Data Structures Comparison for Element Deletion Including Stack and Queue

MAJA SAREVSKA
Faculty of Informatics,
European University Skopje,
REPUBLIC OF NORTH MACEDONIA

Abstract: - This paper presents the analysis of element deletion in different data structures, like linked lists, binary search trees, stacks, and queues. For random data sequence, we build the linked list and binary search tree and we compare the element deletion procedure. We find the much more stable performance of the binary search tree than that of the linked list. Additionally, we analyze the stack as a complex data structure and its basic operations. We present the element deletion with the basic operations for array stack implementation and we note the limitations. Also, we present the element deletion problem for a queue as a complex data structure, implemented with an array. We present the element deletion with the basic queue operations and we point out the limitations.

Key-Words: - Array, Linked List, Binary Search tree, Stack, Queue.

Received: April 25, 2024. Revised: October 26, 2024. Accepted: November 27, 2024. Published: December 27, 2024.

1 Introduction

One of the most important aspects of any programming language is Data Structures (DS). Data are organized and stored in DS to be efficiently used for data operations. DS are arranged data in a particular way, saved in the memory so it can be retrieved to be used later, [1].

An array DS is a basic concept in programming, it is a collection of items of the same data type stored in contiguous memory locations, [2]. This DS is efficiently used in programming for manipulating and organizing data with access to any array element using indices. An additional fundamental DS in programming is a Linked List (LL), which is made of a set of nodes, where each node is represented with data and reference or link to the next node. This DS efficiently adds and deletes elements in the LL. A Binary Tree (BT) is a tree DS where each node can have at most two children, and these two nodes are referred to as the left child and the right child. BTs have many applications in computer science, like data storage and retrieval. Also, they can be used to implement algorithms such as searching, sorting, and graph algorithms, [3]. A Binary Search Tree (BST) is a special type of binary tree in which the left child has a value less than the node's value and the right child has a value greater than the node's value. This property provides efficient application of various data operations like deleting, searching, or inserting elements in the tree and it is called the BST property, [4], [5]. A stack is a linear data structure where elements are stored in the LIFO (Last In First Out) principle where the last element inserted would be the first element to be deleted, [4], [5]. A stack is an Abstract Data Type (ADT), that is popularly used in most programming languages. It is named stack because it has similar operations as the real-world stacks, for example − a pack of cards or a pile of plates, etc. Stack is considered a complex data structure because it uses other data structures for implementation, such as Arrays, Linked lists, etc. A queue is a linear data structure where elements are stored in the FIFO (First In First Out) principle where the first element inserted would be the first element to be accessed, [4], [5]. A queue is an Abstract Data Type (ADT) similar to a stack, the thing that makes a queue different from the stack is that a queue is open at both ends. The data is inserted into the queue through one end and deleted from it using the other end. Queue is very frequently used in most programming languages.

The goal of this paper is to analyze the element deletion in different DS, like array, LL, stack, queue, and BST. Namely, BST can efficiently delete and insert elements same efficiently like in LL for random data sequences. This quantity is represented by program steps in the programming language C. Section 2 defines illustratively and with programming code the appropriate DSs. Section 3 represents the array and BST comparison for

element search and LL and BST element deletion comparison results for random data sequences. Section 4 represents the analysis for stack and queue implemented with arrays. And in Section 5 some concluding remarks are given.

2 DSs in the Programming Language C

An array is a set of items of the same data type, stored in contiguous memory locations that may be accessed efficiently with indices, [4]. If the array is sorted then applying the appropriate algorithm we can do an element search in a very efficient and fast way. Figure 1 presents the array illustratively. Figure 1(a) presents the element deletion. First, the element should be found, then removed from the array, and afterward, all upper elements must be moved down for one position.

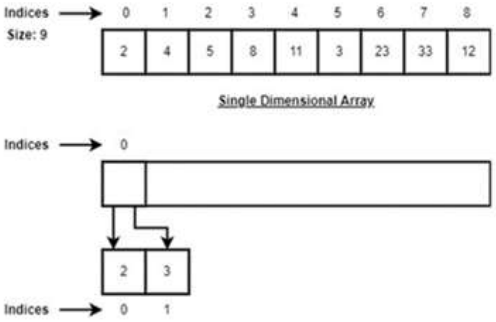

Fig. 1: Array DS, illustratively and programming code for definition in programming language C

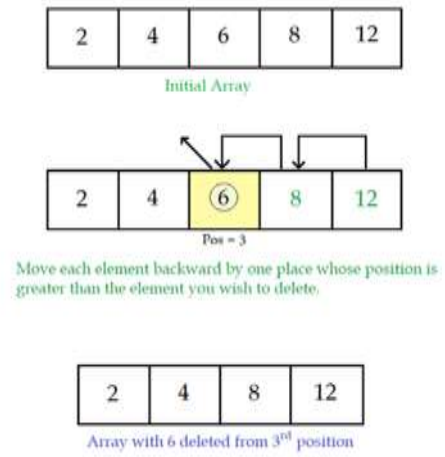

Fig. 1(a): Array DS, element deletion

LL DS is a collection of elements called nodes, where each node is represented with data and reference or link to the next node. In this DS we can efficiently add and delete elements, [4]. Figure 2 illustrates the LL DS. Figure 2(a) presents the illustration of element deletion and the formation of new links while deleting the element.

LL DS is very convenient for adding and deleting a node, in our analysis we will focus only on element deletion, although the conclusion may be easily driven in the case of adding an element.

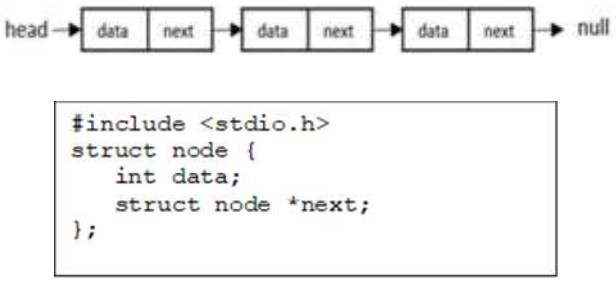

Fig. 2: LL DS, illustratively and programming code for definition in programming language C

Fig. 2(a): LL DS, element deletion

BST is a special type of BT, where the value of the left child is less than the value of the parent node and the value of the right child is greater than the value of the parent node, [4], [5]. Figure 3 illustrates the BST DS. Figure 3(a) presents the element deletion in the BST and node rearranging after element removal.

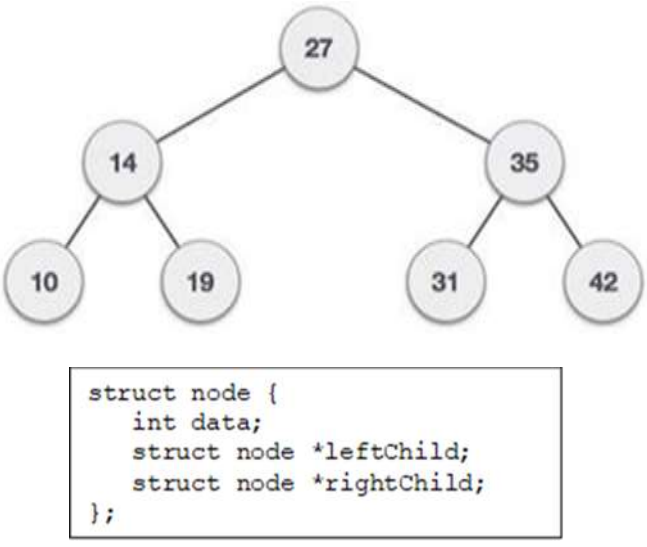

Fig. 3: BST DS, illustratively and programming code for definition in programming language C

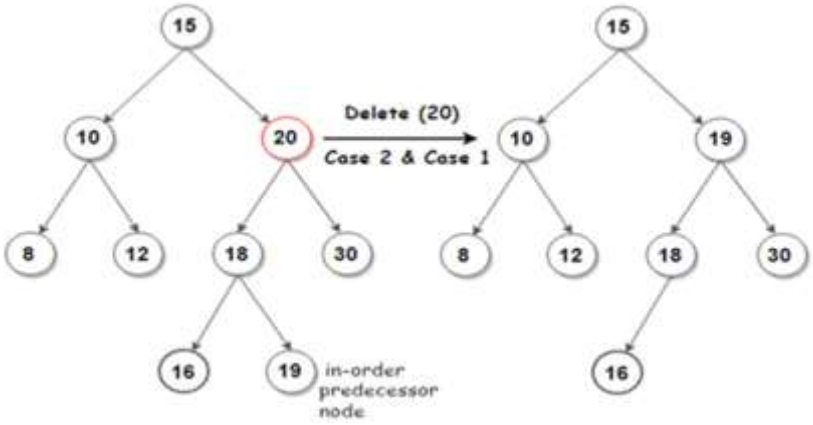

Fig. 3(a): BST DS, element deletion, [6]

A stack allows all data operations at one end only, [4]. At any given time, we can only access the top element of a stack. A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement a stack using arrays, which makes it a fixed-size stack implementation.

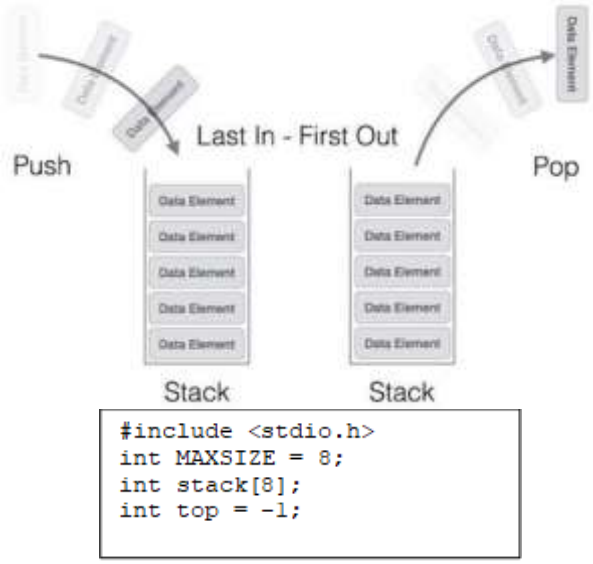

Fig. 4: Stack representation and implementation in C

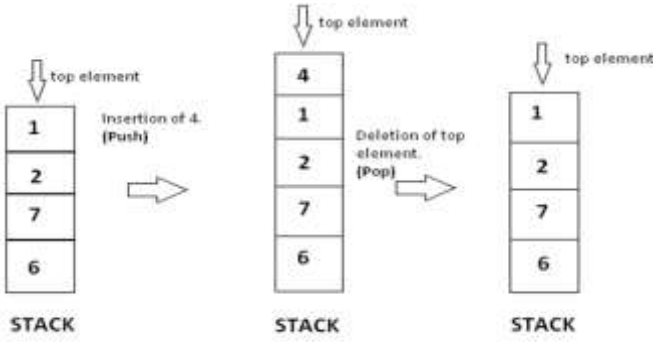

Fig. 4(a):–Basic operations in a stack

Stack operations are usually performed for initialization, usage and, de-initialization of the stack ADT.

The most fundamental operations in the stack ADT include push(), pop(), peek(), isFull(), and isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the stack.

Stack uses pointers that always point to the topmost element within the stack, hence called the top pointer. Figure 4(b) presents the code for the push operation, and Figure 4(c) presents the code for the pop operation. The idea is to delete arbitrary elements in the stack using these operations, which will be explained in the next section

```
int MAXSIZE = 8;
int stack[8];
int top = -1;

int isfull(){
   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

int push(int data){
   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack
is full.\n");
   }
}
```

Fig. 4(b): push() operation in a stack

Similar to the stack ADT, a queue ADT can also be implemented using arrays, linked lists, or pointers, we will implement queues using a one-

dimensional array, [4]. Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.

```
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;

int isempty(){
   if(top == -1)
      return 1;
   else
      return 0;
}

int pop(){
   int data;
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data,
Stack is empty.\n");
   }
}
```

Fig. 4(c): pop() operation in a stack

The most fundamental operations in the queue ADT include: enqueue(), dequeue(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the queue.

Queue uses two pointers − front and rear. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing). The enqueue() is a data manipulation operation that is used to insert elements into the stack. The following algorithm (5b) describes the enqueue() operation more simply. The dequeue() is a data manipulation operation that is used to remove elements from the stack. The following algorithm (5c) describes the dequeue() operation in a simpler way. The idea is to delete an arbitrary element in the queue using only these operations, which will be explained in the next section.

```
#include <stdio.h>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
```

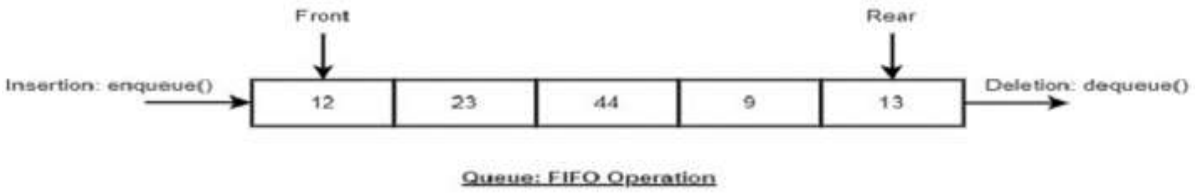

Fig. 5: Queue DS and its implementation in C

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
bool isFull(){
   return itemCount == MAX;
}
bool isEmpty(){
   return itemCount == 0;
}
void insert(int data){
   if(!isFull()) {
      if(rear == MAX-1) {
         rear = -1;
      }
      intArray[++rear] = data;
      itemCount++;
   }
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
bool isFull(){
   return itemCount == MAX;
}
bool isEmpty(){
   return itemCount == 0;
}
int removeData(){
   int data = intArray[front++];
   if(front == MAX) {
      front = 0;
   }
   itemCount--;
   return data;
}
```

Fig. 5(b,c): -enqueue() and dequeue() oparations in a queue

3 Array, Linked List, and Binary Search Tree

The simulation experiment is done on the following data example, sorted array: {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55, 66}. For the array, we use the Binary Search algorithm and we count the steps in the program. For BST we shuffle the elements, build the tree, and search for the appropriate element, and we also count the programming steps. We make thousands of trials and estimate the mean number of steps. Then we compare the steps, by presenting them in the chart in Figure 6.

For example, we may notice that the number of programming steps to find the element 66 in the array is 5 while for the BST is 3. Overall the values are comparable.

For the programming code in C, the reader is kindly asked to contact the authors. In the program, we define function shuffle, which shuffles the elements in the sorted array. This is done because in general the items in the sequence are random when we build the LL or BST. Then we define a function find(item), that does the binary search for the element in the sorted array. Namely, if the element that should be found is greater than the middle element then we continue the search in the upper sub-half array. If not in the lower one. Iteratively we repeat this procedure while we find the element. Then we define the function insert(data), to build the BST by adding the elements (nodes) one by one. We define the function search(data). If the element that should be found is smaller than the node value we continue to search in the left sub-tree, else we search in the right sub-tree. We repeat this iteratively until we find the element. Then we define the main() function where we do the binary search, count the procedure steps while array binary search, shuffle the elements, build the BST, do the element search in the BST, and count the procedure steps.

To check the complexity of BST element deletion for random data we first must generate the data. We generate a random integer sequence of length C with a maximal value of MAX. For the programming code in C, the reader is kindly asked to contact the authors. In the program, we define function shuffle, which shuffles the elements in the sequence. This is done because in general the items in the sequence are random when we build the LL or BST. We define the function insert(data), to build the BST by adding the elements (nodes) one by one. We define the function search(data). If the element that should be found is smaller than the node value we continue to search in the left sub-tree, else we search in the right sub-tree. We repeat this iteratively until we find the element. Then we define a function minValueNode, which is necessary when deleting the node. The node with minimal value in the right sub-tree should replace the deleted node, and other nodes should be rearranged. We define the function deleteNode for node deletion. We defined the function insertFirst(data) to build the LL by adding the nodes one by one. Next, we define the function delete to delete a node from the LL and reorganize the links. Then we define the main() function where we shuffle the elements, build the BST and LL, do the element search in the BST, and delete the desired element, do the element search in the LL, and delete the desired element, and we count the procedure steps. We do the shuffling of the data sequence 1000 times and we estimate the average programming steps and its standard deviation. The results are presented in Figure 7. The axis is presented by concrete data integer values rather than their position in the sequence. We may notice more stable results for the BST compared to the LL. This is because there is a need for more programming steps for element search in LL compared to the BST.

We performed a more wide analysis, using different data sequence saizes and different maximal integer value in the data. Here we present only the small part of the results, But we can mention that the time complexity is increased as we attempt to delete the deeper element in the BST which are actually the last elements in the data sequence while building the tree. Also, we concluded that the value of the maximum in the data sequence has no influence on the complexity.

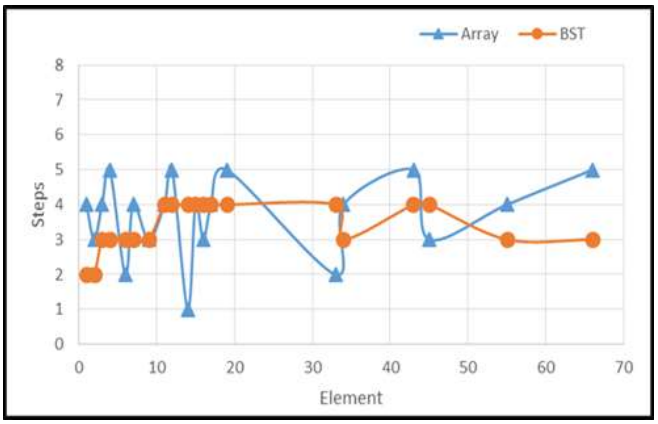

Fig. 6: Steps comparison between Array and the BST

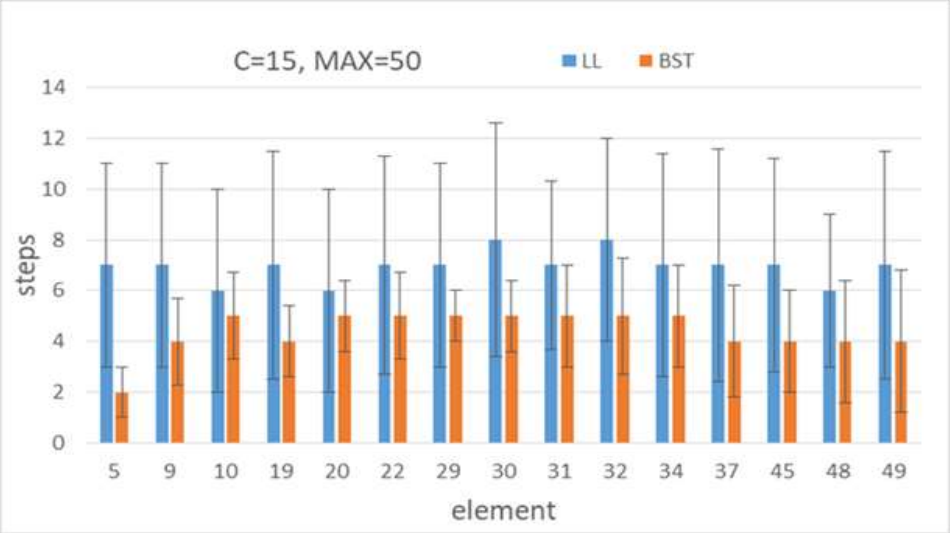

Fig. 7: programming steps with standard deviation

```
void delete(int data, int *comparisons)
{
   int i, reference,sdata;
   sdata=pop();
   (*comparisons)++;
   while (data!=sdata)
     {
      push1(sdata);
      sdata=pop();
      (*comparisons)++;
     }
   reference=(*comparisons);
   for (i=0;i<reference-1;i++)
     {
      sdata=pop1();
      push(sdata);
      (*comparisons)++;
     }
}
```

Fig. 8:Code for the element deletion in a stack

```
void delete(int data, int *comparisons)
{
   int i, queuedata, reference;
   queuedata=removeData();
   (*comparisons)++;
   while (data!=queuedata)
     {
      insert(queuedata);
      queuedata=removeData();
      (*comparisons)++;
     }
   reference=(*comparisons);
   for (i=0;i<C-reference+1;i++)
     {
      queuedata=removeData();
      insert(queuedata);
      (*comparisons)++;
     }
}
```

Fig. 9: code for the element deletion in a queue

4 Stack and Queue Element Deletion with Array Implementation

When we are talking about element deletion from a stack we mean deletion of an arbitrary element, using only stack operations as defined in Section 2. We use array implementation as explained there. For that purpose, we need an additional stack1 for which we will define the operations push1() and pop1() same as for the main stack but on the additional array. Let's say we want to delete the nth element in the stack. First, we pop() all n-1 elements from the top of the stack, one by one, and push them into the additional stack with push1() one by one. Then we remove the nth element from the stack. Afterwards, we return the n-1 elements from stack1 with push() operation into the stack The code of the function is presented in Figure 8.

As we may conclude the time complexity is proportional to C, the length of the stack and the space complexity is proportional to 2C (doubled size). All this makes array implementation very inconvenient for element deletion for stacks.

We also implement a queue with an array as explained in Section 2. To delete an arbitrary element in the queue using standard operations in the queue we do not need an additional queue as we may use the same for element location. Let's say we want to delete the nth element. Then we remove n-1 from the front of the queue one by one and we insert them at the back of the queue. When we reach the nth element we just remove it. Afterward, we remove the rest of the C-n elements from the front and insert them at the back of the queue. As we may notice the time and space complexity for element deletion for a queue with an array is proportional to the data size C. The programming code is presented in Figure 9. This makes array implementation very inconvenient for element deletion for queues.

5 Conclusion

We presented the analysis of element deletion in different data structures, like linked lists, binary search trees, stacks, and queues. For random data sequence, we presented the linked list and binary search tree and we compared the element deletion procedure. We found a much more stable performance of the binary search tree than that of the linked list. Additionally, we analyzed the stack and its basic operations. We presented the element deletion with the basic operations for array stack implementation and we noted the limitations. Also, we presented the element deletion problem for a queue, implemented with an array. We presented the element deletion with the basic queue operations and we pointed out the limitations. The future analysis should focus on stack and queue implementation with linked lists and pointers, where we expect better performances in the sense of time and space complexity for the element deletion problem.

References:

[1] Rubi Dhankhar , Sapna Kamra , Vishal Jangra,"Tree concept in data structure", *2014 IJIRT,* Vol. 1, Issue 7, ISSN: 2349-6002.

[2] Sthuti J, Namith C, Shanthanu Nagesh, "Data Structures and its Applications in C", *International Research Journal of Engineering and Technology (IRJET),* Vol. 08, Issue 4, Apr. 2021.

[3] Dimitrios Samoladas; Christos Karras; Aristeidis Karras; Leonidas Theodorakopoulos; Spyros Sioutas, "Tree Data Structures and Efficient Indexing Techniques for Big Data Management: A Comprehensive Study". PCI '22: *Proceedings of the 26th Pan-Hellenic Conference on Informatics*, November 2022, pp.123–132.

[4] Tutorialspoint. Data Structures and Algorithms (DSA) Tutorial , [Online]. https://www.tutorialspoint.com/data_structures_algorithms/index.htm (Accessed Date: October 15, 2024).

[5] GeeksforGeeks. Data Structures Tutorial, [Online]. https://www.geeksforgeeks.org/data-structures/ (Accessed Date: October 15, 2024).

[6] Techie Delight, [Online]. https://www.techiedelight.com/ (Accessed Date: October 15, 2024).

Contribution of Individual Authors to the Creation of a Scientific Article (Ghostwriting Policy)

The author contributed in the present research, at all stages from the formulation of the problem to the final findings and solution.

Sources of Funding for Research Presented in a Scientific Article or Scientific Article Itself

No funding was received for conducting this study.

Conflict of Interest

The author has no conflict of interest to declare that is relevant to the content of this article.