Dijkstra's Shortest Path Tree Algorithm
This algorithm is nearly identical to one for finding the minimum spanning tree. Both use Priority or Best First Search for Vertices Approach.

Example from Sedgewick

On running the algorithm for this graph and inserting output code into the algorithm, the following output was obtained:
Graph::SPT_Dijkstra( vertex s )
Begin
  // G = (V, E)
  for each v ∈ V
    dist[v] = ∞
    parent[v] = null  // have a special null vertex
    hpos[v] = 0  // indicates that v ∉ heap
  pq = new Heap  // priority queue (heap) initially empty
  v = s  // s will be the root of the shortest path tree
  dist[s] = 0

  while v ≠ null  // Loop should repeat |V|-1 times
    for each u ∈ adj(v)  // Examine each neighbour u of v
      d = wgt(v,u)
      if dist[v] + d < dist[u]
        dist[u] = dist[v] + d  // After changing dist[u] either insert
        if u ∉ pq  // it or sift it up in the heap and record
          pq.insert( u)  // its new parent vertex.
        else
          pq.siftUp( hpos[u])
        end if
      end if
    end for
    v = pq.remove()  // while end
  return parent
End
Complexity
- Heap contains vertices. So a heap operation requires a maximum of $\log_2 V$ steps.
- Initialisation for loop – requires $V$ steps
- Each node bar the first is inserted into heap once - requires at most $V \log_2 V$ steps
- Each edge is examined once and may modify a vertex’s position on the heap, a siftUP() which requires at most $E \log_2 V$

Therefore
\[
\text{complexity} = O(V + V \log_2 V + E \log_2 V ) \\
\approx O((V + E) \log_2 V)
\]

For a **sparse** graph $E \approx kV$ (k some constant), so complexity $\approx O((V + kV) \log_2 V )$ 
\[
= O(V \log_2 V)
\]

For a **dense** graph $E \approx V^2$, so complexity $\approx O((V + V^2) \log_2 V )$ 
\[
= O(V^2 \log_2 V)
\]

Data Structures Required
- a C++ class called **Graph** to encapsulate most of the data structures and methods.
- a Node structure. Use these to store graph edges in the linked lists.
- array of linked lists to store the graph in memory. Declared with 
  \[
  \textbf{Node ** adj;}
  \]
- an int array $\text{dist}[ ]$ to record the current distance of a graph vertex from the starting vertex $s$.
- an int array $\text{parent}[ ]$ to store the SPT.
- a priority queue or heap $pq$ to find the next vertex nearest $s$ so that it can be added to the SPT. $pq$ will contain an internal heap array $h[ ]$.
- an int array $hPos[ ]$ which records the position of any vertex with the heap array $h[ ]$. So vertex $v$ is in position $hPos[v]$ in $h[ ]$. 

Dijkstra Shortest Path Tree Algorithm
Implementation Comments

- See notes on MST algorithm. Both algorithms are very similar.

- A possible class declaration is:

```cpp
class AdjGraph {
    private:
        struct Node {
            int vert;
            int wgt;
            Node * next;
        };
        int V, E;
        Node * z;
        Node **adj;

    public:
        AdjGraph();
        AdjGraph(char filename[]);
        int * SPT_Dijkstra( int s);
};
```

- In the main method could have

```cpp
AdjGraph g;
spt = g.SPT_Dijkstra( 1);```