Temasek Junior College

2023 JC2 H2 Computing

Data Structures 6: Binary Trees and Binary Trees Traversal

1 Introduction to the Tree Data Structure

A **tree** is a non-linear data structure that consists of a set of nodes connected by edges in a hierarchical organisation. It is an undirected graph with no cycles.

In the study of the tree data structure, we need to keep in mind some important terminologies.

Size:
The **size** of a tree is equal to the number of nodes ***(N)*** in a tree.

Edges:
- An edge is a connection between two nodes.
- The number of edges ***(E)*** of a tree is equal to the number of nodes ***(N)*** minus one, so ***E = N - 1***.

Root:
- The root of a tree is the starting node for traversals.
- If the tree has a root, it is called a **rooted tree**.

Branch:
- A **branch** is a path from the root to an end point.

Leaf:
- The end point of a branch is called a **leaf**.

Height:
- The **height** of a tree is equal to the number of edges (connections) that connect the root node to the leaf node that is furthest away from it (i.e. the longest branch).

Depth:
- The number of edges from the root to a node is called the **depth** of the node**.**

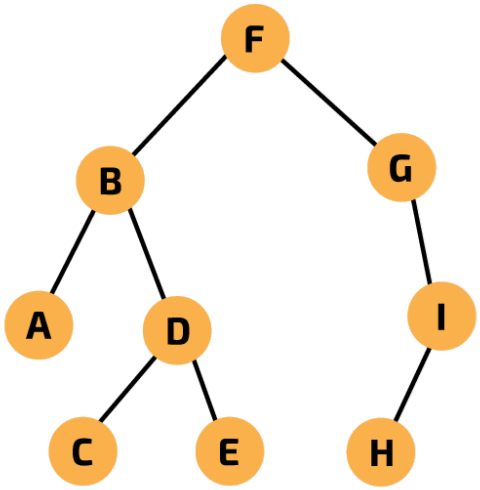
Connected:
- There is a path from any node to any other node.
- There is no node, or set of nodes, that is disconnected from the others.
- Any two nodes may either be directly connected by an edge or indirectly connected (with multiple nodes between them forming a path).

Undirected:
- There is no direction associated with an edge

Cycle:
- A path that starts and ends at the same node.
- Since a tree has no cycles, there is no closed path between any two nodes (there may be multiple nodes in between the selected starting node and ending node, but there is no path leading back to the starting node from the ending node)

Question

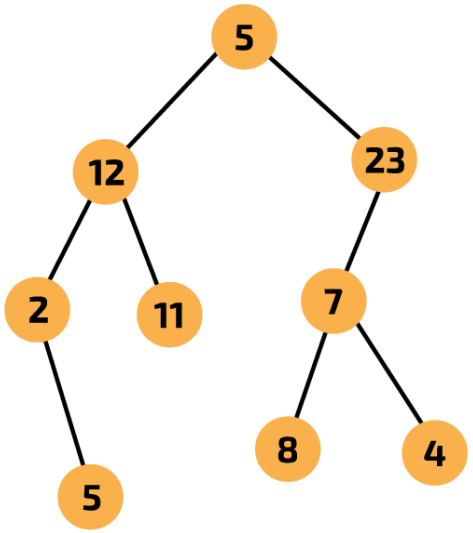
The diagram below shows a tree structure with node F as the root.



(a) What is the size of the tree?
(b) What is the height of the tree?
(c) What is the depth of node E?
(d) What is the depth of node A?
(e) What is the total number of edges?

2 Rooted Trees

A **rooted tree** is a tree with one node that has been designated as the root. Unlike real trees, when representing rooted trees in a diagram, the root is commonly situated above the other nodes and the branches descend to the **leaf** nodes.



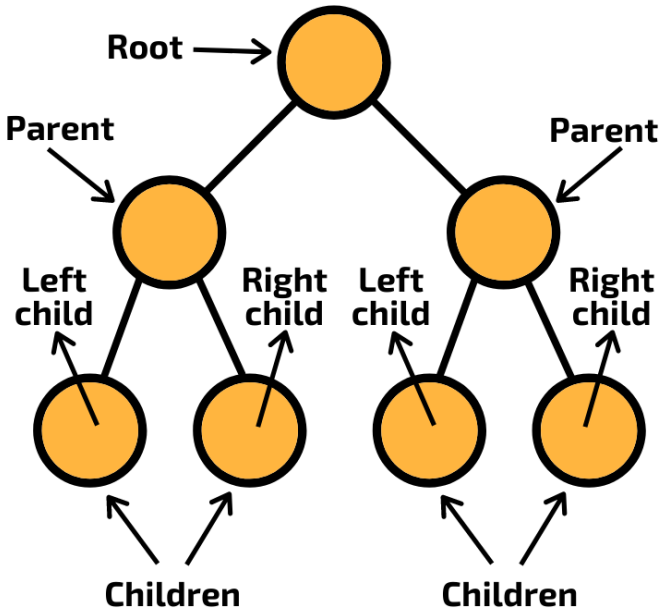
In a rooted tree, the nodes are connected by parent–child relationships. If you mark a path from the root towards a node, a **parent** node is a node that comes directly before another node in the path, which is considered its **child**. A node can have any number of children. A **leaf** is a node with no

children. In the diagram above, the node marked 12 is the parent of the nodes marked 2 and 11. The node marked 5 is a leaf node. Which other nodes are leaf nodes?

It follows that the root is the only node with no parent, and all other nodes are descendants of the root.

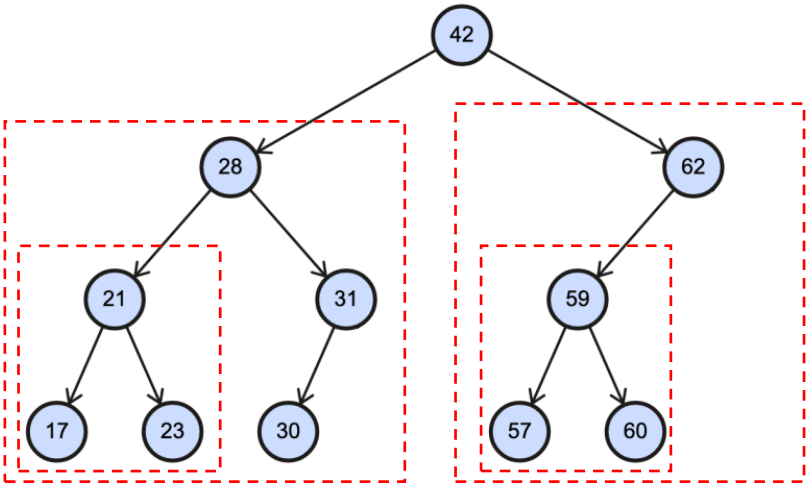
3 Binary Trees

A **binary tree** is type a rooted tree structure.



In a binary tree:

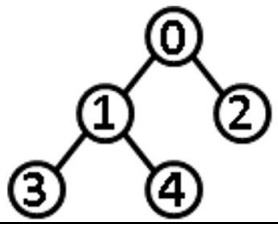
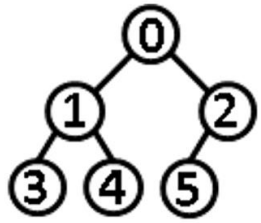
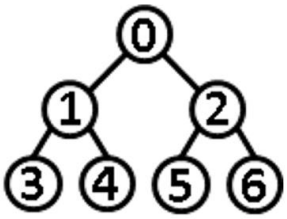
- Every node has one parent, except for the root.
- Every node can have **at most two** child nodes.
- Child nodes from the same parent are called siblings.
- Nodes with no child nodes are leaf nodes.
- Within the hierarchical structure, a parent node can have at least one sub-tree, which is also a binary tree.



A **full binary tree** is a binary tree where every node has two children except for the leaves.

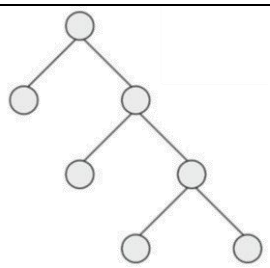
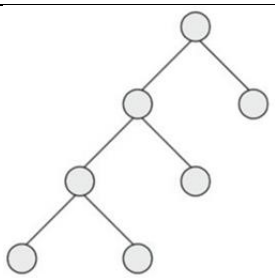
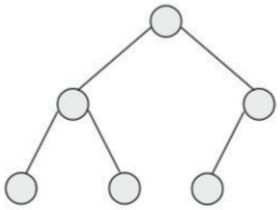
A **complete binary tree** is a binary tree where every level, except possibly the last, is completely filled. Nodes are connected as far left as possible.

A **perfect binary tree** is a binary tree where every level, including the last level, is full with nodes.

		
Full Binary Tree	Complete Binary Tree	Perfect binary Tree

Keep in mind these following traits:

- A full tree is not always complete or perfect, but there are instances where a full tree can **also** be complete *OR* perfect.
- A complete tree is not always full — some nodes can have one child, but a full tree, by nature, cannot have a node with one child, only 0 or 2. A complete tree can never be perfect.
- A perfect tree is always complete and full — each node has 2 children, but the last level has 0 children.

		
Full binary tree Not complete Not perfect	Full binary tree Not complete Not perfect	Complete binary tree Not full Not perfect

4 Binary Trees Traversals

Processing the data in a binary tree often requires you to traverse the tree which means systematically visiting every node once.

Three common algorithms for traversing trees are the **in-order**, **pre-order** and **post-order**. Each method has specific uses. The tree can be traversed from left to right or right to left, depending on the application.

Possible applications include:

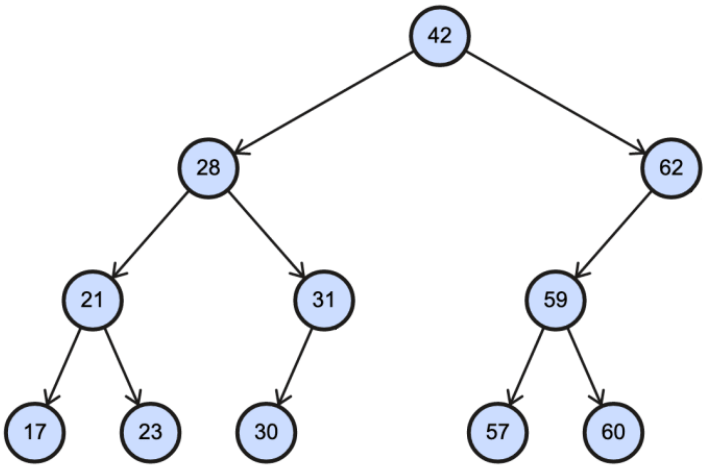
- Printing all values in a binary tree
- Determining if there is one or more nodes with some property
- Making a copy

4.1 In-order traversals

In an in-order traversal, each node is visited **between** each of its subtrees. The simplified algorithm for a left-to-right in-order traversal is:

- First, visit the left subtree of the node.
- Then, visit the node itself.
- Then visit the right subtree of the node.

Consider the following binary tree:



Following the algorithm, we have the following order:

17 → 21 → 23 → 28 → 30 → 31 → 42 → 57 → 59 → 60 → 62

Are you able to see the pattern for the in-order traversal?

The following is a subroutine in pseudocode for the in-order traversal:

```
SUBROUTINE in_order(node)

    IF node.left ≠ -1 THEN
       in_order(node.left)
    ENDIF

    PRINT(node.data)

    IF node.right ≠ -1  THEN
       in_order(node.right)
    ENDIF

ENDSUBROUTINE
```

In the pseudocode:

- `node.data` will return the value of the node
- `node.left` will return the value of the left child node
- `node.right` will return the value of the right child node
- A return value of `-1` will indicate that there is no child node

The algorithm is recursive, so it is repeatedly called until all of the nodes have been processed.

The may be times you may be presented with a right to left traversal, in which case, the order will be reversed. Check the pseudocode carefully to see which side is handled first.

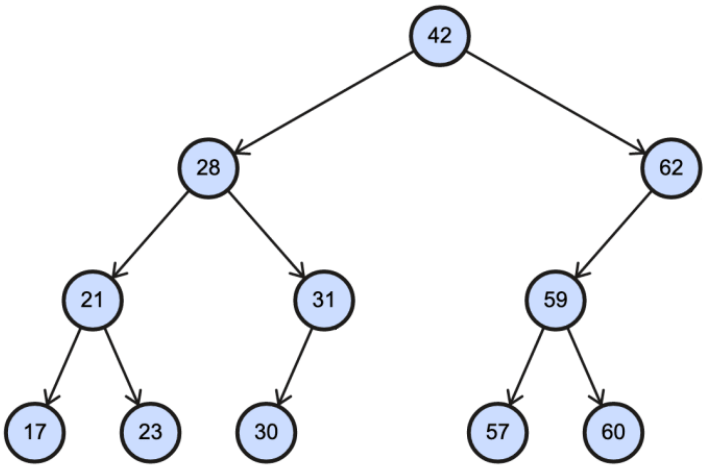
4.2 Pre-order Traversals

The term **pre-order** specifies the point in the traversal at which the nodes contents are processed.

In a pre-order traversal, each node is visited **before** the algorithm traverses either of the node's subtrees. The simplified recursive algorithm for a left-to-right pre-order traversal is:

- First visit the node itself.
- Then visit the left subtree of the node.
- Then visit the right subtree of the node.

Consider the following binary tree:



Following the algorithm, we have the following order:

42 → 28 → 21 → 17 → 23 → 31 → 30 → 62 → 59 → 57 → 60

Are you able to see the pattern for the pre-order traversal?

The following is a possible subroutine in pseudocode for the pre-order traversal:

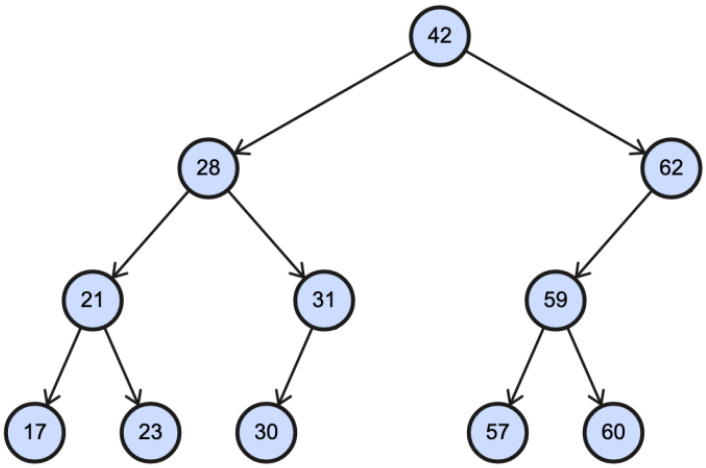
```
SUBROUTINE pre_order(node)

    PRINT(node.data)

    IF node.left ≠ -1 THEN
      pre_order(node.left)
    ENDIF

    IF node.right ≠ -1  THEN
      pre_order(node.right)
    ENDIF

ENDSUBROUTINE
```

4.3 Post-order Traversals

In a post-order traversal, each node is visited **after** both of its subtrees. The simplified algorithm for a left-to-right post-order traversal is:

- First visit the left subtree of the node.
- Then visit the right subtree of the node.
- Then visit the node itself.

Consider the following binary tree:



Following the algorithm, we have the following order:

17 → 23 → 21 → 30 → 31 → 28 → 57 → 60 → 59 → 62 → 42

Are you able to see the pattern for the post-order traversal?

The following is a subroutine in pseudocode for the post-order traversal:

```
SUBROUTINE post_order(node)

    IF node.left ≠ -1 THEN
       post_order(node.left)
    ENDIF

    IF node.right ≠ -1  THEN
       post_order(node.right)
    ENDIF

    PRINT(node.data)

ENDSUBROUTINE
```