Data and File Structures Laboratory

Binary Trees

Malay Bhattacharyya

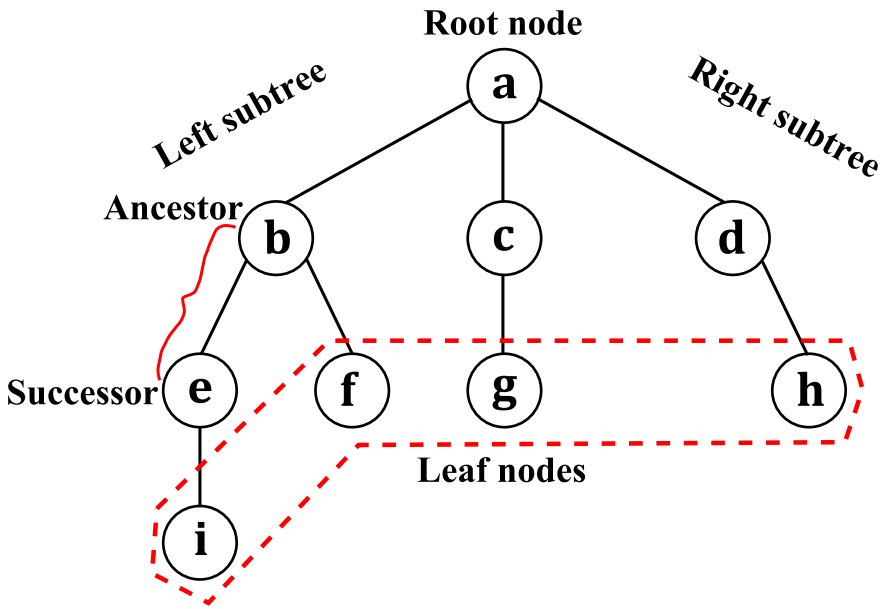
Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

September, 2019

Basics of a tree

A tree is recursively defined as a set of one or more nodes where a single node is designated as the root of the tree and all the remaining nodes can be partitioned into subtrees of the root.



Note: Tree is a non-linear data structure (a data item can be linked to more than two data items).

Types of trees

Trees can be of different types based on their use as data structures. Some of these are listed below.

- **Forests:** Disjoint union of trees.
- **Binary trees:** Trees having at most two subtrees per node.
- **Binary search trees:** Binary trees in which the nodes are arranged in an order (based on the data items that they keep).
- **Expression trees:** Trees that are used to store algebraic expressions.
- **Tournament trees:** Trees that are used to represent players using the leaf nodes and winners using the remaining nodes.

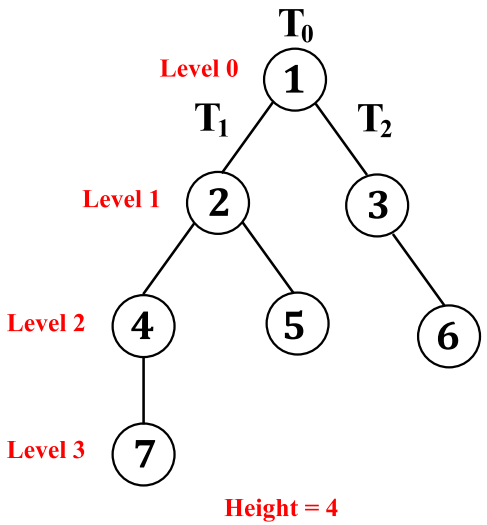
Note: Trees are data structures that are mainly used to store data items (labels or keys on the nodes) hierarchically.

Binary trees

Definition (Binary Tree)

A binary tree T over a domain D is either an empty set (empty binary tree) or a recursively-defined triplet $\langle T_0, T_1, T_2 \rangle$ such that

- T_0 is a node over D (root), and
- T_1 and T_2 are binary trees over D (left and right subtree, respectively).



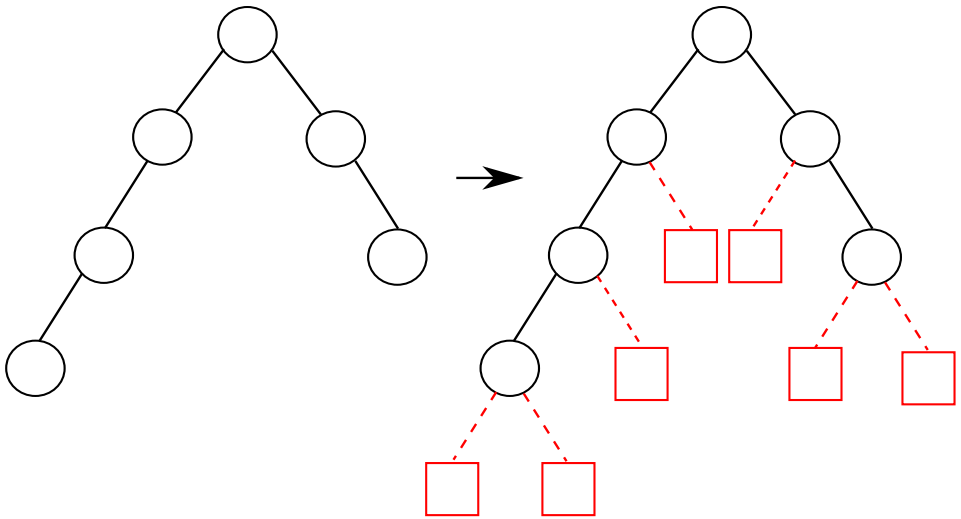
Terminologies in binary trees

- **Left child:** Left successor node of a node.
- **Right child:** Right successor node of a node.
- **Parent:** Ancestor node of a set of children.
- **Sibling:** Nodes that are at the same level and share the same parent.
- **Degree of a node:** Number of children of a node.

Note: Every node other than the root node has a parent.

Extended binary trees

A binary tree is turned into an *extended binary tree* by adding special nodes to those nodes, except the root node, which contain a null subtree (left or right) in the original tree.

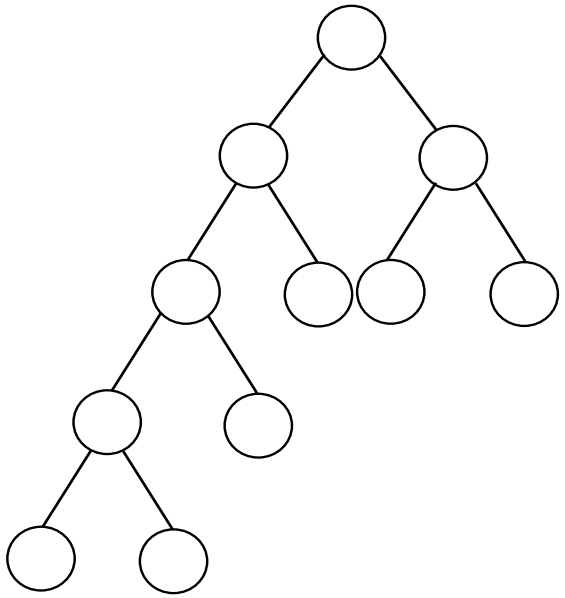


Note: The extended nodes are distinguishable from original nodes.

Full binary trees

Definition (Full Binary Tree)

A binary tree is said to be full binary tree (or 2-tree) if each node in the tree has either no child or exactly two children.



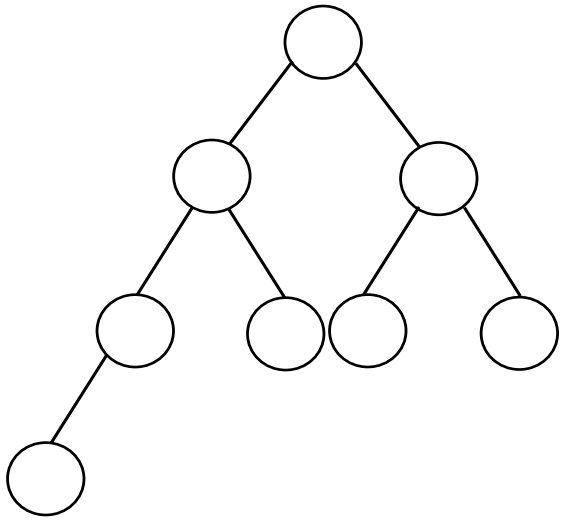
Note: An extended binary tree becomes a full binary tree.

Complete binary trees

Definition (Complete Binary Tree)

A binary tree is said to be complete binary tree if it satisfies both of the following:

- Every level, except possibly the last, is completely filled.
- All nodes appear as far left as possible.



Complete binary trees

Some properties:

1. A complete binary tree of height h comprises at least 2^h and at most $2^{h+1} - 1$ nodes.
2. The height of a complete binary tree having exactly n nodes is $\lceil \log_2(n+1) \rceil$.
3. The maximum possible number of parents with absent children in a complete binary tree having n nodes is $(n+1)$.
4. The number of internal nodes in a complete binary tree having n nodes is $\lfloor n/2 \rfloor$.

Conventional implementation of binary trees

- Individual dynamic memory allocation per node.
- The allocated memories (to the nodes) may be scattered all over the heap area.

```
typedef struct treeNode{
    DATA d;
    struct treeNode *leftChild, *rightChild;
    struct treeNode *parent; // This is optional
}BTNODE;
```

Note: The token `DATA` symbolizes the data type accommodated.

Conventional implementation of binary trees

Adding a new node:

```
BTNODE * addNode(int data){
    BTNODE *node;
    if(NULL == (node = Malloc(1, BTNODE))) // PS: common.h
        ERR_MESG("out of memory");
    node->d = data;
    // Initialize left and right child as NULL
    node->leftChild = NULL;
    node->rightChild = NULL;
    return(node);
}
```

Note: The new data is kept with NULL left and right pointers.

Conventional implementation of binary trees

Creating the root node:

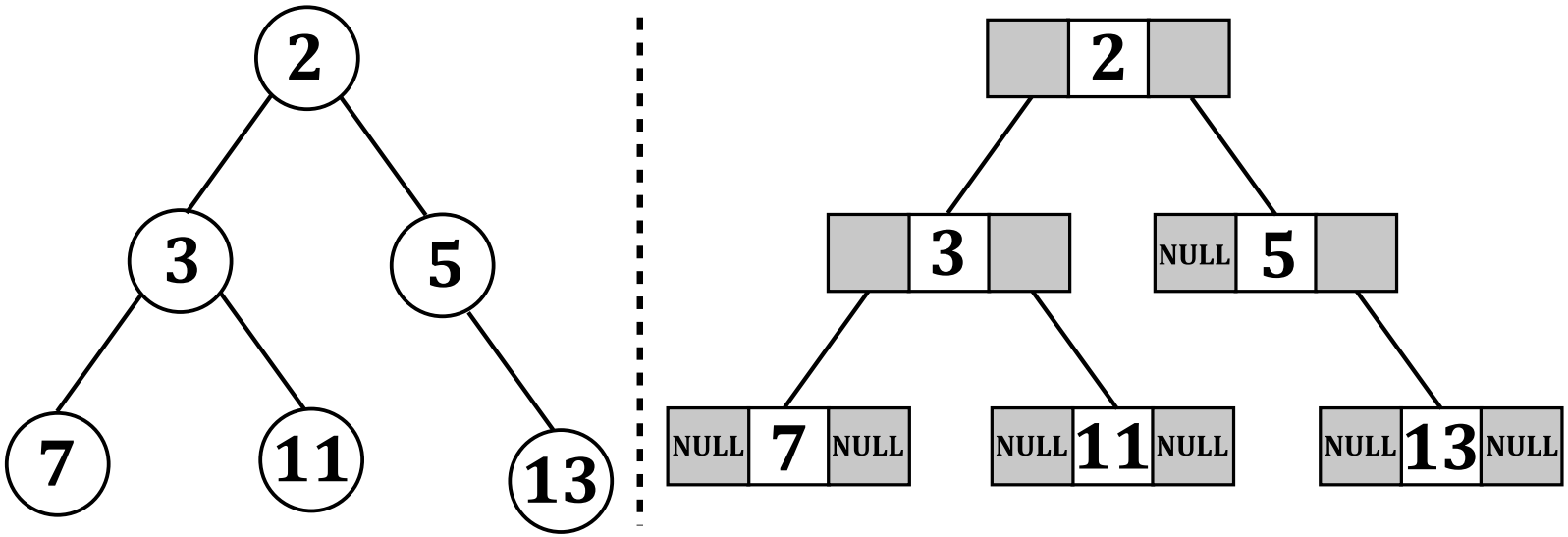
```
int d1, d2, d3;
BTNODE *root = addNode(d1);
```

Adding nodes to the root node:

```
root->left = addNode(d2);
root->right = addNode(d3);
```

Note: The root is `NULL` for an empty tree.

Conventional implementation – An example



Alternative implementation of binary trees

- An initial dynamic memory allocation and successive reallocations as and when required.
- The allocated memories (to the nodes) are located within the same array.

```
typedef struct treeNode{
    DATA d;
    int leftChild, rightChild;
    int parent; // This is optional
}BTNODE;
```

Note: The token `DATA` symbolizes the data type accommodated.

Alternative implementation – An example

Initially: root = NULL

	d	leftChild	rightChild
free → 0	–	1	NULL
1	–	2	NULL
2	–	3	NULL
3	–	4	NULL
⋮	⋮	⋮	⋮
$n-1$	–	NULL	NULL

Note: The value NULL is treated as '-1' during implementation.

Alternative implementation – An example

Finally:

	d	leftChild	rightChild
root → 0	2	1	2
1	3	3	4
2	5	NULL	5
3	7	NULL	NULL
4	11	NULL	NULL
5	13	NULL	NULL
free → 6	–	7	NULL
⋮	⋮	⋮	⋮
$n-1$	–	NULL	NULL

Note: The value NULL is treated as '-1' during implementation.

Sequential implementation of binary trees

- One time memory allocation for a one-dimensional array.
- The left and right child of the node at index i are at index $2i$ and $2i + 1$, respectively.
- It might involve a lot of unused allocated memories.

```
int h; // Keeps the height
typedef struct treeNode{
    DATA d[POWER(2, h)]; // POWER(a, b) computes a^b
}BTNODE;
```

Note: A binary tree having height h can have at most $(2^h - 1)$ nodes.

Sequential implementation – An example

		d	
	0	NULL	
root →	1	2	
	2	3	
	3	5	parent (at index i)
	4	7	
	5	11	
	6	NULL	no left child (at index $2i$)
	7	13	right child (at index $2i + 1$)

Note: The value NULL is treated as '-1' during implementation.

Traversal of a binary tree

- Pre-order traversal
- In-order traversal
- Post-order traversal
- Level-order traversal

Pre-order traversal

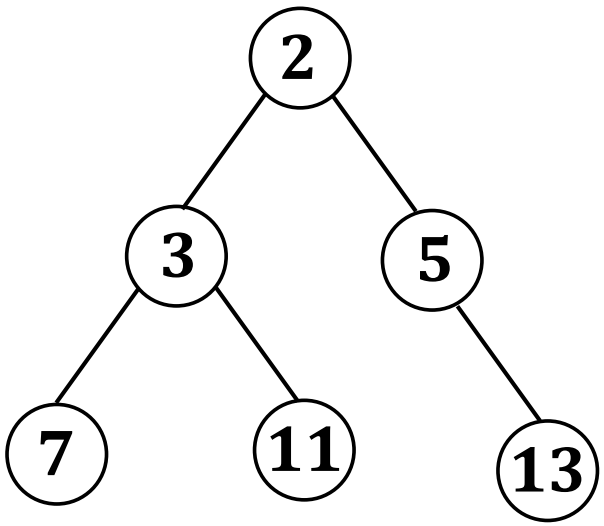
Perform the following operations recursively at each node:

1. Visit the root node.
2. Traverse the left subtree.
3. Traverse the right subtree.

Pre-order traversal

Perform the following operations recursively at each node:

1. Visit the root node.
2. Traverse the left subtree.
3. Traverse the right subtree.



Order of visited nodes: 2, 3, 7, 11, 5, 13

Pre-order traversal

```
int preOrder(BTNODE *node){
    if(NULL == node) // Reached a leaf node
        return 0;
    // Visit the node
    PRINT(node->d);
    preOrder(node->leftChild);
    preOrder(node->rightChild);
}
```

In-order traversal

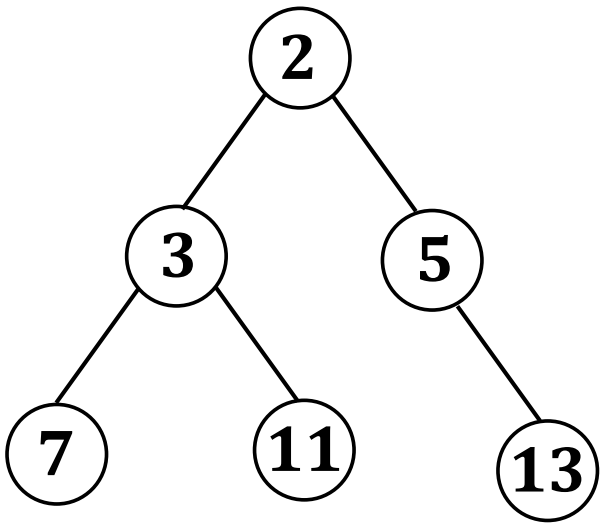
Perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Visit the root node.
3. Traverse the right subtree.

In-order traversal

Perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Visit the root node.
3. Traverse the right subtree.



Order of visited nodes: 7, 3, 11, 2, 5, 13

In-order traversal

```
int inOrder(BTNODE *node){
    if(NULL == node) // Reached a leaf node
        return 0;
    // Visit the node
    inOrder(node->leftChild);
    PRINT(node->d);
    inOrder(node->rightChild);
}
```

Post-order traversal

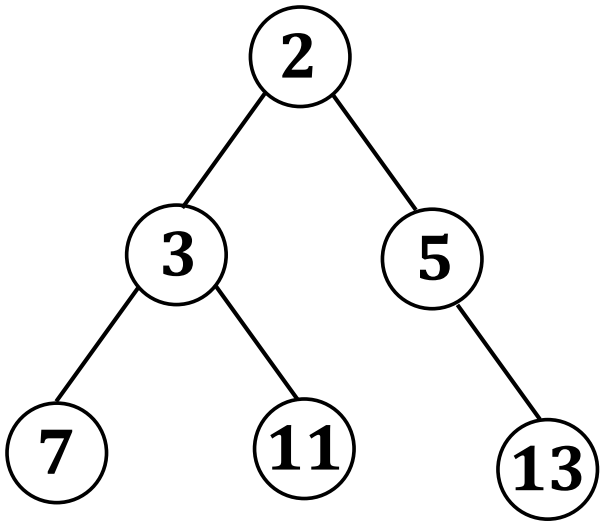
Perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root node.

Post-order traversal

Perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root node.



Order of visited nodes: 7, 11, 3, 13, 5, 2

Post-order traversal

```
int postOrder(BTNODE *node){
    if(NULL == node) // Reached a leaf node
        return 0;
    // Visit the node
    postOrder(node->leftChild);
    postOrder(node->rightChild);
    PRINT(node->d);
}
```

Level-order traversal

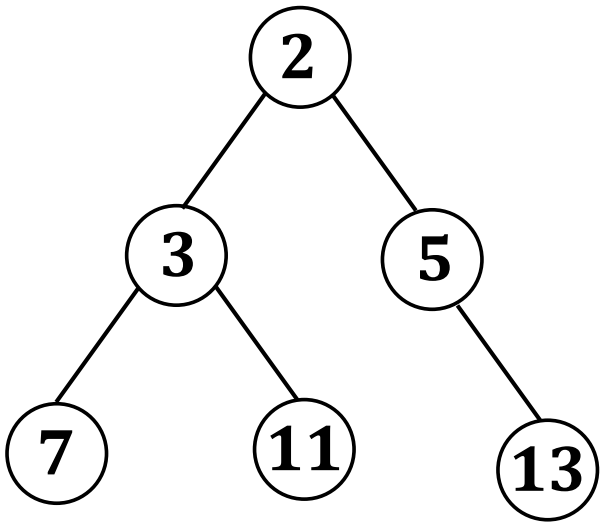
Perform the following operations recursively at each node:

1. Visit the root node.
2. Traverse all the subtrees from left to right at the next level.

Level-order traversal

Perform the following operations recursively at each node:

1. Visit the root node.
2. Traverse all the subtrees from left to right at the next level.



Order of visited nodes: 2, 3, 5, 7, 11, 13

Constructing binary trees from traversal results

In-order traversal: 7, 3, 11, 2, 5, 13
Pre-order traversal: 2, 3, 7, 11, 5, 13

1. Consider the first data item visited in pre-order traversal as the root node.
2. Data items on the left and right side of the root node in the in-order traversal sequence form the left and right subtree of the root node, respectively.
3. Recursively select each data item from pre-order traversal sequence and create its left and right subtrees from the in-order traversal sequence.

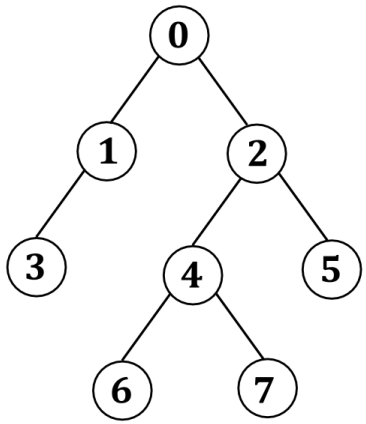
Constructing binary trees from traversal results – An example

In-order traversal: 7, 3, 11, 2, 5, 13
Pre-order traversal: 2, 3, 7, 11, 5, 13

- **Step 1:** Data item '2' is the root node (first in pre-order).
- **Step 2:** Left and right subtree of '2' comprises $\{7, 3, 11\}$ and $\{5, 13\}$, respectively (items on the left and right of '2' in in-order).
- **Step 3:** '3' and '5' are the root nodes of the left and right subtree of '2' (first of those data items in pre-order).
- **Step 4:** Left and right subtree of '3' comprises $\{7\}$ and $\{11\}$, respectively (items on the left and right of '3' in in-order).
- **Step 5:** Left and right subtree of '5' comprises $\emptyset$ and $\{13\}$, respectively (items on the left and right of '5' in in-order).
- **Step 6:** Subtrees of other nodes are $\emptyset$.

Problems – Day 13

1. Let a first-n natural binary tree (FnBT) is defined as a binary tree having the first n natural numbers as its data items. Consider a special representation of FnBTs given in the form of an array A such that the parent of node i is $A[i]$. Note the correspondence between the index of elements in A and the data items in the tree. For the root node, the parent is denoted by '-1'. E.g., the array representation $A = \{-1, 0, 0, 1, 2, 2, 4, 4\}$ corresponds to the following FnBT. Write a program to construct the FnBT from an input array.



Problems – Day 13

2. Suppose a complete binary tree is represented with an array. In this array, the root node is placed at index 0. The left and right child of any arbitrary node at the index i is placed at the index $2i + 1$ and $2(i + 1)$, respectively. Write a program to construct the tree hierarchically by taking its data items as an array from the user and print the pre-order traversal result.
3. In a tournament tree, each non-leaf node represents the winner of the match played between the players represented by its children nodes. Such trees are used to record the winner at each level up to the root (the overall winner). Given a binary tree, write a program to verify whether it is a tournament tree or not.
4. Write a program to construct a binary tree given (from the user) its pre-order and post-order traversal results. Note that, the constructed tree will not be unique.

Problems – Day 13

5. Write non-recursive functions for the pre-order, in-order and post-order traversals of a given binary tree. You may reuse your own implementation of a stack for this program.
6. Given a binary tree with integer-valued nodes, and a *target* value, determine whether there exists a root-to-leaf path in the tree such that the sum of all node values along that path equals the target. Modify your program to consider all *paths*, not just root-to-leaf paths.
7. Given a binary tree stored in an array (using the alternative implementation), and the indices of two nodes in the tree, find the index of the node that is the lowest common ancestor of the given nodes.