es

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

In general, array is considered [illegible] at the compile time and array memory is allocated either fro [illegible] ck 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:

in

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, 0x108 and 0x10C respectively.

```
[(1)]     [(2)]    [(3)]    [(4)]
0x100     0x104    0x108    0x10C
```

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 known 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 an array are contiguous in memory, we can access any element randomly using index e.g. intArr[3] will directly access the fourth element of the array. (For newbies, array indexing start from 0 and that's why the fourth element is indexed with 3). Also, due to contiguous memory for successive elements in the 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 the next node and this forms the basis of the link from one node to the next node. Therefore, it's called a Linked list. Though storing the location of the next node is overhead in linked list but it's required. Typically, we see the linked list node declaration as follows:

```
struct llNode
{
  int dataInt;

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

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 the next node. Apart from this difference, we can see that the array could have several unused elements because memory has already been allocated. But the 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);
```

Though the memory of this array is allocated from Heap, the elements can still be accessed via the index mechanism e.g. dynArr[i]. Basically, based on the programming problem, we have combined one benefit of the array (i.e. random access of elements) and one benefit of the 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 a 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 a 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 the embedded system manages its own memory. In short, if we need to perform our own memory management, instead of relying on the 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 the linked list using an array. Now, let us first see how this can be done. Suppose, type of a node in linked list (i.e. the 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];
```

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 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 the last node (i.e. fourth node) is set to -2. This (i.e. -2) is done to denote the end of the linked list. Also, the 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 the second node from our linked list, the memory for this node is still there because the underlying array is still there. But the nextIndex of the first node now points to the 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 an algorithmic manner.

There are multiple ways to do so. If we take the simplistic approach of creating linked list using an 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 the current head node (i.e. head index) of the linked list. Finally, make the index of this new node as the 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 the 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 indexes for linked list of empty nodes. By doing so, whenever, we need to insert a new node in the 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 a 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 takeaways 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 a memory perspective could provide us with 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

GeeksforGeeks has prepared a complete interview preparation course with premium videos, theory, practice problems, TA support and many more features. Please refer Placement 100 for details

Recommended Posts:

Find the middle of a given linked list in C and Java
Program for n'th node from the end of a Linked List
Write a function to get Nth node in a Linked List
Given only a pointer/reference to a node to be deleted in a singly linked list, how do you delete it?
Detect loop in a linked list
Write a function to delete a Linked List
Write a function that counts the number of times a given int occurs in a Linked List
Reverse a linked list
Given only a pointer to a node to be deleted in a singly linked list, how do you delete it?
Write a function to get the intersection point of two Linked Lists
Function to check if a singly linked list is palindrome