

## MODULE 3

### LINKED LIST & MEMORY MANAGEMENT



Prepared By Mr. EBIN PM, AP, IESCE

1

## SELF REFERENTIAL STRUCTURE

- Self Referential Structure is the Data Structure in which the **pointer refers (points) to the structure of the same type.**
- In other words, **structures pointing to the same type of structures** are self-referential in nature.
- A self referential structure is used to create data structures like **linked lists, tree, graph** etc.

### Self Referential Structures

```
struct node {  
    int data1;  
    char data2;  
    struct node* link;  
};
```

Prepared By Mr. EBIN PM, AP, IESCE

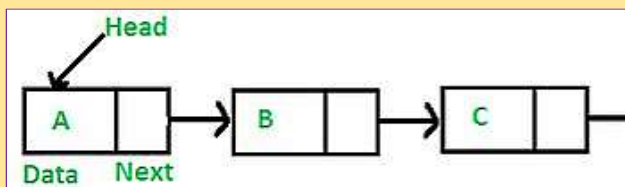
EDULINE

2

**Eg: In singly linked list**

```
struct node{
int data;
struct node *next; // self-referential
};
```

- In the above declaration next is the pointer to the structure of type node. Here **next is the pointer** which will contains the address of the structure of the same type (i.e **address of next node**) and data will contain the actual data.



Prepared By Mr.EBIN PM, AP, IESCE

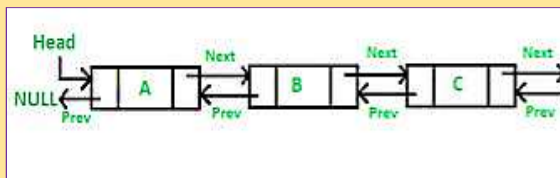
EDULINE

3

**Eg: In Doubly linked list**

```
struct node{
int item;
struct node *prev; //self-referential
struct node *next; //self-referential
};
```

- In the above declaration **prev & next are the pointer** to the structure of type node. Here prev pointer contains the address of the previous node and next pointer contains the address of the next node.



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

4

## DYNAMIC MEMORY ALLOCATION

### ❖ Memory Allocation Process

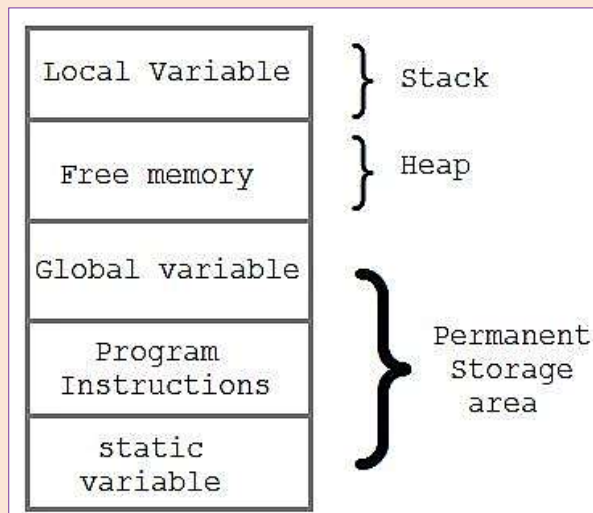
- Global variables, static variables and program instructions get their memory in permanent storage area whereas local variables are stored in a memory area called Stack.
- The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.
- The process of allocating memory at runtime is known as **dynamic memory allocation**. Library routines known as memory management functions are used for allocating and freeing memory during execution of a program.

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

5

### Program memory layout



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

6

- There are 4 library functions provided by C defined under `<stdlib.h>` header file to facilitate dynamic memory allocation in C programming. They are:

`malloc()`

`calloc()`

`free()`

`realloc()`

- **malloc()** function is used for **allocating block of memory** at runtime. This function reserves a block of memory of the given size and returns a pointer of type void. This means that we can assign it to any type of pointer using typecasting.

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

7

### Syntax:

```
void* malloc(byte-size)
```

### Eg:

```
int *x;
```

```
x = (int*)malloc(50*sizeof(int)); //memory space allocated to  
                                variable x
```

```
free(x); //releases the memory allocated to variable x
```

- **calloc()** is another memory allocation function that is used for allocating memory at runtime. `calloc` function is normally used for allocating memory to derived data types such as **arrays** and **structures**.

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

8

**Syntax:**

```
void *calloc(number of items, element-size)
```

**Eg:**

```
struct employee
{
    char *name;
    int salary;
};
typedef struct employee emp;
emp *e1;
e1 = (emp*)calloc(30,sizeof(emp));
```

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

9

➤ **realloc()** changes memory size that is already allocated dynamically to a variable.

**Syntax:**

```
void* realloc(pointer, new-size)
```

**Eg:**

```
int *x;
x = (int*)malloc(50 * sizeof(int));
x = (int*)realloc(x,100); //allocated a new memory to variable x
```

➤ The **free( )** function is used to de-allocate the previously allocated memory using malloc( ) or calloc( ) functions.

**Syntax :** free (ptr\_var);

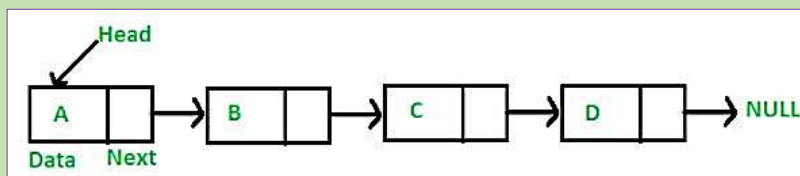
Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

10

## LINKED LIST

- A linked list is a **linear data structure**, in which the **elements are not stored at contiguous memory locations**. The elements in a linked list are linked using pointers as shown in the below image:



- In simple words, a linked list consists of **nodes** where each node contains a data field and a reference(link) to the next node in the list.

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

11

### ❖ Comparison Between Array & Linked List

No	ARRAY	LINKED LIST
1	Difficult to perform insertion & deletion operation	Easy to perform insertion & deletion operation
2	It is easy to access an element from an array	To access an element from a list, we must start from beginning of the list and then take address of next element from current node
3	Array element access is random access and it is fast	In linked list , access is sequential and it is slow
4	Array need space to store only the data element. No extra space is required	In linked list additional space is required to store the pointers
5	Array elements are stored in contiguous memory locations.	List elements need not be stored in contiguous memory location
6	For insertion & deletion, it takes more time. For dictionary operation array take less time	Insertion and deletion take less time. But dictionary operation take very less time.

Prepared By Mr.EBIN PM, AP, IESCE

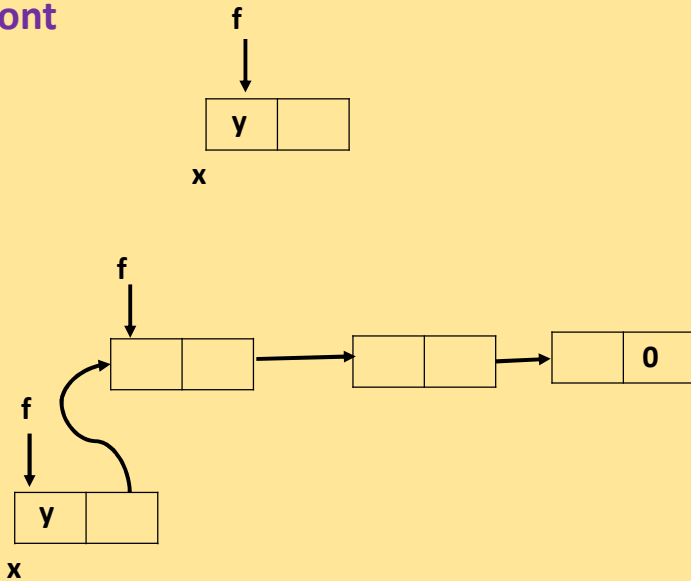
EDULINE

12

### ❖ Inserting a node on the front

```

Algorithm insert front (f)
{
// f is a pointer points to first node
new (x);
read (y);
x.data = y;
if (f==nil)
{
f = x;
x.link=nil;
}
else
{
x.link=f;
f=x;
}
}
    
```



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

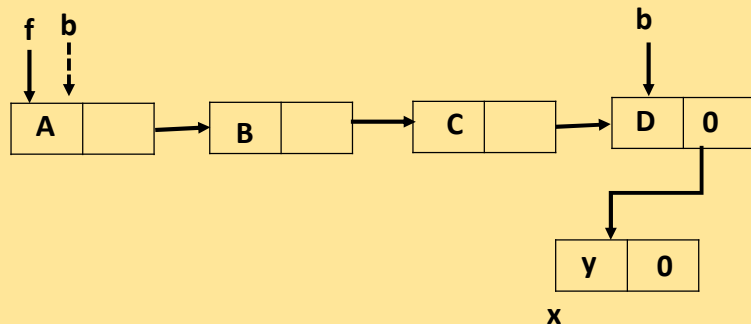
13

### ❖ Inserting a node at the end of list

```

Algorithm insert end (f)
{
// f is a pointer points to first node
new (x);
read (y);
x.data = y;
x.link=0;
if (f= nil)
f = x;
else
b= f; /* b is a dummy pointer*/
while (b.link ≠ nil)
{
b = b.link;
}
b.link = x;
}
    
```

If a list exist, we must traverse that list for finding the end of the list. For that purpose we create a dummy pointer **b** which is also point to the first node. This **b** become move and **f** points first node at all times. If **b.link=nil**, then we points **b.link** to **x**



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

14

### ❖ Inserting a node between two nodes after a given data value

```

Algorithm insert between (f, d)
{
// f is a pointer points to first node
// d is the data value
new (x);
read (y);
x.data = y;
if (f == nil)
{
f = x;
f.link = nil;
return;
}
else
p = f;

```

```

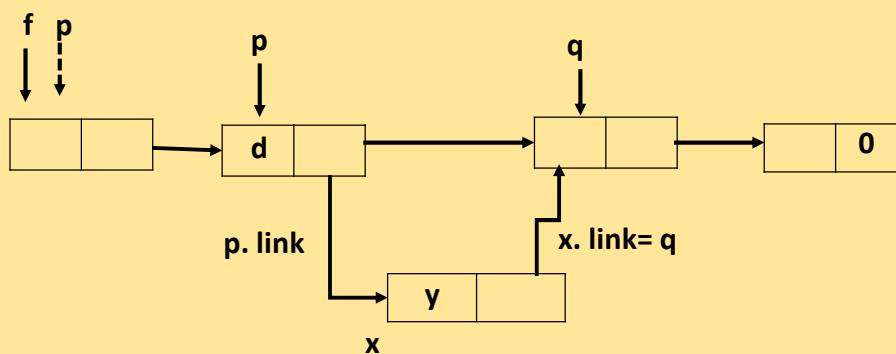
while (p.data ≠ d)
{
p = p.link;
}
if (p.link == nil)
{
p.link = x;
x.link = nil;
}
else
{
q = p.link;
p.link = x;
x.link = q;
}
}

```

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

15



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

16



- If a list exist, then a pointer p is set which points to f. Then we traverse this pointer p upto p.data=d.
- Then we check, if p.link=nil. If yes, the new node x will assign as the last node and convert its link part using nil value.
- Else , we create a new variable q and p.link is assigned to that q.
- P.link is assigned to x and x.link is also given to q

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

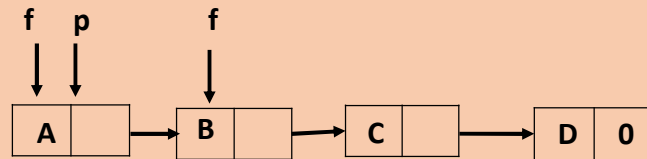
17

### ❖ Delete first node from linked list

```

Algorithm delete (f)
{
// f is a pointer points to first node
if (f==nil)
    print("Deletion not possible");
else
    {
    p = f;
    f = f.link;
    dispose (p);
    }
}

```



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

18

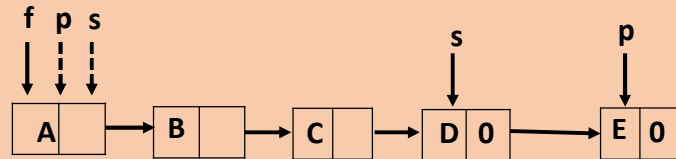
## ❖ Delete last node from linked list

Algorithm delete last node from list (f)

```

{
// f is a pointer points to first node
if (f==nil)
    print("Do nothing");
elseif (f.link==nil)
    dispose(f);
else
    {
    p = f;
    while(p.link!=0)
        {
        s=p;
        p = p.link;
        }
    s.link =0;
    dispose (p);
    }
}

```



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

19

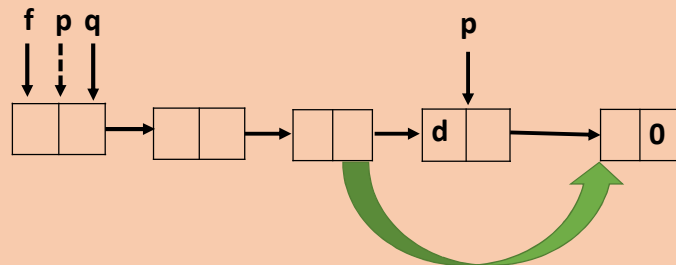
## ❖ Delete a node from middle, after a given data value.

Algorithm delete middle (f, d)

```

{
// f is a pointer points to first node
// d is the data value
p=f;
if (f.data == d)
    {
    f=f.link;
    dispose (p);
    }
else
    {
    while(p.data!=d)
        {
        q=p;
        p=p.link;
        }
    q.link=p.link;
    dispose (p);
    }
}

```



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

20

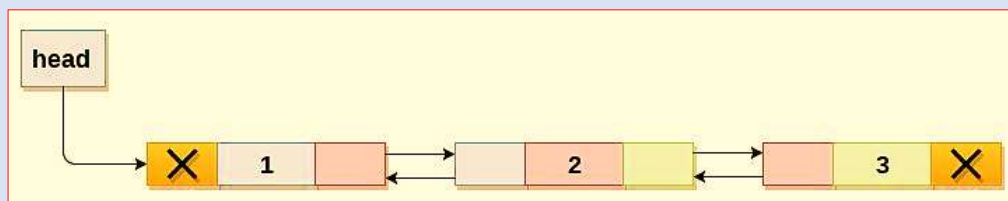
## DOUBLY LINKED LIST

- In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we **can not traverse back**.
- We can solve this kind of problem by using a double linked list.
- In a double linked list, every node has a link to its previous node and next node.
- So, we **can traverse forward** by using the next field and **can traverse backward** by using the previous field.
- Every node in a double linked list contains three fields and they are shown in the following figure

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

21



- In double linked list, the **first node must be always pointed by head**.
- Always the **previous field of the first node must be NULL**.
- Always the **next field of the last node must be NULL**.

➤ In C, structure of a node in doubly linked list can be given as :

```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
}
```

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

22

### ❖ Memory Representation of a doubly linked list

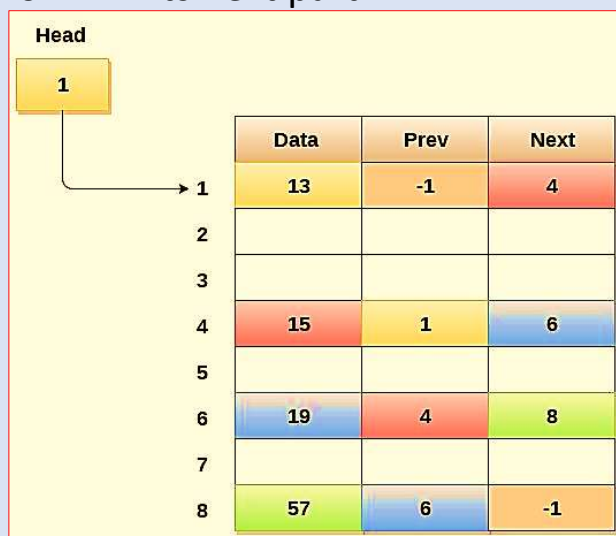
- Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion.
- However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).
- In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1.
- Since this is the first element being added to the list therefore the prev of the list contains null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

23

- We can traverse the list in this way until we find any node containing null or -1 in its next part.



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

24

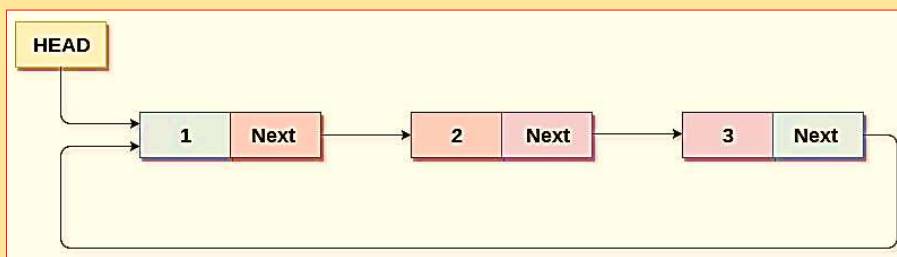
## CIRCULAR SINGLY LINKED LIST

- In a circular Singly linked list, the **last node of the list contains a pointer to the first node of the list**. We can have circular singly linked list as well as circular doubly linked list.
- We traverse a circular singly linked list until we reach the same node where we started.
- The circular singly linked list has **no beginning and no ending**.
- There is **no null value present in the next part of any of the nodes**.
- The following image shows a circular singly linked list.

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

25



- Circular linked list are mostly used in **task maintenance** in operating systems.
- There are many examples where circular linked list are being used in computer science including browser surfing where **a record of pages visited in the past by the user, is maintained in the form of circular linked lists** and can be accessed again on clicking the previous button.

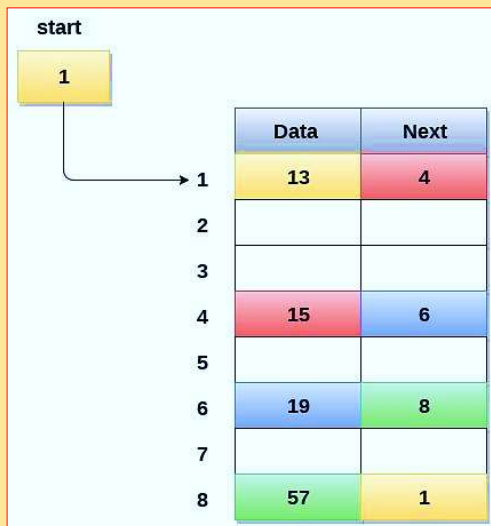
Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

26

### ❖ Memory Representation of circular linked list

- The last node of the list contains the address of the first node of the list.



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

27

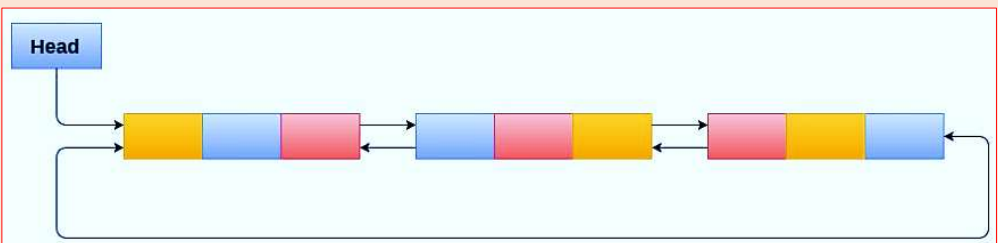
## CIRCULAR DOUBLY LINKED LIST

- Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node.
- Circular doubly linked list **doesn't contain NULL** in any of the nodes.
- The **last node of the list contains the address of the first node of the list.**
- The **first node of the list also contains the address of the last node in its previous pointer.**
- A circular doubly linked list is shown in the following figure.

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

28

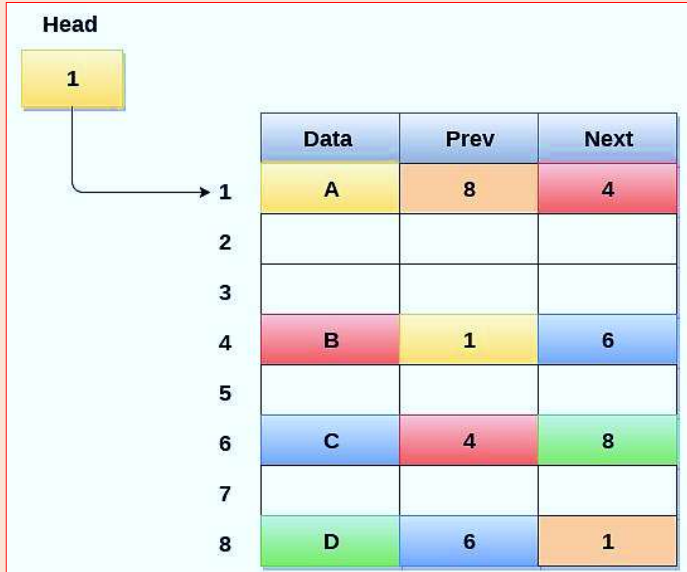


The diagram shows a circular doubly linked list. A blue box labeled 'Head' has two arrows pointing to the first node. The list consists of three nodes, each represented as a horizontal bar divided into three colored segments: yellow, blue, and red. The nodes are arranged in a circle. The first node's yellow segment points to the second node's blue segment, and its red segment points to the second node's red segment. The second node's blue segment points to the third node's yellow segment, and its red segment points to the third node's red segment. The third node's red segment points to the first node's yellow segment, and its yellow segment points to the first node's blue segment. A long arrow from the 'Head' box also points to the first node's yellow segment.

- Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands **more space per node** and more expensive basic operations.
- However, a circular doubly linked list provides easy manipulation of the pointers and the **searching becomes twice as efficient**.

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 29

### ❖ Memory Management of Circular Doubly linked list



The diagram illustrates memory management for a circular doubly linked list. A yellow box labeled 'Head' contains the number '1' and has an arrow pointing to the first row of a table. The table has three columns: 'Data', 'Prev', and 'Next'. The rows are numbered 1 through 8. The data values are A, B, C, and D, and the pointer values are 8, 1, 4, and 6, respectively. The connections are as follows: Node 1 (A) has Prev=8 and Next=4; Node 4 (B) has Prev=1 and Next=6; Node 6 (C) has Prev=4 and Next=8; Node 8 (D) has Prev=6 and Next=1.

	Data	Prev	Next
1	A	8	4
2			
3			
4	B	1	6
5			
6	C	4	8
7			
8	D	6	1

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 30

- The variable head contains the address of the first element of the list i.e. 1 hence the starting node of the list contains data A is stored at address 1.
- Since, each node of the list is supposed to have three parts therefore, the starting node of the list contains address of the last node i.e. 8 and the next node i.e. 4.
- The last node of the list that is stored at address 8 and containing data as 6, contains address of the first node of the list as shown in the image i.e. 1.
- In circular doubly linked list, the last node is identified by the address of the first node which is stored in the next part of the last node therefore the node which contains the address of the first node, is actually the last node of the list.

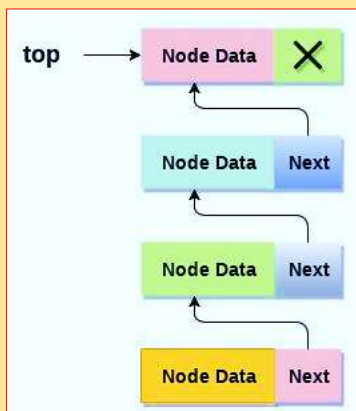
Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

31

## STACK USING LINKED LIST (LINKED STACK)

- In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

32

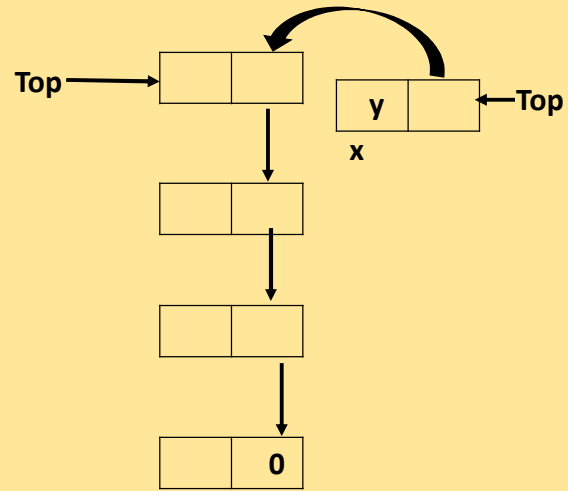


❖ **Algorithm to insert an element in to the stack (PUSH operation)**

```

Algorithm add (i , y)
{
// add y in to  $i^{th}$  stack
new (x);
read (y);
x.data=y;
if(top[i]≠ nil)
{
top[i] = x;
x.link = nil;
}
else
{
x.link=top[i];
top[i]=x;
}
}

```



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

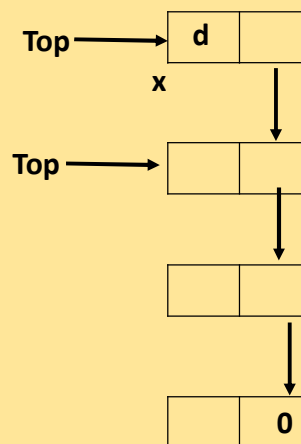
33

❖ **Algorithm to delete an item from a linked stack (POP operation)**

```

Algorithm add (i , y)
{
// Delete top node from  $i^{th}$  stack and set to y
if(top[i]≠ nil)
{
print ("stack is empty");
}
else
{
x = top[i];
y=x.data;
top[i]=x.link;
dispose(x);
}
}

```



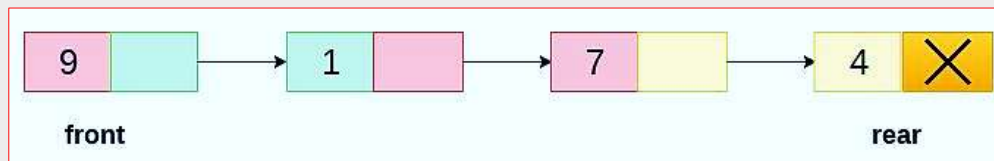
Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

34

## QUEUE USING LINKED LIST(LINKED QUEUE)

- In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.
- In the linked queue, there are two pointers maintained in the memory i.e. **front** pointer and **rear** pointer.
- The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

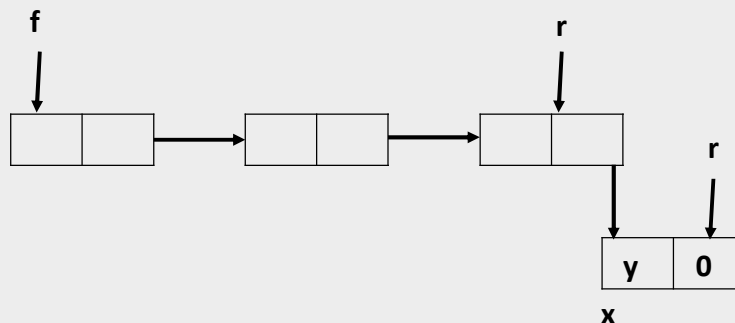
35

### ❖ Enqueue an item in to linked queue (Insertion)

```

Algorithm enqueue (i , y)
{
  // add y in to ithQueue
  new (x);
  x.data = y;
  x.link = nil;
  if(front[i] != nil)
  {
    front[i].link = x;
    rear[i] = x;
  }
  else
  {
    rear[i].link = x;
    rear[i] = x;
  }
}

```



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

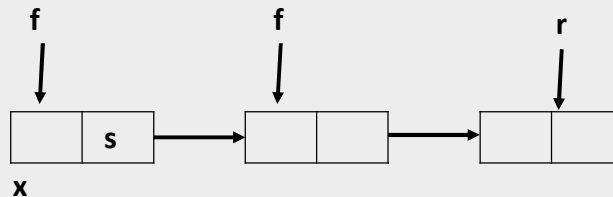
36

## ❖ Dequeue an item from linked queue (deletion)

```

Algorithm dequeue (i , y)
{
  if(front[i] == nil)
  {
    print("Empty Queue");
  }
  else
  {
    x=front[i];
    y=x.data;
    front[i]=x.link;
    dispose(x);
  }
}

```



Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

37

## MEMORY ALLOCATION SCHEME

**FIRST FIT** - In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

**BEST FIT** - The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

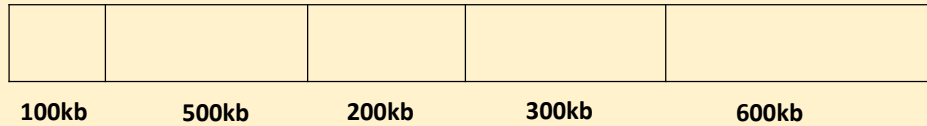
**WORST FIT** - In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

38

**Q:** Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?



### FIRST FIT

212K is put in 500K partition (remaining  $500k - 212k = 288k$ )

417K is put in 600K partition (remaining  $600k - 417k = 183k$ )

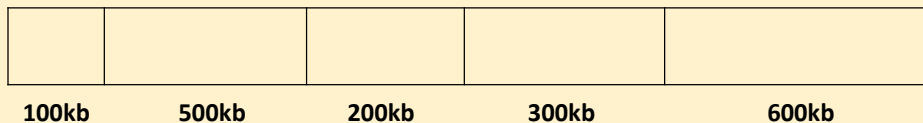
112K is put in 288K partition (new partition  $288K = 500K - 212K$ )

426K must wait

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

39



### BEST FIT

212K is put in 300K partition

417K is put in 500K partition

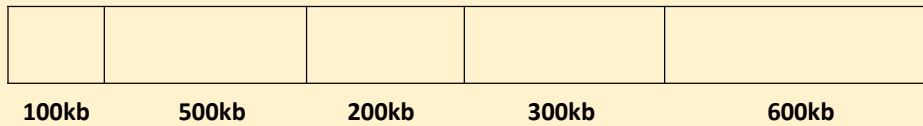
112K is put in 200K partition

426K is put in 600K partition

Prepared By Mr.EBIN PM, AP, IESCE

EDULINE

40

**WORST FIT**

212K is put in 600K partition (remaining  $600k - 212k = 388k$ )

417K is put in 500K partition

112K is put in 388K partition

426K must wait

In this example, best-fit turns out to be the best.