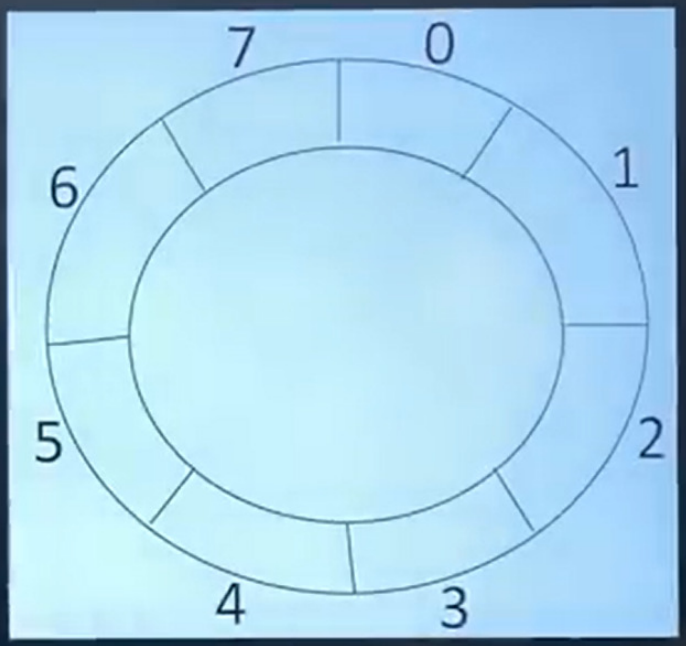
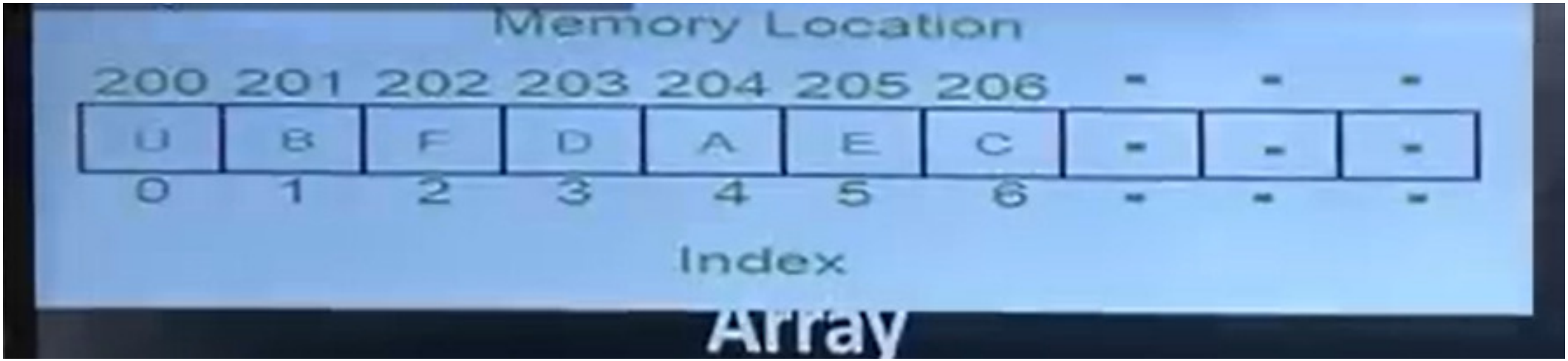
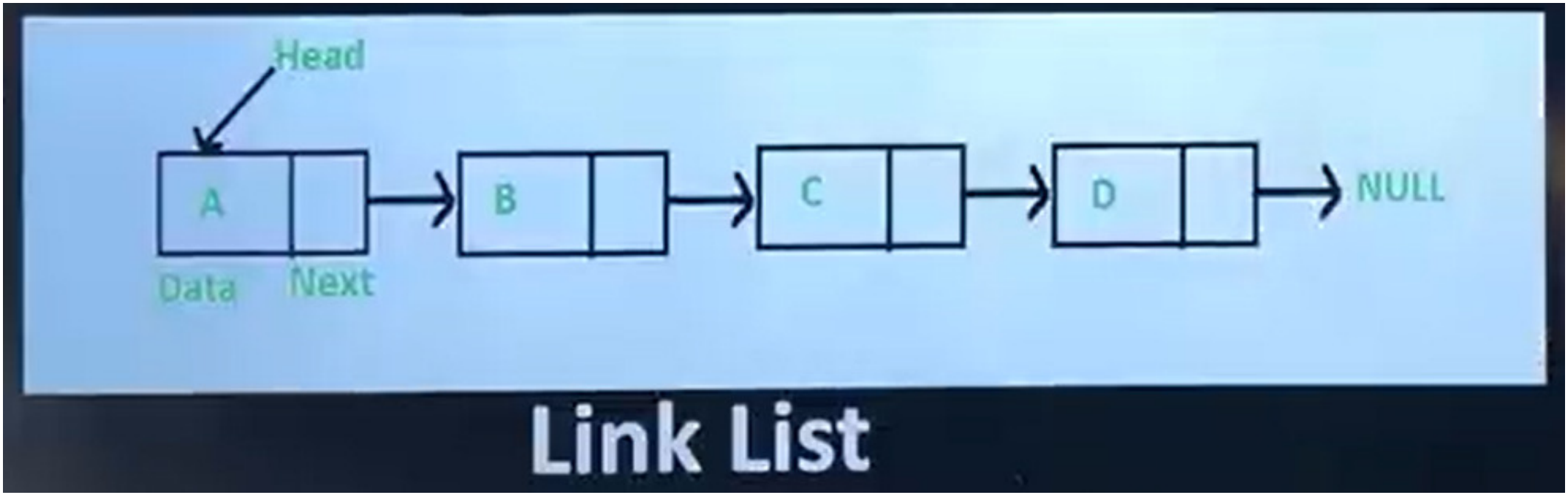
Circular Queue

Insertion

```
Enqueue(QUEUE, N, F, R, ITEM)
{
    if ((F==0 && R==N-1) :: (F == R + 1))
        Write over flow and exit
    if (F = = -1)
        Set F = 0 && R = 0
    Else
        R = (R + 1)%N
    Queue[R] = ITEM
}
```




Memory Location
200 201 202 203 204 205 206 - - -
U B F D A E C - - -
0 1 2 3 4 5 6 - - -
Index
Array


Head
A
B
C
D
NULL
Data
Next
Link List

(Ch-3) Linked lists

```
int main()
{
    Node* head = createNode(10);
    head->next = createNode(20);
    head->next->next = createNode(30);
    printf("Linked List: ");
    traverseList(head);
    return 0;
```

Implementation of link list

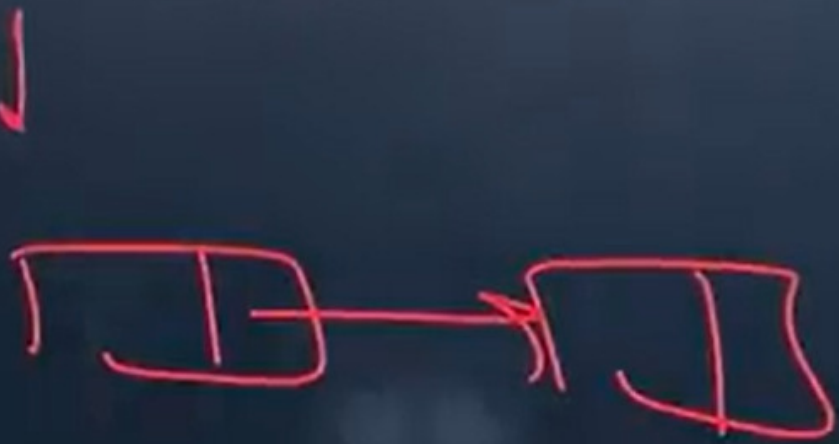
```
struct node
{
    int data;
    struct node *next;
};
```

(Ch-3) Linked lists

```c
#include <stdio.h>
#include <stdlib.h>
// Define the Node structure
typedef struct Node
{
    int data;
    struct Node* next;
}Node;

Node* createNode(int data)
{
    Node* newNode = (Node*) malloc(sizeof(Node));
    if (!newNode)
    {
        printf("Memory error\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

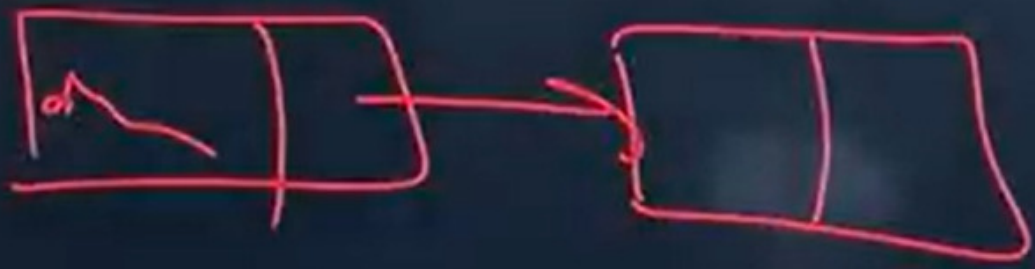
(Ch-4) Stack

```
void initialize(Stack* s)
{
    s->top = NULL;
}

int isEmpty(Stack* s)
{
    return s->top == NULL;
}
```



(Ch-4) Stack

```
typedef struct Node
{
    int data;
    struct Node* next;
} Node;

typedef struct
{
    Node* top;
} Stack;
```



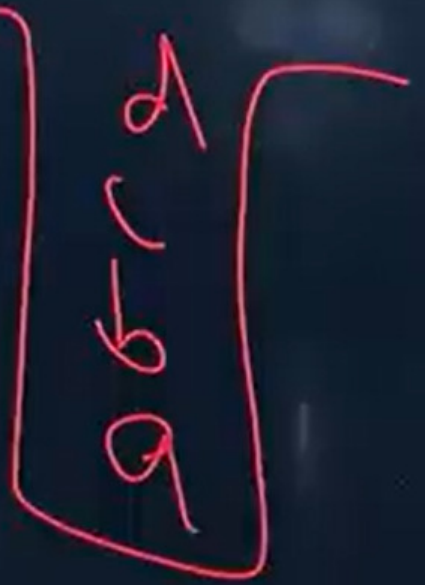
(Ch-4) Stack

```c
int main()
{
    char str[] = "Hello, World!";
    printf("Original String: %s\n", str);
    reverseString(str);
    printf("Reversed String: %s\n", str);
    return 0;
```

(Ch-4) Stack

```
void reverseString(char str[])
{
    int length = strlen(str);
    Stack s;
    initialize(&s);
    for (int i = 0; i < length; i++)
    {
        push(&s, str[i]);
    }
    for (int i = 0; i < length; i++)
    {
        str[i] = pop(&s);
    }
}
```

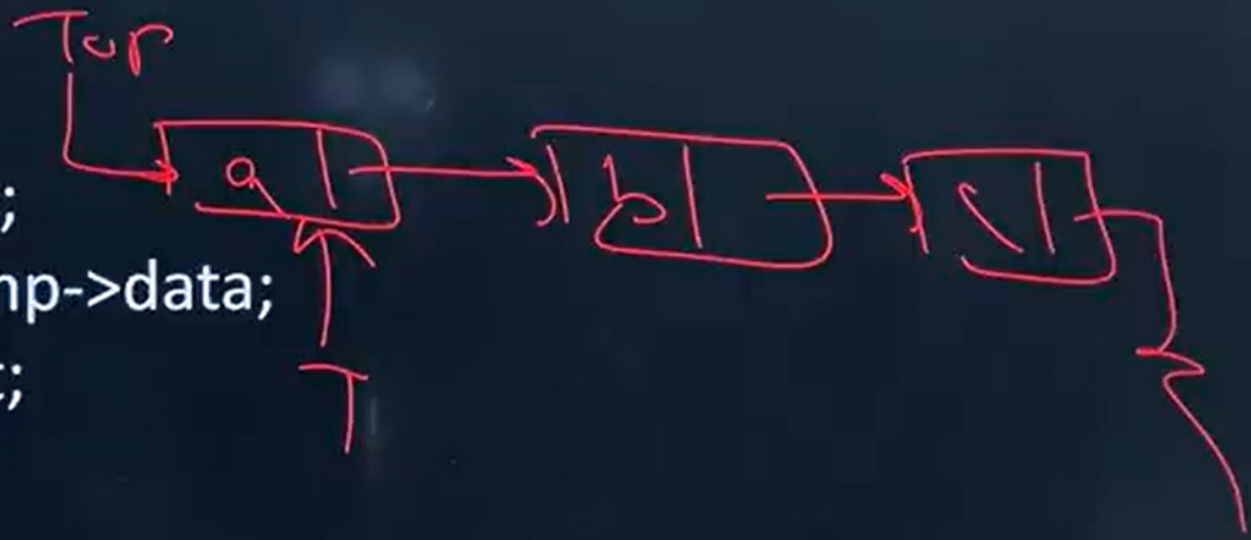
a b c d



(Ch-4) Stack

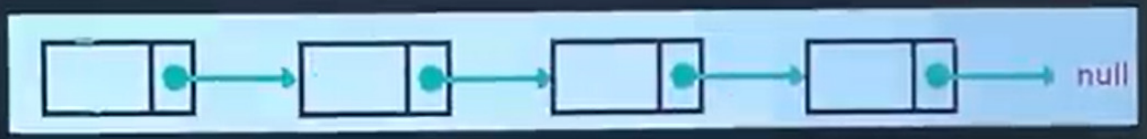
```
void push(Stack* s, int item)
{
    Node* newNode = (Node*)
    malloc(sizeof(Node));
    if (newNode == NULL)
    {
        printf("Stack overflow!\n");
        exit(1); // Exit with an error code
    }
    newNode->data = item;
    newNode->next = s->top;
    s->top = newNode;
}
```

```
int pop(Stack* s)
{
    if (isEmpty(s))
    {
        printf("Stack underflow!\n");
        exit(1);
    }
    Node* temp = s->top;
    int poppedData = temp->data;
    s->top = s->top->next;
    free(temp);
    return poppedData;
}
```

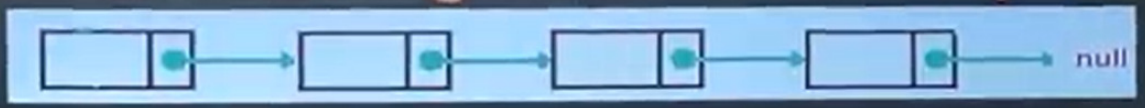


Q Write a C-style pseudocode for searching a key in a link list recursively, where pointer head have the address of the first node of the list?

```
Node* searchKeyRecursive(Node* current, int key)
{
    if (current == NULL)
    {
        return NULL;
    }
    if (current->data == key)
    {
        return current;
    }
    return searchKeyRecursive(current->next, key);
}
```



Q Write a C-style pseudocode for inserting a node with a key in the starting of link-list?

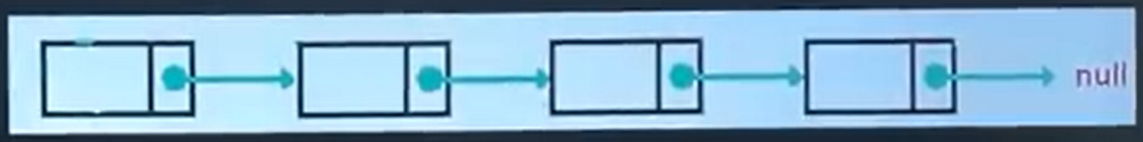



```
void insertAtBeginning(Node** head, int key)
{
    Node* newNode = createNode(key);
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}
```

Q Write a C-style pseudocode for inserting a node with a key after a location in a link-list?

```
void insertAfter(Node* prevNode, int key)
{
    if (prevNode == NULL)
    {
        printf("The given previous node cannot be NULL.\n");
        return;
    }
    Node* newNode = createNode(key);
    newNode->data = data;
    newNode->next = prevNode->next;
    prevNode->next = newNode;
}
```

Q Write a C-style pseudocode for deleting a node from the starting of link-list?



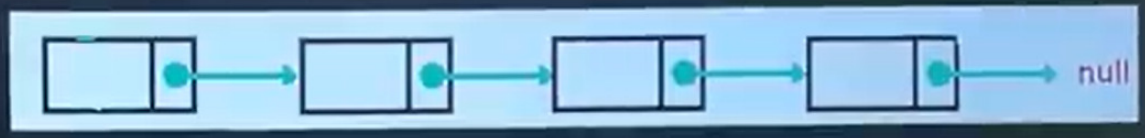

```
void deleteAtBeginning(Node** head)
{
    if (*head == NULL)
    {
        printf("List is already empty.\n");
        return;
    }
    Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}
```

Q Write a C-style pseudocode for deleting a node after a given location from the starting of link-list?

```
void deleteAfter(Node* prevNode)
{
    if (prevNode == NULL || prevNode->next == NULL)
    {
        printf("The given node is NULL or there's no node after it to delete.\n");
        return;
    }
    Node* temp = prevNode->next;
    prevNode->next = temp->next;
    free(temp);
}
```

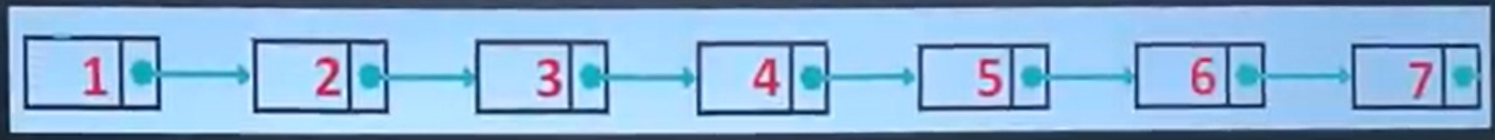
Q Write a C-style pseudocode for reversing a link-list in a iteratively?

```
void reverseListIterative(Node** head)
{
    Node* prev = NULL;
    Node* current = *head;
    Node* next = NULL;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head = prev;

}
```



Basic Operations on a Linked List

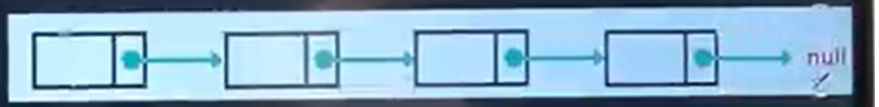
Q The following C function takes a single-linked list of integers as a parameter and rearranges the elements of the list. The function is called with the list containing the integers 1, 2, 3, 4, 5, 6, 7 in the given order. What will be the contents of the list after the function completes execution?




```
struct node
{
        int value;
        struct node *next;
};
void rearrange(struct node *list)
{
        struct node *p, * q;
        int temp;
        if ((!list) || !list->next)
                return;
        p = list;
        q = list->next;
        while(q)
        {
                temp = p->value;
                p->value = q->value;
                q->value = temp;
                p = q->next;
                q = p ? p->next:0;
        }
}
```

Q Write a C-style pseudocode for reversing a link-list in a recursively?

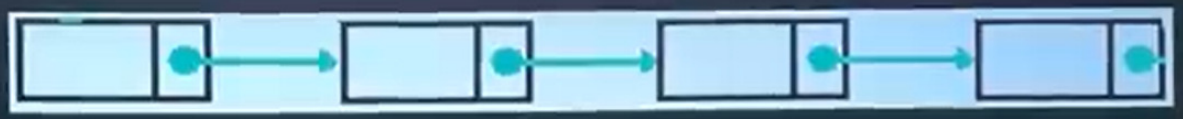
```
Node* reverseListRecursive(Node* head)
{
    if (head == NULL || head->next == NULL)
    {
        return head;
    }
    Node* rest = reverseListRecursive(head->next);
    head->next->next = head;
    head->next = NULL;
    return rest;
}
```



(Ch-3) Linked lists

Basic Operations on a Linked List

Q The following C function takes a simply-linked list as input argument. It modifies the list by moving the last element to the front of the list and returns the modified list. Some part of the code is left blank. Choose the correct alternative to replace the blank line.

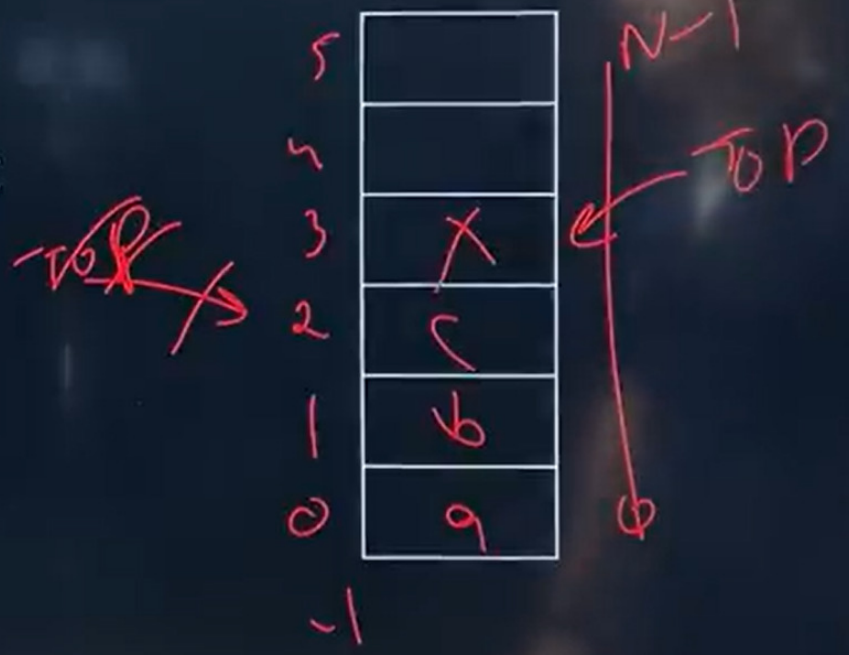


```
typedef struct node
{
        int value;
        struct node *next;
}Node;

Node *move_to_front(Node *head)
{
        Node *p, *q;
        if ((head == NULL: || (head->next == NULL))
                return head;
        q = NULL; p = head;
        while (p-> next !=NULL)
        {
                q = p;
                p = p->next;
        }
        _____________________________
        return head;
}
```

(A) q = NULL; p->next = head; head = p;
(B) q->next = NULL; head = p; p->next = head;
(C) head = p; p->next = q; q->next = NULL;
(D) q->next = NULL; p->next = head; head = p;

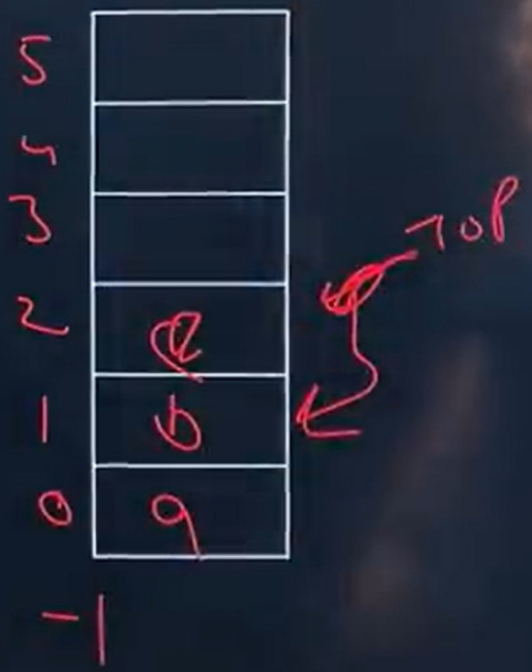
Push operation: - The process of adding new element to the top of stack is called push operation. the new element will be inserted at the top after every push operation the top is incremented by one. in the case the array is full and no new element can be accommodated it is called over-flow condition.

```
PUSH (S, N, TOP, x)
{
    if (TOP==N-1)
        Print stack overflow and exit
    TOP = TOP + 1
    S[TOP] = x
    exit
}
```

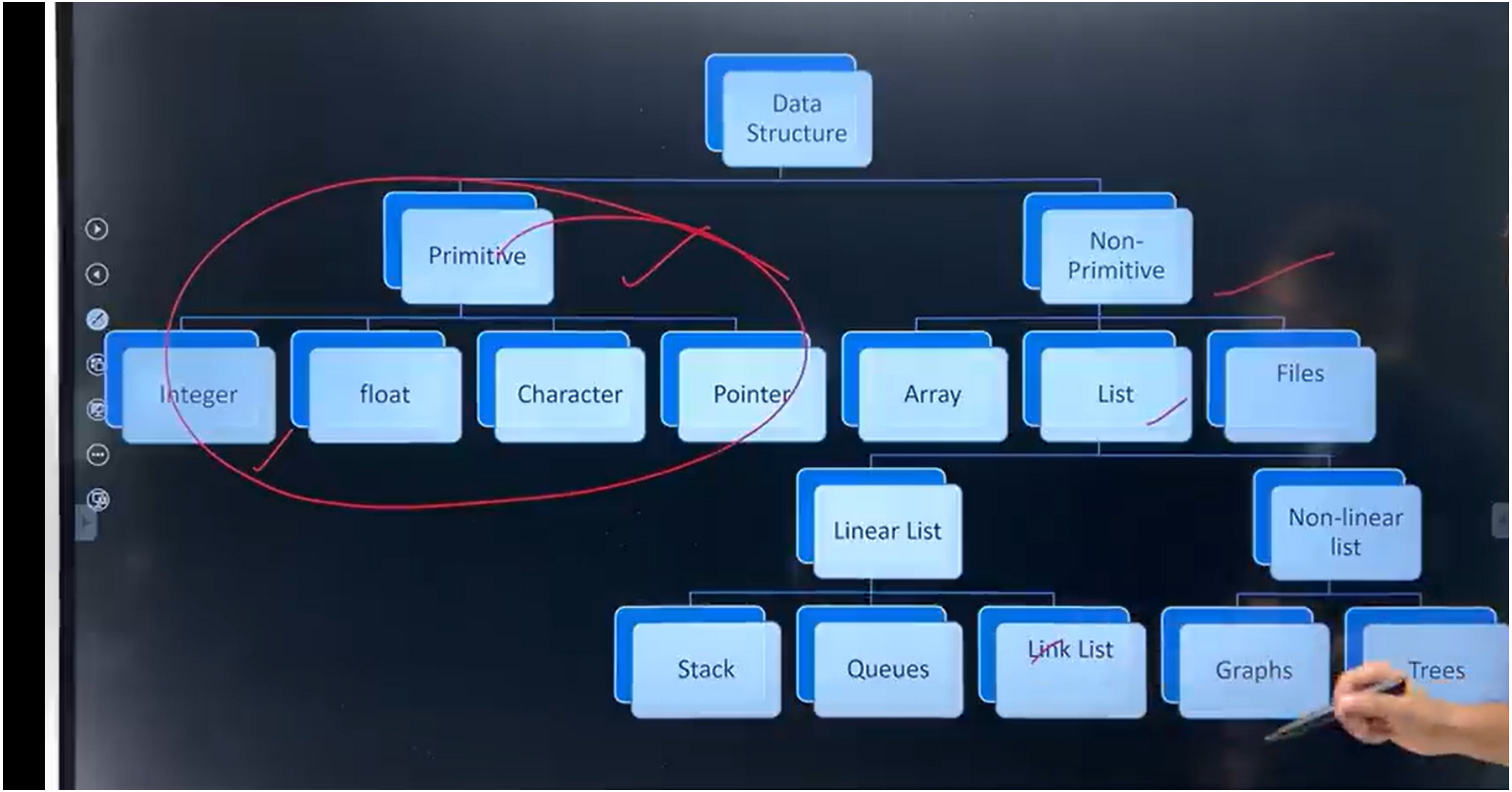


Pop: - The process of deleting an element. from the top of stack is called POP operation, after every POP operation the stack is decremented by one if there is no element in the stack and the POP operation is requested then this will result into a stack underflow condition.

```
POP (S, N, TOP)
{
   if (TOP==-1)
        print underflow and exit
   y = S[TOP]
   TOP=TOP-1
   return(y) and exit
}
```

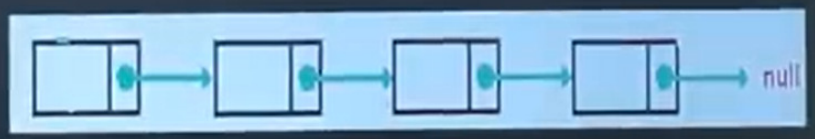


Aspect	Recursion	Iteration
Basic Concept	Function calls itself to solve sub-problems.	Uses loops to repeatedly execute code blocks.
Memory Usage	Typically uses more memory due to call stack.	Uses less memory as it doesn't rely on the call stack.
Termination	Requires a base case to prevent infinite loops.	Requires a loop exit condition.
Ease of Implementation	Can be more intuitive for certain problems.	Often simpler and more straightforward for repetitive tasks.
Performance	Might be slower due to overhead of function calls.	Typically faster due to direct loop mechanics.


Data Structure
Primitive
Non-Primitive
Integer
float
Character
Pointer
Array
List
Files
Linear List
Non-linear list
Stack
Queues
Link List
Graphs
Trees

Aspect	Array	Linked List
Memory Allocation	Contiguous memory locations.	Non-contiguous memory locations.
Size Flexibility	Fixed size.	Dynamic size, can grow or shrink as required.
Access Time	O(1) for direct access due to indexing.	O(n) for accessing an element as it requires traversal.
Insertion/Deletion	O(n) in worst case as shifting may be required.	O(1) if the pointer to the node is known.
Memory Efficiency	More memory efficient for a known size of data.	Extra memory for pointers, which can be overhead for small data sizes.

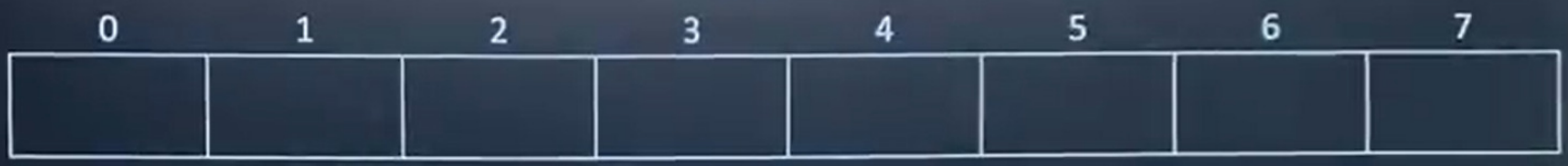
Q Write a C-style pseudocode for Traversing a link list iteratively, where pointer head have the address of the first node of the list?



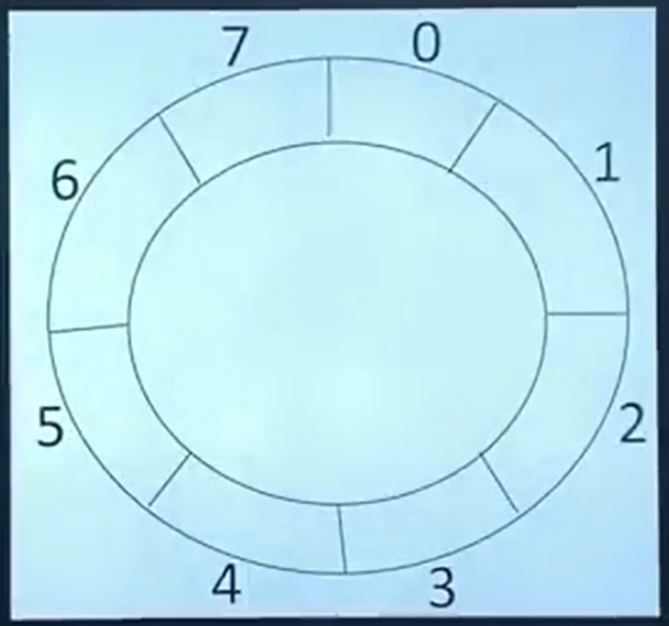

```
void traverseList(Node* head)
{
    Node* current = head;
    while (current != NULL)
    {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}
```

Circular Queue

Deletion

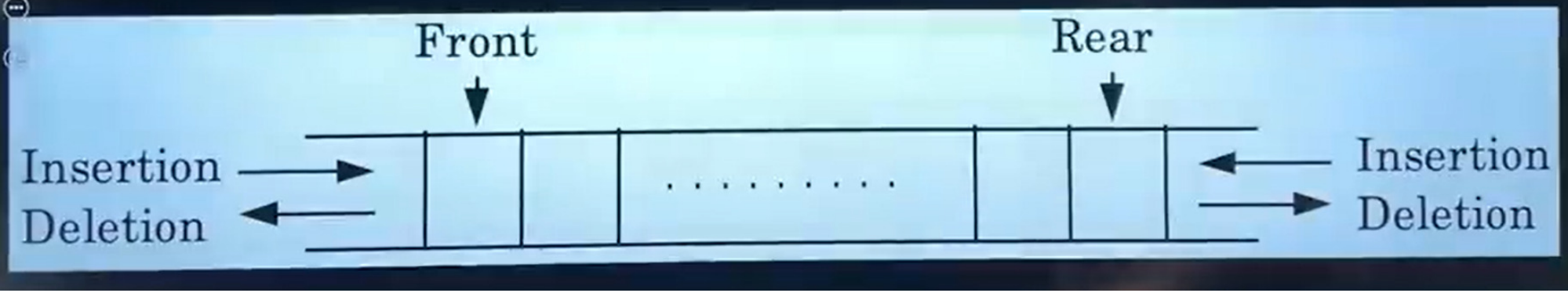



```
Dequeue(QUEUE, N, F, R, ITEM)
{
    if (F == - 1)
        Write under flow and exit
    ITEM = QUEUE[F]
    if (F = = R)
        Set F = -1 && R = -1
    Else
        F = (F + 1)%N
    Return item
}
```

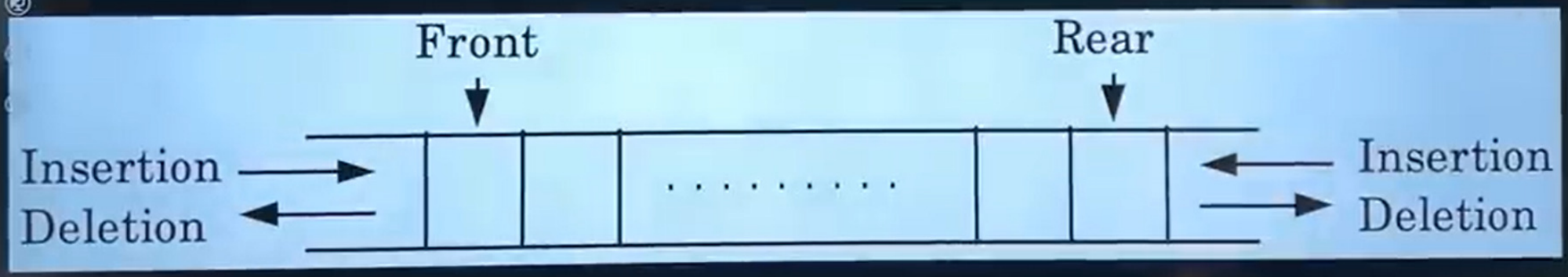


Dequeue

- In a dequeue, both insertion and deletion operations are performed at either end of the queues. That is, we can insert an element from the rear end or the front end. Also deletion is possible from either end.
- This dequeue can be used both as a stack and as a queue.
- There are various ways by which this dequeue can be represented. The most common ways of representing this type of dequeue are :
  - Using a doubly linked list
  - Using a circular array



- Types of dequeue :
  - **Input-restricted dequeue** : In input-restricted dequeue, element can be added at only one end but we can delete the element from both ends.
  - **Output-restricted dequeue** : An output-restricted dequeue is a dequeue where deletions take place at only one end but allows insertion at both ends.

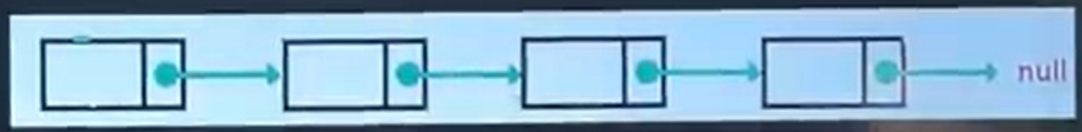


Priority Queue

- A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules.
  - An element of higher priority is processed before any element of lower priority
  - Two element with the same priority are processed according to the order in which they were added to the queue.

Q Write a C-style pseudocode for searching a key in a link list iteratively, where pointer head have the address of the first node of the list?

```
Node* searchKeyIterative(Node* head, int key)
{
    Node* current = head;
    while (current != NULL)
    {
        if (current->data == key)
        {
            return current;
        }
        current = current->next;
    }
    return NULL;
}
```



Q Write a C-style pseudocode for Traversing a link list recursively, where pointer head have the address of the first node of the list?

```
void traverseListRecursive(Node* current)
{
    if (current == NULL)
    {
        printf("NULL\n");
        return;
    }
    printf("%d -> ", current->data);
    traverseListRecursive(current->next);
}
```