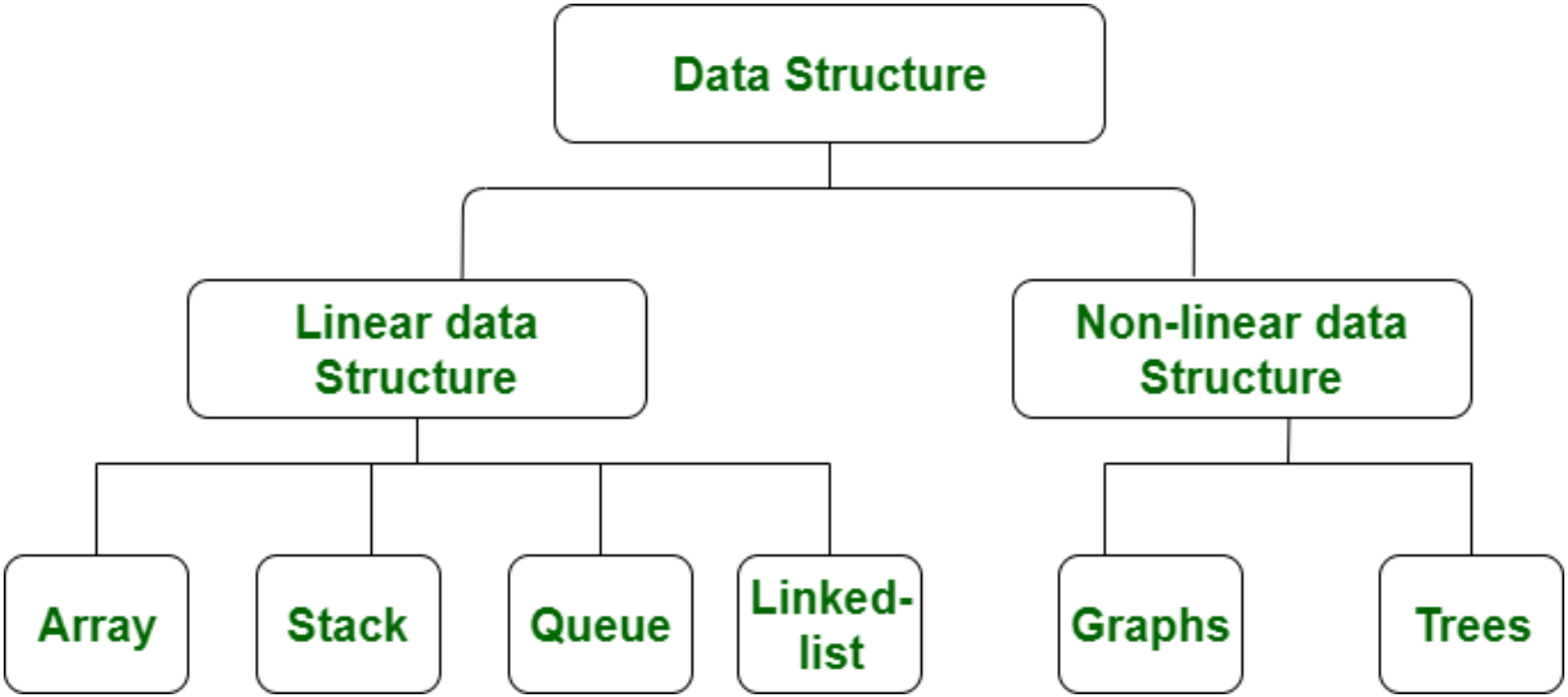
UNIT- III

LINKED LISTS

Contents

Concept, Comparison of sequential and linked organizations, Primitive operations, Realization of Linked Lists, Realization of linked list using arrays, Dynamic Memory Management, Linked list using dynamic memory management, Linked List Abstract Data Type, Linked list operations, Head pointer and header node, Types of linked list- Linear and circular linked lists, Doubly Linked List And operations, Circular Linked List, Singly circular linked list, doubly circular linked list, Polynomial Manipulations - Polynomial addition, Multiplication of two polynomials using linked list. Generalized Linked List (GLL) concept, representation of polynomial and sets using GLL. Case Study- Garbage Collection.

Introduction



Drawbacks of an array

- It is a static data structure, that is, the maximum capacity of an array should be known before the compilation process.
- Another drawback of arrays is that the elements in an array are stored a fixed distance apart, and the insertion and deletion of elements in between require a lot of data movement.

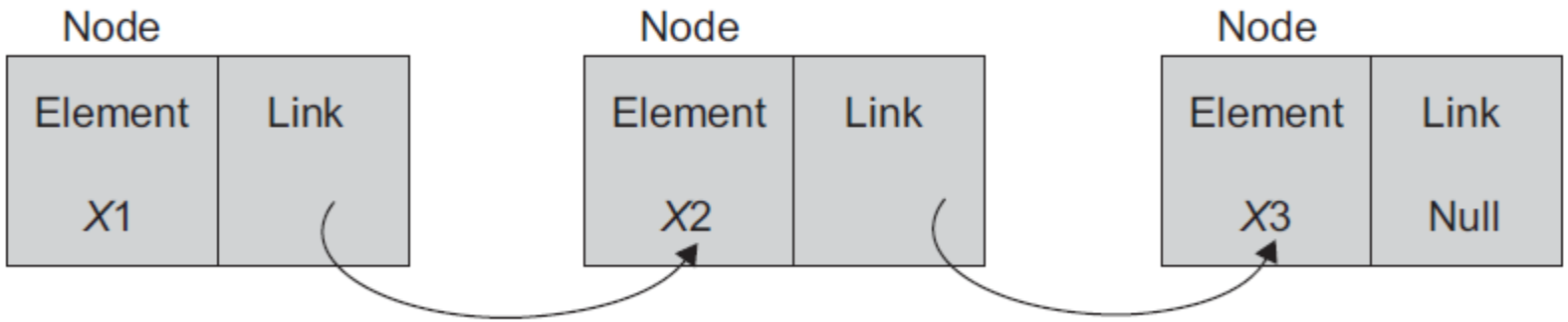
Solution

Link List

- Arrays and linked lists are examples of linear lists.
- Linear lists are those in which each member has a unique successor.
- Arrays contain consecutive memory locations that are a fixed distance apart, whereas linked lists do not necessarily contain consecutive memory locations.
- These data items can be stored anywhere in the memory in a scattered manner.
- To maintain the specific sequence of these data items, we need to maintain link(s) with a successor (and/or a predecessor).
- It is called as a *linked list* as each node is linked with its successor (and/or predecessor).

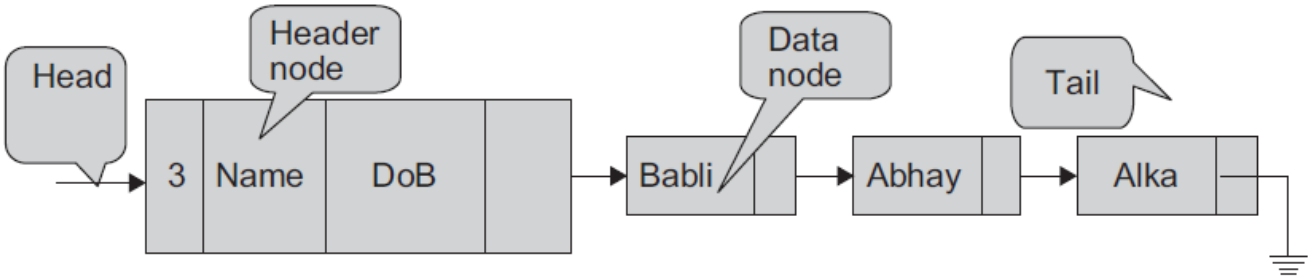
Linked List

- A *linked list* is an ordered collection of data in which each element (node) contains a minimum of two values
- 1. The data member(s) being stored in the list.
- 2. A pointer or link to the next element in the list.



Linked List Terminology

- The following terms are commonly used in discussions about linked lists:
- ***Header node*** A header node is a special node that is attached at the beginning of the linked list. This header node may contain special information (metadata) about the linked list.



Linked List Terminology

- The following terms are commonly used in discussions about linked lists:
- ***Data node*** The list contains data nodes that store the data members and link(s) to its predecessor (and/or successor).

Linked List Terminology

- ***Head pointer*** The variable , which represents the list, is simply a pointer to the node at the head of the list.
- ***Tail pointer*** Similar to the head pointer that points to the first node of a linked list, we may have a pointer pointing to the last node of a linked list called the *tail pointer*.

Comparison of Sequential and Linked Organizations

Sequential Organizations	Linked Organizations
Successive elements of a list are stored a fixed distance apart.	Elements can be placed anywhere in the memory
It provides *static allocation*, which means, the space allocation done by a compiler once cannot be changed during execution, and the size has to be known in advance.	Dynamic allocation (size need not be known in advance), that is, space allocation as per need can be done during execution.
As individual objects are stored a fixed distance apart, we can access any element randomly.	As objects are not placed in consecutive locations at a fixed distance apart, random access to elements is not possible.

Comparison of Sequential and Linked Organizations

Sequential Organizations	Linked Organizations
Insertion and deletion of objects in between the list require a lot of data movement	Insertion and deletion of objects do not require any data shifting.
It is space inefficient for large objects with frequent insertions and deletions	It is space efficient for large objects with frequent insertions and deletions.
An element need not know/store and keep the address of its successive element	Each element in general is a collection of data and a link. At least one link field is a must.

Primitive Operations

The following are basic operations associated with the linked list as a data structure:

1. Creating an empty list
2. Inserting a node
3. Deleting a node
4. Traversing the list
5. Searching a node
6. Updating a node

Primitive Operations

Some more operations, which are based on the basic operations, are as follows:

7. Printing the node or list
8. Counting the length of the list
9. Reversing the list
10. Sorting the list using pointer manipulation
11. Concatenating two lists
12. Merging two sorted lists into a third sorted list

Realization of linked lists

- In a linked organization, the data elements are not necessarily placed in continuous locations.
- The relationship between data elements is by means of a link.
- Along with each data element, the address of the next element is stored.

Realization of linked listS

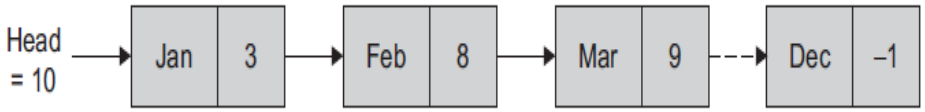
- Realization of Linked List Using Arrays
- Linked List Using Dynamic Memory Management

Realization of Linked List Using Arrays

- Let *L* be a set of names of months of the year.
- *L* = {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}
- Here, *L* is an ordered set.
- The elements of the list are stored in the one-dimensional array, **Data**.
- The elements are not stored in the same order as in the set *L*. They are also not stored in a continuous block of locations.
- To maintain the sequence, the second array, **Link**, is added.
- The values in this array are the links to each successive element. Here, the list starts at the 10th location of the array.
- Let the variable Head denote the start of the list.

Realization of Linked List Using Arrays

Index		Data	Link
1		Jun	4
2		Sep	7
3		Feb	8
4		Jul	12
5			
6		Dec	–1
7		Oct	14
8		Mar	9
9		Apr	11
10	Head →	Jan	3
11		May	1
12		Aug	2
13			
14		Nov	6
15			



Linked organization

Realization of linked list using arrays

Realization of Linked List Using Arrays

- Even though data and link are shown as two different arrays, they can be implemented using one 2D array as follows:
- int Linked_List[max][2];
- the realization of a linked list using a 2D array where *L* = {100, 102, 20, 51, 83, 99, 65}, Max = 10 and Head = 2.

	Index	Data	Link
	0	20	3
	1	99	7
Head →	2	100	5
	3	51	6
	4		
	5	102	0
	6	83	1
	7	65	–1
	8		
	9		

Realization of Linked List Using Dynamic Memory Management

- The users can dynamically allocate and de-allocate memory using new and delete operator.
- The **new operator** denotes a request for memory allocation on the Free Store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.
- Example:
- int *p = NULL; // Pointer initialized with NULL
- p = new int; // Then request memory for the variable

Realization of Linked List Using Dynamic Memory Management

- int *p = new int[10]
- Dynamically allocates memory for 10 integers continuously of type int and returns a pointer to the first element of the sequence

Realization of Linked List Using Dynamic Memory Management

- delete operator used to deallocate dynamically allocated memory.
- delete p;
- delete q;

data structure of node

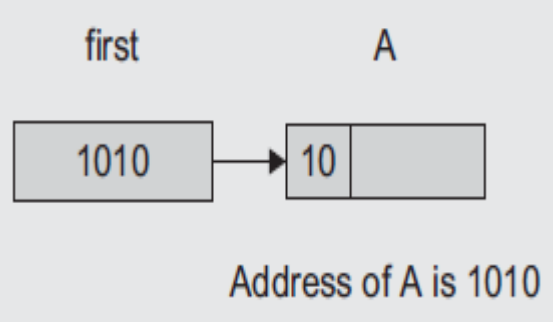
- As every node is a group of two (or more) data elements which are of different data types, they are logically grouped using the data type, *object*.
- The declaration of the data structure of a node is given as follows:

```
class Node
{
public:
   int data;
   Node *link;
};
```

- The statement Node *link defines the link field of a node. Here, Node is a data type of the pointer variable link.

- Consider the following piece of code:

```
class Node
{
public:
    int data;
    Node *link;
} *first, A;
first = &A;
A.data = 10;
A.link = Null;
```



Insertion of a node

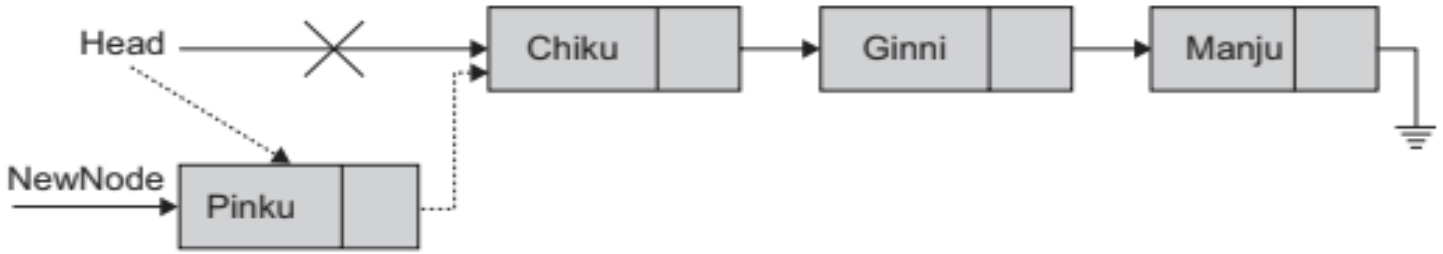
- Depending on the type of list or need of the user, insertion can be made at the
  - Beginning
  - Middle
  - At the end of the list

Insertion of a Node at the First Position

- The following two steps will insert NewNode at the beginning of the linked list.

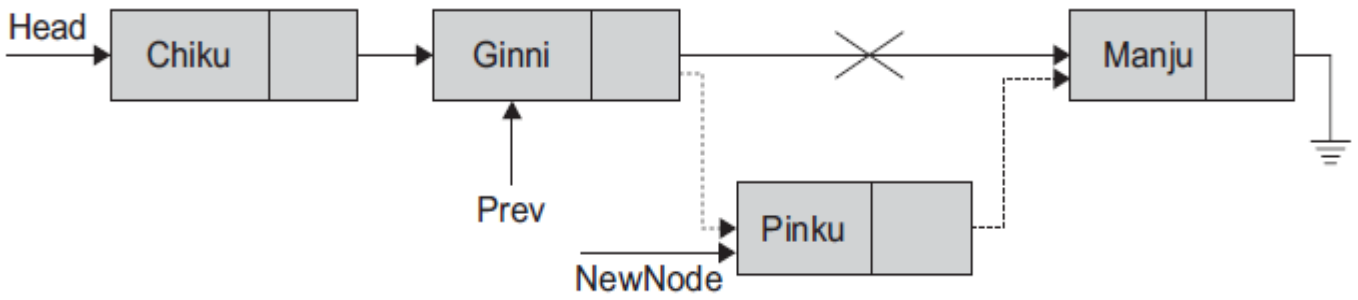
  NewNode->link = Head;

  Head = NewNode;



Insertion of a Node at a Middle Position

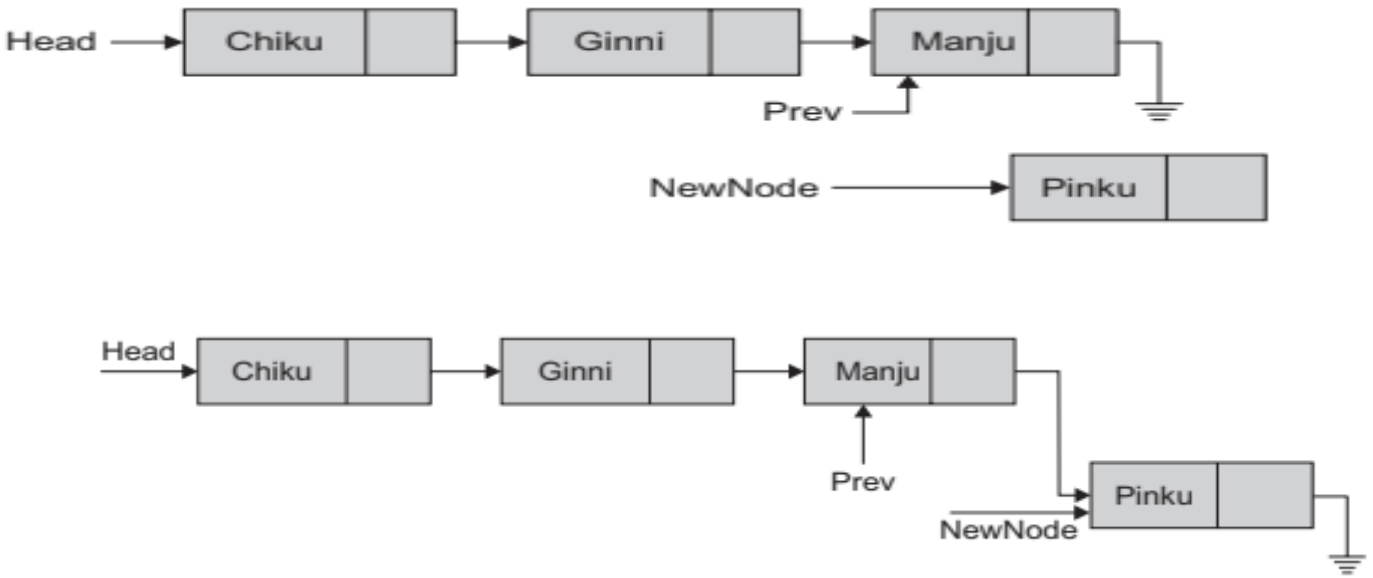
- Assume that a node is to be inserted at some position other than the first position.
- Let Prev refer to the node after which NewNode node is to be inserted.
- We need the following two steps:
- NewNode->link = Prev->link;
- Prev->link = NewNode;



Insertion of a Node at the End

- NewNode->link = Prev->link

  Prev->link = NewNode;



Deleting the First Node

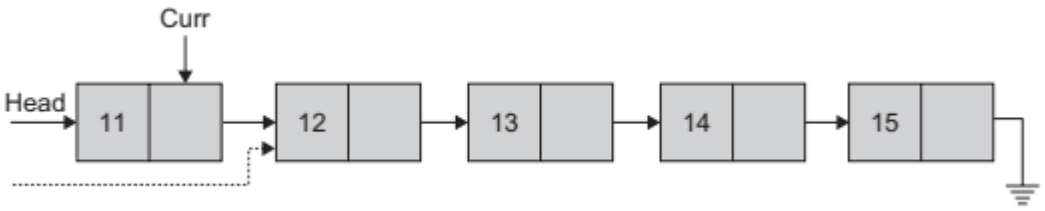
- This can be accomplished by the statements,

  Curr = Head;

  Head = Head->link;

- 2. Now, release the memory allocated for the first node.

  delete Curr;



Deleting a Middle Node

- Let curr point to the node to be deleted, and prev be the predecessor of curr. Then, the following statements will delete the node curr.

  prev->link = curr->link;

  delete curr;

Types of Linked List

- linked lists can be classified broadly as follows:

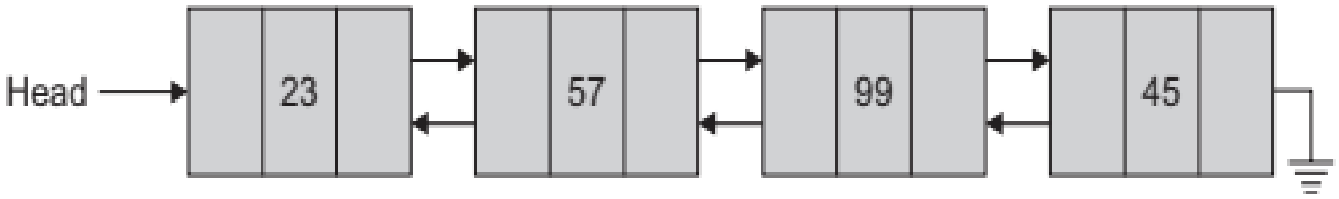
  1. Singly linked list
  2. Doubly linked list
  3. Circular linked list
  4. Circular Doubly linked list

Singly Linked List

- A linked list in which every node has one link field, to provide information about where the next node of the list is, is called as *singly linked list* (SLL).
- It has no knowledge about where the previous node lies in the memory.
- In SLL, we can traverse only in one direction.

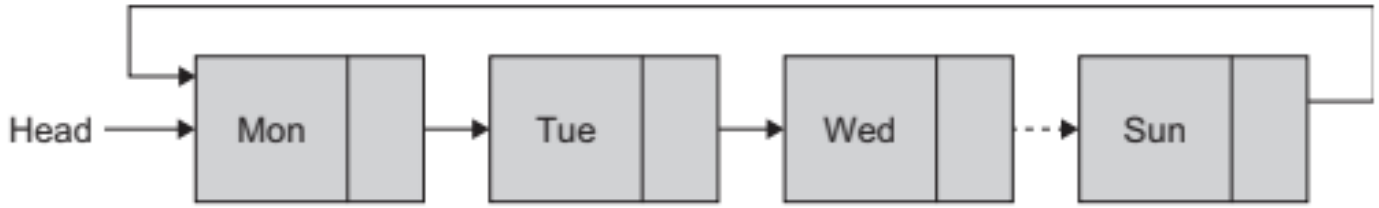
Doubly Linked List

- In a doubly linked list (DLL), each node has two link fields to store information about the one to the next and also about the one ahead of the node.
- Hence, each node has knowledge of its successor and also its predecessor.
- In DLL, from every node, the list can be traversed in both the directions



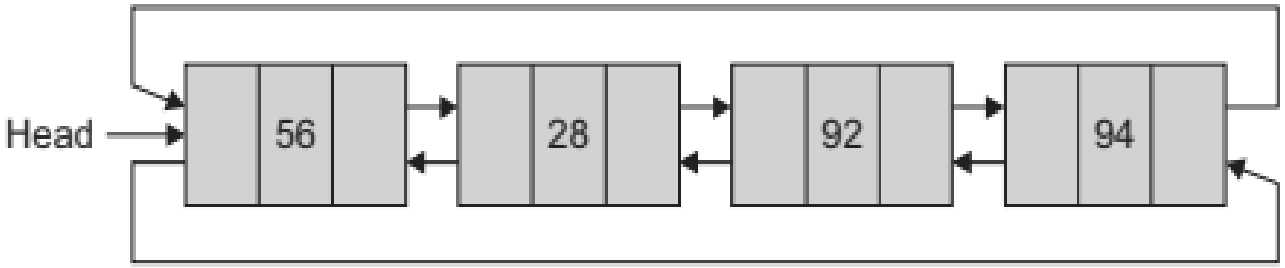
Circular Linked List

- The **circular linked list** is a linked list where all nodes are connected to form a circle.
- In a circular linked list, the first node and the last node are connected to each other which forms a circle.
- There is no NULL at the end.



Circular Doubly Linked List

- Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.



Doubly Linked List operation

- Deletion of a node from a doubly Linked List
- Insertion of a node in a doubly Linked List

Deletion of a node from a doubly Linked List

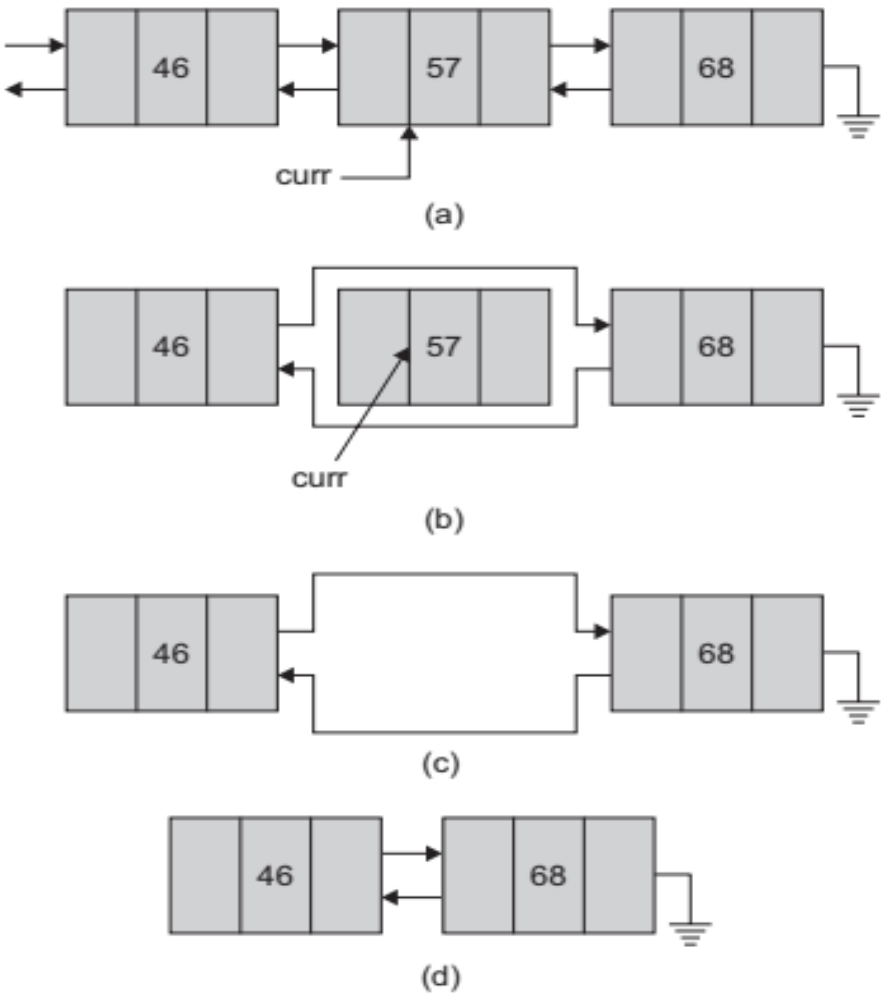
- The core steps involved in this process are the following:

  (curr->Prev)->Next = curr->Next;

  (curr->Next)->Prev = curr->Prev;

  delete curr;

Deletion of a node from a doubly Linked List



Insertion of a node in a doubly Linked List

- Let us assume that the node Current is to be inserted in between the two nodes say node1 and node2.
- We have to modify the following links:

  node1->Next, node2->Prev, Current->Prev, and Current->Next
- The statements to insert a node in between node1 and node2 are as follows:

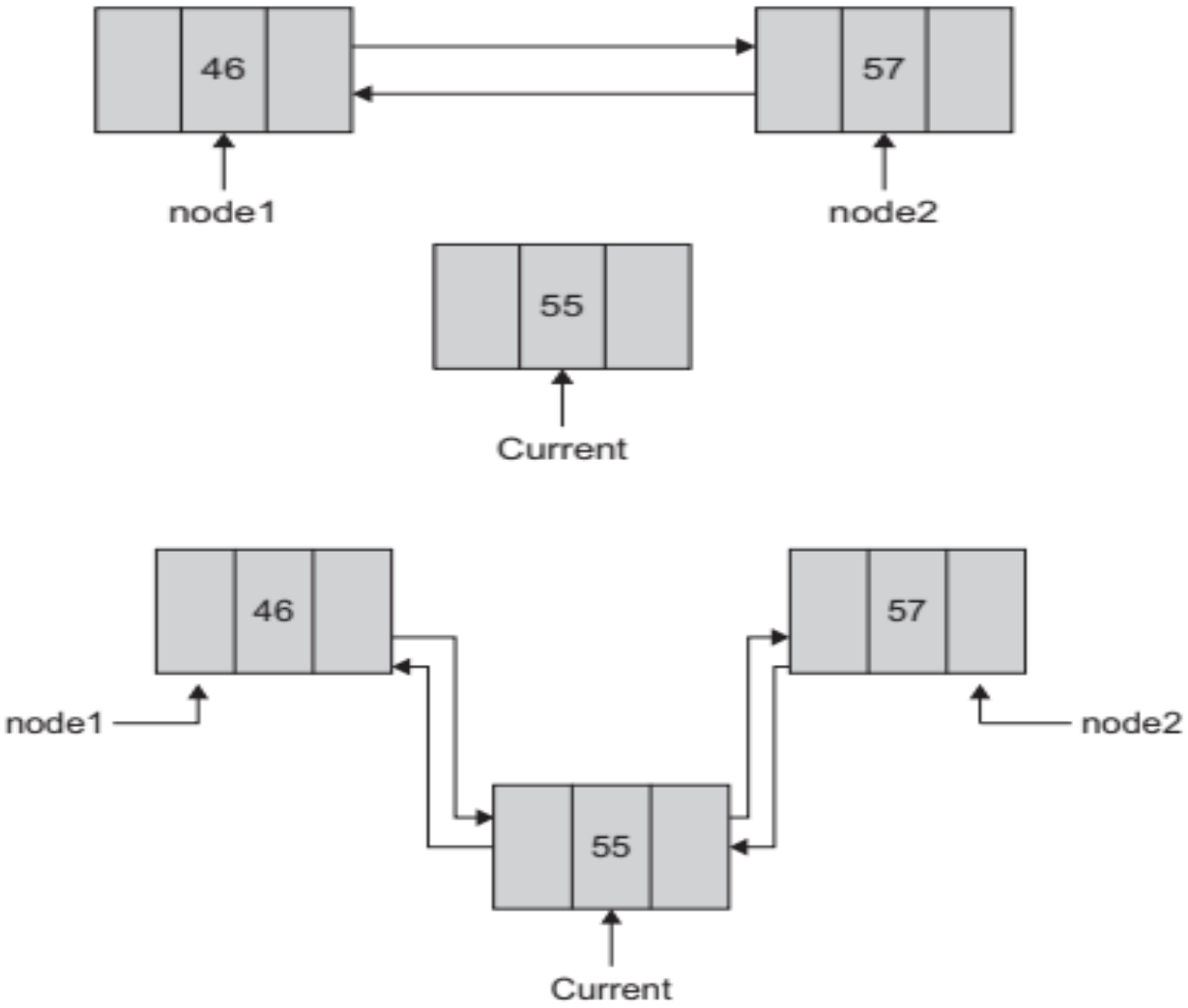
  node1->Next = Current;

  node2->Prev = Current;

  Current->Next = node2;

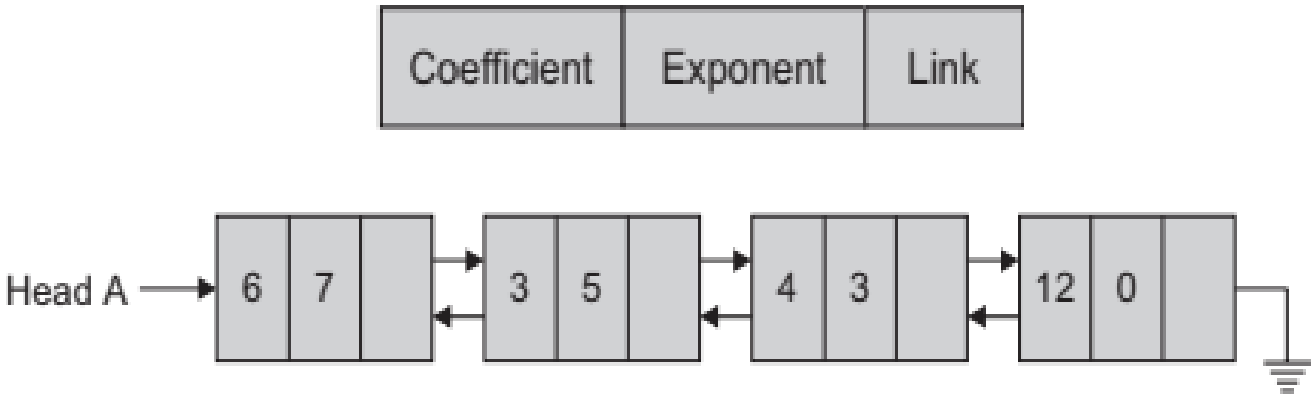
  Current->Prev = node1;

Insertion of a node in a doubly Linked List



Polynomial Manipulations

- The polynomial, say $A = 6x^7 + 3x^5 + 4x^3 + 12$ would be stored as in Fig.
- A node will have 3 fields, which represent the coefficient and exponent of a term and a pointer to the next term.



Polynomial addition

- Let two polynomials A and B be
- $A = 4x^9 + 3x^6 + 5x^2 + 1$
- $B = 3x^6 + x^2 - 2x$
- The polynomial A and B are to be added to yield the polynomial C.
- The two polynomials A and B are stored in two linked lists with pointers ptr1 and ptr2

Polynomial Addition- Algorithm

1. Create a new linked list, newHead to store the resultant list.

2. Traverse both lists until one of them is null.

3. If any list is null insert the remaining node of another list in the resultant list. Otherwise compare the degree of both nodes, a (first list's node) and b (second list's node). Here three cases are possible:

1) If the degree of a and b is equal, we insert a new node in the resultant list with the coefficient equal to the sum of coefficients of a and b and the same degree.

2) If the degree of a is greater than b, we insert a new node in the resultant list with the coefficient and degree equal to that of a.

3) If the degree of b is greater than a, we insert a new node in the resultant list with the coefficient and degree equal to that of b.

Polynomial Multiplication

1. Let A and B be two polynomials.
2. Let the number of terms in A be M and number of terms in B be N.
3. Let C be the resultant polynomial to be computed as C = A X B
4. Let us denote the i^{th} term of the polynomial Bas tB_i . For each term tB of the polynomial B, repeat steps 5 to 7 where i = 1 to N.
5. Let us denote the j^{th} term of the polynomial A as tAj. For each term of tAj of the polynomial A, repeat steps 6 to 7 where j =1 to M. 6. Multiply tAj and tBi . Let the new term be tCk = tAj X tBi .
7. Compare tCk with each term of the polynomial C. If a term with equal exponent is found, then add the new term tCk to that term of the polynomial C, else search for the appropriate position for the term tCk and insert the same in the polynomial C.
8. Stop.

Generalized linked list

- *Generalized lists* are defined recursively as lists whose members may be single data elements or other generalized lists.
- In other words, a generalized list is a finite sequence of $n \geq 0$ elements, a_1, a_2,... a_n, which we write as list $A = (a_1, a_2, ..., a_n)$, where a_i is either an atom or the list.

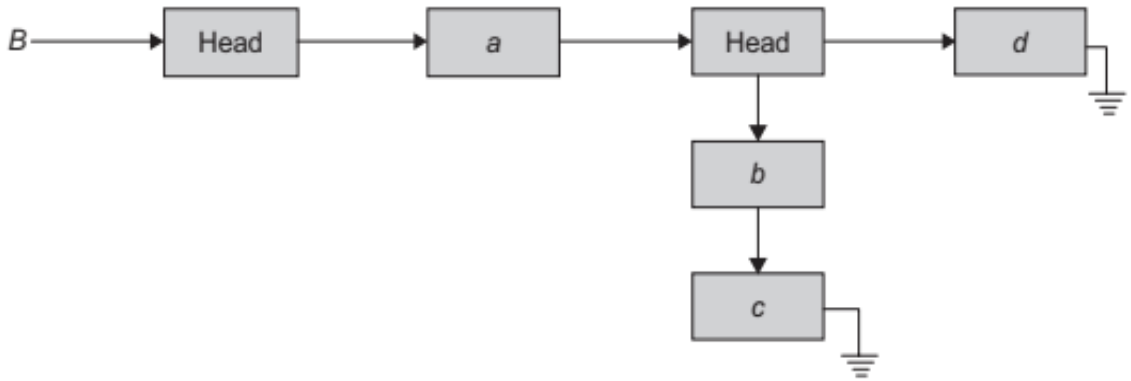
Generalized linked list

- Some examples of generalized lists are the following:

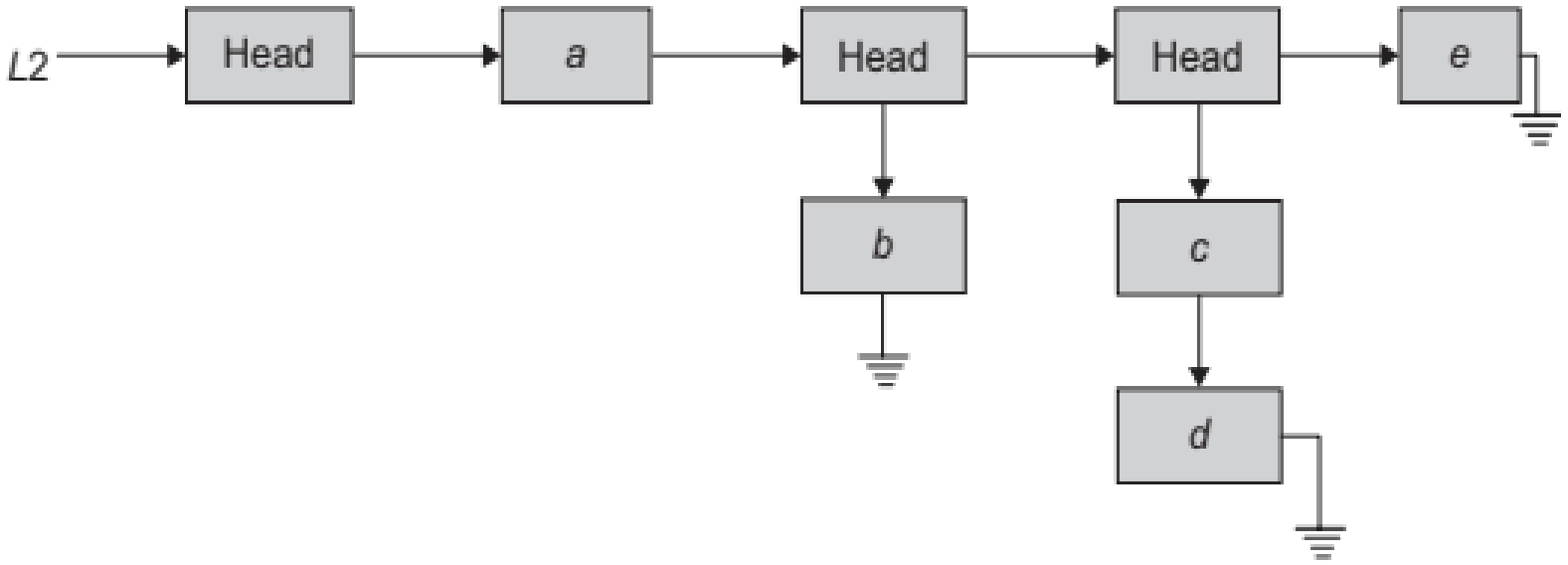
1. $A = ()$	The empty (or null) list.
2. $B = (a, (b, c), d)$	List of three elements—the first element is *a*, the second element is list (*b*, *c*), and the third element is *d*.
3. $C = (B, B, A)$	List of length 3 with the first and the second element as list *B* and the third element as list *A*, which is a null list.
4. $D = (a, b, D)$	List of length 3 which is recursive as it includes itself as one of the elements. It can also be written as $D = (a, b, (a, b, (a, b, ...)...$

Generalized linked list

- One of the better approaches to visualize the generalized lists is using a header node.
- In this approach, each generalized list has a header node labelled Head. Figure shows the pictorial representation of list *B*.



- $L2 = (a, (b), (c,d), e)$



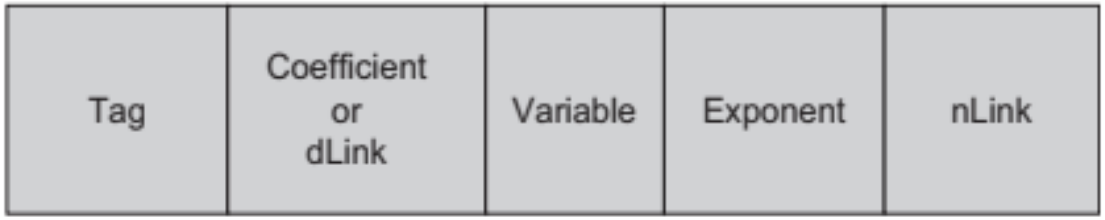
Application of Generalized linked list

- Consider the following polynomial P with three variables x ,y, and z. Consider the two-variable polynomial Q of x and y.
- $Q(x, y) = 5x^4y^3 + 6x^6y^5 + 3x^5y^2 + xy$
- Now, similar to a single variable polynomial, we can represent this polynomial Q(x,y) as a sequential organization with four fields: coefficient, Exp_X, Exp_Y, and nLink as in Fig.

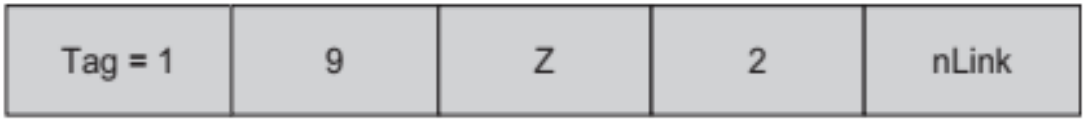
Coefficient	Exp_X	Exp_Y	nLink

Polynomial representation

- We can write such a polynomial as one with a single variable whose each term node would be as in Fig



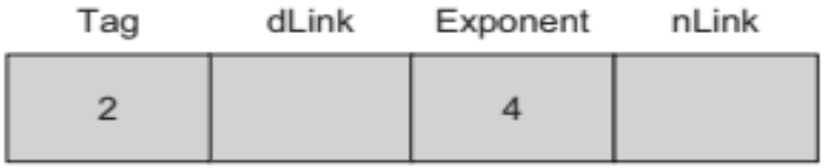
- For example, the term as $9z^2$ would be represented as



- Each node would be one of the three—the header node, data node with constant coefficient, and the data node whose coefficient is a polynomial. These can be pictorially viewed as follows

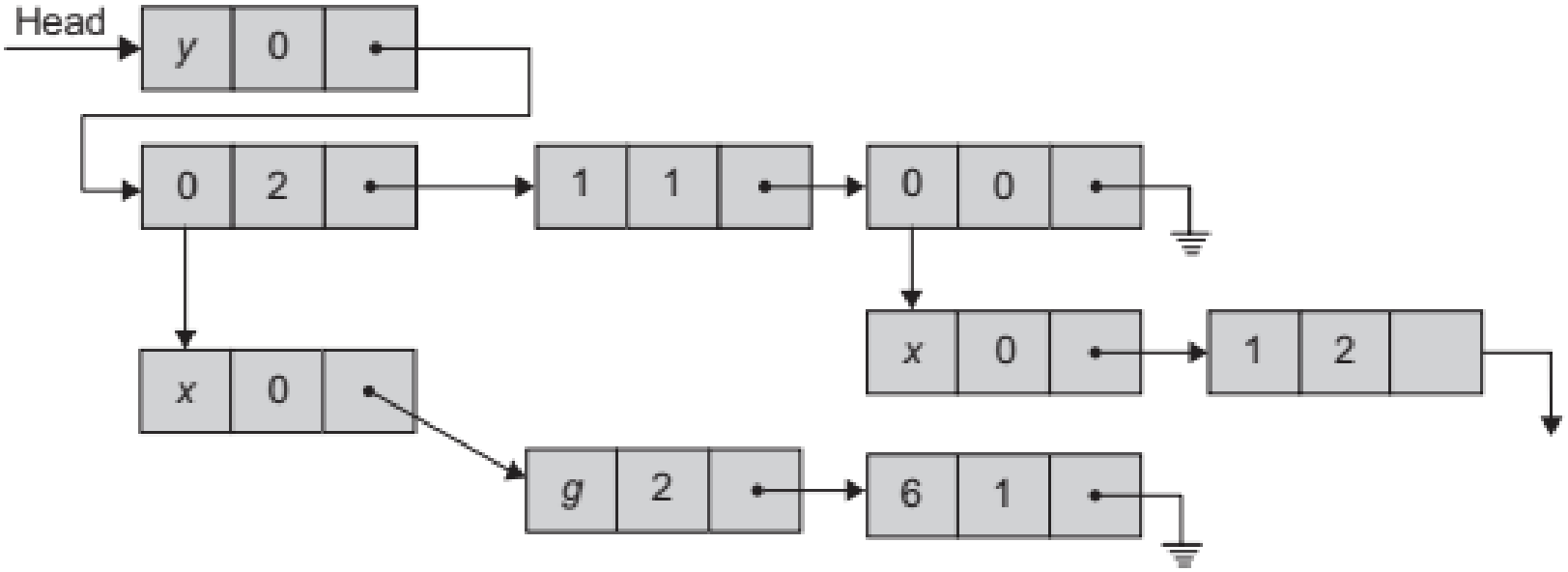
Representation of header node

Representation of data node with constant coefficient



Representation of data node with polynomial coefficient

- The GLL for $9x^2y^2 + 6xy^2 + y + x^2$

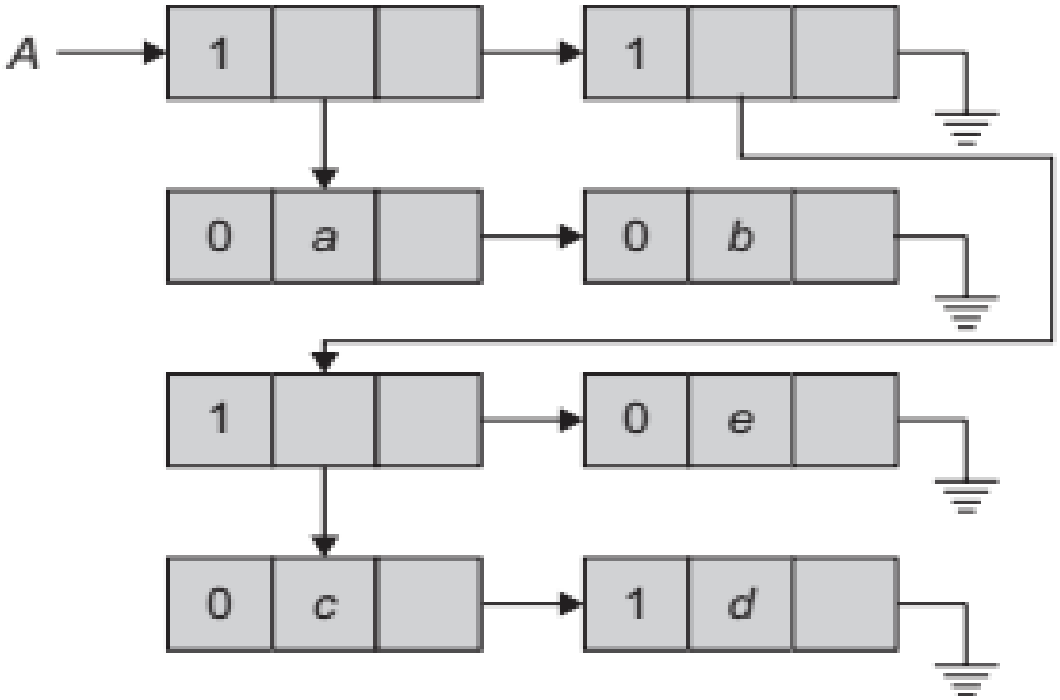


Representation of sets Using GLL

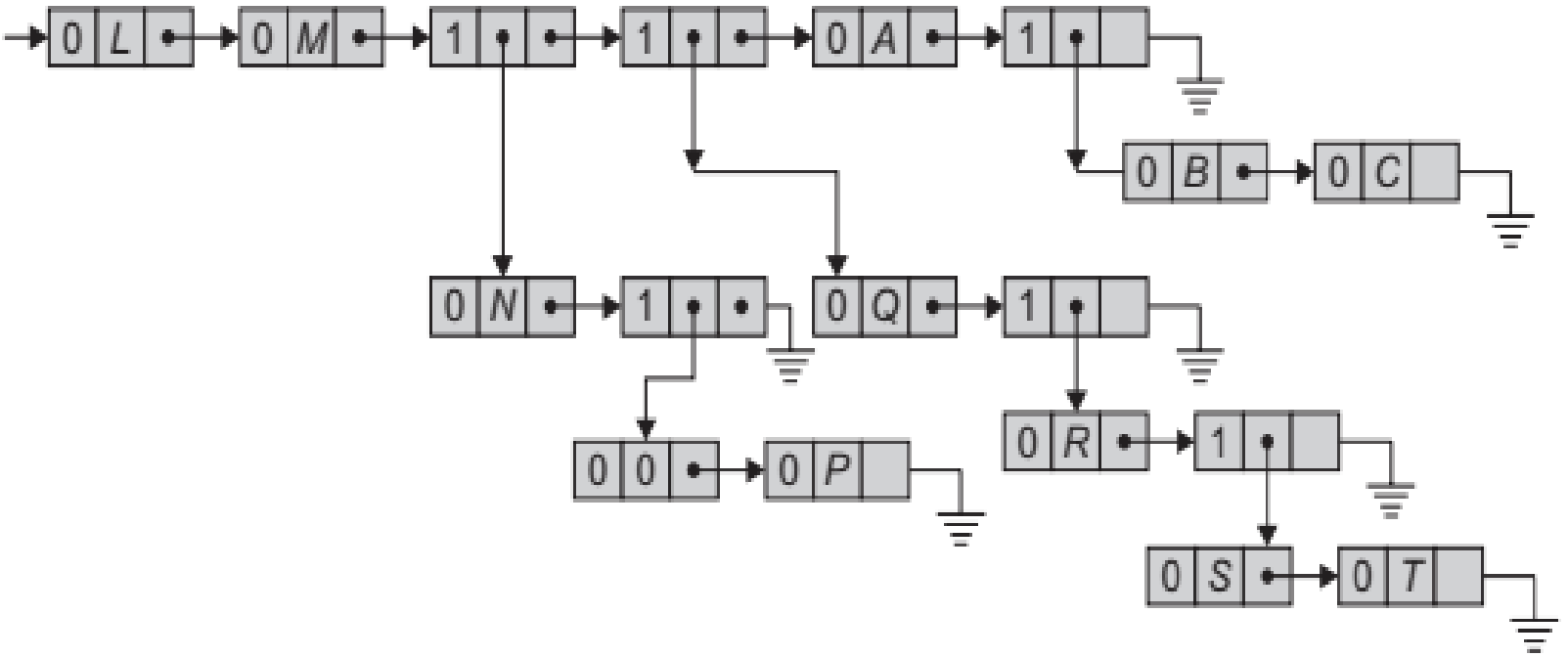
- Let A be a set, $A = \{a, b, \{c, d, \{\ \}\}, \{e, f\}, g\}$. Here, A consists of elements that are either atoms or sets. Hence, we need a GLL node to convey whether the member of set is an atom or a set.
- The tag field is set to 0 if the member is an atom and is set to 1 if it is another list.

Representation of sets Using GLL

- GLL representation for the set
- $A = \{\{a, b\}, \{\{c, d\}, e\}\}$



$X = \{L, M, \{N, \{O, P\}\}, \{Q, \{R, \{S, T\}\}, A, \{B, C\}\}$



Application of linked list—garbage collection

- It is a dynamic technique for memory management and heap allocation that examines and identifies dead memory blocks before reallocating storage for reuse.
- Garbage collection's primary goal is to reduce memory leaks.
- Garbage collection frees the programmer from having to deallocate and return objects to the memory system manually.
- Garbage collection can account for a considerable amount of a program's total processing time, and as a result, can have a significant impact on performance.
- Stack allocation, region inference, memory ownership, and combinations of various techniques are examples of related techniques.

Application of linked list—garbage collection

- The basic principles of garbage collection are finding data objects in a program that cannot be accessed in the future and reclaiming the resources used by those objects.
- Garbage collection does not often handle resources other than memory, such as network sockets, database handles, user interaction windows, files, and device descriptors.
- Methods for managing such resources, especially destructors, may be sufficient to manage memory without the requirement for GC.
- Other resources can be associated with a memory sector in some GC systems, which, when collected, causes the task of reclaiming these resources.

Thank You