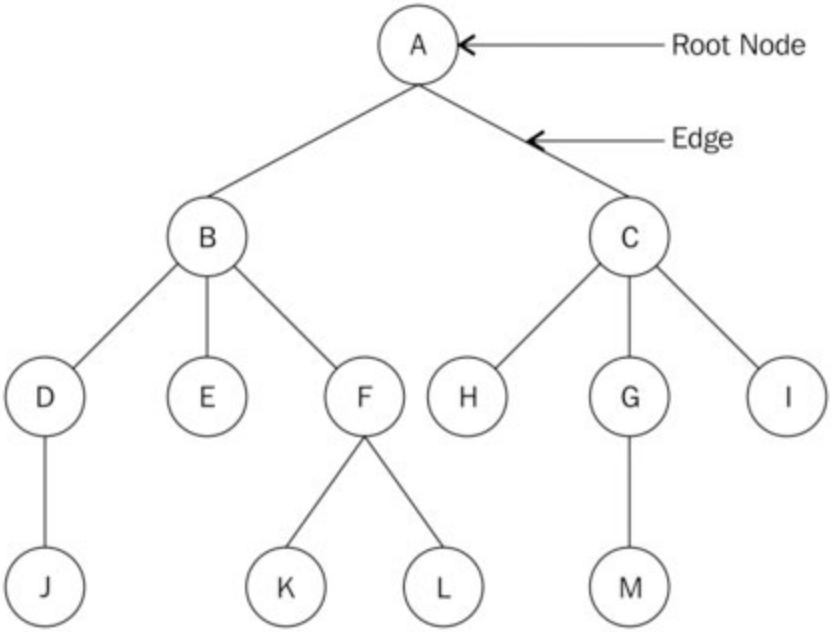
Module 6

Trees

- What is a tree?

  a hierarchical form of data structure



- What is a node?

  any structure that holds data

- What is a root node?

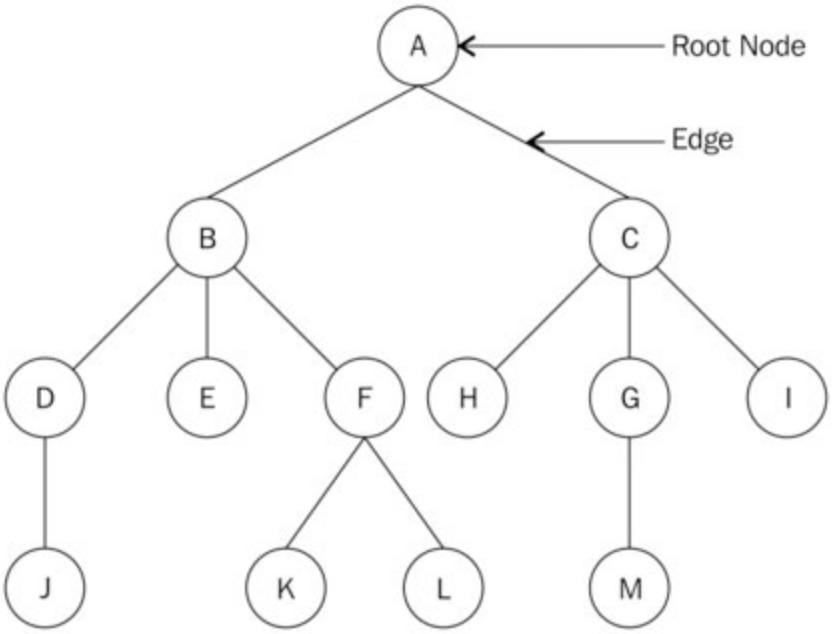
  the only node from which all other nodes come

- What is a sub-tree?

a tree with its nodes being a descendant of some other tree

▼ What is a degree?

- **The number of sub-trees of a given node.**
- A tree consisting of only one node has a degree of 0.
- See tree above: the degree of node A is 2.



▼ What is a leaf node?

- **This is a node with a degree of 0.**
- See tree above: nodes J, E, K, L, H, M, and I are all leaf nodes.

▼ What is an edge?

- **The connection between two nodes.**
- An edge can sometimes connect a node to itself, making the edge appear as a loop.

▼ What is a parent?

- **A node in the tree with other connecting nodes is the parent of those nodes.**
- See tree above: node B is the parent of nodes D, E, and F.

▼ What is a child?

- **This is a node connected to its parent.**
- See tree above: nodes B and C are children of node A, the parent and root node.

▼ What is a sibling?

- **All nodes with the same parent are siblings.**
- See tree above: this makes the nodes B and C siblings.

▼ What is a level?

- **The level of a node is the number of connections from the root node.**
- The root node is at level 0.
- See tree above: nodes B and C are at level 1.

▼ What is the height of a tree?

- **This is the number of levels in a tree.**
- See tree above: our tree has a height of 4.

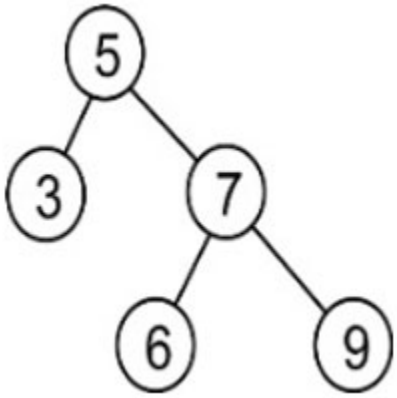
▼ What is depth?

- **The depth of a node is the number of edges from the root of the tree to that node.**
- See tree above: the depth of node H is 2.

▼ What is a binary tree?

- **A tree in which each node has a maximum of two children.**
- Each child is identified as being the right or left child of its parent.

- Since the parent node is also a node by itself, each node will hold a reference to a right and left node even if the nodes do not exist.



▼ Example: building a binary tree node class

- Trees are built up of nodes.
- The nodes that make up a tree need to contain data about the parent-child relationship.

```
def __init__(self, data):
  self.data = data
  self.right_child = None
  self.left_child = None
```

- A node is a container for data and holds references to other nodes.
- Being a binary tree node, these references are to the left and the right children.

```
n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")
```

- Next, we connect the nodes to each other.

- We let n1 be the root node with n2 and n3 as its children.
- Finally, we hook n4 as the left child to n2, so that we get a few iterations when we traverse the left sub-tree.

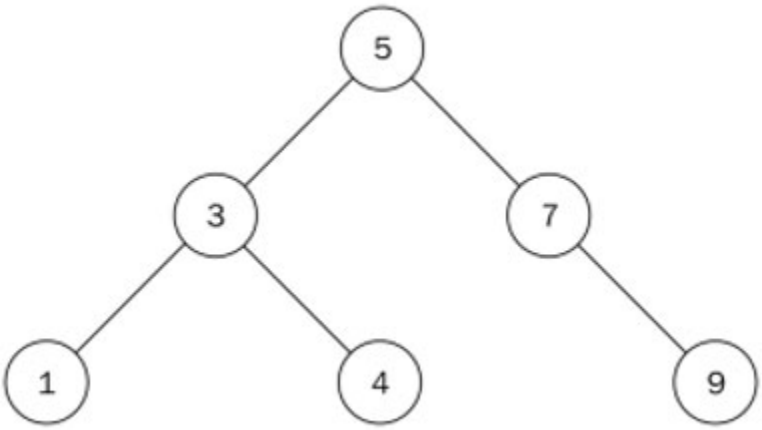
```
n1.left_child = n2
n1.right_child = n3
n2.left_child = n4
```

- We will traverse the left sub-tree.
- We print out the node and move down the tree to the next left node.
- We keep doing this until we have reached the end of the left sub-tree.

```
current = n1
while current:
    print(current.data)
    current = current.left_child
```

▼ What is a binary search tree (BST)?

- **it is structurally a binary tree but functionally, it is a tree that stores its nodes in a way to efficiently search through the tree**
- For a given node with a value, all the nodes in the left sub-tree are less than or equal to the value of that node.
- All the nodes in the right sub-tree of this node are greater than that of the parent node.



▼ BST insert and delete operations must what?

maintain the principle that gives the BST its structure

▼ What is big O for BST insertion?

Insertion of a node in a BST takes **O***(h)*, where *h* is the height of the tree.

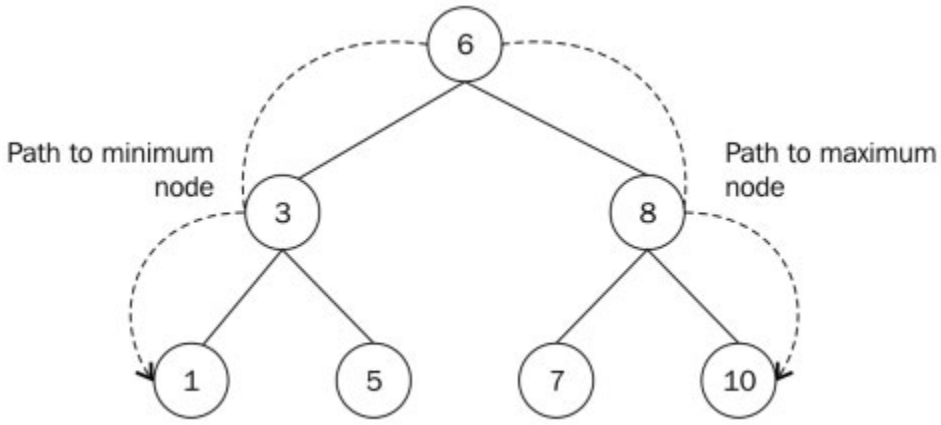
▼ What is big O for BST deletion?

The delete operation takes **O***(h)*, where *h* is the height of the tree.

▼ How to find the minimum node in a BST?

- To find the node with smallest value, start traversal from the root of the tree and visit the left node each time you reach a sub-tree.
- It takes **O***(h)* to find the minimum value in a BST, where *h* is the height of the tree.

```
def find_min(self):
  current = self.root_node
  while current.left_child:
      current = current.left_child
  return current
```



▼ How to find the maximum node in a BST?

- To find the node with largest value, start traversal from the root of the tree and visit the right node each time you reach a sub-tree.
- It takes **O***(h)* to find the maximum value in a BST, where *h* is the height of the tree.

```
def find_max(self):
  current = self.root_node
  while current.right_child:
      current = current.right_child
  return current
```

▼ What is an inorder traversal?

- The algorithm will use recursion to visit each subtree of the root node in the following order: First the left subtree—then the node value—then the right subtree.
- This will cause a sequence of ascending ordered values as the output.
- Example pseudocode:

```
//In-order traversal with recursion
    public class func traverseInOrder(node:BinaryTreeNode?) {
          // The recursive calls end when we reach a Nil leaf
        guard let node = node else {
            return
```

```
        }
        // Recursively call the method again with the leftChild,
        // then print the value, then with the rigthChild
        BinaryTreeNode.traverseInOrder(node: node.leftChild)
        print(node.value)
        BinaryTreeNode.traverseInOrder(node: node.rightChild)
    }
```

▼ Example: `Computing the successor of a position in a binary search tree.`

```
def after(p):
  if right(p) is not None then {successor is leftmost position in p's right subtree}
    walk = right(p)
    while left(walk) is not None do
    walk = left(walk)
    return walk
  else {successor is nearest ancestor having p in its left subtree}
    walk = p
    ancestor = parent(walk)
    while ancestor is not None and walk == right(ancestor) do
    walk = ancestor
    ancestor = parent(walk)
  return ancestor
```

▼ Example: `Recursive search in a binary search tree.`

```
def TreeSearch(T, p, k):
  if k == p.key() then
    return p          {successful search}
  else if k  p.key() and T.right(p) is not None then
    return TreeSearch(T, T.right(p), k)   {recur on right subtree}
  return p          {unsuccessful search}
```

▼ What are AVL trees?

- **Any binary search tree *T* that satisfies the height-balance property is said to be an AVL tree**
- The tree is named after the initials of its inventors: Adel'son-Vel'skii and Landis
- Given a binary search tree *T*, we say that **a position is balanced if the absolute value of the difference between the heights of its children is at most 1**, and we say that it is unbalanced otherwise.

- Thus, **the height-balance property characterizing AVL trees is equivalent to saying that every position is balanced.**
- The insertion and deletion operations for AVL trees begin similarly to the corresponding operations for (standard) binary search trees, but with **post-processing for each operation to restore the balance of any portions of the tree that are adversely affected by the change.**
- AVL trees are strictly balanced trees and inserting a new node can break that balance. Once we put the new node in the correct subtree, we need to ensure that the balance factors of its ancestors are correct. This process is called *retracing*. If there is any balance factor with wrong values (not in the range [-1 , 1]), we will use rotations to fix it.
- Deletion from a regular binary search tree results in the structural removal of a node having either zero or one children. Such a change may violate the height-balance property in an AVL tree. Particularly, if position *p* represents the parent of the removed node in tree *T*, there may be an unbalanced node on the path from *p* to the root of *T*.