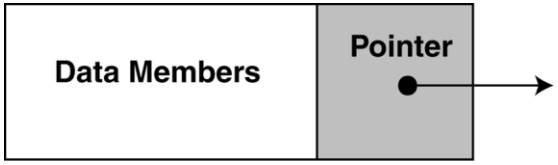
Linked List

The composition of a Linked List

- A linked list is a series of connected *nodes*, where each node is a data structure.
- Each node in a linked list contains one or more members that represent data.
- In addition to the data, each node contains a pointer, which can point to another node.

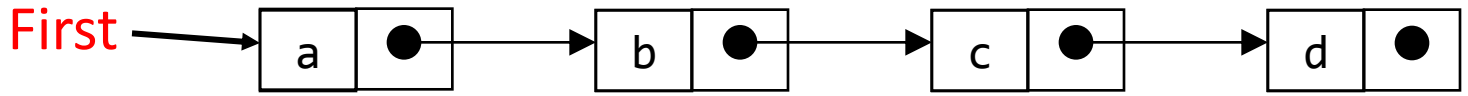


Linked Lists over Arrays

- A linked list can easily grow or shrink in size.
- Insertion and deletion of nodes is quicker with linked lists than with arrays.
- The size of the array is fixed. Linked list size is not fixed.
- In an array items have a particular position, identified by its index. In a list the only way to access an item is to traverse the list.
- Data accessing is faster with arrays than with Linked list.
- Array items are stored contiguously. Linked list items are stored non-contiguously.
- Linked list require extra space to hold address of next node.

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

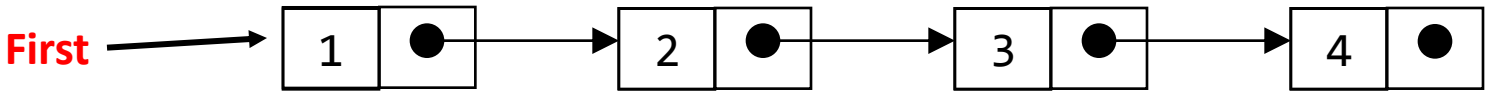
First contains the address of the first node of the list

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)

Singly-linked lists

- Here is a singly-linked list (**SLL**):



- Each node contains a value and a link to its successor (the last node has no successor)
- The first points to the first node in the list (or contains the null link if the list is empty)

Array Implementation of Linked Lists

```
#define NUMNODES 100
struct  nodetype {
                    int data, link;
} ;
struct  nodetype  node[NUMNODES];
```

A pointer to node is represented by array index. That is, a pointer is an integer between 0 to NUMNODES -1 that references a particular element of array node. The null pointer is represented by the integer -1.

node[p] is used to reference node(p)

info(p) is referenced by node[p].info

next(p) is referenced by node[p].next

null is represented by -1

Limitations of Array implementation

- The number of nodes that are needed often cannot be predicted when a program is written.
- Whatever number of nodes are declared must remain allocated to the program throughout its execution.
- The solution to this problem is to allow nodes that are dynamic. That is, when a node is needed, storage is reserved for it, and when it is no longer needed, the storage is released.

Linked List Using Dynamic Variables

Each node in the Linked List has two fields namely information field and a pointer to next node in the list. In addition, an external pointer points to the first node in the list.

```
struct node {
        int data;
        struct node *link;   /* link to next node */
} ;
typedef struct   node *NODE
```

NODE is defined using **typedef** as pointer to **struct node**

getnode()

The **getnode()** function returns the address of the new node, if availability list is not empty.

```
NODE getnode()
{
        NODE p;
        p = (NODE) malloc(sizeof(struct node));
        if (p == NULL)
        {
           printf("Out of memory");
            exit(0);
        }
        return (p);
}
```

The execution of the statement

p = getnode();

should place the address of an available node into p.

freenode()

The function **freenode()** will de-allocate the allocated memory.

```
void freenode(NODE p)
 {
        free (p);
}
```

Operations in a single linked list

- Insertion
- Deletion
- Searching or Iterating through the list to display items.

Insertion Description

- Insertion at the front of the list
- Insertion at the end of the list
- Insertion in the middle of the list

Deletion Description

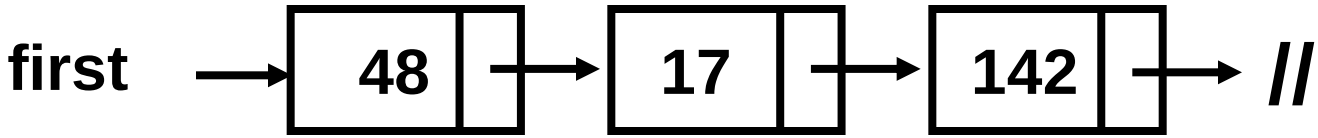
- Deleting from the front of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Insertion at the front

Steps:

- Create a Node
- Set the node data Values
- Connect the pointers

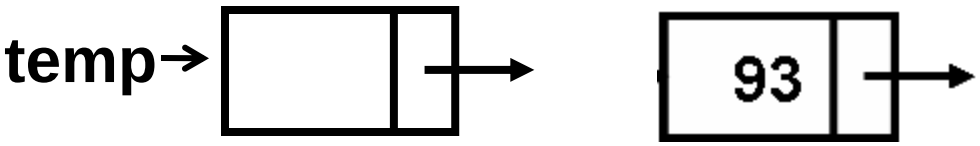
Insert front



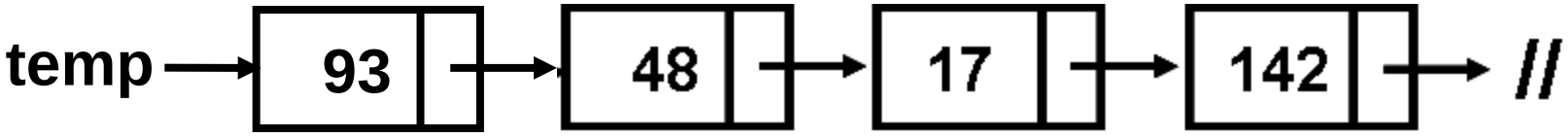
- Follow the previous steps and we get

Step 1

Step 2



Step 3



insert_front()

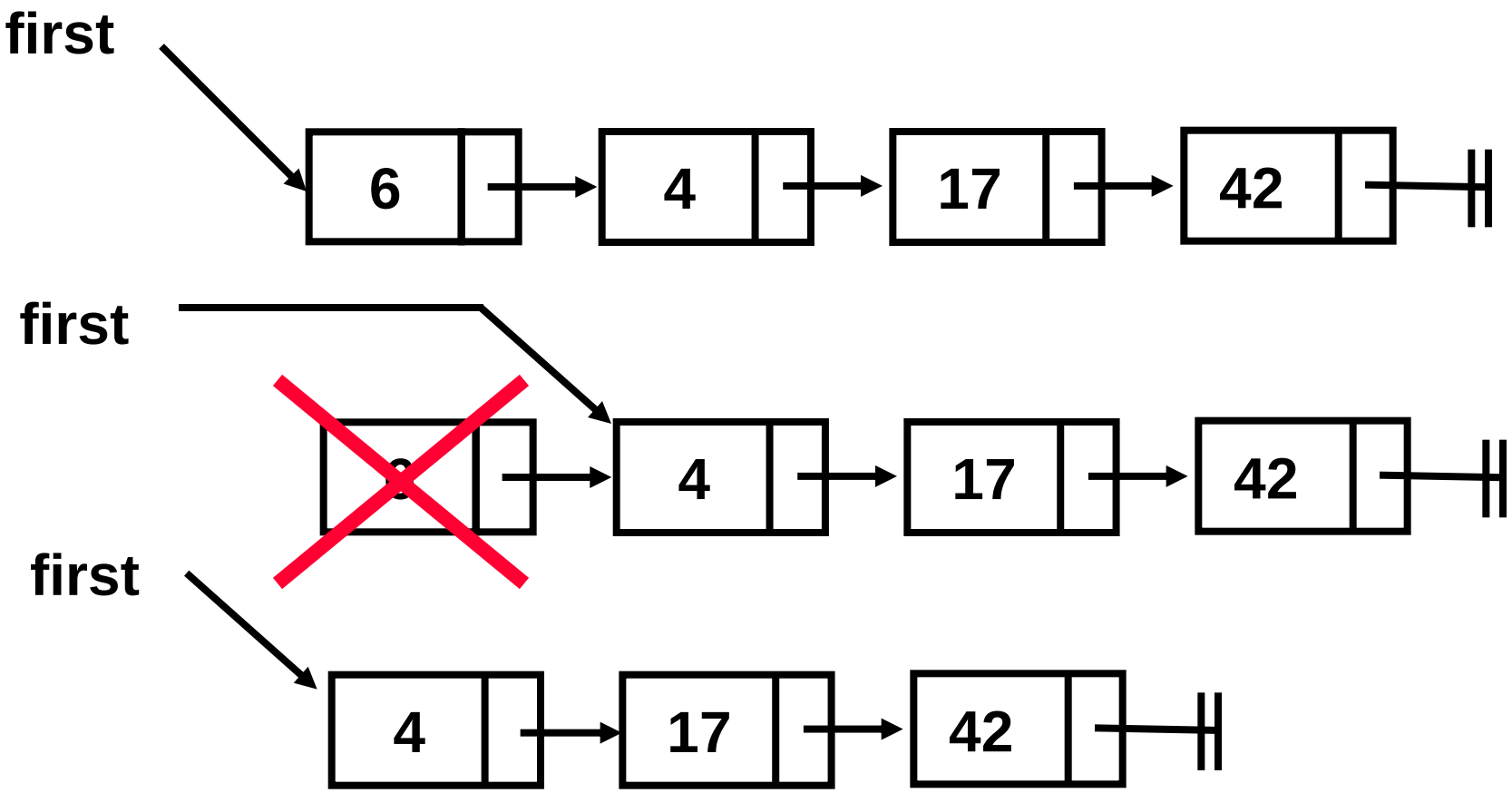
```
NODE insert_front(int  item, NODE first)
 {
    NODE temp; / * Node to be inserted */
    temp = getnode();   /* Get a node */
    temp -> data = item;
    temp -> link = first;
    first = temp;
    return (first);
}
```

Delete front

Steps

- Break the pointer connection
- Re-connect the nodes
- Delete the node

Delete front



delete_front()

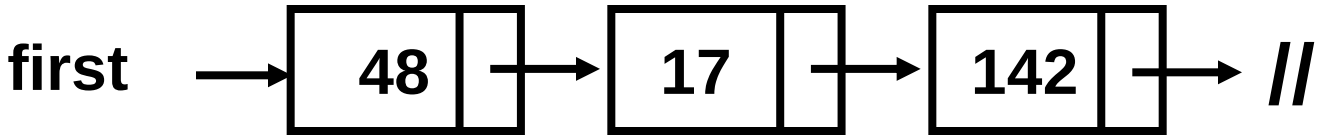
```
NODE delete_front(NODE first)
 {
    NODE temp;
    if (first  == NULL)
    {
       printf("List is empty");
       return first;
    }
    temp = first;
    first = temp->link;
    freenode (temp);
    return first;
}
```

Insert rear

Steps:

- Create a Node
- Set the node data Values
- Connect the pointers

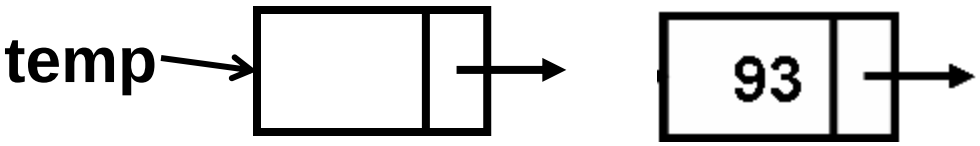
Insert rear



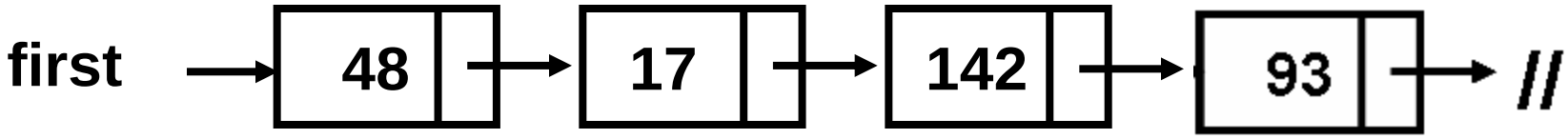
- Follow the previous steps and we get

Step 1

Step 2



Step 3



insert_rear()

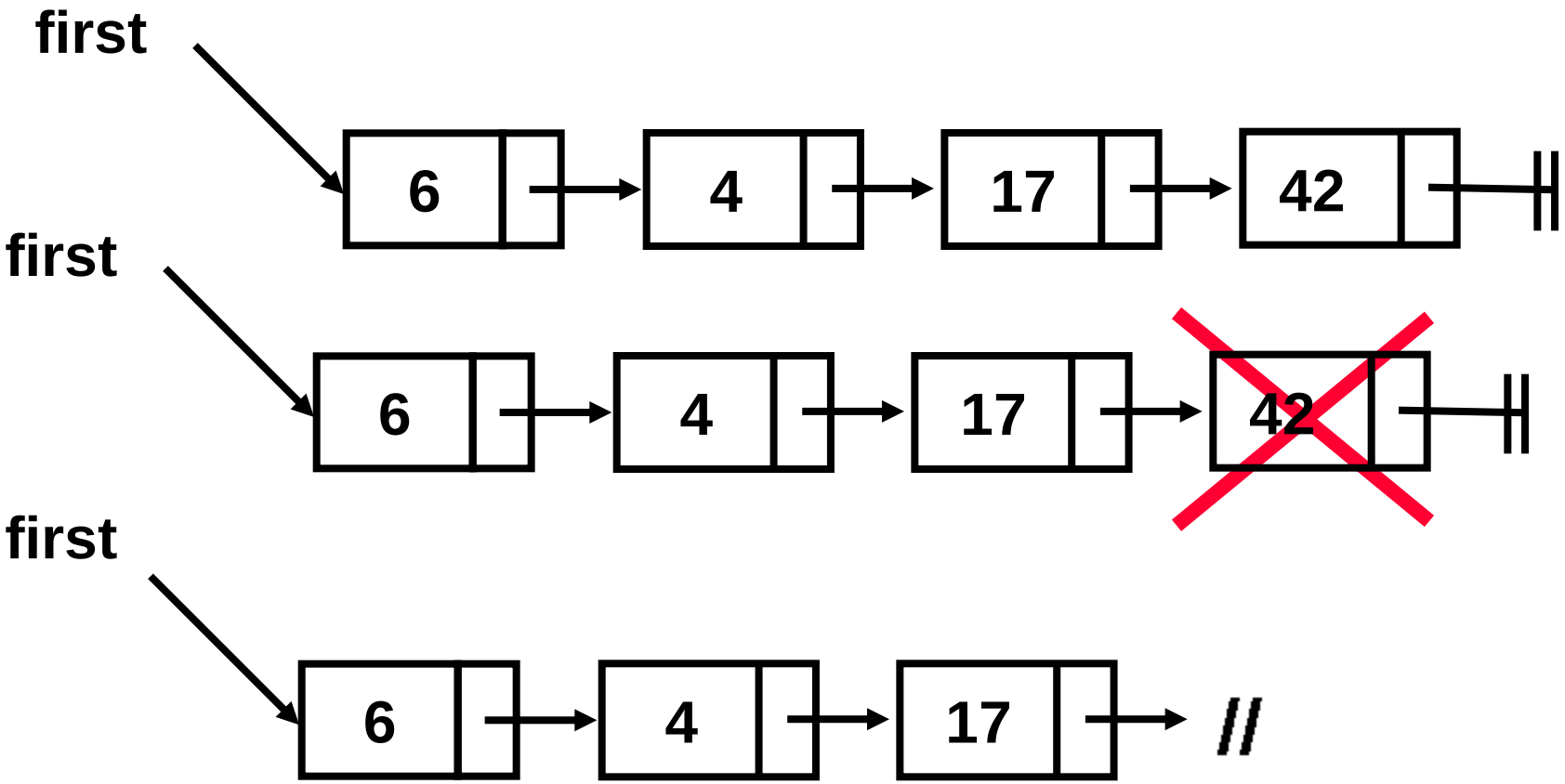
```
NODE insert_rear(int item, NODE first)
 {
    NODE temp;   / * Node to be inserted */
    NODE cur;    /* Will hold address of last node */
    temp = getnode();
    temp->data= item;
    temp->link= NULL;
    if (first  == NULL)
    {
       return temp;
    }
    cur = first;
    while (cur->link != NULL)
   {
    cur = cur->link;
    }
    cur->link = temp;
    return first;
}
```

Delete rear

Steps

- Break the pointer connection
- Set previous node pointer to NULL
- Delete the node

Delete rear



delete_rear() delete an item from rear end of list

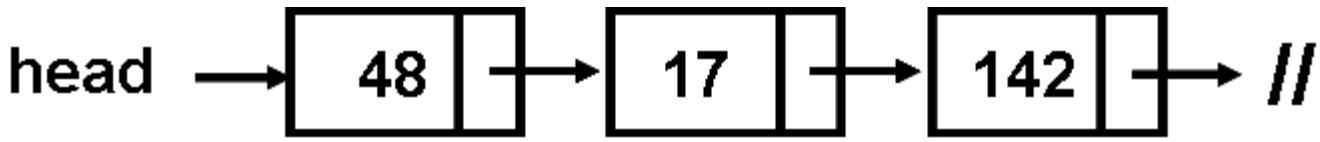
```
NODE delete_rear(NODE first)
{
   NODE cur, prev;   /* will hold address of last and previous nodes */
    if first == NULL)
   {
      printf("List is empty");
      return first;
   }
   if (first->link == NULL)  /* if one node is present delete it*/
   {
      printf("Item deleted is %d", first->data);
      freenode (first);
      first = NULL;
      return first;
    }
   cur  =  first;
   while (cur->link!= NULL)
  {
    prev = cur;
    cur = cur->link;
   }
   printf("Item deleted is %d", cur->data);
   freenode(cur);
   prev->link= NULL;
   return first;
}
```

Insertion in the middle

Steps:

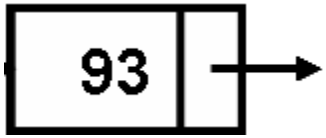
- Create a Node
- Set the node data Values
- Break pointer connection
- Re-connect the pointers

Insert Middle

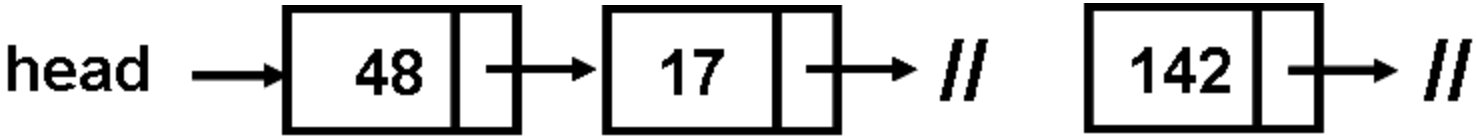


Step 1

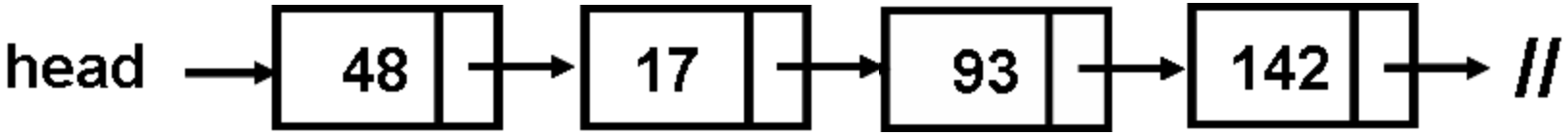
Step 2



Step 3



Step 4



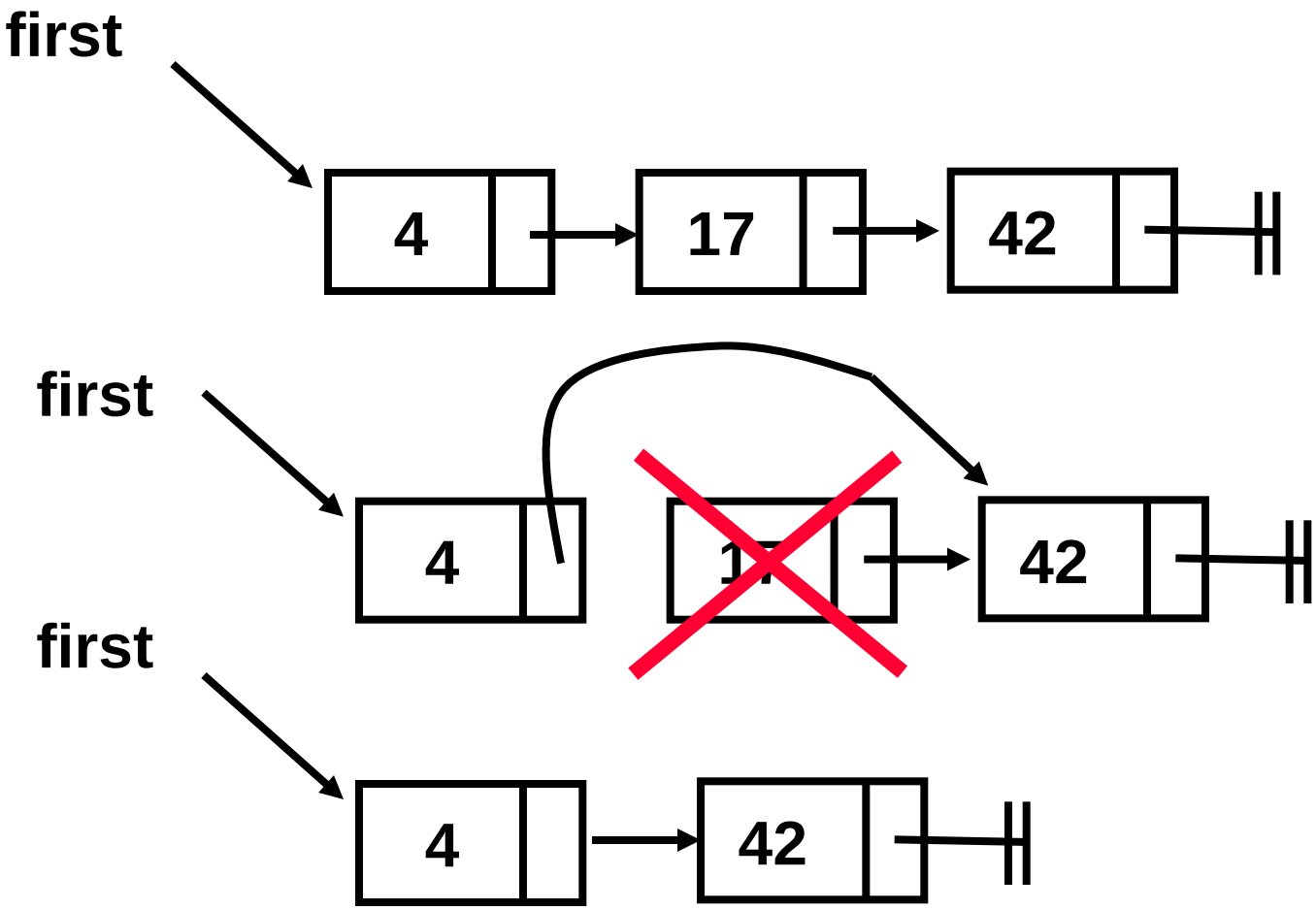
```
NODE insert_middle(int item,  int pos,  NODE first)
{
   NODE temp, prev,  cur;
   int count;
   temp = getnode();
   temp->data = item;
   temp->link= NULL;
   if (pos ==1)
      {
           temp->link = first;
           first = temp;
           return first;
      }
   count = 1;
   cur = first;
   while (cur->link != NULL && count != pos)
           {
               prev = cur;
               cur = cur->link;
               count++;
            }
   if (count = pos)
     {
          prev->link = temp;
          temp->link= cur;
          return first;
      }
   printf("Invalid position");
   return first;
}
```

Deleting from the Middle

Steps

- Set previous Node pointer to next node
- Break Node pointer connection
- Delete the node

Delete Middle



```
NODE delete_middle(int pos, NODE first)
{
   NODE prev,  cur;
   int count;
   if (first == NULL)
   {
      printf("List is empty");
      return first;
   }
  if (pos ==1)
      {
            cur = first;
            first = cur->link;
            freenode(cur);
            return first;
      }
   count = 1;
   cur = first;
   while (cur->link != NULL && count != pos)
            {
                prev = cur;
                cur = cur->link;
                count++;
             }
    prev->link = cur->link;
    freenode(cur);
   return first;
}
```

Linked Stacks and Queues

1. Implementation of Stacks using Linked List:

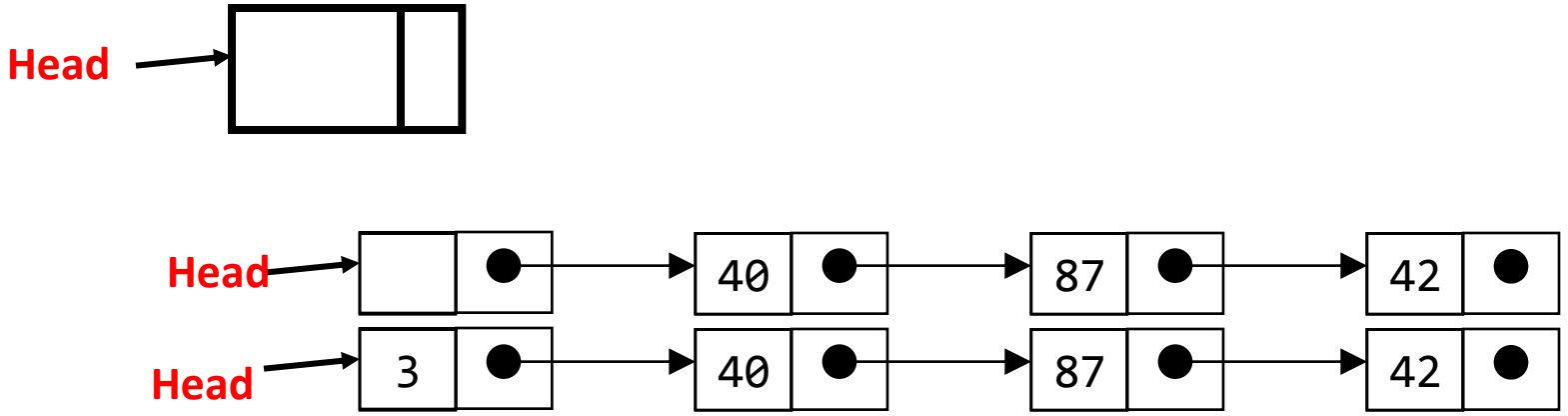
Use **insert_front()** and **delete_front()** functions

2. Implementation of Queues using Linked List:

Use **delete_front()** and **insert_rear()** functions

Linked List with Header Nodes

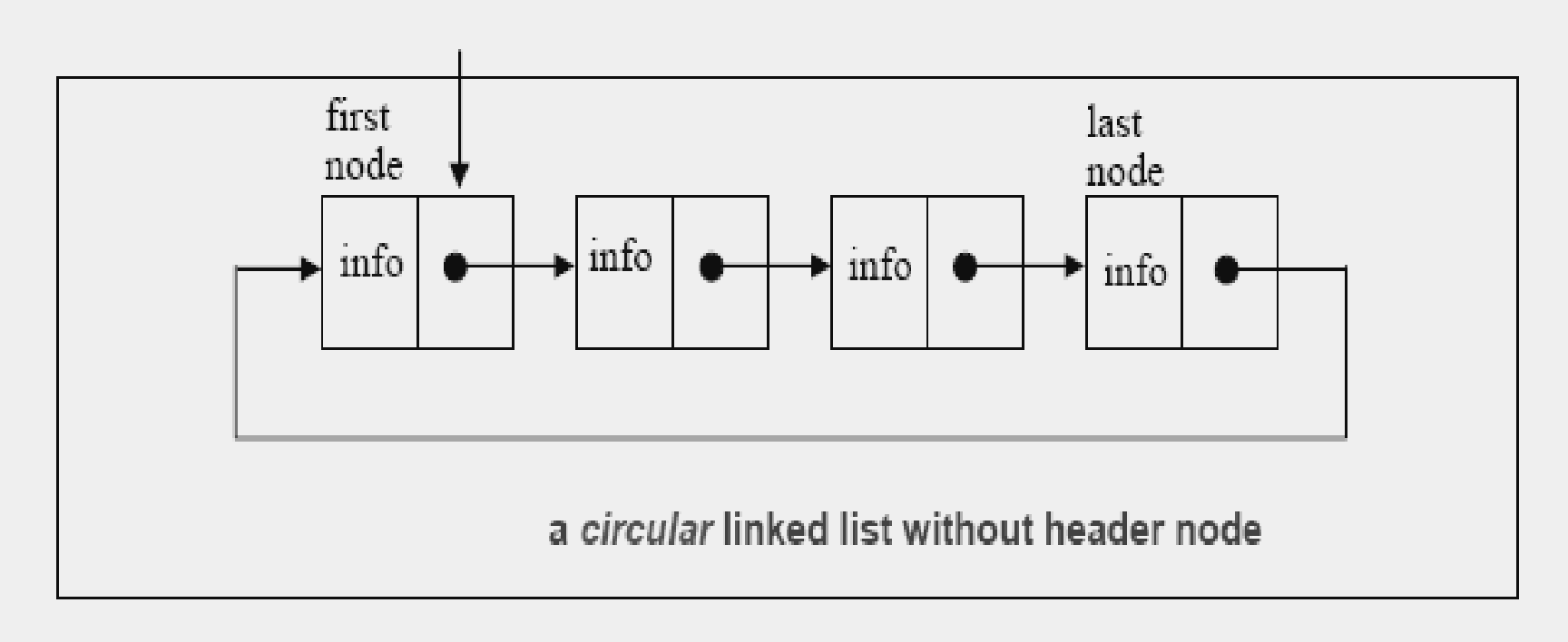
Sometimes it is desirable to keep an extra node at the front of a list, such a node does not represent an item in the list and is called a header node or a list header. The **info** field of such node usually does not contain any information. Sometimes, **info** field of header node contains the number of nodes in the list.



List with Header node

Circular Linked list

A linked list where the link field of last node contains address of first node is called a circular linked list .It is represented as follows.



Advantages of Circular linked list over single linked list

1. **Traversing is simple.**

   In a single linked list, starting from the middle node we cannot visit all nodes. However in circular linked list one can traverse the entire linked list starting from any node.

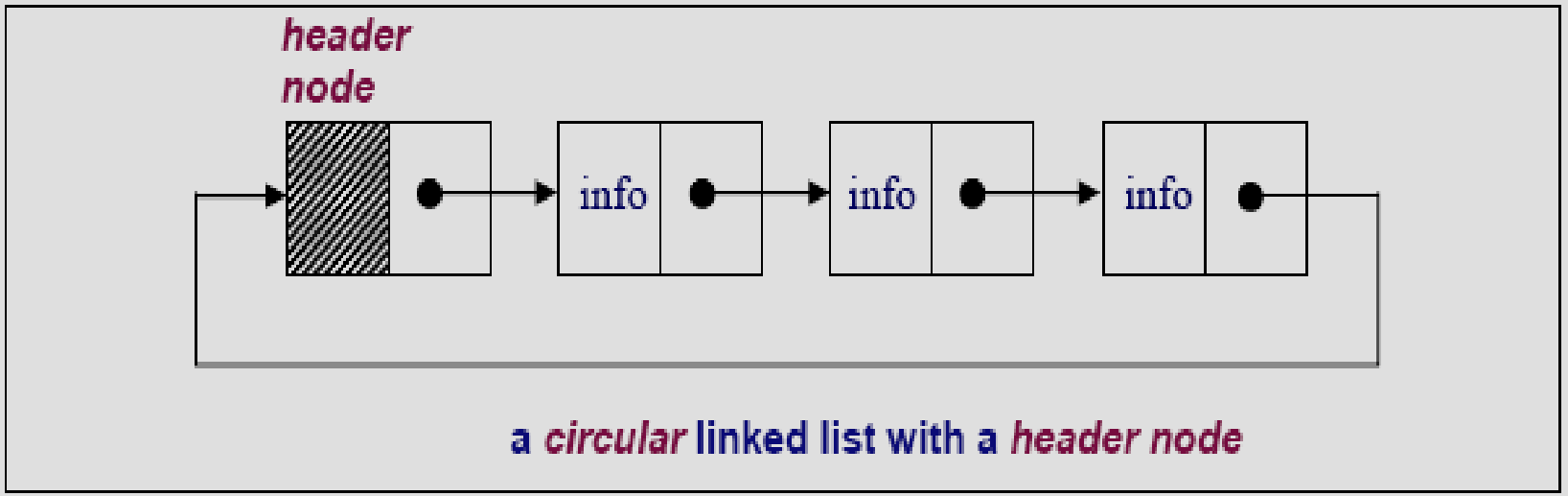
2. **Deletion requires less number of inputs in circular linked list over single linked list.**

   In single linked list if we wish to delete a node, we require the address of that node and also we require the address of first node. However, in circular linked list, if we want to delete a node we require the address of that node. However, we do not require the address of first node.

Disadvantages of circular linked list

1. If one is not careful while processing a circular linked list, then we can end up in an infinite loop.
2. In circular linked list it is not possible to distinguish last node from other nodes since its link field also contain address .

To over come this we use a special node called head node at beginning of the linked list . The head node link field points to first node and link field of last node contain address of head node. The data field of head node contains the number of nodes in the list.



Circular Linked List : Insert_front()

```
NODE insert_front(int  item, NODE head)
 {
    NODE temp; / * Node to be inserted */
    temp = getnode();   /* Get a node */
    temp -> data = item;
    temp -> link = head->link;
    head->link = temp;
    head->data += 1;
    return head;
}
```

Circular Linked List : Insert_rear()

```
NODE insert_rear(int item, NODE head)
 {
    NODE temp, cur;   / * Node to be inserted */
    temp = getnode();
    temp->data= item;
    cur = head->link;
    while (cur->link != head)
   {
    cur = cur->link;
    }
    cur->link = temp;
    temp->link = head;
    head->data += 1;
    return head;
}
```

Circular Linked List : delete_front()

```
NODE delete_front(NODE head)
 {
    NODE temp;
    if (head -> link == head)
    {
       printf("List is empty");
       return head;
    }
    temp = head->link;
    head = temp -> link;
    freenode (temp);
    head->data -= 1;
    return head;
}
```

Circular Linked List : delete_rear()

```
NODE delete_rear(NODE head)
 {
    NODE cur, prev;   /* will hold address of last and previous nodes */
     if head->link == head)
    {
       printf("List is empty");
       return head;
    }
    prev = head;
    cur  =  head->link;
    while (cur->link!= head)
   {
     prev = cur;
     cur = cur->link;
    }
    prev->link= cur->link;
    freenode(cur);
    head->data  -= 1;
    return head;
}
```

Circular Linked List : insert_middle()

```
NODE insert_middle(int item,  int pos,  NODE head)
{
   NODE temp, prev,  cur;
   int i;
   temp = getnode();
   temp->data = item;
   if (pos ==1)
      {
           temp->link =  head->link;
           head ->link = temp;
           return head;
      }
   prev  = head;
   cur = head-> link;
   for (i = 1; i < pos; i++)
           {
               prev = cur;
               cur = cur->link;
            }
     prev->link = temp;
     temp->link= cur;
     head -> data  +=  1;
     return head;
}
```

Circular Linked List : delete_middle()

```
NODE delete_middle(int pos, NODE head)
{
   NODE prev,  cur;
   int i;
   if (head->link == head)
   {
      printf("List is empty");
      return head;
   }
    if (pos ==1)
      {
            cur = head->link;
            head = cur->link;
            freenode(cur);
            return head;
      }
   prev = head;
   cur = head->link;
   for (i = 1; i < pos; i++)
           {
               prev = cur;
               cur = cur->link;
            }
   prev->link = cur->link;
   freenode(cur);
   head->data -= 1;
   return head;
}
```

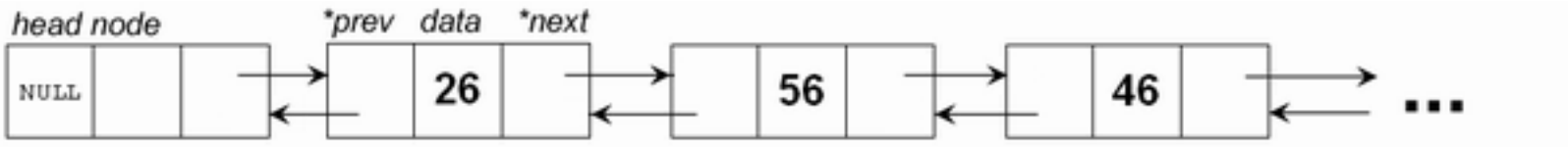
Union and Intersection

- Intersection is a set of elements that are common in both lists while union is a set of all unique elements in both the lists

Disadvantages of circular linked list

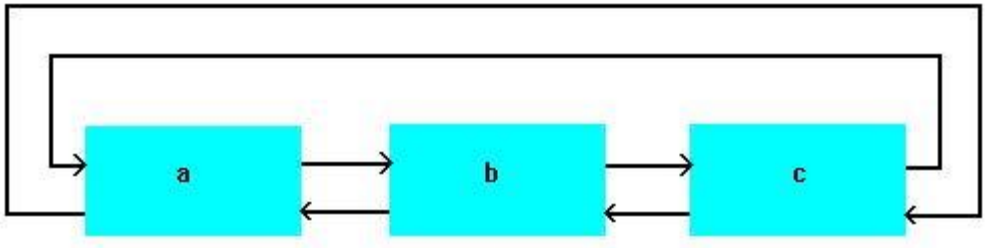
- One cannot traverse backward.

Double Linked Lists



Double linked list may be either linear or circular and may or may not contain a header node

Circular double Linked Lists



Double Linked List Using Dynamic Variables

Each node in the double Linked List has three fields namely data field, a left pointer, and a right pointer .

```
struct  node {
                int data;
                struct node *llink,*rlink;
                } ;
typedef struct   node *NODE
```

NODE is defined using **typedef** as pointer to **struct node**

Double Linked List : Insert_front()

```
NODE insert_front(int  item, NODE head)
 {
    NODE temp, cur;
    temp = getnode();   /* Get a node */
    temp -> data = item;
    cur = head -> rlink;
    head -> rlink = temp;
    temp -> llink = head;
    temp -> rlink = cur;
    cur -> llink = temp;
    return head;
}
```

Double Linked List : Insert_rear()

```
NODE insert_rear(int item, NODE head)
 {
    NODE temp, cur;   / * Node to be inserted */
    temp = getnode();
    temp->data= item;
    temp->rlink= NULL;
    cur = head->rlink;
    while (cur->rlink != NULL)
   {
    cur = cur->rlink;
    }
    cur->rlink = temp;
    temp->llink = cur;
    return head;
}
```

Double Linked List: delete_front()

```
NODE delete_front(NODE head)
 {
    NODE cur, temp;
    if (head ->rlink  == NULL)
    {
       printf("List is empty");
       return head;
    }
    temp = head -> rlink;
    cur = temp -> rlink;
    head -> rlink = cur;
    cur -> llink = head;
    freenode (temp);
    return head;
}
```

Double Linked List: delete_rear()

```
NODE delete_rear(NODE head)
 {
    NODE cur, prev;
    if head -> rlink == NULL)
    {
       printf("List is empty");
       return head;
    }
    prev  =  head;
    cur = head -> rlink
    while (cur->rlink!= NULL)
   {
     prev = cur;
     cur = cur->rlink;
    }
    prev->rlink= NULL;
    freenode(cur);
    return head;
}
```

Advantages Linked List

The Linked List advantages are collected because of the array disadvantages, array disadvantages are:

1. Array Size
2. Memory allocation
3. Insertion and Deletion

Polynomials

- Representing Polynomials As Singly Linked Lists
  - The manipulation of symbolic polynomials, has a classic example of list processing.
  - In general, we want to represent the polynomial:

$$P(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$$

    - Where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that

      $e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 \geqq 0$.
  - We will represent each term as a node containing **coefficient** and **exponent** fields, as well as a **pointer** to the next term.

Linked-list implementation of polynomials

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

Structure of the polynomial node

coef (cf)	expon (px)	link

Polynomial representation

```
struct  node {
                int cf;
                int px;
                struct node *link;
                } ;
typedef struct   node *NODE
```

Adding Polynomials

- 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.

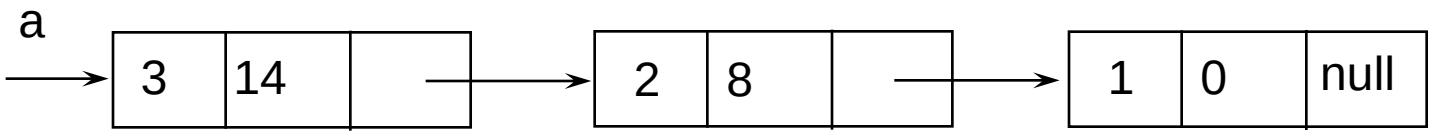
[go to next node]

Case 3: exponent of p1 = exponent of p2

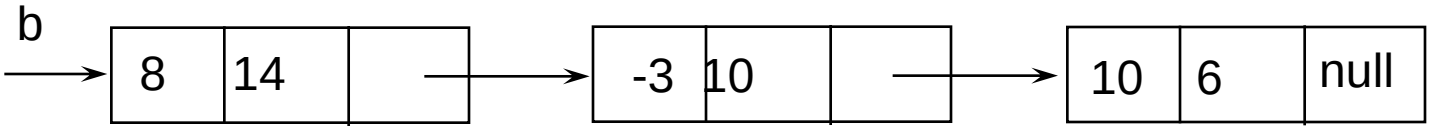
- Create a new node in p3 with the same exponent and with the sum of the coefficients of p1 and p2.

Examples

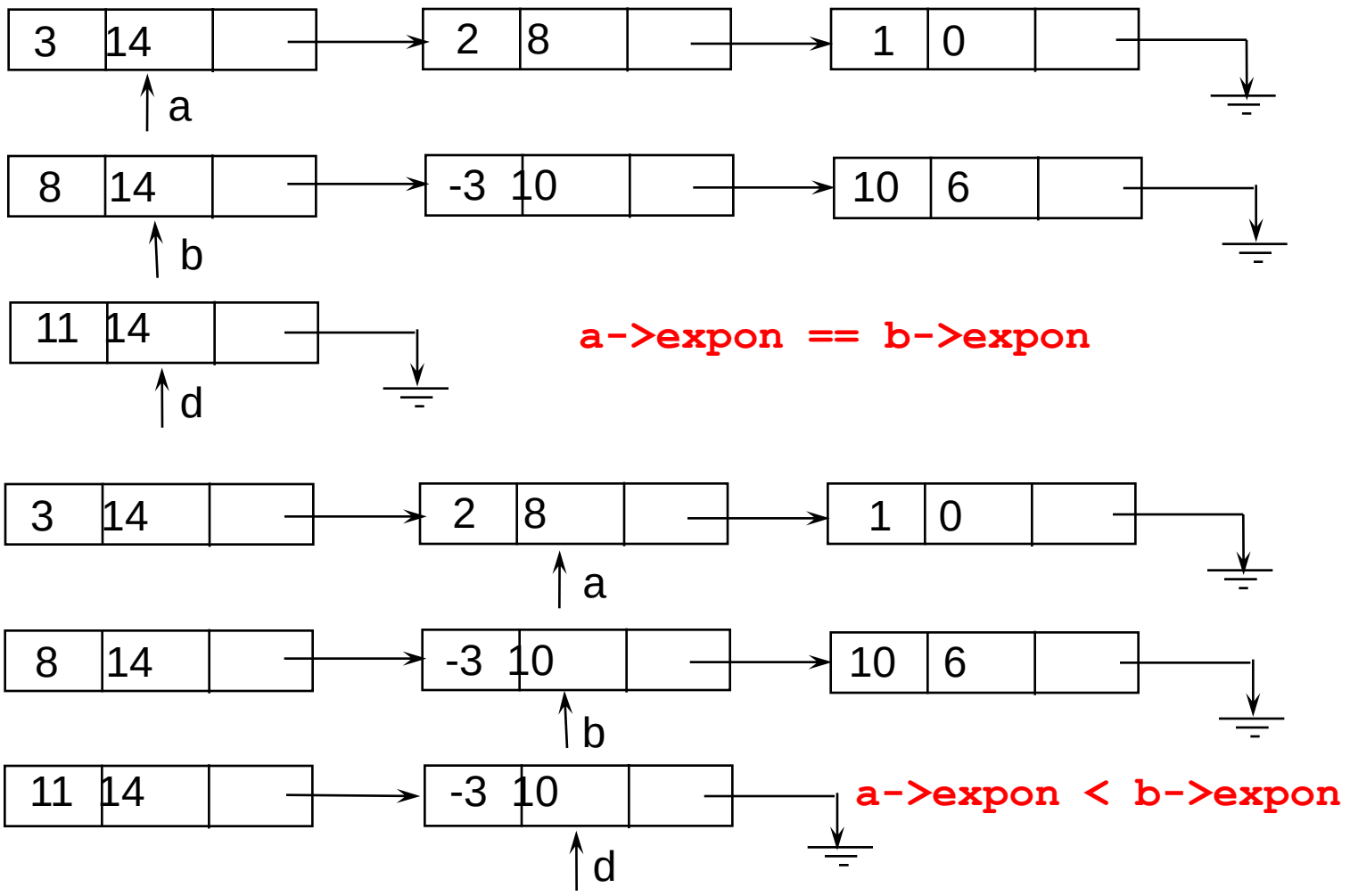
$$a = 3x^{14} + 2x^{8} + 1$$



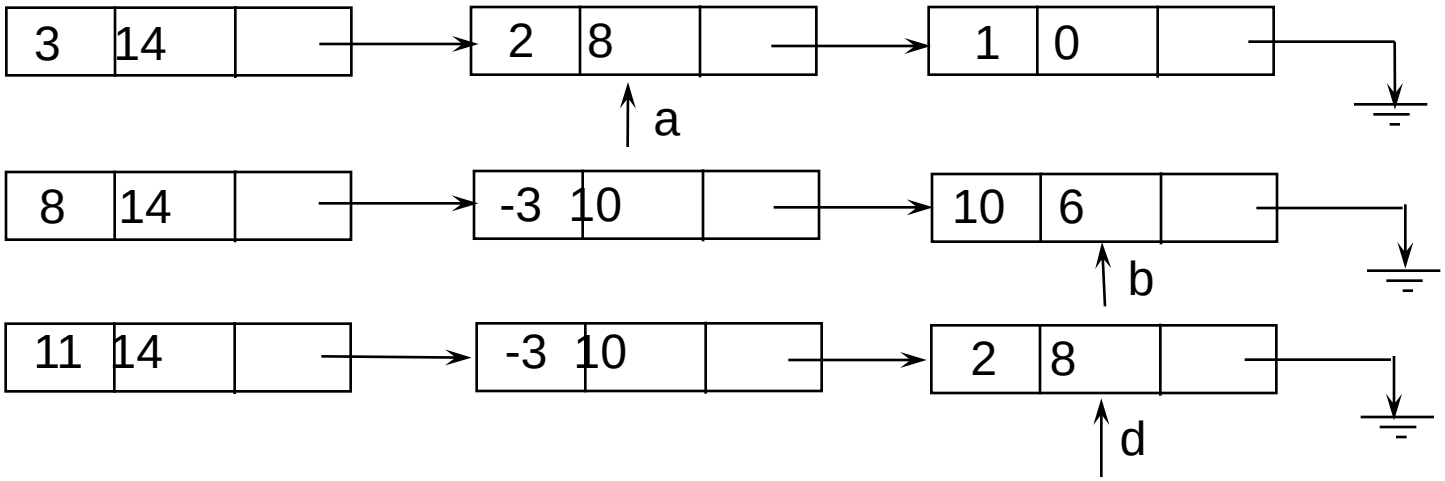
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



Adding Polynomials



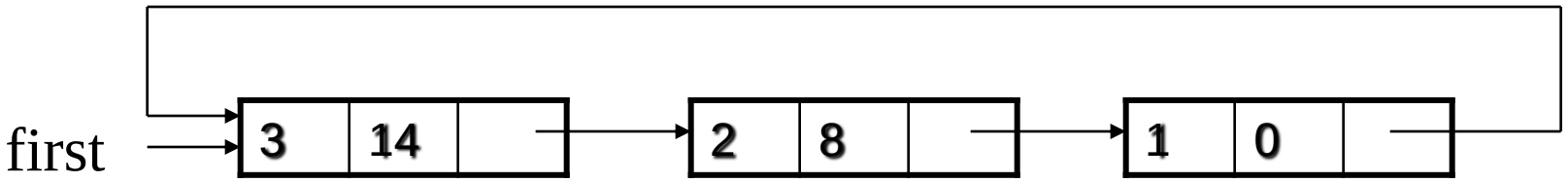
Adding Polynomials (*Continued*)




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

Polynomial: Circularly Linked Lists

- Circular Linked list
  - The link field of the last node points to the first node in the list.
- Example
  - Represent a polynomial *ptr* = $3x^{14}+2x^{8}+1$ as a circularly linked list.



Polynomial: Circularly Linked Lists

- We must handle the **zero polynomial** as a special case. To avoid it, we introduce a ***head node*** into each polynomial
  - each polynomial, zero or nonzero, contains one additional node.
  - The *expo*n and *coef* fields of this node are irrelevant.

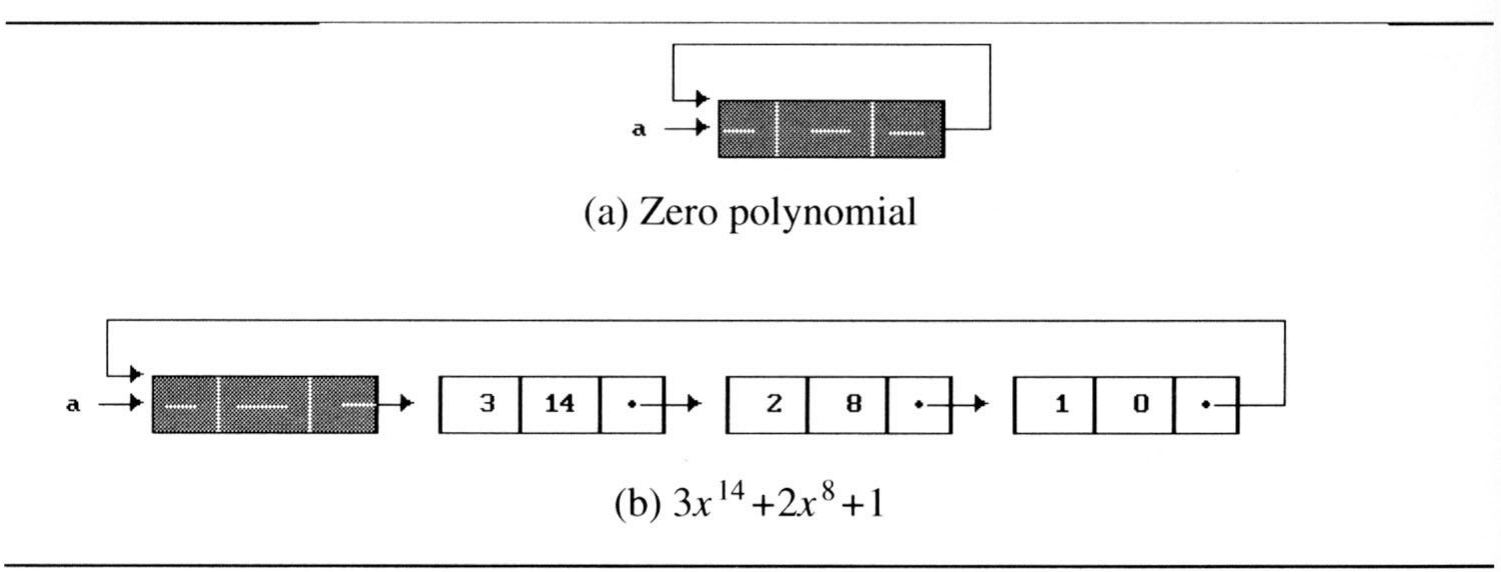


Figure 4.15: Polynomial representations

Polynomial: insert_rear()

```
NODE insert_rear(int cf, int x, NODE head)
 {
    NODE temp, cur;
    temp = getnode();
    temp -> cf = cf;
    temp -> px = x;
    cur = head->link;
    while (cur->link != head)
   {
    cur = cur->link;
    }
    cur->link = temp;
    temp->link = head;
    return head;
}
```

Polynomial Addition

```
struct  node {
                int cf;
                int px;
                int flag;
                struct node *link;
                } ;
typedef struct   node *NODE
```

In polynomial addition: flag is used in the second polynomial to check corresponding term is present in 1st polynomial. If flag is 0, it means that the corresponding term is not present in 1st polynomial.

Polynomial: insert_rear()

```
NODE insert_rear(int cf, int x, NODE head)
 {
    NODE temp, cur;
    temp = getnode();
    temp -> cf = cf;
    temp -> px = x;
    temp -> flag = 0;
    cur = head->link;
    while (cur->link != head)
   {
    cur = cur->link;
    }
    cur->link = temp;
    temp->link = head;
    return head;
}
```

Polynomial Addition

```
NODE add_poly(NODE h1, NODE h2, NODE h3)
{
    NODE p1, p2;
    int x1, x2, cf1, cf2, cf;
    p1 = h1 -> link;
    while (p1 != h1)
    {
        x1 = p1 -> px;       /* obtain term of 1st polynomial */
        cf1 = p1 -> cf;
        p2 = h2->l ink;
        while (p2 != h2)
        {
            x2 = p2 -> px;    /* search this term in 2nd polynomial */
            cf2 = p2 -> cf;
            if (x1 == x2) break;
            p2 = p2 -> link;
        }
        if ( p2 != h2)  /* If term of 1st polynomial is present */
        {
             cf = cf1 + cf2;  /* add coefficients */
             p2 ->flag = 1;
            if (cf != 0)
                h3 = insert_rear(cf, x1, h3)   /* insert into resultant polynomial */
         }
        else
             h3 = insert_rear(cf1, x1, h3);    /* insert  1st term into resultant polynomial */
        p1 = p1 -> link;      /* obtain next term of 1st polynomial */
    }
    continued next page
```

Polynomial Addition Continued

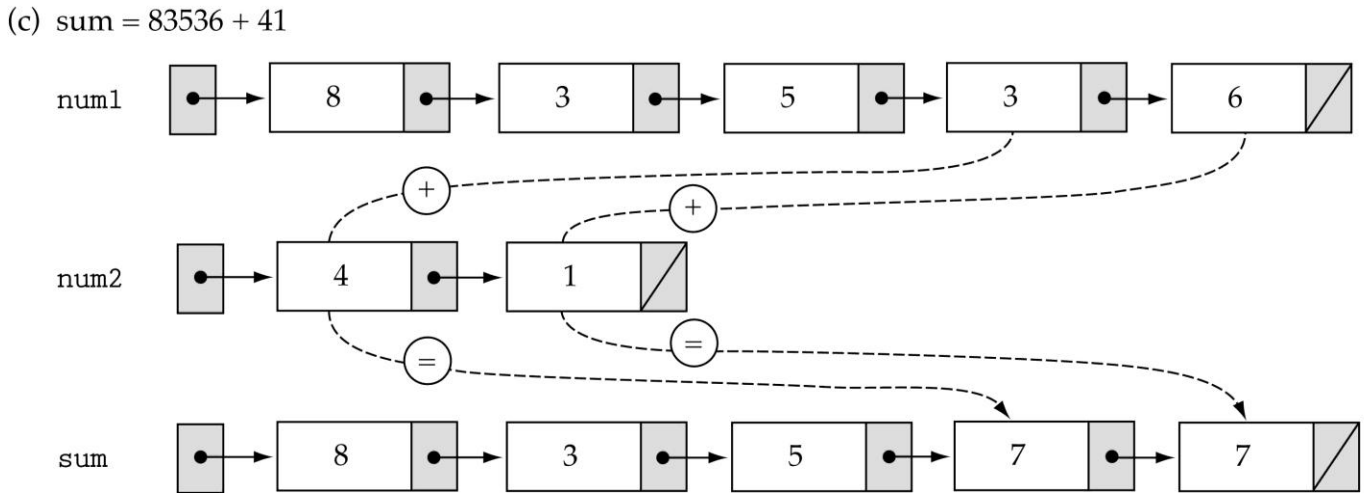
```
    p2 = h2 ->link;  /* add remaining  terms of second polynomial to the result */
    while(p2 != h2)
    {
        if (p2 -> flag == 0)
        {
            h3 = insert_rear(p2 -> cf, p2 -> px, h3);
        }
        p2 = p2 -> link;
    }
    return h3;
}
```

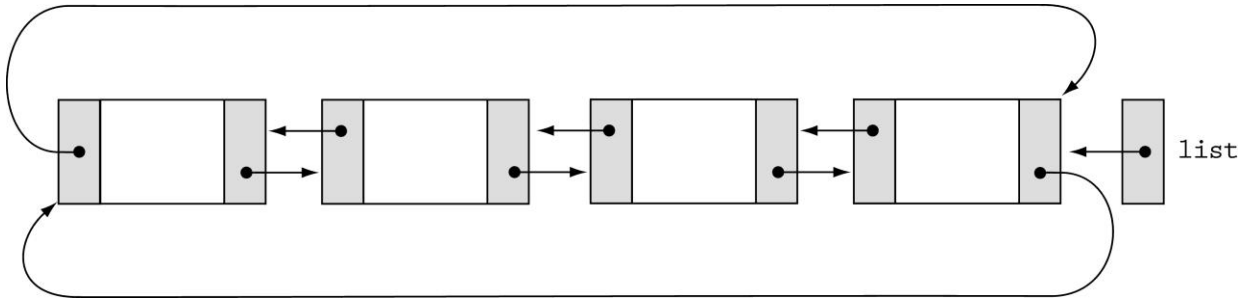
Case Study: Implementing a large integer

- The range of integer values varies from one computer to another
- For *long* integers, the range is

  [-2,147,483,648 to 2,147,483,647]
- How can we manipulate larger integers?

Case Study: Implementing a large integer (cont.)





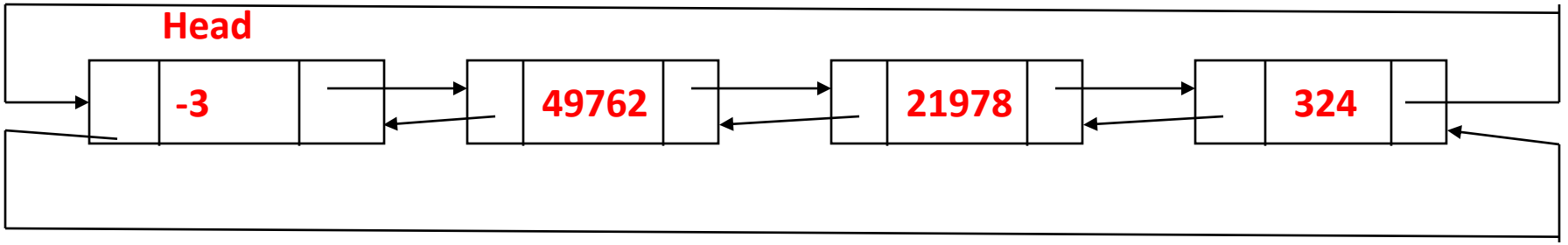
Addition of Long positive integers

```
struct  node {
                long  int data;
                struct node *llink, *rlink;
                } ;
typedef struct   node *NODE
```

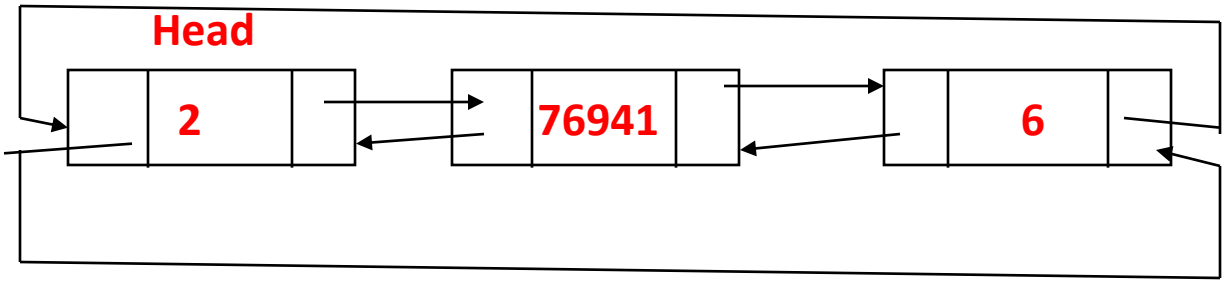
To add two long positive integers, their digits are traversed from right to left, corresponding digits and a possible carry from the previous digits' sum are added.

The long integers are represented by storing their digits from right to left in the list so that first node on the list contains the least significant digit (rightmost) and the last node contains the most significant digit (leftmost). However, to save space, we keep five digits in each node.

Long Integers as circular doubly linked lists



The integer -3242197849672



The integer 676941