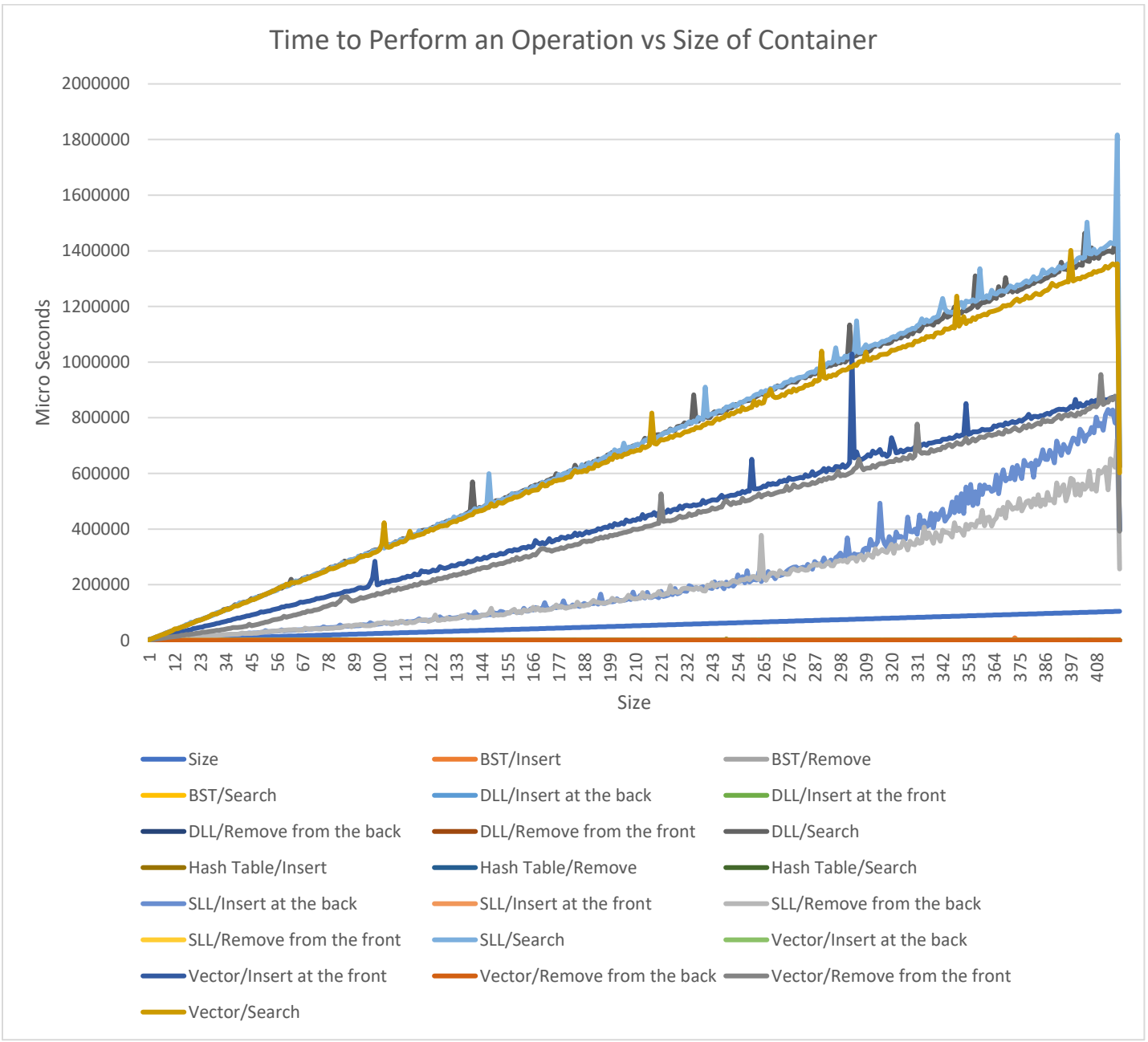
Armanul Ambia

Analysis Report – Final Project

This project involved analyzing the time complexities of multiple different data structures in terms of distinct operations and comparing their efficiency. Particularly, this included analysis on the C++ Standard Template Library (STL) vector (vector), doubly linked list (list), singly linked list (forward_list), binary tree (map), and Hash Table (unordered_map).

Vectors, doubly linked lists, and singly linked lists were compared among five operations: inserting at the back, inserting at the front, removing from the back, removing from the front, and searching for an element (worst case assumed and a nonexistent value used). Binary Trees and Hash Tables were compared among three operations: inserting an item, removing an item, and searching for an item.

The results of running the project are shown in the graph below. This graph has the time in microseconds on the y-axis with the size of the container in terms of the number of elements along the x-axis. The different lines represent the various operations as well as the type of data structure that was tested.



In addition to the above graph, the program also aggregated times for all the operations on a single data set, as shown below.

	Vector	Doubly Linked List	Singly Linked List	Binary Search Tree	Hash Table	Total
Total Time (seconds)	637.384	293.792	479.082	.535571	.293831	1411.09

Inserting at the back:

The first operation conducted is inserting at the back of a structure, and this was completed on vector, doubly linked list, and singly linked list. Inserting at the end of the STL vector class is completed with the push_back function and has an efficiency of O(1) amortized. This means that usually, it is constant except for the single case where the underlying array needs resizing. Essentially this means that adding to the back of the vector will add to the next available slot in the underlying array. However, if the array is full, it needs resizing and has a greater complexity of O(n). This occurs since the values need to be copied over to the new array. This process is only completed once the underlying array fills up, so during the majority of cases, push_back will insert it in the back with constant time. Hence why the actual complexity is O(1) amortized. This is reflected by the values in the table. Most values are less than 10 microseconds regardless of the size of the container. Still, occasionally the values will increase to close to the size of the container, showing the resizing of the array.

Inserting at the back of a doubly linked list (STL list) is accomplished using push back as well. A doubly linked list includes a pointer to the tail, and each node has a next pointer. Adding to the back will simply require the tail to point to the new node and then replace the tail pointer with a pointer to the new node. This is accomplished in constant time, and push_back has an efficiency of O(1). The table shows this efficiency since most of the values for the doubly linked list insert at the back are 0, indicating constant time.

Inserting at the back of a singly linked list (STL forward_list) has to be completed manually as the forward_list has no tail pointer and no random access iterator. The forward_list is first looped through, and an iterator is initialized to the spot before_begin then updated during the loop. This iterator should point to the last item in the list at the end. After this is completed, the insert_after function is used by passing in the position so that the new item is added to the back of the singly linked list. Efficiency suffers as a result. Although insert_after is constant using an iterator position, looping through the forward_list to find the position has O(n) complexity. As the size grows, this method has a complexity of O(n) and is further slowed by the limitations of the machine. In the values recorded in the table, insert at the back for STL forward_list for large sizes has time complexities as high as 8 * O(n).

One example of when inserting at the back becomes essential is when tracking a student's exam scores. The exam scores are held in a data structure, and new exams are inserted into the back of the structure. The best option to use in this case would be the doubly linked list. Doubly linked list will take up less contiguous memory than the vector and will always have an efficiency of O(1). This makes it more efficient than a vector with O(1) amortized or forward_list with O(n).

Inserting in the Front:

The second operation conducted is inserting in the front of a structure, and this was completed on vector, doubly linked list, and singly linked list. Inserting at the front of an STL vector is done using the insert method. This insert method takes in an iterator to a position and the item to insert. In this case, cbegin is used as the iterator position, and this results in the worst-case efficiency. The insert method for vector has an efficiency of constant plus linear in the distance between position and the end. Because of this, adding to the front of the vector using insert has a complexity of O(n). Inserting a value at the front requires every other value in the underlying array to be shifted.

Inserting at the front of a doubly linked list is completed using the push_front method. Doubly linked list has a head pointer to the starting element. To add to the front, a new node is created that points to the head, and then this node is set to head. This operation is done in constant time, so push_front has a complexity of O(1). This is displayed in the table since the majority of values for the doubly linked insert at the front are 0, indicating constant time.

Inserting at the front of a singly linked list is accomplished using the push_front method as well. Similar to the doubly linked list, singly linked list has a head pointer to the starting element. To add to the front, a new node is created that points to head, and then this node is set to head. This operation is done in constant time, so push_front has a complexity of O(1). This is displayed in the table since the majority of values for the doubly linked insert at the front are 0, indicating constant time.

One example of when inserting at the front can be used is a program that tracks tax reports. Usually, the most recent year is the most crucial tax return document and is used for verification in many places. Every year when taxes are filed, the current return is pushed to the front of the data structure where it can be retrieved the fastest through a search. To implement this data structure, a vector should not be used since inserting has a time complexity of O(n). Both a doubly linked list and a singly linked list can be used as well. Since a reference to the back of the list may not be necessary and, in most cases having only a single pointer to the head is enough, a singly linked list is the best option for this data structure. Inserting at the front will have a complexity of O(1), and the structure will be efficient.

Inserting into Binary search Tree and Hash Table:

In these two containers, a regular insertion was used to test the complexity. All operations conducted on binary search trees and hash tables were far more efficient and faster than any of the other structures. Inserting into the binary tree and inserting into the hash table are done the same way syntactically, a new key is created, and the value is matched with it. This key and value pair is then added to the structure.

A binary search tree is implemented using std::map, and inserting into this has a time complexity of O(log N). This is due to the nature of the binary tree where the key is compared to the midpoint repeatedly until a spot to insert the item is found. The table depicts how the values continue to increase slowly as the size grows but does not exceed relatively small values in the hundreds indicating the O(log N) complexity.

A hash table is implemented using std::unordered_map, and inserting into this has a time complexity of O(1). This process is instantaneous since a hash table uses a specific hash function to map the key to a specific index in the data structure. Aside from the first element added, the table of values shows that the hash table's insert values are all close to 0.

An example of a program that would use constant insertion is one that holds all of a video game's character information. The character name would be a key, and all other information would be part of the value. This key, value pair approach makes it ideal for one of these containers. If the characters need to be ranked by some category such as age or power, a BST is more useful even if it has slightly higher time complexity. However, if no order is necessary and the characters simply need to be stored, then the Hash Table is a more efficient choice with constant complexity.

Removing from the back:

The third operation was removing from the back of a structure, and this was completed on vector, doubly linked list, and singly linked list. Removing from the back of an STL vector is done using the pop_back method. This method does not take any arguments and simply removes the last item in the container. The complexity of this is constant with O(1) efficiency. The data shows that this operation had a value of 0 for all measurements, aside from one indicating constant time.

Removing from the back of a doubly linked list is also done using the pop_back method. This method works very similarly to the vector pop back and also has a constant O(1) time complexity. The table also has similar data with a value of 0 for the majority of measurements in the column.

Removing from the back of a singly linked list is more complex, and there is not a direct way to accomplish this. The erase_after function is used. Two temporary variables are used, a predecessor that is an iterator to before_begin and a current that is an iterator to begin. The data structure is traversed until the predecessor is set to the node before the last node. Then, this predecessor is passed into the erase_after function. While the erase_after function is constant, traversing to find the point to erase has linear complexity, and so the overall operation has a time complexity of O(n). This is reflected by the data points as the values continue to increase as the size increases

An example of when to use the remove from back feature is when implementing a student roster program. A student roster program would take in new students as they entered the school into the front of the structure and remove them from the back as they graduated. A singly linked list should not be used since the time complexity is O(n) and would not be effective in removing the students. The choices are either a vector or a doubly linked list. Since both have the same complexities for removing from the back, a vector may be a slightly better choice since a vector also has direct access to every student with a random-access iterator. In contrast, a doubly linked list does not offer this direct access, but in terms of merely removing from the back, either can suffice.

Removing from the front:

The fourth operation was to remove an item from the front of a data structure and was tested on vector, doubly linked list, and singly linked list. Removing the first item from the vector was accomplished using the erase function. The function takes in an iterator to the element that is to be removed from the vector. The time complexity of this function is linear since the function calls the destructor once for the element removed and then calls the assignment operator to move all the elements after that position backward. In total, this requires n operations since the first item is being removed and causes the O(n) complexity. The data shows that this takes an increasing amount of time as the size increases.

Removing from the front of a doubly linked list is done with the pop_front function. The pop_front function does not take in any arguments and simply removes the first element. This process is done with O(1) complexity since it only takes one operation to move the head pointer to the next element.

Removing from the front of a singly linked list is done with the pop_front function. The pop_front function for singly linked list works the same way as the doubly linked list. It takes no arguments and works with O(1) complexity. The head pointer is updated to point to the next element. Both the singly linked list and doubly linked list remove from the front have mostly 0 values in the data table.

An example of when removing from the front may be necessary is when implementing a line for an amusement park. In this example, people joining the line would enter the back of the structure people getting on the ride would be removed from the front. This would be implemented most likely with a doubly linked list since people could enter the back and remove from the front with constant complexity. Singly linked list may have some drawbacks since adding to the back has O(n) complexity. Vector would not be used since the complexity to remove from the font is O(n)

Removing from Binary search Tree and Hash Table:

Similar to the insertion into the binary search tree and hash table, the removal is much more efficient in these two structures than the previous discussed. Both of them have the same syntactic implementation in std::map and std::unordered_map but have two different uses.

Removing from the binary search tree involves finding the key and then removing it. This is done in a similar way to the insert by constantly checking the midpoint until the key is found. This operation has an O(log N) complexity. The data values for this operation increase as the size of the container increasing showing the O(log N) complexity.

Removing from the hash table is done using the same hashing algorithm as insert and then removing the element at the index. This also has O(1) complexity. The key is checked with the hash algorithm, and then it is removed. The data values for this operation are almost all 0 regardless of size.

An example of removing elements from data structures would be a program that has a data structure holding the top 100 fastest cars in the world. This amount would change very frequently as new cars are being produced, and cars would continuously be removed and added to this structure. If the cars were ranked in order, then the BST would be the best option since it would require a higher complexity but still allow the data order to persist. If the order does not matter, then the Hash Table would be efficient and provide an O(1) constant complexity for removing cars and adding cars.

Searching for an Element:

The final operation conducted was searching for an element, and this was done on vector, doubly linked list, and singly linked list. All elements were searched for using a target key that was nonexistent to test the worst-case complexity. Searching for an element in all three of the structures is done using a linear search through the structure. The vector is looped through until the element is found, and then it is returned. If the data structure is empty, then this was instantaneous, but aside from this case, the function had a complexity of O(n) for all three search methods. This is reflected in the values for vector, doubly linked list, and singly linked list search as all of them increase with the size of the container.

An implementation of this linear search feature would be a program that holds all of the times for a cross country time. Searching is an important feature of any sports database, and the program would need to be able to find the athlete based on keys such as their name or personal record. The best option to use would be a doubly linked list. All of the structures discussed can search with O(n) complexity, but overall, adding and removing taken into account makes the doubly linked list most efficient. Furthermore, the vector would require a contiguous block of memory that can be a problem, especially if the database becomes large. The singly linked list would have limitations to adding and removing from the back.

Searching for an Element in a Binary Search Tree and Hash Table:

The search was also applied to the binary search tree and hash table using the specific find methods found in std::map and std::unordered map. Searching for an element in the BST has a time complexity of O(log N) and works the same as the insert and remove in terms of finding the object by cutting the data in half repeatedly. The table shows the increasing time of the data values as the size increases.

The hash table search uses the same algorithm as insert and remove to find the hash value and then returns the key that corresponds. This process is also completed in constant O(1) complexity with data values of 0 or close to 0 regardless of the size of the containers.

If search does not need to be computed linearly, then both a binary search tree and hash table are far superior alternatives to the previous three data structures. The time complexity is far superior to the linear searches, and these have added benefits. For example, by using a BST, a network interface can be implemented that shows all the routers and networks on an area. Based on the internet protocols and IPs, the BST can find the fastest network routes between two routers by conducting a binary search.

Additionally, an implementation that makes use of hash searching is a password manager for many companies and databases. Passwords in many cases need to be encrypted, and so when a user types in a password, it is encrypted character by character using a hashing algorithm, and then this hash is stored in the computer. Hashing has security benefits since even if the company data is compromised, the hacker can only see the password hashes and not the actual passwords themselves. Additionally, this makes it more secure for the user and gives an O(1) or instantaneous log in time.