Linked Lists

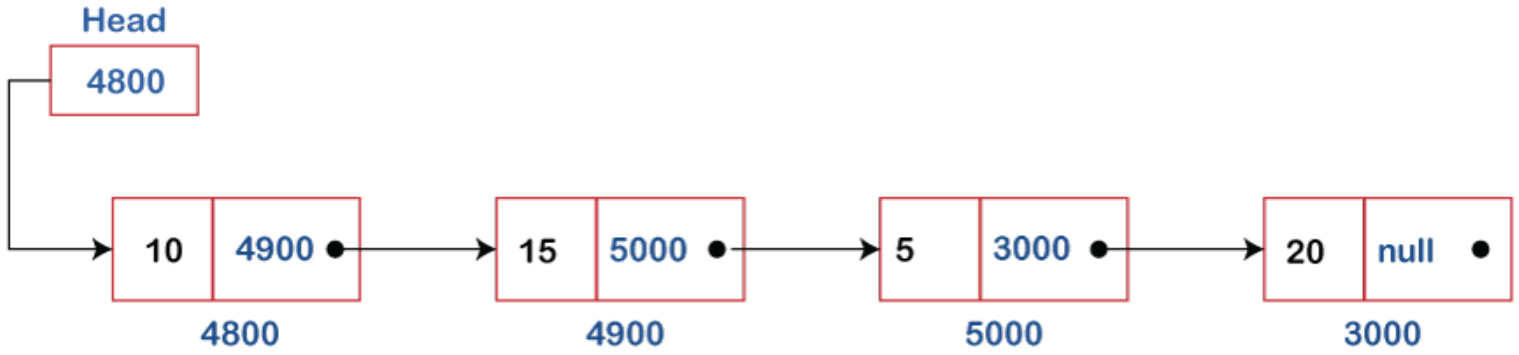
By Asst Prof Christopher Uz

Course: DSAA

Pillai College of Engineering

What is Linked List?

- A linked list can also be defined as the collection of the nodes in which one node is connected to another node
- Node consists of two parts; one is the data part and the second one is the address part
- Advantages: Dynamic DS, memory efficient
- Disadvantages: Traversal, Reverse traversing(doubly linked list but needs more memory)



Array v/s Linked List

Array

- An array is a collection of elements of a similar data type
- Array elements store in a contiguous memory location
- Array works with a static memory
- Array takes more time while performing any operation like insertion, deletion, etc.
- Accessing any element in an array is faster as the element in an array can be directly accessed through the index

Linked List

- A linked list is a collection of objects known as a node where node consists of two parts, i.e., data and address
- Linked list elements can be stored anywhere in the memory or randomly stored
- The Linked list works with dynamic memory
- Linked list takes less time while performing any operation like insertion, deletion, etc.
- Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list

Array v/s Linked List…

Array

- Array elements are independent of each other
- In the case of an array, memory is allocated at compile-time
- Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused

Linked List

- Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node
- In the case of a linked list, memory is allocated at run time
- Memory utilization is efficient in the case of a linked list as the memory can be allocated or deallocated at the run time according to our requirement

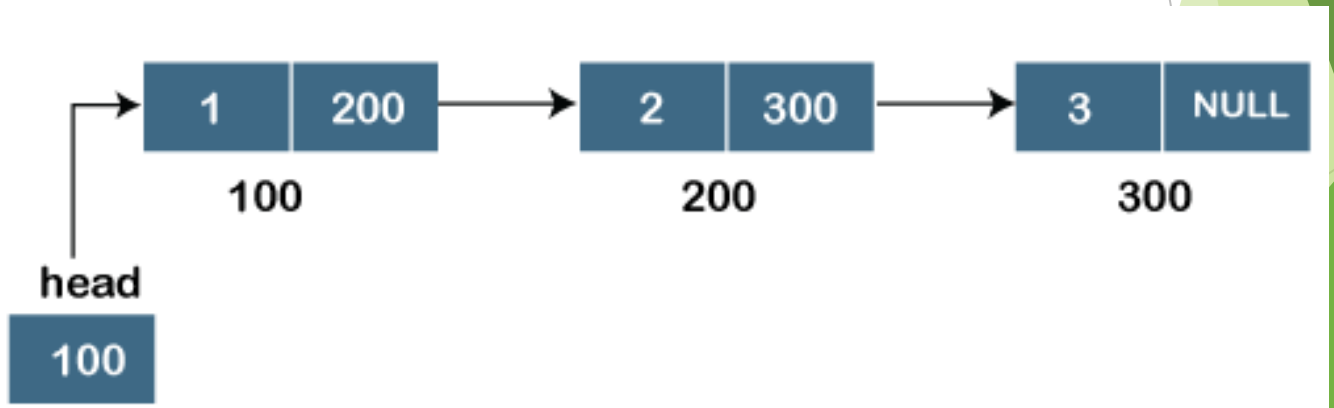
Types of Linked List

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Doubly Circular Linked List

Singly Linked List

- Regular linked list is also known as singly linked list
- The pointer that holds the address of the initial node is known as a **head pointer**
- Only forward traversal is possible

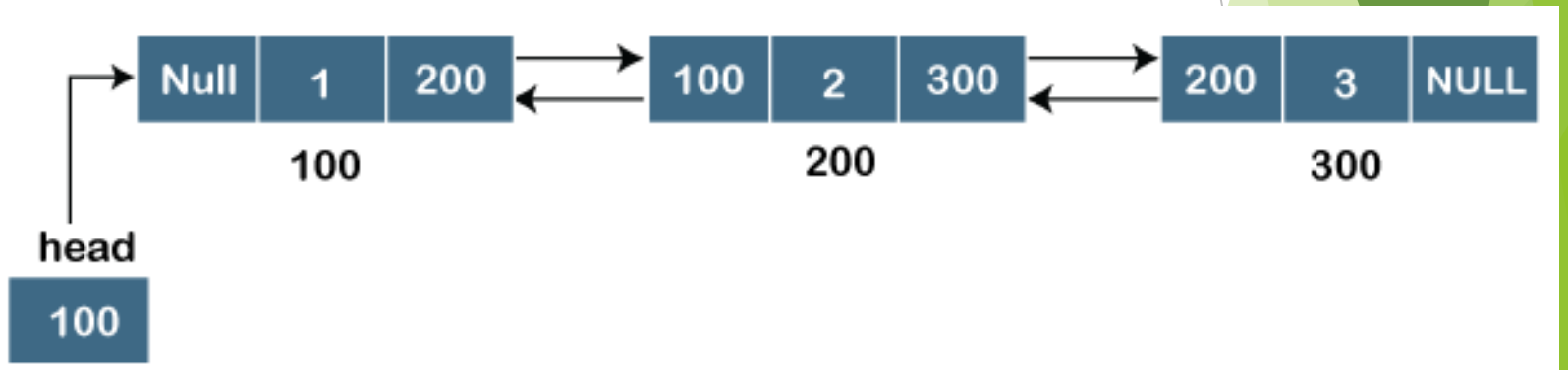
```
struct node
{
   int data;
   struct node *next;
}
```



Doubly Linked List

- The doubly linked list contains **two pointers**
- Contains three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.
- Both forward and backward traversal is possible

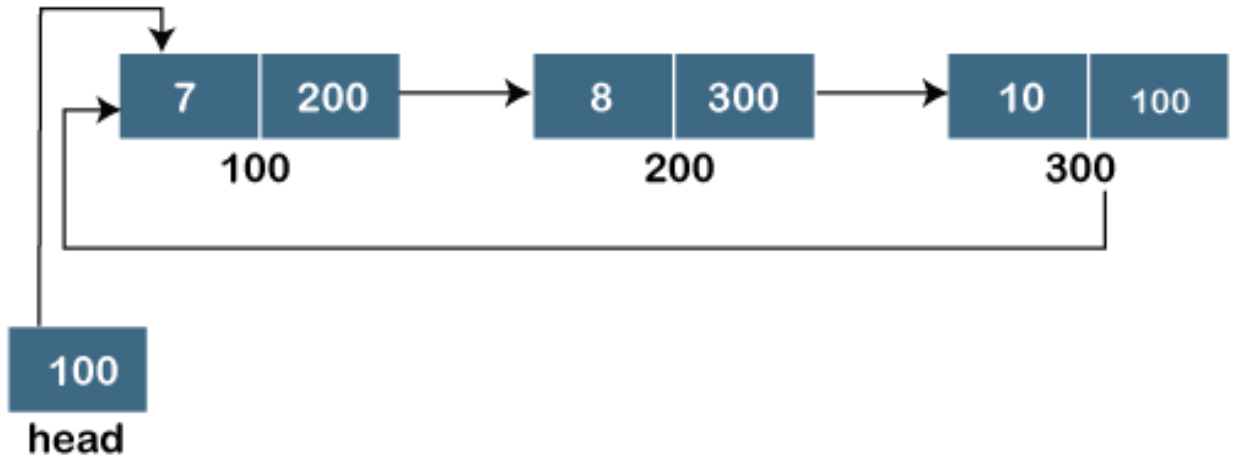
```
struct node
{
  int data;
  struct node *next;
  struct node *prev;
}
```



Circular Linked List

- It is a variation of a singly linked list
- The circular linked list is a list in which the last node connects to the first node
- The circular singly liked list has no beginning and no ending
- No null value present in the next part of any of the nodes

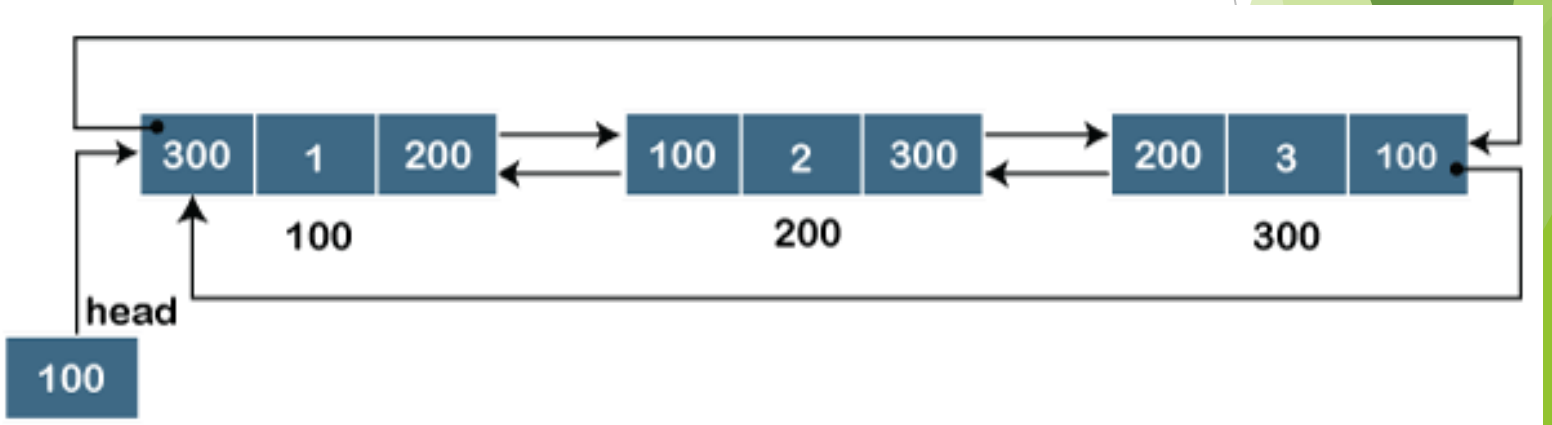
```
struct node
{
   int data;
   struct node *next;
}
```



Doubly Circular Linked List (Extra)

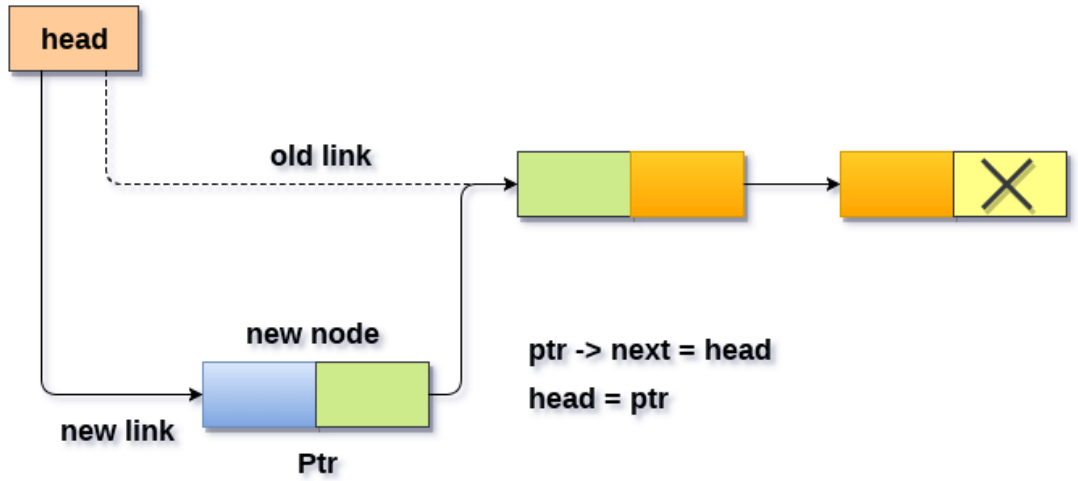
- It has the features of both the circular linked list and doubly linked list
- The last node is attached to the first node and thus creates a circle
- Also, each node holds the address of the previous node

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



Insert at beginning (Singly Linked List)

- Allocate memory and store data for new node
- Change next of new node to point to head
- Change head to point to recently created node

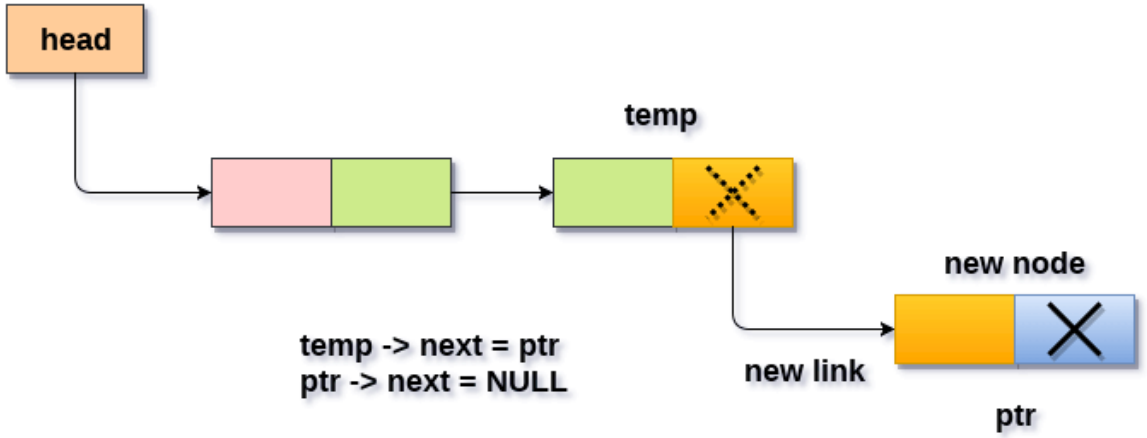


Insert at beginning (Singly Linked List)…

```
struct node *temp;
temp = (struct node *)malloc(sizeof(struct node));
temp->info = item;
temp->next = NULL;
if(head == NULL) {
        head = temp;
}
else {
        temp->next = head;
        head = temp;
}
```

Insert at ending (Singly Linked List)

- Allocate memory and store data for new node
- Traverse to last node
- Change next of last node to recently created node

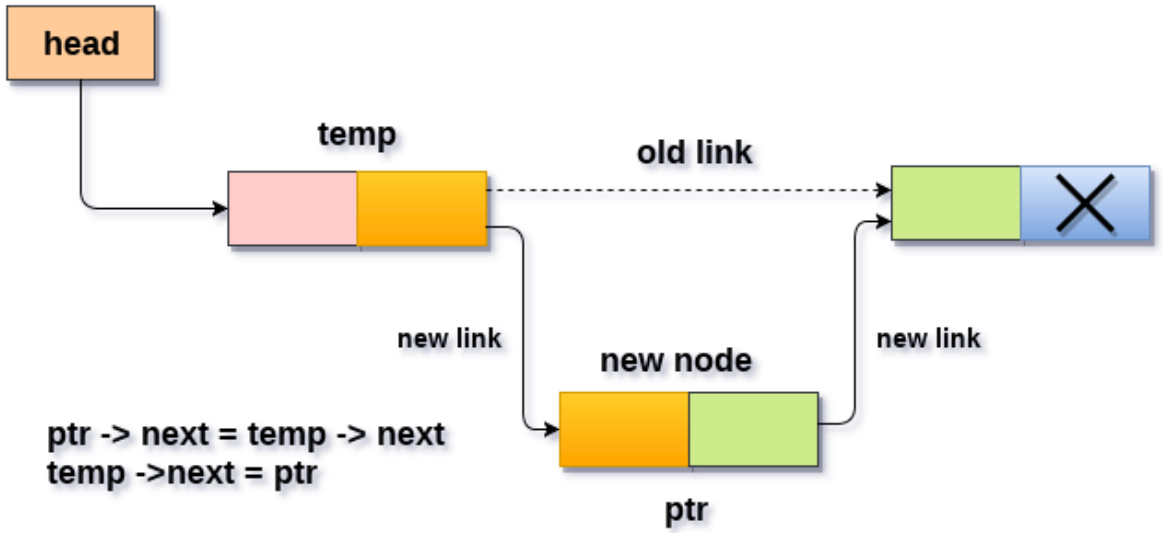


Insert at ending (Singly Linked List)…

```
struct node *temp,*ptr;
temp = (struct node *) malloc(sizeof(struct node));
temp->info = item;
temp->next = NULL;
if(head == NULL)
{
    head = temp;
}
else
{
    ptr = head;
    while(ptr->next != NULL)
    {
        ptr = ptr->next;
    }
    ptr->next = temp;
}
```

Insert at position (Singly Linked List)

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

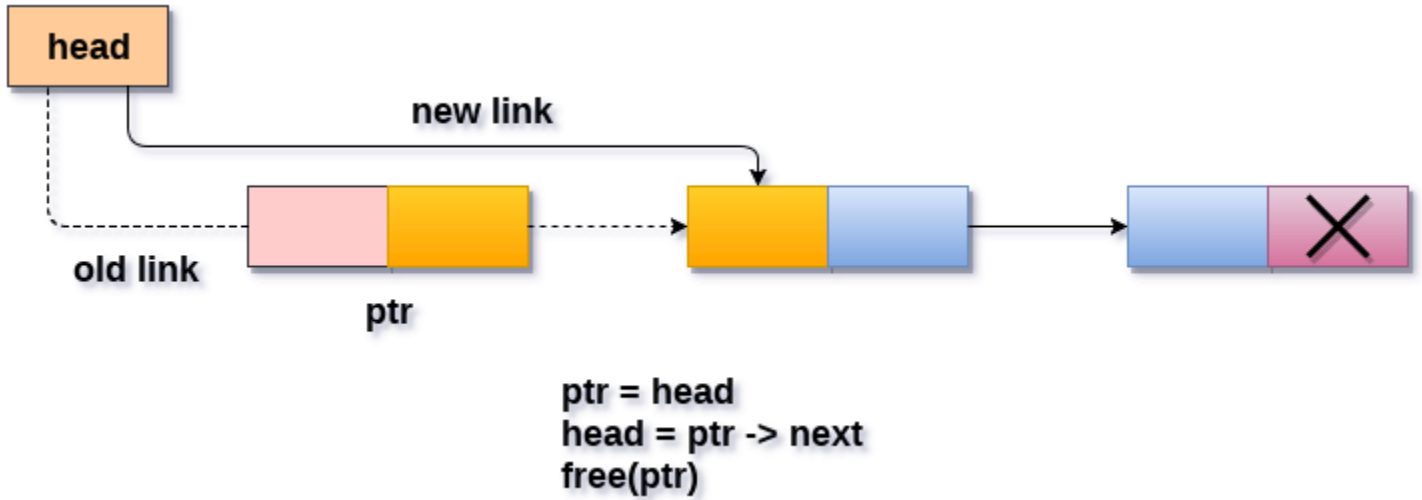


Insert at position (Singly Linked List)…

```
struct node *ptr,*temp;
temp = (struct node *) malloc(sizeof(struct node));
// Use scanf to get user i/p
pos = someposition;
temp->info = somevalue;
temp->next = NULL;

if(pos == 1) {
        temp->next = head;
        head = temp;
}
else {
        for(i=1,ptr=head;i<pos-1;i++) {
                ptr = ptr->next;
                if(ptr == NULL) {
                        printf("Position not found\n");
                        return;
                }
        }
        temp->next = ptr->next ;
        ptr->next = temp;
}
```

Delete at beginning (Singly Linked List)

- Point head to its next node
- Free the previous first node

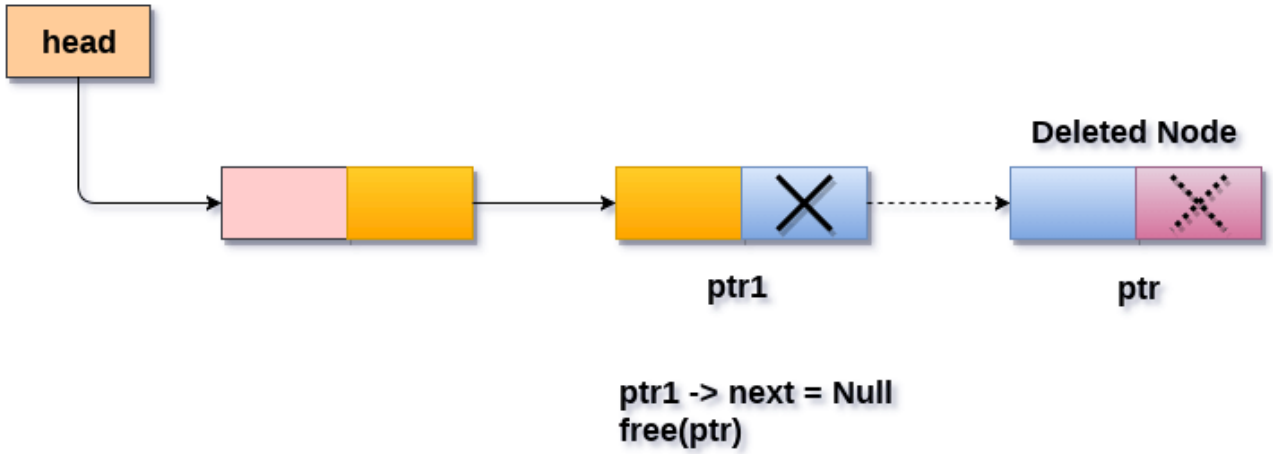


Delete at beginning (Singly Linked List)…

```
struct node *ptr;
if(head == NULL)
{
        printf("List is Empty:\n");
        return;
}
else
{
         ptr = head;
         head = head->next ;
         free(ptr);
}
```

Delete at ending (Singly Linked List)

- Traverse to second last node
- Change its next pointer to null
- Free the previous last node



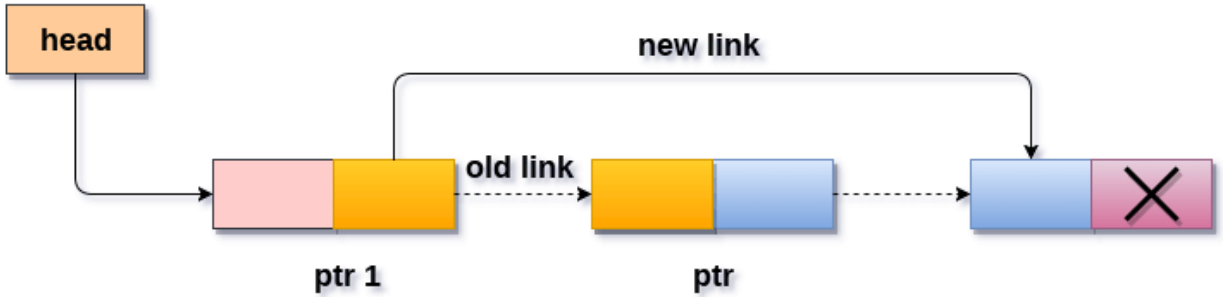
Delete at ending (Singly Linked List)…

```
struct node *temp,*ptr;
if(head == NULL) {
      printf("List is Empty:");
      exit(0);
}
else if(head->next == NULL) {
       ptr = head;
       head = NULL;
       free(ptr);
}
```

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

Delete at position (Singly Linked List)

- Traverse to the node before the node to be deleted
- Change next pointers to exclude the node from the chain
- Free the deleted node




```
ptr1 -> next = ptr -> next
free(ptr)
```

Delete at position (Singly Linked List)…

```
struct node *temp,*ptr;
if(head == NULL) {
        printf("The List is Empty:\n");
        exit(0);
}
else {
        pos = someposition;
        if(pos == 1) {
            ptr = head;
            head = head->next ;
            free(ptr);
        }
```

```
else {
        ptr = head;
        for(i=1;i<pos;i++)  {
            temp = ptr;
            ptr = ptr->next;
            if(ptr == NULL) {
                printf("Position not Found:\n");
                return;
            }
        }
        temp->next = ptr->next ;
        free(ptr);
}}
```

Traversing (Singly Linked List)

```
struct node *ptr;
if(head == NULL) {
        printf("List is empty:\n");
        return;
}
else {
        ptr = head;
        printf("The List elements are:\n");
        while(ptr != NULL) {
                printf("%d\t",ptr->info );
                ptr = ptr->next ;
        }
}
```

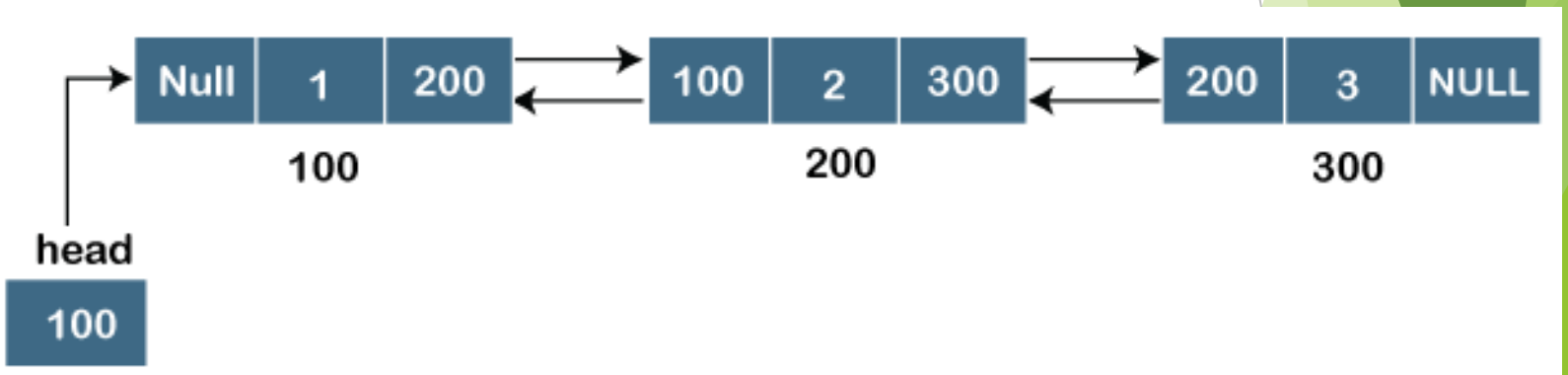
Update (Singly Linked List)

```
update_data(int old, int new) {
    int pos = 1;

    if(head == NULL) {
       printf("Linked List not initialized");
       return;
    }

    current = head;
    while(current->next != NULL) {
        if(current->data == old) {
            current->data = new;
            printf("\n%d found at position %d, replaced with %d\n", old, pos, new);
            return;
        }
        current = current->next;
        pos++;
    }
    printf("%d does not exist in list\n", old);
}
```

Doubly Linked List

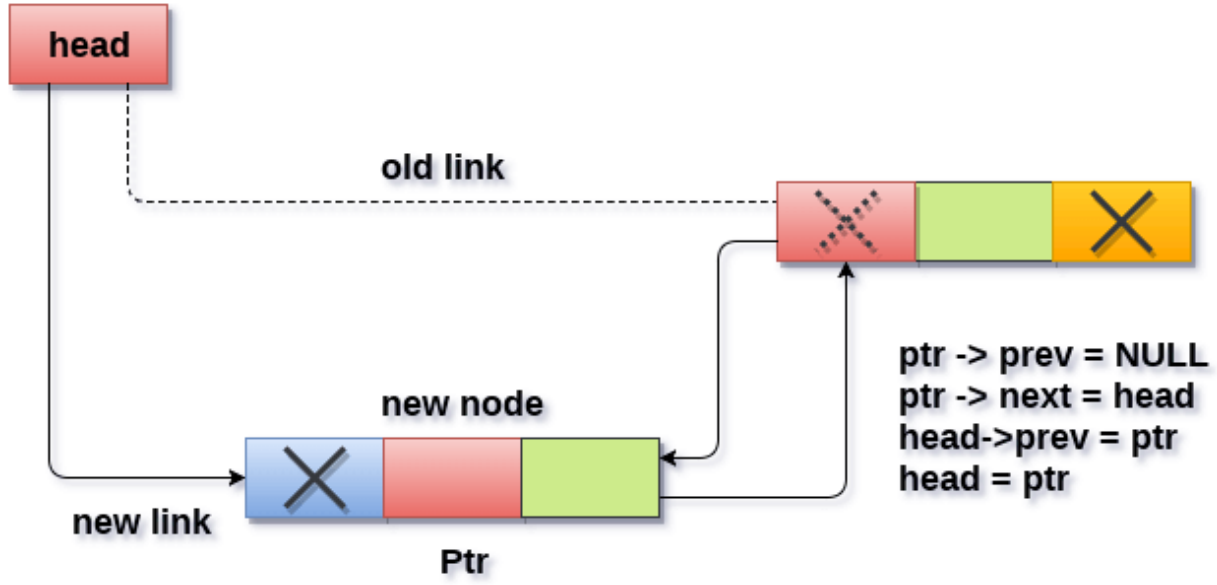
- The doubly linked list contains **two pointers**
- Contains three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.
- Both forward and backward traversal is possible

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



Insert at beginning (Doubly Linked List)

- Allocate memory and store data for new node
- Set prev and next pointers of new node
- Make new node as head node and point prev of the first node to new node

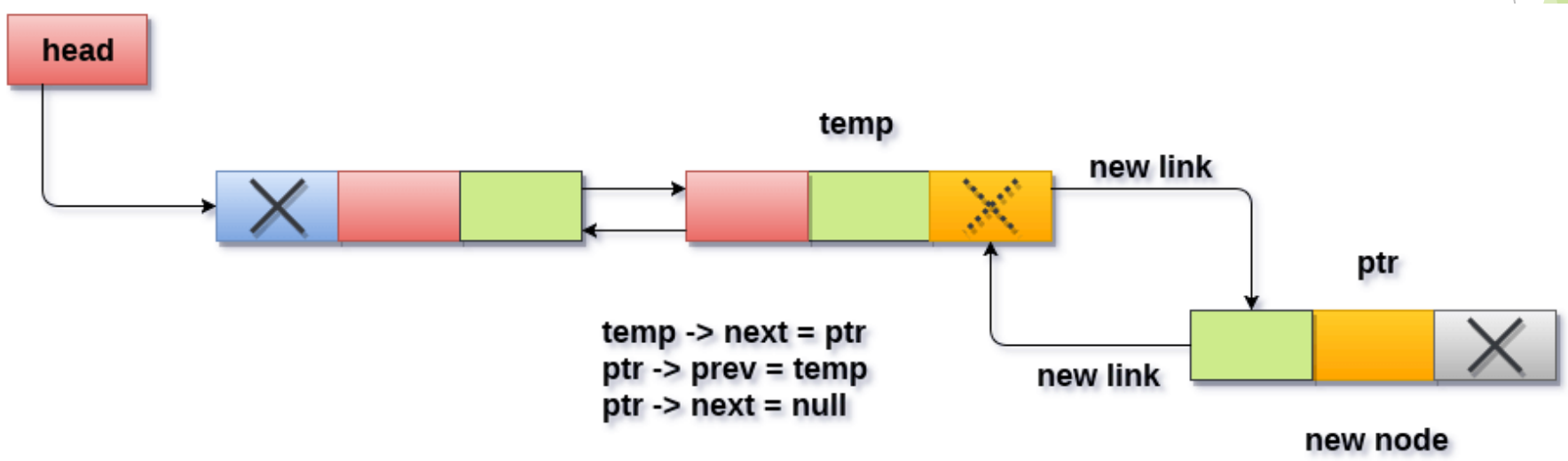


Insert at beginning (Doubly Linked List)…

```
struct node *ptr = (struct node *)
malloc(sizeof(struct node));
ptr->data = item;
if(head == NULL)
{
   ptr->next = NULL;
   ptr->prev = NULL;
   head = ptr;
}
else
{
   ptr->prev = NULL;
   ptr->next = head;
   head->prev = ptr;
   head = ptr;
}
```

Insert at ending (Doubly Linked List)

- Allocate memory and store data for new node
- Set prev and next pointers of new node and the current last node



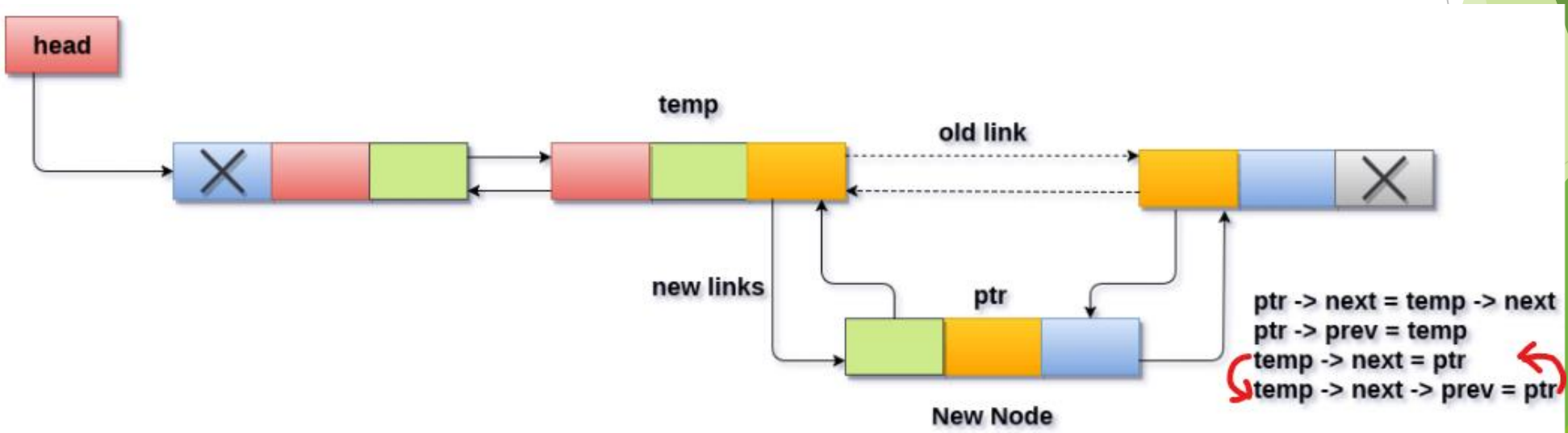
Insert at ending (Doubly Linked List)…

```
struct node *ptr = (struct node *)
malloc(sizeof(struct node));
struct node *temp;
ptr->data = item;

if(head == NULL) {
        ptr->next = NULL;
        ptr->prev = NULL;
        head = ptr;
}
```

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

Insert at position (Doubly Linked List)

- Allocate memory and store data for new node
- Set the next pointer of new node and previous node
- Set the prev pointer of new node and the next node



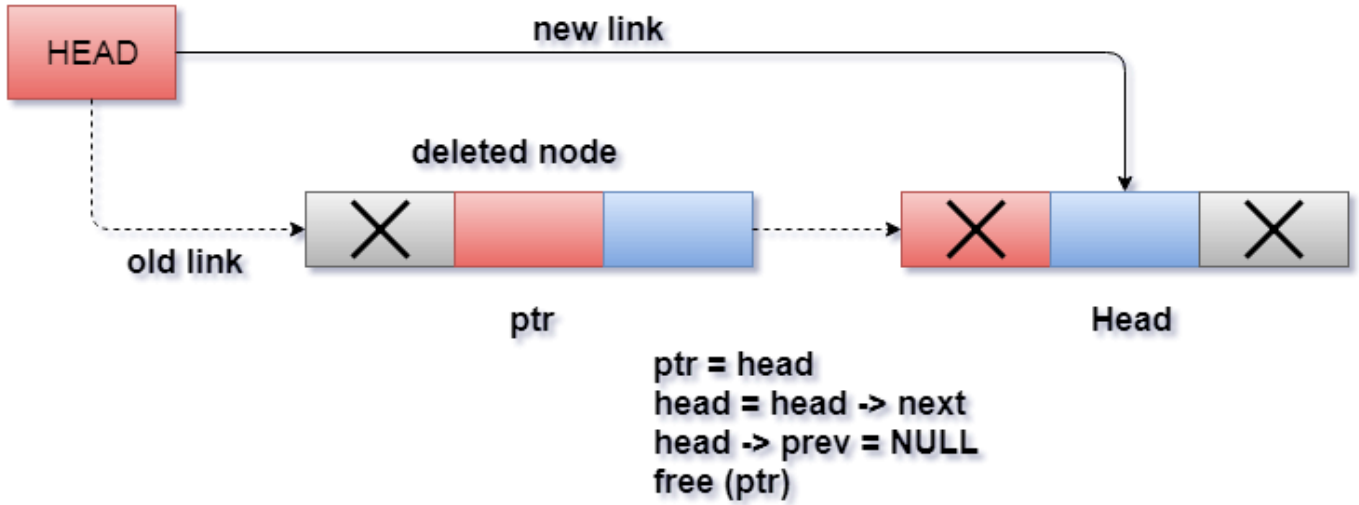
Insert at position (Doubly Linked List)…

```
struct node * newNode, *temp;
temp = head;
i=1;
while(i<position-1 && temp!=NULL) {
        temp = temp->next;
        i++;
}
if(position == 1) {
        referInsertAtBeginningslide(data);
}
else if(temp == last) { //*last points to the last node
        referInsertAtEndSlide(data);
}
```

```
else if(temp!=NULL) {
        newNode = (struct node *)
        malloc(sizeof(struct node));
        newNode->data = data;
        newNode->next = temp->next;
        newNode->prev = temp;
        temp->next->prev = newNode;
        }
        temp->next = newNode;
}
```

Delete at beginning (Doubly Linked List)

- Point head to its next node
- Set prev of this new head node point to NULL
- Free the previous first node

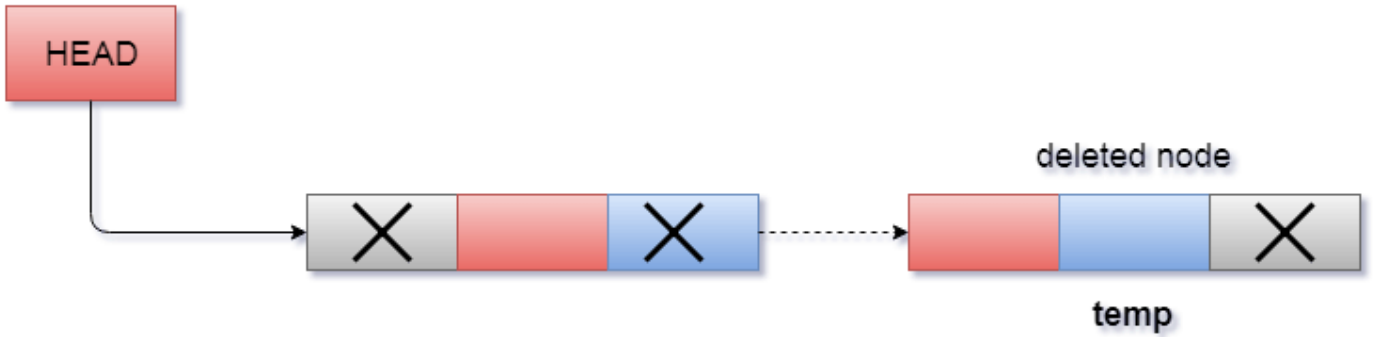


Delete at beginning (Doubly Linked List)…

```
struct node *ptr;
if(head == NULL) {
        printf("Unable to delete. List is empty.\n");
}
else {
        ptr = head;
        head = head->next;
        if (head != NULL)
                head->prev = NULL;
        free(ptr);
}
```

Delete at ending (Doubly Linked List)

- Traverse the list to reach the last node of the list
- Make the next pointer of the second last node to NULL
- Free the last node

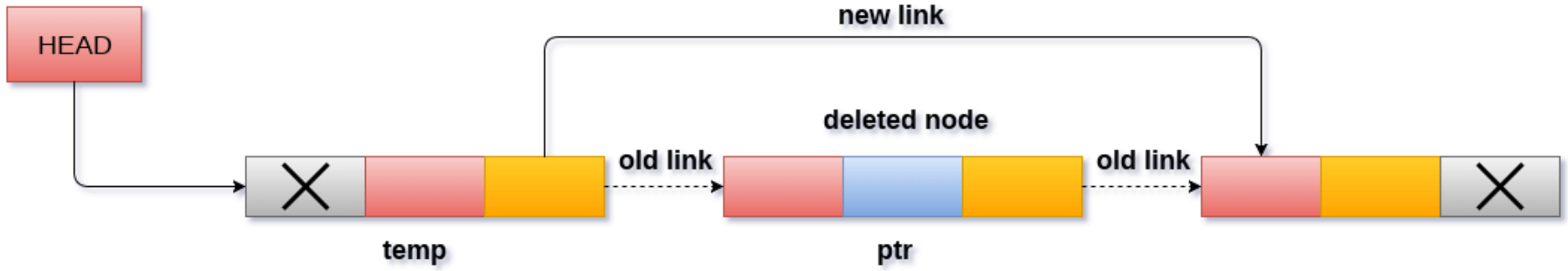


Delete at ending (Doubly Linked List)…

```
struct node *ptr;
if(head == NULL) {
        printf("\n Empty Linked List\n");
}
else if(head->next == NULL) {
        ptr = head;
        head = NULL;
        free(ptr);
}
else {
        ptr = head;
        while(ptr->next != NULL) {
                ptr = ptr -> next;
        }
        ptr -> prev -> next = NULL;
        free(ptr);
}
```

Delete at position (Doubly Linked List)

- Traverse to the node before the node to be deleted
- Change next pointers to exclude the node from the chain
- Free the deleted node




```
temp -> next = ptr -> next
ptr ->next -> prev = temp
free(ptr)
```

Delete at position (Doubly Linked List)…

```
struct node *current;
int i;
current = head;
for(i=1; i<position && current!=NULL; i++)
{
      current = current->next;
}

if(position == 1)
{
      referDeleteFromBeginningSlide();
}
else if(current == last)
{
      referDeleteFromEndSlide();
}
else if(current != NULL)
{
      current->prev->next = current->next;
      current->next->prev = current->prev;
      free(current);   // Delete the n node
}
```

Update (Doubly Linked List)

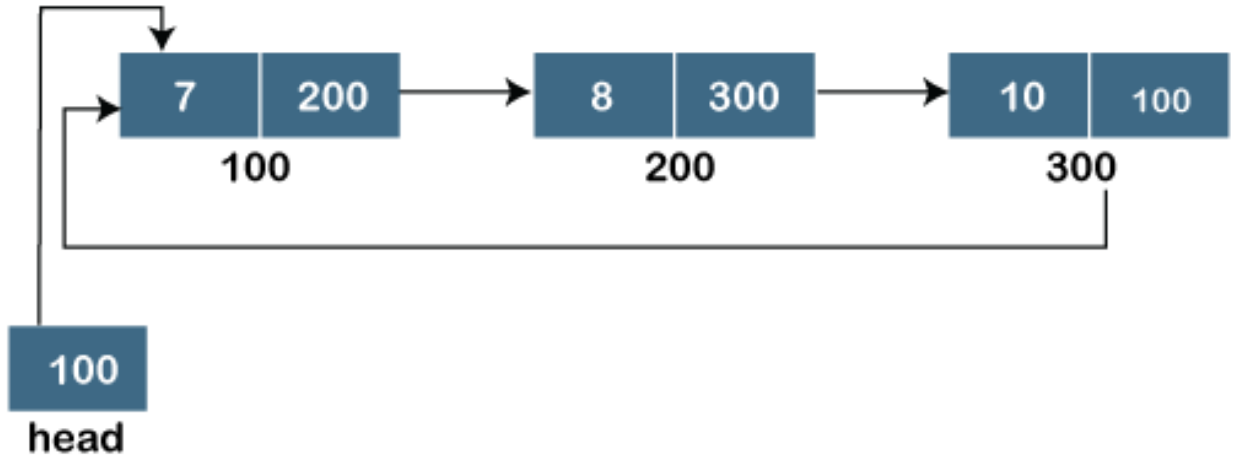
```
update_data(int old, int new) {
    int pos = 0;

    if(head == NULL) {
       printf("Linked List not initialized");
       return;
    }

    current = head;
```

```
while(current != NULL) {
       pos++;
       if(current->data == old) {
          current->data = new;
          printf("\n%d found at position %d, replaced with
%d\n", old, pos, new);
          return;
       }
       if(current->next != NULL)
          current = current->next;
       else
          break;
    }
printf("%d does not exist in list\n", old); }
```

Circular Linked List

- It is a variation of a singly linked list
- The circular linked list is a list in which the last node connects to the first node
- The circular singly liked list has no beginning and no ending
- No null value present in the next part of any of the nodes

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



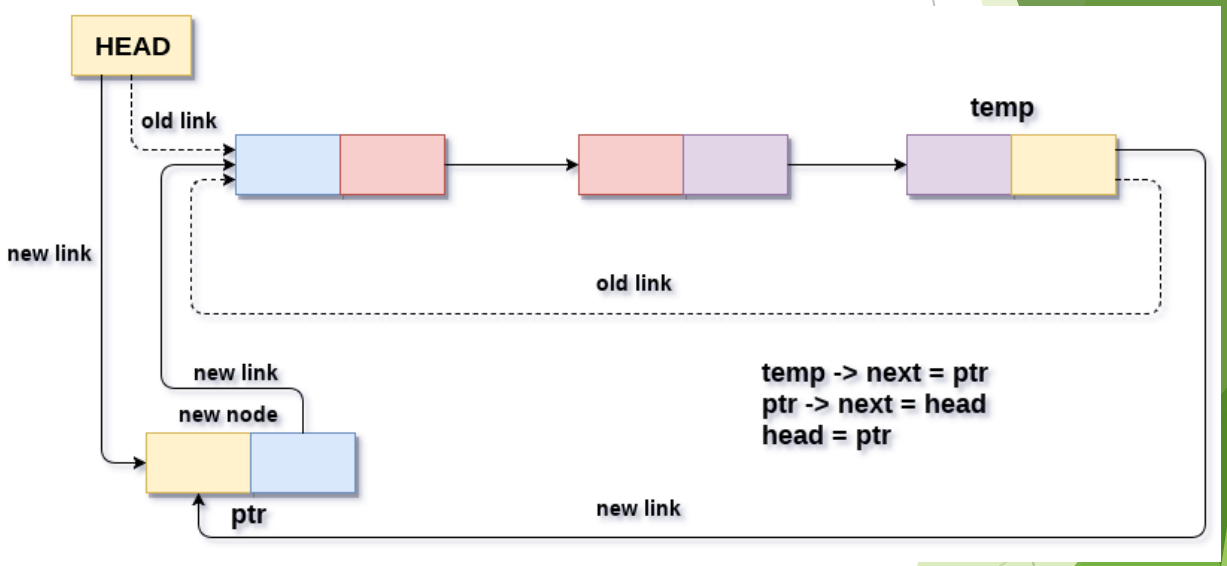
Traverse (Circular Linked List)

```
struct node *current;
int n = 1;

if(head == NULL) {
      printf("List is empty.\n");
}
```

```
else {
      current = head;
      printf("DATA IN THE LIST:\n");

      do {
       printf("Data %d = %d\n", n, current->data);
       current = current->next;
       n++;
      } while(current != head);
}
```

Insert at beginning (Circular Linked List) [Extra Begins]

- Allocate memory and store data for new node
- Traverse the list to reach last node
- Change next pointer of last node to point to new node
- Change the next pointer of new node to head
- Make the new node the new head



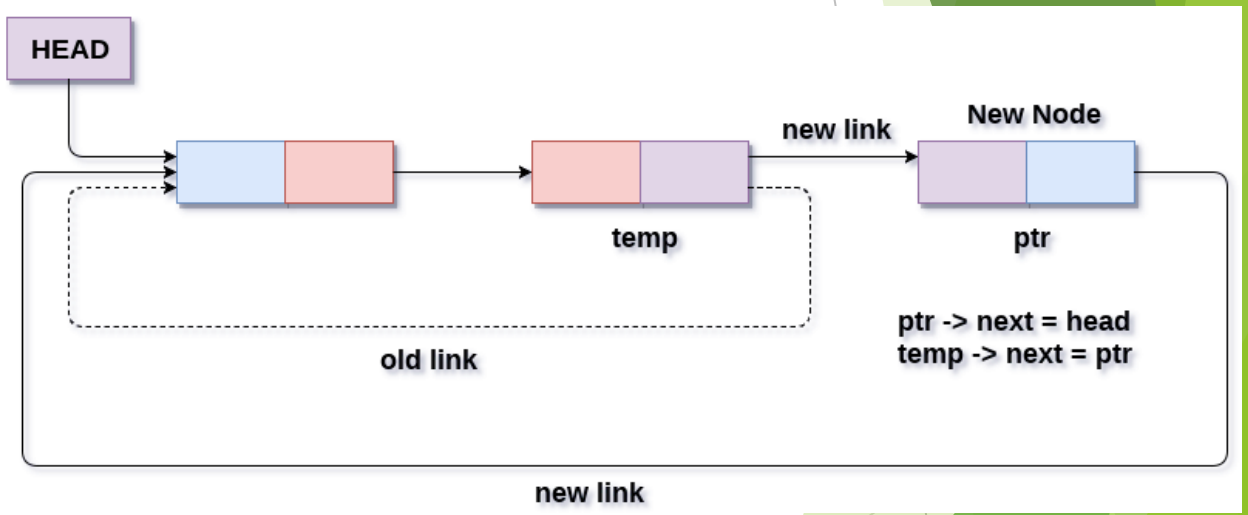
Insert at beginning (Circular Linked List)…

```
struct node *ptr = (struct node *)
malloc(sizeof(struct node));
struct node *temp;
ptr->data = item;

if(head == NULL)
{
      head = ptr;
      ptr->next = head;
}
```

```
else
{
      temp = head;
      while(temp->next != head)
            temp = temp->next;
      temp->next = ptr;
      ptr->next = head;
      head = ptr;
}
```

Insert at ending (Circular Linked List)

- Allocate memory and store data for new node
- Traverse the list to reach last node
- Change next pointer of last node to point to new node
- Change the next pointer of new node to head

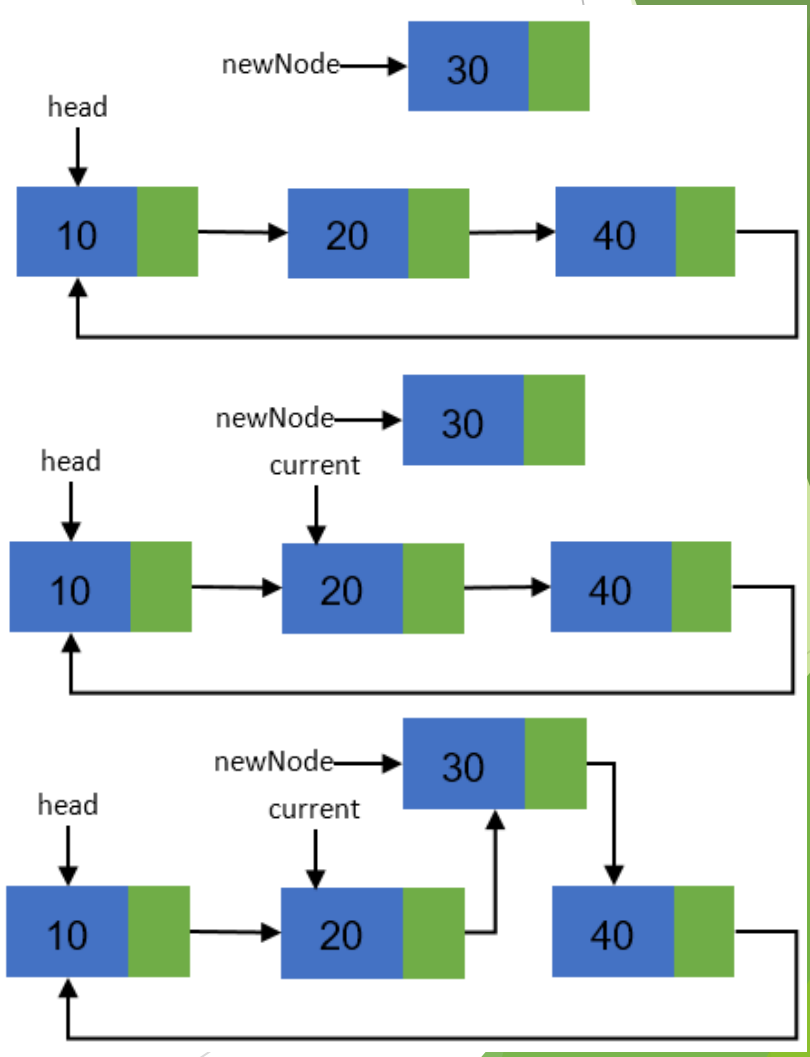


Insert at ending (Circular Linked List)…

```
struct node *ptr = (struct node *)
malloc(sizeof(struct node));
struct node *temp;
ptr->data = item;

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

Insert at position (Circular Linked List)

- Allocate memory and store data for new node
- Traverse the list to reach n-1 position
- Update the next pointer of new node to next pointer of n-1 node
- Change next pointer of n-1 node to point to new node

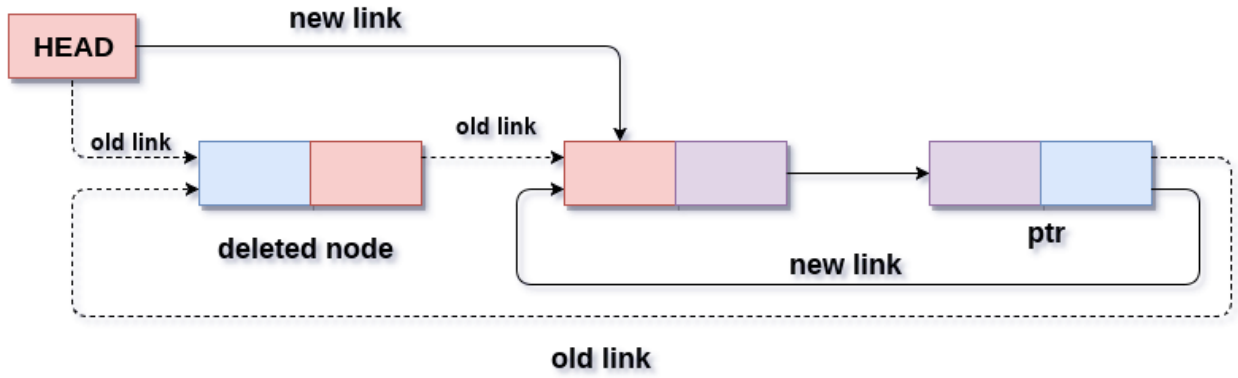


Insert at position (Circular Linked List)…

```
struct node *newNode, *current;
int i;
if(head == NULL) {
        printf("List is empty.\n");
}
else if(position == 1) {
        referInsertAtBeginningSlide(data);
}
else {
        newNode = (struct node *)
            malloc(sizeof(struct node));
        newNode->data = data;
        current = head;
        for(i=2; i<=position-1; i++) {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
}
```

Delete at beginning (Circular Linked List)

- Traverse the list to reach last node
- Change next pointer of last node to point to next of the head
- Free the head pointer
- Update the head to the next pointer of new node



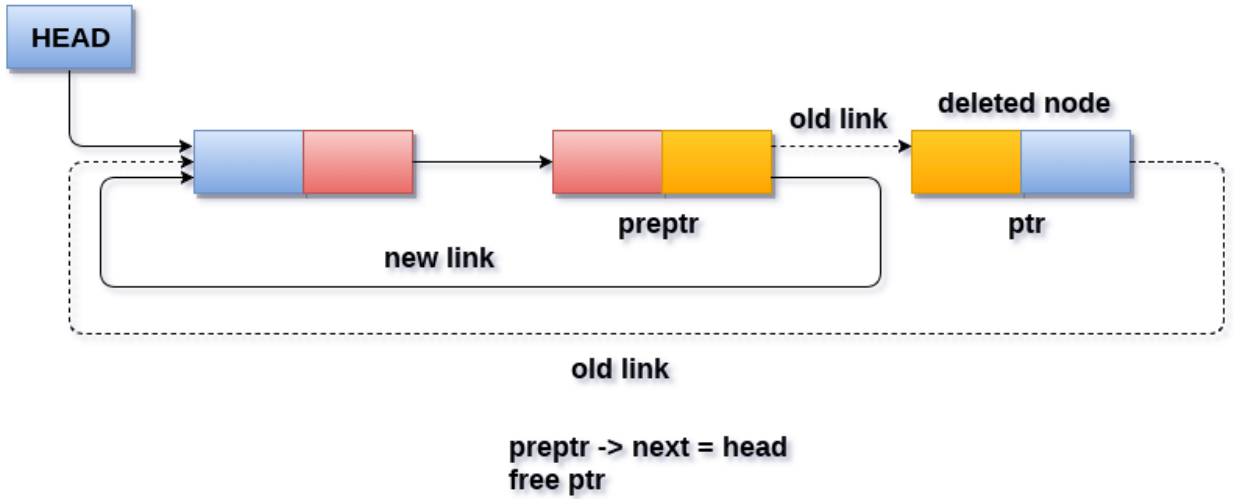

```
ptr -> next = head -> next
free head
head = ptr -> next
```

Delete at beginning (Circular Linked List)…

```
struct node *ptr;
if(head == NULL) {
        printf("\nEmpty Linked List\n");
}
else if(head->next == head) {
        ptr = head;
        head = NULL;
        free(ptr);
}
else {
        ptr = head;
        while(ptr -> next != head)
                ptr = ptr -> next;
        ptr->next = head->next;
        free(head);
        head = ptr->next;
}
```

Delete at ending (Circular Linked List)

- Traverse the list to reach last node
- Change next pointer of second last node to point to head
- Free the last node



Delete at ending (Circular Linked List)…

```
struct node *ptr, *preptr;
if(head == NULL) {
        printf("\nEmpty Linked List\n");
}
else if(head->next == head) {
        ptr = head;
        head = NULL;
        free(ptr);
}
else {
        ptr = head;
        while(ptr->next != head) {
                preptr=ptr;
                ptr = ptr->next;
        }
        preptr->next = ptr->next;
        //preptr->next = head
        free(ptr);
}
```

Update (Circular Linked List) [Extra Ends]

```
update_data(int old, int new) {
    int pos = 1;

    if(head == NULL) {
       printf("Linked List not initialized");
       return;
    }

    current = head;
    while(current->next != head) {
        if(current->data == old) {
            current->data = new;
            printf("\n%d found at position %d, replaced with
%d\n", old, pos, new);
            return;
        }
        current = current->next;
        pos++;
    }
    printf("%d does not exist in list\n", old);
    }
```

Application of Linked List

- Linked Lists can be used to implement Stacks, Queues
- Linked Lists can also be used to implement Graphs (Adjacency list representation of Graph)
- Implementing Hash Tables: Each Bucket of the hash table can itself be a linked list (Open chain hashing)
- Undo functionality in Photoshop or Word. Linked list of states
- A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term
- However, for any polynomial operation, such as addition or multiplication of polynomials, linked list representation is much easier to deal with
- Linked lists are useful for dynamic memory allocation

Polynomial Manipulation

- While representing a polynomial using a linked list, *each polynomial term represents a node* in the linked list
- To get better efficiency in processing, we assume that the term of every polynomial is stored within the linked list in the order of decreasing exponents
- Also, no two terms have the same exponent, and no term has a zero coefficient and without coefficients
- The 3 parts of a node representing polynomial are coefficient, exponent and link
- The structure of a node of a linked list that represents a polynomial is shown below:

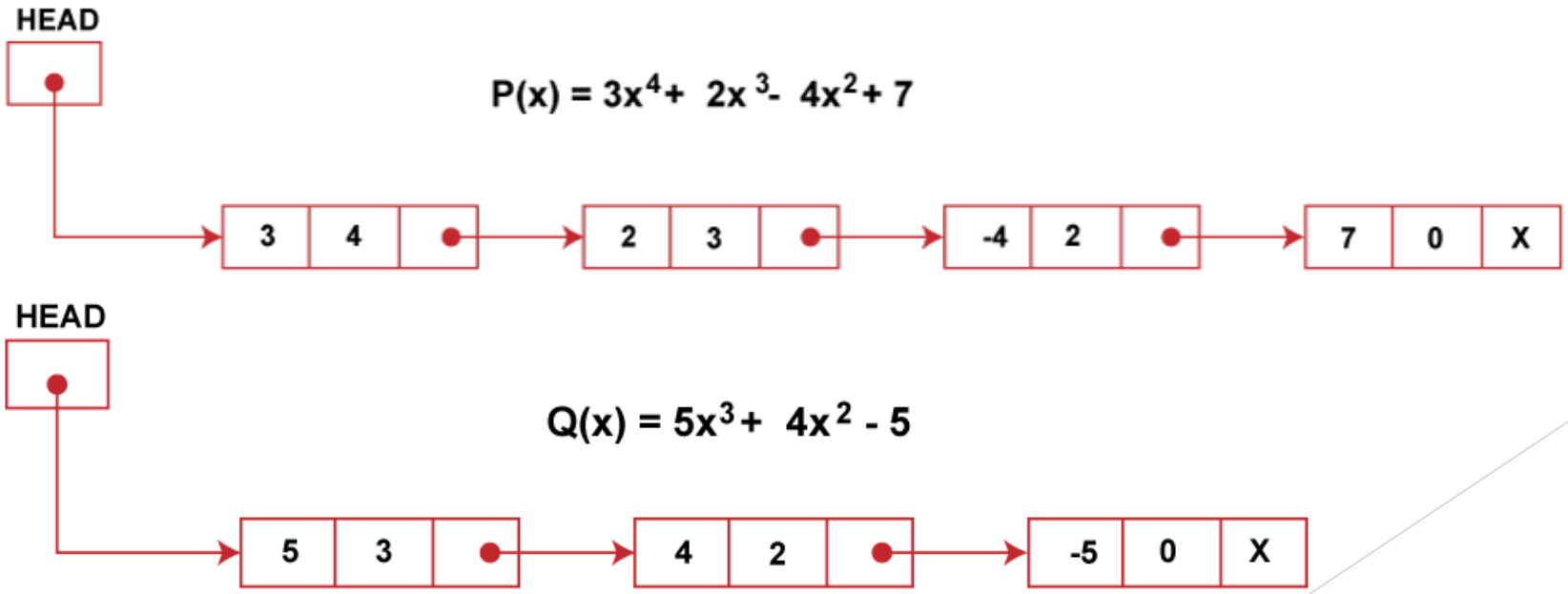
Node representing a term of a polynomial

Polynomial Addition using Linked List (Algorithm)

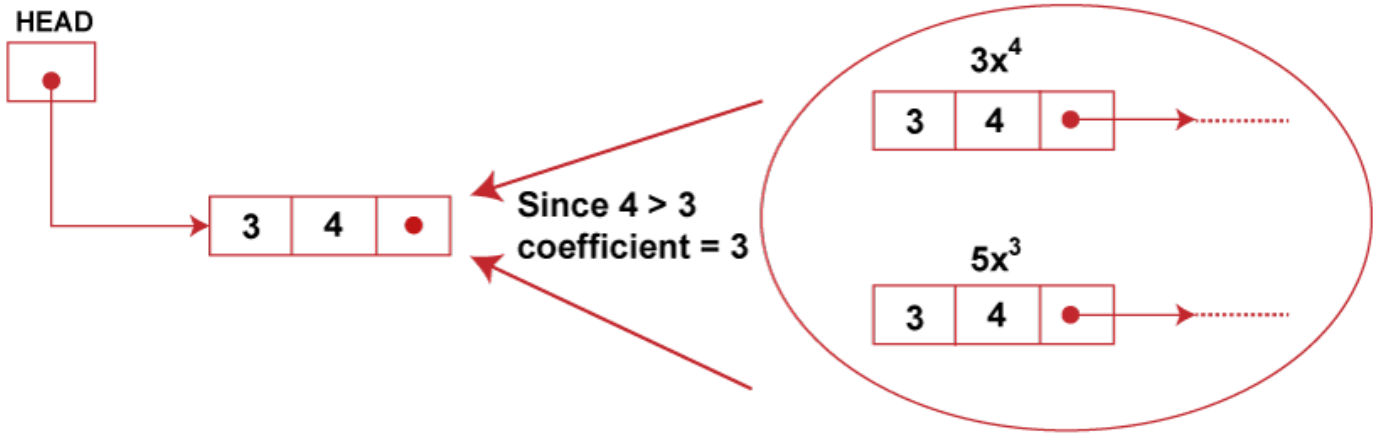
- Step 1: loop around all values of linked list and follow step 2 & 3
- Step 2: if the value of a node's *exponent is greater copy this node to result* node and head towards the next node
- Step 3: if the values of both node's exponent *is same add the coefficients and then copy the added value with node to the result*
- Step 4: Print the resultant node

Polynomial Addition using Linked List Sample

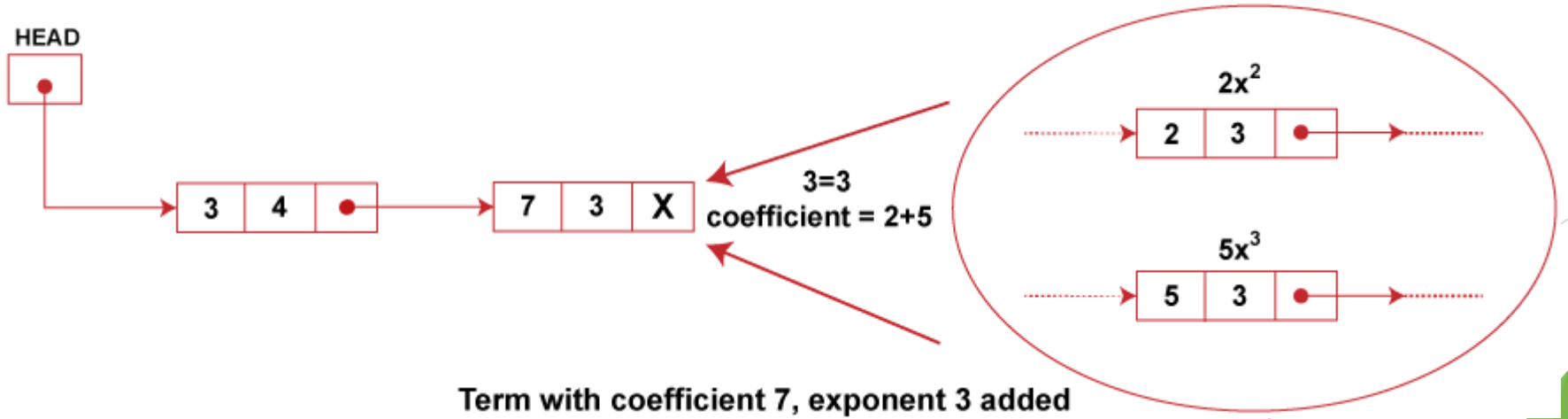
- Let us consider an example to show the addition of two polynomials
- $P(x) = 3x^4 + 2x^3 - 4x^2 + 7$
- $Q(x) = 5x^3 + 4x^2 - 5$



Polynomial Addition using Linked List Sample…

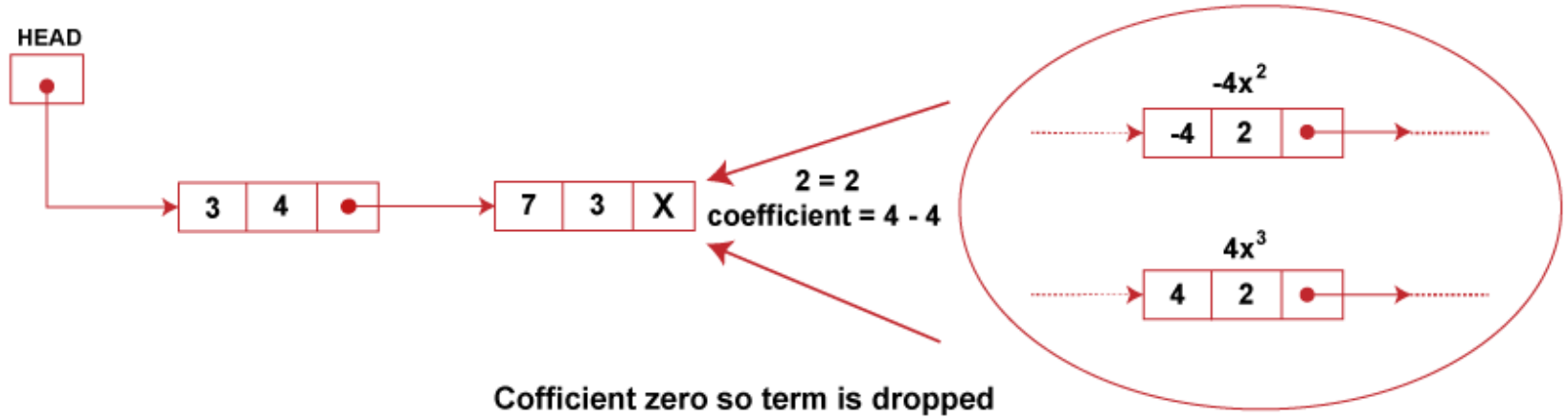


Term with coefficient 3, exponent 4 added

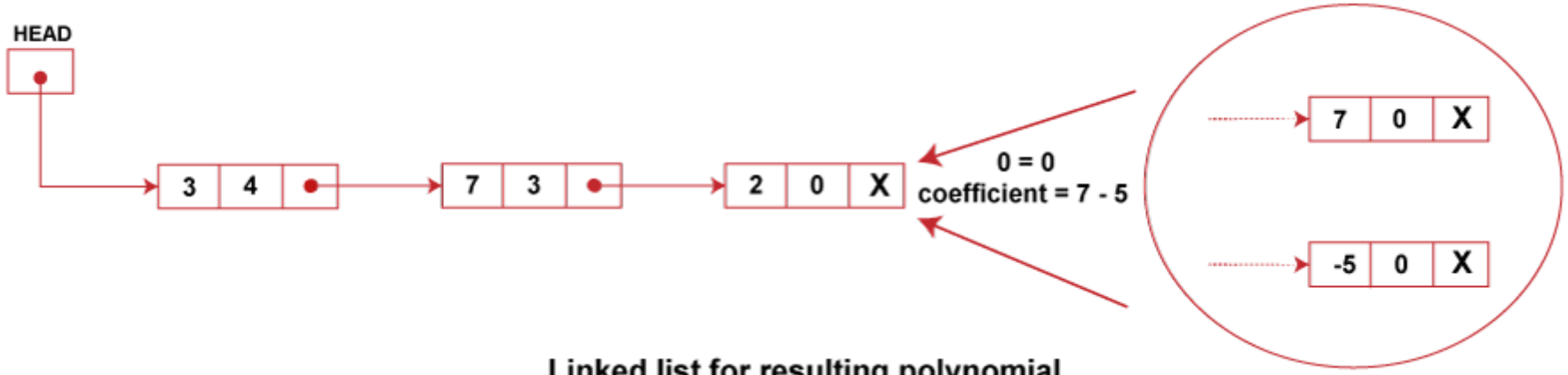


Term with coefficient 7, exponent 3 added

Polynomial Addition using Linked List Sample…



Cofficient zero so term is dropped



Linked list for resulting polynomial

Any Questions ?