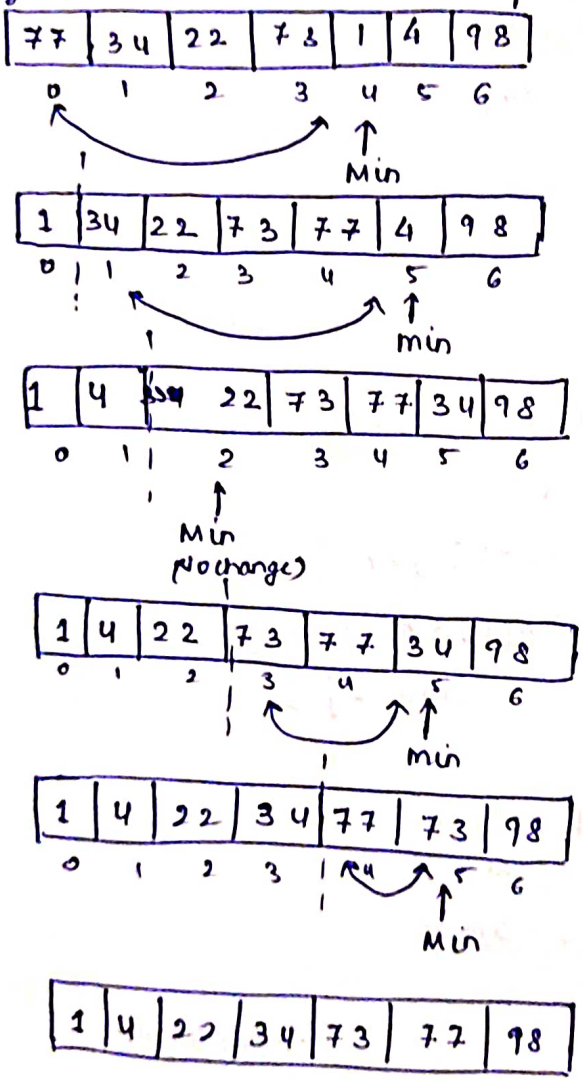
NAME : Ananya Prasad
REG NO : 20BCE10093

CODE/SLOT: CSE2002 / C11+D11
FACULTY : Ms. Meenakshi Choudhary.

(a)1) Selection sort

* It uses the brute force approach.
* It is a Inplace sorting algorithm : It takes constant amount of extra memory. Amount of extra memory is directly proportional to the size of the array.
* PRINCIPLE : Sorts the elements in an array by finding minimum element in each pass from unsorted part and keeps it in the beginning. This divides the array into sorted and unsorted part.

RUN ⇒



→ Sorted array.

* Select the minimum value and swap it from the first element of the unsorted array.

Pseudocode⇒

```
Selsort (A, n)                    → all are sorted by penultimate loop.
{       for  i ← 0 to n-2
        {    imin ← i
             for j ← i+1 to n-1
             {    if (A[j] < A[i Min])
                       iMin ← j
             }                          → swap
             temp ← A[i]
             A[i] ← A[iMin]
             A[iMin] ← temp
        }
}
```

PROGRAM

```
Void selsort (int A[], int n)
{       for (int i=0 ; i<n-1 ; i++)
        {    int iMin = i;
             for (int j = i+1 ; j<n; j++)
             {    if (A[j] < A[iMin])
                       iMin = j ;
             }
             int temp = A[i];
             A[i] = A[iMin];
             A[iMin] = temp;
        }
}
int main ()
{       int A[] = {7, 34, 22, 73, 1, 4, 98}
        selsort (A, 7)
        for (int i =0; i<7 ; i++)
 }              cout << A[i] << " ";
```

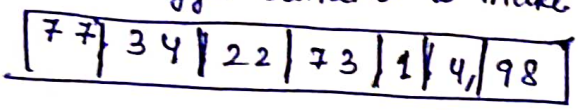
OUTPUT ⇒ 1 4 22 34 73 77 98 (Time Complexity = $O(n^2)$

(b) Insertion sort

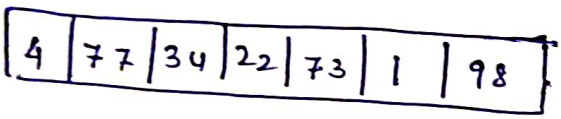
* Not the best sorting method, but better than selection and bubble sort.
* Uses brute force approach.
* RUN →

Array iterates from arr[1] to arr[n]
Compare the current element with its predecessor.
If the element is smaller, compare it to elements before.
Move the bigger elements to make room for swapped elements

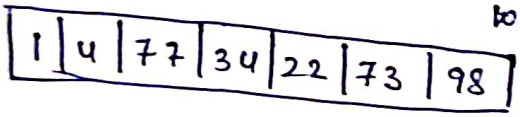
eg



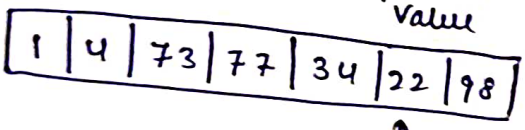
↑
Value (stored) → creates a hole (Any value can be chosen)



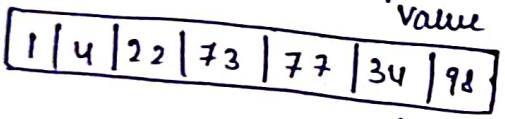
↑
Value (creates a hole, shift the greater elements towards right)



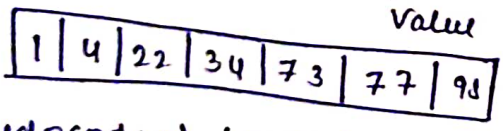
↑ Value



↑ Value



↑ Value



→ Sorted.

Pseudocode →

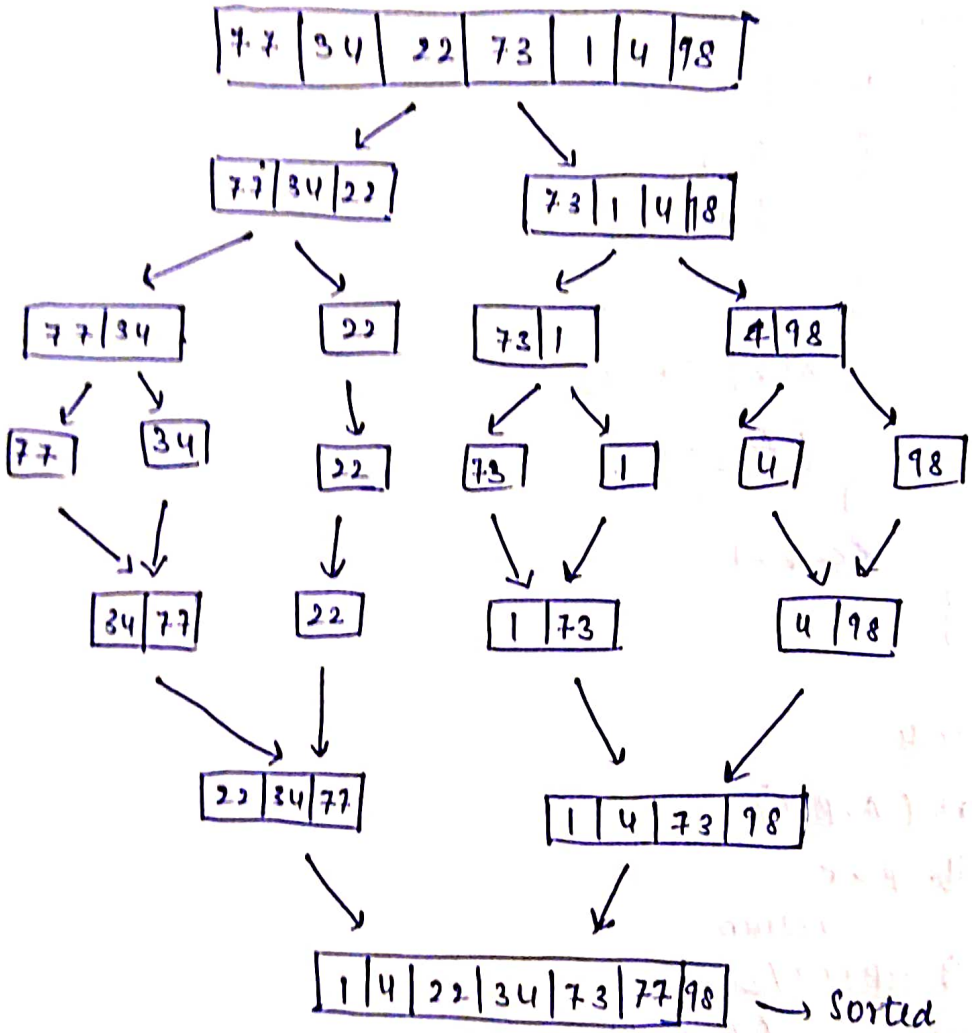
```
Insort (A, n)
{   for i ← 1 to n-1
    {   value ← A[i]
        hole ← i
        while (hole > 0 && A[hole-1] > value)
        {       A[hole] ← A[hole-1]
                hole ← hole - 1
        }
        A[hole] ← value
    }
}
```

Time complexity → $O(n^2)$

2(a) **Merge Sort**

* Uses divide and conquer approach.
* Works on the principle of breaking an array into smaller parts, and then arrange them in ascending order.

eg 77, 34, 22, 73, 1, 4, 98



* Divide the array from the middle into two halves
* Call mergesort in first half and second half.
* Merge the two halves together.
* Find middle point to divide the array → $m = l + (r-l)/2$

Call for merge sort in first half → call mergesort (arr, l, m)

Call for merge sort in second half → call mergesort (arr, m+l, r)

Merge → call merge (arr, l, m, r)

pseudo-code

```
Merge_sort (A, p, r):
        if p > r
                return
        q = (p+r)/2
        mergesort (A, p, q)
        merge sort (A, q+1, r)
        merge (A, p, q, r)
```

(c) Datastructure :

* Way to store data in a computer system for efficient use.
* It is a way of data organisation so that the function of data structure remains independent of implementation.
* The functional definition of a data structure is known as ADT (Abstract data type)
* A well designed datastructure helps in pursuing critical operations with limited memory and space.
* Data structure has two fundamental objectives :
  (i) ways to store data
  (ii) Types of operations performed on it

The way the data is stored affects the performance of program.

eg arrays, stack, queues, trees etc.

Data structures are classified as:

(i) Simple data structure
(ii) Compound data structure
(iii) Linear data structure → (stacks, queues, lists)
(iv) Non-linear data structure → (tree, graphs)

LINEAR DATA STRUCTURE	NONLINEAR DATA STRUCTURE
* Sequentially connected * Single run, all elements traversable. * Easy to implement * Doesnt used memory efficiently	* Hierarchically connected * elements present at various levels. and needs multiple runs. * difficult to understand and implement * Very efficient memory use
Example Arrays, lists, Stacks, queues.	Example Heaps, trees, graphs

(3)

LINEAR SEARCH

* Starts searching from ar[0] and compares each element till searched element is not found.
* Doesnt need to be a sorted array.
* Can be applied on any data structure - array, linked list, etc.
* Based on sequential approach
* Used for small data sets.
* Can be implemented on both, single and multi-dimensional array

TIME COMPLEXITIES →

Worst case → O(n)

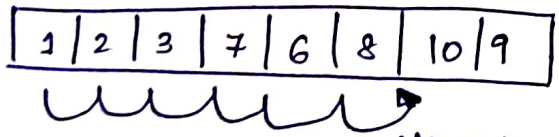
Best case → O(1)

Avg case = O(n/2)

* Slow in speed.

eg

Search 8



element found.
(compare to each element.

Codes →

```
void lsearch (int obj, int item, int start);
{   int start = 0;
    for (int i = start; i < objlen; i++)
    {       if (obj[i] == item)
                return i;
            return -1;
    }
}
```

BINARY SEARCH

It finds the position of the wanted element.

Sorted array is mandatory.

Applied only on data structures that have two way reversal.

* Based on divide and conquer approach
* used on large data sets
* Can be implemented only on a multi-dimensional array.

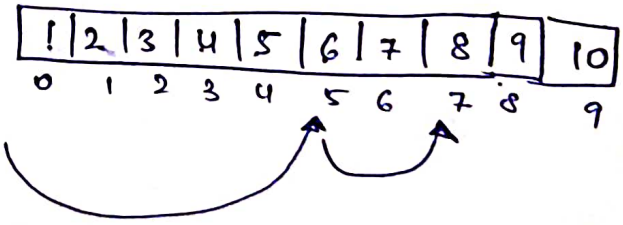
Worst case → $O(\log_2 n)$

Best case → O(1)

Average case = O(log n)

* Fast implementation

Search 8.



Codes → (assuming that the array is sorted.

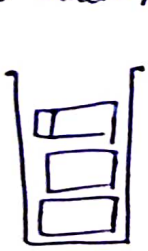
```
int bsearch (int a[], int s, int item)
{   int beg, mid, last;
    beg = 0;
    last = size - 1;
    while (beg <= last)
    {   mid = (beg + last)/2;
        if (item == a[mid])
            beg = mid + 1;
        else
            last = mid - 1;
    }
    return -1;
}
```

5) ADT (Abstract data type)

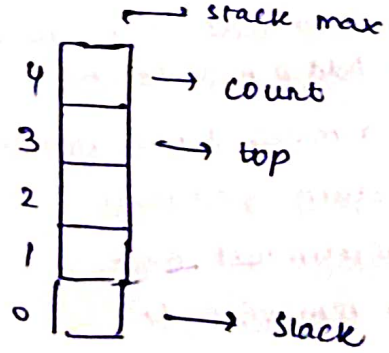
* The abstract data type is a special datatype, which is defined by a set of values and set of operations.
* "Abstract" is used because we can use these datatypes for different operations. But the operation working is concealed from the user.
* It is made up of primitive, existing datatypes, but operation logics are hidden
* So, basically, they are definitions of data and operation but do not have implementation details.

→ STACK ADT

* Elements of same type arranged in a sequential order.
* Operations:
  Initialize() → initializing it to be empty
  Push() → Insert an element
  Pop() → Delete an element
  isEmpty() → checks if stack is empty
  isFull() → checks if stack is full
* It uses FIFO data structure (Last in first out). The element which is placed last, is accessed first.
* Data is not stored in each node, the pointer to the data is stored
* The program allocates memory for the data and address is passed to the stack ADT



Data packets



The head node and data node are encapsulated in ADT.
Head structure contains a pointer to the top and count the number of entries

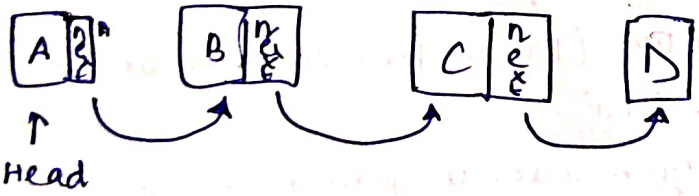
```
typedef struct node
{
        void *Dataptr;
        struct node *link;
}
stack_node;

type def struct
{
        int count;
        Stacknode *top
} Stack;
```

LINKED LIST

It is a type of ADT which has a collection of nodes, the nodes can be accessed sequentially..

No random access to a node.

Nodes are connected to the next node and with previous node, called links. When nodes are connected with only one (next) pointer, its called a singly linked list.



Main operations:

Prepend	→ add a node at beginning
append	→ Add a node at end
pop( )	→ remove a node from end
head( )	→ return first node
tail( )	→ return last node
remove(Node)	→ remove node

```
def binary_search(arr, val, start, end):
    if start == end:
        if arr[start] > val:
            return start;
        else:
            return start + 1;
    if start > end
        return start;

    mid = (start + end) / 2
    if arr[mid] < val:
        return binary_search(arr val, mid+1, end)
    elif (arr[mid] > val):
        return binar-search (arr, val, start, mid-1)
    else:
        return mid;
```

insertion.

```
sort (arr)
    for(i= 1, i < length, i++)
        val = arr[i];
        i = binary_search (arr, val, 0, i-1)
        arr = arr[i] + val + arr[j+1:] + arr[i+1]
    return arr.
```

print → 84

from 81, 82, 56, 57, 30, 19, 84, 47, 42, 25.

THANK YOU MA'AM