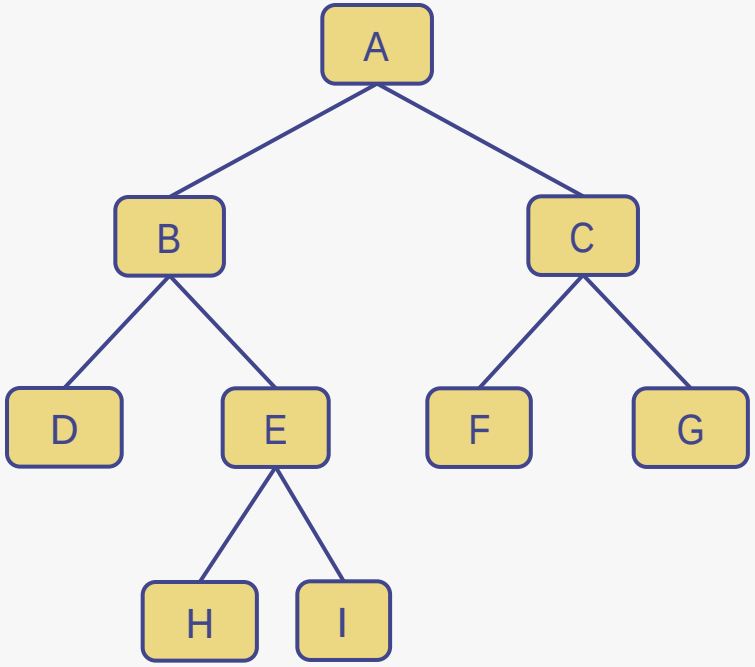
Searching

Binary Trees recap

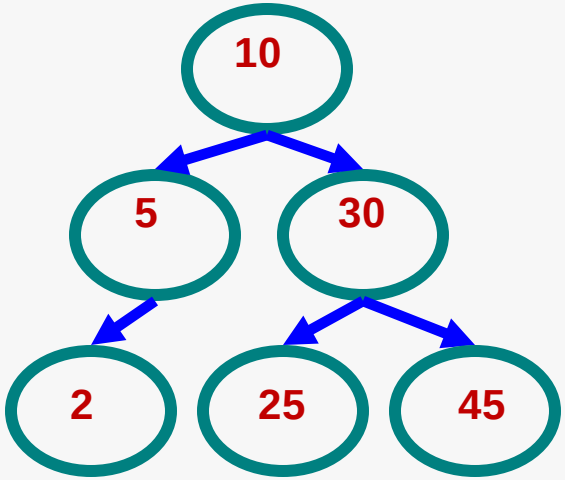
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for a ***proper*** binary tree)
  - We call the children of an internal node ***left child*** and ***right child***
  - The left and right subtrees are also binary search trees

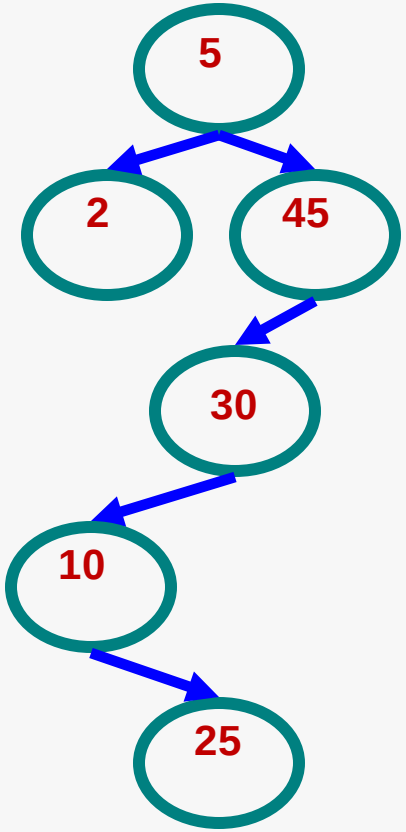


Binary Search Trees (BST)

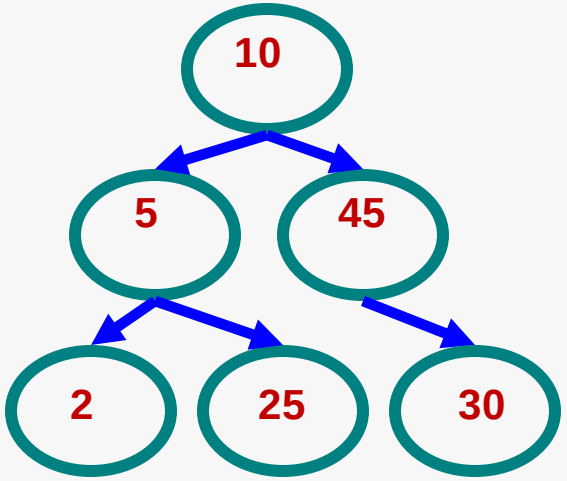
- BSTs are a type of binary tree with a *special organization of data*
- This data organization leads to O(log n) complexity for searches, insertions and deletions in certain types of the BST (balanced trees).

Binary Search Trees





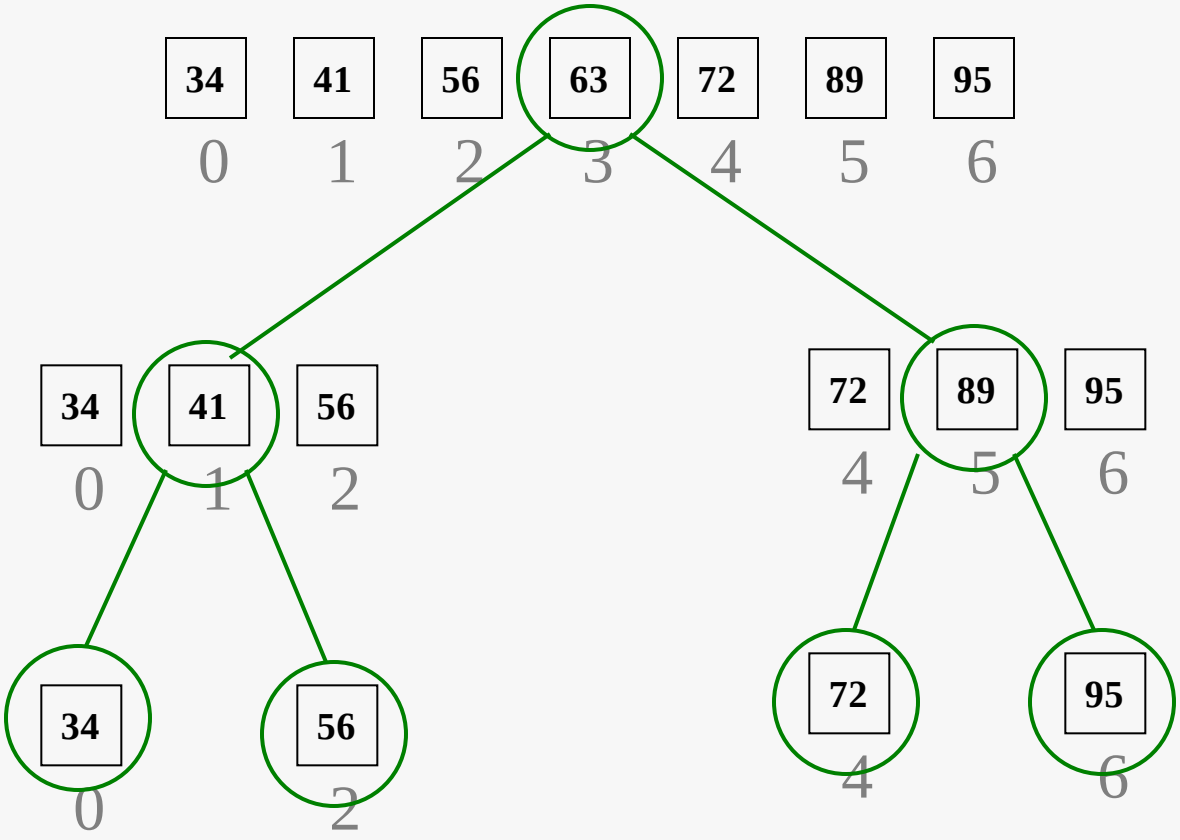
Binary search trees



Non-binary search tree – why?

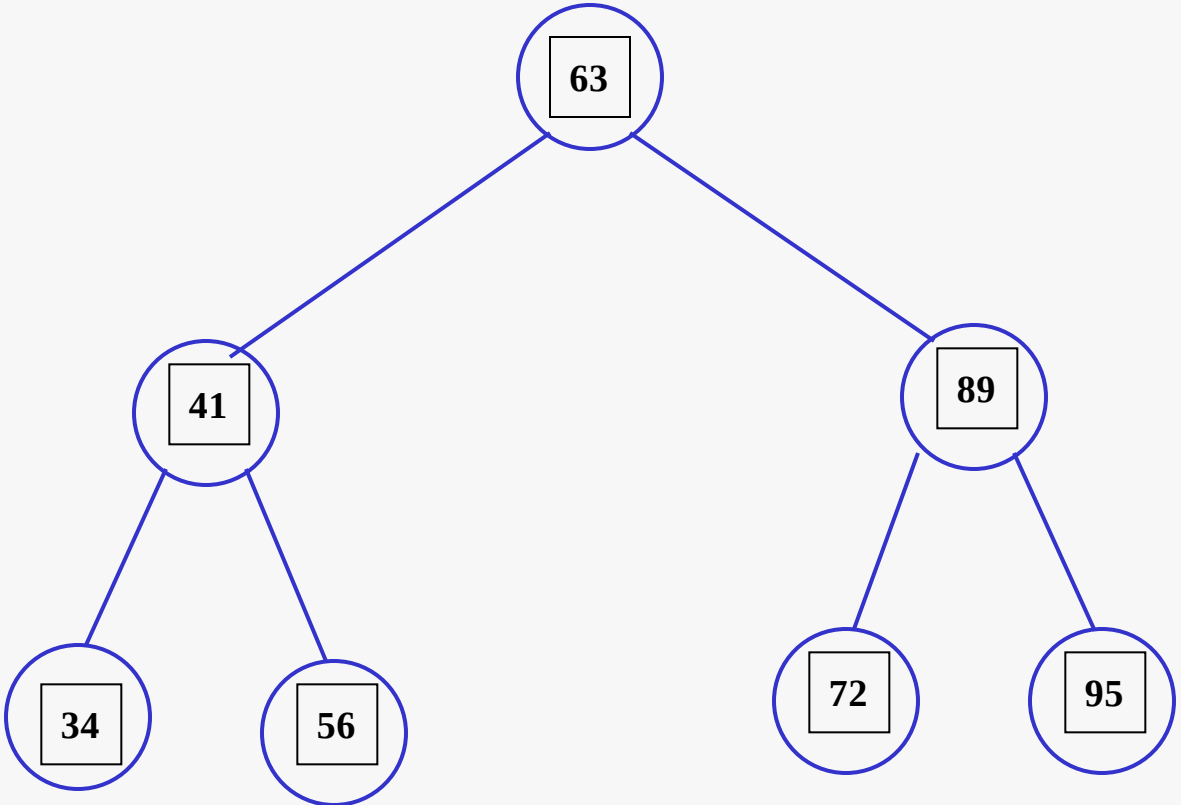
Binary Search Algorithm

Binary Search algorithm of an array of ***sorted*** items reduces the search space by one half after each comparison



Organization Rule for BST

- the values in all nodes in the left subtree of a node are less than the node value
- the values in all nodes in the right subtree of a node are greater than the node values



BST Operations: *Search*

- implements the binary search based on comparison of the items in the tree
- the items in the BST must be **comparable** (e.g integers, string, etc.)
- The search starts at the root, then
  - it probes down, comparing the values in each node with the target, till it finds the first item equal to the target
  - returns this item or `null` if there is none

pseudo code for search in BST

if the tree is empty

    return `null`

else if the item in the node equals the target

    return the node value

else if the item in the node is greater than the target

    return the result of searching **recursively** the ***left subtree***

else if the item in the node is smaller than the target

    return the result of searching **recursively** the ***right subtree***

BST Operations: *Insertion*

- places a new item near the "frontier" of the BST while retaining its organization of data:
  - **starting at the root** it probes down the tree till it finds a node whose left or right pointer is empty and is a logical place for the new value
  - uses a binary search to locate the insertion point
  - is based on comparisons of the new item and values of nodes in the BST

    NOTE: contents in nodes must be **comparable**

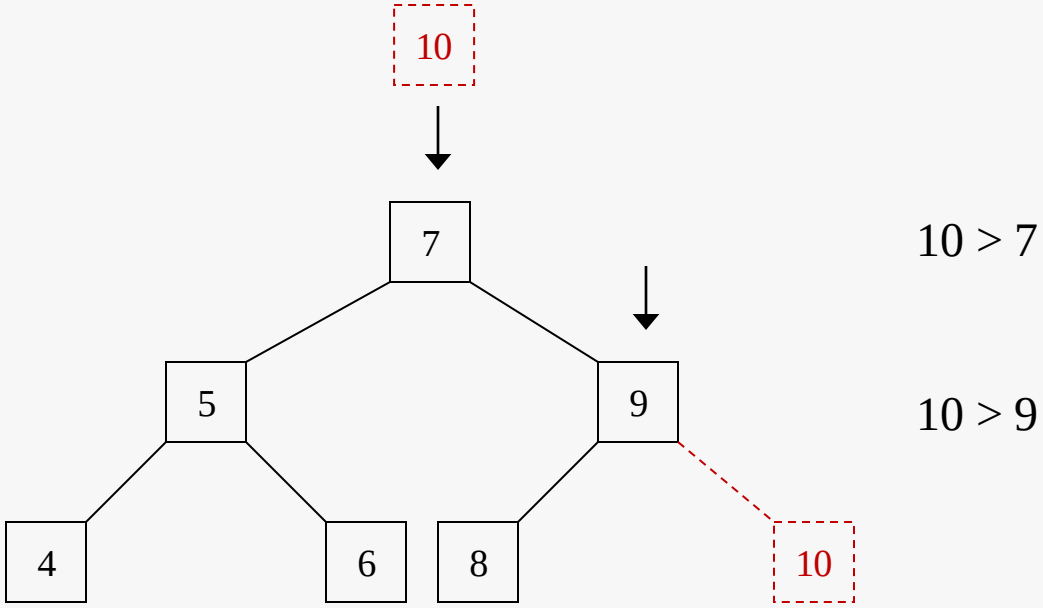
Insertion in BST – two scenarios

Scenario 1: *The tree is empty*

- Set the root to a new node containing the item

Scenario 2: *The tree is not empty*

- Recursively call a method to locate then insert the item



pseudo code for Insertion in BST

```
if  tree is empty   (scenario 1)
      create a root node with the new key
else    (scenario 2)
      // compare key with the top node
      if  key =  node key
            duplicate found so do not do anything
      else if  key >  node key
            // compare key with the right subtree key:
             if  subtree is empty create a leaf node and add key information
             else recursively call algorithm with right subtree
       else  key <  node key
            // compare key with the left subtree:
             if  subtree is empty create a leaf node and add key information
             else recursively call algorithm with left subtree
```

BST Shapes

- The order of supplying the data determines where it is placed in the BST, which determines the **shape** of the BST
- Create BSTs from the same set of data presented each time in a different order:

(a) `17 4 14 19 15 7 9 3 16 10`

(b) `9 10 17 4 3 7 14 16 15 19`

(c) `19 17 16 15 14 10 9 7 4 3`

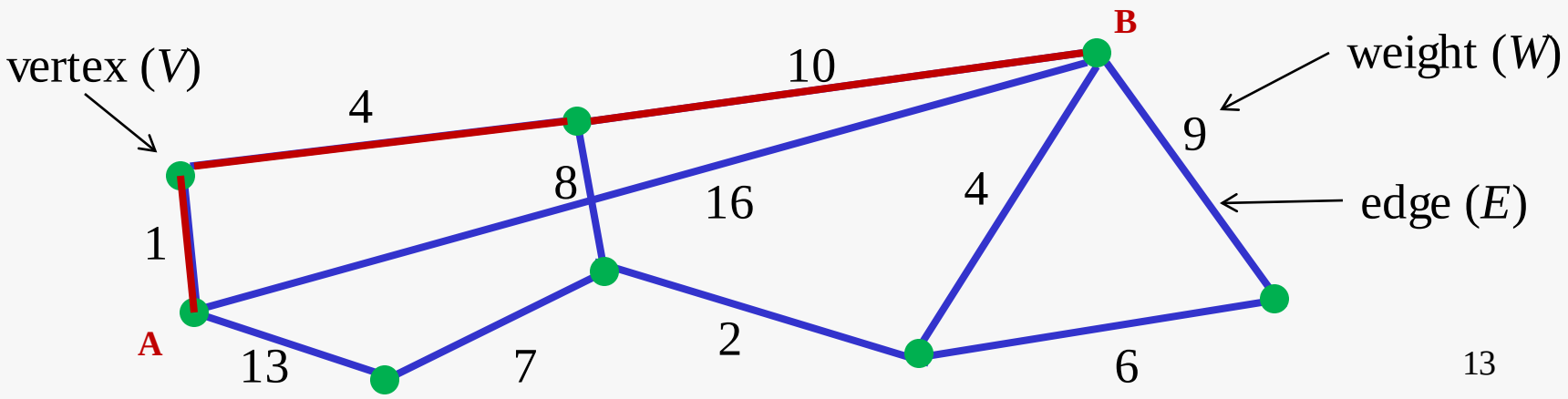
Did you identify the shape of (c) before you started to draw it?

The use of graphs in searching

- Weights can be distance, money, load, etc
- Digraph $G = (V,E)$ with weight function $W: E \rightarrow R$ (assigning real values to edges)
- Weight of path $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

$w(p_{AB})=1+4+10=15$



Minimum Spanning trees

- Suppose that G is a weighted graph. A *minimum spanning tree* is a spanning tree of G which **minimises** the weights on the edges in the tree.
- **Kruskal's** algorithm (also known as the greedy algorithm) finds a spanning tree with the minimum weight
- There may be more than one spanning tree with this smallest weight

Example use of Kruskal's algorithm

Problem:

We want to know how many kilometres of cable is required to give complete coverage of a company's PC network ? This means that communication can be made between any two users using two or more PCs.

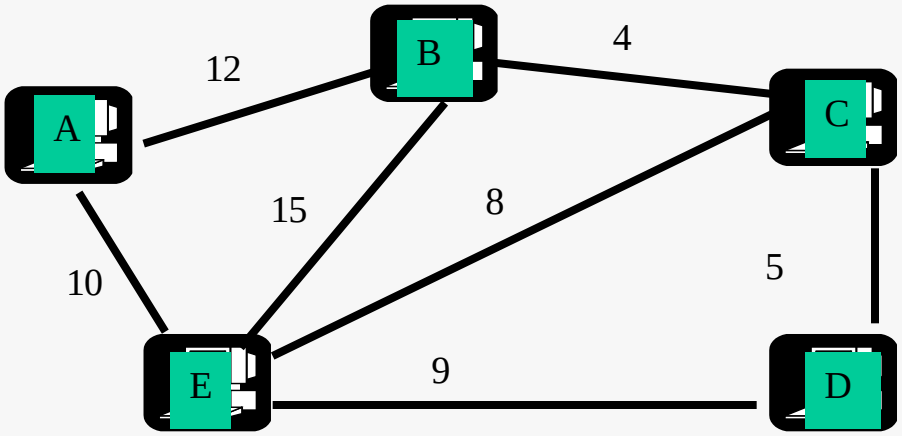
Kruskal's Algorithm

This method builds the tree as you go along starting with just the vertices

Algorithm:

1. The graph *T* initially consists of the vertices of *G* and no edges.
2. Add an edge to *T* having minimum weight but **which does not give a cycle in *T*.**
3. Repeat 2 until T has *n* - 1 edges.

Kruskal's Algorithm



Step 1

Label vertices

Step 2

Write a list of edges

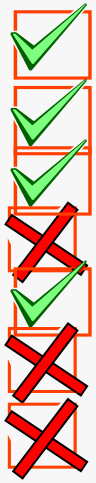
in order of magnitude

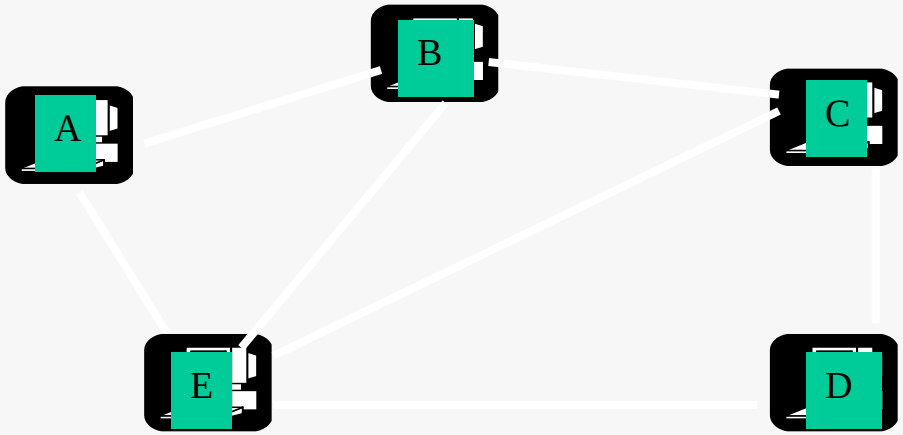
Step 3

Build a spanning tree from the vertices

Accepting each edge in order **if no cycle is formed**

BC	4
CD	5
CE	8
DE	9
AE	10
AB	12
BE	15

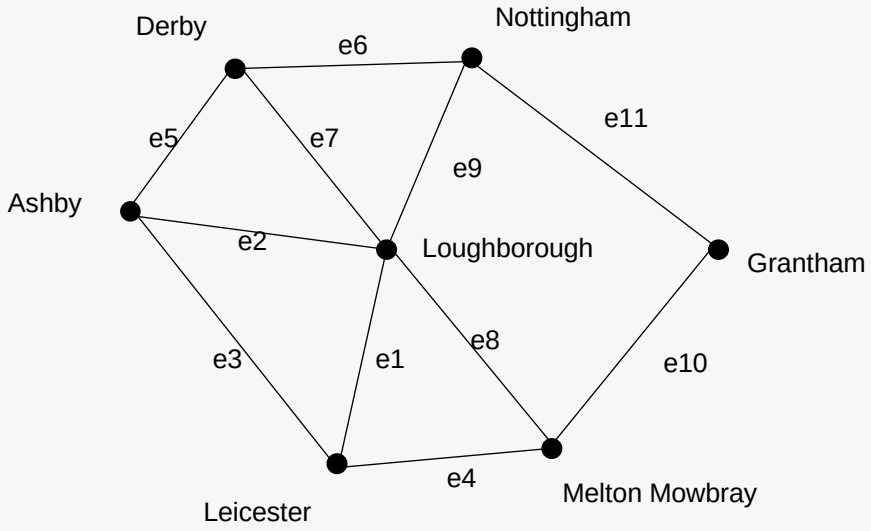




Total cost for this tree will be found by adding up the edges accepted

4+5+8+10 = 27km

Kruskal's Algorithm



edge	weight	
e1	11	**accept**
e2	12	**accept**
e3	15	**reject**
e4	15	**accept**
e5	16	**accept**
e6	16	**accept**
e7	16	**reject**
e8	17	**reject**
e9	18	**reject**
e10	20	**accept**
e11	24	**reject**

The minimum length of cable is 11+12+15+16+16+20=90

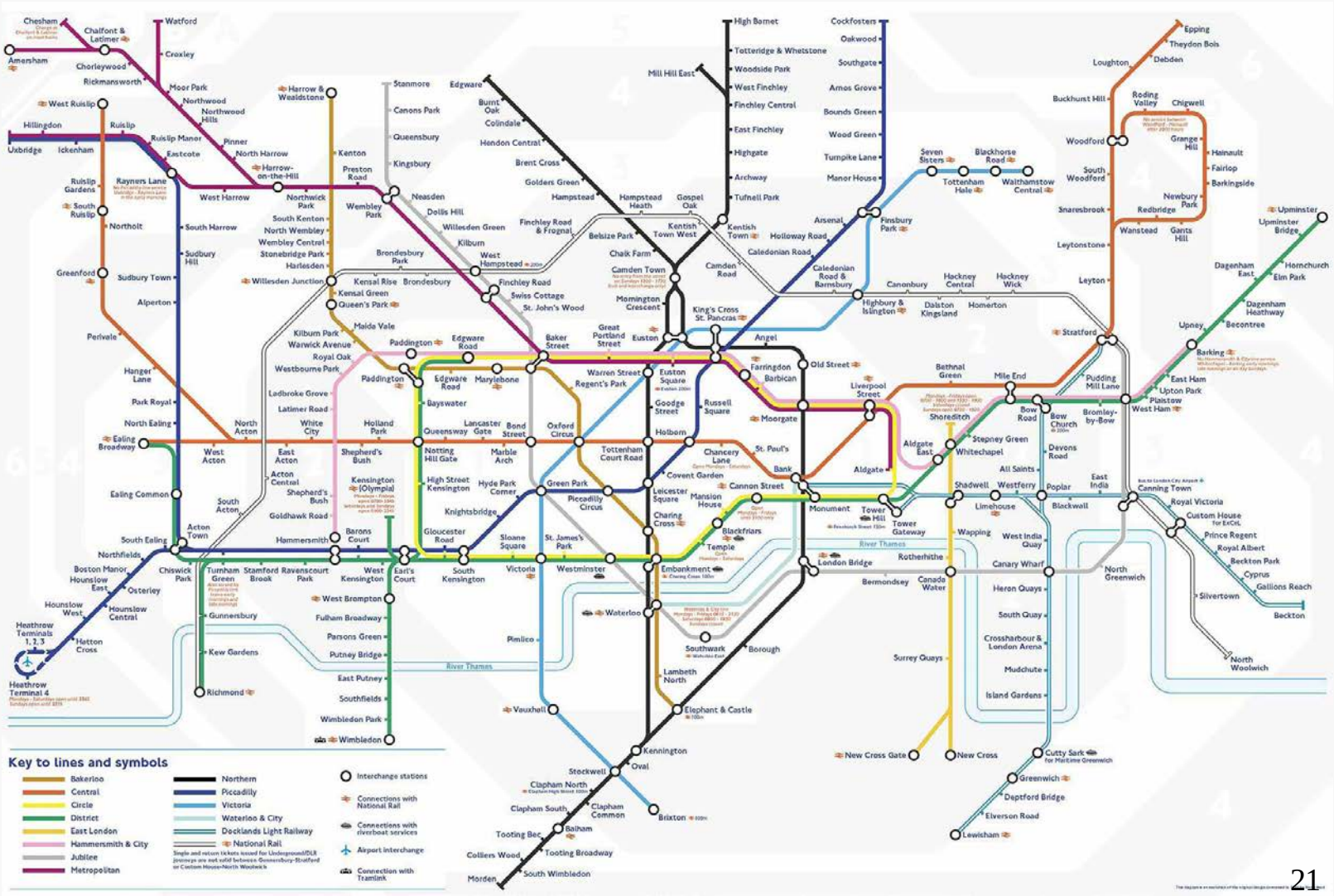
Points to note about Kruskal's alg

- You must arrange the edges in *increasing* size, check you haven't left any out.
- You must add the edges to the tree in the right order, it may not connect at first
- If two edges are the same it does not matter which order you arrange them
- Not all answers will be the same but all minimal spanning trees will give the same totals
- For a graph with n vertices the final tree should have n-1 edges

Finding the shortest Path

- Shortest path = a path of the minimum weight
- Many problems can be modeled using graphs with weights assigned to their edges:
  - Airline flight times (London to Perth via?)
  - Telephone communication costs (laying cables)
  - static/dynamic network routing and response times (fastest connection based on line speed)
  - robot motion planning (getting round obstacles)
  - map/route generation in traffic (Sat Nav)

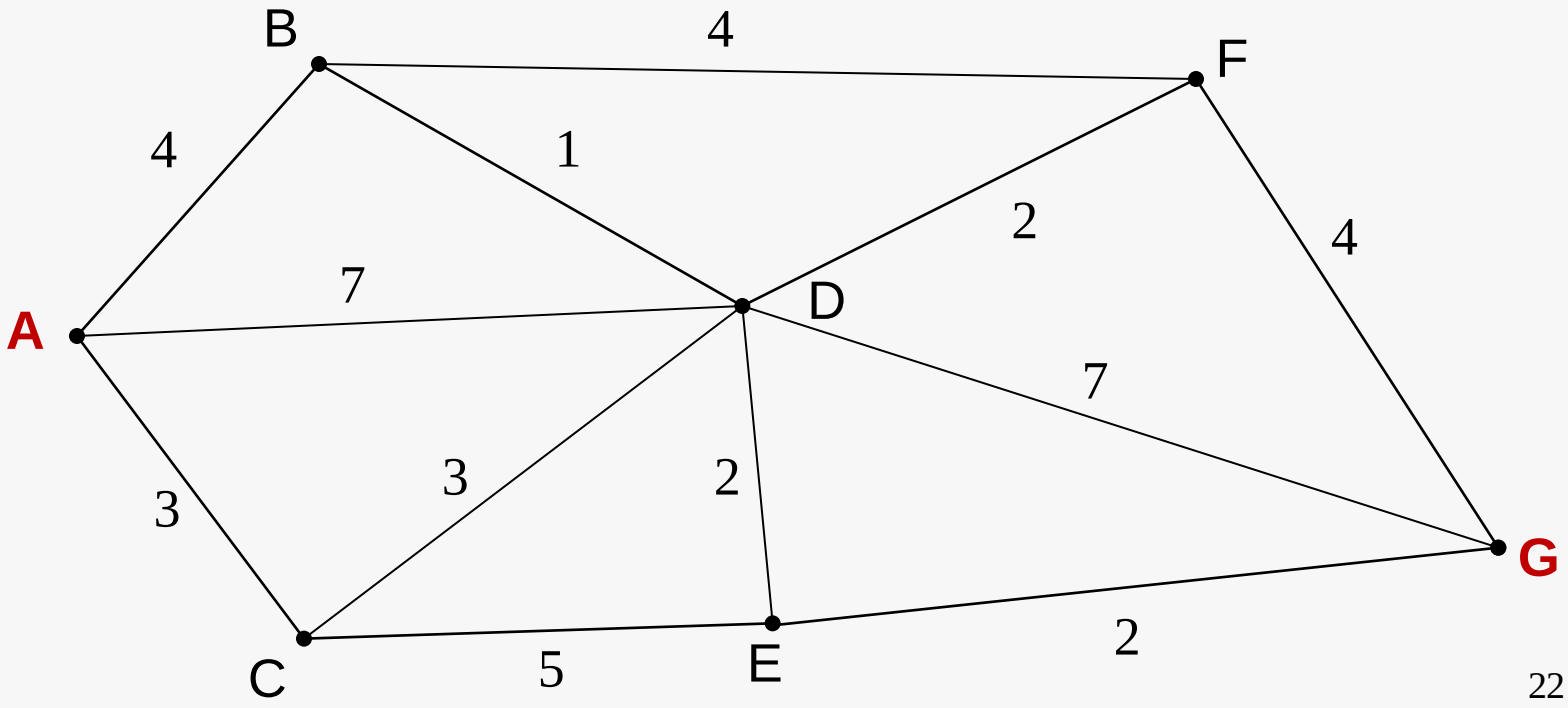
London underground map



Dijkstra's algorithm

Problem:

Find the shortest path from the start vertex (A) to another vertex in the network (G).



Dijkstra's algorithm

1. Label the start vertex with ***permanent cost*** 0 and order label 1

2. Assign ***temporary costs*** to all the vertices that can be reached directly from the start

3. Select the vertex with the ***smallest temporary cost*** and make its label **permanent**. Add the correct order cost.

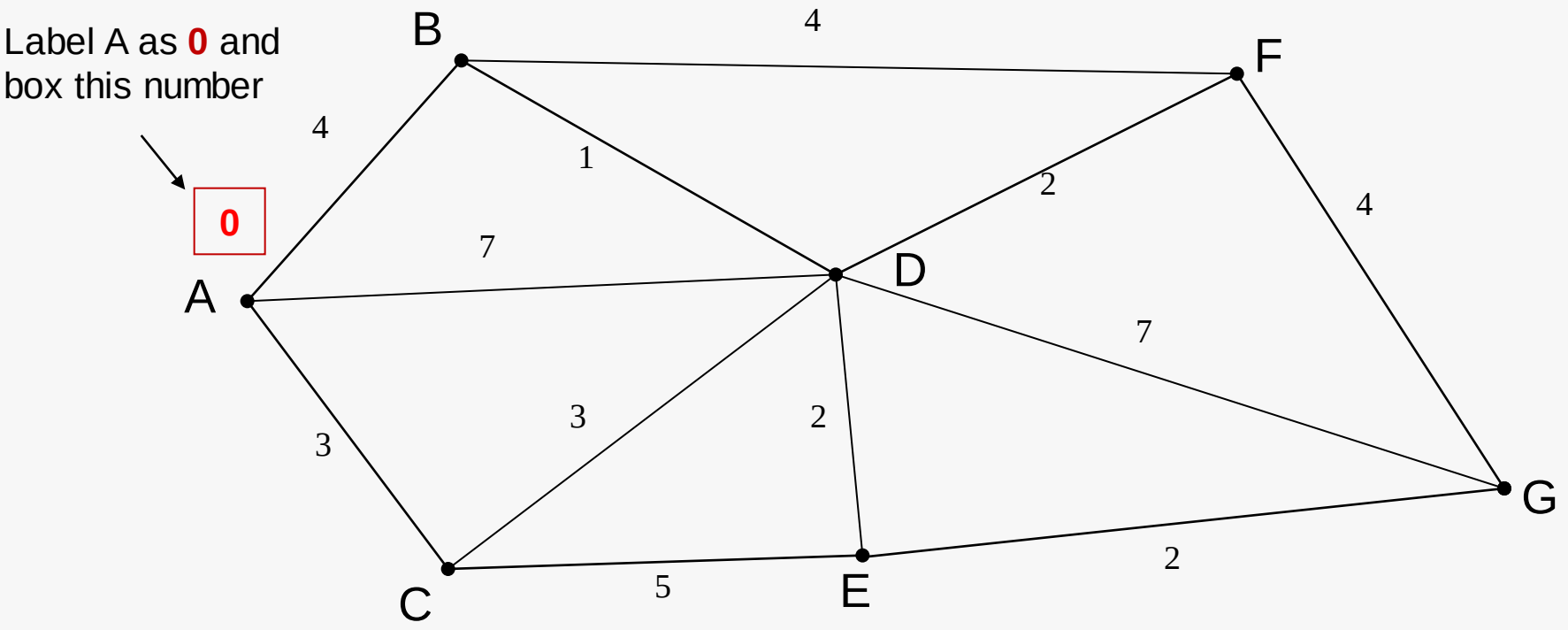
4. Put temporary costs on each vertex that can be reached directly from the vertex you have just made permanent. The temporary costs must be equal to the sum of the permanent cost added to the direct distance from it. If there is an existing temporary cost at a vertex, it should be replaced only if the new cost is smaller.

5. Select the vertex with the smallest temporary costs and make its cost permanent by boxing it.

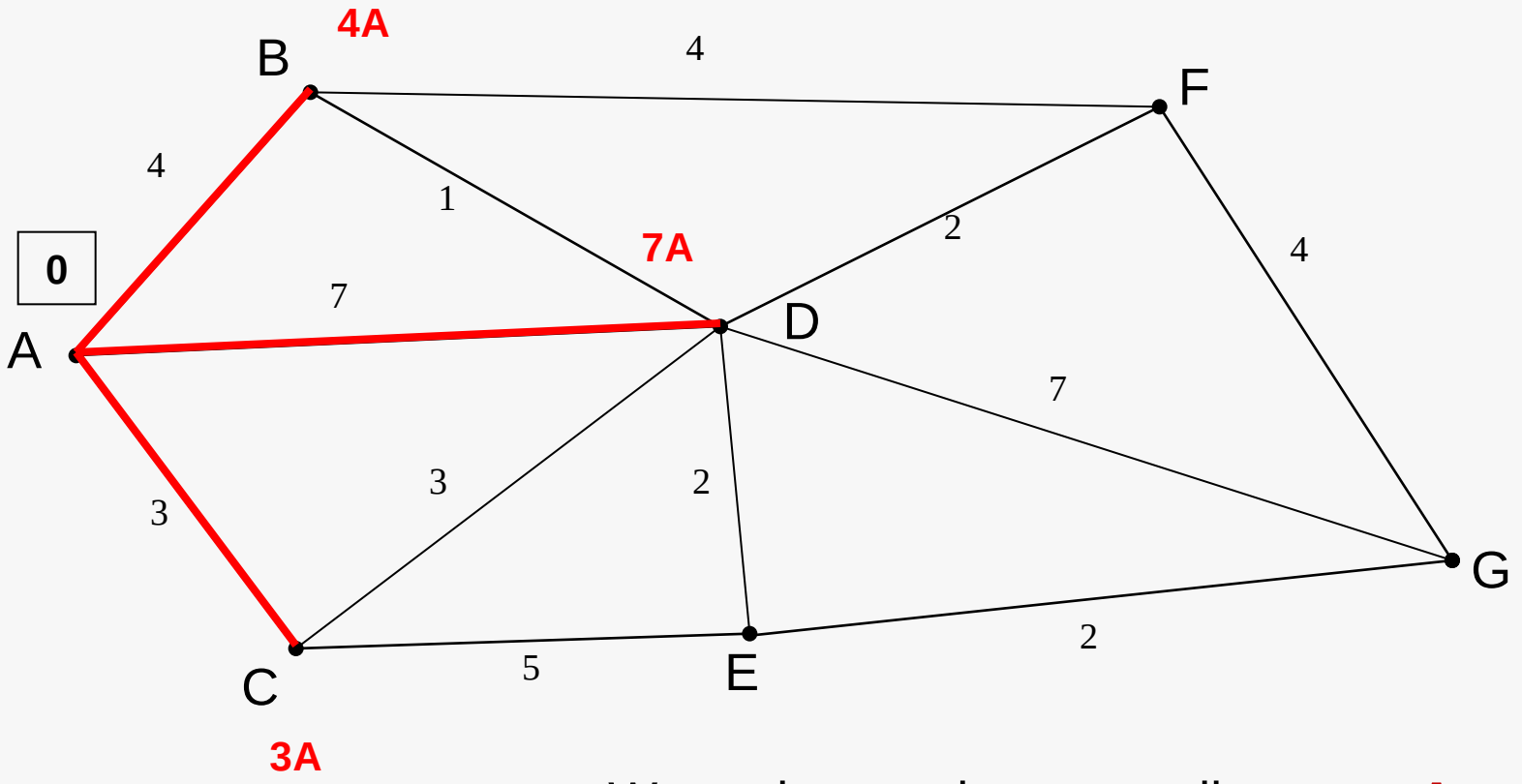
6. Repeat until the finishing vertex has a permanent (boxed) cost.

7. To find the shortest path(s), trace back from the end vertex to the start vertex. Write the route forwards and state the length.

Dijkstra's algorithm

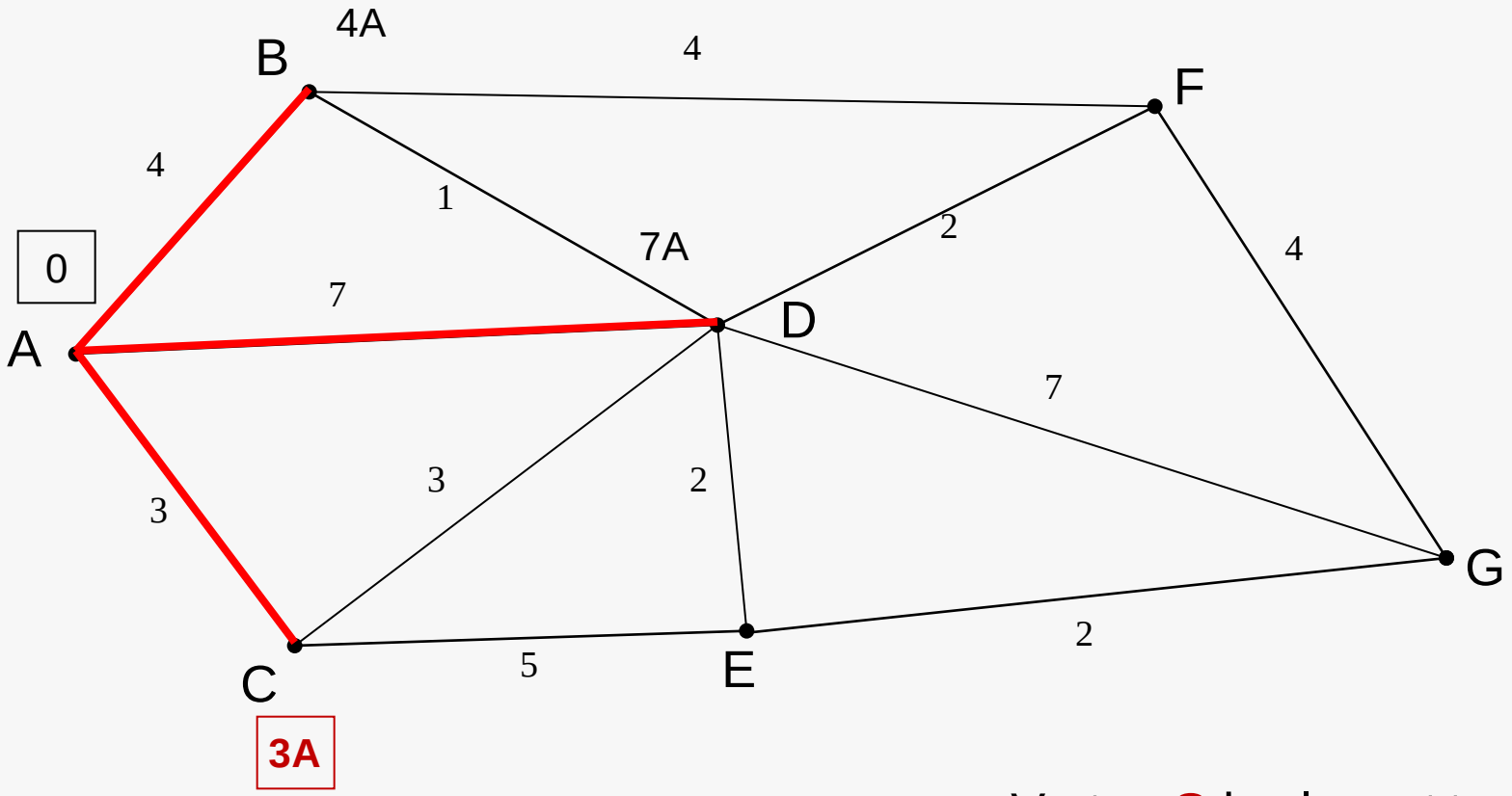


Dijkstra's algorithm



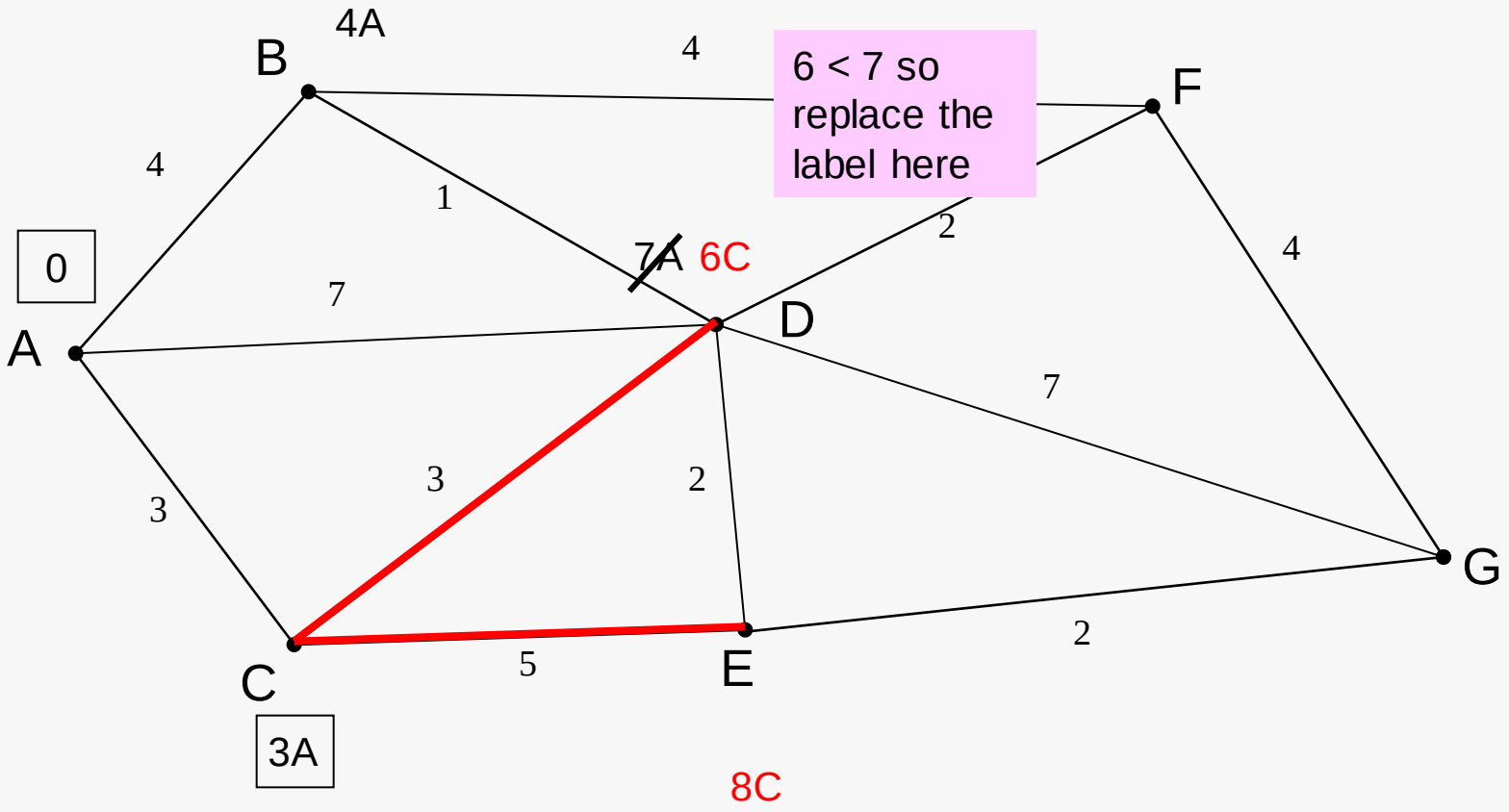
We update each vertex adjacent to **A** with a '*working value' for its distance from A* and indicate the route is from A

Dijkstra's algorithm



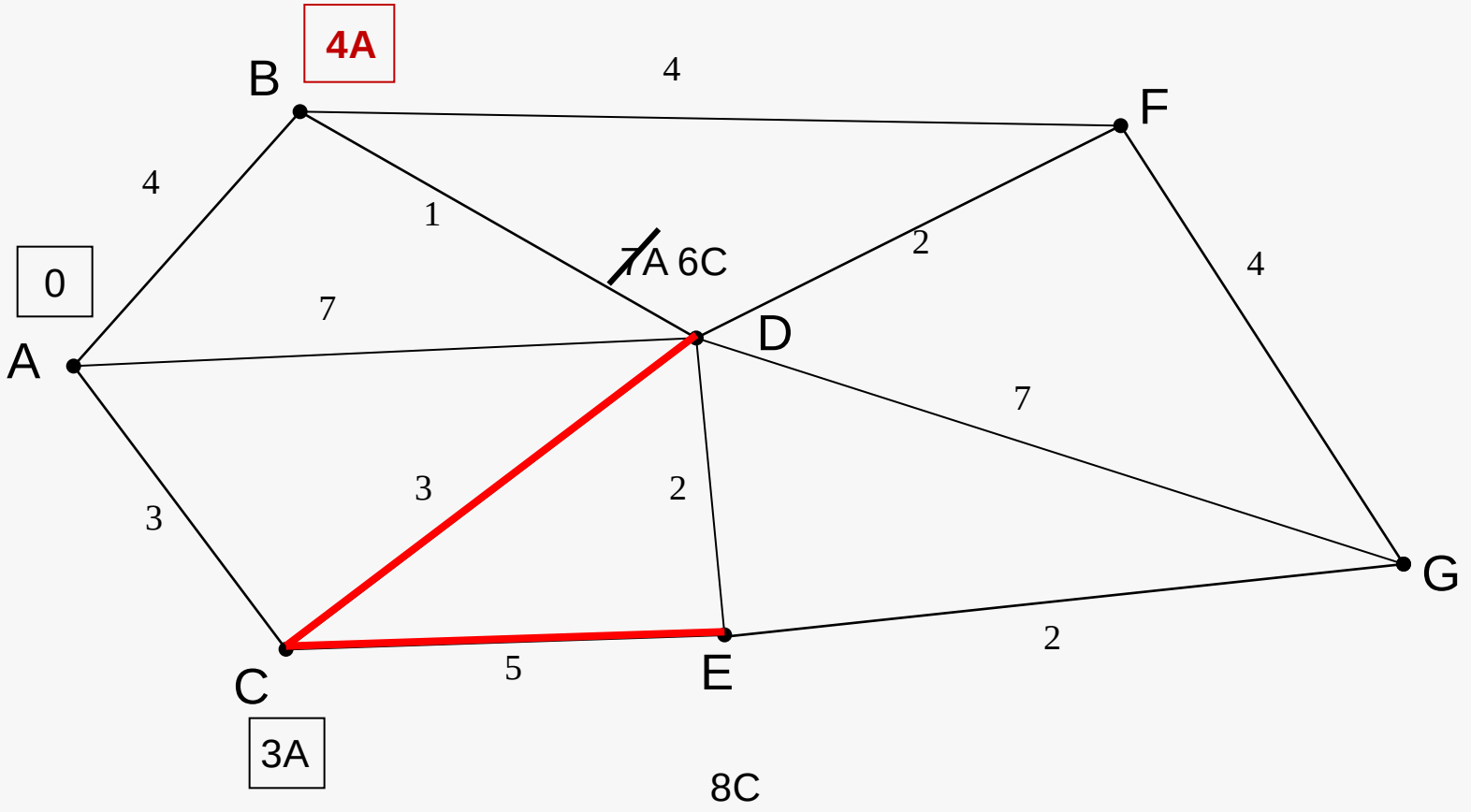
Vertex **C** is closest to A so we give it a box.

Dijkstra's algorithm



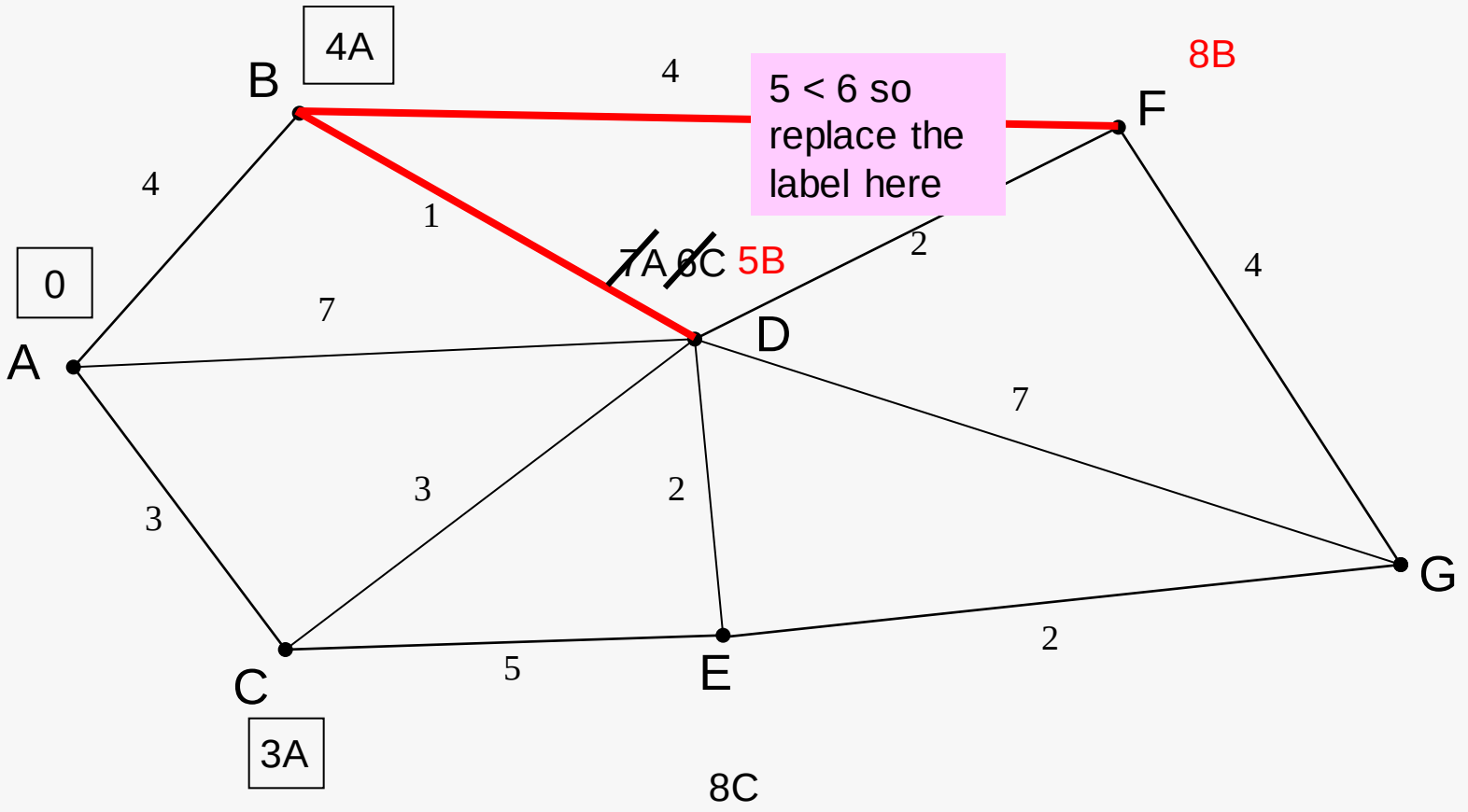
We update each vertex adjacent to **C** with a '*working value' for its total distance from A*, by adding its distance from C to C's permanent label of 3.

Dijkstra's algorithm



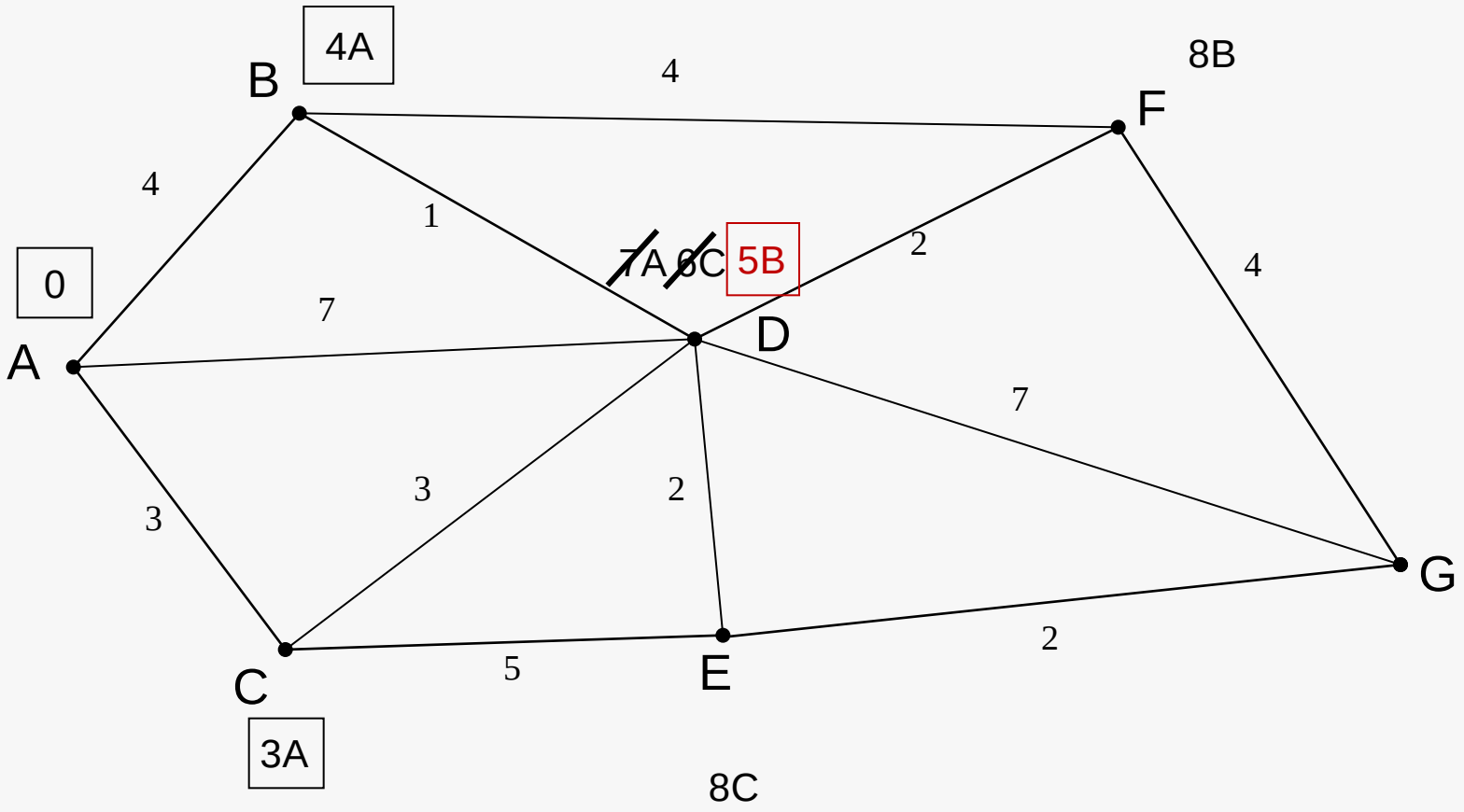
The vertex with the smallest temporary label is **B**, so box this.

Dijkstra's algorithm



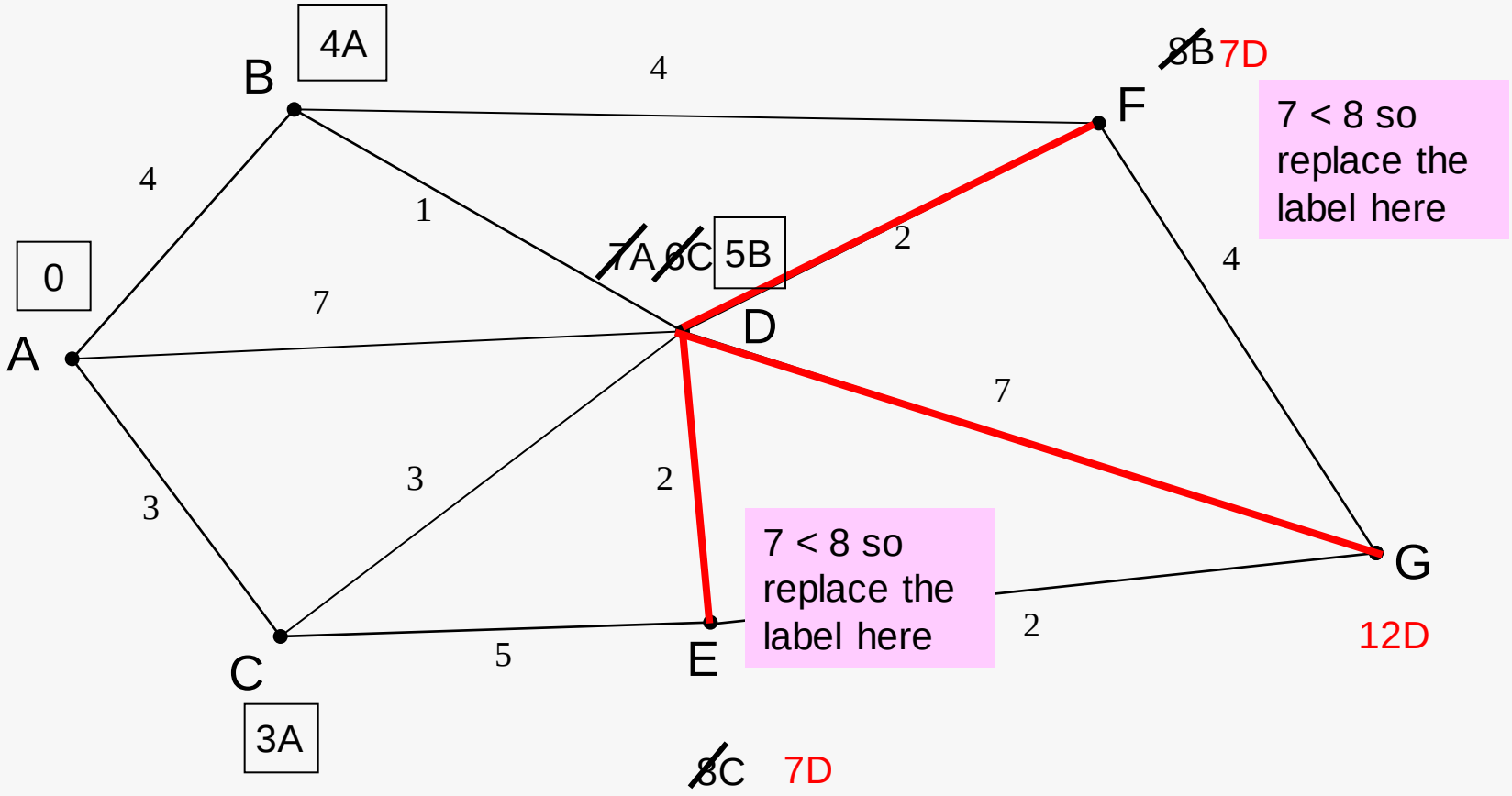
We update each vertex adjacent to **B** with a '*working value' for its total distance from A*, by adding its distance from B to B's permanent label of 4.

Dijkstra's algorithm



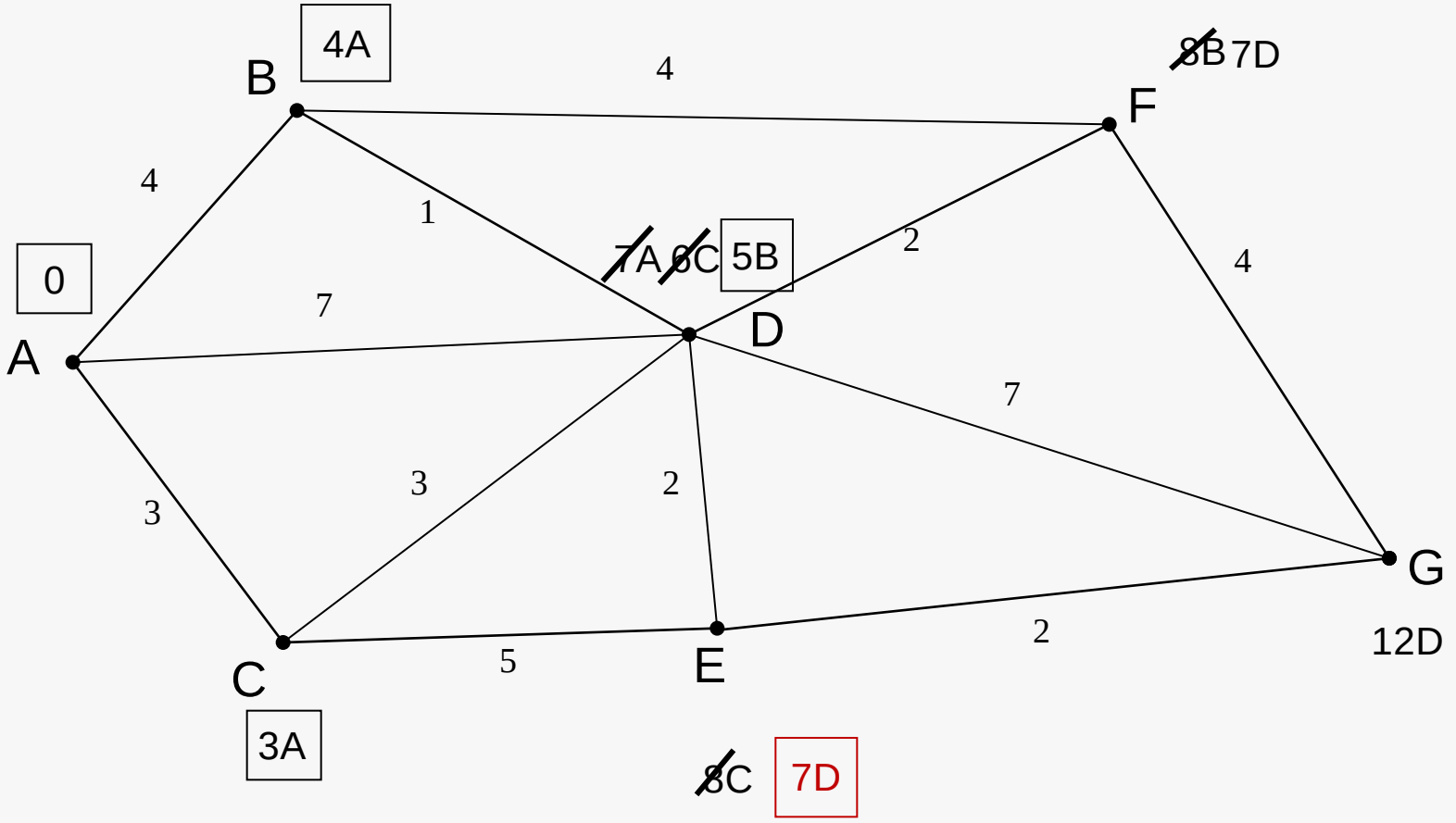
The vertex with the smallest temporary label is **D**, so box this.

Dijkstra's algorithm



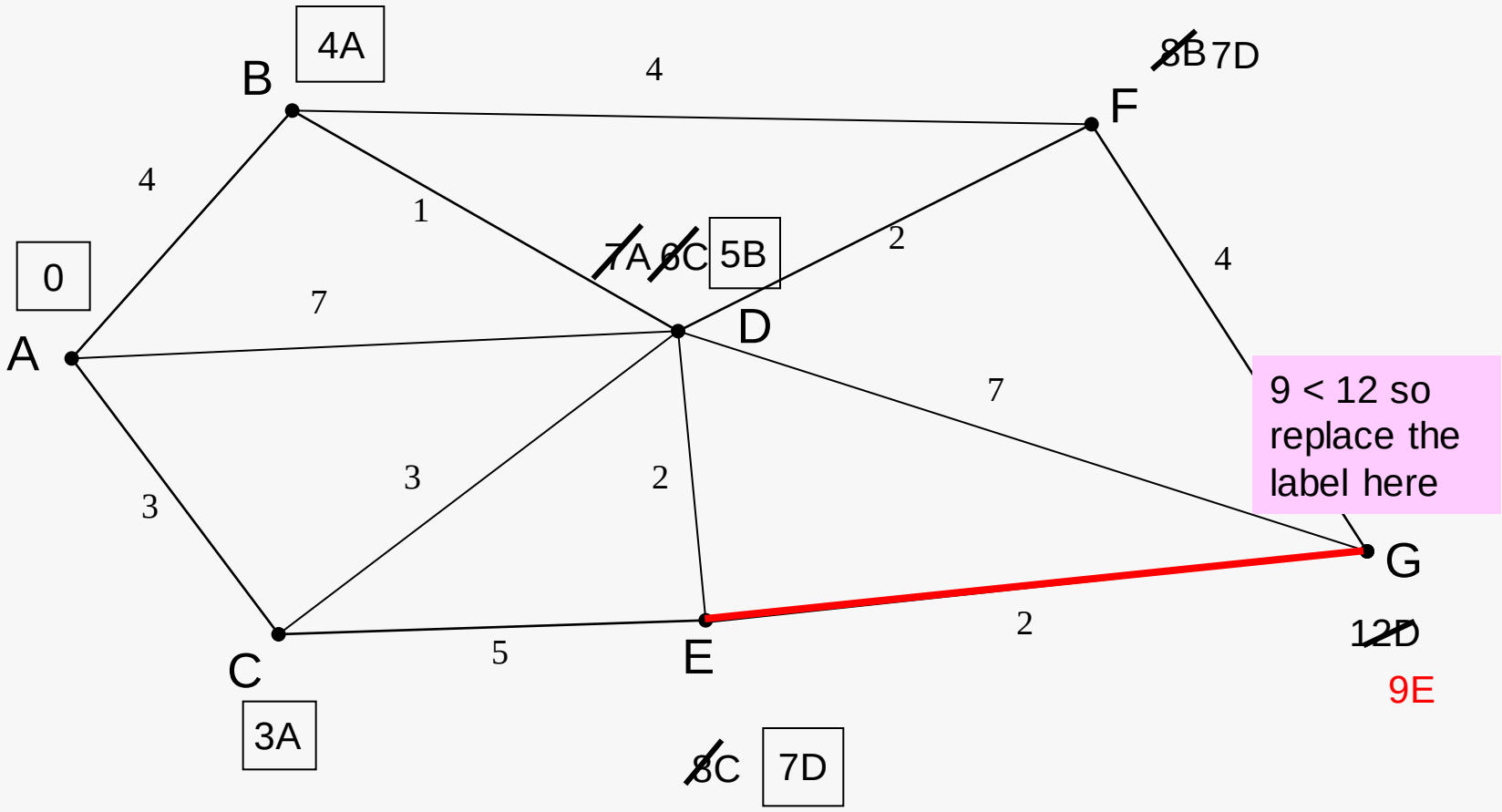
We update each vertex adjacent to **D** with a *'working value' for its total distance from A*, by adding its distance from D to D's permanent label of 5.

Dijkstra's algorithm



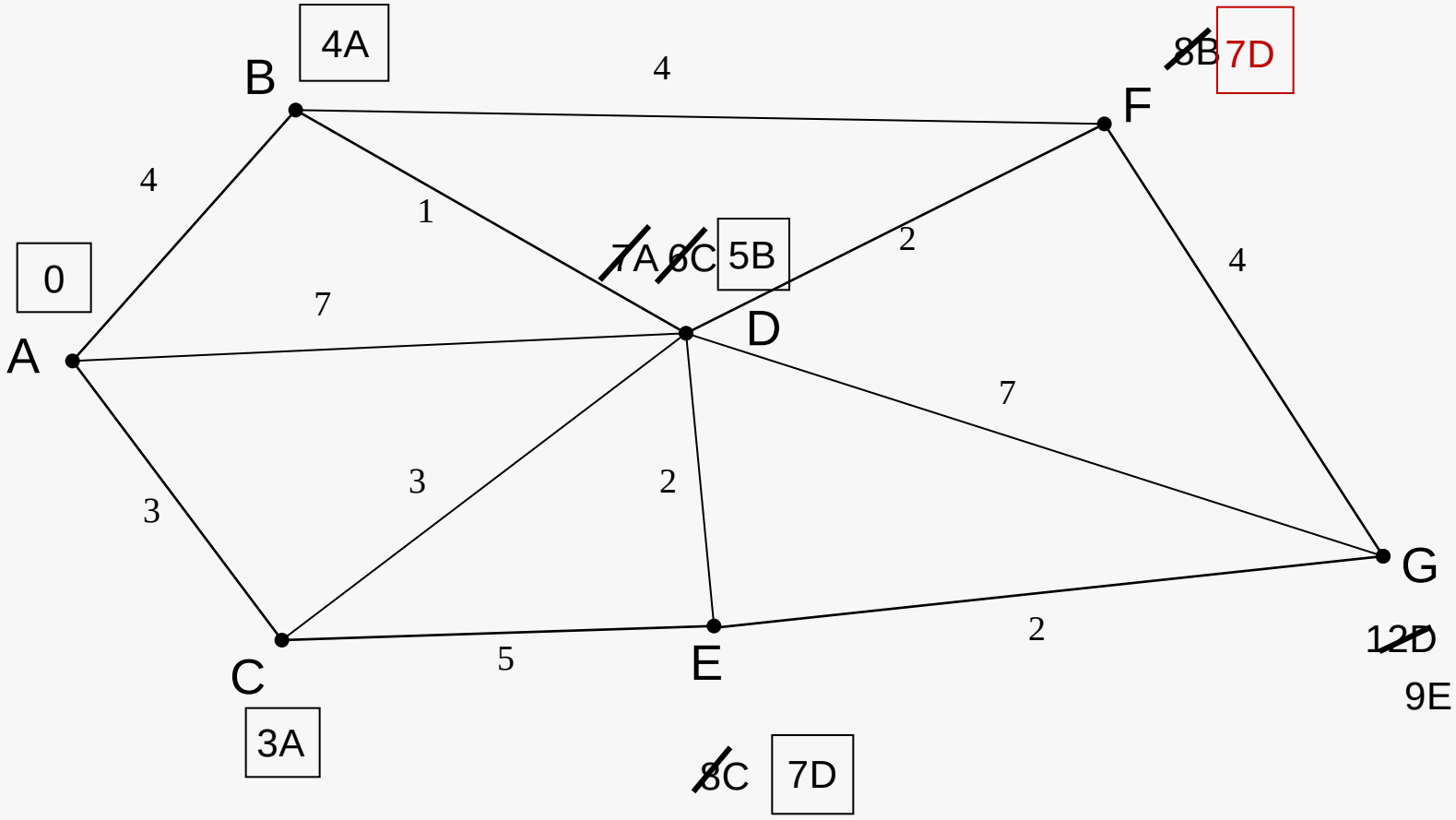
The vertices with the smallest temporary labels are **E** and **F**, so choose one and box it.

Dijkstra's algorithm



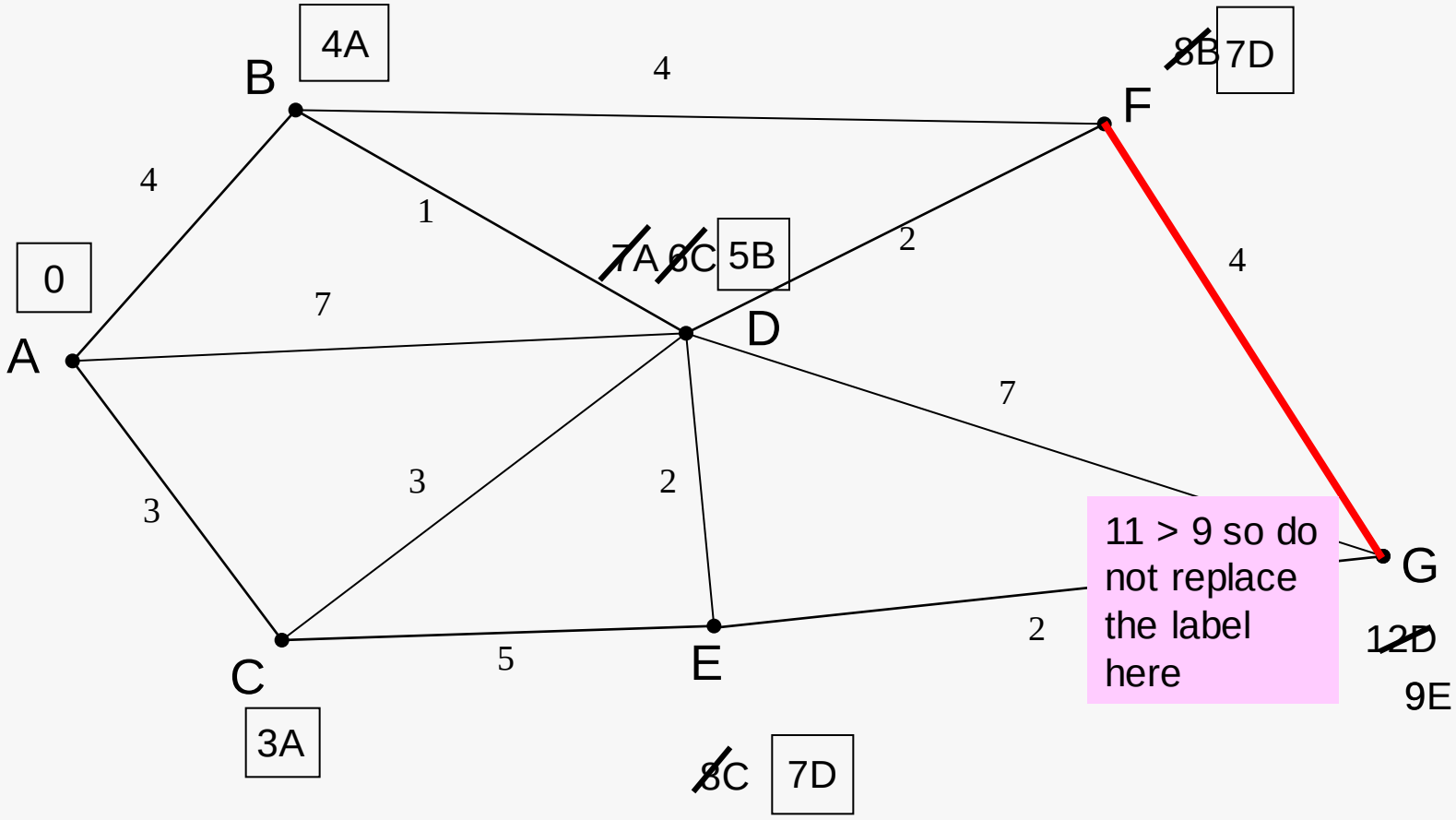
We update each vertex adjacent to **E** with a *'working value' for its total distance from A*, by adding its distance from E to E's permanent label of 7.

Dijkstra's algorithm



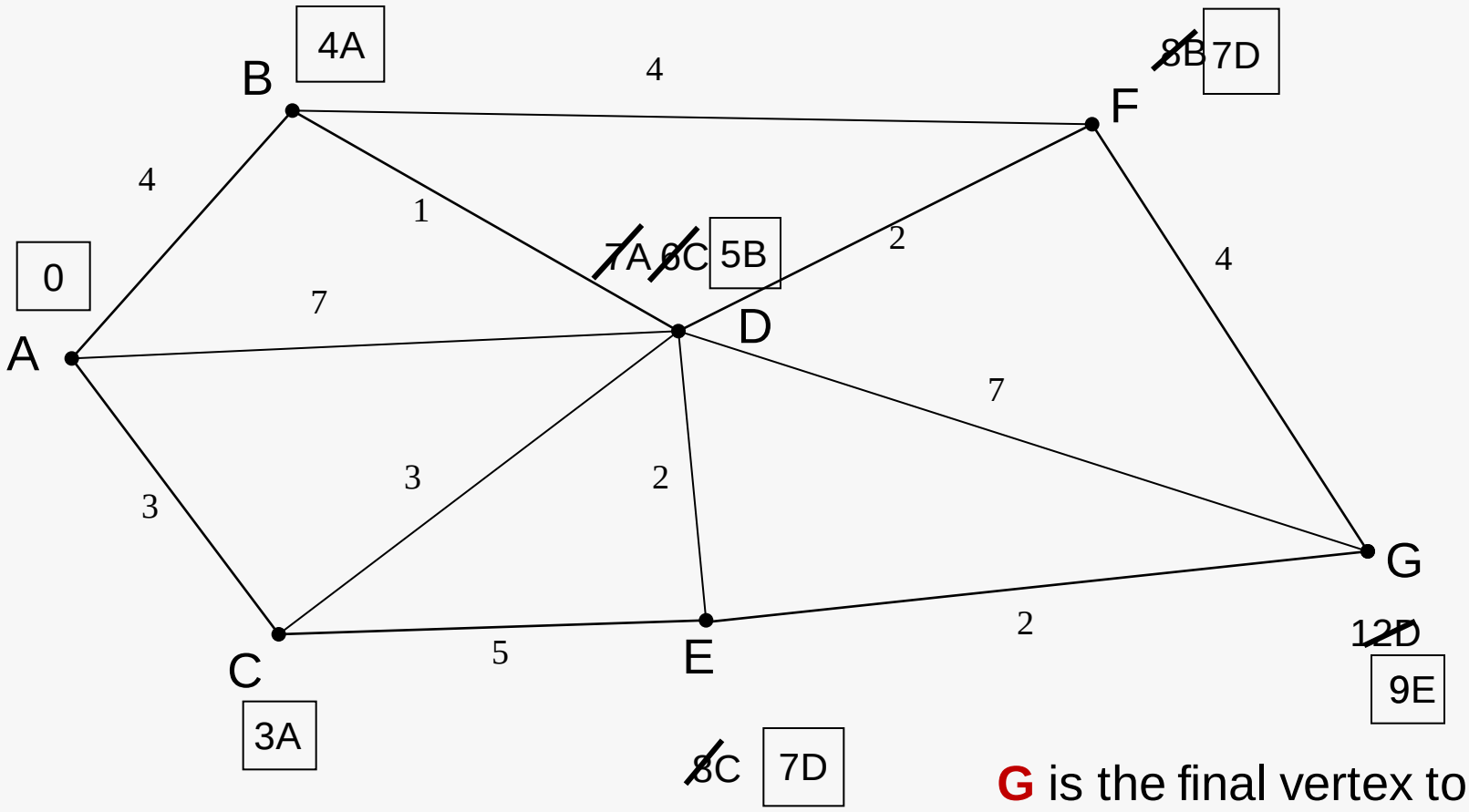
The vertex with the smallest temporary label is **F**, so make box this.

Dijkstra's algorithm



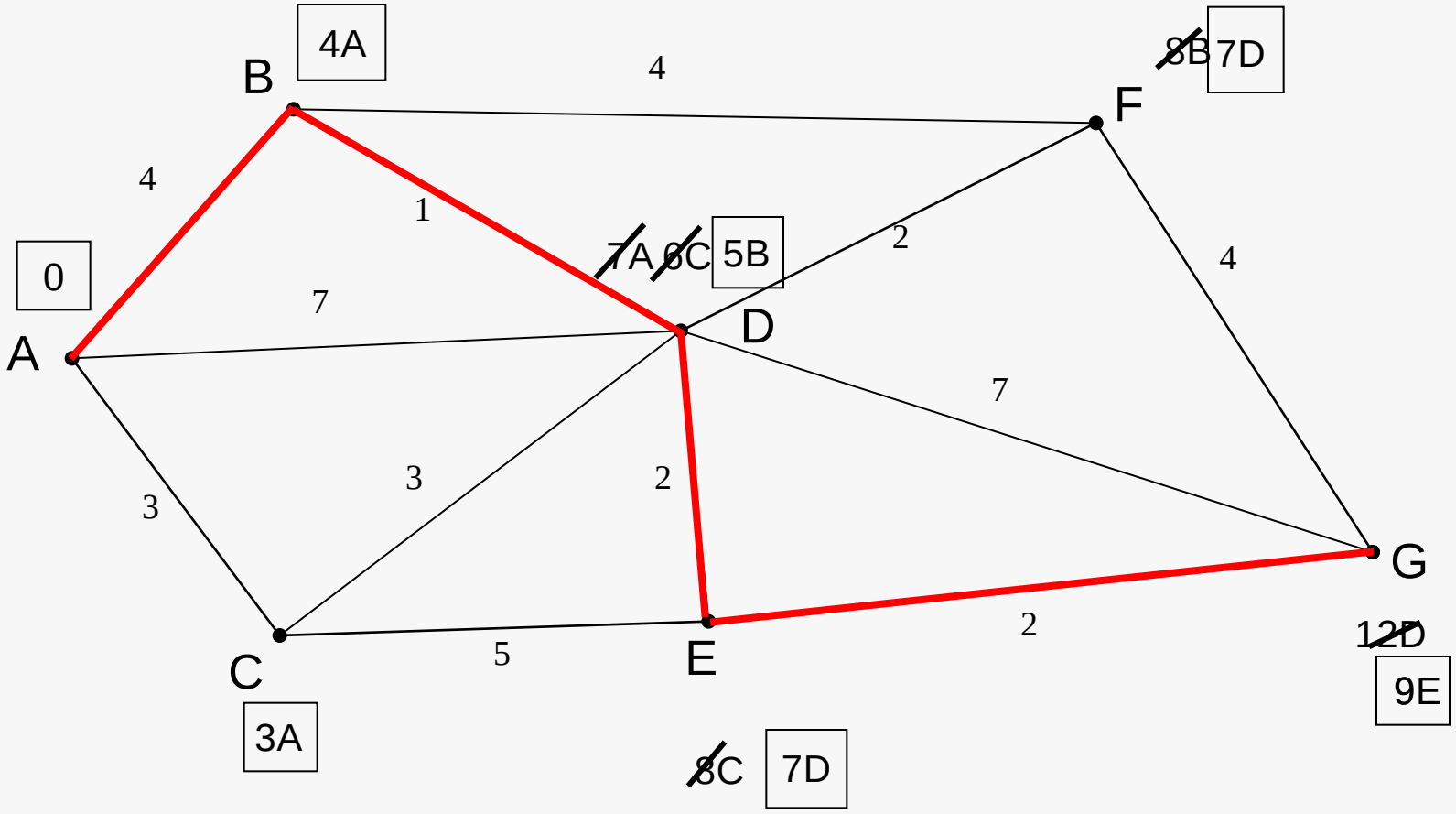
We update each vertex adjacent to **F** with a *'working value' for its total distance from A*, by adding its distance from F to F's permanent label of 7.

Dijkstra's algorithm



G is the final vertex to be boxed.

Dijkstra's algorithm



To find the shortest path from A to G, start from G and work backwards, using the letters in the boxes.

The shortest path is ABDEG, with length 9.

Points to note

- Dijkstra's algorithm can be used to find the shortest path from any start vertex to any other vertex in the network
- Dijkstra's algorithm assumes that
  - the graph is connected
  - edges are undirected
  - non-negative weights
- Some good examples based on animation to show how the algorithm works
- There are other search algorithms such as Prims, Bellman-Ford, etc ...go and research these to broaden your knowledge