Array

In my array set I set a size of 10 initially, and count the array items as they are inserted. When my array fills up, the array size is doubled, and then that doubled size is added to the array. The array sorted, so a new item is first compared with every item until it is not less than the item it is being compared with. It then checks if it is equal to that item. If it is not, it is inserted in that array spot, and array items larger than it must be shuffled to the right, starting with the right most item. A best case scenario for this algorithm is if there are no items yet in the array, as that would be O(1), only one insert is required. Otherwise the cost is O(n) as each item is either compared with the new item or shuffled to the right.

My algorithm for deletion is similar to insertion. The delete item is compared with each item from the start, and when it reaches an item that it is no longer less than it stops. It checks if it is equal to that item. If it is, it deletes it and shuffles all items larger than it to the left. Again the cost is O(n) as each item in the array is either being compared with the delete item or being shuffled.

The above two algorithms are brute force algorithms, where each item is checked and compared or shuffled.

My search algorithm is a binary search. I created a recursive function which splits the array in two every iteration, until it finds where the search item should lie, and then it checks if it is there. This creates a cost of O(log n), which is substantially less than a brute force algorithm would be as was evident in my testing.

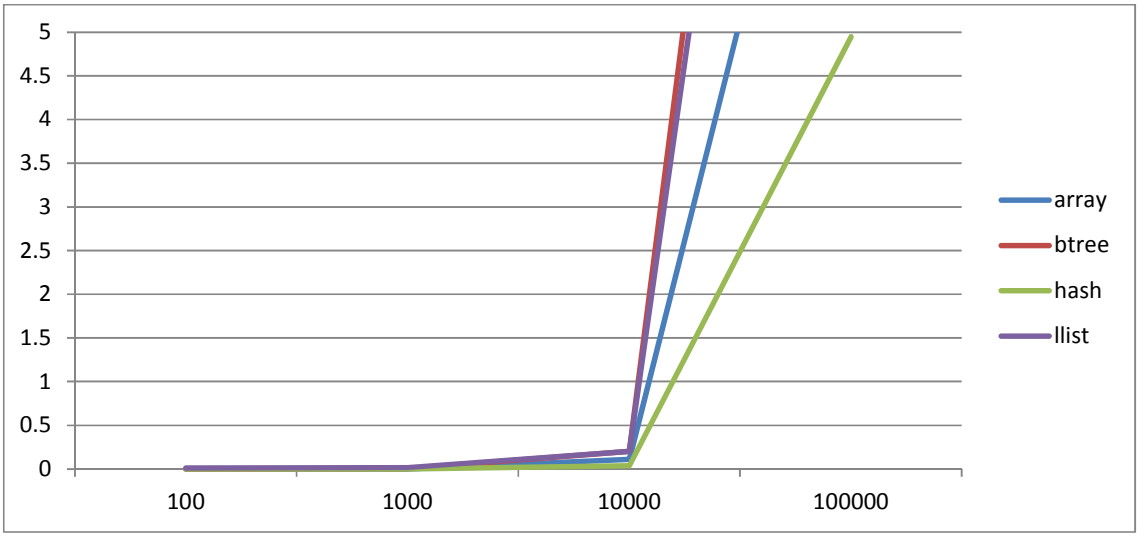
Hash

For my hash set I created an array of 10. I also created a hash that sums the digits of the number being inserted, then takes the modulus of 10 to determine which bucket of the array to insert. Each bucket contains a linked list. The efficiency of the hash set relies on the numbers being inserted being distributed more or less evenly between the array buckets. It also increases in efficiency with more buckets. A worst case scenario for a hash set is if all numbers inserted are put in the same bucket. That would cause the hash set to become a linked list. Best case would be if the insertions are evenly distributed, in which case the cost of insertions, searches and deletions would be divided by the number of buckets in the array. My linked lists within the hash set are not in sort order which decreases the insertion cost, but increases the deletion and search costs.

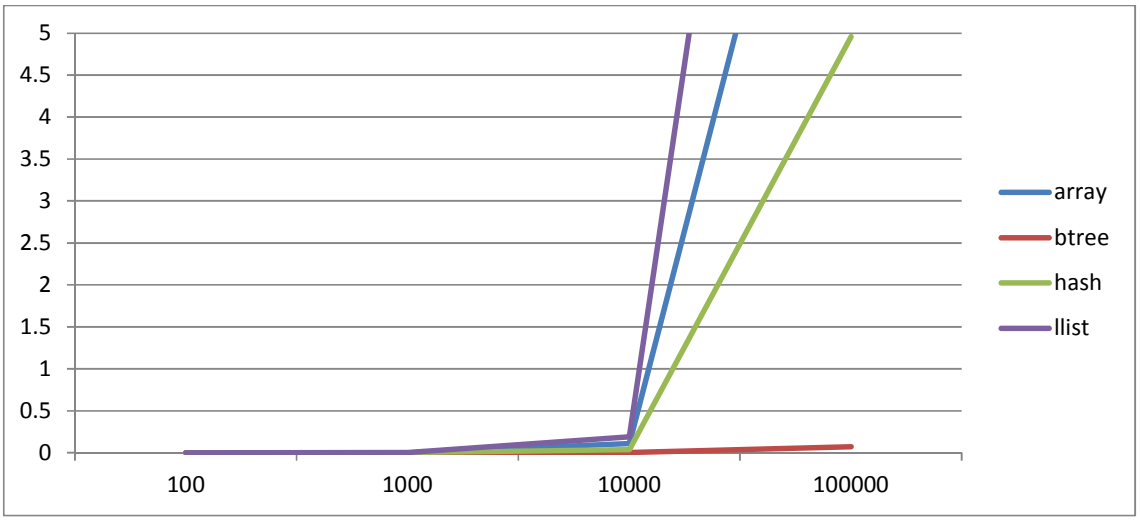
Binary Tree

My binary tree set keeps track of the first insertion as the parent node, and each node in turn keeps track of its own parent, right child and/or left child. This binary tree relies on the first insertion being more or less an median of insertions into the tree. In this case, the search, insertion and deletion costs are greatly decreased and are O(log n). However, because my tree does not actually keep track and guarantee that it is balanced, it is open for a worst case scenario which would be if the insertions are in sort order. In such a case, as I saw in my testing, the binary tree works the same as a linked list and the cost is O(n).

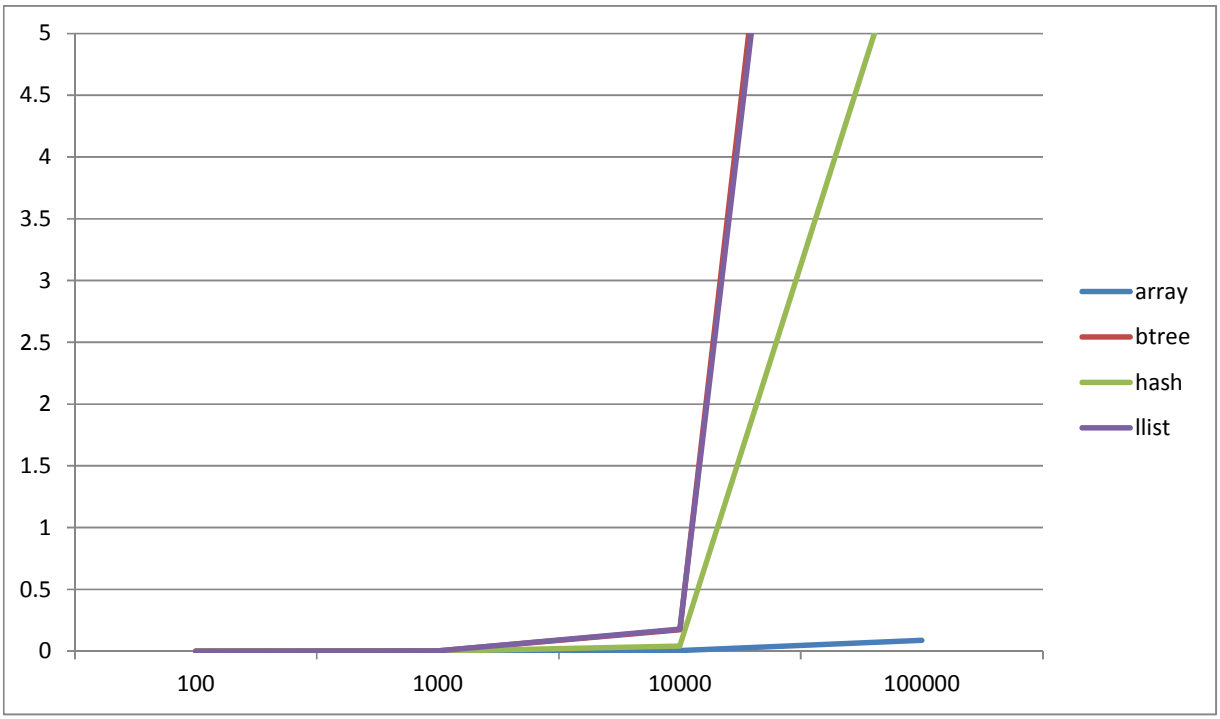
This graph shows inserting sorted data ranging from 100 to 100000



This graph shows inserting unsorted data ranging from 100 to 100000. The only major distinction from the above graph of sorted data is the binary tree, which acts like a linked list when the data inserted is already sorted. It would behave the same way if the data were sorted in descending order.



This graph shows a search with data that was inserted in sort order. Again, the binary tree is the same as a linked list and loses all benefits when it is inserted in sort order. The array search is very quick because of the binary search algorithm which is O(log n).



This graph shows deletions in different sets ranging from 100 to 100000