Dynamic Vector Analysis

Accessing Element
Runtime: O(1)
Space: O(1)
Explanation: Because elements in a vector are stored next to each other in memory, accessing an element given an index can be done in constant time and doesn't require additional memory allocation.

Removing Element
Runtime: O(n)
Space: O(n)
Explanation: While removing an element from a vector could be accomplished in constant time, that is only if the element happens to be located at the end of the vector. When analyzing the worst case, we must assume that we are removing from the beginning, requiring n elements to be shifted over into the correct place to keep the vector contiguous. Once the vector has shifted, memory has to be de-allocated, requiring linear space.

Inserting Element
Runtime: O(n)
Space: O(n)
Explanation: Just like with removing an element, adding an element at the end can be accomplished in constant time, however, inserting an element at the beginning of a dynamic vector requires shifting and resizing all n elements of the vector.

Searching Element
Runtime: O(n)
Space: O(1)
Explanation: Searching an element takes linear time as it requires stopping at each index and making a comparison. However, if a vector is sorted, we can apply binary search to perform searches in logarithmic time at the expense of an initial n log n sort time. No additional space is necessary to make the comparisons for searching.

Resizing
Runtime: O(n)
Space: O(n)
Explanation: Resizing a dynamic vector is linear as it requires the vector and its contents to be moved to a different location in memory.

Hash Map Analysis

Accessing Element
Runtime: O(1)
Space: O(1)
Explanation: Hash maps provide constant access as all the program has to do is run the hashing function on the key that is passed in. This does not require extra memory allocation.

Removing Element
Runtime: O(1)
Space: O(1)
Explanation: Removing an element from a hashmap is constant time, as once you locate it using the hash value, you can delete it.

Inserting Element
Runtime: O(1)
Space: O(1)
Explanation: Insertion is constant for the same reason removing an element is constant. Once you run the hash function to get the insertion index, you can insert it immediately with no additional memory allocation requirements.

Searching Element
Runtime: O(1)
Space: O(1)
Explanation: Searching a hashmap is done in linear time as once the key is hashed, it will lead to the index where the value is stored in the hashmap.

Resizing
Runtime: O(n)
Space: O(n)
Explanation: When a hashmap is resized, it requires a new array to be allocated in memory, along with the rehashing of all the contents within the previous hashmap so everything can be moved over properly. Since well designed hashmaps only occasionally have to be resized, the expense incurred by the resize operation is mostly negated.

Hashmap Note: For a hashmap to perform effective constant time operations, it is imperative that the hashing function distribute the elements evenly throughout the underlying array. If there are double mappings or not enough space in the array, we run into collisions which force the worst case runtime of many hashmap operations up to linear time.

Binary Tree Analysis

Accessing Element
Runtime: Balanced Tree: O(log n), Unbalanced Tree: O(n)
Space: O(1)
Explanation: When accessing an element in a balanced binary tree, each node helps you rule out half of the problem space leading to a logarithmic access time.

Removing Element
Runtime: Balanced Tree: O(log n), Unbalanced Tree: O(n)
Space: O(1)
Explanation: Removing an element in a balanced binary tree requires finding the node that needs to be removed and deleting it from the tree meaning that the amount of time it takes to access the node is the amount of time it takes to perform the remove operation.

Inserting Element
Runtime: Balanced Tree: O(log n), Unbalanced Tree: O(n)
Space: O(1)
Explanation: Once the element finds its proper location in the tree structure, it can be inserted at no additional memory costs.

Traversing
Runtime: O(n)
Space: O(n)
Explanation: Resizing a dynamic vector is linear as it requires the vector and its contents to be moved to a different location in memory.

My Overall Recommendation For ABCU:

My recommendation to ABCU would be to identify the specific business needs that require their system to perform the most operations, and subsequently adopt a data structure that is optimized to handle those tasks. For example, in the case where the frequent need is to search for courses in the system, I would strongly recommend a Hash Map due to its exceptional O(1) lookup time. However, if the objective is to efficiently tabulate all course information in alphanumerical order, a binary search tree is the optimal solution as it specializes in maintaining items in order. The most effective approach for ABCU would be to combine these data structures, leveraging their respective strengths to achieve optimal benefits.