A Programmer's approach of looking at Array vs. Linked List

In general, array is considered a data structure for which size is fixed at the compile time and array memory is allocated either from Data section (e.g. global array) or Stack section (e.g. local array). Similarly, linked list is considered a data structure for which size is not fixed and memory is allocated from Heap section (e.g. using malloc() etc.) as and when needed. In this sense, array is taken as a static data structure (residing in Data or Stack section) while linked list is taken as a dynamic data structure (residing in Heap section). Memory representation of array and linked list can be visualized as follows:

An array of 4 elements (integer type) which have been initialized with 1, 2, 3 and 4. Suppose, these elements are allocated at memory addresses 0x100, 0x104, 0x08 and 0x10B respectively.

```
[(1)]       [(2)]       [(3)]       [(4)]
0x100       0x104       0x108       0x10B
```

A linked list with 4 nodes where each node has integer as data and these data are initialized with 1, 2, 3 and 4. Suppose, these nodes are allocated via malloc() and memory allocated for them is 0x200, 0x308, 0x404 and 0x20B respectively.

```
[(1), 0x308]     [(2),0x404]      [(3),0x20B]       [(4),NULL]
  0x200            0x308            0x404             0x20B
```

Anyone with even little understanding of array and linked-list might not be interested in the above explanation. I mean, it is well know that the array elements are allocated memory in sequence i.e. contiguous memory while nodes of a linked list are non-contiguous in memory. Though it sounds trivial yet this the most important difference between array and linked list. It should be noted that due to this contiguous versus non-contiguous memory, array and linked list are different. In fact, this difference is what makes array vs. linked list! In the following sections, we will try to explore on this very idea further.

Since elements of array are contiguous in memory, we can access any element randomly using index e.g. intArr[3] will access directly fourth element of the array. (For newbies, array indexing start from 0 and that's why fourth element is indexed with 3). Also, due to contiguous memory for successive elements in array, no extra information is needed to be stored in individual elements i.e. no overhead of metadata in arrays. Contrary to this, linked list nodes are non-contiguous in memory. It means that we need some mechanism to traverse or access linked list nodes. To achieve this, each node stores the location of next node and this forms the basis of the link from one node to next node. Therefore, it's called Linked list. Though storing the location of next node is

overhead in linked list but it's required. Typically, we see linked list node declaration as follows:

```
struct llNode
{
  int dataInt;

  /* nextNode is the pointer to next node in linked list*/
  struct llNode * nextNode;
};
```

Run on IDE

So array elements are contiguous in memory and therefore not requiring any metadata. And linked list nodes are non-contiguous in memory thereby requiring metadata in the form of location of next node. Apart from this difference, we can see that array could have several unused elements because memory has already been allocated. But linked list will have only the required no. of data items. All the above information about array and linked list has been mentioned in several textbooks though in different ways.

What if we need to allocate array memory from Heap section (i.e. at run time) and linked list memory from Data/Stack section. First of all, is it possible? Before that, one might ask why would someone need to do this? Now, I hope that the remaining article would make you rethink about the idea of array vs. linked-list 🙂

Now consider the case when we need to store certain data in array (because array has the property of random access due to contiguous memory) but we don't know the total size apriori. One possibility is to allocate memory of this array from Heap at run time. For example, as follows:

/*At run-time, suppose we know the required size for integer array (e.g. input size from user). Say, the array size is stored in variable arrSize. Allocate this array from Heap as follows*/

```
int * dynArr = (int *)malloc(sizeof(int)*arrSize);
```

Run on IDE

Though the memory of this array is allocated from Heap, the elements can still be accessed via index mechanism e.g. dynArr[i]. Basically, based on the programming problem, we have combined one benefit of array (i.e. random access of elements) and one benefit of linked list (i.e. delaying the memory allocation till run time and allocating memory from Heap). Another advantage of having this type of dynamic array is that, this method of allocating array from Heap at run time could reduce code-size (of course, it depends on certain other factors e.g. program format etc.)

Now consider the case when we need to store data in a linked list (because no. of nodes in linked list would be equal to actual data items stored i.e. no extra space like array) but we aren't allowed to get this memory from Heap again and again for each node. This might look hypothetical situation to some folks but it's not very uncommon requirement in embedded systems. Basically, in several embedded programs, allocating memory via malloc() etc. isn't allowed due to multiple reasons. One obvious reason is performance i.e. allocating memory via malloc() is costly in terms of time complexity because your embedded program is required to be deterministic most of the times. Another reason could be module specific memory management i.e. it's possible that each module in embedded system manages its own memory. In short, if we need to perform our own memory management, instead of relying on system provided APIs of malloc() and free(), we might choose the

linked list which is simulated using array. I hope that you got some idea why we might need to simulate linked list using array. Now, let us first see how this can be done. Suppose, type of a node in linked list (i.e. underlying array) is declared as follows:

```
struct sllNode
{
  int dataInt;

 /*Here, note that nextIndex stores the location of next node in
  linked list*/
  int nextIndex;
};

struct sllNode arrayLL[5];
```

Run on IDE

If we initialize this linked list (which is actually an array), it would look as follows in memory:

```
[(0),-1]    [(0),-1]    [(0),-1]   [(0),-1]   [(0),-1]
0x500        0x508       0x510      0x518      0x520
```

The important thing to notice is that all the nodes of the linked list are contiguous in memory (each one occupying 8 bytes) and nextIndex of each node is set to -1. This (i.e. -1) is done to denote that the each node of the linked list is empty as of now. This linked list is denoted by head index 0.

Now, if this linked list is updated with four elements of data part 4, 3, 2 and 1 successively, it would look as follows in memory. This linked list can be viewed as 0x500 -> 0x508 -> 0x510 -> 0x518.

```
[(1),1]       [(2),2]      [(3),3]     [(4),-2]     [(0),-1]
 0x500         0x508        0x510       0x518        0x520
```

The important thing to notice is nextIndex of last node (i.e. fourth node) is set to -2. This (i.e. -2) is done to denote the end of linked list. Also, head node of the linked list is index 0. This concept of simulating linked list using array would look more interesting if we delete say second node from the above linked list. In that case, the linked list will look as follows in memory:

```
[(1),2]       [(0),-1]      [(3),3]     [(4),-2]     [(0),-1]
 0x500         0x508         0x510       0x518        0x520
```

The resultant linked list is 0x500 -> 0x510 -> 0x518. Here, it should be noted that even though we have deleted second node from our linked list, the memory for this node is still there because underlying array is still there. But the nextIndex of first node now points to third node (for which index is 2).

Hopefully, the above examples would have given some idea that for the simulated linked list, we need to write our own API similar to malloc() and free() which would basically be used to insert and delete a node. Now this is what's called own memory management. Let us see how this can be done in algorithmic manner.

There are multiple ways to do so. If we take the simplistic approach of creating linked list using array, we can use the following logic. For inserting a node, traverse the underlying array and find a node whose nextIndex is -1. It means that this node is empty. Use this node as a new node. Update the data part in this new node and set the nextIndex of this node to current head node (i.e. head index) of the linked list. Finally, make the index of

this new node as head index of the linked list. To visualize it, let us take an example. Suppose the linked list is as follows where head Index is 0 i.e. linked list is 0x500 -> 0x508 -> 0x518 -> 0x520

```
[(1),1]       [(2),3]      [(0),-1]     [(4),4]     [(5),-2]
 0x500         0x508        0x510        0x518       0x520
```

After inserting a new node with data 8, the linked list would look as follows with head index as 2.

```
[(1),1]       [(2),3]      [(8),0]     [(4),4]     [(5),-2]
 0x500         0x508        0x510       0x518       0x520
```

So the linked list nodes would be at addresses 0x510 -> 0x500 -> 0x508 -> 0x518 -> 0x520

For deleting a node, we need to set the nextIndex of the node as -1 so that the node is marked as empty node. But, before doing so, we need to make sure that the nextIndex of the previous node is updated correctly to index of next node of this node to be deleted. We can see that we have done own memory management for creating a linked list out of the array memory. But, this is one way of inserting and deleting nodes in this linked list. It can be easily noticed that finding an empty node is not so efficient in terms of time complexity. Basically, we're searching the complete array linearly to find an empty node.

Let us see if we can optimize it further. Basically we can maintain a linked list of empty nodes as well in the same array. In that case, the linked list would be denoted by two indexes – one index would be for linked list which has the actual data values i.e. nodes which have been inserted so far and other index would for linked list of empty nodes. By doing so, whenever, we need to insert a new node in existing linked list, we can quickly find an empty node. Let us take an example:

```
[(4),2]    [(0),3]    [(5),5]    [(0),-1]   [(0),1]   [(9),-1]
 0x500      0x508      0x510      0x518      0x520      0x528
```

The above linked list which is represented using two indexes (0 and 5) has two linked lists: one for actual values and another for empty nodes. The linked list with actual values has nodes at address 0x500 -> 0x510 -> 0x528 while the linked list with empty nodes has nodes at addresses 0x520 -> 0x508 -> 0x518. It can be seen that finding an empty node (i.e. writing own API similar to malloc()) should be relatively faster now because we can quickly find a free node. In real world embedded programs, a fixed chunk of memory (normally called memory pool) is allocated using malloc() only once by a module. And then the management of this memory pool (which is basically an array) is done by that module itself using techniques mentioned earlier. Sometimes, there are multiple memory pools each one having different size of node. Of course, there are several other aspects of own memory management but we'll leave it here itself. But it's worth mentioning that there are several methods by which the insertion (which requires our own memory allocation) and deletion (which requires our own memory freeing) can be improved further.

If we look carefully, it can be noticed that the Heap section of memory is basically a big array of bytes which is being managed by the underlying operating system (OS). And OS is providing this memory management service to programmers via malloc(), free() etc. Aha !!

The important take-aways from this article can be summed as follows:

A) Array means contiguous memory. It can exist in any memory section be it Data or Stack or Heap.
B) Linked List means non-contiguous linked memory. It can exist in any memory section be it Heap or Data or

Stack.

C) As a programmer, looking at a data structure from memory perspective could provide us better insight in choosing a particular data structure or even designing a new data structure. For example, we might create an array of linked lists etc.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Category: Data Structures

6 • Reply • Share ›

Ekta Goel • 9 months ago

@Geeksforgeeks
Suppose, these elements are allocated at memory addresses 0x100, 0x104, 0x08 and 0x10B respectively.Third value is 0x108

4 • Reply • Share ›

IIT11 ➚ Ekta Goel • 6 months ago

Yes i also think it should be 0x10C

1 • Reply • Share ›

Ansuraj Khadanga ➚ Ekta Goel • 6 months ago

Yes third value must be 0x108 and fourth value must be 0x10C

• Reply • Share ›

shruti • 5 months ago

Shoudnot in the statement "the above linked list which is represented using two indexes (0 and 5)" , instead of index 5 as the head of the empty linked list it should be index 4, since index 5 is the last node of the linked list containing actual values.

3 • Reply • Share ›

Chirag • 8 months ago

In Linked List using data section, how insertion is performed? Insert @ begin?(as it insert data 4,3,2,1(i.e in reverse order)).
n how do we access linked list, using address of head or through index?
please explain this giving example.

2 • Reply • Share ›

Adauta Garcia Ariel • 7 months ago

I liked this approach it's like "Thinking outside of the box".

1 • Reply • Share ›

Harmanpreet Singh • 12 days ago

i can't understand the concept of maintaining linked list of empty nodes and data nodes . Furthermore i see the example given above uses more than two indexes not just 0 and 5. Kindly please explain.

• Reply • Share ›

Shweta Charchita • 12 days ago

Wouldn't implementing linked lists via array imply fixed size allocation?
Cz its :
struct sllNode array[5]; (that is a fixed size array declaration to create a linked list)

Very nice article though :) :)

• Reply • Share ›

Nishu Sharma ➚ Shweta Charchita • 12 days ago

NO..struct sllNode array[5]; implies that we would be making 5 of such linked lists and each list can contain any number of nodes(which are to be specified dynamically) and each node in every linked list contains two fields : first is the data field and the other one is the address field(as per the above example which u've specified).
Hope this helps :)

• Reply • Share ›

Rohit Saluja • 2 months ago

It is mentioned above that the head of the linked list is the node which has the value 0 as of the next_node but
Now, if this linked list is updated with four elements of data part 4, 3, 2 and 1 successively, it would look as follows in memory. This linked list can be viewed as 0x500 -> 0x508 -> 0x510 -> 0x518.
[(1),1] [(2),2] [(3),3] [(4),-2] [(0),-1]
0x500 0x508 0x510 0x518 0x520
But in this above example there is no such node with the value as 0 of the next_node. So why the head location is 0X500 ?

• Reply • Share ›

jitin maherchandani • 2 months ago

Add next and previous buttons for navigation to the articles

• Reply • Share ›

Mahendra Chhimwal • 4 months ago

Yupp I am with Anmol. Please provide a next and previous tutorial's link for better navigation through tutorials.

• Reply • Share ›

Zac • 4 months ago

(each one occupying 8 bytes)

[(0),-1] [(0),-1] [(0),-1] [(0),-1] [(0),-1]
0x500 0x508 0x510 0x518 0x520
but addressing is wrong for third node. It should be 0x516 ?

• Reply • Share ›

Ishan Gupta ➚ Zac • 2 months ago

That is hexadecimal notion. In hexadecimal after 8 come, 9,A,B,C,D,E,F, then again it starts with 10. As in, 8+8 =10, like in binary where adding 1+1 = 10

again it starts with 10. As in, 8+8 = 10, like in binary where adding 1+1 = 10.

• Reply • Share ›

Avata This comment was deleted.

fadoo ➚ Guest • 2 months ago

bc.....

• Reply • Share ›

Piyush • 4 months ago

Well,i am slightly confused at the part where two indexes were used to maintain the list [list with actual nodes & list of empty nodes].can somebody pls. elaborate.

• Reply • Share ›

DS+Algo ➚ Piyush • 2 months ago

we got two linked lists now, one for maintaining data list and other for empty list to allocate or deallocate memory fast

1 • Reply • Share ›

xerosanyam • 5 months ago

This article is slightly less clear in one reading. It is nice though. :)

• Reply • Share ›

Avata This comment was deleted.

DS+Algo ➚ Guest • 2 months ago

that's hexadecimal

• Reply • Share ›

uday • 6 months ago

In the summary it is mentioned,
B) Linked List means non-contiguous linked memory. It can exist in any memory section be it Heap or Data or Stack.
can the linked list memory in Stack? I think it is not even incase of simulating linked list with array? am i right?

• Reply • Share ›

vicky • 10 months ago

can you please explain that "Why array resides in stack section?"

• Reply • Share ›

Anonymous ➚ vicky • 7 months ago

Dynamic allocation resided in Heap section I mean either your allocating it for array or anything. Any allocation whose size can be known to compiler before runtime resides in Stack section

1 • Reply • Share ›

Surbhi • 10 months ago

int * dynArr = (int *)malloc(sizeof(int)*arrSize);

How can we use index to access elements?

• Reply • Share ›

Adauta Garcia Ariel Surbhi • 7 months ago

As "anonymous" say, you can access using array notation like:
dynArr[index]

Or pointer notation:
*(dinArr + index)

3 • Reply • Share ›

anonymous Surbhi • 10 months ago

by using simple array notation u can access elements
e.g., to access i-th element : dynArr[i];

1 • Reply • Share ›

Akash Verma Surbhi • 5 months ago

Here you go,

//Code to declare a dynamic array

#include<stdio.h>

int main()

{

int size,i;

printf("Enter Array size required\n");

scanf("%d",&size);

int *dynArr=(int *)malloc(sizeof(int)*size); //DYNAMIC ALLOCATION OF MEMORY FOR ARRAY... FROM HEAP

for(i=0;i<size;i++) {="" scanf("%d",&dynarr[i]);="" }="" printing="" the="" elts="" read="" for(i="0;i<size;i++)" {="" printf("%d="" ",dynarr[i]);="" }="" return="" 0;="" }="">

• Reply • Share ›

devakar verma • a year ago

i m not clear about the addressing i.e after 0x508 it comes 0x510.
as you are using hexadecimal representation then how can we get this by adding 8 to 0x508.
please make it clear

• Reply • Share ›

Barcode devakar verma • a year ago

It is in HEX format
after 8 there are 9,A,B,C,D,E,F and then 10
so 8+8 in hex = 10

Hope it is clear now

9 • Reply • Share ›

Naveen K Chandravanshi • a year ago

good forecasting and visualization can be helpful when required in some situation.....thanks.

• Reply • Share ›

drake • a year ago

why do such huge work we can just have array of structures I don't find any use of it...pardon my knowledge If I am wrong but please let me know that where I am wrong ....!!

• Reply • Share ›

Kim Jong-il drake • a year ago

@drake Dear read the take away, He has clearly mentioned that why he has written that.!
I am asking some question which can be answered only on the basis of this notes.
1: Can a linked list have memory in stack? (since we know that all the node is created by malloc so it will be in heap.)
2: If there is a language which does not support malloc then how can we implement linked list.?

7 • Reply • Share ›

Anonymous drake • a year ago

drake, i think that the article is trying to provide some insight on what could be the approach of implementing malloc(). of course, real malloc() would have done lot more optimizations etc. besides, this write-up is asking us to look at array vs. linked list from memory perspective...

3 • Reply • Share ›

Anonymous drake • a year ago

you will understand the difference when you deal with things like computer graphics..you may find it handy to use more pointers than arrays there! you cannot predict how things are going to grow , in this scenario its clever to use a pointer than array!

• Reply • Share ›