Single Linked Lists

COSC 1P03 ❄ Lecture 03 ❄ Tuesday May 14, 2024[1]

Maysara Al Jumaily
amaysara@brocku.ca

Table of Content

3 Single Linked Lists **1**
3.1 Overview . . . 1
3.2 The `print` Method . . . 4
3.3 `insertAtFront` Method . . . 5
3.4 `removeAtFront` Method . . . 7
3.5 `insertAtRearOnePointer` Method . . . 8
3.6 `insertAtRearTwoPointers` Method . . . 10
3.7 `sortedInsertion` Method . . . 11

3 Single Linked Lists

We have learned about arrays, which are a fixed size data structure that allows fast retrieval (*i.e.*, the access of an element) but cannot change its size. Linked lists is a collection of nodes, where each node points to the next, and is a dynamic data structure that allows for the insertion and removal of nodes.

3.1 Overview

A *linkedlist* is a dynamic data structure that stores `Node`s objects where each node holds the element and a reference to the next node. Linkedlists are dynamic in the sense that you are able to add insert/delete nodes to/from the list. It is visualized as: Accessing elements

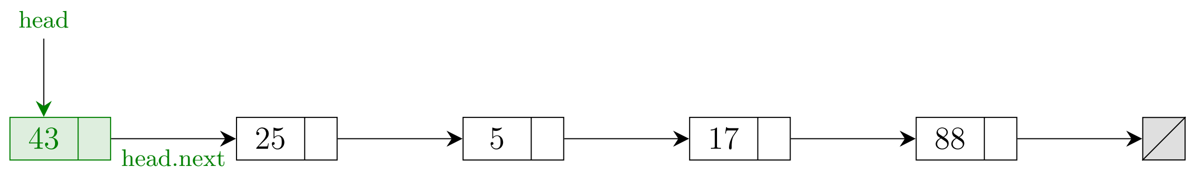


Figure 3.1: A linked list containing five elements `int`egers

must always begin from the `head` (*i.e.*, far-left element). Suppose that we want to access element `75` that is found at the very end of the linked list, then we **must** start from `head` which is `63`, then go to `02`, then `24`, then `31` *then* `75`. This means that, in general, if we had n elements in the linked list and wanted to find some element, then it would be $\mathcal{O}(n)$. The `Node` class is given as:

[1]Complied at: Tuesday May 14th, 2024 8:58am -04:00

```
public class Node{
  public int item;//the data to store
  public Node next;//the reference to the next node
  public Node(int itemPassed, Node nextNodePassed){
    item = itemPassed;
    next = nextNodePassed;
  }
}
```

We can use the following to create an instance of a `Node` class:

```
Node n1 = new Node(1, null);
```

which can be visually represented as:

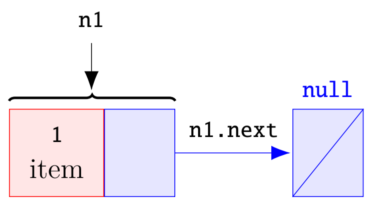
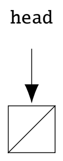


Figure 3.2: `n1` is a `Node` that stores `1` as the item and `null` as its next node.

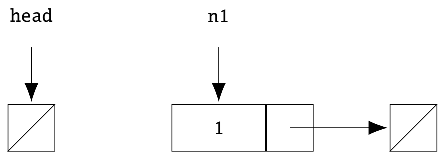
To create a linkedlist, declare an **instance** variable (~~not a local variable~~) of type `Node` denoting the beginning of the list. Usually, this is called `head` or `list`, we will use `head` for the rest of the lecture note. Now, `head` will be initialized to `null` at the beginning. To insert the **first** node, create an instance of `Node` and reinitialize head to that node:

```
public class Test{
  private Node head;//the head of the linkedlist
  public Test(){
    head = null;//no elements in the linkedlist
    //create an instance of Node and make head point to it:
    Node n1 = new Node (1, null);//node storing 1 and points to null
    head = n1;//ensure head is pointing to the first node
  }
}
```

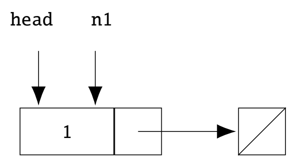
- At first, we have `list` pointing to `null`:



- We then created a local variable `n1` that store `1` as item and `null` as next:



- We linked them using `head = n1;`
  - This is read right-to-left as: "store `n1` in `head`".



- Once we exist the method/constructor, the local variable `n1` will be garbage collected. Only `head` (instance variable) will point to the node.

We will create another class, say `LinkedList.java`, to create the linked list. We will have the following operations:

- Printing the elements in the linked list (`print`)
- Insertion at the front which returns the node removed (`insertAtFront`)
- Removal at the front (`removeAtFront`)
- Insertion at the middle/rear
  - Using a single pointer `p` (`insertOnePointer`)
  - Using a two pointers `q` and `p` (`insertTwoPointers`)
- Removal at the middle/rear which returns the node removed
  - Using a single pointer `p` (`removeOnePointer`)
  - Using a two pointers `q` and `p` (`removeTwoPointers`)

The first thing to have is that our linked list needs to be stored inside of an instance variable. This is a must! Otherwise, it will not work (because of garbage collection). We will refer to our linked list as `head`. Let us write the skeleton of the `LinkedList.java` class and then go deeper into each method.

```java
public class LinkedList{
  private Node head;

  public LinkedList(){
    head = null;
  }

  public void print(){ ... }

  public void insertAtFront(int value){ ... }

  public Node removeAtFront(int valueToRemove){ ... }

  public void insertAtRearOnePointer(int value){ ... }

  public void insertAtRearTwoPointers(int value){ ... }

  public void sortedInsertion(int value){ ... }


}
```

3.2 The `print` Method

Traversing method

The print method is simple to code. It will require that we use a temporary variable `p` that starts at `head` and loops through all of the nodes until we reach `null`. The `System.out.print()` and `System.out.println()` are there to beautify the output.

```java
public void print() {
  Node p = head;
  while (p != null) {
      System.out.print(p.item + " -> ");
      p = p.next;
  }
  System.out.println("null");
}
```

Suppose that we have the linked list found in Figure 3.3 that needs to be printed out. The output will be

```
1 -> 2 -> 3 -> 4 -> 5 -> null
```

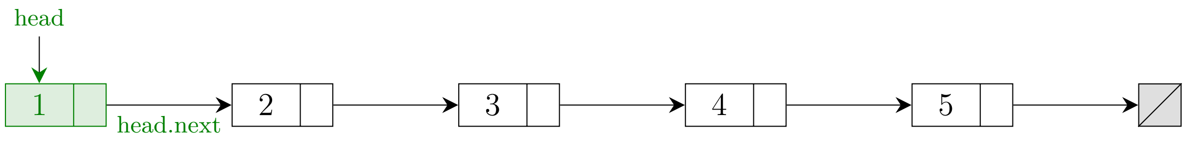
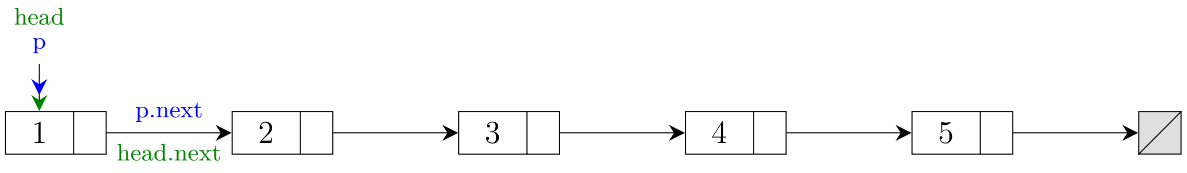


Figure 3.3: A linked list containing five elements `int`egers

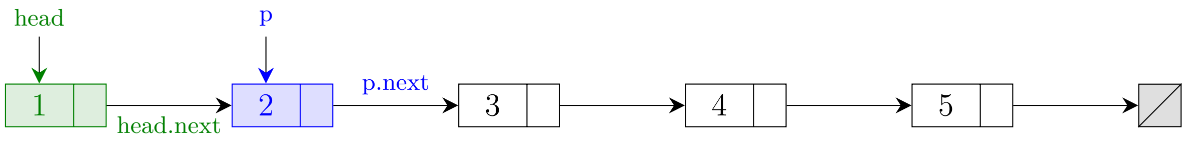
The steps that will be taken are as follows:

Step 1:



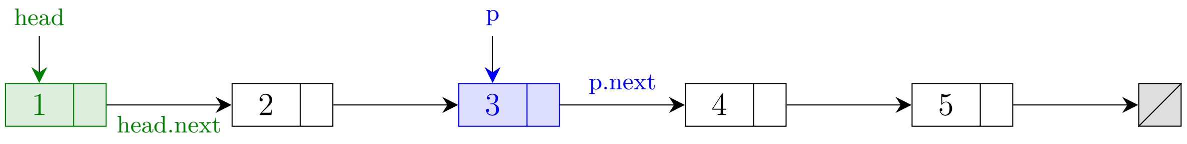
Output: `1 ->`

Step 2:



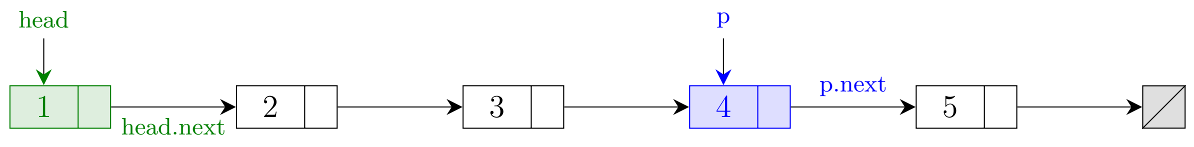
Output: `1 -> 2 ->`

Step 3:



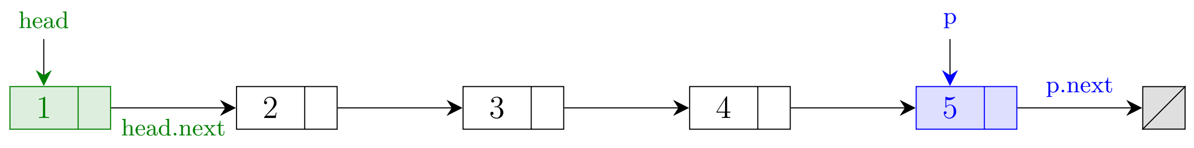
Output: `1 -> 2 -> 3 ->`

Step 4:



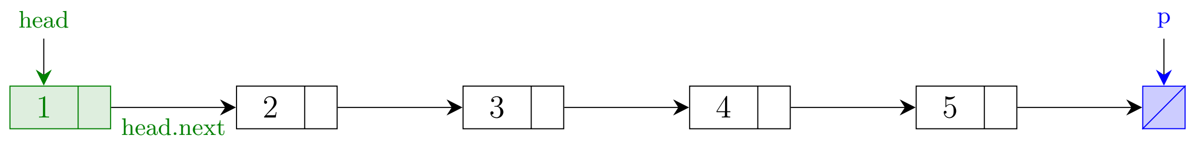
Output: `1 -> 2 -> 3 -> 4 ->`

Step 5:



Output: `1 -> 2 -> 3 -> 4 -> 5 ->`

Step 6:



Output: `1 -> 2 -> 3 -> 4 -> 5 -> null`

Since we `p` is `null`, then we don't enter the **`while`** loop and stop looping. In general, when we have n elements, the time complexity will be $\mathcal{O}(n)$.

3.3 `insertAtFront` Method

The insert a node at the front of the list, it will take $\mathcal{O}(1)$ as we will not loop through the nodes, but insert at the front and update the head to point to the new created node. The first variation of the code is written below.

```java
public void insertAtFront(int value){
  Node n = new Node(value, null);//create a new node that is not in list
  n.next = head; //make it point to the first node in linked list
  head = n; //make head point to the newly created node (i.e., far-left node)
}
```

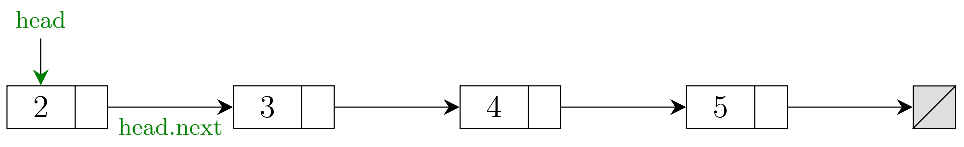
This above will work even if we have a `null` list (*i.e.*, `head` is `null`). There are two variations to this method. Since we know we will point to head, why do we create the new node and make point to `head` instead of `null`. Here is the second variation:

```java
public void insertAtFront(int value){
  Node n = new Node(value, head); //create a new node that points to list
  head = n; //make head point to the newly created node (i.e., far-left node)
}
```

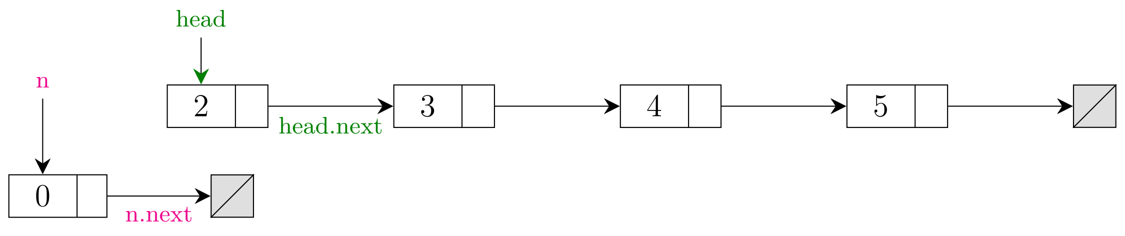
We could make another variation. Since we are going to make head become n, why not make head become the newly created node? Here is the code of the third variation:

```java
public void insertAtFront(int value){
  //create node and ensure it points to head, then move head to point to new node
  head = new Node(value, head);
}
```

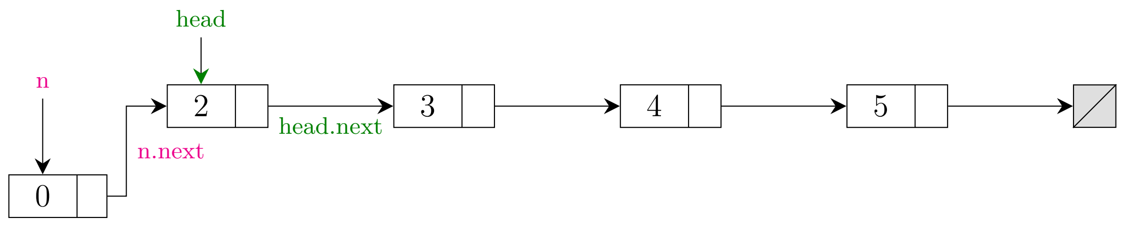
All the three above variations are equivalent. This means that all of the three variations will perform the same thing, which is to insert at the front. To see the logic precisely, we will use the first variation for illustration. Suppose we have the following linked list to insert at the front of and the call was to insert the value 0, *i.e.*, `insertAtFront(0);`:



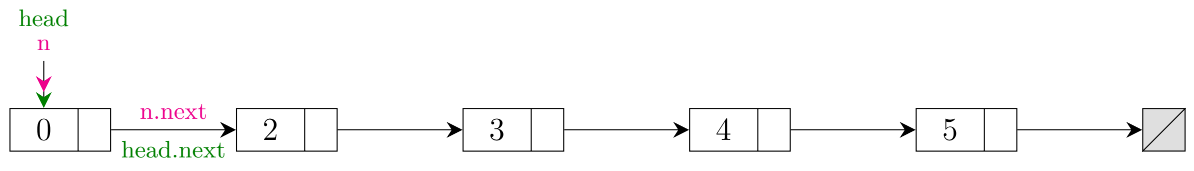
The line
`Node n = new Node(value, null);`
will generate the following where the new node is not a part of the linked list:



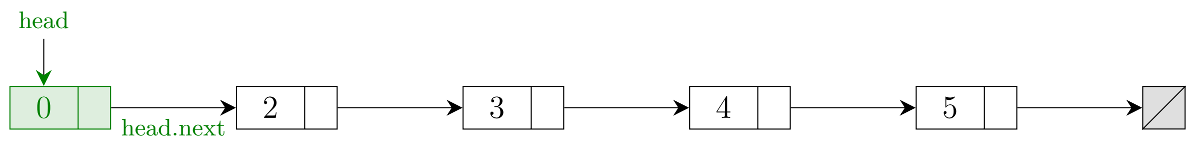
Now, the line
`n.next = head;`
will ensure the new node points to the head node, like so:



Note that our linked list is not complete yet. We must have `head` to point to the far-left node in the list. We need to move it to point to the same node n is pointing to. This is achieved by
`head = n;`
We will get



The temporary variable `n` will be removed via garbage collection once we finishing executing the method and our linked list will look like:



3.4 `removeAtFront` Method

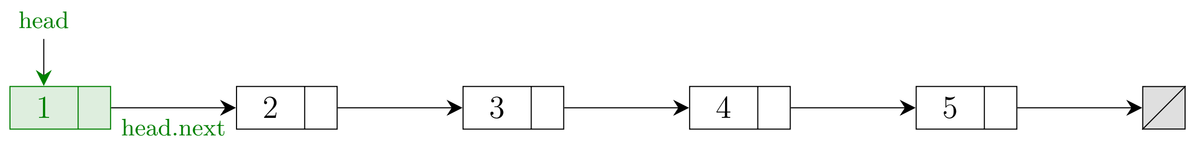
The objective of this method is to remove the first element of the linked list. It will also return the node that will be deleted. It will take $\mathcal{O}(1)$ in terms of time complexity because we don't loop through the elements. Everything is taking action at the beginning of the list. We could call it like so:

```
Node removed = removeAtFront();
System.out.println("The value removed is: " + removed.item);
```

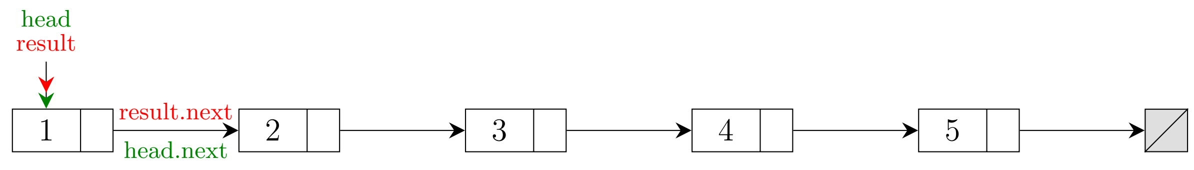
The code will be to store the node that will be removed, remove it from the linked list and then return that stored node. It is a good habit for you, a first-year student, to use the convention of naming the variable `result` to denote the object to be returned. It could be possible that we have a null list to begin with The code will be

```
public Node removeAtFront(){
  Node result;
  result = head;//store the node to remove
  if(head != null){//we have at least one node in the linked list
    head = head.next;//make head point to the adjacent node to the right
  } else {
    //we have null list, maybe crash the program or print out "Error"
  }
  return result;
}
```

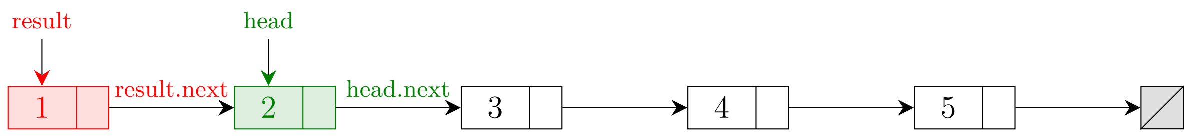
In case we have a `null` list (*i.e.*, `head` is `null`), then the following: `result = head;`, is the same as `result = null;`, which is valid. Suppose we have the following linked list:



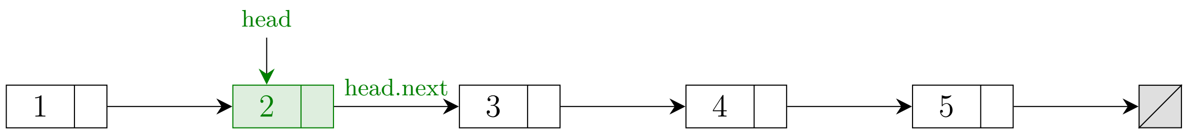
We will have to first execute the line:
`result = head;`
which has the following effect:



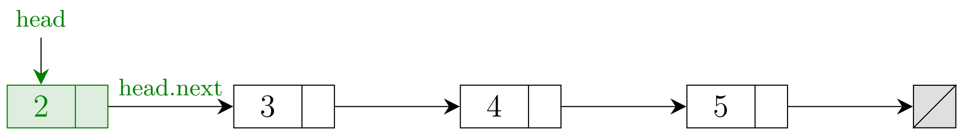
The next line,
`head = head.next;`
will transform the linked list into the following:



We will return the red node which contains the value 1. Now the more important point is that the variable `result` *must* be local variable for it to be removed. Once we leave the method, we will have the following:



To ensure a node is deleted, **NOTING MUST BE POINTING TO IT**. The node could point to other nodes, that is fine, but nothing points to it. Since this is the case, garbage collection will get rid of it. We would have the following linked list after completing the method execution:



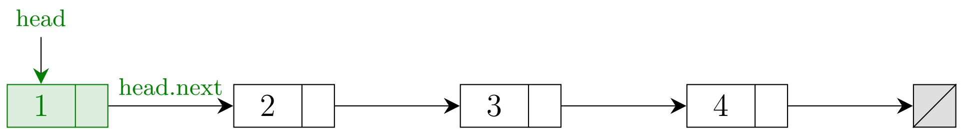
Now, we are able to call the method in the constructor and print out the removed value, as mentioned in the beginning of subsection 3.4.

3.5 `insertAtRearOnePointer` Method

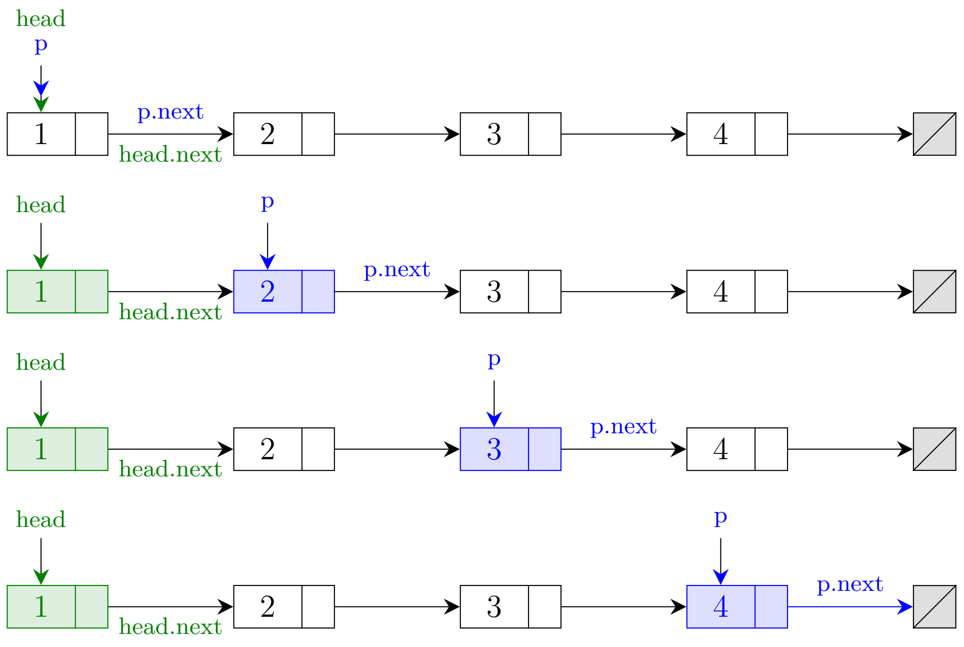
We are able to use one pointer to perform insertion at the rear. The objective is to ensure the pointer `p` points to the last valid node (*i.e.*, the node that points to `null`). We will move our pointer `p` as long as its next not equal to null. The logic is as follows:

```java
public void insertAtRearOnePointer(int value){
  if(head == null){// empty list, just insert
    head = new Node(value, null);// or  head = new Node(value, head);
  } else {// there is at least one node in the list
    Node p = head;
    while(p.next != null){//ensure p points to last valid node
      p = p.next;
    }
    p.next = new Node(value, null);// make the horizontal arrow of last valid
                                   // node point to new node created
  }
}
```

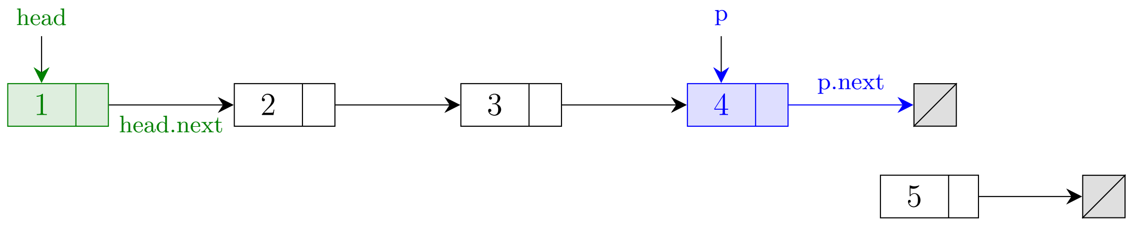
Let us suppose we have multiple elements in the list, as shown:



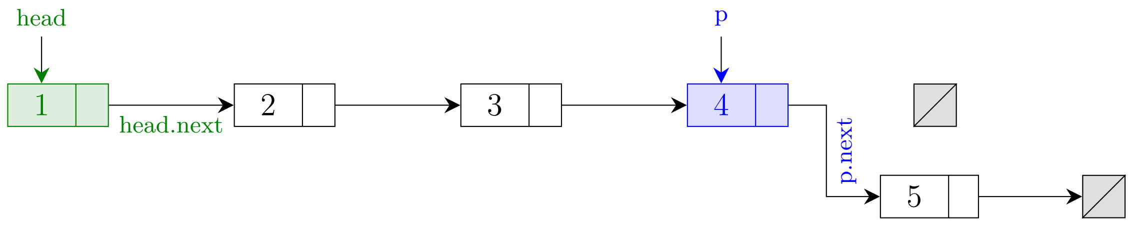
The pointer `p` will move as such:



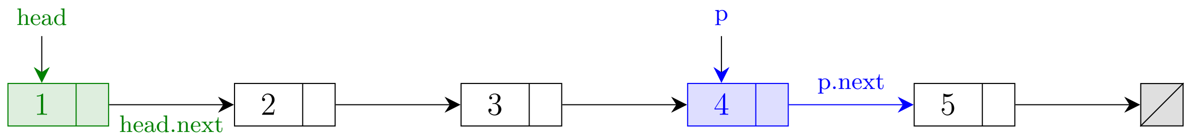
Now, focus on the horizontal arrow in blue, it needs to connect with the new node. Suppose we will insert the value `5` by the following method call:
`insertAtRearOnePointer(5);`
After moving the pointer `p` to the appropriate position, we need to execute the line
`p.next = new Node(value, null);`
Let us focus on the right side of the equal sign and draw it out:
`________ = new Node(value, null);`



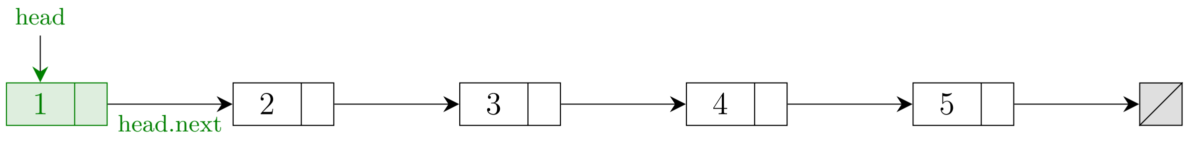
Now, let us evaluate the left-hand side of the equal sign. It states that `p.next` needs to be altered. It will be pointing to the new node, read it as `p.next` becomes the new node, like so:



We will end up with the following:

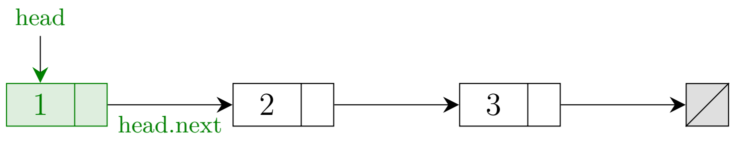


Once we execute the method, we will the variable `p` will be garbage collected and we end up with the following linked list:

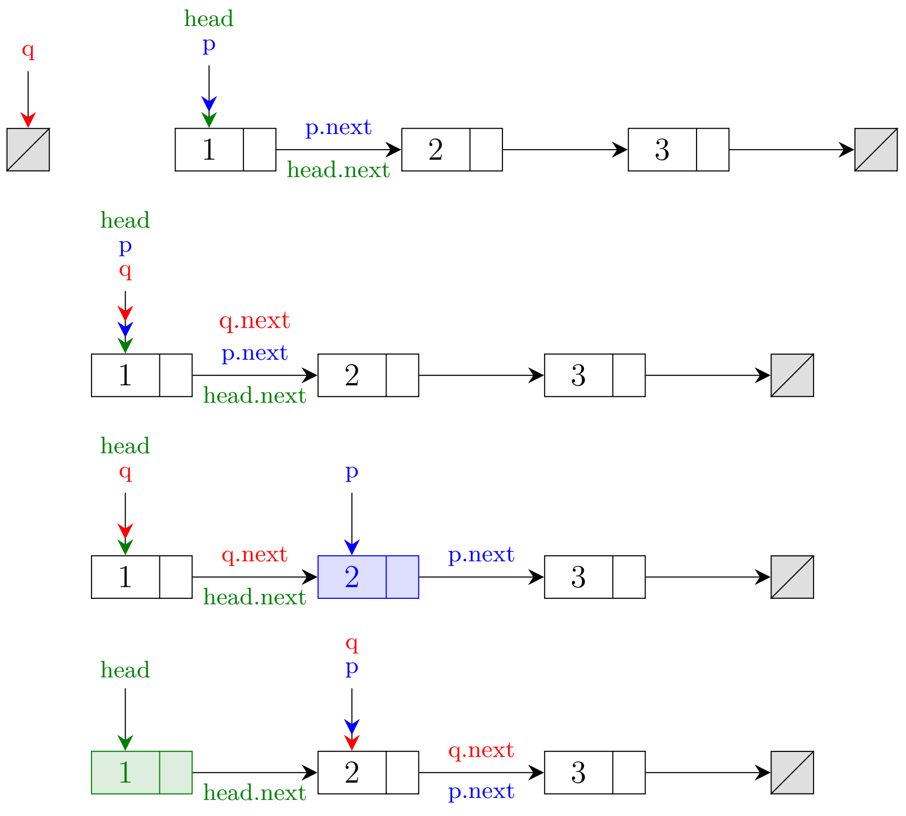


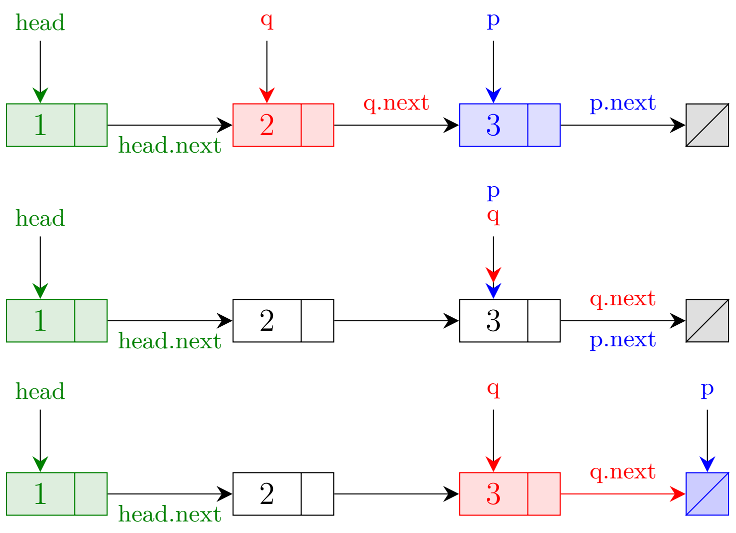
3.6 `insertAtRearTwoPointers` Method

We will insert at the rear, similar to `insertAtRearOnePointer`, but without the usage of `p.next` to check. We will use the two pointers `q` and `p`. Ensure that `q` is *just* before `p`. The objective is to loop until `p` is `null`. Once we reach that, we know that `q` is pointing to the last valid node, which allows us to update `q.next` (the horizontal arrow that points to `null`) to point to a new node. Suppose we have the following list to use for insertion at the rear using two pointers:



The traversal will be as follows (note that a specific instance, both `q` and `p` point to the same node, this occurs at one line of code only):





Now, we have `q.next` (the horizontal red arrow) to point to the new node to create. The code of the method will be as follows:

```java
public void insertAtRearTwoPointers(int value){
  Node q = null;
  Node p = head;
  //keep moving p until it reaches null, and ensure q is just before it
  while(p != null){
    q = p; //move q to point to where p is (both p and q pointing on same node)
    p = p.next;
  }
  // we have two options either q is still null, which means it never moved,
  // which means we have a null list (p is null by default is q never moved)
  if(q == null){//empty list, insert the new node
    head = new Node(value, head);// or  head = new Node(value, null);
  } else {// at least one node in the list and q.next is what needs to be used
    q.next = new Node(value, null);
  }
}
```

3.7 `sortedInsertion` Method

The sorted insertion method will ensure to use two pointer but not just insert at the front or rear, it will insert so that the linked list is sorted. It will use two pointers for this. It will be similar to the `insertAtRearTwoPointers(...)` method but not necessarily inserting at the rear all the time. The `while` loop will need to have an extra condition, which is to loop as long the value is less than the current node's item value. The code is displayed next.

```java
public void sortedInsertion(int value) {
  Node q = null;
  Node p = head;
  //move p as long as we have node and item of current node is less than value
  while (p != null && p.item < value) {
      q = p; //move q to point to where p is (both p and q pointing on same node)
```

```
        p = p.next;
    }
    if (q == null) {//null list or insertion at front
        //In new node, use head instead of null as it will account for both
        //situations, either null list or insertion at front
        head = new Node(value, head);// don't use null, i.e., new Node(value, null)
    } else {// at least one node in the list
        //insert between q.next and p
        q.next = new Node(value, p);
    }
}
```

There is an important note to mention, the `&&` in the condition of the `while` loop is *very* important. Using a single `&` would crash the program at specific instances. Suppose that `p` is currently pointing to `null`, would it be valid to check for `p.item`? No! That is because `null` doesn't have an item, nor a next, nor length, nor anything, which throws a `NullPointerException`. So, we will check the left condition (*i.e.*, `p != null`), this needs to be valid to check the condition to the right. In case it is not valid (*i.e.*, `false`), then we shouldn't other second condition. In Java, two ampersands `&&` will ensure that if the first condition is not valid, the we ignore the second condition and don't even check the second condition. However, if we use a single ampersands `&`, Java will check both conditions, whether they are true or false, both must be checked. Try changing `&&` to `&` and will realize that sorted insertion will not work, not even to insert one element.