Vasu Jogani

Mr. May

Computer Science III

20 Novemeber 2015

Efficiency Comparision for Various Lists

Successful programs need to be capable enough of holding heavy load cycles and loop, efficient and quick at responding to demand and feasible at manipulating. This report would compare all the pros and cons associated with various types of List. There are thousands of different varieties of work for a computer program to do daily, but the results it produces depends on the implementation. Every task has its requirement, and it is accomplished by using different processes. Sometimes the program is efficient, sometimes it is not, but it totally rests on the type of work that program would do. Most tasks have one program common, Lists. There are primarily four different types of basic List that could be used for a range of tasks.

One of the most common types of lists is ArrayList. It is subdivided into two different types: one with size equal to the number of elements in the list and other with the size of the list larger than the number of elements in the list. List with size == 0 is useful when the data collection and analyzing is done on small scale. When one has more “leisure” time, meaning quickness is not required, this list would be preferable because this is measurably slower in processing than the other ArrayList. This list best fits once size, and could be extended whenever single data is added. Since the size is equal to the amount of the data in the list, it would consume little space in the memory, and thus would have more space available for other tasks. One of the prime examples of this list usage could be a daily inventory. On the first day, an empty ArrayList is created, and at the end of the day, the list is updated with the all the sales and products details.

Since the task of adding or removing products from the list is not often, the program would utilize little space in the memory, so the rest of the memory could be used for other tasks throughout the day. When the get() is concerned to retrieve the data while in business, the ArrayList uses index based system unlike other types such as Linked List. This makes it manageable and fast to search and gets the data at the specific index.

Another ArrayList, better than the first, is the list with size > 0. Similar to the conventional ArrayList, this is the index-based list. However, unlike size == 0 ArrayList, where the size is equal to 0 at the beginning, this list starts the list with a considerably large size for multiple purposes. First and foremost, it makes the process of adding more feasible most of the times. Since the list would already be bigger and would already have booked enough space in the memory, add() at the last index would be most efficient 99% of the time. The only time it would extend the time it takes would be when the list would be equal the number of elements in the list, and then the new list would be created and old data has to be copied. While inserting in the middle of the list, then, unlike the first list where the list was copied, the data from the back of the list till the index are shifted back one unit and the data is inserted into that spot. In some worst cases, this will take as long as the other ArrayList while, in other cases, this might also take as long as one loop only. Comparing both ArrayList, size() and set() do not change. One typical example of the usage of this list is the cashier at any store. Once the seller knows how many approximate customers would shop in one day, the list with size > 0 could be created, where all the values are initial set to nothing. As the day progress, each entry of the customer could be added quickly to the list since it is not creating a new list during every single entry. In the rare case, if the list gets filled up, then the new larger list can be created, and the data from the old can be copied to it. Removing data would equally easy since the list would be shifted one index backward after the position of the removed. Even though the company makes a larger list with empty spots, the size() returns the actual number of the entries in the list and ignores the empty spots. The only restriction it has is that it would consume a lot of space in the memory whether it's used or not. So for computer capable and task for constant adding and subtracting, this List would be perfect.

Another legendary type of lists is LinkedList. It is also subdivided into two different types: one is the LinkedList and another is the Double LinkedList. This is another popular List type used in the computer programs to complete various task. Instead of using a traditional index-based list, LinkedList is based on chain of Nodes. The List contains nodes that are a generic objects and have a certain value and have a link method getNext() which returns the next node linked to it. There is no initial limit for the list. Only concern is keeping track of head and tail of the list. Otherwise each node is connected to other and last node would be null. This is a classic example of volatile list, where the size does not matter initially and no memory is used initially. Any node can added at anywhere, just like the ArrayList, but the memory it takes is volatile. There is no fix spot for the node unlike the ArrayList where the memory is already booked according to the size. Each node works individually, and will be spread out over the whole memory. This list would be useful when adding something, not in sorted-order, in any size of lists. For example there is a system that records that entry made by the people, LinkedList could be used so that new node can be added quickest. Since each new entry would be made at the end, only tail needs to be change. And because there is no recreation of the new list or copying of the old one, this would add anything at the end very quickly. Using list for volatile task, this would yield best performance. But if the size is known, the using this for long time would eventually consume more memory. So it depends what the task is.

DoubleLinkedList is another category in the LinkedList. This also contains a chain of Nodes which are not just linked with getNext() but also are linked to each other with getPrve(). This non-index based list is also volatile and works best for the any size lists too. Unlike the LinkedList, the DoubleLinkedList only needs one loop to add anything in between. Since this list allows to use getPrev(), the node previous to it can be accessed and can be used to set it to the next node. Switching back and forth, these functions make it easier to add in between. This can be used when there is a need for adding information in the between the list. For example a register in which when something is added then it is added in between. Unlike shifting or copying the list, this list only breaks the chain and adds the node and call setPrev() and setNext(). But the memory consumed by this list would be more than the others. So, like the other entire list there are some pros and some cons, but this works best if the memory is the major concern and task is to add something in middle.

Further efficiency comparisons were made using the Big-O notation and measuring the time it takes to do a certain task in different lists. The chart below shows the Big-O notation for all methods from all the types of the List. The methods with 0(n) is where it loops and takes more time whereas 0(1) is where the methods work efficiently. There also a close comparison for the times between the methods of the different list. The chart below it represents that.

Data Comparision

Notation	ArrayList *Size==0*	ArrayList *Size>0*	LinkedList	Double LinkedList
Size()	0(1)	0(1)	0(n)	0(n)
Add()	0(n)	*Best Case:* 0(1) *WorstCase:*0(n)	0(n)	0(n)
Add(index)	0(n)	0(n)	0(n)	0(n)
Set()	0(n)	0(n)	0(n)	0(n)
Get()	0(n)	0(n)	0(n)	0(n)
Remove()	0(n)	0(n)	0(n)	0(n)

The chart above shows the comparision in the Big-0 for all the methods in all the types of Lists.

	Add()	*Remove()*	*Add(index)*	*Get()*	*Set()*
ArrayList *Size>0*	0.001057097	0.151217419	0.186091665	2.61864E-4	3.15356E-4
DoubleLinkedLIst	0.107883287	0.213702141	0.105785888	0.32310343	0.324012801
LinkedList	0.10971789	0.104436448	0.113747547	0.323106229	0.325220734

The chart above shows the comparsion in the time it takes to complete the method with 10000 objects.