CS1102: Data Structures and Algorithms

Tutorial 3 Linked Lists (Solutions)

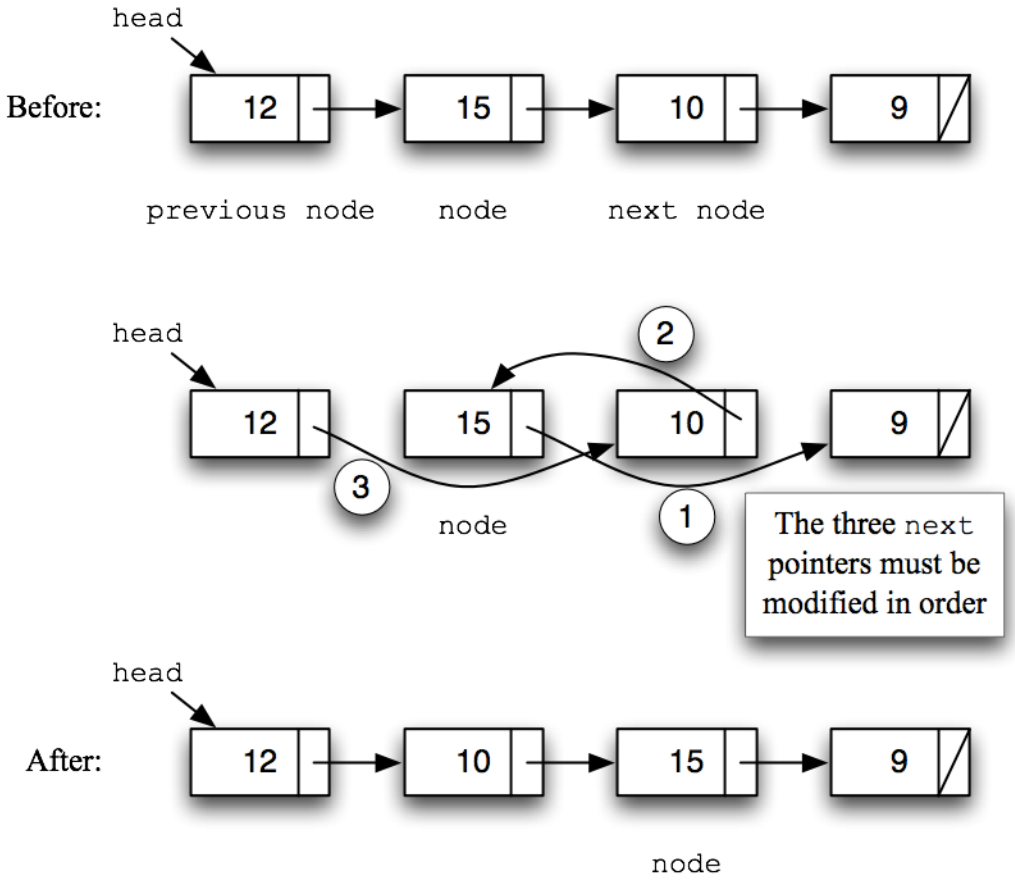
Week of 8/2/2010

1. The following four classes are available to you. The `ListNode` class represents a linked list node, while the `DListNode` class represents a doubly linked list node. Linked lists and doubly linked lists are represented by the `BasicLinkedList` and `DoublyLinkedList` classes respectively.

```
class ListNode<E> {
  private E element;
  private ListNode<E> next;

  public ListNode(E item, ListNode<E> next) { ... }
  public E getElement() { ... }
  public void setElement(E item) { ... }
  public ListNode<E> getNext() { ... }
  public void setNext(ListNode<E> next) { ... }
}

class DListNode<E> extends ListNode<E> {
  private DListNode<E> prev;

  public DListNode(E item, DListNode<E> next,
    DListNode<E> prev) { ... }
  public DListNode<E> getPrev() { ... }
  public void setPrev(DListNode<E> prev) { ... }
}

class BasicLinkedList<E> {
  protected int num_nodes = 0;
  protected ListNode<E> head = null;

  public int size() { ... }
  public ListNode<E> getHead() { ... }
  public void addHead(E item) { ... }
  public void deleteHead() { ... }
}

class DoublyLinkedList<E> {
  protected int num_nodes = 0;
  protected DListNode<E> head = null;

  public int size() { ... }
  public DListNode<E> getHead() { ... }
  public void addHead(E item) { ... }
  public void deleteHead() { ... }
}
```

a. For the `BasicLinkedList` class, write a new method: `public void switchWithNext(ListNode<E> node)` that switches the position of `node` with the node at `node.getNext()`. If switching is not applicable, the method does nothing. Similarly, write a new method: `public void switchWithNext(DListNode<E> node)` for the `DoublyLinkedList` class.

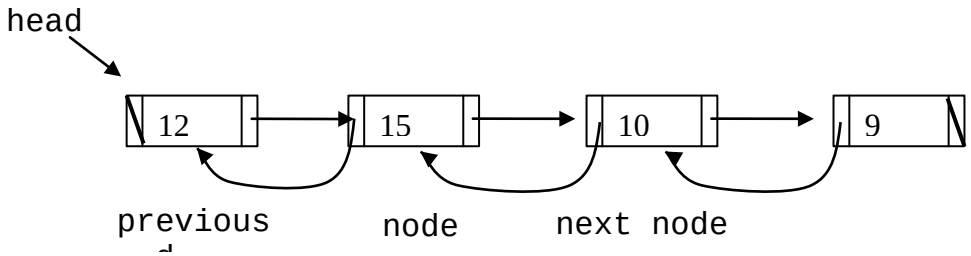
Answer:
The idea behind switching a node with its next node is that we have to modify the `next` pointer for three nodes: the node itself, the next node and the previous node. The order in which we modify these pointers is important – the `next` pointer for the node itself must be modified before we modify the `next` pointer of the next node. Otherwise, we will no longer have reference to the rest of the linked list (via the `next` pointer of the next node). This is illustrated in the diagrams below where we want to switch the node with Integer value 15 with its next node, the one with Integer value 10. The exception to this is when there is no next node. In other words, the node is the last node in the linked list and it has no next node to switch with.

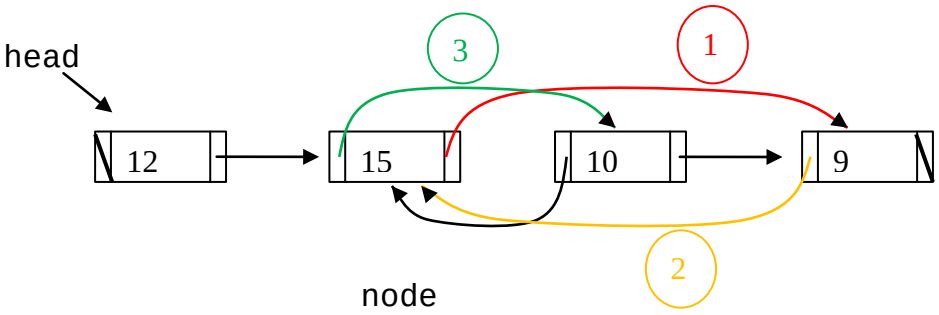


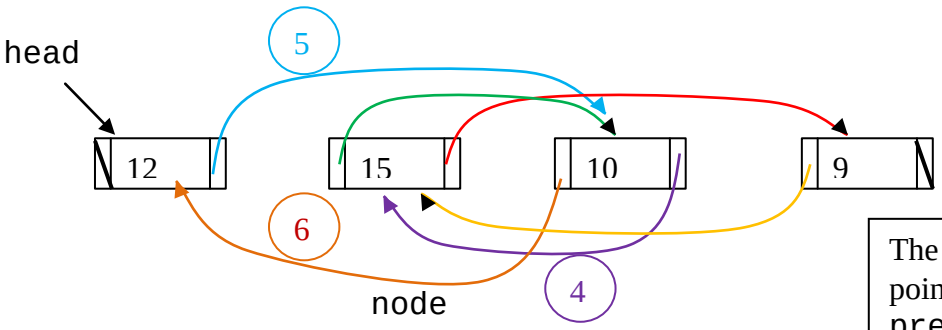
The `DoublyLinkedList` class uses instances of `DListNode` so we can obtain the previous and next nodes easily with `node.getPrev()` and `node.getNext()` respectively. For the `BasicLinkedList` class however, instances of the `ListNode` class is used. So we no longer have the luxury of calling `node.getPrev()`. We have to traverse the linked list from the head and manually find the previous node.

On the other hand, we have to modify up to 3 more pointers for the `DoublyLinkedList` class because each node now has two pointers: `next` and `prev`. The 3 `prev` pointers we have to modify belong to the node itself, the next node, and the next node's next node (if any). The order in which we have to modify all the pointers is described in the following diagram:

Before

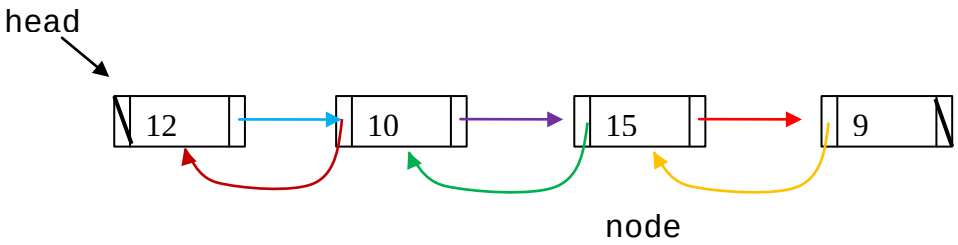






The three `next` pointers and three `previous` pointers must be modified in order

After



The last thing to note is to remember to modify the `head` pointer if it was affected during the switch.

```
public void switchWithNext(ListNode<E> node) {
  ListNode<E> prev = null;
  ListNode<E> next = node.getNext();

  // No switching applicable
  if(next == null) return;

  // Find previous ListNode
  if(head != node) {
    prev = head;
    while(prev != null && prev.getNext() != node) {
      prev = prev.getNext();
    }
  }

  // Do the switch
  node.setNext(next.getNext());     //1
  next.setNext(node);               //2
  if(prev != null) {
    prev.setNext(next);             //3
  } else { // Make sure head is pointing correctly
    head = next;
  }
}

public void switchWithNext(DListNode<E> node) {
  DListNode<E> prev = node.getPrev();
  DListNode<E> next = (DListNode<E>) node.getNext();

  // No switching applicable
  if(next == null) return;

  // Do the switch
  node.setNext(next.getNext());     //1
  node.setPrev(next);               //2
  if(next.getNext() != null) {
    next.getNext().setPrev(node);   //3
  }
  next.setNext(node);
  if(prev != null) {
    prev.setNext(next);             //4
    next.setPrev(prev);             //5
  } else { // Make sure head is pointing correctly
    head = next;
    next.setPrev(null);             //6
  }
}
```

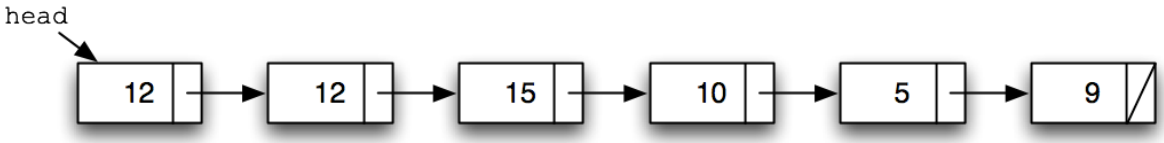
b. The following method is added to the `BasicLinkedList` class:

```
public boolean mysteryMethod() {
  ListNode<E> current = getHead();
  boolean flag = false;
  if(current == null) return flag;
  while(current.getNext() != null) {
    if(current.getElement().compareTo(
        current.getNext().getElement()) > 0) {
      switchWithNext(current);
      flag = true;
    } else {
      current = current.getNext();
    }
  }
  return flag;
}
```

The declaration of the class is also changed to:
`class BasicLinkedList<E extends Comparable<? super E>>`

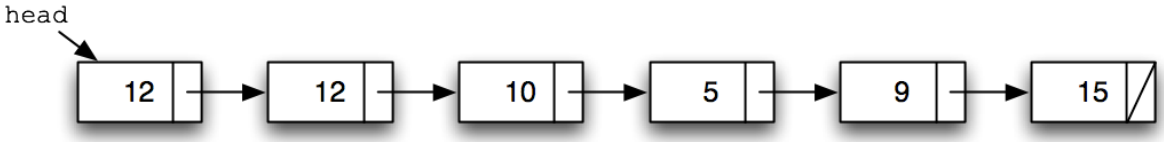
The linked list `myLinkedList` below is an instance of the `BasicLinkedList<Integer>` class. Draw what it would look like after calling `myLinkedList.mysteryMethod()`. Then in one sentence, explain what `myLinkedList.mysteryMethod()` does to `myLinkedList`.

`BasicLinkedList<Integer> myLinkedList`



Answer:

`BasicLinkedList<Integer> myLinkedList`



The method call moved the node with the highest Integer value (15) to the tail of the linked list. Note that the first two nodes (with the same Integer value of 12) were not interchanged by the method call.

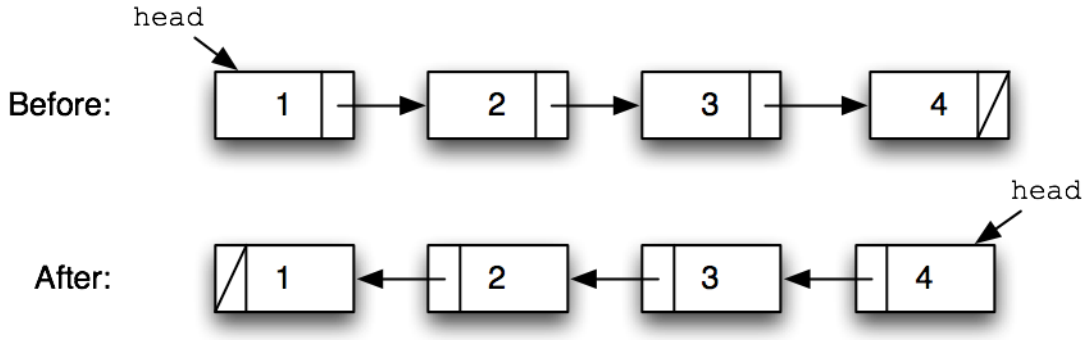
c. The following method is added to the `BasicLinkedList` class in (b):

```
public void mysteriousMethod() {
  boolean flag = mysteryMethod();
  while(flag == true) {
    flag = mysteryMethod();
  }
}
```

In one sentence, explain what `myLinkedList.mysteriousMethod()` does to `myLinkedList`.

Answer:
The method sorts the `ListNode` instances in `myLinkedList` in ascending order.

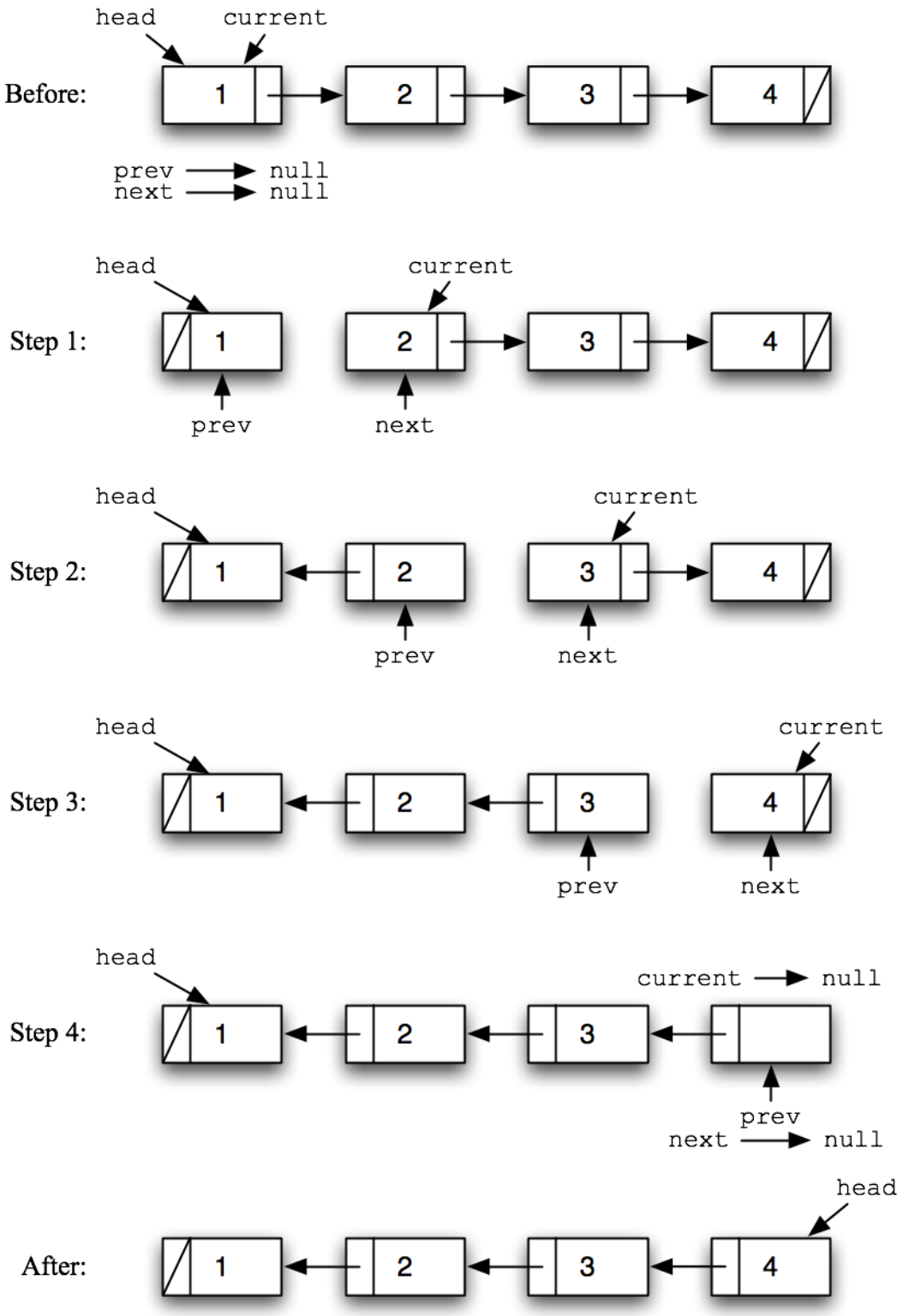
2. Write a method to reverse a linear linked list as illustrated below. Your method should not instantiate any new nodes. Your method should be included in the `BasicLinkedList` class (question 1).




```
class BasicLinkedList<E> {
  // Other methods as defined in question 1

  public void reverse() {
    // Your code here
  }
}
```

Answer:
The idea behind reversing a linear linked list is that we have to modify the `next` pointer for each node to point to its previous node. To accomplish this, we need to keep three references: the current node, the previous node and the next node. The references to the current and previous nodes are obvious – we need to modify the `next` pointer of the current node to point to the previous node. The reference to the next node is necessary because after we have modified the current node, we need to proceed to the rest of the linked list.

This process is illustrated in the diagrams below using the same example as the provided by the question.



```
public void reverse() {
  // Special case when head is null
  if(head == null) return;

  ListNode<E> curr = head;
  ListNode<E> next = null; // This will be correctly set later
  ListNode<E> prev = null;

  // While there is still more nodes
  while(curr != null) {
    // Remember the next node
    next = curr.getNext();

    // Reverse pointer for current node
    curr.setNext(prev);

    // Move on to the next node
    prev = curr;
    curr = next;
  }

  // Set the new head
  head = prev;
}
```

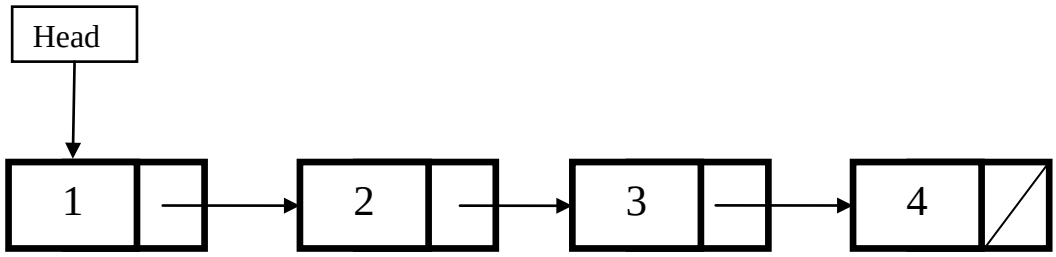
3. (Circular linked list) Using what we have explored in the lectures, write a java program to implement a circular linked list class which extends BasicLinkedList class with a tail pointer. You should write a constructor which creates a circular linked list from a given BasicLinkedList. You should also write an addTail ( ) method, which adds a new node as the last node in the list. And lastly you should write a toString( ) method, which go through the circular list and returns the String form of every element. Please be noted that class E contains or inherits a toString( ) method.

```
public class CircularList<E> extends BasicLinkedList<E>{
   protected ListNode<E> tail;

   public CircularList(){
         super();
         tail=null;
   }

   public CircularList(BasicLinkedList<E> list){
   // Your code here
   }

   public addTail(E item){
   // Your code here
   }

   public String toString(){
   // Your code here
   }
}
```

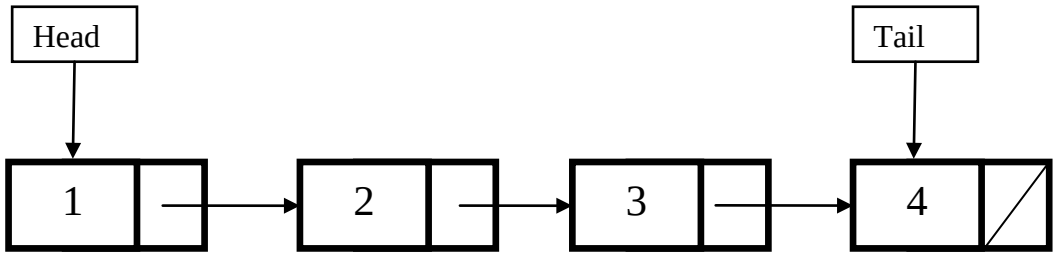
Answer:
A circular linked list is a linked list where the last node is connected to the first. It facilitates the process if it is required to go through the list repeatedly, in programs such as round-robin system for allocating a shared resource. To convert a linked list to a circular list, one needs to connect the last node with the first node. It is also recommended to have a tail node, which marks the end of the circle. When it is necessary to add or delete head, the tail can facilitate the process.

The following diagrams illustrate the process in the constructor to convert from a linked list to a circular list.

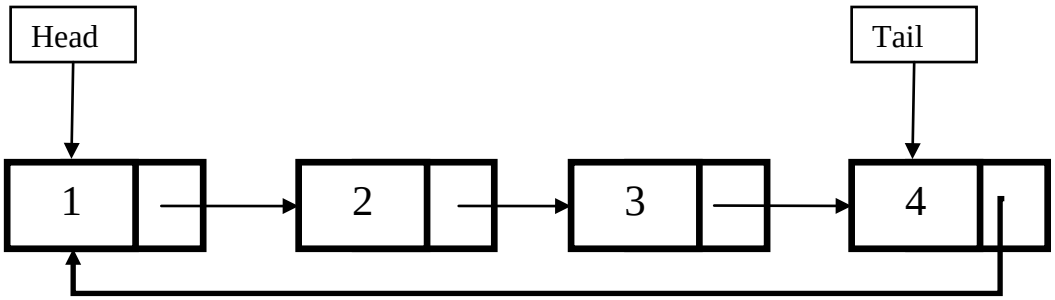
Constructor
(Initial state: the BasicLinkedList Class)



Step 1: Identify Tail

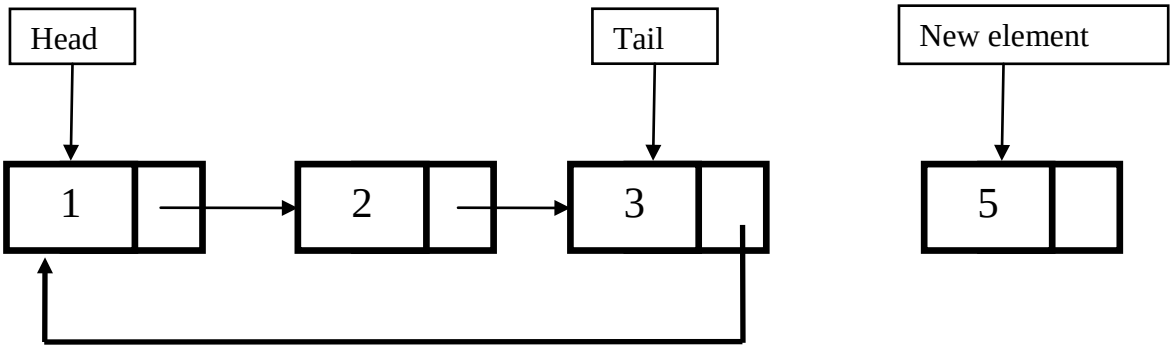


Step 2: change the next element of tail

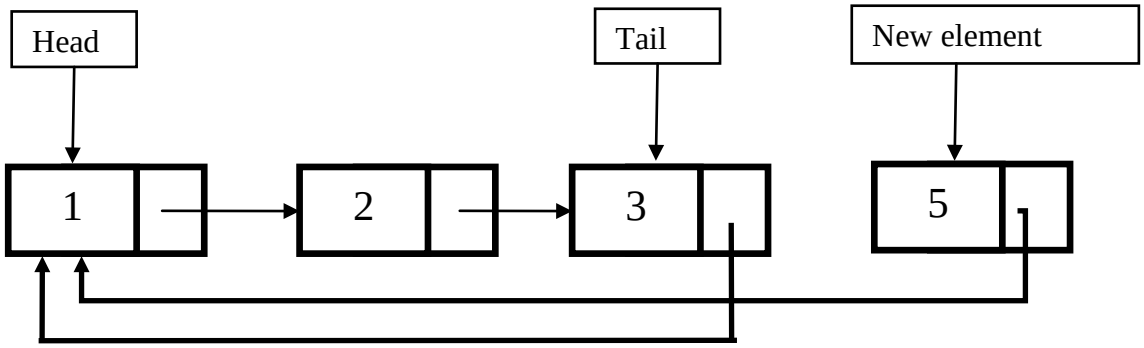


The following diagram shows the process to add a tail element to the circular list.

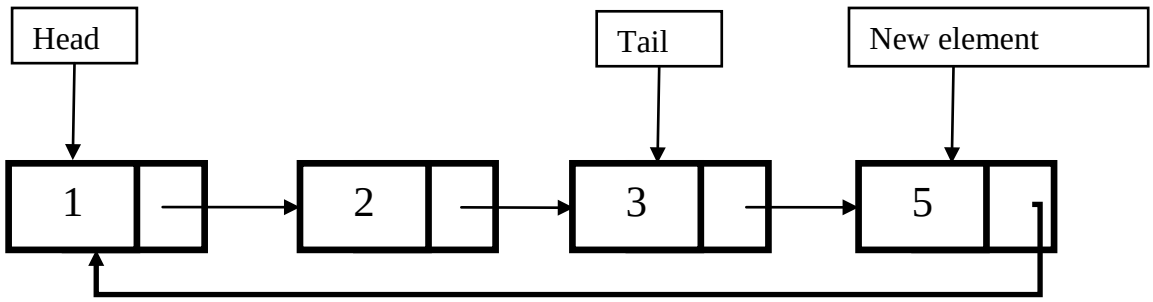
Add Tail



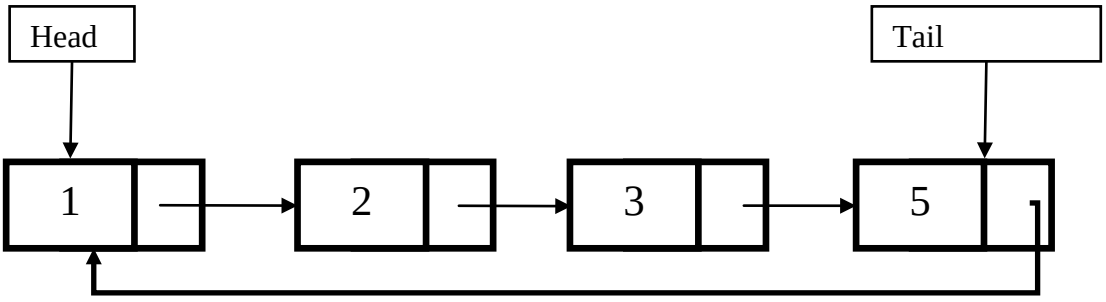
Step 1: set the next of new element as head



Step 2: set the next of tail as new element



Step 3: set tail to the position of new



The code for this question is provided below.

```
public class CircularList<E> extends BasicLinkedList<E>{
   protected ListNode<E> tail;

   public CircularList(){
         super();
         tail=null;
   }

   public CircularList (BasicLinkedList<E> list){
   //constructor from a linked list
         head=list.head;
         num_nodes=list.num_nodes;
         tail=list.head;
         if(tail!=null){
               //look for the tail element
               while(tail.getNext()!=null){
                     tail=tail.getNext();
               }
               tail.setNext(list.head);
         }
   }

   public void addTail(E item) {
         ListNode<E> newNode= new ListNode<E>(item, head);
               //1. Set the next of new element as head
         if(tail==null){
               head=newNode;
               tail=head;
         }
         else{
               tail.setNext(newNode);     //2. Set the next of tail
               tail=newNode;         //3. Set the new value of tail
         }
         num_nodes=num_nodes+1;
   }

   public String toString(){
         ListNode<E> node=head;
         String s="";
         if(node!=null){       //if head is null, return nothing
               //use the toString() method from class E
               s+=node.getElement().toString();
               node=node.getNext();
               while(node!=head){
                     s+=node.getElement().toString();
                     node=node.getNext();
               }
         }
         return s;
   }
}
```

4. The following class extends the `BasicLinkedList<E>` class from question 1:

```
class TailedLinkedList<E> extends BasicLinkedList<E> {
  protected ListNode<E> tail = null;

  public ListNode<E> getTail() { ... }
  public void addTail(E item) { ... }
  public void deleteTail() { ... }
  public void addHead(E item) { ... } // Overridden to handle
  public void deleteHead() { ... }    // tail pointer correctly.
  public void insertAfter(ListNode<E> node, E item) { ... }
}
```

Given an instance of a `TailedLinkedList<Integer>` class from above, write a method that finds the largest k Integer values in that linked list. You may assume that k will always be given as a positive integer. Your method should return a linked list (`TailedLinkedList<Integer>`) of the largest k Integer values. This linked list must be sorted in descending order (the head node stores the largest Integer value). If the size of the given linked list is less than k, your method should return a new linked list of the same size with all the Integer values from the given linked list sorted in descending order. You are not allowed to use arrays in your method. You should use the `TailedLinkedList<Integer>` class instead. Also note that the original linked list must not be destroyed by your method.

```
class LargestIntegers {
  public static TailedLinkedList<Integer> findKLargest(
      TailedLinkedList<Integer> myLinkedList, int k) {
    // Your code here
  }
}
```

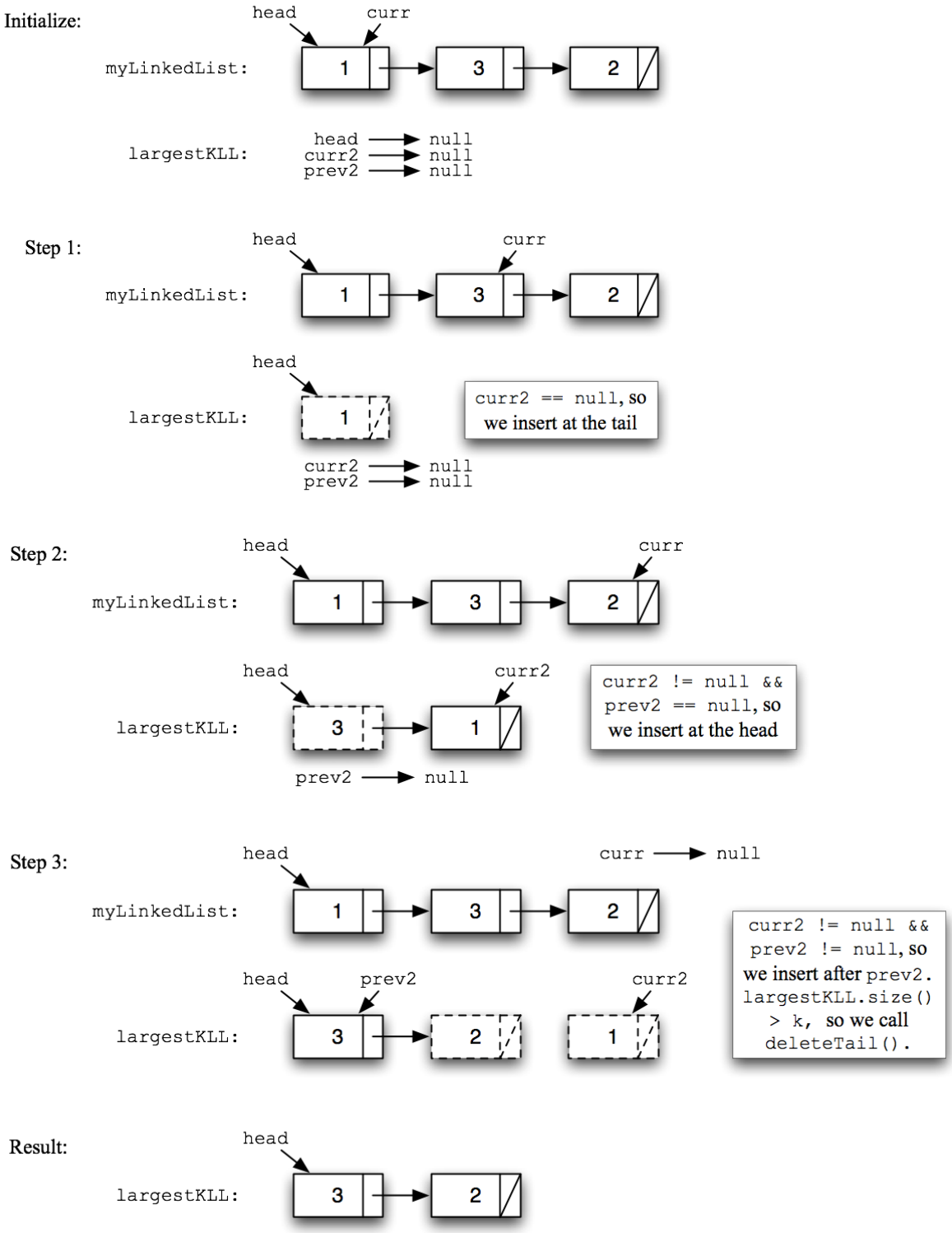
Answer:
The idea is to build a sorted partial-copy (`largestKLL`) of the given linked list (`myLinkedList`) as we traverse `myLinkedList` from head to tail. We say that it is a partial-copy because we only need to keep copies of at most k nodes from `myLinkedList` in `largestKLL`. After the traversal is done, we simply return `largestKLL` as the answer.

The traversal is done as follows. In the beginning, `largestKLL` has 0 nodes. We start traversing `myLinkedList` from its head. We compare its Integer value with the Integer value of every node in `largestKLL` from head to tail until we find the first node in `largestKLL` that has a smaller value. We then make a copy of the current `myLinkedList` node and insert it as the previous node of the found `largestKLL` node. In other words, we are keeping `largestKLL` sorted in descending order by making sure we always insert at the correct position. If the size of `largestKLL` becomes more than k with this insertion, we call `deleteTail()` to trim `largestKLL` back to size k. As a shortcut, when `largestKLL` is of size k before the insertion, we can compare the current `myLinkedList` node directly with the tail of `largestKLL`. If the tail has a larger Integer value, no insertion is necessary. This process continues until `myLinkedList` has been traversed completely.

From this traversal process, we can understand why keeping `largestKLL` to at most k nodes would be more efficient than just trimming it at the end of traversing `myLinkedList`. Keeping the size to at most k nodes allows us to reduce the number of comparisons (to find the insertion point) and insertions in `largestKLL`.

Note that linked lists are not typically used for solving the top-k problem. Later in this course, you will learn of the heap data structure that can be used to solve this problem with better running time.

The following diagrams illustrate our algorithm when k=2:



```
class LargestIntegers {
  public static TailedLinkedList<Integer> findKLargest(
      TailedLinkedList<Integer> myLinkedList, int k) {

    // Use largestKLL to be the tailed linked list of largest
    // k Integer values.
    TailedLinkedList<Integer> largestKLL =
      new TailedLinkedList<Integer>();

    // Main loop: traverse myLinkedList
    ListNode<Integer> curr = myLinkedList.getHead();
    while(curr != null) {
      // Get Integer value of the current node
      Integer tempInt = curr.getElement();

      // If largestKLL is of size k, check with its tail first
      if(largestKLL.size() == k) {
        if(largestKLL.getTail().getElement().compareTo(
            tempInt) > 0) {
          // Tail is bigger, so no insertion is necessary
          // Advance to next node in myLinkedList
          curr = curr.getNext();
          continue; // This skips the rest of the while loop
                    // and goes back to the beginning of the
                    // while loop.
        }
      }

      // Find the first node in largestKLL that is smaller
      // We also need to keep a reference to the previous node
      // so that we can insert a node for tempInt before
      // the smaller node.
      ListNode<Integer> curr2 = largestKLL.getHead();
      ListNode<Integer> prev2 = null;
      while(curr2 != null && curr2.getElement().compareTo(
          tempInt) > 0) {
        // Advance previous and current pointers
        prev2 = curr2;
        curr2 = curr2.getNext();
      }

      // At this point, the while loop above exited because:
      // (1) curr2 == null or (2) curr2 == smaller node
      // For (1), we insert a node for tempInt at the tail
      //   of largestKLL.
      // For (2), there are two cases:
      // (2a) prev2 == null, so we insert node for tempInt at the
      //   head,
      // (2b) prev2 != null, so we insert node for tempInt after
      //   prev2.
      if(curr2 == null) {
        largestKLL.addTail(tempInt);
      } else if(prev2 == null) {
        largestKLL.addHead(tempInt);
      } else {
        // Insert new node for tempInt
```

```
          largestKLL.insertAfter(prev2, tempInt);
        }

        // If size of largestKLL > k, delete the tail
        if(largestKLL.size() > k) {
          largestKLL.deleteTail();
        }

        // Advance to next node in myLinkedList
        curr = curr.getNext();
      }

      // Return the linked list of largest k Integer values
      // Note that if the size of myLinkedList is less than k,
      // the result will still be correct.
      return largestKLL;
    }
}
```