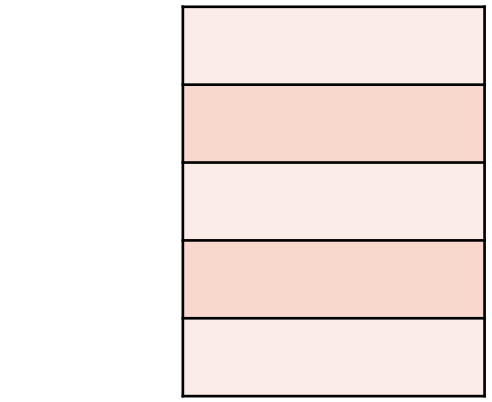
# Data Structures

Stacks, Queues and Linked List

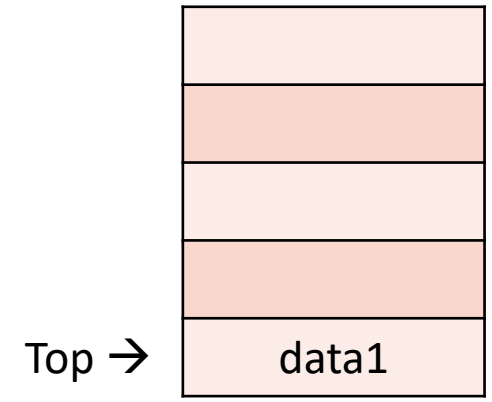
# Stack

- It is a type of Linear Data Structures using C
- Follows LIFO: Last In First Out
- Only the top elements are available to be accessed
- Insertion and deletion takes place from the top
- Major operations:
  - push(element) – used to insert element at top
  - pop() – removes the top element from stack
  - isEmpty() – returns true is stack is empty
  - peek() – to get the top element of the stack
  - isFull() – returns true is stack is full

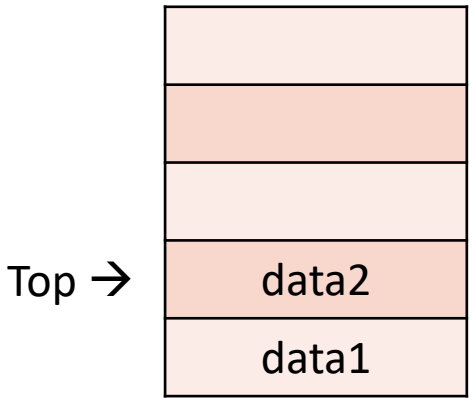
# Full Ascending Stack



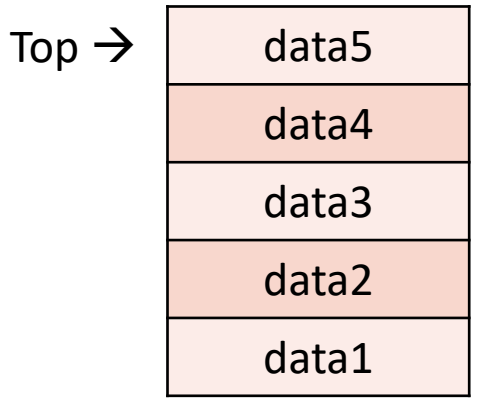

Empty Stack



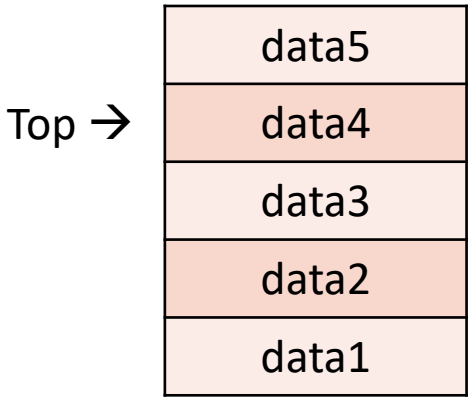

Push - First element



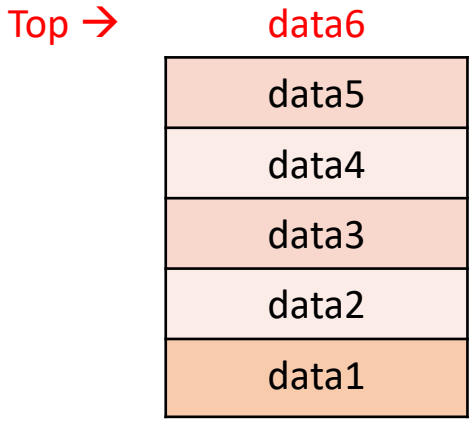

Push - Second element



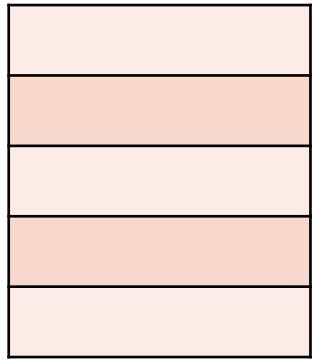

Full Stack




Pop - Fifth element




Over Flow – Push with Full stack




Under Flow – Pop an Empty Stack

# Stack - Structure

```
typedef struct stackk
{
    int top;
    unsigned int size;
    int *array;
}stack_type;
```

# Stack – Memory Allocation

```
stack_type* create(unsigned int size)
{
    //Allocating memory for stack structure
    stack_type *stack_A = (stack_type*) malloc(sizeof(stack_type));
    stack_A->size = size;
    stack_A->top = -1;
    //Allocating continuous memory of specified size
    stack_A->array = (int*)malloc(stack_A->size * sizeof(int));
    return stack_A;
}
```

# Stack – Status check

```
int isFull(stack_type *stack1)
{
    return (stack1->top == ((stack1->size)-1));
}

int isEmpty(stack_type *stack1)
{
    return (stack1->top == -1);
}
```

# Stack - Push

```
int push(stack_type *stack1, int data)
{
    if (isFull(stack1))
        return 1;
    stack1->array[++stack1->top] = data;
    return 0;
}
```

# Stack - Pop

```
int pop(stack_type *stack1)
{
    if (isEmpty(stack1))
        return 1;
    else
        {
            printf("\nPoped element: %d",stack1->array[stack1->top--]);
            return 0;
        }
}
```

# Stack - Peek

```
int peek(stack_type *stack1)
{
    if (isEmpty(stack1))
        return 0;
    else
        return 1;
}
```

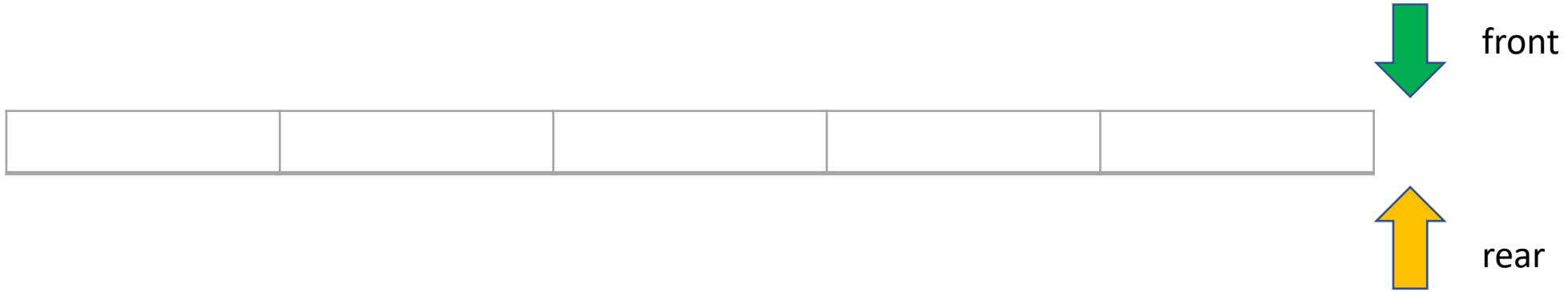
# Stack – User Input Output

- Create a stack
- Display a list of stack functions that can be performed
  - Push
  - Pop
  - Peek
  - isFull
  - isEmpty
- Push
  - Acquire the data
  - Check stack status
  - Display top of the stack if the stack is full

# Stack – User Input Output

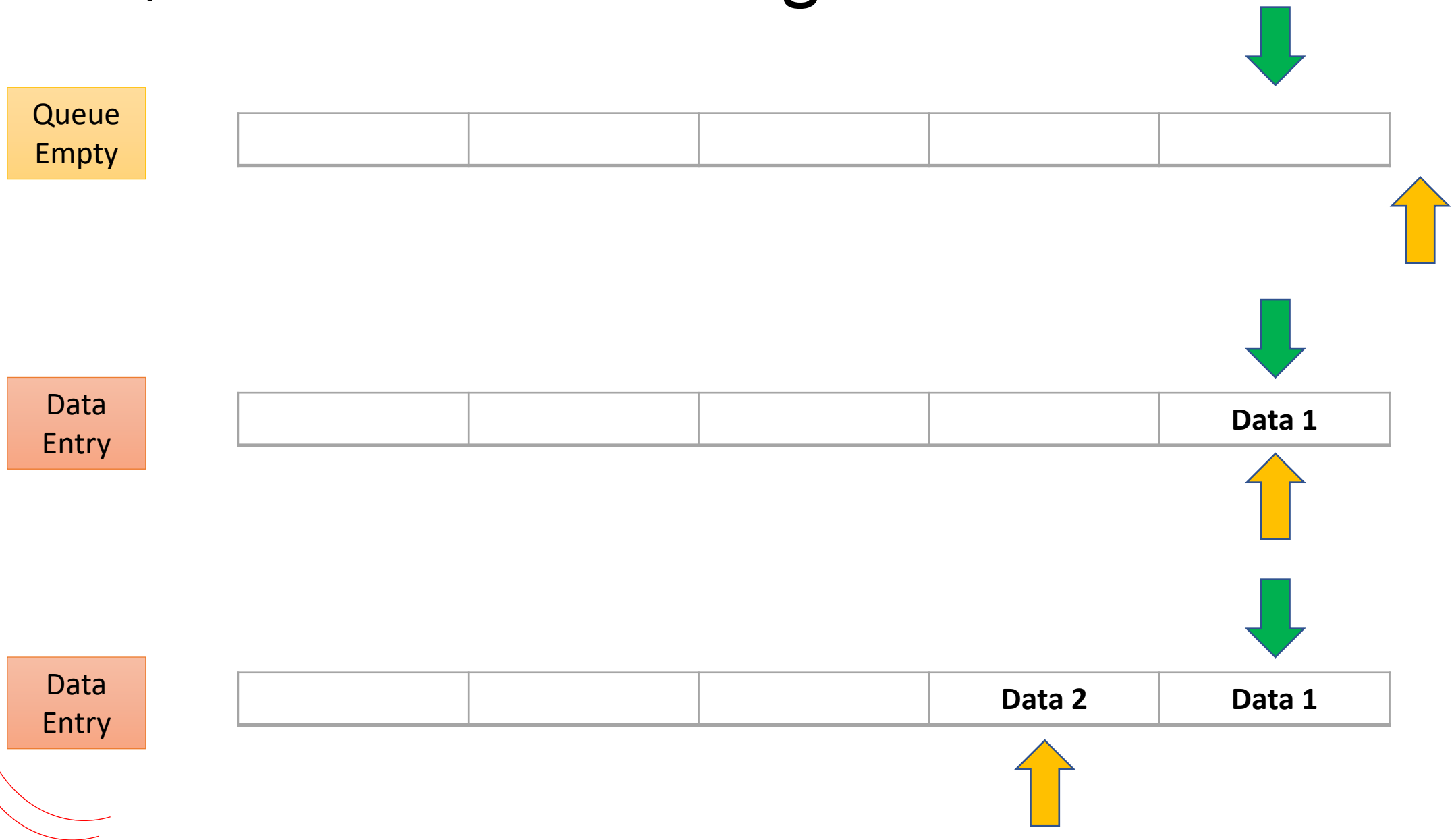
- Pop
  - Check stack status
  - Pop the element and display top of the stack
- Peek
  - Display top of the stack if the stack is not empty
- isFull and isEmpty
  - Check the stack status and print the statement appropriately
- Continue the above steps until the user wants to exit

# Queue

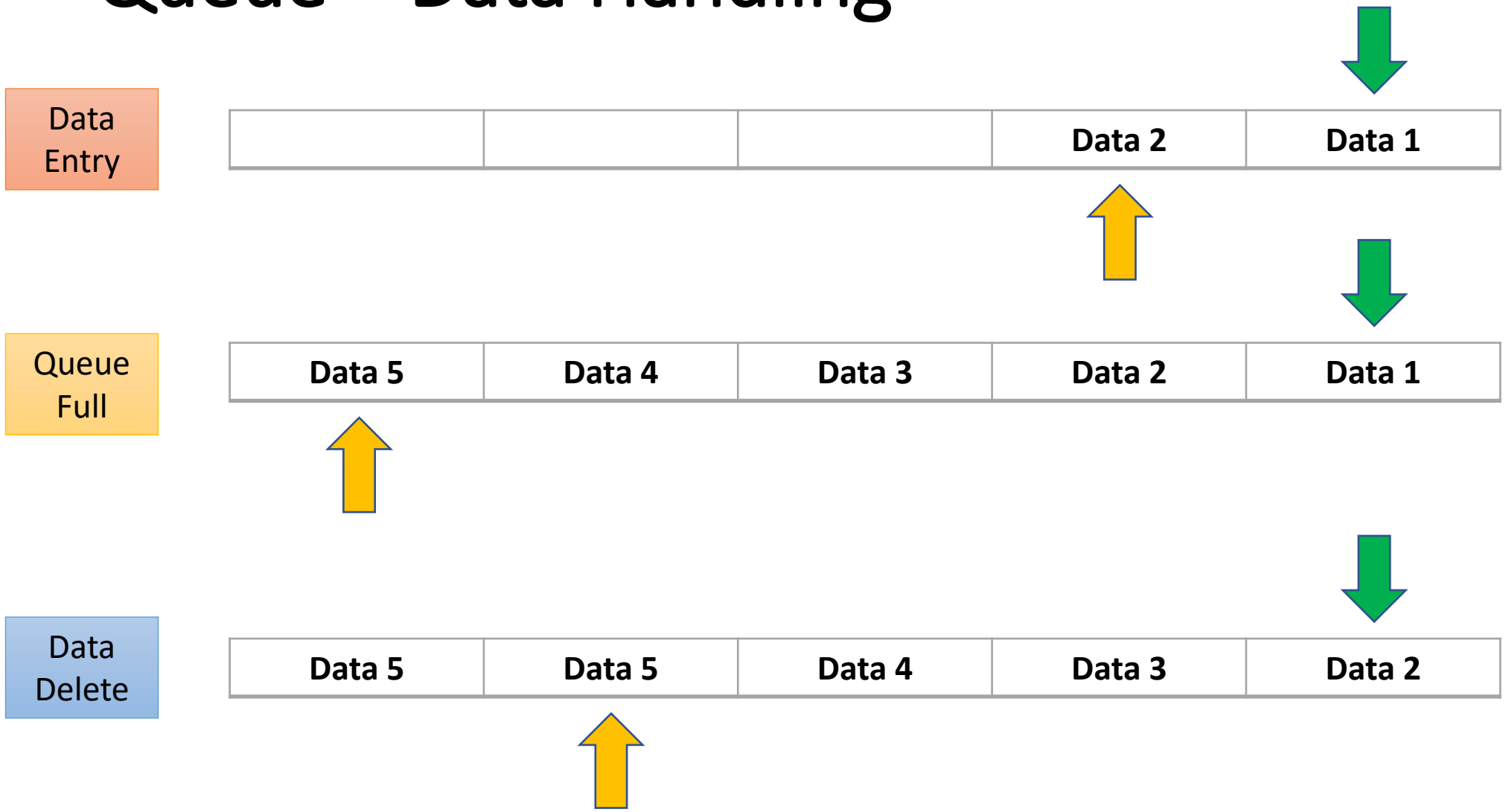



- Consider an empty queue
- It follows FIFO logic to handle data
- It requires two variables/pointers to track front and rear of the Queue

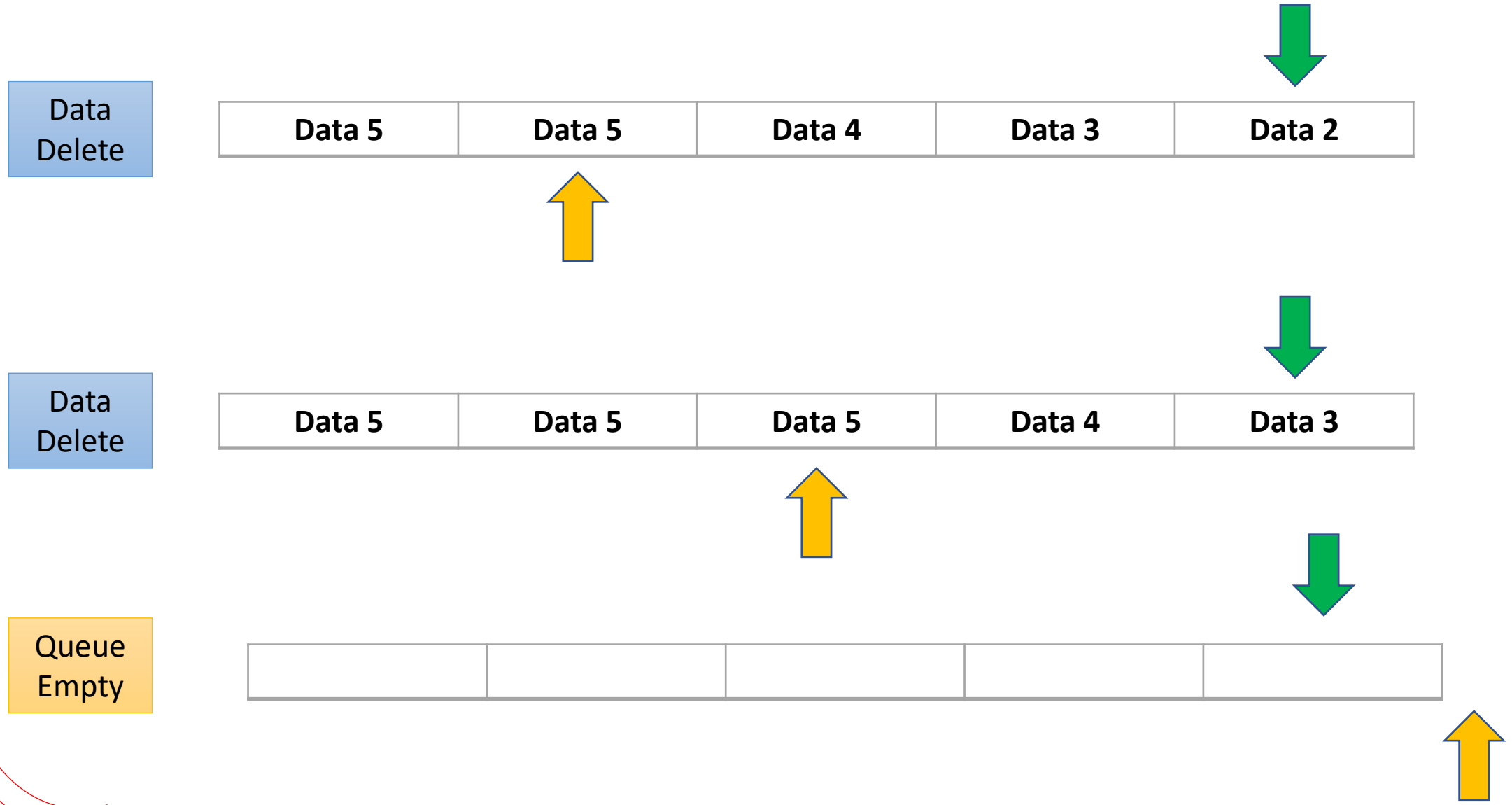
# Queue – Data Handling



# Queue – Data Handling



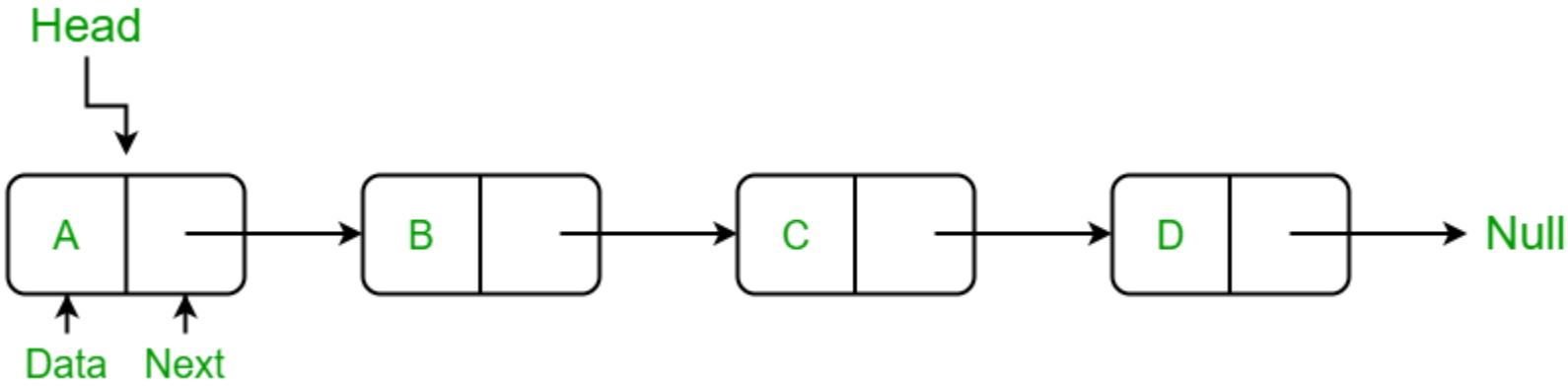
# Queue – Data Handling



# Queue - Functions

- Fix the size of the queue
- Front – points the start of the queue
- Rear – points the last entry of the queue
- Rear < Front – Underflow
- Rear > Size – Overflow
- During Data entry – Rear is incremented
- During Data removal – Data pointed by front is removed, entire queue shifted with rear decremented

# Linked List




- Like arrays, Linked List is a linear data structure
- Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers
- They include a series of connected nodes
- Each node stores the data and the address of the next node

# Array limitations

- The size of the arrays is fixed
- Insertion of a new element / Deletion of a existing element in an array of elements is expensive

# Linked List advantages

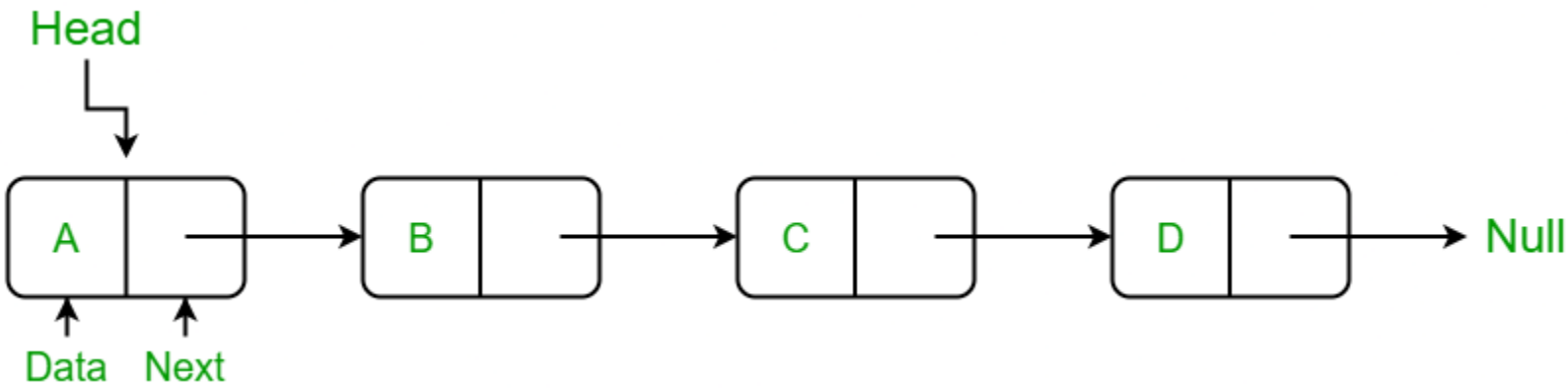
- Dynamic Array
- Ease of Insertion/Deletion

# Linked limitations

- Random access is not allowed – Each time access should start from head node
- Extra memory space for a pointer is required with each element of the list
- Not cache friendly - locality of reference is not there in linked lists like array index

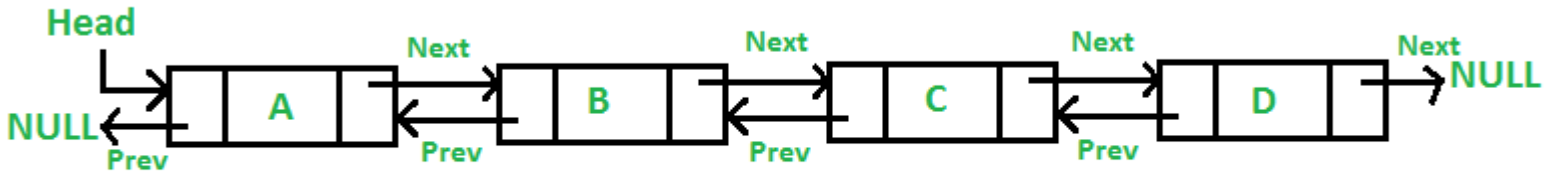
# Types

- Simple Linked List
  - One can move or traverse the linked list in only one direction



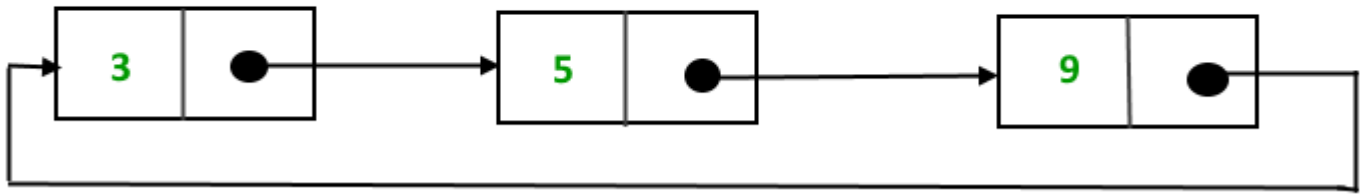
# Types

- Doubly Linked List
  - One can move or traverse the linked list in both directions (Forward and Backward)



# Types

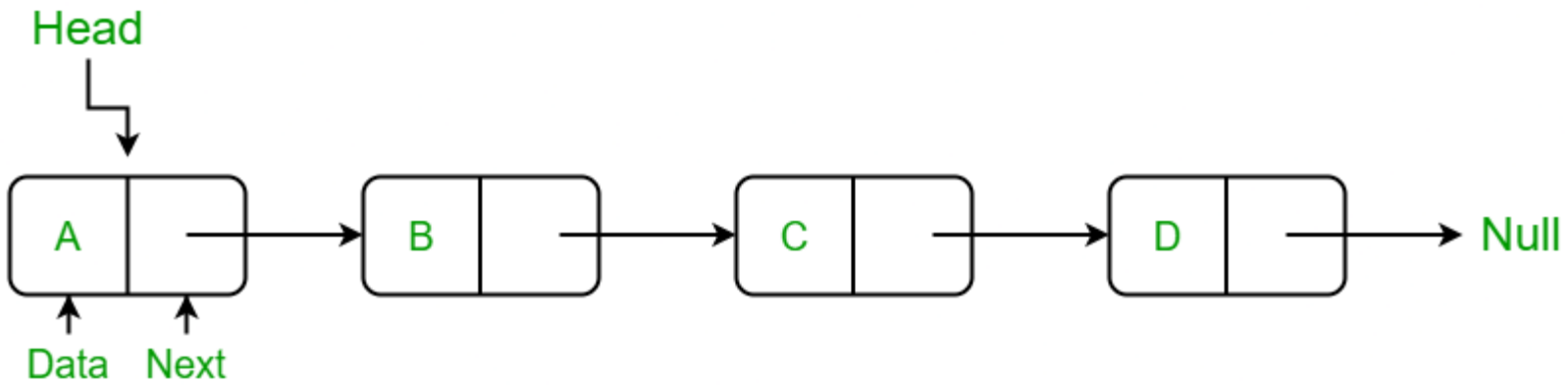
- Circular Linked List
  - The last node of the linked list contains the link of the first/head node of the linked list in its next pointer and the first/head node contains the link of the last node of the linked list in its previous pointer



# Linked list - Operations

- Deletion
- Insertion
- Search
- Display

# Linked List - Node



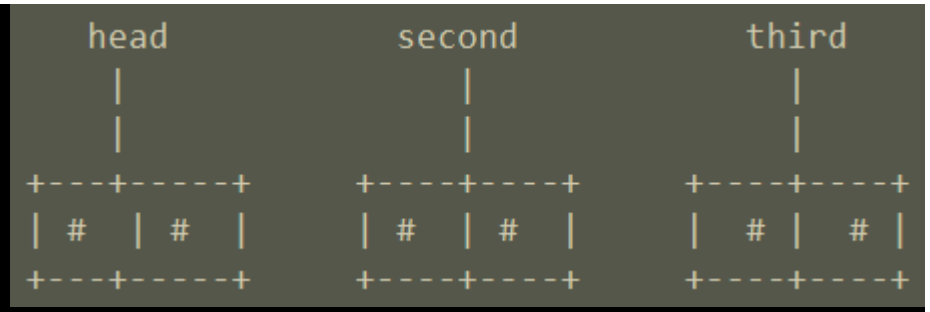


```
typedef struct node_content
{
    int data;
    struct Node* next;
}Node;
```

# Node creation

```
Node *head = NULL;
Node *second = NULL;
Node *third = NULL;


head = (Node *)malloc(sizeof(Node));
second = (Node *)malloc(sizeof(Node));
third = (Node *)malloc(sizeof(Node));
```



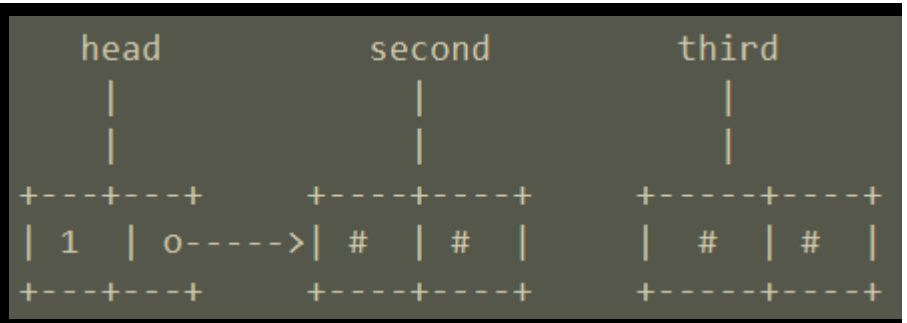

A Null Pointer is a pointer that does not point to any memory location

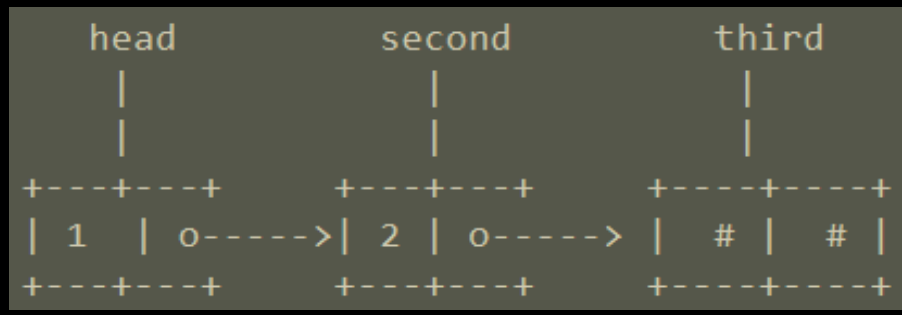
# Node - Linking

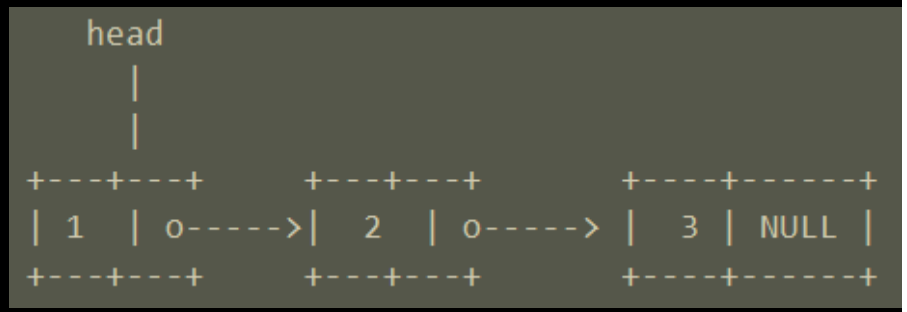
```
head->data = 1;
head->next = second;

second->data = 2;
second->next = third;

third->data = 3;
third->next = NULL;
```



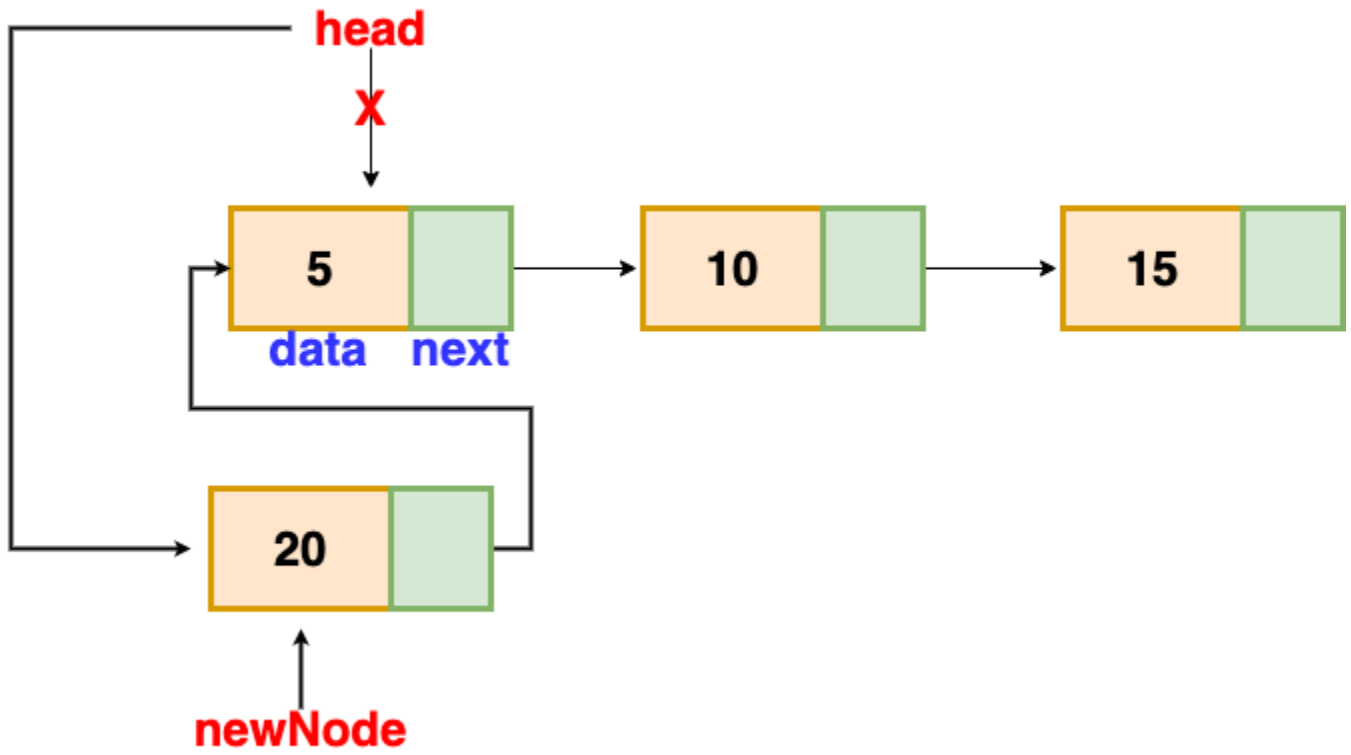
# Traversal of List

```
void printList(Node *n)
{
    printf("\nLinked List Elements:\n");
    while (n != NULL) {
        printf(" %d ", n->data);
        printf("-->");
        n = n->next;
    }
}
```

Output

```
Linked List Elements:
 1 --> 2 --> 3 -->
```

# Insertion - Beginning

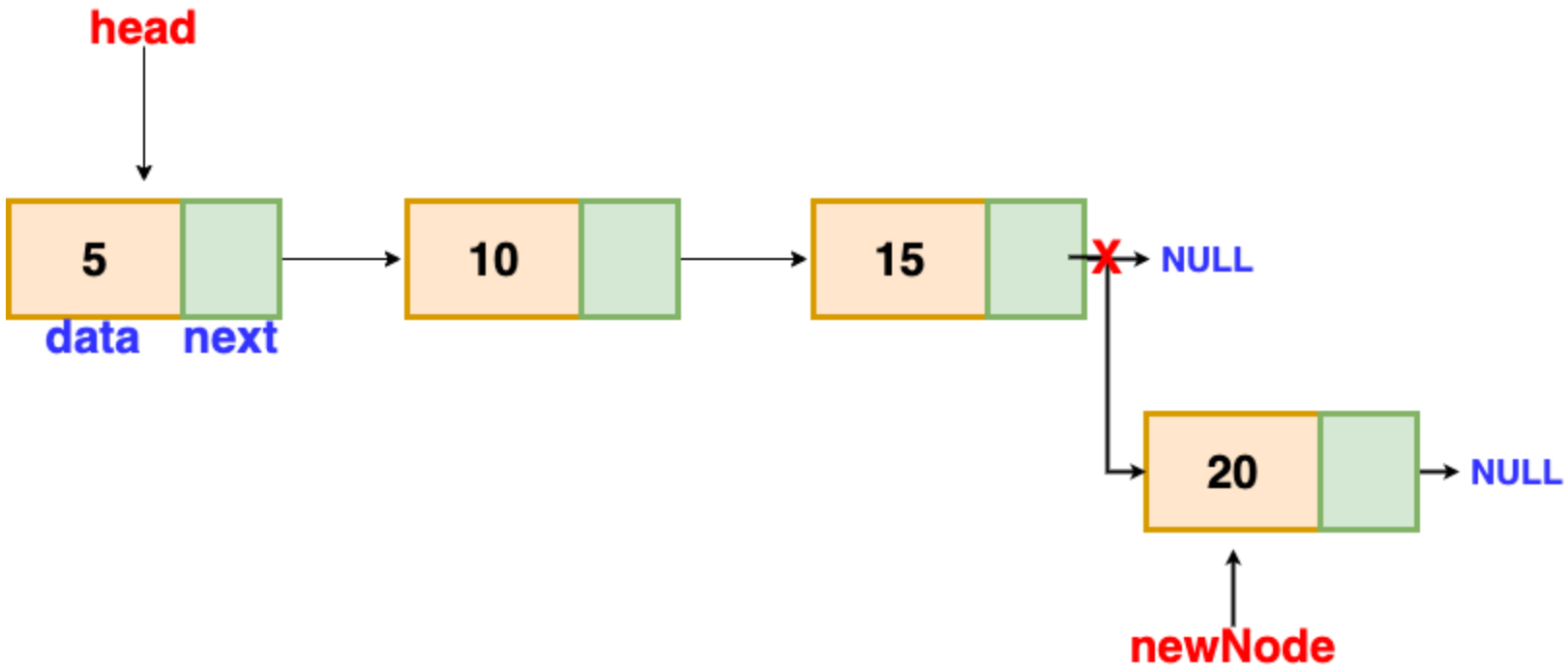



Insertion at the beginning

# Insertion - Beginning

```
Node* insert_begin(Node *head)
{
    Node *newNode;
    newNode = (Node *)malloc(sizeof(Node));
    newNode->data = 4;
    newNode->next = head;
    head = newNode;
    return head;
}
```

# Insertion - End




Insertion at the end

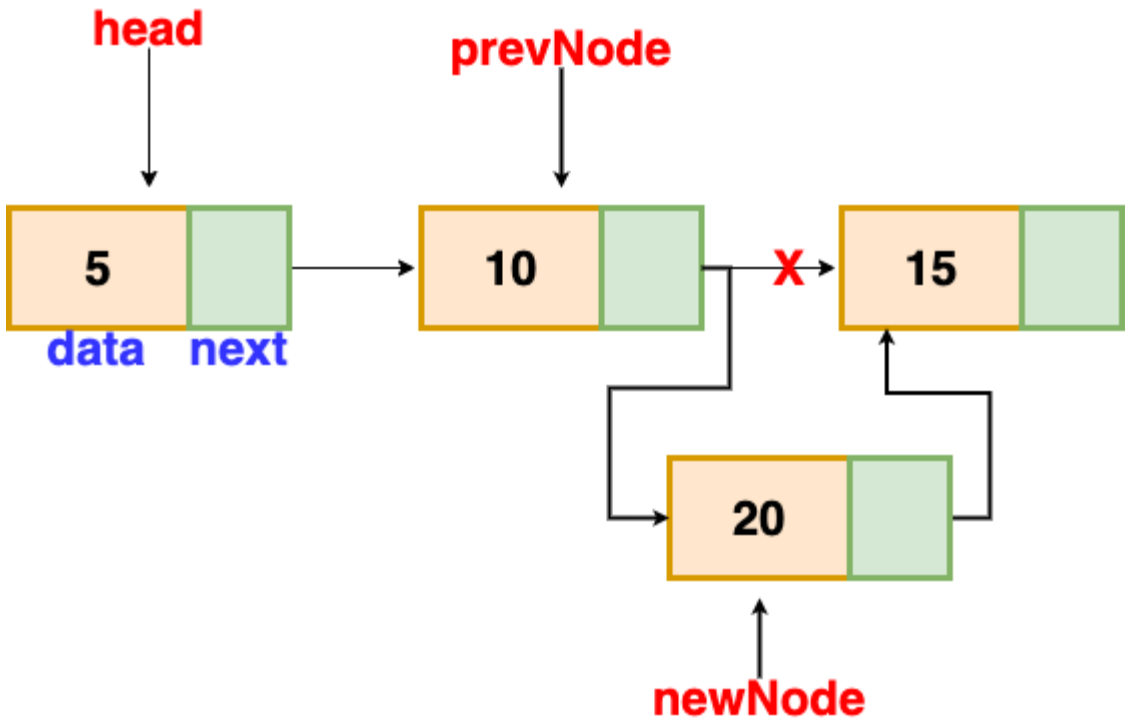
# Insertion - End

```
Node* insert_end(Node *head)
{
    Node *newNode;
    newNode = (Node *)malloc(sizeof(Node));
    newNode->data = 5;
    newNode->next = NULL;

    Node *temp = head;
    while(temp->next != NULL)
    {
        temp = temp->next;
    }

    temp->next = newNode;
    return head;
}
```

# Insertion - middle



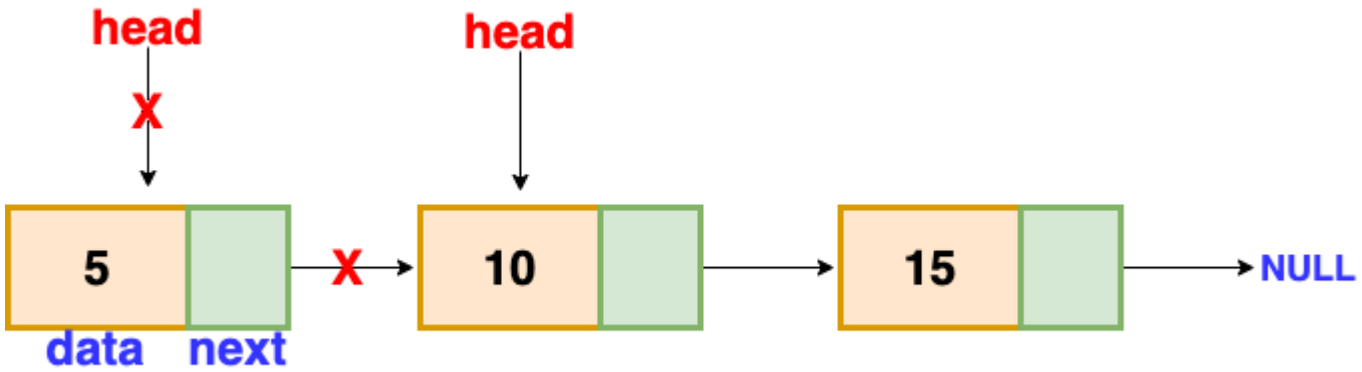

Insertion after a given node

# Insertion - middle

```
Node* insert_element(Node *list, int position, int data)
{
    int size=0, tmp=0;
    Node *list1 = list;
    while (list1 != NULL)
    {
        size++;
        list1 = list1->next;
    }
    list1 = list;
    printf("\nList size: %d",size);
    if(position>size)
    {
        printf("\nPosition not matching the list size");
        return list;
    }
```

```
else
    {
        Node *newNode;
        newNode = (Node *)malloc(sizeof(Node));
        newNode->data = data;
        newNode->next = NULL;
        while (list1 != NULL)
        {
            tmp++;
            if(tmp == (position-1))
            {
                newNode->next = list1->next;
                list1->next = newNode;
            }
            list1 = list1->next;
        }
    }
    return list;
}
```

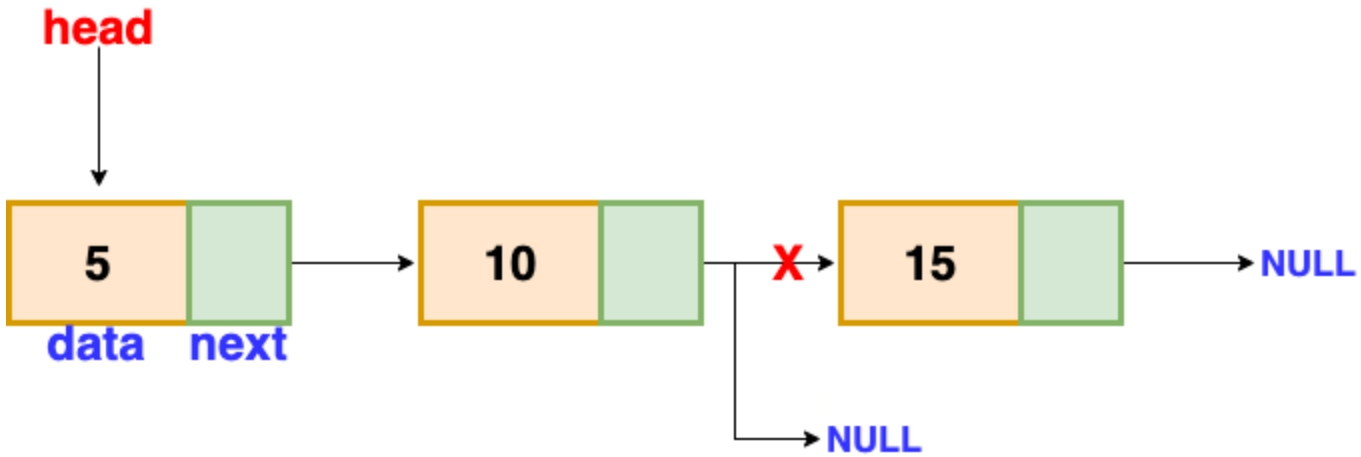
# Delete - Beginning



# Delete - Beginning

```
Node* delete_begin(Node *head)
{
    Node *tmp;
    tmp = head;
    head = head->next;
    tmp->next = NULL;
    free(tmp);
    return(head);
}
```

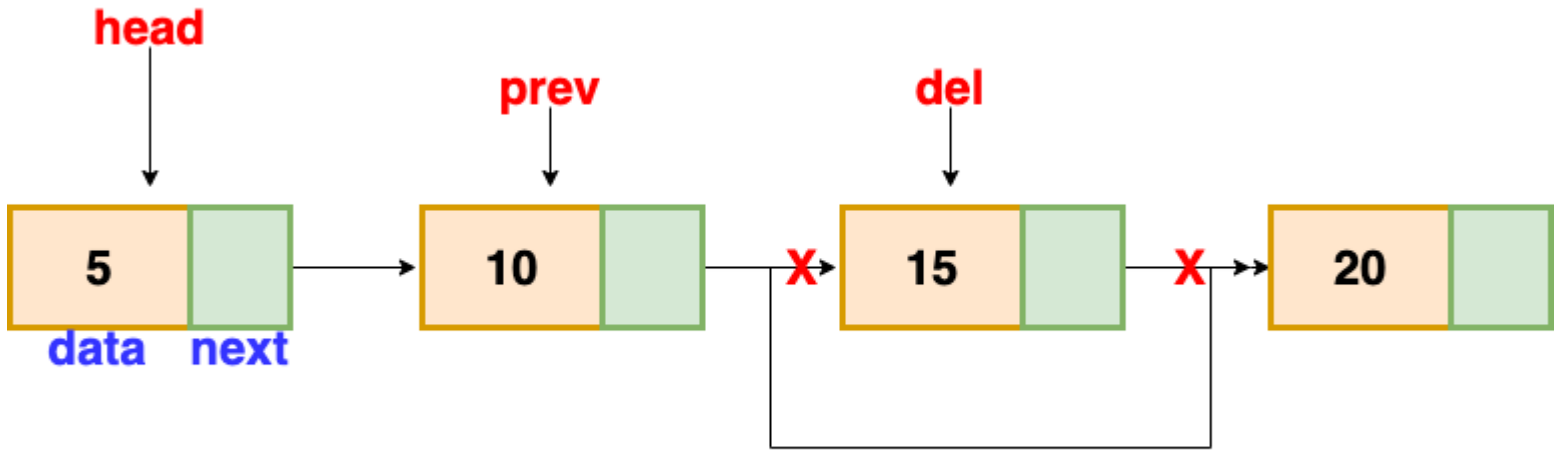
# Delete - End



# Delete - End

```
Node* delete_end(Node *head)
{
    Node *temp = head;
    Node *temp1 = temp->next;
    while((temp1->next) != NULL)
    {
        temp = temp->next;
        temp1 = temp1->next;
    }
    temp->next = NULL;
    free(temp1);
    return head;
}
```

# Delete - Middle




Deleting a Node in Linked List

# Delete - Middle

```
Node* delete_element(Node *list, int position)
{
    int size=0, tmp=0;
    Node *list1 = list;
    Node *list2 = list->next;
    size = llsize(list);
    if(position>size)
    {
        printf("\nPosition not matching the list size");
        return list;
    }
    else
    {
        while (list1 != NULL)
        {
            tmp++;
            if(tmp == (position-1))
            {
                list1->next = list2->next;
                list2->next = NULL;
                free(list2);
                break;
            }
                list1 = list1->next;
                list2 = list2->next;
        }
    }
    return list;
}
```

# Reference


- https://www.geeksforgeeks.org/what-is-linked-list/
- https://www.programiz.com/dsa/linked-list-operations
- https://www.programiz.com/dsa/doubly-linked-list