2 Activity 2 (10 points)

In Week 4, we talked about a series of advanced data types such as trees. In particular, we mentioned **binary search trees** (or BST), which are trees with the following property: for any parent node, its data value must be higher than the data from any child node to its left, and lower than the data of any child node to its right. Creating and maintaining such a tree comes with a cost, as every time new nodes are added, removed or changed, the developer needs to ensure that this key property still holds. In return, the tree is fast to search through - finding a value in it takes `O(log n)`. For large n this should be faster than a regular linked list.

In this activity, you will implement a binary search tree and study its algorithmic complexity. You will also create a linked list and compare the complexities of both structures.

2.1 The binary search tree class (1 point)

To begin with, create a Python file `binarysearchtree.py` and implement a **TreeNode** class. The class should include attributes and methods to store and manipulate:

- A data value (or cargo) of the node;
- A link to its left child;
- A link to its right child;
- A way to print its value;

Now, create a class *BinarySearchTree*. It should have:

- The root of the tree as an attribute;
- An optional limit in size;
- A method `is_empty()` to check if the tree is empty;
- A method `is_full()` to check if the tree is full;

2.2 Operations on BSTs (2 points)

Expand your tree with operations. You should implement the following methods:

- A method `search()`, which receives a data value and searches the tree for it, returning a Boolean indicating whether or not the item is there.
- A method `insert()`, which receives a data value and inserts it as a new node in the tree. The tree may include duplicate values.
- A method `delete()`, which deletes a value from the tree, and mends the tree back together in case it breaks, keeping the tree's binary search property intact.
- A method `traverse()`, which returns the values stored across the nodes of the tree in ascending order.
- A method `print_tree()` to print a visual representation of the tree. Think about how you would prefer to visualise a tree if you could only use `print()` calls; it doesn't need to be the `__str__` method.

2.3 Random trees' simulation (2 points)

You should now have a functioning binary search tree. It is time to study the time complexity of its search. To achieve this, do the following:

1. Create a new script, called `complexity.py` and import your **BinarySearchTree** class;
2. In `complexity.py`, create a function `random_tree(n)`, which takes an input n and generates a tree of size n by populating it with n random integers from 1 to 1000;
3. Then create a list X of equally spaced numbers from 5 to 100 on step size 5 (so, 5, 10, 15 etc.)
4. For each value s in X, generate 1000 random trees of size s, and search them for the value 42, storing the average time spent by the `search` call into a list Y;

Notes :

- The `time` module might be useful here.
- In case your laptop does not have the capacity to run a simulation of this size, reduce either the size of the trees or the size of X or both. In this case, declare two constants in `complexity.py`, one called $TREE_SIZE$ and the other one called $NUMBER_OF_TREES$ where you specify the numbers that work for you.

2.4 Complexity analysis for BSTs (2 points)

We can now see how the time spent to search in a BST varies according to its size n. In *complexity.py*, add more code to:

1. Plot X against Y. X stores links to the trees of different sizes. Y stores the times spent searching. Use the code in Listing 7 for that.
2. Add a comment in *complexity.py* that describes in your own words the complexity of BST search based on the shape of the graph you plotted. **Hint**: relationships could be e.g. linear, sub-linear, exponential, quadratic, logarithmic etc as explained in the lectures in Week 4. Start your comment with the words *Complexity analysis X vs Y"* so that we can find it easily in your code.
3. Create a new list *Y2* with estimates of average search time t under an ideal linear relationship to the size of the trees in X. **Hint**: In a linear relationship, $t = c * n + b$, where c and b are constants. Using the search time t for $n = 5$ and $n = 10$ from Y, you can find c and b and estimate the other values of t in $Y2$.
4. In the same way, create a third list $Y3$ with estimates of average search time under an ideal logarithmic relationship to the size of the trees in X. **Hint**: In a logarithmic relationship, $t = c * log(n) + b$, where c and b are constants. Using the search time t for $n = 5$ and $n = 10$ from Y, you can find c and b and estimate the other values of t in $Y3$. Use $log(n) = log_2(n)$.
5. Plot the three curves (X against Y, $Y2$ and $Y3$) using the code found in Listing 8.
6. Add another comment in *complexity.py* that describes in your words how the initial graph compares to an ideal linear or logarithmic complexity. What could be the reason wny your Y line does not follow exactly the same line as e.g. $Y3$? Is there any issue with our class Binary Search Tree or with the way we created tree objects that we'd need to fix so that Y gets closer

to an ideal complexity? Start your comment with the words *Complexity analysis X vs Y, Y2 and Y3"* so that we can find it easily in your code.

Note : The `time` module's `time()` function records time as seen by the processor. Try not to run any demanding processing on your laptop while executing the code - if you have other processes running in the background, the time recorded will be impacted by them, tampering with your results. This may mean your plots will get wobbly - if that happens, try to stop any other processes that might interfere and run the code a few times until the plotted lines become smoother.

```
import matplotlib.pyplot as plt
plt.plot(X, Y)
plt.xlabel('Size of trees')
plt.ylabel('Search time')
plt.ticklabel_format(axis='both', style='sci', scilimits=(0,0))
plt.show()
```

Listing 7: How to plot a simple X vs Y graph, where X and Y are number lists of the same length

```
import matplotlib.pyplot as plt
plt.plot(X, Y)
plt.plot(X, Y2)
plt.plot(X, Y3)
plt.legend(['BST','Linear','Logarithmic'])
plt.xlabel('Size of trees')
plt.ylabel('Search time')
plt.ticklabel_format(axis='both', style='sci', scilimits=(0,0))
plt.show()
```

Listing 8: How to plot a multi-line graph with line labels

2.5 The linked list class (1 point)

For a better understanding of the importance of complexity, let us now compare BSTs to standard linked lists.

First, create a Python file `linkedlist.py` and implement a **ListNode** class. It should have attributes and methods for:

- A data value (or cargo) stored in the node;
- A link to its next node;
- A way to print the data value in the node;

Now, create a class **LinkedList**. It should have:

- The first node as an attribute;
- An optional limit in size;
- A method `is_empty()` to verify if the list is empty;
- A method `is_full()` to verify if the list is full;
- A method `__str__` to print a visual representation of the list.

2.6 Operations on linked lists (1 point)

Expand your **LinkedList** with proper operations. You should implement the following methods:

- A method `search()`, which receives a data value and searches the list for it, returning a Boolean indicating whether or not the item is there.
- A method `insert()`, which receives a data value as a parameter and inserts it as a new node in the list. Duplicate values are allowed.
- A method `delete()`, which receives a data value as a parameter and deletes its first occurrence from the list.
- A method `traverse()`, which returns the values of the nodes in the linked order.

2.7 Comparing the two data types (1 point)

Repeat the steps from Section 2.3 using linked lists. Use the same list X to generate the lists and save the average search times in a new list $Y4$.

Then analyse the time complexity. To do so:

1. Plot X vs $Y4$ by executing the code found in Listing 9.
2. Add a new comment in your code to describe how linked lists compare to BSTs. Start your comment with the words *Complexity analysis X vs Y, Y2, Y3 and Y4"* so that we can find it easily in your code.

```
import matplotlib.pyplot as plt
plt.plot(X, Y)
plt.plot(X, Y2)
plt.plot(X, Y3)
plt.plot(X, Y4)
plt.legend(['BST','Linear','Logarithmic','LL'])
plt.xlabel('Size of trees')
plt.ylabel('Search time')
plt.ticklabel_format(axis='both', style='sci', scilimits=(0,0))
plt.show()
```

Listing 9: Adding a new plot to the mix