# Prioritizing Articles Based On Topic

## Array vs. Priority Queue

Sakeena Younus

Department of Computer Science, Lake Forest College

Professor Jamshidi

April 11, 2023

**Abstract**

There are many different built in data structures in Java that can be implemented for any problem, but in this paper, I explore the benefits of creating your own custom data structure to assess how much of a difference it makes. The algorithm I am using to code these data structures takes in a topic and list of articles and returns them to the user in order of relevance to the topic. The data structures I am comparing are a custom priority queue of nodes and 2D arrays. While both are sorted with insert sort, the results are extremely different. I assess performance of data structures based on the time complexities, as well as how native characteristics of the data structures warrant themselves to the algorithm.

## Prioritizing Articles Based On Topic - Array vs. Priority Queue

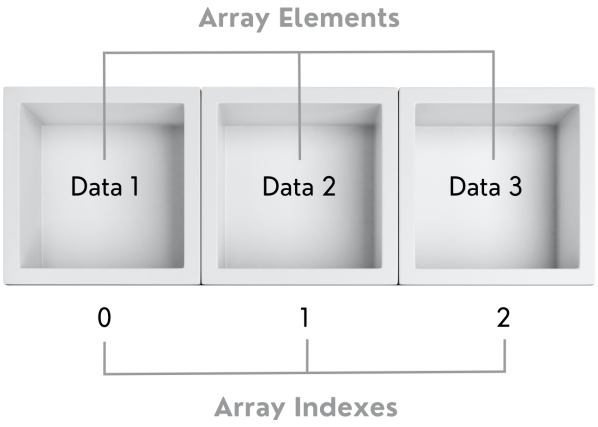
Think of your journey as a student, or just as a curious human being. We've all experienced being struck with an interesting thought, only to Google and find that our question was much too smart to be answered by the amazing pop-up box. So we surf through the plethora of articles not sure which to read first. That's where this algorithm comes into play. With the power of data structures, this algorithm prioritizes a list of articles based on relevancy to any topic. Feeling confused? Let's start from the beginning.

### What is a data structure?

To put it simply, a **data structure** in Computer Science is just a structure to store and organize data in a meaningful, efficient manner (GeeksForGeeks, *Data Structures*). Think of a bunch of files in a filing cabinet, where the files are the data and the cabinet is the data structure. There are many different kinds of data structures that are chosen based on the type of data, as well as how we want to utilize it. Let's start by discussing the simplest data structure in Java, the array.

### What is an array?

To keep it simple, an **array** is a data structure that stores multiple pieces of information together. Think of it as a set of cubbies in a row, each storing its own data. Each cubby also has its own label, which are numbers starting from 0.




The data stored in an array is referred to as *array elements,* and the labels for each element are referred to as *array indexes.* So if I wanted to refer to Data 1, I could say "array index 0."

**Fig. 1 - Array Structure**

There are two important characteristics of arrays:

1. The size of an array can never change. When you create an array, you give the set size to be created, and after that you can never add or delete an array index, just like you could never add a cube to a cubby already in use.
2. All of the data stored in an array have to be the same data type. A **data type** is the type of data being stored, for example, whole numbers, letters, or words. Some basic data types, referred to as primitive types, are:
    - ➔ int: Whole numbers - similar to integers in math
    - ➔ double: Decimal values
    - ➔ char: Individual characters
    - ➔ String: A collection of characters (words, sentences, etc.)

In a single array, you could never store a String at index 0 and an int at index 1. All elements of the array would have to be the same data type, either all Strings or all ints.

Despite being the simplest data structure in Java, arrays are extremely useful and used in many different scenarios. Some real life uses of arrays include:

- ➔ Shopping lists: The items of the shopping list are like the elements of an array, and the list number of the item is like the array indexes. *(See figure below)*

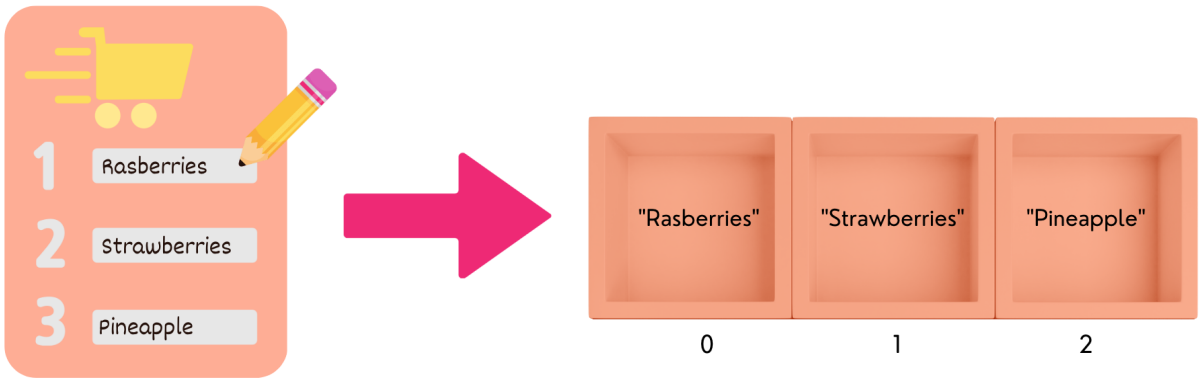
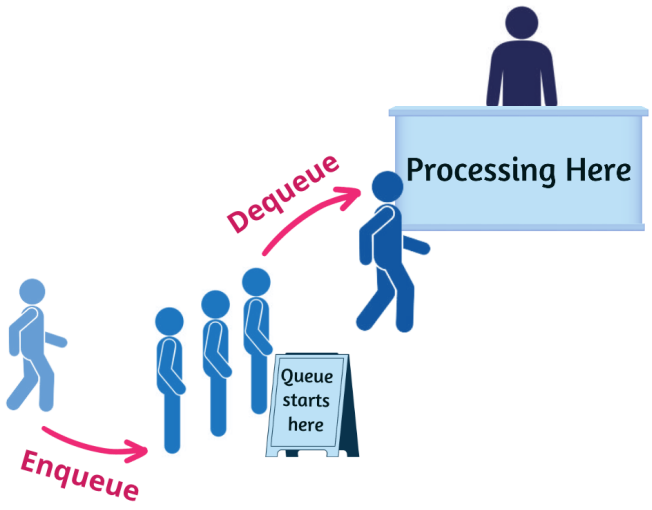



**Fig. 2 - Array Applications**

- ➔ Image Processing: Arrays can be used to store the values of each pixel in an image (GeeksForGeeks, *Applications, Advantages and Disadvantages of Array*). Different algorithms can then be used to manipulate each pixel, changing the image as a whole.
- ➔ Other Data Structures: The structure of arrays are used as a building block for many other data structures. One in particular we will talk about are called queues.

**What is a Queue?**

A queue *(pronounced Q)* is a data structure that holds data before it is processed. The word queue can remind you of people waiting in a line.




Just like a line of people, the data stored in a queue are added to the back and taken from the front to be processed. You can remember this easily by thinking: *queue – first in, first out.*

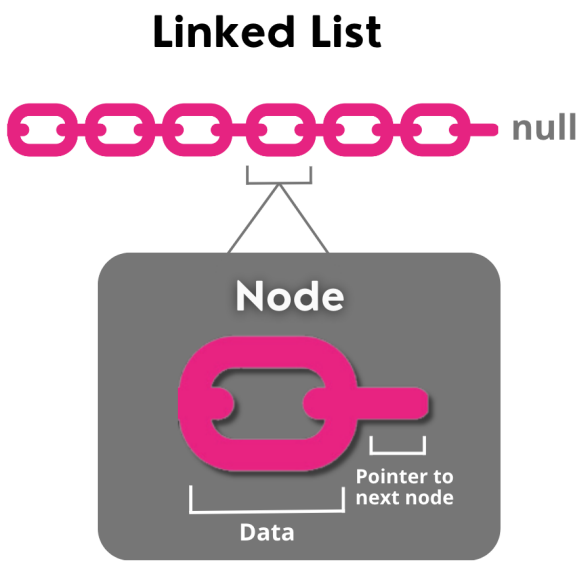
The process of adding data to a queue is called *enqueue,* and the process of removing data from a queue is called *dequeue.*

**Fig. 3 - Queue Process**

Now you may be thinking, *"If queues are based on arrays, how are we adding and removing elements?"* That is an excellent question. The answer is that queues *can* be based on arrays, but are not always. It is definitely possible, but because many problems need to add and remove elements efficiently, queues are created based on a more flexible data structure called a linked list.

**What is a Linked List?**

To understand the concept of a **linked list**, think of a chain made up of many links. Each link is referred to as a node, which contains 2 elements (like a 2 element array).




The first element is the data being stored, and the second element is a pointer to the next node. Every node has to have a pointer to the next node, so the last node just points to **null**, which essentially just means "nothing."

**Fig. 4 - Linked List Structure**

This structure makes it very easy to move nodes (chain links) around, as well as to add and remove nodes, which is exactly why I created my data structure based on linked lists.

**What is My Customized Data Structure?**

I have explained many different kinds of data structures, so here is what mine entails:

I created a modified queue where instead of adding elements to the back and taking from the front, I add the elements to the front and take them from the back, which logically works in the same way

I also structured it so that the data storage part of the queue contains a node holding 2 pieces of information, an article and its priority number.

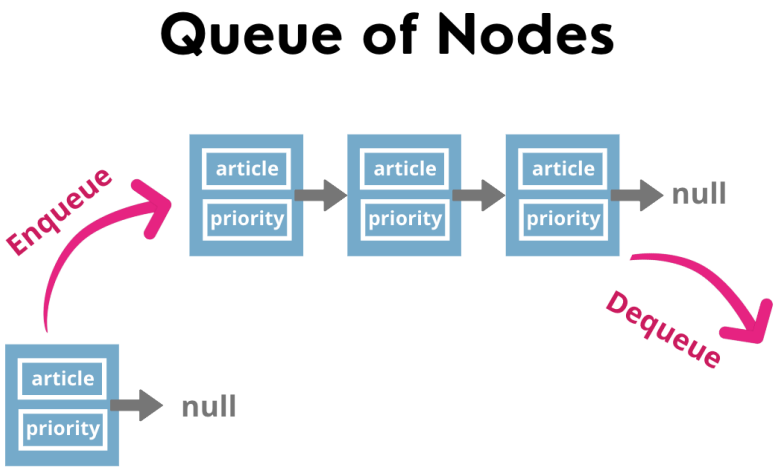



**Fig. 5 - Queue of Nodes Structure**

In my custom data structure, I also included a functionality that automatically orders my entries from highest to lowest using a sorting algorithm called insert sort. A queue that sorts its entries is referred to as a **priority queue**.

Some real life applications of priority queues include:

- ➔ Prioritizing patients in an ER waiting room due to the severity of the patient's condition.
- ➔ Scheduling tasks in busy work environments so that the most urgent tasks are completed first.
- ➔ Prioritizing calls in a call center based on urgency of question, circumstance, or request.

As mentioned earlier, different data structures are chosen based on the kind of data being stored, as well as how the data will be utilized. I customized my data structure to be a priority queue based on linked lists because it suited itself best to my algorithm.

**What is My Algorithm?**

Now that all the background information has been established, we can now discuss the algorithm that warrants the use of all of these data structures.

Steps of my algorithm:

1. Take a topic, for example, “Facts every green cheek conure owner needs to know", as well as any number of article names.
2. Compare each of the article names to the topic and assign the number of similar words as a “priority number” to each of the article
3. Articles are ordered from highest to lowest using insert sort.
4. The list is returned to the user in order of what articles they should read first.

I performed this algorithm using arrays as well as my custom data structure, the priority queue of nodes.

**Algorithm with Arrays**

While the algorithm is the same for both arrays and priorities, the execution is quite different. Let’s go through an example!

**Step 1:** Identify the topic and articles

Topic: “Facts anyone with green cheek conures needs to know."

Articles:

- ➔ *“My green cheek conure saved my life.”*
- ➔ *“Gluten - foe or friend?”*
- ➔ *“What should green cheeks eat?”*

We can store all of the article names in a array containing all strings called “Articles”


**Fig. 6 - Array: Article**

**Step 2:** Compare each of the article names to the topic & assign a priority number

*“My green cheek conure saved my life.”* - **2**

*“Gluten - foe or friend?”* - **0**

*“What should green cheeks eat?”* - **1**

We can store the article names and their priorities in a 2D String array called “result.” A 2D array is where the data being stored in the array…is another array! In *Fig. 7*, you can see that each index of the larger, light gray array, contains another smaller darker gray array of 2 indexes. Think of it as a 3 cube cubby where each cubby has two compartments.

The first compartment stores the article names and the second compartment stores the priority number of the article. Note that the numbers in the array are stored as Strings because arrays can only store 1 datatype.

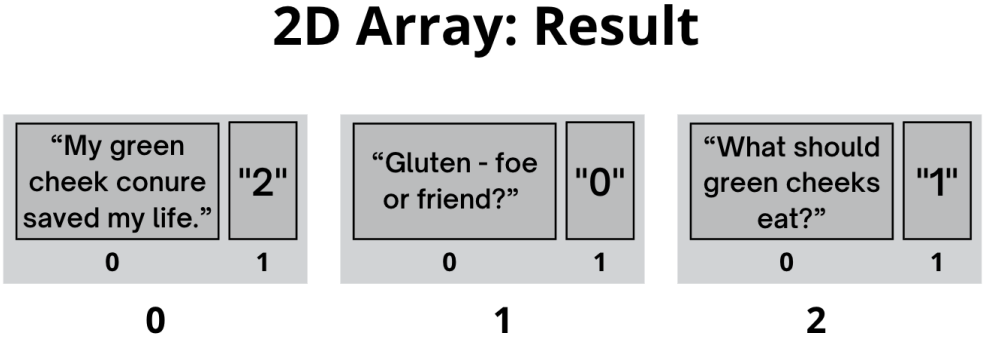



**Fig. 7 - 2D Array: Result**

**Step 3:** Order articles from highest to lowest using Insert Sort

In my insert sort algorithm, I create an empty 2D array called sortedArray to add my values into. Despite taking up more space in memory, it makes it much easier to program and understand the logic behind.

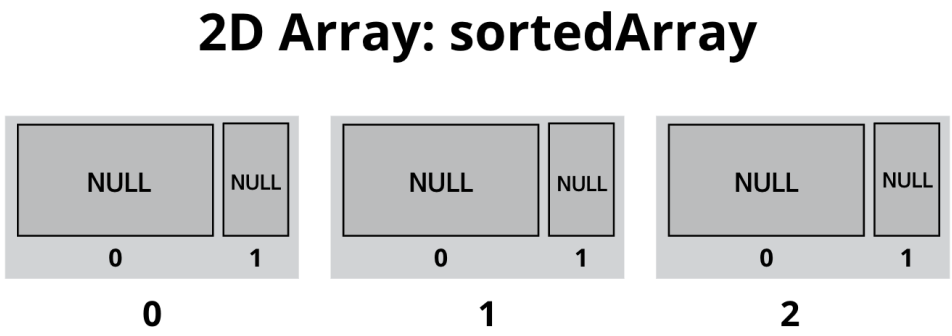



**Fig. 8 - 2D Array: sortArray**

Remember that Null just means “nothing,” so we view “sortedArray” as empty.

Insert sort continues to insert the elements of “Result” into “sortedArray” at the index where the article priorities are being ordered from highest to lowest. Here is what that would look like:

Inserting first element:

Because there is nothing in sortedArray yet, we just add element 1 of “result” into index 1 of “sortedArray.”

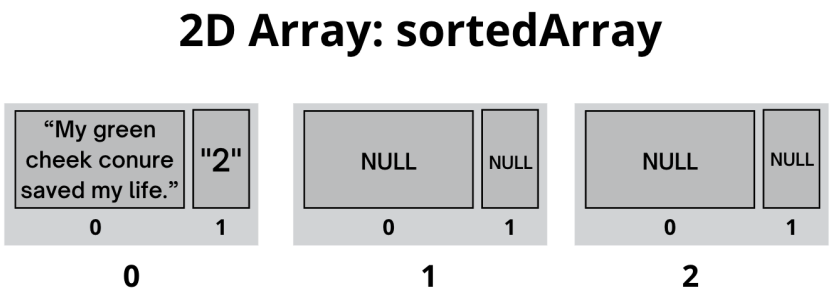



**Fig. 9 - sortedArray Insertion 1**

Inserting second element:

Because 0 (the priority number of the second element in “Result”) is less than 2 (the priority number of the first element in sortedArray), we can just add the new array into the second position of “sortedArray.” *(Remember these articles are being sorted in descending order).*

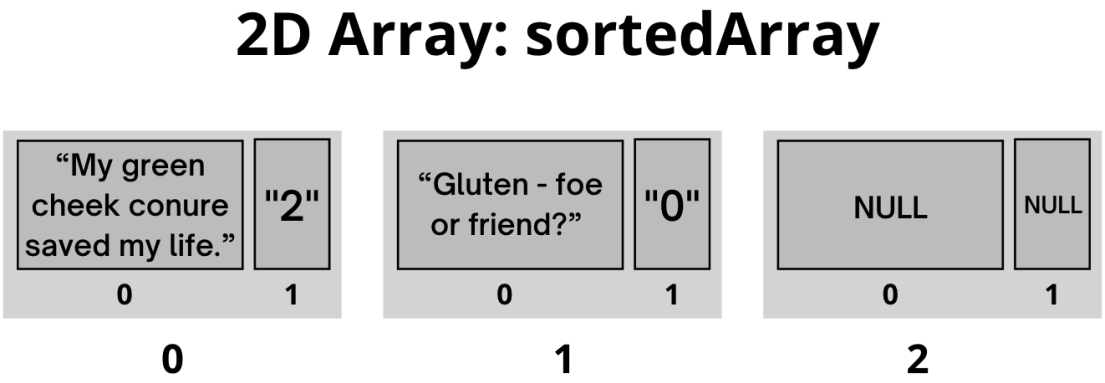



**Fig. 10 - sortedArray Insertion 2**

<u>Inserting third element:</u>

Here comes the tricky part. Because 1 (the priority number of the third element in “result”) is greater than 0 (the priority number of the second element in “sortedArray”), we have to swap the two. Here’s how we do it:

1. Copy over index 1 of “sortedArray” so that it is also the value of index 2 of “sortedArray”:

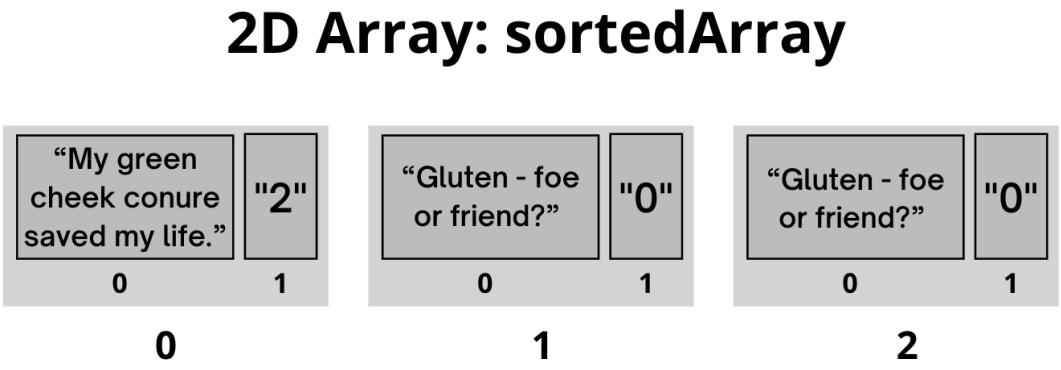



**Fig. 11 - sortedArray Insertion 3 pt. 1**

2. Insert element 3 of “result” into the second position of sortedArray, overriding the values that are already there:

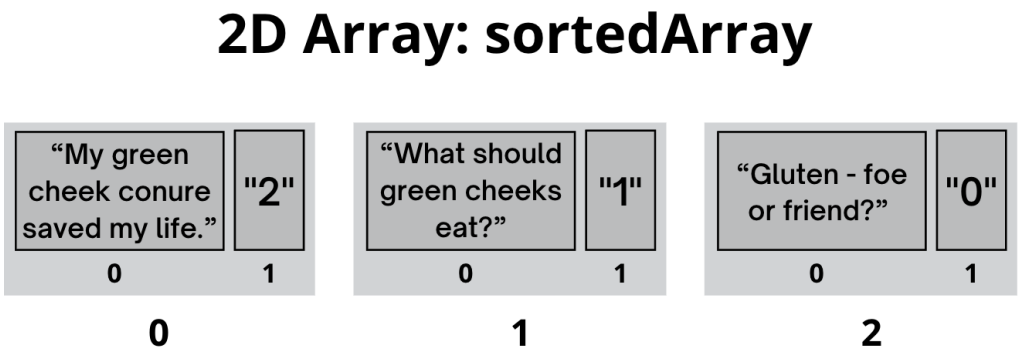



**Fig. 12 - sortedArray Insertion 3 pt. 2**

And there you have it, the 2D array is now sorted from highest to lowest priority numbers! This is a fairly simple process when you have 3 elements, but imagine having 10, 50, or even 1 million elements to sort through! Thankfully the computers are much more efficient than us, but it is our job to program the algorithms to be that way.

Data structures are a very important consideration when it comes to the efficiency of algorithms. Let’s now see how this algorithm works with a priority queue of nodes!

**Algorithm with Priority Queue of Nodes**

**Step 1** of the algorithm with a priority queue is the same as the array: Set the topic, articles, and store the articles in an array.

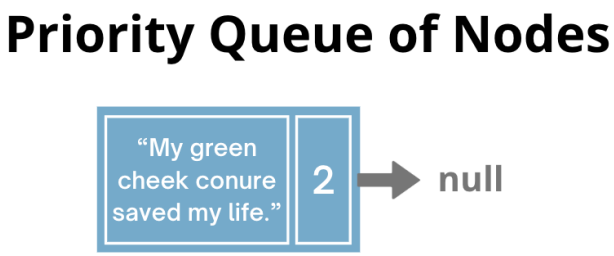
Topic: “Facts anyone with green cheek conures needs to know."


**Fig. 12 - Array: Article**

Step 2: Compare each of the article names to the topic, assign a priority number, and store each article and priority as a Node.

With a priority queue, this step also automatically orders the nodes by connecting pointers to each other in order to showcase the order.

Creating & enqueuing Node 1:




Notice anything different between these node entries vs. the array entries? That’s right, the priority number is not stored as a String! This is because when I created the values for the node, I was able to customize them to be whatever data type I wanted.

**Fig. 13 - Enqueuing Node 1**

Creating & enqueuing Node 2:

Because 0 is less than 2, the priority queue adds the new node right at the front. This may seem like we are ordering in ascending order, but remember, we are dequeuing from the end, which means when we return the values to the user, it will be in descending order.


**Fig. 14 - Enqueuing Node 2**

Creating & enqueuing Node 2:

When a node is first created, the pointer is always set to null, because you have to manually point it to something.

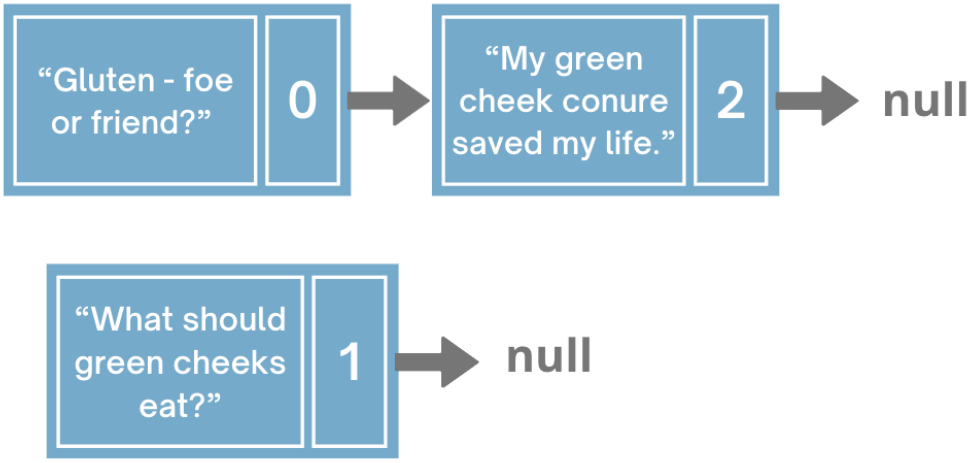



**Fig. 14 - Enqueuing Node 3 pt. 1**

In the last step, node 2 was pointed to node 1 because of the order of priority numbers. In this case, node 3 will need to be inserted between node 1 and 2. How do we do this? We don’t have to overwrite anything like we did with arrays. Instead, we just move the pointers to connect to different nodes:

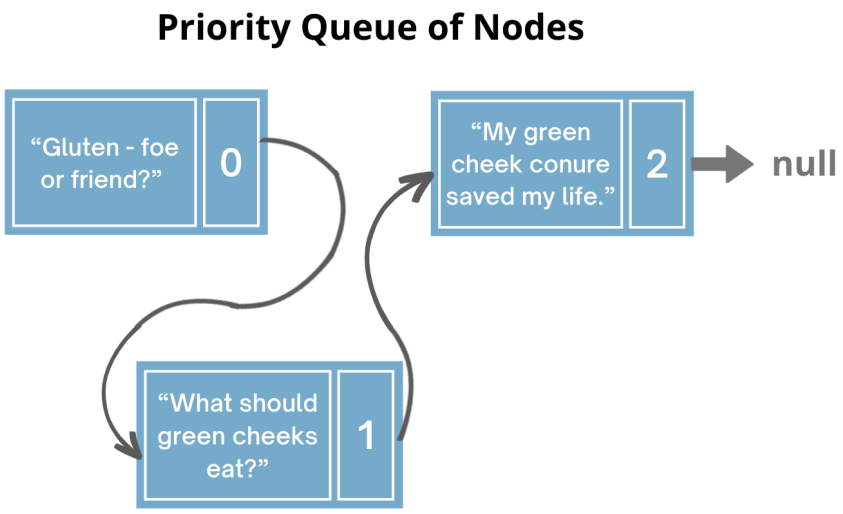



**Fig. 15 - Enqueuing Node 3 pt. 2**

And here is the final result!

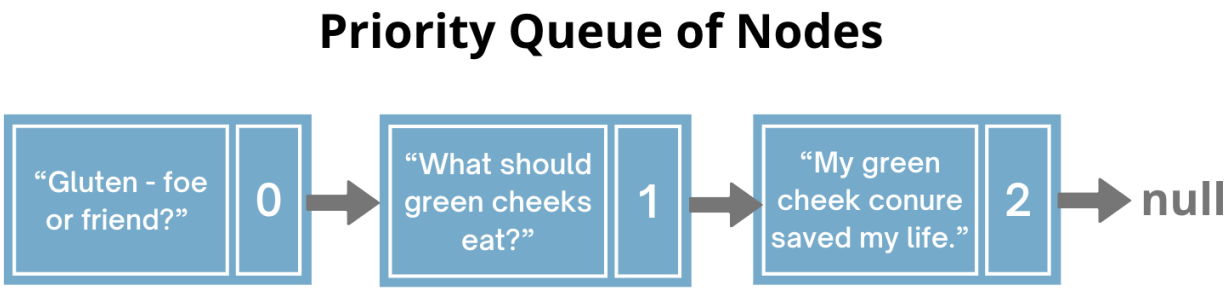



**Fig. 15 - Enqueuing Node 3 pt. 3**

Now that we've walked through both examples, you may be thinking: *"You keep saying that data structures are chosen based on the kind of data and how we want to implement it, but what's actually better in this case?"* Let's talk about it!

**Which Data Structure is More Efficient for this Algorithm?**

A very clear way to compare the efficiency of algorithms is to calculate its **time complexity**, which is the number of steps an algorithm needs to achieve its goal. To get the best idea of

overall efficiency, we calculate the time complexities for the best, average, and worst cases of the algorithm.

Time Complexity of Algorithm with Arrays

Note: For the sake of calculating the time complexity of the algorithm itself, I will be excluding the user input aspect from the calculation and assume the topic and articles are hard coded.

➔ Best case: Ω(n) - Only one article in best case

Let's go method by method to calculate the best case for the algorithm with arrays.

Main Method - Ω(n)

- Creates and assigns 3 variables, topic, articles[], result[][], sortedArray[][] - Ω(1)
- Has 5 print statement: Ω(1)
- Calls articlePriority() once for every article: In the best case, there is only one article in the Article array, so articlePriority will only be called once - Ω(1) *(not including articlePriority)*
- Calls insertSort() once - Ω(1)
- Iterates through articles to print all articles and priority values for user: Ω(1)

Article Priority - Ω(1)

- In the best case scenario, the arrays topicWords[] and articleWords[] both only have one element where the topic and articles are all only one word.
- This means that the nested while loops will both only run once, and priority, j, and i will only be incremented once.
- The return statement only occurs once

Therefore articlePriority() is Ω(1).

insertSort() - Ω(n)

- Manages the control flow of add(), calls it once for article - Ω(n)
- Returns sortedArray - Ω(1)

add() - Ω(1)

- Creates and assigns j - Ω(1)
- In the best case, there is only 1 article, so the while loop won't run the first time, and insertSort() won't call add() again so the while loop will lever run.
- Assigns indexes of sorted array - Ω(1)
- Returns sortedArray - Ω(1)

Overall, the best case for the algorithm with arrays is Ω(1). This is because in the best case, there will be only one article given, and the topic and article name will both only be one word. It could be argued that this is Ω(n), because the time complexity is still dependent on the number of elements in the array "articles[]", however, in this case n = 1, therefore, the best case is Ω(1).

➔ Worst case: O(n^2) - Array of articles is in ascending order

Main method: O(n)

- Creates and assigns topic, articles[], result, and sortedArray: O(1)
- Calls articlePriority() once for every article: O(n)
- Calls insertSort() once: O(1)
- Iterates through articles to print for user: O(n)

articlePriority(): O(nm)

- Creates and assigns topicWords, articleWords, and priority: O(1)
- The first while loop runs for the amount of words in the topic: O(m)
    - Despite this being the worst case, we don't know what the maximum number of words in an article could be, and it is not related in any way to n. If I think of the absolute worst case, the topic name would be the maximum number of words until memory would run out, so I assign m to whatever value that is.

- The second while loop runs for the amount of words in the article: O(k)
    - Because this is the worst case, I am assuming k to be the maximum number of words possible for each article name before memory runs out.
- Return priority: O(1)

insertSort(): O(n)

- Calls add() once for every article: O(n)
- Returns sortedArray: 1

add(): n^2 + n + n = 2n +n^2 → O(n^2)

- The while loop will run n^2 times because the worst case is where the array of articles is in ascending order, where our target result is to be in descending order.
    - In this case, all of the entries will need to be iterated through n times, and they will all also have to be shifted n times: O(n^2)
- Updating sortedArray values: O(n)
- Return sortedArray: O(n)

Overall the worst case for this algorithm with arrays is O(n^2*m). I'm not sure if it makes sense to keep m as part of the time complexity, however, as this is the worst case, it does make an immensely large impact. I dropped k from the final time complexity because if we are assuming m is large enough to fill up memory to capacity, then multiplying by k doesn't change anything. n^2 is already quadratic, and multiplying it by a huge number essentially makes it completely linear vertically for a great deal of time, so multiplying k wouldn't change much about that.

In my research, I found that the maximum number of characters in a String in Java is 2,147,483,647 characters, however this is a limitation on Java indexing and not memory space, and I don't know enough about Java memory to be able to estimate a number that would max it out (*Java string max size - javatpoint*). Overall, the absolute worst case for this algorithm with arrays is: O(n^2*m).

➔ Average case: Θ(n)

I concluded that the worst case for this algorithm with arrays is some extreme variation of O(n^2) such as O(n^2*m). This would practically never be the case, however, because the value of m would have to be so long to fill up all the memory that no one would input that much data into this algorithm. The only exception I can think of is a bot trying to crash my program.

I also concluded that the best case for this algorithm with arrays is Ω(1) when only one article is provided. However, this would almost never be the case because if only one article is provided, this algorithm does nothing for the user, and it could almost always be written off as an error on the user's part.

Additionally, I know that the next best case is where the array has n values already sorted in descending order, and the worst case is where the array is sorted in ascending order. By this logic, it would be reasonable to assume that the average case is an array is half sorted ascending, and half sorted descending. There also exists randomly sorted arrays with the same time complexity, but for the sake of this estimated derivation, let us consider the first case. If the array is exactly half worst case, and exactly half best case, then it would be exactly in between 1 and n^2, which is n.

Let us consider that the average case through actual derivation is actually a little closer to constant time. It would be reasonable to assume that the average case has to be *at least* n, because if there exists even 1 case where the time complexity is Θ(n) and all the rest are constant time, then the time complexity will be Θ(n) because all the 1s will be considered coefficients. Therefore the average case for this algorithm with arrays is Θ(n).

Time Complexity of Algorithm with Priority Queue of Nodes

Note: *For the sake of calculating the time complexity of the algorithm itself, I will calculate the while loop to showcase dequeue (including dequeue), but exclude it from the overall time complexity as it is not necessary for the algorithm or to return the information to the user.

➔ Best case: In the best case, there is only 1 article - Ω(1)

Main method:

- Creates and assigns queue, topic, articles and testArticles - Ω(1)
- Has 10 print statements - Ω(1)
- Calls addSorted() for every articles: Ω(1)
- Calls articlePriority() for every articles: Ω(1)
- Calls peak() once Ω(1);
- While loop to print until dequeue until it is empty: Ω(1)
    - Calls printQueue(): Ω(1)

addSorted: Ω(1)

- Calls enqueue once: Ω(1)

Enqueue(): Ω(1)

- Creates 1 node and increments size once: Ω(1)

articlePriority(): Ω(1)

- In the best case scenario, the arrays topicWords[] and articleWords[] both only have one element where the topic and articles are all only one word : Ω(1)
- This means that the nested while loops will both only run once, and priority, j, and i will only be incremented once : Ω(1)
- The return statement only occurs once : Ω(1)

peek(): Ω(1)

- Calls getEntry(): Ω(1)

printQueue(): Ω(1)

- Prints getEntry() and getPriority(): Ω(1)

Because the best case is when there is only 1 article, no iterations in terms of n occur, therefore the best case is Ω(1).

➔ Worst case:

Main method: O(n) *(while loop that calls dequeue and printQueue not taken into consideration*)*

- Creates and assigns queue, topic,  articles and testArticles - O(1)
- Has 10 print statements - O(1)
- Calls addSorted() for every articles: O(n)
- Calls articlePriority() for every articles: O(n)
- Calls peak() once O(1);
- While loop to print until dequeue until it is empty: O(n^2)
    - Runs n times and calls dequeue() and printQueue() each time
        - Calls printQueue(): O(n)
        - Calls dequeue(): (On)

addSorted(): O(n)

- Goes through the while loop n times to find the spot where the the node needs to be connected O(n)
- Creates node, sets connections, and increments size: O(1)

articlePriority: O(nm)

- Both the array and priority queue call the same articlePriority method in the Article class, so the complexities are the same

peek(): O(1)

- Only calls getEntry(): O(1)
- printQueue(): n
    - Gets and prints nodeAt for all articles: O(2n)
- dequeue(): O(n)
    - Traverses through to get endNode: O(n)
    - Sets deleted entry equal to endNode and sets endNode and connections to null: O(1)

The overall worst case time complexity for the priority queue of nodes is O(nm). The reason it is significantly better (if we don't take m into consideration because of its extreme unlikeliness), is because doing insert sort with the add() method for the arrays was n^2. This is because it was called n times by insertSort() for each article, and within each time, it had to shift elements over n indexes as the worst case is where the array of articles is sorted backwards.

The priority queue of nodes saves much more time because it doesn't have to shift over entries to make room for the new entry, it has the ability to input a new node anywhere just by changing the pointers.

➔ Average case:

If we think about it, the best case is when there is only one article, so all we only have to do is create the startNode. The second to best case is when the articles are already sorted, and the worst case is every single other case.

For this algorithm using priority queues of nodes, there are only two cases, so the average case would then also be Θ(n). If we took the absolute worst case to be O(nm), then the average case would still be Θ(n) because the article name lengths would be average, and we wouldn't have to take m into account. Therefore, the average case of this algorithm with a priority queue of nodes is Θ(n).

However, efficiency in time is not the only component to be taken into consideration when comparing two programs.

**Which Data Structure is Better Overall?**

Let's review some pros and cons of arrays and priority queues of nodes for this particular algorithm.

| | **Arrays** | **Priority Queue of Nodes** |
|---|---|---|

| | | |
|---|---|---|
| **Pros** | ➔ Easy to access individual information quickly & directly<br>➔ Easy to understand<br>➔ Built in labels (indexes) | ➔ Easy to adjust size<br>➔ Insertions and deletions are easy and fast<br>➔ Automatically orders entries<br>➔ Easy to customize nodes |
| **Cons** | ➔ Can't adjust size<br>➔ Insertions and deletions are difficult and time consuming<br>➔ Can only store information with same data types | ➔ Can't access information of nodes individually, have to traverse<br>➔ May not be applicable to use in many situations |

Overall, there are pros and cons to both arrays and a priority queue of nodes, but after coding the same algorithm using both data structures, I would say that the priority queue of nodes is the better option overall. Not only did it decrease the average time complexity from Θ(n^2) to Θ(n), it was also much easier to code, and much more readable.

As seen above, the majority of the characteristics needed to make this algorithm successful are found in the priority queue of nodes. The easy insertions and deletions are key to the structure of a priority queue. Even though information isn't directly accessible like it is in arrays, traversing through the nodes every time to gather necessary entries was much easier, and overall more efficient, than having to traverse and swap entries in the array to insert.

## Conclusion

In conclusion, I have programmed an algorithm to order articles from highest to lowest in terms of relevancy to a given topic with two different data structures: Arrays and a priority queue of nodes. We started from the beginning to understand what data structures are and why they are useful. Additionally, we walked through examples of the algorithm with each data structure, and analyzed which data structure is more fitting to this algorithm through time complexity and overall characteristics native to arrays and priority queues of nodes. Overall, I feel as though the skills I learned throughout this project will be incredibly useful for me in the job market, and I am excited to continue using them! Who knew organizing articles could teach me so much?

**Works Cited**

GeeksforGeeks. (n.d.). *Data Structures*. GeeksforGeeks. Retrieved from https://www.geeksforgeeks.org/data-structures/

GeeksforGeeks. (n.d.). *Applications, Advantages and Disadvantages of Array*. GeeksforGeeks. Retrieved from https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-array-data-structure/

*Java string max size - javatpoint*. www.javatpoint.com. (n.d.). Retrieved from https://www.javatpoint.com/java-string-max-size#:~:text=Therefore%2C%20the%20maximum%20length%20of,length%20of%202%2C147%2C483%2C647%20characters%2C%20theoretically.