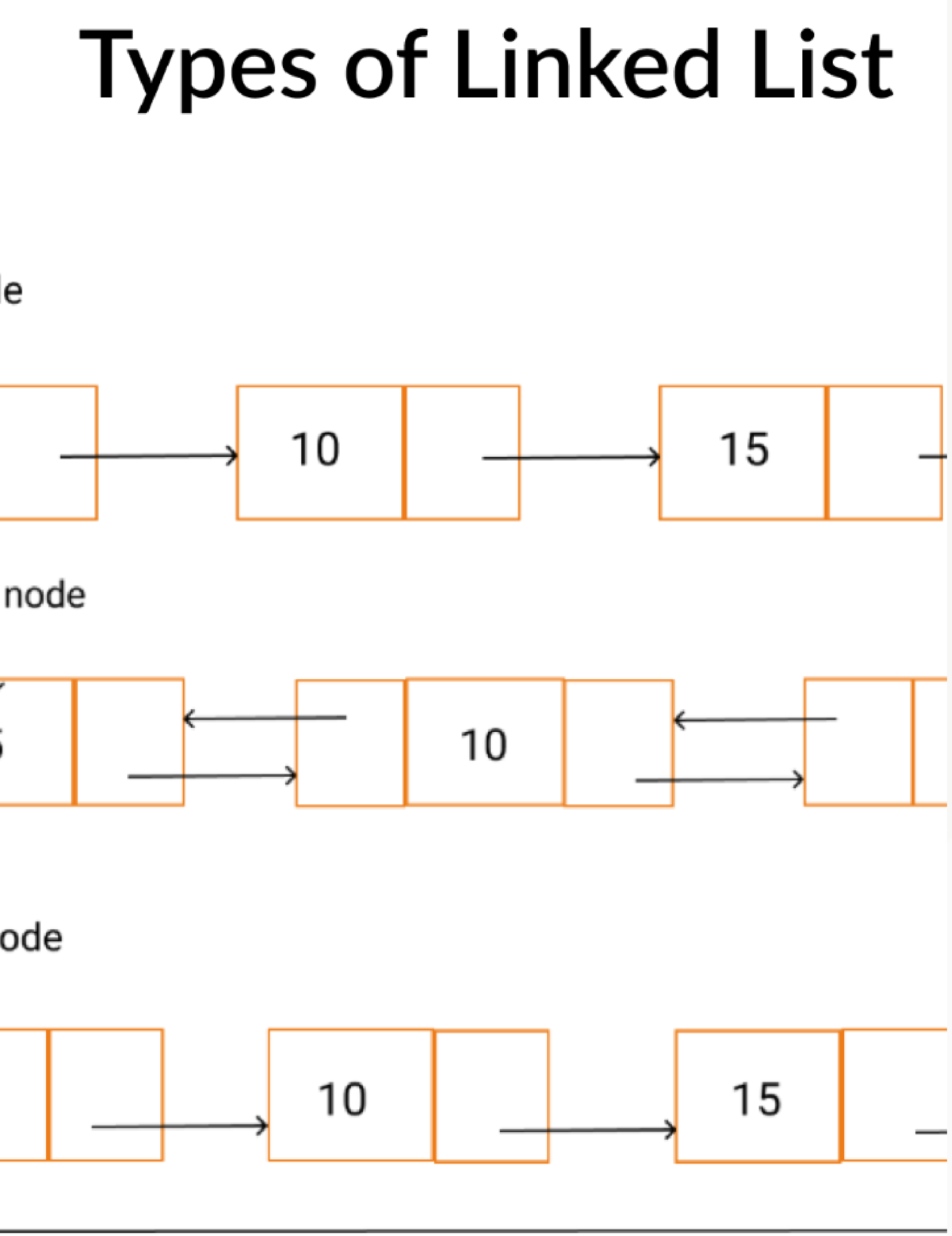


Linked List

Dr. GC Jana

Definition of a Linked List

1 A Sequential Collection

A linked list is a linear data structure comprised of nodes that are connected through pointers, forming a chain-like sequence.

2 Flexible and Dynamic

Unlike arrays, linked lists can grow or shrink dynamically by adding or removing nodes at any position.

3 Elementary Components

A node typically contains two key elements: the data it stores and a reference to the next node in the list.

Advantages of Using a Linked List

Efficient Insertions and Deletions

Linked lists enable efficient insertion and deletion operations by simply adjusting the pointers, without requiring data movement.

Dynamic Size

The ability to grow or shrink dynamically makes linked lists ideal for scenarios where the number of elements changes frequently.

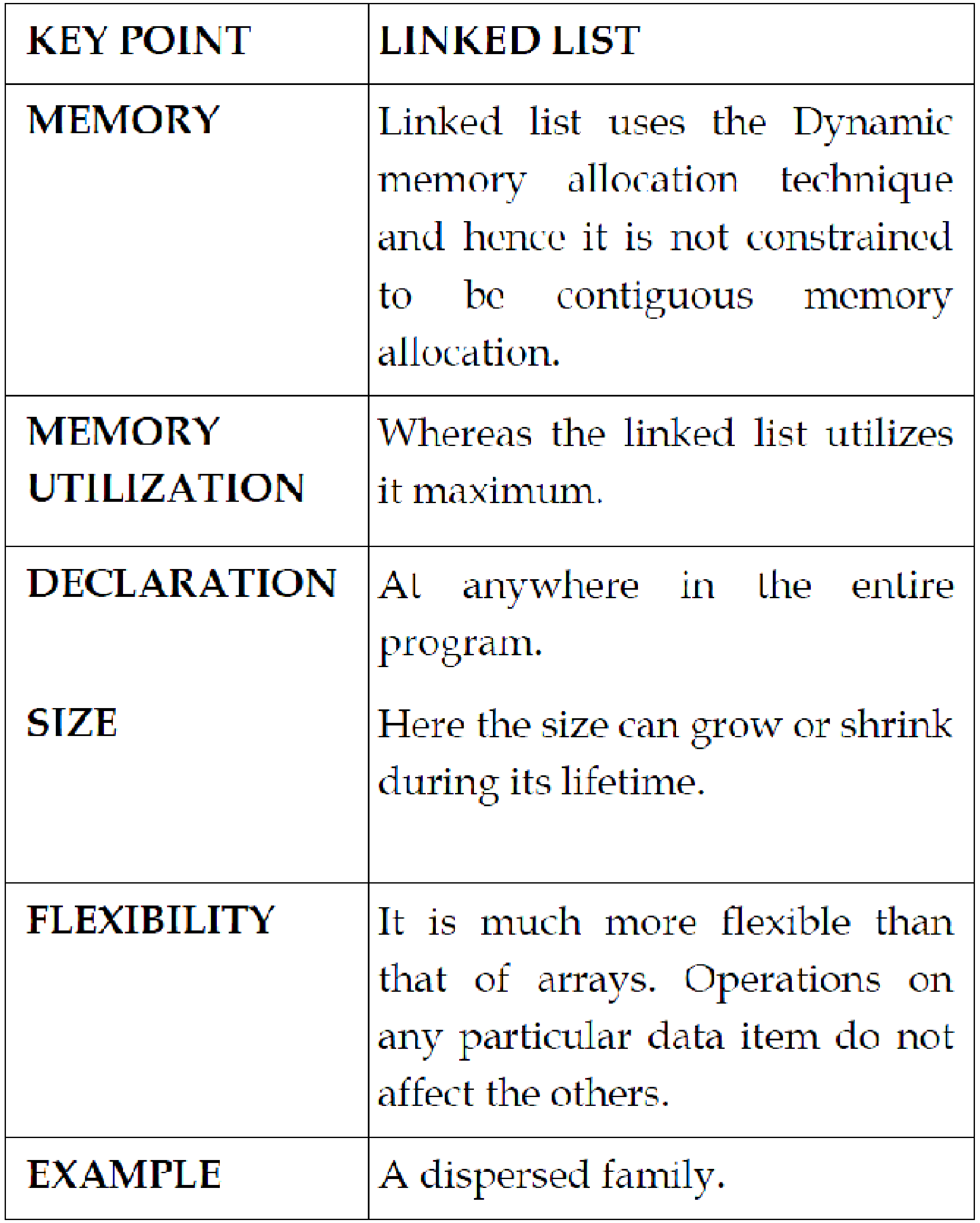
Memory Allocation Flexibility

Nodes in a linked list can be scattered throughout the memory, allowing for flexible memory allocation and efficient use of space.

Implementation Simplicity

Implementing and manipulating linked lists is relatively straightforward, making them popular in programming.

Advantages of Using a Linked List



KEY POINT	LINKED LIST
MEMORY	Linked list uses the Dynamic memory allocation technique and hence it is not constrained to be contiguous memory allocation.
MEMORY UTILIZATION	Whereas the linked list utilizes it maximum.
DECLARATION	At anywhere in the entire program.
SIZE	Here the size can grow or shrink during its lifetime.
FLEXIBILITY	It is much more flexible than that of arrays. Operations on any particular data item do not affect the others.
EXAMPLE	A dispersed family.

Comparison with Other Data Structures

Arrays

Linked lists offer dynamic size and efficient insertions/deletions, whereas arrays provide faster random access and contiguous memory.

Stacks and Queues

Stacks and queues can be implemented using linked lists, providing dynamic behavior and efficient FIFO/LIFO operations.

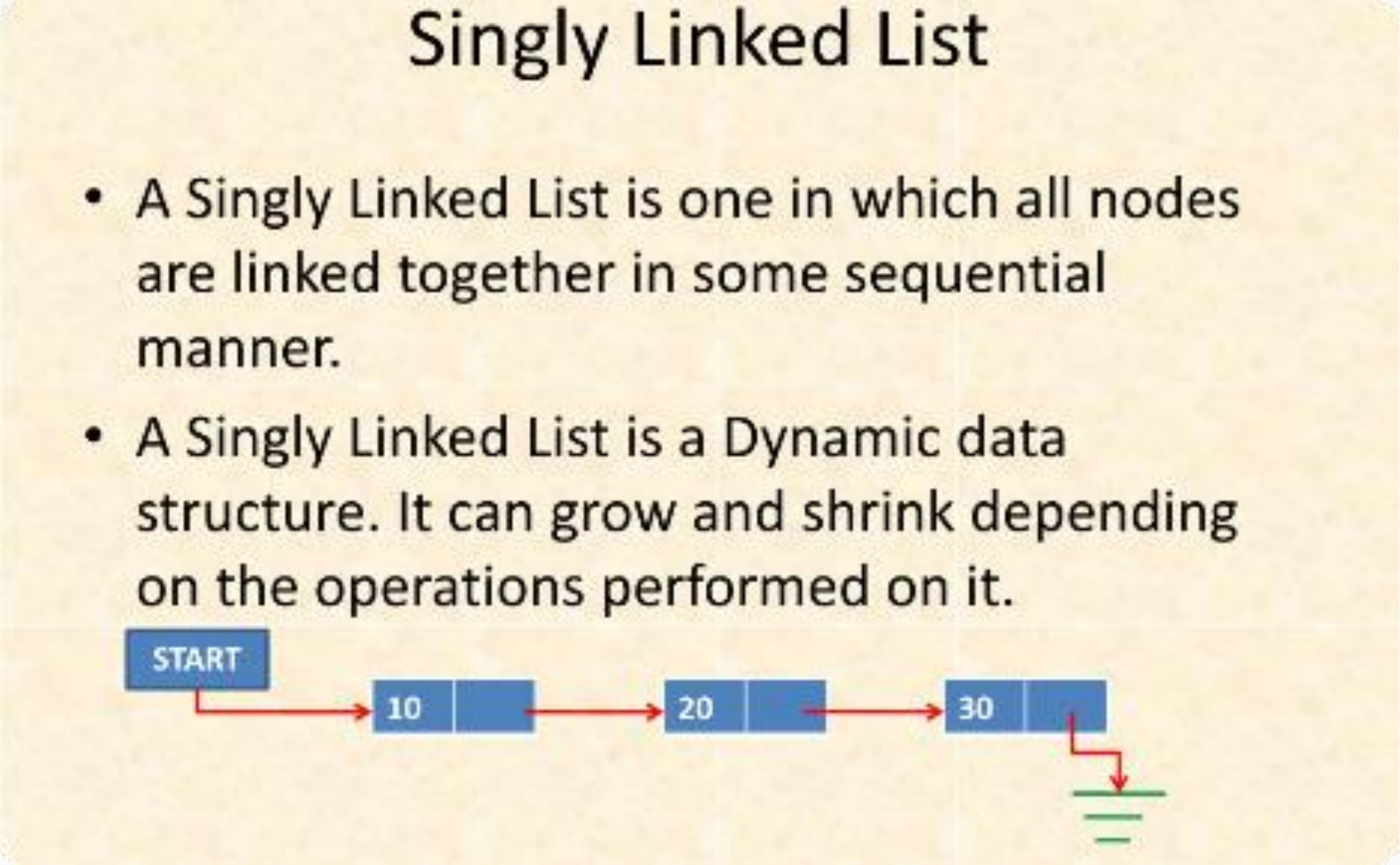
Trees and Graphs

Linked lists serve as the foundation for more complex data structures like trees and graphs, facilitating node linkage.

Types of Linked Lists

Singly Linked List

A basic linked list where each node has a reference to the next node, forming a **unidirectional chain**.



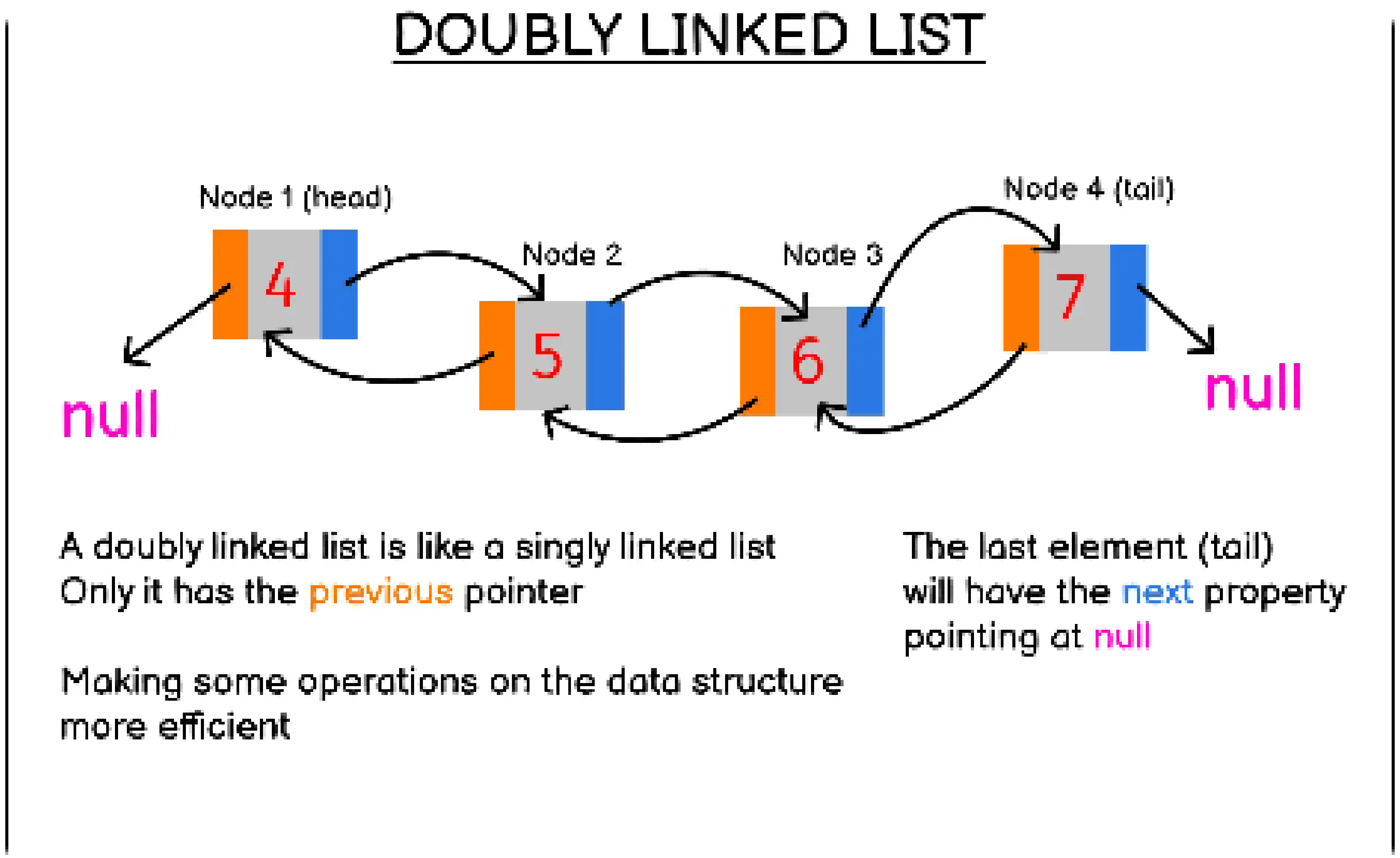
Singly Linked List

- A Singly Linked List is one in which all nodes are linked together in some sequential manner.
- A Singly Linked List is a Dynamic data structure. It can grow and shrink depending on the operations performed on it.

Types of Linked Lists

Doubly Linked List

Similar to a singly linked list, but each node also has a reference to the previous node, enabling **bidirectional traversal.**



DOUBLY LINKED LIST

A doubly linked list is like a singly linked list
Only it has the previous pointer

Making some operations on the data structure more efficient

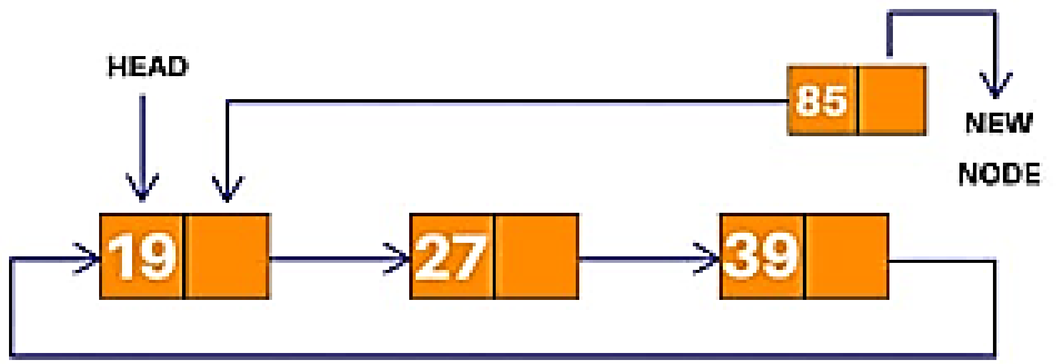
The last element (tail) will have the next property pointing at null

Types of Linked Lists

Circular Linked List

In this variation, **the last node of the list points back to the first node**, creating a circular structure.

We make the new node's next pointer point towards the head of the list



Finally, we make the last node's next pointer point towards the new node

Operations on a Linked List

1

Traversal

Iterate through the linked list to access and process each individual node or its data.

2

Insertion

Insert new nodes at the beginning, end, or any desired position in the linked list.

3

Deletion

Remove nodes from the linked list, adjusting pointers to ensure proper connections.

4

Search

Find specific data within the linked list by traversing the nodes and comparing their values.

5

Reversal

Reverse the order of the nodes in the linked list, altering the sequencing from head to tail.

Singly Linked List: Operations

```
struct Structure_Name{

Data_type info;

struct Structure_Name * link;

};
```

```
void create/append(struct node **q,int num){
    struct node *temp,*r;
    if(*q==NULL){
        temp=(struct node*)malloc(sizeof(struct node));
        temp->link=NULL;
        temp->data=num;
        *q=temp;
    }
    else{
        temp=*q;
        while(temp->link!=NULL)
                temp=temp->link;
        r =(struct node*) malloc(sizeof(struct node));
        r->data=num;
        r->link=NULL;
        temp->link=r;
    }
}
```

```
void display(struct node **q){
    struct node *temp;
    temp=*q;
    while(temp!=NULL){
        printf("%d → ",temp->data);
        temp=temp->link;
    }
}
```

```
void addatbeg(struct node **q,int num){
    struct node *temp;
    temp=(struct node*)malloc(sizeof(struct
         node));
    temp->link=*q;
    temp->data=num;
    *q=temp;
}
```

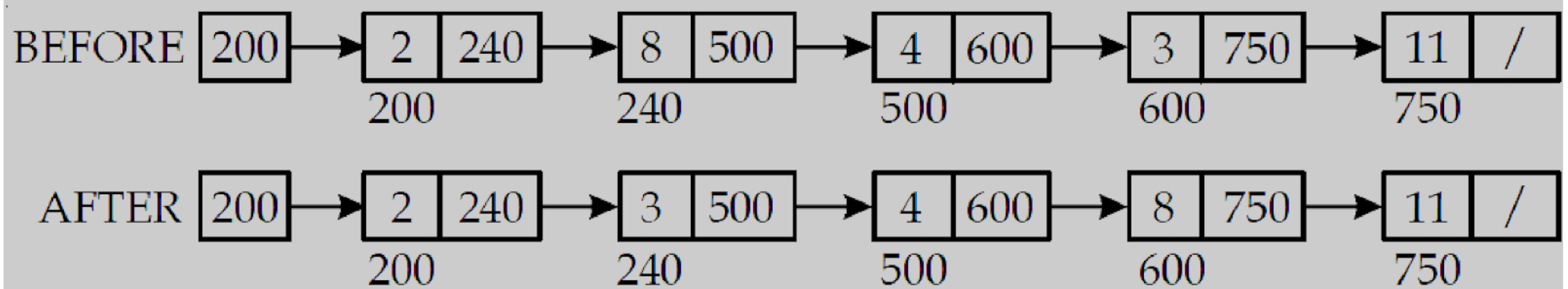
Singly Linked List: Operations

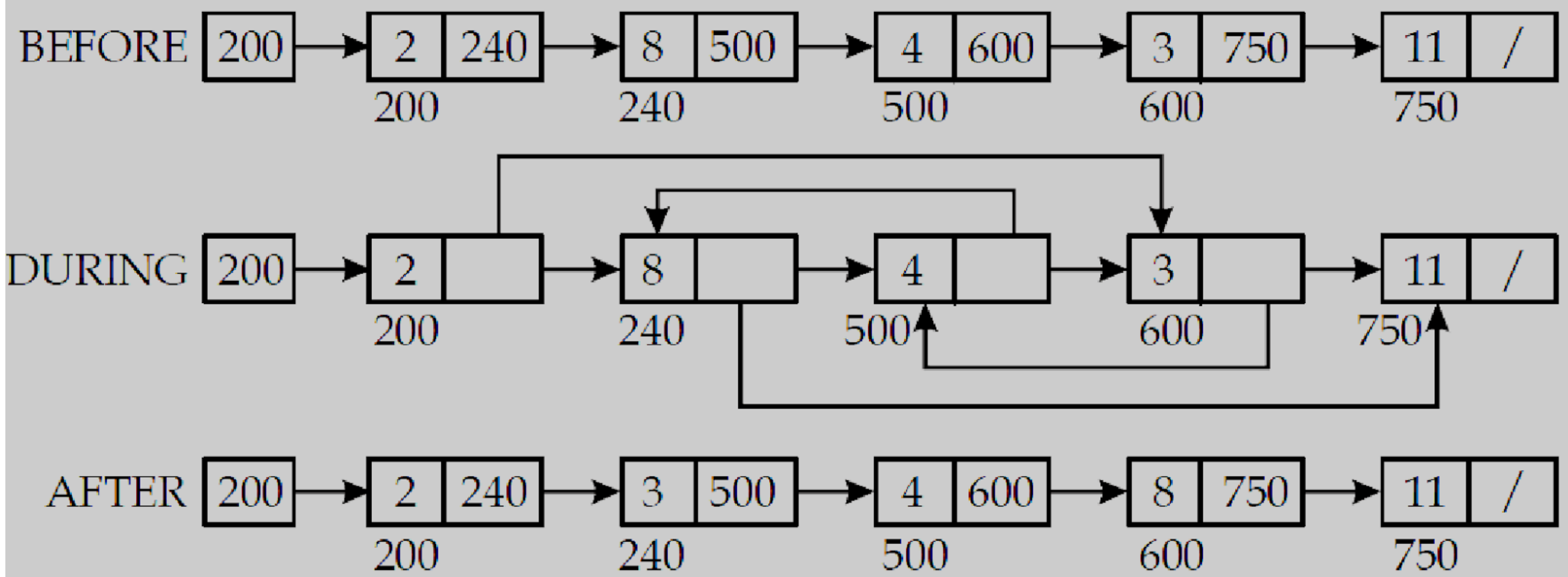
```
void addafter(struct node **q,int loc, int num){
  struct node *temp,*r;
  int i;
  temp=*q;
  for(i=0;i<loc-1;i++){
   temp=temp->link;
   if(temp==NULL){
       printf("no. of nodes are less than the position
to insert a new node");
       return;
   }
   }
   r=(struct node*)malloc(sizeof(struct node));
   r->data=num;
   r->link=temp->link;
   temp->link=r;
}
```

```
void delete(struct node **q,int num){
   struct node *temp,*pre;
   temp=*q;
   while(temp!=NULL){
      if(temp->data==num){
         if(temp==*q)
           *q=temp->link;
         else
           pre->link=temp->link;
      free(temp);
      return;
      }
     else{
       pre=temp;
       temp=temp->link;
    }
   }
   printf("element %d not found",num);
}
```

Singly Linked List: Operations

Sorting





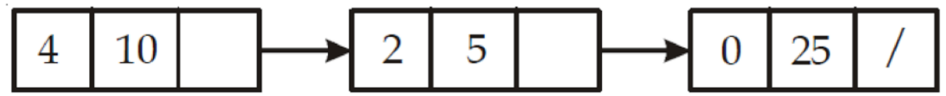
Singly Linked List: Operations

```
void merge(struct node *p, struct node *q, struct
node **s){
   struct node *z;
   z=NULL;
   if(p==NULL && q==NULL)
      return;
   while(p!=NULL && q!=NULL){
       if(*s==NULL){
          *s=malloc(sizeof(struct node));
          z=*s;
       }
       else{
          z->link=malloc(sizeof(struct node));
          z=z->link;
         }
       if(p->data<q->data){
          z->data=p->data; p=p->link;
       }
       else if(q->data<p->data){
          z->data=q->data; q=q->link;
       }
         else{
           z->data=p->data; p=p->link; q=q->link;
       }
     }
   while(p!=NULL){
      z->link=malloc(sizeof(struct node));
      z=z->link;
      z->data=p->data;
      p=p->link;
   }
   while(q!=NULL){
      z->link=malloc(sizeof(struct node));
        z=z->link;
      z->data=q->data;
      q=q->link;
   }
   z->link=NULL;
}
```

Polynomial Representation Using Linked List

```
void polyadditon (struct node *p, struct node *q, struct node
**s){
      struct node * temp;
      if (q == NULL &&  p ==NULL){
      return;
      }
      while ( p ! = NULL && q! = NULL){
        if (*s == NULL){
            *s = malloc (size of (struct node));
             temp = * s;}
         else{
           temp → link = malloc (size of (struct node));
           temp = temp → link;
        }
          if (p → exp < q → exp){
            temp → coef = q → coef;
            temp → e × p = q → e × p;
            q = q → link;}
          else {
          if (p →  exp > q → exp){
           temp → coef = p → coef;
           temp → exp = p → exp;
           p = p → link;
          }
          else {
            temp →   coef = p + coef + q → coef;
            temp → exp = q → exp;
            q = q → link ;
            p = p → link;
   }}}
```

Polynomial 10*x*^4 + 5*x*^2 + 25 is represented as following:



```
while (p ! = NULL){
    if (* s == NULL)
      * s=malloc(sizeof(struct node)); temp = *s;
    else   {
      temp → link = malloc (size of (struct
      node)); temp = temp → link;
    }
    temp → coef = p → coef;
    temp → exp = p → exp;
    p = p → link; }
    while (q ! = NULL){
    if (*s == NULL){
          *s = malloc (sizeof(struct node));
          temp = *s ;
        }
        else {
          temp →link=malloc(sizeof(struct
        node));
        }
}
}
```

Common Problems and Challenges with Linked Lists

1

Memory Overhead

Each node in a linked list requires additional memory for the data payload and the next/previous node pointers.

2

Traversal Complexity

Locating a specific node in a linked list requires iterating through the nodes from the beginning until the desired node is found.

3

Unsuitable for Random Access

Unlike arrays, linked lists do not support direct access to elements based on their indices, slowing down element retrieval.

Applications of linked list

1. **Dynamic Memory Allocation:**

2. **Implementation of Stacks, Queues, Trees, and Graphs:**

3. **Symbol Table in Compilers:**
   - Symbol tables store information about variables, functions, and other symbols in a program, and linked lists facilitate efficient management and retrieval of this information.

4. **Dynamic Memory Management in Operating Systems:**
   - In operating systems, linked lists are commonly used to manage free blocks of memory.
   - Memory allocation and deallocation can be efficiently handled by maintaining linked lists of free memory blocks.

5. **Music Player Playlist:**
   - Linked lists are suitable for representing playlists in music players.
   - Each node in the linked list represents a song, and the links between nodes define the order of the playlist.

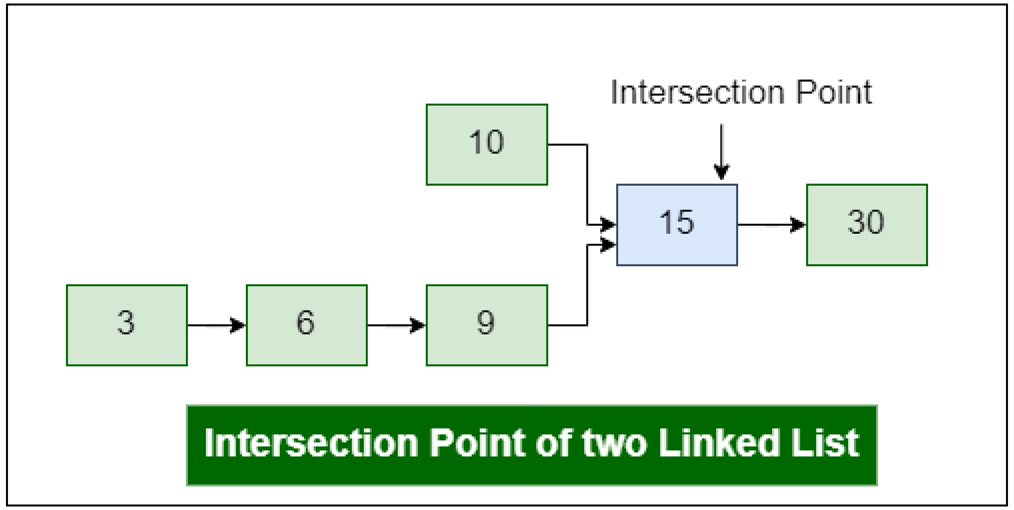
6. **Hash Table Chaining:**
   - Linked lists are used in combination with hash tables for collision resolution through chaining.
   - In case of a hash collision, elements with the same hash value can be stored in a linked list attached to the corresponding hash table index.

7. **Polynomial Representation:**
   - Linked lists can be used to represent polynomials efficiently.
   - Each node represents a term in the polynomial, with the links indicating the degree of the term.

8. **Job Scheduling in Operating Systems:**
   - Linked lists are employed in job scheduling algorithms in operating systems.
   - Each node represents a job, and the links define the scheduling order.

Example: Write a function to get the intersection point of two Linked Lists

There are two singly linked lists in a system. By some programming error, the end node of one of the linked lists got linked to the second list, forming an inverted Y-shaped list. Write a program to get the point where two linked lists merge.



Intersection Point of two Linked List

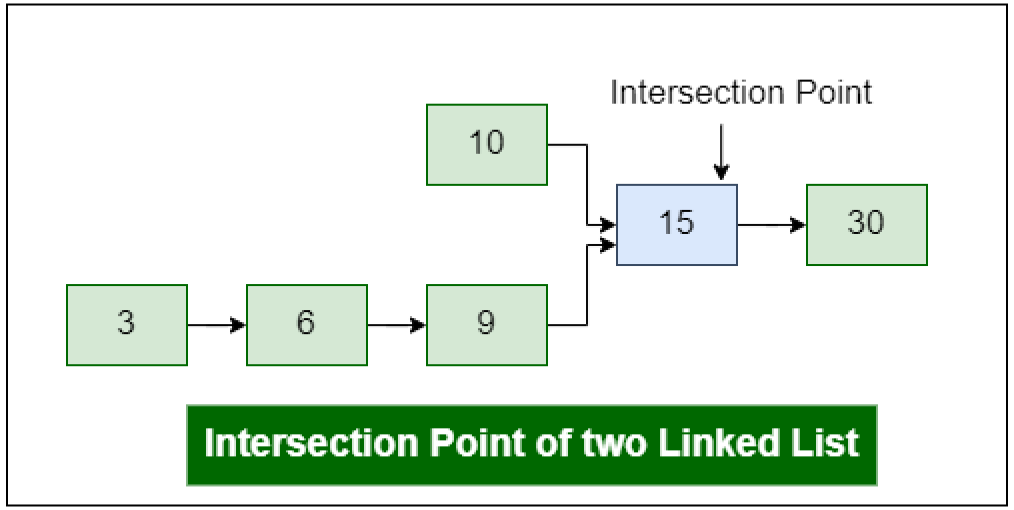
Approach-1

- Use 2 nested for loops.
- The outer loop will be for each node of the 1st list and the inner loop will be for the 2nd list.
- In the inner loop, check if any of the nodes of the 2nd list is the same as the current node of the first linked list.

The **time complexity of this method will be O(M * N)** where M and N are the numbers of nodes in two lists.
Space Complexity = O(1)

Example: Write a function to get the intersection point of two Linked Lists

There are two singly linked lists in a system. By some programming error, the end node of one of the linked lists got linked to the second list, forming an inverted Y-shaped list. Write a program to get the point where two linked lists merge.



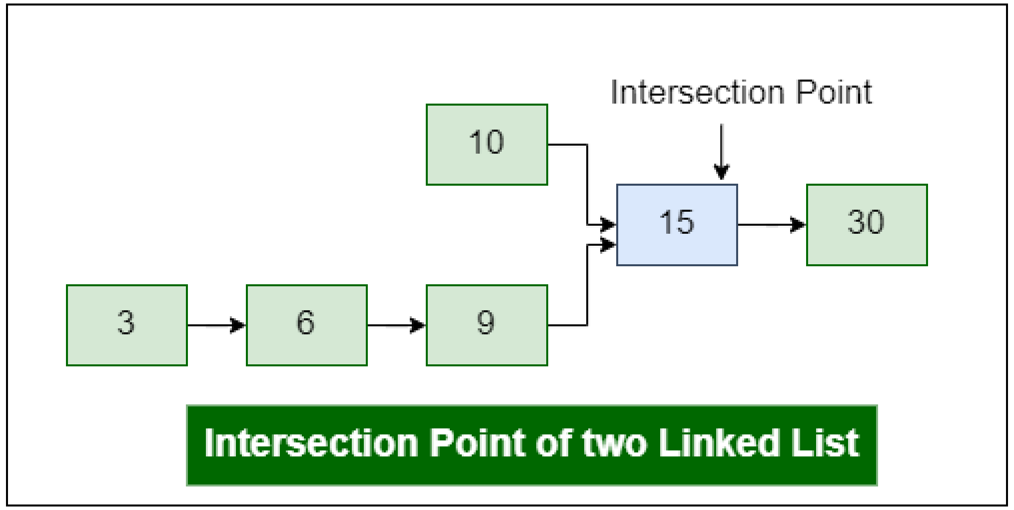
Approach-2

- Create an empty hash set.
- Traverse the first linked list and insert all nodes' addresses in the hash set.
- Traverse the second list. For every node check if it is present in the hash set. If we find a node in the hash set, return the node.

The **time complexity of this method will be O(N)** where M and N are the numbers of nodes in two lists.
Space Complexity = O(M), Let N > M

Example: Write a function to get the intersection point of two Linked Lists

There are two singly linked lists in a system. By some programming error, the end node of one of the linked lists got linked to the second list, forming an inverted Y-shaped list. Write a program to get the point where two linked lists merge.



Intersection Point of two Linked List

Approach-3

- Get the count of the nodes in the both lists, let the count be c1, c2.
- Get the difference of counts **d = abs(c1 – c2)**
- Now traverse the bigger list from **the first node to d** nodes so that from here onwards both the lists have an equal no of nodes
- Then traverse both lists in parallel till a common node encountered.

Time Complexity: O(M+N)
Auxiliary Space: O(1)

Thank you!