STM JOURNALS

International Journal of
Data Structure Studies

ISSN: 3049-2440
Volume 4, Issue 1, 2026
January–June
DOI (Journal): 10.37591/IJDSS

https://journals.stmjournals.com/ijdss

Review IJDSS

An Auxiliary Array Indexing Approach for Efficient Binary Search in Linked Lists

Rejay Pavunkumar[1,*], Royal Rodrigues[2], Rohini Bagul[3], Vaishnavi Biradar[4]

Abstract

The paper covers an algorithm for searching a linked list structure using binary search. Binary search is a classic example of an algorithm that follows the divide-and-conquer approach. Binary search may be used to find elements in an array. Trying to apply the conventional binary search to a linked list simply does not work out very well; it still has an O(n) time complexity, the same as linear search. This is because in linked lists, you cannot immediately go to a particular element at once; you must go through the entire list, which requires O(n) time to look up. To solve this problem, this paper is going to suggest a method that will be able to carry out binary search in a linked list with a lower time complexity of O(log2n). The approach that has been suggested makes use of an auxiliary array that helps in indexing in the linked list, enabling the access of nodes in constant time (O(1)). As a result, the binary search operation will be much better, and its performance in the linked list implementation will be better.

Keywords: Binary search, linked list, auxiliary array, indexing, array of pointers

INTRODUCTION

Efficient data retrieval is one of the most important dimensions of algorithm design and analysis in computer science. Algorithms capable of finding information quickly and accurately have become a necessity with the ever-increasing amount of data in contemporary computing systems. Of these, one can say the binary search algorithm is one of the most efficient methods of searching [1, 2]. Binary search is based on the divide-and-conquer paradigm, which involves recursively dividing a problem into smaller subproblems recursively until the required solution is achieved. This can be repeated to reduce the search space by a factor of two in the case of searching. The algorithm initially compares the target value and the middle of the list. If the target exceeds the mid-element, the search is repeated in the right half of the list. If the target is smaller, the left half is searched. This is repeated until the target is located or the search interval is vacant.

***Author for Correspondence**
Rejay Pavunkumar
E-mail: rejoypkumar@gmail.com

[1-3]Student, Master of Computer Application, Thakur Institute of Management Studies & Career Development & Research, Kandivali East, Mumbai, Maharashtra, India
[4]Assistant Professor, Master of Computer Application, Thakur Institute of Management Studies & Career Development & Research, Kandivali East, Mumbai, Maharashtra, India

Received Date: March 10, 2026
Accepted Date: March 31, 2026
Published Date: May 07, 2026

Citation: Rejay Pavunkumar, Royal Rodrigues, Rohini Bagul, Vaishnavi Biradar. An Auxiliary Array Indexing Approach for Efficient Binary Search in Linked Lists. 2026; 4(1): 21–28p.

The main advantage of the binary search is that its time complexity is a logarithmic number O(log2n); therefore, the algorithm can effectively work with large datasets with minimum comparisons [1, 3]. For example, with one million elements in a list, fewer than 20 comparisons are required to search the list. Such a benefit in performance is because binary search is based on random access capability, which allows it to access any data item directly in terms of its index [2]. Nevertheless, this is a huge and challenging requirement for some data structures, especially linked lists. A linked list is a dynamic data structure, which is a pointer-based data structure, consisting of nodes, each node in a sequence

containing data and a pointer to the next node in the sequence. The final node usually refers to NULL [4]. Compared to arrays, the elements of linked lists are not stored in adjacent memory locations; hence, they do not have random access. The element that provides an average time complexity of O(n) to find the midpoint element [5, 6].

Despite the above-mentioned drawbacks, which exist in the linked list data structure, the most desired argument to utilize the linked list data structure, among the rest, depending on the need and depending on the most preferable structure, the list may grow or decrease in size, depending on the frequency with which the deletions and insertions are made [5, 7].

The operation of insertion or deletion in a linked list takes O(1) time, that is, it only takes the modification of the reference pointers and does not need the complete array to be shifted as in the case of the array structure, Nonetheless, this flexibility introduces a bottleneck in the development of search algorithms that require random access, such as binary search [4, 1]. In an attempt to correct the performance gap between linked lists and other data structures capable of random access, other authors have proposed techniques that entail partial indexing or other data structures that would make linked lists accessible at random [6–8]. Such efforts are meant to deal with the dynamism of linked lists and make these data structures less time-consuming to traverse. Based on this, it has been proposed to use hybrid data structures as well as intermediate pointer arrays as methods that would help the nodes and, at the same time, preserve the integrity of the list access [5, 9].

Modern methods that would help enhance the access of nodes within linked lists are the use of auxiliary index tables, which hold addresses or pointers of nodes to facilitate direct access of nodes with shorter search time to logarithmic complexity [8, 10]. Based on these concepts, this study presents an Auxiliary Array Indexing Approach that accelerates a binary search on a linked list. The way it works is as follows: you maintain a separate array with your linked list. This array contains the address of the memory of each node on the linked list. Under this arrangement, you can traverse directly to any node in constant time; you no longer need to traverse all the nodes before you come across the one you are interested in. At this point, you have a fast O(log2n) Binary Search that you would want, and you retain all the flexibility that a linked list provides. Certainly, you are imposing the O(n) space overhead of that auxiliary array. However, this is a fair price to pay. The result is a solution that is not only faster but also capable of dealing with scenarios where you are required to resize the list on the fly and still search quickly.

The remainder of this paper is organized as follows: Section II reviews the available literature on binary search in linked and hybrid data structures. The literature review describes the intended methodology and implementation. In the later Section, we discuss the results and analyze the complexity of the experiment. Finally, the last Section presents the conclusions and future research directions.

LITERATURE REVIEW

I also researched the topic of binary search to understand its performance in various data structures, particularly linked lists.

Chakraborty and Sadakane produced an early paper that is especially prominent. He proposed additional index tables, also known as an additional array, which store the address of nodes. Using this configuration, it is much faster to jump to nodes, and there is not as much fuss in the flexible character of the linked list. This concept is essentially preconditioned with the idea of auxiliary- array indexing, which was afterwards adopted by the researchers.

Datta and Bhattacharjee invented the dual-pointer technique for locating the middle of a singly linked list. This method uses two pointers operating at different speeds, commonly referred to as fast and slow

pointers. It is an effective technique that is faster than traversing the list node by node. However, despite this strategy, as long as the list is traversed sequentially, the time complexity remains O(n). Thus, although the technique improves efficiency, it does not overcome the inherent limitations of sequential traversal.

Parmar and Kumbharana [1] compared Linear and binary search algorithms, where they were tested on linked lists, dynamic arrays, and static arrays. They discovered that binary search is only maintained at its O(log2n) rate when one has access to some random access structures. Therefore, when linked lists are used, some sort of artificial indexing is required to achieve the same level of performance.

Oommen and Pal [2] delved into the theory of binary search. They demonstrated that the algorithm is based on the possibility of seizing any element in real time. This observation led subsequent researchers to attempt to add indexing or additional pointers to non-built-in structures.

Yadav and Gupta [5] took a step further. They invented a hybrid design that combines the flexibility of linked lists and the speed of access of arrays. They employed a partial indexing technique that increased the search performance and reduced the difference between sequential and random access methods.

This idea was further explored by Krishan et al. [9]. They generated pointer arrays that could be placed between dynamic data elements. Their scheme of adaptive indexing has hit the right balance—it made the look-up fast without consuming too much space, which indeed matters in systems where data changes a lot.

Zhao [7] examined how auxiliary indexing balances space and time. They discovered that these structures consume O(n) space, but the price is compensated by the speed gain and logarithmic search time, which is worth the price when there are tons of data. In the dynamic data setup.

He and Watson [6] conducted performance tests, that is, pitting the optimized search techniques such as interpolation and jump search. It turned out that auxiliary indexing was more easily scaled, and they were able to keep things running smoothly regardless of the data size.

Hong [10] conducted large-scale experiments related to auxiliary array indexing in large, linked datasets. They enhanced cache locality and reduced access latency. Therefore, this approach is effective in practical workloads. delivers.

METHODOLOGY

The greatest difficulty with the successful implementation of a good binary search algorithm on a traditional linked list lies in the absence of random access features. Unlike arrays, where every object is referenced at a constant time of its index, a linked list requires a linear search to reach a particular node. The middle element, an important step in the binary search algorithm, involves searching the list, which is printed out up to the location where the head node is found, and then to the point where it crosses the midpoint of the list. This traversal is complex by the time of access per middle element, and the analysis of a binary search is no more efficient than that of a linear search.

To overcome this inherent drawback, the suggested methodology suggests an auxiliary data structure, which is an array of pointers that are added to the linked list. This list provides direct access to the nodes, thereby restoring the logarithmic performance of the binary search algorithm. The hybrid method preserves the Dynamic Properties of a linked list, while this entails the integration of the effective search SQL of arrays.

Auxiliary Pointer Array

Here, an additional array, arr[n], is created. This array contains as many nodes as there are in the linked list, and the memory address of the respective node is stored in the slot. This means that you can

directly jump to any node by using its index (there is no need to browse the list step at a time). Under such an arrangement, your linked list will literally become a hybrid: you get the standard sequencing of a linked list and the fast, arbitrary access that you would hope to be offered by an array. This is what is required to perform a binary search.

To fill this array, all you have to do is visit each list element from the head to the tail. You write the memory address of each node at the next position in arr. Subsequently, any node can be accessed in the array at any time with its index. When one links the original list, there are no modifications made to the original list, but only accelerated access.

Middle Node Determination

The identification of the middle node plays a major role in the binary search. When you have constructed an auxiliary array, you need only apply the standard formula to compute the middle index-begin and end are the initial and final indices. was computed using the following standard formula:

$$\text{mid} = \frac{\text{big+end}}{2}$$

Under this arrangement, one can access the middle node by making a direct call to arr[mid]. One does not have to go through the list repeatedly to find the middle element. Consequently, the middle node can be acquired in O(1) time, which is better than the previous method.

Binary Search Algorithm

Once the auxiliary pointer array has been established, it is time to perform a binary search. Here is how it goes:

1. Compare the key and the data of the middle node

If head->data is the key you are looking for, you are done. You found it.

2. In case the key is larger than the data of the middle node

Change your attention to the right-hand side of the array. The search set beg to mid + 1 and continues searching.

3. In case the key is less than the data of the middle node

Move to the left part instead. Update end to mid – 1.

Repeat these steps until you locate the key or all elements that you have attempted. The entire search is still fast, of that traditional logarithmic time complexity, because you are directly going to the middle every time with the aid of that auxiliary array.

Dry Run Example

Let us demonstrate how this works. For example, if our list is already ordered as such: {1, 2, 3, 4, 5} and we wish to locate the value of '2', we must first create an auxiliary (support) array where we store pointers to each linked list node in the order they were linked. At this point in time, we have set beg = 0 and end = 4. We can determine the middle index by calculating mid. Check to see what value is located at arr[2], which is the node pointer for value 3. Value 3 is less than value 2, and we will thus set end = 1. We will use the following calculation to determine mid:

$$\text{mid} = \frac{0 + 4}{2} = 2$$

Next round: mid is (0 + 1)/2, so mid = 0. arr[0] holds 1, which is less than 2. Updates beg to 1. Now, arr[1] has the value 2— bingo, that is the key. So, the method works. The search quickly finds the element, just as you would hope with a binary search.

Complexity Analysis

The time complexity is $O(\log_2 n)$ because each step cuts the search interval in half. However, the space complexity is O(n) because room is required for the extra array. This approach is much more efficient than the old-school linked list binary search, which drags along with O(n) time.

ALGORITHM TO FIND MIDDLE IN LINKED LIST

If you need to locate the middle or any location in a linked list very fast, you can use the following trick: construct an auxiliary pointer array (also known as a helper array). Make it the same size as your linked list. Sequentially work down the list and at each node, place the memory address of this node in the corresponding position of this new array. When that is in place, all that needs to be done is go straight to any node by simply searching its address in the array. Access to any node is now O(1) – light speed each time, regardless of the length of the list (Figures 1 and 2).

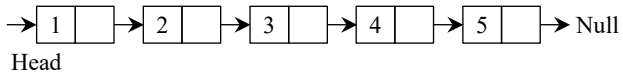

Figure 1. Linked list.

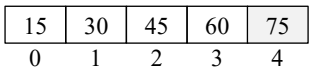

Figure 2. An array of pointers.

Each element of the array stores the memory address of the corresponding node in a linked list.

SOURCE CODE TO ASSIGN NODE ADDRESSES TO ARRAY

The following function demonstrates how the addresses of all linked list nodes are assigned to an auxiliary pointer array.

```
def create_pointer_array(head_node): aux_array = []
current_node = head_node

while current_node is not None:
aux_array.append(current_node) current_node
= current_node.next
return aux_array
```

FINDING THE MIDDLE OF A LINKED LIST

The middle of the linked list can be obtained easily using an array of pointers. We only had to find the middle of the array and assign the middle value of the array to a node-type pointer.

```
//pseudo code to find the middle of the linked list
def binary_search_linked_list(aux_array, target): low = 0
high = len(aux_array) - 1
while low <= high:
# FINDING MIDDLE OF LINKED LIST (via
array index)
mid = (low + high) // 2
# O(1) Access to the middle node
mid_node = aux_array[mid]
mid_value = mid_node.data
if mid_value == target:
return mid_node
elif target > mid_value: low = mid + 1
 else:
 high = mid - 1 return
 None
```


BINARY SEARCH AFTER FINDING MIDDLE ELEMENT

```
def binary_search_linked_list(aux_array, target):
low = 0
high = len(aux_array) - 1

while low <= high:
mid = (low + high) // 2

mid_node = aux_array[mid]
mid_value = mid_node.data

if mid_value == target:
return mid_node
elif target > mid_value:
low = mid + 1
else:
high = mid - 1
return None
```

DRY RUN OF CODE

Let the initial linked list be as shown in Figure 3. The data in the linked list is in sorted order. Hence, we can perform a binary search in this linked list. In Figure 4, an array of pointers(arr) is shown, in which each entry is the address of the corresponding node. Suppose that the key element is 2.

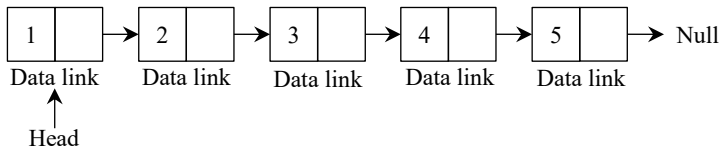

Figure 3. Illustration of a linked list containing five nodes.

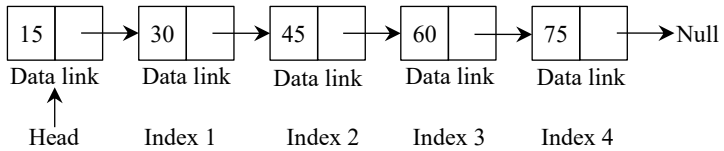

Figure 4. Auxiliary array pointers.

At the start, beg is 0 and end is 4.
So, mid works out to (0+4)/2, which gives 2.
That means head points to a[mid], or 45.
If you look at head->data, you will see it is 45.
You can see how this looks in Figure 5.

Now, since head->data(3) is greater than the key value(2). Hence, there will be a decrement in the end index of the array(arr).
Hence, now beg=0, end=3 Thus mid=(0+3)/2=1
Thus, head=a[mid] head=30 head->data=2

This is shown in Figures 6 and 7.

Now, head->data=2, which is equal to the key value, which is 2. Hence, the key value is found in the linked list.

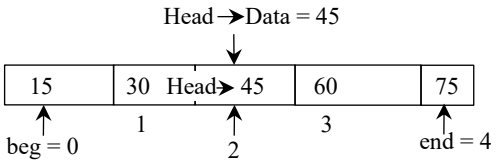

Figure 5. The head pointer currently designates the third node of the list.

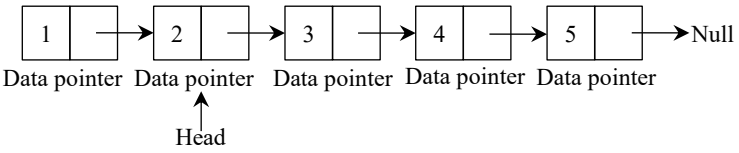

Figure 6. The head pointer's current assignment is the list's second node.

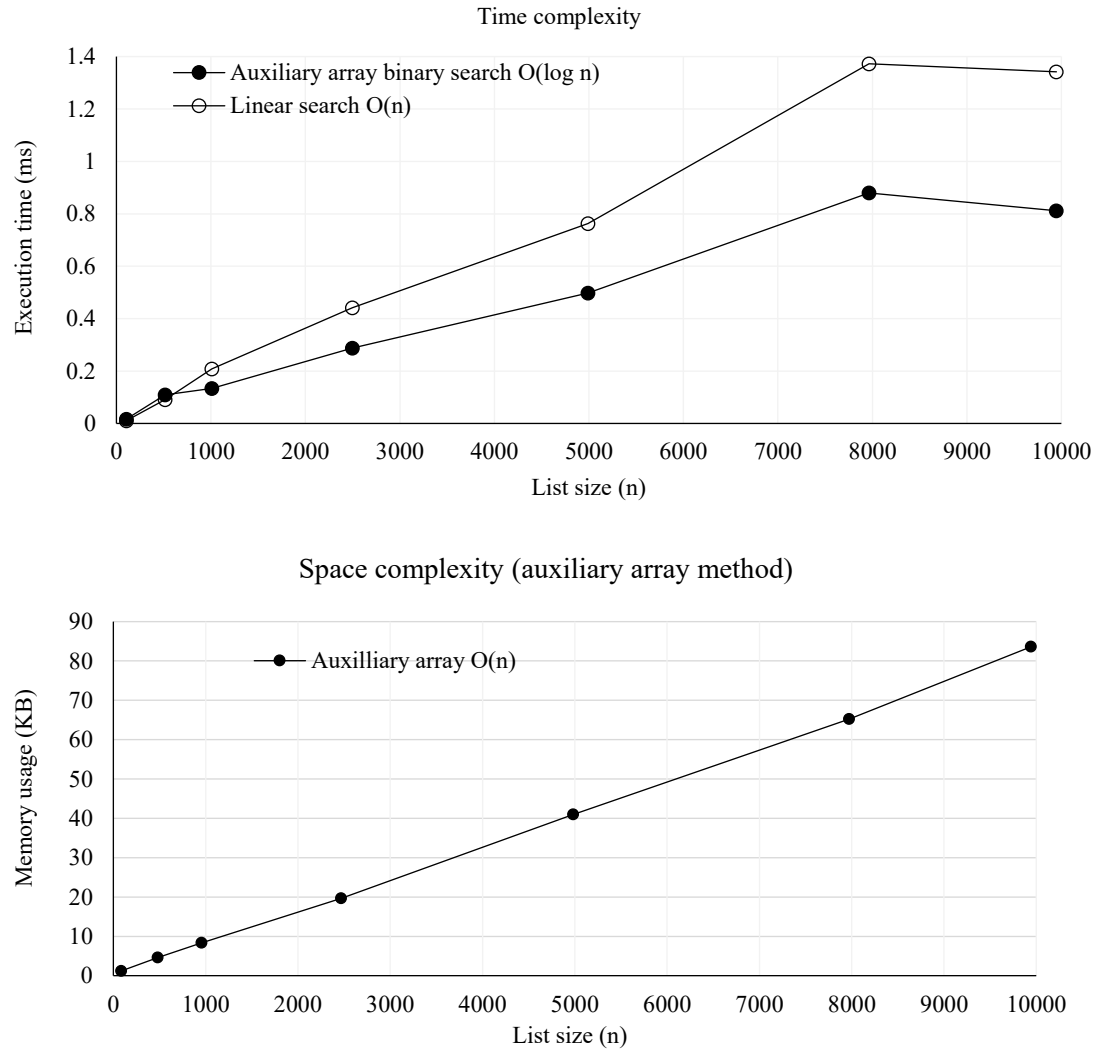

Figure 7. Performance.

CONCLUSION

The new approach significantly speeds up search efficiency, as can be seen in the time complexity graph. With a regular linear search on a linked list (O(n)), the execution time continues to increase as

the list becomes larger. However, when binary search is used with an auxiliary array ($O(\log_2 n)$), the process is much faster, especially when passing 4,000 elements. This is where the difference really jumps out. The binary search keeps the execution times much lower, even with huge datasets. For example, with 10,000 items, a linear search takes approximately 1.4 ms. It finished at approximately half that time.

This boost is due to the change in access patterns of the auxiliary array. It maintains a separate index that points directly to the right-linked list nodes; therefore, there is no need to trudge through the list to find the middle. The algorithm works like a proper divide-and-conquer method: it keeps cutting the search space in half until it finds what you are looking for, or until there is nowhere left to look. It gets you faster, but you consume more memory. Now you can see the space complexity graph. The auxiliary array aggravates the memory load by an additional O(n), which, in other words, just increases in tandem with your list. If your list increases to 2,000 and then to 10,000 items, the memory consumption increases to approximately 80 kb. Nevertheless, it is a fair price considering that operations go much faster, and it comes in handy when one must process large data volumes, and speed can be crucial.

Eventually, there is a happy medium in the use of auxiliary arrays. This makes the slowness of linked list access more or less similar to random access. Hence, this entire solution, binary search on a linked list with an auxiliary array, allows searching to be much more scalable and practical for applications where performance matters.

REFERENCES

1. Parmar VP, Kumbharana CK. Comparing linear search and binary search algorithms to search an element from a linear list implemented through static array, dynamic array and linked list. Int J Comput Appl. 2015;121(3):13–17. doi:10.5120/21519-4495.
2. Sanu K. Binary search in linked list. Int J Recent Technol Eng. 2019;8(4):2684-6. doi:10.35940/ijrte.D7296.118419.
3. Joseph M, Keshwani P. Comparison between linear search and binary search algorithms. National Conference on Innovative Solutions for Rural Development of Chhattisgarh (NCISRDC), Chhattisgarh, India. 2018 Feb 23–24. p. 63–66. Available from: https://www.ijariit.com/conference-proceedings/15%20CSE%20107.pdf
4. Datta S, Bhattacharjee P. Implementation of binary search on a singly linked list using dual pointers. Int J Comput Sci Inf Technol. 2014;5(2):1908–1910.
5. Yadav R, Sreedevi I, Gupta D. Bio-inspired hybrid optimization algorithms for energy efficient wireless sensor networks: a comprehensive review. Electronics. 2022;11(10):1545. doi:10.3390/electronics11101545.
6. He J, Watson LT, Ramakrishnan N, Shaffer CA, Verstak A, Jiang J, et al. Dynamic data structures for a direct search algorithm. Comput Optim Appl. 2002;23(1):5–25. doi:10.1023/A:1019992822938.
7. Zhao H, Deep S, Koutris P. Space-time tradeoffs for conjunctive queries with access patterns. 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS), Seattle, Washington, USA. 2023. p. 59–68. doi:10.1145/3584372.3588675.
8. Chakraborty S, Sadakane K. Indexing graph search trees and applications [Preprint]. 2019. arXiv:1906.07871. doi:10.48550/arXiv.1906.07871.
9. Krishan K, Gupta G, Bhathal GS. Query load management: an approach for optimizing database performance. OPSEARCH. 2025:1–8. doi:10.1007/s12597-025-01012-x.
10. Hong B, Kim G, Ahn JH, Kwon Y, Kim H, Kim J. Accelerating linked-list traversal through near-data processing. 2016 International Conference on Parallel Architectures and Compilation Techniques (PACT), Haifa, Israel. 2016. p. 113–124. doi:10.1145/2967938.2967958.

