FIT1045/53 Algorithmic Problem Solving – Workshop 10.

Objectives

The **objectives of this workshop** are:

- To implement Quicksort
- To think about and implement Binary Search Tree algorithms

Useful Links:

For this workshop, you may find it useful to review some of the following concepts:

- Recursion (Lecture 14)

Task 1:

Background: Quicksort is a recursive sorting algorithm. The idea is that, to sort a given list, we need to pick an item from the list called a **pivot**. We then divide the list into two parts, those items less than the pivot, and those items greater than (or equal to) the pivot. We place all items less than the pivot on the left of the pivot, and the items greater than (or equal to) the pivot on the right. This process is called **partition**.

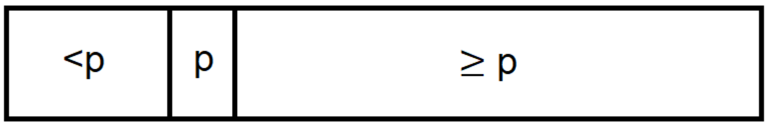

Figure 1: The state of a list after partition. p is the pivot

We then need to sort the items on the left of the pivot, and the items on the right of the pivot, since we know that the pivot is certainly in its correct final position. We do this by recursively quicksorting the left and right parts of the list, stopping when a sublist has 1 or fewer elements, since this sublist is trivially sorted.
Note that the pivot must be chosen somehow. It is easy to simply choose the first item in the list as the pivot, or you can choose it at random. This will not always divide the list exactly in half (sometimes it will not even be close to half) but that is ok.

a) Implement Quicksort. That is, write a function `quicksort(a_list)` which takes as input a list and returns a list containing the same elements, in sorted order. The algorithm must be an implementation of quicksort (discussed above).

b) **(Optional, non assesed)** Implement in-place quicksort. This means that you do not create any lists other than the input list, the elements of the original list simply get shuffled around.

Task 2:

a) Write a function `count(T, v)` which takes as input a binary tree in the form shown in lectures, and determines how many nodes are in the subtree rooted at node `v`. **Hint: use recursion**.

b) Using the function from part a), write a function `balance(T, v)` which takes as input a binary tree `T` and a node `v` and computes the balance of `v` (number of nodes in the left subtree of `v` - number of nodes in the right subtree of `v`)

Example:

```
tree = [(2, 1), (3, None), (5, 4), (None, None), (None, None), (None, None)]
count(tree, 0) will return 6
count(tree, 1) will return 2
count(tree, 2) will return 3
count(tree, 4) will return 1

balance(tree, 0) = count(tree, 2) − count(tree, 1) = 3 − 2 = 1
```

Task 3 (Optional):

Write a function `insert(bst, labels, item)` which takes as input a bst of the form shown in lectures, and inserts `item` to the correct position in `bst` using the binary search tree insertion algorithm.
Example:

```
tree = [(2, 1), (3, None), (5, 4), (None, None), (None, None), (None, None)]
labels = [9, 12, −1, 10, 5, −8]
after running insert(tree, labels, 3) the state of the bst would be
tree = [(2, 1), (3, None), (5, 4), (None, None), (6, None), (None, None), (None, None)]
labels = [9, 12, −1, 10, 5, −8, 3]
```