Array

Array is a collection of same data type stored in continuous memory location. We can access the elements using index and by using index we can able to perform all manipulations like add, delete, sort and more. The index is always starts with 0. In arrays we also have 2-D array and 3-D array as well.

There are two types of arrays they are **static** arrays and **dynamic** arrays. Static arrays are we have to initialize the size at the compile time itself and we can't able to increase the size later. Whereas the dynamic arrays are vice versa, we can add new items into the array even it is full, it will automatically double the size.

To access an element in an array at any index the time taken is O(1). It is because first it will get the base index address and then using that we can able to get the address of required and index.

address of i^{th} element = base index address + offset address (i^{th} * size of the data type).

Linked List

Linked List is the linked representation of list, linked list is made up of nodes which contains data and link. It has three types of linked lists. The values in linked list won't store in sequential manner but still they are connected through link.

1) Single Linked List: It can only move forward can't move back, it contains data and link which is next node memory location/address and also the first node address will be stored in a pointer. And the last node will have null value in the link it will represent that it is the last element present.
2) Double Linked List: It can move forward and backward within the data, it consists of three parts they are previous node address, data and the next node address. The first node will have null value in the previous node address so that it represents as first element and the last node will have null value in the next node address to represent it as last element.

3) Circular Linked List: It is almost similar to the double linked list but the only change is that the next node address in the last element won't have null it will have the first node address so that it can form a circle.

Comparison & Contrast between Array and Linked list

- The one of the major difference between arrays and linked list is that size, if we declare array size we can't increase the size, whereas in linked list we can keep on increasing and adding new elements we doesn't have to mention the size of the linked list prior.
- In linked list there is no wastage of space whereas in array sometimes we have to initialize the array with larger size and we won't be using much space as we have declared and the rest of the space will be useless.
- We can access the elements in arrays faster than linked list.
- In linked list there is an wastage of memory for storing the address of next node, but not like that in arrays.

- Insertion/deletion of elements in middle of the array takes longer than linked list.
- Arrays are stored in contiguous memory location but the linked list are stored in non-contiguous memory location.
- Elements in array are accessed by using index whereas in the linked list we have to traverse sequentially from the head node.

Comparison:

Access time for elements in array is O(1)	Access time for elements in linked list is O(n)
Insertion/Deletion of an element in middle of the array is O(n).	Insertion/Deletion of an element at end O(1) and in middle O(n) of the linked list
Simple to implement	More complex and difficult to implement
Fixed size and not flexible	More flexible and increase the size dynamically

Stack

First In Last Out (FILO), if you are adding an element you have to add on top of the first element and also if you want to access the first element you have to first remove the all elements which are on top of it. The insertion and deletion will be done only on the end. It is an linear data structure. It is also mainly used for recursion. The main operation in stack are:

Pop(): It will delete the last added item in the stack.

Push(): It will add the new element at the end of the stack.

Top(): Without deleting the element, it will provide the value of the last added element in the stack.
Size(): Returns the size of the stack.

isFull(): It returns true if the stack is full and we can't add any new element else returns false.

isEmpty(): It returns true if there are no elements in the stack else false.

Queue

First In First Out (FIFO), it has two open ends like a tube, the first entered element will be removed first. It has two endpoints as well head and tail. It also have various characteristics such as fast and flexible, it can handle multiple data since we can access both ends. Various operations can be done using queue are:

Enqueue(): To add new element at rear.

Dequeue(): To delete the element from front.

Peek(): To get the value at the front without deleting.

Rear(): to get the element from rear/back/tail without deleting.

isFull(): It returns true if the queue is full and we can't add any new element else returns false.

isEmpty(): It returns true if there are no elements in the queue else false.

Types of queues are as follows:
Input restricted queue: inserting element is restricted to only one end and can delete on both the ends.

Output restricted queue: deleting element restricted to one end but inserting can be done on both ends.

Double ended queue: both insertion and deletion can be done on both ends.
Circular queue: both the ends are connected to each other, but still maintains the FIFO order.

Priority queue: It prioritizes the data and will create the queue accordingly, larger the value higher the priority.

Among all of these types priority queue is most important because in order to create heap memory. The whole heap memory is based on this concept itself.

Hash Tables

Taking input as a large number or string and converting them by doing modulus operation and get the smaller value also called as index of the element to store in the hash tables is known as hash function. The insertion, deletion and searching of an element takes O(1). It is more compatible and also uniformly distributes the keys as well. In order to overcome a scenario where the hash function provides same result for more than one input value this is called collision, it might cause an issue that's why it also has various types as follows:

Separate chaining: If we face any duplicate values during insertion we will create an extra block for the new value and insert that key value within the same index, which means increasing horizontally for that particular index. There are few advantages using this method easy to implement and we can keep on add the elements and also less sensitive to the hash functions. It also has disadvantages as well they are by following this method we have to waste lot of space and also sometimes the table won't even full, and the main thing is that retrieve the element from the index which has added rooms will take lot of time. The time complexity of this particular method is O(1+(n/m)).

Open Addressing: It is also collision resolving technique, all keys will be stored within the table itself. The hash function will specify the order of slots to probe the elements in just only one slot. There are various types of probing as well:
Linear Probing: In this technique we will also add what is the iteration of hashing like whether it is first time or second time of performing hash function this value is also added to the key value and then we will perform mod operation with the table size by doing this even if we had face an collision we can iterate one more time hash function then we can able to get the different key value by doing

this we can able to insert the values, by performing the same we can able to delete and search elements by matching the result key value and the hash function value as well.

$$h_i = (H(x) + i) \text{ mod table size}$$

Quadratic Probing: It is almost similar to linear probing but the only difference is that instead of adding the iteration of hash function value it will square the time of iteration and then will add it to the key value.

$$h_i = (H(x) + i^2) \text{ mod table size}$$

Double Hashing: It is also similar to linear probing only but instead of one hash function it will be having two hash functions.

$$h_i = (H(x) + I * H_2(x)) \text{ mod table size}$$

Among all these three opens addressing collision resolving techniques, linear probing is very simple and easy to implement, the average probing has better clustering than linear probing, but the double hashing requires more computation time and bad performance. Overall, the time complexity is far better than the separate chaining.

Divide & Conquer

By using recursion we will implement this divide and conquer approach we will divide the whole array by 2 until all the elements are split up and then we will perform the operation by paring up based on how we have divided them.

Lets say there is an array of 8 elements and find the maximum number, first we will divide by 2 then we will have two segments each will have 4 elements, then we will divide more further and we will be having 2 elements and then we further divide again and all the elements are separated and later we perform the operation like we will compare the values between two elements and then will assign to the previous level and we keep on doing until it reaches one element which is our result.

Merge sort, quick sort and binary search are using this algorithm.

Fractional Knapsack Problem

This algorithm is try to fit as possible we can in an assigned size array. It is also works based on the weight and value ratio. Lets say we have an capacity which can weighs of total w=50 and we have three bags A 10, B 20, and C 30. So instead keeping just 2 and 3 itself we will try to fit all three bags as possible we can like we can fill whole A and B and we still left with C we can't fit all the C weight instead we can able to fit up to 2/3rd value of C which means 20, by doing this way we can able to fit all the three of the elements instead of just 2. The time complexity of this algorithm is O(nlogn).

Graph Traversals (Breadth-First Search, Depth-First Search)

Breadth-First Search:

This algorithm uses queue implementation and will visit all nodes and make sure it hasn't left any node without visiting. First we have to select any node and from that it will check for any adjacent nodes and will visit all the adjacent nodes once all are completed then it will give the control to one of the adjacent nodes and will search for the adjacent nodes again and will traverse to all the adjacent nodes till it can't find any adjacent nodes and will return the resulting nodes.

Depth-First Search:

This is almost similar to the BFS, the only difference is that it won't stop at checking the adjacent nodes itself it will try to finish traversing to all the leaf nodes of left side first after completing left side it will traverse in the right side later. And also DFS uses stack implementation.

Differences:

BFS will try to get the shortest and shallowest path, whereas DFS will not guarantee the shallowest path and it won't give shortest path.

BFS uses more memory than DFS.

BFS uses queue(FIFO) implementation, DFS uses stack(FILO) implementation.

BFS used in unweighted graphs, p2p networks and also uses in web crawlers. DFS used in scheduling the tasks such as data serialization and compilation task order and also used for solving puzzles like maze it will travel to each path and find outs the solution.

Dijkstra's & Bellman-Ford

These two algorithms was introduced to find the shortest path bet ween the source and destination. Let's take Directed Acyclic Graph, Dijkstra's algorithm will make all nodes as infinite and later makes source node as 0 and will check for the predecessor and will evaluate all the possible paths and will move accordingly once it did that and it will also evaluate the distance of all the nodes and will store them in Min-heap priority queue. After moving to the nearest node then again it will remove them from the predecessor and in queue list later it will check for the predecessors again and will evaluate the shortest path and will goes on till it find the shortest path to the destination. The one major flaw in it is that we can't able to visit the node again which is already visited and also it can't handle the negative weighed graphs as well.

In order to over come this problems Bellman and ford discovered this algorithm named after them it is Bellman-Ford algorithm, it is almost similar to the Dijkstra's it will make all the nodes infinite and make the source node as 0 and will make a list of predecessor's and then it will count the number of sweeps as 1 and later we will also check that did we update any nodes or not, by counting the nodes we will get to know that we have visited all nodes like if the number sweeps is n-1 except the source node it means we have visited all the nodes, if we have exceed the sweep count than number of nodes that means we were in a negative cycle which is keep on going in order to aware of that we are using the sweep count and if we haven't update any nodes that is also a check to us that we have reached the destination. By using these two algorithms this is how we will find the shortest path between the source and destination.

Greedy algorithm: This algorithm is used to find the minimal or maximum possible result and it also has short execution time. It follows the problem-solving paradigm to select the optimal solution of the choices to provide the better result.

Sorting

The sorting is mainly used in various applications like computer graphics, data comparison and so on. There are various types of sorting available and each one of the sorting will be used according to the inputs and according to the problem.

Insertion sort: This is a sorting algorithm which uses pair wise swap technique, let's say we have an array and we need to sort it so first we will take the element in the 1^{st} index and we will compare it with the 0^{th} index and if it is less than then we will swap the elements like this we will compare the all indexes and we will swap the numbers accordingly. But it is costly it takes a greater number of swaps such as $O(n^2)$ swaps. If we want to find an element using this sort it will take more time which means O(n) time complexity instead of that if we use binary search in this sorting algorithm it will be very quick and provide of O(logn) time complexity.

Merge sort: This algorithm uses the divide and conquer approach, first it will divide the array into two parts till it reaches individual elements once it did that then it will compare with the next element and combine them accordingly and will keep on doing until it is sorted. The time complexity of this is O(nlogn). During this sorting most of the work is always done in the leaves itself that is why it requires O(n) space.

The one major difference between insertion sort and merge sort are that the auxiliary space for insertion sort is O(1) which is less than merge sort which is O(n).

Binary search: If we want to find an element using this algorithm, first we will sort the array and then later we will divide the array by n/2 and we will evaluate and check on which half it might be present and will use that half and further divided till we find that element.

Bubble sort: This is the initial and very basic sorting algorithm, in this algorithm we will two pairs and check the elements which one is greater and will swap accordingly, it will be sorted in ascending order. The number of times completed the whole iteration is called as pass and the total number of iterations will be considered as using n = n-1 (n = number of elements). Lets say we have 5 elements and on the first pass we will check i and i+1 index and if the i+1 index is lesser than i index then we will perform swap, we will perform the same operations till we sorted all elements.

Binary search tree: In this algorithm we will select one parent node and we will have all the lesser elements than the root node will be placed in left side and all the greater elements will be placed on right side of the node, by doing this insertion and deletion will be very quick.

Searching: The searching is nothing, but we have to find the target element within the provided elements, this have two types of searching sequential search and interval search. The sequential search is searching for elements in datatypes such as array, linked list and etc., eg linear search algorithm. The interval search is a technique where we will divide the elements to half till it reaches

the target element, it will be applied of data types such as graphs, trees and etc., eg., binary search algorithm.

Linear search: It is the simple and basic sequential search algorithm, it is used to search it we have less data, for large set of data it will take so much time. First we will start from the 0^{th} index and we will try to match the target element with the i^{th} index element if it matches we will stop the search otherwise we will move to the next element. The worst case time complexity of this algorithm is O(n).

Bucket sort: On this algorithm first we will take an array of elements and then we will calculate the number of buckets required by multiplying the size of array and then we will have the number of buckets and we will sore the elements within the respective buckets and we will sort the elements in the bucket by using any sorting algorithms, once we have sorted all the buckets then we will combine the buckets sequentially and we will get the sorted list.

Counting sort: This sorting algorithm only works on only positive integers, and it is not an comparison based algorithm. On the given set of array we will first find the largest element in the array and the we will create an array for count and we will store the cumulative count in that particular array and once we did that after we will traverse to the index of the element in the count array and by using that count value we will reduce by one and store that in the output array. It is also often used in radix sort as well.

Radix sort: The radix sort is also an non comparative algorithm, this algorithm works like first we will compare the 0's place and we will sort accordingly, later we will compare the elements in 1's space and we will sort again, like this we will keep on doing till we sort according to the highest number.

Heap sort: On this algorithm we will take the binary tree and will have the max heap after that we will compare the elements and make sure all the elements which are smaller will be store in left and greater will be on right and once we did that we will increase the level and we will keep on doing that until we fill up each level except that last level.

Quick sort: This algorithm also uses the divide and conquer rule, first we will select the pivot it can be first, last or middle element. Once we selected the pivot we will move all the elements which are less than the pivot will be moved to left and this will also continued on the right side of the pivot as well, we will select one pivot on the right side of the array and we will sort again. We will keep on doing till it is sorted.

Selection sort: On this algorithm we will divide the elements into two parts such as unsorted and sorted array, first we will select the smallest element and we will move to the left most and we will keep on doing that there will be no elements in the unsorted array. Once we moved one element to left we will move to the next element.

Shell sort: This is based on the insertion sort, first we will take an array and will try to create edges and by using those edges we will create sub-lists and we will apply the insertion sort algorithm on those sub-lists till the edges reaches to 1. This was introduced because to avoid the insertion sort

fault where we have to do the large shifts between the elements. In order to avoid that this has been introduced.

Problems faced & solutions:

Leetcode problem 203:

I tried to solve 203 problem where we have to remove the elements from the linked list, I was unable to figure out how to skip to the next value by avoiding the element to remove.

Once I understand the logic like how I want to implement and understanding the linked list working, then I was able to skip to the next element and most of the time the list got shorter like removing additional elements as well, later debugging the code then I got to know where the logic went wrong and fixed it again. I was actually skipping the required values as well and removing all the data.

Leetcode problem 83:

After solving the above problem it is almost the same problem instead of deleting all the duplicate elements we will be removing all duplicates. Modified the logic like we will check the current value to the next value if they are matching we will skip the step otherwise we will do regular traversal.

Leetcode problem 1:

This was simple problem haven't faced an issue, I have make sure that if we add two indexes we will able to get the target value and also make sure that the indexes are not same and once we found the two elements to get the target value we have returned the indexes.

Leetcode problem 66:

This was tricky problem, I first tried solving by converting the whole array into digit and did the increment and then tried to convert that digit into int array. But I was able to achieve till increment part but after that converting that into array was very tricky. So I figured out another approach like traverse the array from the last and make sure the digit is less than 9 and did the increment if its 9 we replace it with 0 and incremented the next digit as well. Later I have also covered what if it has only one digit and did the increment as well.

Leetcode problem 27:

I have taken two indexes one will traverse the whole array and one will increment only when the element is not the one required by doing that we can able to get the count and I have returned that count.

Leetcode problem 20:

First I have created an empty stack and then later checked the input is null or not, then traversed the string by converting it into characters and placed closing bracket for every opening bracket in stack and by verifying the stack whether we have balanced parenthesis we will return the output. In order to come up with this solution I tried push everything into the stack but by doing that I was unable to verify. Later I tried counting as well but I was able to get the counting of parenthesis, but I was not sure how to return the expected output. Later I came up with this solution by having only closing tags in the stack so that we will know whether there is an unwanted closing tags or correct pair of tags are there or not.

Leetcode problem 169:

In order to get the element which appears more than n/2, first I tried counting all the elements occurrences and returned the highest one it worked, but I have observed that what if we sort the array and return the n/2th value because we have to find the n/2 occurrences and it worked every time if the array is sorted the highest occurrence value is present at this index only which is n/2.

Leetcode problem 88:

To solve this problem I thought of storing the values of num1 array first in one array and after that store the num2 array values but it will take additional space and it is not that effective, so I thought of using the first array and will store the num2 array values by doing that we won't have to create an additional array and it will be more efficient.

Leetcode problem 217:

For this problem initially I thought of checking each element and store the count in a map, but it became more code and it was complicated, so instead of that I have sorted the array first once it was done most of the work was done and we just need to check the next element if it matches we do have duplicate elements and returns true, if not we will return false.

Leetcode problem 349:

While trying to solve this problem, I thought of using one loop and will compare the elements but I realized that by using only one loop I will compare only just one element not the whole elements so then I used two for loops finding out the duplicate elements but storing them was a difficult one I thought of using the array but I have to know the size of the array and I thought of using among the two arrays which has highest size to use that one but later realized by doing that we will be wasting so much of space and it is not that efficient and used list and by doing that I was also able to make sure we won't store the duplicates in the list as well later I have converted the list to array and return the resultant array.