# Tree data structures and tree traversals

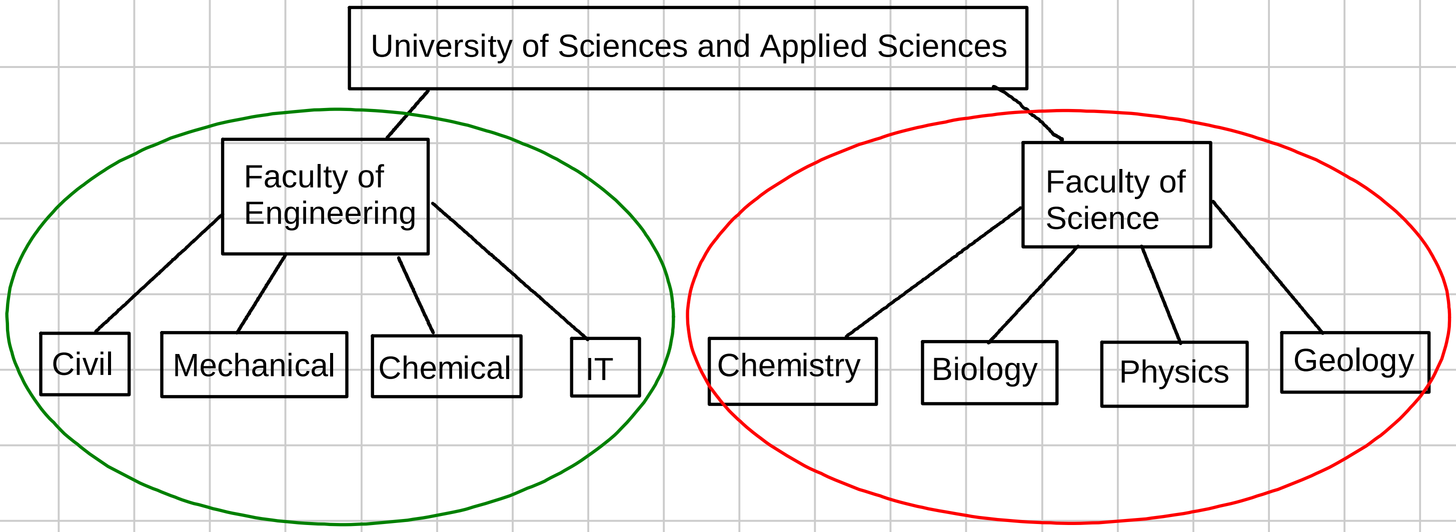
**1. Background**
**2. Data structure**
**3. Traversals**

## 1. Background

Why does one need to know how to form and use a tree data structure?

- STL does not have 'tree' in its containers.

- A tree is useful for storing data that is hierarchic.



Tree parts: nodes, edges, root, children, parents, leaves (external node), internal node, subtree

Note:

1. All trees are assumed to be rooted trees.

2. Children and subtrees are ordered from left to right.  Hence the leftmost child is the first child and the rightmost child is the last child.

## 2. Data structure

Usually in a tree the nodes hold the information.

We need a device (a data type) to hold node data.  We will use a *Node*.

In pseudocode we will use the following for general tree:

• *Node.key* = the key of *Node*

• *Node.parent* = pointer to parent of *Node* or NIL if root

• *Node.children* = all children of *Node*

• *Node.children*[*i*] = pointer to *i*'th child of *Node*, or NIL if child does not exist

• The root node is *root*.

In pseudocode we will use the following for binary tree:

• *Node.key* = the key of *Node*

• *Node.parent* = pointer to parent of *Node* or NIL if root

• *Node.left* = pointer to left child or NIL if child does not exist

• *Node.right* = pointer lto right child or NIL if child does not exist.

• The root node is *root*.

**NOTE**

• individual node can be accessed by its key

In C++ for general tree:

```cpp
struct Node
{
 // key = a unique identifier for the node
 int key;
 // name = name we give to the node
 std::string name;
 // children = pointers to the children of the node
 std::vector<Node*> children;
 // parent = pointer to the parent of the node
 Node* parent;
}
```

In C++ for binary tree:

```cpp
struct Node
{
 // key = a unique identifier for the node
 int key;
 // name = name we give to the node
 std::string name;
 // leftChild = pointer to the left child
 Node* leftChild;
 // rightChild = pointer to the left child
 Node* rightChild;
 // parent = pointer to the parent of the node
 Node* parent;
}
```

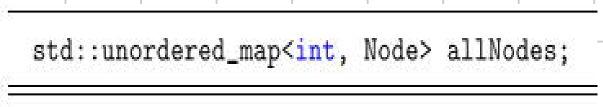
**NOTES**

- If node does not exist, then pointer value is `nullptr`.

- Individual node is accessed by its `key`.

- If a node can have many children, then a `vector` container is probably inefficient. Some other container should be used, e.g. `underordered_set`.

- All tree nodes should be stored in a container. For example:



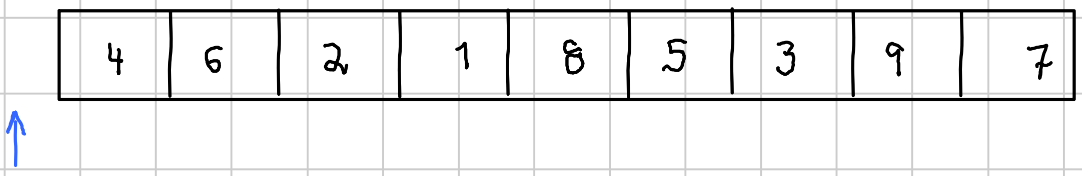
```
std::unordered_map<int, Node> allNodes;
```

## 3. Traversals

**Q:** What is a traversal?

**A:** When all of the elements of a data structure are visited.

Single dimensional array traversal:



We consider three tree traversals: preorder traversal, postorder traversal and inorder traversal.

These three differ in the order in which the nodes are visited.

**Preorder traversal**

First visit the parent and then visit its subtrees in order.

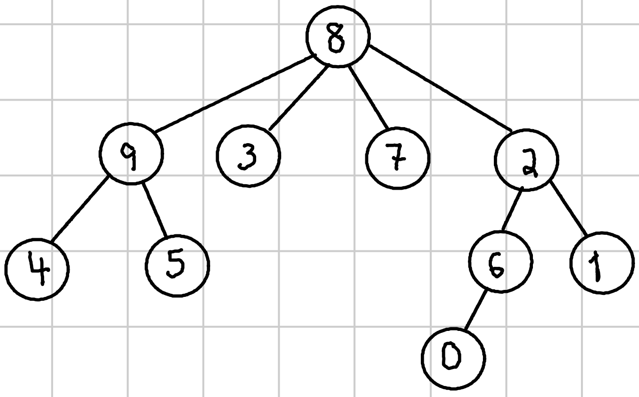
```
1  PRETRAVERSAL(node)
2  input node is some rooted tree node
3  if $node \neq NIL$ then
4     ▷ perform some operation using node
5     forEach child in node.children
6        PRETRAVERSAL(child)
7     end
8  end
```

Assume: operation in line 4 is outputting node key to user

**Example**

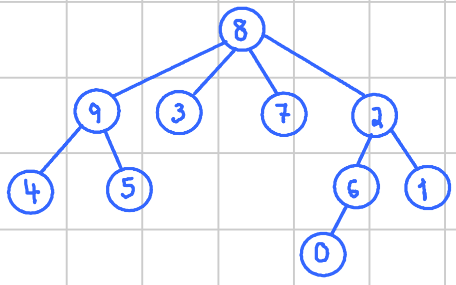
Compute

PRETRAVERSAL( 8 )



| recursion level | line | computation |
|---|---|---|
| 1 | 4 | output 8 |
| 1 | 6 | PRETRAVERSAL( 9 ) |
| 2 | 4 | output 9 |
| 2 | 6 | PRETRAVERSAL( 4 ) |
| 3 | 4 | output 4 |
| 2 | 6 | PRETRAVERSAL( 5 ) |
| etc | | |

order of all output: 8, 9, 4, 5, 3, 7, 2, 6, 0, 1



□

**Postorder traversal**

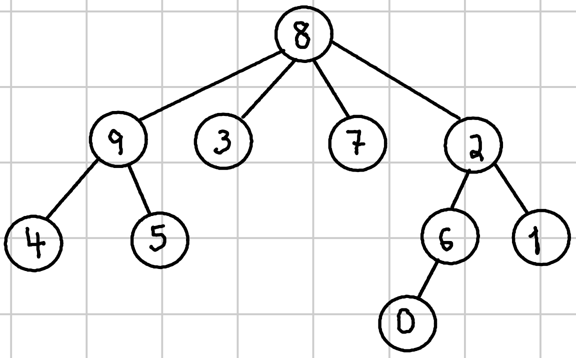
First visit the subtrees in order and then visit the parent.

```
1  POSTTRAVERSAL(node)
2  input node is some rooted tree node
3  if node ≠ NIL then
4     forEach child in node.children
5        POSTTRAVERSAL(child)
6     end
7     ▷ perform some operation using node
8  end
```

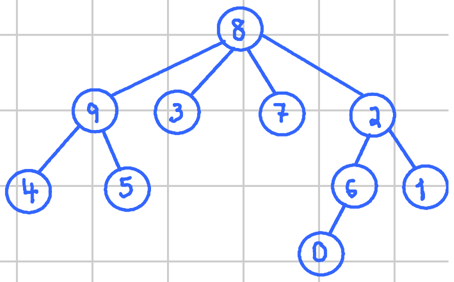
**Example**

Compute

POSTTRAVERSAL( ⑧ )

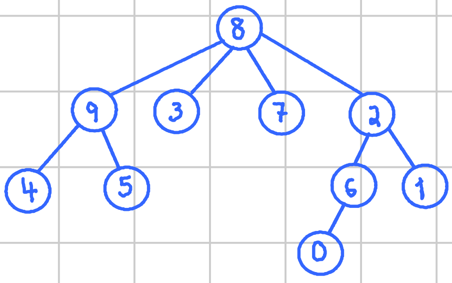


| recursion level | line | computation |
|---|---|---|
| 1 | 5 | POSTTRAVERSAL( ⑨ ) |
| 2 | 5 | POSTTRAVERSAL( ④ ) |
| 3 | 7 | output 4 |
| 2 | 5 | POSTTRAVERSAL( ⑤ ) |
| 3 | 7 | output 5 |
| 2 | 7 | output 9 |



etc

order of all output: 4, 5, 9, 3, 7, 0, 6, 1, 2, 8



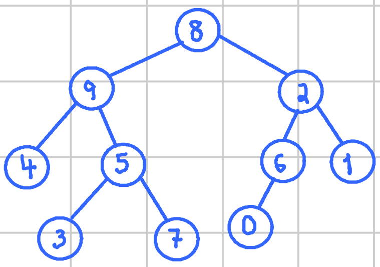
□

**Inorder traversal**

This order is only valid for binary trees.

First the left subtree is visited, then the parent is visited, and finally the right subtree is visited.

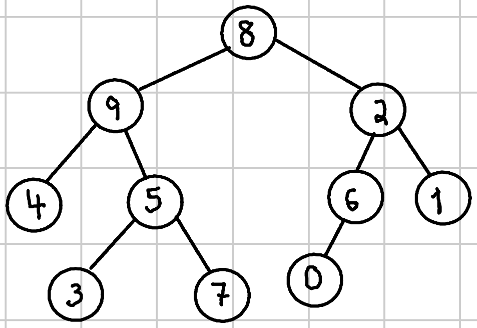
```
1  INTRAVERSAL(node)
2  input node is some rooted tree node
3  if node ≠ NIL then
4     INTRAVERSAL(node.left)
5     ▷ perform some operation using node
6     INTRAVERSAL(node.right)
7  end
```



**Example**

Compute

INTRAVERSAL( $\textcircled{8}$ )



| recursion level | line | computation |
|---|---|---|
| 1 | 4 | INTRAVERSAL( $\textcircled{9}$ ) |
| 2 | 4 | INTRAVERSAL( $\textcircled{4}$ ) |
| 3 | 5 | output 4 |
| 2 | 5 | output 9 |
| 2 | 6 | INTRAVERSAL( $\textcircled{5}$ ) |

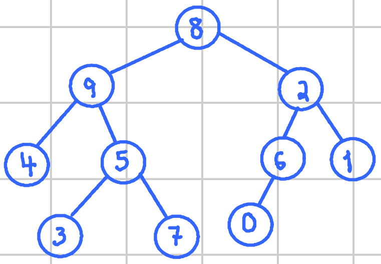
3 4

4 5

etc

INTRAVERSAL( $\textcircled{3}$ )

output 3



order of all output: 4, 9, 3, 5, 7, 8, 0, 6, 2, 1

□

**NOTES**

- PRETRAVERSAL, POSTTRAVERSAL and INTRAVERSAL are all recursive.

- Preorder traversal is useful when some information about the parent should be available or relayed to the children.

- Postorder traversal is useful when some information about the children should be available or relayed to the parent.

- Inorder traversal is useful for sorting all nodes by theirs keys when using a binary search tree.