Measuring Algorithms Performance in Dynamic Linked List and Arrays

Majlinda Fetaji[1], Mirlinda Ebibi[1], Bekim Fetaji[1]
[1]South East European University, Contemporary Sciences and Technologies, Tetovo, Macedonia

Abstract - **The focus of the research is on investigating the organization and structure of a list of data in order to find more efficient algorithmic solution. The aim of the realised experiment was to analyze, compare and measure the efficiency of algorithms for searching, insertion and deletion in a list of elements. An experimental case study is done through implementing the list as: 1) Array Data Structure and as 2) Dynamic Linked List Data Structure and measuring the performance. The contribution of the study is based on the realized experimental case study analyses, its insights and recommendations for improving the efficiency of algorithms for searching, insertion and deletion in a list of elements. The recommendations, and insights gained from the realized measurements are presented and discussed.**

Keywords. **Dynamic Linked List (DLL), array, algorithm, data structure.**

1. Introduction

In the majority of software applications executed in computing devices like PC's, and especially today in tablets, smart-phones and other mobile devices and as they become more powerful in processing, the amounts of processed (stored and retrieved) information are getting larger. Representing information is fundamental to computer science [1].

In algorithmic solution of a problem the choice of data representation is not a simple process and is a reasonably difficult one. It is usually determined on the programmer preference which is not a proper approach.

The main research focus of this research study is how to organize and structure a list of data to find an efficient algorithmic solution based on the most frequent performed basic operations.

An investigation is conducted to determine the structuring of list of elements (data) also taking into consideration the actual problem characteristics, the sample (sorted or unsorted) and the size of the input data set, etc.

2. Measuring algorithms efficiency

According to [2, 3, 4] an algorithm is a step-by-step well-defined computational procedure for performing some task in a finite amount of time, that takes some values as input, manipulates the data following the prescribed steps and produces some values as output. That information is a collection of data about the actual problem processed by the algorithm, and are organized and accessed in a systematic way [3] in data structures. Algorithm provides the logic or the steps to find the solution; while the data provide the needed input values [3]. Programs are executable implementations (in a programming language) of the algorithmic solution of a given problem and the used data structures to store input and output data. Once a correct algorithm is designed, an important step is to determine the efficiency of the algorithm, how much time or space resources the algorithm will require [1,2].
An algorithm which finds a correct solution only is not sufficient. It might perform long time and be inefficient which becomes more obvious by increasing the size of input data set. An algorithm that solves a problem but requires a year and/or requires a gigabyte of main memory is not useful [2]. A straightforward way of solving a problem may not be the best one.

There are often more then solutions for a problem. How do we choose between them? The one that runs in shortest time? Or, the most efficient one? What is an efficient algorithm? How to measure the efficiency?

A solution is said to be efficient if it solves the problem within the required resource constraints which include the total space available to store the data and the time allowed to perform each subtask or if it requires fewer resources than known alternatives [1]. The mathematical instrument to measure the efficiency (performance) is the analysis of an algorithm. It is used to: i) estimate resource consumption of an algorithm, and ii) for comparation of relative costs of different algorithms for a given problem. The choice of the most efficient algorithm among variant solutions is based on the measurements of 1. Space complexity (memory resourses an algorithm uses), and 2. Time complexity (time resourses or the time an algorithm runns). In the most cases, only time resourses are considered since the techniques used to determine memory requirements are a subset of those used to determine time requirements [1].

The running time becomes an important issue; a natural measure of "goodness," which mean computer solutions should run as fast as possible [3]. The running time of an algorithm is affected by many factors [2, 3, 5]: i) the hardware (the processor's speed, the clock rate, memory, disk) and software platform (the operating system, the programming language, the compiler), iv) the size of the input data set, etc. The running time is most important resource to analyze while considering the size of input data set [2]. To be able to classify some data structures and algorithms as "good," we must have precise ways of analyzing them [3]. There are two ways to estimate or analyze the time complexity of a program: 1) experimental study and 2) theoretical study. The first way to analyze the running time of an algorithm is realized by measuring the time the algorithm runs for different sized input data sets. Measuring how long the algorithm runs for each input samples with given size, we can visualize the growth rate of the running time (how it increases by increasing the input size).

The second way to estimate the time complexity or the running time T(n) of an algorithm is to determine how many of each primitive instructions (such as arithmetic operations add, multiply, etc., assignment, comparation, invoking a method, etc.,) is executed. Instead of measuring the time execution of primitive operations (each executes in a constant time), assume a one time unit per operation, we determine the number of primitive operations as a function of the size of the input indicated by n.
This machine model is widely accepted whereas the above factors which influence the runnning time are not considered. Changing the hardware/ software platform influences the running time T(n) by a constant factor, but does not influence the growth rate of the running time T(n) [3]. That model is normal computer, in which instructions or primitive operations are executed sequentially and takes exactly one time unit to do each (simple instruction) and infinite memory is assumed [2]. Even though these assumptions about the model could be real problems for computer applications, this is a general way of analyzing the running times of algorithms that [3]: i) Considers all possible inputs, ii) Can compare efficiency of any two algorithms independently from the hardware/software platform, and iii) No need to implement and execute algorithms.

By using this method each algorithm is assigned a function T(n) which determines the dependency of the running time from the input data size n or it's growth rate. When analyzing the efficiency of an algorithm, usually worst case of an algorithm execution is analyzed, when the algorithm performs worst or performs maximum possible number of primitive operations[6]. This methodology is the asymptotic algorithm analysis which determines the running time in big-Oh notation. It gives the upper bound of the growth rate of the running time function.
T(n) is O(f(n)) if T(n) is asymptotically less than or equal to f(n), or lim T(n) /f(n) = 0, for n-> . If this function, by increasing the size of the input, has a slow growth rate, then we can say that the algorithm is efficient. Furthermore, if we have to choose among variant algorithms, we'll chose the one with the running time with a slowest growth rate. In our study, to overcome disadvantages of both methods [3], we use both methods for being most accurate.

3. Experimental case study

Experimental studies have been done focused on analyzing the efficiency of algorithms for storing and retrieving data respectively: search, insert and delete operations in the two basic data structures for implementing a list of elements: arrays and dynamic linked lists. It is realized by measuring the running time for the same input data set for respective algorithms using both data structures. The running time of an algorithm can be observed by running the algorithm on different sized input data set and storing the values of the time the algorithm starts and ends. The measurements can be taken in a precise way in C++ by using the C standard library function called clock() (and clock_t defined in header file time.h), registering the start time and end time of the algorithm, performing it 10000 times and then finding the difference (finish - start))/10000, which is the time an algorithm runs in seconds.

Performed are several tests on many different input data sets of various sizes. Then in (x,y) coordinate system, we can visually represent the dependency of the running time (the y coordinate) on the size of the input data set (the x coordinate). The algorithms to be tested in a) an array and b) a dynamic linked list, are 1. the linear search algorithms, 2. the insert algorithm, and 3. the delete algorithm.

Taking the size of the input starting from 1, 4, 16, 64, …, 262144, altering by 4 factor. By running the three algorithms, for both data structures, the implemented program will calculate the running times of the respective algorithms for each data input. The input samples will be randomly generated data in the program. The measured running times are presented in the following Figure 1 to Figure 8 as diagrams of the growth rates of above stated algorithms' running times:

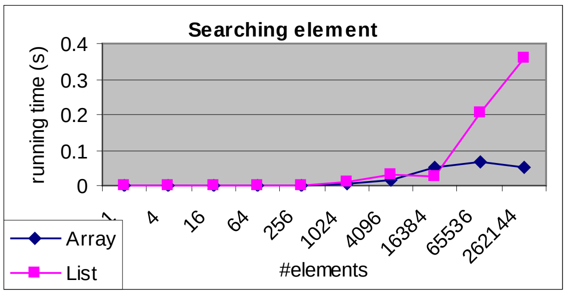

Figure 1. Searching element in a list

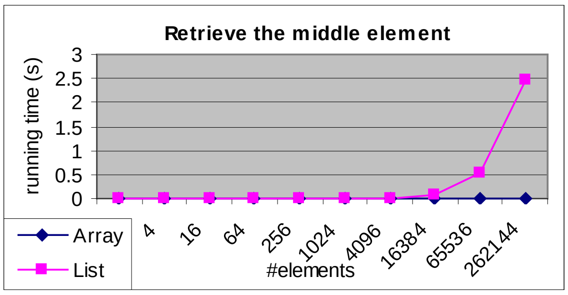

Figure 2. Retrieving an element

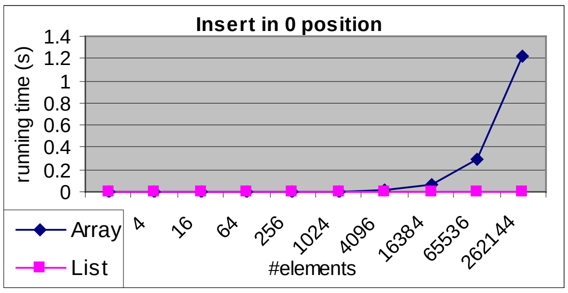

Figure 3. Insert element at 0 position in a list

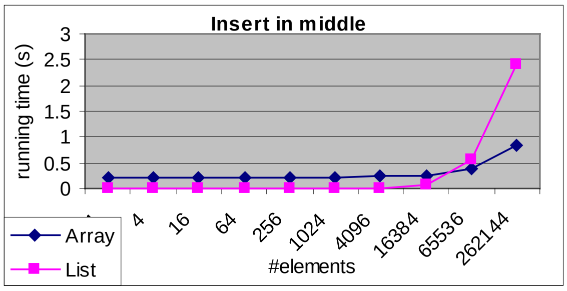

Figure 4. Insert element in the middle in a list

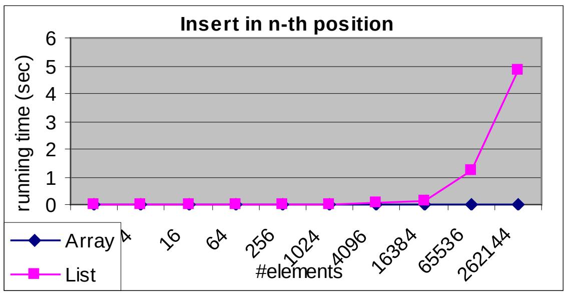

Figure 5. Insert element in n-th position of a list

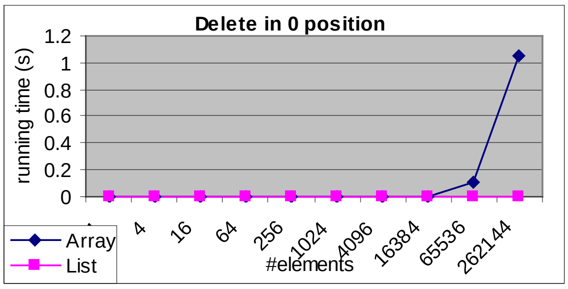

Figure 6. Delete element at 0 position in a list

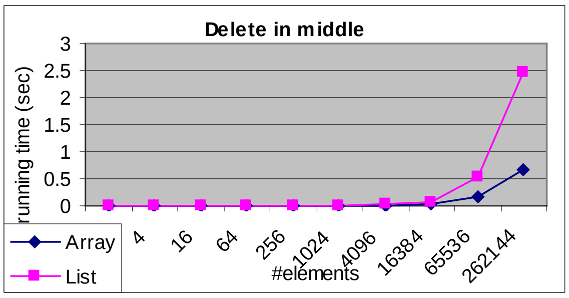

Figure 7. Delete element in the middle of list

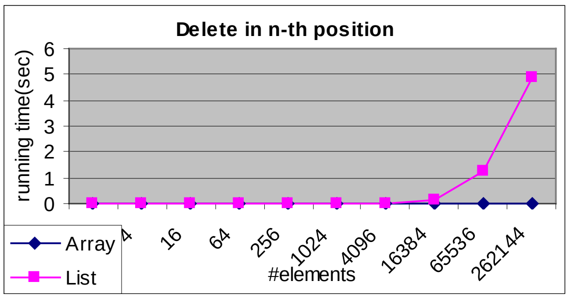

Figure 8. Delete n-th element position in list

4. The concept

Three algorithms are theoretically analyzed by counting the primitive operations as a function of the input size n.Then algorithms are asymptotically analyzed to find the upper bound of the running time-***T(n)*** which is a function ***f(n)*** asymptotically bigger then ***T(n).*** It is done by eliminating all constants and lower order terms from the function ***T(n),*** which is expressed in terms of *big-Oh* notation T(n)=O(f(n)).

5. Linear search

When applied a linear search of an element x in an array a[0,...n-1] of n elements (the size of the input), then in the worst case x might be the last element, or not in the list then the number of comparisons is equal to the number of elements n, then by counting the number of primitive operations:

T(n)= 1+ n+1 + n + n*{1+1} + 1
▲ ▲ ▲ ▲ ▲
(init. i to 0) (testings) (incr.) (n iterat.) (return)
T(n)=4n+3=O(n), f(n)=n- the upper bound.

We can say that the running time of a linear search algorithm in an array of elements by growing the number of elements grows worst as a linear function.

Retrieving an element at the k-th position in an array, only one return operation is performed and a constant time is required. The running time is T(n)=O(1), a constant time.

In DLL, in the worst case the element x might be the last element or not in the list, then the running time T(n)= n iterations * constant number of primitive operations, by eliminating constants T(n)=O(n).

Retrieving an element at the k-th position in a DLL, requires k primitive operations to find the position k which in worst case k=n, the number of elements. Then the running time is T(n)= n iterations * constant number of primitive operations; by eliminating constants we have T(n)=O(n), f(n)=n – the upper bound is a linear function.

If using a sorted sample of data, when use an array, a binary search can be performed starting from n sized array, then n/2, n/4,…, 1, 0. As we start with experiment for array of size 1, 4, 16, 64, … 262144, let us take for example size n=64. The binary search continues searching in one of the halfs, so new array size will be n=32, 16, …, n=1; we can write as 26, 25, …, 20. We have: 2i=n / log =>log 2i= log n => i*log 2= log n => i = log n/ log 2= log n; number of iterations i of performing some primitive operations (constant number which is eliminated from T(n)) is log n, so we have T(n)=O(log n); A pretty good time, grows slowly. In DLL, a binary search cannot be performed.

6. Insertion

When inserting an element x in an array a[0,...n-1] of n elements (the size of the input), in position 0, then all elements from the first one at the position 0 to the last one at the position n-1 must be shifted for one position to the right, so we have n primitive operations of shifting. By counting the number of primitive operations:

T(n)= 1+ n+1 + n + n*{4} + 1 + 1
▲ ▲ ▲ ▲ ▲ + ▲
(init. i to 0)+(testings)+(incr.)+(n iterat.) + (position the new element at the pos. 0 + increment n)

T(n)=6n+4=O(n), f(n)=n- the upper bound We can say that the running time of an algorithm for insert operation at the position 0 in an array of elements by growing the number of elements n grows worst as a linear function. Furthermore, when inserting elements, usually more insertions are done, if n then we have O(n*n)=O(n2). This is really bad.

Inserting an element at the k-th position in an array, requires n-k-1 primitive operations for shifting, 1 for assigning a[k], 1 for incrementing n, where the worst case is k<=n/2, n the number of elements. Then T(n)= 1+n-k+n-k+1+(n-k-1)*{4}+ 1; n-k <=n, for each k<=n/2...(1)

Proof:, true for n=1; k<=0; 1-0=1<=1

Suppose true for n, true by induction for n+1 for each k<=(n+1)/2, k-1/2<=n/2;

n+1: n+1-k=n-k+1<=n+1, so T(n)=6(n-k)+4 <=6n+4=O(n), a linear function.

if k=n, the running time is:

T(n)= 1+ 1+ 1 = 2 = O (1), a constant function. If using a sorted sample of data, the case of inserting an element x in an array when x is greater then all elements in the array, the algorithm performs n primitive operations to find the position, then 1 for setting the new element, 1 for incrementing n, then the running time is T(n)=n+1+1=O(n). Or, when x is less then all elements in the array, the algorithm performs 1 operation to find the position (ususally testing the value of x if is less then a[0], n primitive operations for shifting n elements to the right, 1 for setting the new element, 1 for incrementing n, then the running time is T(n)=1+n+1+1=O(n).

We have totally different results in a dynamic linked list (DLL). Inserting element at the position 0 in a DLL requires only 5 primitive operations for creating the node, setting the element's value and linking the new element's node at the position 0 in the DLL, which means we have a running time T(n)=O(1). By counting the number of primitive operations, we have: T(n)= 1+1+1+1+1=5. We can say that the running time of an algorithm for insert operation at the position 0 in a DLL of elements grows worst as a constant function by growing the number of elements n, with constant growth rate.
It is the best growth rate.

Inserting an element at the k-th position in a DLL, requires 5 primitive operations as inserting in position 0, without considering the operations to find the position k in the list. Then the running time of insert in k-th position in a DLL is T(n)=O(1). In the worst case k>n/2 requires k primitive operations for finding the position k, and constant time for inserting a node. Then the running time is T(n)= k + 5<= n+5= O(n), a linear function.

If using a sorted sample of data, the worst case of inserting an element x is when x is greater then all elements in DLL and considering finding the position which is n+1 primitive operations, the running time is O(n).

T(n)= 1+ n +1+1+1 + 1 = n+5= O(n), a linear function. If x is less then all elements, the best case, we have insert in position 0 and T(n)=O(1)..

7. Deletion

When deleting an element x in position 0 in an array a[0, ..., n-1] of n elements (the size of the input), all elements from the second one at the position 1 to the last one at the position n-1 must be shifted for one position to the left, so we have n-1 primitive operations of shifting, one operation of decrementing the n number of elements. So, T(n)=O(n), where f(n)=n; By counting the number of primitive operations:

T(n)= 1+ n + n -1 + (n-1)*{4}+ 1
▲ ▲ ▲ ▲ ▲
(init.i_to_0)(testings)(incr.)(n-1iterat.)(decr. n)
T(n)=6n+4=O(n), f(n)=n- the upper bound.

The running time of an algorithm for delete operation at the position 0 in an array by growing the number of elements n grows worst as a linear function. Furthermore, when deleting elements, usually more deletions are done, if n then we have O(n*n)=O(n2). This is really bad.

Deleting an element at the k-th position in an array, requires n-k-1 primitive operations for shifting, 1 for decrementing n, which in worst case k<=n/2, then T(n)= 1+n-k+n-k+1+(n-k-1)*{4}+ 1<=6n+4=O(n), a linear function.
if k=n, the running time is:
T(n)= 1+ 1+ 1 = 2 = O (1),

Different results are in a dynamic linked list (DLL). Deleting element at the position 0 in a DLL requires only 3 primitive operations for deleting the node, positioning new pointer to the node at the position 0, shifting the beginning of the list to the next node in the DLL, and freeing the memory caught by the pointer; which means we have a constant running time. By counting the number of primitive operations, we have: T(n)= 1+1+1=3. T(n)=O(1).
We can say that the running time of an algorithm for delete operation at the position 0 in a DLL of elements grows worst as a constant function by growing the number of elements n. DLL is the best choice in this case.
Deleting an element at the k-th position in a DLL, requires 3 primitive operations as deleting in position 0, without considering the operations to find the position k in the list. Then the running time of delete in k-th position in a DLL is T(n)=O(1). The worst case when considering the operations to find the position k in the DLL, and k=n, then the running time is T(n)= n + 3 (prim. op.) = O(n), a linear function. In the worst case k>n/2 requires k primitive operations for finding the position k, and constant time for deleting a node. Then the running time is T(n)= k + 3<= n+3= O(n), a linear function.In a sorted sample of data, deletion is same in both data structures: arrays and DLLs.

8. Conclusion

As conclusion the contribution of the study is based on the realized experimental case study analyses, its insights and recommendations for improving the efficiency of algorithms for searching, insertion and deletion in a list of elements. The main contribution of this paper is to determine which Data Structure is more appropriate to use in particular cases categorized by more frequent operations performed and characteristics of the datasets to be used.The findings from experimental study show the following recommendations based on the most frequent operations needed to be performed:

i) for searching an element in a list, the growth rate of the running time T(n) is similar when use both data structures. It is obvious in the diagram of the running time’s growth rate shown in Figure 1. We can use both data structures, no priority. If we search an element to retrieve from the given position k, then an array has priority, the growth rate is constant function while when use DLL it grows rapidly by increasing n. It is obvious in the diagram of the running time’s growth rate shown in Figure 2.

ii) for inserting an element at the position 0 in a list, the growth rate of T(n) when use array is much higher then when use a DLL, we should use a DLL data structure. While inserting at the position k=middle in a list, the growth rate of T(n) in both data structures increases similarly; but as the position tends to n=the input size, when use array the growth rate of T(n) becomes almost constant function while when use a DLL, T(n) grows rapidly. It is obvious in the diagrams of the running times’ growth rates shown in Figure 3, 4 and 5. We suggest to use an array in cases when inserting at position k>=middle and use a DLL for k<=n/2 as much as needed.

iii) for deleting an element in a list we have similar results as for inserting an element. So, it is concluded that, the It is obvious in the diagrams of the running times' growth rates shown in Figure 6, 7 and 8.

Findings from the theoretical study show similar results and give following recommendations, based on the most frequent operations needed to be performed:

i. for searching an element in list, we can use both data structures, no priority; while when retrieving an element from the given position k, an array has priority. If data are sorted then for searching element an array has priority.

ii. for inserting an element at the position 0 or k<=n/2 in a list, we should use a DLL data structure; and for inserting an element at the position k->n in a list, we should use an array. We must mention here that in DLL, the search of the position k takes certain number of operations while in an array, a direct access to the position k is on hand. If data are sorted then we have additional linear search to find the position in both cases.

iii. for deleting an element results are same as with insertion.

iv. arrays are better for cases opposite to cases DLLs are better.

Recommendations based on advantages of DLLs over arrays and vice versa; and additional theoretical studies, are:

i. we should use arrays only when we know the number of elements, in other cases a memory waist or under estimation of size happens. In cases we do not know the size better use DLLs. DLLs allocate memory for a new element dynamically when needed. Even though additional memory for the pointer is needed.

ii. we should use arrays when retrieving an element is the most frequent operation, the direct access to array elements makes retrieving an element almost instantly, while in DLLs it must be realized by a linear search and might take n (input size) primitive operations at worst.

iii. we should use arrays if sorted data are available, and searching is the most frequent operation, we can apply binary search (it is performed in O (logn) time) and direct access to arrays' elements.

In order to increase efficiency of the program, structuring of data can be made based on: 1. the most frequent operations and 2. the size and the sample of input data (sorted or unsorted).

All of the analyses data is provided in the appendix.

References

[1] Clifford A. Shaffer, (1997). A Practical Introduction to Data Structures and Algorithm Analysis Third Edition (Java Version), Publisher: Prentice Hall; Java ed edition (1997), ISBN-13: 978-0136609117.

[2] Mark Allen Weiss, (1996) Data Structures and Algorithm Analysis in C. Publisher: Addison Wesley; 2nd edition ISBN-13: 978-0201498400.

[3] Michael T. Goodrich & Roberto Tamassia, (2006), Data Structures and Algorithms in Java, 4th edition, Publisher: John Wiley & Sons, Inc., ISBN: 9780471738848.

[4] Nicklaus Wirth, "Algorithms + Data Structures=Programs", Prentice-Hall Inc., 1976, ISBN: 0-13-022418-9.

[5] Gonet G. H., and Baeza, Yates R., (2006) Handbook of Algorithm and Data Structures in C and Pascal, 2nd Edition, Publisher: Addison Wesley, ISBN 0-201-41607-7.

[6] Dusan Malbaski, Aleksandar Kupusinac, Srdjan Popov, The Impact of Coding Style on the Readability of C Programs, TTEM, Vol.6. No.4. ISSN: 1840-1503, 2011.

[7] Yeguas, E ,Joan-Arinyo, R, Luzon, MV, Modeling the Performance of Evolutionary Algorithms on the Root Identification Problem: A Case Study with PBIL and CHC Algorithms, EVOLUTIONARY COMPUTATION Volume: 19 Issue: 1 Pages: 107-135 DOI: 10.1162/EVCO_a_00017, 2011

Corresponding author: Bekim Fetaji
Institution: South East European University, Macedonia.
E-mail: b.fetaji@seeu.edu.mk