2 : Arrays and Linked Lists

IT 3206– Data Structures and Algorithms

Level II - Semester 3

Overview

- This section will illustrate how arrays are defined, declared, initialized, and accessed.
- It will also discuss the different operations that can be performed on array elements.
- Further this section will discuss about different types of linked lists and the operations that can be performed on these lists.

Intended Learning Outcomes

- At the end of this lesson, you will be able to;
  - Explain the use of Arrays and Linked Lists.
  - Compare the difference between Arrays and Linked Lists
  - Apply Arrays and Linked Lists for problem solving

List of subtopics

2.1 Introduction to Arrays
- 2.1.1 One dimensional arrays
- 2.1.2 Multi dimensional arrays
- 2.1.3 Basic operations on arrays

2.2 Comparison of Arrays and Linked Lists

2.3 Singly Linked Lists

2.4 Doubly Linked Lists

2.5 Circular Linked Lists

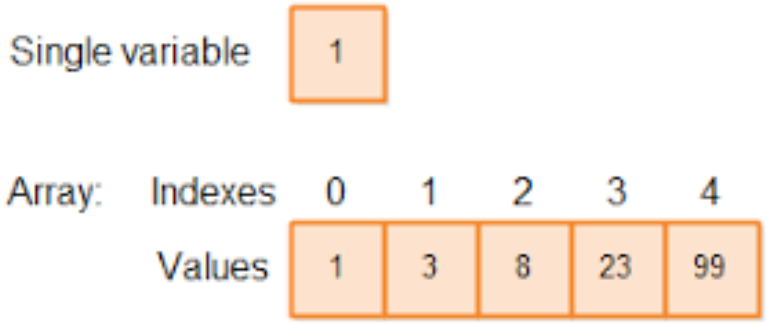
2.6 Skip Lists

2.1 Introduction to Arrays

- An array is a data structure that contains a group of elements.
- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value
- Array is the simplest data structure where each data element can be randomly accessed by using its index number
- Typically, these elements are all same data type, such as an integer or string.
- Arrays are commonly used in computer programs to organize data so that a related set of values can be easily sorted or searched.

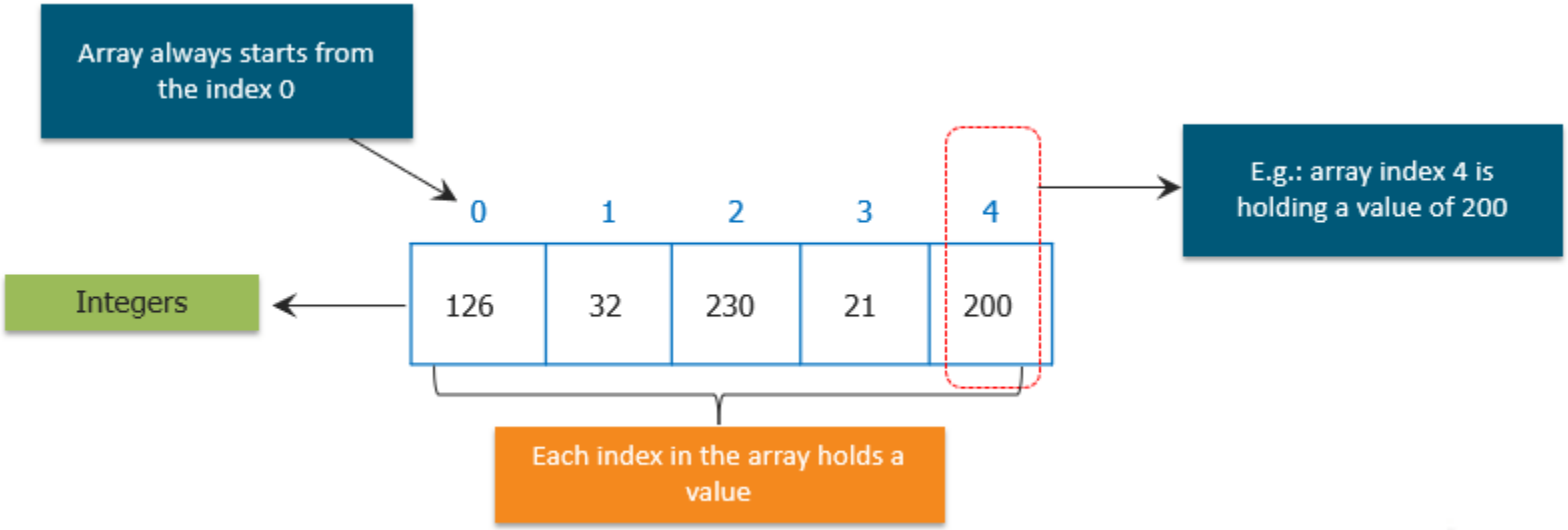
2.1 Introduction to Arrays

- In Java, the array is not a primitive type. Instead, it behaves very much like an object.
- Thus, many of the rules for objects also apply to arrays.
- One memory block is allocated for the entire array to hold the elements of the array.
- The array elements can be accessed in constant time by using the index of the particular element as the subscript.



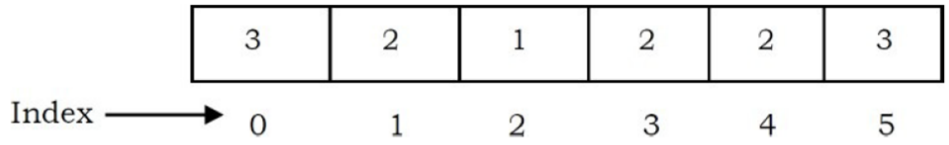
2.1.1 One dimensional arrays

- **Array Representation**



2.1.1 One dimensional arrays

- **Array Representation**



- Arrays are indexed starting at zero.
- Array declaration

  int[] intArray; // defines a reference to an array

  intArray = new int[100]; // creates the array, and sets intArray to refer to it

  - Or you can use the equivalent single-statement approach:

  int[] intArray = new int[100];

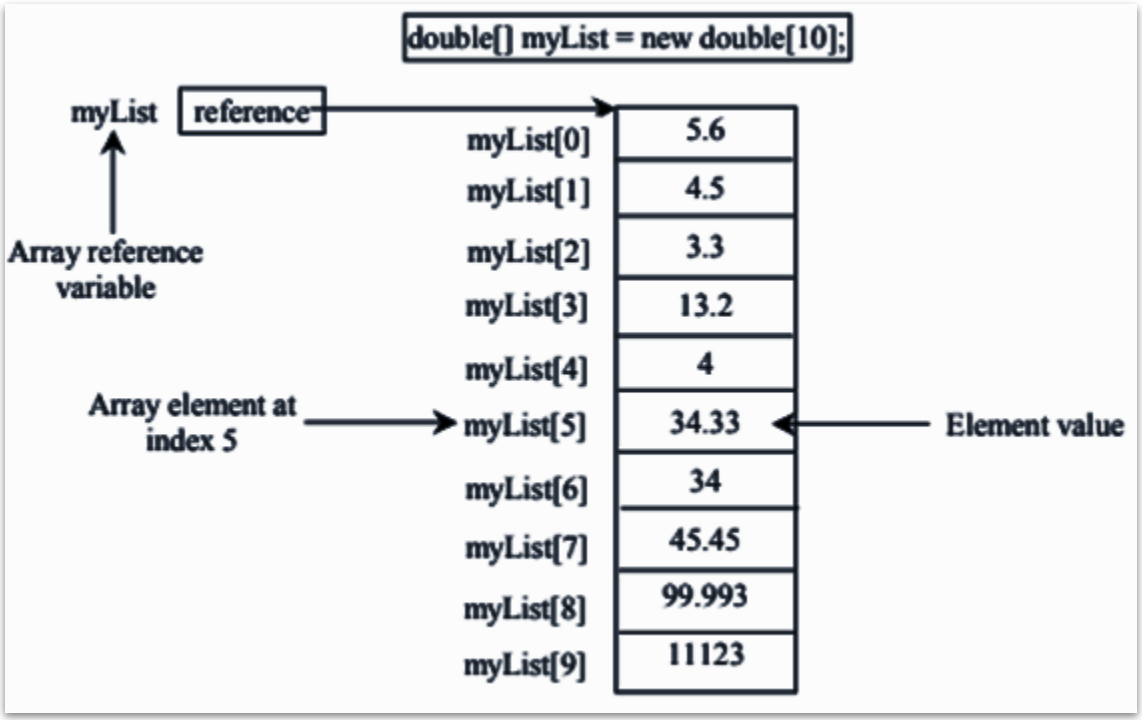
  - You can also use an alternative syntax for [ ] operator, placing it after the name instead of the type:

  int intArray[] = new int[100]; // alternative syntax

2.1.1 One dimensional arrays

- Example for declaring an array variable myList with 10 elements of double type

double[] myList = new double[10];



2.1.1 One dimensional arrays cont.

- **Accessing Array Elements**
  - Array elements are accessed using an index number in square brackets.

    temp = intArray[3]; // get contents of fourth element of array

    intArray[7] = 66; // insert 66 into the eighth cell
  - Remember that in Java, as in C and C++, the first element is numbered 0, so that the indices in an array of 10 elements run from 0 to 9.
  - If you use an index that's less than 0 or greater than the size of the array less 1, you'll get the **Array Index Out of Bounds** runtime error.

2.1.1 One dimensional arrays cont.

- **Initialization**
  - Say you create an array of objects like this:

    autoData[] carArray = new autoData[4000];
  - Until the array elements are given explicit values, they contain the special null object.
  - You can initialize an array of a primitive type to something besides 0 using this syntax:

    int[] intArray = { 0, 3, 6, 9, 12, 15, 18, 21, 24, 27 };
    - this single statement takes the place of both the reference declaration and the use of new to create the array.
  - The numbers within the curly brackets are called the initialization list.
  - The size of the array is determined by the number of values in this list.

2.1.1 One dimensional arrays cont.

- An Example for printing elements in an array

```
class ArrayApp
{
public static void main(String[] args)
{
long[] arr; // reference to array
arr = new long[100]; // make array
int nElems = 0; // number of items
int j; // loop counter

//-----------------------------------------------------------
arr[0] = 77; // insert 10 items
arr[1] = 99;
arr[2] = 44;
```

2.1.1 One dimensional arrays cont.

```
arr[3] = 55;
arr[4] = 22;
arr[5] = 88;
arr[6] = 11;
arr[7] = 00;
arr[8] = 66;
arr[9] = 33;
nElems = 10; // now 10 items in array
//------------------------------------------------------
for(j=0; j<nElems; j++) // display items
        System.out.print(arr[j] + " ");
System.out.println("");
```

2.1.2 Multi dimensional arrays

- Sometimes arrays need to be accessed by more than one index.
- In Java and Java Programming languages, we can create an array of arrays. These arrays are known as multidimensional arrays.
- A **multidimensional array** is an array that is accessed by **more than one index.**
- A common example of this is a matrix.
- It is allocated by specifying the size of its indices, and each element is accessed by placing each index in its own pair of brackets.

2.1.2 Multi dimensional arrays

- For example

```
int[][] a = new int[3][4];
```

Here, we have created a multidimensional array named **a**. It is a 2-dimensional array, that can hold a maximum of 12 elements.

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

You can think the array as a table with 3 rows and each row has 4 columns.

2.1.2 Multi dimensional arrays cont.

- An example to print the contents of a two-dimensional public class MatrixDemo

```
{
   public static void printMatrix( int [ ][ ] m )
    {
          for( int i = 0; i < m.length; i++ )
           {
                       if( m[ i ] == null )
                      System.out.println( "(null)" );
                       else
                      {
                      for( int j = 0; j < m[i].length; j++ )
                       System.out.print( m[ i ][ j ] + " " );
                       System.out.println( );
                      }
           }
   }
```

2.1.2 Multi dimensional arrays cont.

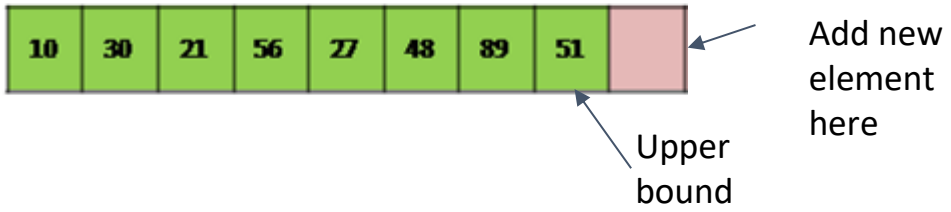
```
public static void main( String [ ] args )
 {
        int [ ][ ] a = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
        int [ ][ ] b = { { 1, 2 }, null, { 5, 6 } };
        int [ ][ ] c = { { 1, 2 }, { 3, 4, 5 }, { 6 } };

        System.out.println( "a: " ); printMatrix( a );
        System.out.println( "b: " ); printMatrix( b );
        System.out.println( "c: " ); printMatrix( c );
}
}
```

2.1.3 Basic operations on arrays

- **Insertion**
  - Inserting an item into the array is easy; we use the normal array syntax:
    arr[0] = 77;
  - We also keep track of how many items we've inserted into the array

Algorithm for inserting an element in the end of the array



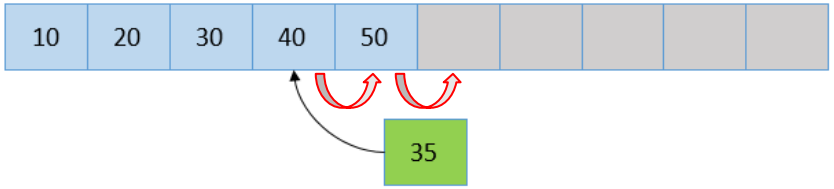
Step 1: Set upper_bound = upper_bound + 1

Step 2: Set A[upper_bound] = VAL

Step 3: EXIT

2.1.3 Basic operations on arrays

Algorithm for inserting element in the middle of the array



INSERT (A, N, POS, VAL).

- A - the array in which the element has to be inserted
- N - the number of elements in the array
- POS - the position at which the element has to be inserted
- VAL - the value that has to be inserted

Step 1: SET I = N // *Initialize Counter*

Step 2: Repeat Steps 3 and 4 while I >= POS

Step 3: SET A[I + 1] = A[I] //*Move Ith element downward*

Step 4: SET I = I – 1 // *Decrease Counter*

[END OF LOOP]

Step 5: SET N = N + 1 // *Reset N*

Step 6: SET A[POS] = VAL // *Insert element*

Step 7: EXIT

2.1.3 Basic operations on arrays

- **Searching**
  - To search for an item, we step through the array, comparing a particular value with each element.

Algorithm for searching element in an array

- A is a linear array with N elements
- K is a positive integer such that K<=N.

Step 1:Start

Step 2:Set J=0

Step 3:Repeat Step 4 & 5 while J<N

Step 4: If A[J] = ITEM THEN GO TO Step 6

Step 5:Set J=J+1

Step 6:Print J, Item

Step 7:Stop

2.1.3 Basic operations on arrays

- Example for search a key from an array :

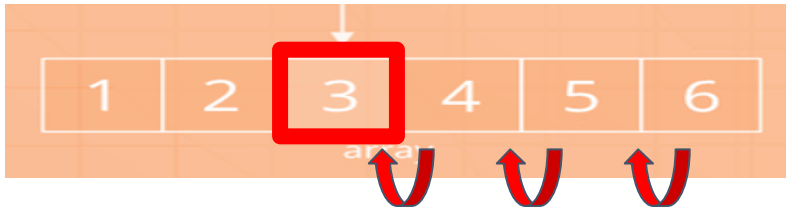
```
searchKey = 66; // find item with key 66
for(j=0; j<nElems; j++) // for each element,
        if(arr[j] == searchKey) // found item?
        break; // yes, exit before end
if(j == nElems) // at the end?
        System.out.println("Can't find " + searchKey); // yes
else
        System.out.println("Found " + searchKey); // no
```

- The searchKey variable holds the value we're looking for.
- If the loop variable j reaches the last occupied cell with no match being found, the value isn't in the array. Appropriate messages are displayed. Example :Found 66 or Can't find 27.

2.1.3 Basic operations on arrays cont.

- **Deletion**
  - Deletion begins with a search for the specified item.
  - For simplicity, we assume that the item is present.

Algorithm for deleting an element in the middle of the array



Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N – 1
Step 3: SET A[I] = A[I + 1]
Step 4: SET I = I + 1
[END OF LOOP]
Step 5: SET N = N – 1
Step 6: EXIT

2.1.3 Basic operations on arrays cont.

- Example for delete an element from an array

```
deleteKey = 55; // delete item with key 55
for(j=0; j<nElems; j++) // look for it
if(arr[j] == deleteKey)
        break;
for(int k=j; k<nElems-1; k++) // move higher ones down
        arr[k] = arr[k+1];
nElems--; // decrement size
```

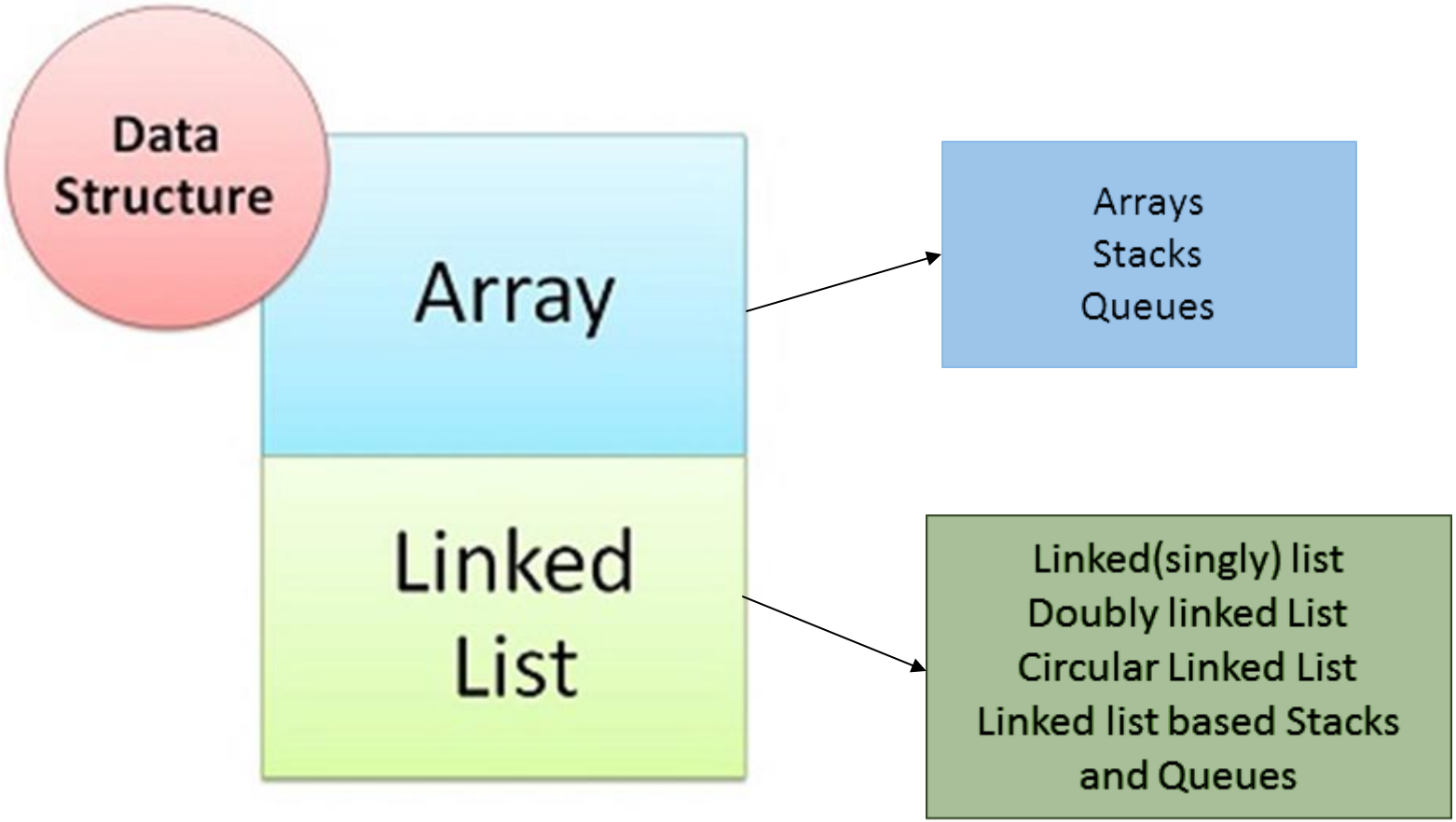
- When we find it, we move all the items with higher index values down one element to fill in the “hole” left by the deleted element, and we decrement nElems.
- In a real program, we would also take appropriate action if the item to be deleted could not be found.

2.1.3 Basic operations on arrays cont.

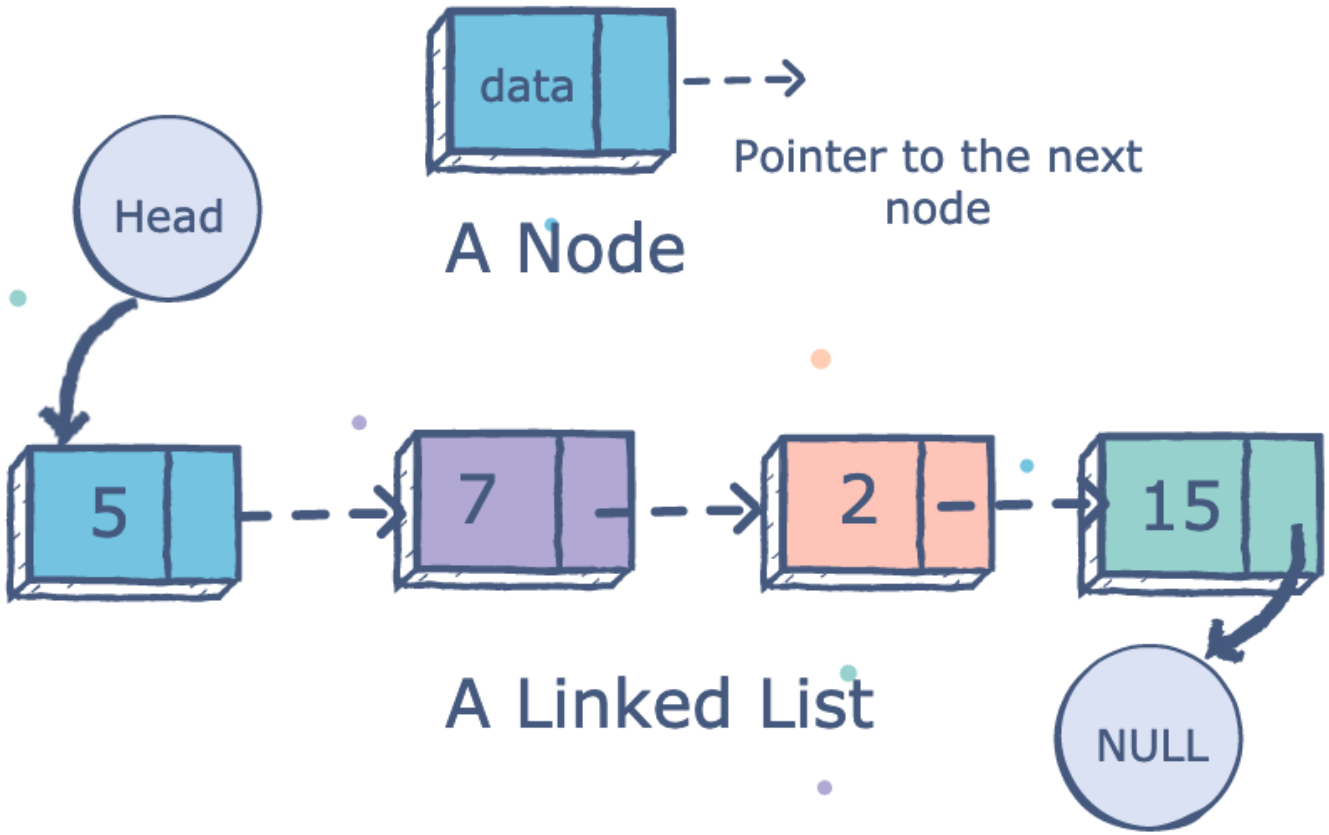
- **Display elements**
  - Displaying all the elements is straightforward: We step through the array, accessing each one with arr[j] and displaying it.
  - Example:

```
for(j=0; j<nElems; j++) // display items
        System.out.print( arr[j] + " ");
```

Introduction to Linked Lists



Introduction to Linked Lists



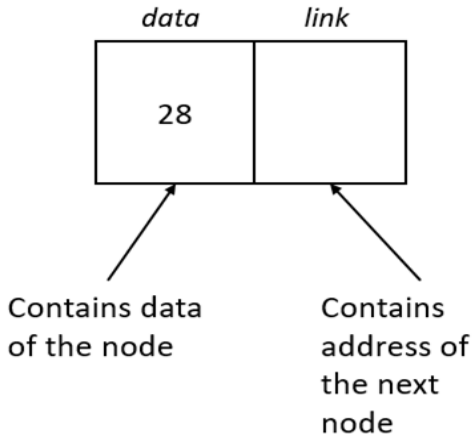
Introduction to Linked Lists

- The array is a linear collection of data elements
- It has following limitations:
  - Rigid Structure.
  - Can be hard to add/remove elements.
  - Cannot be dynamically resized in most languages.
  - Memory loss

- These limitations can be overcome by using linked structures
- While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store.
- For example, if we declare an array as int marks[10], then the array can store a maximum of 10 data elements but not more than that.

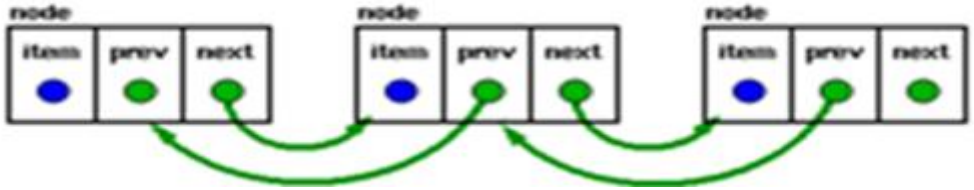
Introduction to Linked Lists cont.

- A linked structure is a collection of nodes storing data and links to other nodes.
- A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.



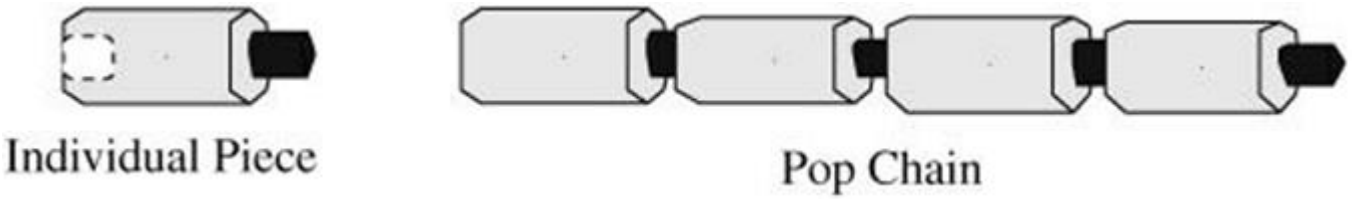
Introduction to Linked Lists cont.

- A linked list has the following properties
  - Successive elements are connected by pointers
  - The last element points to NULL
  - Can grow or shrink in size during execution of a program
  - Can be made just as long as required (until systems memory exhausts)
  - Does not waste memory space (but takes some extra memory for pointers). It
  - allocates memory as list grows.
- Linked structures can be implemented in a variety of ways, the most flexible implementation is by using a separate object for each node.
- A list where we can navigate backward as well



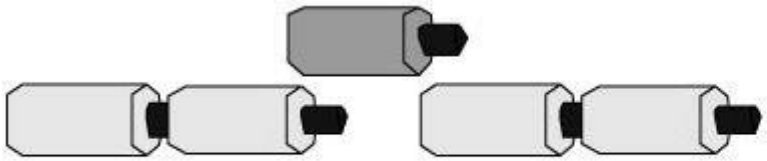
Introduction to Linked Lists cont.

- Think of each element in a linked list as being an individual piece in a child's pop chain.
- To form a chain, insert the connector into the back of the next piece



Introduction to Linked Lists cont.

- Inserting a new piece into the chain
  - ❑ breaking a connection and
  - ❑ reconnecting the chain at both ends of the new piece.

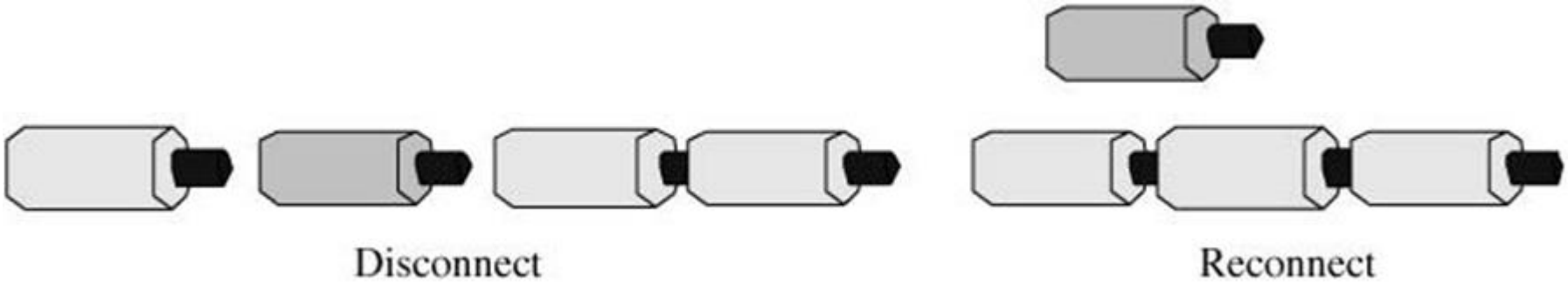


Disconnect

Reconnect

Introduction to Linked Lists cont.

- Removal of a piece
  - breaking its two connections
  - removing the piece, and then reconnecting the chain.



2.2 Comparison of Arrays and Linked Lists

Arrays	Linked lists
Fixed size	Dynamic size
Random access	No random access
Insertions and deletions are inefficient	Insertions and deletions are efficient
No memory wastage, if array is full or almost full. Otherwise memory wastage	Since memory is dynamically allocated (according to our requirement) there is no waste of memory
Sequential access is faster	Sequential access is not faster
Elements are in contiguous memory locations	Elements are not in contiguous memory locations

2.2 Comparison of Arrays and Linked Lists cont.

- **Advantages of Arrays**
  - Simple and easy to use
  - Faster access to the elements (constant access)
  - No need to declare large number of variables individually.
  - Variables are not scattered in memory,they are stored in contiguous memory

2.2 Comparison of Arrays and Linked Lists cont.

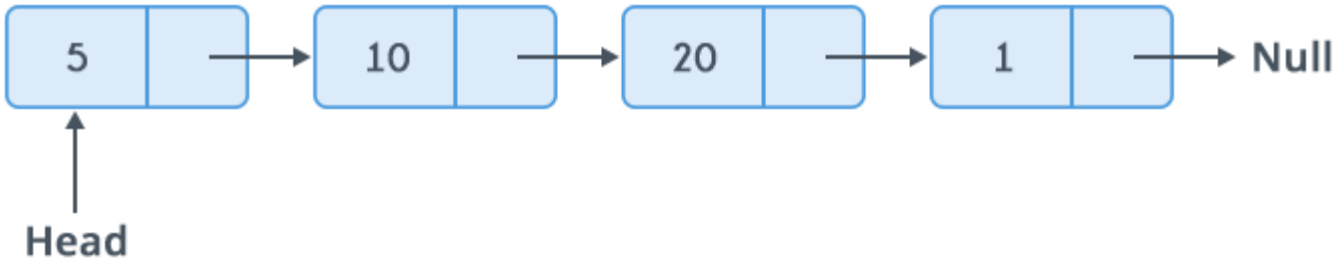
- **Disadvantages of Arrays**
  - Fixed size: The size of the array is static (specify the array size before using it).
  - One block allocation: To allocate the array itself at the beginning, sometimes it may not be possible to get the memory for the complete array (if the array size is big).
  - Complex position-based insertion: To insert an element at a given position, we may need to shift the existing elements. This will create a position for us to insert the new element at the desired position. If the position at which we want to add an element is at the beginning, then the shifting operation is more expensive.

2.2 Comparison of Arrays and Linked Lists cont.

- **Advantages of Linked Lists**
  - Linked lists can be expanded in constant time : With a linked list, we can start with space for just one allocated element and add on new elements easily without the need to do any copying and reallocating.

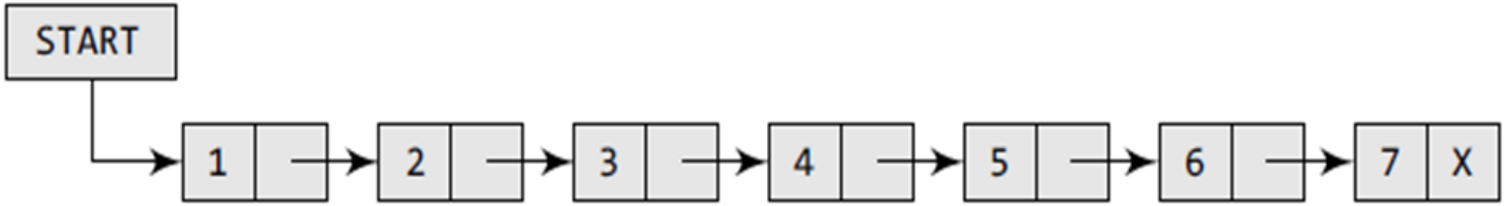
- **Disadvantages of Linked Lists**
  - Access time to individual elements : Array is random-access, which means it takes O(1) to access any element in the array. Linked lists take O(n) for access to an element in the list in the worst case.
  - Linked lists waste memory in terms of extra reference points.

2.3 Singly Linked Lists



- Generally “linked list” means a singly linked list.
- This list consists of a number of nodes in which each node has a next pointer to the following element.
- Each node consists of the data element and a link to the next node in the list.
- The link of the last node in the list is NULL, which indicates the end of the list.

2.3 Singly Linked Lists



- Linked lists contain a pointer variable START /HEAD that stores the address of the first node in the list.
- We can traverse the entire list using START which contains the address of the first node;
- If START = NULL, then the linked list is empty and contains no nodes
- Linked List type declaration example :

```
Public class ListNode {
        private int data;
        private ListNode next;
```

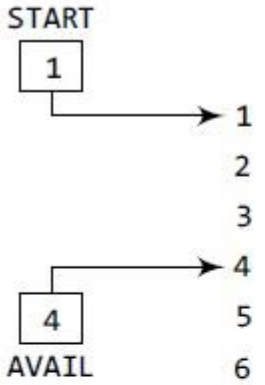
2.3 Singly Linked Lists

```
public ListNode (int data){
        this.data = data;
}
public void setData (int data){
        this.data  = data;
}
public int getData(){
        return data;
}
public void setNext (ListNode next){
        this.next = next;
}
public ListNode getNext(){
        return this.next;
}
}
```

2.3 Singly Linked Lists

- Linked list with **AVAI**L and **START** pointers

  - Every linked list has a pointer variable **START** which stores the address of the first node of the list.
  - Likewise, for the free pool (which is a linked list of all free memory cells), we have a pointer variable **AVAIL** which stores the address of the first free space.



	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4			6
5	S04	98	7
6			9
7	S05	55	8
8	S06	34	10
9			14
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			-1
15	S11	56	-1

2.3 Singly Linked Lists

- For example, when a new student's record has to be added, the memory address pointed by **AVAIL** will be taken and used to store the desired information.
- After the insertion, the next available free space's address will be stored in **AVAIL.**
- According to the above table, when the first free memory space is utilized for inserting the new node, AVAIL will be set to contain address 6.

2.3 Singly Linked Lists cont.

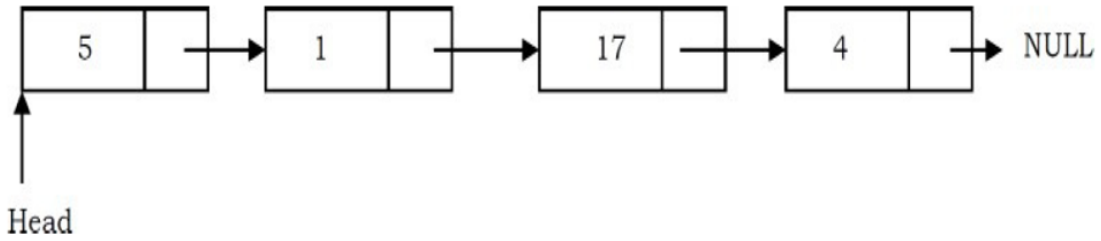
- **Basic Operations on a List**
  - Traversing the list
  - Inserting an item in the list
  - Deleting an item from the list

2.3 Singly Linked Lists cont.

- **Traversing the list**

Let us assume that the head points to the first node of the list. To traverse the list we do the following

- Follow the pointers.
- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to NULL.



2.3 Singly Linked Lists cont.

- **Algorithm for traversing a linked list**

Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: Apply Process to PTR.DATA
Step 4: SET PTR = PTR.NEXT
[END OF LOOP]
Step 5: EXIT

2.3 Singly Linked Lists cont.

- **Algorithm to print the number of nodes in a linked list**

Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4: SET COUNT = COUNT +1
Step 5: SET PTR = PTR.NEXT
[END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT

2.3 Singly Linked Lists cont.

The function given below can be used for printing the list data with extra print function.

```
Public int ListLength (ListNode headNode) {
    int length = 0;
    ListNode currentNode = headNode;
    while(currentNode != null){
            length++;
            currentNode = currentNode.getNext();
    }
    return length;
}
```

- The ListLength() function takes a linked list as input and counts the number of nodes in the list.

2.3 Singly Linked Lists cont.

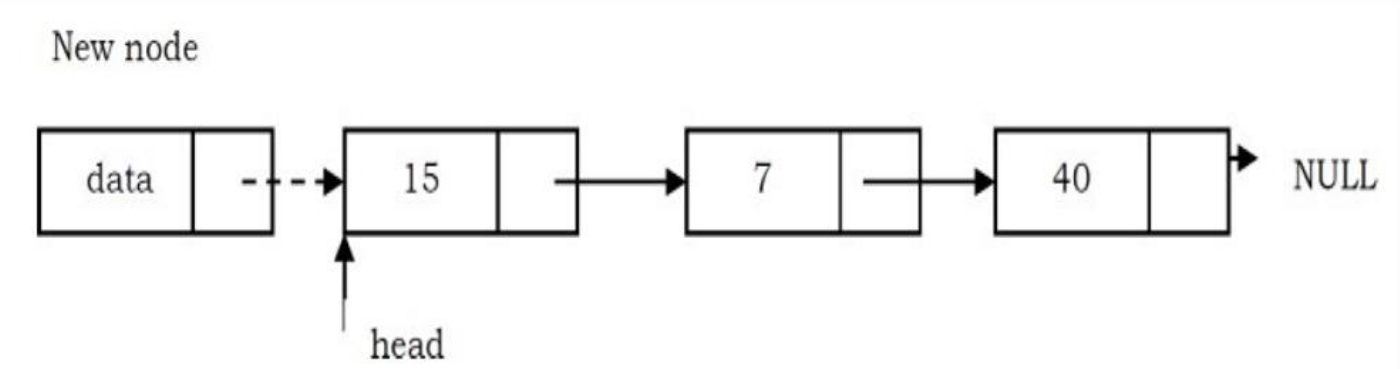
- **Insertion into a singly-linked list has three cases:**
  - Inserting a new node before the head (at the beginning)
  - Inserting a new node after the tail (at the end of the list)
  - Inserting a new node at the middle of the list (random location)

Inserting a Node in Singly Linked List at the Beginning

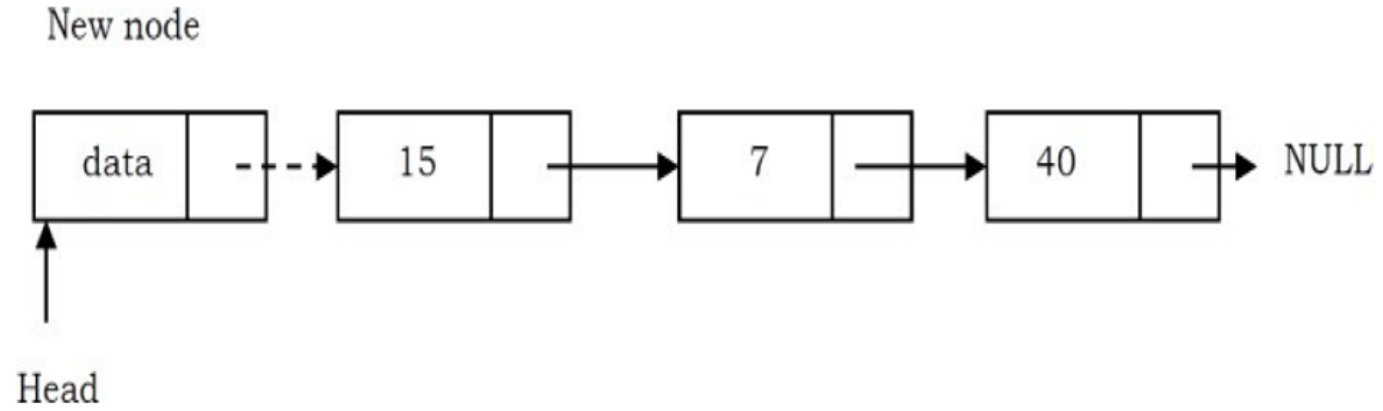
- In this case, a new node is inserted before the current head node.
- Only one next pointer needs to be modified (new node's next pointer) and it can be done in two steps:
  - Update the next pointer of new node, to point to the current head.
  - Update head pointer to point to the new node.

2.3 Singly Linked Lists cont.

- Update the next pointer of new node, to point to the current head.



- Update head pointer to point to the new node.



2.3 Singly Linked Lists cont.

- **Algorithm for inserting a Node in Singly Linked List at the Beginning**

Step 1: IF AVAIL = NULL
Write OVERFLOW

Go to Step 7
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL.NEXT
Step 4: SET NEW_NODE.DATA = VAL
Step 5: SET NEW_NODE.NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT

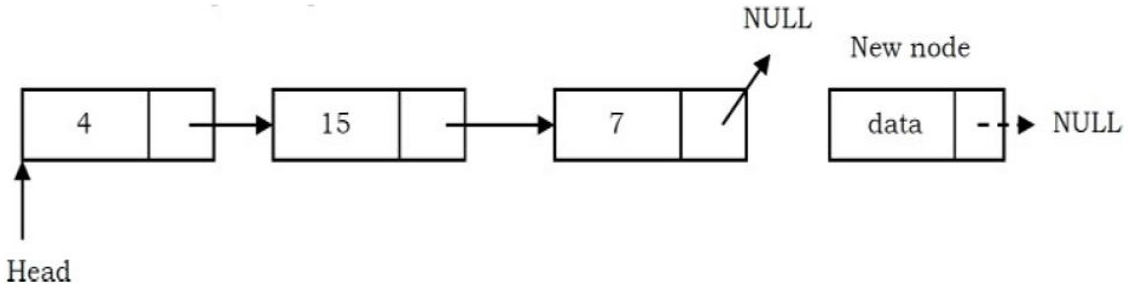
AVAIL is advanced

2.3 Singly Linked Lists cont.

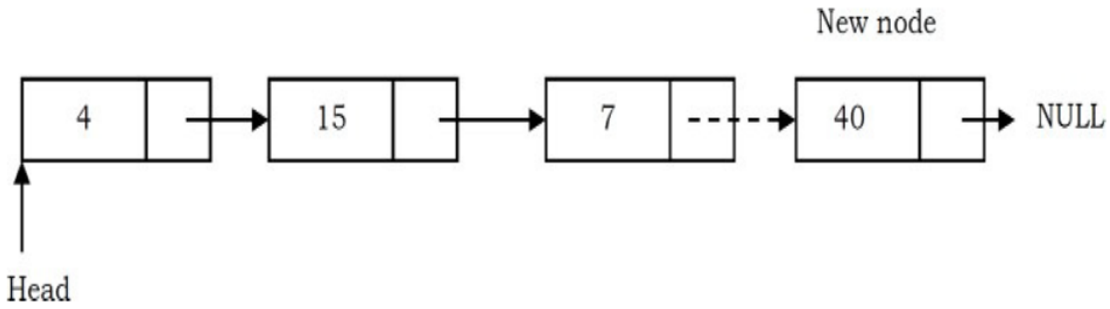
Inserting a Node in Singly Linked List at the End

In this case, we need to modify two next pointers (last nodes next pointer and new nodes next pointer).

- New nodes next pointer points to NULL.



- Last nodes next pointer points to the new node.



2.3 Singly Linked Lists cont.

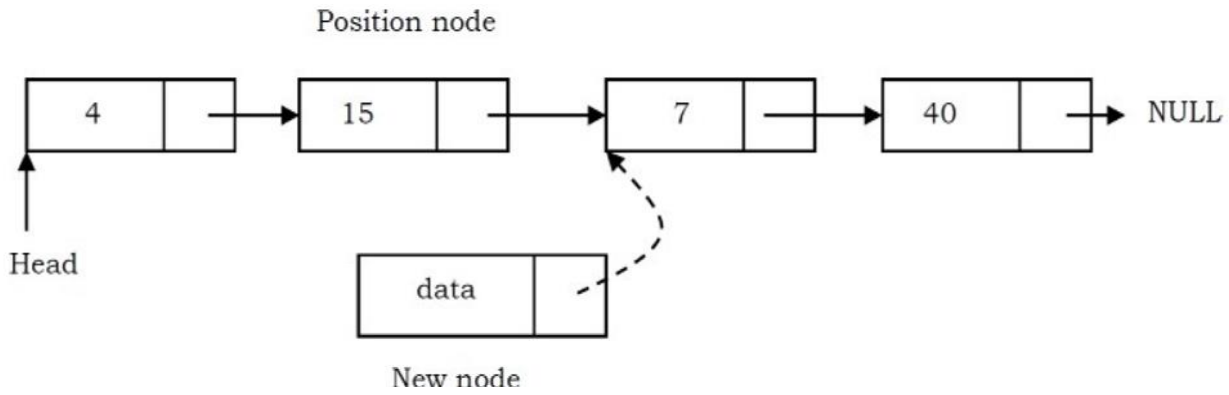
- **Algorithm for inserting a Node in Singly Linked List at the End**

Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 10
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL.NEXT
Step 4: SET NEW_NODE.DATA = VAL
Step 5: SET NEW_NODE.NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR.NEXT != NULL
Step 8: SET PTR = PTR NEXT
[END OF LOOP]
Step 9: SET PTR NEXT = NEW_NODE
Step 10: EXIT

2.3 Singly Linked Lists cont.

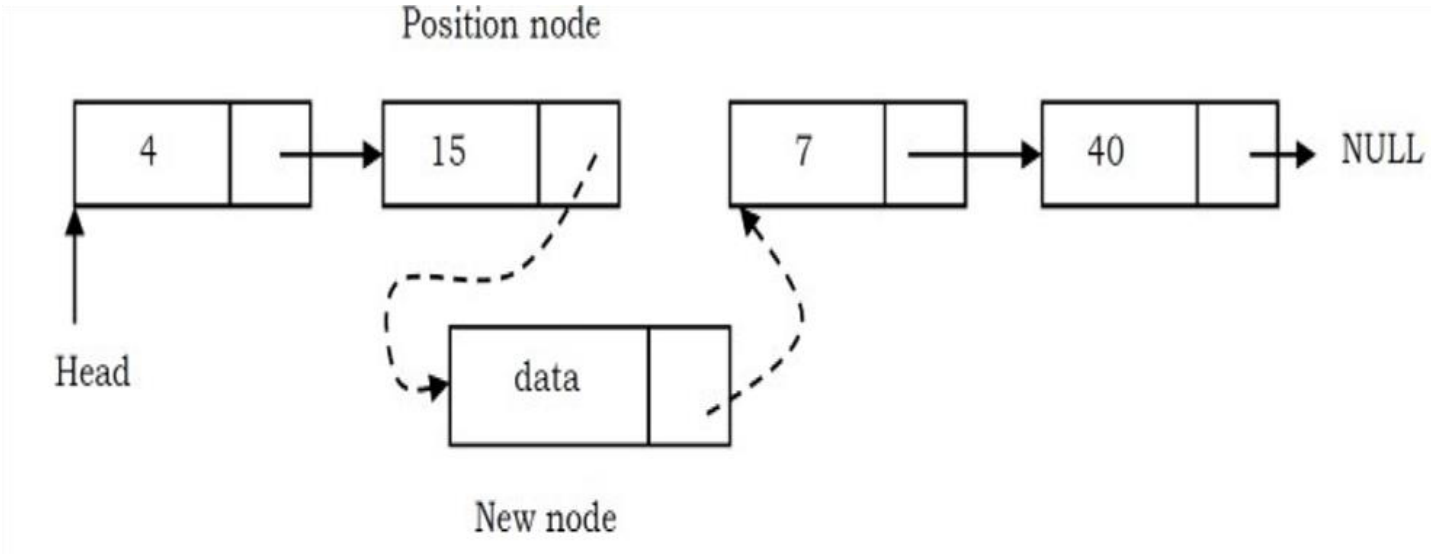
Inserting a Node in Singly Linked List at the Middle

- Let us assume that we are given a position where we want to insert the new node. In this case also, we need to modify two next pointers.
  - If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node. For simplicity let us assume that the second node is called position node. The new node points to the next node of the position where we want to add this node.



2.3 Singly Linked Lists cont.

- Position node's next pointer now points to the new node.



2.3 Singly Linked Lists cont.

- **Algorithm to insert a new node after a node that has value NUM**

Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 12
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL.NEXT
Step 4: SET NEW_NODE.DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR.DATA != NUM
Step 8: SET PREPTR = PTR
Step 9: SET PTR = PTR.NEXT
[END OF LOOP]
Step 1 : PREPTR .NEXT == NEW_NODE
Step 11: SET NEW_NODE. NEXT = PTR
Step 12: EXIT

2.3 Singly Linked Lists cont.

- Example 1 : Insert a node at the beginning of the list

```
public void insertBegin(int new_data)
{
      Node new_node = new Node(new_data);
     new_node.next = head;
     head = new_node;
}
```

2.3 Singly Linked Lists cont.

- Example 2 : Insert a node at the end of the list

```
public void insertEnd(int new_data)
{
    Node new_node = new Node(new_data);
    if (head == null)
    {
        head = new Node(new_data);
        return;
    }
     new_node.next = null;
     Node last = head;
     while (last.next != null)
                Node  last = last.next;
     last.next = new_node;
    return;
}
```

2.3 Singly Linked Lists cont.

- Example 3 : Insert a node after a given node

```
public void insertAfter(Node prev_node, int new_data)
{
    if (prev_node == null)
    {
        System.out.println("The given previous node cannot be
  null");
        return;
    }
      Node new_node = new Node(new_data);
      new_node.next = prev_node.next;
      prev_node.next = new_node;
}
```

2.3 Singly Linked Lists cont.

Singly Linked List Deletion

Similar to insertion, here also we have three cases.

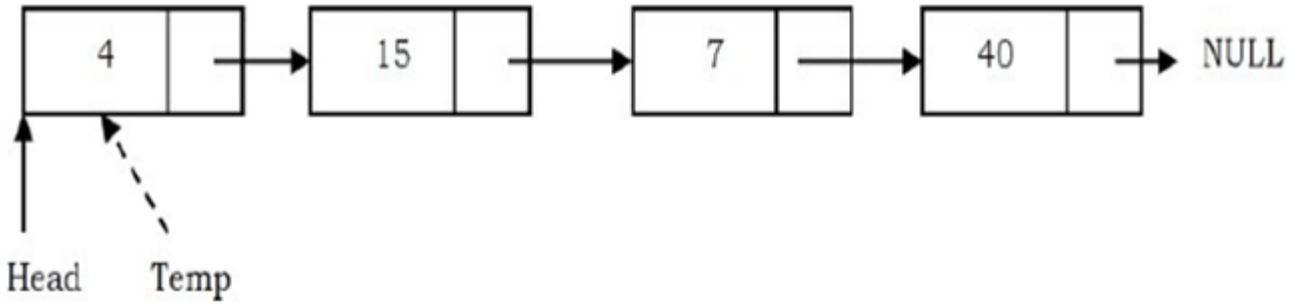
- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

Deleting the First Node in Singly Linked List

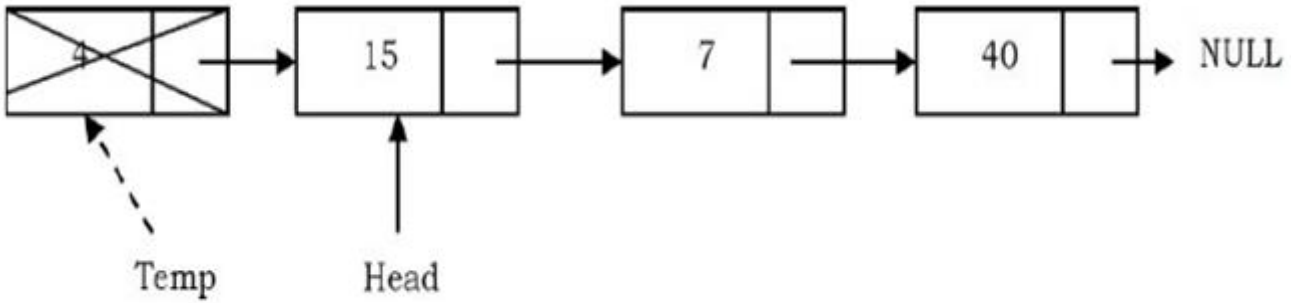
- First node (current head node) is removed from the list. It can be done in two steps:
  - Create a temporary node which will point to the same node as that of head.
  - move the head nodes pointer to the next node and dispose of the temporary node.

2.3 Singly Linked Lists cont.

- Create a temporary node which will point to the same node as that of head.



- Now, move the head nodes pointer to the next node and dispose of the temporary node.



2.3 Singly Linked Lists cont.

- **Algorithm to delete the First Node in Singly Linked List**

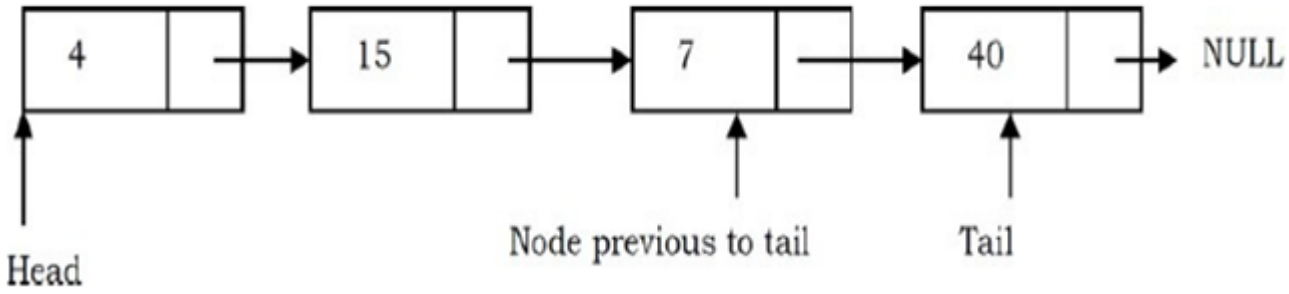
```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START.NEXT
Step 4: FREE PTR
Step 5: EXIT
```

2.3 Singly Linked Lists cont.

Deleting the Last Node in Singly Linked List

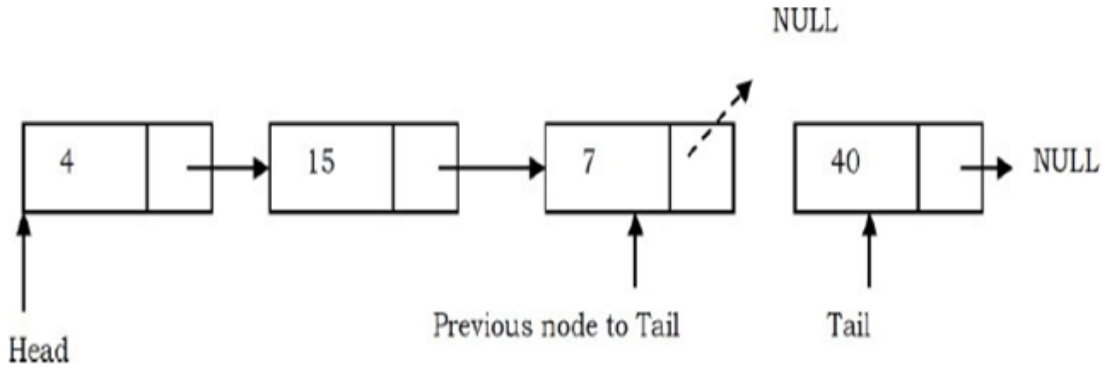
In this case, the last node is removed from the list. It can be done in three steps:

- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail node and the other pointing to the node before the tail node.

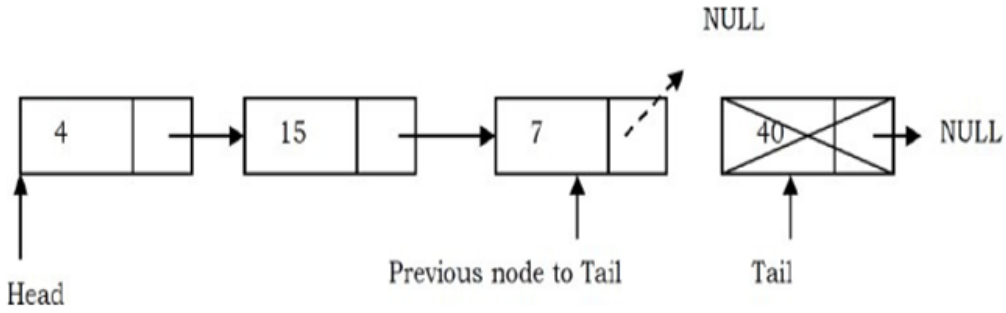


2.3 Singly Linked Lists cont.

- Update previous node's next pointer with NULL



- Dispose of the tail node.



2.3 Singly Linked Lists cont.

- **Algorithm to delete the Last Node in Singly Linked List**

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR.NEXT != NULL
Step 4:    SET PREPTR = PTR
Step 5:    SET PTR = PTR.NEXT
         [END OF LOOP]
Step 6: SET PREPTR .NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

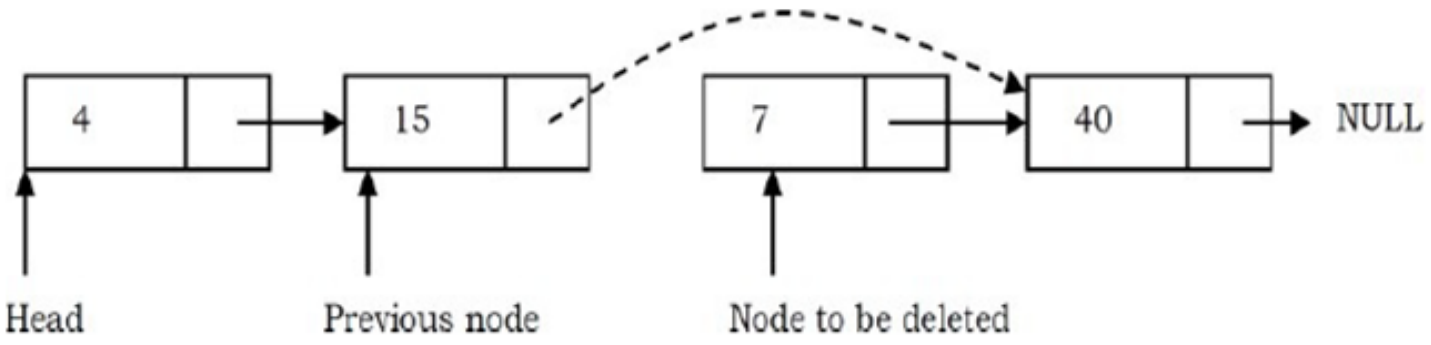
2.3 Singly Linked Lists cont.

Deleting an Intermediate Node in Singly Linked List

In this case, the node to be removed is always located between two nodes.

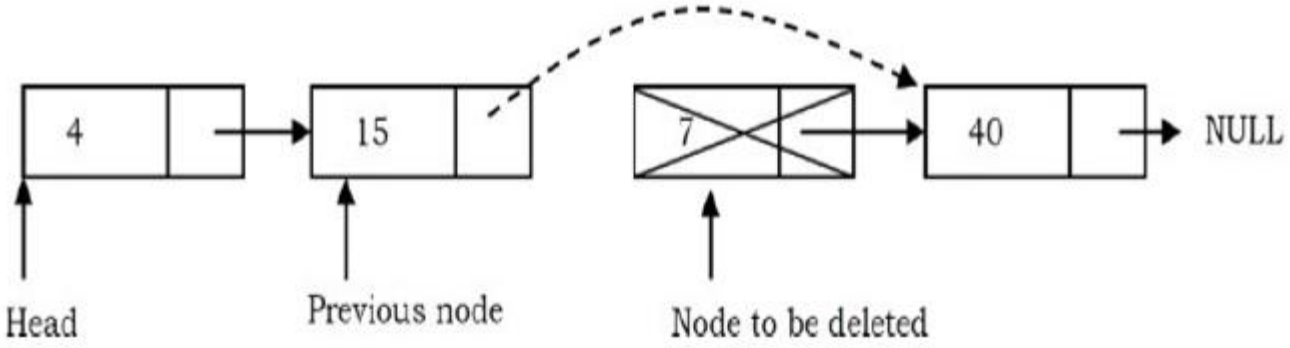
Head and tail links are not updated in this case. Such a removal can be done in two steps:

- Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.



2.3 Singly Linked Lists cont.

- Dispose of the current node to be deleted.



2.3 Singly Linked Lists cont.

- **Algorithm to delete the node after a given node**

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 9

[END OF IF]

Step 2: SET PTR = START

Step 3: SET PREPTR = PTR

Step 4: Repeat Steps 5 and 6 while PREPTR.DATA != NUM

Step 5: SET PREPTR = PTR

Step 6: SET PTR = PTR.NEXT

[END OF LOOP]

Step 7: SET TEMP = PTR

Step 8: SET PREPTR. NEXT = PTR .NEXT

Step 9 : EXIT

2.3 Singly Linked Lists cont.

Example 1 : Delete first node in Singly Linked List

```
        static Node removeFirstNode(Node head)
{
  if (head == null)
  {
        return null;
  }
    Node temp = head;
    head = head.next;
    return head;
}
```

2.3 Singly Linked Lists cont.

Example 2 :Delete Last node in Singly Linked List

```
static Node removeLastNode(Node head) {

    if(head == null) {
        return null;
    }

    Node temp = head;
    while(temp.next.next != null) {
        temp = temp.next;
    }
    temp.next = null;

    return head;
}
```

2.4 Doubly Linked Lists

- A doubly linked list allows bidirectional traversal by storing two links per node.
- The advantage of a doubly linked list is that given a node in the list, we can navigate in both directions.
- Doubly linked list cell consists of three parts:
  - Data
  - A pointer to the next node
  - A pointer to the previous node
- Doubly Linked List type declaration example :

```
Public class ListNode {
        private int data;
        private ListNode prev;
        private ListNode next;
```

2.4 Doubly Linked Lists Cont.

Doubly Linked List Insertion

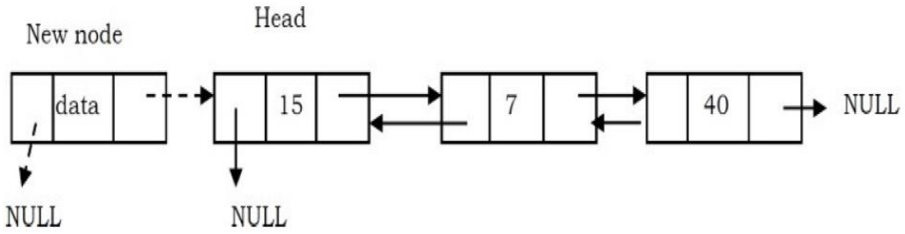
- Insertion into a doubly-linked list has three cases (same as singly linked list):
  - Inserting a new node before the head.
  - Inserting a new node after the tail (at the end of the list).
  - Inserting a new node at the middle of the list.

Inserting a Node in Doubly Linked List at the Beginning

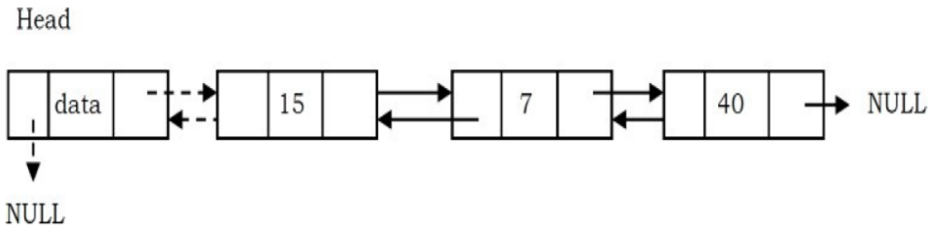
- In this case, new node is inserted before the head node. Previous and next pointers need to be modified and it can be done in two steps:
  - Update the right pointer of the new node to point to the current head node (dotted link in below figure) and also make left pointer of new node as NULL.
  - Update head node's left pointer to point to the new node and make new node as head. Head

2.4 Doubly Linked Lists Cont.

- Update the right pointer of the new node to point to the current head node (dotted link in below figure) and also make left pointer of new node as NULL.



- Update head node's left pointer to point to the new node and make new node as head. Head



2.4 Doubly Linked Lists Cont.

- **Algorithm to insert a new node at the beginning**

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE= AVAIL
Step 3: SET AVAIL = AVAIL.NEXT
Step 4: SET NEW_NODE.DATA = VAL
Step 5: SET NEW_NODE.PREV = NULL
Step 6: SET NEW_NODE.NEXT = START
Step 7: SET START.PREV = NEW_NODE
Step 8: SET START =NEW_NODE
Step 9: EXIT
```

2.4 Doubly Linked Lists Cont.

Inserting a Node in Doubly Linked List at the Ending

- In this case, traverse the list till the end and insert the new node.
    - New node right pointer points to NULL and left pointer points to the end of the list.
    - Update right pointer of last node to point to new node.

2.4 Doubly Linked Lists Cont.

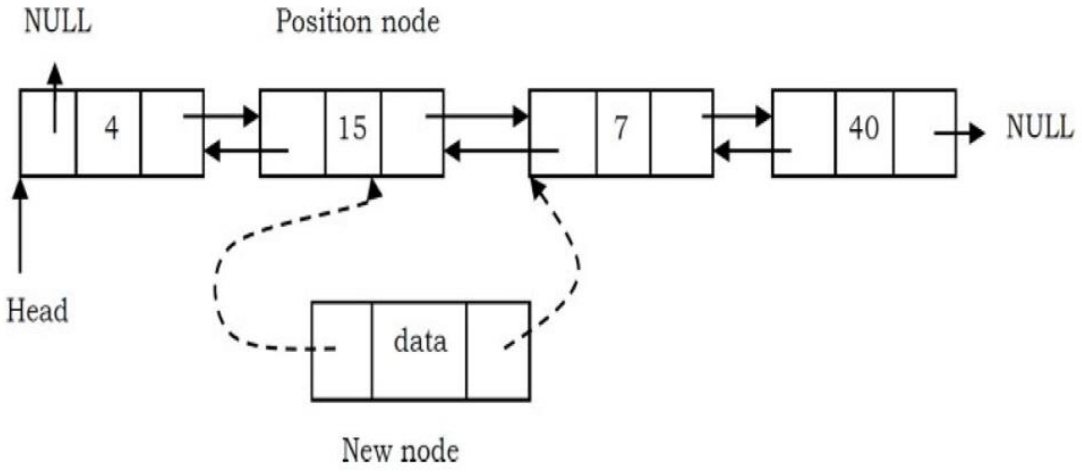
- **Algorithm to insert a new node at the end**

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE  = AVAIL
Step 3: SET AVAIL = AVAIL..NEXT
Step 4: SET NEW_NODE..DATA = VAL
Step 5: SET NEW_NODE.NEXT = NULL
Step 6: SET PTR = START
Step 7:Repeat Step 8 while PTR NEXT != NULL
Step 8:     SET PTR = PTR NEXT
       [END OF LOOP]
Step 9: SET PTR NEXT = NEW_NODE
Step 10 : SET NEW_NODE.PREV = PTR
Step 11: EXIT
```

2.4 Doubly Linked Lists Cont.

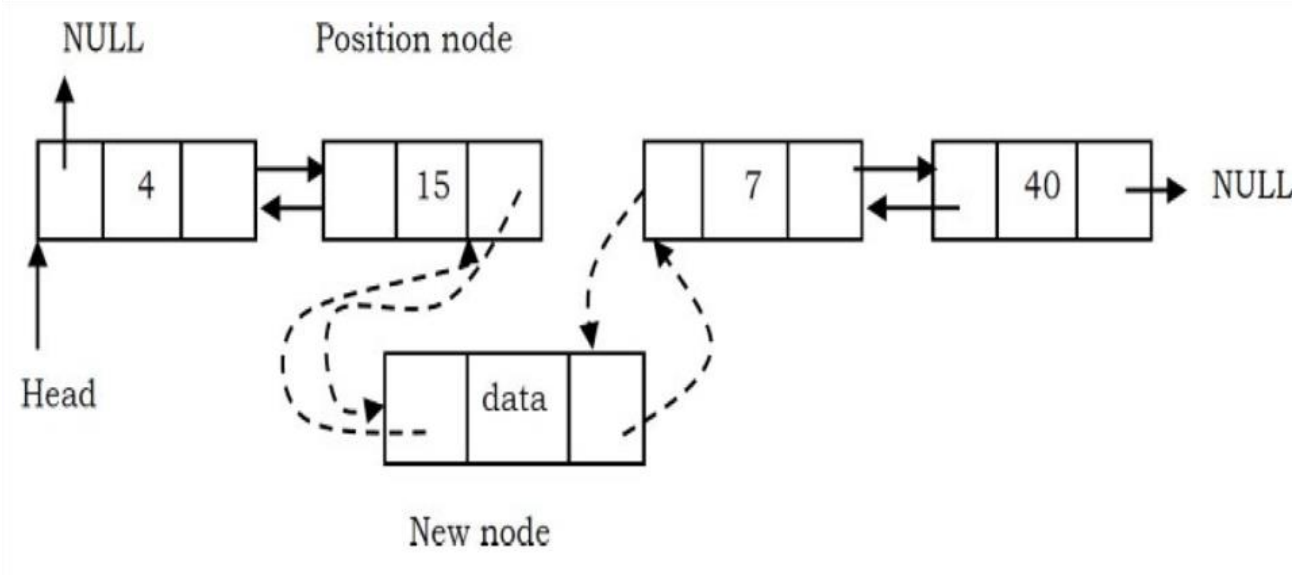
Inserting a Node in Doubly Linked List at Middle

- Similar to singly linked lists, traverse the list to the position node and insert the new node.
  - New node right pointer points to the next node of the position node where we want to insert the new node. Also, new node left pointer points to the position node.



2.4 Doubly Linked Lists Cont.

- Position node right pointer points to the new node and the next node of position node left pointer points to new node.



2.4 Doubly Linked Lists Cont.

- **Algorithm to insert a new node at the middle**

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
          [END OF IF]
Step 2: SET NEW_NODE  = AVAIL
Step 3: SET AVAIL = AVAIL..NEXT
Step 4: SET NEW_NODE..DATA = VAL
Step 5: SET PTR = START
Step 6:Repeat Step 7 while PTR DATA != NULL
Step 7:     SET PTR = PTR NEXT
       [END OF LOOP]
Step 8: SET NEW_NODE. NEXT = PTR.NEXT
Step 9: SET NEW_NODE.PREV = PTR
Step 10 : SET PTR.NEXT =NEW_NODE
Step 11 : SET PTR.NEXT.PREV =NEW_NODE
Step 12: EXIT
```

2.4 Doubly Linked Lists Cont.

Doubly Linked List Deletion

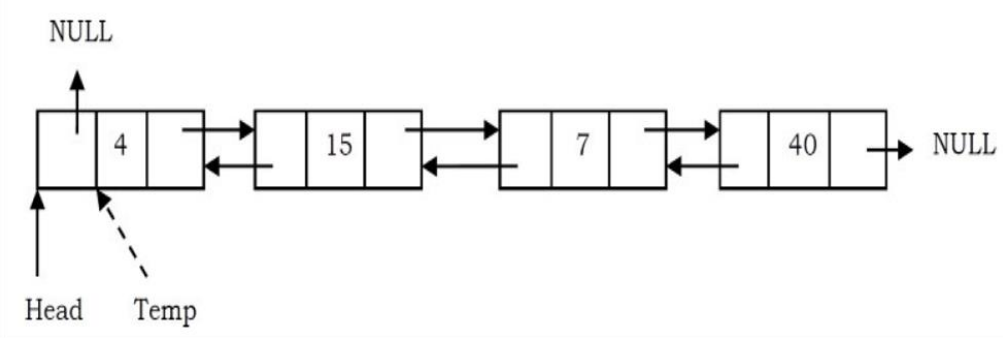
- Similar to singly linked list deletion, here we have three cases:
  - Deleting the first node
  - Deleting the last node
  - Deleting an intermediate node

Deleting the First Node in Doubly Linked List

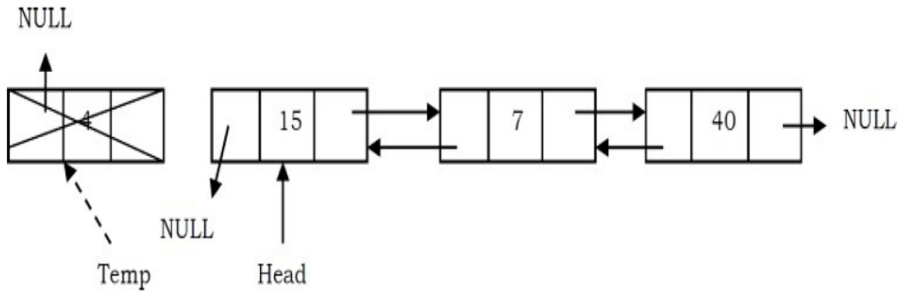
- In this case, the first node (current head node) is removed from the list. It can be done in two steps:
  - Create a temporary node which will point to the same node as that of head.
  - Now, move the head nodes pointer to the next node and change the heads left pointer to NULL. Then, dispose of the temporary node.

2.4 Doubly Linked Lists Cont.

- Create a temporary node which will point to the same node as that of head.



- Move the head nodes pointer to the next node and change the heads left pointer to NULL. Then, dispose of the temporary node.



2.4 Doubly Linked Lists Cont.

- **Algorithm to delete the first node**

Step 1: IF START = NULL
Write UNDERFLOW
Go to Step 5
[END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START.NEXT
Step 4 : SET START. PREV = NULL
Step 5: EXIT

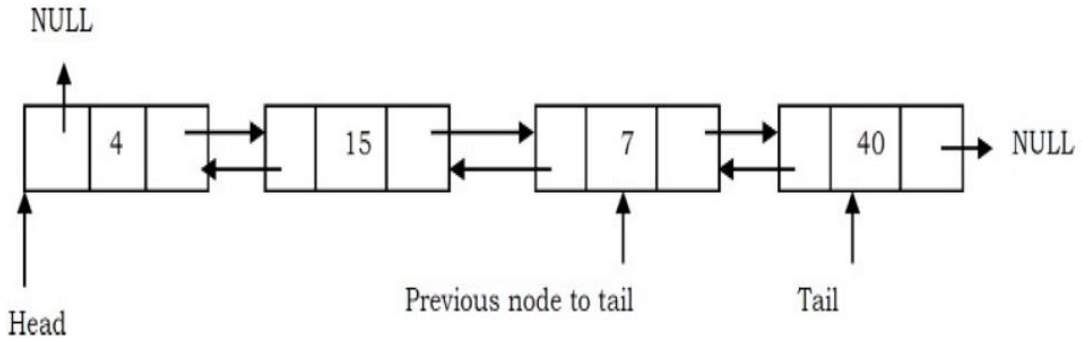
2.4 Doubly Linked Lists Cont.

Deleting the Last Node in Doubly Linked List

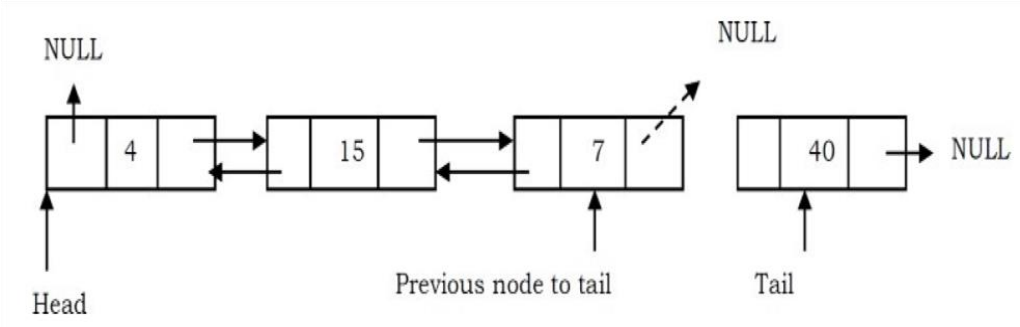
- This operation is a bit trickier than removing the first node, because the algorithm should find a node, which is previous to the tail first. This can be done in three steps:
  - Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail and the other pointing to the node before the tail.
  - Update the next pointer of previous node to the tail node with NULL.
  - Dispose the tail node.

2.4 Doubly Linked Lists Cont.

- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail and the other pointing to the node before the tail.

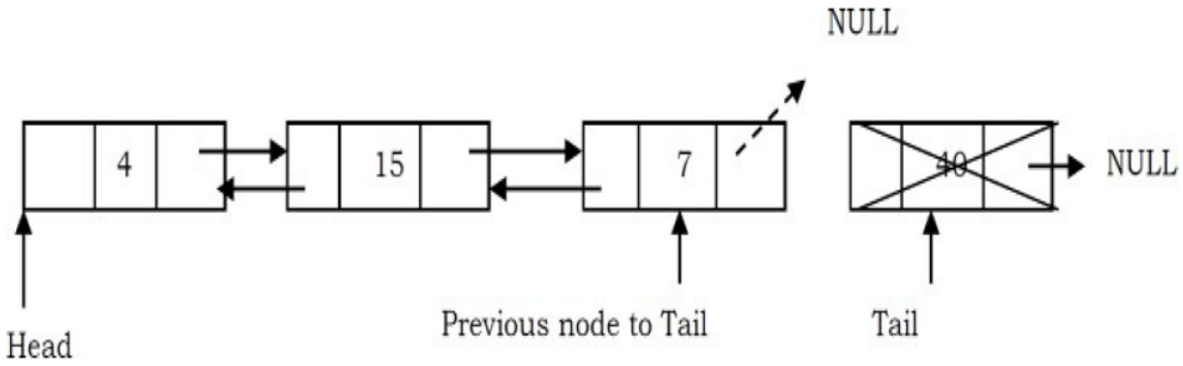


- Update the next pointer of previous node to the tail node with NULL.



2.4 Doubly Linked Lists Cont.

- Dispose the tail node.



2.4 Doubly Linked Lists Cont.

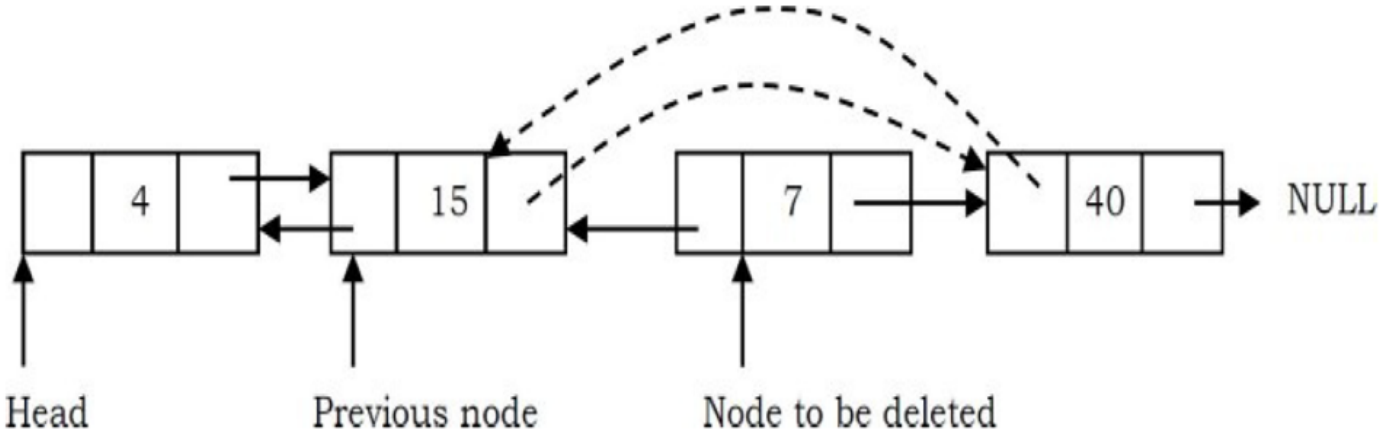
- **Algorithm to delete the last node**

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR.NEXT != NULL
Step 4:    SET PTR = PTR NEXT
            [END OF LOOP]
Step 5: SET PTR.PREV.NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

2.4 Doubly Linked Lists Cont.

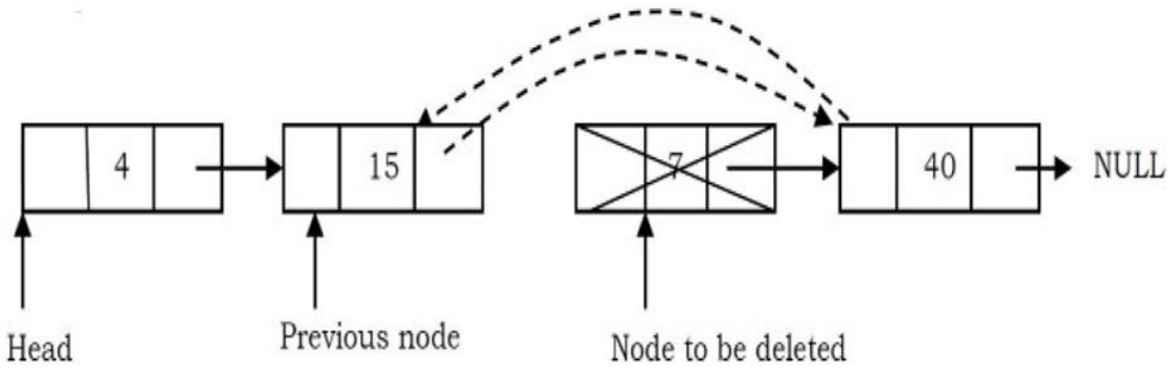
Deleting the Intermediate Node in Doubly Linked List

- In this case, the node to be removed is always located between two nodes, and the head and tail links are not updated. The removal can be done in two steps:
  - Similar to the previous case, maintain the previous node while also traversing the list. Upon locating the node to be deleted, change the previous node's next pointer to the next node of the node to be deleted.



2.4 Doubly Linked Lists Cont.

- Dispose of the current node to be deleted.



2.4 Doubly Linked Lists Cont.

- **Algorithm to delete a node after a given node**

Step 1: IF START = NULL
Write UNDERFLOW
Go to Step 9
[END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR.DATA != NUM
Step 4: SET PTR = PTR.NEXT
[END OF LOOP]
Step 5: SET TEMP = PTR.NEXT
Step 6: SET PTR. NEXT = TEMP. NEXT
Step 7: SET TEMP. NEXT.PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT

2.4 Doubly Linked Lists Cont.

- **Example: Adding To The End**

```
public void addToEnd ()
{
        Node anotherNode = new Node (currentDataValue);
        Node temp;
                if (isEmpty() == true)
                        head = anotherNode;
                else {
                        temp = head;
                while (temp.next != null){
                        temp = temp.next;
                }
                temp.next = anotherNode;
                anotherNode.previous = temp;
        }
        currentDataValue += 10;
        length++;
}
```

2.4 Doubly Linked Lists Cont.

- **Example :Adding Anywhere (1)**

```
public void addToPosition (int position)
{
    Node anotherNode = new Node (currentDataValue);
    Node temp;
    Node prior;
    Node after;
    int index;

    if ((position < 1) || (position > (length+1)))
    {
    System.out.println("Position must be a value between 1-" +
    (length+1));
    }
```

2.4 Doubly Linked Lists Cont.

- **Example :Adding Anywhere (2)**

```
else
{
// List is empty
if (head == null)
        {
        if (position == 1)
        {
                currentDataValue += 10;
                length++;
                head = anotherNode;
        }
else
System.out.println("List empty, unable to add node to " +"position " +
position);
}
```

2.4 Doubly Linked Lists Cont.

- **Example :Adding Anywhere (3)**

```
// List is not empty, inserting into first position.
else if (position == 1)
{
    head.previous = anotherNode;
    anotherNode.next = head;
    head = anotherNode;
    currentDataValue += 10;
    length++;
}
```

2.4 Doubly Linked Lists Cont.

- **Example :Adding Anywhere (4)**

```
// List is not empty inserting into a position other than the first
else
{
prior = head;
index = 1;
// Traverse list until current is referring to the node in front
// of the position that we wish to insert the new node into.
while (index < (position-1))
{
        prior = prior.next;
        index++;
}
after = prior.next;
```

2.4 Doubly Linked Lists Cont.

- **Example :Adding Anywhere (5)**

```
            // Set the references to the node before the node to be inserted.
            prior.next = anotherNode;
            anotherNode.previous = prior;
            // Set the references to the node after the node to be
            // inserted.
            if (after != null)
            after.previous = anotherNode;
            anotherNode.next = after;
            currentDataValue += 10;
            length++;
            }
        }
}
```

2.4 Doubly Linked Lists Cont.

- **Example: Deleting A Node (1)**

```
public void delete (int key)
{
int indexToDelete;
int indexTemp;
Node previous;
Node toBeDeleted;
Node after;
indexToDelete = search(key);
// No match, nothing to delete.
if (indexToDelete == -1)
{
        System.out.println("Cannot delete element with a data value of "
        + key + " because it was not found.");
}
```

2.4 Doubly Linked Lists Cont.

- **Example :Deleting A Node (2)**

```
else
{
// Deleting first element.
if (indexToDelete == 1)
{
        head = head.next;
        length--;
}
else
{
        previous = null;
        toBeDeleted = head;
        indexTemp = 1;
```

2.4 Doubly Linked Lists Cont.

- **Example :Deleting A Node (3)**

```
        while (indexTemp < indexToDelete)
        {
            previous = toBeDeleted;
            toBeDeleted = toBeDeleted.next;
            indexTemp++;
        }
        previous.next = toBeDeleted.next;
        after = toBeDeleted.next;
        after.previous = previous;
        length--;
        }
    }
}
```

2.5 Circular Linked Lists

- In singly linked lists and doubly linked lists, the end of lists are indicated with NULL value.
- But circular linked lists do not have ends. While traversing the circular linked lists we should be careful; otherwise we will be traversing the list infinitely.

2.5 Circular Linked Lists Cont.

- While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started.
- In circular linked lists, each node has a successor.
- Note that unlike singly linked lists, there is **no node** with **NULL pointer** in a circularly Linked list.
- In some situations, circular linked lists are useful.
  - For example, when several processes are using the same computer resource (CPU) for the same amount of time, we have to assure that no process accesses the resource before all other processes do (round robin algorithm).
  - When we are surfing the Internet, we can use the Back button and the Forward button to move to the previous pages that we have already visited.

2.5 Circular Linked Lists Cont.

- An Example Of Traversing A Circular Linked List

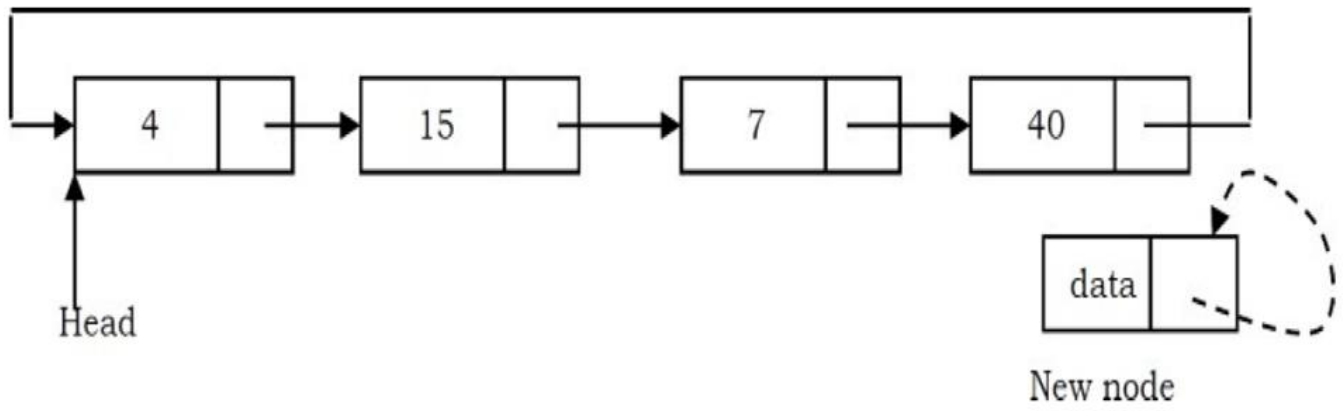
```
public void display ()
{
        Node temp = list;
        System.out.println("Displaying list");
        if (isEmpty() == true)
        {
                System.out.println("Nothing to display, list is empty.");
        }
        do
        {
                System.out.println(temp.data);
                temp = temp.next;
        } while (temp != list);
        System.out.println();
}
```

2.5 Circular Linked Lists Cont.

Inserting a Node at the End of a Circular Linked List

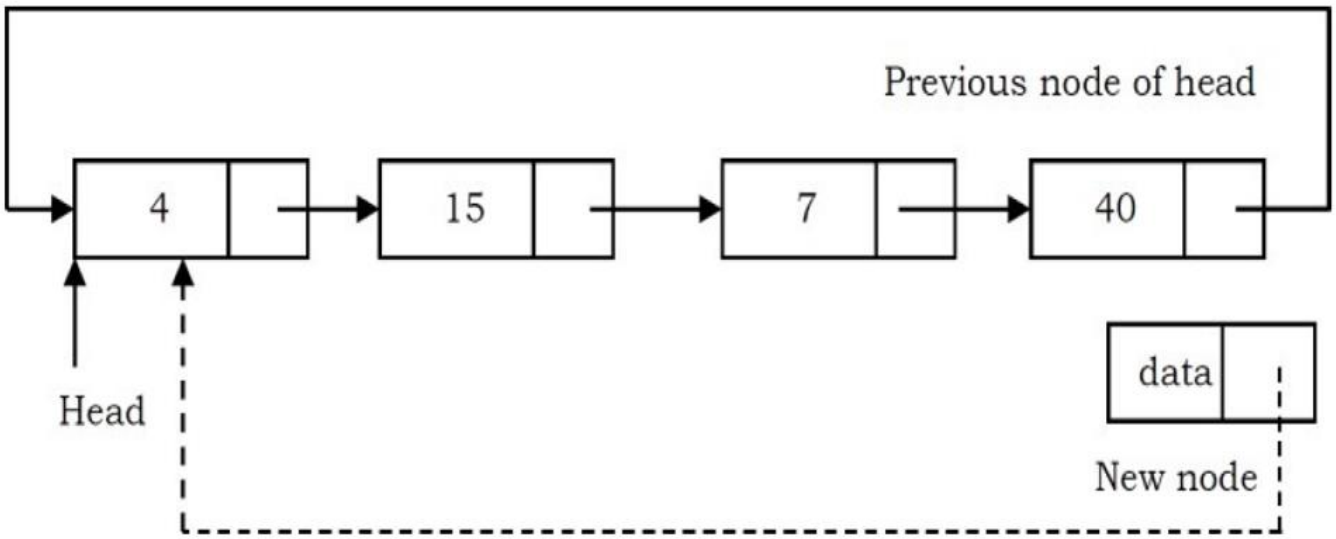
- Let us add a node containing data, at the end of a list (circular list) headed by head. The new node will be placed just after the tail node (which is the last node of the list), which means it will have to be inserted in between the tail node and the first node.

- Create a new node and initially keep its next pointer pointing to itself.



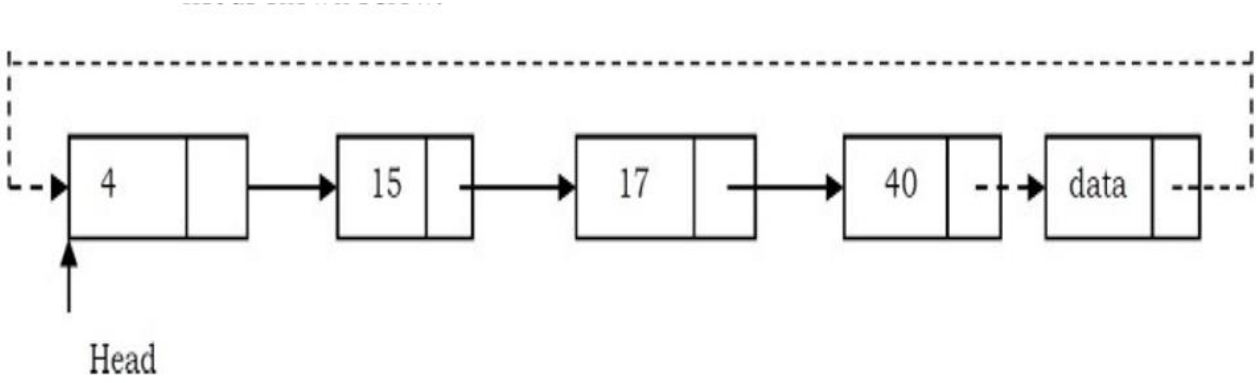
2.5 Circular Linked Lists Cont.

- Update the next pointer of the new node with the head node and also traverse the list to the tail. That means in a circular list we should stop at the node whose next node is head.



2.5 Circular Linked Lists Cont.

- Update the next pointer of the previous node to point to the new node and we get the list as shown below.

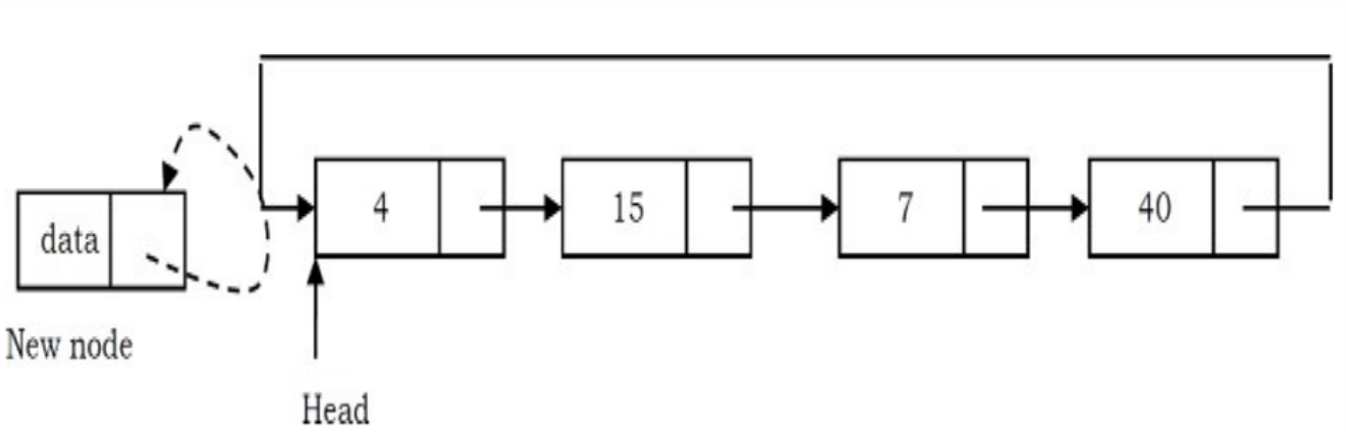


2.5 Circular Linked Lists Cont.

Inserting a Node at the front of a Circular Linked List

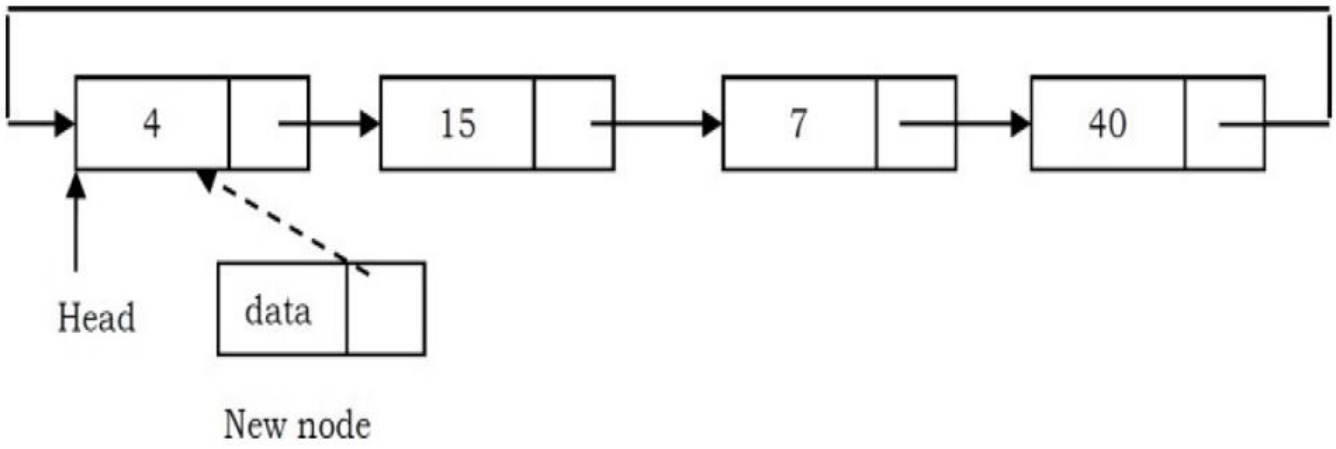
- The only difference between inserting a node at the beginning and at the end is that, after inserting the new node, we just need to update the pointer. The steps for doing this are given below:

- Create a new node and initially keep its next pointer pointing to itself.



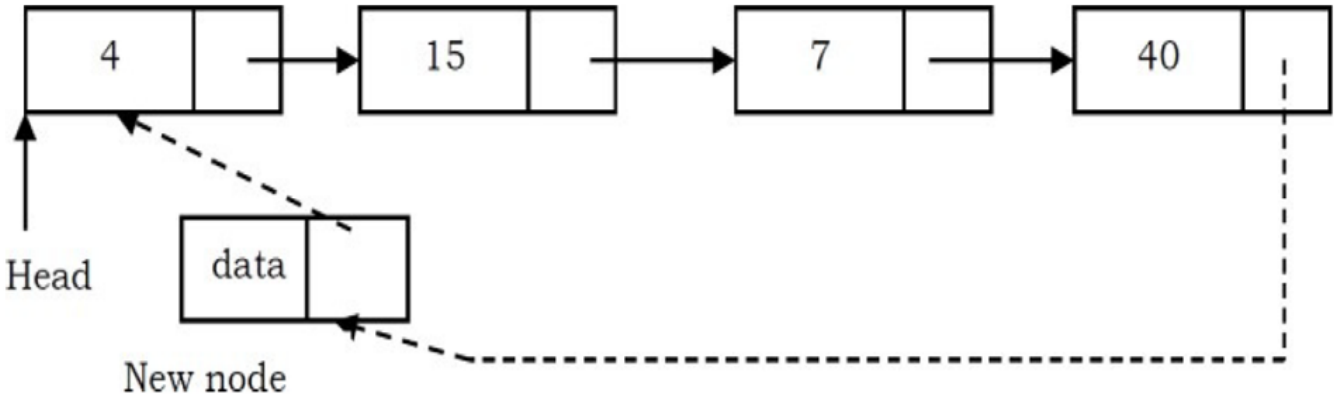
2.5 Circular Linked Lists Cont.

- Update the next pointer of the new node with the head node and also traverse the list until the tail. That means in a circular list we should stop at the node which is its previous node in the list.



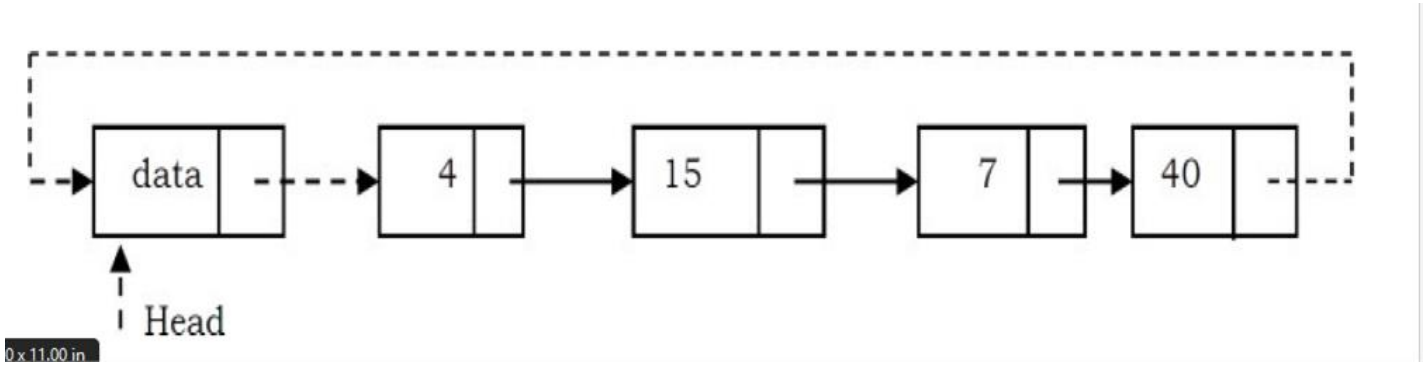
2.5 Circular Linked Lists Cont.

- Update the previous head node in the list to point to the new node.



2.5 Circular Linked Lists Cont.

- Make the new node as the head.

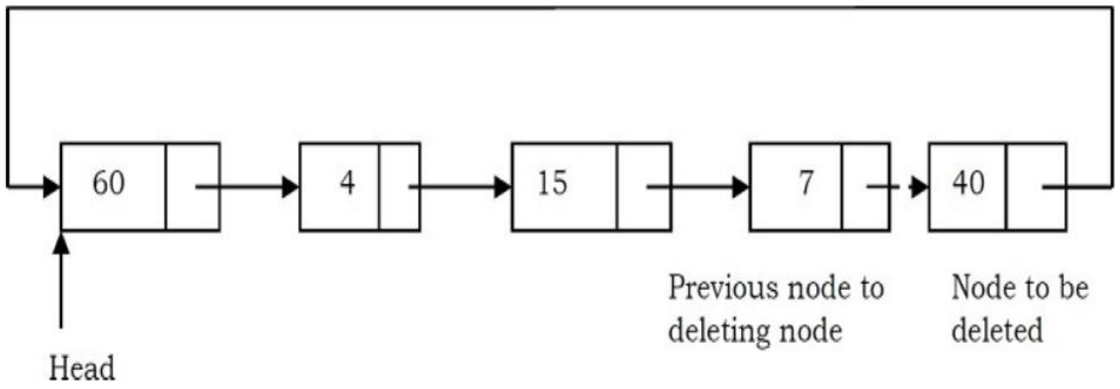


2.5 Circular Linked Lists Cont.

Deleting the Last Node in a Circular Linked List

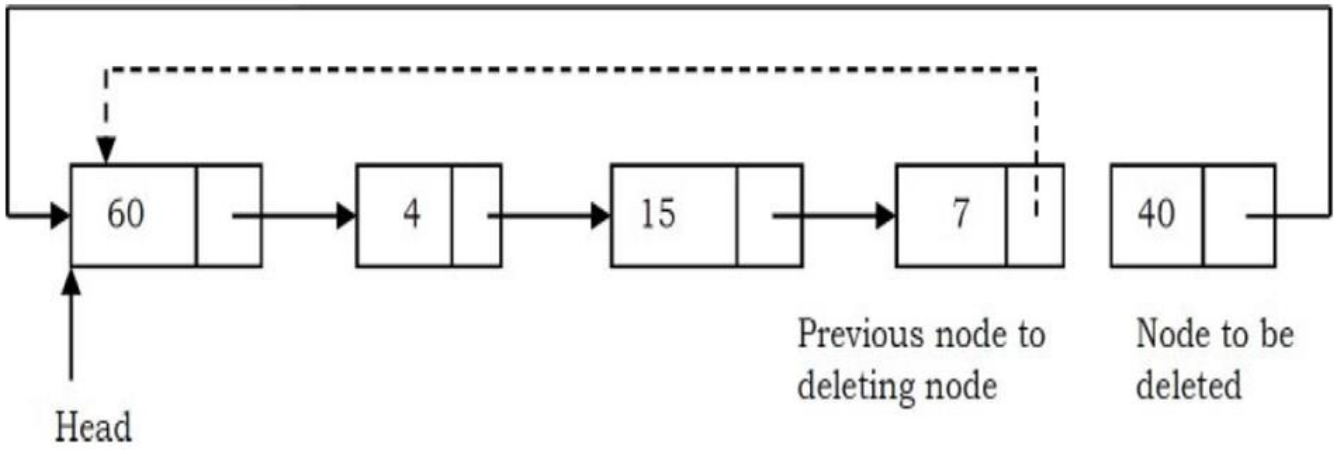
The list has to be traversed to reach the last but one node. This has to be named as the tail node,and its next field has to point to the first node. Consider the following list.

- To delete the last node 40, the list has to be traversed till you reach 7. The next field of 7 has to be changed to point to 60, and this node must be renamed.
- Traverse the list and find the tail node and its previous node.



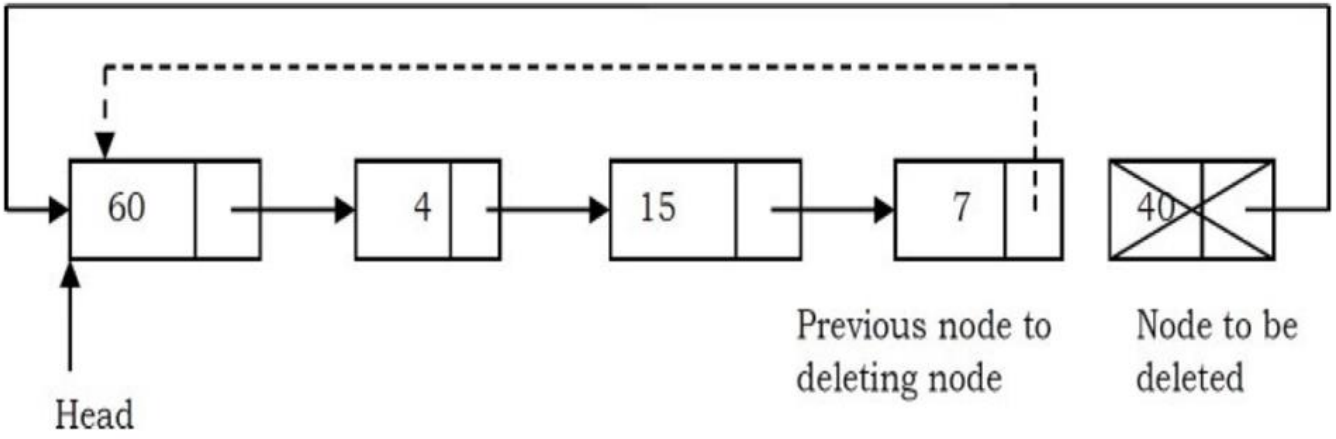
2.5 Circular Linked Lists Cont.

- Update the next pointer of tail node's previous node to point to head.



2.5 Circular Linked Lists Cont.

- Dispose of the tail node.

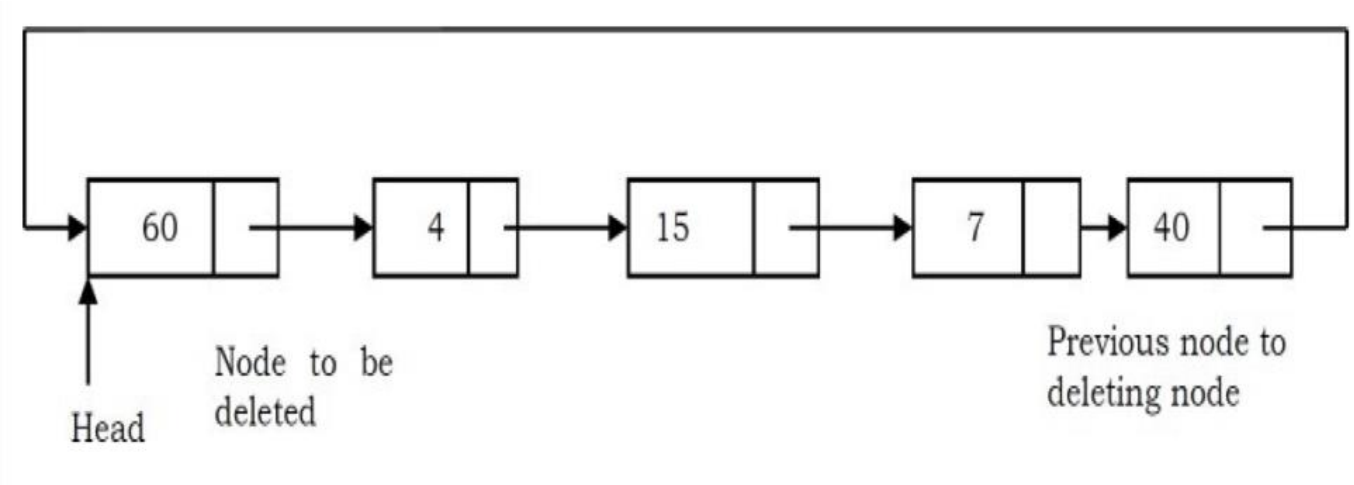


2.5 Circular Linked Lists Cont.

Deleting the First Node in a Circular Linked List

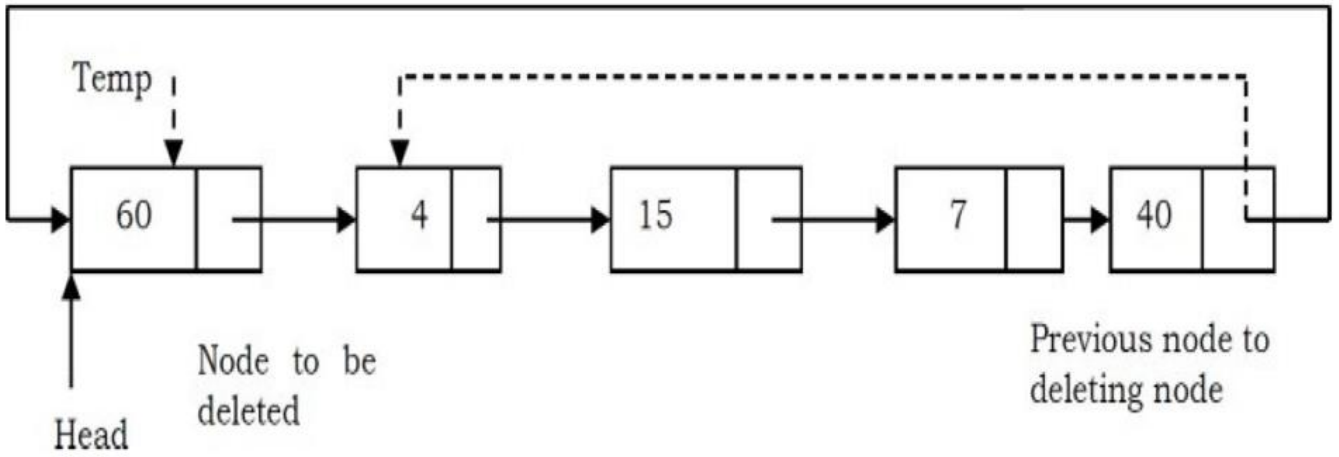
The first node can be deleted by simply replacing the next field of the tail node with the next field of the first node.

- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.



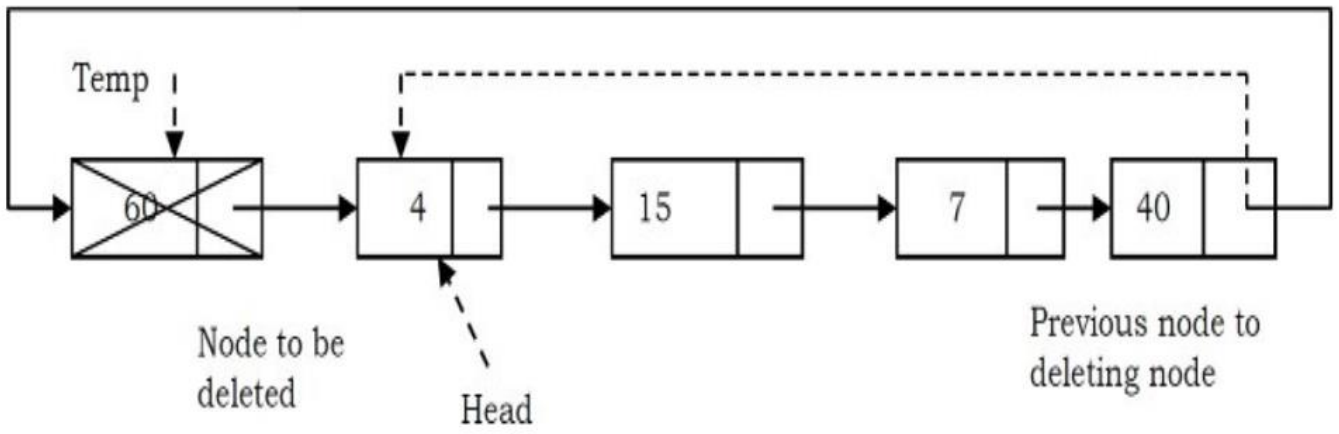
2.5 Circular Linked Lists Cont.

- Create a temporary node which will point to the head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



2.5 Circular Linked Lists Cont.

- Now, move the head pointer to next node. Create a temporary node which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



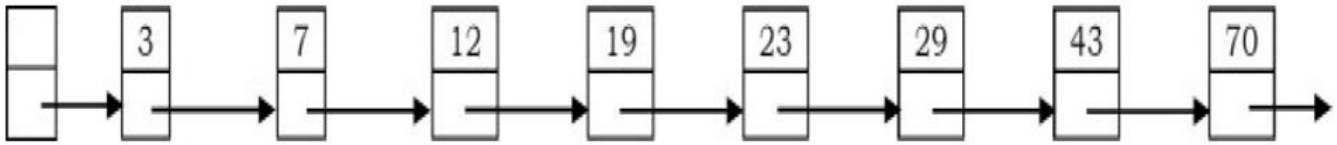
2.6 Skip Lists

- Skip lists are a probabilistic alternative to balanced trees.
- Skip list is a data structure that can be used as an alternative to balanced binary trees
- It is basically a linked list with additional pointers such that intermediate nodes can be skipped.
- The skip list is used to store a sorted list of elements or data with a linked list.
- It allows the process of the elements or data to view efficiently.
- It allows the user to search, remove, and insert the element very quickly.
- It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

2.6 Skip Lists Cont.

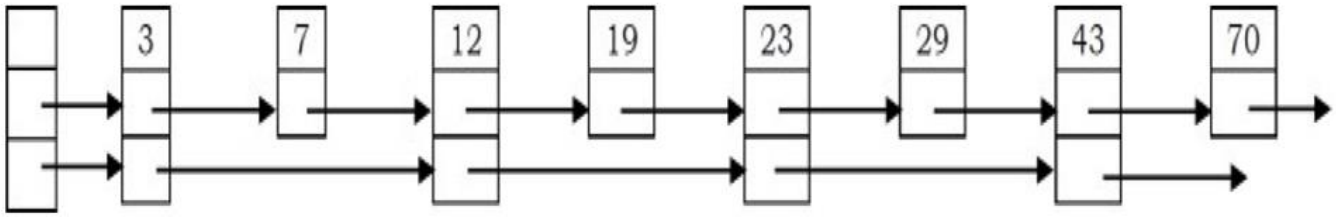
- In an ordinary sorted linked list, search, insert, and delete are in O(n) because the list must be scanned node-by-node from the head to find the relevant node.
- If somehow we could scan down the list in bigger steps (skip down, as it were), we would reduce the cost of scanning.
- This is the fundamental idea behind Skip Lists.

Skip Lists with One Level

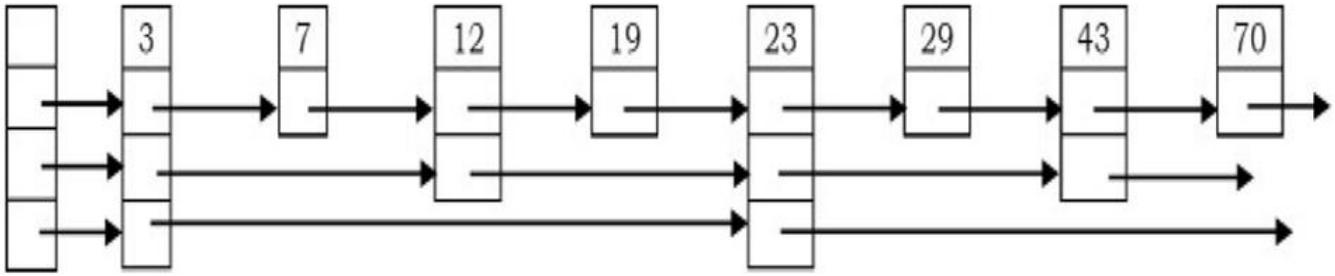


2.6 Skip Lists Cont.

Skip Lists with Two Levels



Skip Lists with Three Levels

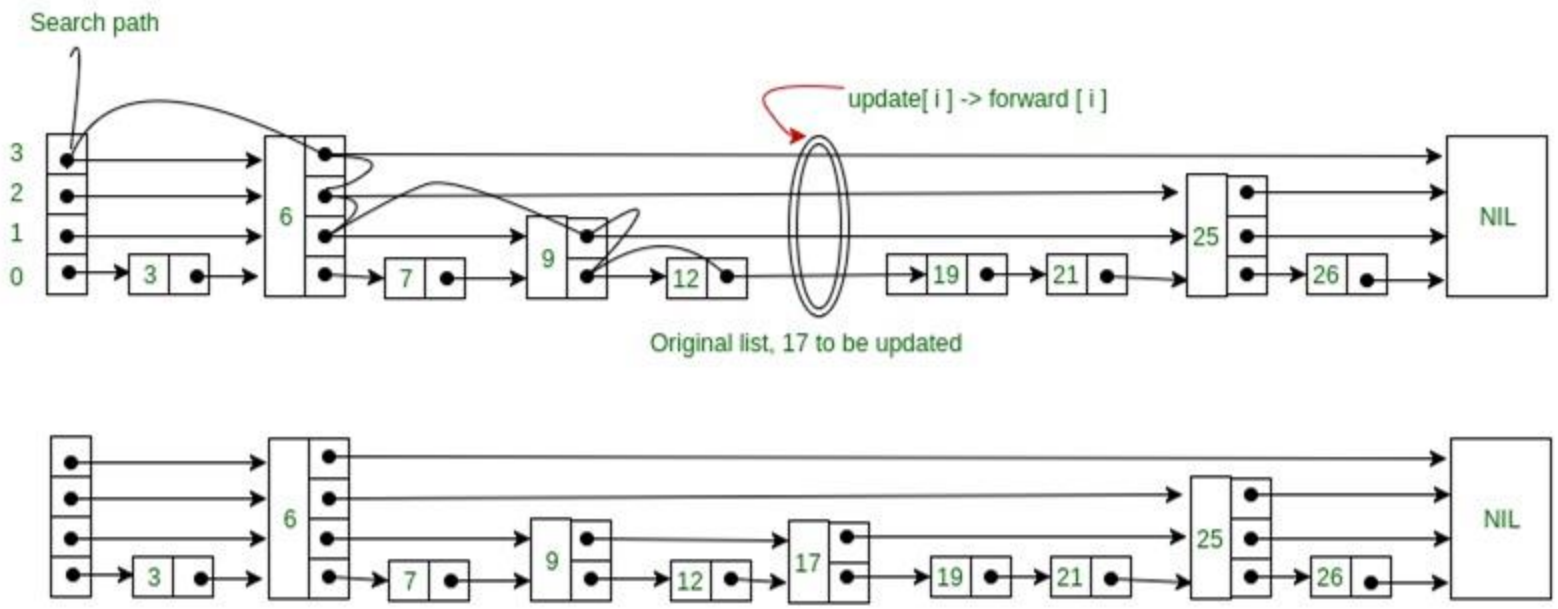


2.6 Skip Lists Cont.

- In a skip list of n nodes, for each k and i such that 1 <= k <= [log n] and 1 <= i <= [n/2k-1] - 1, the node in position 2^{k-1} • i points to the node in position 2^{k-1} • (i + 1).
- This means that every second node points to the node two positions ahead, every fourth node points to the node four positions ahead, and so on.
- This is accomplished by having different numbers of reference fields in nodes on the list:
  - Half of the nodes have just one reference field
  - one-fourth of the nodes have two reference fields
  - one-eighth of the nodes have three reference fields and etc.
- The number of reference fields indicates the level of each node, and the number of levels is maxLevel = [lg n] + 1.

2.6 Skip Lists Cont.

- **Insertion Operation in Skip Lists**



2.6 Skip Lists Cont.

- **Insertion Operation in Skip Lists**

We will start from highest level in the list and compare key of next node of the current node with the key to be inserted. Basic idea is If :

a. Key of next node is less than key to be inserted then we keep on moving forward on the same level
b. Key of next node is greater than the key to be inserted then we store the pointer to current node **i** at **update[i]** and move one level down and continue our search.

At the level 0, we will definitely find a position to insert given key.

2.6 Skip Lists Cont.

- **Insertion Operation in Skip Lists**

```
Insert(key)
    p = Search(key)
    q = null
    i = 1
    repeat
        i = i + 1                  //Height of tower for new element
        if i >= h
            h = h + 1
            createNewLevel()       //Creates new linked list level
```

2.6 Skip Lists Cont.

- **Insertion Operation in Skip Lists Cont.**

```
while (p.above == null)
    p = p.prev              //Scan backwards until you can go up
p = p.above
q = insertAfter(key, p)     //Insert our key after position p
until CoinFlip() == 'Tails'
n = n + 1
return q
```

We use a function called CoinFlip() that mimics a fair coin and returns either heads or tails. Finally, the function insertAfter(a, b) simply inserts the node a after the node b.

2.6 Skip Lists Cont.

- **Insertion Operation in Skip Lists Cont.**

  - First, we always insert the key into the bottom list at the correct location.
  - Then, we have to *promote* the new element. We do so by flipping a fair coin.
  - If it comes up heads, we promote the new element. By flipping this fair coin, we are essentially deciding how big to make the tower for the new element.
  - We scan backwards from our position until we can go up, and then we go up a level and insert our key right after our current position.
  - While we are flipping our coin, if the number of heads starts to grow larger than our current height, we have to make sure to create new levels in our skip list to accommodate this.

2.6 Skip Lists Cont.

- **Search Operation in Skip Lists**

Searching an element is very similar to approach for searching a spot for inserting an element in Skip list. The basic idea is if ：

1. Key of next node is less than search key then we keep on moving forward on the same level.
2. Key of next node is greater than the key to be inserted then we store the pointer to current node **i** at **update[i]** and move one level down and continue our search.

At the lowest level (0), if the element next to the rightmost element (update[0]) has key equal to the search key, then we have found key otherwise failure.

2.6 Skip Lists Cont.

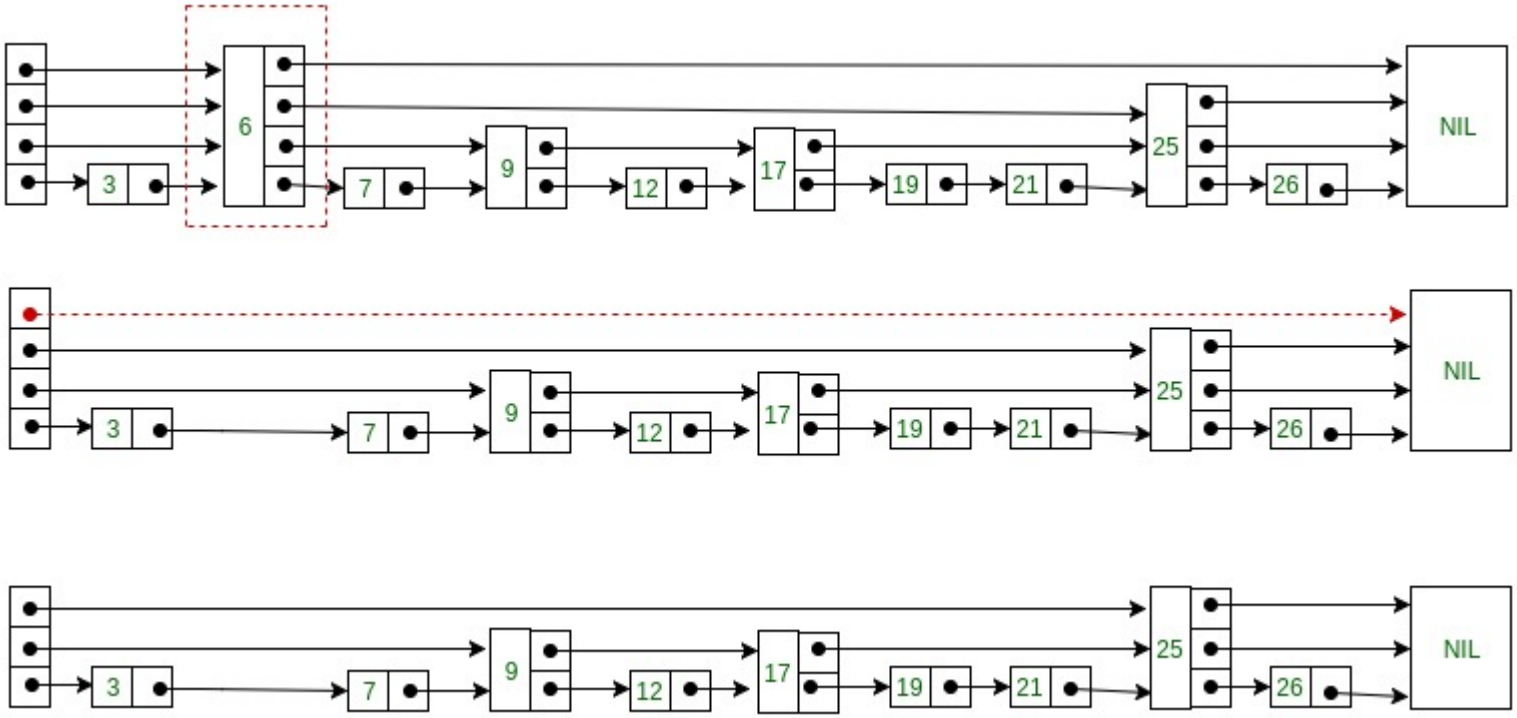
- **Search Operation in Skip Lists**

```
Search(key)
    p = top-left node in S
    while (p.below != null) do        //Scan down
        p = p.below
        while (key >= p.next) do      //Scan forward
            p = p.next

      return p
```

2.6 Skip Lists Cont.

- **Deletion Operation in Skip Lists**



2.6 Skip Lists Cont.

- **Deletion Operation in Skip Lists**

  - Once the element is located, rearrangement of pointers is done to remove element form list just like we do in singly linked list.
  - We start from lowest level and do rearrangement until element next to update[i] is not k.
  - After deletion of element there could be levels with no elements, so we will remove these levels as well by decrementing the level of Skip list.

2.6 Skip Lists Cont.

- **Deletion Operation in Skip Lists**

```
Delete(key)
    Search for all positions p_0, ..., p_i where key exists
    if none are found
        return
    Delete all positions p_0, ..., p_i
      Remove all empty layers of skip list
```

Delete can be implemented in many ways. Since we know when we find our first instance of key, it will be connected to all others instances of key, and we can easily delete them all at once.

Summary

Arrays	An array is a collection of elements of the same data type. The elements of an array are stored in consecutive memory locations and are referenced by an index.
Linked Lists	A linked list is a collection of data elements called nodes in which the linear representation is given by links from one node to the next node.
Doubly Linked Lists	A doubly linked list is a linked list which contains a pointer to the next as well as the previous node in the sequence. Therefore, it consists of three parts.