Trees and their search

Mustafa Hajij
MSDS program
University of San Francisco

Trees

- Definition:

  - A tree is a hierarchical data structure composed of nodes connected by edges, widely used in computer science for organizing and representing hierarchical relationships.

- Characteristics:

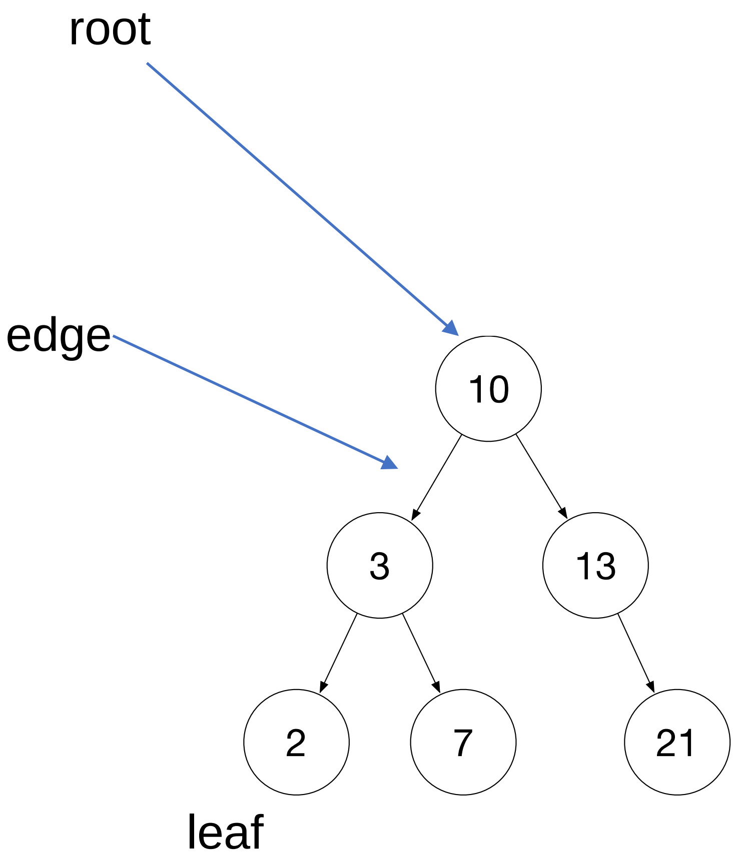
  - Nodes: Elements within the structure.

  - Edges: Connections between nodes, defining relationships.

  - Root: Topmost node serving as the starting point.

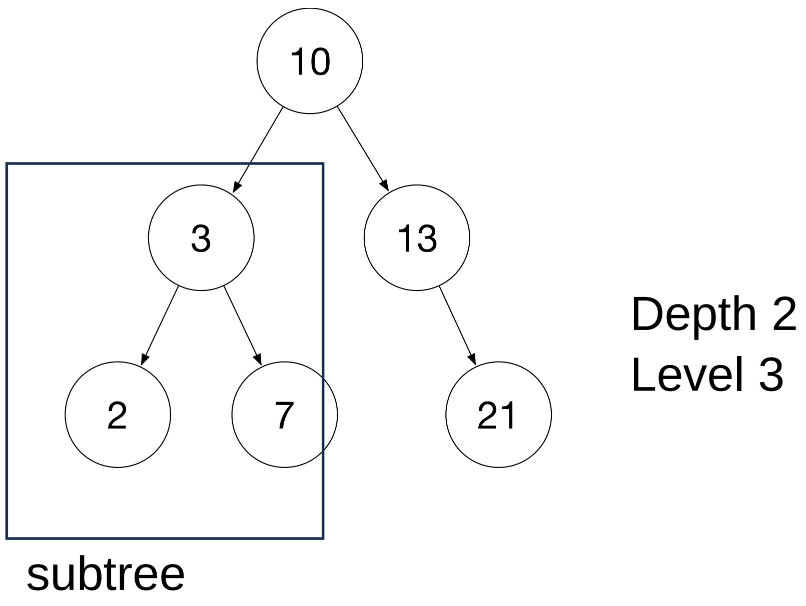
  - Parent and Child Nodes: Nodes organized hierarchically with parent-child relationships.

  - Leaf Nodes: Endpoints without child nodes.



Key Concepts in Tree Structures

- **Subtree:**
  - A subtree is a portion of a tree consisting of a node and all its descendants.
- **Depth and Level:**
  - Depth: The length of the path from the root to a node.
  - Level: One more than the depth of a node.
- **Binary Trees:**
  - A binary tree is a specialized tree where each node has at most two children, known as the left child and the right child.



Depth first search

- **Definition:**
  - Depth-First Search (DFS) is a traversal algorithm for trees, systematically exploring each branch before backtracking.
- **Visitation Order:**
  - The order of visiting nodes during DFS remains consistent: discover and finish nodes in the same order.
- **Traversal Order:**
  - The specific traversal order (pre-order, in-order, post-order) depends on the location of actions during traversal.

Depth first search

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def dfs_preorder(node):
    if node is not None:
        print(node.value)
        dfs_preorder(node.left)
        dfs_preorder(node.right)

# Example Usage:
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

dfs_preorder(root) # 1,2,4,5,3
```

Depth first search

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def dfs_preorder(node):
    if node is not None:
        print(node.value)
        dfs_preorder(node.left)
        dfs_preorder(node.right)

# Example Usage:
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

dfs_preorder(root) # 1,2,4,5,3
```

```
def dfs_postorder(node):
    if node is not None:
        dfs_postorder(node.left)
        dfs_postorder(node.right)
        print(node.value)

# Example Usage (continued):
dfs_postorder(root)
```

```
Output : 4 5 2 3 1
```

Question: What is T(n) for the above?

Compare DFS on trees against binary search

```
def DFS_tree(p:TreeNode):
    if p is None: return
    print(p.value)
    DFS_tree(p.left)
    DFS_tree(p.right)
```

```
def binary_search(p:TreeNode, x:object):
    if p is None: return None
    if x<p.value:
        return binary_search(p.left, x)
    if x>p.value:
        return binary_search(p.right, x)
    return p
```

What is the complexity of the above two algorithms ?

Compare DFS on trees against binary search

```
def DFS_tree(p:TreeNode):
    if p is None: return
    print(p.value)
    DFS_tree(p.left)
    DFS_tree(p.right)
```

```
def binary_search(p:TreeNode, x:object):
    if p is None: return None
    if x<p.value:
        return binary_search(p.left, x)
    if x>p.value:
        return binary_search(p.right, x)
    return p
```

$$T(n) = k + \boxed{2}T(n/2)$$

$$T(n) = k + T(n/2)$$

Graphs: simplistic class

```
class Graph:
    def __init__(self):
        self.vertices = {}
        self.edges = []


    def add_vertex(self, vertex):
        if vertex not in self.vertices:
            self.vertices[vertex] = []

    def add_edge(self, from_vertex, to_vertex):
        if from_vertex in self.vertices and to_vertex in self.vertices:
            self.vertices[from_vertex].append(to_vertex)
            self.edges.append((from_vertex, to_vertex))
```

```
# Example Usage:
graph = Graph()

# Adding vertices
graph.add_vertex("A")
graph.add_vertex("B")
graph.add_vertex("C")

# Adding edges
graph.add_edge("A", "B")
graph.add_edge("B", "C")
graph.add_edge("C", "A")
```

Graphs: simplistic class

```
class Graph:
    def __init__(self):
        self.vertices = {}
        self.edges = []


    def add_vertex(self, vertex):
        if vertex not in self.vertices:
            self.vertices[vertex] = []

    def add_edge(self, from_vertex, to_vertex):
        if from_vertex in self.vertices and to_vertex in self.vertices:
            self.vertices[from_vertex].append(to_vertex)
            self.edges.append((from_vertex, to_vertex))
```

```
# Example Usage:
graph = Graph()

# Adding vertices
graph.add_vertex("A")
graph.add_vertex("B")
graph.add_vertex("C")

# Adding edges
graph.add_edge("A", "B")
graph.add_edge("B", "C")
graph.add_edge("C", "A")
```

Question : how do we traverse the nodes of a graph?

DFS graphs VS trees

```
def DFS_graph(p:Node):
    if p is None: return
    print(p.value)
    for q in p.neighbors:
        DFS_graph(q)
```

```
def DFS_tree(p:TreeNode):
    if p is None: return
        print(p.value)
    DFS_tree(p.left)
    DFS_tree(p.right)
```

What is wrong with this implementation ?

DFS graphs VS trees

```
def DFS_graph(p:Node):
    if p is None: return
    print(p.value)
    for q in p.neighbors:
        DFS_graph(q)
```

```
def DFS_tree(p:TreeNode):
    if p is None: return
        print(p.value)
    DFS_tree(p.left)
    DFS_tree(p.right)
```

What is wrong with this implementation ?

If the graph has cycles this program will never terminate

DFS on graph, corrected : avoiding cycles

```
def DFS_graph (p: Node, visited: set) -> None:
    if p is None or p in visited:
        return
    visited.add(p)  # Ensure the node is marked as visited before recursion.
    print(p.value)

    for neighbor in p.neighbors :
        DFS_graph(neighbor, visited)
```