Linear Data Structure	Non-linear Data Structure
In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

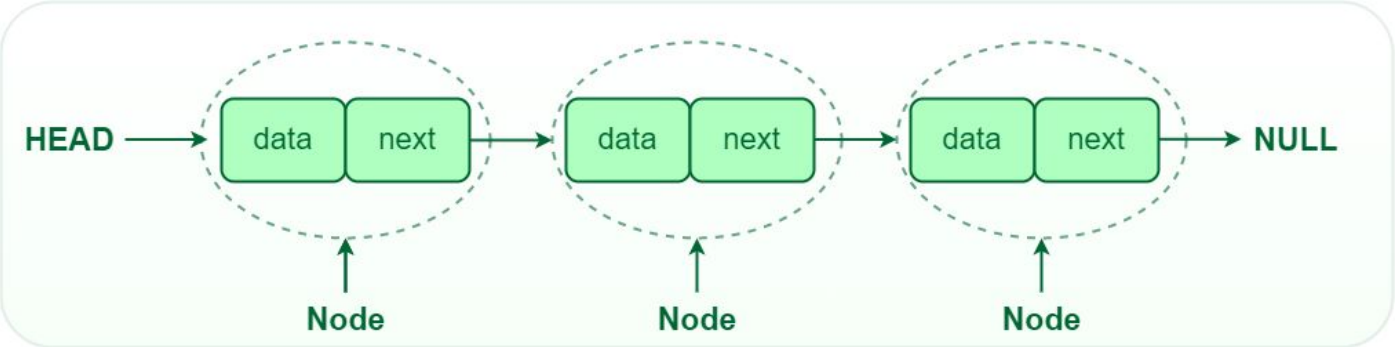
Linear Data Structures

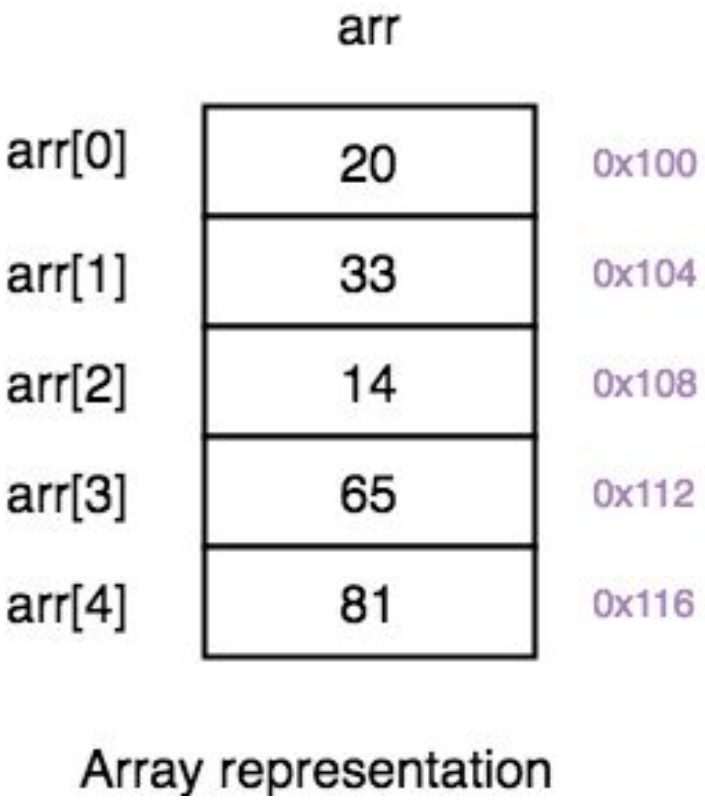
Linear list

- A sequence of elements
- There is first and last element
- Each element has previous and next
  - Nothing before first
  - Nothing after last

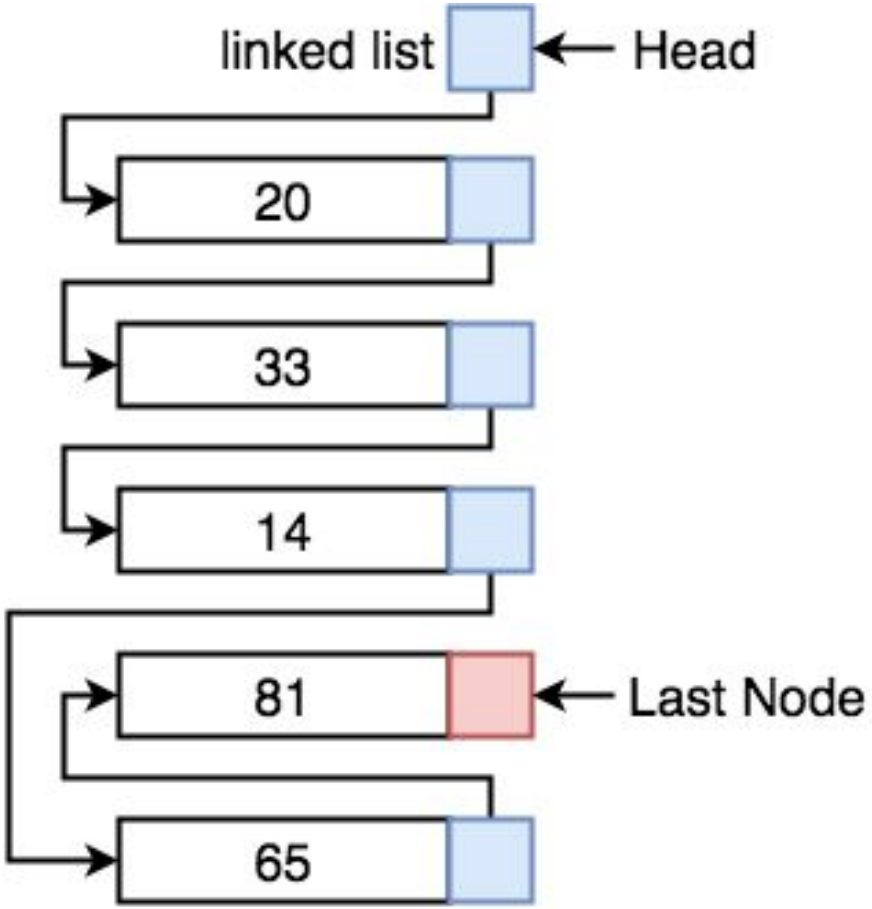
Why linked lists?

- A linked list is a dynamic data structure.
  - It can grow or shrink in size during the execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.
- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.





Array representation



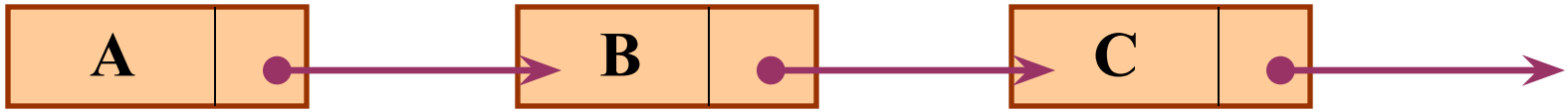
Array Vs. link list

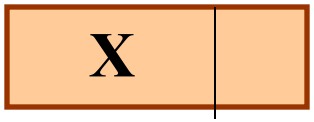
Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements 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]

Operations on LL

- What we can do with a linear list?
  - Delete element
  - Insert element
  - Find element
  - Traverse list

Illustration: Insertion





Item to be inserted

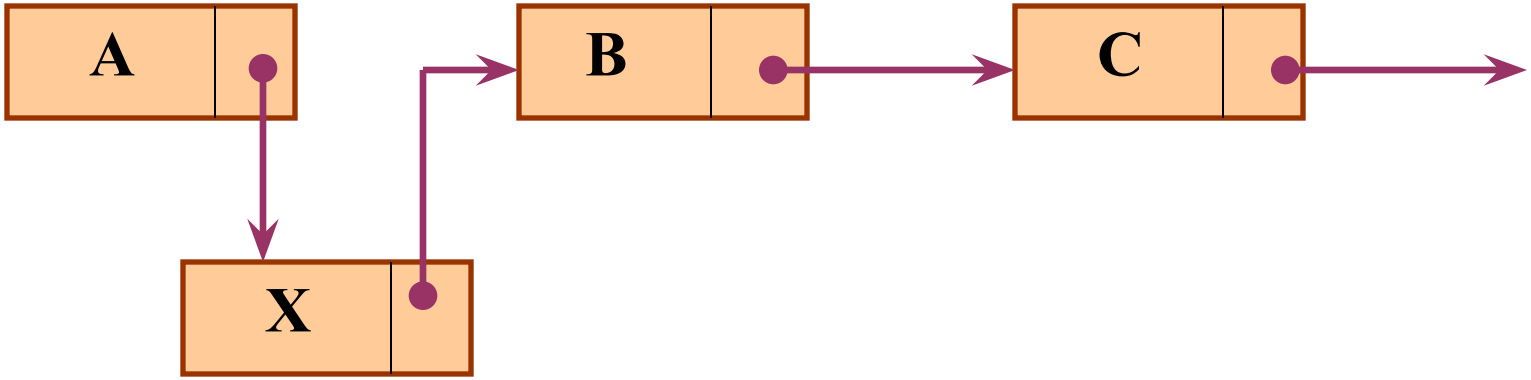
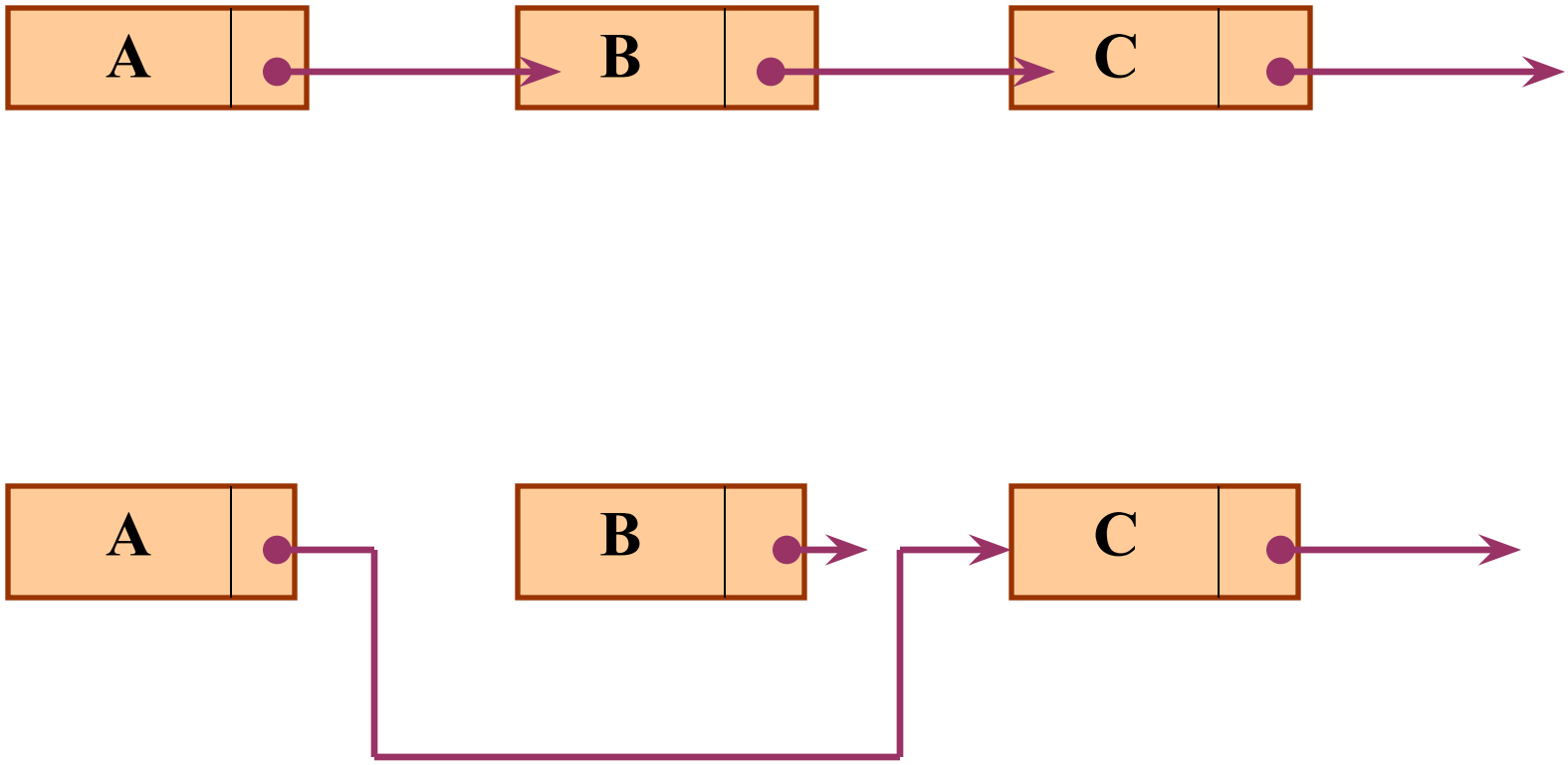
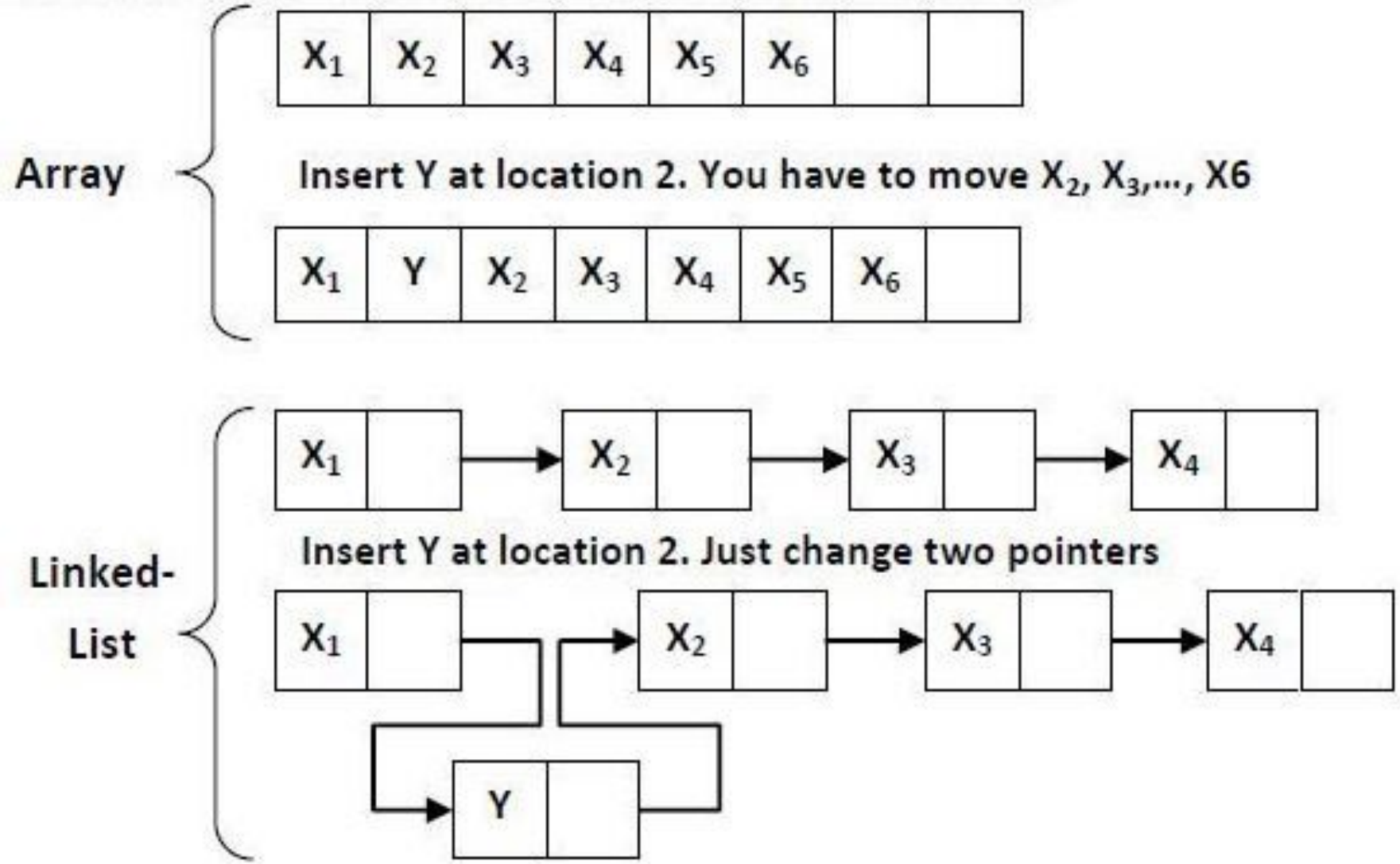


Illustration: Deletion



In essence ...

- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link it to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.


Array
X1 X2 X3 X4 X5 X6
Insert Y at location 2. You have to move X2, X3,…, X6
X1 Y X2 X3 X4 X5 X6
Linked-List
X1 X2 X3 X4
Insert Y at location 2. Just change two pointers
X1 X2 X3 X4
Y

Traverse: list ⇨ elements in order

- get_first(list) -
  - returns first element if it exists
- get_next(list) -
  - returns next element if it exists
- Both functions return NULL otherwise
- Calling get_next in a loop we will get one by one all elements of the list

How we can implement a list?

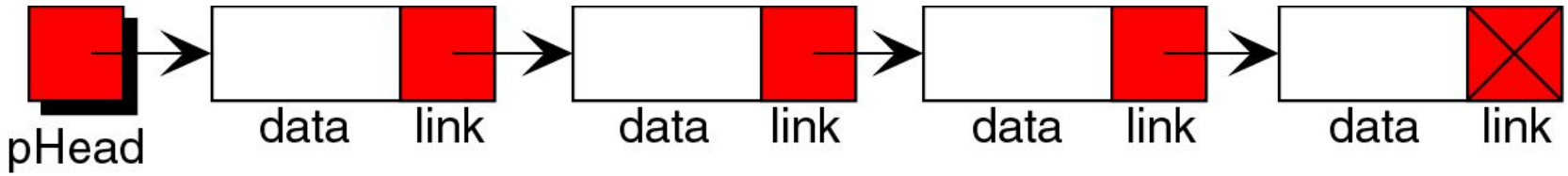
- Array?
- Search is easy (sequential or binary)
- Traversal is easy:

```
for(i = first; i <= last; ++i)
    process(a[i]);
```

- Insert and delete is *not* easy
  – a good part of the array has to be moved!
- Hard to guess the size of an array

A *linked list* implementation

- Linked list is a chain of elements
- Each element has data part and link part pointing to the next element



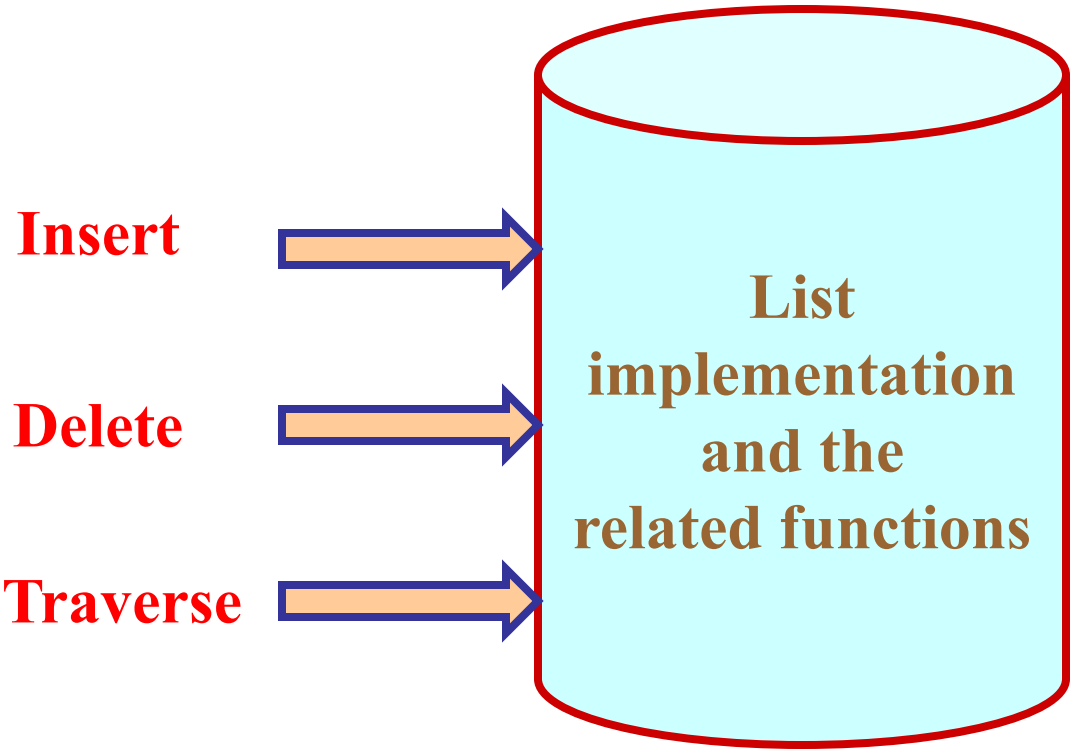
(a) A linked list with a head pointer: pHead

(b) An empty linked list

Main operations

- Create list
- Add node
  - beginning,middle or end
- Delete node
  - beginning,middle or end
- Find node
- Traverse list

Conceptual Idea



https://www.programiz.com/dsa/linked-list-operations

Example: Working with linked list

- Consider the structure of a node as follows:

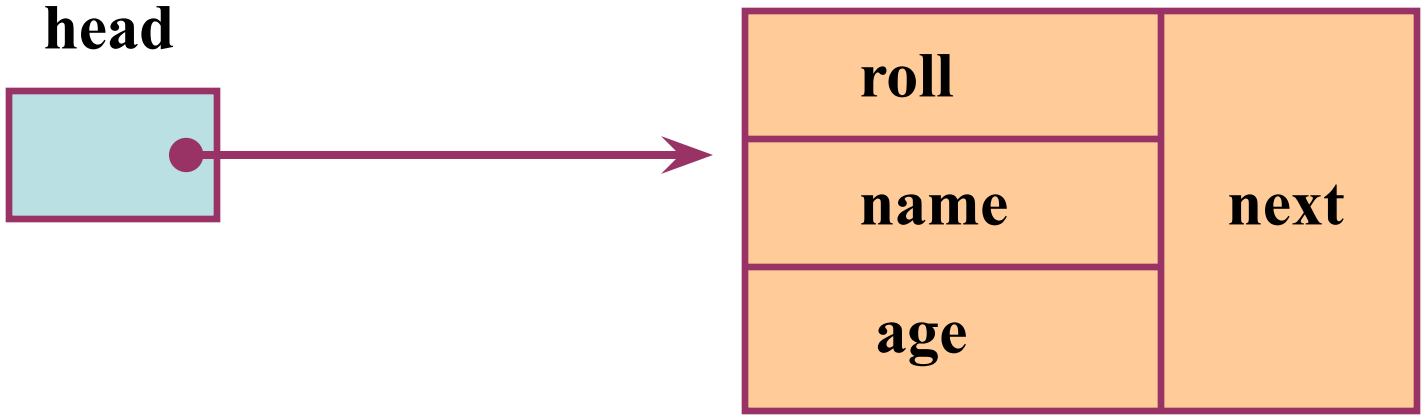
```
struct  stud  {
                int     roll;
                char  name[25];
                int     age;
                struct  stud  *next;
              };

        /* A user-defined data type called “node” */
typedef  struct  stud  node;
node  *head;
```

##typedef is used to define a data type in C.

Creating a List

- To start with, we have to create a node (the first node), and make head point to it.

```
head = (node *) malloc (sizeof (node));
```



Contd.

- If there are n number of nodes in the initial linked list:
  - Allocate n records, one by one.
  - Read in the fields of the records.
  - Modify the links of the records so that the chain is formed.

```
void  create_list  (node  *list)
{
   int  k, n;
   node  *p;
   printf  ("\n How many elements?");
   scanf ("%d", &n);

   list  =  (node *)  malloc (sizeof (node));
   p = list;
   for  (k=0; k<n; k++)
   {
      scanf ("%d %s %d", &p->roll,
                          p->name, &p->age);
      p->next  = (node *) malloc
                              (sizeof (node));
      p  =  p->next;
   }
   free (p->next);
   p->next  =  NULL;
}
```

To be called from the main() function as:

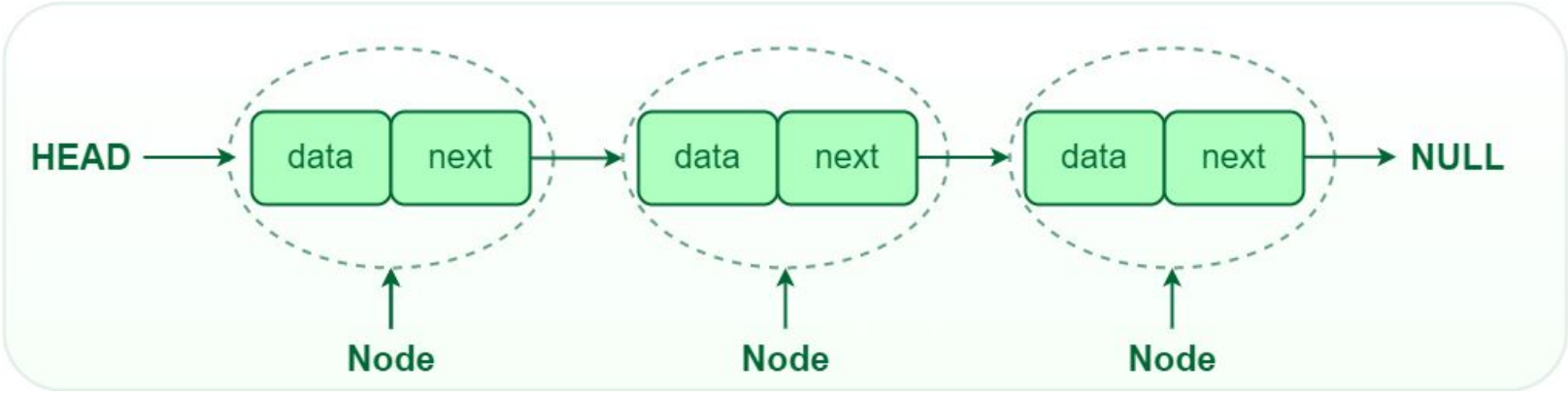
```
node  *head;
.......
create_list (head);
```

sizeof() is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to `malloc`.

malloc() is used to dynamically allocate a single block of memory in C, it is available in the header file `stdlib.h`.

Traversing the List

- Once the linked list has been constructed and head points to the first node of the list,
  - Follow the pointers.
  - Display the contents of the nodes as they are traversed.
  - Stop when the next pointer points to NULL.



Single-linked list

```
void  display_list  (node  *list)
{
   int  k = 0;
   node  *p;

   p = list;
   while  (p != NULL)
   {
     printf (“Node %d:  %d %s %d”, k, p->roll,
                                        p->name, p->age);
    k++;
     p  =  p->next;
   }
}
```

Inserting a Node in the List

- The problem is to insert a node before a specified node.
  - Specified means some value is given for the node (called key).
  - Here it may be roll.
- Convention followed:
  - If the value of roll is given as negative, the node will be inserted at the end of the list.

- When a node is added at the beginning,
  - Only one next pointer needs to be modified.
    - head is made to point to the new node.
    - New node points to the previously first element.
- When a node is added at the end,
  - Two next pointers need to be modified.
    - Last node now points to the new node.
    - New node points to NULL.
- When a node is added in the middle,
  - Two next pointers need to be modified.
    - Previous node now points to the new node.
    - New node points to the next node.

```
void  insert_node  (node  *list)
{
   int  k = 0, rno;
   node  *p, *q, *new;

   new  =  (node *)  malloc (sizeof (node));
   scanf ("%d %s %d", &new->roll, new->name,
                                        &new->age);

   printf ("\nInsert before roll (-ve for end):");
   scanf ("%d", &rno);

   p = list;
   if  (p->roll  ==  rno)      /* At the beginning */
   {
      new->next  =  p;
      list  =  new;
   }
```

```
   while  ((p != NULL) && (p->roll != rno))
   {
      q = p;
      p  =  p->next;
   }

   if  (p == NULL)          /* At the end */
   {
      q->next = new;
      new->next = NULL;
    }

    if  (p->roll  == rno)     /* In the middle */
   {
      q->next = new;
      new->next = p;
   }
}
```

The pointers q and p always point to consecutive nodes.

Deleting an Item

- Here also we are required to delete a specified node.
  - Say, the node whose roll field is given.
- Here also three conditions arise:
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.

```
void  delete_node  (node  *list)
{
   int  rno;
   node  *p, *q;

   printf ("\nDelete for roll :");
   scanf ("%d", &rno);

   p = list;
   if  (p->roll == rno)          /* Delete the first element */
   {
      list = p->next;
      free (p);
   }
```

```
  while  ((p != NULL) && (p->roll != rno))
   {
      q = p;
      p  =  p->next;
   }


   if  (p == NULL)              /* Element not found */
       printf ("\nNo match :: deletion failed");

   if  (p->roll == rno)         /* Delete any other element */
   {
      q->next  =  p->next;
      free (p);
   }
}
```

https://www.log2base2.com/data-structures/linked-list/linked-list-in-c.html

Types of Linked List

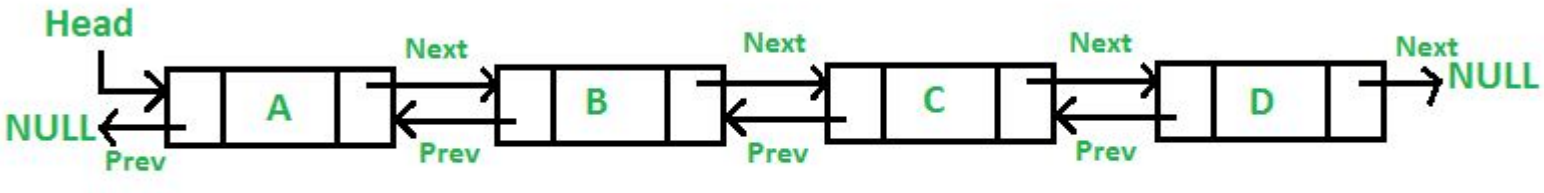
Singly Linked List: A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.

To create and display Singly Linked List:

https://www.w3resource.com/c-programming-exercises/linked_list/c-linked_list-exercise-1.php (code is well commented)**

https://www.javatpoint.com/program-to-create-and-display-a-singly-linked-list

Doubly linked list



A **doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field.

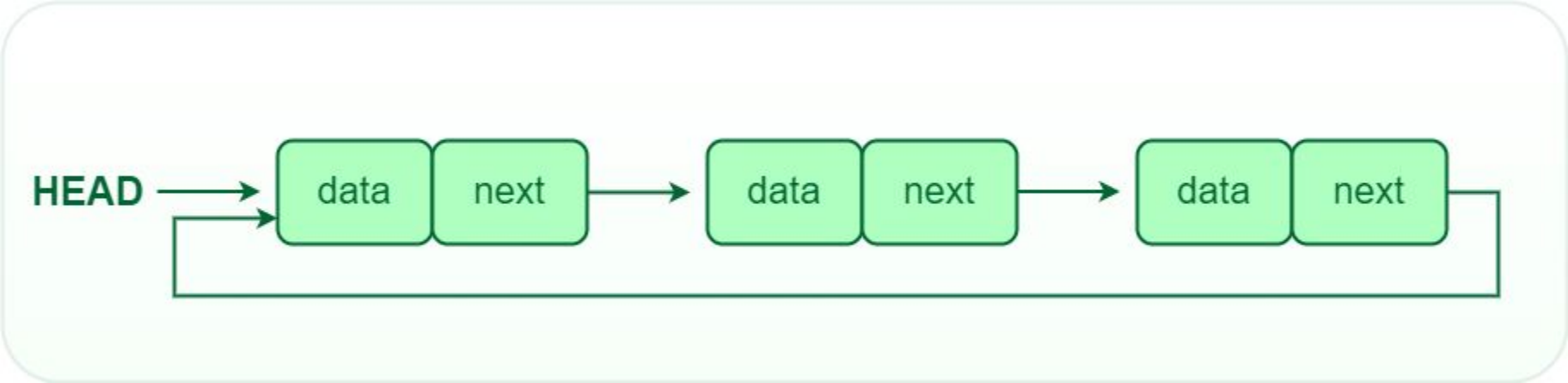
In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it **requires additional memory for the backward reference.**

Code: https://www.programiz.com/dsa/doubly-linked-list

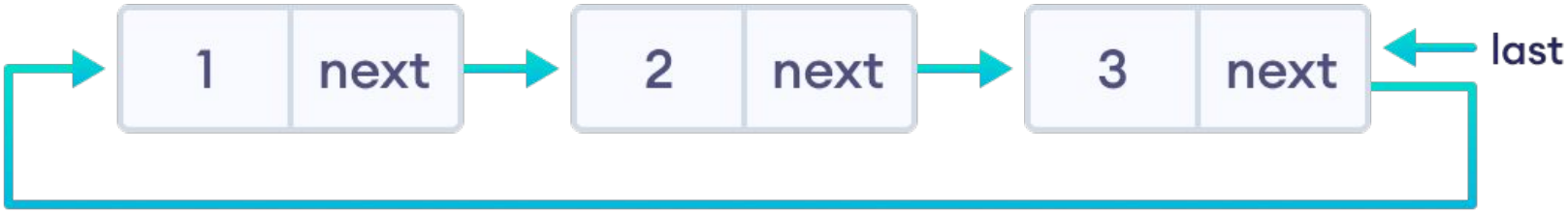
A doubly linked list is a type of linked list in which each node consists of 3 components:

- *prev - address of the previous node
- data - data item
- *next - address of next node

- **Circular Linked List:** The Circular Linked List is similar to the Singly Linked List. The only key difference is that the **last node contains the address of the first node,** forming a **circular loop** in the Circular Linked List. It can be either singly or doubly linked.



Circular Linked List



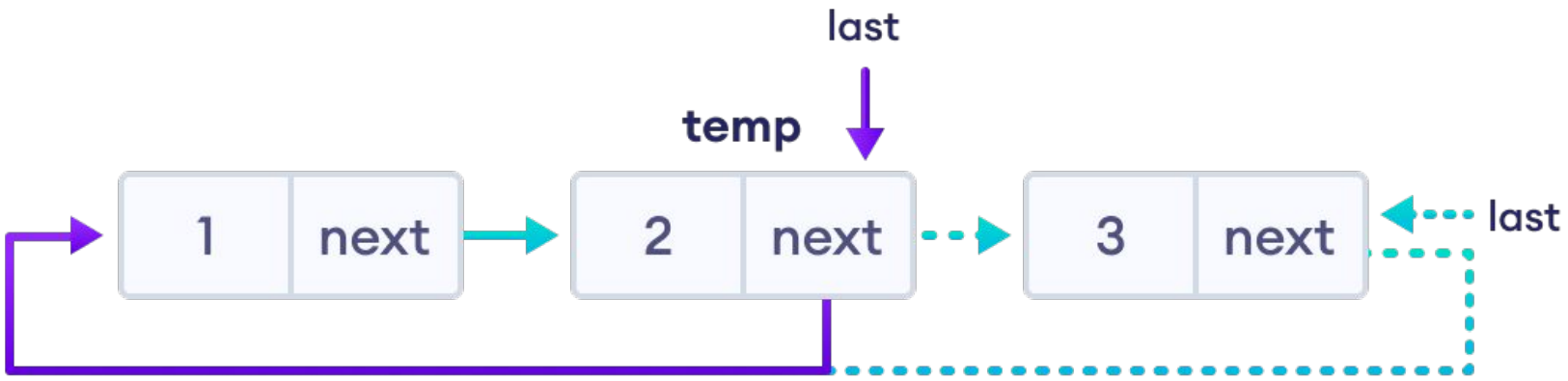
1. If the node to be deleted is the only node

- free the memory occupied by the node
- store NULL in `last`

2. If last node is to be deleted

- find the node before the last node (let it be `temp`)
- store the address of the node next to the last node in `temp`
- free the memory of last
- make `temp` as the last node

Deletion of Last Node



3. If any other nodes are to be deleted

- travel to the node to be deleted (here we are deleting node 2)
- let the node before node 2 be `temp`
- store the address of the node next to 2 in `temp`
- free the memory of 2

Code: https://www.programiz.com/dsa/circular-linked-list

Counting Nodes in Circular LL

Code:
https://www.educative.io/answers/how-to-count-nodes-in-a-circular-linked-list

What is a Polynomial?

What is Polynomial?

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + ... + jx + k$), where a, b, c ...., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

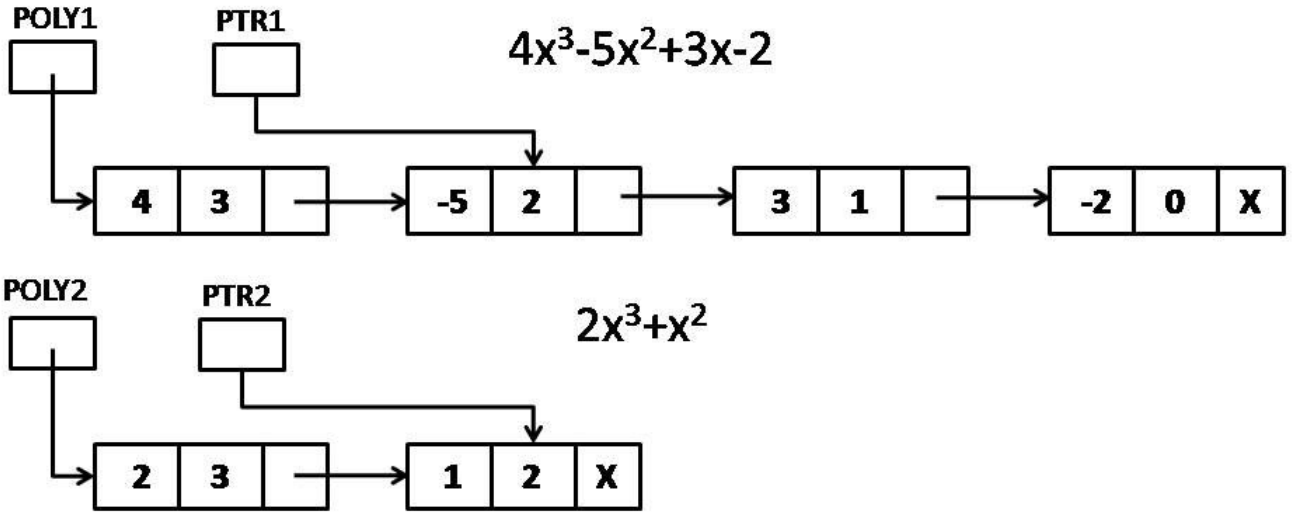
- one is the coefficient
- other is the exponent

Example:

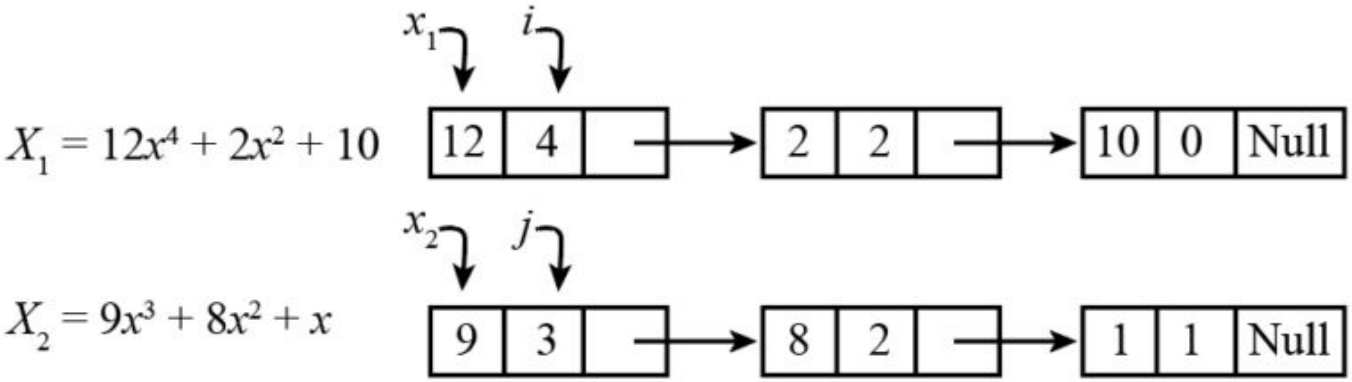
$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

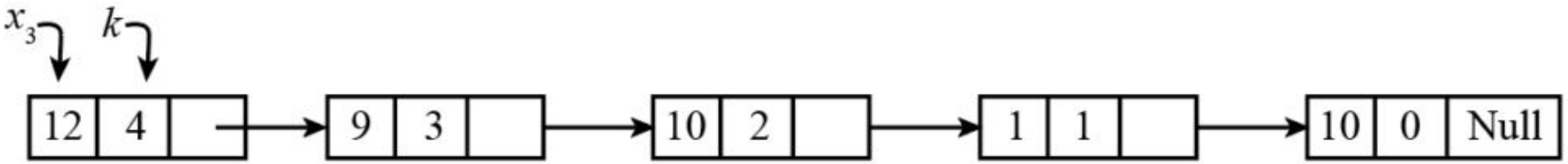
- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent
- For adding two polynomials that are stored as a linked list, we need to add the coefficients of variables with the same power.



Adding two polynomials using Linked List



The resultant liked list :-



Code & Steps: https://www.javatpoint.com/application-of-linked-list

https://mycareerwise.com/programming/category/linked-list/polynomial-addition-using-linked-list-313

Linked List Applications

- Dynamic memory allocation
- Implemented in stack and queue
- In undo functionality of softwares
- Hash tables, Graphs

Advantages of Linked Lists

. **Dynamic Size:** Linked lists can grow or shrink dynamically, as memory allocation is done at runtime.
. **Insertion and Deletion:** Adding or removing elements from a linked list is efficient, especially for large lists.
. **Flexibility:** Linked lists can be easily reorganized and modified without requiring a contiguous block of memory.

Disadvantages of Linked Lists

. **Random Access:** Unlike arrays, linked lists do not allow direct access to elements by index. Traversal is required to reach a specific node.
. **Extra Memory:** Linked lists require additional memory for storing the pointers, compared to arrays.

Applications of Singly Linked List are as following:

1. It is used to implement **stacks** and **queues** which are like fundamental needs throughout computer science.
2. To prevent the collision between the data in the **hash map**, we use a singly linked list.
3. If we ever noticed the functioning of a casual notepad, it also uses a singly linked list to perform undo or redo or deleting functions.
4. We can think of its use in a photo viewer for having look at photos continuously in a slide show.
5. In the system of train, the idea is like a singly linked list, as if you want to add a Boggie, either you have to take a new boggie to add at last or you must spot a place in between boggies and add it.

Applications of Circular Linked List are as following:

1. It can also be used to implement queues by maintaining a pointer to the last inserted node and the front can always be obtained as next of last.
2. Circular Doubly Linked Lists are used for the implementation of advanced data structures like Fibonacci Heap. Also reference to the previous node can easily be found in this.
3. It is also used by the Operating system to share time for different users, generally uses a Round-Robin time-sharing mechanism (this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking).
4. Multiplayer games use a circular list to swap between players in a loop.
5. In photoshop, word, or any paint we use this concept in undo function.

Applications of Doubly Linked List are as following:

1. Great use of the doubly linked list is in navigation systems, as it needs front and back navigation.
2. In the browser when we want to use the back or next function to change the tab, it used the concept of a doubly-linked list here.
3. Again, it is easily possible to implement other data structures like a binary tree, hash tables, stack, etc.
4. It is used in music playing system where you can easily play the previous one or next one song as many times one person wants to. Basically it provides full flexbility to perform functions and make the system user-friendly.
5. In many operating systems, the thread scheduler (the thing that chooses what processes need to run at which times) maintains a doubly-linked list of all the processes running at any time. This makes it easy to move a process from one queue (say, the list of active processes that need a turn to run) into another queue (say, the list of processes that are blocked and waiting for something to release them).
6. It is used in a famous game concept which is a deck of cards.