INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING & COMPUTER SYSTEMS (IJSECS)
ISSN: 2289-8522 e-ISSN: 2180-0650
VOL. 6, ISSUE 1, 52 – 63
DOI: https://doi.org/10.15282/ijsecs.6.1.2020.6.0069

ORIGINAL ARTICLE

Balanced Linked List (BaLL): A Modified Linked List for Efficiently Maintaining a Sorted Sequence of Data

Mutaz Rasmi Abu Sara

Department of Computer Science, Taibah University, Madinah al Munawara, Saudi Arabia.

ABSTRACT – In this paper, a modified version of linked list, called *balanced linked list* (*BaLL*), has been proposed. It performs better than standard *sorted linked list* (*SoLL*) both theoretically and experimentally for maintaining a sorted sequence of data. The modification in BaLL than SoLL is to maintain a special middle node instead of head, and to maintain the remaining nodes in two sorted lists of almost equal size. Three most basic operations, search, insert and delete, have been considered in BaLL and have been compared with the same set of operations in SoLL. For these three operations, BaLL performs theoretically 50% better and experimentally around 50% better than SoLL.

ARTICLE HISTORY
Received: 14 Jan 2021
Accepted: 25 April 2021

KEYWORDS
Array
Sorted Sequence
Linked List
Balanced Linked List
Sorted Linked List

INTRODUCTION

Linked list is a basic and primitive data structure. It is widely used in computer programming. It consists of a sequence of nodes, where each node contains data and a link to the next node. It is a sequential data structure in the sense that the data appear one after another in a sequence of nodes. Its data are accessed and traversed from the beginning to the end. Linked list can be doubled, meaning that each node has a second pointer which points to the previous node. A doubly linked list can be traversed in both directions following the two pointers. Standard textbooks on algorithms and data structures, such as (Cormen et al., 2009), (Weiss, 2011), (Skiena and Steven, 2020), (Goodrich and Tamassia, 2014), (Sedgewick and Wayne, 2011) contain this basic information about linked list.

In a linked list, data can be added or deleted by adding or deleting nodes. There is no limit on how many data can be added to a linked list, as it depends upon the availability of the memory. As long as there is free space in memory, nodes can be added. This is an advantage of linked lists compared to arrays, where the number of elements that can be added is fixed beforehand. Again, standard textbooks, such as (Cormen et al., 2009), (Weiss, 2011), (Skiena and Steven, 2020), (Goodrich and Tamassia, 2014), (Sedgewick and Wayne, 2011), can be seen for differences between linked list and array.

Adding and deleting data from a linked list can be in any sequence. For example, an element can be added at the beginning of the list, at the end of the list, or after a particular element in the list. Adding elements in a linked list without following any sequence results into an unsorted linked list.

Linked list can also be maintained so that it contains the data in a sorted fashion. Such a linked list is called a sorted linked list. In such a sorted linked list each data is added by first searching the appropriate position of the new element in the list, and then inserting the new element so that after insertion the list remains sorted. See (Carraway, 1996), (Shene, 1996), (Verma and Kumar, 2013), (Sanu, 2019), (Koganti and Yijie, 2018) for different approaches for maintaining sorted linked list (both single and doubly linked list.)

Adding and deleting elements from a linked list is easier compared to an array. Because, in an array, in order to insert an element in a particular position, all elements in the right side should be shifted to make a space for the new element. Similarly, after deleting an element from an array, all elements in the right should be shifted left to fill out the space created after deletion. In contrast, addition and deletion are easier in linked lists, as only the pointers of a very few nodes need to be updated. See (Cormen et al., 2009), (Weiss, 2011), (Skiena and Steven, 2020), (Goodrich and Tamassia, 2014), (Sedgewick and Wayne, 2011) for comparisons between linked list and array.

Sorted linked list can have many advantages over an unsorted linked list. From a sorted linked list, a sorted sequence can be printed simply by traversing the elements from the starting node to the end node, and without first doing an expensive sorting operation on the list. The time taken by this printing will be linearly proportional to the number of elements in the list. Searching an element in the list is also faster in a sorted linked list, as it is not necessary to go until the end of the list to search the element. Starting from the head, the search can continue as long as the elements are smaller than the target element. Consequently, the time required for such a search will be much smaller than searching in an unsorted list. Similarly, deleting an element will also be faster, as searching the element that is to be deleted will take much shorter time.

There are several abstract data types (ADT) which are based on arrays or trees and can maintain sorted sequence of data. For example, sorted array (Snyder, 2006), (Hijazi and Qatawneh, 2017), binary search trees (Knuth, 1971), (Eberl et al., 2020), (Nipkow et al., 2020), heaps (Haiming et al., 2017), B-Trees (Cormen et al., 2009), (Jie and Pengfei, 2017), and AVL Trees (Cormen et al., 2009) are some of such data structures. In sorted array, search can be performed in O(log

*CORRESPONDING AUTHOR | Mutaz Rasmi Abu Sara | ✉ mabusara@taibahu.edu.sa

n) time by binary search. Insert and delete can be performed in O(n) time. For the other data structures, all of search, insert and delete can be performed in O(log n) time (Cormen et al., 2009), (Weiss, 2011), (Skiena and Steven, 2020), (Goodrich and Tamassia, 2014), (Sedgewick and Wayne, 2011).

In contrast, there are some ADT that are based on linked lists and that maintain a sorted sequence of data. Skip list (Goodrich and Tamassia, 2014) is such a data structure. Skip list can perform search, insert and delete in O(log n) time. However, skip list is comparatively more complicated to implement and needs a lot of extra space.

Sanders et al. in their book (Sanders et al., 2019) have reviewed a collection of useful and efficient data structures that have their keys stored in the nodes, but the raw data stored in a sorted linked list. Such data structures include binary search trees, (a,b)-trees, and red-black trees. The use of such data structures is in implementation of breadth-first heuristics, database indexing, and sweep line algorithms, among others.

There are also some approaches for sorting elements in a linked list. Carraway (Carraway, 1996) and Shene (Shene, 1996) proposed different approaches to perform standard sorting algorithms on single and doubly linked lists. The standard sorting algorithms that they considered are bubble sort, selection sort, shell sort, merge sort, quick sort, and tree-based sorting algorithms (Cormen et al., 2009), (Weiss, 2011), (Skiena and Steven, 2020), (Goodrich and Tamassia, 2014), (Sedgewick and Wayne, 2011), (Haiming et al., 2017), (Jie and Pengfei, 2017), (Zhi-Gang, 2020), (Srinivasan et al., 2017), (Kowalski et al., 2020), (Gautam, 2020), such as heap sort (Cormen et al., 2009), (Weiss, 2011), (Skiena and Steven, 2020), (Goodrich and Tamassia, 2014), (Sedgewick and Wayne, 2011). Carraway (Carraway, 1996) also proposed a new sorting algorithm, called *sediment sort*, that he claims to perform better than the standard sorting algorithms on linked lists. Both Carraway (Carraway, 1996) and Shene (Shene, 1996) provided several experimental results on comparison of these sorting algorithms while implemented on linked lists.

In an unpublished work, Verma and Kumar (Verma and Kumar, 2013) presented a sorting algorithm, called *list sort*, that works on linked lists. Their idea is to maintain several smaller linked lists of size small (say, 10 nodes) to high (say, 10000 nodes). Each list is sorted. The lists themselves are also sorted by their first values and their last values. Elements are inserted into the smaller lists first if they have empty spaces. Occasionally lists are merged to get a bigger single linked list. The authors claim that their sorting algorithm can perform better than the standard sorting algorithms, such as quick sort and merge sort.

(Sanu, 2019) presented a technique for implementing binary search in linked lists. Recall that binary search is performed in a sorted array (Cormen et al., 2009), (Weiss, 2011), (Skiena and Steven, 2020), (Goodrich and Tamassia, 2014), (Sedgewick and Wayne, 2011), (Kleinberg and Tardos, 2005). Because a binary search needs to access the array elements by indices, it is not possible to implement binary search in a linked list in a straightforward way. In (Sanu, 2019), the author maintained an extra array of pointers to each node of a linked list. Thus, for finding the middle element in a binary search, the middle index of the extra array is used to find the middle node in the linked list. This technique is continued for the left and right half of the linked list as the binary search moves on. Some other works on searching in the linked list can be found in (Koganti, 2019).

PROBLEM STATEMENT, CONTRIBUTION, AND METHODOLOGY

All the previous works on the sorted linked list can be broadly categorized into two types. First, the works that deal with maintaining a sorted sequence of data. They additionally maintain data structures (such as binary search tree or an extra array) on top of a sorted linked list. Second, the works that sort the linked list itself. They also maintain additional data structures (such as layers of linked lists) on top of the original sorted link list. Therefore, both the types can be considered as complicated to implement and analyze.

Based on the above observation, in this paper, the following problem is considered: Design and implement a simple version of single linked that can maintain a sorted sequence of data without incorporating any additional data structures. Moreover, analyze and compare the outcome of this new implementation with the most basic form of the single linked list both theoretically and experimentally.

In order to achieve the above goal, in this paper, a simple approach is followed. The most basic form of a single linked list has been modified to work efficiently for maintaining a sorted sequence of data. The structure of the proposed linked list, called *balanced linked list* (*BaLL* for short), is little different than a *sorted single linked list* (also called *SoLL* for short). It has a middle node that contains the median of all data. It has two pointers: upper and lower. Upper pointer points to a sorted linked list and works as a head for that linked list. The data in that linked list are sorted from high to low. Similarly, the lower pointer points to and works as a head of another single linked list, which is sorted from low to high.

Most common parameters for measuring the performance of a data structure like arrays and linked lists are measuring the running time of the basic operations. The most basic operations in this type of data structures are search, insert and delete (Cormen et al., 2009), (Sanders et al., 2019), (Malik, 2017), (Imran, 2020), (Wengrow, 2020). There are other operations such as finding maximum and minimum, merging two lists, range queries, initial building of the data structure, etc. In another way to say, scientists see how quickly these three operations can be performed in those data structures. In this paper, in order to measure the performance of BaLL and to compare with SoLL, the three operation searches, insert, and delete have been considered. Their running time has been measured by counting the total time taken by each of those operations running a large number of times with random input.

In this paper, it has been shown that theoretically BaLL performs 50% better than SoLL while maintaining a sorted sequence of data and performing search, insert and delete on them. Both BaLL and SoLL have been implemented with

search, insert and delete, and have been compared for a variety of data. Experimental results show that the performance of BaLL is better than SoLL by an amount that varies from 20% to 60% and remains mostly around 50%.

The rest of the paper is organized as follows. In the Preliminaries section, we provide some preliminaries related to computing the running time of algorithms that are performed on data structures. Section SoLL and BaLL reviews SoLL and describes BaLL in detail, and describes how search, insert and delete are performed on them. Theoretical Performance of BaLL section shows why theoretically BaLL can perform 50% better than SoLL. The Experimental Performance of BaLL section presents the experimental results. The Conclusion section concludes the paper with some future works.

PRELIMINARIES

It is well known that the *running time* of an algorithm is measured as the number of *main steps* that are executed in the worst case when the algorithm runs. The definition of "main steps" can vary for different types of algorithms. For algorithms such as search, insert and delete in data structures such as in linked lists and arrays, the main steps are the *number of comparisons*. In these algorithms, there are other operations, such as assignment and pointer updates. But the number of those operations are constant and are very small compared to the number of comparisons. Therefore, the number of comparisons works as a dominating factor for determining the running time of those algorithms and are ultimately considered as the only measure of the running time (Cormen et al., 2009), (Weiss, 2011), (Skiena and Steven, 2020), (Goodrich and Tamassia, 2014), (Sedgewick and Wayne, 2011), (Kleinberg and Tardos, 2005).

For such algorithms, the number of comparisons is denoted as O(f(n)). It means that the number of comparisons is no more than cf(n), where c is a positive constant and n is the number of elements in the data structures.

While O(f(n)) captures the performance of almost all the common algorithms, a more precise analysis can be done by looking into the constant c inside O(f(n)). For example, two algorithms may have the same running time of O(f(n)), say O(n), but the constant inside O(n) may vary. One algorithm may have an actual running time which is no more than 2n, while the other algorithm may have the running time that is no more than 4n. Then by the O() notation, both algorithms have the same running time of O(n), but in reality the first algorithm is twice faster than the second algorithm.

With the above observation as the main focus, recently, Hasan et al. (Hasan et al., 2020) have proposed an extended O() notation, where they bring forward the hidden constant c and write the running time as O(cf(n)). By their proposal, running time of 2n and 4n for two different algorithms will be represented as O(2n) and O(4n), so that their difference is easily visible by looking into the O(cf(n)) notation. The authors also supported their proposal with examples from some analysis of some famous algorithms such as quick sort and median finding (Hasan et al., 2020).

The above concept will be useful in our results in this paper too for analyzing SoLL and BaLL. For all operations that we perform on SoLL and BaLL, the worst-case running time is the same as O(n). However, a closer look into the constant c inside the O(n) will reveal that the actual running time of those operations in SoLL are about n/2, whereas those for BaLL are about n/4. That means, theoretically BaLL is about 50% faster than SoLL. See Theoretical Performance of BaLL Section for details.

SOLL AND BALL

In this section, SoLL and BaLL will be explained in detail along with their operations search, insert and delete. Note that SoLL is the most basic form of a single linked list where elements can be searched, inserted, and deleted by maintaining the sorted sequence. (Only the name "SoLL" has been given anew in this paper.) The necessary algorithms for these operations in SoLL can be found in any standard textbook, such as in (Cormen et al., 2009), (Weiss, 2011), (Skiena and Steven, 2020), (Goodrich and Tamassia, 2014), (Sedgewick and Wayne, 2011), (Sanders et al., 2019) in chapters related to linked list. In particular, the book by Sanders et al. (Sanders et al., 2019) provides an entire chapter (Chapter 7) on data structures and algorithms that are used based on linked lists for maintaining a sorted sequence of data in the linked list. The basic idea is to use an additional data structure, which they call *navigation data structure*, for facilitating the search, insert and delete so that after any operation the list remains sorted. The navigation data structure can be a binary search tree, an (a,b)-tree, a red-black tree, etc. See Figure 1.

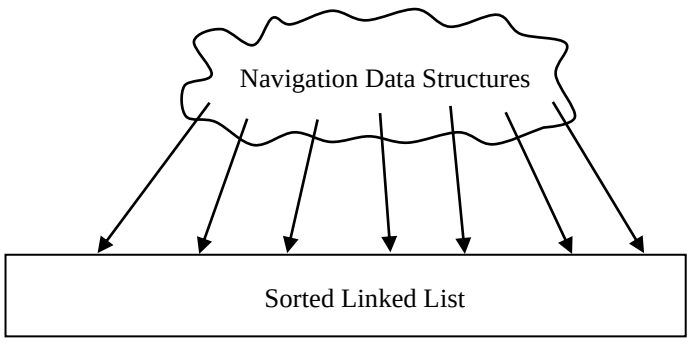

Figure 1. Additional data structures used to maintain sorted sequence in linked list (Sanders et al., 2019).

There are other approaches too to maintain a sorted sequence in SoLL. The approach followed by Carraway (Carraway, 1996) is to use the new sorting algorithm sediment sort in the linked list for maintaining the sorted sequence. He presented his algorithm for doubly linked lists. Similarly, Verma et. al. (Verma and Kumar, 2013) used the new sorting algorithm list sort for maintaining a sorted sequence. Shene in his paper (Shene, 1996) mentioned that other well-known sorting algorithms can also work for maintaining a sorted sequence in linked lists. He compared the traditional sorting algorithms such as selection sort, merge sort, quick sort, bubble sort, and tree-based sorting algorithms along with the sediment sort.

In either of the above-mentioned approaches, there are additional complexities of work for arranging additional data structures and their algorithms. Because of that, the simplicity of the standard linked list is lost. This leads to one of the motivations of this paper, which is to follow a simplistic approach and to work on a standard single linked list. The description of SoLL and its algorithms provided below can be achieved from the standard textbooks that have been mentioned before. However, their descriptions are precisely presented here for the completeness of the paper.

BaLL in this paper is a new concept. Unlike SoLL, there is no previous literature to the knowledge of the author that explicitly works on BaLL, especially on the operations search, insert and delete in BaLL. Therefore, the description of BaLL as well as its algorithms as provided below will be new and have almost no scope to be analyzed with respect to previous work at this time.

SOLL

Please refer to Figure 2 and Algorithm 1. Each node in a SoLL has two fields: the *data* and the *next* pointer. Next points to the next node in the list. For the last node, the next points to null. There is a pointer called *head* that points to the first node of the list. Initially the list is empty and the head points to null. When SoLL is not empty, its nodes are sorted from low to high by their data.

For searching an element x in a SoLL, it starts from the head and gradually moves to the next nodes by following the next pointers one after another until x is found (for a successful search) or there is no more node in the list (for an unsuccessful search).

For inserting a new element x into the list, a new node is created with its data as x and with its next pointer as null. If x is the very first element of the list, then the head is made to point to this new node. Otherwise, a correct position is searched for x so that after the insertion the SoLL remains sorted. Then the new node is inserted in that position by updating the next pointer of the new node as well as the next pointer of the node after which the new node will be inserted.

For the deletion of an element x from a SoLL, first x is searched within the SoLL. If x is found in the list, then the node is deleted by changing the next pointer of the previous node of x (saved during the search) to point to the node after x (which may be null). See Figure 2 for an illustration and Algorithm 1 for the pseudo code of search, insert, and delete in SoLL.

BALL

Please refer to Figure 3 and Algorithm 2. Each node in a BaLL, except the *middle* node, has two fields: the *data* and the *next* pointer. The middle node has three fields: the *data*, the *upper* pointer, and the *lower* pointer. There is a pointer called *mid* that points to the middle node. Initially, the list is empty and the mid points to null. When BaLL is not empty and has n elements, it consists of three parts: (1) the middle node, (2) *upper sorted linked list* (*upper SoLL*), (3) *lower sorted linked list* (*lower SoLL*). The middle node contains the data that is the median of the n elements. The upper SoLL contains the first $\lfloor n/2 \rfloor$ elements in a sorted sequence from high to low. The lower SoLL contains the remaining data in a sorted sequence from low to high. In addition, sizes of the upper and lower SoLLs are saved into two variables. These two variables will help maintain the size of the two SoLLs almost equal (differ by at most one.)

For searching an element x in BaLL, it is first compared with the middle node. If it is found there, then the search returns mid. Otherwise, if x is smaller than the middle node, then x is searched in the upper SoLL. It starts from the first node of the upper SoLL and keeps going until x is found or the end of the list is reached. During this search the previous (denoted as *prev*) of the current node is recorded. This prev node will be helpful during insertion and deletion. If x is larger than the middle node, then it is searched similarly in the lower SoLL.

For insertion, when the very first element is inserted into the BaLL, the middle node is created with its data as x and the left and right pointers as null. For each subsequent insertion of an element x, it is compared with the middle node. If x is the same as the middle data, then x is inserted as the first node of lower SoLL. If x is smaller than the middle data, then the upper SoLL is searched for an *appropriate position* of x by using a similar method that was used for searching x. By an appropriate position of x, it means that we continue going in the upper SoLL as long as the elements are bigger than x. During this move, the previous node (denoted as *prev*) of the current node is saved. When stopped, x is inserted as a new node after the prev. This insertion can be done by assigning the next pointer of prev to the next pointer of current node and by assigning the next pointer of x to the current. Similarly, if x is bigger than the middle node, then in a similar way it is inserted in an appropriate position in the lower SoLL. After the insertion, the upper and lower SoLLs remain sorted.

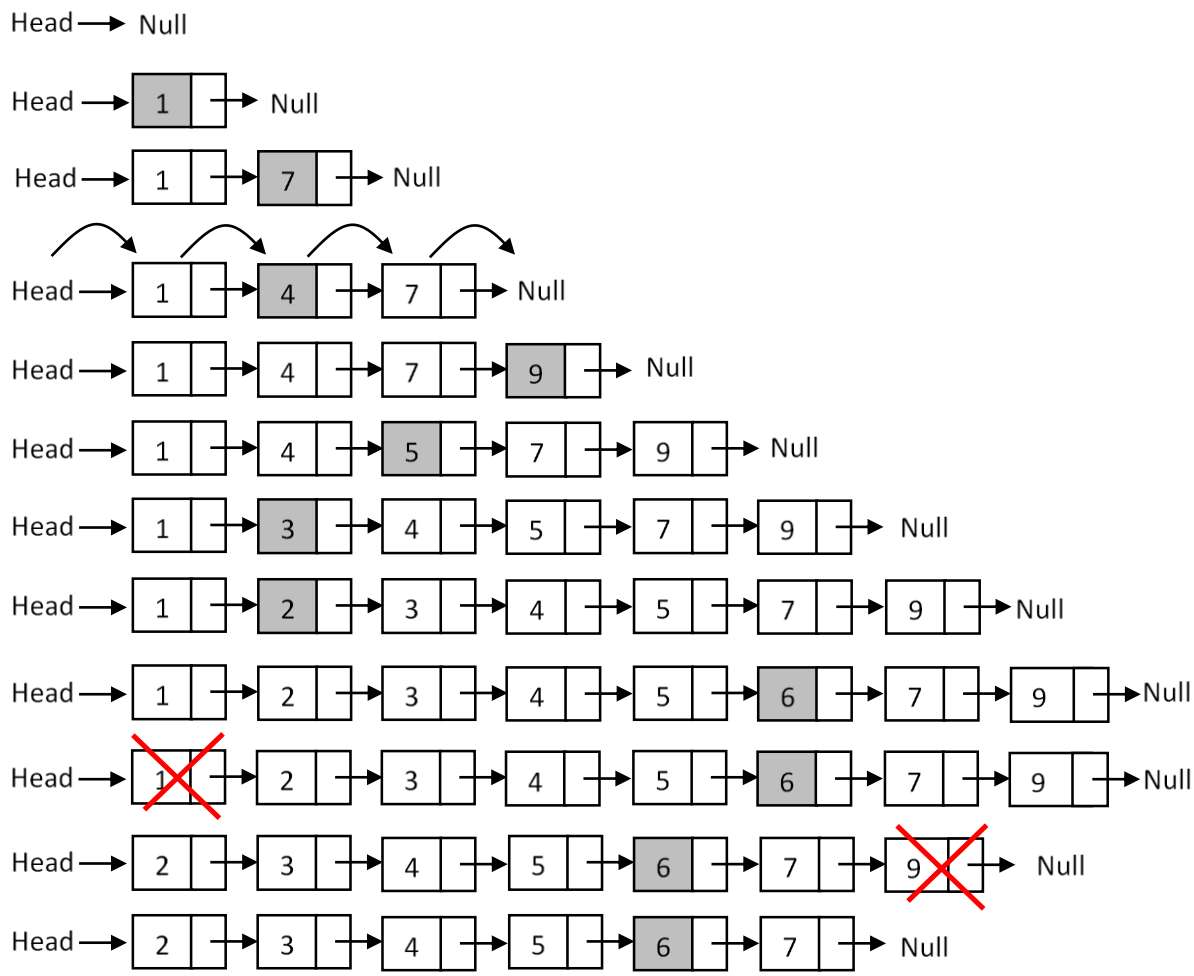

Figure 2. SoLL with search, insert and delete. The inserted elements are (shown as shaded) 1, 7, 4, 9, 5, 3, 2 and 6 in sequence. A searching step before inserting 9 is shown by arrows. The last three lines show the delete of 1 and 9 in sequence.

Search in SoLL (x)

1. Start from head
2. While x (or null) not found
3. keep going by following the next node
4. keep track of the previous (prev) of the current node x
5. Return the node if x found, otherwise return null

Sorted Insert in SoLL (x)

1. Search for an appropriate position of x by **Search in SoLL (x)**
2. Insert x there by manipulating the next pointers of x and prev

Delete from SoLL (x)

1. Search for x by **Search in SoLL (x)**
2. Delete x by manipulating the next pointer of prev

Algorithm 1. Pseudo code for SoLL.

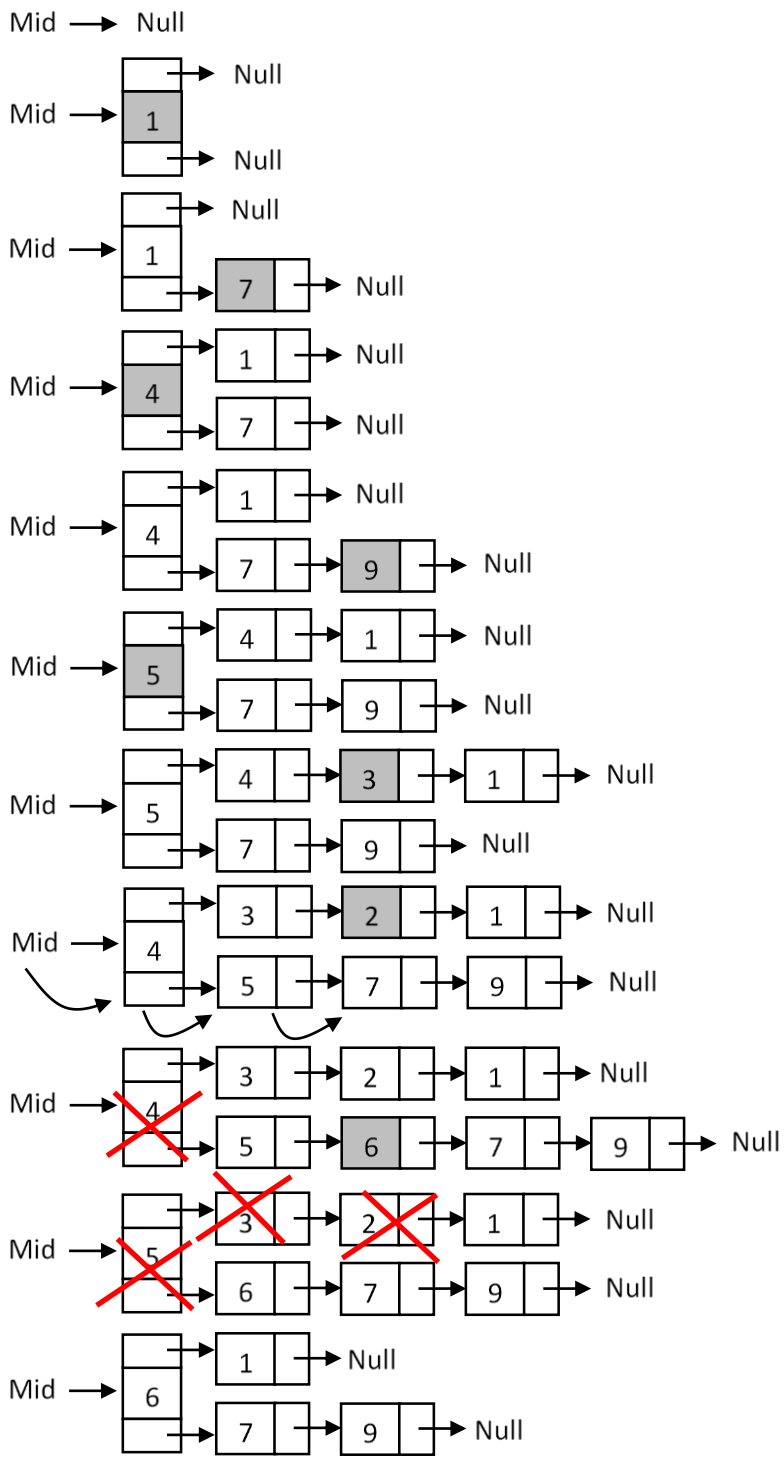

Figure 3. BaLL with search, insert and delete. A sequence of insertion (shown shaded) of 1, 7, 4, 9, 5, 3, 2, and 6. For inserting 5, it is first inserted in the lower SoLL, then it is moved to the middle node. A sequence of delete for 4, 5, 3 and 2 is shown in the last three blocks of the picture. After deleting 4, node 5 was moved to the middle node. After deleting 5, node 6 was moved to the middle node.

For deleting an element x from BaLL, first, x is searched in the BaLL starting from the mid. If x is in the middle node, then it checks the upper and lower SoLLs for which one has a bigger size. The first element from that SoLL is brought at the middle node and that node is deleted. If they have the same size, then the first node from the lower SoLL is brought at the middle node. In the second case, if x is not the same as the middle node, then if it is smaller than the middle node, then x is searched in the upper SoLL. Otherwise, it is searched in the lower SoLL. During this search, the node previous to the current node is preserved. When x is found, the node containing x is deleted by changing the next pointer of the prev node to the next pointer of the current node.

After each insertion or deletion, the size of the upper and lower SoLLs are updated accordingly. Moreover, if these two sizes differ by more than one, then the middle element is shifted to the SoLL that has smaller size and the first element

from the other SoLL is brought into the middle element. These can all be done by manipulating the pointers of the middle node as well as the first nodes of the two SoLLs. This will keep the middle node as the median of the entire BaLL after an insertion or a deletion.

Search in BaLL (x)

1. Compare x with mid.data. If x is same as the mid.data, then return mid
2. If x is smaller than mid. data then
3. follow mid.upper pointer in upper SoLL
4. keep going until x (or null) is found
5. keep track of the previous node (prev) of the current node
6. Else if x is or bigger than mid.data, then
7. follow mid.lower pointer in lower SoLL
8. keep going until x (or null) is found
9. keep track of the previous node (prev) of the current node
10. Return the node if x found, otherwise return null

Sorted Insert in BaLL (x)

1. Search for a node that is bigger or same as x (similar to **Search in BaLL(x)**)
2. Keep track of the previous (prev) of the current node
3. Insert x after prev by manipulating the next pointers of x and prev
4. Adjust the middle node if the size of the upper and lower SoLL differ by more than one

Delete from BaLL (x)

1. Search for an appropriate position of x by **Search in BaLL(x)**
2. If x is in the mid node, then
3. delete mid
4. bring the first node from the upper SoLL or lower SoLL whichever has bigger size
5. If they have same size, then bring from the lower SoLL
6. Else
7. delete x from upper or lower SoLL by manipulating the next pointer of prev
8. Adjust the middle node if the size of the upper and lower SoLL differ by more than one

Algorithm 2. Pseudo code for BaLL.

THEORETICAL PERFORMANCE OF BALL

In this section, it has been shown that theoretically BaLL performs 50% better than SoLL. First, we prove this for search. Then the result will follow for insertion and deletion.

Theorem 1:

In the worst case and average case, searching an element in a BaLL takes 50% less steps than SoLL.

Proof:

For a SoLL, in the worst case, finding x can take n comparisons, as x may be the very last element or x may not be present in the SoLL. On the other hand, for a BaLL, one comparison is needed to check x to be in the middle node. Then the search can continue until the end of either the upper SoLL or the lower SoLL. As the size of these two SoLLs is $\lfloor n/2 \rfloor$, it will take another $\lfloor n/2 \rfloor$ comparisons to find x in one of those two SoLLs. So, the total number of comparisons for searching x is $\lfloor n/2 \rfloor+1$. This gives that the improvement of BaLL over SoLL is

$$\frac{\lfloor n/2 \rfloor + 1}{n} \times 100\%$$

= 50% for large values of n.

For average cases, in the SoLL, x can be searched at any of the n positions starting from head. Assuming that the value x can be completely random, all positions of x, where x can be found, are equally likely. So, the average number of comparisons required to find x is

$$\frac{1}{n}(1+2+3+\ldots+n)$$
$$= \frac{n(n+1)}{2n}$$

$$= \frac{n+1}{2}$$

For a BaLL, after comparing with the middle node, the search can continue until the end of the upper or lower SoLL. On average, x can be found anywhere from the first to the last position of that SoLL. As the size of the upper and lower SoLLs is $\lfloor n/2 \rfloor$ and each position in a SoLL is equally likely for x, the average number of comparisons required is

$$1+\frac{1}{\lfloor n/2 \rfloor}(1+2+3+\ldots+\lfloor n/2 \rfloor)$$
$$= \frac{\lfloor n/2 \rfloor+3}{2}$$

This gives that the improvement of BaLL over SoLL for searching x is

$$\frac{\lfloor n/2 \rfloor + 3}{n+1} \times 100\%$$

= 50%, as n gets bigger.

This ends the proof of the theorem.

Corollary 1:

In the worst case and average case, inserting and deleting an element in a BaLL takes 50% less steps than SoLL.

Proof:

Once an element x is searched, insertion or deletion of x takes in addition some constant number of pointer updates for both SoLL and BaLL (please see the pseudo code in Figure. 3 and 5 respectively). This constant amount of extra work has no effect as n grows higher and higher. Therefore, as a whole, the performance of insertion or deletion of an element x in BaLL remains the same as that for searching, which is 50% better than in SoLL.

This ends the proof of the corollary.

EXPERIMENTAL PERFORMANCE OF BALL

Both SoLL and BaLL have been implemented in C++ and run with the three operations search, insert and delete. Searches are done after the SoLL and BaLL are filled with elements by insert. A variety of input sizes has been taken with different ranges of input values. The following notations are used for explaining the experimental results below.

I: Input size (total number of integers taken randomly one by one for search, insert, and delete)

R: Range of input values (all I inputs have values within this range)

T_S: Time in ms taken by SoLL for doing an operation (search, insert or delete) for all I inputs

T_B: Time in ms taken by BaLL for doing an operation (search, insert or delete) for all I inputs

M: Improvement of BaLL over SoLL, which is computed as:

$$M = \frac{T_S - T_B}{T_S} \times 100\%.$$

Four different values of R have been considered, which are 0 to 10, 0 to 100, 0 to 1000, and the entire range of integers (that means an open range). For each range R, inputs are taken for thirteen different sets with size I as 2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}, 2^{18} and 2^{19}. In each set, all the I number of inputs are taken randomly from R. For example, for $I = 2^7$ and R = 0 to 10, we have taken 2^7 random integers whose values are within 0 to 10. In this way, for each of the operations search, insert and delete, there are fifty-two input settings. See Table 1.

In most settings, especially for the larger I, BaLL substantially outperforms SoLL for all three operations. For each input size, the corresponding table entry shows the total time taken by SoLL and BaLL in ms to perform that many operations in the list with input values taken randomly from the range. See Table 1.

In few cases, mostly for lower values of I, BaLL does not perform better than SoLL. In those cases, the improvement M is negative. We do not show, and skip, those negative values in the table.

Figure 4 shows graphically the performance of BaLL over SoLL. In each of the graphs in this figure, X-axis represents the higher values of input size, namely from 2^{16} to 2^{19}. Y-axis represents the time taken by the operations in ms. From the graphs in this figure it is evident that as the input size I gets bigger, the performance of BaLL is increased for all value ranges R of the inputs. Moreover, for a higher range of R the performance of BaLL is better. This indicates that as the input gets more random the BaLL performs better.

As a whole, for higher input size and large range of input values, BaLL performs better than SoLL. This indicates that in real life input, where the data can be completely random and can have any values, BaLL can be a suitable data structure for maintaining a sorted sequence of data.

Table 1. Experimental results of search, insert and delete in SoLL and BaLL.

I	R	Search			Insert			Delete		
		T_S	T_B	M	T_S	T_B	M	T_S	T_B	M
2^7		0.001	0	100%	0.004	0.001	75%	0	0.001	-
2^8		0	0.001	-	0.006	0.002	67%	0	0.001	-
2^9		0.002	0.002	0%	0.01	0.003	70%	0.002	0.001	50%
2^{10}		0.011	0.005	55%	0.015	0.03	-	0.006	0.004	33%
2^{11}		0.047	0.03	36%	0.044	0.035	20%	0.027	0.013	52%
2^{12}		0.223	0.102	54%	0.185	0.073	61%	0.117	0.05	57%
2^{13}	**0-10**	0.783	0.42	46%	0.523	0.282	46%	0.446	0.182	59%
2^{14}		2.343	1.478	37%	2.654	1.2	55%	1.285	0.678	47%
2^{15}		6.032	3.182	47%	8.267	4.941	40%	6.501	2.297	65%
2^{16}		59.06	25.653	57%	26.387	15.768	40%	35.134	8.154	77%
2^{17}		456.259	200.427	56%	211.705	128.755	39%	184.006	70.315	62%
2^{18}		3301.93	1592.32	52%	1272.31	817.431	36%	1100.45	673.198	39%
2^{19}		14486.6	6343.68	56%	5363.31	3625.19	32%	5075.28	3209.79	37%
2^7		0	0	-	0.002	0.009	-	0	0.001	-
2^8		0	0	-	0.01	0.011	-	0	0	-
2^9		0.001	0	100%	0.01	0.012	-	0.001	0	100%
2^{10}		0.003	0.002	33%	0.018	0.012	33%	0.005	0.002	60%
2^{11}		0.014	0.008	43%	0.04	0.019	53%	0.018	0.005	72%
2^{12}		0.052	0.033	37%	0.138	0.064	54%	0.07	0.017	76%
2^{13}	**$0\text{-}10^2$**	0.337	0.202	40%	0.515	0.356	31%	0.357	0.08	78%
2^{14}		1.679	0.876	48%	2.177	1.549	29%	1.455	0.377	74%
2^{15}		10.462	4.417	58%	7.774	4.536	42%	6.552	1.867	72%
2^{16}		85.622	38.233	55%	28.113	24.2	14%	23.994	12.729	47%
2^{17}		598.799	298.47	50%	224.464	139.593	38%	283.718	129.33	54%
2^{18}		4016.57	2246.04	44%	1307.42	767.282	41%	1418.58	963.62	32%
2^{19}		17324.4	8919.71	49%	5702.76	3433.34	40%	6440.65	4351.47	32%
2^7		0	0	-	0.005	0.005	0%	0	0	-
2^8		0	0.001	-	0.009	0.011	-	0	0	-
2^9		0.001	0.001	0%	0.007	0.009	-	0.001	0.001	0%
2^{10}		0.004	0.004	0%	0.009	0.008	11%	0.004	0.002	50%
2^{11}		0.014	0.014	0%	0.013	0.012	8%	0.013	0.007	46%
2^{12}		0.214	0.134	37%	0.036	0.029	19%	0.043	0.023	47%
2^{13}	**$0\text{-}10^3$**	1.001	0.749	25%	0.159	0.3	-	0.193	0.393	-
2^{14}		2.63	2.472	6%	0.826	1.325	-	0.91	1.084	-
2^{15}		7.848	7.592	3%	3.997	4.254	-	4.008	2.385	40%
2^{16}		82.647	49.903	40%	25.79	16.276	37%	24.992	18.054	28%
2^{17}		642.513	323.513	50%	235.616	132.649	44%	290.901	138.527	52%
2^{18}		4411.88	2379.55	46%	1314.32	746.572	43%	1532.23	1109.35	28%
2^{19}		19027.1	8936.06	53%	5929.24	3603.88	39%	6821.12	4791.97	30%
2^7		0	0	-	0.001	0.002	-	0	0	-
2^8		0.001	0.001	0%	0.001	0.002	-	0.001	0	100%
2^9		0.003	0.003	0%	0.002	0.004	-	0.005	0.003	40%
2^{10}		0.013	0.01	23%	0.004	0.007	-	0.028	0.016	43%
2^{11}		0.058	0.056	3%	0.025	0.017	32%	0.117	0.005	96%
2^{12}		0.265	0.215	19%	0.123	0.155	-	0.416	0.32	23%
2^{13}	-	1.138	0.824	28%	0.355	0.231	35%	1.728	1.006	42%
2^{14}		7.731	2.556	67%	1.908	0.848	56%	4.367	2.672	39%
2^{15}		12.864	7.006	46%	3.961	2.783	30%	15.353	5.267	66%
2^{16}		139.32	50.753	64%	34.478	13.205	62%	64.993	28.515	56%
2^{17}		694.367	357.735	48%	266.506	114.277	57%	556.135	180.845	67%
2^{18}		3022.93	1533.62	49%	1349.03	663.232	51%	2146.6	1174.2	45%
2^{19}		15253.2	8156.11	47%	6207.32	3124.55	50%	8465.57	4995.52	41%

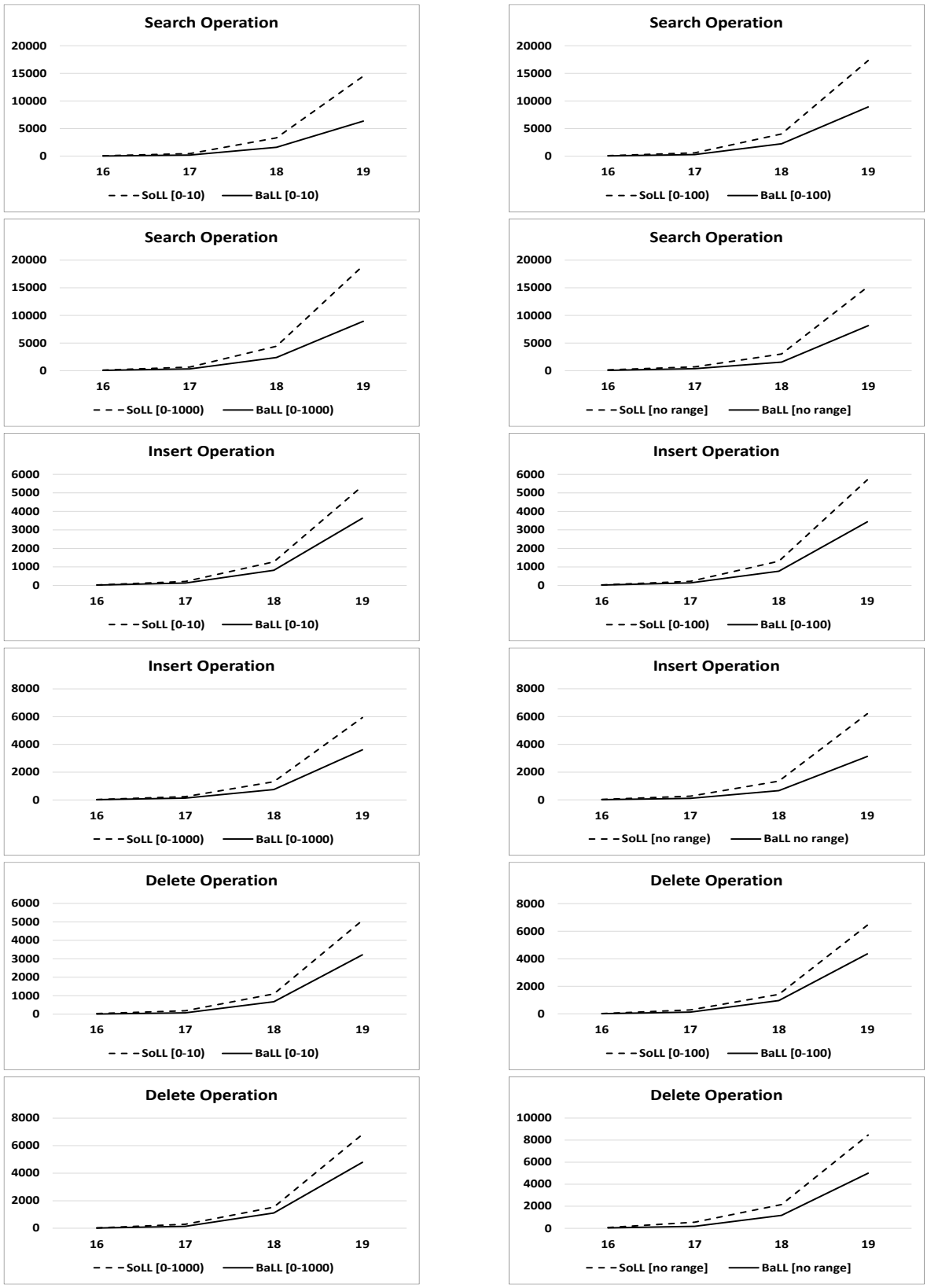

Figure 4. Comparison of SoLL and BaLL for higher input size.

CONCLUSION

In this paper, a modified version of linked list, called BaLL, has been presented for maintaining a sorted sequence of data. The modification is to maintain a mid-pointer instead of a head pointer of a single linked list. The two sides of the mid pointer are sorted from low to high and from high to low respectively. Three standard operations, search, insert and delete, have been considered in BaLL. Algorithms for these operations have been presented. The performance of BaLL has been compared with standard sorted linked list (SoLL) both theoretically and experimentally. Search, insert and delete operations in BaLL have been compared with similar operations in SoLL. It has been shown that theoretically BaLL performs 50% better than SoLL. Experimentally the performance of BaLL is better than SoLL by around 50%.

There can be several future works possibly related to BaLL. In this paper, BaLL has been considered only for a single linked list. It would be interesting to see how this concept can be implemented with a doubly linked list. The experimental results compare BaLL with SoLL. It would be interesting to see if BaLL can be compared to other ADT that maintain sorted data, such as B-tree, heap, and binary search trees. The experiment in this paper has been done with computer generated random values. It would also be interesting to do the experiment with some real-life input data.

ACKNOWLEDGEMENT

The author would like to thank Mohammad Klaib for helping in experiments and Masud Hasan for helping in writing the article.

REFERENCES

Carraway, J. (1996). Doubly-Linked Opportunities. *ACM SIG3C 3C ONLINE*, Vol. 3, No. 1, pp.9-12.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein C. (2009) Introduction to Algorithms, 3rd ed. The MIT Press, Cambridge.

Eberl, M., Haslbeck, M. W., Nipkow, T. (2020). Verified analysis of random binary tree structures. *International Journal of Automated Reasoning*, Vol. 64, pp.879–910.

Gautam, S. (2020). SSD Based Indexing and Tree Data Structures. Manuscript submitted to an *IEEE Journal*.

Goodrich, M. T., Tamassia, R. (2014). Algorithm Design and Applications. Wiley.

Haiming, L., Ping, C., Yong, W. (2017). Heap Sorting Based on Array Sorting. *Journal of Computer and Communications*, Vol. 05, No. 12, pp.57-62.

Hasan, M., Lopez-Ortiz, A., Sedgewick, R. (2020). Extended Order Notation. Manuscript Available at: *https://www.researchgate.net/publication/348835358_Extended_Order_Notation.*

Hijazi, S., Qatawneh, M. (2017). Study of Performance Evaluation of Binary Search on Merge Sorted Array Using Different Strategies. *International Journal of Modern Education and Computer Science*, Vol. 12, pp.1-8.

Imran, A. (2020). 40 Algorithms Every Programmer Should Know: Hone your problem-solving skills by learning different algorithms and their implementation in Python Paperback, Illustrated ed., Packt Publishing.

Jie, L., Pengfei, C. (2017). Improvement of B-Trees. *7th International Conference on Advanced Design and Manufacturing Engineering* (ICADME 2017), Atlantis Press. DOI: https://doi.org/10.2991/icadme-17.2017.53

Kleinberg, J., Tardos, E. (2005). Algorithm Design. 1st ed. Pearson.

Knuth, D. E. (1971). Optimum binary search trees. *ACTA Inform.*, Vol. 1, pp.14-25.

Koganti, H. (2019). Searching in A Sorted Linked List and Sort Integers into a Linked List, A Thesis in Computer Science Presented to the Faculty of The University of Missouri-Kansas City in Partial Fulfilment of The Requirements for The Degree Master of Science.

Koganti, H., Yijie, H. (2018). Searching in a Sorted Linked List. *International Conference on Information Technology* (ICIT2018), Bhubaneswar, India, pp.120-125.

Kowalski, T., Kounelis, F., Pirk, H. (2020). High-Performance Tree Indices: Locality matters more than one would think. *11th International Workshop on Accelerating Analytics and Data Management Systems* (ADMS'20), Tokyo, Japan.

Malik, D. S. (2017). C++ Programming: Program Design Including Data Structures. 8th ed, Cengage Learning.

Nipkow, T., Eberl, M., Haslbeck, M.P.L. (2020). Verified Textbook Algorithms. In: Hung D.V., Sokolsky O. (eds) Automated Technology for Verification and Analysis. ATVA 2020. *Lecture Notes in Computer Science*, vol 12302. Springer.

Sanders, P., Mehlhorn, K., Dietzfelbinger, M., Dementiev, R. (2019). Sorted Sequences. In: *Sequential and Parallel Algorithms and Data Structures*. Springer, 2019.

Sanu, K. (2011). Binary Search in Linked List. *International Journal of Engineering and Advanced Technology* (IJEAT), Vol. 9, No. 1, pp. 2684-2686.

Sedgewick, R., Wayne, K. (2011). Algorithms, 4th ed. Addison-Wesley Professional.

Shene, C. K. (1996). A comparative study of linked list sorting algorithms. *3C ON-LINE* Vol. 3, No. 2, pp.4–9.

Skiena, Steven S. (2020). The Algorithm Design Manual (Texts in Computer Science), 3rd ed. Springer.

Snyder, K. (2006). System and method for implementing dynamic set operations on data stored in a sorted array. US Patent 7,069,272.

Srinivasan, K., Kumar, R., Singla, S. (2017). Robust and efficient algorithms for storage and retrieval of disk-based data structures. *International*

Conference on Applied System Innovation (ICASI), Sapporo, pp.934-937.

Verma, A. K., Kumar, P. (2013). List Sort: A New Approach for Sorting List to Reduce Execution Time. *DBLP computer science bibliography*. Avaliable at: https://arxiv.org/abs/1310.7890.

Weiss, M. A. (2011). Data Structures and Algorithm Analysis in Java, 3rd ed. Pearson.

Wengrow, J. (2020). A Common-Sense Guide to Data Structures and Algorithms, Level Up Your Core Programming Skills, 2nd ed, Pragmatic Bookshelf.

Zhi-Gang, Z. (2020). Analysis and Research of Sorting Algorithm in Data Structure Based on C Language. 5th International Conference on Intelligent Computing and Signal Processing (ICSP) 2020 pp.20-22, Suzhou, China.