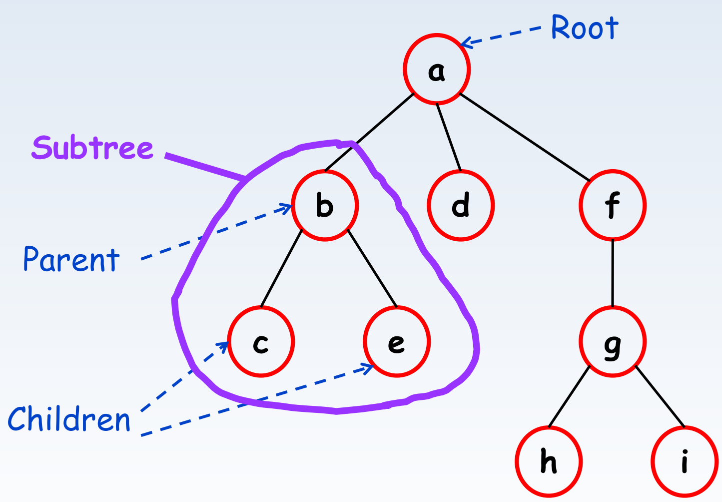
Data Structures

Trees

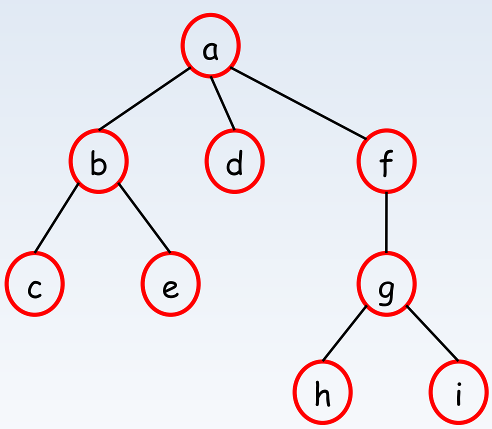
Tree Definition

- The tree is a finite set of nodes.
  - There is a special node called the root.
  - The remaining nodes make up *n* separate, disjoint sets.
  - Each set has a separate tree structure.
  - These sets are called subtrees.
- The nodes one level below each node are the child nodes of that node.
- The node located one level above a node on the way to the root is the parent node.

Example



Definitions



Degree: the number of subtrees the root has in a subtree or tree.

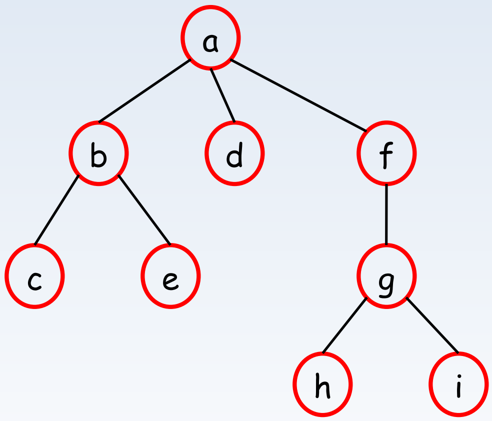
Example:

- a → 3
- f → 1

Leaf: Node with degree 0.

Example: c, d, e, h, i

Definitions



Level: Root is assumed to be at level 1 and the level increases down the tree.

Example:

- b: 2
- h: 4

Depth: The depth of the tree is the maximum node level.

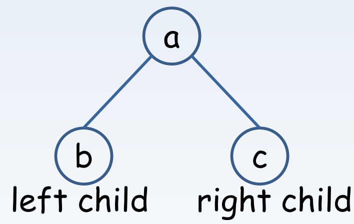
Depth of example tree: 4

Why a Tree?

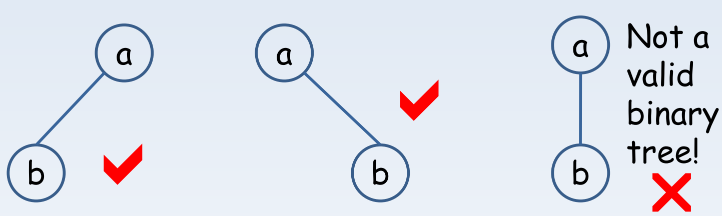
- The list structure is not suited to storing and accessing large amounts of data.
  - Especially if fast access is desired.
- Many operations can be realized with O(log n) complexity on the tree structure.
  - For now, it suffices for us to know that tree operations, on average, can be performed much faster than list operations.
- Due to this speedup, tree structures are used in databases and many areas where indexing is necessary.
- In this course, we will study in detail the binary search tree, which is the simplest tree structure.
- You will learn about different tree structures in Advanced Data Structures.

Binary Tree

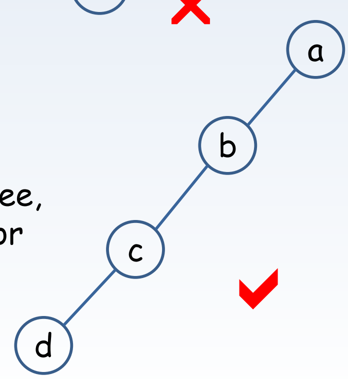
- Every node has at most two subtrees.
  - The degree of a node cannot be larger than 2.
- Subtrees are called left and right subtrees.



Binary Tree Examples



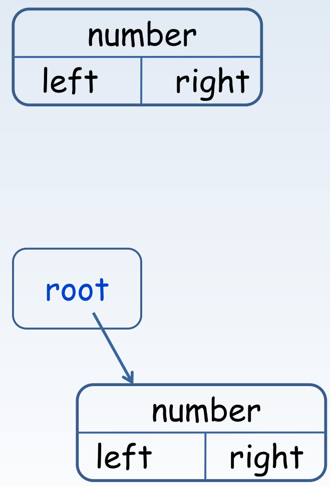
- Left and right subtrees are interpreted differently.
- Even if a node has a single subtree, this subtree's being on the left or the right changes the tree.



Data Structure

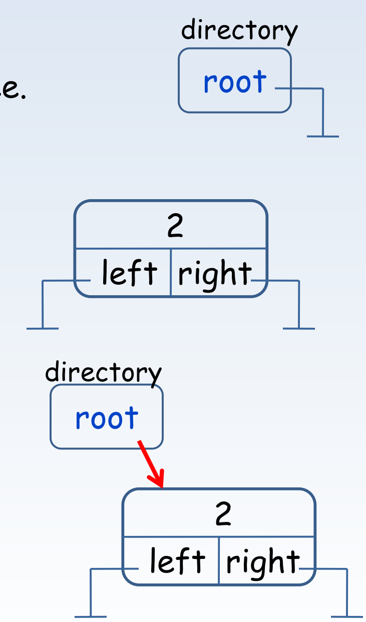
```
struct node {
  int number;
  node *left;
  node *right;
};

struct tree {
  node *root;
};
```



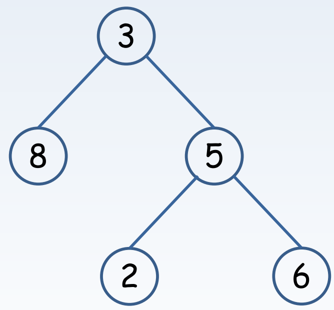
Using the Tree Type

- We must first create an empty tree.
  ```
  tree directory;
  directory.root = NULL;
  ```
- Let us create a new node. Let the number in the node be “2”.
  ```
  node *p;
  p = new node;
  p->number = 2;
  p->left = p->right = NULL;
  ```
- Let us add this node as the root to our “directory”.
  ```
  directory.root = p;
  ```



Tree Representation

- From this point forward, we will not show left and right pointers in the diagrams. These always exist at each node.
- A simpler diagram:



Tree Traversal

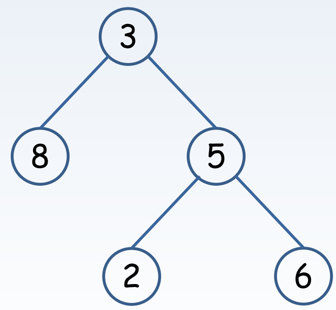
- Traversing a binary tree by visiting all nodes can be done in three ways:
  - Preorder
  - Inorder
  - Postorder
- All three types of traversals are, by definition, recursive operations.

Preorder

- Traversal order
  - First, node itself
  - Then, left subtree
  - Finally, right subtree
- This operation repeats for each node.

Example:

3 8 5 2 6

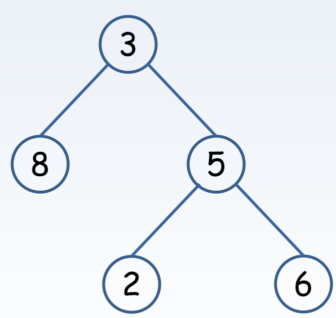


Inorder

- Traversal order
  - First, left subtree
  - Then, node itself
  - Finally, right subtree
- This operation repeats for each node.

Example:

8 3 2 5 6

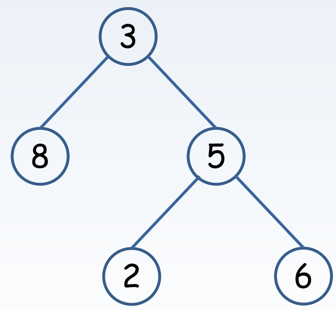


Postorder

- Traversal order
  - First, left subtree
  - Then, right subtree
  - Finally, node itself
- This operation repeats for each node.

Example:

8 2 6 5 3



Coding the Traversal Methods

- All three traversal methods can easily be programmed recursively.
- Let us assume that the previously defined structure called "directory" has been created.
- Let us write the traversal functions that will work on this structure and print the data in the nodes to the screen.

Preorder Function

```
void Preorder(node *nptr) {
   if (nptr) {
     cout << nptr->number << endl;
     Preorder(nptr->left);
     Preorder(nptr->right);
   }
}
```


Inorder Function

```
void Inorder(node *nptr) {
  if (nptr) {
    Inorder(nptr->left);
    cout << nptr->number << endl;
    Inorder(nptr->right);
  }
}
```

Postorder Function

```
void Postorder(node *nptr) {
  if (nptr) {
    Postorder(nptr->left);
    Postorder(nptr->right);
    cout << nptr->number << endl;
  }
}
```


Iterative Inorder Function

```
void Inorder(node *root) {
  node *current;
  char flag = 1;
  stack s;
  s.create();
  current = root;
  while (flag) {
    while (current != NULL) {
      s.push(current);
      current = current->left;
    }
    if ( !s.isempty() ) {
       current = s.pop();
       cout << current->number
            << endl;
       current = current->right;
    }
    else
       flag = 0;
  }
}
```

Iterative Inorder Example

- Let us solve using the stack:
  1 3 4 5 8 10 12