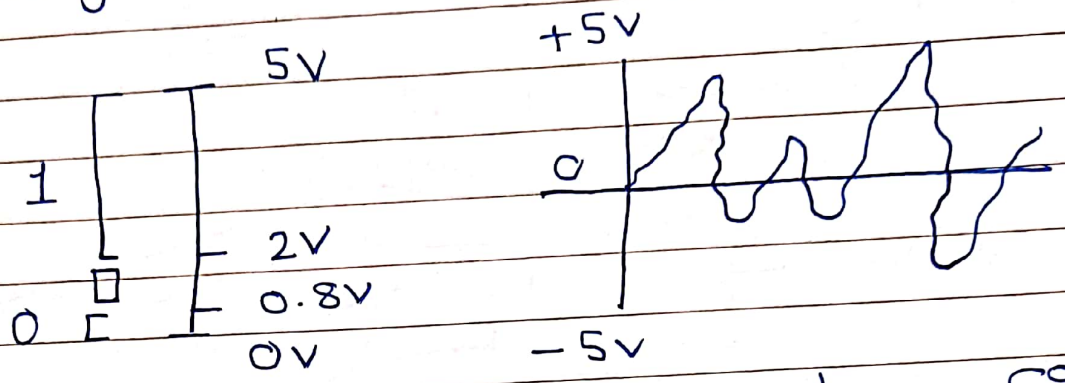
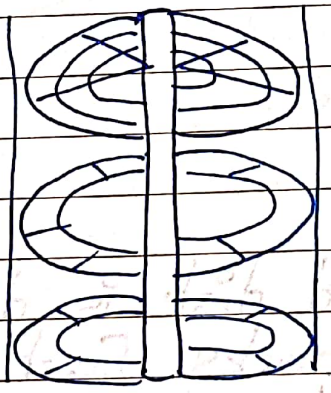
Data Structures

* **Binary 0 and 1:**



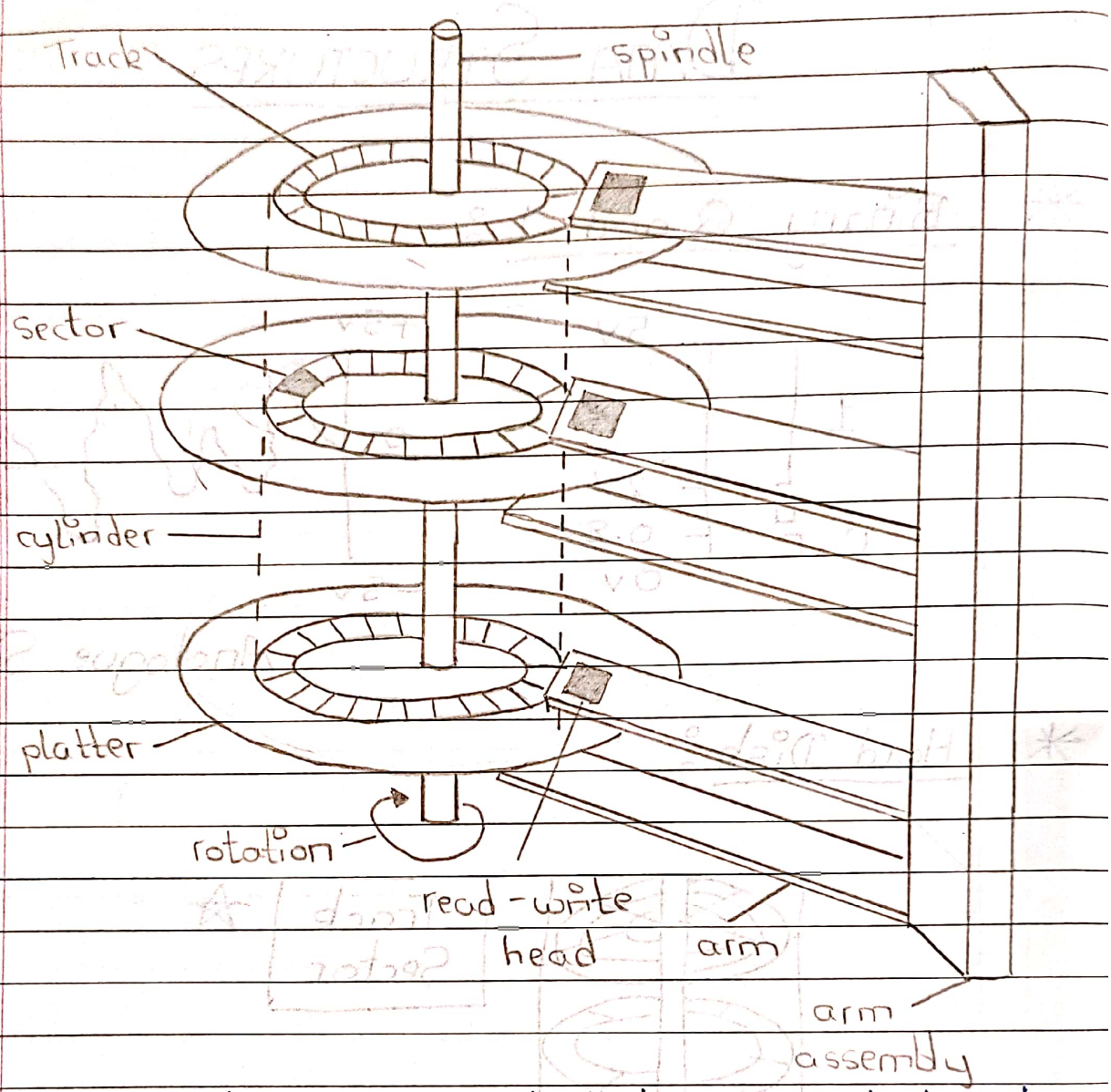
Analogue Signal

* **Hard Disk:**



Track
Sector

☆ **HW:**
1. HDD diagram and explanation
   - Label
   - Track, sector?
2. ASCII
3. Binary No. systems
4. Decimal to binary conversions
5. Memory bits



— Each platter of a hard disk is divided into a number of concentric tracks. Each track is divided into number of sectors each of which can store the same amount of data.

* ASCII – American Standard Code for Information Interchange.

A → 65	1 → 49	a → 97
Z → 90	9 → 57	z → 122

* Memory Units :

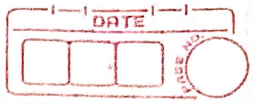
1 Bit = Binary Digit
8 Bits = 1 Byte
1024 Bytes = 1 KB
1024 KB = 1 MB
1024 MB = 1 GB
1024 GB = 1 TB

* ASCII -

- A standard data-encoding format for electronic communication between computers. It assigns standard numeric values to letters, numerals, punctuation marks, and other characters used in computers.

* HDD

offset, base value (processor)




```
#include <stdio.h>
#include <conio.h>

void main()
{
   char name[]; int i; int l;
   clrscr();
   printf("Enter your name: \n");

   for (i=0;

   printf("Enter length: \n");
   scanf("%d", &l);

   printf("Enter name: \n");

   for (i=0; i<l; i++)
      {
         scanf("%c", &name[i]);
      }
   printf("Name is: \n");
   for (i=0; i<l; i++)
      {
         printf("%c", name[i]);
      }
}
```

* HW: WAP to create 3 arrays to store first middle and last name.

```
#include <stdio.h>
#include <conio.h>

void main()
{
String name char name[];
                printf("Enter name:
```

* Types of Statement :

- Infix → A+B
- Prefix → +AB
- Postfix → AB+

* Conversion of Infix to Prefix, Postfix using Stack :

- Priority / Precedence and Arithmetic Operators

Brackets Highest

a. Highest Precedence : ^ or $
b. Mid Precedence : *, /
c. Lowest Precedence : +, −

Eg: A+(B*C)

→ A+(BC*)

A(BC*)+

ABC*+

Infix ↓ Postfix

Q To Prefix : A+(B*C)

→ A+(*BC)

+A(*BC)

+A*BC

Q (A+B) * C

→ (+AB) * C
*+ABC

(AB+) * C
(AB+) C *
AB+C *

Q (A+B) * (C-D) to Postfix

→ (AB +) * (CD-)

(AB +) (CD-) *

AB+CD- *

Q A $B * C-D+ E/ F/ (G+H)

→ A $B * C-D + E / F / (GH+)

(AB $) * C-D + E / F / (GH+)

(AB $C *) - D + E / F / (GH+)

(AB $ C *) - D + (EF/) / (GH+)

(AB $C *) - D + (EF /GH + /)

(AB $ C *) - (DEF/GH +/ +)

AB $ C * DEF / GH + / + —

* Using Stack

Q A+B*C

Symbol	Stack	Postfix
A	Empty	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
	+	ABC*
		ABC*+

Q (A+B)*C

Symbol	Stack	Postfix
(	~~Empty~~ (	Empty
A	(	A
+	(+	A
B	(+	AB
)	Pop, Empty	AB+
*	*	AB+
C	*	AB+C
		AB+C*

Q ((A-(B+C))*D)$(E+F)

→

Symbol	Stack	Postfix
(	(	
(	((	
A	((	A
-	((-	A
(	((-(	A
B	((-(	AB
+	((-(+	AB
C	((-(+	ABC
)	((-	ABC+
)	(	ABC+-
*	(*	ABC+-
D	(*	ABC+-D
)		ABC+-D*
$	$	ABC+-D*
(	$(	ABC+-D*
E	$(	ABC+-D*E
+	$(+	ABC+-D*E
F	$(+	ABC+-D*EF
)	$	ABC+-D*EF+
		ABC+-D*EF+$

* Evaluating Postfix Expression

Step 1: Initialize the empty stack

Step 2: Scan the postfix expression from left to right

Step 3: For each element (operand and operator) in the expression.

a. If its an operand, push it onto the stack
b. If its an operator, pop the required number of operands from the stack, apply the operator and push the result back on to the stack.

c.

Step 4: After scanning all elements, the final result will be on the top of the stack

Q 623+-382/+*2$3+

Symbol	Operand 1	Operand 2	Value	Stack
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3				1, 3
8				1, 3, 8
2				1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7

2				7, 2
$	7	2	49	49
3				49, 3
+	49	3	52	52

Q 231*+9-

Symbol	Operand 1	Operand 2	Value	Stack
2				2
3				2, 3
1				2, 3, 1
*	3	1	3	2, 3
+	2	3	5	5
9	2	3	5	5, 9
-	5	9	-4	-4

Q 100 200 +2/5 *7 +

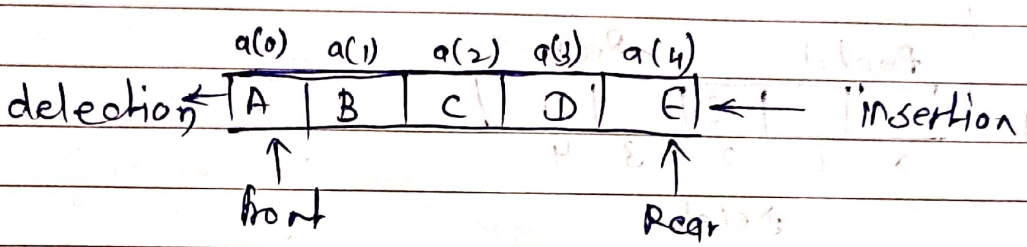
Symbol	Operand 1	Operand 2	Value	Stac
100				100
200				100, 20
+	100	200	300	300
2				300, 2
/	300	2	150	150
5				150, 5
*	150	5	750	750
7				750, 7
+	750	7	757	757

= 757

29 Aug 2023

Queue:

- Ordered collection of homogenous elements.
- non-primitive linear Data structure.
- new element added at 'rear' end existing element ~~and~~ are deleted at 'Front' end
- Mechanism is FIFO



isFull() - True/False

isEmpty - True/false.

↓ element is not available → element available

There are 3 types of queue:

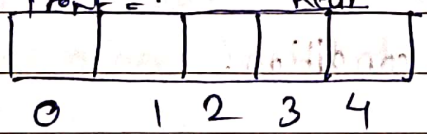
Queue

enqueue - insert

dequeue - delete

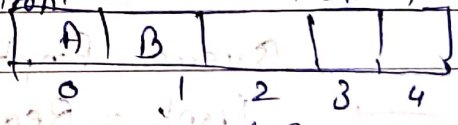
insertion:

1) Front = -1 Rear = -1



Queue empty

3) Front = 0 Rear = 1

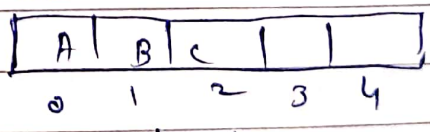


insert B

2) front=0 Rear=0

insert A

4) Front = 0 Rear = 2



insert C

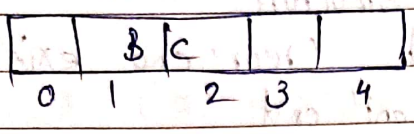
1 Bit = Binary Digit
8 Bits = 1 Byte
1024 Bytes = 1 KB
1024 KB = 1 MB

1024 MB = 1 GB
1024 GB = 1 TB

Page No.:
Date:
YOUVA

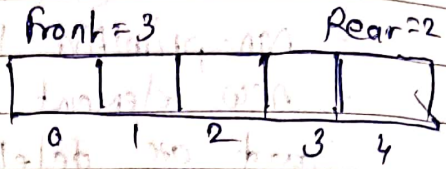
Deletion:

1)



Delete B

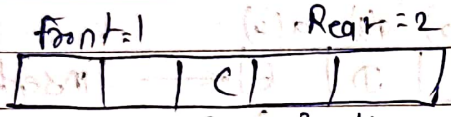
3)



Delete

Queue is empty.

2)

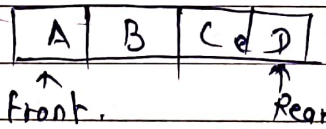


Delete

Queue Full (over Flow) condition:

→ Rear = Max - 1.

→ Queue is fully occupied & enqueue() operation is called, queue overflow occur.

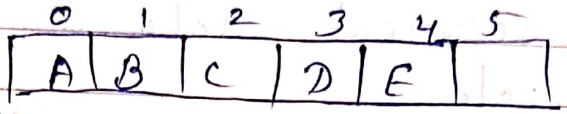


Queue Empty (underFlow) condition.

→ Front = Rear = -1

→ When queue is fully empty & dequeue() operation is called queue underflow occur.

Disadvantage of linear Queue:

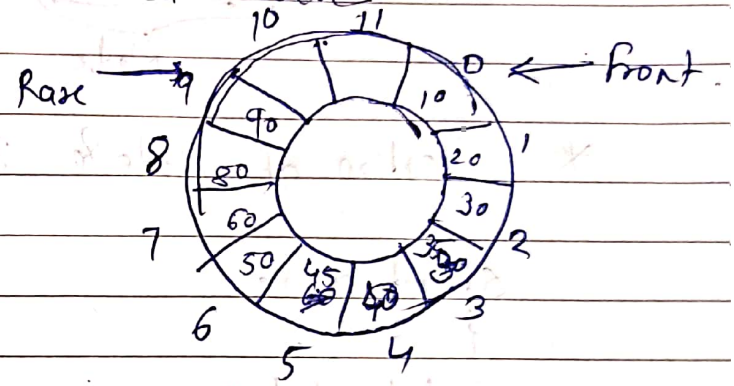


if we want to A:

Memory wasted unused → | | B | C | D | E | |

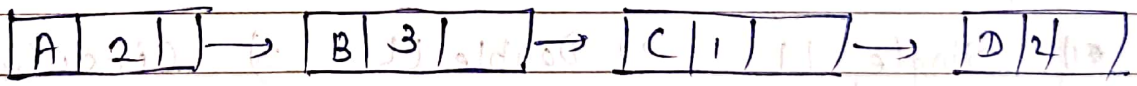
If we delete one element, we cannot use that vacant place ga again.

* To overcome Drawback of linear Queue use Circular Queue



* Priority Queue
- Each element assigned priority
- element of higher priority is processed before element of lower priority
- two element with same priority are processed according to order in which they are added to queue

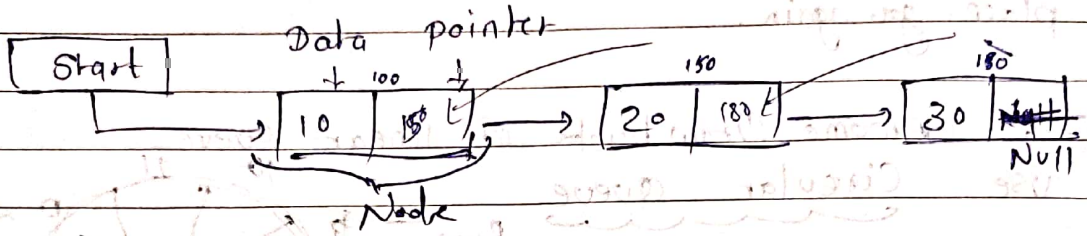
Data ← | info | priority | NEXT | →



Unit-3 Linked list

4/9/2023

→ Handle dynamic data element.
→ Node represent data & link or pointer connect each node
→ Last node contain NULL in next ~~find~~ field.
→ LL can grow & shrink its size as requirement
→ It does not waste memory.



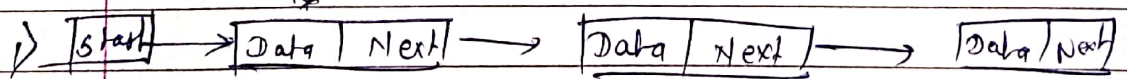
* Creation of Node & Declaration of LL.

```
struct node
{
  int data;
  struct node *next;
};
struct node *n;
n = (struct node*) malloc (sizeof (struct node));
```

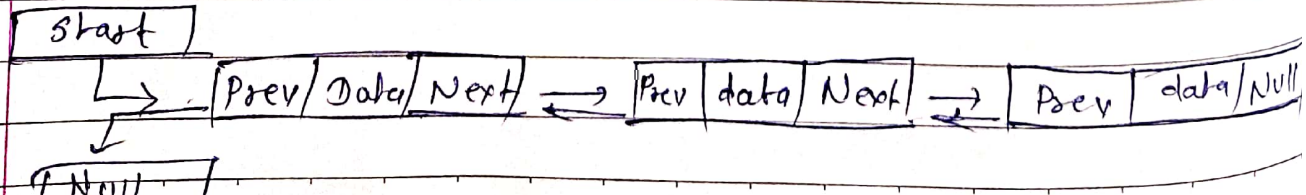
* Types of linked list:

*1) Single LL 2) Double LL 3) Circular LL

1)



2) Double LL



3) Circular LL