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

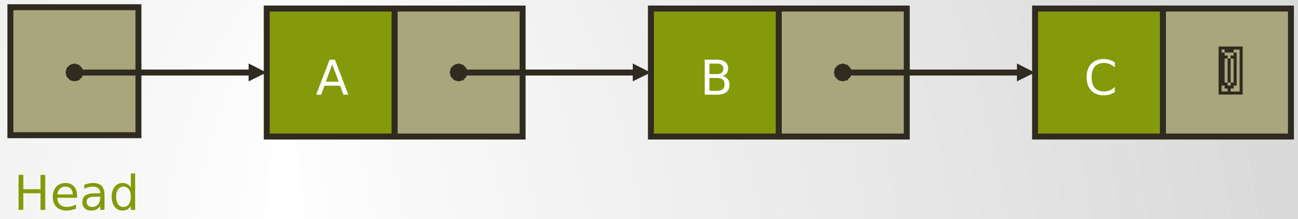
List Overview

- 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

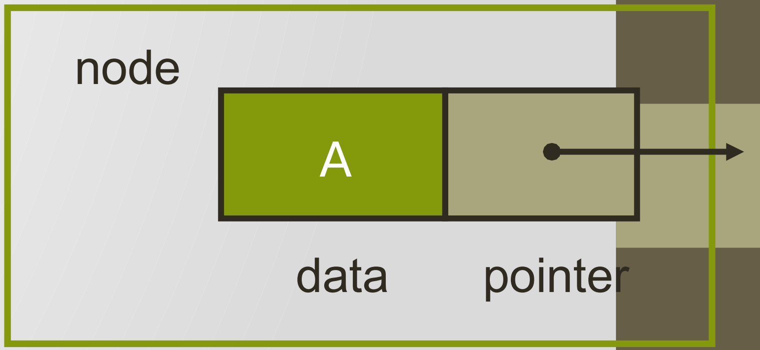
Linked List Terminologies

- **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.

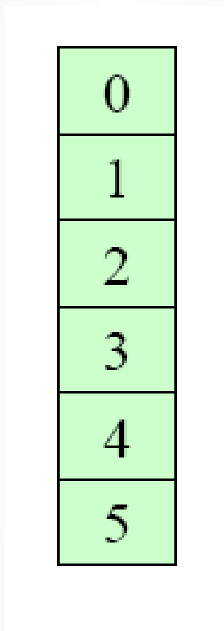
Arrays Vs Lists

- 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.

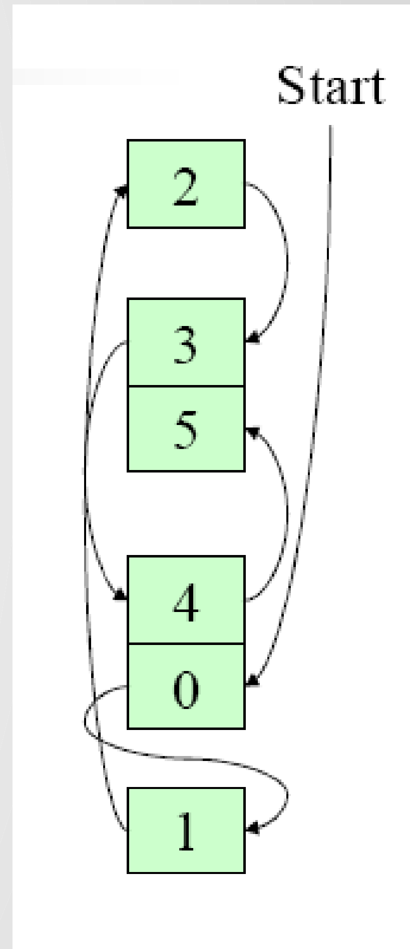
Array versus Linked Lists

- 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



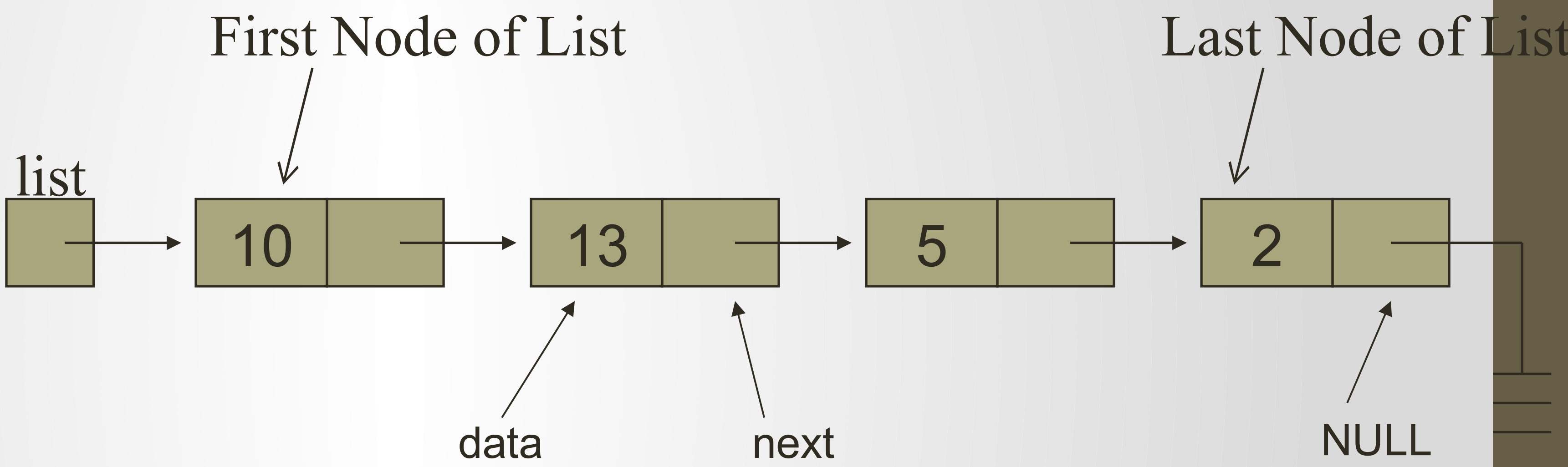
A Linked List



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

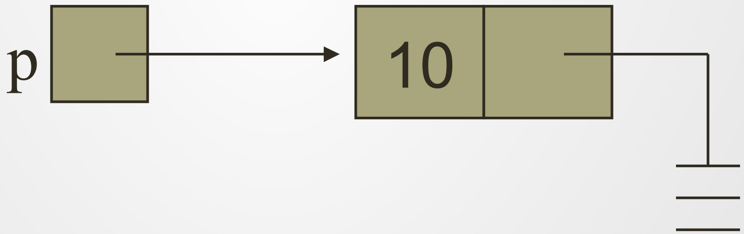
```
struct Node {
    int data;         // data in node
    Node *next;       // Pointer to next node
};
```

Node *p;

p = ~~new Node;~~ malloc(sizeof(struct Node))

p - > data = 10;

p - > next = NULL;



The NULL pointer

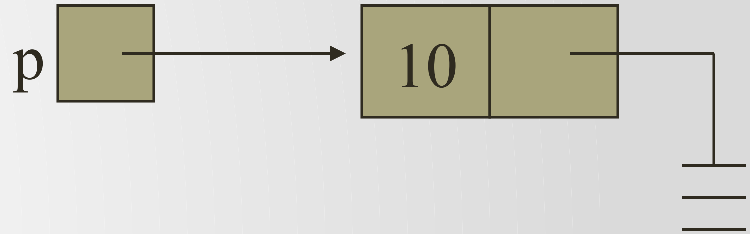
NULL is a special pointer value that does not reference any memory cell.

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:

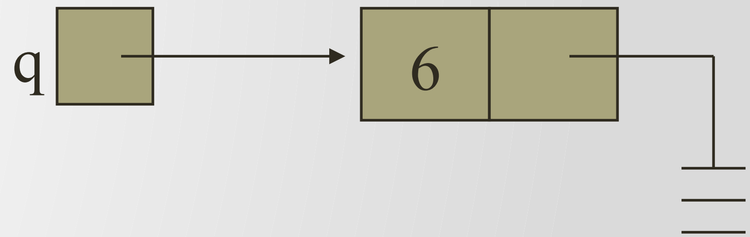
```
int *p;
p = NULL;
```

Adding a node to a list

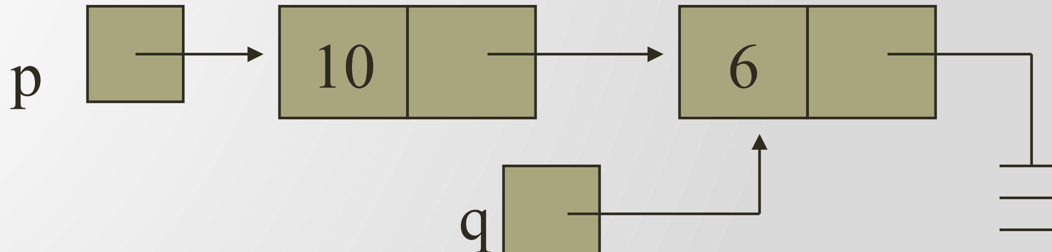
```
Node *p, *q;

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



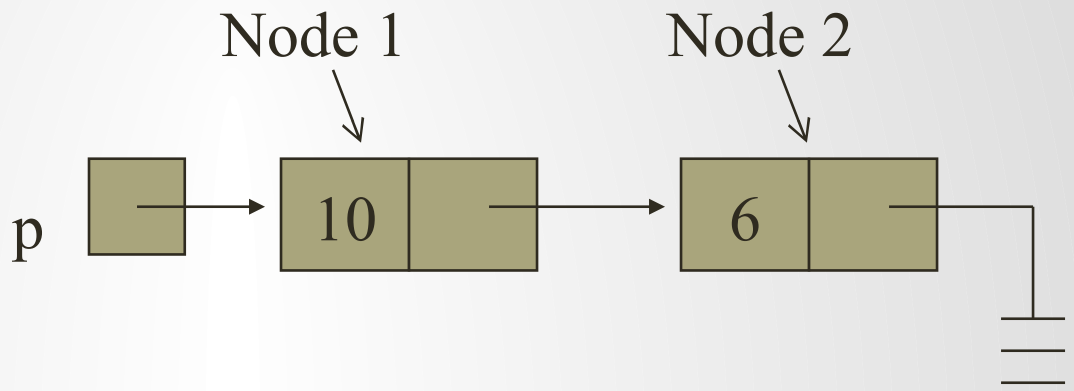

```
q = new Node;
q - > data = 6;
q - > next = NULL;
```




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



Accessing List Data



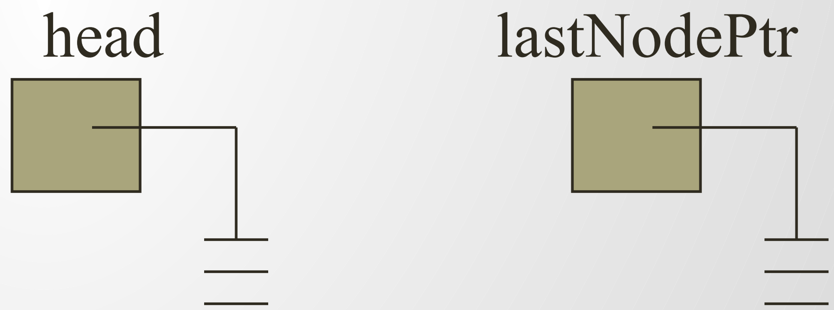
<u>Expression</u>	<u>Value</u>
p	Pointer to first node (head)
p - > data	10
p - > next	Pointer to next node
p - > next - > data	6
p - > next - > next	NULL pointer

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
```

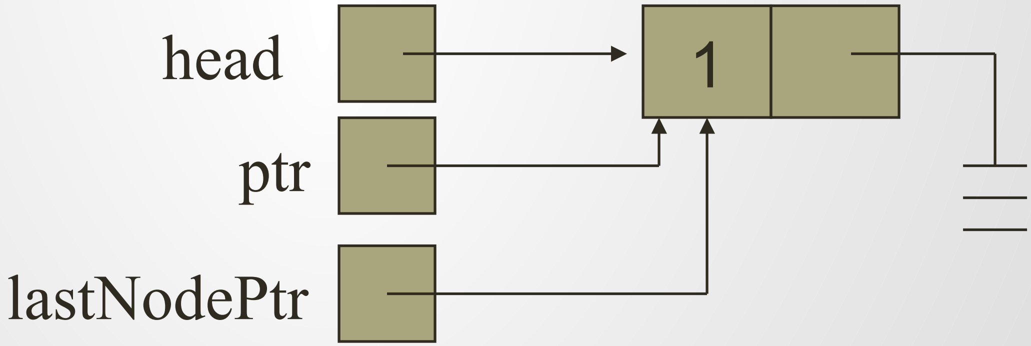
Building a list from 1 to n

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

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



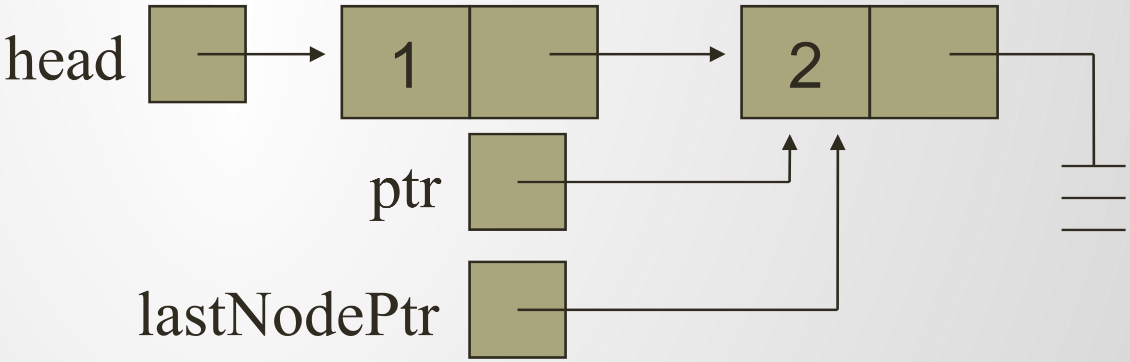
Creating the first node

```
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
```

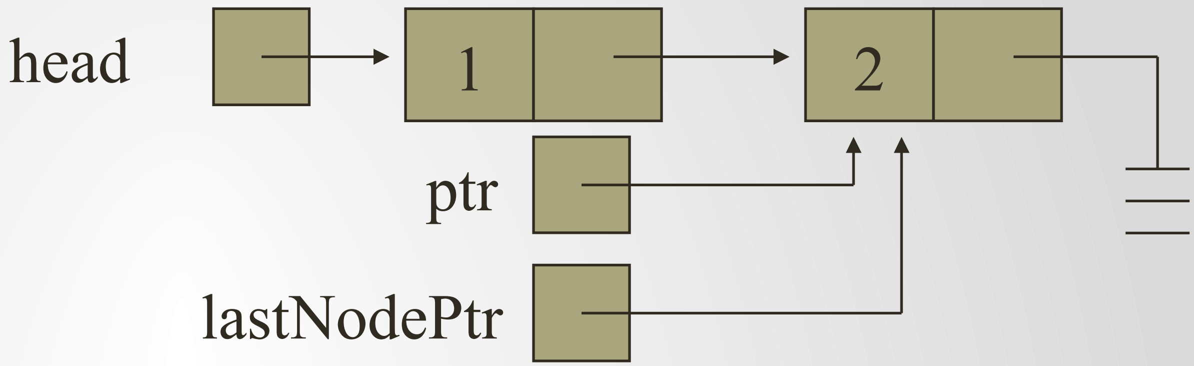


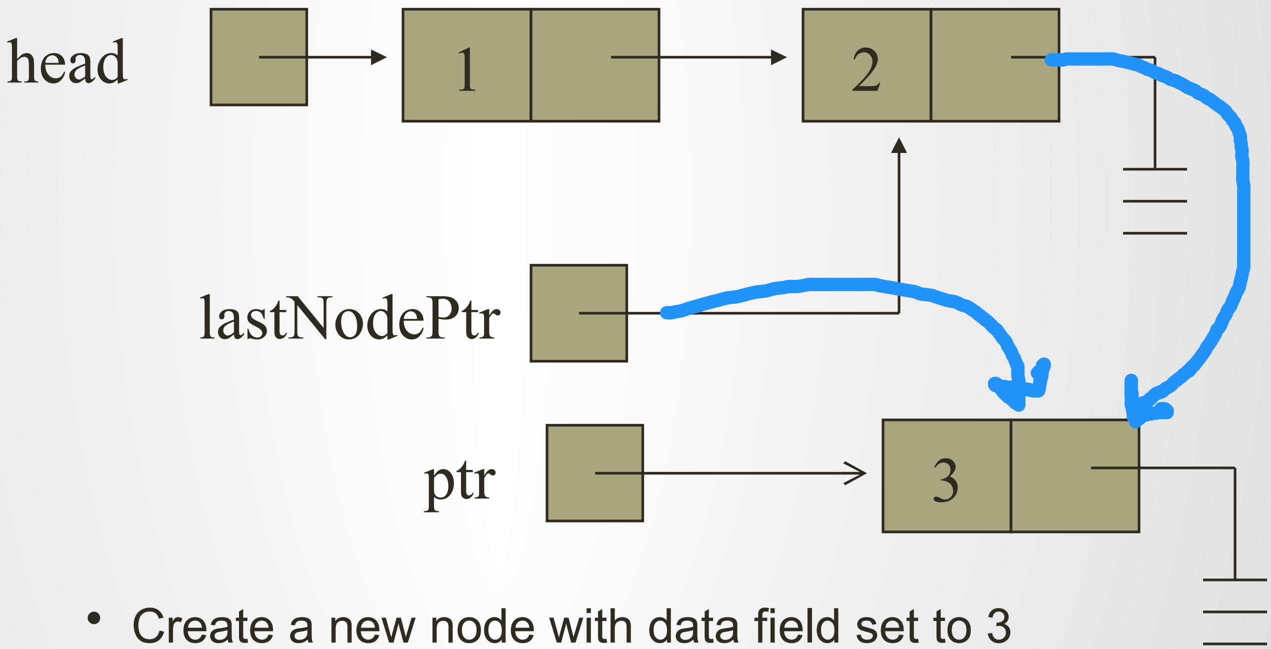
Adding more nodes

```
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
}
```

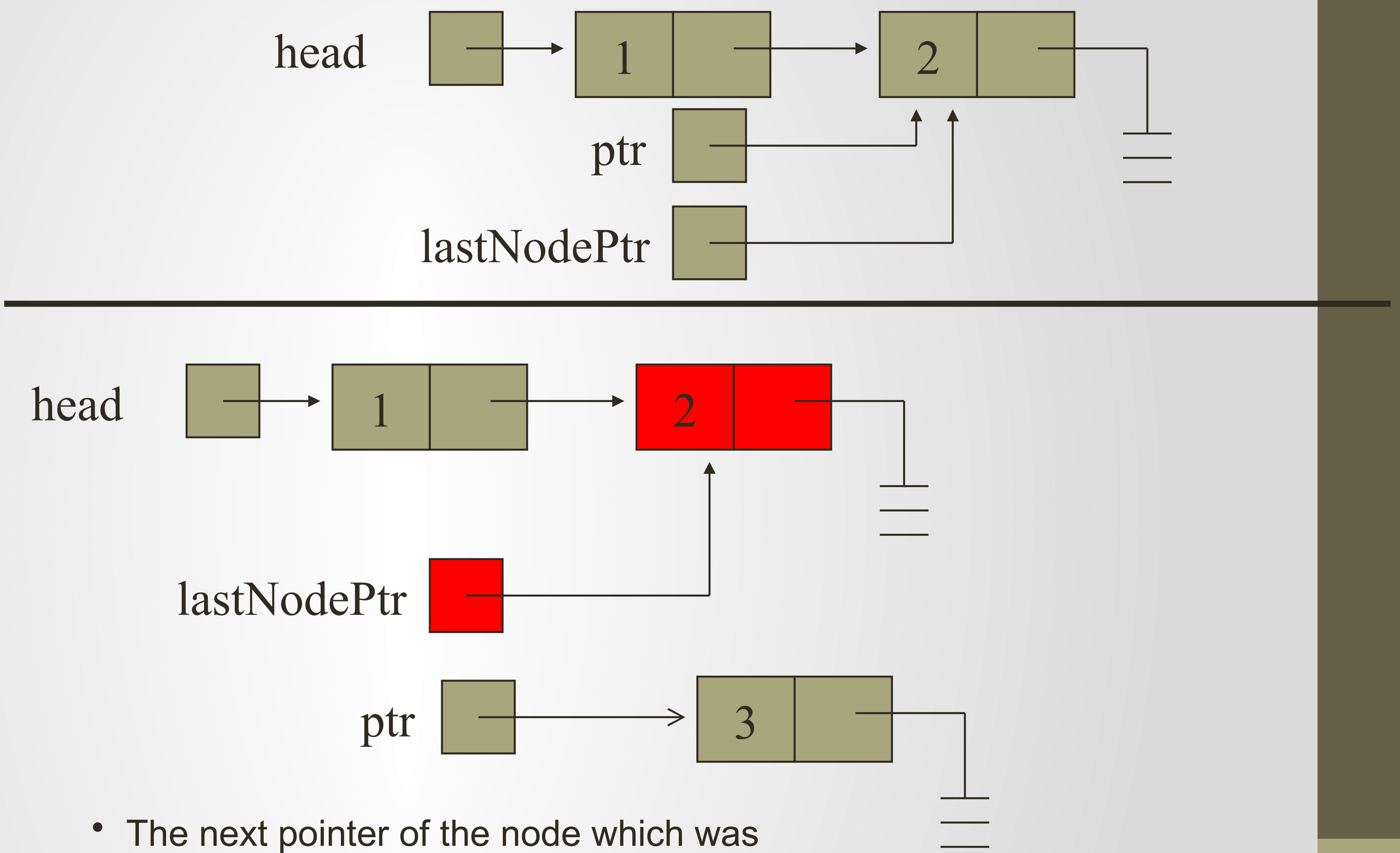


Initially

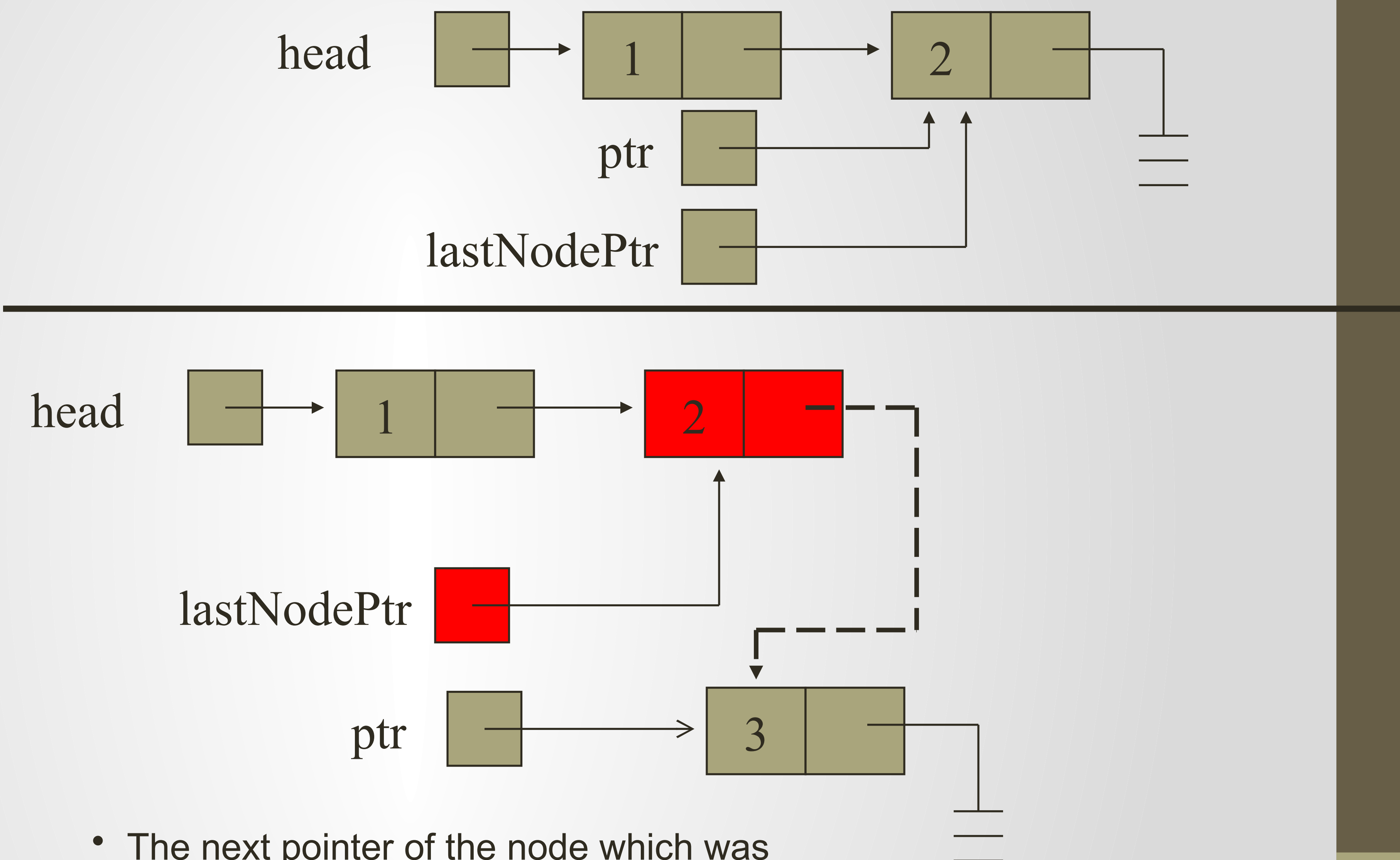




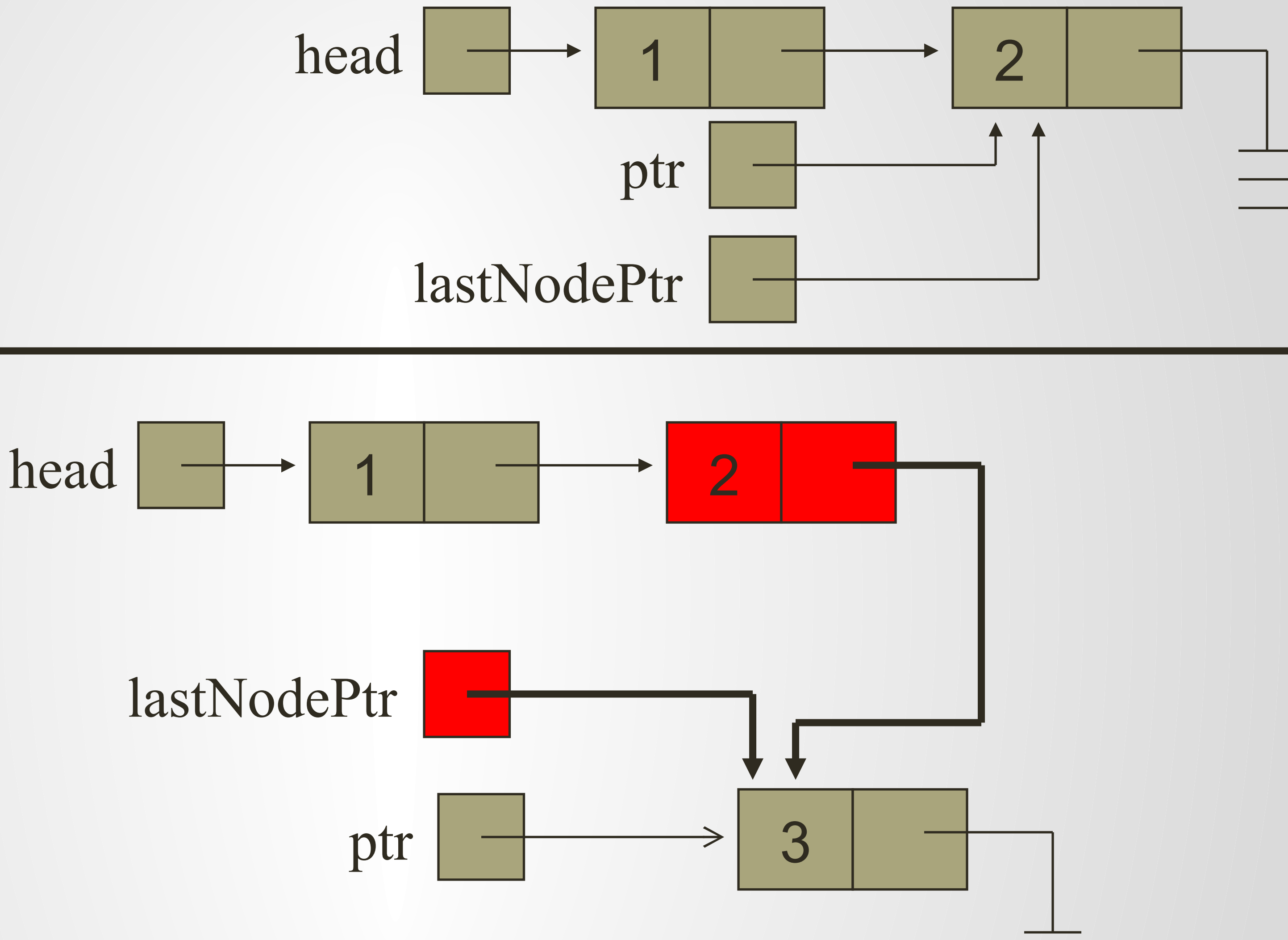
- Create a new node with data field set to 3
- Its next pointer should point to NULL



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

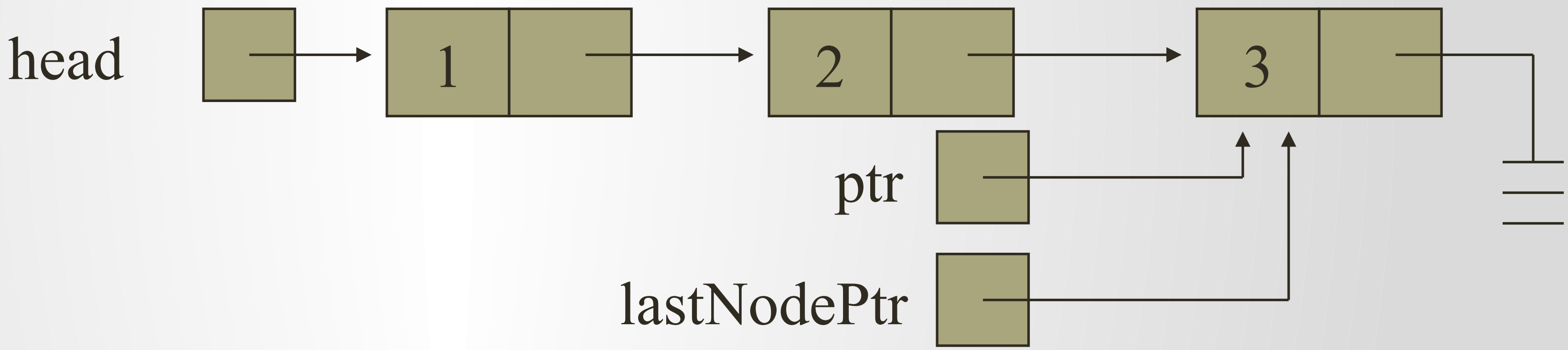


- The next pointer of the node which was previously last should now point to newly created node “lastNodePtr->next=ptr”
- LastNodePtr should now point to the newly created Node “lastNodePtr = ptr;”

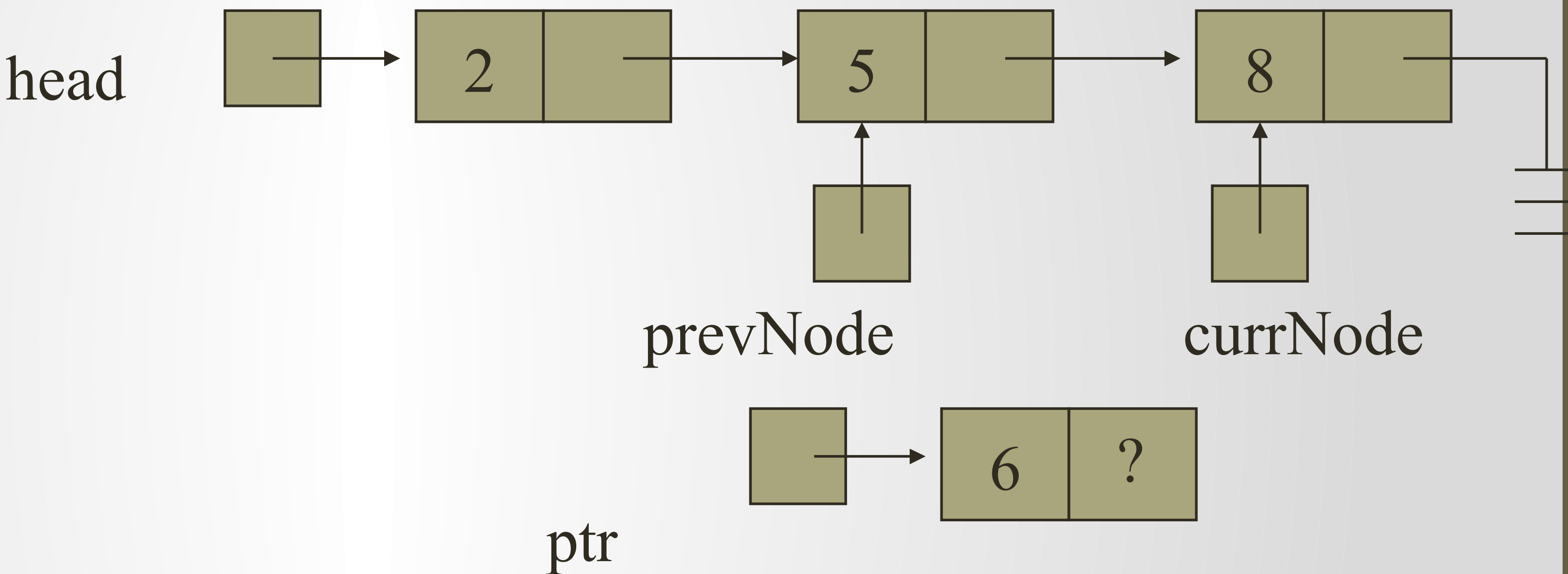


- LastNodePtr should now point to the newly created Node “lastNodePtr = ptr”;

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

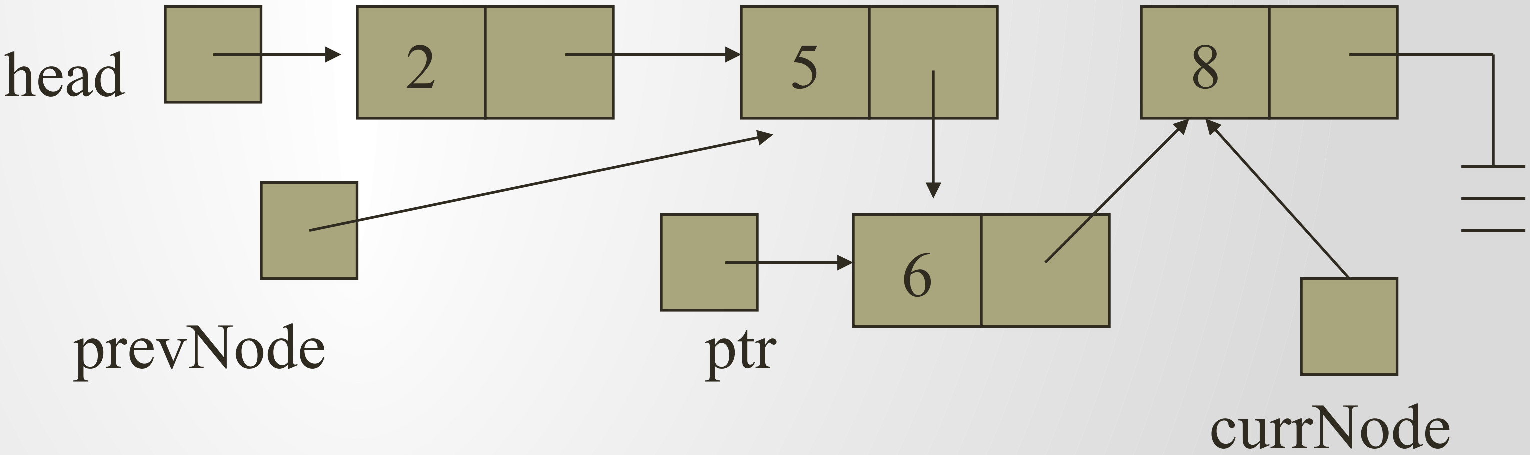
Continuing the insert

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

```
ptr - > next = currNode;
```

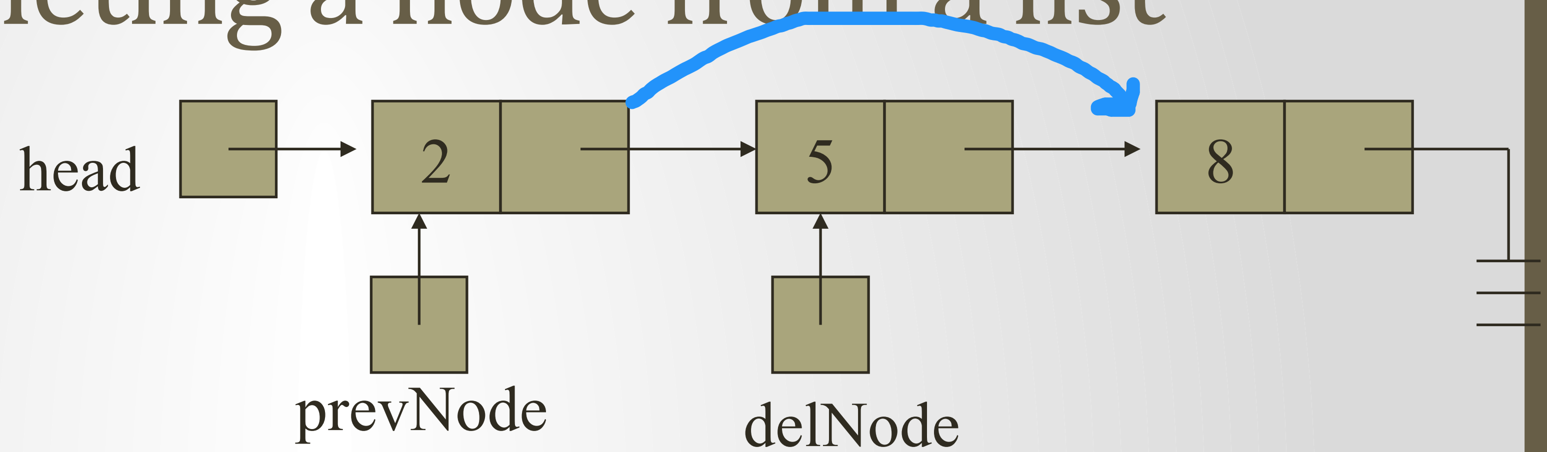
Step 4: Make previous node point to the new node:

```
prevNode - > next = ptr;
```



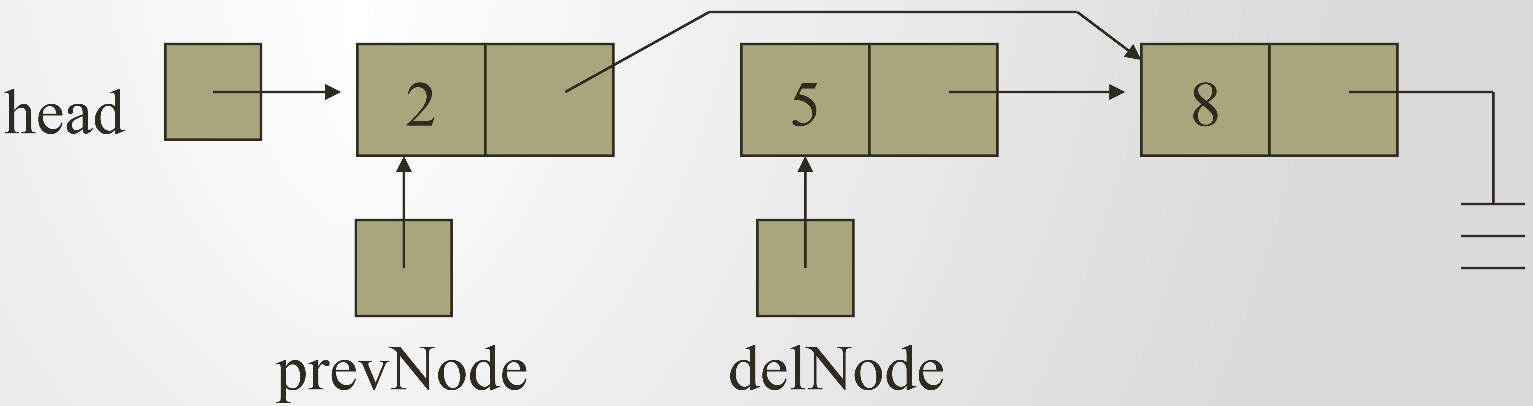
Now The new link has been added in the linked list

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.

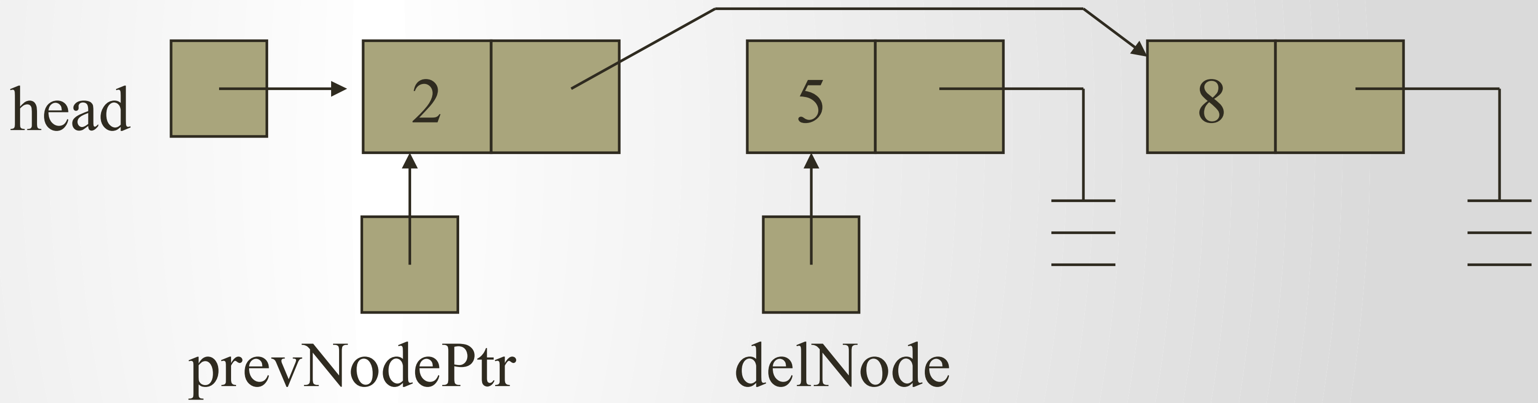
prevNode - > next = delNode - > next;



Finishing the deletion

Step 2: Remove the pointer from the deleted link.

delNode - > next = NULL;

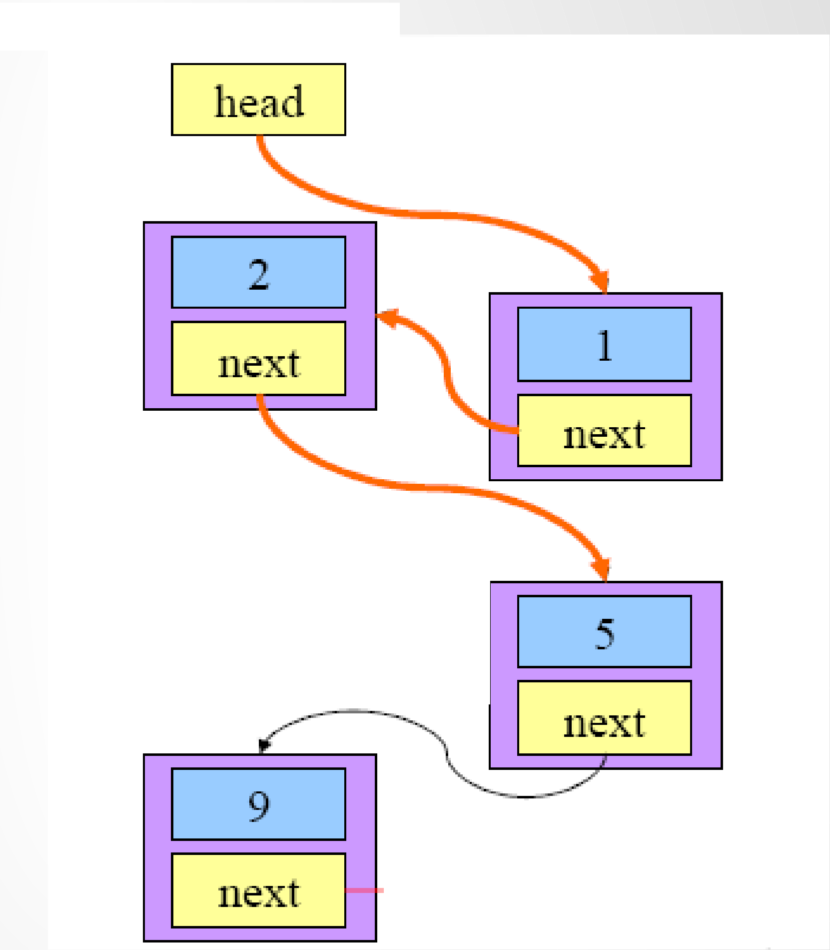


Step 3: Free up the memory used for the deleted node:

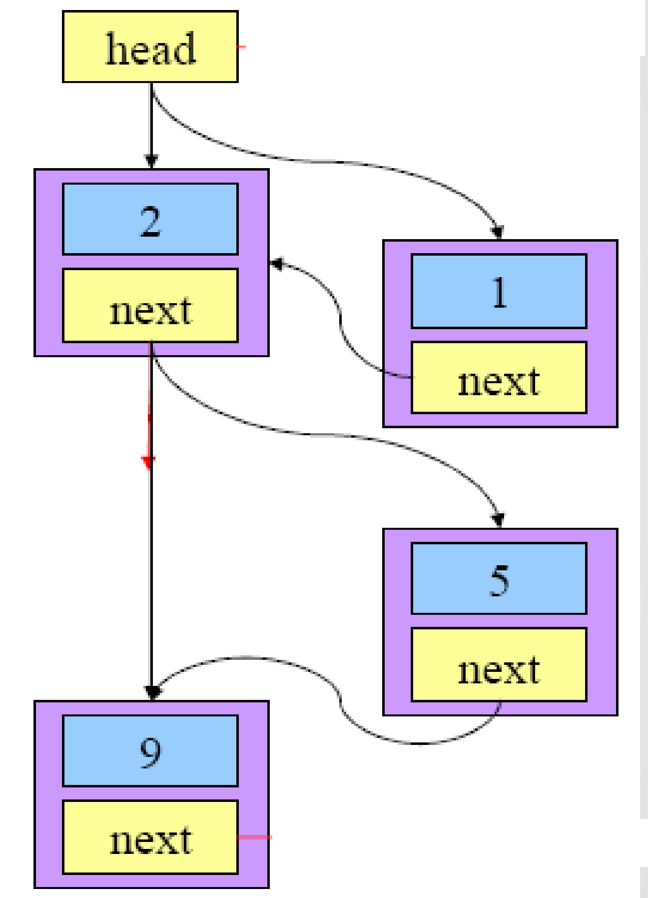
~~delete delNode;~~ free(delNode);

List Operations - Summarized

Traversing a Linked List



Insertion in a Linked List



Deletion from a Linked List