**Data Structure Types: 1. The Array**

1. **Arrays**

* Def: An array is an ordered series by arrangement. It’s the grouping of liked elements together.
* Once you ask for an array. The computer will provide you a section with an address within the memory to place your array in

**1.1 Fixed Arrays**

**Fixed Arrays – Introduction**

* Def: it is an array that stays at a specific size. It is an array which we cannot be increased by size (length), and you can insert as many elements as you wish as long as the array does not exceed its size. In cases when you want to insert more elements than what you have. You will need to delete some data and insert new data in the locations of the deleted data. Or use dynamic arrays
* Please note: we don’t care about decreasing the size of the array. Because if we have less elements than the set size of our array, we will not get an error. But if we try to have more elements than how much the length of the array can handle then we will get an error
* An array -> a = {7,8,9,10,1,2, B} -> this is an array of six pieces of data. Note: in python arrays must be of the same data type (integer, float, or string)
* The length of the array is n. Therefore, the last element of an array it is n-1
* If you type a [1] = 8, and if you type a[6] = B, but if you type a[7] you will get an error in high level languages like Python or Jave. However, in low level languages like C. you will start a “buffer overflow attack”. Where, you will call on data from other arrays within the memory, that are located next to the array you were working on

**Fixed Arrays – Run Times**

* Def: A fixed array is an array for which the size or length is determined when the array is created and/or allocated at the beginning. It sorts the data in a linear sequence.
* An array works by allocating a section of memory for storage
* The average run time it takes the computer to perform to a specific operation on fixed array are presented below:

|  |  |  |
| --- | --- | --- |
| Operation on a fixed array | Run Time | Explained |
| Insert an element at a random location | O(n)  Linear | At worst case scenario we would have to move every element in the array. Therefore, n operations |
| Insert an element at the front | O(n)  Linear | We would need to move every element at the array regardless, because we are inserting at the beginning. Therefore, n number of operations |
| Insert an element at the end | O(1)  Constant | The element will be added at the end. Therefore, it is a constant number of operations for 1 element you need one operation. Therefore. Other elements in the array do not need to move |
| Delete an element from a  Random location | O(n)  Linear | At worst case scenario we would have to move every element in the array. Therefore, n operations |
| Delete an element from the front | O(n)  Linear | We would need to move every element at the array regardless, because we are deleting at the beginning. Therefore, n number of operations |
| Delete an element from the Back | O(1)  Constant | The element will be removed from the end. Therefore, it is a constant number of operations for 1 element you need one operation. Therefore. Other elements in the array do not need to move |
| Search for an element in an unsorted array | O(n)  Linear | The brute force method of checking each element must be used. This means at worst case scenario we need to check every single element in the array. Therefore, n number of operations |
| Search for an element in a sorted array | Log(n) | The array is sorted from small to large. Therefore we perform a binary search algorithm |
| access time(how long it takes to get to an element within an array). For EXP in Array x -> x[3] is how we can get to the fourth element | O(1)  Constant | The specific element that we are required we can get to it at a constant time |

* The search for an element in a sorted array was log(n) is because:
  + We perform a binary search algorithm, which is possible, because of the way the array is set up in a sorted pattern
  + Log(n) means with the more elements we have the slower our run times becomes
  + The binary search algorithm uses the (left + right )/2 method, which is broken down to multiple steps, that divides the problem with each iteration:
    - Step 1: start from the full array. Therefore,
      * Left is the beginning of the array
      * Right is the end of the array
      * Therefore, each time you divide the problem in half to make it easier
    - For Example, y = {1,5,7,8,10,12,17,19,20,50,52}. Find if the number 19 is in your list. There are 11 elements
      * Step 1
        + Left is 0
        + Right is 10
        + 0+10/2 = 5
        + Now check the if the element number 5 is the element you are looking for. If it is you are done if not then check the points below:

X[5] =12

If x[5]’s answer is less than the number we are looking for then we do not need to search for anything to the left of that number, because our number will always be larger than these other numbers, because our number which we are looking for is larger than x[5]’s answer which is 12

If x[5]’s answer is more than the number we are looking for then we do not need to search for anything to the right of that number, because our number will always be smaller than these other numbers, because our number which we are looking for is smaller than x[5]’s answer which is 12

* + - * Steps 2
        + We know that we should only search to the right of x[5]. Therefore. Left = element 6 and right = last element
        + Left is 6
        + Right is 10
        + 6+10/2 = 8
        + Now check if element the answer of x[8] = 20 is larger or smaller than what you are looking for 19 if it smaller then you know your answer is between element x[6] and element x[8]. If it larger then you know your answer is between element x[9] and x[10] (the last element in the array x[10].
    - Note: if you are in a situation where you have left = 6 and right =7. Therefore
      * 6+7/2 = 6.5
      * We do not round this number to upper bound or lower bound we **truncate** it. Which means take only the integer. The integer is 6 in this case. You do this even if your answer was 6.687 you take 6. There are many truncation rules you may look at
  + **The binary search algorithm** continuously divides the number of operations n by half with each iteration, making the number of operations less each time. For example, 512 has only 8 operations -> log2(512) =8. Therefore, you get the log graph, which means **the average change** in the time of run time decreases with the increase of number of operations

**1.2 Circular Arrays**

**Circular Arrays – Introduction**

* A circular array is more efficient than a fixed array. The better efficiency in comparison to the fixed array occurs in the Inset(front) and delete(front) features of an array
* A circular array is more complex in comparison to the fixed array
* The circular array uses mod when it is appending elements to the start and end of the array
  + When inserting an element to the start of the array. Everything else does not shift down.
  + **Exp**. If X is your circular array. The length of X is 10. Then X[(front\_position1)mod(the length of the array] is the required input to append to the start of the array
  + X[(front\_position - 1)mod(the length of the array]
    - This method creates the following -> X[-1]mod10
    - The results of that can be found following the below method:
      * When X is negative in X % N and N is positive
        + Find the highest multiple of N equal or lower than the target number, X.
        + Then ask, how much higher is X than that
      * -1 mod 10 >
        + The lowest multiple of 10 equal or below -1 is -10
        + Therefore, the answer is 9, because -1 is 9 times higher than -10
  + This is how an element is appended in a circular element. Therefore, it is much faster. The circular array is not like the fixed array. The fixed array has the start and ending of an array as fixed positions. However, in the circular array the start and end of the array are detected by markers. Thus, when we append, we are moving that marker not the whole cell.
    - The formula used is X[(front\_position - 1)mod(the length of the array] to move the marker
    - The formular for the back is X[(front\_position + 1)mod(the length of the array]
    - Then we call on the start or end of the array. We will call on the markers
  + When deleting the first element (the element that has the start-cursor pointing) at it from the front of a circular array then the start- cursor will move to the next element.
  + When deleting the last element (the element that has the end-cursor pointing at it) from the end of a circular array then the end- cursor will move to the prior element.
* **The mod**: The reminder of the division of two numbers. The results of mod should always be a whole integer
  + Exp1: 5%2=1
    - 5 apples is what we have in the field
    - 2 apples is how much your net can take at once
    - 5%2 = 1. 1 is number of apples that remind after collecting full two apples in the net as many times as we could
  + Exp2: 5%3=2
    - 5 apples is what we have in the field
    - 3 apples is how much your net can take at once
    - 5%3 = 2. 2 is number of apples that remind after collecting full three apples in the net as many times as we could
  + Exp3: 10%3=1
    - 10 apples is what we have in the field
    - 3 apples is how much your net can take at once
    - 10%3 = 1. 1 is number of apples that remind after collecting full three apples in the net as many times as we could.
  + Exp4: 10%2=0
    - 10 apples is what we have in the field
    - 2 apples is how much your net can take at once
    - 10%2 =0 . 0 is number of apples that remind after collecting full two apples in the net as many times as we could.
  + Exp5: 10%20=10
    - 10 apples is what we have in the field
    - 20 apples is how much your net can take at once
    - 10%20 =10 . when the divisor is larger than the first number. Then the mod answer is equal to the first number
  + Exp5: 0%20=0
    - 0 apples is what we have in the field
    - 20 apples is how much your net can take at once
    - 0%20 =0 . when the divisor is larger than the first number. Then the mod answer is equal to the first number
  + Exp6: 10%0=undefined
    - Anything mod 0 is undefined. Imagine dividing by 0
  + Mod with negative numbers
    - When X is negative in X % N and N is positive
      * Find the highest multiple of N equal or lower than the target number, X.
      * Then ask, how much higher is X than that
      * Exp1 -11 % 5
        + The lowest multiple of 5 equal or lower than -11 is -15.
        + Therefore the answer is 4, because -11 is 4 times higher than -15.
      * Exp2 -5% 3
        + The lowest multiple of 3 equal or lower than -5 is -6.
        + Therefore the answer is 1, because -5 is 1 times higher than -6.
      * Exp3 -1% 3
        + The lowest multiple of 3 equal or lower than -1 is -3.
        + Therefore the answer is 2, because -1 is 2 times higher than -3.
      * Exp4 -3% 3 = 0
        + If the reminder is 0 then the final answer is 0
    - When N is negative in X % N and X is positive
      * this changes the direction of the “hand on the clock.” So, if X % N is M, then X % -N is -(+N-M), unless M is 0. In which case M is 0
      * . Exp1 11 % -5
        + X%+N = M -> 11% +5 = 1
        + +N-M -> 5-1 = 4
        + -(+N-M) = -(4) = -4
      * Exp2 5% -3
        + X%+N = M -> 5% +3 = 2
        + +N-M -> 3-2 = 1
        + -(+N-M) = -(1) = -1
      * Exp3 1% -3
        + X%+N = M -> 1% +3 = 1
        + +N-M -> 3-1 = 2
        + -(+N-M) = -(2) = -2
      * Exp4 3% -3 = 0
        + If the reminder is 0 then the final answer is 0
    - When both X and N in X % N are negative
      * Then treat is as of both X and N were positive and just add a (-) in front of their answer

**Circular Arrays – Run Times**

* The average run time it takes the computer to perform to a specific operation on fixed array are presented below:

|  |  |  |
| --- | --- | --- |
| Operation on a fixed array | Run Time | Explained |
| Insert an element at a random location | O(n)  Linear | At worst case scenario we would have to move every element in the array. Therefore, n operations |
| Insert an element at the front | O(1)  Constant | Due to usage of mod |
| Insert an element at the end | O(1)  Constant | Due to usage of mod |
| Delete an element from a  Random location | O(n)  Linear | At worst case scenario we would have to move every element in the array. Therefore, n operations |
| Delete an element from the front | O(1)  Constant | Due to usage of mod |
| Delete an element from the Back | O(1)  Constant | Due to usage of mod |
| Search for an element in an unsorted array | O(n)  Linear | The brute force method of checking each element must be used. This means at worst case scenario we need to check every single element in the array. Therefore, n number of operations |
| Search for an element in a sorted array | Log(n) | The array is sorted from small to large. Therefore we perform a binary search algorithm |
| access time(how long it takes to get to an element within an array). For EXP in Array x -> x[3] is how we can get to the fourth element | O(1)  Constant | The specific element that we are required we can get to it at a constant time |

**1.3 Dynamic Arrays**

**Dynamic Arrays – Introduction**

* A dynamic array allows you to increase the size of the array after you have fully filled its initial size. Unlike the fixed array who requires you to remove elements after its size has been filled
* **Method 1**: Copying the original array to a new array (minimum increase amount)
  + In this method, every time our array requires to get bigger we increase it by the minimum amount we are required to increase it by
  + The dynamic array allows you to insert more data than the size of your initial array, by copying the information from the current array to a bigger array and making that bigger sized array its new array
    - The reason behind this is that once an array is created it takes specific amount of memory and we can not go back and change that section of the memory. Therefore, dynamic arrays go behind the scenes and assign a new array and copy everything of the initial array to the new array, which requires more memory than the original array
  + The main issue with dynamic arrays in using this method is that when you insert an element that would change the size of the array. The run time of that is O(n) and not O(1). Because, that operation requires the copying each element of the initial array to a new array. Therefore, n number of operations
* **Method 2**: Copying the original array to a new array (2x increase amount), x is the length of the array (The more efficient way in comparison to method1)
  + In this method, every time our array requires to get bigger, we increase it exponentially. Therefore, we can decrease the number of times it needs to be increased in total
  + Each time we increase the size exponentially we are copying all the original data from the most recent sized array to an array that is 2x larger
    - The actual exponential copying process takes O(n) run time. However, because we are copying with such large sizes, we do not need to perform the exponential copying as much. Therefore, performing the addition of new elements will average out to be around O(1) over time
  + The reason why this method works faster than method 1 is because, following this method will decrease the number of times we need to increase our array.
    - The main benefits to the dynamic array using this method are that over time. The run time of adding elements will go to be around log(n) which is slightly above O(1), and we can use this method will give us a run time of O(1) or slightly more than that. This type of relationship is known as **amortized**
* Please note
  + we don’t care about decreasing the size of the array. Because if we have less elements than the set size of our array, we will not get an error. But if we try to have more elements than how much the length of the array can handle then we will be required to copy our array to a new array within the memory
  + For this, we use the terms **logical size**, and **physical size**. **The** **logical size** is how many pieces of data are actually allocated in the array. **The physical size** is the size of the array itself. So, what we want to do, is every time our **logical size** reaches our **physical size**, we want to double the **physical size**.

**Dynamic Arrays – Run Times**

* A dynamic array, similar to fixed array run time

**2.1 Approximately O(1) “Constant” run time from O(n) “Linear”**

* Commonly in CS we approximate a run time to be O(1) even though it is slightly larger.
* EXP, in cases when the run time is equal to the average of adding few O(n) and many (1). Then our run time will be slightly larger than O(1) but we will approximate it to be O(1).
* In cases when you have a set pattern. For EXP, for each O(n) there are two O(1). Then your run time will be O(n). Therefore, unless the amount of data of O(1) can significantly exceed the amount of data for O(n). Then, we can not approximate our run time from O(n) to O(1)

**2.1 A Dynamic Circular Array**

* This array would just be a dynamic array with front and end cursors. This would allow you to delete and add in constant time, and to make sure that when the array runs out of room it wouldn't take up much time either.