Linked List

Data Structure and Algorithms

Slides credit: Ms. Saba Anwar

Outline

- Limitations of Array
- Linked List
  - Operations
  - Variations of Linked List
- Array vs Linked List

Limitations of Arrays

Arrays are stored in contiguous memory blocks. And have advantages and disadvantages due to it.

- It is very easy to access any data element from array using index
- We need to know size of array before hand.
- We cannot resize array. Because They are **static** in size
  - We can relocate existing array to new array, but still expensive
- Contiguous block cannot be guaranteed→ insufficient blocks size
- Insertion and deletion is very expensive because it needs shifting of elements

Solution: Linked list

- A **dynamic** data structure in which each data element is linked with next element through some link. Because each element is connected/linked, it will be easy to insert and delete an element without shifting.

Dynamic Data Structure

Data structures whose size is not known and they can grow/shrink during use are created using dynamic memory allocation concept.

Each block of data is allocated dyanamically in memory using the new operator.

How Data Structure will be made?

- If we need a linear list of 5 data elements then 5 blocks are created and every block is connected to each other by storing address of next block in previous block. Now this list of block is linked.
- If we need another block, create and link with existing block
- If we need to delete, remove link
  - Do not forget to delete the dyanmically allocated memory using delete operator.

So size is actually increasing/decreasing whereas its not possible in arrays

Linked List

Linked list is a linear collection of homogenous data elements where each element is connected through a link. The links are maintained using pointers.

A single element in linked list is normally called **Node**. Every node has two parts:

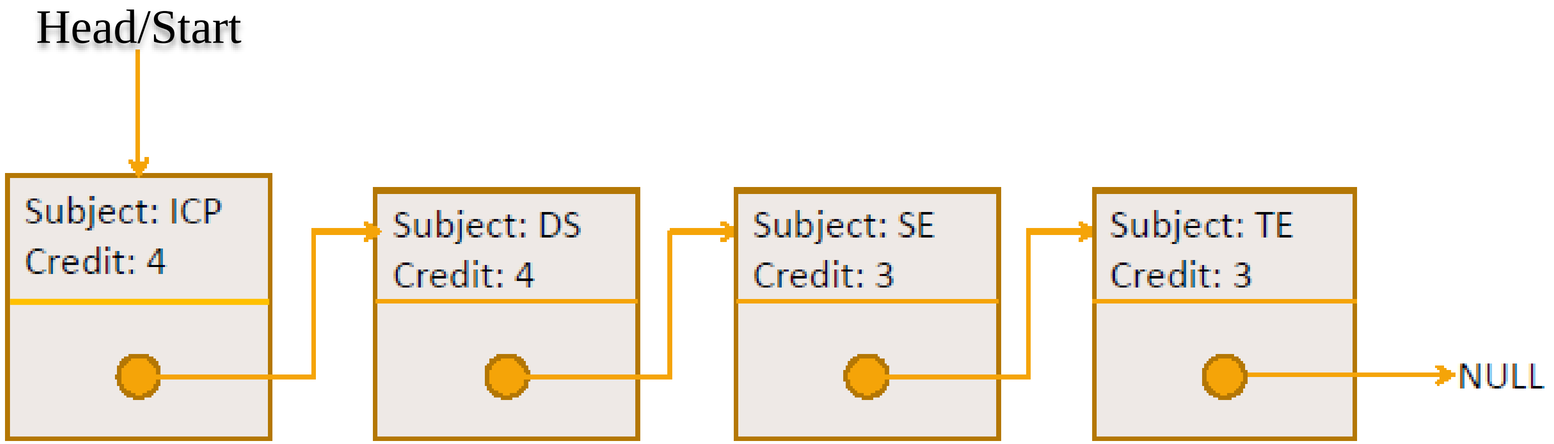
- Data: actual information
- Next Link- a reference to next node in memory

If next link is NULL, it means it is the last node of list.

In order to build a linked list we need to maintain both data and address to next node

Linked List

Linked list with 4 nodes



We always need location of the first node of list in order to process it for any purpose

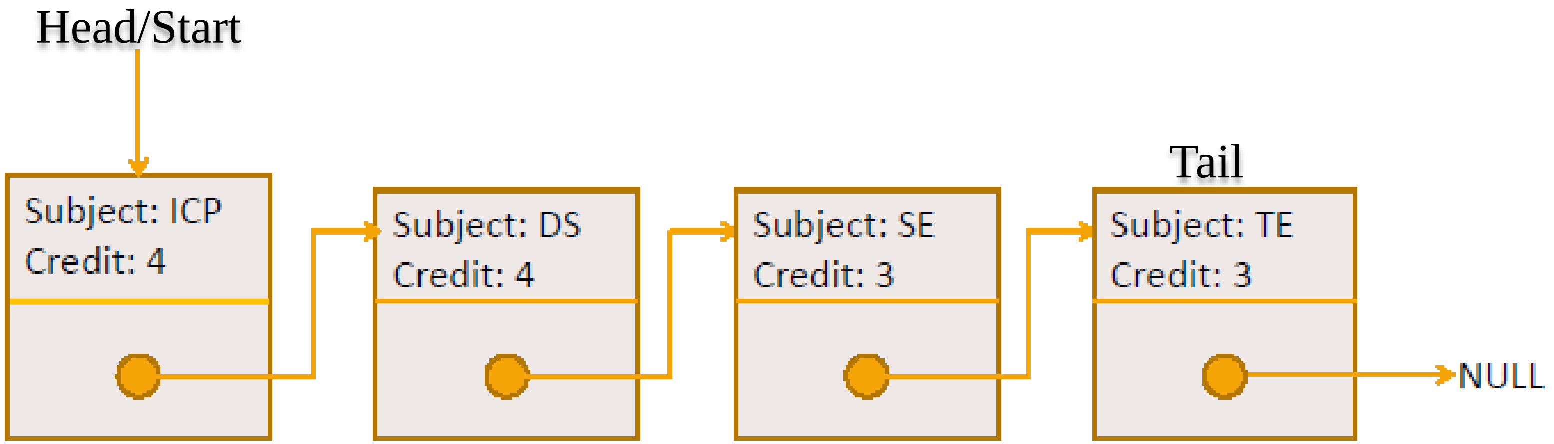
Head or Start node is reference which points to address of the first node of list.

Because all nodes are connected only through pointers, so if we have this head, any node can be accessed by traversal.

If head is NULL, it means list is empty

Linked List

Linked list with 4 nodes



We always need to know the last node of list.

- **Tail node** last node of list whose next pointer will be null.
- We can use head to traverse through list to search for tail node.
- Or just like head we can maintain a tail reference too

Node

Node can be represented using either structure or class.

Class is also used in C++, but we will focus on struct only. Such kind of struct, which has pointer referring to the same type of variable, is called self-reeferential structure.

C++

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

Node Operations:

Constructing a new node

```
Node* node=new Node
```

Accessing the data value

```
node->data
```

Accessing the next pointer

```
node->next
```

Node examples

Node can have individual data members or other objects as well.

```
class Student{
private:
        String name;
        float gpa;
        Location address;
        Student* next;
//methods
}
```

```
class Location{
public:
        int house;
        int street;
        String town;
        String city;
}
```

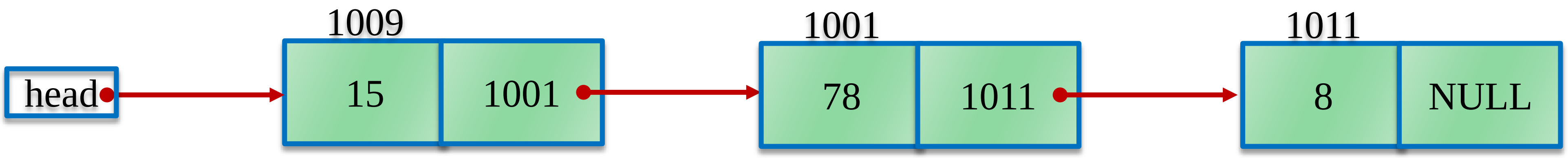
Linked List Memory Representation

Linked list has nodes and memory has cells, nodes of list are distributed in memory cells, each node is an object that is dynamically created at run time and a free memory cell is allocated to that node.

Let say head node of a linked list is located at memory cell 1009. Its data is only an integer value.

It points to list's 2nd node which is located at memory location 1001

2nd node points to 3rd node which is located at 1011.



3rd node points to NULL address, means it is end of list

Linked List Operations

- Traversal
  - Search, print, update etc.
- Insertion
- Deletion

Search

Algorithm: SEARCH(Head, V)

Input: reference to first node of list and value to be searched

Output: return node if value is found other wise null

Steps:

start

```
1.  Set ptr=Head
2.  While (ptr != NULL)
3.        if ptr.data==V
4.            return ptr
5.        end if
6.        ptr=ptr.next                 // update ptr so it can  refer to next node
7.  End While
8.  return NULL
```

end

Search

Algorithm: PRINT(Head)

Input: reference to first node of list

Output: print all nodes

Steps:

start

1. Set ptr=Head
2. While (ptr != NULL)
3. print ptr.data
4. ptr=ptr.next // update ptr so it can refer to next node
5. End While

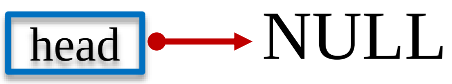
end

Insertion

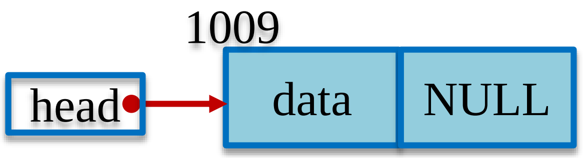
- Inserting a new node involves two things:
  - Creating a new node
  - Linking this node to its logical predecessor and successor node, so all nodes still remain linked
- There can be three scenarios to insert a new node
  - Insertion at Start
  - Insertion at End
  - Insertion at Middle

Insertion at Start

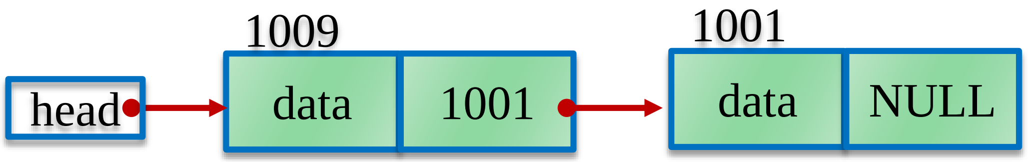
If List is Empty



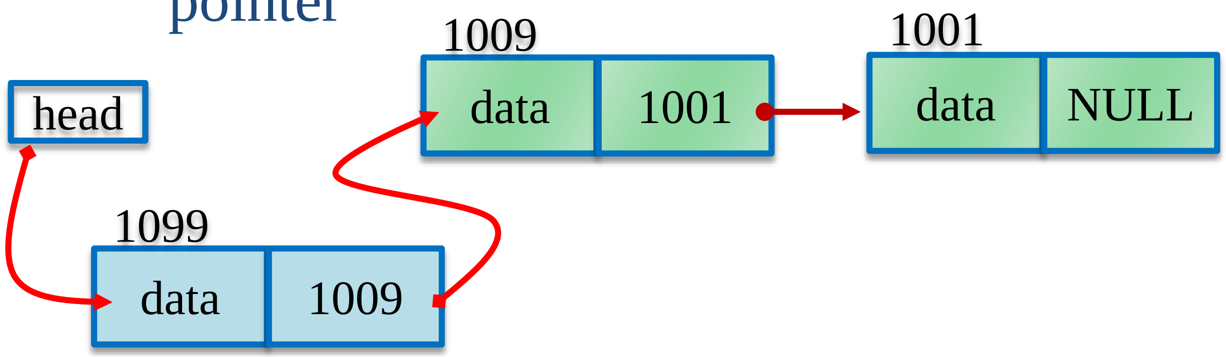
Create a Node and Update Head



List is not Empty



Create new node, and update head pointer



Insertion at Start

Algorithm: INSERT_START(Head, V)

Input: head and data for new node
Output: list with new node inserted
Steps:

```
Start
1. newestNode = new Node(V)          //create new node
2. If Head==NULL                     // list is empty
3.     Head=newestNode
4. Else                              //list is not empty
5.     newestNode.next=Head
6.     Head=newestNode
7. End If
End
```

Insertion at Start (2)

Algorithm: INSERT_START(Head, newstNode)

Input: head node and new node

Output: list with new node inserted

Steps:

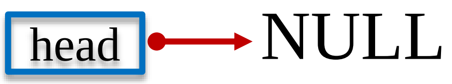
Start

1. If Head==NULL // list is empty
2. Head=newestNode
3. Else //list is not empty
4. newestNode.next=Head
5. Head=newestNode
6. End If

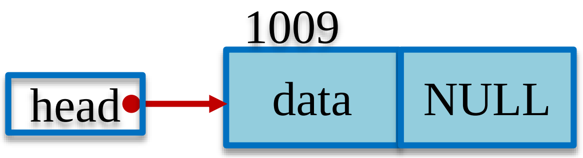
End

Insertion at End

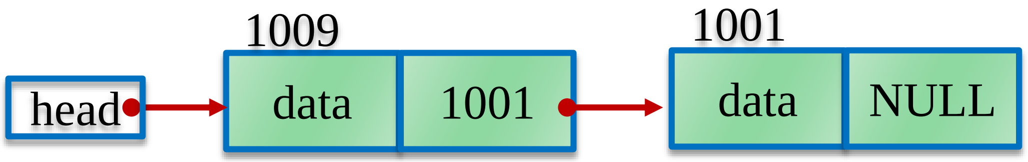
If List is Empty



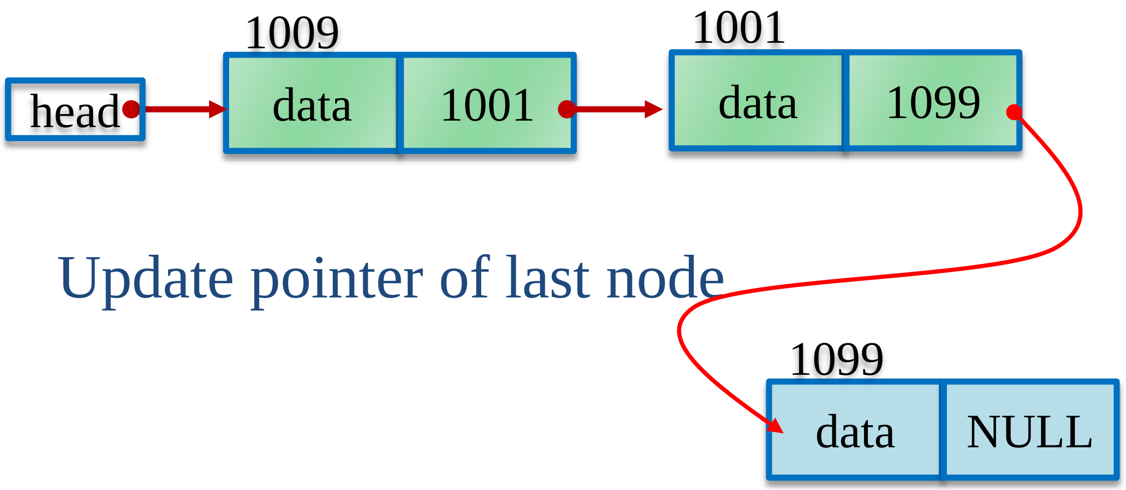
Create a Node and Update Head



Non-Empty List



Create new node



Update pointer of last node

Insertion at End

If list is not empty then we need to traverse the list to find the last node in order to link the newly crated node in existing list.

Algorithm: INSERT_END(Head, V)

Input: head node and data to be inserted

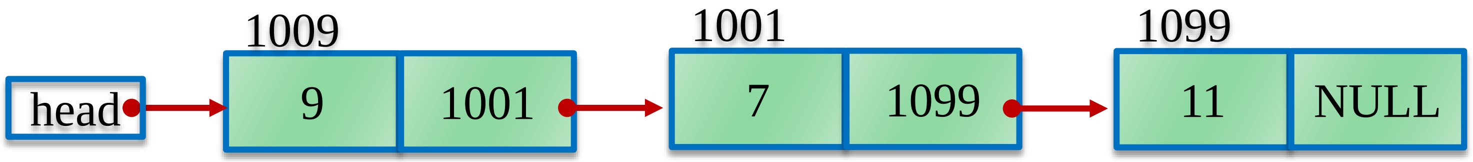
Output: list with new node inserted

Steps:

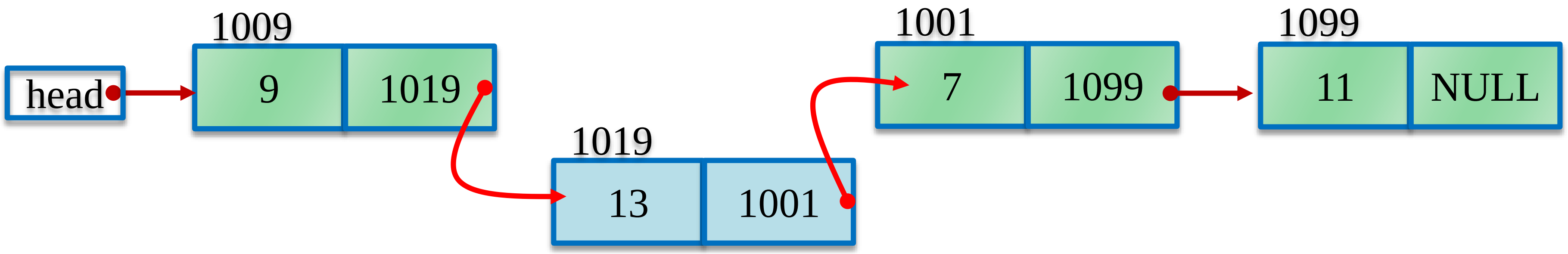
```
Start:
newestNode = new Node(V)              //create new node
If Head==NULL                         // list is empty
    Head=newestNode
Else                                  //list is not empty. Search last node
   Set ptr=Head
    While(ptr.next!= NULL)
        ptr=ptr.next
     End While                        // loop will terminate when last node is found
     ptr.next=newestNode              // update the next pointer of ptr(last node)
End If
End
```

Insertion at Middle

Let say we want to insert 13 in following list, at location 2.

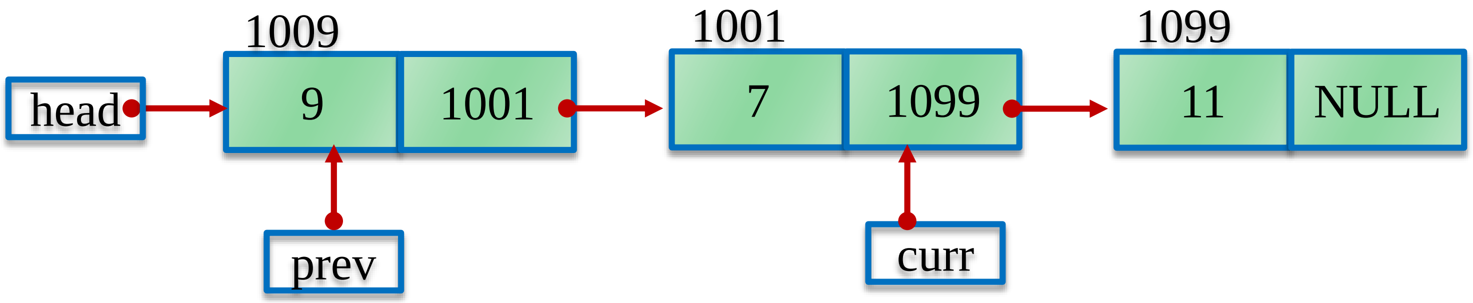


In this case, first we need to locate the 2nd node. That is node with data=7. Now this will become 3rd node and new node will be inserted before this node.

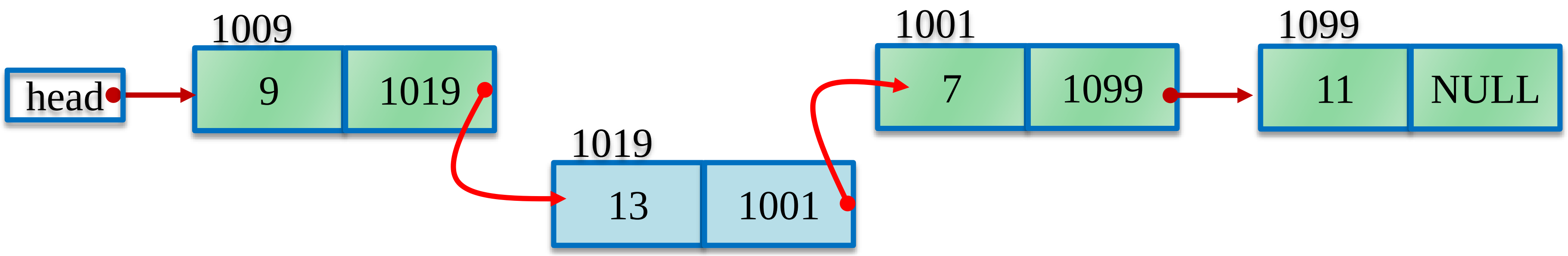


Insertion at Middle

In order to insert somewhere middle of linked list, we need to maintain two pointers, current and previous.



So when new node is inserted, we can easily change next links of new node and previous node.



Insert at Middle

Algorithm: INSERT_MIDDLE(Head, loc, newestNode)

Input: head node, new node and loc of new node

Output: list with new node inserted

Steps:

Start

```
1.  If Head==NULL        // list is empty
2.      Head=newestNode
3.  Else If Loc==1       // update head
4.      newestNode.next=Head
5.      Head=newestNode
6.  Else
6.    Set Curr=Head, Prev= NULL
7.    nodeCount=0
8.  While (Curr != NULL)
9.        nodeCount=nodeCount+1
10.       If nodeCount ==Loc
11.         newestNode.next=Curr
12.         prev.next=newestNode
13.         Break
14.       End If
15.         Prev=Curr
16.         Curr=Curr.next
17.     End While
18. End If
```

End

Insert at Middle (2)

Algorithm: INSERT_MIDDLE(Head, loc, newestNode)

Input: head node, new node and loc of new node

Output: list with new node inserted

Steps:

Start

```
1.  If Head==NULL        // list is empty
2.      Head=newestNode
3.  Else
4.     Set Curr=Head, Prev= NULL,  nodeCount=0
5.  While (Curr != NULL)
6.           nodeCount=nodeCount+1
7.          If nodeCount ==Loc
8.             newestNode.next=Curr
9.      if (prev!=NULL)
10.                  prev.next=newestNode
11.              else
1.                  newestNode.next=Head
2.                  Head=newestNode
3.                End if
8.              Break
9.           End If
10.             Prev=Curr
11.              Curr=Curr.next
12.        End While
13.  End If
```

End

Variations of Insert

How the working of algorithm will be changed:

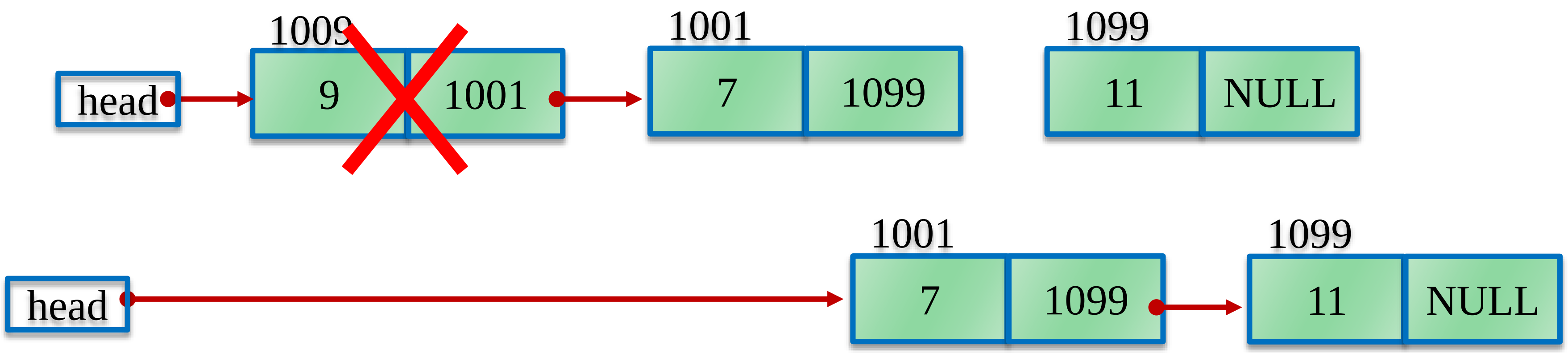
- Insertion after a given data value/location
- Insertion in a sorted linked list
  - List is already sorted and after insertion it must remain sorted
  - No sorting algorithm required

Deletion

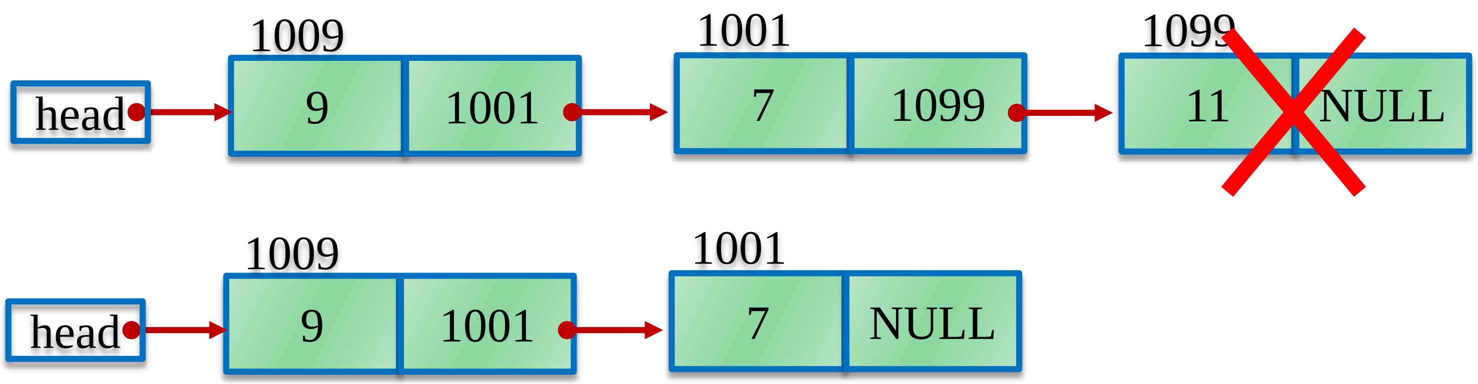
- Deleting a new node involves two things:
  - Unlinking the node in a way that its logical predecessor gets connected to next node of list to maintain linking
- There can be three scenarios to delete node
  - Deletion from Start
  - Deletion from End
  - Deletion from Middle

Deletion

Deletion From Start



Deletion From End



DELETE

Algorithm: DELETE_START(Head)

Input: reference to first node

Output: new list with first node deleted

Steps:

Start

1. If Head!=NULL// list is not empty
2. Head=Head.next
3. End If

End

Algorithm: DELETE_END(Head)

Input: reference to first node

Output: new list with lat node deleted

Steps:

Start:

1. If Head!=NULL// list is not empty
2. If Head.next==NULL// there is only one node
3. Head=NULL
4. Else
5. Set Curr=Head, Prev=NULL
6. While (Curr.next!=NULL)
7. Prev=Curr
8. Curr=Curr.next
9. End While
10. Prev.next=NULL
11. End If
12. End If

End

Deletion

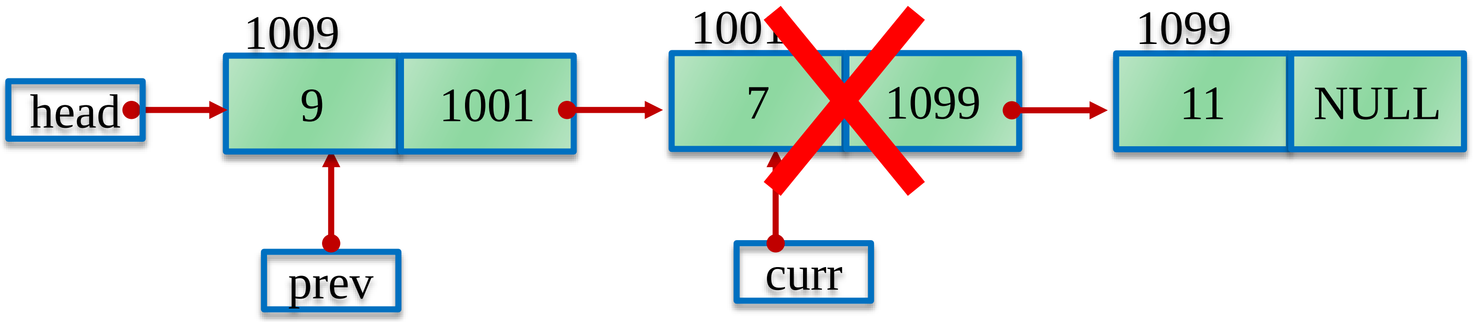
Deletion at Location

This case involves searching a specific node to delete

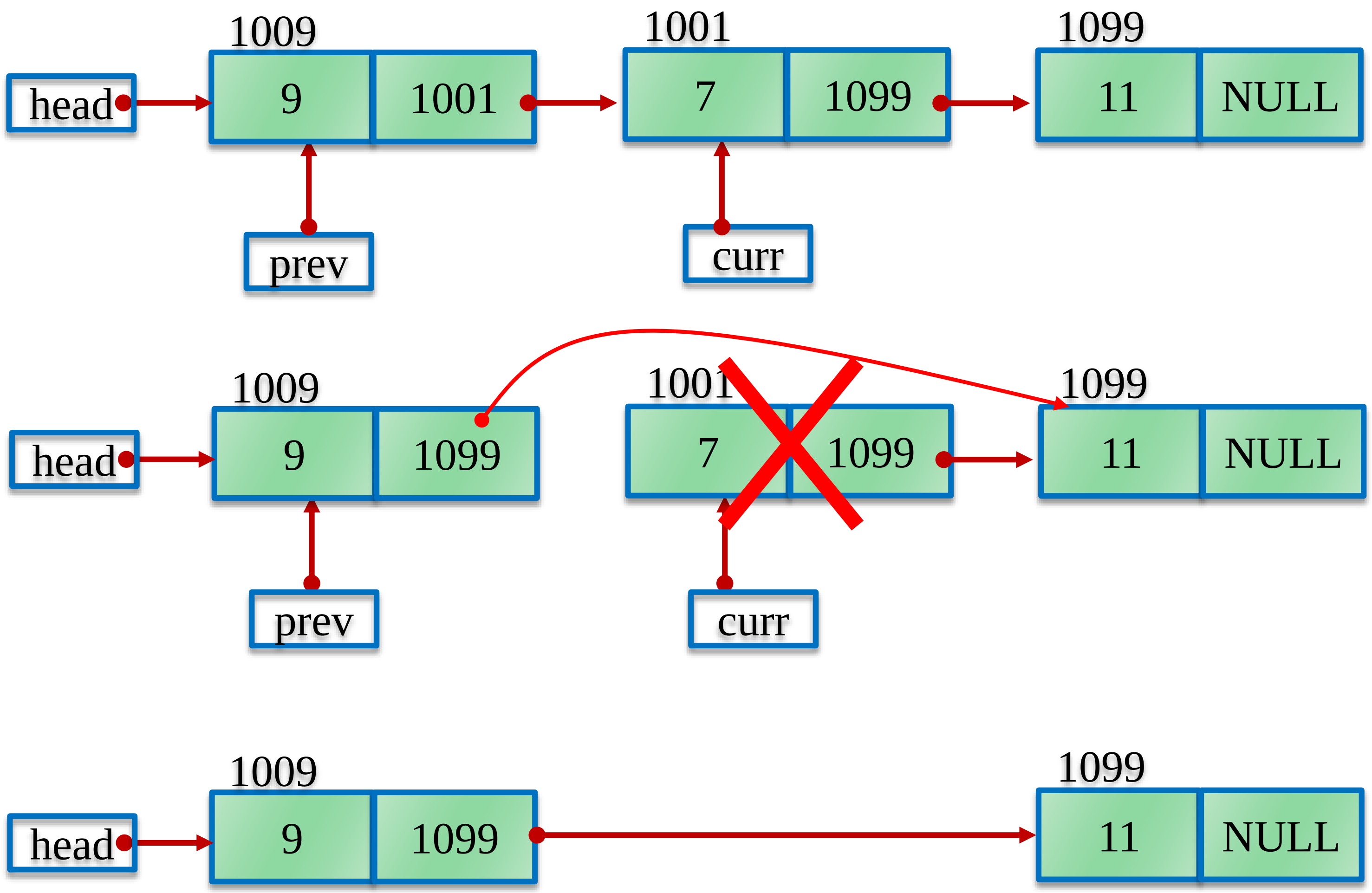
We also need to find current and previous node pointers in order to maintain links.

Current will be our required node. Let say we need to delete 2nd from following list

We need to search 2nd in list, and then need to find its previous node also, so before deletion we can link previous node to next node of 2^{nd} node



Deletion


1009
1001
1099
head
9
1001
7
1099
11
NULL
prev
curr
1009
1001
1099
head
9
1099
7
1099
11
NULL
prev
curr
1009
1099
head
9
1099
11
NULL

DELETE

Algorithm: DELETE_LOCATION(Head, Loc)

Input: reference to first node and loc

Output: new list with node deleted

Steps:

Start:

```
1.  Set nodeCount=0
2.  If Head!=NULL   // list is not empty
3.      If Loc==1       // this is head node
4.          temp = Head
5.          Head=Head.next
6.          delete temp
7.      Else
8.          Set Curr=Head, Prev=NULL
```

```
1.          While (Curr!=NULL)
2.              nodeCount=nodeCount+1
3.              If (nodeCount==Loc)
4.                  Prev.next=Curr.next
5.                  break
6.              End If
7.              Prev=Curr
8.              Curr=Curr.next
9.          End While
10.     End If
11. End If
```

Variations of DELETE

How DELETE_LOCATION will change?

- If a data value is given instead of location.?
- If a node is given instead of location?

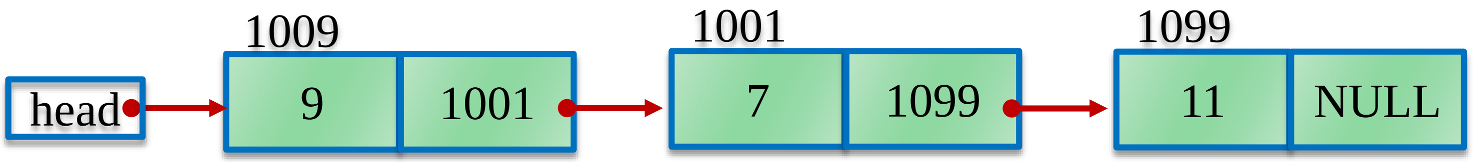
Types of Linked List

Depending upon how links are maintained, there can be variationsof linked list:

- Singly Linked List
  - We have discussed it already
- Doubly Linked List
- Circular Linked List

Singly Linked List

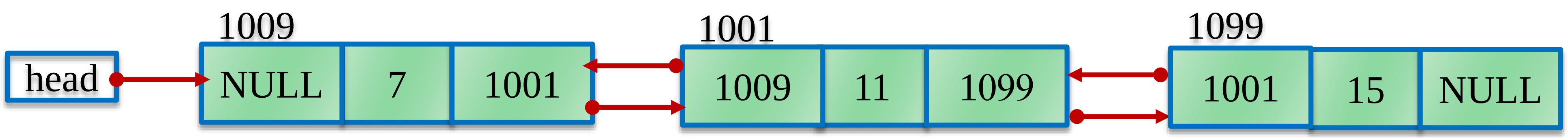
Every node contains only one next link which points to next node in list



Last nodes points to NULL.

Doubly Linked List

Every node contains two links, **next** which points to next node and **previous** which points to previous node in list



Note that prev and next links hold address of nodes. So, prev link of node located at 1001 points to 1009 and next link points to 1099.

Previous link of first node is NULL

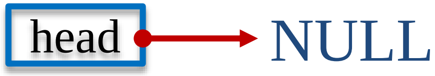
Next link of last node is NULL

Doubly linked list can be traversed from start to end and from end to start.

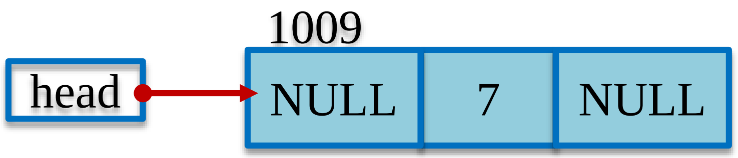
If we have tail node

Insertion at Start

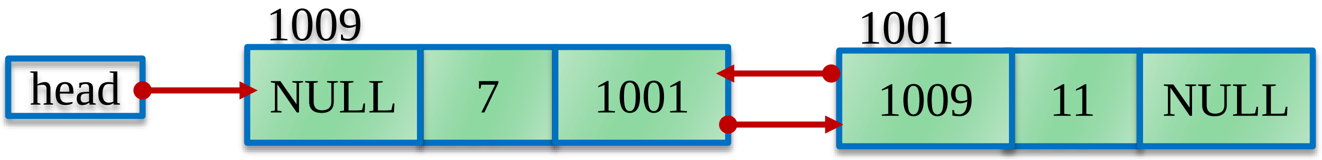
If List is Empty?



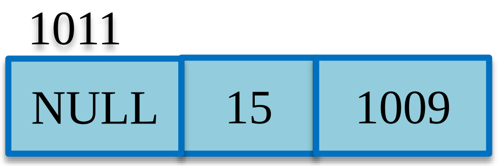
Create new node, and update head



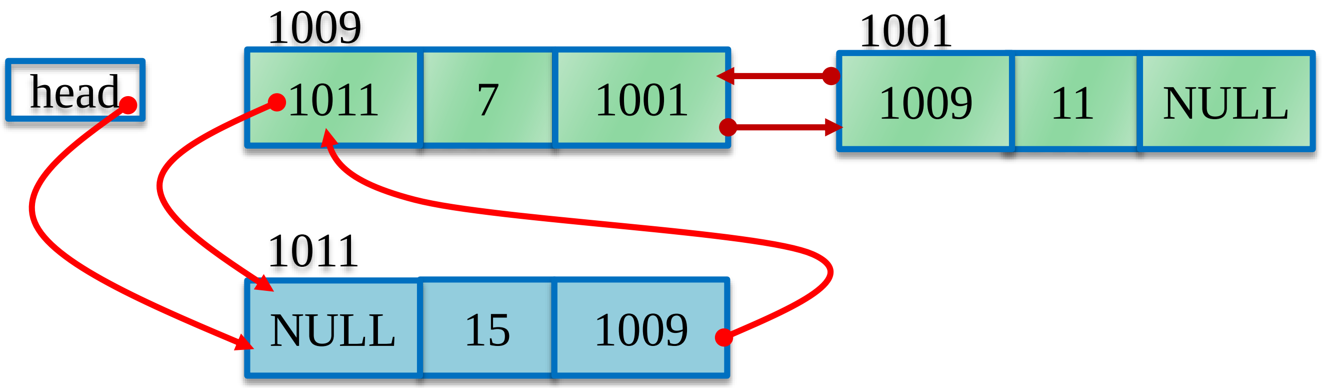
If List is non-Empty?



Create new node



Change links



Insertion at Start

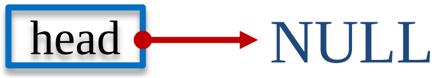
Algorithm: INSERT_START(Head, newstNode)

Input: head node and new node
Output: list with new node inserted
Steps:

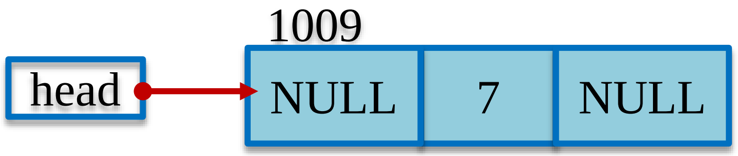
```
Start
1.  If Head==NULL                    // list is empty
2.      Head=newestNode
3.  Else                             //list is not empty
4.      newestNode.next=Head
5.      Head.prev=newestNode
6.      Head=newestNode
7.  End If
End
```

Insertion at End

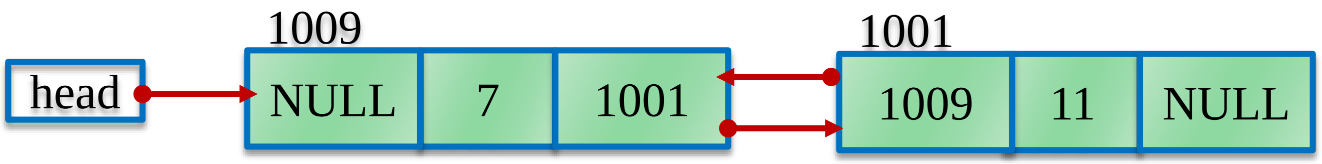
If List is Empty?



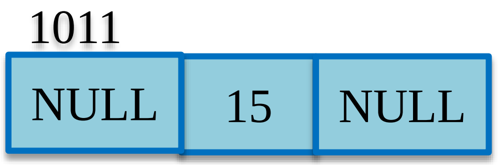
Create new node, and update head



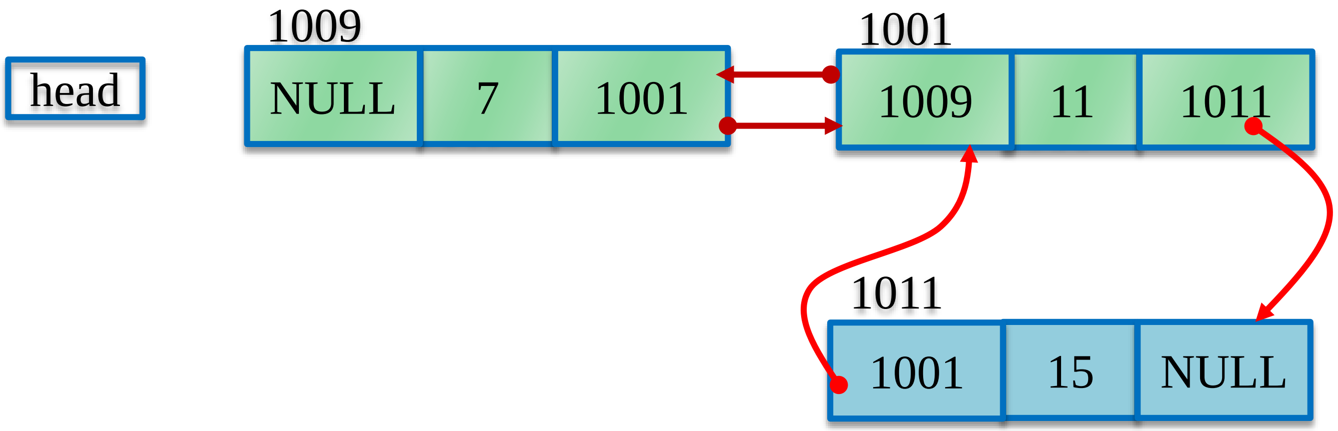
If List is non-Empty?



Create new node



Change links



Insertion at End

If list is not empty then we need to traverse the list to find the last node in order to link the newly crated node in existing list.

Algorithm: INSERT_END(Head, V)

Input: head node and data to be inserted

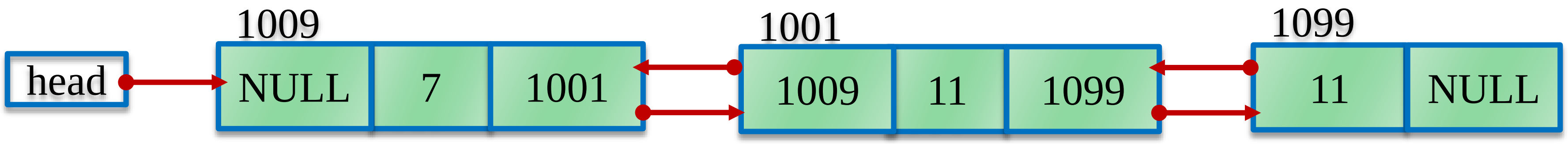
Output: list with new node inserted

Steps:

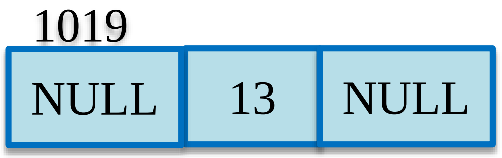
```
Start:
1.  newestNode = new Node(V)            //create new node
2.  If Head==NULL                       // list is empty
3.      Head=newestNode
4.  Else                                //list is not empty. Search last node
5.      Set ptr=Head
6.      While(ptr.next!= NULL)
7.          ptr=ptr.next
8.      End While                       // loop will terminate when last node is found
9.      ptr.next=newestNode             // update the next pointer of ptr(last node)
10.     newestNode.prev=ptr
11. End If
End
```

Insertion at Middle

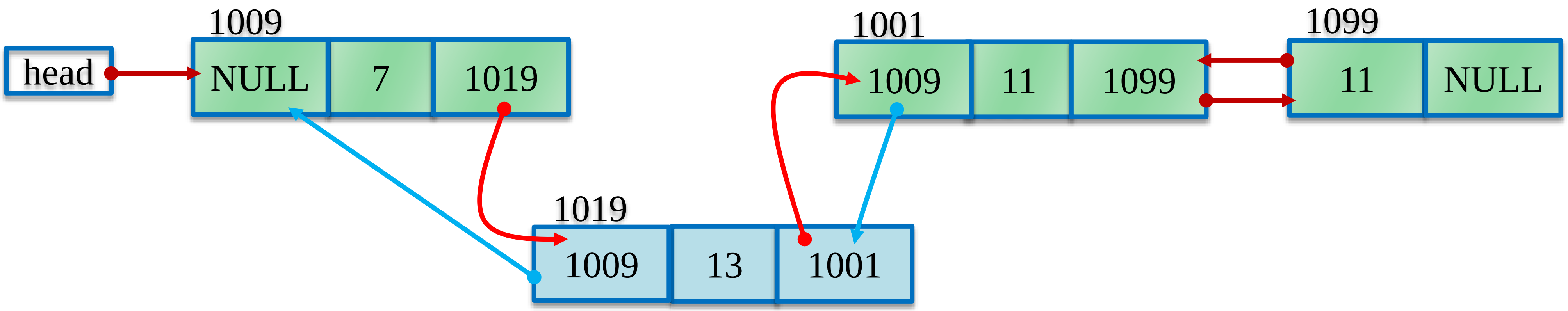
Let say we want to insert 13 in following list, at location 2.



New Node

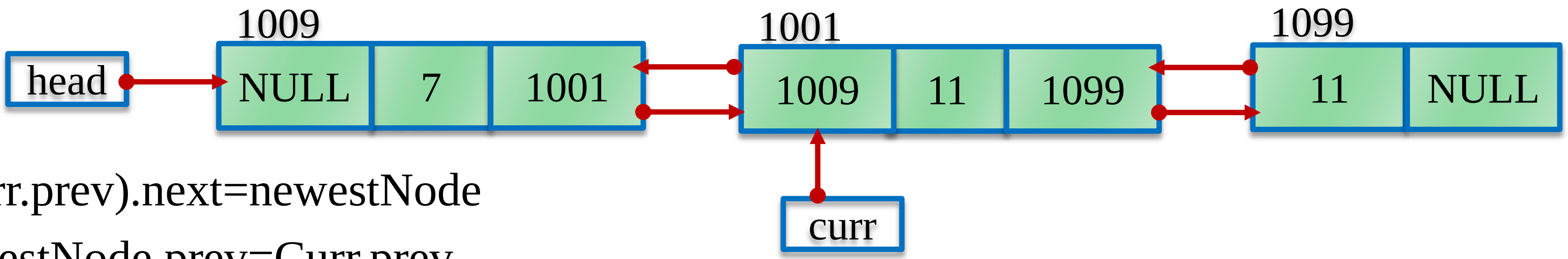


Linking

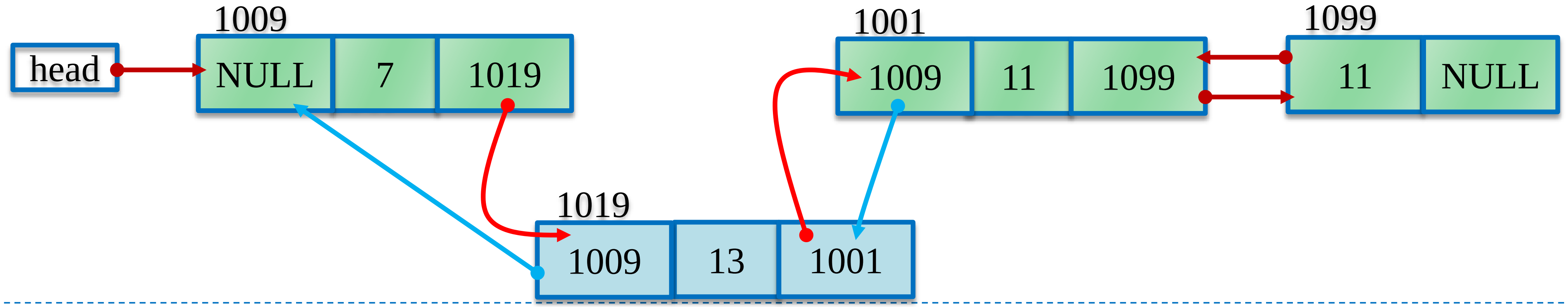


Insertion at Middle

In case of doubly linked list, we only need to keep one pointer that points to current node. Carefully read the statements. One for each link.




```
(Curr.prev).next=newestNode
newestNode.prev=Curr.prev
newestNode.next=Curr
Curr.prev=newestNode
```



Insert at Middle

Algorithm: INSERT_MIDDLE(Head, loc, newestNode)

Input: head node, new node and loc of new node

Output: list with new node inserted

Steps:

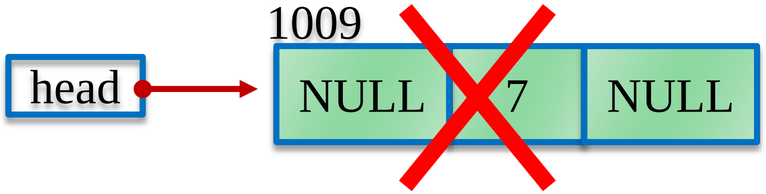
Start

```
1.   If Head==NULL        // list is empty
2.       Head=newestNode
3.   Else If Loc==1       // insertion at start
4.          newestNode.next=Head
5.          Head.prev=newestNode
6.          Head=newestNode
7.   Else
6.     Set Curr=Head
7.
8.   Set nodeCount=0
9.   While (Curr != NULL)
10.          nodeCount=nodeCount+1
11.          If nodeCount ==Loc
12.              (Curr.prev).next=newestNode
13.               newestNode.prev=Curr.prev
14.               newestNode.next=Curr
15.               Curr.prev=newestNode
16.              Break
17.          End If
18.                 Curr=Curr.next
19.        End While
20.  End If
```

End

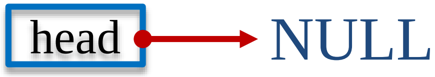
Deletion at Start

If this is the last node?

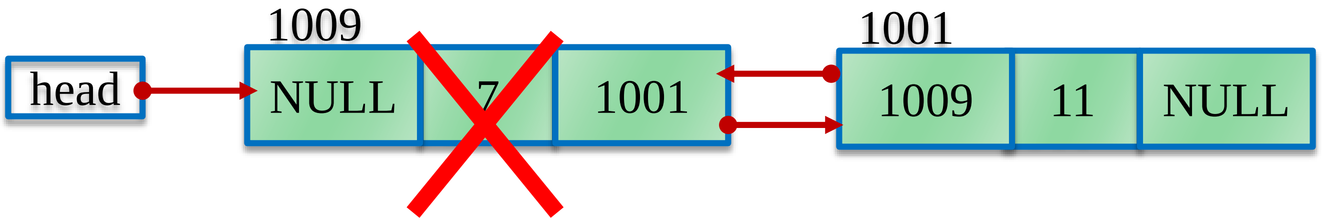


Update head

Delete node



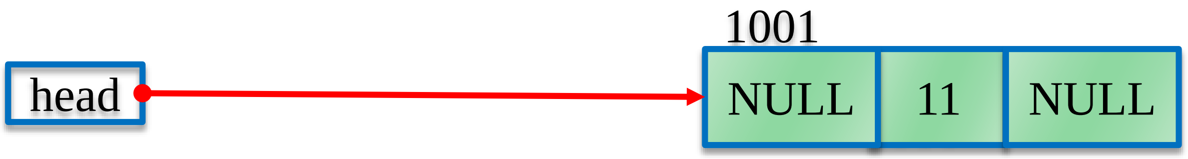
Else?



Update head

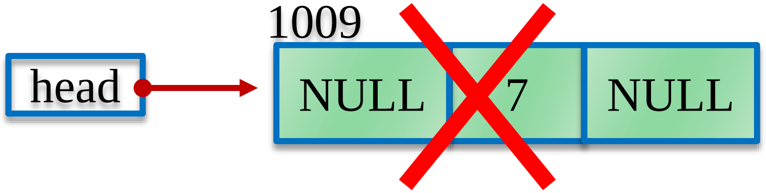
Update prev link of next node

Delete node



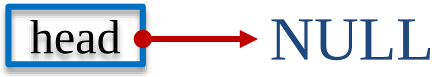
Deletion at End

If this is the last node?

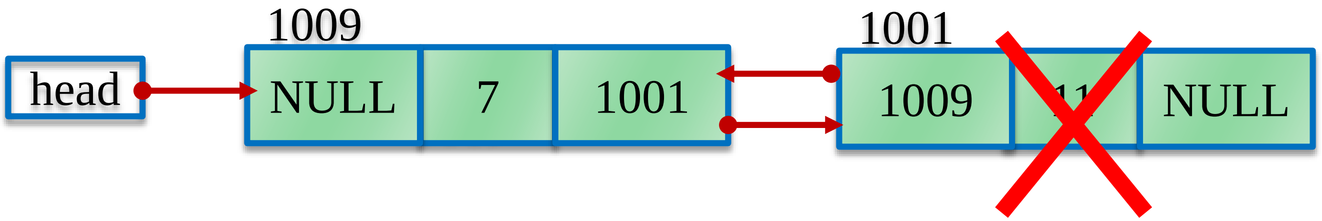


Update head

Delete node

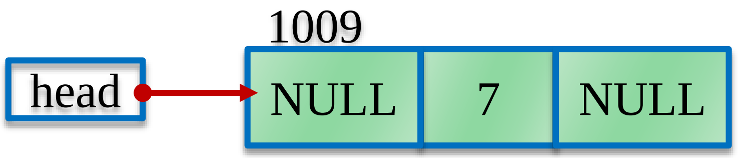


Else?



Update next link of prev node

Delete node



DELETE

Algorithm: DELETE_START(Head)

Input: reference to first node

Output: new list with node deleted

Steps:

Start

1. If Head!=NULL// list is not empty
2. Head=Head.next
3. Head.prev=NULL
4. End If

End

Algorithm: DELETE_END(Head)

Input: reference to first node

Output: new list with node deleted

Steps:

Start:

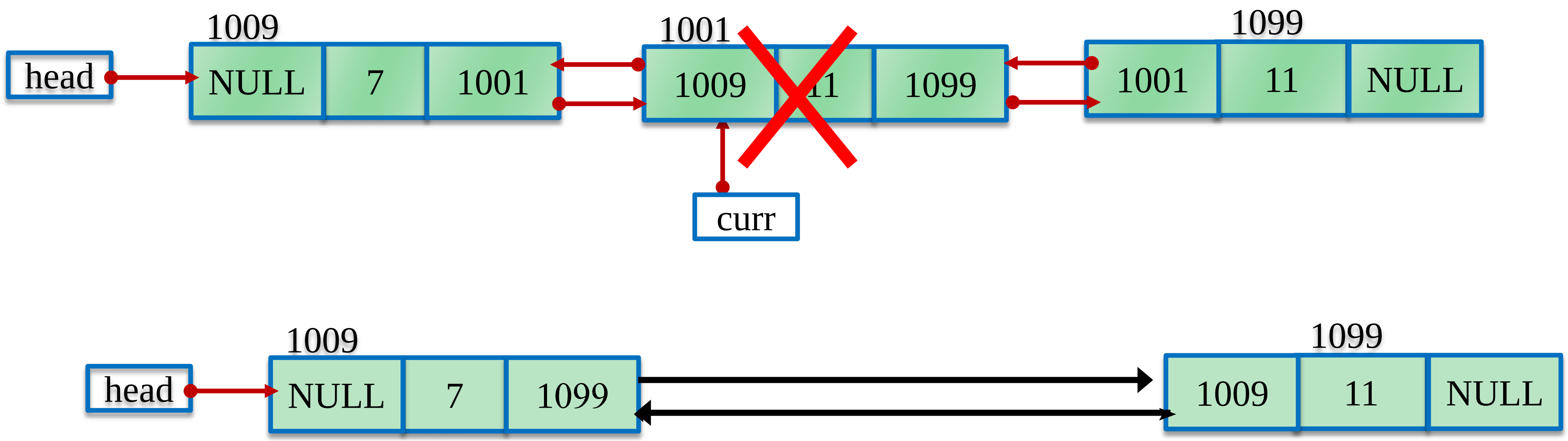
1. If Head!=NULL// list is not empty
2. If Head.next==NULL// there is only one node
3. Head=NULL
4. Else
5. Set Curr=Head
6. While (Curr.next!=NULL)
7. Curr=Curr.next
8. End While
9. (Curr.prev).next=NULL
10. Curr.prev=NULL
11. End If
12. End If

End

//if you are maintaining tail reference, then it must be updated during deletion

Deletion at Location

Let say we need to delete 2nd node.



DELETE

Algorithm: DELETE_LOCATION(Head, Loc)

Input: reference to first node and loc

Output: new list with node deleted

Steps:

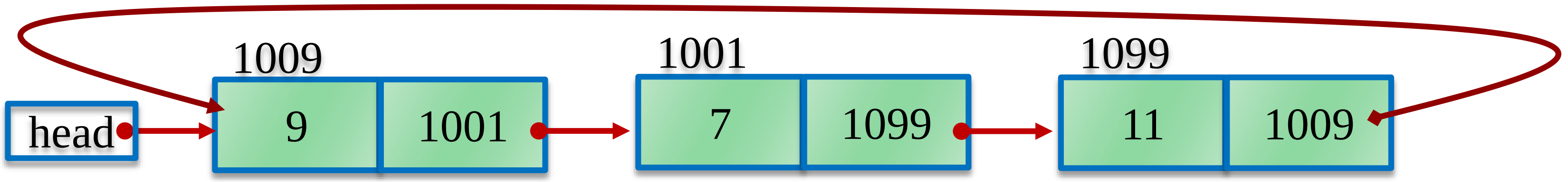
Start:

```
1.  Set nodeCount=0
2.  If Head!=NULL        // list is not empty
3.      If Loc==1        // delete at start
4.          Head=Head.next
5.          Head.prev=NULL
6.      Else
7.          Set Curr=Head
```

```
1.  While (Curr!=NULL)
2.          nodeCount=nodeCount+1
3.          If (nodeCount==Loc)
4.              (Curr.prev).next=Curr.next
5.              (Curr.next).prev=Curr.prev
6.              break
7.          End If
8.          Curr=Curr.next
9.      End While
10.   End If
11. End If
```

Circular Linked List-Single

Every node contains only one next link which points to next node in list.



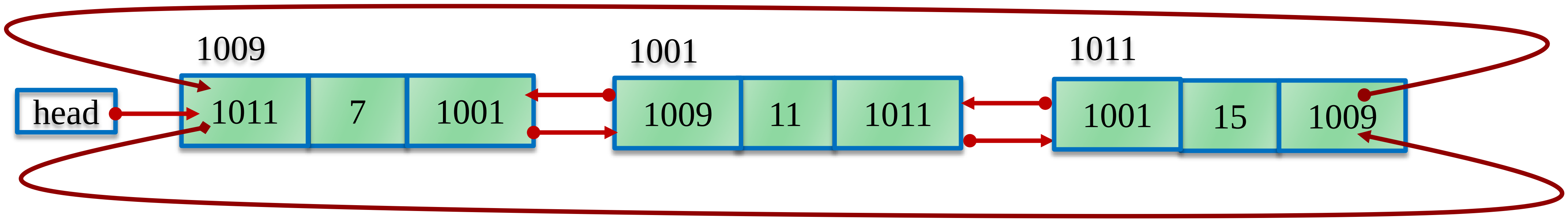
Last nodes points to First node of list

What change needs to be done in singly linked list algorithms?

How to know that node is last node in list?

Circular Linked List-Double

Doubly linked list, with last node pointing to first node and first node pointing to last node.



Previous link of first node is Last node

Next link of last node is first node

Circular Linked List

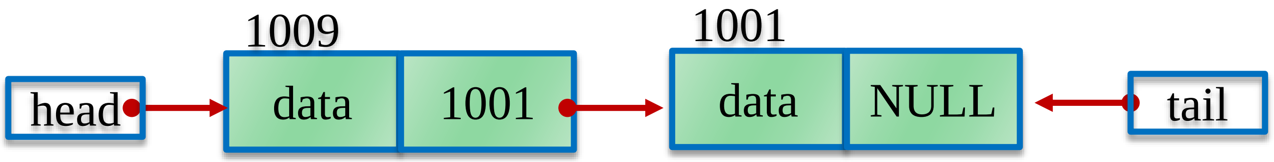
What change will be required in following algorithms of both single and double linked list:

- Insert
- Delete
- Search

When loop will terminate?

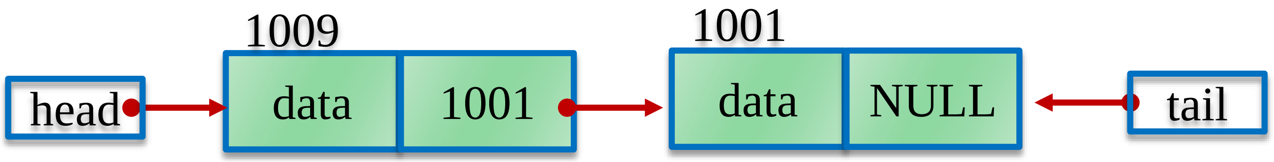
Tail Node

- Whenever we do insertion/deletion to end of list, we need to search for last node, it involves loop
- Can we avoid this loop?
  - By maintaining a reference to last node just like we do for first node.
  - It will save time
- How?



Tail Node

- Whenever we do insertion/deletion to end of list, we need to search for last node, it involves loop
- Can we avoid this loop?
  - By maintaining a reference to last node just like we do for first node.
  - Will save time?
    - Let Say Tail points to last node
  - Insertion at End needs last node
    - Tail.next=newestNode;
    - Tail=NewestNode
    - If double linked list, then set prev link of new node
  - Deletion at End needs 2^{nd} last node
    - In single linked list: No benefit, we always need to search for 2^{nd} last node
    - In Double: we can go to previous node of double linked list.



Modified Insert_End –Single Linked List

- **Algorithm: INSERT_END(Head, Tail, V)**
  - **Input: head node and data to be inserted**
  - **Output: list with new node inserted**
  - **Steps**:

Start:

1. newestNode = new Node(V) //create new node
2. If Head==NULL // list is empty
3. Head=newestNode
4. Tail=newestNode
5. Else //list is not empty. Search last node
6. Tail.next=newestNode // update the next pointer of ptr(last node)
7. Tail=newestNode;
8. End If

End

Array vs. Linked List

- Memory allocation
  - Static vs. dynamic
  - Contiguous vs linked
- Space utilization
  - Array is fixed whereas linked list can grow/shrink
  - Single node vs single cell

Applications of Linked List

Where size is not fixed, and no random access.

Few example:

- Other data structures
  - Stack, queue, trees, skip list, graphs
- Browser's back button
  - To go to previous URLs
- Card Game
  - Deck of cards, no random access

Practice Problems

- Sorting linked list
- Remove duplicates
  - Sorted vs unsorted
- Move Node
  - Given two lists, remove node from one list and push it to start of other list
- Sorted Merge
  - Merge two sorted lists into a new sorted list
- Sorted Insert
  - List will remain sorted after insertion
- Reversing list
  - By rearranging nodes
- Copying
  - Shallow vs. deep copy
- Concatenating
  - Append one list to other list’s end