Single Linked Lists Continued and Doubly Linked Lists

COSC 1P03 ✵ Lecture 04 ✵ Tuesday May 21, 2024[1]

Maysara Al Jumaily
amaysara@brocku.ca

Table of Content

3 Single Linked Lists Continued

We will quickly go over sorted insertion and removal operations and why having a tail reference is beneficial.

3.1 `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.

```
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
```

[1] Complied at: Tuesday May 21st, 2024 11:43am -04:00

```
    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.

3.2 `removeAtFront` Method

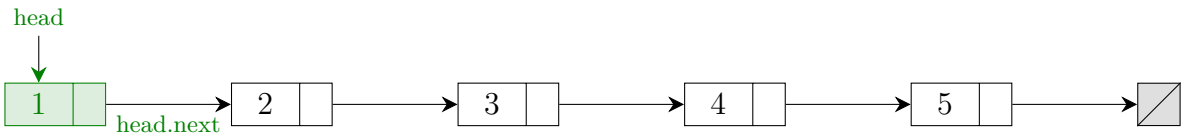
The objective of this method is to remove the first element of the linked list. It will also return the node that will be deleted. It will take $\mathcal{O}(1)$ in terms of time complexity because we don't loop through the elements. Everything is taking action at the beginning of the list. We could call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

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

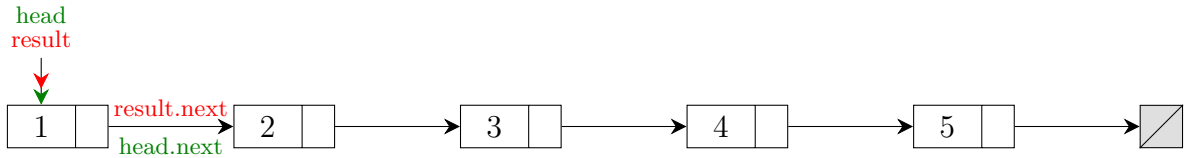
The code will be to store the node that will be removed, remove it from the linked list and then return that stored node. It is a good habit for you, a first-year student, to use the convention of naming the variable `result` to denote the object to be returned. It could be possible that we have a null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

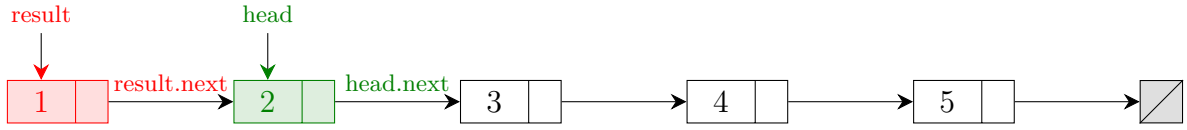
In case we have a `null` list (*i.e.*, `head` is `null`), then the following: `result = head;`, is the same as `result = null;`, which is valid. Suppose we have the following linked list:



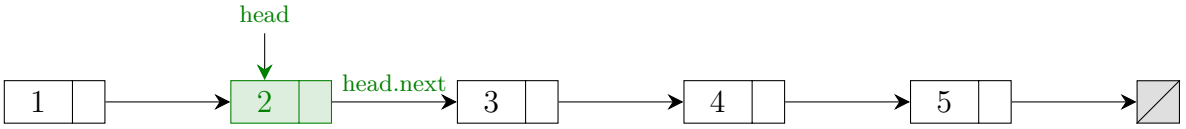
We will have to first execute the line:
`result = head;`
which has the following effect:



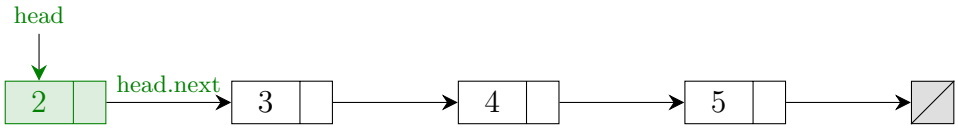
The next line,
`head = head.next;`
will transform the linked list into the following:



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 3.2.

3.3 Deletion

To remove a node somewhere in the middle or as the last node where there are several nodes linked, two references are needed. We will ensure to have `p` point to the node to insert and `q` point to the node previous to it. That way, we will link `q.next` to `p.next`. The idea is to loop as long as `p` is not `null` and `&&` that the current item value is not the one we are interested in. Move `q` and then `p` (remember that `p` is ahead of `q`). Eventually, we need to

get out of the `while` loop. Then, we will check the positions of p and q. It could be that we didn't find the item (*i.e.*, the item doesn't exist in the linked list), or removal at the front, or removal at the middle or rear (the operation of removal at the middle or rear with a linked list of several elements is the same). The code will be the following:

```
private void remove(int valueToRemove) {
  Node p = head;
  Node q = null;
  while (p != null && valueToRemove != p.item) {
    q = p;
    p = p.next;
  }
  if (p == null) { // the element not found
    System.out.println(
      "Error, the element " + valueToRemove + " doesn't exist in list"
    );
  } else {
    if (q == null) { // first element to delete
      head = head.next;
    } else { // somewhere in the middle
      q.next = p.next; // Remove!
    }
  }
}
```

3.4 Tail Reference

A *tail* reference is a variable of type `Node` that references the *last* valid node in the linked list. Typically speaking, insertion at the rear will take $\mathcal{O}(n)$ operations. A tail reference will ensure insertion at the rear will take $\mathcal{O}(1)$ operations. Note that it allows [1] for insertion at the rear, *not* the deletion of the last valid node! Essentially, we will start with a `null` linked list and both `head` and `tail`, which are both instance variables, will point to `null`. Suppose the value 1 will be inserted, then both `head` and `tail` point to the that stores 1. Now, suppose we will insert the value 2 in the linked list. We know that `tail` points to the last valid node, which means its next (*i.e.*, `tail.next`) will point to `null`. hence, we could ensure that `tail.next` points to the new node:

```
tail.next = new Node(2, null);
```

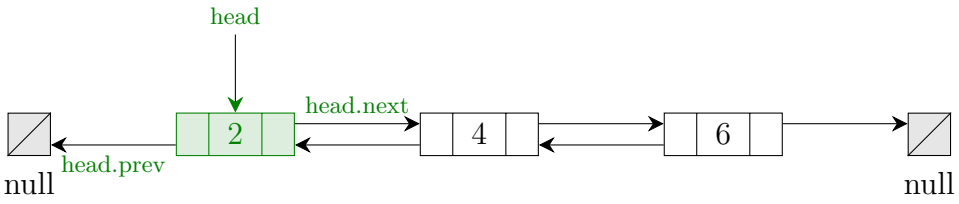
Right now, `tail` points to the node of 1. This is not enough be we said that `tail` always points to the last valid node. Now, we have 2 as the last valid node, hence, we need to shift `tail` one node to the right. by writing

```
tail = tail.next;
```

Now, we have `head` points to the node of 1 and `tail` points to the node of 2. We repeat the process until we have a linked list of n nodes and `tail` is referencing the last valid node. When we want to insert a node, it will be $\mathcal{O}(1)$ as we could `tail` which already points to the last valid node. Remember to shift tail one node to the right.

4 Doubly/Symmetric linked list

A *doubly* or *symmetric* linked list is a data structure containing nodes where each node has a prev, item and next. The objective is to allow for traversal forward (*i.e.*, using next) and backward (*i.e.*, using prev).



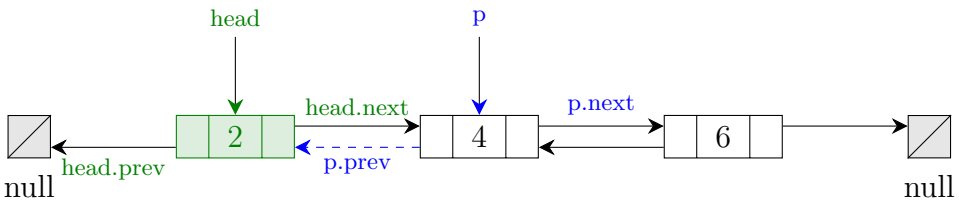
The `Node` class will be the following:

```
public class Node {
  public Node prev;
  public int item;
  public Node next;

  public Node(Node prevPassed, int itemPassed, Node nextPassed) {
    prev = prevPassed;
    item = itemPassed;
    next = nextPassed;
  }
}
```

4.1 Sorted Insertion

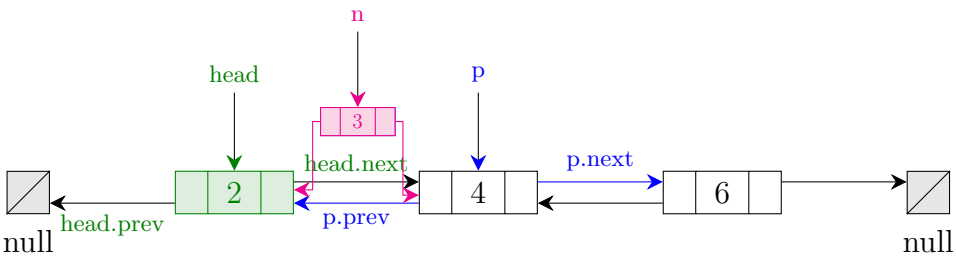
The following is a theoretical explanation on how insertions work here. Please have a look at the code for a complete implementation and dealing with edge cases. Note that we could use two pointer to insert (`q` and `p`), or just `p`, as `p.prev` would be `q`. To insert (front, middle, or rear), we can use one pointer `p`. This is because we are able to go back and forward, not just forward. Suppose we want to insert (in a sorted manner) the value 3 in the above doubly linked list. Then we need to ensure that `p` moves and points to the node containing the value 4:



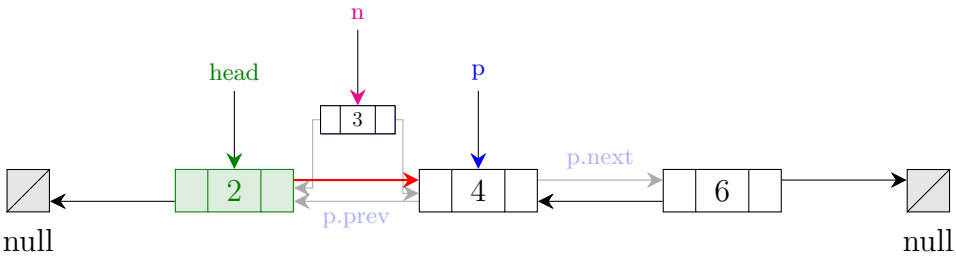
The new node will be between nodes 2 and 4. So, the previous of the new node is node 2, and the next is node 4. We can create it as such:

```
Node n = new Node(p.prev, 3, p);
```

To create the following:



In the line above, `p.prev` refers to the entire node of 2 and `p` refers to the entire node of 4. Now we want to update the horizontal arrows so that the node of 2 doesn't point to 4, but to 3, and node 4 doesn't point to 2, but to 3. Let us focus on the following horizontal arrow in red:



Let us try to change it relative to `p`. This means that if we started at `p`, how can we get to access it (don't use `head.next` because if we have lots of elements and wanted to insert in the middle, we don't have access to `head.next`). The code need to change that red horizontal arrow is

`p.prev.next`

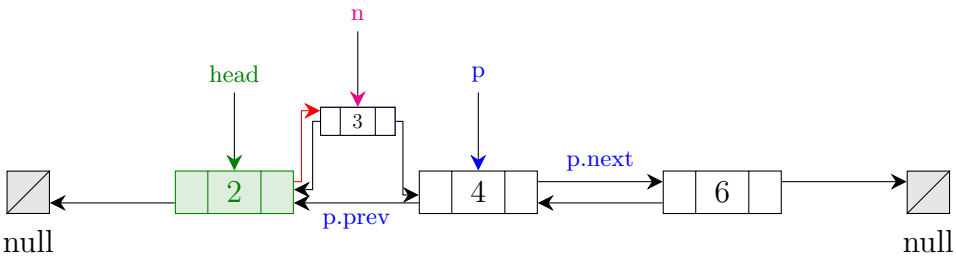
Let us understand it:

- Start at `p`, (*i.e.*, `p.`)
- Go to the node containing the value 2, (*i.e.*, `p.prev`)
- Now, within the current node that contains the value 2, we have a prev and a next, select/zoom in on its next, which is the red arrow (*i.e.*, `p.prev.next`)

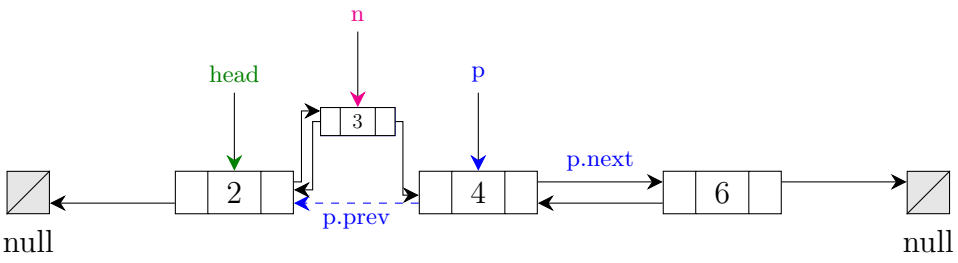
That red horizontal arrow needs to point to the new node, which is specified by `n`. Hence, writing the following line

```
p.prev.next = n;
```

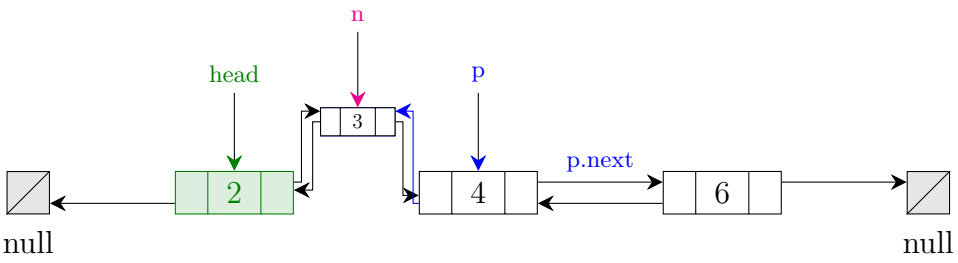
will transform the doubly linked list to the next figure



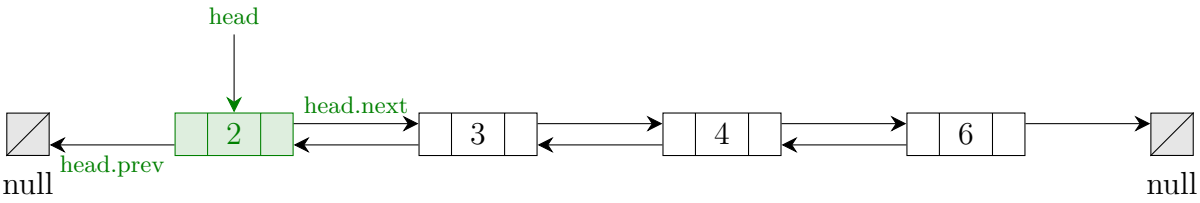
We successfully connected the node containing 2 with the node containing 3. We now need to ensure the previous of the node containing the value 4 points to the node containing 3. We need to change `p.prev` as highlighted in blue and dashed.



To access that blue dashed arrow relative to `p`, we use the simple line
`p.prev`
Now, we need to make it point to the new node that was inserted, which is accessed through `n`, like so
`p.prev = n;`
We will have the following doubly linked list:



Once we exist the method that inserted the new node, we will lose the temporary variables `n` and `p`. The final doubly linked list will look like the following:



In a nutshell, create the new node and make its prev point to `p.prev` its next point to `p`. Now, ensure the left link points to the new node using `p.prev.next = n;` and the right link points to the new node using `p.prev = n;`. The code of inserting a new node `n` before the node `p` is:

```
Node n = new Node(p.prev, 3, p);//make new node point to left of p and p
p.prev.next = n;//update the left link to point to new node
p.prev = n;//update the right link to point to new node
```

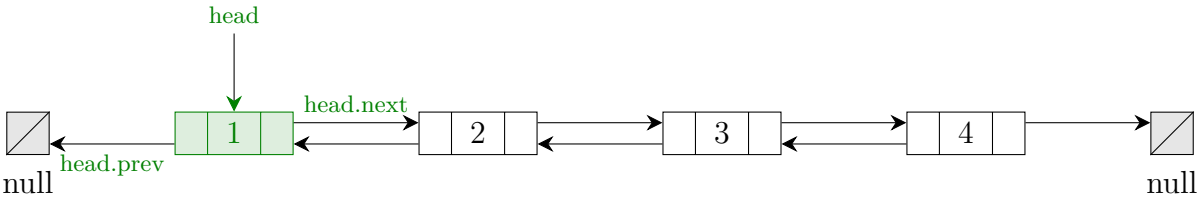
The code of the entire of sorted insertion is found next. It uses two pointers, `q` and `p` but remember that `q` is just `p.prev`.

```
public void sortedInsertionDoublyLinked(int value){
  Node p = head;
  Node q = null;
  while(p != null && p.item < value){
    q = p;
```

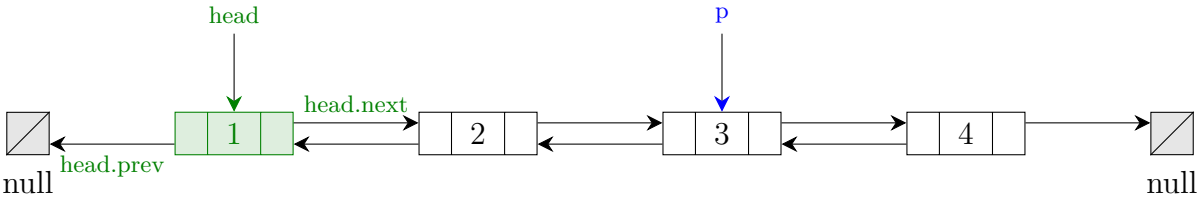
```
    p = p.next;
  }
  if(q == null){//empty list or insertion at front
    head = new Node(null, value, p);
    if(p != null){//not empty list but insertion at the front
      p.prev = head;//Make p's ← point to the new node
    }
  } else {
    Node n = new Node(q, value, p);
    q.next = n;//Make q's → point to the new node
    if(p != null){//we didn't insert at the very end of list
      //ensure the prev points to the new node
      p.prev = n;
    }
  }
}
```

4.2 Deletion

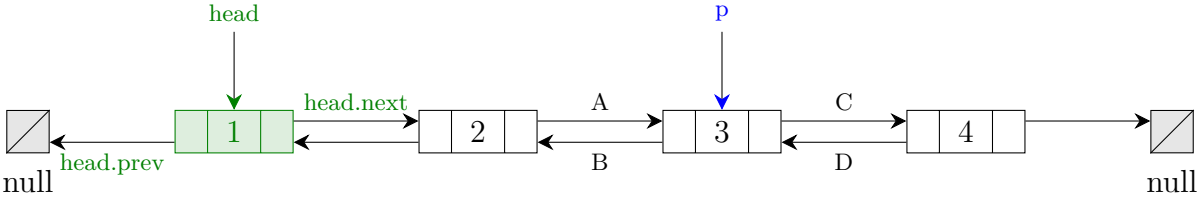
The concept of deletion requires that we change two horizontal arrows to remove an element. Suppose that we want to remove the value 3 in the following linked list:



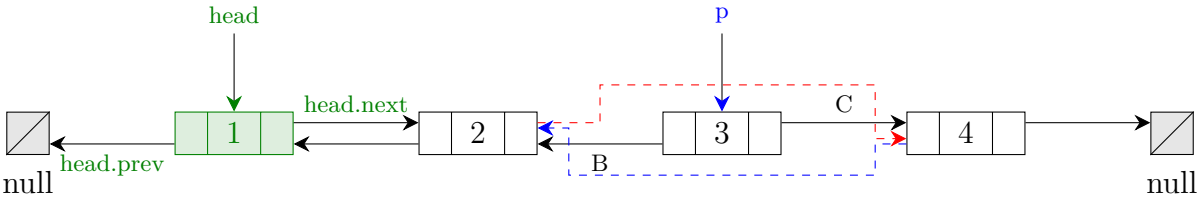
We will ensure that the local and temporary pointer `p` points to the node containing the value 3, as such



We have four different horizontal arrow associated here, let us label them and see which ones needs to be removed:



To delete a node, we need to ensure that **NOTING MUST BE POINTING TO IT**. It could point to other nodes, sure, but nothing points to it. We see that pointers A and D are the ones in question. They will need to be changed. Arrow A needs to point to the node containing 4 (*i.e.*, `p.next`) and arrow D need to point to the node containing 2 (*i.e.*, `p.prev`). The visualization will be as follows:

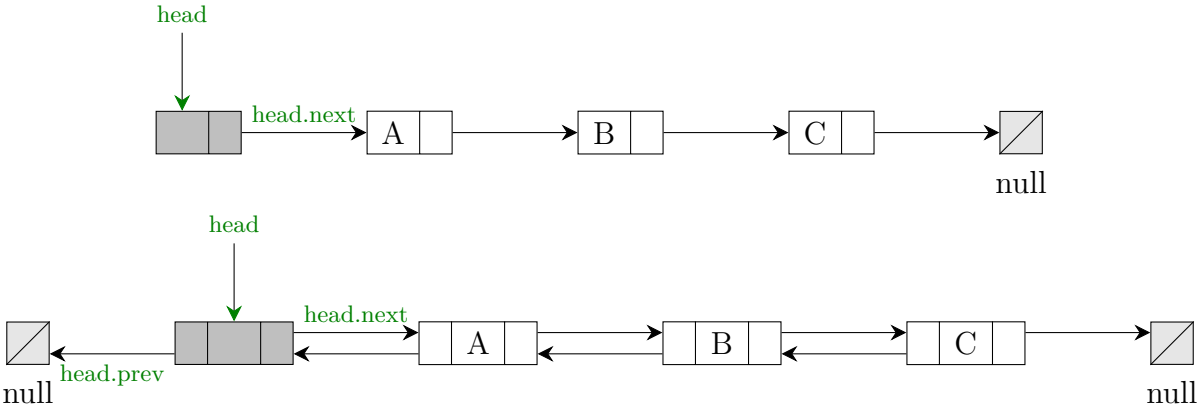


To change the red arrow, we need to access it first. The code of changing it would be
`p.prev.next = ________;`
Similar to subsection 4.1, we need to start at `p`, go left (*i.e.*, the node that contains the value 2) and zoom on its next, which is the horizontal arrow. It will need to point to `p.next`. Hence, the complete line is
`p.prev.next = p.next;`
The blue arrow will be pointing to `p.prev`. However, to access it, we need to start at `p`, move once to the right (*i.e.*, `p.next`) and then zoom on the prev of the node, which is given by `p.next.prev`. The code to make the blue arrow point to the node containing the value 2 is:
`p.next.prev = p.prev;`
The code that is needed to remove an element along with dealing with edge cases is shown below.

```
public void remove(int value){
  Node p = head;
  while(p != null && p.item != value){
    p = p.next;
  }
  if(p == null){//empty list or element not found in list
    System.out.println("Either empty list or element not found");
  } else {
    int storedItem = p.item; //in case we need to return it, but not required
    if(p.prev == null){//removal at the front
      head = head.next;
    } else {//arrow A (red arrow) in the explanation
      p.prev.next = p.next;
    }
    if(p.next != null){//removal in the middle, not rear, arrow D (blue arrow)
      p.next.prev = p.prev;
    }
  }
}
```

4.3 Header node

A *header* node is used in single or doubly linked list as the first node in the list. It doesn't contain any information and is there to simplify the code of insertion and deletion in a single/doubly linked list. For example, if we have to insert the elements `"A"`, `"B"` and `"B"`, then the linked list will be the following:



There are four nodes, but three data nodes. The first node is the header node, which is an extra node that is inserted by default and is never removed from the linked list. The "correct" begin of the linked list will be `head.next`. For example, if we wanted to loop through such a linked list, then we don't have `Node p = head;`, instead, we use `Node p = head.next;`. As mentioned above, the benefit of having a header node is to simplify the insertion/deletion code. Otherwise, it is not beneficial. For instance, the code sorted insertion in a single linked list (left) versus a single linked list with a header node (right) is found below:

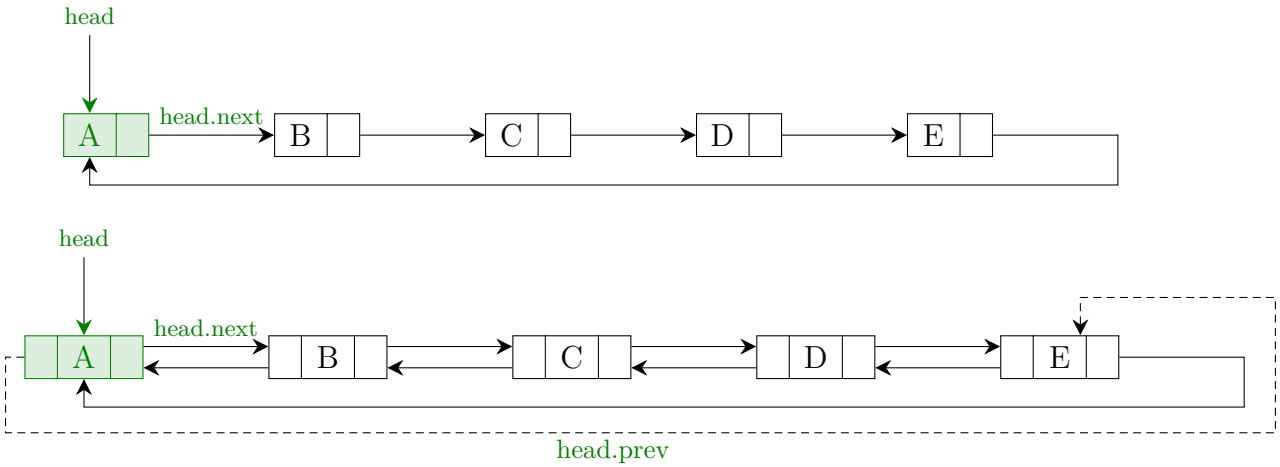
```
//sorted insertion without header node
public void sortedInsertion(int value){
  Node q = null;
  Node p = head;
  while (p != null && p.item < value){
    q = p;
    p = p.next;
  }
  if (q == null) {
    head = new Node(value, head);
  } else {
    q.next = new Node(value, p);
  }
}
```

```
//sorted insertion with header node
public void sortedInsertion(int value){
  Node q = null;
  Node p = head;
  while (p != null && p.item < value){
    q = p;
    p = p.next;
  }
  // doesn't matter what q is, use
  // its next and insert the node
  q.next = new Node(value, p);
}
```

4.4 Circular Linked List

A *circular* single/doubly linked list is a linked list that doesn't contain any `null`s (unless there are no node). In a single linked list, the last valid node points to `head`. In a doubly linked list, the last valid node points to `head` and `head.prev` points to the last valid node.

The benefit is that we could go directly from the last node to the first node. Another benefit is that we could traverse the linked list from any node. We don't have to start at the `head`. Lastly, instead of have the last valid node to point to `null`, which is a waste, we could make it point to `head`. In a circular linked list, there are no `null`s, hence, looping through a circular linked list will be different than the typical approach. Here are examples:



To loop through a circular linked list, use head as the starting point. In case we reach `head` again, then we ended where we started. It is not a must that we use head as the starting point. We could use another node as our starting point and if we reach that node again, it means we looped through all of the elements. The idea is that we have `p` starting at the first node `head`, print `p`'s content, move `p` to the next node, and now, we continue looping as long as `p != head`.

```
if(head != null){//ensure there are nodes in the linked list
  p = head;
  do {
    //use a do-while loop as the code will run at least once, even if the
    //condition is false.
    System.out.println(p.item);
    p = p.next;
  } while (p != head);
}
```