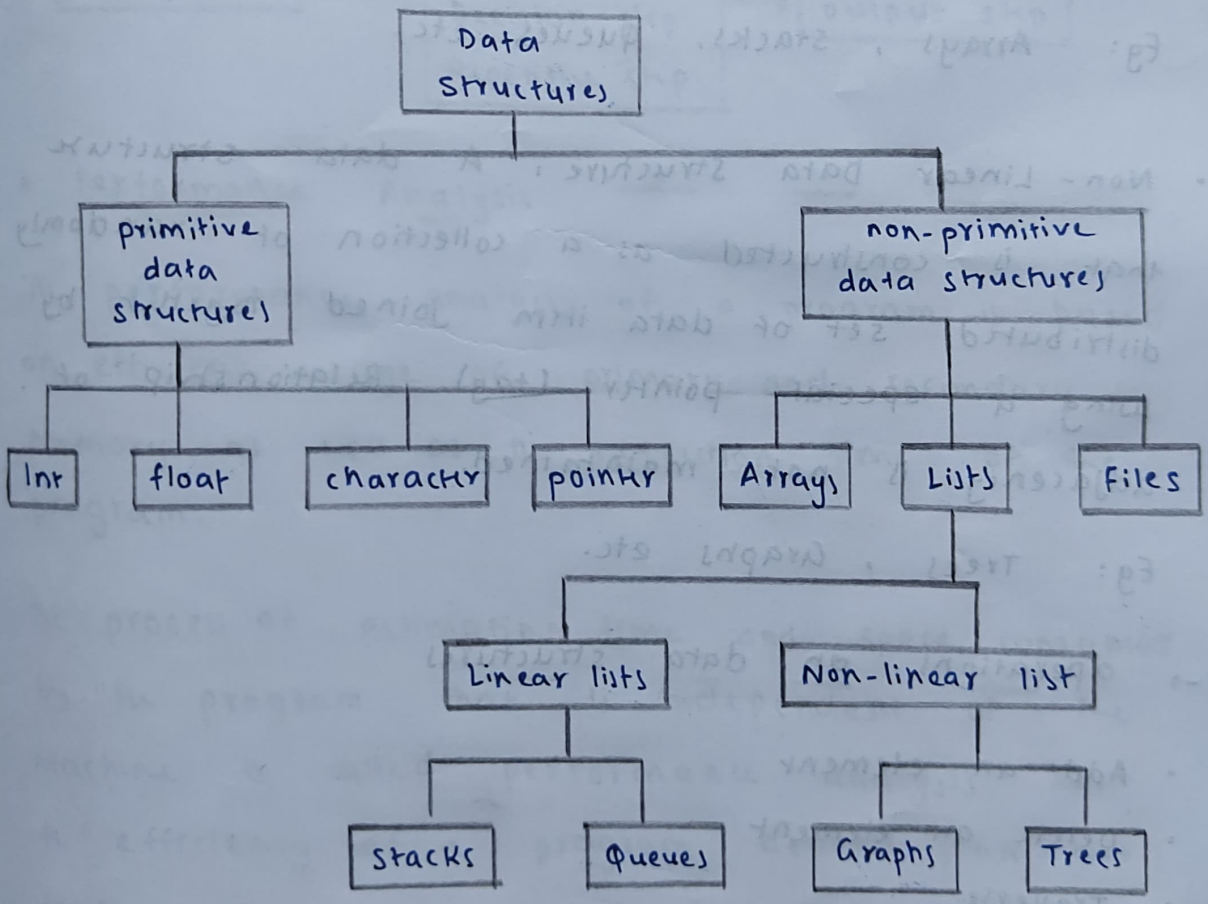
M-1 Introduction to Data Structures, searching & sorting

* Data Structures

A data structure is a way of organizing and storing data in a computer so that it can be accessed and used efficiently. It refers to the logical or mathematical representation of data, as well as the implementation in a computer program.

→ classification of data structures



Primitive data structures : are known as basic data structures, These data structures are directly operated by machine instructions. They have different representations on different computers.

Non-primitive data structures : are highly developed complex data structures. Basically, these are developed from the primitive data structures. These are responsible for organizing the group of homogenous and heterogenous data elements.

- Linear Data Structure : A data structure in which data elements are arranged sequentially or linearly in a continuous arrangement where each element is attached to its previous and next adjacent elements.

  Eg: Arrays, stacks, Queues etc.

- Non-Linear Data Structure : A data structure that is constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). Relationship of adjacency is not maintained.

  Eg: Trees, Graphs etc.

→ Operations on data structures

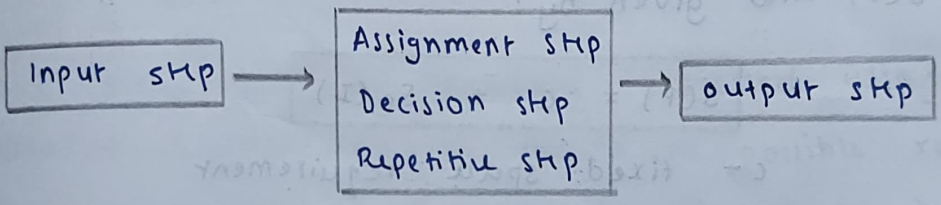
- Add an element
- Delete an element
- Traverse
- Sort the elements
- Search for a data element
- Update an element
- Merging

* Algorithm

An Algorithm is a set of finite rules or instructions to be followed that aim to solve a particular problem.

The word 'Algorithm' originates from the arabic word 'Algorism' which is linked to the name of the Arabic Mathematician Abu Abdullah Muhammad ibn Musa Al-khawarizmi.

→ Structure of an algorithm



* Performance Analysis

The performance analysis of a program is based on efficient usage of primary and secondary memory as well as the execution time of the program.

The process of estimating time and space consumed by the program that is independent of the machine is called performance analysis. So, the efficiency of a program depends on two factors:

- Space complexity
- Time complexity

→ Space Complexity

Space Complexity is the total amount of memory space used by an algorithm or program, including the space of input values, for execution.

To find space complexity, it is enough to calculate the space occupied by the variables used in an algorithm / program.

The space requirement S of a program P is denoted by S(P) and given by:

$$S(P) = C + S_P(I)$$

C - fixed space requirement

I - variable space requirement

Space complexity = Auxiliary Space + Space used by input values

Note: The best algorithm / program should have the least space complexity possible. Lesser space used will result in faster execution.

Big O Notation	Space complexity details
O(1)	Constant space complexity. Program doesn't contain loops, recursion, or calls to other functions.
O(n)	Linear space complexity. Program contains loops or recursion.

→ Time Complexity

The amount of time taken by an algorithm as a function of the length of the input to solve or run given problem is called the time complexity of the algorithm.

To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

- Big O Notation

Time complexity is frequently expressed using Big O Notation. It represents the maximum possible running time for an algorithm given the size of the input.

O(1) - constant time complexity

If an algorithm takes same amount of time to execute no matter how big the input is, its called to have constant time complexity

Eg: `c = a + b` complexity : O(1)

O(n) - linear time complexity

Running time is directly proportional to input size.

Eg:
```
for i = 1 to n:
    x = x + 2
```
$3n+1$ $O(n)$

O(n^2) - Quadratic time complexity

Eg:
```
for i = 1 to n:
    for j = 1 to n:
        x = x + 1
```
$3n^2 + 3n + 1$ $O(n^2)$

* Asymptotic Notations:

→ Big oh (O): $f(n) = O(g(n))$, if there exists a positive integer n_0 and a positive number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$.

$f(n)$	$g(n)$	
$16n^3 + 45n^2 + 12n$	n^3	$f(n) = O(n^3)$
$34n + 40$	n	$f(n) = O(n)$
50	1	$f(n) = O(1)$

Here $g(n)$ is upperbound of the function $f(n)$.

→ Omega (Ω): $f(n) = \Omega(g(n))$, if there exists a positive integer n_0 and a positive number c such that $|f(n)| \geq c|g(n)|$ for all $n \geq n_0$.

$f(n)$	$g(n)$	
$16n^3 + 8n^2 + 2$	n^3	$f(n) = \Omega(n^3)$
$24n + 9$	n	$f(n) = \Omega(n)$

Here $g(n)$ is lowerbound of the function $f(n)$.

→ Theta (Θ): $f(n) = \Theta(g(n))$, if there exists a positive integer n_0 and two positive constants c_1 and c_2 such that $c_1|g(n)| \leq f(n) \leq c_2|g(n)|$ for all $n \geq n_0$.

$f(n)$	$g(n)$	
$16n^3 + 30n^2 - 90$	n^2	$f(n) = \Theta(n^2)$

Here $g(n)$ is both an upper bound and a lower bound for the function $f(n)$ for all values of n, $n \geq n_0$.

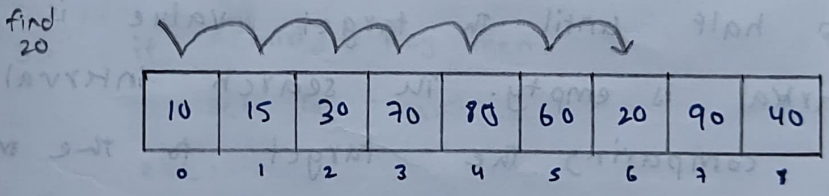
* Searching Algorithms :

Searching is the fundamental process of locating a specific element or item within a collection of data. The primary objective of searching is to determine whether given element exists in a data structure and to find its precise location.

→ Linear Search

In this technique, the key is compared with each element in a given array sequentially until the desired element is found.

If the key is found, position is retrieved. If the key is not found, the function returns 0 or -1. This technique is usually used to search in an unordered list.



Algorithm :

1. Read array A of n elements and key.
2. flag = 0
3. for i in range(n)
   if A[i] == key
   flag = i
   return flag
4. if flag
   print - search is successful
   else
   print - search is unsuccessful

Complexity of linear search :

For linear search, we need to count the no. of comparisons performed but each comparison may or may not search the desired item.

case	best case	worst case	Avg case
if item is present	O(1)	O(N)	O(N/2)
if item is not present	O(N)	O(N)	O(N)

→ Binary Search

Binary search is a search algorithm used to find the position of a target value within a sorted array. It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty. The search interval is halved by comparing the target to the middle value.

Logic of binary search :

- Divide the search space into two halves by finding the middle index 'mid'.
- Compare the middle element of the search space with the key.
- If the key is found at middle element, process is terminated.
- If the key is not found at the middle element, choose which half will be used as the next search space.

- If the key is smaller than the middle element, then the left side is used for next search.
- If the key is larger than the middle element, then the right side is used for next search.

- The process is continued until the key is found or the total search space is exhausted.

Algorithm:

Step 1: low = 0, high = n-1, mid = (low + high) / 2

Step 2: repeat step 3 and step 4 while low <= high and A[mid] != key

Step 3: if item < A[mid], then
set high = mid - 1
else
set low = mid + 1

Step 4: set mid = $\frac{low + high}{2}$

Step 5: if A[mid] = key then:
set loc = mid
else:
set loc = NULL

Step 6: exit

Complexity of binary search:

each comparison eleiminates half items of the list.
after first comparison N/2 then N/4 then N/8 so on.
no. of comparisons $\Rightarrow$ $N/2^i = 1$ $\Rightarrow$ $i = \log n$ $\therefore O(\log n)$

case	best case	worst case	Avg case
if item is present	1	O(logn)	O(logn)
if item is not present	O(logn)	O(logn)	O(logn)

→ Uniform Binary Search O(log n)

Uniform Binary search is an optimization of Binary Search algorithm when many searches are made on same array or many arrays of same size. In normal binary search, we do arithmetic operations to find the mid-points. Here we precompute mid points and fills them in look up table. This works faster than addition and shift to find the mid point.

→ Interpolation Search O(log log n)

Interpolation search is a searching formula that uses an interpolation formula to estimate the position of a target value in a sorted array. Unlike binary search, selecting the middle element, interpolation search makes a more intelligent guess based on distribution of data.

$$pos = lo + \left[\frac{(x - arr[lo]) * (hi - lo)}{(arr[hi] - arr[lo])} \right]$$

→ Fibonacci Search O(log n)

As the name suggests, the Fibonacci search algorithm uses the fibonacci numbers to search for an element in a sorted input array.

$$F_n = F_{n-1} + F_{n-2}$$

0, 1, 2, 3, 5, 8, 13, 21, 34 ...

* Sorting Techniques :

Sorting in general refers to various methods of arranging or ordering things based on criteria's (numerical, chronological, alphabetical etc.).

There are many approaches to sorting data and each has its own merits and demerits.

→ Bubble Sort

This sorting technique is also known as exchange sort, which arranges values by iterating over the list several times and in each iteration the larger value sinks down and lighter values bubble up.

• Logic :

We go through the array of data elements by comparing consecutive elements, if the top or prior element is greater than the lower or subsequent element then they are swapped otherwise no change occurs. This complete iteration is called a pass. We perform multiple passes until the list is sorted.

A

8	5	7	3	2
0	1	2	3	4

n = 5

8	5	5	5	5
5	8	7	7	7
7	7	8	3	3
3	3	3	8	2
2	2	2	2	8

1st pass
4 comparison

5	5	5	5
7	7	3	3
3	3	7	2
2	2	2	7
8	8	8	8

2nd Pass
3 comparison

5	3	3	
3	5	2	
2	2	5	
7	7	7	
8	8	8	

3rd Pass
2 comparison

3	2		
2	3		
5	5		
7	7		
8	8		

4th pass
1 comparison

No. of passes = n-1 = 4

• Algorithm:

Step 1 : Repeat steps 2 and 3 for i = 1 to n-1

Step 2 : Set j = 1

Step 3 : Repeat while j <= n-i-1 :

(A) if a[i] < a[j] then :

interchange a[i] and a[j]

(B) set j = j+1

Step 4 : exit

• Time Complexity :

The time complexity of an algorithm depends on the number of comparisons ; iterations or swaps. In Bubble sort, given an array of size n, we have n-1 passes. First pass has n-1 comparisons, second has n-2 comparisons, third n-3 and so on.

So, total comparisons = $1+2+3+4...+n = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$

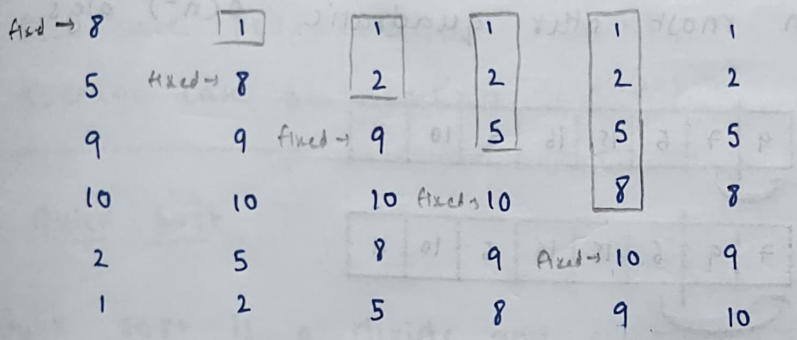
∴ time complexity is :

Worst case scenario	$O(n^2)$
Best case performance	$O(n)$
Average case performance	$O(n^2)$

→ Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (Asc. order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array : sorted subarray & remaining sub array.

In every iteration of selection sort, the fixed element is compared with all elements and swapped if greater, giving the minimum element after the iteration which is then moved to the sorted subarray. The fixed pointer moves down after every iteration or pass.



- Algorithm

```
for i = 1 to n-1
Set : minIndex = 1
for j = i+1 to n
    if arr[j] < arr[minIndex]
        Set : min_Index = j
Swap (arr[i], minIndex)
Exit
```

- Time complexity

Selecting the lowest element requires scanning all n elements (this takes n-1 comparisons) and then swapping it into first position. scanning next lowest requires n-2 comparisons. $\Rightarrow$ $n-1 + n-2 + n-3 + \dots + 2 + 1 = n(n-1)/2$

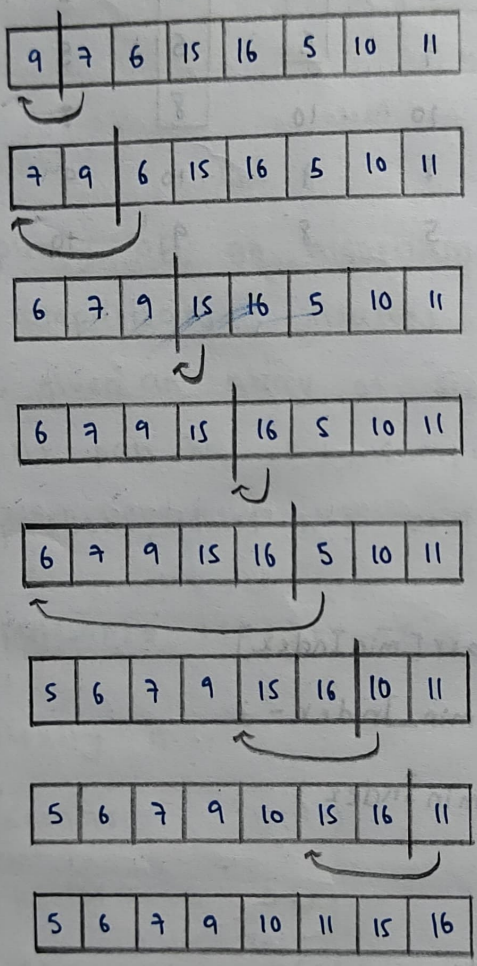
Worst case scenario $O(n^2)$

Best case performance $O(n^2)$

Average case performance $O(n^2)$

→ Insertion Sort

In this algorithm, the array is visually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part. It has simple implementation. Efficient for small data sets. More efficient in practice than most other quadratic $O(n^2)$ algs.



• Algorithm

Step 1: Iterate from arr[1] to arr[n] over the array.

Step 2: Compare the current element (key) to its predecessor.

Step 3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

. Time complexity

If there are n elements to be sorted. Then, this procedure is repeated n-1 times to get sorted list of array.

∴ Insertion sort time complexity is:

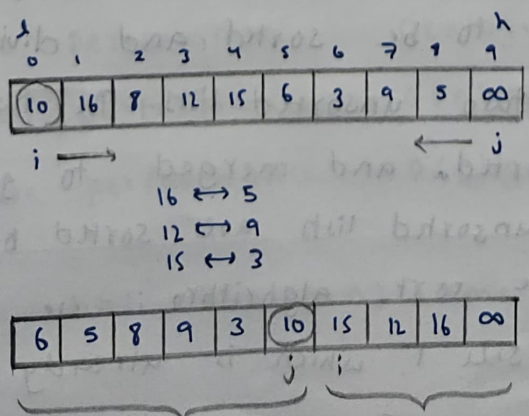
Worst case performance	$O(n^2)$
Best case performance	$O(n)$
Average case performance	$O(n^2)$

→ Quick Sort

Quick sort is a divide and conquer algorithm which means it divides an array into subarrays of low and high elements and then recursively sorts the subarrays.

It uses partitioning which takes an element (low) and finds its correct position by iterating two variables i and j in opposite directions and swapping A[i] and A[j] when A[i] > pivot and A[j] < pivot. When i > j, the j is the position of pivot after sorting.

Then the left and right hand side of pivot undergo quick sort to arrange the elements.



- Algorithm

Step 1: Consider first element as pivot a[low]

Step 2: Initialize i to low index and j to high index

Step 3: Repeat the following steps until i<j

a) keep incrementing i while a[i] ≤ pivot

b) keep decrementing j while a[j] > pivot

c) if i<j then swap(a[i], a[j])

Step 4: If i is j then swap(a[j], pivot)

j is the position of pivot

Then perform quick sort again on divided sub arrays.

- Time Complexity

On average make O(nlogn) comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons. Quick sort is often faster in practice than other O(nlogn) algorithms.

Worst case performance	$O(n^2)$
Best case performance	$O(n \log_2 n)$
Average case performance	$O(n \log_2 n)$

→ Merge Sort

Merge sort is based on divide and conquer method. It takes the list to be sorted and divide it in half to create two unsorted lists. The two unsorted lists are then sorted and merged to get a sorted list. The two unsorted lists are sorted by continually calling the merge sort algorithm; we eventually get a list of size 1 which is already sorted. Then 2 lists of size 1 are then merged.

It takes the array, left most and right most index of the array to be sorted as arguments.

Middle index (mid) of the array is calculated as (left + right) / 2.

Check if (left < right) cause we have to sort only when left < right because when left = right it is anyhow sorted.

Sort the left part by calling MergeSort() function again over the left part MergeSort (array, left, mid) and the right part by recursive call of MergeSort (array, mid+1, right). Lastly merge the two arrays using the merge function.

- Algorithm

If $l < r$:

Step 1 : Find the middle point to divide the array into two halves, $m = l + (r-1)/2$

Step 2 : Call merge sort for first half : ms(arr, l, m)

Step 3 : Call merge sort for second half : ms(arr, m+1, r)

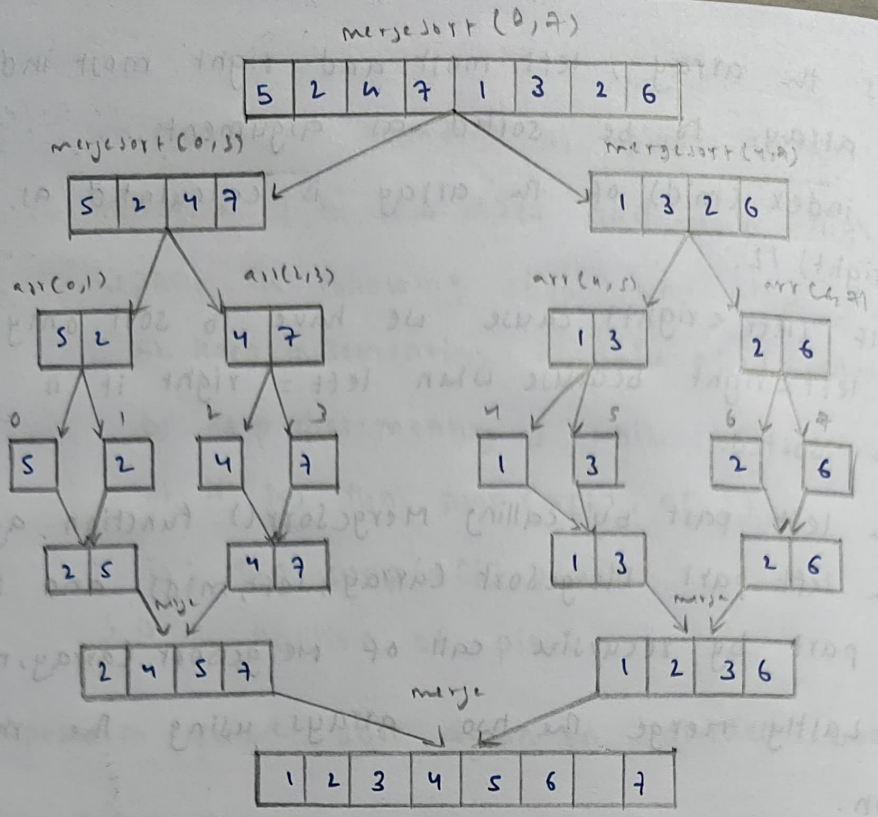
Step 4 : Merge the two halves sorted in step 2 and 3 :

Call merge (arr, l, m, r)

- Time Complexity

Worst case performance	$O(n \log_2 n)$
Best case performance	$O(n \log_2 n)$
Average case performance	$O(n \log_2 n)$

Merge sort has $O(n \log_2 n)$ time complexity.



* Comparison of Sorting algorithms

Sorting algs	Time Complexity			Space C.
	Best case	Avg. case	Worst case	Worst case
Bubble Sort	$O(n)$	(On^2)	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quick Sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n)$
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$