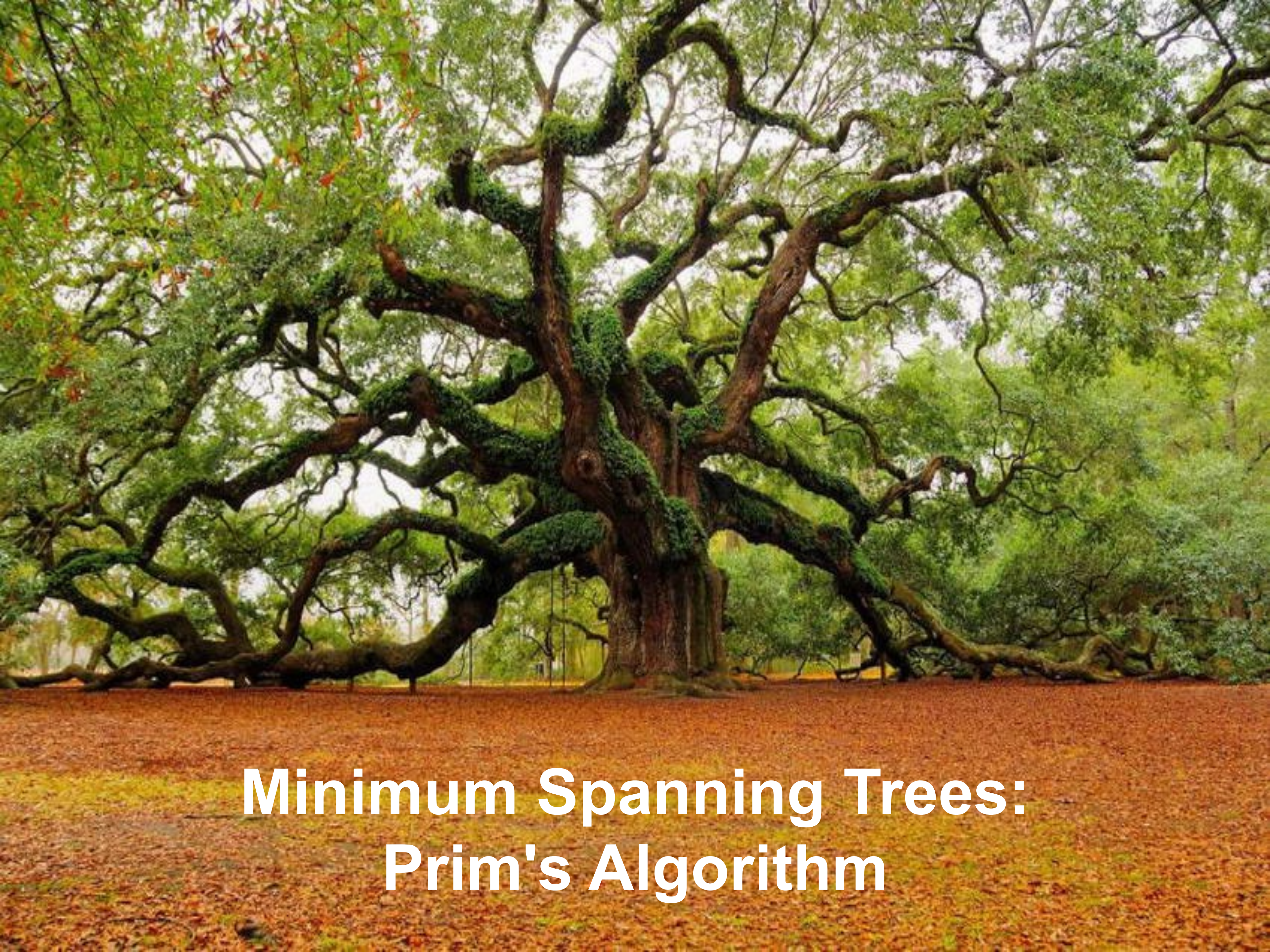
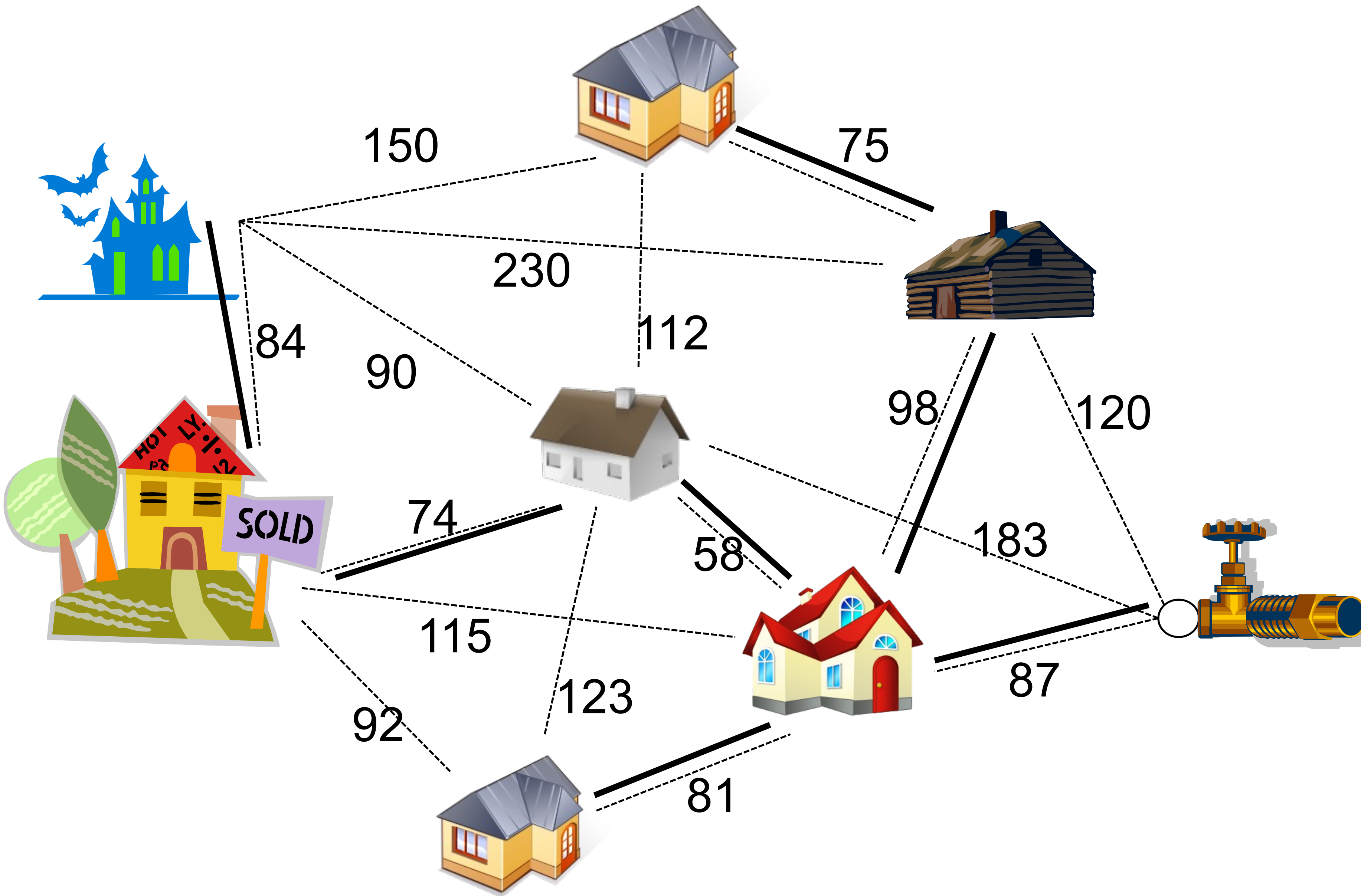


Minimum Spanning Trees: Prim's Algorithm

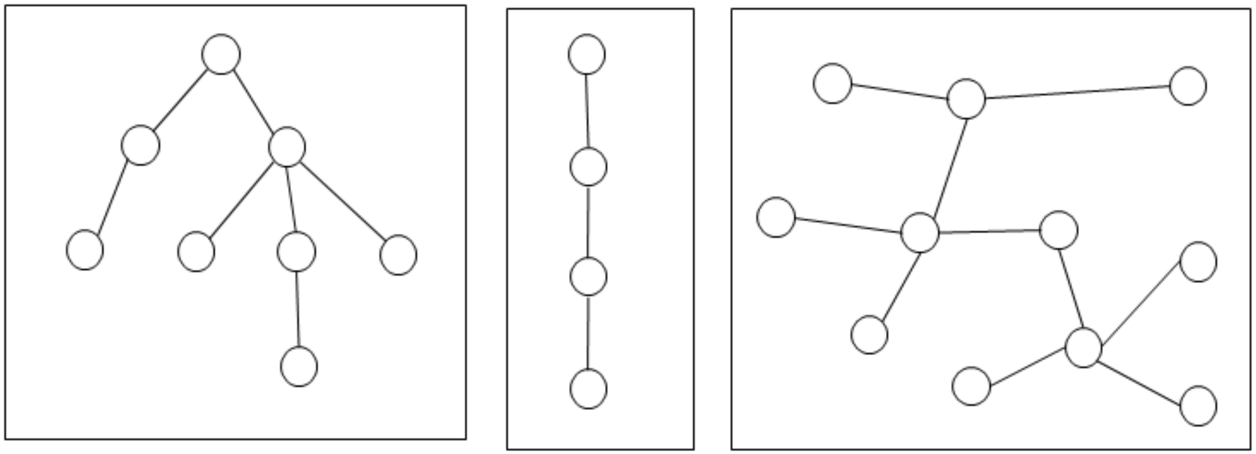

150
75
230
84
90
112
98
120
74
58
183
115
87
123
92
81
SOLD

CS1113

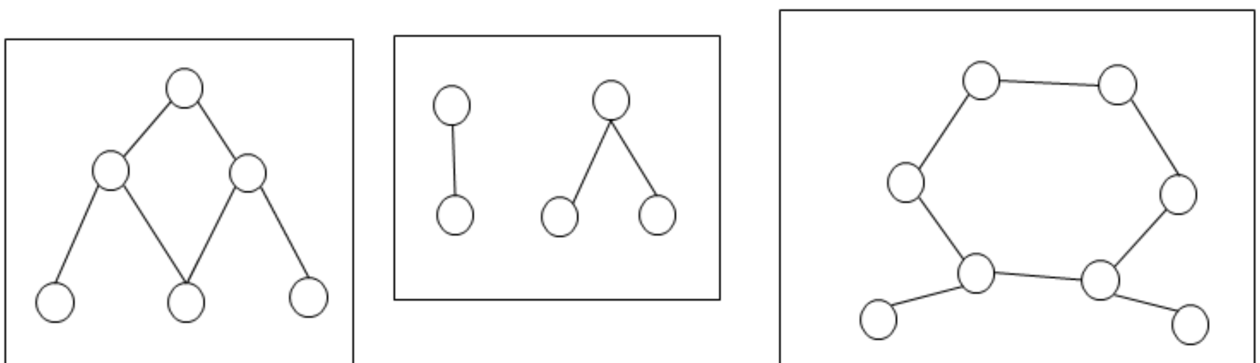
Trees

A tree is a connected undirected simple graph with no cycles

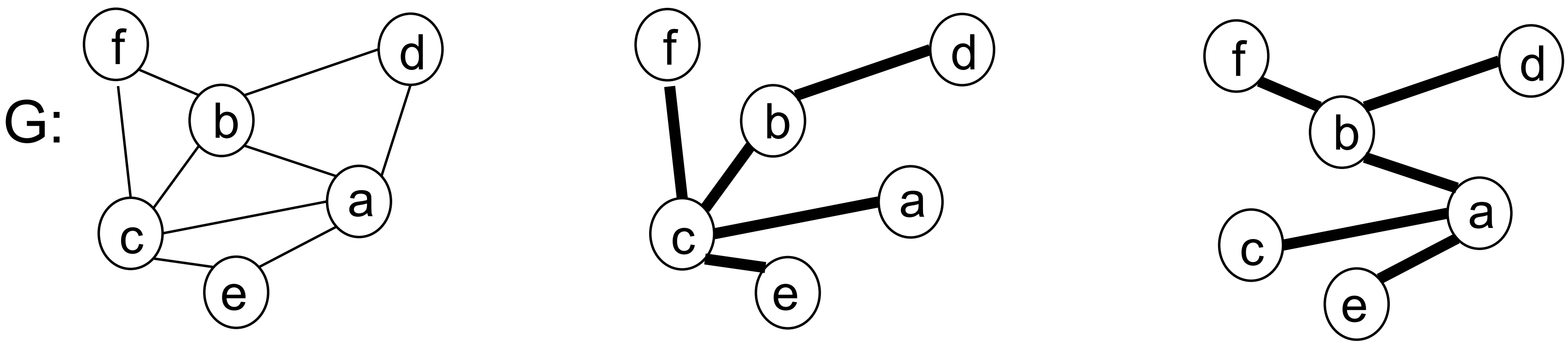
trees:



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.



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

CS1113

```
Algorithm: Prim's
Input: connected undirected graph G = (V,E) with edge
       weights and n vertices
Output: the edges S of a spanning tree (V,S) for G

1. T := [v], where v is any vertex in V
2. S := []
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.     add e into S
6.     add y into T
7. return S
```

```
Algorithm: Prim's
Input: connected undirected graph G = (V,E) with edge
       weights and n vertices
Output: the edges S of a spanning tree (V,S) for G

1. T := [v], where v is any vertex in V
2. S := []
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.    add e into S
6.    add y into T
7. return S
```

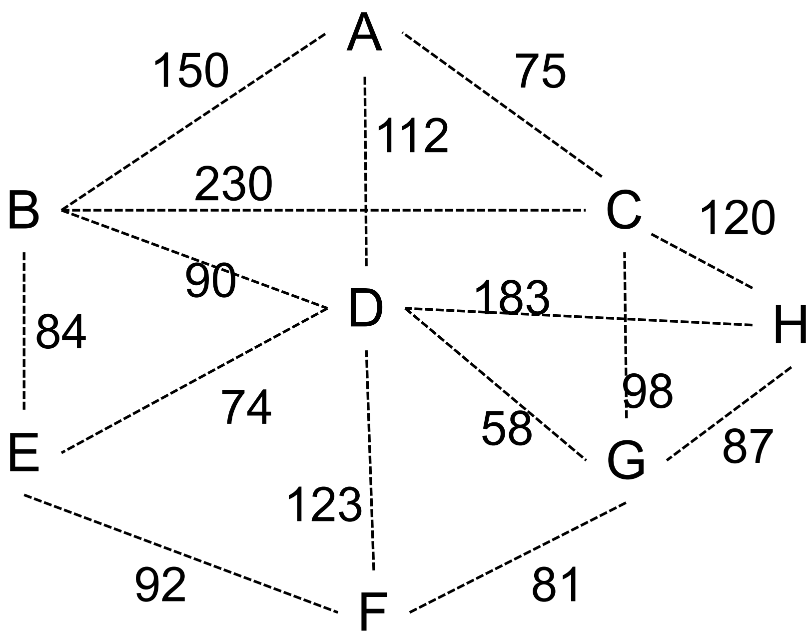
What data structures should we use?

```
prim():  # pseudocode, specifying ADTs

create an APQ pq, which will contain costs and (vertex,edge) pairs
create an empty dictionary locs for locations of vertices in pq
for each v in G
    add (∞, (v,None)) into pq and store location in locs[v]
create an empty list tree, 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 locs
    if e is not None, append e to tree
    for each edge d incident on v
        w = d's opposite vertex from v
        if w is in locs            # and so it is not yet in the tree
            cost = d's cost
            if cost is cheaper than w's entry in pq
                replace ?:(w,?) in pq with cost: (w, d)
 return tree
```

```
prim():  # pseudocode, specifying ADTs

create an APQ pq, which will contain costs and (vertex,edge) pairs
create an empty dictionary locs for locations of vertices in pq
for each v in G
    add (∞, (v,None)) into pq and store location in locs[v]
create an empty list tree, 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 locs
    if e is not None, append e to tree
    for each edge d incident on v
        w = d's opposite vertex from v
        if w is in locs          # and so it is not yet in the tree
            cost = d's cost
            if cost is cheaper than w's entry in pq
                replace ?:(w,?) in pq with cost: (w, d)
return tree
```



Complexity analysis (n vertices and m edges)

heap APQ

$O(n \log n)$

n times round loop
Each time, $O(1)$ to remove from *locs* and add to *tree*, $O(\log n)$ to remove from *pq*, so for the loop without edge updates $O(n \log n)$.

Internal loop for *d* edges, where *d* is max degree of any vertex, but overall there are *m* edges, so *m* times round that loop. Each time $O(\log n)$ to update *pq*. So $O(m \log n)$.

```
prim():  # pseudocode, specifying ADTs

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

unsorted list APQ

$O(n)$

n times round loop
Each time, $O(1)$ to remove from *locs* and add to *tree*, $O(n)$ to find and remove from *pq*, so for the loop without edge updates $O(n^2)$.

Internal loop for *d* edges, where *d* is max degree of any vertex, but overall there are *m* edges, so *m* times round that loop. Each time $O(1)$ to update *pq*. So $O(m)$.

So $O(m \log n)$ overall.
If sparse, = $O(n \log n)$
If dense, = $O(n^2 \log n)$

So $O(n^2)$ overall.

(It is easy to show that Prim must produce a spanning tree.)
Is Prim's algorithm guaranteed to produce a *minimum* spanning tree?

Proof: Let S (!= T) be any minimum spanning tree. Let T be the output of Prim's algorithm. We will show that T has cost less than or equal to S.
Assume the edges of T 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 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 that 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 spanning tree which we call S_{k+1}.
S_{k+1} contains all the edges from T_{k+1}, and its cost is $\leq$ cost of S.
Repeat this argument, but the first edge not in S_{k+1} will now be later than e_{k+1}. Keep repeating the process until we create S_{e-1}, which contains exactly the edges of T, so S_{e-1} = T, and cost(T) = cost(S_{e-1}) $\leq$ cost(S).

Next lecture

Further graph algorithms