Name / Ahmed Mohamed Mustafa Ghalwash

Department / CS

Section / 1

Solve Assignment

(i) Heap-Sort algorithm

a.Required Algorithms for Heap Sort

1. **Build Max Heap**: Convert the input array into a max heap.
   - For an element at index i, ensure that it is larger than its children.
   - Use a heapify function recursively to fix violations of the heap property from bottom to top.
2. **Heapify**: Rearrange the heap to maintain the max heap property.
   - For a node i in the array, compare it with its left and right children (indices 2i + 1 and 2i + 2).
   - Swap it with the largest child if a violation is found and call heapify recursively on the affected subtree.
3. **Heap Sort**:
   - Build the max heap using the Build Max Heap function.
   - Iteratively remove the largest element (root) and place it at the end of the array.
   - Reduce the heap size and call heapify on the root node after each removal.

b. Analysis

1. **Time Complexity**:
   - Building the heap: *O(n)*
   - Heapify (logarithmic operation): *O(log n)*, applied n−1 times during extraction.
   - Overall: *O(n log n)*.
2. **Space Complexity**:
   - In-place sorting: O(1) auxiliary space.
   - Does not require additional memory beyond the input array.
3. **Advantages**:
   - Suitable for large datasets due to its efficiency.
   - In-place sorting saves memory.
4. **Disadvantages**:
   - Not stable: Relative order of equal elements is not preserved.
   - Complexity in implementation compared to simpler algorithms like Merge Sort.

C. Implementation(C#)

```
using System;
class Program
{
  static void Main(string[] args)
  {
    int[] data = { 3, 1, 4, 1, 5, 9, 2, 6, 5 };
    HeapSort(data);

    Console.WriteLine("Sorted array: " + string.Join(", ", data));
  }

  static void Heapify(int[] arr, int n, int i)
  {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
    {
      largest = left;
    }

    if (right < n && arr[right] > arr[largest])
    {
      largest = right;
    }

    if (largest != i)
    {
      int swap = arr[i];
      arr[i] = arr[largest];
      arr[largest] = swap;

      Heapify(arr, n, largest);
    }
  }

  static void HeapSort(int[] arr)
  {
    int n = arr.Length;

    for (int i = n / 2 - 1; i >= 0; i--)
    {
      Heapify(arr, n, i);
    }
```

```
        for (int i = n - 1; i > 0; i--)
        {

            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            Heapify(arr, i, 0);
        }
    }

}
```

(ii) Kruskal’s algorithm

a.Required Algorithms for Heap Sort

1. **Input Representation**: Represent the graph with edges in the form of (source, destination, weight).
2. **Sorting Edges**: Sort all edges by weight in non-decreasing order.
3. **Disjoint Set Functions**:
    - `Find(u)`: Find the root of the subset to which element u belongs.
    - `Union(u, v)`: Merge two subsets.
4. **Main Kruskal's Algorithm**:
    - Initialize MST as empty.
    - For each edge (in sorted order):
        - If the edge doesn’t form a cycle (check using `Find`):
            - Add the edge to the MST.
            - Perform `Union` on the two sets.

b. Analysis

- **Time Complexity**:
    - Sorting edges: *O(E log E)*, where E is the number of edges.
    - Union-Find operations: Nearly *O(1)* per operation with path compression.
    - Overall: *O(E log V)*, where *V* is the number of vertices.
- **Space Complexity**: *O(V+E)* for edge list and disjoint set structures.
- **Greedy Nature**: Ensures minimum-cost edge is added while maintaining MST properties.

C. Implementation(C#)

```
using System;
using System.Collections.Generic;
```

```
namespace Kruskal
{
  internal class Program
  {
    static void Main(string[] args)
    {
      Graph graph = new Graph(4);

      graph.AddEdge(0, 1, 10);
      graph.AddEdge(0, 2, 6);
      graph.AddEdge(0, 3, 5);
      graph.AddEdge(1, 3, 15);
      graph.AddEdge(2, 3, 4);

      graph.KruskalMST();
    }
  }

  class Graph
  {
    private int vertices;
    private List<Edge> edges;

    public Graph(int vertices)
    {
      this.vertices = vertices;
      edges = new List<Edge>();
    }

    public void AddEdge(int source, int destination, int weight)
    {
      edges.Add(new Edge { Source = source, Destination = destination, Weight = weight });
    }

    private int Find(int[] parent, int i)
    {
      if (parent[i] != i)
        parent[i] = Find(parent, parent[i]);
      return parent[i];
    }

    private void Union(int[] parent, int[] rank, int x, int y)
    {
      int xRoot = Find(parent, x);
```

```
    int yRoot = Find(parent, y);

    if (rank[xRoot] < rank[yRoot])
        parent[xRoot] = yRoot;
    else if (rank[xRoot] > rank[yRoot])
        parent[yRoot] = xRoot;
    else
    {
        parent[yRoot] = xRoot;
        rank[xRoot]++;
    }
}

public void KruskalMST()
{
    edges.Sort();

    int[] parent = new int[vertices];
    int[] rank = new int[vertices];
    for (int v = 0; v < vertices; v++)
    {
        parent[v] = v;
        rank[v] = 0;
    }

    List<Edge> mst = new List<Edge>();

    foreach (var edge in edges)
    {
        int x = Find(parent, edge.Source);
        int y = Find(parent, edge.Destination);

        if (x != y)
        {
            mst.Add(edge);
            Union(parent, rank, x, y);
        }
    }

    Console.WriteLine("Edges in the MST:");
    foreach (var edge in mst)
    {
        Console.WriteLine($"{edge.Source} -- {edge.Destination} == {edge.Weight}");
    }
}
```

```
        class Edge : IComparable<Edge>
        {
            public int Source, Destination, Weight;

            public int CompareTo(Edge other)
            {
                return this.Weight.CompareTo(other.Weight);
            }
        }
    }

}
```