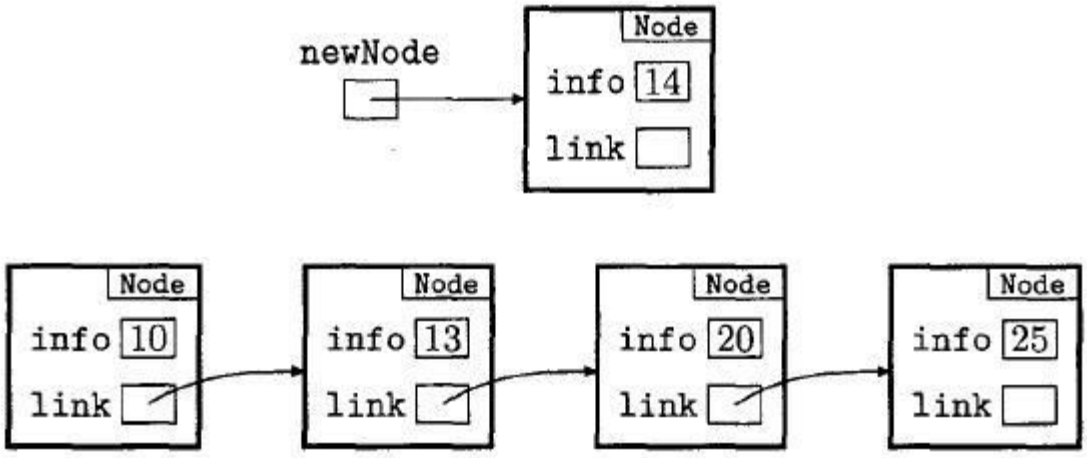
Linked List – Exercise

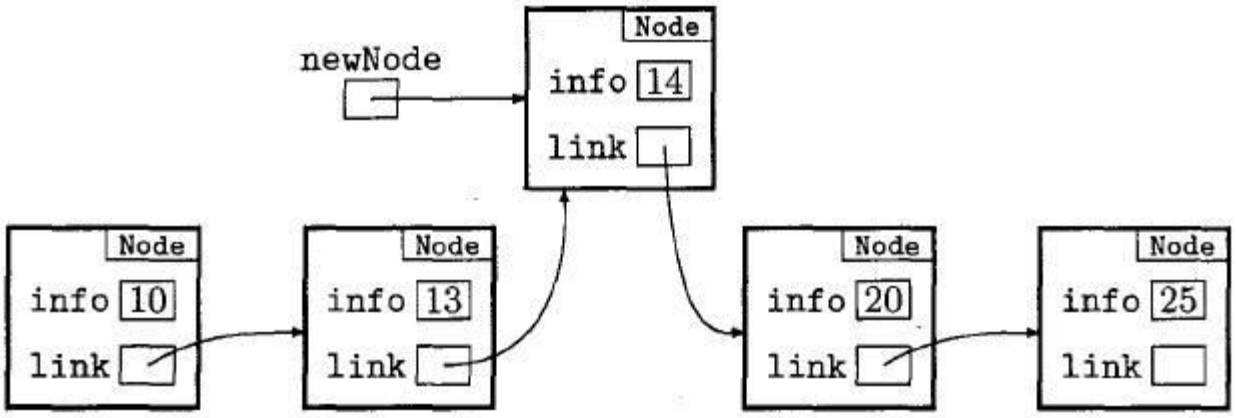
More Operations on Linked Lists

Although the list manipulation techniques discussed in the previous section are adequate for some applications, many problems require more advanced techniques, using one or more auxiliary reference variables to keep track of nodes while links are being adjusted.

As a first example, let us consider the problem of inserting a new node into a list where the nodes of the list are to be ordered. For the simple list structure that we are using, let us suppose that we want to maintain our lists in ascending order by the info fields of the nodes. To illustrate the problem, consider the portion of the list shown in the next diagram and suppose that we want to insert a node containing the value 14 into its correct position.



The insertion process consists, essentially, of simply changing the appropriate link fields to achieve the structure shown in the next diagram.



The insertion is carried out in two phases: first the correct location for the new node is found and then links are adjusted so that the node is inserted in the list at this point. To find the correct location for the value 14, we traverse the list using an auxiliary reference variable until we reach the node containing the next largest value, 20. The new node should be inserted just before this one. At this point, it is easy to link the new node to the node containing 20 but we cannot go back along the list to link the previous node, containing 13, to the new node because the links only point forward, not backward. One solution to this problem is to use two auxiliary reference variables in our traversal: one to the current node being examined and the other to the previous node. With these variables we can, if necessary, link the previous node to the new one. This process is implemented in the next example.

Example 1

The method `insert` will insert a node containing the value `item` in its `info` field in a linked list in which the `info` fields of the nodes are in ascending order.

```
public void insert (int item)
{
   // Find correct location in list for new item
   Node current = head;
   Node previous = null;
   boolean located = false;
   while (!located && current != null)
      if (item < current.info)
         located = true;
      else
      {
         previous = current;
         current = current.link;
      }

      // Create a new node containing item and
      // referring to correct location in list
      Node newNode = new Node(item,current);

      // set link to refer to new node
      if (current == head)
         head = newNode;
      else
         previous.link = newNode;
}
```

To delete a node containing a particular value in its `info` field, we must first search the list to find the node that is to be deleted and then adjust links so that this node is snipped out and the reference to the deleted node is redirected to the following node (if there is one). Here again, we must be careful to keep track of the node preceding the one currently being examined so that we can adjust its link, if necessary.

Example 2

This method searches an unordered linked list for a node containing the value `item` in its `info` field. If it finds such a node, it deletes it from the list. If there is no such node, the method does nothing; if there is more than one node containing `item`, only the first is deleted.

```
public void delete (int item)
{
   // Search for node to be deleted
   Node current = head;
   Node previous = null;
   boolean found = false;
   while (!found && current != null)
      if (current.info == item)
         found = true;
      else
      {
         previous = current;
         current = current.link;
      }
   // Perform deletion, if item found
   if (found)
      if (current == head)
         head = head.link;
      else
         previous.link = current.link;
}
```

Section 1:

1. Create the following instance methods in the Linked list definition from the previous exercise.

   a. Sort() –
      - will sort the fractions in linked list in order of value of the fraction (from smallest to largest)

```
/*
 *This method sorts the List in order of values of Fractions
(smallest to greatest)
 *pre: none
 *post: none
*/
public void Sort () {
      Fraction small;
      for (Fraction top = head; top != null; top = top.link) {
```

```
                        small = top;
                        for (Fraction i = top; i != null; i = i.link) {
                                if ((double) i.num/i.den < (double)
    small.num/small.den) {
                                        small = i;
                                }
                                int n = small.num;
                                int d = small.den;
                                small.num = top.num;
                                small.den = top.den;
                                top.num = n;
                                top.den = d;
                        }
                }
        }
```

b. Insert(int n, int d)

- will insert a fraction with num=n and den =d in the linked list in order of value.

```
    /*
     * This method inserts a fraction with num = n and den = d in the linked
list in order of value.
     * pre: numerator and denominator
     * post: none
    */
    public void Insert (int n, int d) {
           Fraction previous = null;
           Fraction current = head;
           boolean located = false;
           while (!located && current != null) {
                  if ((double) n/d < (double) current.num/current.den) {
                         located = true;
                  }
                  else {
                         previous = current;
                         current = current.link;
                  }
           }
           Fraction newFraction = new Fraction (n,d,current);
          // set link to refer to new node
          if (current == head)
             head = newFraction;
          else
             previous.link = newFraction;
    }
```

c. Remove(int n, int d)

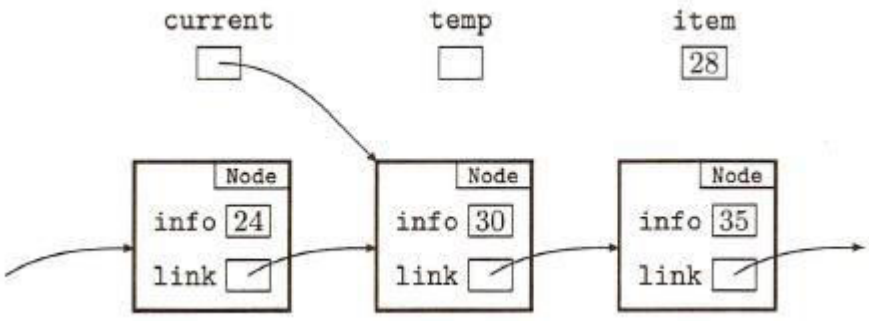
- will search and remove the first fraction with num=n and den=d in the linked list.

```
          /*
           *This method Removes a Fraction from List
           *pre: numerator and denominator
           *post: none
```

```
        */
        public void Remove(int n, int d) {
                Fraction current = head;
                Fraction previous = null;
                boolean located = false;
                if (head == null)
                        System.out.println("This list is EMPTY !");
                else {
                        while (!located && current != null) {
                                if (current.num == n && current.den == d)
                                        located = true;
                                else {
                                        previous = current;
                                        current = current.link;
                                }
                        }
                }
                if (located == false)
                        System.out.println(n + "/" + d + " does not EXIST is
List " + current.num + "/" + current.den);
                else if (current == head)
                        head = head.link;
                else {
                        previous.link = current.link;
                }
        }
```

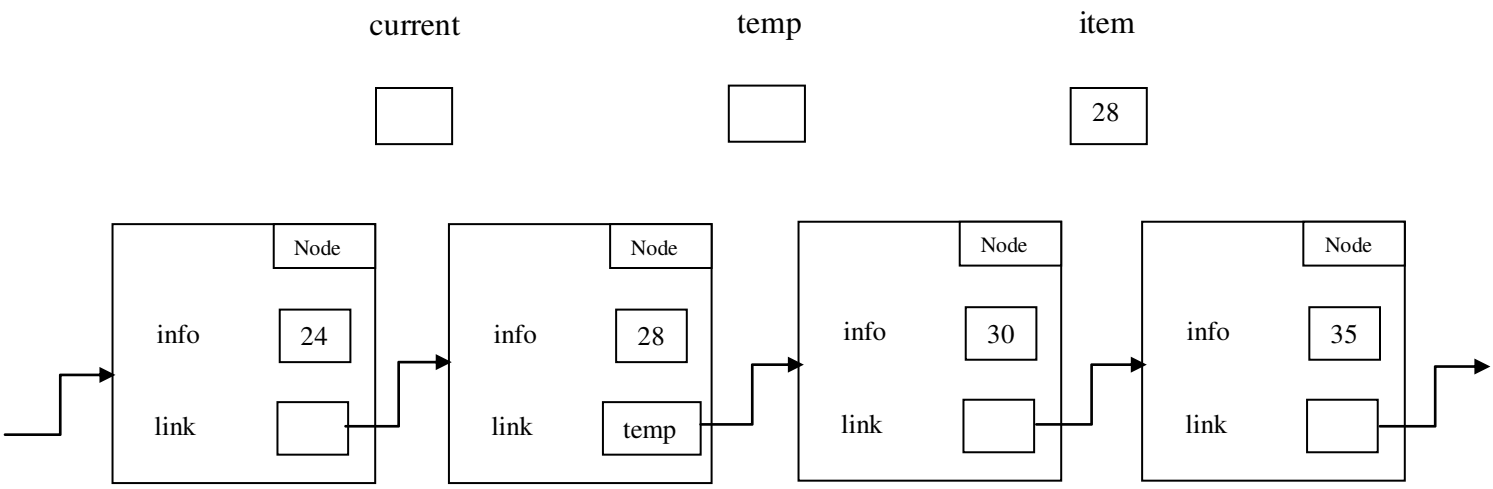
Section 2:

1. The diagram shows a portion of a linked list. In the diagram, both `current` and `temp` are references to objects of the type `Node` that we have been studying while `item` is of type `int`.



Draw a diagram, similar to the one given here, illustrating the result of executing the following fragment.

```
temp = new Node(current.info,current.link);
current.info item;
current.link = temp;
```



2. Write an instance method called `contains` for the `List` class (from the previous exercise). The method should have header

```
public boolean contains (int item)
```

The method should return `true` if and only if its implicit `List` object contains `item`.

```
        /*
         * This method returns true only when its implicit List object
contains item.
         * pre: item
         * post = true (if it contains item) or false (Does not contain
item)
        */
        public boolean contains (int item) {
                boolean located = false;
                for (Node temp = head; temp != null; temp = temp.link) {
                        if (temp.info == item) {
                                located = true;
                                break;
                        }
                }
                return located;

        }
```

3. In Chapter 10, we developed a fast-searching method (binary search) that took advantage of the fact that a sorted array can be searched much more efficiently than an unsorted one. If the nodes of a linked list are ordered, can we write a form of binary search for a linked list? Justify your answer.

   If the nodes of a linked list are ordered, we can write a form of binary search for a linked list by:

   - Finding the length of the list to find the median Node of the list.

- If the median value is **greater** than the item being searched, then the code should disregard all the Nodes that contain **greater** values then the median Node. Hence, we only analyze the Nodes between head and the median.
- If the median value is **less** than the item being searched, then the code should disregard all the Nodes that contain **smaller** values than the median Node. Hence, we only analyze the Nodes from median to the end of the List.
- This process goes until we find the item being searched or until we go through the entire List.

4. Write an instance method `deleteAll` for the `List` class. The method should have one `int` parameter, item. It should delete all nodes in the list that contain `item` in their `info` fields.

```
/*
     * This method deletes all Nodes in the List that contain item in
their info fields.
     * pre: item
     * post: none
    */
    public void deleteAll (int item) {
           Node previous = null;
           Node current = head;
           while (current != null) {
                  if (item == head.info) {
                         head = head.link;
                         current = current.link;
                  }
                  else if (item == current.info) {
                         previous.link = current.link;
                         current = current.link;
                  }
                  else {
                         previous = current;
                         current = current.link;
                  }
           }

    }
```

5. Complete the definition of the method whose header is

```
public boolean isOrderedIncreasing ()
```

The method should return true if and only if the `info` fields of the implicit `List` object are in strictly increasing order. Assume that the `info` fields are references to objects for which a `compareTo` method exists.

```
    /*
      * This method returns true only if the info fields of the List
```

```
object are in strictly increasing order
        * pre: none
        * post: returns true or false
       */
   public boolean isOrderedlncreasing () {
          boolean increasing = true;
          Node current = head;
          while (current.link != null) {
                 if (current.info > current.link.info) {
                        increasing = false;
                        break;
                    }
                 else {
                        current = current.link;
                 }
          }
          return increasing;

   }
```

6. The technique shown in Question 1 can be used to insert a value into an ordered list without using two auxiliary references. Write an instance method that uses this technique to achieve the same effect as the method shown in Example 1. (Be sure that your method works at both ends of the list.)

```
      /*
       * This method inserts an item in List in the correct order
       *  pre: item
       *  post: none
      */
      public void Insert (int item) {
          Node current = head;
          boolean largest = true;
          while (current.link != null) {
                 if (item < current.info) {
                        largest = false;
                        break;
                 }
                 else
                        current = current.link;

          }

          if (largest == true) {
                 Node temp = new Node (item, null);
                 current.link = temp;
          }
          else {
                 Node temp = new Node (current.info, current.link);
                 current.info = item;
                 current.link = temp;
          }
   }
```

7. Write an instance method `isldentical` for the `List` class. The method should have the header

```
boolean isldentical (List other)
```

The method should return `true` if and only if its implicit `List` object is identical to other.

```
   /*
    * This method returns true only if its implicit List object is identical
to other
    * pre: List other
    * post: true (identical) or false (not identical)
   */
   boolean isldentical (List1 other) {
          boolean identical = true;
          Node current = this.head;
          Node current1 = other.head;
          while (current != null && current1 != null ) {
                if (current.info != current1.info) {
                      identical = false;
                      break;
                }
                else {
                      current = current.link;
                      current1 = current1.link;
                }
          }
          return identical;
   }
```