

# Lecture 12 – Garbage Collection

## COSE212: Programming Languages

Jihyeok Park



2023 Fall

- Mutation makes it possible to update the **contents** of a **data structure** or a **variable** after its creation.
  - BFAE – FAE with **mutable boxes**
  - MFAE – FAE with **mutable variables**
  - Evaluation with **memories**, finite maps from addresses to values:

$$\begin{array}{ll} \text{Memories} & M \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V} \\ \text{Addresses} & a \in \mathbb{A} \end{array}$$

- Mutation makes it possible to update the **contents** of a **data structure** or a **variable** after its creation.
  - BFAE – FAE with **mutable boxes**
  - MFAE – FAE with **mutable variables**
  - Evaluation with **memories**, finite maps from addresses to values:

$$\begin{array}{ll} \text{Memories} & M \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V} \\ \text{Addresses} & a \in \mathbb{A} \end{array}$$

- In this lecture, we will learn **memory management** techniques to **deallocate** unreachable memory cells:
  - **Stack and Heap**
  - **Manual Memory Management**
  - **Garbage Collection (GC)**

## 1. Stack and Heap

Tail-Call Optimization (TCO)

## 2. Manual Memory Management

## 3. Garbage Collection

Reference Counting

Mark-and-Sweep GC

Copying GC (Two-Space GC)

Other GC Algorithms

## 1. Stack and Heap

Tail-Call Optimization (TCO)

## 2. Manual Memory Management

## 3. Garbage Collection

Reference Counting

Mark-and-Sweep GC

Copying GC (Two-Space GC)

Other GC Algorithms

In the previous lecture, we have seen the memory in the following MFAE expression has **unreachable** memory cells as follows:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y;
  x * x
};
f(5);           /* 36 */
y = 3;
f(5);           /* 64 */
    
```

$$\sigma = [$$

$$y \mapsto a_0$$

$$f \mapsto a_1$$

$$]$$

$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|c|} \hline 3 & v & 6 & 8 & \dots \\ \hline \end{array} \end{array}$$

where  $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

In the previous lecture, we have seen the memory in the following MFAE expression has **unreachable** memory cells as follows:

```
/* MFAE */
var y = 1;
var f = x => {
  x = x + y;
  x * x
};
f(5);           /* 36 */
y = 3;
f(5);           /* 64 */
```

$$\sigma = [$$

$$y \mapsto a_0$$

$$f \mapsto a_1$$

$$]$$

$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|c|} \hline 3 & v & 6 & 8 & \dots \\ \hline \end{array} \end{array}$$

where  $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

Then, how to **detect** and **dealloc** unreachable memory cells?

In the previous lecture, we have seen the memory in the following MFAE expression has **unreachable** memory cells as follows:

```
/* MFAE */
var y = 1;
var f = x => {
  x = x + y;
  x * x
};
f(5);           /* 36 */
y = 3;
f(5);           /* 64 */
```

$$\sigma = [$$

$$y \mapsto a_0$$

$$f \mapsto a_1$$

$$]$$

$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|c|} \hline 3 & v & 6 & 8 & \dots \\ \hline \end{array} \end{array}$$

where  $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

Then, how to **detect** and **deallocate** unreachable memory cells?

Let's delete **unreachable** memory cells when the program **exits** functions!

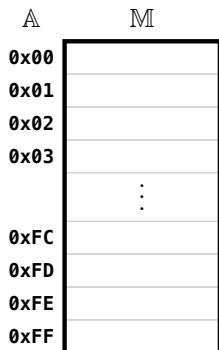


We can **divide** the memory into two parts:

- **Stack** – for **local variables** and **function parameters**
- **Heap** – for **dynamically allocated** memory cells

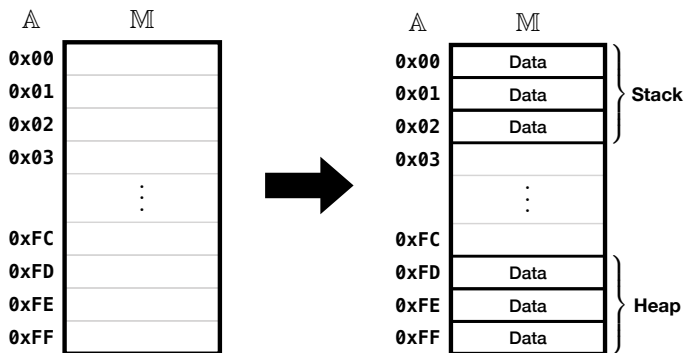
We can **divide** the memory into two parts:

- **Stack** – for **local variables** and **function parameters**
- **Heap** – for **dynamically allocated** memory cells



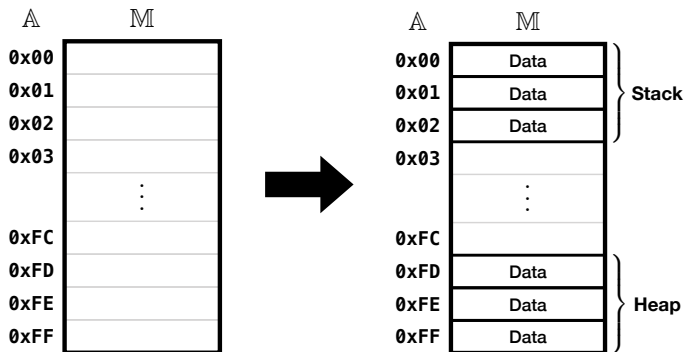
We can **divide** the memory into two parts:

- **Stack** – for **local variables** and **function parameters**
- **Heap** – for **dynamically allocated** memory cells



We can **divide** the memory into two parts:

- **Stack** – for **local variables** and **function parameters**
- **Heap** – for **dynamically allocated** memory cells



Create a **new stack frame** when the program **enters** a function, and **delete** the stack frame when it **exits** the function.

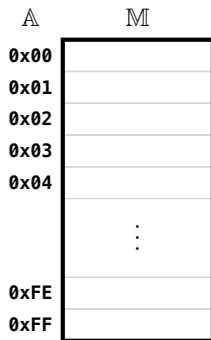
For example, consider the following Scala program:

```
case class Box(var k: Int)

def f(x: Int): Int =
  var y = Box(1)
  var z = g(2)
  x + y.k + z

def g(b: Int): Int =
  var c = Box(b)
  c.k + 3

var a = 1
var d = f(42)
a + d
```



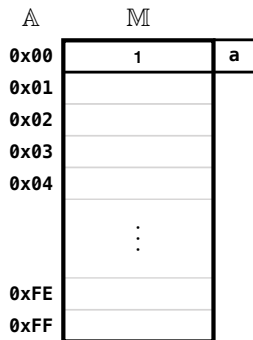
For example, consider the following Scala program:

```
case class Box(var k: Int)

def f(x: Int): Int =
  var y = Box(1)
  var z = g(2)
  x + y.k + z

def g(b: Int): Int =
  var c = Box(b)
  c.k + 3

var a = 1                                /* a -> 0x00 */
var d = f(42)
a + d
```



For example, consider the following Scala program:

```
case class Box(var k: Int)

def f(x: Int): Int = /* x -> 0x01 */
  var y = Box(1)      /* y -> 0x02 */ *
  var z = g(2)
  x + y.k + z

def g(b: Int): Int =
  var c = Box(b)
  c.k + 3

var a = 1              /* a -> 0x00 */
var d = f(42)
a + d
```

A	M	
0x00	1	a
0x01	42	x
0x02	0xFF	y
0x03		
0x04		
	⋮	
0xFE		
0xFF	1	

f

A **new stack frame** is created when it enters the function `f`.

For example, consider the following Scala program:

```
case class Box(var k: Int)

def f(x: Int): Int = /* x -> 0x01 */
  var y = Box(1)      /* y -> 0x02 */
  var z = g(2)
  x + y.k + z

def g(b: Int): Int = /* b -> 0x03 */
  var c = Box(b)      /* c -> 0x04 */
  c.k + 3              /* 5 */

var a = 1              /* a -> 0x00 */
var d = f(42)
a + d
```

A	M	
0x00	1	a
0x01	42	x
0x02	0xFF	y
0x03	2	b
0x04	0xFE	c
	:	
	:	
0xFE	2	
0xFF	1	

A **new stack frame** is created when it enters the function `g`.



For example, consider the following Scala program:

```
case class Box(var k: Int)

def f(x: Int): Int = /* x -> 0x01 */
    var y = Box(1)    /* y -> 0x02 */
    var z = g(2)      /* z -> 0x03 */
    x + y.k + z       /* 48 */ *

def g(b: Int): Int =
    var c = Box(b)
    c.k + 3

var a = 1             /* a -> 0x00 */
var d = f(42)
a + d
```

A	M	
0x00	1	a
0x01	42	x
0x02	0xFF	y
0x03	5	z
0x04	⋮	
0xFE	2	
0xFF	1	

After exiting the function `g`, its stack frame is **deleted**. The memory cells allocated for `b` and `c` in the stack frame are **deallocated**.

For example, consider the following Scala program:

```
case class Box(var k: Int)

def f(x: Int): Int =
  var y = Box(1)
  var z = g(2)
  x + y.k + z

def g(b: Int): Int =
  var c = Box(b)
  c.k + 3

var a = 1           /* a -> 0x00 */
var d = f(42)       /* d -> 0x01 */
a + d               /* 49 */
```

A	M	
0x00	1	a
0x01	48	d
0x02		
0x03		
0x04		
	:	
	:	
0xFE	2	
0xFF	1	

After exiting the function `f`, its stack frame is **deleted**. The memory cells allocated for `x`, `y`, and `z` in the stack frame are **deallocated**.

# Tail-Call Optimization (TCO)

Here is another example with a recursive function:

```
def sum(x: Int, acc: Int): Int =  
  if (x < 1) acc  
  else sum(x - 1, x + acc) // tail-call  
  
sum(1000, 0)
```

A	M		
0x00	1000	x	sum
0x01	0	acc	
0x02			
0x03			
0x04			
0x05			
	⋮		
0xFE			
0xFF			

# Tail-Call Optimization (TCO)

Here is another example with a recursive function:

```
def sum(x: Int, acc: Int): Int =  
  if (x < 1) acc  
  else sum(x - 1, x + acc) // tail-call  
  
sum(1000, 0)
```

A	M		
0x00	1000	x	sum
0x01	0	acc	
0x02	999	x	sum
0x03	1000	acc	
0x04			
0x05			
	⋮		
0xFE			
0xFF			

# Tail-Call Optimization (TCO)

Here is another example with a recursive function:

```
def sum(x: Int, acc: Int): Int =  
  if (x < 1) acc  
  else sum(x - 1, x + acc) // tail-call  
  
sum(1000, 0)
```

A	M		
0x00	1000	x	sum
0x01	0	acc	
0x02	999	x	sum
0x03	1000	acc	
0x04	998	x	sum
0x05	1999	acc	
	:	:	
0xFE	873	x	sum
0xFF	118999	acc	

It fails with a **stack overflow** error.

# Tail-Call Optimization (TCO)

Here is another example with a recursive function:

```
def sum(x: Int, acc: Int): Int =  
  if (x < 1) acc  
  else sum(x - 1, x + acc) // tail-call  
  
sum(1000, 0)
```

A	M		
0x00	1000	x	sum
0x01	0	acc	
0x02	999	x	sum
0x03	1000	acc	
0x04	998	x	sum
0x05	1999	acc	
	:	:	
0xFE	873	x	sum
0xFF	118999	acc	

It fails with a **stack overflow** error.

However, is it really necessary to keep **all the stack frames**?

# Tail-Call Optimization (TCO)

Here is another example with a recursive function:

```
def sum(x: Int, acc: Int): Int =  
  if (x < 1) acc  
  else sum(x - 1, x + acc) // tail-call  
  
sum(1000, 0)
```

A	M		
0x00	1000	x	sum
0x01	0	acc	
0x02	999	x	sum
0x03	1000	acc	
0x04	998	x	sum
0x05	1999	acc	
	:	:	
0xFE	873	x	sum
0xFF	118999	acc	

It fails with a **stack overflow** error.

However, is it really necessary to keep **all the stack frames**? **No!**

Scala supports **tail-call optimization** (TCO).

# Tail-Call Optimization (TCO)

Here is another example with a recursive function:

```
def sum(x: Int, acc: Int): Int =  
  if (x < 1) acc  
  else sum(x - 1, x + acc) // tail-call  
  
sum(1000, 0)
```

A	M		
0x00	1000	x	sum
0x01	0	acc	
0x02			
0x03			
0x04			
0x05			
	⋮		
0xFE			
0xFF			

Why? the function call is in **tail-call position** (i.e., the final action in the function). It means that it directly returns the result without any further computation.

Thus, we can safely **discard** the current stack frame **before** calling the function, and it is called **tail-call optimization** (TCO).



# Tail-Call Optimization (TCO)

Here is another example with a recursive function:

```
def sum(x: Int, acc: Int): Int =  
  if (x < 1) acc  
  else sum(x - 1, x + acc) // tail-call  
  
sum(1000, 0)
```

A	M		
0x00	999	x	sum
0x01	1000	acc	
0x02			
0x03			
0x04			
0x05			
	⋮		
0xFE			
0xFF			

Why? the function call is in **tail-call position** (i.e., the final action in the function). It means that it directly returns the result without any further computation.

Thus, we can safely **discard** the current stack frame **before** calling the function, and it is called **tail-call optimization** (TCO).

# Tail-Call Optimization (TCO)

Here is another example with a recursive function:

```
def sum(x: Int, acc: Int): Int =  
  if (x < 1) acc  
  else sum(x - 1, x + acc) // tail-call  
  
sum(1000, 0) // 500500
```

A	M		
0x00	0	x	sum
0x01	500500	acc	
0x02			
0x03			
0x04			
0x05			
	⋮		
0xFE			
0xFF			

Why? the function call is in **tail-call position** (i.e., the final action in the function). It means that it directly returns the result without any further computation.

Thus, we can safely **discard** the current stack frame **before** calling the function, and it is called **tail-call optimization** (TCO).

## 1. Stack and Heap

Tail-Call Optimization (TCO)

## 2. Manual Memory Management

## 3. Garbage Collection

Reference Counting

Mark-and-Sweep GC

Copying GC (Two-Space GC)

Other GC Algorithms

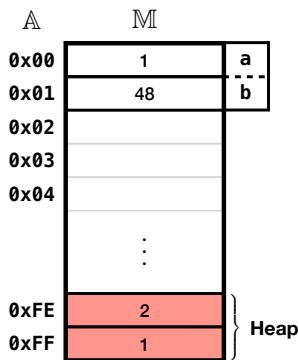
Let's see the previous example again:

```
case class Box(var k: Int)

def f(x: Int): Int =
  var y = Box(1)
  var z = g(2)
  x + y.k + z

def g(b: Int): Int =
  var c = Box(b)
  c.k + 3

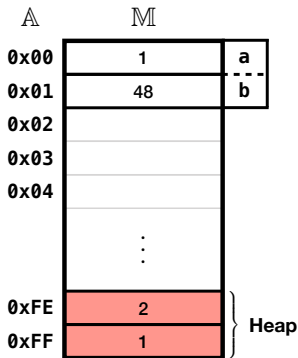
var a = 1           /* a -> 0x00 */
var b = f(42)       /* b -> 0x01 */
a + b               /* 49 */
```



Unfortunately, we still cannot deallocate memory cells (e.g., 0xFE and 0xFF) dynamically allocated in the **heap** rather than the **stack**.

One way to resolve this is using the **manual memory management**, and C++ is an example language that supports it with special keywords for memory allocation (`new`) and deallocation (`delete`) in heap, respectively:

```
struct Box { int k; Box(int k): k(k) {} };
int f(int x) {
    Box* y = new Box(1);    // alloc  0xFF
    int z = g(2);
    int k = y->k;
    return x + k + z;
}
int g(int b) {
    Box* c = new Box(b);    // alloc  0xFE
    int k = c->k;
    return k + 3;
}
int a = 1;    /* a -> 0x00 */
int b = f(42); /* b -> 0x01 */
a + b;        /* 49 */
```



One way to resolve this is using the **manual memory management**, and C++ is an example language that supports it with special keywords for memory allocation (`new`) and deallocation (`delete`) in heap, respectively:

```
struct Box { int k; Box(int k): k(k) {} };
int f(int x) {
    Box* y = new Box(1);    // alloc  0xFF
    int z = g(2);
    int k = y->k; delete y; // dealloc 0xFF
    return x + k + z;
}
int g(int b) {
    Box* c = new Box(b);    // alloc  0xFE
    int k = c->k; delete c; // dealloc 0xFE
    return k + 3;
}
int a = 1;    /* a -> 0x00 */
int b = f(42); /* b -> 0x01 */
a + b;        /* 49 */
```

A	M	
0x00	1	a
0x01	48	b
0x02		
0x03		
0x04		
	⋮	
0xFE		
0xFF		

## Pros:

- **Efficient** – Users can **explicitly** deallocate memory cells allocated in heap whenever they want.

## Cons:

- **Error-prone** – Users have all the **responsibility** to deallocate memory cells allocated in heap:
  - **Memory leak** occurs if users forget to deallocate memory cells.

```
b = new Box(42); ...
```

- **Dangling pointer** occurs if users deallocate memory cells too early.

```
b = new Box(42); ... delete b; ... b->k;
```

- **Double free** occurs if users deallocate memory cells more than once.

```
b = new Box(42); ... delete b; ... delete b;
```

## 1. Stack and Heap

Tail-Call Optimization (TCO)

## 2. Manual Memory Management

## 3. Garbage Collection

Reference Counting

Mark-and-Sweep GC

Copying GC (Two-Space GC)

Other GC Algorithms



Is there any way to **automatically** deallocate memory cells in heap?

Is there any way to **automatically** deallocate memory cells in heap? **Yes!**

**Garbage collection** (GC) is a representative technique for **automatic memory management**.

Let's learn several GC algorithms:

- **Reference counting**
- **Mark-and-sweep GC**
- **Copying GC** (Two-space GC)
- Others

Is there any way to **automatically** deallocate memory cells in heap? **Yes!**

**Garbage collection** (GC) is a representative technique for **automatic memory management**.

Let's learn several GC algorithms:

- **Reference counting**
- **Mark-and-sweep GC**
- **Copying GC** (Two-space GC)
- Others

Before explaining them, let's represent memory cells in heap in a **graphical way** without actual addresses.

From now on, we will use the **graphical representation** of memory cells:

```
case class A(var x: A)

var a = A(A(A(null))) *

var b = A(A(null))

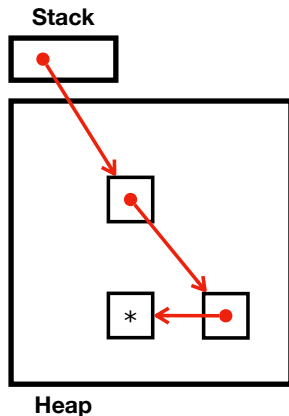
a.x.x.x = a.x

b.x = a

a.x = A(null)

a = a.x

b = null
```



From now on, we will use the **graphical representation** of memory cells:

```
case class A(var x: A)
```

```
var a = A(A(A(null)))
```

```
var b = A(A(null))
```

\*

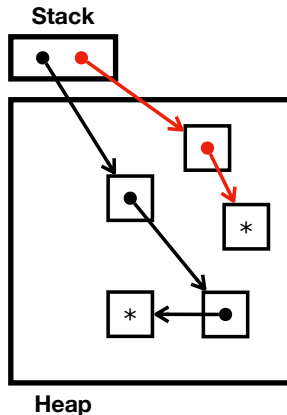
```
a.x.x.x = a.x
```

```
b.x = a
```

```
a.x = A(null)
```

```
a = a.x
```

```
b = null
```



From now on, we will use the **graphical representation** of memory cells:

```
case class A(var x: A)
```

```
var a = A(A(A(null)))
```

```
var b = A(A(null))
```

```
a.x.x.x = a.x
```

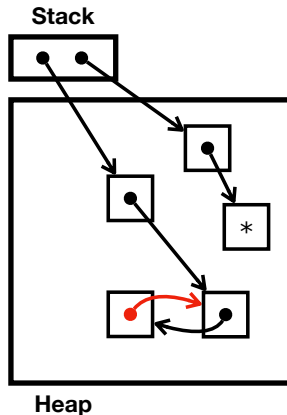
\*

```
b.x = a
```

```
a.x = A(null)
```

```
a = a.x
```

```
b = null
```



From now on, we will use the **graphical representation** of memory cells:

```
case class A(var x: A)
```

```
var a = A(A(A(null)))
```

```
var b = A(A(null))
```

```
a.x.x.x = a.x
```

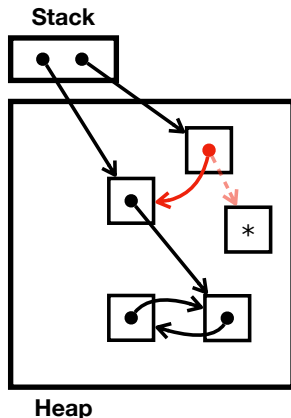
```
b.x = a
```

\*

```
a.x = A(null)
```

```
a = a.x
```

```
b = null
```



From now on, we will use the **graphical representation** of memory cells:

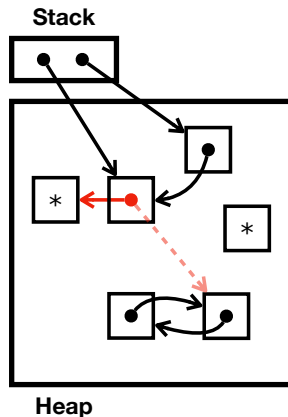
```
case class A(var x: A)
var a = A(A(A(null)))
var b = A(A(null))

a.x.x.x = a.x

b.x = a

a.x = A(null)
a = a.x
b = null
```

\*





From now on, we will use the **graphical representation** of memory cells:

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

a.x.x.x = a.x

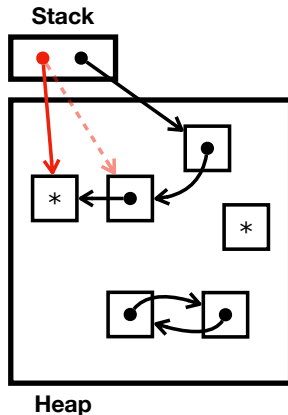
b.x = a

a.x = A(null)

a = a.x

b = null
```

\*

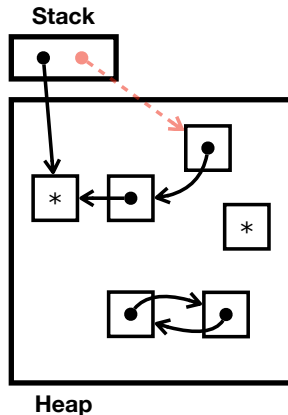


From now on, we will use the **graphical representation** of memory cells:

```
case class A(var x: A)
var a = A(A(A(null)))
var b = A(A(null))

a.x.x.x = a.x
b.x = a
a.x = A(null)
a = a.x
b = null
```

\*



From now on, we will use the **graphical representation** of memory cells:

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

a.x.x.x = a.x

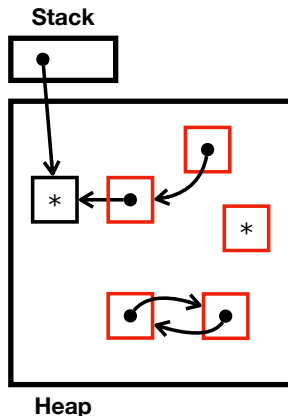
b.x = a

a.x = A(null)

a = a.x

b = null
```

\*



We need to deallocate five unreachable memory cells in heap.

**Reference counting** is a simple GC algorithm that keeps track of the **number of references** to each memory cell in heap.

- 1 **Initialize** the reference count of each cell to 0.
- 2 When a reference to a cell is created, **increment** its reference count.
- 3 When a reference to a cell is deleted, **decrement** its reference count.
- 4 When the reference count of a cell reaches 0, **deallocate** the cell.

Many programming languages use reference counting to implement GC:

- Python, Swift, Perl, Objective-C, etc.

**Reference counting** is a simple GC algorithm that keeps track of the **number of references** to each memory cell in heap.

```
case class A(var x: A)

var a = A(A(A(null))) *

var b = A(A(null))

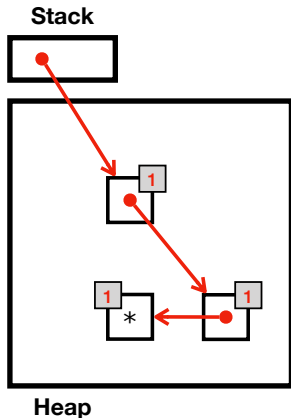
a.x.x.x = a.x

b.x = a

a.x = A(null)

a = a.x

b = null
```



**Reference counting** is a simple GC algorithm that keeps track of the **number of references** to each memory cell in heap.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

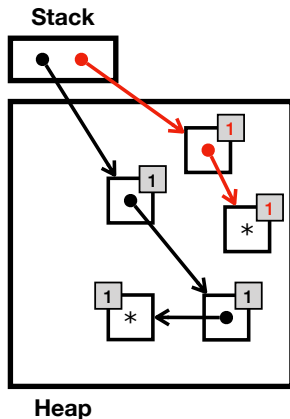
a.x.x.x = a.x

b.x = a

a.x = A(null)

a = a.x

b = null
```



**Reference counting** is a simple GC algorithm that keeps track of the **number of references** to each memory cell in heap.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

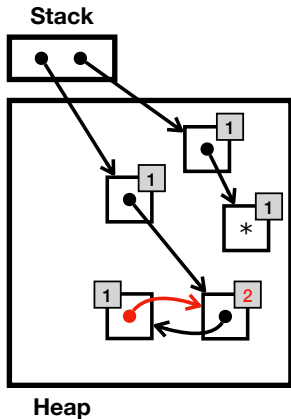
a.x.x.x = a.x      *

b.x = a

a.x = A(null)

a = a.x

b = null
```



**Reference counting** is a simple GC algorithm that keeps track of the **number of references** to each memory cell in heap.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

a.x.x.x = a.x

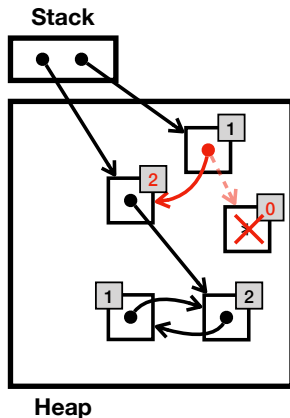
b.x = a

a.x = A(null)

a = a.x

b = null
```

\*





**Reference counting** is a simple GC algorithm that keeps track of the **number of references** to each memory cell in heap.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

a.x.x.x = a.x

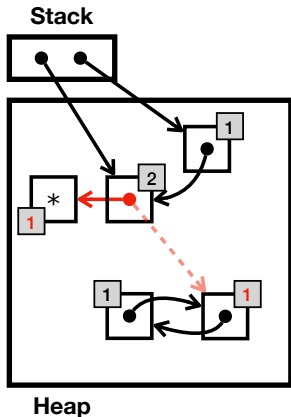
b.x = a

a.x = A(null)

a = a.x

b = null
```

\*



**Reference counting** is a simple GC algorithm that keeps track of the **number of references** to each memory cell in heap.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

a.x.x.x = a.x

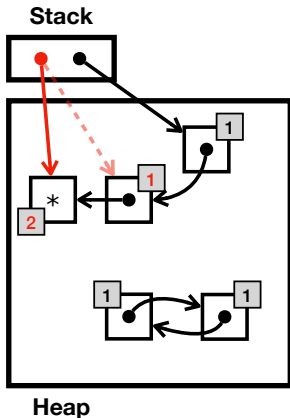
b.x = a

a.x = A(null)

a = a.x

b = null
```

\*



**Reference counting** is a simple GC algorithm that keeps track of the **number of references** to each memory cell in heap.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

a.x.x.x = a.x

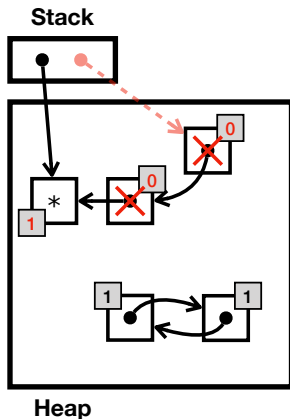
b.x = a

a.x = A(null)

a = a.x

b = null
```

\*



**Reference counting** is a simple GC algorithm that keeps track of the **number of references** to each memory cell in heap.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

a.x.x.x = a.x

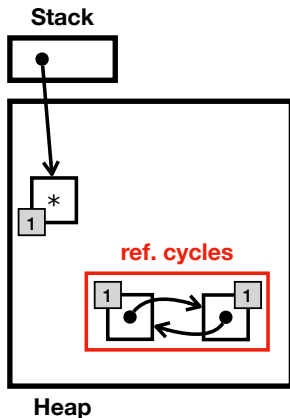
b.x = a

a.x = A(null)

a = a.x

b = null
```

\*



Unfortunately, we cannot deallocate unreachable **reference cycles**.

**Reference counting** is a simple GC algorithm that keeps track of the **number of references** to each memory cell in heap.

## Pros:

- **Easy to implement** – Simply increment and decrement the reference count when a reference is created and deleted.
- **Low overhead** – Deallocation is immediate and takes a short time.

## Cons:

- **Reference cycles** – It cannot deallocate unreachable reference cycles.
- **Reference count cost** – It requires space to store reference counts.
- **Free List and Fragmentation** – It requires a **free list** to keep track of available free memory cells in heap, and it also suffers from **fragmentation** making it difficult to allocate large objects.

**Mark-and-Sweep GC** is one of **tracing GC** algorithms that **traverses the heap** to find unreachable objects when it is **triggered** under some conditions.

- ① **Mark** all memory cells as **unreachable** (white).
- ② **Mark** all memory cells referenced by roots as **unscanned** (gray).
- ③ Repeat until there are no unscanned (gray) memory cells:
  - ① **Pick** an unscanned (gray) memory cell.
  - ② **Mark** memory cells referenced by the picked one as **unscanned** (gray).
  - ③ **Mark** the picked memory cell as **scanned** (black).
- ④ **Deallocate** (sweep) all memory cells that are still marked as **unreachable** (white).

Assume that the GC is triggered at the following program point, and let's perform **mark-and-sweep** GC.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

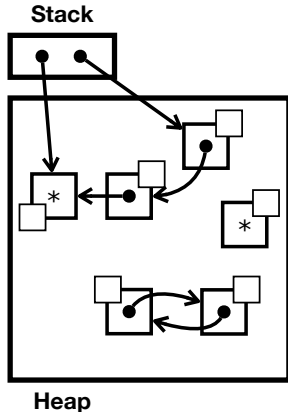
a.x.x.x = a.x

b.x = a

a.x = A(null)

a = a.x

b = null
```



Assume that the GC is triggered at the following program point, and let's perform **mark-and-sweep** GC.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

a.x.x.x = a.x

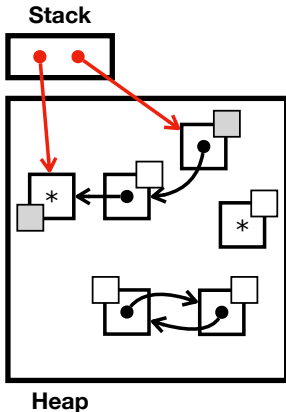
b.x = a

a.x = A(null)

a = a.x

b = null
```

\*





Assume that the GC is triggered at the following program point, and let's perform **mark-and-sweep** GC.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

a.x.x.x = a.x

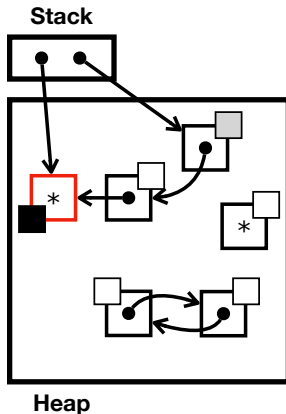
b.x = a

a.x = A(null)

a = a.x

b = null
```

\*



Assume that the GC is triggered at the following program point, and let's perform **mark-and-sweep** GC.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

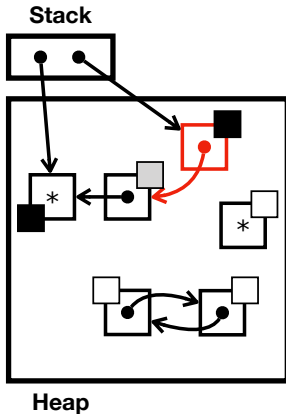
a.x.x.x = a.x

b.x = a

a.x = A(null)

a = a.x

b = null
```



Assume that the GC is triggered at the following program point, and let's perform **mark-and-sweep** GC.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

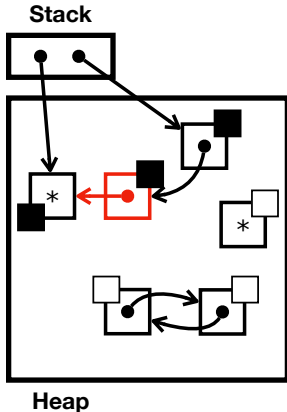
a.x.x.x = a.x

b.x = a

a.x = A(null)

a = a.x

b = null
```



Assume that the GC is triggered at the following program point, and let's perform **mark-and-sweep** GC.

```
case class A(var x: A)

var a = A(A(A(null)))

var b = A(A(null))

a.x.x.x = a.x

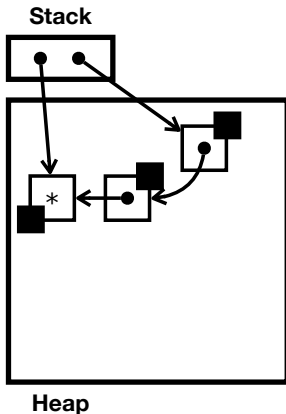
b.x = a

a.x = A(null)

a = a.x

b = null
```

\*



**Mark-and-Sweep GC** is one of **tracing GC** algorithms that **traverses the heap** to find unreachable objects when it is **triggered** under some conditions.

## Pros:

- **Reference cycles** – It can deallocate unreachable reference cycles.

## Cons:

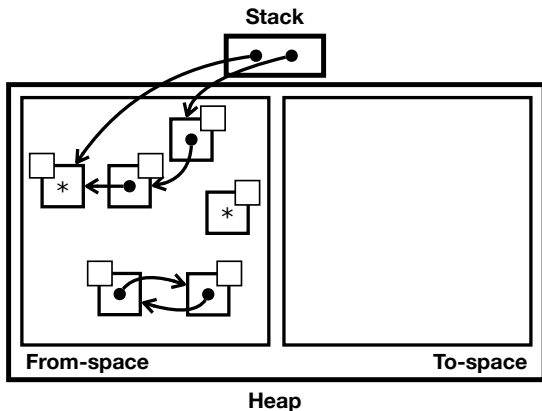
- **Stop-the-world** – It **stops** the program execution during GC.
- **Free List** and **Fragmentation** – It requires a **free list** to keep track of available free memory cells in heap, and it also suffers from **fragmentation** making it difficult to allocate large objects.

Similar to mark-and-sweep GC, **copying GC** (Two-space GC) is another **tracing GC** algorithm. However, it **copies** all the reachable objects and reorganizes them in a **compact** layout by splitting the heap into two spaces: **from-space** and **to-space**.

- **Allocation** – It allocates memory cells only in **from-space**.
- **Deallocation** – It deallocates all the unreachable objects as follows:
  - ① **Mark** all memory cells as **unreachable** (white).
  - ② **Copy** all memory cells referenced by roots as **unscanned** (gray) and copy them from the **from-space** to the **to-space**
  - ③ **Update** the data of the original memory cell to point to the copied one.
  - ④ Repeat until there are no unscanned (gray) memory cells
    - ① **Pick** an unscanned (gray) memory cell in the **from-space**.
    - ② **Copy** memory cells referenced by the picked one as **unscanned** (gray).
    - ③ **Update** the data of the original memory cell to point to the copied one.
    - ④ **Mark** the picked memory cell as **scanned** (black).
  - ⑤ **Swap** from-space and to-space.

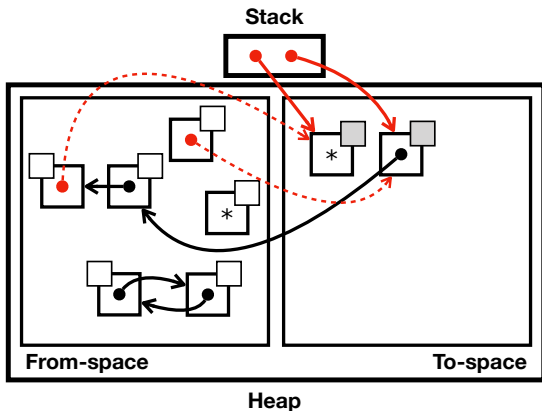
# Copying GC (Two-Space GC)

Similar to mark-and-sweep GC, **copying GC** (Two-space GC) is another **tracing GC** algorithm. However, it **copies** all the reachable objects and reorganizes them in a **compact** layout by splitting the heap into two spaces: **from-space** and **to-space**.



# Copying GC (Two-Space GC)

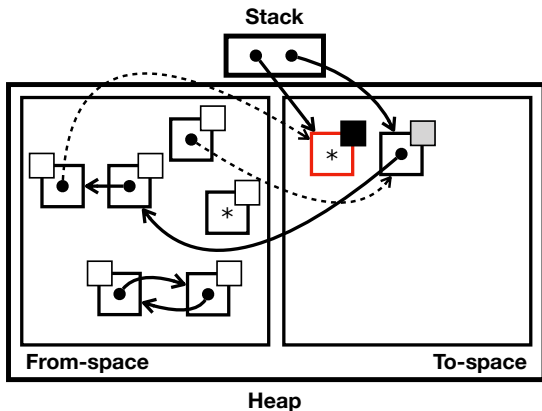
Similar to mark-and-sweep GC, **copying GC** (Two-space GC) is another **tracing GC** algorithm. However, it **copies** all the reachable objects and reorganizes them in a **compact** layout by splitting the heap into two spaces: **from-space** and **to-space**.





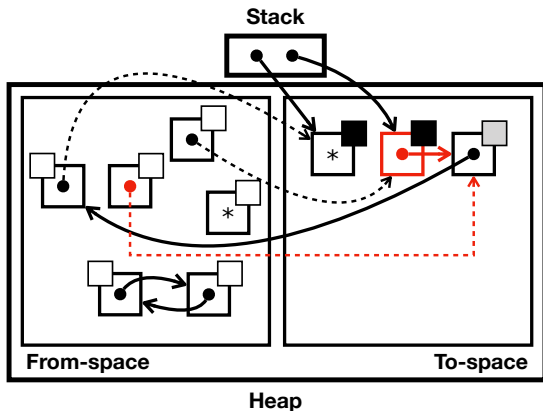
# Copying GC (Two-Space GC)

Similar to mark-and-sweep GC, **copying GC** (Two-space GC) is another **tracing GC** algorithm. However, it **copies** all the reachable objects and reorganizes them in a **compact** layout by splitting the heap into two spaces: **from-space** and **to-space**.



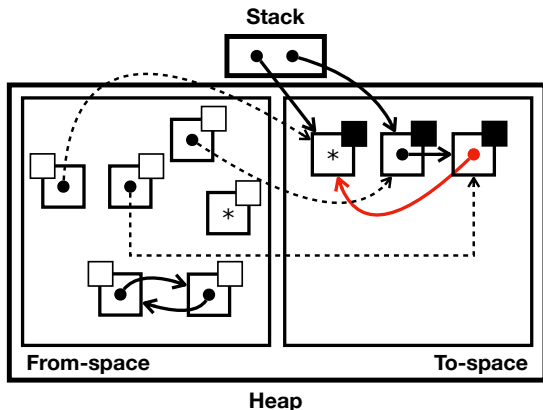
# Copying GC (Two-Space GC)

Similar to mark-and-sweep GC, **copying GC** (Two-space GC) is another **tracing GC** algorithm. However, it **copies** all the reachable objects and reorganizes them in a **compact** layout by splitting the heap into two spaces: **from-space** and **to-space**.



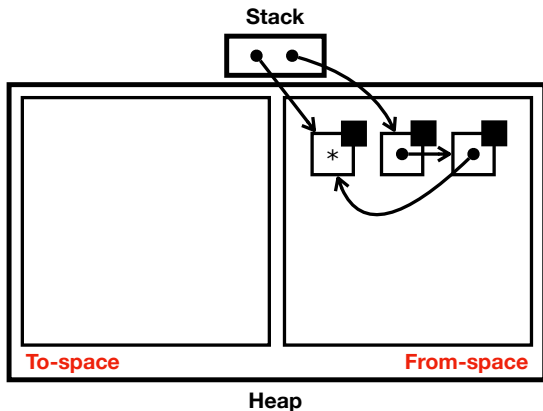
# Copying GC (Two-Space GC)

Similar to mark-and-sweep GC, **copying GC** (Two-space GC) is another **tracing GC** algorithm. However, it **copies** all the reachable objects and reorganizes them in a **compact** layout by splitting the heap into two spaces: **from-space** and **to-space**.



# Copying GC (Two-Space GC)

Similar to mark-and-sweep GC, **copying GC** (Two-space GC) is another **tracing GC** algorithm. However, it **copies** all the reachable objects and reorganizes them in a **compact** layout by splitting the heap into two spaces: **from-space** and **to-space**.



Similar to mark-and-sweep GC, **copying GC** (Two-space GC) is another **tracing GC** algorithm. However, it **copies** all the reachable objects and reorganizes them in a **compact** layout by splitting the heap into two spaces: **from-space** and **to-space**.

## Pros:

- **Reference cycles** – It can deallocate unreachable reference cycles.
- No more **Free List** and **Fragmentation** – After deallocation process, the heap is always **contiguous**. Thus, it is enough to keep track of the first free memory cell for allocation.
- **Fast Allocation** – It does not require any extra work to find free memory cells in the free list for allocation.

## Cons:

- **Stop-the-world** – It **stops** the program execution during GC.
- **Only half of the heap** (from-space) is used for allocation.
- **Expensive copying process** – It copies all the reachable objects from the from-space to the to-space.

Existing real-world programming languages utilize more sophisticated GC algorithms, mix diverse GC algorithms, or even provide options to choose different GC algorithms:

- **Generational GC** – e.g, Java<sup>1</sup>, Python<sup>2</sup>
- **Concurrent GC** – e.g., Java<sup>3</sup>, Golang<sup>4</sup>
- **Ownership, Borrowing, and Lifetimes** – e.g., Rust<sup>5</sup>
- **Escape Analysis** – e.g., Java<sup>6</sup>
- etc.

---

<sup>1</sup><https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

<sup>2</sup><https://devguide.python.org/internals/garbage-collector/>

<sup>3</sup><https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html>

<sup>4</sup><https://tip.golang.org/doc/gc-guide>

<sup>5</sup><https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html>

<sup>6</sup><https://blogs.oracle.com/javamagazine/post/escape-analysis-in-the-hotspot-jit-compiler>

## 1. Stack and Heap

Tail-Call Optimization (TCO)

## 2. Manual Memory Management

## 3. Garbage Collection

Reference Counting

Mark-and-Sweep GC

Copying GC (Two-Space GC)

Other GC Algorithms

- Lazy Evaluation

Jihyeok Park

[jihyeok\\_park@korea.ac.kr](mailto:jihyeok_park@korea.ac.kr)

<https://plrg.korea.ac.kr>