Unit III

LINKED LIST

Presented By

Dr.Nandkishor Karlekar

Contents

Introduction, Representation of Linked List,
Linked List v/s Array,
Types of Linked List - Singly Linked List,
Circular Linked List, Doubly Linked List,
Operations on Singly Linked List and Doubly Linked List,
Stack and Queue using Singly Linked List,
Singly Linked List
Application-Polynomial Representation and Addition

List

- The term “list” refers to a linear collection of data items such that there is a first element, second and …. A last element. Data processing frequently involves storing and processing data organized into lists.
- Violet, Blue, Green, Orange, Red.
- One way to store such lists is by means of arrays. Arrays use sequential mapping i.e. the data elements are stored in memory fixed distances apart. This makes it easy to compute the location of any element in the array. However, arrays have certain disadvantages.

Limitations of Arrays

- An array is a static data structure i.e. the size of the array remains fixed. Thus, even if the data structure actually uses less amount of storage to store elements or possibly no storage at all, the unutilized memory cannot be used for other purposes. Moreover, if more than the allotted space is required, it cannot be increased during runtime.
- Most real-time applications process variable size data. Thus sequential mapping proves inadequate due to the reasons mentioned in 1.
- Often, we need to insert and delete data. Insertion and deletion from the end is easy but in order to insert an element in between, the remaining elements will have to be shifted by one position. Even for deletion, the elements will have to be shifted one position before. This will require a lot of processing time.

Linked List

- One solution to the above problem is that instead of using sequential representation, a linked representation should be used. i.e. unlike an array where elements are stored sequentially in memory, items in a list may be located anywhere in memory. To access elements in the correct order, we store the address or locations of the next element, with each element of the list.
- **Definition :** A linked list is an ordered collection of data elements where the order is given by means of links i.e. each item is connected or linked to another item.
- Basically a linked list consists of “nodes”. Each node contains an item field and a link. The item field may contain a data item or a link.

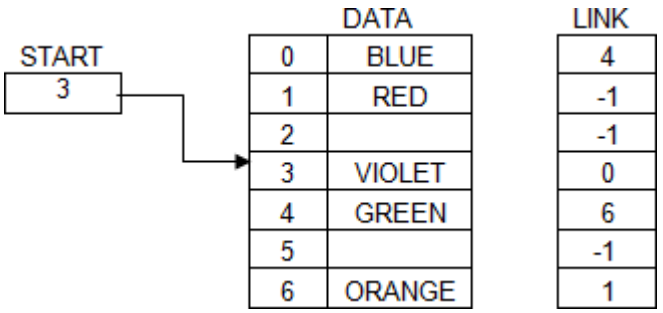
Linked list may be implemented in two ways:

- **Static Representation :**
  - An array is used to store the elements of the list. The elements may not be stored in a sequential order. The correct order can be stored in another array called “Link”. The values in this array are pointers to elements in the data array.

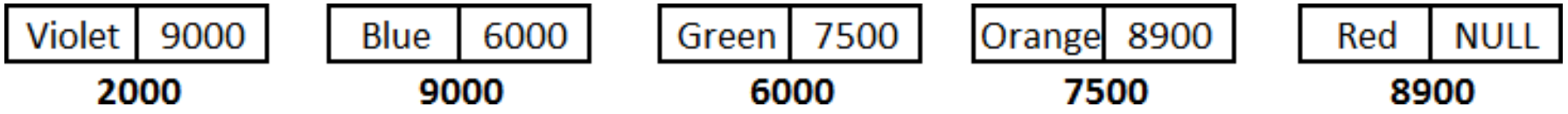
- **Dynamic representation**
  - The static representation uses arrays, which is a static data structure and has its own limitations.
  - Another way of storing a list in memory is by dynamically allocating memory for each node and linking them by means of pointers since each node will be at random memory locations. We will need a pointer to store the address of the first node

Example

- Static representation



- Dynamic representation



Applications

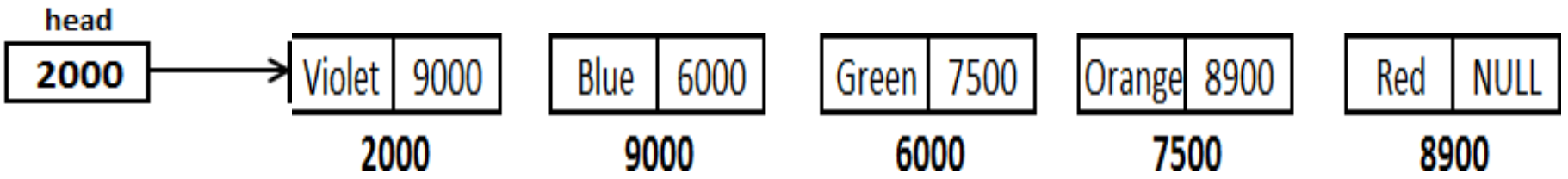
- Hash tables use linked lists for collision resolution
- Any "File Requester" dialog uses a linked list
- Binary Trees
- Stacks and Queues can be implemented with a doubly linked list
- Relational Databases (e.g. Microsoft Access)

Types of Linked List

- Singly Linked List (SLL)
- Doubly Linked List (DLL)
- Circular Linked List (CLL)
  - Circular Singly Linked List (CSLL)
  - Circular Doubly Linked List (CDLL)
- Generalized Linked List (GLL)

Singly Linked List

- Each node consists of only one link, to point to the next node or element.
- **Each node of the list has two elements:**
  - the item being stored in the list *and*
  - a pointer to the next item in the list

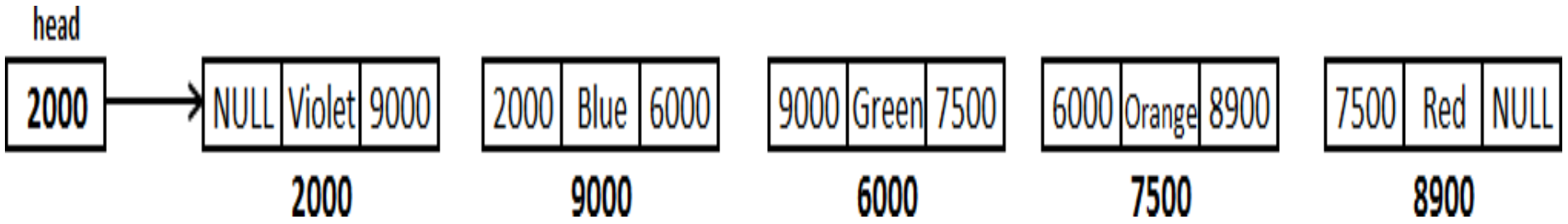


Node Structure for SLL

```
struct SLL
{
        int data;
        struct SLL *next;
};
typedef struct SLL sll;
sll *head = NULL;
```

Doubly Linked List

- Each node consists of two pointer links, one to the next node, and one to the previous node.
- **Each node of the list has two elements:**
  - a pointer to the previous item in the list
  - the item being stored in the list *and*
  - a pointer to the next item in the list



Node Structure for DLL

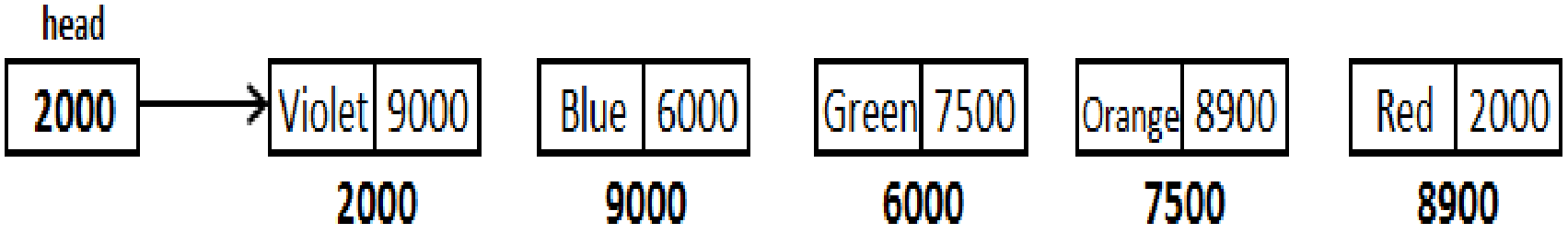
```
struct DLL
{
        struct DLL *prev;
        int data;
        struct DLL *next;
};
typedef struct DLL dll;
dll *head = NULL;
```

Circular link list

- Circular lists are like singly linked lists, except that the last node contains a pointer back to the first node rather than the null pointer. From any point in such a list, it is possible to reach any other point in the list. If we begin at a given node and travel the entire list, we ultimately end up at the starting point.
- Note that a circular list does not have a natural "first or "last" node. We must therefore, establish a first and last node by convention - let external pointer point to the last node, and the following node be the first node.

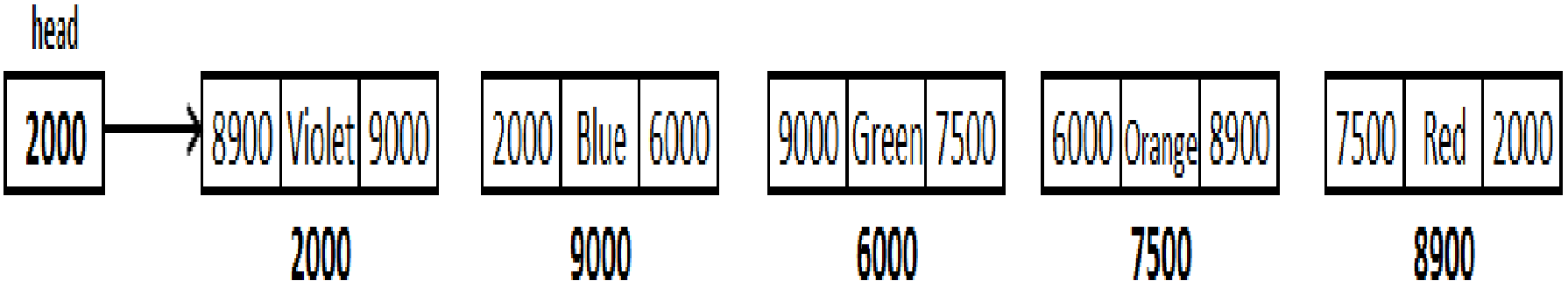
Circular Singly Linked List

- In circular doubly linked list the previous pointer of first node and the next pointer of the last node is pointed to head node.



Circular Doubly Linked List

- In circular doubly linked list the previous pointer of first node and the next pointer of the last node is pointed to head node.



Linked List Operations

- Basic operations:
  - Append a node to the end of the list
  - Insert a node into the list
  - Traverse the list
  - Delete a node from the list
  - Delete/destroy the list

Create a New Node

- Allocate memory for the new node:
  - Node = (sll *) malloc(sizeof(sll));
- Initialize the contents of the node:
  - Node->data = X
  - Node->next = NULL;

Appending a Node

- Add a node to the end of the list
- Basic process:
  - Create a new node (as already described)
  - Add the node to the end of the list:
  - If the list is empty, set head pointer to this node
  - Else,
  - Traverse the list to the end
  - Set the pointer of the last node to point to the new node

Inserting a Node Into a Linked List

- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - A pointer to locate the node with a data value greater than that of the node to be inserted
  - A pointer to ‘trail behind’ one node, to point to the node before the point of insertion
- A new node is inserted between the nodes pointed at by these pointers

Inserting a Node Into a Linked List

- 3 cases
  - Insert at start
  - Insert in between
  - Insert at end

Traversing a Linked List

- Visit each node in a linked list and display the
- data in each node
- Basic process:
  - Set a pointer to the contents of the head pointer
  - While the pointer is not NULL
    - Process the data
    - Go to the next node by setting the pointer to the pointer field of the current node in the list
  - End while

Deleting a Node

- Used to remove a node from a linked list
- If the list uses dynamic memory, then delete the node from memory
- Change the address field of previous node to point to the next node of the node to be deleted.
- Requires two pointers:
  - One to locate the node to be deleted
  - One to point to the node before the node to be deleted

Deleting a Node

- 3 Cases
  - Delete the first node
  - Delete the last node
  - Delete in between node

Destroying a Linked List

- Must remove all nodes used in the list
- To do this, use the list traversal to visit each Node
- For each node:
  - Unlink the node from the list
  - If the list uses dynamic memory, then free the node's memory
- Set the list head to NULL

Other Operations

- Display the link list in reverse order
- Revert the link list
- Merge two link list
- Split the link list into two halves
- Count the no. of nodes in a link list
- Sort the link

Linked list as an ADT

```
template < class T >
class LinkedList
{ private:
class ListNode
{ public:
T value; // Node value
ListNode *next; // Next Node
ListNode ( T v, ListNode *n = NULL) : value( v ), next( n ) { }
}; // ListNode
ListNode *head;
public:
LinkedList ( ListNode *ptr = NULL ) { head = ptr; }
~LinkedList ( void );
void appendNode ( T );
void insertNode ( T );
void deleteNode ( T );
void displayList ( void );
}; // LinkedList
```

Link list as an ADT

Data Object :

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

Methods :

Create()
Destroy()
Add(element)
Remove(element)
Traverse()
IsEmpty()
Search(element)

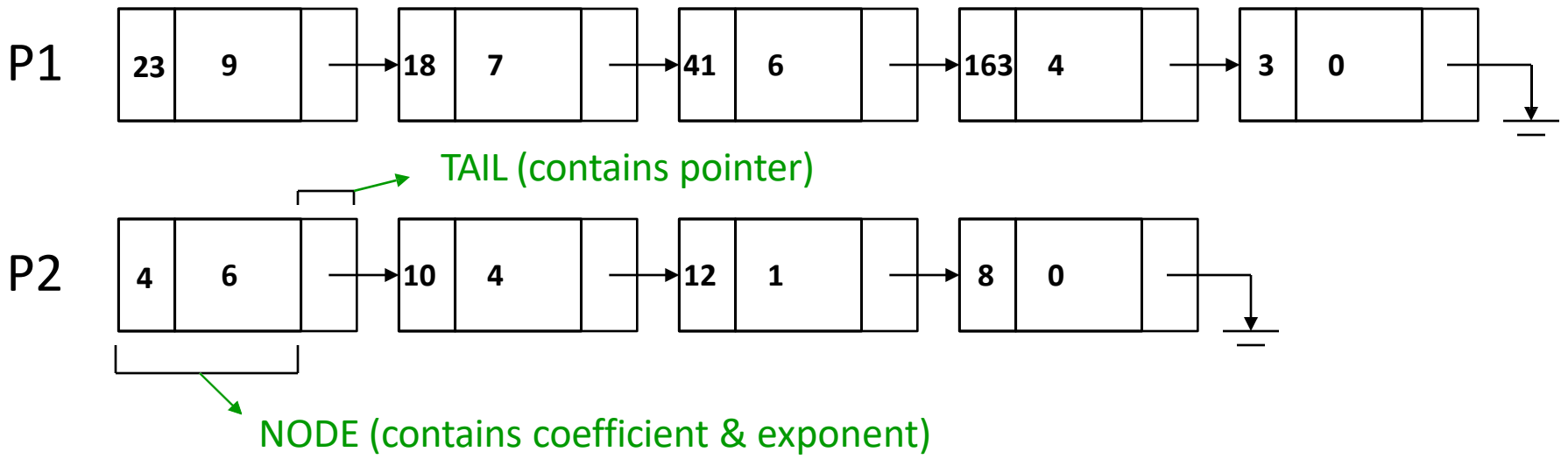
Applications of Linked Lists

- Stacks and Queues Implemented with Linked Lists
- Polynomials Implemented with Linked Lists
  - Remember the array based implementation?
    - Hint: two strategies, one efficient in terms of space, one in terms of running time

Representation of polynomials using linked lists

- Linked list Implementation:

- $p1(x) = 23x^9 + 18x^7 + 41x^6 + 163x^4 + 3$
- $p2(x) = 4x^6 + 10x^4 + 12x + 8$



Representation and manipulations of polynomials using linked lists

- Advantages of using a Linked list:
  - save 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 of using a Linked list :
  - can't go backwards through the list
  - can't jump to the beginning of the list from the end.

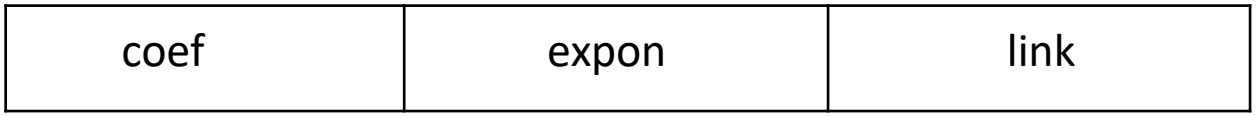
Polynomials

$$A(x) = a_{m-1} x^{e_{m-1}} + a_{m-2} x^{e_{m-2}} + \ldots + a_0 x^{e_0}$$

Representation

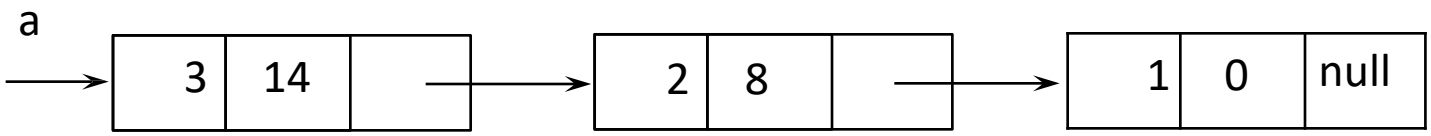
```
struct poly_node
{
        int coef;
        int expon;
        struct poly_node *next;
};

typedef struct poly_node mypoly;
```

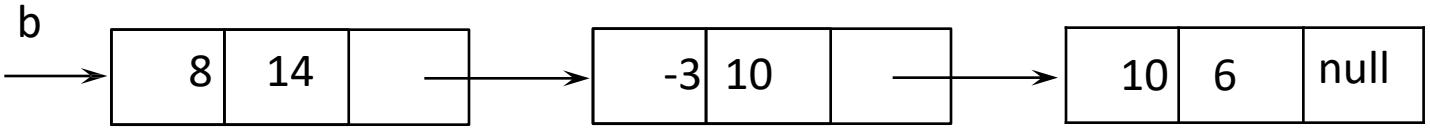


Example

$$a = 3x^{14} + 2x^{8} + 1$$



$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



Operations on Polynomial (Addition)

- Adding polynomials using a Linked list representation: (storing the result in p3)

To do this, we have to break the process down to cases:

- Case 1: exponent of p1 > exponent of p2
- Copy node of p1 to end of p3.

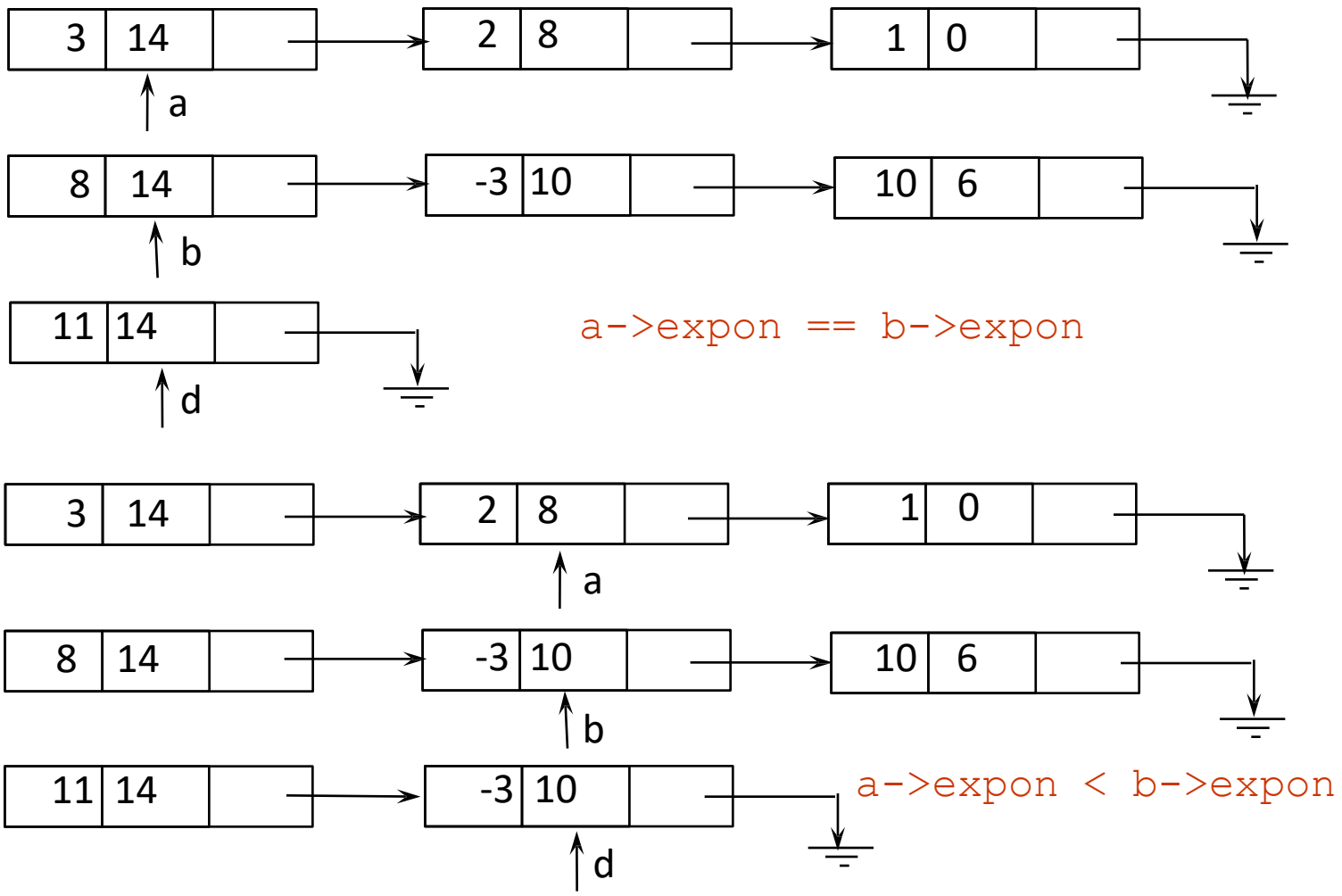
[go to next node]

- Case 2: exponent of p1 < exponent of p2
- Copy node of p2 to end of p3.

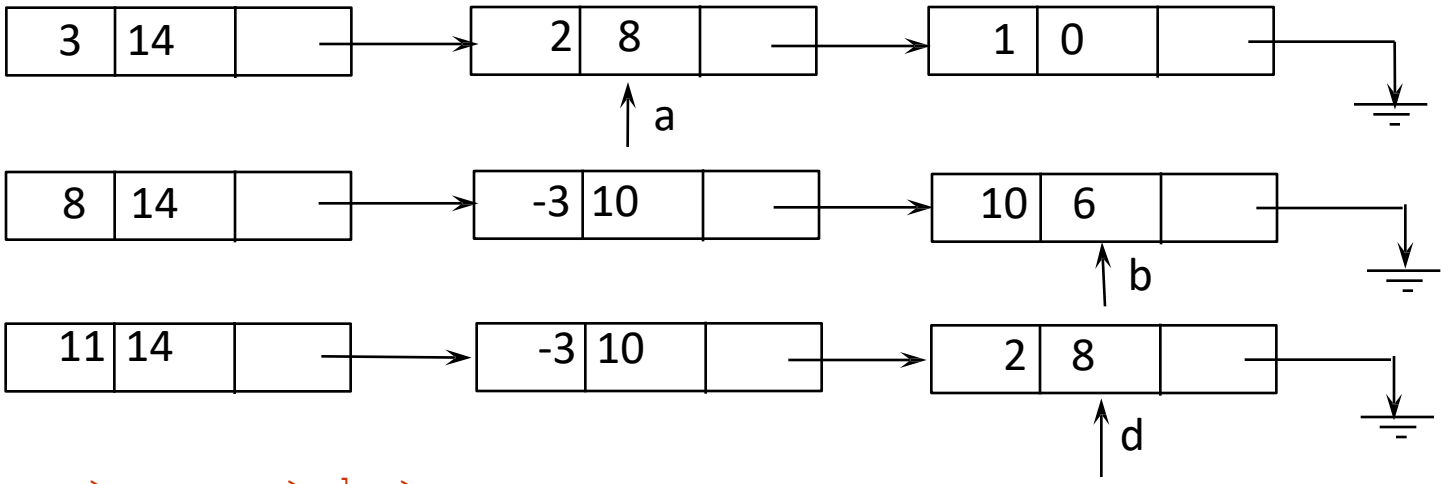
[go to next node]

- Case 3: exponent of p1 = exponent of p2
  - Create a new node in p3 with the same exponent and with the sum of the coefficients of p1 and p2.

Adding Polynomials



Adding Polynomials (cont'd)




```
a->expon > b->expon
```

Comparison of a Sequential and Linked Memory organization

Sequential Organization	Linked Organization
Successive elements of list are stored at fixed distance apart.	Elements can be placed anywhere in the memory.
Provides static allocation of memory, which means space allocation, is done by compiler once cannot be changed during execution and size has to be known in advance.	Provides dynamic allocation of memory, which means space allocation, is done during execution and size need not be known in advance.
As individual objects are stored at fixed distance apart, we can access any element randomly.	As individual objects are not stored at fixed distance apart, we cannot access element randomly.
Insertion and deletion of objects in between the list requires a lot of data movement.	Insertion and deletion of objects in between the list do not require any data movement.
Space inefficient for large objects and large quantity	Space efficient for large objects and large quantity
Element need not keep address of its successive element	Every element keeps address of its successive element.
No additional space is required as elements are stored in sequence.	The only overhead is we need additional space for the link field of each element. But still additional space for the link field is not a penalty when large objects are to be stored.
Sequential organizations can also be allocated during runtime by use of pointers	Linked organization needs use of pointers and dynamic memory allocation.

Generalized list

- List : collection of elements.
- A *generalized list, A, is a finite sequence of n≥0* elements, α_0, ..., α_{n-1}, where α_i is either an atom or a list. The element α_i, $0 \leq i \leq n-1$, that are not atoms are said to be the sublists of A.

  The list A is written as $A=(\alpha_0, \alpha_1, ..., \alpha_{n-1})$.
  Where A is the name of the list.
  n is the length of A.
  α_0 is the head of A.
  $(\alpha_1, ..., \alpha_{n-1})$ is the tail of A.
  Lists may be shared by other lists.
  Lists may be recursive.

Examples of generalized lists

- D=()
  - the null or empty list; its length is zero.
- A=(a,(b,c))
  - a list of length two; its first element is the atom a , and its second element is the linear list (b,c).
- B=(A,A,())
  - a list of length three whose first two elements are the list A, and the third element is the null list.
- C=(c,C)
  - a recursive list of length two.
  - C=(a,(a,(....).

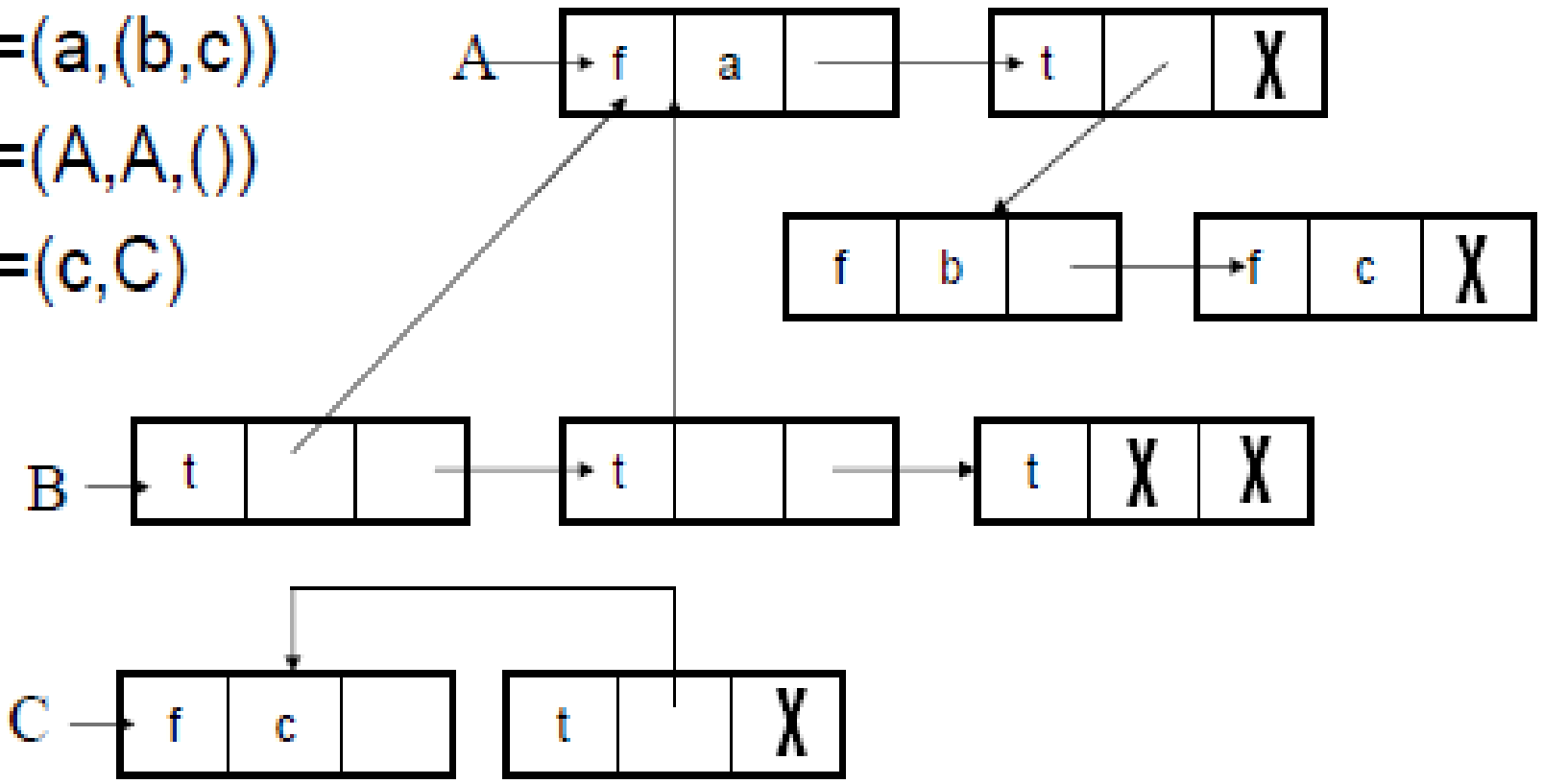
GLL Example

D=()

A=(a,(b,c))

B=(A,A,())

C=(c,C)



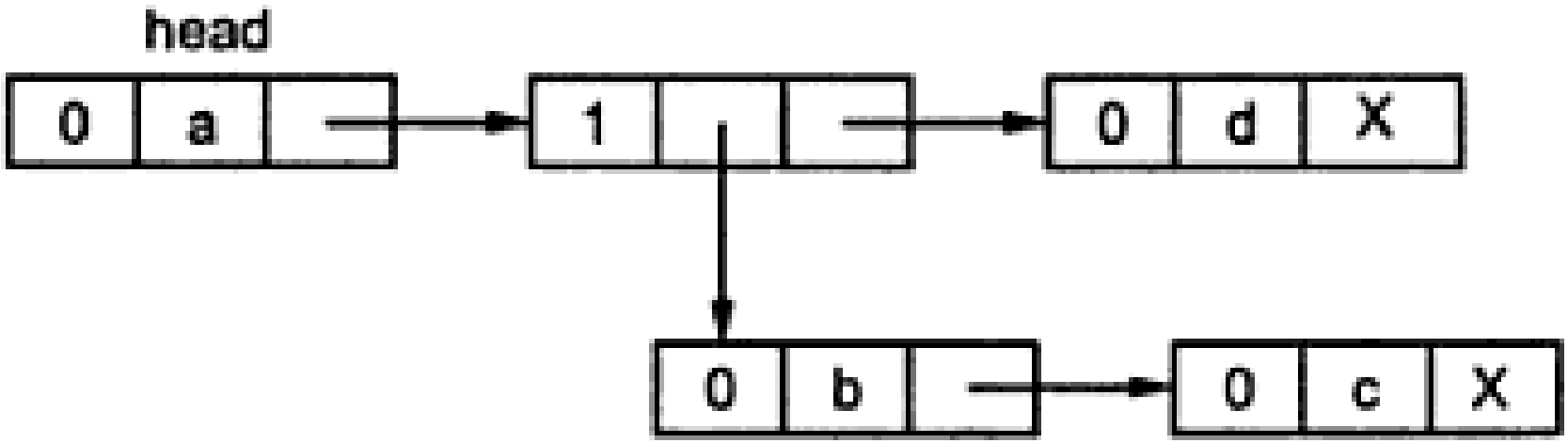
Node structure for generalized list

Flag field	Data Field / Pointer to sublist	Pointer to next node

```
struct GLL
{
      int  tag;
      union data_dlink
      {
            char data;
             struct GLL *down;
      }u1;
       struct GLL *next;
};
typedef struct GLL gll;
gll *head;
```

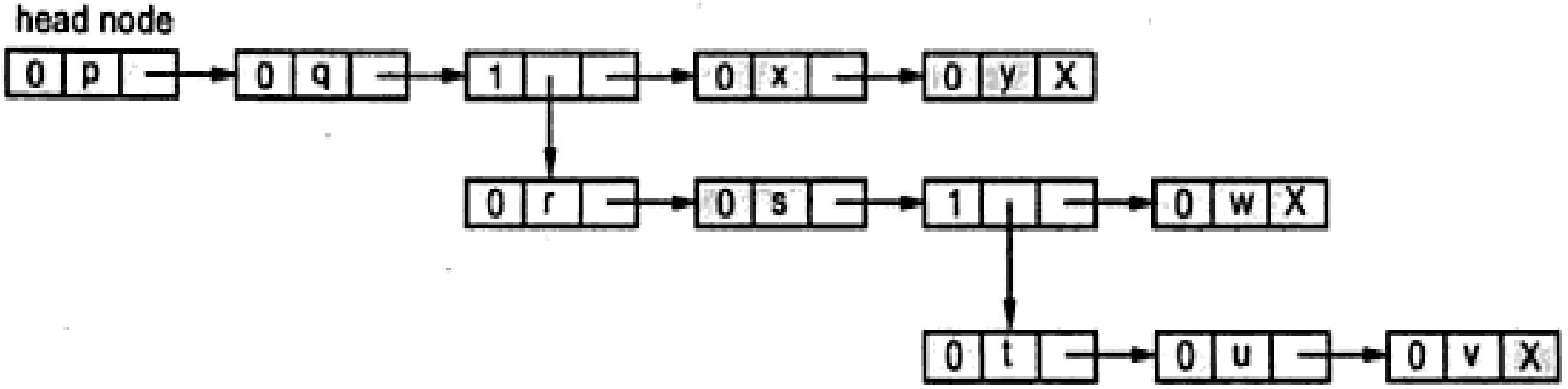
Example

G = (a , (b , c), d)



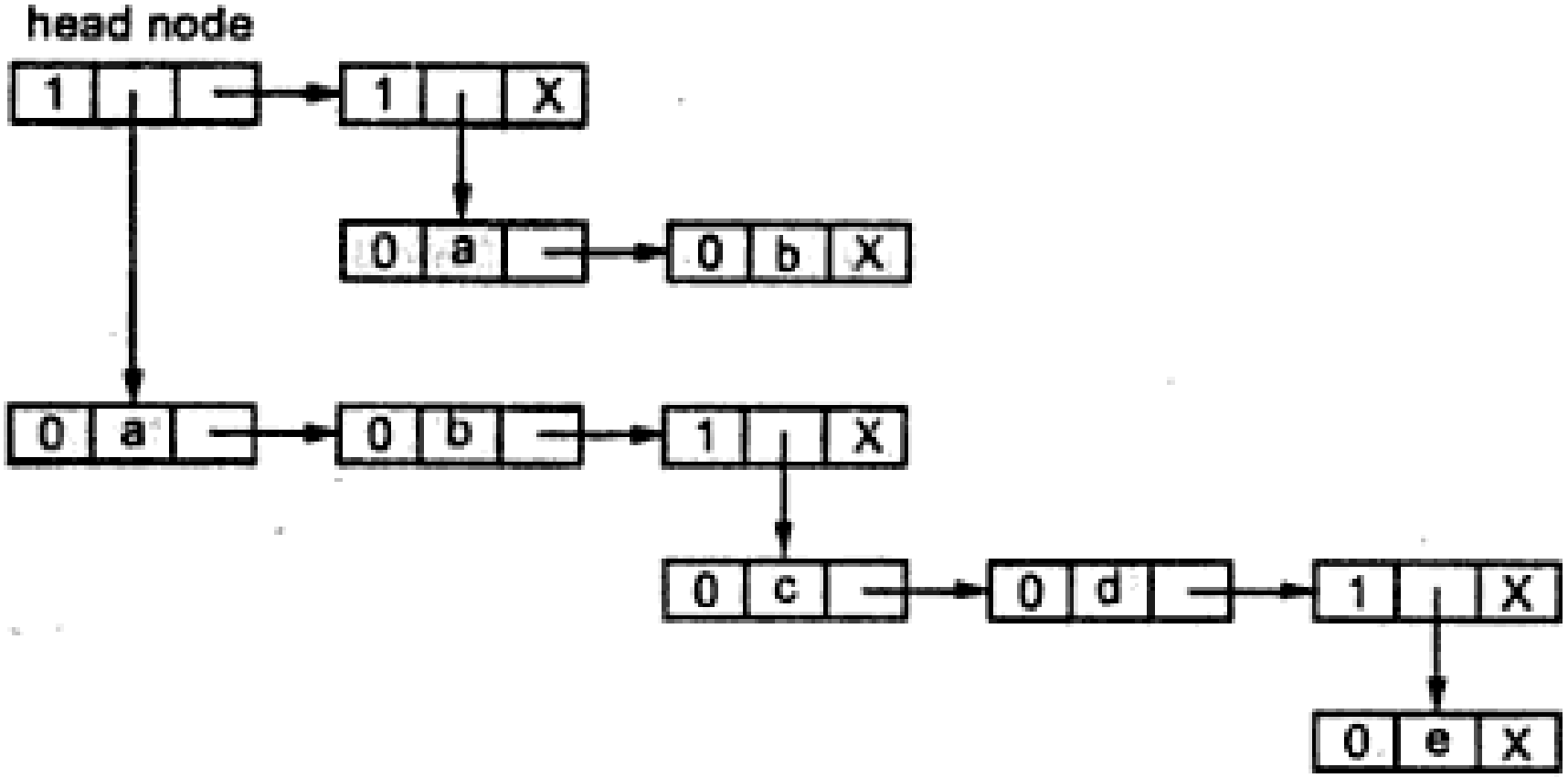
Example

G = (p , q, (r, s, (t, u, v), w), x, y)



Example

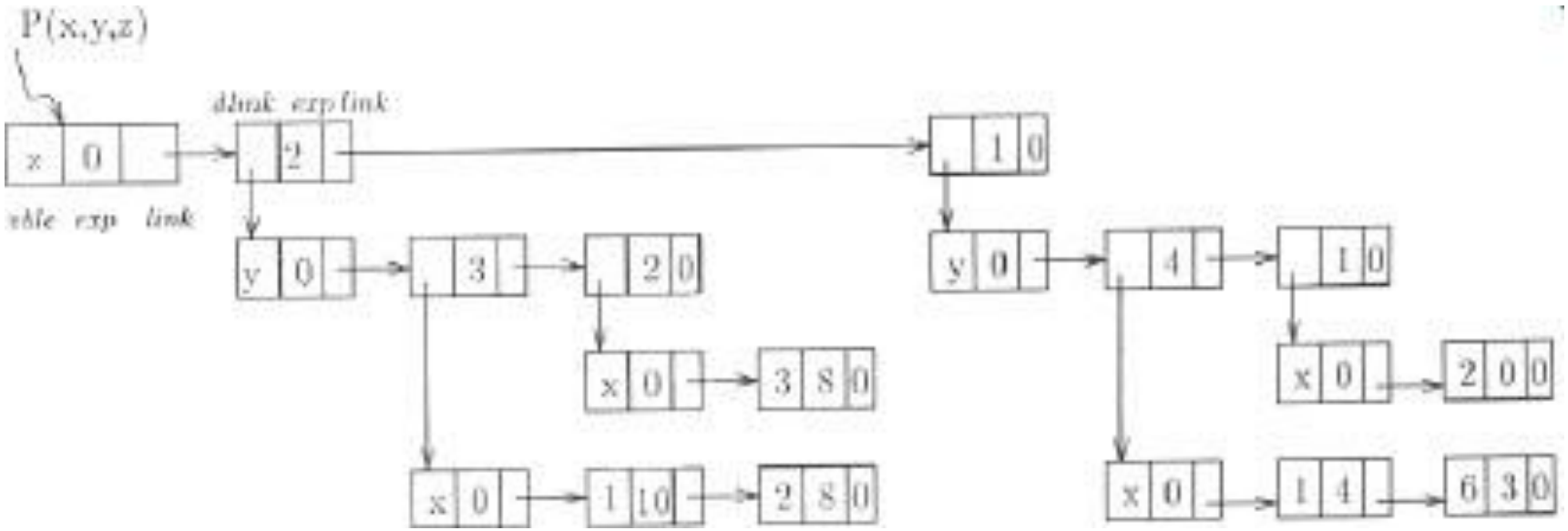
G = ((a, b, (c, d, (e))), (a, b))



Representation of polynomials using Generalized list

$$x^{10}y^3z^2+2x^8y^3z^2+3x^8y^2z^2+x^4y^4z+6x^3y^4z+2yz$$

$$((x^{10}+2x^8)y^3+3x^8y^2)z^2+((x^4+6x^3)y^4+2y)z$$



Polynomial representation using GLL

- Three types of nodes
- Type 1 :

Tag field	Variable field	Pointer to next node / next pointer

- Type 2

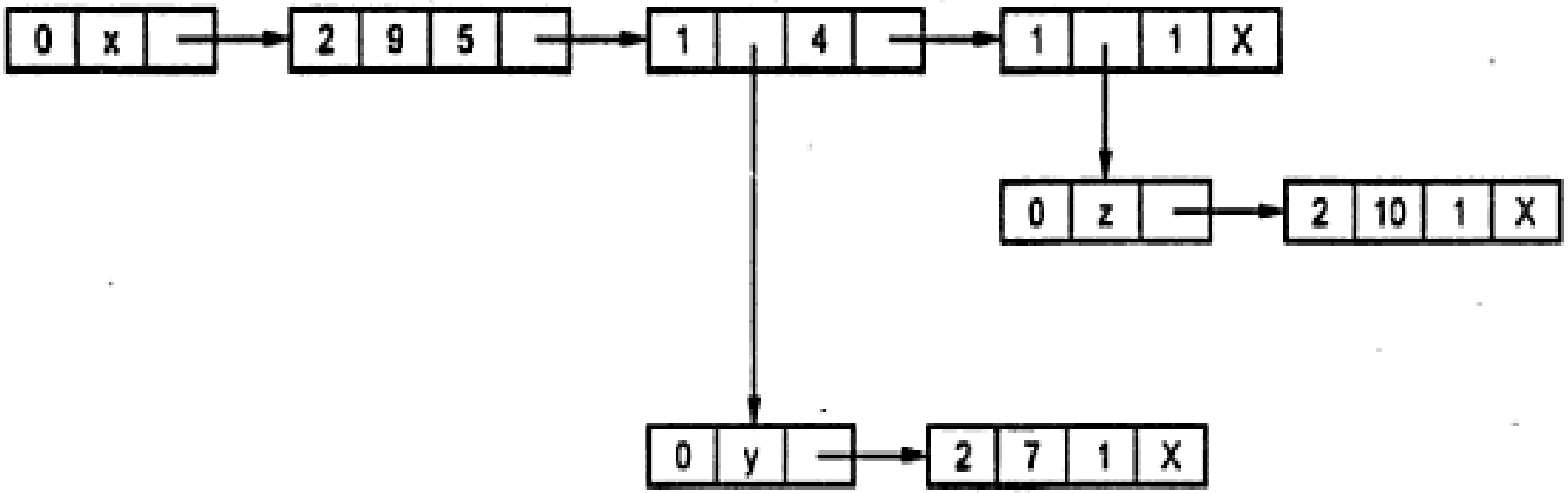
Tag field	Pointer to sub polynomial / dlink	Exponent field	Pointer to next node / next Pointer

- Type 3

Tag field	Coefficient field	Exponent field	Pointer to next node / next Pointer

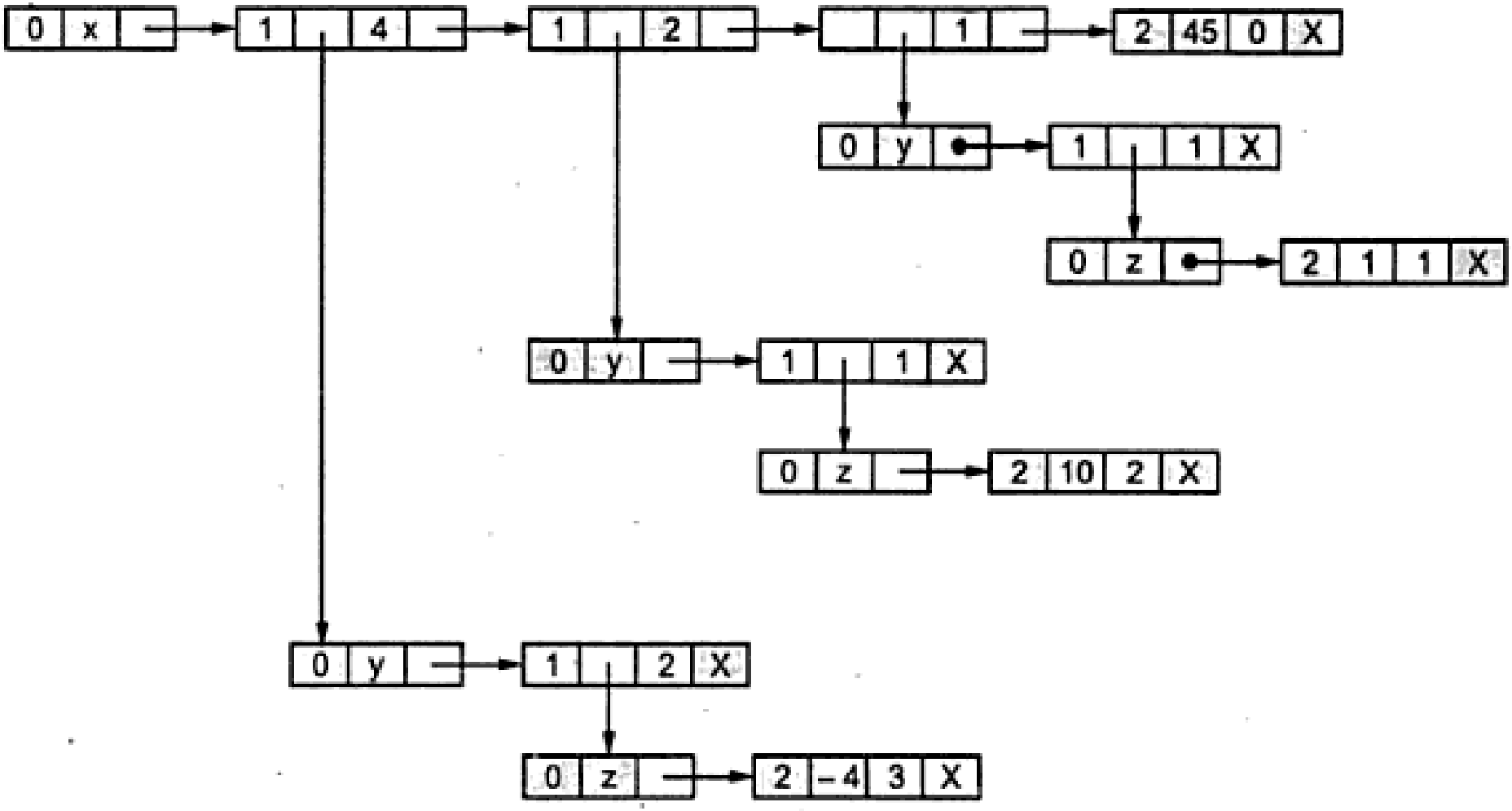
Example 1

$$9x^5 + 7xy^4 + 10xz$$



Example 2

$$-4x^4y^2z^3+10x^2yz^2+7xyz+45$$



Node Structure

```
struct GLL
{
    int  tag;
    union data_coefexp
    {
        char data;
        struct coef_exp
        {   union coef_dlink
            {
                int coef;
                struct GLL *down;
            }u2;
            int exp;
        } s1;
    }u1;
    struct GLL *next;
};
```