(Sol) Data structure recap

Important!

- **Many possible solutions:** the solution below is only one example solution. As discussed in the class, there are many ways of implementing certain operations for the same data structure. As long as your answer is consistent, your answer is valid. (i.e., if you assumed that the size of unsorted array increases by 1 when it needs more space for insert, you need to use the same assumption for all three tables.)
- No average/amortized time complexity for now, as explained in the class. I'm not offering the solution so you won't be tempted to study it.

Fill in the table below with the worst-case time complexity, and add explanation below. [unsorted array, insert] is provided as an example. Assume that there are n items in the data structure already. All data structures start with size of n items. For the hash-based data structures, assume that the the size of the data structure doubles when the load factor is met. The load factor is 75%.
[ETA] The range is the key range. For example, if the key is the student ID and the value is the student records, then the range is the range of the student IDs.

The degree of B*-tree is m and the upper bound - the lower bound in the search by range is r. For simplicity, assume $r << n$.

	unsorted array	sorted array	binary search tree	B*-tree	HashMap/HashSet/HashTable
insert	$O(n)$	$O(n)$	$O(n)$	$O(\log_m n)$	$O(n)$
delete	$O(n)$	$O(n)$	$O(n)$	$O(\log_m n)$	$O(n)$
search by key (exact match)	$O(n)$	$O(\lg n)$	$O(n)$	$O(\log_m n)$	$\Theta(1)$
search by range	$\Theta(n)$	$O(\lg n + r)$	$O(n)$	$O(r \log_m n)$	$\Theta(r)$

▼ Unsorted array

- insert: $O(n)$. When the array is full, we need to get a new array and copy all n items from the old to the new array before placing the new item at the end.
- delete: $O(n)$. When the first element in the array is deleted, then $n-1$ elements need to be shifted to the left, so $O(n)$.
- search by key: $O(n)$. In the worst case every element needs to be checked.
- search by range: $\Theta(n)$. Every element needs to be checked against the range.

▼ Sorted array

- insert: $O(n)$. The new element could be the smallest and n elements would have to be shifted to the right
- delete: $O(n)$. The deleted element could be the smallest and $n-1$ elements would have to be shifted to the left
- search by key: $O(\lg n)$ using binary search

- search by range: $O(\lg n + r)$ using binary search on the lower bound and then scan linearly.

▼ Binary search tree (balancing not required, so the height is $O(n)$)

- insert: $O(n)$. If the tree is not balanced, insertion could happen at the leaf.
- delete: $O(n)$. To delete, we need to find that node first, and it may take $O(n)$.
- search by key: $O(n)$ when the tree height is $O(n)$.
- search by range: $O(n)$ traversing the whole tree

▼ B*-tree (balanced search tree, height $O(\log_m n)$)

- insert: $O(\log_m n)$. insertion could cause splitting nodes at every level
- delete: $O(\log_m n)$. deletion could cause merging nodes at every level
- search by key: $O(\log_m n)$. target could be in the leaf node
- search by range: depends on the variants. B+ trees would take $O(\log_m n)$ and linear scan of leaf nodes $O(r)$. Other variants would take $O(r \times \log_m n)$.

▼ Hash* (assuming one item per bucket)

- insert: $O(n)$. insertion could cause the buckets to double and existing n items would need to be rehashed.
- delete: $O(n)$. deletion could trigger shrinking the buckets and existing n items would need to be rehashed.
- search by key: $\Theta(1)$. one hash function and index lookup.
- search by range: $\Theta(r)$. needs to look up for every value in the range.

Fill in the table below with the worst-case space complexity under the same assumptions.

	unsorted array	sorted array	binary search tree	B*-tree	HashMap/HashSet/HashTable
space complexity	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

▼ Unsorted array and sorted array

- The worst case is insert when the array is full. we need to get a new array of size $2n$, so the space complexity is $\Theta(n)$.

▼ Binary search tree

- For n items, the space required is n nodes, so $O(n)$.

▼ B*-tree

- For n items, the maximum number of nodes is $2n/m$, so $O(n)$.

▼ Hash*

- The number of buckets should be large enough to avoid collision as much as possible, so typically the number of buckets is $1.25 * n$, $O(n)$.

▼ Unsorted array

- insert: $O(1)$. To insert $2n$ items, the total time it takes is $O((n+n+n)/2n)$, so the amortized time complexity is $O(1)$. Similarly, inserting another $2n$ items after this takes $O((2n+2n)/2n)$, so the amortized time complexity is still $O(1)$.