RIP Linked List

Benoît Sonntag, Dominique Colnet

RIP Linked List

Benoit Sonntag[1*] and Dominique Colnet[2*]

[1*] University of Strasbourg, France .
[2*]LORIA, University of Lorraine, France.

*Corresponding author(s). E-mail(s): Benoit.Sonntag@lisaac.org; Dominique.Colnet@loria.fr;

Abstract

Linked lists have always been an excellent teaching tool in programming. The question arises as to whether it is really worthwhile to use linked lists in the programs we run on a daily basis. It seems that in most cases array-based data structures are more advantageous, both in terms of memory space and, most importantly, in terms of execution speed. While it is easy to calculate the complexity of the operations, what about the actual execution efficiency? In this paper we try to answer this question by introducing a new benchmark. Our survey compares several linked-list implementations with some array-based implementations. We also propose a new array-based data structure that is well suited for list operations.

Keywords: linked lists, arrays, memory cache, performance

1 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 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 circumstances of use. 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[2].

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 [1, 2, 5], also some web articles [3, 4, 6, 7], but we found no research publication directly related to the main topic of this article[3].

2 Representations based on linked lists

We consider three implementation options for linked lists: `NoCacheList` (figure 1), `LinkedList` (figure 2) and `SingleList` (figure 3).

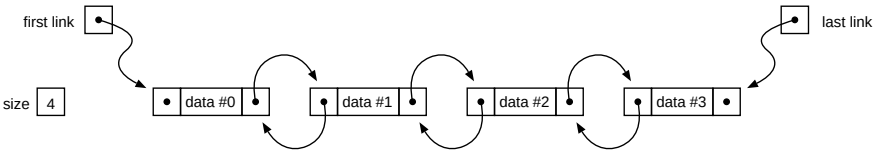

Fig. 1 Memory layout of `NoCacheList`. Doubly linked list with memory cache for the `size`.

Figure 1 shows the memory representation of `NoCacheList`. It is a chained list that store both the head and tail pointers. To avoid having to traverse the

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

[2] All the benchmarks have been tested in the same programming language, with the generation of a C code, on the same 16 GB machine, with the same compiler `gcc`.

[3] By the way, we are looking for a good conference to publish this paper which deserves to be shared. If you have a suggestion, please, drop me a mail (`dominique.colnet@loria.fr`).

list to find out its length, a memory cache, called `size`, is used to store this information. In the example of the figure 1, the list holds 4 data. Note that this representation corresponds to the `LinkedList` class of the `Java` standard library. 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.

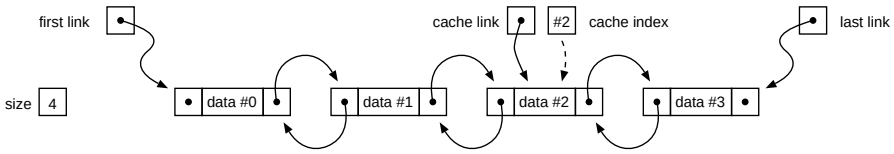

Fig. 2 Memory layout of `LinkedList`. Two-way linked list with memory cache for the `size` and the last visited index (`cache link` and `cache index`).

The memory cache technique can also be used to store the location corresponding to the last access made in the list (see `LinkedList` on figure 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 Figure 2, if the 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.

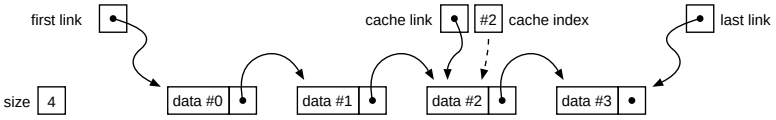

Fig. 3 Memory layout of `SingleList`. One-way linked list with memory cache for the `size` and the last visited index (`cache link` and `cache index`).

The third representation considered, `SingleList` on the figure 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 $24 \times N$ bytes.

3 Array-based representations

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

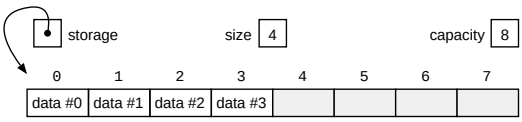

Fig. 4 Memory layout of `ArrayList`. Used area on the left and supply area on the right. Same indexing in the native array and in the user interface.

The `ArrayList` representation shown in the figure 4 is quite common in programming languages libraries. This representation has exactly the same name in the `Java` library. In `Cpp` 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. In the example of the figure 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 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 $16 \times N$ 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.

The figure 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,

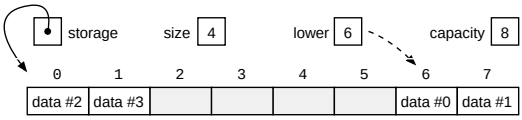

Fig. 5 Memory layout of `ArrayRing`. The storage area is used in a circular fashion, from left to right. The variable `lower` is used to locate the internal index of `data` #`0`.

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`[4].

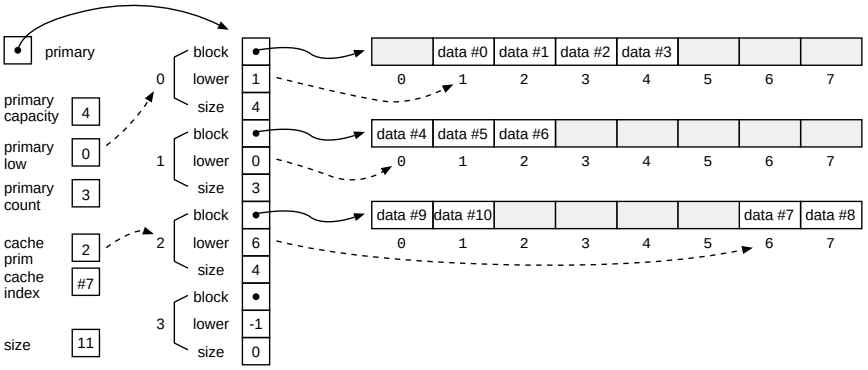

Fig. 6 Memory layout of `ArrayBlock`. A resizable primary table and storage fixed-size areas in power of 2. Circular management of all the tables.

The figure 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

[4]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.

system, that is 2048 elements[5], 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.

4 The Bjarne Stroustrup benchmark

Algorithm 1 The Bjarne Stroustrup benchmark

1: $list \leftarrow emptyList()$;
2: **for** $i \leftarrow 1, N$ **do** ▷ Step 1: filling of $list$
3: $value \leftarrow$ random number ;
4: $index \leftarrow 0$;
5: **while** $(index \leq size(list) - 1) \wedge (value(list, index) < value)$ **do**
6: $index \leftarrow index + 1$;
7: **end while**
8: $list \leftarrow add(list, index, value)$;
9: **end for**
10: **for** $i \leftarrow 1, N$ **do** ▷ Step 2: clearing of $list$
11: $index \leftarrow$ random in $[0, size(list) - 1]$;
12: $list \leftarrow remove(list, index)$;
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

[5]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.

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.

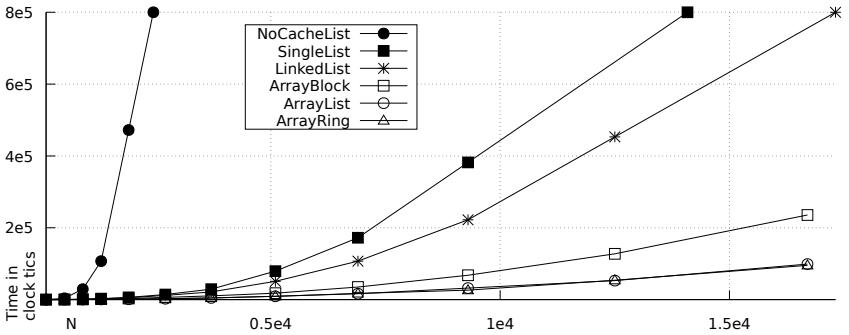

Fig. 7 The benchmark of B. Stroustrup with all the implementations of the paper but not going too far for the value of N. Strong separation between linked list and array based structures.

Figure 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 figure 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. The three best results are obtained with array-based representations: `ArrayList`, `ArrayRing` and `ArrayBlock`.

Figure 8 also shows the execution of B. Stroustrup's benchmark keeping only the array-based implementations in order to push the N value further. 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 results.

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

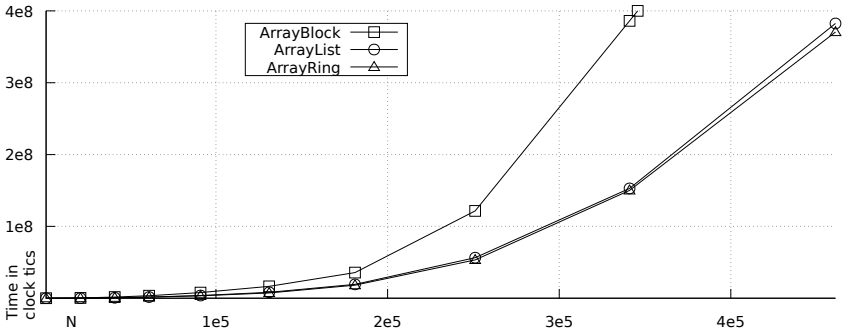

Fig. 8 Figure 7 continued. B. Stroustrup's benchmark, keeping only the array-based implementations and going a little bit further for N.

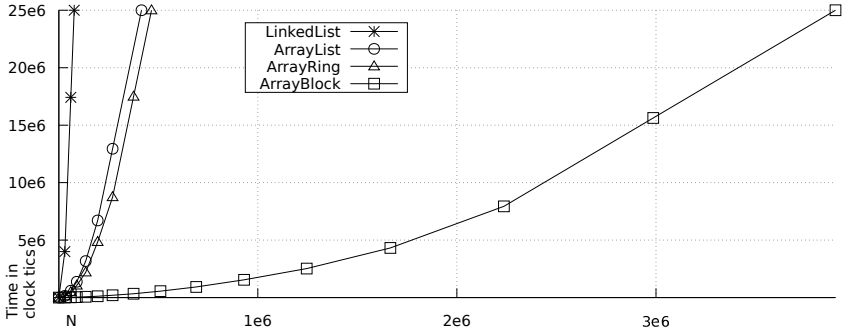

Fig. 9 Variation of benchmark B. Stroustrup: dichotomous insertion during the first phase. It is then possible to use a larger list by using `ArrayBlock`.

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 figure 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 slightly better than `ArrayList` because it is possible to choose 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 $O(N/4)$.

Algorithm 2 The fairbench: the right benchmark for lists.

```
1:  list ← emptyList() ;    index ← 0 ;
2:  for i ← 1, N do                                  ▷ Step 1: filling of the list
3:      if (i/N < 1/3) then
4:          list ← addLast(list, data(i)) ;
5:      else if (i/N < 2/3) then
6:          list ← addFirst(list, data(i)) ;
7:      else
8:          index ← index + 1 ;
9:          list ← add(list, index, data(i)) ;
10:     end if
11: end for
12: for i ← 1, N do                                  ▷ Step 2: traversal of the list
13:     sum ← sum + value(list, i) ;
14: end for
15: index ← N/2 ;                                    ▷ Step 3: clearing of the list
16: for i ← 1, N do
17:     if (i/N < 1/3) then
18:         index ← index − 1 ;
19:         list ← remove(list, index) ;
20:     else if (i/N < 2/3) then
21:         list ← removeFirst(list) ;
22:     else
23:         list ← removeLast(list) ;
24:     end if
25: end for
```

5 Fairbench: just fine for linked lists

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 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 we 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.

Figure 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 figure clearly separates the losers (`SingleList`, `ArrayList` and `ArrayRing`) from the winners (`LinkedList` and `ArrayBlock`). Without being

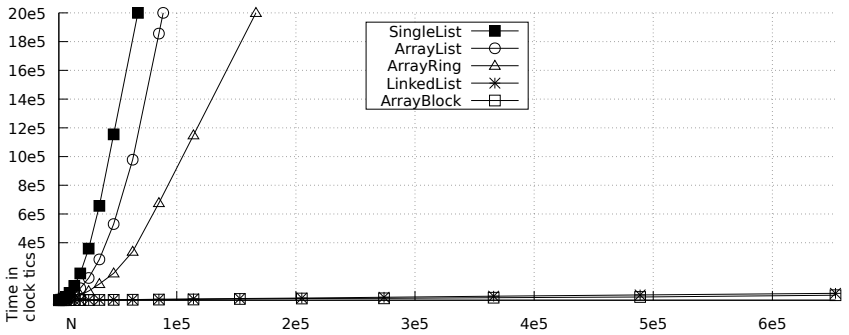

Fig. 10 Fairbench without going too far for N. Winners: `LinkedList` and `ArrayBlock`.

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.

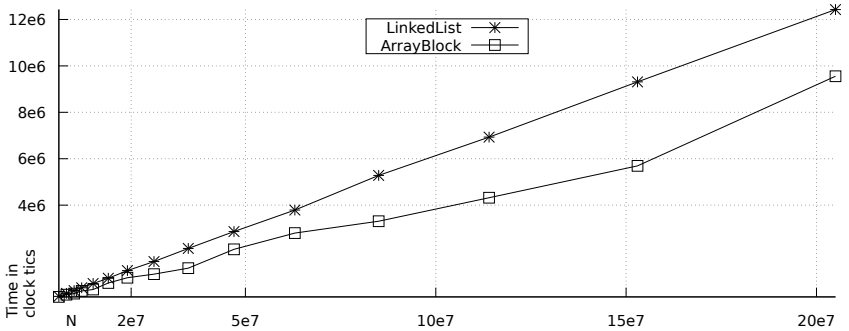

Fig. 11 Fairbench with N that saturates a 16 GB (gigabyte) memory. `LinkedList` needs around 13 GB where `ArrayBlock` needs around 10 GB for the largest value of N.

Figure 11 shows the results of running fairbench maximizing the value of N over the memory of the computer used for testing[6]. Although fairbench is designed to favor linked implementations, it is still `ArrayBlock` which behaves better than `LinkedList`. Of all the data structures presented, `LinkedList` is the most memory hungry, so it is `LinkedList` that determines the maximum value of N. All the following figures also maximize the value of N.

A closer look at the very good results of `LinkedList` shows that fairbench favors the very accurate cache of `LinkedList` (figure 2) over the larger cache of `ArrayBlock` (figure 6). However, even if the index only moves by `1`, the

[6]Note that we found no relative performance differences when using either a more powerful computer and/or with more memory. Similarly, we found no relative difference when changing the `C` compiler.

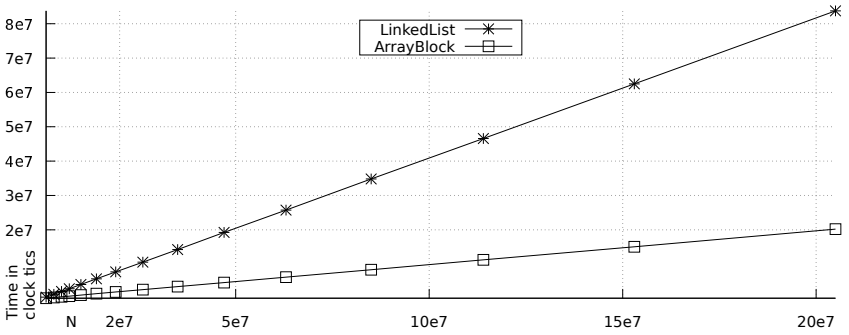

Fig. 12 Fairbench with modifications of lines 8 and 18 in the algorithm 2. The variable *index* is incremented using a random number in range $[1, 128]$.

cache of `LinkedList` still needs to be updated. For `ArrayBlock`, as long as the accessed index remains in the same block, the cache value does not change. A simple modification of fairbench, shown in figure 12, reveals the effect of the large cache of `ArrayBlock`.

6 `addLast` / `removeLast` or `addFirst` / `removeFirst`

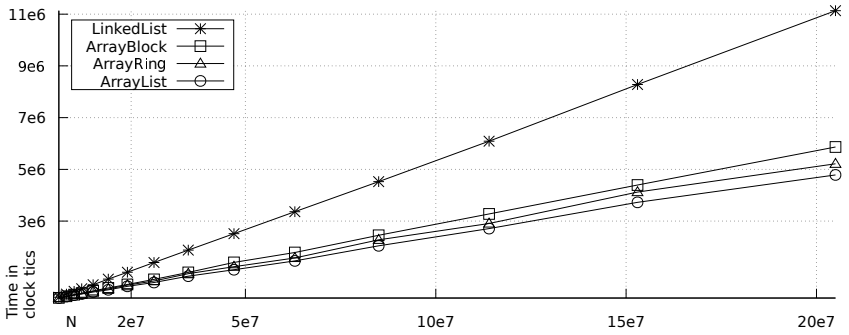

Fig. 13 Step 1: N times `addLast`. Step 2: one whole traversal. Step 3: N times `removeLast`

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 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

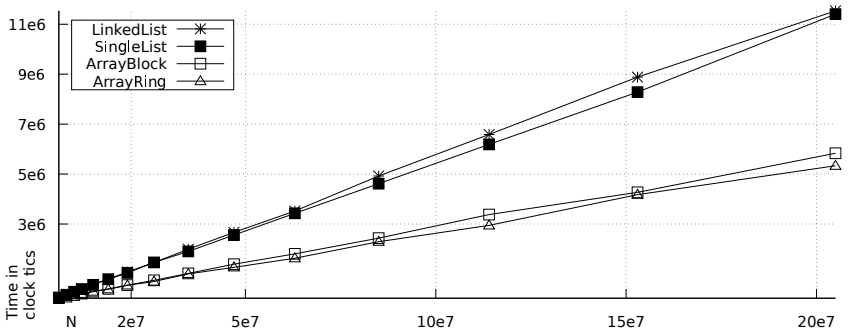

Fig. 14 Step 1: N times `addFirst`. Step 2: one whole traversal. Step 3: N times `removeFirst`.

withstand these constraints. The results are shown in figures 13 and 14. Even though these results speak for themselves, we should mention that it is not possible to add `SingleList` in figure 13 because in the case of `removeLast`, the complexity is about $O(n)$. No surprise, it is also not possible to present `ArrayList` on the figure 14.

7 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.

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 all the tests we have done. However, we could invent and test many types of benchmarks, but we focused on extreme cases that theoretically give the linked list an advantage.

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`.

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.

References

[1] Computerphile. Arrays vs linked lists - computerphile, 2018. URL https://www.youtube.com/watch?v=DyG9S9nAlUM.

[2] Caleb Curry. Arrays vs linked lists - data structures and algorithms, 2021. URL https://www.youtube.com/watch?v=dMy2hq9OUMc.

[3] Dat Hoang. Performance of array vs. linked-list on modern computers, 2018. URL https://dzone.com/articles/performance-of-array-vs-linked-list-on-modern-comp.

[4] Johnny's Software Lab. The quest for the fastest linked list, 2021. URL https://johnysswlab.com/the-quest-for-the-fastest-linked-list/.

[5] Bjarne Stroustrup. Why you should avoid linked list, 2012. URL https://www.youtube.com/watch?v=YQs6IC-vgmo. Keynote speaker in GoingNative 2012.

[6] Baptiste Wicht. C++ benchmark - std::vector vs std::list, 2012. URL https://baptiste-wicht.com/posts/2012/11/cpp-benchmark-vector-vs-list.html.

[7] Baptiste Wicht. C++ benchmark – std::vector vs std::list vs std::deque, 2012. URL https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html#.