**Algorithms and Data Structures**

**Dynamic Set and Empirical Study**

**Fatma Al Sayegh**

**2515006a**

**Running the code:**

Open Driver.java, change the FILE\_PATH variable to your file path, and run driver.java.

**Part 1:**

**A-B)**

For both implementations, the classes had Item as generics, and both classes had a static Node class with Item generics as well. The Node class would store the Linked nodes and the held key.

* The Doubly Linked list class is **DoublySet** has in its static Node class instance members a next and previous **Node<Item>**, and an **Item** as a key.
* The Binary Search tree class is **BinarySearchSet** and has in its static Node class a left and right **Node<Item>**, and an Item as a key

The methods in these Set classes are:

* **getNewNode(Item):** returns a new unlinked Node object with Item as its key
* **getNodeFromItem(Item)**: returns the node object in the set from that item, null if it doesn’t exit. Does this by looping through the nodes and checking if the key matches the item.
  + **DoublySet**: Loops by starting at the head, comparing the node’s key, and then jumping into the next node and repeating until there is a match. So the worst case is **O(n)** where n is the number of elements according to setSize
  + **BinarySearchSet**: Loops by starting at the head and comparing the values, if its less, it goes to the left node, if its more, it goes to the right node. It keeps doing that until it finds a match. Since each time it splits either left or right, basically going down one row in the tree, then the worst case is **O(h)** where h is the height of the tree = log(n) if balanced. That is since the maximum number of rows it can go down is h
* **setSize**: counts the size of the set
  + **DoublySet**: traverses all nodes and implements a counter, takes **O(n)** since it goes through all elements.
  + **BinarySearchSet**: uses a recursive method size(Node, counter) which increments the counter and re-calls itself for the left and right nodes. Traverses all elements so takes **O(n)** where n is the number of elements
* **setEmpty**: returns if the set is empty by checking if the root or the head is null
* **isElement(Item):** returns true if an element exists, does that by checking if getNodeFromItem returns a value or null. Takes time as long as getNodeFromItem
* **add(Item)**: adds an item to the set
  + for both implementations, if the head or root nodes are null, they are set to a new node created by the getNewNode method.
  + **DoublySet**: Starts at head and keeps going through the next nodes, if any node has the same key as the one being added, stop because distinct. When last node is reached, add new node there. Has to traverse through all elements so it takes **O(n)**
  + **BinarySearchSet**: Starts at root and goes left if less than or right if greater than, stops if its equal since distinct. Stops when there isn’t left or right to go to and inserts element there. Each time it goes left or right, it goes down a row in the tree, so the worst case is **O(h)** where h is the height of the tree.
* **remove(Item)**: finds the item in the tree then removes it.
  + DoublySet: gets the element by using getNodeFromItemand then updates the left and right nodes, has a worst case same as getNodeFromItem which is **O(n)** where n is the size of the tree
  + BinarySearchSet: gets the element by using getNodeFromItemwhich takes **O(h)**, in cases where it has no left or no right, the total time is **O(h)**.  
    In cases where it has a left and right child, the minimum has to be found which takes O(h), so the total worst case is **O(h)**
* **union(Set b):** where a Set is either a DoublySet or a BinarySearchSet Creates a new set and loops through the elements of the current Set and Set b, then calls add on each element
  + **DoublySet:** Goes through all the elements of the first set and calls add upon them, does the same for the second set. Since the add method takes O(k) and going through the first set takes O(n), and the second O(m) then the total worst case is **O(n^2 + m^2)**
  + **BinarySearchSet:** Loops through all elements of both trees, this takes O(n) for the first and O(m) for the second. Each time it calls add which takes O(h) where h is the height of the new tree which is log(n+m) If balanced. So the worst case is **O(h(n+m))** where n and m are the sizes of first and second set
* **intersect(Set b):** loops through the first set and does intersection
  + **DoublySet:** Does intersection by creating a root node and setting the next and previous ones. Looping through the first set takes O(n), checking if its an element takes O(m), setting the next node takes O(1), so the total time is **O(n\*m)**
  + **BinarySearchSet:** Loops through the elements of first tree which is O(n) and check if its in tree b which is O(m), if they are, adds them to a new tree which is O(h). So worst case is **O(n\*(m+h))**
* **difference(Set b):** loops through the first set, checks if element is not in the Set b and adds it to a new tree.
  + **DoublySet:** Creates a root node, loops through all the elements of the first set, this takes O(n), and sets the next element of the node if its not an element in the other set, checks using isElement which takes O(m), so the total time is **O(m\*n)**
  + **BinarySearchSet:** Loops through first set which takes O(n), check if its element in other set which takes O(m), adds to new set which takes O(h), so **O(n\*(m+h))**
* **subset(Set b):** checks if the Set tree is a subset of b, does so by looping over all elements of the first Set then checking if they are all elements of b using isElement
  + **DoublySet:** Loops through all elements in first which takes O(n), checks if isElement in second which takes O(m), so O(n\*m)
  + **BinarySearchSet:** Loops through first set which takes O(n), checks if in second which takes O(m), so total is **O(n\*m)**

**C) For isElement:** If the doubly linked list were sorted, checking for isElement still have the same time complexity since elements have to be accessed in order and not randomly. Meaning to check if an element exist, the algorithm would still have to start from the root and go forward, so the time remains the same.

**For Add:** The element now has to be added in a sorted manner, meaning it shouldn’t always be added at the end of the tree, the element however may have to be added at the end of the tree, so the worst case is still O(n)

**D)** To improve the union of the BST’s, we have to improve the time O(h\*(n+m)). Adding the elements to the tree is what cases the h to appear in the worst case. We can try to remove it by keeping track of the leaves while creating the union. This could be done by creating an array for the first binary tree and an array for the second one. The binary trees would be stored in the arrays from Left to right starting from the bottom left node to the bottom right one using Inorder traversal. This way the BST would project its shape onto a 2d array. This operation takes O(n) for the first tree and O(m) for the second one. The total now is O(m+n). Both arrays are sorted because of the nature of BST’s, so they can be merged to create a new sorted array with distinct elements. This operation takes also m+n time since they are already sorted and there will be only one loop. Finally, the BSTS could be reconstructed from the arrays, where the root could be the center of the array, and we go left or right accordingly. This would also take m+n, so the total complexity would be O(m+n)

**Part 2:**

1. Upon loading the file, running the isElement 100 times for each different type of sets, and computing the average, it turned out that the average time for the Binary Search tree was 5447.91 nanoseconds, which is around 0.005447 milliseconds. The average time for the Doubly Linked List was 215159.51 nanoseconds which is around 0.215 milliseconds. The times makes sense since the binary tree’s max time for isElement is O(h) where h is the height, whereas the max time for the doubly linked list is O(n). In all cases, the height of the binary tree is less than the number of elements in it, so it will take less time than the doubly linked list. The binary search tree is around 40 times faster.
2. Both set sizes turned out to be 16536 elements
3. The height of the tree was computed by recursion and turned out to be 35