# Mission 11: Search and Rescue

Start date: 18 September 2017
**Due: 22 September 2017, 23:59**

Readings:

- None

## Background

A small scare broke out in the server room of the spaceship last evening, a massive rack of servers came crashing down due to a misaligned screw. Thankfully our engineers onboard the spaceship have managed to recover most of the data, and are frantically trying to resume normal IT infrastructure operations onboard the spaceship.

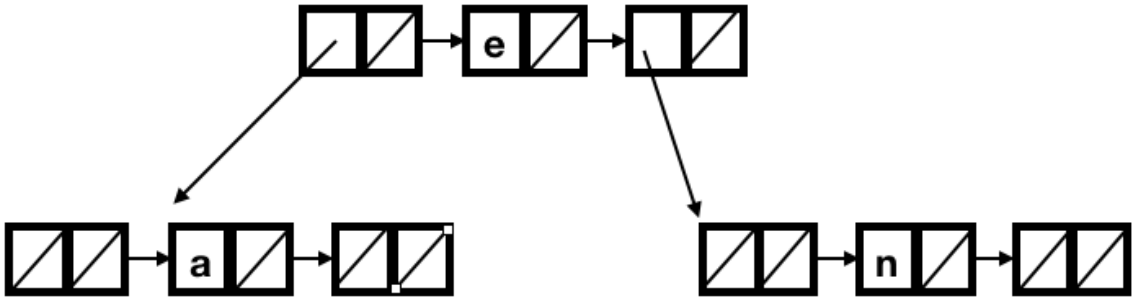
The accident, unfortunately, corrupted the master list of all student names. This list is crucial for issuing the graduation documents of the Source Academy, and the captain needs it in a few weeks. The IT staff have spent hours restoring the name list to the best of their ability, using advanced fuzzy-quantum-de-corruptification techniques. To improve the speed of accessing the records, the IT staff have arranged the student names in a binary search tree. Your job is to verify that the process worked, for selected student names.

This mission has **three** tasks. Please remember to press "*Save*" button before you submit this mission.

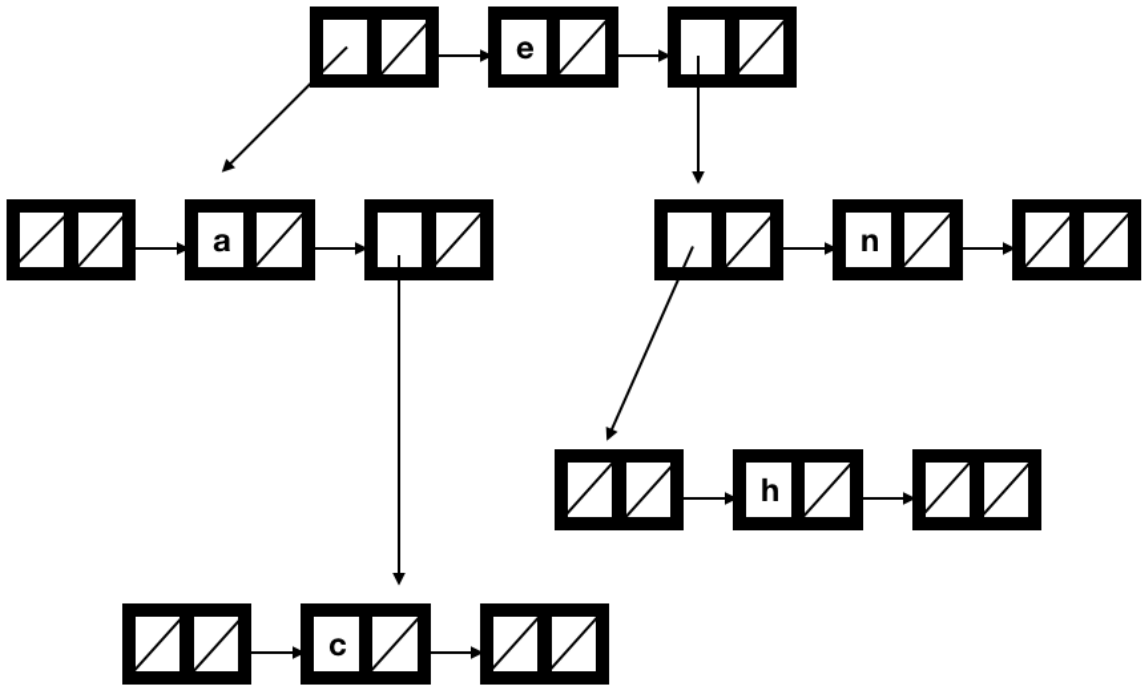
## Task 1:

Recall from the lectures that trees of data items are lists whose elements are either data items or trees. We will use trees to represent the student names so that we can very efficiently find out whether a given name is included or not. So in our case the data items are all strings, such as `"Tan Weijie, Benjamin"`. Recall that the comparison operators `<`, `>`, `<=`, `>=` and `===` in Source can be used to compare two strings lexicographically.

Before we get to search for such names, let us recall **binary trees**, a special kind of trees, from the lectures. A binary tree is either the empty list or a list with three elements. The first element is a binary tree, called its *left subtree*, the second element is a data item called its *value*, in our case a string, and the third element is a binary tree, called its *right subtree*. Such a binary tree is depicted below.




A **binary search tree** is a binary tree that has the following property: All data items in its left subtree are smaller than its value and all data items in its right subtree are bigger its value (We assume no duplicates here). Here is an example of binary search tree.




You are given a binary search tree called `cadet_names` with strings as data items, each representing the name of one of our 120 cadets in the Source Academy.

The abstraction of binary trees supports the access functions `is_empty_binary_tree`, `left_subtree_of`, `value_of` and `right_subtree_of` to test if the given binary tree is empty, and if not to get its left subtree, value and right subtree, respectively. With these functions, we do not need to specify how a binary tree is represented.

Your first task in this mission is to write the function `binary_search_tree_to_string`,

that takes a binary search tree as argument whose data items are all strings, and returns a string. The result string contains all strings of the binary search tree, each followed by a newline character, in alphabetical order. Note that a newline character is produced by `\n`, for example `"Hello\nWorld"` is a string with the characters `Hello`, followed by a newline character, followed by the characters `World`.

For example, your function `binary_search_tree_to_string` returns the following string when the binary search tree depicted above is passed as argument.

```
a
c
g
h
n
```

## Task 2:

Write a function `find` that takes a binary search tree `bst`, whose data items are strings, as first argument, and a string `name` as arguments. The result of `find(bst, name)` should be `true` if `s` occurs as value in `bst`, and `false` otherwise.

## Task 3:

The runtime of the function `find` not only depends on the number of data items in the binary search tree, but also on the arrangement of the data items in the tree.

1. Give a binary search tree `bst1` with 7 data items such that your function `find` needs 14 string comparisons, before `find(bst1, "hello");` returns `true`.
2. Give a binary search tree `bst2` with 7 data items such that for any string `s`, the call `find(bst2, s)` requires at most 6 string comparisons.
3. Let us call `c(n)` the maximal number of string comparisons needed for finding any value in a binary search tree of size `n`. If we arrange the values in the binary search tree in the best possible way, give the order of growth for the function `c(n)`, using Theta notation.

## Submission

Submit your mission on the Source Academy.

**IMPORTANT**: Make sure that everything for your programs to work is on the left hand side (Editor) and not in the Side Content! This is because only the programs in the Editor are preserved when your Avenger grades your submission.