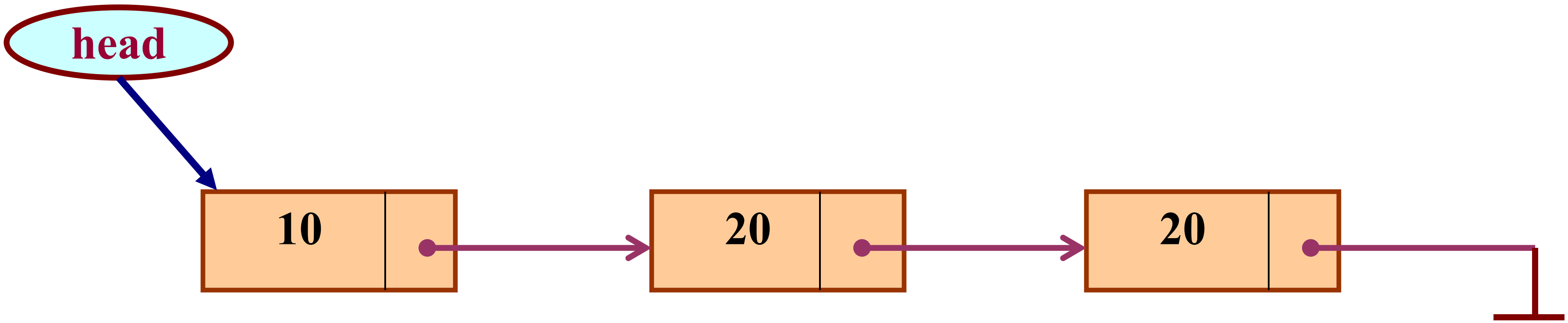
Linked List

Alok Kumar Jagadev

Linked List

- Linked List is a commonly used linear data structure
- Consists of group of nodes in a sequence
- Each node holds data (info) and the address of the next node forming a chain like structure
- Head: pointer to the first node
- The last node points to NULL



Linked List

- Linked lists
  - Abstract data type (ADT)
- Basic operations of linked lists
  - Insert, find, delete, print, etc.
- Variations of linked lists
  - Single linked lists
  - Double linked lists
  - Circular linked lists

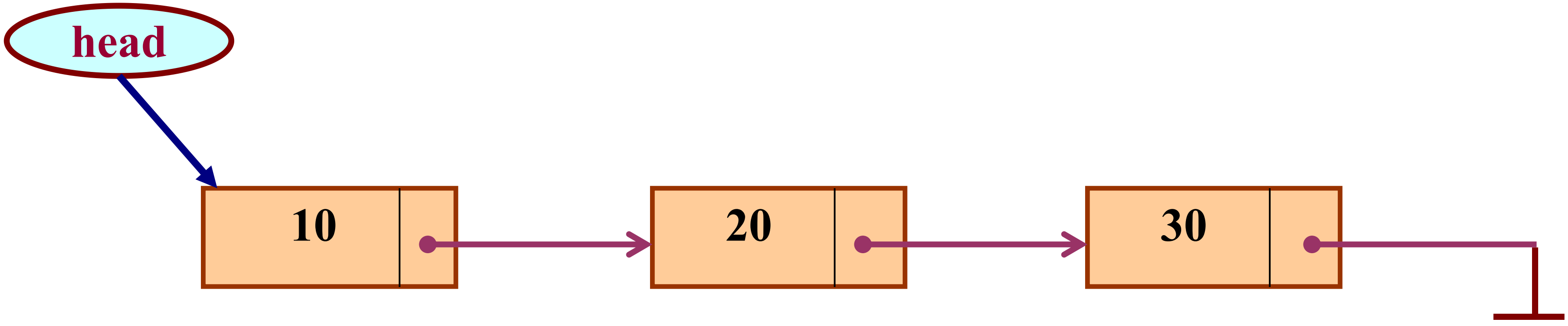
Array versus Linked Lists

- Arrays are suitable for:
  - Inserting/deleting an element at the end
  - Randomly accessing any element
  - Searching the list for a particular value

- Linked lists are suitable for:
  - Inserting an element
  - Deleting an element
  - Applications where sequential access is required
  - In situations where the number of elements cannot be predicted beforehand

Types of Lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
  - Linear single-linked list (or simply linear list)

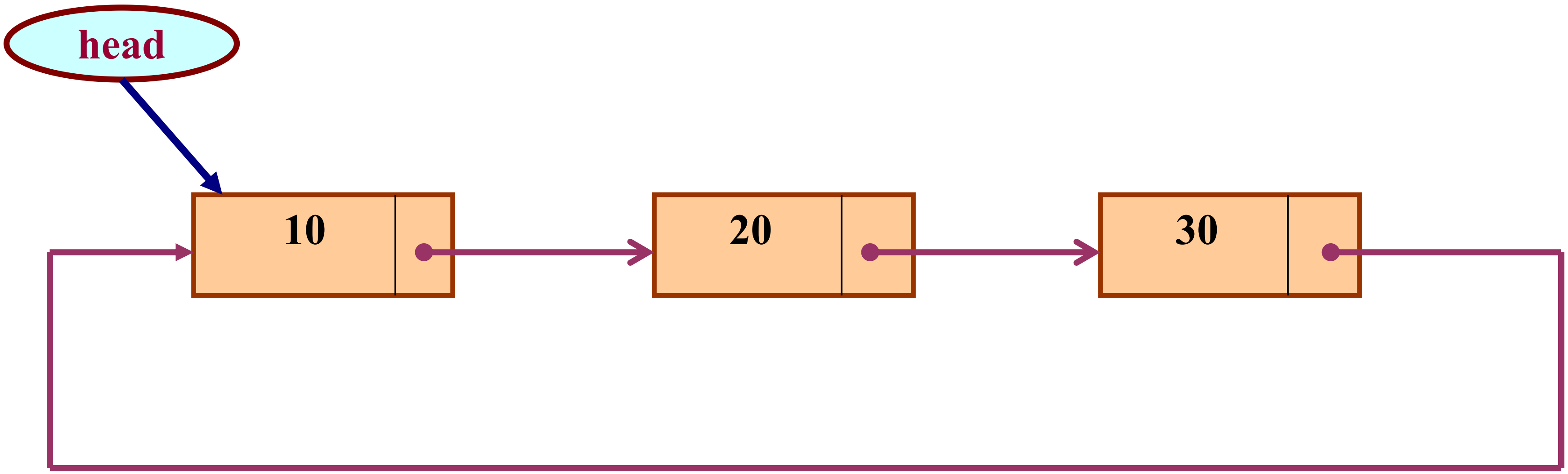


Single-linked lists vs. 1D-arrays

ID-array	Single-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.	Extra storage needed for references; however uses exactly as much memory as it needs
Access is faster because of greater locality of references [Reason: Elements in contiguous memory locations]	Access is slower because of low locality of references [Reason: Elements not in contiguous memory locations]

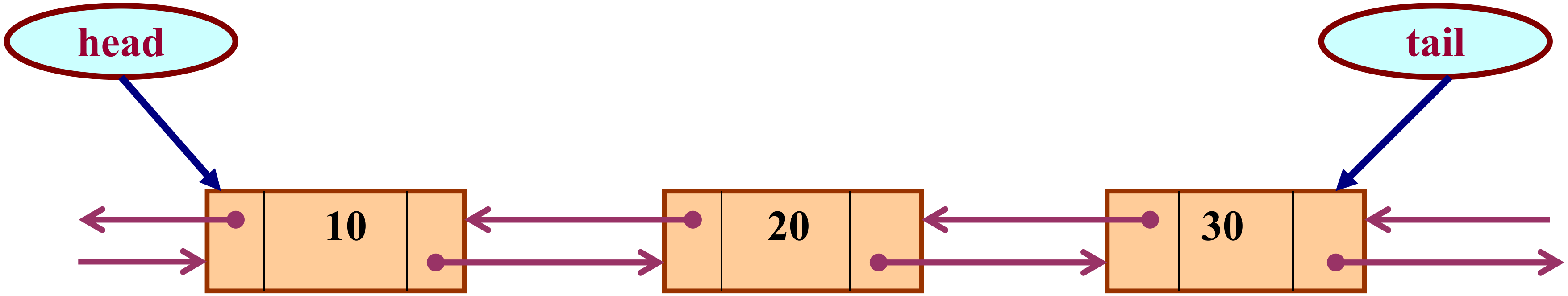
Circular Linked List

- Circular linked list
  - The pointer from the last element in the list points back to the first element.



Double Linked List

- Double linked list
  - Pointers exist between adjacent nodes in both directions.
  - The list can be traversed either forward or backward.
  - Usually two pointers are maintained to keep track of the list, *head* and *tail*.



Why Linked List?

- Arrays can be used to store linear data of similar types, but arrays have the following limitations.
  - size of the arrays is fixed
  - upper limit on the number of elements must know in advance.
  - Allocated memory is for the total array irrespective of the usage.

- Inserting a new element in an array of elements is expensive
  - the room has to be created for the new elements and
  - to create room existing elements have to be shifted.

Basic Operations on a List

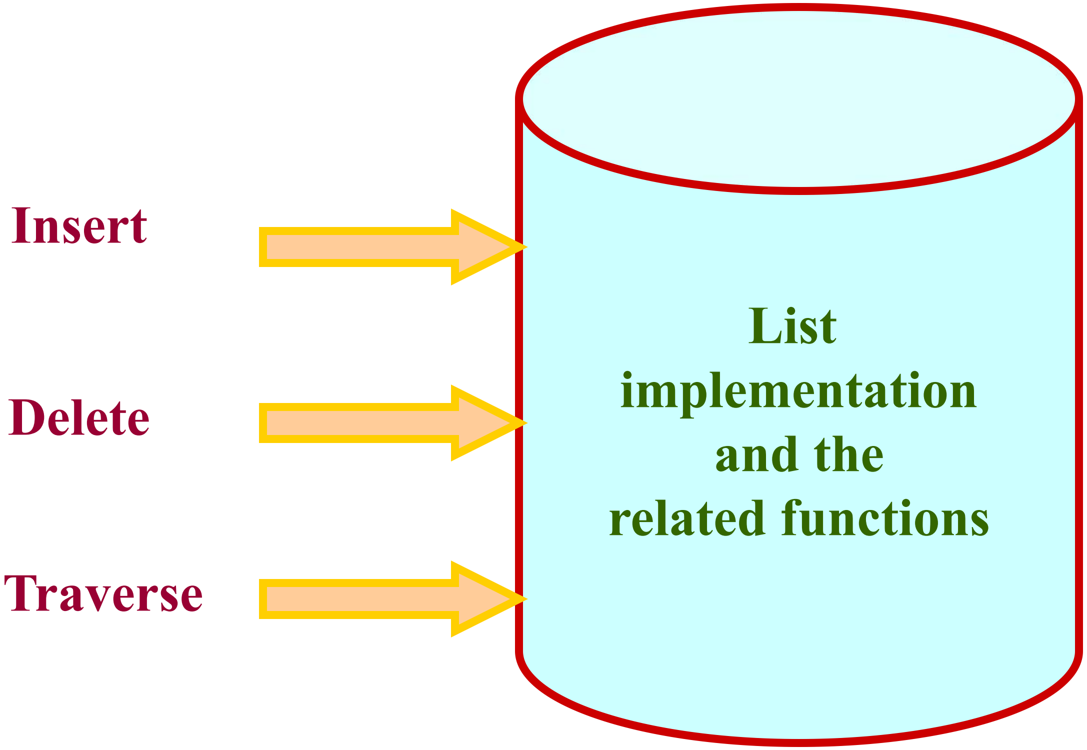
- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

List is an Abstract Data Type

- What is an abstract data type?
  - data type defined by the user
  - Typically more complex than simple data types like *int*, *float*, etc.

- Why abstract?
  - Because details of the implementation are hidden.
  - When some operations on the list are performed, just the functions are called.
  - Details of how the list is implemented or how the insert function is written is no longer required.

Conceptual Idea



Structure of a Node

- Declare Node structure
  - data: int-type data in this example
  - next: a pointer to the next node in the list

```
struct Node {
        int data;                   // data
        struct Node* next;          // pointer to next node
};
```

Create a List

```
void createList()  {
    int  k, n;
    struct Node  *p, *head;
    printf  ("Number of nodes: ");
    scanf ("%d", &n);
    for  (k=0; k<n; k++)      {
        if (k == 0) {
            head = (struct Node *) malloc(sizeof(struct Node));
            p = head;
        }
        else {
            p->next  = (struct Node *) malloc(sizeof(struct Node));
            p = p->next;
        }
        scanf ("%d", &p->data);
    }
    p->next  =  NULL;
}
```

Traversing the List

- Once the linked list has been constructed and *head* points to the first node of the list,
  - Follow the pointers
  - Display the contents of the nodes as they are traversed
  - Stop when the *next* pointer points to NULL

Traversing the List

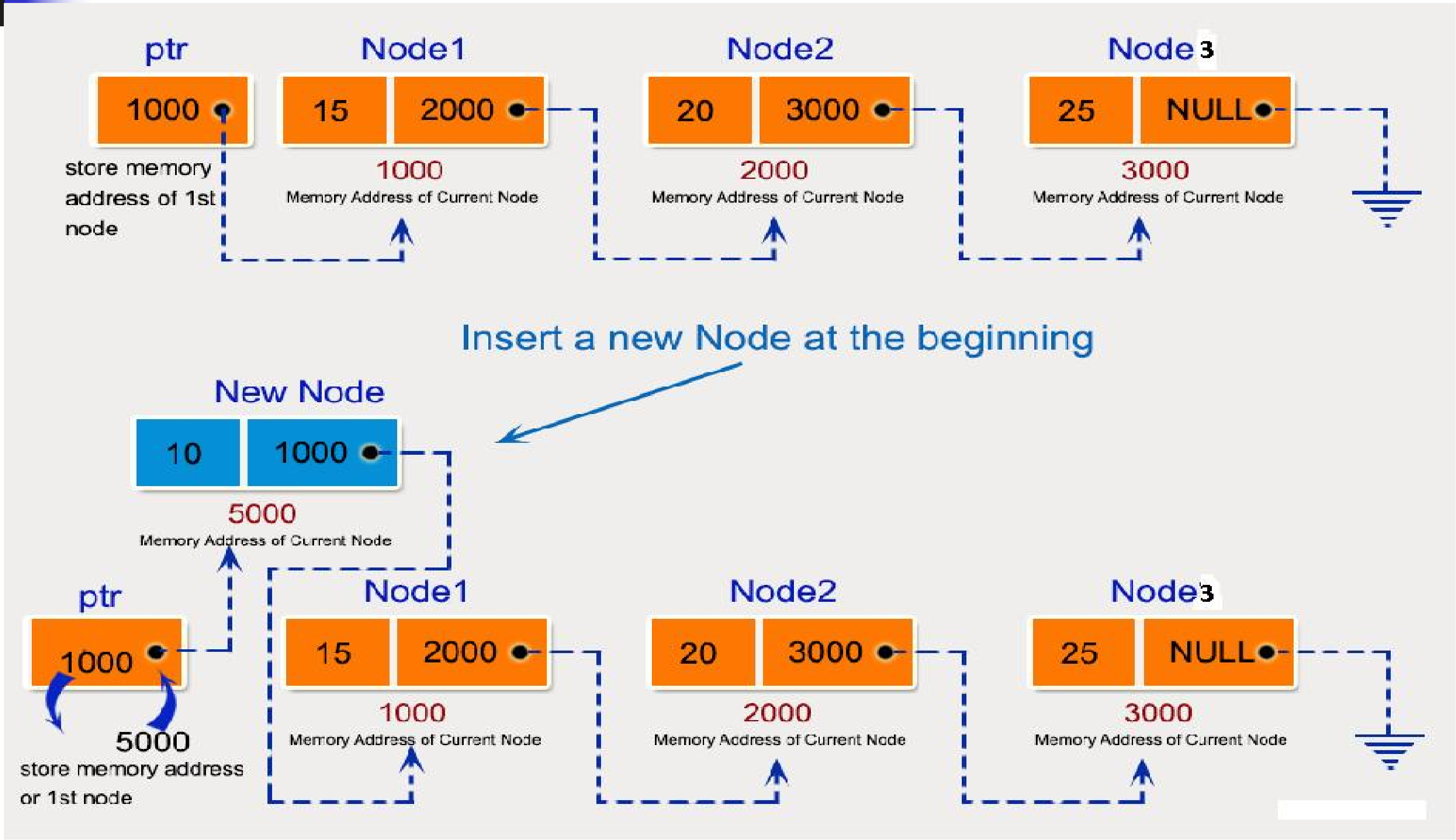
```
void display ()  {
  int  count = 1;
  struct Node  *p;
  if(head == NULL)  {
       printf("\nEmpty List...");
       return;
  }
  p = head;
  while (p != NULL)   {
     printf ("\nNode: %d:  %d", count, p->data);
     count++;
     p = p->next;
  }
  printf ("\n");
}
```

Inserting a Node in a List

Inserting a Node in a List

- Insert at beginning of the list:
  - Only one next pointer needs to be modified.
    - *head* is made to point to the new node.
    - New node points to the previously first node.
- Insert at end of the list:
  - Two next pointers need to be modified.
    - Last node points to the new node.
    - New node points to NULL.
- When a node is added in the middle (at any position)
  - Two next pointers need to be modified.
    - Previous node now points to the new node.
    - New node points to the next node.

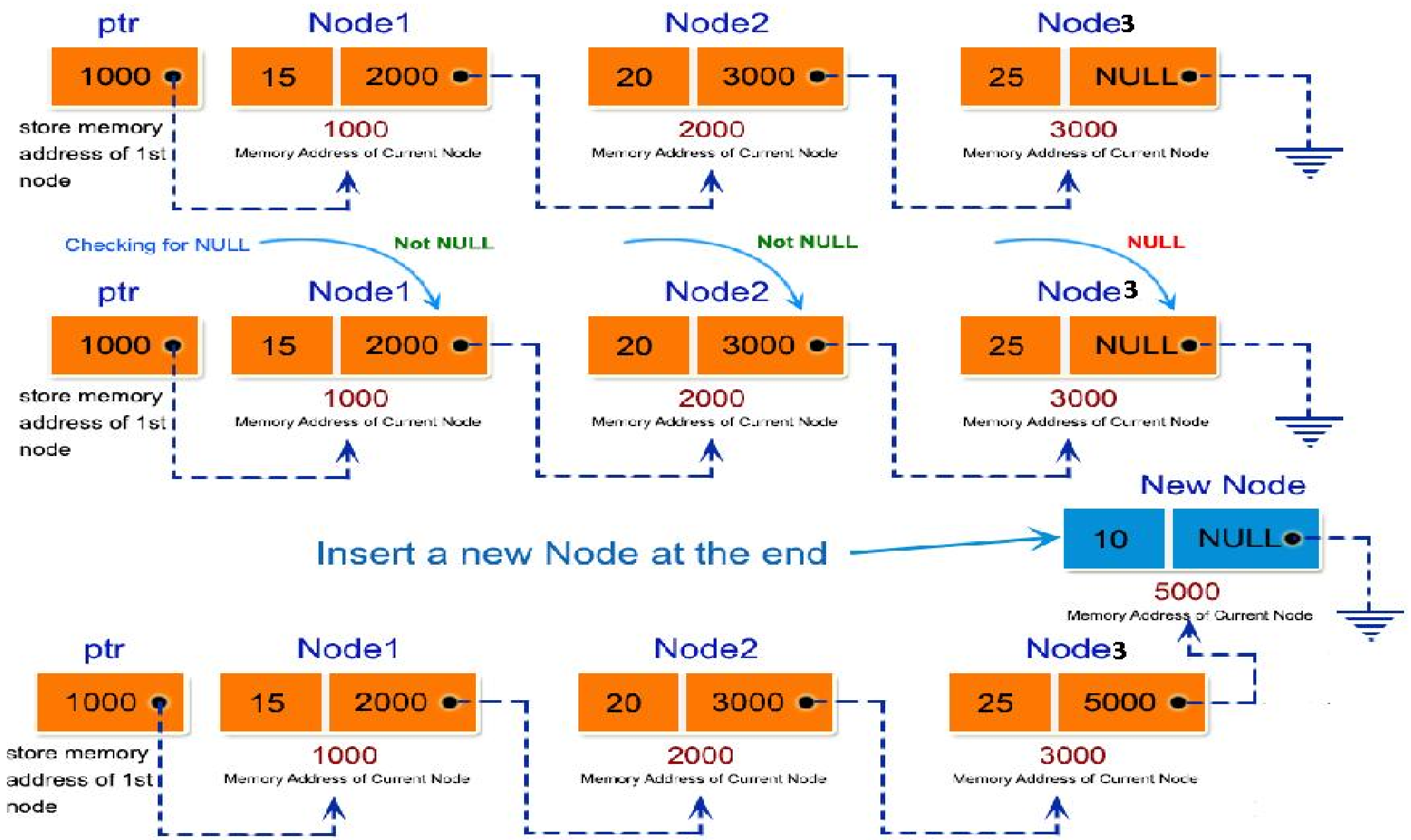
Inserting at Begining



Inserting at Begining

```
void insertAtBegining() {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    printf("\nEnter the new data: ");
    scanf("%d", &newNode->data);
    newNode->next = head;
    head = newNode;
}
```

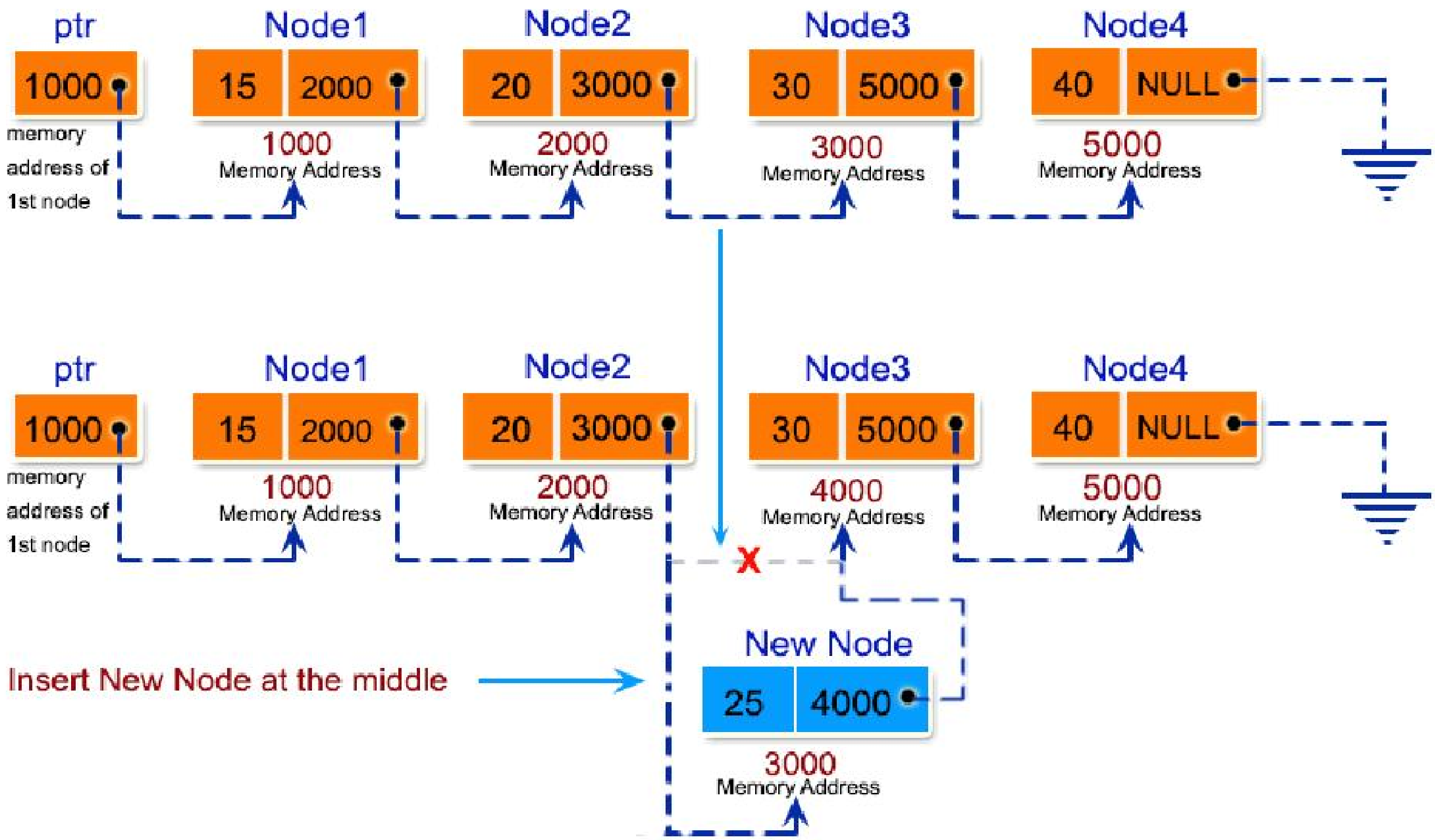
Inserting at End



Inserting at End

```
void insertAtEnd() {
    struct Node* last;
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    printf("\nEnter the new data: ");
    scanf("%d", &newNode->data);
    newNode->next = NULL;
    last = head;
    if (head == NULL) {
      head = newNode;
      return;
    }
    while (last->next != NULL)
       last = last->next;
    last->next = newNode;
}
```

Inserting After a Node



Inserting After a Node

```
void insertAfter() {
    struct Node , *prev, *newNode;
    prev=head;
    while(prev->next != NULL && prev->data != val)
          prev = prev->next;
     if (prev->data == val)    {
         newNode =(struct Node*) malloc(sizeof(struct Node));
         printf("\nEnter the new data: ");
        scanf("%d", &newNode->data);
        newNode->next = prev->next;
        prev->next = newNode;
    }
    else
         printf( "Value %d is not in list\n", val);
}
```

Insert a node at a specific position

```
void insertAtPos()  {
  int pos, i, nodes=0;
  struct Node *newNode, *prev;
  struct Node* curr=head;
  while(curr != NULL){
        nodes++;
        curr=curr->next; }
  printf("\nEnter the position: ");
  scanf("%d", &pos);
  if(pos < 1 || pos > nodes) {
     printf("Invalid Input...");
     return; }
  newNode =(struct Node*) malloc(sizeof(struct Node));
  printf("\nEnter the new data: ");
  scanf("%d", &newNode->data);
  newNode->next=NULL;
  if(pos==1)  {
     newNode->next=head;
     head=newNode;
  }
  i = 1;
  curr = head;
  while(i<pos) {
     i++;
     prev = curr;
     curr = curr->next;
  }
  newNode->next = prev->next;
  prev->next = newNode;
}
```

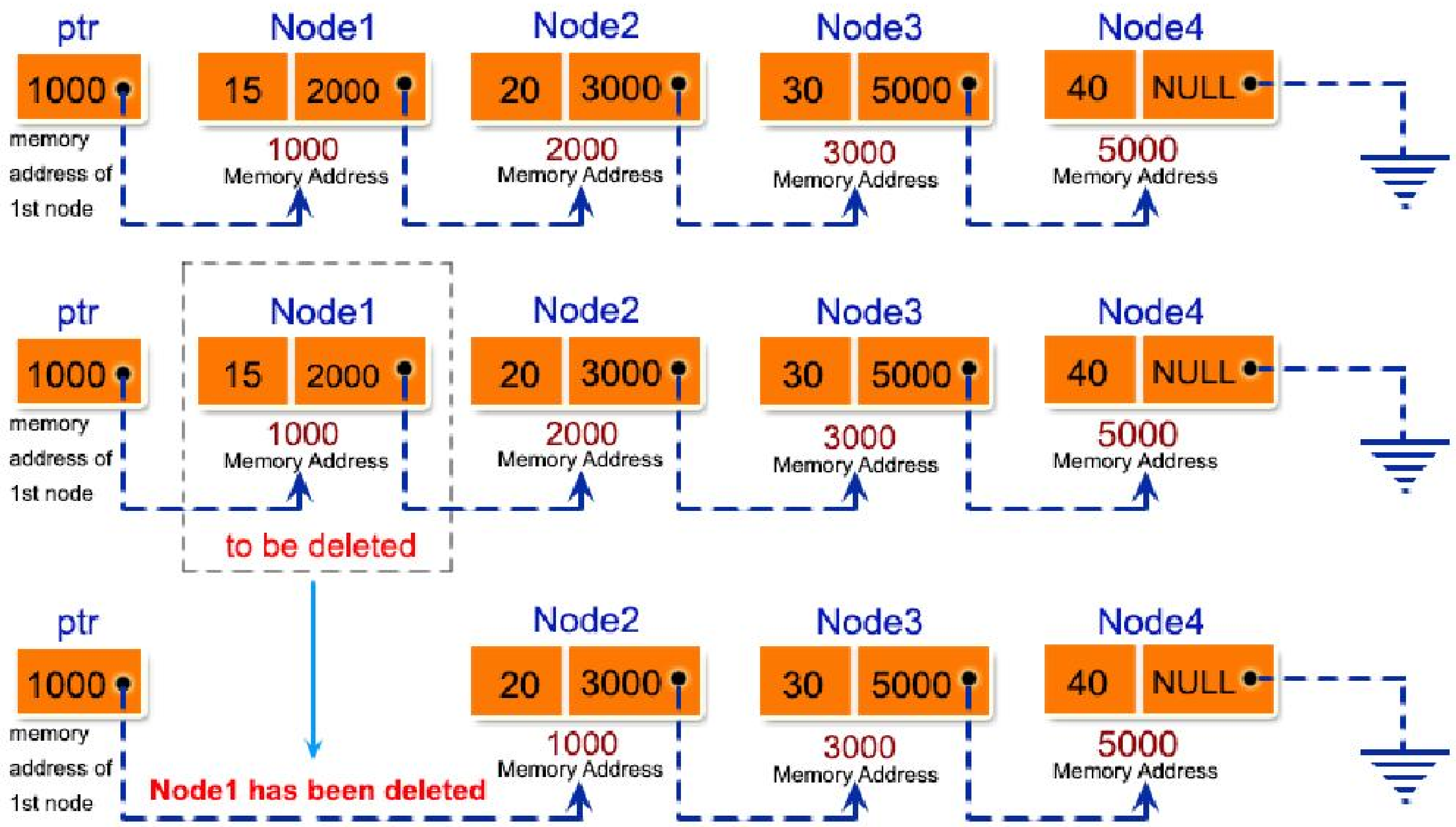
Deleting a Node in a List

Deleting a Node in the List

To delete a node from linked list, need to do following steps:

- Find previous node of the node to be deleted
- Change the next of previous node
- Free memory for the node to be deleted

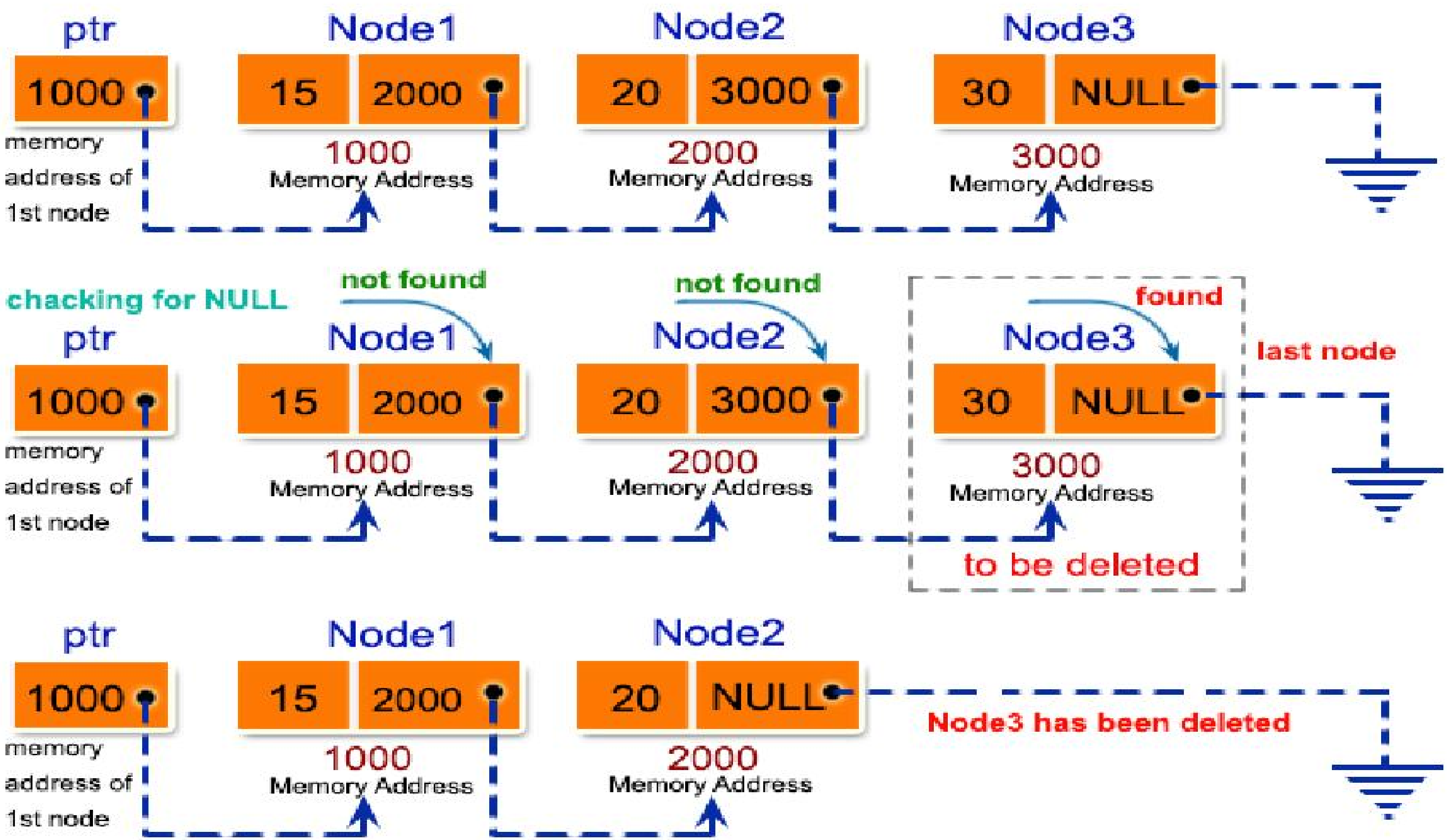
Deleting a Node at Beginning



Deleting a Node at Beginning

```
void deleteAtBeginning() {
    struct Node* temp = head;
    if (temp == NULL)  {
       printf(\nEmpty list...");
       return;
    }
    printf(\nValue of the deleted node = %d", temp->data);
    head = head->next;
    free(temp);
}
```

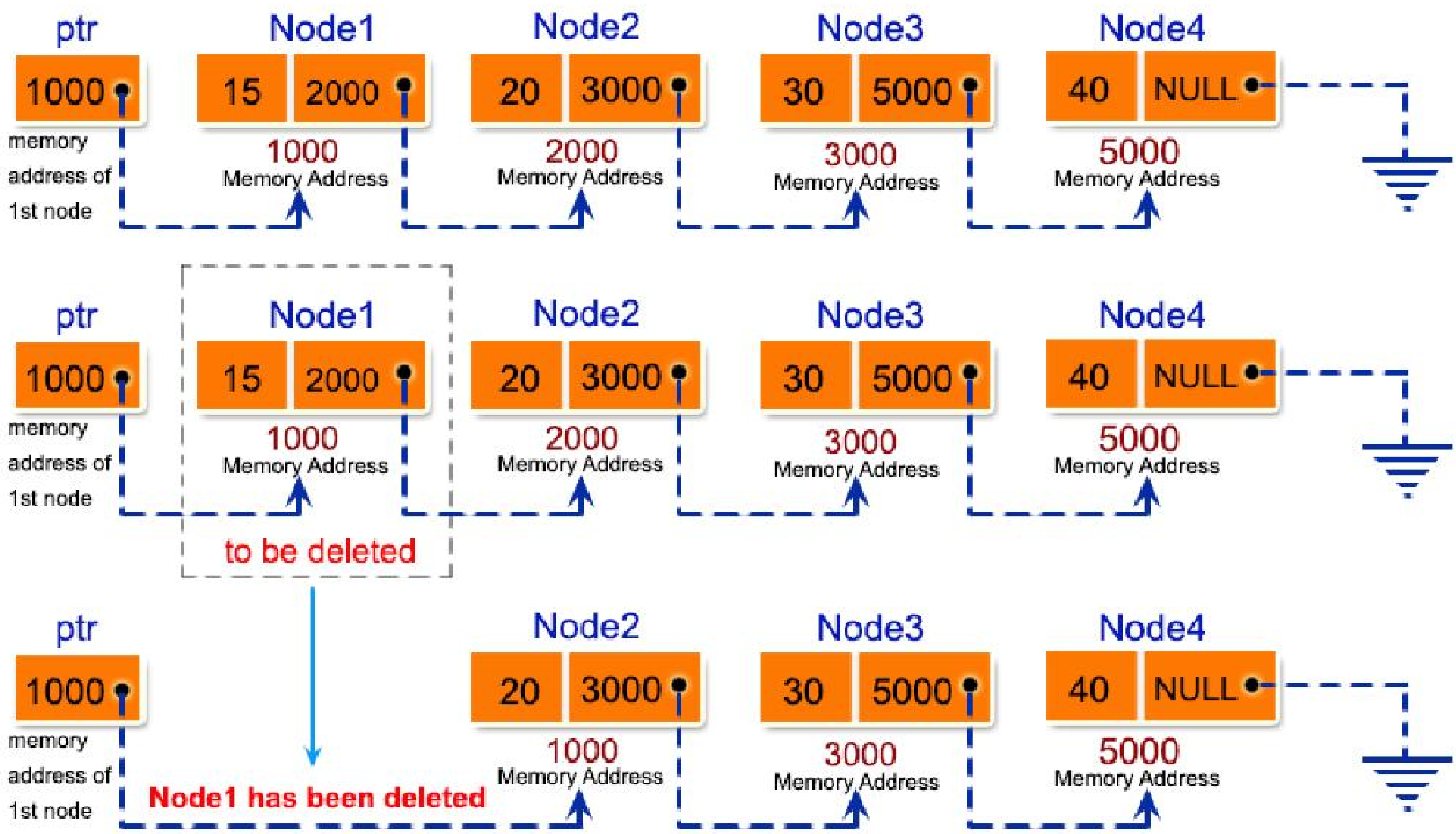
Deleting a Node at End



Deleting a Node at End

```
void deleteAtEnd() {
   struct Node* temp = head;
   struct Node* prev;
   if (temp == NULL)  {
      printf(\nEmpty list...");
      return;
   }
   if (temp->next == NULL)  {
      printf(\nList contains only node, its
                  value = %d", temp->data);
      head = NULL;
      free(temp);
      return;
   }
   while (temp->next != NULL )  {
      prev = temp;
      temp = temp->next;
   }
   printf(\nValue of the deleted node = %d",
            temp->data);
   prev->next = temp->next;
   free(temp);
}
```

Deleting a Node at any Position



Deleting a Node at any Position

```
void deleteAtAnyPosition(int key) {
    struct Node* temp = head, *prev;
    if (temp == NULL )  {
        printf("\nEmpty List...");
        return;
    }
    if (temp != NULL && temp->data == key)  {
        printf(\nValue of the deleted node = %d",
                temp->data);
        head = temp->next;
        free(temp);
        return;
    }
    while ( temp  && temp->data != key)  {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) {
       printf("\nSearched value does not exist
                in the list...");
       return;
    }
    printf(\nValue of the deleted node = %d",
              temp->data);
    prev->next = temp->next;
    free(temp);
}
```

Detect the loop in a single linked list

```
int detectLoop() {
    struct Node *slowp = head, *fastp = head;
    if (head == NULL || head->next == NULL)  {
       printf(\nEmpty list or list contains one node...");
       return 0;
    }
    while (slowp && fastp && fastp->next) {
        slowp = slowp->next;
        fastp = fastp->next->next;
        if (slowp == fastp)
            return 1;
    }
    return 0;
}
```

Reverse a single linked list

```
void reverse() {
    struct Node* prev = NULL, *ptr;
    struct Node* curr = head;
    if(head == NULL)  {
      printf("\nEmpty List ...");
      return;
    }
    while (curr != NULL)  {
        ptr  = curr->next;
        curr->next = prev;
        prev = curr;
        curr = ptr;
    }
    head = prev;
}
```

Double Linked List

A Double Linked List contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in single linked list.

Node of a double linked list

```
struct Node {
    int data;
    struct Node* next;   // Pointer to next node
    struct Node* prev;   // Pointer to previous node
};
```

Double Linked List

Following are advantages/disadvantages of DLL over single linked list.

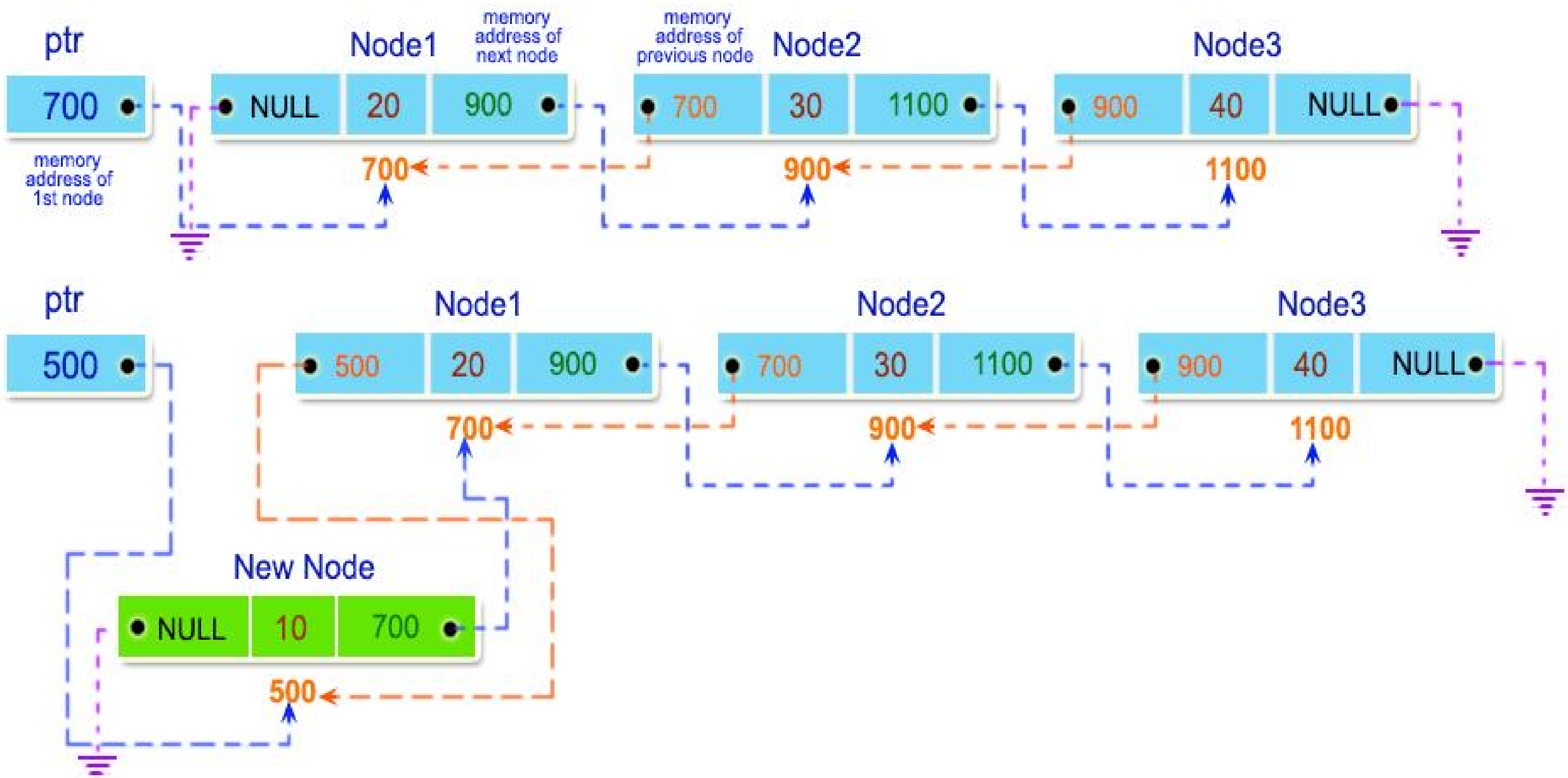
Advantages:

1) A DLL can be traversed in both forward and backward directions.
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
3) Quickly insert a new node before a given node.
4) In single linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, can get the previous node using previous pointer.

Disadvantages:

1) Every node of DLL requires extra space for an previous pointer.
2) All operations require an extra pointer previous to be maintained.
   - For example, in insertion, need to modify previous pointers together with next pointers.

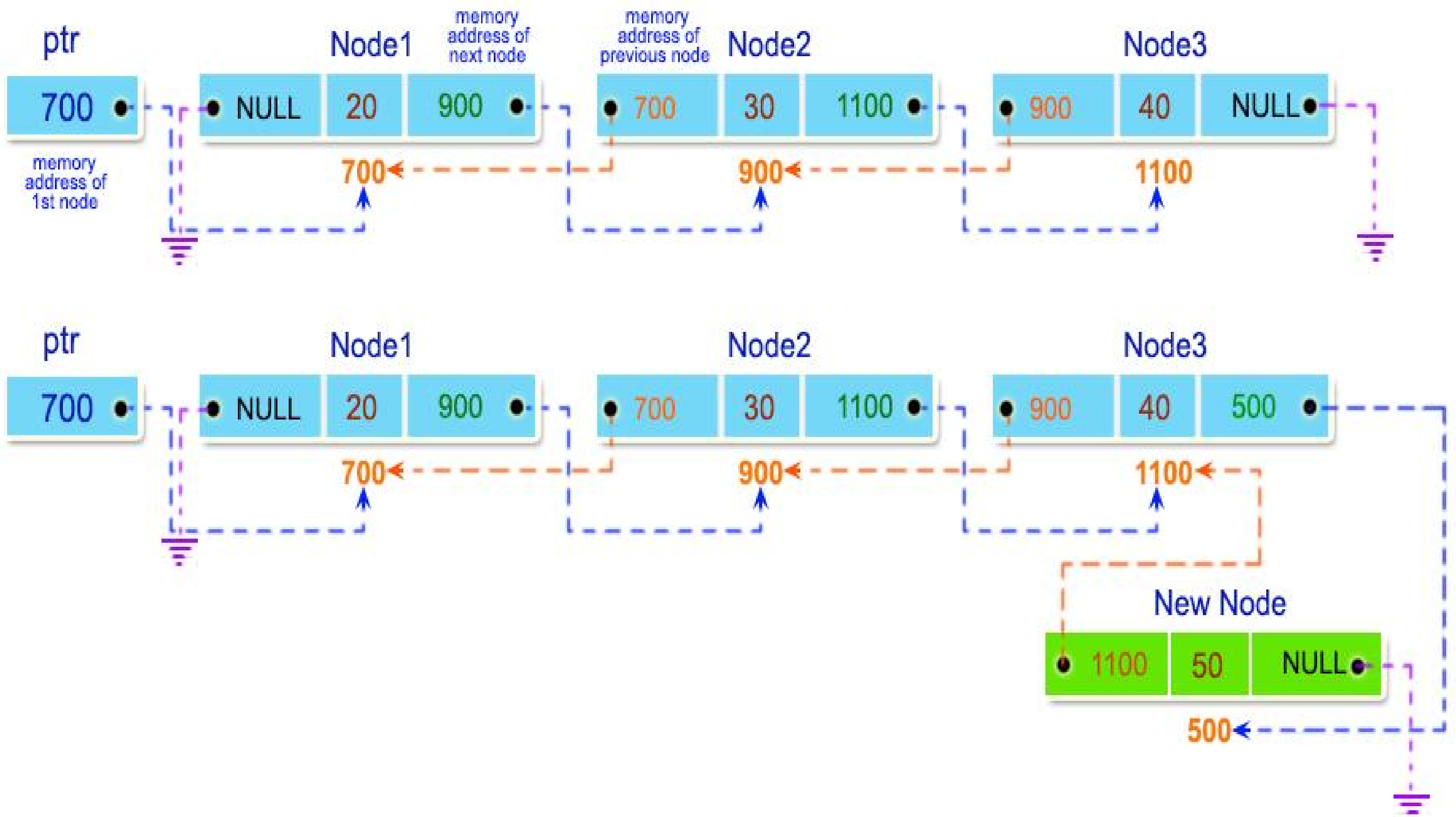
Inserting at Beginning in a DLL



Inserting at Beginning in a DLL

```
void insertAtBegining() {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    printf("\nEnter the new data: ");
    scanf("%d", &newNode->data);
    newNode->next = head;
    new_node->prev = NULL;
    if (head != NULL)
        head->prev = newNode;
    head = newNode;
}
```

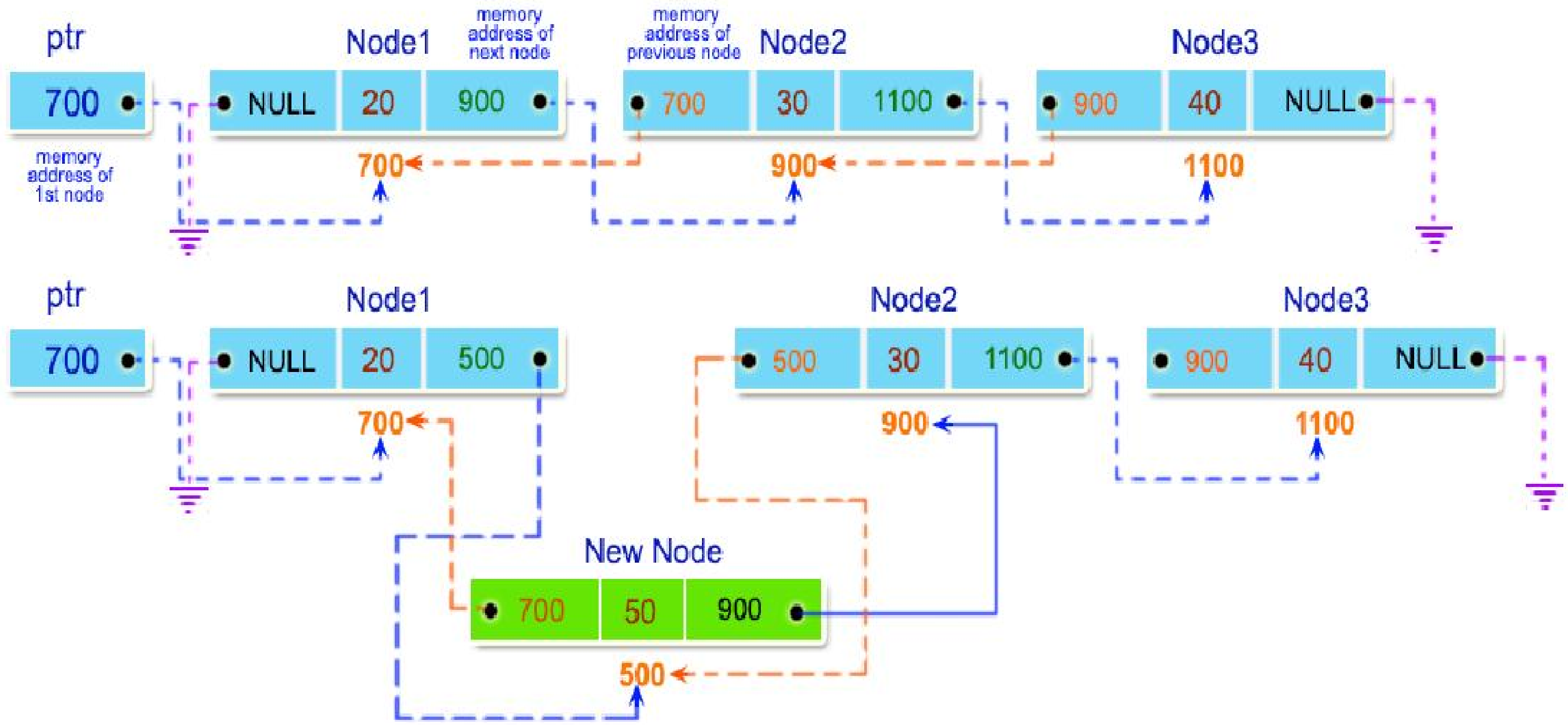
Inserting at End in a DLL



Inserting at End in a DLL

```
void insertAtEnd() {
    struct Node* last = head
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    printf("\nEnter the new data: ");
    scanf("%d", &newNode->data);
    newNode->next = NULL;
    if (head == NULL) {
        newNode->prev = NULL;
        head = newNode;
        return;     }
    while (last->next != NULL)
        last = last->next;
    last->next = newNode;
    newNode->prev = last;
    return; }
```

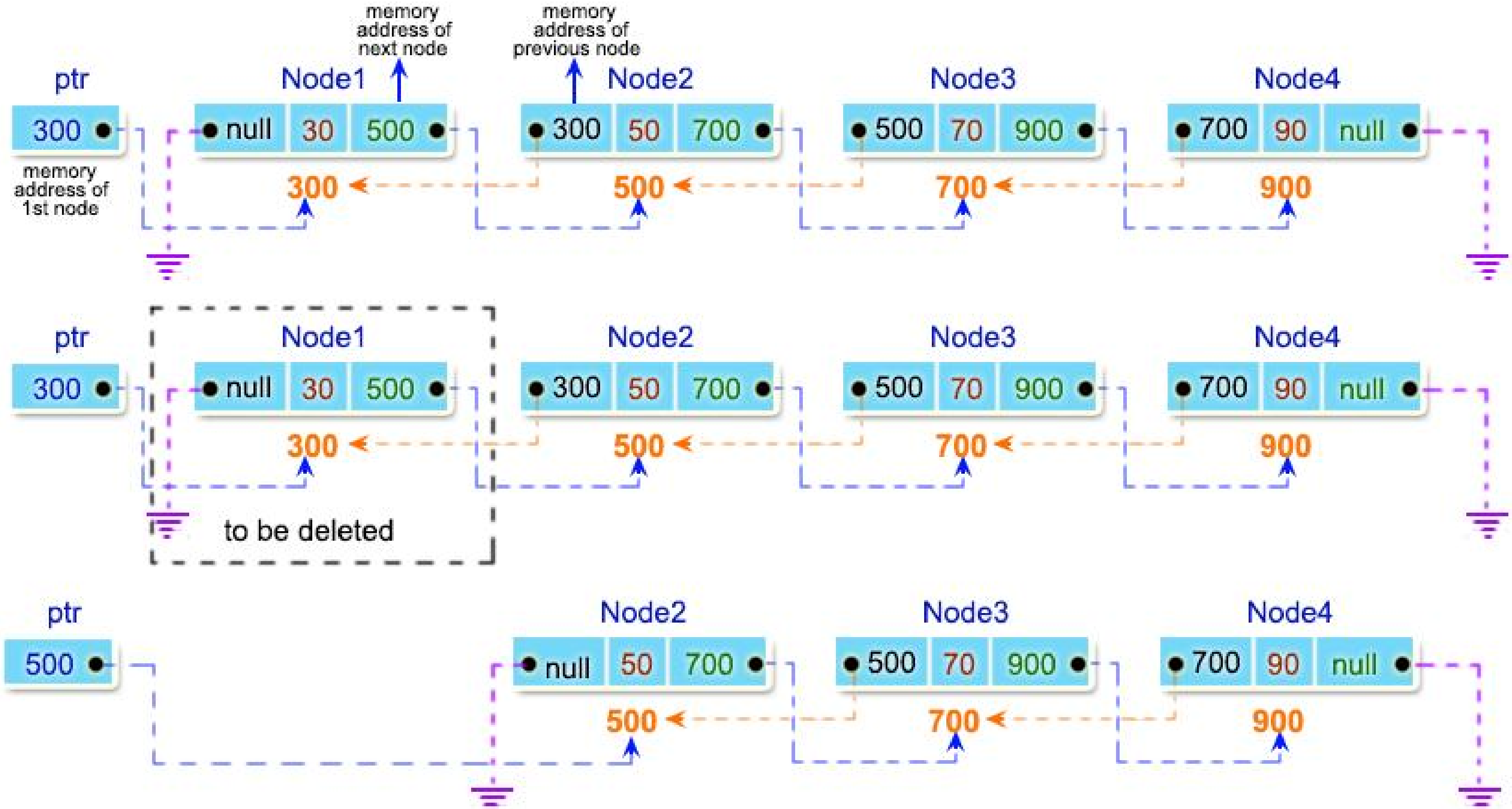
Inserting at Any Position in a DLL



Inserting at Any Position in a DLL

```
void insertAtAnyPosition() {
    int i = 1, pos;
    struct Node* newNode, *curr;
    curr = head;
    if(head == NULL) {
      printf("\nEmpty list...");
      return;
    }
    printf("\nEnter the position at which it will
                be inserted: ");
    scanf("%d", &pos);
    if(pos == 1) {insertAtBeginning(); return;}
    while(i< pos-1 && curr!=NULL)  {
        curr = curr->next;
        i++;     }
    if(curr->next == NULL) { insertAtEnd();
        return;}
    if(curr != NULL)  {
      newNode = (struct Node*)
                    malloc(sizeof(struct Node));
      printf("\nEnter the new data: ");
      scanf("%d", &newNode->data);
      newNode->next = curr->next;
      newNode->prev = curr;
      if(curr->next != NULL) curr->next->prev
                                = newNode;
      curr->next = newNode;
    }
    else printf("Invalid position...\n");   }
```

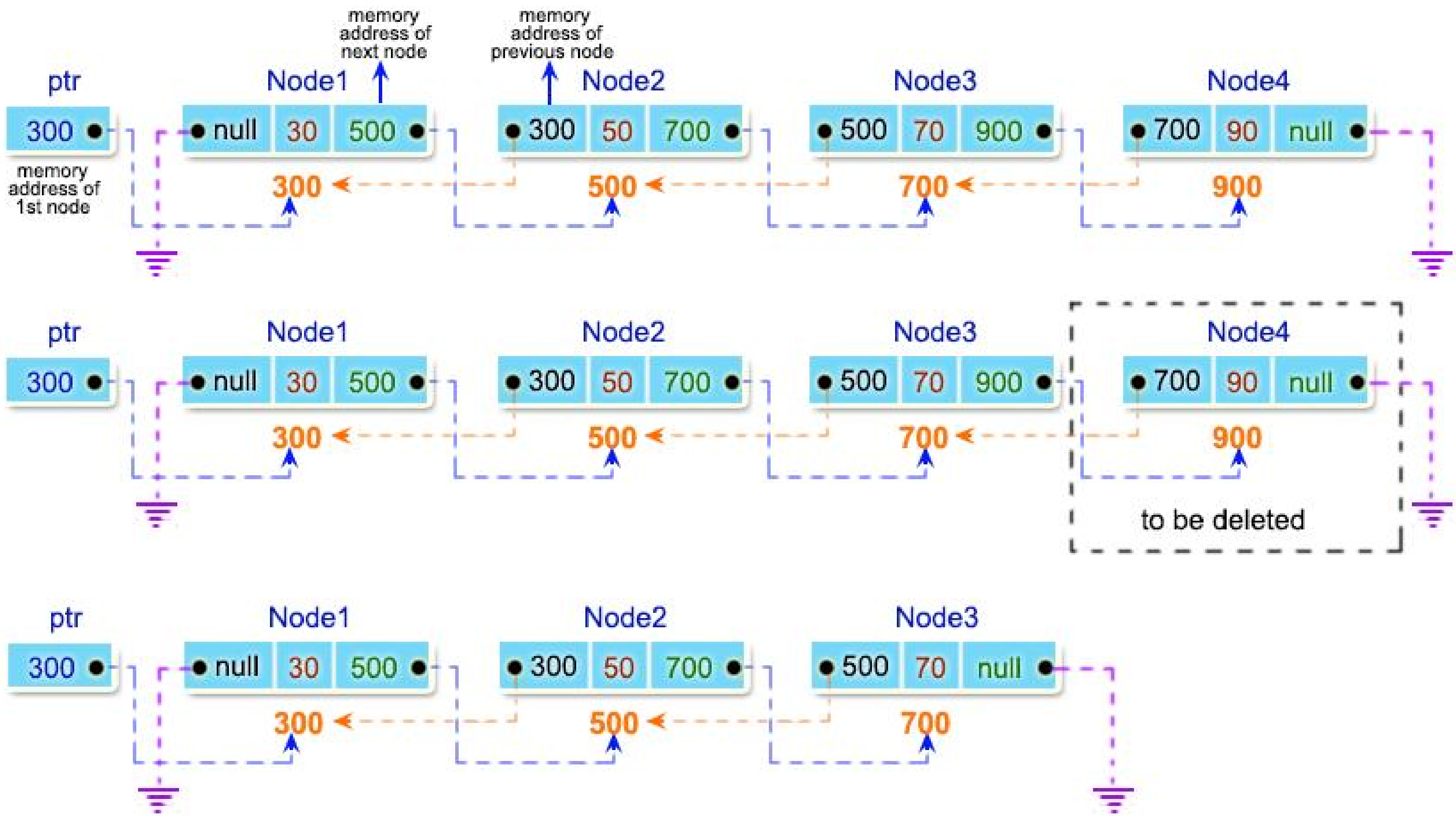
Delete node from the beginning of a double linked list



Delete node from the beginning of a double linked list

```
void deleteATBeginning()  {
    struct Node *temp;
    if(head == NULL) {
        printf("Empty List, Delete is not possible...\n");
        return;
    }
    temp = head;
    head = head->next;
    if(head!=NULL) head->prev = NULL;
    free(temp);
}
```

Delete node from the end of a double linked list

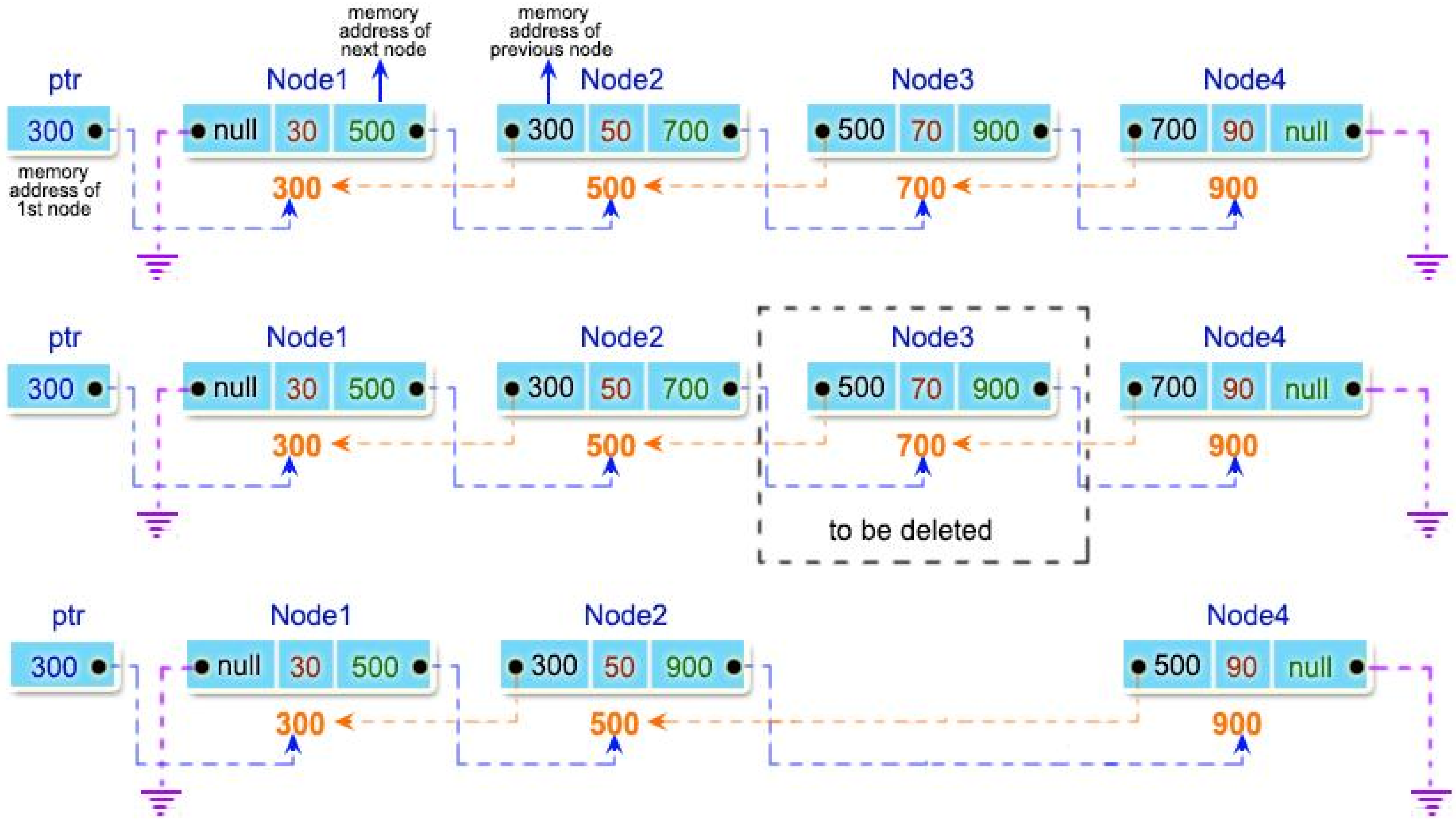


Delete node from the end of a double linked list

```
void deleteAtEnd() {
        struct Node *temp;
        if(head == NULL) {
                printf("Empty List, Delete is not possible...\n");
                return;
        }
        temp = head;

        while(temp->next != NULL) temp=temp->next;
       if(temp->prev != NULL) temp->prev->next = NULL;
        else head = NULL;
       free(temp);
}
```

Delete node from a position of a double linked list



Delete node from a position of a double linked list

```
void deleteAtPosition(int pos) {
	struct Node *temp;
	int i;
	if(head == NULL) {
		printf("Empty List, Delete is not possible...\n");
		return;
	}
	if(pos == 1) deleteAtBeginning();
	temp = head;
	i=1;
	while(i<pos && temp!=NULL)  {
		i++;
		temp=temp->next;
	}
	if(temp==NULL)  printf("Invalid Position...\n");
	else if(temp->next == NULL) deleteAtEnd();
	else  {
		temp->prev->next = temp->next;
		temp->next->prev = temp->prev;
		free(temp);
	}
}
```

Delete all the even nodes from a double linked list

```
void deleteEvenNodes() {
    struct Node* curr = head;
    struct Node* nxt;
    if(head==NULL) {
        printf("Empty List, Invalid deletion...\n");
        return;
    }
    while (curr != NULL) {
        nxt = curr->next;
        if (curr->data % 2 == 0)  {
            // If node to be deleted is head node
            if (head == curr)
                head = curr->next;
            // if node to be deleted is NOT the last node
            if (curr->next != NULL)
                curr->next->prev = curr->prev;
            // if node to be deleted is NOT the first node
            if (del->prev != NULL)
                del->prev->next = del->next;
            free(curr);
        }
        curr = nxt;
    }
}
```

Circular Single Linked List

Why Circular?

- In a single linked list, for accessing any node of a linked list, always traverse from the first node.
- If reached at any node in the middle of the list, then it is not possible to access nodes that precede the given node.
- This problem can be solved by slightly altering the structure of single linked list.
- In a single linked list, next part of the last node is NULL
- If this link points to the first node then it can reach preceding nodes.

Circular Single Linked List

Insertion:

- A node can be added in three ways:
- Insertion in an empty list
- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

Insertion in an empty List

```
void insertToEmptyList() {
    if (head != NULL)
      return;
    struct Node *curr = (struct Node*)malloc(sizeof(struct Node));
    printf("\nEnter the data: ");
    scanf("%d", &curr->data);
    curr -> next = curr;
    head = last = curr;
}
```

Insertion at the beginning

```
void insertAtBeginning() {
  if (last == NULL) {
    insertToEmptyList();
    return;
  }
  struct Node *curr = (struct Node *)malloc(sizeof(struct Node));
  printf("\nEnter the data: ");
  scanf("%d", &curr->data);
  curr -> next = last -> next;
  last -> next = curr;
  head = curr;
}
```

Insert at End

```
void insertAtEnd() {
  struct Node *newNode = (struct Node*)
                  malloc(sizeof(struct Node));
  printf("\nEnter the data: ");
  scanf("%d", &newNode->data);
  if (last == NULL) {
    newNode -> next = newNode;
    head = last = newNode;
  }
  newNode -> next = last -> next;
  last -> next = newNode;
  last = newNode;
}
```

```
void insertAtEnd() {
  struct Node *curr;
  struct Node *newNode = (struct Node*)
                  malloc(sizeof(struct Node));
  printf("\nEnter the data: ");
  scanf("%d", &newNode->data);
  if (head == NULL) {
    newNode->next = newNode;
    head = newNode;
  }
  curr = head;
  while(curr->next != head)
     curr = curr->Next;
  newNode->next = curr->next;
  curr->next = newNode;  }
```

Header Linked List

- A header node is a special node that is found at the beginning of the list.
- A list that contains this type of node, is called the header-linked list.
- This type of list is useful when information other than each node value is needed.
- For example, suppose there is an application in which the number of nodes in a list is often calculated.
  - Usually, a list is always traversed to find the length of the list.
  - However, if the current length is maintained in an additional header node that information can be easily obtained.

Create a Header Linked List

```
void createHeaderList() {
    struct node *newNode, *curr;
    newNode = (struct Node*) malloc(sizeof(struct Node));
    sacnf("%d", &newNode->data);
    newNode->next = NULL;
    if (start == NULL) {
        start = (struct Node*) malloc(sizeof(struct Node));
        start->next = newNode;
    }
    else {
        curr = start->next;
        while (curr->next != NULL)
            curr = curr->next;
        curr->next = newNode;
    }
}
```

Display a Header Linked List

```
void display() {
    struct Node* curr;
    curr = start->next;
    while (curr != NULL) {
        printf("%d ", curr->data);
        curr = curr->next;
    }
}
```