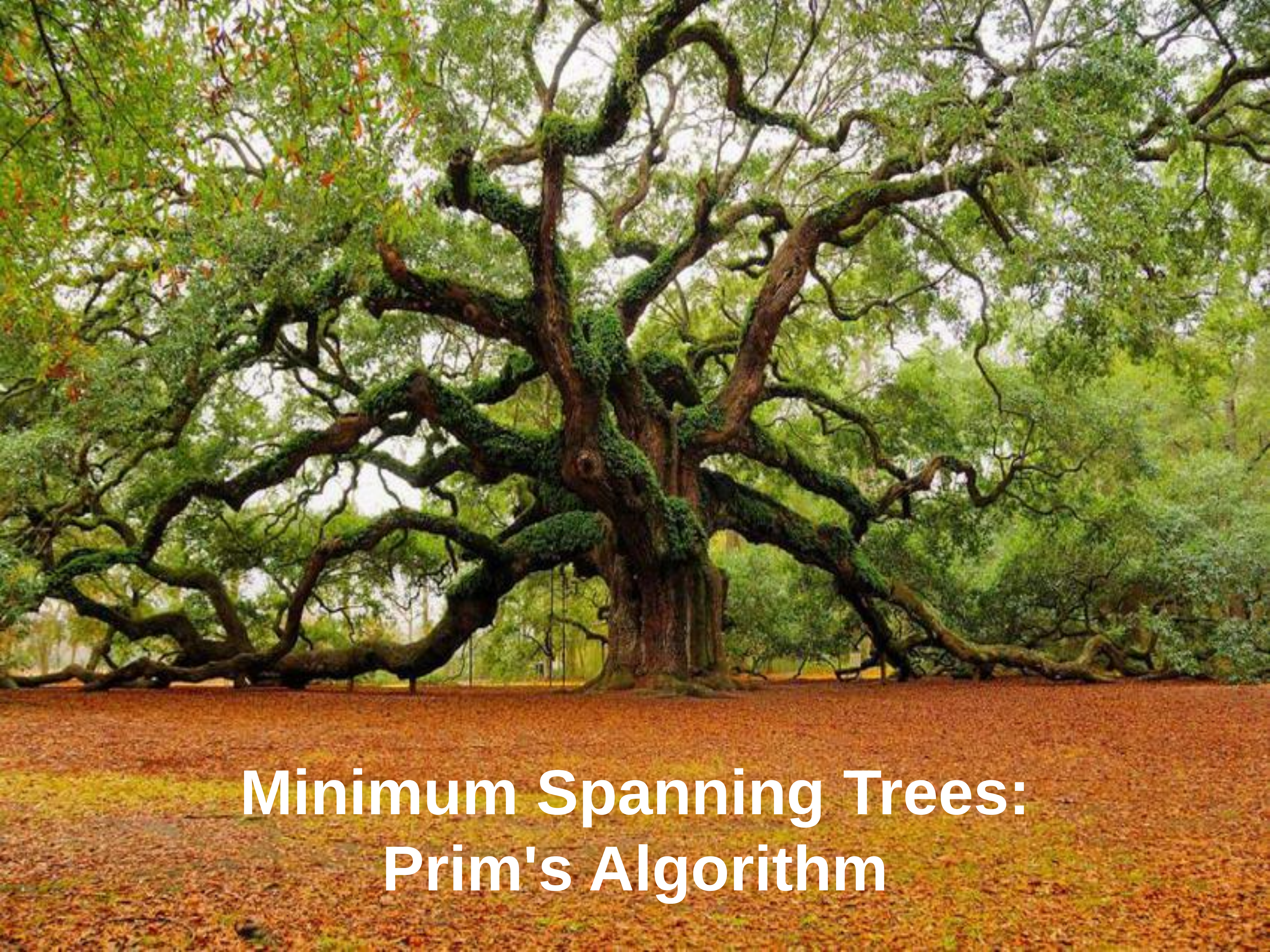
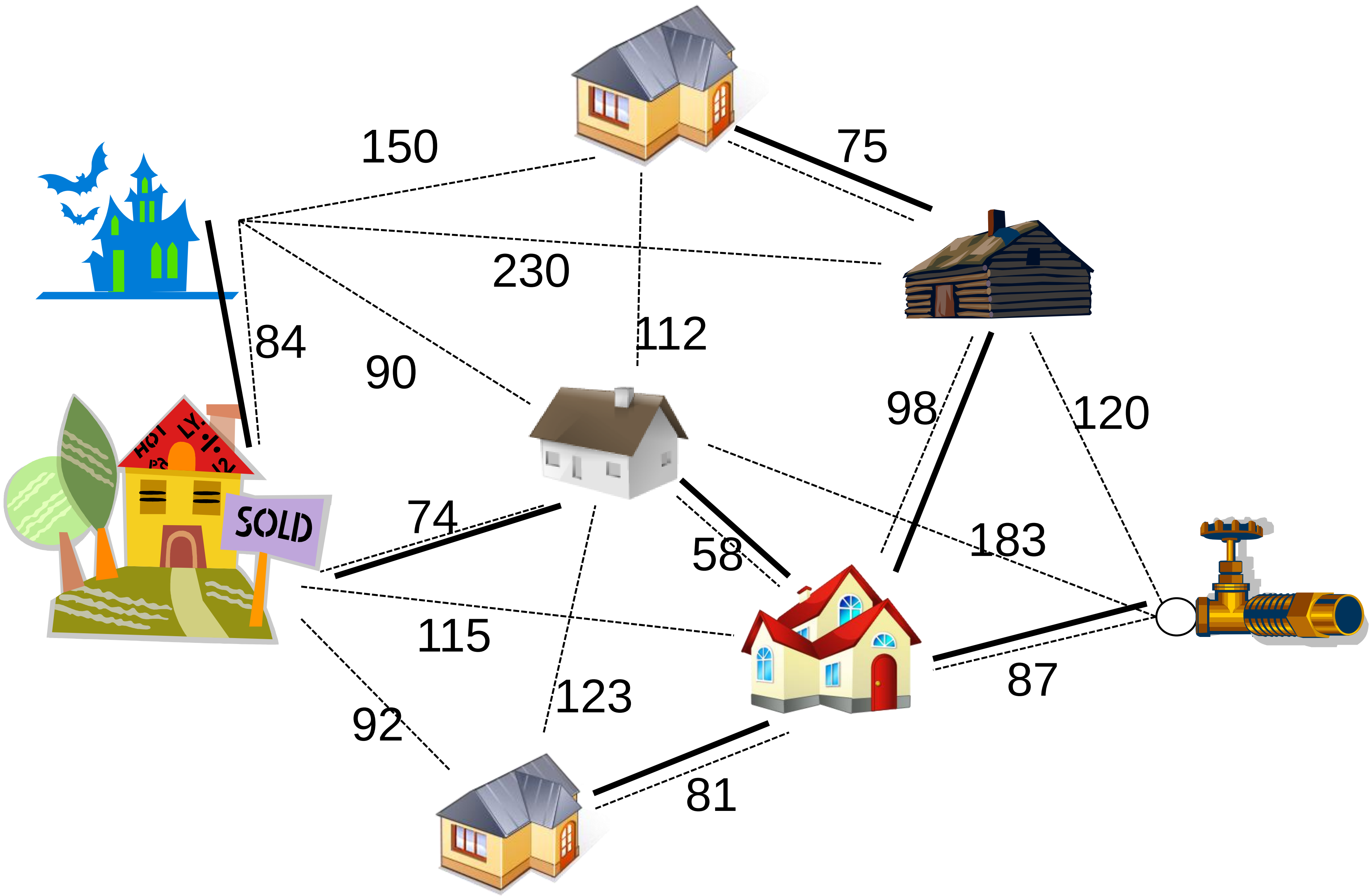


Minimum Spanning Trees: Prim's Algorithm

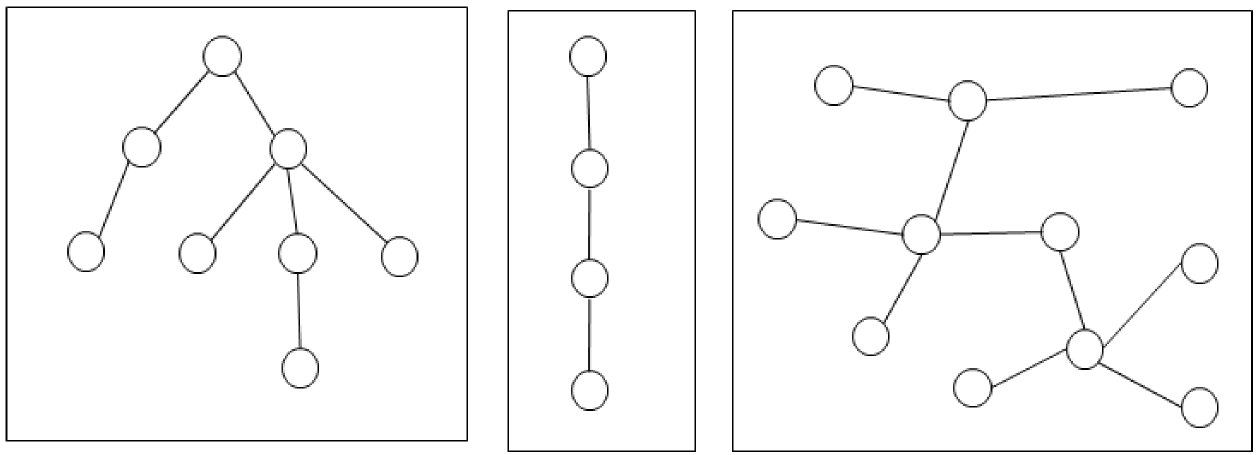

150
75
230
84
90
112
98
120
74
58
183
115
87
123
92
81
SOLD

CS1113

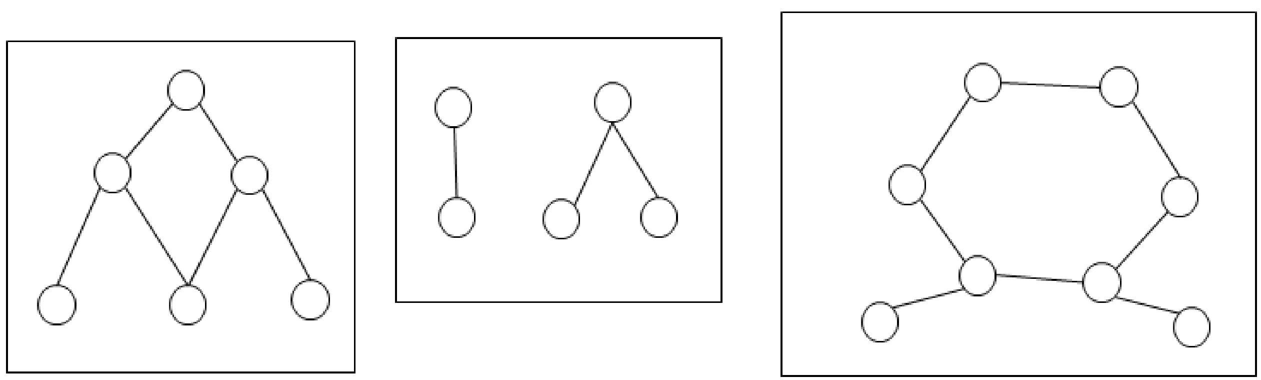
Trees

A tree is a connected undirected simple graph with no cycles

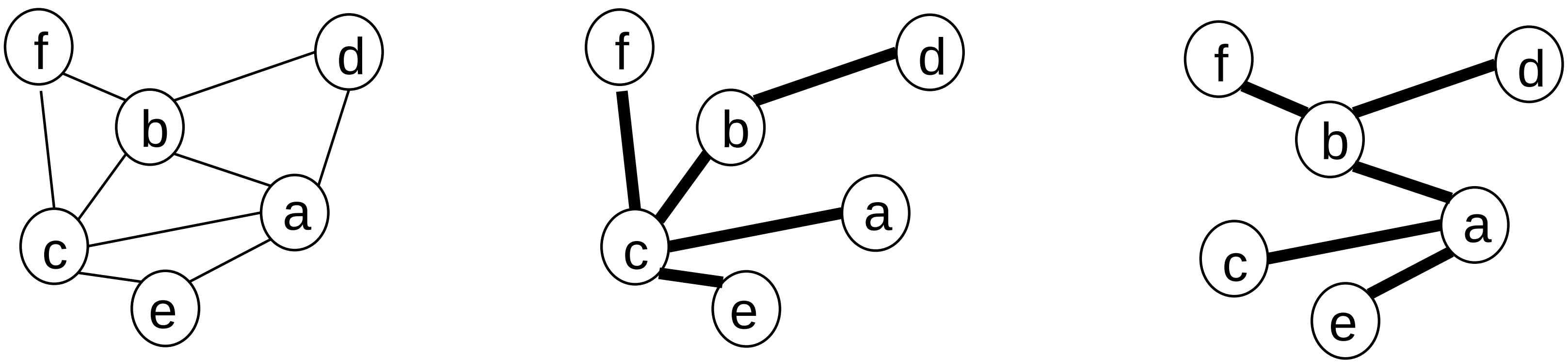
trees:



not trees:



For an undirected connected simple graph G = (V,E), a *spanning tree* is a subgraph of G that is a tree and which contains every vertex in V.

G:


If the graph G has numerical weights on each edge, then a *minimum spanning tree* is a spanning tree which has the lowest sum of weights of the selected edges.

CS1113

Prim's Algorithm

```
Algorithm: Prim's
Input: connected undirected graph G = (V,E) with edge
       weights and n vertices
Output: a spanning tree T=(V,F) for G

1. T := {v}, where v is any vertex in V
2. F := { }
3. for each i from 2 to n
4.     e := {w,y}, an edge with minimum weight in E such
           that w is in T and y is not in T
5.     F := F ∪ {e}
6.     T := T ∪ {y}
7. return (T,F)
```

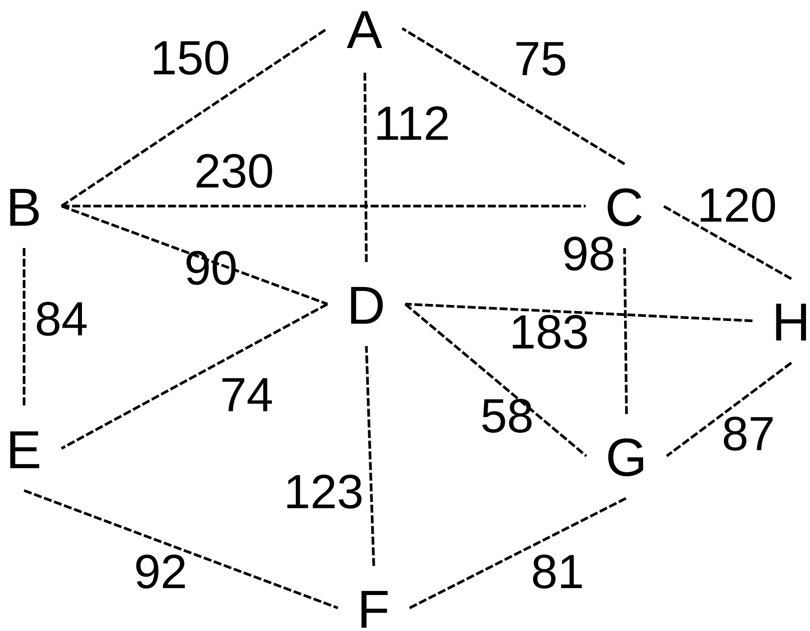
Algorithm: Prim's

```
Input: connected undirected graph G = (V,E) with edge
       weights and n vertices
Output: a spanning tree T=(V,F) for G

1. T := {v}, where v is any vertex in V
2. F := { }
3. for each i from 2 to n
4.    e := {w,y}, an edge with minimum weight in E such
          that w is in T and y is not in T
5.    F := F ∪ {e}
6.    T := T ∪ {y}
7. return (T,F)
```

```
prim(): #pseudocode

add each vertex in G into a free dictionary
create an empty dictionary locs for locations of vertices in APQ
create an APQ pq, which contains costs and (vertex,edge) pairs
for each v in G
    add (∞, (v,None)) into pq and store location in locs[v]
create an empty list, which will be the output (the edges in the tree)
while pq is not empty
    remove c:(v,e), the minimum element, from pq
    remove v from free
    remove v  from locs
    append e to the list
    for each edge d incident on v
        w = d's opposite vertex
        cost = d's cost
        if w is in free, and cost is cheaper than w's entry in pq
            replace ?:(w,?) in pq with cost: (w, d)
 return the list
```


A
B
C
D
E
F
G
H
150
75
112
230
120
98
90
84
183
74
58
87
123
92
81

Complexity:

creating the dictionaries: O(n)
creating the APQ: O(n log n)
n times round the loop
  for each time, O(1) operations to remove and add to structures, so O(n)
  compare up to d edges, where d is the maximum degree of any vertex and for each one maybe update apq. Each update is O(log n). But, over the full algorithm, can only do at most m updates to the apq (since there are only m edges)
So O(m log n)

So in total O((n+m)log n)

As for Dijkstra, if the graph is dense, using an unsorted list APQ would give us $O(n^2)$

Is Prim's algorithm guaranteed to produce a minimum spanning tree?

Let T be the output of Prim's algorithm, with edges that were added in the sequence e_1, e_2, ..., e_{n-1}. We will call T_i the tree with edges e_1 to e_i.
Let S be any minimum spanning tree. If S = T, we are done.
If not, let e_{k+1} be the first edge added by Prim which is not in S. One vertex of e_{k+1} is in T_k, and the other is not. Add this edge into S.
Since S was a spanning tree, and we have added an edge, it must now contain a circuit, and that circuit contains the edge e_{k+1}.
But that circuit must contain at least one edge not in T_{k+1} (which is a tree).
Starting at e_{k+1}, and moving in the direction into $\{e_1, ..., e_k\}$, move round the circuit until you find an edge x which is not in T_{k+1}, and which has at least one endpoint in T_k. Prim chose e_{k+1} before x, and so $w(e_{k+1}) \leq w(x)$.
Delete x from $S \cup \{e_{k+1}\}$. This must give a tree T', it contains all the edges from T_{k+1}, and its cost is no higher than S, so it is an MST.
Repeat this argument, but the first edge not in T' will now be later than e_{k+1}. Keep repeating the process until we demonstrate that the last edge added by Prim must also have been correct.

```
def mst(self):
    state = {v:False for v in self._structure}
    locs = {}
    apq = APQHeap()
    for v in self._structure:
        locs[v] = apq.add(float('inf'), (v,None))

    tree = []

    while not apq.is_empty():
        key, (v,e) = apq.remove_min()
        del state[v]
        tree.append(e)
        del locs[v]
        v_edges = self.get_edges(v)
        for e in v_edges:
            w = e.opposite(v)
            cost = e.element()
            if w in state and cost < apq.get_key(locs[w]):
                apq.update_key(locs[w],cost)
                apq.update_value(locs[w], (w,e))
    return tree
```

Next lecture

Further graph algorithms