Beginning

1. **Definition :**

* Data structures is about how data can be stored in different structures.
* Algorithms is about how to solve different problems, often by searching through ( *tìm kiếm* ) and manipulating (*thao tác*) data structures.
* Theory about Data Structures and Algorithms (DSA) help us to use large amounts of data to solve problems efficiently.

1. **What are Data Structures?**

* Data structure is a way to store data.
* We structure data in different ways depending on what data we have, and what we want to do with it.
* If we want to store data about people we are related to, we use a family tree as the data structure.
  + We choose a family tree because we have information about people we are related to and how they are related, and we want an overview -> Easily find a specific family member, several generations back.
* Data structures give us the possibility to manage large amounts of data efficiently for uses such as large databases and internet indexing services.
* Data structures are essential ingredients in creating fast and powerful algorithms => help in managing and organizing data, reduce complexity, and increase efficiency.
* There are 2 different kinds of data structures :
  + Primitive data structures : the basic data structures provided by programming languages to represent single values, such as : integers, floating-point numbers, characters and booleans.
  + Abstract Data Structure : higher-level structures that are built using primitive data types and provide more complex and specialized operations. Include : arrays, linked lists, stacks, queues, trees and graphs.

1. **What are Algorithms?**

* An algorithm is a set of step-by-step instructions(*hướng dẫn*) to solve a given problem or archive a specific goal
* When we talk about algorithms in Computer Science, the step-by-step instructions are written in a programming language, and instead of food ingredients, an algorithm uses data structures.
* Algorithms are fundamental to computer programming as they provide step-by-step instructions for executing(*thực hiện*) tasks.
  + An efficient algorithm can help us to find the solution we are looking for, and to transform(*biến*) a slow program into a faster one.
* Example :
  + Finding the fastest route in a GPS navigation system.
  + Navigating(*điều khiển*) an airplane or a car (cruise control)
  + Finding what users search for (search engine)
  + Sorting, for example sorting movies by rating.

1. **Data Structures together with Algorithms :**

* Data Structures and Algorithms go hand in hand ( *song hành* ).
  + A data structure is not worth much if we cannot search through it or manipulate it efficiently using algorithms
  + The algorithms are not worth much without data structures to work on.
* DSA is about finding efficient ways to store and retrieve(*truy xuất*) data, to perform operations on data, and to solve specific problems.
* By understanding DSA , we can :
  + Decide which data structure or algorithm is the best for a given situation.
  + Make programs that run faster or use less memory.
  + Understand how to approach(*tiếp cận*) complex problems and solve them in a systematic way

1. **Where are data structures and algorithms needed?**

* Used in virtually every software system, from operating systems to web applications:
  + Manage large amounts of data, such as social networks or search engines.
  + Schedule tasks, to decide which task a computer should do first.
  + Plan routes, like in GPS systems to find the shortest path from A to B.
  + Optimize processes, such as arranging tasks so they can be completed as quickly as possible.
  + Solve complex problems: From finding the best way to pack a truck to making a computer ‘learn’ from data
* DSA is fundamental in nearly every part of the software world :
  + Operating Systems
  + Database Systems
  + Web Application
  + Machine Learning
  + Video Games
  + Cryptographic Systems
  + Data Analysis
  + Search Engines

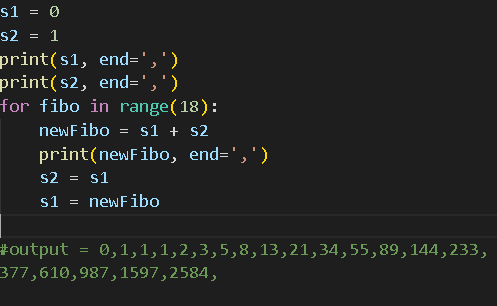
1. **Theory and Terminology :**

| **Algorithm** | **A step-by-step instructions to solve a specific problem** |
| --- | --- |
| **Data Structure** | **A way of organizing data so it can be used efficiently. Include : arrays, linked lists, binary trees** |
| **Time complexity** | **A measure of the amount of time an algorithm takes to run, depending on the amount of data the algorithm is working on** |
| **Space Complexity** | **A measure of the amount of memory an algorithm takes to run, depending on the amount of data the algorithm is working on** |
| **Big O Notation** | **A mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value of infinity.**  **Used to describe the time complexity of an algorithm** |
| **Recursion** | **A programming technique where a function calls itself** |
| **Divide and conquer** | **A method of solving complex problems by breaking them into smaller, more manageable sub-problems, solving the sub-problems, and combining the solutions.**  **Recursion is often used when using this method** |
| **Brute Force** | **A simple and straightforward way an algorithm can work by simply trying all possible solutions and then choosing the best one** |

**A Simple Algorithm**

1. **Fibonacci Numbers:**
   1. **Start with the two first Fibonacci numbers 0 and 1.**
      1. **Add the two previous numbers together to create a new Fibonacci number.**
      2. **Update the value of the two previous numbers.**
2. **Loops and Recursion(*đệ quy*).**
   1. **An implementation of the Fibonacci algorithm above using a for loop.**

* **Two variables to hold the previous two Fibo numbers**
* **A for loop that runs 18 times**
* **Create new Fibo numbers by adding the two previous ones**
* **Print the new Fibo number**
* **Update the variables that hold the previous two Fibo numbers**

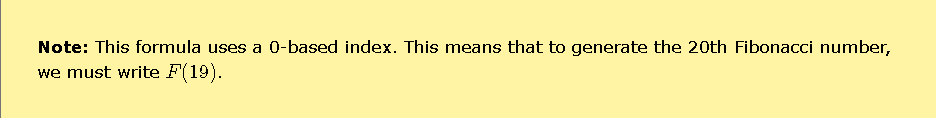
****

* 1. **An implementation of the Fibonacci algorithm above using recursion.( Function calls itself )**
* **Replace the for loop with recursion -> Encapsulate much of the code in a function, and we need the function to call itself to create a new Fibo number as long as the produced number of Fibo numbers is below, or equal to, 19.**

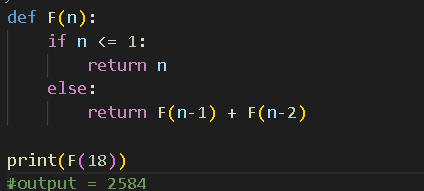
****

* 1. **Finding the** *n***th Fibonacci number using recursion.**
* **The formula :**

***F(n) = F(n - 1) + F(n - 2)***

****

* **We can let the function call itself as long as** *n*  **is less than or equal to, 1.**

****

**Arrays**

**Arrays**

# **DSA Arrays :**

1. Arrays :

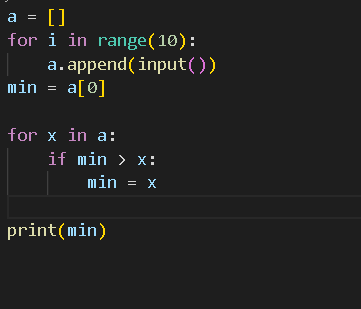
* Is a data structure used to store multiple elements
* Like a list, the first value is 0,the late value is length - 1.

1. Find the lowest value in an array:

* Go through the values in the array one by one.
* Check if the current value is the lowest so far, and if it is , store it.
* After looking at all the values, the stored value will be the lowest of all values in the array.

1. Implementation : -> **Can do with any kind of variables in the array**.

* Create a variable “minVal” and set it equal to the first value of the array.
* Go through every element of the array.
* If the current element has a lower value than “minVal”, -> Updating.
* After looking at all the elements -> the “minVal” contains the lowest value.
* **=>** We call “pseudocode” : is a description of what a program does, using language that is something between human language and a programming language.



**\*\*\*\* Algorithm Time Complexity \*\*\*\***

This method is



# **Bubble Sort :**

# => (nổi bọt/bong bóng) -> **Just sort array that contains int** .

* Is an algorithm that sorts an array from the lowest value to the highest value.
* **How it works**:
  + Go through the array, one value at a time.
  + For each, compare the value with the next value.
  + If the value is higher than the next one, swap the values so that the highest value comes last.
  + Go through the array as many times as there are values in the array.

1. Manual Run Through :

* Start with an unsorted array: [7, 12, 9, 11, 3]
* Look at the two first values-> Compare: [7, 12, 9, 11, 3]
* Done -> Look at the next couple -> Compare : [7, 12, 9, 11, 3]
* Swap them cause 9 < 12 :

[7, 9, 12, 11, 3]

* Look at the next couple -> Compare : [7, 9, 12, 11, 3]
* Swap this couple:

[7, 9, 11, 12, 3]

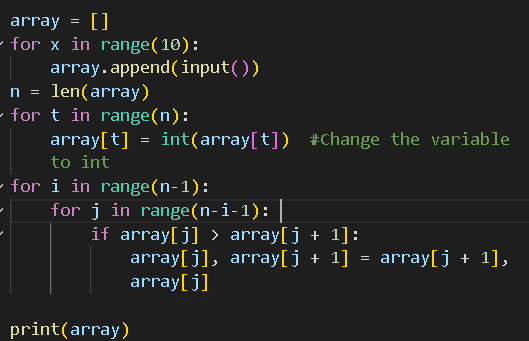
* Look at the next couple -> Compare : [7, 9, 11, 12, 3]
* Swap them :

[7, 9, 11, 3, 12]

=> We need to run through the arrays 4 times, to sort the arrays of 5 values.

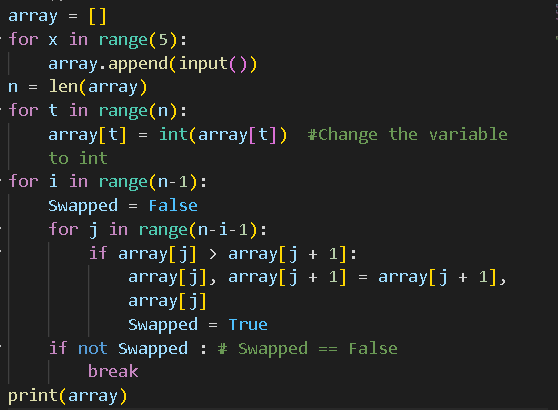
1. Implementation :

* An array with values to sort
* An inner loop that goes through the array and swaps values if the first value is higher than the next value. This loop must loop through one less value each time it runs
* An outer loop that controls how many times the inner loop must run. For an array with n values -> the outer loop must run n - 1 times.



1. Improvement :

* In case the array is almost sorted already, The Bubble Sort algorithm will continue to run, without swapping elements and that is ***not necessary***.



**\*\*\*\* Algorithm Time Complexity \*\*\*\***

This method is

***(for best case, just about***

****

# **Selection Sort :**

* Look through the array again, again and again, moving the next lowest values to the front, until the array is sorted.
* **How is works** :
  + Go through the array to find the lowest value
  + Move the lowest value to the front of the unsorted part of the array.
  + Go through the array again as many times as there are values in the array.

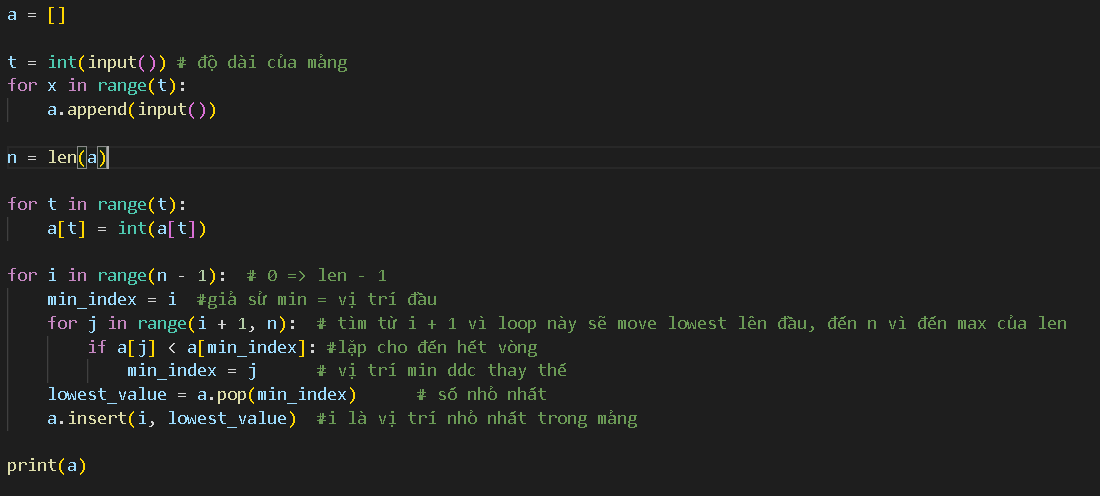
1. Manual Run Through :

* Start with an unsorted array : [7, 12, 9, 11, 3]
* Go through the array, one value at a time -> Find the lowest value : [7, 12, 9, 11, 3]
* Move the lowest value to front of the array : [3, 12, 9, 11, 7]
* Look through the array -> Find the lowest : [3, 7, 12, 9, 11] *7 is already at the front of the array*
* Loop this until the array is sorted : [3, 7, 9, 11, 12]

=>Run through the arrays 4 times to sort the array of 5 values.

1. Implementation :

* An array with values to sort.
* An inner loop that goes through the array, finds the lowest value, moves it to the front of the array. This must loop through one less value each time it runs
* An outer loop that controls how many times the inner loop must run. -> The outer loop must run n - 1 times for an array with n values.

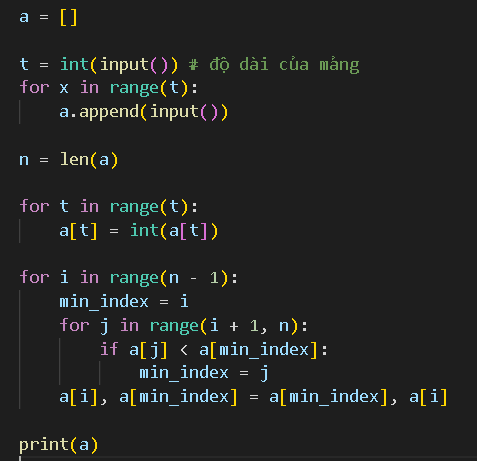


1. Selection Sort Shifting Problem :

* All the elements which are not chosen without the lowest will be shifted twice(1: the lowest removes, 2: the lowest inserts).
* Shifting operations require extra time for the computer to do -> Can be a problem.

1. Solution : Swap Values :

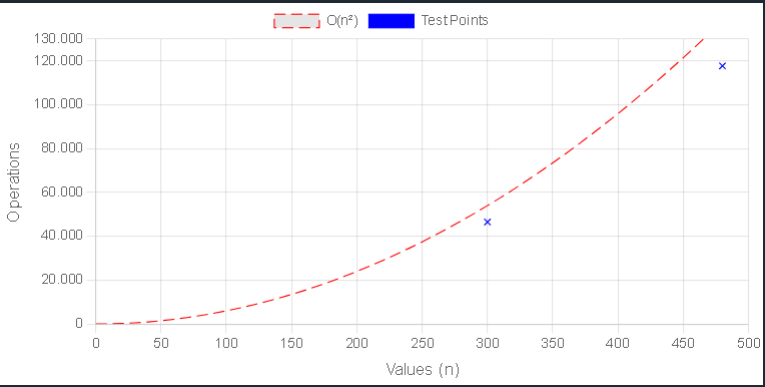
* The lowest value ends up in the correct position, and it does not matter where we put the other value we are swapping with cause it is not sorted yet



**\*\*\*\* Algorithm Time Complexity \*\*\*\***

This method is = **(*for both worst and best cases*)**

(About elements are compared to find the lowest value)



# **DSA Insertion Sort :**

1. Insertion Sort :

* Use one part of the array to hold the sorted values, and the other part of the array to hold va;ues that are not sorted yet.
* **How it works :** 
  + Take the first value from the unsorted part of the array.
  + Move the value into the correct place in the sorted part of the array.
  + Go through the unsorted part of the array again as many times as there are values.

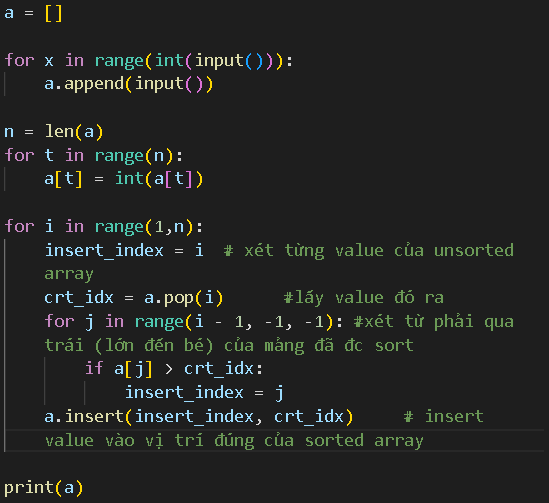
1. Manual Run Through :

* Start with an unsorted array: [7, 12, 9, 11, 3]
* Consider the first value as the initial sorted part of the array. [7, 12, 9, 11, 3]
* The next value should now be moved into the correct position in the sorted part of the array. [7, 12, 9, 11, 3]
* Consider the next value. [7, 12, 9, 11, 3]
* The value 9 must now be moved into the correct position inside the sorted part of the array. [7, 9, 12, 11, 3]
* The next value. [7, 9, 12, 11, 3]
* We move it into the correct position. [7, 9, 11, 12, 3]
* And last. [3, 7, 9, 11, 12]

=> Run through the array 4 times to sort the array of 5 values.

1. Implementation :

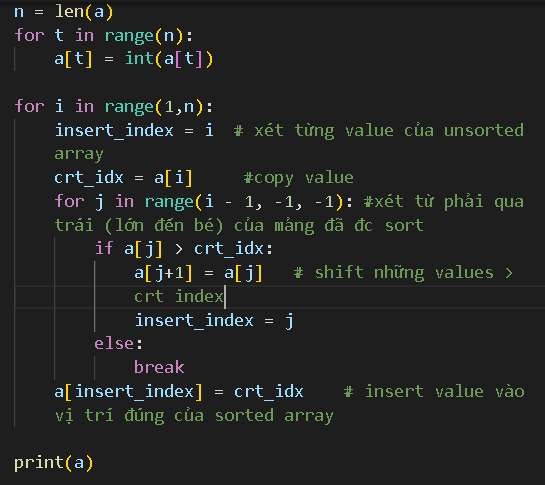
* An array with value to sort
* An outer loop that picks values to be sorted. Must run n - 1 times for an array with n values.
* An inner loop that goes through the sorted part of the array, to find where to insert the value. If the value to be sorted is at index i, the sorted part of the array starts at index 0 and ends at index i - 1.



1. Improvement :

* All the values from the left the value which removes and inserts will be shifted twice. => Take a lot of time especially for an array with many elements.

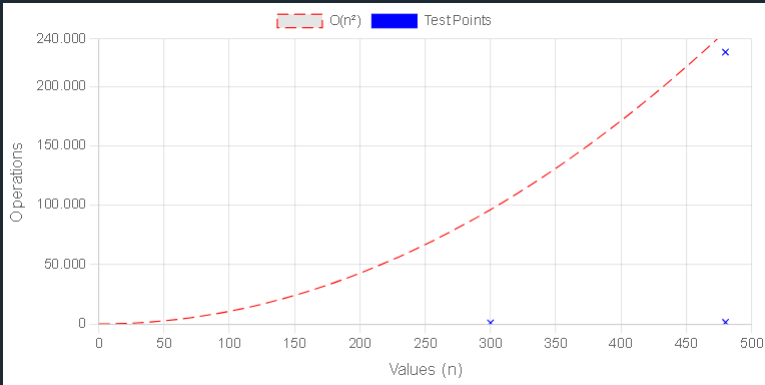
1. Improved Solution :



=> No need to continue comparing values when we have already found the correct place.

**\*\*\*\* Algorithm Time Complexity \*\*\*\***

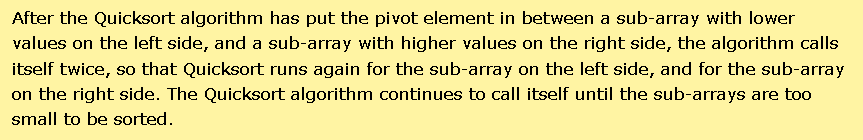
This method is = **(Only for best case)**

****

# **DSA Quicksort: The fastest sorting algorithms.**

* Take an array of values, choose one of the values as the ‘pivot’ element, and move the other values so that lower values are on the left of the pivot element, and higher values are on the right of it.
* The last element of the array is chosen to be the pivot element ( or any element in the array really )
* The Quicksort algorithm does the same operation recursively on the sub-arrays to the left and right side of the pivot element.

\*\*\*\* : **Recursion** is when a function calls itself.



* **How it works :** 
  + Choose a value in the array to be the pivot element.
  + Order the rest of the array so that lower values than the pivot element are on the left and higher values are on the right.
  + Swap the pivot element with the first element of the higher values so that the pivot element lands in between the lower and the higher values.
  + Do the same operation ( recursively ) for the sub-arrays on the left and right side of the pivot element

1. Manual Run Through :

* Start with an unsorted array : [11,9,12,7,3]
* We choose the last value as the pivot element : [11,9,12,7,3]
* The rest of the values in the array are all greater than 3, and must be on the right side of 3 . [3,9,12,7,11]
* Choose the last value as a new pivot element ( 3 is now in the correct position ) [3,9,12,7,11]
* The 7 must be to the left and 12 must be right : [3,9,7,12,11]
* Swap 11 with 12 so that the lower values are on the left and 12 on the right:

[3,9,7,11,12]

* Choose 7 as the pivot element in sub-array [ 9,7], to the left of 11.

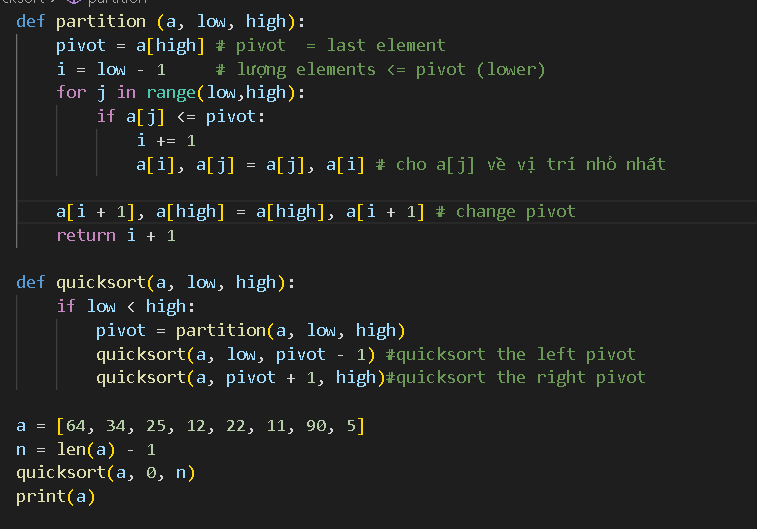
[3,9,7,11,12]

* Sawp 9 with 7.

[3,7,9,11,12]

1. Implementation :

* An array with values to sort
* A quickSort method that calls itself (***recursion***) if the sub-array has a size larger than 1.’
* A partition method that receives a sub-array, moves values around, swaps the pivot element into the sub-array and returns the index where the next split in sub-arrays happens.



**\*\*\*\* Algorithm Time Complexity \*\*\*\***

This method is  **for worst case**

**But actually :**

****

# **DSA Counting Sort :**

* Counting Sort is **fast** when the range of possible values k is smaller than the number of values n.
* **How it works :** 
  + Create a new array for counting how many there are of the different values.
  + Go through the array that needs to be sorted.
  + For each value, count it by increasing the counting array at the corresponding index.(chỉ số tương ứng)
  + After counting the values, go through the counting array to create the sorted array.
  + For each count in the counting array, create the correct number of elements, with values that correspond to the counting array index.
* ***Conditions for Counting Sort* :** 
  + **Integer values** : the number of possible different values k is not too big compared to the number of values n
  + **Non negative values.** These would be outside the counting array
  + **Limited range of values.** The counting array we need for sorting will be larger than the original array we have that needs sorting -> the algorithm becomes ineffective.

1. Manual Run Through :

* Start with an unsorted array :

Array = [2, 3, 0, 2, 3, 2]

* Create another array for counting how many there are of each value . The array has 4 elements to hold values 0 - 3

Array = [2, 3, 0, 2, 3, 2]

CountArray = [0, 0, 0, 0]

* The first element is 2 -> increase the counting array element at index 2.

Array = [2, 3, 0, 2, 3, 2]

Count Array = [0, 0, 1, 0]

* After counting a value, we remove it and count the next value.

Array = [ 3, 0, 2, 3, 2]

CountArray = [0, 0, 1, 1]

* The next value is 0 -> Increase index 0

Array = [0, 2, 3, 2]

CountArray = [1, 0, 1, 1]

* Continue until all values are counted.

Array = [ ]

CountArray = [1, 0, 3, 2]

* Create the elements from the initial array, from the lowest to highest.

Array = [0 ]

CountArray = [0, 0, 3, 2]

* From the counting array, we dont need to create elements with value 1.

Array = [0 ]

CountArray = [0, 0, 3, 2]

* We push 3 elements with value 2 into the end of the array.Also decrease the counting array.

Array = [0, 2, 2, 2 ]

CountArray = [0, 0, 0, 2]

* At last, we add 2 elements with value 3 at the end of the array.

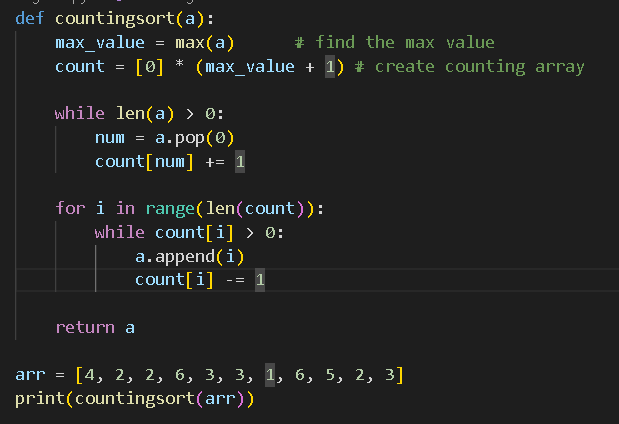
Array = [0, 2, 2, 2, 3, 3]

CountArray = [0, 0, 0, 0]

1. Implementation :

* An array with values to sort.
* A ‘countingSort’ method that receives an array of integers.
* An array inside the method to keep count of the values.
* A loop inside the method that counts and removes values, by incrementing elements in the counting array.
* A loop inside the method that recreates the array by using the counting array, so that the elements appear in the right order.

=> We need to find out what the highest value in the array is -> the counting array can be created with the correct size.

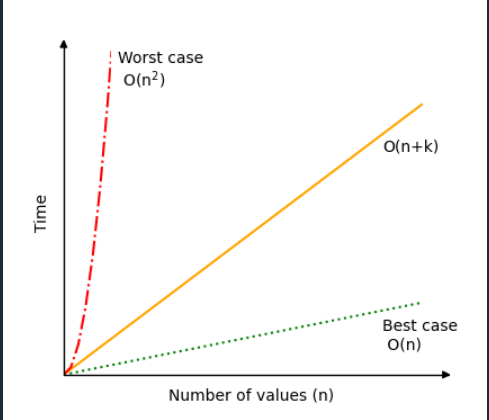


**\*\*\*\* Algorithm Time Complexity \*\*\*\***

This method is

**(the best case: )**

**(the worst case: )**

****

# **DSA Radix Sort :**

# sort the array by individual digits, start with the rightmost

* **How it works** :
  + Start with the least significant digit (rightmost digit)
  + Sort the values based on the digit in focus by first putting the values in the correct bucket based on the digit in focus, and then put them back into the array in the correct order.
  + Move to the next digit and sort again, like in the step above until there are no digits left.

1. Stable Sorting:

* Radix Sort must sort the elements in a stable way for the result to be sorted correctly.
* The result would be the same if the algorithms are stable or not.
* It is important for Radix Sort that the sorting is done in a stable way because the elements are sorted by just one digit at a time.

1. Manual Run Through

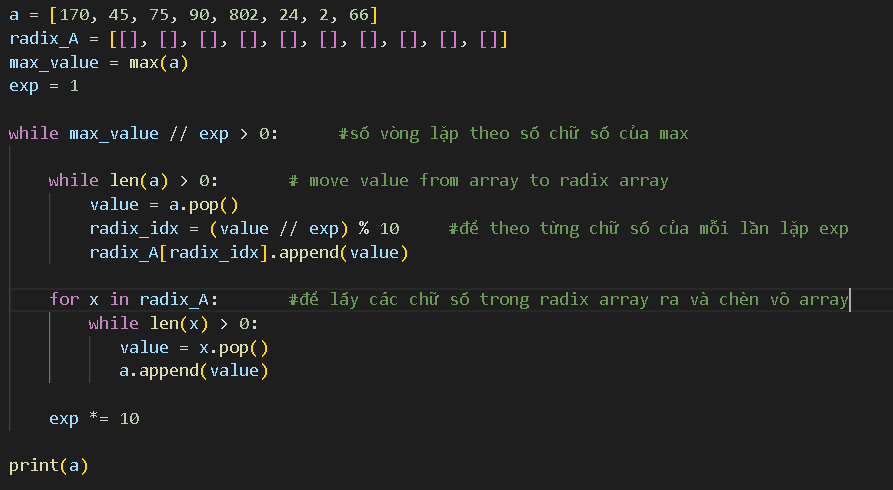
* Start with an unsorted array and an empty array to fit values with corresponding radices 0 till 9.

Array = [33, 45, 40, 25, 17, 24]  
radixArray = [ [ ], [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] ]

* Start sorting by focusing on the least significant digit ( the rightmost )  
  Array = [33, 45, 40, 25, 17, 24]  
  radixArray = [ [ ], [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] ]
* Move the elements into the correct positions in the radix array according to the digit in focus.  
  Array = []  
  radixArray = [ [40 ], [ ] , [ ] , [33] , [ 24] , [45, 25 ] , [ ] , [17 ] , [ ] , [ ] ]
* Move the elements back into the initial array, and the sorting is now done for the least significant digit.  
  Array = [40, 33, 24, 45, 25, 17]  
  radixArray = [ [ ], [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] ]
* Move the focus to the next digit:   
  Array = [40, 33, 24, 45, 25, 17]  
  radixArray = [ [ ], [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] ]
* Move the elements into the radix array according to the focused digit  
  Array = [ ]  
  radixArray = [ [ ], [17 ] , [ 24, 25] , [33 ] , [ 40, 45] , [ ] , [ ] , [ ] , [ ] , [ ] ]
* Move the elements back into the start of Array.  
  Array = [17,24, 25, 33, 40, 45]  
  radixArray = [ [ ], [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] ]

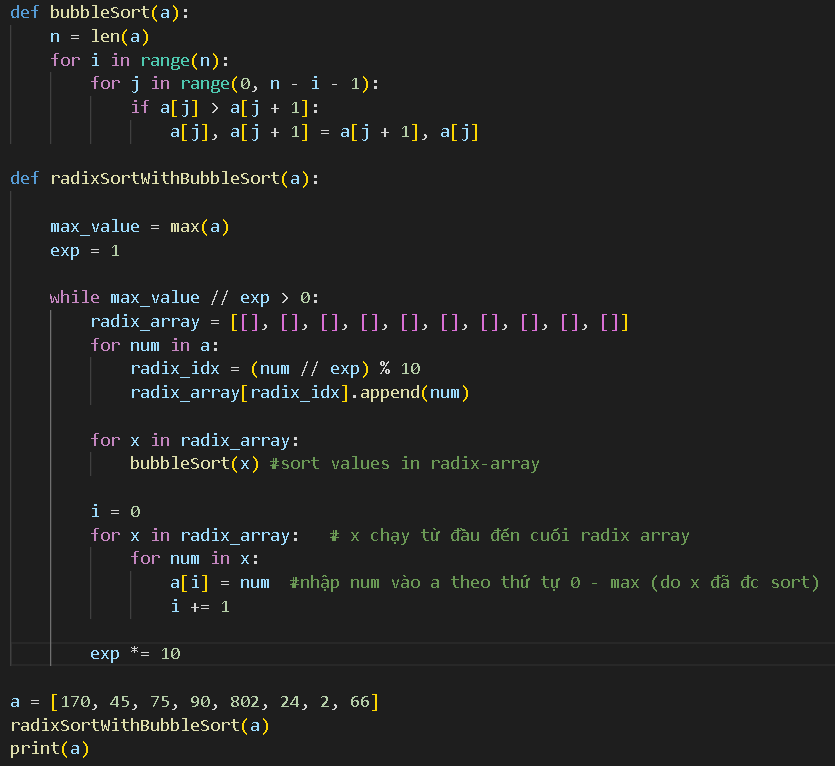
1. Implementation :

* An array with non negative integers that need to be sorted
* A two dimensional array with index 0 to 9 to hold values with the current radix in focus.
* A loop that takes values from the unsorted array and places them in the correct position in the two dimensional radix array.
* A loop that puts values back into the initial array from the radix array.
* An outer loop that runs as many times as there are digits in the highest value.



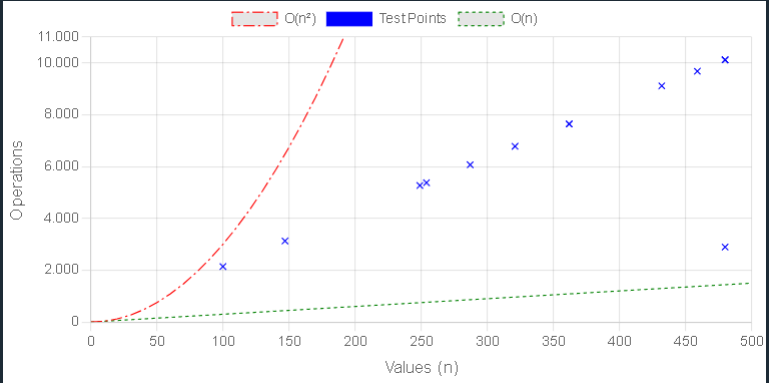
1. Radix Sort Using Other Sorting Algorithms:

* When it comes down to sorting on a specific digit, any stable sorting algorithm will work (counting sort or bubble sort).



**\*\*\*\* Algorithm Time Complexity \*\*\*\***

This method is



# **DSA Merge Sort :**

* Is a divide-and-conquer algorithm that sorts an array by first breaking it down into smaller arrays, and then building the array back together the correct way so that it is sorted.
* **Divide** : The algorithm starts with breaking up the array into smaller and smaller prices until one such sub-array only consists of one element.
* **Conquer** : The algorithm merges the small pieces of the array back together by putting the lowest values first, resulting in a sorted array.
* **How it works :** 
  + Divide the unsorted array into 2 sub-arrays, half the size of the original.
  + Continue to divide the sub-arrays as long as the current piece of the array has more than one element.
  + Merge 2 sub-arrays together by always putting the lowest value first.
  + Keep merging until there are no sub-arrays left.

1. Manual Run Through :

* Start with an unsorted array, it splits in half until the sub-arrays only consist of 1 element. The Merge Sort function calls itself 2 times, once for each half of the array.The first sub-array will split into the smallest pieces first.   
  [ 12, 8, 9, 3, 11, 5, 4]

[ 12, 8, 9] [ 3, 11, 5, 4]

[ 12] [ 8, 9] [ 3, 11, 5, 4]

[ 12] [ 8] [ 9] [ 3, 11, 5, 4]

* The splitting of the first sub-array is finished, -> merge. 8 and 9 are the first 2 elements to be merged  
  [ 12] [ 8, 9] [ 3, 11, 5, 4]
* The next sub-arrays to be merged are [12] and [8, 9].  
  [ 8, 9, 12] [ 3, 11, 5, 4]
* The second bis sub-array is split recursively.  
  [ 8, 9, 12] [ 3, 11, 5, 4]  
  [ 8, 9, 12] [ 3, 11] [ 5, 4]  
  [ 8, 9, 12] [ 3] [11] [5, 4]
* 3 and 11 are merged back together  
  [ 8, 9, 12] [ 3,11] [5, 4]
* Sub-array with 5 and 4 is split and merged :   
  [ 8, 9, 12] [ 3, 11] [ 4, 5]
* The 2 sub-arrays on the right are merged and compared:   
   [ 8, 9, 12] [ 3, 4, 5, 11]
* The 2 last remaining sub-arrays are merged.  
  Before [ 8, 9, 12] [ 3, 4, 5, 11]

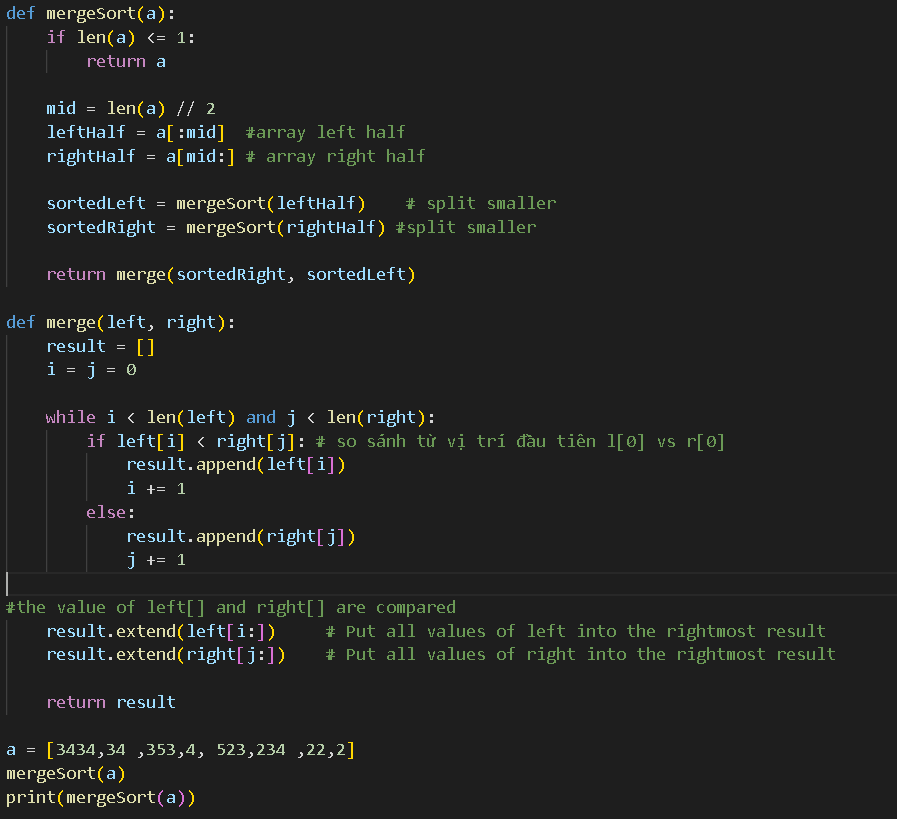
After: [ 3, 8, 9, 12] [ 4, 5, 11]

* And compare each value in the right sub-array step-by-step until the big array is sorted:

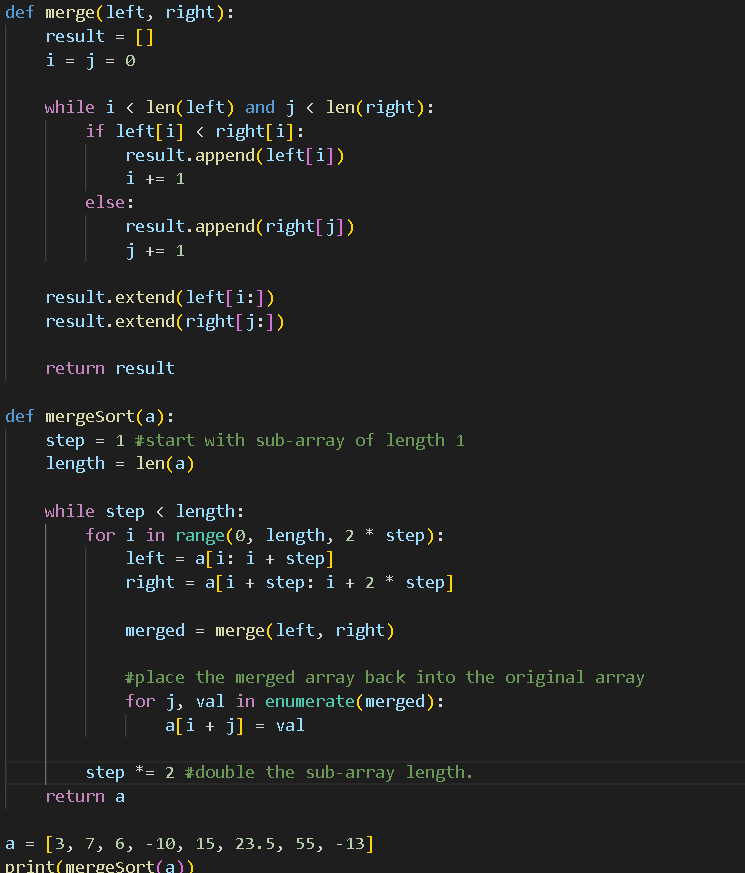
[ 3,4,5,8,9,11,12]

1. Implementation :

* An array with values that needs to be sorted.
* A function that takes an array, splits it in two, calls itself with each half of that array so that the arrays are split again and again recursively, until a sub-array only consists of 1 value.
* Another function that merges the sub-arrays back together in a sorted way.

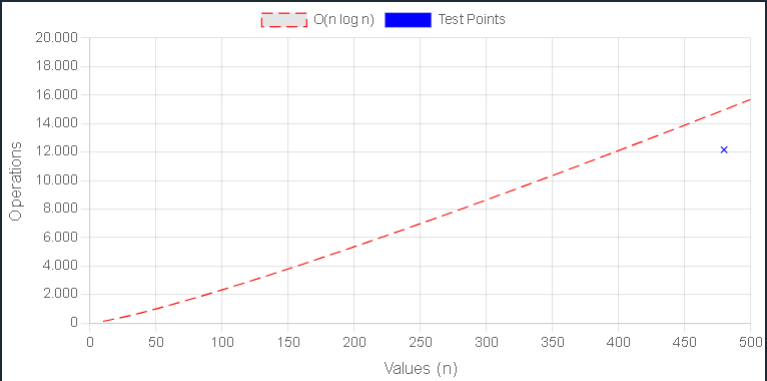


1. Merge Sort without Recursion :   
   ***bao h học python thì xem lại***



**\*\*\*\* Algorithm Time Complexity \*\*\*\***

This method is

****

# **DSA Linear Search :**

* The big difference between *sorting* algorithm and *searching* algorithms is that sorting algorithm modify the array, but searching algorithm leave the array unchanged.
* **How it works :** 
  + Go through the array by value from the start
  + Compare each value to check if it is equal to the value we are looking for.
  + If value is found, return the index of that value,
  + If the end of the array is reached and the value is not found, return -1 to indicate that the value was not found.

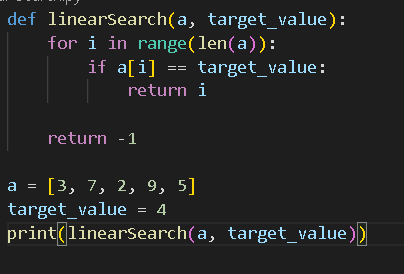
1. Manual Run Through.

* We will search for value 11:
* Start with an array of random values.  
  [12, 8 , 9, 11, 5, 11]
* Look ar the first value in the array, and compare to 11.  
  [12, 8 , 9, 11, 5, 11]
* Move on to the next value and compare to 11:  
  [12, 8 , 9, 11, 5, 11]
* Check the next value:  
  [12, 8 , 9, 11, 5, 11]
* Move on to the next value  
  [12, 8 , 9, 11, 5, 11]

=> We have found it -> Value 11 is found at index 3 => Return index position 3. -> Finished

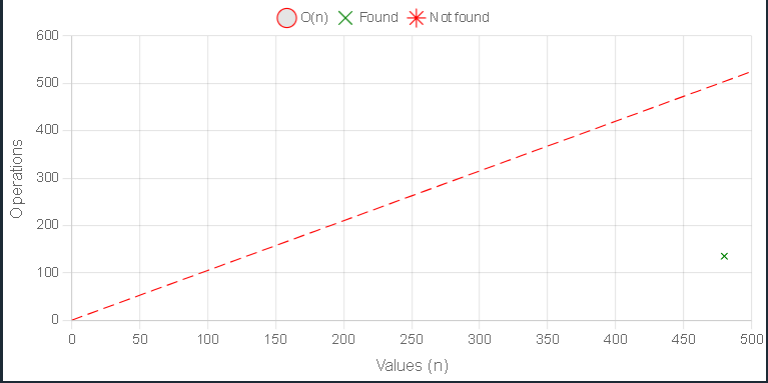
1. Implementation:

* An array with values to search through.
* A target value to sear
* A loop that goes through the array from the start to end
* An if-statement that compares the current value with the target value, and returns the current index if the target value is found
* After the loop, return -1, because at this point we know the target value has not been found.



**\*\*\*\* Algorithm Time Complexity \*\*\*\***

This method is

****

# **Binary Search :**

* Search through an array and return the index of the value it searches for
* Work by checking the value in the center of the array.
* **How it works :** 
  + Check the value in the center of the array
  + If the target value is lower, search the left half of the array. If the target value is higher, search the right half.
  + Continue step 1 and 2 for the new reduced part of the array until the target value is found or until the search area is empty.
  + If the value is found, return the target value index. If the target value is not found, return -1.

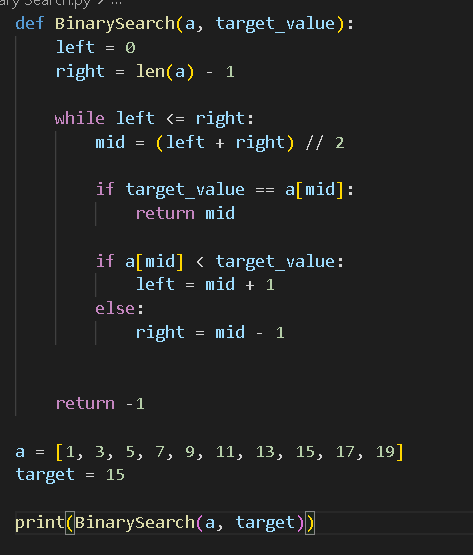
1. Manual Run Through:

* Start with an array :   
  [2, 3, 7, 7, 11, 15, 25]
* The value in the middle of the array at the index 3, compare to 11  
  [2, 3, 7, 7, 11, 15, 25]
* 7 < 11 -> Seach to the right of index 3.the values to the right are [11, 15, 25]. The next value to check is the middle value 15, at index 5  
  [2, 3, 7, 7, 11, 15, 25]
* 15 if higher than 11 -> Must search to the left of index 5. -> Only index 4 left   
  [2, 3, 7, 7, 11, 15, 25]

=> Found it -> At index 4 -> Return the index position 4.

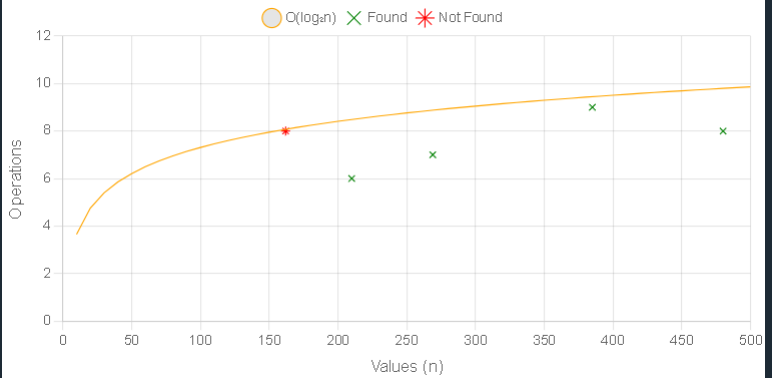
1. Implementation :

* An array with values to search through.
* A target value to search for.
* A loop that runs as long as left index is less than , or equal to, the right index.
* An if-statement that compares the middle value with the target value, and returns the index if the target is found.
* An if-statement that checks if the target value is less than, or larger than, the middle value, and updates the “left” or “right” variables to narrow down the search area.
* After the loop, return -1, because at this point we know the target value has not been found.



**\*\*\*\* Algorithm Time Complexity \*\*\*\***

This method is

****

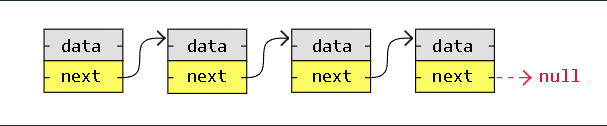
**Linked Lists**

**Linked Lists**

# **DSA Linked lists :**

* A **Linked list** is a list where the nodes are linked together. Each node contains data and a pointer. The way they are linked together is that each node points to where in the memory the next node is placed.

1. Linked Lists:



* **Benefit :** 
  + Nodes are stored wherever there is free space in memory, the nodes do not have to be stored contiguously right after each other like elements are stored in arrays.
  + When adding or removing nodes, the rest of the nodes in the list do not have to be shifted.

1. Linked Lists and Arrays :

* **Linked Lists** : Consist of nodes and is a linear data structure we make ourselves.
  + Nodes store links to other nodes.
* **Arrays** : Is an existing data structure in the programming language that we can use.
  + Elements do not need to store links to other elements.
* **Compare :**

|  | **Arrays** | **Linked Lists** |
| --- | --- | --- |
| *An existing data structure in programming language* | Yes | No |
| *Fixed size in memory* | Yes | No |
| *Elements, or nodes are stored right after each other in memory ( contiguously )* | Yes | No |
| *Memory usage is low ( each node only contains data, no links to other nodes )* | Yes | No |
| *Elements, or nodes can be accessed directly ( random access )* | Yes | No |
| *Elements, or nodes can be inserted or deleted in constant time, no shifting operation in memory needed* | No | Yes |

# **DSA Linked Lists in Memory :**

1. Computer Memory :

* Is a storage the program uses when it is running. This is where the variables, arrays, and linked lists are stored.

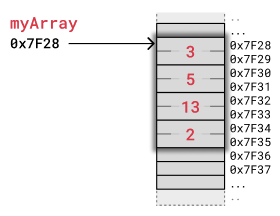
1. Variables in Memory :

* Assume the integer **17** is stored as two bytes (16 bits ), and the address in memory to myNumber is 0x7F30.
* 0x7F30 is actually the address to the first of the 2 bytes of memory where the myNumber integer value is stored.
* When the computer goes to 0x7F30 to read an integer value -> it knows that it must read both the first and second byte, since integers are two bytes on the specific computer.

1. Arrays in Memory :

* Elements in an array are stored contiguously in memory -> Each element is stored right after the previous element.

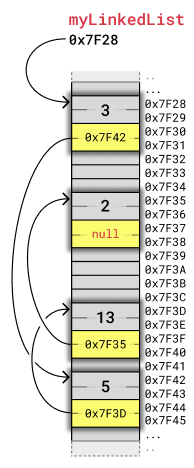
**How it works :**

****

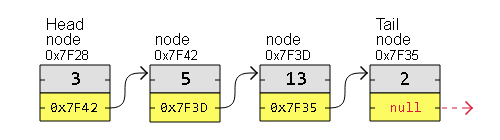
* An array of integers : myArray = [3, 5, 13, 2] is stored in memory
* The computer has only got the address of the first byte of myArray ,so to access the 3rd element with code myArray[2] the computer start at 0x7F28 and jumps over the 2 first integers.
* The computer know that an integer is stored in 2 bytes, so it jumps 2x2 bytes forward from 0x7F28 and reads value 13 starting at address 0x7F32 .  
  When removing or inserting elements in an array, every element that comes after must be either shifted up to make place for the new element, or shifted down to take the removed element’s place.

1. Linked Lists in Memory :

* We can **create** a linked list instead of storing like an array.
* Used in many scenarios ( like dynamic data storage ), stack and queue implementation or graph representation, to mention some of them.
* Consist of nodes with some sort of data and at least one pointer, or link, to other nodes.
* **Benefit :** 
  + Nodes are stored wherever there is free space in memory, the nodes do not have to be stored contiguously right after each other like elements are stored in arrays.
  + When adding or removing nodes, the rest of the nodes in the list do not have to be shifted.
* **How it works :**

****

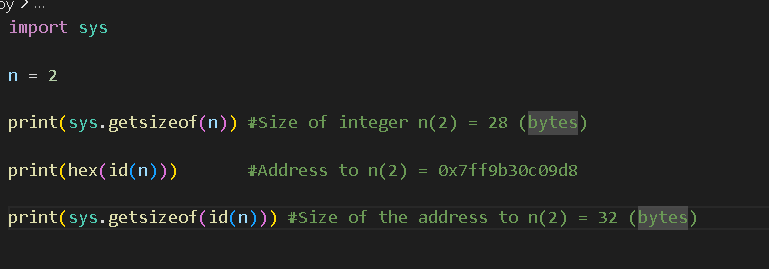
* + The linked list has 4 nodes with value 3, 5, 13, 2 and each node has a pointer to the next node in the list.
  + Each node takes up 4 bytes :
    - 2 bytes are used to store an integer value
    - 2 bytes are used to store the address to the next node in the list
  + How many bytes that are needed to store integers and addresses **depend on** the architecture of the computer



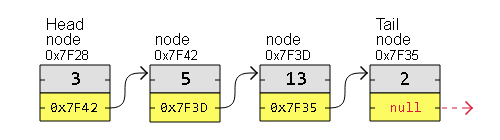
* + We cannot access a node directly -> To get to node number 5 in a linked list, we must start with the first node call “*head*”, to get to the next node, and do so while keeping track of the number of nodes we have visited until we reach node number 5.
* Learning able Linked Lists helps us to understand concepts like **memory allocation** and **pointer** .

1. Memory in Modern Computers:

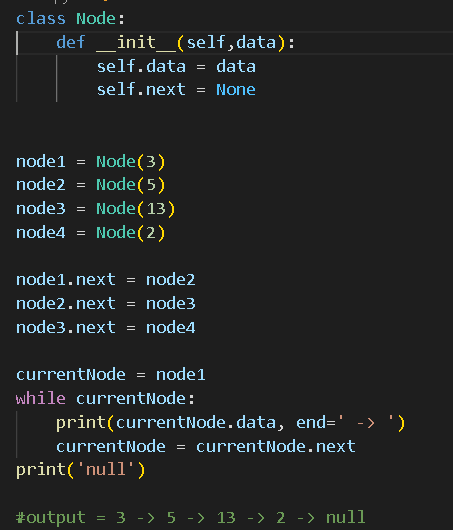
* We have used the memory in an 8 bit **microcontroller** to keep it simple and easier to understand.
* Memory in modern computers work in the same way in principle as memory in an 8 bit microcontroller, but:
  + more memory is used to store integers
  + more memory is used to store addresses.



1. Linked List Implementation :



* The Node class represents what a node is : The node contains data and a link to the next node.
* The Node class is used to create 4 nodes, the nodes are then linked together and printed at the end.

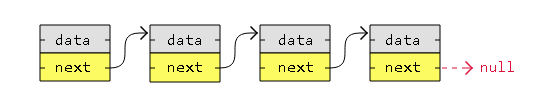


# **DSA Linked Lists Types :**

1. Types of Linked Lists :

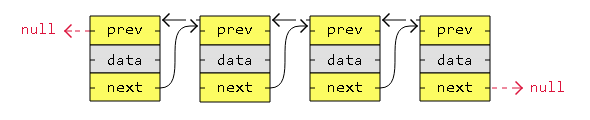
* There are three basic forms:
  + Singly Linked Lists
  + Doubly Linked Lists
  + Circular Linked Lists.

1. **Singly Linked List** : the simplest kind of linked lists. -> Take up less space in memory because each node has only 1 address to the next node, like this:



=> Have “*null*” in the end or start

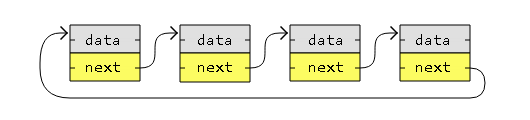
1. **Doubly Linked Lists** : has nodes with addresses to both the previous and the next node, and therefore takes up more memory. -> It is **good at** move both up and down in the list.



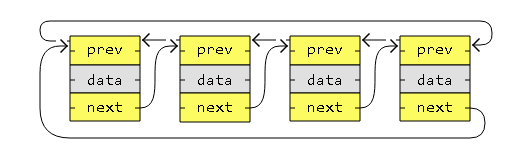
=> Have “*null*” in the end or start

1. **Circular Linked List** : like a singly or doubly linked list with the first node, the “*head*” and the last node, the”*tail*”, connected.

* More complex code is needed to explicitly check for start and end nodes in certain applications.
* **Good at** cycle through continuously.
* A Singly circular linked list:

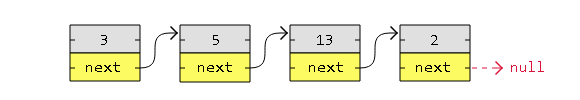


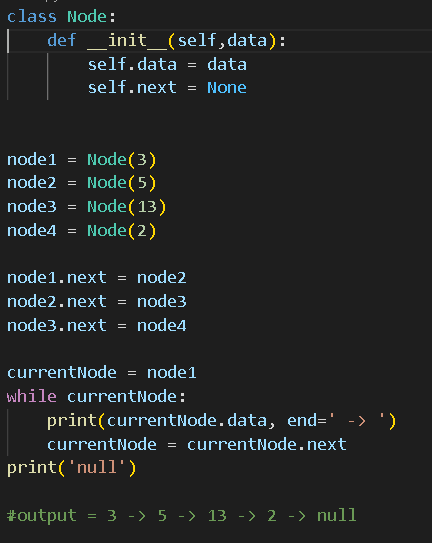
* A doubly circular linked list:



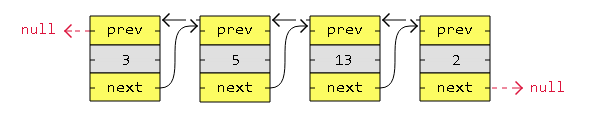


1. Linked List Implementations:
   1. Singly Linked List Implementation:



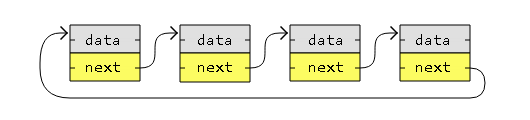


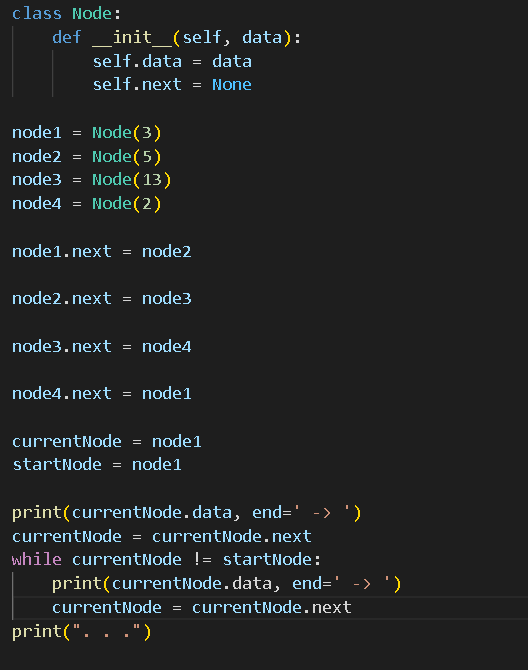
* 1. Doubly Linked List Implementation :



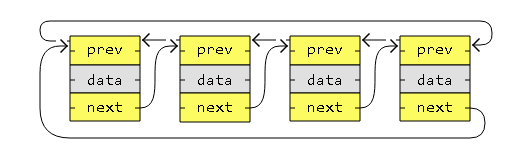


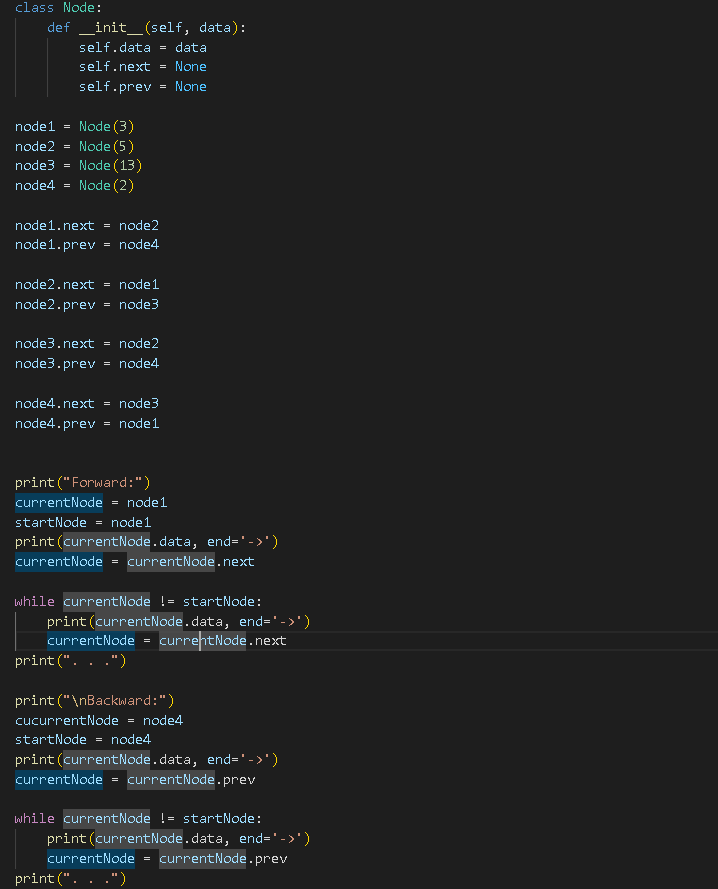
* 1. Circular Singly Linked List Implementation:





* 1. Circular Doubly Linked List Implementation :



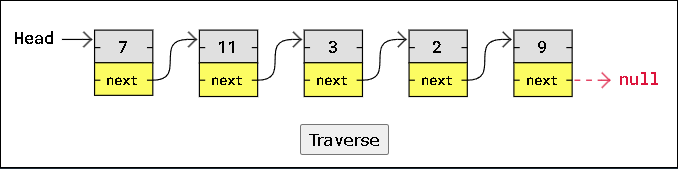


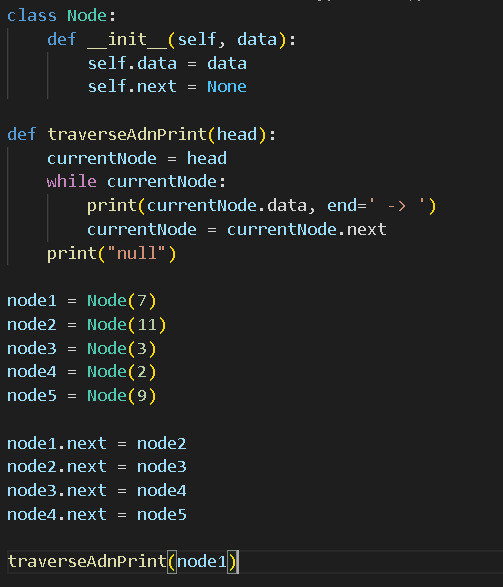
# **DSA Linked Lists Operations :**

* Basic things we can do with linked lists :
  + Traversal
  + Remove a node
  + Insert a node
  + Sort

1. Traversal (duyệt) of a Linked List :

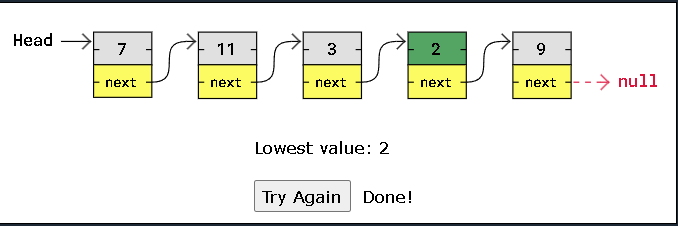
* Go through the linked list by following the links from one node to the next.
* Use cases :
  + Search for a specific node
  + Read or modify the mode’s content
  + Remove the node
  + Insert a node right before or after that node.
* To traverse a singly linked list :
  + Start with the first node in the list, the head node.
  + Follow that node’s next link.
  + The next node’s next link , until the next address is null:

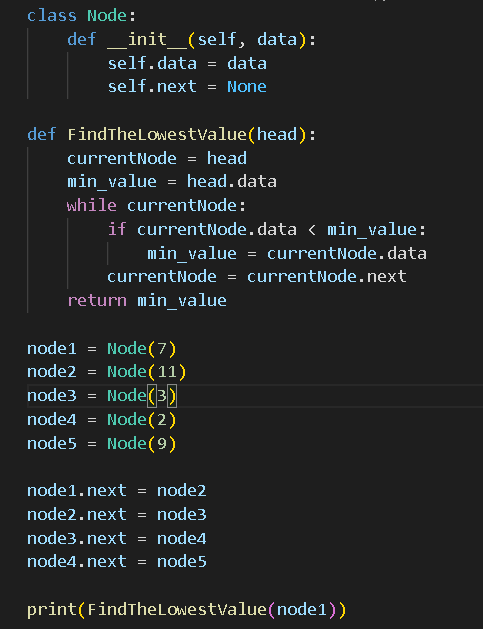




1. Find the lowest Value in a linked list :

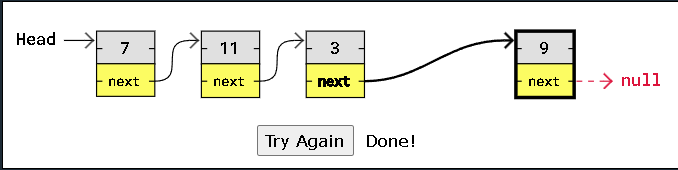
* Similar to how we found the lowest value in an array -> Need to follow the next link to get the next node
* -> Need to traverse the list in the previous code -> We must also update the current lowest value when we find a node with a lower value :

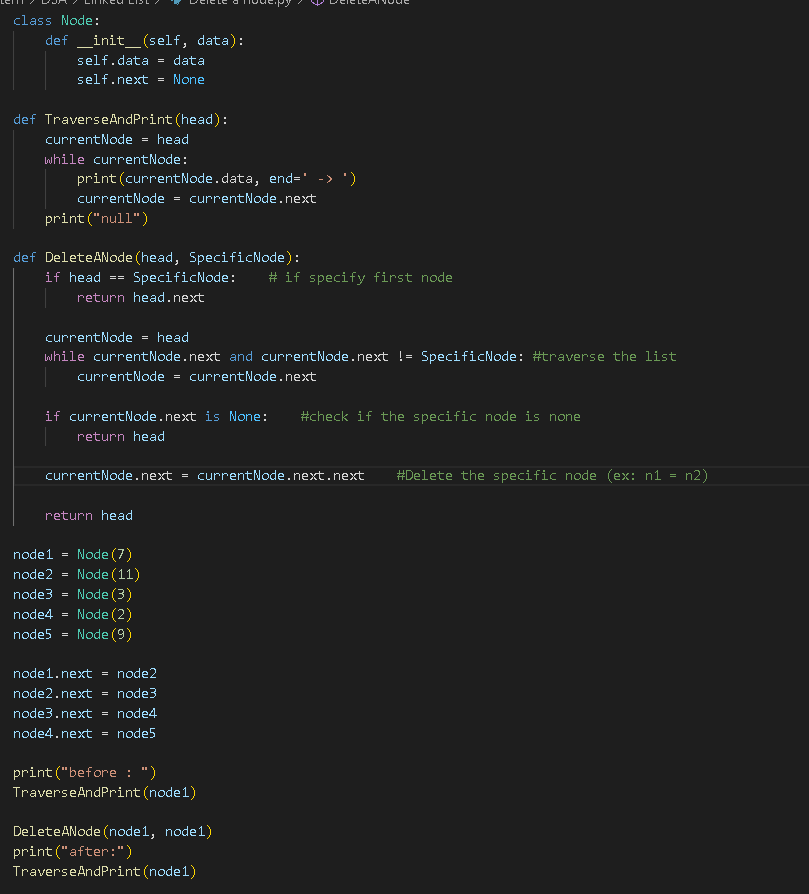




1. Delete a Node in a linked list :

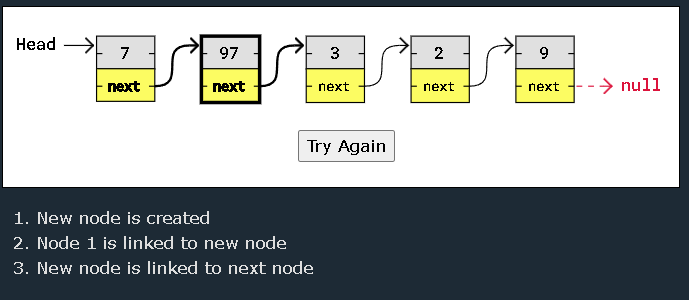
* Important to connect the nodes on each side of the node before deleting it, so that the linked list is not broken.
* -> Before deleting a node, we need to get the next pointer from the previous node -> Connect the previous node to the new next node before deleting the node in between.

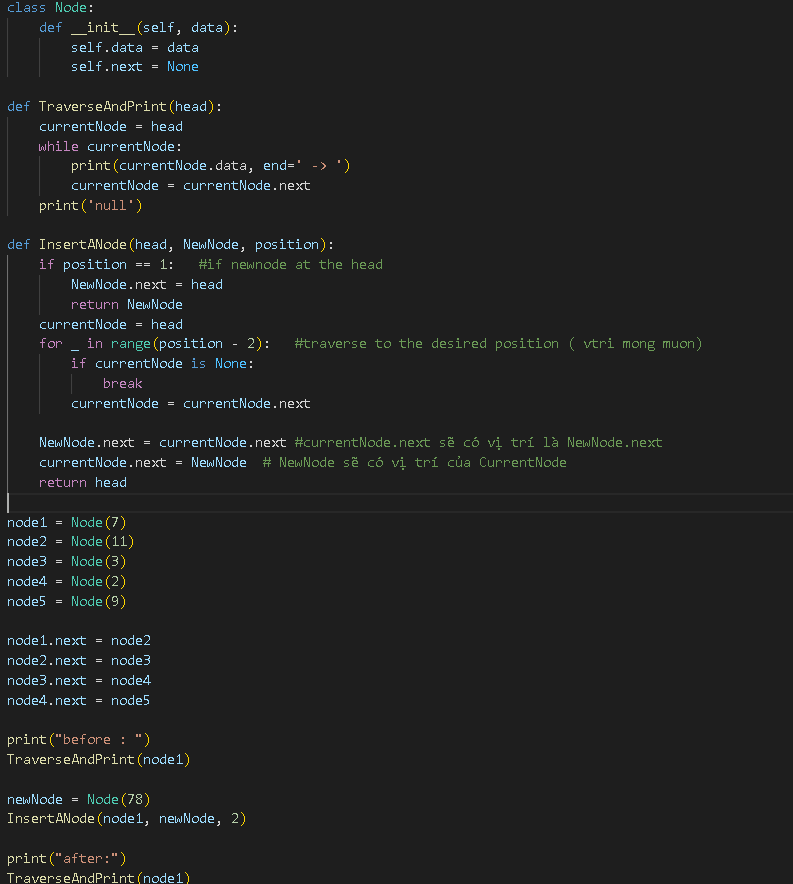




1. Insert a Node :

* Similar to deleting a node -> Need to take care of the next pointers to make sure we do not break the linked list.
* To insert a node :
  + Create the node,
  + At the position where we insert it -> Adjust the pointers -> The previous node points to the new node
  + The new node points to the correct next node.





1. Other Linked Lists Operations :

* There are a lot of other operations that could be done with linked lists : sorting,...

**\*\*\* : *We cannot sort linked lists with sorting algorithms like* Counting Sort, Radix Sort , Quicksort *because they are indexes to modify array elements directly based on their position.***

1. Linked Lists and Arrays :

* Linked lists are not allocated to a fixed size in memory ***like arrays are***, -> **Do not** require to move the whole list into a larger memory space when the fixed memory space fills up.
* Linked lists nodes are not laid out one right after the other in memory (contiguously) -> Do not have to shifted up or down in memory when nodes are inserted or deleted
* Linked lists nodes **require** more memory to store one or more links to other nodes. <-> Array elements **do not require** that much memory , because array elements do not contain links to other elements.
* Linked list operations are **usually harder** to program and require more lines than similar array operations, -> because programming languages have better built in support for arrays.
* We must traverse a linked list to find a node at a specific position, <-> with arrays we can access an element directly by writing myArray[x].

***\*\*\*\** : We don't need to write code to handle when an array fills up its memory space, and we do not have to shift elements up or down in memory when an element is removed or inserted,but these things still happen in the background and can cause problems in time critical applications.**

# **Time Complexity of Linked Lists Operations :**

* The **exact time** it takes for an algorithm to run depends on programming language, computer hardware, differences in time needed for operations on arrays vs linked lists, and many other things as well

| **Operations** | **Time Complexity** | **Auxiliary Space** | **Explanation** |
| --- | --- | --- | --- |
| Insertion at beginning | O(1) | O(1) | Constant-time pointer updates. |
| Insertion at end | O(n) | O(1) | Traversal required to find the last node. |
| Insertion at position | O(n) | O(1) | Traversal to the desired position, then constant-time pointer updates. |
| Deletion at beginning | O(1) | O(1) | Constant-time pointer update. |
| Deletion at end | O(n) | O(1) | Traversal required to find the second last node. |
| Deletion at position | O(n) | O(1) | Traversal to the desired position, then constant-time pointer updates |
| Searching in linked lists | O(n) | O(1) | Traversal through the list to find the desired value. |

**Stacks**

# **DSA Stacks :**

* Like a pile of pancakes.
  + The pancakes are both added and removed from the top.

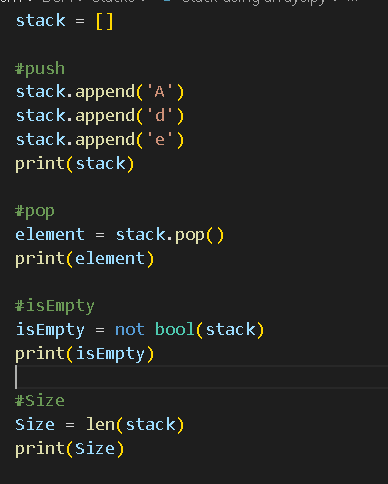
-> When removing a pancake, it will always be the last pancake we added.

-> ***Last in First out*.**

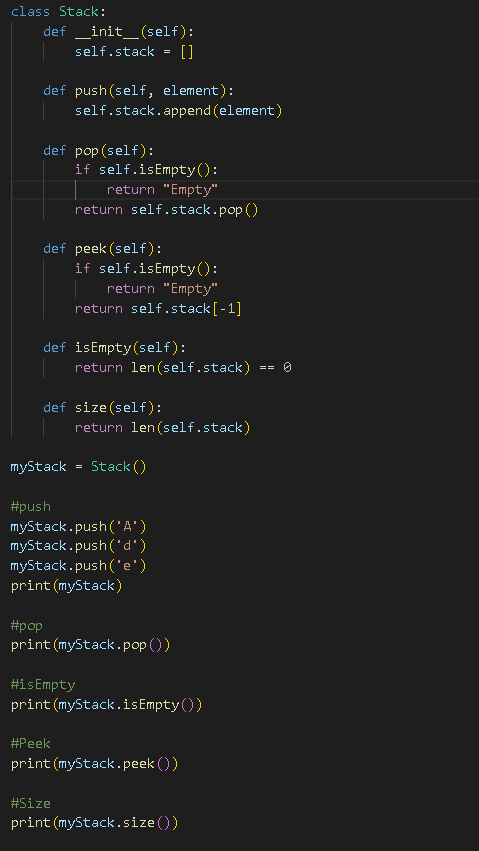
* Basic operations :
  + **Push** : Add a new element on the stack
  + **Pop** : removes and returns the top element from the stack
  + **Peek** : return the top element on the stack
  + **isEmpty** : check if the stack is empty
  + **Size** : find the number of elements in the stack.
* Can be implemented by using arrays or linked lists.
* Can be used to implement undo mechanisms (cơ chế hoàn tác)
  + To revert to previous states.
  + To create algorithms for depth-first search in graphs or for backtracking.
* Often mentioned together with **Queues**.

1. Stack implementation using Arrays.

* Reasons to implement stacks using arrays :
  + **Memory Efficient** : array elements do not hold the next elements address like linked list nodes do
  + **Easier to implement and understand** : Using arrays to implement stacks require less code than using linked lists -> Easier to understand.
* Reason for **not** using arrays to implement stacks :
  + **Fixed size** : An array occupies (account) a fixed part of the memory -> It could take up more memory than needed, or cannot hold more elements if filling up the array.

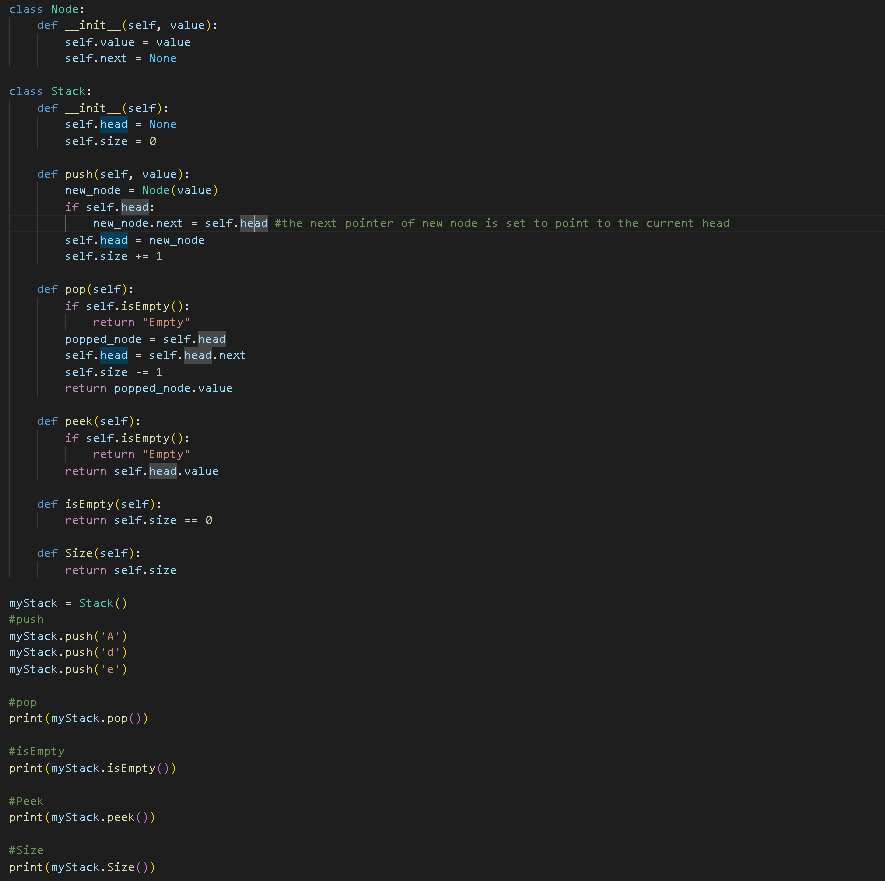
****

**=> Should create a stack class :**

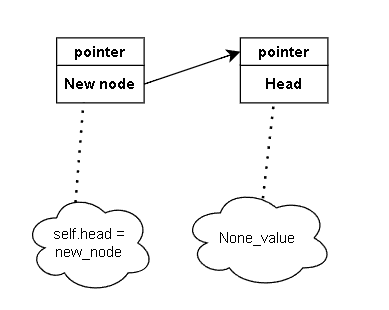
****

1. Stack implementation using Linked Lists :

* Reason for using linked lists :
  + **Dynamic size** : the stack can grow and shrink (giảm) dynamically <-> Unlike with arrays.
* Reason for **not** using linked lists:
  + **Extra memory** : Each stack element must contain the address to the next element.
  + **Readability** : The code might be harder to read and write for some because it is longer and more complex.



\* ***Push***:



# **DSA Queues :**

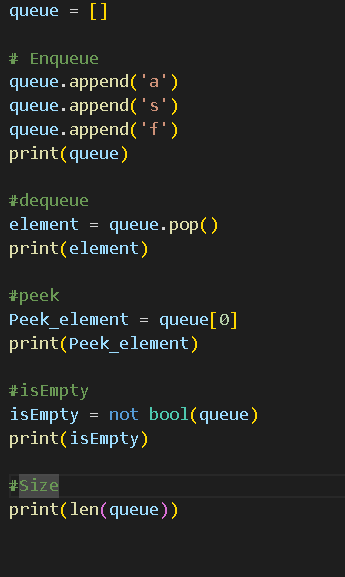
* A data structure that can hold many elements.
* Like people standing in line in a supermarket.
  + The first person to stand in line is also the first who can pay and leave the supermarket.

=> ***First in First out*** .

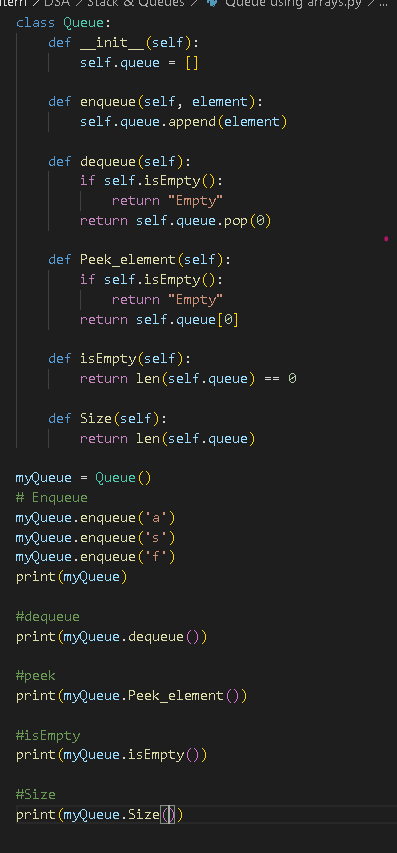
* Basic operations :
  + **Enqueue** : Add a new element to the queue.
  + **Dequeue** : Remove and return the first element from the queue.
  + **Peek** : Return the first element in the queue.
  + **isEmpty** : Check is the queue is empty
  + **Size** : Find the number of elements in the queue.
* Can be implemented by using arrays and linked lists.
* Can be used to :
  + implement job scheduling for an office printer
  + order processing for e-tickets.
  + create algorithms for breadth-first (chiều rộng) search in graphs.
* Often mentioned together with Stacks.

1. Queue implementation using Arrays :

* Reason to implement queues using arrays :
  + **Memory efficient** : Array elements do not hold the next elements' address.
  + **Easier to implement and understand** : using arrays require less code.
* Reason for **not** using arrays to implement queues :
  + **Fixed size** : An array occupies a fixed part of the memory -> It could take up more memory than needed, or cannot hold more elements if filling up the array. And resizing an array can be costly (tốn kém)
  + **Shifting cost** : Dequeue causes the first element in a queue to be removed -> other elements must be shifted to take the removed elements’ place -> Inefficient and can cause problems.
  + **Alternatives** : Some programming languages have built-in data structures optimized for queue operations that are better than using arrays.

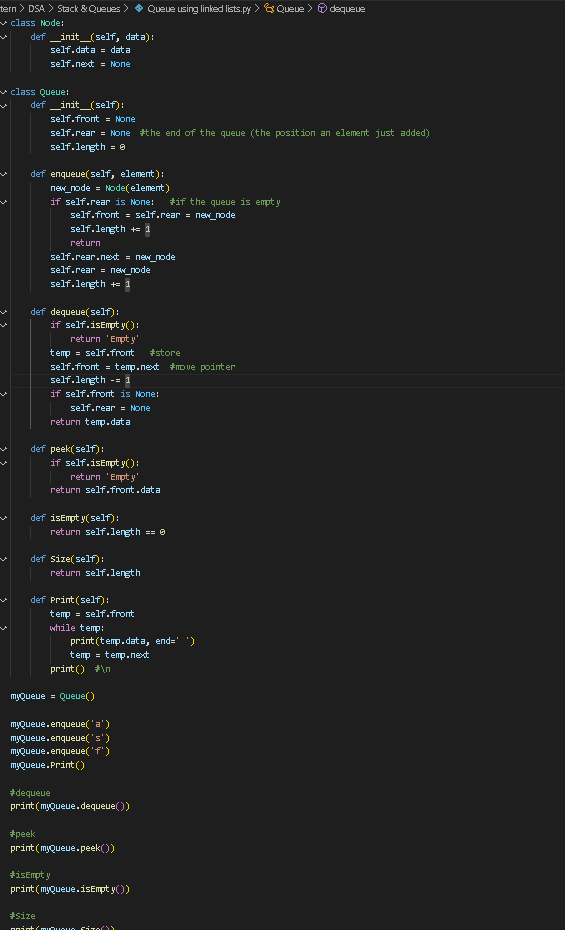


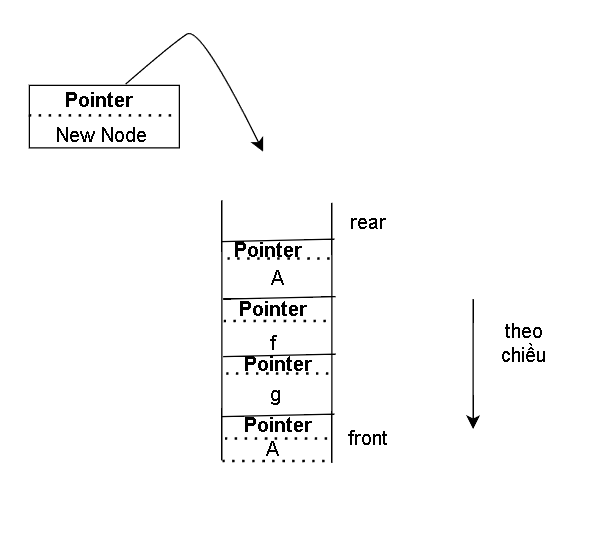
=> Using Queue class :



1. Queue implementation using Linked lists:

* Reason for using linked lists :
  + **Dynamic size :** The queue can grow and shrink dynamically
  + **No shifting** : The front element of the queue can be removed (enqueue) without having to shift other elements in the memory
* Reason for **not** using linked lists :
  + **Extra memory** : Each queue element must contain the address to the next element
  + **Readability** : The code might be harder to read and write because it is longer and more complex.





**Hash Tables**

|  | **Hash Set** | **Hash Map** |
| --- | --- | --- |
| *Uniqueness and storage* | Every element is a unique key. | Every entry is a key-value-pair, with a key that is unique, and a value connected it |
| *Use case* | Checking if an element is in the set, ( *like checking if a name is on a guest list.* ) | Finding information based on a key,( *like looking up who owns a certain telephone number*.) |
| *Is it fast to search, add and delete elements?* | Yes, average O(1) | Yes, average O(1) |
| *Is there a hash function that takes the key, generates a hash code, and that is the bucket where the element is stored?* | Yes | Yes |

# 

# **Hash Table :**

* Is a data structure designed to be fast to work with.
* Are preferred -> because:
  + Searching for, adding, deleting can be done really quickly ( **even** for large amounts of data )
* Finding an element is done really fast because there is a way to go directly to where the element is stored -> using something call “**hash function**”

1. Building a Hash Table from Scratch :

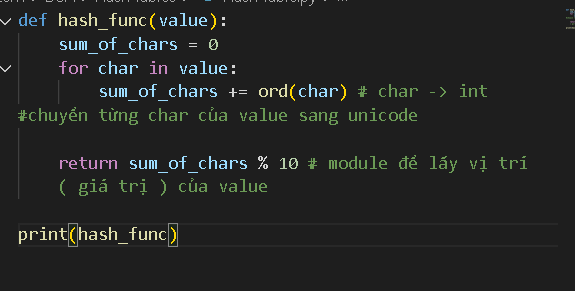
* Build the Hash Set in **5 steps** :
  + Starting with an array.
  + Storing names using a hash function.
  + Looking up an element using a hash function.
  + Handling collisions.
  + The basic Hash Set code example and simulation.
  1. Starting with an array:

my\_array = ['Pete', 'Jones', 'Lisa', 'Bob', 'Siri']

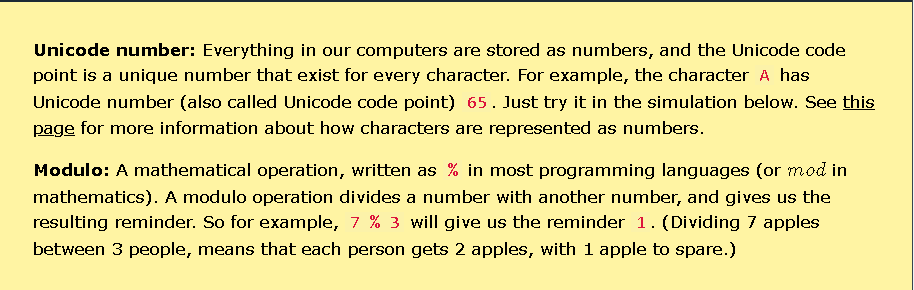
* To find ‘Bob’ -> We need to compare each name, element until we find ‘Bob’
* If the array was sorted alphabetically -> quickly to find Binary Search by using Binary Search but would mean a big operation of shifting elements in memory when inserting or deleting names. => Use a Hash Table.
* Assume there is at most 10 names in a list:

my\_array = [None, None, None, None, None, None, None, None, None, None]

* 1. Storing names using a hash function : Store a name directly
* Convert the value into a number that equals one of the Hash Set’s index numbers. ( 0 -> 9 )



(ex “B” has unicode = 66; “o” has 111, “b” has 98 => Adding = 275 => Modulo 10 of 275 = 5 => ‘Bob’ Should be stored at 5)



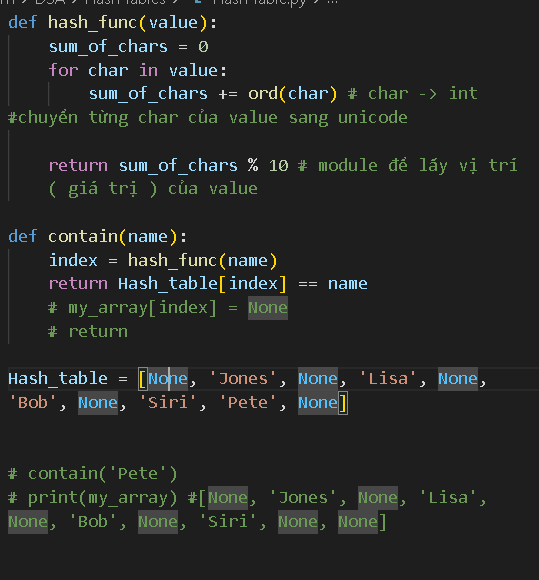
* After storing ‘Bob’ at index 5 :

my\_array = [None, None, None, None, None, ‘Bob’, None, None, None, None]

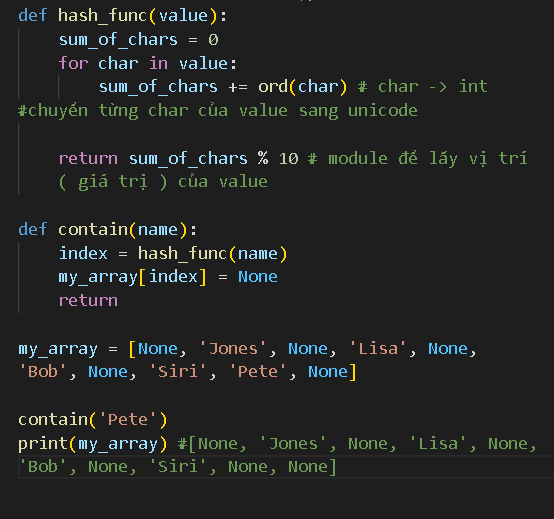
-> Do the same with others:

my\_array = [None, ‘Jones’, None, ‘Lisa’, None, ‘Bob’, None, ‘Siri’, ‘Pete’, None]

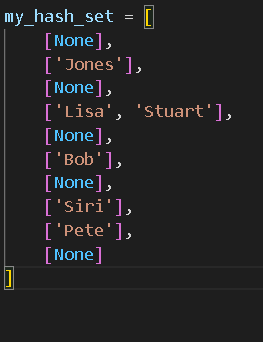
* 1. Looking up a name using a hash function :
* To find out if ‘Pete’ is stored in the array, we give the name ‘Pete’ to our hash function -> get back hash code 9-> Go directly to the element at index 9:



We can set a specific value to **None**

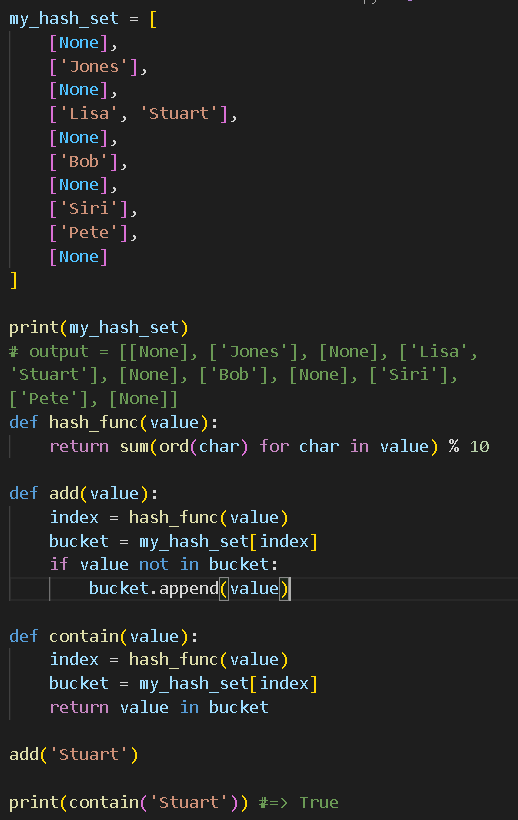


* 1. Handling collisions :
* Add ‘Stuart’ to the hash set -> at index 3
* Trying this will -> **collision** because ‘Lisa’ is already stored at index 3.
* To fix -> Make room for more elements in the same bucket -> **Chaining** .



\* Searching **‘Stuart’** in the Hash Set now means that using the hash function we end up in **bucket 3**, but then we must first check **‘Lisa’** in that bucket, before we find ‘Stuart’ as the second element in **bucket 3**.

* 1. Hash Set code example and simulation.



1. Uses Cases :

* Checking if something is in a collection ( finding a book in a library )
* Storing unique items and quickly finding them ( storing phone numbers )
* Connecting values to keys ( linking names to phone numbers )

=> **The most important reason** : Hash Tables are very fast compared to Arrays and Linked Lists (especially for large sets).

**\*\*\*\* Algorithm Time Complexity \*\*\*\***

Have just O(1)

\*\*\* : **Summarized :**

* Hash Table elements are stored in storage containers called **buckets**.
* Every Hash Table element has a part that is unique is called the **key.**
* A **hash function** takes the key of an element to generate a **hash code**.
* The hash code -> Bucket the element belongs to => We can go directly to that Hash Table element : To modify or delete it or just check if it exists.
* A **collision** happens when 2 hash table elements have the same hash code <-> They belong to the same **bucket**. => **Solution** :
  + **Chaining** : using arrays or linked lists to allow more than 1 element in the same bucket.
  + **Open Addressing** : Store an element in the next available bucket if that element is already in a bucket.

# **DSA Hash Sets :**

* Is a form of Hash Table data structure -> Usually holds a large number of elements.
* Hash Set -> Search, add and remove elements really fast.
* Hash Set -> Lookup, to check if an element is a part of a set
  + **Hash code** : A number generated from an element’s unique value (key), to determine what bucket that Hash Set element belongs to.
  + **Unique elements** : A Hash Set cannot have more than one element with the same value
  + **Bucket** : A Hash Set consists of many such buckets, or containers to store elements. ( 2 elements have the same hash code -> Belong to the same bucket ). => Often implemented as arrays or linked lists <-> Need to be able to hold more than one element.

1. Finding the Hash Code :

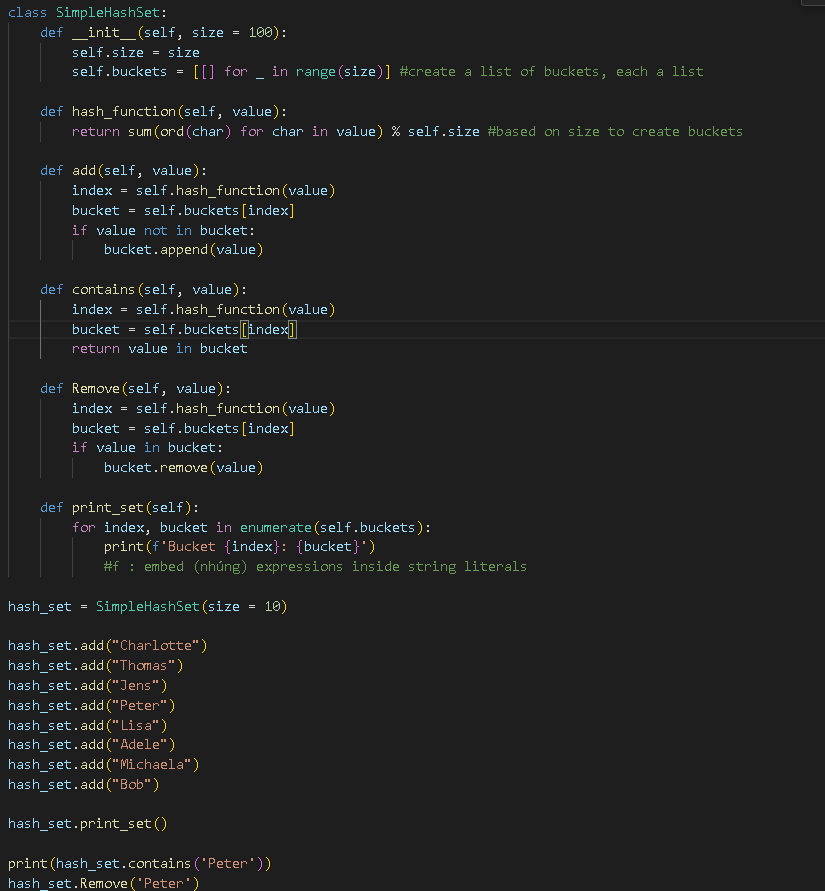
* A Hash Code is generated by a **hash function**.
  + Hash function : takes the name written in the input **->** Sums up the Unicode code points for every character in that name **->** Hash function does a modulo 10 operation ( % 10 ) on the sum of characters to get hash code as a number from 0 to 9.
* The name is put into one of ten possible buckets, according to the hash code of that name.
* The same hash code is generated and used when we want to search for or remove a name from the Hash Set.

1. Direct Access in Hash Sets :

* The best case is when only 1 element in a Bucket : ***O(1)***  -> Easy to search, add and remove
* The worst case is when we have to search through the other names in the bucket before we find an element : ***O(n)***
* **Solution** : Create a hash function that will distribute the elements evenly between the buckets, and to have around as many buckets as Hash Set elements.

1. Hash Set Implementation : ***Set*** data type

* Create class SimpleHashSet which have a method : \_\_init\_\_ to initialize the Hash Set
  + A method hash\_function for the hash function
  + Methods for basic Hash Set operations : add, contains, remove.



# **DSA Hash Maps :**

* Holding a large number of entries.
* Hash Map -> Search, add, modify and remove fast.
* Hash Map -> Find detailed information about something.

**=>** Would be useful if more information about each person was attached to be corresponding social security number (last name, birth date, address ,.. )

* **Entry** : Consists of a key and a value, forming a key-value pair.
* **Key** : Unique for each entry in the Hash Map -> Generate a hash code determining the entry’s bucket in the Hash Map -> Ensure that every entry can be efficiently located.
* **Hash Code**: A number generated from an entry’s key, to determine what bucket that Hash Map entry belongs to.
* **Bucket** : A Hash Map consists of many such buckets or containers -> to store entries.
* **Values** : Can be nearly any kind of information (name, birth date, address of a person)

1. Finding the Hash Code :

* Hash function : takes the name written in the input **->** Sums up the Unicode code points for every character in that name **->** Hash function does a modulo 10 operation ( % 10 ) on the sum of characters to get hash code as a number from 0 to 9.
* A person is stored in one of ten possible buckets in the Hash Map, according to the hash code of that person’s social security number.
* Hash code is generated and used when we want to search for or remove a person from the Hash Map.
  + Hash Code gives us instant access as long as there is just one person in the corresponding bucket.

1. Direct Access in Hash Maps :

* Searching for Charlotte -> We must use social security number 123-4567 (the Hash Map key), which generates the hash code 8 ( sum the security number and modulo 10 ) => We go straight to bucket 8 to get her name, without searching through other entries .
* This is the best case : ***O(1)*** for searching, adding and removing.
* In the worst case : ***O(n)*** <-> All people are stored in the same bucket -> We need to compare with all the other social security numbers in the bucket before we find the person.
* **Solution** : Create a hash function that will distribute the elements evenly between the buckets, and to have around as many buckets as Hash Set elements.

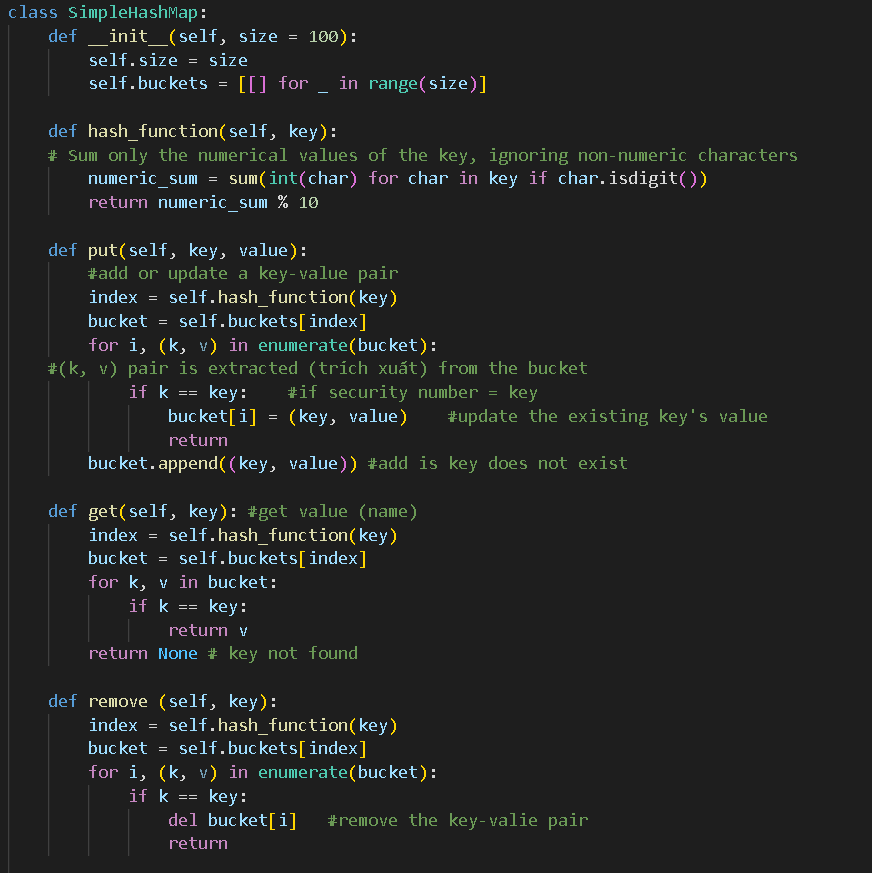
**\*\*\*\* : Note** : A social security number can be really long, like 11 digits -> It is possible to store 100 billion people with unique social security numbers.

Using an array where each person’s social security number is the index in the array where this person is stored is therefore a huge waste of space (mostly empty buckets)

Using a Hash Map ( a database with similar properties ) makes more sense as the number of buckets can be adjusted (điều chỉnh ) to the number of people.

1. Hash Map Implementation : ***dictionary***data type

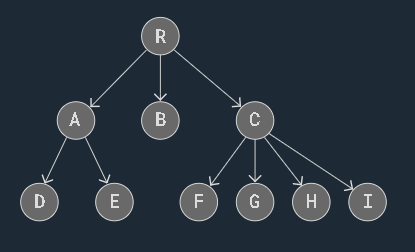
* Create a class SimpleHashMap:
  + Method \_\_init\_\_ to initialize the Hash Map
  + Method hash\_function for hash function.
  + Methods for basic operations:
    - put
    - get
    - remove.



**Trees**

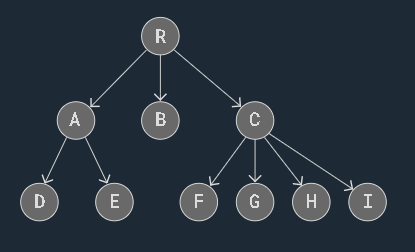
# **DSA Trees :**

* Each node contains data and can be linked to other nodes.
* A single element can have multiple ‘next’ elements, allowing the data structure to branch out in various directions.



* Uses Cases :
  + **Hierarchical (***phân* *cấp***) Data** : File systems, organizational models,...
  + **Databases** : Used for quick data retrieval.
  + **Routing (***định* *tuyến***) Tables** : Route data in network algorithms
  + **Sorting/Searching** : Sort data and search for data.
  + **Priority Queues (***hàng* *chờ ưu tiên***)** : Implemented using trees ( binary heaps ).

1. Tree Terminology and Rules :



* The first node (**R**): the **root** node.
* A link connecting one node to another (**->**) : an **edge**.
* A **parent** (**internal**) node has links to its **child** node.
* A node can have zero, one, or many child nodes.
* A node can only have 1 parent node.
* Node without links to other child : **leaves / leaf nodes** ( like a small kid )
* The **tree height** (*counting the longest branch*) is the *maximum* number of *edges* from the *root* node to a *leaf* node.
* The **height of a node** (*counting the longest branch a node has*) is the *maximum* number of *edges* between the node and a *leaf* node.
* The **tree size** is the number of nodes in the tree.

1. Types of Trees :

* Used to represent hierarchical relationships.
* **Binary Trees** : Each node has up to 2 children : The left child node and the right child node.
* **Binary Search Trees (BSTs)** : A type of Binary Tree where for each node, the left child node has a lower value; the right child node has a higher value.
* **AVL Trees** : A type Binary Search Tree that self-balances -> for every node, the difference in height between the left and right subtrees is at most one (<= 1).
  + This balance is maintained through rotations when nodes are inserted or deleted.

# **DSA Binary Trees :**

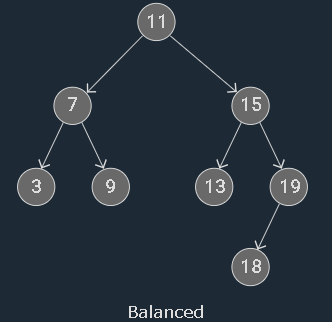
* A type tree data structure where each node can have a maximum of two child nodes, the left and right child nodes.
* **Benefits** :
  + Algorithms : Traversing, searching, insertion and deletion become easier to understand, to implement and run faster.
  + Keeping data sorted in a Binary Search Tree(**BST**) makes searching very efficient.
  + Balancing trees is easier to do with a limited number of child nodes (an AVL Binary Tree)
  + Binary Trees can be represented as arrays, making the trees more memory efficient.

1. Binary Trees vs Arrays and Linked Lists:

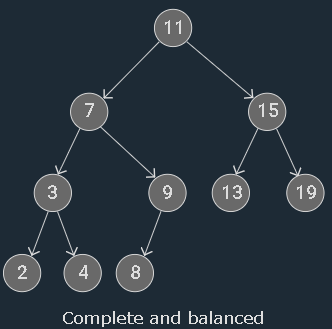
| **Arrays** | **Linked Lists** | **Binary Trees**  (BST and AVL trees) |
| --- | --- | --- |
| Fast when we want to access an element directly | Fast when inserting or deleting nodes, no memory shifting needed | Fast when deleting or inserting a node, with no shifts in memory needed. |
| Inserting and deleting elements require other elements to shift in memory -> ***time consuming*** | Access an element inside the list, the list must be traversed -> ***time consuming*** |  |

1. Types of Binary Trees :

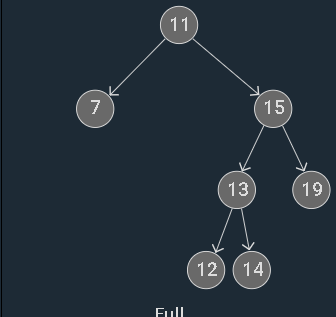
* A **balanced** Binary Tree has at most 1 in difference between its left and right subtree heights, for each node in the tree.



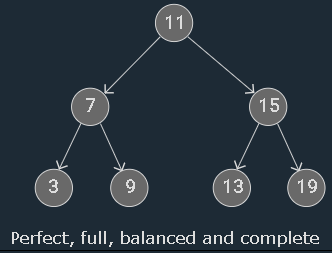
* A **complete** binary tree has all levels full of nodes, except the last level, which can also be full or filled from left to right -> the properties of a complete Binary Tree means to be **balanced**.
  + All nodes at a given level are filled before moving to the next level
  + If there are nodes at the last level, they are filled from left to right without any gaps.



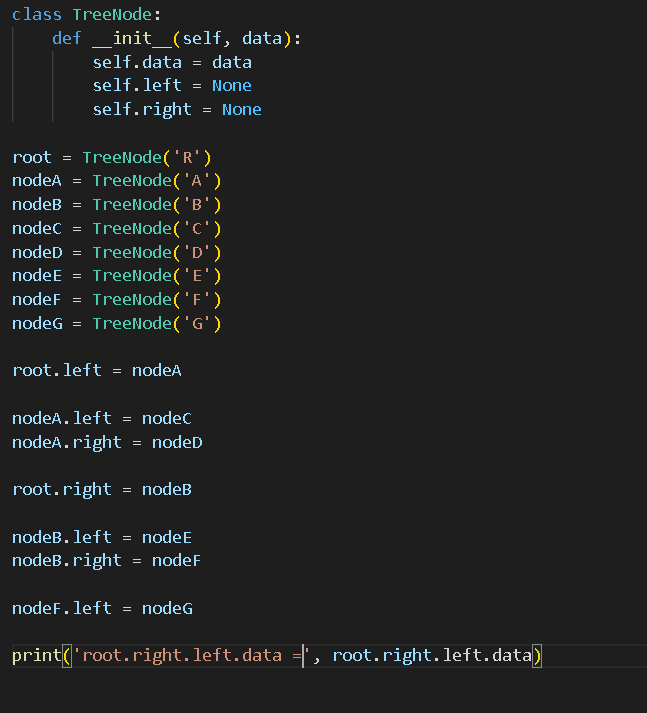
* A **full** Binary Tree where each node has either 0 or 2 child nodes



* A **perfect** Binary Tree has all leaf nodes on the same levels -> all levels are full of nodes and all internal nodes have 2 child nodes => The properties : **full**, **balanced**, **complete**.



1. Binary Tree Implementation :

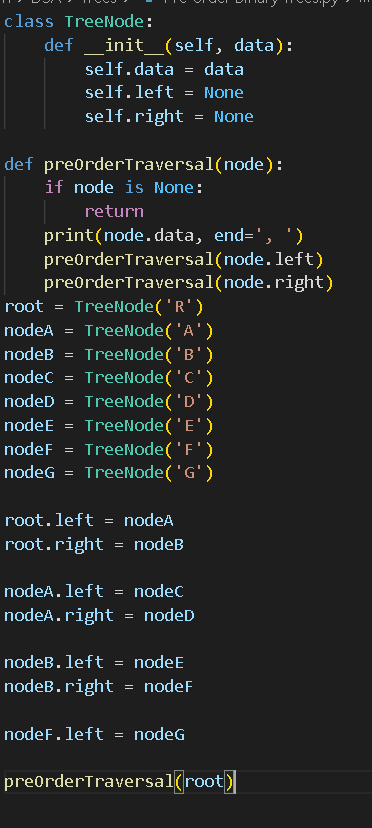
****

1. Binary Tree Traversal :

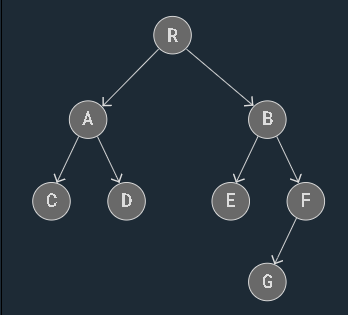
* Going through a Tree by visiting every node, one node at a time.
* Trees can branch out in different directions ( non-linear ) -> There are different ways of traversing Trees.
* 2 main categories of Tree traversal :
  + **Breadth First Search (BFS)** : When the nodes on the same level are visited before going to the next level in the tree -> The tree is explored in a more sideways direction.
  + **Depth First Search (DFS)** : When the traversal moves down the tree all the way to the leaf nodes, exploring the tree branch by branch in a downwards direction.
    - 3 different types of DFS : pre-order ; in-order ; post-order.

# **DSA Pre-order Traversal :**

* Is a type of DFS where each node is visited in a certain order.
* Visiting the root node first -> Recursively do a pre-order traversal of the left subtree -> Do the same with the right subtree.
* **Used for** : Creating a copy of the tree, prefix notation of an expression tree.
* This traversal is “pre” order because the node is visited “before” the recursive pre-order traversal of the left and right subtrees.

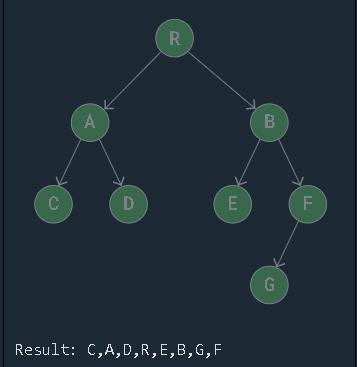


* The preOrderTraversal() : keep traversing the left subtree recursively before going on to traversing the right subtree.
  + The first time node is None is when the left child of node C is given as an argument.
  + After None is returned the first time when calling C’s left child, C’s right child also returns None, -> The recursive calls continue to propagate back -> A’s right child D is the next to be printed.

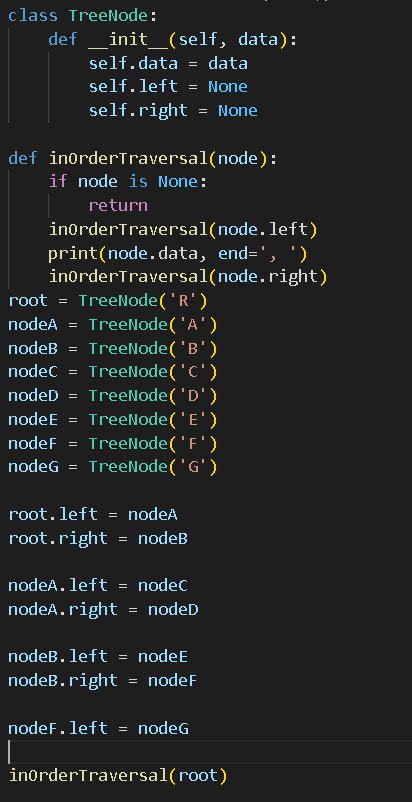


# **DSA In-order Traversal :**

* Is a type of DFS where each node is visited in a certain order.



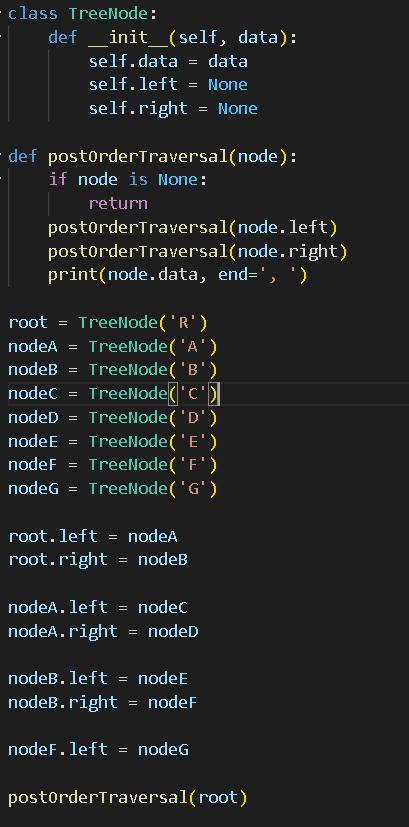
* This does a recursive In-order Traversal of the left subtree, visits the root node, and finally does a recursive In-order Traversal of the right subtree.
* **Used for** BST where it returns values in ascending order.
* This traversal “in” order : The node is visited in between the recursive function calls.
  + The node is visited after the In-order Traversal of the left subtree, before the In-order Traversal of the right subtree.



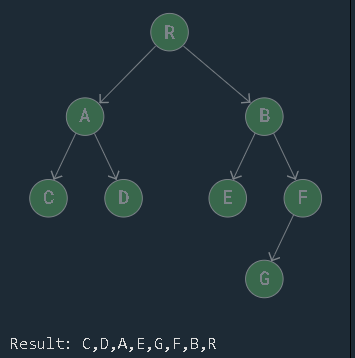
* The function keeps calling itself until to the latest right node
  + After printing the latest node m the previous function calls continue to run -> “A” gets printed -> “D” -> “R” -> So on.

# **Post-order Traversal of Binary Trees :**

* A type of DFS where each node is visited in a certain order.
* Work by recursively doing a Post-order Traversal of the left subtree and the right tree -> Visit the root node.
* **Used for** : deleting a tree, post-fix notation of an expression tree.
* This traversal “post” is that visiting a node is done “after” the left and right child nodes are called recursively.

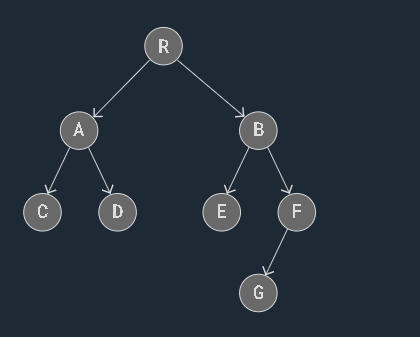


* The function keeps traversing the left subtree recursively, until None is returned when C’s left child node is called as the node argument.
* After C’s left and right child return None -> Print “C”.
* The function continues to propagate back to previous recursive function calls -> Next node to be printed “D” -> “A”.

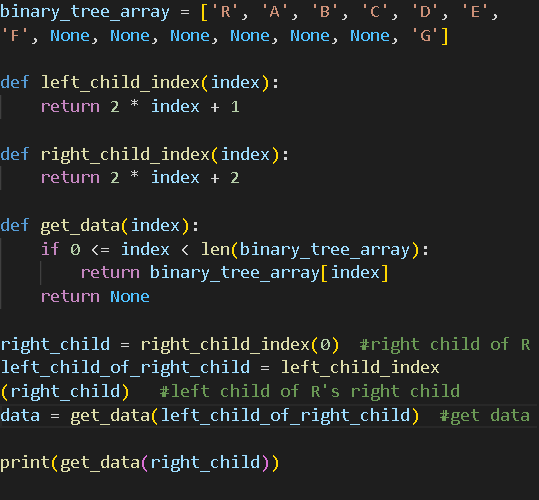


# **DSA Array Implementation :**

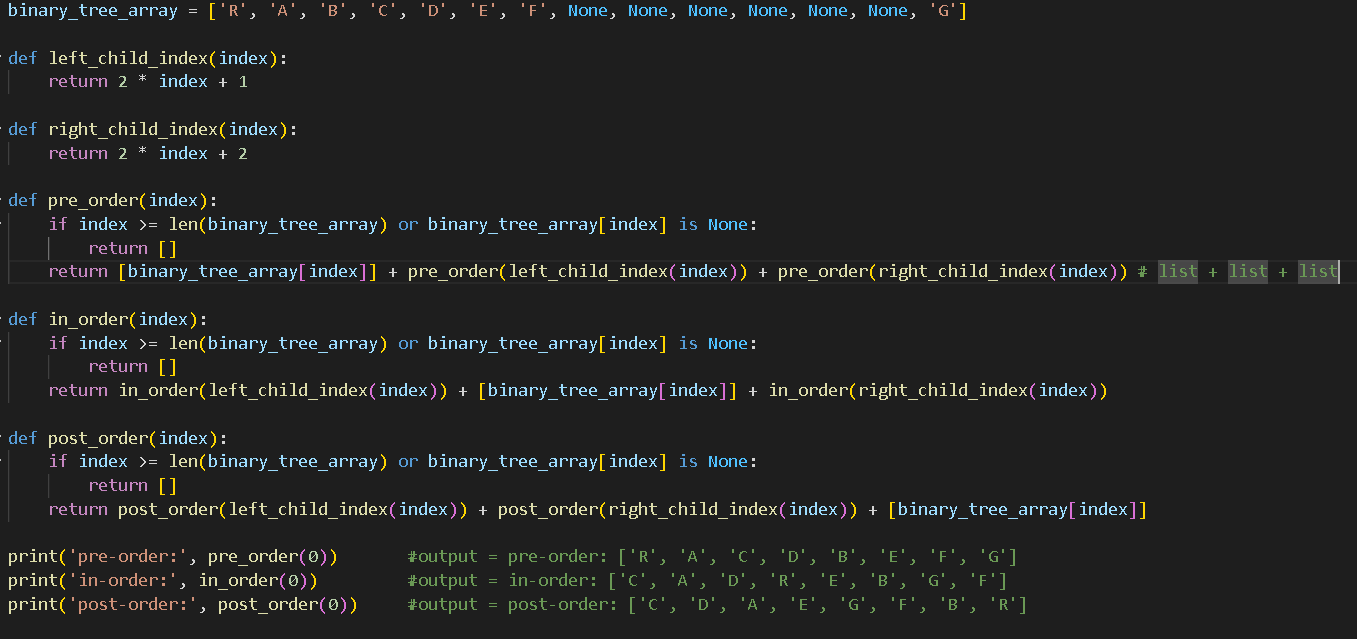
* To avoid the cost of all the shifts in memory that we get from using Array -> Implement Binary Trees with pointers from 1 element to the next.
* An array implementation of a Binary Tree can make sense as it needs less memory -> Easier to implement , faster for curtain operations due to cache locality.
* **Cache Locality** :
  + When the first cache memory in the computer stores parts of memory that was recently accessed
  + When the cache stores parts of memory that is close to the address that is currently accessed.
  + This happens because it is likely that the CPU needs something in the next cycle that is close to what it used in the previous cycle, either close in time or close in space.



* The root is index 0
* The right child is index 2i + 2
* The left child is index 2i + 1



* To avoid wasting space on empty Array elements -> Binary Tree stored using Array implementation should be “perfect” Binary Tree , or a nearly perfect one.
* How the three different DFS traversals can be done on an Array implementation of a Binary Tree.

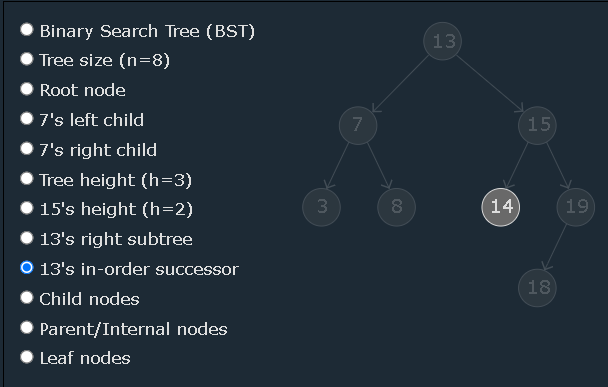


# **DSA Binary Search Trees :**

* A Binary Tree where every node’s left child has a lower value and every node’s right child has a higher value.
* A clear advantage : search, delete, insert are fast and done without having to shift values in memory.

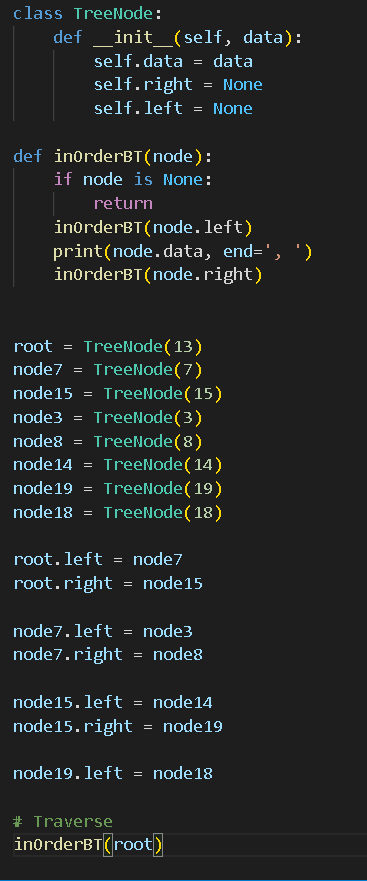
1. Binary Search Trees :

* A type of Binary Tree where the following properties must be true for any node “X” in the tree :
  + The X node’s left child and all its descendants (children, children’s children, and so on) have lower values than X’s value.
  + The right child, and all its descendants have higher values than X’s value.
  + Left and right subtrees must also be BST.
* These properties make it faster to search, add and delete values than a regular binary tree.
* The **size** of a tree is number of nodes in it (*n*)
* A **subtree** starts with one of the nodes in the tree as a local root, and consists of that node and all its descendants.
* The **descendants** of a node are all the child nodes of that node, and all their child nodes, and so on ( Just start with a node, and the descendants will be all nodes that are connected below that node )
* The **node’s height** is the maximum number of edges between that node and a leaf node.
* A **node’s in-order successor** is the node that comes after it if we were to do in-order traversal.



1. Traversal of a Binary Search Tree :

* Do an in-order traversal and check if the resulting list of values are in an increasing order.



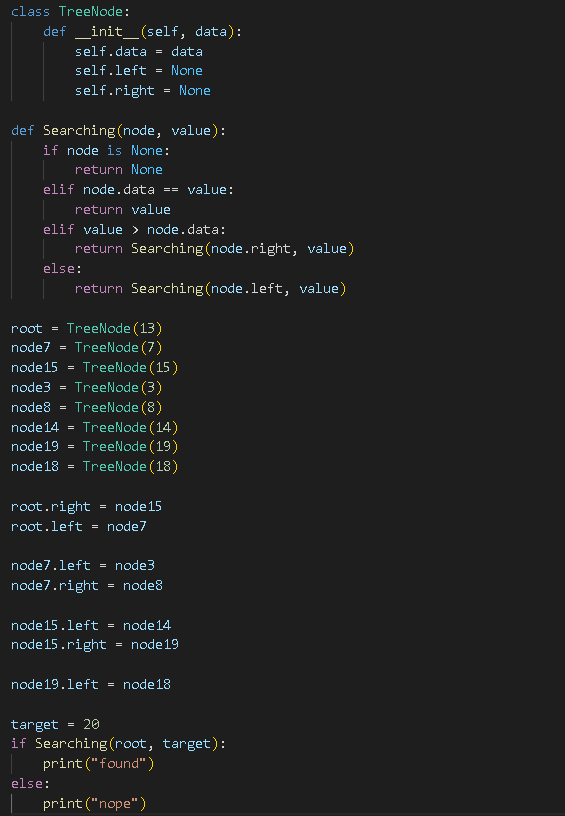
1. Search for a Value in a BST:

* The array must be sorted already **->** searching for a value in an array can then be done really fast.
* **How it works** :
  + Start with the root node.
  + If this is the value we are looking for, return.
  + If the value we are looking for is higher -> Continue searching in the right subtree.
  + If the value we are looking for is lower -> Continue searching in the left subtree.
  + If the subtree we want to search does not exist, depending on the programming language -> return None or NULL, or something similar -> To indicate that the value is not inside the BST.

**\*\*\*\* Algorithm Time Complexity \*\*\*\***

This method is

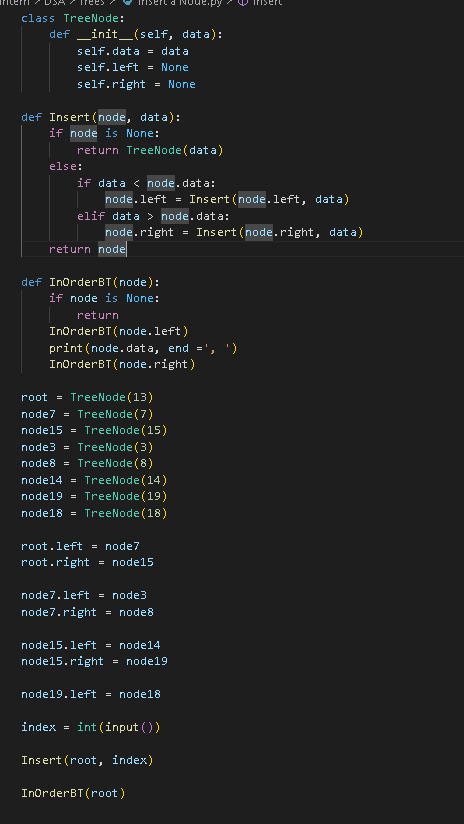
* *h is the height of the tree*



* It takes longer time to search the unbalanced tree because it is higher

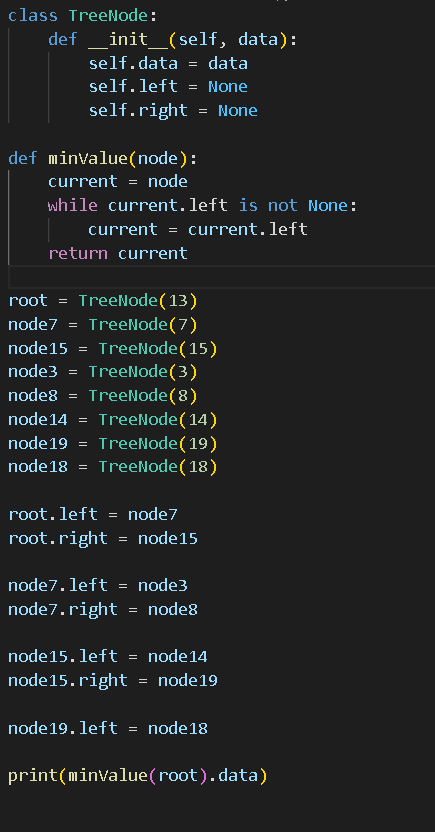
1. Insert a Node :

* **How it works :** 
  + Start with the root node.
  + Compare each node :
    - Is the value lower -> Go left
    - Is the value higher -> Go right
  + Continue to compare nodes with the new value until there is no right or left compared with -> That is where the new node is inserted.



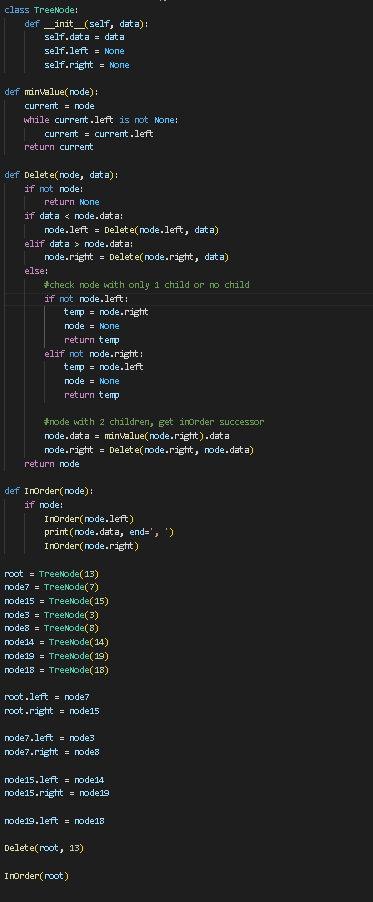
1. Find the lowest Value :

* **How it works** :
  + Start at the root node of the subtree.
  + Go left as far as possible.
  + The node we end up in is the node with the lowest value is the BST subtree.



1. Delete a Node :

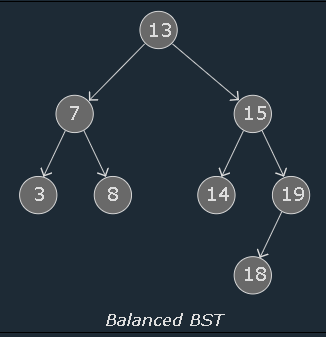
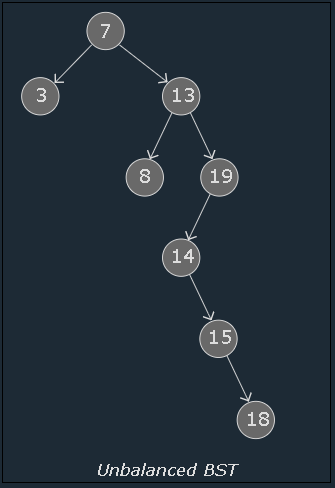
* **How it works** :
  + If the node is a leaf node -> remove it by removing the link to it
  + The node only has one child node -> Connect the parent node of the node we want to remove to that child node.
  + If the node has both right and left child nodes :
    - Find the node’s in-order successor
    - Change values with that node
    - Delete it.



1. Compared to Other DS :

| **Data Structure** | **Searching for a value** | **Delete / Insert leads to shifting in memory** |
| --- | --- | --- |
| Sorted Array | O(log*n*) | YES |
| Linked List | O(*n*) | NO |
| Binary Search Tree | O(log*n*) | NO |

1. BST Balance and Time Complexity :

* The higher the tree is (*h*), the longer the operation will take.
* Searching for a value is *O(*log*n)* because the tree is “balanced”
* The height of the left and right subtrees of any node only differs by 1.
* For a balanced BST, with a large number of nodes(big *n*), the height : *h* ≈ *logn* => O(*h)* = O(*logn)* 
* In case the BST is completely unbalanced, the height of the tree is approximately the same as the number of nodes : *h* ≈ *n* => O(*h)* ≈ O(*n)  
  *

# **DSA AVL Trees :**

* The **AVL** tree is a type of BST named after 2 Soviet Inventors Georgy : “**A**delson-**V**elsky and Evgenii **L**andis” invented the AVL tree in 1962.
* AVL trees are self-balancing, which means that the tree height is kept to a minimum -> A very fast runtime is guaranteed for searching, inserting and deleting nodes : with **time complexity** : O(log*n*).

1. AVL Trees :

* By keeping balance, the AVL Tree ensures a minimum tree height -> search, insert, delete operations can be done really fast.

1. The Balance Factor :

* The subtree heights are stored at each node for all nodes in an AVL Tree -> The balance factor is calculated based on its subtree heights to check if the tree has become out of balance.
* The **Balance Factor** (BF) for a node (X) is the difference in height between its right and left subtrees.

*BF(X) = height(rightSubtree(X)) - height(leftSubtree(X))*

* + BF = 0 : Balanced
  + BF > 0 : “Right heavy”
  + BF < 0 : “left heavy”
  + If BF < -1 or BF > 1 -> The tree is considered not in balance, and a rotation is needed to restore balance.

1. The Four “out-of-balance” Cases :

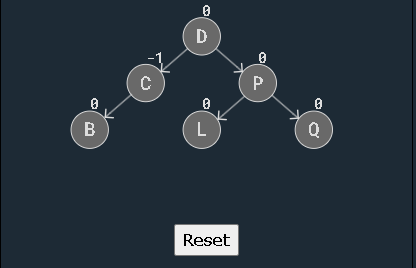
* There are four different ways an AVL tree can be out of balance, and each of these cases require a different rotation operation.

| **Case** | **Description** | **Rotation to restore balance** |
| --- | --- | --- |
| Left-left (LL) | The unbalanced node and its left child node are both left-heavy | A single right rotation. |
| Right-Right (RR) | The unbalanced node and its right child node are right-heavy | A single left rotation. |
| Left-Right (LR) | The unbalanced node is left-heavy, and its child node is right-heavy | First do a left rotation on the left child node ->Do a right rotation on the unbalanced nod. |
| Right-Left (RL) | The unbalanced node is right-heavy, and its right child node is left-heavy | First do a right rotation on the right child node -> Do a left rotation on the unbalanced node. |

1. The Left-Left Case (LL) :

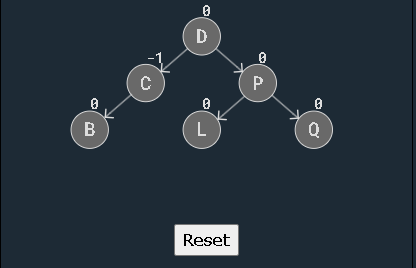
* When the unbalanced node is left heavy, and its left child node is also left heavy -> A single right rotation on the unbalanced node is enough to restore balance.

**\*\*\*\***: The left child is always lower than the node, and the right child is always higher.



1. The Right-Right Case (RR):

* When a node is unbalanced and right heavy, and the right child node is also right heavy.
* The RR case happens 2 times :





**Tab 7**