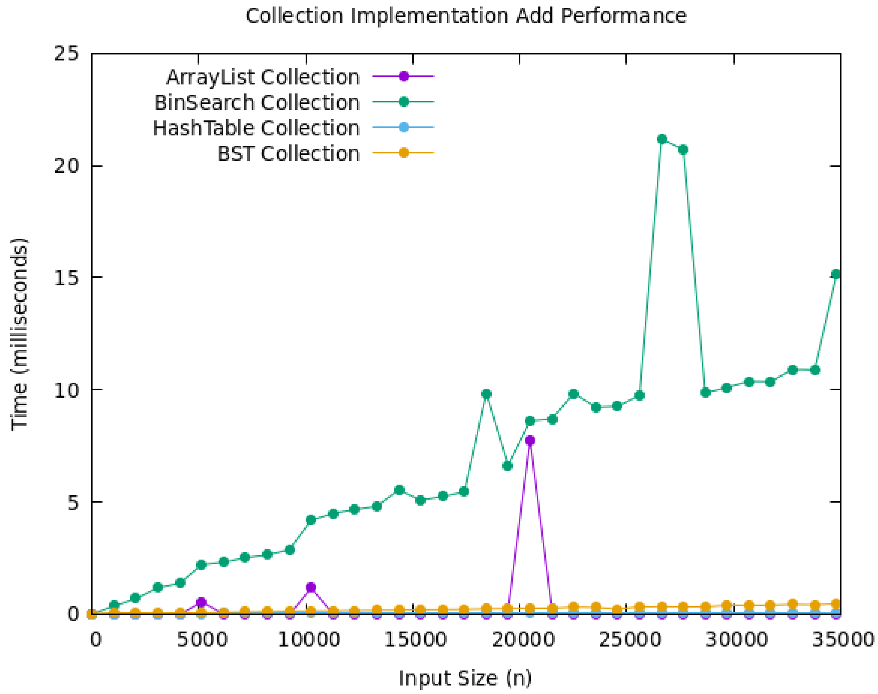
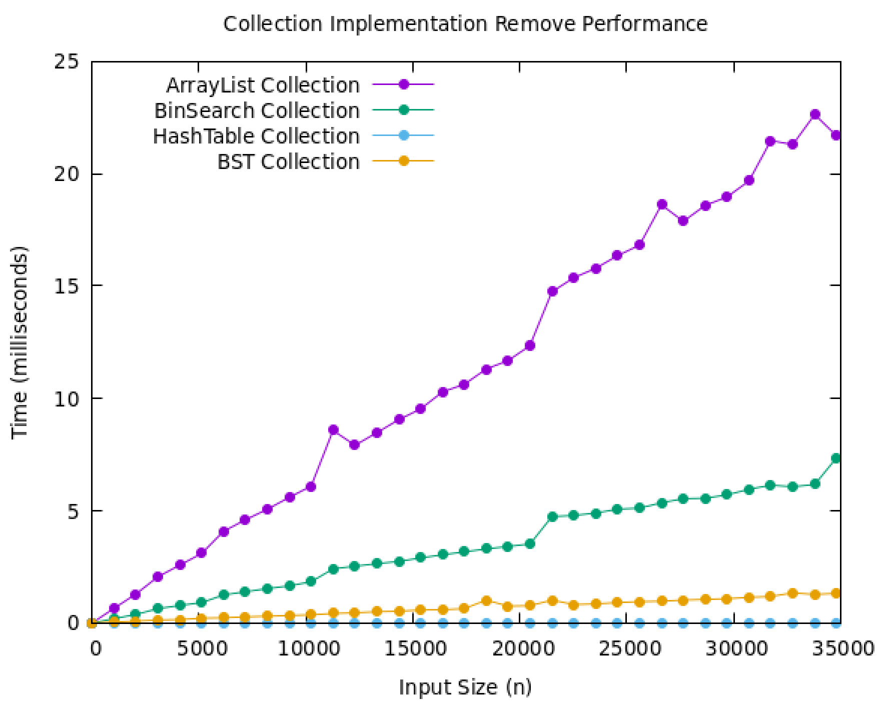
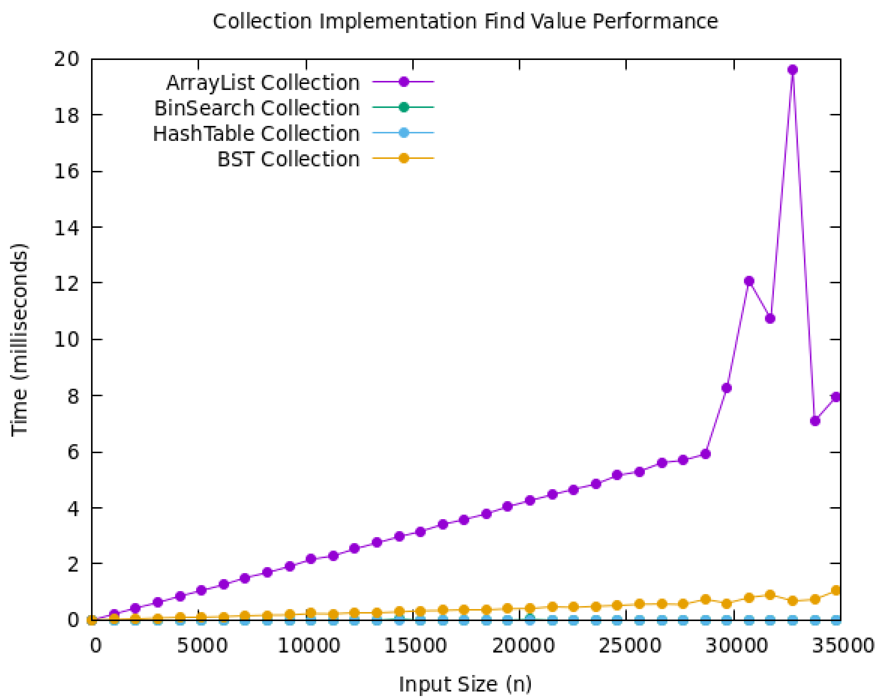


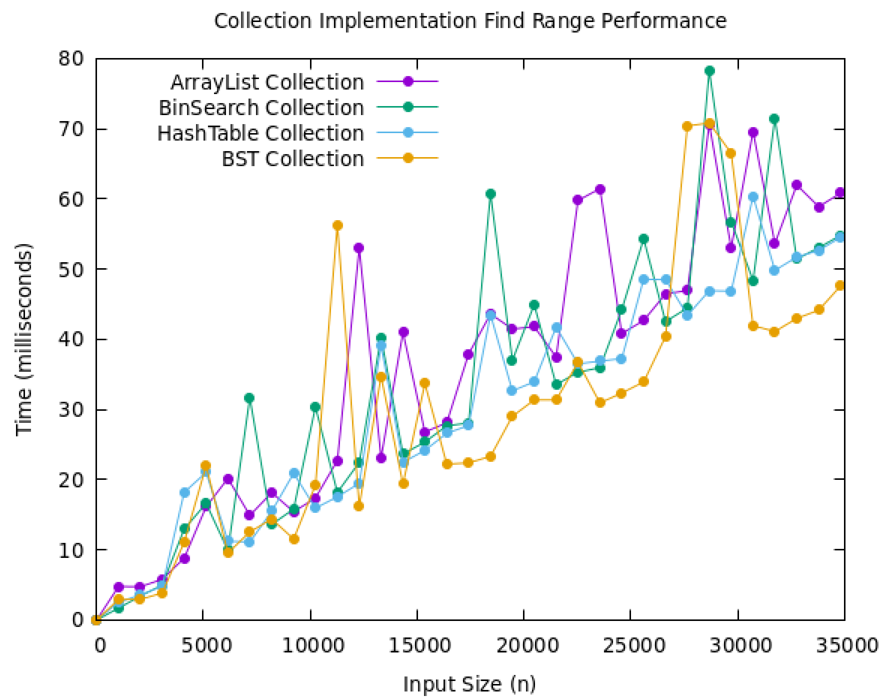
Binary search tree takes the shortest time because we just need to traverse and add to the end of a node in its suitable position. There is no required resizing, and we don't need to find the index to add, instead we just traverse one part of the list, either left or right, do not need to traverse the whole list



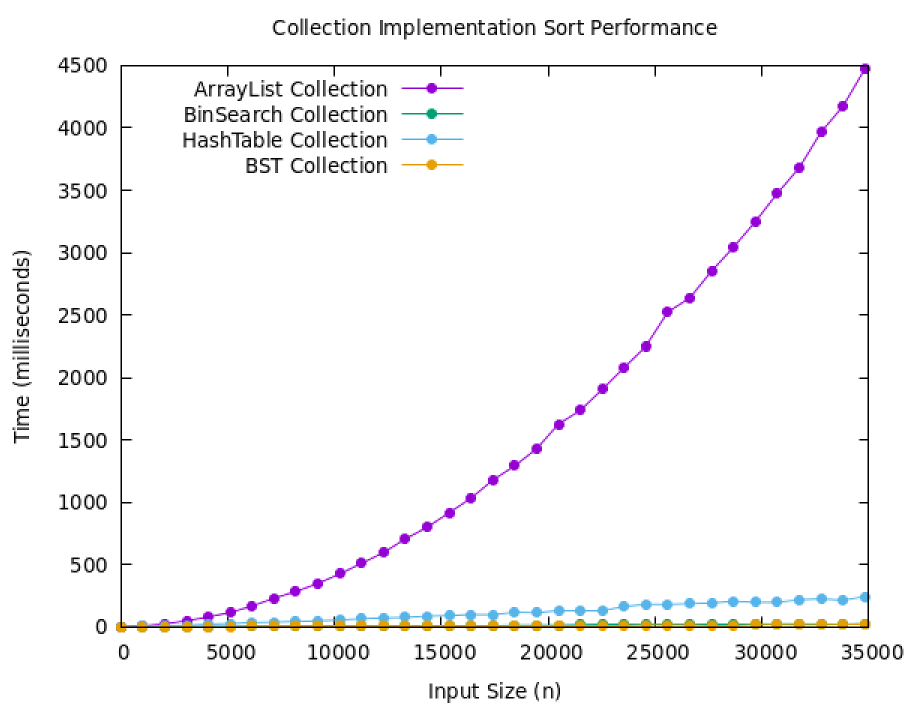
Binary search tree takes the second shortest time. In array list and binary search collection, we are traversing the whole list to remove. In hash table collection, it takes an even shorter time because we can just hash to the index without traversing. BInary search tree while have to traverse through the list does not need to traverse the whole thing because we know its either left to right, one path only



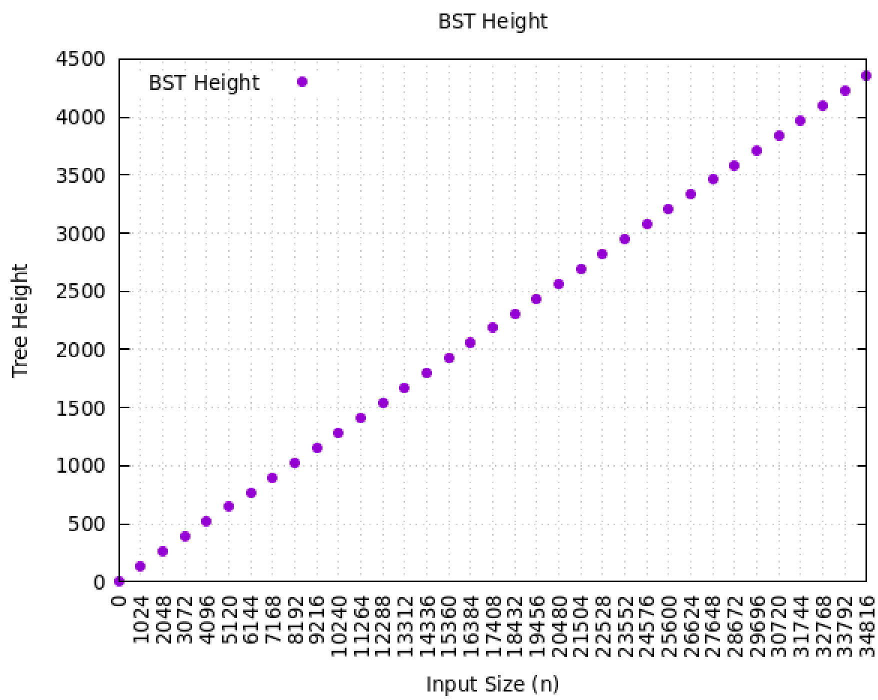
For find, binary search tree takes a longer time than hash table collection and binary search collection because we don't have an array that we can index directly. Instead we compare and traverse eiter to the left or right to find. Therefore, our time complexity is O(Logn)



For find range, binary search tree takes the shortest time. This is because even though we have to compare the nodes, we know for instance that if a key is lower than the left bound k1 of the range, we don't need to search all nodes to its left because they are smaller than the node and won't be In the range. There is some order to binary search tree so we do not need to traverse and compare each node



For sorting binary search tree is one of the fastest. This is because, compare to array list it is already sorted as all nodes on the any node left is smaller than any node and all to the right is larger. Therefore, we only need an inorder traversal to get the sorted list. In comparison, in array list, element is placed at the end and in hash table collection, it is not hashed to be surely sorted.



Height increases as the number of nodes increases.

Operations	Collection Implementation				
	Array List	LInked LIst	Sorted array	Hash Table	Binary search tree
Add	Constant time o(1) except when array is full O(1) by amortization prove	O(1) because tail pointer can be access to and point to new node immediately No worst case because there is tail pointer	Adding is O(N). This is because we have to shift all the gelements that are after the index of adding to the left. Worst case scenario is adding at the front because all	Adding to the front is always constant time O(1) except when array is full but it is still O(1) by amortization prove	Adding to the end of the node by average is O(log n) because we only traverse a path not the whole tree. However, worst case can be O(n) like adding to end of a linked list if the

			elements have to shift to the right. This yields O(N)		nodes are added in ascending or descending order
Remove	O(N) because one loop to find the index whose key matches the parameter key and a second loop needed to transverse and copy the value to new array. The worst case is when have to remove at front because all the remaining array element have to shift by one	O(N^2) because one outer loop to find the index whose key matches the parameter key and an inner loop needed to transverse to pointer whose neighbor node will be deleted. Slower because of caching. The worst case is when have to remove at end because have to transverse to end to set tail to second last node	For removing, it consist of two parts, finding the index and removing. Finding the index to remove is only O(log n) because binary search is used. However, removing will still require O(N) at worst case scenario if we remove at front, because all elements after that have to be shifted to the left	Removing is O(1) because finding the index is only O(1) due to the hash function. Then, we have to loop through the chains that have collision. However, worst case O(N)is when all the nodes on the same array index because this will make a linked list	Removing is O(log n) average because we need to traverse through a path and shift the lower nodes either to the deleted nde left or right. Worst case is deleting a node from a ascending or descending order which will be O(N) because we need to traverse whole list to find node and delete
Find value	O(N) because one loop to find the index at which the key matches and the parameter key. Indexing is constant. Memory is also contiguously allocated so	O(N^2) because one outer loop to find the index at which the key matches and the parameter key and an inner loop to transverse the link. The worst case is when	Find value is just O(log n) unless the value is not in the list, because then we still have to split all the list to one element which takes O(N)	Finding value is O(1) because it uses hash function to get to the index and loop through the list on the array index to get to the node to be removed.	Finding value is also O(log n) except if the first node is the node searched. Other than that, it takes O(log n) to search the left or right path. Worst case is O(N) if our nodes are in ascending or

	easy retrieval. The worst case is when value does not exist or at the end of list because still to have to transverse to end in outer loop	value does not exist or at the end because still to have to transverse to end		Worst case is O(N) when all the nodes on the same array index and we remove the last element in the node	descending order because it will be like deleting at the end of a linked list where you have to traverse till the end.
Find range	O(N) because one loop to find the index at which the key matches and the parameter key. Indexing is constant. Memory is also contiguously allocated so easy retrieval. The worst case is when all the elements in array is in the range is the worst case	O(N^2) because one outer loop to find the index at which the key matches and the parameter key and an inner loop to transverse the link. Memory is not contiguously allocated so hard to allocate even if the next key is also in range. The worst case is when all the elements in linked list is in the range And memory allocated far apart from each other is the worst case	For finding range, there are two parts. The first is finding the first index to start, which is key 1.This would take binary search O(logn) average case, to find the index. After that, we just have to traverse the list till we reach the second key. In worst case scenario, the first key is not found which would mean we take O(N) time to find the first key and the second key is larger than the last element. This would mean 0(2n)	For finding range, there are two parts. The first is finding the first index to start, which is key 1.This would take hash table O(1) average case, to find the index. After that, we just have to traverse the list till we reach the second key. In worst case scenario, the first key is not found which would mean we take O(N) time to find the first key and the second key is larger than the last element.	For finding range, we find the node which is between the k1 and k2 bound and if a node is smaller than k1, we don't go to its left and if a node is larger than k2 we don't go to the right. Therefore, instead of traversing the whole list, as it is sorted in a certain way, it is only O(log n). Worst case is if all nodes are in descending order, because then we need to traverse the whole tree which will be O(N)

Sort	O(nl ogn) because of quick sort time complexity Worst case: Array is sorted or in reverse sorted order o(n^2)	O(n logn) because of merge sort time complexity Consistent for best and worst o(n logn)	For sorting, there is no best or worst case, since the keys are always sorted. So O(1)	Best case is O(1) if the hash function inserts the elements in a sorted fashion while it will be O(nl ogn) at worst case if it is placed in reversed order and we must use quick sort O (N^2)	For sorting, the average time complexity is O(N) because you are traversing the whole tree in an in order traversal. The worst case is if it is in reverse sorted because it will be like sorting a linked list which we need to use log n which will be O(n log n)