DIT181 Data Structures and Algorithms: Assignment 3

Question 1 (30 points)

Add the following methods to the code you implemented by using the skeleton code `Tree.java`, which we have provided you (it is a modified version of the skeleton file for Assignment 2):

`public boolean isAVL():` Checks if the tree is an AVL tree.

`public void insertAVL(Item i):` Inserts `item i` into an AVL tree. Beware that the tree **must** still stay as an AVL tree after the item has been inserted.

`public void removeAVL(Item i):` Removes `item i` from an AVL tree. Beware that the tree **must** still stay as an AVL tree after the item has been removed.

Question 2 (40 points)

In this part of your assignment, you will implement linear and quadratic probing hash table, and perform simulations to compare the observed performance of hashing with the theoretical results. For this question, you will use the `HashTable.java` skeleton file. Implement a probing hash table and insert 10,000 randomly generated integers into the table and count the average number of probes used. This average number of probes used is the **average cost** of a successful search. Repeat this test 10 times and calculate *minimum*, *maximum* and *average* values of **average cost**. Run the test for both **linear** and **quadratic probing** (you may want to use two separate java files for that), and do it for final load factors $\lambda = 0.1, 0.2, 0.3., 0.4, 0.5, \ldots, 0.9$. Always choose the capacity of the table so that no rehashing is needed. For instance, for final load factor $\lambda = 0.4$, in order to be insert 10,000 integers into the hash table, the size of the table should be approximately 25,000 (i.e., $10000/\lambda = 10000/0.4 = 25000$). You must make some adjustments to make table size a **prime number**. For instance, 25,013 is the prime number that is slightly larger than 25,000. You can refer to the following link for the list of prime numbers: http://compoasso.free.fr/primelistweb/page/prime/liste_online_en.php

At the end of your simulations, please fill in the following table.

Load Factor (λ)	**Average Cost for a Successful Search**					
	Linear Probing			**Quadratic Probing**		
	minimum	**maximum**	**average**	**minimum**	**maximum**	**average**
0.1	0	3	0.059	0	3	0.0546
0.2	0	7	0.1269	0	4	0.1192
0.3	0	8	0.1953	0	5	0.1897
0.4	0	12	0.3299	0	6	0.2692
0.5	0	18	0.4681	0	9	0.3941
0.6	0	27	0.7605	0	11	0.5277
0.7	0	50	1.0912	0	15	0.7162
0.8	0	120	1.9734	0	23	0.9924
0.9	0	239	3.8572	0	46	1.5411

Question 3 (30 points)

Implement heapsort algorithm by extending the `Lab3Sorting` class from the the skeleton code file `Lab3Sorting.java`, which is similar to `Lab1Sorting.java` provided for Assignment 1. Compare the performance of *insertion sort*, *mergesort*, *quicksort* and *heapsort* algorithms using the Bench class provided with the skeleton code. To do that you will need to port your implementations of insertion sort, mergesort and quicksort from Assignment 1. The benchmarking code will run the following:

1. Each sorting algorithm (i.e., insertion sort, mergesort, quicksort and heapsort algorithm) will be run with a randomly ordered sequence of integers as input, and its run time in seconds will be reported.
2. Similarly, each sorting algorithm will be run with an almost sorted sequence of integers as input, with its run time reported.
3. Each sorting algorithm will be also run with a sorted sequence of integers as input, with its run time reported.
4. The steps 1-3 will be repeated for array lengths of 10, 30, 100, 300, 1000, 3000, 10000, 30000, 100000

After running the benchmark, fill in the following table with the results you obtain:

Input size (N)	**Sorting Algorithm Run Time (s)**				
	Input	**Insertion sort**	**Quicksort**	**Mergesort**	**Heapsort**
10	Random				
	95% sorted				
	Sorted				
30	Random				
	95% sorted				
	Sorted				
100	Random				
	95% sorted				
	Sorted				
300	Random				
	95% sorted				
	Sorted				
1000	Random				
	95% sorted				
	Sorted				
3000	Random				
	95% sorted				
	Sorted				
10000	Random				
	95% sorted				

	Sorted				
30000	Random				
	95% sorted				
	Sorted				
100000	Random				
	95% sorted				
	Sorted				

After gathering all the data, guess the complexity of every algorithm for every of the three kinds of inputs.

Grading Details

Question 1 (30 points)

- **7 points** for the implementation of the method `public boolean isAVL()`
- **8 points** for the implementation of the method `public void insertAVL(Item i)`
- **15 points** for the implementation of the method `public void removeAVL(Item i)`

Question 2 (40 points)

- **10 points** for the implementation of linear probing
- **10 points** for the implementation of quadratic probing
- **10 points** for the implementation of the simulations for linear probing
- **10 points** for the implementation of the simulations for quadratic probing

Question 3 (30 points)

- **14 points** for the implementation of heapsort algorithm
- **16 points** for executing the simulations to fill in the table