ANÁLISIS COMPARATIVO DE LENGUAJES

Guía de estudio correspondiente a la Teoría sobre: Administración de la Memoria

Depto. de Informática

Universidad Nacional de San Luis - Año 2022

ADMINISTRACIÓN DE LA MEMORIA

Notas de Clase

Programming Languages – Design and Implementation – Terrence Pratt:

Capítulo 6 – Secciones 6.3.1 y 6.3.2 (Págs 238-245)

Capítulo 9 – Subsección Stack-Based Implementation (Págs 351-353)

Capítulo 10 - Págs 415-432

ADMINISTRACIÓN DE LA MEMORIA

Los subprogramas son la estructura básica a partir de la cual se construyen los programas. Es interesante estudiarlos desde el punto de vista de:

Diseño de programas.

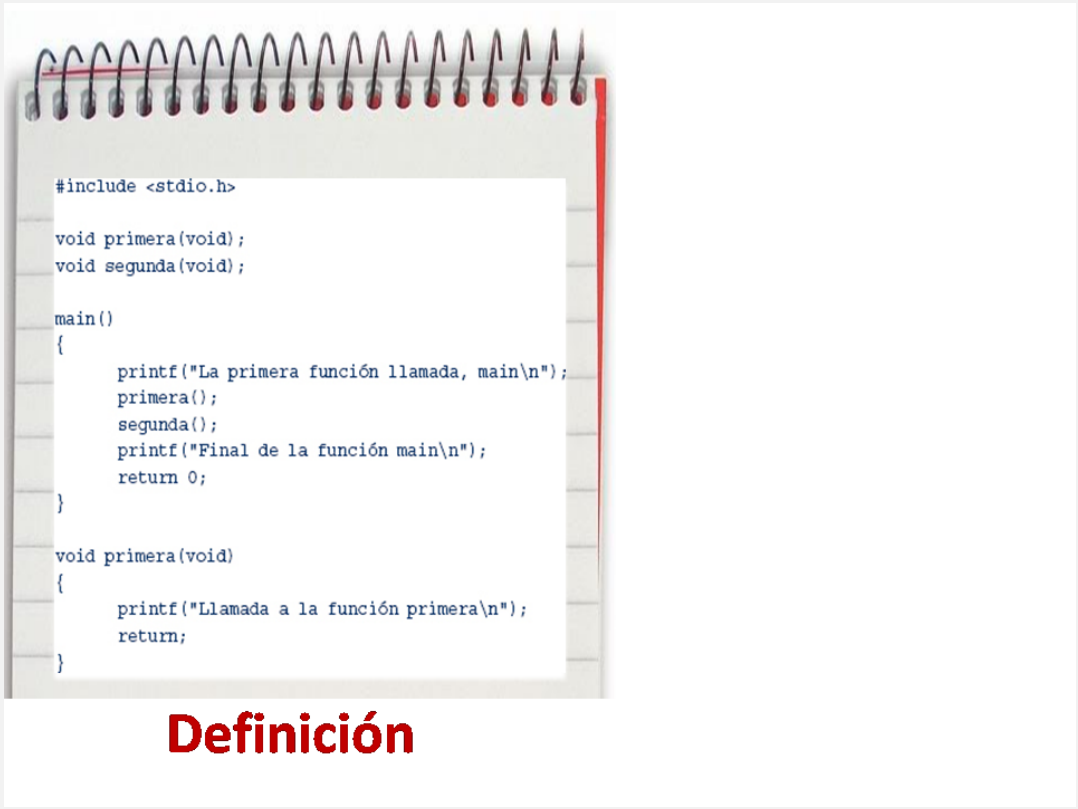
Diseño de lenguajes.

Para entender cabalmente, todos los elementos que requieren memoria en tiempo de ejecución, es importante analizar primero cual es la "imagen" de un programa fuente en ejecución, es decir, diferenciar entre la definición y activación de subprogramas.

DEFINICIÓN Y ACTIVACIÓN DE SUBPROGRAMAS

```
#include <stdio.h>

void primera(void);
void segunda(void);

main()
{
      printf("La primera función llamada, main\n");
      primera();
      segunda();
      printf("Final de la función main\n");
      return 0;
}

void primera(void)
{
      printf("Llamada a la función primera\n");
      return;
}
```

DEFINICIÓN Y ACTIVACIÓN DE SUBPROGRAMAS



```c
#include <stdio.h>

void primera(void);
void segunda(void);

main()
{
      printf("La primera función llamada, main\n");
      primera();
      segunda();
      printf("Final de la función main\n");
      return 0;
}

void primera(void)
{
      printf("Llamada a la función primera\n");
      return;
}
```

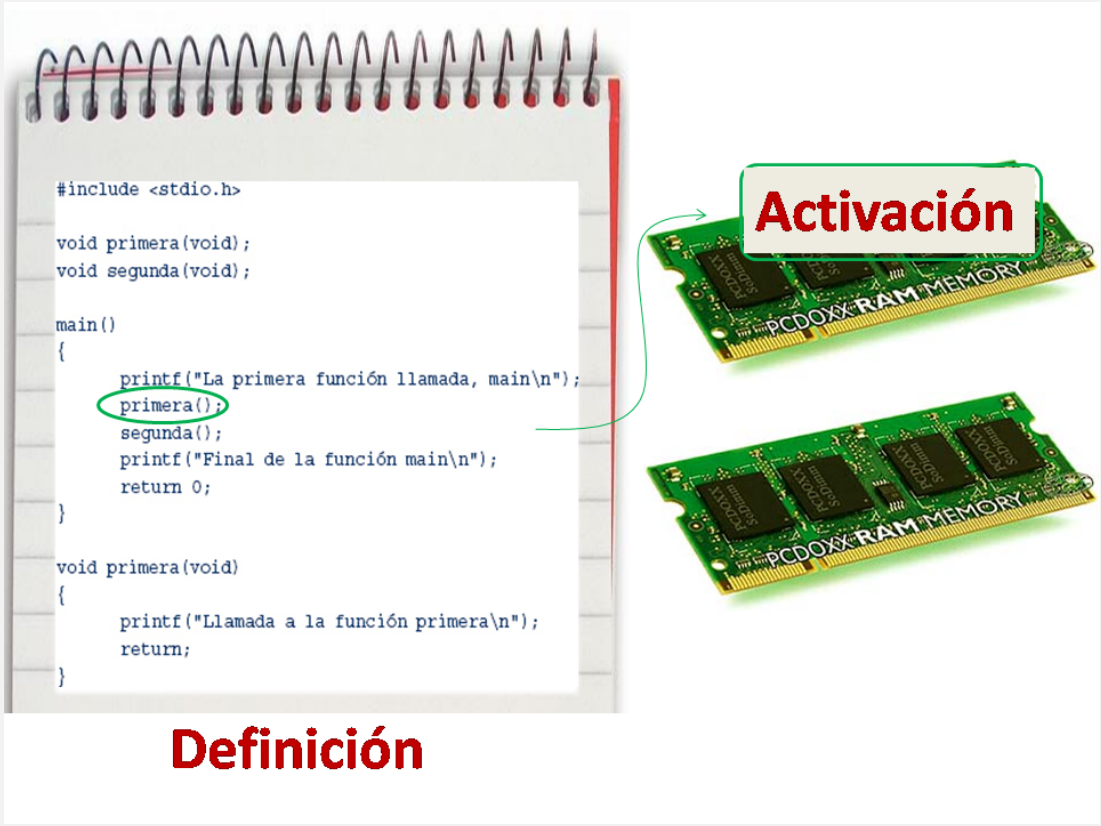
Definición

DEFINICIÓN Y ACTIVACIÓN DE SUBPROGRAMAS

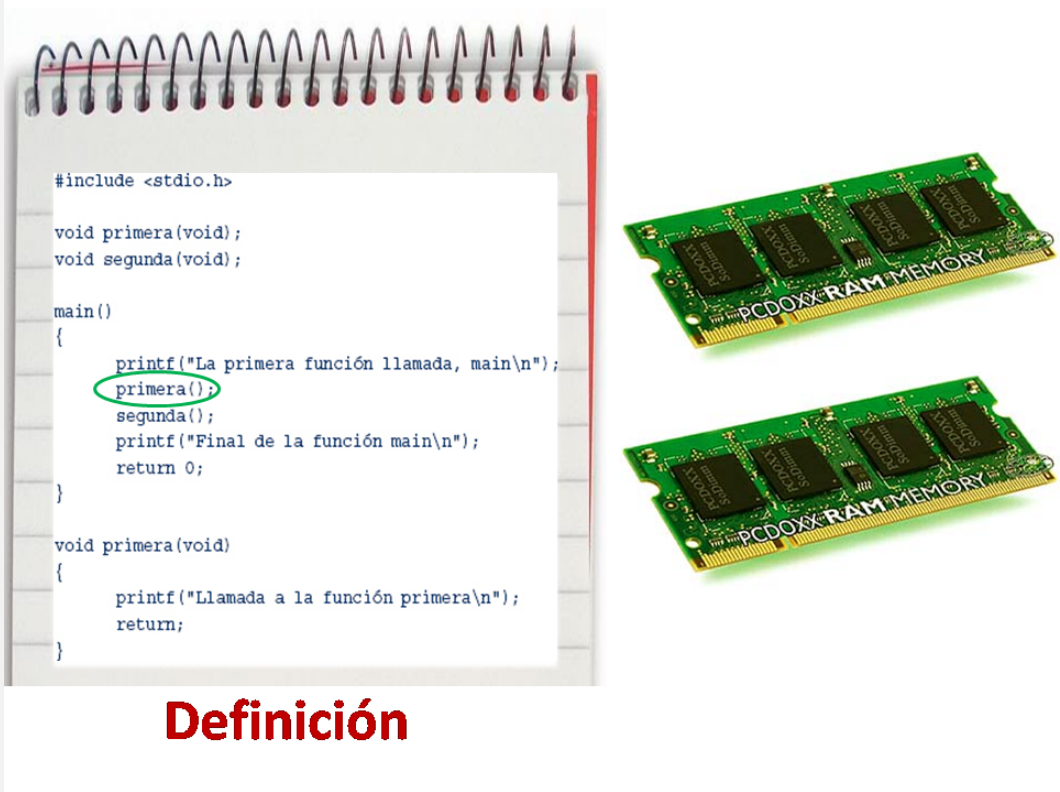
```c
#include <stdio.h>

void primera(void);
void segunda(void);

main()
{
      printf("La primera función llamada, main\n");
      primera();
      segunda();
      printf("Final de la función main\n");
      return 0;
}

void primera(void)
{
      printf("Llamada a la función primera\n");
      return;
}
```

Definición

DEFINICIÓN Y ACTIVACIÓN DE SUBPROGRAMAS



DEFINICIÓN Y ACTIVACIÓN DE SUBPROGRAMAS



Cuando se retorna luego de la ejecución del subprograma `primera()`

DEFINICIÓN Y ACTIVACIÓN DE SUBPROGRAMAS

Definición de un subprograma: la realiza el programador y es una propiedad estática de un programa.

Activación de un subprograma: se crea cada vez que el subprograma es invocado durante la ejecución del programa.

Diferencia entre definición y activación de subprograma = Diferencia entre tipo de dato y OD.

Definición de un subprograma permite al compilador armar el template (plantilla) que será usado para construir cada activación.

DEFINICIÓN Y ACTIVACIÓN DE SUBPROGRAMAS

Segmento de código (estático): código ejecutable, constantes, elementos que no varían, instrucciones de creación y destrucción del registro de activación.

Registro de activación (dinámico): parámetros, variables locales, punteros a otros registros de activación.

DEFINICIÓN Y ACTIVACIÓN DE SUBPROGRAMAS

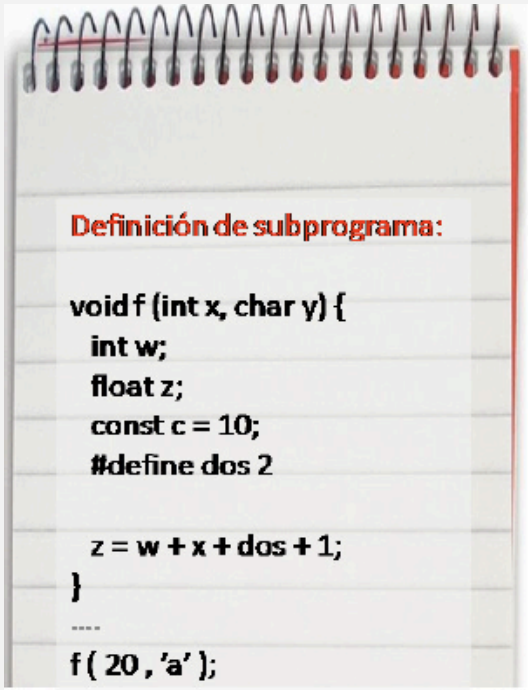
Definición de subprograma:

```
void f (int x, char y) {
  int w;
  float z;
  const c = 10;
  #define dos 2

  z = w + x + dos + 1;
}
....
f ( 20 , 'a' );
```

Para la invocación genera:

```
push (20)
push ('a')
call f
.....
```

DEFINICIÓN Y ACTIVACIÓN DE SUBPROGRAMAS



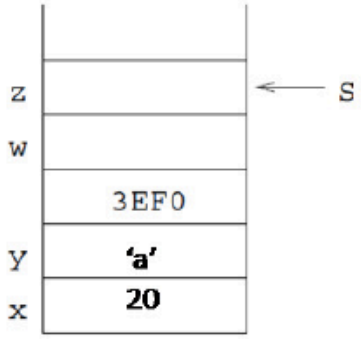
Definición de subprograma:

```
void f (int x, char y) {
  int w;
  float z;
  const c = 10;
  #define dos 2

  z = w + x + dos + 1;
}
----
f ( 20 , 'a' );
```

Segmento de Código

Prólogo
f: add SP #2
lda SP #-4
adda SP #-3
sta SP
....
sub SP #2
ret
Epílogo
1
10
2

Registro de Activación



Para la invocación genera:

```
push (20)
push ('a')
call f
.....
```

PLANTILLA DE ACTIVACIÓN

Es usada para construir cada **activación**. Consta de:

- El segmento de código (estático). Uno solo compartido por todas las activaciones.
- El registro de activación (dinámico). Se crea uno por cada activación. (¿Por qué?).

SEGMENTO DE CÓDIGO

Cuando se produce la invocación a un subprograma, tanto como cuando se retorna después de la ejecución, toman lugar una serie de acciones transparentes al programador, a saber:

Prólogo: seteo del registro de activación, la transmisión de parámetros, creación de los links para referencias no locales, y toda actividad previa a la ejecución que sea necesaria.

Epílogo: es el conjunto de instrucciones insertadas por el traductor al final del bloque de código ejecutable, para llevar a cabo las acciones antes mencionadas.

GESTIÓN DE ALMACENAMIENTO DE LOS DATOS

Depende de las características (o restricciones) de los lenguajes. es un aspecto importante a considerar desde el punto de vista del diseñador hasta el del programador del lenguaje.

- **FORTRAN:** administración estática.
- **PASCAL:** almacenamiento basado en pilas.
- **LISP, JAVA:** administración dinámica basada en *heap* con Recolección de Basura.

ELEMENTOS QUE REQUIEREN MEMORIA

- Segmento de código.
- Programas del sistema en tiempo de ejecución.
- Estructuras de datos definidas por el usuario y constantes.
- Puntos de retorno de subprogramas.
- Ambientes de referenciación.
- Almacenamiento temporario para evaluación de expresiones.
- Almacenamiento temporario en la transmisión de parámetros.
- Buffers de entrada-salida.
- Datos del sistema.

OPERACIONES QUE REQUIEREN (LIBERAN) MEMORIA

- LLamadas a subprogramas (y operaciones de retorno).
- Operaciones de creación (y destrucción) de estructuras de datos.
- Operaciones de inserción (y borrado) de componentes.

CONTROL DE ALMACENAMIENTO

¿Quién debe tener el control del almacenamiento (memoria), el programador o el sistema?

- El **programador**: puede ser una tarea muy difícil (Reg. de activación, almacenamiento temporal, etc.)… Puede interferir con la gestión controlada por el sistema. Una versión intermedia es el control de las estructuras dinámicas (*malloc, new*) pero puede generar problemas…
- El **sistema**: menos eficaz en la liberación del espacio que no se utiliza pero más seguro. Se restringe la posibilidad de manejo dinámico de estructuras…

FASES DE LA GESTIÓN DE ALMACENAMIENTO

Asignación inicial: disposición inicial del almacenamiento a administrar. Incluye los mecanismos que permiten determinar si un bloque de memoria está libre o no y asignar memoria a los procesos que lo requieran.

Recuperación: proceso por el cual el administrador de memoria recupera almacenamiento que ya ha sido asignado, usado y que posteriormente queda disponible.

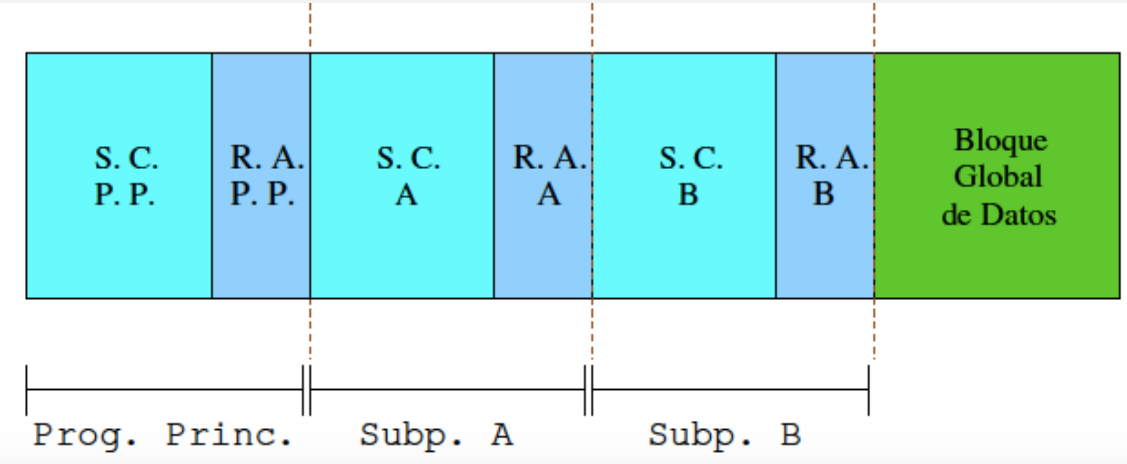
Compactación y nuevo uso: la compactación es el proceso de construir bloques grandes de almacenamiento libre, a partir de fragmentos pequeños. El nuevo uso se basa normalmente en los mismos mecanismos utilizados en la asignación inicial.

MECANISMOS DE ADMINISTRACIÓN DE MEMORIA

- Estática
- Dinámica
  - Basada en Pilas
  - Heap (montículos o parvas)

ADMINISTRACIÓN DE LA MEMORIA ESTÁTICA

- La forma de administración más simple (FORTRAN, COBOL).
- El espacio a utilizar se determina en traducción y permanece fijo durante la ejecución.
- Eficiente en tiempo y espacio durante ejecución (no hay administración dinámica).
- Incompatible con subprogramas recursivos y asignación (o liberación) de memoria en puntos arbitrarios.



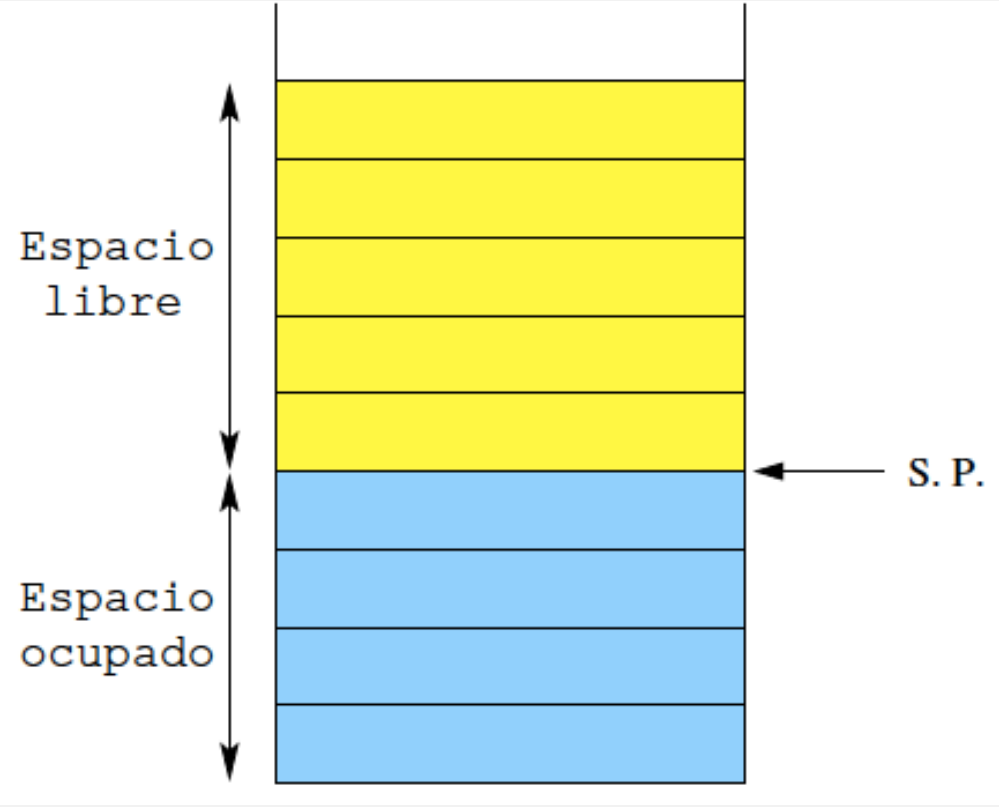
ADMINISTRACIÓN DE MEMORIA BASADA EN PILAS

Es la técnica de administración **dinámica** de memoria más simple.

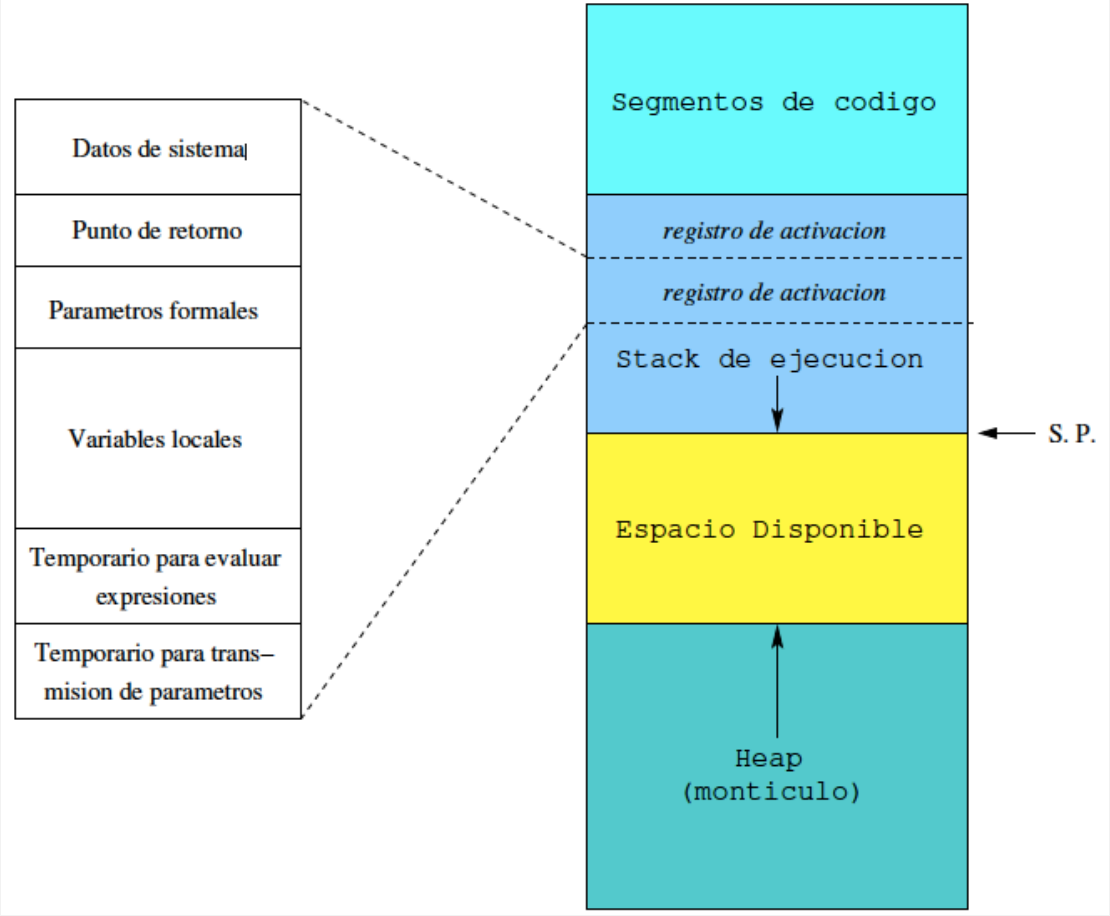
Esquema muy utilizado para el almacenamiento de registros de activación.

- **Asignación inicial**:
  - * Todo el espacio libre es tratado como un bloque secuencial en memoria.
  - * Todo lo que requiere para mantener los bloques libres/ocupados es un **puntero de pila (SP)**.
- **Recuperación**:
  - * Se reduce a reacomodar el SP.
- La **compactación** es automática y el **nuevo uso** es trivial.

ADMINISTRACIÓN DE MEMORIA BASADA EN PILAS



ORGANIZACIÓN TRIPARTITA (PASCAL, C)

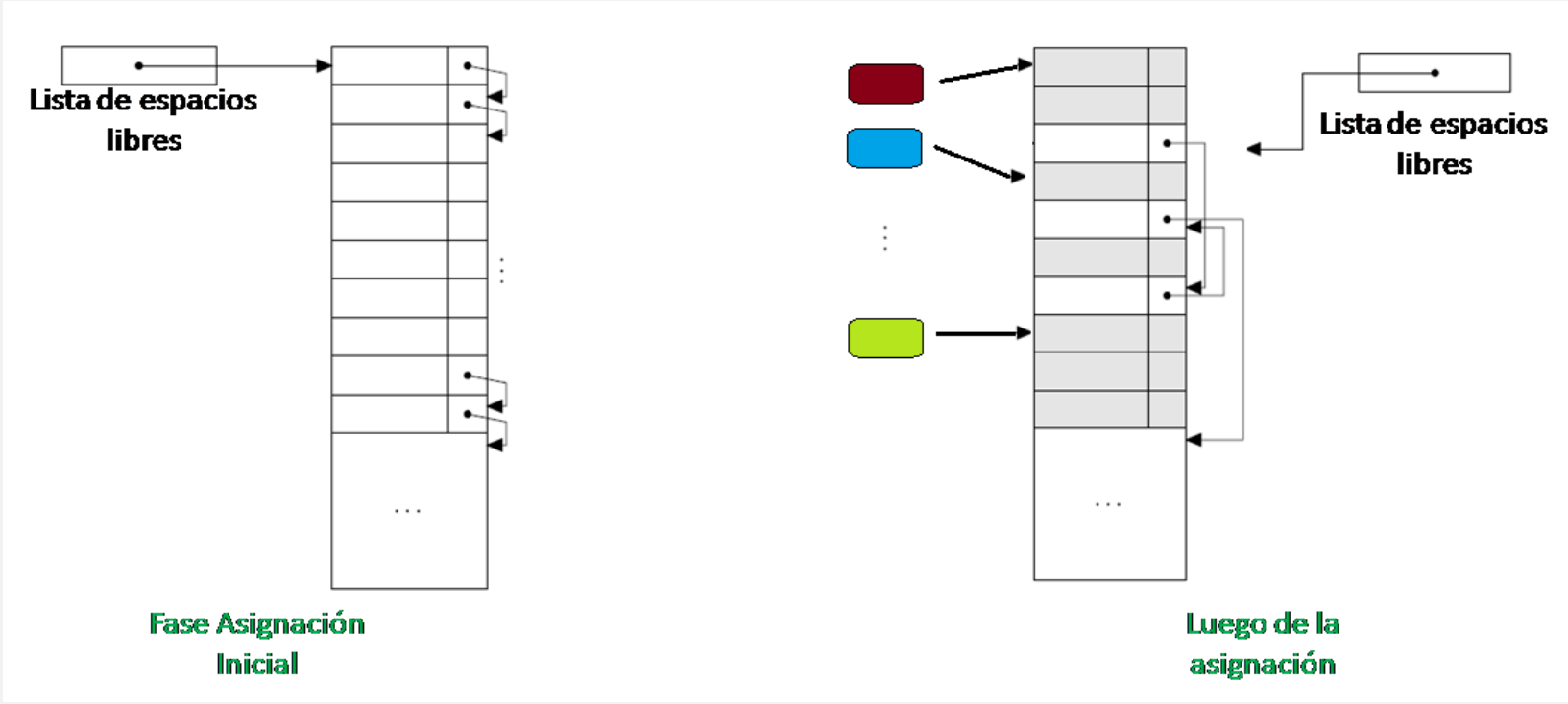


ADMINISTRACIÓN DE MEMORIA BASADA EN HEAP

Heap: bloque de almacenamiento dentro del cual se asignan/liberan bloques de manera no estructurada. Útil para la asignación (o liberación) dinámica de memoria en puntos arbitrarios de la ejecución del programa.

- Según el tamaño de los elementos asignados puede categorizarse en:
  - ✓ Elementos de tamaño fijo (ETF).
  - ✓ Elementos de tamaño variable (ETV).
- Cuando se solicita memoria del heap, el administrador la reserva (en caso de ser posible) y retorna un puntero.
- Los elementos libres forman una lista de espacios disponibles.

ADMINISTRACIÓN DE MEMORIA BASADA EN HEAP: *CON ELEMENTOS DE TAMAÑO FIJO (ETF)*



HEAP CON ETF: ENFOQUES PARA LA RECUPERACIÓN

Objetivo: identificar los elementos que ya no están en uso para devolverlos a la lista de espacios disponibles.

Soluciones:

1) Devolución explícita (por parte del programador o el sistema)
2) Contador de referencias
3) Recolección de basura

1) DEVOLUCIÓN EXPLÍCITA DE LA MEMORIA

El programador (o el sistema) explícitamente indica que el bloque de memoria ya no está en uso (free, delete o dispose).

Ventaja: Permite aprovechar información precisa acerca de cuando el bloque de memoria ya no es necesario.

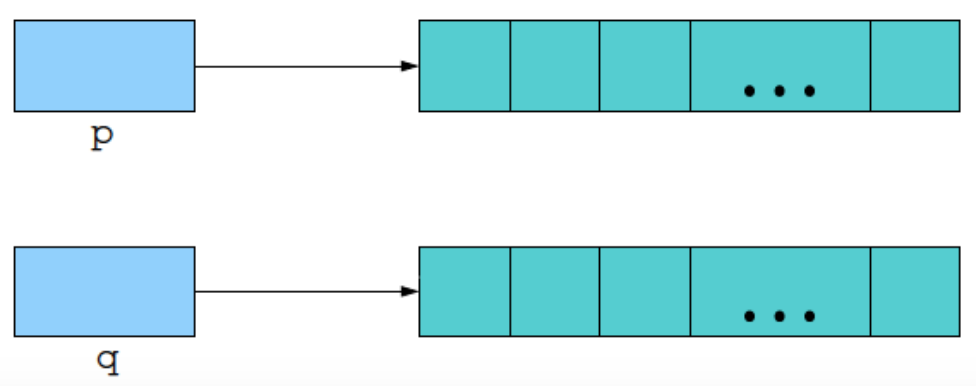
Problemas:

- **Basura**: bloque de memoria inaccesible (se han eliminado todos los pasos de acceso), y que no ha sido devuelto a la lista de espacios disponibles.
- **Referencia desactivada**: ruta de acceso (puntero) a un bloque de memoria que ya ha sido devuelto a la lista de espacios disponibles. Más dañinas que la basura.

I) DEVOLUCIÓN EXPLÍCITA DE LA MEMORIA

Generación de basura

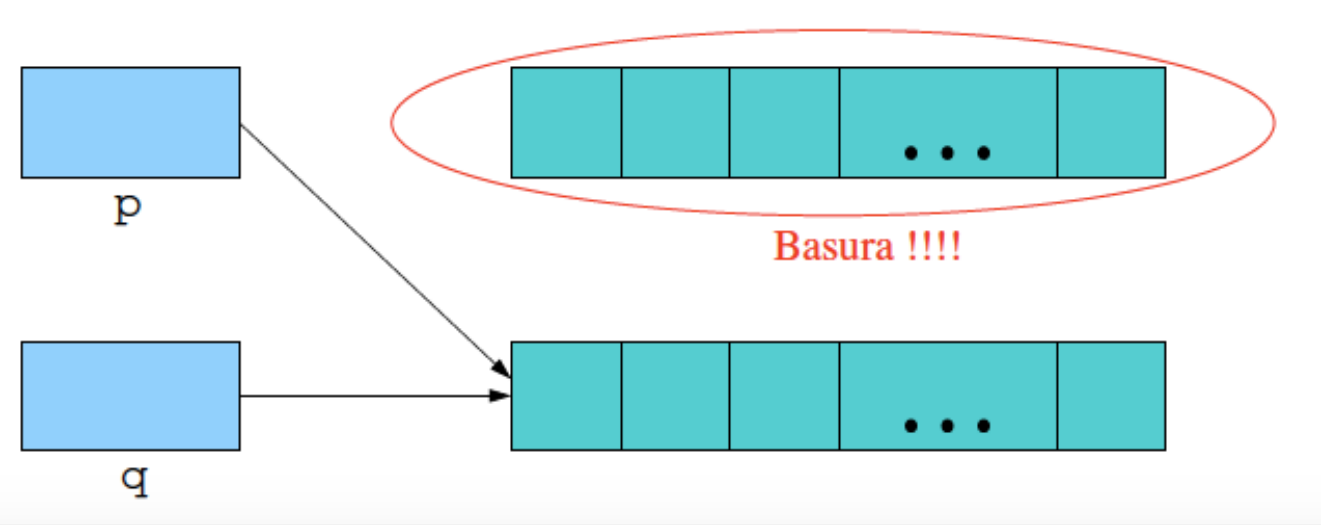
```
char *p, *q;
......
p = (char *) malloc(sizeof(char) * 100);
q = (char *) malloc(sizeof(char) * 100);
```



1) DEVOLUCIÓN EXPLÍCITA DE LA MEMORIA

Generación de basura

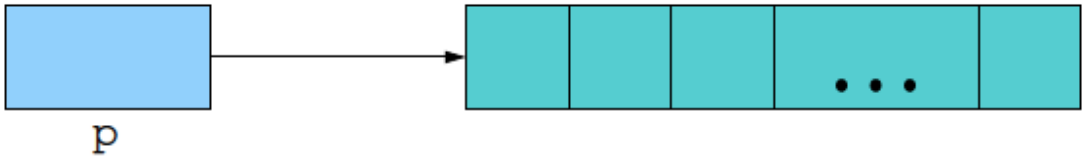
```
char *p, *q;
......
p = (char *) malloc(sizeof(char) * 100);
q = (char *) malloc(sizeof(char) * 100);
p = q;
```



I) DEVOLUCIÓN EXPLÍCITA DE LA MEMORIA

Generación de referencia desactivada

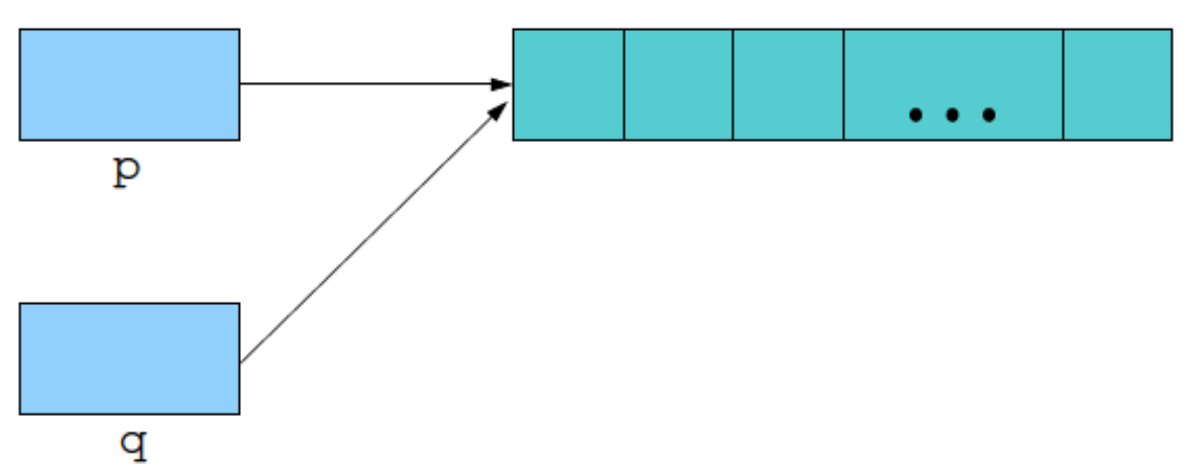
```
char *p, *q;
.......
p = (char *) malloc(sizeof(char) * 100);
q = p;
free((void *) p);
```



I) DEVOLUCIÓN EXPLÍCITA DE LA MEMORIA

Generación de referencia desactivada

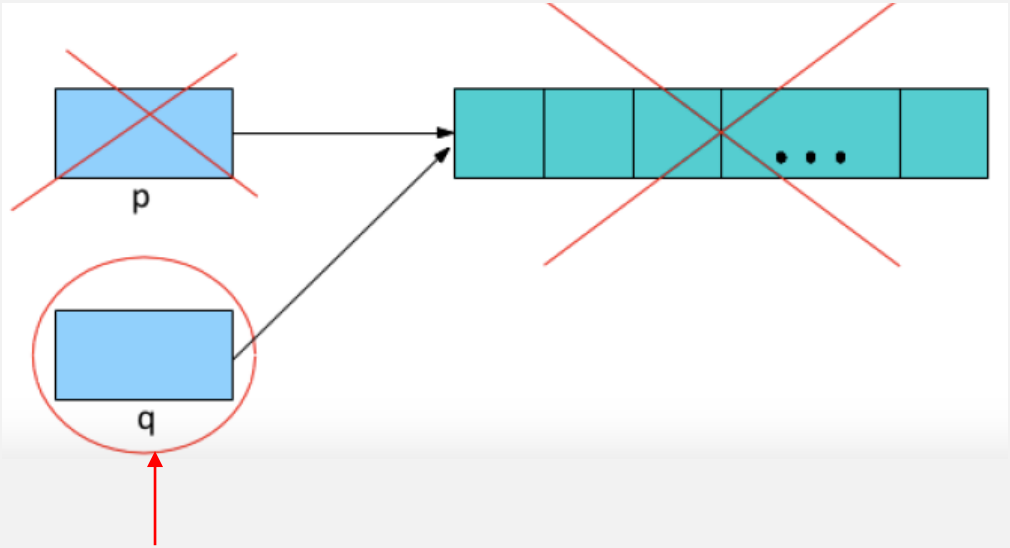
```
char *p, *q;
......
p = (char *) malloc(sizeof(char) * 100);
q = p;
free((void *) p);
```



I) DEVOLUCIÓN EXPLÍCITA DE LA MEMORIA

Generación de referencia desactivada

```
char *p, *q;
......
p = (char *) malloc(sizeof(char) * 100);
q = p;
free((void *) p);
```



2) CONTADOR DE REFERENCIAS

Cada elemento **e** del heap tiene un espacio extra para un contador de referencias (**CRe**) que guarda el número de punteros que apuntan a **e**.

- **Funcionamiento:**
  - Cuando **e** es asignado por primera vez desde la lista de espacios disponibles, **CRe** = 1.
  - Cada vez que un nuevo puntero a **e** es creado ⇒**CRe** = **CRe** + 1.
  - Cada vez que un puntero a **e** es destruido ⇒**CRe** = **CRe** - 1.
  - Cuando **CRe** = 0, **e** es liberado y retornado a la lista de espacios disponibles.
- **Ventajas:** evita la generación de basura y referencias desactivadas.
- **Desventaja:** Costo de mantenimiento de los contadores, espacio y de ejecución (pensar en la asignación p = q).

3) RECOLECCIÓN DE BASURA

Se utiliza cuando no es factible (o es muy costoso) evitar la basura y referencias desactivadas a la vez.

Idea:

- Permitir la generación paulatina de basura (pero no referencias desactivadas).
- Cuando la lista de espacios libres se agote, se invocará un mecanismo de recolección de basura.

Ventajas: evita las referencias desactivadas y se ejecuta esporádicamente.

Desventaja: es costoso.

MECANISMO DE RECOLECCIÓN DE BASURA

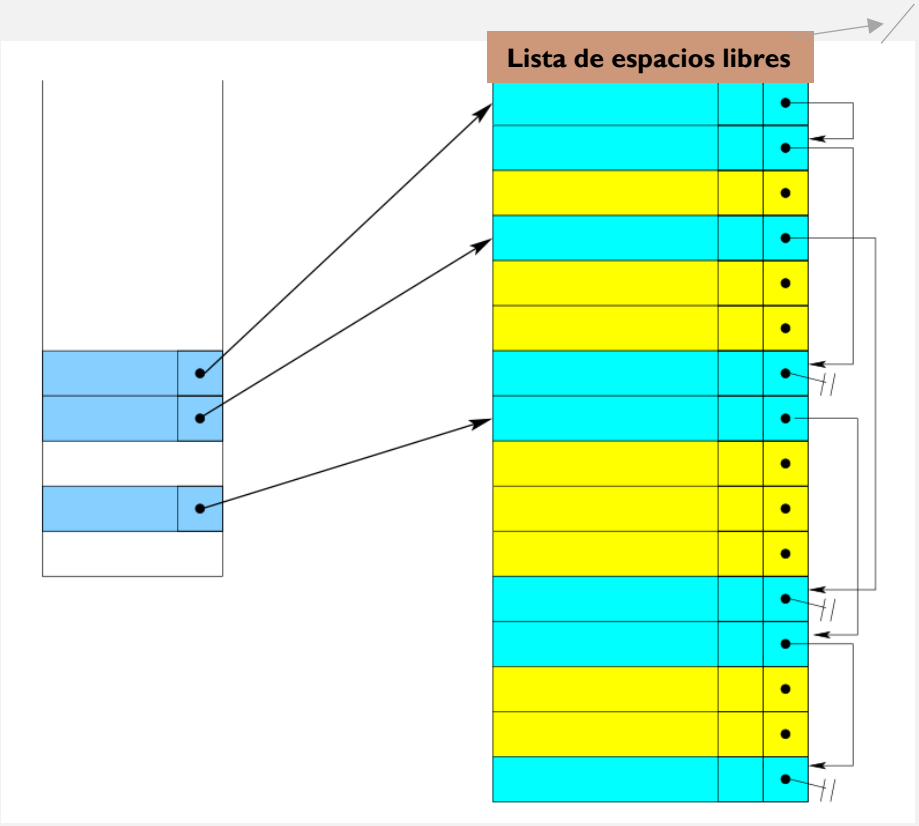
Etapas:

- **Marcar** los elementos activos (en uso), se utiliza un bit de recolección de basura (BRB). Inicialmente todo elemento en el heap es marcado como basura.

- **Barrer** (secuencialmente) el heap y encadenar los elementos basura en la lista de espacios disponibles.

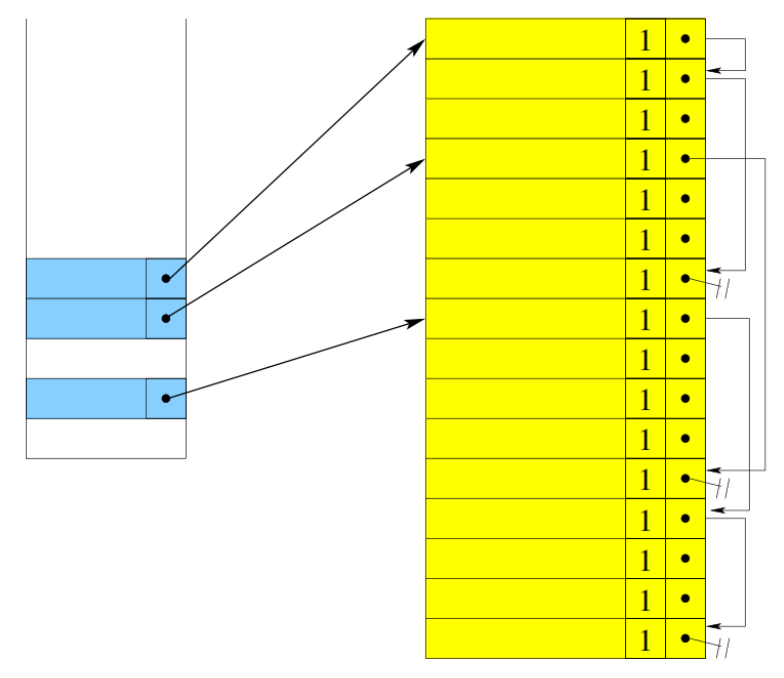
MECANISMO DE RECOLECCIÓN DE BASURA

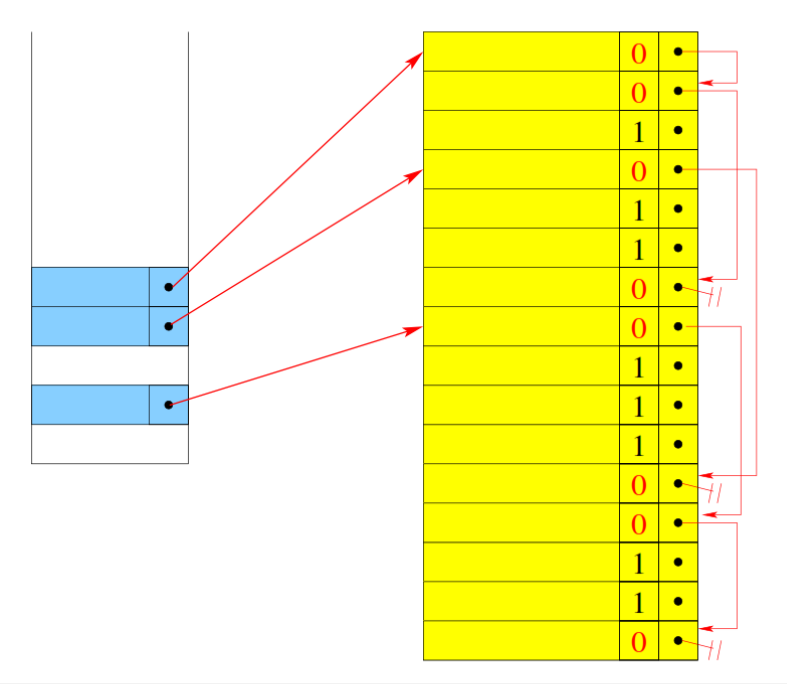
Antes de aplicar el mecanismo...



MECANISMO DE RECOLECCIÓN DE BASURA

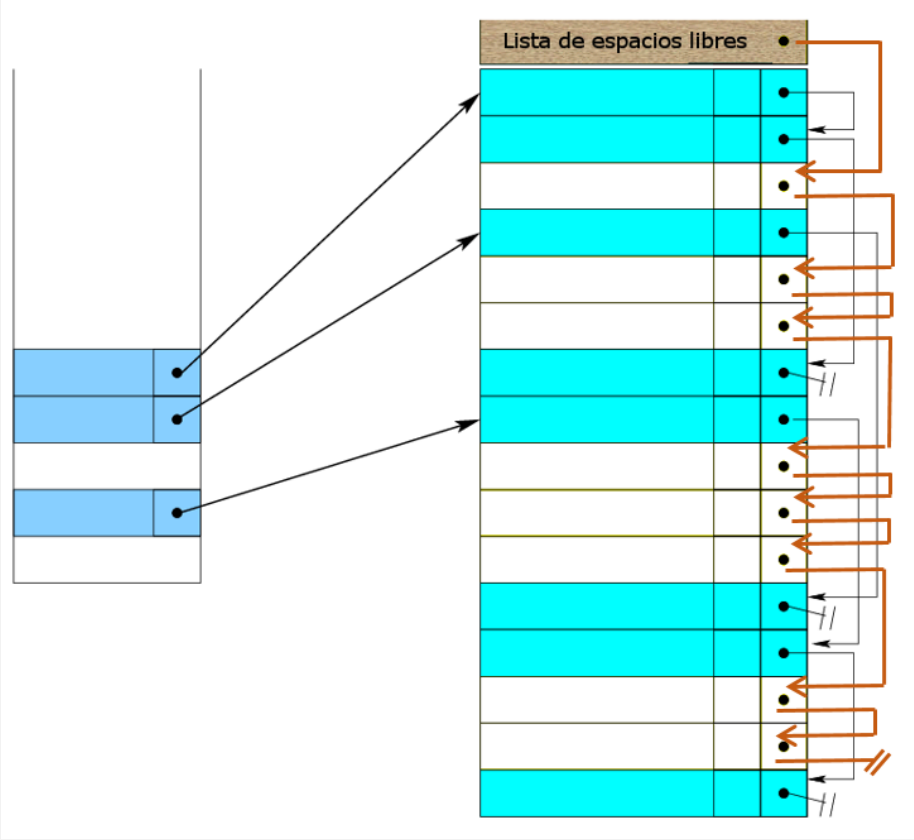
Etapa de marcado:





MECANISMO DE RECOLECCIÓN DE BASURA

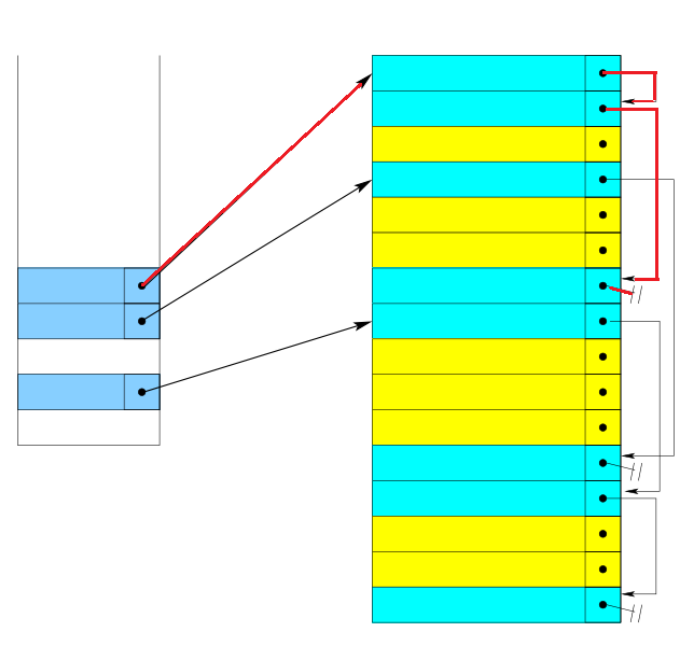
Luego de la etapa de barrido:



MECANISMO DE RECOLECCIÓN DE BASURA

Condiciones que se deben cumplir (para la etapa de marcado):

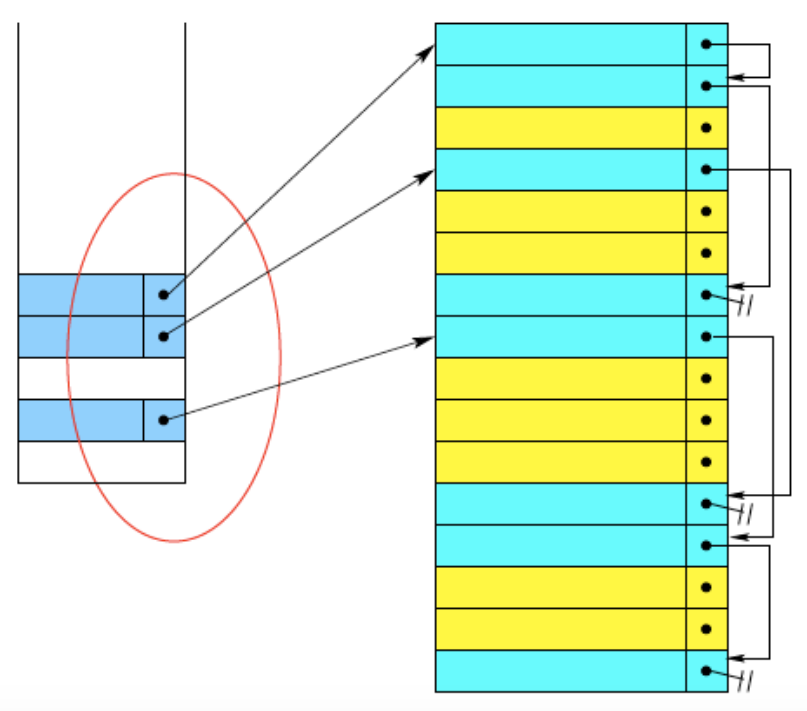
Todo elemento activo **se alcanza a través de una cadena de punteros que comienza fuera del heap** (no existen direcciones computadas).



MECANISMO DE RECOLECCIÓN DE BASURA

Condiciones que se deben cumplir (para la etapa de marcado):

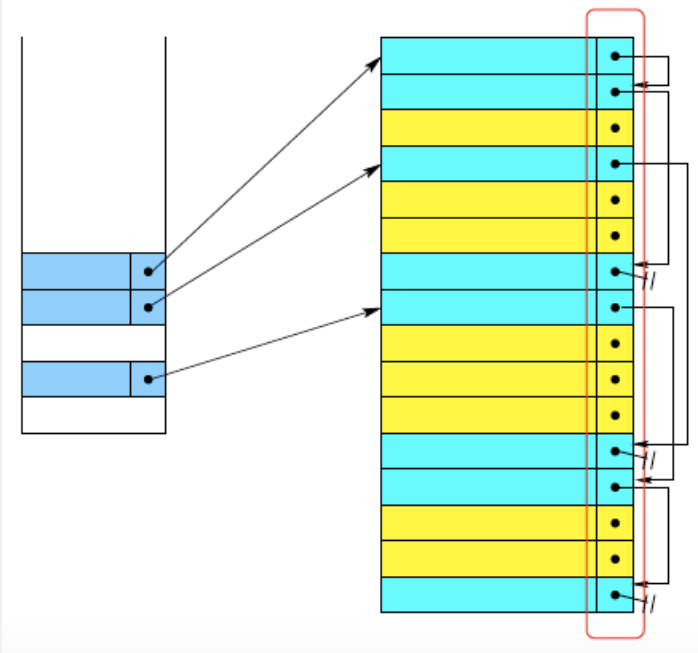
Debe ser posible identificar todo puntero fuera del heap, que apunta a un elemento dentro de él.



MECANISMO DE RECOLECCIÓN DE BASURA

Condiciones que se deben cumplir (para la etapa de marcado):

Debe ser posible **identificar** dentro de cada elemento activo en el heap **los campos** que contienen punteros a otros bloques del heap.



HEAP CON ETF: COMPACTACIÓN Y NUEVO USO

La compactación no es un problema ya que los bloques disponibles son del mismo tamaño.

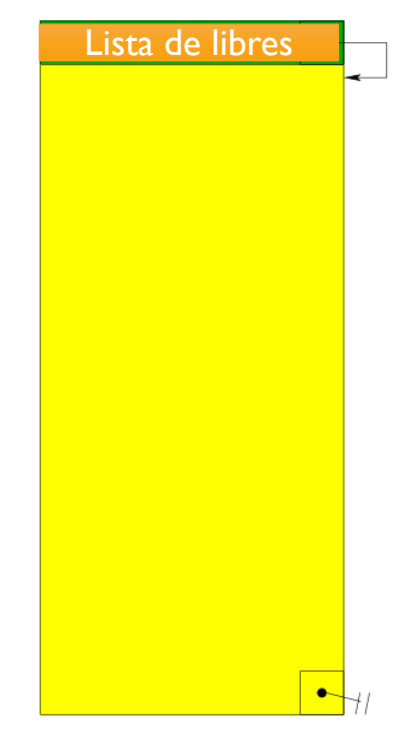
El nuevo uso es trivial (los bloques tan pronto sean recuperados, estarán disponibles para su nuevo uso).

ADMINISTRACIÓN DE MEMORIA BASADA EN HEAP: *CON ELEMENTOS DE TAMAÑO VARIABLE (ETV)*

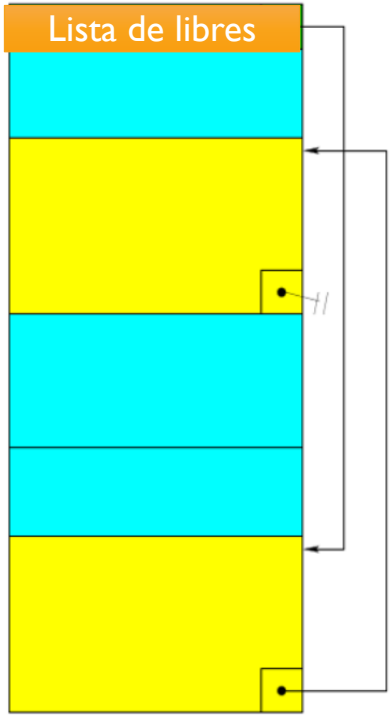
- La gestión de este mecanismo es más compleja debido al tamaño variable de los elementos.
- El espacio recuperado tal vez no pueda ser usado inmediatamente.
- El tamaño de los bloques que se asignan desde el heap puede variar (*malloc()*, *new()*, etc.).

ADMINISTRACIÓN DE MEMORIA BASADA EN HEAP: *CON ELEMENTOS DE TAMAÑO VARIABLE (ETV)*

Asignación inicial



Luego del uso



HEAP CON ETV: ENFOQUES PARA LA RECUPERACIÓN

Objetivo: identificar los elementos que ya no están en uso para devolverlos a la lista de espacios disponibles.

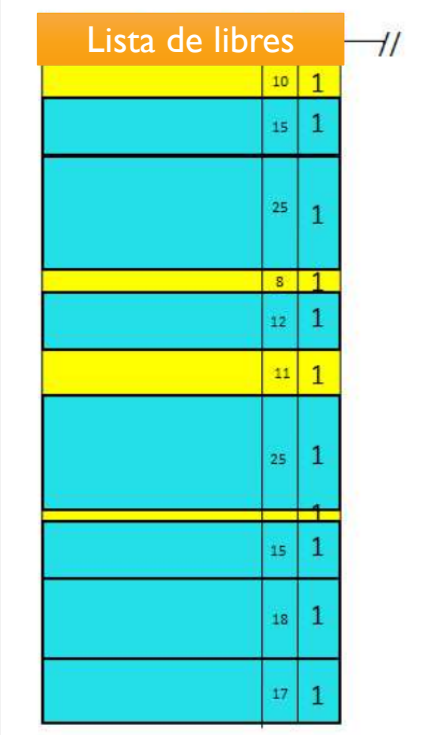
Soluciones:

1) Devolución explícita (por parte del programador o el sistema)
2) Contador de referencias
3) Recolección de basura ----▸ se dificulta la etapa de barrido porque se debe determinar la longitud del bloque a devolver a la lista de espacios disponibles.

HEAP CON ETV: ENFOQUES PARA LA RECUPERACIÓN

Recolección de basura ----▸ se dificulta la etapa de barrido porque se debe determinar la longitud del bloque a devolver a la lista de espacios disponibles.

Solución: agregar un entero indicando la longitud/tamaño del bloque.



HEAP CON ETV: COMPACTACIÓN Y NUEVO USO

La compactación intenta solucionar problemas de fragmentación de la memoria por el uso de un heap con ETV.

Puede existir espacio suficiente pero estar distribuido en pequeños fragmentos por todo el heap.

Formas de Compactación:

- **Parcial:** se realiza sólo entre bloques adyacentes (los elementos activos no se pueden desplazar o es muy costoso).
- **Cabal:** todos los elementos activos son desplazados a un extremo del heap. Es muy costoso porque los apuntadores a elementos activos deben ser actualizados.

HEAP CON ETV: COMPACTACIÓN Y NUEVO USO

En el nuevo uso se busca en la lista de los espacios libres un bloque del tamaño apropiado.

Se retorna el espacio solicitado y el sobrante se convierte en un nuevo bloque libre del heap.

Políticas de asignación de un bloque libre:

- Primer ajuste.
- Mejor ajuste.
  Problema.... fragmentación.
  Solución: compactar todo el espacio libre en uno de los extremos del heap (compactación cabal).

¿DUDAS?