4 ΔΥΝΑΜΙΚΕΣ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Υπάρχει ένας αριθμός προγραμματιστικών προβλημάτων, τα οποία είναι δύσκολο να επιλυθούν με τη βοήθεια των στατικών δομών δεδομένων που αναφέραμε μέχρι τώρα. Αυτό γιατί οι δομές αυτές είναι προκαθορισμένες και παραμένουν σταθερές σε όλη τη διάρκεια της εκτέλεσης του προγράμματος. Για παράδειγμα εάν θεωρήσουμε την παρακάτω δήλωση της μεταβλητής **A**.

```
int[ ] A = new int[1000];
```

Ο αριθμός των στοιχείων του πίνακα **A** που είναι διαθέσιμα κατά τη διάρκεια εκτέλεσης του προγράμματος θα είναι ακριβώς 1000, κάτι που προκαθορίζεται όταν το πρόγραμμα μεταγλωττίζεται. Επί πλέον ο χώρος μνήμης που απαιτείται για την αποθήκευση των στοιχείων αυτών είναι επίσης προκαθορισμένος. Η κατάσταση αυτή δεν αποτελεί πρόβλημα στην περίπτωση που γνωρίζουμε από την αρχή πόσα στοιχεία θα χρειαστούμε, αυτό όμως τις περισσότερες φορές δεν είναι εφικτό.

Ένα δεύτερο πρόβλημα δημιουργείται από το γεγονός ότι τα στοιχεία του πίνακα θεωρούνται συνεχόμενα. Έτσι εάν χρειαστεί να παρεμβάλουμε ένα στοιχείο ανάμεσα σε άλλα δύο (ανάγκη που προκύπτει, όταν θέλουμε για παράδειγμα να εισάγουμε μία τιμή σε έναν ταξινομημένο πίνακα χωρίς να αλλοιώσουμε την ιδιότητά του αυτή) πρέπει να καταφύγουμε στη χρονοβόρα διαδικασία αντιγραφής ενός μεγάλου πλήθους στοιχείων σε άλλη περιοχή του πίνακα. Η λύση στα προβλήματα αυτά μπορεί να δοθεί με τη χρήση των δυναμικών δομών δεδομένων.

Μια δομή δεδομένων η οποία μπορεί να δημιουργηθεί και να μετατρέπεται στη διάρκεια της εκτέλεσης του προγράμματος αυξομειώνοντας το μέγεθός της ή το χώρο που καταλαμβάνει στη μνήμη ονομάζεται **δυναμική δομή δεδομένων (*dynamic data structure*)**. Μία τέτοια δυναμική δομή δεδομένων είναι η **συνδεδεμένη λίστα (*linked list*)**, που εξετάζουμε στη συνέχεια.

4.1 ΛΙΣΤΕΣ

Λίστα (*list*) είναι ένα διατεταγμένο σύνολο από 0 ή περισσότερα στοιχεία τα οποία, κατά κανόνα, είναι όλα του ίδιου τύπου. Μία λίστα διαφέρει από έναν πίνακα στο ότι το μέγεθός της είναι γενικά μεταβλητό. Συχνά παριστάνουμε μία λίστα παραθέτοντας τα στοιχεία της ως εξής:

$a_1, a_2, \ldots, a_n$

όπου **n >= 0** και κάθε a_i είναι κάποιου συγκεκριμένου τύπου. Ο αριθμός των στοιχείων της λίστας ονομάζεται **μήκος (*length*)** της λίστας. Θεωρώντας ότι **n >= 1**, λέμε ότι το a_1 είναι το πρώτο στοιχείο της λίστας και το a_n το τελευταίο. Εάν **n = 0**, τότε έχουμε την **κενή** λίστα (μία λίστα που δεν έχει κανένα στοιχείο).

4.1.1 Υλοποίηση Λίστας με τη Βοήθεια Δεικτών

Μια λίστα μπορεί να θεωρηθεί σαν μία συλλογή από κόμβους, οι οποίοι είναι συνδεδεμένοι μεταξύ τους. Κάθε κόμβος μπορεί να υλοποιηθεί με τη βοήθεια μιας δυάδας πεδίων. Το πρώτο πεδίο περιλαμβάνει ένα στοιχείο της λίστας (την πληροφορία που σχετίζεται με το στοιχείο αυτό) και το δεύτερο ένα δείκτη ο οποίος δείχνει στον επόμενο κόμβο της λίστας. Τα δύο αυτά πεδία μπορούν να οριστούν στα πλαίσια της κλάσης **Node**, η οποία δίνεται στη συνέχεια. Η δημιουργία ενός καινούργιου κόμβου μπορεί να περιλαμβάνει αρχικά την περίπτωση ενός κενού κόμβου, ο οποίος δεν έχει καμία πληροφορία στο πεδίο **item** (= **null**) και δεν είναι συνδεδεμένος με κανέναν άλλο κόμβο (**next** = **null**). Μπορεί όμως να περιλαμβάνει την περίπτωση που και τα δύο αυτά πεδία έχουν συγκεκριμένες τιμές κατά τη δημιουργία τους.

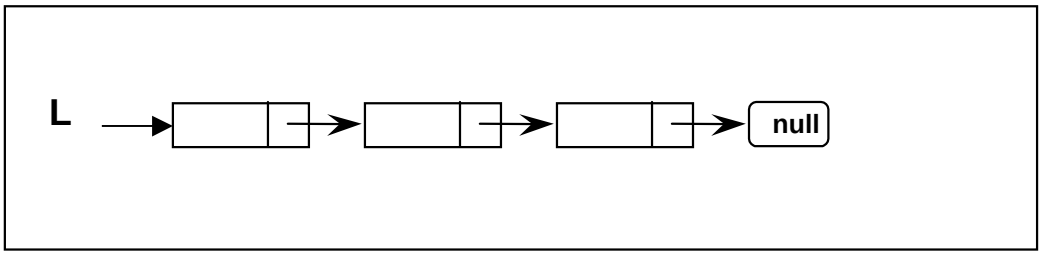
```
class Node

    {   Object item;
        Node next;

        public Node( ) {
            this(null,null);
        }

        public Node(Object it, Node n) {
            item = it;
            next = n;
        }
    .....
    }
```

Οι λίστες που υλοποιούνται με αυτό τον τρόπο ονομάζονται **συνδεδεμένες λίστες *(linked lists)***. Η σχηματική αναπαράσταση μίας τέτοιας λίστας η οποία περιλαμβάνει τρεις κόμβους δίνεται στο σχήμα 4.1:



Σχήμα 4.1 : Συνδεδεμένη λίστα με τρεις κόμβους

Πρέπει να τονιστεί ότι στις συνδεδεμένες λίστες δεν υπάρχει άμεσος τρόπος πρόσβασης σε έναν οποιονδήποτε κόμβο της. Ο μόνος που «γνωρίζει» την ύπαρξη ενός κόμβου είναι ο προηγούμενός του, καθώς γνωρίζει τη διεύθυνση του σαν τιμή που είναι αποθηκευμένη στο πεδίο του **next**. Με την έννοια αυτή για να έχουμε πρόσβαση σε έναν κόμβο πρέπει να περάσουμε ακολουθιακά από όλους τους προηγούμενους αρχίζοντας από τον πρώτο.

Ο πρώτος κόμβος της λίστας πρέπει οπωσδήποτε να «δεικνύεται» από μία μεταβλητή τύπου **Node** (η μεταβλητή **L** στο παράδειγμα του σχήματος 4.1). Η μεταβλητή αυτή ονομάζεται **δείκτης** (***pointer***). Στην περίπτωση της Java ο δείκτης αυτός δεν είναι

τίποτε άλλο από μία μεταβλητή τύπου αναφοράς (*reference*), μία μεταβλητή που αναπαριστά ένα αντικείμενο της κλάσης **Node**. Ο δείκτης στον πρώτο κόμβο χαρακτηρίζει τη συνδεδεμένη λίστα σαν δομή δεδομένων και πρέπει να παραμένει εκεί σε όλη τη διάρκεια εκτέλεσης του προγράμματος που τη χειρίζεται.

Δείκτες είναι επίσης όλες οι τιμές των πεδίων **next** όλων των κόμβων της συνδεδεμένης λίστας: δείχνουν στον επόμενο κόμβο. Το πεδίο του τελευταίου κόμβου έχει την τιμή **null** (δείχνει στο **null**). Στο παράδειγμα του σχήματος 4.1:

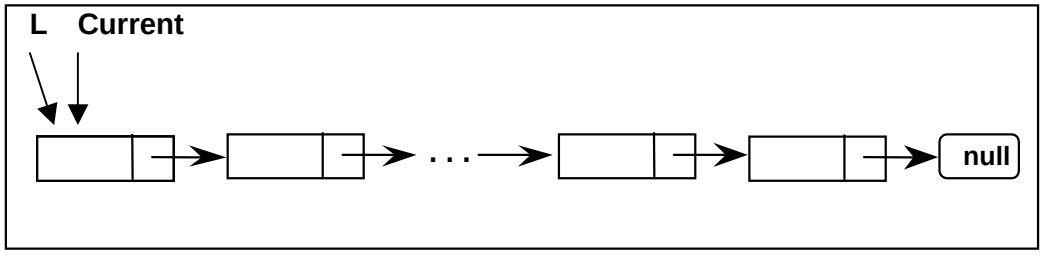
L.next είναι ο δείκτης που δείχνει στο δεύτερο κόμβο και
L.next.next είναι ο δείκτης που δείχνει στον τρίτο κόμβο

Οι πιο βασικές λειτουργίες που απαιτούνται για το χειρισμό μιας συνδεδεμένης λίστας ανήκουν σε μία από τις παρακάτω τρεις κατηγορίες:

- Επίσκεψη κάποιου κόμβου της συνδεδεμένης λίστας
- Προσθήκη ενός κόμβου στη συνδεδεμένη λίστα
- Διαγραφή ενός κόμβου από τη συνδεδεμένη λίστα

Πρόσβαση σε κάποιο κόμβο της λίστας:
Για να επισκεφθούμε κάποιον συγκεκριμένο κόμβο μίας λίστας (για να έχουμε πρόσβαση σ' αυτόν) εργαζόμαστε ως εξής:

- Αρχικά τοποθετούμε έναν βοηθητικό δείκτη ο οποίος δείχνει στην αρχή της λίστας (σχ. 4.2α)
  Current = L;

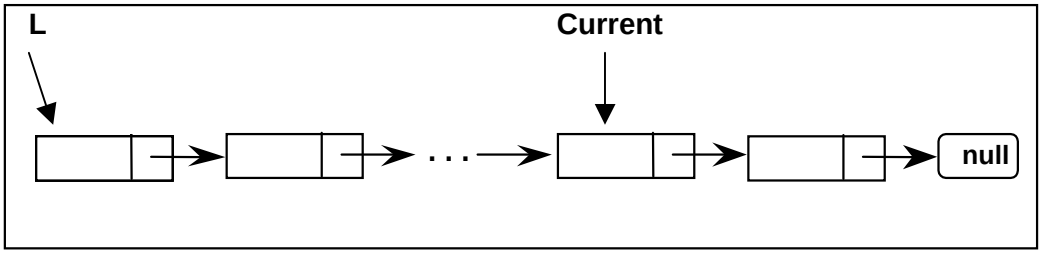


Σχήμα 4.2α : Ο βοηθητικός δείκτης **Current** δείχνει στην αρχή της λίστας

- Στη συνέχεια μετακινούμε τον δείκτη **Current** στον επόμενο κόμβο
  Current = Current.next; ή ισοδύναμα (βλέπε κλάση **Node** στο τμήμα κώδικα 1)

Current = Current.getNext( );

- Η παραπάνω διαδικασία μετακίνησης στον επόμενο κόμβο γίνεται επαναληπτικά όσες φορές χρειάζεται, ώστε να τοποθετηθούμε στον επιθυμητό κόμβο (σχ. 4.2β).



Σχήμα 4.2β : Ο βοηθητικός δείκτης **Current** δείχνει στον επιθυμητό κόμβο

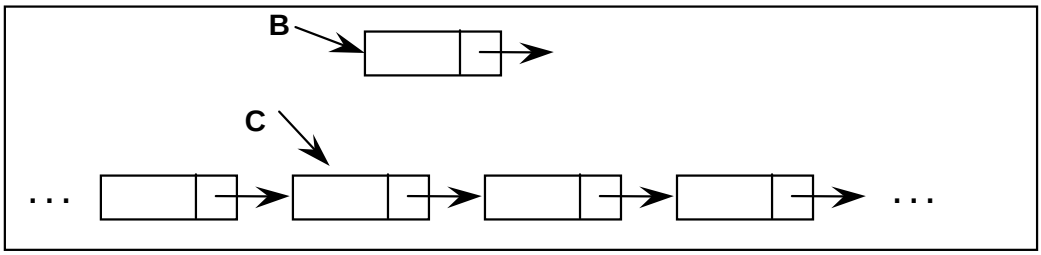
Εισαγωγή ενός νέου κόμβου στη λίστα:

Για να τοποθετήσουμε έναν κόμβο σε μία συνδεδεμένη λίστα πρέπει να γνωρίζουμε τον προηγούμενό του ή ισοδύναμα να τον δείχνουμε με τη βοήθεια ενός δείκτη τύπου **Node**, έστω ***C*** στο παράδειγμα του σχήματος 4.3α. Στη συνέχεια η εισαγωγή ακολουθεί τα εξής βήματα:

- Αρχικά δημιουργούμε έναν καινούργιο κόμβο (σχ.4.3α)

  B = new Node(T, null)

  όπου **T** η τιμή που θέλουμε να έχει το πεδίο πληροφορίας **item** του κόμβου.

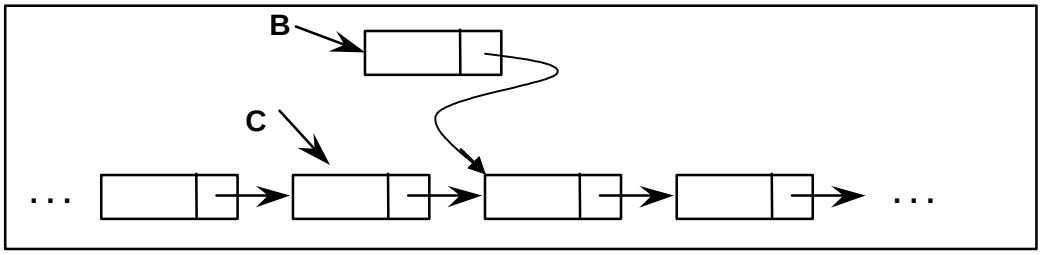


Σχήμα 4.3α : Δημιουργία του κόμβου **B** προς εισαγωγή

- Στη συνέχεια συνδέουμε τον κόμβο αυτό με τον επόμενο του ***C*** (σχ. 4.3β)

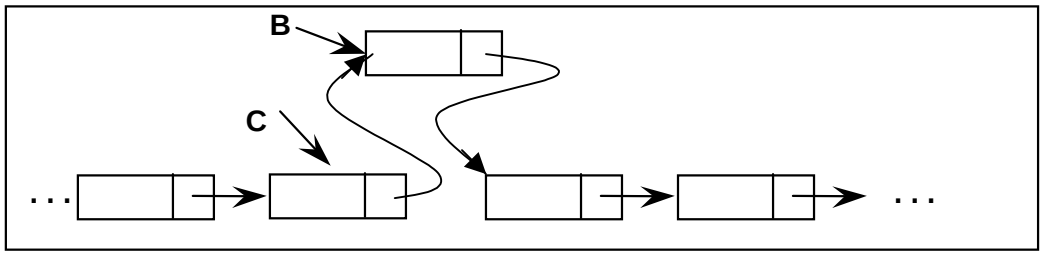
  B.next = *C*. next; ή ισοδύναμα (βλέπε κλάση Node στο τμήμα κώδικα 1)

  B.setNext(*C*.getNext( ));



Σχήμα 4.3β : Ο επόμενος του κόμβου **B** γίνεται ο επόμενος του **C**

- Τέλος κάνουμε επόμενο του κόμβου ***C*** τον κόμβο **B** (σχ. 4.3γ)
  ***C*.next** = **B;** ή ισοδύναμα ***C*.setNext(B);**



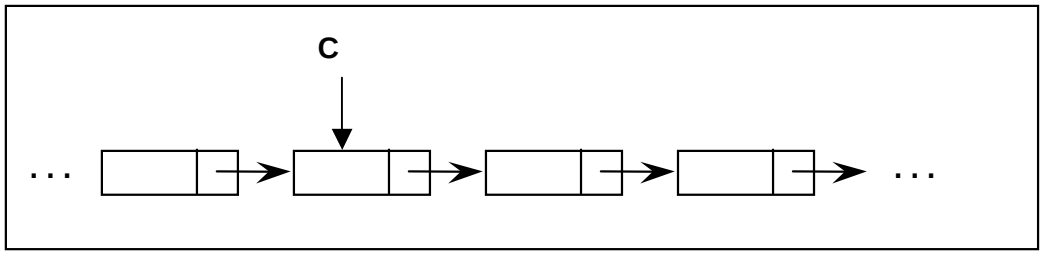
Σχήμα 4.3γ : Ο επόμενος του κόμβου **C** γίνεται ο νέος κόμβος **B**

Διαγραφή ενός κόμβου από τη λίστα:

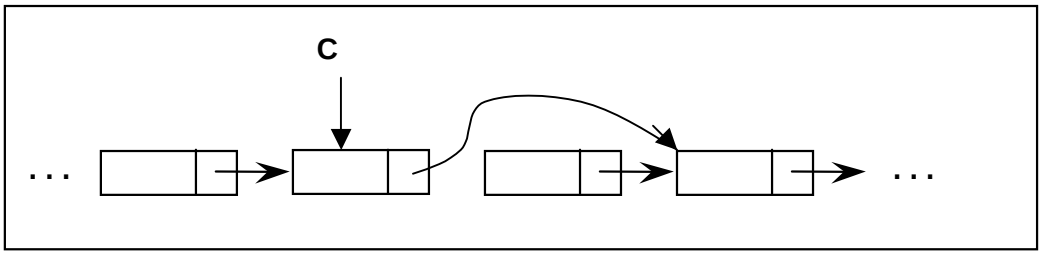
Για να διαγράψουμε έναν κόμβο από μία συνδεδεμένη λίστα πρέπει επίσης, όπως και στην περίπτωση της εισαγωγής, να γνωρίζουμε τον προηγούμενό του ή ισοδύναμα να τον δείχνουμε με τη βοήθεια ενός δείκτη τύπου **Node**, έστω ***C*** στο παράδειγμα του σχήματος 4.4α. Στη συνέχεια διαγράφουμε τον επόμενο κόμβο του *C* παρακάμπτοντάς τον ως εξής:

***C*.next** = ***C*.next.next;** ή ισοδύναμα ***C*.setNext(*C*.getNext( ).getNext( ))**

Η κατάσταση της λίστας πριν και μετά τη διαγραφή φαίνεται στα σχήματα 4.4α και 4.4β.

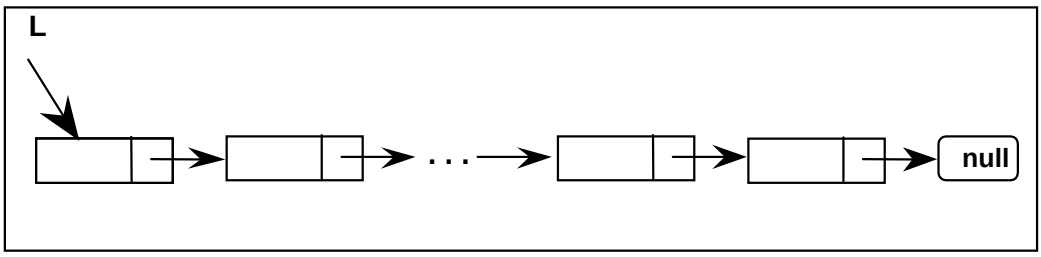


Σχήμα 4.4α : Πριν τη διαγραφή του επόμενου κόμβου του **C**



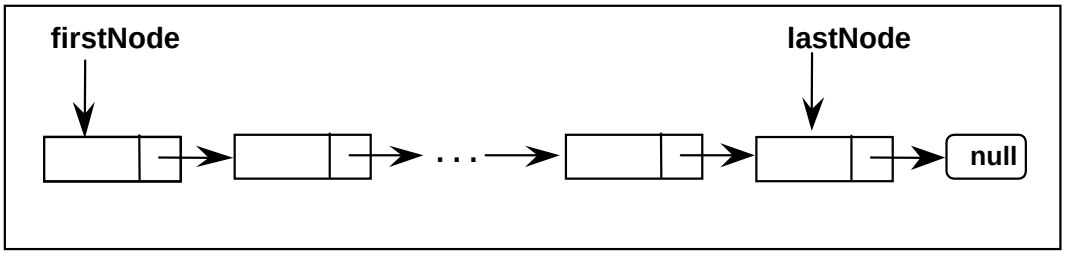
Σχήμα 4.4β : Μετά τη διαγραφή του επόμενου κόμβου του **C**

Στα σχήματα 4.5α ,β και γ φαίνονται τρεις διαφορετικοί τρόποι αναπαράστασης μιας συνδεδεμένης λίστας. Οι δύο πρώτοι αναφέρονται στην απλά συνδεδεμένη λίστα, που έχουμε συζητήσει.



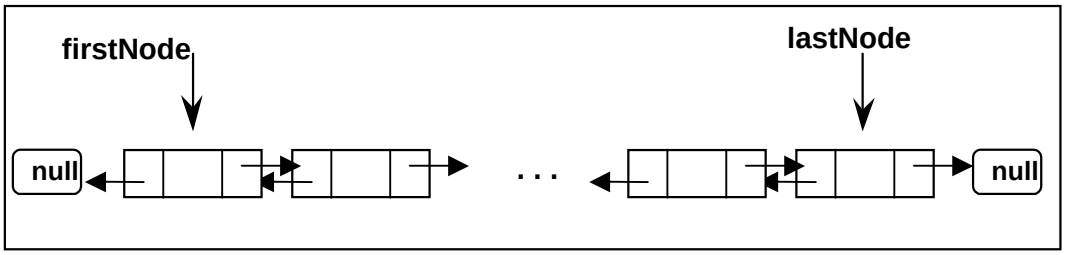
Σχήμα 4.5α : Απλά συνδεδεμένη λίστα με πρόσβαση στην αρχή

Στη δεύτερη περίπτωση φροντίζουμε επιπλέον να διατηρούμε έναν δείκτη στον τελευταίο κόμβο, για να έχουμε έτσι ευκολότερη πρόσβαση σε αυτόν.



Σχήμα 4.5β : Απλά συνδεδεμένη λίστα με πρόσβαση στην αρχή και στο τέλος

Στην τρίτη περίπτωση αναφερόμαστε στη **διπλά συνδεδεμένη λίστα**, όπου κάθε κόμβος της διαθέτει ένα επιπλέον πεδίο τύπου Node το οποίο δείχνει στον προηγούμενο κόμβο της λίστας.



Σχήμα 4.5γ : Διπλά συνδεδεμένη λίστα

Στη συνέχεια δίνεται η υλοποίηση της συνδεδεμένης λίστας του σχήματος 4.5β: Για τη λίστα αυτή ορίζουμε δύο δείκτες **firstNode** και **lastNode** που δείχνουν αντίστοιχα στον πρώτο και τελευταίο κόμβο της. Στην περίπτωση της κενής λίστας θεωρούμε ότι οι δύο αυτοί δείκτες δείχνουν στο **null**.

Υπεύθυνη για τη διαχείριση της συνδεδεμένης λίστας είναι η κλάση **LinkedList**, στα πλαίσια της οποίας ορίζονται οι πράξεις:

- **insertFirst**: εισάγει ένα στοιχείο στην αρχή της λίστας
- **insertLast**: εισάγει ένα στοιχείο στο τέλος της λίστας
- **removeFirst**: διαγράφει και επιστρέφει το πρώτο στοιχείο της λίστας
- **removeLast**: διαγράφει και επιστρέφει το τελευταίο στοιχείο της λίστας
- **isEmpty**: ελέγχει εάν η λίστα είναι κενή
- **size**: επιστρέφει το μέγεθος (αριθμό κόμβων) της λίστας
- **printList**: εκτυπώνει όλα τα στοιχεία της λίστας

Στο τμήμα κώδικα 1 δίνεται η αναπαράσταση/υλοποίηση ενός κόμβου της συνδεδεμένης λίστας και στο τμήμα κώδικα 2 δίνεται η κλάση που υλοποιεί τη συνδεδεμένη λίστα. Θεωρούμε ότι οι κλάσεις **Node** και **LinkedList** βρίσκονται στο ίδιο πακέτο. Με την έννοια αυτή η κλάση LinkedList έχει πρόσβαση στα πεδία **item** και **next** της **Node**.

Εναλλακτικά, και για μια πιο ασφαλή υλοποίηση, θα έπρεπε:

- Τα πεδία **item** και **next** της κλάσης **Node** να δηλωθούν **private** και
- Στα πλαίσια της κλάσης LinkedList να γίνεται χρήση των πεδίων αυτών με τη βοήθεια των μεθόδων **getNext**, **setNext**, **getItem** και **setItem**.

```
class Node
//Ορισμός κόμβου μιας απλά συνδεδεμένης λίστας (linked list)
{       Object item;
        Node next;

        public Node( ) {
            this(null,null);
        }

        public Node(Object it, Node n) {
            item = it;
            next = n;
        }

        public void setItem(Object newItem) {
            item = newItem;
        }

        public void setNext(Node newNext) {
            next = newNext;
        }

        public Object getItem( ) {
            return(item);
        }

        public Node getNext( ) {
            return(next);
        }
}
```

Τμήμα Κώδικα 1

```
import java.io.*;

public class LinkedList

//Υλοποίηση μιάς απλά συνδεδεμένης λίστας (linked list)
{
      private Node firstNode, lastNode;

      public LinkedList( ) {
          firstNode = lastNode = null;
      }

      public Node getFirst( ) {
        return firstNode;
      }

      public Node getLast( ) {
        return lastNode;
      }

      public boolean isEmpty( ) {
          return(firstNode == null);
      }

      public void insertFirst(Object newItem) {
          if (isEmpty( ))
              firstNode = lastNode = new Node(newItem, null);
          else
              firstNode = new Node(newItem, firstNode);
      }

      public void insertLast(Object newItem) {
          if (isEmpty( ))
              firstNode = lastNode = new Node(newItem, null);
          else
              lastNode = lastNode.next = new Node(newItem, null);
      }
```

```
public Object removeFirst( ) throws ListEmptyException
{
    Object removeItem;
    if (isEmpty( ))
        throw new ListEmptyException("Empty List!!!");
    removeItem = firstNode.item;
    if (firstNode == lastNode)
        firstNode = lastNode = null;
    else
        firstNode = firstNode.next;
    return removeItem;
}

public Object removeLast( ) throws ListEmptyException
{
    Object removeItem;
    if (isEmpty( ))
        throw new ListEmptyException("Empty List!!!");
    removeItem = lastNode.item;
    if (firstNode == lastNode)
        firstNode = lastNode = null;
    else
    {   Node current = firstNode;
        while (current.next != lastNode)
            current = current.next;
        lastNode = current;
        current.next = null;
    }
    return removeItem;
}

public void printList( )
{
    if (isEmpty( ))
        System.out.println("Empty List");
    else
    {   Node current = firstNode;
        while (current != null)
        {   System.out.print(current.item.toString() + " ");
            current = current.next;
        }
```

```
                System.out.println("\n");
            }
        }
}
-------------
public class ListEmptyException extends RuntimeException
{
        public ListEmptyException(String err)
        { super(err); }
}
```

Τμήμα Κώδικα 2

Στο τμήμα κώδικα 3 δίνεται η υλοποίηση της μεθόδου για την ταξινόμηση των στοιχείων μιας συνδεδεμένης λίστας (θεωρήστε ότι ο τύπος του item είναι String).

```
public class ListToSort extends LinkedList
{
        public ListToSort() { super(); }

        public LinkedList SortList() {
            Node trace, current, min;

            trace = getFirst();
            while (trace!=null)
                {   current = trace;
                    min = trace;
                    while (current!=null)
                    {   if ((current.getItem()).compareTo(min.getItem())<0)
                            min=current;
                        current = current.getNext();
                    } //endwhile current
                String temp = trace.getItem();
                trace.setItem(min.getItem());
                min.setItem(temp);
                trace = trace.getNext();
                }//endwhile trace
            return this;
        }
}
```

Τμήμα Κώδικα 3

4.2 Υλοποίηση Στοίβας και Ουράς με τη βοήθεια συνδεδεμένης λίστας

Η υλοποίηση των δομών δεδομένων στοίβα και ουρά με τη βοήθεια ενός πίνακα είχε το μειονέκτημα της συγκεκριμένης χωρητικότητας η οποία πρέπει να επιλέγεται κατά στιγμή της δημιουργίας. Στο τμήμα κώδικα 3 φαίνεται ο τρόπος με τον οποίο μπορεί να χρησιμοποιηθεί η δομή δεδομένων συνδεδεμένη λίστα για την υλοποίηση μίας συνδεδεμένης στοίβας (linked stack). Η υλοποίηση της στοίβας σε αυτή την περίπτωση γίνεται με την τεχνική της σύνθεσης (composition) καθώς χρησιμοποιείται μία άλλη δομή τοπικά στην κλάση. Η τεχνική αυτή ονομάζεται και εξουσιοδότηση (delegation)

```
public class LinkedStack implements Stack
// Υλοποίηση στοίβας με την τεχνική της σύνθεσης (composition)
// Η τεχνική αυτή ονομάζεται και Εξουσιοδότηση (Delegation)

 {
        private LinkedList S;

        public LinkedStack( ) {
          S=new LinkedList();
        }

        public int size() {
          return S.size();
        }

        public boolean isEmpty() {
           return S.isEmpty();
        }

        public Object top( ) throws StackEmptyException{
          Object temp;
          if (S.isEmpty()) throw new StackEmptyException("Empty Stack!");
          temp = S.removeFirst();
          S.insertFirst(temp);
          return temp;
        }
```

```
        public void push(Object item) {
          S.insertFirst(item);
        }

        public Object pop( ) throws StackEmptyException {
          try {
                return S.removeFirst();
          }
          catch (ListEmptyException str) {
                throw new StackEmptyException("EmptyStack");
          }
        }
}
```

Τμήμα κώδικα 4

Στο τμήμα κώδικα 5 δίνεται μία εναλλακτική υλοποίηση της στοίβας όπου χρησιμοποιείται μόνον η κλάση Node.

```
import Node;
public class LinkedStack2 implements Stack
// Υλοποίηση της στοίβας με τη χρήση μόνον της κλάσης Node
{
        private Node top;
        private int size;

        public LinkedStack2( ) {
          top = null;
          size = 0;
        }

        public int size( ) {
          return size;
        }

        public boolean isEmpty( ) {
           return (top == null);
        }
```

```
    public Object top( ) throws StackEmptyException {
      if (isEmpty( )) throw new StackEmptyException("Empty Stack!");
      return top.getItem();
    }

    public void push(Object item) {
      Node newTop = new Node( );
      newTop.setItem(item);
      newTop.setNext(top);
      top = newTop;
      size++;
    }

    public Object pop( ) throws StackEmptyException {
      try {
            Object temp = top.getItem();
            top = top.getNext();
            size--;
            return temp;
      }
      catch (ListEmptyException str) {
            throw new StackEmptyException("EmptyStack");
      }
    }
}
```

Τμήμα κώδικα 5

Στο τμήμα κώδικα 6 δίνεται η υλοποίηση της στοίβας σαν επέκταση της κλάσης LinkedList με τη μέθοδο της κληρονομικότητας.

```
public class LinkedStack3 extends LinkedList implements Stack

// Υλοποίηση της στοίβας με τη μέθοδο της κληρονομικότητας
// από την κλάση LinkedList

// Ερώτηση: Τί γίνεται με τις άλλες μεθόδους της LinkedList?
```

```
{
        public LinkedStack3( )
        { super( ); }

        public int size() {
          return super.size();
        }

        public boolean isEmpty() {
           return super.isEmpty();
        }

        public Object top( ) throws ListEmptyException {
          Object temp;
          temp = removeFirst();
          insertFirst(temp);
          return temp;
        }

        public void push(Object item) {
          insertFirst(Object);
        }

        public Object pop( ) throws ListEmptyException {
          return removeFirst();
        }

}
```

Τμήμα κώδικα 6

Στο τμήμα κώδικα 7 δίνεται η υλοποίηση της δομής δεδομένων ουρά με τη βοήθεια μίας συνδεδεμένης λίστας. Χρησιμοποιείται η τεχνική της σύνθεσης. Με αντίστοιχους τρόπους με αυτούς της στοίβας μπορεί αντίστοιχα να υλοποιηθεί και η ουρά με διαφορετικούς τρόπους.

```
public class LinkedQueue  implements Queue {
// Υλοποίηση ουράς με την τεχνική της σύνθεσης (composition)

        private LinkedList Q;

        public LinkedQueue( ) {
          Q=new LinkedList();
        }

        public int size() {
          return Q.size();
        }

        public boolean isEmpty() {
           return Q.isEmpty();
        }

        public Object front( ) throws QueueEmptyException {
          Object temp;
          if (Q.isEmpty()) throw new QueueEmptyException("Empty Queue!");
          temp = Q.removeFirst();
          Q.insertFirst(temp);
          return temp;
        }

        public void enqueue(Object item) {
          Q.insertLast(item);
        }

        public Object dequeue( ) throws QueueEmptyException {
          try {
                return Q.removeFirst();
          }
          catch (ListEmptyException str) {
                throw new QueueEmptyException("EmptyQueue");
          }
        }
}
```

Τμήμα κώδικα 7