III. GESTIÓN DE LA MEMORIA EN TIEMPO DE EJECUCIÓN

III.1. Organización de la memoria en tiempo de ejecución

Cuando un programa se ejecuta sobre un sistema operativo existe un proceso previo llamado cargador que suministra al programa un bloque contiguo de memoria sobre el cual ha de ejecutarse. El programa resultante de la compilación debe organizarse de forma que haga uso de este bloque. Para ello el compilador incorpora al programa objeto el código necesario.

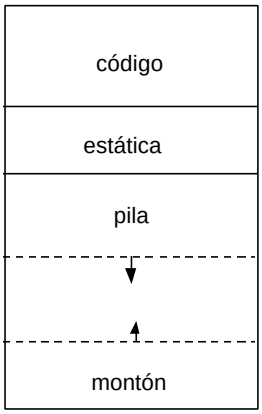


fig. 1. Organización de la memoria en la ejecución.

Las técnicas de gestión de la memoria durante la ejecución del programa difieren de unos lenguajes a otros, e incluso de unos compiladores a otros. En este capítulo se estudia la gestión de la memoria que se utiliza en lenguajes procedurales como FORTRAN, PASCAL, C, MODULA-2, etc. La gestión de la memoria en otro tipo de lenguajes (funcionales, lógicos, ..) es en general diferente de la organización que aquí se plantea.

Para lenguajes imperativos, los compiladores generan programas que tendrán en tiempo de ejecución una organización de la memoria similar (a grandes rasgos) a la que aparece en la figura 1. En este esquema se distinguen las secciones de:

III.2. El código

Es la zona donde se almacenan las instrucciones del programa ejecutable en código máquina, y también el código correspondiente a los procedimientos y funciones que utiliza. Su tamaño puede fijarse en tiempo de compilación.

Algunos compiladores fragmentan el código del programa objeto usando "*overlays*". Estos "*overlays*" son secciones de código objeto que se almacenan en ficheros independientes y que se cargan en la memoria central (RAM) dinámicamente, es decir, durante la ejecución del programa. Los *overlays* de un programa se agrupan en zonas y módulos, cada uno de los cuales contiene un conjunto de funciones o procedimientos.

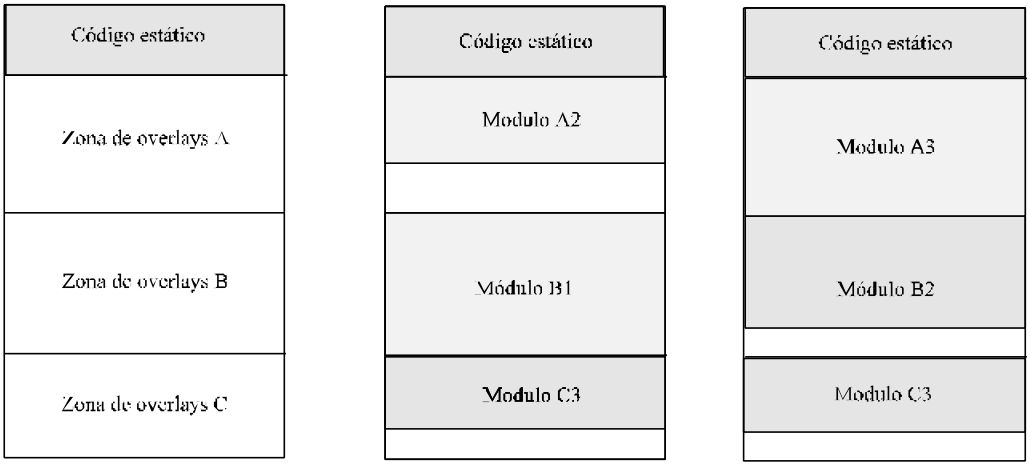


fig.2. Ejemplos de configuraciones de la zona de código utilizando "*overlays*"

Durante el tiempo de ejecución sólo uno de los módulos de cada uno de los *overlays* puede estar almacenado en memoria. El compilador reserva en la sección de código una zona contigua de memoria para cada *overlay*. El tamaño de esta zona debe ser igual al del mayor módulo que se cargue sobre ella. Es función del programador determinar cuantas zonas de *overlay* se definen, qué funciones y procedimientos se

encapsulan en cada módulo, y como se organizan estos módulos para ocupar cada uno de los *overlays*. Una restricción a tener en cuenta es que las funciones de un módulo no deben hacer referencia a funciones de otro módulo del mismo *overlay*, ya que nunca estarán simultáneamente en memoria.

El tiempo de ejecución de un programa con *overlays* es mayor, puesto que durante la ejecución del programa es necesario cargar cada módulo cuando se realiza una llamada a alguna de las funciones que incluye. También es tarea del programador diseñar la estructura de *overlays* de manera que se minimice el número de estas operaciones. La técnica de *overlays* se utiliza cuando el programa a compilar es muy grande en relación con la disponibilidad de memoria del sistema, o bien si se desea obtener programas de menor tamaño.

III.3. La Memoria Estática

La forma mas fácil de almacenar el contenido de una variable en memoria en tiempo de ejecución es en memoria estática o permanente a lo largo de toda la ejecución del programa. No todos los objetos (variables) pueden ser almacenados estáticamente. Para que un objeto pueda ser almacenado en memoria estática su tamaño (número de bytes necesarios para su almacenamiento) ha de ser conocido en tiempo de compilación. Como consecuencia de esta condición no podrán almacenarse en memoria estática:

? Los objetos correspondientes a procedimientos o funciones recursivas, ya que en tiempo de compilación no se sabe el número de variables que serán necesarias.

? Las estructuras dinámicas de datos tales como listas, árboles, etc. ya que el número de elementos que las forman no es conocido hasta que el programa se ejecuta.

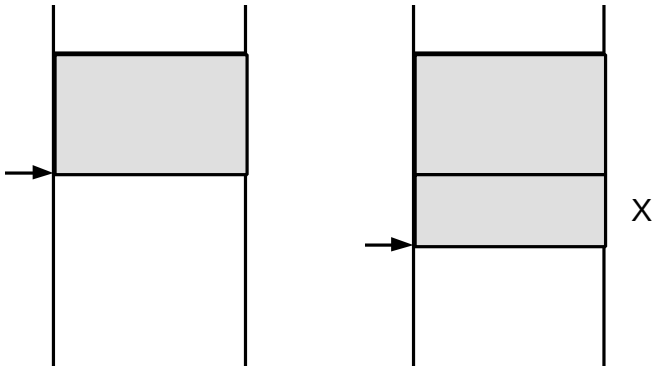


fig.3. Alojamiento en memoria de un objeto X

Las técnicas de asignación de memoria estática son sencillas. A partir de una posición señalada por un puntero de referencia se aloja el objeto X, y se avanza el puntero tantos bytes como sean necesarios para almacenar el objeto X. La asignación de memoria puede hacerse en tiempo de compilación y los objetos están vigentes desde que comienza la ejecución del programa hasta que termina.

En los lenguajes que permiten la existencia de subprogramas, y siempre que todos los objetos de estos subprogramas puedan almacenarse estáticamente (por ejemplo en FORTRAN-IV) se aloja en la memoria estática un registro de activación correspondiente a cada uno de los subprogramas. Estos registros de activación contendrán las variables locales, parámetros formales y valor devuelto por la función, tal como indica la fig. 4b.

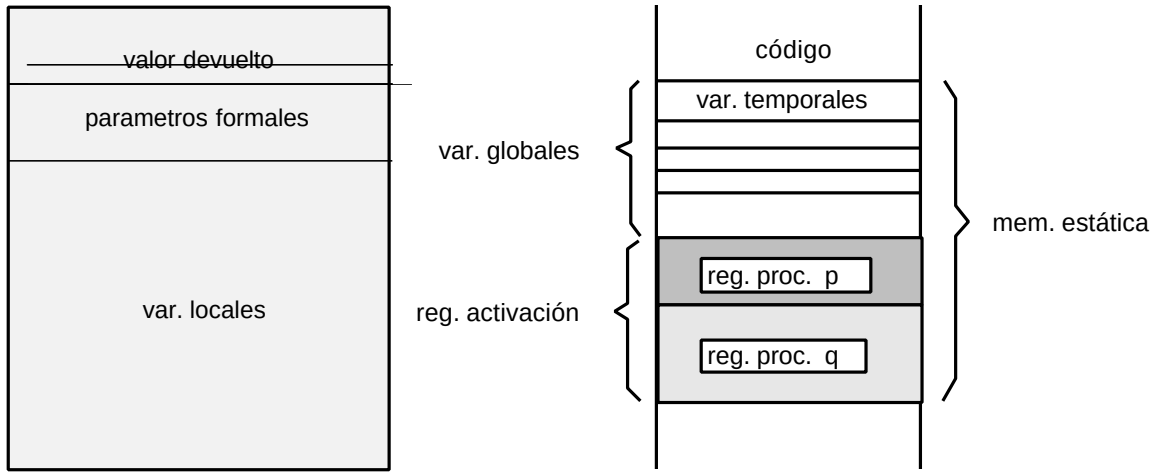


fig. 4a. Registro de activación **fig. 4b. Estructura de la memoria estática en FORTRAN-IV**

Dentro de cada registro de activación las variables locales se organizan secuencialmente. Existe un solo registro de activación para cada procedimiento y por tanto no están permitidas las llamadas recursivas. El proceso que se sigue cuando un procedimiento p llama a otro q es el siguiente:

? p evalúa los parámetros de llamada, en caso de que se trate de expresiones complejas, usando para ello una zona de memoria temporal para el almacenamiento intermedio. Por ejemplos, si la llamada a q es q((3*5)+(2*2),7) las operaciones previas a la llamada propiamente dicha en código máquina han de realizarse sobre alguna zona de memoria temporal. (En algún momento debe haber una zona de memoria que contenga el valor intermedio 15, y el valor intermedio 4 para sumarlos a continuación). En caso de utilización de memoria estática ésta zona de temporales puede ser común a todo el programa, ya que su tamaño puede deducirse en tiempo de compilación.

? q inicializa sus variables y comienza su ejecución.

Dado que las variables están permanentemente en memoria es fácil implementar la propiedad de que conserven o no su contenido para cada nueva llamada

III.4. La Pila

La aparición de lenguajes con estructura de bloques trajo consigo la necesidad de técnicas de alojamiento en memoria más flexibles, que pudieran adaptarse a las demandas de memoria durante la ejecución del programa. En estos lenguajes, cada vez que comienza la ejecución de un procedimiento se crea un registro de activación para contener los objetos necesarios para su ejecución, eliminándolo una vez terminada ésta.

Dado que los bloques o procedimientos están organizados jerárquicamente, los distintos registros de activación asociados a cada bloque deberán colocarse en una pila en la que entrarán cuando comience la ejecución del bloque y saldrán al terminar el mismo. (fig. 5b) La estructura de los registros de activación varía de unos lenguajes a otros, e incluso de unos compiladores a otros. Este es uno de los problemas por los que a veces resulta difícil enlazar los códigos generados por dos compiladores diferentes. En general, los registros de activación de los procedimientos suelen tener algunos de los campos que pueden verse en la fig. 5a.

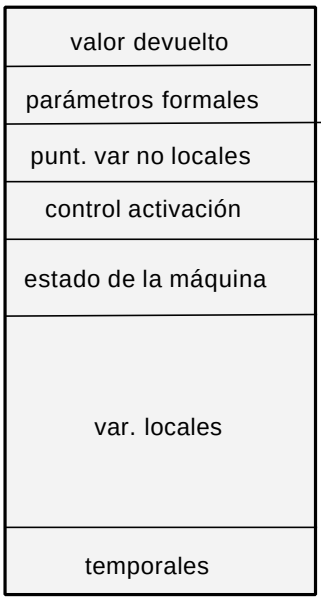


fig. 5a. Registro de activación

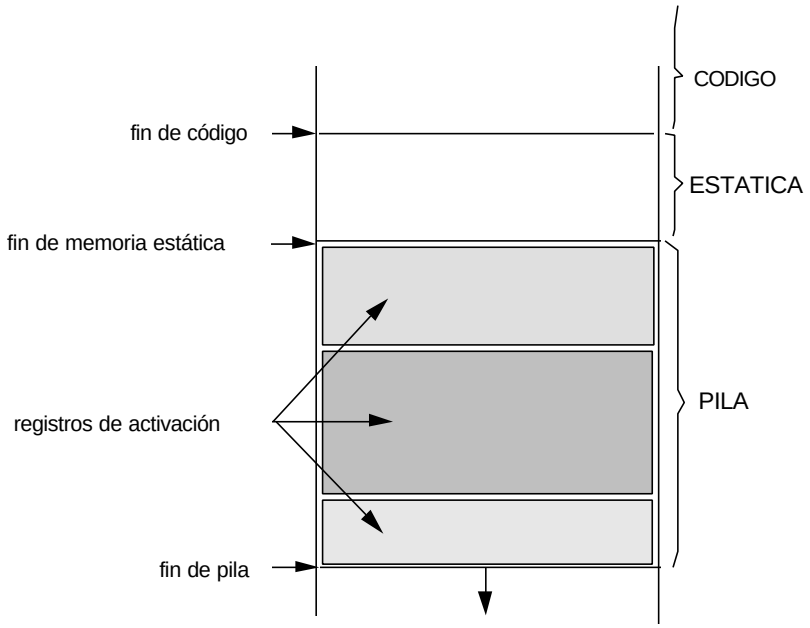


fig. 5b. Estructura de la pida

El *puntero de control de activación* guarda el valor que tenía el puntero de la cima de la pila antes de que entrase en ella el nuevo registro, de esta forma una vez que se desee desalojarlo puede restituirse el puntero de la pila a su posición original. Es decir, es el puntero que se usa para la implementación de la estructura de datos "Pila" del compilador

En la zona correspondiente al *estado de la máquina* se almacena el contenido que hubiera en los registros de la máquina antes de comenzar la ejecución del procedimiento. Estos valores deberán ser repuestos al finalizar la ejecución del procedimiento. El código encargado de realizar la copia del estado de la máquina es común para todos los procedimientos.

El *puntero a las variables no locales* permite el acceso a las variables declaradas en otros procedimientos. Normalmente no es necesario usar este campo puesto que puede conseguirse lo mismo con el puntero de control de activación, sólo tiene especial sentido cuando se utilizan procedimientos recursivos.

Al igual que en el alojamiento estático los registros de activación contendrán el espacio correspondiente a los *parámetros formales* (variables que aparecen en la cabecera) y las *variables locales*, (las que se definen dentro del bloque o procedimiento) así como una zona para almacenar el *valor devuelto* por la función y una zona de *valores temporales* para el cálculo de expresiones

Para dos módulos o procedimientos diferentes, los registros de activación tendrán tamaños diferentes. Este tamaño por lo general es conocido en tiempo de compilación ya que se dispone de información suficiente sobre el tamaño de los objetos que lo componen. En ciertos casos esto no es así como por ejemplo ocurre en C cuando se utilizan arrays de dimensión indefinida. En estos casos el registro de activación debe incluir una zona de desbordamiento al final cuyo tamaño no se fija en tiempo de compilación sino solo cuando realmente llega a ejecutarse el procedimiento. Esto complica un poco la gestión de la memoria, por lo que algunos compiladores de bajo coste suprimen esta facilidad.

El procedimiento de gestión de la pila cuando un procedimiento q llama a otro procedimiento p, se desarrolla en dos fases, la primera de ellas corresponde al código que se incluye en el procedimiento q antes de transferir el control a p, y la segunda, al código que debe incluirse al principio de p para que se ejecute cuando reciba el control.

- El procedimiento autor de la llamada (q) evalúa las expresiones de la llamada, utilizando para ello su zona de variables temporales, y copia el resultado en la zona correspondiente a los parámetros formales del procedimiento que recibe la llamada.
- El autor de la llamada (q) coloca el puntero de control del procedimiento al que llama (p) de forma que apunte al final de la pila y transfiere el control al procedimiento al que llama (p)
- El receptor de la llamada (p) salva el estado de la máquina antes de comenzar su ejecución usando para ello la zona correspondiente de su registro de activación.
- El receptor de la llamada (p) inicializa sus variables y comienza su ejecución.

Al terminar la ejecución del procedimiento llamado (p) se desaloja su registro de activación procediendo también en dos fases. La primera se implementa mediante instrucciones al final del procedimiento que acaba de terminar su ejecución (p), y la segunda en el procedimiento que hizo la llamada tras recobrar el control:

- El procedimiento saliente (p) antes de finalizar su ejecución coloca el valor de retorno al principio de su registro de activación.
- Usando la información contenida en su registro el procedimiento que finaliza (p) restaura el estado de la maquina y coloca el puntero de final de pila en la posición en la que estaba originalmente.
- El procedimiento que realizó la llamada (q) copia el valor devuelto por el procedimiento al que llamó (p) dentro de su propio registro de activación (de q).

La fig. 6b. muestra el estado de la pila (durante el tiempo de ejecución) cuando se alcanza la instrucción señalada en el código de la fig. 6a:

```
función q(...// param. formales q //...)
{
      // var. locales de q
      .......
      x = p(23+y*z......);
}
función p(...// param. formales p //...)
{
      // var. locales de p
      ...........
         p(........);
      ...........
}
```

fig. 6a. Código fuente

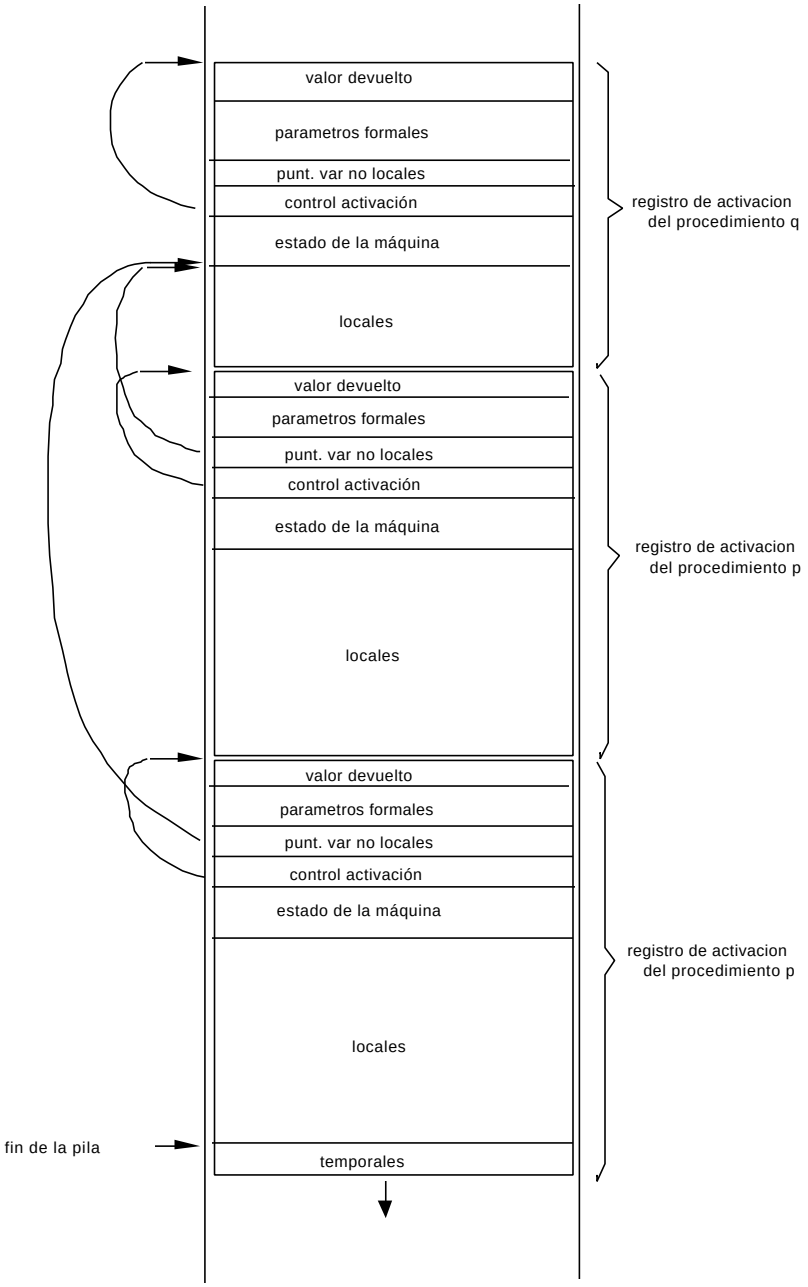


fig. 6b. Registros de activación en la pila durante la ejecución del programa

Dentro de un procedimiento, las variables locales se referencian siempre como direcciones relativas al comienzo del registro de activación, o bien al comienzo de la zona de variables locales. Por tanto, cuando sea necesario acceder desde un procedimiento a variables definidas en otros procedimientos cuyo ámbito sea accesible, será necesario proveer dinámicamente la dirección de comienzo de las variables de ese procedimiento. Para ello se utiliza dentro del registro de activación el *puntero de enlace a variables no locales* Este puntero, señalará al comienzo de las variables locales del procedimiento inmediatamente superior. El acceso a alguno de los antecedentes a éste, será posible gracias a que el registro superior guarda igualmente un puntero de enlace en una posición fija con respecto al comienzo de sus variables locales. Cuando los procedimientos se llaman a sí mismos recursivamente, el ámbito de las variables impide por lo general que una activación modifique las variables locales de otra activación del mismo procedimiento, por lo que en estos casos el procedimiento inmediato superior será, a efectos de enlace, el que originó la primera activación, tal como puede apreciarse en la figura 6b.

III.5. El montón

Cuando el tamaño de un objeto a colocar en memoria puede variar en tiempo de ejecución, no es posible su ubicación en memoria estática, ni tan siquiera en la pila. Son ejemplos de este tipo de objetos las listas, los árboles, las cadenas de caracteres de longitud variable, etc. Para manejar este tipo de objetos el compilador debe disponer de un área de memoria de tamaño variable y que no se vea afectada por la activación o desactivación de procedimientos. Este trozo de memoria se llama montón (traducción literal del termino ingles *heap* que se utiliza habitualmente en la literatura técnica). En aquellos lenguajes de alto nivel que requieran el uso del montón, el compilador debe incorporar al programa objeto el código correspondiente a la gestión del montón. Las operaciones básicas que se realizan sobre el montón son:

? Alojamiento: Se demanda un bloque contiguo para poder almacenar un objeto de un cierto tamaño.
? Desalojo: Se indica que ya no es necesario conservar un objeto, y que por tanto, la memoria que ocupa debe quedar libre para ser reutilizada en caso necesario por otros objetos.

Según sea el programador o el propio sistema el que las invoque, estas operaciones pueden ser explícitas o implícitas respectivamente. En caso de alojamiento explícito el programador incluye en el código fuente una instrucción que demanda una cierta cantidad de memoria para la ubicación de un dato (por ejemplo en PASCAL mediante la instrucción `new`, fig. 7a; en C mediante `malloc`, fig. 7b; etc.). La cantidad de memoria requerida puede ser calculada por el compilador en función del tipo correspondiente al objeto que se desea alojar, o bien puede ser especificardo directamente por el programador. El resultado de la función de alojamiento es por lo general un puntero a un trozo contiguo de memoria dentro del montón que puede usarse para almacenar el valor del objeto. Los lenguajes de programación imperativos utilizan por lo general alojamiento y desalojo explícitos. Por el contrario los lenguajes lógicos y funcionales evitan al programador la tarea de manejar directamente los punteros realizando las operaciones implícitamente.

```
var
    pt : ^real;
begin
    :
    new(pt);
    :
    free(pt);
    :
end;
```

fig. 7a. PASCAL

```
double *pt;
main()
{
    :
    pt = (double*) malloc(sizeof(double));
    :
    free(pt);
    :
}
```

fig.7b. C

La gestión del montón requiere técnicas adecuadas que optimicen el espacio que se ocupa o el tiempo de acceso, o bien ambos factores. A continuación se estudian algunas de estas técnicas.

III.5.1. Bloques de longitud fija

Existen diversas técnicas para la gestión del montón. El caso mas sencillo es cuando se puede garantizar que todas as demandas de memoria que se realicen serán de igual tamaño. En este caso el montón puede gestionarse como una lista encadenada de registros vacíos de longitud fija, tal como puede apreciarse en la figura 8, en los que se reserva un campo que contiene un puntero que mantiene los enlaces: Cuando el registro se ocupa los bytes del puntero no son necesarios y pueden utilizarse para almacenar el objeto.

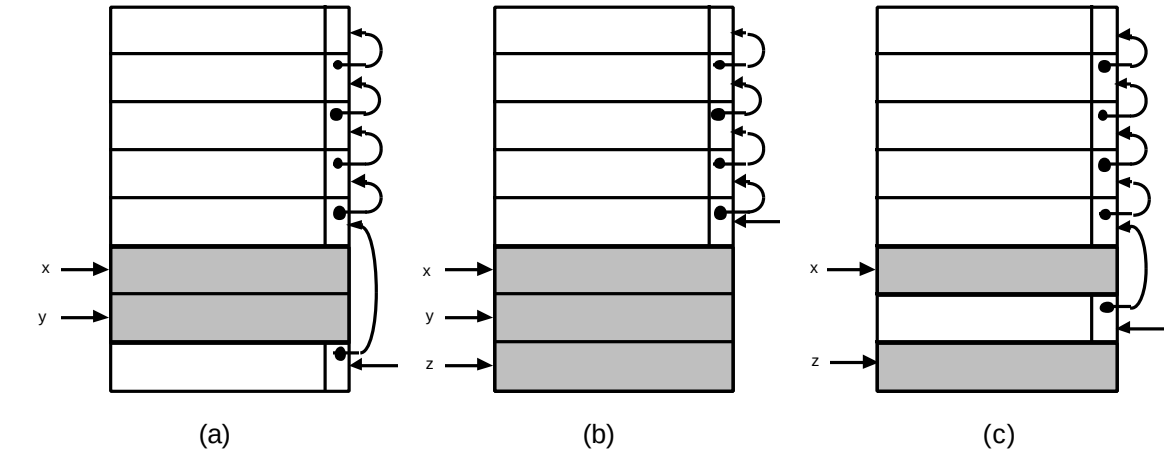


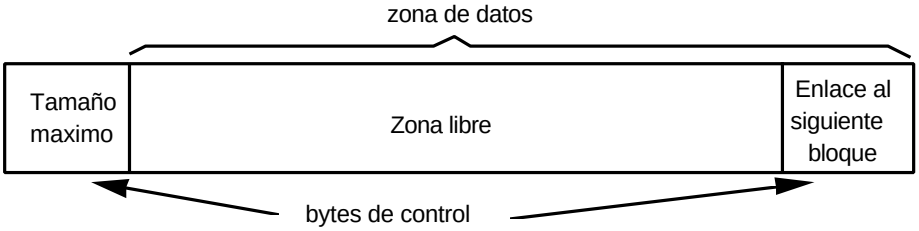
fig. 8. Procedimiento de alojo y desalojo para registros de longitud fija

En esta figura (a) aparece una representación esquemática de la configuración del montón en un instante determinado en el que se guardan los valores de las variables referenciadas mediante los punteros `x` e `y`. La figura (b), representa el alojamiento en el montón de una nueva variable apuntada por `z`. En la figura (c), aparece el montón tras desalojar de él el registro referenciado por el puntero.

Sin embargo, lo mas común es que los bloques que se demandan del montón no tengan todos ellos un mismo tamaño. En este caso la gestión se complica un poco existiendo varias técnicas que permiten realizar esta labor:

III.5.2. Montón con estructura simple.

La estructura mas sencilla que pueden tener los registros del montón consiste en reservar en ellos dos campos de control, uno para contener las referencias de enlace al siguiente registro libre del montón (al igual que en el caso anterior) y otro para guardar la longitud máxima del dato que puede colocarse en ese registro.



Para realizar las operaciones de alojo y desalojo en memoria existen varias estrategias, según el criterio que se siga a la hora de seleccionar el bloque libre que que se va a utilizar para la ubicación de un cierto objeto de tamaño N.

Estrategia del primer bloque

La estrategia del primer bloque consiste en recorrer la lista de bloques disponibles hasta encontrar uno de longitud mayor o igual que el tamaño del objeto a alojar (N). Si el bloque encontrado resulta ser de tamaño igual a N, se coloca en él el objeto y con ello termina la operación, si el primero que se encuentra es estrictamente mayor, se divide en dos trozos, uno de longitud N en el que se ubica el objeto, y otro del tamaño restante que se incluye en la lista de bloques libres. Evidentemente existe una mejora que debe incluirse en este método y es la comprobación de que el bloque restante tiene un tamaño suficiente para los campos de control. Una ligera modificación de este método consiste en considerar un umbral mínimo (superior al tamaño de los campos de control) para el tamaño de la parte restante, por debajo del cual no se realiza partición de bloque. Esto se hace para evitar una excesiva fragmentación del montón, con la consiguiente aparición de bloques pequeños de escasa operatividad.

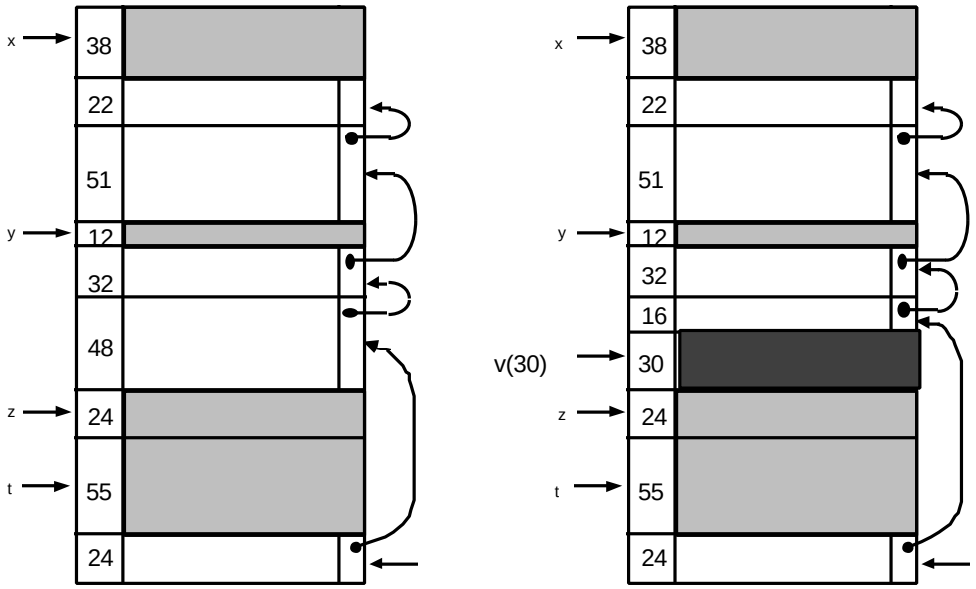


fig. 9. Alojamiento en el montón siguiendo la estrategia del primer bloque

Estrategia del mejor bloque

La siguiente estrategia consiste en recorrer la lista de bloques libres buscando aquel cuyo tamaño sea el más cercano a la cantidad de memoria requerida para la ubicación del objeto. Esta estrategia, que en principio pudiera suponerse mejor ha demostrado en la práctica no ser tan eficiente como la anterior. Téngase en cuenta, que para colocar un objeto, hay que recorrer toda la lista de bloques libres hasta encontrar el mejor mientras que con la primera estrategia basta con encontrar uno que sea mayor o igual. Además, la búsqueda del mejor bloque conlleva la creación de muchos bloques pequeños o restos inutilizables. Por todo ello salvo en casos muy excepcionales, esta estrategia se considera peor que la anterior.

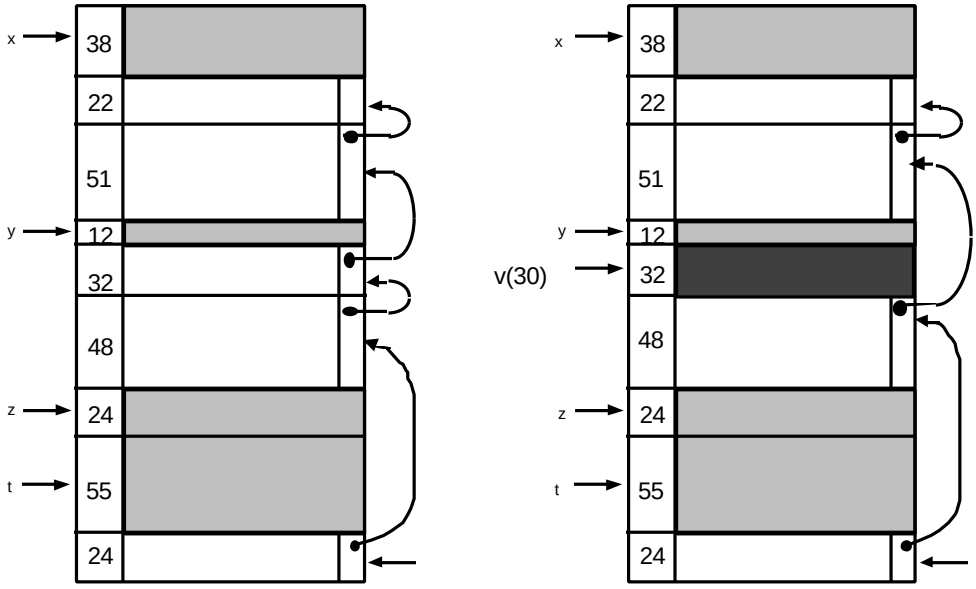


fig. 10. Alojamiento en el montón siguiendo la estrategia del mejor bloque

Estrategia del peor bloque

Esta estrategia consiste en buscar en la lista de bloques libres el de mayor tamaño, seleccionándolo siempre para la ubicación de cualquier objeto tras la división correspondiente. Esta estrategia tiene el inconveniente de la búsqueda, al igual que la estrategia del mejor bloque, pero sin embargo no genera bloques

pequeños. El inconveniente propio es que sistemáticamente acaba con los bloques grandes, utilizándolos a veces para ubicar objetos pequeños, y desaprovechando así espacios que pueden ser necesarioa para alojar objetos realmente grandes.

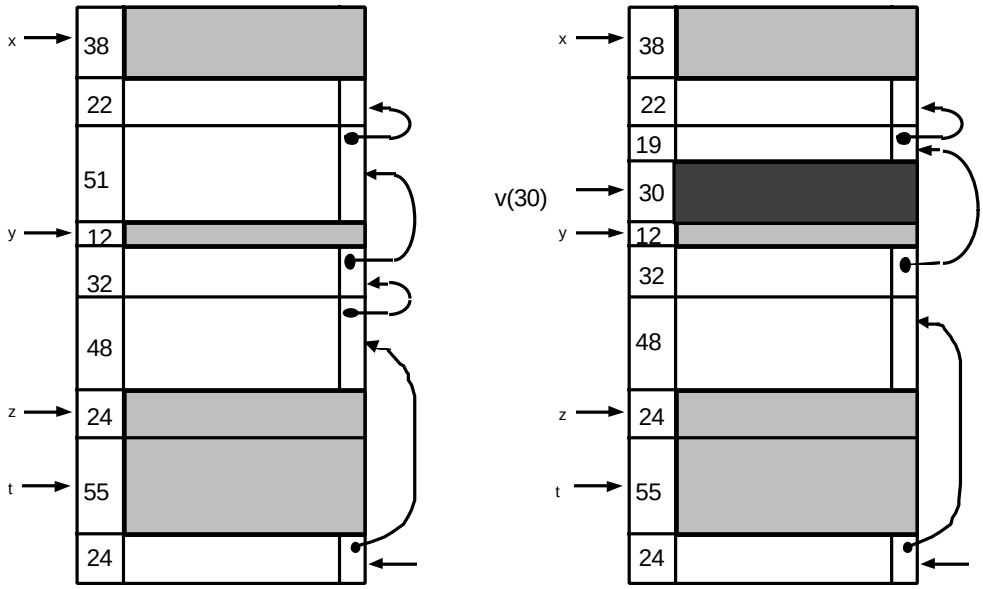


fig. 11. Alojamiento en el montón siguiendo la estrategia del peor bloque

Una posible mejora de estas estrategias de alojamiento consiste en mantener ordenada la lista de bloques libres. El ordenamiento puede hacerse en función del tamaño de los bloques, de mayor a menor o de menor a mayor, según se elija la estrategia del peor o del mejor bloque respectivamente. Con esta modificación la búsqueda resulta más rápida, especialmente en la estrategia del peor bloque.

Otro tipo de ordenación que puede realizarse es según las posiciones físicas de los bloques en memoria (en las figuras anteriores aparece así). Esto tiene como veremos tiene la gran ventaja de facilitar la unión de bloques contiguos que hayan quedado desalojados.

La operación de desalojo más simple consiste en insertar al comienzo de la lista de bloques libres el bloque recientemente liberado. Si esta lista está ordenada el proceso de desalojo es mas lento ya que debe buscarse la posición adecuada para la inserción en la lista.

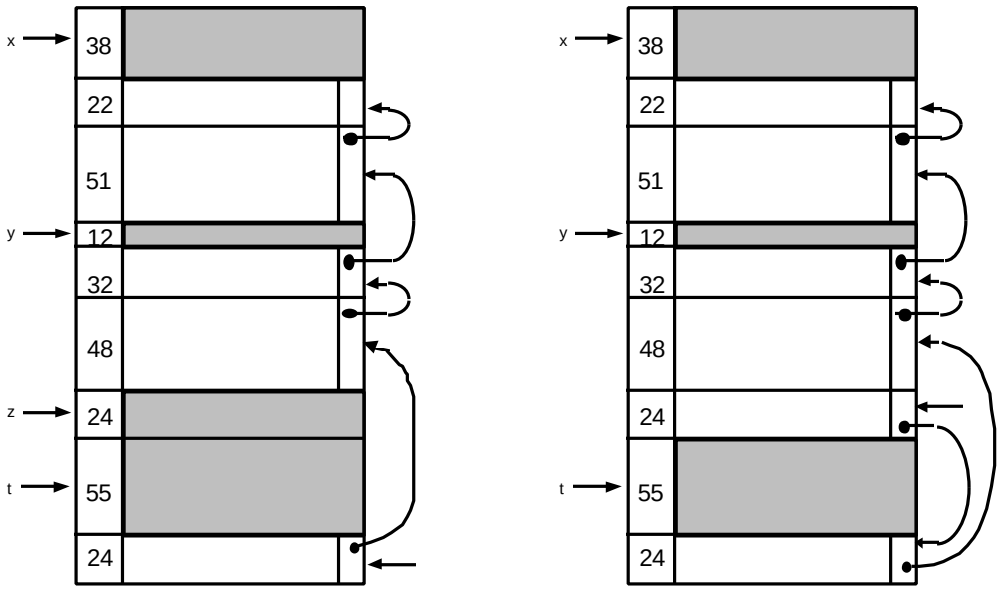
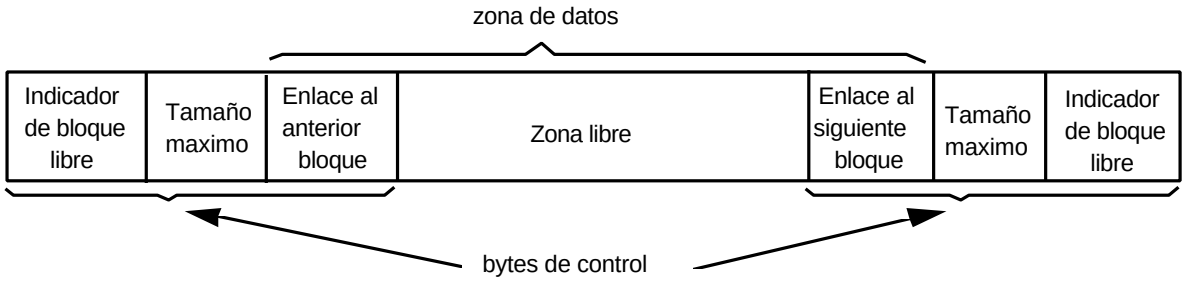


fig. 12. Desalojo de un bloque del montón con estructura simple

En esta técnica de bloques de estructura simple es difícil la reconstrucción de bloques grandes a partir de bloques pequeños que vayan quedando desalojados, ya que en general no se dispone de la información necesaria para saber si dos bloques son o no contiguos. Para realizar esta tarea es necesario mantener la lista de bloque libres ordenada en función de las posiciones de memoria que ocupan los bloques, o bien realizar compactaciones periódicas, por ejemplo, cuando no haya disponible un bloque del tamaño deseado. Sin embargo, existen otras técnicas de gestión del montón que solucionan este problema de forma más eficiente

III.5.4. Montón con estructura de bloques marcados (Boundary tag)

Para mantener siempre compactados en un solo bloque los bloques libres contiguos puede usarse una técnica basada en la siguiente estructura para los bloques:



Los bloques tienen una estructura simétrica, a ambos lados existen tres campos reservados, dos de ellos son los ya utilizados en la estructura simple que se ha visto anteriormente; uno para indicar el tamaño del mayor objeto que puede contener el bloque y otro para mantener los enlaces de la lista de bloques libres. Es necesario mantener una lista doblemente enlazada, utilizando para ello los punteros de la parte izquierda y derecha. Además, debe utilizarse un campo situado en los extremos del bloque para indicar si esta libre o no.

Las operaciones de alojamiento en memoria de nuevos objetos son las mismas que en caso anterior, salvo por el mantenimiento de esta nueva estructura de lista doblemente enlazada. Son por tanto aplicables las estrategias del primer bloque, del mejor bloque o del peor bloque.

En el caso de la operación de desalojo de un bloque las operaciones a realizar son las siguientes:

1. Comprobar si el bloque anterior está libre, para o cual hay que consultar la posición contigua al bloque que se desaloja (posición -1 con referencia al comienzo del bloque que se libera).Si no está libre ir al paso 3.
2. En caso de que esté libre, repetir el proceso inspeccionando si el bloque anterior del anterior está también libre. Esto es posible ya que se puede calcular el número de bytes que hay que retroceder para inspeccionar el indicador de bloque libre del registro anterior al anterior, puesto que la longitud del bloque anterior esta disponible.(en la posición -2)
3. Realizar el proceso descrito anteriormente (pasos 1 y 2) pero con los enlaces hacia delante
4. Una vez hallada la zona de memoria contigua no ocupada por ningún objeto, reestructurarla como un único registro.
5. Incluir el bloque hallado anteriormente en la lista doblemente enlazada de bloques libres.

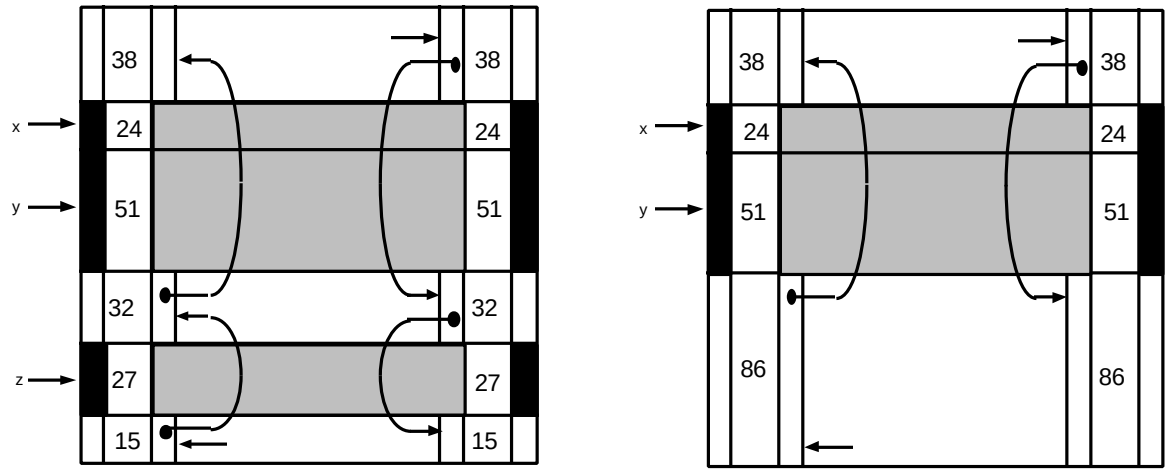
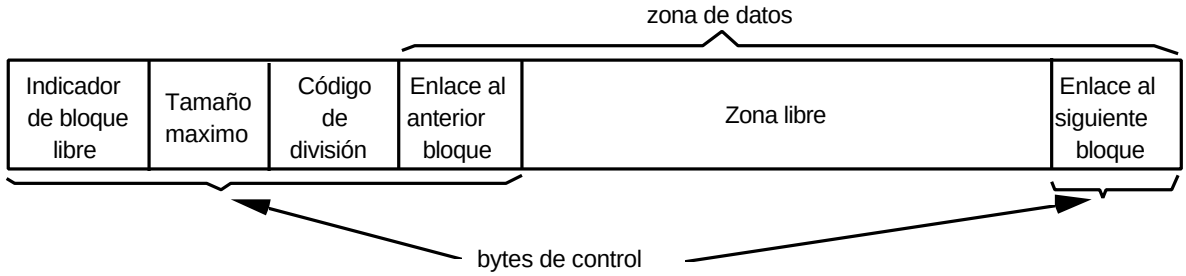


fig. 13. Desalojo de un bloque del montón con estructura de bloque marcados

En el ejemplo de la figura vemos el resultado de la operación de desalojo del bloque ocupado por z. El cálculo del tamaño del nuevo bloque libre se ha realizado contando con que la longitud del campo indicador de bloque libre es 1, y el que contiene el tamaño de longitud 2.

III.5.5. Montón con estructura de división fija o de bloques compañeros (buddy-system)

La característica fundamental de este método es que sólo permite la existencia de bloques cuya longitud corresponda a un número de una serie finita de longitudes, por ejemplo, sólo bloques de tamaños 4,8,16,28,48,80,132,.... Caracteriza también a este método la existencia de múltiples listas doblemente enlazadas de bloques libres, una para cada uno de los tamaños disponibles. La estructura de los bloques que se utiliza contiene los campos que se indican enel siguiente esquema:



La estrategia para la asignación en memoria de un objeto de tamaño N es la siguiente:

1. Se halla el número de la serie de tamaños inmediatamente superior o igual a N, es decir si la serie es $F_0,F_1...F_{max}$ debe localizarse el valor F_n , seindo 0 ? ?n ? ?max, tal que:

   $$F_{n-1} < N ? F_n$$

2. Se busca en la lista correspondiente a los bloques libres de longitud F_n, si hay alguno disponible. En caso afirmativo se aloja en él el objeto, si no, se busca en la lista de tamaño inmediatamente superior (es decir F_{n+1}), y así sucesivamente hasta encontrar un bloque en el que quepa el objeto. Sea F_m el tamaño del bloque que se obtiene tras este proceso.

3. Si $F_m > F_n$, se divide el registro correspondiente de tamaÒo F_m, de forma que se obtengan dos nuevos registros cuyos tamaños serán R_p y R_q respectivamente. La serie de números base para este método debe elegirse de tal forma que los registros que se obtengan de la división tengan tamaños correspondientes a números de la serie, es decir $R_p = F_{m-1}$ y $R_q = F_{m-k}$. Es más, el tamaño de uno de los nuevos registros

que se obtengan al realizar la división debe ser el correspondiente al valor inmediato anterior de la serie. Esta condición impone la siguiente restricción :

$F_m = F_{m-1} + F_{m-k} + ?$

en donde ?? corresponde a la porción de memoria utilizada por los bytes de control. (originalmente sólo hay un registro y finalmente hay dos, por lo que del registro original debe salir también el espacio correspondiente a un grupo de bytes de control)

4. Una vez dividido el registro, y colocados los nuevos registros que aparecen tras la división en las listas de bloques libres correspondientes se repiten los pasos anteriores (pasos 2 y 3), hasta encontrar un registro de tamaño F_n inmediatamente igual o superior a N.

Cada vez que se divide un bloque (en el paso 3) se obtienen dos bloques a los que se denomina bloque izquierdo y bloque derecho. Ambos bloques están contiguos en memoria. Además conociendo el tamaño del bloque izquierdo se puede hallar automáticamente el tamaño de su bloque derecho compañero y viceversa.

Para registrar el número de divisiones y la calidad de bloque izquierdo o derecho, se incluye en la estructura de los bloques un campo al que se ha denominado en la figura *control de división*. Cada vez que se divide un registro se incrementa en una unidad el *control de división* del bloque izquierdo resultante, y se coloca un 0 en el *control de división* del bloque derecho, así, con un mismo campo se tiene toda la información necesaria. Asi, por ejemplo, si dividimos un bloque derecho (cuyo *control de división* debe ser 0) los *controles de división* de los nuevos bloques izquierdo y derecho serán respectivamente 1 y 0. Si se divide un bloque cuyo *control de división* es 3 (se trata de un bloque izquierdo) se obtienen otros dos nuevos bloques, el primero de los cuales es un bloque izquierdo cuyo *control de división* es 4, y el segundo es un bloque derecho cuyo *control de división* es 0.

Para realizar la operación de desalojo de un bloque se procede según el siguiente algoritmo:

1. Se inspecciona el campo de *control de división* del bloque a desalojar, para determinar si se trata de un bloque izquierdo o derecho, y se obtiene su tamaño consultando el campo adyacente .
2. Si se trata de un bloque izquierdo, se comprueba si a continuación de él, en la posición correspondiente hay un bloque libre. Dado que se sabe que es un bloque izquierdo y que se conoce su longitud, puede determinarse fácilmente cual es la posición de memoria del byte correspondiente al *indicador de bloque libre* del bloque compañero. Si el bloque contiguo está libre, se comprueba adamás que su longitud corresponde a la del bloque compañero del bloque que se intenta desalojar. En caso afirmativo se funden ambos bloques en uno solo (correspondiente al bloque que en su momento dio origen a los dos), actualizando convenientemente las listas de bloques libres. El *control de división* del bloque resultante será una unidad inferior a la del bloque izquierdo que intentamos desalojar. Una vez agrupados ambos bloque se repite el proceso de desalojo descrito en este punto para el bloque resultante, cuyo *control de división* indicará si se trata de un bloque izquierdo o derecho
3. Si el bloque a desalojar es un bloque derecho, se realiza la operación simétrica La información sobre el tamaño del registro compañero, que en este caso será necesaria para inspeccionar los registros de control del bloque adyacente, puede obtenerse mediante la aplicación de la fórmula de la serie, a partir del tamaño del propio bloque a desalojar.

Las series de números que se utilizan para este método pueden ser:

- series de Fibonacci: (según la formula $F_n = F_{n-1} + F_{n-2} + ??$

- series de división binaria: (según la formula $F_n = F_{n-1} + F_{n-1} + ??$

por ejemplo, Si se utiliza 1 byte para el *indicador de bloque libre*, 2 bytes para indicar el *tamaño máximo* del objeto que puede almacenar, y 1 byte para el *control de división*, en total se tiene que ? = 4

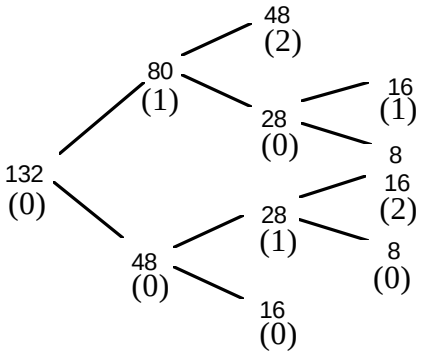
Fijados los primeros elementos de la serie, el resto de tamaños puede obtenerse mediante la fórmula correspondiente, por ejemplo, si $F_1 = 4$ y $F_2 = 8$, se obtendrá la serie de Fibonacci:

4,8,16,28,48,80,132,216.. Si se emplea la fórmula de división binaria a partir de un tamaño inicial $F_1 = 4$, se obtendrá la serie: 4,12,28,60,124,252 ....

Ejemplo:

Sea un área de memoria de tamaño 132, sobre la que se aplica la técnica de subdivisión fija basada en la serie finita 4,8,16,28,48,80,132. En la figura inferior se representa el estado del montón al alojar en él objetos de tamaño 12, figura (a), de tamaño 35, figura (b), y finalmente dos objetos de tamaño 12, figuras (c) y (d).

A la derecha aparece el grafo que representa las divisiones que se han llevado a cabo en la configuración final y entre paréntesis los códigos de división correspondientes a cada bloque.



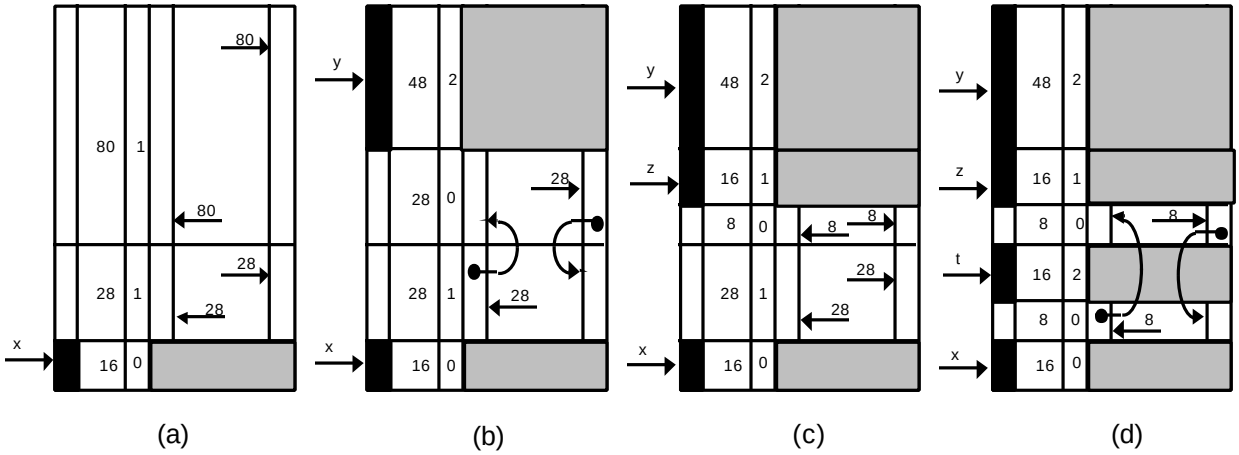


fig. 14. Alojamiento en el montón según la técnica de subdivisión fija

III.5.6. Recolección de basura (Garbage collection)

Hasta el momento se ha estudiado el problema de la asignación y desalojo de memoria sin tener en cuenta el momento en el que este último se lleva a cabo. En los lenguajes de programación imperativos, el desalojo suele ser explícito, y por tanto se realiza en el momento elegido por el programador. Es por tanto responsabilidad exclusiva del programador la recuperación de la memoria asignada a objetos que dejan de estar en uso y el evitar que se produzcan referencias colgadas, es decir referencias a posiciones de memoria que se supoone que están libres. Este es uno de los problemas que hace que la programación en estos lenguajes sea tediosa y es la causa de gran parte del tiempo empleado en la depuración de los mismos.

```
:
new(p);
p ^ := 27;
new(p);
p ^ := 45;
:
```

Por ejemplo, el segmento de código PASCAL que aparece a la izquierda, reclama primeramente memória del montón para almacenar el valor 27, referenciándolo mediante el puntero p. Al realizar de nuevo la petición de memoria el puntero p señalará a otro bloque del montón quedando el bloque que contenía el valor 27 sin referencia. Una vez perdida la referencia a este bloque, en general no es posible en los lenguajes imperativos reclamar esta memoria que permanecerá ocupando espacio hasta el final de la ejecución del programa.

En lengujes lógicos y funcionales, tales como LISP, PROLOG o SMALLTALK, se libera al programador de la tarea de controlar explícitamente el alojo y desalojo de objetos en el montón. Como contrapartida, el compilador o interprete de este tipo de lenguajes debe incorporar las rutinas necesarias para

detectar qué objetos hay que desalojar y cuando hay que realizar esta operación: En general un objeto debe desalojarse del montón si se detecte que no está en uso por el programa (que no está referenciado). A este proceso se le denomina recolección de basura.

Las técnicas tradicionales de recolección de basura son la desocupación sobre la marcha (*free-as-you-go*) y la técnica de señalar y barrer (*mark-and-sweep*).

III.5.6.1. Desocupación sobre la marcha (free-as-you-go)

El método de desocupación sobre la marcha consiste en desalojar (implícitamente) los bloques del montón tan pronto como se detecta que no están referenciados por ningún puntero. Para ello es necesario que el sistema lleve internamente un contador del número de punteros que se dirigen hacia cada uno de los bloques del montón. Este contador será parte integrante de los bytes de control del bloque y será actualizado cada vez que se cree o modifique un puntero dirigido al montón. El compilador debe incluir el código correspondiente a estas operaciones intercalándolo en el programa objeto.

Así, por ejemplo, la simple instrucción `p:=q;` siendo `p` y `q` punteros al montón requiere las siguientes operaciones previas a la copia de direcciones:

1. Acceder al bloque que señala `p` y su contador.
2. Comprobar que el número de referencias al bloque señalado anteriormente por `p` no es cero, y si lo fuera desalojar dicho bloque.
3. Acceder al bloque señalado por `q` e incrementar su contador.

Los inconvenientes de este método son la gran cantidad de operaciones necesarias cada vez que se accede al montón y su ineficacia en el caso de estructuras dinámicas circulares (por ejemplo, listas en las que el último elemento vuelve a apuntar al primero), ya que en ellas la estructura considerada como un todo puede estar sin referencia desde el programa, y sin embargo todos los contadores de sus bloques tendrán al menos una referencia.

III.5.6.2. Método de Marcar y Barrer (mark and sweep)

Este método consiste en activar un procedimiento especial cuando se detectan ciertas condiciones en el uso de la memoria (por ejemplo, cuando queda poca memoria libre, o cuando finaliza un módulo, etc..) Este procedimiento se denomina *marcar y barrer* y tiene dos partes:

1. Buscar todas las referencias que existan hacia el montón, y marcar los bloques accesibles.
2. Barrer (colocar en la lista de bloques libres) todos los bloques no marcados en la fase anterior.

Este método es clásico en lenguajes como LISP, y es frecuente que muchos textos al hablar de *recolección de basura* se hagan referencia exclusivamente al método de marcar y barrer, ya que es el más frecuentemente empleado.

Es asunto de discusión el momento en el que debe activarse el procedimiento de marcar y barrer En general se activa cuando se detectan determinadas condiciones en el estado de la memoria (si queda libre menos de un cierto umbral) o bien, tras determinadas operaciones de la ejecución del programa (salida de un procedimiento, etc.). Por ejemplo, si el disparo del procedimiento de marcar y barrer se efectúa cuando queda muy poca memoria libre en el montón, el inconveniente es que probablemente haya un mayor número de objetos alojados en el montón y consiguientemente la fase de marcado sea más costosa computacionalmente. Si por el contrario se hacen llamadas al procedimiento cuando queda suficiente memoria libre puede que tras la ejecución del procedimiento no se genere una cantidad de memoria libre apreciable y se requieran pronto nuevas llamadas.

De todas formas, el mayor inconveniente de esta técnica se presenta en los sistemas de tipo interprete que funcionan en tiempo real, para los cuales la activación del procedimiento recolector de basura

supone una pausa en la ejecución del programa, tediosa e inexplicable desde el punto de vista del usuario (programador).

Desde el punto de vista de la implementación, el procedimiento de marcar y barrer es una función autónoma que el compilador incluye en el programa objeto junto con el resto de funciones y procedimientos traducidos del programa fuente.

III.5.7. Compactación

La compactación consiste en mover físicamente los bloques del montón a fin de crear grandes espacios contiguos de memoria. En general no se incluyen técnicas de compactación ya que son tremendamente costosas en tiempo de ejecución, al tener que revisar nuevamente todas las referencias que existan en el programa y reubicar gran cantidad de objetos.

A veces puede optarse por incluir un mecanismo de compactación junto con la técnica de marcar y barrer, ya que en ambos casos es necesario computar las referencias activas al montón. Sin embargo, éste no es el caso más habitual, y no debe confundirse la recolección de basura, relacionada con el desalojo implícito, con la compactación, que implica el movimiento de los objetos activos de unas posiciones de memoria a otras.

III.6. Referencias colgadas

Se llaman *referencias colgadas* o *suspendidas* a los punteros que hacen referencia a posiciones de memoria que el sistema considera que están libres. Este es un problema clásico de los lenguajes de programación imperativos. Los lenguajes lógicos y funcionales no presentan estos problemas ya que el programador no gestiona directamente los punteros.

Pueden producirse referencias colgadas en cualquier zona de memoria, estática, pila o montón, aunque las situaciones más frecuentes se producen al trabajar con la memória dinámica (pila y montón)

Por ejemplo, debido al método de almacenamiento de las variables locales de los bloques del programa en registros de activación, que se alojan y desalojan de una pila automáticamente, ha de tenerse especial cuidado cuando se emplean dentro de un bloque de programa referencias directas a memoria (punteros) puesto que pueden producirse referencias a posiciones de memoria que ya hayan sido desalojadas de la pila, (consideradas como libres). Sirva el siguiente ejemplo en C para ilustrar este problema.

```
int main ()
{
   int *p,*q;
   p = kk();
   q = gg();
}
int *kk()
{
   int i=27;
   return &i
}
int *gg()
{
   int j=45;
   return &j
}
```

La variable `p` toma la dirección de la variable local `i`, de la función `kk()` pero tan pronto como termina la ejecución de esta función, la porción de memoria que ocupaba la variable local `i` queda libre a disposición de cualquier otro registro que entre en la pila, por lo que el contenido del mismo es incierto desde ese momento. De hecho, si a continuación se realiza una llamada a la función `gg()` el espacio utilizado por la nueva variable local `j` es el mismo al que apunta `p` y por tanto el valor de `*p` cambiará súbita e inexplicablemente.

Otro ejemplo de referencias colgadas, esta vez en el montón, se produce al ejecutar el siguiente trozo de código C. La instrucción `p=(int*)malloc(sizeof(int));` asigna a `p` una dirección de memoria en la que hay capacidad suficiente para alojar un objeto de tipo entero. En este bloque del montón se guarda el valor `27`. La siguiente instrucción copia el valor de `p` (la dirección de memoria del montón) en `q`. El problema sobreviene al desalojar el bloque apuntado por `q`,

```
int *p, *q;

main()
{
   :
   p=(int*)malloc(sizeof(int));
   *p = 27;
   q = p;
   free(q);
   :
}
```

ya que si a partir de este momento se intenta acceder al valor `*p`, el resultado es impredecible.

III.7. Padding

A veces según el tipo de instrucciones de la máquina es necesario redondear el número de bytes requeridos para el alojamiento de un determinado objeto hasta un valor múltiplo de 2,4,8,16, etc. Esto se debe a que ciertas instrucciones de la máquina esperan encontrar sus datos comenzando siempre en una determinada posición de memoria múltiplo de uno de estos valores.

La asignación de memoria debe tener en cuenta estos requisitos, lo que conlleva en algunos casos un desperdicio de memoria al que se llama *padding* Para optimizar la utilización de memoria en maquinas con esta característica algunos lenguajes incluyen la posibilidad de especificar que los datos se empaqueten para su almacenamiento, utilizando un procedimiento propio para el desempaquetado cada vez que se requiera hacer uso de ellos. Este es el caso de la declaración `packed` usada en PASCAL