C++ Containers Exercise

Richard Kroesen (774056)
30-7-2024 Aarhus

1. Introduction

This report is about the C++ programming language data containers, which are useful for saving an extensive amount of data which is needed for the application.

The purpose is to compare the standard vector, set, and list in performance of insertion and deletion from the standard C++ library. As an additional exploration a C++ array is being evaluated to gain another insight.

2. Data Containers Comparison

In this chapter, the data containers are compared in performance of insertion and deletion.

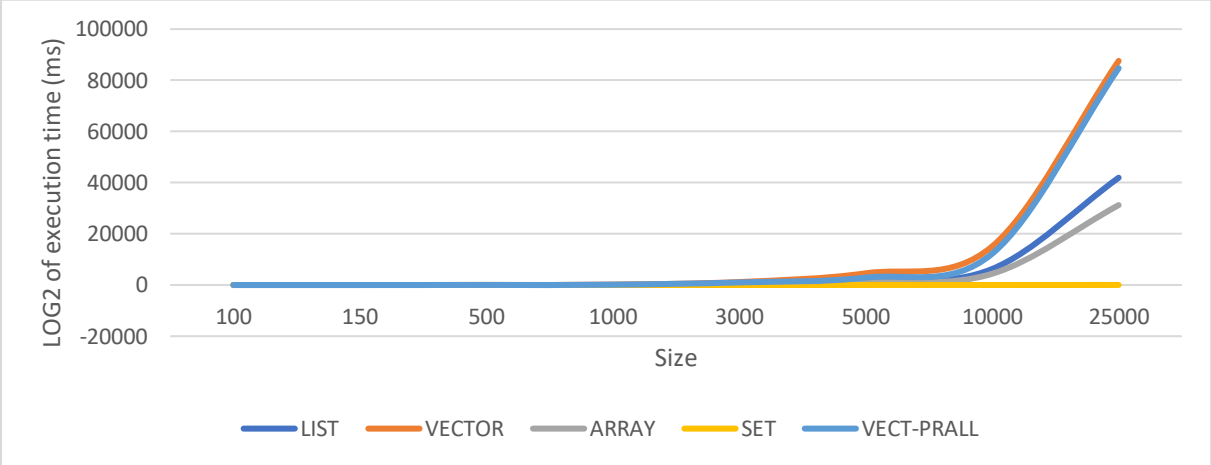

Figure 1 Insertion Comparison, linear

In the graph above it is clear that the set function is much outstandingly quicker with respect to others. As the second the C++ array seems to also be a good way to insert and sort the element. The drawback is ofcourse the required memory allocation during the programs lifecycle.

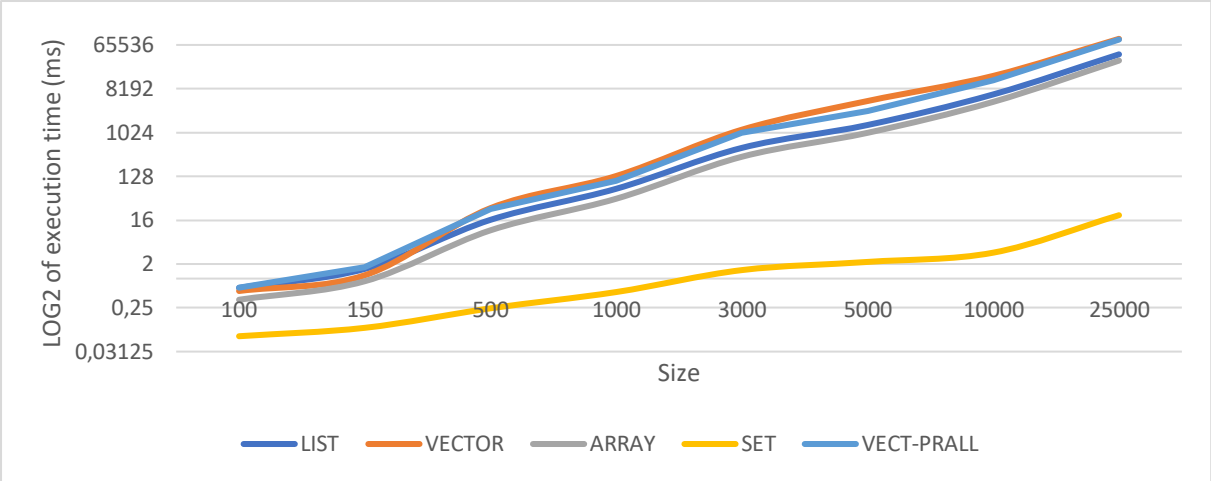

Figure 2 Insertion Completion, logarithmic

When the graph is represented using the logarithmic scale on the y-axis, the distinction is more clear between the containers. The differences between SET and other containers is clear.

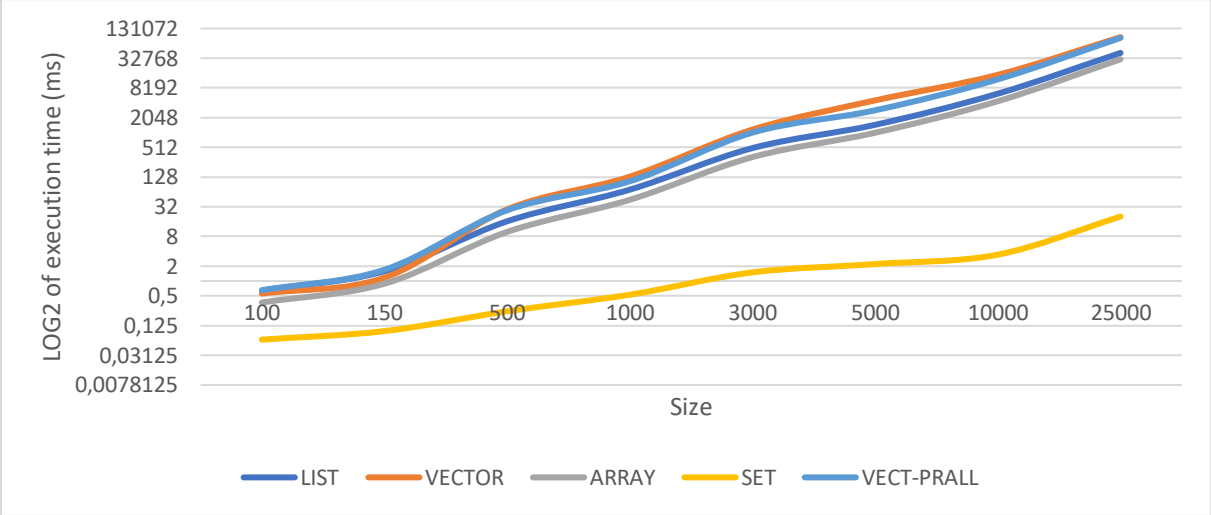

Figure 3 Deletion of elements at random indexes, logarithmic

2.1 Comparison Tradeoff

Now the question is what data structure is the best one to tackle the assignments problem? Which is the performance of random inserting and random removement within a given sequence.

Out of the measurements the set conainer is the best for inserting and removing at a random interval in an assending sorted data sequence.

2.2 Complementary Experiment

As an additional measurement the code setup is reused for custom structs.

```
1. struct MyStruct {
2.     int value;
3.     int large_array[100];
4. 
5.     bool operator<(const MyStruct& other) const {
6.         return value < other.value;
7.     }
```

```
 8.
 9.     bool operator==(const MyStruct& other) const {
10.         return value == other.value;
11.     }
12. };
```

Table 1 Struct structure, CPP abstract

The struct contain a value and a larger array. The struct has also additional operators like the compare and equals functions. Which are used by std::sort to sort.

Container Type/Size	**100**	**200**	**500**	**1000**
Reserved Vector	2,5	5,0	40,9	191,8
Vector	2,7	5,1	41,1	190,1
Reserved List	1,3	1,8	12,3	50,3
Array	2,0	3,8	29,0	130,0
Set	0,2	0,2	0,3	0,5

Table 2 Container comparison with custom structs

For this measurement the container types which could be pre-allocated are utilized. As an result, it is visible that the reserved list actually performs better than in other measurements.

3. Conclusion

Based on the theory the expectation is that the std::vector is the fastest at random access opartions. However, may be slower than others when inserting/deleting and resize is needed.

An alternative is the std::list which is a bit faster than the vector at inserting and removing elements. The std::set is the most fastest for inserting and removing elements for a ascending sorted sequence.

It is interesting to discover that the std::set is outperforming the other options by a various amount. The vector is the slowers, because it probably needs needs additional cycles to find available space when it becomes larger. In figure 1 all the containers have absolutely the simular performance, except the std::set which outperforms again.

That being said, the vector could be improved by the reservation of memory. It reduces on average already 10% of the exection time.

The most interesting thing from this assignment and unexpected is that the array was really that much faster than other containers. Even though a significant amount of flexibility is lost due to the useage of an array. This insight is an interesting trigger to explore further more within the Embedded Systems context especially.

Appendix A - Compiler & Build details

- **Operating System**: Windows 11 Pro, version 23H2, build 22631.3880.
- **Compiler**: I used the GCC 13.2.0 with MinGW, and I enforced using C++ 20 standard in the configuration.
- **Standard**: C++ 20
- **Build-Type:** in release