UNIT-2

Prepared by,
P. Amaranatha Reddy
Asst. Prof., CSE dept,
VIGNAN'S University.

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Representing Stack with Linked List.
- Representing Queue with Linked List.

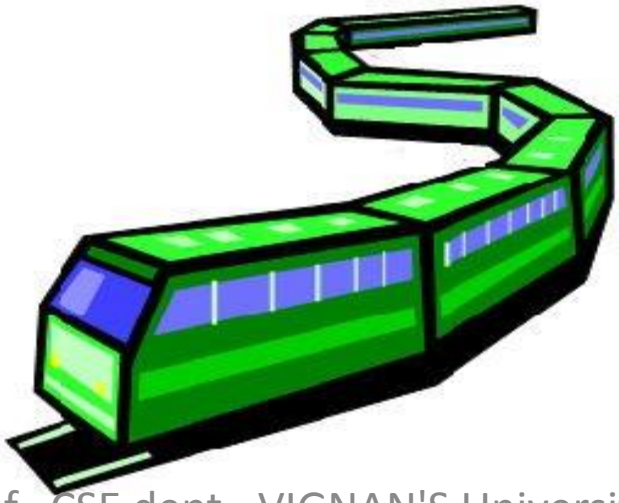
- In **array**, elements are stored in consecutive memory locations.
- To occupy the adjacent space, block of memory that is required for the array should be allocated before hand.
- Once memory is allocated, it cannot be extended any more. So that array is called the **static data structure**.
- Wastage of memory is more in arrays.
- Array has fixed size
- But, **Linked list** is a dynamic data structure, it is able to grow in size as needed.

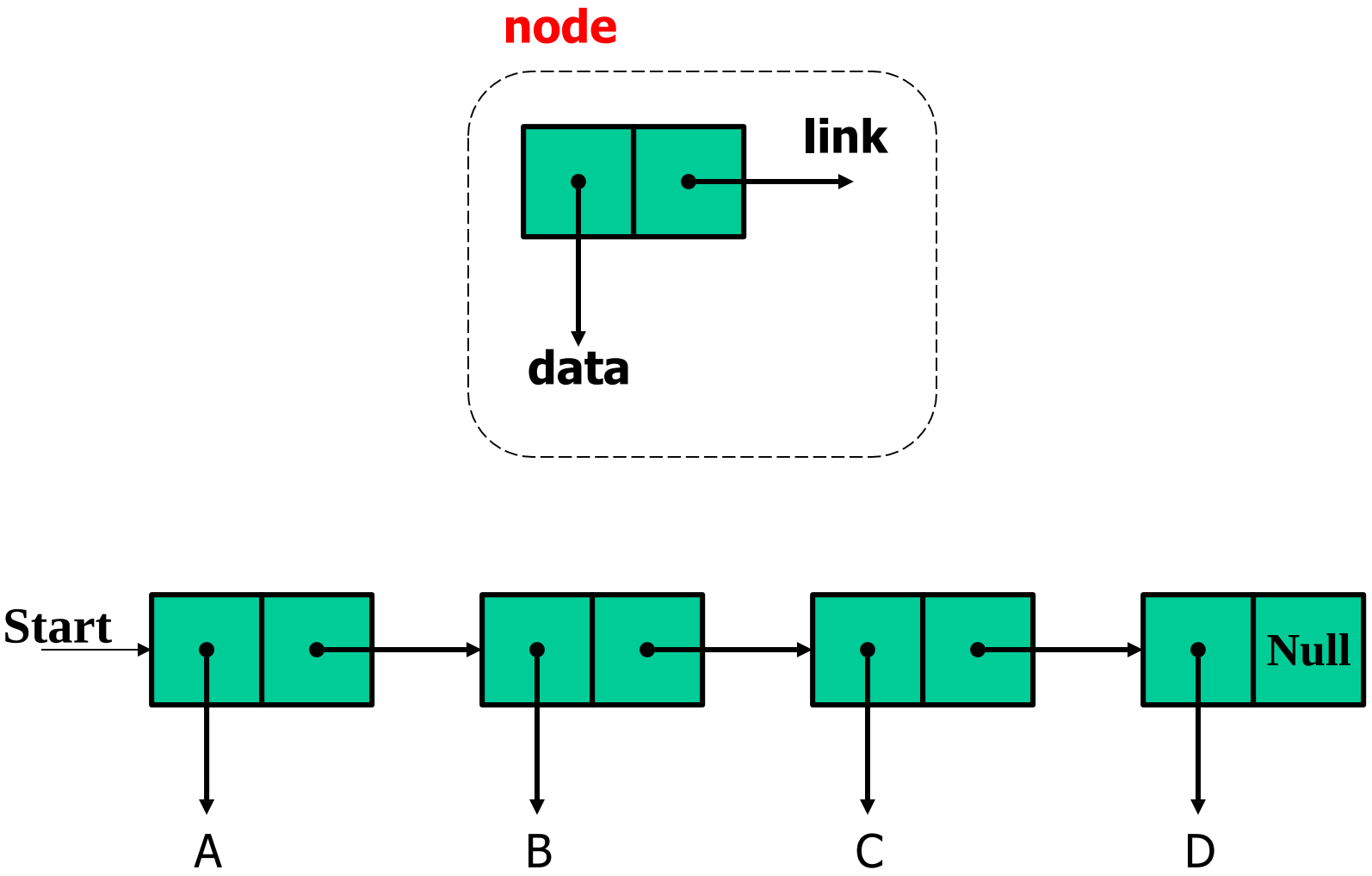
What is Linked List?

- A linked list is a linear collection of homogeneous data elements, called **nodes,** where linear order is maintained by means of links or pointers.

- Each node has two parts:
  - The first part contains the data (information of the element) and
  - The second part contains the address of the next node (link /next pointer field) in the list.

- Data part of the link can be an integer, a character, a String or an object of any kind.




node
link
data
Start
Null
A
B
C
D

Linked Lists

- **Linked list**
  - Linear collection of self-referential structures, called *nodes*, connected by pointer *links.*
  - Accessed via a pointer to the first node of the list.
  - Subsequent nodes are accessed via the link-pointer member stored in each node.
  - Link pointer in the **last node is set to null** to mark the end of list.
  - Data stored dynamically – each node is created as necessary.
  - Length of a list can increase or decrease.
  - Becomes full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

Types of linked lists

- **Singly linked list**
  - Begins with a pointer to the first node
  - Terminates with a null pointer
  - Only traversed in one direction

- **Circular, singly linked list**
  - Pointer in the last node points back to the first node

- **Doubly linked list**
  - Two “start pointers”- first element and last element
  - Each node has a forward pointer and a backward pointer
  - Allows traversals both forwards and backwards

- **Circular, doubly linked list**
  - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

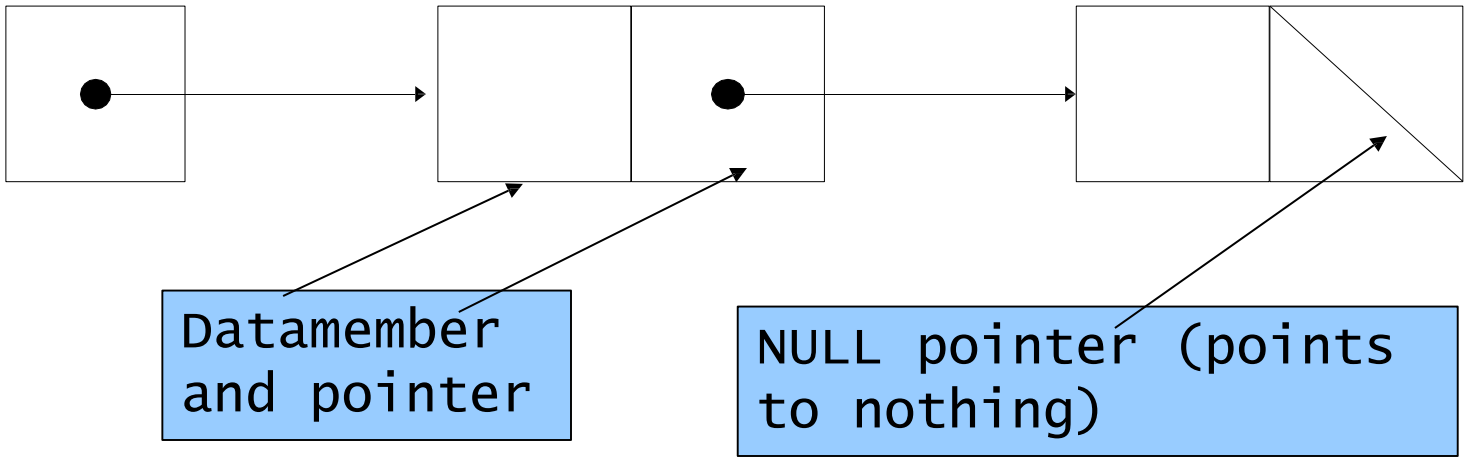
Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
- **malloc**
  - Takes number of bytes to allocate
    - Use **sizeof** to determine the size of an object
  - Returns pointer of type **void ***
    - A **void *** pointer may be assigned to any pointer
    - If no memory available, returns **NULL**
  - **newPtr = malloc(sizeof(struct node));**
- **free**
  - Deallocates memory allocated by **malloc**
  - Takes a pointer as an argument
  - **free (newPtr);**

Self-Referential Structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a **NULL** pointer (**0**)

- Two self-referential structure objects linked together



Singly linked list operations

Insertion:

- **Insertion of a node at the front**
- **Insertion of a node at any position in the list**
- **Insertion of a node at the end**

Deletion:

- **Deletion at front**
- **Deletion at any position**
- **Deletion at end**

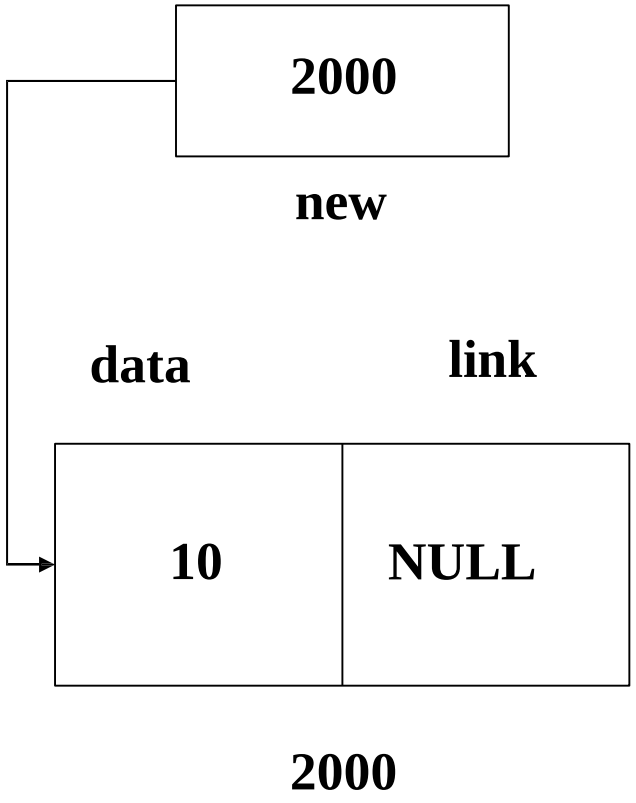
Display:

- **Displaying/Traversing the elements of a list**

Singly linked lists

Node Structure

```
struct node
{
      int data;
      struct node *link;
}*new, *ptr, *header, *ptr1;
```

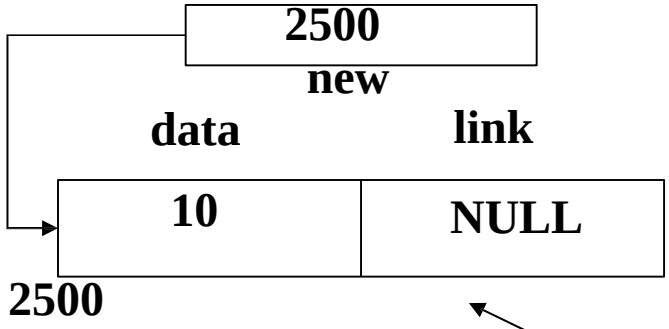


Creating a node

```
new = malloc (sizeof(struct node));
new -> data = 10;
new -> link = NULL;
```

Inserting a node at the beginning

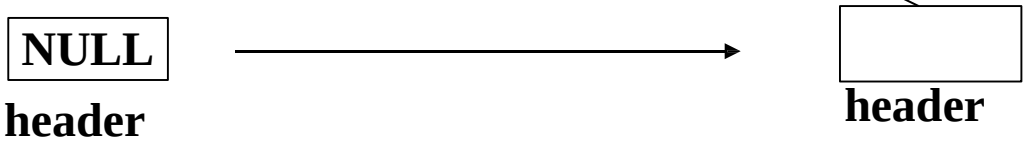
Create a node that is to be inserted



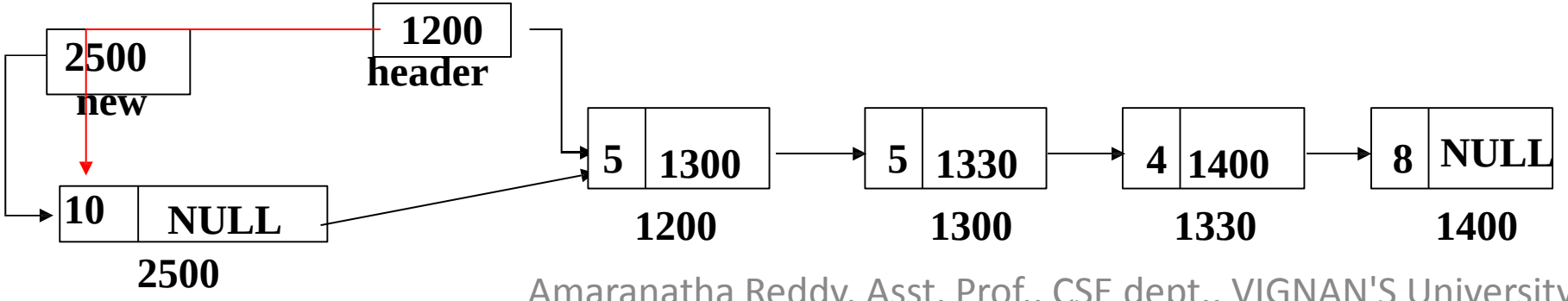
Algorithm:

1. Create a new node.
2. if (header = = NULL)
3. new -> link = NULL;
4. header = new;
5. else
6. {
7. new -> link = header;
8. header = new;
9. }

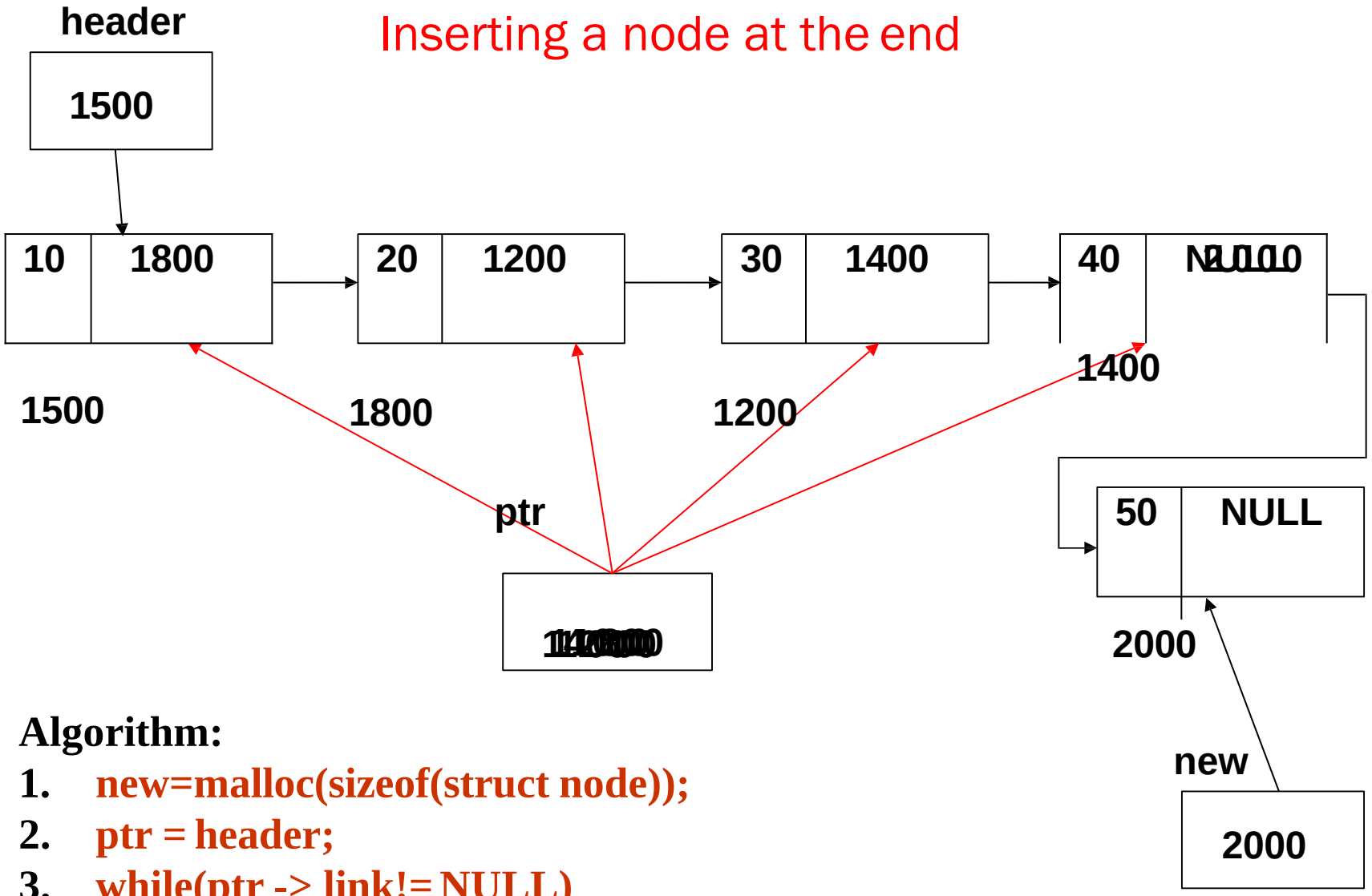
If the list is empty



If the list is not empty



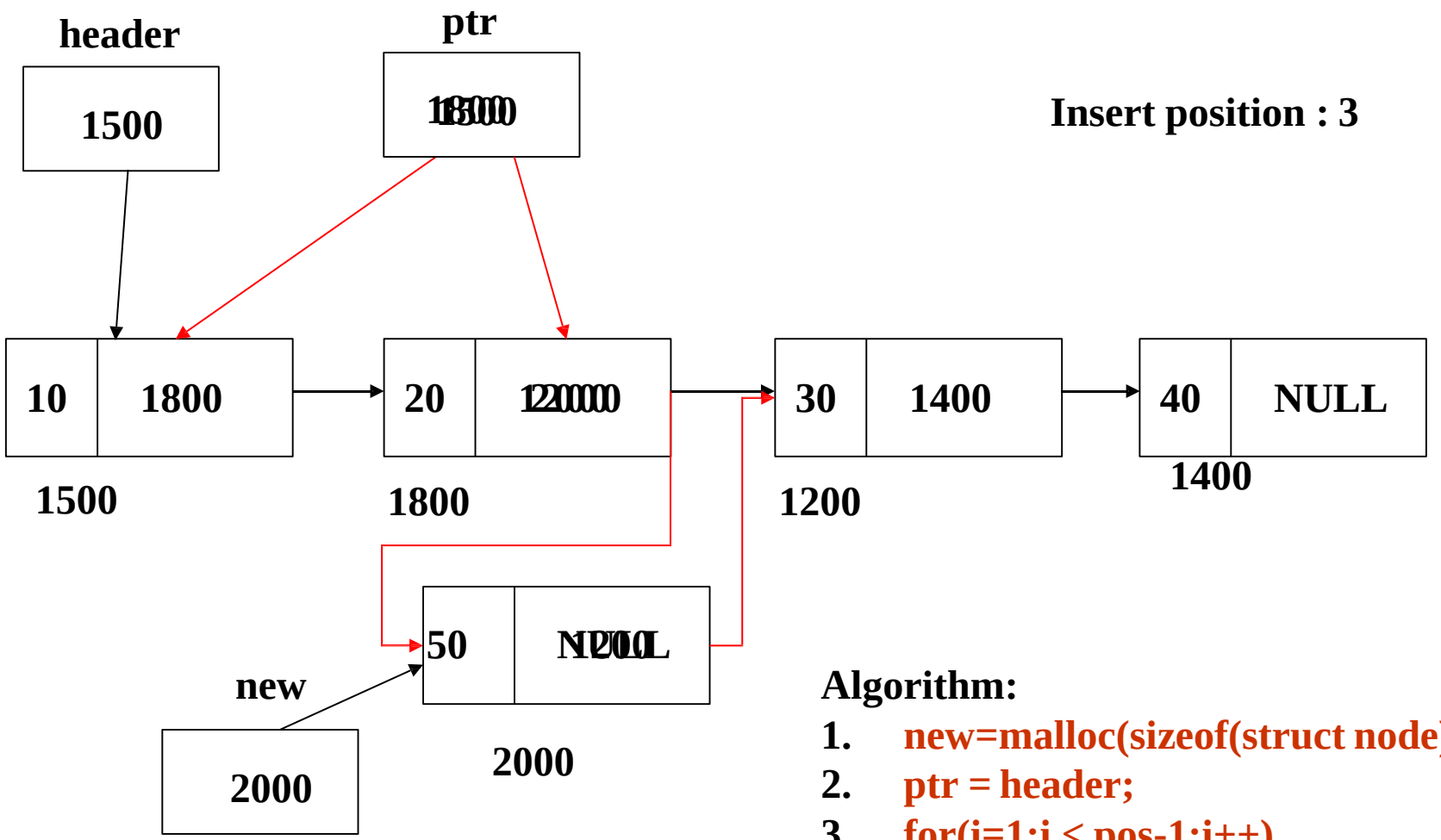
Inserting a node at the end



Algorithm:

1. `new=malloc(sizeof(struct node));`
2. `ptr = header;`
3. `while(ptr -> link!= NULL)`
4. `ptr = ptr -> link;`
5. `new -> link = NULL;`
6. `ptr -> link = new;`

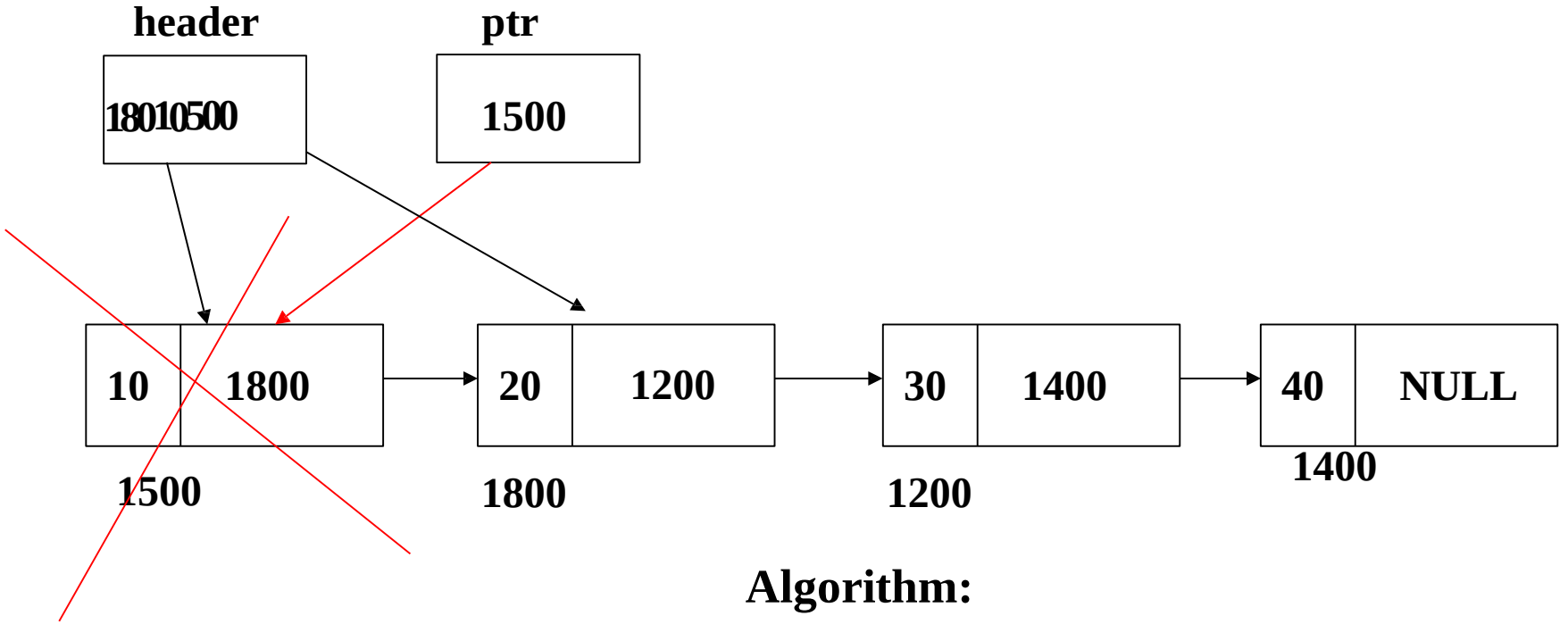
Inserting a node at the given position



Algorithm:

1. `new=malloc(sizeof(struct node));`
2. `ptr = header;`
3. `for(i=1;i < pos-1;i++)`
4. `ptr = ptr -> link;`
5. `new -> link = ptr -> link;`
6. `ptr -> link = new;`

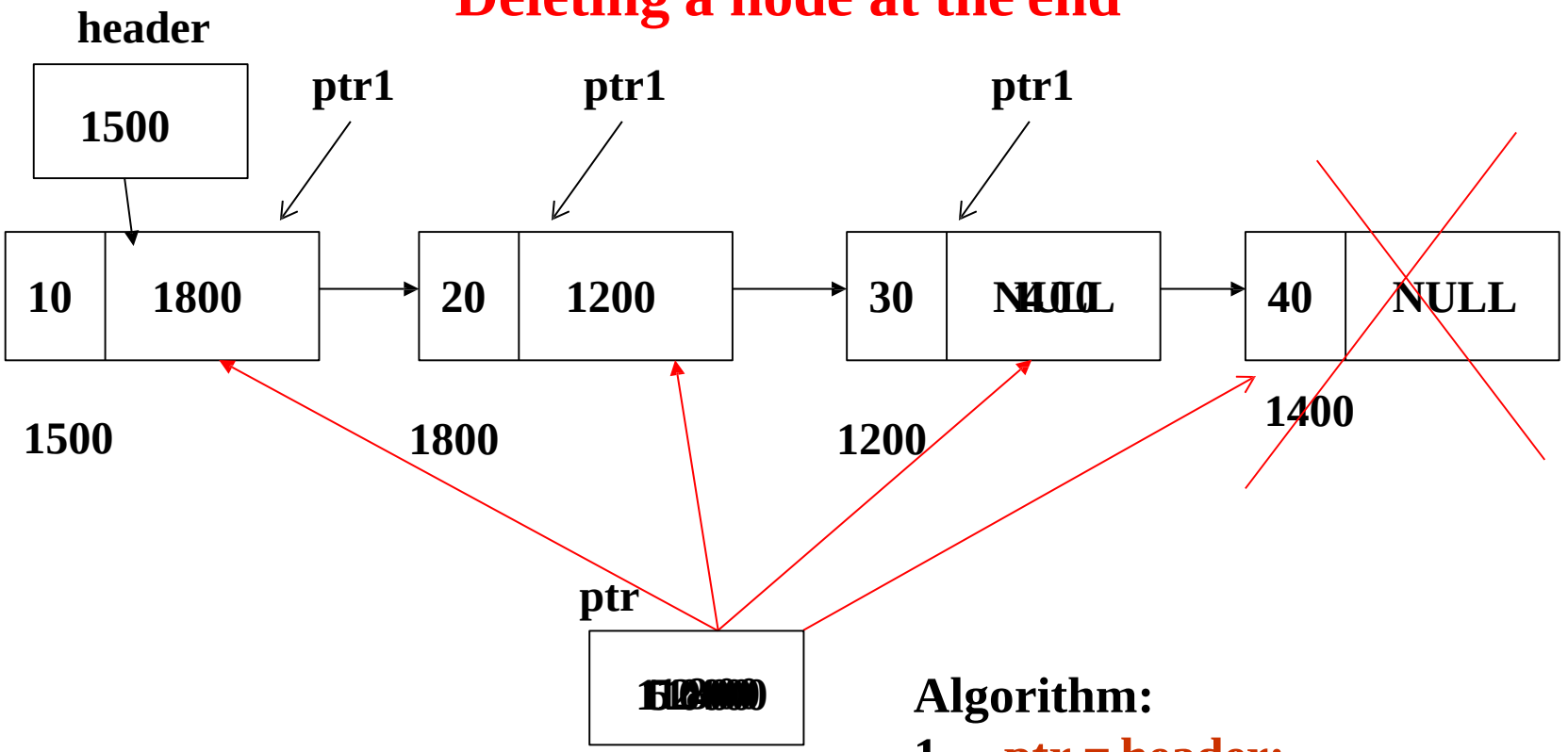
Deleting a node at the beginning



Algorithm:

```
1.  if (header = = NULL)
2.        print  "List is Empty";
3.  else
4.  {
5.        ptr = header;
6.        header = header -> link;
7.        free(ptr);
8.  }
```

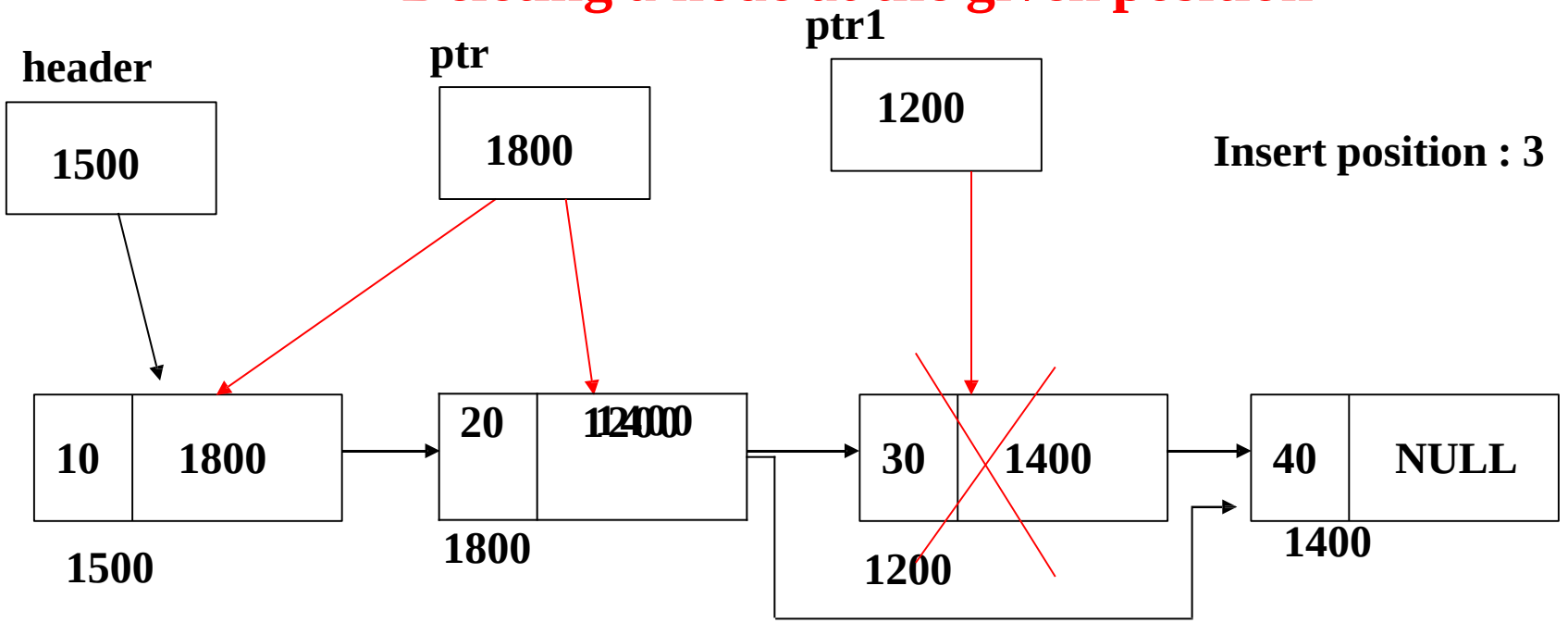
Deleting a node at the end



Algorithm:

1. ptr = header;
2. while(ptr -> link != NULL)
3. ptr1=ptr;
4. ptr = ptr -> link;
5. ptr1 -> link = NULL;
6. free(ptr);

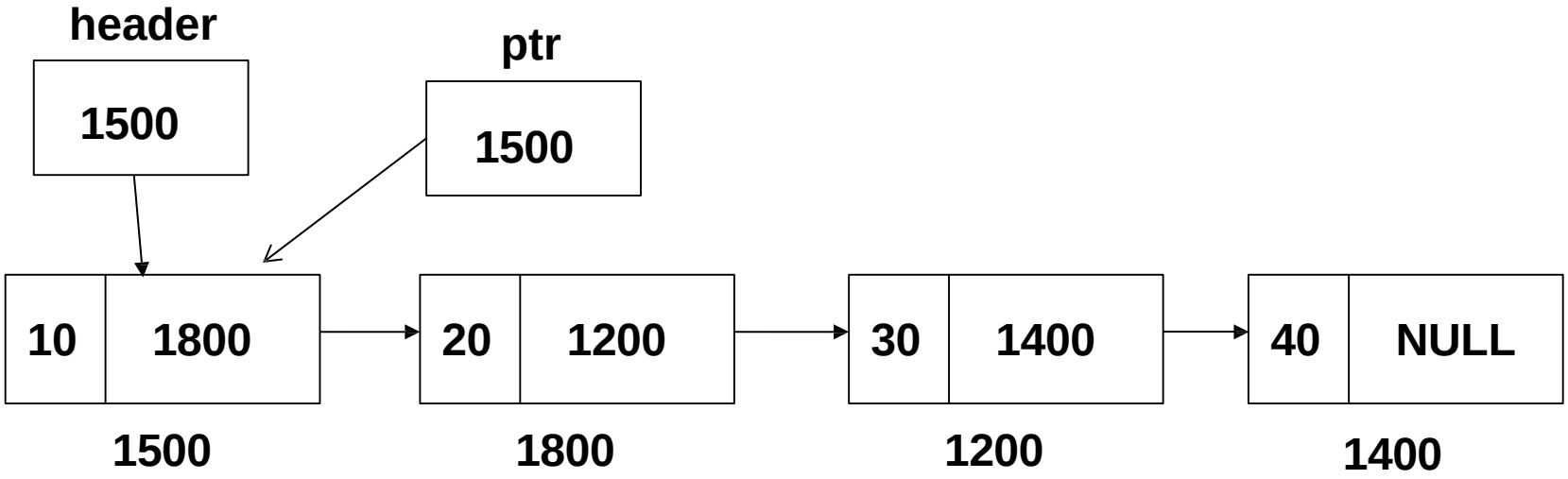
Deleting a node at the given position



Algorithm:

1. ptr = header ;
2. for(i=1;i<pos-1;i++)
3. ptr = ptr -> link;
4. ptr1 = ptr -> link;
5. ptr -> link = ptr1-> link;
6. free(ptr1);

Traversing an elements of a list



Algorithm:

```
1.  if(header = = NULL)
2.        print  "List is empty";
3.  else
4.        for (ptr = header ; ptr != NULL ; ptr = ptr -> link)
5.              print "ptr->data";
```

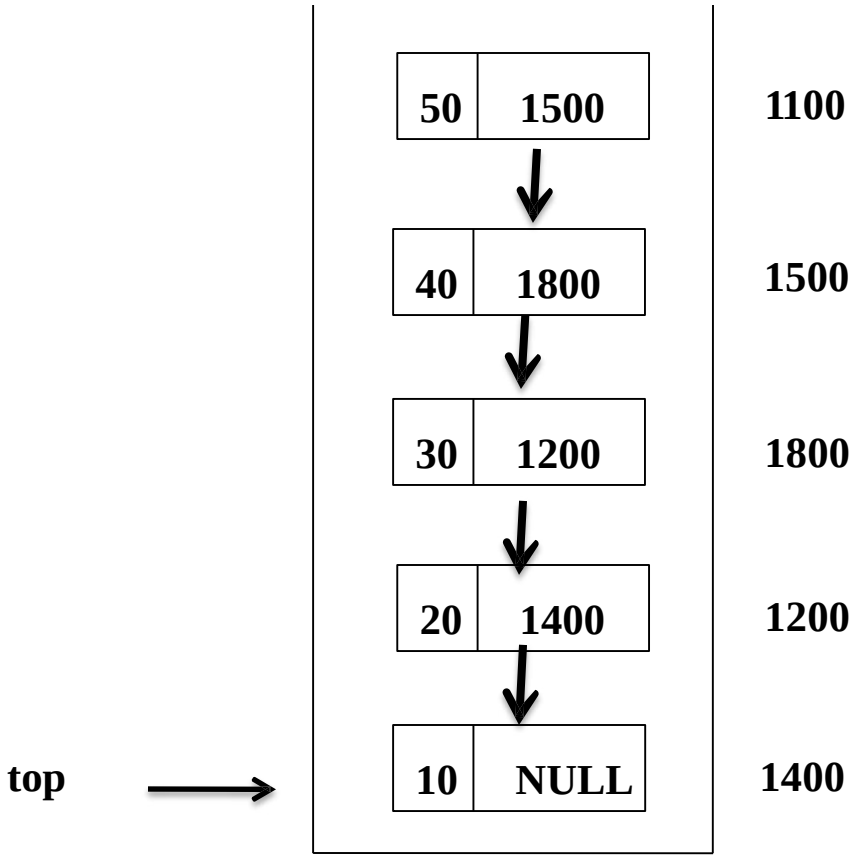
Representing Stack with Linked List

- Disadvantage of using an array to implement a stack or queue is the wastage of space.
- Implementing stacks as linked lists provides a feasibility on the number of nodes by dynamically growing stacks, as a linked list is a dynamic data structure.
- The stack can grow or shrink as the program demands it to.
- A variable **top** always points to top element of the stack.
- top = NULL specifies stack is empty.

Example:

- The following list consists of five cells, each of which holds a data object and a link to another cell.

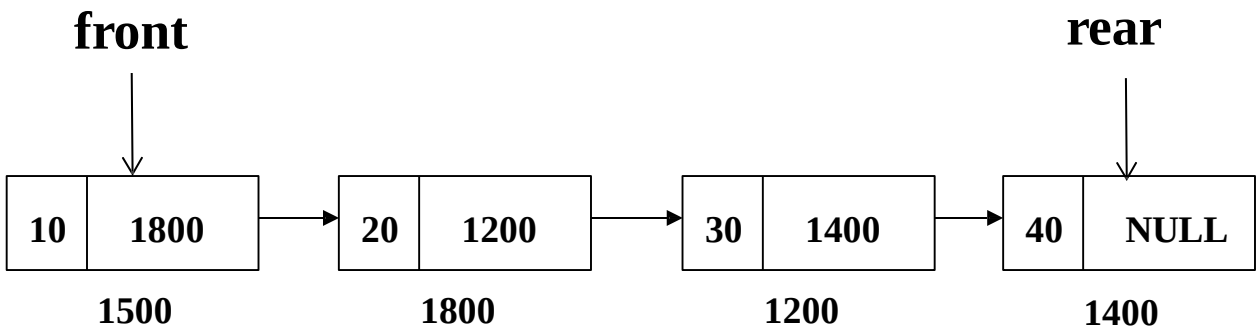
- A variable, **top**, holds the address of the first cell in the list.



Representing Queue with Linked List

- New items are added to the end of the list.
- Removing an item from the queue will be done from the front.

- A pictorial representation of a queue being implemented as a linked list is given below.



- The variables **front** points to the front item in the queue and **rear** points to the last item in the queue.

Doubly linked list

- In a singly linked list one can move from the header node to any node in one direction only (left-right).

- A doubly linked list is a two-way list because one can move in either direction. That is, either from left to right or from right to left.

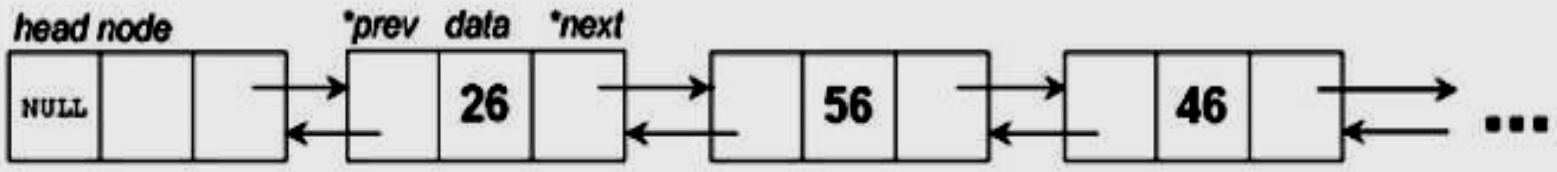
- It maintains two links or pointer. Hence it is called as doubly linked list.

PREV	DATA	NEXT

Structure of the node

- Where, DATA field - stores the element or data, PREV- contains the address of its previous node, NEXT- contains the address of its next node.

An example of a doubly linked list



Doubly linked list operations

Insertion:

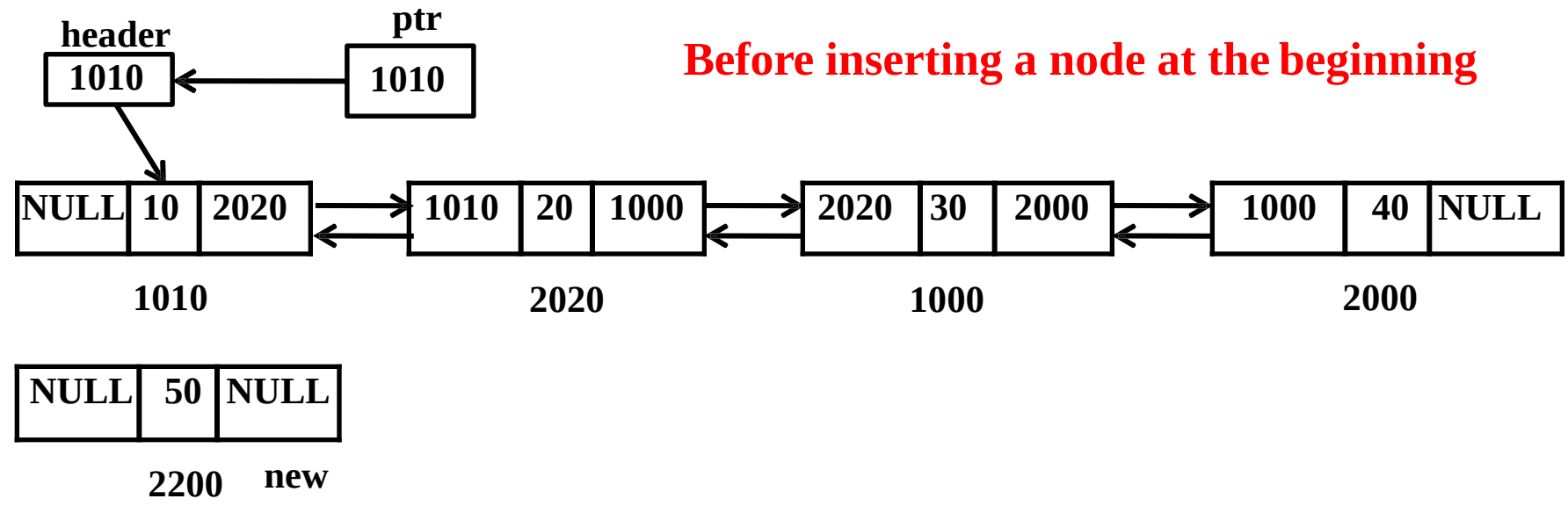
- **Insertion of a node at the front**
- **Insertion of a node at any position in the list**
- **Insertion of a node at the end**

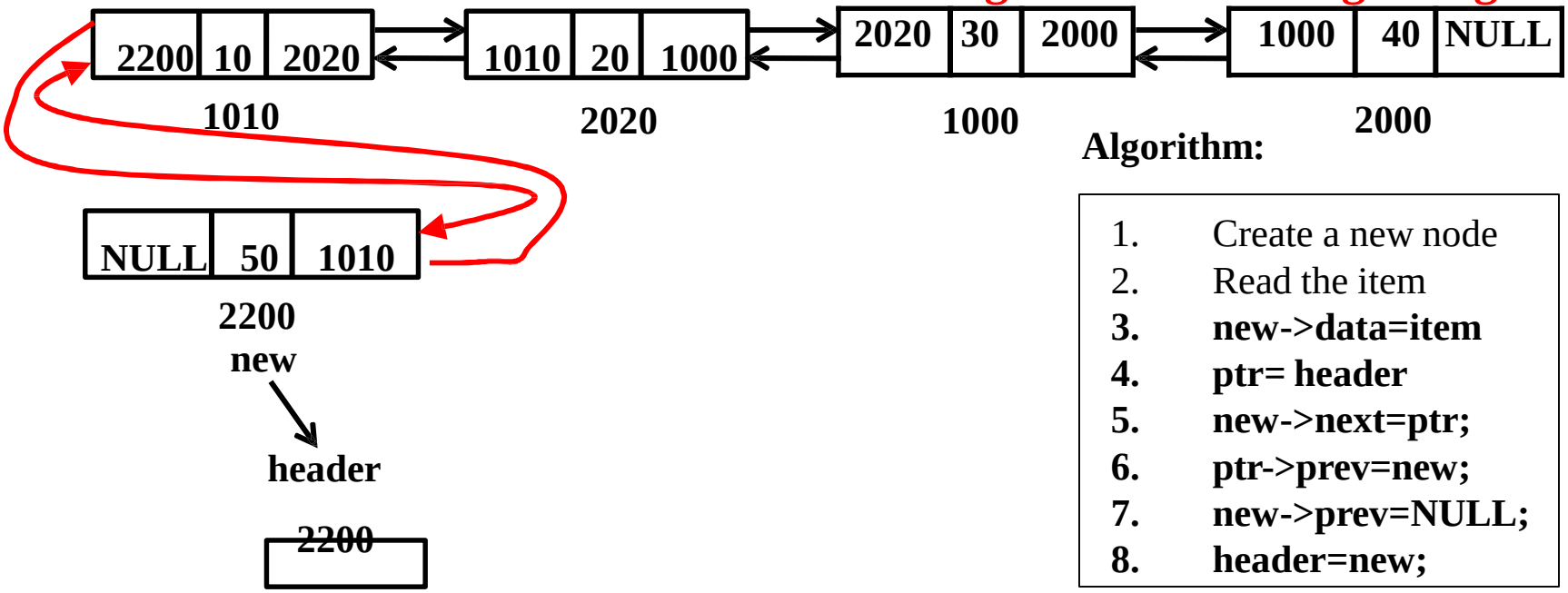
Deletion:

- **Deletion at front**
- **Deletion at any position**
- **Deletion at end**

Display:

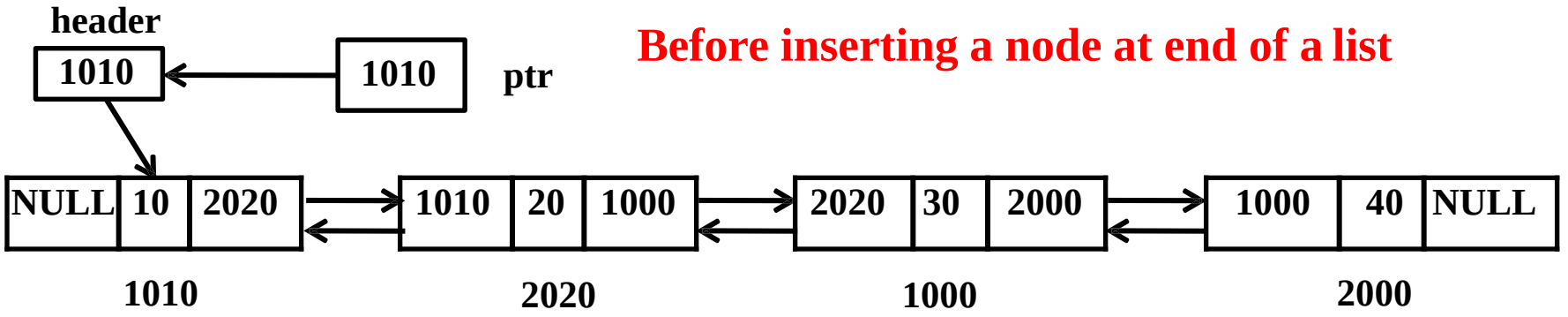
- **Displaying/Traversing the elements of a list**

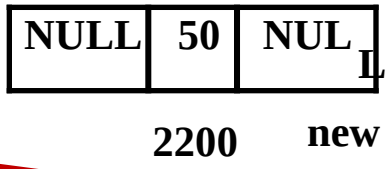




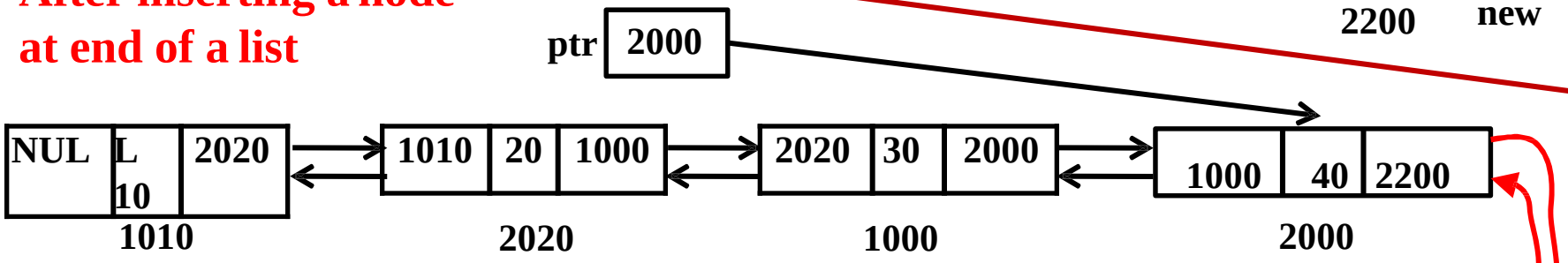
Algorithm:

1. Create a new node
2. Read the item
3. **new->data=item**
4. **ptr= header**
5. **new->next=ptr;**
6. **ptr->prev=new;**
7. **new->prev=NULL;**
8. **header=new;**



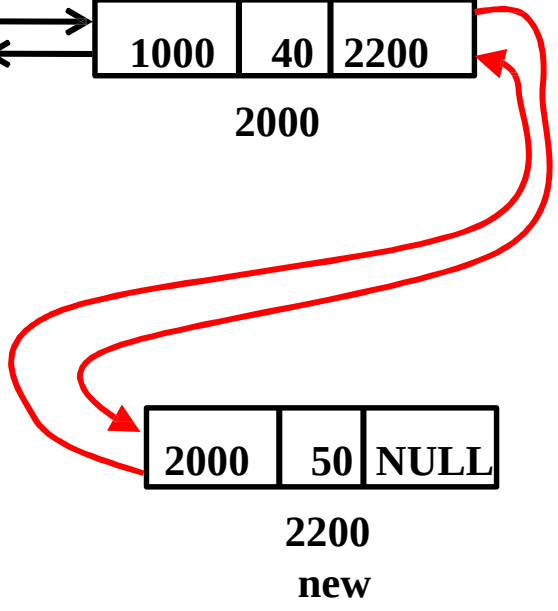


After inserting a node at end of a list



Algorithm:

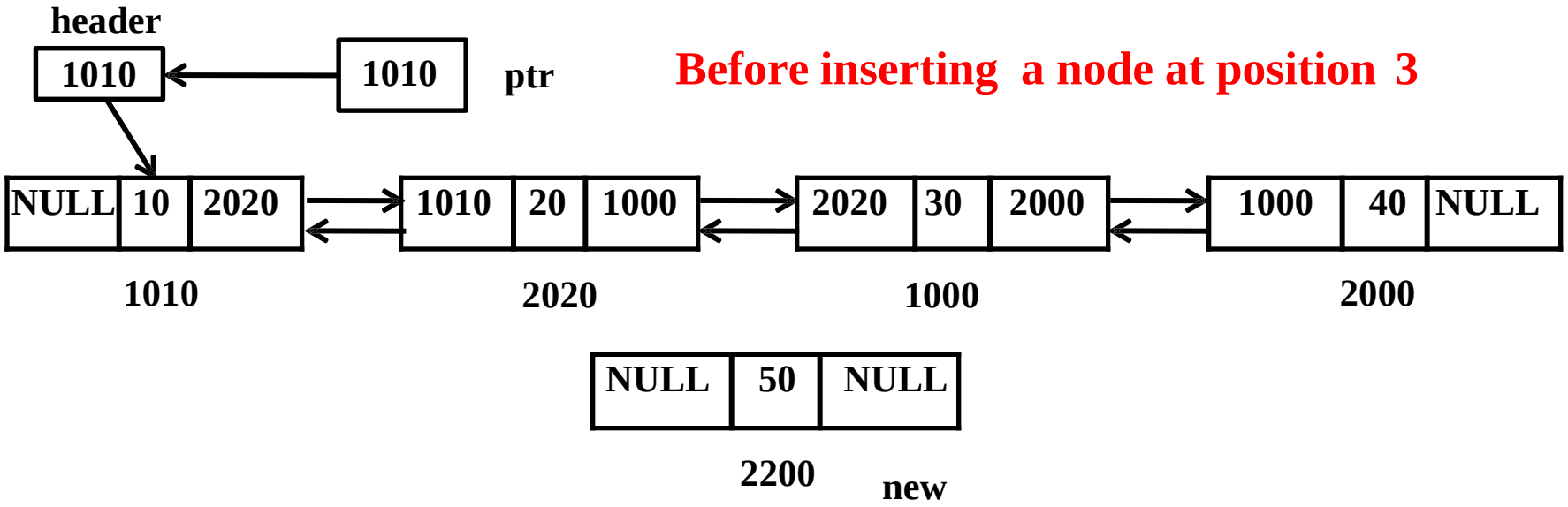
1. Create a new node
2. Read the item
3. new->data=item
4. ptr= header
5. **while(ptr->next!=NULL)**
   1. **ptr=ptr->next;**
6. **new->next=NULL;**
7. **new->prev=ptr;**
8. **ptr->next=new;**

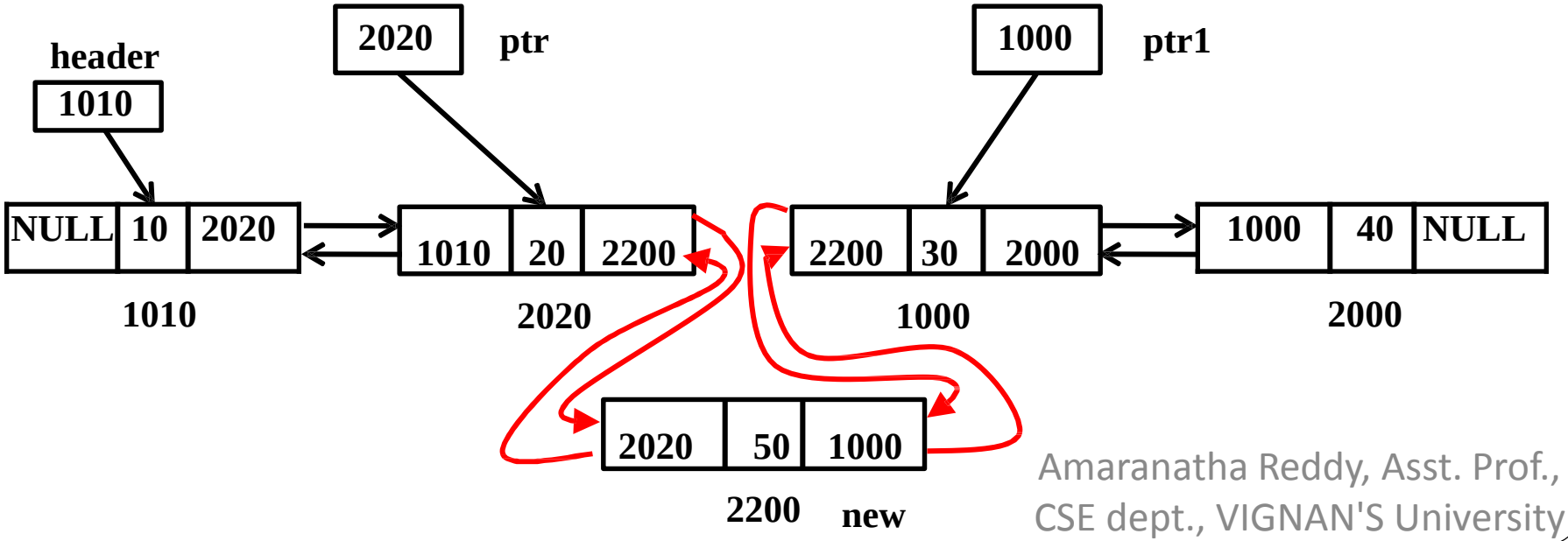


Insertion of a node at any position in the list

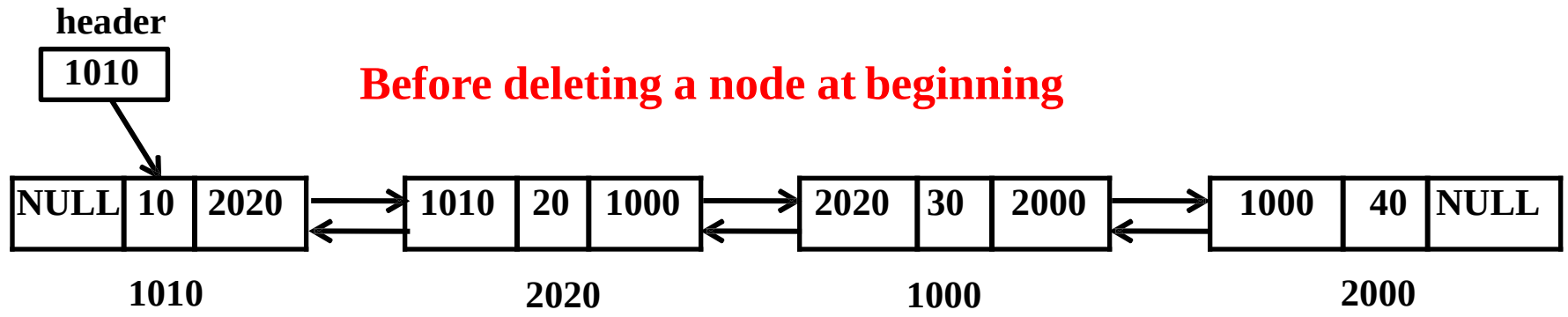
Algorithm:

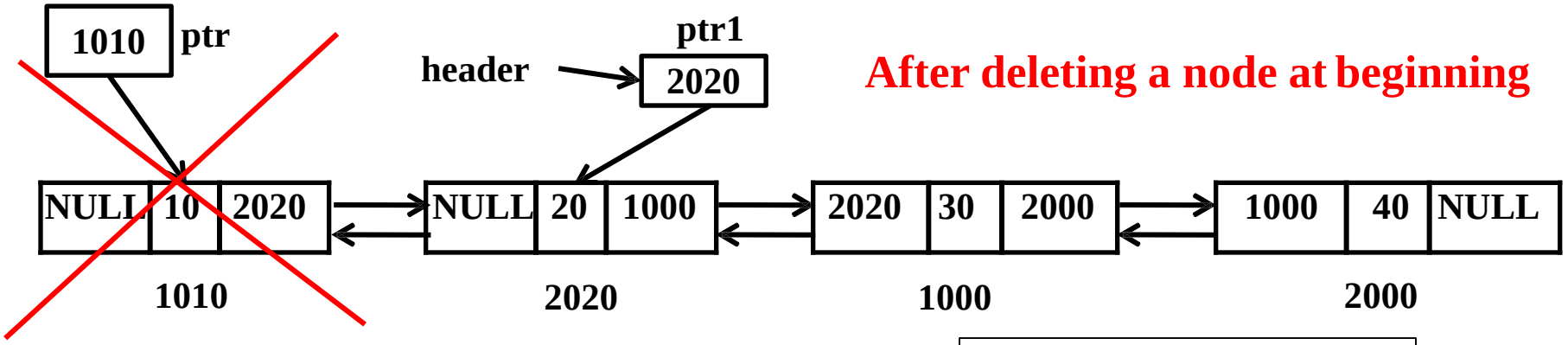
1.create a node new
2. read item
3. new->data=item
4. ptr=header;
5.Read the position where the element is to be inserted
6. for(i=1;i<pos-1;i++)
    6.1 ptr=ptr->next;
7. if(ptr->next = = NULL)
    1. new->next = NULL;
    2. new->prev=ptr;
    3. ptr->next=new;
8. else
    1. ptr1=ptr->next;
    2. new->next=ptr1;
    3. ptr1->prev=new;
    4. new->prev=ptr;
    5. ptr->next=new;
9. end





Amaranatha Reddy, Asst. Prof.,
CSE dept., VIGNAN'S University





Algorithm:

1.ptr=header

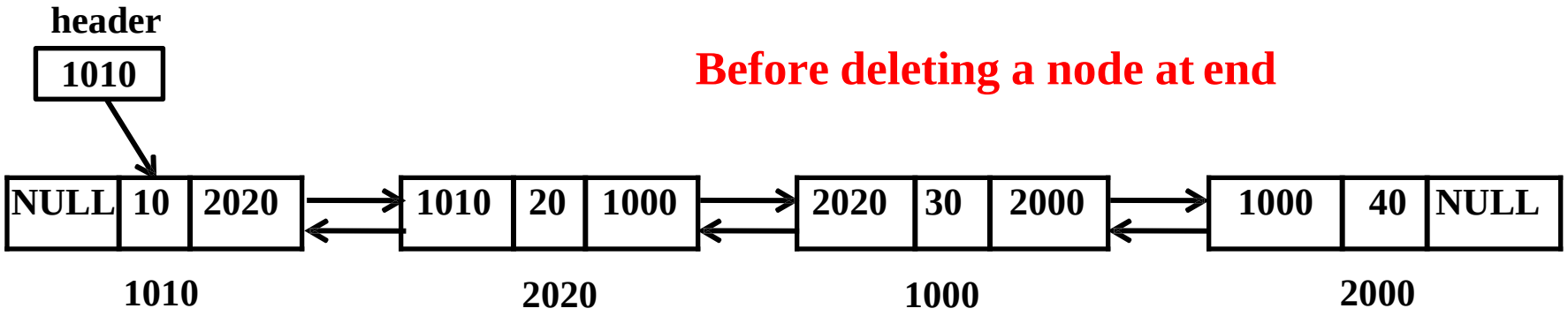
2.ptr1=ptr->next;

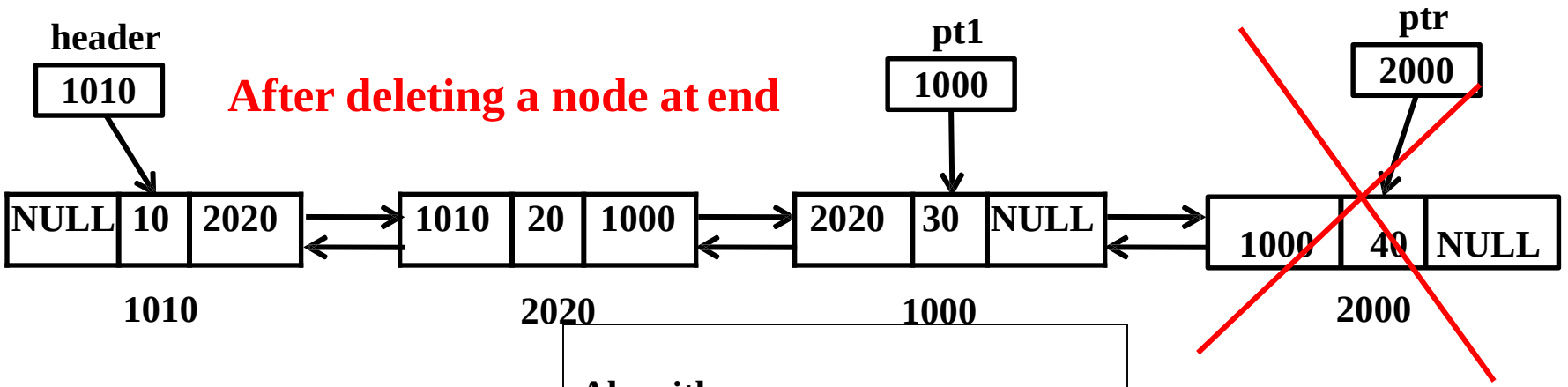
3.header=ptr1;

4.if(ptr1!=NULL)

1.ptr1->prev=NULL;

5. free(ptr);





Algorithm:

1. ptr=header
2. **while(ptr->next!=NULL)**
   1. **ptr=ptr->next;**
3. end while
4. **ptr1=ptr->prev;**
5. **ptr1->next=NULL;**

Deletion at any position

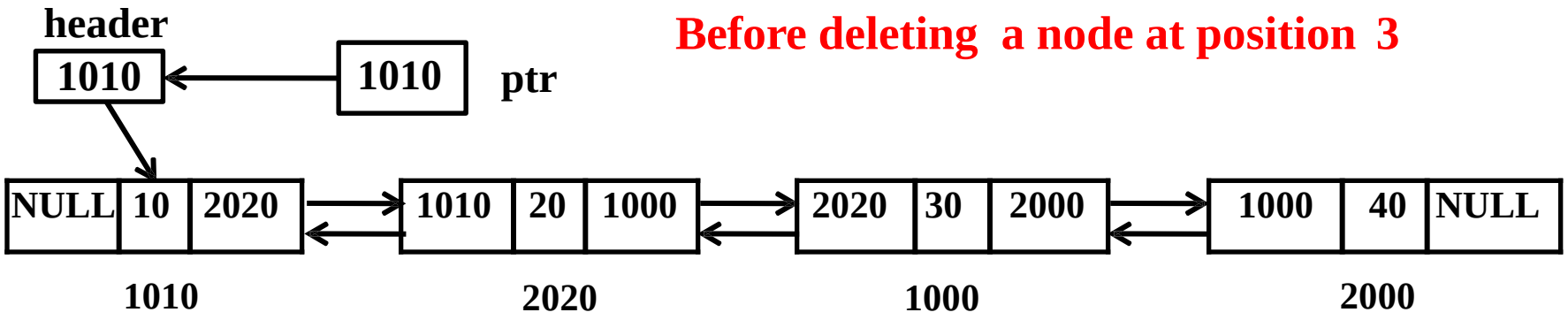
Algorithm:

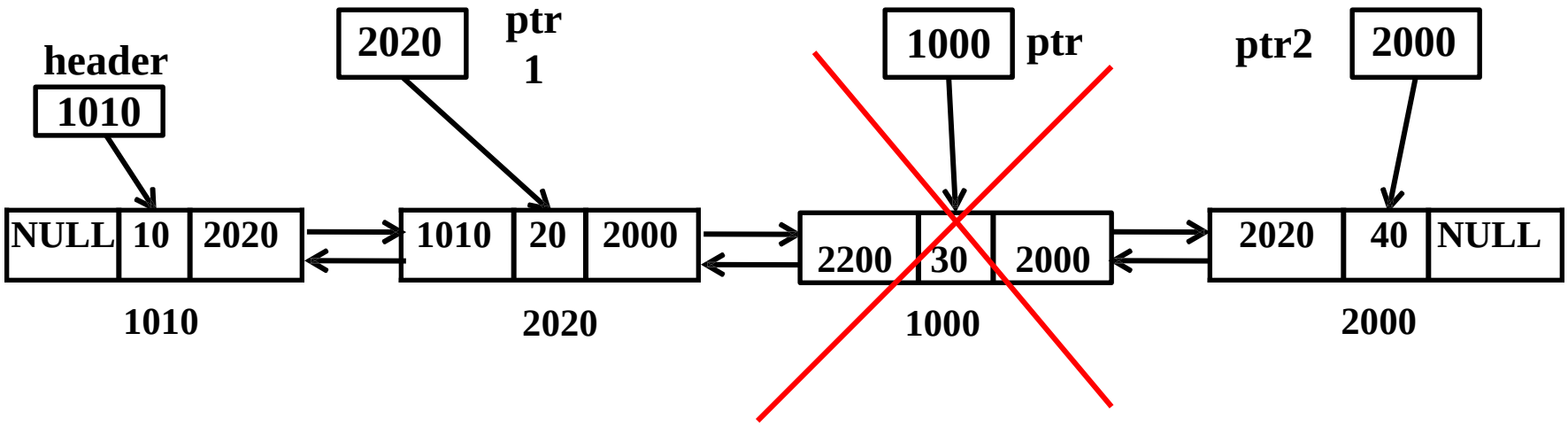
1. ptr=header;
2. **while(ptr->next!=NULL)**
   1.for(i=0;i<pos-1;i++)
      1. ptr=ptr->next;
   2.if(i = = pos-1)
      1. break;
3. end while

4. if(ptr = = header)
   //if the deleted item is first node
   1. ptr1=ptr->next;
   2. ptr1->prev=NULL;
   3. header=ptr1;
   4. end if

5.else
   1. ptr1=ptr->prev;
   2. ptr2=ptr->next;
   3. ptr1->next=ptr2;
   4. ptr2->prev=ptr1;

6. end else
7. end if

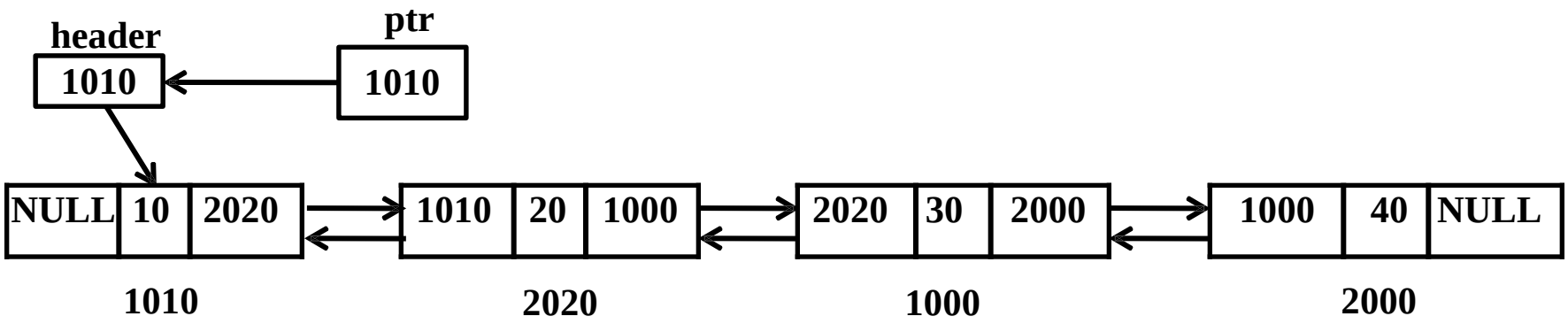

Before deleting a node at position 3
header
1010
1010
ptr
NULL
10
2020
1010
20
1000
2020
30
2000
1000
40
NULL
1010
2020
1000
2000


After deleting a node at position 3
header
1010
2020
ptr
1
1000
ptr
ptr2
2000
NULL
10
2020
1010
20
2000
2200
30
2000
2020
40
NULL
1010
2020
1000
2000

Displaying elements of a list

Algorithm:

1. ptr=header;
2. if(header = = NULL)
    1. printf("The list is empty\n");
3. else
    1. print “The elements in farword order: “
    2. while(ptr!=NULL)
        1. print “ptr->data”;
        2. if(ptr->next = = NULL)
            1. break;
        3. ptr=ptr->next;
    4. print “The elements in reverse order: “
    5. while(ptr!=header)
        1. if(ptr->next = = NULL)
            1.print “ptr->data”;
        2. el
            se print “ptr->data”;
            1. ptr=ptr->prev;
            2. print “ptr->data”;
        3.end 1.else
4. end else

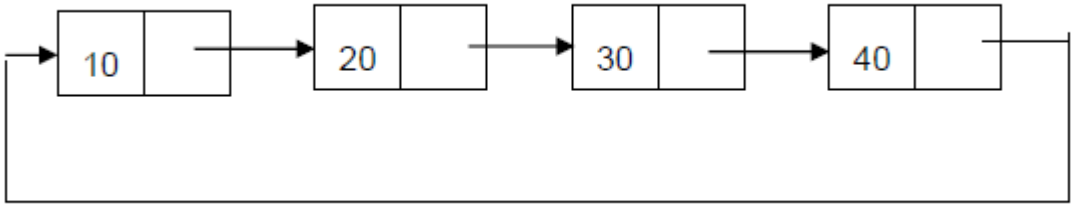


Forward Order : 10 20 30 40

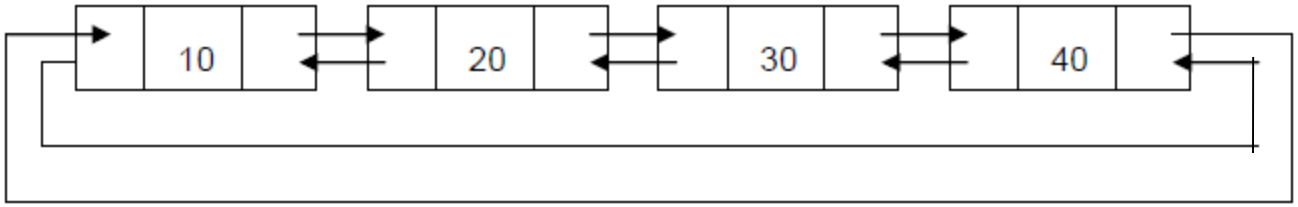
Reverse Order : 40 30 20 10

Circular linked list

- The linked list where the last node points the header node is called circular linked list.



Circular singly linked list



Circular doubly linked list