least # of edges

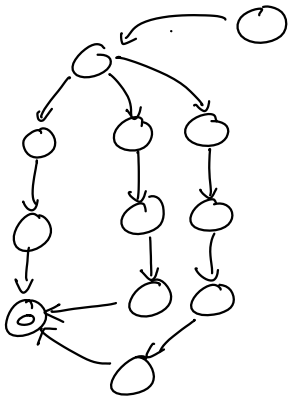
Hashtable-based

```
Function shortestPathBFS(g, src, dst):
    exploration ← new Queue (LinkedQueue)
    exploration.enqueue(src)
    previous ← new Dictionary (HashTable)
    previous.insert(src, src)
    While exploration is not empty:
        current ← exploration.dequeue()
        If current == dst:
            path ← new List
            While current ≠ src:
                path.insertFirst(current)
                current ← previous.get(current)
            EndWhile
            path.insertFirst(src)
            Return path
        EndIf
        For neighbor In g.getNeighbors(current):
            If neighbor is not a key in previous:
                previous.insert(neighbor, current)
                exploration.enqueue(neighbor)
            EndIf
        EndFor
    EndWhile
EndFunction
```

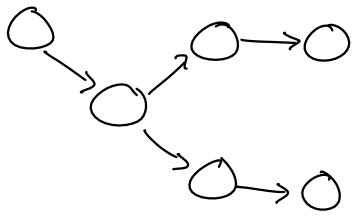
O(|V|)

O(|V|)

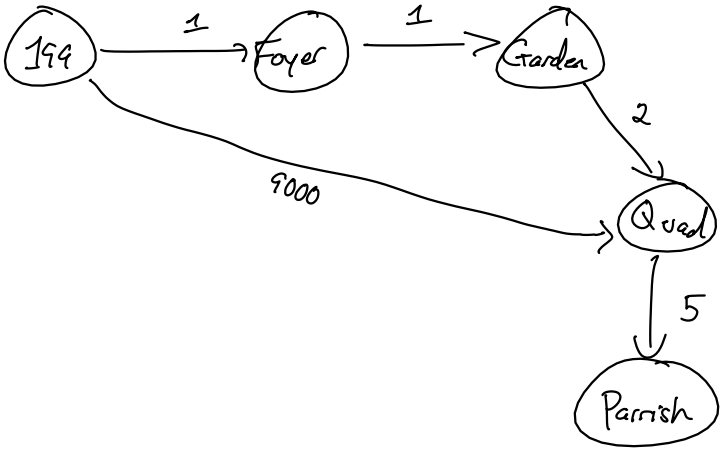
O(|E|)



```
Function allBFS(g, src):                    <- returns dictionary mapping each vertex to prev
    exploration <- new Queue
    exploration.enqueue(src)
    previous <- new Dictionary
    previous.insert(src, src)
    While exploration is not empty:
        current <- exploration.dequeue()
        For each neighbor In g.getNeighbors():
            If neighbor is not a key in previous:
                previous.insert(neighbor, current)
                exploration.enqueue(neighbor)
            EndIf
        EndFor
    EndWhile
    Return previous
EndFunction
```



O(|V|) O(|V|) O(|E|)




```
Function sssp(g, src):  <-- returns Dictionary mapping vertices to costs
    exploration <- new MinHeap()
    exploration.insert(0, src)
    costs <- new Dictionary()
    costs.insert(src, 0)
    While exploration is not empty:
        current <- exploration.remove()
        currentCost <- costs.get(current)
        For each edge In g.getOutgoingEdges(current):
            neighbor <- edge.destination
            newCost <- currentCost + edge.weight
            If neighbor not a key in costs:
                costs.insert(neighbor, newCost)
                exploration.insert(newCost, neighbor)
            Else If costs.get(neighbor) > newCost:
                costs.update(neighbor, newCost)
                exploration.insert(newCost, neighbor)
            EndIf
        End For
    EndWhile
    Return costs
EndFunction
```