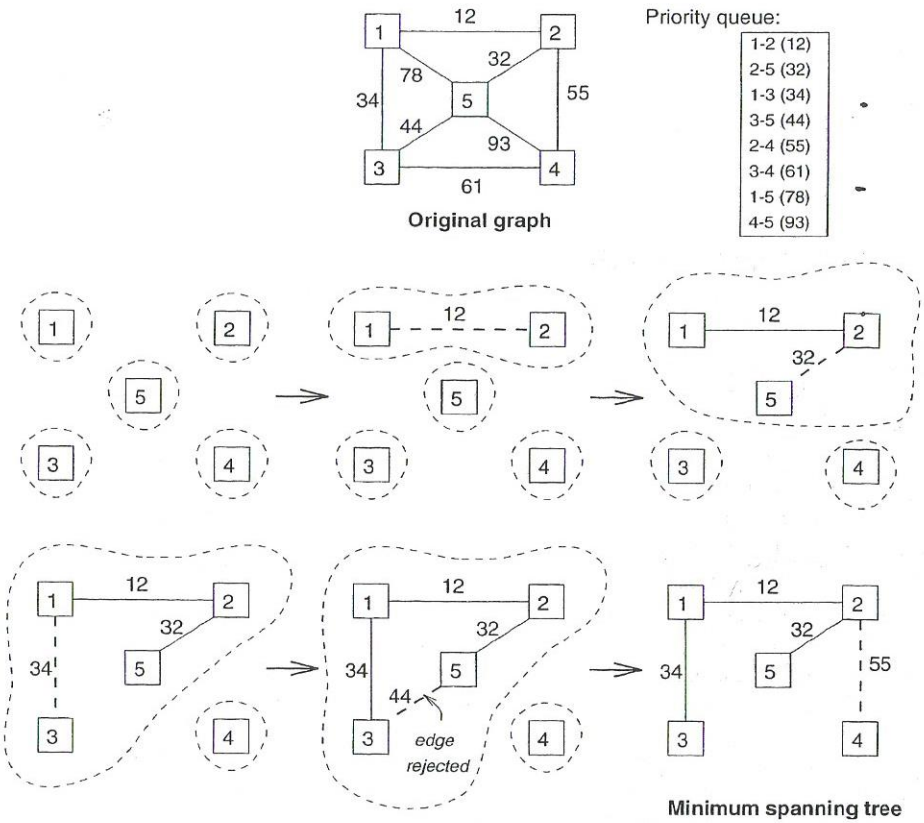

Figure 7.5 Kruskal's algorithm for computing the minimum spanning tree.

```
fillPQ :: (Ord n, Ord w, Ord c) => [(n,w,c)] -> PQueue (c,n,w)
                                -> PQueue (c,n,w)
fillPQ [] pq           = pq
fillPQ ((x,y,w):es) pq = fillPQ es (enPQ (w,x,y) pq)

kruskal :: (Num w, Ix n, Ord w) => Graph n w -> [(w,n,n)]
kruskal g = kruskal' (fillPQ (edgesU g) emptyPQ)
                     (newTable [(x,x) | x<- nodes g])
                     [] 1
   where n             = length (nodes g)
         kruskal' pq t mst i
             | i==n    = mst
             | otherwise = let e@(_,x,y)    = frontPQ pq
                               pq'          = dePQ pq
                               (updated,t') = unionFind (x,y) t
```



```
                in if updated
                   then kruskal' pq' t'(e:mst) (i+1)
                   else kruskal' pq' t  mst     i
```

`kruskal'` implements Step 3 of the algorithm, `mst` is the list containing the edges of the minimum spanning tree and the last parameter keeps track of its length to avoid recomputing it at each step. The iteration stops when $n-1$ edges (where n is the number of nodes) have been added to the solution since a minimum spanning tree always consists of $n-1$ edges. Notice that this program uses no less than three ADTs. Therefore, the efficiency of this algorithm depends on how these ADTs are implemented. Potentially every edge is examined resulting in $O(|E|)$ iterations at the outer level in the worst case.

7.5.2 Prim's algorithm

The second algorithm described is called Prim's algorithm. It works by considering all vertices instead of all edges in the graph. The algorithm divides vertices into two sets: those which have been included in the tree (denoted T) and those which have not (denoted R). Initially, one vertex is arbitrarily placed into the set T (for example, the vertex 1). Then at each stage in the algorithm, among all the edges (u, v) where $u \in T$ and $v \in R$, the edge with the minimum weight is selected. An example is illustrated in Figure 7.6.

The advantage of this algorithm over the previous one is that there is no need to sort all edges and repeatedly update a table of vertices. We now show an implementation of Prim's algorithm. In the next chapter, we will show that this is an example of a *greedy* algorithm.

```
prim g             = prim' [n] ns []
 where (n:ns)      = nodes g
  es               = edgesU g
  prim' t [] mst = mst
        prim' t r mst
                   = let e@(c,u',v') = minimum[(c,u,v)|(u,v,c)<-es,
                                               elem u t, elem v r]
                     in prim' (v':t) (delete v' r) (e:mst)
```

where the function `delete` removes an item from a list.

7.6 Depth-first search trees and forests†

The depth-first search traversal program described in Section 7.3.1 essentially indicates the order in which nodes are visited. A *depth-first search tree* shows not only the nodes but also the edges traversed. Figure 7.7 shows the depth-first search trees that correspond to the depth-first search traversals displayed in Figure 7.2.

This section describes how to implement algorithms that generate depth-first search trees.