PC 108

Data Structure and Algorithm

"Don't try to learn many things quickly, you won't learn anything, it'll just waste your time."

- Mosh 2021

Description: PC 108 - Data Structure and Algorithm focus on the understanding of how the data structure works and how this affects our application. At the same time, it also teaches how to make an efficient algorithm. This also gives the students the experience how to analyze an algorithm

1. **Data Structures and Algorithm Overview**

   Data Structures - are a specialized means of organizing and storing data in computers in such a way that we can perform operations on the stored data more efficiently. Data structures have a wide and diverse scope of usage across the fields of Computer Science and Software Engineering.

   - In short this is the way you store your data by any means like how you store your value in a variable.

   Algorithm - a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

   Why we need to learn Data Structures and Algorithm?

   As per description, we already know how important they are in terms of how the programmer code. Data structures and algorithm focuses on application performance. The way you structure your data and the way you organize your code will affect how your application perform.

   Just imagine this, Facebook has millions of posts available, what if they load all post in your apps one time? Do you think any of that post will show up in your apps? Or it just crashes your apps? That's why its important to know how to structure your data for application performance.

2. **Big O Notation**

   Before we dive into the data structure, you need to know about Big O Notation first to understand later how we can measure the performance of our codes

“Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity”
— Wikipedia’s definition of Big O notation

Basically, Big O Notation use to describe the performance of an algorithm which help us to determine if this algorithm is scalable or not.

Here are some common **Growth Rates** in data structure and algorithm

The growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.

1. Constant or O(1)

Here is a sample code of how **O(1)** can be represented

```
    static void PrintName(string[] names)
    {
        Console.WriteLine(names[0]);
    }
}
```

This is a method that print name, here you can see that it only print the first value of an array and nothing else.

I have another scenario here, can you guess what’s the runtime complexity for this?

```
static void PrintName(string[] names)
{
    Console.WriteLine(names[0]);
    Console.WriteLine(names[1]);
}
```

If you guess **O(2)** then you’re correct. But since they are running in parallel action, we can consider the complexity as **O(1).** So no matter how many lines we deal as long as it just access direct value then we consider that as **O(1).**

2. Linear or O(n)

We have another scenario here.

```
static void PrintNames(string[] names)
{
    for(int i=0; i < names.Length; i++)
    {
        Console.WriteLine(names[i]);
    }
}
```

This is a method that print all the names available in the parameter. The time complexity of this method will be **O(n)** where ***n*** is the size of the array. As the n size grow, it also grows the cost of this algorithm.

We have this method with some additional codes, what do you think is the complexity of this method?

```
static void PrintNames(string[] names)
{
    Console.WriteLine(names[0]);
    for(int i=0; i < names.Length; i++)
    {
        Console.WriteLine(names[i]);
    }
    Console.WriteLine(names[1]);
}
```

We break this down to this

```
static void PrintNames(string[] names)
{
    Console.WriteLine(names[0]); // O(1)
    for(int i=0; i < names.Length; i++) // O(n)
    {
        Console.WriteLine(names[i]);
    }
    Console.WriteLine(names[1]);// O(1)
}
```

See the green color, that's the complexity per line. So if we total that, it will be **O(2 + n),** 2 because we have two **O(1)**, and one **O(n)**. However, that two **O(1)** don't really matter since we are now depending on the **O(n)** as our bases of the complexity, so we can simplify this to **O(n)** as the complexity of this method.

Another scenario here:

```
static void DoublePrintName(string[] names)
{
    for(int i=0; i < names.Length; i++) // O(n)
    {
        Console.WriteLine(names[i]);
    }
    for(int i=0; i < names.Length; i++) // O(n)
    {
        Console.WriteLine(names[i]);
    }
}
```

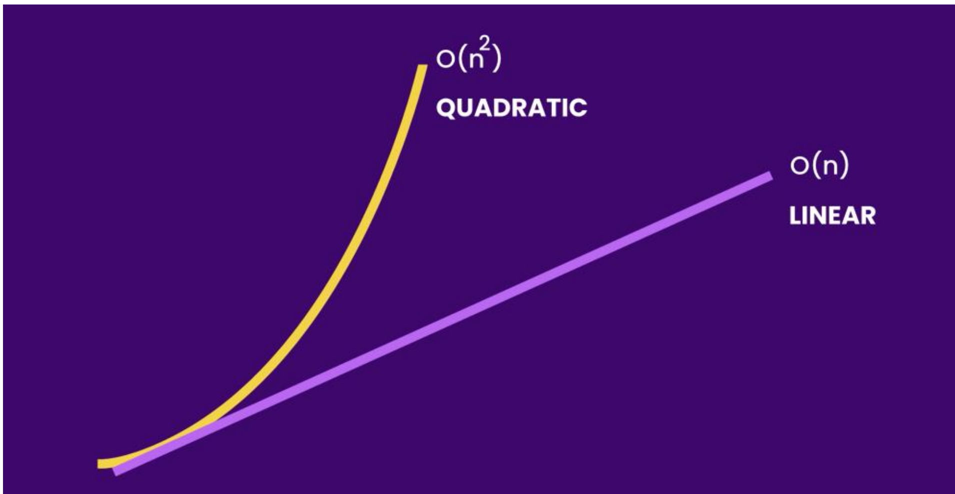
In this case we have two **O(n)**, and this become **O(n + n)** that can also be simplified to **O(n)** since they are running in parallel.

3. Quadratic or O(n^2)

Now we have nested loops scenario here

```
static void NestedPrintNames(string[] names)
{
    for(int i=0; i < names.Length; i++) // O(n)
    {
        for(int j=0; j < names.Length; j++) // O(n)
        {
            Console.WriteLine(names[j]);
        }
    }

}
```

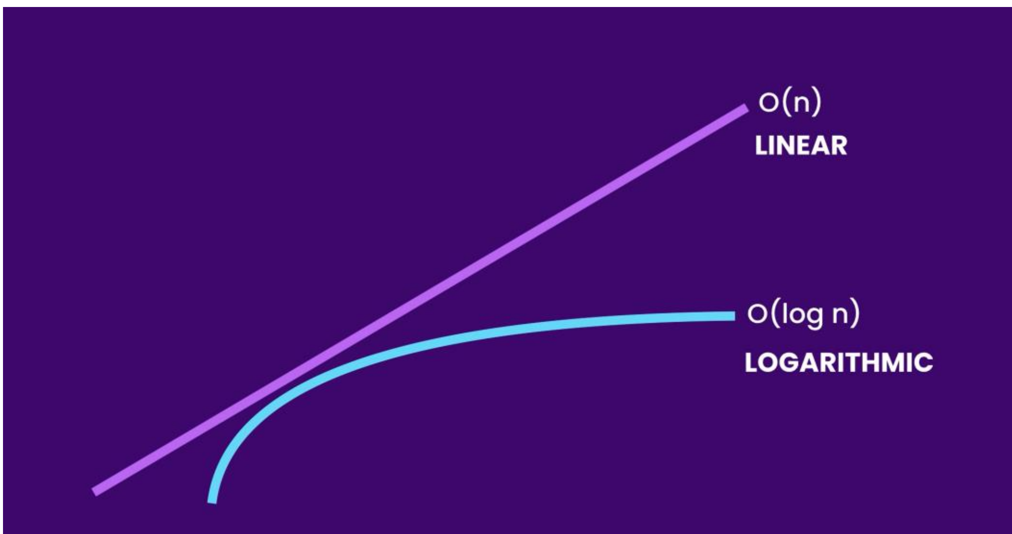
There is a loop inside a loop or what we called a nested loop, what do you think is the complexity here? So it should be **O(n * n)** or in short **O(n^2)** or Big O of ***n*** squared. Now what this means? This diagram below will show how this differ from the **O(n)**.



This means that the algorithm runs with quadratic complexity will slower than the algorithm running in linear.

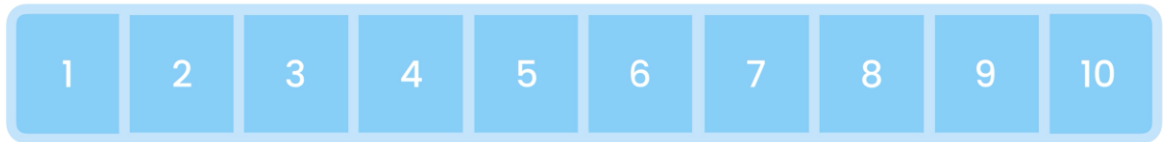
4. Logarithmic or O(log n)

We might not be able to tackle this example with actual coding since this require an advance coding to show you how this is measured. But to understand this, we can compare this to **O(n)**,

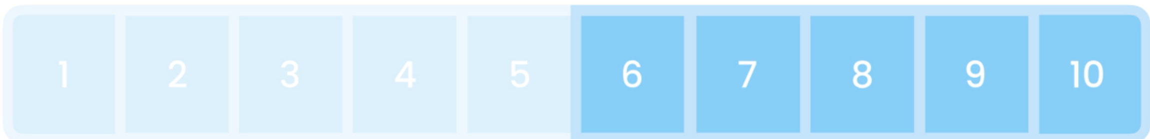


This complexity is much faster compared to **O(n)** because on some point it will trim down the number of processes somewhere in the middle.

For example:

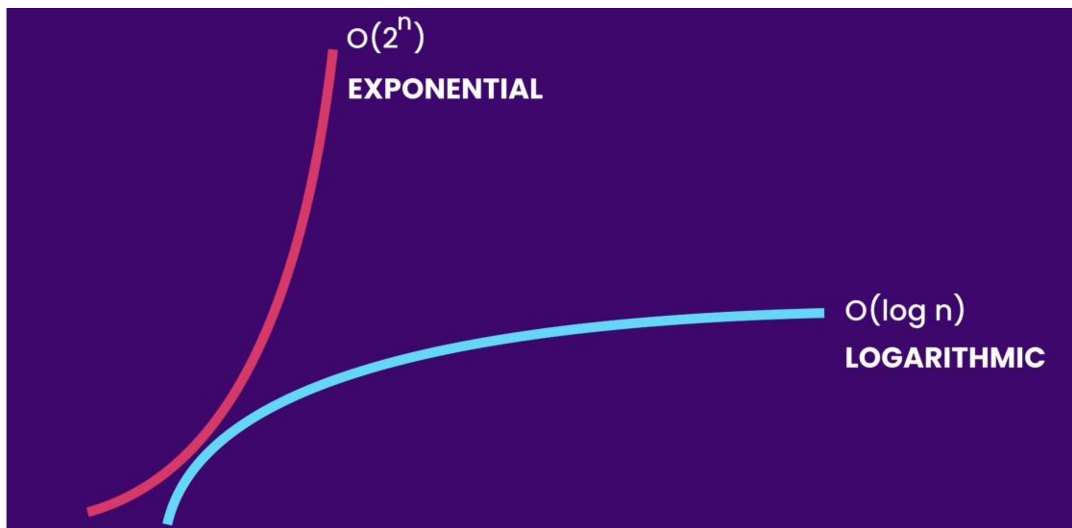


Imagine this array with numbers and I want to find the number 8 from this array, using linear or **O(n)**, you need to start from the beginning and compare if the first number is equal to the number you are looking for, so that's how linear works. But in logarithmic or **O(log n)**, we create an algorithm that will trim down the process. How? We trim it to half.

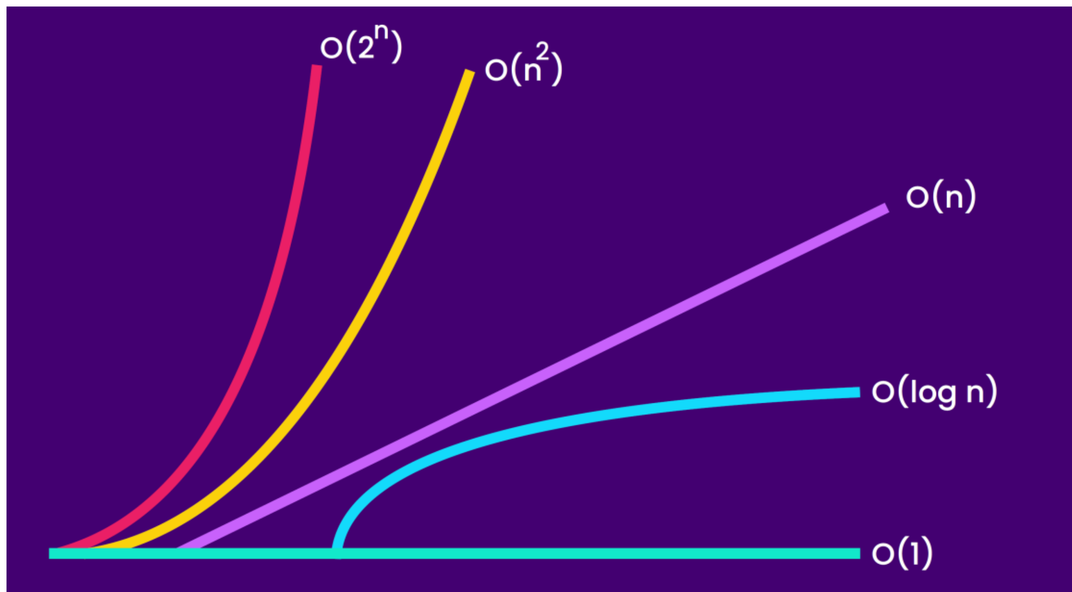


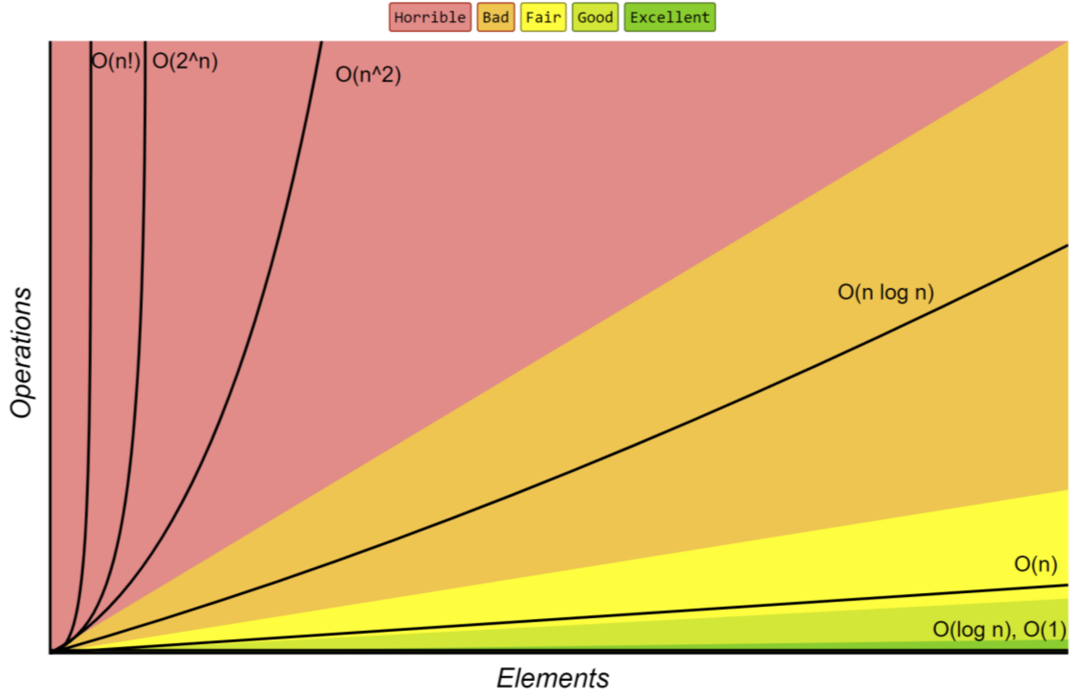
Is the second half first number is greater than or equal to the number we are looking? In this case we have 6 which means we don't need the first half as it will not have the number we are looking for. This means we already trimmed down the process by half. That's how the logarithmic measurement in theory.
As I said we might not be able to do an actual coding for this but will see if you have created this algorithm in the future activities.

5. **Exponential or O(2^n)**
   Exponential growth is opposite for logarithmic growth, as the input size grows, it requires more time to run the program. We also don't have code example for this.



In summary for all growth, we have these chart to compare them all. Note that these are not only growth

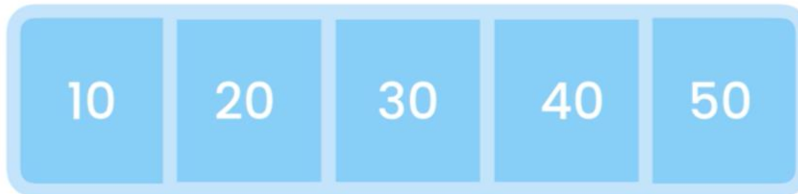




Code References: https://code.sololearn.com/c3SWUo4PqlkE

3. **Data Structure – Arrays**

What are arrays? Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
We all know that values are stored in memory, imagine this image below as computer memory and we store the value in this memory.



This is how arrays works. You declare how many allocations you want to prepare for your values.

To declare an array, you need to specify what is the size of your array.

In C#, this is how to declare an array with size of 5.

```
string[] pets = new string[5];
```

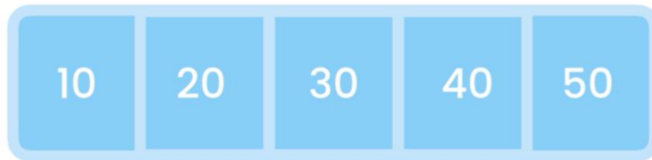
To add or access value inside the array, you can also do this

```
string[] pets = new string[5];
pets[0] = "Dog";
pets[1] = "Cat";
pets[2] = "Rabbit";

Console.WriteLine(pets[2]);
```

You need to specify what index of the memory allocation you want to add the value or access the value. Take note that array indexes start from ***0***.

Note that this course is not related to any programming languages, so you may use whatever programming language that you are comfortable with, but you need to do your research on how to do it in other languages.

Based on this array, how can we get the third value (30).



So to get the third value, you should have ***numbers[2]*** since the index starts with ***0***. What would be the complexity when we access the value for each location? It should be constant or **O(1)**. Why? Because we directly access on the specify location using the index.

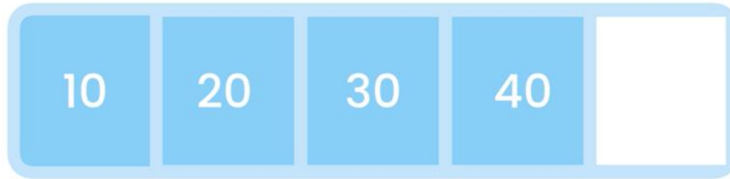
Here is a scenario that we need to create an algorithm to solve the problem.

Since our array size is only 5 and we want to add another item, what will we do to add more space for us to store? In array we should have fix size and we can't do anything to add more space on that. To increase the size, we need to declare again with more size. For instance, I want to add more numbers in our array.

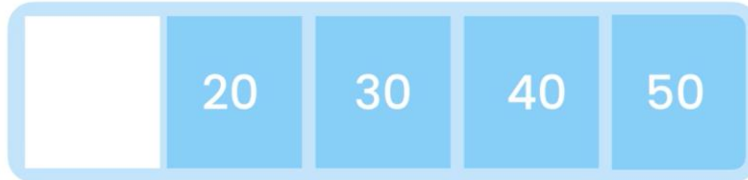
```
int[] numbers = new int[5];

int[] moreNumbers = new int[10]
```

We need to declare another array to handle that more space. We need to copy the value from smaller size array to larger size. The runtime complexity of this process would be linear or **O(n).** It is because we will go each item and copy it to the newly created larger size array.

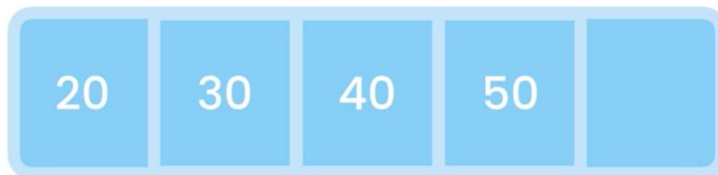
What if we need to remove item from array? The best case would be you remove the last item. It means that the runtime complexity is **O(1)** because other item isn't affected by the changes.



And the worst case in removing item from array will be removing it from the beginning



This is because you need to move all the items from the right to the ***0*** index and the runtime complexity would be **O(n)**



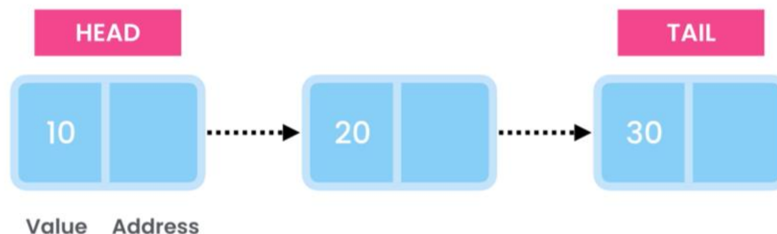
In summary, to create a dynamic array, we need to do the following

1. Create a class for dynamic array with initial size
2. Create an insert method to insert item, and when the size is already full, we need to create a new size probably twice the initial value and copy the previous value to the newly created array.
3. Create a removeAt method and pass the index you want to remove, if the item is not from the last index, you need to transfer the rest to from the removed item.

This is just an algorithm on what we can do to solve the problem, we don't have codes yet at this point. C# has already an implementation for dynamic array and it is called ***ArrayList***. You can check the sample from this site https://www.tutorialsteacher.com/csharp/csharp-arraylist. You can search dynamic array for your preferred language, or you can create based on that algorithm that we listed above.

4. Data Structure – Linked List

What is LinkedList? A LinkedList is a linear data structure which stores element in the non-contiguous location. The elements in a linked list are linked with each other using pointers. Or in other words, LinkedList consists of nodes where each node contains a data field and a reference(link) to the next node in the list. First node is called Head, and the last node is called Tail.



What will be the complexity for LinkedList?

Lookup By Value – finding value inside the linked list would **O(n)** because you need to look all over the list to find the value you are looking for.
Lookup By Index – unlike array, linked list value will be anywhere else in the memory which means we need again to search for all the value to look for the value, which means it should be **O(n).**
Insert at the beginning – since we have reference of the head, we just move the head to the item we inserted so this means the complexity is **O(1).**
Insert at the end – the same with the head, we also have reference for the end which is the tail, which means we just move the tail to the insert item, so the complexity is **O(1).**
Insert in the middle – since we don't have any specific reference what's in the middle, then you need to find that node where you want to insert so the complexity in **O(n).**
Delete from the beginning – So deleting from the beginning means you just remove the head move the head reference to the next node. Since we already have link to the next node, this means its easy for us to know where the next node is. This means the complexity is **O(1)**.
Delete from the end – We already have reference for the end of the list, which is the tail, but we don't have reference from the previous node, since linked list only have reference for the next node and this means we need to find the head and look for the last item. So the complexity will be **O(n)**.
Delete from the middle – Same with insert in the middle the complexity will be **O(n)**.

Arrays vs **Linked Lists**

Space in memory

1. Static arrays have a fixed sized – which means you just need to allocate what's the needed space.
2. Dynamic arrays grow by 50% to 100% of initial size – this means if you increase the size once you already have extra item from the declared size, other space would be wasted.
3. In other hand, Linked Lists don't waste space, as it grow as what you needed, however linked list should contain value and address, while arrays only contains value, that means linked list need more space.
4. Use arrays if you know ahead of time the total items that you need, or use dynamic array if you have rough estimates on how many items it should be and just increase the size later on.

Time complexity

You can see here in the image how they are differ in time complexity.

	ARRAYS	LINKED LISTS
Lookup		
By Index	O(1)	O(n)
By Value	O(n)	O(n)
Insert		
Beginning/End	O(n)	O(1)
Middle	O(n)	O(n)
Delete		
Beginning	O(n)	O(1)
Middle	O(n)	O(n)
End	O(n)	O(n)

In summary, it depends on what you are trying to solve to know what you need to use. Everything has its own advantages and disadvantages.

5. Data Structure – Stacks

What are stacks? Stack is a special type of collection that stores elements in LIFO style (Last In First Out). This is like a stack of plate, where the last plate you put will be the first one you will

get. Most common use of stack is like the navigation in your browser, where you can only go back or forward. Or the undo/redo in any application.

Operations for Stacks

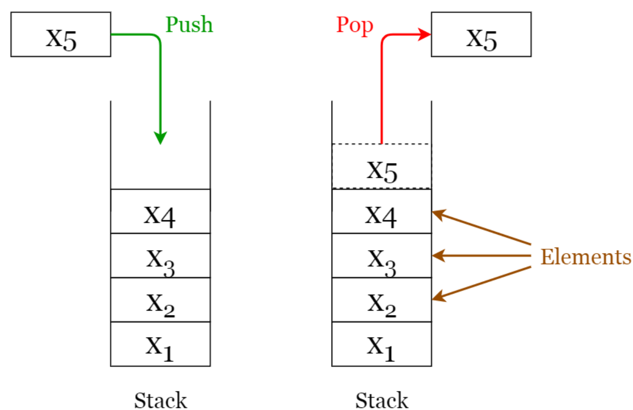
Push – Insert new item in the stack

Pop – Remove and return the top item in stack

Peek – Return the top item without removing

Empty – Return true if stack is empty (depends on the language used)

Count – Return the count of items in the stack (depends on the language used)



Other operation may vary depending on the language you will use. Please check what's the methods in the language you use.

Exercise #1

Problem: Using stack data structure, create a method that will return a reversed string.

Input: Hello World!

Expected Output: !dlroW olleH

Algorithm

1. Create a stack
2. Find a way to get each character and push it to the created stack. In this way we will get the first character to the bottom of the stack and next character should follow.
3. We can declare a variable to handle the reversed string.
4. Since we already have the last character at the top of the stack, we just pop it and the return value will assign it to the variable that we created.
5. Return whatever the value of the variable we created.

You can check how stack implemented here

C++ - https://www.geeksforgeeks.org/stack-in-cpp-stl/

C# - https://www.tutorialsteacher.com/csharp/csharp-stack

Java - https://www.javatpoint.com/java-stack

See my solution in C++ here: https://code.sololearn.com/cXXvjWJF3sV0

6. Data Structure – Queues

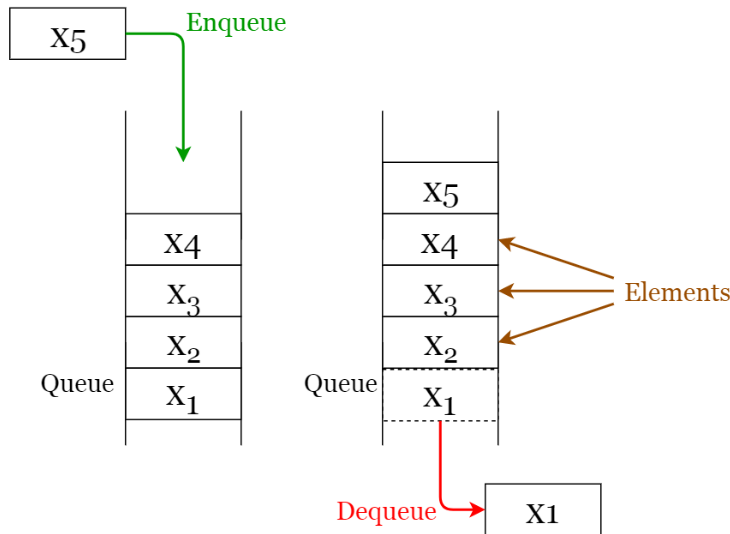
A queue is a FIFO (First In First Out — the element placed at first can be accessed at first) structure which can be commonly found in many programming languages. This structure is named as "queue" because it resembles a real-world queue — people waiting in a queue.

Queue operations

Given below are the 2 basic operations that can be performed on a queue.

Enqueue: Insert an element to the end of the queue.

Dequeue: Delete the element from the beginning of the queue.



Applications of queues

- Used to manage threads in multithreading.
- Used to implement queuing systems (e.g.: priority queues).

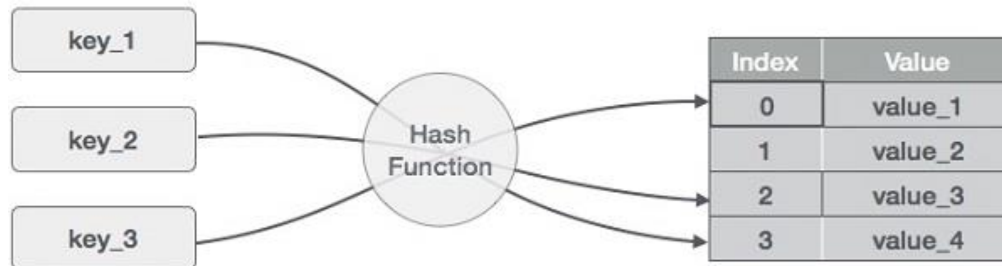
7. Data Structure – Hash Tables

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Here are some implementations using different languages.

IMPLEMENTATIONS

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



Hash Table Complexity

HASH TABLES

Insert	O(1)
Lookup	O(1)
Delete	O(1)

How to do it?

For this example, I will use C# as our language and the reason is that C++ is quite hard to demonstrate, but if you wish to use C++ for this, you can visit this link for reference https://www.javatpoint.com/post/cpp-map. You can also search for other implementation to your preferred language.

Problem: Find employee using employee number.

You have list of employees and using their employee number, you need to find their name.

```
IDictionary<int, string> employees = new Dictionary<int, string>();
employees.Add(9809, "John Doe"); // O(1)
employees.Add(3456, "Josephine Smith"); // O(1)
employees.Add(4512, "Juan dela Cruz"); // O(1)

Console.WriteLine(employees[4512]); //O(1)
Console.WriteLine(employees[9809]); //O(1)
Console.WriteLine(employees[3456]); //O(1)
```

In this code above, we declare a hash table using Dictionary, now you have Key which has int data type, and value with string data type

```
IDictionary<int, string> employees = new Dictionary<int, string>();
```

Now we have also need to add data, first is the Key (9809) which represent as the employee number and the Value (John Doe) as the employee's name.

To get the value using the Key (employee number) you just use it as an index of the array

```
employees[9809]
```

That's how easy to access this value. The time complexity is the same as accessing array using index.

If you want to check all items from the dictionary you just need to use looping to do it.

```
foreach(var emp in employees)
    Console.WriteLine("Key: {0}, Value: {1}", emp.Key, emp.Value);
```

Code Referrence: https://code.sololearn.com/cv4tmKKitlfw

Exercise #2: Convert string value into a key value pair format using hash table data structure. Where key is the character, and the value is how many times this character is repeated.

Input: hello world

Output: { h=1, e=1, l=3, o=2, w=1, r=1, d=1 }

Midterm Exam

Ask your instructor for your midterm exam.

Sorting Algorithm

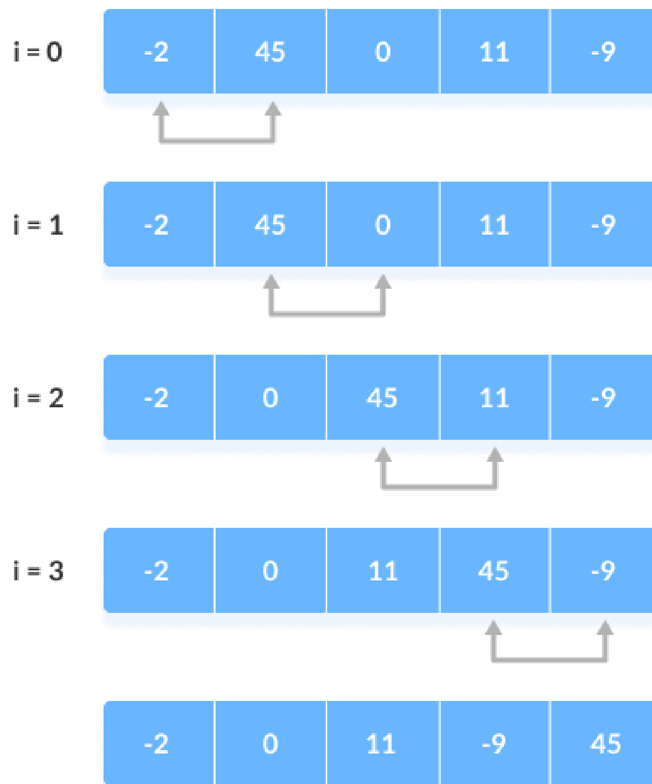
1. **Bubble Sort**

   Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are not in the intended order.

   Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

   Working of Bubble Sort
   Suppose we are trying to sort the elements in ascending order.

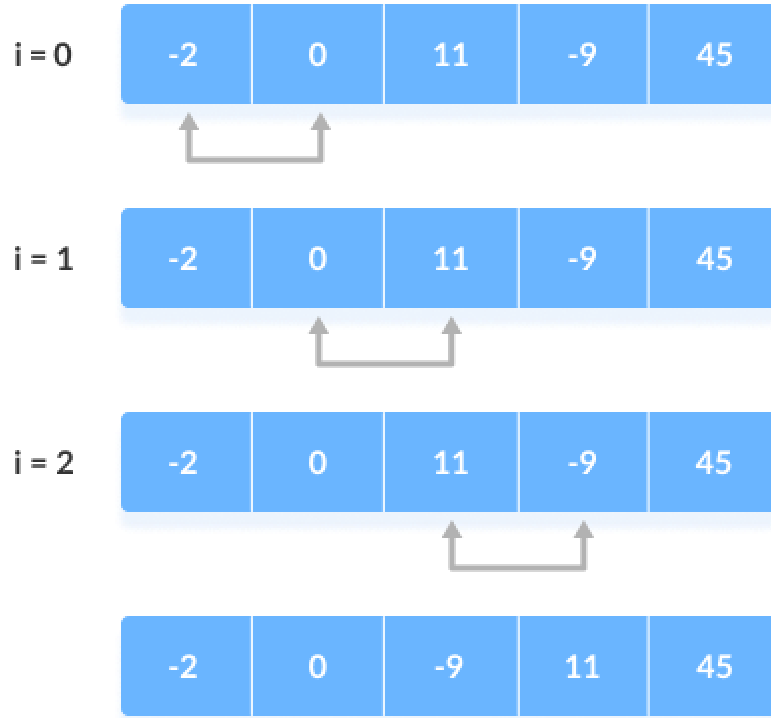
   1. First Iteration (Compare and Swap)
      a. Starting from the first index, compare the first and the second elements.
      b. If the first element is greater than the second element, they are swapped.
      c. Now, compare the second and the third elements. Swap them if they are not in order.
      d. The above process goes on until the last element.



2. Remaining Iteration

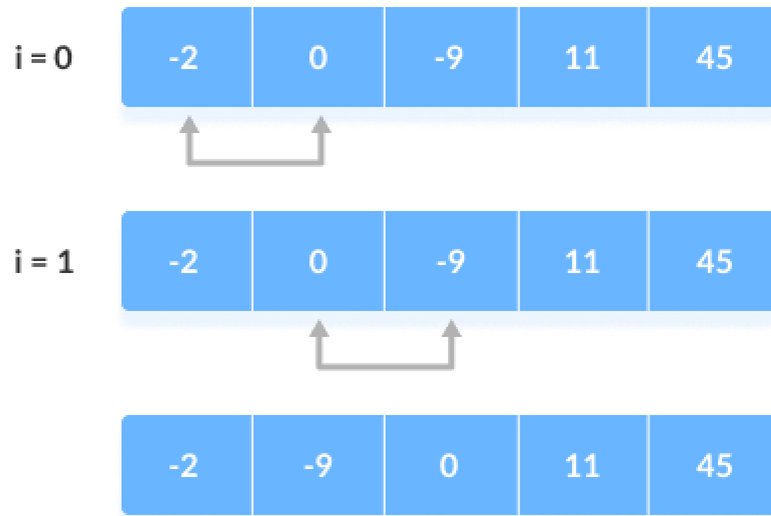
The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.



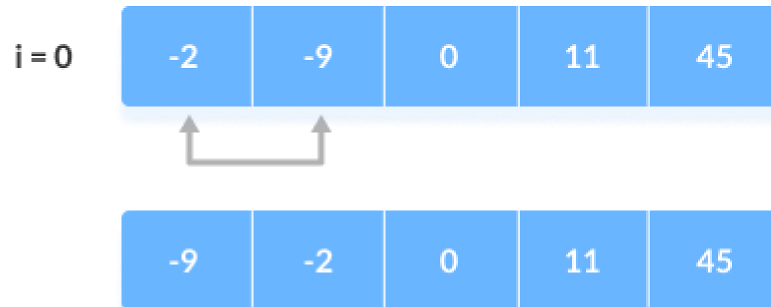
In each iteration, the comparison takes place up to the last unsorted element.

step = 2



The array is sorted when all the unsorted elements are placed at their correct positions.

step = 3



Bubble Sort Algorithm

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

Code for Bubble Sort: https://code.sololearn.com/cvogTAP9iOC7