Task 1: Coarse-grained data structures

Design Considerations:
The CoarseGrainedList & the CoarseGrainedTree both have two global variables, namely a pointer that points to the beginning of the data structure (called the root of the binary tree, and the head of the list) and the Lock that locks the entire data structure whenever an element is added or removed. Both locks use the Reentrantlock java library. Both variables are declared as volatile so that multiple threads can have access to the same variable. To achieve coarse-grained locking, both the add and the remove function in each class are formatted as follows:

```
lock.lock();
try{
        //adding/removing element
}finally{
      lock.unlock();
}
```

a. Singly Linked List

Each node in the list is represented through a class called Node, which contains the element of type T and a pointer to the next Node in the list. The program reasons in cases every time an element is added to or removed from the list.
For example, when an element t is added it could either be (a) appended to the end, (b) inserted into the middle, or (c) added to the beginning of the list, or (d) the list is empty and the element is the first element in the list. The if-else if statement from lines 29-35 in the ‘add’ function takes care of the latter two cases (c & d). The else statement that follows in line 36 takes care of the first two cases (a & b) by entering a loop that finds a suitable position to add the element.
To remove an element, the function checks whether the element to be removed is the head (a), whether it’s the last element in the list (b) or else just removes the node (c). To do so, the function loops through the list to find the element passed as parameter to the function.

b. Binary-tree

Each node in the binary tree is represented through a Node class, which contains the element of type T, a pointer to the parent node, and two pointers pointing to the left and right children of the node. Only the root has its parent pointer set to null and leaf nodes in the tree have both their children pointers set to null.
When an element is added to the list, the function checks whether the list is empty to create a new root. In case it is not, the function finds the suitable position to add the node to the tree. This is done through a walk down the tree, where the walk depends on the value of the element to be added. When the element is smaller than the node’s element encountered, the walk continues along it’s left child (right otherwise) until the node does not have any children.

Removing an element presents more cases to consider. The node to be removed may either have no children (a), a right or a left child (b), or two children (c). Case (a) is simple because the node to remove just needs to be unlinked from the parent. Similarly, the child of the node to be removed in case (b) needs to be linked with the parent of the node to remove and vice versa. Lines 96-101 take care of cases a & b. Case (c) is trickier because it involves finding the node's successor, linking it with the parent and the child of the node to be removed, and making sure that the link between the successor's child is not broken in case the successor is not the direct child of the node to remove.

Hypotheses:
The performance of both the linked list and the binary tree is decreased when the number of threads operating on the data structure increases, i.e. it does not scale well to the number of threads. This is because using coarse-grained locking is a very inefficient locking procedure as it leaves the entire data structure inaccessible for other threads. Operations are therefore not performed concurrently, and the contention between the threads to hold the lock decreases the overall performance.

a. *Singly Linked List:*

Operations on the singly linked list will be performed in a reasonable time when the number of elements stored is small. This is because it takes less loop iterations to find the node's element to be deleted/added as less elements need to be traversed starting from the head. For example, the worst case scenario occurs when an element is added/removed from the end of the list. Thus the worst case time complexity is $\Theta(n)$ where n is the number of elements in the list. On the other hand, the best case scenario occurs when an element is added/removed to the front, i.e. the constant time c it takes to perform an operation by the CPU.
One of the biggest downfalls to the list performance is that elements can only be accessed sequentially starting from the head. In comparison, arrays can access elements randomly in constant time. However, that does not necessarily mean that adding/removing elements will take less time in an array. Both data structures will take similar time because an array will need to increment/decrement the memory location of every element that follows after adding/removing an element. In the worst case that is n (adding/removing from the beginning of the list) and constant time c in the best case (adding/removing at the end of the list).

b. *Binary Tree:*

Operations on a binary tree have a way better performance than when they do on the linked list, as it has slower order of growth than the linked list. For example, adding an element to the tree takes $\Theta(h)$ time (where h is the height of the tree) because elements are always appended to the tree as leafs. Deleting an element has the worst time complexity of $\Theta(h)$ when the node to be deleted is the leaf node, or the successor of the removed node is a leaf. However, it may be argued that the average time complexity is $\Theta(h/2)$, when the node and its successor are within the upper half of the tree.

Thus the performance of the binary tree scales well to the number of elements stored since operations take $\Theta(h)$ time. However, the height of the binary tree varies depending on the order in which the elements are added. For example, if they are added in a non-decreasing order, the height of the tree h will equal to the number of elements n stored in the tree. Thus, operations will take n time (as bad as the linked list). However, according to *Theorem 12.4* the 'expected height of a randomly built binary tree on n nodes, is $\Theta(\lg n)$' [1]. It is no coincidence that the same operations on a heap data structure take the same amount of time, as a heap stores values in a complete binary tree.

Evaluation:

Experimental setup:

To test the above hypothesis, the running time of the two data structures is measured on a DAS5 node. The input size n increases by intervals of 3,000 starting by 3,000 and ends at 60,000 elements. Each recorded running time is the average of 10 runs with the given input size. This should provide enough reliable data to determine the time complexity of the two data structures.

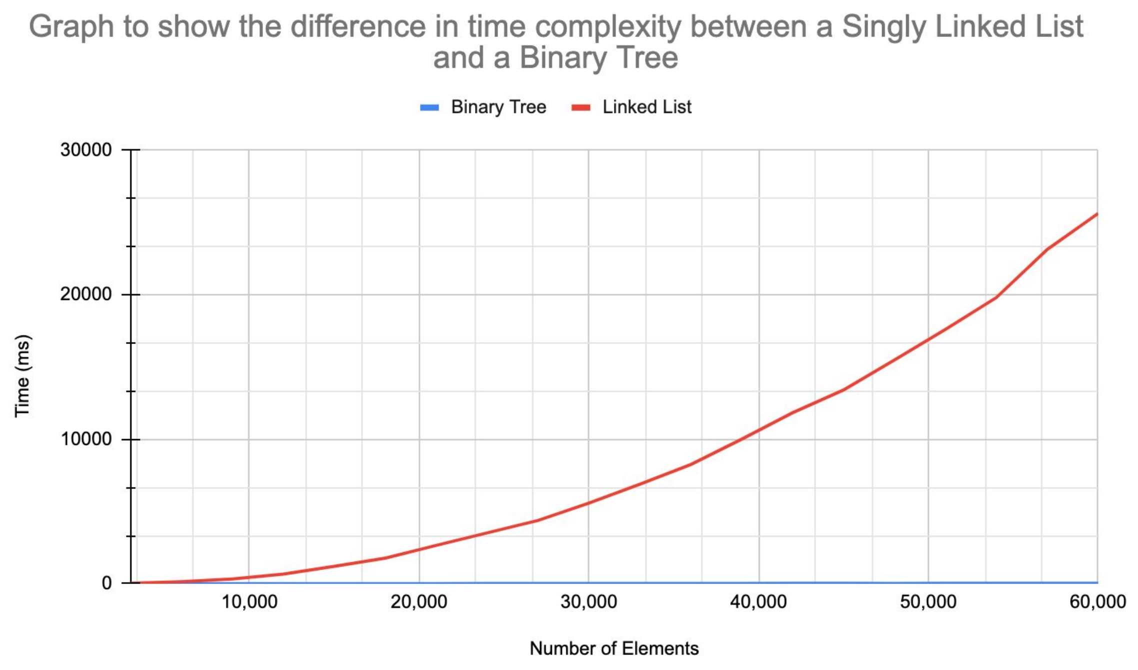
In addition, the efficiency of the coarse-grained locking mechanism is tested by having 1 thread, 500 threads and 1,000 threads operating on the data structure with the different input sizes. 1,000 threads (and even more) operating on a data structure occurs in production, which is why it is used as the upper bound in this experiment.

Independent variable: Number of elements.
Dependent Variable: Time (ms).
Controlled Variables: Operations are done with integers, WorkTime is always set to 0, number of threads.

Table to show the time complexity of the two data structures as the input size increases and the number of threads operating on them.

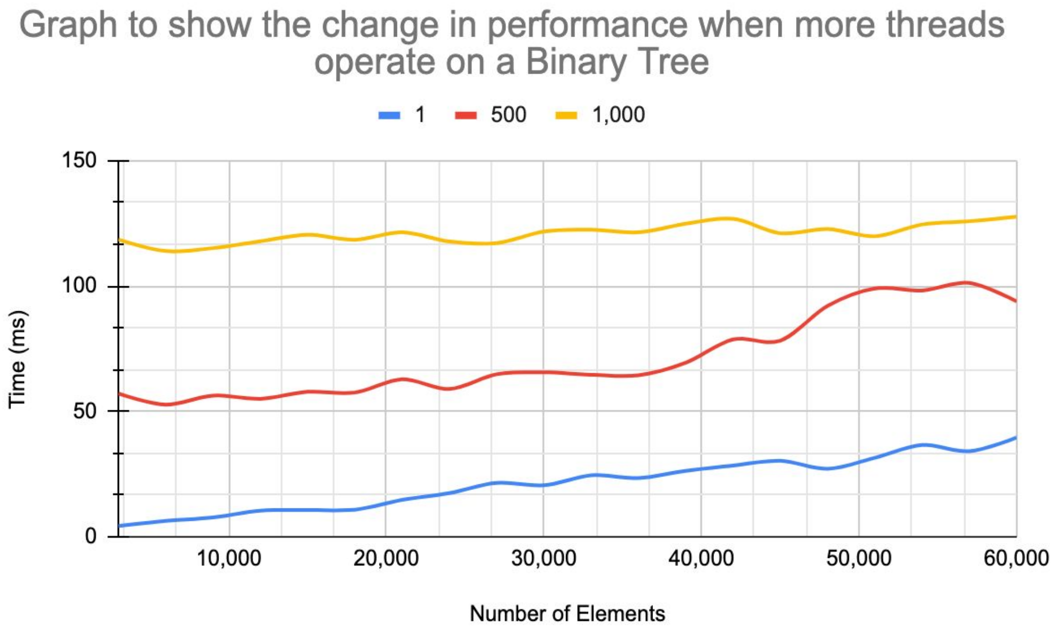
	Number of threads operating on Data Structure					
	1 thread		*500 threads*		*1,000 threads*	
Number of Elements	*Linked List (time in ms)*	*Binary Tree (time in ms)*	*Linked List (time in ms)*	*Binary Tree (time in ms)*	*Linked List (time in ms)*	*Binary Tree (time in ms)*
3,000	33.29	4.30	91.09	57.09	149.50	118.50
6,000	140.19	6.30	259.79	52.70	350.70	113.90
9,000	333.39	7.70	696.20	56.29	830.29	115.09
12,000	669.70	10.40	1,514.50	55.00	1,753.09	117.80
15,000	1,204.30	10.70	2,537.19	57.79	2,830.39	120.40
18,000	1,769.90	10.80	4,223.70	57.50	4,061.19	118.40
21,000	2,658.10	14.70	5,943.70	62.79	5,982.00	121.40
24,000	3,519.69	17.40	7,893.79	58.90	8,320.80	117.69
27,000	4,378.89	21.50	10,006.20	64.80	10,683.00	117.09
30,000	5,581.79	20.50	12,474.79	65.59	13,129.90	121.69
33,000	6,894.29	24.50	15,359.90	64.59	16,982.69	122.40
36,000	8,251.00	23.40	18,561.80	64.40	20,123.10	121.40
39,000	10,004.90	26.29	21,591.09	69.40	24,458.40	124.80
42,000	11,835.29	28.40	24,927.69	78.69	27,840.50	126.80
45,000	13,412.59	30.29	27,865.59	78.19	31,217.00	121.00
48,000	15,480.90	27.10	31,753.80	92.09	35,349.89	122.69
51,000	17,599.80	31.50	36,759.80	99.00	39,156.89	119.80
54,000	19,807.90	36.59	41,7510	98.19	45,731.30	124.50
57,000	23,133.40	34.09	47,413.40	101.19	51,492.10	125.80
60,000	25,622.84	39.59	51,940.00	93.80	55,478.70	127.59

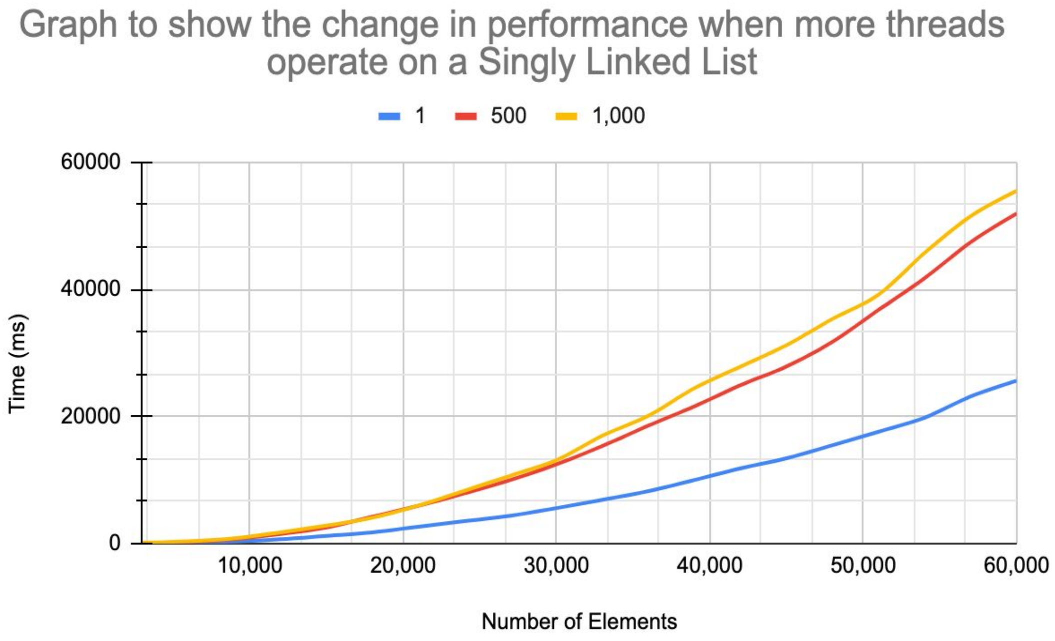
Analyzing Performance of Data structure without coarse-grained locking:



The above graph shows the running time of both the singly linked list in red and the binary tree in blue when one thread operates on them. As expected, the time complexity of the linked list increases significantly as the input size increases, while the time complexity of the binary tree does not seem to increase at all (not visible due to the large vertical scale on the graph). This proves the given hypothesis. When the input size increased from 21,000 elements to 42,000 elements, the running time of the linked list increased from 2,658.1ms to 11,835.29 ms, while the running time of the binary tree only marginally increased from 14.7ms to 28.4ms. This is because the linked list has only one access point, the head, and needs to traverse in the worst case n elements to perform an operation, thus its time complexity is $\Theta(n)$. While the binary tree also only has one access point, the root, it only needs to traverse the height of the tree h to perform an operation where the height of the tree is usually $\lg n$, thus its time complexity is $\Theta(h)$. This also explains why up until about 10,000 elements the singly linked list performed reasonably well, as the input size was smaller.

Analyzing Performance of Data structure with coarse-grained locking:





The above graphs show the running time of the singly linked list and the running time of a binary tree, respectively, when 1, 500 and 1,000 threads operate on them. As expected, both data structures perform far better when less threads operate on them compared to multiple threads. For example, when the linked list has 30,000 elements, operations with 1 thread took 5,581.79 ms while 1,000 threads took 13,129.9 ms. Although both 1 thread and 1,000 threads execute the operations sequentially (due to coarse-grained locking), they do not have the same running time because 1,000 threads need to compete with each other to hold the lock, which decreases performance. Furthermore, the graphs show that the operations with 500 perform better than operations with 1,000. When the binary tree stored 30,000 elements, the running time of 500 threads was 65.59 ms while the running time of 1,000 threads was 121.69 ms, a difference of 56.1 ms (almost double the time). Similarly, when the linked list stored 30,000 elements, execution with 1,000 threads took 655.11 ms longer than it did with 500 threads. The time difference between the execution times of the different threads in the linked list is larger than that of the binary tree, because more threads had to compete for the lock. This is because operations on the linked list take longer, i.e. the thread is in the critical section for longer causing a halt in the execution of many other threads, meaning that more threads compete to hold the lock. Operations on the binary tree take far less time, meaning that there are less spinning threads and thus less contention. Regardless, the number of threads had a far bigger impact on the performance of the binary tree than it did on the linked list, as the time difference is proportional to the running time of the binary tree as is shown in the graph.

Reliability of the Data:
Although each data entry is the average of 10 runs, running the same data structure with the same amounts of threads and elements will never result in the same average and sometimes fluctuate greatly. This may be because the running time of each operation depends on the location the element is added or removed in the data structure, resulting in best case, average case and worst case running time as discussed in the hypothesis. This explains the fluctuations in the running times of the binary tree and may explain why the running times of the linked list seem to take the form of a quadratic function rather than a linear one.

Conclusion:
To conclude, the report has highlighted that the binary tree performs better than the linked list because it has a faster running time. Furthermore, it has shown that the performance of both data structures decreases when the number of threads operating on them increases, since coarse-grained locking does not allow concurrent execution and contention between the threads to hold the lock causes performance to decrease.

References:
[1] Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein Clifford. *Introduction to Algorithms*. Cambridge, Mass: MIT Press, 2001. Print.