Week-11

11. Implement Binary search tree and its operations using list.

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

    def insert(self, val):
        if not self.val:
            self.val = val
            return
        if self.val == val:
            return
        if val < self.val:
            if self.left:
                self.left.insert(val)
                return
            self.left = BSTNode(val)
            return
        if self.right:
            self.right.insert(val)
            return
        self.right = BSTNode(val)

    def preorder(self, vals):
        if self.val is not None:
            vals.append(self.val)
            if self.left is not None:
                self.left.preorder(vals)
            if self.right is not None:
                self.right.preorder(vals)
        return vals

    def inorder(self, vals):
        if self.left is not None:
            self.left.inorder(vals)
        if self.val is not None:
            vals.append(self.val)
        if self.right is not None:
            self.right.inorder(vals)
        return vals

    def postorder(self, vals):
        if self.left is not None:
            self.left.postorder(vals)
        if self.right is not None:
            self.right.postorder(vals)
        if self.val is not None:
            vals.append(self.val)
        return vals

# Test the BST and its traversal methods
nums = [12, 6, 18, 19, 21, 11, 3, 5, 4, 24, 18]
bst = BSTNode()
for num in nums:
    bst.insert(num)
```

```
print("Preorder:")
print(bst.preorder([]))
print("Inorder:")
print(bst.inorder([]))
print("Postorder:")
print(bst.postorder([]))
```

Output:
Preorder:
[12, 6, 3, 5, 4, 11, 18, 19, 21, 24]
Inorder:
[3, 4, 5, 6, 11, 12, 18, 19, 21, 24]
Postorder:
[4, 5, 3, 11, 6, 24, 21, 19, 18, 12]

Week 12

12 A.implementations of BFS

```
graph = {
  '5': ['3', '7'],
  '3': ['2', '4'],
  '7': ['8'],
  '2': [],
  '4': ['8'],
  '8': []
}

def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)

  while queue:
    m = queue.pop(0)
    print(m, end=" ")  # Print with space for readability
    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

print("Following is the Breadth-First Search:")
visited = []  # Initialize visited as empty list
queue = []    # Initialize queue as empty list
bfs(visited, graph, '5')
```

Output:
Following is the Breadth-First Search:
5 3 7 2 4 8

B.

```
graph = {
   '4': ['8'],
   '8': []
}
visited = set()
def dfs(visited, graph, node):
   if node not in visited:
       print(node)
       visited.add(node)
       for neighbour in graph[node]:
           dfs(visited, graph, neighbour)
print("Following is the Depth-First Search:")
dfs(visited, graph, '4')
```

Output:

```
Following is the Depth-First Search:
4
8
```

Week 13

13.implement Hash functions

A. demonstration working of Hash functions

```
val1 = 121
val2 = 121.09
val3 = "GeeksforGeeks"  # Corrected the typo here
print("The integer value is", hash(val1))
print("The float value is", hash(val2))
print("The string value is", hash(val3))
tuple_val = (1, 2, 3, 4, 5)  # Naming this tuple_val for clarity, though 'tuple' is not incorrect
print("The tuple value is", hash(tuple_val))
```

Output:

```
the integer value is 121
teh float value is 207525870829240441
the string value is -5857547886710058896
the tuple value is -5659871693760987716
```

B.Hash functions for custom objects

```
class Student:
   def __init__(self, name, email):
       self.name = name
       self.email = email
s = Student("Arun", "arun@abc.com")
result = hash(s)
print("Hash value:", result)
```

Output:

```
Hash value: 142940959773
```