Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

Data Structures

S. Y. B. Tech CSE / AIDS

Semester – III

DEPARTMENT OF COMPUTER ENGINEERING & TECHNOLOGY

Syllabus

UNIT 4

Linked List: Linked List as an Abstract Data Type, Representation of Linked List Using Sequential Organization, Representation of Linked List Using Dynamic Organization, Types of Linked List: singly linked, Circular Linked Lists, Doubly Linked List, Doubly Circular Linked List, Primitive Operations on Linked List,

Polynomial Manipulations-Polynomial addition.

Generalized Linked List (GLL) concept, Representation of Polynomial using GLL.

Case Study : Garbage Collection

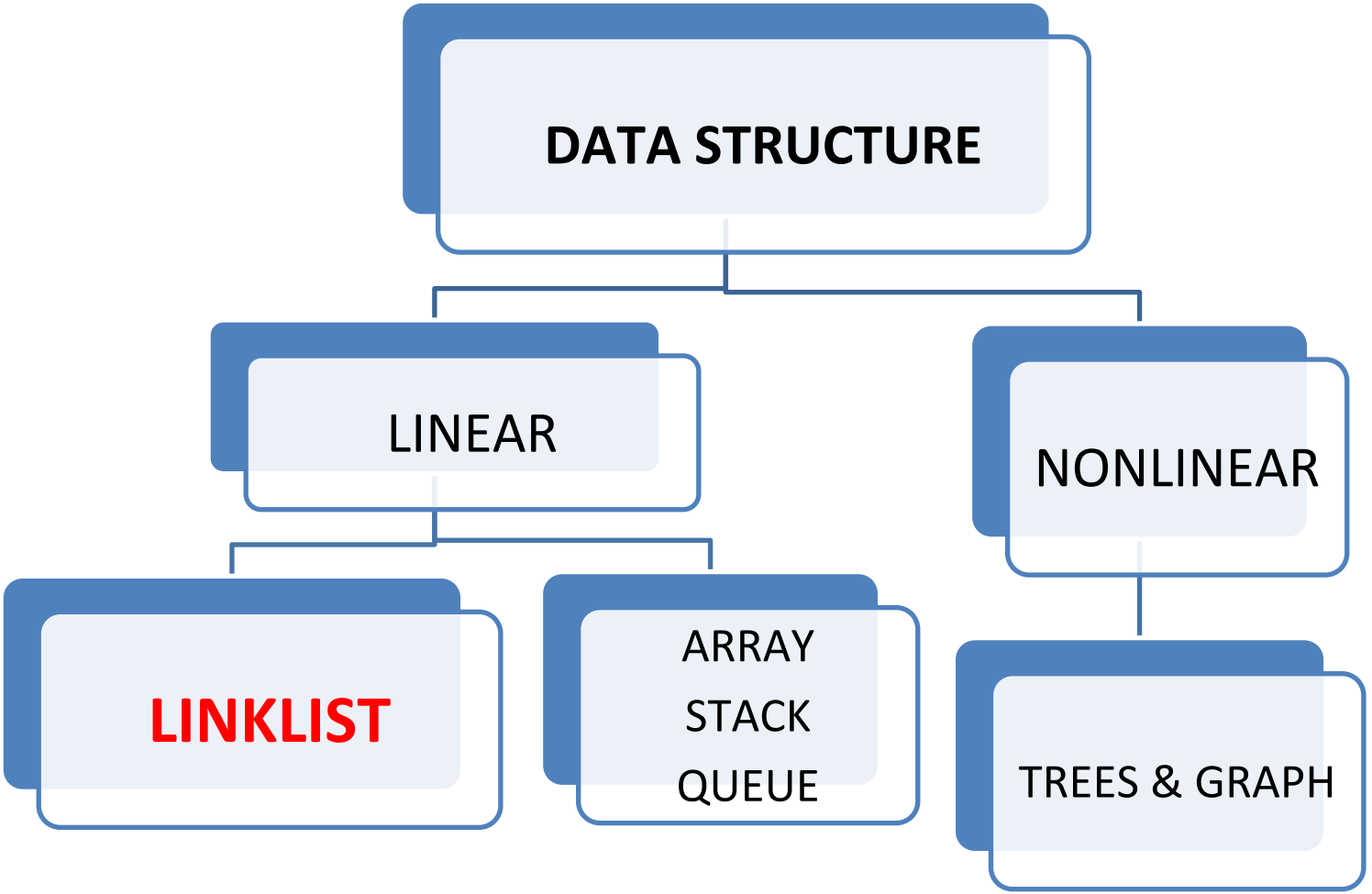
Linked List

Topics to be Covered

- Linked List as an ADT
- Representation of Linked List Using Sequential Organization
- Representation of Linked List Using Dynamic Organization
- Operations on Linked List
- Polynomial operations using linked list
- Circular Linked List, Doubly Linked List , Generalized Linked List (GLL)
- Case Study : Garbage Collection

MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Types of Data Structures



Linked List ADT

structure Linked List (item)

declare CREATE() -> Linked list

insert(item, linked list) -> linked list

delete(linked list) -> linked list

ISEMPS(linked list) -> boolean;

for all L ∈ Linked list, i ∈ item let

ISEMPS(CREATE) ::= true

ISEMPS(insert(i,L)) ::= false

delete(CREATE) ::= error

delete(insert(I,L)) ::= L

end Linked List

Introduction to linked list

- Representation of simple data structures using an array and a sequential mapping are studied.
- These representations had the property that successive nodes of the data object were stored fixed distance apart.
- If the element a_{ij} of a table was stored at location L_{ij}, then $a_{i, j+1}$ was at the location $L_{ij + c}$ for some constant c;

(BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, TAT, VAT, WAT)

- Disadvantage of a sequential mapping for ordered lists
  - insertion and deletion of arbitrary elements become expensive.(Add word GAT will require shifting)

contd…

- Solution to this problem of data movement in *sequential representations is achieved by using linked representations* .

- Unlike a sequential representation where successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory.

- Another way of saying this is that in a sequential representation the order of elements is the same as in the ordered list, while in a linked representation these two sequences need not be the same.

contd…

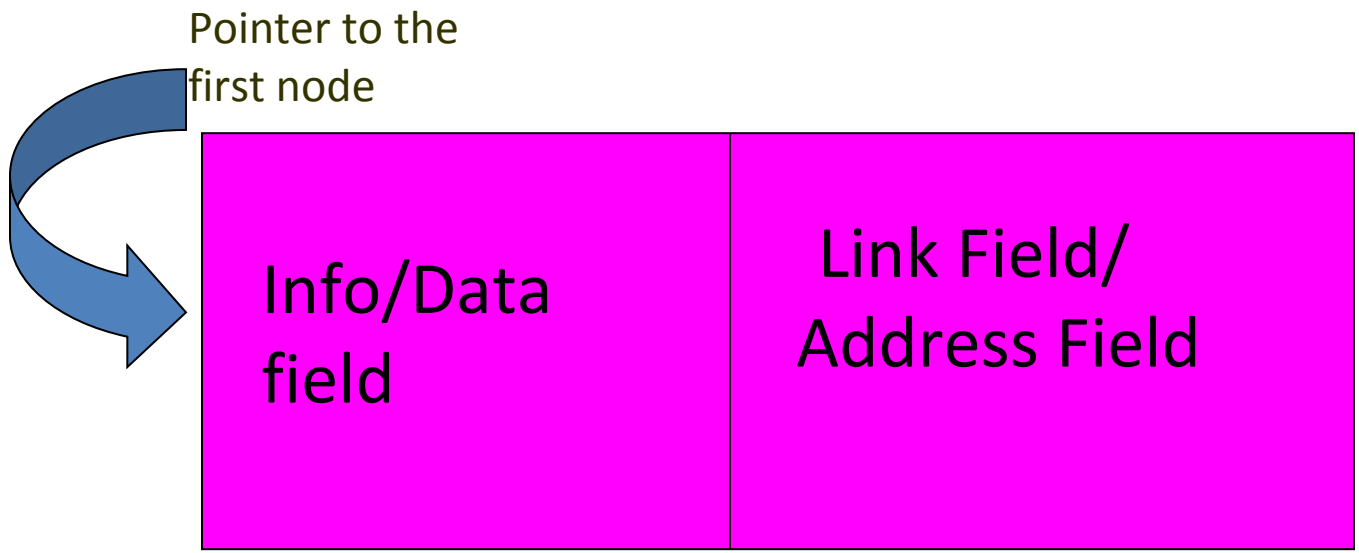
- To access elements in the list in the correct order, with each element we store the address or location of the next element in that list.
- Thus, associated with each data item in a linked representation is a pointer to the next element.
- This pointer is often referred to as a link.
- In general, a *node is a collection of data, DATA1, ...,* DATA*n and links LINK1, ...,LINKm. Each item in a node is called a field. A field contains either a data* item or a link.

Linked lists

- Linked lists are appropriate when the number of data elements to be represented in the data structure at once is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- Each node does not necessarily follow the previous one physically in the memory.
- Linked lists can be maintained in sorted order by inserting or deleting an element at the proper point in the list.

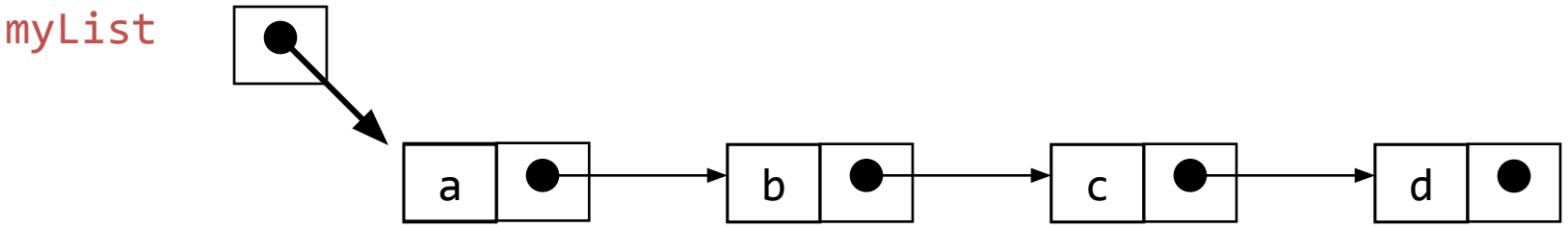
What is Linked List?

- A linked list is an ordered collection of finite homogeneous data elements called nodes where the linear order is maintained by means of links or pointers.



Anatomy of a linked list

- A linked list consists of:
  - A sequence of nodes



Each node contains a value

and a link (pointer or reference) to some other node

The last node contains a null link

The list may (or may not) have a header

More terminology

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)

Realization of Linked List Using Arrays

- Let L ={Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
- L is an ordered set
- List is stored in Data -1D array
- Elements are not stored in the same order as in the set L.
- To maintain the sequence ,second array ,Link is added
- List starts at 10^{th} Loc

MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec

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

Head →

MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Dynamic memory management

- In which memory resources are allocated whenever they are needed, and freed whenever they are not necessary anymore

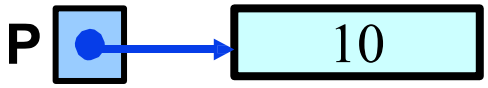
- Linked lists, are defined precisely in such a way as to allow for dynamically allocating and deallocating memory, depending on the requirements of the application.

Pointers in C

- **Declare Pointer P of type integer: Int *P;**
- **Allocate memory and assign its address to P:**

  P = (int *) malloc (sizeof (int));
- **Store value 10 in the allocated memory:**

***P = 10;**



Notes:

1. The new allocated memory is anonymous (i.e. does not have a name of its own
2. The only way we can access this location is via P.

Pointers in C – Dynamic Memory Allocation

- **But what does this complicated line do?:**

P = (int *) malloc (sizeof (int));

Read This First !!!

1

int Sizeof(int):

A function that takes a primitive data type like int, float, char, etc. and just return its size in bytes!

2

void * malloc(int):

A function that takes a *number* of bytes, and dynamically allocate memory of that size in bytes, and finally returns its address!

(in this case to P)

3

But, what is this void * in front of malloc?

Well, we do not know what type of address to return until we use malloc in our programs!!

So, that's why we cast void * to int * in this example, since P is declared of type int* .

Types of linked lists

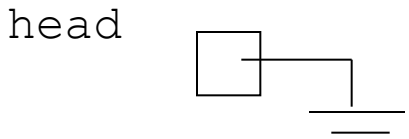
- The number of pointers are maintained depending on the requirements and usage

- Based on this ,linked list are classified into three categories.
  - Singly linked lists
  - Circular linked lists
  - Doubly linked lists

Empty List

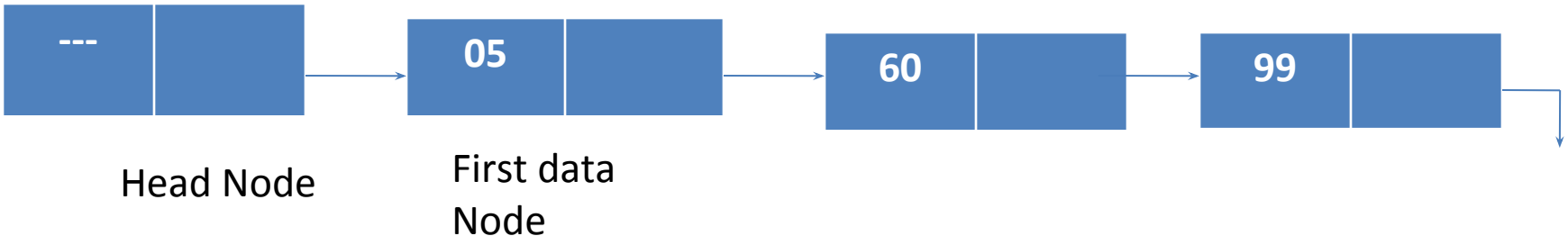
- Empty Linked list is a single pointer having the value of NULL.

```
head = NULL;
```



Singly Linked List

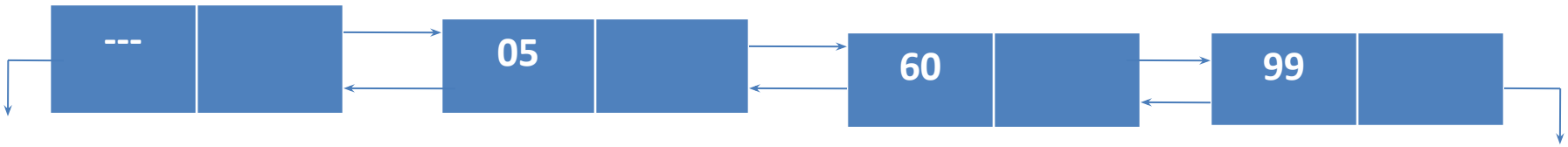
A linked list in which each node contains only one link field pointing to the next node in the list



First node is called the header node where no data element is stored, but the link field holds the address of the node containing very first data element.

Doubly Linked List

•A doubly linked list is a linear data structure in which each node can have any number of data fields and two link fields which are used to point to the previous node and the next node.



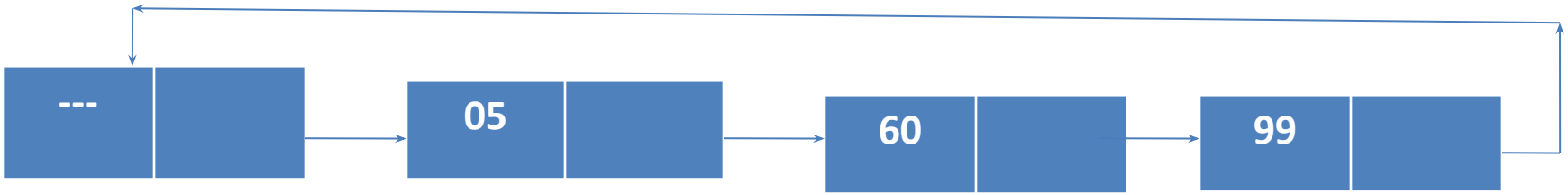
Head Node

First data Node

Circular Linked List

A linked list where the last node points to the header node is called a circular linked list.

There are no NULL links.

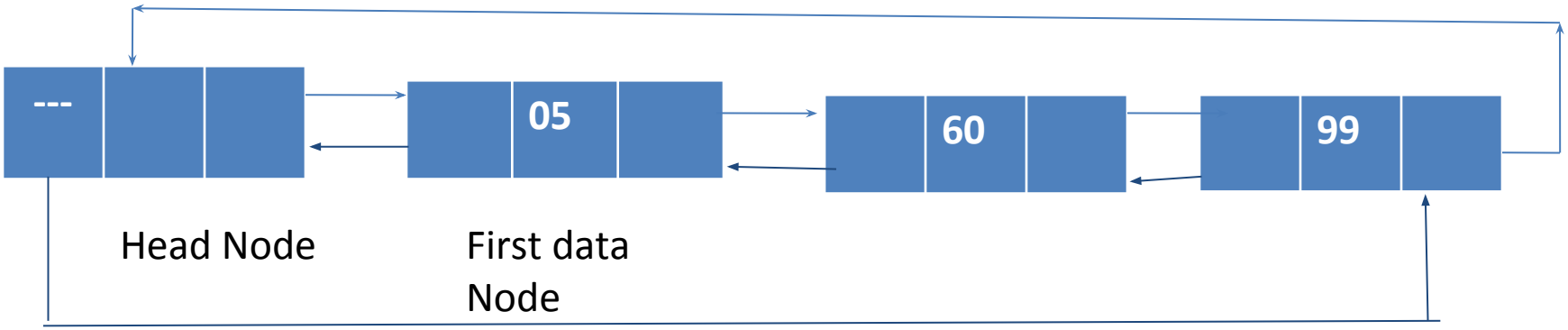


Head Node

First data Node

Circular Doubly Linked List

A Doubly linked list where the last node points to the header node and Header node Points to last node is called a circular Doubly linked list.
There are no NULL links.



MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Basic Operations on a list

- Creating a List
- List Traversal
- Inserting an element in a list
- Deleting an element from a list
- Searching a list
- Reversing a list
- Merging two linked lists into a larger list

Data Structure of Node

```
struct node
{
    int data;
    struct  node *next;
};
struct node *head;
```

Creation of SLL

```
head=(struct node *)malloc(sizeof(struct node));
head->next=NULL;

//Creation of Link List

 Algorithm create(*H)
{
       temp=H;
       repeat untill choice ='y'
      {
          allocate memory to curr;
           accept curr->data;
            curr->next=NULL;
           temp->next=curr;
           temp=curr;      //temp=temp->next
           Read choice;
        }
}
```

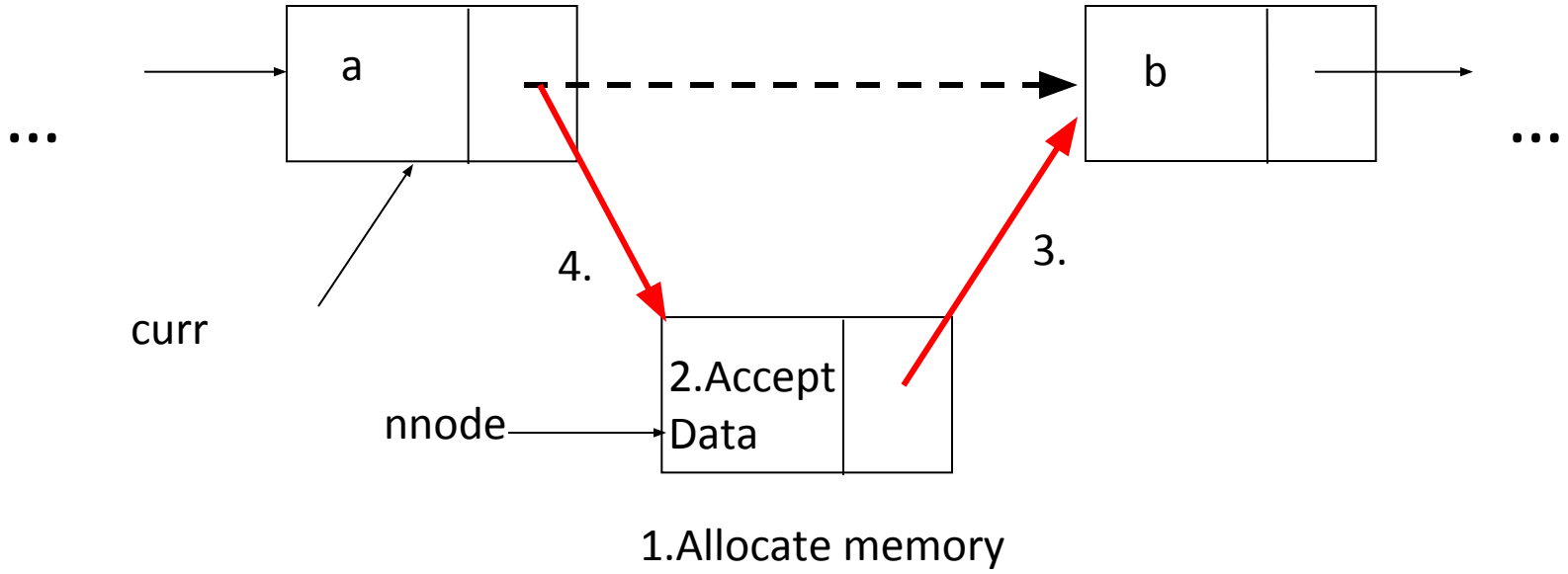
Display Link List

```
Algorithm display(*H)
{
   if H->next ==NULL
        Print " list is empty "
else
{
   //print head node values
   curr=H->next;
   while(curr!=NULL)
   {
        Print curr,curr->data,curr->next;
        curr=curr->next;
   }
}
```

Finding length of Link List

```
Algorithm  len(*H)
{
    curr=H->next;
    while(curr!=NULL)
    {
          i++;
          curr=curr->next;
     }
      return(i);
}
```

Insertion in a linked list

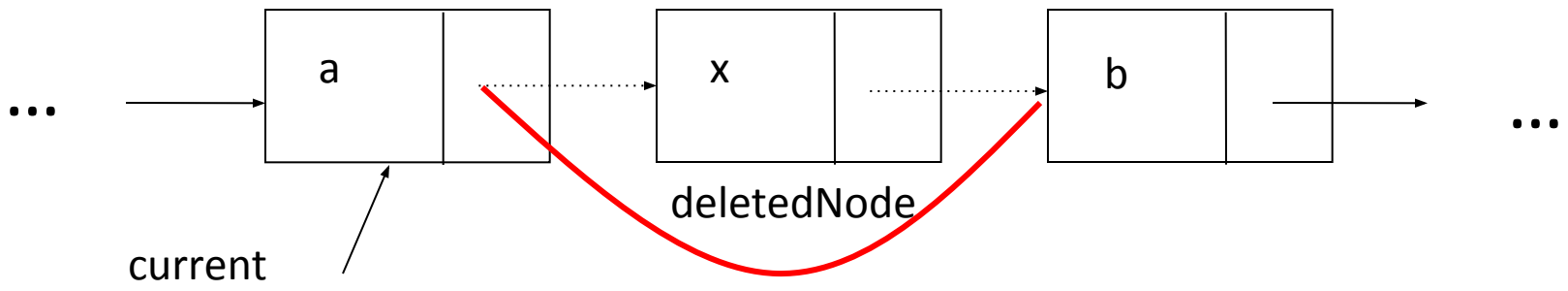



```
1.Allocate memory for nnode ;
2.Accept data for nnode;
3.nnode->next = curr->next;
4.curr->next = nnode;
```

Insert a new node by position

```
Algorithm Insertbypos(*H)
{
    i=1 ;   curr=H;
    //allocate memory for nnode node;
    read  nnode->data and pos;          //accept  data & position to be inserted:
    k=len();
    If(pos>k+1)
            //Print “Data can't be inserted”;
    else
    {
     while(curr!=NULL  && i<pos)
     {
          i++;
          curr=curr->next;
     }
          nnode->next=curr->next;
          curr->next=nnode;
     }
}
```

Deletion from a linked list




```
Node *deletedNode = current->next;
current->next = current->next->next;
delete deletedNode;
```

Delete a node by position

```
Algorithm delpos(*H)
{
  prev=H; ctr=1;
  curr=H->next;
  read pos;
//Accept position of data to be deleted:

 k=len();
 if (k<pos)
  //display Data can't be deleted;
 else
 {
```

```
while(ctr<pos && curr!=NULL)
{
       ctr++;
      prev=curr;
      curr=curr->next;
}
     temp=curr;
     prev->next=curr->next;
     curr->next=NULL;
     free(temp);
   }
}
```

Reverse of Linked List(Using Pointers)

```
Algorithm  reverse(*H)
{
    prev=NULL;
    curr=head->next;
    while(curr!=NULL)
    {
     future=curr->next;
     curr->next=prev;
     prev=curr;
     curr=future;
    }
    head->next=prev;
}
```

Sorting of SLL by using pointers

```
Algorithm sort(*H)
{
    len=len(H);
    for  i=1  to len-1
    {
            prev=H;
      curr=H->next;
            for  j=0 to <l-i
     {
            temp=curr->next;
```

```
if(curr->data > temp->data
{
        prev->next=temp;
       curr->next=temp->next;
       temp->next=curr;
       prev=temp;
}
else
{
        prev=curr;
        curr=curr->next;
}
}   //end for inner for
}  //end for outer for
} //end Algorithm
```

Merging of SLL by using pointers

```
Algorithm merge(*H1,*H2)
{
 curr1=H1->next;
 curr2=H2->next;
 if(curr1->data<curr2->data)
{
   temp=head1;
   flag=1;
}
else
{
    temp=head2;
    flag=0;
}
while(curr1!=NULL && curr2!=NULL)
{
    if(curr1->data<curr2->data)
    {
            temp->next=curr1;
            temp=curr1;
            curr1=curr1->next;
    }
else
{
      temp->next=curr2;
      temp=curr2;
      curr2=curr2->next;
   }
 }
   if(curr1==NULL)
          temp->next=curr2;
   if(curr2==NULL)
          temp->next=curr1;
   if(flag==1)
          display(head1);
   else
          display(head2);
}
```

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]

Doubly-Linked Lists

- **It is a way of going *both* directions in a linked list, *forward* and *reverse*.**
- **Many applications require a quick access to the *predecessor* node of some node in list.**

A node in a doubly-linked list store two references:

- A ***next*** link; that points to the next node in the list, and
- A ***prev*** link; that points to the previous node in the list.

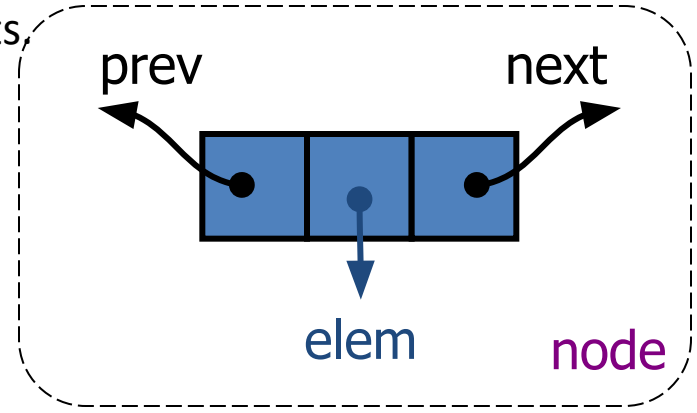
Advantages:

Convenient to traverse the list backwards.

Simplifies insertion and deletion because you no longer have to refer to the previous node.

Disadvantage:

Increase in space requirements.

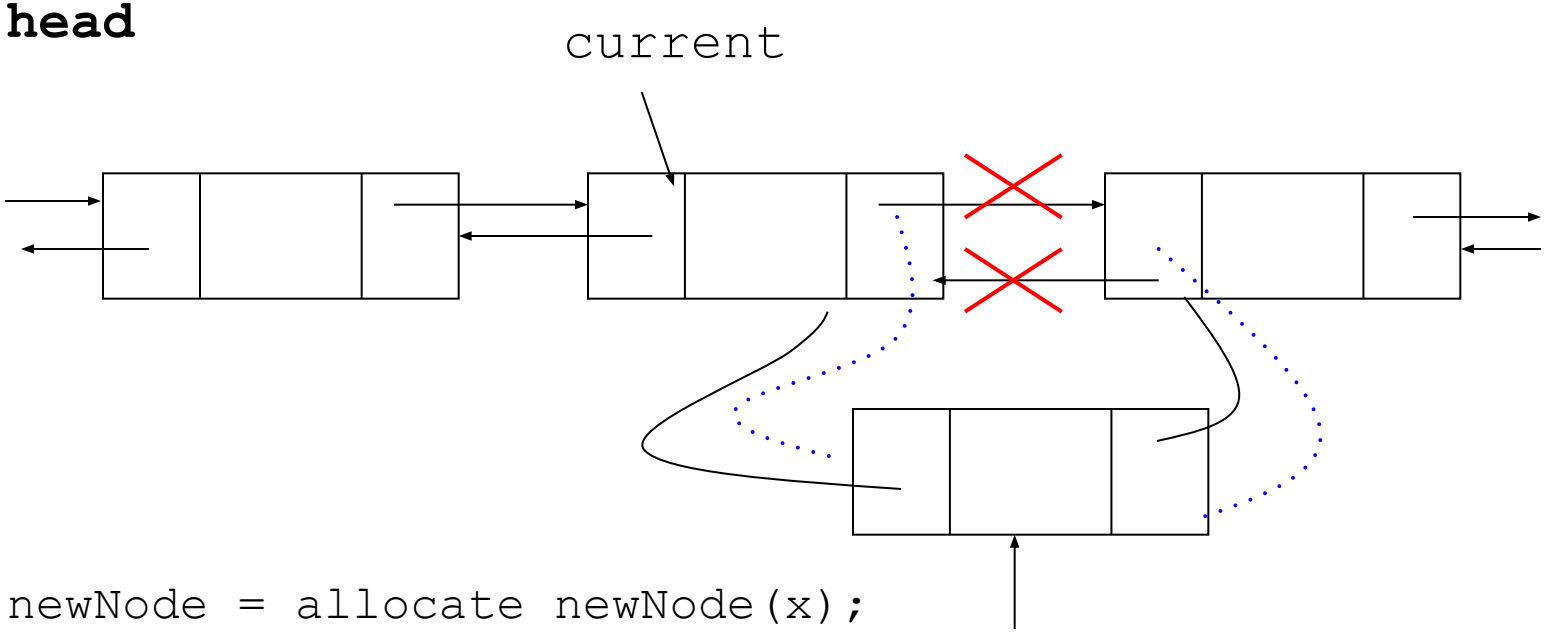


Doubly Link List creation

```
struct  node
{
  char data[20];
  node *next,*prev;
};
```

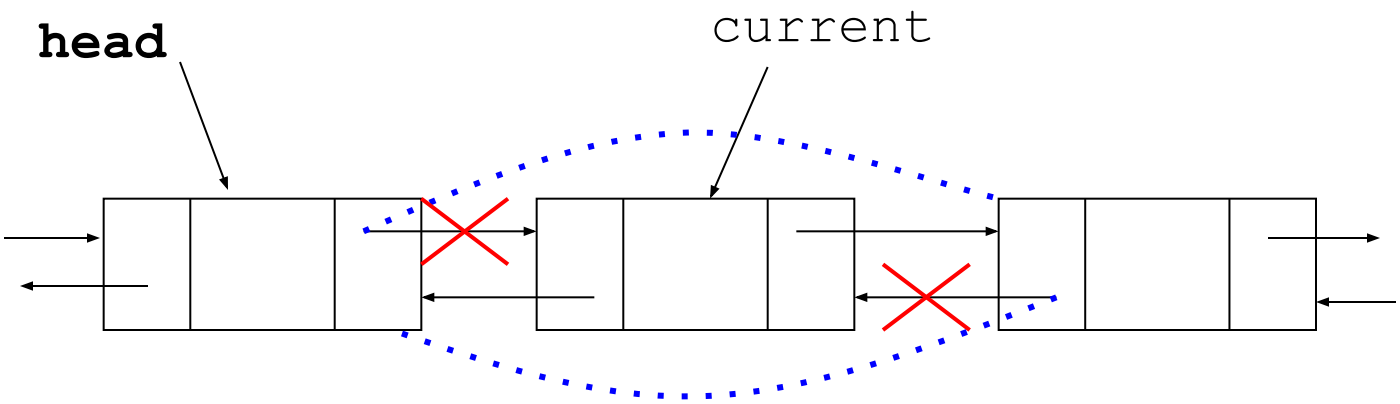
```
Algorithm Create(*H)
{
 temp=H;
 repeat till choice =y
 {
    //allocate memory for new node
    //in curr pointer
    temp->next=curr;
    curr->prev=temp;
    curr->next=NULL;
    temp=curr;
 }
    Read choice;
}
```

Insertion




```
newNode = allocate newNode(x);
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

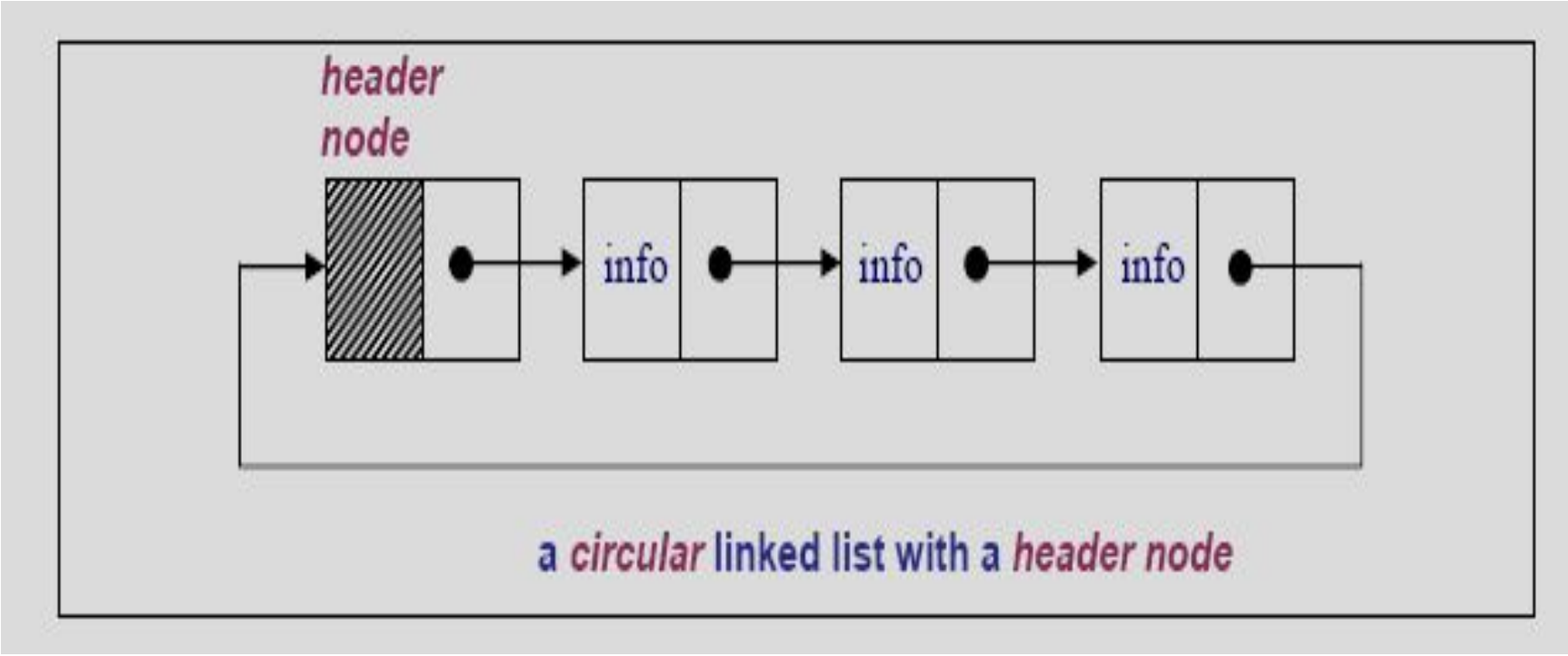
Deletion




```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
delete oldNode;
current = head;
```

MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Circular Linked Lists



Allocation of memory for head node using constructor

```
head=(struct node *)malloc(sizeof(struct node));
head->next=head;

//Creation of Link List
Algorithm create(*H)
{       temp=H;
       repeat until choice =‘y’ {
          allocate memory to curr;
           accept curr->data;
            curr->next=H;
           temp->next=curr;
           temp=curr;      //temp=temp->next
           Read choice;
        }
}
```

Advantages over Singly-linked Lists

- Quick update operations:

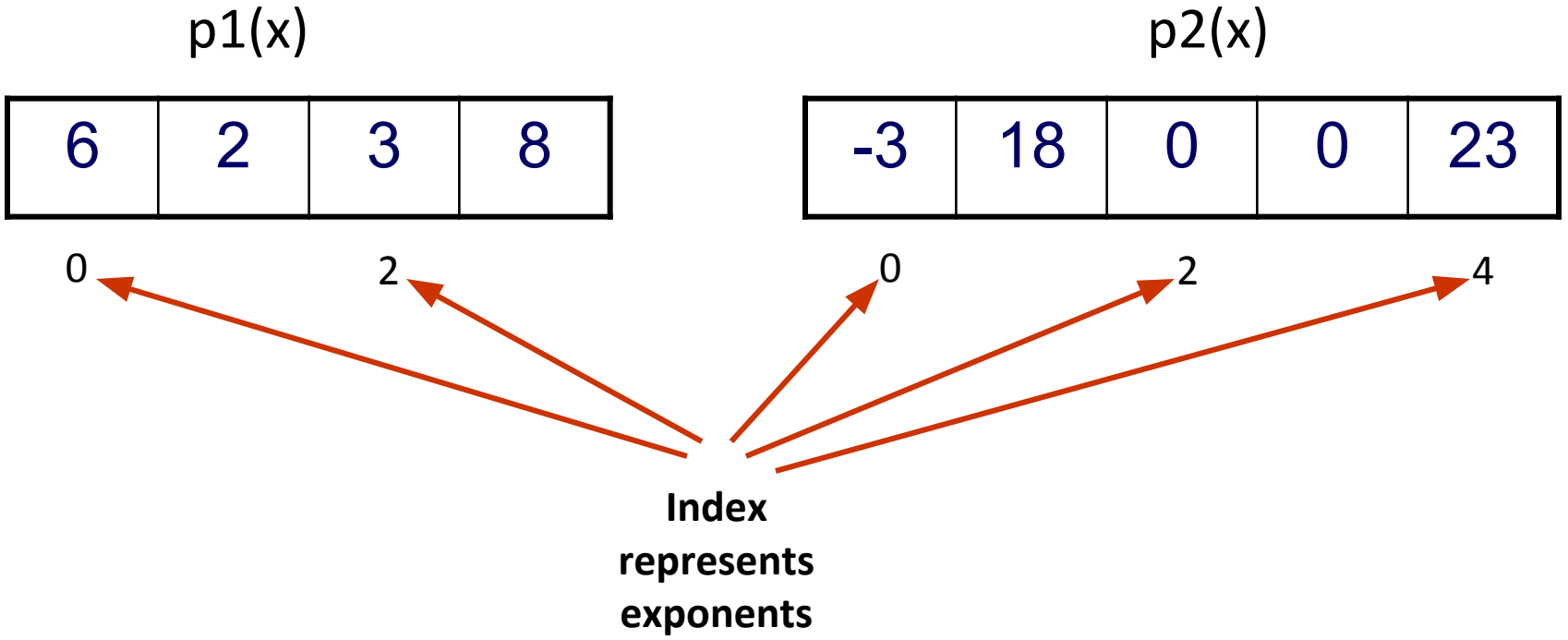
  such as: insertions, deletions at *both* ends (head and tail), and also at the middle of the list.
- A node in a doubly-linked list store two references:
  - A ***next*** link; that points to the next node in the list, and
  - A ***prev*** link; that points to the previous node in the list.

DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some applications)
  - 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)

• Polynomial

- Array Implementation:
- $p1(x) = 8x^3 + 3x^2 + 2x + 6$
- $p2(x) = 23x^4 + 18x - 3$



• Polynomial (continued)

•This is why arrays aren't good to represent polynomials:

• $p3(x) = 16x^{21} - 3x^5 + 2x + 6$

6	2	0	0	-3	0	.....	0	16

WASTE OF SPACE!

• Polynomial (continued)

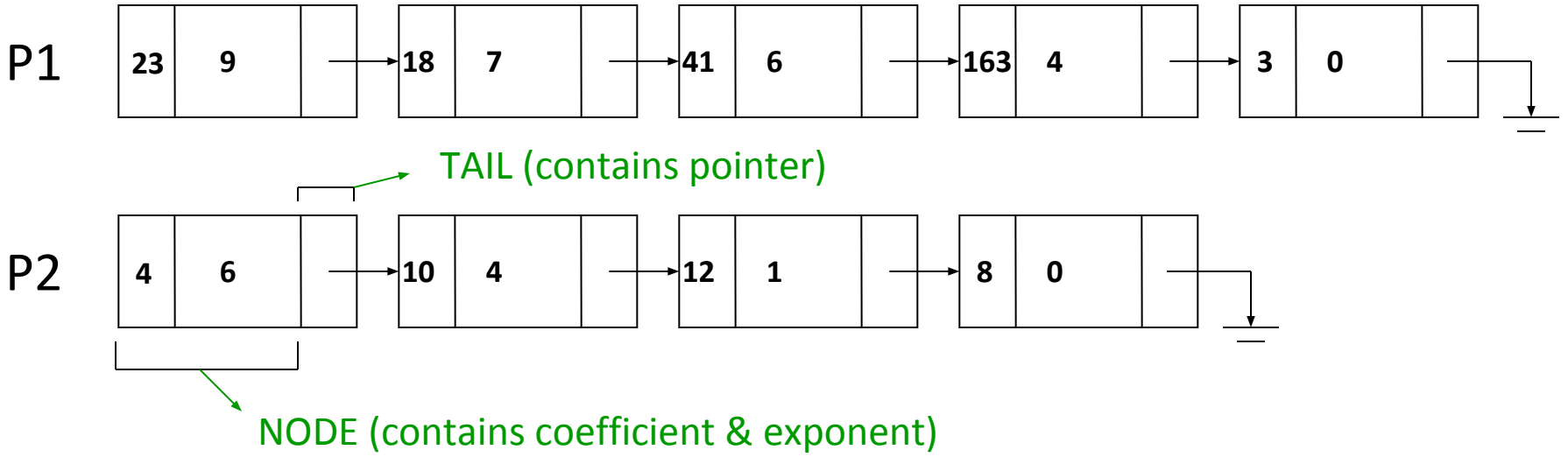
- Advantages of using an Array:
  - only good for non-sparse polynomials.
  - ease of storage and retrieval.
- Disadvantages of using an Array:
  - have to allocate array size ahead of time.
  - huge array size required for sparse polynomials.

  Waste of space and runtime.

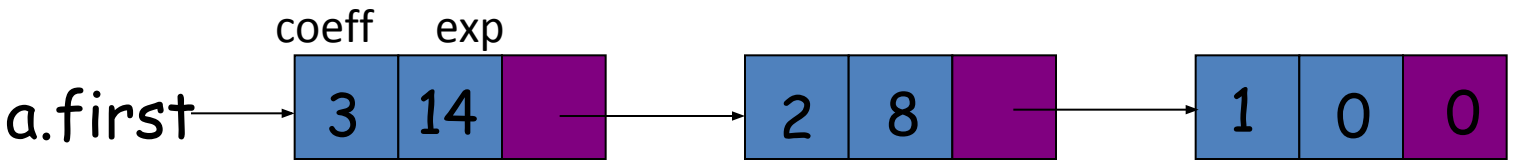
• Polynomial (continued)

- Linked list Implementation:

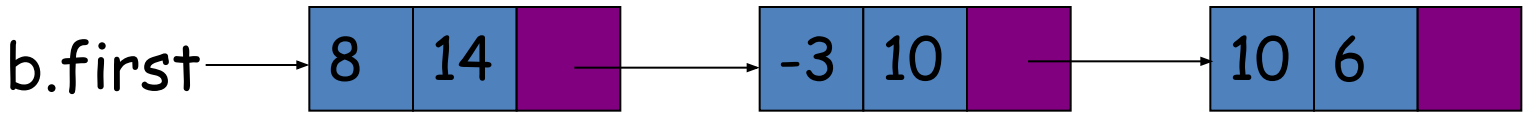
- $p1(x) = 23x^9 + 18x^7 + 41x^6 + 163x^4 + 3$
- $p2(x) = 4x^6 + 10x^4 + 12x + 8$



Revisit Polynomials



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



**

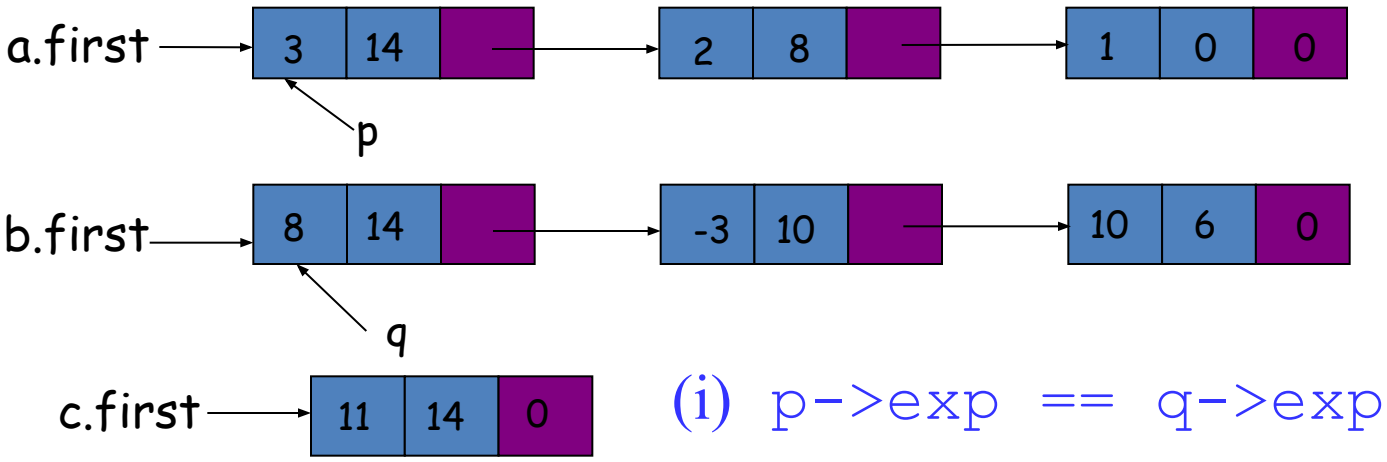
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

Node structure of Polynomial

```
struct polyNode
{
   // All members in "struct" are
public
   int coef;   // coefficient
   int exp;    // exponent
   struct polyNode *next;
};
```

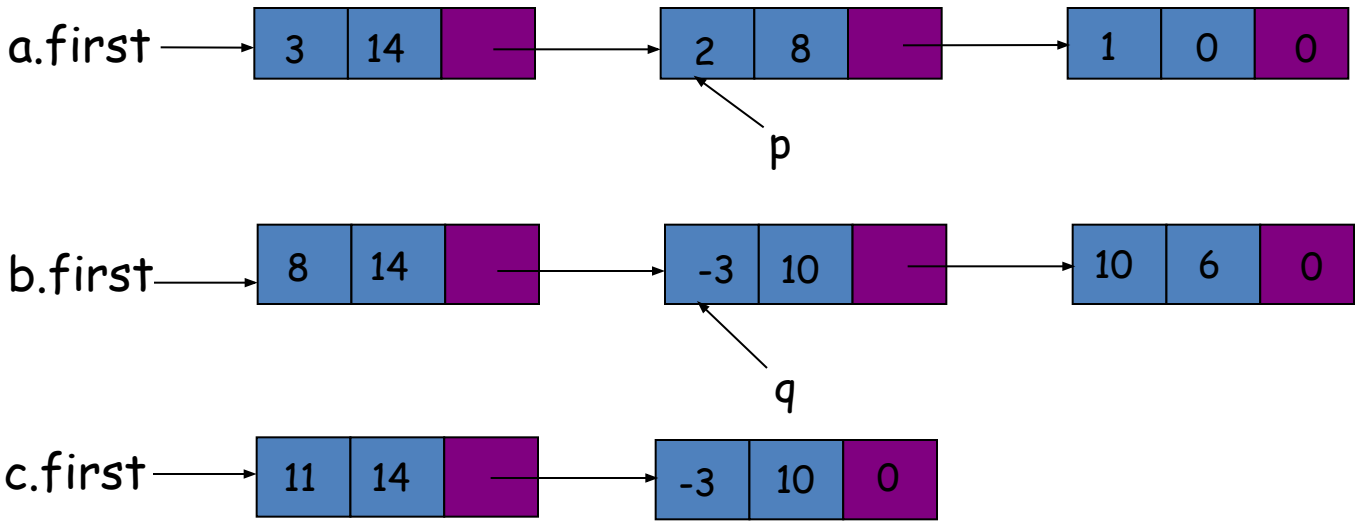
Addition of Two Polynomials (1)

- It is an easy way to represent a polynomial by a linked list.
- Example of adding two polynomials a and b



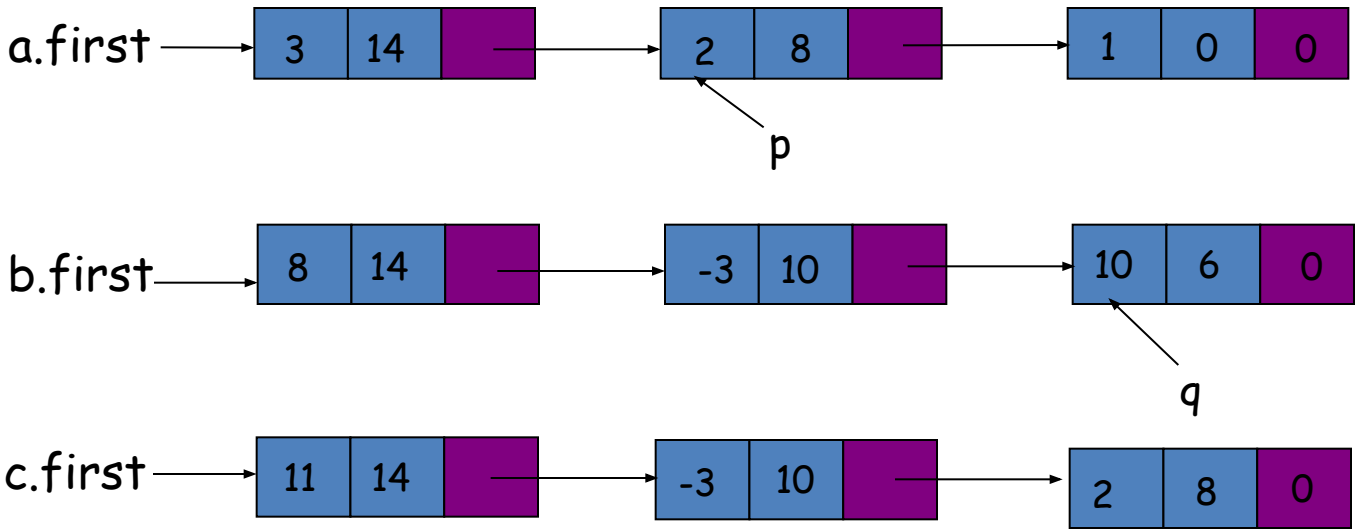
(i) `p->exp == q->exp`

Addition of Two Polynomials (2)



(ii) `p->exp < q->exp`

Addition of Two Polynomials (3)



(iii) `p->exp > q->exp`

MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

• Polynomial (continued)

- 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.

Addition of Polynomial

- Allocate the memory space for head;
- Head->exp=-1

```
Algorithm add(*H1,*H2)
{
    Allocate a memory for  H3;
        head3->exp=-1;
        t3=H3;
        t1=H1->next;
        t2=H2->next;
        while(t1->exp!=-1||t2->exp!=-1)
        {
              if(t1->exp==t2->exp)
              {
                    Allocate the  memory  for temp;
                    Add  t1 coeff and t2 coeff in t3 coeff
                    copy one of the exponent in t3 exp
                    t3->next=temp;
                    temp->next=head3;
                    t3=temp;
                    Move t1 to next node ;
                     Move t2 to next node
              }
```

Addition of Polynomial

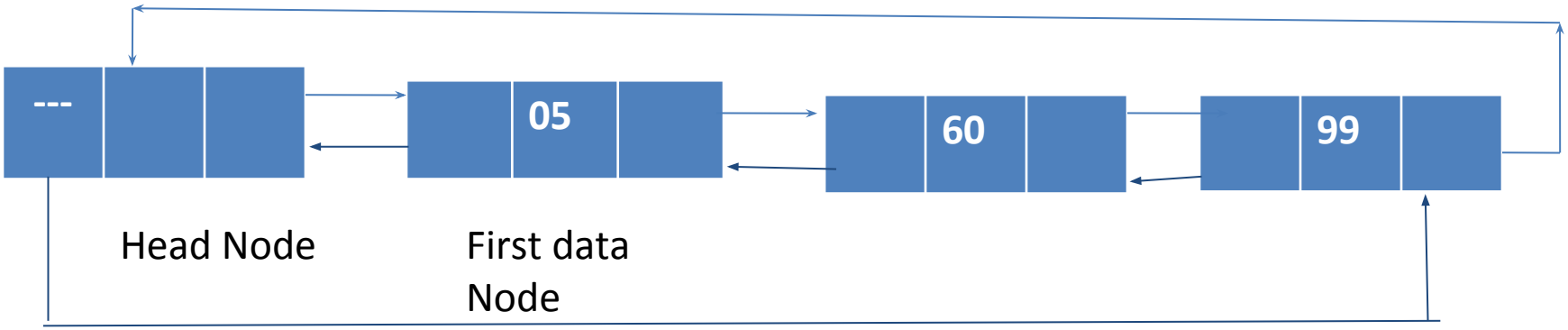
```
else
        if exponent of p1 < exponent of p2
            Copy node of p2 to end of p3.
        else                                    //exponent of p1 > exponent of p2
            Copy node of p1 to end of p3

    }//end of while

    }   //end algorithm
```

Evaluation of Polynomial

```
Algorithm  eval(*H, x)
{
    cur=H->next;
    while  not end of list
    {
     using value of x find the sum of terms;
     move cur to next node ;
    }
    print  sum;
}
```

Doubly Circular Linked List

A Doubly linked list where the last node points to the header node and Header node Points to last node is called a circular Doubly linked list.
There are no NULL links.



Basic Operations on DCLL

- Create
- Display in forward and backward direction
- Delete a Node
- Add a Node

Creating DCLL

```
struct node
{
int data;
struct node *next,*prev;
};
```

Allocate memory to a head node;

```
head->next=head;
head->prev=head;
```

```
Algorithm_CreateCDLL(head)
{
temp=head;
repeat if choice==’y’

{
curr=allocate memory;
accept curr data;
temp->next=curr;
curr->prev=temp;
head->prev=curr;
curr->next=head
temp=curr;
}
```

Display DCLL

```
Display_forward(Head)
{
temp=head->next;
while(temp!=head)
{
print temp->data;
temp=temp->next;
}
```

```
Display_backward(Head)
{
temp=head->prev;
while(temp!=head)
{
print temp->data;
temp=temp->prev;
}
```

Adding a node in DCLL

```
AddanodeCDLL(head)
{
len=lenCDLL(head);
curr=allocate memory;
accept curr->data;
curr->next=curr->prev=NULL;
ask user to enter position in POS
if(POS>len+1)
print ‘not possible to add’
if(POS<len)
{
count=1
prevptr=head;
temp=head->next;
while(temp!=head && count!=POS)
{
prevptr=temp;
temp=temp->next;
}
prevptr->next=curr;
curr->next=temp;
temp->prev=curr;
curr->prev=prevptr;
}//end of if
if(POS==len+1)//adding node at end
{
last=head->prev;
last->next=curr;
curr->prev=last;
curr->next=head;
head->prev=curr;
}
```

Deleting a node from DCLL

```
deleteanodeCDLL(head)
{
len=lenCDLL(head);
ask user to enter position of node to delete in POS
temp=head->next;
count=1;
while(temp!=head && Count!=POS)
{
prevptr=temp;
temp=temp->next;
}
prevptr->next=temp->next;
temp->next->prev=prevptr;
temp->next=temp->prev=NULL;
free(temp);
}
```

Generalized Lists

- A generalized list, A, is a finite sequence of $n \geq 0$ elements, $a_0, a_1, a_2, \ldots, a_{n-1}$, where a_i, is either an atom or a list. The elements a_i, $0 \leq i \leq n-1$, that are not atoms are said to be the sublists of A.

- A list A is written as $A = (a_0, \ldots, a_{n-1})$, and the length of the list is n.

- A list name is represented by a capital letter and an atom is represented by a lowercase letter.

- a_0 is the head of list A and the rest $(a_1, \ldots, a_{n-1})$ is the tail of list A.

Examples of Generalized Lists

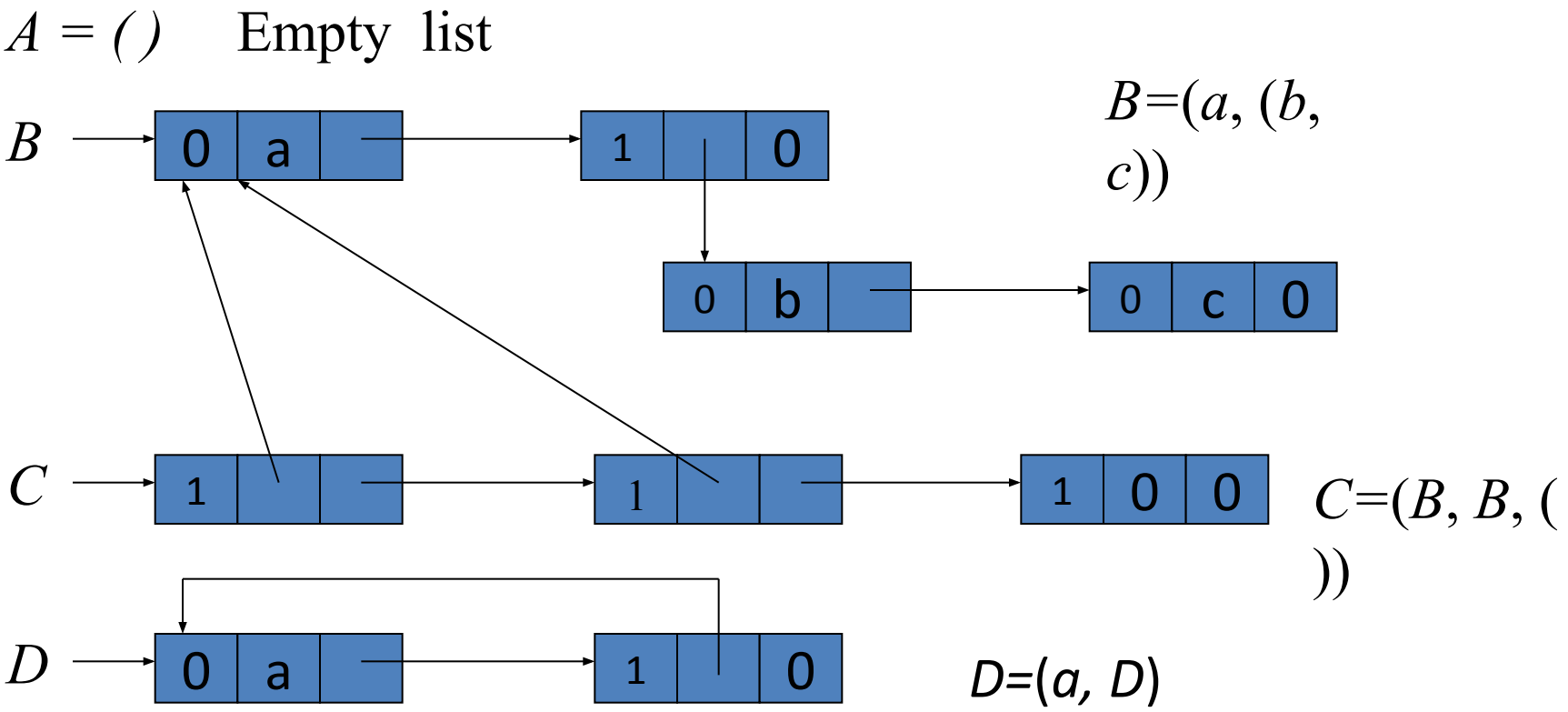
- *A* = ( ): the null, or empty, list; its length is zero.
- *B* = (*a*, (*b, c*)): a list of length two; its first element is the atom *a*, and its second element is the linear list (*b, c*).
- *C* = (*B, B,* ( )): A list of length three whose first two elements are the list *B*, and the third element is the null list.
- *D* = (*a, D*): is a recursive list of length two; *D* corresponds to the infinite list *D* = (*a*, (*a*, (*a*, …))).
- head(*B*) = '*a*' and tail(*B*) = (*b, c*), head(tail(*C*))=*B* and tail(tail(*C*)) = ( ).
- Lists may be shared by other lists.
- Lists may be recursive.

- It is a little surprising that every generalized list can be represented using the node structure

Tag=0/1	Data	Link

- Tag=0 means data
- Tag = 1 means pointer to the sublist

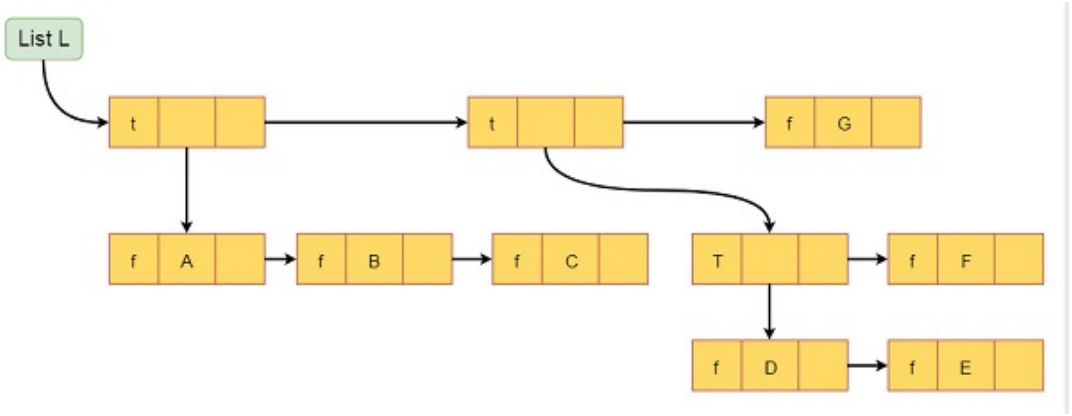
Representations of Generalized Lists



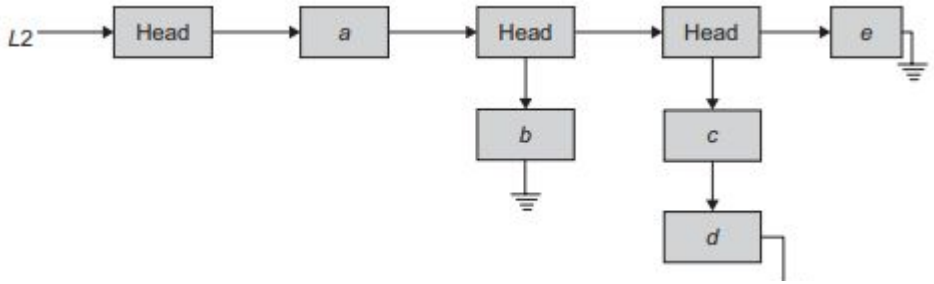
**

GLL Examples

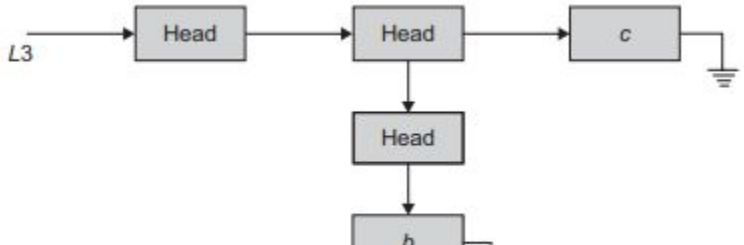
- L1=((A, B, C), ((D, E), F), G).



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



L3 = (a, ((b)), c).



Polynomial Representation using Generalized Linked List

Flag	Variable, Coefficient	Exponent	nLink

- Flag = 0 *variable* is present
- Flag = 1 *down pointer* is present
- Flag = 2 *coefficient* and *exponent* is present

General Polynomial

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

- *P*(*x*, *y*, *z*)=

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

- Rewritten as $Cz^2 + Dz$, where *C* and *D* are polynomials.
- Again, in *C*, it is of the form $Ey^3 + Fy^2$, where *E* and *F* are polynomials.
- In general, every polynomial consists of a variable plus coefficient-exponent pairs. Each coefficient may be a constant or a polynomial.

- Suppose we need to devise a data representation for them and consider one typical example, the polynomial *P(x,y,z) =*

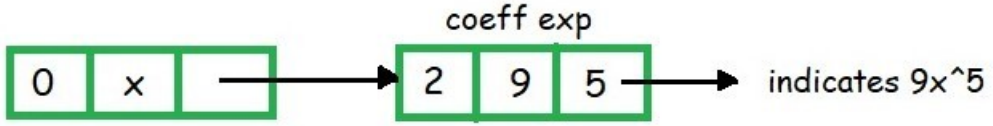
$$x^{10} y^3 z^2 + 2x^8 y^3 z^2 + 3x^8 y^2 z^2 + x^4 y^4 z + 6x^3 y^4 z + 2yz$$

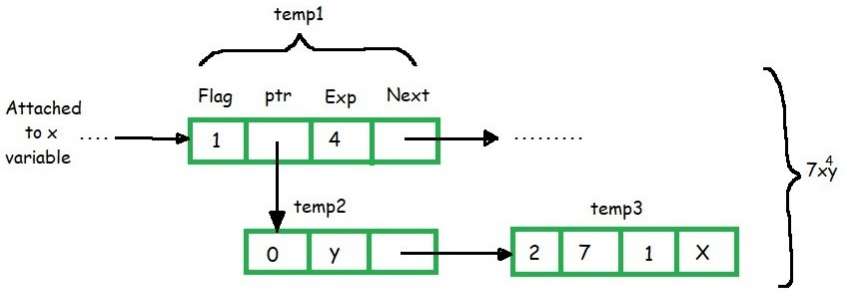
- One can easily think of a sequential representation for *P, say using* nodes with four fields: COEF, EXPX, EXPY, and EXPZ.
- But this would mean that polynomials in a different number of variables would need a different number of fields, adding another conceptual inelegance to other difficulties.
- These nodes would have to vary in size depending on the number of variables, causing difficulties in storage management.
- The idea of using a general list structure with fixed size nodes arises naturally if we consider re-writing
- *P(x,y,z) as*

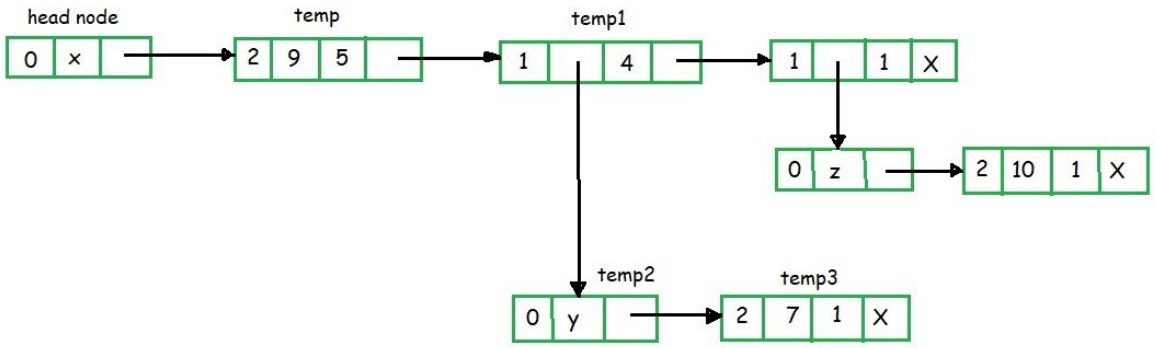
$$((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 + 6x^3)y^4 + 2y)z$$

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

- Flag = 0 *variable* is present
- Flag = 1 *down pointer* is present
- Flag = 2 *coefficient* and *exponent* is present

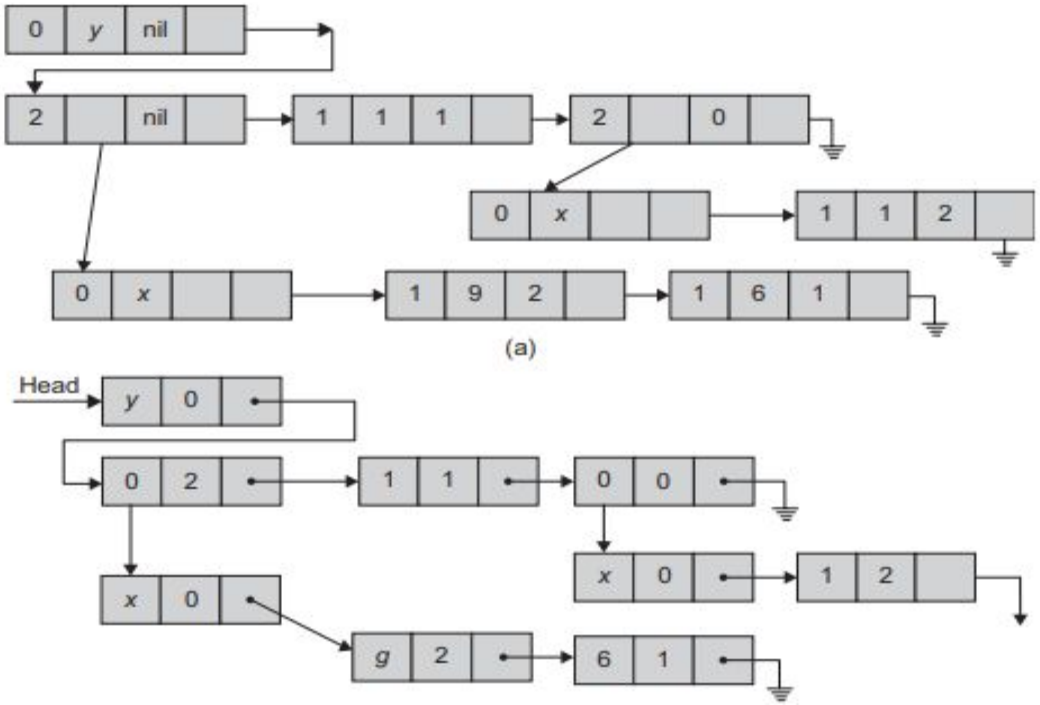






$$P(x, y) = 9x^2y^2 + 6xy^2 + y + x^2$$

$$P = y^2(9x^2 + 6^x) + y + x^2y^0$$



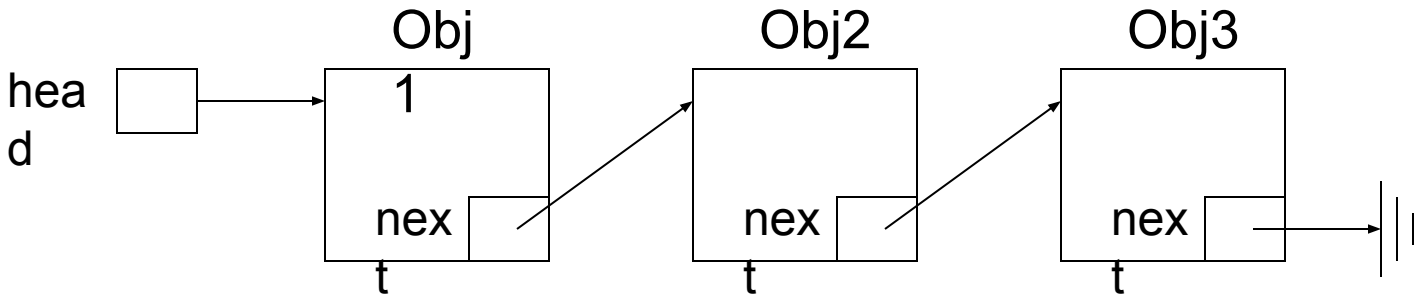
Garbage Collection

- After series of memory allocation and deallocation, there are blocks of free memory scattered throughout the available heap space
- To be able to reuse this memory, the memory allocator will usually link the freed blocks together in a free list by writing pointers to the next free block.
- Components of os ,memory management module, maintain this list.

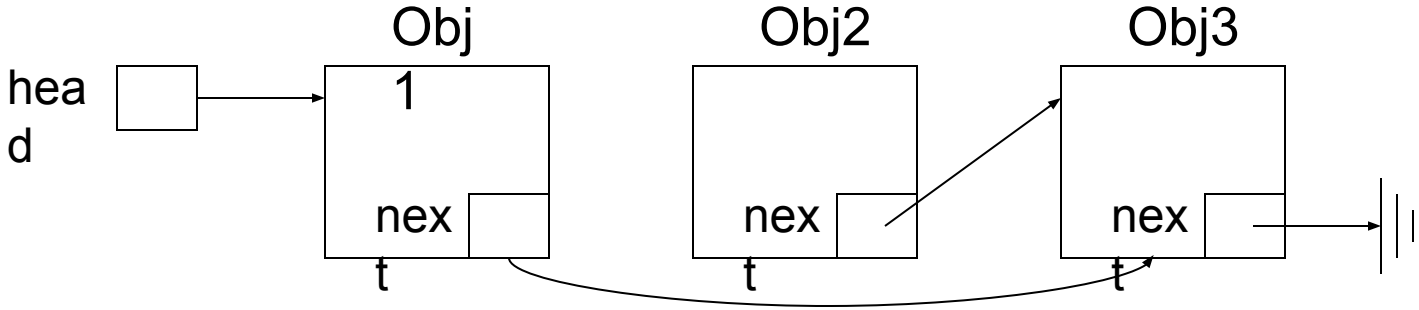
Garbage collection contd..

- Os periodically collects all the free blocks and inserts into the free pool.
- This is called as garbage collection.
- When CPU is idle, the garbage collection starts.
- It is invisible to programmer.

Example



- Assume programmer does the following
  - obj1.next = obj2.next;



Example

- Now there is no way for programmer to reference obj2
  - it's garbage
- In system without garbage collection this is called a *memory leak*
  - location can't be used but can't be reallocated
  - waste of memory and can eventually crash a program
- In system with garbage collection this chunk will be found and reclaimed

- **Mark Phase**
  When an object is created, its mark bit is set to 0(false).
- Set the marked bit for all the reachable objects (or the objects which a user can refer to) to 1(true)

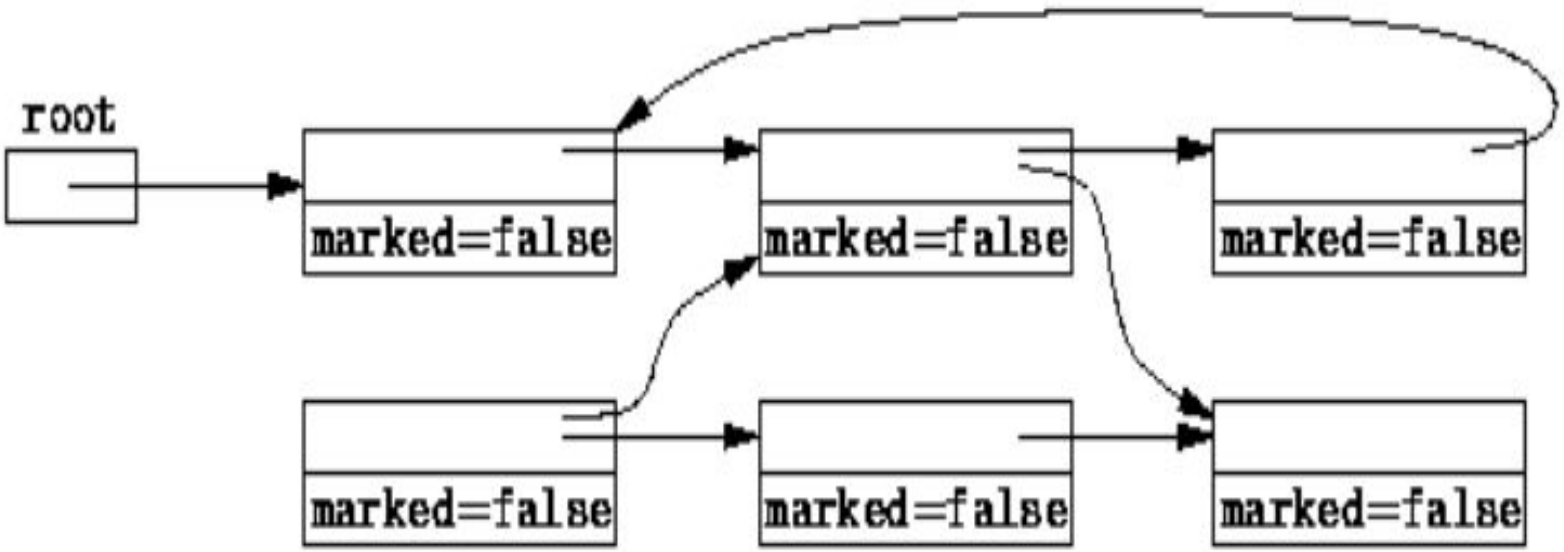
- **Sweep Phase**
  As the name suggests it “sweeps” the unreachable objects
  it clears the heap memory for all the unreachable objects.

- All those objects whose marked value is set to false are cleared from the heap memory, for all other objects (reachable objects) the marked bit is set to false

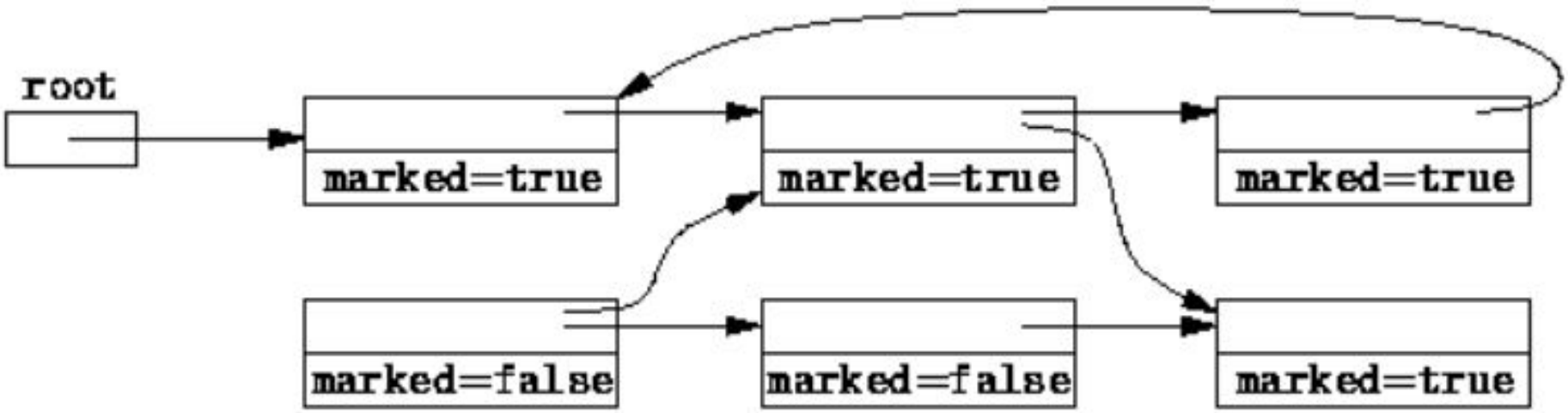
- Mark value for all the reachable objects is set to false,

- Since we will run the algorithm (if required) and again we will go through the mark phase to mark all the reachable objects.

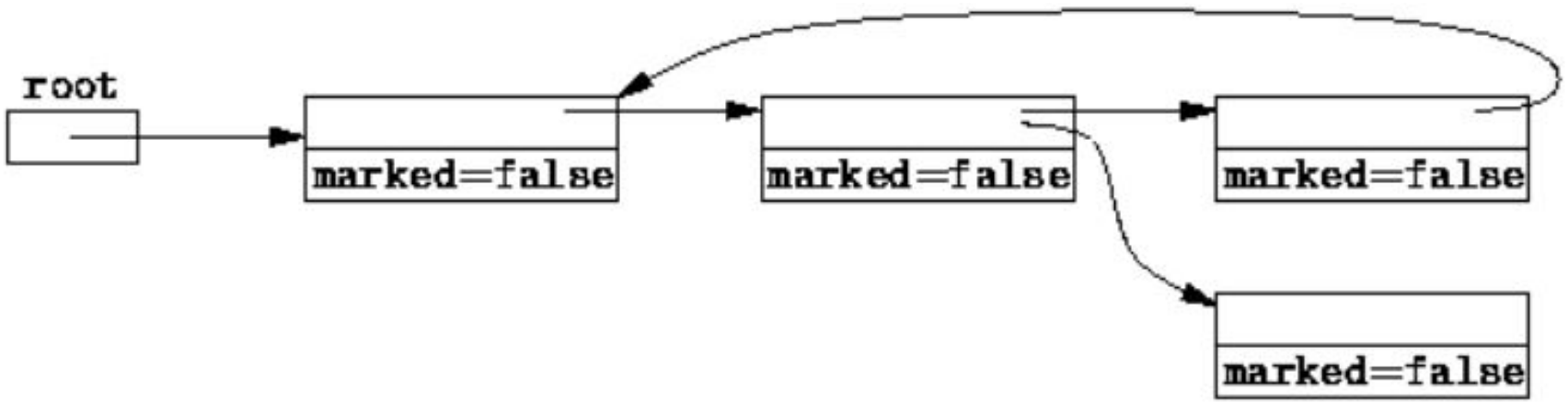
Example:
a) All the objects have their marked bits set to false.



b) Reachable objects are marked true

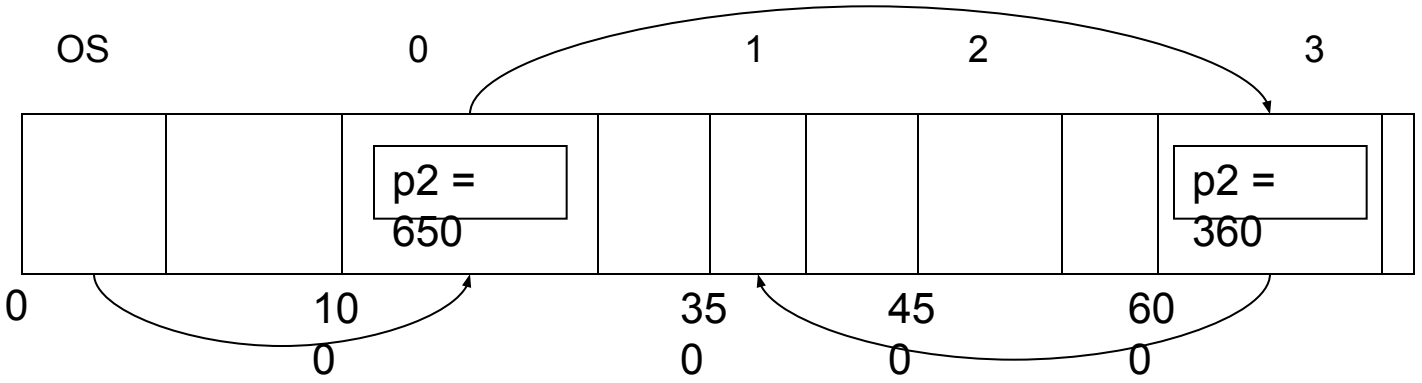


c) Non reachable objects are cleared from the heap.



Mark-and-Sweep

- Basic idea
  - go through all memory and mark every chunk that is referenced
  - make a second pass through memory and remove all chunks not marked



- Mark chunks 0, 1, and 3 as marked
- Place chunk 2 on the free list (turn it into a hole)