

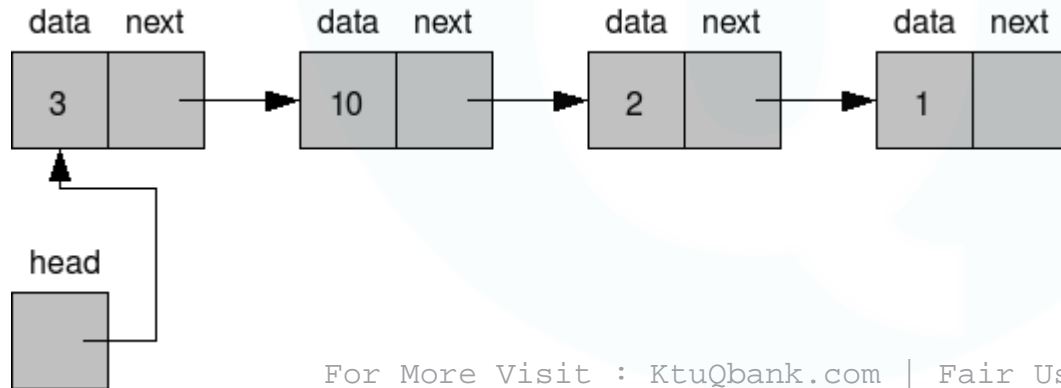
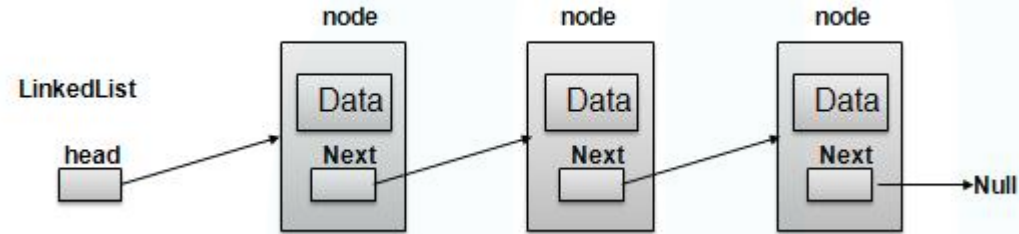
Linked Lists

Linked List :: Basic Concepts

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

Linked List :: Basic Concepts

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



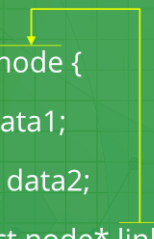
Self Referential Structures

- Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.
- Structures pointing to the same type of structures are self referential in nature.

Self Referential Structures

Self Referential Structures

```
struct node {  
    int data1;  
    char data2;  
    struct node* link;  
};
```



- In the above example 'link' is a pointer to a structure of type 'node'.
- Hence, the structure 'node' is a self-referential structure with 'link' as the referencing pointer.
- An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

Self Referential Structures

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

Self Referential Structures

Such a structure can be represented as:

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

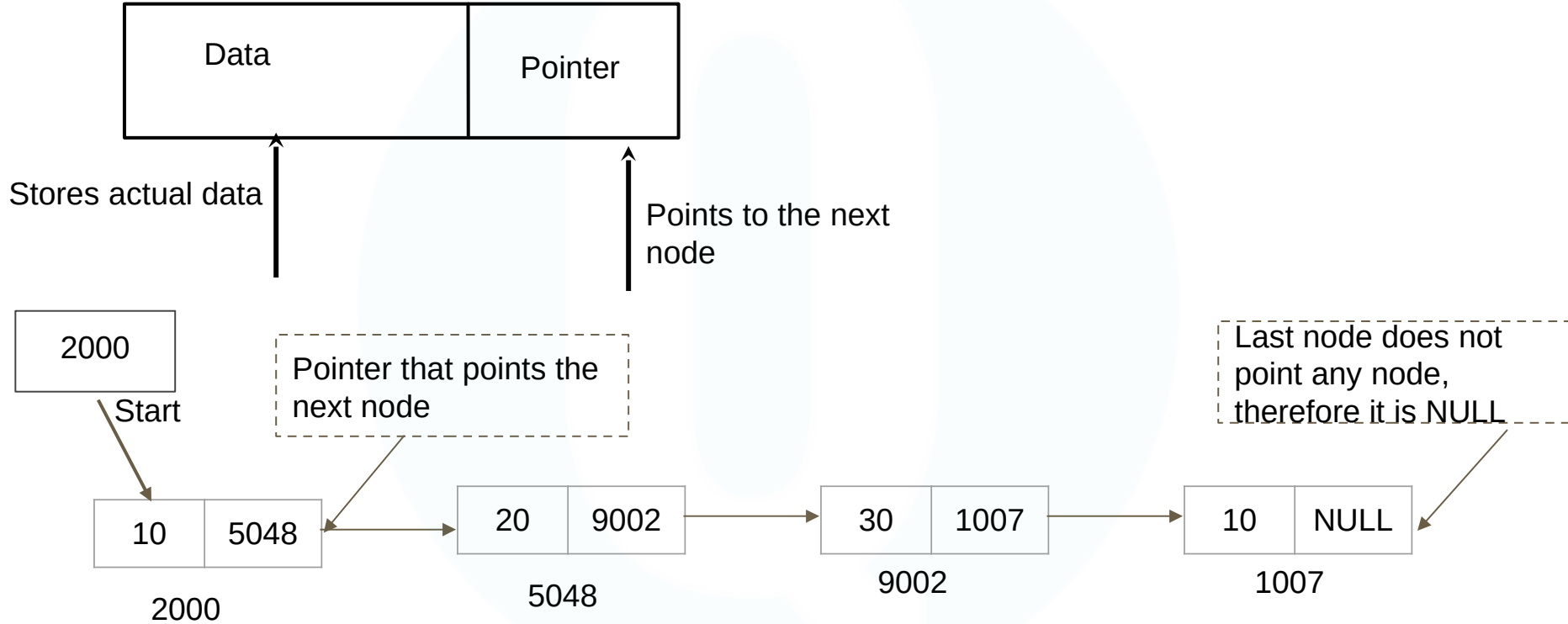
- Such structures that contain a member field pointing to the same structure type are called self-referential structures.



Definition

- It is a list or collection of data items that can be stored in scattered locations (positions) in memory by establishing link between the items.
- To store data in scattered locations in memory we have to make link between one data item to another.
- So, each data item or element must have two parts: one is **data part** and another is **link (pointer) part**.

Schematic Representation



Definition [cont..]

- Each data item of a linked list is called a node. **Data part** contains (holds) **actual data** (information) and the **link part points to the next node** of the list.
- To locate the list an external pointer is used that points the first node of the list.
- The link part of the last node will not point any node. That means it will be null.

Create a new node

1. Node Declaration

A node can be declared/define using code as follows:

```
struct node
```

```
{
```

```
int data; ← Variable for data part
```

```
struct node *next; ← Variable for pointer part  
                    (A pointer that will point  
                    next node)
```

```
}
```

Create a new node [cont..]

2. Declare variable (pointer type) that point to the node:

```
struct node *start;
```

3. Allocate memory for new node:

```
start = (struct node*)malloc(sizeof(struct node));
```

4. Enter value:

```
start → data = item;           //item is a variable
```

```
start → next = NULL;
```

Operations on Linked Lists

- Insertion
- Deletion
- Searching

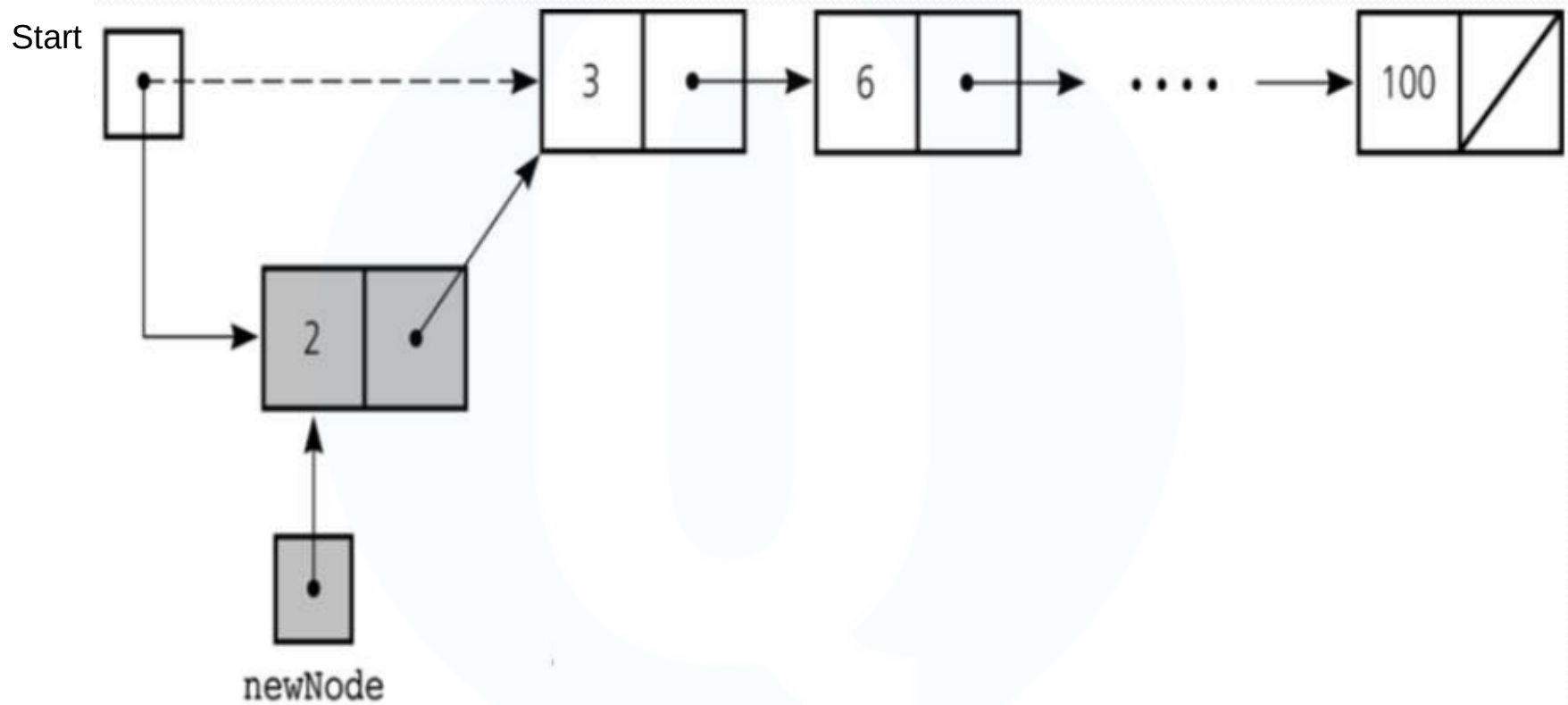
Insertion

- Insertion at beginning.
- Insertion at end.
- Insertion after a specific node.

Insertion at the beginning

1. Make the next pointer of the node point towards the first node of the list
2. Make the start pointer point towards this new node

If the list is empty simply make the start pointer point towards the new node

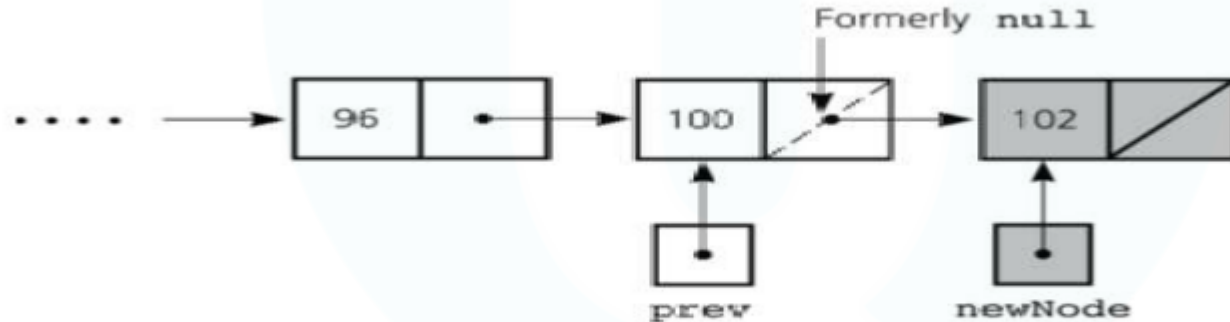


C Code for Insertion at beginning

```
void insert_beg(node* start, int item)
{
    int item;
    new_node=(struct node*)malloc(sizeof(struct node));
    scanf("%d",&item);
    new_node->data=item;
    new_node->next=NULL;
    if(start==NULL)
        start=new_node;
    else{
        new_node->next=start;
        start=new_node;
    }
}
```

Insertion at the End

1. Make the next pointer of the last node point to the new node.
2. Make the next pointer of the new node to NULL



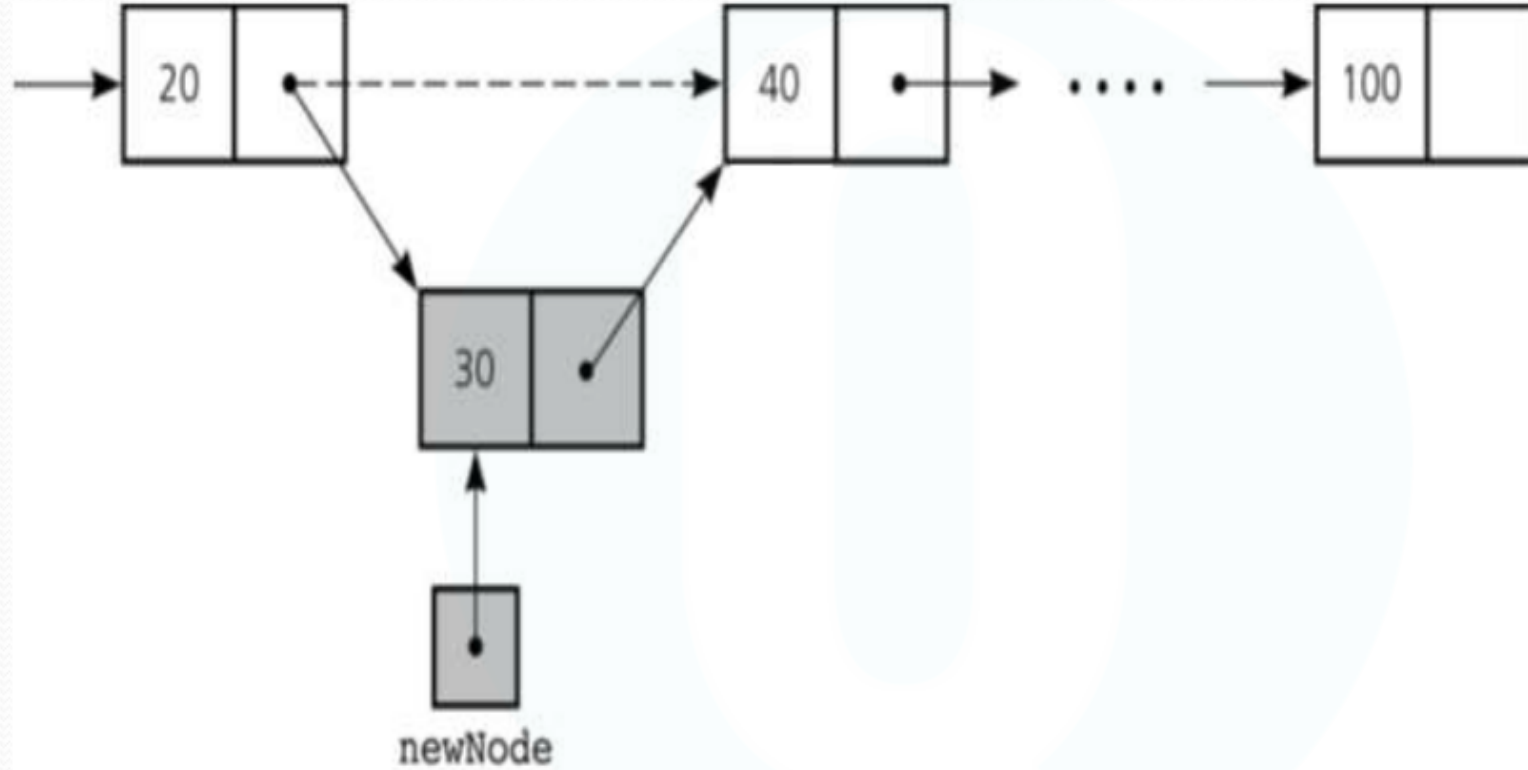
C code

```
Struct node *temp;  
while(temp->next!=NULL)  
{  
    temp=temp->next;  
}  
temp->next=new_node;  
new_node->next=NULL;
```

```
void insert_end(node* new_node)
{
    node *temp=start;
    if(start==NULL)
    {
        start=new_node;
        printf("\nNode inserted successfully at the end...!!!\n");
    }
    else{
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=new_node;
        new_node->next=NULL;
    }
}
```

Insertion after a Specific Node

- Make the next pointer of the node to be inserted point to the next node of the node after which you want to insert the node
- Make the next pointer of the node after which the node is to be inserted, point to the node to be inserted



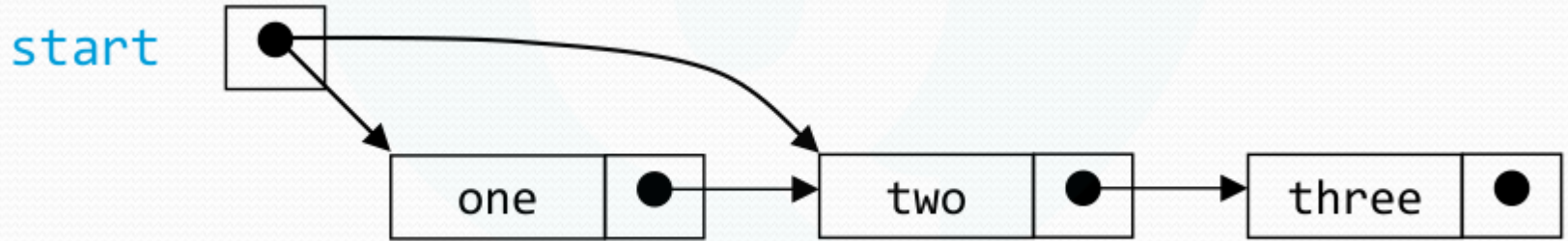
```
void insert_mid(node* new_node)
{
    node *temp=start;
    if(start==NULL)
    {
        start=new_node;
        printf("\nNode inserted successfully at the end...!!!\n");
    }
    else{
        while(temp!=Insert_position)
        {
            temp=temp->next;
        }
        temp=prev->next;
        prev->next=new_node;
        new_node->next=temp;
    }
}
```


Deletion

- Deleting the first node
- Deleting the last node
- Deleting the intermediate node

Deleting First Node

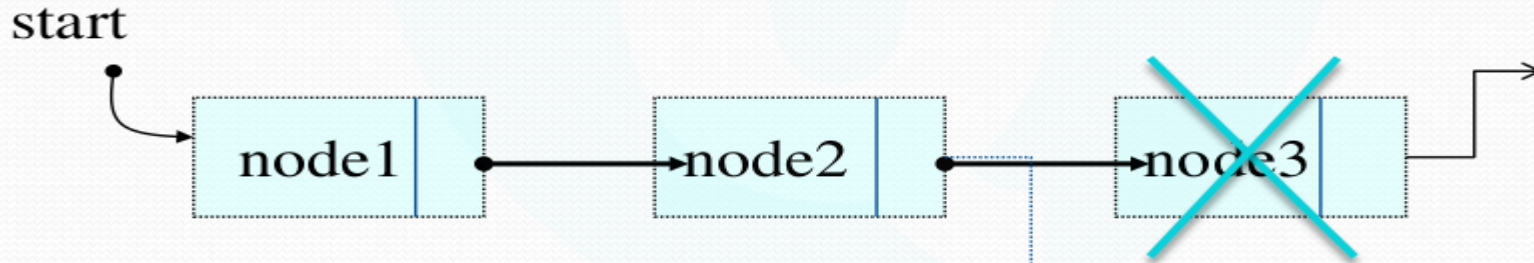
- Making the start pointer point towards the 2nd node
- Deleting the first node using delete keyword



```
void del_first()
{
    if(start==NULL)
        printf("\nError.....List is empty\n");
    else
    {
        node* temp=start;
        start=temp->next;
        delete(temp);
        printf("\nFirst node deleted successfully....!!!");
    }
}
```

Deleting the last node

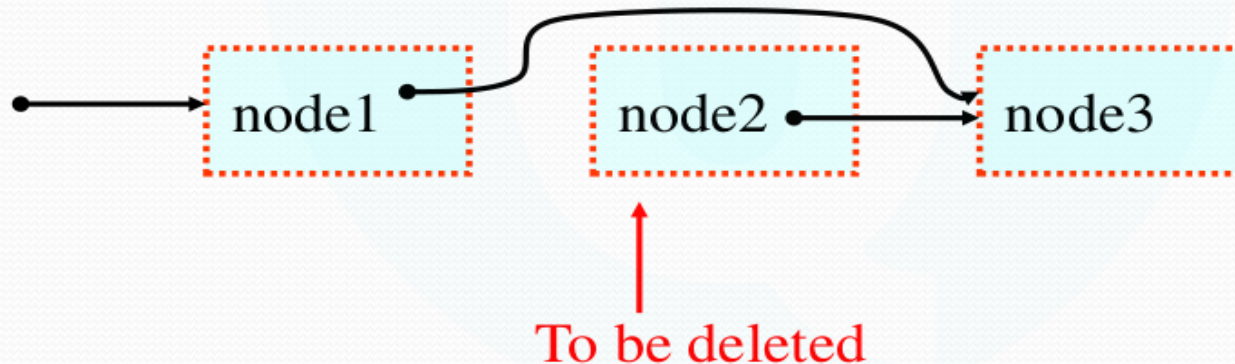
- Making the second last node's next pointer point to NULL
- Deleting the last node via delete keyword



```
void del_last()
{
    if(start==NULL)
        printf("\nError....List is empty");
    else
    {
        node* temp=start, *prev;
        while(temp->next!=NULL)
        {
            prev=temp;
            temp=temp->next;
        }
        temp->next=NULL;
        delete(temp);
    }
}
```

Deleting an Internal Node

- Make the next pointer of the node previous to the node being deleted, point to the successor node of the node to be deleted
- Delete the node using delete keyword



C Code

```
void del(int c)
{
    node *p,*q;
    p =start;
    q=p->next;
    if(p==NULL)
        printf("\nNode not found\n");
    else{
        while(q->data != c)
        {
            q = q->next;
            p = p->next;
        }
        p->next = q->next;
        delete(q);
    }
}
```

Searching

- Searching involves finding the required element in the list
- Linear search traversal
- In linear search, each node is traversed till the data in the node matches with the required value

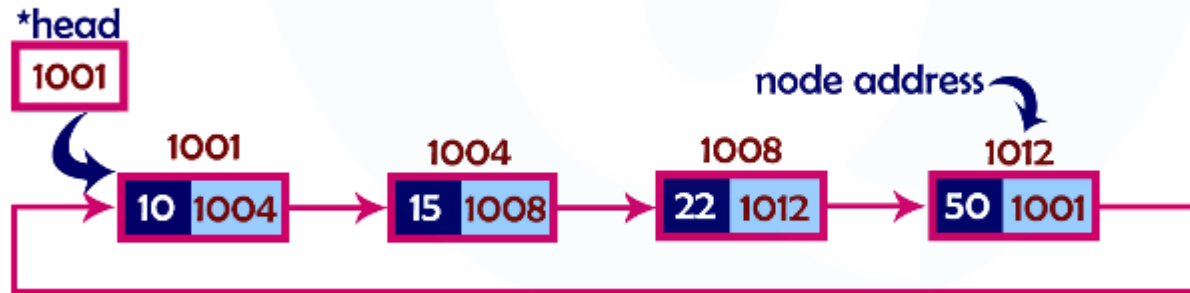

```
void search(int x)
{
    node*temp=start;
    while(temp!=NULL)
    {
        if(temp->data==x)
        {
            printf("FOUND %d",temp->data);
            break;
        }
        temp=temp->next;
    }
}
```

Types of Linked Lists

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

Circular Linked Lists

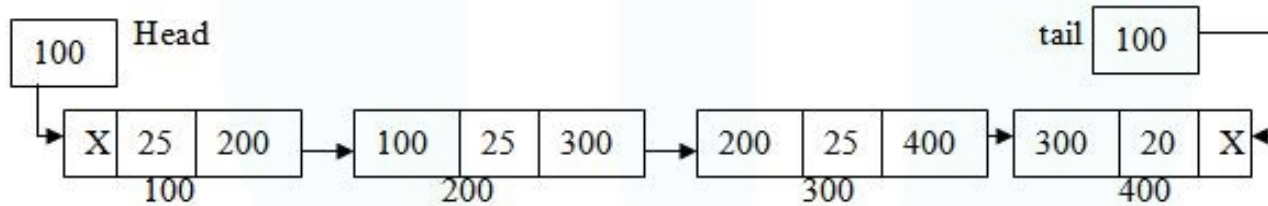
- A circular linked list is a list where each node has two parts; one is data part to hold the data and another is pointer part that points the next node
- The last node's pointer points the first node of the list.
- And there is an external pointer to the list that points the first node.



Doubly Linked List

- A doubly or two way linked list is a list where each node has three parts:
 - One link or pointer to the previous (backward) node
 - One data part to hold the data
 - Another link or pointer to the following (forward) node.
- There is an external pointer to the first node of the list.
- It is also called two-way linked list.

Doubly Linked List (Graphical representation)



A node of doubly linked list

Node declaration:

```
struct node
```

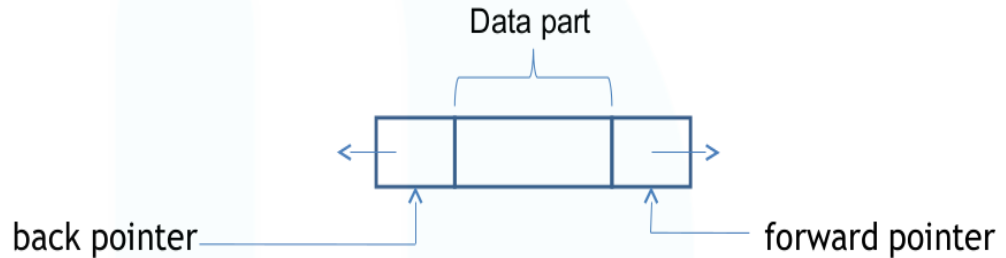
```
{
```

```
node *back; //back pointer
```

```
int data;
```

```
node *next; //forward pointer
```

```
}
```



DLL's compared to SLL's

Advantages:

- Can be traversed in either direction (may be essential for some programs)
- Some operations, such as deletion and inserting before a node, become easier

Disadvantages:

- Requires more space
- List manipulations are slower (because more links must be changed)
- Greater chance of having bugs (because more links must be manipulated)

Inserting at beginning

```
void insert_beg(node *p)
{
    if(start==NULL)
    {
        start=p;
        printf("\nNode inserted successfully at the beginning\n");
    }
    else
    {
        node* temp=start;
        start=p;
        temp->previous=p; //making 1 st node's previous point to the new
node
p->next=temp; //making next of the new node point to the 1st
```

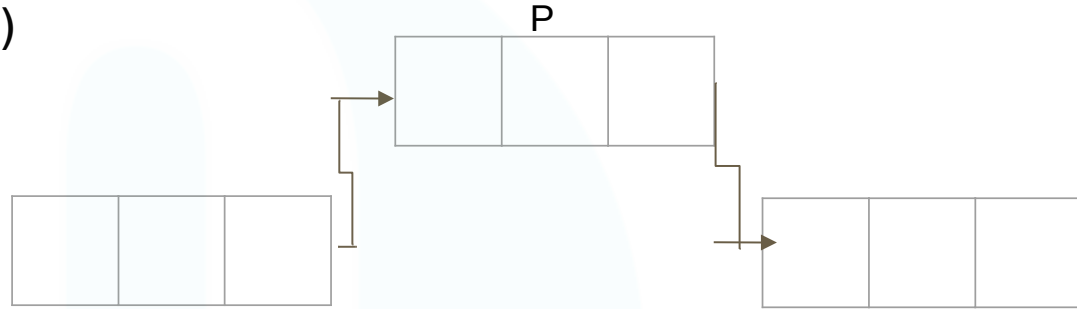

Inserting at the End

```
void insert_end(node* p)
{
    if(start==NULL){
        start=p;
        printf("\n Node inserted successfully at the end");}
    else{
        node* temp=start;
        while(temp->next!=NULL){
            temp=temp->next; }
        temp->next=p;
        p->previous=temp;
        p->next=NULL;
        printf("\n Node inserted successfully at the end\n");
    }
}
```

Inserting after a node

```
void insert_after(int c,node* p)
```

```
{  
    temp=start;  
    for(int i=1;i<c-1;i++)  
    {  
        temp=temp->next;  
    }  
    p->next=temp->next;  
    temp->next->previous=p;  
    temp->next=p;  
    p->previous=temp;  
    printf("\n Inserted successfully");  
}
```



Deleting a node

```
void del_at(int c)
```

```
{
```

```
node*s=start;
```

```
{
```

```
for(int i=1;i<c-1;i++)
```

```
{
```

```
    s=s->next;
```

```
}
```

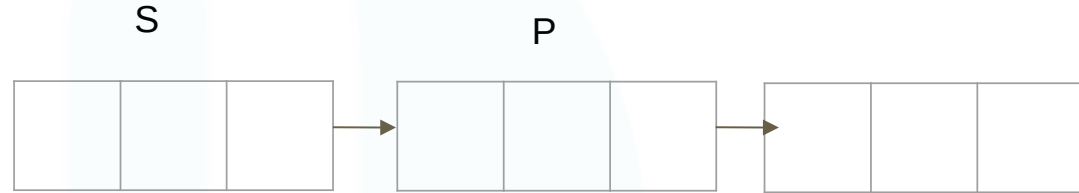
```
node* p=s->next;
```

```
s->next=p->next;
```

```
p->next->previous=s;
```

```
delete(p);
```

```
printf("\ndeleted successfully");
```



Arrays Vs Linked Lists

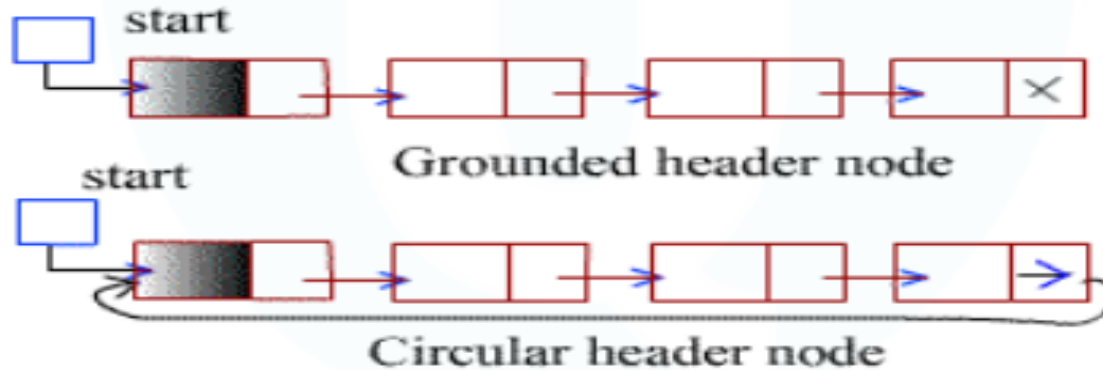
Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

Header Nodes

- A header linked list is a linked list which always contains a special node called the *header node* at the beginning of the list.
- It does not represent an item in the list.
- There are two types of header list:
 - *Grounded header list* : is a header list where the last node contain the null pointer.
 - *Circular header list* : is a header list where the last node points back to the header node.

Header Nodes[cont..]

- The information portion of such a node could be used to keep global information about the entire list such as:
 - Number of nodes (not including the header) in the list
 - Pointer to the last node in the list
 - Pointer to the current node in the list



Applications of Linked Lists

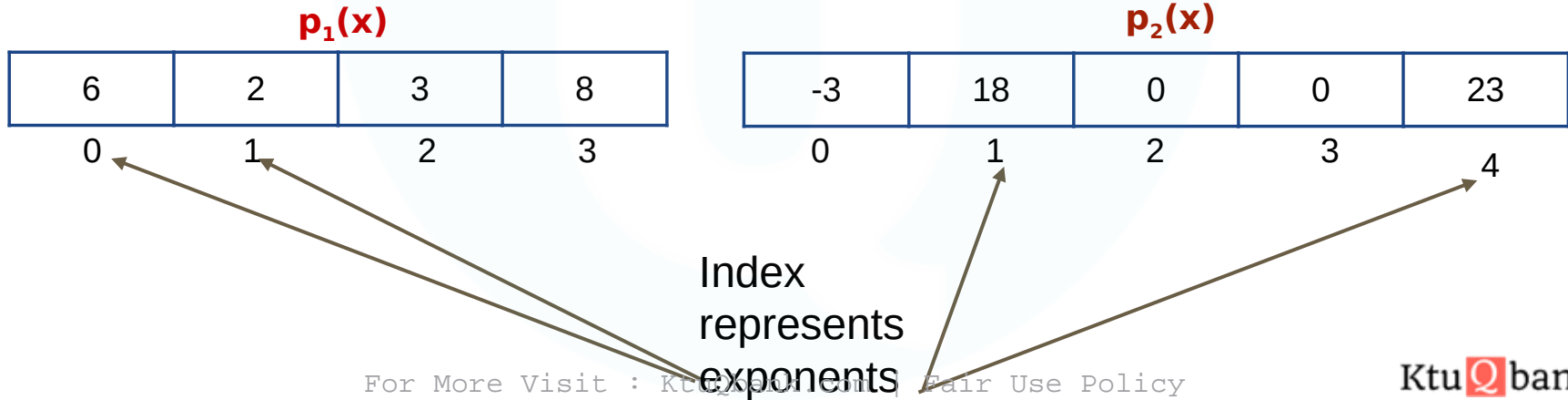
- Linked Lists can be used to implement Stacks , Queues.
- Linked Lists can also be used to implement Graphs. (Adjacency list representation of Graph).
- Polynomial Representation
- Memory Management

Polynomial Representation

● Array Implementation:

$$p_1(x) = 8x^3 + 3x^2 + 2x + 6$$

$$p_2(x) = 23x^4 + 18x - 3$$



Disadvantages

● $p_3(x) = 16x^{21} - 3x^5 + 2x + 6$

6	2	0	0	0	-3	0	16
0	1				5			21

Waste of Space..!

Disadvantages

- Only good for non-sparse polynomials.
- Have to allocate array size ahead of time.
- Huge array size required for sparse polynomials.
Waste of space and runtime.

Polynomial Representation - Linked Lists

- Data Part has 2 parts:

- Coefficient
- Exponent

Coefficient	Exponent	Pointer
-------------	----------	---------

- $3x^2$

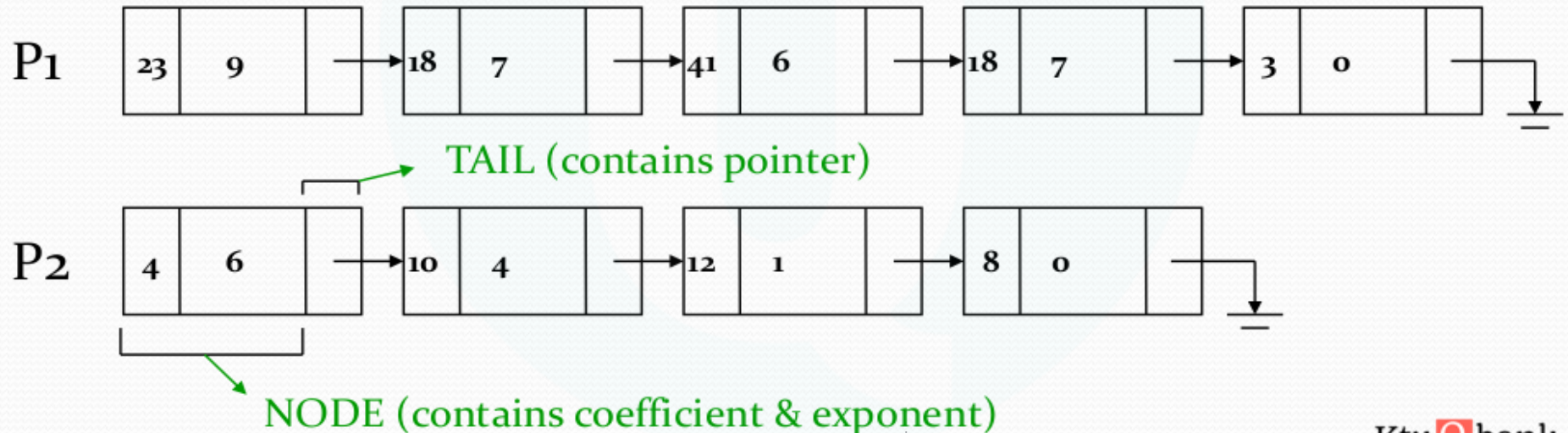
3	2	Address of next node
---	---	----------------------

Polynomial Representation

● Linked list Implementation:

$$p_1(x) = 23x^9 + 18x^7 + 41x^6 + 163x^4 + 3$$

$$p_2(x) = 4x^6 + 10x^4 + 12x + 8$$



Advantages

- Saves space (don't have to worry about sparse polynomials) and easy to maintain
- Don't need to allocate list size and can declare nodes (terms) only as needed

Disadvantages

- Can't go backwards through the list
- Can't jump to the beginning of the list from the end.

Polynomials

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \dots + a_1 x^1 + a_0 x^0$$

Representation:

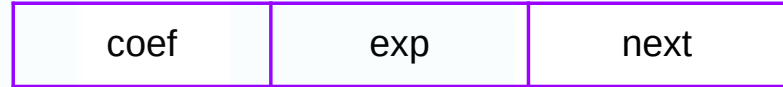
```
struct poly_node {
```

```
int coef;
```

```
int exp;
```

```
struct polynode * next;
```

```
};
```



Addition of Polynomials

Algorithm:

1. Read the number of terms in the first polynomial P
2. Read the coefficient and exponent of the first polynomial
3. Read the number of terms in the second polynomial Q
4. Read the coefficient and exponent of the second polynomial
5. Create a list for storing result as R
5. Set the temporary pointers p and q to traverse the two polynomials respectively

Algorithm[cont..]

6. Compare the exponents of two polynomials starting from the first nodes

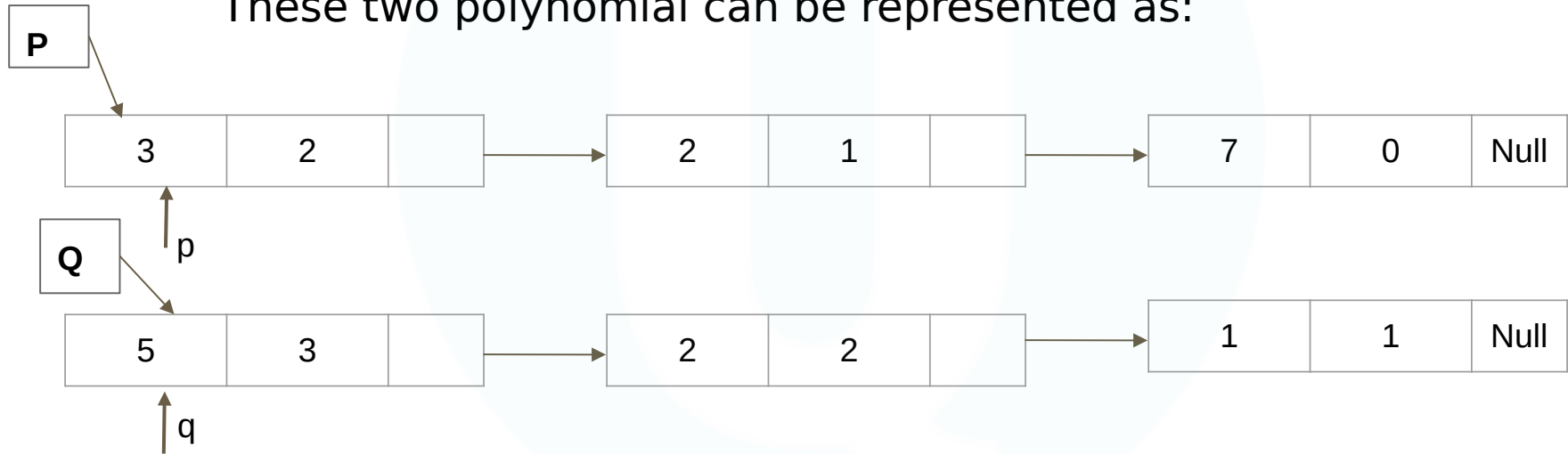
- a) If both exponents are equal then add the coefficient and store it in the resultant linked list. Move the pointer p and q to next.
- b) If the exponent of the current term in the first polynomial P is less than the exponent of the current term of the second polynomial then added the second term to the resultant linked list. And, move the pointer q to point to the next node in the second polynomial Q.
- c) If the exponent of the current term in the first polynomial P is greater than the exponent of the current term in the second polynomial Q, then the current term of the first polynomial is added to the resultant linked list. And move the pointer p to the next node.

Example:

$$P = 3x^2 + 2x + 7$$

$$Q = 5x^3 + 2x^2 + x$$

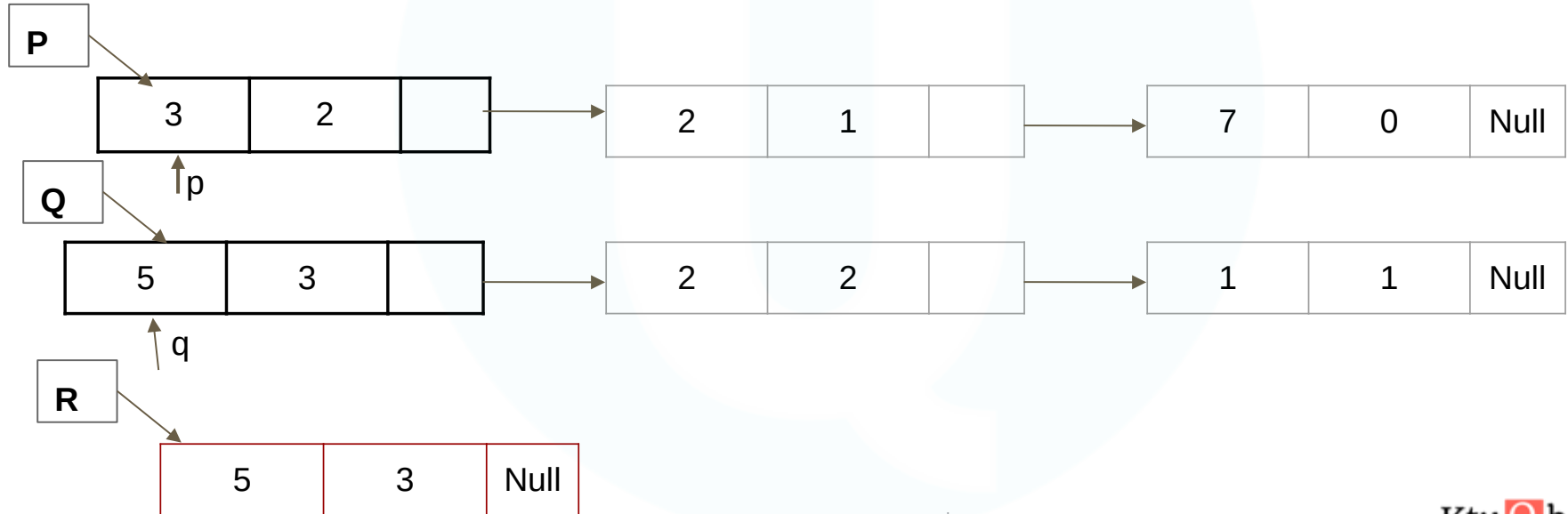
These two polynomial can be represented as:



Step 1. Compare the exponent of p and the corresponding exponent of q. Here,

$$\text{exp}(p) < \text{exp}(q)$$

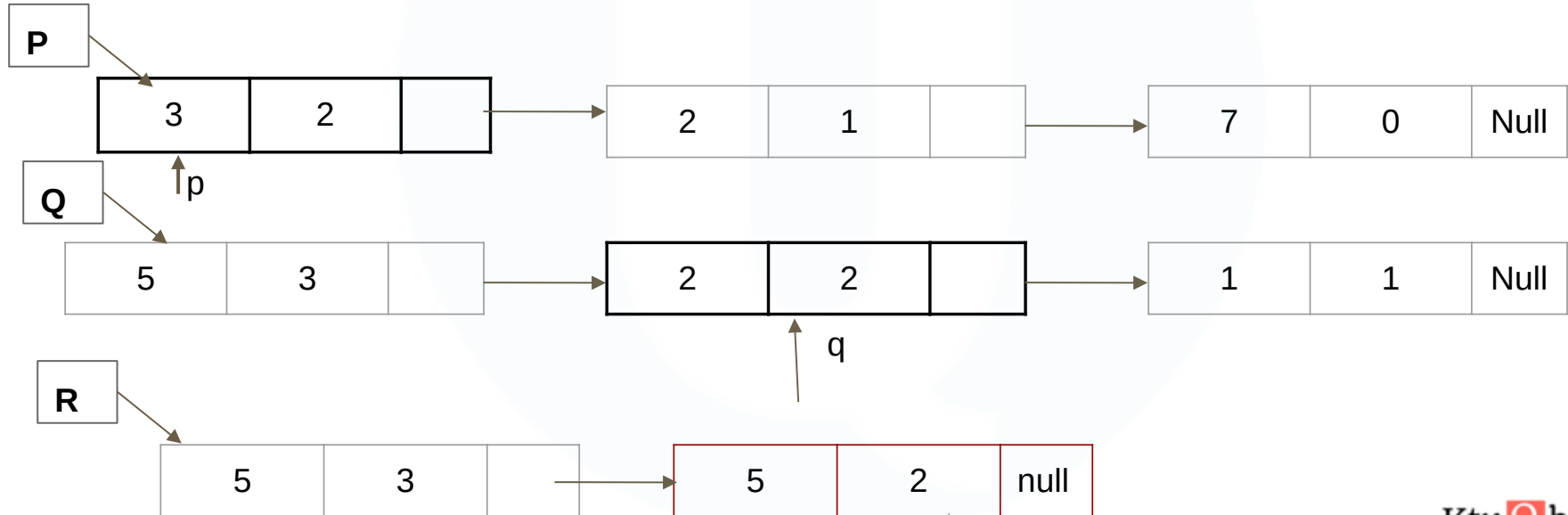
So, add the terms pointed to by q to the resultant list. And now advance the q pointer.



KtuQbank
Step 2: Compare the exponent of the current terms. Here,

$$\text{expo}(p) = \text{expo}(q)$$

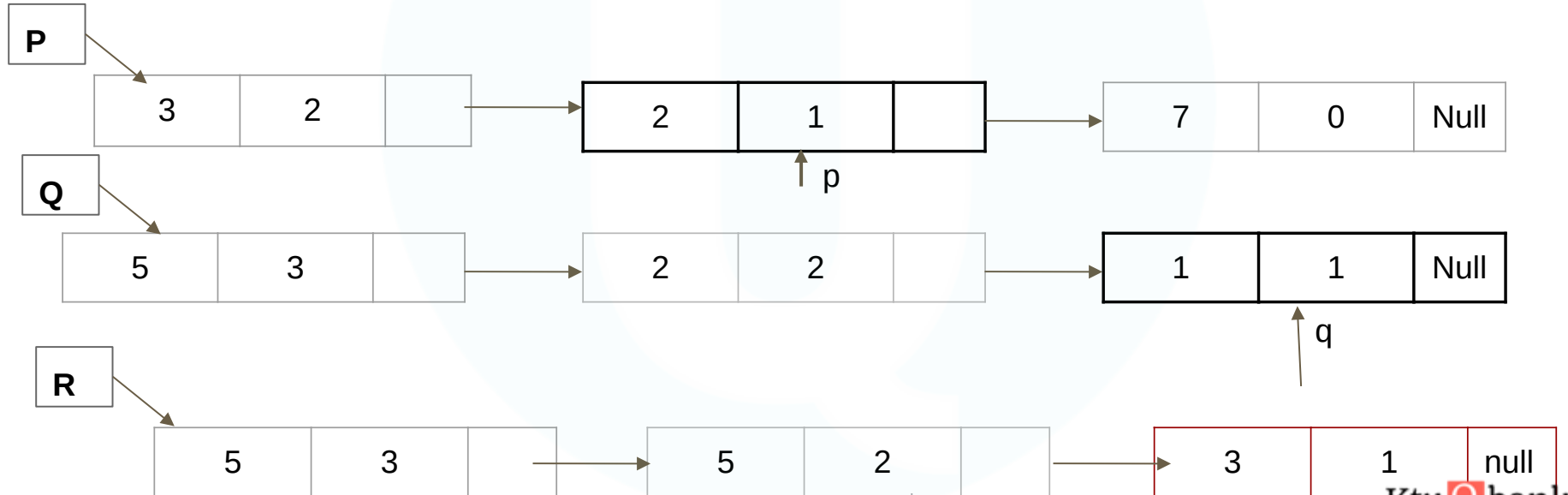
So, add the coefficients of these two terms and link this to the resultant list. And, advance the pointers p and q to their next nodes.



KtuQbank Step 3: Compare the exponents of the current terms again

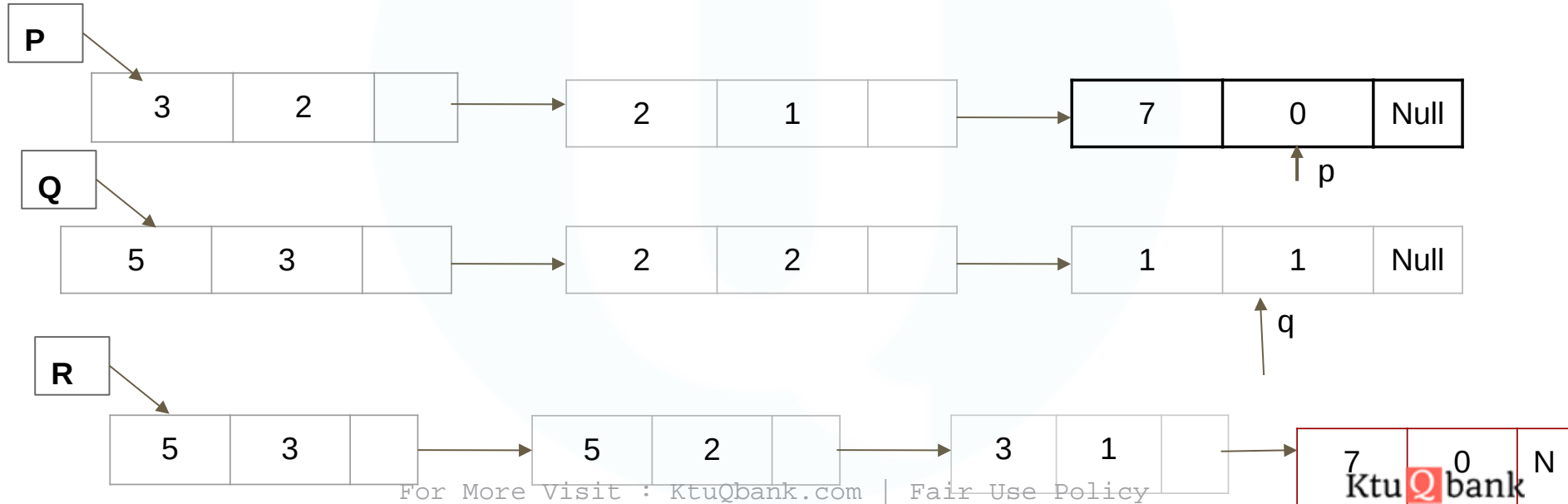
$$\text{expo}(p) = \text{expo}(q)$$

So, add the coefficients of these two terms and link this to the resultant linked list. Q reaches the NULL and p points the last node



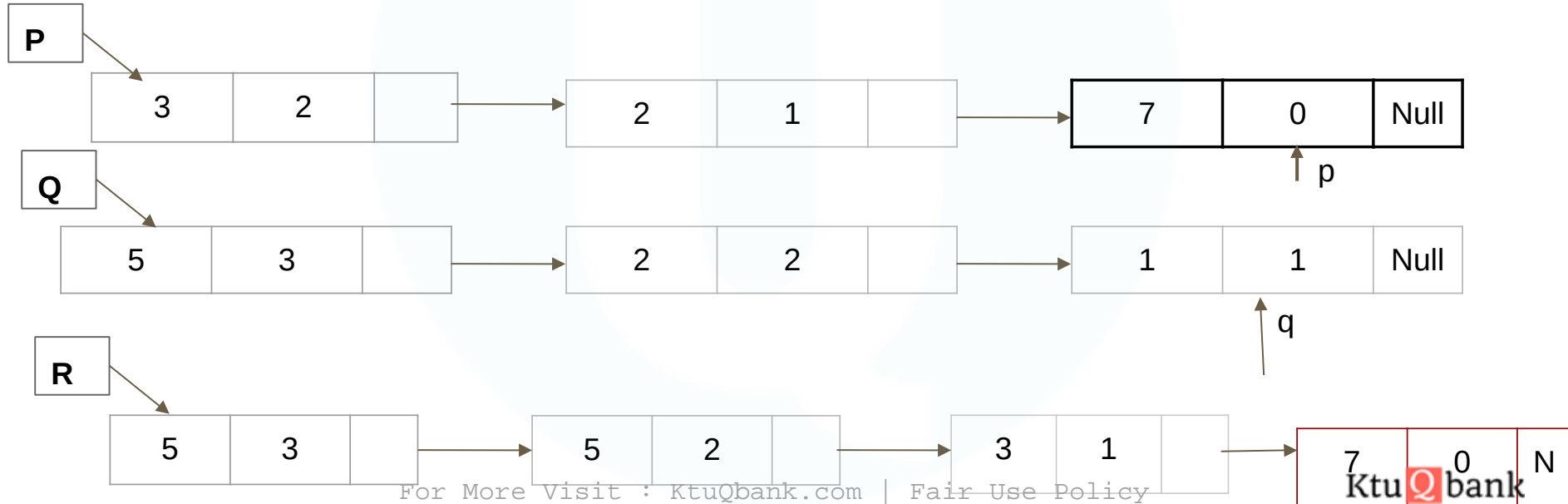
KtuQbank
Step 3: There is no node in the second polynomial to compare with. So, the last node in the first

polynomial is added to the end of the resultant linked list.



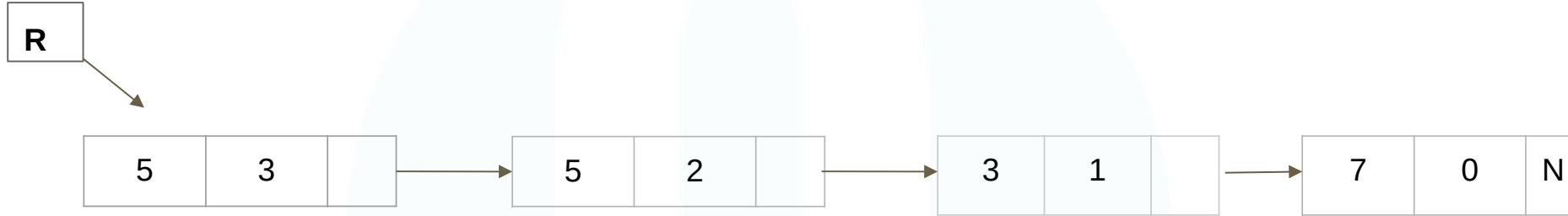
KtuQbank
Step 4: There is no node in the second polynomial to compare with. So, the last node in the first

polynomial is added to the end of the resultant linked list.



Step 5. Display the resultant linked list. The resultant linked list is pointed to by the pointer R

$$\mathbf{R = 5x^3 + 5x^2 + 3x + 7}$$



Memory Management

- Is the task carried out by the OS and hardware to accommodate multiple processes in main memory
- If only a few processes can be kept in main memory, then much of the time all processes will be waiting for I/O and the CPU will be idle
- Hence, memory needs to be allocated efficiently in order to pack as many processes into memory as possible

Memory Allocation

- **Memory allocation** is the process of setting aside sections of memory in a program to be used to store variables, and instances of structures and classes.
- There are two basic types of memory allocation:
 - Static Memory Allocation
 - Dynamic Memory Allocation

Static Memory Allocation

- When you declare a variable or an instance of a structure or class. The memory for that object is allocated by the operating system. The name you declare for the object can then be used to access that block of memory.

Declaration of variables, structures
and classes at the beginning of a
class or function



```
simple SimArray[50];  
simple sim1;  
int    x;  
double d;  
char   ch;
```

Dynamic Memory Allocation

- When you use dynamic memory allocation you have the operating system designate a block of memory of the appropriate size while the program is running.
- This is done either with the **new** operator or with a call to the **malloc** function. The block of memory is allocated and a pointer to the block is returned. This is then stored in a pointer to the appropriate data type.

Memory allocated while the program is running using calls to *new* (C++) or *malloc()* (C)



simple	*sptr;
int	*iptr;
double	*dptr;
char	*cptr;
MyList	*list;

Memory Management with

- Keep a linked list of allocated and free memory segments. Each node has the following information.

PID_or_hole, start_address, size, pointer_to_next



- Each entry in the list holds the following data
 - P or H : for Process or Hole
 - Starting segment address
 - The length of the memory segment
 - The next pointer
- With this representation, merging free regions after a release is easy

Memory Management with Linked Lists[cont..]

- A terminating process can have four combinations of neighbours
- If X is the terminating process the four combinations are:

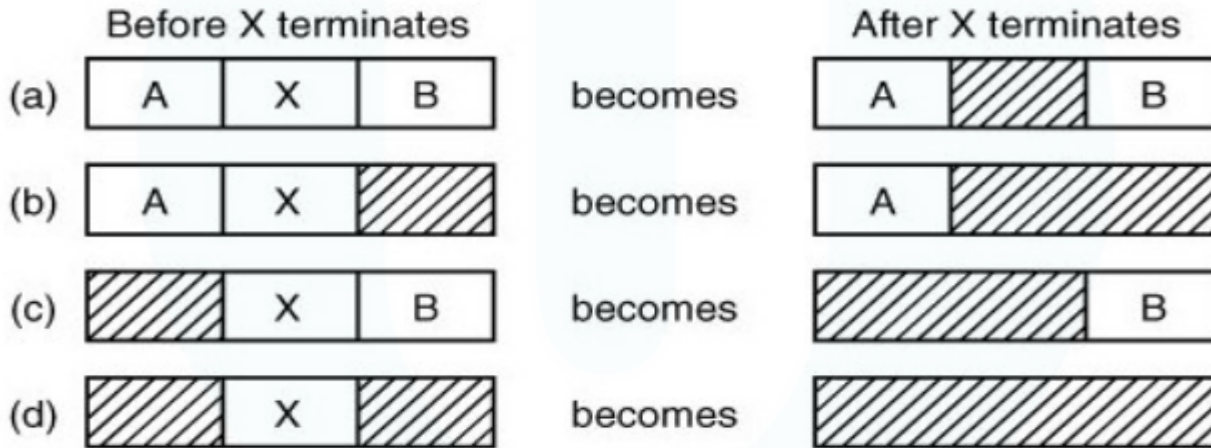


Fig: Four neighbor combinations for the terminating process, X.

Memory Management with Linked Lists[cont.]

- In the first option we simply have to replace the P by an H, other than that the list remains the same.
- In the second option we merge two list entries into one and make the list one entry shorter.
- Option three is effectively the same as option 2.
- For the last option we merge three entries into one and the list becomes two entries shorter.
- In order to implement this scheme it is normally better to have a **doubly linked list** so that we have access to the previous entry.

Memory Allocation Strategies

- **First Fit:**

The simplest algorithm is first fit. The process manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

Memory Allocation Strategies

● Next Fit:

It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

● Best Fit:

Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual

Memory Allocation Strategies

- **Worst Fit:**

Always take the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

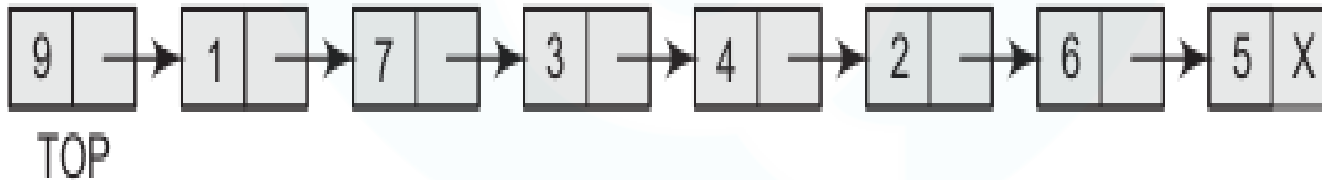
- **Quick Fit:**

maintains separate lists for some of the more common sizes requested. For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-KB holes, the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on. With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge is possible is expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

Implementation of Stacks and Queues Using Linked Lists

Linked Representation Of Stacks

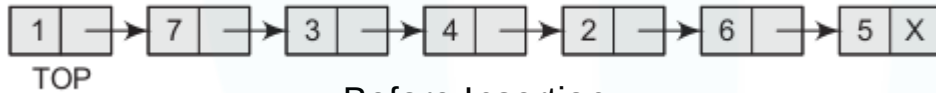
- Every node has two parts—one that stores data and another that stores the address of the next node.
- The **START** pointer of the linked list is used as **TOP** . All insertions and deletions are done at the node pointed by **TOP** .
- If **TOP = NULL** , then it indicates that the stack is **empty**.



Operations On a Linked Stack

● PUSH Operation:

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.



Before Insertion



After Inserting 9

Algorithm

Step 1: Allocate memory for the new node and name it as NEW_NODE

Step 2: SET NEW_NODE -> DATA = VAL

Step 3: IF TOP = NULL

 SET NEW_NODE -> NEXT = NULL

 SET TOP = NEW_NODE

ELSE

 SET NEW_NODE -> NEXT = TOP

 SET TOP = NEW_NODE

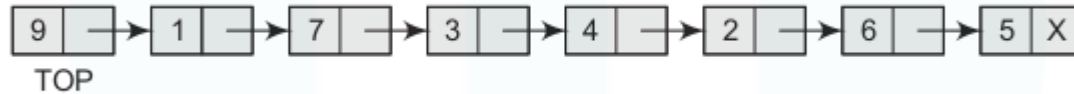
[END OF IF]

Step 4: END

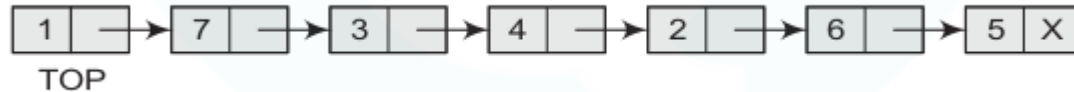
Operations On a Linked Stack

● Pop Operation:

The pop operation is used to delete the topmost element from a stack.



Linked Stack



After POP Operation

Algorithm

Step 1: IF TOP = NULL

PRINT " UNDERFLOW "

Goto Step 5

[END OF IF]

Step 2: SET PTR = TOP

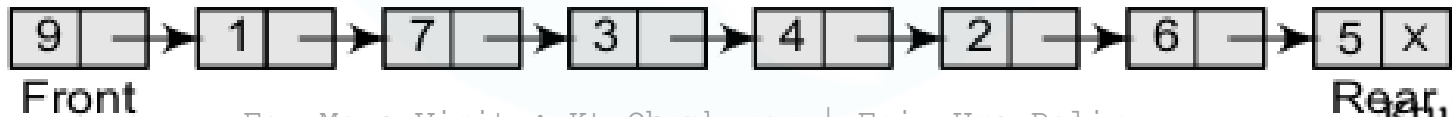
Step 3: SET TOP = TOP -> NEXT

Step 4: FREE PTR

Step 5: END

Linked Representation Of Queue

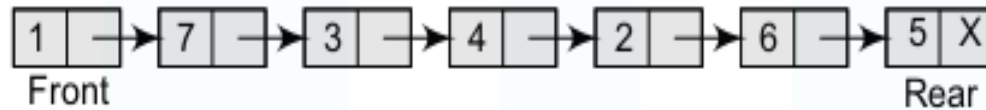
- Every element has two parts, one that stores the data and another that stores the address of the next element.
- The START pointer of the linked list is used as FRONT . Here, we will
- also use another pointer called REAR , which will store the address of the last element in the queue.
- All insertions will be done at the rear end and all the deletions will be done at the front end.
- If $\text{FRONT} = \text{REAR} = \text{NULL}$. then it indicates that the queue is empty.



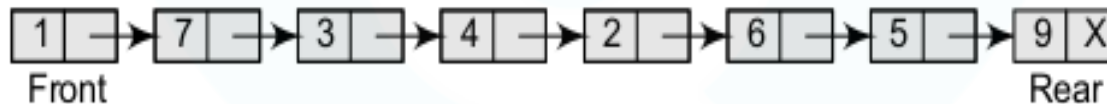
Operations on Linked Queues

● Enqueue:

The insert operation is used to insert an element into a queue. The new element is added as the last element of the queue.



Linked queue



Linked queue after inserting a new node

Step 1: Allocate memory for the new node and name it as PTR

Step 2: SET PTR -> DATA = VAL

Step 3: IF FRONT = NULL

SET FRONT = REAR = PTR

SET FRONT -> NEXT = REAR -> NEXT = NULL

ELSE

SET REAR -> NEXT = PTR

SET REAR = PTR

SET REAR -> NEXT = NULL

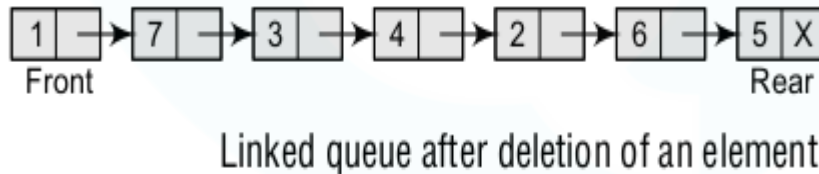
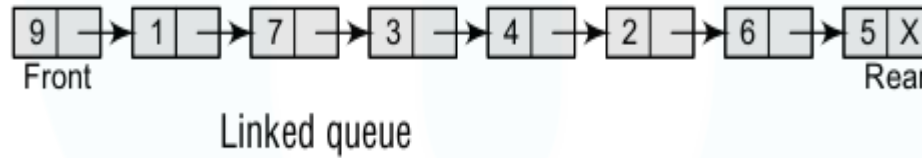
[END OF IF]

Step 4: END

Operations on Linked Queues

● Dequeue Operation:

The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT .



Algorithm

Step 1: IF FRONT = NULL

Write " Underflow "

Go to Step 5

[END OF IF]

Step 2: SET PTR = FRONT

Step 3: SET FRONT = FRONT -> NEXT

Step 4: FREE PTR

Step 5: END

Example:

1. Given Memory Partition: 100 KB, 300 KB, 150 KB, 650 KB, 450 K

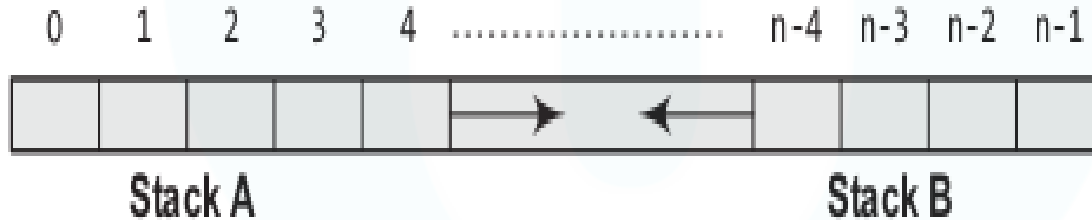
Processes of Size: 212 KB, 315 KB, 127 KB, 470 KB

2. Given Memory Partition: 100 KB, 500 KB, 200 KB, 300 KB, 600 KB

Processes of Size: 212 KB, 417 KB, 112 KB, 426 KB

Multiple Stacks

- More than one stack in the same array of sufficient size.



Assignment 1

1. What is a Priority Queue? Give its applications.
2. Explain about Multiple queues.
3. Give an algorithm to count the number of elements in a linked list.