CS11001/CS11002 Programming and Data Structures (PDS) (Theory: 3-0-0)

Teacher: Sourangshu Bhattacharya
sourangshu@gmail.com
http://cse.iitkgp.ac.in/~sourangshu/

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

Multi Dimensional Arrays

Two Dimensional Arrays

- We have seen that an array variable can store a list of values.
- Many applications require us to store a table of values.
- The table contains a total of 20 values, five in each line.
  - The table can be regarded as a matrix consisting of four rows and five columns.
- C allows us to define such tables of items by using two-dimensional arrays.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Student 1	75	82	90	65	76
Student 2	68	75	80	70	72
Student 3	88	74	85	76	80
Student 4	50	65	68	40	70

Declaring 2-D Arrays

- General form:

  ```
  data_type   array_name [row_size][column_size];
  ```

- Examples:

  ```
  int  marks[4][5];
  float  sales[12][25];
  double  matrix[100][100];
  ```

Accessing Elements of a 2-D Array

- Similar to that for 1-D array, but use two indices.
  - First indicates row, second indicates column.
  - Both the indices should be expressions which evaluate to integer values.

- Examples:

```
x[m][n] = 0;
c[i][k] += a[i][j] * b[j][k];
a = sqrt (a[j*3][k]);
```

Read the elements of a 2-D array

- By reading them one element at a time

```
for  (i=0; i<nrow; i++) {
    for  (j=0; j<ncol; j++) {
        scanf  ("%d", &a[i][j]);
    }
}
```

- The ampersand (&) is necessary.
- The elements can be entered all in one line or in different lines.

Print the elements of a 2-D array

```
for  (i=0; i<nrow; i++)
          for  (j=0; j<ncol; j++)
             printf  ("\n %d", a[i][j]);
```

– The elements are printed one per line.

```
for  (i=0; i<nrow; i++)
     for  (j=0; j<ncol; j++)
        printf  ("%d", a[i][j]);
```

– The elements are all printed on the same line.

```
for  (i=0; i<nrow; i++)  {
     printf  ("\n");
     for  (j=0; j<ncol; j++)
        printf ("%d   ", a[i][j]);
   }
```

– The elements are printed nicely in matrix form.

Example: Matrix Addition

```
#include  <stdio.h>

void main()
{
   int  a[100][100], b[100][100],
           c[100][100], p, q, m, n;

   scanf (“%d %d”, &m, &n);

   for  (p=0; p<m; p++) {
      for  (q=0; q<n; q++) {
         scanf (“%d”, &a[p][q]);
      }
   }

   for  (p=0; p<m; p++) {
      for  (q=0; q<n; q++) {
         scanf (“%d”, &b[p][q]);
      }
   }

   for  (p=0; p<m; p++) {
      for  (q=0; q<n; q++) {
         c[p][q] = a[p][q] + b[p][q];
      }
   }

   for  (p=0; p<m; p++) {
      printf  (“\n”);
      for  (q=0; q<n; q++)  {
         printf (“%d  ”, a[p][q]);
      }
   }
}
```

How to print three matrices side by side?

2 3 4	1 2 3	3 5 7
2 1 3	6 7 5	8 8 8
2 1 5	3 3 3	5 4 8

Passing 2-D Arrays

- Similar to that for 1-D arrays.
  - The array contents are not copied into the function.
  - Rather, the address of the first element is passed.
- For calculating the address of an element in a 2-D array, we need:
  - The starting address of the array in memory.
  - Number of bytes per element.
  - Number of columns in the array.
- The above three pieces of information must be known to the function.

The Actual Mechanism

- When an array is passed to a function, the values of the array elements are not passed to the function.
  - The array name is interpreted as the address of the first array element.
  - The formal argument therefore becomes a pointer to the first array element.
  - When an array element is accessed inside the function, the address is calculated using the formula stated before.
  - Changes made inside the function are thus also reflected in the calling program.

Example Usage

```
#include  <stdio.h>

main()
{
   int  a[15][25],  b[15]25];
   :
   :
   add (a, b, 15, 25);
   :
}
```

```
void  add (int x[][25],int y[][25], int rows, int cols)
{
   :
}
```

We can also write

int x[15][25], y[15][25];

Number of columns

Example: Transpose of a matrix

```
void  transpose (int x[][100], int n)
{
    int  p, q;

    for  (p=0; p<n; p++) {
        for  (q=0; q<n; q++)
          {
              t = x[p][q];
              x[p][q] = x[q][p];
              x[q][p] = t;
          }
    }
}
```

10	20	30
40	50	60
70	80	90

a[100][100]

transpose(a,3)

10	20	30
40	50	60
70	80	90

The Correct Version

```
void  transpose (int x[][100], n)
{
   int  p, q;

   for  (p=0; p<n; p++)
      for  (q=p; q<n; q++)
        {
           t = x[p][q];
           x[p][q] = x[q][p];
           x[q][p] = t;
        }
}
```

10	20	30
40	50	60
70	80	90

↓

10	40	70
20	50	80
30	60	90

Multi-Dimensional Arrays

- How can you add more than two dimensions?
  - int a[100];
  - int b[100][100];
  - int c[100][100][100];
  - …….
  - How long?
  - Can you add any dimension?
  - Can you add any size?

Exercise

- Write a function to multiply two matrices of orders m x n and n x p respectively.

Homework

- Step -1: Read the number of persons from the user.
- Step -2: Read the first name of each of the persons.
- Step -3: Alphabetically sort their names.
- Step -4: Print the sorted list.

- Input:
  - Enter the number of persons: 3
  - Enter their first name:
    - Tridha
    - Susmita
    - Pranab

- Output:
    - Pranab
    - Susmita
    - Tridha

Multi Dimensional Array Initialization

- Example 1

```
int values[3][4] = {
                    {1,2,3,4},
                    {5,6,7,8},
                    {9,10,11,12}
                 };
```

- Example 2

```
int values[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

2D array to 1D array

- How?
  - Example 2D array

    1 2 3

    4 5 6

    7 8 9
  - Row-wise representation

    1 2 3 4 5 6 7 8 9
  - Column-wise representation

    1 4 7 2 5 8 3 6 9

- Why?
  - Chunk of memory is required.
  - May not be available.
  - 2D array of size 50X50 is available, but not 1D array of size 2500 **– POSSIBLE??**
  - 1D array of size 2500 is available, but not 2D array of size 50X50 **– POSSIBLE??**

2-D array representation in C

- Starting from a given memory location, the elements are stored row-wise in consecutive memory locations.

Example:

```
int A[5][4];
```

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]
A[3][0]	A[3][1]	A[3][2]	A[3][3]
A[4][0]	A[4][1]	A[4][2]	A[4][3]

2-D array representation in C

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]
A[3][0]	A[3][1]	A[3][2]	A[3][3]
A[4][0]	A[4][1]	A[4][2]	A[4][3]

A[0]0] A[0][1] A[0]2] A[0][3]	A[1][0] A[1][1] A[1][2] A[1][3]	A[2][0] A[2][1] A[2][2] A[2][3]
Row 0	**Row 1**	**Row 2**

- x: starting address of the array in memory
- c: number of columns
- k: number of bytes allocated per array element

a[i][j] ➔ is allocated at $x + (i * c + j) * k$

Problems

1. Write a C program to multiply two matrices ~~of orders m x n and n x p respectively~~.

2. Write a C program to multiply to large matrices.

Interactive Input

```
#include <stdlib.h>
void main()
{
	int **mat,nrows,ncols,i;


	……….
	mat=(int **)malloc(sizeof(int *)*nrows);
	for(i=0;i<nrows;i++)
		mat[i]=(int *)malloc(sizeof(int)*ncols);
	……………
	……………

	for(i=0;i<nrows;i++)
		free(mat[i]);
	free(mat);
}
```

Memory allocation (2D pointer)

Memory allocation (1D pointer)

Memory deallocation (1D pointer)

Memory deallocation (2D pointer)

2-D Array Allocation

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

int **allocate(int h, int w)
  {
   int **p;
   int i,j;

  p=(int **) calloc(h, sizeof (int *) );
   for(i=0;i<h;i++)
    p[i]=(int *) calloc(w,sizeof (int));
   return(p);
  }
```

Allocate array of pointers

Allocate array of integers for each row

```
void read_data(int **p,int h,int w)
 {
    int i,j;
    for(i=0;i<h;i++)
    for(j=0;j<w;j++)
      scanf ("%d",&p[i][j]);
 }
```

Elements accessed like 2-D array elements.

2-D Array Allocation

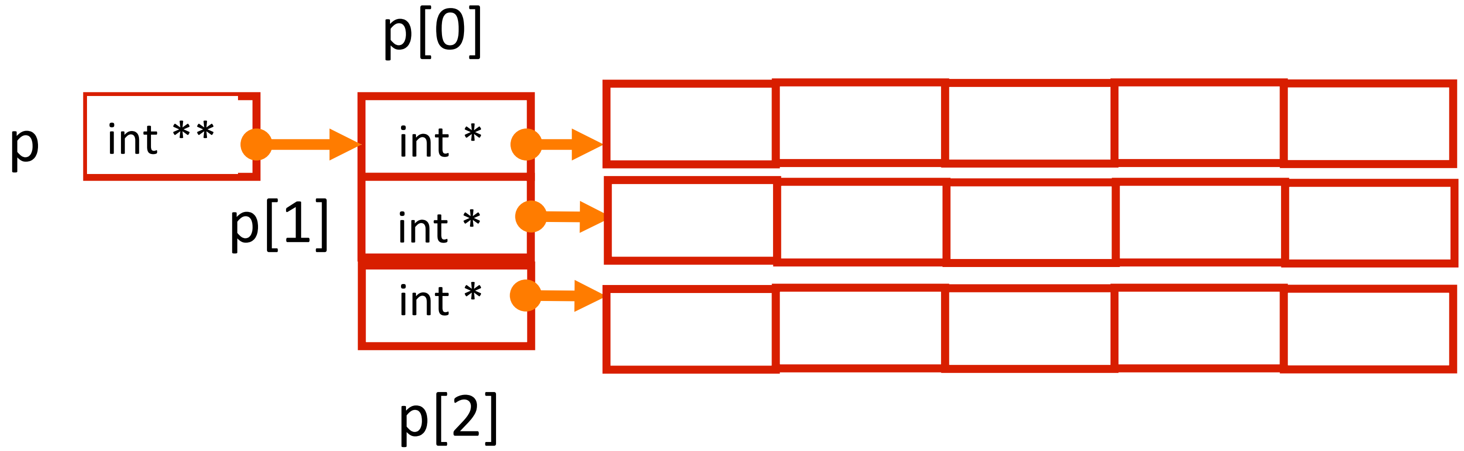
```
void print_data(int **p,int h,int w)
 {
    int i,j;
     for(i=0;i<h;i++)
     {
     for(j=0;j<w;j++)
       printf("%5d ",p[i][j]);
      printf("\n");
     }
}
```

```
Give M and N
3 3
1 2 3
4 5 6
7 8 9

 The array read as
    1    2    3
    4    5    6
    7    8    9
```

```
void main()
{
  int **p;
  int M,N;

  printf("Give M and N \n");
  scanf("%d%d",&M,&N);
  p=allocate(M,N);
  read_data(p,M,N);
  printf("\n The array read as \n");
  print_data(p,M,N);
}
```

Pointer to Pointer

```
int **p;
p=(int **) malloc(3 * sizeof(int *));
p[0]=(int *) malloc(5 * sizeof(int));
    p[1]=(int *) malloc(5 * sizeof(int));
    p[2]=(int *) malloc(5 * sizeof(int));
```



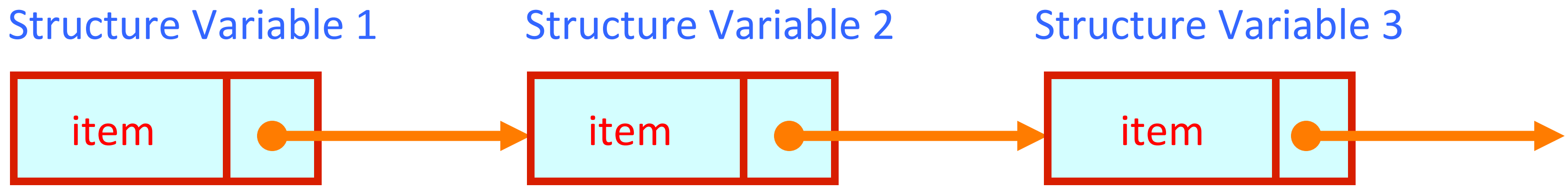
Linked Lists

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 manipulation of 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.

Linked List

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

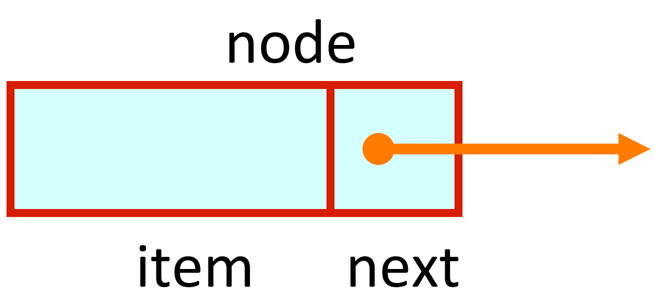


Linked List Facts

- Each structure of the list is called a node, and consists of two fields:
  - Item(s).
  - Address of the next item in the list.
- 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.

Declaration of a linked list

```
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.

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.

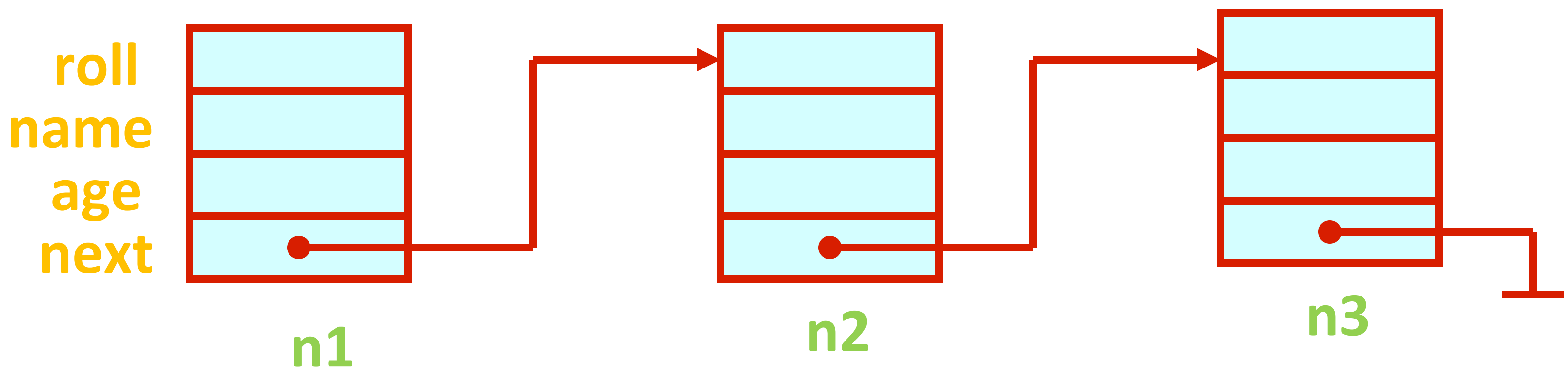
```
struct stud n1, n2, n3;
```

Illustration

- 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:



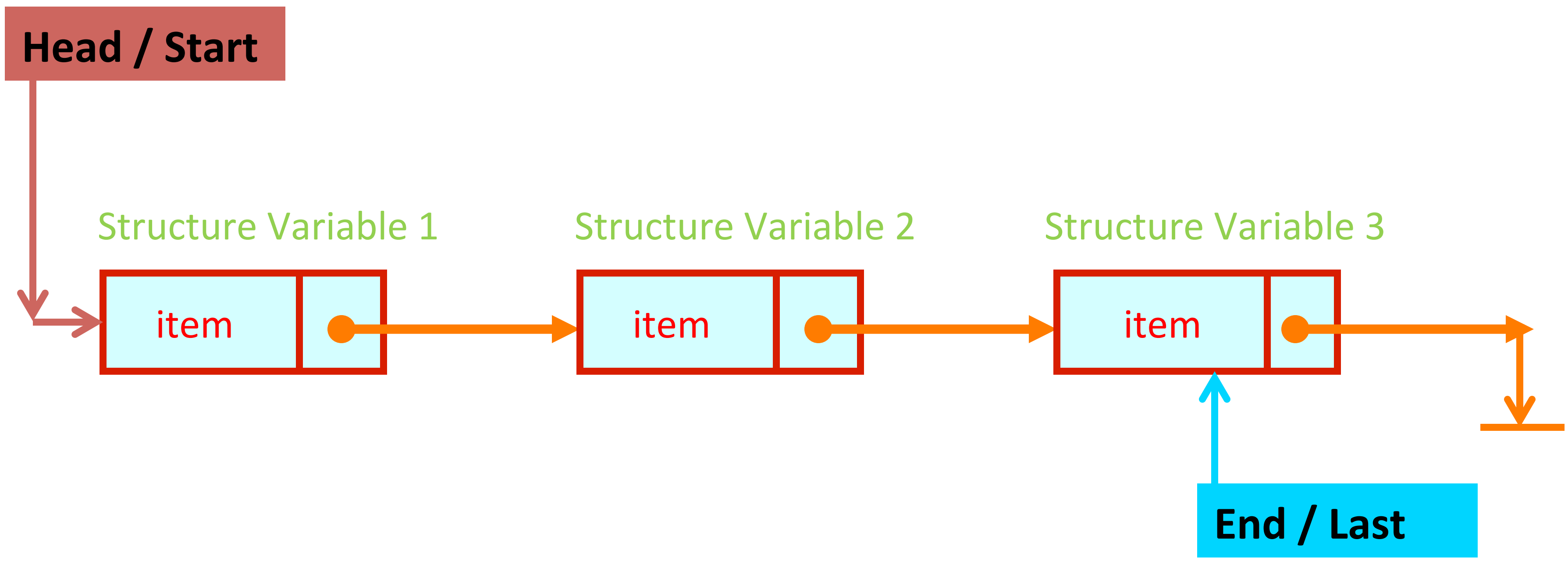
Example

```
#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);
```

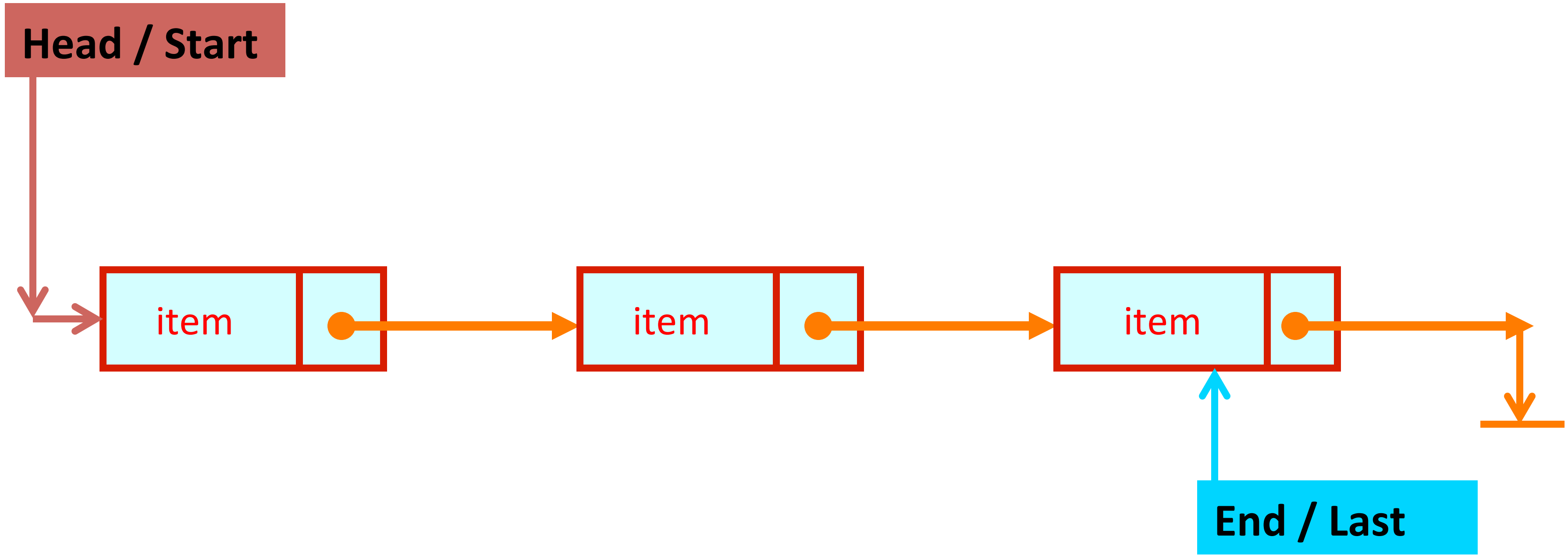
```
   n1.next  =  &n2 ;
   n2.next  =  &n3 ;
   n3.next  =  NULL ;

/* 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;
   }
}
```

Linked List

- Where to start and where to stop?



Print items of a linked list



Example

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

main()
{
   struct  stud  n1, n2, n3, *p;
……………..
 p  =  &n1 ;  /* point to 1st element */
   while  (p != NULL)
   {
       printf (“\n %d %s %d”, p->roll, p->name, p->age);
          p  =  p->next;
   }
……………
}
```

Insert into a linked list

Step 1: Create a new node.

Step 2: Copy the item.

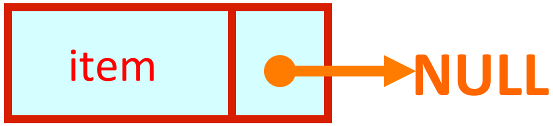
Step 3: Make the link/next as NULL (point nowhere)

Step 4:

- **Case 1:** If there does not exists any linked list.
  - *Step 4a:* Make the new node as head node.
  - *Step 4b:* Go to End.

- **Case 2:** Else
  - *Step 4c:* Locate the insertion point.
  - *Step 4d:* Insert the new node.
  - *Step 4e:* Adjust the link.
  - *Step 4f:* Go to End.

Insert into a linked list



Step 1: Create a new node.

Step 2: Copy the item.

Step 3: Make the link/next as NULL (point nowhere)

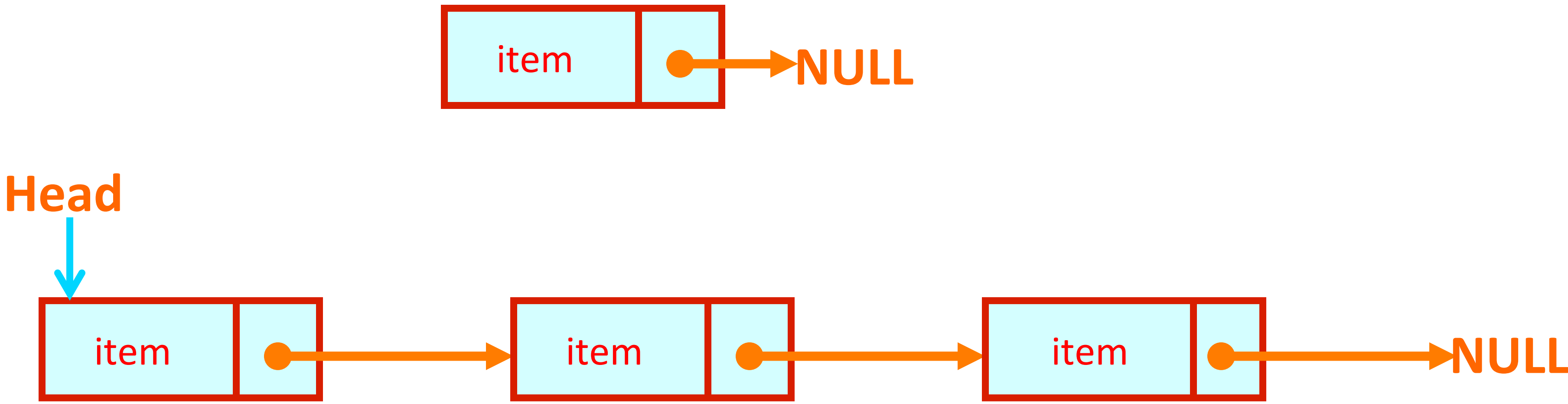
Step 4:

Case 1: If there does not exists any linked list.

Step 4a: Make the new node as head node.

Step 4b: Go to End.

Insert into a linked list



Step 1: Create a new node.

Step 2: Copy the item.

Step 3: Make the link/next as NULL (point nowhere)

Step 4:

Case 2: Else

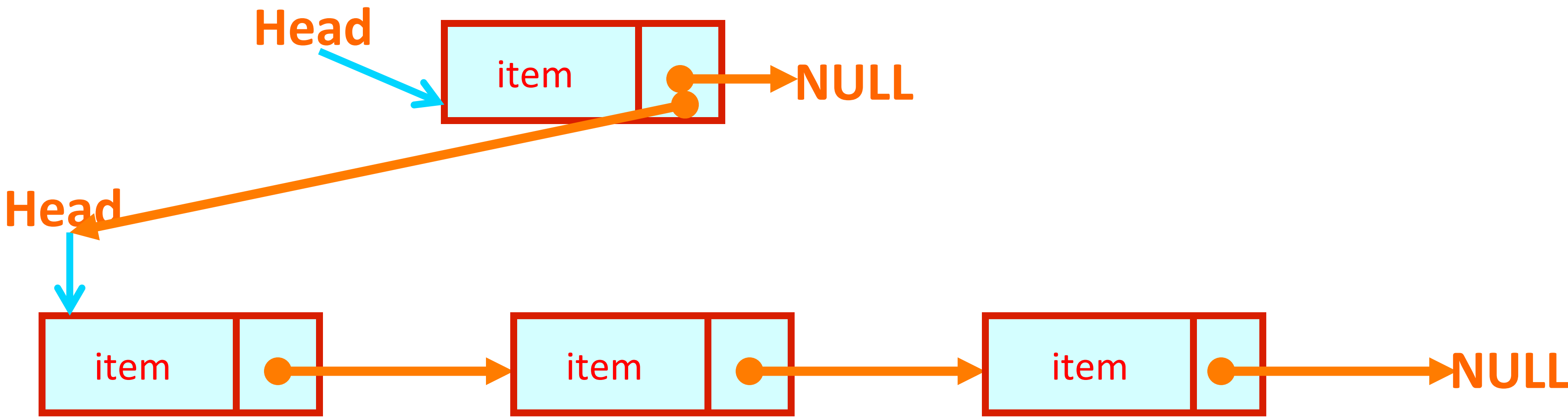
Step 4c: Locate the insertion point.

Step 4d: Insert the new node.

Step 4e: Adjust the link.

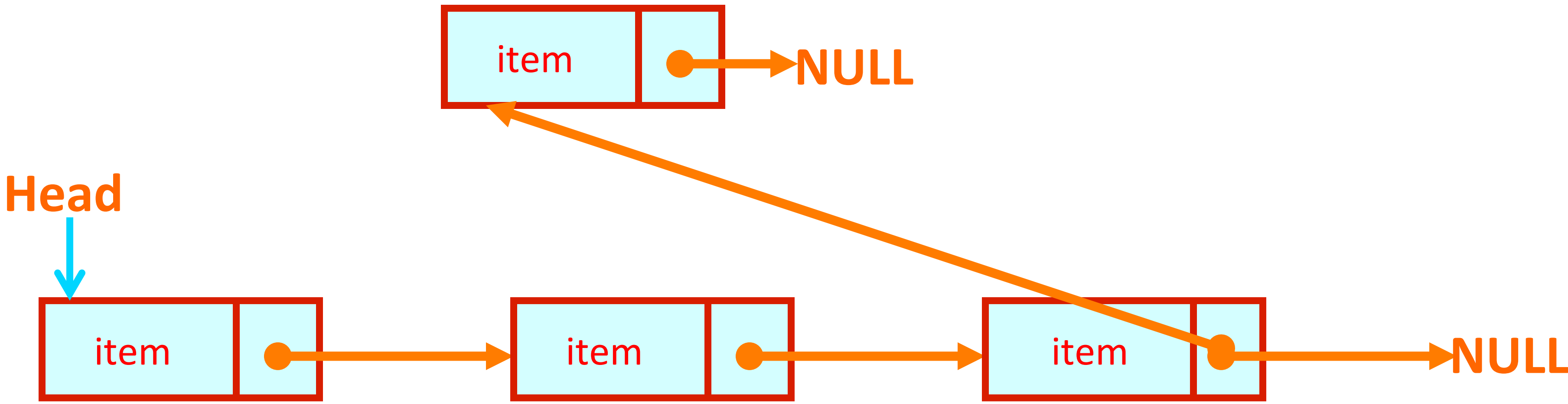
Step 4f: Go to End.

Insert into a linked list: Head Node



Step 4ci: Make the next of new node as the address of existing head node.
Step 4cii: Copy the address of the new node as the head node.

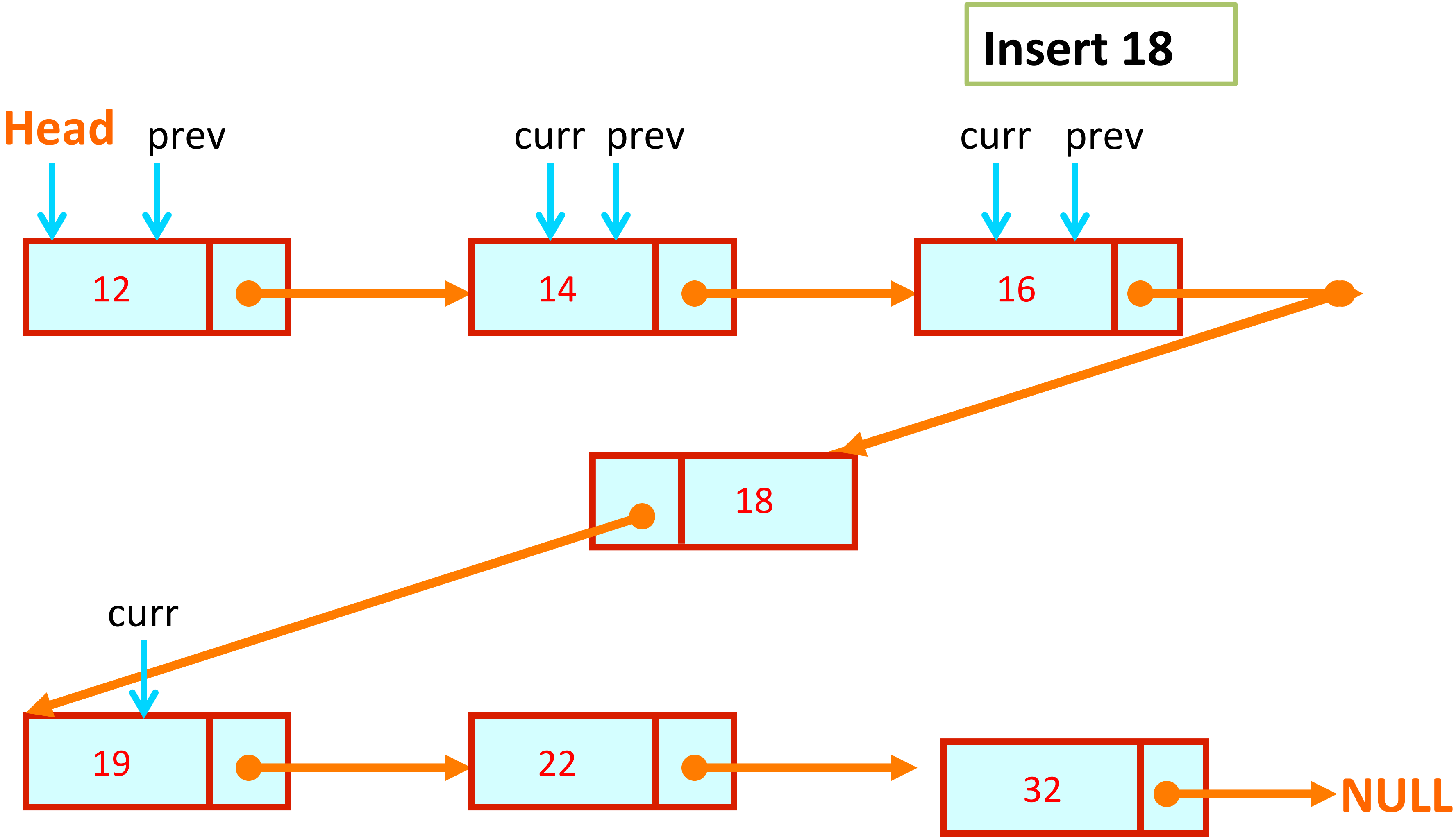
Insert into a linked list: End Node



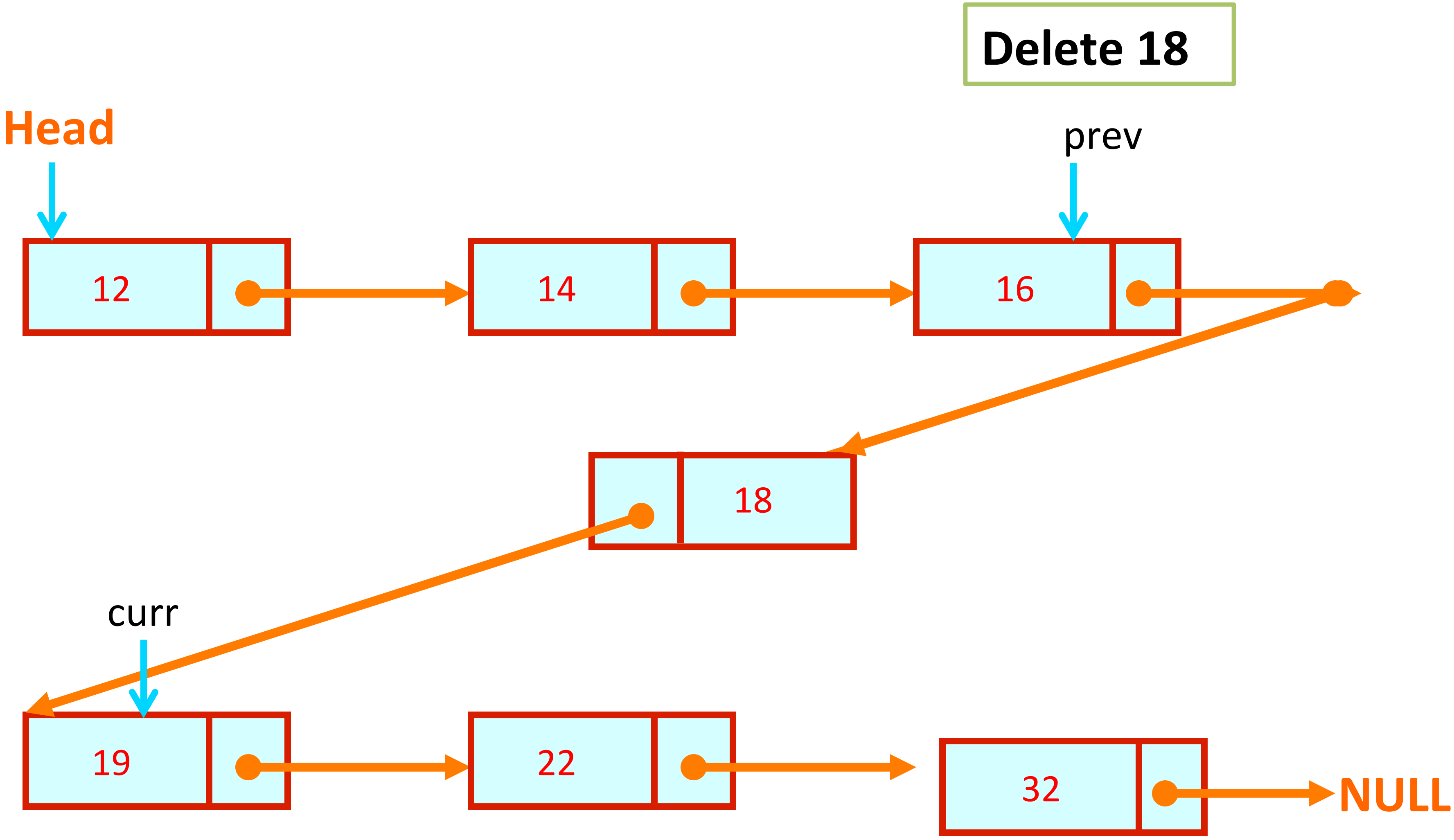
Step 4ci: Traverse till last/end node.

Step 4cii: Make the next of last node as the address of new node.

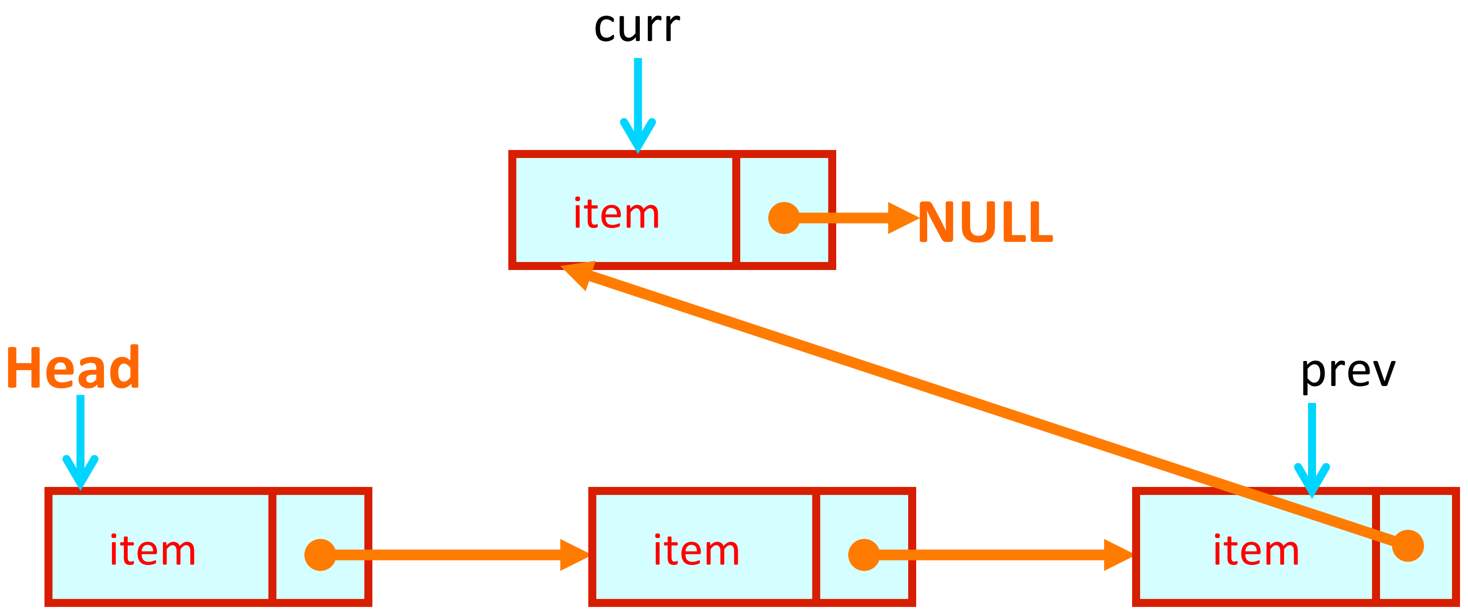
Insert into a sorted linked list



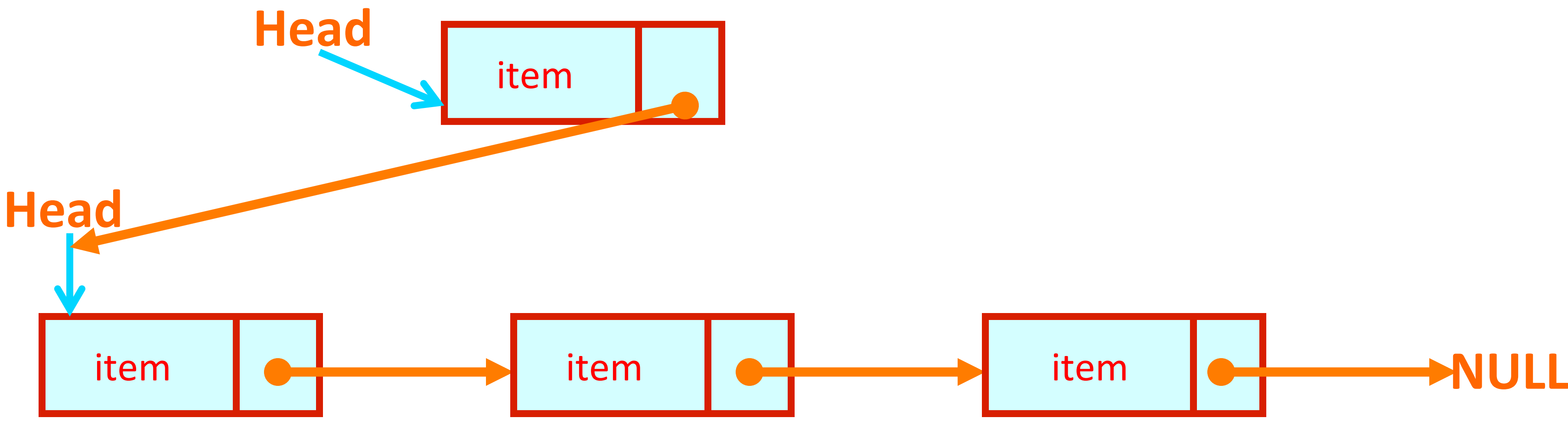
Delete a specific node from linked list



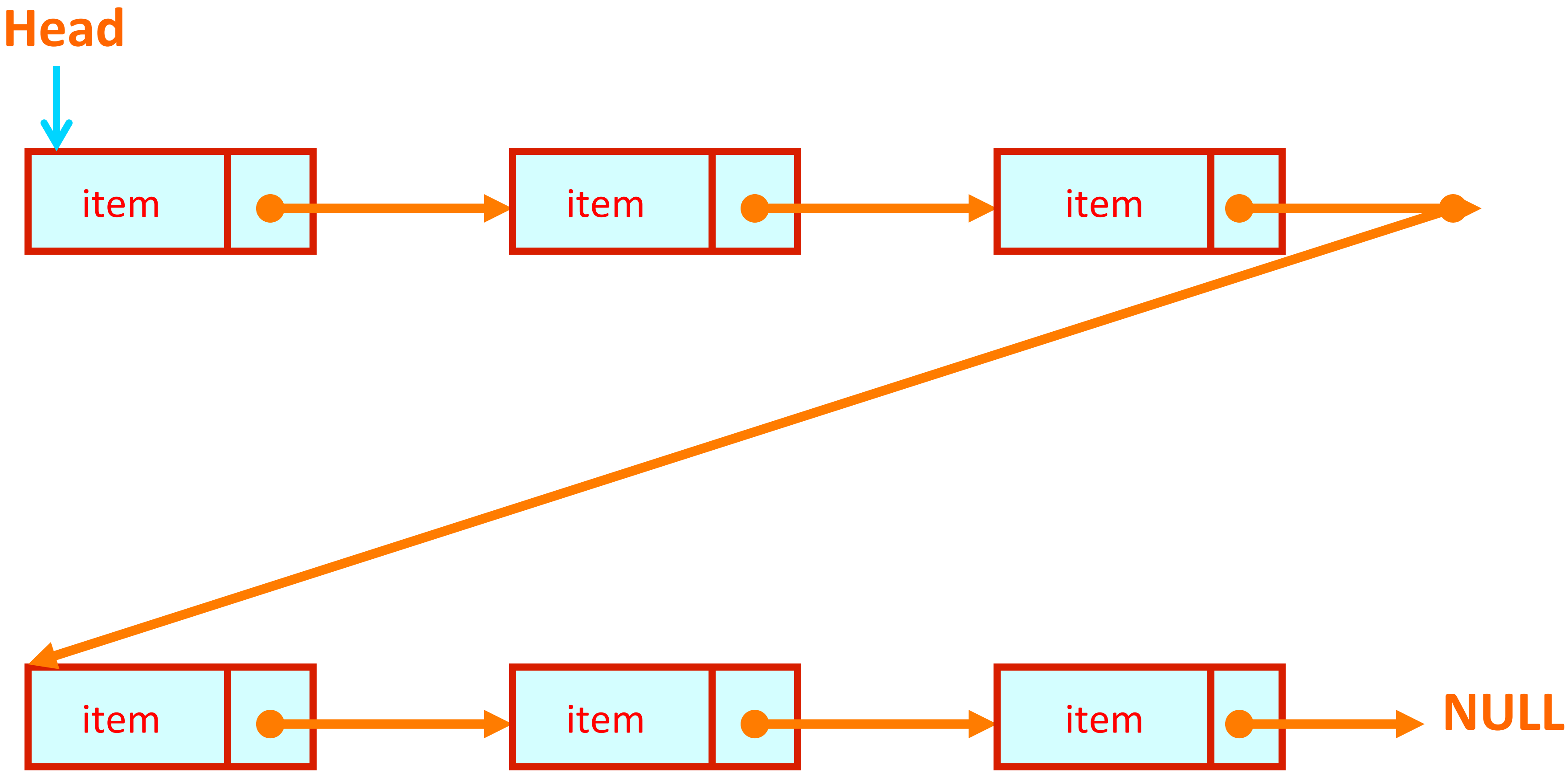
Delete End Node from linked list



Delete Head Node from a linked list



Linked list and Dynamic Memory Allocation



Linked list and Dynamic Memory Allocation

1. We need not have to know how many nodes are there.

2. Dynamic memory allocation provides a flexibility on the length of a linked list.

3. Example,

```
struct node {
    int item;
    struct node *next;
};
struct node *head, *temp;

temp=(struct node *)malloc(sizeof(struct node)*1);
temp->next=NULL;
temp->item=10;
head=temp;

free(temp);
```

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.

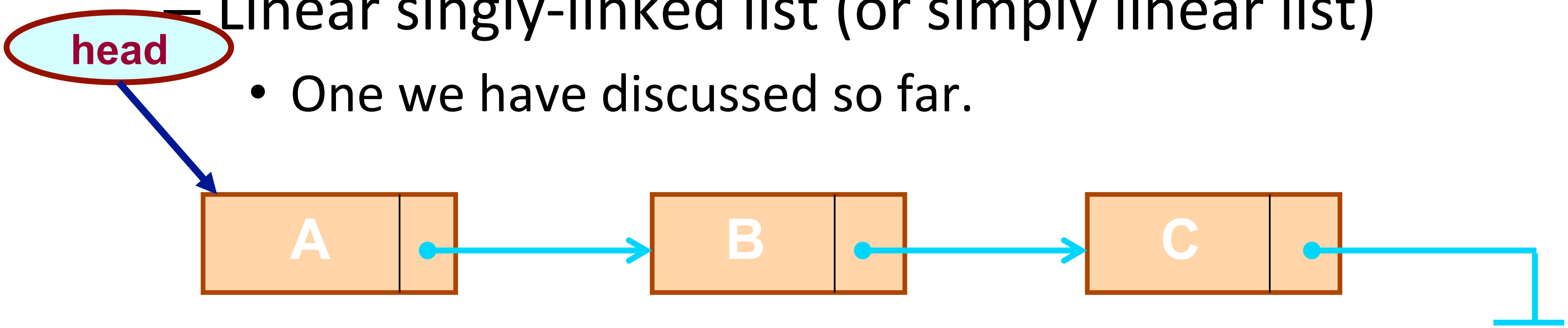
- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

Types of Lists

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

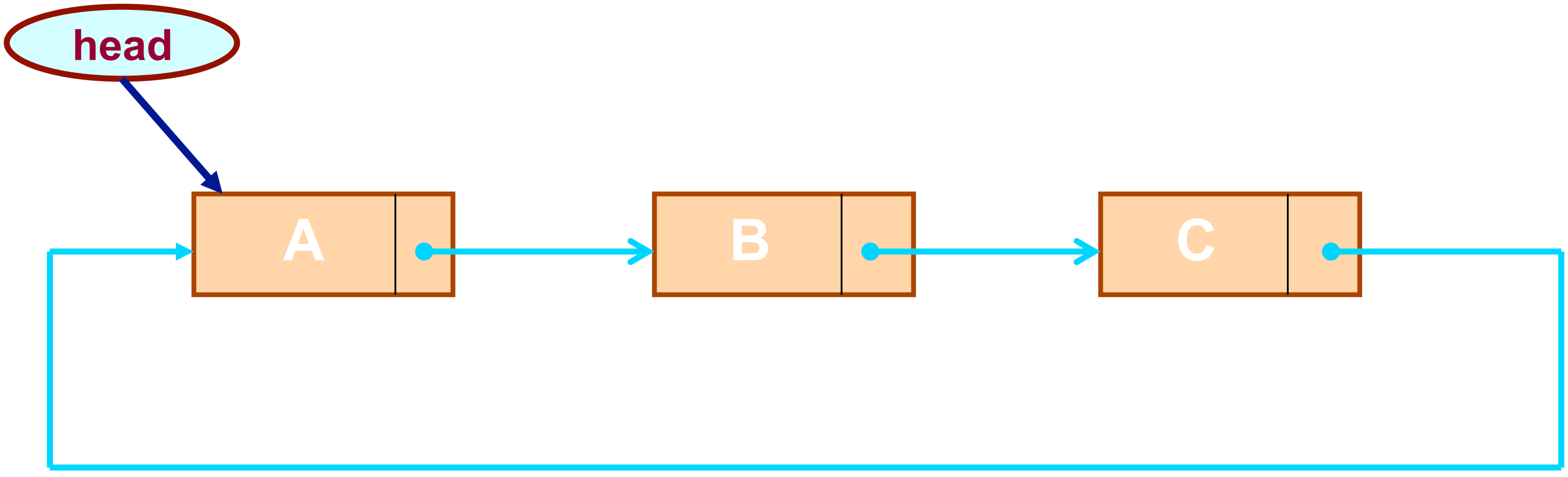
  – Linear singly-linked list (or simply linear list)

    - One we have discussed so far.



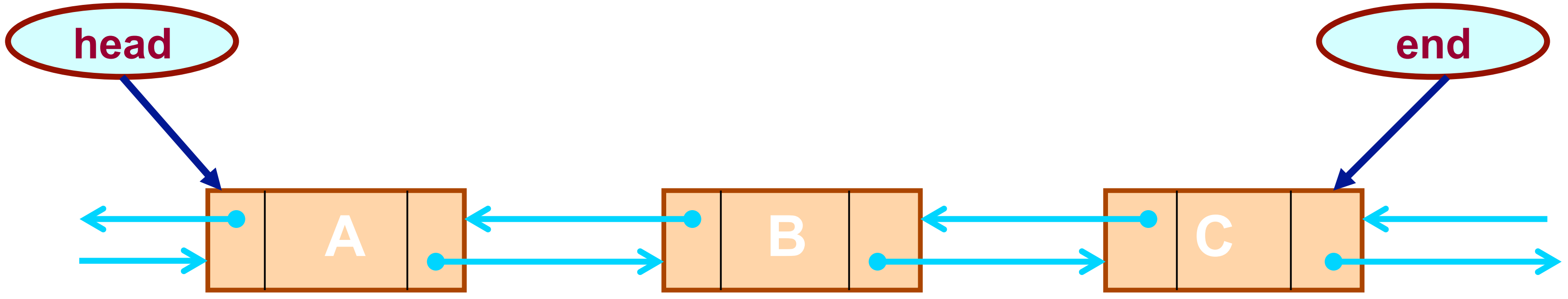
– **Circular linked list**

- 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*.



Sparse Matrix

- Why to use Sparse Matrix instead of simple matrix ?
  - Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
  - Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Sparse Matrix

- Storing non-zero elements with triples- (Row, Column, value).

- Two common representations:
  - Array representation
  - Linked list representation

Array Representation

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

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

Array Representation

- Row: Index of row, where non-zero element is located

- Column: Index of column, where non-zero element is located

- Value: Value of the non zero element located at index – (row,column)

- Total number non-zero elements.

Dense to sparse matrix

```
// C program for Sparse Matrix Representation
// using Linked Lists
#include<stdio.h>

int main()
{
    // Assume 4x5 sparse matrix
    int sparseMatrix[4][5] =
    {
        {0 , 0 , 3 , 0 , 4 },
        {0 , 0 , 5 , 7 , 0 },
        {0 , 0 , 0 , 0 , 0 },
        {0 , 2 , 6 , 0 , 0 }
    };

    int size = 0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)
            if (sparseMatrix[i][j] != 0)
                size++;
```

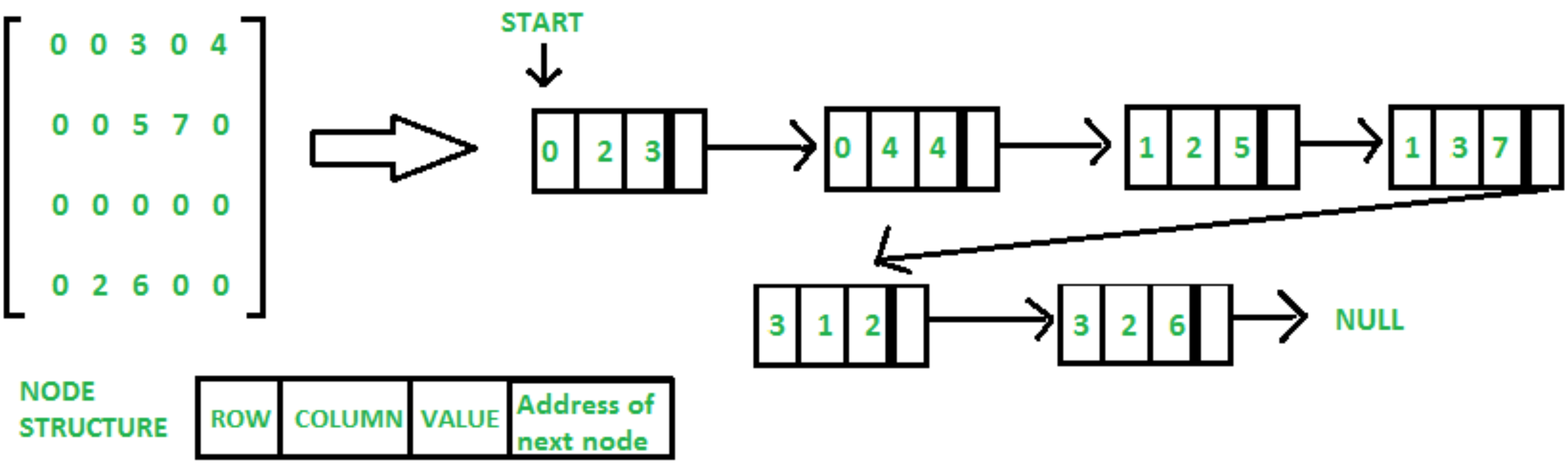
Dense to sparse matrix

```
 // number of columns in compactMatrix (size) must be
    // equal to number of non - zero elements in
    // sparseMatrix
    int compactMatrix[3][size];

    // Making of new matrix
    int k
    for (i = 0; i < 4; i++)
        for (j = 0; j < 5; j++)
            if (sparseMatrix[i][j] != 0)
            {
                compactMatrix[0][k] = i;
                compactMatrix[1][k] = j;
                compactMatrix[2][k] = sparseMatrix[i][j];
                k++;
            }

return 0;
}
```

Linked-list Representation

- Row: Index of row, where non-zero element is located
- Column: Index of column, where non-zero element is located
- Value: Value of the non zero element located at index – (row,column)
- Next node: Address of the next node

Linked-list Representation



Dense to sparse matrix

```
// C program for Sparse Matrix Representation
// using Linked Lists
#include<stdio.h>
#include<stdlib.h>

// Node to represent sparse matrix
struct Node
{
    int value;
    int row_position;
    int column_postion;
    struct Node *next;
};
```

Dense to sparse matrix

```
// Function to create new node
void create_new_node(struct Node** start, int non_zero_element,
                     int row_index, int column_index )
{
    struct Node *temp, *r;
    temp = *start;
    if (temp == NULL)
    {
        // Create new node dynamically
        temp = (struct Node *) malloc (sizeof(struct Node));
        temp->value = non_zero_element;
        temp->row_position = row_index;
        temp->column_postion = column_index;
        temp->next = NULL;
        *start = temp;
    }
    else
    {
        while (temp->next != NULL)
            temp = temp->next;
        // Create new node dynamically
       …
    }
}
```

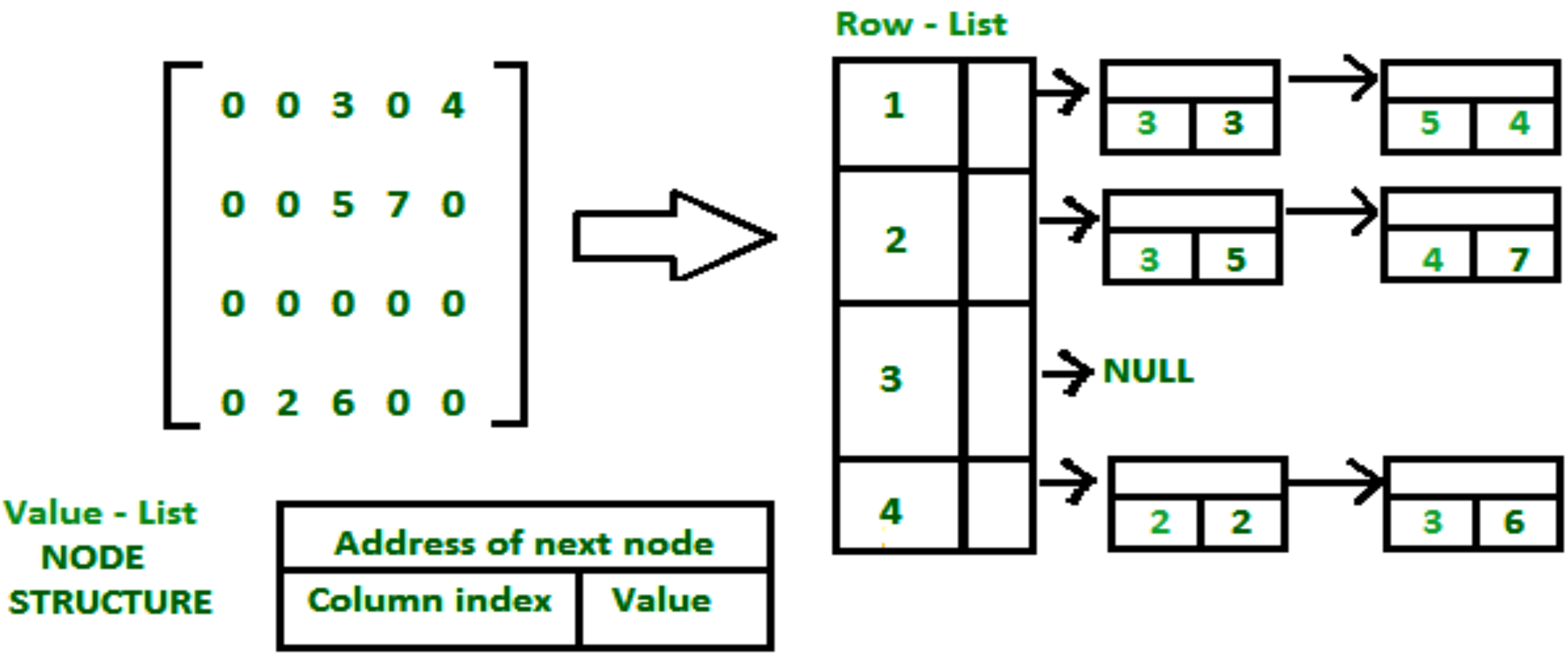
Dense to sparse matrix

```
// Driver of the program
int main()
{
   // Assume 4x5 sparse matrix
/* Start with the empty list */
    struct Node* start = NULL;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)

            // Pass only those values which are non - zero
            if (sparseMatric[i][j] != 0)
                create_new_node(&start, sparseMatric[i]
[j], i, j);

    PrintList(start);
    return 0;
}
```

Dense to sparse matrix

```
// This function prints contents of linked list
// starting from start
void PrintList(struct Node* start)
{
    struct Node *temp, *r, *s;
    temp = r = s = start;
    printf("row_position: ");
    while(temp != NULL)
    {
         printf("%d ", temp->row_position);
        temp = temp->next;
    }
    printf("\n");
    printf("column_postion: ");
    while(r != NULL)
    {
        printf("%d ", r->column_postion);
        r = r->next;
    }
    printf("\n");
    printf("Value: ");
    …
}
```

List of lists Representation



List of Lists Representation

- Store the indices as sorted.
- How to do addition ?

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

List is an Abstract Data Type

- A class of objects whose logical behavior is defined by a set of values and a set of operations.

- What is an abstract data type (ADT)?
  - It is a data type defined by the user.
  - It is defined by it's behavior (semantics)
  - 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.

Example

Write a C Program to store student course information using structures with Dynamically Memory Allocation.

Input:

Enter number of records: 2
Enter name of the subject and marks respectively:
Programming
22
Enter name of the subject and marks respectively:
Structure
33

Output:

Displaying Information:
Programming 22
Structure 33

Example

Write a C program to create a 2 dimensional array initialized with zero using dynamic memory allocation. Input is number of rows and columns.

Input :
Number of rows: 3
Number of columns: 4

Output :
Print the array.