PROJECT II

Title: Sorting using a Queue vs Sorting a Binary Search Tree

Students:

1. Junaid Magdum (94685845)
2. Probuddho Chakraborty (57605211)

Data Structures Implemented:

1. Queue
2. Binary Search Tree

Summary:

We see daily applications of sorting in real life. Many data structures can be used to implement various types of sorting. In this project, we aim to use Queue and Binary Search Tree data structures to understand two different approaches to sorting. The programming language used here is Python.

Queue:

- A Queue is a linear structure which follows an order in which the operations are performed.
- The order is First-In-First-Out (FIFO).

Binary Search Tree:

Binary Search Tree is a node-based binary tree data structure which has the following properties:

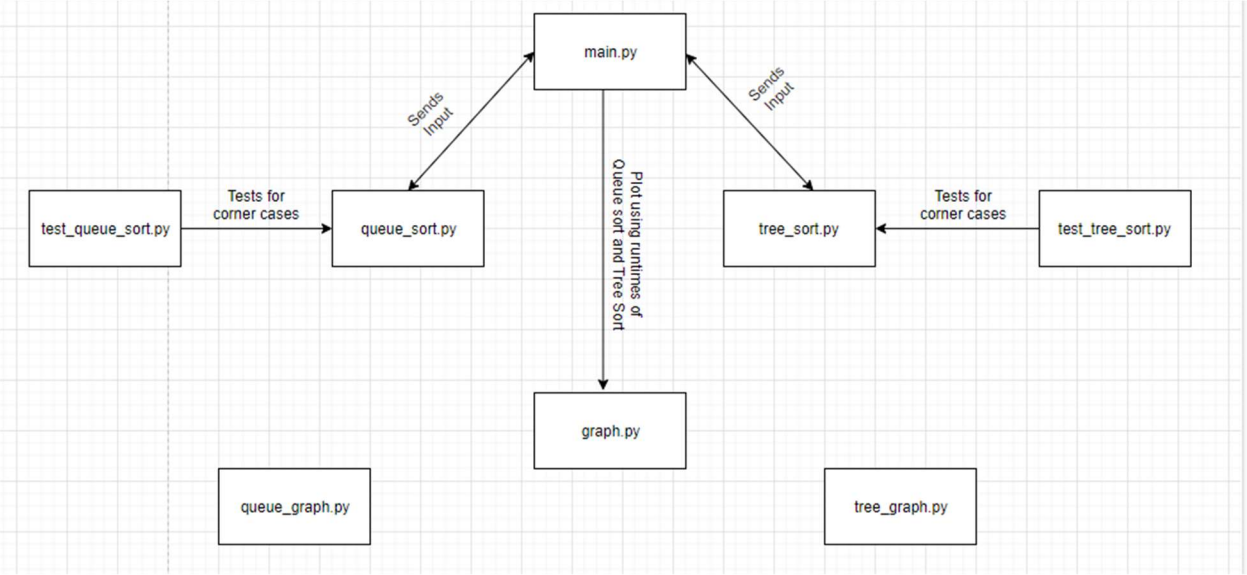
- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

Language: Python

Design Choices:

- For every type of sort, there is a dedicated python file that implements logic
- The code is modularized, and all the other files are imported in the main file and the appropriate functions are called from there.

Design of Experiments:



Algorithm Implementation Files:

1. **main.py**
    a. The main file initially, imports all other files with their specific functions
    b. We then define a lambda function that will calculate current time in milliseconds
    c. The first function creates a random dataset and returns this in the form of an array/ list.
    d. In the main function following things are done:
        i. The upper bound for the size of the dataset is set.
        ii. For a specific number of times:
            1. Random datasets are generated.
            2. The size of the datasets is appended to a list (all_lengths) which will later be used for plotting the graphs.
            3. Sorting using a Binary Search Tree (sortTree) is performed and the time for this operation is appended to a list (tree_time) which will later be used for plotting the graphs.
            4. Sorting using a Queue (sortQueue) is performed and the time for this operation is appended to a list (queue_time) which will later be used for plotting the graphs.
            5. The plot function from the graph.py file is called with the lists as arguments

2. **queue_sort.py**
    a. This file details all the functions for performing sorting using a queue

b. The basic concept for sorting using a queue is finding the index of the minimum element in the input array and inserting the minimum element at the rear end of the array.
c. This is done until a sorted array is obtained.

3. **tree_sort.py**
    a. This file details all the functions for performing sorting using a binary search tree
    b. The basic concept for sorting using a binary search tree is inserting the elements of the array while maintaining the properties of the binary search tree.
    c. Once the binary search tree is complete, we return the inorder traversal of the tree to get the sorted list.

4. **graph.py**
    a. This file contains the code for plotting of the collective graphs.
    b. The plot function takes the lists (all_lengths, tree_time, queue_time) as input arguments and plots the graph that shows both of the graphs simultaneously.
    c. The x-axis represents the time taken for the sorting techniques and the y-axis represents the dataset sizes that correspond to each individual iteration for which the time for both data structures has been recorded.
    d. We have used scatter plots so as to better represent large datasets.

5. **queue_graph.py**
    a. This file contains the code for the singular graph of queue sort vs dataset size

6. **tree_graph.py**
    a. This file contains the code for the singular graph of tree sort vs dataset size

7. **test_queue_sort.py**
    a. Test module that feeds in sample values to the queue_sort module and checks for corner cases.
    b. Types of inputs: sorted list, reverse sorted list and a happy case.

8. **test_tree_sort.py**
    a. Test module that feeds in sample values to the tree_sort module and checks for corner cases.
    b. Types of inputs: sorted list, reverse sorted list and a happy case.

Design of Experiments:

1. Run main.py
2. The main method calls in a random data set method that generates data.
3. The main file calls the dedicated sort methods with the input as the random data set.
4. Once both the data sets are sorted, we record their running time in a list.
5. Using the time list and the data set size, we plot different graphs for experimental analysis.

Experimental Analysis:

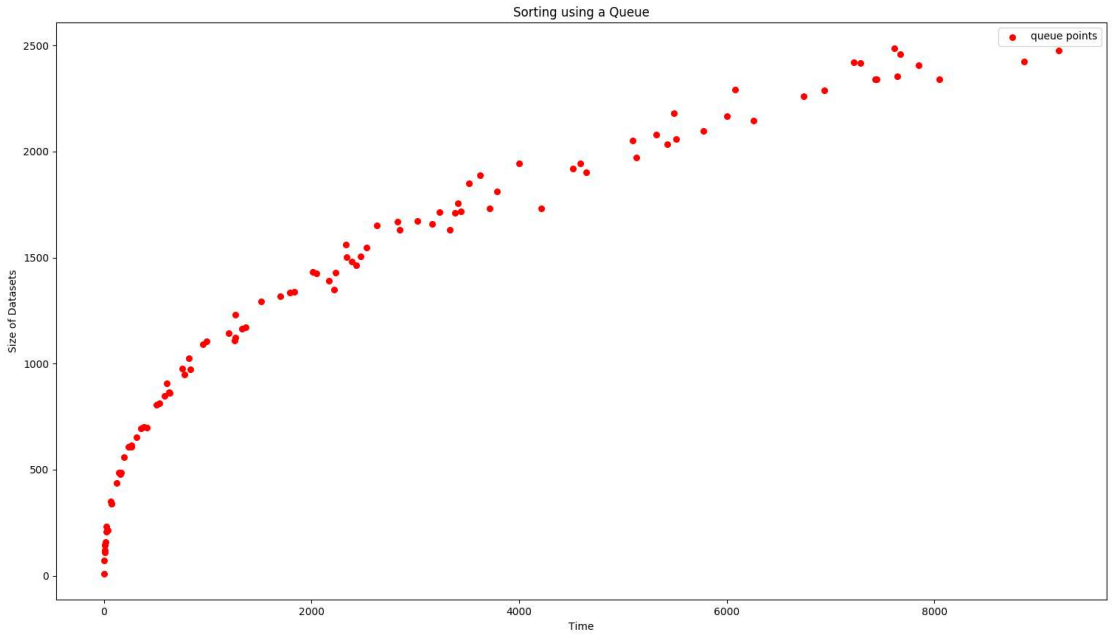
1. **Time Complexity:**
   I. After observation of the tree_time and queue_time lists, it is evident that sorting using a binary search tree is much faster as compared to that using a queue.
   II. This is because the average time complexity for tree sort is $O(n\ logn)$ as compared to $O(n^2)$ for queue sort, where 'n' is the number of elements in the input.

2. **Space Complexity:**
   I. The space taken by tree sort however is more as compared to queue sort since queue sort performs the sorting operation in-place.
   II. Hence, no extra memory is used for sorting using a queue i.e. $O(1)$ space complexity.
   III. Comparatively, tree sort inserts the elements of list as nodes in a tree and hence, the space taken up during tree sort is the number of nodes generated.
   IV. Hence, final space complexity for sorting using a binary search tree is $O(n)$, where 'n' is the number of unique elements in the input.

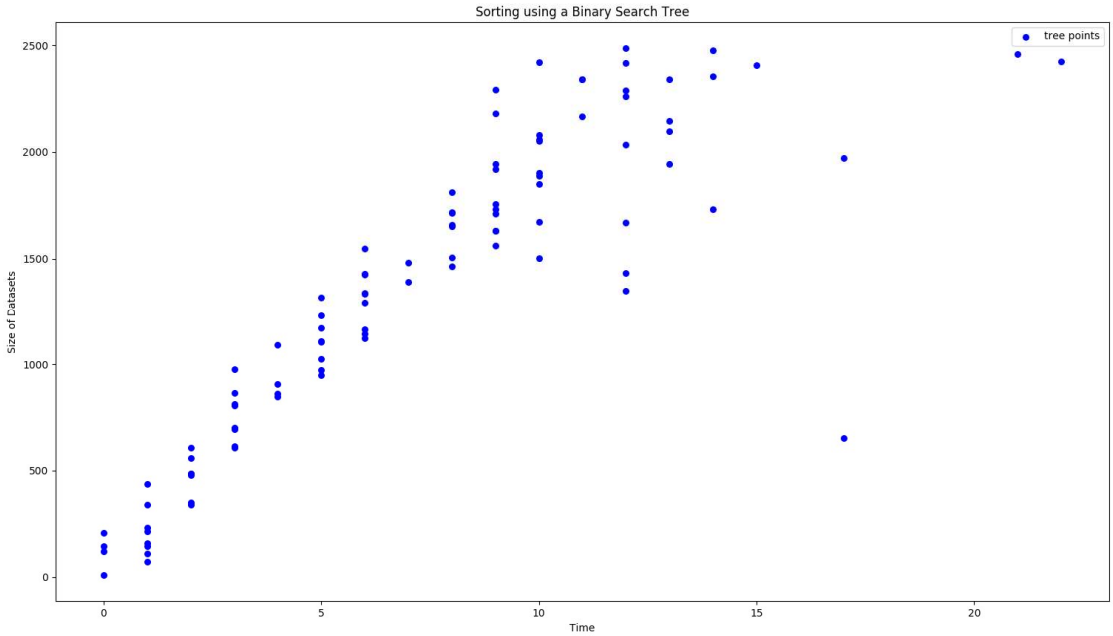
   Note: For each average test case, several datasets are generated, and the dataset sizes are the same even though they are generated in a completely random order. This maintains consistency of the dataset while introducing randomness.

Complexity	**Sorting using a Queue**	**Sorting using a Binary Search Tree**
Time Complexity	$O(n^2)$	$O(n\ logn)$
Space Complexity	$O(1)$	$O(n)$

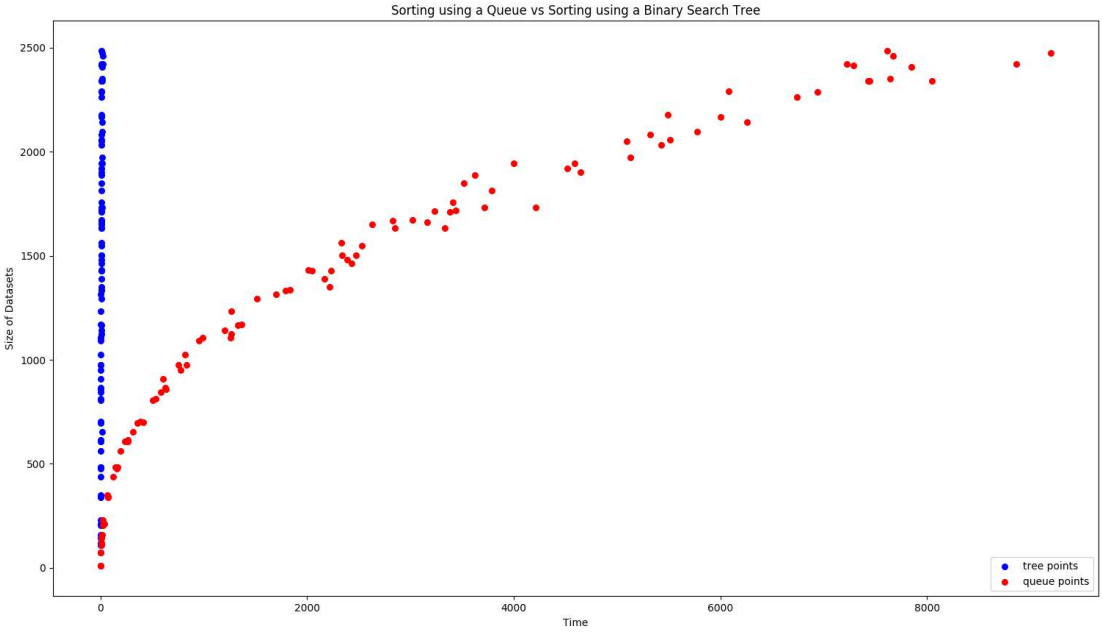
3. **Graphs:**
   - Sorting using a Queue



- Sorting using a Binary Search Tree



- Sorting using a Queue vs Sorting using a Binary Search Tree



4. Corner Cases

I. Data is already sorted

a. Tree Sort Performance

The binary search tree becomes a right skewed binary search tree that tries to traverse the entire height of the tree. If the height of the tree is greater than the recursive stack's memory, then the tree sort fails in this case.

b. Queue Sort Performance

Queue sort doesn't understand that the data is already sorted. It tries to find the minimum and fit it right again at the same spot. This is an unnecessary check, but it never fails. The time complexity is still $O(n^2)$.

II. Data is sorted in reverse manner

a. Tree Sort Performance

The binary search tree becomes a left skewed binary search tree that tries to traverse the entire height of the tree. If the height of the tree is greater than the recursive stack's memory, then tree sort fails in this case.

c. Queue Sort Performance

Queue sort makes the maximum number of comparisons in this case. It tries to find the minimum value, which is always at the end of the list, and fit it at its correct spot. The time complexity is still $O(n^2)$.

Conclusion:

- Sorting using a Queue vs Sorting using a Binary Search Tree varies based on the focus of the user and his respective application.
- Based on the above analysis and graphs, time sensitive applications will benefit from using tree sort as their preferred sorting choice between the two.
- On the contrary, applications with an emphasis on memory usage, should look to queue sort for sorting an array as it completes the entire procedure with no extra space used.
- The edge case for tree sort is that the recursive stack's memory should be greater than the number of nodes so that it doesn't fail for cases where the data is sorted (ascending or descending).