OPEN ACCESS
Indo-JC
Indonesia Journal of Computing
ISSN 2460-9056
socj.telkomuniversity.ac.id/indojc

Ind. Journal on Computing
Vol. 5, Issue. 3, December 2020. pp. 47-62
doi:10.34818/indojc.2021.5.3.527

Hybrid Array List: An Efficient Dynamic Array with Linked List Structure

Mutaz Rasmi Abu Sara #1, Mohammad F. J. Klaib #2, Masud Hasan #3

Department of Computer Science, Taibah University, Madina Al Munawarah, Saudi Arabia
[1] mabusara@taibahu.edu.sa
[2] mklaib@taibahu.edu.sa
[3] hmasud@taibahu.edu.sa

Abstract

In this paper, a novel and efficient dynamic array, called *hybrid array list (HAL)*, has been presented. A HAL H has the structure of a linked list, but each node is an array of size at most $2c$, where c is an initial array size provided by the user. As elements are added or deleted in H, it grows or shrinks by the number of nodes as well as by the size of the arrays in the nodes. We consider the operations *append*, *insert* and *delete* as well as a helping operation *actual position* in H. These operations in worst case run in $O(1)$, $O(m+c)$, $O(m+c)$ and $O(m)$ times, respectively, where m is the number of nodes in H. Worst-case running time of similar operations in standard linked list or array are $O(n)$, where n is the total number of elements. As both m and c are no more than n, theoretically HAL is faster than array and linked list. Experimentally, HAL has been implemented and compared with similar operations in array list of Java and vector of C++. Our results show that HAL can perform substantially better when c is about half of the total number of elements.

Keywords: Dynamic array, linked list, data structure, hybrid array list

Abstrak

Kertas kerja ini akan membentangkan suatu tatasusunan yang baru dan dinamik, dinamakan *hybrid array list* (*HAL*). HAL H mempunyai struktur suatu senarai berantai, tetapi setiap nod, paling besar, ialah suatu tatasusunan bersaiz $2c$, yang mana c ialah saiz tatasusunan awal yang ditetapkan oleh pengguna. Apabila elemen ditambah atau dihapuskan dalam H, ia akan membesar atau menguncup mengikut bilangan nod dan juga mengikut saiz tatasusunan dalam nod tersebut. Kami mengambilkira operasi *append, insert* dan *delete* serta suatu operasi pembantu *actual position* dalam H. Pada jangkaan terburuk, operasi ini berlari $O(1)$, $O(m+c)$ dan $O(m)$ kali masing-masing, yang mana m adalah bilangan nod dalam H. Kes terburuk masa berlari operasi serupa dalam senarai berantai standad atau tatasusunan adalah $O(n)$, yang mana n ialah jumlah bilangan elemen. Oleh kerana kedua-dua m dan c adalah tidak lebih dari n, maka secara teori *HAL* adalah lebih laju dari tatasusunan dan senarai berantai. Sebagai eksperimen, kami telah melaksanakan *HAL* dan membandingkan operasi tersebut dengan operasi serupa dalam senarai tatasusunan JAVA dan vector C++. Keputusan kami adalah *HAL* menunjukkan prestasi jauh lebih baik apabila c lebih kurang separuh dari jumlah bilangan elemen-elemen.

Katakunci: Tatasusunan dinamik, senarai berantai, struktur data, senarai tatasusunan hibrid

Received on January, 2020. Accepted on December, 2020

I. INTRODUCTION

Array and linked list are two most basic and primitive data structures used in computer programming. Both of them contain a set of similar items. An array has fixed size and is defined by the user at the beginning of a program. The memory allocated for an array is contiguous in the physical memory of a computer. The elements are stored in sequence in the memory. Once defined, the size of an array cannot be changed. The most important advantage of an array is that its elements can be accessed randomly by their indices in constant time per element. See Fig. 1.

On the other hand, a linked list stores element in nodes---one node for each element. A node contains an element as well as a link to the next node (and another link to the previous node if the linked list is a doubly linked list). The memory for each node is allocated dynamically during the run time of a program. Therefore, the memory allocated by a linked list is not contiguous. This is an advantage in the sense that for a large number of elements a large size of contiguous memory is not required (unlike an array). Another advantage of a linked list is that the number of elements is not fixed. As long as there is enough memory, the elements can be added in a linked list. However, accessing elements in a linked list is slower than that in an array. In a linked list, an element must be accessed sequentially after moving from the start node to the destination node. In worst case, this time is linear to the number of elements in a linked list. See Fig. 1 for more detail.

Array and linked list also greatly differ in insertion and deletion of elements. In an array, insertion and deletion at specific position is expensive, as elements are to be shifted towards left or right respectively. This can take time linear to the number of elements in the array. In contrast, insertion and deletion are easier and faster in a linked list. A node can be inserted at any position in contestant time by manipulating some fixed number of links of the nodes involved. This is constant per insertion or deletion.

There are some other comparisons between array and linked list. Standard textbooks on algorithms and data structures contain chapters on array and linked list with discussion on their relative advantages and disadvantages. Such as, the textbooks [1-5] can be seen for this purpose.

A. Dynamic arrays

For a long time, researchers are trying to find some data structures that can capture the advantages of both array and linked list. In particular, the effort was made to find a data structure that can access the elements in a

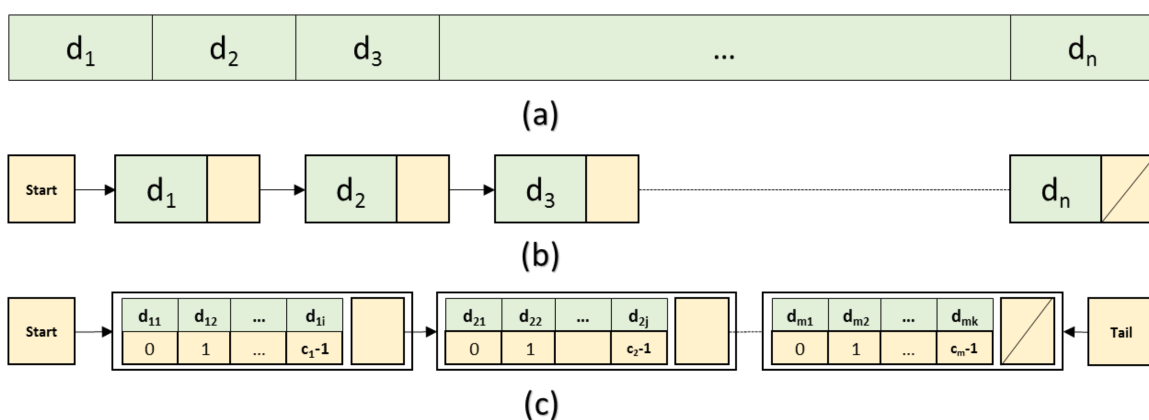

Fig. 1. (a) An array. (b) A linked list. (c) A hybrid array list (HAL).

faster way like an array, and at the same time can maintain variable number of elements like a linked list. The most successful innovation so far is the concept of dynamic array [1, 6-16, 18].

The basic principle of a dynamic array is to create an array of fixed size at the beginning. Then the elements are appended, inserted or deleted as like an array. Because of appends and insertions, at any time if the dynamic array becomes full, a new array is created with an increased size. The elements from the old array is copied to the new array along with the newly appended or inserted element. Then the old array is freed. The factor by which the size of the array is increased can vary. However, the usual value of this factor is 1.125 [14], 1.5 [11, 13, 15, 16] or 2 [1, 6, 12].

Similarly, if the dynamic array size becomes too small because of deletions, a new array is created with smaller size and the elements from the old array, excluding the deleted element, are copied to the new array. Then the old array is freed. Again, the factor by which the array size is reduced can vary like 0.5 [1] or 0.25 [1].

B. Variations of dynamic array

It may be difficult to find a non-trivial computer program that does not use an array. However, the use of dynamic array is also becoming more common as the programmers want to get the advantages of variable size of a dynamic array. This can be noticed from the practical point of view. Popular and widely used programming languages such as Java, C++ and Python implement dynamic arrays in their standard library functions. Dynamic array is implemented as *array list* in Java [11, 17], *vector* in C++ [12, 15] and *list* in Python [6, 14]. Their implementations are different based on the factor by which the size of the array is increased, which is 1.5 for Java arraylist [11] and C++ vector [15], and 1.125 for Python list [14]. Various popular software and platforms also use or support dynamic array in their codes, such as Android (because its applications are developed in Java), Facebook [16] and Emacs [18, 19].

Also, theoretically, there are several variations of dynamic arrays. The most basic version of a dynamic array can be found in the textbooks on algorithms and data structures, such as in Cormen et al. [1]. In Cormen et al. [1], the dynamic array is explained as an abstract data type (ADT), called *dynamic table*. The actual structure of a dynamic table can be an array, or an array-implemented stack, heap, queue or hash table. The table size is initially one. The elements are inserted and deleted into the table. When the table is full (due to the insertions), its size is doubled by allocating a new table, copying the elements from the old table to the new table, and finally freeing up the old table. Similarly, when the empty slots in the table (due to the deletions) are much high, its size is made half by allocating a new table, copying the elements from the old table to the new table, and finally freeing up the old table. The authors in [1] showed that the cost of an insertion or a deletion in a dynamic table take constant amortized time on average. (Please recall that an *amortized time* is the average time required to perform a sequence of operations over all the operations performed [Page 451 of 1, Page 34 of 5].)

Gap buffer [18] is a variation of dynamic array that works well when the insertion and deletion operations are performed close to each other in the array position. For example, in text editors, users usually perform insertion and deletion close to their current typing places. Therefore, gap buffers are useful in text editors [18, 19], such as in Emacs [18,19].

A hashed array tree (HAT) [9] is a dynamic array that maintains an array of pointers each pointing to a fixed size array. Altogether these arrays are like leaves of same length and the array of pointers are their parents. As the elements are stored in separated arrays of smaller size instead of one big array, HAT can reduce the amount of shifting for insertion and deletion as well as the amount of copying due to automatic resizing of the array when it is full.

Tiered vector is another variation of dynamic array presented by Goodrich et al. [8]. By their data structure, insertion and deletion can be performed in $O(n^{1/2})$ time, which is much less than $O(n)$ required by an array.

Brodnik et al. [7] described a variation of dynamic array, which they call resizable array, where the insertion and deletion operations can be performed in amortized as well as worst-case constant time.

C. Results in this paper

In this paper, yet another variation of dynamic array which is called *hybrid array list (HAL)* is presented. The structure of a HAL H is a linked list. Each node, also called a *chunk,* in H is an array of size at most $2c$, where c is an initial array size determined by the user. As the elements are added and deleted in H, it grows or shrinks both by the number of nodes in H as well as the sizes of the arrays in the nodes of H. See Fig. 1 for an illustration.

Four operations are considered in H, which are *actual position*, *append*, *insert* and *delete*. At any time, all chunks of H are full of elements (i.e., no empty space in their array), except the last chunk, for which there may be empty spaces at the end of the array. Therefore, H can be considered as a single array with all empty spaces at its end. An append operation adds an element to H at its end, which is the last available position in the last chunk. A new chunk may be created based on whether the current size of the last chunk is c. Insert and delete use an index that is counted from the beginning of H. An actual position operation finds the chunk that contains that index. An insert operation inserts an element in the relative position of the index in that chunk if the chunk is the last one in H. Otherwise, a new chunk is created with one size bigger and all elements as well as the new element in relative position are copied to the new chunk. The old chink is deleted. A delete operation works in a similar way.

The running time of these operations are $O(m)$ for actual position, $O(m+c)$ for insert and delete, and $O(1)$ for append, where m is the number of chunks in H. The value of m greatly depends upon the value of c. However, in general, both m and c are much smaller than n and never more than n, where n is the total number of elements in H. As in both array and linked list, similar operations run in worst case in $O(n)$ time, HAL is theoretically faster than linked list and array. (Please note that writing 'O' notation with more than one parameter, as we have written here with m and c, is not uncommon and is used for better accuracy in complexity analysis. For example, see [Sections 15.4 and 23.2 of 1] for similarly written running times for some well-known algorithms.)

Experimentally, HAL is implemented with the abovementioned operations and compared with similar operations in array list of Java and vector of C++. The experimental results show that H can perform substantially faster in practice when c is about half of n.

The value of c plays an important role in the efficiency of HAL. When c is high and close to n, the value of m is much small. In that case, a HAL works more like an array. On the other hand, when c is small compared to n, the value of m is much high. In that case, a HAL works more like a linked list. When the value of c is about half of n, a HAL uses the benefit of both an array and a linked list, and therefore, works more efficiently. This is evident from our experimental results too, which show that HAL performs much faster when c is about the half of n (also see Section III).

II. Hybrid Array List

We denote a chunk by C. We also use C specifically to denote the *current chunk* that we are working on. We denote by $|C|$ the number of elements in C. We use p as the index of the last element in the last chunk. These are the notations along with c (initial chunk size), m (number of chunks in H) and n (total number of elements in H) that we shall use most frequently in the rest of the paper.

A. HAL operations

a. Initialization

In the initialization phase, a new HAL *H* is created. It creates, by a *CreateNode* function, only one empty chunk *C* of size *c*. The current chunk is *C*. See Algorithm 1 for the pseudo code of the initialization phase in more detail.

All the steps for initializations, including the CreateNode function, run in constant time. Therefore, the following theorem is evident.

Theorem 1: An initialization of *H* takes *O*(1) time.

```
InitializeHAL(chunkSize):
Define a global counter totalNumberofElements ← 0
Define a global integer atPosition ← 0
Define a global variable startNode ← CreateNode(chunkSize)
Define a global variable tail ← startNode
Define a global variable current = start;
Define a global variable previous = null;

CreateNode (chunkSize):
Create a newNode that contains an array ARR[chunkSize]
nextNode ← NULL
```

Algorithm 1: Pseudo code for Initialization and CreateNode().

b. Actual position

An actual position operation, denoted as ActualPosition(*i*), works as follows. The index *i* is from the beginning of *H*. Let *C* be the chunk that contains this index *i*. Let *j* be the index of *C* where *i maps* to. Note that *j*<=*i*. If *C* is the only chunk in *H*, then *i* and *j* have the same value. If *H* has more than one chunk, then *j* is *i* minus the total number of elements in the chunks before *C*. ActualPosition(*i*) returns *j*. Please see Algorithm 2 for the pseudo code of ActualPosition(*i*).

```
ActualPosition(index)
Define previous ← NULL
Define counter ← current.ARR.length;
while (index >= counter and current.next != null)
        previous ← current
        current ← current.next
        counter ← counter + current.arr.length
Return actualPosition ← Position – (counter - current.ARR.length)
```

Algorithm 2: Pseudo code for ActualPosition(*i*).

The following theorem gives the running time of ActualPosition(*i*).

Theorem 2: ActualPosition(*i*) in *H* runs in *O*(*m*) time.

Proof: In worst case, *i* may map to the last chunk of *H*. Therefore, we need to traverse all *m* chunks of *H* by following the next links. This takes *O*(*m*) time.

c. Append

An *append* operation, denoted as Append(x), adds an element x in the first available position, which is p+1, in the last chunk. If the last chunk has c elements, then a new chunk C of size c is created at the end of H and x is inserted at the first position of C. Please see Fig. 2 and see Algorithm 3 for the pseudo code of Append(x). Note that append operations always keep the size of a chunk within c.

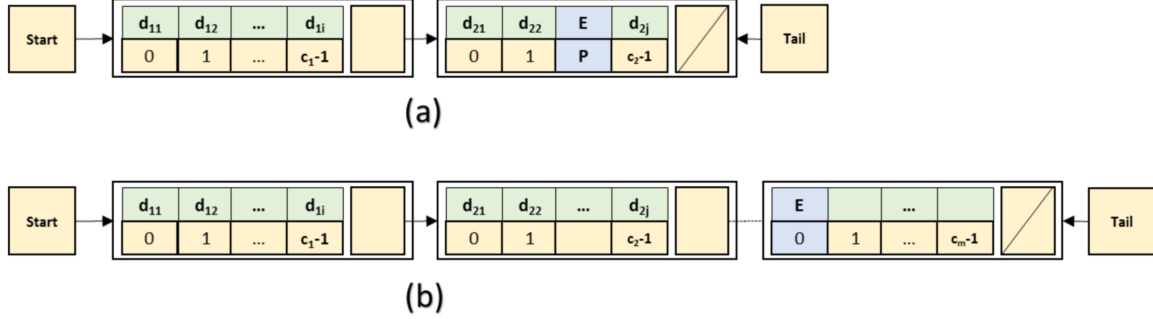

Fig. 2. Appending an element in H. (a) Last chunk has less than c elements. (b) Last chunk has c elements.

The following theorem gives the running time of Append(x).

Theorem 3: Append(x) in H can be done in $O(1)$ time.

Proof: All the steps in Append(x) run in $O(1)$ time. Therefore, the total number of time is $O(1)$.

```
Append(newDataElement):
if (atPosition >= chunkSize)
        atPosition ← 0
        tail.next ← CreateNode(chunkSize);
        tail ← tail.next
tail.ARR[atPosition] ← newDataElement;
atPosition ← atPosition + 1
totalNumberofElements ← totalNumberofElements + 1
```

Algorithm 3: Pseudo code for Append(x).

d. Insert

An insert operation, denoted as Insert(x, i), inserts an element x in position i in H. If i is more than the number of elements in H, then x is simply appended at the end of H by Append(x). This will ensure that the newly inserted element is contiguous with the exiting elements. If i is no more than the number of elements in H, then Insert(x, i) works in two steps. First it finds, by using ActualPosition(i), the chunk that contains the index i as well as the actual position of i in that chunk. Let that chunk be C and let j be the actual position of i in C.

In the next step, it inserts x in C at j. The insertion is easy if C is the last chunk. If C is the last chunk and there is an empty space in C, then x is inserted at j by shifting all elements after j by one position to the right. If C has no empty space, then a new chunk C' of size c is created at the end of H. The last element of C is copied

to the first position of C', and thus making the last space empty in C. The remaining elements after j in C are shifted by one position to the right and x is inserted at j. Finally, C' is made the tail node. Note that an insertion in the last chunk keeps its size c.

If C is not the last chunk, then there are two cases based on the value of $|C|$: (1) $|C| < 2c$ and (2) $|C| = 2c$. In Case (1), a new chunk C' is created with size $|C|+1$. All elements of C from position 0 to j-1 are copied to C' in positions 0 to j-1, then x is inserted at position j in C', and then all elements of C from position j to $|C|$-1 are copied to C' in positions j+1 to $|C|$. Finally, C is replaced by C' in H with necessary link updates and C is deleted.

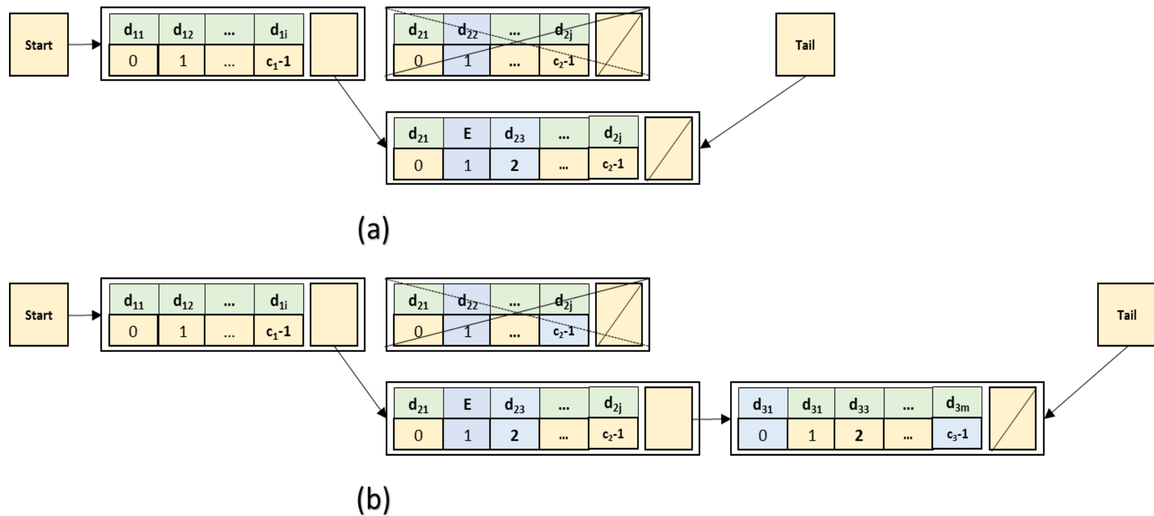

Fig. 3. Inserting an element x into H. The chunk in which x is to be inserted has (a) less than $2c$ elements and (b) $2c$ elements.

In Case (2), there can be further two cases: ($2a$) $j < c$ and ($2b$) $j >= c$. In both cases, two new chunks C' and C'' are created of size c each. In Case ($2a$), all elements of C from position 0 to j-1 are copied to C' in positions 0 to j-1, then x is inserted at position j in C', and then all elements of C from position j to c-2 are copied to C' in positions j+1 to c-1. All elements of C from position c-1 to $2c$-2 are copied to C'' in positions 0 to c. In Case (2b), all elements of C from position 0 to c-1 are copied to C' in positions 0 to c-1. All elements of C from position c to j-1 are copied to C'' in positions 0 to j-1, then x is inserted into C'' in position j, and all elements of C from position j to $2c$-2 are copied to C'' in positions j+1 to $2c$-1. Finally, C is replaced by C' and C'' in sequence in H with necessary link updates and C is deleted. Please see Fig. 3 and please see Algorithm 4 for the pseudo code of Insert(x, i). Note that, insert operations allow the size of a chunk to grow beyond c, but it keeps the size within $2c$.

The following theorem gives the running time of Insert(x, i).

Theorem 4: Insert(x, i) in H can be done in $O(m+c)$ time, where m is the number of chunks in H and c is the chunk size.

Proof: By Theorem 2, finding C and j takes $O(m)$ time. If C is the last chunk, then for inserting x in C, in worst case all elements of C may need to be shifted right by one position when the value of j may be one. Since C can have at most c elements, it takes $O(c)$ time. Moreover, a new chunk may need to be created when C is full, which would take $O(1)$ time. Therefore, the total time for inserting in the last chunk is $O(c) + O(1) = O(c)$.

If C is not the last chunk, then copying one by one all elements of C along with x to C' and possibly to C'' takes $O(|C|+1)$. Since C is split when $|C|$ is $2c$, we have $|C| <= 2c$. Therefore, time for copying is $O(2c+1) = O(c)$. Finally, deleting C from H and inserting C' and C'' in H require constant amount of pointer updates. Therefore, total time for Insert(x, i) is $O(m)+O(c) = O(m+c)$. (Please remember that writing this running time as $O(m+c)$ gives a better accuracy than writing it as $O(\max\{m, c\})$.)

```
Insert(newDataElement, Position):
if (Position >= totalNumberofElements)
        Append(newDataElement)
        exit
Define actualPosition ← ActualPosition(Position)
if (current = tail)
        if (atPosition <= tail.ARR.length - 1)
                Shift all elements starting from actualPosition to the right.
                tail.ARR [actualPosition] ← newDataElement;
                if(atPosition = tail.ARR.length - 1){
                        temp1 ← CreateNode(chunkSize)
                        tail.next ← temp1
                        tail ← temp1
                        atPosition ← tail.ARR[0]
                else
                        atPosition ← atPosition + 1

        else
                temp1 ← CreateNode(chunkSize)
                tail.next ← temp1
                temp1.ARR [0] ← tail. ARR[tail.ARR.length - 1];
                Shift all elements starting from actualPosition to the right, to insert the new element at
                the actualPosition.
                tail. ARR [actualPosition] ← newDataElement;
                tail ← temp1
                atPosition ← temp1.ARR[1]
else
        if (current.ARR.length = (chunkSize * 2) - 1)
                temp1 ← CreateNode(chunkSize)
                temp2 ← CreateNode(chunkSize)
                Copy elements from the current chunk to the new chunks while adding a new element to
                the appropriate chunk using actualPosition
                temp2.next ← current.next
                temp1.next ← temp2
                previous.next ← temp1
        else
                temp ← CreateNode(chunkSize + 1)
                Copy elements from the current chunk to the new chunk while adding a new element at
                actualPosition
                temp.next ← current.next
                previous.next ← temp

totalNumberofElements ← totalNumberofElements + 1
```

Algorithm 4: Pseudo code for Insert(x, i).

e. Delete

A *delete* operation, denoted as Delete(i), deletes the element that is in position i in H. Similar to an insertion, Delete(i) works in two steps. First it finds, by using ActualPosition(i), the chunk that contains the index i as well as finds the actual position of i in that chunk. Let that chunk be C and let j be the actual position of i in C. If C is the last chunk and there are other elements than x, then x is deleted by shifting all elements from position j to the last element by one position to the left. If x is the only element in C, then C is deleted.

If C is not the last chunk, then there are two cases: (1) $|C| = 1$ and (2) $|C| > 1$. In Case (1), C will be deleted with necessary pointer updates in H. In Case (2), a new chunk C' is created with size $|C|$-1. All elements of C from position 0 to j-1 are copied to C' in position 0 to j-1, and then all elements of C from position j+1 to $|C|$-1 are copied to C' in position j to $|C|$-2. Finally, C is replaced by C' in H with necessary link updates and C is deleted. Please see Fig. 4 and please see Algorithm 5 for the pseudo code of Delete(x).

Theorem 5: Delete(x) in H can be done in $O(m+c)$ time, where m is the number of chunks in H and c is the chunk size.

Proof: The analysis is similar to that for an insertion. By Theorem 2, finding C and j takes $O(m)$ time. If C is the last chunk, then for deleting x from C, in worst case all elements of C may need to be shifted left by one position when the value of j is one. As C can have at most c elements, it takes $O(c)$ time. If C needs to be deleted, then that can be done by updating some pointers in $O(1)$ time. Therefore, the total time for deleting from the last chunk is $O(c) + O(1) = O(c)$.

If C is not the last chunk, then copying one by one all elements of C along without x to C' takes $O(|C|)$. Since, C can have at most $2c$ elements, time for copying these elements is $O(2c) = O(c)$. Finally, deleting C from H and inserting C' in H require constant amount of pointer updates. Therefore, total time for Delete(x) is $O(m)+O(c) = O(m+c)$.

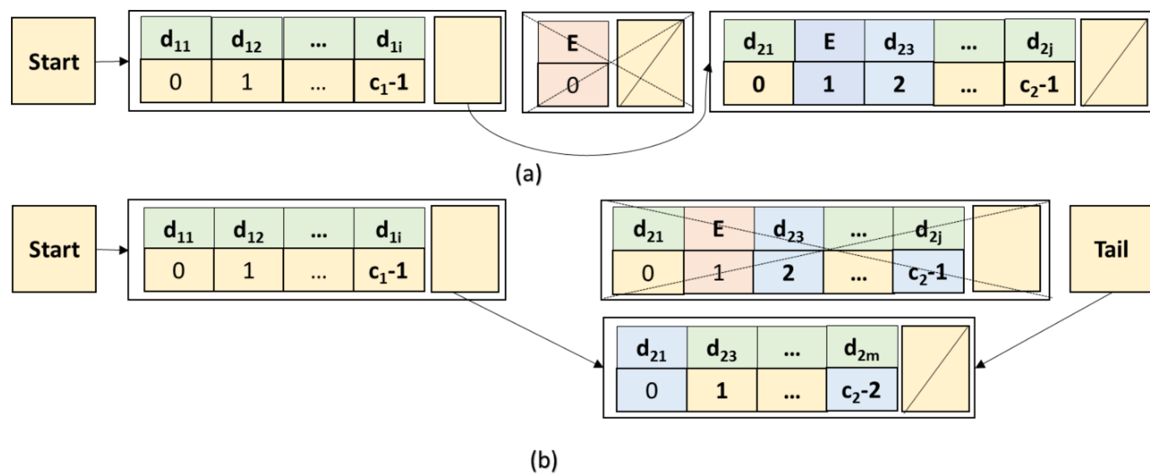

Fig. 4. Deleting an element x from H. The chunk from which x is to be deleted has (a) 1 element and (b) more than 1 element.

```
DeleteElement (Position)
Define actualPosition ←ActualPosition(Position)
if (current = tail) {
        if (tail.ARR.length = 1)
                temp ← CreateNode(chunkSize)
                if(startNode == tail)
                        startNode ← temp
                tail ← temp
                atPosition ← tail.ARR[0]


        else
                Shift all elements starting from actualPosition + 1 to the left.
                atPosition = atPosition - 1;

else
        if (current.ARR.length = 1)
                if (current = startNode)
                        startNode ← NULL
                        tail ← NULL
                else
                        previous.next ← current.next
        else
                temp ← CreateNode (current.ARR.length - 1)
                temp.next ← current.next
                Copy elements from the current chunk to the new chunk without the element at
                actualPosition
                previous.next ← temp

totalNumberofElements ← totalNumberofElements - 1
```

Algorithm 5: Pseudo code for Delete(x).

III. Experimental Results

HAL has been implemented in Java with the operations append, insert, and delete for H. Experiments have been done with these operations in different input settings and these operations have been compared with similar library functions in Java and C++.

A. Only append

In the first input setting, the experiment performs the function Append(x) only. Initially, H is empty. The elements are added in H by Append(x). The chunk size is varied from 1 to 10^8. The number of elements appended are 10^5 and 10^7. For chunk size 10, the experiment has been done in more refined form by varying the number of append from 10^3 to 10^8. Same elements have been inserted into an arraylist of Java by its *append(data)* function as well as into a vector of C++ by its *push_back(data)* function. Then comparisons have been made between the total running times taken by these appends required by HAL, Java array list and C++ vector. The result is shown in Table 1.

From this table it can be seen that in all cases, HAL substantially outperforms both Java arraylist and C++ vector. When the chunk size or the number of append become very high like 10^7, HAL gives an out of memory error. Java arraylist gives the similar error for these cases as well as other cases with lower chunk size and lower number of appends.

TABLE 1
EXPERIMENTAL RESULTS FOR THE FIRST INPUT SETTING WITH ONLY APPEND OPERATIONS.

Original Size (Chunk size)	Number of Additions	Time (ms)			Speed up Over	
		ArrayList (java)	Vector (C++)	HAL	ArrayList (java) (%)	Vector (C++) (%)
10	10^3	1	1	1	0.00%	0.00%
	10^4	2	10	1	50.00%	90.00%
	10^5	7	100	3	57.14%	97.00%
	10^6	159	993	12	92.45%	98.79%
	10^7	3145	9373	424	86.52%	95.48%
	10^8	OutOfMemoryError	91286	OutOfMemoryError		
10^0	10^5	7	118	1	85.71%	99.15%
10^1		6	107	3	50.00%	97.20%
10^2		5	101	3	40.00%	97.03%
10^3		6	102	2	66.67%	98.04%
10^4		6	110	3	50.00%	97.27%
10^5		5	101	2	60.00%	98.02%
10^6		9	103	9	0.00%	91.26%
10^7		19	102	13	31.58%	87.25%
10^8		OutOfMemoryError	108	OutOfMemoryError		
10^0	10^7	2320	9276	469	79.78%	94.94%
10^1		1913	9262	101	94.72%	98.91%
10^2		1693	9071	103	93.92%	98.86%
10^3		1716	9245	98	94.29%	98.94%
10^4		OutOfMemoryError	9084	107		98.82%
10^5		OutOfMemoryError	9279	105		98.87%
10^6		OutOfMemoryError	9132	109		98.81%
10^7		OutOfMemoryError	9326	95		98.98%
10^8		OutOfMemoryError	9043	OutOfMemoryError		

B. *Only insert*

In the next input setting, experiments have been done for the function Insert(x, i) only. Before applying the insert operations, H contains 10^5 elements, which were added by append functions. Then elements are inserted into H. Insertions are made in random positions (i.e., with random values of i). The chunk size is varied from 1 to 10^5. The number of elements inserted are from 1 to 10^5. Same experiments have been done with Java arraylist by its *insertElement(data, index)* function as well as with C++ vector by its *insert(index, data)* function. Then comparisons have been made for the total running time taken by these insertions in HAL, Java arraylist and C++ vector. The result is shown in Table 2. From this table it can be seen that when the chunk size is 10^2, 10^3 and 10^4, HAL substantially outperforms both Java arraylist and C++ vector.

C. *Only delete*

In the next input setting, experiments have been done for the function Delete(x, i) only. It has been done in a way similar to insert. Before applying the delete operations, H contains 10^5 elements, which were added by append functions. Then elements are deleted from H. Deletions are done in random positions (i.e., with random values of i). The chunk size is varied from 1 to 10^5. The number of elements deleted are from 1 to 10^5.

Table 2
Experimental results for the second input setting with only insert operations.

Original Size (Chunk size)	Number of Elements already available in HAL by append	Number of Insertion (Randomly)	Time (ms)			Speed up Over	
			ArrayList (java)	Vector (C++)	HAL	ArrayList (java) (%)	Vector (C++) (%)
10^0	10^5	10^0	0	0	0		
		10^1	0	0	16		
		10^2	17	3	31	-82.35%	-933.33%
		10^3	31	30	852	-2648.39%	-2740.00%
		10^4	267	301	1955	-632.21%	-549.50%
		10^5	2852	4207	63976	-2143.20%	-1420.70%
10	10^5	10^0	0	0	0		
		10^1	0	0	0		
		10^2	0	3	0		100.00%
		10^3	38	27	31	18.42%	-14.81%
		10^4	318	308	416	-30.82%	-35.06%
		10^5	3895	4498	10114	-159.67%	-124.86%
10^2	10^5	10^0	0	0	0		
		10^1	0	1	0		100.00%
		10^2	3	3	2	33.33%	33.33%
		10^3	37	28	7	81.08%	75.00%
		10^4	312	301	78	75.00%	74.09%
		10^5	3465	4176	858	75.24%	79.45%
10^3	10^5	10^0	0	0	0		
		10^1	0	0	0		
		10^2	15	3	4	73.33%	-33.33%
		10^3	32	29	6	81.25%	79.31%
		10^4	779	300	497	36.20%	-65.67%
		10^5	3195	4187	882	72.39%	78.93%
10^4	10^5	10^0	0	0	0		
		10^1	0	0	0		
		10^2	0	3	0		100.00%
		10^3	31	29	32	-3.23%	-10.34%
		10^4	615	312	677	-10.08%	-116.99%
		10^5	3603	4182	3117	13.49%	25.47%
10^5	10^5	10^0	0	0	3		
		10^1	0	0	17		
		10^2	0	3	47		
		10^3	31	29	808	-2506.45%	-2686.21%
		10^4	863	298	2129	-146.70%	-614.43%
		10^5	3479	4320	30213	-768.44%	-599.38%

Same experiments have been done with Java arraylist by its *delete(index)* function as well as with C++ vector by its *erase(index)* function. Then comparisons have been made for the total running time taken by these deletions required in HAL, Java arraylist and C++ vector. The result is shown in Table 3. From this table it can be seen that when the chunk size is 10^2, 10^3 and 10^4, HAL substantially outperforms both Java arraylist and C++ vector.

TABLE 3
EXPERIMENTAL RESULTS FOR THE THIRD INPUT SETTING WITH ONLY DELETE OPERATIONS.

Original Size (Chunk size)	Number of Elements (already available in HAL by append)	Number of Deletion (Randomly)	Time (ms)			Speed up Over	
			ArrayList (java)	Vector (C++)	HAL	ArrayList (java) (%)	Vector (C++) (%)
10^0	10^5	10^0	0	0	0		
		10^1	0	0	0		
		10^2	0	5	15		-200.00%
		10^3	31	55	468	-1409.68%	-750.91%
		10^4	608	467	1492	-145.39%	-219.49%
		10^5	811	2127	10051	-1139.33%	-372.54%
10	10^5	10^0	0	0	0		
		10^1	0	1	0		100.00%
		10^2	0	5	0		100.00%
		10^3	31	48	31	0.00%	35.42%
		10^4	577	458	967	-67.59%	-111.14%
		10^5	1248	2133	6644	-432.37%	-211.49%
10^2	10^5	10^0	0	0	0		
		10^1	0	1	0		100.00%
		10^2	0	5	0		100.00%
		10^3	32	48	0	100.00%	100.00%
		10^4	561	458	32	94.30%	93.01%
		10^5	1373	2140	329	76.04%	84.63%
10^3	10^5	10^0	0	0	0		
		10^1	0	0	0		
		10^2	0	4	0		100.00%
		10^3	31	42	0	100.00%	100.00%
		10^4	702	456	15	97.86%	96.71%
		10^5	1373	2114	125	90.90%	94.09%
10^4	10^5	10^0	0	0	0		
		10^1	0	1	0		100.00%
		10^2	31	5	0	100.00%	100.00%
		10^3	468	48	31	93.38%	35.42%
		10^4	640	451	203	68.28%	54.99%
		10^5	1170	9179	1028	12.14%	88.80%
10^5	10^5	10^0	0	0	0		
		10^1	0	0	15		
		10^2	0	5	16		-220.00%
		10^3	31	49	218	-603.23%	-344.90%
		10^4	799	464	2108	-163.83%	-354.31%
		10^5	1170	2126	10221	-773.59%	-380.76%

D. Random, append, insert and delete

In the next two settings, experiments have been done for randomly applying the three functions Append(x), Insert(x, i) and Delete(x, i). In the first of these two settings, H has 10^5 elements, which were added by append functions. Then elements are randomly appended, inserted and deleted from H. Insertion and deletions are done

in random positions. The chunk size is varied from 1 to 10^5. The number of operations performed are from 1 to 10^5.

Table 4
Experimental results for appending, insertion and deletion applied randomly on non-empty H.

Original Size (Chunk size)	Number of Elements already available in HAL by append	Number of Append, Insertion & Delete (Randomly)	Time (ms)			Speed up Over	
			ArrayList (java)	**Vector (C++)**	**HAL**	**ArrayList (java) (%)**	**Vector (C++) (%)**
10^0	10^5	10^0	0	0	2	0	
		10^1	0	0	7	0	
		10^2	2	3	8	-300.00%	-166.67%
		10^3	25	24	113	-352.00%	-370.83%
		10^4	651	257	1046	-60.68%	-307.00%
		10^5	1474	2794	18496	-1154.82%	-561.99%
10	10^5	10^0	0	0	1		
		10^1	1	0	2	-100.00%	
		10^2	3	2	4	-33.33%	-100.00%
		10^3	26	25	15	42.31%	40.00%
		10^4	706	260	354	49.86%	-36.15%
		10^5	1527	2791	6844	-348.20%	-145.22%
10^2	10^5	10^0	0	0	0		
		10^1	0	1	0		100.00%
		10^2	3	2	2	33.33%	0.00%
		10^3	15	25	7	53.33%	72.00%
		10^4	255	257	47	81.57%	81.71%
		10^5	1998	2837	272	86.39%	90.41%
10^3	10^5	10^0	0	0	0		
		10^1	0	1	0		100.00%
		10^2	3	2	3	0.00%	-50.00%
		10^3	32	25	7	78.13%	72.00%
		10^4	644	258	33	94.88%	87.21%
		10^5	2111	2800	175	91.71%	93.75%
10^4	10^5	10^0	0	0	0		
		10^1	0	1	0		100.00%
		10^2	2	3	7	-250.00%	-133.33%
		10^3	47	24	32	31.91%	-33.33%
		10^4	202	256	166	17.82%	35.16%
		10^5	1838	2810	1349	26.61%	51.99%
10^5	10^5	10^0	0	0	0		
		10^1	0	1	9		-800.00%
		10^2	0	3	36		-1100.00%
		10^3	28	28	140	-400.00%	-400.00%
		10^4	847	262	1304	-53.96%	-397.71%
		10^5	1856	2792	13767	-641.76%	-393.09%

Same experiments have been done with corresponding append, insert or delete functions in Java arraylist by its *append (data), insertElement (data, index),* and *delete(index)* functions. It has been done similarly in C++ vector by its *push_back(data), insert(index, data),* and *erase(index)* functions. Then comparison have been

made for the total running time taken by these operations in HAL, Java arraylist and C++ vector. The result is shown in Table 4. From this table it can be seen that when the chunk size is 10^2, 10^3 and 10^4, HAL substantially outperforms both Java arraylist and C++ vector.

In the second of these two settings, the same experiment has been done, but *H* remains empty at the beginning. The result is shown in Table 5. Once again, from this table it can be seen that when the chunk size is 10^2, 10^3 and 10^4, HAL substantially outperforms both Java arraylist and C++ vector.

TABLE 5
EXPERIMENTAL RESULTS FOR APPENDING, INSERTION AND DELETION APPLIED RANDOMLY ON EMPTY H.

Original Size (Chunk size)	**Number of Elements, intially HAL is empty**	**Number of Append, Insertion & Delete (Randomly)**	**Time (ms)**			**Speed up Over**	
			ArrayList (java)	**Vector (C++)**	**HAL**	**ArrayList (java) (%)**	**Vector (C++) (%)**
10^0	0	10^1	0	0	0		
		10^2	1	0	0	100.00%	
		10^3	2	5	0	100.00%	100.00%
		10^4	9	40	31	-244.44%	22.50%
		10^5	178	577	7456	-4088.76%	-1192.2%
10	0	10^1	0	0	0		
		10^2	0	0	0		
		10^3	1	4	0	100.00%	100.00%
		10^4	7	45	0	100.00%	100.00%
		10^5	181	602	390	-115.47%	35.22%
10^2	0	10^1	0	0	0		
		10^2	0	0	0		
		10^3	1	4	0	100.00%	100.00%
		10^4	7	39	0	100.00%	100.00%
		10^5	179	583	46	74.30%	92.11%
10^3	0	10^1	0	0	0		
		10^2	0	0	0		
		10^3	2	4	0	100.00%	100.00%
		10^4	7	40	16	-128.57%	60.00%
		10^5	191	571	140	26.70%	75.48%
10^4	0	10^1	0	0	0		
		10^2	0	0	15		
		10^3	1	4	16	-1500.00%	-300.00%
		10^4	7	43	140	-1900.00%	-225.58%
		10^5	194	592	1297	-568.56%	-119.09%
10^5	0	10^1	0	2	15		-650.00%
		10^2	0	2	31		-1450.0%
		10^3	2	4	125	-6150.00%	-3025.0%
		10^4	7	43	1248	-17728.5%	-2802.3%
		10^5	188	595	12152	-6363.83%	-1942.3%

IV. CONCLUSION

In this paper, a new and efficient dynamic array, called hybrid dynamic array (HAL), has been presented. HAL has been presented with some basic operations, such as append, insert, delete. The running time of these operations are within $O(1)$, $O(m+c)$ and $O(m+c)$, respectively, where m is the number of chunks in a HAL and c is a user defined parameter. As m and c can be smaller than n, these running times are much faster than $O(n)$, which is taken by some similar operations in array and linked list. HAL has also been implemented in Java and have been compared with Java and C++ by their similar dynamic array library functions. The experimental results show that HAL can perform faster than Java and C++ when c is about half of n. Dynamic arrays are in high use in popular programming languages such as in Java, C++ and Python. They are also in use in the popular software and platforms such as in Android, Facebook and Emacs. Therefore, we believe that HAL can be a good choice by the programmers and software developers. In future, we would like to revise HAL so that it performs better for higher or lower values of c as well.

REFERENCES

[1] Coreman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009) Introduction to Algorithms. The MIT Press, Cambridge.
[2] Goodrich, M. T. and Tamassia, R. (2014) Algorithm Design and Applications. Wiley, U.S.
[3] Kleinberg, J. and Tardos, E. (2005) Algorithm Design. Pearson, U.S.
[4] Weiss, M. A. (2011) Discrete Structures and Algorithms Analysis in java. Pearson, U.S.
[5] Goodrich, M. T. and Tamassia, R. (2001) Algorithm Design: Foundations, Analysis, and Internet Examples, Wiley, U. S.
[6] Lambert, K. A. (2009) Fundamentals of Python: From First Programs through Data Structures. Course Technology, Boston.
[7] Brodnik, A., Carlsson, S., Demaine, E. D., Munro, I. J., and Sedgewick, R. (1999), Resizable Arrays in Optimal Time and Space. Proceedings of the Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, Vancouver, BC, Canada, 11- 14 August, pp. 37-48, Springer, Berlin, Heidelberg.
[8] Goodrich, M. and Kloss, J. (1999) Tiered Vectors: Efficient Dynamic Arrays for Rank-Based Sequences, Proceedings of the Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, Vol. 1663, Vancouver, BC, Canada, 11-14 August, pp 205-216, Springer, Berlin, Heidelberg
[9] Sitarski, E. (1996), HATs: Hashed Array Trees., Dr. Dobb's Journal, 21, 11.
[10] IC_TECH_REPORT_200244 (2002) Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays, EPFL, Geneva, Switzerland.
[11] Oracle, The source code of java.util.ArrayList class from OpenJDK 6, http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/e0e25ac28560/src/share/classes/java/util/ArrayList.java, last accessed: 16 April 2019.
[12] Sergei Danielian, C++ STL vector: definition, growth factor, member functions, https://web.archive.org/web/20150806162750/http:/www.gahcep.com/cpp-internals-stl-vector-part-1/, last accessed: 16 April 2019.
[13] Google Groups, vector growth factor of 1.5. comp.lang.c++.moderated, https://groups.google.com/forum/#!topic/comp.lang.c++.moderated/asH_VojWKJw%5B1-25%5D, last accessed: 16 April 2019
[14] python.org, List object implementation from python.org, http://svn.python.org/projects/python/trunk/Objects/listobject.c, last accessed: 16 April 2019.
[15] Brais, Hadi, Dissecting the C++ STL Vector: Part 3 - Capacity & Size, https://hadibrais.wordpress.com/2013/11/15/dissecting-the-c-stl-vector-part-3-capacity/, last accessed: 16 April 2019.
[16] GitHub, Inc. facebook/folly, https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md, last accessed: 16 April 2019.
[17] Oracle, Javadoc on ArrayList, https://docs.oracle.com/javase/10/docs/api/java/util/ArrayList.html, last accessed: 16 April 2019.
[18] Mark C. Chu-Carroll, Gap Buffers, or, Don't Get Tied Up With Ropes?, https://scienceblogs.com/goodmath/2009/02/18/gap-buffers-or-why-bother-with-1, last accessed: 16 April 2019.
[19] The Buffer Gap, https://www.gnu.org/software/emacs/manual/html_node/elisp/Buffer-Gap.html, last accessed: 16 April 2019.