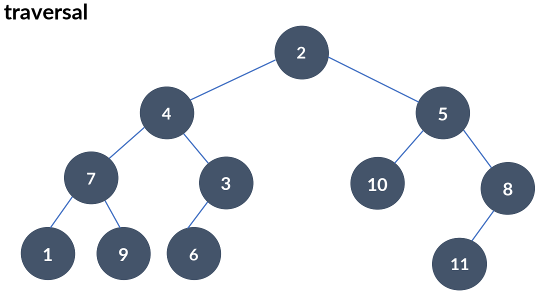
- **3.2. Tree traversal**
  - **InOrder traversal**

- Traverse the left subtree
- Visit the node
- Traverse the right subtree

```
void printInorder(treeNode *tree)
{
        if( tree != NULL )
    {
            printInorder(tree->left);
            printf("%s\n", tree->word);
            printInorder(tree->right);
        }
}
```


3. BINARY TREE

- **3.2. Tree traversal**
  - **InOrder traversal**



1, 7, 9, 4, 6, 3, 2, 10, 5, 11, 8

3. BINARY TREE

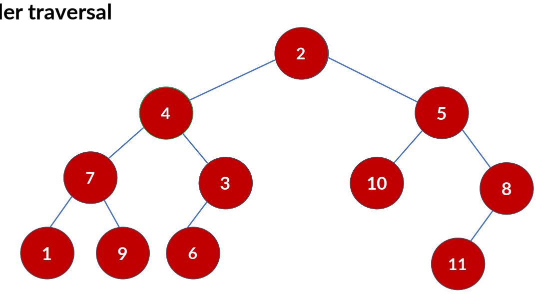
- **3.2. Tree traversal**
  - **PostOrder traversal**

- Traverse the left subtree
- Visit the node
- Traverse the right subtree

```
void printPostorder(treeNode *tree)
{
        if( tree != NULL )
    {
            printPostorder(tree->left);
            printPostorder(tree->right);
            printf("%s\n", tree->word);
        }
}
```


3. BINARY TREE

- **3.2. Tree traversal**
  - **PostOrder traversal**



1, 9, 7, 6, 3, 4, 10, 11, 8, 5, 2

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Chapter 6- Tree

Lesson 3. Calculate the height, depth of the node on the tree

ONE LOVE. ONE FUTURE.

OBJECTIVES

After the lesson, students can

1. Understand the concept of the **height** and **depth** of the internal on tree
2. Implement the algorithm to calculate the height and depth of nodes on the tree

CONTENTS

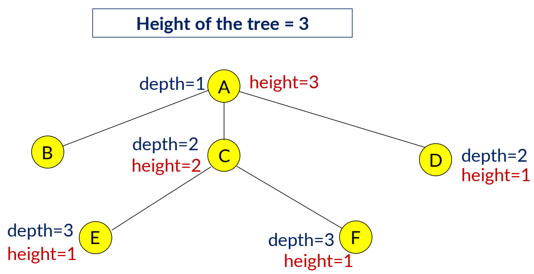
1. **Definitions of height and depth of the node**
   1.1. Definition
   1.2 Example
2. Algorithm to find the height and depth

1. DEFINITION OF THE HEIGHT AND DEPTH OF NODE

- 1.1. Definition
  - **The height** of a node in the tree is equal to the length of the longest path from that node to the leaf plus 1.
    - The height of a tree is the height of the root.
  - **The depth/level** of a node is equal to 1 plus the length of the unique path from the root to it.

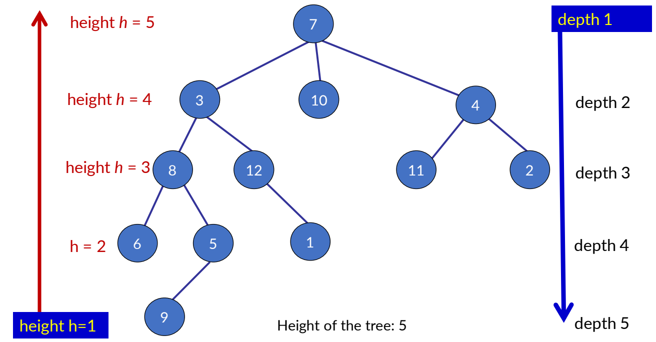
1. DEFINITION OF THE HEIGHT AND DEPTH OF NODE

- 1.2. Example



1. DEFINITION OF THE HEIGHT AND DEPTH OF NODE

- 1.2. Example



CONTENTS

1. Definitions of height and depth of the node

2. Algorithm to find the height and depth

2.1 Height of the node

2.2 Depth of the node

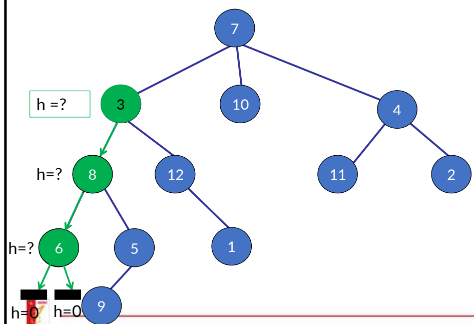
2. ALGORITHM TO FIND THE HEIGHT AND DEPTH

- Tree representation

```
struct Node{
    int id;
    Node* leftMostChild;
    Node* rightSibling;
};
Node* root;
```

2. ALGORITHM TO FIND THE HEIGHT AND DEPTH

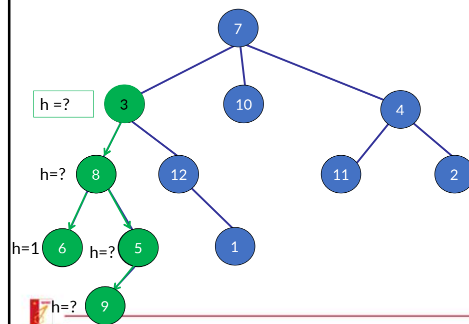
- 2.1. Find the height of a node
  - Height of a node




```
int height(Node* p){
    if(p == NULL) return 0;
    int maxh = 0;
    Node* q = p->leftMostChild;
    while(q != NULL){
        int h = height(q);
        if(h > maxh) maxh = h;
        q = q->rightSibling;
    }
    return maxh + 1;
}
```


2. ALGORITHM TO FIND THE HEIGHT AND DEPTH

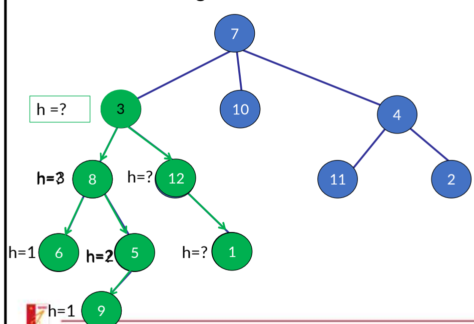
- 2.1. Find the height of a node
  - Height of a node




```
int height(Node* p){
    if(p == NULL) return 0;
    int maxh = 0;
    Node* q = p->leftMostChild;
    while(q != NULL){
        int h = height(q);
        if(h > maxh) maxh = h;
        q = q->rightSibling;
    }
    return maxh + 1;
}
```


2. ALGORITHM TO FIND THE HEIGHT AND DEPTH

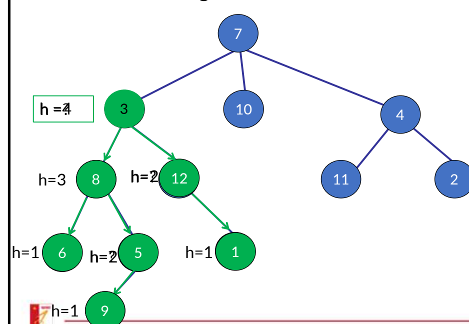
- 2.1. Find the height of a node
  - Height of a node




```
int height(Node* p){
    if(p == NULL) return 0;
    int maxh = 0;
    Node* q = p->leftMostChild;
    while(q != NULL){
        int h = height(q);
        if(h > maxh) maxh = h;
        q = q->rightSibling;
    }
    return maxh + 1;
}
```


2. ALGORITHM TO FIND THE HEIGHT AND DEPTH

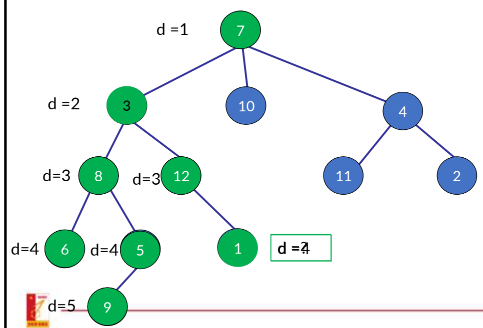
- 2.1. Find the height of a node
  - Height of a node




```
int height(Node* p){
    if(p == NULL) return 0;
    int maxh = 0;
    Node* q = p->leftMostChild;
    while(q != NULL){
        int h = height(q);
        if(h > maxh) maxh = h;
        q = q->rightSibling;
    }
    return maxh + 1;
}
```

2. ALGORITHM TO FIND THE HEIGHT AND DEPTH

- 2.2. Find the depth of a node
  - Depth of a node




```
int depth(Node* r, int v, int d){
    // d is the depth of node r
    if(r == NULL) return -1;
    if(r->id == v) return d;
    Node* p = r->leftMostChild;
    while(p != NULL){
        if(p->id == v) return d+1;
        int dv = depth(p,v,d+1);
        if(dv > 0) return dv;
        p = p->rightSibling;
    }
    return -1;
}
int find_depth(Node* r, int v){
    return depth(r,v,1);
}
```


THANK YOU !