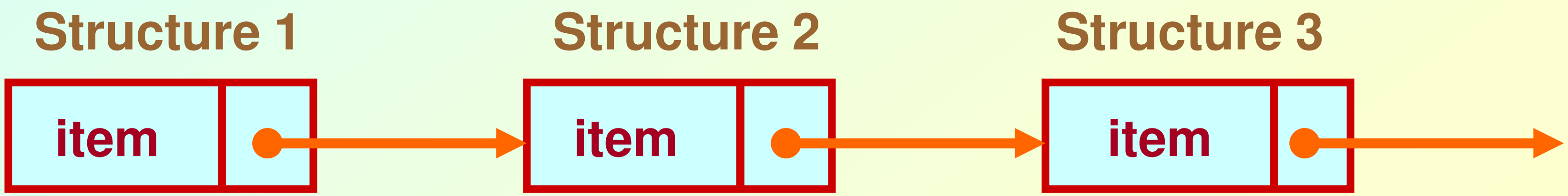
Self-referential Structures and Linked List

Linked List :: Basic Concepts

- A list refers to a set of items organized sequentially.
  - An array is an example of a list.
    - The array index is used for accessing and manipulating array elements.
  - Problems with array:
    - The array size has to be specified at the beginning.
    - Deleting an element or inserting an element may require shifting of elements in the array.

Contd.

- A completely different way to represent a list:
  - Make each item in the list part of a structure.
  - The structure also contains a pointer or link to the structure containing the next item.
  - This type of list is called a ***linked list***.



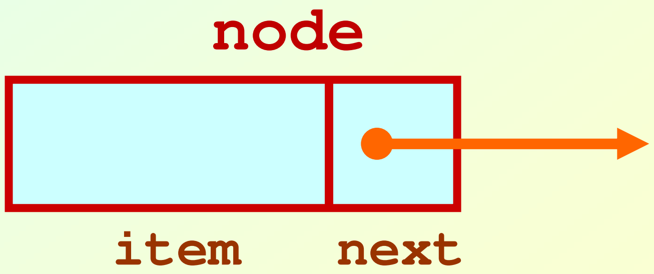
Contd.

- **Each structure of the list is called a *node*, and consists of two fields:**
  - **One containing the data item(s).**
  - **The other containing the address of the next item in the list (that is, a pointer).**
- **The data items comprising a linked list need not be contiguous in memory.**
  - **They are ordered by logical links that are stored as part of the data in the structure itself.**
  - **The link is a pointer to another structure of the same type.**

Contd.

- Such a structure can be represented as:

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



- Such structures which contain a member field pointing to the same structure type are called ***self-referential structures***.

Contd.

- In general, a node may be represented as follows:

```
struct node_name
     {
        type  member1;
        type  member2;
         .........
        struct node_name *next;
     }
```

Illustration

- **Consider the structure:**

```
struct stud
    {
        int  roll;
        char name[30];
        int  age;
        struct stud *next;
    }
```

- **Also assume that the list consists of three nodes `n1`, `n2` and `n3`.**

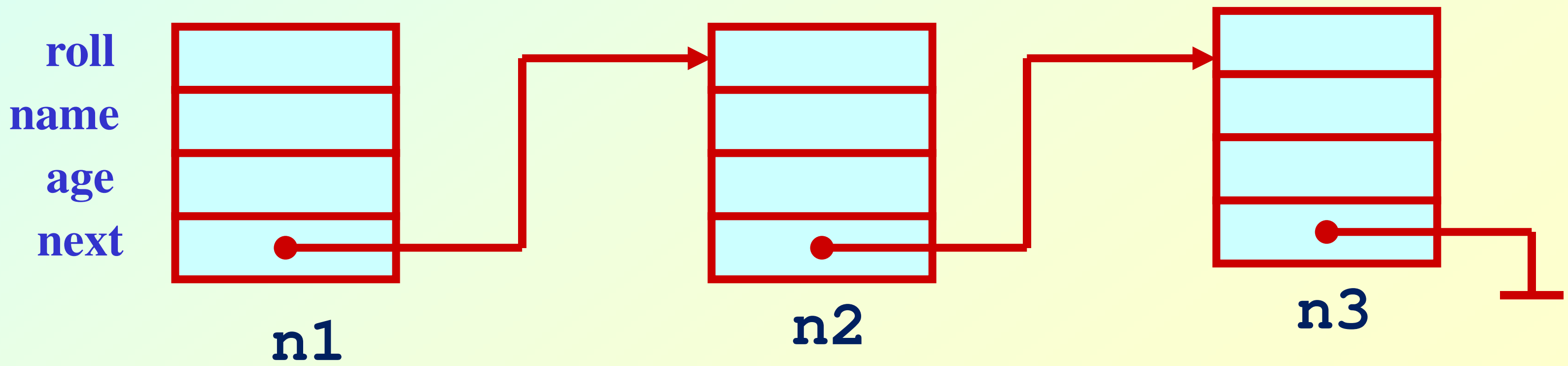
```
struct stud n1, n2, n3;
```

Contd.

- **To create the links between nodes, we can write:**

```
n1.next = &n2;
n2.next = &n3;
n3.next = NULL;    /* No more nodes follow */
```

- **Now the list looks like:**



- **Some important observations:**
  - **The `NULL` pointer is used to indicate that no more nodes follow, that is, it is the end of the list.**
  - **To use a linked list, we only need a *pointer to the first element* of the list.**
  - **Following the chain of pointers, the successive elements of the list can be accessed by *traversing* the list.**

Example: without using function

```
#include  <stdio.h>
struct  stud
  {
      int   roll;
      char  name[30];
      int   age;
      struct  stud  *next;
  }

main()
{
    struct  stud  n1, n2, n3;
    struct  stud  *p;

    scanf ("%d %s %d", &n1.roll, n1.name, &n1.age);
    scanf ("%d %s %d", &n2.roll, n2.name, &n2.age);
    scanf ("%d %s %d", &n3.roll, n3.name, &n3.age);
```

```
    n1.next  =  &n2;
    n2.next  =  &n3;
    n3.next  =  NULL;

   /* Now traverse the list and print the elements */

    p = &n1;   /* point to 1st element */
    while (p != NULL)
    {
      printf ("\n %d %s %d", p->roll, p->name, p->age);
      p = p->next;
    }
}
```

A function to carry out traversal

```
#include  <stdio.h>
struct  stud
  {
      int   roll;
      char  name[30];
      int   age;
      struct  stud  *next;
  }


void  traverse (struct stud *head)
{
   while (head != NULL)
   {
     printf ("\n %d %s %d", head->roll, head->name,
                                head->age);
     head = head->next;
   }
}
```

The corresponding main() function

```
main()
{
    struct  stud  n1, n2, n3, *p;

    scanf ("%d %s %d", &n1.roll, n1.name, &n1.age);
    scanf ("%d %s %d", &n2.roll, n2.name, &n2.age);
    scanf ("%d %s %d", &n3.roll, n3.name, &n3.age);

    n1.next  =  &n2;
    n2.next  =  &n3;
    n3.next  =  NULL;

    p = &n1;
    traverse (p);
}
```

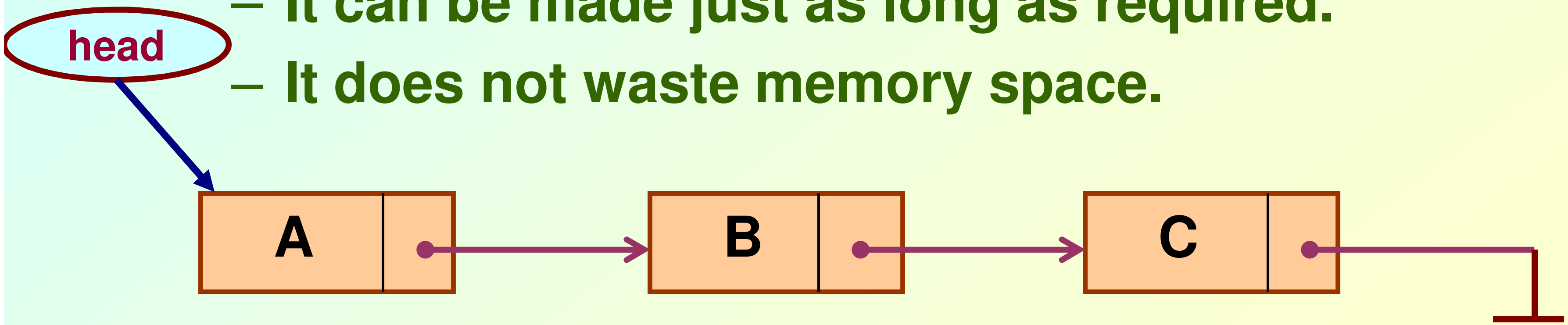
Alternative and More General Way

- **Dynamically allocate space for the nodes.**
  - Use `malloc()` or `calloc()` for allocating space for every individual nodes.
  - No need for allocating additional space unnecessarily like in an array.

Linked List in more detail

Introduction

- **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.



- 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.

Illustration: Insertion

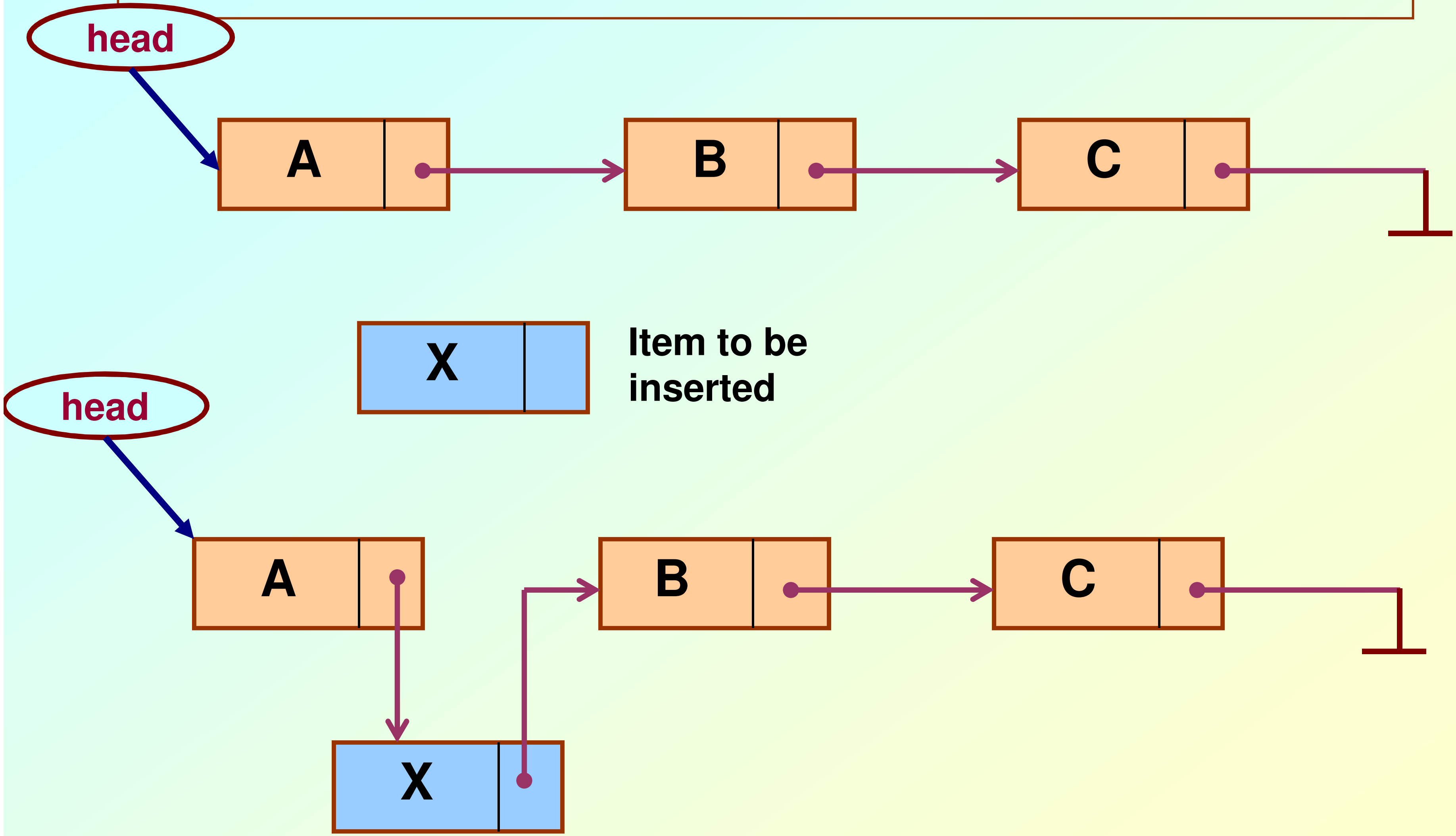
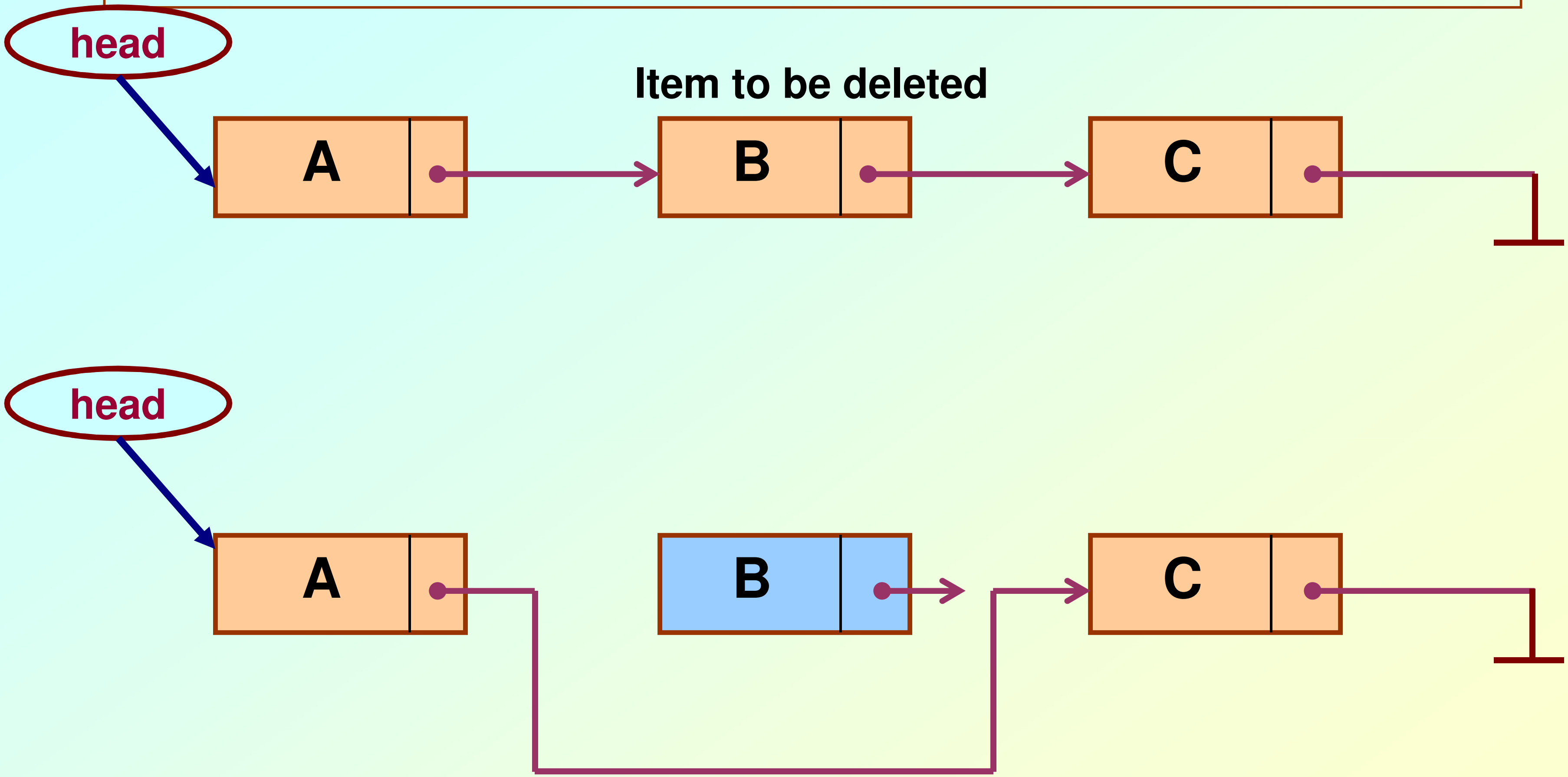


Illustration: Deletion



In essence ...

- **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.

Array versus Linked Lists

- **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.