# Algorithms and Data Structures

## I. Background

In computer science academia the realm of data structures is frequently brought up. However, with various programming languages and their respective IDEs comes inherited libraries that essentially shortcut the use of said data structure. Moreover, it is crucial to understand how they operate on an empirical level, in order to distinguish their levels of efficiency. The case studies of this task are doubly-linked lists and binary search trees as ADTs, both of which will be re-created via Java with set operations: ADD(S, x), REMOVE(S, x), IS-ELEMENT(S, x), SET-EMPTY(S), SET-SIZE(S), UNION(S,T), INTERSECTION(S, T), DIFFERENCE(S, T), SUBSET(S, T), where S, T are sets, and x, is an element. After implementing, analyses will be performed, which cover: discussing the complexity of ADD and IS-ELEMENT if it's assumed that the doubly-linked list is sorted, discussing the naivety of performing UNION on binary search trees by individual taking the elements in one and inserting them into the other. The second part of this task will cover the differences in the IS-ELEMENT times between the doubly-linked list and binary search tree, the doubly linked list size, and depth (or height) of the binary search tree.

## II. How to run the program

To run the program only the main method needs to be manipulated. Sample code, in addition to the code required for the purposes of the task, are displayed for ease of comprehension. Make sure reference is made to the respective Binary Search Tree or Doubly-Linked List object before calling a method on it. Note that the given implementation also contains extra operations such as 'print' and 'display', for the Doubly-Linked List and Binary Search Tree respectively.

## III. Part 1: Doubly Linked List

A Doubly-Linked list is made up of two classes: DoublyLinked<X> (where X specifies the type of the elements within the list), and Node, which constructs the elements in the list. Each node has (for the purposes of the task) an integer element, and, as per the requirements of a Doubly-Linked List, a pointer to the succeeding and preceding nodes.

ADD(S,x): First, via the constructor, create a new node, 'temp', with the data/element field assigned to x, the head/next assigned to null – noting the new node will be appended to the end – and then the tail/previous assigned to the tail of the doubly-linked list. If a tail (the final) element exists, then assign the 'next' value for the tail node to be x. Otherwise, assign the tail of the list to temp. Additionally, if the head is also null, then assign head to temp. Running time: **O(1)**

REMOVE(S,x): As the element to delete is passed as an integer, first the method identifies the corresponding node. Then, consider whether the list is empty or not. Next, if non-empty, we consider separately if the head is the required node to delete or a node after it. Running time: **O(1)**

IS-ELEMENT(S,x): First, initialize a Boolean flag variable as false. Then starting at the head of the list, check if the corresponding element of the node is the number required, and if so, set the flag to true. Return the flag in the end. Running time: **O(n)**

SET-EMPTY(S): If the head is empty, then the list is empty. Running time: **O(1)**

SET-SIZE(S): Start at the head and successively move to the following node. If each node is null then add 1 to a count variable. Running time: **O(n)**

UNION(S,T): To unionize sets S and T, iterate through each element of T, and if not contained in S, then insert. Running time: **O(n)**

INTERSECTION(S,T): Similar implementation to UNION(S,T). Running time: **O(n)**

DIFFERENCE(S,T): Similar implementation to UNION(S,T). Running time: **O(n)**

SUBSET(S,T): Similar implementation to UNION(S,T) except method initializes a Boolean flag to true, and if any node is encountered in T that is not in S, then set flag to false. Running time: **O(n)**

## IV. Part 1: Binary Search Tree

Binary Search Tree is made up of a BST class and a Node class. The BST class initializes a root node, which is the starting point of a given binary search tree. The node class consists of the following fields: data (which is the integer the node is representing), and nodes left and right, which are the respective branches.

ADD(S,x): Running time: O(n)

REMOVE(S,x): Running time: $O(n^2)$

IS-ELEMENT(S,x): Running time: O(n), although best-case would be O(log n) in base 2.

SET-EMPTY(S): Running time: O(1)

SET-SIZE(S): Running time: $O(n^2)$

UNION(S,T): Running time: $O(n^2)$

INTERSECTION(S,T): Running time: $O(n^3)$

DIFFERENCE(S,T): Running time: $O(n^3)$

SUBSET(S,T): Running time: $O(n^3)$

## V. Part 1: Sub-part C

Take the following scenario into consideration: the Doubly-Linked List implementation assumes/maintains a sorted list. This has an impact on the running times of ADD and IS-ELEMENT. For the former, the running time is increased, as the add method must now consider whether the given element to add conforms to the sorting type of the list. For IS-ELEMENT, the running time is decreased, as by knowing the sorting type and the element to determine the existence of, the program can cut the sample size of the list more-so than in an unordered, random list.

## VI. Part 1: Sub-part D

A naive implementation of the UNION method for the Binary Search Tree implementation of Abstract Data Types would be to take one set and to add each element into the other. The issue with this implementation is that the set to add the elements to may not be a completely filled (or balanced) tree. Therefore, when using our add operation

we would have to traverse a height of O(n), where in a balanced case it would be O(log n). Therefore, it would behove one to initialize a new list, add the elements of set S to it in a balanced fashion, and then similarly for set S, thus constructing a tree more efficient in traversing.

## VII. Part 2A

The average IS-ELEMENT time for both ADT implementations are as follows:

Average time for Doubly-Linked List: 85167 ns

Average time for Binary Search Tree: 138 ns

Output of SET-SIZE: 20 000

Height of Binary Search Tree with the required set (using height()): 36