Module IV
Linked List

Dynamic Data Structure

- Kind of data structure that *changes its size during runtime*.
- A dynamic data structure (DDS) refers to an organization or collection of data in memory that has the *flexibility to grow or shrink in size*, enabling a programmer to control exactly how much memory is utilized.

Need of Dynamic DS

- In array, elements are stored in consecutive memory locations.
- To occupy the adjacent space, block of memory that is required for the array should be allocated before hand.
- Once memory is allocated, it cannot be extended any more. So that array is called the static data structure.
- Wastage of memory is more in arrays.
- Array has fixed size
- But, Linked list is a dynamic data structure, it is able to grow in size as needed.

Static and Dynamic DS

Static Data structure

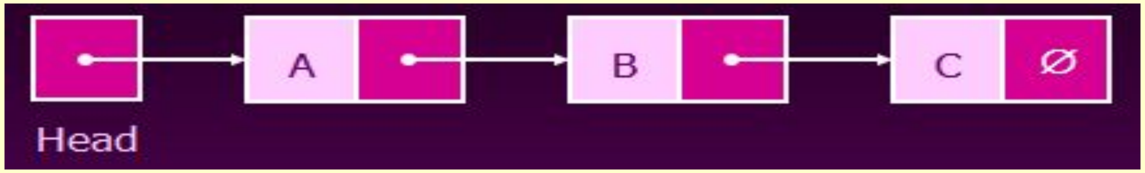
- Only alter the data present in the data structure at runtime
- Pre-define or calculate the size of the data structure

Dynamic Data Structure

- Both the data present in the data structure and the size of the data structure can be easily changed
- We can easily increase the size of the data structure at the runtime of the program execution.

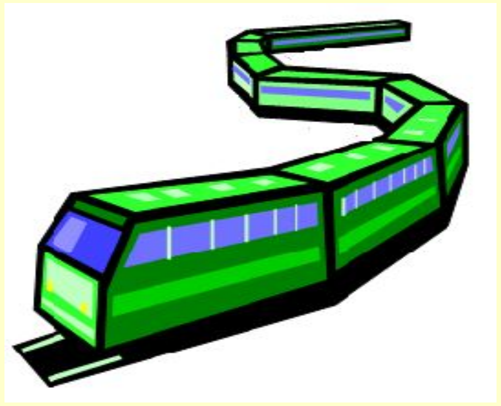
Linked List

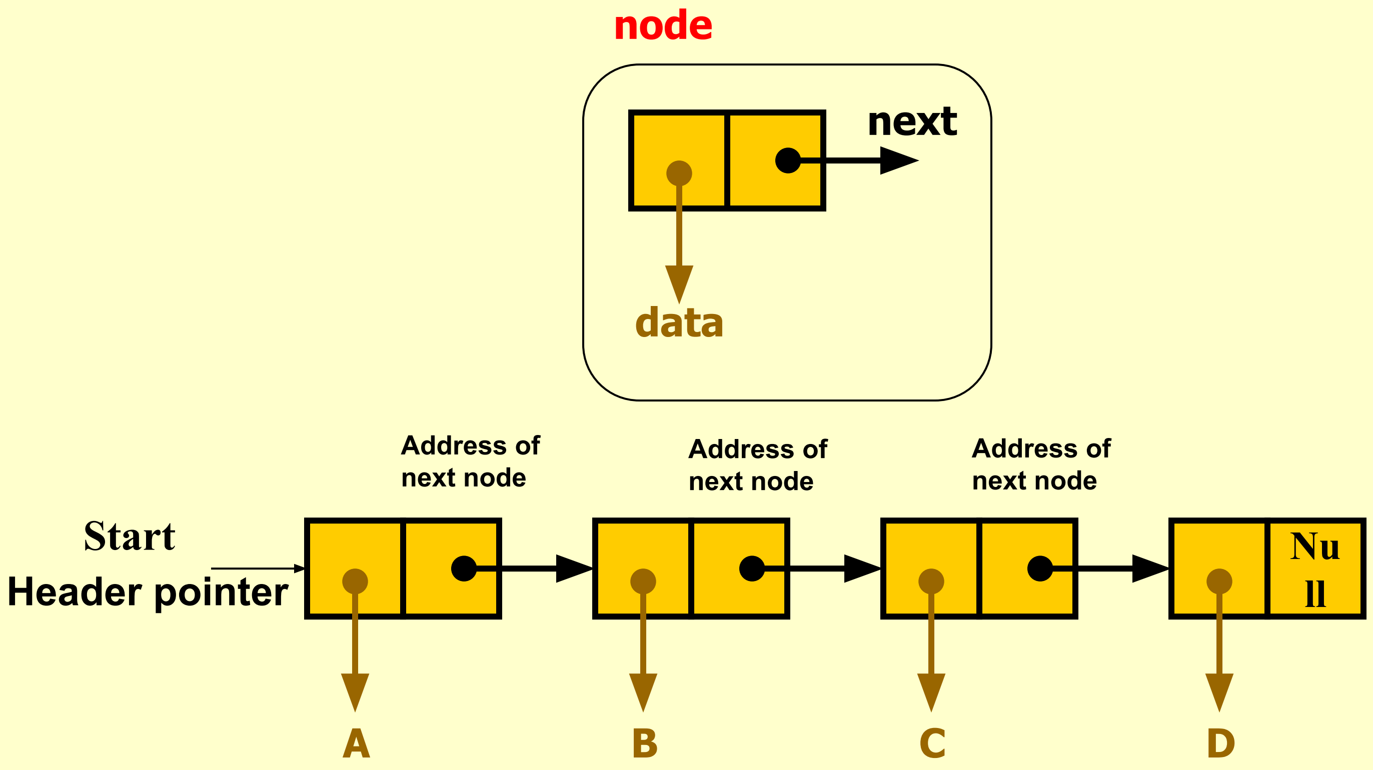
- It is a dynamic data structure
- A sequence composed of many Nodes, each Node is composed of two parts: Data, Link (or Pointer; point to another node).
- A node in a linked list is a structure that has at least two fields:
- Data: The data part holds the useful information, the data to be processed.
- Link: Used to chain the data together. It contains a pointer that identifies the next element in the list.



What is Linked List?

- A Linked list is a linear collection of homogeneous data elements, called **nodes,** where linear order is maintained by means of links or pointers.
- Each node has two parts:
  - The first part contains the data (information of the element) and
  - The second part contains the address of the next node (next /next pointer field) in the list.
- Data part of the next can be an integer, a character, a String or an object of any kind.




node
next
data
Address of next node
Address of next node
Address of next node
Start
Header pointer
Nu ll
A
B
C
D

Linked Lists

- **Linked list**
  - Linear collection of self-referential structures, called *nodes,* connected by pointer *links.*
  - Accessed via a pointer to the first node of the list.
  - Subsequent nodes are accessed via the next-pointer member stored in each node.
  - next pointer in the **last node is set to null** to mark the end of list.
  - Data stored dynamically – each node is created as necessary.
  - Length of a list can increase or decrease.
  - Becomes full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

Types of Linked lists

- **Singly Linked list**
  - Begins with a pointer to the first node
  - Terminates with a null pointer
  - Only traversed in one direction

- **Circular, singly linked list**
  - Pointer in the last node points back to the first node

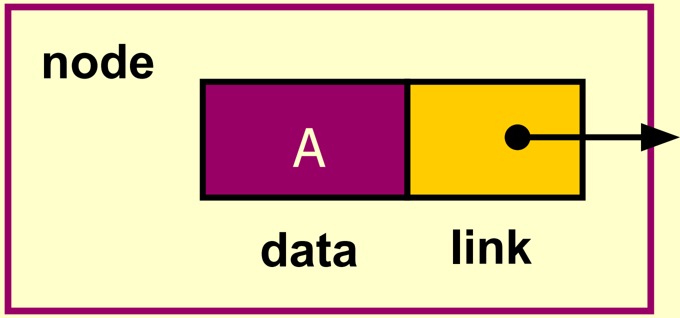
- **Doubly linked list**
  - Two “start pointers”- first element and last element
  - Each node has a forward pointer and a backward pointer
  - Allows traversals both forwards and backwards

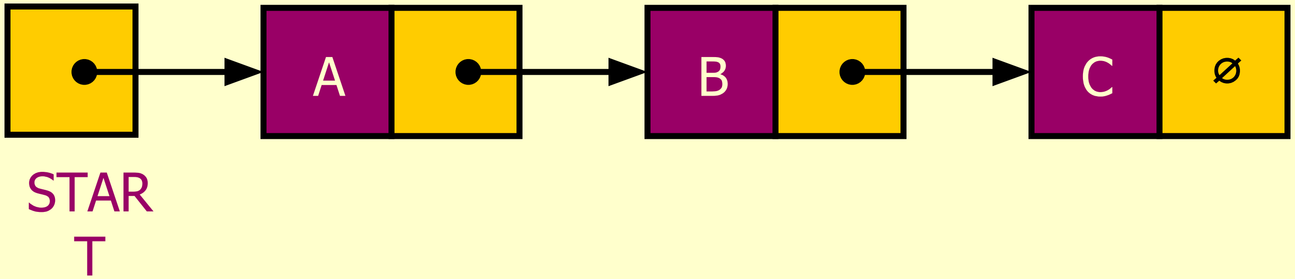
- **Circular, doubly linked list**
  - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

Single Linked List

- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data i.e info field (any type)
  - Pointer to the next node in the list that contains address of next node.
- *START*: pointer to the first node

  Or in some books its Head
- The last node points to NULL





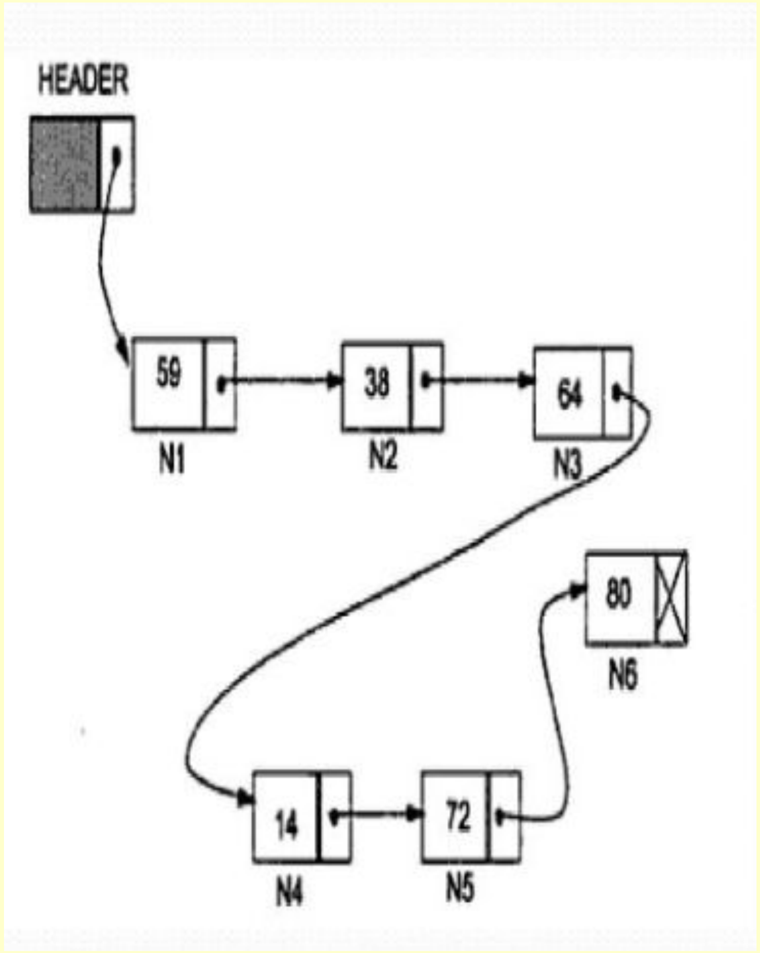
Representation of Linked List

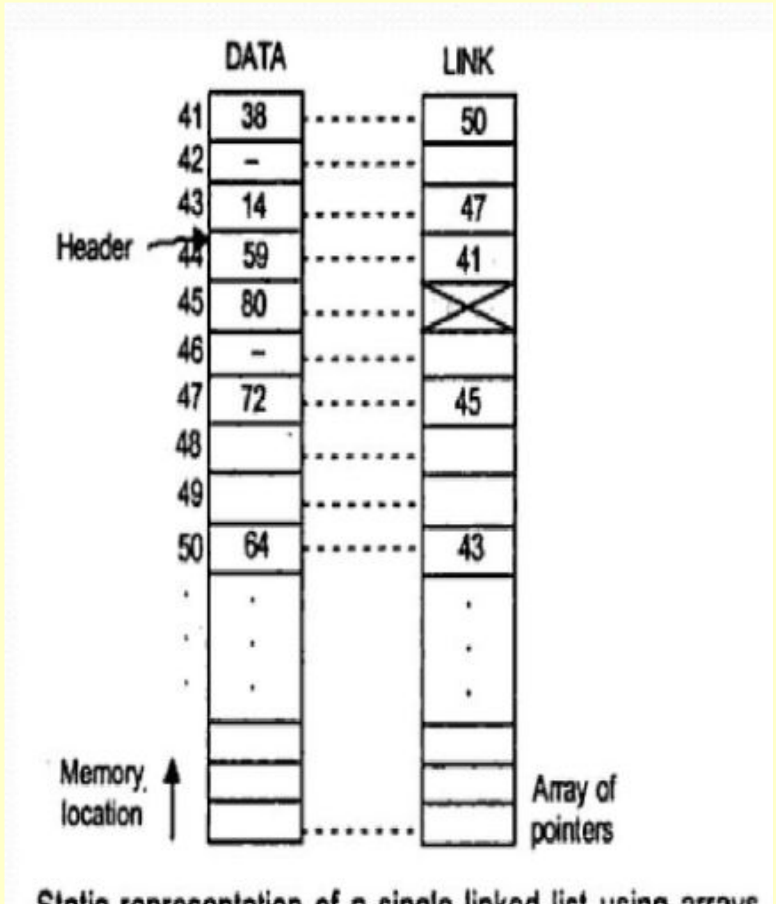
- There are two ways to represent linked list in the memory

1. Static representation using array
2. Dynamic representation using free pool of storage

Static representation of Linked List

- Static representation maintains two array.
- One for data (info) and another for link (next)





Static representation of a single linked list using arrays.

Dynamic representation of Linked List

- In this method, there is a memory bank (collection of free memory) and a memory manager (program)
- During the creation of LL, whenever a node is required the request is placed to the memory manager which will then search the memory bank for the block requested and if found grants the required block to the caller.
- Another program is garbage collector returns the unused memory to the memory bank
- This is called dynamic memory allocation
- Dynamic representation of LL using dynamic memory allocation scheme

Creating Node

- Create a class Node which has two attributes: data and next. Next is a pointer to the next node
- We can create a node using either structure or class

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

- Allocating memory to node
- struct node *newNode = (struct node*)malloc(sizeof(struct node));

Allocating memory to Node & access elements

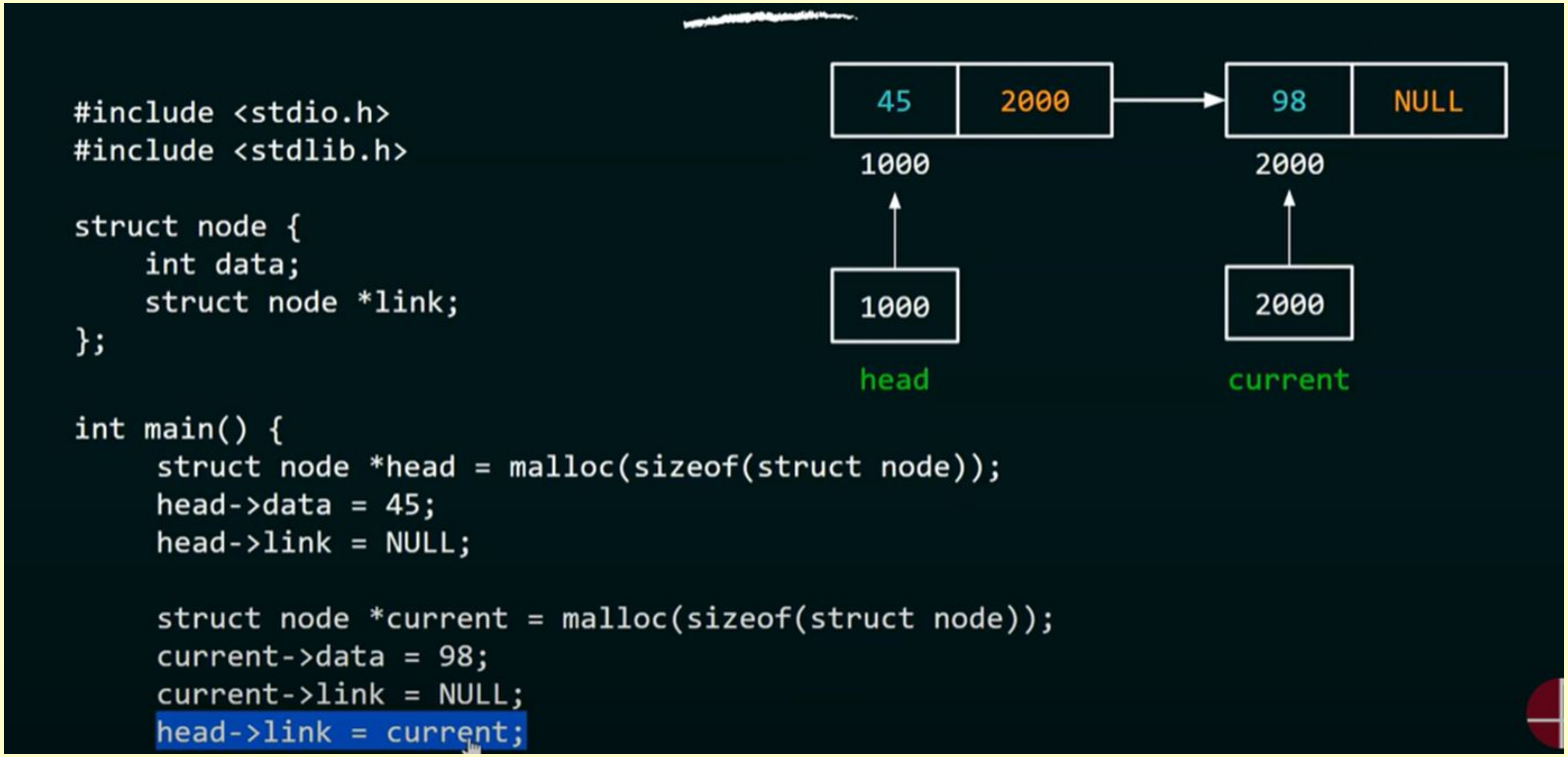
- Dynamically allocate memory to node

```
struct node *newNode = (struct node*)malloc(sizeof(struct
    node));
```

- Accessing elements of node

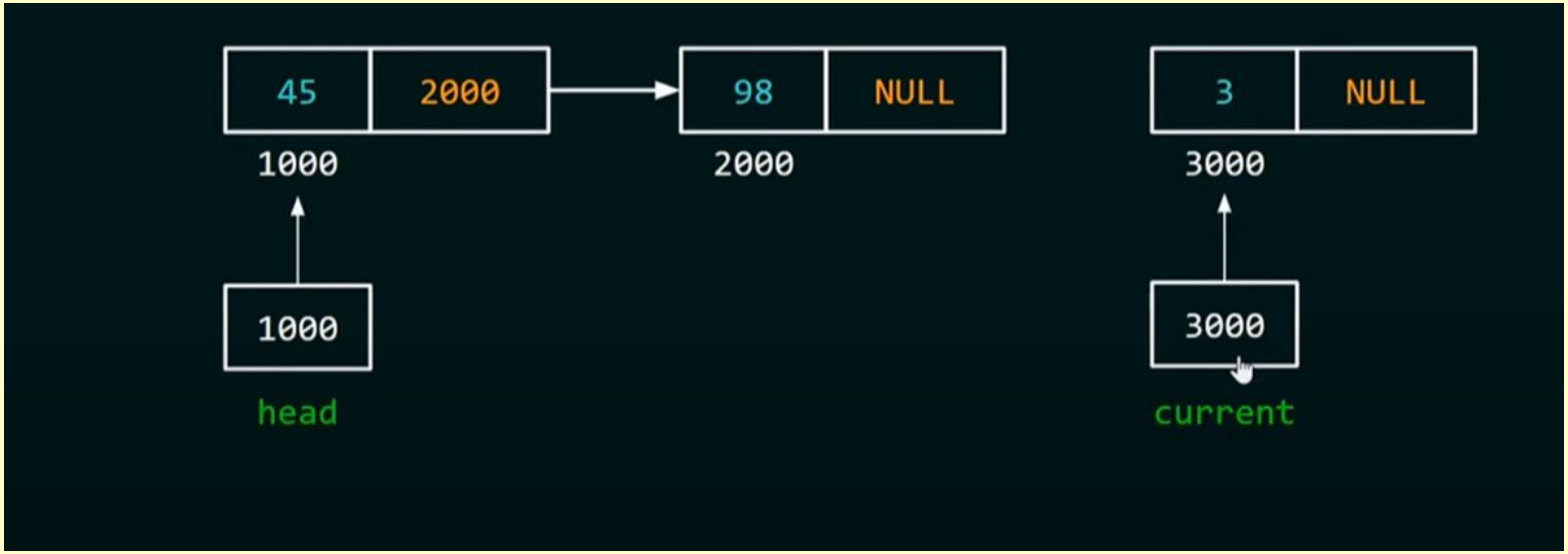
```
newNode->data = data;
newNode->link = NULL;
```

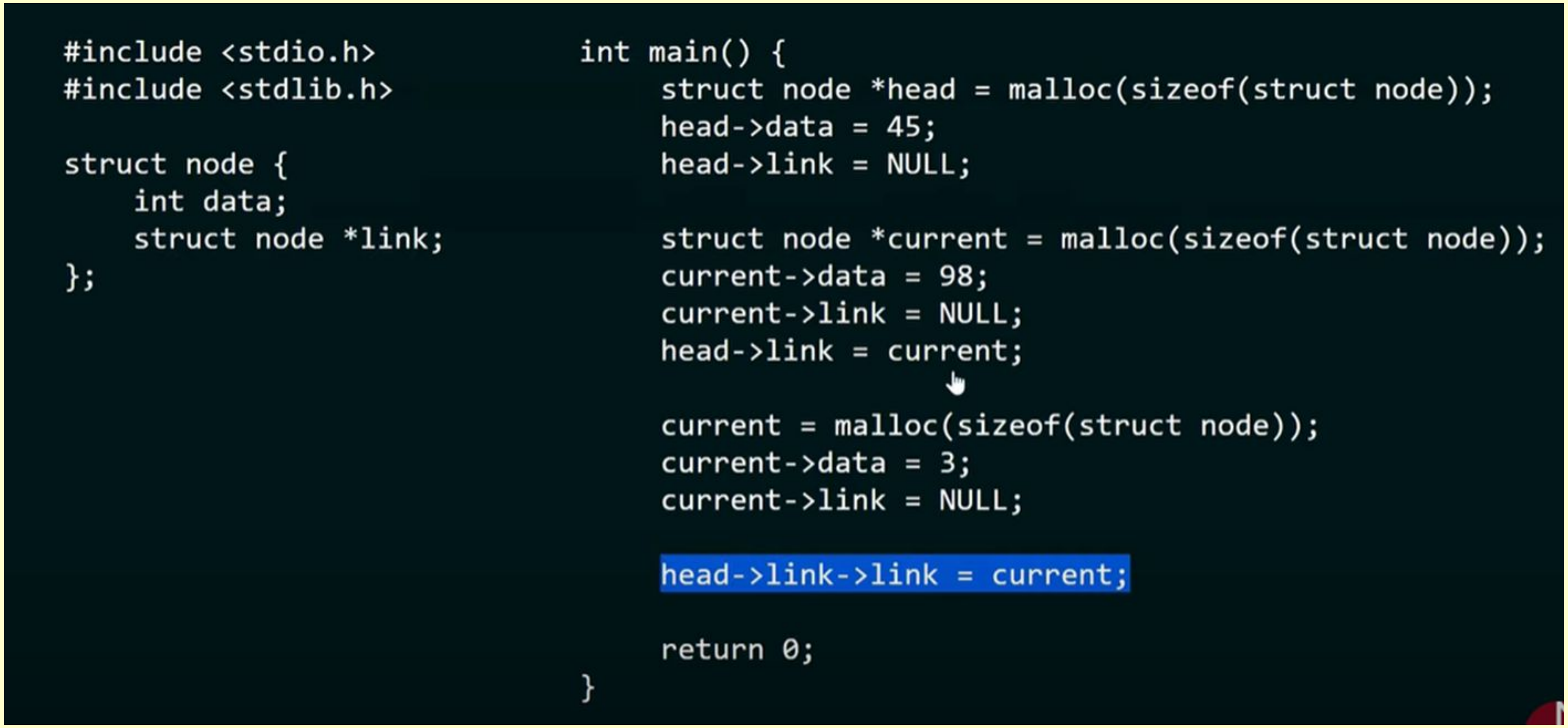
SINGLE LINKED LIST



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

struct node {
    int data;
    struct node *link;
};

int main() {
     struct node *head = malloc(sizeof(struct node));
     head->data = 45;
     head->link = NULL;

     struct node *current = malloc(sizeof(struct node));
     current->data = 98;
     current->link = NULL;
     head->link = current;
```

CREATE LINKED LIST FOR BELOW NODES





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

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

```
int main() {
    struct node *head = malloc(sizeof(struct node));
    head->data = 45;
    head->link = NULL;

    struct node *current = malloc(sizeof(struct node));
    current->data = 98;
    current->link = NULL;
    head->link = current;

    current = malloc(sizeof(struct node));
    current->data = 3;
    current->link = NULL;

    head->link->link = current;

    return 0;
}
```

Singly Linked list operations

Insertion:

- Insertion of a node at the front
- Insertion of a node at any position in the list
- Insertion of a node at the end

Deletion:

- Deletion at front
- Deletion at any position
- Deletion at end

Display:

- Displaying/Traversing the elements of a list

Searching

•Insertion & Deletion in Linked List

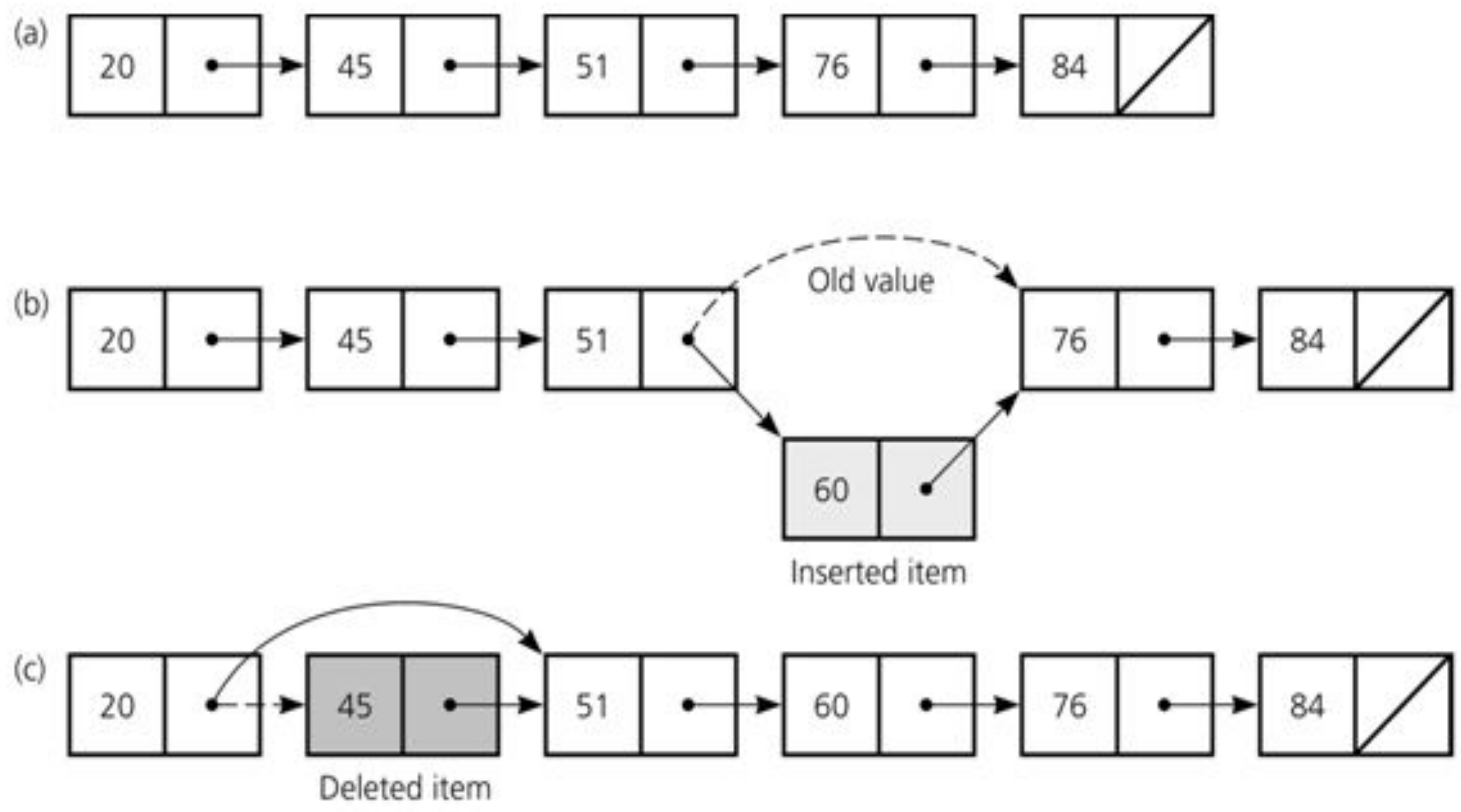


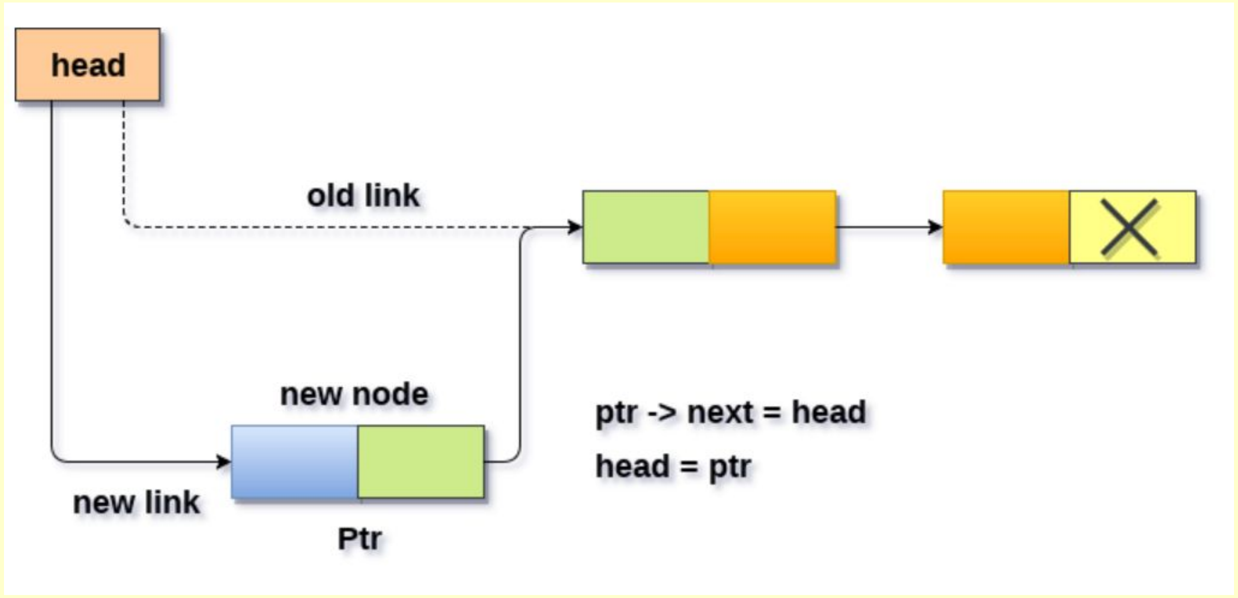
Figure a) A linked list of integers; b) insertion; c) deletion

Inserting at Beginning

```
void insert_beg(start,item)
{
 struct node *nitem;
nitem = (struct node *) malloc(sizeof(struct node));
 nitem->data=item;
 nitem->link=start;
 start=nitem;
}
```

- **Start is also written as head**
- **Data is also written as info**
- **Link is also written as next**

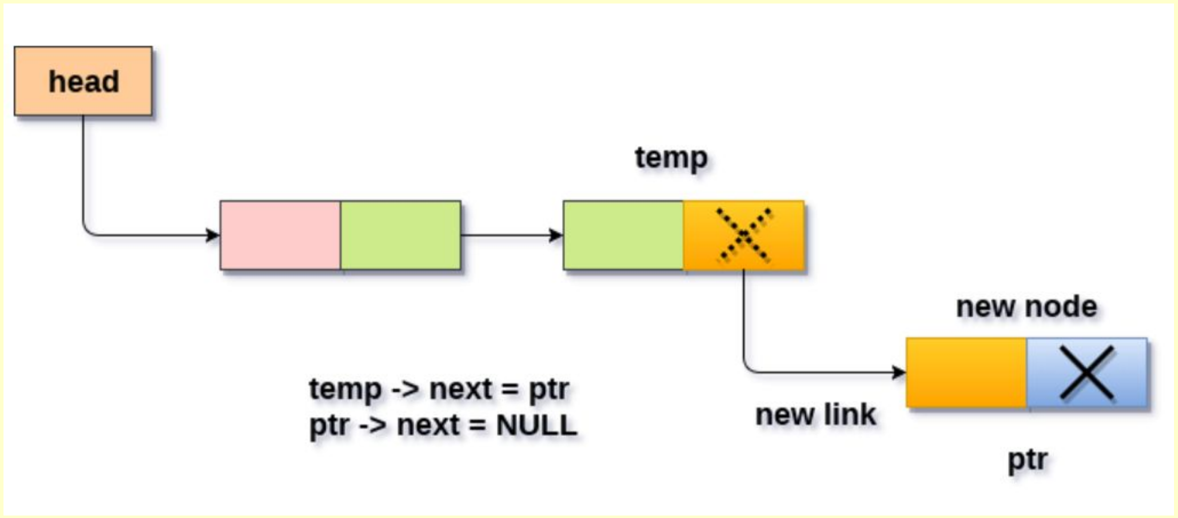
-



Inserting at End

```
void insert_end(start,item)
{
 struct node *ptr;
 struct node *nitem = (struct node *)malloc(sizeof(struct node));
 nitem->data=item;
 nitem->link = NULL;
 if (start==NULL)
  start = nitem;
 else
 {
    ptr = start;
    while(ptr->link !=NULL)
      ptr = ptr->link;
    ptr->link=nitem;
 }
}
```

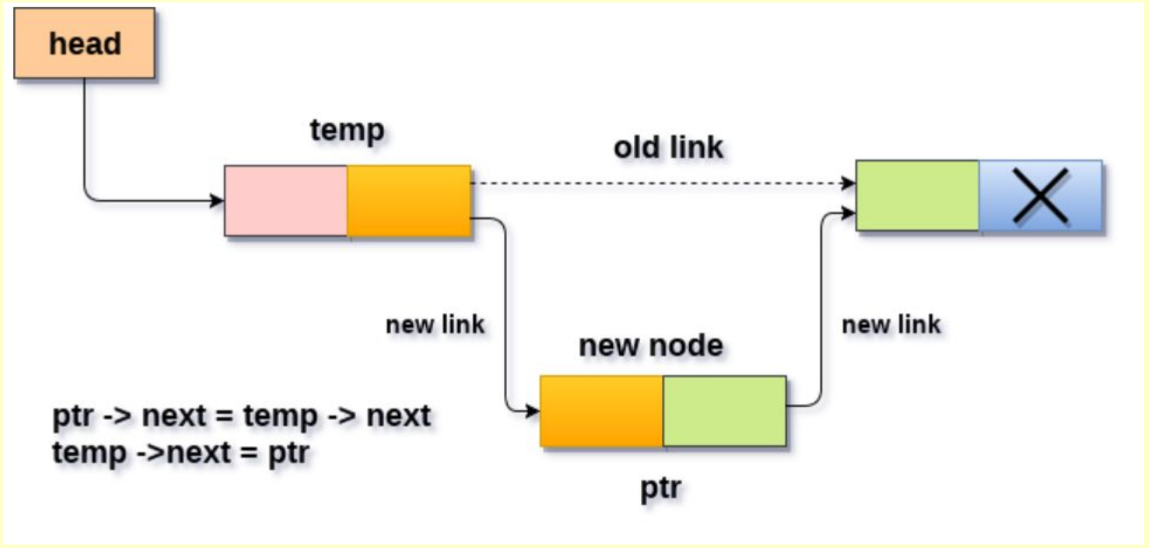
- **Start is also written as head**
- **Data is also written as info**
- **Link is also written as next**



Inserting after a given node (or position)

- **Start is also written as head**
- **Data is also written as info**
- **Link is also written as next**

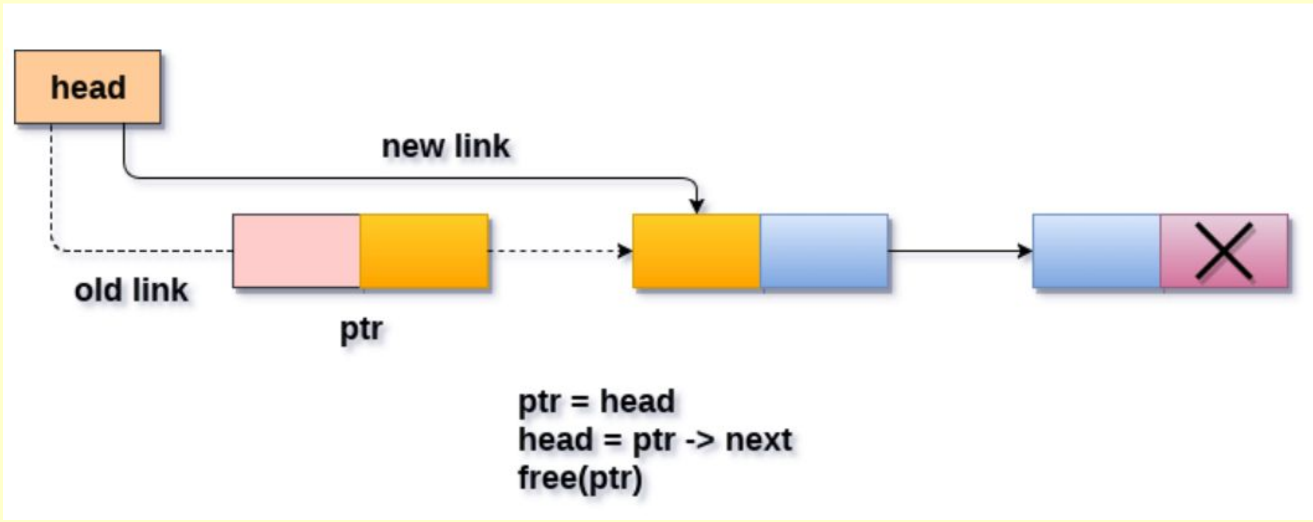
```
void insert_posi(start,item,posi)
{
 struct node *ptr;
 struct node *nitem = (struct node *) malloc(sizeof(struct node));
 nitem->data=item;
 if(posi==1)
   {
    nitem->link=start;
    start=nitem;
   }
 else
  {
   ptr=start;
   for(i=2;i<=posi-1;i++)
       ptr=ptr->link;
   nitem->link=ptr->link;
   ptr->link=nitem;
  }
}
```



Deletion at Beginning

```
void delete_beg(start)
{
 if(start==NULL)
  {
   printf("list is empty");
  }
 else
  {
   struct node *nitem;
   nitem = start;
   start = start->link;
   free(nitem);
  }
}
```

- **Start is also written as head**
- **Data is also written as info**
- **Link is also written as next**

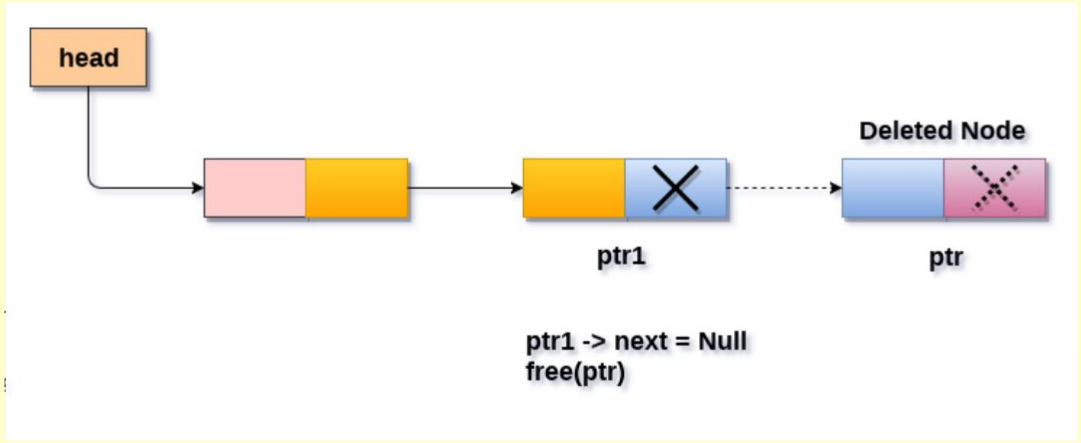


```
ptr = head
head = ptr -> next
free(ptr)
```

Deletion at End

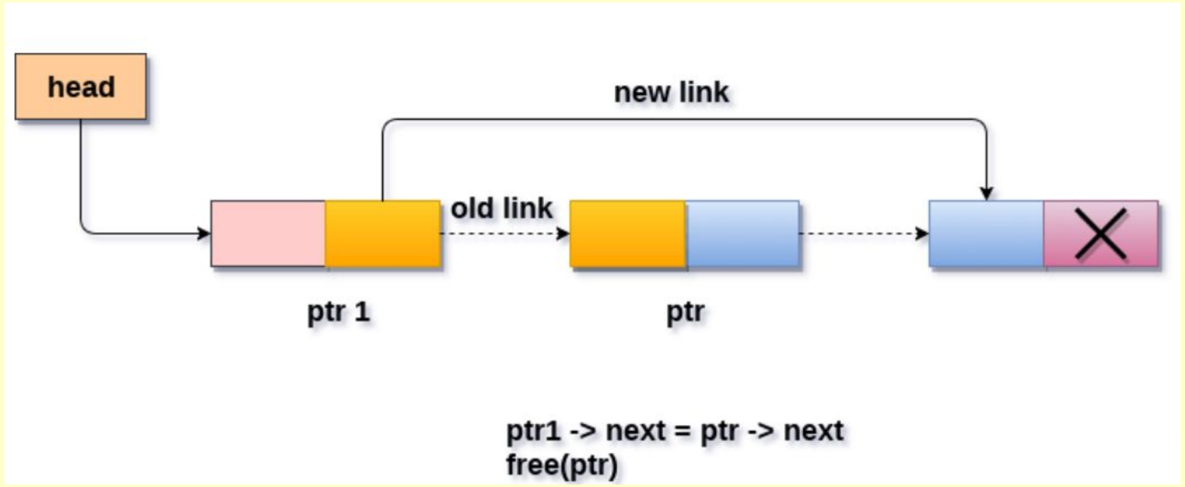
```
void delete_end(start)
{
  struct node *locp,*loc;
  if(start==NULL)
     printf("list is empty");
  else if(start->link==NULL)
    {  loc = start; start = NULL; free(loc); }
  else{
   locp = start;
   loc = start->link;
   while(loc->link !=NULL)
   {   locp=loc; loc = loc->link; }
   locp->link = NULL;
   free(loc);
  }
}
```

- **Start is also written as head**
- **Data is also written as info**
- **Link is also written as next**



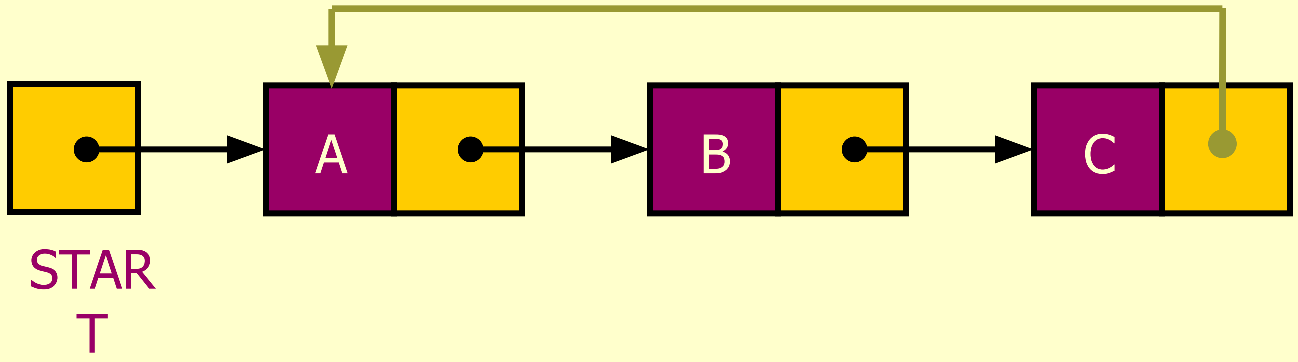
Deletion the node (LOC) following a given node (LOCP) or at position

```
void delete_posi(start,posi)
{
 struct node *loc, *locp;
 if(start==NULL)
   printf("list is empty");
else if(posi==1)
  { loc=start; start=start->link; free(loc);}
 else
  {
   locp=start;
   loc=start->link;
   for(int i=3;i<=posi;i++)
     { locp=loc;  loc=loc->link; }
  locp->link=loc->link;
  free(loc);
 }
}
```



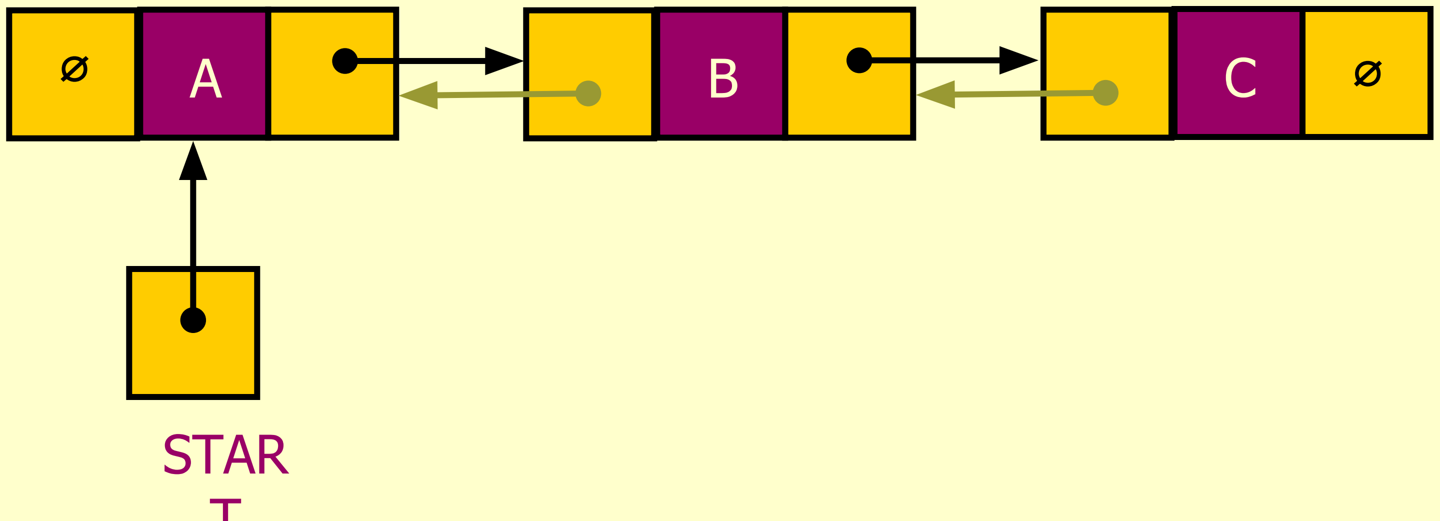
Variations of Linked Lists

- *Circular linked lists*
  - The last node points to the first node of the list

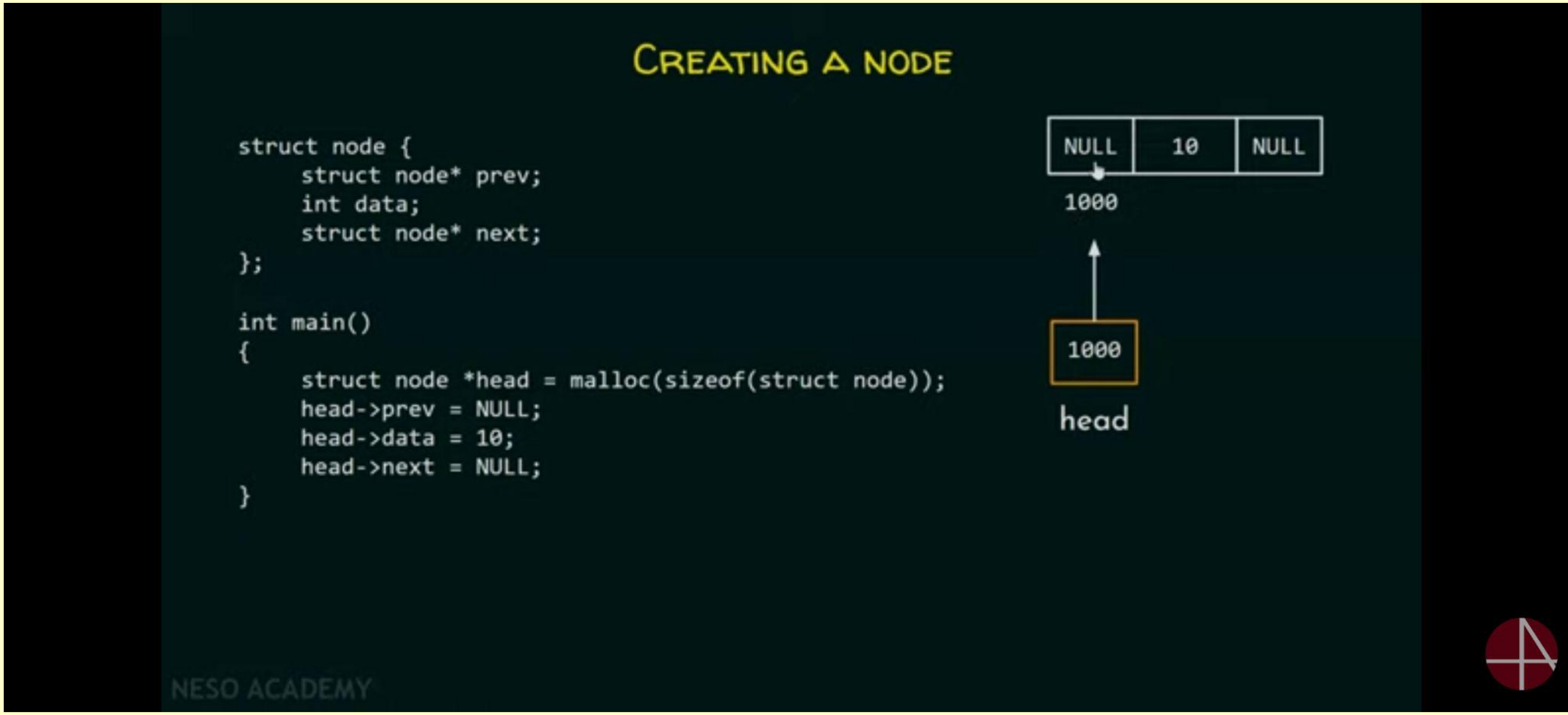


Variations of Linked Lists

- *Doubly linked lists*
  - Each node points to not only successor but the predecessor
  - There are two NULL: at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards
  - Each node has three fields : DATA, NLINK, PLINK

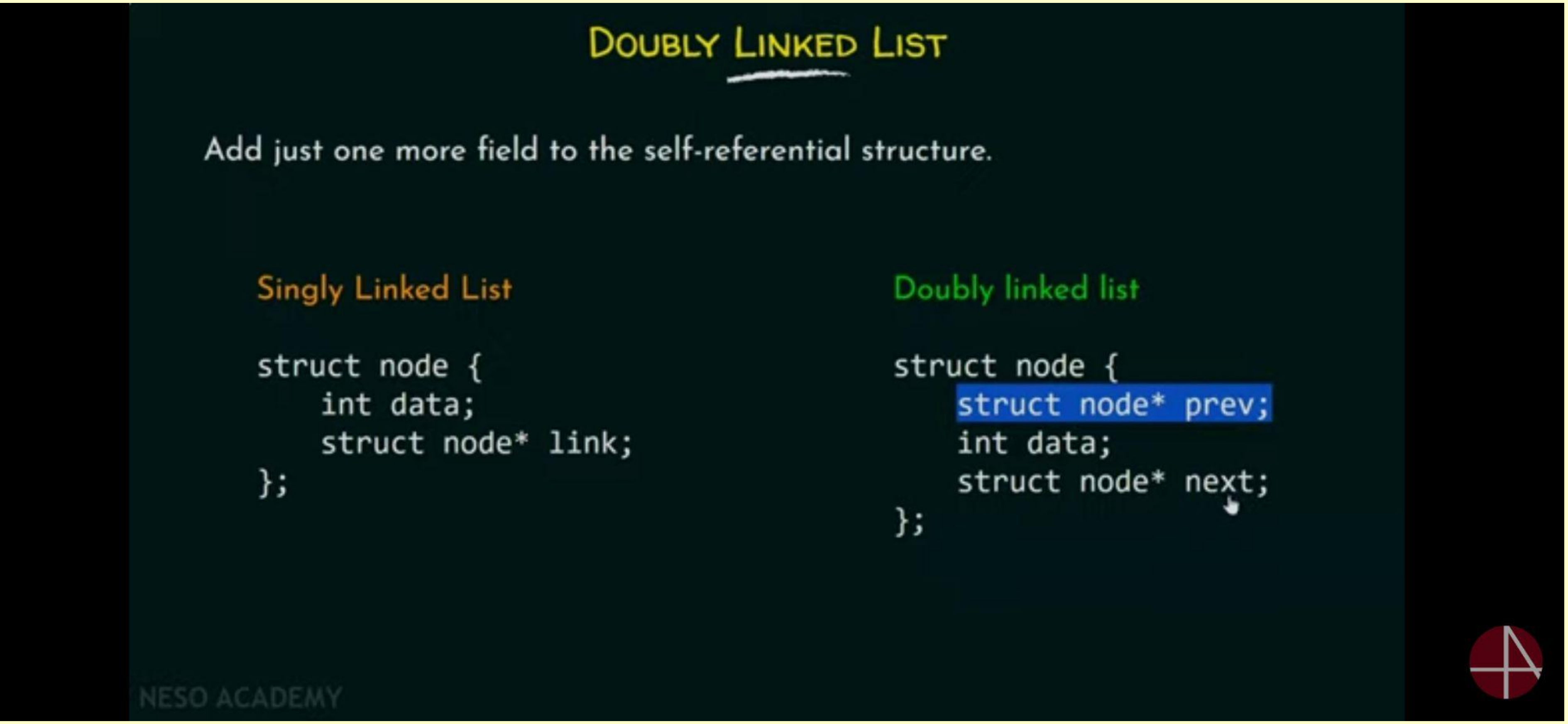


CREATING A NODE IN DLL



Creating a node

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

int main()
{
     struct node *head = malloc(sizeof(struct node));
     head->prev = NULL;
     head->data = 10;
     head->next = NULL;
}
```



Doubly Linked List

Add just one more field to the self-referential structure.

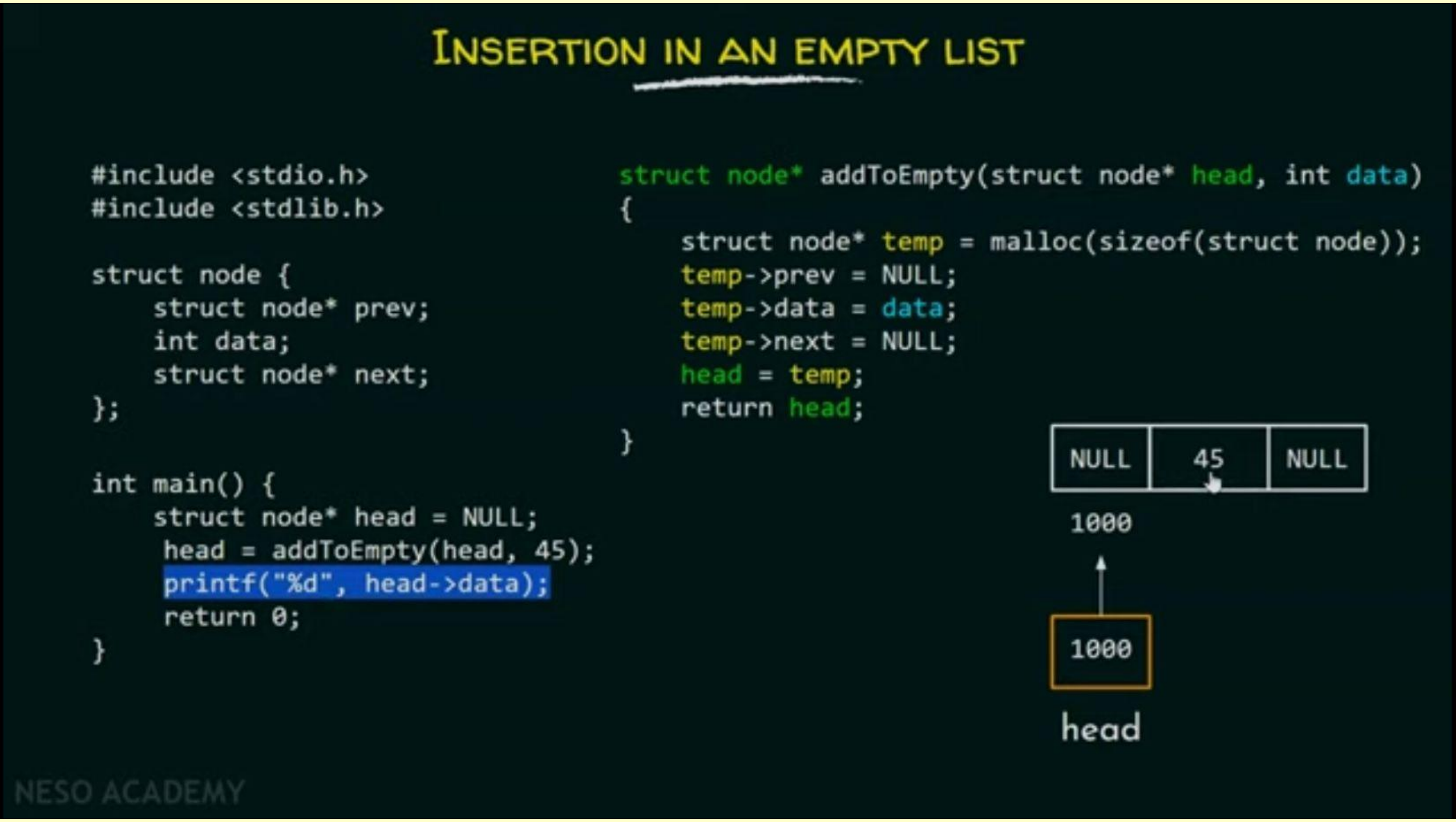
Singly Linked List

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

Doubly linked list

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

INSERTION IN AN EMPTY LIST



INSERTION IN AN EMPTY LIST

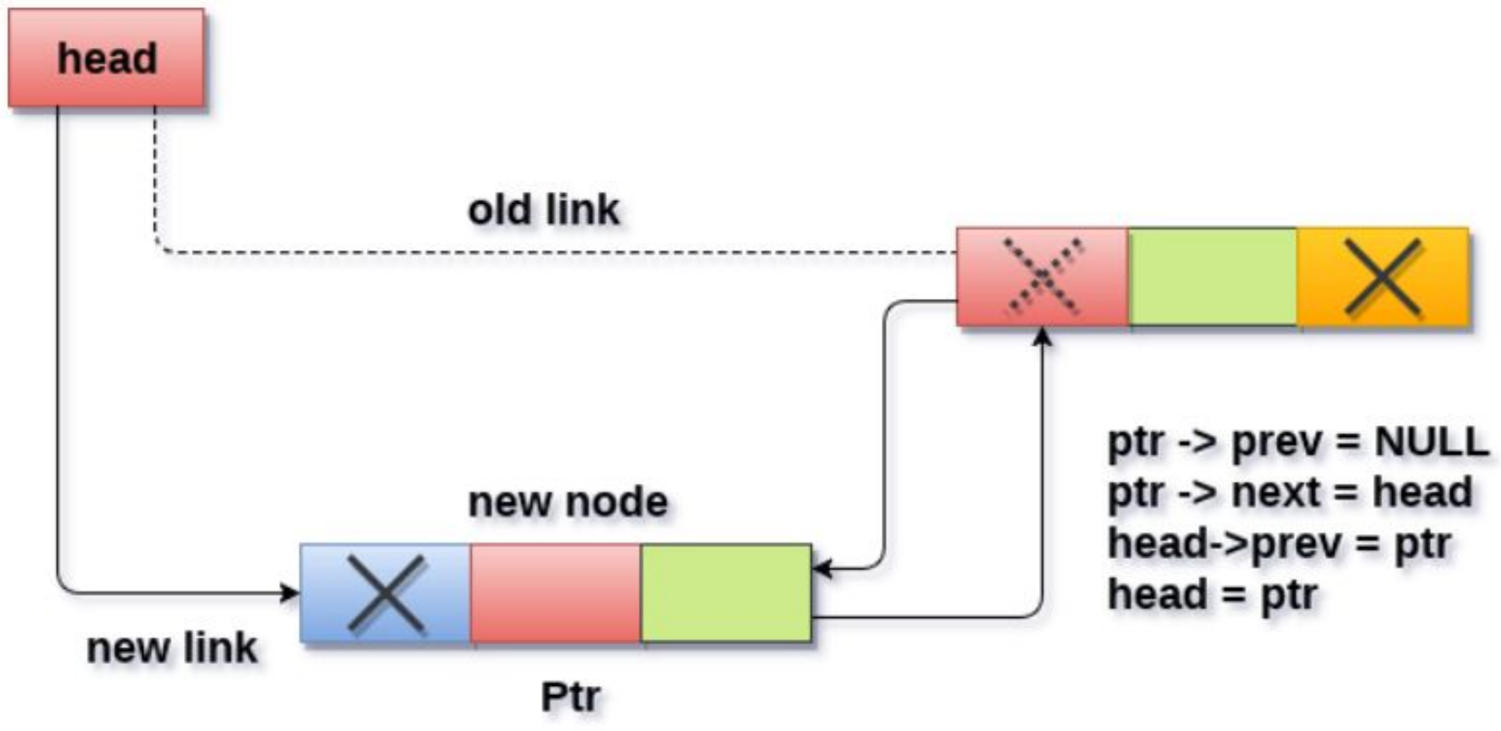
```
#include <stdio.h>
#include <stdlib.h>

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

int main() {
    struct node* head = NULL;
     head = addToEmpty(head, 45);
     printf("%d", head->data);
     return 0;
}
```

```
struct node* addToEmpty(struct node* head, int data)
{
    struct node* temp = malloc(sizeof(struct node));
    temp->prev = NULL;
    temp->data = data;
    temp->next = NULL;
    head = temp;
    return head;
}
```

INSERTION AT THE BEGINNING

```
○ Step 1: IF ptr = NULL
○    Write OVERFLOW
     Go to Step 9
     [END OF IF]Step 2: SET NEW_NODE = ptr
○ Step 3: SET ptr = ptr -> NEXT
○ Step 4: SET NEW_NODE -> DATA = VAL
○ Step 5: SET NEW_NODE -> PREV = NULL
○ Step 6: SET NEW_NODE -> NEXT = START
○ Step 7: SET head -> PREV = NEW_NODE
○ Step 8: SET head = NEW_NODE
○ Step 9: EXIT
```

INSERTION AT THE BEGINNING



```
ptr -> prev = NULL
ptr -> next = head
head->prev = ptr
head = ptr
```

INSERTION AT THE END

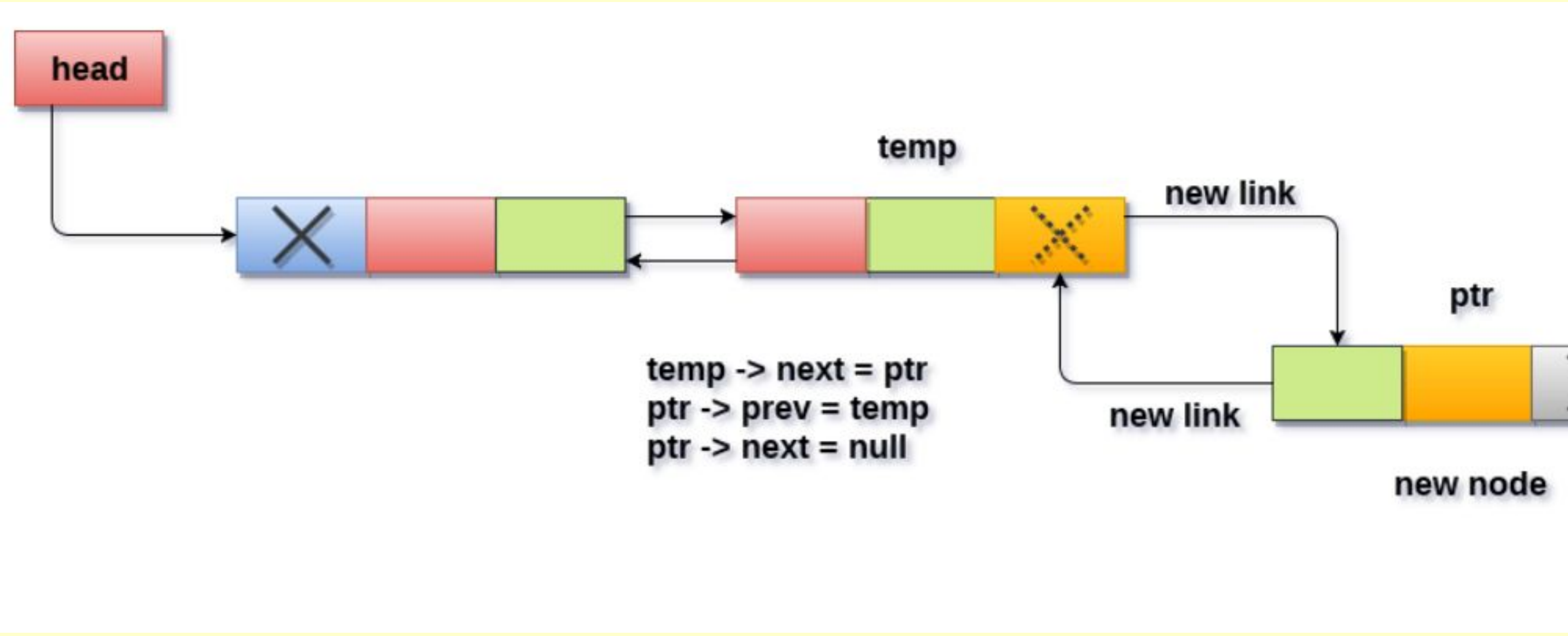
Algorithm

- **Step 1:** IF PTR = NULL
- Write OVERFLOW

  Go to Step 11

  [END OF IF]**Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET NEW_NODE -> NEXT = NULL
- **Step 6:** SET TEMP = START
- **Step 7:** Repeat Step 8 while TEMP -> NEXT != NULL
- **Step 8:** SET TEMP = TEMP -> NEXT
- [END OF LOOP]**Step 9:** SET TEMP -> NEXT = NEW_NODE
- **Step 10C:** SET NEW_NODE -> PREV = TEMP
- **Step 11:** EXIT

INSERTION AT THE END DLL



temp -> next = ptr
ptr -> prev = temp
ptr -> next = null

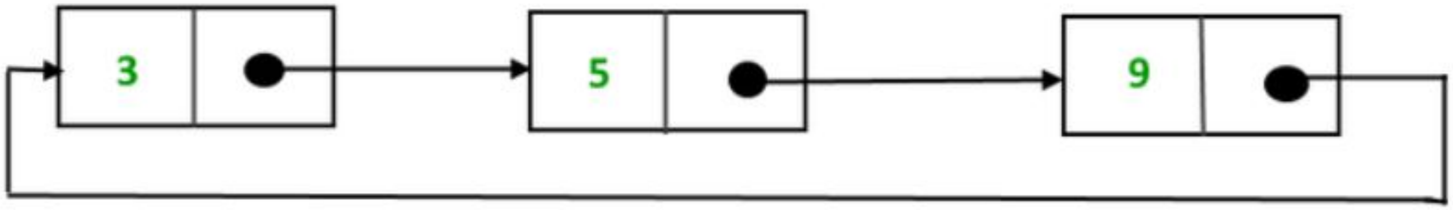
CIRCULAR LINKED LIST

The **circular linked list** is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.

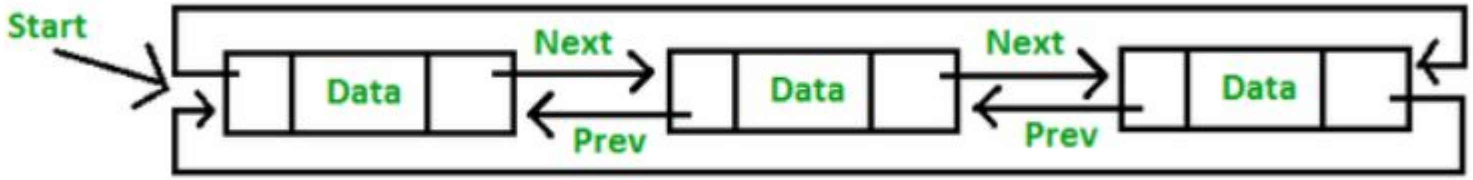
There are generally two types of circular linked lists:

- **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 traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.

- **Circular Doubly linked list:** Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.



Representation of Circular singly linked list



Representation of circular doubly linked list

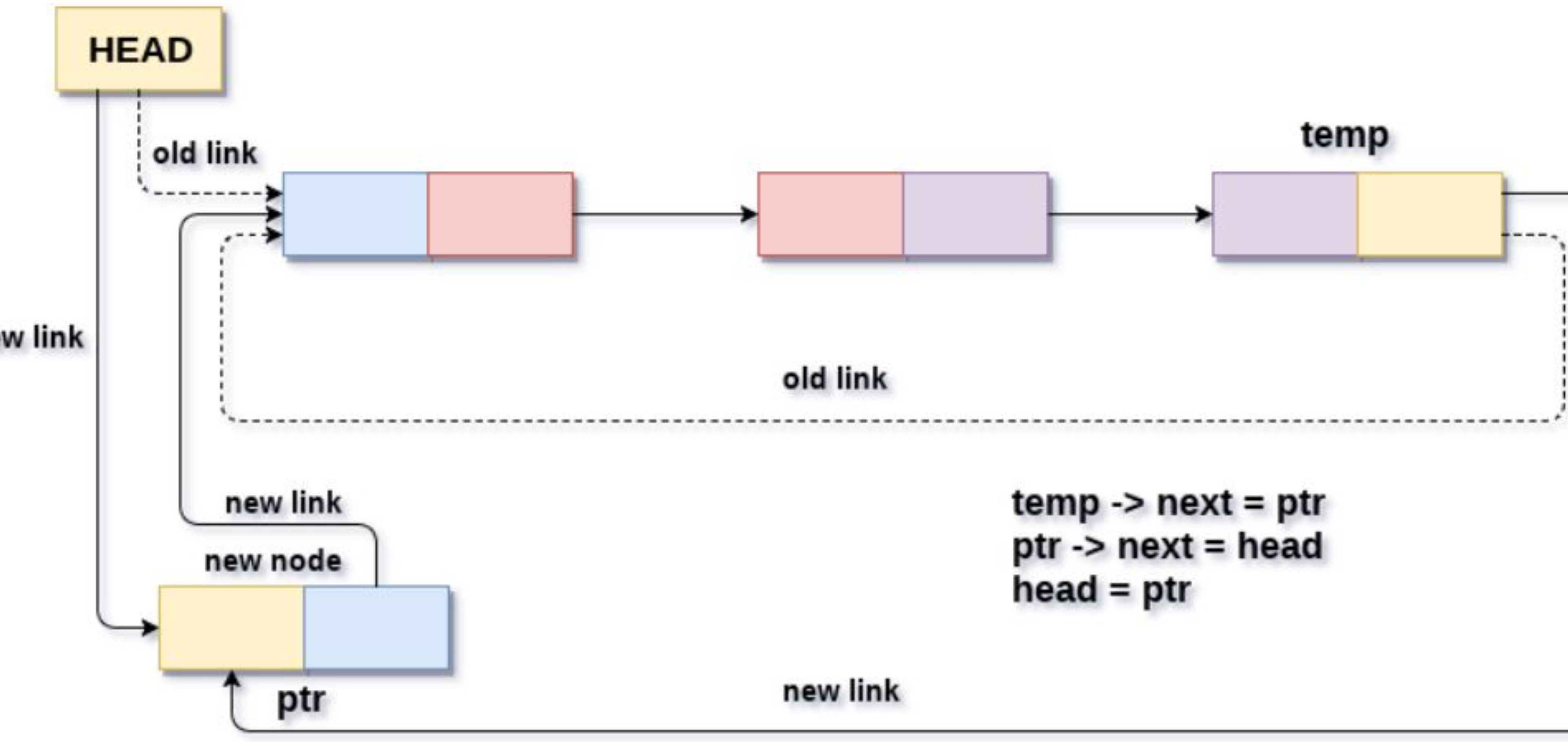
INSERTION AT THE BEGINNING CLL

Algorithm

- **Step 1:** IF PTR = NULL
- Write OVERFLOW

  Go to Step 11

  [END OF IF]**Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET TEMP = HEAD
- **Step 6:** Repeat Step 8 while TEMP -> NEXT != HEAD
- **Step 7:** SET TEMP = TEMP -> NEXT
- [END OF LOOP]**Step 8:** SET NEW_NODE -> NEXT = HEAD
- **Step 9:** SET TEMP → NEXT = NEW_NODE
- **Step 10:** SET HEAD = NEW_NODE
- **Step 11:** EXIT



temp -> next = ptr
ptr -> next = head
head = ptr

SPARSE MATRIX

Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

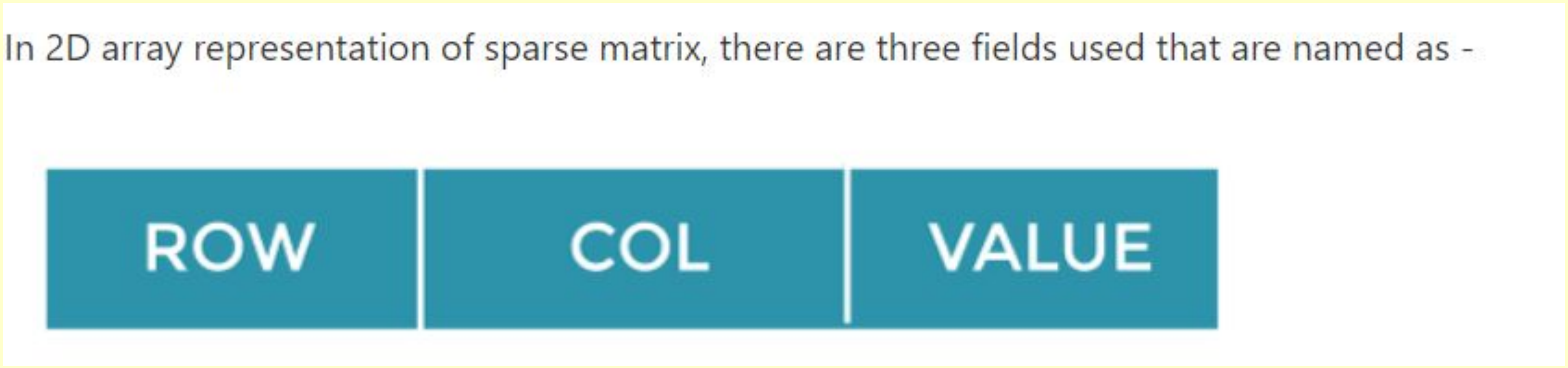
Why is a sparse matrix required if we can use the simple matrix to store elements?

There are the following benefits of using the sparse matrix -

Storage - We know that a sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements. It evaluates only the non-zero elements.

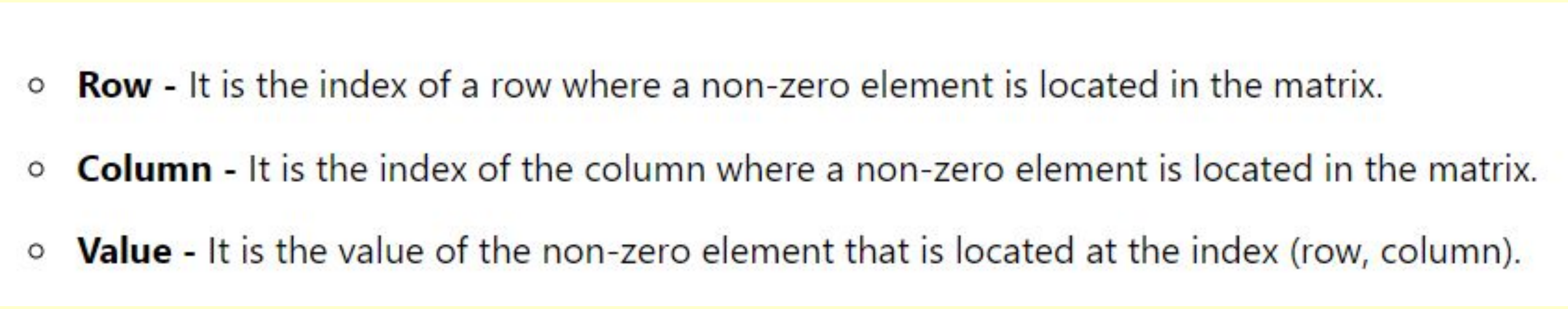
Computing time: In the case of searching in sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

ARRAY REPRESENTATION OF SPARSE MATRIX



In 2D array representation of sparse matrix, there are three fields used that are named as -

ROW	COL	VALUE



- **Row -** It is the index of a row where a non-zero element is located in the matrix.
- **Column -** It is the index of the column where a non-zero element is located in the matrix.
- **Value -** It is the value of the non-zero element that is located at the index (row, column).

Let's understand the array representation of sparse matrix with the help of the example given below -

Consider the sparse matrix -

$$\text{Sparse Matrix} \longrightarrow \begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 4 & 0 & 5 \\ 1 & 0 & 0 & 3 & 6 \\ 2 & 0 & 0 & 2 & 0 \\ 3 & 2 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 \\ \hline \end{array}$$

In the below structure, first column represents the rows, the second column represents the columns, and the third column represents the non-zero value. The first row of the table represents the triplets. The first triplet represents that the value 4 is stored at 0th row and 1st column. Similarly, the second triplet represents that the value 5 is stored at the 0th row and 3rd column. In a similar manner, all triplets represent the stored location of the non-zero elements in the matrix.

The size of the table depends upon the total number of non-zero elements in the given sparse matrix. Above table occupies 8x3 = 24 memory space which is more than the space occupied by the sparse matrix.

So, what's the benefit of using the sparse matrix? Consider the case if the matrix is 8*8 and there are only 8 non-zero elements in the matrix, then the space occupied by the sparse matrix would be 8*8 = 64, whereas the space occupied by the table represented using triplets would be 8*3 = 24.

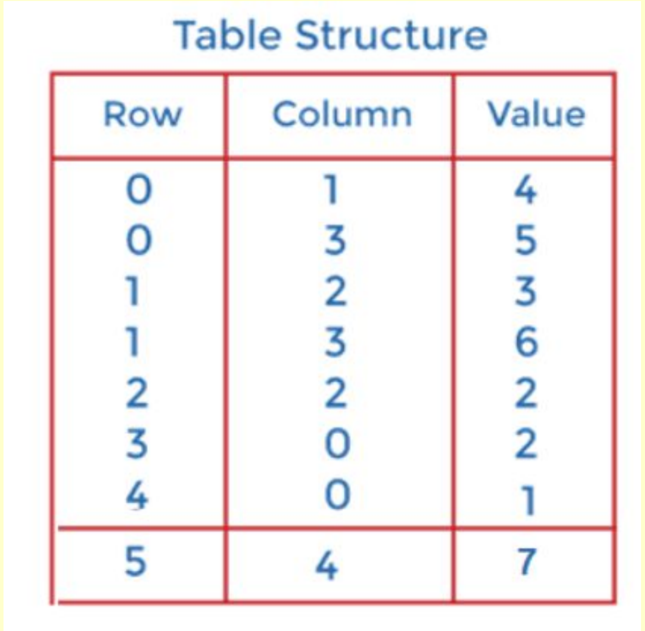


Table Structure

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1
5	4	7

```c
#include <stdio.h>
int main()
{
    // Sparse matrix having size 4*5
    int sparse_matrix[4][5] =
    {
        {0 , 0 , 6 , 0 , 9 },
        {0 , 0 , 4 , 6 , 0 },
        {0 , 0 , 0 , 0 , 0 },
        {0 , 1 , 2 , 0 , 0 }
    };
   // size of matrix
    int size = 0;
    for(int i=0; i<4; i++)
    {
        for(int j=0; j<5; j++)
        {
            if(sparse_matrix[i][j]!=0)
            {
                size++;
```

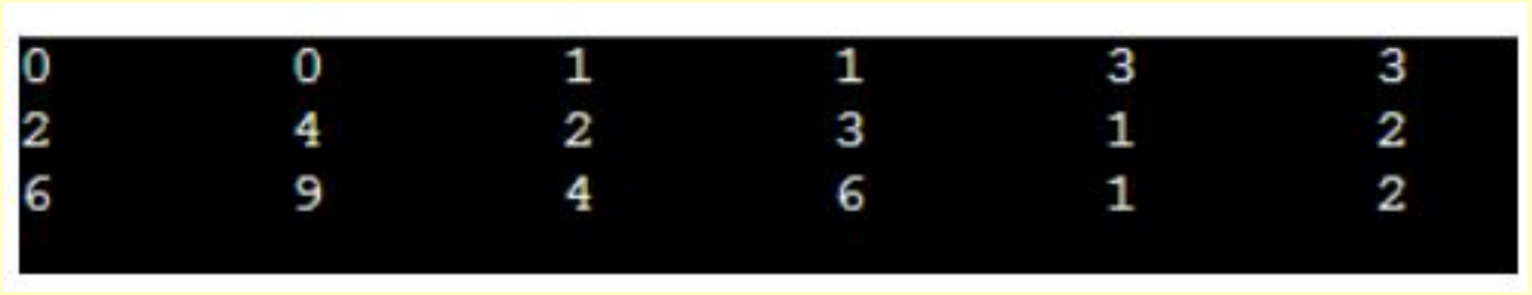
```
// Defining final matrix
    int matrix[3][size];
     int k=0;
   // Computing final matrix
    for(int i=0; i<4; i++)
    {
        for(int j=0; j<5; j++)
        {
            if(sparse_matrix[i][j]!=0)
            {
                matrix[0][k] = i;
                matrix[1][k] = j;
                matrix[2][k] = sparse_matrix[i][j];
                k++;
            }
      }
    }
   // Displaying the final matrix
    for(int i=0 ;i<3; i++)
    {
        for(int j=0; j<size; j++)
        {
            printf("%d ", matrix[i][j]);
            printf("\t");
```

OUTPUT

Output

In the output, first row of the table represent the row location of the value, second row represents the column location of the value, and the third represents the value itself.

In the below screenshot, the first column with values 0, 2, and 6 represents the value 6 stored at the 0th row and 2nd column.



```
0        0        1        1        3        3
2        4        2        3        1        2
6        9        4        6        1        2
```