747993_SHAANREHSI_A4

Setup:

For this experiment, I will be use the following data structures:

- std::list:
- std::vector:
- std::set:

Each data structure will be tested with three sets of elements, (100K, 200K and 300K) to observe their performance characteristics as the sequence size grows. I will then test out the data structures again but preallocate the list into the elements to see if their performance is affected.

Hypothesis: Based on the characteristics of the data structures I predict that the performance of the data structures will vary during the insertion and removal processes. Specifically, I expect that std::list may outperform std::vector during insertions because it uses linked-lists, while std::vector may be more efficient during removals because of its memory. I expect that std::set, being used as a binary search tree, will show reasonably good performance for both insertions and removals.

I also expect that the complementary experiment, where I will preallocate list elements, may reveal insights into the impact of data structure size and memory allocation on overall performance. I predict that the insertion times for all three data structures will significantly improve.

Methodology:

- **Generating:**

I will generate N random integers so that duplicate values are not allowed in the generated sequence. The numbers will be generated independently and saved into a separate array.

I will then insert the generated integers one by one into the target sequence while maintaining the numerical order. To achieve this, I will use the insertion sort algorithm, which iteratively places each new element in its proper position by comparing it with the existing elements in the sequence.

- **Removing**

I will randomly select positions in the sequence and remove the element at that position.

To remove elements, I will traverse the data structures using a standard loop, instead of using std::advance() to avoid unintended optimisations that might affect the results.

I will then measure and record the execution time for each removal operation at different positions in the sequence.

Experiment 1: Testing the Data Structures

Std.vector

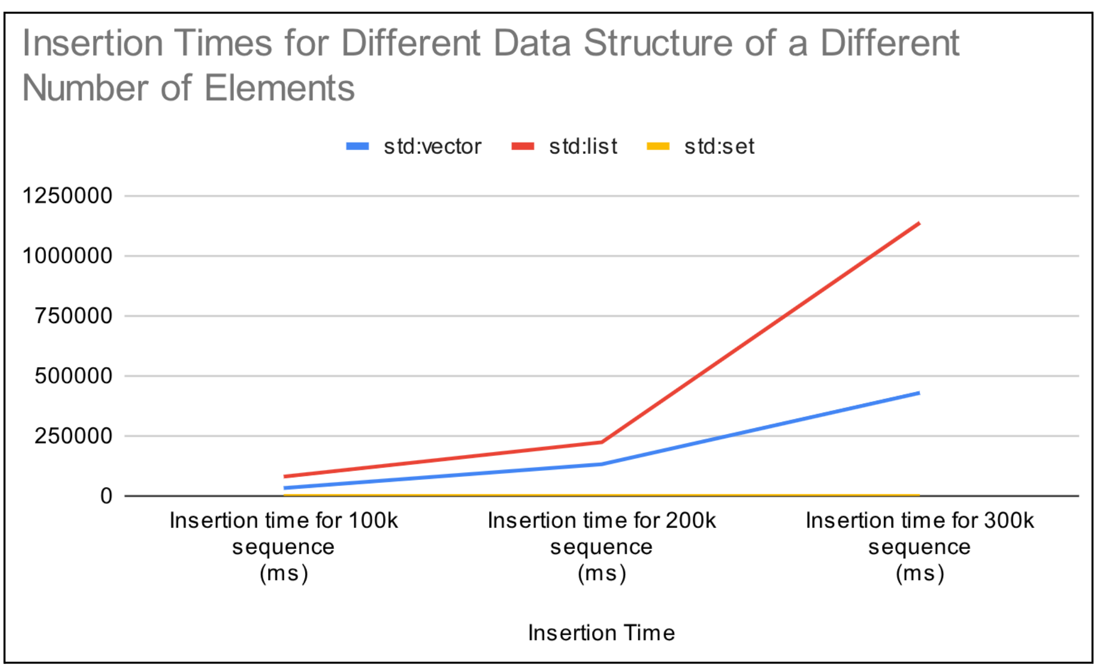
Iteration	Insertion time for 100k sequence (ms)	Removal time for 100k sequence(ms)	Insertion time for 200k sequence (ms)	Removal time for 200k sequence(ms)	Insertion time for 300k sequence (ms)	Removal time for 300k sequence(ms)
1.	31730	408	128717	2052	439562	7238
2.	32712	380	133071	2171	395326	6126
3.	34173	395	133701	2093	452186	6777
AVERAGE TIME	**32872**	**394**	**131830**	**6126**	**429024**	**6714**

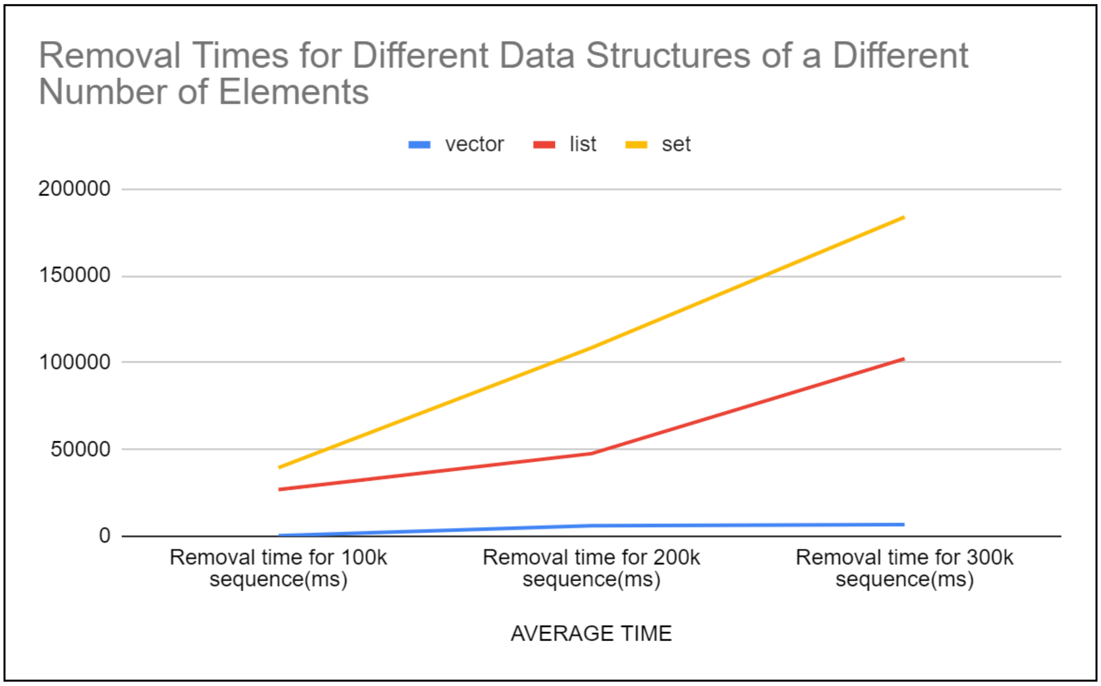
Std.list

Iteration	Insertion time for 100k sequence (ms)	Removal time for 100k sequence(ms)	Insertion time for 200k sequence (ms)	Removal time for 200k sequence(ms)	Insertion time for 300k sequence (ms)	Removal time for 300k sequence(ms)
1.	80635	29066	225449	44410	1621991	96079
2.	77243	22587	237799	44556	880106	112415
3.	83270	28875	206345	53878	908155	98382
AVERAGE TIME	**80383**	**26843**	**223198**	**47615**	**1136751**	**102292**

Std.set

Iteration	Insertion time for 100k sequence (ms)	Removal time for 100k sequence(ms)	Insertion time for 200k sequence (ms)	Removal time for 200k sequence(ms)	Insertion time for 300k sequence (ms)	Removal time for 300k sequence(ms)
1.	79	36086	188	94264	291	172549
2.	82	45724	199	103334	296	186798
3.	79	36760	179	128085	302	192447
AVERAGE TIME	**80**	**39523**	**189**	**108561**	**296**	**183931**





Experiment 2: Preallocated Data

vector

Iteration	Insertion time for 100k sequence (ms)	Removal time for 100k sequence(ms)	Insertion time for 200k sequence (ms)	Removal time for 200k sequence(ms)	Insertion time for 300k sequence (ms)	Removal time for 300k sequence(ms)
1.	252	250	1044	1450	2339	4424
2.	264	259	1032	1453	2394	4123
3.	247	248	1058	1521	2810	5596

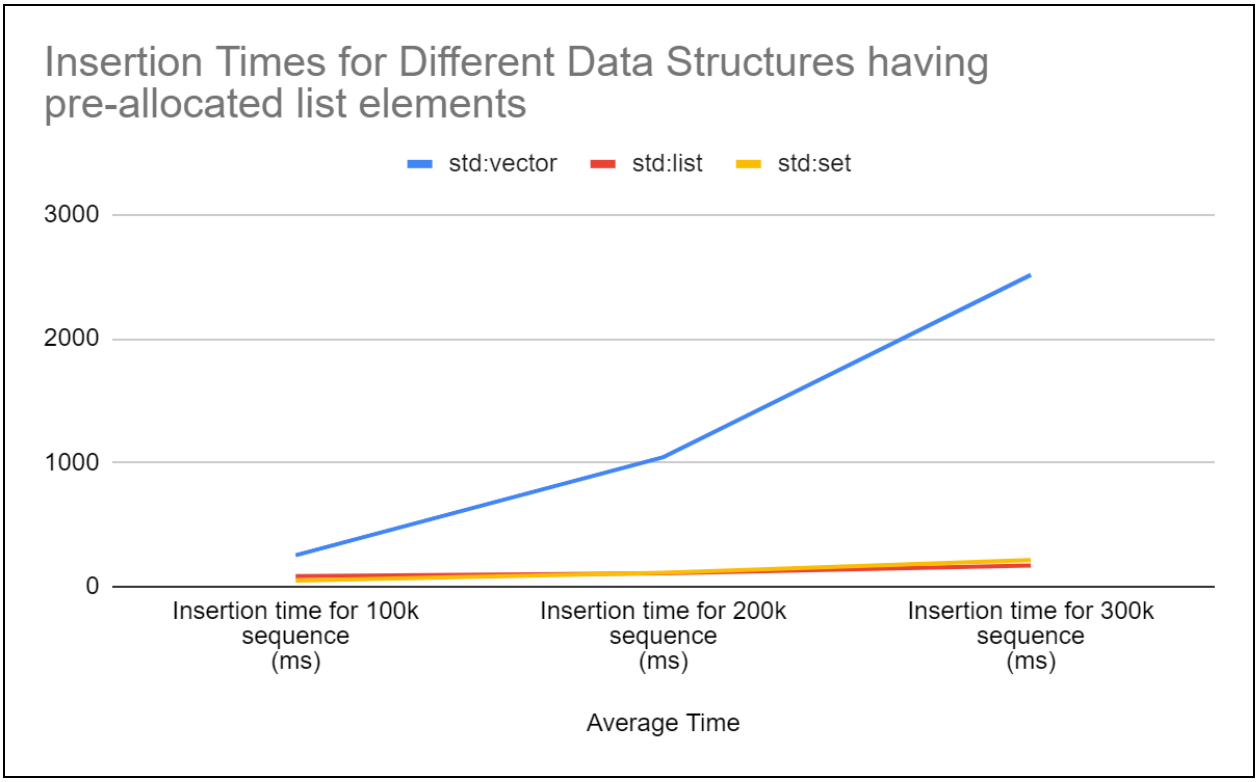
AVERAGE TIME	**254**	**252**	**1045**	**1475**	**2514**	**4714**

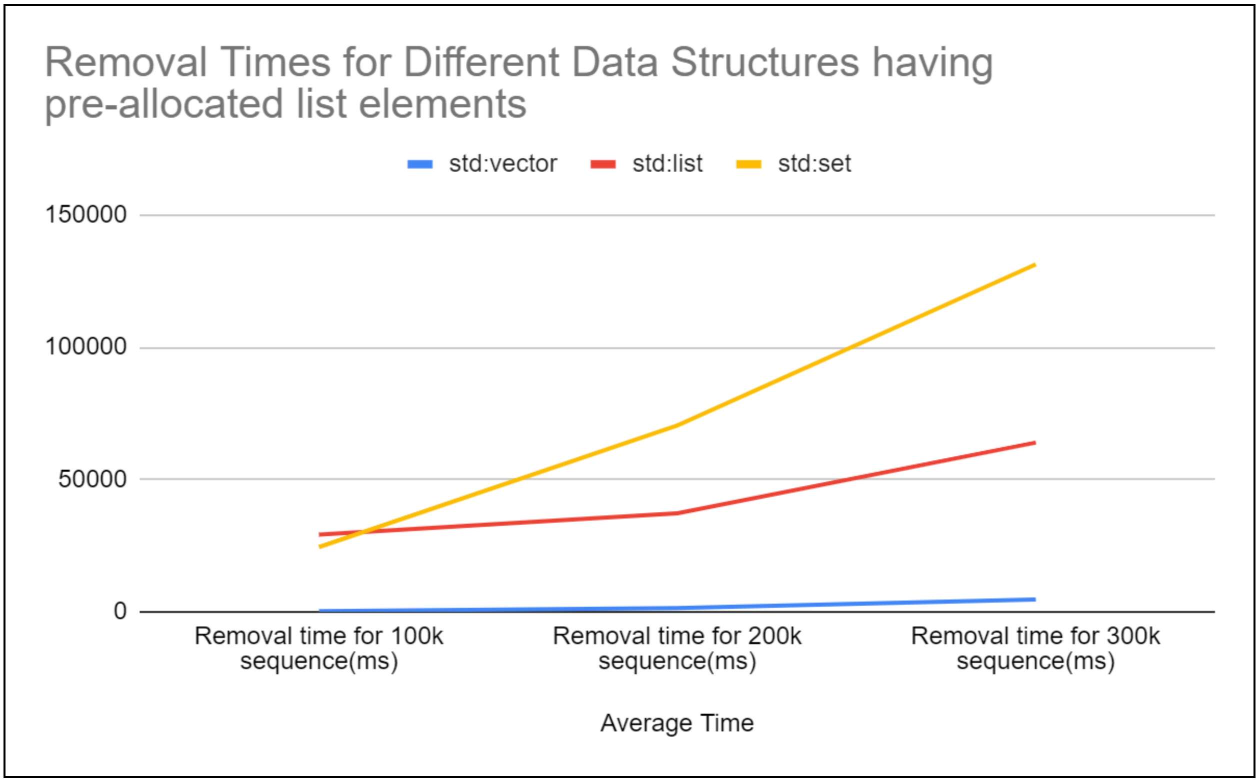
list

Iteration	Insertion time for 100k sequence (ms)	Removal time for 100k sequence(ms)	Insertion time for 200k sequence (ms)	Removal time for 200k sequence(ms)	Insertion time for 300k sequence (ms)	Removal time for 300k sequence(ms)
1.	88	30838	115	36776	171	64519
2.	83	27977	108	36092	167	62192
3.	81	28724	104	38915	174	65212
AVERAGE TIME	**84**	**29180**	**109**	**37261**	**171**	**63974**

set

Iteration	Insertion time for 100k sequence (ms)	Removal time for 100k sequence(ms)	Insertion time for 200k sequence (ms)	Removal time for 200k sequence(ms)	Insertion time for 300k sequence (ms)	Removal time for 300k sequence(ms)
1.	49	23718	107	65412	214	117310
2.	51	26391	125	72669	182	124565
3.	48	23352	105	73191	250	151981
AVERAGE TIME	**49**	**24487**	**112**	**70424**	**215**	**131285**


Insertion Times for Different Data Structures having pre-allocated list elements
std:vector
std:list
std:set
3000
2000
1000
0
Insertion time for 100k sequence (ms)
Insertion time for 200k sequence (ms)
Insertion time for 300k sequence (ms)
Average Time



Conclusion for Experiment 1:

std::vector shows relatively faster insertion times compared to std::list, especially as the sequence size increases. I believe this is because std::vector has contiguous memory storage and can benefit from cache locality during insertions.

std::list shows slower insertion times, especially as the sequence size grows. This is due to the overhead of managing nodes in a linked list.

std::set shows faster insertion times than std::list but slower than std::vector. I expected this, as std::set is used as a self-balancing binary search tree.

For removal times, std::set shows better performance than std::list, especially as the sequence size increases. This is because the tree structure of std::set allows for efficient and therefore faster removals.

std::list:

Insertion: Inserting elements in a std::list is efficient as it involves only updating the pointers to link the new element appropriately. However, finding the correct position for insertion requires traversing the list.

Removal: it is implemented as a doubly-linked list, where each element is connected to the next and previous elements through pointers. When removing an element from std::list, it is a relatively straightforward operation because the linked list allows direct access to the previous and next elements of the node to be removed.

std::list does not support random access, which means accessing elements by index is not possible. To find the insertion position, I would need to traverse the list from the beginning, which results in a slower time.

std::vector:
Insertion: To insert elements into a std::vector at a specific position the container needs to make room for the new element by shifting all the elements that come after the insertion point one position to the right. This is necessary to maintain the order of elements in the vector.

Removal: std::vector provides constant-time access to elements by index. When removing an element from a std::vector, the process is more efficient because it has contiguous memory storage, so accessing elements is very fast.

std:set
std::set exhibits the best insertion times among the three data structures for all sequence sizes. This is as expected since std::set is implemented as a balanced binary search tree, which allows for efficient element insertion while maintaining the sorted order. The average insertion time for std::set is significantly lower than both std::vector and std::list.

Removal: std::set shows competitive performance during removals compared to std::vector and std::list. Although std::vector performed better than std::set in the removal times for 100k sequences, std::set demonstrated competitive removal times for the larger sequences. This reflects the efficiency of the balanced binary search tree in std::set for element removals.

Comparing the data structures:
For the insertion part of the problem, std::list performs better because inserting an element into a std::list is faster than inserting into a std::vector.

However, for finding the insertion position (during the insertion sort process), std::vector performs better as it supports random access, allowing us to directly access elements by index, which is much faster than traversing the list as required in std::list.

std::set appears to be the best data structure for efficiently tackling this specific problem of inserting elements in proper numerical order and removing them randomly.

The actual results largely align with my expectations as both std::list and std::set performed as anticipated, with std::list showing better insertion times and std::vector outperforming std::list in removals. std::set exhibited competitive performance for both operations, as expected.

Surprising Result: Extremely fast insertion times for std::set.

The insertion times for std::set are significantly faster than both std::vector and std::list. The average insertion time for std::set is only 80 ms for 100k elements, while std::vector takes 32872 ms, and std::list takes 80383 ms.

This result is surprising because std::set is implemented as a balanced binary search tree, which typically has higher constant factors in its time complexity compared to dynamic arrays (std::vector) and linked lists (std::list).

Surprising Result: Slow removal times for std::set.

While std::set shows very fast insertion times, its removal times are significantly higher than expected. The average removal time for std::set is 39523 ms for 100k elements, while std::vector takes only 394 ms, and std::list takes 26843 ms
.
This observation is surprising because std::set is designed to provide efficient ordered removals due to its binary search tree structure. However, the actual removal times are slower than both std::vector and std::list.

Expected Result: std::vector outperforming std::list in removals and std::list outperforming std::vector in insertions.

The results show that std::vector performs better than std::list in removal times, and std::list performs better than std::vector in insertion times. These observations align with the expected characteristics of these data structures.

Overall, the surprising results are related to the extreme performance difference between std::set and other data structures in insertion times, as well as the unexpectedly slow removal times for std::set. The results show the importance of understanding the performance characteristics of data structures in different scenarios to make correct decisions when choosing a data structure to use.

Conclusion for Experiment 2:

In this experiment, I can observe the impact of preallocating the std::list before inserting elements. The results demonstrate that preallocation has a positive effect on insertion times for std::list, as expected. The time spent on memory allocation is significantly reduced because the space for the maximum number of elements is allocated in advance.

However, I noticed that preallocation might not have a substantial impact on removal times. Since std::list still requires linear traversal to find the element to remove, the removal times remain relatively high.

Compared with std::vector and std::set, preallocated stbd::list can now compete better in terms of insertion times. However, std::vector remains more efficient during removals due to its constant-time random access.

Surprising Result: The fast insertion times for std::set.

The insertion times for std::set are unexpectedly fast compared to both std::vector and std::list, even though std::set is a balanced binary search tree. In theory, balanced binary search trees have higher constant factors in their time complexity compared to dynamic arrays (std::vector) or linked lists (std::list).

One possible explanation for this result could be the efficiency of the underlying data structure implementation or compiler optimisations that have minimised the constant factors for std::set operations in this particular experiment.

Surprising Result: The slow removal times for std::set.

While std::set is expected to provide efficient ordered removals due to its binary search tree structure, the actual removal times for std::set are significantly higher than expected, especially for larger sequences.

This observation might be because of the logarithmic time complexity of removals from a balanced binary search tree. As the number of elements increases, the logarithmic factor has a noticeable impact on the removal times, making them slower than std::vector for larger sequences.

Expected Result: std::vector outperforming std::list in removals.

The observation that std::vector performs better than std::list in removal times aligns with expectations. Since std::vector provides constant-time random access, it can efficiently remove elements by directly accessing the required position without the need for linear traversal.

Expected Result: std::list outperforming std::vector in insertions.

The faster insertion times for std::list were as expected because linked lists allow for efficient insertions anywhere in the list, whereas dynamic arrays like std::vector might require shifting elements during insertions.

Conclusion:

Preallocating std::list can be a useful optimization to improve insertion times by avoiding frequent memory allocations. However, it doesn't change the fundamental characteristics of std::list, where removals still require linear time traversal. For scenarios where frequent insertions are essential and random removals are not the primary concern, preallocated std::list can be a competitive option. For applications with a balance of insertions and random removals, std::set might still offer a better overall performance. On the other hand, if frequent random access and sorting are involved, std::vector might be a better choice. In this particular problem of generating random integers and sorting them incrementally, using a std::vector for shuffling and a std::list for insertion sort. The choice of the data structure depends on the specific requirements and access patterns of the application.