1. What is binary tree. write any two applications of tree

A binary tree is a data structure composed of nodes, where each node has at most two children, referred to as the left child and the right child. The topmost node is called the root.

Any two applications are:

1) Binary Search Tree
2) Expression Tree

OR

Define expression tree with an example

An expression tree is a binary tree representation of mathematical expressions, where each leaf node represents an operand and each internal node represents an operator is called expression tree

Example:

(5+3)×(12−4)(5+3)×(12−4).

```
          [*]
        /     \
     [+]       [-]
    /   \     /   \
  [5]   [3] [12]  [4]
```

2. Define priority queue data structure OR binary search tree

Binary search tree : A binary search tree is a binary tree data structure in which each node has at most two children, referred to as the left child and the right child. The left sub tree less than root node and right sub tree in greater than equal to root node

Priority queue : priority queue is the abstract data structure where each element in the queue has a priority queue

3. Define DFS and BFS traversal technique

Depth-First Search (DFS):

Depth-First Search is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It operates by visiting a node's children recursively until it reaches a leaf node, then backtracks and explores the other branches.

Breadth-First Search (BFS):

Breadth-First Search is another graph traversal algorithm that explores all neighbor nodes at the present depth befor e moving on to nodes at the next depth level. It operates by visiting nodes level by level, exploring all nodes at a given level before moving to the next level.

OR

Define hashing and hash function

- Hashing

♣ Hashing is a technique of mapping keys, and values into the hash table by using a hash function.
♣ Hashing is the process of converting one value into another based on a specified key or string of character

- Hash function

♣ Hash function converts or maps the keys to specific entries in the table.
♣ Hash function generates hash value. Hash value specifies the slot or location of key in hash table
♣ For example, following set of keys: 765, 431, 96, 142, 579 and a hash table, T, containing M = 13 elements. We can define a simple hash function as follows: h(key) = key % M

4. Explain with an illustration how to design queue ADT along with any 4 operations

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle.

Queue ADT:

1. **enqueue(item):** This operation adds an element to the back of the queue.
2. **dequeue():** This operation removes and returns the element from the front of the queue.
3. **peek():** This operation returns the element at the front of the queue without removing it.
4. **isEmpty():** This operation checks if the queue is empty or not.

Program…

```
class Queue:
  def __init__(self):
    self.queue = []

  def enqueue(self, item):
    self.queue.append(item)
```

```
def dequeue(self):
    if not self.isEmpty():
        return self.queue.pop(0)
    else:
        print("Queue is empty. Cannot dequeue.")
        return None

def peek(self):
    if not self.isEmpty():
        return self.queue[0]
    else:
        print("Queue is empty. Cannot peek.")
        return None

def isEmpty(self):
    return len(self.queue) == 0
```

```
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)

print("Queue:", q.queue)

print("Dequeued item:", q.dequeue())
print("Queue after dequeue:", q.queue)

print("Peeked item:", q.peek())

print("Is the queue empty?", q.isEmpty())
```

OR

List the tree traversal techniques and design a code snippet to traverse the tree in pre-order

- **Pre-order Traversal**:
- **In-order Traversal**:
- **Post-order Traversal**:

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

def preorder(root):
    if root:
        print(root.val, end=" ")
        preorder(root.left)
        preorder(root.right)

# Example usage:
# Create a binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Perform pre-order traversal
print("Pre-order Traversal:")
preorder(root)
```

5. write a code segment to traverse the tree using BFS technique

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


def bfs_traversal(root):
  if root is None:
    return []


  result = []
  queue = [root]


  while queue:
    level_size = len(queue)
    current_level = []


    for _ in range(level_size):
      node = queue.pop(0)
      current_level.append(node.val)
```

```
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.extend(current_level)

    return result
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)
print("BFS Traversal:", bfs_traversal(root))
```

OR

With example illustrate the following terms used in binary tree root node, height of a noda, path and width of the tree

```
     1
    / \
   2   3
  / \   \
 4   5   6
```

1. **Root Node**: The root node is the topmost node in the tree. It is the starting point for traversing the tree. In this example, node 1 is the root node.
2. **Height of a Node**: The height of a node is the number of edges on the longest path from that node to a leaf node. It represents the length of the longest downward path to a leaf from that node. The height of a leaf node is 0. In this example:
   - Height of node 1: 2 (path: 1 -> 3 -> 6)
   - Height of node 2: 1 (path: 2 -> 5)
   - Height of node 3: 0 (leaf node)

   - Height of node 4: 0 (leaf node)
   - Height of node 5: 0 (leaf node)
   - Height of node 6: 0 (leaf node)
3. **Path**: A path in a binary tree is a sequence of nodes connected by edges. It represents the route from one node to another. In this example, some paths are:
   - Path from node 1 to node 6: [1, 3, 6]
   - Path from node 2 to node 5: [2, 5]
4. **Width of the Tree**: The width of a tree is the maximum number of nodes at any level in the tree. It indicates the maximum number of nodes that exist in any level of the tree. In this example, the width of the tree is 3 (at level 2, with nodes 4, 5, and 6).

6. A) with example explain any two types of binary tree.

Complete Binary Tree:

- In a complete binary tree, all levels are completely filled except possibly for the last level, which is filled from left to right. This property ensures that the tree is as compact as possible.
- Complete binary trees are used in heap data structures.
- Example:

```
   1

  /  \

 2    3

/\  /

4   5 6
```

Full Binary Tree:

- A Full Binary Tree is a binary tree in which every node other than the leaves has two children. All leaf nodes are at the same level.
- n this example, every internal node (non-leaf node) has exactly two children, fulfilling the condition of a Full Binary Tree.
- Example:

-
  ```
      1
  ```

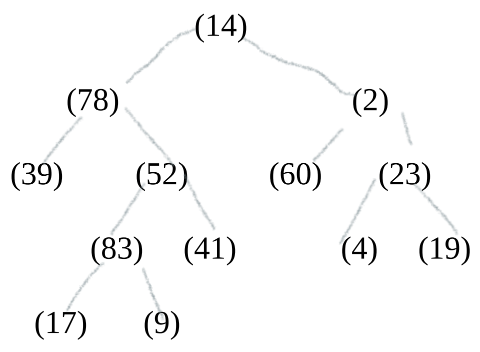
-
  ```
    /    \
  ```

-
  ```
   2      3
  ```

-
  ```
   / \     / \
  ```

-
  ```
  4   5  6   7
  ```

B) explain DFS traversal technique and traversal the following tree in pree-order using DFS. Write its stack contents.



Pre-order DFS Traversal:

1. Visit the root (14).
2. Recursively traverse the left subtree (78).
3. Recursively traverse the left subtree (39).
4. Recursively traverse the left subtree (83).
5. Recursively traverse the left subtree (17).
6. Backtrack to the parent node (83) and traverse its right subtree (9).
7. Backtrack to the parent node (39) and traverse its right subtree (52).
8. Recursively traverse the left subtree (41).
9. Backtrack to the parent node (52) and traverse its right subtree (41).
10. Backtrack to the parent node (78) and traverse its right subtree (2).
11. Recursively traverse the left subtree (60).
12. Recursively traverse the left subtree (4).
13. Backtrack to the parent node (60) and traverse its right subtree (19).
14. Backtrack to the parent node (2) and traverse its right subtree (23).

So, the pre-order traversal of the given tree is: 14, 78, 39, 83, 17, 9, 52, 41, 60, 4, 19, 2, 23.

Program

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

def pre_order_dfs(root):
    if root is None:
        return []

    stack = [root]
    result = []
    stack_contents = []

    while stack:
        node = stack.pop()
        result.append(node.val)

        stack_contents.append(node.val)

        if node.right:
                stack.append(node.right)
            if node.left:
                stack.append(node.left)

        return result, stack_contents

root = TreeNode(14)
root.left = TreeNode(78)
root.right = TreeNode(2)
root.left.left = TreeNode(39)
root.left.right = TreeNode(52)
root.right.left = TreeNode(60)
root.right.right = TreeNode(23)
root.left.right.left = TreeNode(83)
root.left.right.right = TreeNode(41)
root.right.left.right = TreeNode(4)
root.right.right.left = TreeNode(19)
root.left.right.left.left = TreeNode(17)
root.left.right.right.left = TreeNode(9)

result, stack_contents = pre_order_dfs(root)
print("Pre-order DFS Traversal:", result)
print("Stack Contents during Traversal:", stack_contents)
```

Output:

```
Pre-order DFS Traversal: [14, 78, 39, 83, 17, 9, 52, 41, 2, 60, 4, 19, 23]
Stack Contents during Traversal: [14, 78, 39, 83, 17, 9, 52, 41, 2, 60, 4,
19, 23]
```

OR

6.a) what is hash collision ? explain how do you avoid hash collision using open hashing

A hash collision occurs when the pythons hash() function generates the same hash value for two different inputs in index.

One way to avoid hash collisions is by using open hashing, also known as separate chaining. In open hashing, each bucket in the hash table maintains a linked list or another data structure to store multiple key-value pairs that hash to the same index.

1. **Hash Table Initialization**: Initialize an array (hash table) with a fixed number of buckets to store key-value pairs.
2. **Hash Function**: Use a hash function to map keys to indices in the hash table. The hash function should distribute keys uniformly across the available buckets to minimize collisions.
3. **Collision Handling**: When a collision occurs (i.e., two or more keys hash to the same index), instead of overwriting the existing value, insert the new key-value pair into a data structure associated with that bucket, such as a linked list.

4. **Search and Retrieval**: To search for a key in the hash table, apply the hash function to determine the bucket index and then traverse the associated data structure (e.g., linked list) to find the key.

By using open hashing, hash collisions are handled gracefully by storing multiple key-value pairs in the same bucket.

b) assume an initially empty hash table with 11 entries in which the following hash function is used:h(key)=(2*key+3)%11 show the contents of the hash table after the following keys are inserted (in the order listed). Avoid hash collision if occur with linear probing method.[67,815,45,39,20,901,34]

1. Insert 67:
   - h(67)=(2×67+3)%11=137%11=4h(67)=(2×67+3)%11=137%11=4 (no collision, insert at index 4).
2. Insert 815:
   - h(815)=(2×815+3)%11=1633%11=7h(815)=(2×815+3)%11=1633%11=7 (no collision, insert at index 7).
3. Insert 45:
   - h(45)=(2×45+3)%11=93%11=5h(45)=(2×45+3)%11=93%11=5 (no collision, insert at index 5).
4. Insert 39:
   - h(39)=(2×39+3)%11=81%11=4h(39)=(2×39+3)%11=81%11=4 (collision at index 4, probe linearly).
   - Probe 4 + 1 = 5 (collision, continue probing).
   - Probe 5 + 1 = 6 (no collision, insert at index 6).
5. Insert 20:
   - h(20)=(2×20+3)%11=43%11=10h(20)=(2×20+3)%11=43%11=10 (no collision, insert at index 10).
6. Insert 901:
   - h(901)=(2×901+3)%11=1805%11=0h(901)=(2×901+3)%11=1805%11=0 (no collision, insert at index 0).
7. Insert 34:
   - h(34)=(2×34+3)%11=71%11=7h(34)=(2×34+3)%11=71%11=7 (collision at index 7, probe linearly).
   - Probe 7 + 1 = 8 (no collision, insert at index 8).

Now, let's represent the hash table after the insertion of all keys:

```
markdown
Index:   0    1    2    3    4    5    6    7    8    9    10
-----------------------------------------------------------------
Key:    901   -    -    -   67   45   39  815   34   20    -
```

7.construct a binary tree for the data items

[56,67,90,23,45,56,34,12,110,29,5,78]. And traverse the resulting tree in in-order,post-order and pre-order.

Let's go through the process step by step:

1. **Construct Binary Tree**:
   - Insert each data item into the binary tree.
2. **In-order Traversal**:
   - Visit nodes in left-root-right order.(LNR)
3. **Post-order Traversal**:
   - Visit nodes in left-right-root order.(LRN)
4. **Pre-order Traversal**:
   - Visit nodes in root-left-right order.(NLR)

```
        56
       /   \
     23     67
    / \       \
  12   45      90
   \          /
    5        78
            /
          29
            \
             110
```

In-order traversal (LNR):

- `5, 12, 23, 45, 56, 67, 78, 90, 110`

- **Post-order traversal** (LRN):

- `5, 12, 45, 23, 78, 110, 90, 67, 56`

- **Pre-order traversal** (NLR):

`56, 23, 12, 5, 45, 67, 90, 78, 29, 110`

OR