Chapter 12

Advance Structures

Dr. Niroj Kumar Pani

nirojpani@gmail.com

Department of Computer Science Engineering & Applications
Indira Gandhi Institute of Technology
Sarang, Odisha

Chapter Outline…

- A Brief Introduction to Data Structures
- Sparse Matrixes
  - What is a Sparse Matrix and Where is it Used?
  - Representation of Sparse Matrix
  - Assignments - I
- Linked Lists: An Introduction
  - Limitations of Arrays (The Need for Linked Lists)
  - What is a Linked List?
  - Advantages of Linked Lists Over Arrays
  - Types of Linked Lists

- Single Linked Lists
  - What is a Single Linked List?
  - Representing a Node in ‘C’
  - Important Operations on Single Linked Lists
  - Assignments - II
- Double Linked Lists
  - What is a Double Linked List?
  - Representing a Node in ‘C’
  - Important Operations on Double Linked Lists
  - Assignments - III
- Circular & Header Linked Lists
  - Circular Linked List
  - Header Linked Lists

- **Applications of Linked Lists**
    - **Applications of Linked Lists in Computer Science**

A Brief Introduction to Data Structures

Data Structures: Why & What?

- Structuring - means - organizing (arranging).

  Data structure - means - data organization.

- Why to organize data?
  - One of the important characteristics of data (digital data) is, it need to be:
    - accessed, and
    - processed
  - In order to access and process data efficiently it need to be organized.

- The same set of data may be organized in different ways, depending upon what operations we want to perform upon them. Each of these organizations
  - refers to a different data structure, and
  - supports a different set of operations.

- **[Data Structure - Definition]:** A data structure refers to a logical or mathematical model of
  - a particular organization of data, along with
  - a set of operations that can be performed on this data organization.

Data structure = Organized set of data (values) + set of allowed operations

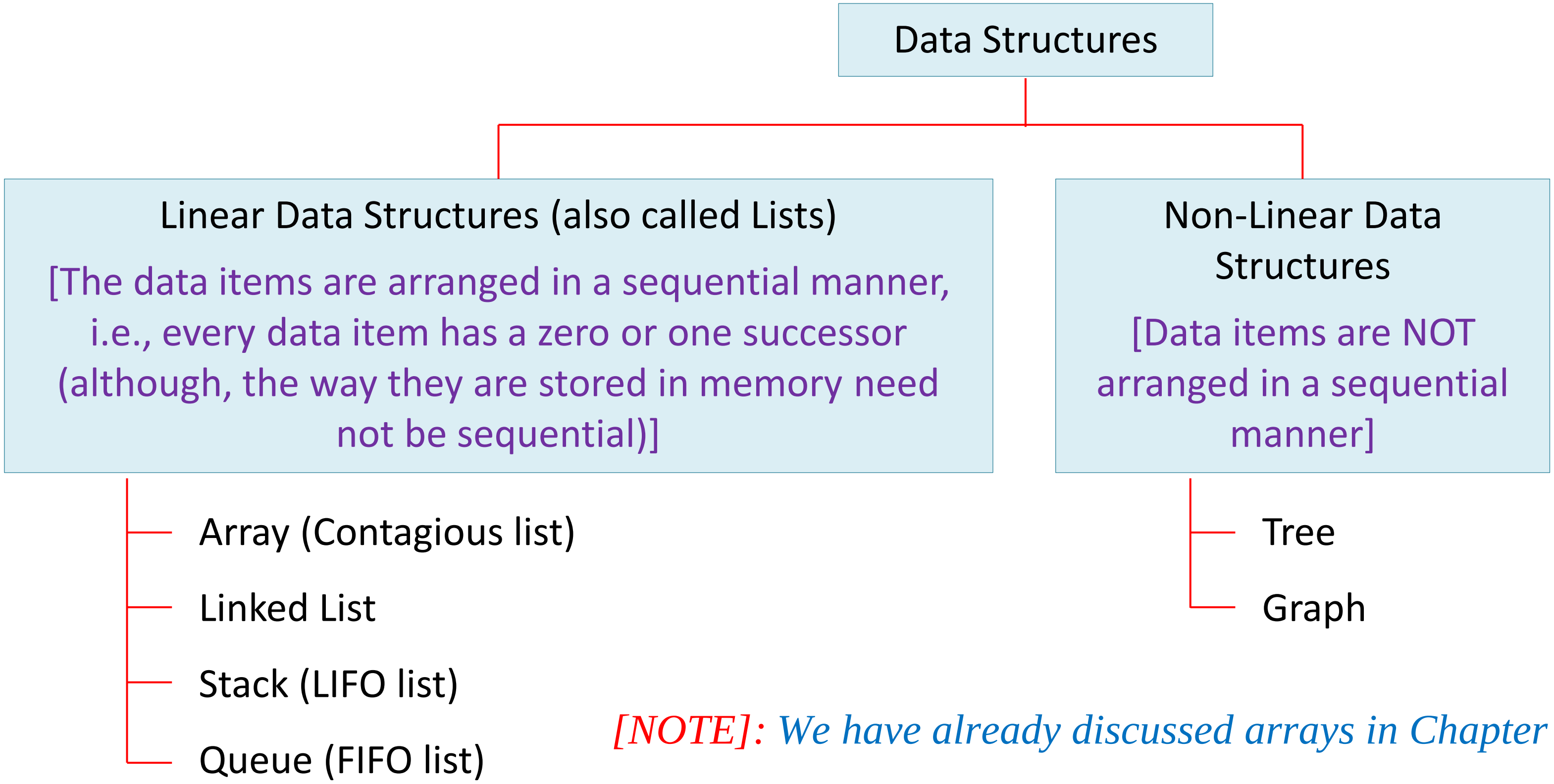
The choice a particular data organization depends upon two considerations:

1. **The set of operations that need to be performed.**
2. **Whether the organization is mirroring the actual relationship of data in the real world or not.**

Classification of Data Structures

Data structures may be classified in a verity of ways.

- **The most common classification**



Data Structures

Linear Data Structures (also called Lists)

[The data items are arranged in a sequential manner, i.e., every data item has a zero or one successor (although, the way they are stored in memory need not be sequential)]

- Array (Contagious list)
- Linked List
- Stack (LIFO list)
- Queue (FIFO list)

Non-Linear Data Structures

[Data items are NOT arranged in a sequential manner]

- Tree
- Graph

[NOTE]: We have already discussed arrays in Chapter 6. In this chapter, we will discuss sparse matrix (a special kind of 2-D array) and linked lists.

- **Other classifications**
  - **Homogeneous Vs. Non-Homogeneous**
    - **If all the elements have the same data type the data structure is called homogeneous (e.g., arrays), otherwise the data structure is called non-homogeneous.**
  - **Static Vs. Dynamic**
    - **Static: Fixed sized (e.g., arrays)**
    - **Dynamic: Variable sized (e.g., linked lists)**

Operations on Data Structures

- Different data structures support different set of operations. Some of the common and important operations are:
  - Creation: A data structure say a tree is created from the given data set.
  - Insertion: Adding a new data item to a data structure.
  - Deletion: Removing a data item from a data structure.
  - Modification: Updating a data item in a data structure.
  - Traversal: Visiting each data item in a data structure exactly once.
  - Searching: Finding the location of a data item in a data structure.
  - Sorting: Arranging the data items in a data structure in some logical order.

Sparse Matrixes

What is a Sparse Matrix and Where is it Used?

- **[Sparse Matrix - Definition]:** A sparse matrix is matrix (2-D array) most of whose elements are zeroes.

  For Example, consider the following matrix

$$A\ [6]\ [6] = \begin{bmatrix} 5 & 10 & 0 & 0 & 2 & 0 \\ 0 & 1 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 \end{bmatrix}$$

In this matrix, out of 36 elements only 7 elements are non-zero. Such a matrix is called a sparse matrix.

- **Use:** Sparse matrixes are commonly used in scientific applications.

Representation of Sparse Matrix

- In the regular representation of spare matrix (the way we have represented the sparse matrix in the pervious slide), there is a wastage of lots of memory space, since most of the elements in the matrix are zeroes.
- In order to avoid this wastage of memory space, sparse matrixes are normally represented (stored) in an alternative form, in which we explicitly store only the non-zero elements of the matrix along with there positions. This representation of is called the 3-tuple form / representation.

The 3-Tuple Representation

- The Technique: In 3-tuple form,
  - Each non-zero element of the matrix is represented by the following 3-tuple: <row-number, column-number, value> (since, these 3-tuples are sufficient to identify each no-zero element in the matrix uniquely),
  - And the matrix is represented as a *list* of these 3-tuples.

■ Example:

Sparse Matrix in Regular Form	The Same Sparse Matrix in 3-Tuple Form
$A\ [4]\ [5] = \begin{bmatrix} 0 & 0 & 0 & 8 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 2 & 0 & 0 & 0 & 5 \end{bmatrix}$	$S\ [6]\ [3] = \begin{array}{c} \begin{array}{ccc} \text{Row} & \text{Col} & \text{Value} \end{array} \\ \begin{bmatrix} 4 & 5 & 5 \\ 0 & 3 & 8 \\ 1 & 0 & 2 \\ 2 & 2 & 3 \\ 3 & 0 & 2 \\ 3 & 4 & 5 \end{bmatrix} \end{array}$

Observations: In the 3-tuple representation,

- The 1st row is reserved to provide the following information about the original sparse matrix (matrix in regular form):
  - The number of rows
  - The number of columns
  - The number of non-zero elements

  e.g., the 1st row in S "$4\ 5\ 5$" indicates that A has 4 rows, 5 columns, and 5 non-zero elements.
- The remaining rows gives information about the individual elements in the original matrix, e.g., the 2nd row in S "$0\ 3\ 8$" indicates that at row number 0 and column number 3 of A the 1st non-zero element is located, which is 8.
- The total number of rows = 1 + No. of non-zero elements in the original matrix.
- Total number of columns = 3 (always).
- Therefore, a lot of memory space is saved.

■ Conversion of Sparse Matrix to 3-Tuple Form (from Regular Representation) - C Program Implementation:

```
/* PR6_7.c: This program converts a 4 × 5 sparse matrix (given in regular form) to 3-tuple form. */

#include <stdio.h>
#include <conio.h>
#define ROW 4
#define COL 5

void ConvertSparse (int a[][COL], int m, int n)
/* Function Definition */
/* 'a' is the original sparse matrix, 'm' is the number of rows, and  'n' is the number of columns. */

{
    int sparse [m*n][3];
    /* 'sparse' is the new sparse matrix in 3-tuple form.
    Initially, we take the number of rows of sparse matrix to be large enough (m*n) */

    int i, j, r=1;
    int nonZeroElementCount=0;
```

[Cont.]

```
/* Count the number of non-zero elements in original sparse matrix */
 for (i=0;i<m;i++)
 {
     for (j=0;j<n;j++)
     {
         if (a[i][j] != 0)
            nonZeroElementCount++;
     }
 }
/*Construct the new sparse matrix in 3-tuple form */
 sparse[0][0] = m;
 sparse[0][1] = n;
 sparse[0][2] = nonZeroElementCount;
 for (i=0;i<m;i++)
 {
     for (j=0;j<n;j++)
     {
         if (a[i][j] != 0)
         {
            sparse[r][0] = i;
            sparse[r][1] = j;
            sparse[r][2] = a[i][j];
            r++;
         }
     }
 }
```

[Cont.]

```
    /* Print the sparse matrix */
    /* The number of rows in sparse = nonZeroElementCount+1 and the number of columns = 3 */
    printf("\n The sparse matrix in 3-tuple form is: \n\n");
    for (i=0;i<nonZeroElementCount+1;i++)
    {
        for (j=0;j<3;j++)
        {
            printf("\t%d", sparse[i][j]);
        }
        printf("\n\n");
    }
 }
/* End of subroutine ConvertSparse */

/* Starting main function. */
 void main()
 {
    int a[ROW][COL];
    int i, j;
```

[Cont.]

```
    /* Enter the elements within the 4X5 sparse matrix */
    printf("Enter the elements within the 4X5 sparse matrix:\n\n ");
    for (i=0;i<ROW;i++)
    {
        for (j=0;j<COL;j++)
        {
            printf("\tEnter the values for a[%d][%d]: ", i, j);
            scanf("%d", &a[i][j]);
        }
    }

    /* View the 4X5 sparse matrix */
    printf("\n\nYour sparse matrix is:\n\n ");
    for (i=0;i<ROW;i++)
    {
        for (j=0;j<COL;j++)
        {
            printf("\t%d",a[i][j]);
        }
        printf("\n\n");
    }

    /* Call the subroutine ConvertSparse */
    ConvertSparse(a,ROW,COL);
    getch();
}  /* End of main */
```

Output

```
Enter the elements within the 4X5 sparse matrix:

        Enter the values for a[0][0]: 4
        Enter the values for a[0][1]: 0
        Enter the values for a[0][2]: 5
        Enter the values for a[0][3]: 0
        Enter the values for a[0][4]: 0
        Enter the values for a[1][0]: 0
        Enter the values for a[1][1]: 0
        Enter the values for a[1][2]: 0
        Enter the values for a[1][3]: 6
        Enter the values for a[1][4]: 0
        Enter the values for a[2][0]: 0
        Enter the values for a[2][1]: 3
        Enter the values for a[2][2]: 0
        Enter the values for a[2][3]: 9
        Enter the values for a[2][4]: 0
        Enter the values for a[3][0]: 0
        Enter the values for a[3][1]: 0
        Enter the values for a[3][2]: 0
        Enter the values for a[3][3]: 0
        Enter the values for a[3][4]: 0
```

[Cont.]

```
Your sparse matrix is:

       4       0       5       0       0

       0       0       0       6       0

       0       3       0       9       0

       0       0       0       0       0

The sparse matrix in 3-tuple form is:

       4       5       5

       0       0       4

       0       2       5

       1       3       6

       2       1       3

       2       3       9
```

Assignments - I

Complete the experiments given in “Lab Manual - Section 14”**.**

Linked Lists: An Introduction

Limitations of Arrays (The Need for Linked Lists)

- Structuring data items as an array (contiguous list) has several advantages:
  - The access time is always constant.
  - Searching is easier.
- However, arrays suffer from following limitations:
  - Arrays are static (the size of an array has to be defined during its creation). This results in two problems.
    1. If the number of elements is less than the defined size, there is a wastage of memory space.
    2. If the number of elements is greater than the defined size, there is a possibility of undesirable result.

    Though, these problems can be handled by declaring the array size at runtime (i.e., by using dynamic memory management technique), it isn't that handy while working with arrays.

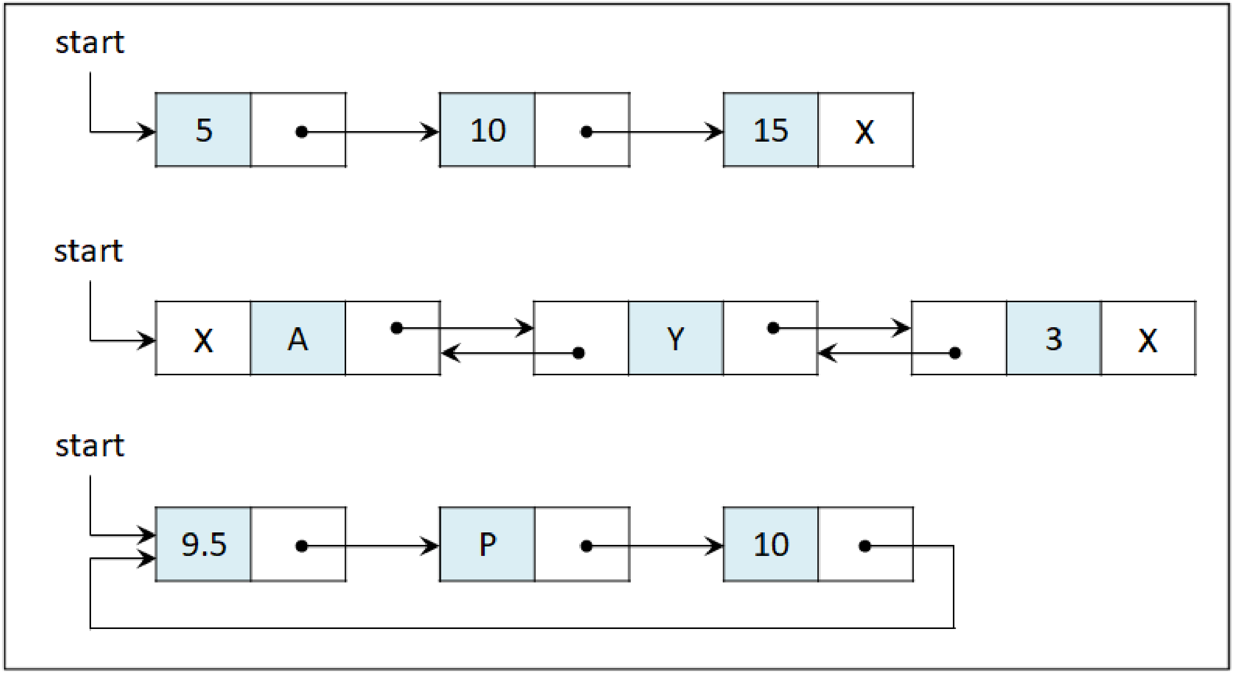
- An array is a contagious list (array elements are physically stored in contagious memory locations). This results in two problems.
    1. An array can't be created if we don't have enough *contiguous* space in memory.
    2. Insertion and deletion operations are difficult to perform.

- The afore-mentioned limitations can be overcome if:
    - We allocate memory space to each item in a list dynamically (i.e., at runtime as per requirement).
    - We use a linked representation for the list of items (i.e., we establish a link from the 1st item in the list to the 2nd , from the 2nd item to the 3rd and so on.) instead of string them contiguously (as in the case of arrays). So, the list is NOT physically contiguous but logically continuous.

Both these goals are fulfilled by the linked lists.

What is a Linked List?

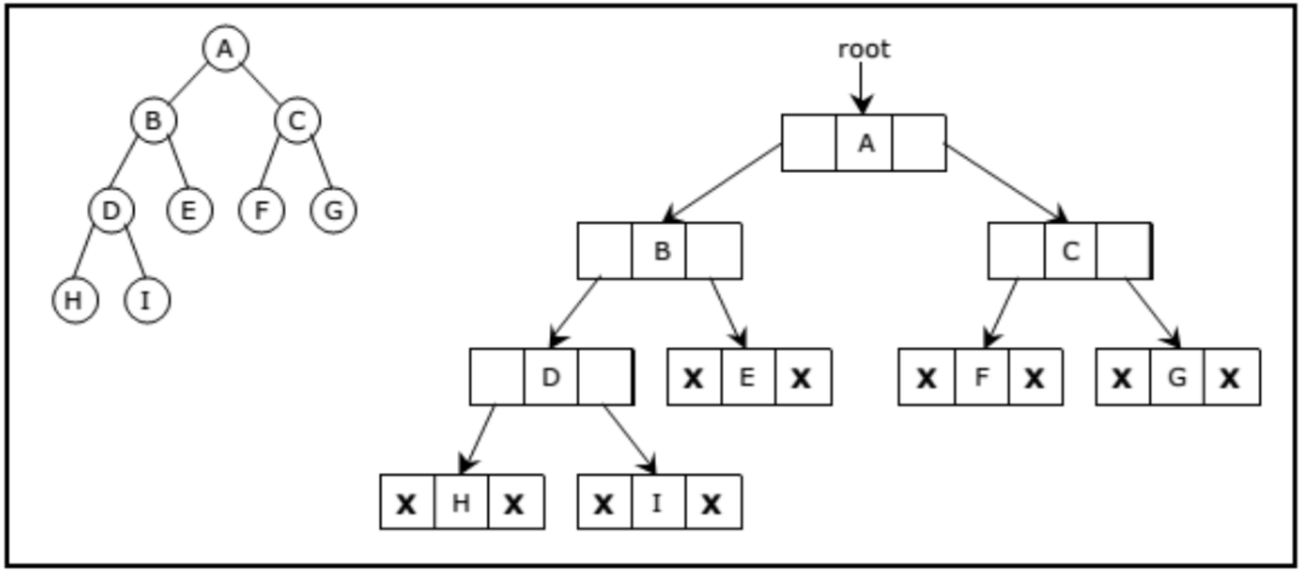
- **[Definition]: A linked list is a dynamic, ordered collection of “nodes” where each node consists of the following parts (fields):**
  - **A data / information field that contains the information of the node.**
  - **One or more pointer / address / link fields that points to** (contains the address of) **the adjacent node(s) in the list.**



[Fig. 12.1: Linked List Examples]

- **[NOTES]:**
  - A linked list may be homogeneous or non-homogeneous. In this chapter, we primarily use homogeneous (integer) lists.
  - Though we have classified linked list as a linear data structure it can be used to represent non-linear data structures such as trees and graphs.

Example:



[Fig. 12.2: Linked List Representation of a Binary Tree]

Self-Referential Structures:

- Self-referential structures are those structures that have one or more members which point to the same type of structure.
- A linked list is perfect example of a self-referential structure, because in it the pointer field in one node points to another node.

```
struct Node
{
    int info;
    struct Node *next;
};
```

Advantages of Linked Lists Over Arrays

- Linked lists offer the following advantages over arrays:
  1. Linked list is dynamic (its size can be increased or decreased as per requirement). So, no memory space is wasted.
  2. Insertion and deletion operations can be performed quickly.

Types of Linked Lists

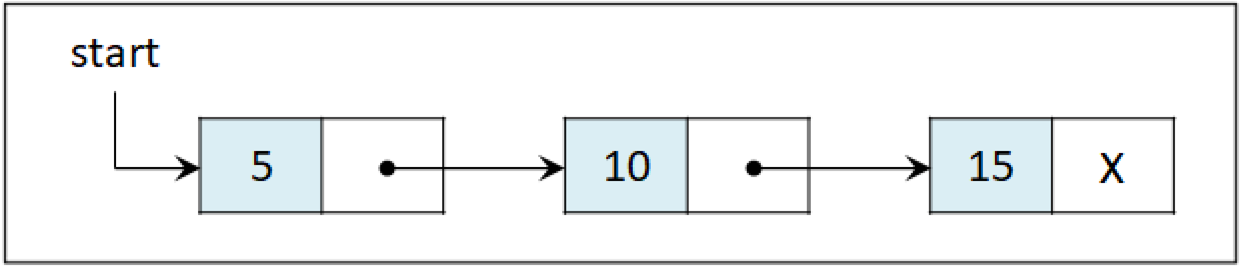
- There are different types of lined lists. We shall discuss the following types:
    1. Single linked list / one-way list
    2. Double linked list / two-way list
    3. Circular linked list
    4. Header linked list

Single Linked Lists

What is a Single Linked List?

- [Definition]: A single linked list (one-way list) is a dynamic, ordered collection of nodes where each node consists of two parts (fields):
  - A data / info field (called info) that contains the information of the node.
  - A pointer / address / link field (called next) that points to (contains the address of) the next node in the list.

  In addition to this,
  - A special pointer called "start (or, list)" points to the first node of the list.
  - The "next" field of the last node contains NULL (X) which indicates the end of the list.



[Fig. 12.3: A Single Linked List]

Representing a Node in ‘C’

- **In a single linked list, a node contains of two parts:**
  - **An** info **field.**
  - **A** next **field (that points to the next node).**

So, a node of a single linked list can be represented by the following C structure:

```
struct Node
{
    int info;
    struct Node *next;
};

typedef struct Node SNode;
```

[NOTE]: “How to create a node” is discussed when we discuss “how to create a single linked list” in the next section.

Important Operations on Single Linked Lists

- We discuss the following important operations on a single linked list:
  1. Creating a single linked list.
  2. Traversing a single linked list (visiting each node exactly once).
  3. Inserting a node in a singe linked list.
     - Inserting at the beginning
     - Inserting at the end
     - Inserting at an intermediate position after a given node
  4. Deleting a node from a single linked list.
     - Deleting the first node
     - Deleting the last node
     - Deleting an intermediate node after a given node
  5. Searching an element (a value) in a single linked list.
  6. Sorting a single linked list (sorting the node values).

Creating a SLL

- **Technique: It is a two-step process.**

  ① **Create the first node** (since every single linked list contains at least one node we must create at least one node, i.e., the first node in the list).

  ② **Create and insert subsequent nodes on demand** (till the user wants) **to the end of the linked list**.

C Function:

```
/* Function to create a single linked list. */
SNode* CreateSLL()
{
    SNode *start;
    int item;
    char ch;

    /* Step 1: Creating the 1st node of the list. */
    start = (SNode *)malloc(sizeof(SNode));
    if (start == NULL)
    {
        printf("\nOut of memory space");
        return;
    }

    printf("Enter the value (info field) of the 1st node: ");
    scanf("%d", &start->info);
    start->next = NULL;
    printf("1st node with info field %d created.", start->info);
```

[Cont.]

```
    /* Step 2: Creating and inserting subsequent nodes on demand to the end of the list. */
    printf("\n\nDo you want to add more nodes (press 'Y' to continue,
    any other key to quit)?: ");
    ch = getche();

    while(ch=='Y'||ch=='y')
    {
        printf("\nEnter the value (info field) of the new node: ");
        scanf("%d", &item);
        InsertAtEndInSLL(start, item);
        printf("\nDo you want to add more nodes (press 'Y' to
        continue, any other key to quit)?: ");
        ch = getche();
    }
    return start;
}
/* End of function CreateSLL() */
```

[Cont.]

```
/* Function to insert a node at the end of a single linked list. This function is called by CreateSLL() */
void InsertAtEndInSLL (SNode *start, int item)
{
    SNode *newNode, *ptr;

    /* Step 1: Move a pointer "ptr" to the last node of the linked list. */
    ptr = start;
    while(ptr->next != NULL)
        ptr = ptr->next;

    /* Step 2: Create a new node. Store "item" in its "info" field and "NULL" in its "next" field. */
    newNode = (SNode *)malloc(sizeof(SNode));
    if (newNode == NULL)
    {
        printf("\nOut of memory space");
        return;
    }
    newNode->info = item;
    newNode->next = NULL;

    /* Step 3: Point the "next" part of the last node (pointer "ptr") to the new node */
    ptr->next = newNode;

    printf("A new node with info field %d added
    to the linked list.\n ", newNode->info);
}
/* End of function InsertAtEndInSLL() * /
```

Traversing a SLL

- **C Function:**

```c
/* Function to traverse a single linked list */
void TraverseSLL(SNode *start)
{
    SNode *ptr;
    if (start == NULL)
    {
        printf("\n\nEmpty List");
    }
    else
    {
        ptr = start;
        printf("\n\nThe single linked list is: ");
        while (ptr != NULL)
        {
            printf("%d -> ", ptr->info);
            ptr = ptr->next;
        }
        printf("\b\b\b    ");
        printf("\n\n");
    }
}
```

■ A Complete C Program that Creates & Traverses a Single Linked List:

```
/* PR12_1.c: A program to create a single linked list and then traverse it. */

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

/* Single linked list node definition */
struct Node
{
    int info;
    struct Node *next;
};
typedef struct Node SNode;
```

[Cont.]

```
/* Prototype declaration for the function "InsertAtEndInSLL()" */
void InsertAtEndInSLL (SNode *start, int item);

/* Function to create a single linked list */
SNode* CreateSLL()
{
    SNode *start;
    int item;
    char ch;

    /* Step 1: Creating the 1st node of the list. */
    start = (SNode *)malloc(sizeof(SNode));
    if (start == NULL)
    {
        printf("\nOut of memory space");
        return;
    }

    printf("Enter the value (info field) of the 1st node: ");
    scanf("%d", &start->info);
    start->next = NULL;
    printf("1st node with info field %d created.", start->info);
```

[Cont.]

```
    /* Step 2: Creating and inserting subsequent nodes on demand to the end of the linked list. */
    printf("\n\nDo you want to add more nodes (press 'Y' to continue,
    any other key to quit)?: ");
    ch = getche();

    while(ch=='Y'||ch=='y')
    {
        printf("\nEnter the value (info field) of the new node: ");
        scanf("%d", &item);
        InsertAtEndInSLL(start, item);
        printf("\nDo you want to add more nodes (press 'Y' to
        continue, any other key to quit)?: ");
        ch = getche();
    }
    return start;
}
/* End of function CreateSLL() */
```

[Cont.]

```
/* Function to insert a node at the end of a single linked list. This function is called by CreateSLL() */
void InsertAtEndInSLL (SNode *start, int item)
{
    SNode *newNode, *ptr;

    /* Step 1: Move a pointer “ptr” to the last node of the linked list. */
    ptr = start;
    while(ptr->next != NULL)
        ptr = ptr->next;

    /* Step 2: Create a new node. Store “item” in its “info” field and “NULL” in its “next” field. */
    newNode = (SNode *)malloc(sizeof(SNode));
    if (newNode == NULL)
    {
        printf("\nOut of memory space");
        return;
    }
    newNode->info = item;
    newNode->next = NULL;

    /* Step 3: Point the “next” part of the last node (pointer “ptr”) to the new node */
    ptr->next = newNode;

    printf("A new node with info field %d added
    to the linked list.\n ", newNode->info);
}
/* End of function InsertAtEndInSLL() */
```

[Cont.]

```
/* Function to traverse a single linked list */
void TraverseSLL(SNode *start)
{
    SNode *ptr;
    if (start == NULL)
    {
        printf("\n\nEmpty List");
    }
    else
    {
        ptr = start;
        printf("\n\nThe single linked list is: ");
        while (ptr != NULL)
        {
            printf("%d -> ", ptr->info);
            ptr = ptr->next;
        }
        printf("\b\b\b    ");
        printf("\n\n");
    }
}
```

[Cont.]

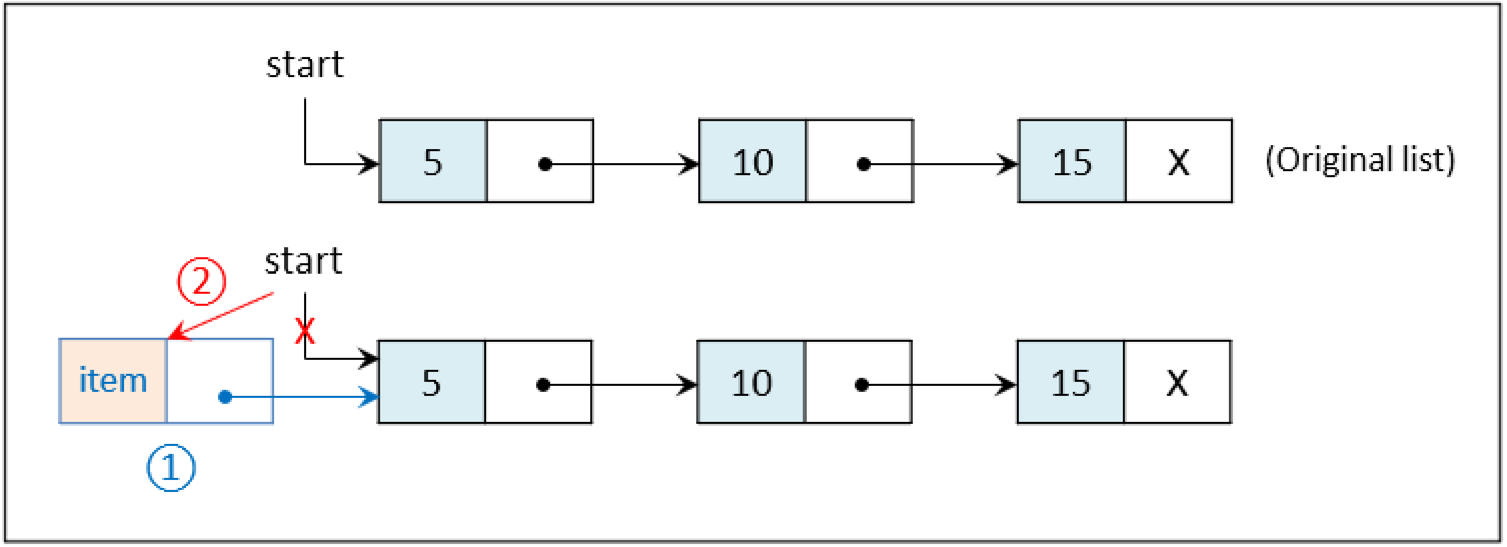
```
/* Start of main() */
void main()
{
    SNode *start;
    start = CreateSLL();
    TraverseSLL(start);
}
/* End of main() */
```

Output

```
Enter the value (info field) of the 1st node: 5
1st node with info field 5 created.

Do you want to add more nodes (press ‘Y' to continue, any other key
to quit)?: y
Enter the value (info field) of the new node: 3
A new node with info field 3 added to the linked list.

Do you want to add more nodes (press ‘Y' to continue, any other key
to quit)?: y
Enter the value (info field) of the new node: 9
A new node with info field 9 added to the linked list.

Do you want to add more nodes (press ‘Y' to continue, any other key
to quit)?: n

The single linked list is: 5 -> 3 -> 9
```

Inserting a Node at the Beginning of a SLL

- **Technique:**



① Create a new node. Store “item” in its “info” field and point its “next” field to “start”.

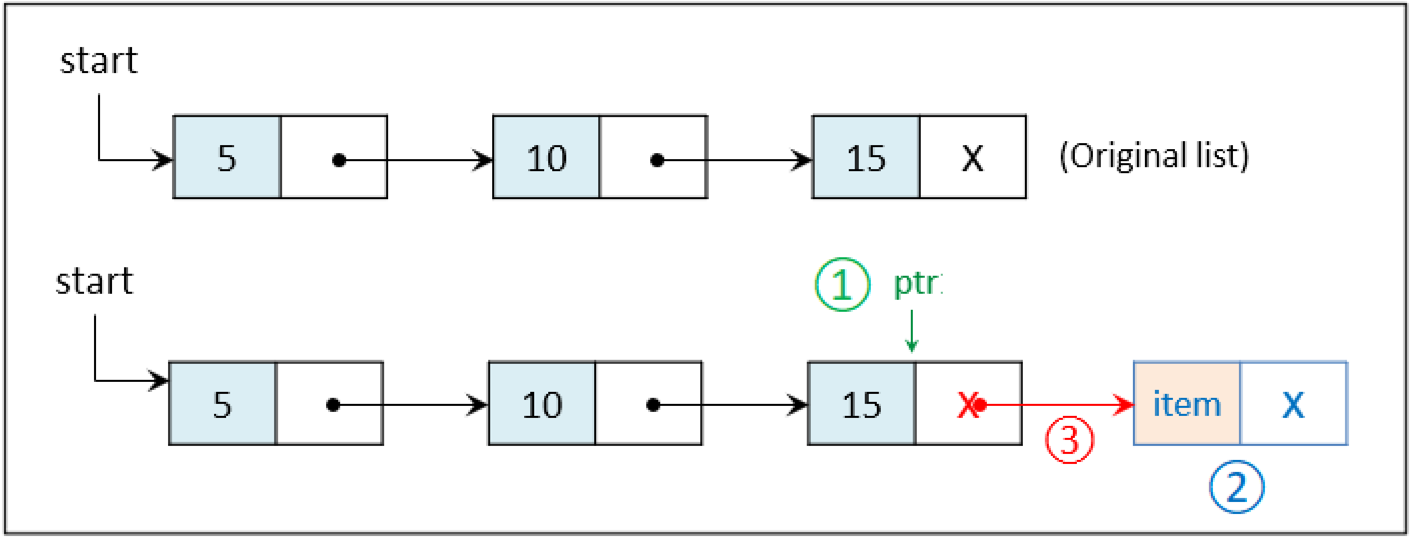
② Point “start” to the new node.

- **C Function:**

```c
/* Function to insert a node at the beginning of a single linked list */
SNode* InsertAtBeginingInSLL(SNode* start, int item)
{
    SNode *newNode;

    /* Step 1: Create a new node. Store “item” in its “info” field and point its “next” field to “start”. */
    newNode = (SNode *)malloc(sizeof(SNode));
    if (newNode == NULL)
    {
        printf("\nOut of memory space");
        return;
    }

    newNode->info = item;
    newNode->next = start;

    start = newNode;  /* Step 2: Point “start” to the new node. */
    return start;
}
```

Inserting a Node at the End a SLL

- **Technique:**



① Move a pointer "ptr" to the last node of the linked list.

② Create a new node. Store "item" in its "info" field and "NULL" in its "next" field.

③ Point the "next" part of the last node (the pointer "ptr") to the new node.

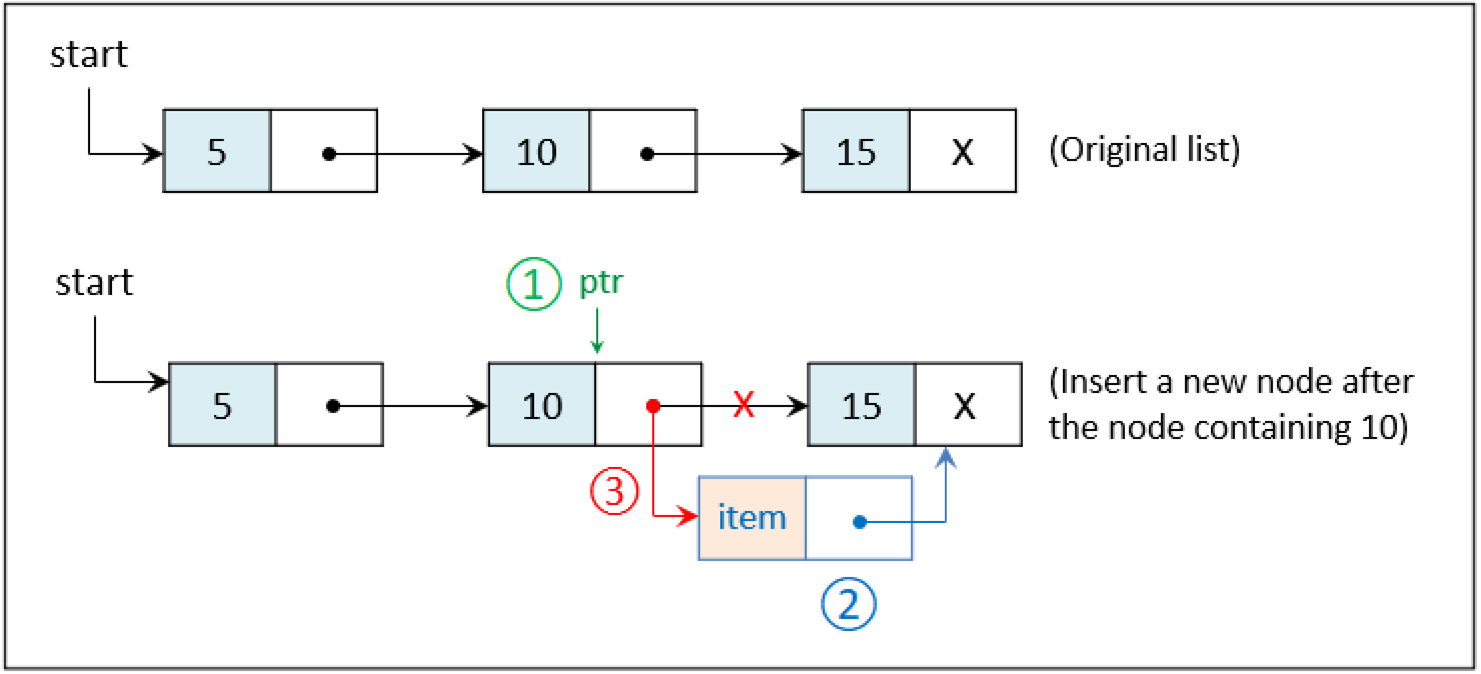
[NOTE]: We have already seen "how to insert a node at the end of a single linked list" (through the function "`InsertAtEndInSLL()`") in "creating a single linked list".

- **C Function:**

```
/* Function to insert a node at the end of a single linked list. */
void InsertAtEndInSLL (SNode *start, int item)
{
    SNode *newNode, *ptr;

    /* Step 1: Move a pointer “ptr” to the last node of the linked list. */
    ptr = start;
    while(ptr->next != NULL)
        ptr = ptr->next;

    /* Step 2: Create a new node. Store “item” in its “info” field and “NULL” in its “next” field. */
    newNode = (SNode *)malloc(sizeof(SNode));
    if (newNode == NULL)
    {
        printf("\nOut of memory space");
        return;
    }
    newNode->info = item;
    newNode->next = NULL;

    /* Step 3: Point the “next” part of the last node (pointer “ptr”) to the new node */
    ptr->next = newNode;

    printf("A new node with info field %d added
    to the linked list.\n ", newNode->info);
}
```

Inserting a Node at an Intermediate Position after a Given Node in a SLL

■ Technique:



① Move a pointer "ptr" to the node containing "data" (i.e., the node after which a new node is to be inserted).

② Create a new node. Store "item" in its "info" field and point its "next" field to "ptr->next".

③ Point "ptr->next" to the new node.

- **C Function:**

```
/* Function to insert a node at an intermediate position after a given node (a node containing "data" in
its info field) in a single linked list. */
void InsertAfterNodeInSLL (SNode *start, int data, int item)
{
    SNode *newNode, *ptr;

    /* Step 1: Move a pointer "ptr" to the node containing "data". */
    ptr = start;
    while((ptr->info != data)&&(ptr->next != NULL))
    {
        ptr = ptr->next;
    }
    if(ptr->next == NULL)
    {
        printf("\n%d is not found in the linked list.", data);
        exot(0);
    }
```

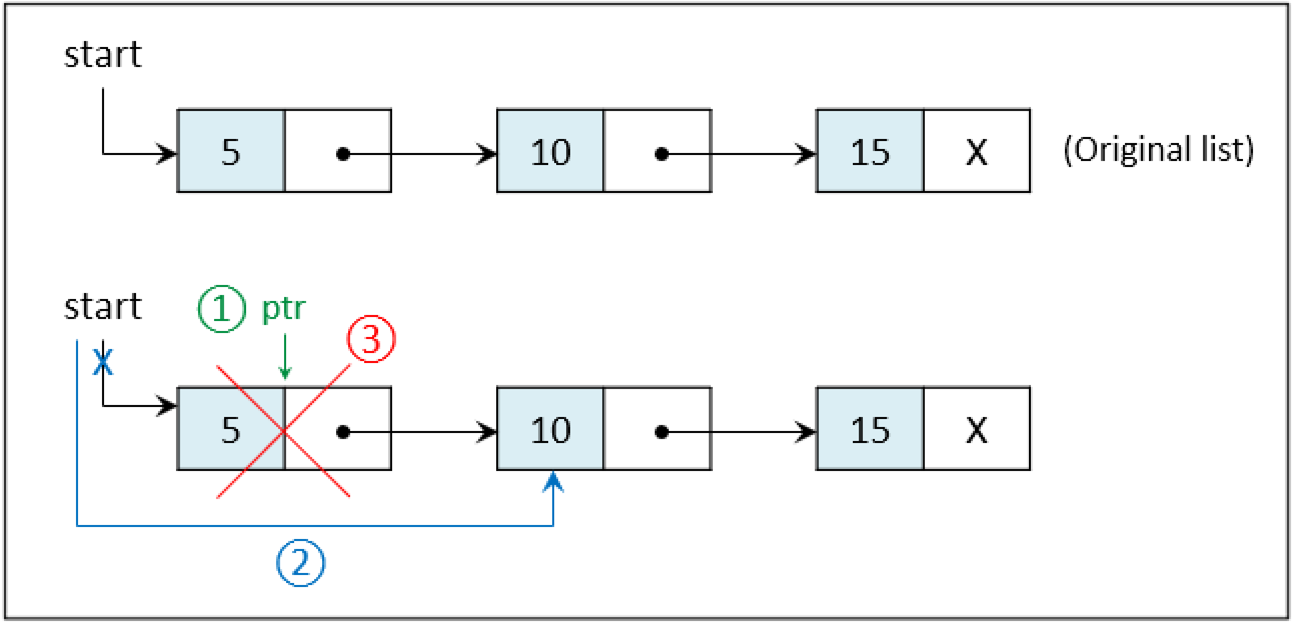
[Cont.]

```
    /* Step 2: Create a new node.
    Store “item” in its “info” field and point its “next” field to “ptr->next”. */
    newNode = (SNode *)malloc(sizeof(SNode));
    if (newNode == NULL)
    {
        printf("\nOut of memory space");
        return;
    }
    newNode->info = item;
    newNode->next = ptr->next;

    /* Step 3: Point “ptr->next” to the new node. */
    ptr->next = newNode;

    printf("A new node with info field %d added after the node
    containing %d in the linked list.\n ", newNode->info, ptr->info);
}
```

Deleting the First Node from a SLL

- [NOTE] - Considerations while deleting the first and the last nodes from a single linked list:
  - If the linked list contains just one node, then after deletion of the node, the linked list shall be empty. So, in such case, after deletion of the node, "start" should point to NULL.
  - When the node is deleted its memory space must be freed (released).

■ Technique (Deleting the First Node):



① Point a pointer “ptr” to the first node (i.e., to “start”).

② Point “start” to “ptr->next”.

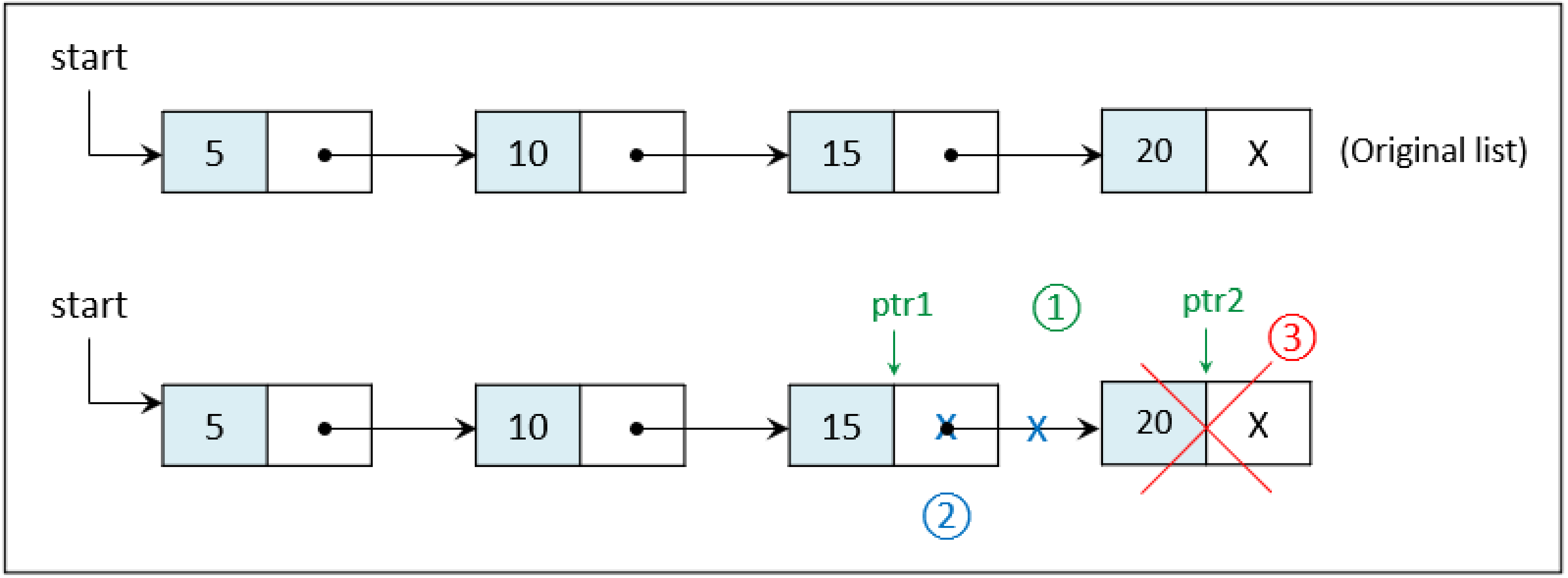
③ Release the node pointed out by “ptr”.

C Function:

```
/* Function to delete the first node from a single linked list. */
void DeleteFirstNodeInSLL (SNode *start)
{
    SNode *ptr;

    /* Checking for a single node. */
    if(start->next == NULL)
    {
        free(start);
        start = NULL;
        return start;
    }

    /* More than one node. */
    ptr = start;  /* Step 1: Point a pointer “ptr” to the first node (i.e., to “start”). */
    start = ptr->next;  /* Step 2: Point “start” to “ptr->next”. */
    free(ptr);  /* Step 3: Release the node pointed out by “ptr”. */

    return start;
}
```

Deleting the Last Node from a SLL

- **Technique:**



① Move a pointer "ptr1" to the last but one node, and another pointer "ptr2" to the last node.

② Store "NULL" in "ptr1->next".

③ Release the node pointed out by "ptr2".

- **C Function:**

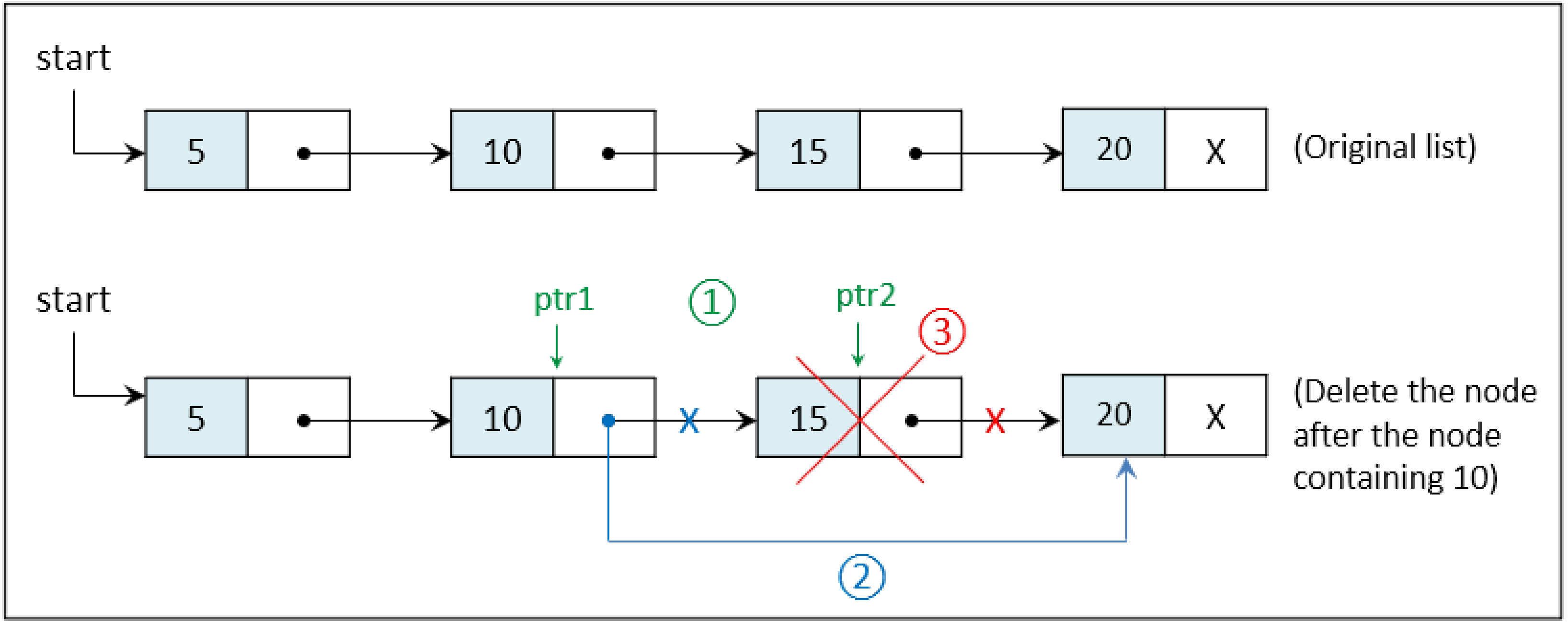
```
/* Function to delete the last node from a single linked list. */
void DeleteLastNodeInSLL (SNode *start)
{
    SNode *ptr1, *ptr2;

    /* Checking for a single node. */
    if(start->next == NULL)
    {
        free(start);
        start = NULL;
        return start;
    }
```

[Cont.]

```
    /* More than one node. */

    /* Step 1: Move a pointer “ptr1” to the last but one node,
    and another pointer “ptr2” to the last node. */
    ptr1 = start;
    ptr2 = start->next;
    while((ptr2->next) != NULL)
    {
        ptr2 = ptr2->next;
        ptr1 = ptr1->next;
    }

    ptr1->next = NULL;  /* Step 2: Store “NULL” in “ptr1->next”. */

    free(ptr2);  /* Step 3: Release the node pointed out by “ptr2”. */

    return start;
}
```

Deleting an Intermediate Node after a Given Node in a SLL

- **Technique:**



① Move a pointer "ptr1" to the node containing "data" (i.e., whose next node is to be deleted), and another pointer "ptr2" to the node which is to be deleted (i.e., ptr1->next).

② Point "ptr1->next" to "ptr2->next".

③ Release the node pointed out by "ptr2".

C Function:

```c
/* Function to delete an intermediate node after a given node (a node that contains “data” in its info
field) in a single linked list. */
void DeleteAfterNodeInSLL (SNode *start, int data)
{
    SNode *ptr1, *ptr2;

    /* Step 1: Move a pointer “ptr1” to the node containing “data” (whose next node is to be deleted),
    and another pointer “ptr2” to the node which is to be deleted (i.e., ptr1->next). */
    ptr1 = start;
    while((ptr1->info != data)&&(ptr1->next != NULL))
    {
        ptr1 = ptr1->next;
    }
    if(ptr1->next == NULL)
    {
        printf("\n%d is not found in the linked list.", data);
        exit(0);
    }
    ptr2 = ptr1->next;

    ptr1->next = ptr2->next;  /* Step 2: Point “ptr1->next” to “ptr2->next”. */

    free(ptr2);  /* Step 3: Release the node pointed out by “ptr2”. */
}
```

Searching an Element (a Value) in a SLL

- **Technique:**

  Move a pointer “ptr” from “start” to the last node till the “value” is not found in the “info” fields of one of the nodes. If the “value” is found, display the “node number”, otherwise display “value not found”.

- **C Function:**

```
/* Function to search an item (a value) in a single linked list. */
void SearchInSLL (SNode *start, int item)
{
    SNode *ptr;
    int count = 1;

    ptr = start;
    while(ptr != NULL)
    {
        if(ptr->info == item)
        {
            printf("\n%d is found at node %d.", item, count);
            exit(0);
        }
        count++;
        ptr = ptr->next;
    }
    printf("\n%d is not found in the linked list.", item);
}
```

Sorting a SLL (Sorting the Node Values)

- **C Function:**

```
/* Function to sort the node values in a single linked list. */
void SortSLL (SNode *start)
{
    SNode *ptr1, *ptr2;
    int temp;

    for(ptr1 = start; (ptr1->next != NULL); ptr1 = ptr1->next)
    {
        for(ptr2 = ptr1->next; ptr2 != NULL; ptr2 = ptr2->next)
        {
            if(prt1->info > ptr2->info)
            {
                temp = ptr1->info;
                ptr1->info = ptr2->info;
                ptr2->info = temp;
            }
        }
    }
}
```

Assignments - II

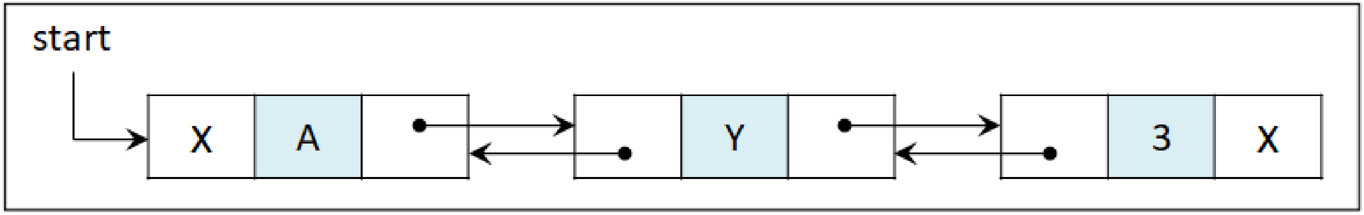
Complete the experiments given in “Lab Manual - Section 15”**.**

Double Linked Lists

What is a Double Linked List?

- [Definition]: A double linked list (two-way list) is a dynamic, ordered collection of nodes where each node consists of three parts (fields):
  - A data / info field (called info) that contains the information of the node.
  - A pointer / address / link field (called prev) that points to (contains the address of) the previous node in the list.
  - A pointer / address / link field (called next) that points to (contains the address of) the next node in the list.

  In addition to this,
  - A special pointer called "start (or, list)" points to the first node of the list.
  - The "prev" field of the first node and "next" field of the last node contains NULL (X).



[Fig. 12.3: A Double Linked List]

Double Linked Lists Vs. Single Linked Lists

- Double linked lists have their own advantages and disadvantages when compared to the single linked list:
    - Double linked lists offer two-way movement. So, some operations like searching becomes twice faster in comparison to single linked lists.
    - However, double linked lists need more space (because of the extra pointer field) in comparison to single linked lists.

Representing a Node in ‘C’

- **Since, in a double linked list, a node contains of three parts:**
  - **A** prev **field (that points to the previous node).**
  - **An** info **field.**
  - **A** next **field (that points to the next node).**

A node of a double linked list can be represented by the following C structure:

```
struct Node
{
    int info;
    struct Node *prev;
    struct Node *next;
};

typedef struct Node DNode;
```

[NOTE]: “How to create a node” is discussed when we discuss “how to create a double linked list” in the next section.

Important Operations on Double Linked Lists

- We discuss the following important operations on a double linked list:
  1. Creating a double linked list.
  2. Traversing a double linked list (visiting each node exactly once).
  3. Inserting a node in a double linked list.
     - Inserting at the beginning
     - Inserting at the end
     - Inserting at an intermediate position after a given node
  4. Deleting a node from a double linked list.
     - Deleting the first node
     - Deleting the last node
     - Deleting an intermediate node after a given node
  5. Searching an element (a value) in a double linked list.

Creating a DLL

- **Technique: The process is same as creating a single linked list. It’s a two-step process.**

  ① **Create the first node** (since every double linked list contains at least one node we must create at least one node, i.e., the first node in the list).

  ② **Create and insert subsequent nodes on demand** (till the user wants) **to the end of the linked list**.

C Function:

```
/* Function to create a duoble linked list. */
DNode* CreateDLL()
{
    DNode *start;
    int item;
    char ch;

    /* Step1: Creating the 1st node of the list. */
    start = (DNode *)malloc(sizeof(DNode));
    if (start == NULL)
    {
        printf("\nOut of memory space");
        return;
    }

    printf("Enter the value (info field) of the 1st node: ");
    scanf("%d", &start->info);
    start->prev = start->next = NULL;
    printf("1st node with info field %d created.", start->info);
```

[Cont.]

```
    /* Step 2: Creating and inserting subsequent nodes on demand to the end of the list. */
    printf("\n\nDo you want to add more nodes (press 'Y' to continue,
    any other key to quit)?: ");
    ch = getche();

    while(ch=='Y'||ch=='y')
    {
        printf("\nEnter the value (info field) of the new node: ");
        scanf("%d", &item);
        InsertAtEndInDLL(start, item);
        printf("\nDo you want to add more nodes (press 'Y' to
        continue, any other key to quit)?: ");
        ch = getche();
    }
    return start;
}
/* End of function CreateDLL()*/
```

[Cont.]

```
/* Function to insert a node at the end of a double linked list. This function is called by CreateDLL() */
void InsertAtEndInDLL (DNode *start, int item)
{
    DNode *newNode, *ptr;

    /* Step 1: Move a pointer "ptr" to the last node of the linked list. */
    ptr = start;
    while(ptr->next != NULL)
        ptr = ptr->next;

    /* Step 2: Create a new node. Store "item" in its "info" field, "ptr" in its "prev" field,
    and "NULL" in its "next" field. */
    newNode = (DNode *)malloc(sizeof(DNode));
    if (newNode == NULL)
    {
        printf("\nOut of memory space");
        return;
    }
    newNode->info = item;
    newNode->prev = ptr;
    newNode->next = NULL;

    ptr->next = newNode; /* Step 3: Point the "next" part of the last node to the new node */

    printf("A new node with info field %d added
    to the linked list.\n ", newNode->info);
}
/* End of function InsertAtEndInDLL() */
```

Traversing a DLL

- **C Function:**

```
/* Function to traverse a double linked list */
void TraverseDLL(DNode *start)
{
    DNode *ptr;
    if (start == NULL)
    {
        printf("\n\nEmpty List");
    }
    else
    {
        ptr = start;
        printf("\n\nThe double linked list is: ");
        while (ptr != NULL)
        {
            printf("%d <-> ", ptr->info);
            ptr = ptr->next;
        }
        printf("\b\b\b\b    ");
        printf("\n\n");
    }
}
```

A Complete C Program that Creates & Traverses a Double Linked List:

```
/* PR12_2.c: A program to create a double linked list and then traverse it. */

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

/* Double linked list node definition */
struct Node
{
    int info;
    struct Node *prev;
    struct Node *next;
};
typedef struct Node DNode;
```

[Cont.]

```
 /* Prototype declaration for the function “InsertAtEndInDLL()” */
 void InsertAtEndInDLL (DNode *start, int item);

/* Function to create a double linked list */
DNode* CreateDLL()
{
    DNode *start;
    int item;
    char ch;

    /* Step 1: Creating the 1st node of the list. */
    start = (DNode *)malloc(sizeof(DNode));
    if (start == NULL)
    {
        printf("\nOut of memory space");
        return;
    }

    printf("Enter the value (info field) of the 1st node: ");
    scanf("%d", &start->info);
    start->prev = start->next = NULL;
    printf("1st node with info field %d created.", start->info);
```

[Cont.]

```
    /* Step 2: Creating and inserting subsequent nodes on demand to the end of the list. */
    printf("\n\nDo you want to add more nodes (press 'Y' to continue,
    any other key to quit)?: ");
    ch = getche();

    while(ch=='Y'||ch=='y')
    {
        printf("\nEnter the value (info field) of the new node: ");
        scanf("%d", &item);
        InsertAtEndInDLL(start, item);
        printf("\nDo you want to add more nodes (press 'Y' to
        continue, any other key to quit)?: ");
        ch = getche();
    }
    return start;
}
/* End of function CreateDLL() */
```

[Cont.]

```
/* Function to insert a node at the end of a double linked list. This function is called by CreateDLL() */
void InsertAtEndInDLL (DNode *start, int item)
{
    DNode *newNode, *ptr;

    /* Step 1: Move a pointer "ptr" to the last node of the linked list. */
    ptr = start;
    while(ptr->next != NULL)
        ptr = ptr->next;

    /* Step 2: Create a new node. Store "item" in its "info" field, "ptr" in its "prev" field,
    and "NULL" in its "next" field. */
    newNode = (DNode *)malloc(sizeof(DNode));
    if (newNode == NULL)
    {
        printf("\nOut of memory space");
        return;
    }
    newNode->info = item;
    newNode->prev = ptr;
    newNode->next = NULL;

    ptr->next = newNode; /* Step 3: Point the "next" part of the last node to the new node */

    printf("A new node with info field %d added
    to the linked list.\n ", newNode->info);
}
/* End of function InsertAtEndInDLL() */
```

[Cont.]

```c
/* Function to traverse a double linked list */
void TraverseDLL(DNode *start)
{
    DNode *ptr;
    if (start == NULL)
    {
        printf("\n\nEmpty List");
    }
    else
    {
        ptr = start;
        printf("\n\nThe double linked list is: ");
        while (ptr != NULL)
        {
            printf("%d <-> ", ptr->info);
            ptr = ptr->next;
        }
        printf("\b\b\b\b    ");
        printf("\n\n");
    }
}
```

[Cont.]

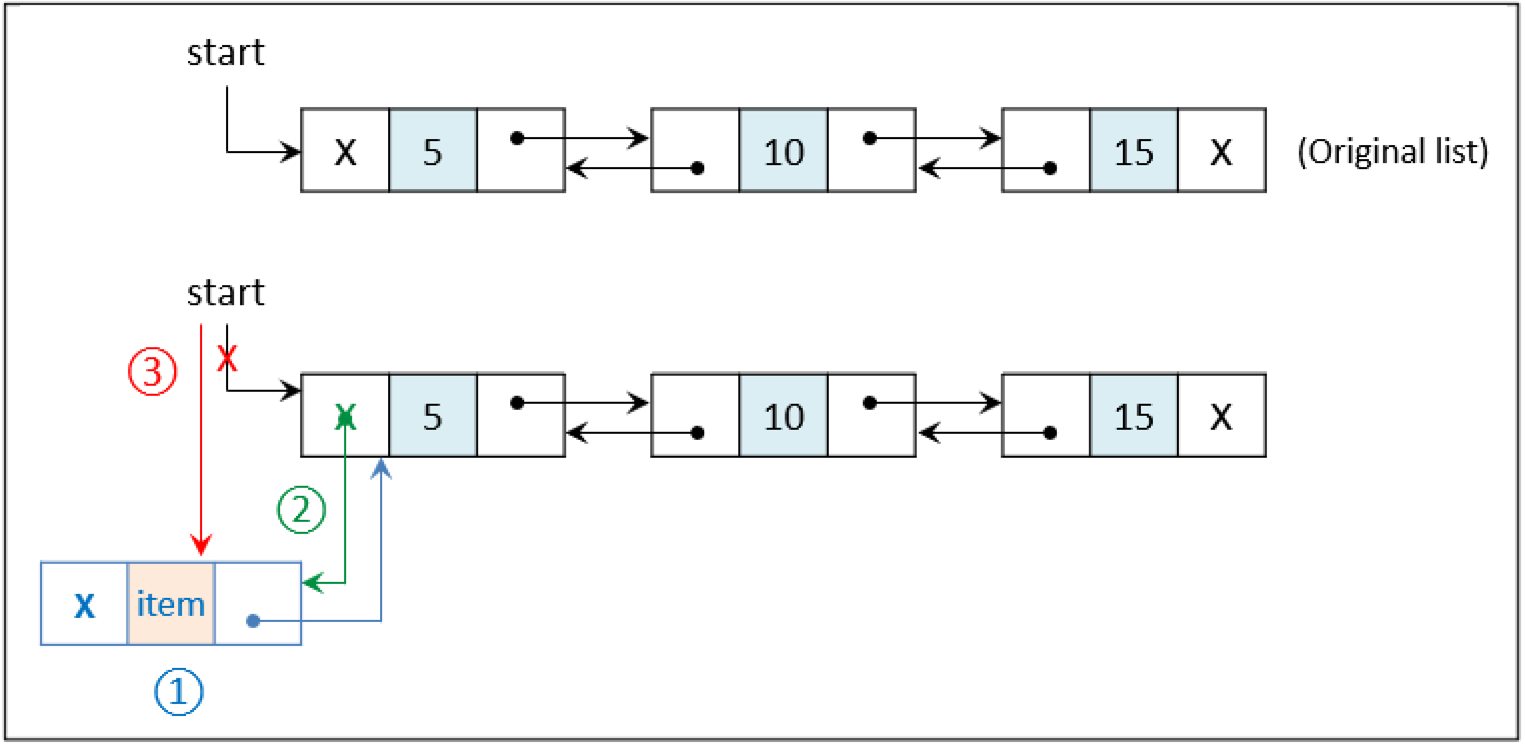
```
/* Start of main() */
void main()
{
    DNode *start;
    start = CreateDLL();
    TraverseDLL(start);
}
/* End of main() */
```

Output

```
Enter the value (info field) of the 1st node: 5
1st node with info field 5 created.

Do you want to add more nodes (press ‘Y' to continue, any other key
to quit)?: y
Enter the value (info field) of the new node: 3
A new node with info field 3 added to the linked list.

Do you want to add more nodes (press ‘Y' to continue, any other key
to quit)?: y
Enter the value (info field) of the new node: 9
A new node with info field 9 added to the linked list.

Do you want to add more nodes (press ‘Y' to continue, any other key
to quit)?: n

The double linked list is: 5 <-> 3 <-> 9
```

Inserting a Node at the Beginning of a DLL

- **Technique:**



① Create a new node. Store “item” in its “info” field, “NULL” in its “prev” field, and point its “next” field to “start”.

② Point “start->prev” to the new node.

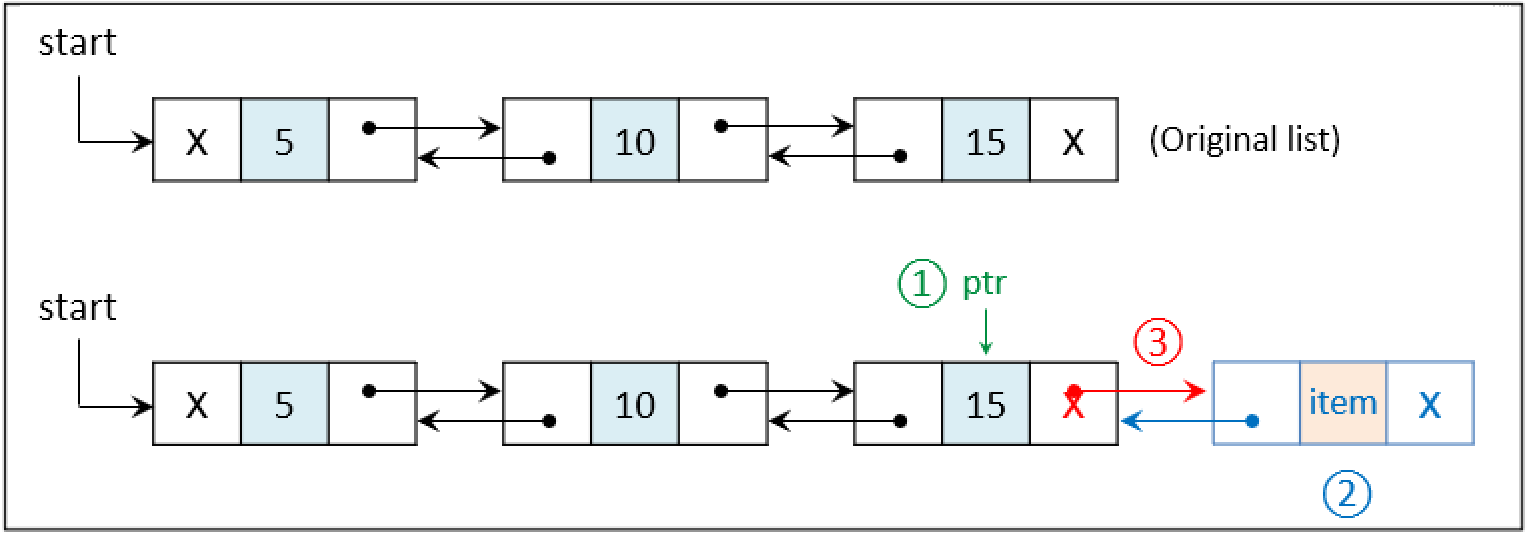
③ Point “start” to the new node.

- ## C Function:

```
/* Function to insert a node at the beginning of a double linked list */
DNode* InsertAtBeginingInDLL(DNode* start, int item)
{
    DNode *newNode;

    /* Step 1: Create a new node. Store “item” in its “info” field, “NULL” in its “prev” field,
    and point its “next” field to “start”. */
    newNode = (DNode *)malloc(sizeof(DNode));
    if (newNode == NULL)
    {
        printf("\nOut of memory space");
        return;
    }
    newNode->info = item;
    newNode->prev = NULL;
    newNode->next = start;

    start->prev = newNode;  /* Step 2: Point “start->prev” to the new node. */

    start = newNode;  /* Step 3: Point “start” to the new node. */

    return start;
}
```

Inserting a Node at the End a DLL

- **Technique:**



① Move a pointer "ptr" to the last node of the linked list.

② Create a new node. Store "item" in its "info" field, point its "prev" field to "ptr", and store "NULL" in its "next" field.

③ Point the "next" field of the last node (the pointer "ptr") to the new node.

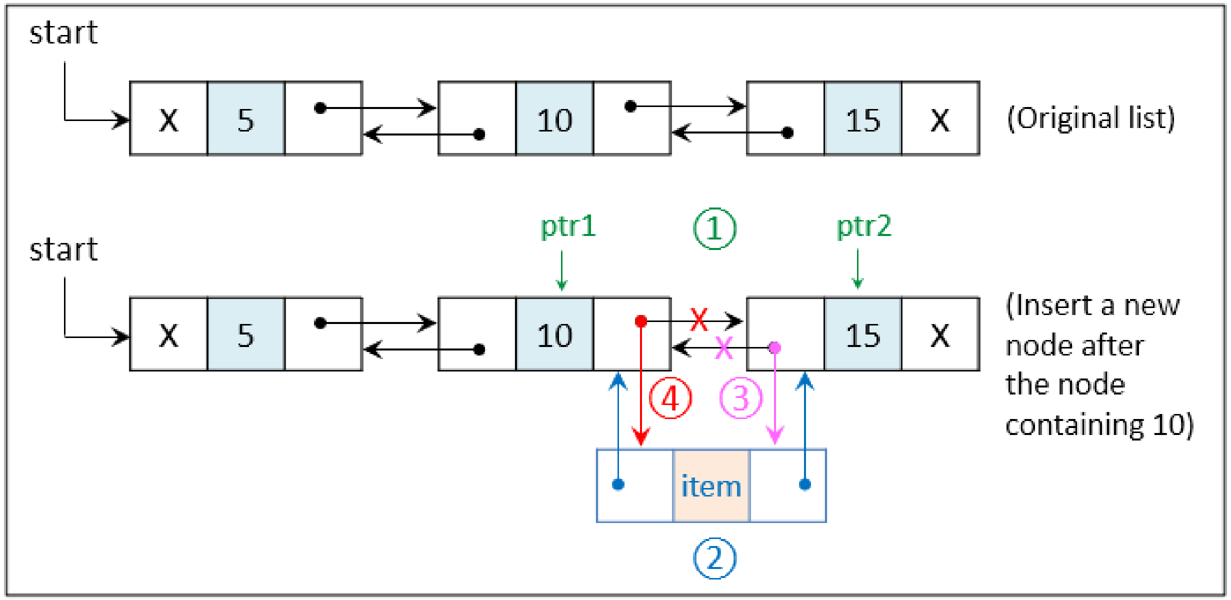
[NOTE]: We have already seen "how to insert a node at the end of a double linked list" (through the function "`InsertAtEndInDLL()`") in "creating a double linked list".

C Function:

```
/* Function to insert a node at the end of a double linked list. */
void InsertAtEndInDLL (DNode *start, int item)
{
    DNode *newNode, *ptr;

    /* Step 1: Move a pointer "ptr" to the last node of the linked list. */
    ptr = start;
    while(ptr->next != NULL)
        ptr = ptr->next;

    /* Step 2: Create a new node. Store "item" in its "info" field, point its "prev" field to "ptr",
    and store "NULL" in its "next" field. */
    newNode = (DNode *)malloc(sizeof(DNode));
    if (newNode == NULL)
    {
        printf("\nOut of memory space");
        return;
    }
    newNode->info = item;
    newNode->prev = ptr;
    newNode->next = NULL;

    ptr->next = newNode;  /* Step 3: Point the "next" field of the last node to the new node. */

    printf("A new node %d added to the list.\n ", newNode->info);
}
```

Inserting a Node at an Intermediate Position after a Given Node in a DLL

- **Technique:**



① Move a pointer “ptr1” to the node containing “data” (i.e., to the node after which a new node is to be inserted), and another pointer “ptr2” to its next node (i.e., to ptr1->next).

② Create a new node. Store “item” in its “info” field, point its “prev” field to “ptr1” and its “next” field to “ptr2”.

③ Point “ptr2->prev” to the new node.

④ Point “ptr1->next” to the new node.

- **C Function:**

```
/* Function to insert a node at an intermediate position after a given node (a node containing “data” in
its info field) in a double linked list. */
void InsertAfterNodeInDLL (DNode *start, int data, int item)
{
    DNode *newNode, *ptr1, *ptr2;;

    /* Step 1: Move a pointer “ptr1” to the node containing “data” (i.e., the node after which a new
    node is to be inserted), and another pointer “ptr2” to its next node (i.e., ptr1->next). */
    ptr1 = start;
    while((ptr1->info != data)&&(ptr1->next != NULL))
    {
        ptr1 = ptr1->next;
    }
    if(ptr1->next == NULL)
    {
        printf("\n%d is not found in the linked list.", data);
        exot(0);
    }
    ptr2 = ptr1->next;
```

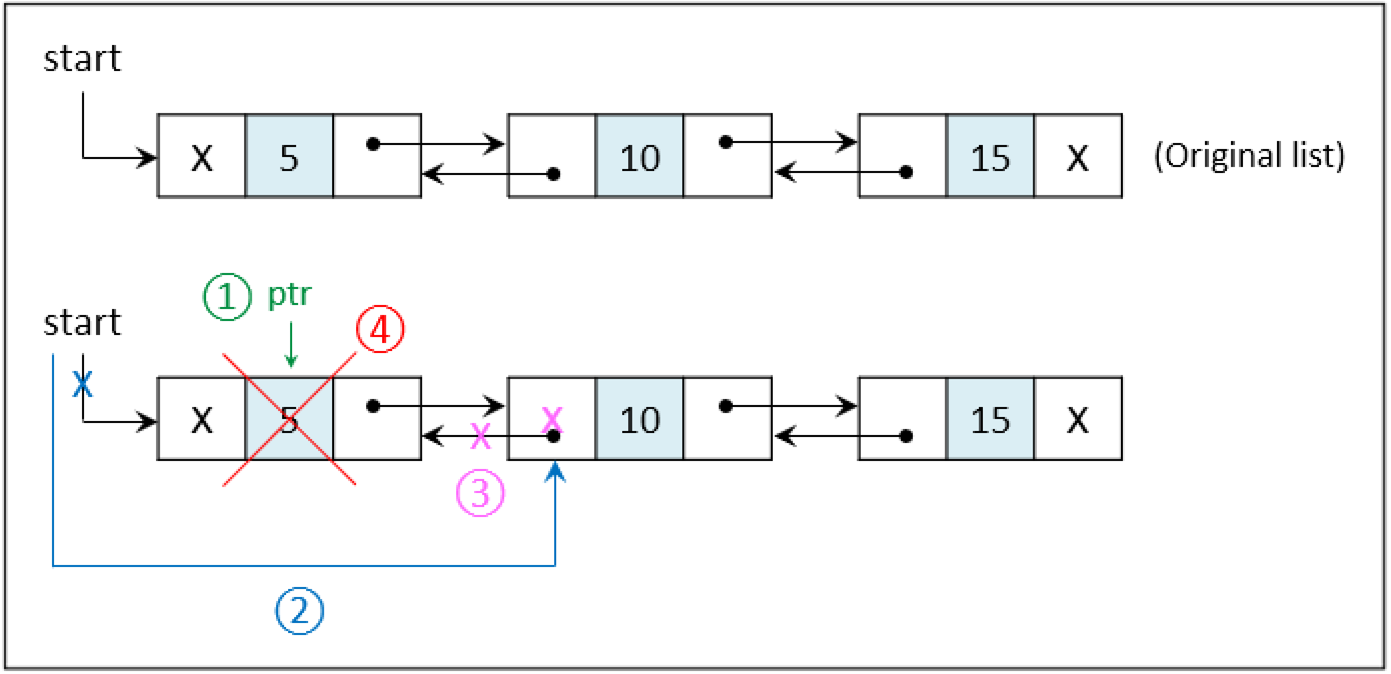
[Cont.]

```
    /* Step 2: Create a new node. Store “item” in its “info” field, point its “prev” field to “ptr1”
    and its “next” field to “ptr2”. */
    newNode = (DNode *)malloc(sizeof(DNode));
    if (newNode == NULL)
    {
        printf("\nOut of memory space");
        return;
    }
    newNode->info = item;
    newNode->prev = ptr1;
    newNode->next = ptr2;

    ptr1->next = newNode;  /* Step 3: Point “ptr2->prev” to the new node. */
    ptr2->next = newNode;  /* Step 3: Point “ptr1->next” to the new node. */

    printf("A new node with info field %d added after the node
    containing %d in the linked list.\n ", newNode->info, ptr1->info);
}
```

Deleting the First Node from a DLL

- **[NOTE] - Considerations while deleting the first and the last nodes from a double linked list:**
  - **If the linked list contains just one node, then after deletion of the node, the linked list shall be empty. So, in such case, after deletion of the node, “start” should point to NULL.**
  - **When the node is deleted its memory space must be freed (released).**

■ Technique (Deleting the First Node):



① Point a pointer “ptr” to the first node (i.e., to “start”).

② Point “start” to “ptr->next”.

③ Store “NULL” in “start->prev”.

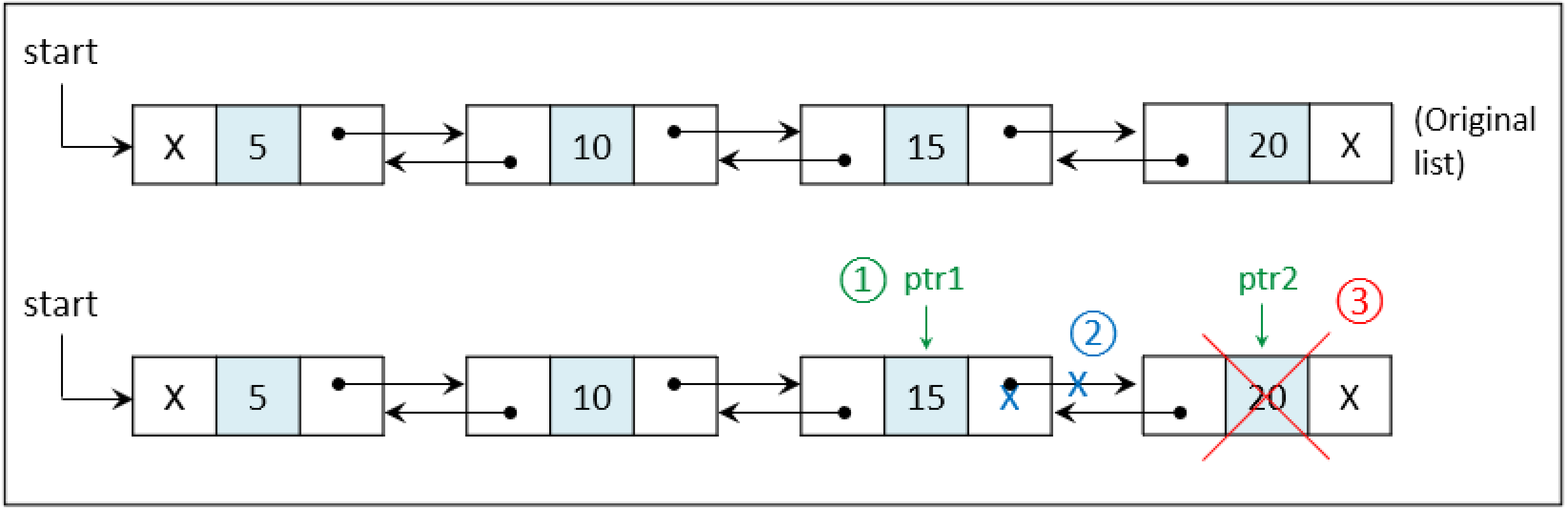
④ Release the node pointed out by “ptr”.

- **C Function:**

```
/* Function to delete the first node from a double linked list. */
void DeleteFirstNodeInDLL (DNode *start)
{
    DNode *ptr;

    /* Checking for a single node. */
    if(start->prev == start->next == NULL)
    {
        free(start);
        start = NULL;
        return start;
    }

    /* More than one node. */
    ptr = start;  /* Step 1: Point a pointer “ptr” to the first node (i.e., to “start”). */
    start = ptr->next;  /* Step 2: Point “start” to “ptr->next”. */
    Start->prev = NULL;  /* Step 3: Store “NULL” in “start->prev”. */
    free(ptr);  /* Step 3: Release the node pointed out by “ptr”. */

    return start;
}
```

Deleting the Last Node from a DLL

- **Technique:**



① Move a pointer “ptr1” to the last but one node, and another pointer “ptr2” to the last node.

② Store “NULL” in “ptr1->next”.

③ Release the node pointed out by “ptr2”.

- **C Function:**

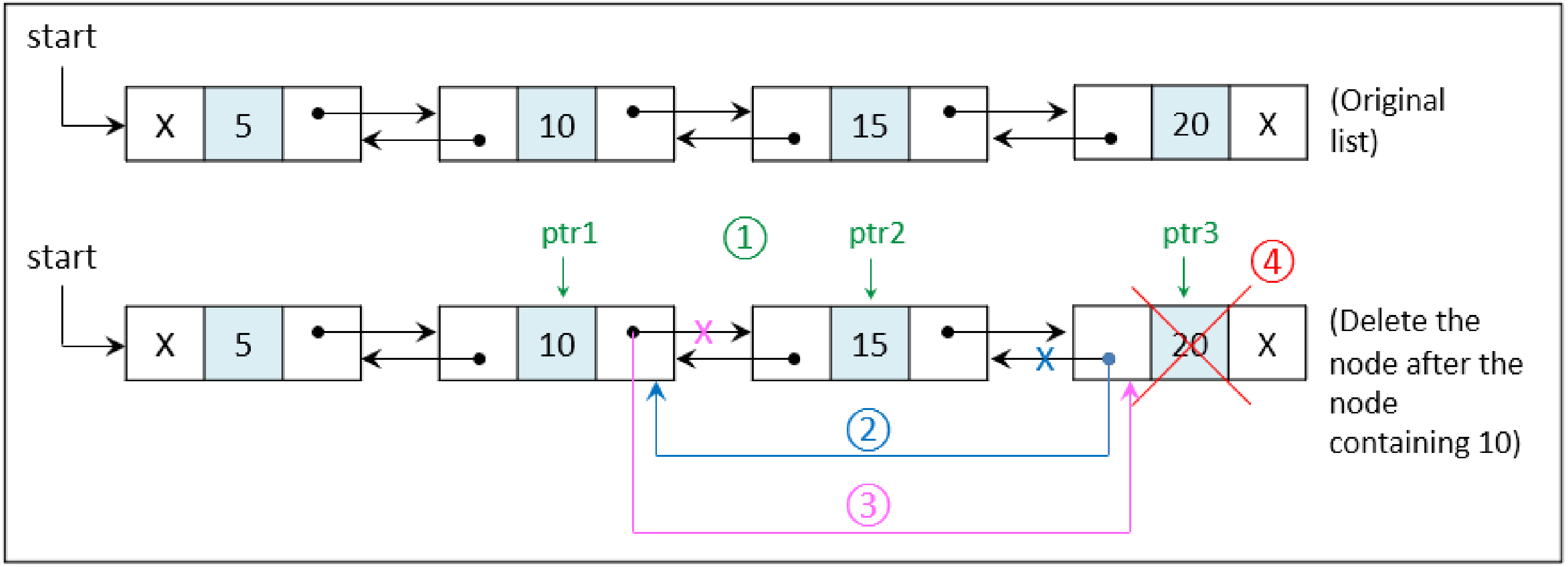
```
/* Function to delete the last node from a double linked list. */
void DeleteLastNodeInDLL (DNode *start)
{
    DNode *ptr1, *ptr2;

    /* Checking for a single node. */
    if(start->prev == start->next == NULL)
    {
        free(start);
        start = NULL;
        return start;
    }
```

[Cont.]

```
    /* More than one node. */

    /* Step 1: Move a pointer “ptr1” to the last but one node,
    and another pointer “ptr2” to the last node. */
    ptr1 = start;
    ptr2 = start->next;
    while((ptr2->next) != NULL)
    {
        ptr2 = ptr2->next;
        ptr1 = ptr1->next;
    }

    ptr1->next = NULL;  /* Step 2: Store “NULL” in “ptr1->next”. */

    free(ptr2);  /* Step 3: Release the node pointed out by “ptr2”. */

    return start;
}
```

Deleting an Intermediate Node after a Given Node in a DLL

- **Technique:**



① Move a pointer “ptr1” to the node containing “data” (i.e., whose next node is to be deleted), a pointer “ptr2” to the node which is to be deleted (i.e., ptr1->next), and one more pointer “ptr3” to it next node (i.e., to ptr2->next).

② Point “ptr3->prev” to “ptr1”.

③ Point “ptr1->next” to “ptr3”.

④ Release the node pointed out by “ptr2”.

C Function:

```
/* Function to delete an intermediate node after a given node (a node that contains "data" in its info
field) in a double linked list. */
void DeleteAfterNodeInDLL (DNode *start, int data)
{
    DNode *ptr1, *ptr2, *ptr3;

    /* Step 1: Move a pointer "ptr1" to the node containing "data" (i.e., whose next node is to be
    deleted), a pointer "ptr2" to the node which is to be deleted (i.e., ptr1->next), and one more pointer
    "ptr3" to it next node (i.e., to ptr2->next). */
    ptr1 = start;
    while((ptr1->info != data)&&(ptr1->next != NULL))
    {
        ptr1 = ptr1->next;
    }
    if(ptr1->next == NULL)
    {
        printf("\n%d is not found in the linked list.", data);
        exit(0);
    }
    ptr2 = ptr1->next;
    ptr3 = ptr2->next;

    ptr3->prev = ptr1;  /* Step 2: Point "ptr3->prev" to "ptr1".*/
    ptr1->next = ptr3;  /* Step 3: Point "ptr1->next" to "ptr3". */
    free(ptr2);  /* Step 4: Release the node pointed out by "ptr2". */
}
```

Searching an Element (a Value) in a DLL

- **Technique: The process is exactly same as in the case of single linked lists.** Move a pointer “ptr” from “start” to the last node till the “value” is not found in the “info” fields of one of the nodes. If the “value” is found, display the “node number”, otherwise display “value not found”.

- ## C Function:

```c
/* Function to search an item (a value) in a double linked list. */
void SearchInDLL (DNode *start, int item)
{
    DNode *ptr;
    int count = 1;

    ptr = start;
    while(ptr != NULL)
    {
        if(ptr->info == item)
        {
            printf("\n%d is found at node %d.", item, count);
            exit(0);
        }
        count++;
        ptr = ptr->next;
    }
    printf("\n%d is not found in the linked list.", item);
}
```

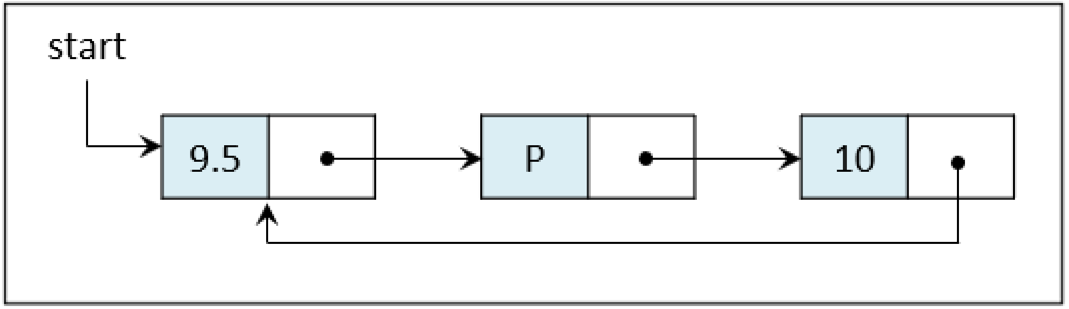
Assignments - III

Complete the experiments given in “Lab Manual - Section 16”.

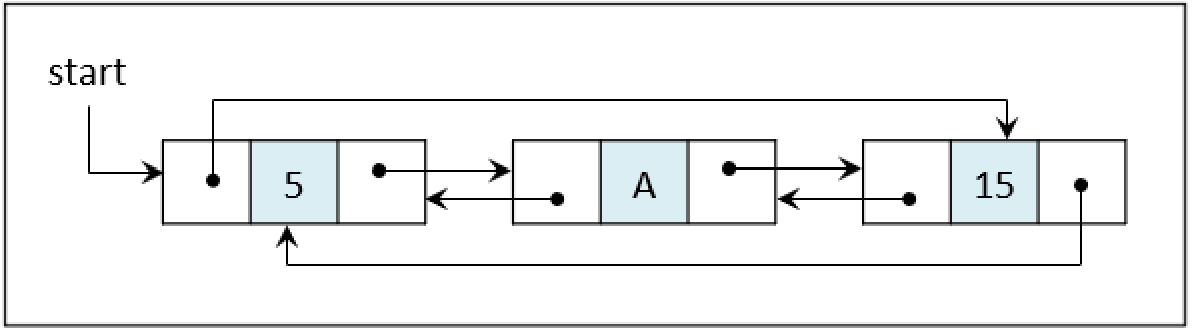
Circular & Header Linked Lists

Circular Linked Lists

- A circular linked list is a linked list where all the nodes are linked to form a circle. None of the nodes point to NULL.
- Variations:
  - Single Circular Linked Lists: It is a single linked list in which, the "next" field of the last node points back to the first node to form a circle instead of pointing to NULL.



- **Double Circular Linked Lists:** **It is a double linked list in which, the "prev" field of the first node points to the last node and the "next" field of the last node points to the 1st node to form a circle instead of pointing to NULL.**



Important Operations

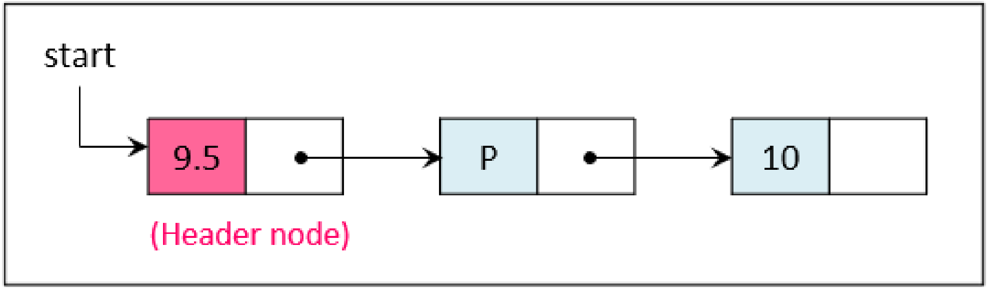
- List of important operations on single / double circular linked lists include:

    1. Creating a list.
    2. Traversing a list.
    3. Inserting a node in a list.
        - Inserting at the beginning
        - Inserting at the end
        - Inserting at an intermediate position after a given node
    4. Deleting a node from a list.
        - Deleting the first node
        - Deleting the last node
        - Deleting an intermediate node after a given node

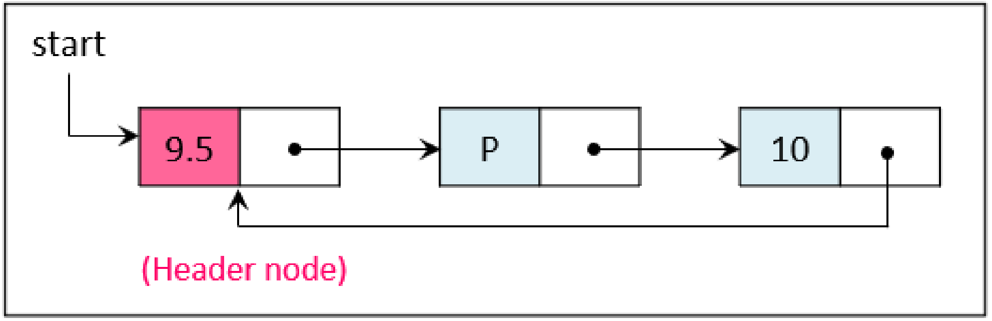
The technique of these operations is similar to that of the single / double linked lists, with just one addition. Here, we also have to manipulate the extra pointers in the first and the last nodes (wherever applicable).

Header Linked Lists

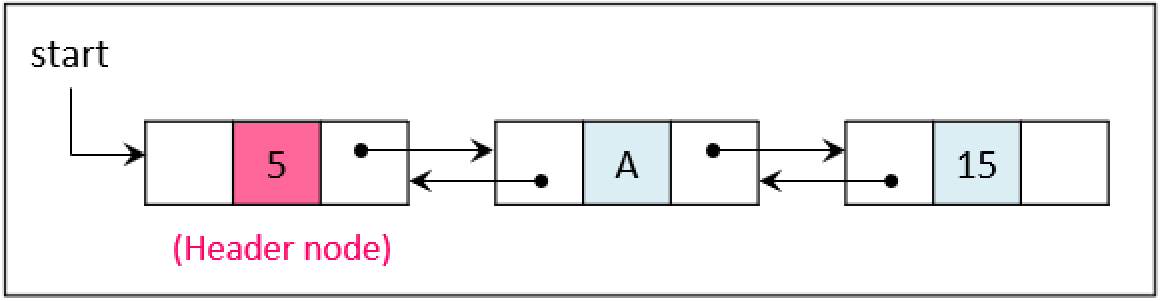
- A header linked list is a linked list which always contains a special node, called the “header node” at the beginning of the list (normally). The header node is commonly used for identification purpose.
- Variations:
  - Single linked list with a header node:



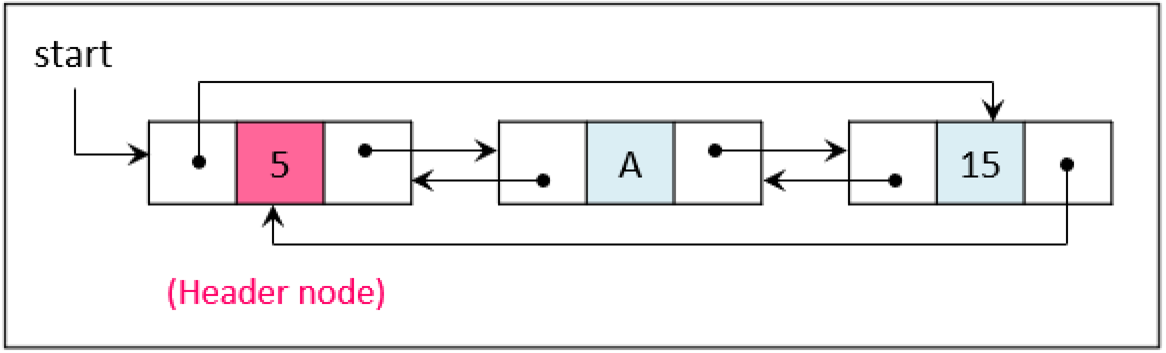
  - Single circular linked list with a header node:



- Double linked list with a header node:



- Double circular linked list with a header node:



Application of Linked Lists

Applications of Linked Lists In Computer Science

- In computer science, linked lists are mainly used for two purposes.
    1. As a base data structure to represent other data structures such as stacks, queues, trees, graphs (adjacency list representation), hash tables etc.
    2. In representation and manipulation of polynomials.

Use of Linked List in Polynomial Representation & Manipulation

- **What is a Polynomial?:** A single variable polynomial (with non-negative exponent) is defined as the following list:

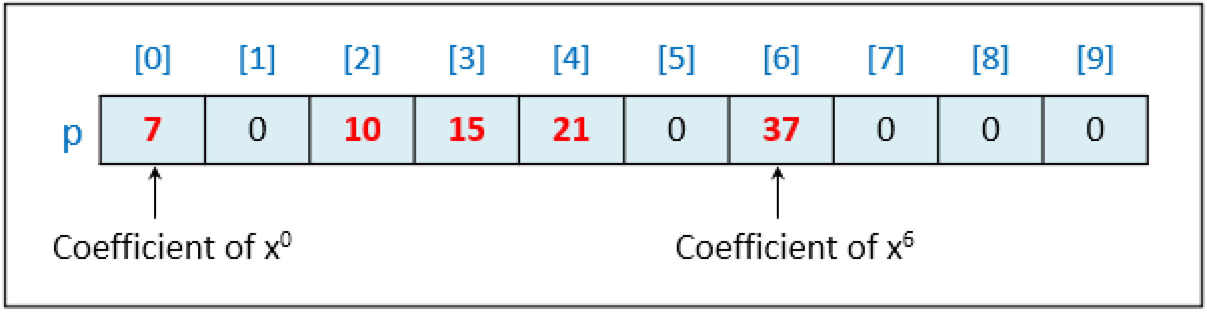
$\boxed{\mathrm{p(x)} = \sum_{i=0}^{n} a_i \, xi}$ e.g., $\boxed{\mathrm{p(x)} = 37x^6 + 21x^4 + 10x^2 + 3}$

- **Representations of Polynomials:** Within a computer, polynomials can be represented (and hence manipulated) by using the following two list structures:
  - Array
  - Linked list

- **Array Representation of Polynomials:** The following example demonstrates:

The polynomial $\mathrm{p(x)} = 7 + 10x^2 + 15x^3 + 21x^4 + 37x^6$

is represented as:



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
p	7	0	10	15	21	0	37	0	0	0

Coefficient of x^0

Coefficient of x^6

Analysis: In array representation, there is a wastage of memory space. So, this implementation is generally not used. Polynomials are represented by using a single linked list.

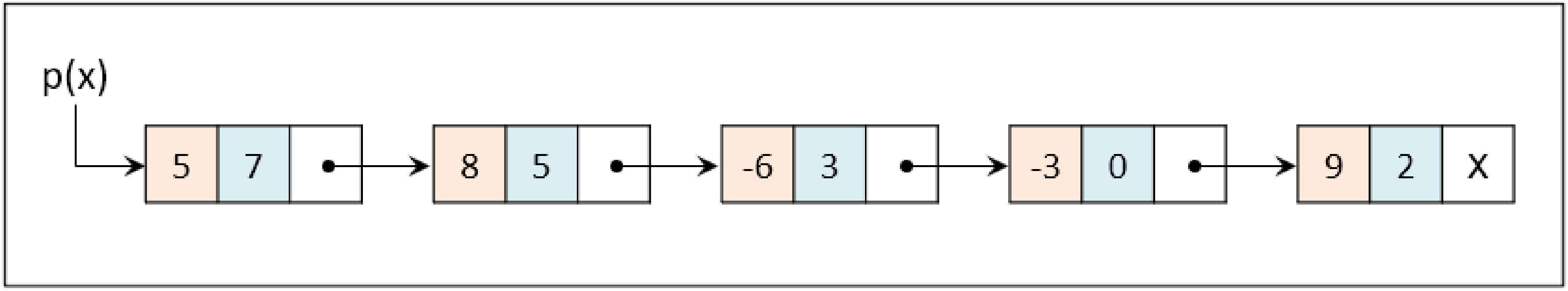
■ Linked List Representation of Polynomials:

In linked list representation, each element of a polynomial is represented as a node having three components as shown below:

coeff	exp	next

For example, the polynomial $p(x) = 5x^7 + 8x^5 - 6x^3 + 9x^2 - 3$

is represented as:



End of Chapter 12