Unidad III - Listas

- Lista de Enlace Simple
- Los Algoritmos de Concatenación e Inversión
- Lista Doblemente Enlazada
  - Algoritmo de Inserción-Ordenada
- Lista de Enlace Circular
- Listas Enlazadas frente a Arrays

- Java en castellano (Tutorial Estructuras de Datos y Algoritmos en Java)
  http://www.programacion.com/java/tutorial/jap_data_alg/
- Tutorial original en Inglés
  http://www.javaworld.com/

Listas Enlazadas

Además de los arrays, otra de las estructuras de datos muy utilizadas es la lista enlazada. Esta estructura implica cuatro conceptos: clase auto-refenciada, nodo, campo de enlace y enlace.

- **Clase auto-referenciada:** una clase con al menos un campo cuyo tipo de referencia es el nombre de la clase:

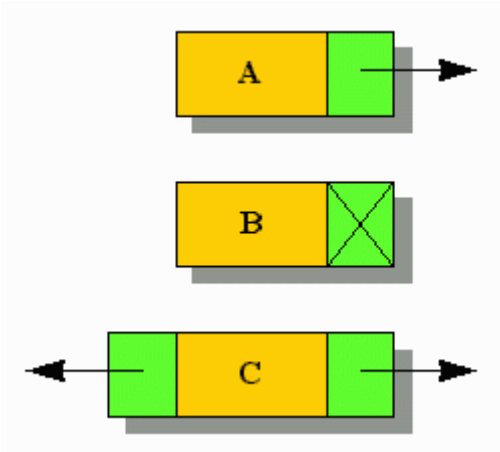
```
class Employee {
   private int empno;
   private String name;
   private double salary;

   public Employee next;

   // Other members

}
```

`Employee` es una clase auto-referenciada porque su campo `next` tiene el tipo `Employee`.

- **Nodo:** un objeto creado desde una clase auto-referenciada.
- **Campo de enlace:** un campo cuyo tipo de referencia es el nombre de la clase. En el fragmento de código anterior, `next` es un campo de enlace. Por el contrario, `empno`, `name`, y `salary` son campos no de enlace.
- **Enlace:** la referencia a un campo de enlace. En el fragmento de código anterior, la referencia `next` a un nodo `Employee` es un enlace.

Los cuatro conceptos de arriba nos llevan a la siguiente definición: *una **lista enlazada** es una secuencia de nodos que se interconectan mediante sus campos de enlace*. En ciencia de la computación se utiliza una notación especial para ilustrar las listas enlazadas. En la siguiente imagen aparece una variante de esta notación que utilizaré a lo largo de esta sección:

- Java en castellano (Tutorial Estructuras de Datos y Algoritmos en Java)
  http://www.programacion.com/java/tutorial/jap_data_alg/
- Tutorial original en Inglés
  http://www.javaworld.com/

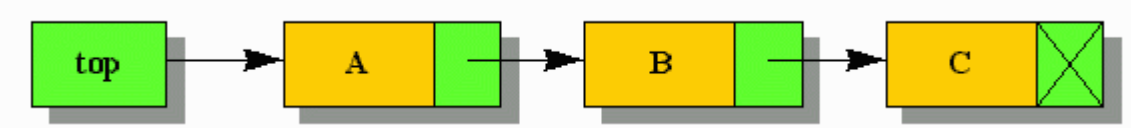


La figura anterior presenta tres nodos: A, B y C. Cada nodo se divide en áreas de contenido (en naranja) y una o más áreas de enlace (en verde). Las áreas de contenido representan todos los campos que no son enlaces, y cada área de enlace representa un campo de enlace. Las áreas de enlace de A y C tienen unas flechas para indicar que referencian a otro nodo del mismo tipo (o subtipo). La única área de enlace de B incorpora una X para indicar una referencia nula. En otras palabras, B no está conectado a ningún otro nodo.

Aunque se pueden crear muchos tipos de listas enlazadas, las tres variantes más populares son la lista de enlace simple, la lista doblemente enlazada y la lista enlazada circular. Exploremos esas variantes, empezando con la lista enlazada.

Lista de Enlace Simple

Una **lista de enlace simple** es una lista enlazada de nodos, donde cada nodo tiene un único campo de enlace. Una variable de referencia contiene una referencia al primer nodo, cada nodo (excepto el último) enlaza con el nodo siguiente, y el enlace del último nodo contiene `null` para indicar el final de la lista. Aunque normalmente a la variable de referencia se la suele llamar `top`, usted puede elegir el nombre que quiera. La siguiente figura presenta una lista de enlace simple de tres nodos, donde `top` referencia al nodo A, A conecta con B y B conecta con C y C es el nodo final:



Un algoritmo común de las listas de enlace simple es la inserción de nodos. Este algoritmo está implicado de alguna forma porque tiene mucho que ver con cuatro casos: cuando el nodo se debe insertar antes del primer nodo; cuando el nodo se debe insertar después del último nodo; cuando el nodo se debe insertar entre dos nodos; y cuando la lista de enlace simple no existe. Antes de estudiar cada caso consideremos el siguiente pseudocódigo:

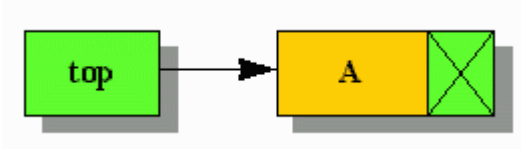
```
DECLARE CLASS Node
  DECLARE STRING name
  DECLARE Node next
END DECLARE

DECLARE Node top = NULL
```

Este pseudocódigo declara una clase auto-referenciada llamada `Node` con un campo no de enlace llamado `name` y un campo de enlace llamado `next`. También declara una variable de referencia `top` (del tipo `Node`) que contiene una referencia al primer `Node` de una lista de enlace simple. Como la lista todavía no existe, el valor inicial de `top` es `NULL`. Cada uno de los siguientes cuatro casos asume las declaraciones de `Node` y `top`:

- **La lista de enlace simple no existe::**
  Este es el caso más simple. Se crea un `Node`, se asigna su referencia a `top`, se inicializa su campo no de enlace, y se asigna `NULL` a su campo de enlace. El siguiente pseudocódigo realiza estas tareas:

```
top = NEW Node

top.name = "A"
top.next = NULL
```

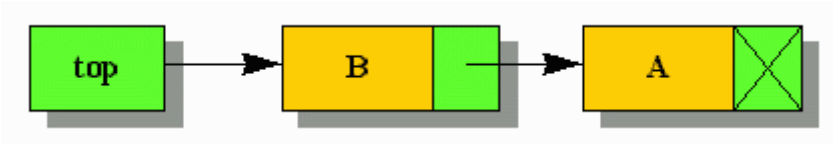
  En la siguiente imagen se puede ver la lista de enlace simple que emerge del pseudocódigo anterior:



- **El nodo debe insertarse antes del primer nodo:**.
  Se crea un `Node`, se inicializa su campo no de enlace, se asigna la referencia de `top` al campo de enlace `next`, y se asigna la referencia del `Node` recién creado a `top`. El siguiente pseudocódigo (que asume que se ha ejecutado el pseudocódigo anterior) realiza estas tareas:

```
DECLARE Node temp

temp = NEW Node
temp.name = "B"
temp.next = top
top = temp
```

  El resultado del listado anterior aparece en la siguiente imagen:



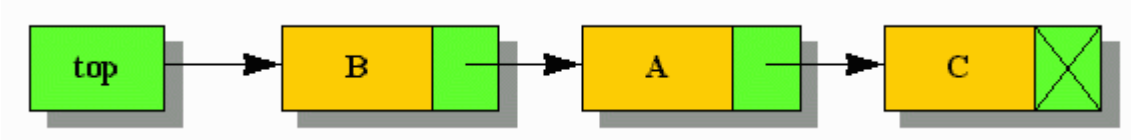
- **El nodo debe insertarse detrás del último nodo:**
  Se crea un `Node`, se inicializa su campo no de enlace, se asigna `NULL` al campo de enlace, se atraviesa la lista de enlace simple hasta el último `Node`, y se asigna la referencia del `Node` recién creado al campo `next` del último nodo. El siguiente pseudocódigo realiza estas tareas:

```
temp = NEW Node
temp.name = "C"
temp.next = NULL

DECLARE Node temp2

temp2 = top

// We assume top (and temp2) are not NULL
// because of the previous pseudocode

WHILE temp2.next IS NOT NULL
   temp2 = temp2.next
END WHILE
```

- Java en castellano (Tutorial Estructuras de Datos y Algoritmos en Java)
  http://www.programacion.com/java/tutorial/jap_data_alg/
- Tutorial original en Inglés
  http://www.javaworld.com/

```
// temp2 now references the last node

temp2.next = temp
```

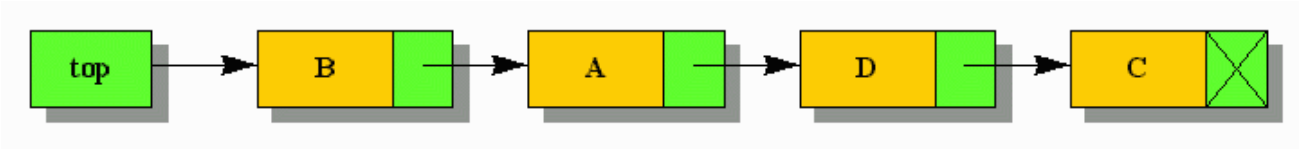
La siguiente imagen revela la lista después de la inserción del nodo `C` después del nodo `A`.



- **El nodo se debe insertar entre dos nodos:**
  Este es el caso más complejo. Se crea un `Node`, se inicializa su campo no de enlace, se atraviesa la lista hasta encontrar el `Node` que aparece antes del nuevo `Node`, se asigna el campo de enlace del `Node` anterior al campo de enlace del `Node` recién creado, y se asigna la referencia del `Node` recién creado al campo del enlace del `Node` anterior. El siguiente pseudocódigo realiza estas tareas:

```
temp = NEW Node
temp.name = "D"

temp2 = top

// We assume that the newly created Node is inserted after Node
// A and that Node A exists. In the real world, there is no
// guarantee that any Node exists, so we would need to check
// for temp2 containing NULL in both the WHILE loop's header
// and after the WHILE loop completes.

WHILE temp2.name IS NOT "A"
   temp2 = temp2.next
END WHILE

// temp2 now references Node A.

temp.next = temp2.next
temp2.next = temp
```

La siguiente imagen muestra la inserción del nodo `D` entre los nodos `A` y `C`.



El siguiente listado presenta el equivalente Java de los ejemplos de pseudocódigo de inserción anteriores:

```
// SLLInsDemo.java

class SLLInsDemo {
   static class Node  {
      String name;
      Node next;
   }

   public static void main (String [] args) {
      Node top = null;

      // 1. The singly linked list does not exist
```

- Java en castellano (Tutorial Estructuras de Datos y Algoritmos en Java)
  http://www.programacion.com/java/tutorial/jap_data_alg/
- Tutorial original en Inglés
  http://www.javaworld.com/

```
      top = new Node ();
      top.name = "A";
      top.next = null;

      dump ("Case 1", top);

      // 2. The singly linked list exists, and the node must be inserted
      //    before the first node

      Node temp;

      temp = new Node ();
      temp.name = "B";
      temp.next = top;
      top = temp;

      dump ("Case 2", top);

      // 3. The singly linked list exists, and the node must be inserted
      //    after the last node

      temp = new Node ();
      temp.name = "C";
      temp.next = null;

      Node temp2;

      temp2 = top;

      while (temp2.next != null)
         temp2 = temp2.next;

      temp2.next = temp;

      dump ("Case 3", top);

      // 4. The singly linked list exists, and the node must be inserted
      //    between two nodes

      temp = new Node ();
      temp.name = "D";

      temp2 = top;

      while (temp2.name.equals ("A") == false)
         temp2 = temp2.next;

      temp.next = temp2.next;
      temp2.next = temp;

      dump ("Case 4", top);
   }

   static void dump (String msg, Node topNode) {
      System.out.print (msg + " ");

      while (topNode != null) {
         System.out.print (topNode.name + " ");
         topNode = topNode.next;
      }
      System.out.println ();
   }
}
```

El método `static void dump(String msg, Node topNode)` itera sobre la lista e imprime su contenido. Cuando se ejecuta `SLLInsDemo`, las repetidas llamadas a este método dan como resultado la siguiente salida, lo que coincide con las imágenes anteriores:

```
Case 1 A
```

```
Case 2 B A
Case 3 B A C
Case 4 B A D C
```

Nota:

`SLLInsDemo` y los ejemplos de pseudocódigo anteriores empleaban un algoritmo de búsqueda lineal orientado a listas enlazadas para encontrar un `Node` específico. Indudablemente usted utilizará este otro algoritmo en sus propios programas:

- **Búsqueda del último `Node`:**

```
// Assume top references a singly linked list of at least one Node.

Node temp = top // We use temp and not top. If top were used, we
                // couldn't access the singly linked list after
                // the search finished because top would refer
                // to the final Node.
WHILE temp.next IS NOT NULL
   temp = temp.next
END WHILE

// temp now references the last Node.
```

- **Búsqueda de un `Node` específico:**

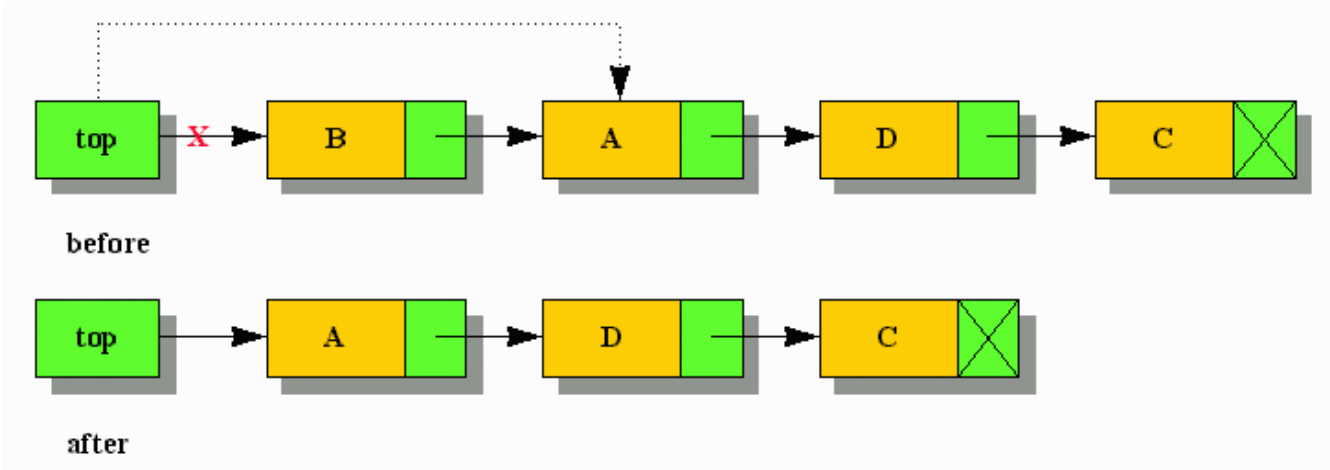
```
// Assume top references a singly linked list of at least one Node.

Node temp = top

WHILE temp IS NOT NULL AND temp.name IS NOT "A" // Search for "A".
   temp = temp.next
END WHILE

// temp either references Node A or contains NULL if Node A not found.
```

Otro algoritmo común de las listas de enlace simples es el borrado de nodos. Al contrario que la inserción de nodos, sólo hay dos casos a considerar:

- **Borrar el Primer nodo:**
  Asigna el enlace del campo `next` del nodo referenciado por `top` a `top`:

```
top = top.next; // Reference the second Node (or NULL if there is only one
Node)
```

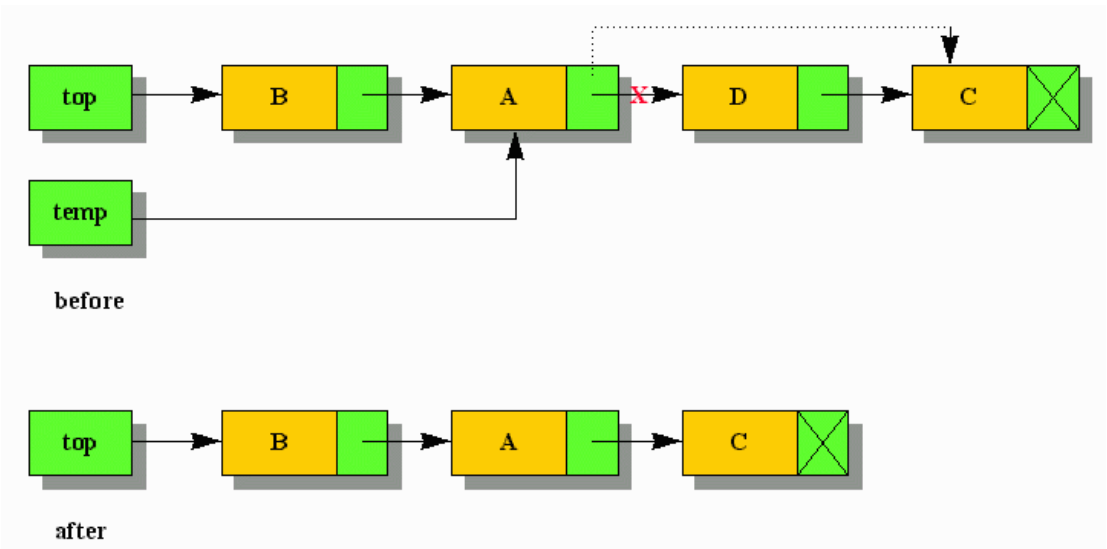
La siguiente imagen presenta las vistas anterior y posterior de una lista donde se ha borrado el primer nodo. en esta figura, el nodo `B` desaparece y el nodo `A` se convierte en el primer nodo.



- **Borrar cualquier nodo que no sea el primero:**
  Localiza el nodo que precede al nodo a borrar y le asigna el enlace que hay en el campo `next` del nodo a borrar al campo `next` del nodo que le precede. El siguiente pseudocódigo borra el nodo `D`:

```
temp = top
WHILE temp.name IS NOT "A"
   temp = temp.next
END WHILE

// We assume that temp references Node A

temp.next = temp.next.next

// Node D no longer exists
```

La siguiente figura presenta las vistas anterior y posterior de una lista donde se ha borrado un nodo intermedio. En esa figura el nodo `D` desaparece.



El siguiente listado representa el equivalente Java a los pseudocódigos de borrado anteriores:

```
// SLLDelDemo.java

class SLLDelDemo {
```

- Java en castellano (Tutorial Estructuras de Datos y Algoritmos en Java)
  http://www.programacion.com/java/tutorial/jap_data_alg/
- Tutorial original en Inglés
  http://www.javaworld.com/

```
   static class Node {
      String name;
      Node next;
   }

   public static void main (String [] args) {
      // Build Figure 6's singly linked list (i.e., B A D C)

      Node top = new Node ();
      top.name = "C";
      top.next = null;

      Node temp = new Node ();
      temp.name = "D";
      temp.next = top;
      top = temp;

      temp = new Node ();
      temp.name = "A";
      temp.next = top;
      top = temp;

      temp = new Node ();
      temp.name = "B";
      temp.next = top;
      top = temp;

      dump ("Initial singly-linked list", top);

      // 1. Delete the first node

      top = top.next;

      dump ("After first node deletion", top);

      // Put back B

      temp = new Node ();
      temp.name = "B";
      temp.next = top;
      top = temp;

      // 2. Delete any node but the first node

      temp = top;

      while (temp.name.equals ("A") == false)
         temp = temp.next;

      temp.next = temp.next.next;

      dump ("After D node deletion", top);
   }

   static void dump (String msg, Node topNode) {
      System.out.print (msg + " ");

      while (topNode != null) {
         System.out.print (topNode.name + " ");
         topNode = topNode.next;
      }
      System.out.println ();
   }
}
```

Cuando ejecute `SLLDelDemo`, observará la siguiente salida:

```
Initial singly linked list B A D C
After first node deletion A D C
After D node deletion B A C
```

Cuidado:

Como java inicializa los campos de referencias de un objeto a `null` durante la construcción del objeto, no es necesario asignar explícitamente `null` a un campo de enlace. No olvide estas asignaciones de `null` en su código fuente; su ausencia reduce la claridad del código.

Después de estudiar `SLLDelDemo`, podría preguntarse qué sucede si asigna `null` al nodo referenciado por `top`: ¿el recolector de basura recogerá toda la lista? Para responder a esta cuestión, compile y ejecute el código del siguiente listado:

```
// GCDemo.java

class GCDemo {
   static class Node {
      String name;
      Node next;

      protected void finalize () throws Throwable {
         System.out.println ("Finalizing " + name);
         super.finalize ();
      }
   }

   public static void main (String [] args) {
      // Build Figure 6's singly linked list (i.e., B A D C)

      Node top = new Node ();
      top.name = "C";
      top.next = null;

      Node temp = new Node ();
      temp.name = "D";
      temp.next = top;
      top = temp;

      temp = new Node ();
      temp.name = "A";
      temp.next = top;
      top = temp;

      temp = new Node ();
      temp.name = "B";
      temp.next = top;
      top = temp;

      dump ("Initial singly-linked list", top);

      top = null;
      temp = null;

      for (int i = 0; i < 100; i++)
           System.gc ();
   }

   static void dump (String msg, Node topNode)  {
      System.out.print (msg + " ");

      while (topNode != null){
         System.out.print (topNode.name + " ");
         topNode = topNode.next;
      }
      System.out.println ();
   }
}
```

`GCDemo` crea la misma lista de cuatro nodos que `SLLDelDemo`. Después de volcar los nodos a la salida estándar, `GCDemo` asigna `null` a `top` y a `temp`. Luego, `GCDemo` ejecuta `System.gc ();` hasta 100 veces. ¿Qué sucede después? Mire la salida (que he observado en mi plataforma Windows):

```
Initial singly-linked list B A D C
Finalizing C
Finalizing D
Finalizing A
Finalizing B
```

La salida revela que todos los nodos de la lista de enlace simple han sido finalizados (y recolectados). Como resultado, no tiene que preocuparse de poner a `null` todos los enlaces de una lista de enlace simple cuando se quiera deshacer de ella. (Podría necesitar tener que incrementar el número de ejecuciones de `System.gc ();` si su salida no incluye los mensajes de finalización.)

Los Algoritmos de Concatenación e Inversión

Existen muchos algoritmos útiles para listas de enlace simple. Uno de ellos es la concatenación, que implica que puede añadir una lista de enlace simple al final de otra lista.

Otro algoritmo útil es la inversión. Este algoritmo invierte los enlaces de una lista de enlace simple permitiendo atravesar los nodos en direccion opuesta. El siguiente código extiende la clase anterior para invertir los enlaces de la lista referenciada por `top1`:

```
>
// CIDemojava

class CIDemo {

   static class DictEntry {
      String word;
      String meaning;
      DictEntry next;
   }

   // ListInfo is necessary because buildList() must return two pieces
   // of information

   static class ListInfo {
      DictEntry top;
      DictEntry last;
   }

   public static void main (String [] args) {
      String [] wordsMaster = { "aardvark", "anxious", "asterism" };

      ListInfo liMaster = new ListInfo ();
      buildList (liMaster, wordsMaster);

      dump ("Master list =", liMaster.top);

      String [] wordsWorking = { "carbuncle", "catfish", "color" };
      ListInfo liWorking = new ListInfo ();
      buildList (liWorking, wordsWorking);

      dump ("Working list =", liWorking.top);

      // Perform the concatenation

      liMaster.last.next = liWorking.top;
      dump ("New master list =", liMaster.top);

      invert (liMaster);
      dump ("Inverted new master list =", liMaster.top);
   }

   static void buildList (ListInfo li, String [] words) {
      if (words.length == 0)
          return;

      // Create a node for first word/meaning
```

```
        li.top = new DictEntry ();
        li.top.word = words [0];
        li.top.meaning = null;

        // Initialize last reference variable to
        // simplify append and make concatenation possible.

        li.last = li.top;

        for (int i = 1; i < words.length; i++) {

             // Create (and append) a new node for next word/meaning

             li.last.next = new DictEntry ();
             li.last.next.word = words [i];
             li.last.next.meaning = null;

             // Advance last reference variable to simplify
             // append and make concatenation possible

             li.last = li.last.next;
        }

        li.last.next = null;
    }

    static void dump (String msg, DictEntry topEntry) {
        System.out.print (msg + " ");

        while (topEntry != null) {
            System.out.print (topEntry.word + " ");
            topEntry = topEntry.next;
        }
        System.out.println ();
    }

    static void invert (ListInfo li) {
        DictEntry p = li.top, q = null, r;

        while (p != null)  {
            r = q;
            q = p;
            p = p.next;
            q.next = r;
        }
        li.top = q;
    }
}
```

`CIDemo` declara un `DictEntry` anidado en la clase de más alto nivel cuyos objetos contienen palabras y significados. (Para mantener el programa lo más sencillo posible, he evitado los significados. Usted puede añadirlos si lo desea). `CIDemo` también declara `ListInfo` para seguir las referencias el primero y último `DictEntry` de una lista de enlace simple.

El thread principal ejecuta el método `public static void main(String [] args)` de `CIDemo`. Este thread llama dos veces al método `static void buildList (ListInfo li, String [] words)` para crear dos listas de enlace simple: una lista maestra (cuyos nodos se rellenan con palabras del array `wordsMaster`), y una lista de trabajo (cuyos nodos se rellenan con palabras del array `wordsWorking`). Antes de cada llamada al método `buildList (ListInfo li, String [] words)`, el thread principal crea y pasa un objeto `ListInfo`. este objeto devuelve las referencias al primero y último nodo. (Una llamada a método devuelve directamente un sólo dato). Después de construir una lista de enlace simple, el thread principal llama a `static void dump (String msg, DictEntry topEntry)` para volcar un mensaje y las palabras de los nodos de una lista en el dispositivo de salida estándar.

Se podría estar preguntando sobre la necesidad del campo `last` de `ListInfo`. Este campo sirve a un doble propósito: primero, simplifica la creación de cada lista, donde se añaden los nodos. Segundo, este campo simplifica la concatenación, que se queda sólo en la ejecución de la siguiente línea de código: `liMaster.last.next = liWorking.top;`. Una vez que se completa la concatenación, y el thread

principal vuelva los resultados de la lista maestra en la salida estándar, el thread llama al método `static void invert (ListInfo li)` para invertir la lista maestra y luego muestra la lista maestra invertida por la salida estándar.

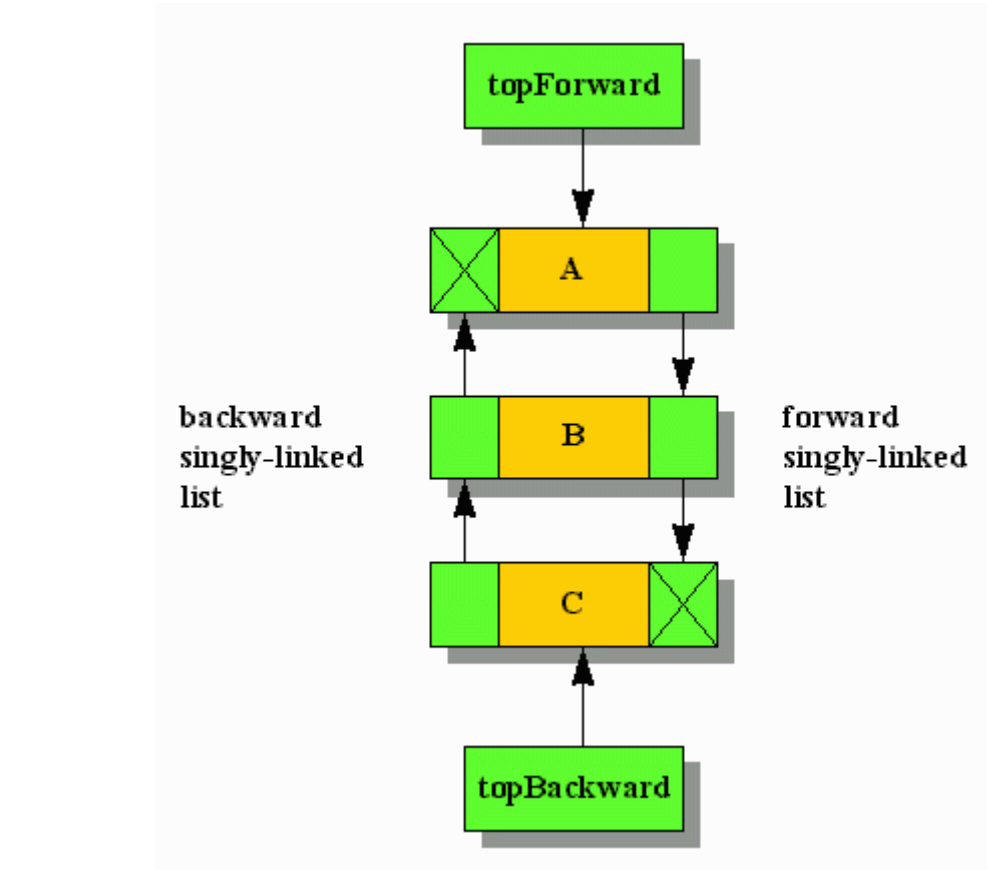
Cuando ejecute `CIDemo` verá la siguiente salida:

```
Master list = aardvark anxious asterism
Working list = carbuncle catfish color
New master list = aardvark anxious asterism carbuncle catfish color
Inverted new master list = color catfish carbuncle asterism anxious aardvark
```

Lista Doblemente Enlazada

Las listas de enlace simple restringen el movimiento por lo nodos a una sola dirección: no puede atravesar una lista de enlace simple en dirección opuesta a menos que primero utilice el algoritmo de inversión para invertir los enlaces de los nodos, lo que lleva tiempo. Después de atravesarlos en dirección opuesta, probablemente necesitará repetir la inversión para restaurar el orden original, lo que lleva aún más tiempo. Un segundo problema implica el borrado de nodos: no puede borrar un nodo arbitrario sin acceder al predecesor del nodo. Estos problemas desaparecen cuando se utiliza una lista doblemente enlazada.

Una *lista doblemente enlazada* es una lista enlazada de nodos, donde cada nodo tiene un par de campos de enlace. Un campo de enlace permite atravesar la lista hacia adelante, mientras que el otro permite atravesar la lista hacia atrás. Para la dirección hacia adelante, una variable de referencia contiene una referencia al primer nodo. Cada nodo se enlaza con el siguiente mediante el campo de enlace `next`, excepto el último nodo, cuyo campo de enlace `next` contiene `null` para indicar el final de la lista (en dirección hacia adelante). De forma similar, para la dirección contraria, una variable de referencia contiene una referencia al último nodo de la dirección normal (hacia adelante), lo que se interpreta como el primer nodo. Cada nodo se enlaza con el anterior mediante el campo de enlace `previous`, y el primer nodo de la dirección hacia adelante, contiene `null` en su campo `previous` para indicar el fin de la lista. La siguiente figura representa una lista doblemente enlazada de tres nodos, donde `topForward` referencia el primer nodo en la dirección hacia adelante, y `topBackward` referencia el primero nodo la dirección inversa.



- Java en castellano (Tutorial Estructuras de Datos y Algoritmos en Java)
  http://www.programacion.com/java/tutorial/jap_data_alg/
- Tutorial original en Inglés
  http://www.javaworld.com/

Truco:

Piense en una lista doblemente enlazada como una pareja de listas de enlace simple que interconectan los mismos nodos.

La inserción y borrado de nodos en una lista doblemente enlazada son operaciones comunes. Estas operaciones se realizan mediante algoritmos que se basan en los algoritmos de inserción y borrado de las listas de enlace simple (porque las listas doblemente enlazadas sólo son una pareja de listas de enlace simple que interconectan los mismos nodos).

El siguiente listado muestra la inserción de nodos para crear la lista de la figura anterior, el borrado de nodos ya que elimina el nodo `B` de la lista, y el movimiento por la lista en ambas direcciones:

```
// DLLDemo.java

class DLLDemo {
   static class Node {
      String name;
      Node next;
      Node prev;
   }

   public static void main (String [] args) {
      // Build a doubly linked list

      Node topForward = new Node ();
      topForward.name = "A";

      Node temp = new Node ();
      temp.name = "B";

      Node topBackward = new Node ();
      topBackward.name = "C";

      topForward.next = temp;
      temp.next = topBackward;
      topBackward.next = null;

      topBackward.prev = temp;
      temp.prev = topForward;
      topForward.prev = null;

      // Dump forward singly linked list

      System.out.print ("Forward singly-linked list: ");

      temp = topForward;
      while (temp != null){
         System.out.print (temp.name);
         temp = temp.next;
      }

      System.out.println ();

      // Dump backward singly linked list

      System.out.print ("Backward singly-linked list: ");

      temp = topBackward;
      while (temp != null){
         System.out.print (temp.name);
         temp = temp.prev;
      }

      System.out.println ();

      // Reference node B

      temp = topForward.next;
```

```
      // Delete node B

      temp.prev.next = temp.next;
      temp.next.prev = temp.prev;

      // Dump forward singly linked list

      System.out.print ("Forward singly-linked list (after deletion): ");

      temp = topForward;
      while (temp != null){
         System.out.print (temp.name);
         temp = temp.next;
      }

      System.out.println ();

      // Dump backward singly linked list

      System.out.print ("Backward singly-linked list (after deletion): ");

      temp = topBackward;
      while (temp != null){
         System.out.print (temp.name);
         temp = temp.prev;
      }
      System.out.println ();
   }
}
```

Cuando se ejecuta, `DLLDemo` produce la siguiente salida:

```
Forward singly-linked list: ABC
Backward singly-linked list: CBA
Forward singly-linked list (after deletion): AC

Backward singly-linked list (after deletion): CA
```

Algoritmo de Inserción-Ordenada

Algunas veces querrá crear una lista doblemente enlazada que organice el orden de sus nodos basándose en un campo no de enlace. Atravesar la lista doblemente enlazada en una dirección presenta esos nodos en orden ascendente, y atravesarla en dirección contraria los presenta ordenados descendentemente. El algoritmo de ordenación de burbuja es inapropiado en este caso porque requiere índices de array. Por el contrario, *inserción-ordenada* construye una lista de enlace simple o una lista doblemente enlazada ordenadas por un campo no de enlace para identificar el punto de inserción de cada nuevo nodo. El siguiente litado demuestra el algoritmo de inserción-ordenada:

```
// InsSortDemo.java

class InsSortDemo {
   // Note: To keep Employee simple, I've omitted various constructor and
   // nonconstructor methods. In practice, such methods would be present.

   static class Employee {
      int empno;
      String name;

      Employee next;
      Employee prev;
   }

   public static void main (String [] args) {
      // Data for a doubly linked list of Employee objects. The lengths of
      // the empnos and names arrays must agree.

      int [] empnos = { 687, 325, 567, 100, 987, 654, 234 };
```

```
      String [] names = { "April", "Joan", "Jack", "George", "Brian", "Sam",
"Alice" };

      Employee topForward = null;
      Employee topBackward = null;

      // Prime the doubly linked list by creating the first node.

      topForward = new Employee ();
      topForward.empno = empnos [0];
      topForward.name = names [0];
      topForward.next = null;
      topForward.prev = null;
      topBackward = topForward;

      // Insert remaining Employee nodes (in ascending order -- via empno)
      // into the doubly linked list.

      for (int i = 1; i < empnos.length; i++) {
           // Create and initialize a new Employee node.

           Employee e = new Employee ();
           e.empno = empnos [i];
           e.name = names [i];
           e.next = null;
           e.prev = null;

           // Locate the first Employee node whose empno is greater than
           // the empno of the Employee node to be inserted.

           Employee temp = topForward;
           while (temp != null && temp.empno <= e.empno)
              temp = temp.next;

           // temp is either null (meaning that the Employee node must be
           // appended) or not null (meaning that the Employee node must
           // be inserted prior to the temp-referenced Employee node).

           if (temp == null) {

              topBackward.next = e; // Append new Employee node to
              // forward singly linked list.

              e.prev = topBackward; // Update backward singly linked
              topBackward = e;      // list as well.

           }
           else{
               if (temp.prev == null) {
                  e.next = topForward; // Insert new Employee node at
                  topForward = e;      // head of forward singly linked
                                       // list.
                  temp.prev = e;       // Update backward singly linked
                                       // list as well.
               }
               else {
                  e.next = temp.prev.next; // Insert new Employee node
                  temp.prev.next = e;      // after last Employee node
                                           // whose empno is smaller in
                                           // forward singly linked list.
                  e.prev = temp.prev;      // Update backward
                  temp.prev = e;           //singly linked list as well.
               }
           }
      }

      // Dump forward singly linked list (ascending order).

      System.out.println ("Ascending order:\n");

      Employee temp = topForward;
      while (temp != null) {
```

```
            System.out.println ("[" + temp.empno + ", " + temp.name + "] ");
            temp = temp.next;
         }

         System.out.println ();

         // Dump backward singly linked list (descending order).

         System.out.println ("Descending order:\n");

         temp = topBackward;
         while (temp != null) {
            System.out.println ("[" + temp.empno + ", " + temp.name + "] ");
            temp = temp.prev;
         }

         System.out.println ();
      }
   }
```

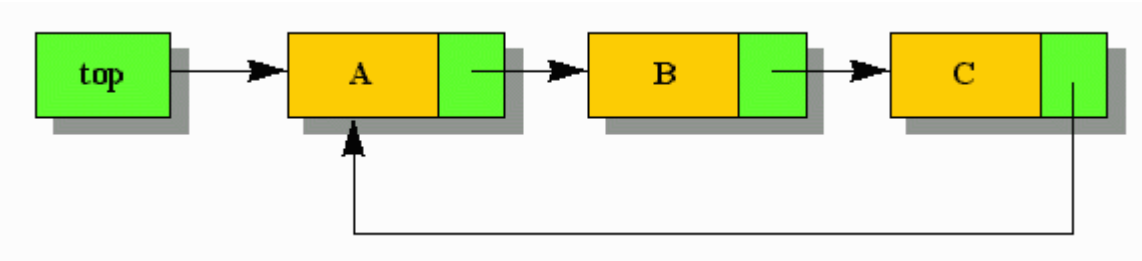
`InsSortDemo` simplifica su operación creando primero un nodo `Employee` primario. Para el resto de nodos `Employee`, `InsSortDemo` localiza la posición de inserción apropiada basándose en el campo no de enlace `empno`, y luego inserta el `Employee` en esa posición. Cuando ejecute `InsSortDemo`, podrá observar la siguiente salida:

```
Ascending order:

[100, George]
[234, Alice]
[325, Joan]
[567, Jack]
[654, Sam]
[687, April]
[987, Brian]

Descending order:

[987, Brian]
[687, April]
[654, Sam]
[567, Jack]
[325, Joan]
[234, Alice]
[100, George]
```

Tanto la inserción-ordenada como la ordenación de burbuja exhiben prácticamente el mismo rendimiento.

Lista de Enlace Circular

El campo de enlace del último nodo de una lista de enlace simple contiene un enlace nulo, ocurre lo mismo en los campos de enlace del primer y último elemento en ambas direcciones en las listas doblemente enlazadas. Supongamos que en vez de esto los últimos nodos contiene un enlace a los primeros nodos. En esta situación, usted terminará con una *lista de enlace circular*, como se ve en la siguiente figura:



- Java en castellano (Tutorial Estructuras de Datos y Algoritmos en Java)
  http://www.programacion.com/java/tutorial/jap_data_alg/
- Tutorial original en Inglés
  http://www.javaworld.com/

Las listas de enlace circular se utilizan con frecuencia en procesamiento repetitivo de nodos en un orden específico. Dichos nodos podrían representar conexiones de servidor, procesadores esperando una sección crítica, etc. Esta estructura de datos también sirve como base para una variante de una estructura de datos más compleja: la *cola* (que veremos más adelante).

Listas Enlazadas frente a Arrays

Las listas enlazadas tienen las siguientes ventajas sobre los arrays:

- No requieren memoria extra para soportar la expansión. Por el contrario, los arrays requieren memoria extra si se necesita expandirlo (una vez que todos los elementos tienen datos no se pueden añadir datos nuevos a un array).
- Ofrecen una inserción/borrado de elementos más rápida que sus operaciones equivalentes en los arrays. Sólo se tienen que actualizar los enlaces después de identificar la posición de inserción/borrado. Desde la perspectiva de los arrays, la inserción de datos requiere el movimiento de todos los otros datos del array para crear un elemento vacío. De forma similar, el borrado de un dato existente requiere el movimiento de todos los otros datos para eliminar el elemento vacío.

En contraste, los arrays ofrecen las siguientes ventajas sobre las listas enlazadas:

- Los elementos de los arrays ocupan menos memoria que los nodos porque no requieren campos de enlace.
- Los arrays ofrecen un acceso más rápido a los datos, mediante índices basados en enteros.

Las listas enlazadas son más apropiadas cuando se trabaja con datos dinámicos. En otras palabras, inserciones y borrados con frecuencia. Por el contrario, los arrays son más apropiados cuando los datos son estáticos (las inserciones y borrados son raras). De todas formas, no olvide que si se queda sin espacio cuando añade ítems a un array, debe crear un array más grande, copiar los datos del array original el nuevo array mayor y eliminar el original. Esto cuesta tiempo, lo que afecta especialmente al rendimiento si se hace repetidamente.

Mezclando una lista de enlace simple con un array uni-dimensional para acceder a los nodos mediante los índices del array no se consigue nada. Gastará más memoria, porque necesitará los elementos del array más los nodos, y tiempo, porque necesitará mover los ítems del array siempre que inserte o borre un nodo. Sin embargo, si es posible integrar el array con una lista enlazada para crear una estructura de datos útil (por ejemplo, las *tablas hash*).