RIP Linked List

Empirical Study to Discourage You from Using Linked Lists Any Further

Benoît Sonntag
Université de Strasbourg
Strasbourg, France
Benoit.Sonntag@lisaac.org
0009-0000-2806-1970

Dominique Colnet
Université de Lorraine - LORIA
Nancy, France
Dominique.Colnet@loria.fr
0009-0006-7368-2235

***Abstract*—Linked lists have long served as a valuable teaching tool in programming. However, the question arises: Are they truly practical for everyday program use? In most cases, it appears that array-based data structures offer distinct advantages, particularly in terms of memory efficiency and, more importantly, execution speed. While it's relatively straightforward to calculate the complexity of operations, gauging actual execution efficiency remains a challenge. This paper addresses this question by introducing a new benchmark. Our study compares various linked list implementations with several array-based alternatives. We also demonstrate the ease of incorporating memory caching for linked lists, enhancing their performance. Additionally, we introduce a new array-based data structure designed to excel in a wide range of operations.**

***Index Terms*—linked lists, arrays, memory cache, performance, memory blocks**

I. Introduction

This article is a feedback and practical analysis of list data structures. After many years of programming practice, we realized that we never use a linked list anymore. Is this famous list structure implementation that we all studied at one time or another during our studies really useful ?

The theoretical advantages of a linked list are however numerous and attractive:

1) It allows a constant incremental allocation of the memory. Indeed, the addition of an element is equivalent to the allocation of a single cell in the list.
2) There are never any memory moves of cells during the life of the linked list.
3) Knowing the location of the insertion or removal of an element, the operation requires a constant number of instructions.

As for the drawbacks, it is the necessity of a partial and sequential path from cell to cell to reach an ith element that degrades performance. Many more complex implementations are possible to partially compensate for this shortcoming. We study two of them here: the presence of a backward chaining ("doubly linked" list), and the index cache management. When computers did not have much RAM and the speed of moving from one memory area to another was still critical, advantages 1 and 2 made the linked list profitable. In this paper, we want to know if there are still situations where the linked list is an advantageous and efficient implementation.

Eager to have a concrete and recent study of the structures in list and in search of the best strategy, the Bjarne Stroustrup's benchmark[1] seems to provide elements of an answer. Here, we propose to pursue the study and analyze the behavior of different list implementations using B. Stroustrup's benchmark. In this study, we introduce a new implementation called `ArrayBlock`, with the claim to have a relatively advantageous behavior in all real-life usage scenarios. We also propose another benchmark, we called `Fairbench`, which seems more relevant to test the efficiency and the behavior of linked lists in conditions closer to a realistic usage.

Note also that there is a lot of educational material about linked lists and/or the use of arrays, there are also some youtube vidéos [2], [3], [8], also some web articles [4], [5], [9], [10], but we found no research publication directly related to the main topic of this article.

II. Linked-lists Representations

We explore three linked list implementation options: `NoCacheList` (Fig. 1), `LinkedList` (Fig. 2) and `SingleList` (Fig. 3).

Fig. 1 shows the memory representation of `NoCacheList`. It is a doubly linked list that stores both the head and tail pointers. To avoid having to traverse the list to find out its length, a memory cache, called `size`, is used to store this information. In the example of Fig. 1, the list holds 4 pieces of data. Note that this representation corresponds exactly to the `LinkedList` class of the `Java` standard library[2]. As in `Java`, index 0 allows access to the first element (`data#0`), the second is at index 1 (`data#1`), and so on. Of course, if an element is added or removed, the `size` attribute must be updated accordingly.

The memory cache technique can also be used to store the location corresponding to the last access made in the list (see `LinkedList` on Fig. 2). The two variables, `cache index` and `cache link` store the user index and the pointer to the last accessed link respectively. In the example in Fig. 2, if the

[1]Bjarne Stroustrup's keynote in GoingNative 2012: Why you should avoid Linked List. https://www.youtube.com/watch?v=YQs6IC-vgmo

[2]The `Java` 8 (or later version of the language) is exactely like `NoCacheList`. At the time we are writing this article, there is still no other cache than the `size` cache.

979-8-3503-8166-5/24/$31.00

Memory Layout of `NoCacheList`

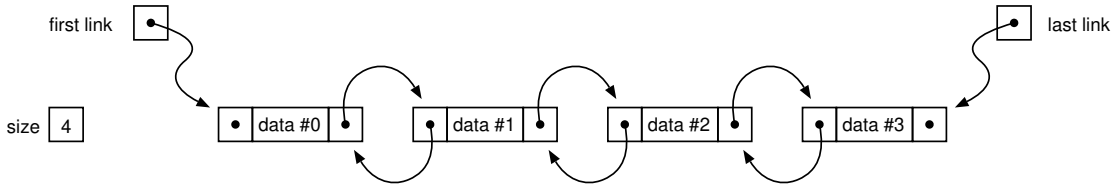

Fig. 1. A doubly linked list with a memory cache for `size`, containing 4 data items. The head pointer is stored in `first link`, and the tail pointer in `last link`.

Memory Layout of `LinkedList`

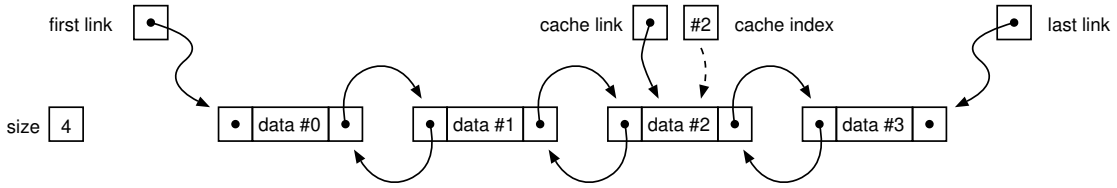

Fig. 2. Same layout as the one of Fig. 1, but with an extra cache of the last visited index (`cache link` and `cache index`).

Memory Layout of `SingleList`

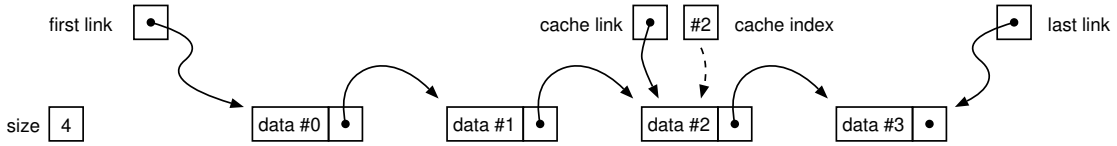

Fig. 3. One-way linked list with memory cache for the `size` and the last visited index (`cache link` and `cache index`).

user wants to access `data #2`, the fastest way is to go through the index cache. Thanks to the double linking and the index cache, sequential runs, from left to right or also from right to left, are in constant time for any list size.

The third representation considered, `SingleList` on the Fig. 3, corresponds to a single-linked list and, as in the previous case, has a memory cache for both the index and the size. Obviously, because of its single chaining, a `SingleList` will only be efficient when traversing from the left to the right. With the three previous representations, `NoCacheList`, `LinkedList` and `SingleList`, we cover the different possibilities for linked lists in a relatively exhaustive way.

One of the major drawbacks of linked lists is the amount of memory used by pointers. Even though the memory addresses of today's machines are limited to 48 bits, due to re-alignment problems, each pointer currently costs 64 bits. Thus, for a list of integers or pointers, the memory space of a list of N elements in case of double linking is $N \times 3 \times 64$ bits, that is $N \times 24$ bytes.

III. Array-based representations

Using contiguous memory areas (i.e., native arrays) saves memory space. For array-based representations, we have chosen two standard, well-known forms: `ArrayList` (Fig. 4) and `ArrayRing` (Fig. 5). Finally, the third representation is the implementation we call `ArrayBlock` (Fig. 6).

The `ArrayList` representation shown in the Fig. 4 is quite common in programming languages libraries. This representation has exactly the same name in the `Java` library. In `C++` this data structure is also known as `std::vector`. The principle of this data structure is to provide a storage area that is at least equal to, and often larger than what is needed, to avoid having to constantly adjust the size of the corresponding memory array [1]. In the example of the Fig. 4, the storage memory block consists of 8 slots, 4 of which are used and 4 are in reserve. The variable `storage` holds the pointer to the storage area and the variable `capacity` holds the allocation size of the storage area. The variable `size` stores the fact that only 4 slots are used. From the user's point of view, in order to comply with the same access interface as for the lists, the `4` stored datas are accessible via the index interval `[0,size-1]`. This representation is very simple because the access to the storage area is done without having to modify the index given by the user. This array representation is particularly well-suited for adding/deleting in queue. For example, deleting the last data item is simply a

Memory Layout of `ArrayList`

Fig. 4. Used area on the left and supply area on the right. Same indexing in the native array and in the user interface.

Memory Layout of `ArrayRing`

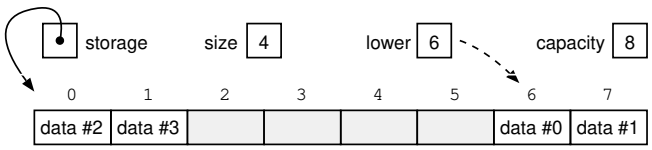

Fig. 5. The storage area is used in a circular fashion, from left to right. The variable `lower` is used to locate the internal index of `data#0`.

Memory Layout of `ArrayBlock`

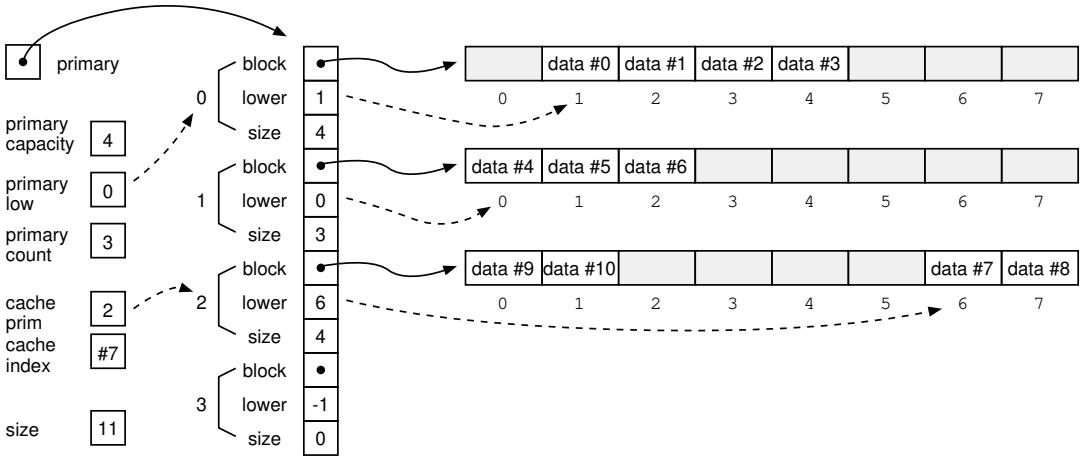

Fig. 6. A resizable primary table and storage fixed-size areas in power of 2. Circular management of all the tables.

decrement of `size`. In the case of adding at the last position, if there are available slots in reserve, the operation is also trivial. Obviously for an insert or an addition at the beginning, the operations become more complicated. For example, to insert at the first position, all the elements must be shifted one place to the right in order to make room for the new element at the index `0`.

Althought the memory capacity is twice the number of elements, the memory used is $N \times 2 \times 64$ bits, that is $N \times 16$ bytes. Thus with a reserve area of the same size as the used area, the memory consumption remains reasonable compared to the space taken up by a doubly linked list (i.e. $N \times 24$ bytes).

The Fig. 5 gives an example of the `ArrayRing` representation which allows to solve the problem of the addition in the first position quite simply. The principle is to use the storage area in a circular way. To do this, we add a variable `lower` that allows us to know where the data that the user accesses with index `0` is located. In the storage area, starting from this point, the data are stored from left to right, and, when we reach the end of the storage area, we start again from the beginning. The math that gets you from the user index to the storage area index is just an addition with `lower`. Whether it is a leading or trailing addition/deletion, the `ArrayRing` representation is of course very powerful. As in the case of `ArrayList`, the insertion anywhere other than head or tail remains problematic and requires potentially consequential moves. Nevertheless, the `ArrayRing` representation remains quite efficient when the insertion is close to either end (`0` or `size-1`). Note that it is always better to have a capacity that is a power of 2. In fact, the modulo that is necessary for the circular overflow of

the indices is calculated using the bitwize operator `and`[3].

The Fig. 6 gives an example of the `ArrayBlock` representation which is intended to behave more efficiently for all cases of insertions/deletions. This representation consists in using a resizable primary table that allows access to secondary level tables, the blocks, which are all of the same size. All blocks as well as the primary table itself are managed in a circular way, according to the same principle as for `ArrayRing`. The size of a secondary table is therefore a power of two, and relatively close to the size of memory page of the operating system, that is 2048 elements[4], which is the value that gave the best performance. Moreover, as in the case of lists, the representation `ArrayBlock` has an index cache thanks to the variables `cache prim` and `cache index`. The variable `cache prim` is used to store the index of the block corresponding to the last access. The variable `cache index` returns the user index corresponding to the first data of the corresponding block. Thus, as seen before, when memory accesses are located in a certain area, ideally close to `cache index`, we can restart the search from the block corresponding to the `cache prim` index. The strategy of the insertion and deletion algorithms is to preserve as much as possible, about a third, for free spaces within each block. In this way we avoid shifts in the primary table as much as possible. In this article, we will not go into detail about the insert and delete strategies, which can be very different and whose effectiveness depends mainly on the tests performed.

Without claiming to be completely exhaustive, these three array-based representations, `ArrayList`, `ArrayRing` and `ArrayBlock` provide a fairly complete overview.

IV. The Bjarne Stroustrup benchmark

Algorithm 1 The Bjarne Stroustrup benchmark

```
1:  list ← emptyList()
2:  for i ← 1, N do                          ▷ Step 1: filling of list
3:      val ← random number
4:      idx ← 0
5:      while (idx ≤ size(list) − 1) ∧ (item(list, idx) < val) do
6:          idx ← idx + 1
7:      end while
8:      list ← insert(list, idx, val)
9:  end for
10: for i ← 1, N do                          ▷ Step 2: clearing of list
11:     idx ← random in [0, size(list) − 1]
12:     list ← remove(list, idx) ;
13: end for
```

The benchmark proposed by B. Stroustrup consists of two phases (see algorithm 1). The first phase consists, for a given N value, in progressively building a sorted list composed of N randomly selected values. The second phase consists in removing the N values one by one, by randomly choosing the index of the removed value for each removal. Note that during the first phase of the insertion, as indicated by B. Stroustrup, we naively and sequentially search for the right position to make the insertion. It is not a dichotomous search for the right place to insert, as one might think.

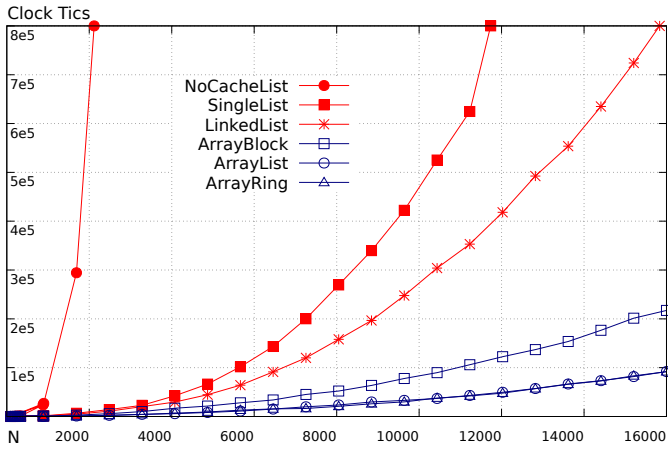

Fig. 7. The benchmark of B. Stroustrup using all the implementations of the paper, emphasizing limited N values. Even for relatively small collections, a clear distinction is evident between linked lists (red) and array-based structures (blue).

Fig. 7 shows the results for the B. Stroustrup benchmark with all the data structures previously described. Without having to go very far for the value of N, as announced by B. Stroustrup, there is a clear separation between linked and array-based implementations. In addition, this first run also shows the importance of having a cache for the last access index. In fact, the `NoCacheList` implementation is very slow already for a very small value of N. Even if it is possible to integrate the index memory cache into an iterator, it is still preferable to integrate it directly into the list as soon as the manipulation interface allows access to the elements via an indexing mechanism.

Still on the Fig. 7 and still on chained implementations, we can see the interest of the bidirectional linking, between `SingleList` and `LinkedList`. In fact, thanks to double chaining, it is possible to go backwards from the index cache, which is not possible with single chaining. As one might expect, `SingleList` should be reserved for algorithms that essentially only traverse in the ascending direction of the indices (i.e. from left to right).

The three best results are obtained with array-based representations: `ArrayList`, `ArrayRing`, and `ArrayBlock`. Note here that `ArrayBlock` is significantly slower than the other two array-based representations. Indeed, on arrays of relatively small size, as in this initial benchmark, using `ArrayBlock` results in a time loss.

[3]Let c be the capacity of the table which is a power of two. Given a valid index i in the table and an offset Δ with respect to that index, the corresponding index is given by $((i \pm \Delta) \& (c-1))$. If c is statically known, the calculation will only take one processor cycle.

[4]MMU (Memory Management Unit) is generally 4096 bytes in size. This is equivalent to 512 words of 64 bits. This choice of a 4 KB page was particularly well suited to 32-bit architectures. However, it is generally accepted that the use of a larger table of 8 KB or even 16 KB is preferable on 64-bit architectures.

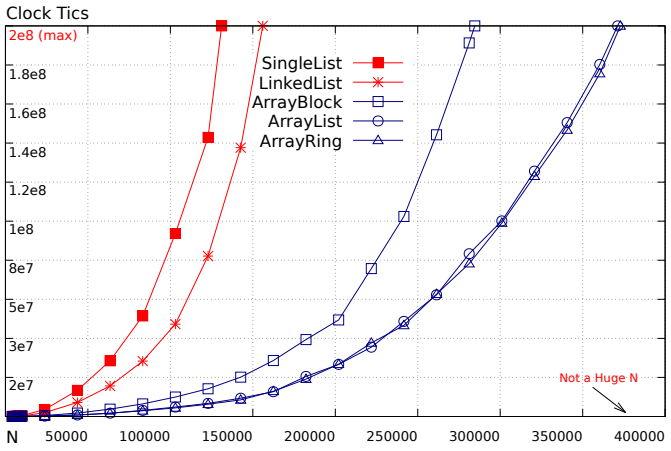

Fig. 8. With B. Stroustrup's benchmark, but pushing further for the value of N. The excessively slow representations are no longer shown (you will no longer see `NoCacheList`).

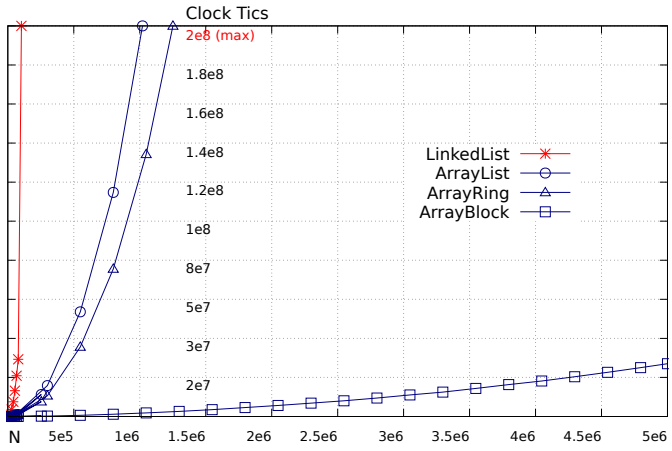

Fig. 9. Variation of benchmark B. Stroustrup: dichotomous insertion during the first phase. It is then possible to use a larger list by using `ArrayBlock`. Memory saturation would take too much time.

Note in passing that, in practice, it is essential for the programming language being used to facilitate a seamless switch between representations. In object-oriented languages, when the library is well-designed, the change of representation is achieved by modifying only the collection creation instruction. It is the mechanism of dynamic binding, or even better, the compiler that statically takes care of redirecting operations to the corresponding implementation.

Disregarding `NoCacheList`, Fig. 8 also illustrates the execution of B. Stroustrup's code, extending the N value. However, even though the three array-based implementations are clearly more efficient than the chaining-based ones, the execution times deteriorate very quickly for values of N that remain very modest. The complexity induced by the sequential insertion algorithm during the first insertion phase is of the order of $O(N^2)$ in direct correlation with our experimental results. Still referring to Fig. 8, even if N is greater, using `ArrayBlock` still results in a non-negligible time loss (around 2 times slower).

To visualize the relevance of the `ArrayBlock` structure in the case of random insertion/deletion on large data structures, we have slightly modified B. Stroustrup's benchmark by replacing the sequential search for the insertion location (lines 4 to 7 of the algorithm 1) with a dichotomous search, which reduces the complexity of the first phase of the benchmark to $O(log_2(N))$. The results for this modified version of the benchmark are shown in Fig. 9. The best results are clearly obtained with the `ArrayBlock` implementation. In fact, for the `ArrayList` and `ArrayRing` implementations, a deletion or an insertion implies on average a shift of $N/2$ elements. For `ArrayBlock`, the number of elements to move does not depend on N; the shift are directly related to the constant size of a block.

Note that `ArrayRing` performs marginally better than `ArrayList` because it allows choosing the most advantageous direction for shifting the elements. As for the bad performance of `LinkedList`, the problem does not come from the insertion or deletion which is in constant time, but from the random access into the list which has an average complexity of $O(N/4)$.

V. Fairbench: just fine for linked lists

Algorithm 2 The fairbench: the right benchmark for lists.

1: $list \leftarrow emptyList()$; $idx \leftarrow 0$
2: **for** $i \leftarrow 1, N$ **do** ▷ Step 1: list filling
3: **if** $(i/N < 1/3)$ **then**
4: $list \leftarrow addLast(list, someData(i))$
5: **else if** $(i/N < 2/3)$ **then**
6: $list \leftarrow addFirst(list, someData(i))$
7: **else**
8: $idx \leftarrow idx + 1$ ▷ or random incr. Fig. 12/13/14
9: $list \leftarrow insert(list, idx, someData(i))$
10: **end if**
11: **end for**
12: **for** $i \leftarrow 1, N$ **do** ▷ Step 2: list traversal
13: $sum \leftarrow sum + value(list, i)$
14: **end for**
15: $idx \leftarrow N/2$; ▷ Step 3: list clearing
16: **for** $i \leftarrow 1, N$ **do**
17: **if** $(i/N < 1/3)$ **then**
18: $idx \leftarrow idx - 1$; ▷ or random decr. Fig. 12/13/14
19: $list \leftarrow remove(list, idx)$
20: **else if** $(i/N < 2/3)$ **then**
21: $list \leftarrow removeFirst(list)$
22: **else**
23: $list \leftarrow removeLast(list)$
24: **end if**
25: **end for**

The idea of the fairbench (see algorithm 2), is to design a benchmark that is truly adapted to the concept of a list, in order to know whether linked implementations have good

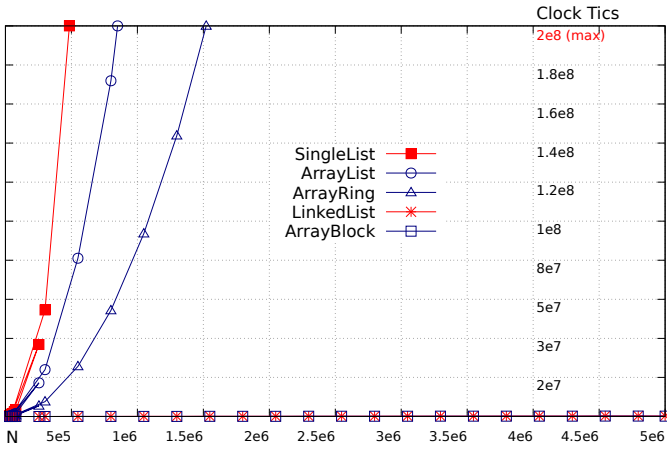

Fig. 10. Fairbench: to provide a chance for linked lists to win. Without going too far for N. And the winners are: `LinkedList` and `ArrayBlock`.

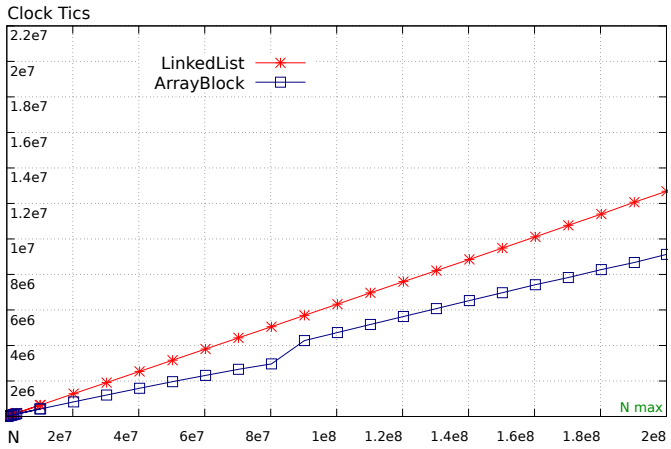

Fig. 11. Fairbench with N that saturates a 16 GB (gigabyte) memory in order to determine the true winner. `LinkedList` needs around 13 GB where `ArrayBlock` needs around 10 GB for the largest value of N (see Section VII for details).

performance compared to array-based implementations, and this for a consequent value of N.

The first phase (see *step 1* of the algorithm 2) for the fairbench is to grow the collection to its maximum of N using only `addLast` for the first third, then using only `addFirst` for the second third, and finally adding the last third during a single sequential run in the direction of increasing indices. Before emptying the list completely, we perform a complete run from right to left to calculate the sum of all the values (see *step 2* of the algorithm 2). The third and last phase consists in emptying the list by proceeding in the opposite way to the filling phase (see *step 3* of the algorithm 2). This benchmark is clearly designed to benefit linked lists as much as possible.

Fig. 10 shows the results of fairbench without going too far for the value of N in order to be able to distinguish which implementations are eliminated first. This Fig. clearly separates the losers (`SingleList`, `ArrayList` and `ArrayRing`) from the winners (`LinkedList` and `ArrayBlock`). Without being ridiculous, the value of N separating the winners from the losers, remains relatively modest considering the memory of today's computers. Note that the poor performance of `SingleList`, a list with single linking, is mainly explained by the use of `removeLast` in this benchmark.

Fig. 11 shows the results of running fairbench maximizing the value of N over the memory of the computer used for testing[5]. Of all the data structures presented, `LinkedList` is the most memory hungry, so it is `LinkedList` that determines the maximum possible value of N (see Section VII for more details on the computers/compilers used).

Although fairbench is designed to favor linked implementations, it is still `ArrayBlock` which behaves better than `LinkedList`. However, the difference in execution speed, while already noticeable, did not correspond to our observations during development[6]. Upon closer examination of the excellent results achieved by `LinkedList`, it becomes evident that fairbench tends to optimize for the precise memory caching of `LinkedList`.

Indeed, the memory cache of a `LinkedList` refers to the memory location of a single element (see Fig. 2). If access to the element immediately before or after is necessary, the memory cache of the `LinkedList` must be updated accordingly. Regarding `ArrayBlock` (see Fig. 6), the memory cache designates an entire block, requiring fewer updates. As long as two consecutive accesses do not vary more than the average number of elements per block, the cache of an `ArrayBlock` often remains valid. Thus, in Algorithm 2, we attempted to replace the index variations in lines 8 and 18 with random variations.

After several attempts, the hypothesis of a memory cache too specific for linked lists seems to be proving correct. Thus, in Figures 12, 13, and 14, we replace the increment/decrement of 1 with a progressively larger random increment/decrement. As we can see, the situation deteriorates significantly with the increasing increments. That being said, it is evident that these modifications to the Fairbench algorithm are, in a way, a return to Bjarne Stroustrup's benchmark. Nevertheless, it is observed that on collections of significant sizes, it is always `ArrayBlock` that achieves the best performance.

VI. Endgame: Add First or Remove Last

To complete our comparison, we have added two benchmarks, both of which also clearly favor linked representations. One of them favors adding and deleting at the head of the list: we add only with `addFirst`, and we empty the list only with `removeFirst`. The other one favors adding and deleting in

[5]All the subsequent figures are designed to maximize the value of N, as denoted by the green tag located in the bottom right corner.

[6]We use `ArrayBlock` in the development of large-scale software, including compilers and robotics systems, and noticed an even more pronounced difference when transitioning from `LinkedList` to `ArrayBlock`.

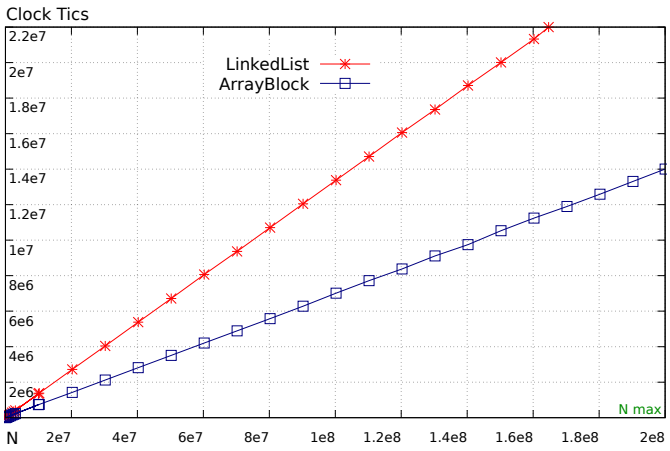

Fig. 12. Fairbench with modifications of lines 8 and 18 in the algorithm 2. The variable idx is incremented/decremented using a random number in range $[1, 32]$.

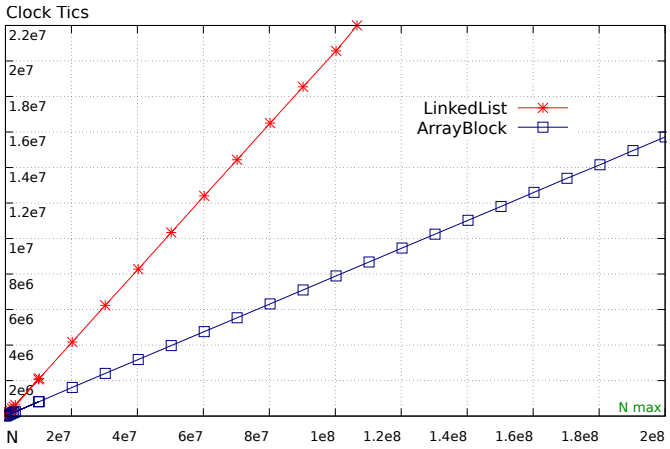

Fig. 13. Fairbench with modifications of lines 8 and 18 in the algorithm 2. The variable idx is incremented/decremented using a random number in range $[1, 64]$.

Algorithm 3 addLast / traversal / removeLast benchmark

```
1: list ← emptyList()
2: for i ← 1, N do                          ▷ Step 1: filling of list
3:     list ← addLast(list, randomNumber)
4: end for
5: for i ← 1, N do                          ▷ Step 2: list traversal
6:     sum ← sum + value(list, i)
7: end for
8: for i ← 1, N do                          ▷ Step 3: clearing of list
9:     list ← removeLast(list) ;
10: end for
```

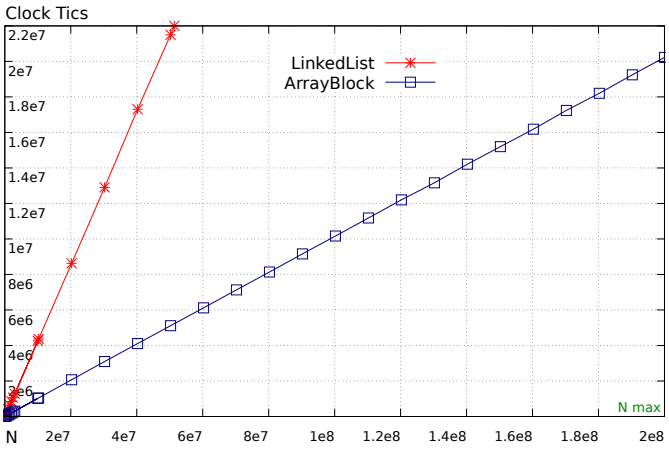

Fig. 14. Fairbench with modifications of lines 8 and 18 in the algorithm 2. The variable idx is incremented/decremented using a random number in range $[1, 128]$.

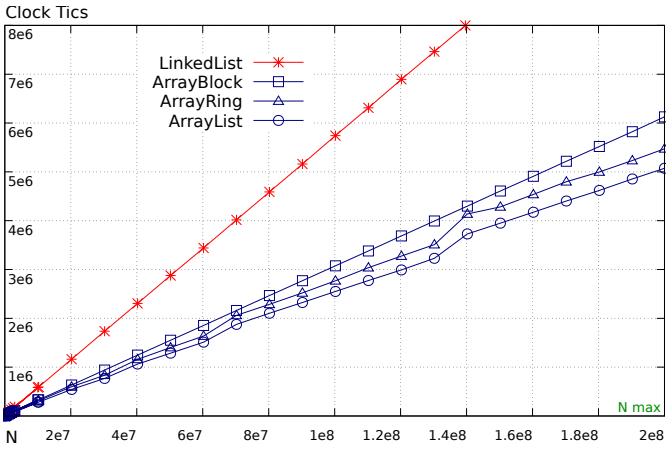

Fig. 15. addLast / traversal / removeLast (algorithm 3).

queue: we add only with `addLast`, and we empty the list only with `removeLast`. In both cases, before emptying the list, we make a single run from left to right, thus accessing all the elements.

Moreover, since we are only interested in lists of a significant size, we saturate 16 GB of memory in both cases and keep only the implementations that withstand these constraints. The results are shown in figures 15 and 16. Even though these results speak for themselves, we should mention that it is not possible to add `SingleList` in Fig. 15 because in the case of `removeLast`, the complexity is about $O(n)$. No surprise, it is also not possible to present `ArrayList` on the Fig. 16.

VII. Benchmarking Methodology

All the benchmarks presented in this article were conducted on the same computer configuration, consistently utilizing the same programming language, the same compiler with identical options, and the same library.

Algorithm 4 addFirst / traversal / removeFirst benchmark

```
1: list ← emptyList()
2: for i ← 1, N do                          ▷ Step 1: filling of list
3:     list ← addFirst(list, randomNumber)
4: end for
5: for i ← 1, N do                          ▷ Step 2: list traversal
6:     sum ← sum + value(list, i)
7: end for
8: for i ← 1, N do                          ▷ Step 3: clearing of list
9:     list ← removeFirst(list) ;
10: end for
```

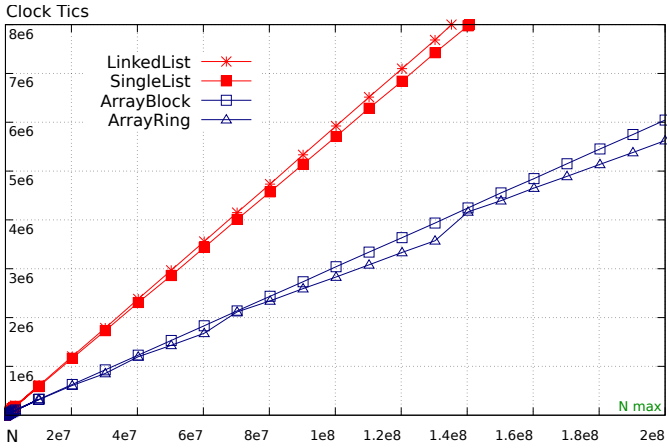

Fig. 16. addFirst / traversal / removeFirst (algorithm 4).

We have, of course, verified that the change of processor has no effect. Whether using an Intel Core or AMD processor is inconsequential. No relative differences were observed following the processor change. The performance curves exhibit a remarkable relative stability.

The tests were conducted on memory capacities ranging from 4 to 16 GB, and similarly, no relative disparities were detected across these different configurations. The curves presented in the article correspond to a 16 GB memory, and the maximum value of N ($N_{\max} = 2 \times 10^8$) was chosen to reach this maximum value with the most memory-intensive data structure (see Fig. 11). For those of you who will rerun the tests on your own machine, don't forget to decrease or increase the maximum value of N based on the maximum size of your memory[7].

Furthermore, it should be noted that transitioning from one operating system to another, whether it be Linux or MacOS, had no notable impact. It is highly conceivable that the results remain consistent when using Microsoft Windows or, of course, also an Android smartphone.

Regarding execution times, we chose to display only the number of clock ticks, considering that an absolute value in minutes or seconds would not add any significant information[8].

[7]A constant is provided for this purpose in the code accompanying the article.

[8]Nevertheless, it is worth noting that it took approximately 5 hours to recalculate all the figures in this article.

Additionally, each execution is repeated multiple times, and only the minimal value in terms of execution time is retained. Working on a machine dedicated to benchmark execution, we also observed a rapid convergence towards a stable result. Thus, after 3 or 4 executions of the same test, we converge towards a minimal value that remains stable thereafter.

It is during the implementation of a project combining compilation and robotics that we developed our own data structure, named `ArrayBlock`. The outstanding results achieved with `ArrayBlock` during these developments motivated the writing of this article.

Regarding the language utilized in the context of our robotics project, it is important to note that we used the `Lisaac` programming language [7] [6]. This language allows us to achieve optimal performance, and we primarily use it because it generates excellent C code. The interfacing with robotics peripherals is thus straightforward. Since `Lisaac` code is initially translated into C before benefiting from all the optimizations of the C compiler, it is evident that we would gain nothing by writing all these benchmarks directly in C.

Furthermore, since the `Lisaac` language is an object-oriented language, all benchmark algorithms are written only once and shared through inheritance. The `Collection` type serves as a common abstraction for all the data structures presented in this article. Therefore, each time, the same code inherited from `Collection` is applied to all the different data structures being compared.

Regarding the compilation of the C code, all the benchmarks in this article were compiled with gcc. The use or non-use of gcc optimization options has no relative effect. The change of the C compiler also does not disturb the ranking.

VIII. Conclusion

Indeed, our study shows that it is very difficult to find much interest in using linked lists in real applications. As long as we are on small values of N, linked implementations rarely have a significant advantage. However, if we stick to the idea of using linked lists, this article also highlights the importance of implementing an index cache for the managing of a linked list[9]. As mentioned earlier, the optimal approach is to integrate a cache within the data structure itself. Additionally, if iterators also exist in your library, it is advisable to have a dedicated cache for each iterator [11].

Overall, the increase in RAM size and the speed of today's processors allows more complex implementations to be used. Also, the processor's memory caches give advantage to contiguous accesses and speed up data shifting. Our `ArrayBlock` implementation takes advantage of these technological innovations and gives very good results in most all the tests we have done. In fact, as soon as the size of the collection becomes substantial, `ArrayBlock` is likely the best choice. However, we could invent and test many types of benchmarks, but we focused on extreme cases that theoretically give the linked list an advantage.

[9]At the time we are writing this article, the `C++ std::list` data structure have no index memory cache.

The memory representation of `ArrayBlock` is very similar to the more basic implementation of the virtual memory management on current processors. The two levels of the MMU indirection table on 32-bit processors (4 levels for 64-bit processors) are similar to our primary table. Then, the fixed 4KB pages are similar to our small circular arrays of fixed size in powers of 2. The primary table gives the flexibility to add non-contiguous blocks for fast insertions, and our small contiguous arrays bring the speed of direct access to an element. The circular index management for both the primary table and the small contiguous arrays allows the complexity of adding or deleting to be divided by two. The cost of a circular index management is negligible compared to the benefits. We show it perfectly here with quite surprising results with a simple implementation of the circular management of an array `ArrayRing`.

Note that currently, the implementation of Python[10] uses a data structure equivalent to an `ArrayList` for the native [] operator, even in the case of a substantial collection. For high-level languages designers, the search for an ideal list structure implementation under all circumstances is also important. A high-level language whose goal is to simplify the choice of data structures by using a single structure for lists, especially for untyped languages, must pay attention to this implementation or else its overall performance will be severely degraded. For this important issue, our implementation is clearly a very polyvalent solution.

[10]CPython 3.12.1 file `Objects/listobject.c`

Conflict of Interest

The authors declare no conflict of interest.

Author Contribution

The authors contributed equally to this work and are listed in alphabetical order of their first names.

Funding

Both authors are civil servants of the French government. We gratefully acknowledge the French government for providing us with the freedom to pursue our research interests.

References

[1] Dominique Colnet and Benoît Sonntag. Exploiting array manipulation habits to optimize garbage collection and type flow analysis. *Software: Practice and Experience*, 45(12):1639–1657, December 2015.
[2] Computerphile. Arrays vs linked lists - computerphile, 2018.
[3] Caleb Curry. Arrays vs linked lists - data structures and algorithms, 2021.
[4] Dat Hoang. Performance of array vs. linked-list on modern computers, 2018.
[5] Johnny's Software Lab. The quest for the fastest linked list, 2021.
[6] Benoit Sonntag and Dominique Colnet. Efficient compilation strategy for object-oriented languages under the closed-world assumption. *Software: Practice and Experience*, 44(5):565–592, 2013.
[7] Benoit Sonntag and Dominique Colnet. Lisaac: the power of simplicity at work for operating system. In *In 40th conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'2002)*, pages 45–52. Australian Computer Society, 2022.
[8] Bjarne Stroustrup. Why you should avoid linked list, 2012. Keynote speaker in GoingNative 2012.
[9] Baptiste Wicht. C++ benchmark - std::vector vs std::list, 2012.
[10] Baptiste Wicht. C++ benchmark – std::vector vs std::list vs std::deque, 2012.
[11] Olivier Zendra and Dominique Colnet. Adding external iterators to an existing eiffel class library. pages 188–199, 02 1999.