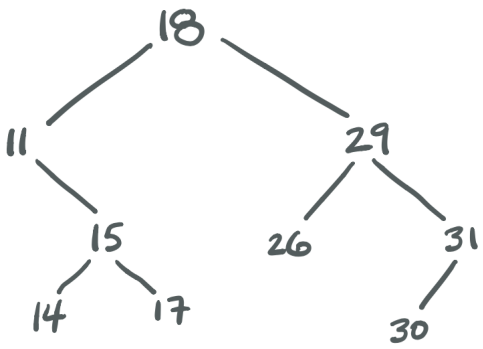
[7 MARKS]

This question involves a series of small short-answer questions. You can type your answer for all of them (no diagrams are required). We have provided you with a file called **Q1_answers.txt**. Use PyCharm to open that file and add your answers to it. Hand in this file on MarkUs.

Part (a) [1 MARK]

What is the post-order traversal of this tree?

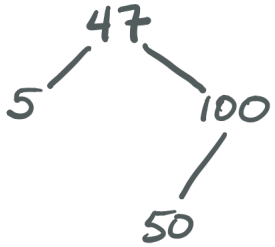


Part (b) [1 MARK]

Suppose we insert the following values into an empty binary search tree, in this order: 9, 3, 5, 4, 10, 7. What is the height of the resulting tree?

Part (c) [1 MARK]

Give two different sequences of insertion into an empty binary search tree that will yield this result:



Part (d) [3 MARKS]

Here is the helper function for our mergesort implementation:

```
def _merge(lst1: List, lst2: List) -> List:
    """Return a sorted list with the elements in <lst1> and <lst2>.

    Precondition: <lst1> and <lst2> are sorted.
    """
    index1 = 0
    index2 = 0
    merged = []
    while index1 < len(lst1) and index2 < len(lst2):
        if lst1[index1] <= lst2[index2]:
            merged.append(lst1[index1])
            index1 += 1
        else:
            merged.append(lst2[index2])
            index2 += 1
    # assert ???
    return merged + lst1[index1:] + lst2[index2:]
```

Below are several assertions that we might place after the while loop. For each indicate whether it always succeeds, always fails, or may succeed or fail depending on the input lists. Assume that the preconditions are met.

1. `index1 < len(lst1) and index2 < len(lst2)`

2. `index1 < len(lst1) or not(index2 >= len(lst2))`

3. `not(index1 < len(lst1)) or not(index2 < len(lst2))`

Part (e) [1 MARK]

Here is a graph showing the results of an experiment comparing the performance of insertion sort, mergesort, and quicksort on increasingly large inputs. The elements in the lists being sorted were in random order. Which sorting technique corresponds to A, B, and C in the graph?