

Introduction to Linked List and its Operations

Class:- Ba/B.Sc.(CS)/B.Sc.(ECO)-IV Sem

Subject:- Data Structure and Programming Using C++

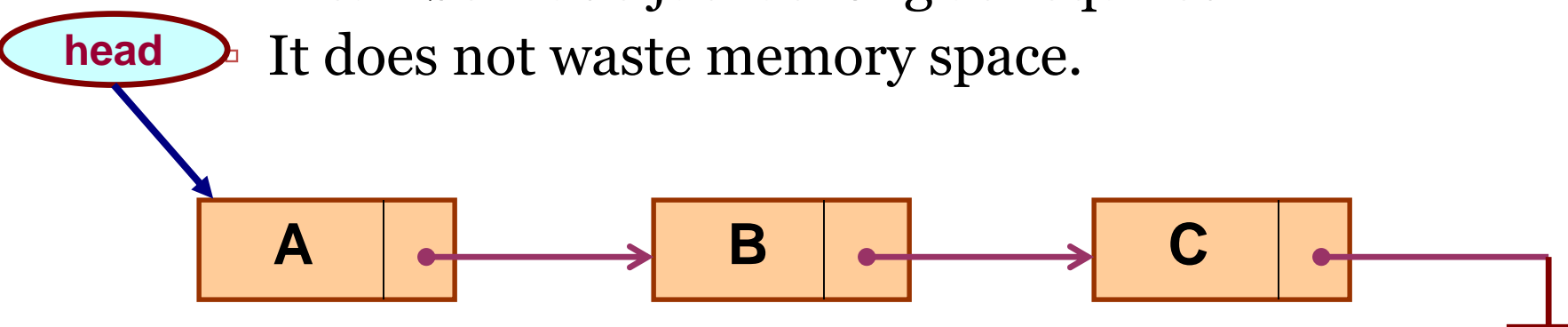
Submitted By:-

Ms. Sakshi kapoor

PG Department of Computer Science and IT

What is a Linked List?

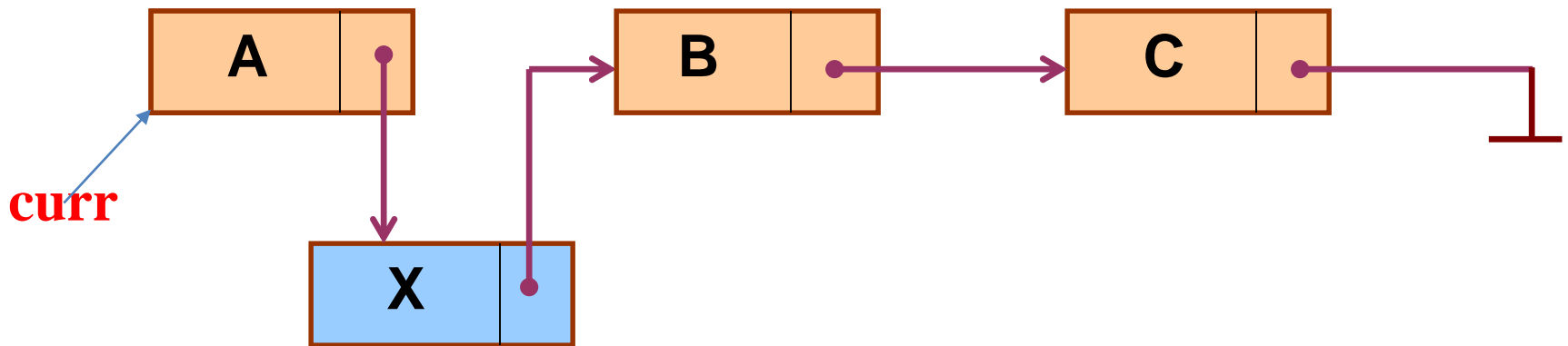
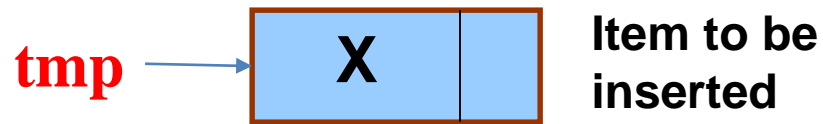
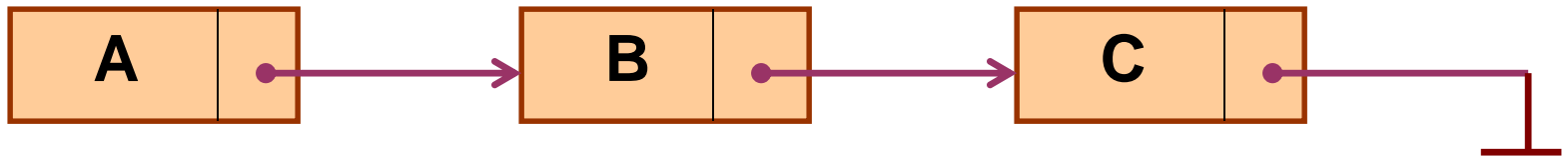
- A linked list is a data structure which can change during execution.
 - Successive elements are connected by pointers.
 - Last element points to NULL.
 - It can grow or shrink in size during execution of a program.
 - It can be made just as long as required.
 - It does not waste memory space.



How to Use Linked List

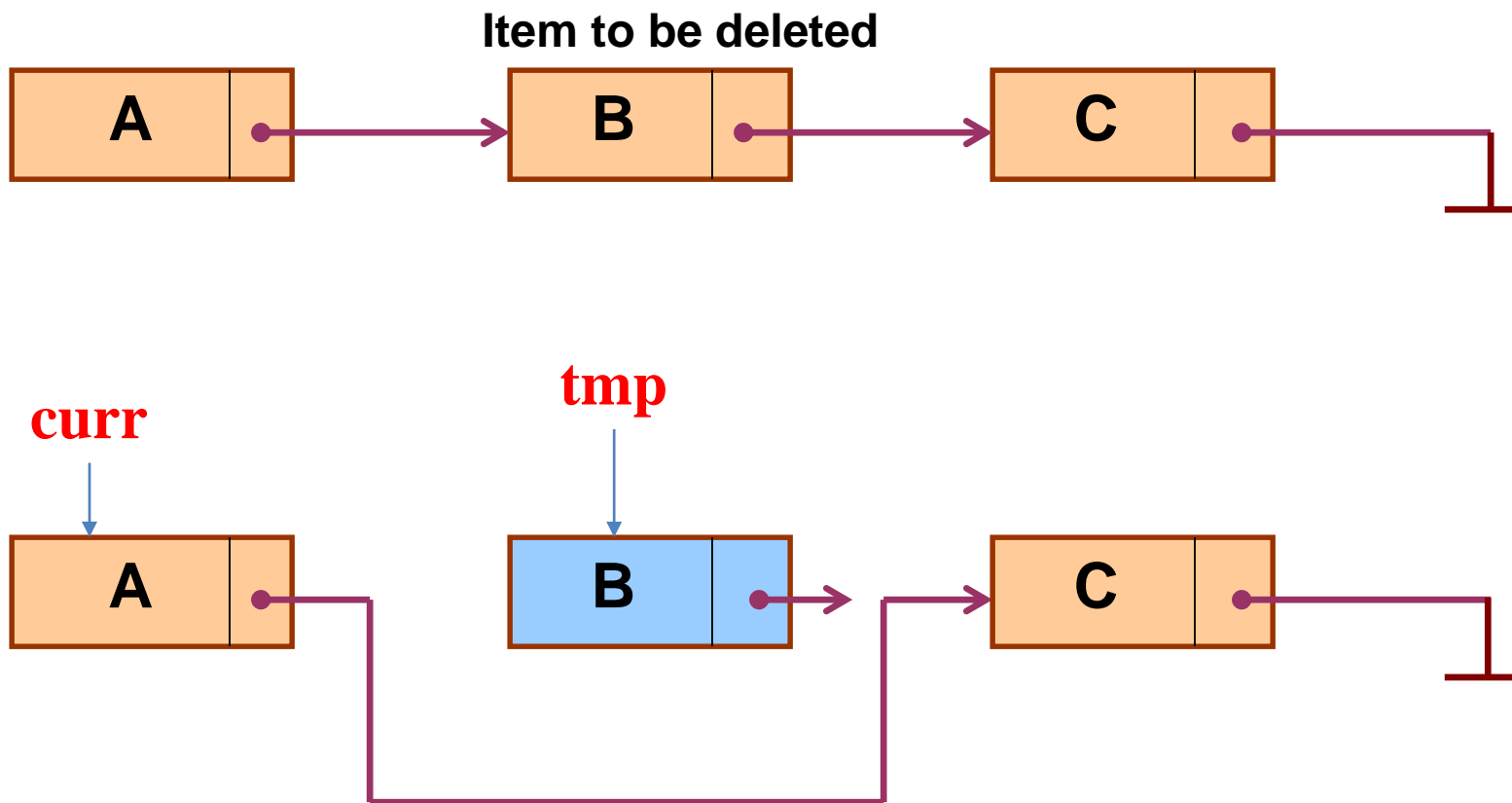
- Keeping track of a linked list:
 - Must know the pointer to the first element of the list (called *start*, *head*, etc.).
- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
 - Insert an element.
 - Delete an element.

Representation of a Linked List



Programming Code for Creation of a Linked list

```
typedef struct nd {  
    struct item data;  
    struct nd * next;  
} node;  
  
void insert(node *curr)  
{  
    node * tmp;  
  
    tmp=(node *) malloc(sizeof(node));  
    tmp->next=curr->next;  
    curr->next=tmp;  
}
```



```
typedef struct nd {  
    struct item data;  
    struct nd * next;  
} node;  
  
void delete(node *curr)  
{  
    node * tmp;  
    tmp=curr->next;  
    curr->next=tmp->next;  
    free(tmp);  
}
```

- For insertion:
 - A record is created holding the new item.
 - The next pointer of the new record is set to link it to the item which is to follow it in the list.
 - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
 - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.

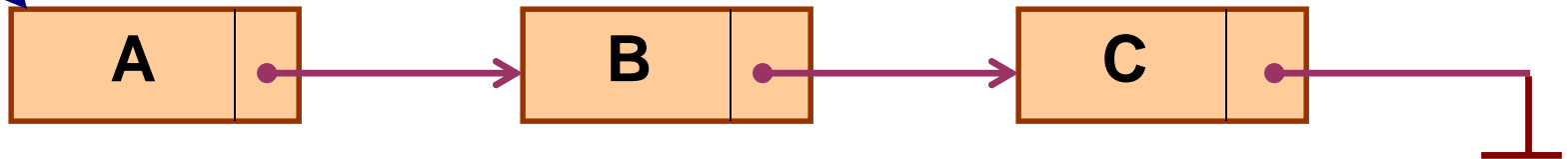
- Arrays are suitable for:
 - Inserting/deleting an element at the end.
 - Randomly accessing any element.
 - Searching the list for a particular value.
- Linked lists are suitable for:
 - Inserting an element.
 - Deleting an element.
 - Applications where sequential access is required.
 - In situations where the number of elements cannot be predicted beforehand.

Types of Lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.

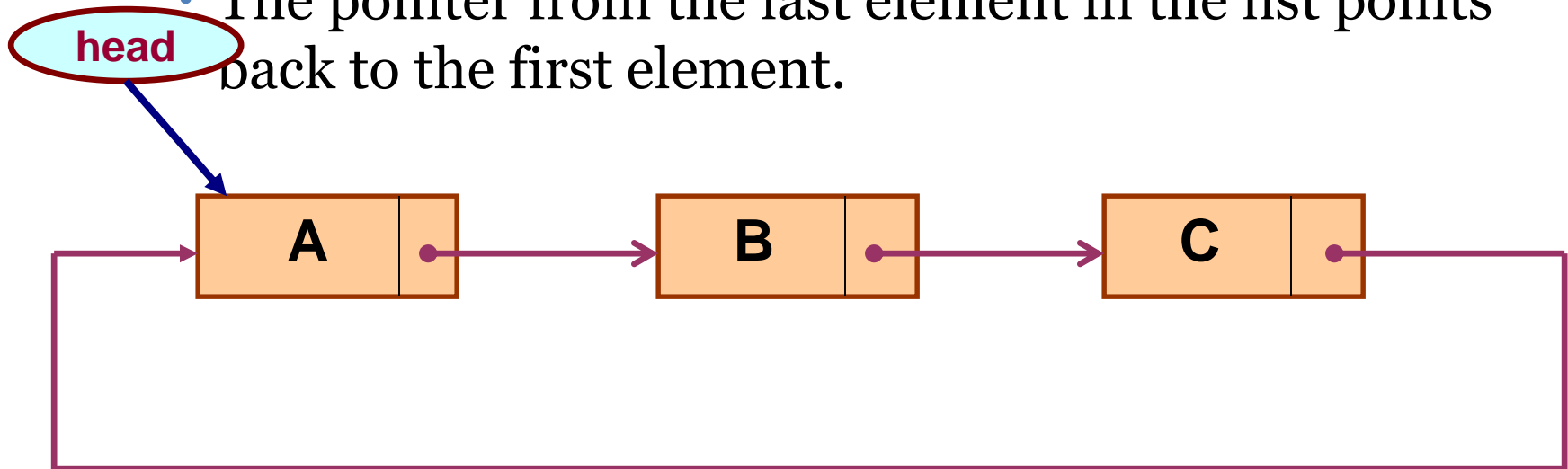
head Linear singly-linked list (or simply linear list)

- One we have discussed so far.



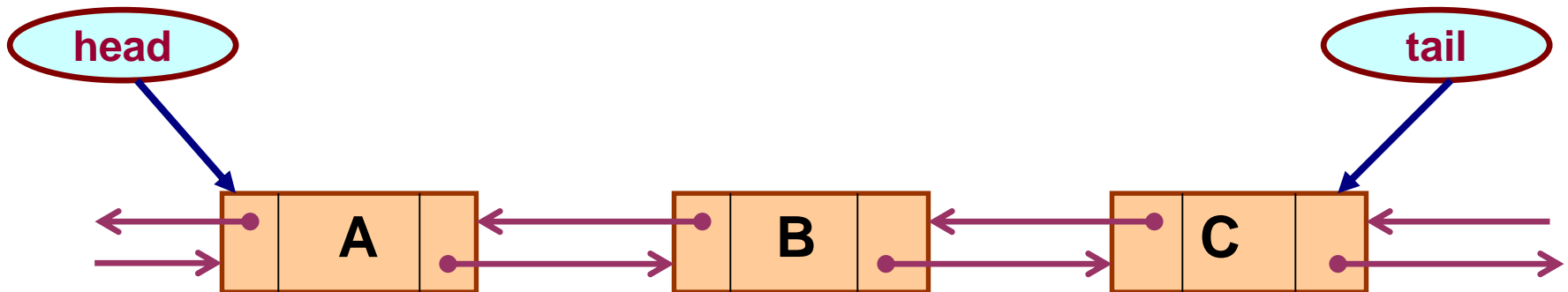
- Circular linked list

- The pointer from the last element in the list points back to the first element.



Doubly linked list

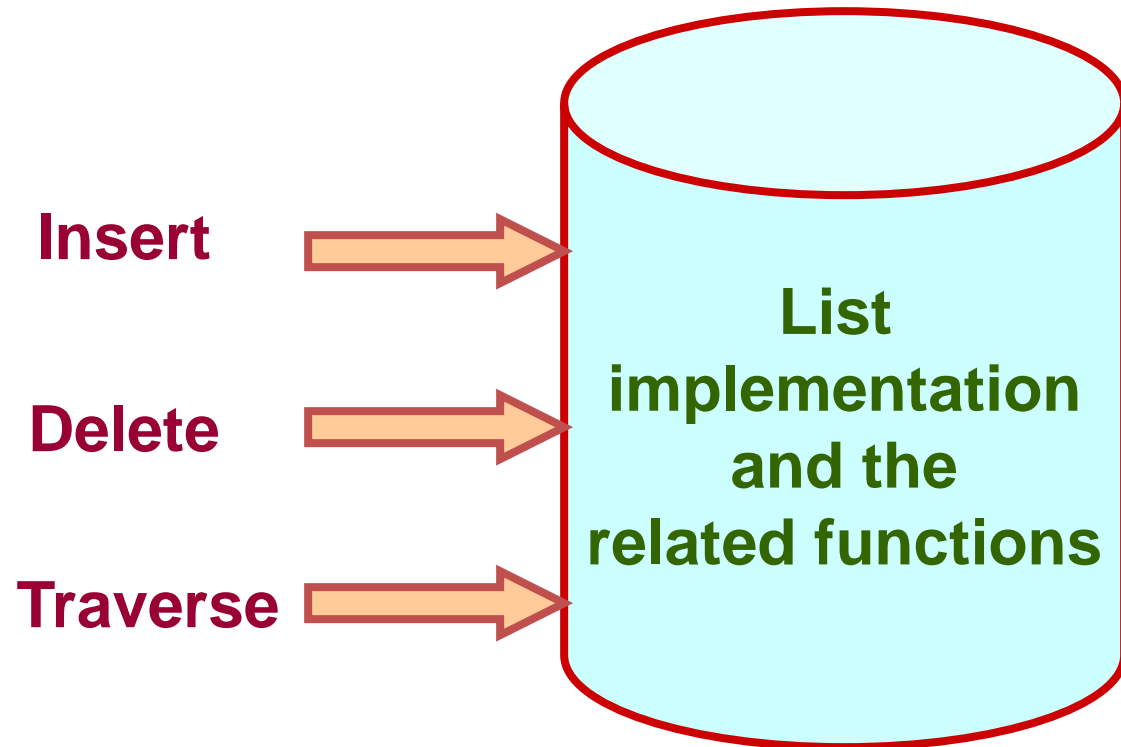
- Pointers exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- Usually two pointers are maintained to keep track of the list, *head* and *tail*.



Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

- What is an abstract data type?
 - It is a data type defined by the user.
 - Typically more complex than simple data types like *int*, *float*, etc.
- Why abstract?
 - Because details of the implementation are hidden.
 - When you do some operation on the list, say insert an element, you just call a function.
 - Details of how the list is implemented or how the insert function is written is no longer required.



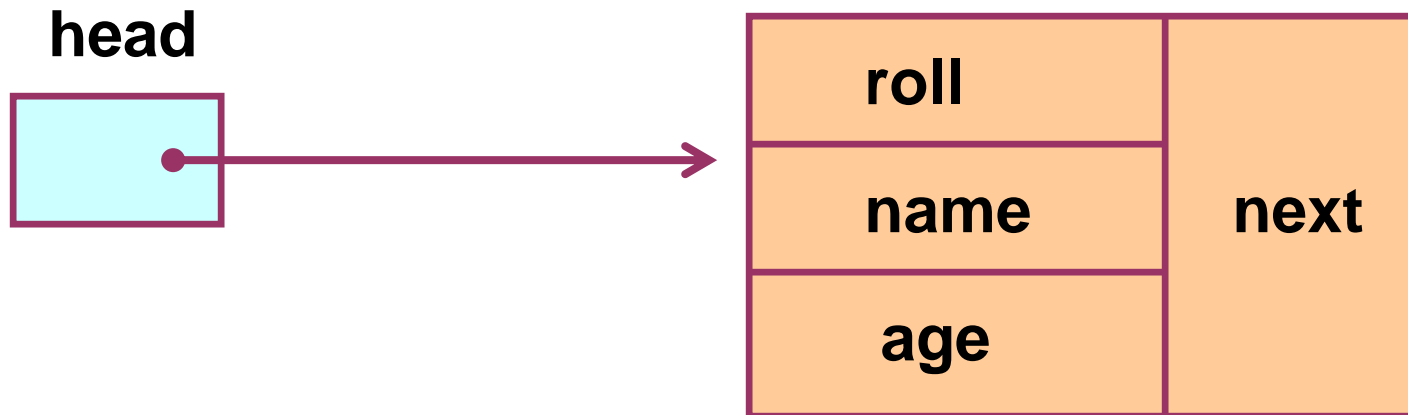
- Consider the structure of a node as follows:

```
struct stud {  
    int    roll;  
    char  name[25];  
    int    age;  
    struct stud *next;  
};
```

```
/* A user-defined data type called "node" */  
typedef struct stud node;  
node *head;
```

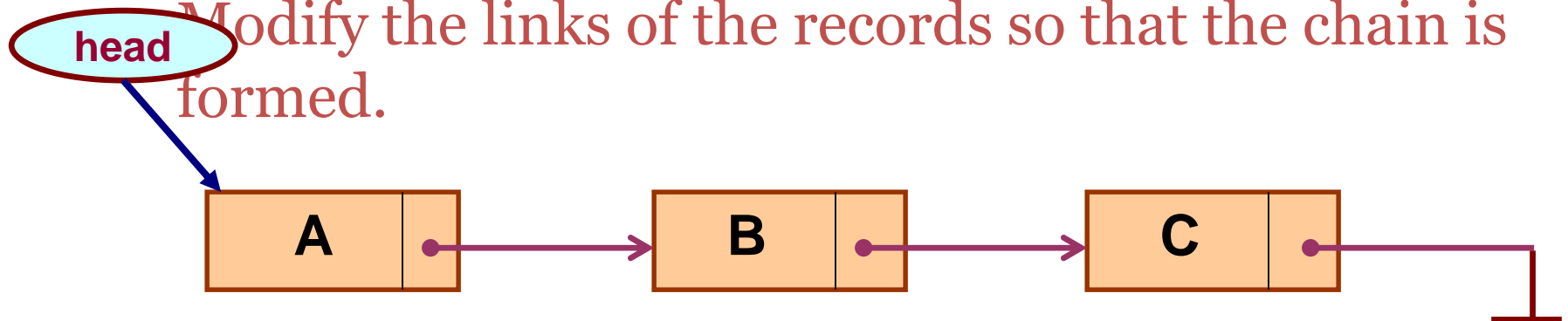

- To start with, we have to create a node (the first node), and make **head** point to it.

```
head = (node *) malloc(sizeof(node));
```



- If there are **n** number of nodes in the initial linked list:
 - Allocate **n** records, one by one.
 - Read in the fields of the records.

Modify the links of the records so that the chain is formed.



```
node *create_list()
{
    int k, n;
    node *p, *head;

    printf ("\n How many elements to enter?");
    scanf ("%d", &n);

    for (k=0; k<n; k++)
    {
        if (k == 0) {
            head = (node *) malloc(sizeof(node));
            p = head;
        }
        else {
            p->next = (node *) malloc(sizeof(node));
            p = p->next;
        }

        scanf ("%d %s %d", &p->roll, p->name, &p->age);
    }

    p->next = NULL;
    return (head);
}
```

Traversing of a Linked List

- Once the linked list has been constructed and *head* points to the first node of the list,
 - Follow the pointers.
 - Display the contents of the nodes as they are traversed.
 - Stop when the *next* pointer points to NULL.

```
void display (node *head)
{
    int count = 1;
    node *p;

    p = head;
    while (p != NULL)
    {
        printf ("\nNode %d: %d %s %d", count,
                p->roll, p->name, p->age);

        count++;
        p = p->next;
    }
    printf ("\n");
}
```

Inserting a node in List

- The problem is to insert a node *before a specified node*.
 - Specified means some value is given for the node (called *key*).
 - In this example, we consider it to be `roll`.
- Convention followed:
 - If the value of roll is given as *negative*, the node will be inserted at the *end* of the list.

- When a node is added at the beginning,
 - Only one next pointer needs to be modified.
 - *head* is made to point to the new node.
 - New node points to the previously first element.
- When a node is added at the end,
 - Two next pointers need to be modified.
 - Last node now points to the new node.
 - New node points to NULL.
- When a node is added in the middle,
 - Two next pointers need to be modified.
 - Previous node now points to the new node.
 - New node points to the next node.

```
void insert (node **head)
{
    int k = 0, rno;
    node *p, *q, *new;

    new = (node *) malloc(sizeof(node));

    printf ("\nData to be inserted: ");
    scanf ("%d %s %d", &new->roll, new->name, &new->age);
    printf ("\nInsert before roll (-ve for end):");
    scanf ("%d", &rno);

    p = *head;

    if (p->roll == rno)          /* At the beginning */
    {
        new->next = p;
        *head = new;
    }
}
```



```
else
{
    while ((p != NULL) && (p->roll != rno))
    {
        q = p;
        p = p->next;
    }

    if (p == NULL)          /* At the end */
    {
        q->next = new;
        new->next = NULL;
    }
    else if (p->roll == rno)
        /* In the middle */
        {
            q->next = new;
            new->next = p;
        }
    }
}
```

**The pointers
q and p
always point
to consecutive
nodes.**

Deleting a node from llist

- Here also we are required to delete a specified node.
 - Say, the node whose `roll` field is given.
- Here also three conditions arise:
 - Deleting the first node.
 - Deleting the last node.
 - Deleting an intermediate node.

```
void delete (node **head)
{
    int rno;
    node *p, *q;

    printf ("\nDelete for roll :");
    scanf ("%d", &rno);

    p = *head;
    if (p->roll == rno)
        /* Delete the first element */
    {
        *head = p->next;
        free (p);
    }
}
```

```
else
{
    while ((p != NULL) && (p->roll != rno))
    {
        q = p;
        p = p->next;
    }

    if (p == NULL)          /* Element not found */
        printf ("\nNo match :: deletion failed");

    else if (p->roll == rno)
        /* Delete any other element */
        {
            q->next = p->next;
            free (p);
        }
    }
}
```