Karlis Siders (2467273S)

Part 0: How To

All work is contained in 3 files: DLLDynamicSet.java, BSTDynamicSet.java, and Tester.java, which is runnable. The Dynamic Set files contain the implementation of the dynamic set with a doubly linked list (DLL) and binary search tree (BST), and Tester.java contains methods for reading an array from a text file and generating an array of given size with random integers and a main method which compares both implementations.

Easiest way to run the code:

1) Make a project called 'ADS-AE2' (I don't know whether the name is necessary) in Eclipse
2) Add 'num' folder with 'int20k.txt' in it and replace 'src' folder from zip file

Part 1

a) Doubly Linked List Dynamic Set

Methods:

- Main:
  - add(): O(n) because the set has to be searched (O(n)) before adding an element (O(1))
  - remove(): O(n) (same as add())
  - isElement(): O(n) because of searching
  - setEmpty(): O(1) because size is kept track of
  - setSize(): O(1) (same as setEmpty())
- Set-theoretical:
  - union(): O(n^2) as it goes through all elements of one set and adds them to another (O(n)), searching for each element to avoid duplicates (O(n))
  - intersection(): O(n^2) (same as union())
  - difference(): O(n^2) (same as above)
  - subset(): O(n^2) (same as above)

b) Binary Search Tree Dynamic Set

Methods:

- Main:
  - add(): Because a set requires searching beforehand, the search algorithm is the leading factor in the running time complexity, which can be **O(n)** in the worst case (if elements are unbalanced in increasing/descending order) but is usually **O(h)**, where h is the height
  - remove(): O(h)/O(n) (same as add())
  - isElement(): O(h)/O(n) (same as above)

    - setEmpty(): O(1) as every node keeps track of the amount of its children
- Set-theoretical:
    - union(): O($n \log n$) as the algorithm traverses the whole tree to search for an existing element (O($\log n$)) and does this for each element in another tree (O(n))
    - intersection():O($n \log n$) (same as union())
    - difference(): O($n \log n$) (same as above)
    - subset(): O($n \log n$) (same as above)

c) Sorted Doubly Linked List Dynamic Set (SDLLDS – palindrome)

ADD would become a bit more complex (**O(2n)**, which is still O(n)) as the original search to eliminate repetition would be needed along with the search for the correct space to insert the element in, which theoretically could be improved to be O(n) if those two searches were combined.

IS-ELEMENT could be improved a bit to be **O(n/2)** (which is still O(n)) in the worst case if the head and tail were compared before deciding to go forwards or backwards in the search. Because linked lists do not support indexing, binary search would still be impossible.

d) Better BST Set Union

The running time of the naïve implementation of union is O($n \log n$), which is worse than linear running complexity. A way to improve it would be to use unionMerge(), which, firstly, traverses both trees (which would be O(n + m) for their respective sizes n + m) to make them into linked lists. Secondly, these two lists are then merged with the merge() algorithm from Merge Sort, which also has O(n + m) complexity. Thirdly, a new BST Set could be created from this merged list, which also requires O(n + m) complexity. Thus, the complexity of unionMerge() would be **O(n + m)**.

Part 2

a) Comparison

The average running time of IS-ELEMENT over 100 calls on both implementations is as follows:

- Dynamic Linked List: 69437 ns
- Binary Search Tree: 6951 ns

As clearly visible, the BST implementation of the Dynamic Set is approximately 10 times faster than DLL's implementation. This is because the running time of DLL is, on average, O(n/2), while the average running time of search in BST is O(h), where h is the height and is, at best, O($\log n$). The height is minimised in balanced binary search trees, while randomly generated arrays (like int20k) are more likely to create BSTs that are close to being balanced.

b) 16536 (which is smaller than 20 000 because repeated integers are excluded)
c) 34