Ciencias computacionales

Propedeutico: Programación

INAOE

Contents

1 Listas **2**
1.1 Definición . . . 2
1.2 Operaciones . . . 2
1.3 Listas basadas en arreglos . . . 3
1.4 Listas ligadas . . . 4
1.5 Listas doblemente ligadas . . . 8
1.6 Listas circulares . . . 10

1 Listas

Material multimedia recomendado:

- Listas ligadas: `https://www.youtube.com/watch?v=2YPMA1p5KoM`

1.1 Definición

En esta sección se presentará la representación se secuencias por medio de listas ligadas. Podemos pensar en una lista ligada como si fuera una cadena, donde cada eslabón almacena un elemento, y también tiene una referencia al siguiente eslabón. Una lista ligadas esta compuesta de nodos los cuales almacenan algún elemento de interés, que podían ser números enteros, caracteres, estructuras, u objetos. Para efectos prácticos en este capítulo utilizaremos listas ligadas compuestas de nodos que almacenan números enteros. Las listas ligadas pueden clasificarse en listas ligadas simples y listas doblemente ligadas, las cuales estudiaremos más adelante. Otra variante de las listas ligadas son las listas circulares.

Una posible definición de lista ligadas es la siguiente:

Lista ligada: Secuencia finita y ordenada de datos conocidos como elementos o nodos. Ordenada en el sentido de que cada elemento tiene una posición (índice) en la lista.

Notación: $< a_0, a_1, \cdots, a_{n-1} >$ para los n elementos de la lista A.

Las siguientes definiciones nos ayudarán a tener una mejor representación clara de listas ligadas:

Lista vacia: Cuando no tiene elementos.

Longitud ó *lenght*: Se refiere al número de elementos en la lista.

Head: Se refiere al primer elemento de la lista.

Tail: Se refiere al último elemento de la lista.

Puede o no haber relación entre el valor de un elemento y su posición en la lista (listas ordenadas o no ordenadas).

Para indicar la posición actual en la que nos encontramos dentro de la lista podemos utilizar el operador | (indicador de posición). Por ejemplo:

Sea la lista: $< 20, 23|12, 15 >$

La posición en la cual estamos trabajando es entre el par de nodos 2 y 3, los cuales almacenan los valores 23 y 12 respectivamente. No hay que confundir los nodos con los elementos (valores) que almacenan. Los accesos a los datos de la lista se realizan con base en la posición de —. El símbolo | divide a la lista en una partición derecha y una izquierda. La longitud de la partición izquierda crece conforme el indicador se mueve hacia la derecha. A continuación se presentan las operaciones que se pueden realizar sobre una lista.

1.2 Operaciones

Una lista nos permite tener una estructura para manipular nuestros datos de una forma cómoda y eficiente para algunas aplicaciones. La declaración en C de las operaciones que podemos realizar en una lista son las siguientes.

```
1  void clear(List L);                  //Reinicializa la lista
2  int insert(List L, Element Item);//Inserción al frente de la partición derecha
3  int append(List L, Element Item);//Inserción al final de la partición derecha
4  int remove(List L);                  //Eliminación el 1er elemento de la partición
5                                       //derecha
6                                       //
7  void setStart(List L);               //Mueve el indicador al inicio,
8  void setEnd(List L);                 //al final,
```

```
int setPos(List L, int pos);     //a  una posición específica  de la lista
                                  //
void prev(List L);                //Mueve el indicador a un elemento previo
void next(List L);                //o al elemento siguiente de la lista
                                  //
int leftLength(List L);           //Devuelve la longitud de la fila izquierda
int rightLength(List L);          // o derecha
                                  //
int getValue(List L, Element Item)//Devuelve el valor del primer elemento de
                                  //la partición derecha.
void print(Element Item);         //Imprime un elemento dado
```

Ejemplo:
Supongamos que se tiene la lista:

List : $< 12|32, 15 >$

- Al realizar la operación:

```
insert(List, 99);
```

Obtendremos como resultado la lista:

List : $< 12|99, 32, 15 >$

- Si decaemos recorrer toda la lista he imprimir el valor de cada uno de los nodos:

```
setStart(List)
While(next(List)){
        getValue(List, it);
        print(it)
```

1.3 Listas basadas en arreglos

Hasta ahora hemos trabajado el tipo de dato abstracto lista y sus operaciones, pero no hemos mencionado como representar una lista. Las dos formas más comunes para representar una lista son por medio de arreglos y por punteros (listas ligadas). Tanto las lista ligadas como las representadas por arreglos tienen algunas ventajas y desventajas. Por lo cual es necesario conocer la aplicación en la que se utilizará la lista, para poder seleccionar la representación más adecuada, y que nos permita aprovechar al máximo los recursos con los que contamos.

Cuando los valores son almacenados en un arreglo unidimensional ($x[0]$ a $x[n]$) estos están organizados como una lista lineal, i.e., los **nodos** están en un orden donde se cumple que:

- $x[0]$ es el primer nodo.
- $x[n]$ es el último nodo.
- Si $0 < k \leq n$, entonces $x[k]$ es precedido por $x[k-1]$.
- Si $0 \leq k < n$, entonces $x[k]$ es seguido por $x[k+1]$.

La principal característica de las listas basadas en arreglos son:

- Tienen una longitud fija, una vez que la lista ha sido creada no puede cambiar su tamaño, y si se requiere insertar más elementos en una lista que ya esta llena es necesario tomar algunas medidas. Por ejemplo, crear una lista nueva con mayor espacios, vaciar todos los elementos de la lista a la lista nueva, insertar el nuevo elemento, eliminar la lista anterior, y hacer referencia a esta nueva lista. Todas estas operaciones tienen una penalidad en cuanto al tiempo de procesamiento, y este tiempo puede no ser tolerado cuando se trabaja gran cantidad de elementos.

- Su longitud debe conocerse cuando la lista es creada. Es necesario reserva una longitud de memoria fija para garantizar que la memoria asignada a la lista es contigua, un solo bloque.
- La lista puede almacenar un número de elementos mucho menor que su máxima compacidad o bien requerir almacenar más elementos del que tiene capacidad. En el primer caso, cuando declaramos una Lista con mucha capacidad, y solo almacenamos pocos valores estaremos desperdiciando memoria que podría ser ocupada para otras tareas. En el segundo caso, es necesario tomar alguna medida para tratar elementos cuando la lista ya esta llena.
- Los elementos deben ser almacenados en posiciones contiguas, y las operaciones para agregar/eliminar elementos deben mantener esta propiedad. En el primer caso, agregar elementos, es necesario recorrer todos los elementos necesarios para dar espacio al elemento a insertar. Y en el segundo caso, eliminar elementos, es necesario recorrer todos los elementos necesarios para no dejar huecos en la lista.

  Ejemplo:

  Supongamos que se tiene la lista:

  List : $< 12|32, 15 >$

  implementada por el arreglo de la figura 1.

Figure 1: Lista representada por arreglo

Al realizar la operación **insert(List, 99)** todos los elementos a la derecha del indicador de posición tendrán que recorrerse para crear un espacio para el nuevo elemento, figura 2. Esto presenta una forma lenta de insertar/eliminar elementos al inicio de la lista, ya que se requiere recorrer varios elementos. Por el contrario, el acceso a cualquier nodo es fácil y directo, solo se debe proveer el índice apropiado.

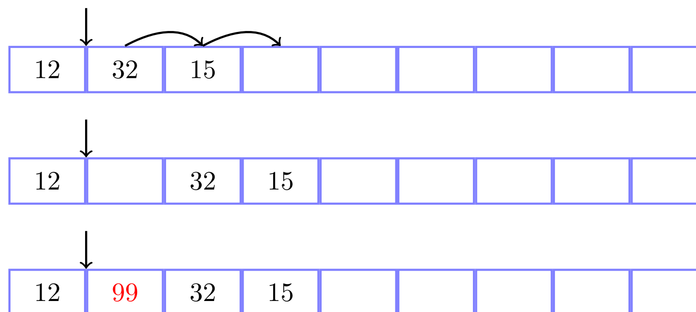

Figure 2: Inserción en lista basada en arreglos

1.4 Listas ligadas

Ahora presentaremos otro tipo de lista, la **lista ligada**. Este tipo de listas no necesitan tener un tamaño predefinido, ya que se va generando de forma dinamica. La listas ligada, en su forma más simple, es una colección de **nodos** que en conjunto forman un ordenamiento lineal. Cada nodo en la lista apunto explícitamente al siguiente nodo. En general, un nodo de la lista almacena un puntero hacia otro nodo y cierta informacion, la cual puede ser un tipo primitivo: entero, char, float, o bien, un tipo definido por el usuario: estructura, union, objeto (en este documento se mostrarán ejemplos con números enteros). La estructura general de un nodo para una lista ligada simplme se muestra en la figura 3.

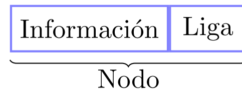

Figure 3: Nodo para lista ligada simple

En una lista ligada no es necesario que los espacios de memoria de los elementos esten contiguos. De esta forma se puede ir pidiendo más memoria al sistema operativo solo cuando se requiera, y hacer que los enlaces de los nodos mantengan las direcciones adecuadas para acceder a los elementos. En la lista ligada simple se tiene un puntero comunmente llamado **head** ó **top** (en el texto se usarán indistintamente estos nombres) que a[unto al primer elemento de la lista. El último nodo de la lista debe apuntar a **NULL**, esto nos indica que no hay más elementos en la lista ligada. La estructura genral de una lista ligada simple se muestra en la figura 4.

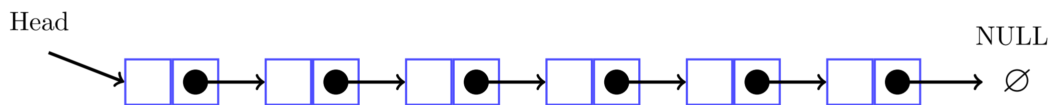

Figure 4: Lista ligada simple

Como ejemplo de lista ligada usaremos la lista:

Lista: $< 20, 23|12, 15 >$

y realizaremos la operación: **insert(List, 10)**. La lista ligada que representa la lista List y la siguiente inserción del elemento 10 se muestra en la figura 5.

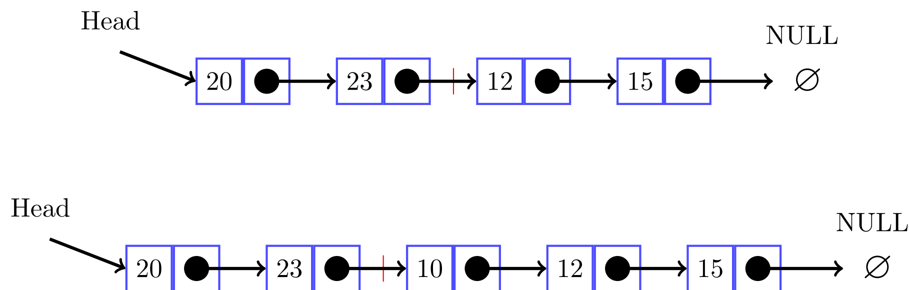

Figure 5: Representación de lista ligada, y operación **insert(Lista, 10)**.

En una lista ligada se puden ir insertando nodos al inicio al final de forma sencilla, e incluso en cualquier otra posición con un poco más de esfuerso. Las estracciones se pueden hacerse simplemente 'rompiendo' los enlaces del nodo que se desesa eliminar y restaurando los enlaces de los nodos vecinos. A diferencia de la implementación basada en arreglos, insertar y eliminar un elemento no requiere recorrer todos los demás elementos. La principal ventaja de una lista ligada es que solo se utiliza el espazio necesario para los elementos que se tienen en un momento dado. Por lo cual, no es necesario conocer de antemano el tamaño de la lista ni reservar un bloque fijo de memoria para almacenar sus elementos, ya que estos no estan en bloque de memoria contiguos.

La figura 6 muestra un ejemplo de las posibles direcciones de memoria que podría tener una lista ligada simple. A pesar de que la representamos en forma lineal, es preciso asumir que las direcciones de los datos no son contiguas. En el ejemplo de la figura 6 el puntero **head** apunta a la dirrección de memoria 2003. La direeción de memoria 2003 almacena un nodo el cual consta de un número entero y un apuntador a otra dirección de memeoria. La dirección del entero es la 2003 y la direeción del apuntador es la dirección 2004, la cual a su vez apunto a la dirección del siguiente nodo, la cual es 2183. La nueva dirección 2183 almacena un nuevo nodo de la lista ligada, y podemos continuar con este proedimiento

siguiendo todos los enlaces a los nodos de la lista hasta llegar al último nodo el cual almacenara un elemento, pero no tendra una dirección a un nodo valido, puntará a la dirección de memoria NULL.

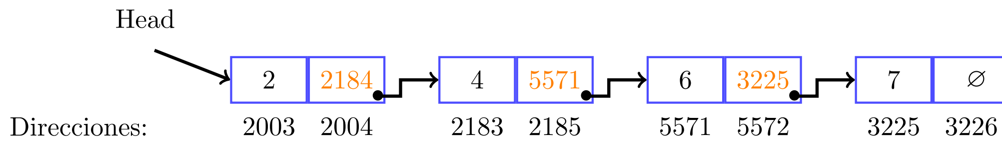

Figure 6: Direcciones de los nodos en una lista ligada.

Ahora pasaremos a representar una nodo en el lenguaje de programación C. La forma más comun de implenetar un nodo en el lenguage de programación C es por medio de una estructura, la cual almacenara el elemento y un puntero al siguiente nodo. Si el elemento a guardar es un numero entero, la siguiente declaración nos permitirá crear una lista ligada:

```
1 typedef struc node{
2         int num;
3         struc node *next;
4 }Node;
5
6 Node *head;
```

Al iniciar la lista ligada se crea un nodo inicial y este nodo es al que hacen referencia los apuntadores: head, curr y tail. Cabe mencionar que el apuntador tail siempre apunta al último nodo de la lista ligada. Este apuntador es opcional, y es común utilizarlo para acelerar la operación de insertar al final de la lista ligada. La representación inicial de la lista ligada se muestra en la figura 7.

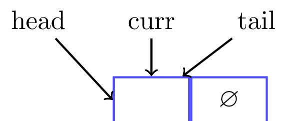

Figure 7: Estado inicial de lista ligada con apuntadores: head, curr y tail.

Acontinuación mostraremos la representación de las operaciones de insertar y eliminar sobre una lista ligada que ya ha sido inicializada. Como se menciono anteriormente, si la lista esta vacia solo hay que crear el nodo con el nuevo valor a insertar, hacer que apunte a NULL, y hacer que los apuntadores head, curr y tail hagan referencia a este nuevo nodo. Pero cuando la lista ligada ya tiene algunos elementos, es necesario actualizar los apuntadores de los nodos vecinos. En la figura 8 se muestra la lista ligada $< \cdots, 23|12, \cdots >$. En la cual se quiere realizar la inserción del elemento 10. Lo primero que se tiene que realizar es crear el nuevo nodo con el elemento 10.

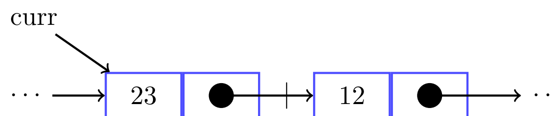

Figure 8: Inserción en lista ligada

Una vez creado el nuevo nodo, lo que tenemos que realizar es la actualización de los apuntadores. La figura 9 muestra la actualización de los apuntadores al insertar el elemento 10, y la lista ligada final despues de actualizar los apuntadores.

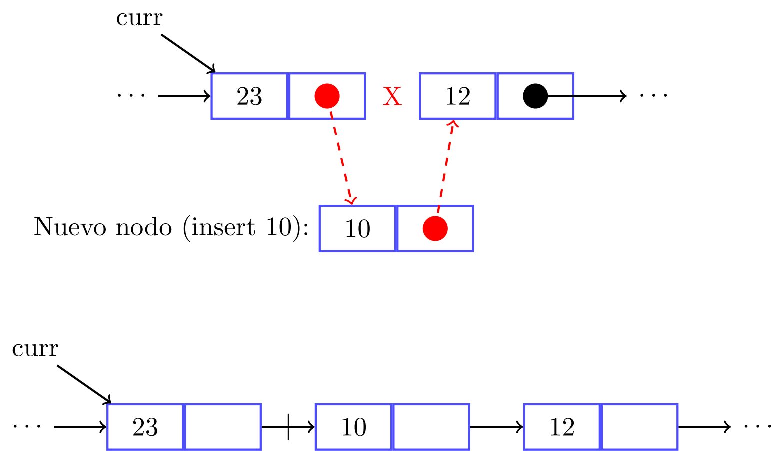

Figure 9: Actualización de apuntadores en la operación insertar.

El código 1.4 muestra la operación insert en el lenguage C.

```
1
2 ltemp = create_new(10);
3
4 ltemp->next = curr->next;
5 curr->next = ltemp;
6
7 if (tail == fence)
8    tail = curr->next;
```

La otra operación que mostraremos es el eliminado de un nodo. Supongamos que en lista ligada $< \cdots, 23|10, 13, \cdots >$ se desea eliminar el elemento 10. Lo único tenemos que hacer es actualizar los apuntadores en los nodos vecinos, de modo que el nodo previo al elemento 10 apunte al elementos que apunta 10. Si el apuntador tail, era el elememto a eliminar se tiene que actualizar este puntero. Por último se debe de liberar la memoria que ocupaba el elemento eliminado. Esto se muestra en la figura 10.

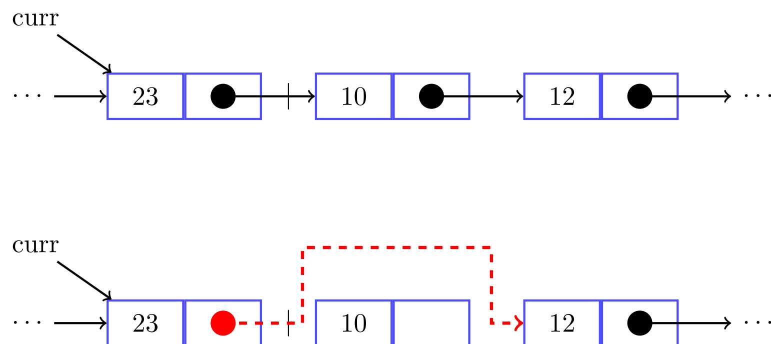

Figure 10: Eliminación de un nodo en una lista ligada.

El código 1.4 realiza la operación de actualizar los punteros al eliminar un node de una lista ligada.

```
1 // Remember link node with value 10
2 ltemp = curr->next;
3
4 //remove from the list and new link
5 curr->next = ltemp->next;
6
```

```
// Reset tail
if (tail == ltemp)
    tail = curr;

// Reclaim space
delete ltemp;
```

En resumen a continuación se muestras las ventajas y desventajas de implementar una lista con arreglos o con punteros:

- Arreglos:
  - Ventajas:
    * Solo requiere el espacio necesario para almacenar el valor del elemento
  - Desventajas:
    * Su tamaño es predeterminado
    * No pueden crecer
    * Una cantidad considerable de espacio se mantiene sin uso si la lista contiene unos cuantos elementos
- Punteros (listas ligadas)
  - Ventaja
    * Solo requieren espacio para los objetos actuales de la lista.
    * El numero de elementos puede variar
  - Desventaja
    * Requiere que un puntero sea agregado a cada nodo
    * Si el tamaño del elemento es pequeño, el costo de los punteros puede ser una fracción significante del almacenamiento total

1.5 Listas doblemente ligadas

Una lista ligada cuenta con dos apuntadores, uno que hace referencia al elemento previo y uno al elemento siguiente.

Las principales características de las listas doblemente ligadas son:

- Permiten un acceso conveniente al nodo previo y siguiente dado un nodo actual.
- Maximizan la capacidad de recorrido de la lista.
- Nodos con dos apuntadores, uno al siguiente elemento de la lista y otro al anterior.
- Las operaciones de inserción y eliminación son poco más costosas que en las listas simplemente ligadas, pues se tienen que actualizar los dos apuntadores.
- El espacio de almacenamiento también es mayor dado los dos apuntadores de cada nodo.

Al igual que las listas simplemente ligadas, las listas doblemente ligadas hacen uso de un puntero head, tail y curr. Head y tail apuntan al inicio y final de la lista, respectivamente. Ambos son nodos que no contienen valor y siempre existen. Curr es un puntero que hace referencia al elemento actual. Ver Figura 11

La inserción de un elemento en una lista ligada requiere re-direccionar varios punteros. Los pasos para insertar un nuevo nodo en una lista doblemente enlazada son:

- Crear el nodo a insertar.
- Hacer que el nuevo nodo apunte al nodo siguiente que apunta curr.

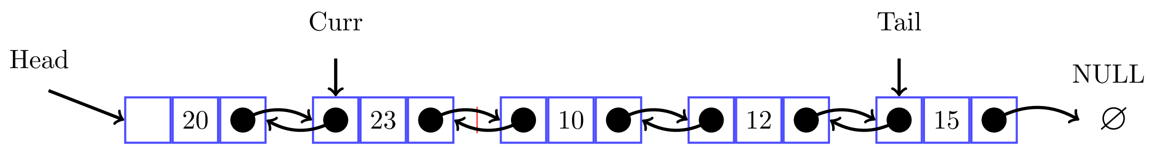

Figure 11: Lista doblemente ligada

- Hacer que el nodo siguiente a curr, en su apuntador previo, apunte al nuevo nodo.
- Hacer que el nuevo nodo, en si puntero previo, apunte a curr.
- Hacer que curr, en su puntero siguiente, apunte al nuevo nodo.

La figura 12 muestra un ejemplo de insertar un nodo en una lista doblemente ligada.

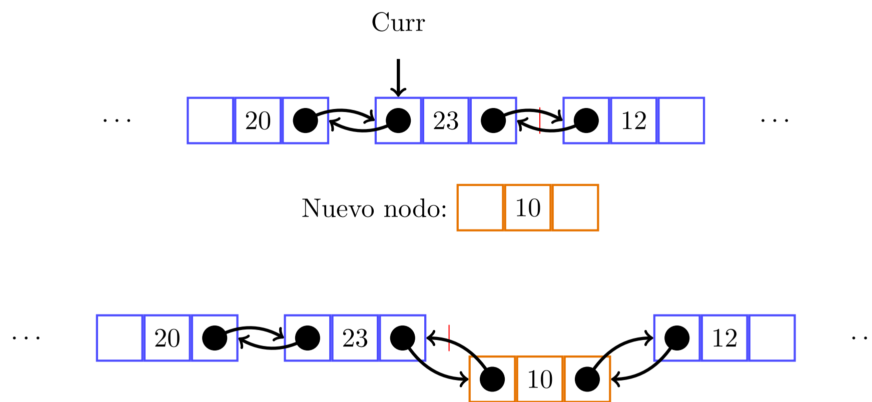

Figure 12: Insertar un nuevo nodo en una lista doblemente ligada.

Eliminar un elemento en una lista ligada requiere que el nodo curr (en su apuntador siguiente) no apunte a su nodo siguiente, sino a 2 delante de el. Y que el nodo que se encuentra dos posiciones delante de curr apunte a curr. Esto deja al nodo que se encuentra entre curr y dos posiciones delante de curr des-referenciado, ver figura 13. Al recorrer la lista este nodo no se tomara en cuenta. Y hay que recordar que en C++ se requiere liberar la memoria que este nodo estaba ocupando.

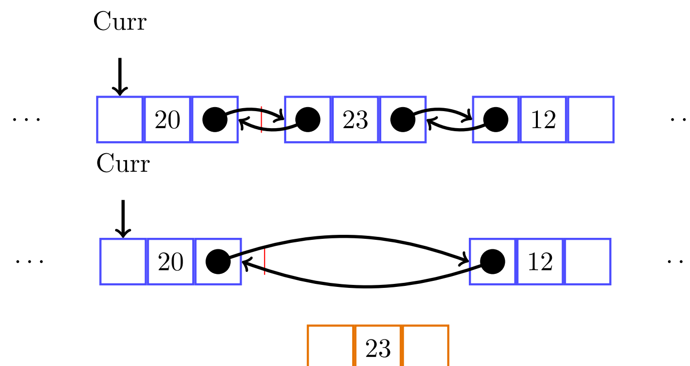

Figure 13: Eliminación de un nodo en una lista doblemente ligada.

En resumen, las listas doblemente ligadas tienen las siguiente características:

- Es posible recorrer la lista en ambas direcciones, hacia delante y hacia atrás.

- Revertir la lista es una operación más simple, solo requiere invertir las apuntadores head y tail.
- En una lista ligada, no hay forma de conocer el nodo previo; en una lista doblemente ligada se tiene un puntero directo al nodo previo.
- Requiere más espacio por el apuntador extra.
- Insertar/eliminar nodos requiere un mayor reajuste de punteros.

1.6 Listas circulares

Las lista circulares son un caso especial de lista ligadas. La principal característica de una lista circular es que el último elemento apunta al primero en lugar de a NULL. Son muy útiles en problemas (algoritmos) donde no hay un claro primer o último elemento. En una lista circular solo se tiene un apuntador a curr, que nos marca el elemento de referencia actual. Las listas circulares pueden ser listas ligadas simples, pero comúnmente se utilizan listas ligadas dobles para poder recorrer la lista en ambas direcciones. Las figura 14 muestra un ejemplo de lista ligada circular.

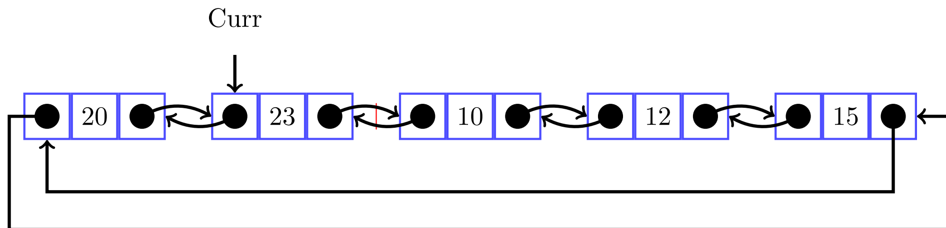

Figure 14: Lista ligada circular