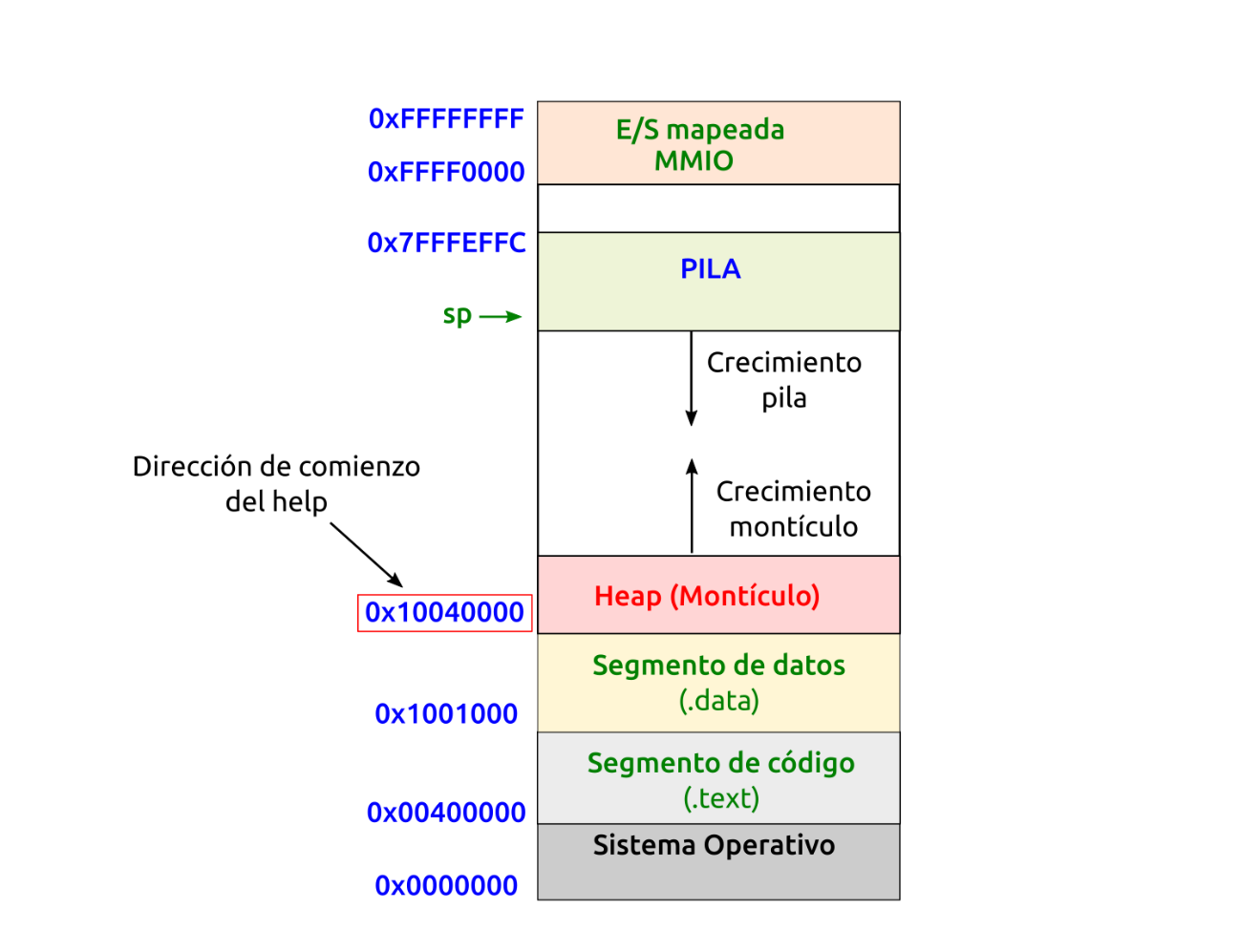
Capítulo 5

**Introducción**

Los programas utilizan **estructuras de datos** para implementar los **algoritmos** para realizan los cálculos. Mediante la **memoria dinámica** podemos crear estas estructuras, y hacer que vayan **creciendo dinámicamente**, según el programa lo vaya necesitando. Aprenderemos cómo utilizar la memoria dinámica en el RISC-V y veremos **ejemplos** de cómo implementar la estructura de datos más sencilla: la **lista enlazada**

## El montículo (Heap)

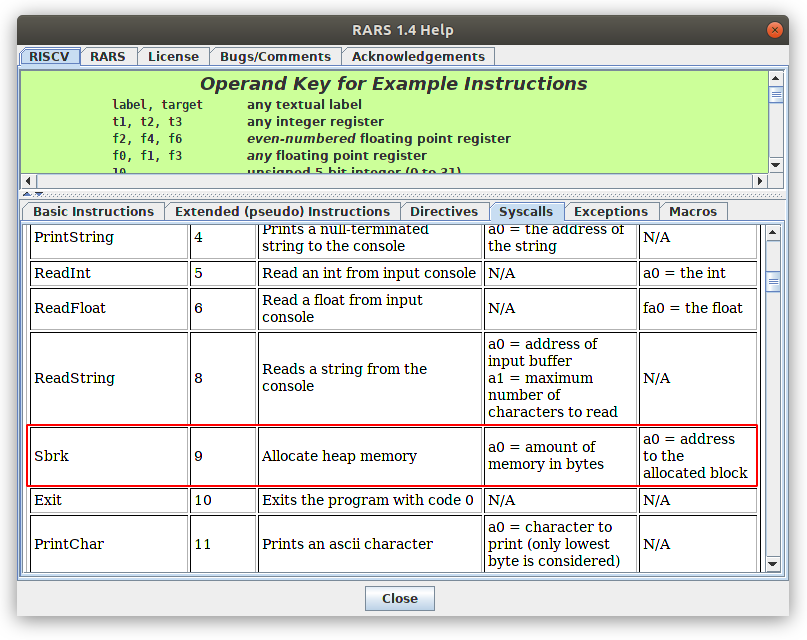
El **montículo** es la zona de la memoria destinada a la **reserva dinámica**. Es una zona que **crece hacia arriba** (hacia direcciones altas) y al igual que la pila, va variando su tamaño. El **mapa de memoria** que incluye todas las zonas que ya conocemos es este



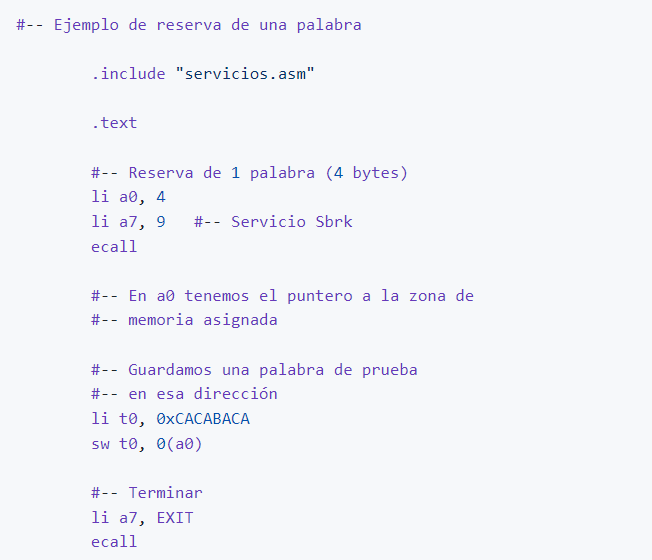
El montículo, en el simulador RARs, se encuentra situado a partir de la **dirección 0x1004000**. Es una zona de memoria que **controla el sistema operativo**, y por tanto tenemos que usar una **llamada al sistema** para reservar la **memoria dinámica** que queramos usar

## Reserva dinámica de memoria

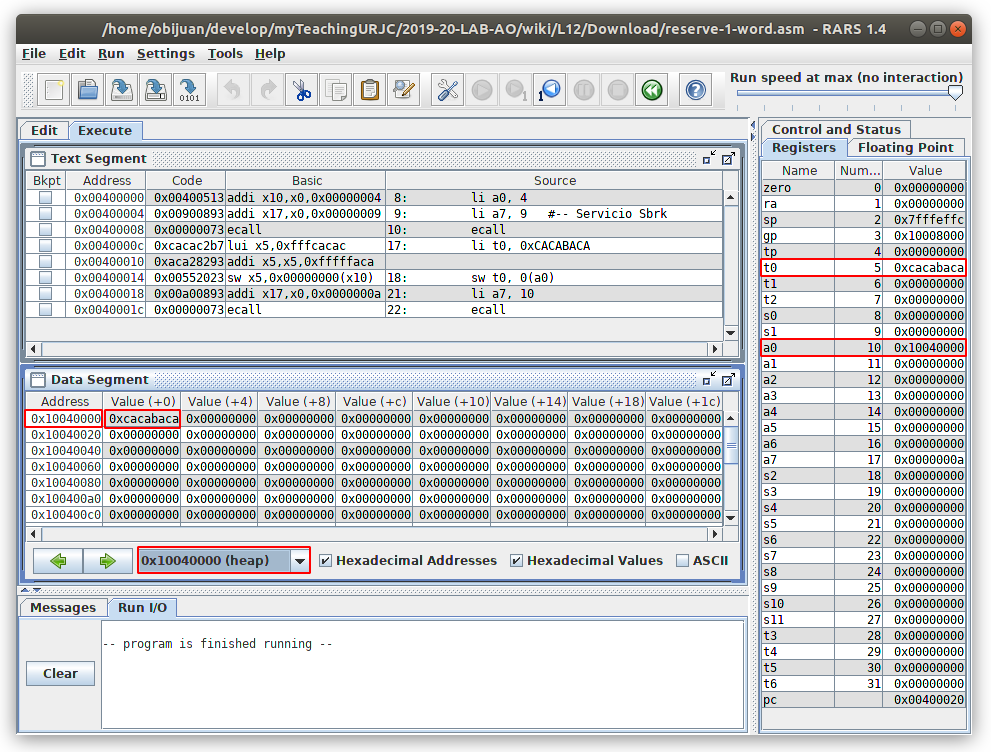
Para **utilizar** la memoria disponible en el montículo se la tenemos que pedir al **sistema operativo**. Esta operación se conoce como **reserva de memoria**. La realizamos invocando el **servicio 9** del **sistema operativo**. Le pasamos como parámetro el **número de bytes** que queremos reservar, por **a0**, y devuelve la **dirección** de la zona asignada, también por a0



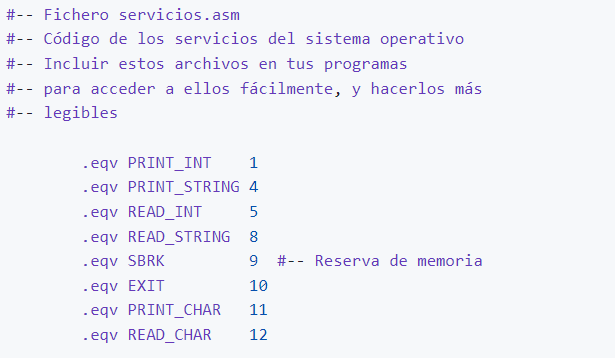
El puntero devuelto es **múltiplo de 4**, por lo que siempre se nos reservarán **un número entero de palabras**. Así, si le solicitamos que nos reserve 1 byte, en realidad nos reserva una palabra (4 bytes)



Lo **ejecutamos** y observamos los **registros**. La dirección asignada la vemos en **a0**. Es la **0x10040000**, justamente la primera dirección del montículo. En la parte inferior, en la **ventana de visualización** de los datos seleccionamos el **Heap** y comprobamos que se ha almacenado en la primera posición el valor **0xCACABACA**

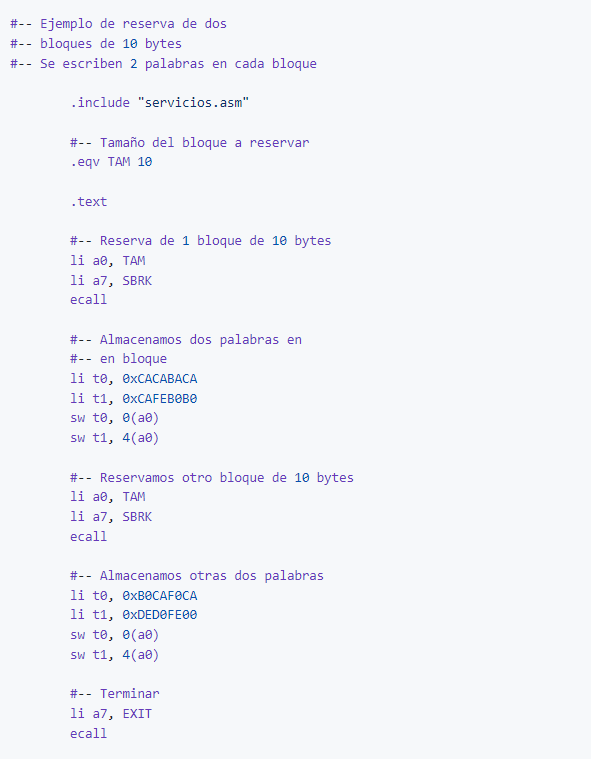


El servicio de reserva de memoria se denomina **SBRK**. A partir de ahora definiremos la **constante SBRK** con el **valor 9** y lo incluiremos en el fichero **servicios.asm** para usarlo en el resto de ejemplos y ejericicios

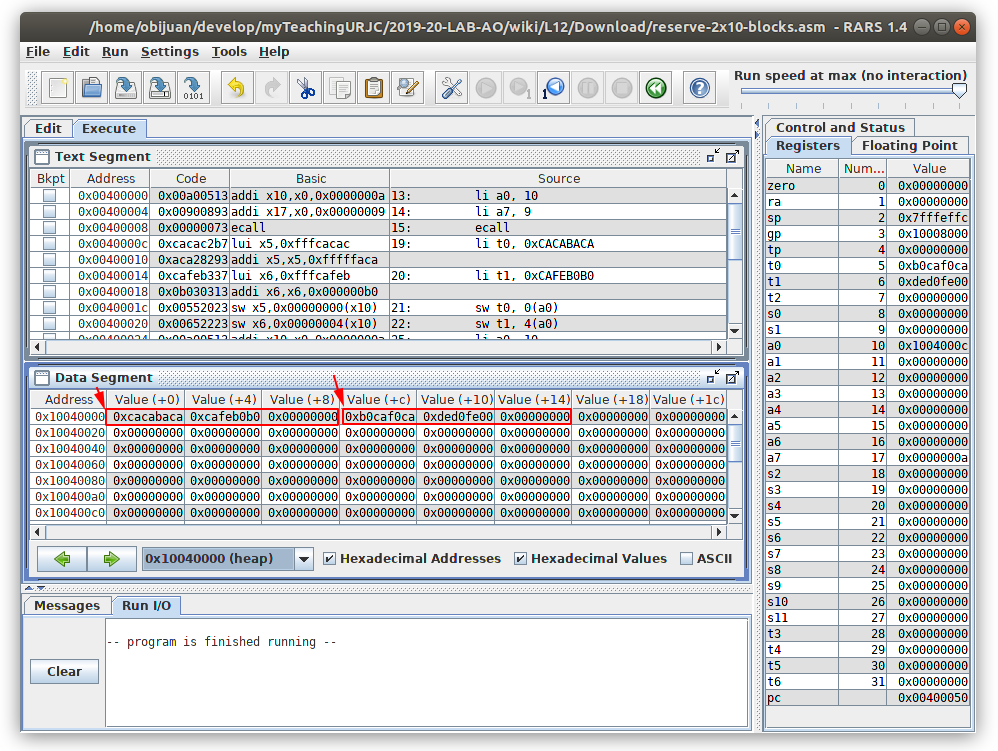


### Ejemplo 2: Reserva de dos bloques de 10 bytes

Como segundo ejemplo, reservaremos **dos bloques de 10 bytes**, almacenaremos **dos palabras** en cada bloque y veremos cómo quedan en **memoria**

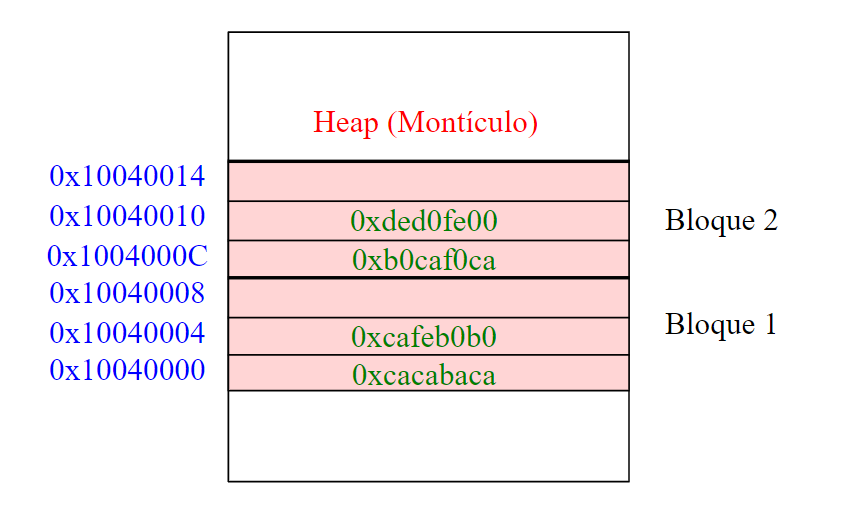


Lo **ejecutamos** y miramos lo que se ha almacenado en el **Heap**



Observamos que **los bloques de la reserva están alineados**: es decir, que la primera palabra del segundo bloque empieza en una dirección **múltiplo de 4**

Así es como queda el **HEAP** al ejecutar el programa



### Liberando la memoria reservada

Lo mismo que reservamos memoria para almacenar datos, también se puede **liberar esta memori**a para que el sistema operativozº se la pueda asignar a otros procesos. Sin embargo, el sistema operativo del simulador RARs **NO dispone** de esta opción todavía. Así que en los ejemplos que hagamos **NO liberaremos la memoria**, aunque en **las aplicaciones reales sí que hay que hacerlo**

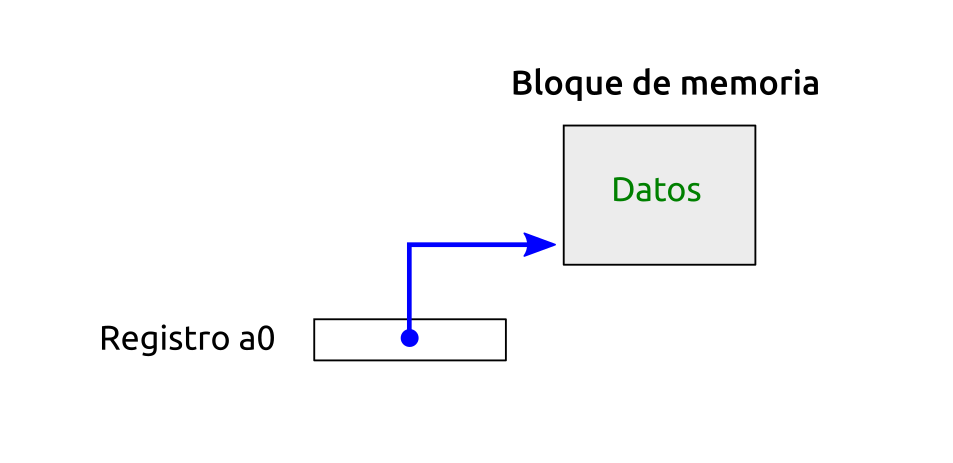
## Estructuras de datos

La **memoria dinámica** la usamos para construir **estructuras de datos**, cuyo **tamaño es variable** con la ejecución del programa. Ejemplo de estructuras de datos son las [listas](https://es.wikipedia.org/wiki/Lista_enlazada), [colas](https://es.wikipedia.org/wiki/Cola_(inform%C3%A1tica)), [pilas](https://es.wikipedia.org/wiki/Pila_(inform%C3%A1tica)), [árboles](https://es.wikipedia.org/wiki/%C3%81rbol_(inform%C3%A1tica)), etc.

En este apartado vamos a ver todos los **conceptos generales** comunes a **todas las estructuras de datos**, y en el siguiente lo aplicaremos a una estructura concreta: la lista enlazada

### Creando bloques

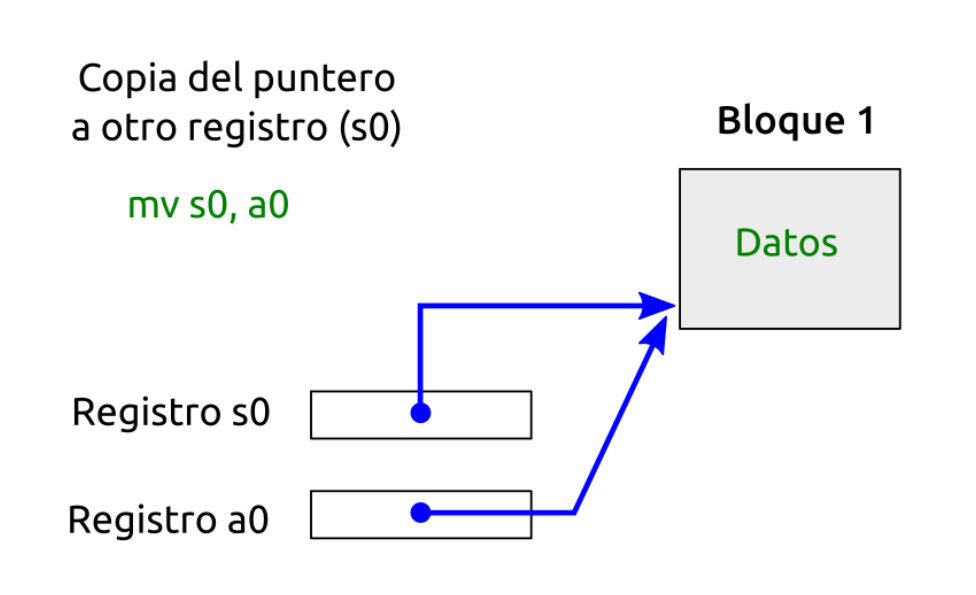
Ya sabemos **reservar bloques de memoria**. Lo representamos de la siguiente manera:



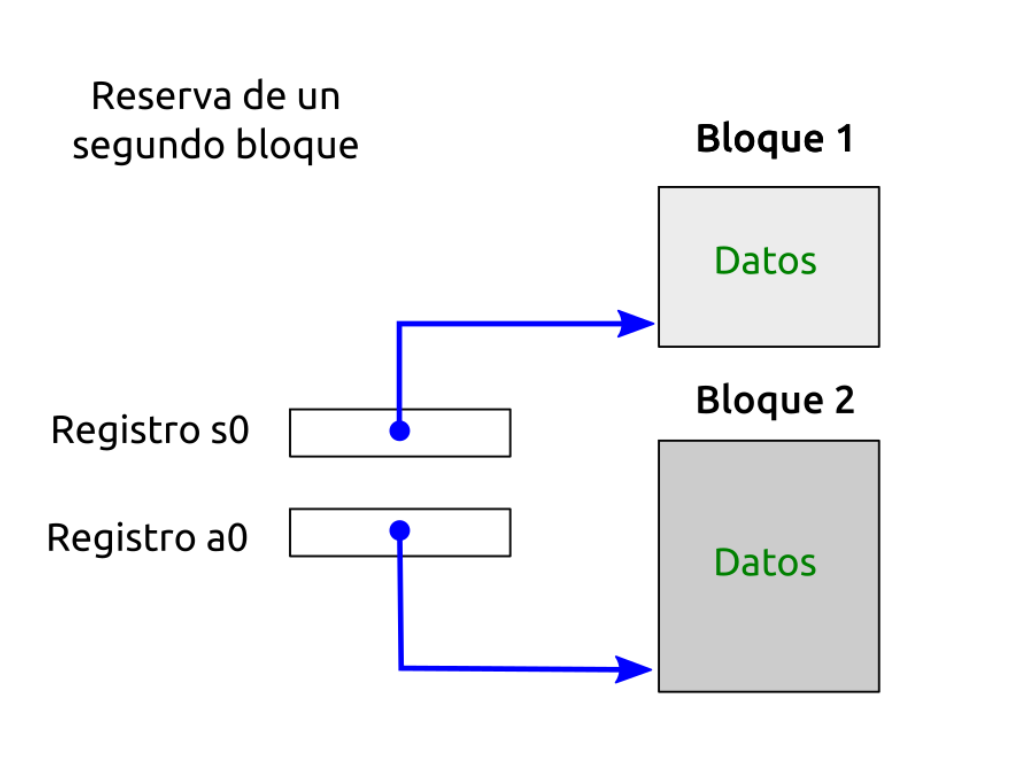
El sistema operativo nos devuelve la **dirección del bloque reservado** en a0. Es un **puntero** a ese bloque. Lo representamos mediante una **flecha** que sale del registro a0 y llega a la base del bloque: es la dirección de la **primera palabra** del bloque. A partir de esa dirección, y dado que conocemos el **tamaño del bloque**, podemos acceder a cualquier de las **palabras intermedias** usando las instrucciones **lw** y **sw** aplicando un **offset**: 0(a0), 4(a0), 8(a0), etc...

Es muy importante **NO perder nunca el puntero a ese bloque**. Si lo perdemos, no tendremos ninguna manera de acceder a él, dado que el sistema operativo nos lo ha asignado y la dirección donde está no la hemos elegido nosotros: no la conocemos

Por ello, si ahora **queremos crear otro bloque**, tendremos que **guardar** el puntero del bloque, que tenemos ahora en a0, en otro registro o en otra posición de memoria. Por ejemplo, vamos a guardarlo en s0. Esto lo representamos gráficamente así:

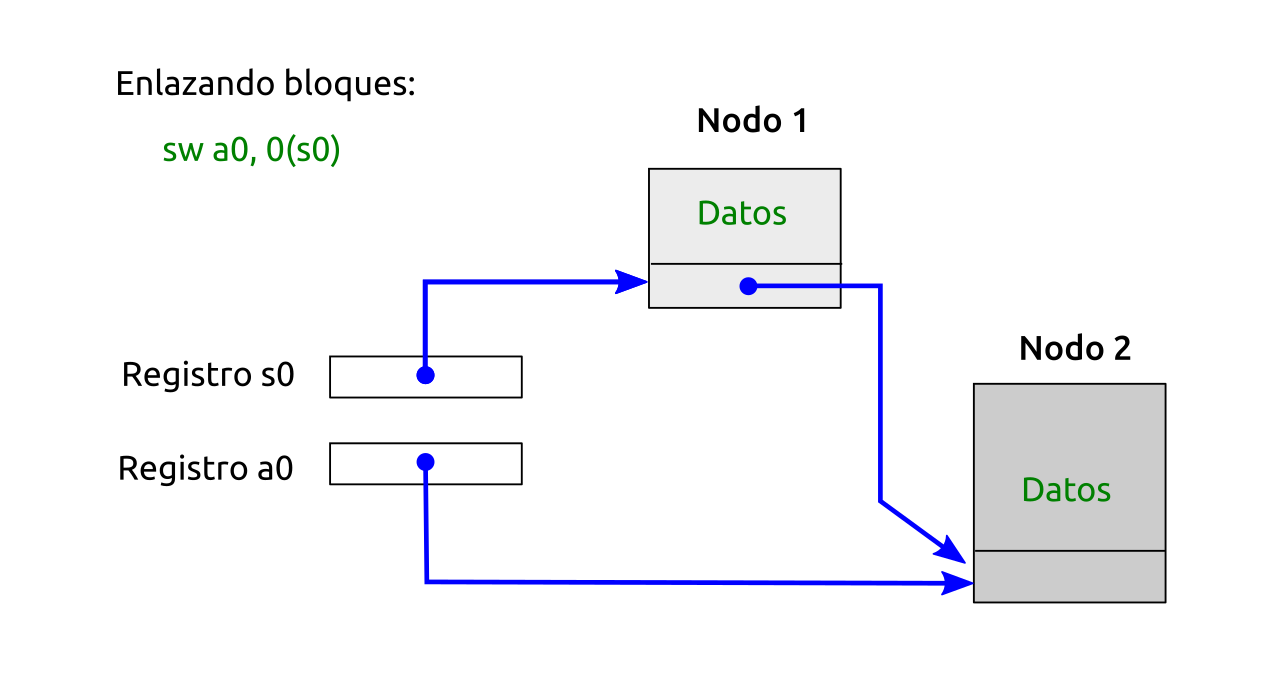


**Tenemos dos punteros**, que apuntan al **mismo bloque**. Ahora ya podemos reservar un **nuevo bloque** cuya dirección se guardará en a0. Como lo hemos copiado previamente, no lo perdemos. Ahora tenemos **dos bloques** con sus dos punteros para acceder a ellos



## Enlazando bloques

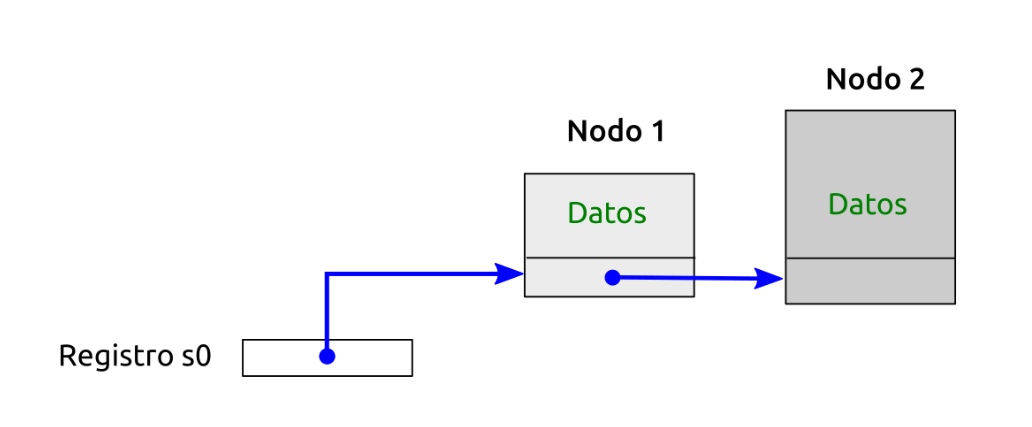
Estos bloques de memoria los podemos **enlazar entre sí**, de forma que un **bloque contenga la dirección de otro bloque**. En el ejemplo que estamos haciendo, si **almacenamos a0** en la **primera palabra** del bloque 1, hacemos que el bloque 1 apunte al 2. Esto lo conseguimos con la instrucción: sw a0, 0(s0)



Ahora los bloques se dice que están **enlazados**. Son por tanto parte de la **misma estructura**. Y en vez de bloques, los llamamos **nodos**, para indicar que son **elementos** que pertenecen a la **misma estructura de datos**. Hemos definido una estructura de datos, que tiene **dos nodos diferentes**

También vemos que el nodo 2 está **referenciado** dos veces. Está apuntado por el nodo 1 y por el registro a0

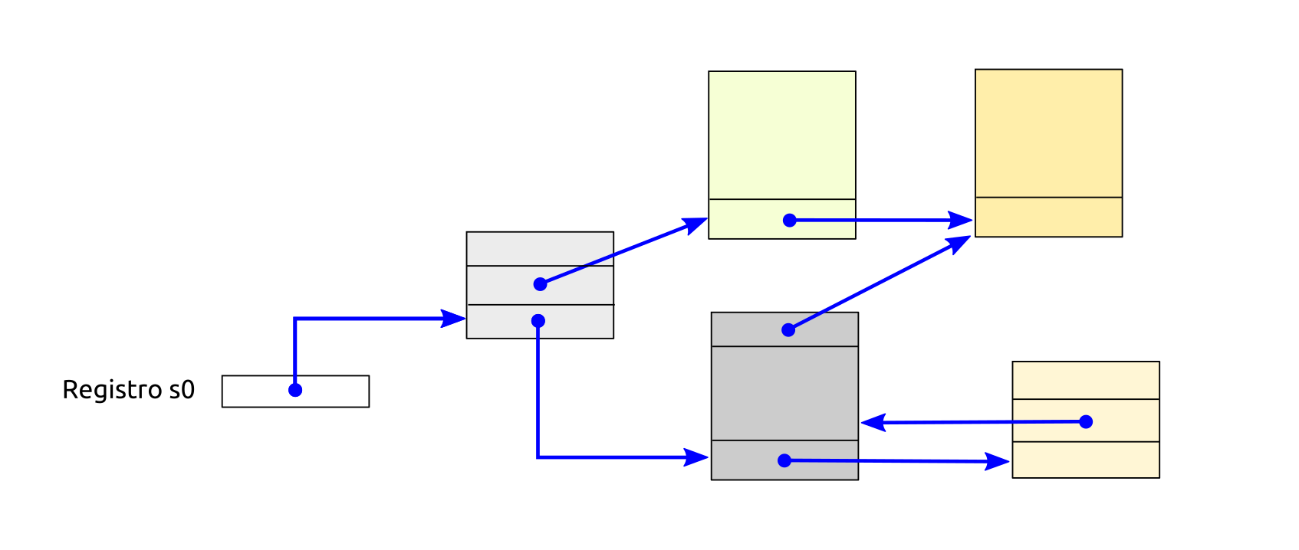
Si ahora usamos el registro a0 para otra cosa, perdemos una referencia, y la estructura de datos que nos queda la representamos de esta forma



**Todos los nodos están referenciados**. Para acceder al nodo 1 hay que usar el registro s0. Y para acceder al nodo 2 la primera palabra del nodo 1

### **Creando estructuras complejas**

**Combinando** estas dos acciones simples: **creación** de un bloque y **enlazado**, construimos estructuras de datos mucho **más complejas**. En el caso más **general**, los nodos puedes ser todos diferentes entre sí, y estar enlazados con muchos otros nodos (no sólo con uno)



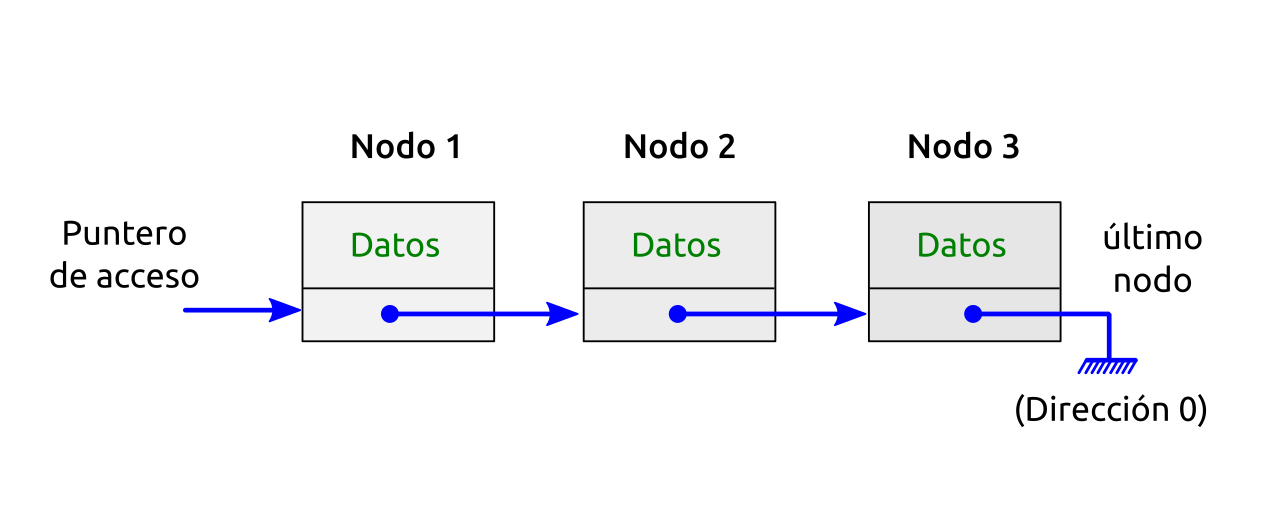
Si nos fijamos, vemos que los nodos tienen formas distintas, esto nos quiere decir que pueden tener datos distintos y no necesariamente tienen que tener el mismo tipo de datos, tamaño, etc… También observamos que en un mismo nodo podemos tener la dirección de más de un nodo.

## Listas enlazadas

Una de las **estructuras básicas** es la [lista enlazada](https://es.wikipedia.org/wiki/Lista_enlazada). Usaremos una **lista simplemente enlazada** para aprender cómo implementarla en el ensamblador del RISC-V

En una lista **todos los nodos son iguales**: tienen el **mismo tamaño** y la **misma estructura**. Cada nodo tiene **dos campos**: uno con **datos** y otro con un **puntero al siguiente nodo**

Este es un ejemplo de una **representación** de una lista simplemente enlazada, de **3 nodos**



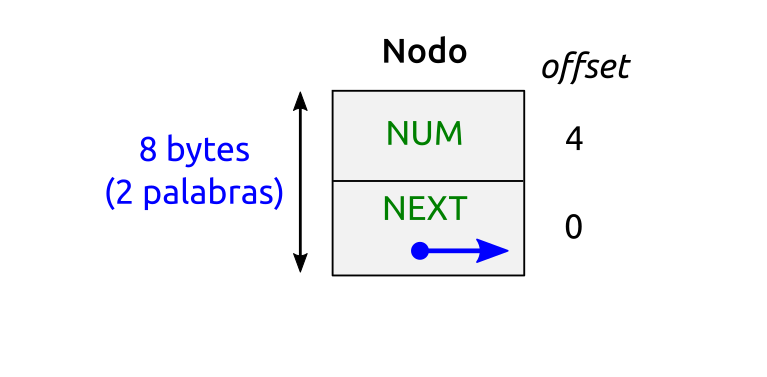
En todos los nodos hay un campo que contiene el **enlace al siguiente nodo**: es un **puntero**. Contiene la dirección de memoria en donde se encuentra almacenado el siguiente nodo. Un **valor de 0** (NULL) indica que es **el último**. Gráficamente lo indicamos con el **símbolo** de **conexión a tierra**

Accedemos al **primer elemento** de la lista mediante un **puntero de acceso**. A partir de acceder al primer nodo, ya podemos saltar a los siguientes nodos y leer información de ellos o modificarlos

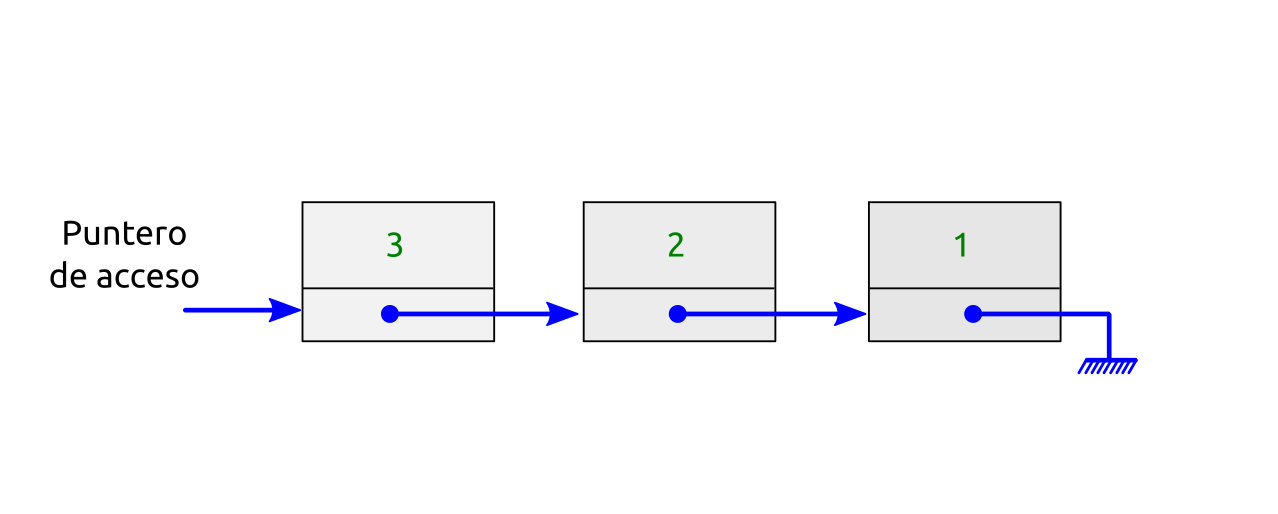
### Ejemplo: Lista de números enteros

Para practicar vamos a trabajar con una **lista de número enteros**. Los nodos tienen **dos palabras** (8 bytes). La situada en la dirección más baja es el **puntero al siguiente nodo**. Este campo lo llamaremos **NEXT**. La siguiente palabra, situada en el **offset 4**, contendrá el número entero. Este campo lo llamamos **NUM**

Esta es la **estructura de los nodos**



Este es el aspecto que tiene la lista cuando hemos introducido los **valores 1, 2** y **3**:

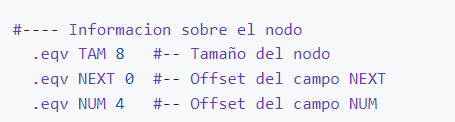


Para que sea más fácil su implementación, **esta lista crece hacia la izquierda**, de forma que el primer número introducido en realidad es el último en la lista, y el último introducido está en el primer nodo

### **Creando un nodo**

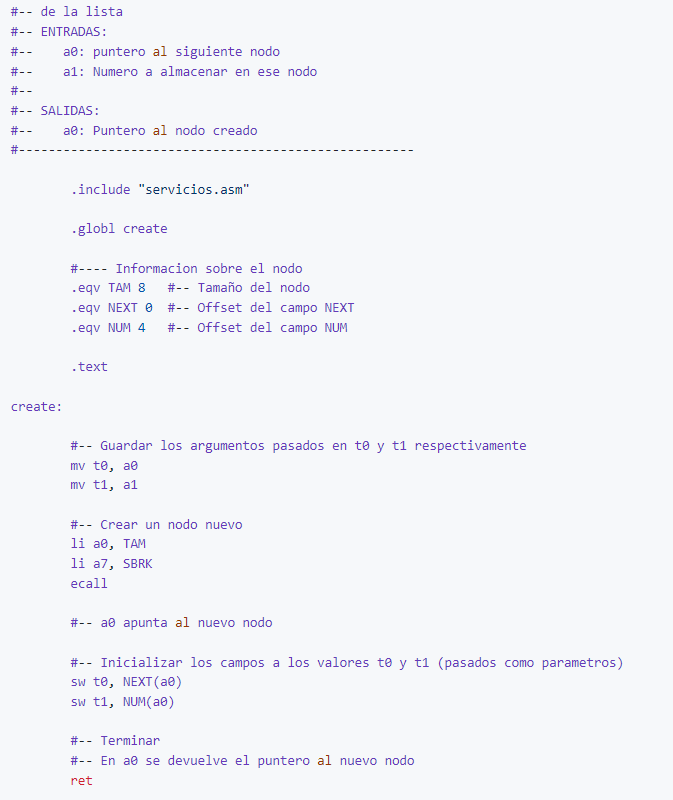
Empezamos creando la **subrutina create()** para crear un nodo e inicializarlo. La subrutina recibe los valores de los campos NEXT y NUM por los registros **a0** y **a1** respectivamente y reserva **8 bytes** de memoria. **Almacena** los valores de los campos y devuelve por **a0** el puntero al nuevo nodo

La **información del nodo** la definimos en **constantes**: su tamaño y el offset de cada uno de los campos:

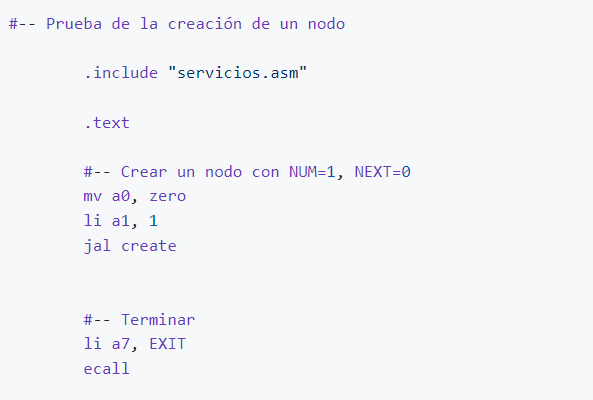


De esta forma el **acceso a los campos** es más sencillo y **menos propenso a errores**. Así, Por ejemplo, para **leer el número** almacenado hay que utilizar esta instrucción:

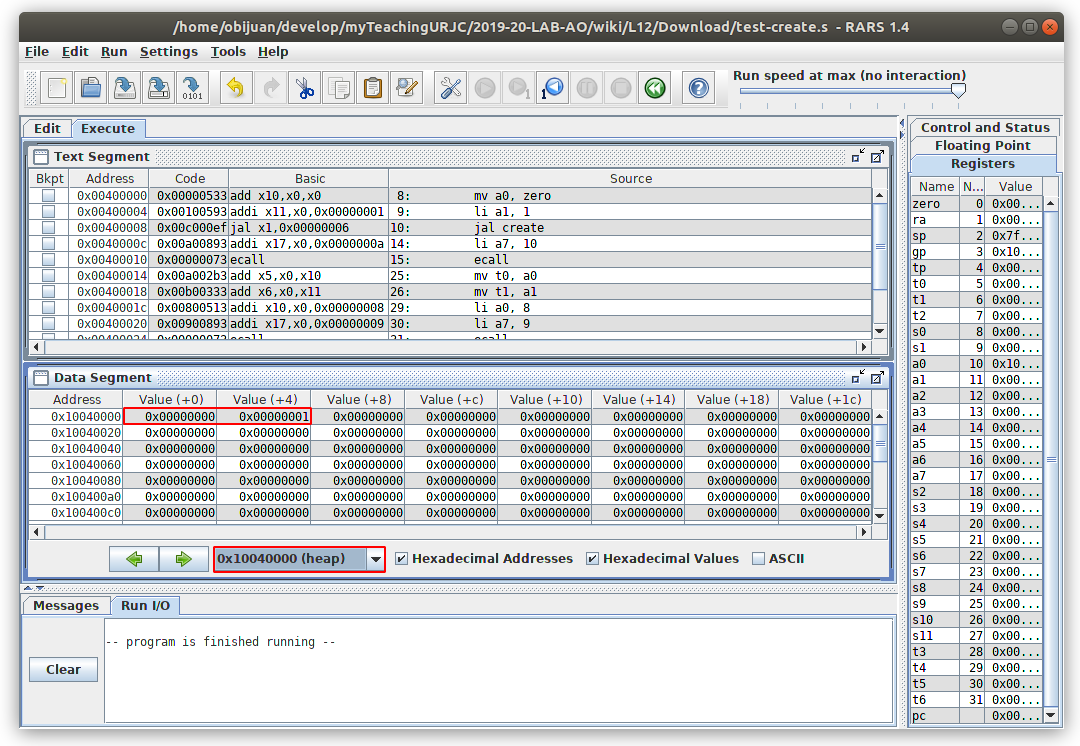
La **subrutina completa** es la siguiente:



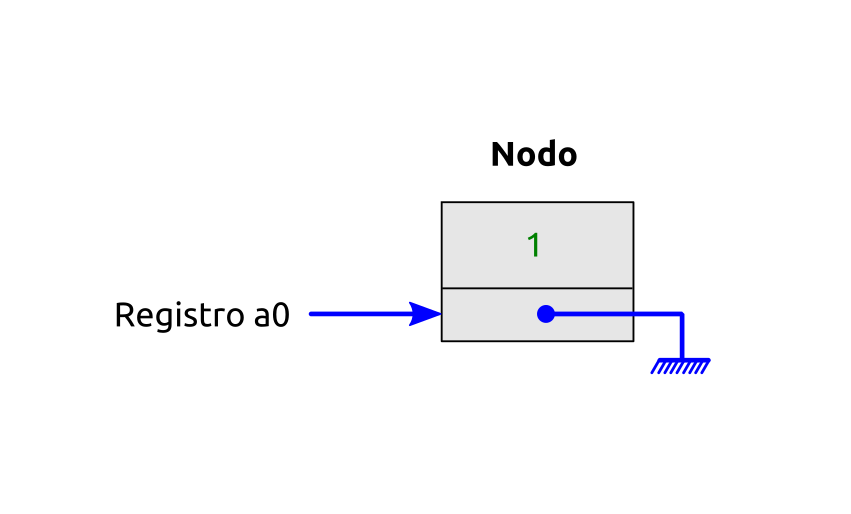
Para probar la **subrutina create()** y comprobar que funciona bien, usamos este **programa principal**, que crea un **nodo** con un **1** en el **campo NUM** y un **0** en **NEXT**



Lo **ejecutamos**, seleccionamos la visualización del **HEAP** y comprobamos que primero hay almacenado un 0 y luego un 1

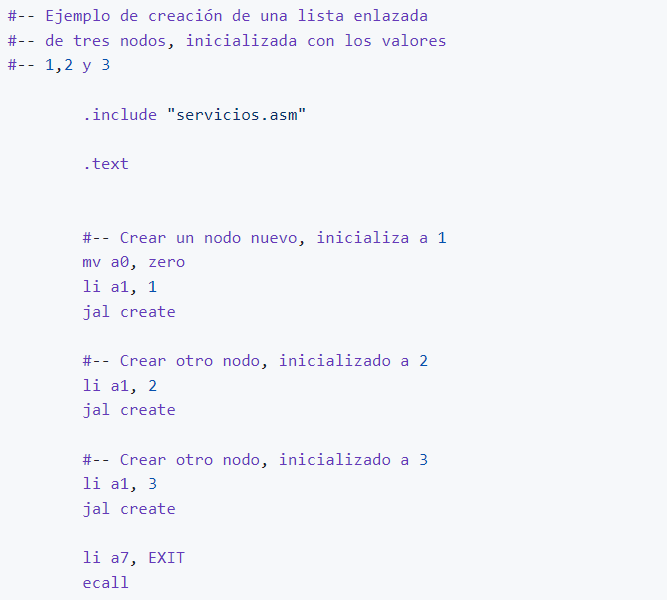


El **nodo creado** lo representamos así:

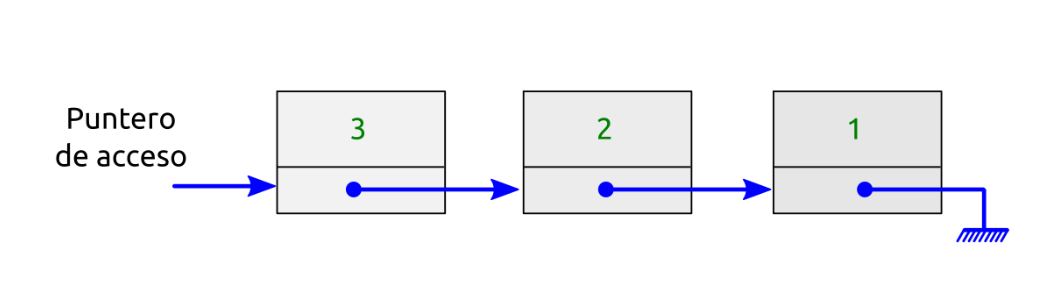


### Creando la lista

Como ejemplo, creamos una **lista con 3 nodos**, cuyos datos serán 1, 2 y 3. Basta con **llamar tres veces** consecutivas a la **función de crear nodo**, pasando como argumento del **campo NEXT** el puntero al nodo anteriormente creado, y como argumento para el **campo NUM** los números 1, 2 y 3 respectivamente

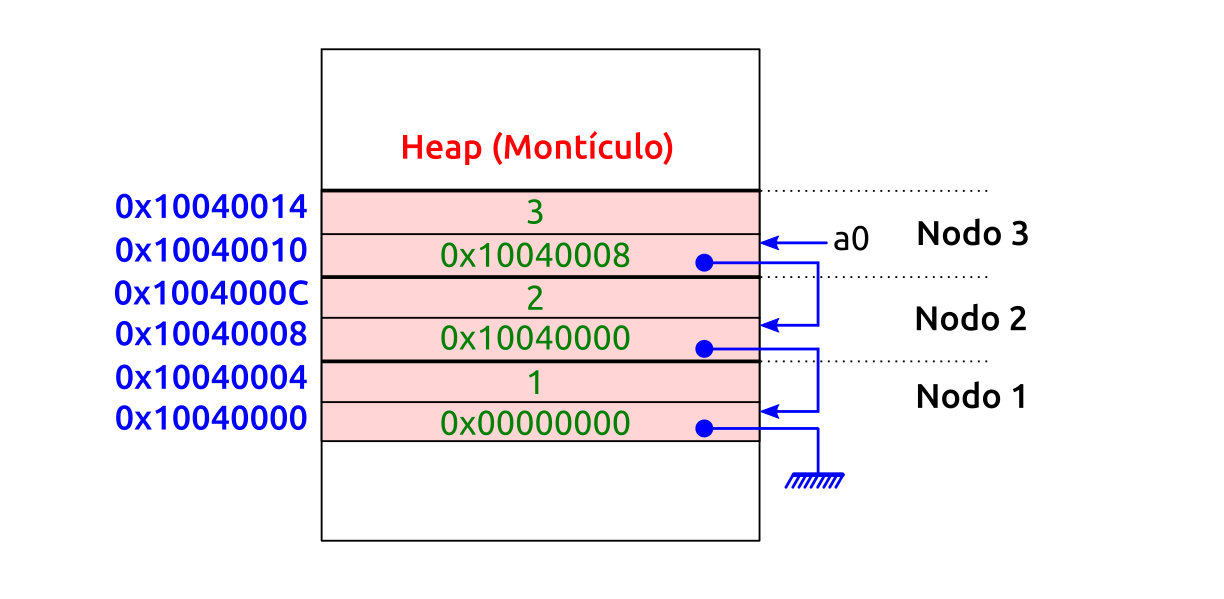


Sólo la primera vez tenemos que pasarle el valor a0, puesto que cada vez que entramos en la función créate, esta nos devuelve el nuevo valor a0. Con este programa hemos creado lo que se representa en la siguiente imagen.



Primero se entraría en el nodo donde pondremos el valor a 1 y le pasamos en a0=0 puesto que es el último nodo y nodo siguiente. Al salir de este nodo por a0 nos retorna el puntero en el que estaría el siguiente nodo en caso de que lo creásemos y es este puntero el que le pasamos a la funciona créate para que sepa donde tiene que empezar el siguiente nodo.

En este dibujo se muestra el **Heap**, y cómo la lista creada está **mapeada** en él



### Imprimiendo la lista

Para **imprimir los valores** almacenados en la lista, hay que **visitar todos los nodos**, empezando por el apuntado por **puntero de acceso**. Esta operación se denomina **recorrer la lista**

Como ejemplo, vamos a **imprimir la lista** creado en los apartados anteriores, que contiene los números 1, 2 y 3. Lo haremos usando un **algoritmo iterativo** y uno **recursivo**

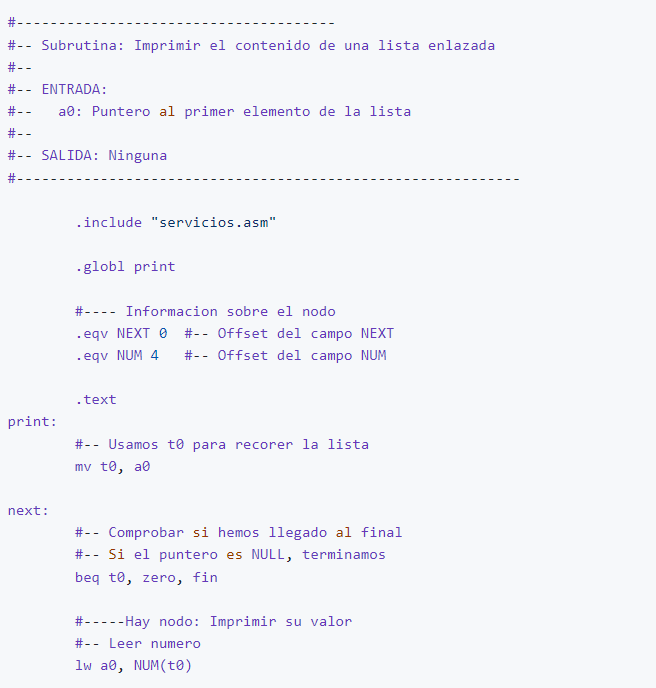
#### Algoritmo iterativo

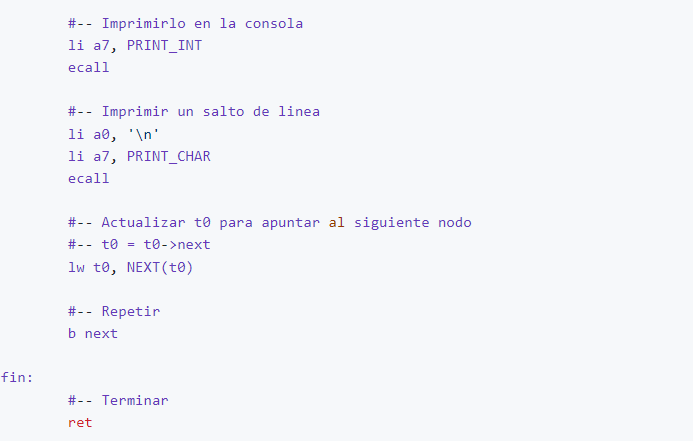
Para recorrer la lista utilizaremos un **puntero índice** que irá apuntando a los nodos de la lista de forma secuencia. Este puntero inicialmente será igual al **puntero de acceso**

Si este **puntero es NULL**, entonces hemos **terminado** de recorrer la lista. De lo contrario, se trata de un nodo. Leemos el **número** contenido en este nodo y lo **imprimimos**. A continuación, leemos el **puntero al siguiente nodo** (campo NEXT) y se lo asignamos a nuestro puntero índice

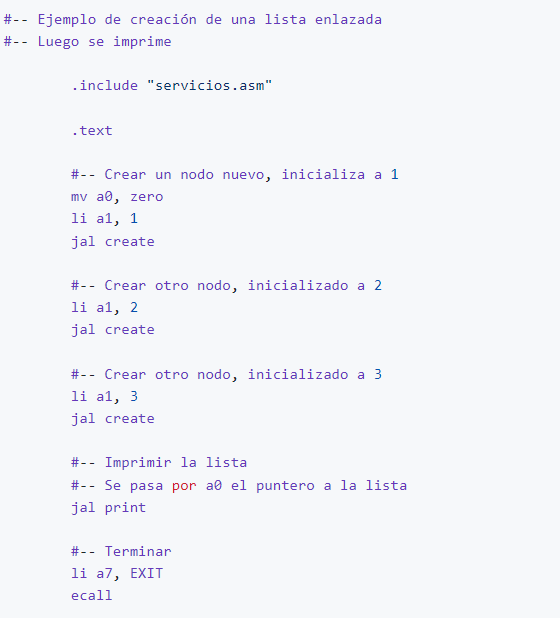
La función para imprimir la lista es **print(**), y como argumento se le pasa el **puntero de acceso**

El **código** es el siguiente:

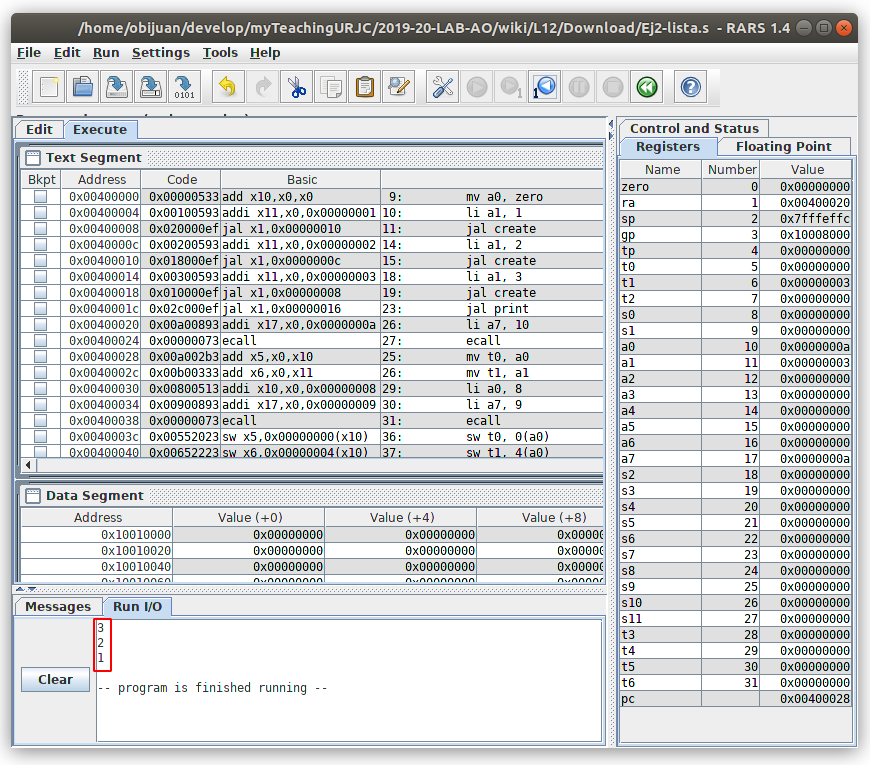




Para probarlo ampliamos el **programa principal** que ya teníamos, para llamar a la **función print** una vez creada la lista:



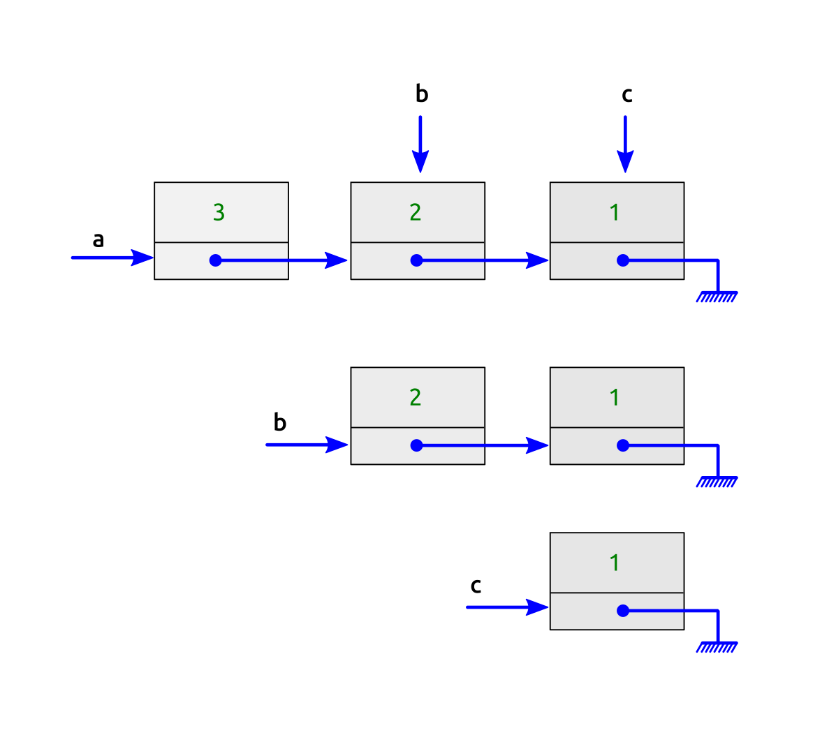
Ahora lo **ejecutamos**. Vemos que se imprime en la consola, en orden inverso. Claro, es por el orden en el que hemos creado la lista (por la izquierda)



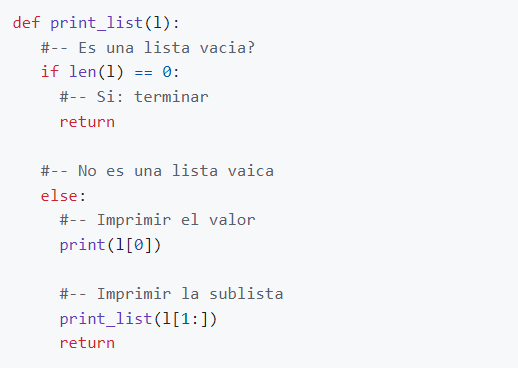
#### Algoritmo recursivo

Las **listas**, al igual que sucecía con las cadenas, son **elementos recursivos**. Dentro de ellas encontramos otras listas. Hay un patrón estructural que se repite en varios niveles. Por ello podemos utilizar también **algoritmos recursivos**

En el ejemplo que estamos haciendo, según el **puntero de acceso** a la lista usado, tendremos una lista de tres elementos, de dos, de uno o una lista vacía. Estas sublistas están contenidas en la lista original

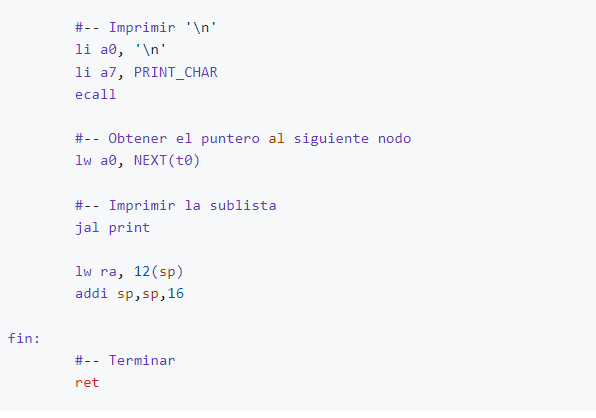


El **algoritmo** que implementaremos para **imprimir la lista** de forma **recursiva** es el siguiente:



Y esta es su **implementación** en ensamblador





El **programa principal** es el **mismo** que el que hemos usado con el algoritmo iterativo