LINEAR DATA STRUCTURE

LINKED LIST

CO1: Classify and Apply the concepts of stacks, queues and linked list in real life problem solving

Reference:

1. Reema Thareja; Data Structures using C; Oxford
2. Ellis Horowitz, Sartaj Sahni; Fundamentals of Data Structures; Galgotia Publications
3. Jean Paul Tremblay, Paul G. Sorenson; An introduction to data structures with applications; Tata McGrawHill

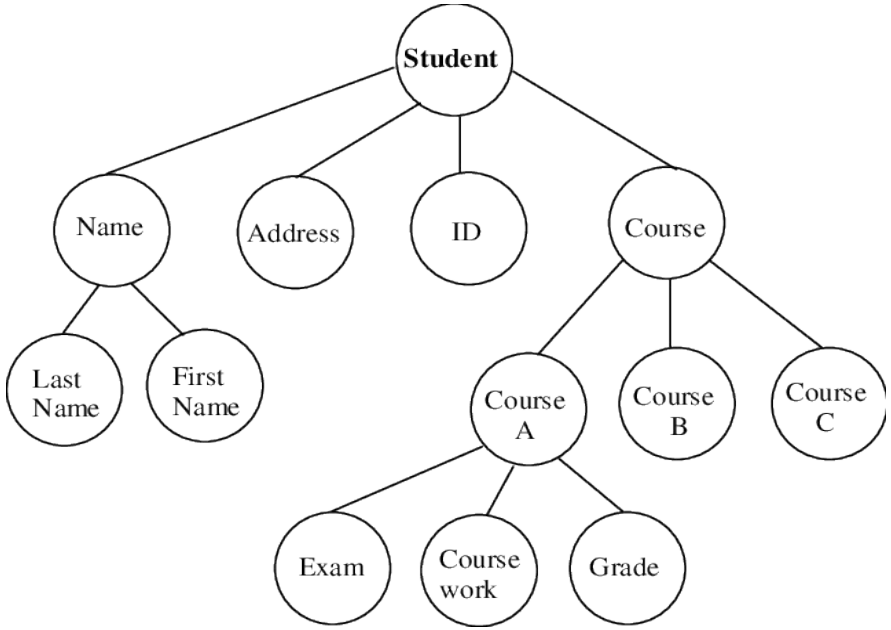
LINKED LIST

- Introduction of data structures
- Types of data structures
- Operations of Data Structures
- Introduction of linked list
- Why linked list
- Linked List – Node Structure
- Types of linked list
- Dynamic Memory Management
- Dynamic memory allocation
- Singly linked list
- Doubly linked list
- Circular Linked List
- Applications of linked list
- Analysis of Algorithms
- Analysis of Array
- Analysis of Linked List

INTRODUCTION OF DATA STRUCTURES

Data Structure:

- Data structure is a way of storing and organizing data in a computer so that it can be used efficiently
- It shows the relationship of one data element with the other and organize it within the memory
- It helps you to analyse the data, store it and organize it in a logical or mathematical manner



INTRODUCTION OF DATA STRUCTURES

Data Structure:

- Data structure is a DFA…..set of Domains D, a set of functions F and a set of axioms A
- Domain (D) : Range of values the data may have
- Functions (F) : Set of operations that can be performed on data
- Axioms (A): Set of rules to perform set of operations (F)

Array X

- Domain (D): Range of integers
- Functions (F): Insert, Delete, Count, Traversal, Sorting, Empty, Full
- Axioms (A): Insert(X, n) -> X

  Empty(X) - > True or False

  Count(X) -> count_no

TYPES OF DATA STRUCTURES

- Primitive and Non-primitive:
  - Primitive Data Structure: The fundamental data types which are supported by a programming language
    - int, char, float, pointers, bool
  - Non-primitive Data Structure: Defines set of derived elements that can be formed using primitive data structure
    - Arrays, Structure, Liked list, Stack, Queue, Tree Graph

- Static and Dynamic:
  - Static Data Structure: It has fixed memory size. Memory is allocated at the time of compilation
    - Array
  - Dynamic Data Structure: It has no fixed size and memory can be allocated dynamically at run time
    - Linked List

TYPES OF DATA STRUCTURES

- Linear and Non-linear:
- Linear Data Structure:
  - Elements are arranged to form a linear sequence
  - Every data element has a unique successor and predecessor
  - There is one to one relationship between the elements
  - Array, Linked Lists, Stack, Queue are examples of Linear Data Structures

- Non-linear data structures:
  - Every data element may have more than one predecessor as well as successor
  - They are used to represent the data containing hierarchical or network relationship among the elements
  - Trees and graphs are examples of non-linear data structures

OPERATIONS ON DATA STRUCTURES

- Traversing: It means to access each data item exactly once so that it can be processed

  To print the names of all the students in a class

- Searching: It is used to find the location of one or more data items that satisfy the given constraint

  Such a data item may or may not be present in the given collection of data items

  To find the names of all the students who has enrolled for course Data structure

- Inserting: It is used to add new data items to the given list of data items

  To add the details of a new student who has recently joined the course Programming

Roll No.	Name of the student	Course
1	Smita Patil	Data Structure
2	Esha Surve	Programming
3	Monali Chaudhari	Data structure
4	Kunal Kamat	Data structure
5	Manthan Naik	Programming
6	Shahid Kapoor	Programming

Smita Patil
Esha Surve
Monali Chaudhari
Kunal Kamat
Manthan Naik

OPERATIONS ON DATA STRUCTURES

- Deleting: It means to delete a particular data item from the given collection of data items

  To delete the name of a student who has left the course

- Sorting: Data items can be arranged in some order like ascending order or descending order depending on the type of application

  Arranging the names of students in a class in an alphabetical order

- Merging: Lists of two sorted data items can be combined to form a single list of sorted data items

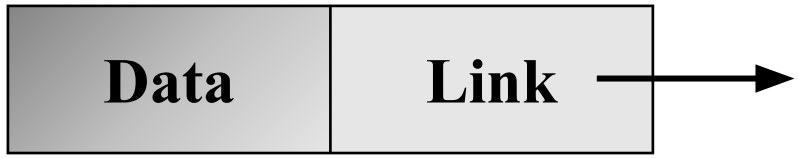
  Combining students of two courses computer and IT

Roll No.	Name of the student	Course
1	Smita Patil	Data Structure
2	Esha Surve	Programming
3	Monali Chaudhari	Data structure
4	Kunal Kamat	Data structure
5	Manthan Naik	Programming
6	Shahid Kapoor	Programming

Roll No.	Name of the student	Course
2	Esha Surve	Programming
4	Kunal Kamat	Data structure
3	Monali Chaudhari	Data structure
6	Shahid Kapoor	Programming
1	Smita Patil	Data Structure

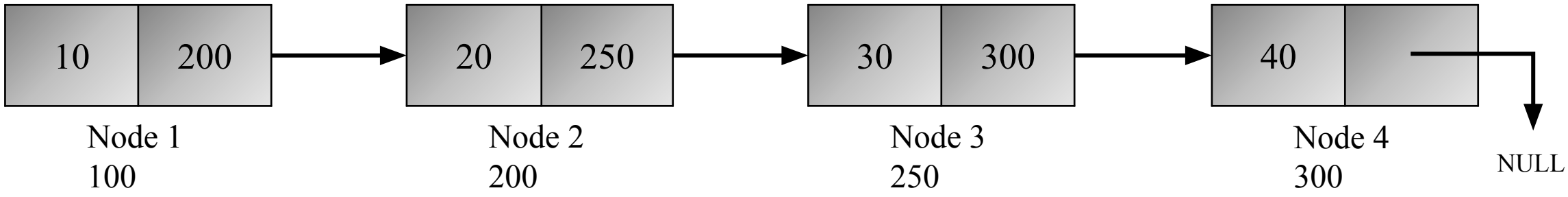
INTRODUCTION OF LINKED LIST

- Linked list is an ordered collection of data elements called as nodes that are randomly stored in memory
- Linked list is a linear and dynamic data structure
- Node is a structure consisting of two fields: Data field and Next field



Node

- Linked list example:



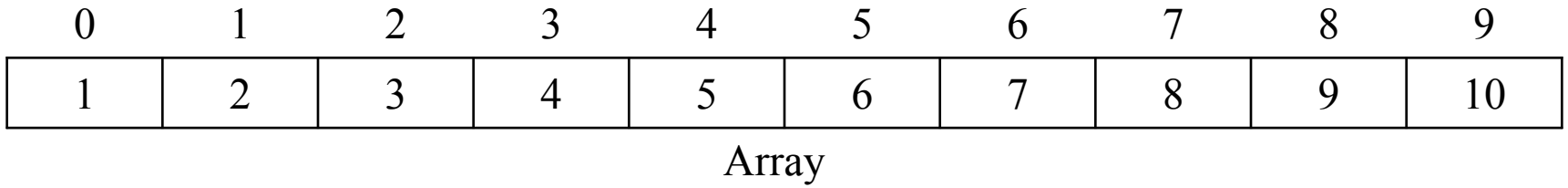
WHY LINKED LIST

Limitations of Array:

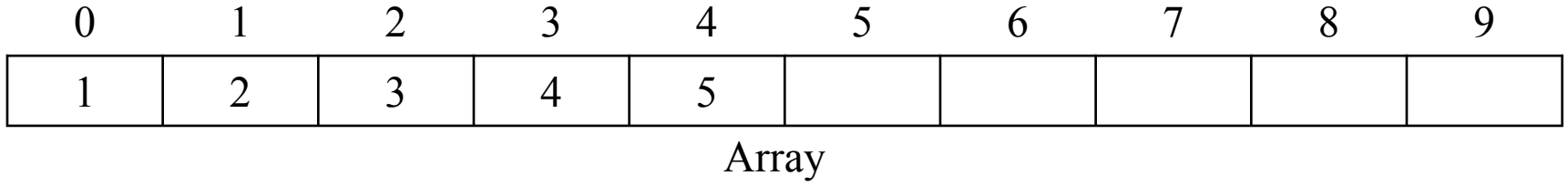
Array declaration:

int a[10];

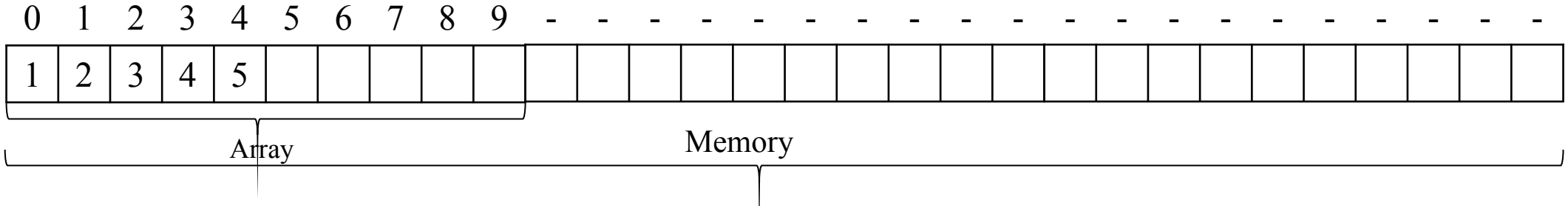
Case 1: User has declared array of size 10 and stored 10 elements in an array



Case 2: User has declared array of size 10 and stored only 5 elements



Case 3: User has declared array of size 10 and want to increase size of an array



WHY LINKED LIST

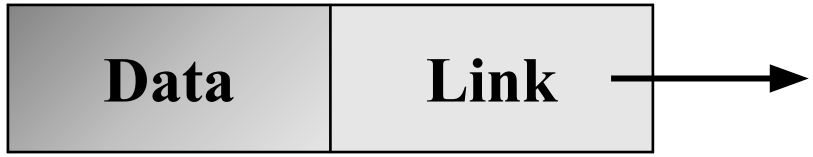
Limitations of Array:

- In array, elements are stored in contiguous memory locations
- Static Data Structure: Size must be known at the time of declaration before compilation of the program
- Insertion and deletion of elements in between the array requires a lot of data movement

Linked List:

- In linked list, elements are not stored contiguous memory locations
- Each node is connected to its successor through the link (implemented using pointers)
- Linked list is used to store similar type of data in memory

LINKED LIST – NODE STRUCTURE



Node

Structure for Linked List:

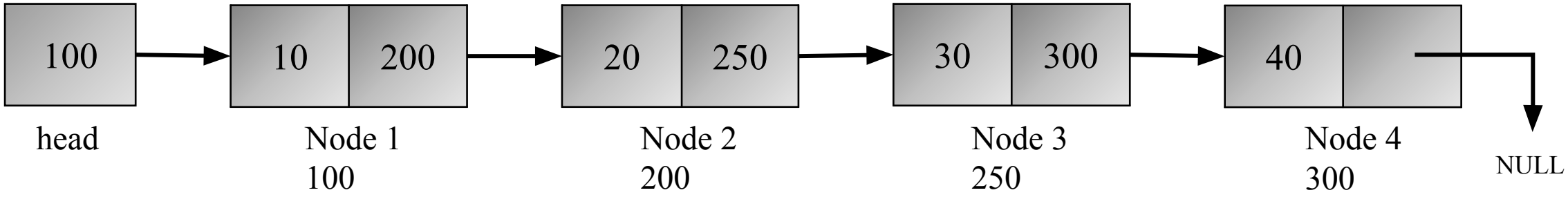
```
struct node
{
    int data;
    struct node *next;
};
```

TYPES OF LINKED LIST

There are 4 Types of Linked List:

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Doubly Circular Linked List

Singly Linked List:



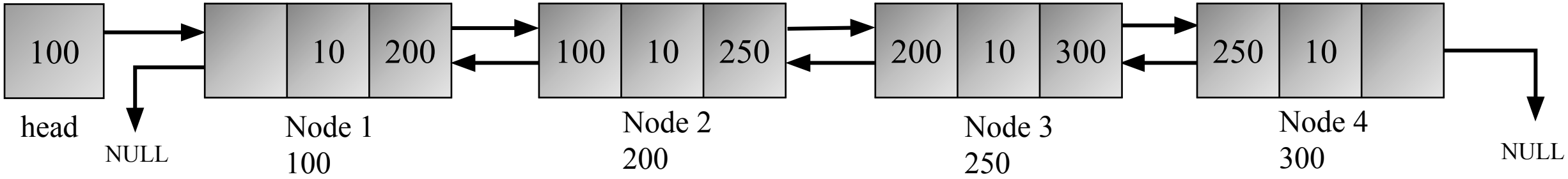

```
struct node
{
    int data;
    struct node *next;
};
```

TYPES OF LINKED LIST

Doubly Linked List:

Node Structure:

Example:

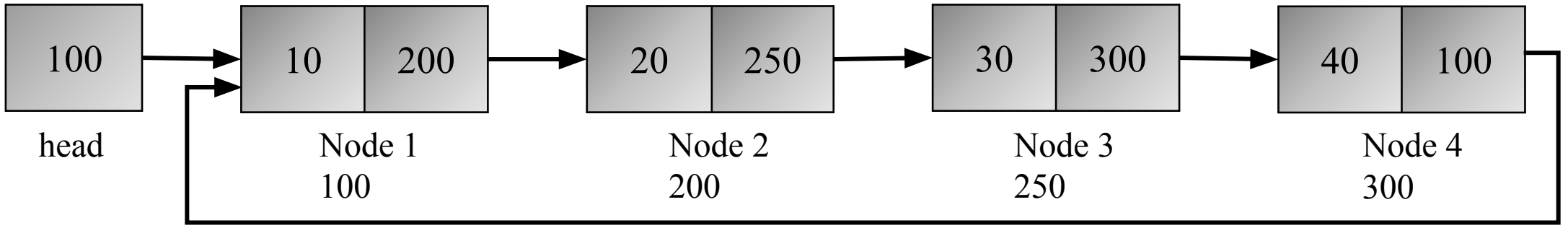



```
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};
```

TYPES OF LINKED LIST

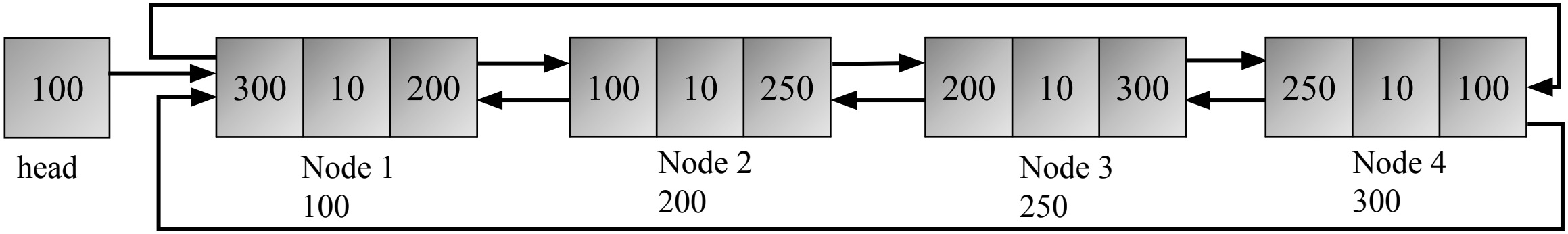
Circular Linked List:

Example:



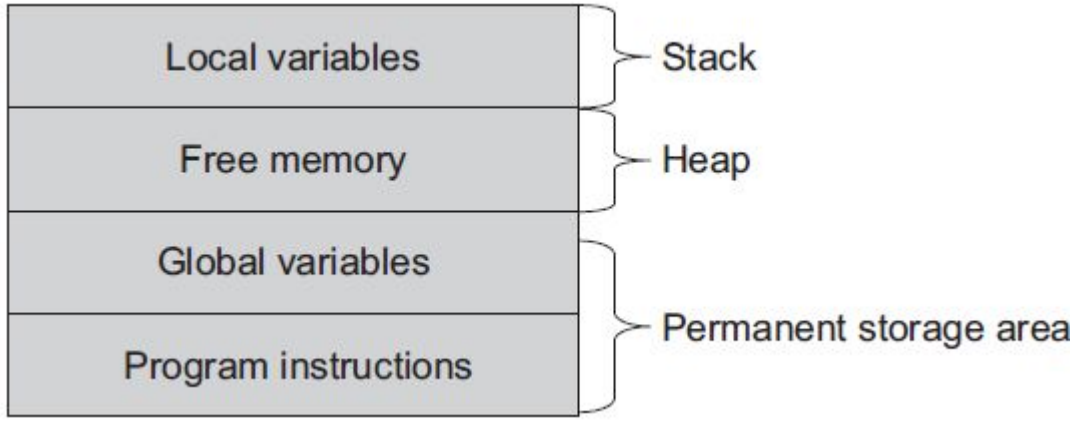
Doubly Circular Linked List:

Example:



DYNAMIC MEMORY MANAGEMENT

- Many languages permit a programmer to specify an array's size at run-time so they have the ability to calculate and assign, during execution, the memory space required by the variables in a program
- The process of allocating memory at run-time is known as *dynamic memory allocation*



- Permanent Storage Area: Program instructions, Global and Static variables are stored
- Stack: Local variables are stored on stack
- Heap: Memory space available between permanent storage area and stack can be used for dynamic allocation during program execution and Size of the heap keeps changing when a program is executed because of the creation and deletion of the variables that are local to the functions and blocks

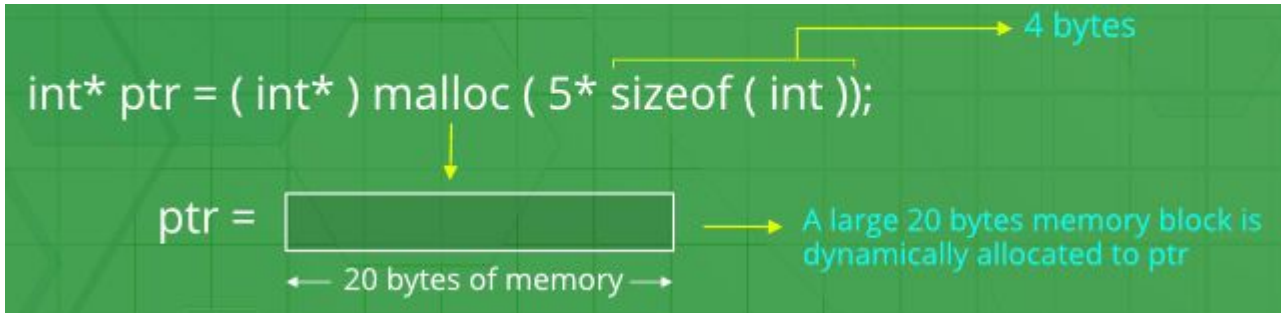
DYNAMIC MEMORY ALLOCATION

- Dynamic memory management technique allows us to allocate memory dynamically or release unwanted space at run time, optimizing the use of storage space
- There are 4 library functions available in C for dynamic memory allocation
  - malloc()
  - calloc()
  - free()
  - realloc()

- malloc(): - Memory Allocation
- Reserves a block of memory of specified no. of bytes and returns a pointer of type void

Syntax: *ptr = (cast-type *) malloc(byte size);*

Example: if we want to allocate space in memory for 5 integers

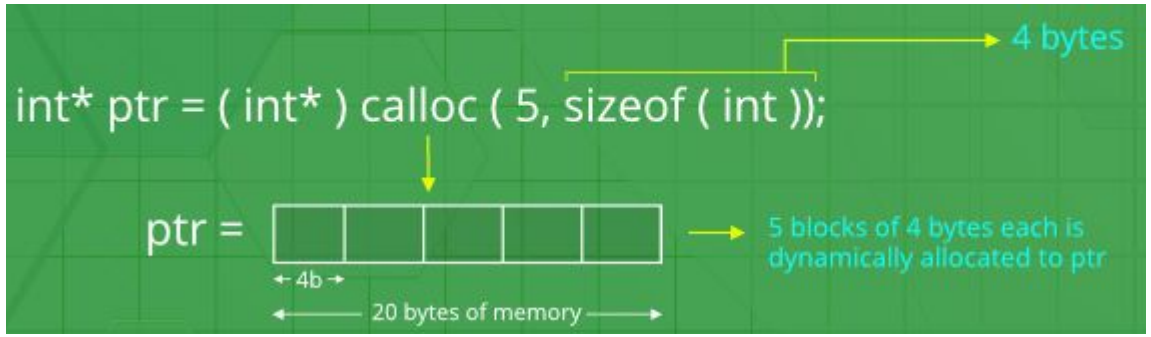


DYNAMIC MEMORY ALLOCATION

- calloc(): - Contiguous Allocation
- Reserves a multiple block of memory each of same size and initializes all bits to zero

Syntax: *ptr = (cast-type *) calloc(n, byte size);*

Example: if we want to allocate space in memory for 5 integers, 5 separate block of memory will be allocated



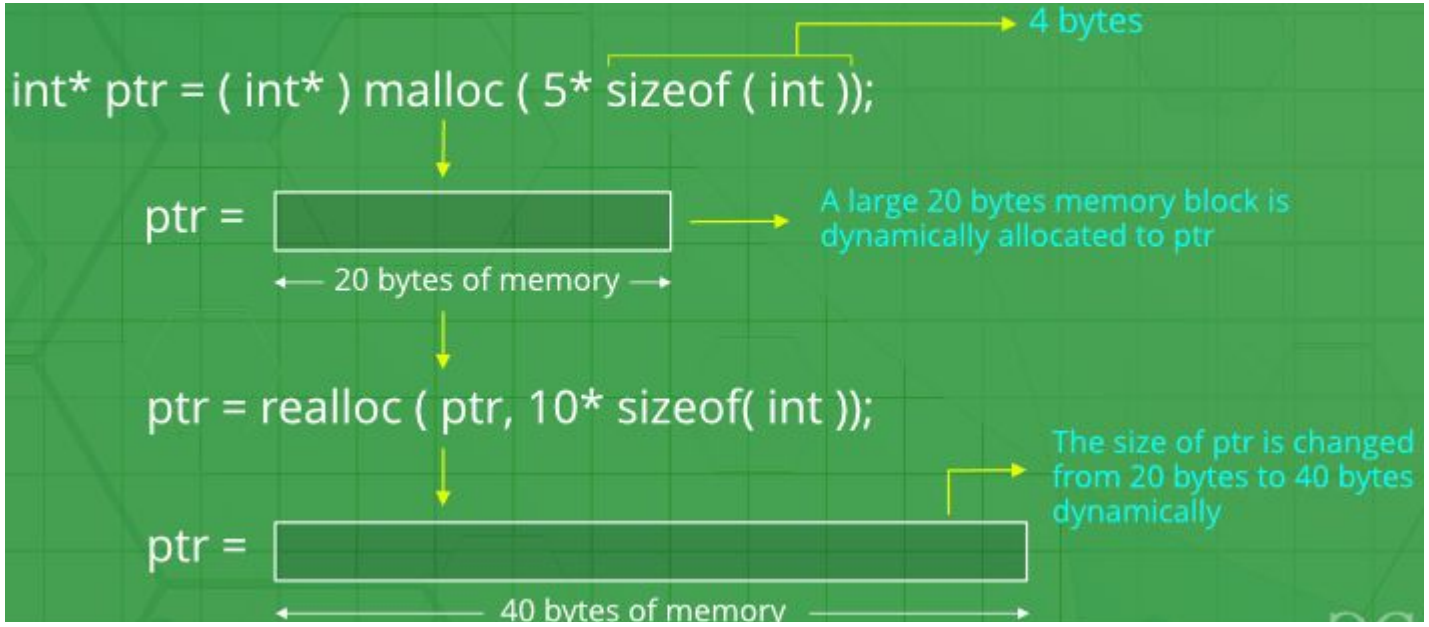
- Free(): - Deallocate memory
- Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syntax: *free(ptr);*

DYNAMIC MEMORY ALLOCATION

- realloc(): - Re Allocation
- If the dynamically allocated memory with the help of malloc() or calloc() is insufficient, realloc() can be used to dynamically re-allocate memory
- Re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value

Syntax: *ptr = realloc(ptr, new byte size);*

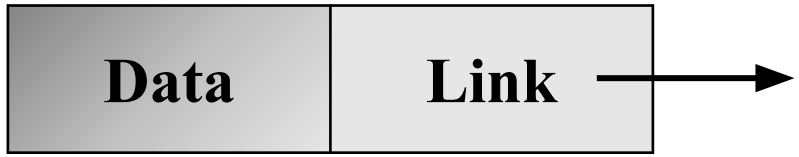


SINGLY LINKED LIST

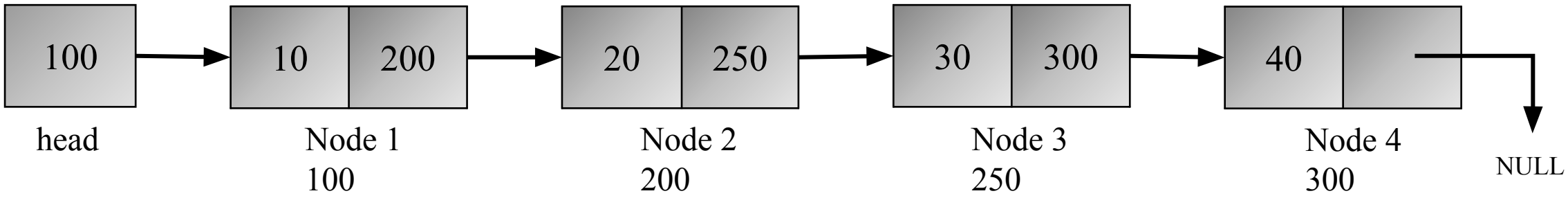
- Introduction
- Creation of linked list
- Display linked list
- Insertion of node in linked list
- Deletion of node from linked list
- Reverse linked list
- Updating linked list

INTRODUCTION

Singly Linked List:



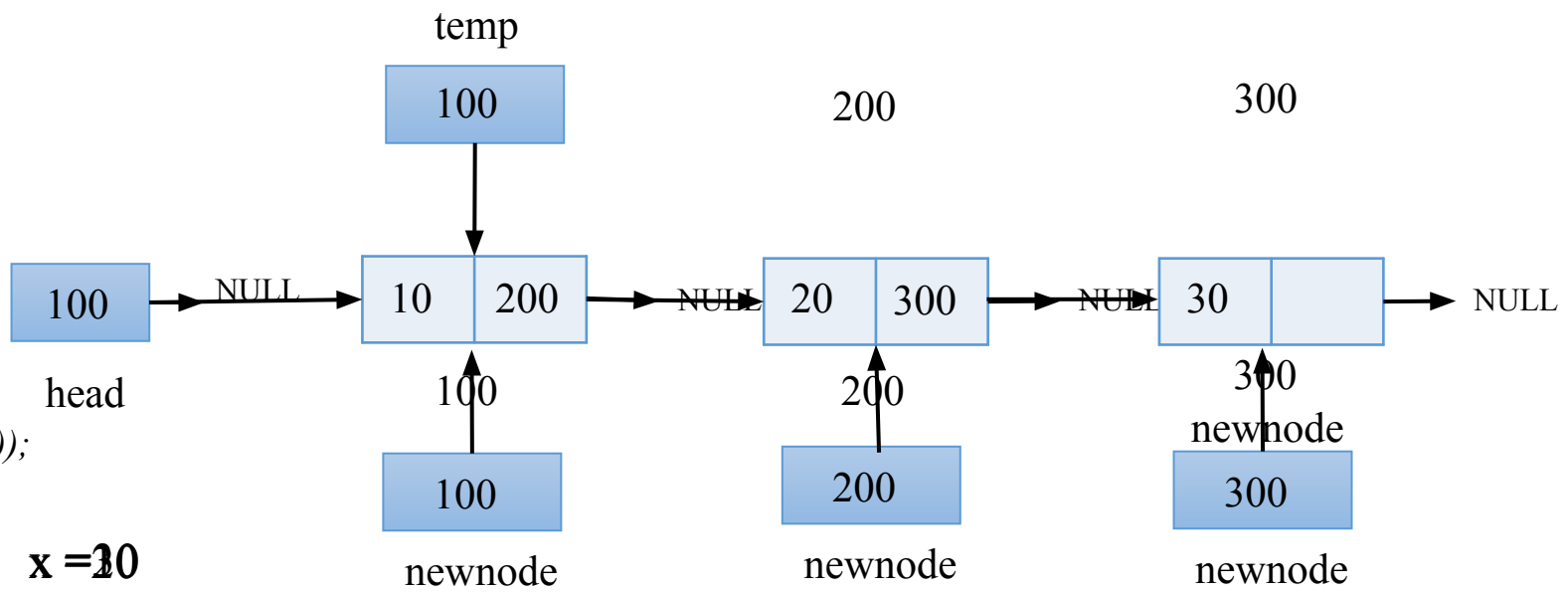
Node




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

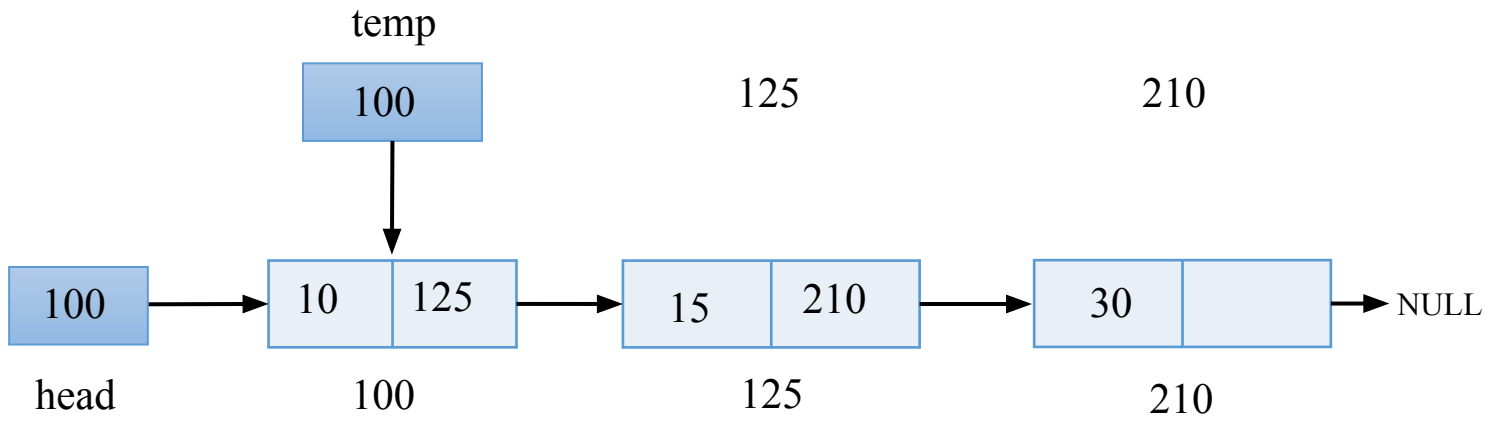
CREATION OF SINGLY LINKED LIST

```
node *createsll( )
{
        node *head = NULL, *temp, *newnode;
    int x; char ans;
    do
    {
        printf("Enter data to node:");
        scanf("%d", &x);
                newnode = (node *)malloc(sizeof(node));
                newnode->data = x;
                newnode->next = NULL;
                if(head  == NULL)
                        head = temp = newnode;
        else
        {
                        temp->next = newnode;
                        temp = newnode;
        }
        printf("Do you want to add more nodes?");
        scanf("%c", &ans);
    }while(ans == 'Y');
    return head;
}
```



DISPLAY SINGLY LINKED LIST

```
void display(node *head)
{
        node *temp = head;
        if(temp == NULL)
        printf("List is empty");
    else
    {
            while(temp!=NULL)
        {
                printf("%d ", temp->data);
                temp = temp->next;
        }
    }
}
```



10 15 30

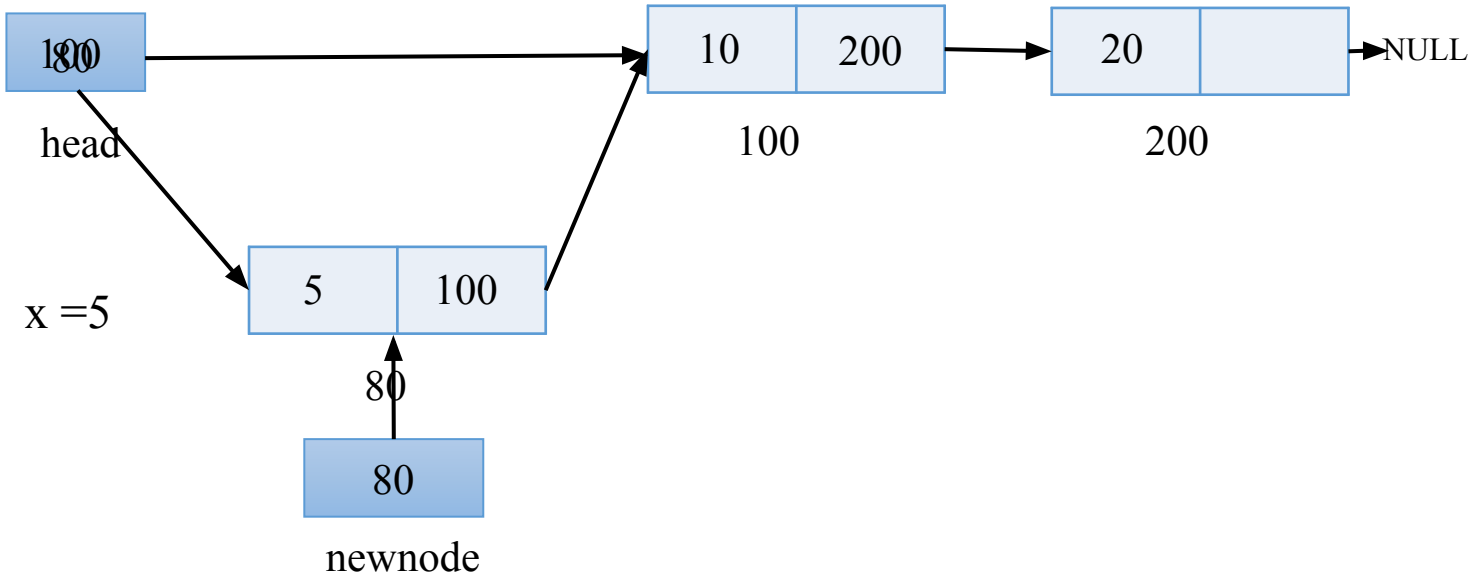
INSERTION OF NODE IN SINGLY LINKED LIST

There are three ways of inserting a node in a linked list –

- Insertion of a node at the beginning
- Insertion of a node in a middle
- Insertion of a node at the end

Insertion of a node at the beginning:

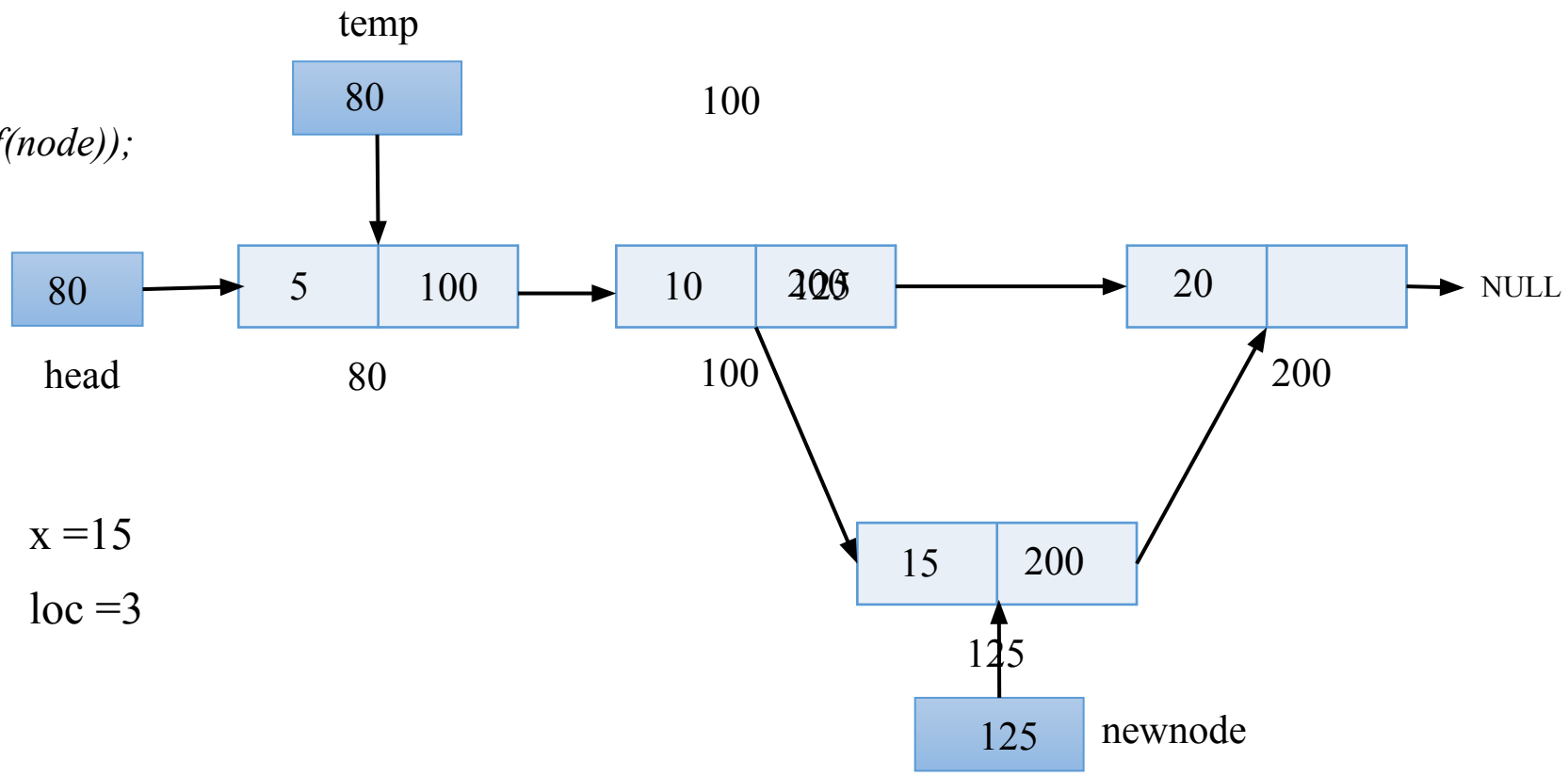
```
node  * insertatbeginning(node *head)
{
        node *newnode;
        int x;
        printf("Enter data to node:");
        scanf("%d", &x);
              newnode = (node *)malloc(sizeof(node));
              newnode->data = x;
              newnode->next = head;
              head = newnode;
}
```



INSERTION OF NODE IN SINGLY LINKED LIST

- Insertion of a node in the middle:

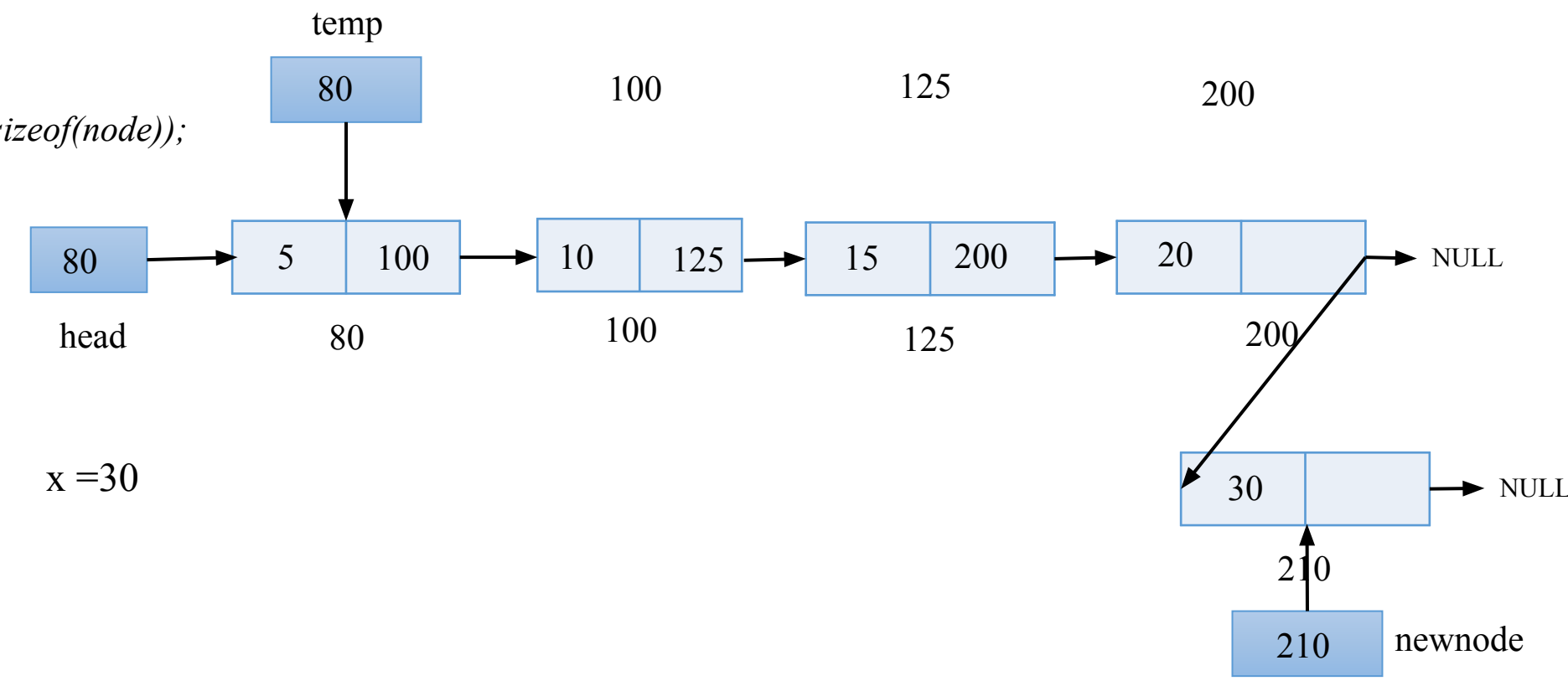
```
node  * insertinmiddle(node *head, int loc)
{
        node *newnode;
        int x, i=1;
        printf("Enter data to node:");
        scanf("%d", &x);
              newnode = (node *)malloc(sizeof(node));
              newnode->data = x;
              temp = head;
              while(i< (loc-1))
        {
                    temp = temp->next;
               i++;
        }
              newnode->next = temp->next;
              temp->next = newnode;
}
```



INSERTION OF NODE IN SINGLY LINKED LIST

- Insertion of a node at the end:

```
node  * insertatend(node *head)
{
        node *newnode;
        int x;
        printf("Enter data to node:");
        scanf("%d", &x);
              newnode = (node *)malloc(sizeof(node));
              newnode->data = x;
              newnode->next  = NULL;
               temp = head;
              while(temp->next != NULL)
        {
                      temp = temp->next;
        }
              temp->next = newnode;
}
```



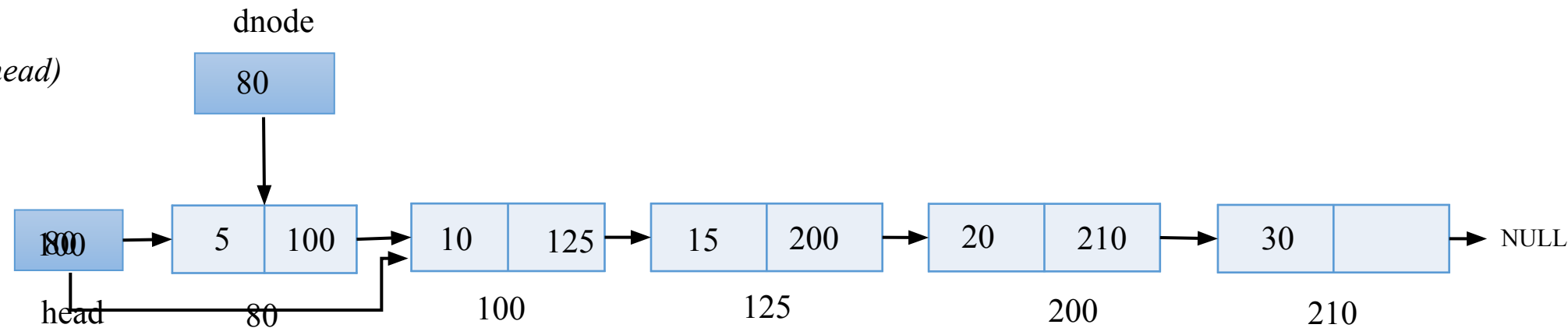
DELETION OF NODE FROM SINGLY LINKED LIST

There are three ways of deleting a node from a linked list –

- Delete the first node
- Delete node from middle of linked list
- Delete the last node

- Deletion of the first node:

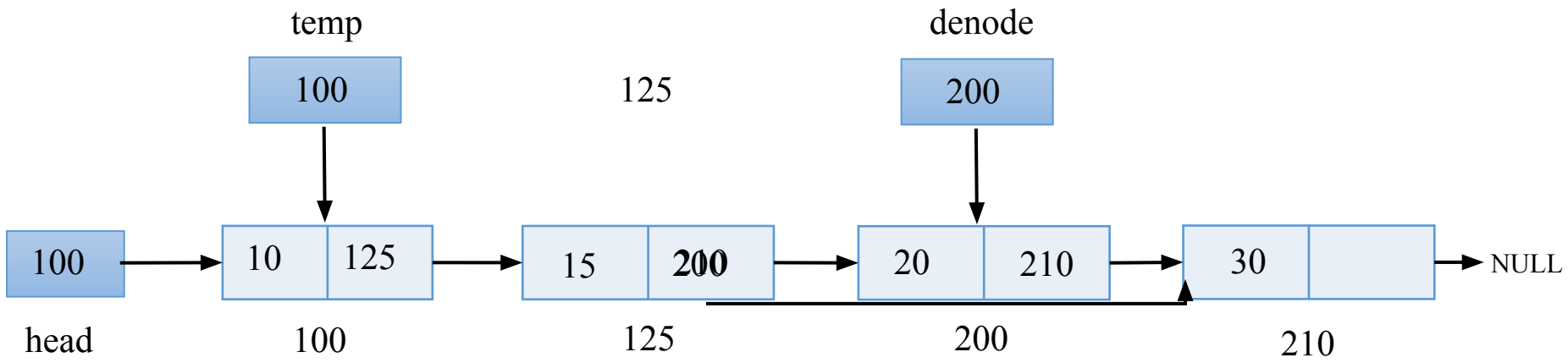
```
node  * deletefirstnode(node *head)
{
        node *dnode;
              dnode = head;
              head = head->next;
              free(dnode);
             return head;
}
```



DELETION OF NODE FROM SINGLY LINKED LIST

- Deletion of a middle node:

```
node  * deletemiddlenode(node *head)
{
        node *temp, *dnode;
        int loc, i=1;
        printf("Enter the location of the node you want to delete: ");
        scanf("%d", &loc);
                temp = head;
                while(i< (loc-1))
        {
                        temp = temp->next;
                 i++;
                }
                dnode = temp->next;
                temp->next = dnode->next;
                free(dnode);
                return head;
}
```

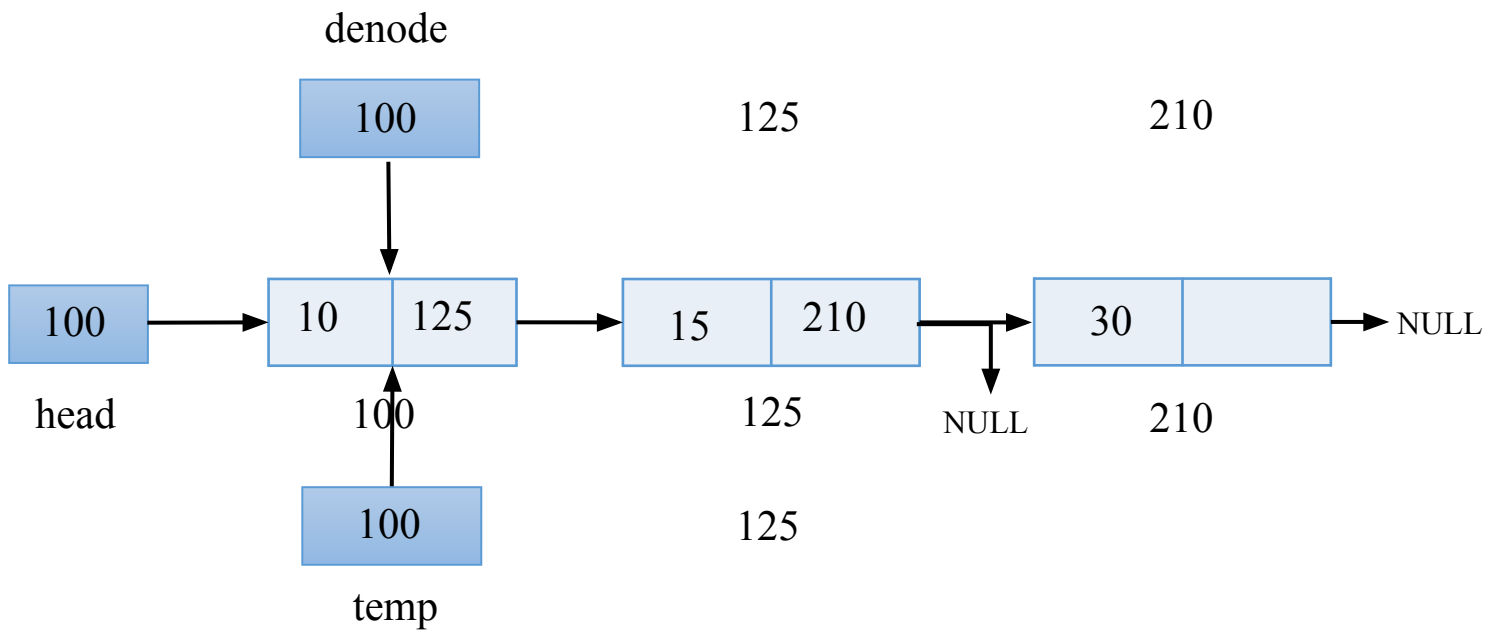


loc = 3

DELETION OF NODE FROM SINGLY LINKED LIST

- Deletion of a last node:

```
node  * deletelastnode(node *head)
{
        node *temp, *dnode;
                dnode = head;
                while(dnode->next != NULL)
        {
                        denode = temp;
                        dnode = dnode->next;
                }
                if(dnode = head)
                        head = NULL;
        else
        {
                        temp->next = NULL;
        }
                free(dnode);
                return head;
}
```

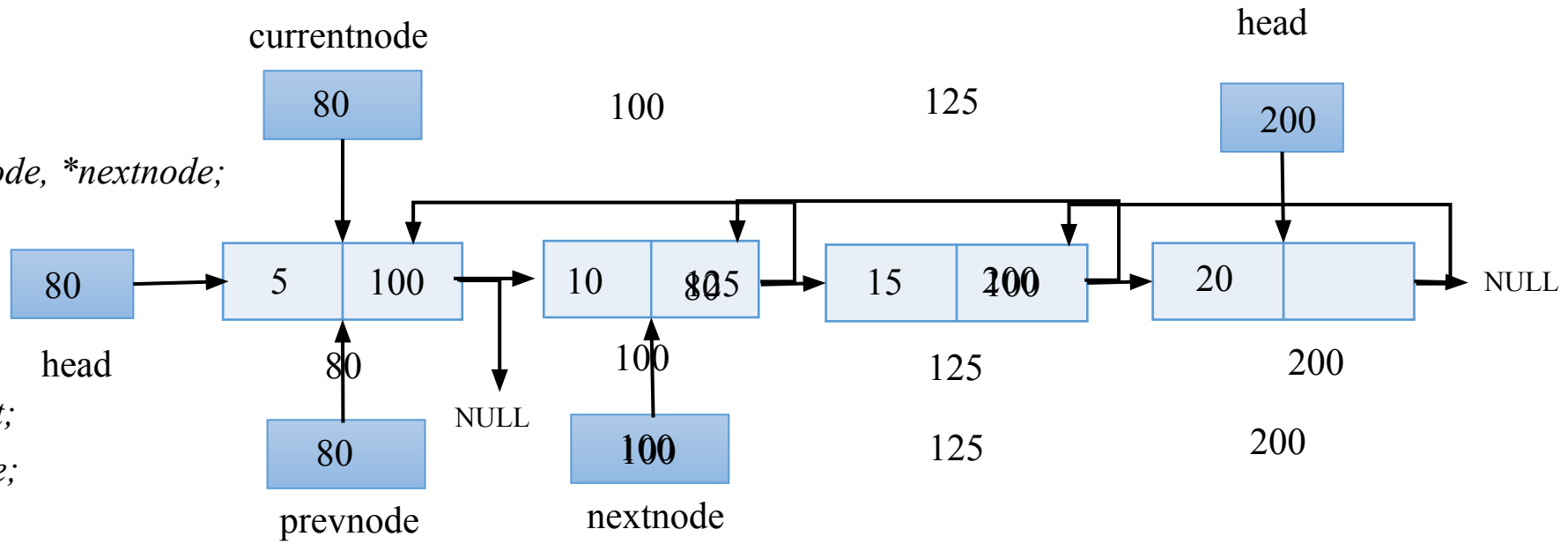


REVERSE SINGLY LINKED LIST

There are two methods to reverse a linked list:

- Iterative Method
- Recursive method

```
void  * displayrevers(node *head)
{
        node *prevnode = NULL, *currentnode, *nextnode;
        currentnode = head;
        while(currentnode != NULL)
        {
                nextnode = currentnode->next;
                currentnode->next = prevnode;
                prevnode = currentnode;
                currentnode = nextnode;
         }
        head = prevnode;
        display(head);
}
```

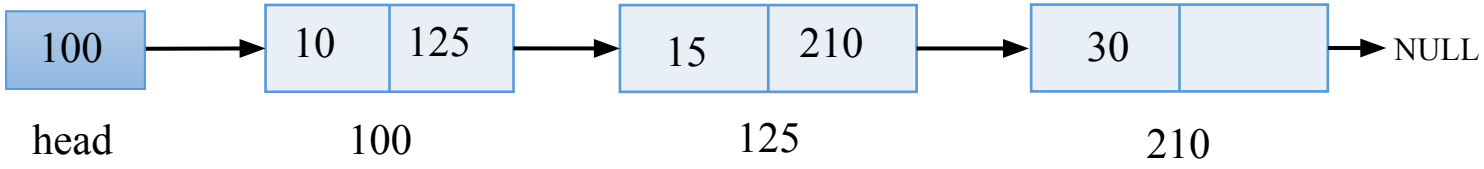


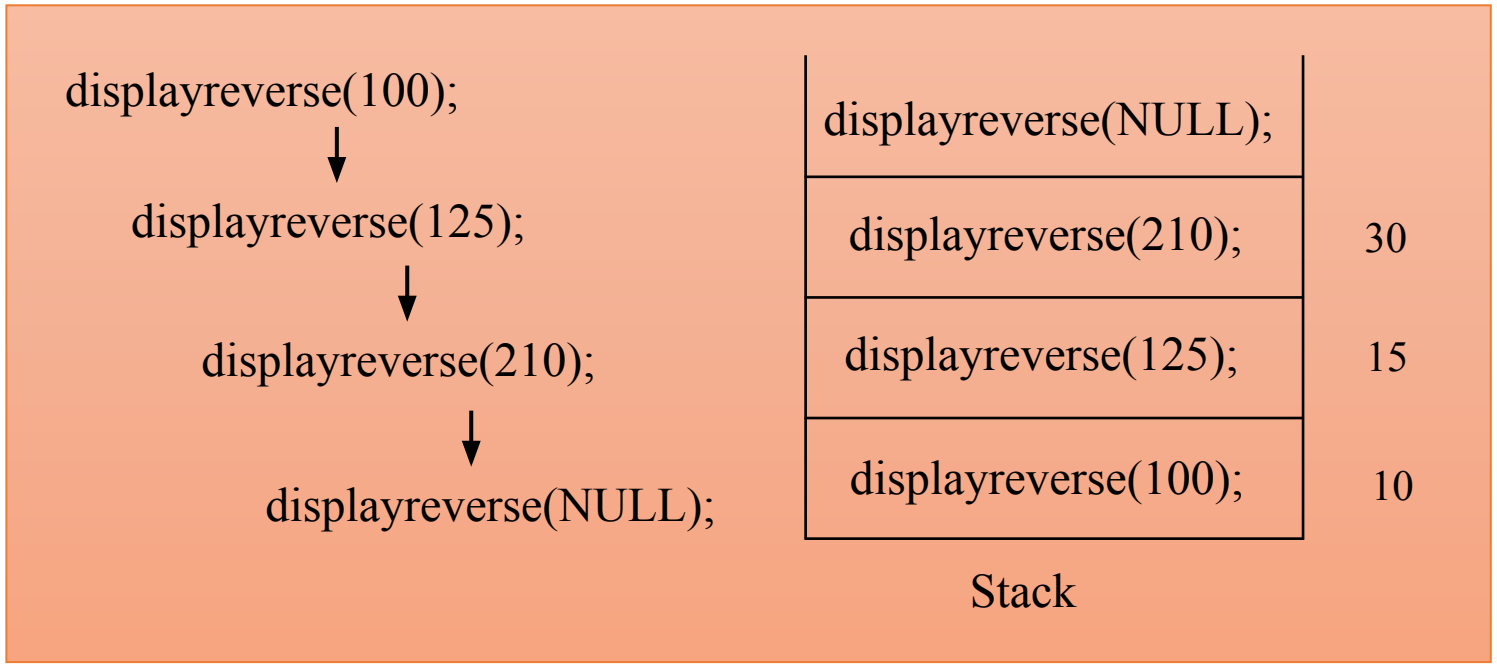
REVERSE SINGLY LINKED LIST

There are two methods to reverse a linked list:

- Iterative Method
- Recursive method

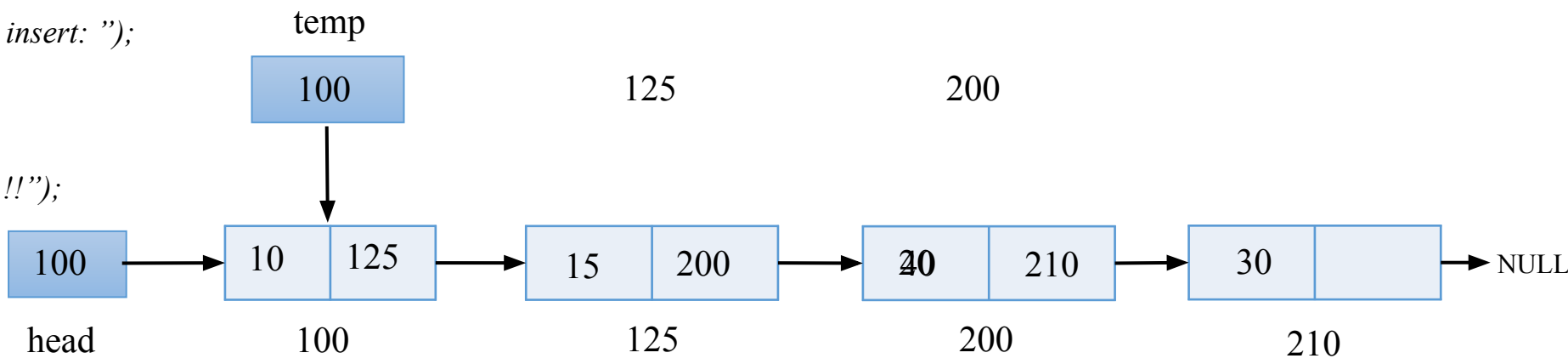
```
void displayreverse(node *head)
{
        if(head != NULL)
    {
                displayreverse(head->next);
                printf("%d", head->data);
    }
}
```





UPDATING SINGLY LINKED LIST

```
void updatesll(node *head, int old)
{
        node *temp;
        int dt;
        printf("Enter new data that you want to insert: ");
        scanf("%d", &dt);
                if(head == NULL)
                   printf("\n Linked List is Empty !!");
        else
        {
                        temp = head;
                       while(temp != NULL)
                  {
                                if(temp->data == old);
                    {
                                        temp->data = dt;
                               break;
                    }
                               temp = temp->next;
         }       }
}
```



old = 20

dt = 40

DOUBLY LINKED LIST

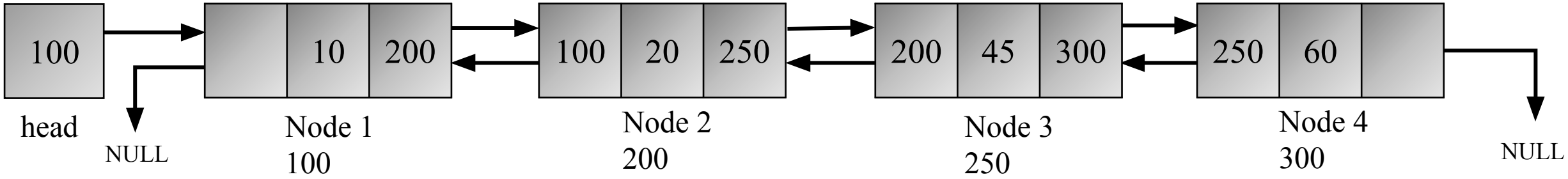
- Introduction of doubly linked list
- Creation of doubly linked list
- Displaying doubly linked list
- Displaying reverse doubly linked list
- Insertion of node in doubly linked list
- Deletion of node from doubly linked list

INTRODUCTION OF DOUBLY LINKED LIST

Doubly Linked List:

Node Structure:

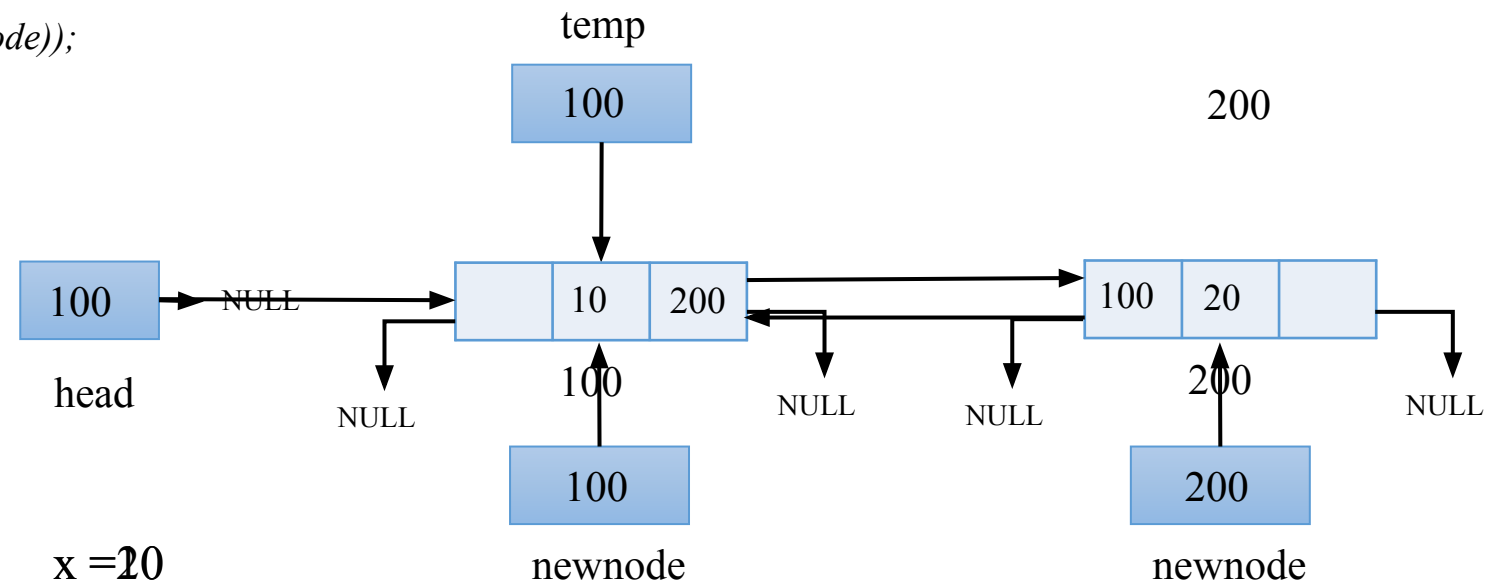
Example:




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

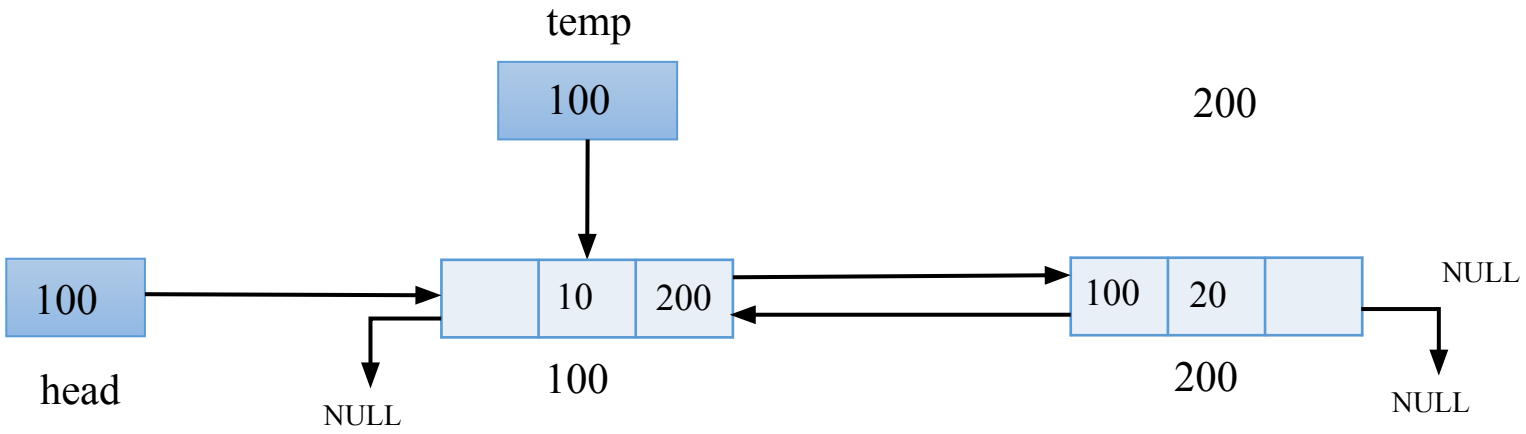
CREATION OF DOUBLY LINKED LIST

```
node *createdll( )
{
        node *head = NULL, *temp, *newnode;
    int x; char ans;
    do
    {
        printf("Enter data to node:");
        scanf("%d", &x);
                newnode = (node *)malloc(sizeof(node));
                newnode->data = x;
                newnode->next = NULL;
                newnode->prev = NULL;
                if(head  == NULL)
                        head = temp = newnode;
        else
        {
                        temp->next = newnode;
                        newnode->prev = temp;
                        temp = newnode;
        }
        printf("Do you want to add more nodes?");
        scanf("%c", &ans);
    }while(ans == 'Y');
    return head;
```



DISPLAY DOUBLY LINKED LIST

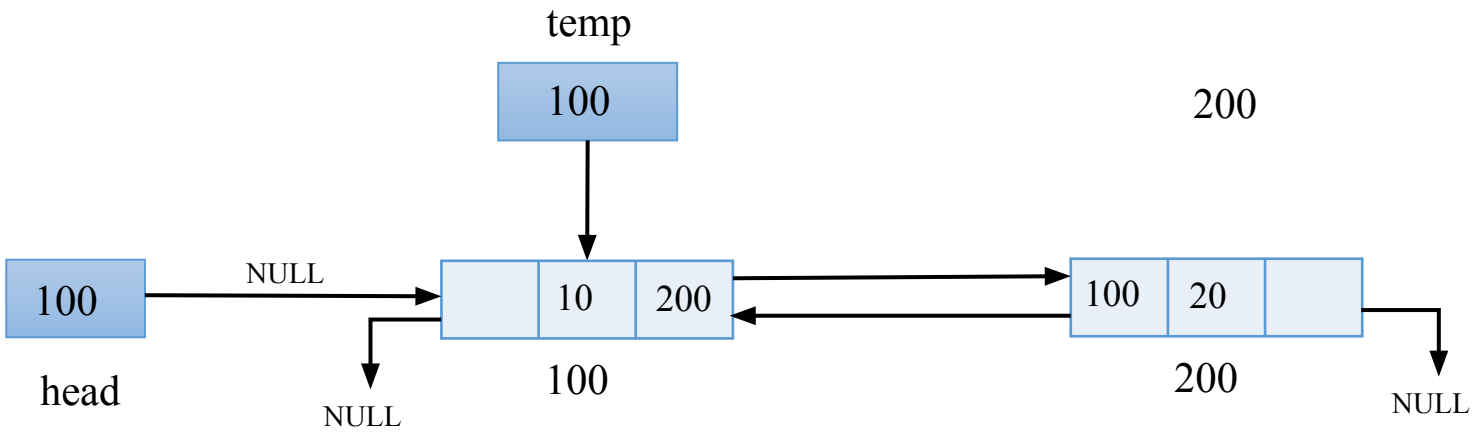
```
void display(node *head)
{
        node *temp = head;
        if(temp == NULL)
        printf("List is empty");
    else
    {
            while(temp!=NULL)
        {
                printf("%d ", temp->data);
                temp = temp->next;
        }
    }
}
```



10 20

REVERSE DISPLAY OF DOUBLY LINKED LIST

```
void displayreverse(node *head)
{
        node *temp = head;
        if(temp == NULL)
        printf("List is empty");
    else
    {
            while(temp->next != NULL)
         {
                temp = temp->next;
         }
            while(temp != NULL)
        {
                printf("%d ", temp->data);
                temp = temp->prev;
        }
    }
}
```



20 10

INSERTION OF NODE IN DOUBLY LINKED LIST

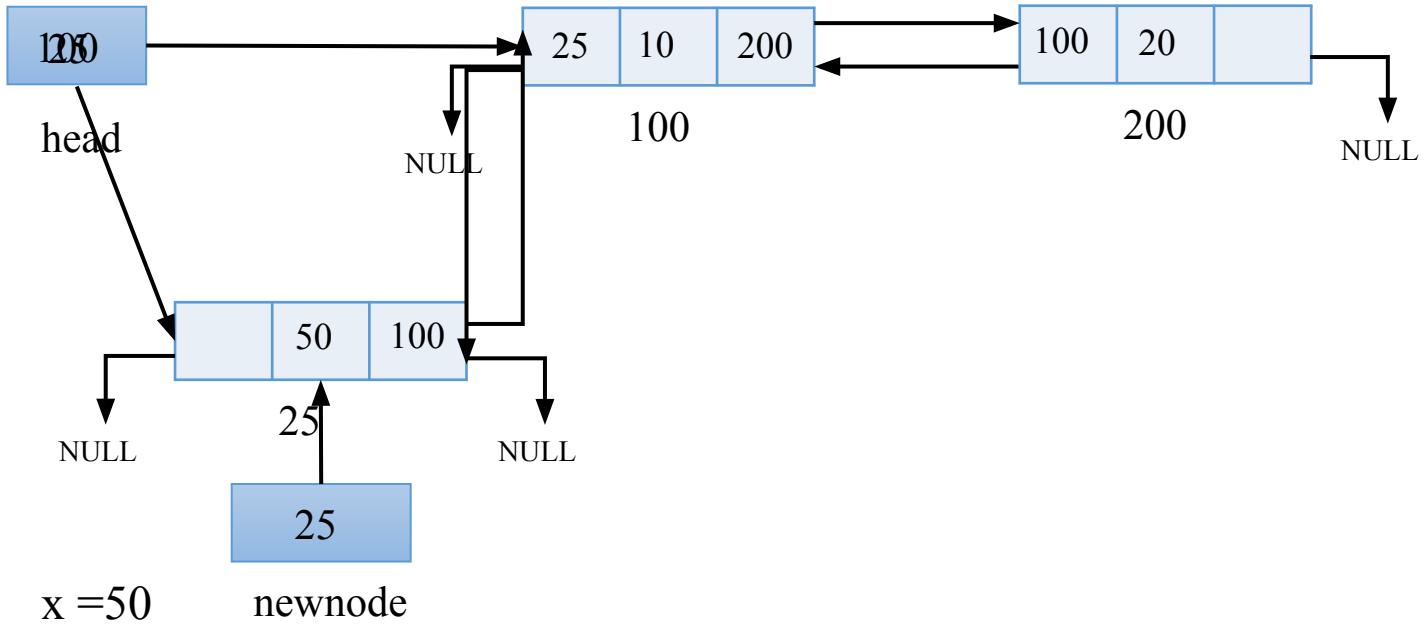
There are three ways of inserting a node in a doubly linked list –

- Insertion of a node at the beginning
- Insertion of a node in a middle
- Insertion of a node at the end

INSERTION OF NODE IN DOUBLY LINKED LIST

- Insertion of a node at the beginning

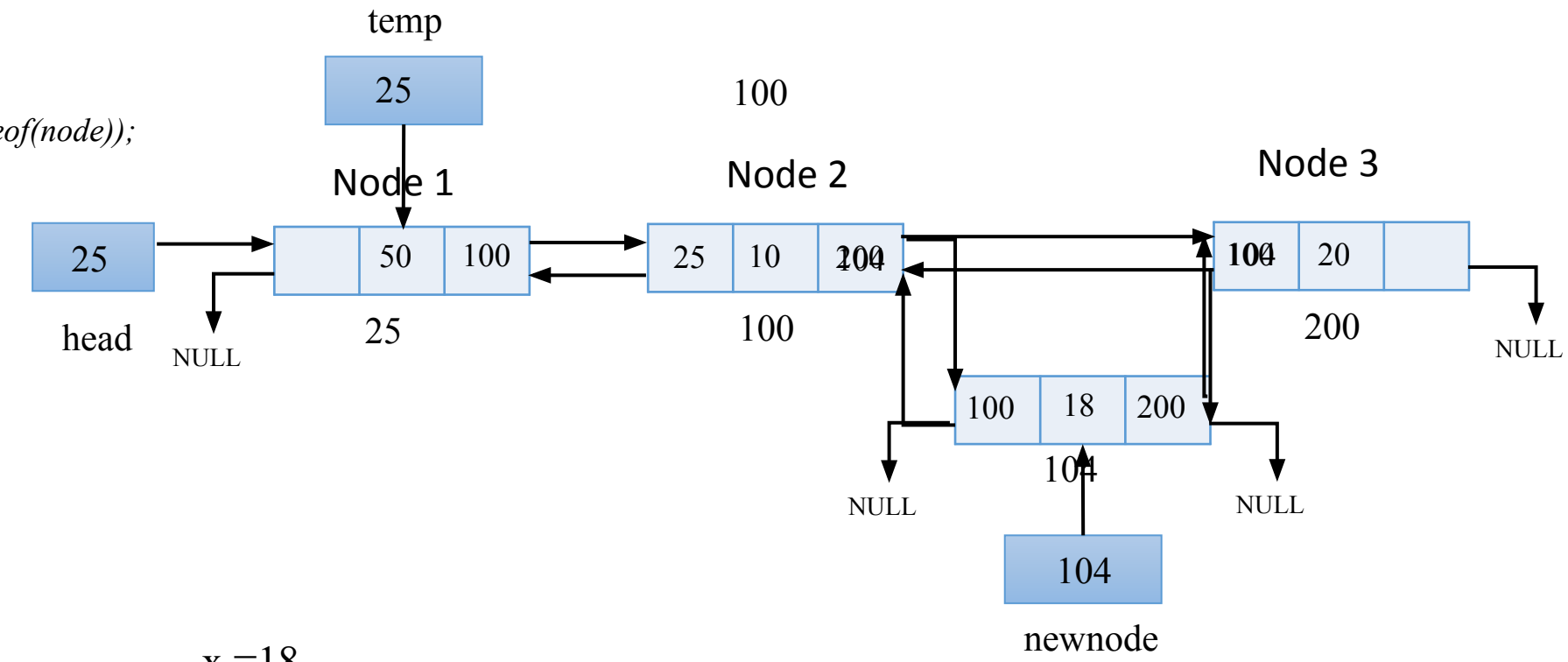
```
node *insertatbeginning(node *head)
{
        node *newnode;
        int x;
        printf("Enter data to node:");
        scanf("%d", &x);
                newnode = (node *)malloc(sizeof(node));
                newnode->data = x;
                newnode->next = NULL;
                newnode->prev = NULL;
                if(head == NULL)
        {
                        head = newnode;
        }
        else
        {
                        newnode->next = head;
                        head->prev = newnode;
                        head = newnode;
        }
}
```



INSERTION OF NODE IN DOUBLY LINKED LIST

- Insertion of a node in the middle

```
node *insertmiddlenode(node *head, int loc)
{
        node *newnode;
        int x, i=1;
        printf("Enter data to node:");
        scanf("%d", &x);
                newnode = (node *)malloc(sizeof(node));
                newnode->data = x;
                newnode->next = NULL;
                newnode->prev = NULL;
                temp = head;
                while(i< (loc-1))
        {
                        temp = temp->next;
                i++;
        }
                newnode->next = temp->next;
                newnode->prev = temp;
                newnode->next->prev = newnode;
                temp->next = newnode;
}
```



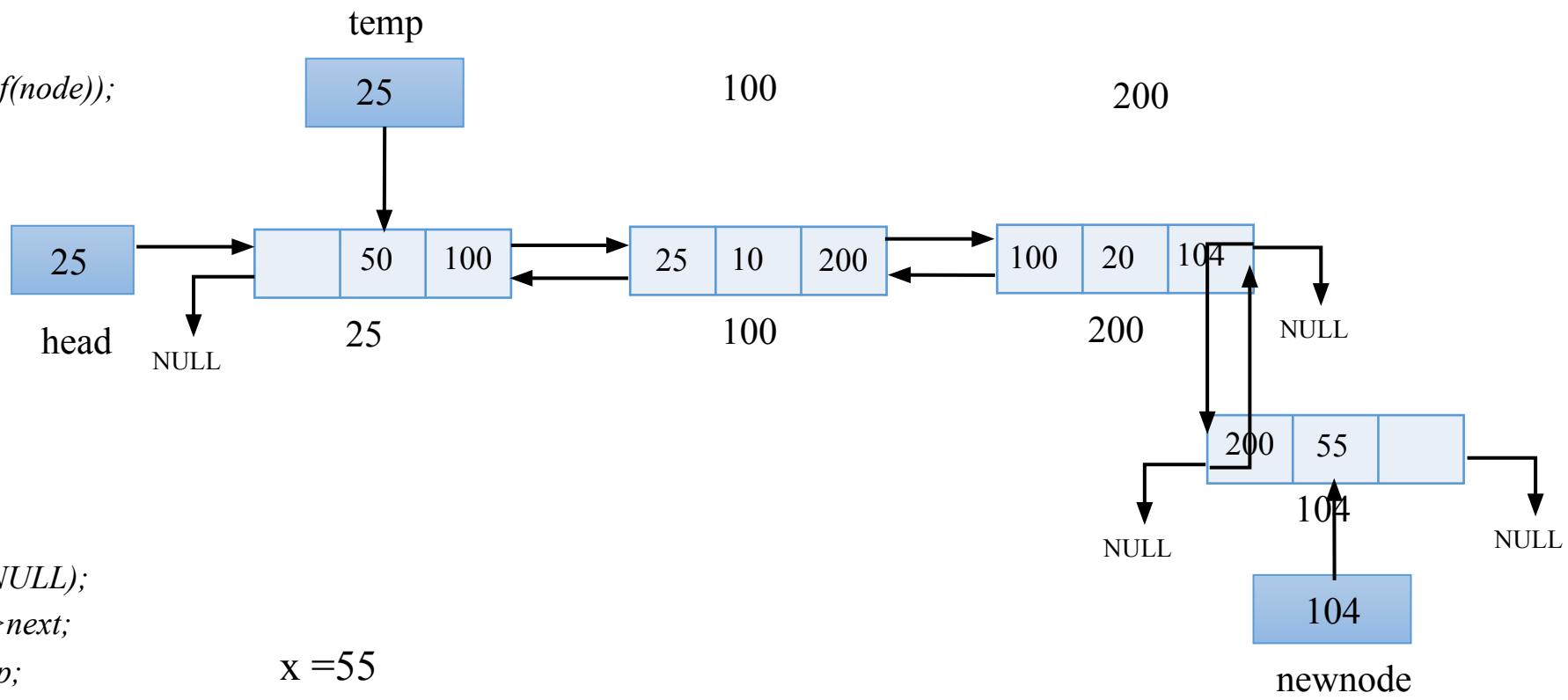
x =18

Loc = 3

INSERTION OF NODE IN DOUBLY LINKED LIST

- Insertion of a node at the end

```
node *insertnodeatend(node *head)
{
        node *newnode;
        int x;
        printf("Enter data to node:");
        scanf("%d", &x);
                newnode = (node *)malloc(sizeof(node));
                newnode->data = x;
                newnode->next = NULL;
                newnode->prev = NULL;
                temp = head;
                if(head == NULL)
        {
                        head = newnode;
        }
        else
        {
                        while(temp->next != NULL);
                                temp = temp->next;
                        newnode->prev = temp;
                        newnode->next = temp->next;
                        temp->next = newnode;
        }
}
```



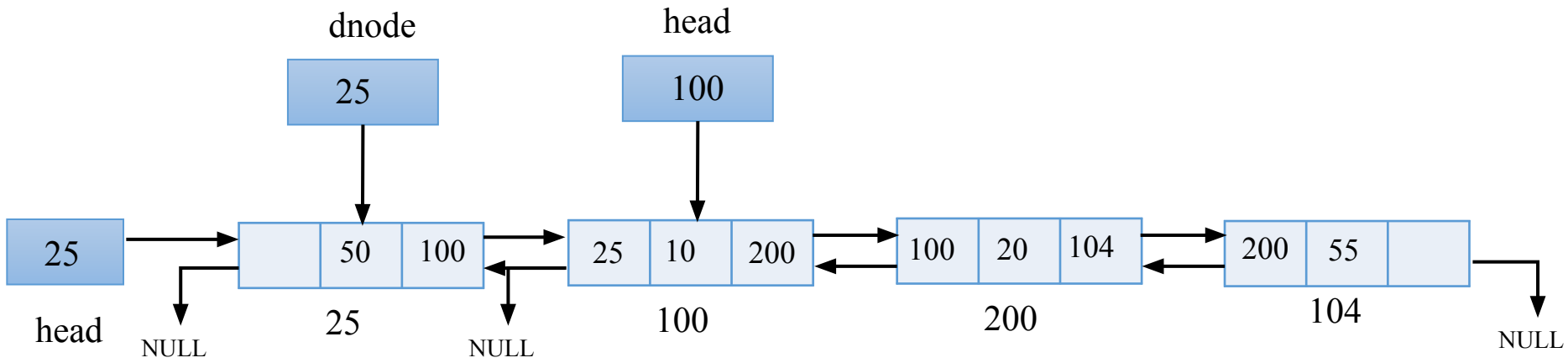
DELETION OF NODE FROM DOUBLY LINKED LIST

There are three ways of deleting a node from a linked list –

- Delete the first node
- Delete node from middle of linked list
- Delete the last node

Deletion of first node

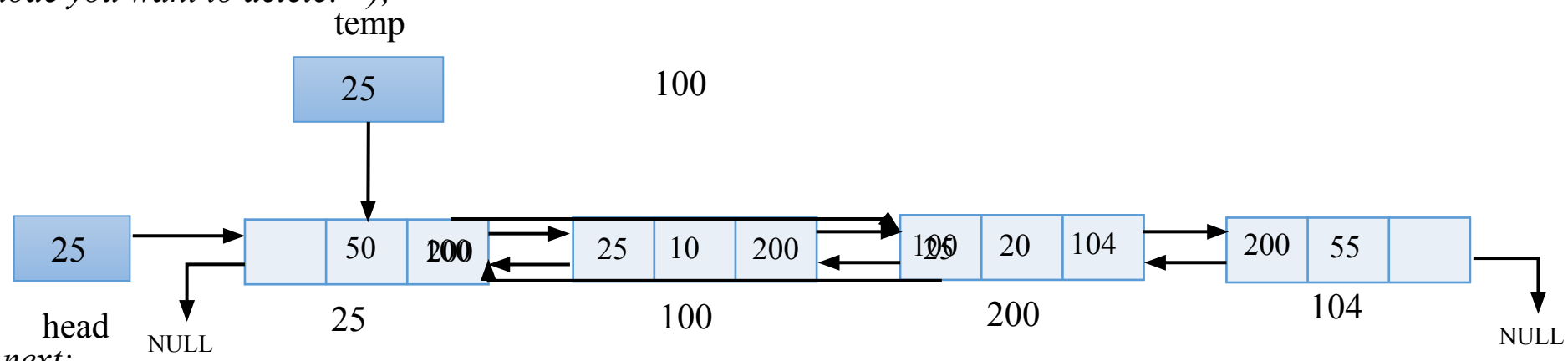
```
node *deletefirstnode(node *head)
{
        node *dnode;
                dnode = head;
                head = head ->next;
                head ->prev = NULL;
                free(dnode);
                return(head);
}
```



DELETION OF NODE FROM DOUBLY LINKED LIST

- Deletion of middle node

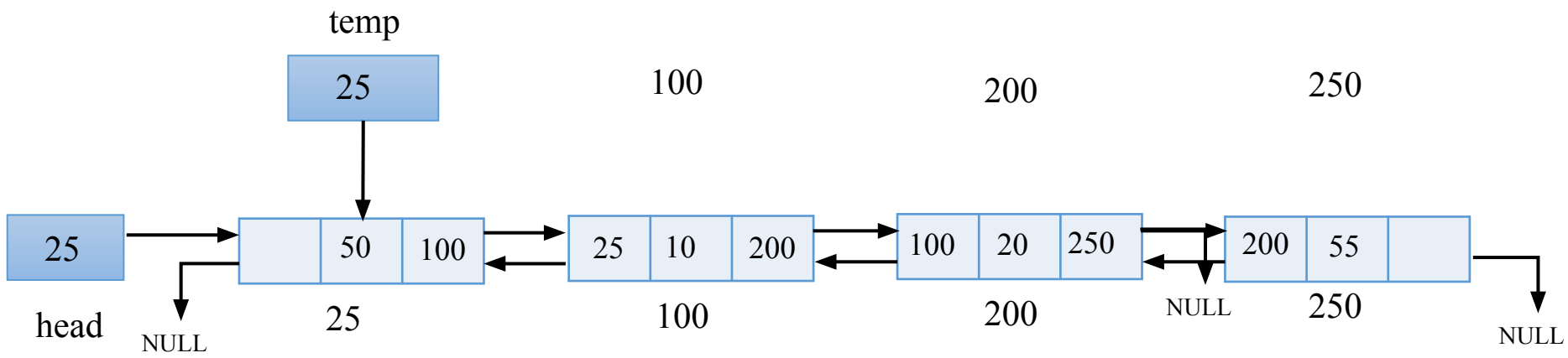
```
node *deletemiddlenode(node *head)
{
        node *temp;
        int loc, i=1;
        printf("Enter the location of the node you want to delete: ");
        scanf("%d", &loc);
                temp = head;
                while(i< (loc))
        {
                        temp = temp->next;
                  i++;
        }
                temp->prev->next = temp->next;
                temp->next->prev = temp->prev;
                free(temp);
                return(head);
}
```



DELETION OF NODE FROM DOUBLY LINKED LIST

- Deletion of last node

```
node *deletelastnode(node *head)
{
        node *temp;
                temp = head;
                while(temp->next != NULL)
        {
                        temp = temp->next;
        }
                temp->prev->next = NULL
                free(temp);
                return(head);
}
```



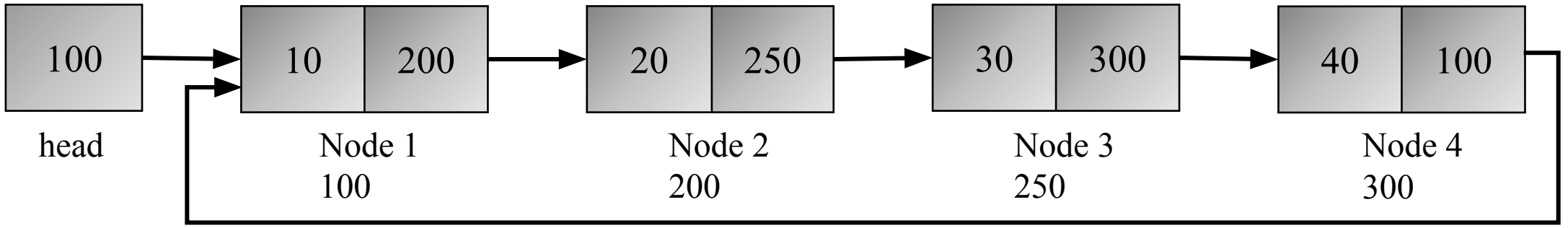
CIRCULAR LINKED LIST

- Introduction
- Creation of circular linked list
- Insertion of node in circular linked list
- Deletion of node from circular linked list

INTRODUCTION

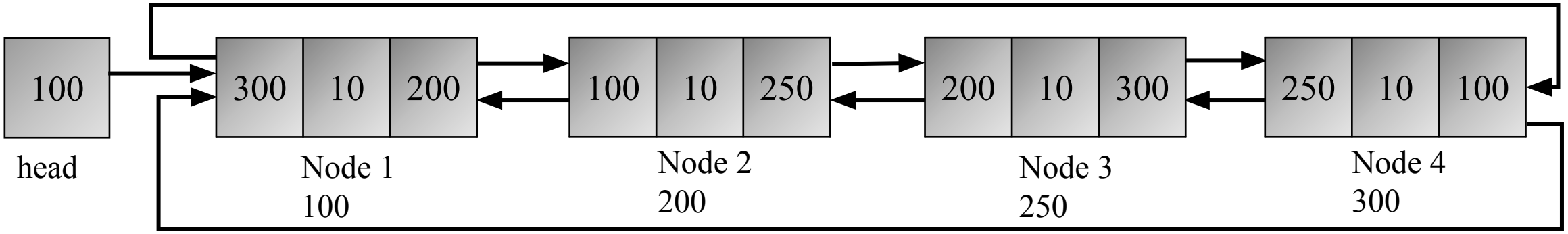
Circular Linked List:

Example:



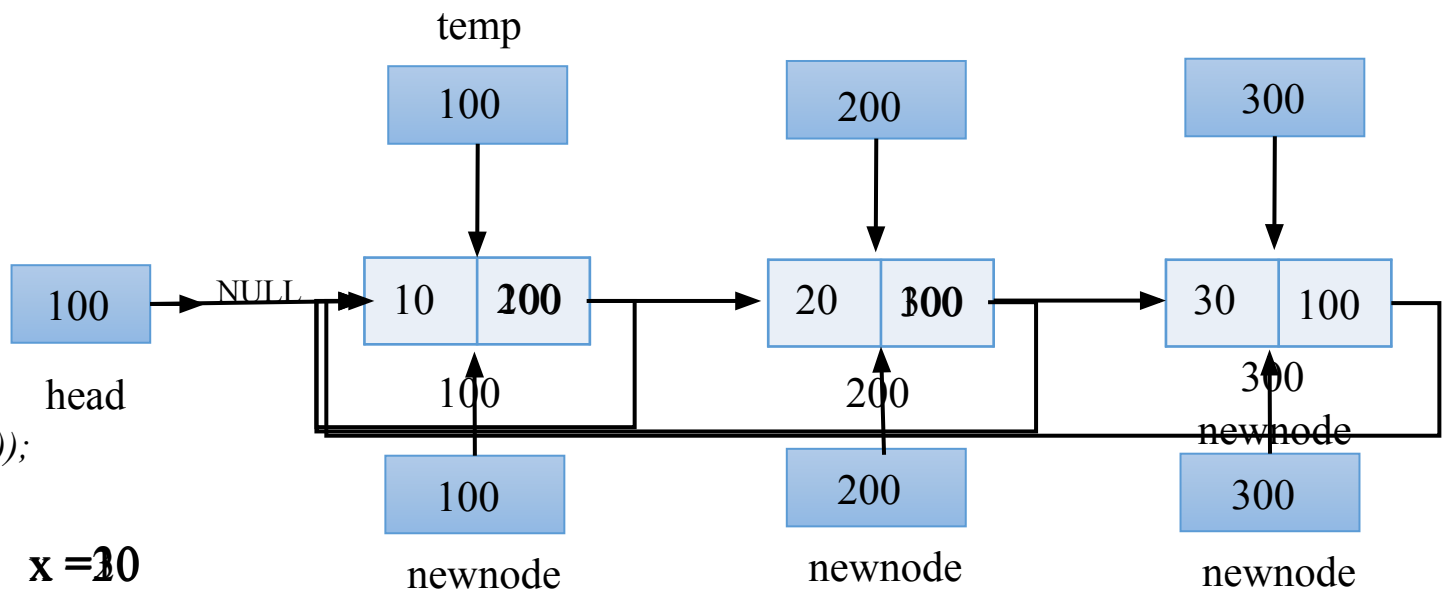
Doubly Circular Linked List:

Example:



CREATION OF CIRCULAR LINKED LIST

```
node *createcll( )
{
        node *head = NULL, *temp, *newnode;
    int x; char ans;
    do
    {
        printf("Enter data to node:");
        scanf("%d", &x);
                newnode = (node *)malloc(sizeof(node));
                newnode->data = x;
                if(head  == NULL)
                        head = temp = newnode;
        else
        {
                        temp->next = newnode;
                        temp = newnode;
        }
                temp->next = head;
        printf("Do you want to add more nodes?");
        scanf("%c", &ans);
    }while(ans == 'Y');
    return head;
}
```

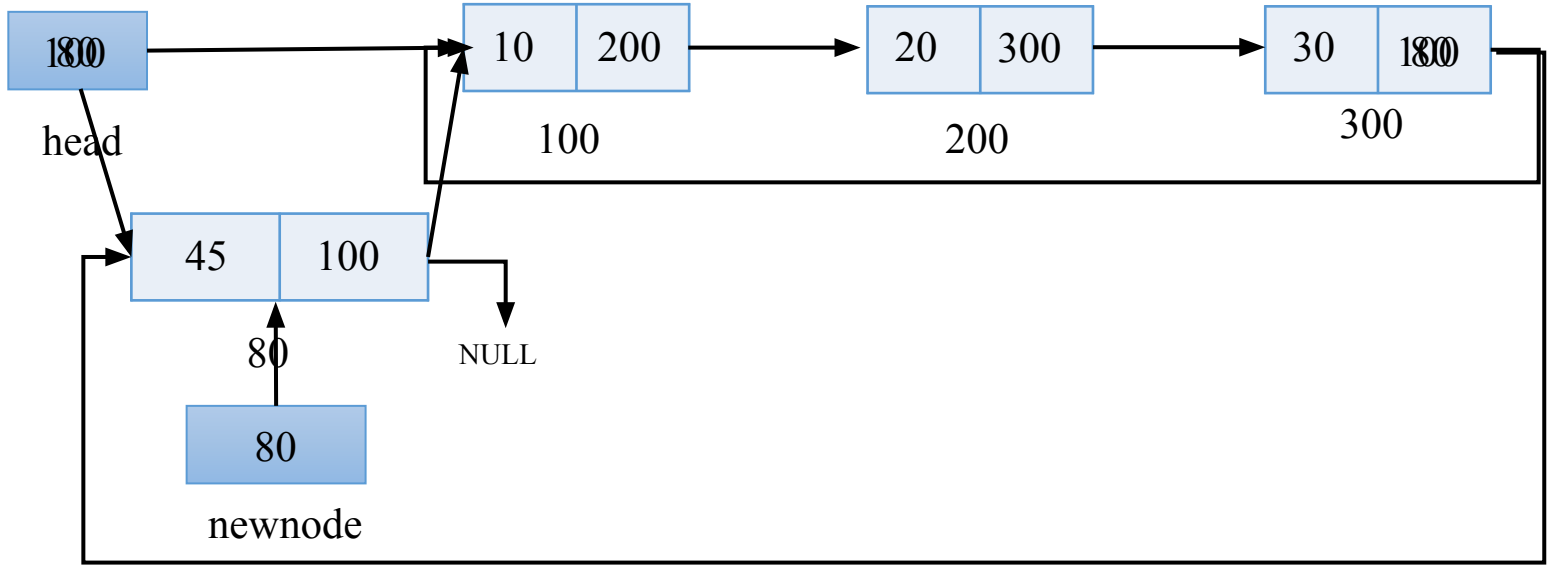


INSERTION OF CIRCULAR LINKED LIST

There are three ways of inserting a node in a linked list –

- Insertion of a node at the beginning
- Insertion of a node in a middle
- Insertion of a node at the end

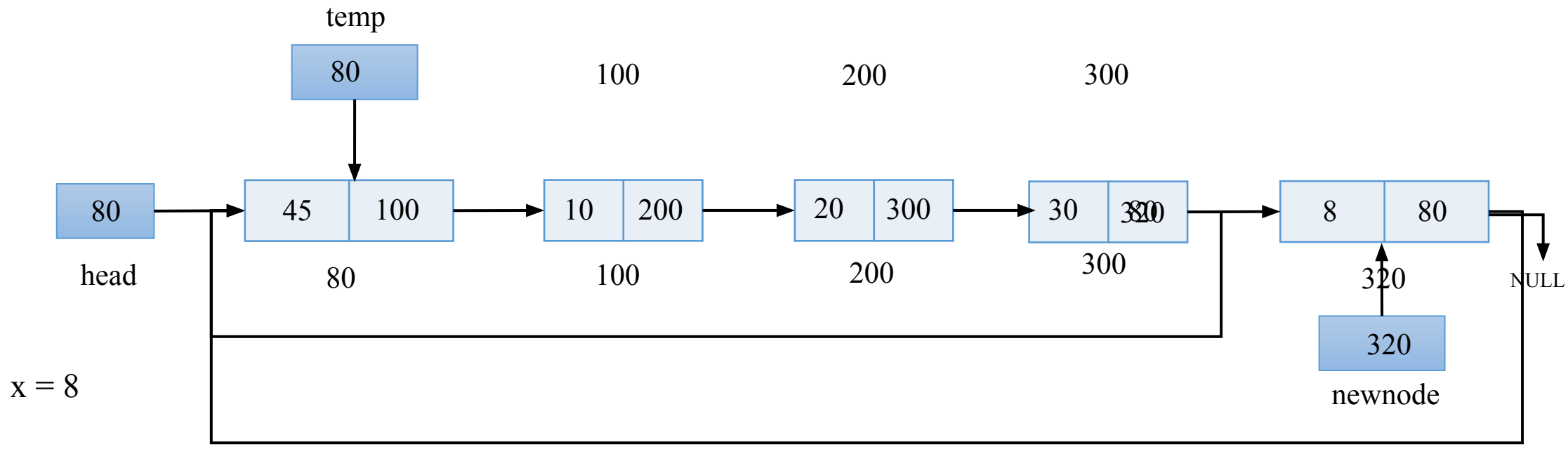
x = 45



INSERTION OF CIRCULAR LINKED LIST

There are three ways of inserting a node in a linked list –

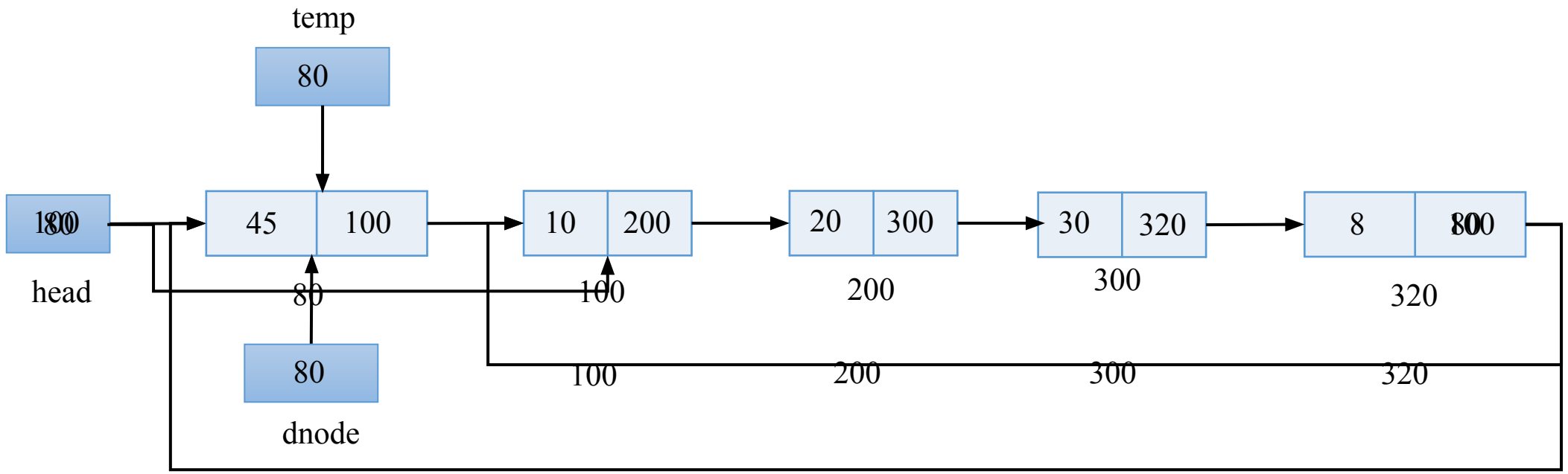
- Insertion of a node at the beginning
- Insertion of a node in a middle
- Insertion of a node at the end



DELETION OF CIRCULAR LINKED LIST

There are three ways of deleting a node from a linked list –

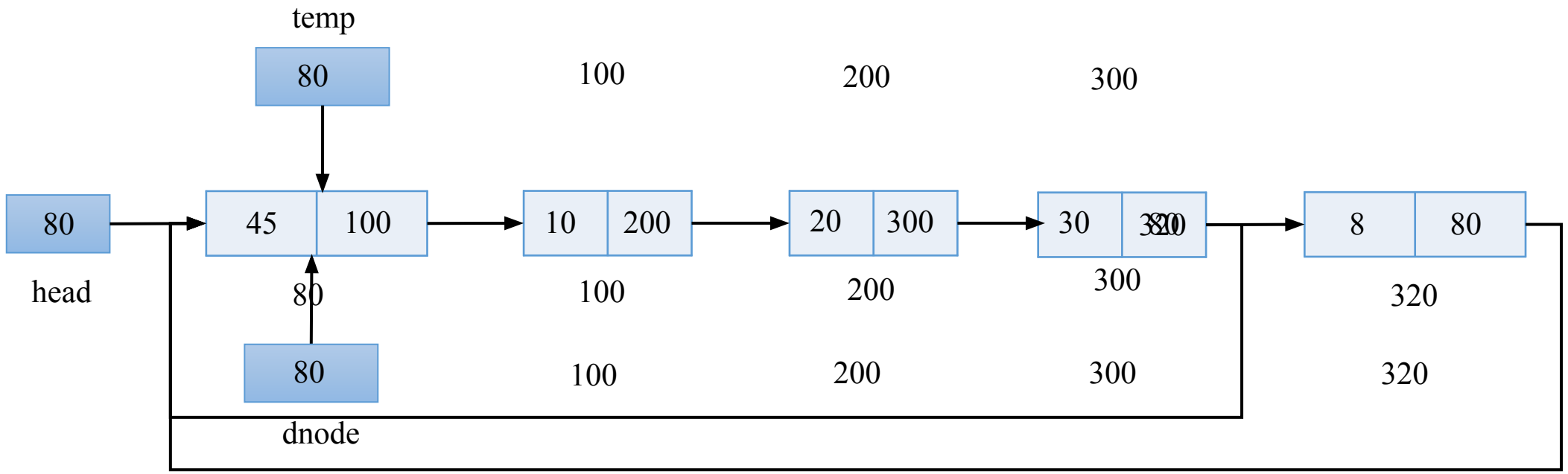
- Delete the first node
- Delete node from middle of linked list
- Delete the last node



DELETION OF CIRCULAR LINKED LIST

There are three ways of deleting a node from a linked list –

- Delete the first node
- Delete node from middle of linked list
- Delete the last node



APPLICATIONS OF LINKED LISTS

Applications in Computer Science:

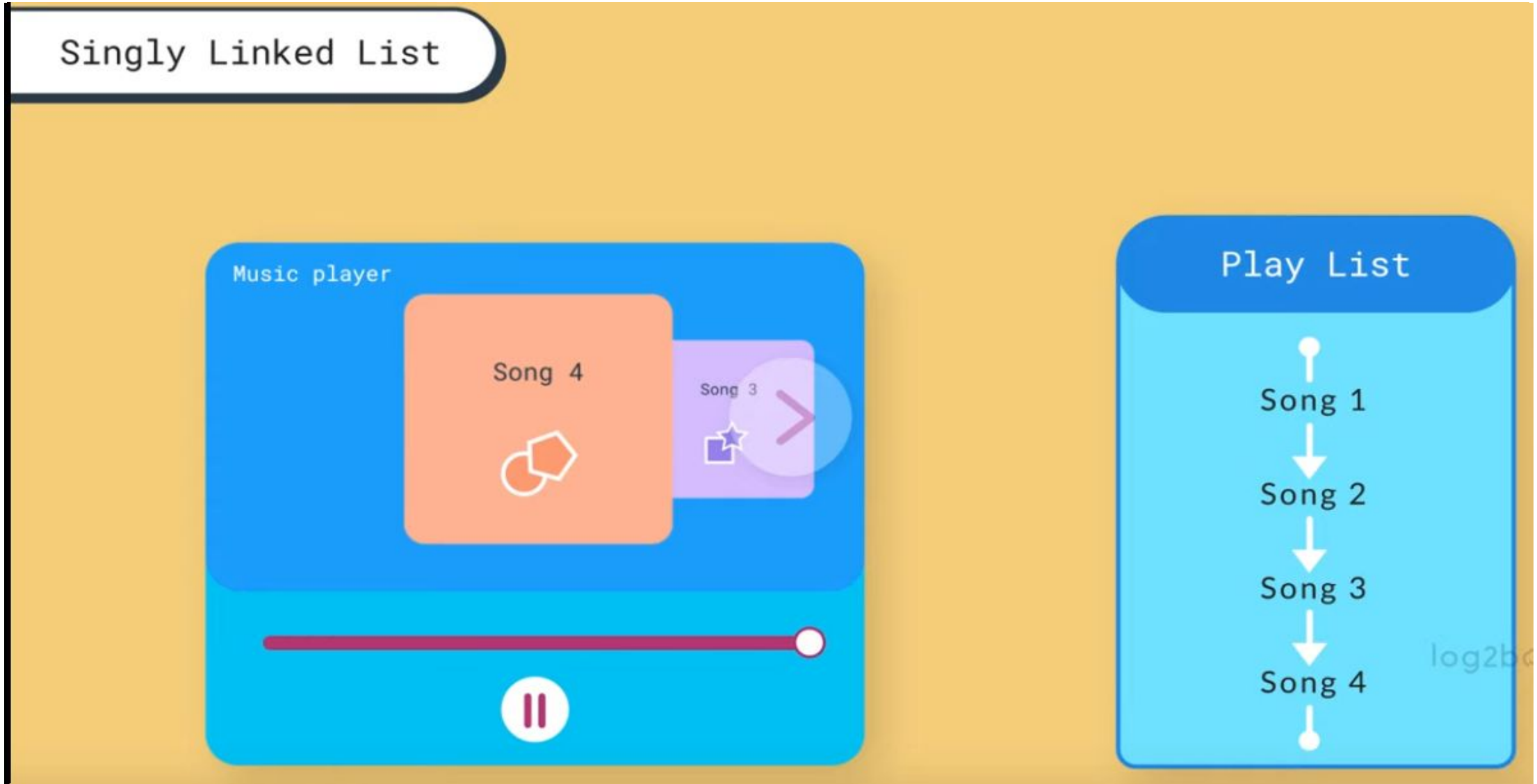
- Implementation of stacks and queues
- Implementation of graphs : Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices
- Maintaining directory of names
- Manipulation of polynomials by storing constants in the node of linked list

Applications in real world:

- Image viewer – Previous and next images are linked, hence can be accessed by next and previous button
- Previous and next page in web browser – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list
- Music Player – Songs in music player are linked to previous and next song, you can play songs either from starting or ending of the list

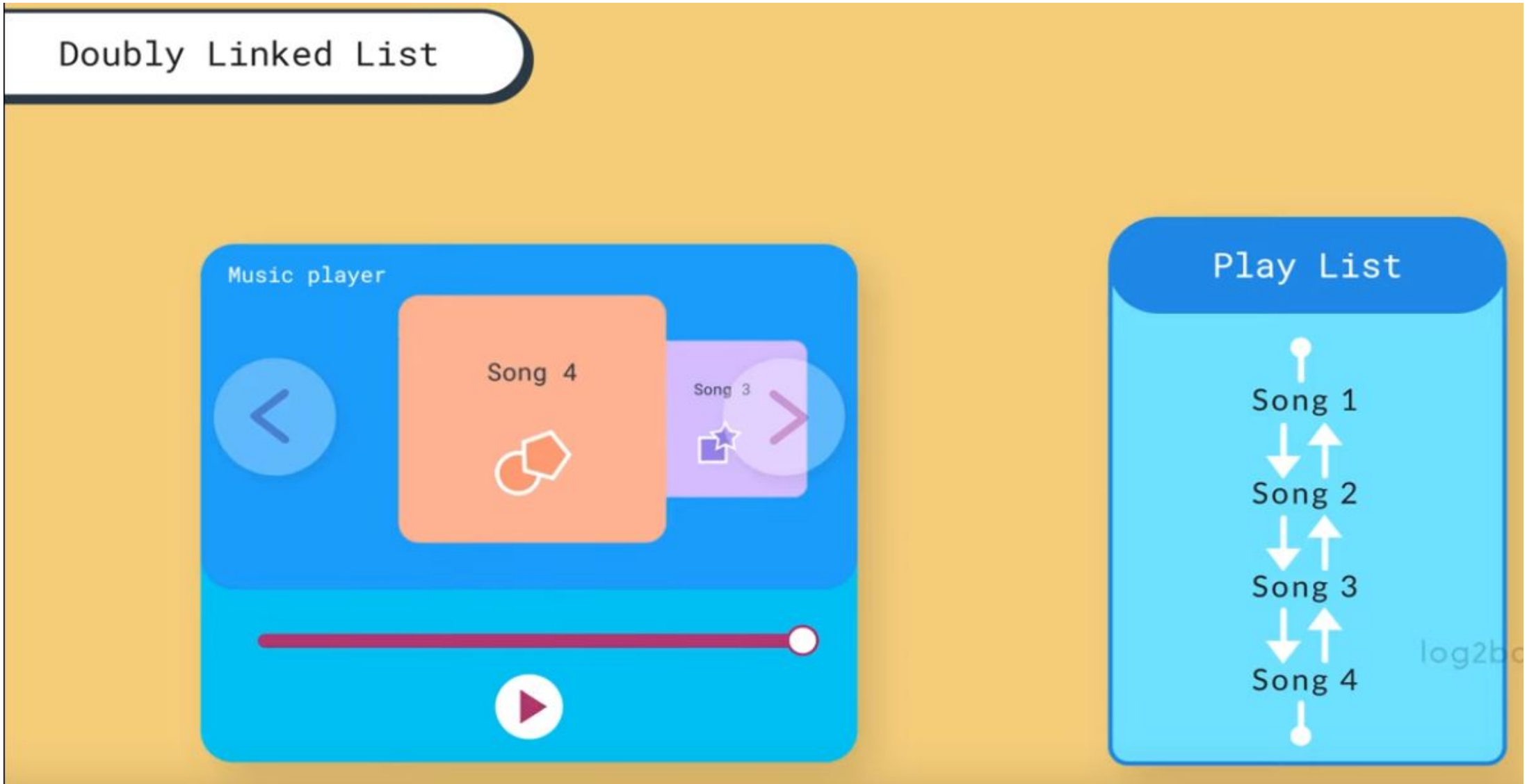
APPLICATION OF SINGLY LINKED LISTS

Music Player



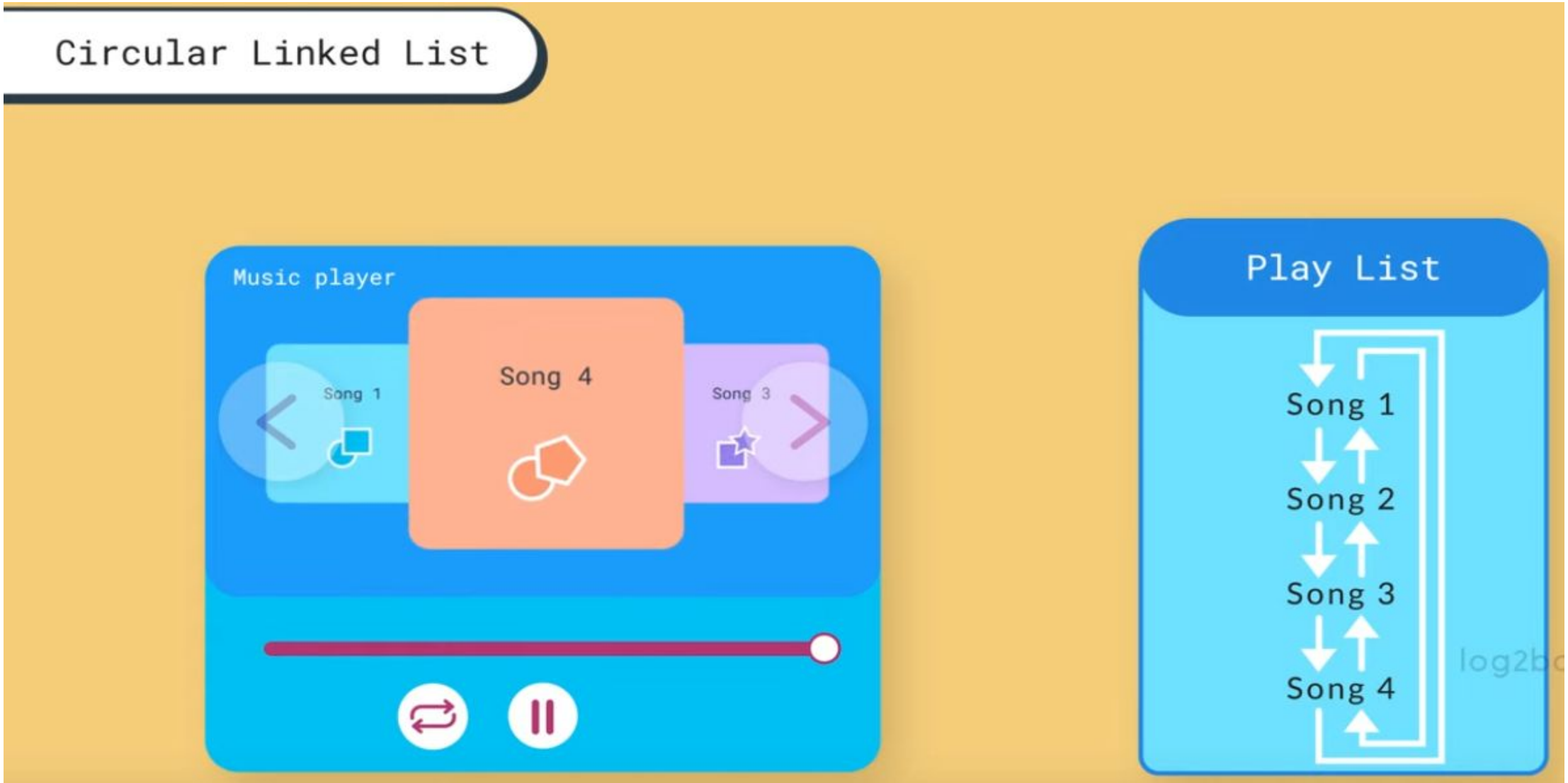
APPLICATION OF DOUBLY LINKED LISTS

Music Player



APPLICATION OF CIRCULAR LINKED LISTS

Music Player

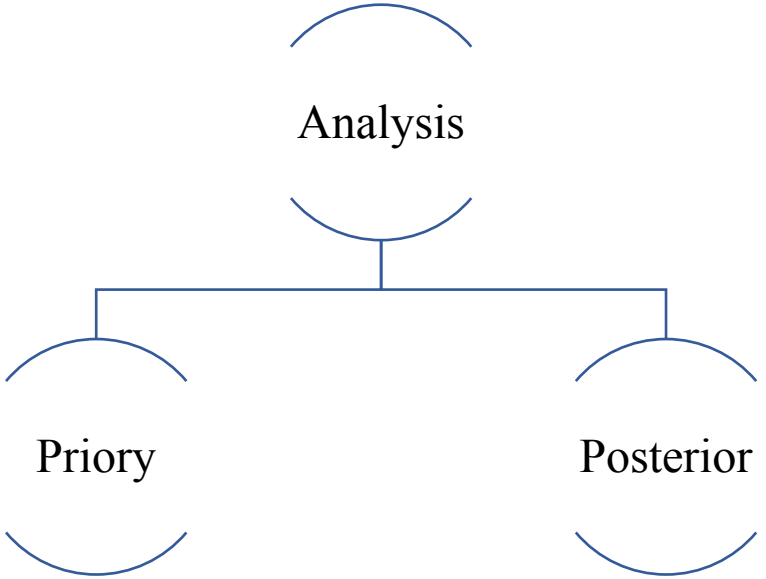


ANALYSIS OF ALGORITHMS

The analysis of algorithm is the process of finding the computational complexity of algorithm – the amount of time, storage, or other resources needed to execute them

Time complexity: Running time of the program. Calculating the time required for each step

Space Complexity: how much space an algorithm needs to complete its task



ANALYSIS OF ALGORITHMS

Time Complexity:

Asymptotic Notation:

- Mathematical way of representing the time complexity
- It is a technique of representing limiting behavior
- Asymptotic notations are used to write fastest and slowest possible running time for an algorithm

Types of analysis -

- Worst-case: f (n) defined by the maximum number of steps taken on any instance of size n

  Big – Oh Notation (O)

- Best-case: f (n) defined by the minimum number of steps taken on any instance of size n

  Omega Notation (Ω)

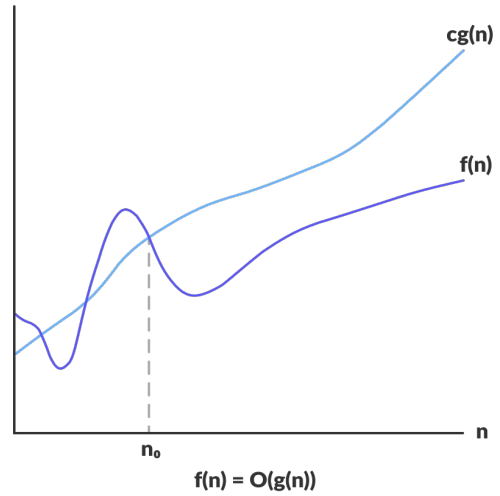
- Average case: f (n) defined by the average number of steps taken on any instance of size n

  Theta Notation (θ)

ANALYSIS OF ALGORITHMS

Asymptotic Notations:

Big – Oh Notation (O)



$f(n) = O\,g(n)$

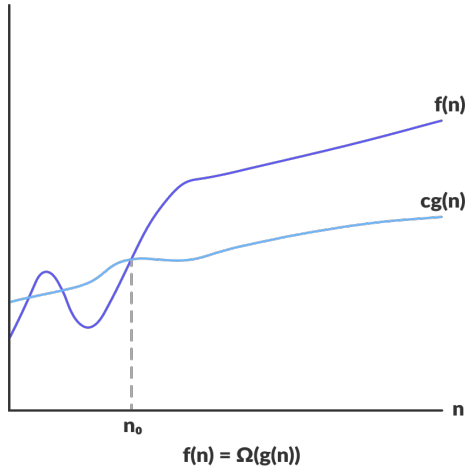
$f(n) \leq c\,.\,g(n)$

Example: $2n^2 + n \leq 3n^2$

$n \leq n^2$

for all $n \geq 1$

Omega Notation (Ω)



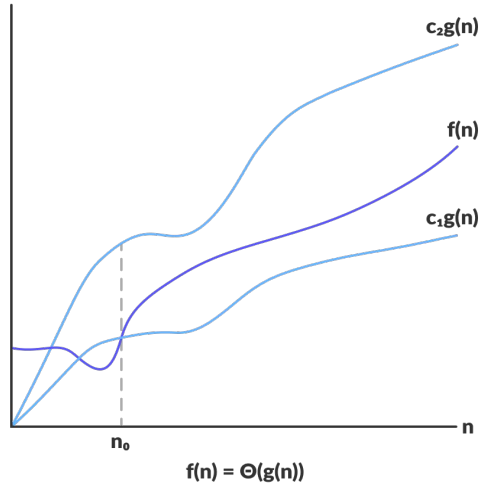
$f(n) = O\,g(n)$

$f(n) \geq c\,.\,g(n)$

Example: $2n^2 + n \geq 2n^2$

for all $n \geq 0$

Theta Notation (θ)



$f(n) = O\,g(n)$

$c1.g(n) \leq f(n) \geq c2\,.\,g(n)$

Example: $2n^2 \leq 2n^2 + n \leq 3n^2$

for all $n \geq 1$

ANALYSIS OF ALGORITHMS

Each instruction affects the overall performance of an algorithm

❖ $O(1)$: Constant

Time complexity of a function is considered as $O(1)$ if it doesn't contain loop, recursion and call to any other non-constant time function

❖ $O(n)$: Linear

Time Complexity of a loop is considered as $O(n)$ if the loop variables is incremented / decremented by a constant amount

❖ $O(n^x)$: Quadratic / Cubic

Time complexity of nested loops is equal to the number of times the innermost statement is executed

❖ $O(\log n)$: Logarithmic

Time Complexity of a loop is considered as $O(\log n)$ if the loop variables is divided by a constant amount

❖ $O(n \log n)$: Logarithmic

Time complexity of a loop is considered as $O(n \log n)$ when a set of data is repeatedly divided into half and each half is processed again independently

ANALYSIS OF ARRAY

Array elements are stored continuously in memory, so the time required to compute the memory address of an array element arr[i] is independent of the array's size: It's the *start address* of arr plus i * (size of an individual element)

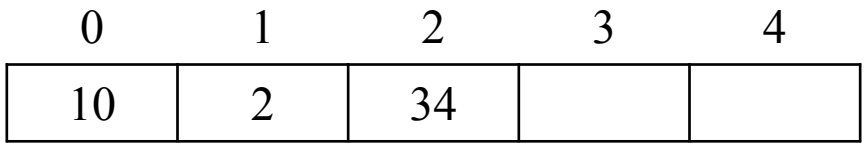
Assume n elements are there in array

Insert:

At the end (Best Case): $O(1)$ - Just insert element at an index

At the beginning (Worst Case): $O(n)$ - adding an element to the first location: all n elements in the array have to be shifted one place to the right before the new element can be added

In the Middle (Average Case): $O(n)$ - Adding an element in the middle so rest of the elements need to be shifted one place to the right before the new element can be added



Array

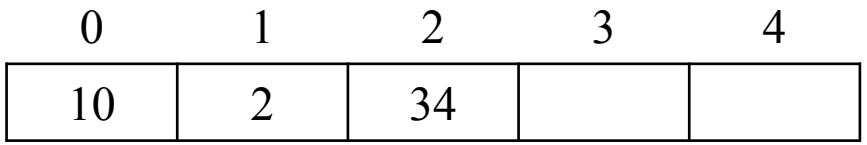
ANALYSIS OF ARRAY

Delete:

At the end (Best Case): $O(1)$ - Just delete last element from the index

At the beginning (Worst Case): $O(n)$ - deleting first element: all n elements in the array have to be shifted one place to the left after the first element is deleted

In the Middle (Average Case): $O(n)$ - deleting middle element: so rest of the elements need to be shifted one place to the left after the middle element is deleted



Array

Access / Update:

Any element can be accessed with index so access time: $O(1)$

Any element can be updated with index given: $O(1)$

Search:

For searching we need to start from first index till you find element: $O(n)$

ANALYSIS OF SINGLY LINKED LISTS

Insert:

- Insertion of a node at the beginning (Best Case) : $O(1)$
- Insertion of a node at the end (Worst Case) - Need to traverse the linked list till the end : $O(n)$

  If the tail pointer is pointing to last node then Insertion of a node at the end : $O(1)$
- Insertion of a node in the middle (Average Case) - Need to traverse the linked list till the position of the node : $O(n)$

Delete:

- Deletion of a node at the beginning (Best Case) : $O(1)$
- Deletion of a node at the end (Worst Case) - Need to traverse the linked list till the end to find the node to be deleted: $O(n)$
- Deletion of a node in the middle (Average Case) - Need to traverse the linked list to find the node till the position of the node : $O(n)$

ANALYSIS OF SINGLY LINKED LISTS

Access:

- Accessing the First node (Best Case) : $O(1)$
- Accessing the Last node (Worst Case) - Need to traverse the linked list till the end : $O(n)$

  If the tail pointer is pointing to last node then Accessing the First node : $O(1)$
- Accessing the Middle node (Average Case) - Need to traverse the linked list till the position of the node : $O(n)$

Search / Find:

- Find the First node (Best Case) : $O(1)$
- Find the Last node (Worst Case) - Need to traverse the linked list till the end : $O(n)$

  If the tail pointer is pointing to last node then Accessing the First node : $O(1)$
- Find the Middle node (Average Case) - Need to traverse the linked list till the node : $O(n)$

ANALYSIS OF DOUBLY LINKED LISTS

Insert:

- Insertion of a node at the beginning (Best Case) : $O(1)$
- Insertion of a node at the end (Worst Case) - Need to traverse the linked list till the end : $O(n)$

  If the tail pointer is pointing to last node then Insertion of a node at the end : $O(1)$
- Insertion of a node in the middle (Average Case) - Need to traverse the linked list till the position of the node : $O(n)$

Delete:

- Deletion of a First node: (Best Case) : $O(1)$
- Deletion of a Last node: (Worst Case) - Need to traverse the linked list till the end to find the node to be deleted: $O(n)$

  If the tail pointer is pointing to last node then deletion of last node : $O(1)$
- Deletion of a Middle node: (Average Case) - Need to traverse the linked list to find the node till the position of the node : $O(n)$

ANALYSIS OF DOUBLY LINKED LISTS

Access:

- Accessing the First node (Best Case) : $O(1)$
- Accessing the Last node (Worst Case) - Need to traverse the linked list till the end : $O(n)$

  If the tail pointer is pointing to last node then Accessing the First node : $O(1)$
- Accessing the Middle node (Average Case) - Need to traverse the linked list till the position of the node : $O(n)$

Search / Find:

- Find the First node (Best Case) : $O(1)$
- Find the Last node (Worst Case) - Need to traverse the linked list till the end : $O(n)$

  If the tail pointer is pointing to last node then Accessing the First node : $O(1)$
- Find the Middle node (Average Case) - Need to traverse the linked list till the node : $O(n)$