Data Structures Queues

Queues

- A **Queue** is a special kind of list, where items are inserted at one end (**the rear**) And deleted at the other end (**the front**).

- Accessing the elements of queues follows a First In,First Out (FIFO) order.
- Example
  - Like customers standing in a check-out line in a store, the first customer in is the first customer served.

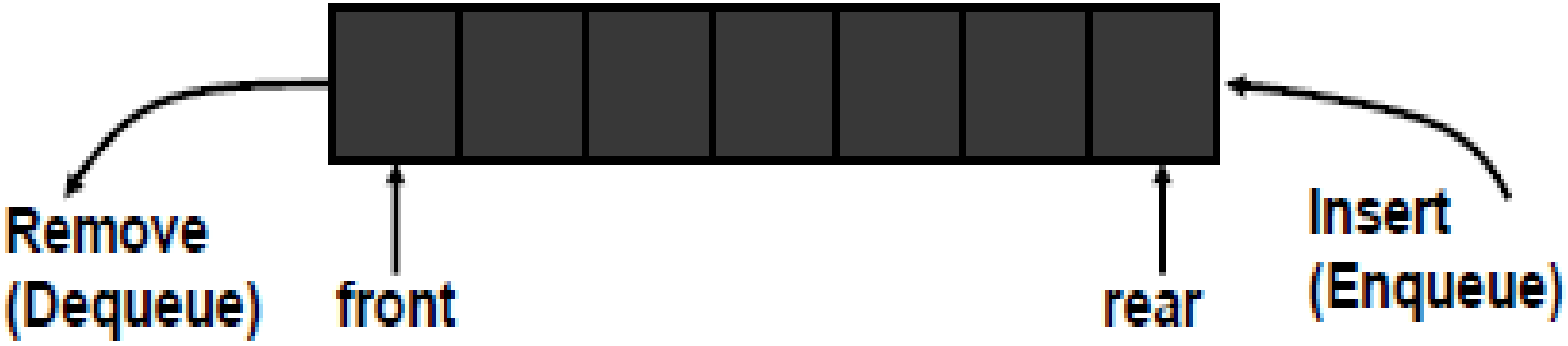
Common Operations on Queues

- **MAKENULL:**
- **FRONT*(Q*):** Returns the first element on Queue Q.
- **ENQUEUE(*x*,*Q*):** Inserts element x at the end of Queue Q.
- **DEQUEUE(*Q*):** Deletes the first element of *Q.*
- **ISEMPTY(*Q*):** Returns true if and only if Q is an empty queue.
- **ISFULL(Q):** Returns true if and only if Q is full.

Enqueue and Dequeue

- Primary queue operations: Enqueue and Dequeue

- **Enqueue** – insert an element at the rear of the queue.
- **Dequeue** – remove an element from the front of the queue.

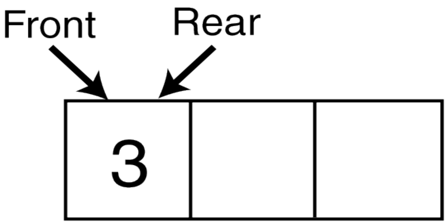


Queues Implementations

- Static
  - Queue is implemented by an array, and size of queue remains fix
- Dynamic
  - A **queue** can be **implemented** as a **linked list**, and *expand* or *shrink* with each *enqueue* or *dequeue* operation.

Static Implementation of Queues

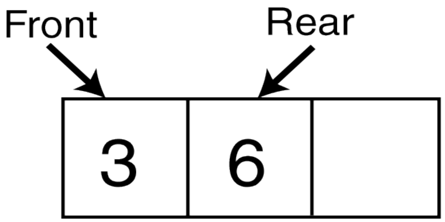
Enqueue(3);



Front Rear

3

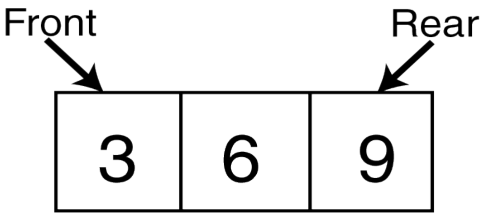
Enqueue(6);



Front Rear

3 6

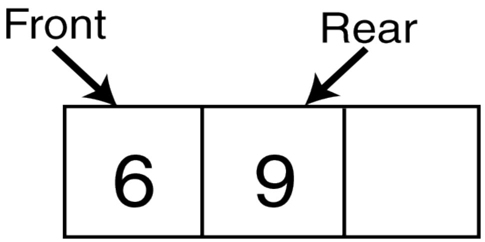
Enqueue(9);



Front Rear

3 6 9

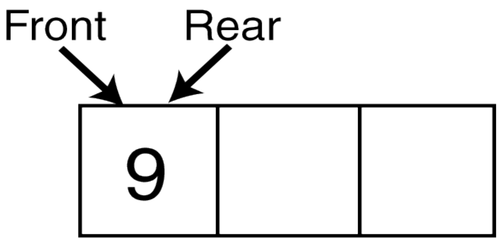
Dequeue();



Front Rear

6 9

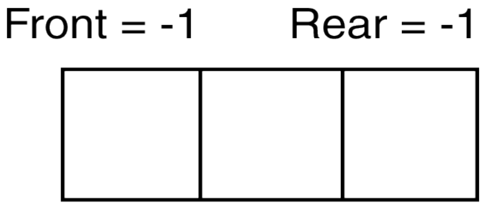
Dequeue();



Front Rear

9

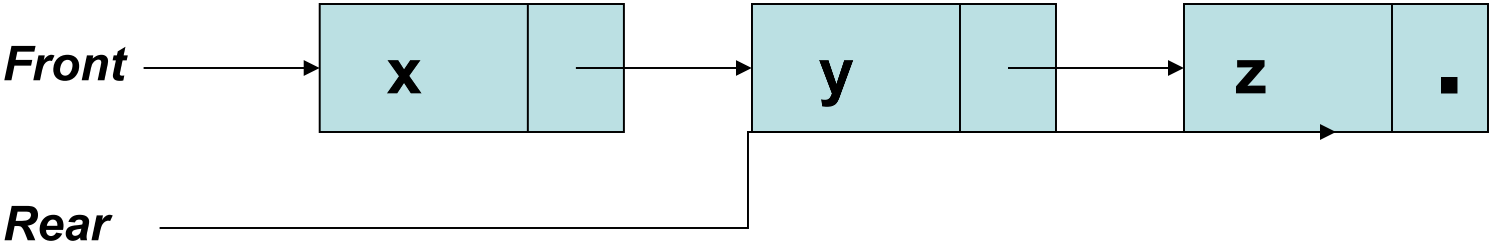
Dequeue();



Front = -1 Rear = -1

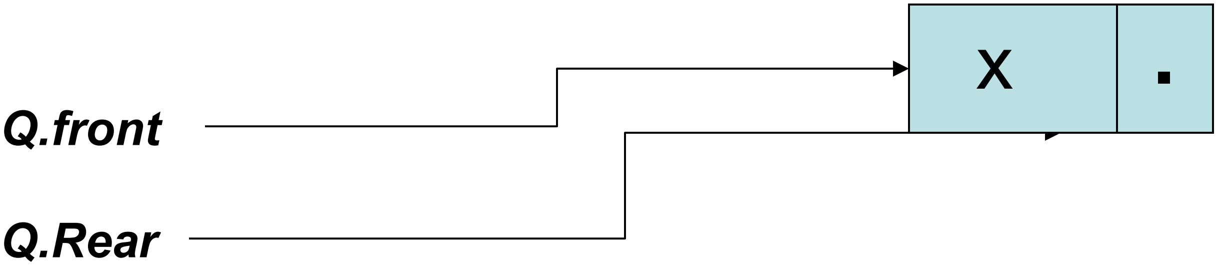
Dynamic Implementation of Queues

- Dynamic implementation is done using pointers.
  - FRONT: A pointer to the first element of the queue.
  - REAR: A pointer to the last element of the queue.

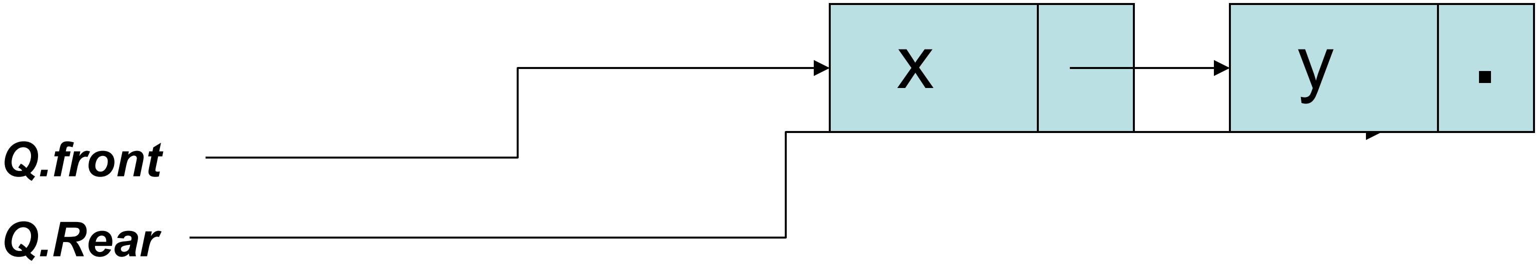


Dynamic Implemenatation

- Enqueue (X)

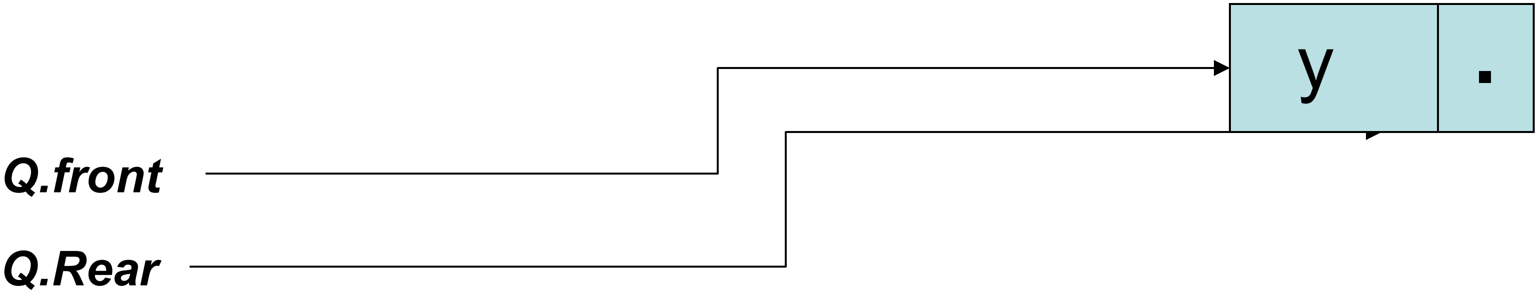


- Enqueue (Y)

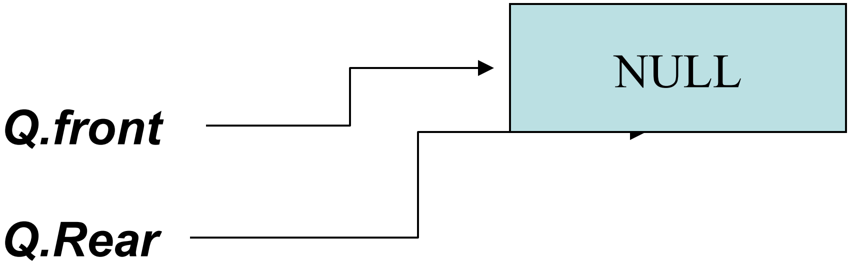


Dynamic Implementation

- Dequeue



- MakeNULL



Dynamic implementation of Queue

```
class DynQueue{
    private:
    struct queueNode
   {
    int num;
     queueNode *next;
   };
   queueNode *front;
   queueNode *rear;
  public:
    DynQueue();
    ~DynQueue();
   void enqueue();
   void dequeue();
   bool isEmpty();
   void displayQueue();
   void makeNull();
}
```

Constructor

```
DynQueue::DynQueue()
{
   front = NULL;
  rear = NULL;
}
```

Enqueue( ) Function

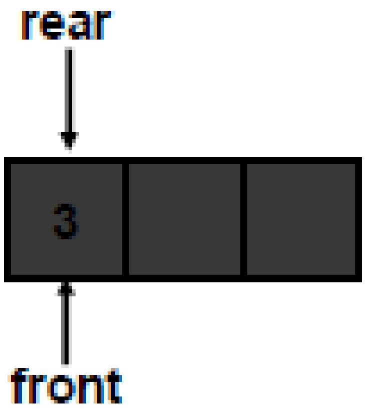
```
void DynQueue::enqueue()
{
    queueNode *ptr;
    ptr = new queueNode;
    cout<<"Enter Data";
    cin>>ptr->num;
    ptr->next= NULL;
  if (front == NULL)
  {
    front = ptr;
    rear = front;
  }
else{
     rear->next=ptr;
     rear = ptr;
  }
```

Dequeue( ) Function

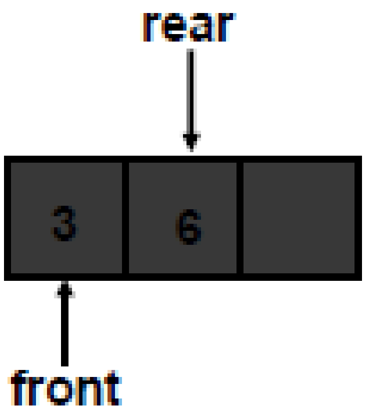
```
void DynQueue::dequeue()
{
    queueNode *temp;
  temp = front;
    if(isEmpty())
    cout<<"Queue is Empty";
  else
  {
    cout<<"data deleted="<<temp->num;
     front = front->next;
     delete temp;
  }
}
```

Static Implementation of Queue

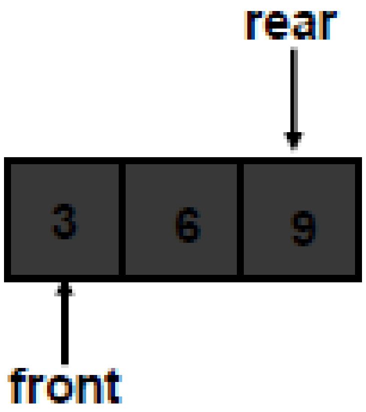
- Static implementation is done using arrays
- In this implementation, we should know the exact number of elements to be stored in the queue.
- When enqueuing, the front index is always fixed and the rear index moves forward in the array.



Enqueue(3)



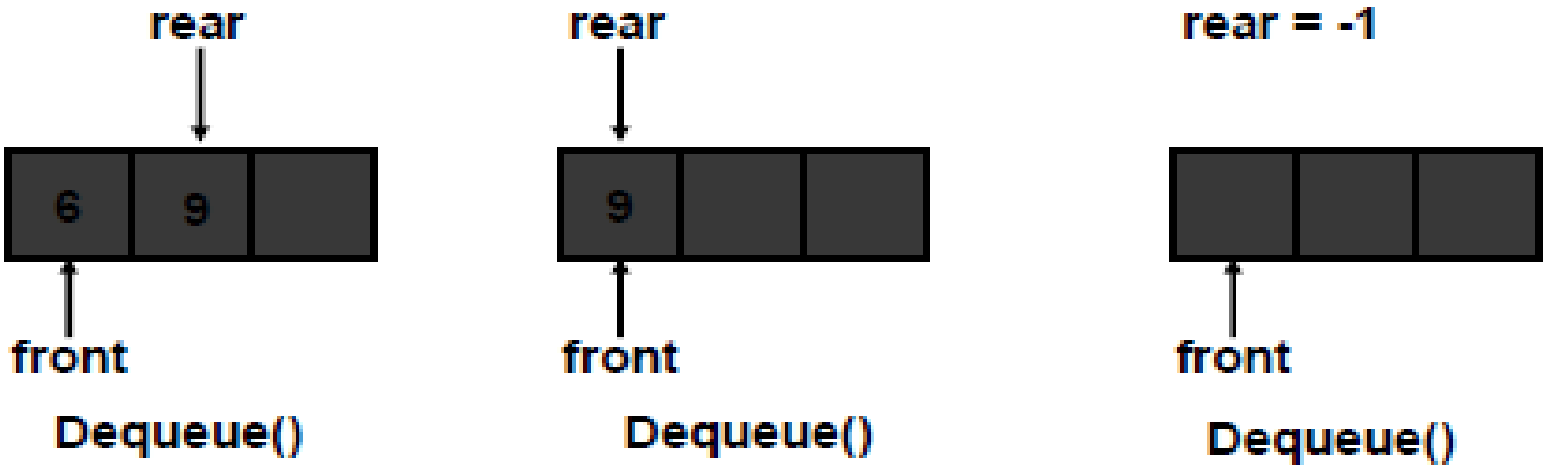
Enqueue(6)



Enqueue(9)

Static Implementation of Queue

- When dequeuing, the front index is fixed, and the element at the front of the queue is removed. Move all the elements after it by one position. (Inefficient!!!)



Static Implementation of Queue

- **A better way**
  - When an item is enqueued, the rear index moves forward.
  - When an item is dequeued, the front index also moves forward by one element
- **Example:**

X = occupied, and O = empty

- (front) XXXXOOOOO (rear)
- OXXXXOOOO (after 1 dequeue, and 1 enqueue)
- OOXXXXXOO (after another dequeue, and 2 enqueues)
- OOOOXXXXX (after 2 more dequeues, and 2 enqueues)

- **The problem here is that the rear index cannot move beyond the last element in the array.**

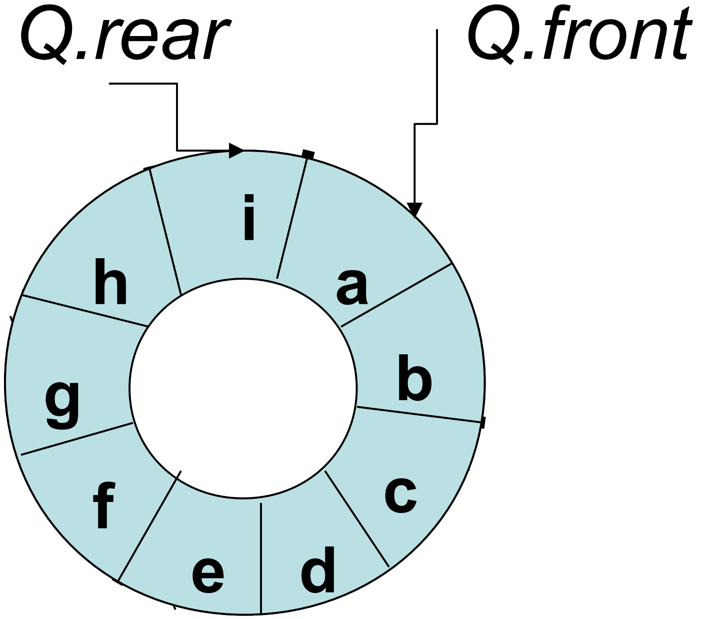
Static Implementation of Queue

- To overcome the above limitation, we can use **circular array implementation of queues.**
- In this implementation, first position follows the last.

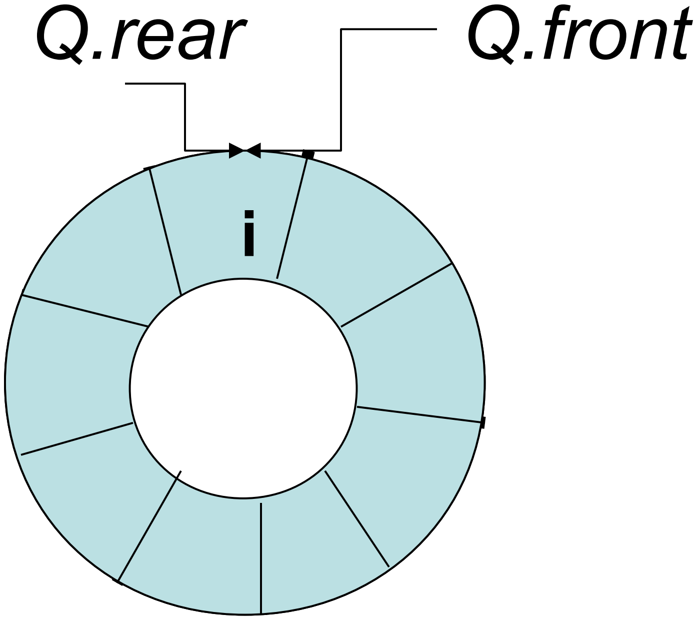
- **When an element moves past the end of a circular array, it wraps around to the beginning, e.g**
  - OOOOO7963 ->4OOOO7963 (after Enqueue(4))
  - After Enqueue(4), the rear index moves from 3 to 4.

- **How to detect an empty or full queue, using a circular array algorithm?**
  - Use a counter of the number of elements in the queue.

Circular Queue



A Completely
Filled Queue



A Queue with
Only 1 Element

Circular Queue Implementation

```
class CirQueue
{
    private:
        int queue[5];
    int rear;
    int front;
    int maxSize;
    int counter;
  public:
        CirQueue();
    void enqueue();
    void dequeue();
    bool isEmpty();
    bool isFull();
    void display();
};
```

Constructor

```
CirQueue::CirQueue()
{
     front = 0;
    rear = -1;
    maxSize = 5;
    counter =0;
}
```

Enqueue( ) Function

```
void CirQueue::enqueue()
{
    if ( isFull())
    cout<<"queue is full";
  else
  {
    rear = (rear + 1) % maxSize;
    cout<<"Enter Data=";
    cin>> queue[rear];
    counter ++;
  }
}
```

Dequeue( ) Function

```
void CirQueue::dequeue()
{
    if ( isEmpty())
    cout<<"Queue is empty";
  else
  {
    cout<< "Element deleted="<<queue[front];
     front = (front +1)% maxSize;
     counter --;
  }
}
```

Display( ) Function

```
void CirQueue::display()
{
    if(isEmpty())
    cout<<"Queue is empty";
  else
  {
        for (int i=0; i<counter; i++)
        cout<< queue[(front+ i)% maxSize]<<endl;;
  }

}
```

isEmpty( ) and isFull( )

```
bool CirQueue::isEmpty()
{
    if (counter == 0)
    return true;
  else
    return false;
}
bool CirQueue::isFull()
{
    if (counter < maxSize)
    return false;
  else
    return true;
}
```

Priority Queues

Introduction

- Stack and Queue are data structures whose elements are ordered based on a sequence in which they have been inserted
- E.g. pop() function removes the item pushed last in the stack
- Intrinsic order among the elements themselves (e.g. numeric or alphabetic order etc.) is ignored in a stack or a queue

Definition

- A priority queue is a data structure in which prioritized insertion and deletion operations on elements can be performed according to their priority values.

- There are two types of priority queues:
  - Ascending Priority queue, and a
  - Descending Priority queue

Types of Priority Queue

- **Ascending Priority queue**: a collection of items into which items can be inserted ***randomly*** but only the ***smallest*** item can be removed
- If "**A-Priority-Q**" is an ascending priority queue then
  - Enqueue() will insert item 'x' into **A-Priority-Q**,
  - minDequeue() will remove the minimum item from **A-Priority-Q** and return its value

Types of Priority Queue

- **Descending Priority queue**: a collection of items into which items can be inserted ***randomly*** but only the ***largest*** item can be removed
- If "**D-Priority-Q**" is a descending priority queue then
  - Enqueue() will insert item x into **D-Priority-Q**,
  - maxDequeue( ) will remove the maximum item from **D-Priority-Q** and return its value

Generally

- In both the above types, if elements with equal priority are present, the FIFO technique is applied.

- Both types of priority queues are similar in a way that both of them remove and return the element with the highest **“Priority”** when the function remove() is called.
  - For an ascending priority queue item with smallest value has maximum “priority”
  - For a descending priority queue item with highest value has maximum “priority”

- This implies that we must have criteria for a priority queue to determine the Priority of its constituent elements.
- the elements of a priority queue can be numbers, characters or any complex structures such as phone book entries, events in a simulation

Priority Queue Issues

- In what manner should the items be inserted in a priority queue
  - Ordered (so that retrieval is simple, but insertion will become complex)
  - Arbitrary (insertion is simple but retrieval will require elaborate search mechanism)
- Retrieval
  - In case of un-ordered priority queue, what if minimum number is to be removed from an ascending queue of n elements (n number of comparisons)
- In what manner should the queue be maintained when an item is removed from it
  - Emptied location is kept blank (how to recognize a blank location ??)
  - Remaining items are shifted

Data Structure & Algorithms

Lecture 6

Linked List

Definition - List

- A list is a collection of items that has a particular order
  - It can have an arbitrary length
  - Objects / elements can be inserted or removed at arbitrary locations in the list
  - A list can be traversed in order one item at a time

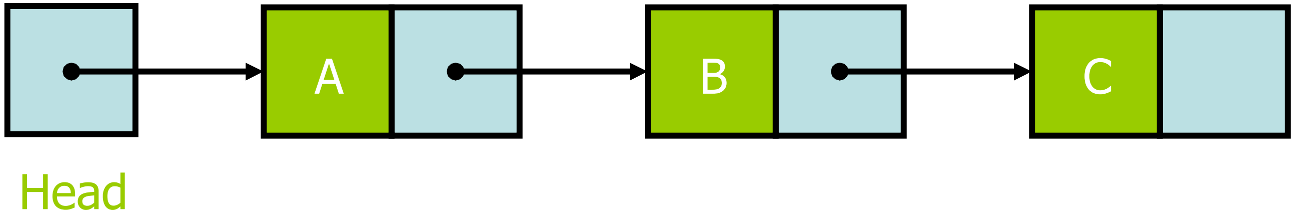
List Overview

- Linked lists
  - Abstract data type (ADT)
- Basic operations of linked lists
  - Insert, find, delete, print, etc.
- Variations of linked lists
  - Singly linked lists
  - Circular linked lists
  - Doubly linked lists
  - Circular doubly linked list

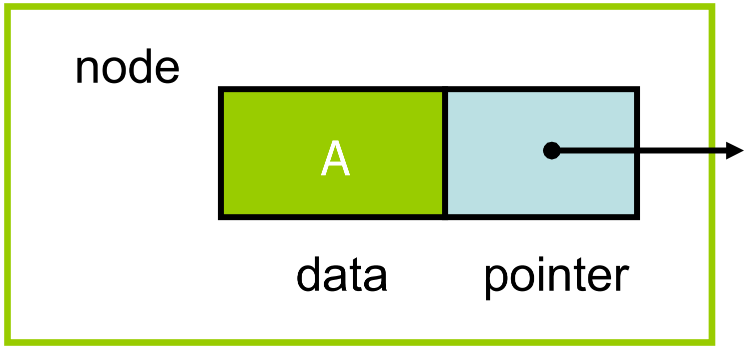
Linked List Terminologies

- **Traversal of List**
  - Means to visit every element or node in the list beginning from first to last.
- **Predecessor and Successor**
  - In the list of elements, for any location n, (n-1) is predecessor and (n+1) is successor.
  - In other words, for any location n in the list, the left element is predecessor and the right element is successor.
  - Also, the first element does not have predecessor and the last element does not have successor.

Linked Lists



- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- *Head*: pointer to the first node
- The last node points to `NULL`

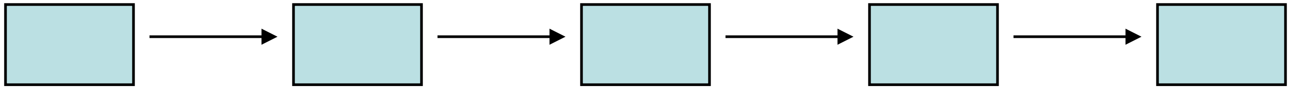


Lists – Another perspective

A list is a linear collection of varying length of homogeneous components.

Homogeneous: All components are of the same type.

Linear: Components are ordered in a line (hence called Linear linked lists).



Arrays are lists..

Arrays Vs Lists

- Arrays are lists that have a fixed size in memory.
- The programmer must keep track of the length of the array
- No matter how many elements of the array are used in a program, the array has the same amount of allocated space.
- Array elements are stored in successive memory locations. Also, order of elements stored in array is same logically and physically.

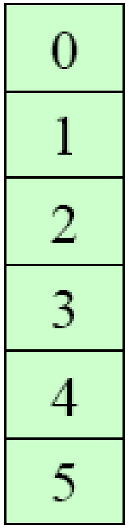
Arrays Vs Lists

- A linked list takes up only as much space in memory as is needed for the length of the list.
- The list expands or contracts as you add or delete elements.
- In linked list the elements are not stored in successive memory location
- Elements can be added to (or deleted from) either end, or added to (or deleted from)the middle of the list.

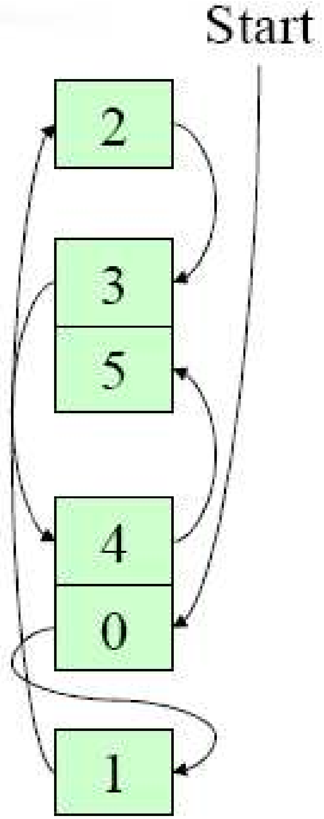
Array versus Linked Lists

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - **Dynamic**: a linked list can easily grow and shrink in size.
    - We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - In contrast, the size of a C++ array is fixed at compilation time.
  - **Easy and fast insertions and deletions**
    - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
    - With a linked list, no need to move other nodes. Only need to reset some pointers.

An Array



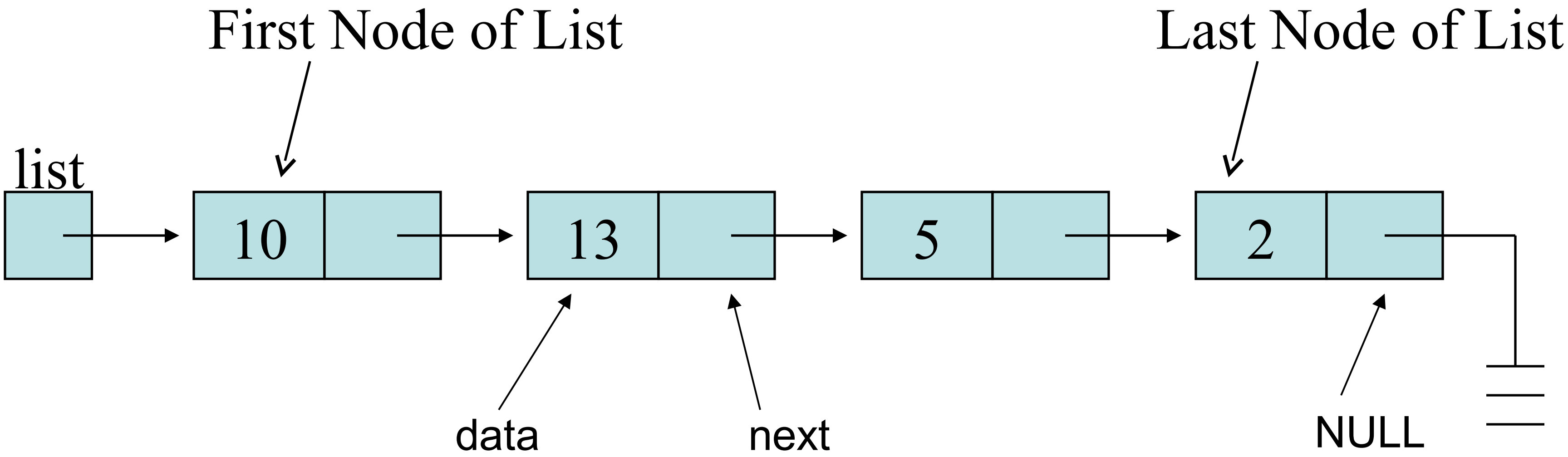
A Linked List



Basic Operations of Linked List

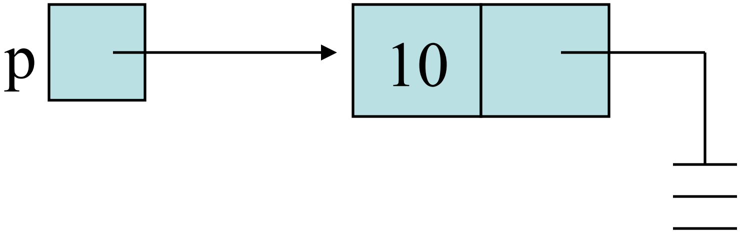
- Operations of `Linked List`
  - `IsEmpty`: determine whether or not the list is empty
  - `InsertNode`: insert a new node at a particular position
  - `FindNode`: find a node with a given value
  - `DeleteNode`: delete a node with a given value
  - `DisplayList`: print all the nodes in the list

An integer linked list



Creating a List node

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

Node *p;
p = new Node;
p - > data = 10;
p - > next = NULL;
```



The NULL pointer

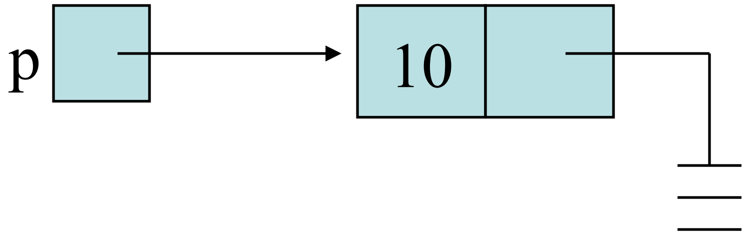
NULL is a special pointer value that does not reference any memory cell.

If a pointer is not currently in use, it should be set to NULL so that one can determine that it is not pointing to a valid address:

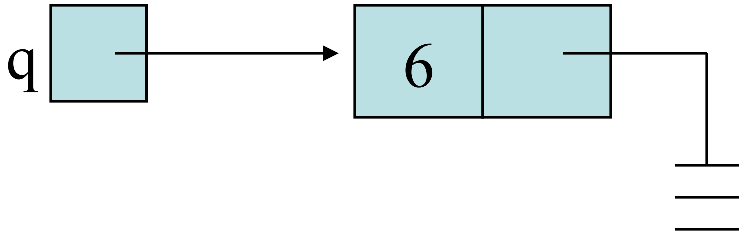
```
int *p;
p = NULL;
```

Adding a node to a list

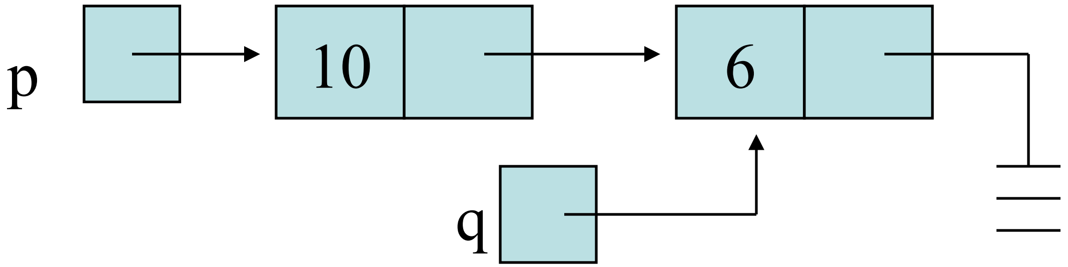
```
Node *p, *q;

p = new Node;
p - > data = 10;
p - > next = NULL;
```



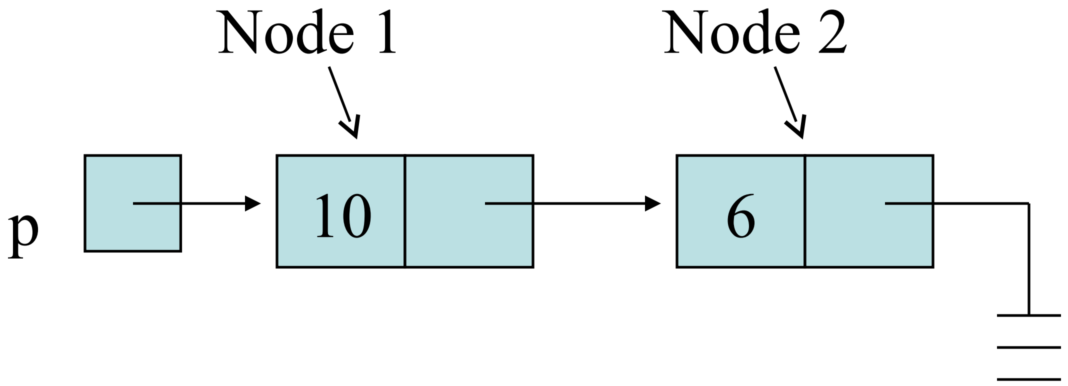
```
q = new Node;
q - > data = 6;
q - > next = NULL;
```



```
p - > next = q;
```



Accessing List Data



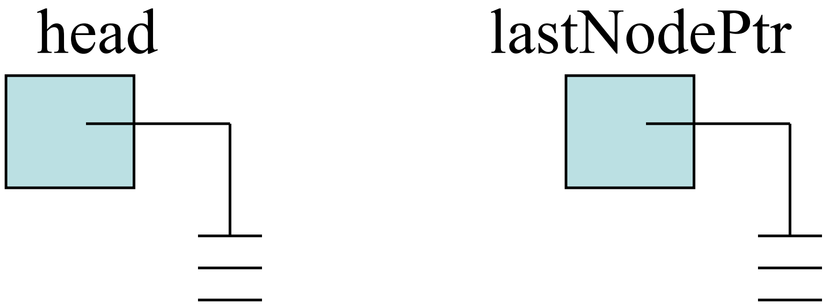
Expression	Value
p	Pointer to first node (head)
p - > data	10
p - > next	Pointer to next node
p - > next - > data	6
p - > next - > next	NULL pointer

Using typedef with pointers

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

typedef Node *NodePtr;     // NodePtr type is a pointer
                // to a Node
Node *p;               // p is a pointer to a Node
NodePtr q;                // q is a pointer to a Node
```

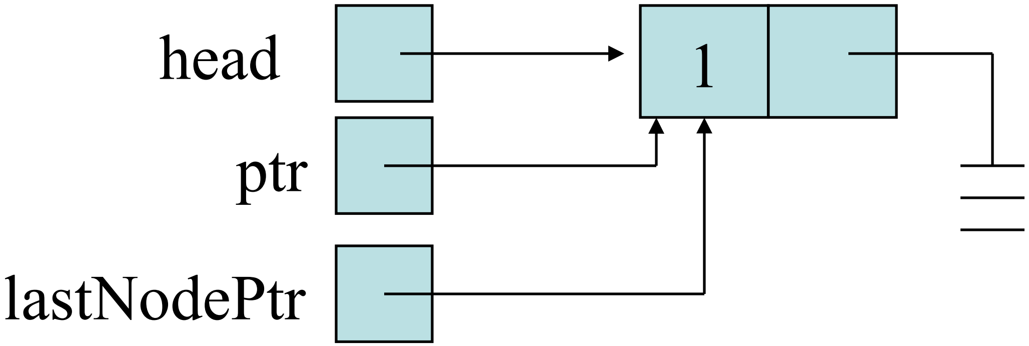
Building a list from 1 to n

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

Node *head = NULL;           // pointer to the list head
Node *lastNodePtr = NULL;    // pointer to last node in list
```



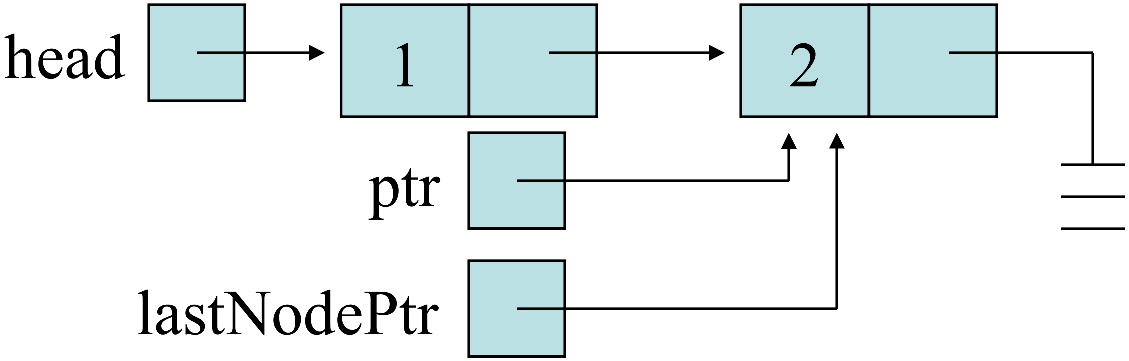
Creating the first node

```
Node *ptr;           // declare a pointer to Node
ptr = new Node;             // create a new Node
ptr - > data = 1;
ptr - > next = NULL;

head = ptr;          // new node is first
lastNodePtr = ptr;     // and last node in list
```

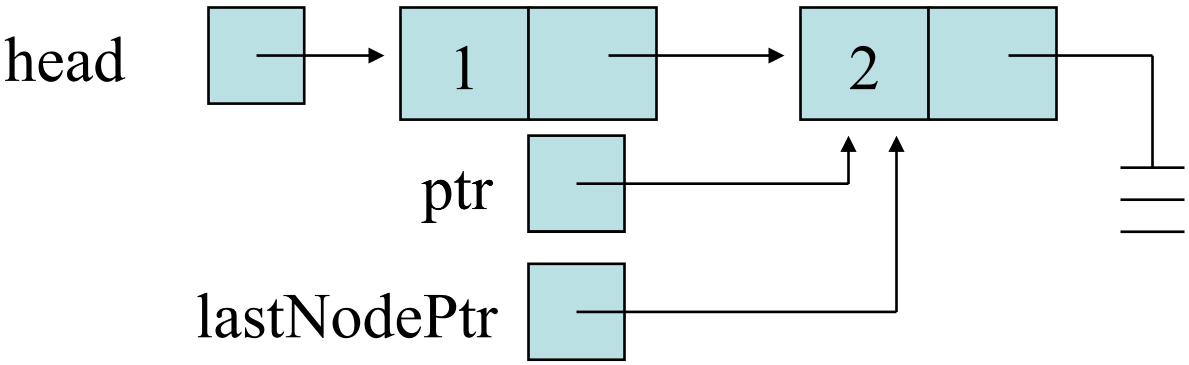


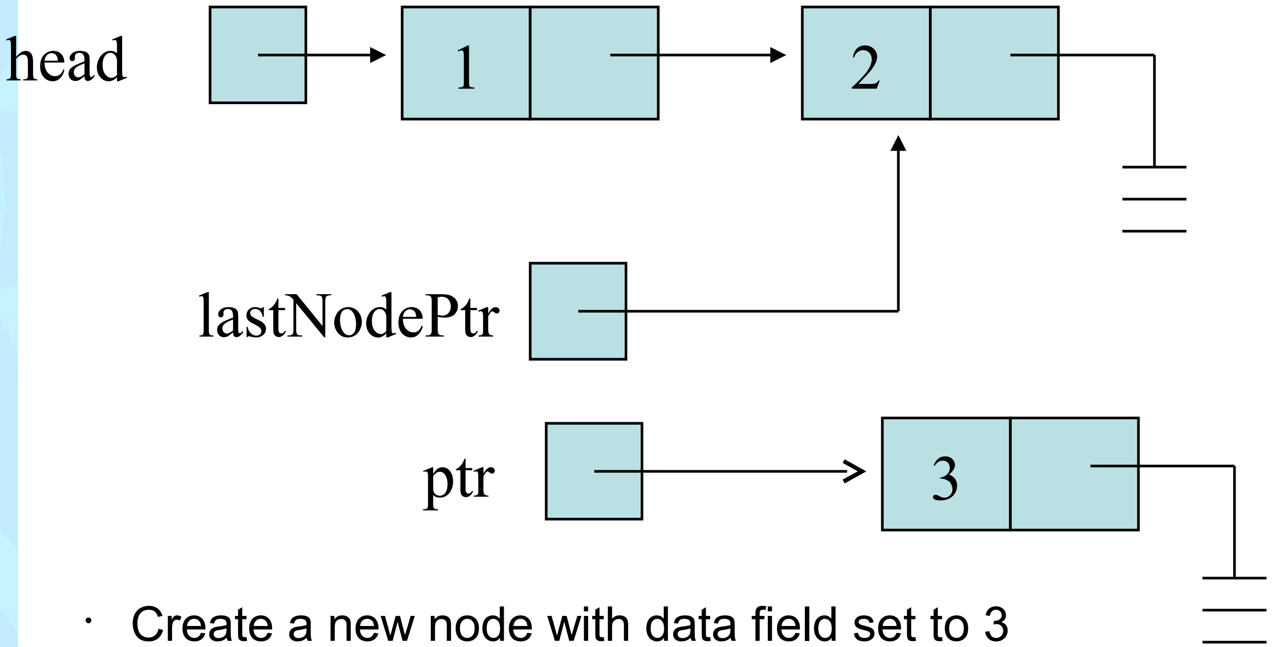
Adding more nodes

```
for (int i = 2; i < = n; i ++ ) {
     ptr = new Node;       //create new node
     ptr - > data = i;
     ptr - > next = NULL;
     lastNodePtr - > next = ptr;  // order is
     lastNodePtr = ptr;           // important
}
```

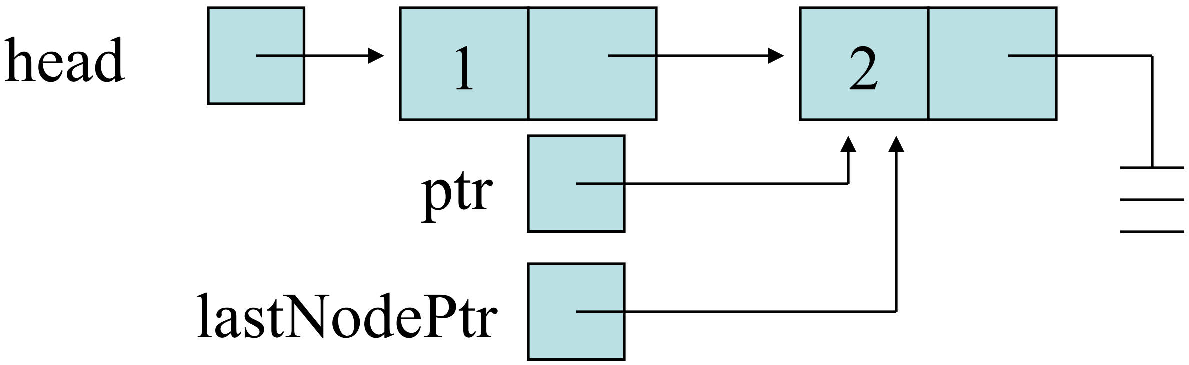


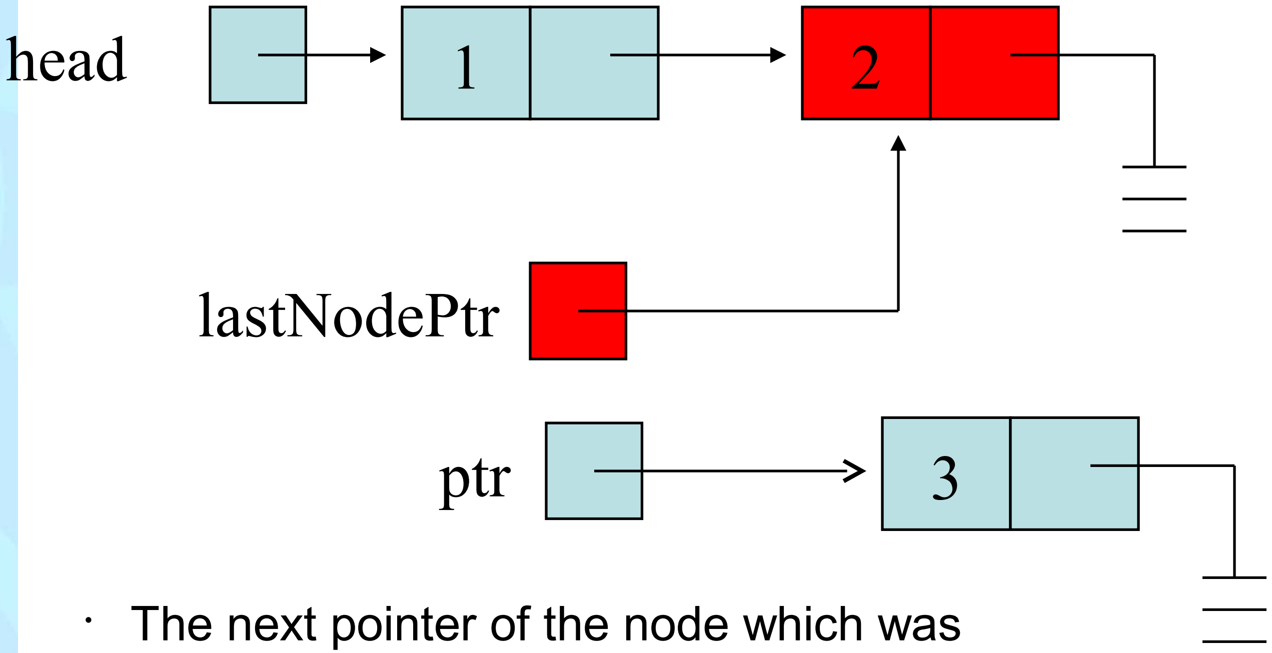
Initially



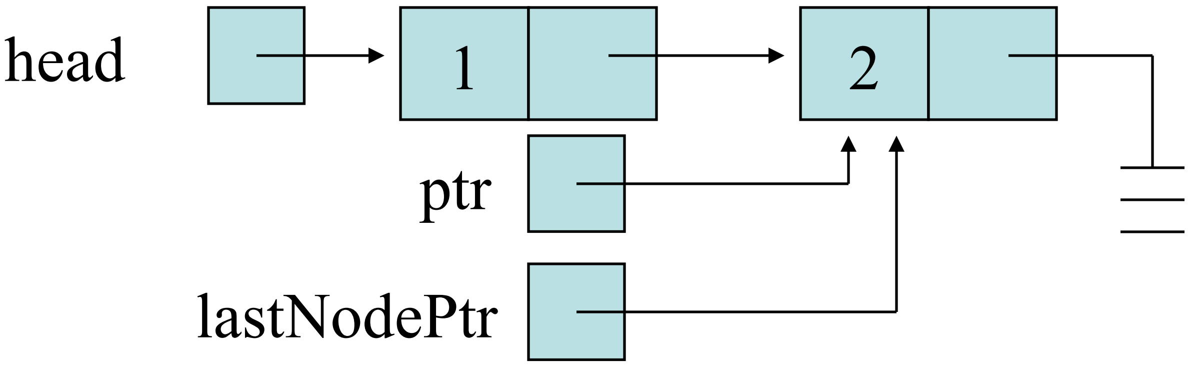


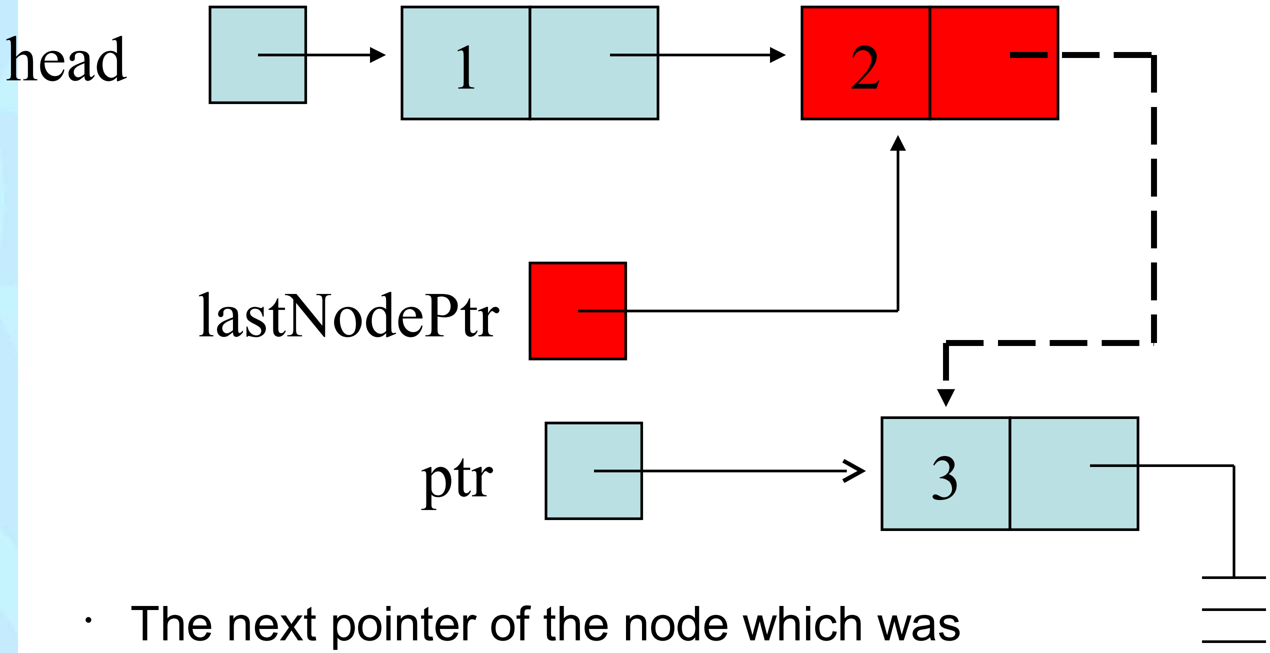
- Create a new node with data field set to 3
- Its next pointer should point to NULL



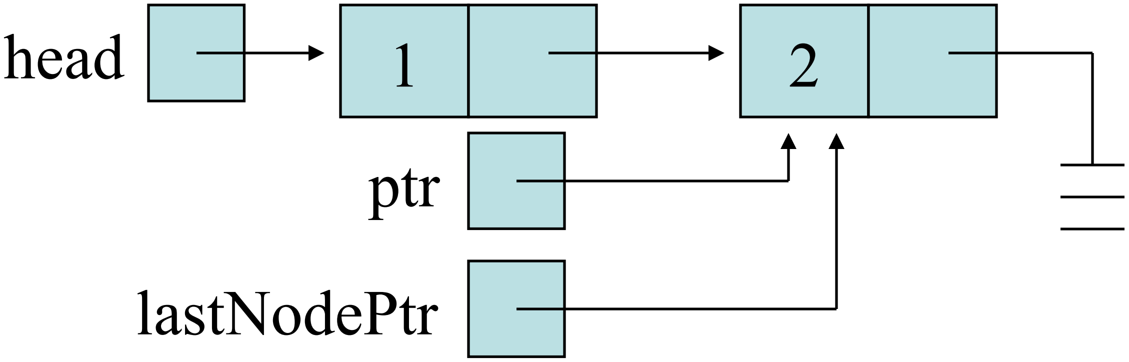


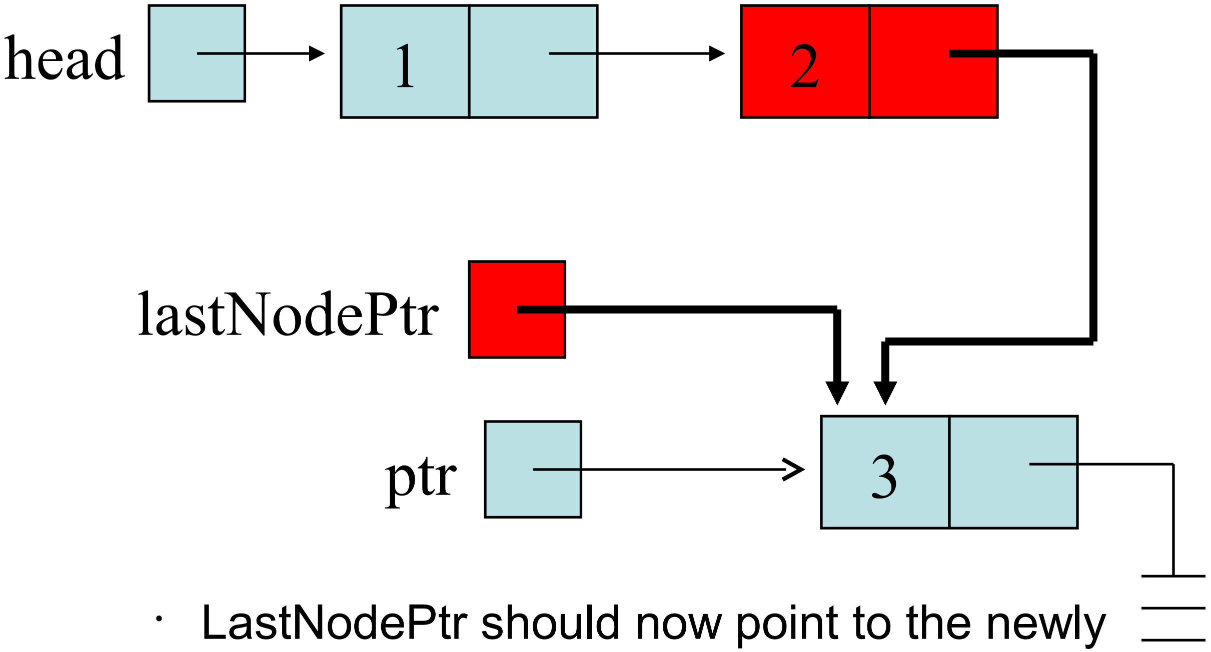
- The next pointer of the node which was previously last should now point to newly created node “lastNodePtr->next=ptr”





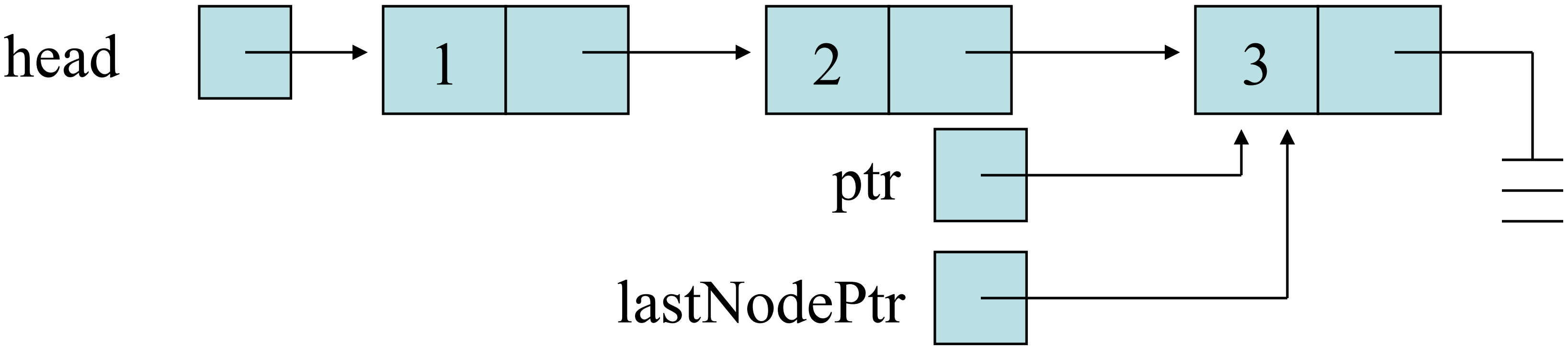
- The next pointer of the node which was previously last should now point to newly created node “lastNodePtr->next=ptr”
- LastNodePtr should now point to the newly created Node “lastNodePtr = ptr;”



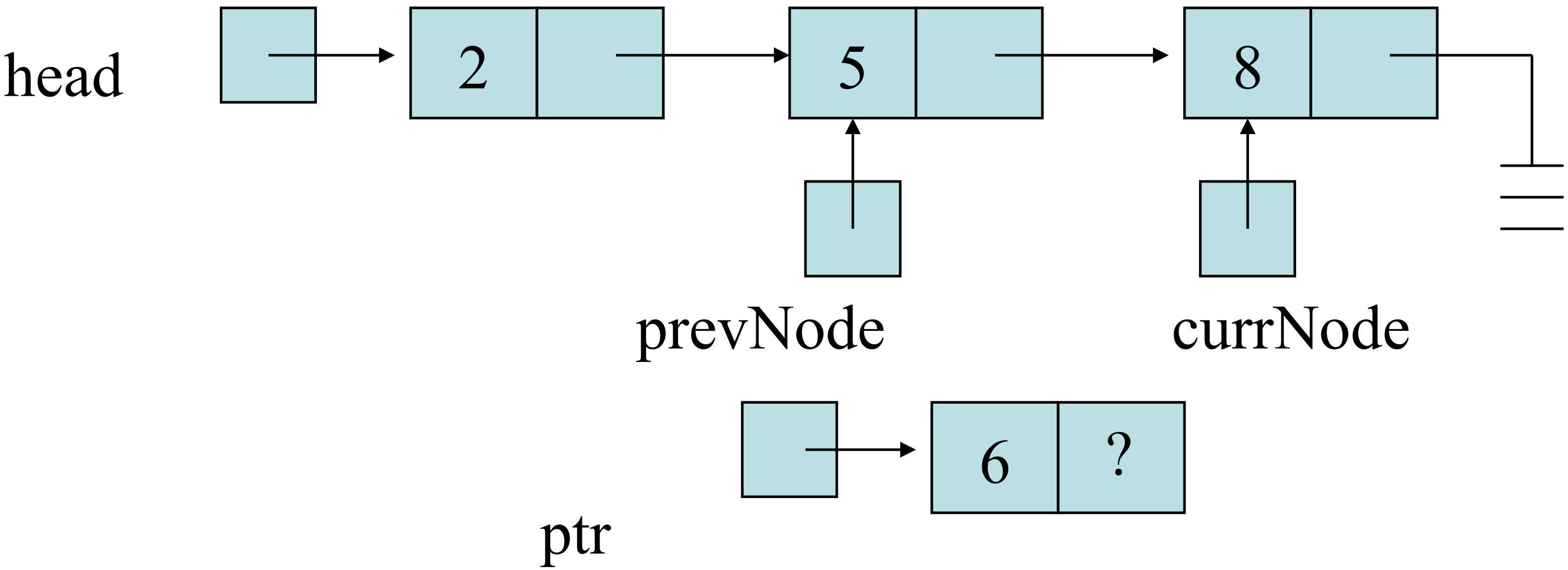


- LastNodePtr should now point to the newly created Node “lastNodePtr = ptr;”

Re-arranging the view



Inserting a node in a list



Step 1: Determine where you want to insert a node.

Step 2: Create a new node:

```
Node *ptr;
ptr = new Node;
ptr - > data = 6;
```

```
Node *ptr, *currNode, *prevNode ;
prevNode = head;
ptr = new Node;
ptr->data = 6;
ptr->next = NULL;
currNode = head->next;
While (currNode->data   <   ptr->data)
{
    prevNode = currNode;
    currNode = currNode->next;
}
```

Note:

when this loop terminates **prevNode** and **currNode** are at a place where insertion will take place. Only the “LINKS” or pointers of the list remain to be adjusted

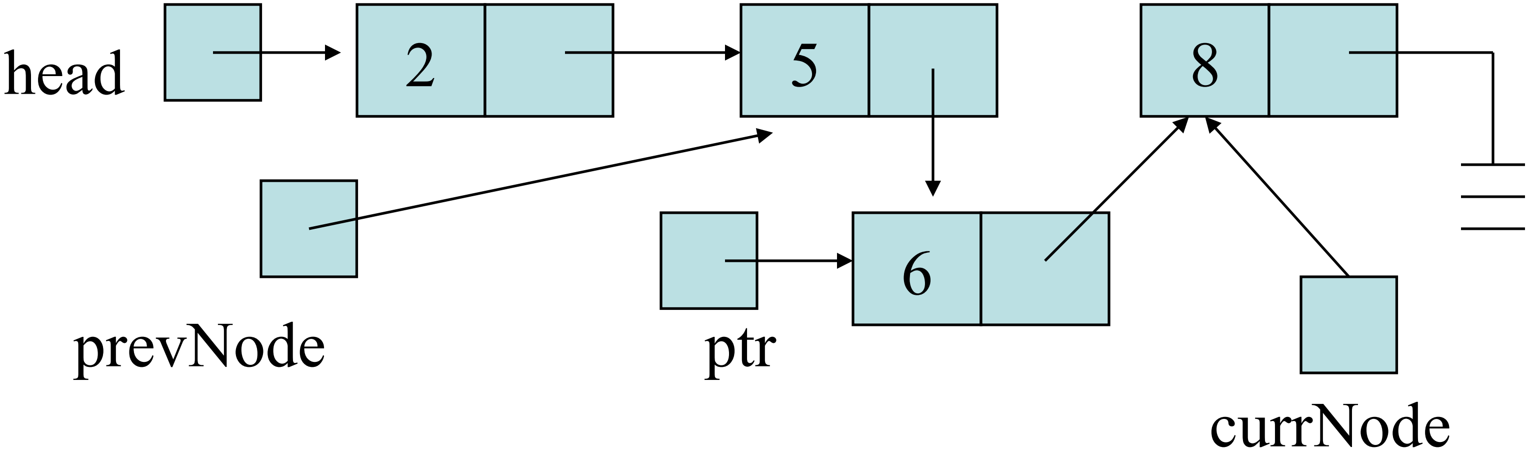
Continuing the insert

Step 3: Make the new node point to the current Node pointer.

ptr - > next = currNode;

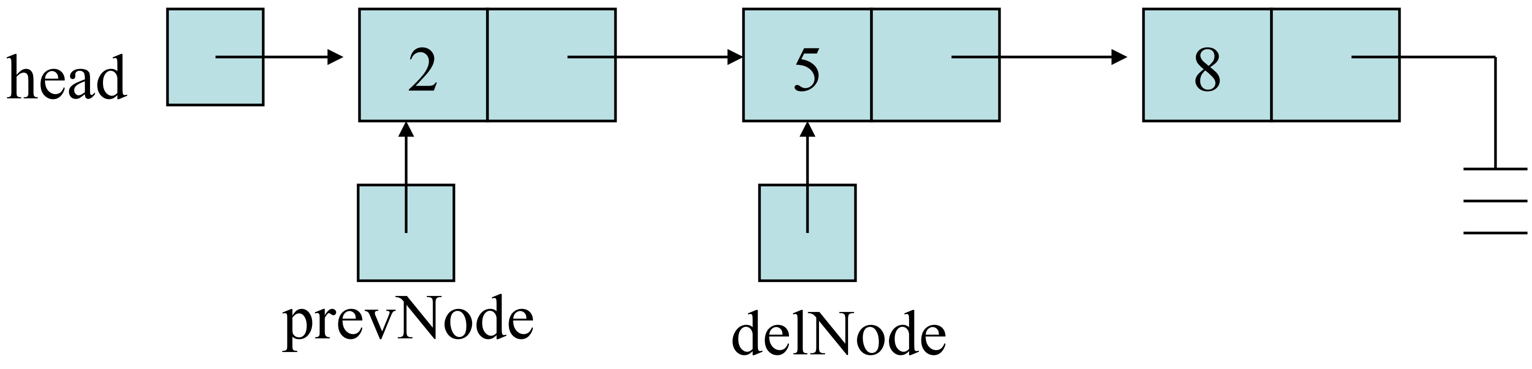
Step 4: Make previous node point to the new node:

prevNode - > next = ptr;



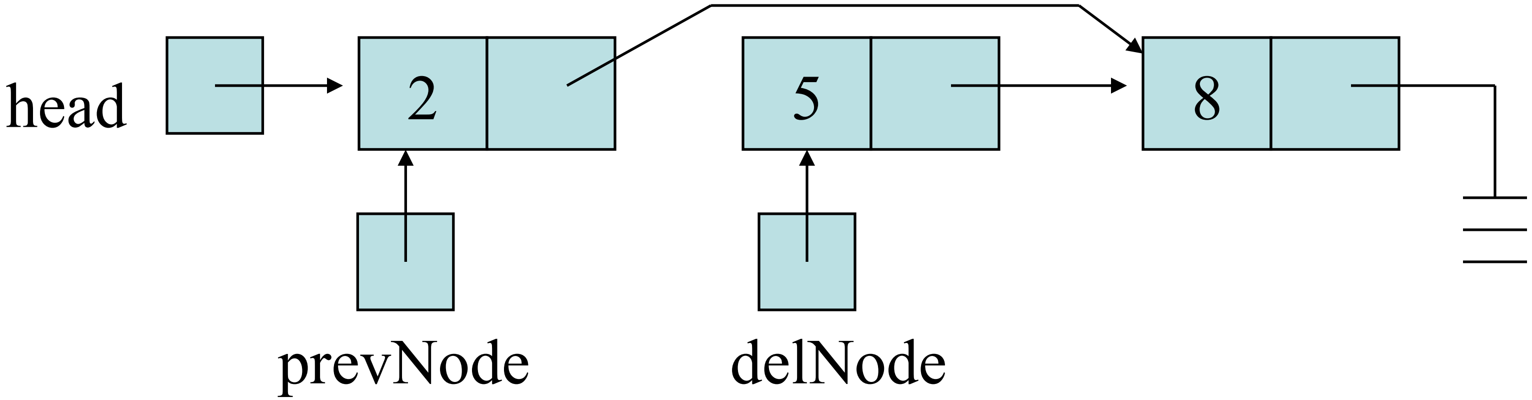
Now The new link has been added in the linked list

Deleting a node from a list



Step 1: Redirect pointer from the Node before the one to be deleted to point to the Node after the one to be deleted.

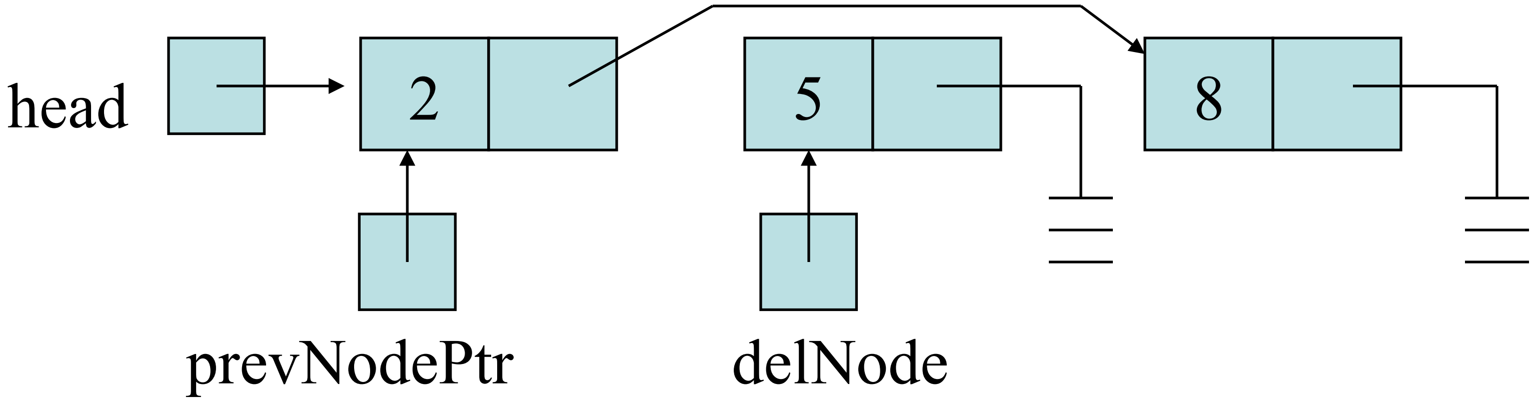
prevNode - > next = delNode - > next;



Finishing the deletion

Step 2: Remove the pointer from the deleted link.

delNode - > next = NULL;

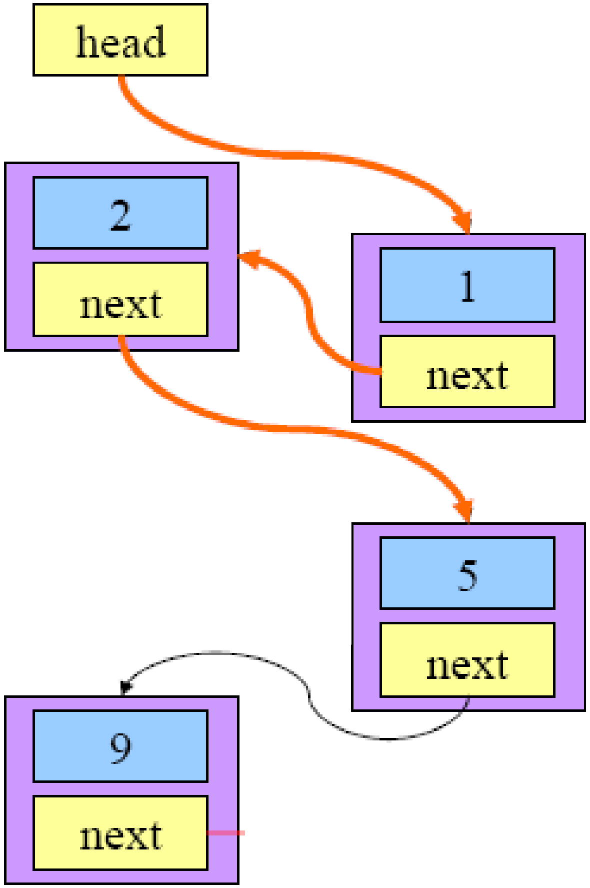


Step 3: Free up the memory used for the deleted node:

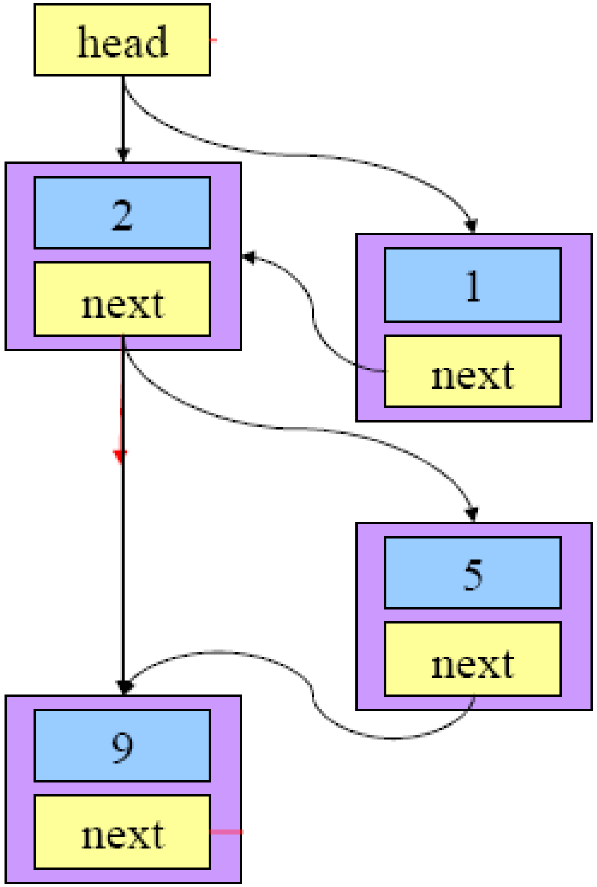
delete delNode;

List Operations - Summarized

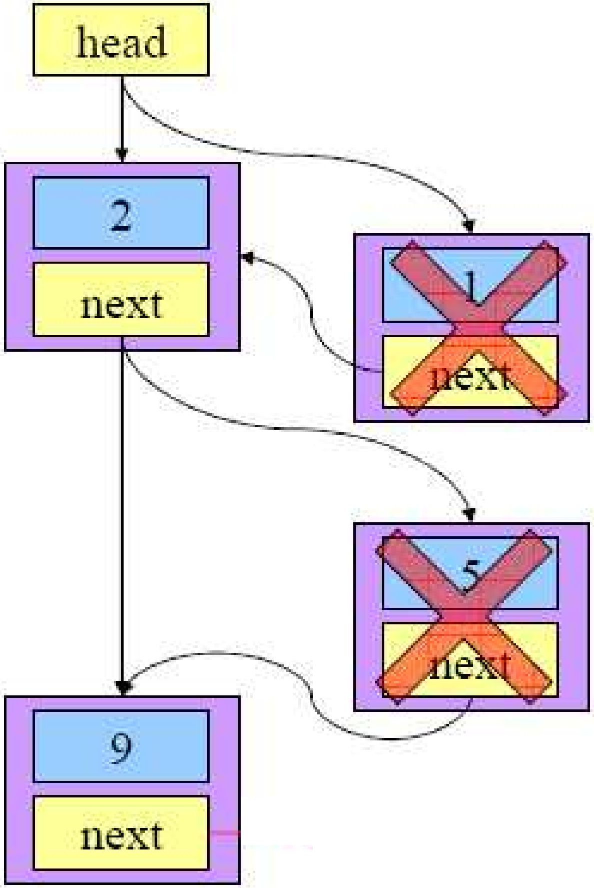
Traversing a Linked List



Insertion in a Linked List



Deletion from a Linked List



MCS

Doubly Linked List

Introduction

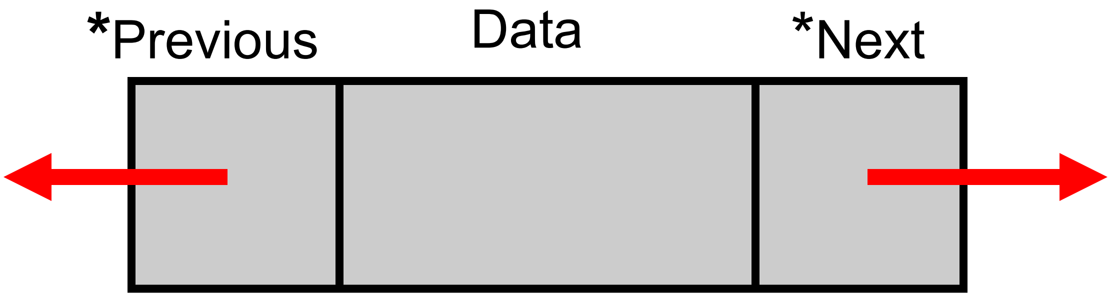
- The singly linked list contains only one pointer field i.e. every node holds an address of next node.

- The singly linked list is uni-directional i.e. we can only move from one node to its successor.

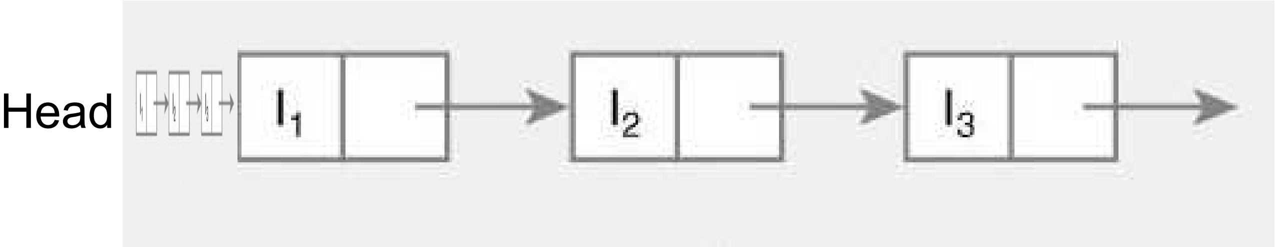
- This limitation can be overcome by **Doubly linked list.**

Doubly Linked List

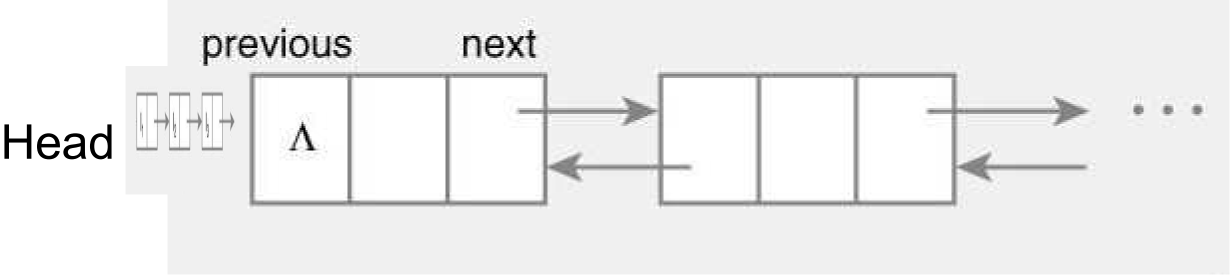
- In Doubly linked list, each node has two pointers.
- One pointer to its successor (NULL if there is none) and one pointer to its predecessor (NULL if there is none).
- These pointers enable bi-directional traversing.



A Singly Linked List

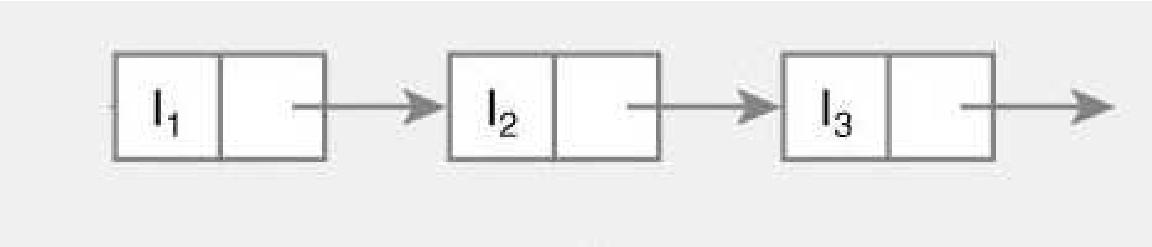


A Doubly Linked List



Comparison of Linked Lists

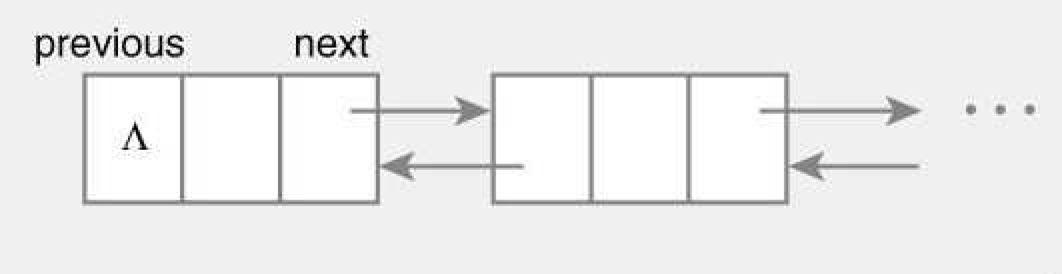
- Linked list



Node* next;

};

- Doubly linked list



int data;

Node *next;

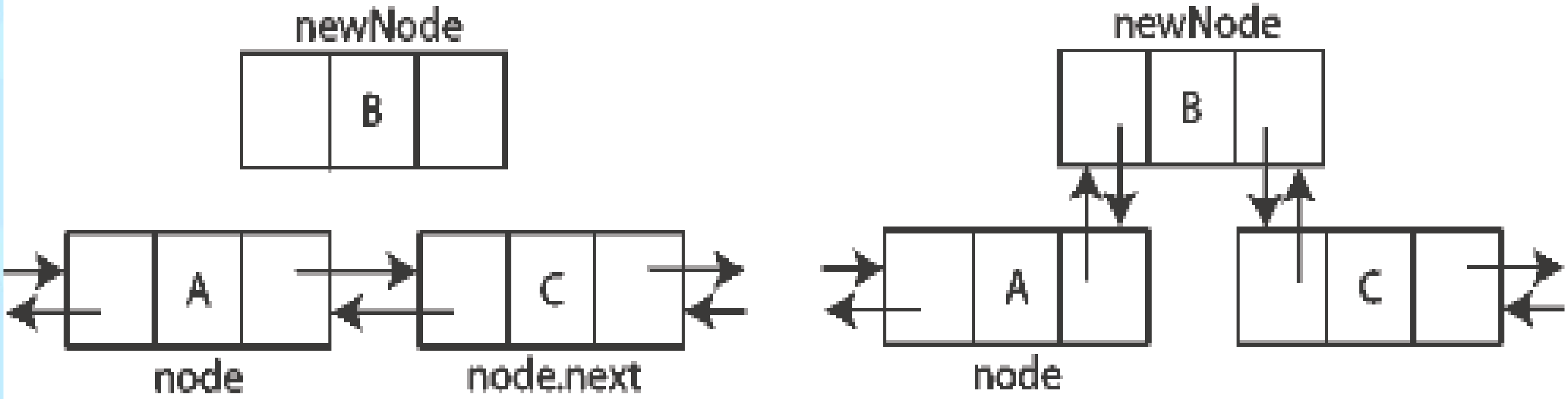
};

Insertion

- In insertion process, element can be inserted in three different places
  - At the beginning of the list
  - At the end of the list
  - At the specified position.

- To insert a node in doubly linked list, you must update pointers in both predecessor and successor nodes.

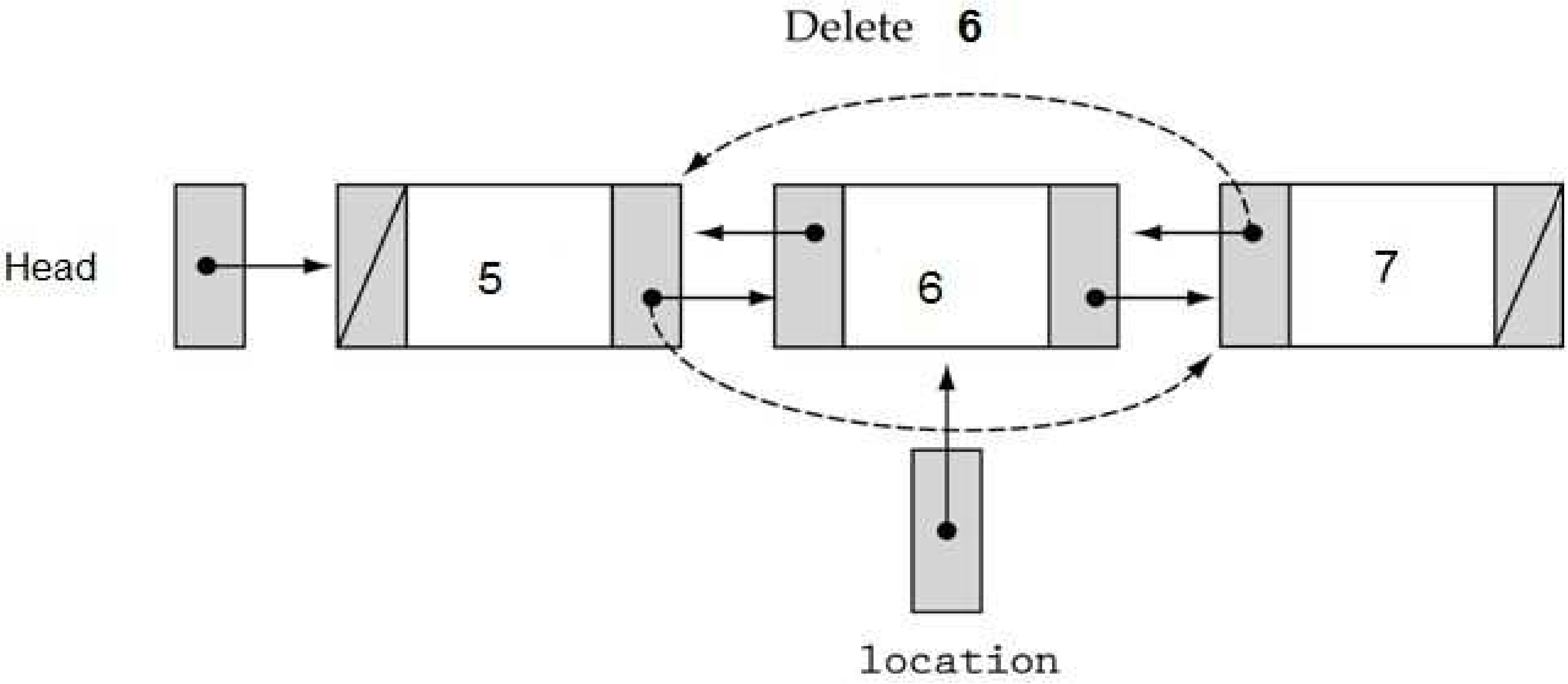
Insertion



Deletion

- In deletion process, element can be deleted from three different places
  - From the beginning of the list
  - From the end of the list
  - From the specified position in the list.
- When the node is deleted, the memory allocated to that node is released and the previous and next nodes of that node are linked

Deletion



Advantages of Doubly Linked List

1. The doubly linked list is bi-directional, i.e. it can be traversed in both backward and forward direction.
2. The operations such as insertion, deletion and searching can be done from both ends.
3. Predecessor and successor of any element can be searched quickly

Disadvantages

1. It consume more memory space.
2. There is a large pointer adjustment during insertion and deletion of element.
3. It consumes more time for few basic list operations.

Adding two polynomials using Linked List

To understand this concept better let's first brush up all the basic contents that are required.

linked list is a data structure that stores each element as an object in a node of the list. every note contains two parts data han and links to the next node.

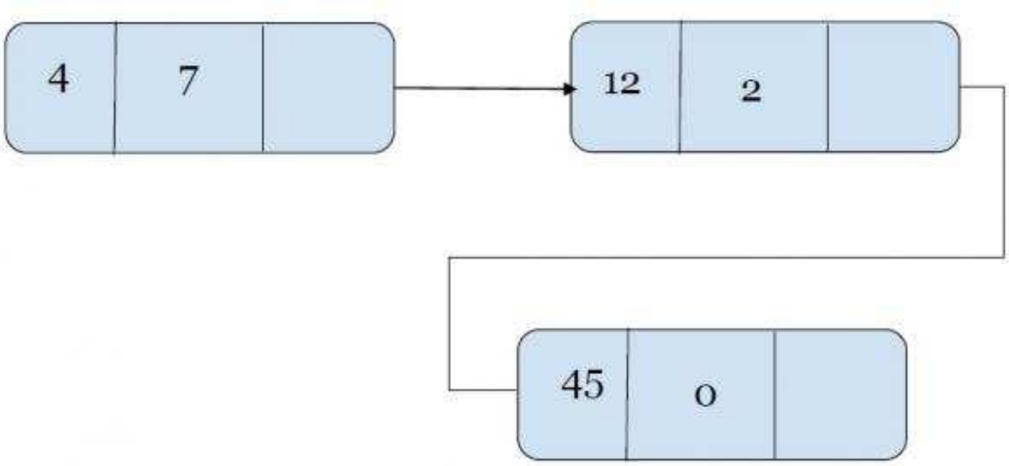
Polynomial is a mathematical expression that consists of variables and coefficients. for example x^2 - 4x + 7

In the **Polynomial linked list**, the coefficients and exponents of the polynomial are defined as the data node of the list.

For adding two polynomials that are stored as a linked list. We need to add the coefficients of variables with the same power. In a linked list node contains 3 members, coefficient value link to the next node.

a linked list that is used to store Polynomial looks like −

Polynomial : $4x^7 + 12x^2 + 45$



This is how a linked list represented polynomial looks like.

Adding two polynomials that are represented by a linked list. We check values at the exponent value of the node. For the same values of exponent, we will add the coefficients.

Example,

```
Input :
p1= 13x^8 + 7x^5 + 32x^2 + 54
p2= 3x^12 + 17x^5 + 3x^3 + 98
```

Output : $3x^{12} + 13x^{8} + 24x^{5} + 3x3 + 32x^{2} + 152$

Explanation − For all power, we will check for the coefficients of the exponents that have the same value of exponents and add them. The return the final polynomial.

Algorithm

Input − polynomial p1 and p2 represented as a linked list.

```
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.
```

Example

```
#include<bits/stdc++.h>

using namespace std;

struct Node{

   int coeff;

   int pow;

   struct Node *next;

};

void create_node(int x, int y, struct Node **temp){

   struct Node *r, *z;

   z = *temp;

   if(z == NULL){

      r =(struct Node*)malloc(sizeof(struct Node));

      r->coeff = x;

      r->pow = y;
      *temp = r;
      r->next = (struct Node*)malloc(sizeof(struct Node));
      r = r->next;
      r->next = NULL;
   } else {
      r->coeff = x;
      r->pow = y;
      r->next = (struct Node*)malloc(sizeof(struct Node));
      r = r->next;
      r->next = NULL;
   }
}
void polyadd(struct Node *p1, struct Node *p2, struct Node
*result){
   while(p1->next && p2->next){
      if(p1->pow > p2->pow){
         result->pow = p1->pow;
         result->coeff = p1->coeff;
         p1 = p1->next;
      }
      else if(p1->pow < p2->pow){
         result->pow = p2->pow;
         result->coeff = p2->coeff;
         p2 = p2->next;
      } else {
         result->pow = p1->pow;
         result->coeff = p1->coeff+p2->coeff;
         p1 = p1->next;
         p2 = p2->next;
      }
      result->next = (struct Node *)malloc(sizeof(struct Node));
      result = result->next;
      result->next = NULL;
   }
   while(p1->next || p2->next){
      if(p1->next){
         result->pow = p1->pow;
         result->coeff = p1->coeff;
         p1 = p1->next;
      }
      if(p2->next){
         result->pow = p2->pow;
         result->coeff = p2->coeff;
         p2 = p2->next;
      }
      result->next = (struct Node *)malloc(sizeof(struct Node));
      result = result->next;
      result->next = NULL;
   }
}
void printpoly(struct Node *node){
   while(node->next != NULL){
      printf("%dx^%d", node->coeff, node->pow);
```

```c
      node = node->next;
      if(node->next != NULL)
         printf(" + ");
   }
}
int main(){
   struct Node *p1 = NULL, *p2 = NULL, *result = NULL;
   create_node(41,7,&p1);
   create_node(12,5,&p1);
   create_node(65,0,&p1);
   create_node(21,5,&p2);
   create_node(15,2,&p2);
   printf("polynomial 1: ");
   printpoly(p1);
   printf("\npolynomial 2: ");
   printpoly(p2);
   result = (struct Node *)malloc(sizeof(struct Node));
   polyadd(p1, p2, result);
   printf("\npolynomial after adding p1 and p2 : ");
   printpoly(result);
   return 0;
}
```