Trees

Jonathan Mwaura

jtmwaura@gmail.com

April 6, 2018

Overview

Revisiting data structures

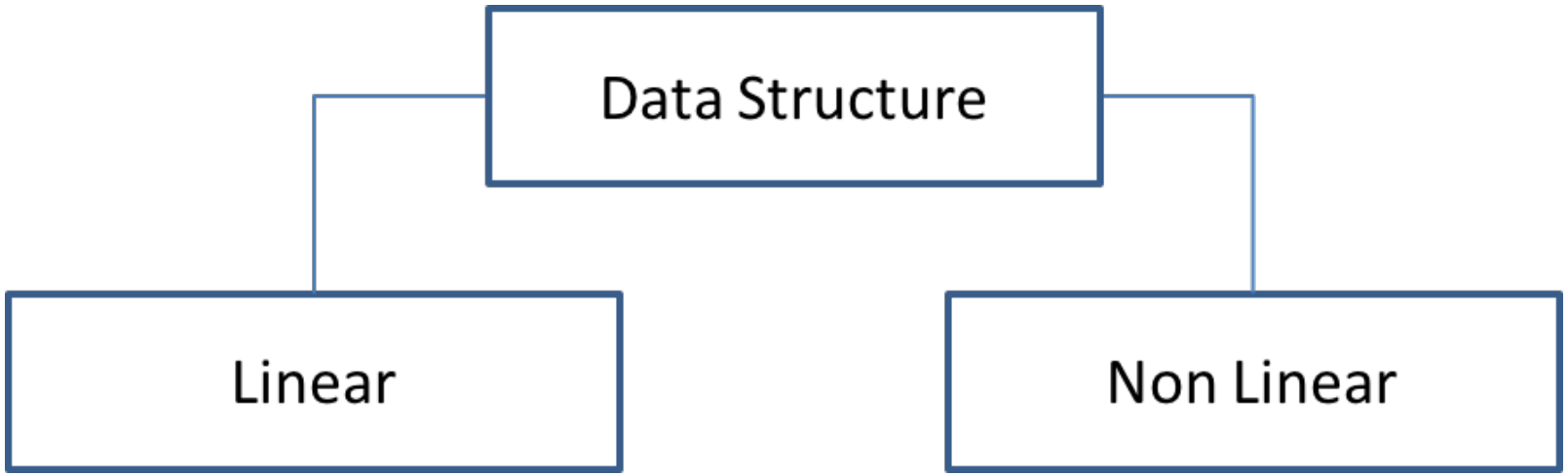


Figure: Introducing data structures

Linear

Data is arranged in a sequential manner:- There is a logical start and a logical end. These include Arrays, Linked Lists, Queues, Stacks etc.

Non Linear/Non sequential

e.g. Tree , Graph, Hash-maps etc.

Revisiting data structures

- What needs to be stored?
  - Different data types are best stored in certain data structures.
- Cost of operations: Cost associated with such operations as search etc
- Memory usage
- Ease of of implementation?

Trees: the why?

Storage and organization of hierarchical data! e.g. a company hierarchy

Large application areas:

- Solving Algebraic Equation
- Information Retrieval e.g. Binary tree search
- Data Mining
- Artificial Intelligence and Robotics
- Machine learning strategies
- Natural Language Processing
- Network routing algorithm

Tree: definition

A collection of entities called nodes linked together with edges to simulate a hierarchy.

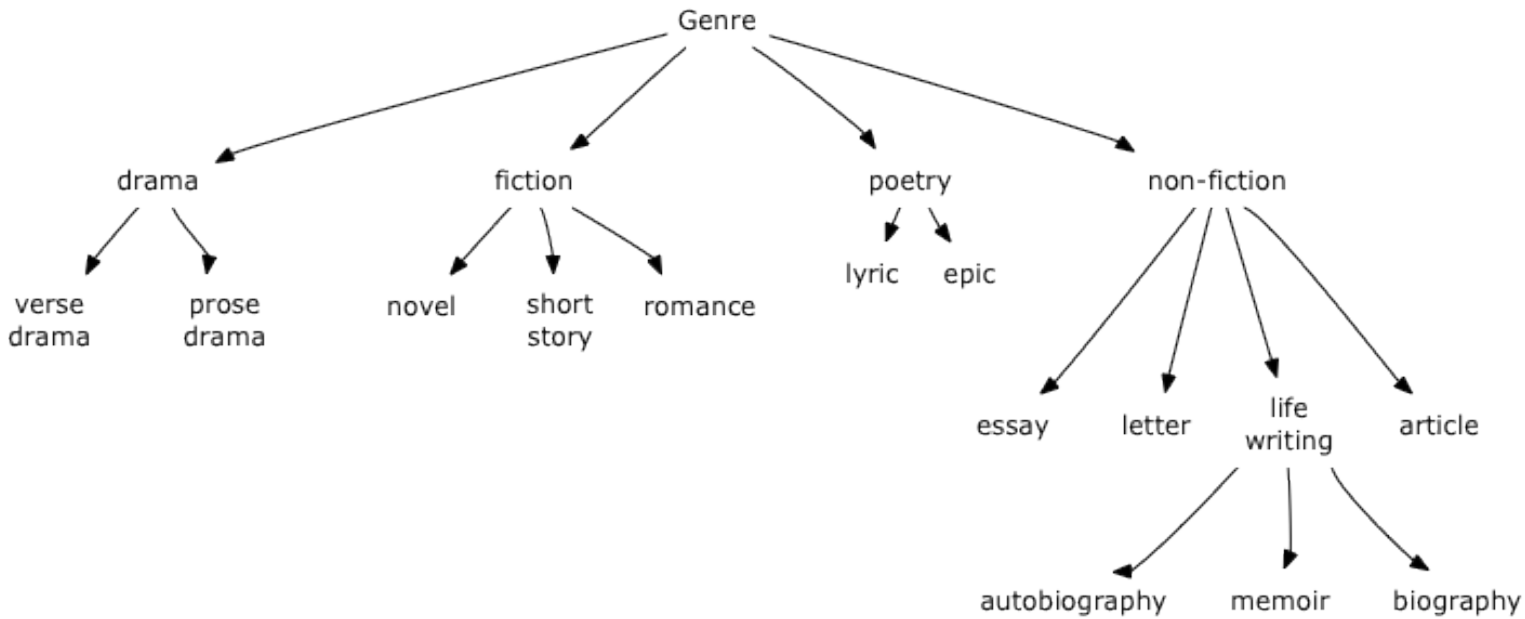


Figure: Tree data structures

Each node of a tree contain data of any type and may contain link/reference to its children. Each edge is a link.

Tree: terminology

- Node: An element of a tree containing data
- Edges: The arrows connecting the nodes are called edges.
- Root : Node at the "top" of a tree - the one from which all operations on the tree commence.
- Leaf/Terminal node : A node with zero children (child).
- Parent node: An immediate predecessor of a node
- Children (child) nodes: All the immediate successors of a node are its child nodes.
- Degree:Number of children a node has.
- Siblings: children of the same parent node (two or more node having the same parent are called siblings)

Tree: terminology

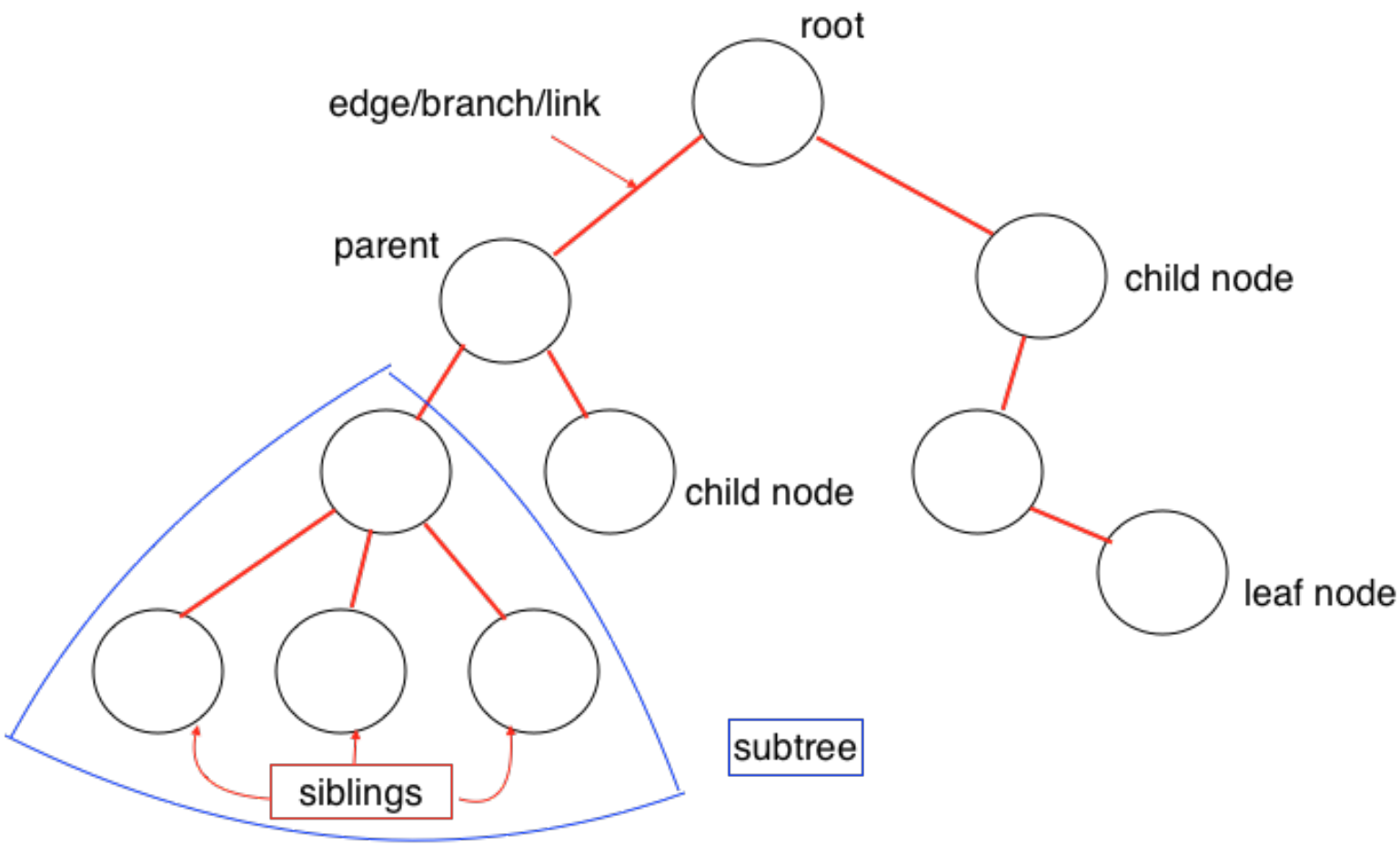


Figure: Understanding the terminology of a tree

Tree: properties

- Recursive:A tree is a recursive structure. A root node connects various sub-trees.
- Nodes: For n nodes, there is exactly n-1 links
- Depth of x: This means the number of edges in a path from root to a node x
- Height of x: This is the number of edges in longest path from x to a leaf
- Height of tree: This is the height of the root node; i.e. longest path from root to a leaf node.
- **Note**: Height and depth are different properties! They may or may not be the same.

Tree: classification

Based on the degree property, different type of trees can be generated.

General Trees

A general tree T is a finite set of one or more nodes such that there is one designated node r, called the root of T, and the remaining nodes are partitioned into n $\geq$ 0 disjoint subsets T_1, T_2, ... , T_n, each of which is a tree, and whose roots r_1,r_2, ... , r_n, respectively, are children of r.

N'ary Trees

An N-Ary tree is a tree in which nodes can have at most N children. A popular N'ary tree is the Binary Tree.

Binary Trees

The simplest form of tree is a binary tree. A binary tree consists of:

- a node (called the root node) and
- left and right sub-trees: Both the sub-trees are themselves binary trees.

A binary tree can have at-most 2 Nodes.

NB: A linked list (a list) is a binary tree where each node is either empty or has at least one child.

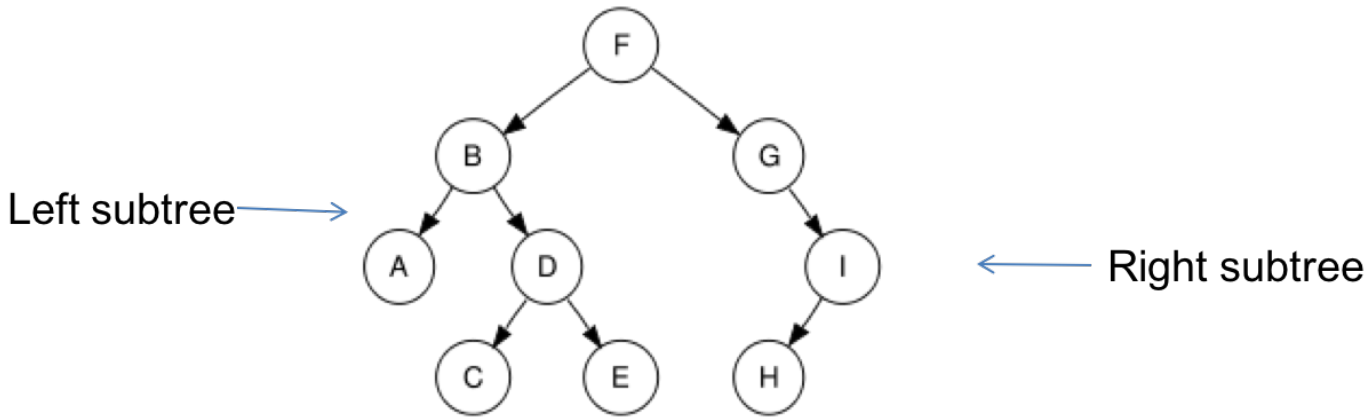


Figure: A binary tree

Linked list representation

```c
#include <stdio.h>
#include <stdlib.h>

struct node;
typedef struct node Node;

struct node {
    int key;
    Node *left;  /* left child */
    Node *right; /* right child */
};
```

Sequential representation using array

- Tree is stored in a single array
- Array index is assigned to each node from leftmost to rightmost node.
- Root node always assign number 1 or could be zero (using array indexing)
- Left child is placed at position [2 * k] (k is position of a parent node e.g. root node)
- Right child is placed at position [2 *k + 1]
- Size of array is dependent on the depth of tree i.e. 2^{d+1}

Representation using arrays

```c
#include <stdio.h>
#include <stdlib.h>

#define NUM_CHILDREN     (2)

struct node;
typedef struct node Node;



struct node {
    int key;
    Node *child[NUM_CHILDREN];
};
```

Representation using array

Tree is stored in a single array

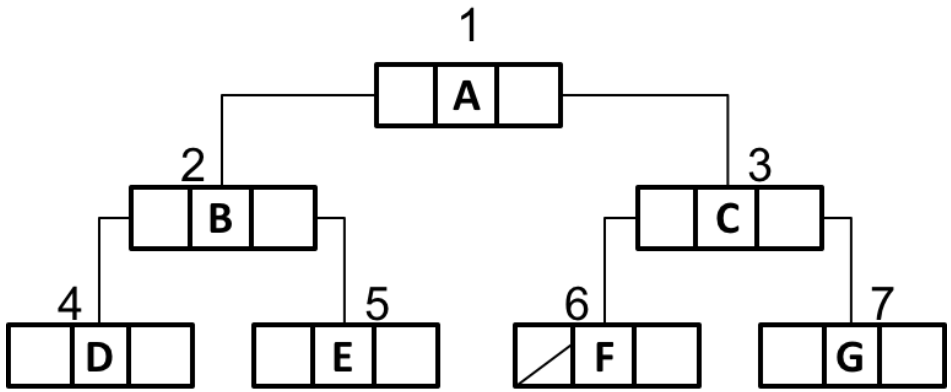


Figure: Linked representation using array

Special forms of Binary Trees

Why categorize trees

We use trees to store and organize data. As such, techniques are needed to minimise cost of inserting, retrieving (searching) and deleting nodes (data stored).

To perform these operations we are interested in:

- the best possible time.
- the average time
- the worst-case time

Searching

Efficient storage of data to facilitate fast searching is an important issue. Trees are important for this purpose.

Linear Search

We search in O(n)

Binary Search

We search in O(logn)

There are other different search algorithms but I wont be looking at them (please find out in various books mentioned in the beginning of the course).

Ordered binary tree

A binary tree is ordered if all its Nodes have the ordering property:

A Node has the ordering property iff

- All the objects contained in the Node's left subtree have keys less than the key at the Node itself;
- All the objects contained in the Node's right subtree have keys greater-than the key at the Node itself.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.

An ordered binary tree is sometimes called a Binary search tree.

Ordered binary tree

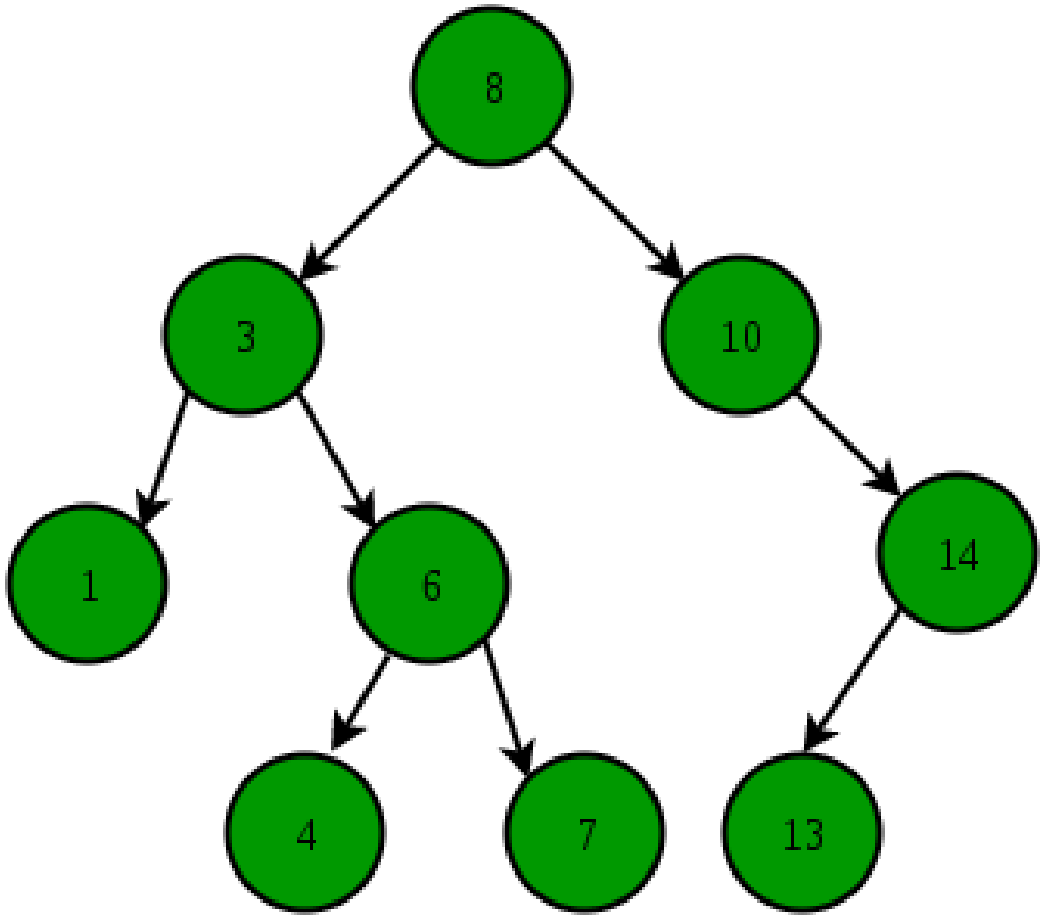


Figure: BST

Operation on BST

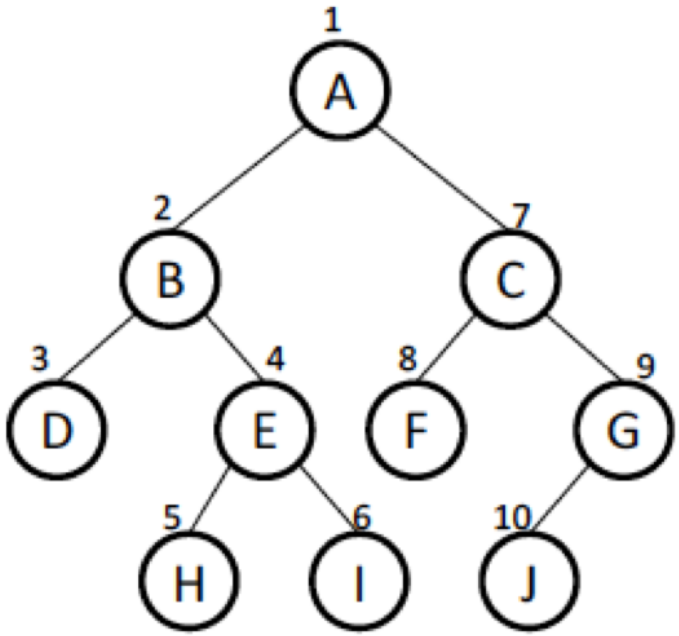
A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal.

Tree traversal techniques

- depth-first traversal
  - PreOrder traversal - visit the parent node first and then left and right children;
  - InOrder traversal - visit the left child, then the parent node and the right child;
  - PostOrder traversal - visit left child, then the right child and then the parent node;
- breadth-first traversal: one type only – level order traversal ; top to bottom and from left to right.

PreOrder Traversal (N L R)

Top-Down -Left -Right approach



Solution:- A , B , D , E , H , I , C , F , G , J

Figure: An pre-order Tree traversal

Preorder Traversal

```
1
2 void doSomethingToAllNodes(Node* root)
3 {
4     if(root) {
5         doSomethingTo(root);
6         doSomethingToAllNodes(root->left);
7         doSomethingToAllNodes(root->right);
8     }
9 }
```

Preorder Traversal - Calculate tree size

```
1 int  numNodes(Node* root){
2     if(root == 0) {
3         return 0;
4      }
5     else {
6       return 1 + numNodes(root->left) + numNodes(root->right);
7      }
8 }
```

Preorder Traversal - Calculate tree height

```
1
2  int treeHeight(Node* root){
3      int lh;      /* height of left subtree */
4      int rh;      /* height of right subtree */
5
6      if(root == 0) {
7          return -1;
8      } else {
9          lh = treeHeight(root->left);
10         rh = treeHeight(root->right);
11         return 1 + (lh > rh ? lh : rh);
12     }
13 }
```

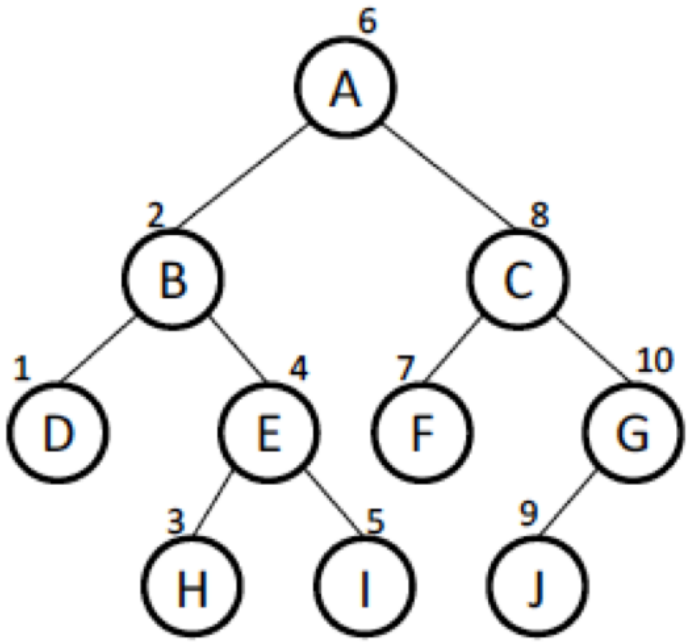
Preorder Traversal - Searching a tree

```
1
2 * returns node with given target key */
3 /* or null if no such node exists */
4
5 Node*  treeSearch(Node* root, int target)
6 {
7     if(root->key == target) {
8         return root;
9     }
10    else if(root->key > target) {
11        return treeSearch(root->left, target);
12    }
13    else {
14        return treeSearch(root->right, target);
15    }
16 }
```

Preorder Traversal - iterative approach

```
1
2 * returns node with given target key */
3 /* or null if no such node exists */
4
5 Node* treeSearch(Node* root, int target)
6 {
7     while(root != 0 && root->key != target) {
8         if(root->key > target) {
9             root = root->left;
10        } else {
11            root = root->right;
12        }
13    }
14
15    return root;
16 }
```

InOrder Traversal (L N R)

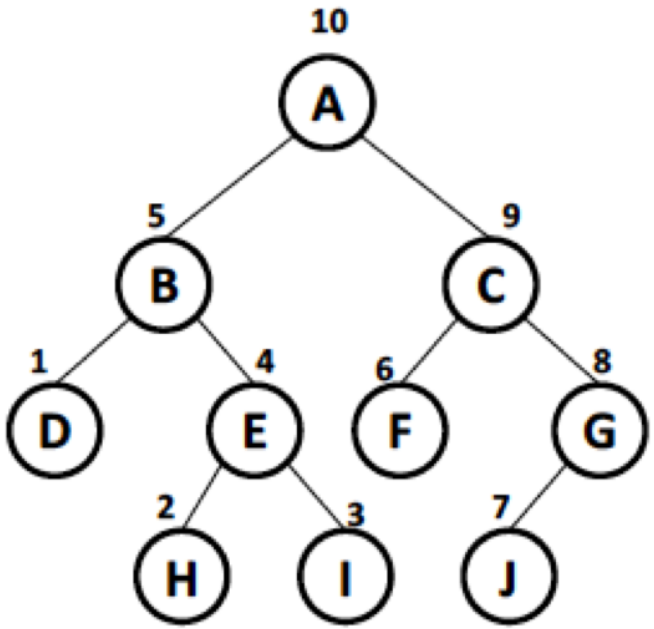


Solution:- D , B , H , E , I , A , F , C , J , G

Figure: An In-order Tree traversal

PostOrder Traversal (L R N)

Left-Right- bottom up traversal.



Solution:- D , H , I , E , B , F , J , G , C , A

Figure: An post-order Tree traversal

If you are given two traversal sequences, can you construct the binary tree?

The following combination can uniquely identify a tree

- Inorder and Preorder.
- Inorder and Postorder.
- Inorder and Level-order(Breadth first)

And following do not

- Postorder and Preorder.
- Preorder and Level-order.
- Postorder and Level-order.

Tree Determination Based on Height

Strictly binary tree

Each node is a leaf or has exactly two children.

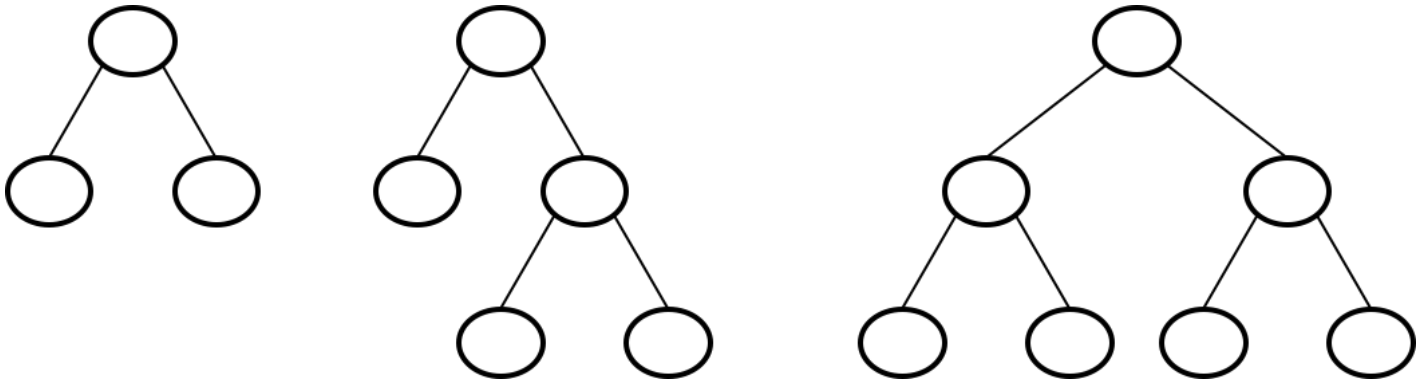


Figure: Strictly binary tree

A strictly or **proper** tree with n leaves always have 2n-1 nodes.
Some literature refer to this tree as a "Full" binary tree.

Complete binary tree

A tree where all terminal/leaf nodes are at the same depth (Perfect binary tree)

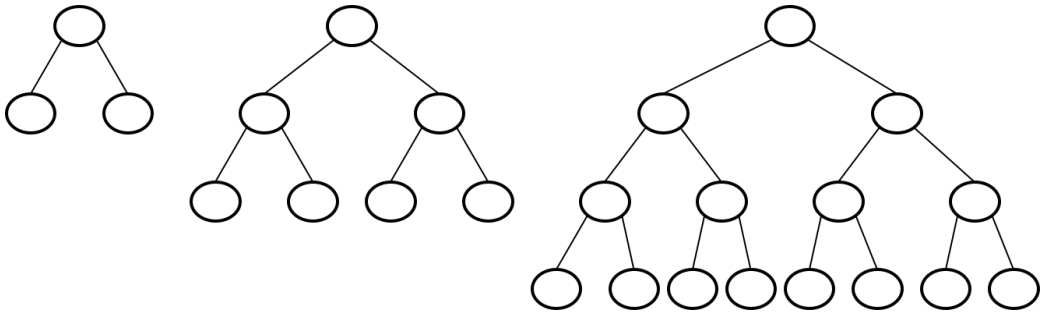


Figure: Complete binary tree

In a special case, there is an **almost complete** binary tree where all levels apart from the last level are complete. In this case the last level nodes must be as left as possible.

At each level the maximum number of nodes is 2^i nodes where i is the level. Maximum number of nodes in a perfect binary tree with height h, will be $2^{h+1} - 1$

Balanced binary tree

A binary tree where the difference between height of left and right sub-trees for every node is not more than k (mostly k=1)

Recall

Height is the number of **EDGES** in longest path from root to leaf.

Height of tree with one node is zero, and therefore height of an empty tree (null) is -1.

A tree is kept balanced to make sure it is dense and it's height minimised. This ensures that cost of operations that depend upon height are minimised.

Tree Rotations

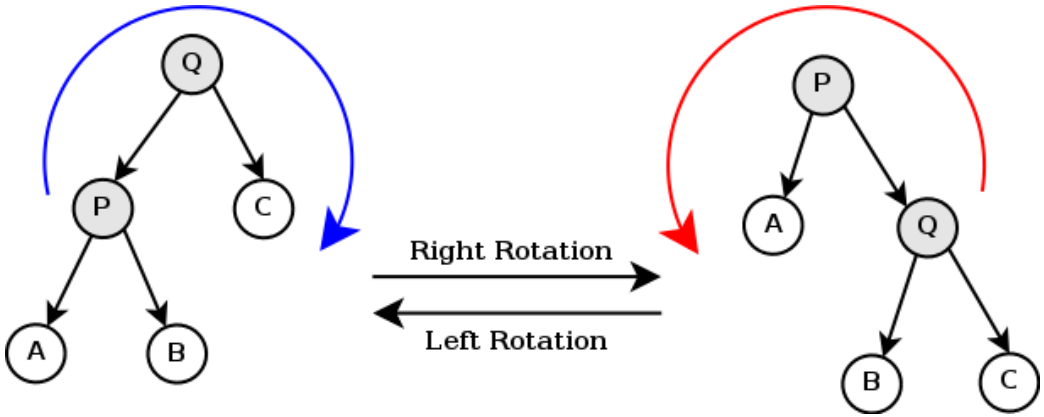


Figure: Tree rotations

Credit: wikipedia.

Balanced binary tree (AVL Trees)

An AVL tree is a balanced binary search tree. Named after their inventors, Adelson-Velskii and Landis, they were the first dynamically balanced trees to be proposed.

- The sub-trees of every node differ in height by at most one.
- Every sub-tree is an AVL tree.

Balanced binary tree (AVL Trees)

Steps to follow for insertion Let the newly inserted node be w

- Perform standard BST insert for w.
- Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

Balanced binary tree (AVL Trees)

Following are the possible 4 arrangements:

- is left child of z and x is left child of y (Left Left Case)
- y is left child of z and x is right child of y (Left Right Case)
- y is right child of z and x is right child of y (Right Right Case)
- y is right child of z and x is left child of y (Right Left Case)

Extended binary tree

If in a binary tree ,each empty leaf node is replaced by a special node then the resulting tree is an extended binary tree or 2-tree A special node is added to a leaf node and nodes that have only one child. The special nodes added to the tree are called external node and original node of tree are called Internal Node

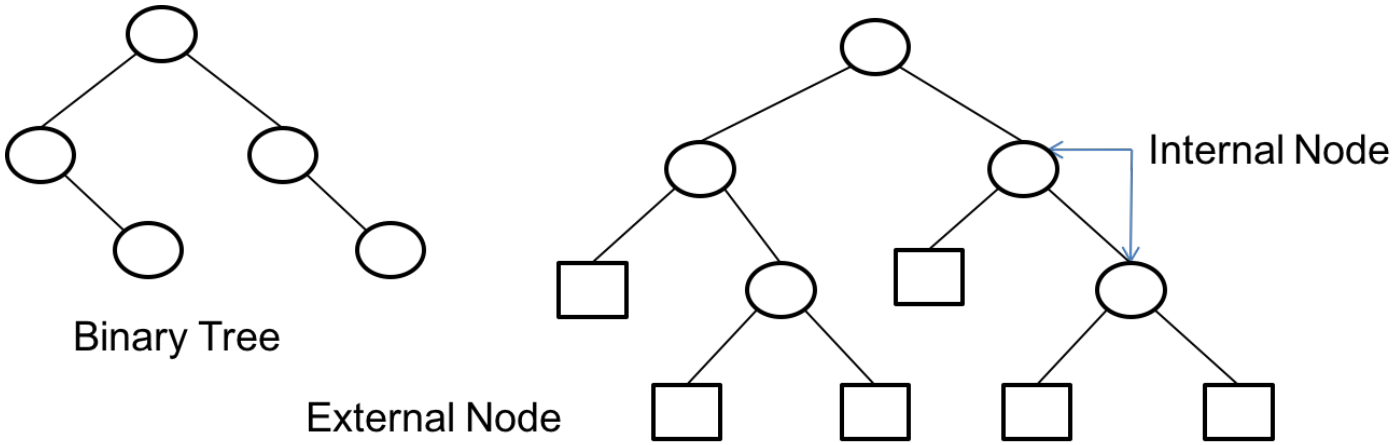


Figure: Extended binary tree

Summary

Tree structures

Powerful data structures that makes use of linear structures and primitive data types to construct a collection of records.

Mainly used for hierarchical data but with varied application areas.

Binary tree

Brings out easiness in searching and sorting data