Assignment 5

1. The reverse of a directed graph *G = (V , E)* is another directed graph *GR = (V , ER)* on the same vertex set, but with all edges reversed; that is, *ER= {(v, u) : (u, v) ∈ E}*. Given the adjacency list for G, give a linear time algorithm to compute the adjacency list for *GR*.

To compute the adjacency list for the reversed graph GR, given the adjacency list for the original directed graph G, you can use the following linear time algorithm:

1. Create an empty dictionary to store the adjacency list for GR.
2. Iterate over each vertex v in the adjacency list of G.
3. For each vertex v, iterate over its outgoing neighbors in G.
4. For each outgoing neighbor u, add v as a key in the reversed graph dictionary.
5. If the key already exists in the reversed graph, append u to the existing list of values.
6. If the key does not exist in the reversed graph, create a new list with u as the first value.
7. Repeat steps 3-6 for all vertices in the adjacency list of G.
8. Return the reversed graph dictionary as the adjacency list for GR.

Here's the Python implementation of the algorithm:

```
def reverse_adjacency_list(adj_list):
    reversed_adj_list = {}

    for vertex in adj_list:
        for neighbor in adj_list[vertex]:
            if neighbor in reversed_adj_list:
                reversed_adj_list[neighbor].append(vertex)
            else:
                reversed_adj_list[neighbor] = [vertex]

    return reversed_adj_list
```

You can use this algorithm by passing your adjacency list of the original directed graph G to the `reverse_adjacency_list` function. The function will return the adjacency list for the reversed graph GR as a new dictionary.

The time complexity of this algorithm is O(V + E), where V is the number of vertices and E is the number of edges in the original graph G.

2. Given a set of numbers, its median, informally, is the "halfway point" of the set. When the set's size n is odd, the median is unique, occurring index $i = (n + 1)/2$. When n is even, there are two medians, occurring at $i = n/2$ and $i = n/2 + 1$, which are called the "lower median" and "upper median," respectively. Regardless of the parity of n, medians occur at $i = \lfloor (n + 1)/2 \rfloor$ (the lower median) and $i = \lceil (n + 1)/2 \rceil$ (the upper median). For simplicity in this question, we use the phrase "the median" to refer to the lower median. So we care about the $i = \lfloor (n + 1)/2 \rfloor$ position.

Design and describe, in detail, a data structure *Median − Heap* to maintain a collection of numbers S that supports *Build*(S), *Insert*(x), *Extract*(), and *Peek*() operations, defined as follows:

- *Build*(S): Produces, in linear time, a data structure *Median − Heap* from an unordered input array S. For describing *Build*(S), you can assume access to the procedure *Find_Median*(S), which finds the median of S in linear time.
- *Insert*(x): Insert element x into *Median − Heap* in O(log n) time.
- *Peek*(): Returns, in O(1) time, the value of the median of *Median − Heap*.
- *Extract*(): Remove and return, in O(log n) time, the value of the median element in *Median − Heap*.

The Median-Heap data structure is designed to maintain a collection of numbers S and efficiently support the Build(S), Insert(x), Extract(), and Peek() operations. The data structure ensures that the median of the numbers in S can be accessed in O(1) time, and the other operations are performed in O(log n) time complexity.

The Median-Heap data structure consists of two heaps: a Max-Heap and a Min-Heap. The Max-Heap stores the lower half of the numbers, while the Min-Heap stores the upper half. The Max-Heap is designed such that the maximum element is always at the root, while the Min-Heap has the minimum element at the root.

Here is a detailed description of each operation:

1. Build(S):
    - Create an empty Max-Heap and an empty Min-Heap.
    - Iterate through each element x in S:
        - Insert x into either the Max-Heap or the Min-Heap, depending on its value and the current size of both heaps.
        - Maintain the heap property for each heap after each insertion.
    - If the size of the Max-Heap is greater than the size of the Min-Heap, remove the maximum element from the Max-Heap and insert it into the Min-Heap.
    - The Median-Heap is now constructed.

    The Build(S) operation has a linear time complexity of O(n) because each element is inserted into the heaps individually.
2. Insert(x):
    - Compare x with the root of the Max-Heap.
    - If x is smaller than the root, insert x into the Max-Heap.
    - Otherwise, insert x into the Min-Heap.
    - Rebalance the heaps by ensuring that the size difference between the Max-Heap and Min-Heap is at most 1.

    The Insert(x) operation has a logarithmic time complexity of O(log n) because the element is inserted into one of the heaps, and the heaps are then rebalanced if necessary.
3. Peek():
    - Return the root element of the Max-Heap, which is the median element.

    The Peek() operation has a constant time complexity of O(1) since the median is stored at the root of the Max-Heap.
4. Extract():
    - Remove the root element from the Max-Heap.
    - Rebalance the heaps if necessary.

- Return the extracted element.

The Extract() operation has a logarithmic time complexity of O(log n) because the element is removed from the Max-Heap, and the heaps are then rebalanced if necessary.

The Median-Heap data structure maintains the property that the lower median is always at the root of the Max-Heap. This property ensures that the Peek() operation can access the median in constant time. The insertion and extraction operations are performed efficiently by maintaining the balance between the Max-Heap and Min-Heap.

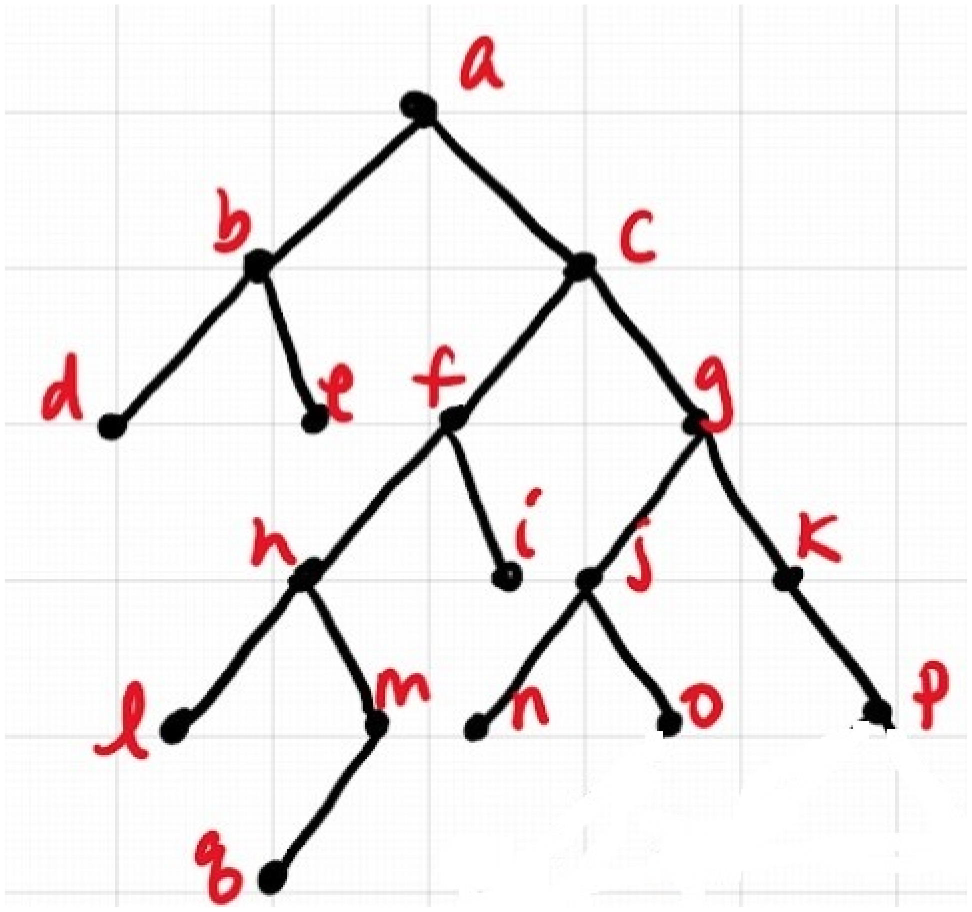
Overall, the Median-Heap data structure provides efficient operations to build, insert, extract, and peek the median, with time complexities of O(n), O(log n), O(log n), and O(1) respectively.

Psuedo Code:

class MedianHeap:
def **init**(self):
self.max_heap = MaxHeap() # Max-Heap to store lower half of the numbers
self.min_heap = MinHeap() # Min-Heap to store upper half of the numbers

```
class MedianHeap:
def init(self):
self.max_heap = MaxHeap()  # Max-Heap to store lower half of the numbers
self.min_heap = MinHeap()  # Min-Heap to store upper half of the numbers

def build(self, S):
    for x in S:
        if self.max_heap.is_empty() or x < self.max_heap.peek():
            self.max_heap.insert(x)
        else:
            self.min_heap.insert(x)

    if self.max_heap.size() > self.min_heap.size():
        max_value = self.max_heap.extract_max()
        self.min_heap.insert(max_value)

def insert(self, x):
    if self.max_heap.is_empty() or x < self.max_heap.peek():
        self.max_heap.insert(x)
    else:
        self.min_heap.insert(x)

    # Rebalance the heaps if necessary
    if self.max_heap.size() - self.min_heap.size() > 1:
        max_value = self.max_heap.extract_max()
        self.min_heap.insert(max_value)
    elif self.min_heap.size() - self.max_heap.size() > 1:
        min_value = self.min_heap.extract_min()
        self.max_heap.insert(min_value)

def peek(self):
    return self.max_heap.peek()

def extract(self):
    median = self.max_heap.extract_max()

    # Rebalance the heaps if necessary
    if self.max_heap.size() < self.min_heap.size():
        min_value = self.min_heap.extract_min()
        self.max_heap.insert(min_value)

    return median
```

3.



Starting with the root vertex a, we can use Breadth-First Search (BFS) or Depth-First Search (DFS) to pass through all of the vertices in this tree.

Whenever we have more than one option, we always pick the vertex that appears earlier in the alphabet. For example, from vertex a, we go to b instead of c.

Clearly explain the difference between Breadth-First Search and Depth-First Search, and determine the order in which the 17 vertices are reached using each algorithm.

Breadth-first search:

- BFS starts at a given vertex (the source vertex) and **explores all its neighbors** before moving on to the next level of neighbors.
- It visits vertices in a **level-by-level** manner, exploring all vertices at the current level before moving to the next level.
- BFS uses a **queue** data structure to keep track of the vertices to be explored.

- **It guarantees that the shortest path** between the source vertex and any other vertex is found, given an unweighted graph.
- BFS is typically implemented using an iterative approach.
- We get the following results from a breadth-first search: {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q}

Depth-First Search (DFS):

- DFS starts at a given vertex (the source vertex) and **explores as far as possible** along each branch before backtracking.
- It explores **vertices as deeply as possible** before moving on to the next branch.
- DFS uses **a stack** (or recursion) to keep track of the vertices to be explored.
- It is often used to explore all connected vertices in a graph or to search for a specific vertex or path.
- DFS **does not guarantee to find the shortest path** in a graph.
- We get the following results from a breadth-first search: {a, b, d, e, c, f, h, l, m, q, i, g, j, n, o, k, p}

3.2 (5 Points)

Let T be an undirected binary tree with n vertices. Write an efficient algorithm that shows how you can walk through the tree by crossing each edge of T exactly twice: once in each direction.

Clearly explain how your algorithm works, why each edge is guaranteed to be crossed exactly twice, and determine the running time of your algorithm.

To walk through an undirected binary tree T and cross each edge exactly twice, we can use a modified version of the Depth-First Search (DFS) algorithm. Here's an efficient algorithm that accomplishes this:

1. Start at the root node of the binary tree T.
2. Mark the current node as visited.
3. For each child (left and right) of the current node:
    - If the child is not visited, recursively visit the child.

        - After visiting the child, traverse back to the current node.
4. Mark the current node as visited again.
5. Repeat steps 3-4 for all unvisited children of the current node.
6. Once all children have been visited and marked twice, move back to the parent node.
7. Repeat steps 3-6 until all nodes in the binary tree T have been visited and marked twice.

The algorithm guarantees that each edge of the binary tree T will be crossed exactly twice for the following reasons:

- In step 2, we mark the current node as visited before traversing its children. This ensures that each edge leading to a child is crossed once in the downward direction.
- In step 4, after traversing all the children, we mark the current node as visited again. This ensures that each edge leading back to the parent node is crossed once in the upward direction.
- By repeating steps 2-4 for all nodes in the tree, we ensure that each edge is crossed exactly twice: once in the downward direction and once in the upward direction.

The running time of this algorithm is O(n), where n is the number of vertices in the undirected binary tree T. Since we visit each node and each edge exactly twice, the time complexity is linear in the size of the tree.

3.3 (5 Points)

Let be the undirected binary tree from problem 3.1. For each pair of vertices, we can compute the distance between these vertices. In the binary tree above, for example, we have dist(d,i)=5 and dist(l,o)=6. We define the **diameter** of T to be
the maximum value of , chosen over all pairs of vertices x and y in T

Clearly explain why the diameter of the above tree is 7.

To compute a diameter for an undirected tree is that to know the distance between every two nodes having in the tree and returning it if we see in the question above if we compute all the lengths between two nodes we get the following distances and we will find the max distance and that's the diameter

a-b distance is 1

a-c distance is 1

a-d distance is 2

…

a-q distance is 5

same with B

…

if we get the distance between the nodes

d-q distance is 7 ⇒ d → b → a → c → f → h → m → q

similarly, if we compute the distance between the nodes

q-p distance is ⇒ q → m → h → f → c → e → n → p

distance is 7 and thus the maximum distance and the diameter of the above tree is 7

3.4 (5 Points)

Let T be an undirected binary tree with n vertices. Create an algorithm to compute the diameter of T. Clearly explain how your algorithm works, why it guarantees the correct output, and determine the running time of your algorithm.

To compute the diameter of an undirected binary tree T with n vertices, we can use a depth-first search (DFS) algorithm. The diameter of a tree is defined as the maximum distance between any two vertices in the tree. Here's how the algorithm works:

1. Initialize a global variable "diameter" to 0.
2. Start a DFS from any arbitrary vertex in the tree (e.g., the root).
3. In the DFS function, perform the following steps:
    - Recursively visit each child of the current vertex.
    - For each child, calculate the height of its subtree (the longest path from that child to any leaf in its subtree) by recursively calling the DFS.

- Update the "diameter" variable by comparing it with the sum of the heights of the two tallest subtrees plus 2 (i.e., the maximum diameter found so far).
- Return the height of the current subtree (the longest path from the current vertex to any leaf in its subtree).

4. After the DFS is complete, the "diameter" variable will hold the diameter of the tree.

The algorithm guarantees the correct output by considering all possible paths between any two vertices in the tree. It computes the height of each subtree and updates the diameter based on the sum of the heights of the two tallest subtrees plus 2 (to account for the two edges connecting the subtrees). By the end of the DFS traversal, the "diameter" variable will hold the longest path found in the tree, which corresponds to the diameter.

The running time of this algorithm is O(n), where n is the number of vertices in the binary tree. The DFS visits each vertex once, and the calculation of the height for each subtree takes constant time. Therefore, the time complexity is linear in the number of vertices in the tree.

Here's an implementation of the algorithm in Python:

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

def compute_diameter(root):
    diameter = 0  # Global variable to hold the diameter

    def dfs(node):
        nonlocal diameter
        if not node:
            return 0

        left_height = dfs(node.left)
        right_height = dfs(node.right)

        # Update the diameter if necessary
        diameter = max(diameter, left_height + right_height + 2)

        # Return the height of the subtree rooted at the current node
        return max(left_height, right_height) + 1

    dfs(root)
    return diameter
```

You can create an undirected binary tree and call the compute_diameter function, passing the root node, to get the diameter of the tree.

Note: The algorithm assumes that the tree is an undirected binary tree. If the input is not guaranteed to be an undirected binary tree, you may need to modify the algorithm accordingly.