An Efficient Scheme for Real-time Information Storage and Retrieval Systems: A Hybrid Approach

Syed Ali Hassan*, Imran Ul Haq*, Muhammad Asif*, Maaz Bin Ahmad† and Moeen Tayyab‡
*Department of Computer Science and Information Technology
Lahore Leads University, Lahore, Pakistan
† COCIS, PAF Karachi Institute of Economics and Technology, Karachi, Pakistan
‡ International Islamic University, Islamabad, Pakistan

Abstract—**Information storage and retrieval is the fundamental requirement for many real-time applications. These systems demand that data should be sorted all the time, real-time insertion, deletion and searching should be supported and system must support dynamic entries. These systems require search operations to be performed from massive databases implemented by various data structures. The common data structures used by these systems are stack, queue or linked list all having their own limitations. The biggest advantage of using stack is that binary search can be performed on it easily while on the other hand insertion and deletion of nodes involves more processing overhead. In linked list, insertion and deletion of nodes is easier but searching operation involves more processing overhead as binary search cannot be performed efficiently on it. In this paper, a hybrid solution is presented for such systems, which provides efficient insertion, deletion and searching operations. Results show the effectiveness of the proposed approach as it outperforms the existing techniques used by these systems.**

Keywords*—*Insertion; deletion; array; linked list; binary search; linear search

I. Introduction

The efficient information retrieval, insertion and searching is the basic need for most of the applications of this modern computing era. These applications require efficient data structures to store and retrieve large amount of information. Normally the information is either stored in arrays or linked list. In arrays, searching can be done efficiently using binary search technique. As binary search is less computationally intensive as compared to the linear search especially when the data set is too large, so it is the desired searching technique used by many real-time applications. But the problem using array is that insertion and deletion of nodes requires more shifting operations which becomes a hurdle to use it in real-time scenarios. Linked list efficiently resolve this issue of real-time insertion and deletion of nodes as it requires only updating the pointers values, so it seems more appropriate to use it in real-time applications. But the main problem using linked list is that binary search cannot be implemented on it directly because we cannot search a node without traversing all the previous nodes. This is because the memory allocation of linked list is not contiguous and is allocated at run time while in arrays the nodes reside on contiguous memory locations.

Linear search algorithm searches a node from array or linked list by inspecting each of the nodes in it and comparing it with the search node. In linear search, time required to find a node directly depends on the total number of elements in the array or linked list. So, the complexity of linear search is O(N) [1], [2] as in Big-O notation. This search technique has the simplest implementation, so it can be applied to array list and all types of linked lists. But it is not efficient when the size of the list is too large. It is useful only when the size of an array or a linked list is small. Binary search is more efficient searching technique and is quite suitable when the number of nodes is more in an array list. The requirement of binary search algorithm is that the elements of an array must be in sorted form [1], [2]. Every iteration of this algorithm makes half the search interval of its previous iteration, so lesser number of comparisons is required to search a node. The complexity of binary search algorithm is O($\log_2 N$). So if we can manage to apply binary search efficiently on linked list, it would become an ideal data structure for supporting real-time insertion, deletion and searching. It may enhance the performance of many real-time applications like vehicle exit-control system.

In this paper, a hybrid solution is presented in order to facilitate the real-time applications in terms of efficient insertion, deletion and searching. In it a linked list is used to store nodes data and a combination of linear and binary searching techniques are used to efficiently find a node. The proposed technique outperforms the existing solutions for these kinds of applications.

The rest of the paper is organized as follows. Section II presents the related work. The proposed methodology is presented in Section III. Section IV presents the experimental analysis. Finally, the conclusion is drawn in Section V.

II. Related Work

As discussed in the previous section, binary search algorithm can only be applied to sorted array whether it is static or dynamic. This algorithm cannot be directly implemented to linked list [2]. Although the advantages of binary search can be obtain through organization of array elements in non linear data structure tree [3]. Binary search tree searches an element in equal amount of time as taken by binary search O($\log_2 N$)[1], [4]. But it is difficult to maintain and manipulate binary search tree.

The second option to implement binary search on linked list is to copy all the elements of linked list into either sorted array or a binary search tree [5]. This option is again not practical for maintenance of the data as each time searching will be faster but require more processing of creating and copying elements.

Extra overload will be faced by processor in obtaining the benefits of binary search. Several other researchers worked in this domain. Kumar et al. [6] discussed that linear search is efficient than binary search if we add sorting time also in case of binary search. The argument can be nullified in applications where sorted data is a requirement. Arora et al. [7] presented a two way linear search approach through which searching is performed from both ends of the list. It is efficient only in cases where the node to be searched belongs to the second half of the list. Chadah et al. [8] tried to reduce the worst case time of binary search algorithm by increasing number of comparisons in each iteration. Naidu et al. [9] targeted the large memory requirements of doubly linked list and proposed an implementation of single linked list to achieve the benefits of doubly linked list. The Ex-Or operation was used in single list in order to traverse in both direction.

The third option is to derive a new methodology that can perform computationally efficient searching in linked list. This may help to develop a real-time information storage and retrieval systems that allows searching, insertion and deletion operations.

III. Proposed Solution

Let us consider a doubly linked list structure that consists of a set of sequentially linked records called nodes. Each node contains 'info' and 'links' fields. The 'info' field stores the information and reference or pointer to the previous and next node in the linked list containing 'links' field. In doubly linked list, navigation is possible in both forward and backward ways easily as compared to single linked list. According to the proposed solution, linked list is organized in order of 'info' field in an ascending order. After that, a track of few selected key nodes is maintained in a separate array of pointers. This pointer array is known as sparse array.

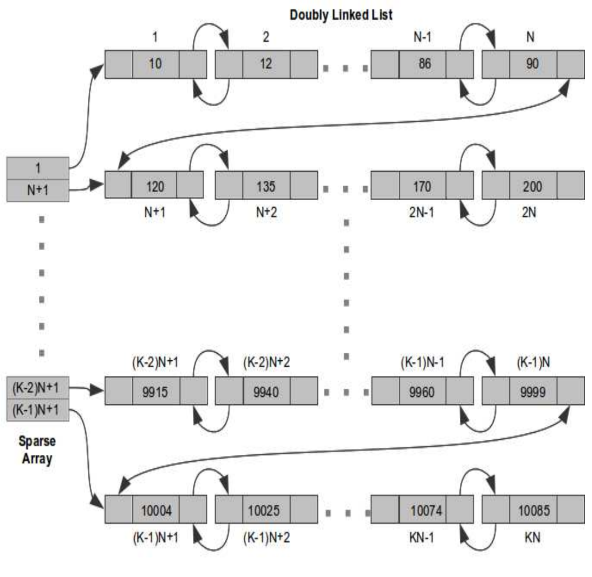

Fig. 1. Arrangement of the doubly linked list and sparse array.

Fig. 1 shows the arrangement of the sorted doubly linked list and sparse array. The sorted linked list consists of $K * N$ number of elements and the first node of each block is taken as key node. The address of each key node is stored in the sparse array. In Fig. 1 there are K numbers of key nodes and N is the number of entries between two key nodes. In this work, sparse array is used to perform the searching, insertion and deletion operations in the linked list. The following sections describe the searching, insertion, deletion and up gradation of sparse array.

A. Searching Operation

To search the desired pattern in the linked list, a hybrid binary linear search technique (HST) is proposed based on the sparse array. In this technique, initially binary search is performed using key nodes sparse array. If the desired pattern is present in the key nodes than output of binary search is its exact location i.e. for searched pattern 10, 120, 9915 and 10004. On the other hand, if the data we are looking for is not located in the key nodes than the outcome of binary search are two key nodes 'Ki' and 'Kf' between whom the desired pattern can be laid. In this case linear search is performed on linked list records between 'Ki' and 'Kf' to find the exact pattern location. For example, if we want to search pattern '9960' than outcome of binary search will be key node '$(K-2)N+1$' and '$(K-1)N+1$' using these key nodes linear search can be performed on linked list. The block diagram of the proposed searching algorithm is shown in Fig. 2.

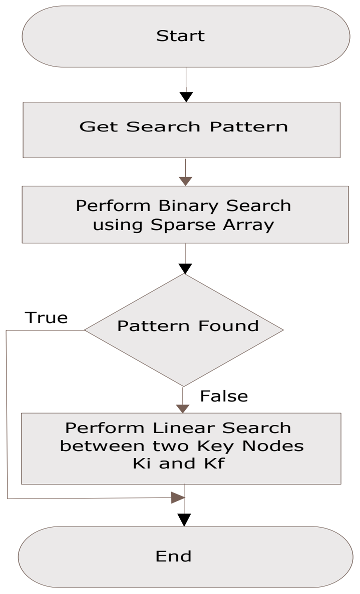
Start
Get Search Pattern
Perform Binary Search using Sparse Array
Pattern Found
True
False
Perform Linear Search between two Key Nodes Ki and Kf
End

Fig. 2. Block diagram for the proposed search technique.

In the proposed HST, binary search helps to reduce the search time by either giving exact match or by reducing the search space by giving the address of two key nodes as a linear searchs starting and ending point. The pseudo code for binary and linear search using sparse array is presented in Algorithm 1 and 2, respectively.

Algorithm 1: SparseArrayBinarySearch(S,n, Pattern,N)

- Input: Sparse array S, n is the total number of elements in S, Pattern is the value to be search and N is the number of records between two key nodes
- Output: Position k such that S[k] $\rightarrow$ info = Pattern, or two key nodes Ki and Kf for linear search if desired

pattern is not found.

```
1)  i ← 0, j ← n-1, k ← (i+j)/2, Ki ← -1, Kf ← -1, and r ← 0
2)  while (i ≤ j)
3)  do
4)  if(S[k] → info > Pattern) then j ← k-1 and r ← 1
5)  else if(S[k] → info < Pattern) then i ← k+1 and r ← -1
6)  else return k // successful search
7)  if(i > j)
8)  Ki ← k
9)  if(r > 0) then Ki ← Ki-1
10) if(Ki > N-1) then r= N-1
11) else if( Ki < 0) then r= 0
12) else Kf ← Ki+N return Ki and Kf //end if(i > j)
13) k ← (i+j)/2 //end while
```

Algorithm 2: SparseArrayLinearSearch(S,Pattern,Ki, Kf)

- Input: Sparse array S, Pattern to be search, linear search starting and ending points Ki and Kf
- Output: Position i such that S[i] → info = Pattern, -1 if desired pattern is not found

```
1) i ← Ki and LN ← S[i]
2) while (i ≤ Kf)
3) do
4) if(LN → info = Pattern) then return i
5) else LN = LN → next
6) i ← i + 1 //end while
7) if (i < Kf) then return i
8) else return -1
```

B. Insertion Operation

To insert a given pattern in a sorted linked list, following five steps are involved. First, a new node is allocated and input pattern is stored in the 'info' field of the node. Second, to concatenate the new node with sorted linked list, insertion location is determined by using HST. Third, 'link' fields of new node; store the addresses of its previous and next node in sorted linked list. Fourth, the 'link' fields of the previous and next node are modified according to new node. Finally, up-gradation of sparse array is made which is necessary for the further operations. Fig. 3 shows the block diagram for node insertion operation.

C. Deletion Operation

This operation is used to delete a specific node from the sorted linked list. The node deletion is a four step process. First, a node whose 'info' field contains the desired pattern is determined using proposed HST. Second, redirection of 'links' is performed in the previous and next nodes of the node to be deleted. Third deleted node is De-allocated. Finally, up-gradation of sparse array is performed according to modified linked list for further operations. Fig. 4 shows the block diagram for node deletion operation.

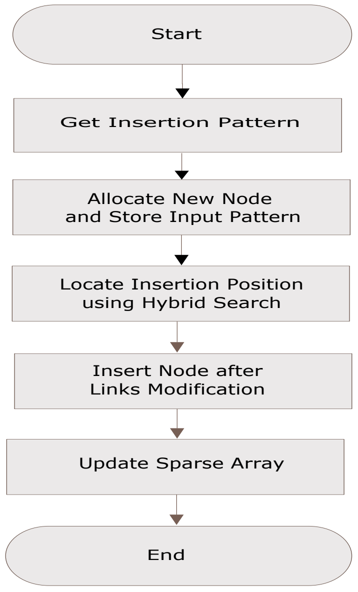
Start
Get Insertion Pattern
Allocate New Node and Store Input Pattern
Locate Insertion Position using Hybrid Search
Insert Node after Links Modification
Update Sparse Array
End

Fig. 3. Block diagram for node insertion operation.

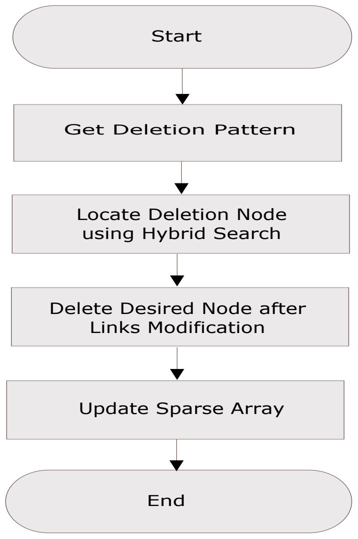
Start
Get Deletion Pattern
Locate Deletion Node using Hybrid Search
Delete Desired Node after Links Modification
Update Sparse Array
End

Fig. 4. Block diagram for node deletion operation.

D. Updating of Sparse Array

The insertion and deletion of nodes from the linked list demand up-gradation of the sparse array. The up-gradation is performed according to deletion or insertion operation. In case of insertion operation, all the key nodes that exist beyond the insertion location will be replaced with their corresponding next node in the linked list. On the other hand, after deletion operation all the key nodes that exist beyond the deletion location will be replaced with their corresponding previous node. The pseudo code for up-gradation of the sparse array is given in Algorithm 3.

Algorithm 3: UpDateSparseArray(S,k,m,n)

- Input: S is sparse array, k denotes the index from where S is needed to be upgrade, m indicates the operation insertion or deletion after which up-gradation is required and n is the numbers of entries in S.

- Output: Upgraded sparse array S

```
1) i ← K
2) while (i ≤ n)
3) do
4) if(m = 0) then S[i] = (S[i] → previous) // in case of deletion
5) else S[i] = (S[i] → next // in case of insertion
6) i ← i + 1 //end while
```

IV. Experimental Analysis

The experiments are conducted on PC with intel-core i3-2100 CPU @ 3.1 GHz and 2 GB RAM. The proposed system is a combination of linked list and sparse array based hybrid search (HS-LL) technique. The comparison of the proposed information storage and retrieval system is done with two possible scenarios. First one is based on array using binary search (BS-AR) and second is based on the linked list and linear search (LS-LL) methodology. The experiments are performed on sorted array and linked list having different range of entries between 5000 and 100,000.

Tables I, II and III list the experimental results in terms of the time taken Ts to search, insert and delete the entries using three information storage and retrieval systems, respectively. The experiments are performed by considering the boundary cases i.e, by performing searching, inserting and deletion operations at the start and end position (index or node) in both array and linked list.

Table I indicates that the data searching from sorted array using binary search (BS-AR) performs equally well in both cases either data to be search is located at the start or end of the array. It is observed that data searching in linked list using linear search technique (LS-LL) is worse when desired data is located at the end of the linked list. In this case, searching time is directly related with the number of entries in the linked list. Table I shows that proposed solution (HS-LL) almost perform equally well as that of binary search in array. Fig. 5 shows the performance analysis of three techniques in terms of time taken to search a node or entry located at the middle of linked list and array. It depicts that HS-LL and BS-AR perform equally well and have better performance than LS-LL technique.

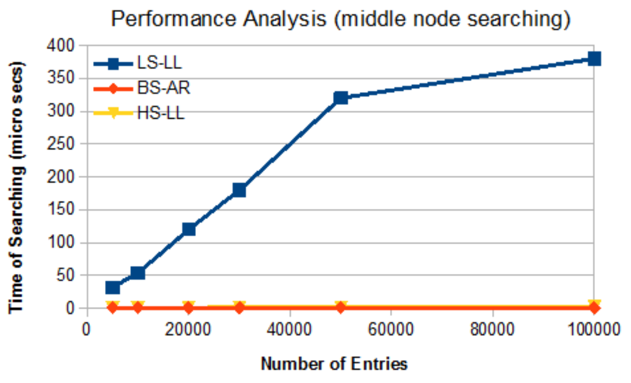

Fig. 5. Data searching using LS-LL, BS-AR and HS-LL.

Table II illustrates that data insertion at the start of the sorted array using BS-AR takes significant time due to shifting of huge amount of data. Contrarily, array insertion is efficient when element is inserted at the last index because no shifting operations are required to sort the array data. Table II also demonstrates that linked list insertion using linear search (LS-LL) at the start is efficient because insertion position is found at the first attempt during linear searching. On the other hand, node insertion using linear search at the end gives the worse performance because huge processing time is consumed to search the node insertion position. Table II depicts that the linked list insertion with proposed hybrid search technique (HS-LL) perform equally well in both cases either data is inserted at the start or end of the sorted linked list. Fig. 6 depicts the performance analysis of three techniques in terms of time taken to insert a node or entry at the middle of linked list and array. It shows that HS-LL have better performance than BS-AR and LS-LL technique.

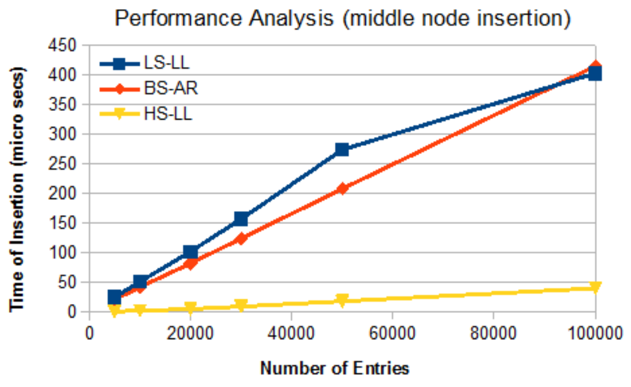

Fig. 6. Data insertion using LS-LL, BS-AR and HS-LL.

Table III shows that for deletion operation behavior of all the techniques is similar to that of insertion operation. The data deletion from the start of the sorted array takes significant time due to shifting of all array elements. On the other hand, it takes small time when element is deleted from the end of array because shifting operations are not required in this case. Table III lists that linked list deletion using linear search from the start of the sorted linked list is efficient because desired node is found at the first attempt during linear searching. Contrarily, last node deletion takes maximum time due to huge searching time to locate the desired node. Table III depicts that the linked list deletion with the proposed hybrid search technique perform equally well in both cases either data is deleted from the start or end of the sorted linked list. Fig. 7 shows the performance analysis of three techniques in terms of time taken to delete a node or entry from the middle of linked list and array. It shows that HS-LL have better performance than BS-AR and LS-LL technique.

The experimental results indicate that the proposed HS-LL solution performs equally well for data searching, insertion and deletion either at start or end. It also outperforms the rest of the two possible scenarios.

It should be noted that the performance of proposed solution is highly correlated with the size of the sparse array. For the large size of sparse array, huge shifting operations are required for its up-gradation. Contrarily, small size reduces the overhead related to sparse array up-gradation process at the cost of increase in linear search. Therefore, size selection of

TABLE I. DATA SEARCHING TIME (μSEC)

No. of Entries	5000		10000		20000		30000		50000		100000	
Position	Start	End	Start	End	Start	End	Start	End	Start	End	Start	End
BS-AR	1	1	1	1	1	1	1	1	1	1	2	1
LS-LL	1	75	1	109	1	248	1	348	1	720	1	771
HS-LL	1	1	1	1	1	1	1	2	2	2	3	3

TABLE II. DATA INSERTION TIME (μSEC)

No. of Entries	5000		10000		20000		30000		50000		100000	
Position	Start	End	Start	End	Start	End	Start	End	Start	End	Start	End
BS-AR	43	1	85	1	164	1	247	1	415	2	831	2
LS-LL	1	51	1	102	1	204	1	312	1	545	1	804
HS-LL	2	1	3	1	8	3	12	6	24	11	58	24

TABLE III. DATA DELETION TIME (μSEC)

No. of Entries	5000		10000		20000		30000		50000		100000	
Position	Start	End	Start	End	Start	End	Start	End	Start	End	Start	End
BS-AR	46	1	91	1	181	2	273	1	458	1	913	2
LS-LL	1	60	1	125	1	251	1	396	1	633	1	768
HS-LL	2	1	3	2	7	3	13	6	22	10	55	22

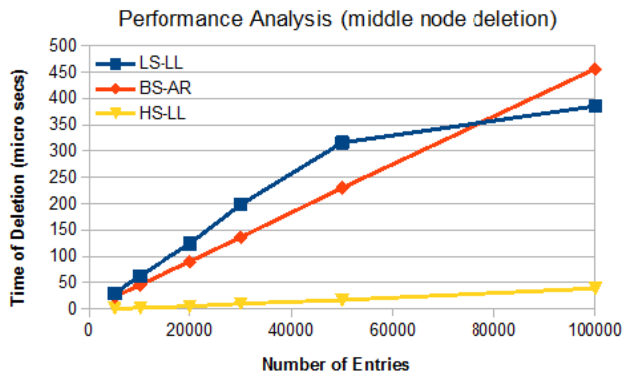

Fig. 7. Data deletion using LS-LL, BS-AR and HS-LL.

sparse must be optimum in such a way that it neither increases the shifting operations nor hurts the searching performance. In this experimental analysis the size of sparse array is taken as 64.

V. CONCLUSION

This paper presented an efficient information storage and retrieval system to facilitate the real-time applications. In this solution, linked list is used to store the information and a hybrid linear binary search technique based on sparse array is proposed to perform efficient data insertion, deletion and searching operations. The experimental results reveal that the proposed methodology outperforms the existing techniques for such kinds of applications.

REFERENCES

[1] Jean-Paul Tremblay and P. G. Sorenson: "An Introduction to data structures with applications", Mcgraw Hill Computer Science Series, 2nd Edition.

[2] A. Oommen and C. Pal: "Binary Search Algorithm", Journal Of Innovative Research In Technology, 1(5), 800-803, 2014.

[3] S. Pushpa1 and P. Vinod: "Binary Search Tree Balancing Methods: A Critical Study", International Journal of Computer Science and Network Security, 7(8),2007.

[4] P. P. Thwe and L. L. W. Kyi: "Modified Binary Search Algorithm for Duplicate Elements", International Journal of Computer and Communication Engineering Research (IJCCER), 2(2), 77-81, 2014.

[5] P. Das and P. M. Khilar: "A Randomized Searching Algorithm and its Performance analysis with Binary Search and Linear Search Algorithms", The International Journal of Computer Science and Applications (TIJCSA), 1(11), 11-18, 2013.

[6] D. Kumar and M. Sharma: "Binary search is faster than the linear search", International Journal of Innovative Research in Technology, 1(5), 796-799, 2014.

[7] N. Arora, G. Bhasin and N. Sharma: "Two way Linear Search Algorithm", International Journal of Computer Applications, 107(21), 6-8, 2014.

[8] A. R. Chadha, R. Misal and T.Mokashi: "Modified Binary Search Algorithm", International Journal of Applied Information Systems (IJAIS), 7(2), 37-40, 2014.

[9] D. Naidu and A. Prasad: "Implementation of Enhanced Singly Linked List Equipped with DLL Operations: An Approach towards Enormous Memory Saving", International Journal of Future Computer and Communication, 3(2), 2014.