STM JOURNALS

International Journal of
Data Structure Studies

Volume 1, Issue 1, 2023
January—June
DOI (Journal): 10.37591/IJDSS

https://journals.stmjournals.com/ijdss

Review IJDSS

Array-Linked Data Structure: Introducing a Hybrid Model of Memory Management and Faster and Easier Insertion and Reallocation Procedures

Kartik Srivastava[1,2]

Abstract

*Here, I have introduced a new data structure titled "Array-Linked Data Structure". It incorporates a hybrid model of memory allocation, introducing a new insertion procedure in an existing data structure which is faster than that for arrays. It also offers O(c) access time where c is a constant. The access time is worse than O(1) for an array but still better than that for a linked list since the index of data could be directly supplied to the access procedure here. The data structure also maintains a DS table for each allocated block of memory which keeps track of auxiliary data structures in the main block. Insert call of the structure offers a complexity of O(t*log(t)) where t is the number of auxiliary data chains in the main block where insertion is supposed to happen. Rest of the procedures are discussed in detail with proofs. The main element of the structure is that it offers a faster insertion procedure and easier reallocation procedure, while maintaining a hybrid model of contiguous memory allocation*

Keywords: Hybrid memory allocation, insertion, DS table

INTRODUCTION

Arrays and Linked list data structures are widely used in almost all computer applications. Arrays are remarkable data structures in the sense that they provide contiguous memory allocations, fast retrieval of data by index id and easy maintenance in terms of memory and overall performance. However, arrays do lack an easy solution to insertion of data at a given location in an existing implementation [1]. They also require significant performance overhead in the reallocation of an existing implementation, since, both the above two operations require shifting of data which could comprise of the entire dataset.

Linked lists provide some of the solutions to these problems by doing away with contiguous memory allocations of its nodes but allows for easy access of the data given a node and insertion of new nodes is easier as it only involves breaking and re-creating the links. However, linked lists lose the advantages of contiguous memory allocations.

***Author for Correspondence**
Kartik Srivastava
E-mail: professkartik@gmail.com

[1]Student, Chemical Engineering, Indian Institute of Technology, Kanpur, Uttar Pradesh, India
[2]Student, Masters of Science, Chemical Engineering, Iowa State University, USA

Received Date: July 05, 2023
Accepted Date: July 12, 2023
Published Date: August 14, 2023

Citation: Kartik Srivastava. Array-Linked Data Structure: Introducing a Hybrid Model of Memory Management and Faster and Easier Insertion and Reallocation Procedures. International Journal of Data Structure Studies. 2023; 1(1): 1–11p.

Hereby, I present a new data structure, titled "Array-Linked Data Structure" that introduces a hybrid model of contiguous memory allocation and allows fast insertions at a given index, fast retrieval of data and faster reallocation and deletion procedures [2].

THE DATA STRUCTURE

We can assume the basic representation of the data structure as an array of doubly connected linked list with each index at array holding onto a link of the linked list [3]. The data structure will also have a DS table which would keep track of auxiliary data chains formed by insertion operations as shown in Figures 1 and 2.

© STM Journals 2023. All Rights Reserved

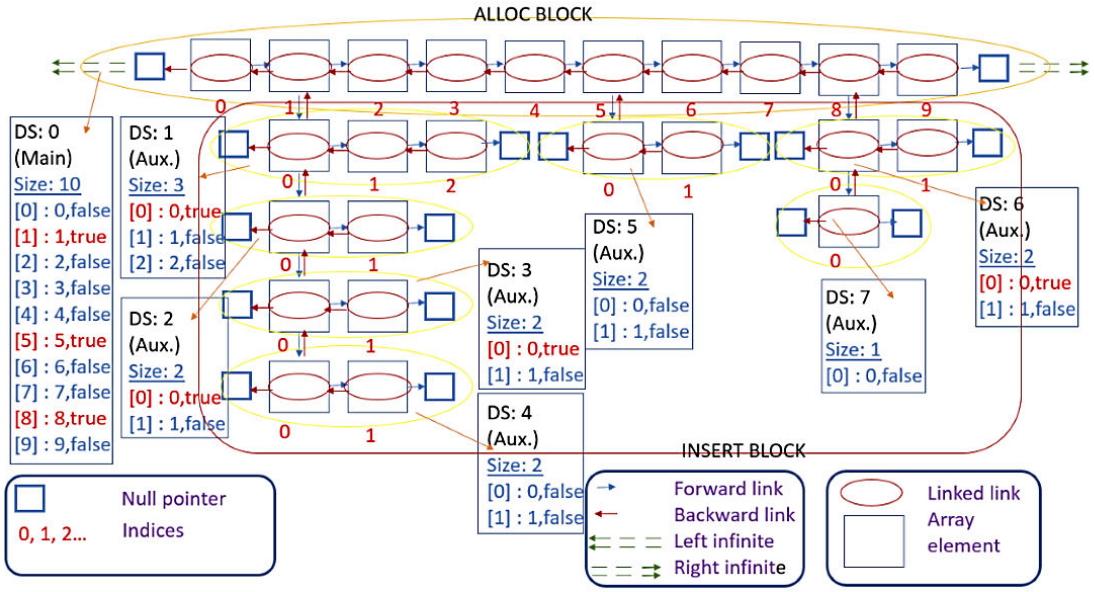

Figure 1. Before DS table is sorted.

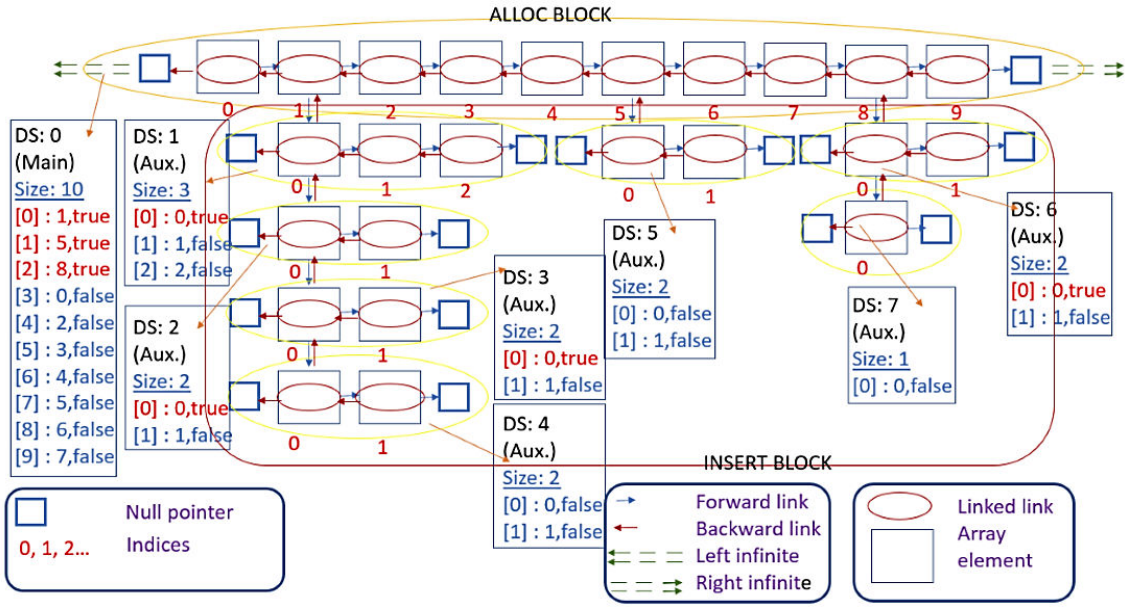

Figure 2. After DS table is sorted.

Figures 1, 2 shows the layout of the data structure. We have an alloc block which is the data structure that gets allocated at first. Each alloc/realloc/insert call implements the data structure with a DS: (id=0 in case of alloc operation denoting the main block and subsequent numbers representing further allocation/insertion operations); a table (which could be an array) that holds onto Boolean entries corresponding to a false if a cell had no insertion operation performed on it (initial state); else if the cell has an auxiliary allocation performed by an insertion operation then it will hold onto a true entry indicating that the corresponding cell has an auxiliary data structure [4]. Above workflow applies to all auxiliary data structures in a recursive fashion. Insertion procedure would also keep a sorted list of the entries in the DS table having a true value in the order of their corresponding indices. Sort procedure would run in O(t*log(t)) time complexity; where t corresponds to the number of true entries in the table. A Realloc() operation, i.e., which would expand the data structure in horizontal direction could be implemented by maintaining a separate DS table for it or its DS table (i.e., further allocated entries) could be appended to the main table. Both the implementations would have its advantages and

© STM Journals 2023. All Rights Reserved

disadvantages, which we will discuss later. It is to be noted that a Realloc() operation would create an additional allocated block in the horizontal direction without having any auxiliary data structures to it [5]. However, an Insert() call with a set of data entries to be inserted at an index would allocate a block of memory and add it as an auxiliary data structure at the passed index, connecting the pointers of the index element and the one at the 0th block of newly allocated block which acts as auxiliary data structure for the passed index. Each DS table will hold onto a size element which would hold onto the size of the contiguous memory block. The "Array-Linked Data Structure" would support Alloc(int N), ReAlloc(AL* BL, int N), Access(AL* BL, int i), Insert(AL* BL, int i, int d[m]), Delete(AL* BL, int i, int j), DeleteAll(AL* BL), Traversal(AL* BL), TraversalBackward(AL* BL) and any other suitable operations. But the above eight operations are the basic key operations which we would discuss further [6].

A light-weight representation of "Array-Linked Data Structure" in C++ with int type as data is as follows:

```
Struct Links                    Struct Array-Linked
{                       {
        int data;               int size;
    Links* front;               pair<int, bool> DS[number_of_entries]; int DS_Id;
    Links* back;                //All entries set to false in the beginning
    Links* down;                Links* start;
        Links* up;
                                        private:
                                            SortDS();
};                      };
```

Next, we will discuss each basic operation supported by the data with its complexity and performance.

Alloc(int N)

The alloc call would take in the size of data as input and allocate a contiguous block of memory. The implementation would be the same way as it is done for the array memory blocks implementation [7]. The linked list created would connect the double links at the allocation stage. A DS table of the same size would be allocated too at the allocation stage with initially all entries being set to FALSE and all indices in the pair set to the DS indices. A size variable to hold on to the size of the allocated block would be held on by the structure. The determination of the type of the size would be based on the size of the allocation block provided as input to the call [8].

---

pseudo-code: Alloc(int N)

```
procedure AL* Alloc(int N)
//Here AL is the return type which is our data structure in consideration
        if N<=0 then
            return NULL
        end if
   //Allocate a memory size of block size N using new or malloc =BL
   //Traverse over the memory block : i=0 (initially)
                (*BL+i)->down = NULL
                (*BL+i)->up = NULL
                if i>0 then
                    (*(BL+i))->back=(*(BL+i–1))
        (*(BL+i–1))->front=(*(BL+i))
```

© STM Journals 2023. All Rights Reserved

```
            end if
                (*(BL+i–1))->front=(*(BL+i))
                        (*(BL+i))->back = (*(BL+i–1))
        //Allocate a DS table of size N using new or malloc = DS
        //Traverse over the memory block of DS table : m
            (*(DS+m)).first =m
            (*(DS+m)).second = FALSE
        BL->DS=DS
        size =N;
        DS_Id=0;
        return BL
end procedure
```

Proof

The proof of correctness is established from the well-established contiguous memory allocation procedure for array. The above allocation call uses the same procedure internally.

However, we would consider some base and end cases.

Case 1: N<=0

This would return NULL from the second line of procedure; which covers the empty allocation case.

Case 2: N<|M| (where M is an integer type upper bound)

This allocate blocks of memory from lines 6–9 Connection of links is obvious.

Case 3: N>|M| (large data input)

The allocation in line 4 should take care of this; if not an error could be raised after line 4.

This Completes the Proof of Correctness for Alloc(int N) Call

The time performance of the block would involve allocation of N memory blocks and allocation of a DS table. It would also require traversal over the memory block to establish links and traversal over the DS table to initialize bool values [9]. Setting of the size and DS_Id are additional two steps. Each linking step would take two operations.

To improve the performance slightly, the two traversals could easily be combined into one.

So, the overall time complexity of the allocation procedure would be:

=2*N* time_allocation (ta)+3*N * time_traversal (tr)+2

=5*N*t+2 (assuming ta~tr=t)

Hence, the time complexity of allocation block would be O(N) in all cases which is not so bad considering the same for the existing allocation procedures.

Space requirements of the procedure would be:

=2*N* size_block (sb)+3*N* size_link(sl)+2

<5*N*s+2 (assuming sl<sb)

Hence, the space requirements would be 5*N+2 in all cases. This is slightly greater than O(N) requirements of the standard array allocation procedure, 5*N>N.

Insert(AL* BL, int i, int d[m])

Here, i is the index at which new data is to be inserted and d[m] is the provided data of size m. We are considering the case of integer data types here. Insertion call would allocate a new block of size m

© STM Journals 2023. All Rights Reserved

and connect the pointers at index i and the first block of memory in the newly allocated block. It would also call SortDS() internally which would sort the DS table's TRUE entries in the ascending order of its indices [10]. It is to be noted here that the sort call would only run on the indices that have auxiliary data chains associated with them and hence it would typically involve fewer entries than the number of entries in the entire allocated block. Hence, t*log(t) performance is expected to be significantly lower than n*log(n); where t is the number of entries in the allocated block having an auxiliary chain and n is the number of entries in the entire allocated block.

```
---------------------------------------------------------------------------------------------------------------
pseudo-code: Insert(AL* BL, int i, int d[m])
---------------------------------------------------------------------------------------------------------------
procedure AL* Insert(AL* BL, int i, int d[m])
        int size_d=d.size()
        if size_d<1
            return
        end if
        //Allocate a memory size of block size_d using new or malloc = ABL
        BL[i]->down=*(ABL+0)
        *(ABL+0)->back = BL[i]->down
        (BL->DS[i]).second = TRUE
        BL->SortDS()
        return BL[i]
end procedure
---------------------------------------------------------------------------------------------------------------
```

Proof

The proof of correctness is similar to that for Alloc(int N) call. The joining of the links of ith entry of parent structure to that of the first block of auxiliary data structure is straightforward too.

SortDS() procedure would use a preexisting implementation of a sorting procedure like quick-sort or merge-sort. Although heap-sort would be best suited for this purpose since the SortDS() procedure involves addition of a new entry to an already sorted list. Sorting could be done in-place in the DS table using a slightly modified version of heap-sort. We would not be discussing the implementation of heap-sort here since it is a well-established, widely used procedure.

This Completes the Proof of Correctness for Insert(AL* BL, int i, int d[m]) call

The time performance of the insert procedure would involve allocation of blocks of size of the data structure d[m] and sorting the DS table.

So, the overall time complexity of the Insert procedure would be:
= size_d * time_allocation (ta)+(t*log(t))*time (to)+3
(here ta is the time for 1 allocation and to is the average time for one typical call in the sorting procedure)

Hence, the time complexity of allocation block would be O(size_d+t*log(t))

Case 1: size_d>t*log(t) (for all t)

In this case, time complexity of the procedure would be O(size_d) which is the best possible complexity for inserting size_d elements.

Case 2: size_d<=t*log(t) (for all t)

In this case, time complexity of the procedure would be O(t*log(t)) which in the worst case can be the O(n*log(n)); where n is the number of elements in the DS table of base data structure.

© STM Journals 2023. All Rights Reserved

So, the time complexity would depend on the number of inserted elements and the number of already inserted auxiliary data structures in the DS table of base data structure. It is to be noted, that in cases of O(t*log(t)) situations I expect the complexity to be lesser than O(n*log(n)) in average cases. We will look at this point in detail in further sections.

Space requirements of the procedure would be:
=2* size_d* size_block (sb)+2*N* size_link(sl)+2
<4*N*s+2 (assuming sl<sb)

Hence, the space requirements would be 4*N+2 in all cases. This is slightly greater than O(N) requirements of the standard array allocation procedure, 4*N>N.

Access(AL* BL, int i)

Here i is the index at which we need to access and return the data from "Array-Linked Data Structure". First step would be to locate the index i in the BL block and last step would be to return the data at the location. Approach is to loop over the DS table for indices having an auxiliary data structure and are smaller than i. If we cannot locate index i we would return a placeholder value or an error could be raised as well.

```
---------------------------------------------------------------------------------------------------------------------
pseudo-code: Access(AL* BL, int i)
---------------------------------------------------------------------------------------------------------------------
procedure int Access (AL* BL, int i)
        if i<0 || i>=BL->size()
            return 0 //placeholder value
        end if
        return AccessSupport(BL, i)
end procedure
---------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------
pseudo-code: AccessSupport (AL* BL, int i)
---------------------------------------------------------------------------------------------------------------------
procedure int AccessSupport (AL* BL, int i)
            if i==0
                return BL[i]
            end if
            for loop BL->DS : d
                if (BL->DS[d]).second == FALSE
                        break
                        //we break here because in a sorted DS table all
                        //FALSE entries are at the bottom
                else
                        if i – (BL->DS).first<0
                            return BL[i]
                        else
                            return AccessSupport(BL[(BL->DS).first], i – (BL->DS).first)
                        end if
                end if
            end for loop
            if  i<BL->size()
                return BL[i]
            else
```

© STM Journals 2023. All Rights Reserved

```
            return 0 //placeholder value
        end if
end procedure
```

Proof

Case 1: i<0 or i>=BL->size()

Line 2 of the procedure Access takes care of this case. We are returning a placeholder value 0 here which could easily be replaced by raising an error.

Case II: i=0

Lines 1–2 of the AccessSupport procedure takes care of this case.

Case 2: ALL DS table entries are FALSE //we have no auxiliary data structures

Lines 16–19 takes care of this case. In this case entry would be found, if at all in the parent data structure.

Case 3: i< index of entry of first occurrence of TRUE in the DS table

Lines 10–11 of AccessSupport procedure takes care of this case.

Case 4: i> index of entry of first occurrence of TRUE in the DS table

Line 13 of AccessSupport procedure takes care of this case by calling the same call recursively. All other cases would be taken care of in the recursive call.

Case 5: i goes through recursive calls and is located after all auxiliary data structure chains; in the main data structure

Lines 17–18 takes care of this case.

Case 6: i goes through recursive calls but isn't present in the entire data structure

Line 20 takes care of this case.

This completes the proof of correctness of Access(AL* BL, int i)

The time complexity of the procedure in the worst case would involve two comparison calls one each at lines 1 and 18 and two comparison calls for each auxiliary chain data entry. This amounts to 2+2*Nt (Total number of cells in the entire data structure including auxiliary data structures).

The standard array access procedure does not involve any comparisons but only one access call. So, the bottleneck with this procedure is the time taken for comparison which mainly involves access of data from pointer locations for each comparison call; which will not be so bad considering the time for access is small. This is the penalty paid for optimizing the insertion procedure to do away with shifting of the data which would involve copying each shifted entry.

Space complexity involves the creation of additional stacks which could be of the order of the depth of the auxiliary data structure; however, it could always be compensated by implementing the same procedure by replacing the recursive call with an iterative implementation.

ReAlloc(AL* BL, int N)

The realloc call would take in the size of data as input and allocate a contiguous block of memory. The implementation would be the same way as it is done for the array memory blocks implementation. The linked list created would connect the double links at the reallocation stage.

Everything else is same as for alloc procedure except for the fact that pointer at index would be connected to the one in the newly allocated block's first pointer.

© STM Journals 2023. All Rights Reserved

```
---------------------------------------------------------------------------------------------------------------------
pseudo-code: ReAlloc(AL* BL, int N)
---------------------------------------------------------------------------------------------------------------------
procedure AL* ReAlloc(AL* BL, int N)
        //Here AL is the return type which is our data structure in consideration
        if N<=0 then
            return BL
        end if
   //Allocate a memory size of block size N using new or malloc = RBL
   //Traverse over the memory block: i
             if i>0 then
                 (*(RBL+i))->back=*(RBL+i–1)
             end if
                 (*(RBL+i–1))->next=*(RBL+i)
        //Allocate a DS table of size N using new or malloc = DS1
        //Traverse over the memory block of DS table: m
             (*(DS+m)).first =m
             (*(DS+m)).second = FALSE
        RBL->DS=DS
        size=N;
        DS_Id=0;
        int size_original = BL.size()
        *(BL + size_original –1)->next =*(RBL+0)
        *(RBL +0)->back =*(BL+size_original–1)
        return BL
end procedure
```

Proof

The proof of correctness is similar to that for Access(int N) procedure and needs no further elaborations. Performance analyses are like that for Access(int N) too.

Delete(AL* BL, int i, int j)

```
---------------------------------------------------------------------------------------------------------------------
pseudo-code: Delete (AL* BL, int i, int j)
---------------------------------------------------------------------------------------------------------------------
procedure AL* Delete (AL* BL, int i, int j)
      if i>j || i<0
            return BL
      AL* iPrevPtr, jNextPtr, prevPtr
      int index =0
      AL*t=BL
      prevPtr = NULL
      iPrevPtr = NULL
      jNextPtr = NULL
      while index <=j && t != NULL
            if index ==i
               iPrevPtr = prevPtr
            end if
            if index ==j
               jNextPtr = t->down != NULL ? t->down : t->next
            end if
            prevPtr =t
```

© STM Journals 2023. All Rights Reserved

```
            if t->down != NULL
              t = t->down
            else
              t = t->next
            end if
            if index >=i
               free(BL[index])
            end if
            index++
      end loop
      if iPrevPtr != jNextPtr
            if iPrevPtr != NULL
                  iPrevPtr->next = jNextPtr
            end if
            if jNextPtr != NULL
                  jNextPtr->back = iPrevPtr
            end if
      end if
      if i==0
         return jNextPtr
      end if
      return BL
end procedure
```

Proof

Case 1: i>j || i<0

Lines 1–2 takes care of this case.

Case 2: BL is NULL

Control flow does not enter the while loop and return is NULL at line 36.

Case 3: i<=j

Condition index <=j in while loop at line 9 takes care of this case.

Case 4: i=0

Lines 35–37 takes care of this case.

Case 5: j> Number of all entries in the entire data structure

Condition t!= NULL in while loop at line 9 takes care of this case.

This completes the proof of correctness of Delete (AL* BL, int i, int j)

Time performance of the procedure is O(j) in all cases and Space requirements are O(1).

DeleteAll(AL* BL)

DeleteAll procedure is same as Delete except that the procedure needs to be modified to check for the last index to be deleted. Above procedure too could be used for this same purpose by passing an out of bounds index.

Traversal (AL* BL)

Traversal procedure too would be written out same as that for Delete; with the change of free call being replaced by a print call or access call depending on the requirements.

© STM Journals 2023. All Rights Reserved

TraversalBackward (AL* BL)

```
---------------------------------------------------------------------------------------------------------------
pseudo-code: TraversalBackward (AL* BL)
---------------------------------------------------------------------------------------------------------------
procedure AL* TraversalBackward (AL* BL)
        AL*t=BL
        if t->down!= NULL
              TraversalBackward(t->down)
        else if t->next!= NULL
              TraversalBackward(t->next)
        else
             Print(t)
       end if
end procedure
---------------------------------------------------------------------------------------------------------------
```

Proof

The proof of correctness is obvious. Time Complexity is of the order of the size of the data structure and space complexity would involve creation of stack traces which could be reduced to order of 1 using an iterative implementation.

CONCLUSION

Here, I have proposed a new data structure titled "Array-Linked Data Structure". The proposed data structure offers a faster solution to insertion problems posed under the context of array data structure; also, it offers a slightly slower access operation, a penalty paid for offering a faster insertion procedure. Nevertheless, it does provide a hybrid form of contiguous memory allocation model, easier insertion procedure which is the key feature of the data structure and a slightly slower, nevertheless reasonable access procedure. It also offers reasonably fast traversals in both directions and deletion procedures. There is one area which needs to be looked at more deeply which is the expected number of auxiliary data structures after a given number of insertion operations. There are many scoops of improvements like making a faster access procedure, etc. Memory requirements of the structure could be fine-tuned as well.

REFERENCES

1. Lokeshwar B, Zaid MM, Naveen S, Venkatesh J, Sravya L. Analysis of Time and Space Complexity of Array, Linked List and Linked Array (hybrid) in Linear Search Operation. In 2022 IEEE International Conference on Data Science, Agents & Artificial Intelligence (ICDSAAI). 2022 Dec 8; 1: 1–6.
2. Sanders P, Mehlhorn K, Dietzfelbinger M, Dementiev R, Sanders P, Mehlhorn K, Dietzfelbinger M, Dementiev R. Representing Sequences by Arrays and Linked Lists. In: Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox. Cham: Springer; 2019; 81–116.
3. Wen F, Qin M, Gratz PV, Reddy AN. Hardware memory management for future mobile hybrid memory systems. IEEE Trans Computer-Aided Design Integr Circuits Syst. 2020 Oct 2; 39(11): 3627–37.
4. Jin H, Li Z, Liu H, Liao X, Zhang Y. Hotspot-aware hybrid memory management for in-memory key-value stores. IEEE Trans Parallel Distrib Syst. 2019 Oct 4; 31(4): 779–92.
5. Cook V, Peterson C, Painter Z, Dechev D. Quantifiability: Concurrent correctness from first principles. arXiv preprint arXiv:1905.06421. 2019 May 15.
6. Java T Point. (2021). Contiguous and Non-Contiguous Memory Allocation in Operating System. [Online]. Available from: https://www.javatpoint.com/contiguous-and-non-contiguous-memory-allocation-in-operating-system.
7. Rahama M. Data Structures and Algorithms. GRIN Verlag; 2013 Jan 7.

© STM Journals 2023. All Rights Reserved

8. Utkarsh Pandey. Time Complexity and Space Complexity. [Online]. Geeks for Geeks. Available from: https://www.geeksforgeeks.org/time-complexity-and-space-complexity/
9. Kumar S, Gaur MS, Kumar K, Sharma PS. A Novel Counting Sort for Real Numbers with Linear Time Complexity. In 2022 IEEE International Conference on Computational Intelligence and Sustainable Engineering Solutions (CISES). 2022 May 20; 60–64.
10. Susanne Windfeld Pedersen. (2022). Insert, Modify, Modify All, Delete, and Delete All Methods - Business Central. [Online]. Microsoft.com. Available from: https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-insert-modify-modifyall-delete-and-deleteall-methods

© STM Journals 2023. All Rights Reserved