a) A minimum spanning tree with weight 73 is the following:

```
  5---1---6
  |     |   |
  1     15  14-15
  |     |     |
  2-4   8-17 |
  |     |   |   |
  3-6   15   |
  |         |
  |         |
  1        |
```

b) The statement is false. To prove this, we provide as a counter-example a graph $G$ for which the (unique) minimum spanning tree is not a shortest path tree for any root $r$. Obviously, also none of the possible shortest path trees is a minimum spanning tree for $G$.

c) We need to change the selection rules so that they work for maximum spanning trees. We consider the invariant that there is a maximum spanning tree $T$ that contains all the chosen edges, and none of the rejected ones. We need to prove that this invariant is preserved after applying a selection rule to choose or reject an edge.

We modify the first selection rule such that it chooses an edge if it has the greatest weight on a cut on which no other edge was chosen. We assume the invariant to be true before this rule is applied. If we choose an edge in $T$, then the invariant is trivially preserved. On the other hand, if we choose an edge $e$ that is not in $T$, then $e$ forms a cycle together with $T$. Consider an edge $e' \in T$ in the cycle that crosses the cut that we used for the choice of $e$. Since we have chosen $e$ as an edge with maximum weight on the cut, we have $c(e) \geq c(e')$. Consequently, the spanning tree $T'$ that we obtain from $T$ by replacing $e'$ with $e$ has a weight that is not smaller than the weight of $T$, and is therefore also a maximum spanning tree. Hence, after choosing $e$, the invariant still holds (with $T'$ instead of $T$).

We modify the second selection rule such that an edge is discarded if it is an undecided edge with the lowest...
weight on a cycle with no discarded edges. We assume the invariant to be true before we apply this rule. If we reject an edge that is not in \( T \), then the invariant is trivially preserved. If, on the other hand, we reject an edge \( e \) of \( T \), then the edge \( e \) divides \( T \) into two components. According to our rule, \( e \) is rejected because it is on a cycle with no other rejected edges. This cycle must contain another edge \( e' \) that is on the cut between the two components of \( T \). Since we have chosen \( e \) as the an edge of minimum weight on the cycle, we have \( c(e) \leq c(e') \). The spanning tree \( T' \) that we obtain by replacing \( e \) with \( e' \) in \( T \) has a weight that is not smaller than the weight of \( T \), and it is therefore also a maximum spanning tree. After rejection of \( e \), the invariant still holds (with \( T' \) instead of \( T \)).

We now know that we are never „making a mistake“ when applying one of the two modified selection rules. For the sake of completeness, we need to prove that we can always apply a selection rule as long as there are undecided edges. Let \((u, v)\) be an undecided edge. If \( u \) and \( v \) lie in different components, then \((u, v)\) is part of at least one cut on which no other edge has been chosen. Since \((u, v)\) is still undecided, we can apply the first selection to choose one edge on the cut (not necessarily \((u, v)\)). If \( u \) and \( v \) lie in the same component, then \((u, v)\) is part of a cycle that otherwise contains only decided edges. Applying the second selection rule, we can discard the edge \((u, v)\).

With our new selection rules, we can adapt the algorithms of Kruskal and of Prim-Dijkstra simply by considering the edges in descending order of their weights. In Kruskal’s algorithm, this simply means that we sort the edges in the reverse order in the beginning. In the algorithm of Prim-Dijkstra, we need to change the priority queue such that an edge with top priority is an edge with maximum weight.

Alternatively, we can modify the edge weights such that we can compute the minimum spanning tree for the modified weights and obtain a maximum spanning tree for the original graph. We could, for example, replace every edge weight with its inverse. This may yield fractional edge weights and will not work with both positive and negative edges weights. To achieve the same result (a reversal of the ordering), it is easier to simply negate the edge weights.

**Exercise 10.2  Fibonacci heaps.**

a) It is possible for a Fibonacci heap to degenerate into a linear list. We provide an iterative algorithm that produces this structure. We start with a heap containing a single tree with two nodes\(^1\), and iteratively produce a longer list. After every iteration, we obtain a heap that has exactly one node in the root list to which a linear list is appended. Let \( k \) be the key of this node, and let \( a, b, c \) be three new keys such that \( a < b < c < k \). In every iteration, we perform the following operations:

\[
\text{insert}(a); \text{insert}(b); \text{insert}(c); \text{extract}\_\text{min}(); \text{decrease}\_\text{key}(c, a); \text{extract}\_\text{min}()
\]

We first add the keys \( a, b, c \) and then perform an \text{extract}\_\text{min}-operation that extracts \( a \) and cleans up the root list. The other two inserted nodes \( b \) and \( c \) (with degree 0) are combined into a new heap of degree 1. The heap containing key \( k \) as root also has degree 1, and hence the two heaps are merged into a single heap with degree 2. More precisely, the heap containing the key \( k \) will be appended to the one with root \( b \), since by definition \( b < k \). This increases the height of the chain by one. The root, however, has now two sons, namely one with key \( k \) and one with key \( c \). We perform a \text{decrease}\_\text{key}-operation setting the value of \( c \) to \( a \). This separates \( c \) from its father \( b \) and appends it to the root list, marking \( b \) in the process. Finally, we perform an \text{extract}\_\text{min}-operation removing \( a \), and we are back in the desired configuration.

b) This is not possible, because otherwise we would be able to sort in linear time. This could be done by first inserting all \( n \) numbers as keys, and then performing \( n \) extractions of the minimum. If both operations run in amortized constant time, we would get an overall running time of \( \mathcal{O}(n) \) for the insert operations, and \( \mathcal{O}(n) \) for the delete\_\text{min}-operations. The total running time would be \( \mathcal{O}(n) \). Since we know that, when only comparisons are allowed, the running time for sorting has a lower bound of \( \Omega(n \log n) \), this is not possible.

c) We start by inserting \( 2^n + 1 \) different keys in the Fibonacci heap. Since no extraction operation was performed, the root list then consist of nodes with degree 0. Each of these nodes is a Binomial tree \( B_0 \) by definition.

---

\(^1\)We can obtain such a heap simply by inserting three elements and then extracting the minimum.
Now we perform an **extract_min**-operation. First, the smallest key is removed from the heap, then the root list is cleaned up. During this phase, all nodes with the same degree are merged pairwise. Each time two nodes are merged, one node becomes the son of the other. This is exactly how a Binomial tree is built. We start with a tree $B_0$ (a node with degree 0 in the Fibonacci heap) and we build a tree $B_1$ by merging the nodes into a heap that now has degree 1. We proceed analogously for degree $i$. This process continues until there are only two nodes left in the root list that have the same degree. By induction, all the elements of the root list are Binomial trees (note again that we only insert and delete nodes).

It is easy to see that at the end we are left with only one node in the root list with degree $n$ containing $2^n$ nodes. By induction, this is exactly $B_n$.

**Exercise 10.3**  
*Algorithm design: path planning in labyrinths.*

We define a directed graph with a node for each possible state of the system. A state is completely determined by three parameters:

1. The position where the robot is located.
2. The direction in which the robot is facing.
3. Whether the robot is standing or moving.

For every position where the robot can be located we then have 8 possible states (4 directions multiplied by 2 possibilities, whether the robot is standing or moving). We construct our graph accordingly with 8 nodes per square in the labyrinth. In addition, we need a node for the target state, representing the escape from the labyrinth. We add an edge for each movement, rotation and stopping operation, with weights according to the given duration of the operations. A shortest path in the graph from the starting state of the robot to the target state corresponds to a sequence of operations of the robot, that allows it to escape as quickly as possible.

The following picture illustrates this construction on a small section of a labyrinth. The labels of the nodes indicate the direction that the robot is facing; double arrows indicate that the robot is moving. We could of course remove the states that are not reachable.

Since, for each position in the labyrinth, we have used a constant number of nodes (precisely 8) and a constant number of edges (at most 16), the running time of Dijkstra’s algorithm on this graph is $O(n \log n)$, where $n$ is the number of squares in the labyrinth.