Notes - Algorithms & Data Structures - Arrays and Linked Lists

- Dr Nick Hayward

A brief intro to data structures with *arrays* and *linked lists*.

Contents

- Intro
- Arrays
- Linked list
    - issues with linked lists
- Benefits of arrays
- Runtime comparison - part 1
- Insertion in the middle
- Deletion
- Runtime comparison - part 2
- General usage preference
- App to memory
- Process and memory usage
    - process and state
- Process manager
    - process scheduling
- Summary

Intro

As we use algorithms in applications and systems, we need to store, retrieve, and manipulate data.

This is a fundamental and key part of working with algorithms.

Each piece of data is stored with an address in the computer's available memory, ready for access by the system and application.

For example, we might store some data as follows

.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	X	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

A defined address for `X`, e.g. `ff0edfbe`, will allow the system to reference and recall the stored data.

So, whenever we need to store some data, the computer will allocate some space in memory and assign an address.

If we want to store multiple items in an organised structure, we may consider a data structure.

If we create an app to store notes, to-do items, and other data records, we might store these items in a list in memory.

There are many different data structures we might consider, including an array or linked list.

Arrays

Initially, we'll consider an *array* data structure for this list of items.

From a conceptual perspective, an *array* will store each list item contiguously in data. In effect, they are stored next to each other, one indexed value after another.

Arrays are implemented in different configurations, and with varied limitations, relative to the chosen programming language. For example, we might consider the following scenario for a basic array

- store the initial list items in contiguous blocks of memory - e.g. 5 items stored
- add a 6th item to array
- 6th block of memory is already allocated to data
  - move 5 blocks of data for array to empty memory and add 6th block
- add 7th item to array
- add 8th item to array
- 8th block of memory is already allocated to data
  - move 7 blocks of data for array to empty memory and add 8th block
- ...

With this simple pattern, we can manage a basic array. It's predicated on available memory blocks, and the efficiency of algorithms to ensure it works smoothly for the application and system. In effect, it becomes reliant on the following

- array data structure algorithm
  - add data
  - move data
  - manage data - including index, size, &c.
- memory management algorithm for underlying system
  - read data
  - move data
  - resize data
  - ...

However, it may not be the best option for each programming language and system. Instead, we might consider an initial reserved size for the array, such as 15 slots in the array for data.

With this option, we know we may now add up to 15 items to our data structure without worrying about resizing or moving the array in data. However, there are also issues with this solution to array and memory management. For example,

- wasted memory allocation for unused slots
    - e.g. add 12 items, and 3 slots are left empty and unused in memory
    - unused memory is still allocated to the data structure, and may not be used elsewhere by default
- more than 15 items will still require a move of array in memory, and a resize of the underlying data structure

Linked list

By comparison, we may also consider a *linked list* data structure to store our list of items in memory.

With a linked list, the data may be stored anywhere in memory. Each item stores the address of the next item in the list. In effect, we may *link* together random memory addresses as a contiguous structure.

.	.	.	.	.	.	.	.
.	X	.	.	.	X	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	X	.	.	.	.	.
.	.	.	.	X	.	.	.
.	.	.	.	.	.	.	.
.	X	.	.	.	.	.	.

When we request an item from the list, it returns the address of the next available item. It's like following a trail of clues to find the answers.

With a linked list, we do not need to move items in memory.

We may also add a new item anywhere in memory, and then save the address to the previous item in the list. With a linked list, we do not need to move items.

So, if there's space available in memory, there is space for a linked list.

issues with linked lists

Whilst a linked list seems a preferable solution, we may still encounter noticeable issues with such lists.

For example, if we need to access item 10 in a linked list we need to know the address location memory. This mean's you need to get the address from the previous item in the linked list.

However, that item needs to get the address from the previous item, and so on to the first item in the linked list.

So, we can quickly see that a linked list is great if we need to access each item one at a time. We may read one item, and then move to the next item, and so on.

If we need to access items out of order, and on a regular basis, then a linked list is a poor choice.

Benefits of arrays

Accessing an array is a noticeable benefit compared to a linked list.

We know the address for every item in the array, and can quickly and easily access an indexed item.

So, arrays are a good option if we need to access random items on a regular basis. We may easily calculate the position of an array item, which contrasts strongly with the rigid pattern of access for a linked list.

Runtime comparison - part 1

Comparative run times for common operations on arrays and lists

	Arrays	Lists
reading	O(1)	O(n)
insertion	O(n)	O(1)

key:

- O(n) = linear time
- O(1) = constant time

We see *linear time* for *array* insertion and *list* reading, and constant time for *array* reading and *list* insertion.

Insertion in the middle

We may need to modify our data storage for an app to allow insertion in the middle of the data structure.

However, our choice of array or linked list will also affect this option and efficiency of insertion.

For example, if we consider a linked list it is as easy as modifying the address reference of the previous element to point to the inserted data item.

By contrast, for arrays we need to shift all of the remaining elements down to create space for the inserted items. If there is not sufficient space in the current address location, we may also need to move the whole array before we can insert new items.

So, we may see a performance benefit for insertion to the middle of a linked list compared to an array.

Deletions

Likewise, which option is preferable for deletion?

For most use cases, it will be simpler to delete an item from a linked list. We only need to modify the address reference for the previous item in the list.

However, for an array, we need to move the whole array to accommodate the deletion.

Runtime comparison - part 2

We may update our comparison table to now include *delete* operations for both arrays and linked lists.

	Arrays	**Lists**
reading	O(1)	O(n)
insertion	O(n)	O(1)
deletion	O(n)	O(1)

key:

- O(n) = linear time
- O(1) = constant time

However, it's worth noting that both insertions and deletions may be `O(1)` run time only if we may access the element instantly.

For example, it is common practice in such algorithms to maintain a record of the first and last items in a linked list. This will then only take `O(1)` run time to delete such items.

General usage preference

After considering both data structures, which option is more commonly used for app development?

Context is, of course, a valid consideration when choosing a data structure. However, *arrays*, for example, may see frequent use due to their support for easy random access of data items.

As we've seen, these data structures support two initial types of access, *random* and *sequential*.

Sequential access provides each data item in a consistent, predictable order, which is exactly what we see when accessing a linked list data structure. In fact, this is the only way to conveniently access data in a linked list.

Another benefit of random access is a speed improvement in reading data, which helps improve the performance of array data structures.

We may also see both arrays and linked lists used as the foundations for other, often specialised data structures.

App to memory

So, as we design an appropriate algorithm, we need to consider how an OS and application handle and use memory. For example, an OS's management of memory is closely associated with *process* requirements and usage.

However, memory management relative to a process (e.g. application) may be considered broadly as follows

- ensure each process has enough memory to execute
  - cannot run out of memory or use other processes' memory allocation
- different memory types must be organised efficiently
  - ensures effective management of each process

So, we may start by managing memory boundaries for different processes.

Process and memory usage

If we consider the above restrictions and limitations of array implementation and management, we need a way to effectively manage this use of memory.

So, as a child process is created, it is assigned an address memory space.

Each process will see their memory space as a contiguous logical unit.

However, such memory addresses might actually be separate across the system. There will be disparate addressed memory spaces for each process, which may then be organised together, as needed, by the system's kernel.

For example, separate memory stores and addresses organised into a contiguous group per process.

One obvious benefit is the efficient use of memory space. There is no need for pre-assigned large chunks of memory or, perhaps, reserved memory that is never used by a process.

The kernel controls access for a process to memory addresses. In effect, the kernel is controlling the conversion of assigned virtual addresses to a physical address in the system's hardware memory.

process and state

Each child process may have a related *state*, associated during the lifetime of the process itself.

This state may be monitored by the system's *kernel*, and a process will wait until resources are available to allow a change in state.

The kernel may then switch processes relative to an update in a process' state.

Process manager

The *process manager* is responsible for processes in a system.

It is controlled by the system's *kernel*, and manages the following

- process creation and termination
- resource allocation and protection
- cooperation with device manager to implement I/O
- implementation of address space
- process scheduling

process scheduling

As noted above, a key part of managing processes in a system is efficient scheduling.

Whilst such scheduling is part of the *process manager*, it is actually maintained by the system's kernel.

The kernel is responsible for switching between processes, checking and migrating available *ready* state processes to execution in an *active* state.

In effect, the *kernel* is selecting processes to execute in the system on the available CPU. So, the *kernel* is choosing the next process to run on the CPU.

This context switch is informed by the required and available *process properties*.

This selection of process is also determined by the nature of the process itself, i.e. is it I/O bound or CPU bound.

An algorithm helps determine the best process choice to ensure a system runs efficiently and without apparent delays. Example algorithms include,

- first-come, first-served
- shortest job next
- round robin
- multi-level priority queue

So, scheduling is meant to provide a fast and efficient system. The kernel chooses processes to allow the system to run fast. For example, it is common to assign priority to a *front-facing* process over one running in the *background*.

Summary

We may summarise the above as follows,

- an array or list may be used to store multiple elements for a given data set
- in an array, all elements are stored contiguously in memory
- in a list, elements may be stored at separate addresses
    - each element stores the address of the next element as well
- arrays commonly provide fast read operations
- linked lists provide fast insert and delete operations