Array vs Linked List: Operations and Their Efficiency

IN-CLASS KINESTHETIC LEARNING ACTIVITY

This is a script of an in-class kinesthetic learning activity that illustrates the pros and cons of storing data in array-based and linked list-based data structures. Using the classroom's physical space as an analogy for computer memory space, free chairs to represent available memory, and students holding songs as example data items, this demo brings the issues of array- and linked list-based implementations into the familiar and relevant context, the classroom. It makes it easier for the students to visualize occupied and free memory space, and to understand the mechanics of memory manipulation operations. Students can see how their choice of a data structure affects the space occupied by a song collection on their phone and the speed of operations on this collection.

PREREQUISITE KNOWLEDGE The students should already have seen the idea that computer memory can be viewed as a sequence of equal-size consecutively-numbered memory slots, each of them having a unique address, and the linear search algorithm.

If students have studied the binary search algorithm, binary search process can be discussed; if not, it should be omitted. Students should already be familiar with the basics of algorithm analysis and big-Oh notation. This activity might still be helpful for providing intuition for using arrays in some situations and linked lists in others if students are not familiar with algorithm analysis. In the latter case, the mention of big-Oh bounds can be replaced with their informal explanations such as "almost all elements need to be moved" or "just a few elements need to change positions".

In my course this activity follows basic Python programming module and students are already familiar with Python list and its operations such as indexing, appending, and deleting.

PROPS Data items (e.g. song titles or numbers) printed on paper in large font, and 5 chairs in front of the classroom, facing the students. The instructor could also solicit song names from the students and write them with a dark marker on standard paper in large bold font.

IMPLEMENTATION NOTES This engaging activity works in a face-to-face classroom of any size as long as the front of the room and data items (songs) can be clearly seen from any point of the room. It's a low-stakes activity for the students, so volunteers might be called or roles can be assigned. In the linked list part, it is possible to engage those students from the back rows who don't usually volunteer.

When trying this activity for the first time, I found it helpful to warn the students that it was an experimental activity, and that I might decide to abort it if it became more confusing than helpful.

Introduction

Instructor: Today we'll compare and contrast two ways of organizing ordered data, such as photos in a photo gallery app, or songs and podcast episodes in a music player app. When we first download a music player app, our song collection is empty. When we start using the player, the song collection grows and changes. When we see a list of tracks on a particular album, we'd like to be able to access, say, track 4 (i.e. to index). When we hear a new song we like, we might want to add it to our collection. When we get bored with a song, we delete it from the collection. We might want to play all songs in the collection (i.e. to iterate). Or we might want to know whether we have a particular song (i.e. to search).

Array list

Find 4 volunteers and distribute the songs among them. (Alternatively, come up with example song titles together with the class and write them down on paper, then distribute among the volunteers). Invite the volunteers to sit in the first four chairs and to hold their songs.

This is the example of data storage in an array of memory slots. Chairs represent memory slots, and people represent data items with songs. In this case, five consecutive memory slots are allocated for the array, and four of them are taken. The address of the array is said to be the address of the element at the position 0.

INDEXING We can quickly access an array element at the particular index. To access a song at position 3 (element at index 3), just take the starting address of the array, add size of three slots to it and this would give us the address of the element at index 3. This quick access is enabled by having elements in contiguous slots in the memory, and giving each element equal number of bits. Accessing an element at index 3 takes O(1), constant time. Imagine a longer array of size n. Accessing an element at position n-1 also takes O(1) time.

ITERATION We can print or display all these data by iteratively picking consecutive elements and displaying their data.

Instructor illustrates the steps of the iteration, starting from the beginning and looking at one song at a time.

ADD ELEMENT We can add a new element to the end of the array.

Give a new volunteer a song and have them take the last chair.

Notice how easy it is to add a new element to the array when there is a free spot at the end of the array. We just place them there, other elements are undisturbed; total time spent on this addition is not affected by the size of the array — it's constant.

What if there're no free spaces?

Someone in class may propose to add another chair, or the instructor can ask "Should we add another chair?" and solicit responses.

We cannot simply add another chair at the end, because that memory space is reserved for some other application, and we may overwrite a piece of data that doesn't belong to us – not good.

We have to find a new contiguous space in memory for 6 or more elements. Problem number one, it may not be possible due to *fragmentation*. This classroom has 6 free chairs but does not have 6 *contiguous* free chairs, similar to how the campus might have free total square footage sufficient to host a stadium but doesn't have a *contiguous* space for that. Problem number two, even if we find the necessary space, all songs that are in the array will now have to stand up and move to their new locations. Five elements will be disturbed in order to add one. Had there been a million elements in place already, a million elements would have to be moved. I.e. we'll have to copy all data that's presently in the array in order to add one new element, making the time for one addition linear in the size of the array.

[I could also, in anticipation of growing the array, reserve a large number, say, a hundred chairs, for the array. As you can imagine,

- this doesn't take care of the problem completely – what if I need to store 105 elements?
- if we're only working with a couple of elements at some point, it leads to the inefficient use of space.]

So, in the best case addition to array takes constant time, but in the worst case it takes linear time.

REMOVE ELEMENT Let's consider element removal.

Remove a song from the beginning by having the first person stand up and return to their seat in class. They no longer participate in the demo. An empty chair remains.

We have a gap in our data that needs to be repaired. Now every song following the deleted one has to stand up and move one slot to the left. If we don't do it, we will lose constant-time indexing (because element 0 will be at position 1).

Ask the students (elements of the array) to stand up, one at a time, move one slot to the left, and take a new seat.

Notice how every song in the array was disturbed when one song was deleted. If we have 1000 elements, 999 of them will have to move when the first one is deleted. So removal of the element is, in the worst case, linear time operation, proportional to the size of the array.

Optionally, consider the insertion and binary search.

INSERTION Inserting an element at a particular position would require moving everyone to the right of that position one slot over to make room for it (in the best-case scenario, when free space is available) and putting a new element into the free index (linear time worst-case performance).

For binary search, first sort the songs in alphabetic order.

BINARY SEARCH Let's see how binary search can be performed in an array. The memory address of the middle element is easy to calculate, just take starting address of the array and add n/2 to it. Memory is random-access, meaning that any slot in the memory is thought to be accessible in small (constant) time, and hence accessing this middle element will take constant time. Checking the song there will also take constant time. Calculating the boundaries of the relevant half of the array will also take constant time. Overall, binary search will take O(log n) time.

Thank the volunteers. Invite them to return their song title and to take their regular seat in the classroom.

Summary An array structure provides a constant-time access to an element at a particular index. Finding an element, if elements are sorted, can be accomplished by efficient binary search algorithm in logarithmic time. Arrays allow is to store ordered data. However, the data update operations, such as addition and removal, require linear time. What if we need more efficient *add* and *remove* operations, and we don't need to *search* that often?

We will now design a different data structure that supports faster addition and removal of elements in some cases.

Linked list

Overview In this new data structure the elements will be chained together like beads of a necklace. Now, the elements in this new data structure (we'll call them "nodes") will not only hold the data (with their right hand), like in an array, but will also point to the next element (with their left hand). These individual elements can be located anywhere in the memory where a free spot exists, not just in the continuous chunk of memory, like was the case with the array.

Linked list technicalities

We'll start with two pointers: head and tail.

Pick two students in the classroom. I usually pick one in a baseball hat to be a "Head" and one with a ponytail to be a "Tail".

Initially they don't point to anything (or point to each other). They will be my only way to access the list.

Create one node by giving a student a sheet with a song, and asking Head and Tail to physically point to them.

Note how this element can be located anywhere where there's a free slot, and this element and Head and Tail don't need to be next to one another.

Add element at the end

Create one more node by giving a student a sheet with a song, and append it to the end of the list: ask the last node pointed to by Tail to point to the new node, and ask the Tail node to point to the new node.

Repeat two more times.

What are we observing? Additions at the tail are easy: only three entities — last node, tail, and the newly added node — need to be updated to point to different memory locations. All other nodes remain completely oblivious to the fact that there were additions. So this operation disturbs a constant number of nodes and requires constant time.

Illustrate addition at the head in a similar manner.

Iteration We can access and view titles by starting at the head and following the *next* pointers up until we reach the Tail.

Instructor illustrates the steps of the iteration, starting from the Head and looking at one element at a time.

Remove element Let's now consider element removal.

Remove element from the beginning by simply resetting the Head to point to the second element.

Removal at the head of the list takes constant time. Only the Head is reset, everyone else is undisturbed. Removing element at the end is trickier. We need to set the Tail to point to a different (penultimate) node, but we have to find it first. The number of operations will be proportional to the length of the list.

Indexing Now what if we need to skip to the third song? Because the nodes are scattered in the memory, we can't just know the address of the third node. We have to start at the Head and follow three pointers to get to the third element. Analogously, if we have to get to the 100th node, we'll have to follow 100 pointers before we get there. Generally, if we go to the nth node, we have to follow n pointers. There is no way to get to the elements in constant time like was the case with the array.

Optionally, consider the insertion and binary search. For binary search, first sort the songs in alphabetic order.

Binary Search How about search? What if we want to find if a particular song is in the list? If songs are not sorted, we have to scan all songs, O(n) time. But if the songs are sorted, we could try to do binary search. We first have to find a middle element. We know its number, but we can't get to it in constant time, we have to traverse half of the list. Binary search cannot be carried out efficiently!

Thank the volunteers. Collect the song cards and invite the volunteers to return to their seats.

Summary

In the linked list additions to the beginning and to the end of the list can be done in constant time. Removals from the front can also be done in constant time. Most elements are undisturbed by these operations and can be located in separate parts in the memory.

However, quick indexing is no longer available to us, and neither is binary search.

Also notice the additional space requirement. In an array, only the songs were stored. In a linked list, each node not only stores the songs, but also the pointer to the next node.

Final notes

Which of the two structures is better? It depends on the purpose.

Solicit (or give) example applications where an array would be preferable to a link list, and vice versa.