Exercise 1: Graphs

We characterize an undirected, weighted graph G as the set V of vertices, the set $E \subseteq V \times V$ of edges, and the weight function $w : E \to \mathbb{R}$. Prove the following statements, or find a counterexample.

(a) In a connected graph G, let (u, v) be an edge with minimal weight, i.e. $w(u, v) \leq w(u', v')$ for all $(u', v') \in E$. Then (u, v) is part of a minimum spanning tree of G.

(b) If all edge weights in G are distinct, then G has exactly one minimum spanning tree.

(c) If $G' = (V, E')$ with $E' \subseteq E$, $|E'| = |V| - 1$ is a subgraph of G and G' is connected, then G' is a spanning tree of G.

(d) If all edge weights in G are distinct, the minimum spanning tree and the maximum spanning tree have at most $\lceil n/2 \rceil$ edges in common.

(e) Let G be a *directed* graph, instead. If all edges in G that have negative weight are not part of any cycle in G, then Dijkstra's algorithm correctly finds all shortest paths in G.

Exercise 2: Binary Search Tree

Consider the following incomplete implementation of a binary search tree:

```
BSTNode <- R6::R6Class("BSTNode",
  public = list(
    value = NULL,
    left = NULL,
    right = NULL,
    parent = NULL,  # either the parent node, or the BST object itself (if this node is root)
    initialize = function(value, parent) {
      self$value <- value
      self$parent <- parent
    },
    print = function() {
      cat(sprintf("BSTNode(value = %s)\n", self$value))
    },
    insert = function(value) {
      if (value == self$value) {
        # value already in tree
        # we choose to not allow duplicates
        return()
      }
      if (value < self$value) {
        target <- "left"
      } else {
        target <- "right"
      }
      if (is.null(self[[target]])) {
        self[[target]] <- BSTNode$new(value, self)
      } else {
        self[[target]]$insert(value)
      }
    },
    search = function(value) {
```

```
      if (value == self$value) {
        return(self)
      }
      # TODO: your code here (exercise part (a))
    },
    findMinimum = function() {
      # TODO: your code here (exercise part (b))
    },
    delete = function() {
      # TODO: your code here (exercise part (c))
    }
  )
)

BST <- R6::R6Class("BST",
  public = list(
    root = NULL,
    insert = function(value) {
      if (is.null(self$root)) {
        self$root <- BSTNode$new(value, self)
      } else {
        self$root$insert(value)
      }
    },
    search = function(value) {
      if (is.null(self$root)) {
        return(NULL)
      }
      self$root$search(value)
    },
    findMinimum = function() {
      if (is.null(self$root)) {
        return(NULL)
      }
      self$root$findMinimum()
    },
    print = function(indent.step = 6) {
      # very simple printer that prints a sideways representation of the tree
      if (is.null(self$root)) {
        cat("Tree:\n<empty>\n")
        return()
      }
      cat("Tree:\n")
      draw <- function(node, indent = 0) {
        if (is.null(node)) return()
        draw(node$right, indent + indent.step)
        cat(sprintf("%*s%s\n", indent, "", node$value))
        draw(node$left, indent + indent.step)
      }
      draw(self$root, 0)
    }
  )
)

# Behavior:
bst <- BST$new()
bst$insert(10)
bst$insert(20)
```

```
bst$insert(5)
print(bst)

#> Tree:
#>       20
#> 10
#>       5

bst$insert(7)
bst$insert(15)
bst$insert(30)
print(bst)

#> Tree:
#>             30
#>       20
#>             15
#> 10
#>             7
#>       5
```

(a) Implement the `search` method of the `BSTNode` class by inserting your code in place of the `TODO` comment. The solution should not take you more than 15 lines of code (excluding comments).

Expected behavior:

```
# continuing with the object from above:
bst$search(10)
#> BSTNode(value = 10)
bst$search(30)
#> BSTNode(value = 30)
bst$search(99)
#> NULL
```

(b) Implement the `findMinimum` method of the `BSTNode` class by inserting your code in place of the `TODO` comment. The function should return the node (not the value!) with the smallest value in the subtree rooted at the current node.

Expected behavior:

```
# continuing with the object from above:
bst$findMinimum()
#> BSTNode(value = 5)
bst$search(20)$findMinimum()
#> BSTNode(value = 15)
```

(c) Implement the `delete` method of the `BSTNode` class by inserting your code in place of the `TODO` comment. The function should delete the node from the tree by changing the `left` or `right` pointer of the parent, as well as the `parent` pointers of its children. You can use `identical(self, ...)` to check if `self` is the left or right child of its parent. Also make sure to cover the case where the node is the root of the tree.

Hint: In the case where the node has two children, it makes sense to use the `findMinimum()` function to find the node to replace the deleted node with.

Expected behavior:

```
# continuing with the object from above:
bst$search(30)$delete()
print(bst)
#> Tree:
#>     20
#>           15
#> 10
```

```
#>             7
#>       5
bst$root$delete()
print(bst)
#> Tree:
#>     20
#> 15
#>             7
#>       5
```