EXERCISE 1 report

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Best case | | Worst case | | Average case | |
|  | Insert | Search | Insert | Search | Insert | Search |
| Binary Tree | O(1) | O(1) | O(n) | O(n) | O(nlogn) | O(nlogn) |
| LinkedList | O(1) | O(1) | O(1) | O(n) | O(1) | O(n) |
| HashTable | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) |

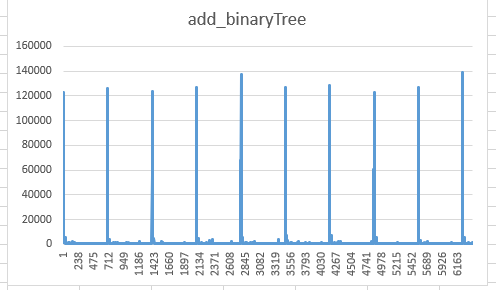
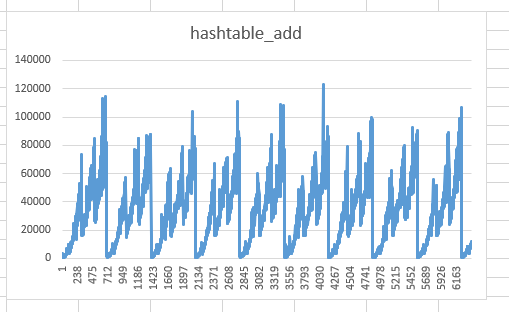
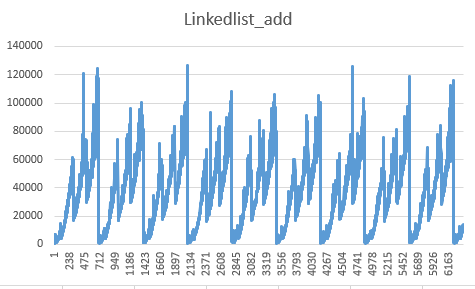
From all the efficiency I analysis from the table, we can get that Hashtable gives us best performance, and binary tree is second , the linkedlist is the worst in the set implementation.

1. Add part:

Hashtable has the best performance, since it contains index and the system can access to the index directly instead of searching from the head or root like linkedlist and binary tree. In the worst case the Hashtable has many collisions which will make the hashtable an linkedlist.

Binary tree is the second since it will abandon half data when it add to the tree. So the time complexity is O(nlogn);

Linkedlist is the worst according to my real data. I think I did the wrong implementation. In fact, the linkedlist should be the fastest in adding operation since node in the head instead of move to the tail.



1. Search part:

Hashtable is the best too in this section since it contains index, the time to search is O(1);

Binary tree is the second since it will abandon half data when it search in the tree, the same as add operation. So the time complexity is O(nlogn);

Linkedlist is the worst according to my real data.That’s true, since every time you need to search from head and to the next node. So the time complexity is O(n)

