Lenguajes de Programación, 2022-1
Nota de clase 12: Recolección de Basura

Karla Ramírez Pulido J. Ricardo Rodríguez Abreu Manuel Soto Romero

6 de enero de 2022
Facultad de Ciencias UNAM

Una de las restricciones computacionales más estudiadas además del tiempo, es el manejo de memoria. Una manera de no preocuparse directamente por el contínuo uso de ésta, es utilizar un programa que ayude a revisar de manera metódica ciertos escenarios en los que la automatización beneficie la ejecución de programas.

12.1. Introducción

Dos de las principales restricciones computacionales más conocidas son el tiempo y el espacio. Durante años se ha debatido la importancia de ambas y su interacción casi siempre contraproducente al tratar de optimizar sólo una [7]. A diferencia del manejo del tiempo que se hace difícilmente explícito mediante bibliotecas dentro de los lenguajes de programación de alto nivel, la memoria ha tenido una rápida y gran evolución observable.

A finales del siglo XX, uno de los lenguajes de programación más usados era C que estableció una manera de solicitar y liberar memoria en el área de lenguajes. Esta forma se convirtió en un estándar por muchos años. C hace uso de dos funciones llamadas `malloc` y `free`, las cuales se encargan de pedir al sistema operativo memoria y liberarla respectivamente del *heap*.

Por otro lado, dentro del estilo de programación funcional, en 1958 fue desarrollado el lenguaje de programación Lisp, el cual implementó poco después, un programa escrito por Daniel Edwards (1959) que funcionaría como el primer programa encargado de manejar la memoria de los programas de manera automática. Este programa sería reconocido posteriormente como el primer programa de recolección de basura [8].

12.2. Definición

Aunque en muchos lenguajes de programación, las acciones de liberación de espacios de memoria se siguen realizando de forma manual y varía dependiendo de la forma en que se escribe el programa, la tendencia actual es que los lenguajes provean un programa que automatice estos procesos.

Definición 12.1. (Recolección de Basura) *La recolección de basura es el proceso de la reclamación automática de la memoria (o espacio) de la computadora* [7].

Al proceso que se encarga de la recolección de basura, y al programa encargado de realizarlo, se le denomina recolector de basura, éste posee definiciones, métodos de procesamiento y algoritmos de recolección.

Las ventajas que ofrecen los lenguajes de programación con recolección de basura automática (por mencionar algunas) son [4, 5]:

1. Minimización de *fugas*[1] de memoria.
2. Persistencia en los datos utilizados a través de la ejecución de un programa.
3. Minimización de errores (principalmente con apuntadores de objetos).
4. Optimizaciones de memoria usando heurísticas predictivas.

Aunque pareciera que siempre es buena idea utilizar un lenguaje de programación que tenga un método automatizado de recolección de basura, existen razones por las cuales se puede argumentar la decisión de delegar el manejo de memoria al programador: [6]

1. El llenado de la memoria puede ser independiente de la ejecución del programa debido a que la memoria es compartida con otros programas del sistema operativo.
2. El tiempo que usa el método de recolección de basura es en general proporcional a la cantidad de espacio o memoria aún utilizado por el programa.
3. El recolector de basura debe acceder forzosamente a *páginas*[2] que se encuentran en almacenamiento secundario, aumentando el tiempo de ejecución.
4. Es un proceso completamente disruptivo; el recolector de basura no puede ser ejecutado mientras el programa se encuentre corriendo.

Las ventajas y desventajas de un recolector de basura se ven directamente asociadas con la implementación, algoritmos y definición de conceptos usados por el recolector para ejecutar soluciones que intenten minimizar un impacto sobre la ejecución de un programa y aumenten los beneficios del manejo de memoria automatizado.

12.3. Detección de basura

Si bien la definición de qué es basura y que no, es completamente abierta a cada diseñador de lenguaje de programación o programador de recolector de basura, hay principalmente dos vertientes que se usan para definir cuando un objeto es basura y ambas poseen un impacto directo sobre la eficiencia y tipo de solución que obtendrá el recolector.

Todos los objetos a considerar tienen que estar almacenados en algún punto donde sean alcanzable al programa que se encuentra siendo ejecutado. Este conjunto en el que puede acceder el programa para operar con los objetos se le llama *conjunto raíz*. En otras palabras, el conjunto raíz es un conjunto de objetos accesibles desde el programa como variables globales, o cualquier variable almacenada en la pila de ejecución [8].

12.3.1. Conteo de referencias

En este método de detección de basura más antiguo, implementado por primera vez junto con el recolector de basura de LISP, tiene la particularidad de que a cada objeto tiene asociado un contador de referencias denominado

[1] También conocidas en inglés como *leaking*.

[2] En ciencias de la computación, una página es una unidad de memoria (`Kb`, `Mb`, `bytes`, etc.) que puede variar de un sistema operativo a otro [7].

C. Este contador tiene la particularidad de ser inicializado al momento de la creación del objeto y mantiene su dirección.

$$Var \quad a = new \quad Var(0x392) \rightarrow C_{0x392} = 1.$$

Para que un objeto[3] sea considerado basura, su contador C debe llegar al valor cero, esto quiere decir que no exista algún objeto apuntando hacia él [6].

En la Figura 1 se puede observar cómo se obtienen las dos direcciones de memoria que tienen conteo de referencia cero.

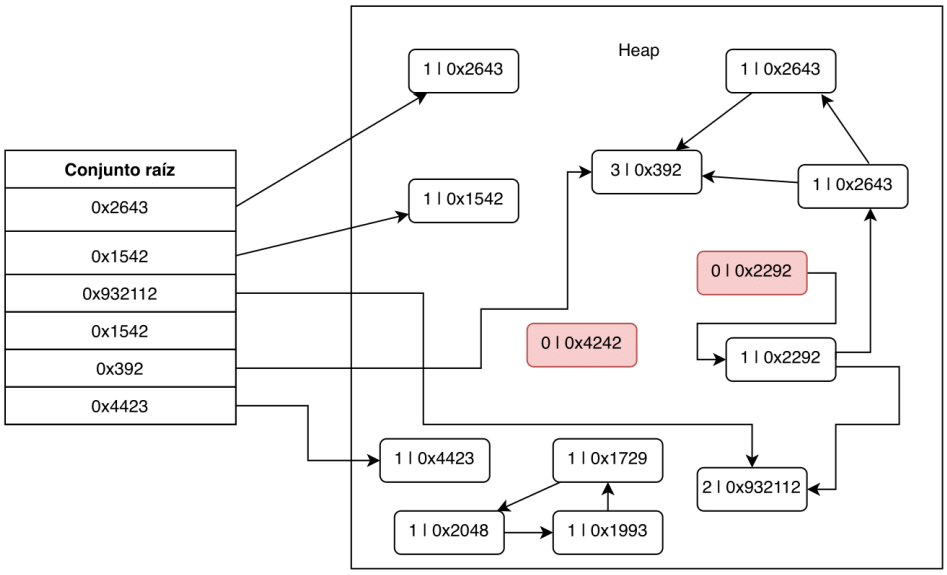

Figura 1: Ejemplo de objetos en memoria siendo eliminados en tiempo T_1.

En la siguiente figura, la Figura 2, ya se observa cómo se eliminaron las direcciones que tenían referencias cero, generando una nueva dirección considerada como basura.

[3]Un objeto puede hacer referencia a un valor, estructura, instancia de una clase, básicamente cualquier cosa que pueda ser almacenada en una variable y ocupe espacio en la memoria.

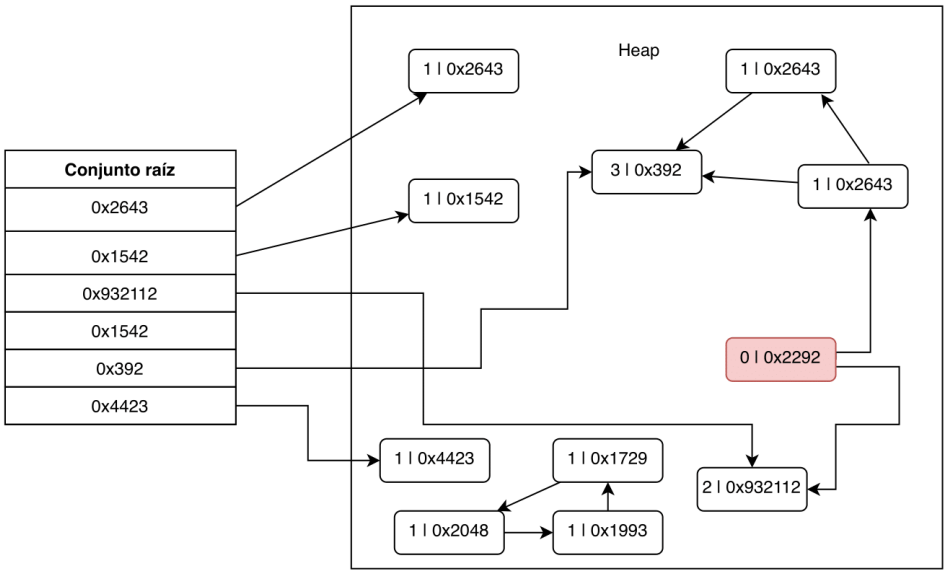

Figura 2: Ejemplo de objetos en memoria siendo eliminados en tiempo T_2

Tanto la Figura 3 como la Figura 4 muestran la continuación de la detección y posterior eliminación de direcciones resultantes como basura al disminuir sus referencias a cero.

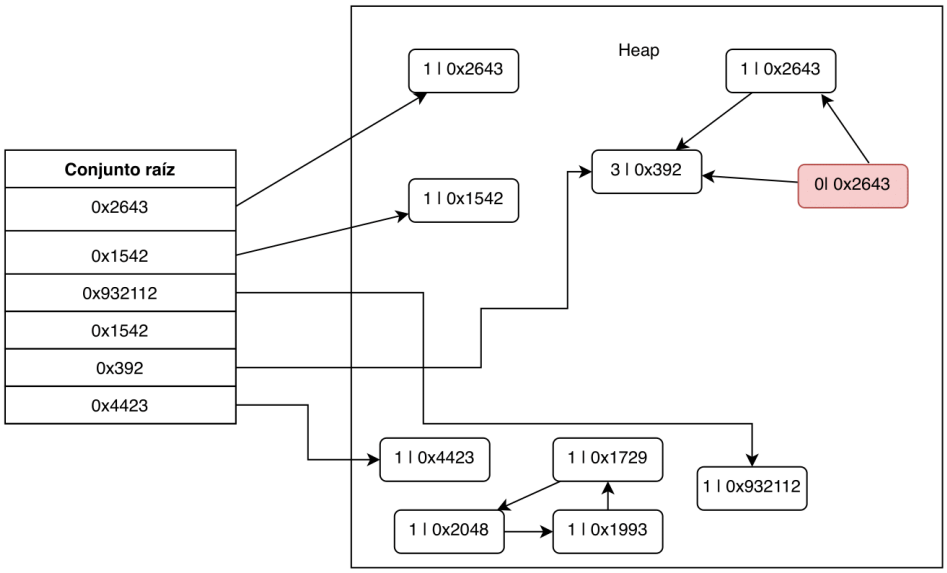

Figura 3: Ejemplo de objetos en memoria siendo eliminados en tiempo T_3

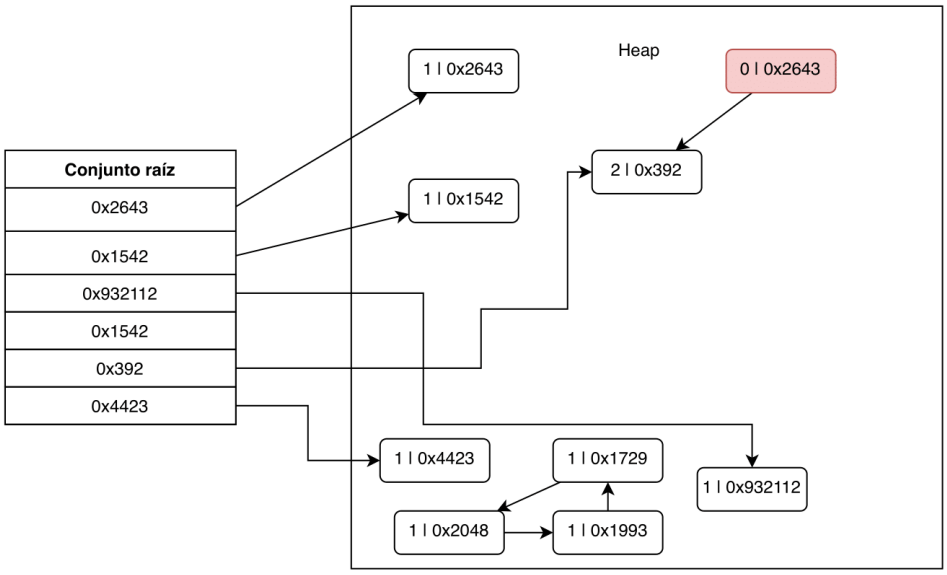

Figura 4: Ejemplo de objetos en memoria siendo eliminados en tiempo T_4

Como se puede apreciar en la Figura 5, existen tres objetos coloreados de amarillo en los que se observa lo no posibilidad de acceder a ellos desde el conjunto raíz, la definición de conteo de referencias permite que dichos objetos no sean eliminados del *heap*. Este problema se le llama **apuntadores colgantes**.

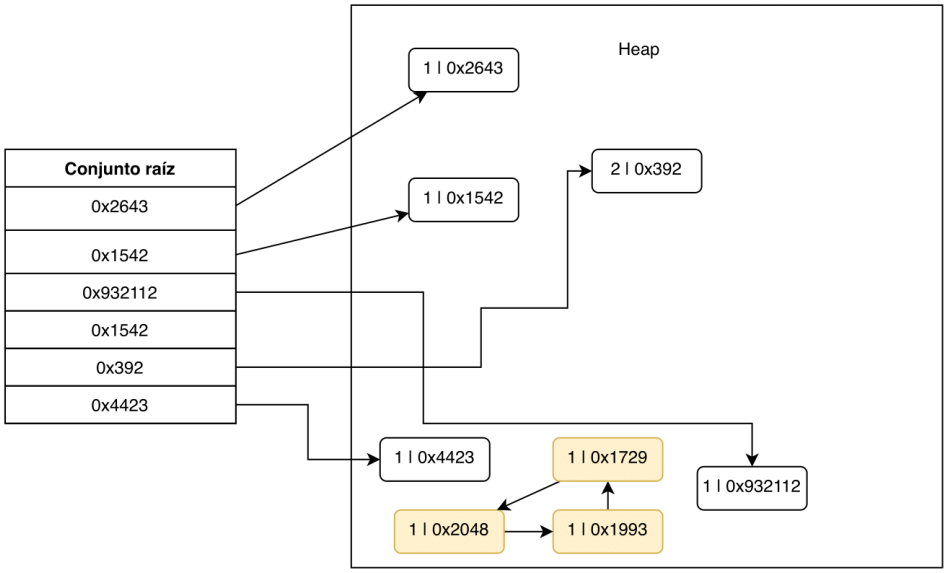

Figura 5: Resultado de eliminar todos los objetos clasificados como basura.

Adicionalmente a los apuntadores colgantes, otra desventaja de este método de detección de basura es el espacio que se necesita para el almacenamiento de la variable de conteo por cada objeto, así la necesidad de actualizar la variable cada que el objeto es referenciado.

12.3.2. Camino del conjunto raíz

Otra forma de definir a un objeto como basura es obtener todos los objetos que son alcanzables desde el conjunto raíz. Todo objeto será considerado basura si al extenuar la búsqueda de caminos, estos objetos nunca fueron marcados como visitados.

Tanto en la Figura 6 como la Figura 7 se observa la detección de los objetos que no son basura que son aquellos que tienen una referencia al conjunto raíz o aquellos que tienen una referencia a un objeto que ya ha sido marcado como *vivo*[4].

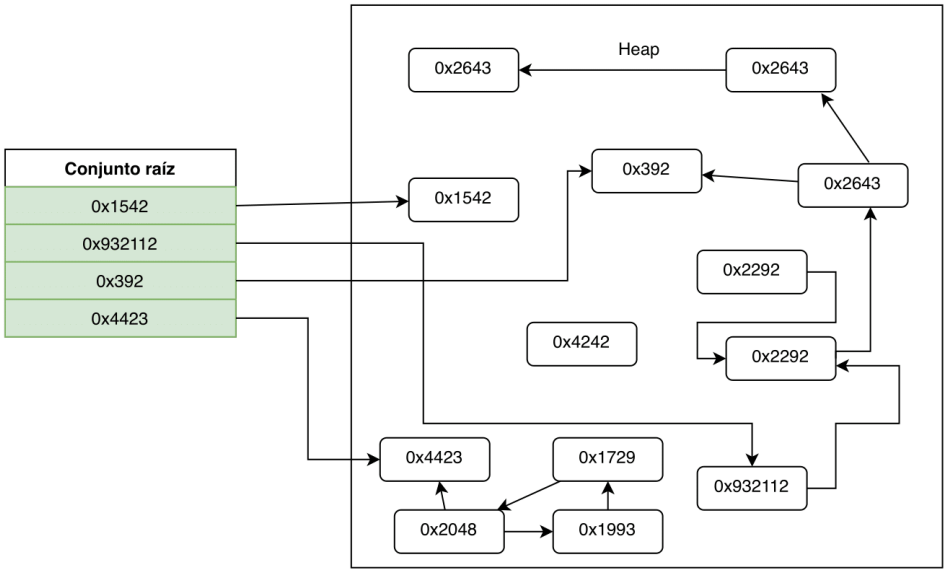

Figura 6: Objetos marcados mediante caminos desde el conjunto raíz en tiempo T_0

Tanto la Figura 7 como la figura 8 muestran la continuación de la detección de direcciones vivas resultantes al extender su iteración sobre vecinos de los previamente considerados objetos vivos.

[4]Un objeto vivo es un objeto que no es considerado basura.

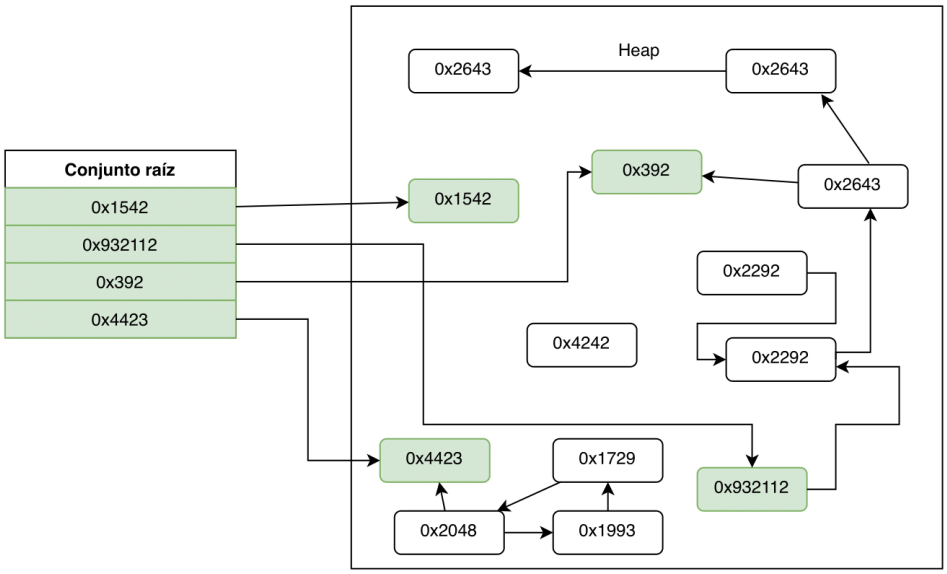

Figura 7: Objetos marcados mediante caminos desde el conjunto raíz en tiempo T_1

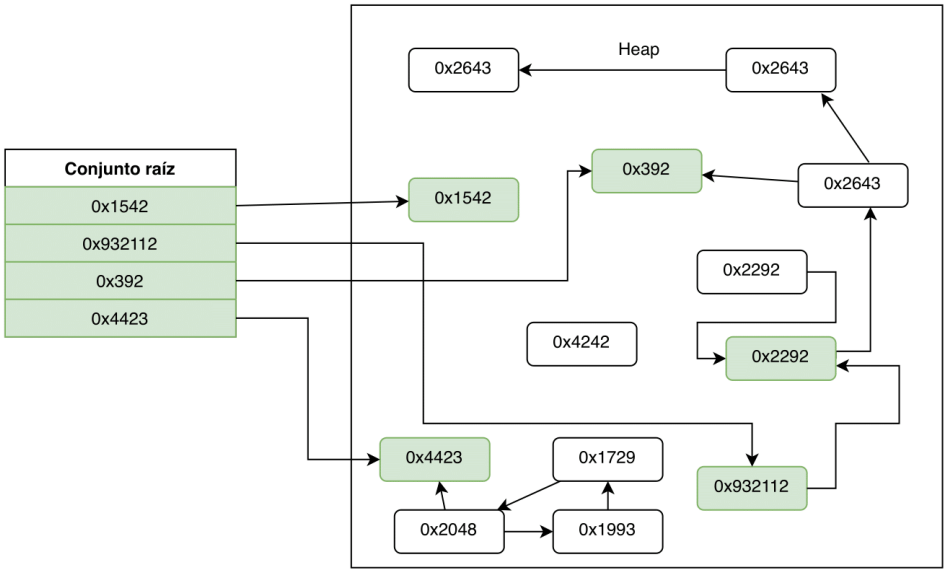

Figura 8: Objetos marcados mediante caminos desde el conjunto raíz en tiempo T_2

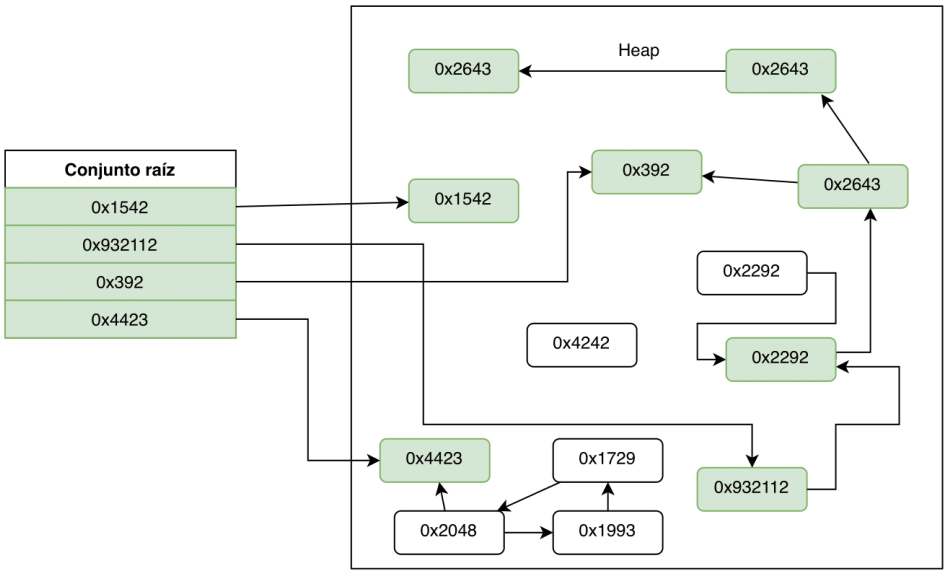

Figura 9: Objetos marcados mediante caminos desde el conjunto raíz en tiempo T_5

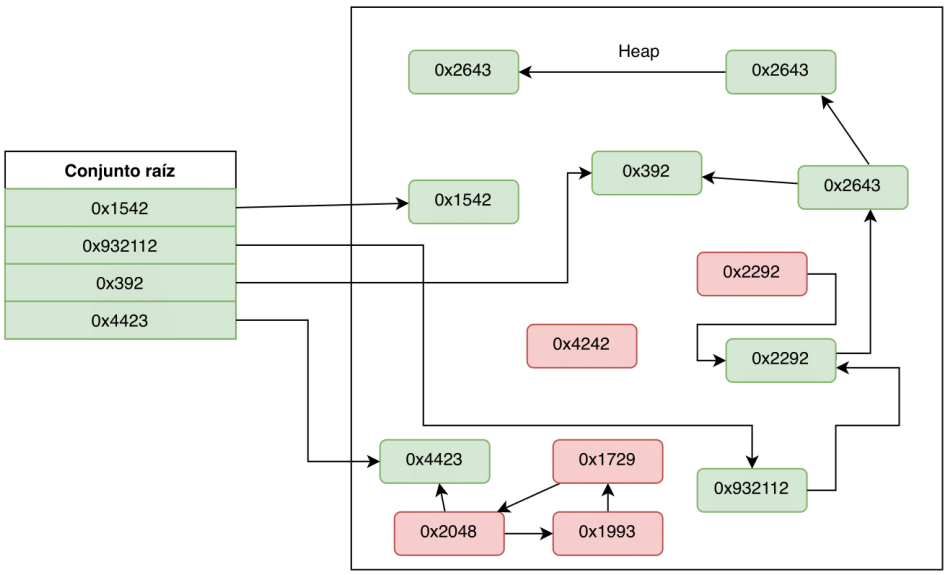

Figura 10: Objetos marcados en rojo como basura en el tiempo T_6 debido a que no se encontró ningún camino.

Una ventaja de este método de detección de basura es que, como se puede ver en la Figura 11, la propiedad de obtener un ambiente transitivo de vecinos que permite detectar y eliminar los apuntadores colgantes.

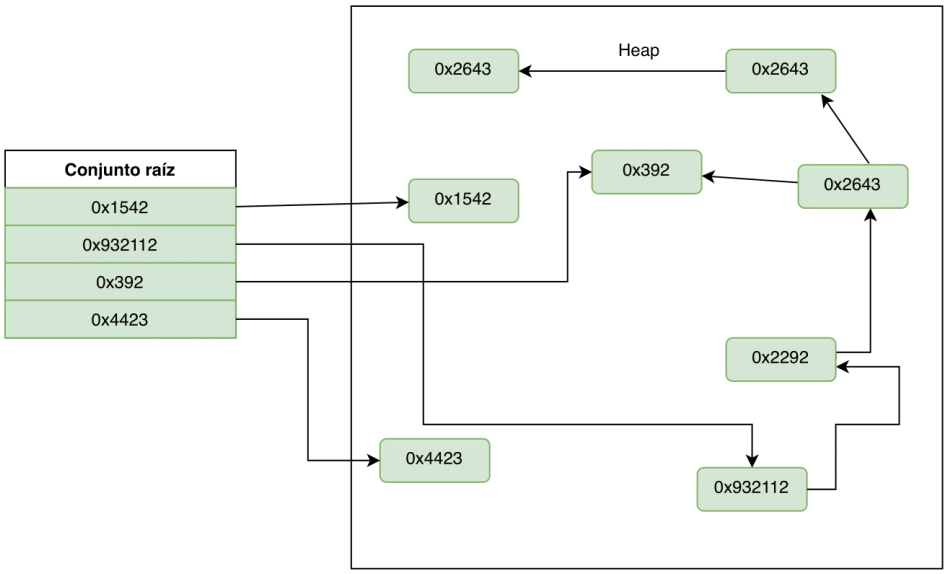

Figura 11: Memoria después de eliminar los objetos basura.

La gran desventaja que tiene este método es el tiempo de ejecución; la búsqueda de caminos tiene el gran problema de las referencias cíclicas. Si existen muchas referencias apuntándose entre ellas, este método se vuelve lento e ineficiente [6].

En los lenguajes de programación actuales con alta optimización en métodos de detección de basura (como JAVA o PYTHON), la detección de basura utiliza una sofisticada combinación de ambos métodos de marcado [9]. La elección de un método sobre otro también impacta directamente en la decisión de qué algoritmo se usará para realizar la eliminación de los objetos marcados.

12.4. Algoritmos de recolección

Existen tantos algoritmos de recolección de basura como implementaciones de lenguajes de programación con recolectores. Un alto porcentaje de estas implementaciones están basadas en un par de algoritmos y son adaptadas a las necesidades de los programadores y capacidades del lenguaje de programación. La elección de qué algoritmo se puede usar tiene como criterios objetivos ser eficiente en tiempo, espacio o ambos, sin embargo, el impacto a las complejidades se puede reducir usando alternativas no tan estrictas de los algoritmos [5].

12.4.1. Marcado y barrido

Este es el algoritmo más sencillo de implementar: utilizando algún método de identificación como los vistos en la sección anterior, lo primero que se debe realizar es un marcado de basura, esto quiere decir que objeto por objeto se revisará si se mantendrá o se debe eliminar. Terminando la etapa del marcado, lo siguiente es la liberación de cada objeto, dejando el espacio que éste ocupaba libre para su futura utilización [5].

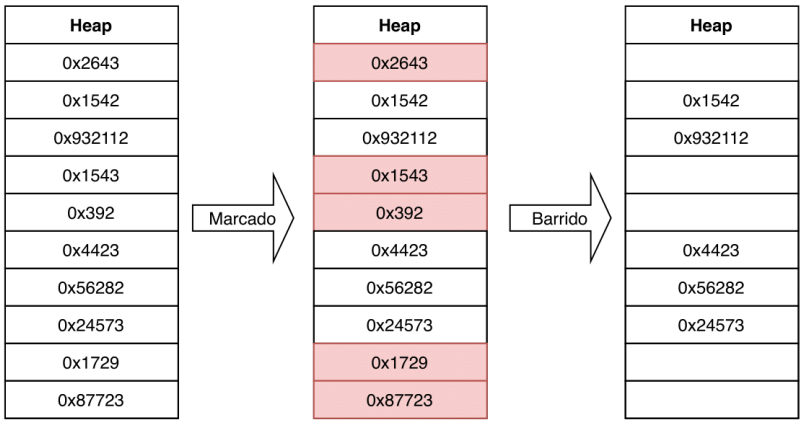

Figura 12: Ejemplo del algoritmo de marcado y barrido.

Ventajas	**Desventajas**
Es rápido.	Existe la fragmentación de memoria.
Es fácil de implementar.	Los espacios liberados tienen un tamaño variable.
Inmediata disponibilidad del espacio liberado.	No arregla el problema del manejo de automatización de memoria de manera eficiente.

El algoritmo termina inmediatamente después de liberar el último elemento marcado como basura. En la siguiente iteración el algoritmo de recolección, se regresa al marcado, ignorando los espacios en blanco.

12.4.2. Marcado y compactación

Este algoritmo es muy similar al de marcado y barrido, con la diferencia de que existe un paso posterior al barrido que es la compactación de los elementos "vivos" en un bloque de referencias. Juntando los bloques vivos y eliminando los espacios vacíos, se asegura que el uso de memoria es eficiente [5].

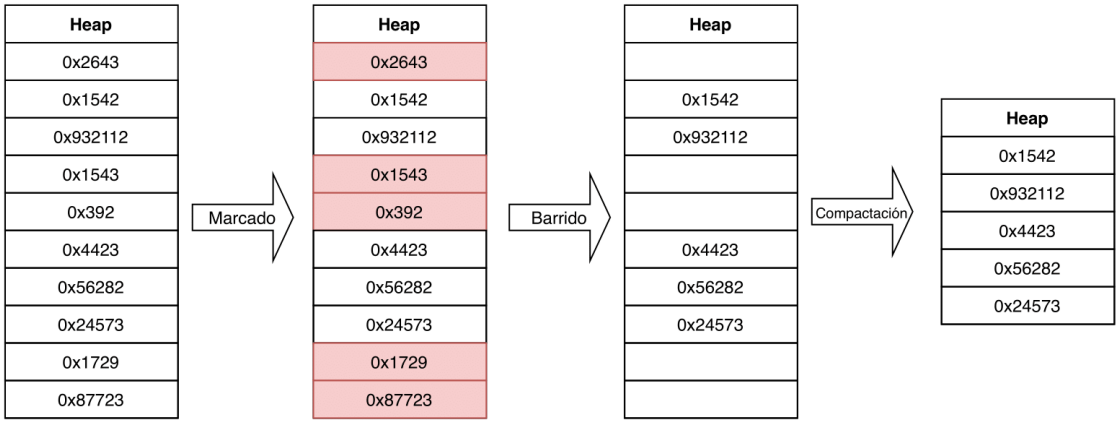

Figura 13: Ejemplo del algoritmo de marcado y compactación.

Ventajas	Desventajas
Es eficiente en memoria utilizada.	Lento en tiempo debido al acomodo de los bloques de memoria.
Es fácil de implementar.	La implementación del movimiento de memoria siempre es una operación de riesgo.
Evita problemas de fragmentación.	

12.4.3. Copia y pega usando semiespacios

Este particular algoritmo consiste en dividir la memoria en dos partes llamadas *FromSpace* y *ToSpace*. El *FromSpace* es la parte que se encuentra siendo trabajada por el programa que se está ejecutando. En el momento que el programa deja de funciones para dar lugar al recolector de basura, todos los objetos vivos pasan del *FromSpace* al *ToSpace* y las etiquetas se invierten, siendo siempre el *FromSpace* el espacio en el que se encuentra siempre trabajando el programa. El algoritmo funciona de la siguiente forma [5]:

Algoritmo 1 Algoritmo Copia y Pega usando semiespacios

```
procedure semiespacios(F,T)
   for all N in F do
      if EsBasura(N) then
         continue
      Copia(N,F)
      Pega (N,T)
   Limpia(T)
   Swap(T,F)
```

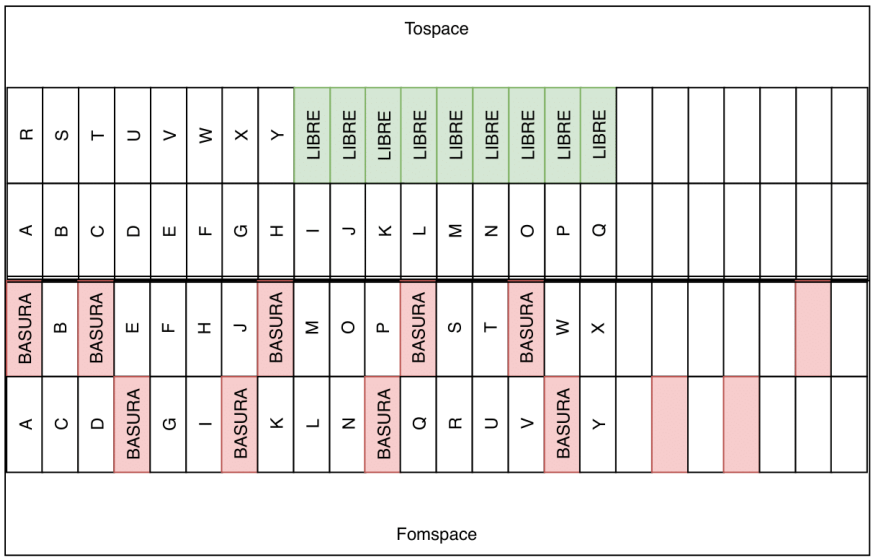

Figura 14: Ejemplo del algoritmo copia y pega usando semiespacios.

Ventajas	Desventajas
Es eficiente en memoria utilizada (si se toma como eficiencia la mitad usada).	Se pierde la mitad del espacio del *heap*.
Es eficiente en tiempo debido a que sólo pasa una vez.	La implementación no es trivial.
Evita problemas de fragmentación.	

12.4.4. Otros algoritmos

Los algoritmos enunciados previamente no son únicos. Ya se había mencionado que existen otros algoritmos de recolección de basura como barrido por copia generacional (*GenCopy)*, marcado y barrido generacional (*GenMS*), conteo de referencias generacional (*GenRC*) por nombrar algunos. Estos algoritmos en su mayoría son algoritmos híbridos que intentan minimizar las desventajas de cada uno o son algoritmos que por su naturaleza, salen del alcance de estas notas [5].

12.5. Métodos de recolección

El recolector de basura como bien se dijo, detiene el proceso e hilo de ejecución del programa que se encuentra en ejecución debido a lo volátil de sus operaciones. La decisión de en qué momento realizar su ejecución es una decisión de implementación del lenguaje en el que se encuentre.

Una vez detenido el hilo del programa en un tiempo K, existen dos principales métodos para decidir la duración de recolección: generacional por tiempo y espacio [5].

12.5.1. Generacional por tiempo

En este método de recolección, el programa de recolección de basura le indica al objeto recolector que debe suspender la ejecución del programa que se encuentre corriendo y ejecutar alguno de sus algoritmos durante cierto tiempo. Por ejemplo:

Si el recolector decide que la forma óptima de limpiar la memoria es ser ejecutado durante 3 segundos, el recolector de basura se ejecutará cada K segundos durante 3 segundos [5].

12.5.2. Generacional por espacio

Suponiendo que ha pasado K tiempo y el recolector empieza a ejecutarse sobre un fragmento de memoria, el método de recolección generacional por espacio le indicará al recolector de basura qué tanto leer para limpiar. Puede ser N números de página, Q, Mb, Kb, etc.

Nótese que tanto generacional por tiempo como general por espacio son ejecutadas cada K tiempo. La diferencia entre ambas es qué factor observar para decidir detener la ejecución del recolector [5].

Referencias

[1] Arora, Sanjeev y Boaz Barak: *Computational complexity: a modern approach.* Cambdrige University Press, 2009.

[2] McCharthy, John: *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Commun.* ACM, 3(4):184195, Abril 1960, ISSN 0001-0782. https://doi.org/10.1145/367177.367199.

[3] Wilson, Paul R.: *Uniprocessor garbage collection techniques*. En Bekkers, Yves and Cohen, Jacques (editores): *Memory Management*, páginas 1-42, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg, ISBN 978-3-540-47315-2.

[4] Huang, Xianglong, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang y Perry Cheng: The Garbage Collection Advantage: Improving Program Locality 39(10):6980, Octubre 2004, ISSN 0362-1340. [7] McCarthy, John: chine, Part I . SIGPLAN Not., https://doi.org/10.1145/1035292.1028983 .

[5] Blackburn, Stephen M., Perry Cheng y Kathyn S. McKinley: *Myths and Realities: The Performance Impact of Garbage Collection.* SIGMETRICTS Perform. Eval. Rev., 32(1):25-36, Junio 2004, ISSN 0163-5999. https://doi.org/10.1145/1012888.1005693.

[6] Deutsch, L. Peter y Daniel G. Bobrow: *An Efficient, Incremental, Automatic Garbage Collector.* Commun. ACM, 19(9): 522-526, Septiembre 1976, ISSN 0001-0782. https://doi.org/10.1145/360336.360345.

[7] Flynn, M. J., Gray, J. N., Jones, A. K., Lagally, K., Opderbeck, H., Popek, G. J., Randell, B., Saltzer, J. H., Wiehle, H. R., Bayer, R., Graham, R. M., & Seegmüller, G. (1978). Operating Systems: An Advanced Course (Lecture Notes in Computer Science, 60) (1978th ed.). Springer.

[8] Bartlett, J. F. (1988). Compacting garbage collection with ambiguous roots. ACM SIGPLAN Lisp Pointers, 1(6), 3-12.

[9] Oracle: *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*. `https://docs.oracle.com/javase/9/gctuning/available-collectors.htm#JSGCT-GUID-F215A508-9E58-40B4-90A5-74E29BF3BD3C`