Evaluation of Classical Data Structures in the Java Collections Framework

Anil L. Pereira

1 Introduction

The Java Collections framework [1] is a unified architecture for representing and manipulating data collections. The classical data structures [2] implemented in the Java Collections framework and considered in this paper are array, array list, linked list, doubly linked list, stack, and queue [3]. This paper asks an important question and attempts to answer it. The question is, what are the important performance considerations of the classical data structures as implemented in the Java Collections framework when using asymptotic analysis [4] for software design? For example, inserting or removing an element at the end of an array list or linked list data structure takes constant time irrespective of the number of elements in the data structure. However, the software execution time relative to the array list is much faster due to its memory being allocated contiguously and with less overhead. Even though the time complexity in this case is the same for both data structures, clearly the array list would be a better choice among the two in order to buffer small amounts of data while transmitting or receiving data at high speeds through a network. The paper seeks to answer the above question by analyzing the performance gap between the data structures when similar operations have equal time and equal space complexity. To the best of the author's knowledge, there is no work reported in the available technical literature that poses the above question and attempts to answer it. Why is this question important? It is important because the performance of software applications for computer networking, Web services, and cloud computing, with respect to speed, scalability, fault tolerance, and quality of service, is critical. Designing software for these applications involves choosing

A. L. Pereira (✉)
School of Science and Technology, Georgia Gwinnett College, Lawrenceville, GA, USA
e-mail: apereira@ggc.edu

H. R. Arabnia et al. (eds.), *Advances in Software Engineering, Education, and e-Learning*, Transactions on Computational Science and Computational Intelligence, https://doi.org/10.1007/978-3-030-70873-3_34

the right kind of data structure. Choosing the right kind of data structure is crucial because its performance with respect to space (memory utilization, i.e., how much memory is used to store data and what is the overhead) and performance of its operations with respect to time (execution speed, i.e., how fast does the software implementation run) play a significant role in determining the overall performance of the application.

Software developers should know how to compare various data structures based on their memory utilization and performance of their operations. As with most choices in computer programming and design, no method is well suited to all circumstances. A linked list data structure might work well in one case, but cause problems in another case. Also, how does the performance of one data structure scale with data size compared to another data structure? To answer this question, software developers can use asymptotic analysis for a theoretical comparison between the data structures regarding the scalability of their performance. However, software developers should be aware of any practical considerations affecting scalability of performance that may arise from the implementation of the data structures and their operations. They should be able to identify any implementation overhead that may adversely affect practical performance, for example, using data types incorrectly (using double where byte would suffice) or excessive recursion where iteration might be used. In this paper, improvements are proposed to obtain better performance than currently available. The required data for performance evaluation was obtained through software implementation conducted in Java. For the software implementation, Java methods to profile available program memory and execution time of operations were used.

The broader impacts of this work can be in academia and research. The Java Collections framework is increasingly used in undergraduate computer science and information technology courses covering data structures. Students can experimentally verify the practical performance of the data structures using the performance evaluation method described in this paper. Researchers can explore and possibly improve the implementation of data structures in similar frameworks of other programming languages.

The paper is organized as follows. Section 2 contains an explanation of data structures. Section 3 discusses array lists and the asymptotic analysis of their operations. Section 4 discusses linked lists and the asymptotic analysis of their operations and compares them to array lists. Section 5 discusses doubly linked lists and the asymptotic analysis of their operations. Section 6 contains performance evaluation. Section 7 explains stacks and discusses how best to implement them. Section 8 explains queues and discusses how best to implement them. Section 9 contains conclusions and future work.

2 Data Structure

A data structure is an organized collection of data. A data structure not only stores data but also supports the operations for manipulating data in the structure. For example, a classical data structure, array list, holds a collection of data in sequential order and is dynamically resizable (its capacity can increase or decrease to accommodate the amount of data). You can find the size of the array list and store, retrieve, delete, and modify data in the array list. Other examples of classical data structures are arrays, lists, stacks, and queues. A list is a collection of data stored sequentially. Insertion and deletion operations are supported anywhere in the list. A stack can be perceived as a special type of list where insertions and deletions take place only at one end, referred to as the top of the stack. A queue represents a waiting list, where insertions take place at the back (also referred to as the tail) of the queue and deletions take place from the front (also referred to as the head) of a queue.

3 Array List

To explain an array list and its limitations, we will first see how the Java program code implementing it complies and executes. Instantiating a generic class by passing a specific type (e.g., String) to the parameterized type is called generic type invocation, as shown in the first line of the Java program code below.

```
ArrayList <String> list = new ArrayList<>(4);
list.add("Atlanta");
list.add("Chicago");
list.add("Denver");ArrayList list = new ArrayList();
list.add("Atlanta");
list.add("Chicago");
list.add("Denver");
```

Because, the class ArrayList is instantiated with the generic type invocation of String, the compiler first checks if only String objects are added to the array list. If any other object, for example, Integer, is added, then there will be a compile time error. Also, String is declared final, meaning it cannot be extended, and thus cannot have subclasses. A class would be made final if the programmer desires that none of its methods be overridden, so that only those specific behaviors that are desired by the programmer are maintained. But, an array list of a class that is not declared final, can contain objects of the subclasses, because of the compiler support for Polymorphism [5]. In the next step, as shown in the fifth line of code above, the compiler performs type erasure. Meaning, the parameterized type is removed. This can be done because at this point only objects of the class or subclass of the generic type invocation would be stored in the array list. The array list has an instance variable named elementData that can reference an array of instances of the class Object, as shown in Fig. 1. The class Object is the root of the hierarchical class tree

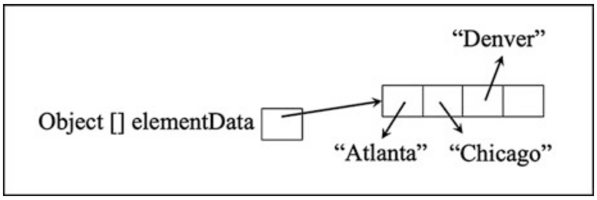

Fig. 1 Array list implemented using an array of the class Object

in Java. In other words, it is the superclass of all the classes. When class ArrayList is instantiated with generic type invocation, an array of Object is also instantiated. Objects of the generic type invocation that are added to the array list are stored in the array.

The performance of the add (or insert) operation, that is, adding (or inserting) an object to the array list, depends upon where in the underlying array the object is being added. If adding objects to the end of an existing sequence of objects (best case), one after another, then it takes constant time to add an object no matter how many objects there are in the array and how many objects are added because the array is indexed. Indexes allow for direct access (also called random access) to memory. This means that it does not matter where in memory you are accessing or storing an object or how many objects are accessed or stored, it takes the same time to access or store a single object, i.e., constant time.

The constant time to add is represented by a time complexity of Big-Theta (1), symbolically noted as Θ (1). Big-Theta notation is used in asymptotic analysis. Asymptotic analysis refers to the study of an algorithm's or operation's performance with respect to resource usage (time complexity, i.e., execution time, and space complexity, i.e., memory size) as the data size grows larger. Asymptotic analysis provides a simplified model of resource usage of an algorithm or operation. Asymptotic notation shows how an algorithm or operation scales when compared to another algorithm or operation. In other words, it shows the rate of growth of cost of an algorithm or operation with respect to time or space as n (the data size grows). Θ is used to indicate that the upper and lower bounds for the cost of an operation are the same within a constant factor.

3.1 Limitations of Array List

If the array list is full, then the elements have to be copied to a new bigger array, and the next element can then be added to the new array. This reallocation is done automatically in an array list. It may not be possible to reallocate if memory is fragmented. The cost of reallocation can be averaged out over many insertions, and the time complexity of an insertion due to reallocation would still be Θ (1).

Insertion at the Beginning of an Array List

For adding to the beginning of the array list (worst case), all elements must be shifted one place to the right before adding the element to the beginning of the array list. Reallocation may be required. The execution time increases linearly with the size of the array list. Asymptotically the time complexity is $\Theta(n)$.

Insertion at a Specified Index in an Array List

Before inserting a new element at a specified index, all the elements at and after the index must be shifted to the right one place and the list size must be increased by 1. On average, half the elements in the array list must be shifted to the right one place when adding (inserting) an element at a particular index. The execution time increases linearly with the size of the array list. Reallocation may be required. Asymptotically, the time complexity is also $\Theta(n)$. This is because, asymptotically $n/2$, the data size to be right shifted if adding in the middle (average case), or, n, the data size to be right shifted if adding at the beginning, does not matter. Linear increase or decrease of the data size (i.e., increase or decrease by a constant factor) does not affect the growth rate of the cost of an operation with respect to its execution time.

Deletion at a Specified Index

To remove an element at a specified index (average case for remove), all the elements after the index must be shifted to the left by one position, and the list size must be decreased by 1. On average, half the elements in the array list must be shifted to the left one place when removing (deleting) an element at a particular index. The execution time increases linearly with the size of the array list. The left shifts are necessary to avoid fragmentation in the array list. Fragmentation adversely affects iteration because the elements are no longer stored contiguously. An array from (which many elements are removed) may also have to be resized in order to avoid wasting too much space, though the cost of resizing can be averaged out over many deletions. Asymptotically, the time complexity is $\Theta(n)$.

Deletion at the Beginning

To remove from the beginning (worst case), all following elements must be shifted to the left one place. The execution time increases linearly with the size of the array list. An array from which many elements are removed may also have to be resized in order to avoid wasting too much space. Asymptotically, the time complexity is also $\Theta(n)$.

Deletion at the End

To remove from the end (best case), the reference to the last element can be replaced by a null pointer. This operation is done in constant time. An array from which many elements are removed may also have to be resized in order to avoid wasting too much space. Asymptotically, the time complexity is $\Theta(1)$.

Asymptotically $n/2$ (data size shifted if adding or removing from middle), n (data size shifted if adding or removing from beginning), or any other linear decrease (e.g., $n/3$, $n/4$, $n/5$, and so on) or increase (e.g., $2n$, $3n$, $4n$, and so on) does not matter. Linear increase or decrease of the data size (i.e., increase or decrease by a constant factor) does not affect the growth rate of the cost of an operation with respect to its execution time. Asymptotic notation of Big-Oh (O), Big-Omega (Ω), Big-Theta (Θ), small-Oh (o), and small-omega (ω) are not the same as the best, worst, or average case. For example, there is a difference between the best, worst, or average case and asymptotic notation like Big-Theta (Θ). For an array list, the best/worst/average case for the add (or insert) operation is addLast/addFirst/addMiddle. They are each $\Theta(1)/\Theta(n)/\Theta(n)$.

4 Linked List

A linked list, as shown in Fig. 2, consists of a chain of objects called nodes, each containing a data element and linked to its next neighbor via a pointer. A node can be defined as a class in Java. The Java class for a node consists of two variables, an element (generic) object reference to data and an object reference to the next neighboring node. Nodes can be non-contiguously stored in memory. The memory size due to storing the references to data and the next node can be considered overhead. Asymptotically, the space complexity of memory overhead in a linked list is $\Theta(n)$. The maximum size, i.e., the maximum number of nodes of a linked list, is constrained by the amount of available heap memory allocated to the program. A linked list is less susceptible to memory fragmentation than array list because nodes in a linked list do not have to be contiguously stored in memory, unlike the object references in an array list.

Data access and reading take longer in a linked list as the number of nodes increases, because on average, in order to access and read an element, all preceding

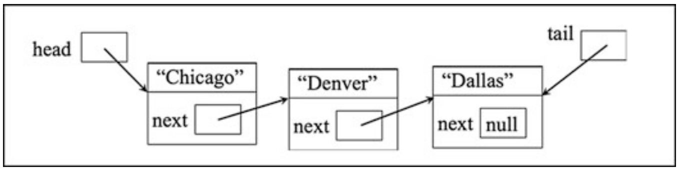

Fig. 2 Linked list of class String containing three nodes

nodes must be traversed to get to the particular node. Asymptotically, the time complexity to access and read a data element in a linked list is $\Theta(n)$.

An array list consists of object references pointing to the data. Object references in an array list are stored contiguously. Array lists require less memory than linked lists for the same number of data elements. Asymptotically, the space complexity of the memory overhead is also $\Theta(n)$. The maximum size, i.e., the maximum number of object references of an array list, is equal to the maximum positive value of the *int* data type, which is (2^{31}–1). However, the maximum size is restricted practically by heap memory allocated to the program. An array list is more susceptible to memory fragmentation than a linked list due to the need for contiguous memory allocation, and in extreme cases reallocation to resize the array list may not be possible due to a memory block of sufficient size being unavailable.

Data access and reading in an array list is done in constant time, because an array list supports random access (direct access). An index (as shown in the Java program code below) is translated to a memory address which allows direct retrieval via the operating system and hardware. Asymptotically, the time complexity to access and read a data element in an array list is $\Theta(1)$.

```
list.get(0) => array[0] => "Atlanta"
list.get(1) => array[1] => "Chicago"
list.get(2) => array[2] => "Denver"
```

4.1 Insertion Operations for a Linked List

There are three implementations of the add operation: addLast(E o), addFirst(E o), and add(int index, E o).

addLast(E o): Creates a new node for the given element and adds the node to the end of the linked list. Takes constant time no matter how big the linked list, because a node needs to be added only at the end without displacing any other nodes before it. Asymptotically, the time complexity is $\Theta(1)$.

addFirst(E o): Creates a new node for the given element and adds the node to the beginning of the linked list. Takes constant time no matter how big the linked list, because a node needs to be added only at the beginning without displacing other nodes after it. Asymptotically, the time complexity is $\Theta(1)$.

add(int index, E o): Creates a new node for the given element and adds the node at a particular position (given by the index) in the linked list. On average, a linked list must be traversed along its nodes (beginning from the first node) in order to reach the point of insertion. Asymptotically, the time complexity is $\Theta(n)$. It takes constant time if a pointer to the last node inserted is maintained. This could be done if a sequence of nodes were inserted one after another. Asymptotically, the time complexity is $\Theta(1)$.

4.2 Deletion Operations for a Linked List

There are three implementations of the remove operation: removeFirst(), removeLast(), and remove(int index).

removeFirst(): Removes the first node of the linked list and returns its reference to the calling program. Takes constant time no matter how big the linked list, because a node needs to be removed only at the beginning without displacing other nodes after it. Asymptotically, the time complexity is $\Theta(1)$.

removeLast(): Removes the last node of the linked list and returns its reference to the calling program. A linked list must be traversed along its nodes (beginning from the first node) in order to reach the last node for removal. Asymptotically, the time complexity is $\Theta(n)$. It takes constant time if the pointer to the node before the last one is maintained. This could be done if a sequence of nodes were removed, one after another. Asymptotically, the time complexity is $\Theta(1)$.

remove(int index): Removes the node at a particular position (given by the index) in the linked list and returns its reference to the calling program. A linked list must be traversed along its nodes (beginning from the first node) in order to reach the node for deletion at the given index. Asymptotically, the time complexity is $\Theta(n)$. It takes constant time if the pointer to the node before the node before the one that was deleted is maintained. This could be done if a sequence of nodes were removed, one after another. Asymptotically, the time complexity is $\Theta(1)$.

5 Doubly Linked List

A doubly linked list contains nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are called a forward pointer and a backward pointer. So, a doubly linked list can be traversed forward and backward. The java class for the node consists of three variables, an element (generic) object reference to data and two object references to the next neighboring nodes. Nodes can be non-contiguously stored in memory. The memory size due to the references to data and the next and previous nodes can be considered overhead. Asymptotically, the space complexity of memory overhead in a doubly linked list is $\Theta(n)$. Data access and reading can be faster than a linked list because traversal can be done in both directions. Asymptotically, the time complexity to access and read a data element in a linked list is $\Theta(n)$.

5.1 Insertion and Deletion Operations for a Doubly Linked List

For a doubly linked list, addFirst is always done in constant time no matter how big the doubly linked list, because displacement of the following nodes is not required. Asymptotically, the time complexity is $\Theta(1)$.

The addLast operation is always done in constant time no matter how big the doubly linked list, because displacement of preceding nodes is not required. Asymptotically, the time complexity is $\Theta(1)$.

For add(index, E o), on average, half the doubly linked list must be traversed in order to reach the point of insertion. Asymptotically, the time complexity is $\Theta(n)$. Asymptotically, the time complexity is $\Theta(1)$, if a pointer to the last node inserted is maintained. This could be done if a sequence of nodes were inserted, one after another.

removeFirst(): Takes constant time no matter how big the doubly linked list, because displacement of the following nodes is not required. Asymptotically, the time complexity is $\Theta(1)$.

removeLast(): Takes constant time no matter how big the doubly linked list, because displacement of preceding nodes is not required. Asymptotically, the time complexity is $\Theta(1)$.

remove(index): On average, half the doubly linked list must be traversed in order to reach the node for deletion at the given index. Asymptotically, the time complexity is $\Theta(n)$. Asymptotically, the time complexity is $\Theta(1)$, if a pointer to the node before the one that was deleted is maintained. This could be done if a sequence of nodes were removed, one after another.

6 Performance Evaluation

Performance evaluation of the add and remove operations for array list and linked list were undertaken on a 2018 MacBook Pro with 2.6 GHz 6-Core Intel Core i7 processor and 32 GB 2400 MHz DDR4 RAM. The profiling software and experiments were implemented using Java version 11.0.1 on Eclipse IDE version 4.10.0. The generic LinkedList class in the Java Collections framework is implemented as a doubly linked list. The time complexity of the addLast operation for both an array list and linked list as discussed in the previous sections is $\Theta(1)$. A time complexity of $\Theta(1)$ means that an operation takes constant time to complete irrespective of the data size. The time complexity does not provide information about the practical execution time of the operation. An operation that is $\Theta(1)$ might take 5 ms to complete when implemented one way and 50 ms to complete for a completely different implementation. Obviously, with respect to execution time, the one that takes 5 ms is the better choice of the two. This kind of information is important when a software developer needs to identify operations in software that perform poorly and improve upon their implementation.

6.1 Performance of Insertion Operations

As shown in Figs. 3 and 4, the practical performance (with respect to execution time and memory usage) of the addLast operation is different for array list and linked list implementations in the Java Collections framework. The memory profile was obtained using the Java Runtime class, and the software execution time was obtained using the Java System class. The profile of the heap memory allocated to the program is as follows: total memory is 512 MB and maximum memory is 8 GB. The objects of integer (wrapper class for the 4-byte *int* data type) were used to store data generated as uniform random integers in the range 0 to 1,000,000, where 0 is inclusive and 1,000,000 is exclusive. The objects are created on the heap. A logarithmic scale is used for n (the data size, i.e., the number of integers) on the x-axis of the graphs.

Figures 3 and 4 show how the execution time and memory size for calling addLast repeatedly (to create an array list or doubly linked list) increases as n increases. For a linked list, a separate node is created for each call of addLast which has greater memory overhead per data point and thus leads to greater total memory allocation compared to an array list. For $n > 200{,}000{,}000$, the memory usage for doubly linked list nears the maximum heap size (8 GB) and for array list nears 70% of the maximum heap size. The performance gap widens to the point where the performance for array list is 8.5 times faster than that for linked list. Also, the execution time and memory usage for array list are little more than doubles when the data size increases from $n = 100{,}000{,}000$ to $n > 200{,}000{,}000$. This is expected. For the same increase in data size, the memory usage for linked list is also little more than doubles; however the execution time is more than triples. This is because, in nearing the maximum heap size, there is greater overhead of growing the heap in the case of a linked list. Also, the Java garbage collector (GC) [6] runs more frequently,

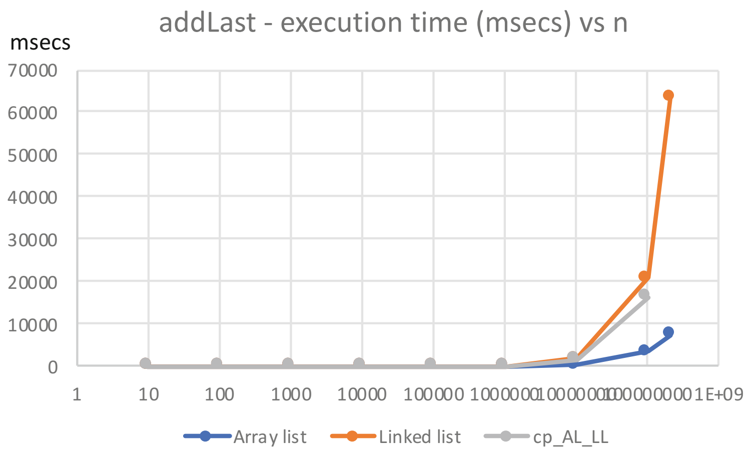

Fig. 3 Execution time vs data size for addLast and cp_AL_LL

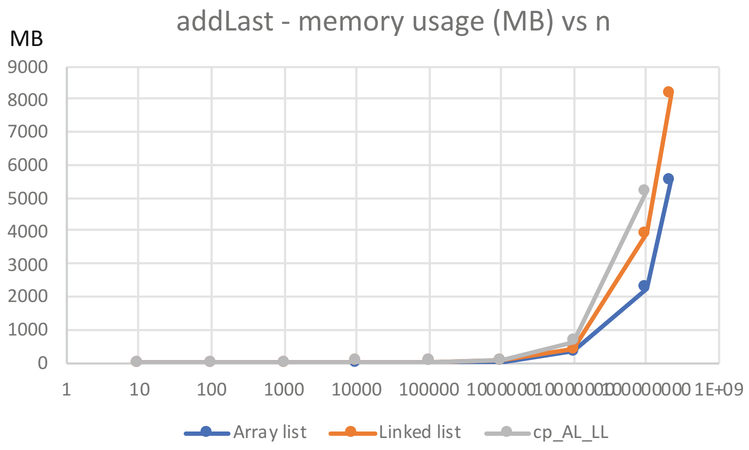

Fig. 4 Memory utilization vs data size for addLast and cp_AL_LL

when more than 70% of the heap is used [7]. GC is a program that deletes objects for which no object reference exists in the program. GC is run automatically and periodically by the Java virtual machine (JVM). It can be invoked using the Runtime class, but is non-deterministic, which means the exact start time of execution cannot be predicted.

The addFirst method shows similar performance to addLast for linked list, but worse performance for array list because of the overhead of right shifting each object reference one place. To construct a doubly linked list in the Java Collections framework that does not exceed the maximum data capacity of an array list, the author proposes that instead of using addFirst or addLast operations for linked list, the software developer should first create an array list and then use the addAll method to create the linked list from the array list. Figure 4 shows that this method (cp_AL_LL) uses more memory because both the array list and linked list are resident on the heap. This also means that the maximum possible data size of the linked list will be half of that which is possible with addLast. However, the proposed method performs better in constructing a linked list and takes 20% less time for $n = 100{,}000{,}000$. If greater data size is desired, then the heap memory size can be increased, which is possible in Eclipse or on the command line when running the program. Using a different data type such as Double (wrapper class for the 8-byte double data type) has negligible effect on the performance.

6.2 *Performance of Deletion Operations*

Figures 5 and 6 show how the execution time and memory size for calling removeLast repeatedly (to delete an array list or linked list) increases as n increases.

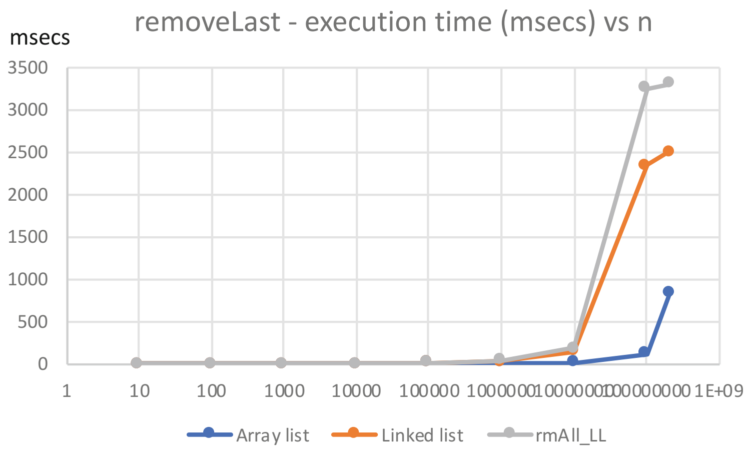

Fig. 5 Execution time vs data size for removeLast and rmAll_LL

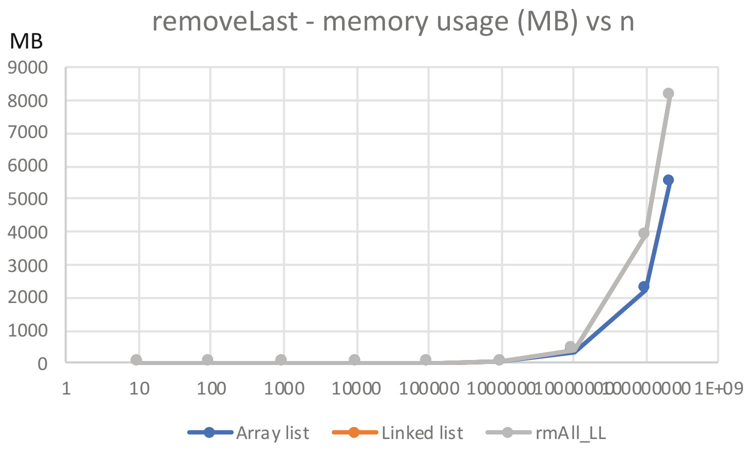

Fig. 6 Memory utilization vs data size for removeLast and rmAll_LL

In Fig. 6, the plot for linked list is hidden because it follows the same trajectory as the plot for the removeAll method for linked list. Also, the plots for the memory usage of linked list and array list in Fig. 6 follow the same trajectory as in Fig. 4. For $n > 200{,}000{,}000$, the performance gap widens to the point where the performance for array list is more than 2.5 times faster than that for linked list. This is because the GC runs more frequently when the heap is greater than 70% its maximum size. Also, prior to deletion of the linked list, the heap is already close to its maximum size; therefore there is no overhead to grow the heap, thus limiting the performance gap. Furthermore, Figs. 3 and 5 show that deleting a linked list is about 25 times faster than creating the link list and deleting an array list is about 10 times faster than creating the array list. It should be noted, however, that the GC did not run during the repeated calls to removeLast and thus the execution time did not include the overheads for object deletion and reduction of the heap size.

As memory usage for linked list reduces to half the maximum heap size for $n = 100{,}000{,}000$, the performance gap widens to the point where the performance for array list is nearly 20 times faster than that for linked list, but the performance for linked list stays about the same. This almost static performance level for linked list even though the memory usage is cut by half is interesting and requires further investigation. The removeFirst method shows similar performance to removeLast for linked list, but worse performance for array list because of the overhead of left shifting each object reference one place. Also, as shown in Figs. 5 and 6, the removeAll method for the LinkedList class shows almost static performance level for linked list even though n and the memory usage are cut by half from nearly the maximum heap size for $n > 200{,}000{,}000$. Again, this is interesting and requires further investigation. Repeated calls to the removeLast method for deletion of the linked list performs about 25% faster than removeAll as the memory usage approaches maximum heap size for $n > 200{,}000{,}000$. Using removeLast also has the advantage of returning the objects containing the data points. The comparatively poorer performance of removeAll is due to its implementation which includes several recursive calls and conditional statements as fail safes. The removeAll method also calls removeFirst, and hence its performance for array list is extremely poor and reduces exponentially. If simply deleting all objects in the array list or linked list is desired, then the author proposes that the clear method be used. The clear method is implemented for the classes ArrayList and LinkedList and simply assigns a null reference to each data object reference. The clear method performs about 60% faster for array list and about 30% faster for linked list when compared to repeated calls of removeLast as the memory usage approaches maximum heap size for $n > 200{,}000{,}000$. The clear method also shows almost static performance level for linked list even though n and the memory usage are cut by half from nearly the maximum heap size for $n > 200{,}000{,}000$. This is interesting and requires further investigation.

7 Stack

A stack can be viewed as a special type of list, where the elements are accessed, inserted, and deleted only from the end, called the top, of the stack. Parsing algorithms used by compilers to determine whether a program is syntactically correct involve the use of stacks. Stacks can be used to evaluate arithmetic expressions. A stack is a last-in-first-out (LIFO) or first-in-last-out (FILO) data structure. It behaves like a stack of books. Objects are pushed (added) to the stack on top of previous objects. Objects are popped (removed) from the top of the stack. Other important applications of stacks are in recursive backtracking and method calls in computer programs.

A linked list can be used to implement a stack because insertion and deletion operations are efficient when done at the front end of the linked list. However, memory overhead is greater than that of an array list. For a doubly linked list, insertion and deletion operations are efficient irrespective of being done at the front end or back end. A doubly linked list supports search better than a linked list, if search functionality is desired. However, memory overhead is greater than a linked list.

The best choice for implementing a stack of objects is an array list because it is faster to add and remove objects, has less memory overhead than a linked list and search, and performs much better than a doubly linked list due to random access. However, for data that consists only of numbers, an array of primitive data type (the type depends on the range of values in the data) has the least memory overhead and best search performance. However, an array is not dynamically resizable and additional implementation will be required to implement a stack that is not of fixed capacity. Furthermore, on systems that allow heap memory size expansion to accommodate extremely large data sizes, the maximum capacity (2^{31}–1) of arrays and array lists may prove restrictive. Also, the size variable for the LinkedList class is of type *int,* thus restricting the maximum positive value to (2^{31}–1). In this case, a doubly linked list should be implemented, because its capacity is restricted only by the maximum heap size. The size variable could be implemented as type *long* (8-bytes) for a maximum positive value of (2^{63}–1). Alternatively, the BigDecimal class could be used to store and manipulate extremely large integers as Strings.

8 Queue

A queue represents a waiting list. A queue can be viewed as a special type of list, where the elements are inserted into the end (tail) of the queue and are accessed and deleted from the beginning (head) of the queue. Queues are widely used in modeling and simulations. They are used in serving requests of a single shared resource (printer, disk, CPU), transferring data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), and interrupt handling in operating systems. A queue is a first-in-first-out (FIFO) data structure. It has the same methods as a stack except that the method push is replaced by enqueue and the method pop is replaced by dequeue. The method enqueue adds an object to the end of the queue and dequeue removes an object from the front of the queue.

A priority queue can be perceived as a special type of queue where data is prioritized for deletion. The data at highest priority is deleted first.

The best choice for implementing a queue is either a linked list or a doubly linked list. Removal operations from the front of a linked list and doubly linked list are efficient. If less memory overhead is desired, then the best choice is a linked list. However, if increased search capability is desired, then the best choice is a doubly linked list. This is because a doubly linked list can be traversed from both directions.

For an array and arraylist, removals from the front end are inefficient because all the following elements must be moved one position toward the front end.

9 Conclusion and Future Work

Insertion and deletion operations of data structures that are asymptotically identical might display severe performance gaps practically. Some of these gaps are identified in this paper and alternative approaches leading to improved performance are proposed. As per the performance evaluation, it was found that a stack can be best implemented using an array list and a queue can be best implemented using a linked list. The stack class in the Java Collections framework uses the class Vector which gives similar performance to an array list. However, Vector is deprecated and class ArrayList is essentially its replacement. Furthermore, to implement a queue the ArrayDeque class can be used because it implements a circular array in which the left shift of elements is eliminated for the removeFirst method. Furthermore, searches are faster because of random access in ArrayDeque. The average case, where data is inserted or deleted from the middle of the array list or linked list, takes about the same time to execute for a single operation. This is expected as right or left shifts are required for an array list and traversal of preceding objects are required for a linked list. Furthermore, the locality of reference for a linked list is far poorer than that of an array list causing greater paging overhead with respect to the CPU cache.

Future work can involve the evaluation of multiple successive calls and maintenance of a reference to avoid multiple traversals in a linked list. Using the Iterator class can speed up the above operation because it maintains references to the nodes in the linked list. Also, for future work, compiler optimization effects on performance can be evaluated. The approach adopted in this paper can be leveraged to evaluate practical performance of data structures implemented in other programming languages such as the C++ standard template library and compare and contrast them with the Java Collections framework.

References

1. Oracle JavaSE Documentation, The collections framework (2018), https://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html. Accessed 23 June 2020
2. Wikipedia The Free Encyclopedia, Data structure (2020), https://en.wikipedia.org/wiki/Data_structure. Accessed 23 June 2020
3. Wikipedia The Free Encyclopedia, Linked list (2020), https://en.wikipedia.org/wiki/Linked_list#Linked_lists_vs._dynamic_arrays. Accessed 23 June 2020
4. C.A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Second edn. (Prentice Hall, New Jersey, 2001)

5. Y.D. Liang, *Introduction to Java Programming and Data Structures, Comprehensive Version*, Eleventh edn. (Pearson, New York, 2017)
6. Oracle Learning Library, Java Garbage Collection Basics (2012), https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html. Accessed 23 June 2020
7. IBM Knowledge Center, Heap Sizing Problems (2020), https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/mm_heapsize_problems.html. Accessed 26 June 2020