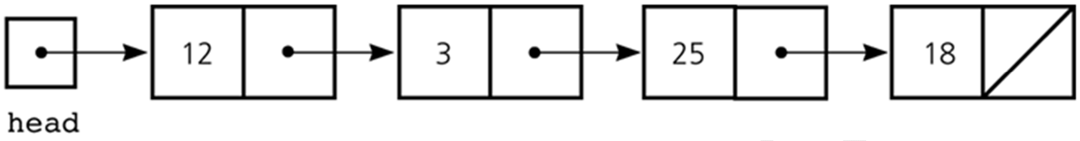
Linked List

Algorithm - abstract way to perform computation tasks

Data Structure - abstract way to organize information

Linked List:



Node:

Data

Next → **null**

Node code:

```
public class Node<T> {
        private T data;
        private Node<T> next;

        public Node(T data) { this.data = data;  }

        public void setData(T data) {  this.data = data;  }
        public T getData() {  return data;  }

        public Node<T> getNext() {  return next;  }
        public void setNext(Node<T> next) {  this.next = next;  }
}
```

Linked List Code:

```
public class LinkedList<T> {
        private Node<T> head;
}
```

Inserting a new node:

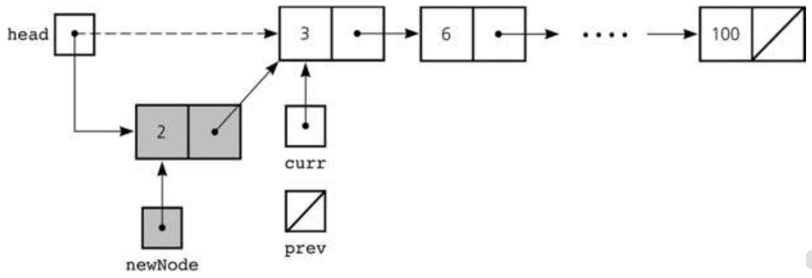
Inserting a Node into a Specified Position of a Linked List:

Three steps to insert a new node into a linked list

- Determine the point of insertion
- Create a new node and store the new data in it
- Connect the new node to the linked list by changing references

Case 1: To insert a node at the beginning of a linked list: (**curr == head**)

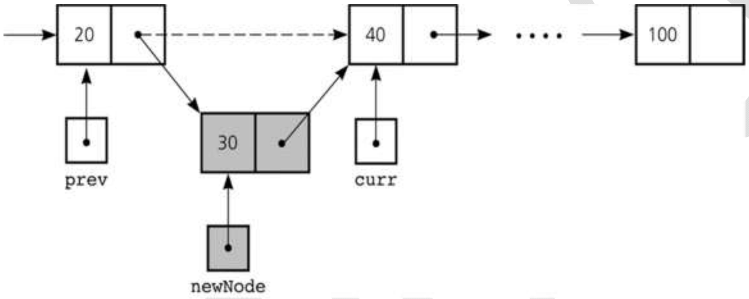
```
newNode.next = head;
head = newNode;
```



What's the time complexity of inserting an item to the head?? → **O(1)**

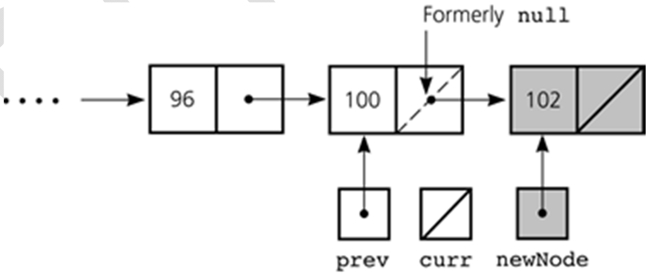
Case 2: To insert a node between two nodes:

```
newNode.next = curr;
prev.next = newNode;
```



Case 3: Inserting at the end of a linked list is a special case if **curr** is **null:**

```
newNode.next = curr;
prev.next = newNode;
```



Time Complexity → **O(n)**

H.W. → implement insert into a sorted linked list

Determining **curr** and **prev**

Determining the point of insertion or deletion for a sorted linked list of objects

```
for ( prev = null , curr = head;
      (curr != null)  &&  (newValue.compareTo(curr.item) > 0);
```

```
            prev = curr , curr = curr.next ) ;   // end for
```

Create a driver class to test linked list classes.
Override the ***toString*** methods first

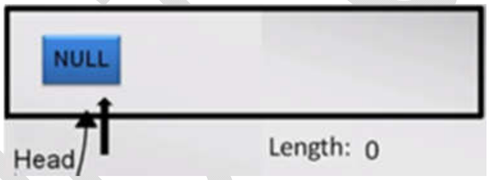
Node toString:

```
            @Override
            public String toString() {  return data.toString();  }
```

LinkedList toString:

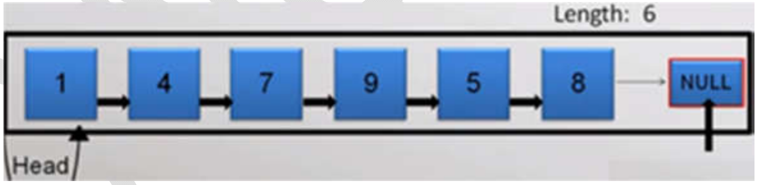
```
            @Override
            public String toString() {
                        String res = "➔";
                        Node<T> curr = head;
                        while (curr != null) {
                                    res += curr + "→ ";
                                    curr = curr.next;
                        }
                        return res + "NULL";
            }
```

Length of Linked List?



Case 1: If it's empty:

Case 2: If not: Make a pointer and move over all the nodes and maintain a **counter**

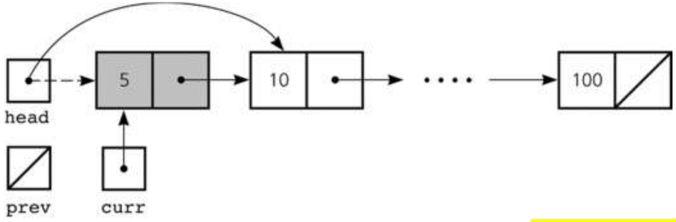


Length code: **Time Complexity ➔ O(n)**

```
public int length() {
            int length = 0;
            Node<T> curr = head;
            while (curr != null) {
                        length++;
                        curr = curr.next;
            }
            return length;
}
```

Deleting Nodes:

Case 1: Deleting the head node:



Simply move the **head** to the **head.next**: head = head.next;

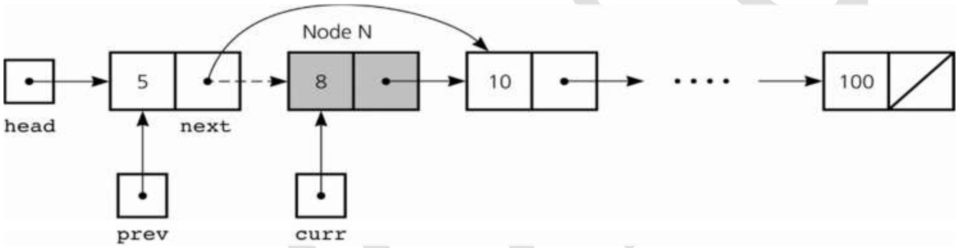
Now first Node has no reference to it ➔ **Garbage**

Time Complexity ➔ **O(1)**

Delete at head code: **// make sure linked list is not empty**

```
public Node<T> deleteAtStart() {
        Node<T> toDel =head;
        head = head.next;
        return toDel;
}
```

Case 2: Delete node **N** which **curr** references:



Set **next** in the node that precedes **N** to reference the node that follows **N**

prev.next = curr.next; // prev.next = prev.**next.next**;

Searching for an Item in a Linked List:

Time Complexity: linear growth ➔ **O(n)**

Find code:

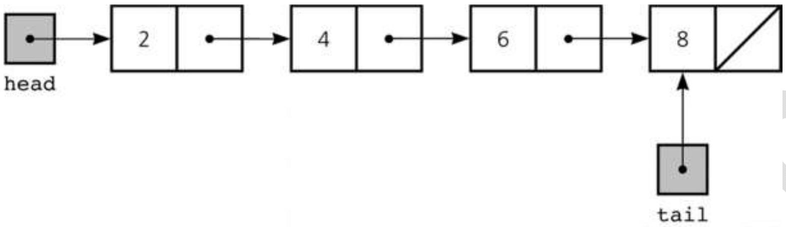
```
public Node<T> find(T data) {
        Node<T> curr =  head;
        while (curr != null) {
                if (curr.getData() == data)    // if (curr.getData().equals(data))
                        return curr;
                curr = curr.next;
        }
        return null;
}
```

Variations of the Linked List:

1- Tail References (Doubly Ended Linked List)

- Remembers where the end of the linked list is.
- Therefore, we can add and delete at both ends.
- To add a node to the end of a linked list

`tail.next = new Node(request, null);`



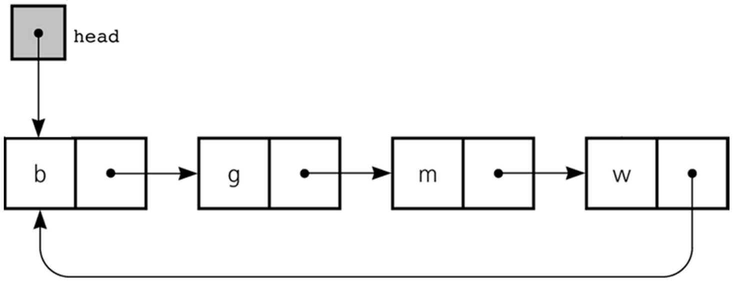
```
public class DoubleEndedList<T> extends LinkedList<T> {
        private Node<T> tail;
        public Node<T> getTail() {        return tail;   }

        public void addAtEnd(T data) {
                Node<T> newNode = new Node<T>(data);
                 if (head == null) {  // empty
                        head = newNode;
                        tail = newNode;
                 }
                        else {
                        tail.setNext(newNode);
                        tail = newNode;
                 }
        }
}
```

Make sure to override addAtStart to set the tail pointer correctly:

```
@Override
public void addAtStart(T data) {
    Node<T> newNode = new Node<T>(data);
    if (head == null) {  // empty
      head = newNode;
      tail = newNode;
    }
    else{
      newNode.setNext(head);
      head = newNode;
    }
}
```

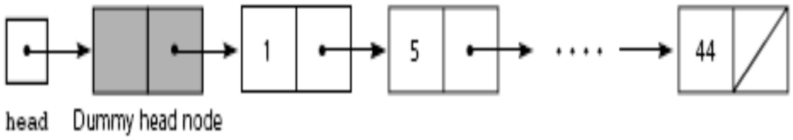
2- Circular Linked List

- Last node references the first node
- Every node has a successor



3- Dummy Head Nodes

- Always present, even when the linked list is empty
- Insertion and deletion algorithms initialize **prev** to reference the dummy head node, rather than **null**



Processing Linked Lists Recursively:

- **Traversal**
  - Recursive strategy to display a list
    Write the first node of the list
    Write the list minus its first node

```
public static void traversList(Node curr) {
   if(curr == null)
      System.out.println("NULL");
   else {
       System.out.print("[" + curr + "]-->");
       traversList(curr.next);
   }
}
```

  - Recursive strategies to display a list backward
    - writeListBackward strategy
      Write the last node of the list
      Write the list minus its last node backward

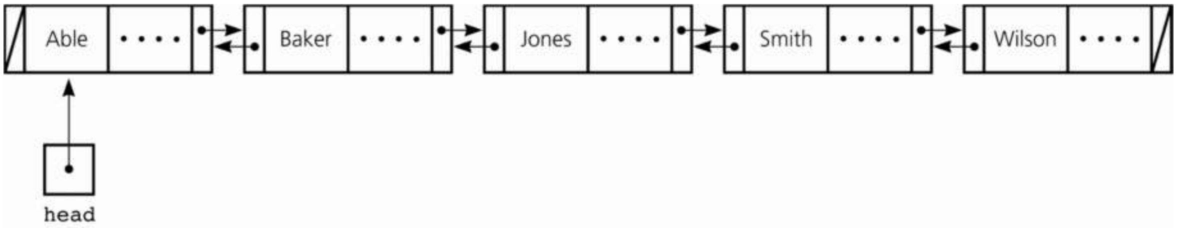
```
public static void traversListBackward(Node curr) {
   if(curr == null)
      System.out.print("NULL");
   else {
      traversListBackward(curr.next);
      System.out.print("<--[" + curr + "]");
   }
}
```

Doubly Linked List

Node:

Date
Next → null
null← Prev

Doubly Linked List: Each node references both its predecessor and its successor:



Doubly Node Code:

```
public class DNode <T extends Comparable<T>>{
   T data;
   DNode next;
   DNode prev;

   public DNode(T data) { this.data = data; }
   public T getData() { return data;  }
   public DNode getNext () { return next; }
   public DNode getPrev () { return prev; }

   public void setNext(DNode next) {   this.next = next; }
   public void setPrev(DNode prev) {   this.prev = prev; }
   public String toString() { return this.data.toString();  }
}
```

Doubly Linked List code:

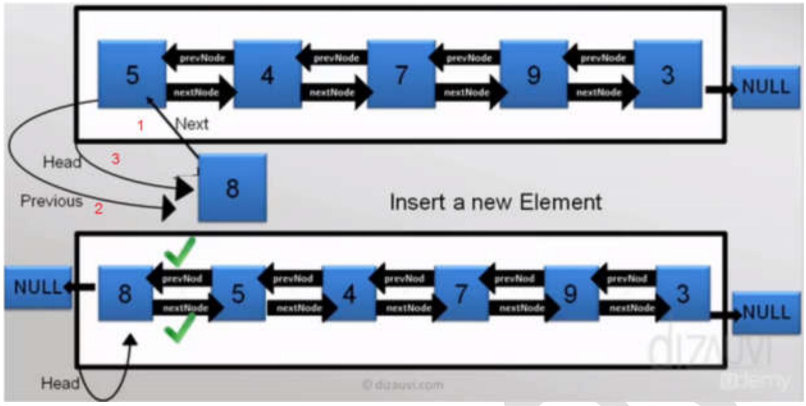
```
public class DLinkedList <T extends Comparable<T>>{
   DNode head;
}
```

Override toString method code:

```
public String toString() {
   String res = "Head-->";
   DNode<T> curr = this.head;
   while (curr != null) {
      res += "["+curr + "]";
      curr = curr.getNext();
      if(curr!=null)
         res +="<=>";
   }
   return res + "-->NULL";
}
```

Insert a new node (not sorted)

Case 1: Insert at head:



```
public void insertAtHead(T data) {
   DNode<T> newNode = new DNode(data);
   if(head==null) // empty linkedlist
      head = newNode;
   else {
      newNode.setNext(this.head);
      head.setPrev(newNode);
      head = newNode;
   }
}
```

Case 2: Insert at end:

Student Activity: insert at last

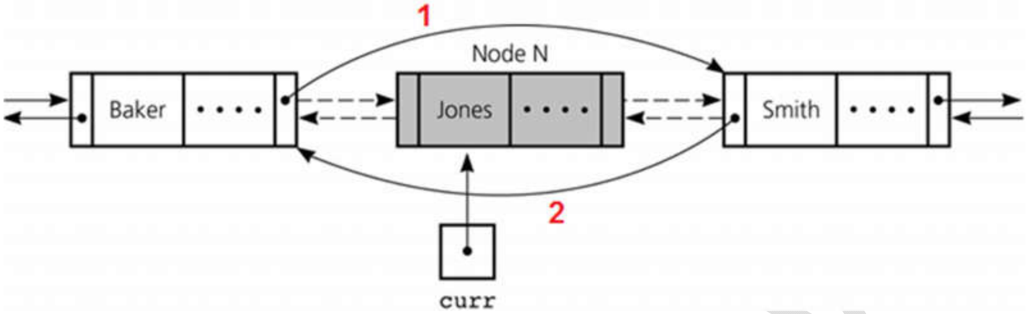
```
public void insertAtEnd(T data) {
   DNode<T> newNode = new DNode(data);
   if (head == null) // empty linkedlist
      head = newNode;
   else {  // find last node
      DNode<T> last = head;
      while(last.getNext() != null)
         last = last.getNext();
      last.setNext(newNode);
      newNode.setPrev(last);
   }
}
```

Length of a doubly linked list code:

```
public int length() {
   int length = 0;
   DNode<T> curr = this.head;
   while (curr != null) {
      length++;
      curr = curr.getNext();
   }
   return length;
}
```

Delete a node:

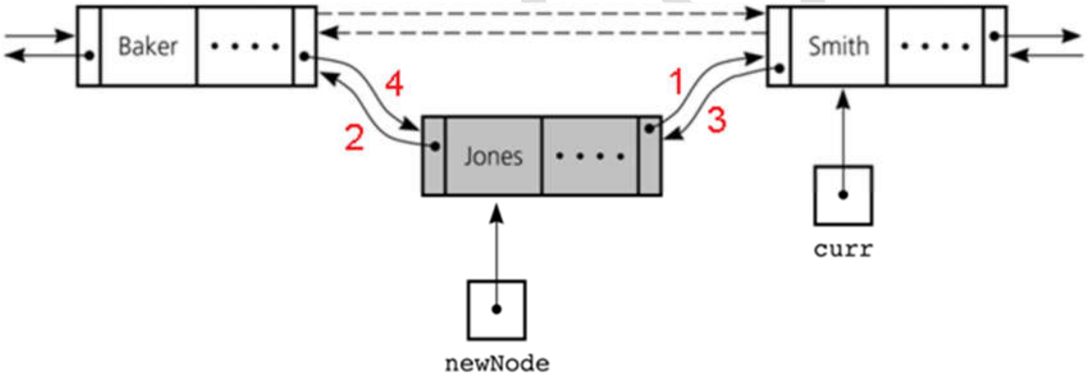
- To delete the node that **curr** pointer references



```
curr.prev.next = curr.next;
curr.next.prev = curr.prev;
```

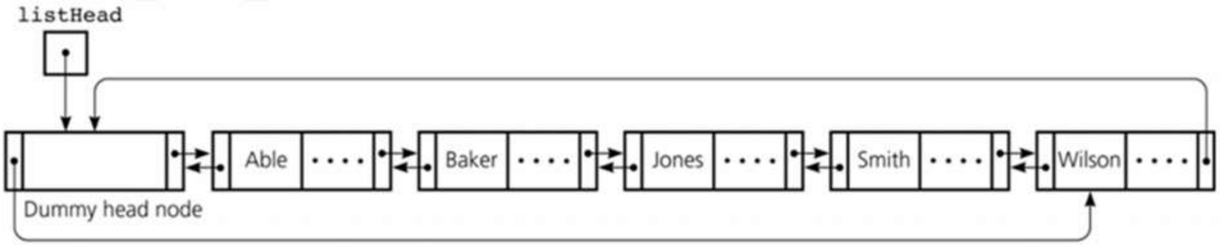
Insert a new Node (Sorted):

- To insert a new node that **newNode** references before the node referenced by **curr**



```
newNode.next = curr;            // 1
newNode.prev = curr.prev;       // 2
curr.prev = newNode;            // 3
newNode.prev.next = newNode;    // 4
```

Circular doubly linked list with dummy head:

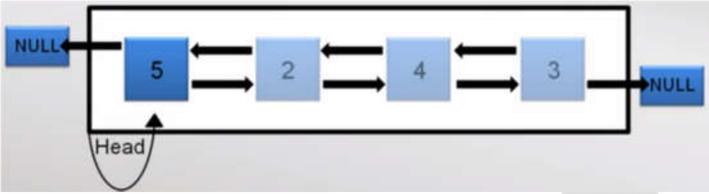


- Preceding reference of the dummy head node references the last node.
- next reference of the last node references the dummy head node.
- **Eliminates special cases for insertions and deletions.**

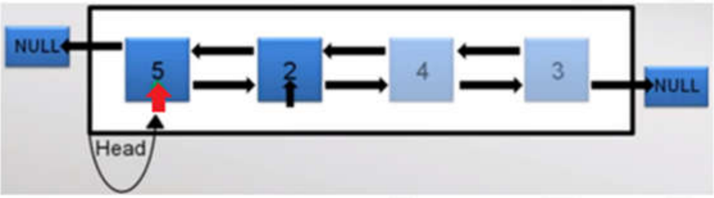
Case Study: Insertion Sort using doubly linked list (Using NO extra space):

Review insertion sort logic and point to problem of insertion and time needed to shift the items
Worst case if the array is reverse sorted

Example: assume we need to sort the following doubly linked list:



Assumption: 1st node is sorted. We start from the 2nd element:



Here:

- The **black** pointer points to the **current** node to be sorted.
- The **red** pointer points to previous node of **current** node to be sorted.
- The **green** pointer points to next node of **current** node to be sorted.

Step 1: The **red** pointer keeps move backward until it reaches a node which has a value **smaller** than the **current** node **or** reach **NULL**.

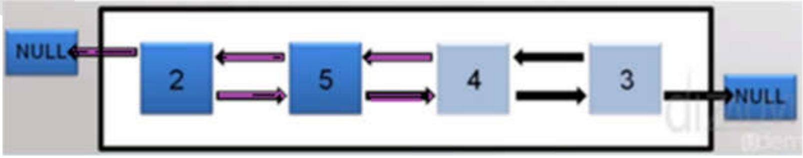
Step 2: the **current** item will be inserted after **red** pointer as follow:

Make sure you maintain references correctly.

To do so draw the expected outcome and follow the steps to change the pointers:

Initial state:

Final state:

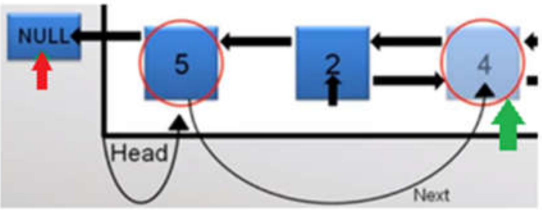


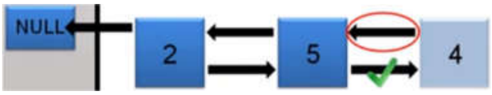
Case 1: insert to head

Step 2.0: **make new green pointer = black.next**

Step 2.1:

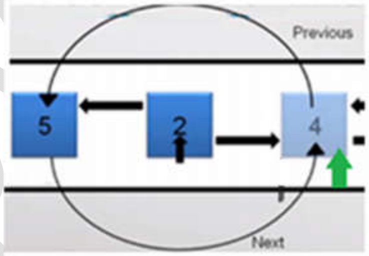
➔ **black.prev.next = green**

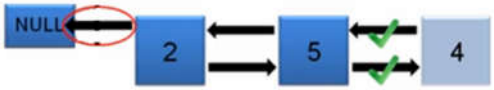




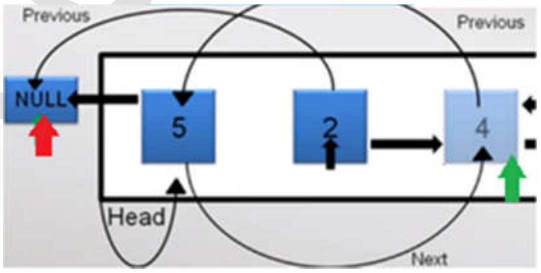
Step 2.2: ➔

if (green != null) green.prev = black.prev



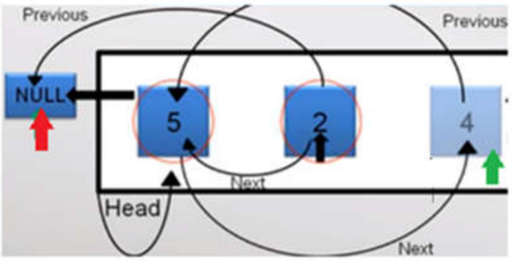


Step 2.3: ➔ **black.prev = red**



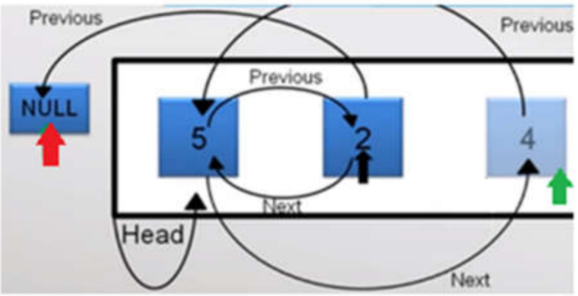
Step 2.4: ➔

```
if(red==null)  black.next = black.next.prev
else           black.next =  red.next
```



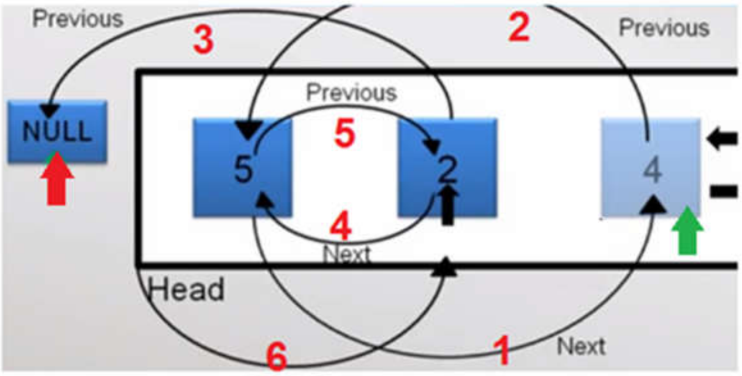
Step 2.5: ➔

```
If (red == null)  black.next.prev = black
else              red.next.prev = black
```



Step 2.6:

```
if (red == NULL )    head = black
else                 red.next = black;
```

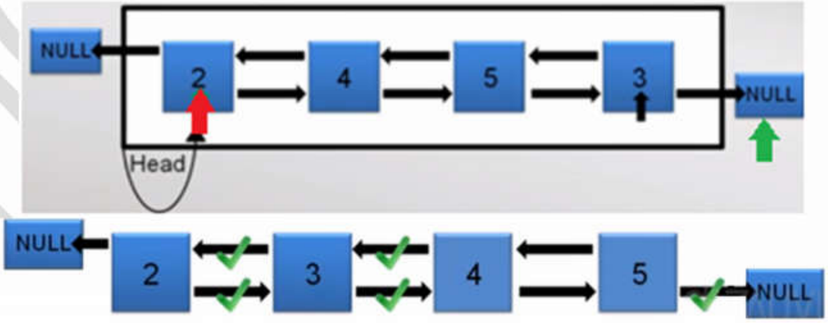


Step 2.7: **black = green**

Case 2: insert 4 in the middle

Practice yourself

Case 3: insert last element



Insertion Sort Code:

```
// Insertion Sort of a Doubly Linked List
public void sort() {
   DNode black = head.next;
   while (black != null) {
      DNode red = black.prev;
      while (red != null && (red.data.compareTo(black.data) > 0 ))  // step 1.0
         red = red.prev;

      DNode green = black.next;               // step 2.0
      if (red != null || (head != black)) {
         black.prev.next = green;             // step 2.1
         if (green!= null) {
            green.prev = black.prev;          // step 2.2
         }
         black.prev = red;                    // step 2.3
      }
      if (red == null) {    // set the black as head
         if (head != black) {
            black.next = head;                // step 2.4
            black.next.prev = black;          // step 2.5
            head = black;                     // step 2.6
         }
      } else {   // red is not null
         black.next = red.next;               // step 2.4
         red.next.prev = black;               // step 2.5
         red.next = black;                    // step 2.6
      }
      black = green;  // step 2.7
   }
}
```