1. We will add a single attribute *ptr* to each node.
   For node x, $x.ptr$ leads to a linked list item. The linked list is a circular doubly linked list which contains the labels to all elements in x's set.
   PRINT-SET(x) hence requires constant time, i.e O(1) time to get the list of all elements in x's set. Since printing each element is O(1), the PRINT-SET(x) operation is linear in the number of elements in the set of x.
   The addition of this pointer affects the other operations as follows:
   - MAKE-SET(x):
     In addition to usual MAKE-SET(x) operations, this implementation will require we create a new linked list, set the label of the linked list item to x and store the pointer of the linked list object in $x.ptr$.
     These operations are constant time, i.e O(1) so this does not affect the asymptotic runtime of MAKE-SET(x)
   - FIND-SET(x):
     In this implementation, FIND-SET(x) does not involve any operations to the linked list hence its asymptotic runtime is not affected.
   - UNION(x, y):
     Assuming you are merging set y into x, the extra operations in this implementation involve appending the linked list of y to x, which is constant time in a circular doubly linked list. After the linked lists are merged, each node will point to the linked list item which has its label.

2. (a) Let T be a minimum spanning tree. Let (u, v) be an edge in this tree, with weight $w(u, v)$ which is the max weight of all the edges in the MST. Let S be the set of all vertices in T, of which s_1 is the set of vertices whose 'next hop' from u is v. Let s_2 be $S - s_1$.
Since a minimum spanning tree cannot have cycles, s_1 and s_2 have to be connected by a single edge, (u, v), which has to be minimum cost.
Hence there is no other spanning tree which can connect the nodes with max edge weight being less than $w(u, v)$. This proves that no bottleneck spanning tree with value less than $w(u, v)$ can exist.
Hence a minimum spanning tree is a bottleneck spanning tree.

(b) To check if a graph G has a bottleneck spanning tree with value b in a linear time algorithm we can us the following algorithm:

```
1. For each edge E in G
     1.1 If edge E has weight greater than b remove it from Graph
2. Let Q be the set of all vertices in G
3. Let S be a queue with a single vertex from Q
4. While S is not empty
     4.1 Pop a random vertex V from S
     4.2 Remove node V from Q
     4.3 For Add all neighbors N of V that exist in Q
          4.3.1 Push N into S, if they don't already exist
5. If set Q is empty return true else false
```

The above algorithm will be linear in number of edges, since each edge is checked once against weight b, then again once when existence of the Spanning Tree is checked.

(c) We could use the following algorithm on input Graph G:

```
1. Let E be the edges in graph G.
2. Let W be the median of weights of |E|
3. Run the Algorithm in 2(b) with G & W
     3.1 If a bottleneck spanning tree exists, remove all edges with
     weight > W, and repeat this algorithm with G'.
     3.2 If a bottleneck spanning tree does not exist, use MST-Reduce
     on the forest of edges with weights less then median,
     and run this algorithm again on G'.
```

The above will be liner in the number of edges as:
$O(E + E/2 + E/4 + E/8 + ... + 1) = O(E)$

3. (a) In the proposed algorithm, we track usp status for every vertex with an additional array data structure which contains a true or false value about whether a path is unique shortest path.
Initially the usp of each vertex is initialized to true. The usp of source node is assumed as true.
The usp status of a vertex is modified in one of two cases:
    - Path with same length as shortest path is found:
      We set the usp of vertex to false.
    - New Shortest Path:
      When a new shortest path is found we set the usp status of vertex to the usp status to the immediate parent of the vertex in the new path.

   (b) The pseudo-code is as follows:

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:
            dist[v] := infinity
            previous[v] := undefined
            usp[v]  := true
    dist[source] := 0
    Q := the set of all nodes in Graph
    while Q is not empty:
            u := node in Q with smallest dist[ ]
            remove u from Q
            for each neighbor v of u:
                    alt := dist[u] + dist_between(u, v)
                    if alt == dist[v]
                            usp[v] = false
                    if alt < dist[v]
                            dist[v] := alt
                            previous[v] := u
                            usp[v] = usp[u]
    return (previous[], dist[], usp[])
```

   (c) Algorithm run on graph of 5 vertices:

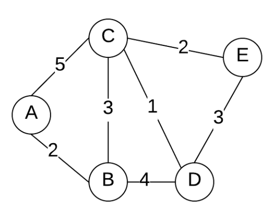

Figure 1: A graph with 5 vertices

		B			C			D			E		
Step	Q	dist	prev	usp	dist	prev	usp	dist	prev	usp	dist	prev	usp
0	A,B,C,D,E	∞	-	T	∞	-	T	∞	-	T	∞	-	T
1	B,C,D,E	2	A	T	5	A	T	∞	-	T	∞	-	T
2	C,D,E	2	A	T	5	A	F	6	B	T	∞	-	T
3	D,E	2	A	T	5	A	F	6	B	F	7	C	F
4	E	2	A	T	5	A	F	6	B	F	7	C	F
5	-	2	A	T	5	A	F	6	B	F	7	C	F

(d) Correctness Proof of *usp*:
From the correctness proof of Dijkstra's Algorithm[1] we use notation $\delta(s, u)$ to define a shortest path between s and u, $u.d$ to define the numeric value of the sum of weights of $\delta(s, u)$ and the conclusion that if u is added to S then $u.d = \delta(s, u)$.
By definition, if any new path to u is found which is equal to $u.d$ then $usp(u) = false$.
Now $usp(u)$ depends on the usp status of all the nodes in $\delta(s, u)$. For a vertex y, if $usp(y)$ is false then all v such that $\delta(s, v)$ passes through y, $usp(v) = false$. Also, all items v in path $\delta(y, u)$, $usp(v) = false$
We use the observation from correctness proof of Dijkstra's Algorithm, that if u is from outside S then the discovered path is minimum distance to u, to prove that if u is added to S, then $usp(parent(u))$ cannot change, i.e since $parent(u).d < u.d$, and $parent(u)$ lies in S.
Hence $usp(u) = usp(parent(u))$ is a valid simplification to the rule where all vertices between u and s have to checked for being *usp* to figure out $usp(u)$.

(e) The runtime of Dijkstra's Algorithm is not affected by adding the usp rules, since creating the usp array is at worst $O(v)$, and setting the status of usp for every e is $O(1)$; which are both asymptotically slower than $\Theta(E \lg V)$.
Hence time bound of this algorithm is $\Theta(E \lg V)$.

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.