① Datatype :- A Datatype is the most basic and the most common classification of data.

The compiler gets to know the form or the type of information that will be used throughout the core. So basically datatype is a type of information transmitted between the programmer and the compiler where the programmer informs the compiler about what type of data is to be stored and also tells how much space it requires in the memory.

Some basic examples are int, float, char etc.

```
Void main ()
{
 int a;
 a = 5;

 float b;
 b = 5.0;

 char c
 c = 'c';

 char d[10];
 d = "example";
```

(2.) Datastructure :- A datastructure is a Collection of different ~~#~~ forms and different types of data having a specific operation that can be performed.

It is a collection of datatype. It is a collection of organising the items in terms of memory and also the way of accessing each item to some define logic.

Some examples of datastructures are Ques, Stacks, linklist, Binary trees and many more.

Datastructure perform some special operation of deletion, insertion and traversion.

For ex :- you have to store data for many employees where each employee has his Name, E-mail ID and mobile number. So this kind of data acquires Complex data which is datastructure comprised of multiple primitive datatype.

(*) Difference between Datatype and Datastructure

| Datatype | Datastructure |
| --- | --- |
| (1) Datatype is a kind or form of a variable which is being used throughout the program. It defines that the particular variable will assign the values of given datatype only. | (1) Datastructure is a collection of different kinds of data that entire data can be represented using an object and can be used throughout the entire program. |
| (2) Implementation through datatype is a form of abstract implementation. | (2) Implementation through datastructure is called concrete implementation. |
| (3) Can hold values and not data | (3) Can hold different kind and types of data within one single object |
| (4) No problem of time complexity. | (4) Time complexity comes into play when working with datastructure. |

(*) Top-down and bottom-up approach

Structure \ procedure oriented programming languages like C programming languages follow top-down approach whereas object oriented programming languages like C++ and Java follows bottom-up approach

The Top-down approach begins with high level design and ends with low level design and development. whereas bottom-up approach begins with low level design or development and ends with high level design

In top-down approach main function is written first and all sub functions are called from main function then Sub-functions are written based on the requirement. whereas In bottom-up approach Code is developed for modules and them these modules are integrated with main function.

Now a days both approaches are combined together followed in modern software design.

Q) Debugging technique :- Debugging is a process of detecting and removing of existing and potential error. (also called as bugs)

In a software code that can cause it to behave unexpectedly or crash. To prevent incorrect operation of a software or system. Debugging is used to find and resolve bugs or defects.

When various sub-systems or modules are tightly coupled debugging becomes harder as any change in one module may cause more bugs to appear in another module. Sometimes it takes more time to debug a program than to code it.

Discription :- The program debugging is the process of isolating and correcting the error.

(i) One simple and effective debugging is to take snapshot of program institution by inserting printf() statement at key points in the main program.

(ii) A second technique is known as ~~Scaff~~ Scaffolding, we insert the comment in the source code so whenever the data which is not

required can be exits deleted.

(iii) Rollback or backtrack, ~~the incorrect results in backtrack~~

if we are able to track the system last configuration in which the system behaves normally. In that condition if the system crashes it will automatically return or roll back to its previous performing condition.

Check points:-

(i-i) PW ~~Variables~~ flag = 0 at key points.

(*) Program testing: Testing is a process of finding bugs and errors in a software product that is done manually by tester or can be automated.

whereas, In the debugging it is the process of fixing the bugs found in developer. The programmer and is responsible for debugging and it can't be automated.

The tester never tests the error but rather find them and return to the programmer.

(*) The view of datastructures.

(i) Abstract
(ii) Implementation
(iii) A

(i) Abstract or logical level :- On this level the programmer has to decide how the elements are related to each other and what operations are needed

At this level the programmer has no concern about the representation of data in memory and how these operations are carried out.

For example:- If you are a delhi and want to go Goa, then you must know the appropriate route from delhi to Goa. Specially, when there are more than one route.

Similarly, In program which datastructure is to be used in which or what kind of data.

(ii) Implementation level :- On the Implementation level we do a specific way for representation of the datastructure.

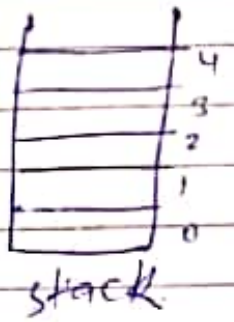
In Implementation we Convert our algorithm into a appropriate program. and to decide how much memory is allocated at static and dynamic binding.

(iii) Application level :- on this level, we used the datastructures which is implemented at earlier level.

For ex:- Stack is to used to hold the processes that are waiting for different resources of the Computer System in a multi programming operating system.

(*) Stacks :- Stack is a linear datastructure in which insertion (Push) and deletion (pop) operation can be done from the same end. i.e. the top of the stack and it follows LIFO structure.

L I F O
↓ ↓ ↓ ↓
last In first out




Implementation of stack

operations :-

(i) Create Empty stack
(ii) Is empty
(iii) Is full
(iv) Push
(v) pop
(vi) peek
(vii) size

Static Implementation array
Dynamic Implementation link list

```
Struct stack
{
  int top;
  int element [MAX];
}
Struct stack *S;
```

```
typedef struct stack
{
  int top;
  int element[MAX];
}
stack;
```

(1)
```
Void create empty_stack (stack *s)
{
   s->top = -1
}
```
→ O(1)

(2)
```
int Is_empty (stack *s)
{
   if (s->top == -1)
    return 1;
   else
    return 0;
}
```
→ O(1)

(3)
```
int Is_Full (stack *s)
{
  if (s->top == (MAX-1)
  return 1;
  else
  return 0;
}
```

(4) Application of stack.

i) Word wise reversal

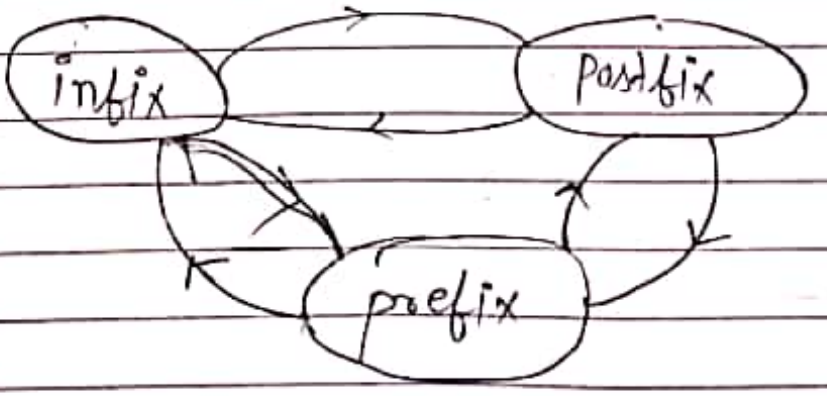
ii) Balanced parenthesis

* Infix evaluation

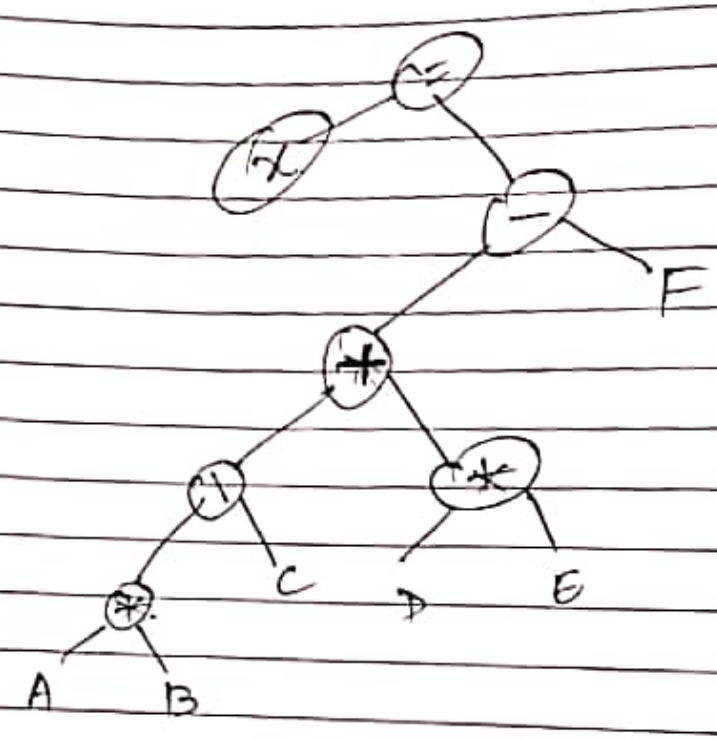
Prefix evaluation

Postfix evaluation.

| Infix | prefix | Postfix |
|---|---|---|
| A + B | + AB | AB + |



$X = A * B / C + D * E - F$




1. precedence
2. Associativity
3. * / ⇒ left associative
   + - equal precedence
   + - left associative
   = least priority.

Exponential

$X = A \uparrow B \uparrow C / D * E + F * G$

$\Rightarrow A \uparrow [\uparrow BC] / D * E + F * G$

$\Rightarrow [\uparrow A \uparrow BC] / D * E + F * G$

$\Rightarrow [/ \uparrow A \uparrow BCD] * E + F * G$

$\Rightarrow [* / \uparrow A \uparrow BCDE] + [* FG]$

$\Rightarrow [+ * / \uparrow A \uparrow BCDE * FG]$

$2 + 3 * 4 - 7 + 6 / 2$

3 1 4 5 2

$2 + 3 * 4 - 7 + 62 \backslash$

$2 + 34 * - 7 + 62 \backslash$

$234 * + 7 - 62 \backslash +$

Application of Stack

1) Postfix Evaluation :-

(a) Scan the given postfix expression from left to right symbol by symbol.

(b) If the current scaned symbol is an operand push the operand into stack.

(c) If the current scaned symbol is an operator.

a.1) Apply two pop operations on the stack.

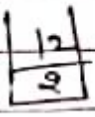
a.2) Apply the operator and push back into the stack

(d) Repeat Step - 2 'a' and 'b' until whole expression is scan.

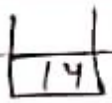
(e) pop the result out from the stack.

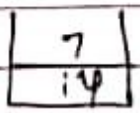
1) 

3 * 4 = 12

2.) 

2 + 12 = 14

3.) 


4) 

14 - 7 = 7

5 

6 \ 2 = 3

6) 

7 + 3 = 10

## Application of Stack

(i) Prefix evaluation.

(a) Scan the given prefix expression from right to left.

(b) If the scaned symbol is an operand push the symbol into the stack.

(c) If the scaned symbol is an operator

c.1) Apply two times pop-operation.

c.2) Apply operators and push the result back into the stack.

d) Repeat Step- 2 and 3.

e) pop the result

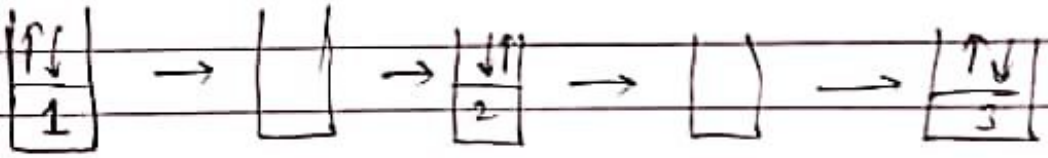
Generating permutation

1.) 123
2. 321
3.) 231

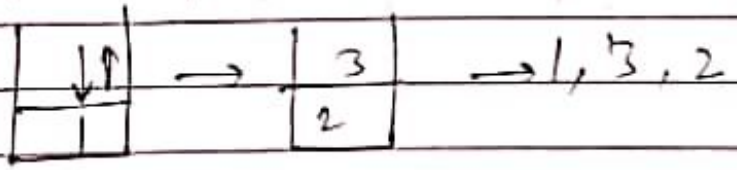
If n = 3 what is possible sequence

| | | | |
|---|---|---|---|
| 1.) | 123 | ✓ ✓ | x |
| 2.) | 132 | ✓ | x ✓ |
| 3.) | 213 | ✓ | ✓ ✓ |
| 4.) | 231 | ✓ | ✓ ✓ |
| 5.) | 312 x | ✓ | ✓ |
| 6.) | 321 | ✓ | ✓ ✓ |

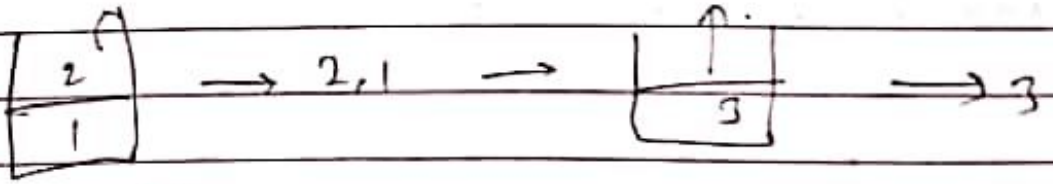
(a)



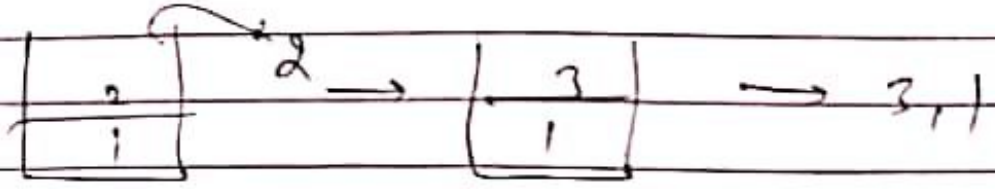

(b)



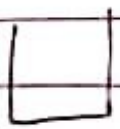

(c)




(d)

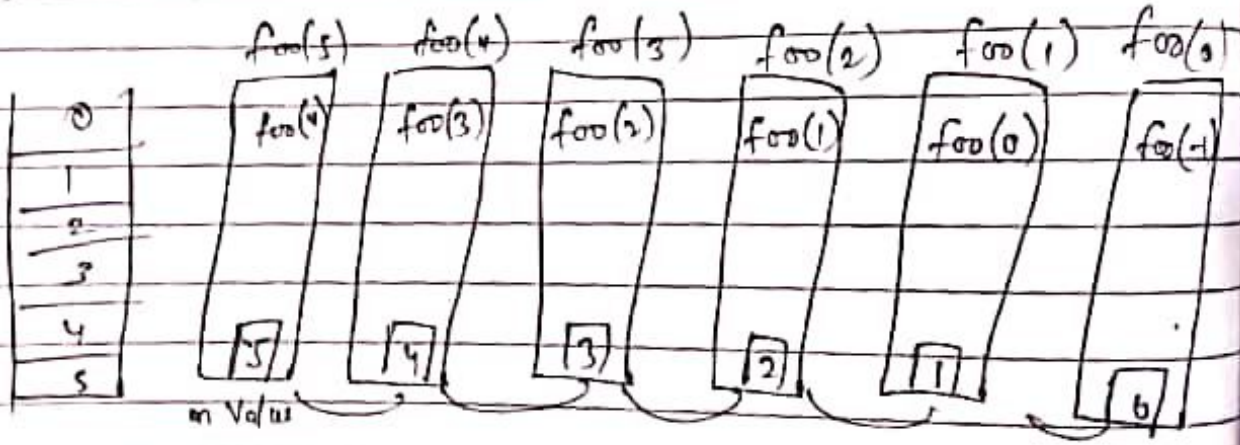



(e)



(*) Iterative Recursion

```
Void foo (int n)
{
  if (n<0)
  else
  {
    foo (n-1)
    printf ("%d", n);
  }
}
```




pop the result from the stack

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|

(*) A Stack is a non-primitive linear datastructure, it is an ordered list in which addition of new data item and deletion of already existing data item is done from only one end known as ~~Sum~~ top of the stack.

As all the deletion and insertion in a stack is done from top of the stack the last added element will be the element will be the first to be removed from the stack.

Ex:- A common model of stack is plates arranged in a marriage party.

(*) Stack implementation :- Stacks can be implemented in two ways.

(i) Static implementation
(ii) Dynamic implementation

(i) Static implementation. :- Static implementation uses arrays to create stack. It is very simple technique but it is not a flexible way of creation.

As the size of the stack has to be declared during program design after that the size cannot be vary. moreover the static implementation is not too efficient with respect to memory utilization.

(i) As the declaration of array for implementing stack is done before the start of the operations (At program design time), flow if there are too few elements to be stored in the stack. The statically allocated memory will be wasted. On the other hand if there are more no. of elements to be stored in the stack we can't be able to change the size of the array to increase its capacity.

Dynamic implementation :- The dynamic implementation is called as linklist representation and it uses pointers to implement the stack type of datastructure.

## Operation on stack

The basic operations that can be performed on stacks are as follows

(i) PUSH :- The process of adding a new element to the top of the stack is called as PUSH operation.

Pushing an element in a stack invoke adding of element, as the new element will be

inserted at the top after every push operation the top is incremented by 1.

In case the array is full and no new element can be accomodated it is called as stack full condition. The condition is called as overflow.

(ii) pop :- The process of deleting an element from the top of the stack is called as pop operation.

After every pop operation the stack is decremented by 1. If there is no element in the stack and pop is performed. Then this will result to stack underflow condition.

(iii) Stack terminology :-

(a) Context :- The environment in which a function executes includes argument variable, local variables and global variables all the context accept the global variable is stored in stack frame.

(b) Stack frames :- The datastructure containing all the data (arguments, local variables, return address) needed each time a procedure or function is called.

(c) Max size :- The term is not a standard one we use this term to refer the maximum

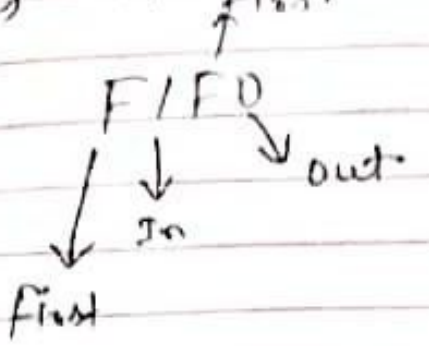
size of stack.

(iv) Top :- This term refers to the top of the stack. The stack top is used to check stack overflow or underflow conditions. Initially top stores -1. This assumption is taken so that the neighbour an element is added to the stack the top is first implemented and then the item is inserted into the location currently indicated by the top.

(v) Stack empty or underflow :- This is the situation when the stack contains no element at this point of the stack. The top is present at the bottom of the stack.

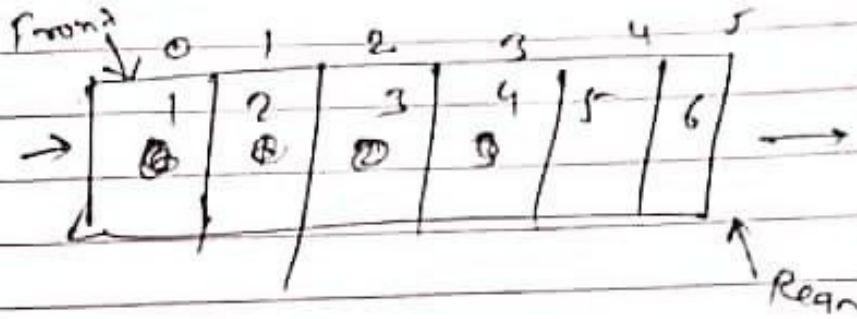
(vi) Stack overflow :- This is the situation when the stacks become full and no more elements can be pushed on to the stack. At this point the stack top is present at the highest location of the stack.

## Queues




Rear - Insertion

front - Deletion




As other list queues can be implemented in two ways.

(i) Static implementation (using arrays).

(ii) Dynamic implementation (using pointers).

⇒ If que is implemented using arrays we must be sure about the exact no. of elements we want to store in the Que. because we have to declare the size of the array at design time. or before the processing stands. ~~In this~~

In this case, the beggining of the array will become the front for the que and the last location of the array will act as rear for the que. the following relation gives the total no. of elements present in the que while implementing using arrays.

$$F - R + 1$$

## Queue

(i) Create empty - queue
(ii) Is-empty
(iii) Is - full
(iv) Insert
(v) Delete
(vi) size

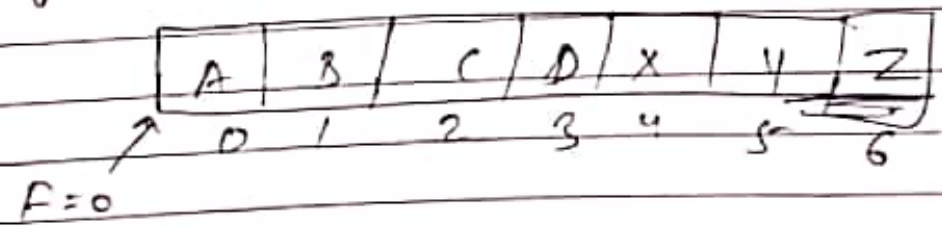
(i)
```
typedef struct Queue 1
{
  int f, r;
  int element [MAX];
}
Struct Queue 1 *q
void create_empty_queue (queue *q)
```

```
{
  q->f = q->r = -1
}
```

(ii) Second law —

```
int Is_empty (queue *q)
{
  if (q->f == q->r == -1)
    return 1;
  else
    return 0;
}
```

(iii) Overflow condition




```
int Is_full (queue *q)
{
  if (q->f == 0) && (q->r == MAX-1)
    return 1;
  else
    return 0;
}
```

④ Void insertion (queue *q, int data)

```
{
  if (q->f == q->r == -1)
      q->f = q->r = 0;
  else if (q->f != 0) && (q->r == max-1)
  {
   for (i = q->f; i <= q->r; i++)
   {
    q->element[i-q->f] = q->element[i]
      q->r = q+r-q+f+1;
        q->f = 0;
   else
   {
    q->r = q+r+1;
   }
  q->element[q->r] = data;
```

Scanned by CamScanner