Singly (Cont.), Doubly Linked Lists and Circular Linked Lists

Subhabrata Samajder

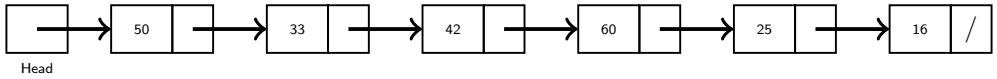
IIIT, Delhi

Winter Semester,

29th March, 2023

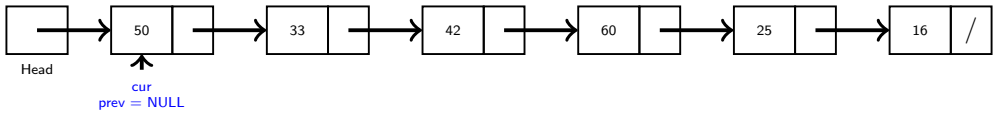
Reversing a Linked List

Reversing a Singly Linked List



Reversing a Singly Linked List

- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.

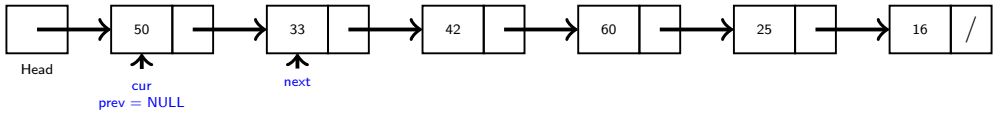


Node *prev = NULL, *cur = NULL, *next = NULL;

cur = pFront;

Reversing a Singly Linked List

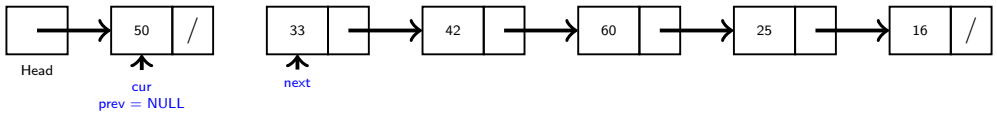
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
  next = cur->pNext;
```

Reversing a Singly Linked List

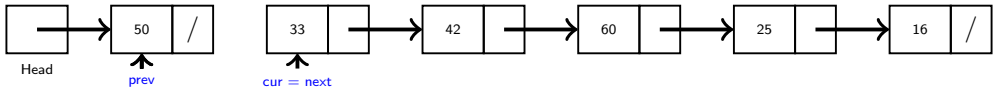
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.




```
Node *prev = NULL, *cur = NULL, *next = NULL;

cur = pFront;

while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
```

Reversing a Singly Linked List

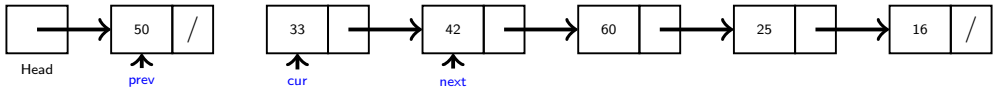
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

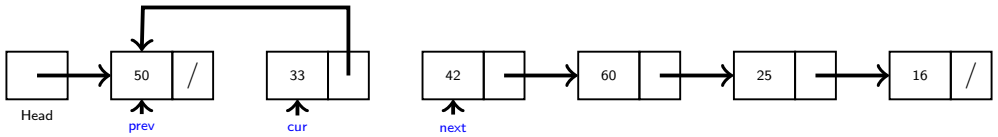
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

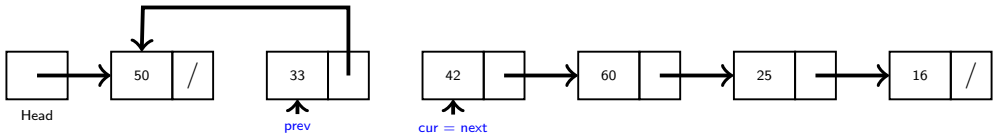
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

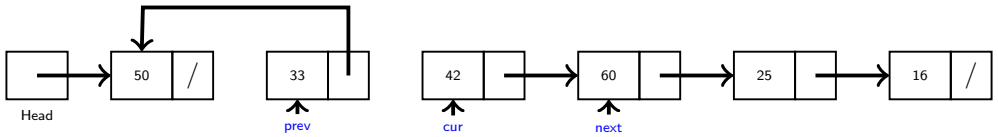
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

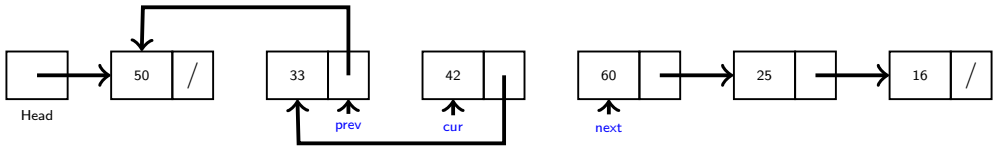
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

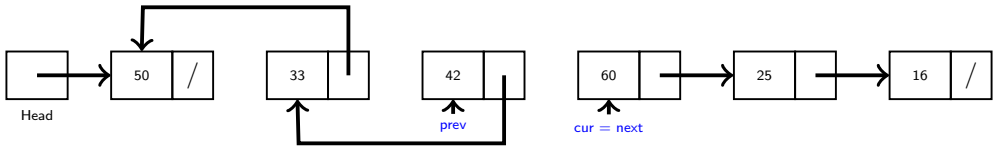
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

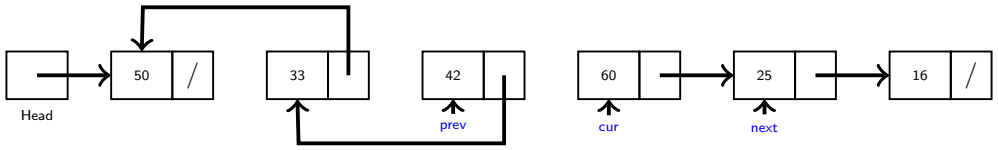
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

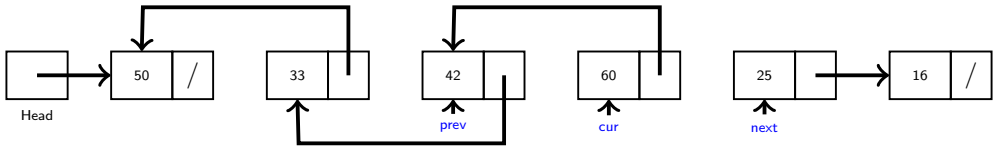
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

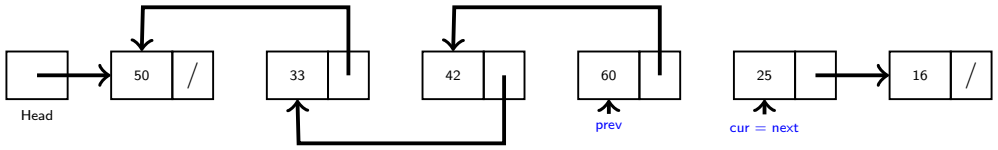
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

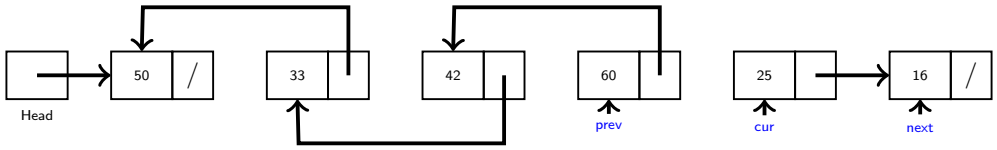
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

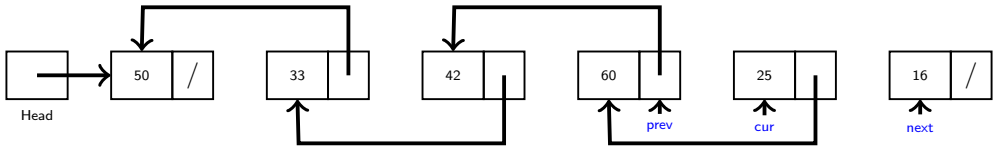
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

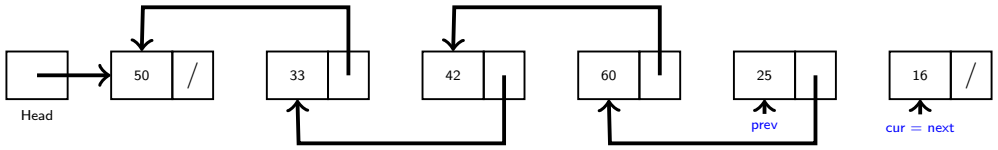
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

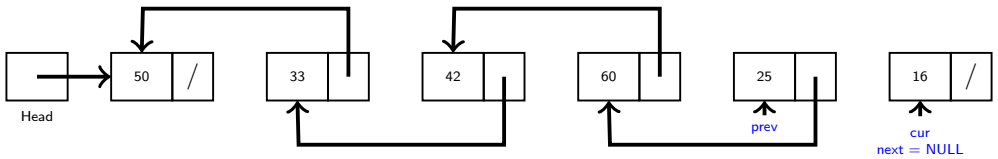
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

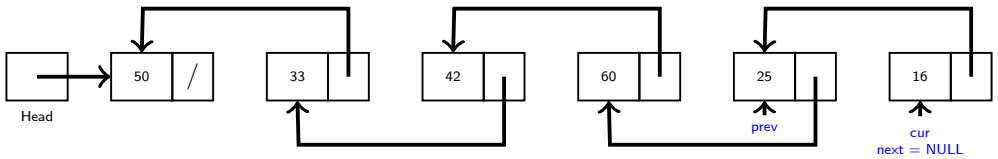
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

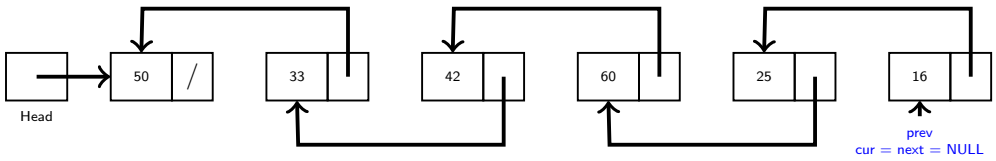
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

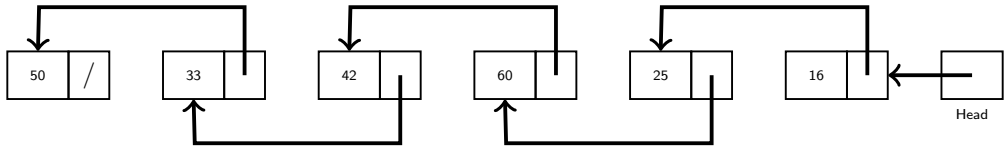
- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
  - Set next = cur->pNext.
  - Set cur->pNext = prev.
  - Set prev = cur.
  - Set cur = next.




```
Node *prev = NULL, *cur = NULL, *next = NULL;

cur = pFront;

while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
```

Reversing a Singly Linked List

- Use 3 pointers: prev, cur, next
- Start with pointing cur to first node, and prev = NULL.
- Traverse the list in a while loop till cur = NULL.
- Set link for Front = prev.




```
Node *prev = NULL, *cur = NULL, *next = NULL;
cur = pFront;
while (cur != NULL) {
   next = cur->pNext;
   cur->pNext = prev;
   prev = cur;
   cur = next; }
pFront = prev;
```

Exercises

1. Create a list with each node contain names of student and their CGPA.
2. Given the above list, find the name of the student having the highest CGPA.
3. Given a list, create two lists with alternate elements of first list.
4. Append a list at end of another list.
5. Check if two lists are identical.

Storing Polynomials in a Linked Lists

Advantages of Linked lists

- Dynamic in nature. Memory allocated at run time.
- Insertion and Deletions are constant time operations (without the searching).
- No need to shift nodes as was necessary with arrays.
- Other data structures like queues, stacks are easily implemented using linked lists

Polynomials

Problem: Add the polynomials

$$5 + 2x + 3x^2,$$
$$7x + 8,$$
$$13 + 9x + 3x^2.$$

Polynomials

Problem: Add the polynomials

$$5 + 2x + 3x^2,$$
$$7x + 8,$$
$$13 + 9x + 3x^2.$$

Note: We need to store only the coefficients and the exponents.

Storing Polynomials Using Arrays

- **Polynomial:** $5 + 2x + 3x^2$
  Array: $[5 \quad 2 \quad 3]$

- **Polynomial:** $7x + 8$
  Array: $[8 \quad 7 \quad 0]$

Storing Polynomials Using Arrays

- **Polynomial:** $5 + 2x + 3x^2$
  Array: $[5 \quad 2 \quad 3]$

- **Polynomial:** $7x + 8$
  Array: $[8 \quad 7 \quad 0]$

- That is, store only the coefficients in proper place.

Issues in Storing Polynomials Using Arrays

- **Polynomial:** $5 + 2x + 3x^2 + 6x^5$
  Array: $[5 \quad 2 \quad 3 \quad 0 \quad 0 \quad 6]$

- **Polynomial:** $5 + 2x + 3x^2 + 7x^{31}$
  Array: $[5 \quad 2 \quad 3 \quad 0 \quad 0 \quad 0 \quad \cdots \quad 0 \quad 7]$

- Need to store so many zeroes in a very large sized array

Storing Polynomials Using Linked Lists

- Let us now see how two polynomials can be added.

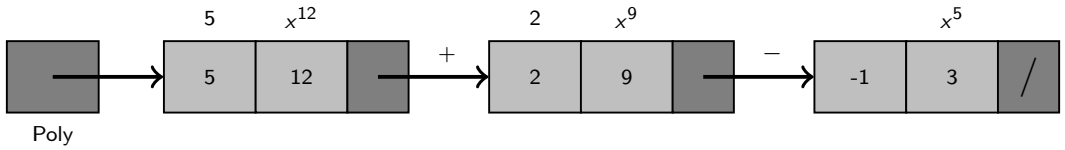
- Let P_1 and P_2 be two polynomials
  - stored as linked lists
  - Each node contains exponent and coefficients values
  - in sorted (decreasing) order of exponents

- **Addition Operation:** Add terms of like-exponents.

Representing a Polynomial Using a Linked List

Store the coefficient and exponent of each term in nodes

```
int item1[] = {5, 12};
int item2[] = {2, 9};
int item3[] = {-1, 3};
```

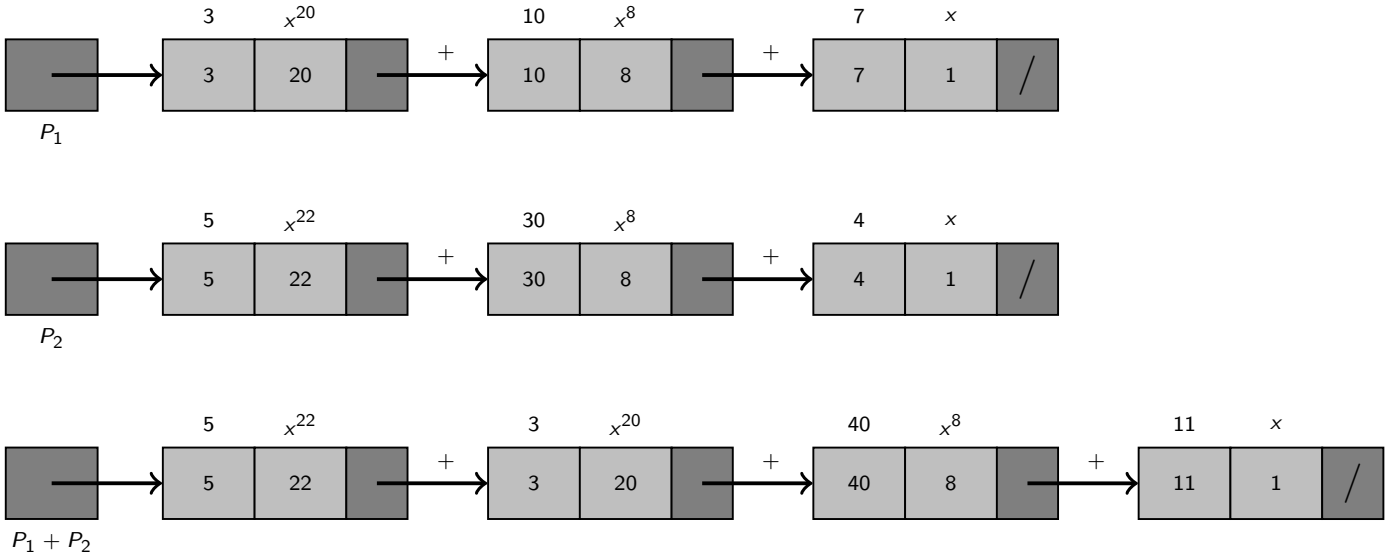


Operations on Polynomials

- P_1 and P_2 are stored as linked lists and are arranged in decreasing order of exponents.
- Scan these and add like terms.
- Store the resulting term only if it has non-zero coefficient.
- The number of terms in the result $(P_1 + P_2)$ need not be known in advance.
- Uses as much space as there are terms in $P_1 + P_2$.

Addition of Two Polynomials

One pass down each list: $\mathcal{O}(n+m)$.



Multiplication of Two Polynomials

- Can be done as repeated addition.
- So, multiply P_1 with each term of P_2.
- Add the resulting polynomials.

Doubly Linked Lists

Doubly Linked Lists

- Permits traversal of list in both directions.
- Useful where navigation in both directions needed.
- Used by browsers to navigate forwards and backwards.
- Various applications use this for redo and undo functionalities.

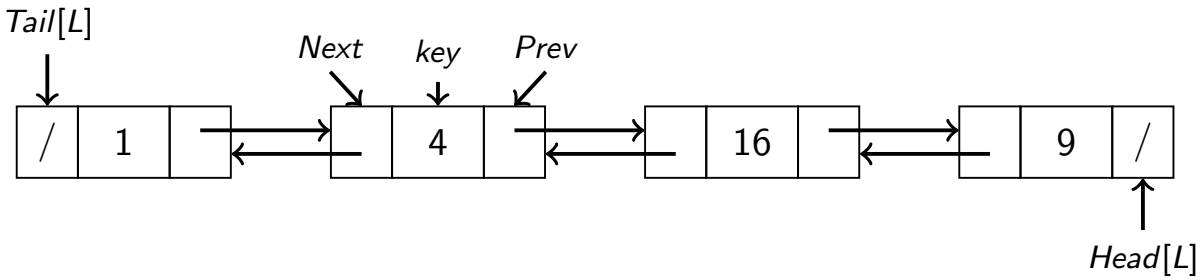
Doubly Linked List (Cont.)

- Each element of a doubly linked list L is an object x with a *key* (or *data*) field and two other pointer fields:
  - $next[x]$ points to it successor in the linked list and
  - $prev[x]$ points to its predecessor.
- **Head of L:** If $prev[x] = \textbf{nil}$.
- **Tail of L:** If $next[x] = \textbf{nil}$.
- **head[L]:** Points to the first element of the list L.
- **tail[L]:** Points to the last element of the list L.
- **Empty List:** If $head[L] = \textbf{nil}$.

C Implementation of a Doubly Linked List Node

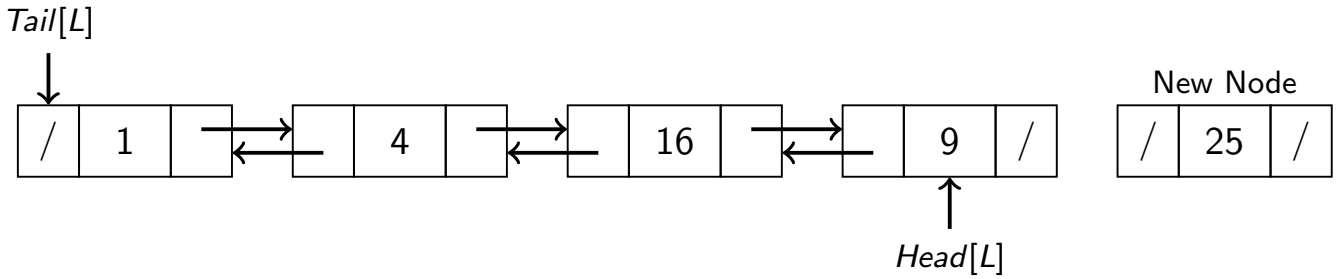
```
typedef struct DLNode {
   int nKey;
   struct DLNode *pPrev, *pNext;
} DLNode;
```

Insertion of an Element at the Head



Insertion of an Element at the Head

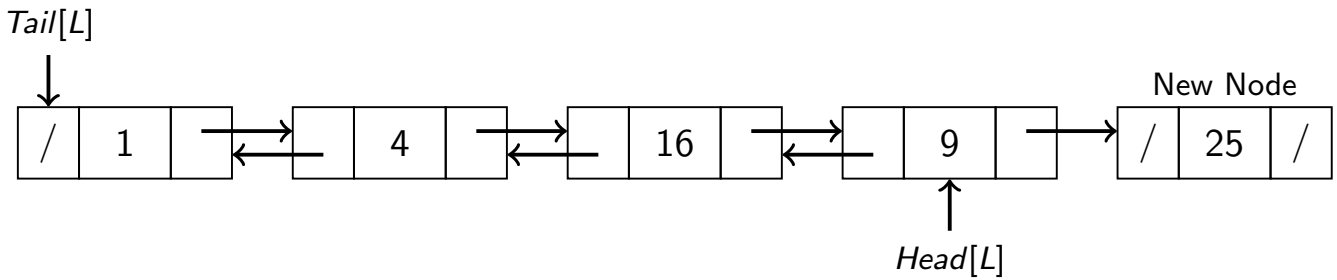
- Create a new node x.




```
DLNode *pTemp;

pTemp = (DLNode *)malloc(sizeof(DLNode));
pTemp->nKey = 25;
pTemp->pPrev = NULL;

pTemp->pNext = NULL;
```

Insertion of an Element at the Head

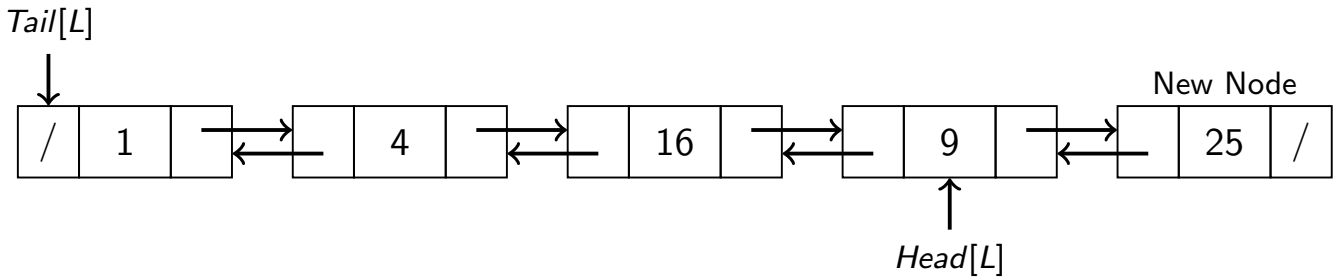
- Create a new node x.
- $head[L].prev = x$.




```
DLNode *pTemp;

pTemp = (DLNode *)malloc(sizeof(DLNode));
pTemp->nKey = 25;
pTemp->pPrev = NULL;
pTemp->pNext = NULL;

pHead->pPrev = pTemp;
```

Insertion of an Element at the Head

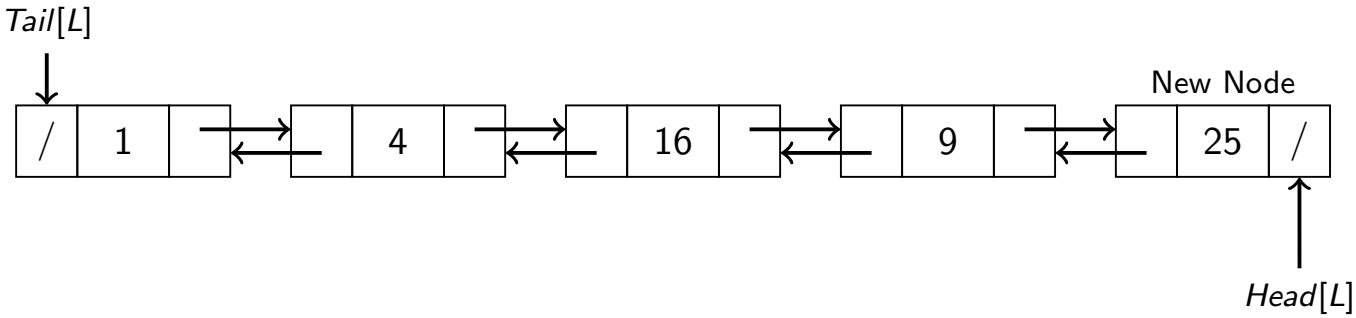
- Create a new node x.
- $head[L].prev = x$.
- $x.next = head[L]$.




```
DLNode *pTemp;

pTemp = (DLNode *)malloc(sizeof(DLNode));
pTemp->nKey = 25;
pTemp->pPrev = NULL;
pTemp->pNext = NULL;

pHead->pPrev = pTemp;
pTemp->pNext = pHead;
```

Insertion of an Element at the Head

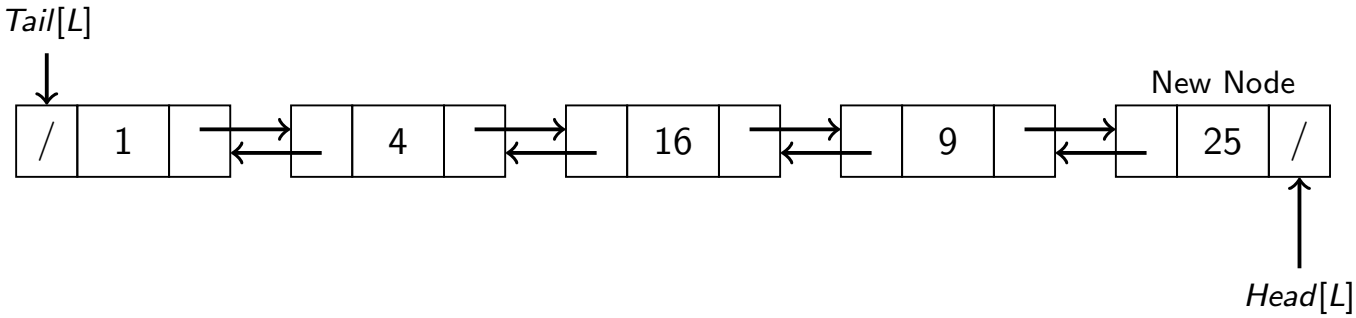
- Create a new node x.
- $head[L].prev = x$.
- $x.next = head[L]$.
- $head[L] = x$.




```
DLNode *pTemp;

pTemp = (DLNode *)malloc(sizeof(DLNode));
pTemp->nKey = 25;
pTemp->pPrev = NULL;
pTemp->pNext = NULL;

pHead->pPrev = pTemp;
pTemp->pNext = pHead;
pHead = pTemp;
```

Insertion of an Element at the Head

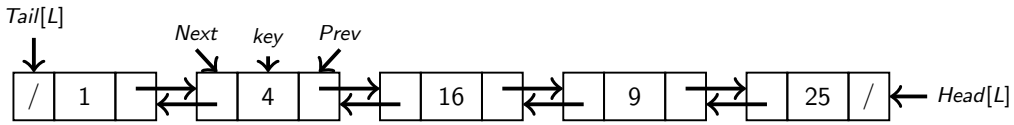
- Create a new node x.
- $head[L].prev = x$.
- $x.next = head[L]$.
- $head[L] = x$.




```
DLNode *pTemp;

pTemp = (DLNode *)malloc(sizeof(DLNode));
pTemp->nKey = 25;
pTemp->pPrev = NULL;
pTemp->pNext = NULL;

pHead->pPrev = pTemp;
pTemp->pNext = pHead;
pHead = pTemp;
```

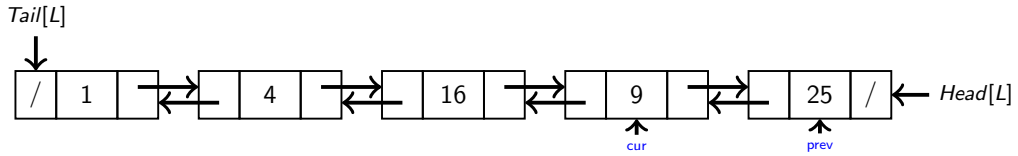
Deleting A Node Containing Data d

- Assume that the list is of length at least 2.
- Let $d = 4$.



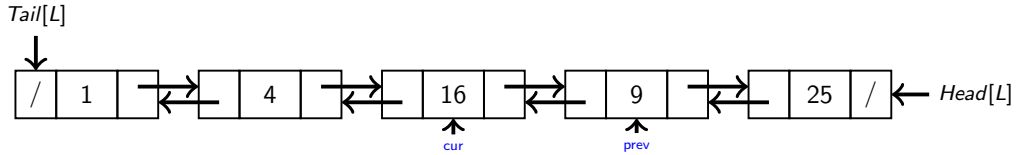
Deleting A Node Containing Data d

- Assume that the list is of length at least 2.
- Let $d = 4$.
- Set pointer prev to the first and cur to the second node.



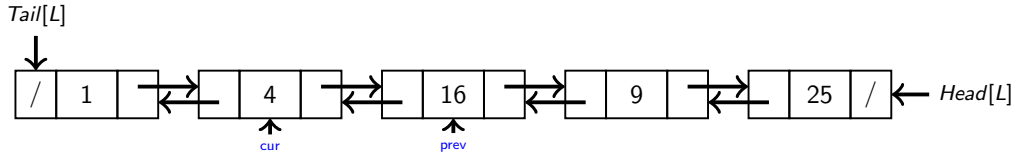
Deleting A Node Containing Data d

- Assume that the list is of length at least 2.
- Let $d = 4$.
- Set pointer prev to the first and cur to the second node.
- Traverse until prev->nKey $= 4$.



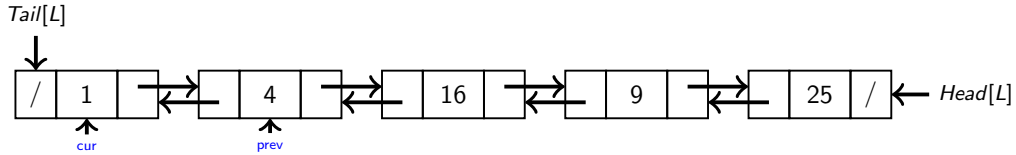
Deleting A Node Containing Data d

- Assume that the list is of length at least 2.
- Let $d = 4$.
- Set pointer prev to the first and cur to the second node.
- Traverse until prev->nKey $= 4$.



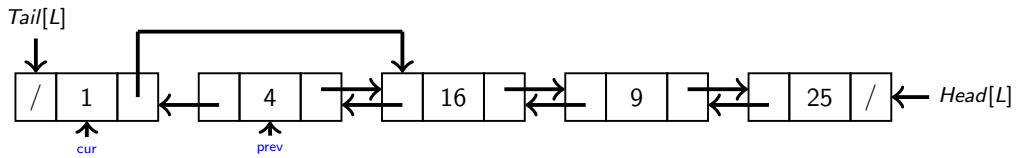
Deleting A Node Containing Data d

- Assume that the list is of length at least 2.
- Let $d = 4$.
- Set pointer prev to the first and cur to the second node.
- Traverse until prev->nKey $= 4$.



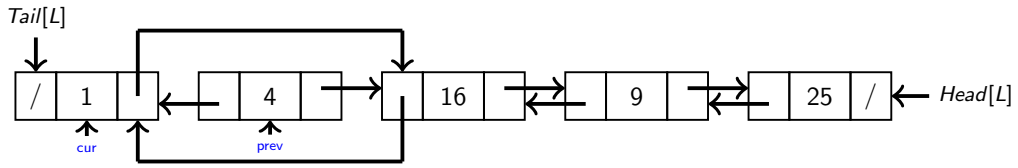
Deleting A Node Containing Data d

- Assume that the list is of length at least 2.
- Let $d = 4$.
- Set pointer prev to the first and cur to the second node.
- Traverse until prev->nKey $= 4$.
- Set cur->pPrev $=$ prev->pPrev.



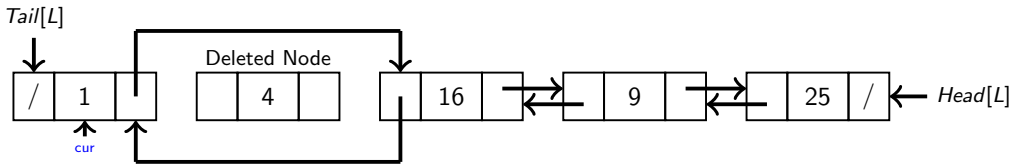
Deleting A Node Containing Data d

- Assume that the list is of length at least 2.
- Let $d = 4$.
- Set pointer prev to the first and cur to the second node.
- Traverse until prev->nKey $= 4$.
- Set cur->pPrev $=$ prev->pPrev.
- Set prev->pPrev->pNext $=$ cur.



Deleting A Node Containing Data d

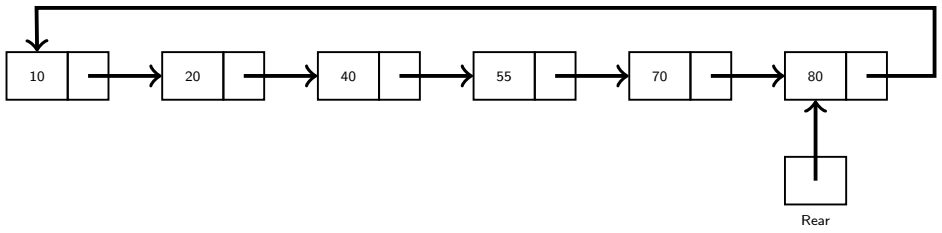
- Assume that the list is of length at least 2.
- Let $d = 4$.
- Set pointer prev to the first and cur to the second node.
- Traverse until prev->nKey $= 4$.
- Set cur->pPrev $=$ prev->pPrev.
- Set prev->pPrev->pNext $=$ cur.
- Set free(prev).



Circular Linked Lists

Circular Linked Lists

- A Circular Linked List is a special type of Linked List
- It supports traversing from the end of the list to the beginning by making the last node point back to the head of the list.
- A Rear pointer is often used instead of a Head pointer.



Motivation

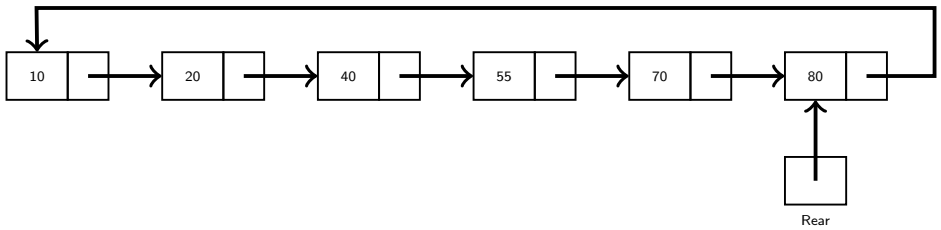
- Usually sorted.
- Useful for playing video and sound files in “looping” mode.
- They are also a stepping stone for implementing graphs.

Circular Linked List Operations

- insertNode(Node *Rear, int item)
  //adds a new node to ordered circular linked list
- deleteNode(Node *Rear, int item)
  //removes a node from circular linked list
- print(Node *Rear)
  //print the Circular Linked List once

Traversing a Circular Linked List

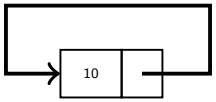
```
void print(Node *Rear){
  Node *Cur;

  if(Rear != NULL){
    Cur = Rear->pNext;
    do{
      printf("%d, ", Cur->nData);
      Cur = Cur->pNext;
    } while(Cur != Rear->pNext);
  }
}
```



Insert Node

- **Empty List:**

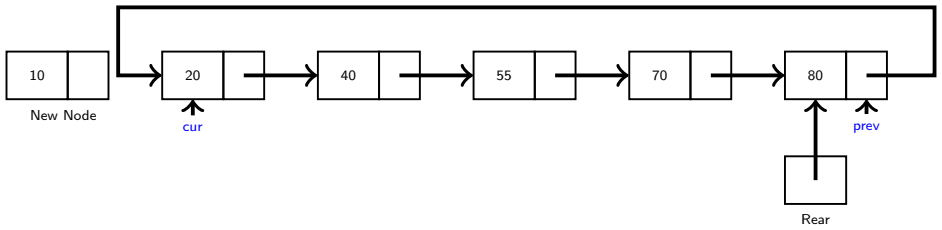
```
Note *New = NULL;

New = (Node *)malloc(sizeof(Node));
New->nData = 10;

Rear = New;

Rear->pNext = Rear;
```



Insert Node

- **Inserting a Node at the Head:**

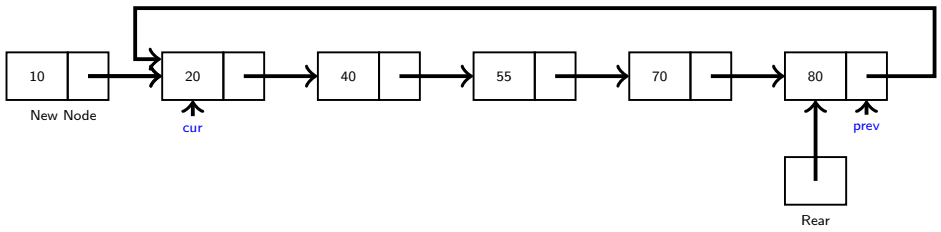
```
Node *New = NULL;

New = (Node *)malloc(sizeof(Node));
New->nData = 10;

cur = Rear->pNext;
prev = Rear;
```



Insert Node

- **Inserting a Node at the Head:**

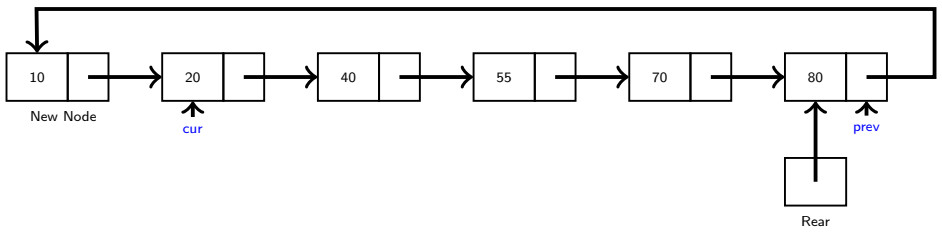
```
Node *New = NULL;

New = (Node *)malloc(sizeof(Node));
New->nData = 10;

cur = Rear->pNext;
prev = Rear;

New->pNext = Cur;
```



Insert Node

- **Inserting a Node at the Head:**

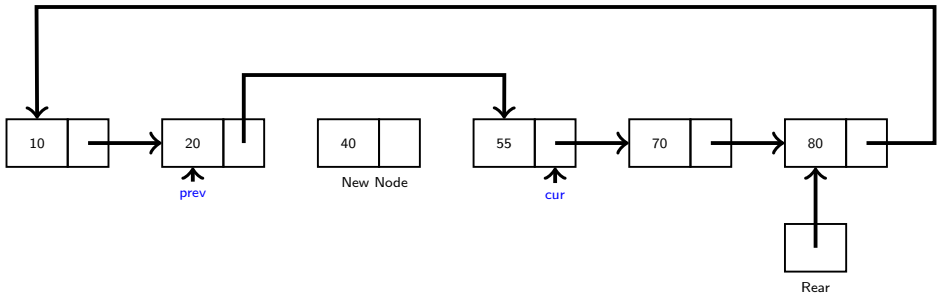
```
Node *New = NULL;

New = (Node *)malloc(sizeof(Node));
New->nData = 10;

cur = Rear->pNext;
prev = Rear;

New->pNext = Cur;

Prev->pNext = New;
```



Insert Node

- **Inserting a Node in the Middle:**

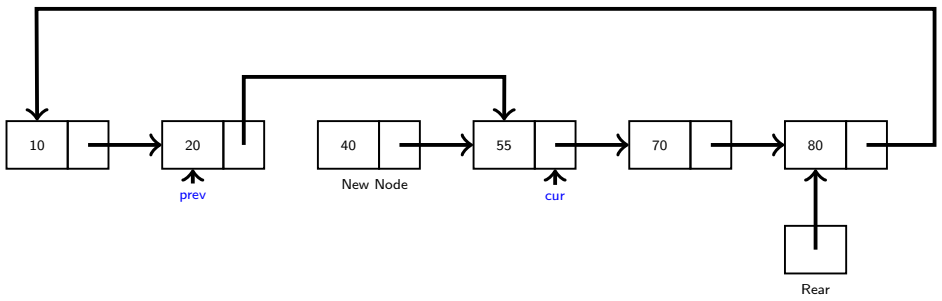
```
Node *New = NULL;

New = (Node *)malloc(sizeof(Node));
New->nData = 40;

cur = Rear.->pNext;
prev = Rear;

//Find the place to insert the node
```



Insert Node

- **Inserting a Node in the Middle:**

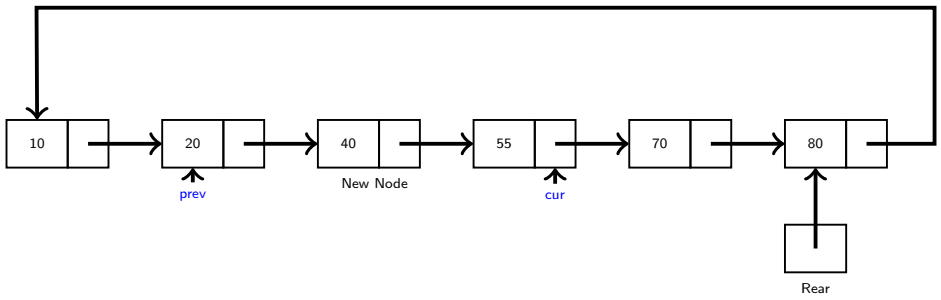
```
Node *New = NULL;

New = (Node *)malloc(sizeof(Node));
New->nData = 40;

cur = Rear.->pNext;
prev = Rear;

//Find the place to insert the node

New->pNext = Cur;
```



Insert Node

- **Inserting a Node in the Middle:**

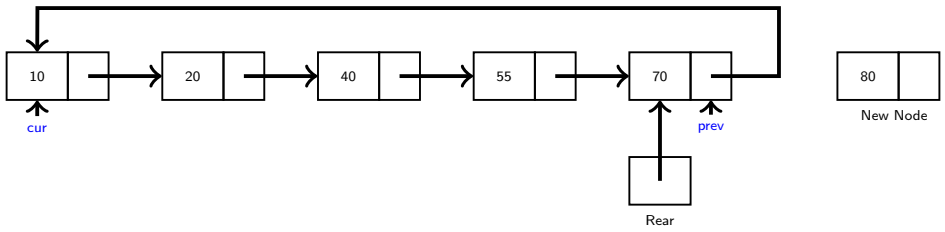
```
Node *New = NULL;

New = (Node *)malloc(sizeof(Node));
New->nData = 40;

cur = Rear.->pNext;
prev = Rear;

//Find the place to insert the node

New->pNext = Cur;

Prev->pNext = New;
```



Insert Node

- **Inserting a Node at the End:**

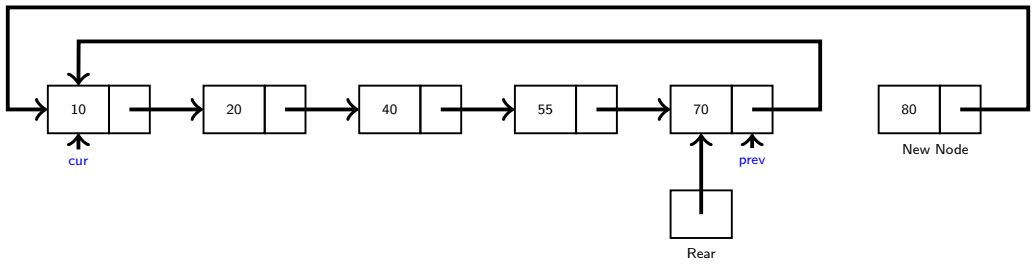
```
Node *New = NULL;

New = (Node *)malloc(sizeof(Node));
New->nData = 80;

cur = Rear->pNext;
prev = Rear;
```



Insert Node

- **Inserting a Node at the End:**

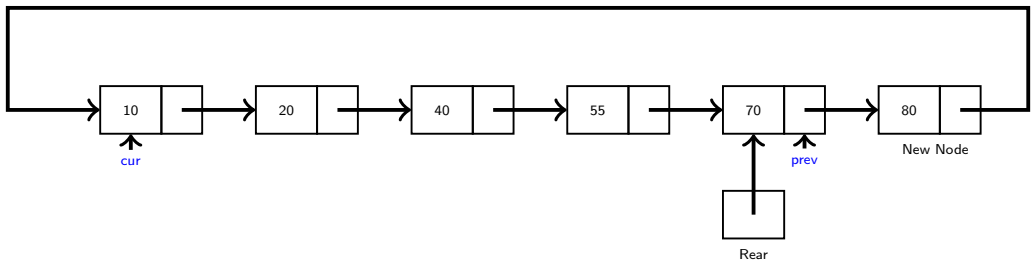
```
Node *New = NULL;

New = (Node *)malloc(sizeof(Node));
New->nData = 80;

cur = Rear->pNext;
prev = Rear;

New->pNext = Cur;
```



Insert Node

- **Inserting a Node at the End:**

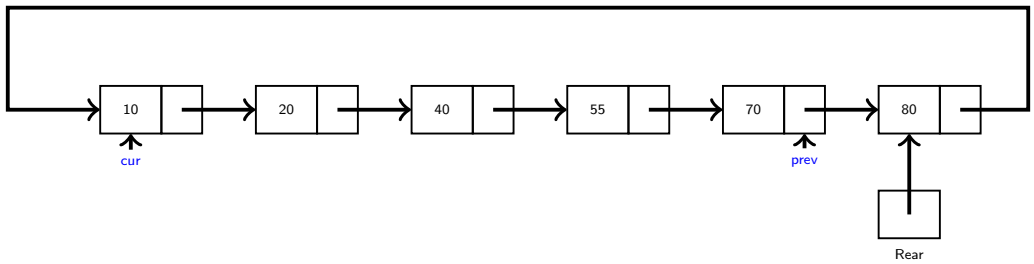
```
Node *New = NULL;

New = (Node *)malloc(sizeof(Node));
New->nData = 80;

cur = Rear->pNext;
prev = Rear;

New->pNext = Cur;
Prev->pNext = New;
```



Insert Node

- **Inserting a Node at the End:**

```
Node *New = NULL;

New = (Node *)malloc(sizeof(Node));
New->nData = 80;

cur = Rear->pNext;
prev = Rear;

New->pNext = Cur;
Prev->pNext = New;

Rear = New;
```

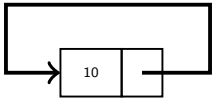


Delete Node

- **List of Size 1:**

  free(Rear);

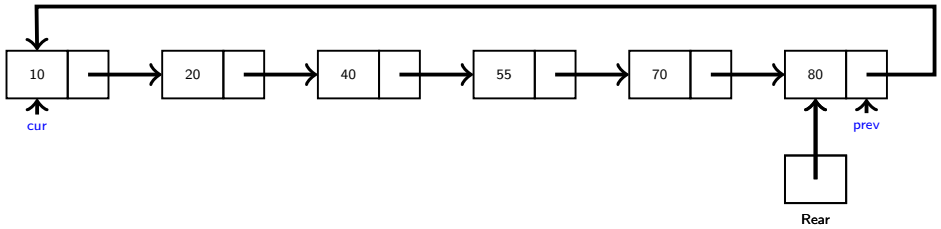
  Rear = NULL;



Delete Node

- **Deleting the Head Node:**

  cur = Rear->pNext;
  prev = Rear;

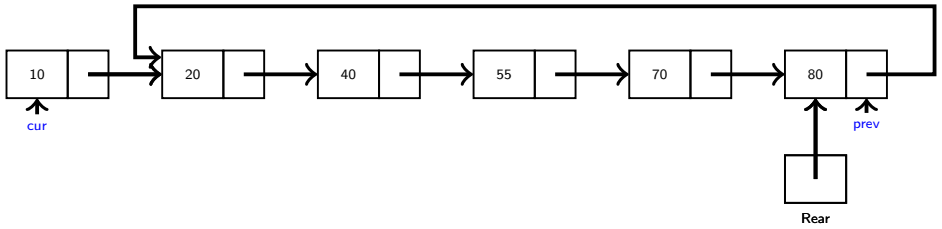


Delete Node

- **Deleting the Head Node:**

  cur = Rear->pNext;
  prev = Rear;

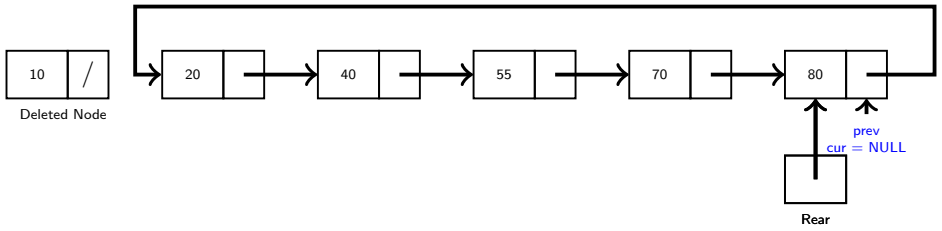
  prev->pNext = cur->pNext;



Delete Node

- **Deleting the Head Node:**

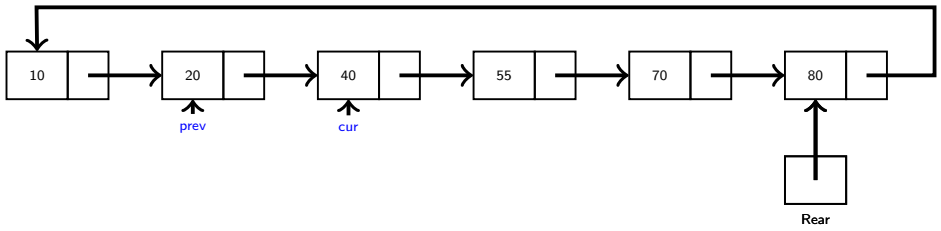
```
cur = Rear->pNext;
prev = Rear;

prev->pNext = cur->pNext;
free(cur);

cur = NULL;
```



Delete Node

- **Deleting a Middle Node:**

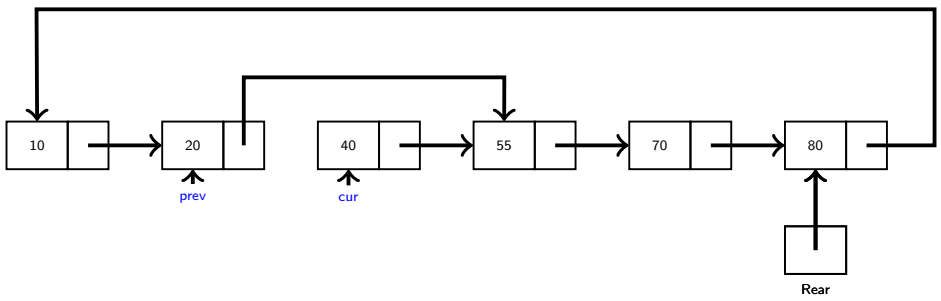
```
cur = Rear->pNext;
prev = Rear;

//Find the node to delete
```



Delete Node

- **Deleting a Middle Node:**

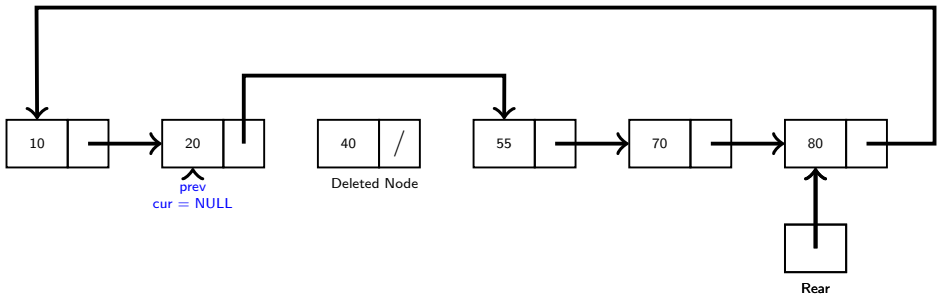
```
cur = Rear->pNext;
prev = Rear;

//Find the node to delete

prev->pNext = Cur->pNext;
```



Delete Node

- **Deleting a Middle Node:**

```
cur = Rear->pNext;
prev = Rear;

//Find the node to delete

prev->pNext = Cur->pNext;
free(cur);
cur = NULL;
```

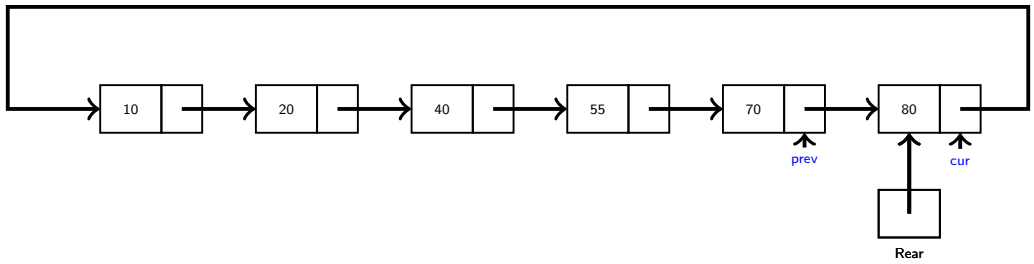


Delete Node

- **Deleting the Node at the End:**

cur = Rear->pNext;
prev = Rear;

//Traverse till the end of the list, i.e., till cur == Rear



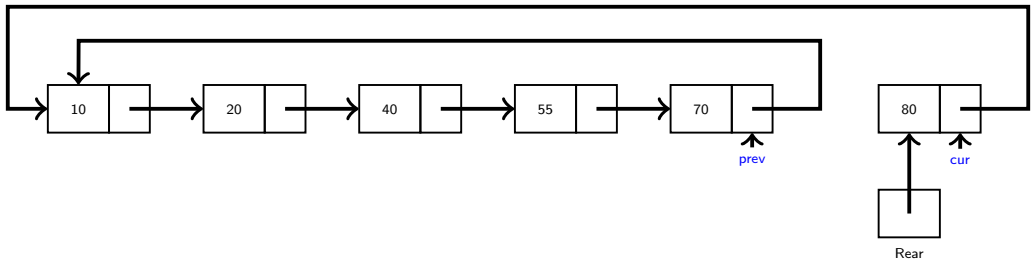
Delete Node

- **Deleting the Node at the End:**

cur = Rear->pNext;
prev = Rear;

//Traverse till the end of the list, i.e., till cur == Rear

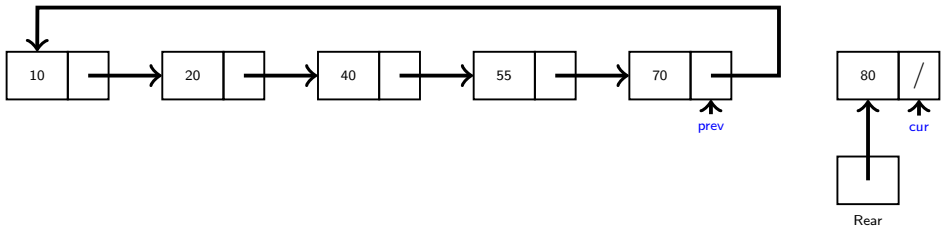
prev->pNext = cur->pNext;



Delete Node

- **Deleting the Node at the End:**

```
cur = Rear->pNext;
prev = Rear;

//Traverse till the end of the list, i.e., till cur == Rear

prev->pNext = cur->pNext;
free(cur);
```

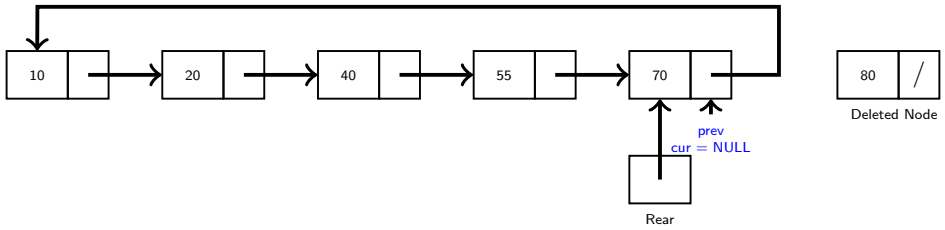


Delete Node

- **Deleting the Node at the End:**

```
cur = Rear->pNext;
prev = Rear;

//Traverse till the end of the list, i.e., till cur == Rear

prev->pNext = cur->pNext;
free(cur);
Rear = prev;
```

cur = NULL



Exercises

1. Assuming that there can exists at most one loop in a singly linked list, find an algorithm to determine whether a single linked list has a loop or not.

Books Consulted

1. Chapter 10.2 of *Introduction to Algorithms* by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein.

Thank You for your kind attention!

Questions!!