Section 07: Solutions

1. Design Decisions

We've now talked about a few ADTs and a variety of data structures. For each of the following scenarios:

(a) Say which ADT you think most closely corresponds to what you want to do with the data.

(b) Describe a data structure you would use to implement that ADT.

(c) Justify your decisions

For these questions we care about the justification. In some cases there may be more than one "right" ADT or data structure.

(a) You are running a gaming server, where each player has a rating. Each time a player makes a `NewGame` request, you should be able to suggest another player who has a similar rating.

Solution:

For the ADT, some reasonable choices would be a `List` or `Set` of players or a `Dictionary` where ratings are the keys and the players are the values.

For the data structure, here are some possibilities:

An `ArrayList` of players, sorted by their rankings:

- Inserting or removing players would be a slow operation ($\mathcal{O}(n)$ time).
- But `NewGame` is very efficient, and easy to implement – just find the player who requested the game (probably with a binary search), and choose someone nearby in the `ArrayList`.

An `AVL tree` of players, again sorted by their rankings:

- Inserting or removing players is much more efficient ($\mathcal{O}(\log n)$ time).
- Implementing `NewGame` is a little complicated, but you can do it in $\mathcal{O}(\log n)$ time.

  What you want is a node that would be nearby if you did an in-order traversal of the entire tree. But the traversal would take a while. You can find the next item in a traversal in only $\mathcal{O}(\log n)$ time. It's a good challenge problem to figure out exactly how to do it (we won't go into the details here).

  But what if you know something about the ranking system? Maybe your ranking system isn't that specific – it's just an integer between 1 and 100. Then we can be even more efficient. Just make an `array` of size 100, where each entry is a list of players with that ranking. If we use a linked list, we can insert in $\mathcal{O}(1)$ time. And finding a nearby player is easy – if there's someone else in your list, choose them. Otherwise, find the closest empty list and grab someone from there – since there are only 100 lists, that's also $\mathcal{O}(1)$. In theory, you could use this approach even if there are more than 100 possible values. In practice, it would be slower than our other solutions if too many of the lists are empty.

Which should you choose? It probably depends on your players. Are players logging on and off constantly? If so you might want the AVL tree for its more efficient insert and removal. But if they're playing a dozen games in an average session, then getting the simple, efficient `NewGame` of the `ArrayList` may outweigh the slower logon/logoff times.

You might be able to come up with other benefits or drawbacks. For example, with the `ArrayList` it's easy to find **everyone** with a similar rating, not just one person.

(b) Your project manager wants to make her job of approving code changes easier. Each push to your code base comes with an automatically generated number, its `magnitude` –an integer representing the size of the change made. Your PM wants to be able to do two things

(i) `reviewBiggestChange` finds the unapproved change of largest `magnitude`

(ii) `approveSmallChanges(k)` given a number k, automatically approves all changes to the code base of `magnitude` less than k.

Solution:

Processing objects in order of importance sounds a lot like a `Priority Queue`. What should we choose for the actual data structure?

One option is a `Max-Heap`. Developers pushing to the database probably won't be too upset with the $\mathcal{O}(\log n)$ time added everytime they push. And $\mathcal{O}(\log n)$ time to `reviewBiggestChange` are both very fast. But `approveSmallChanges` won't be particularly nice. A heap isn't fully sorted, so even if there are only a few small changes to approve, in the worst case you might need $\mathcal{O}(n)$ time.

Another option is to use a `Sorted Doubly Linked List`. Insertion is now pretty slow – $\mathcal{O}(n)$ in the worst case, but maybe your test suite runs so many tests on each push that you won't notice the extra time. `approveSmallChanges` is also now much more efficient – just start at the end with the small changes and delete until you find one above the threshold. The time is proportional to the number of elements approved – much better than $\mathcal{O}(n)$ for the heap. `reviewBiggestChange` is also a bit faster – $\mathcal{O}(1)$ instead of $\mathcal{O}(\log n)$.

Which should you choose? Neither is particularly tricky to implement, but the disparity in `approveSmallChanges` and inserting could be significant. Will inserting slow down developers making pushes? If not the `linked list` is probably the best choice. Otherwise the heap is a reasonable choice.

(c) In your user-updated encyclopedia, when someone types certain keywords into the searchbar, they should be taken directly to an associated article (instead of to a search results page). You need to store a list of keywords along with the page that each keyword will redirect to.

Solution:

We have keys (the keywords) and values (the destination webpage) – we want a dictionary.

Two implementations jump to mind:

AVL trees – which will have $\mathcal{O}(logn)$ time for every relevant operation in the worst case.

Hash Table – which will have $\mathcal{O}(1)$ time on average, but could hit $\mathcal{O}(n)$ time for `find`, `insert`, or `delete` in the worst case.

Which should you choose? It probably depends on how much you trust your users. Since you're letting users choose keys, nefarious users might be able to choose keys that will collide and cause the hash table to slow down. Slow finds would be particularly bad as they could slow down the redirect for users. In that case, an AVL tree would be preferable.

On the other hand, intentionally creating such a set of keys would be difficult. If you can trust your users, or watch for nefarious inputs, the hash table will be faster.

(d) You're running a restaurant for cats. Normally, your restaurant is first-come, first-serve: you want to feed the cats in the order they arrive. But if some of the cats waiting in line are really hungry, they will disturb the other customers by meowing loudly, so you also want to be able to find the hungriest kitties and feed them first.

Solution:

Our first idea might be to use a `Queue`. If we implement the queue with a `doubly linked list`, then adding cats to the back of the queue and removing them from the front are both $\mathcal{O}(1)$. What about finding the hungriest cats? We can search the queue for the hungriest cat in $\mathcal{O}(n)$, but if there are lots of hungry cats, we may have to do this multiple times, potentially increasing the runtime to $\mathcal{O}(n^2)$. If all of the cats are very hungry. Is there a faster way?

We could try using a `Priority Queue`, but since we still want to serve the cats in the order they arrive most of the time, we can't get rid of our normal `Queue` since a `Priority Queue` doesn't remember the order elements were inserted into it. When a cat arrives, we have to add them to both data structures, so adding a cat is $\mathcal{O}(\log n)$, and removing a cat is also $\mathcal{O}(\log n)$ if the heap uses an extra data structure to find elements quickly. (Finding the index of an element in a heap is normally $\mathcal{O}(n)$ since a heap's array is not fully sorted, but with clever use of extra dictionaries, you can decrease the time to $\mathcal{O}(1)$ if you use a hash table to remember the index of each element.)

Now feeding the hungriest cat is $\mathcal{O}(\log n)$, up to $\mathcal{O}(n \log n)$ if all of the cats need to be served in hungriest-first order. (We also have to find and remove each cat from the original `doubly linked list` queue, but this can be done in $\mathcal{O}(1)$ if each cat in the heap has a pointer to the linked list node.) That's a great improvement over $\mathcal{O}(n^2)$! But:

(i) Adding and removing cats in first-come first-serve order is slightly slower: $\mathcal{O}(\log n)$ instead of $\mathcal{O}(1)$.

(ii) We use considerably more memory because each cat needs to be stored in multiple data structures.

(iii) The design is very complex and will probably be harder to implement and test.

Whether this is worth it depends on how often you need to find the hungriest kitties compared to serving everyone in a normal queue order, how much memory the computer has, and how much time you have to implement it.

2. Sorting

(a) Demonstrate how you would use quick sort to sort the following array of integers. Use the first index as the pivot; show each partition and swap.

[6, 3, 2, 5, 1, 7, 4, 0]

Solution:

[Solutions omitted]

(b) Show how you would use merge sort to sort the same array of integers.

Solution:

[Solutions omitted]

3. Sorting Decisions

For each of the following scenarios, say which sorting algorithm you think you would use and why. As with the design decision problems, there may be more than one right answer.

(a) Suppose we have an array where we expect the majority of elements to be sorted "almost in order". What would be a good sorting algorithm to use?

Solution:

Merge sort and quick sort are always predictable standbys, but we may be able to get better results if we try using something like insertion sort, which is $\mathcal{O}(n)$ in the best case.

(b) You are writing code to run on the next Mars rover to sort the data gathered each night (Think about sorting with limited memory and computational power).

Solution:

Since each memory stick costs thousands (millions?) of dollars to send to Mars, an in-place sort is probably your best bet. Among in-place sorts, heap sort is a great choice (since it is guaranteed $\mathcal{O}(n \log n)$ time and doesn't even use much stack memory). Insertion sort meets memory needs, but wouldn't be fast.

(c) You're writing the backend for the website `SortMyNumbers.com`, which sorts numbers given by users.

Solution:

Do you trust your users? I wouldn't. Because of that, I want a worst-case $\mathcal{O}(n \log n)$ sort. Heap sort or Merge sort would be good choices.

(d) Your artist friend says for a piece she wants to make a computer sort every possible ordering of the numbers $1, 2, \ldots, 15$. Your friend says something special will happen after the last ordering is sorted, and you'd like to see that ASAP.

Solution:

Since you're going to sort all the possible lists, you want to optimize for the average case – Quick sort has the best average case behavior, which makes it a really good choice. Merge sort and heapsort also have average speed of $\mathcal{O}(n \log n)$ but they're usually a little slower on average (depending on the exact implementation).

She didn't appreciate your snarky suggestion to "just print $[1, 2, \ldots, 15]$ $15!$ times." Something about not accurately representing the human struggle.

4. Memory – Short Answer

(a) What are the two types of memory locality?

Solution:

Spatial locality is memory that is physically close together in addresses. Temporal locality is the assumption that pages recently accessed will be accessed again.

(b) Does this more benefit arrays or linked lists?

Solution:

This typically benefits arrays. In Java, array elements are forced to be stored together, enforcing spatial locality. Because the elements are stored together, arrays also benefit from temporal locality when iterating over them.

5. Memory – In Context

(a) Based on your understanding of how computers access and store memory, why might it be faster to access all the elements of an array-based queue than to access all the elements of a linked-list-based queue?

Solution:

The internal array within the array-based queue is more likely to be contiguous in memory compared to the linked list implementation of an array. This means that when we access each element in the array, the surrounding parts of the array are going to be loaded into cache, speeding up future accesses.

One thing to note is that the array-based queue won't necessarily automatically be faster then the linked-list-based one, depending on how exactly it's implemented.

A standard queue implementation doesn't support the `iterator()` operation, and a standard array-list based queue implements either $\mathcal{O}(n)$ enqueue or dequeue.

In that case, if we're forced to access every element by progressively dequeueing and re-enqueuing each element, iterating over a standard array-based queue would take $\mathcal{O}\left(n^2\right)$ time as opposed to the linked-list-based queue's $\mathcal{O}(n)$ time. In that case, the linked-list version is going to be far faster then the array-list version for even relatively smaller values of n.

The only way we could have the array-based queue be consistently faster is if it supported $\mathcal{O}(1)$ enqueues and dequeues. (Doing this is actually possible, albeit slightly non-trivial.)

(b) Why might f2 be faster than f1?

```
public void f1(String[] strings) {
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].trim();        // omits trailing/leading whitespace
    }
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].toUpperCase();
    }
}

public void f2(String[] strings) {
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].trim(); // omits trailing/leading whitespace
        strings[i] = strings[i].toUppercase();
    }
}
```

Solution:

Temporal Locality. At each iteration, the specific string from the array is already loaded into the cache. When performing the next process `toUppercase()`, the content can just be loaded from cache, instead of disk or RAM.

(c) Consider the following code:

```
public static int sum(IList<Integer> list) {
    int output = 0;
    for (int i = 0; i < 128; i++) {
        // Reminder: foreach loops in Java use the iterator behind-the-scenes
        for (int item : list) {
            output += item;
        }
    }
    return output;
}
```

You try running this method twice: the first time, you pass in an array list, and the second time you pass in a linked list. Both lists are of the same length and contain the exact same values.

You discover that calling `sum` on the array list is consistently 4 to 5 times faster then calling it on the linked list. Why do you suppose that is?

Solution:

This is most likely due to spatial locality. When we iterate through a linked list, accessing the value at one particular index will load the next few elements into the cache, speeding up the overall time needed to access each element.

In contrast, each node in the linked list is likely loaded in a random part of memory – this means we likely must load each node into the cache, which slows down the overall runtime by some constant factor.

(d) Suppose you are writing a program that iterates over an `AvlTreeDictionary` – a dictionary based on an AVL tree. Out of curiosity, you try replacing it with a `SortedArrayDictionary`. You expect this to make no difference since iterating over either dictionary using their iterator takes worst-case $\Theta(n)$ time.

To your surprise, iterating over `SortedArrayDictionary` is consistently almost 10 times faster!

Based on your understanding of how computers organize and access memory, why do you suppose that is? Be sure to be descriptive.

Solution:

This is almost absolutely because the `SortedArrayDictionary` is implemented with an array, which has much better spatial locality, than how the `AvlTreeDictionary` is most likely implemented, with a series of linked AVL tree nodes. Since we know that iterating over an array is faster than iterating over a linked list, the reasoning behind a faster tree is similar and reasonable.

(e) Excited by your success, you next try comparing the performance of the `get(...)` method. You expected to see the same speedup, but to your surprise, both dictionaries' `get(...)` methods seem to consistently perform about the same.

Based on your understanding of how computers organize and access memory, why do you suppose that is?

(Note: assume that the `SortedArrayDictionary`'s `get(...)` method is implemented using binary search.)

Solution:

Spatial locality can only be taken advantage of when iterating sequentially. With that, it's not surprising that because we have to jump from `i=100` to `i=50`, etc. until we find what we're looking for. If the array is so big that it spans over multiple pages, the locality that regular iteration takes advantage of is not available to jumping around with `get()`.