Deep dive into the Eclipse OpenJ9 GC technologies

Charlie Gracie

IBM Senior Software Developer

charlie_gracie@ca.ibm.com

@crgracie

charliegracie

Important Disclaimers

- THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.
- WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.
- ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.
- ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.
- IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM’S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.
- IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.
- NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:
    - CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS

About me

- Senior Software Developer on the IBM Runtime Technologies Team
- Project Lead on Eclipse OMR and a committer on Eclipse OpenJ9
- Has worked on Garbage Collection Technology for over 10 years with a focus on scalability

OpenJ9 GC technologies

- Open J9 Project
- Garbage collection overview
- Open J9 GC technologies
- Questions

OpenJ9

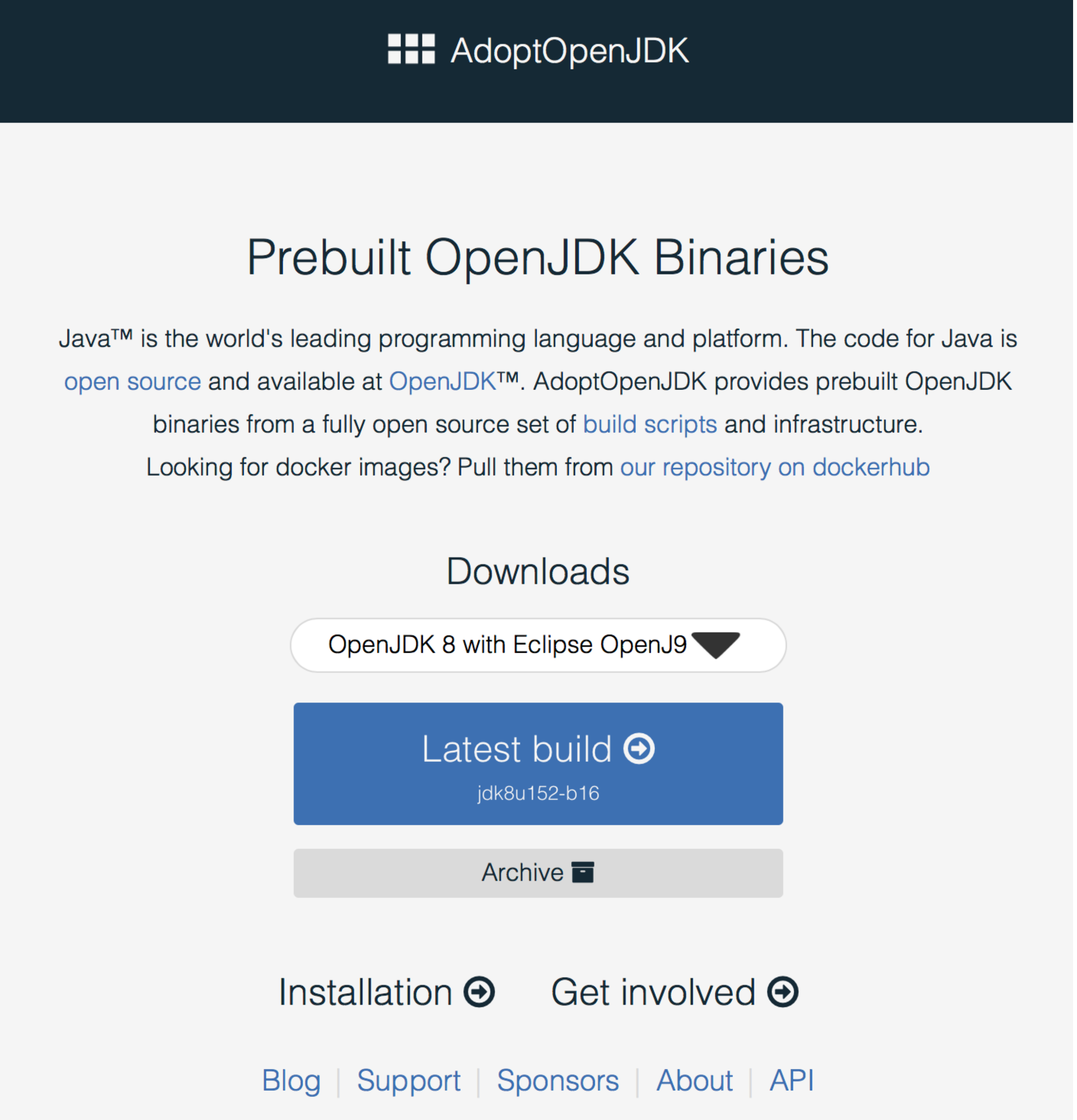
Eclipse OpenJ9
Created Sept 2017

http://www.eclipse.org/openj9
https://github.com/eclipse/openj9

Dual License:
Eclipse Public License v2.0
Apache 2.0

Users and contributors very welcome

https://github.com/eclipse/openj9/blob/master/CONTRIBUTING.md



AdoptOpenJDK

Prebuilt OpenJDK Binaries

Java™ is the world's leading programming language and platform. The code for Java is open source and available at OpenJDK™. AdoptOpenJDK provides prebuilt OpenJDK binaries from a fully open source set of build scripts and infrastructure.

Looking for docker images? Pull them from our repository on dockerhub

Downloads

OpenJDK 8 with Eclipse OpenJ9

The place to get OpenJDK builds

For both:

- OpenJDK with Hotspot
- OpenJDK with Eclipse OpenJ9

https://adoptopenjdk.net/releases.html?variant=openjdk9-openj9

Eclipse OMR
Created March 2016

http://www.eclipse.org/omr
https://github.com/eclipse/omr
https://developer.ibm.com/open/omr/

Dual License:
Eclipse Public License v2.0
Apache 2.0

Users and contributors very welcome

https://github.com/eclipse/omr/blob/master/CONTRIBUTING.md

Garbage Collection

“Garbage Collection (GC) is form automatic memory management. The garbage collector attempts to reclaim memory occupied by objects that are no longer is use by the program.”

Garbage Collection

Positives:

1. Automatic memory management
2. Help reduce certain categories of bugs

Negatives:

1. Requires additional resources
2. Can cause application pauses
3. May introduce runtime costs
4. Application has little control of when memory is reclaimed

OpenJ9 GC goals

1. Implement re-usable technology
2. Provide highly scalable GC technology
3. Favor smaller memory consumption

Garbage collection policies

- -Xgcpolicy:
  1. optthruput – stop the world parallel collector

Garbage collection policies

- -Xgcpolicy:
  1. optthruput – stop the world parallel collector
  2. optavgpause – concurrent collector

Garbage collection policies

- -Xgcpolicy:
  1. optthruput – stop the world parallel collector
  2. optavgpause – concurrent collector
  3. gencon – generational copying collector

Garbage collection policies

- -Xgcpolicy:
  1. optthruput – stop the world parallel collector
  2. optavgpause – concurrent collector
  3. gencon – generational copying collector
  4. balanced – region based generational collector

Garbage collection policies

- -Xgcpolicy:
    1. optthruput – stop the world parallel collector
    2. optavgpause – concurrent collector
    3. gencon – generational copying collector
    4. balanced – region based generational collector
    5. metronome – incremental soft realtime collector

-Xgcpolicy:optthruput

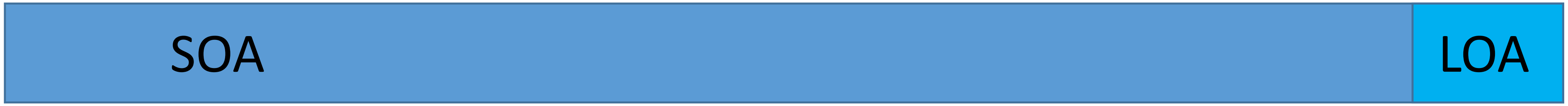
- Parallel global collector
  - GC operations are completed in Stop the world (STW) pauses
  - Mark, sweep and optional compaction collector
  - All GC operations are completed in parallel by multiple helper threads
- GC native memory overhead for mark map and work packets

-Xgcpolicy:optthruput heap

- Flat heap
  - 1 continuous block of memory

-Xgcpolicy:optthruput heap

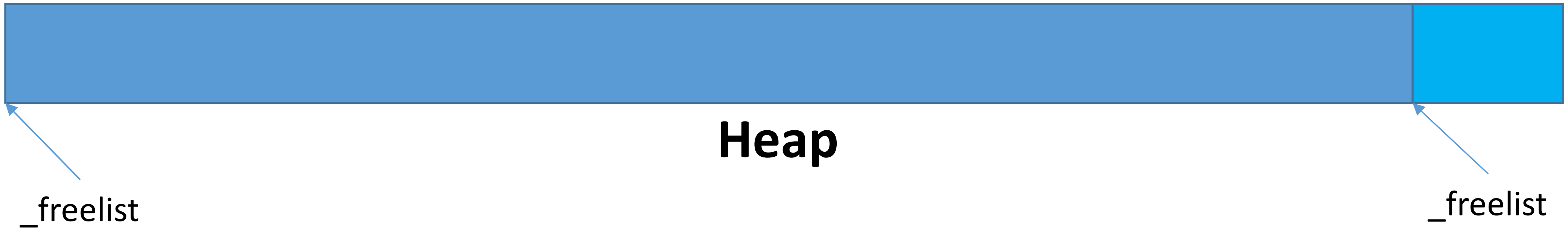
- Flat heap
    - 1 continuous block of memory
- Divided into 2 logical memory areas for allocation
    - SOA – small object area
    - LOA – large object area



Heap

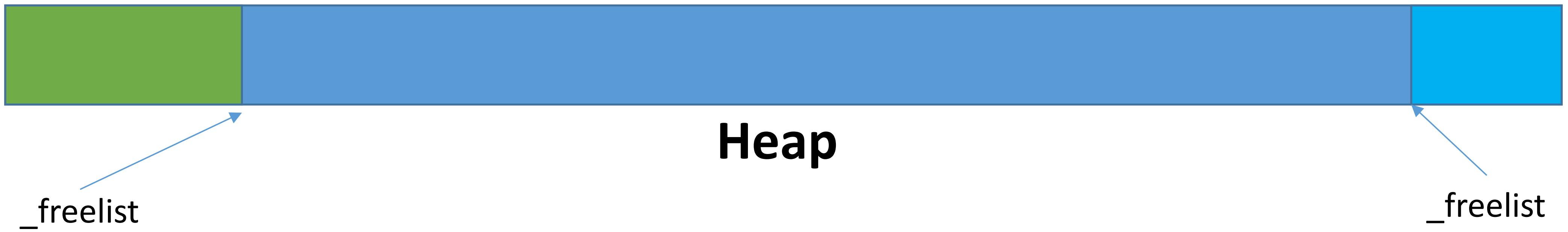
-Xgcpolicy:optthruput heap

- Allocation is a first fit algorithm
  - OpenJ9 uses TLHs to improve allocation performance*



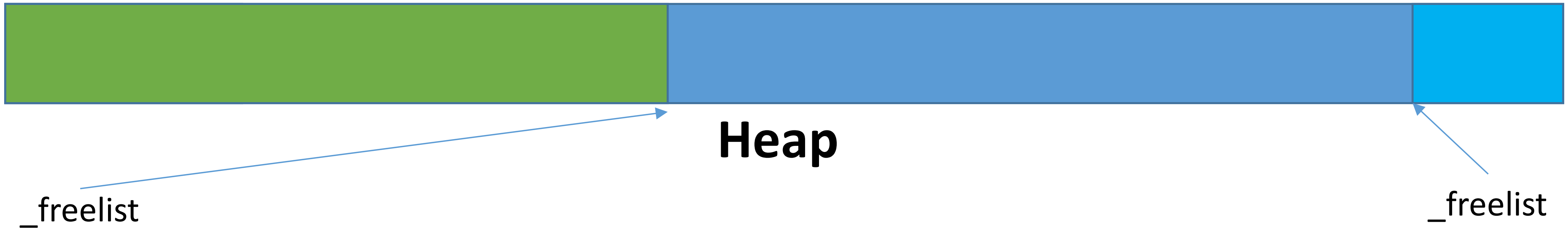
-Xgcpolicy:optthruput heap

- Allocation is a first fit algorithm
  - OpenJ9 uses TLHs to improve allocation performance*



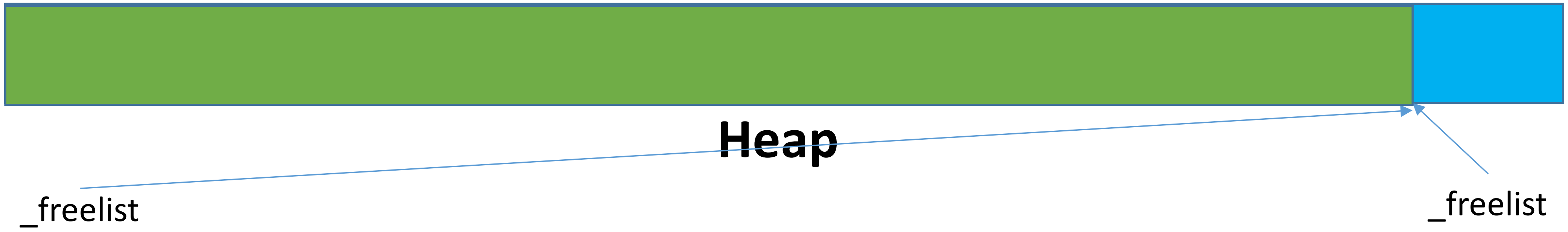
-Xgcpolicy:optthruput heap

- Allocation is a first fit algorithm
  - OpenJ9 uses TLHs to improve allocation performance*



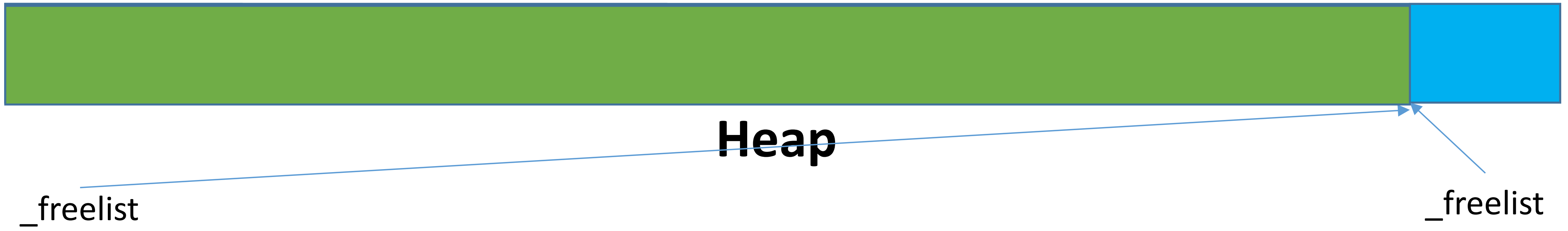
-Xgcpolicy:optthruput heap

- Allocation is a first fit algorithm
  - OpenJ9 uses TLHs to improve allocation performance*
- Allocate until SOA is completely full

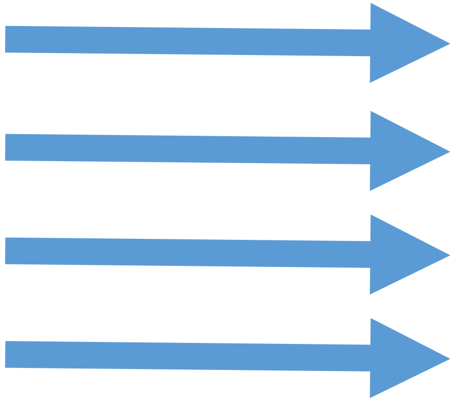


-Xgcpolicy:optthruput heap

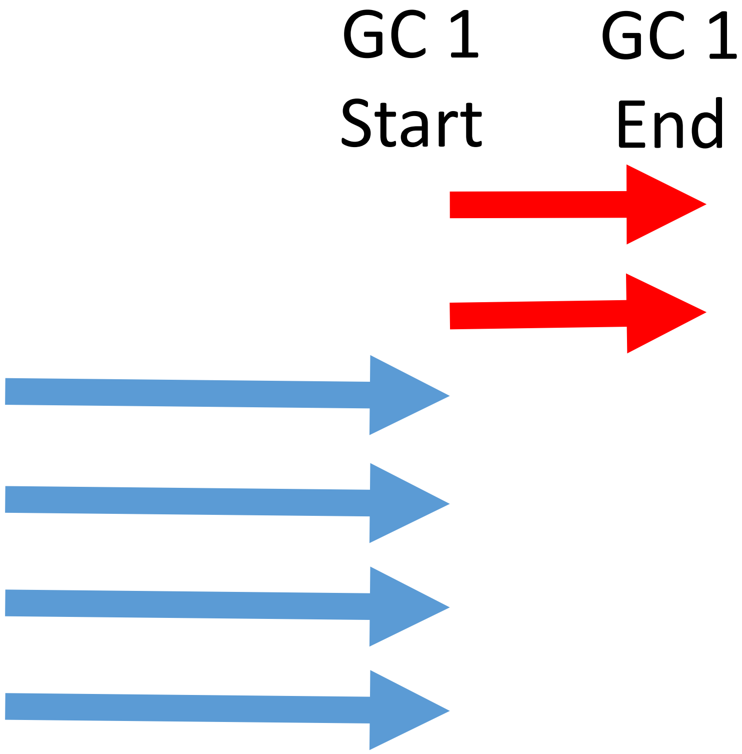
- Allocation is a first fit algorithm
  - OpenJ9 uses TLHs to improve allocation performance*
- Allocate until SOA is completely full
  - Now perform a GC



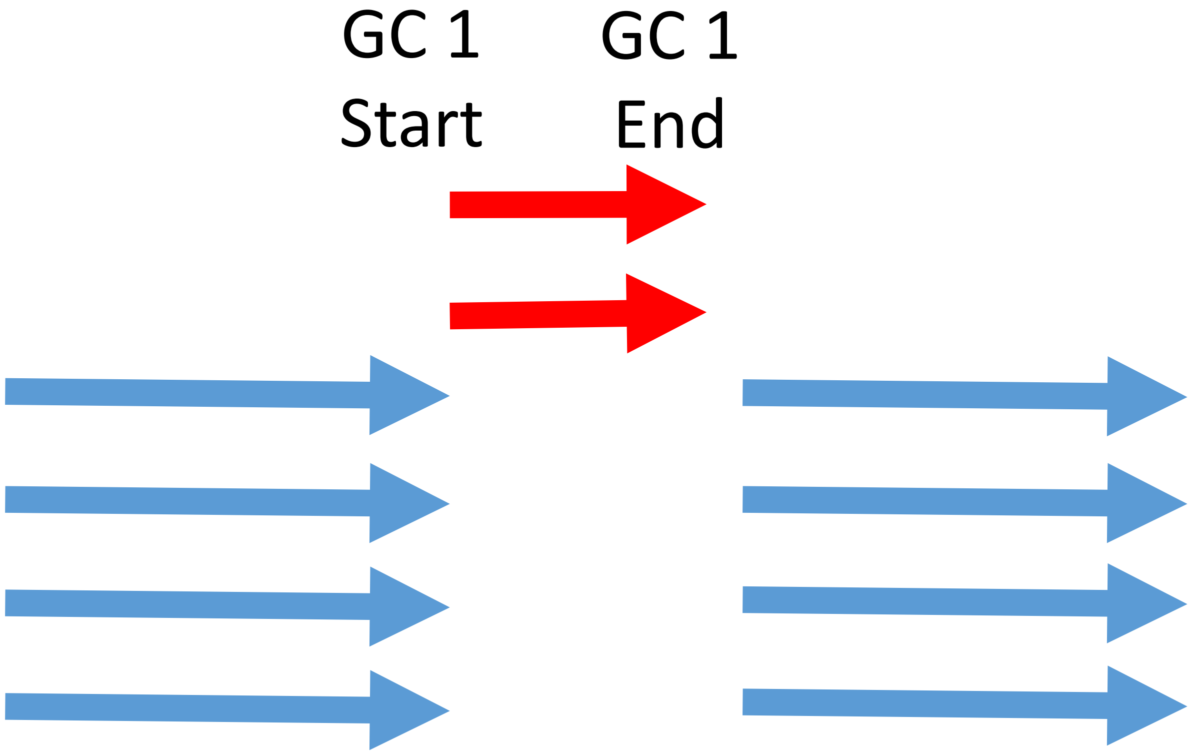
-Xgcpolicy:optthruput GC



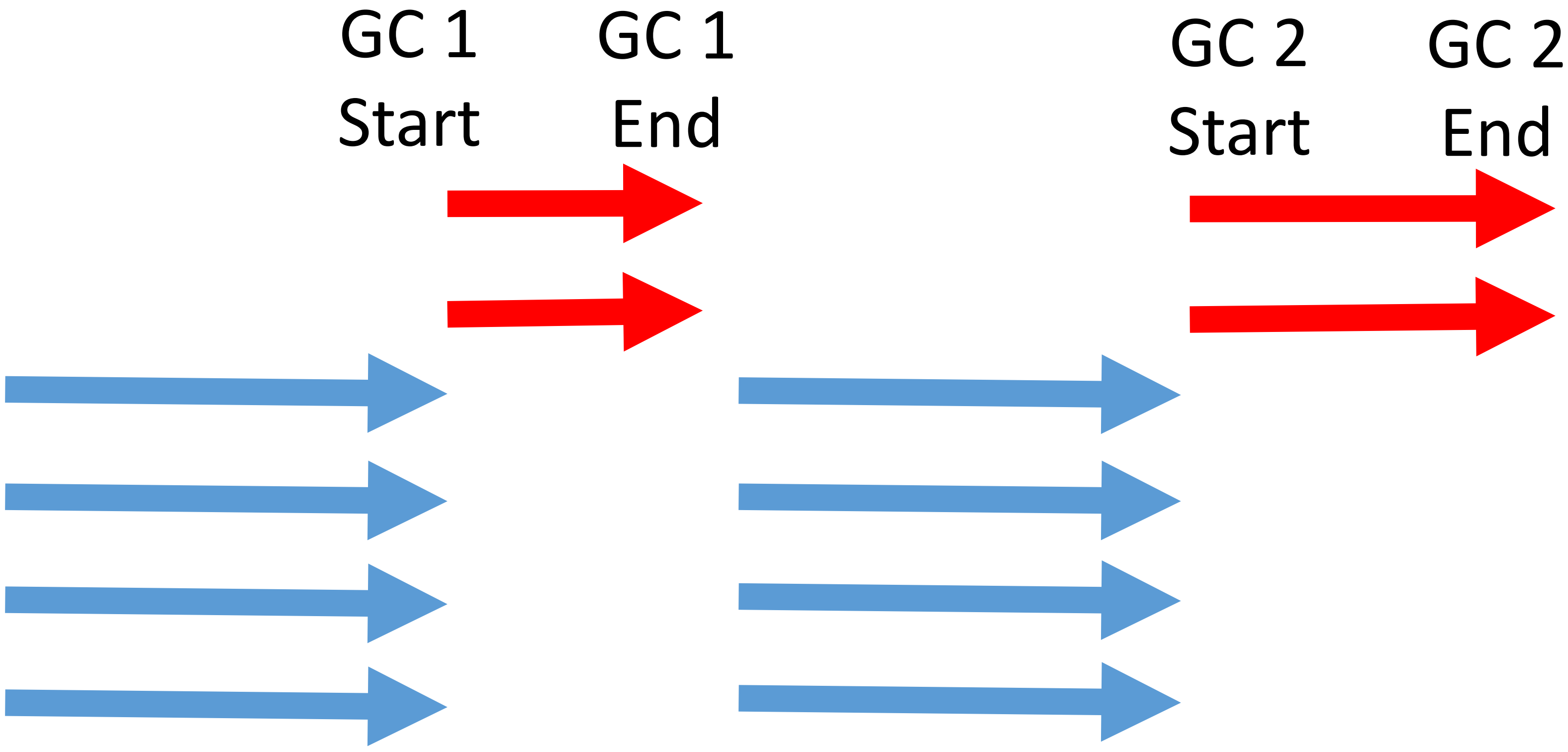
-Xgcpolicy:optthruput GC



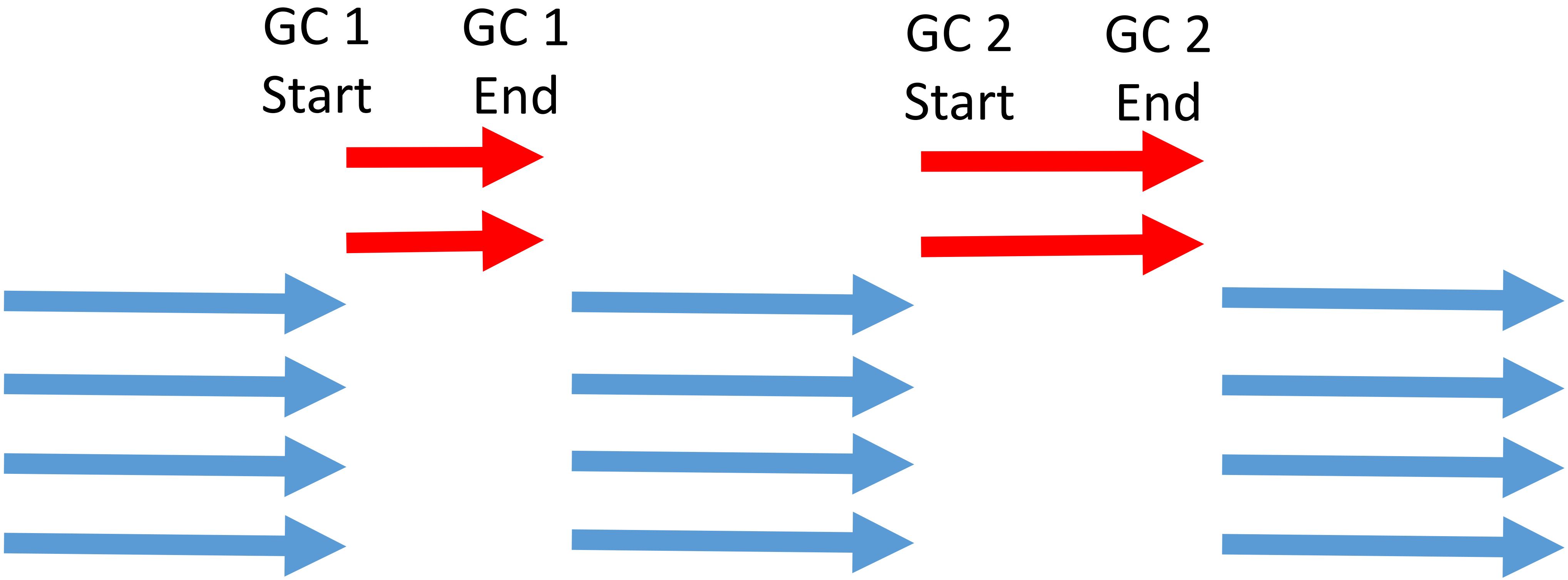
-Xgcpolicy:optthruput GC



-Xgcpolicy:optthruput GC



-Xgcpolicy:optthruput GC

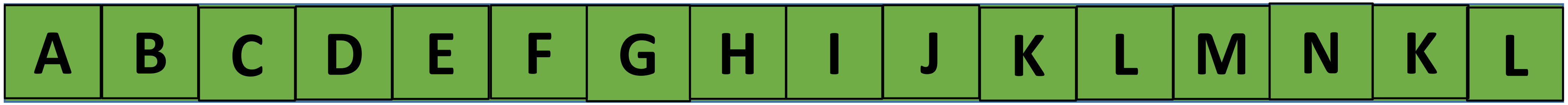


-Xgcpolicy:optthruput GC

Each GC is divided into 3 phases:

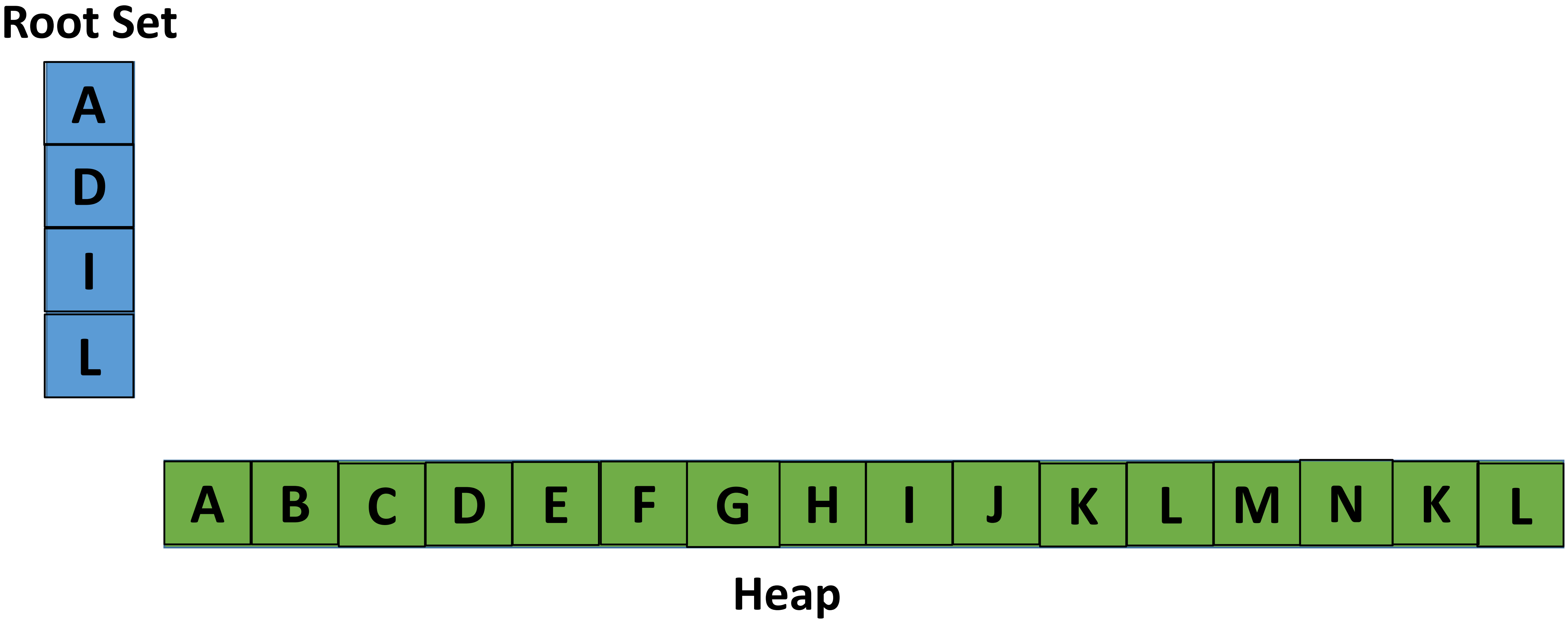
1. Marking – finds all of the live objects
2. Sweeping – reclaims memory for dead objects
3. Compaction – (optional) defragment the heap

-Xgcpolicy:optthruput marking



Heap

-Xgcpolicy:optthruput marking



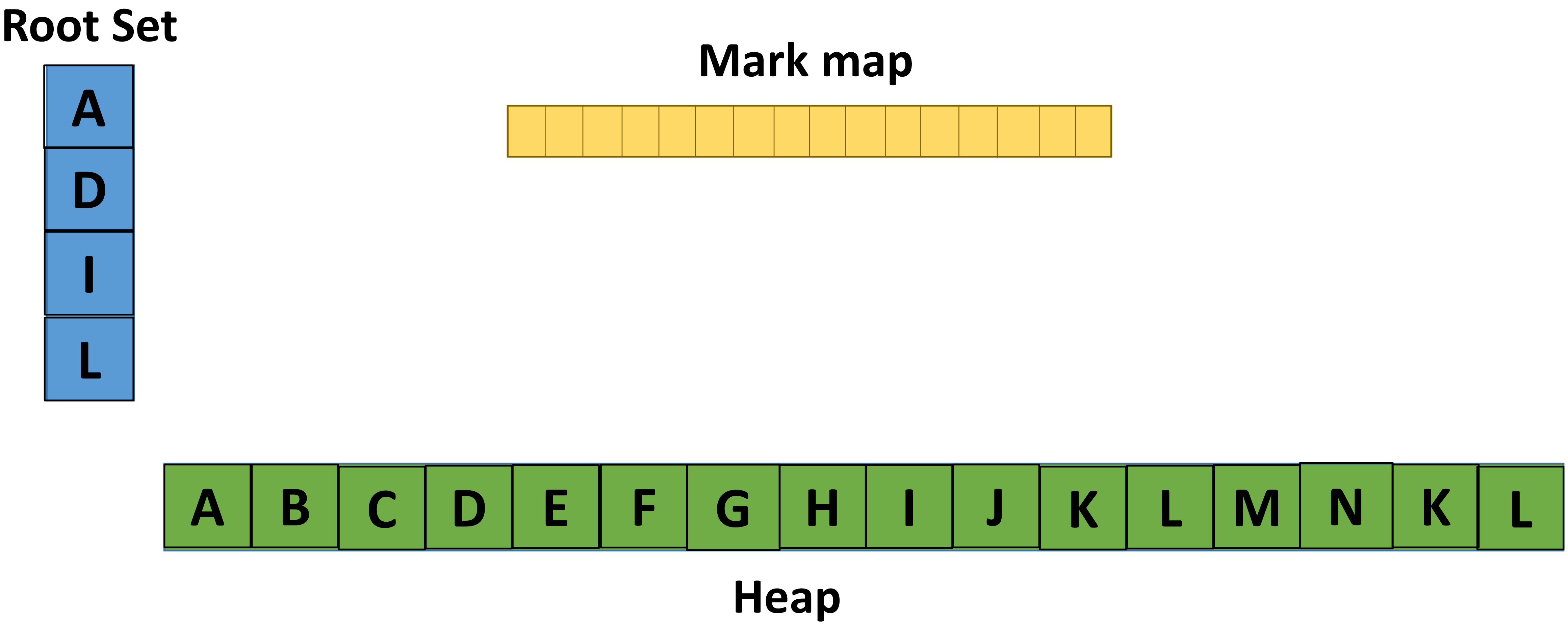
Root Set

A
D
I
L

A B C D E F G H I J K L M N K L

Heap

-Xgcpolicy:optthruput marking



Root Set

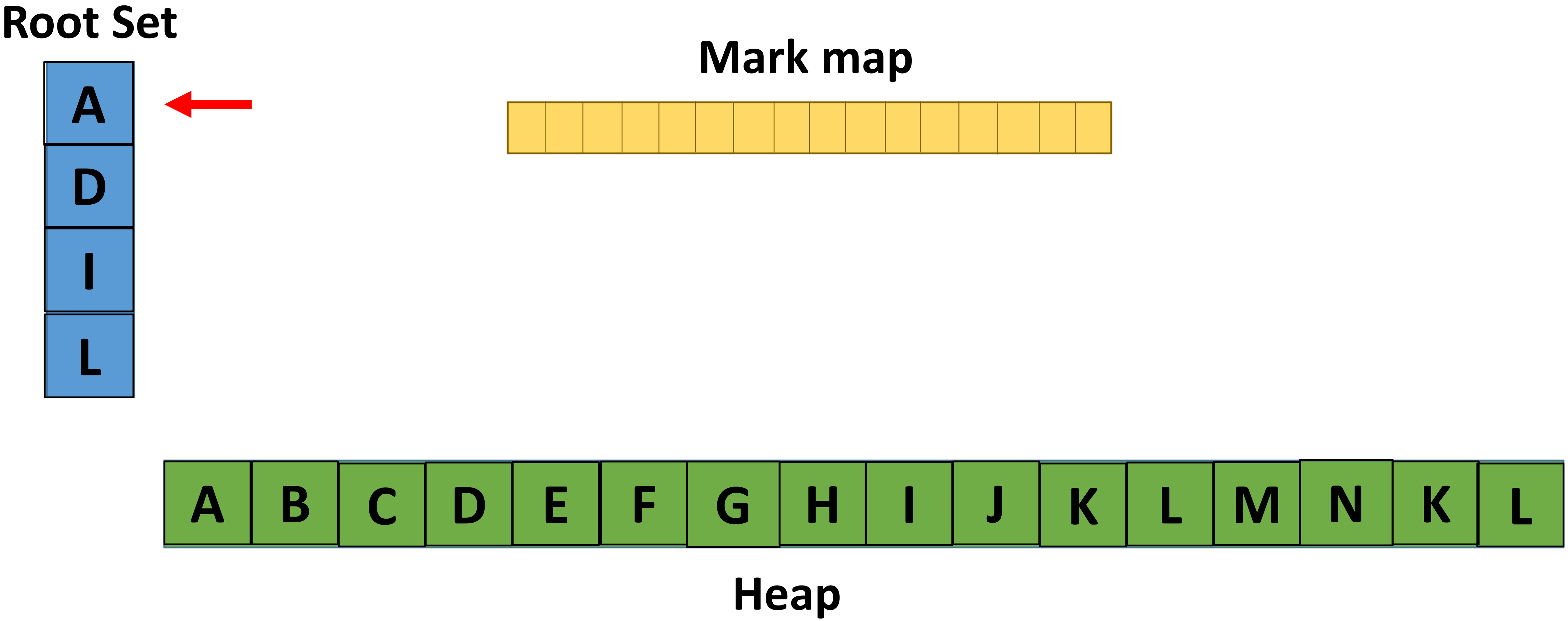
A
D
I
L

Mark map

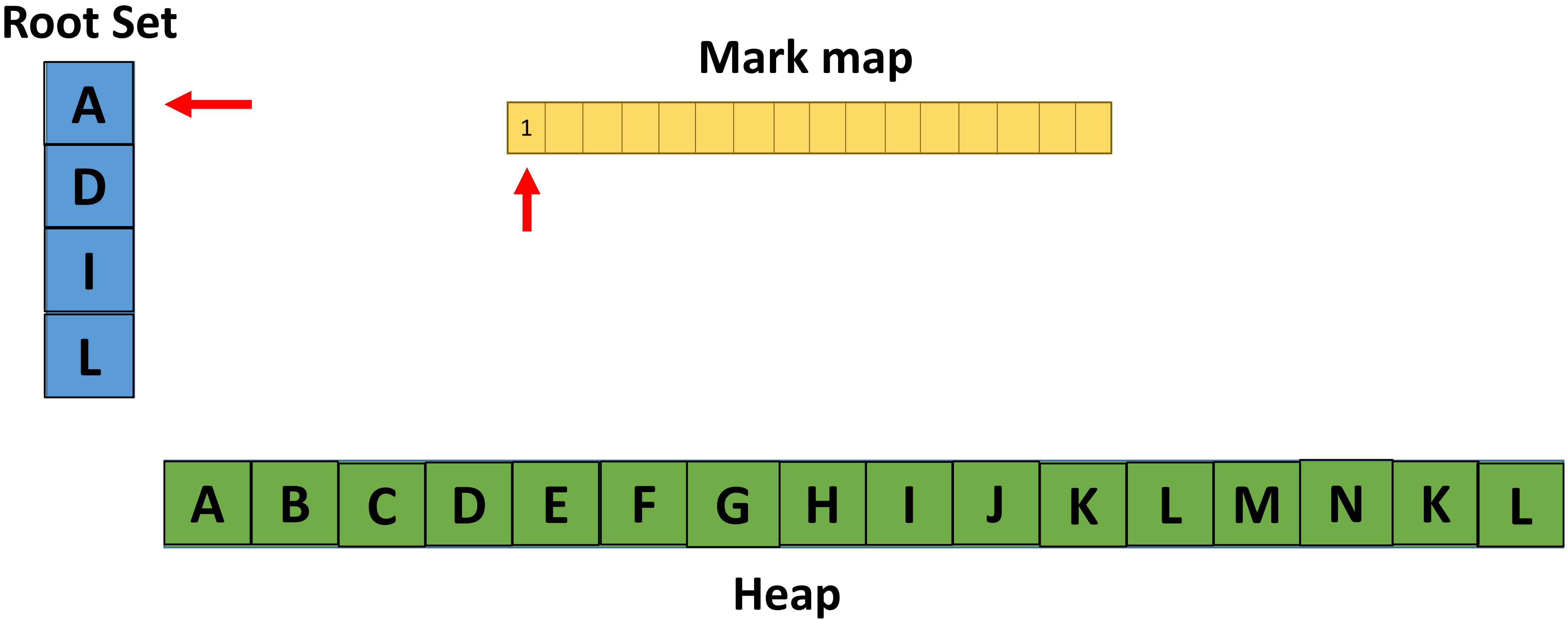
A B C D E F G H I J K L M N K L

Heap

-Xgcpolicy:optthruput marking



-Xgcpolicy:optthruput marking



Root Set

A
D
I
L

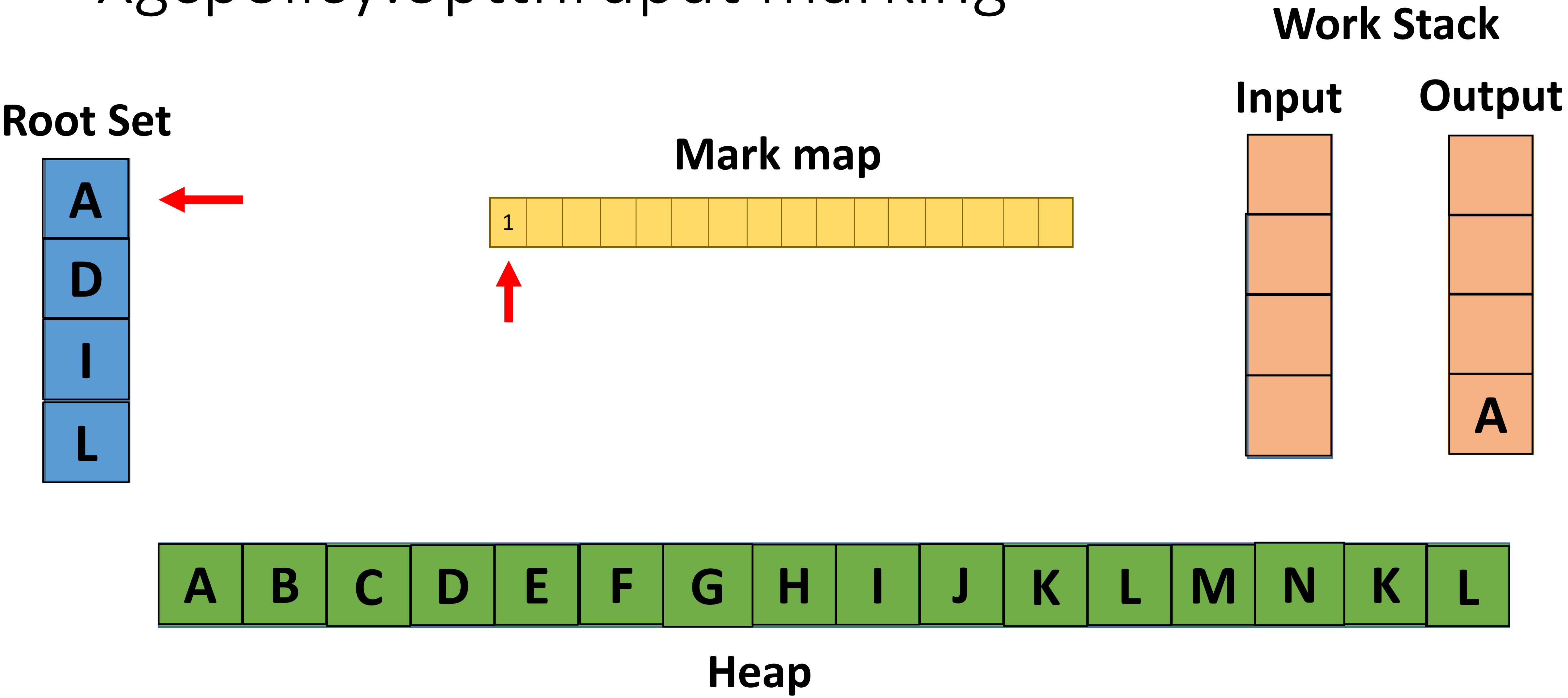
Mark map

1

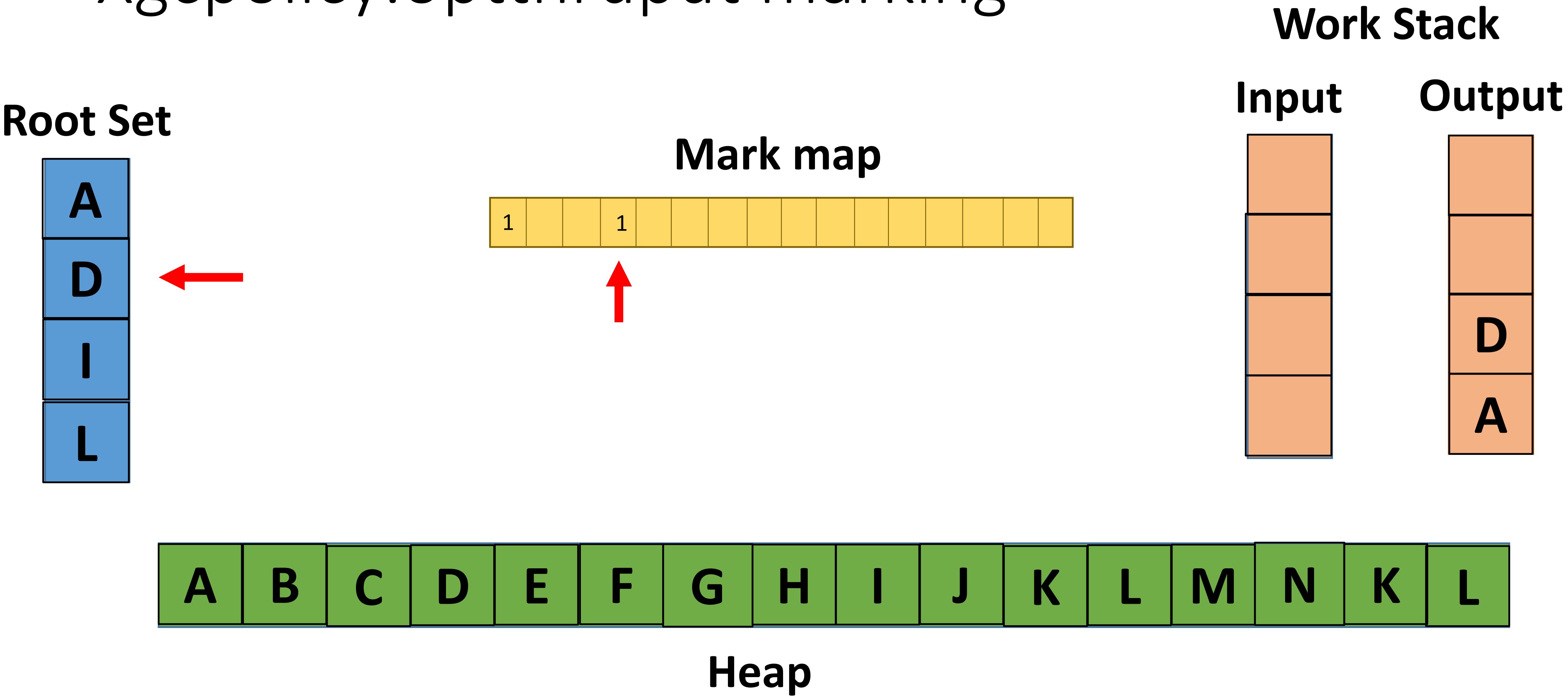
A B C D E F G H I J K L M N K L

Heap

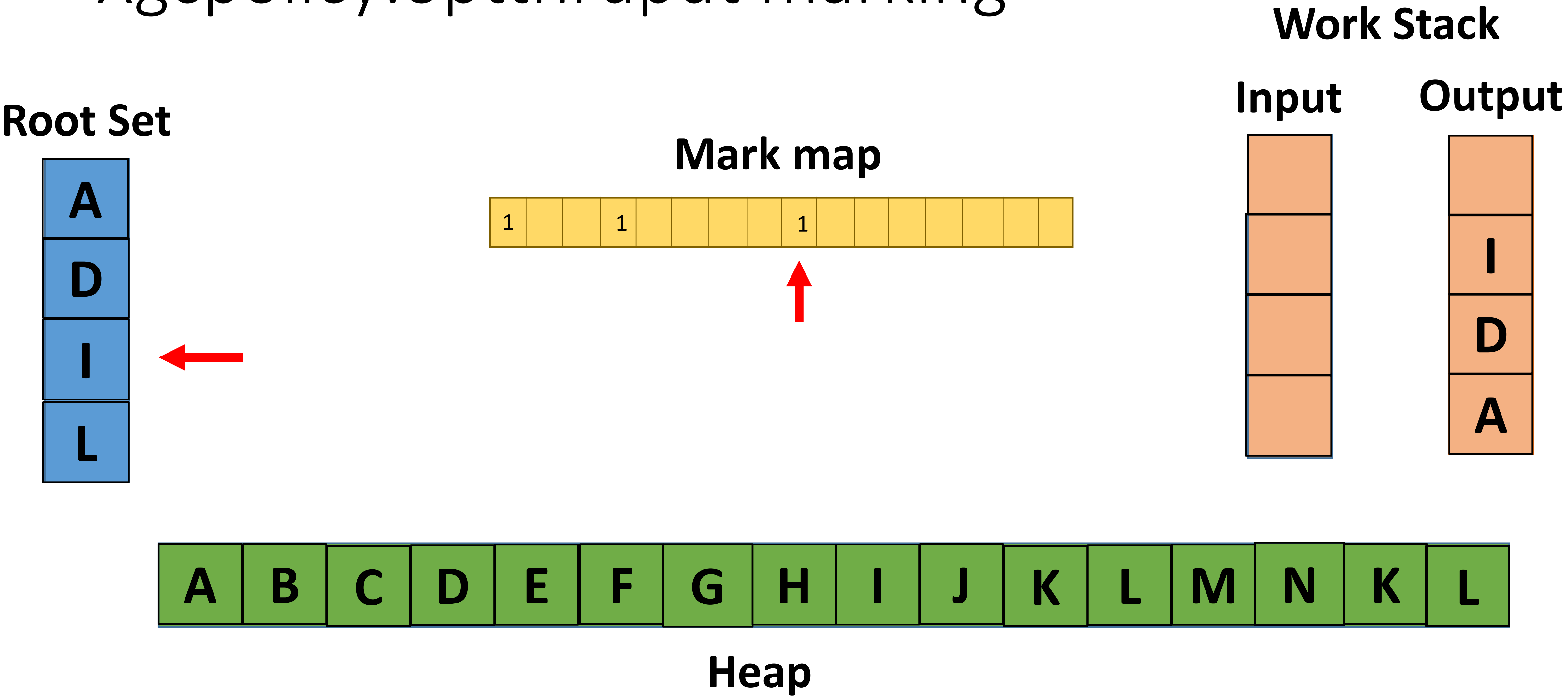
-Xgcpolicy:optthruput marking



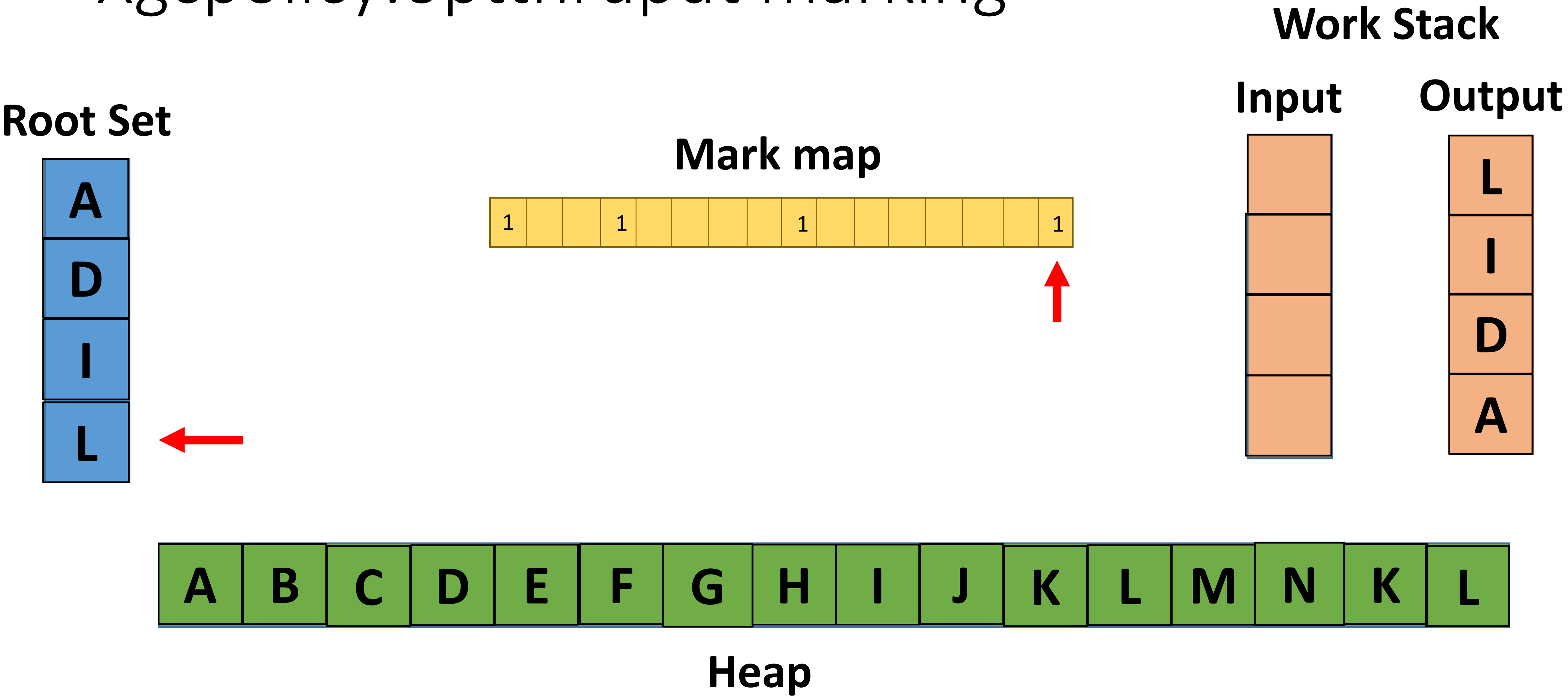
-Xgcpolicy:optthruput marking



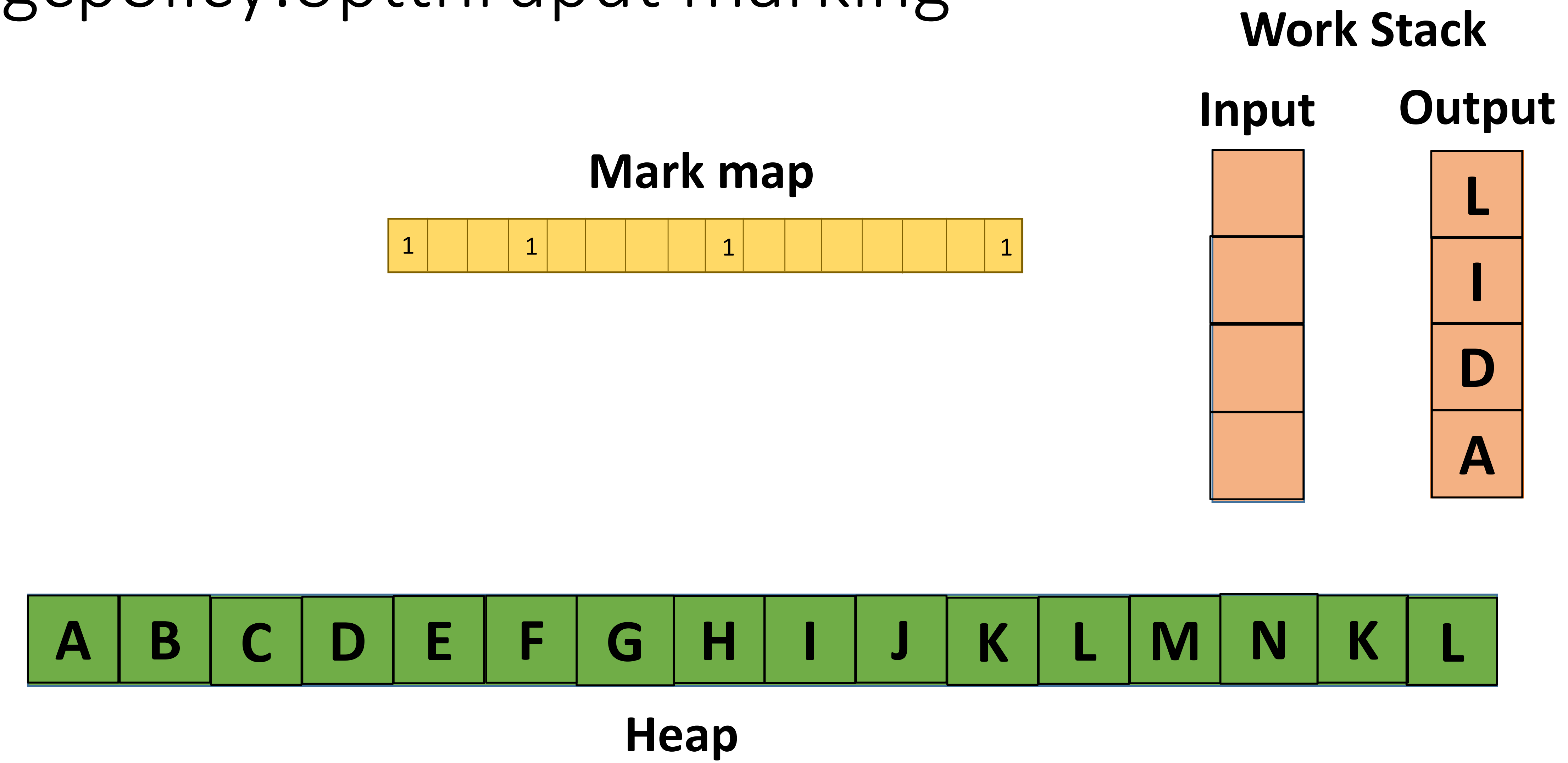
-Xgcpolicy:optthruput marking



-Xgcpolicy:optthruput marking



-Xgcpolicy:optthruput marking



Work Stack

Input	Output
	L
	I
	D
	A

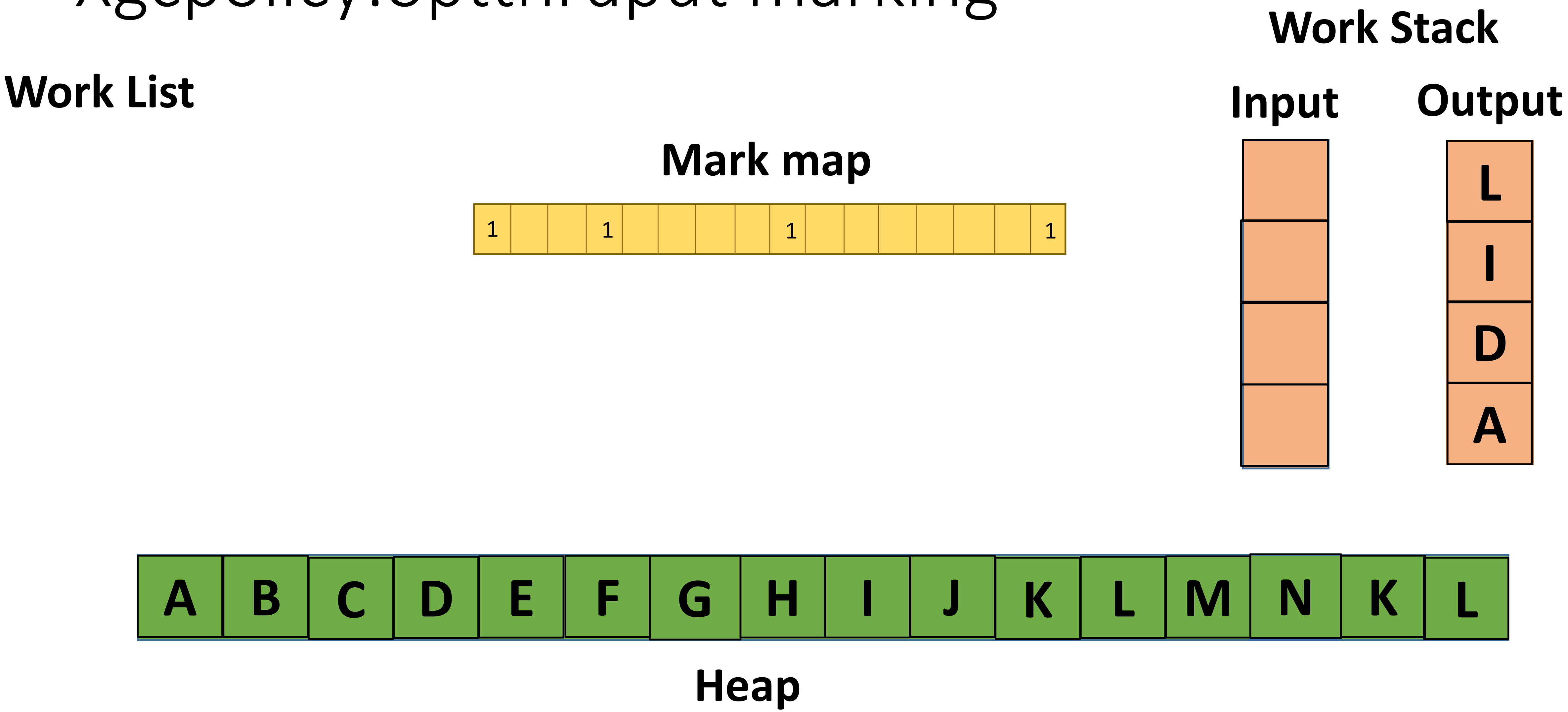
Mark map

1			1					1							1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	K	L

Heap

-Xgcpolicy:optthruput marking



Work List

Mark map

1			1					1							1

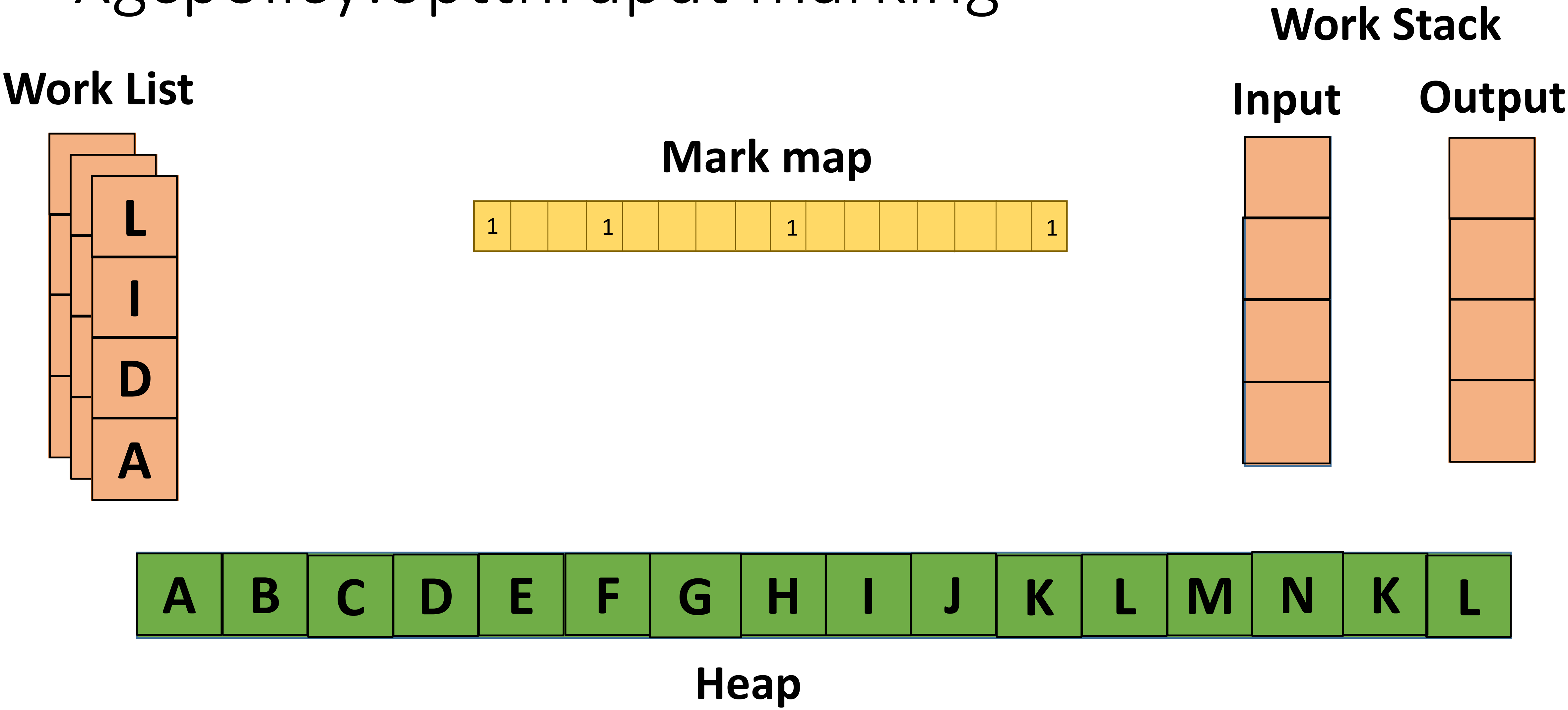
Work Stack

Input	Output
	L
	I
	D
	A

A	B	C	D	E	F	G	H	I	J	K	L	M	N	K	L

Heap

-Xgcpolicy:optthruput marking



Work Stack

Input Output

Work List

L
I
D
A

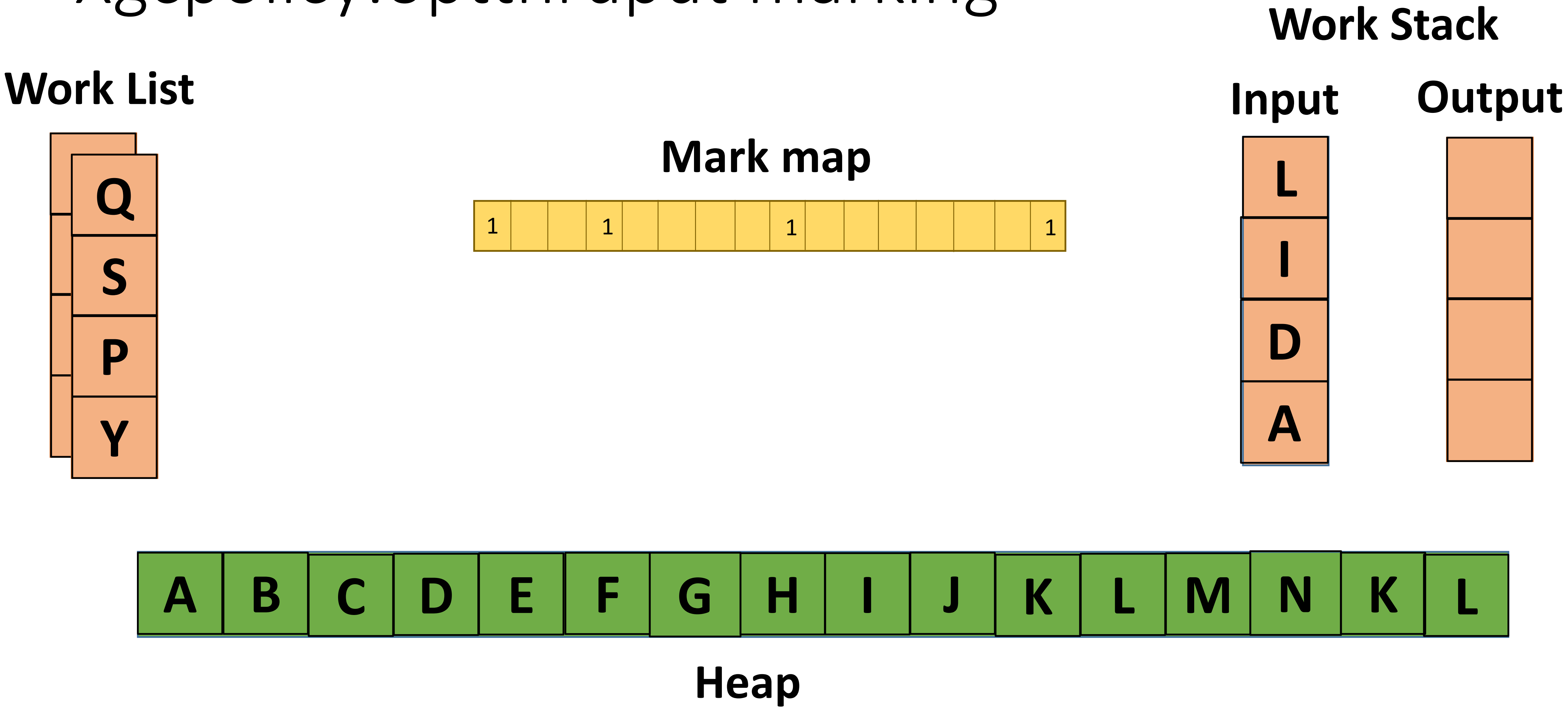
Mark map

1			1					1						1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	K	L

Heap

-Xgcpolicy:optthruput marking



Work List

Q

S

P

Y

Mark map

1 1 1 1

Work Stack

Input

L

I

D

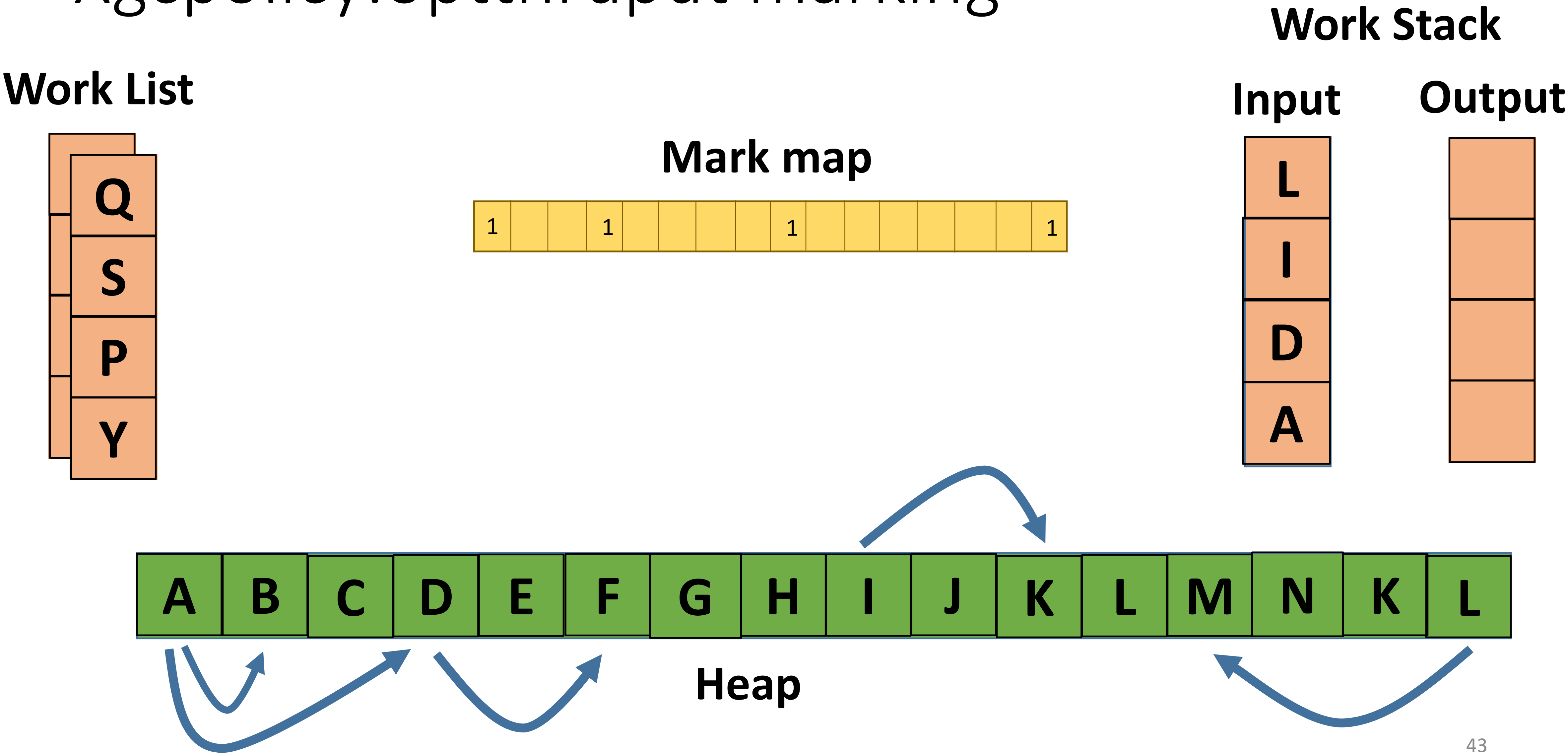
A

Output

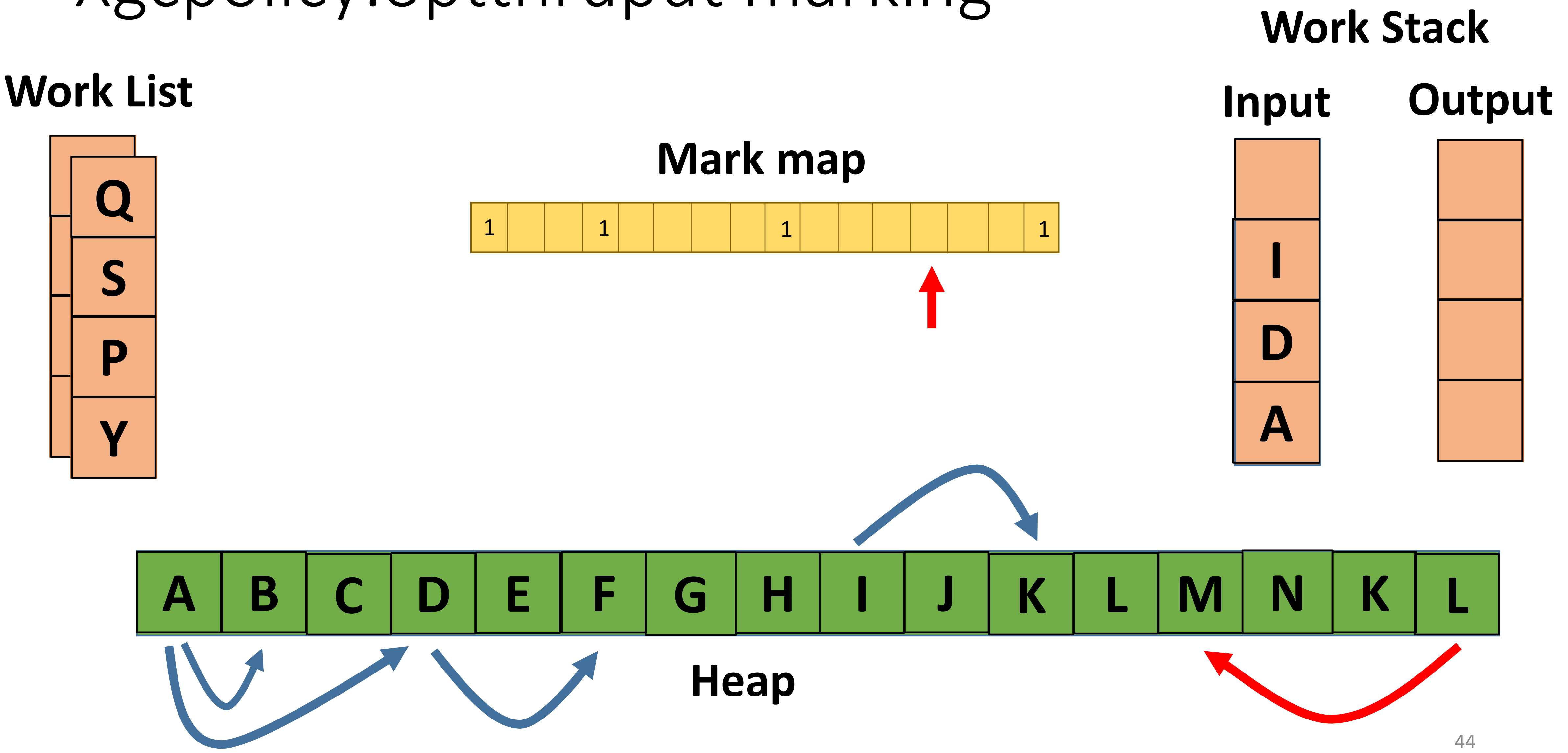
A B C D E F G H I J K L M N K L

Heap

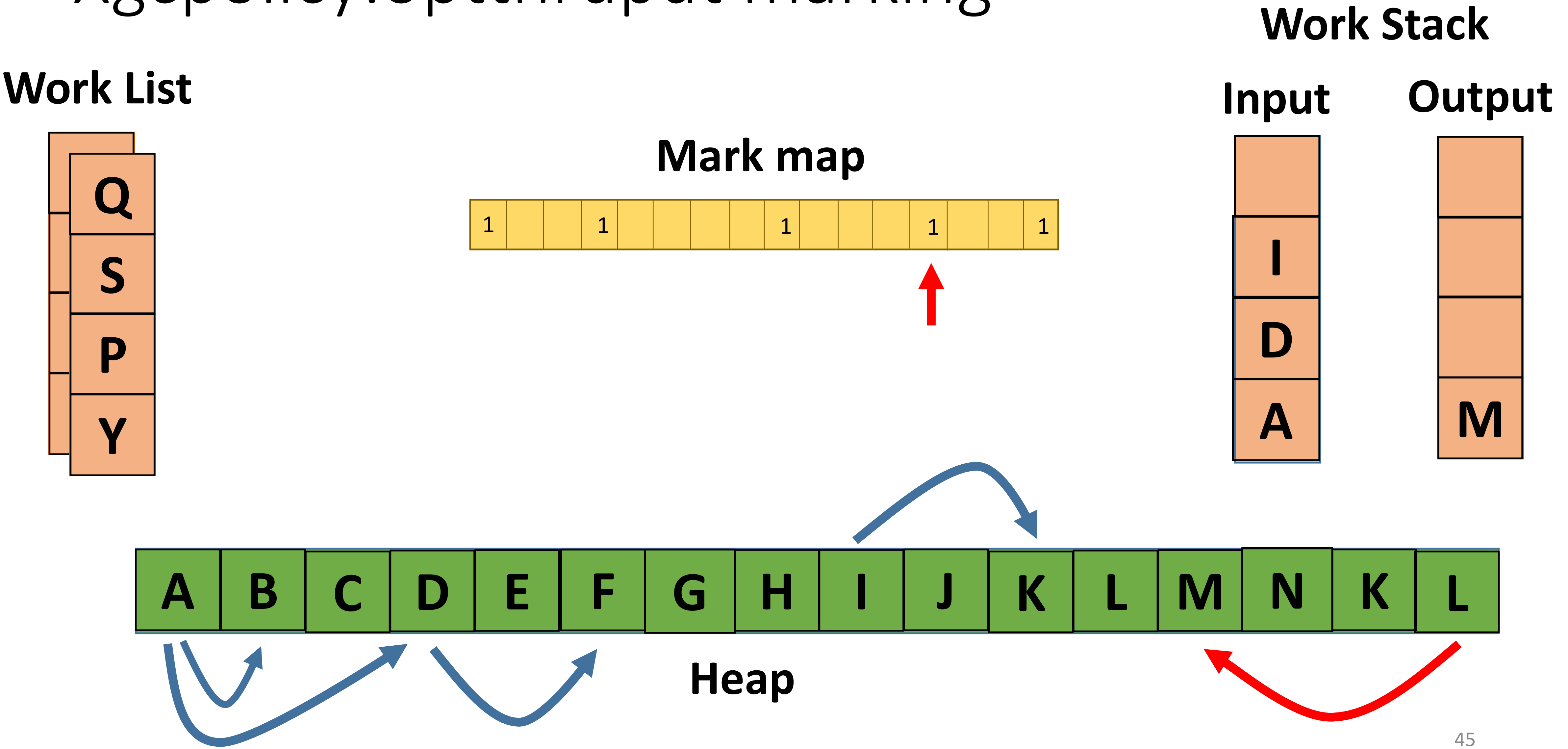
-Xgcpolicy:optthruput marking



-Xgcpolicy:optthruput marking



-Xgcpolicy:optthruput marking



Work List

Q
S
P
Y

Mark map

1			1					1				1			1

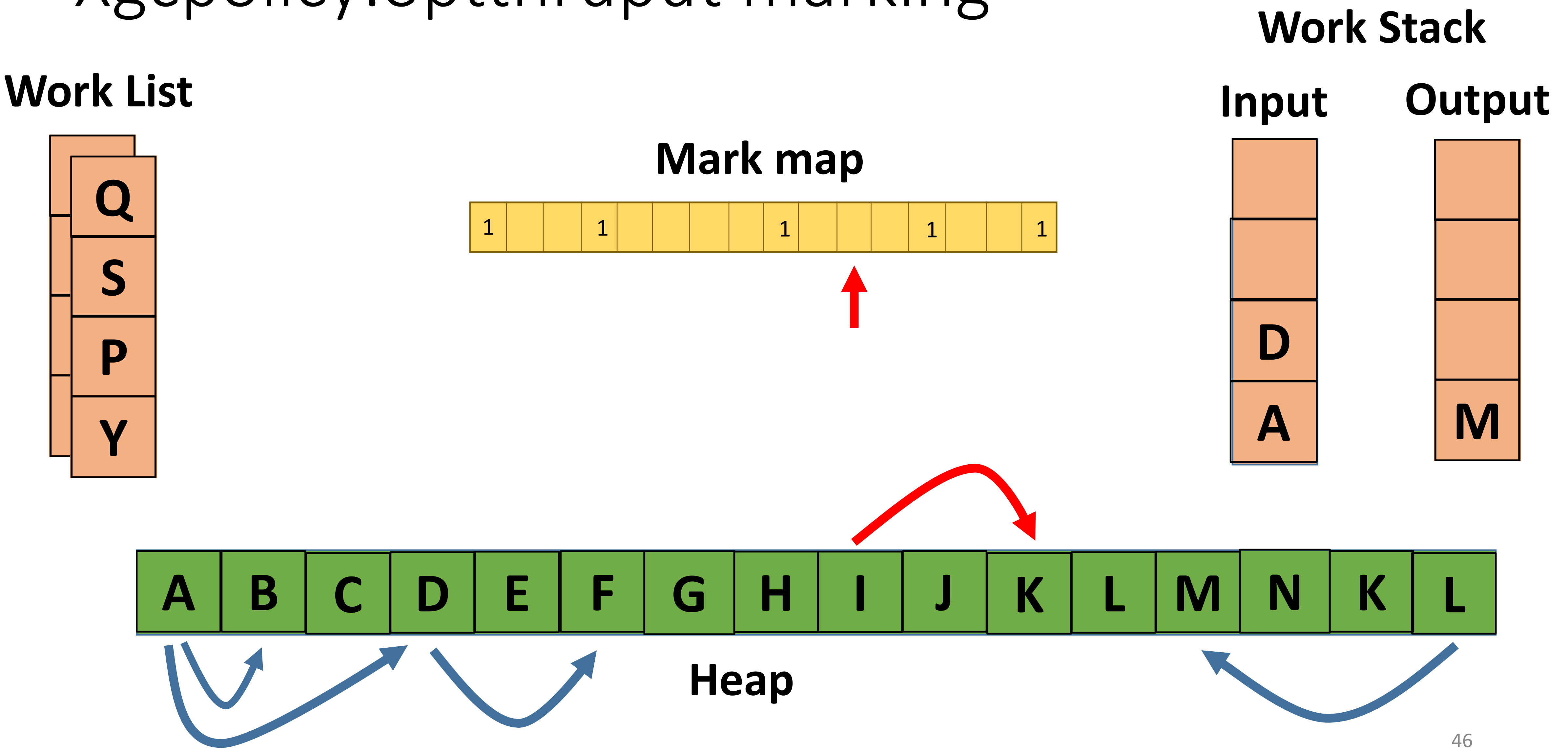
Work Stack

Input	Output
I	
D	
A	M

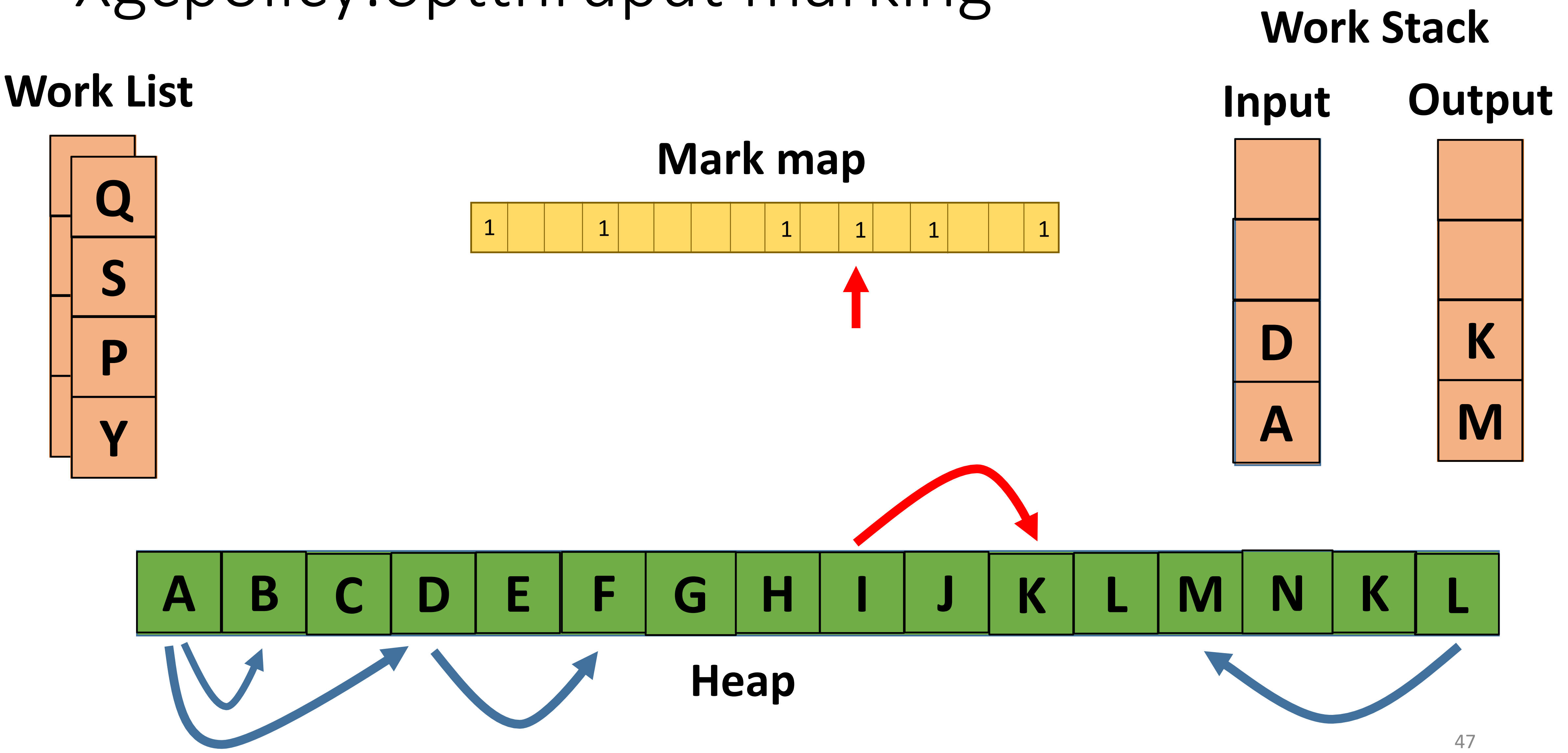
A	B	C	D	E	F	G	H	I	J	K	L	M	N	K	L

Heap

-Xgcpolicy:optthruput marking



-Xgcpolicy:optthruput marking



Work List

Mark map

1			1					1		1		1			1

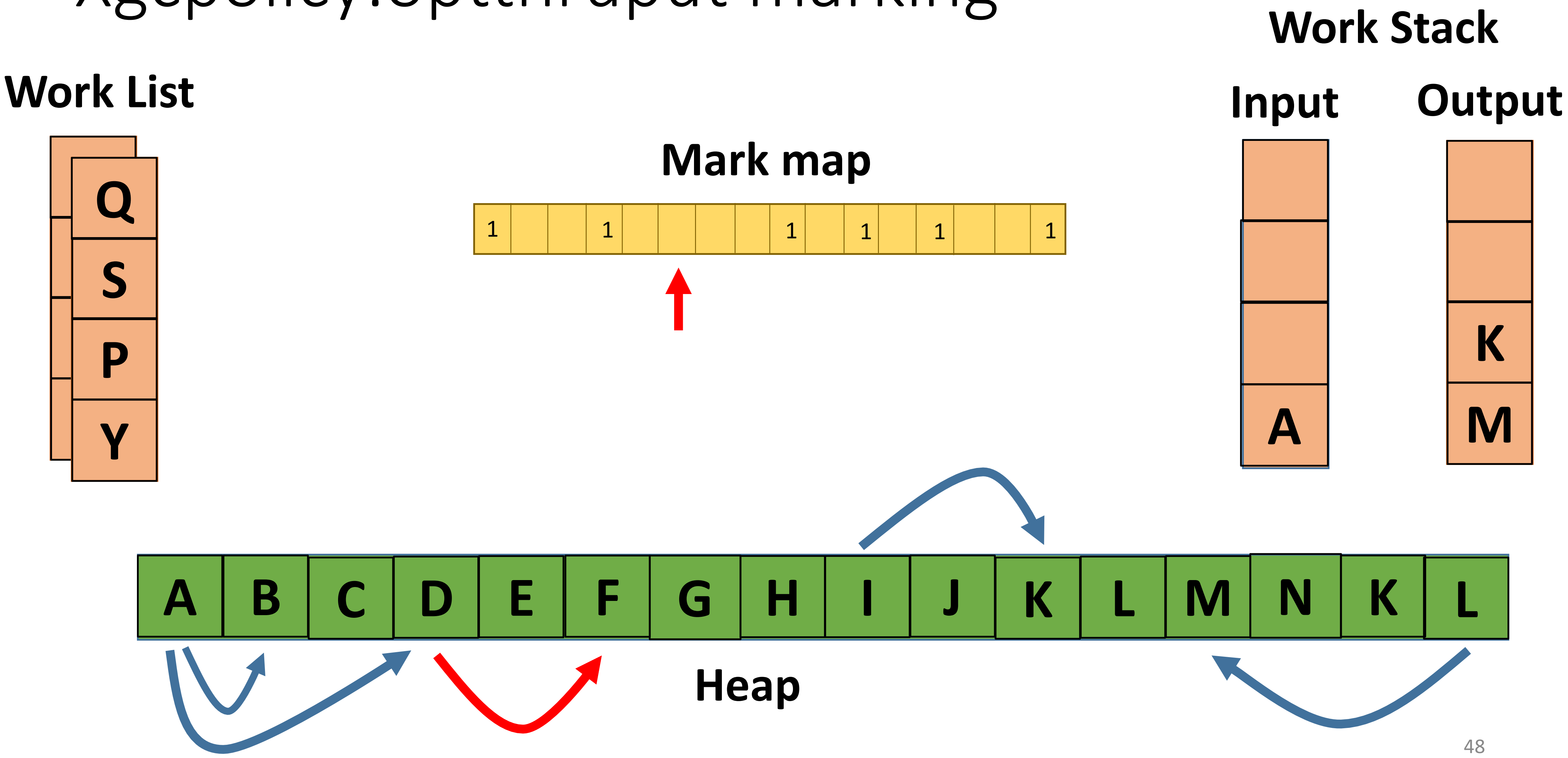
Work Stack

Input

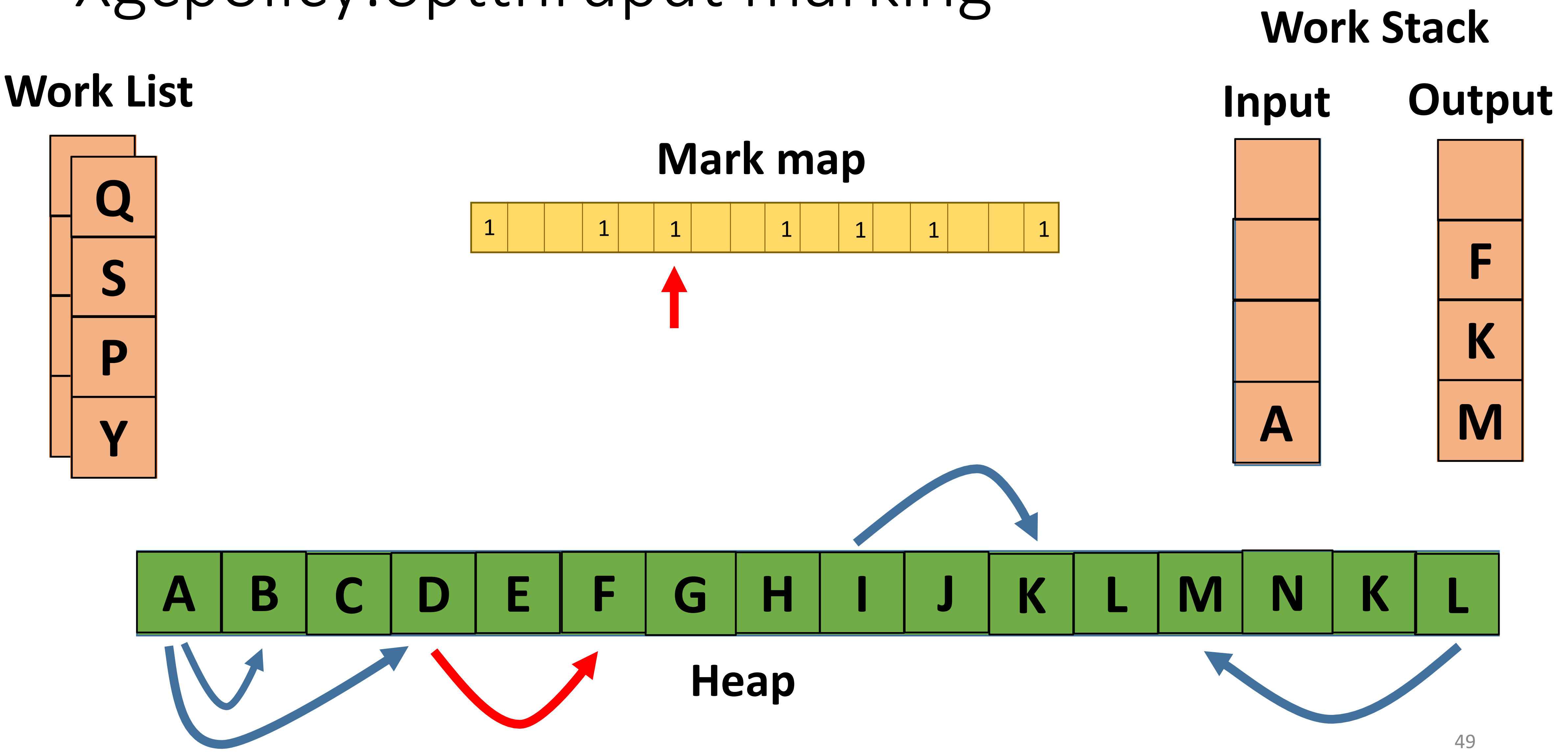
Output

Heap

-Xgcpolicy:optthruput marking



-Xgcpolicy:optthruput marking



Work List

Mark map

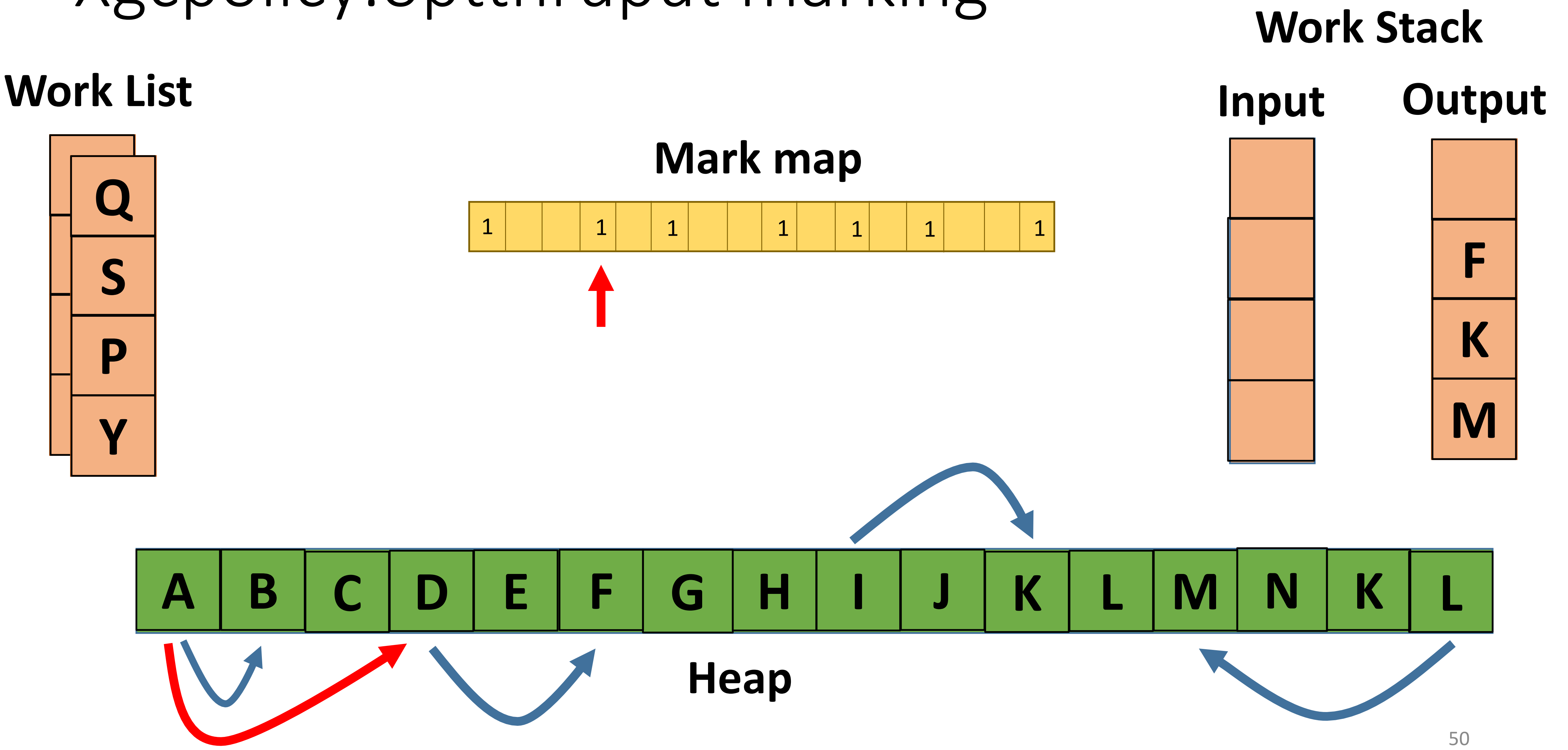
Work Stack

Input

Output

Heap

-Xgcpolicy:optthruput marking



Work List

Q
S
P
Y

Mark map

1			1		1			1		1		1			1

Work Stack

Input

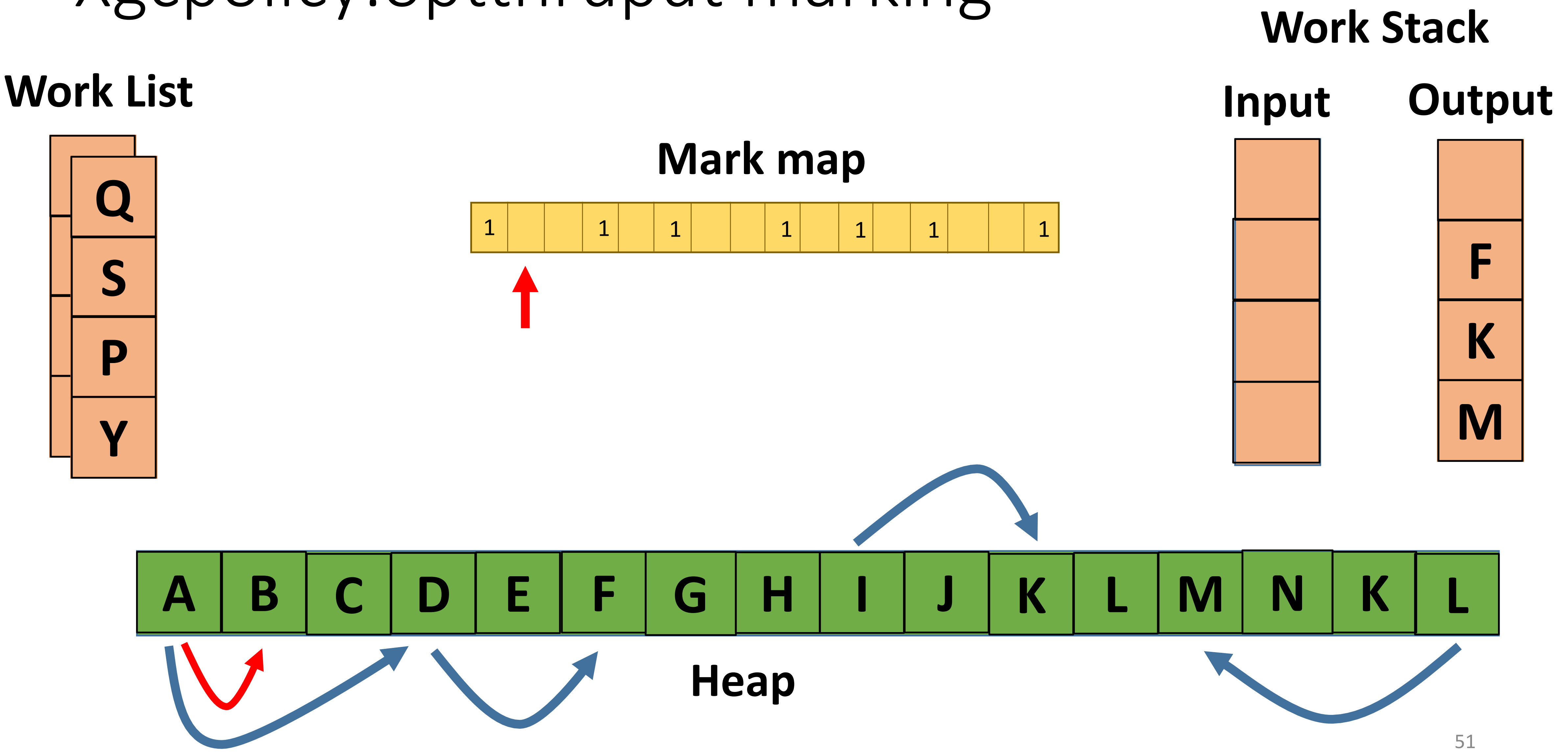
Output

F
K
M

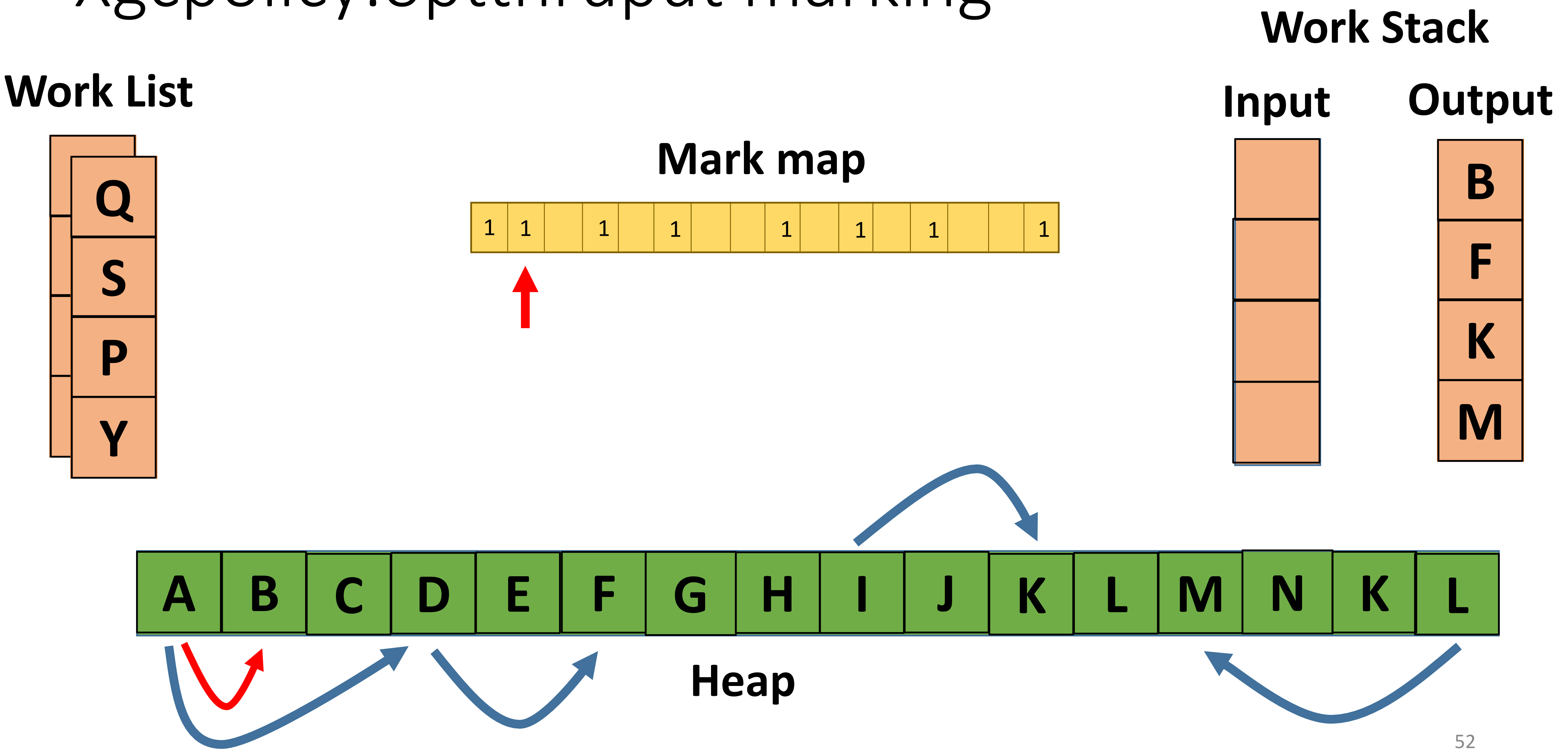
A B C D E F G H I J K L M N K L

Heap

-Xgcpolicy:optthruput marking



-Xgcpolicy:optthruput marking



Work Stack

Input

Output

B

F

K

M

Work List

Q

S

P

Y

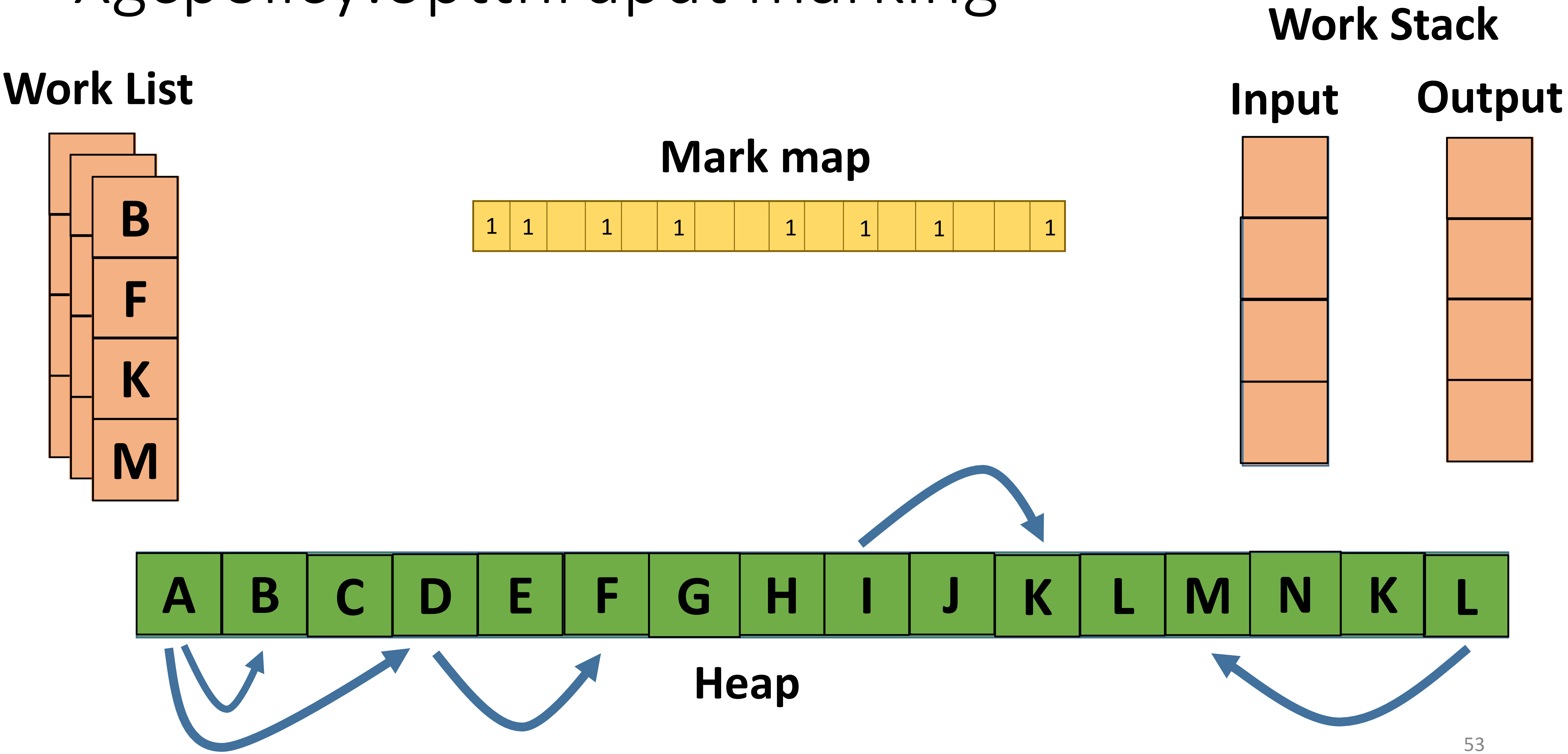
Mark map

1 1 1 1 1 1 1 1

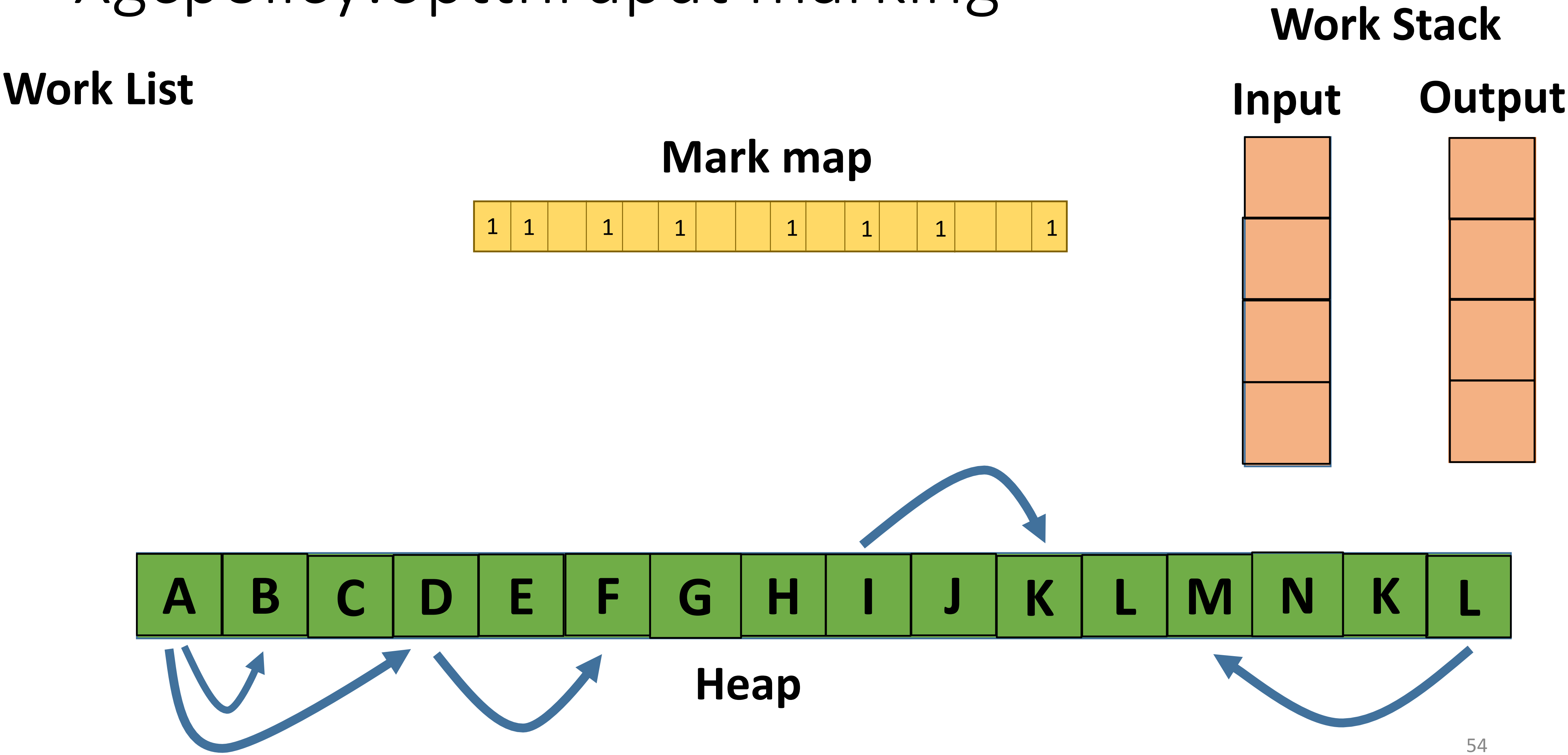
A B C D E F G H I J K L M N K L

Heap

-Xgcpolicy:optthruput marking



-Xgcpolicy:optthruput marking



Work List

Work Stack

Input **Output**

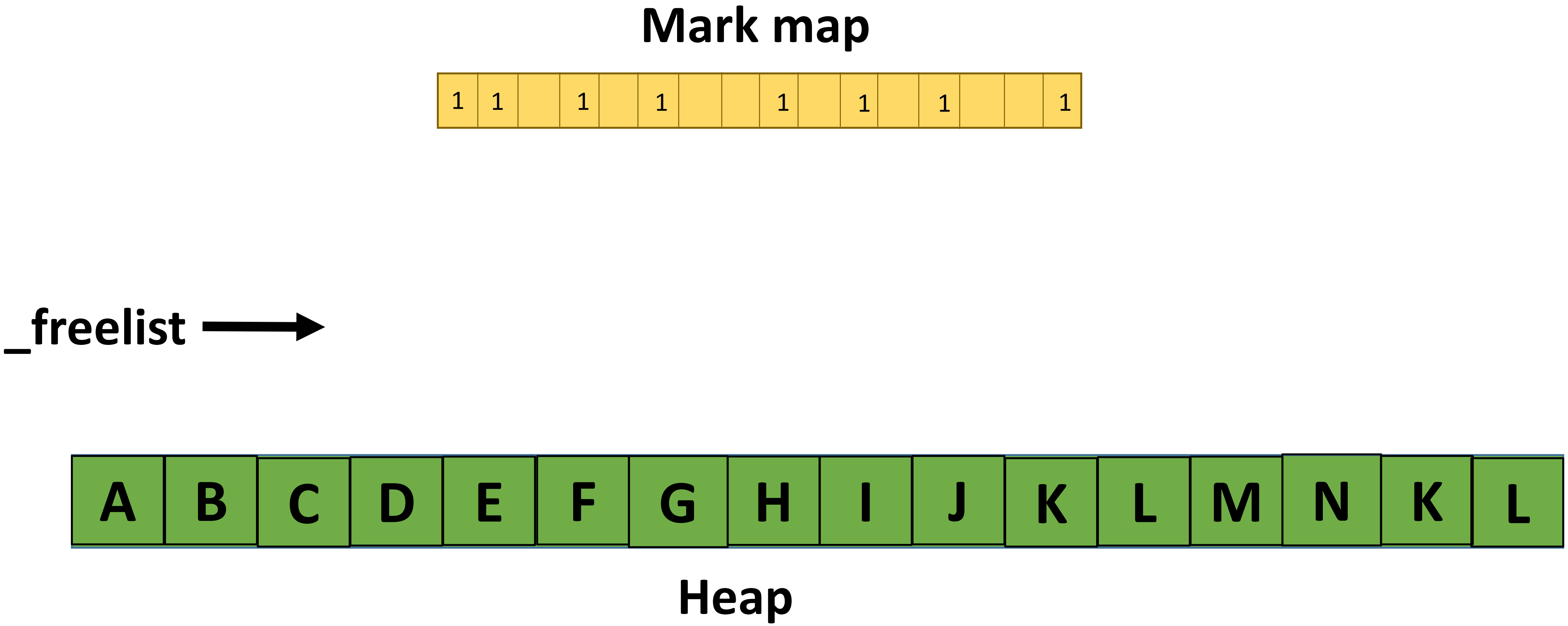
Mark map

1	1		1		1			1		1		1			1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	K	L

Heap

-Xgcpolicy:optthruput sweeping



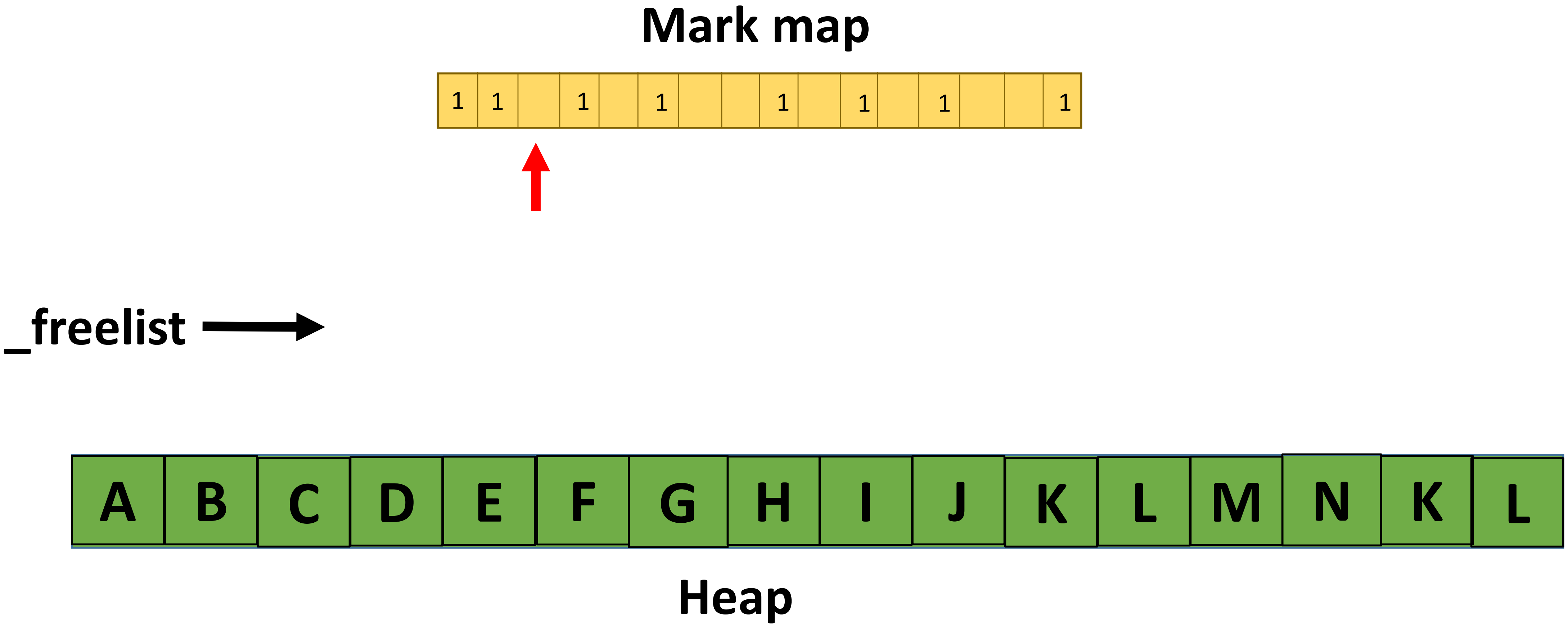
Mark map

1	1		1		1			1		1		1			1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	K	L

Heap

-Xgcpolicy:optthruput sweeping



Mark map

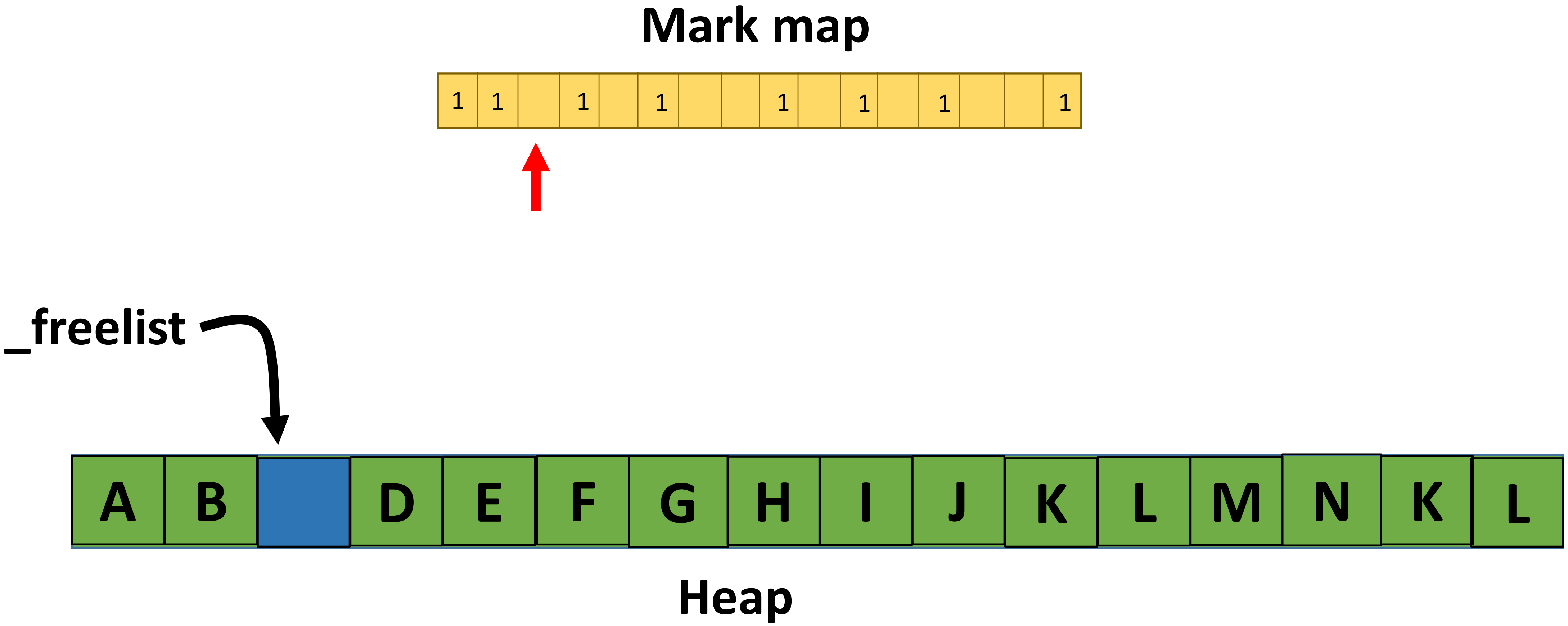
1	1		1		1			1		1		1			1

_freelist

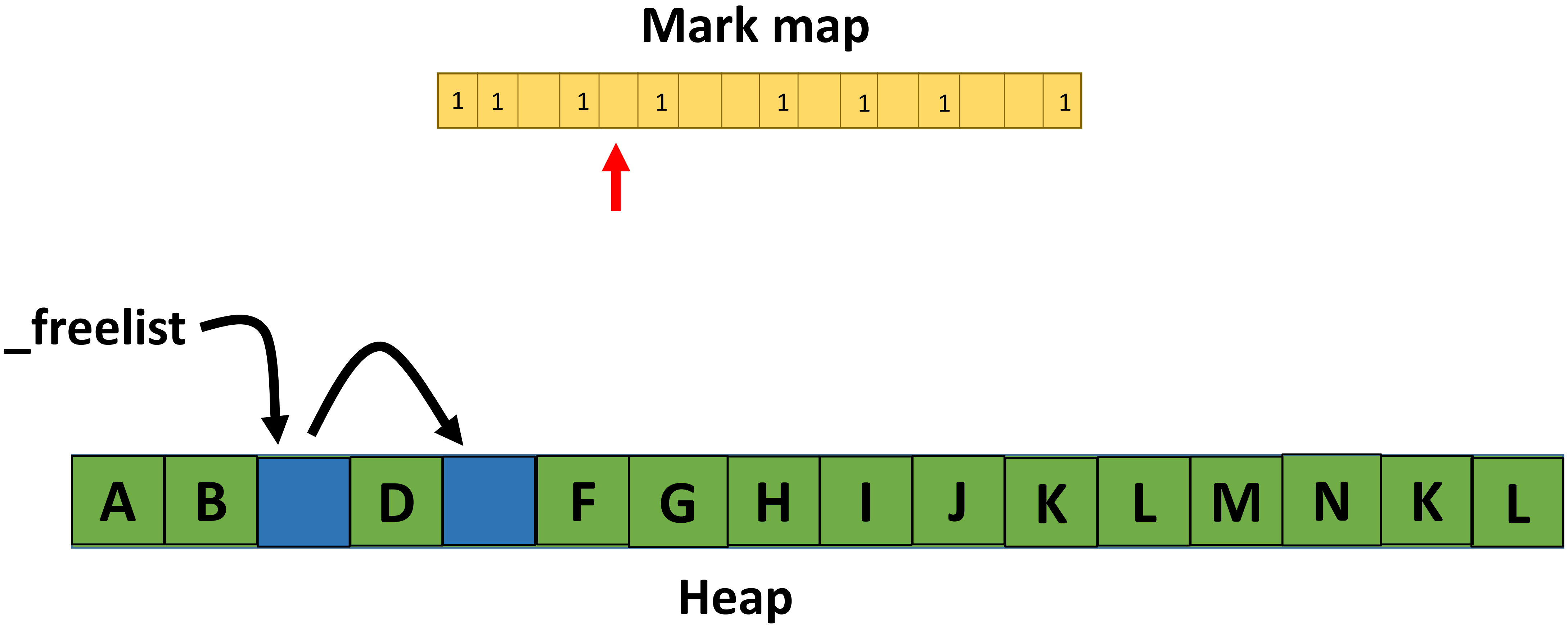
A	B	C	D	E	F	G	H	I	J	K	L	M	N	K	L

Heap

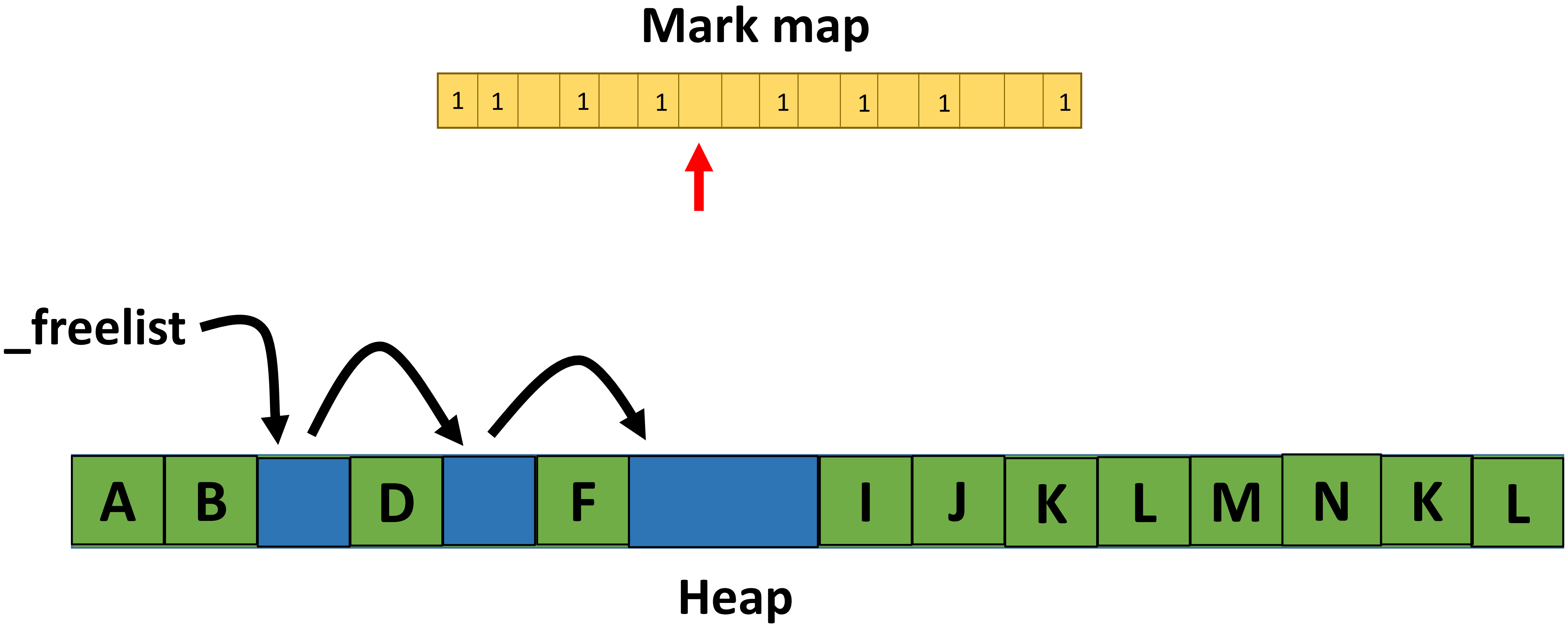
-Xgcpolicy:optthruput sweeping



-Xgcpolicy:optthruput sweeping



-Xgcpolicy:optthruput sweeping



Mark map

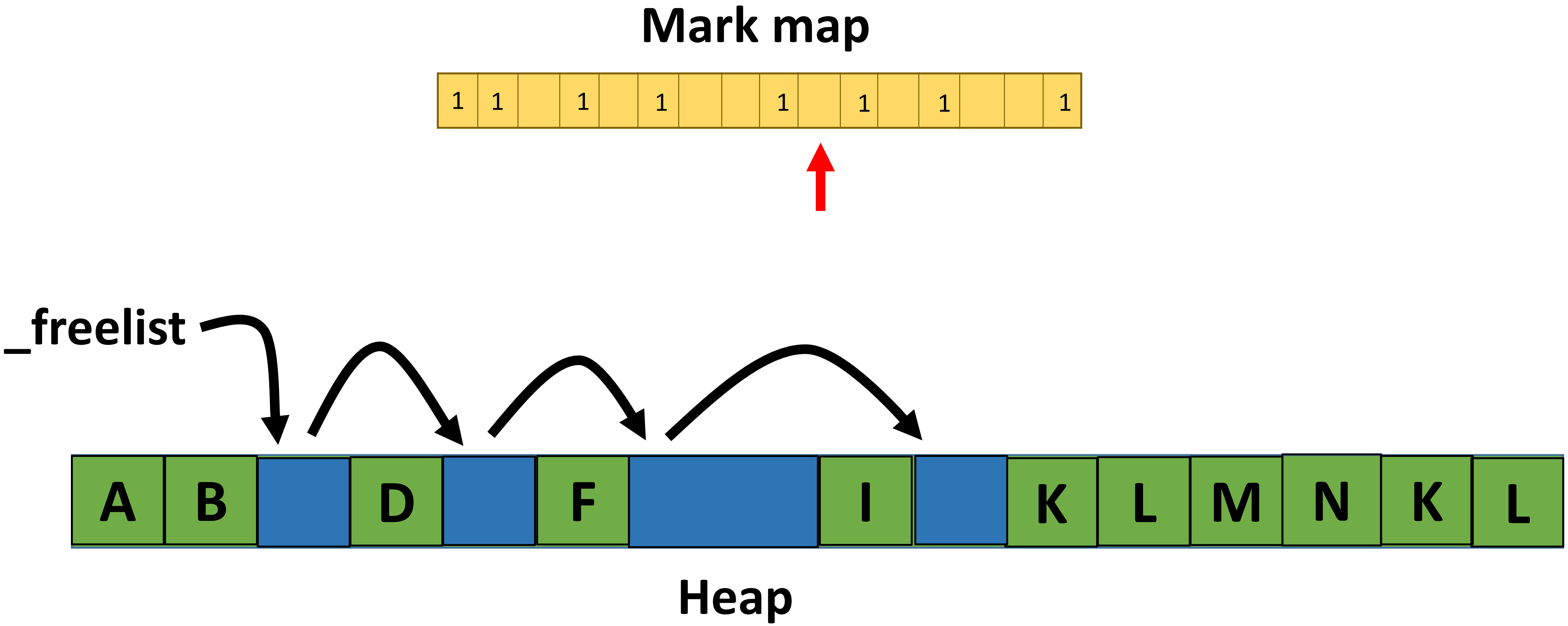
1	1		1		1			1		1		1			1

_freelist

A	B		D		F		I	J	K	L	M	N	K	L

Heap

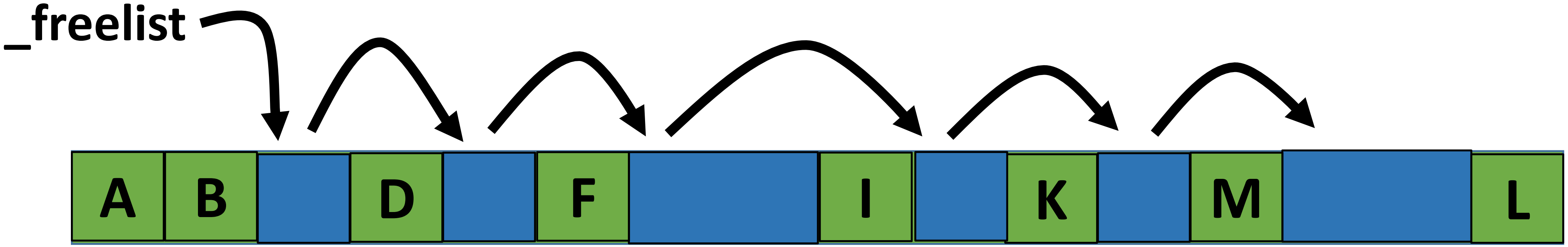
-Xgcpolicy:optthruput sweeping



-Xgcpolicy:optthruput sweeping

Mark map

1	1		1		1			1		1		1			1

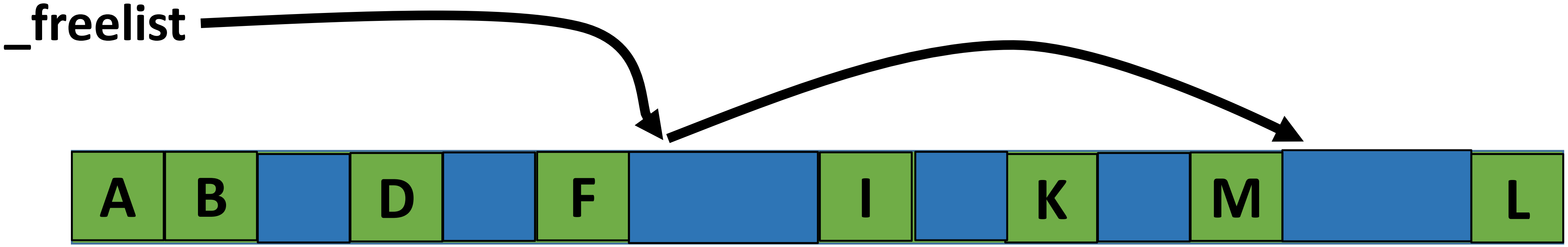


Heap

-Xgcpolicy:optthruput sweeping

Mark map

1	1		1		1			1		1		1			1

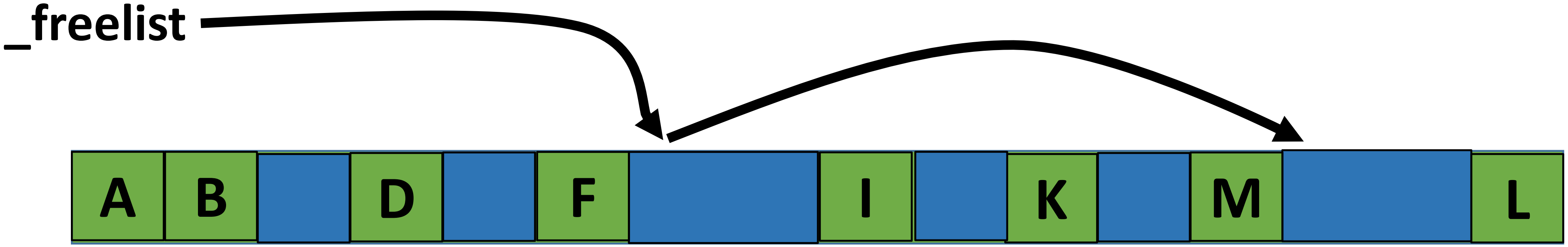


Heap

-Xgcpolicy:optthruput compaction

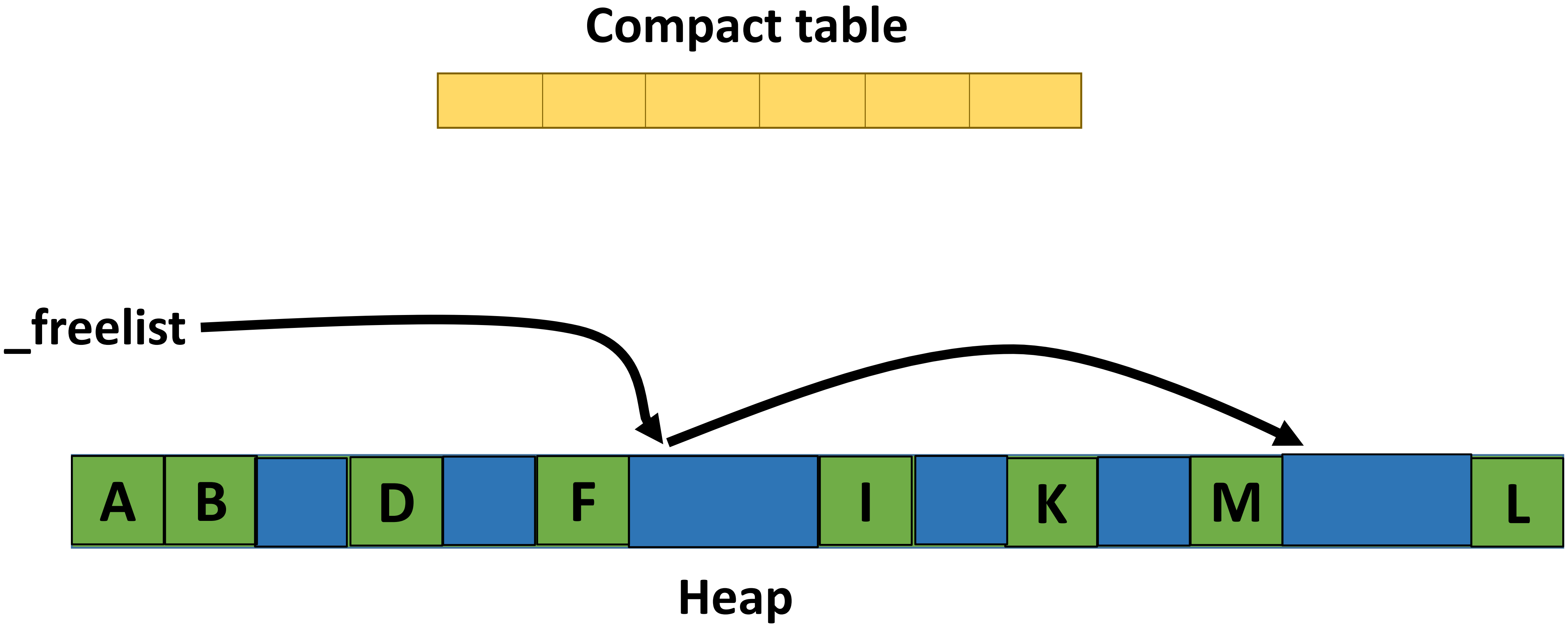
Mark map

1	1		1		1			1		1		1			1



Heap

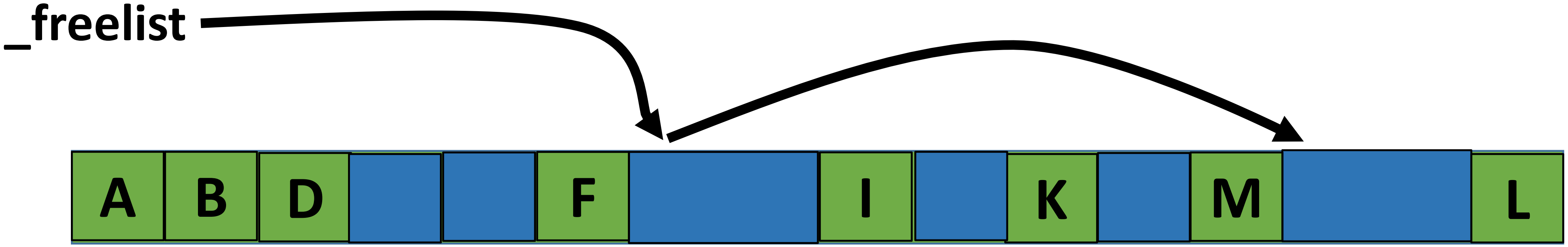
-Xgcpolicy:optthruput compaction



-Xgcpolicy:optthruput compaction

Compact table

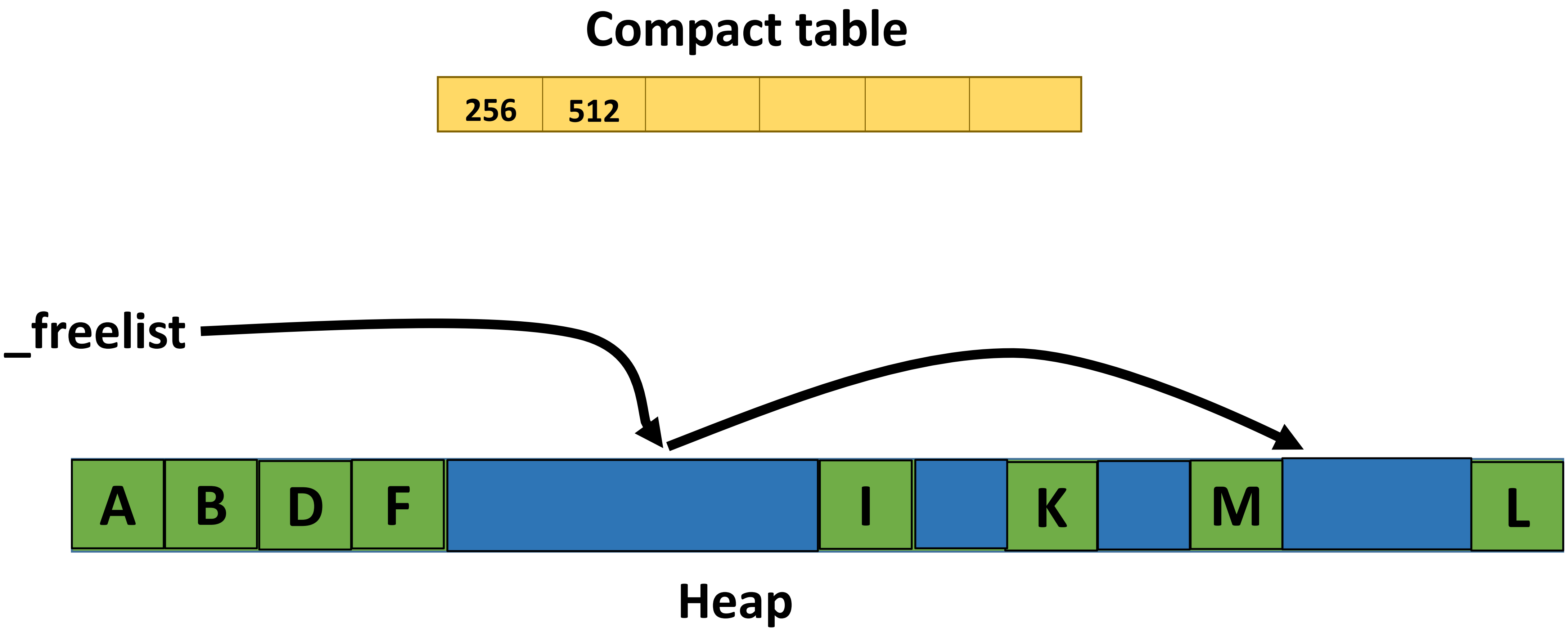
256					



_freelist

Heap

-Xgcpolicy:optthruput compaction



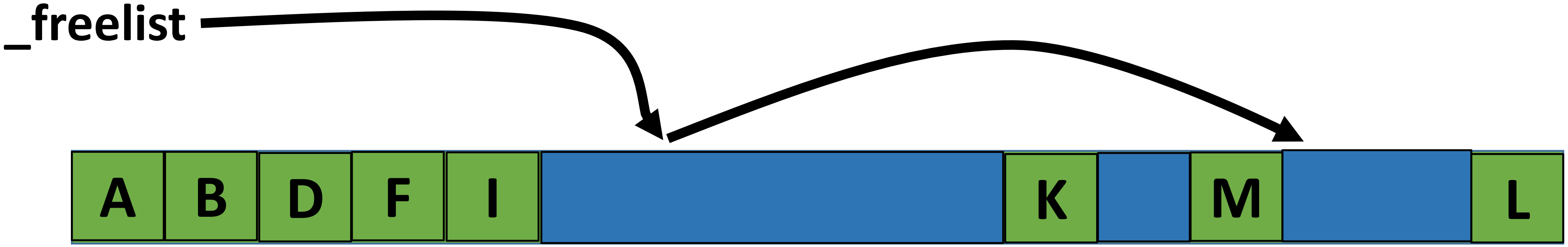
Compact table

256	512				

-Xgcpolicy:optthruput compaction

Compact table

256	512	1024			

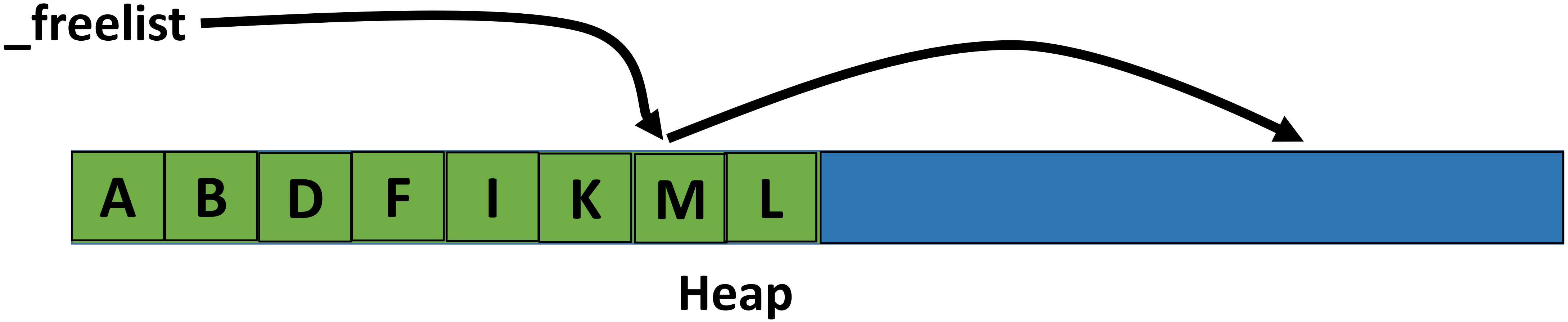


Heap

-Xgcpolicy:optthruput compaction

Compact table

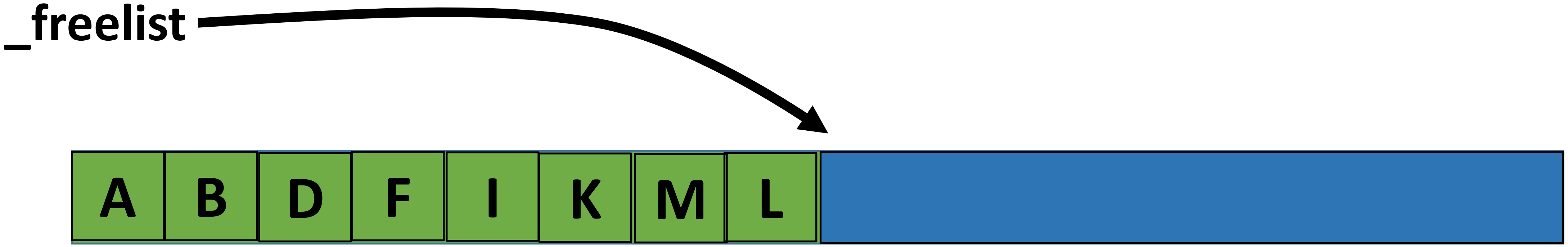
256	512	1024	1280	1536	2048



-Xgcpolicy:optthruput compaction

Compact table

256	512	1024	1280	1536	2048



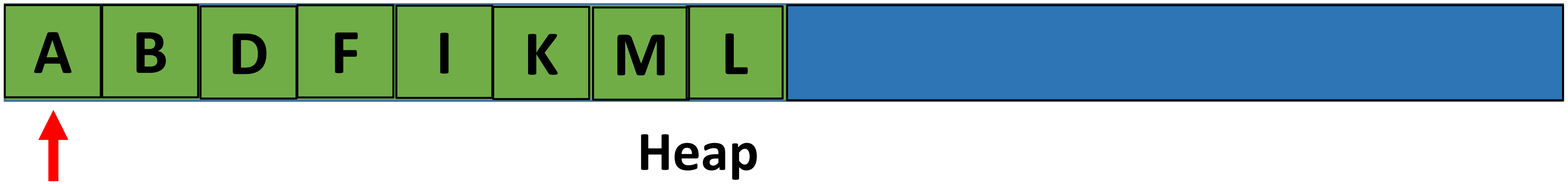
Heap

-Xgcpolicy:optthruput compaction

Compact table

256	512	1024	1280	1536	2048

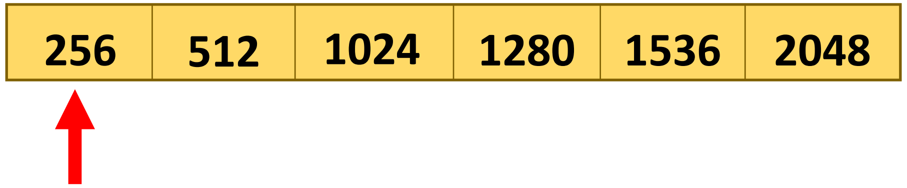
A.field1 = D



Heap

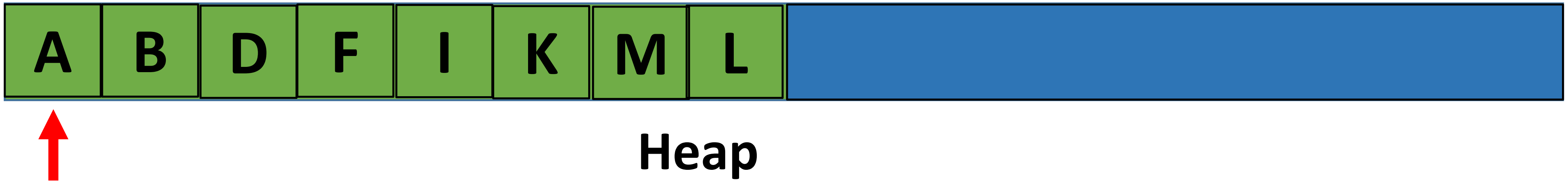
-Xgcpolicy:optthruput compaction

Compact table



256	512	1024	1280	1536	2048

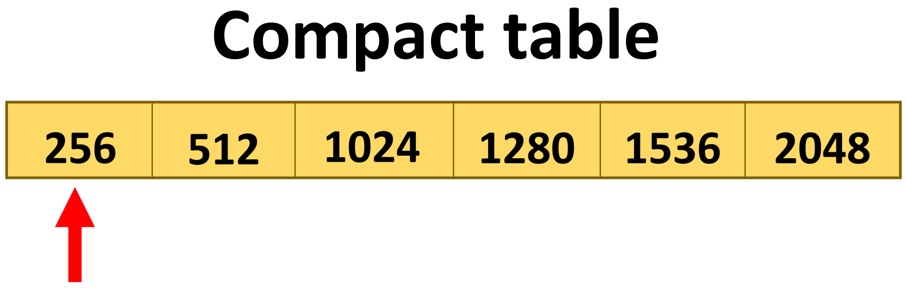
A.field1 = (D – compactTable[0])



A	B	D	F	I	K	M	L	

Heap

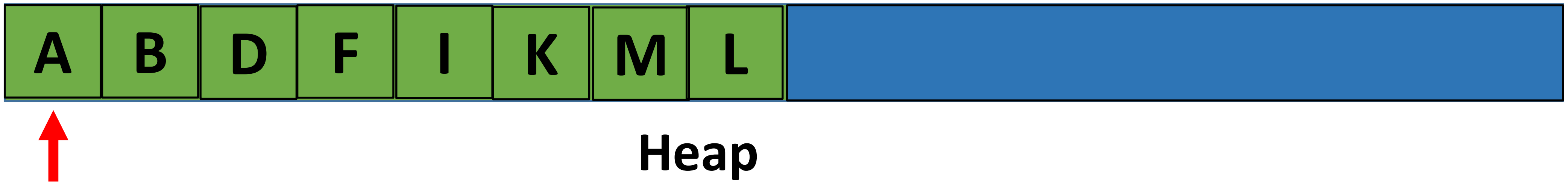
-Xgcpolicy:optthruput compaction



Compact table

256	512	1024	1280	1536	2048

A.field1 = (D – 256)



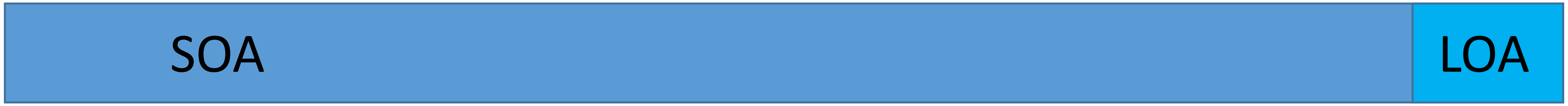
Heap

-Xgcpolicy:optavgpause

- Concurrent global collector
  - Mark and sweep phases are completed concurrently with the application. If required, compaction is completed STW
  - Utilizes a low priority background to complete concurrent work
  - **Application threads also perform concurrent GC work!**
- Improves STW pause times and application responsiveness
- Introduces the requirement for an object write barrier
- GC native memory overhead for mark map, work packets and **card table**

-Xgcpolicy:optavgpause heap

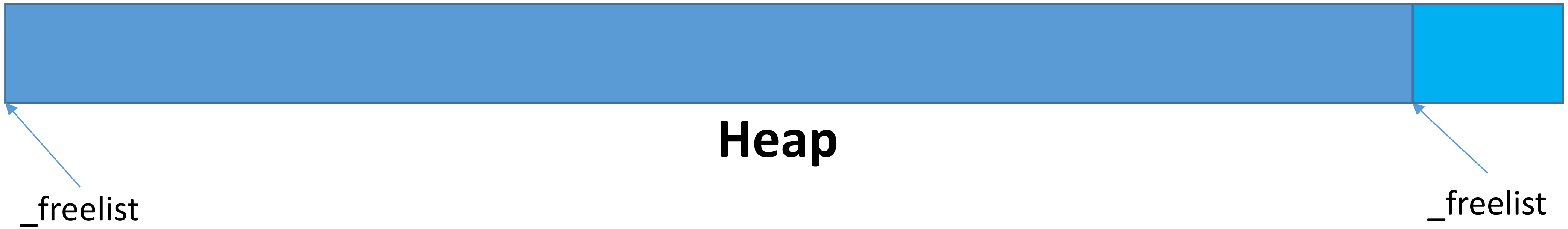
- Heap layout is the same as optthruput



Heap

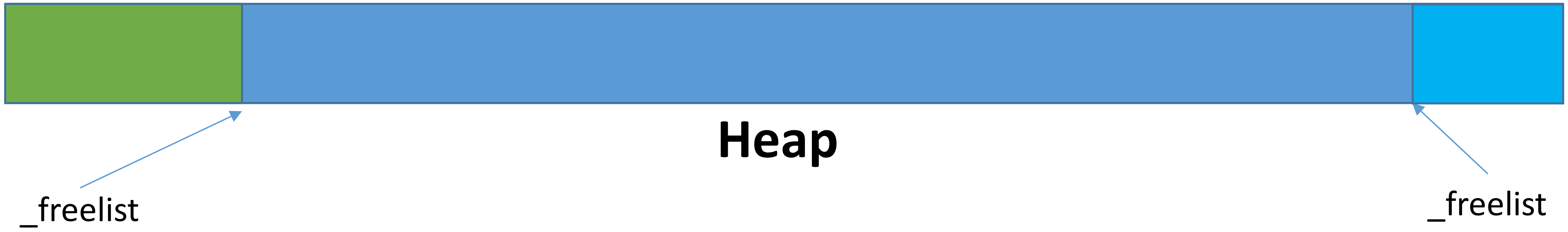
-Xgcpolicy:optavgpause heap

- Allocation is the same as with optthruput



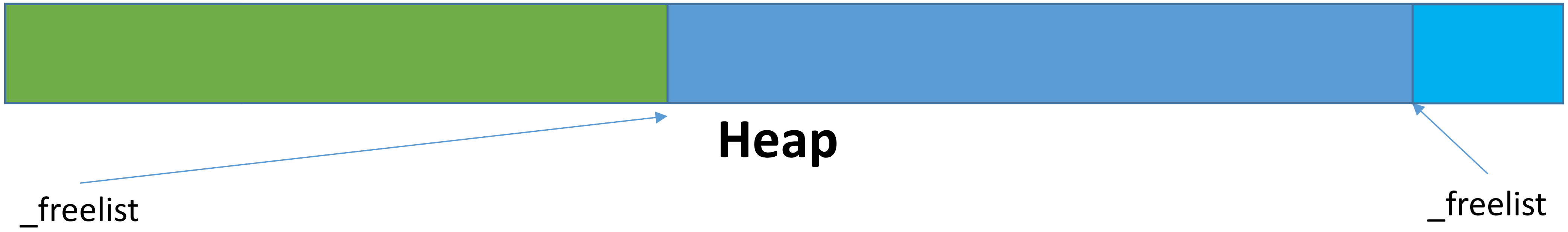
-Xgcpolicy:optavgpause heap

- Allocation is the same as with optthruput



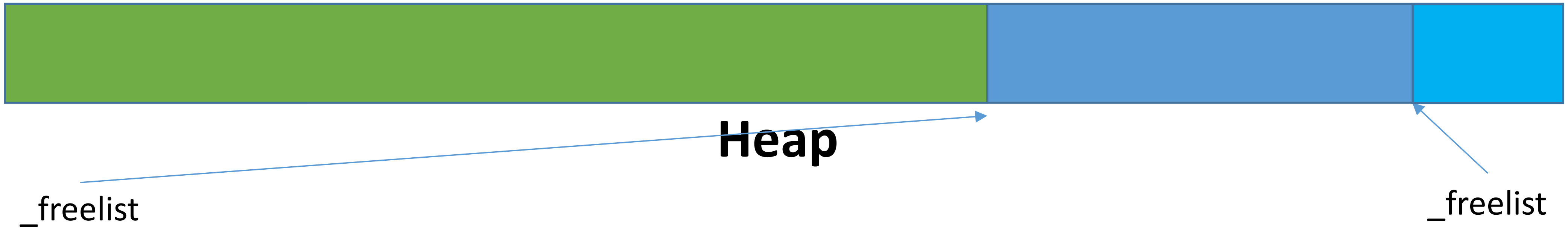
-Xgcpolicy:optavgpause heap

- Allocation is the same as with optthruput



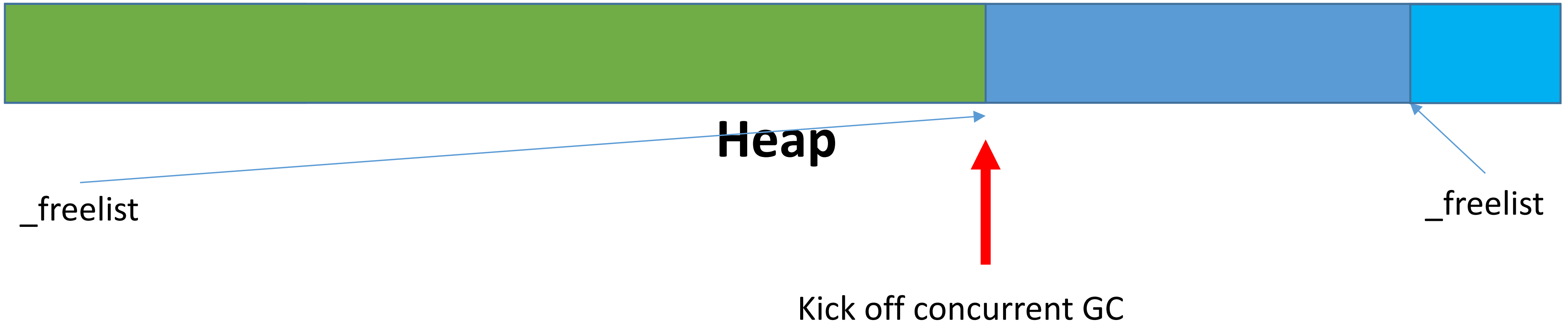
-Xgcpolicy:optavgpause heap

- Allocation is the same as with optthruput



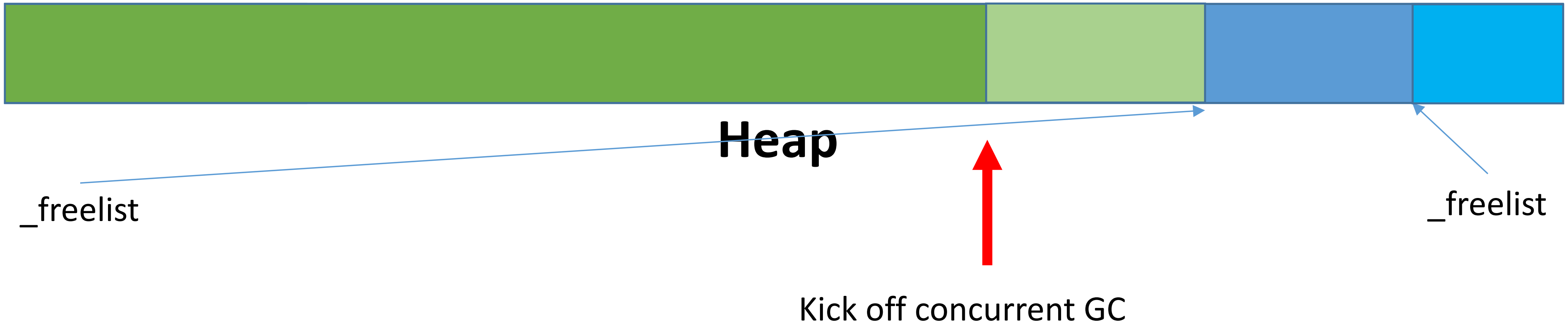
-Xgcpolicy:optavgpause heap

- Allocation is the same as with optthruput
- Start a concurrent GC before the heap memory is exhausted



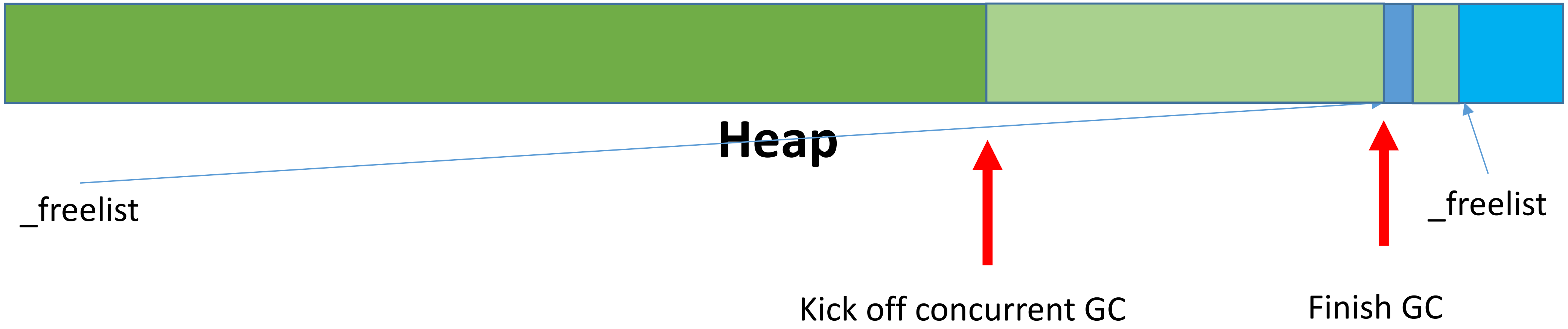
-Xgcpolicy:optavgpause heap

- Allocation is the same as with optthruput
- Start a concurrent GC before the heap memory is exhausted
- Application runs during the GC so allocations continue

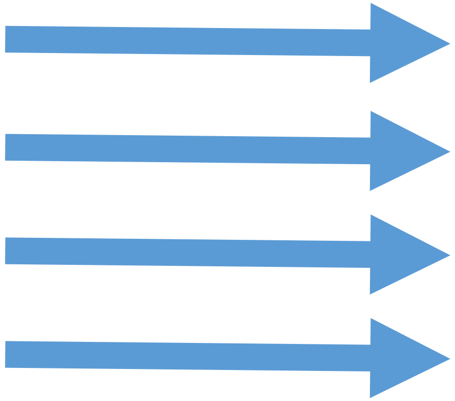


-Xgcpolicy:optavgpause heap

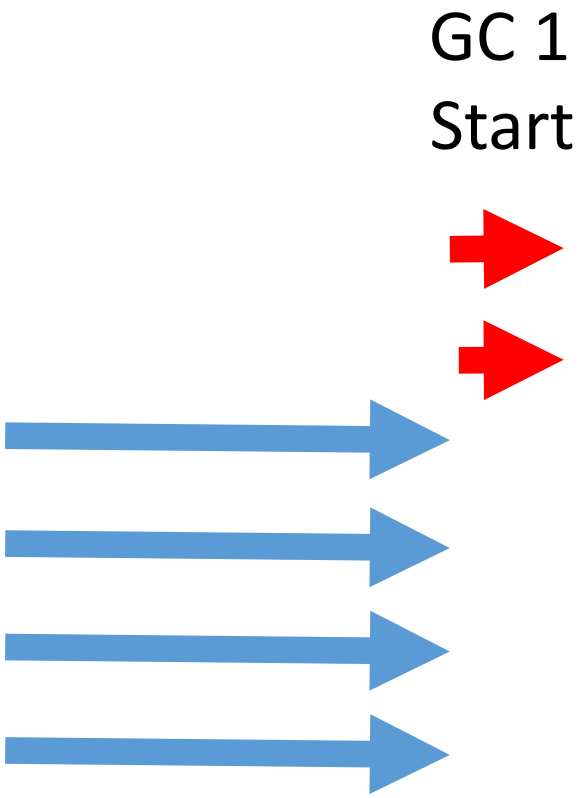
- Allocation is the same as with optthruput
- Start a concurrent GC before the heap memory is exhausted
- Application runs during the GC so allocations continue



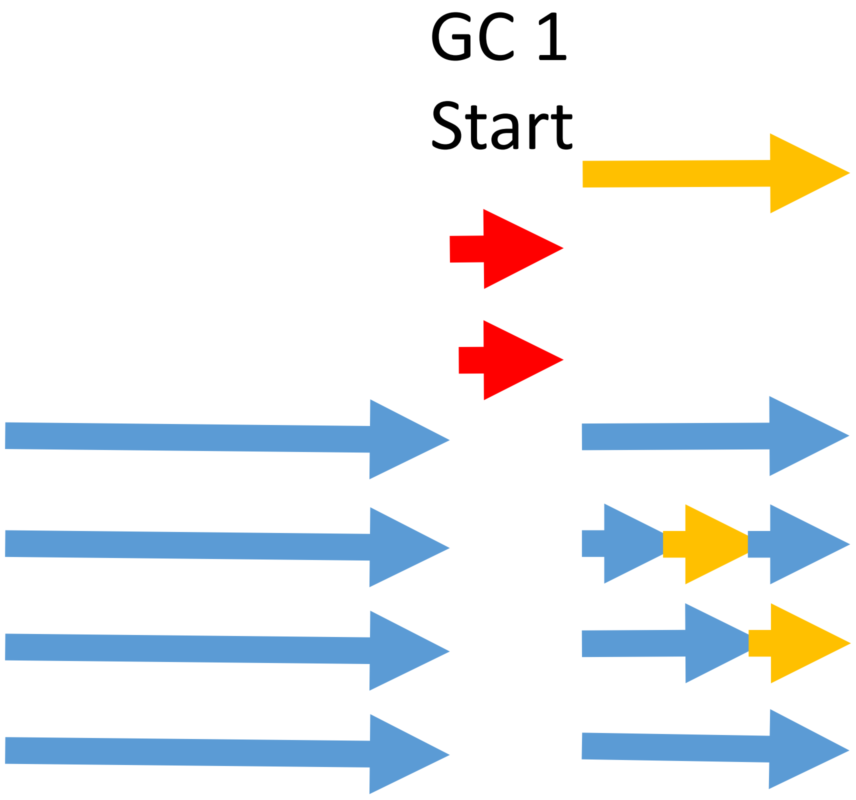
-Xgcpolicy:optavgpause GC



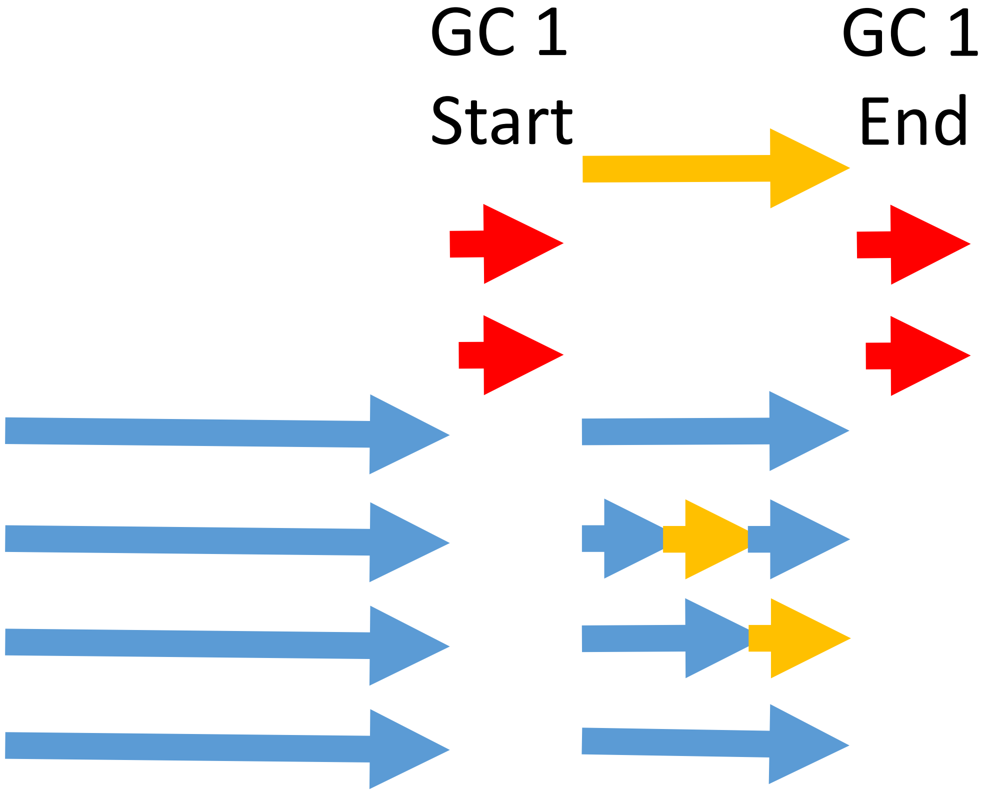
-Xgcpolicy:optavgpause GC



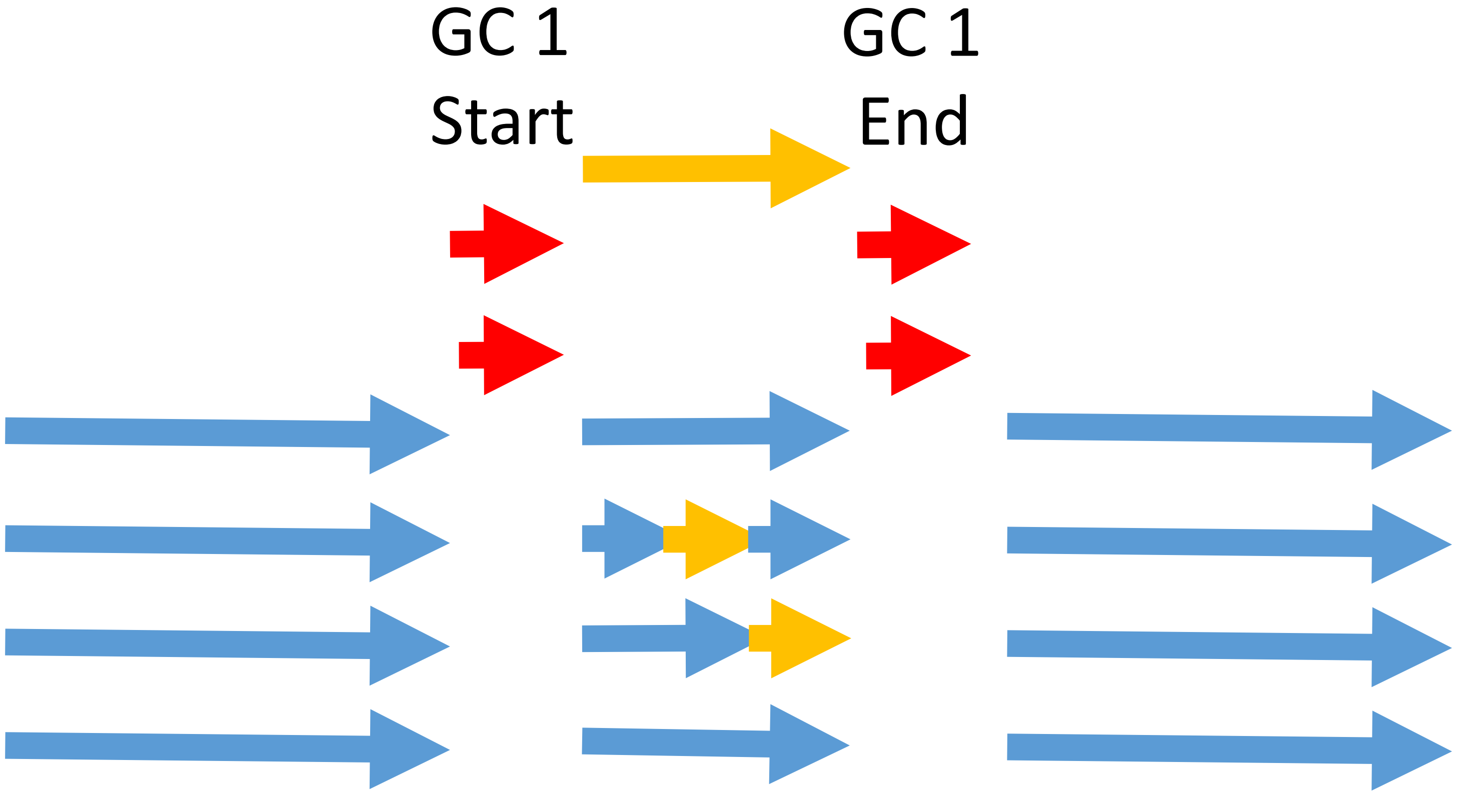
-Xgcpolicy:optavgpause GC



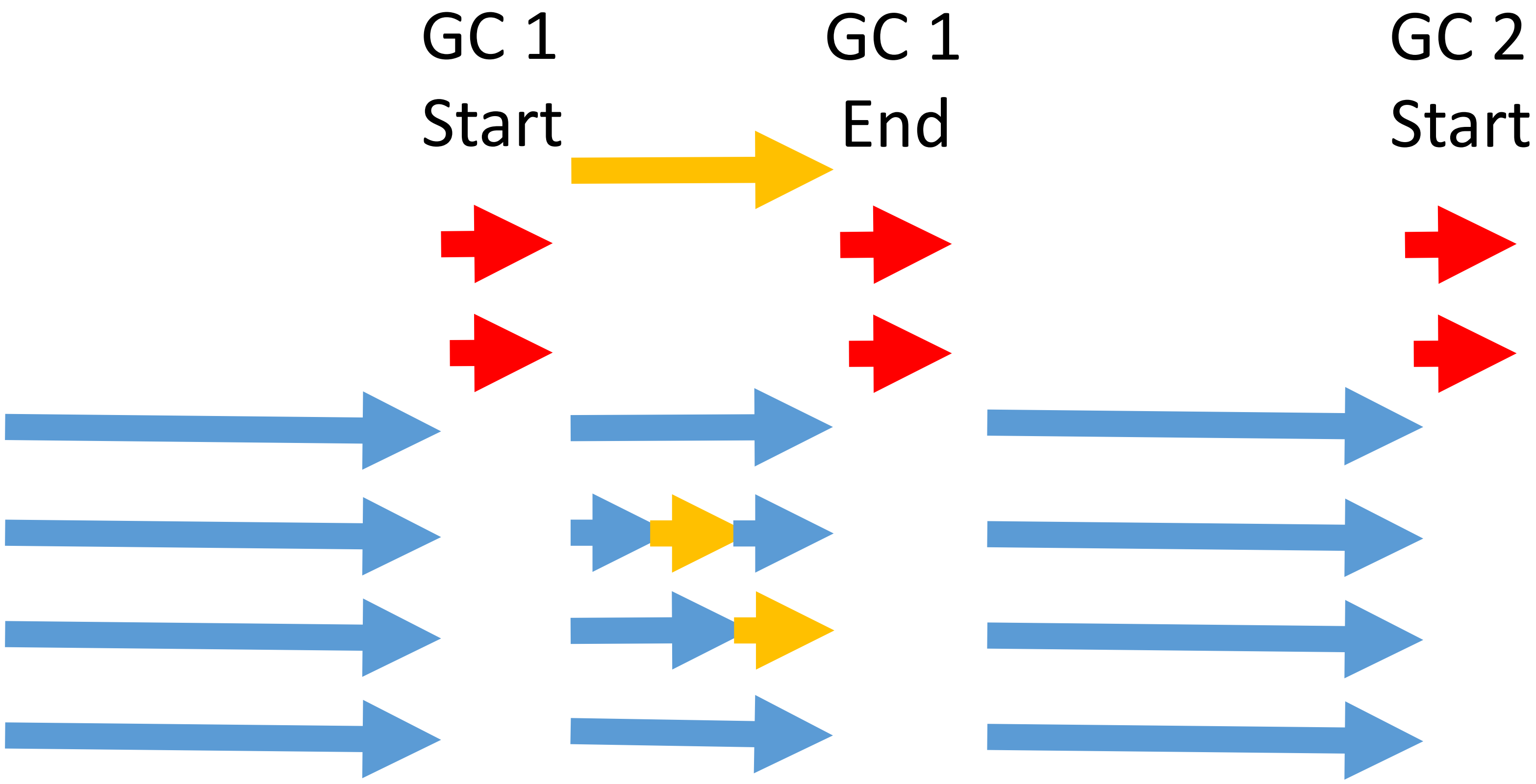
-Xgcpolicy:optavgpause GC



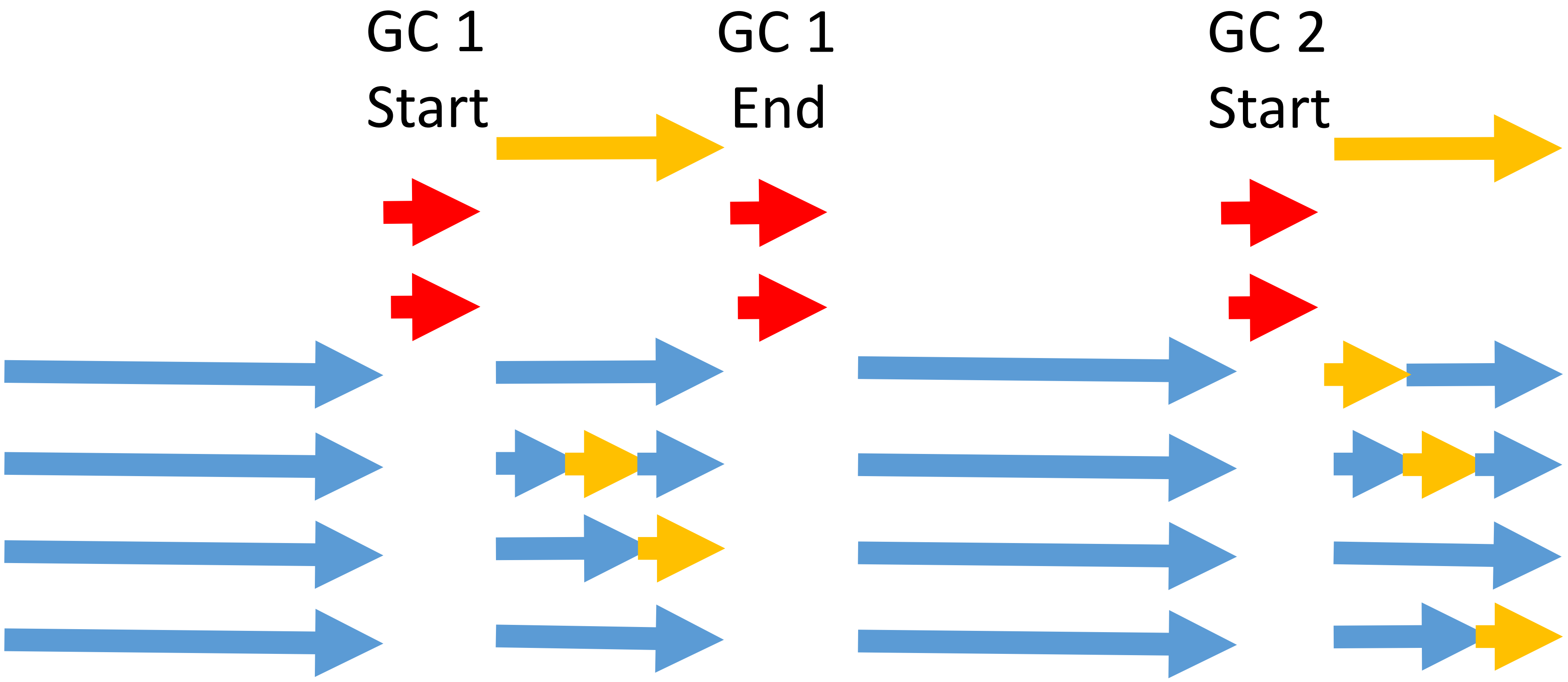
-Xgcpolicy:optavgpause GC



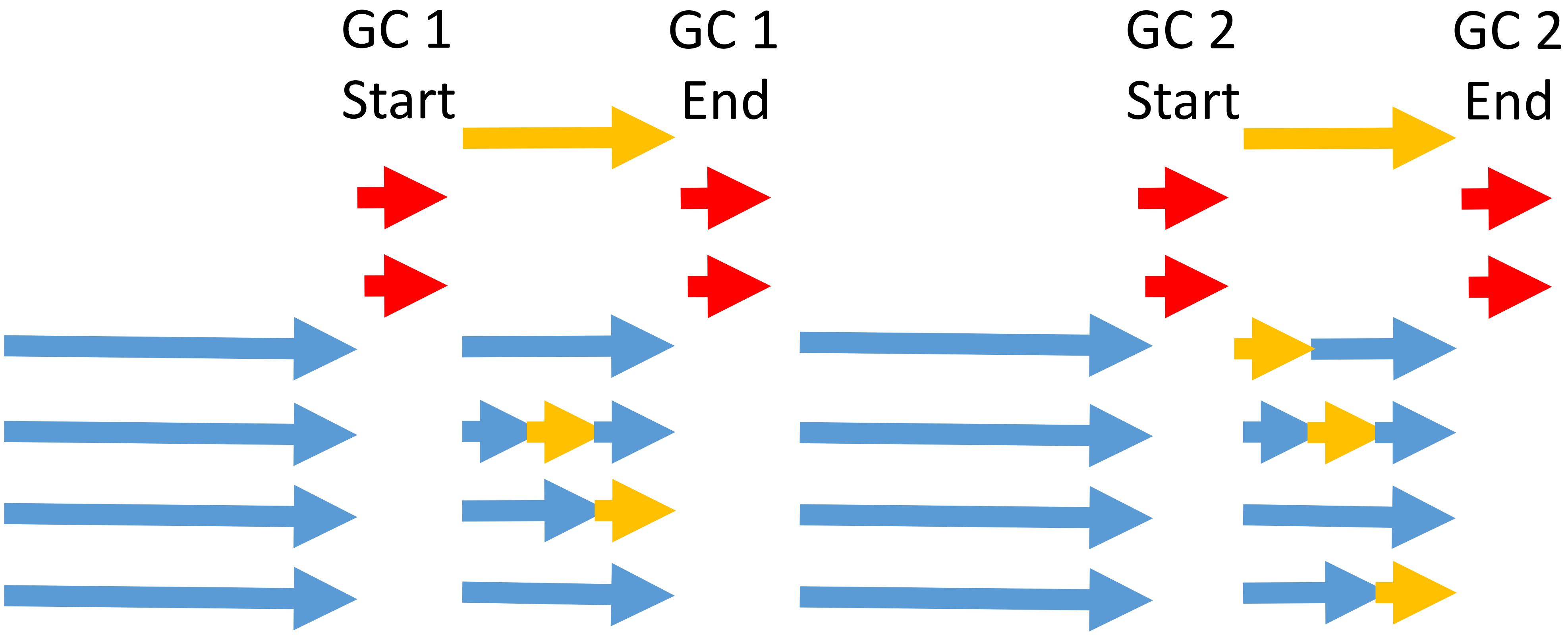
-Xgcpolicy:optavgpause GC



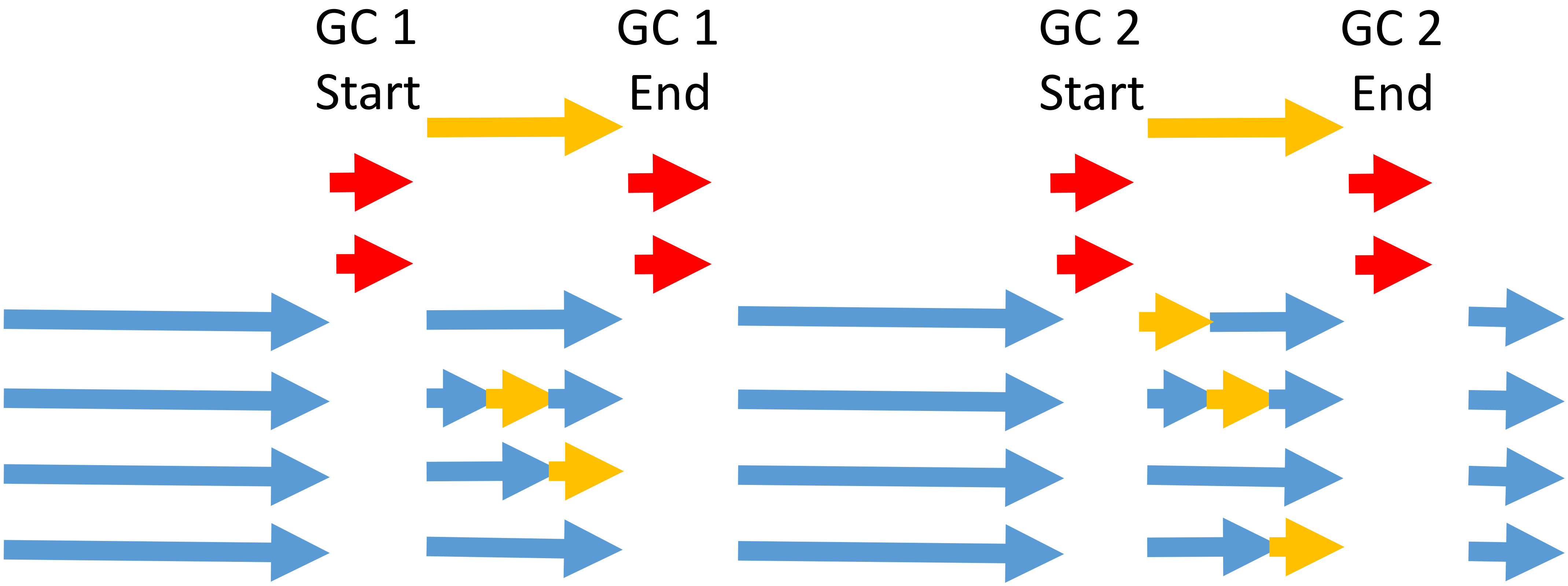
-Xgcpolicy:optavgpause GC



-Xgcpolicy:optavgpause GC



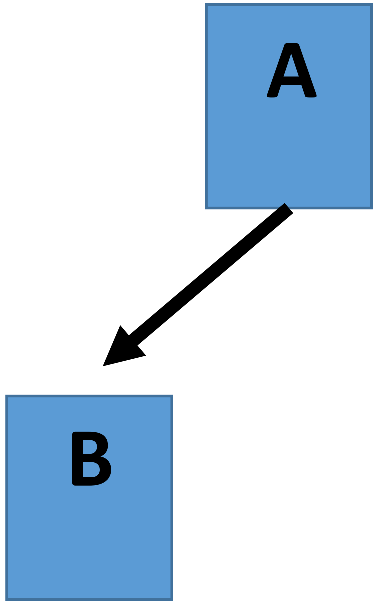
-Xgcpolicy:optavgpause GC



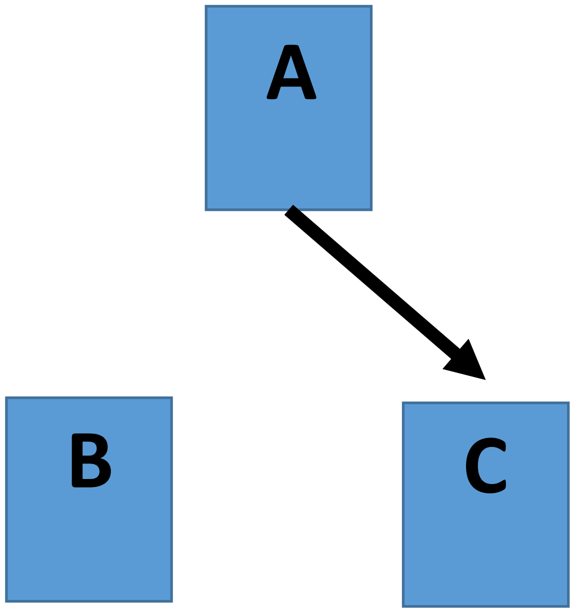
-Xgcpolicy:optavgpause write barrier

- Why is there a write barrier required?

-Xgcpolicy:optavgpause write barrier



-Xgcpolicy:optavgpause write barrier



-Xgcpolicy:optavgpause write barrier

- How is the write barrier implemented?

-Xgcpolicy:optavgpause write barrier

- How is the write barrier implemented?

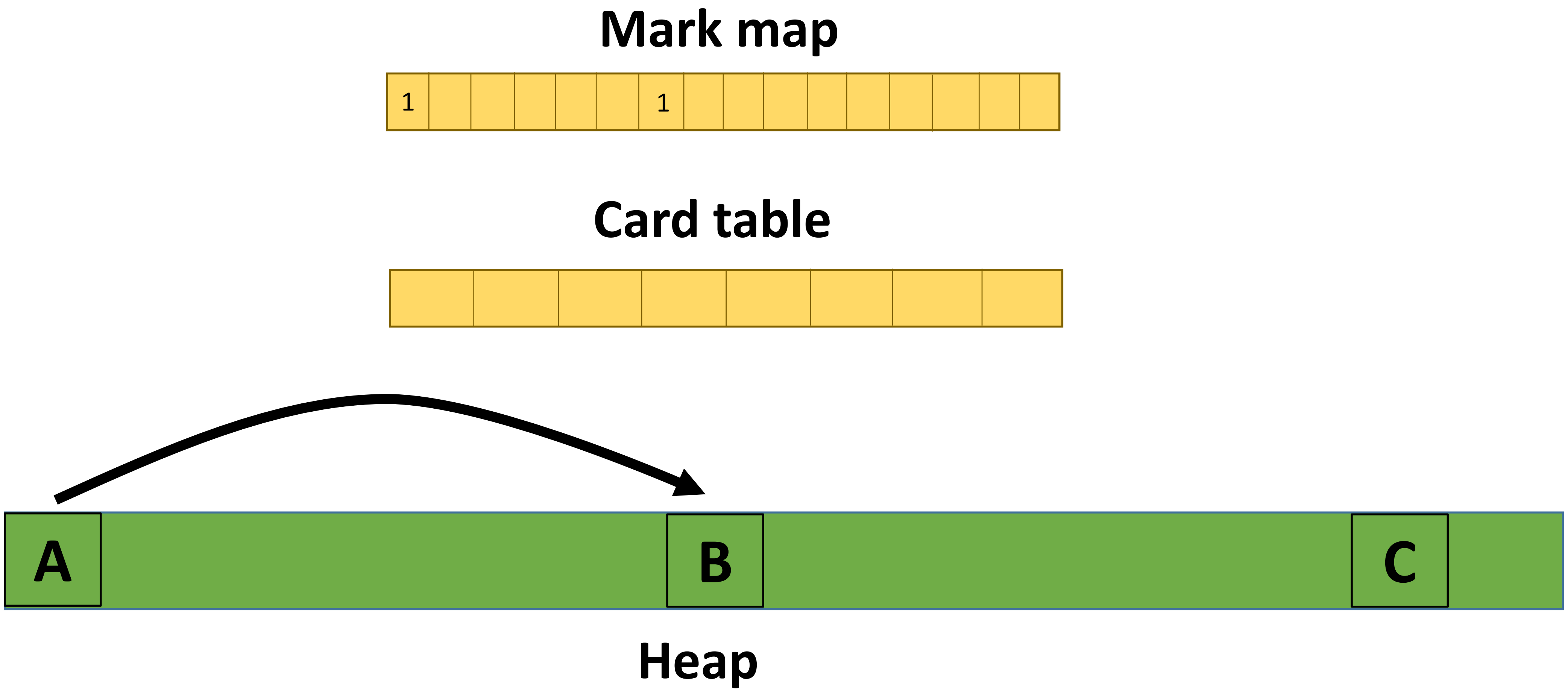
```
private void setField(Object A, Object C) {
   A.field1 = C;
}
```

-Xgcpolicy:optavgpause write barrier

- How is the write barrier implemented?

```
private void setField(Object A, Object C) {
    A.field1 = C;
    if (concurrentGCActive) {
        cardTable->dirtyCard(A);
    }
}
```

-Xgcpolicy:optavgpause write barrier



Mark map

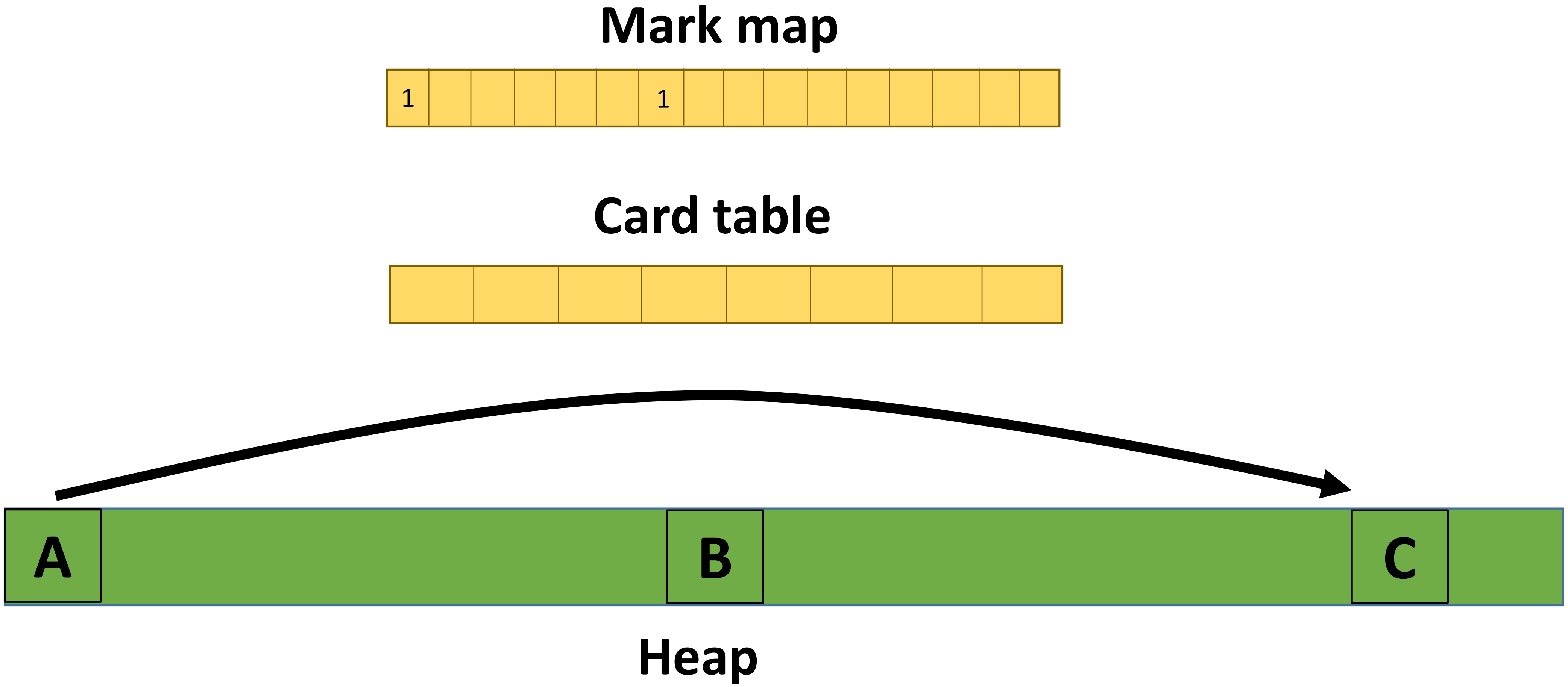
1 1

Card table

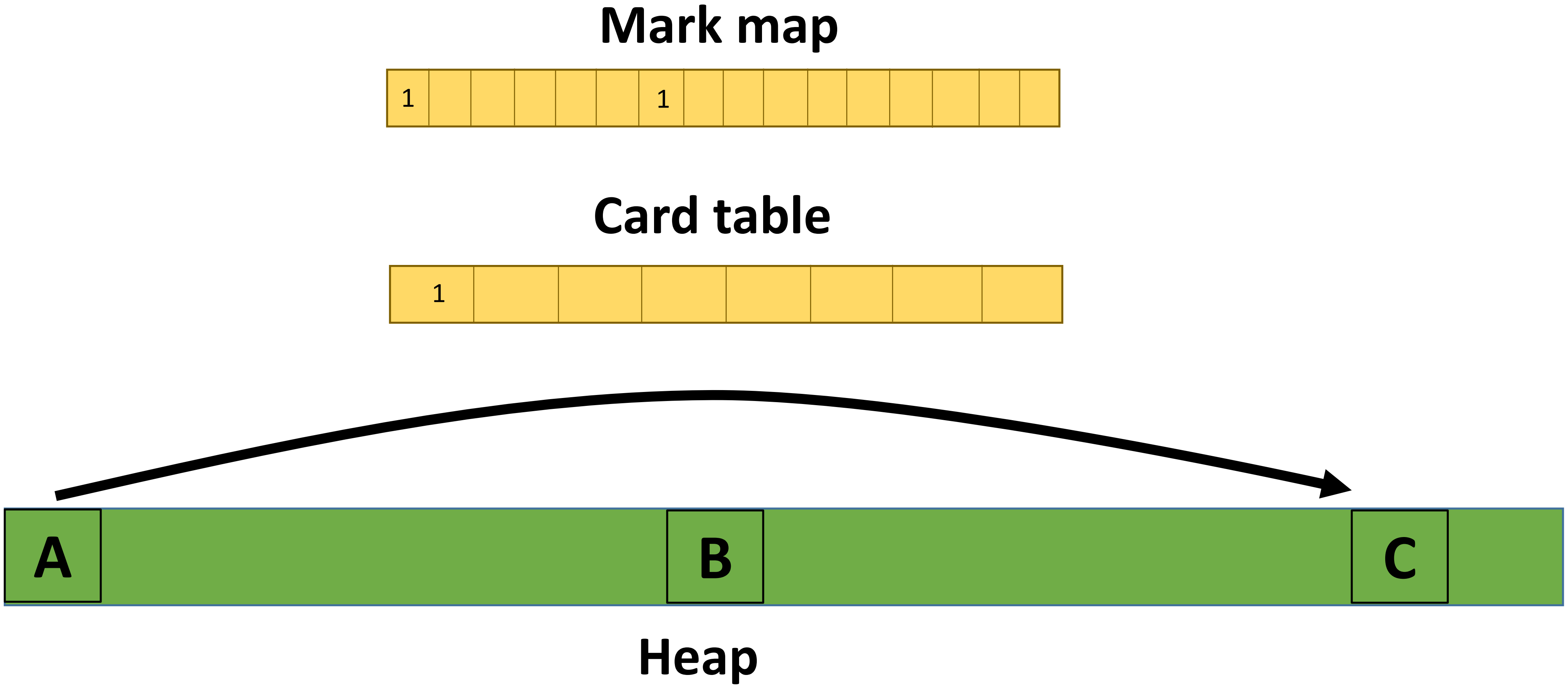
A B C

Heap

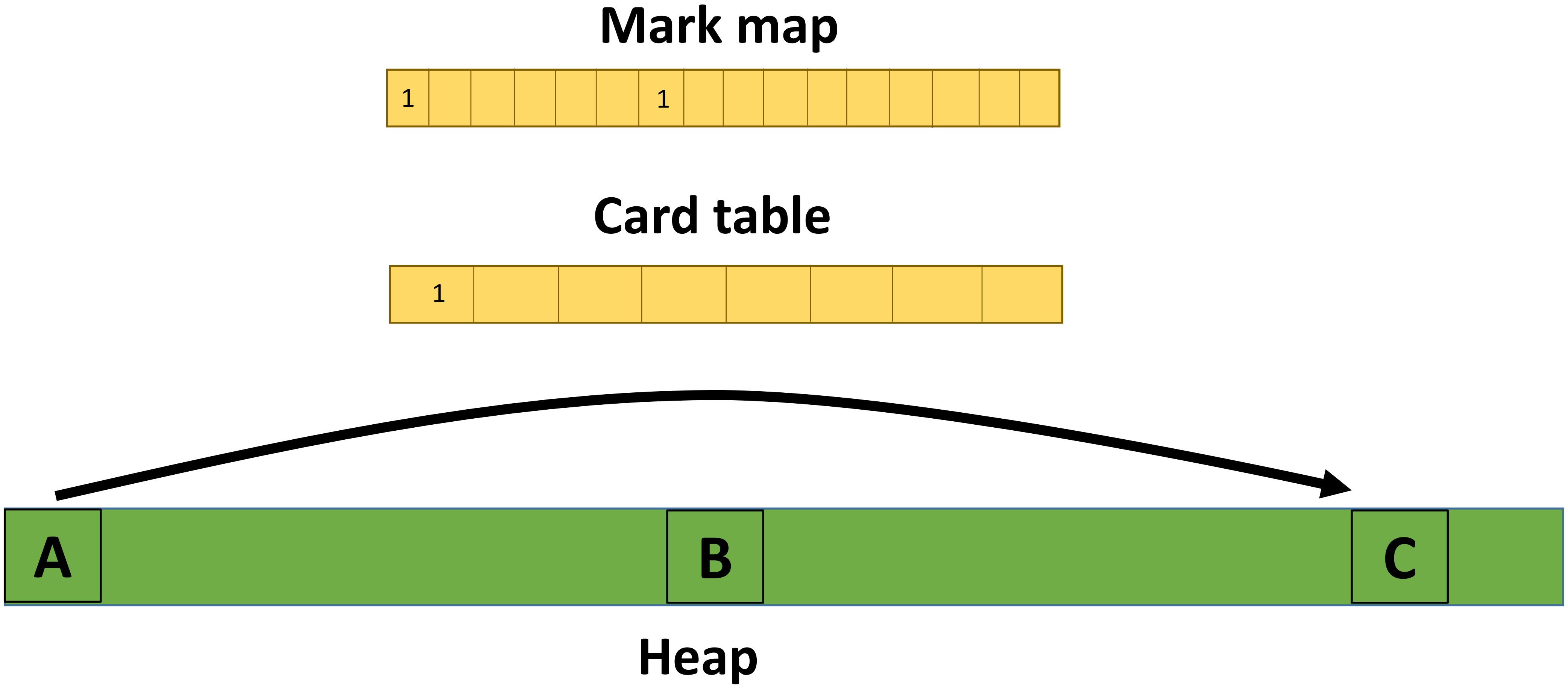
-Xgcpolicy:optavgpause write barrier



-Xgcpolicy:optavgpause write barrier



-Xgcpolicy:optavgpause GC

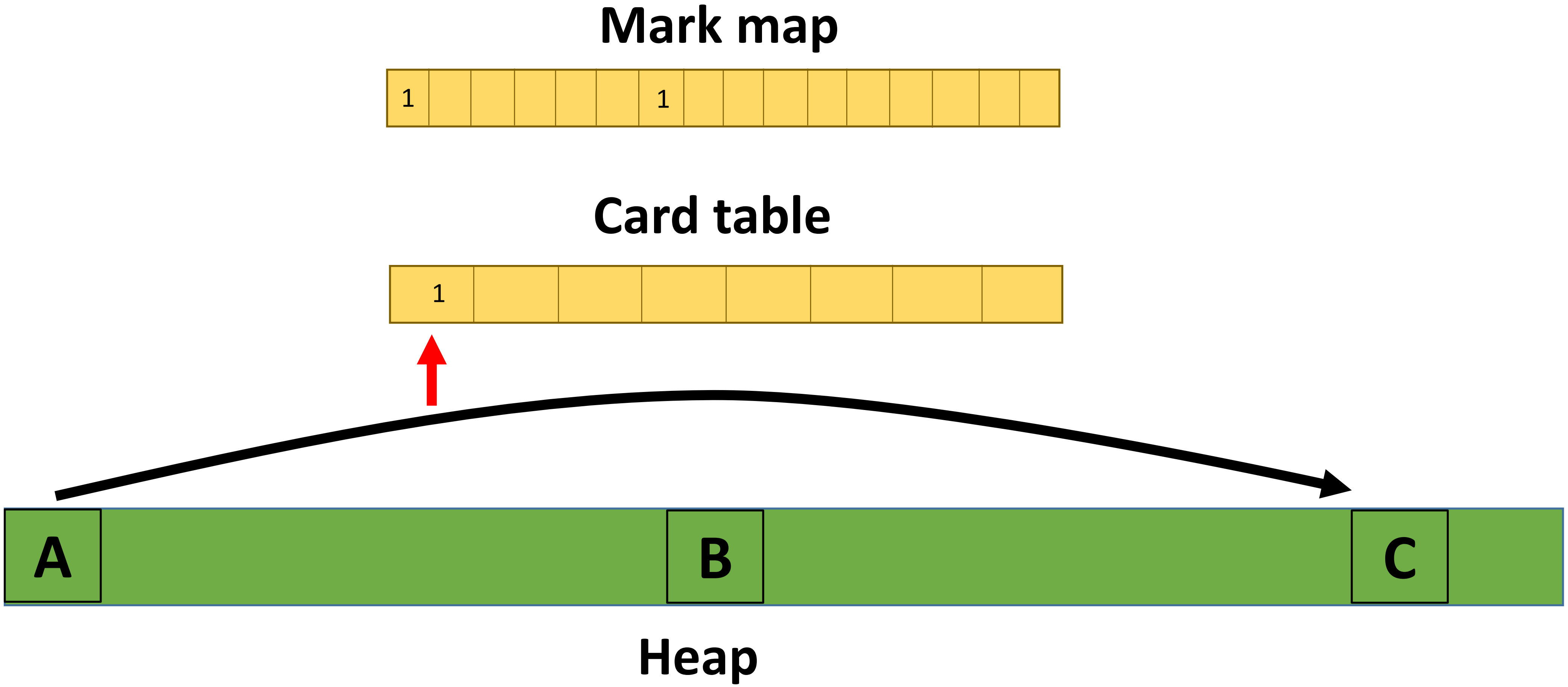


Mark map

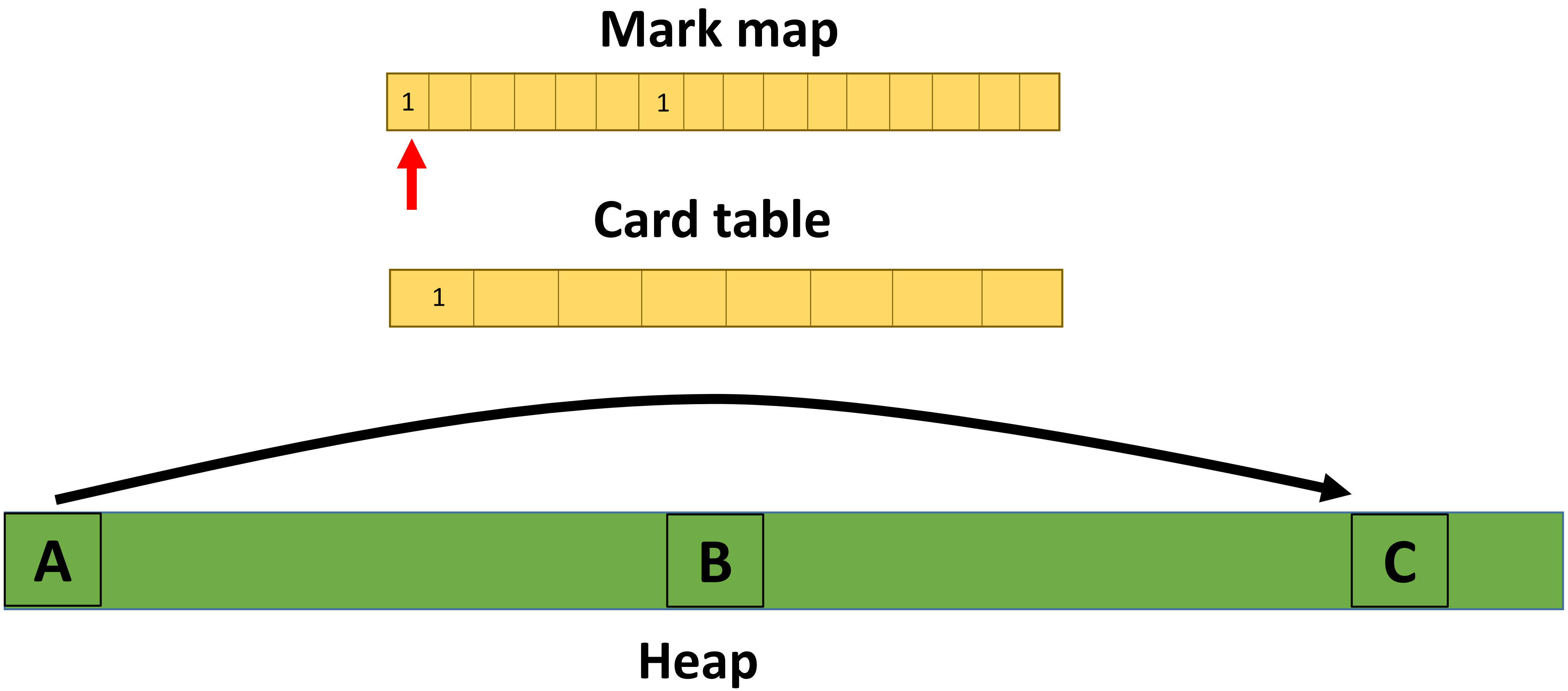
Card table

Heap

-Xgcpolicy:optavgpause GC



-Xgcpolicy:optavgpause GC

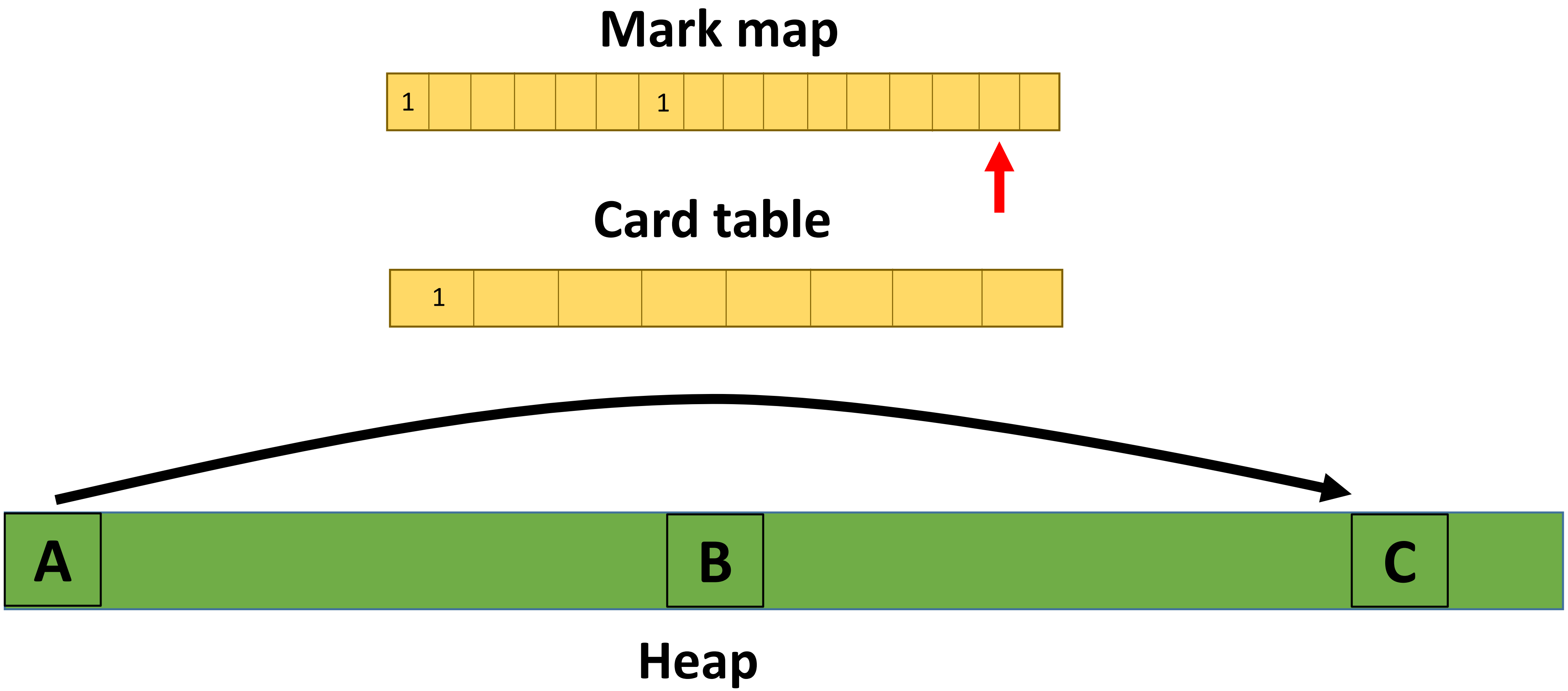


Mark map

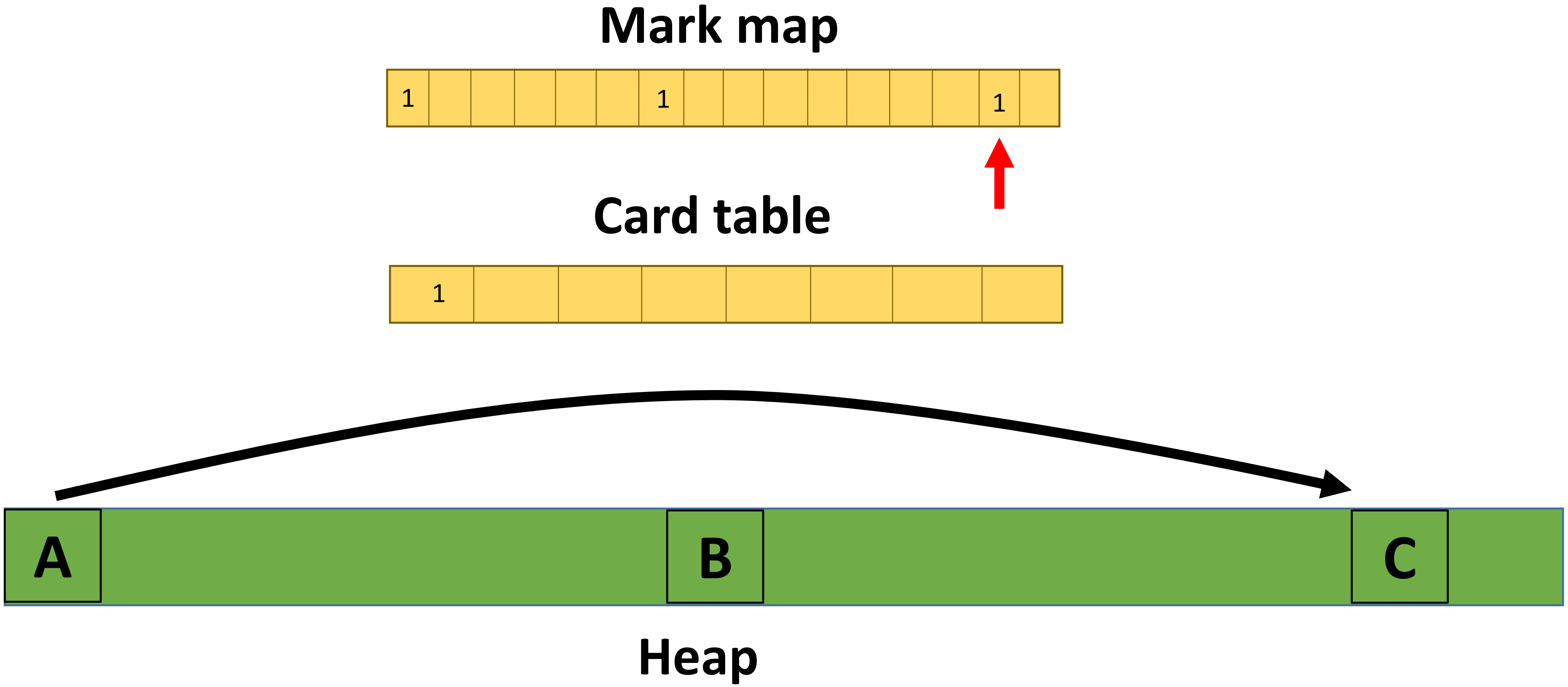
Card table

Heap

-Xgcpolicy:optavgpause GC



-Xgcpolicy:optavgpause GC

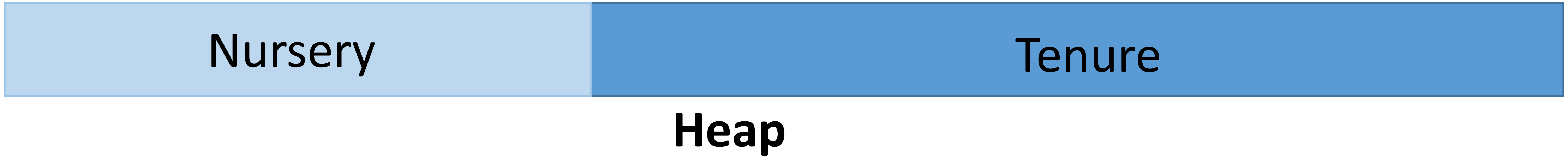


-Xgcpolicy:gencon

- Generational copy collector
  - Focuses collection on a small part of the heap
- Provides a significant reduction in GC STW pause times
- Introduces another write barrier for the remembered set
- GC native memory overhead for mark map, work packets, card table and **copy scan caches**
- Concurrent Global collector from optavgpause

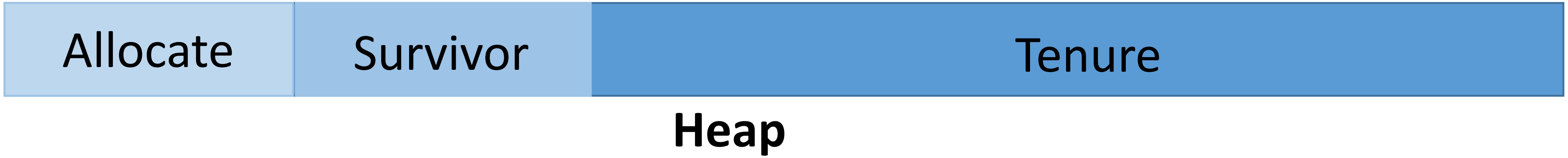
-Xgcpolicy:gencon heap

- Heap is divided into Nursery and Tenure areas
  - Generally the Nursery is smaller than the Tenure area



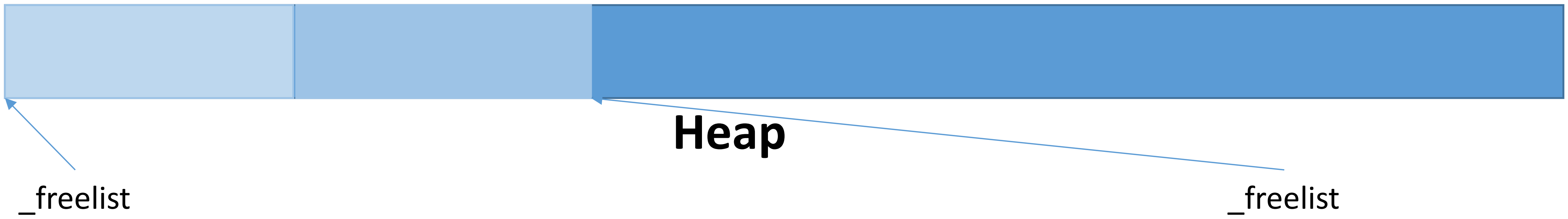
-Xgcpolicy:gencon heap

- Heap is divided into Nursery and Tenure spaces
  - Generally the Nursery is smaller than Tenure
- The Nursery is divided into 2 logical spaces: Allocate and Survivor
  - Objects are allocated in Allocate space*



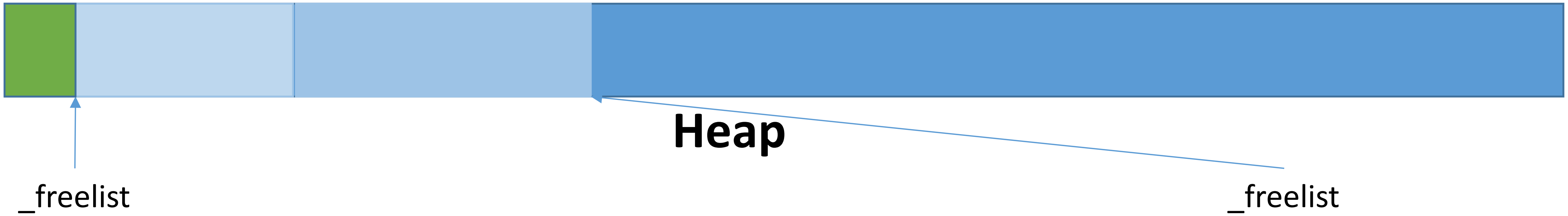
-Xgcpolicy:gencon heap

- Heap is divided into Nursery and Tenure spaces
  - Generally the Nursery is smaller than Tenure
- The Nursery is divided into 2 logical spaces: Allocate and Survivor
  - Objects are allocated in Allocate space*



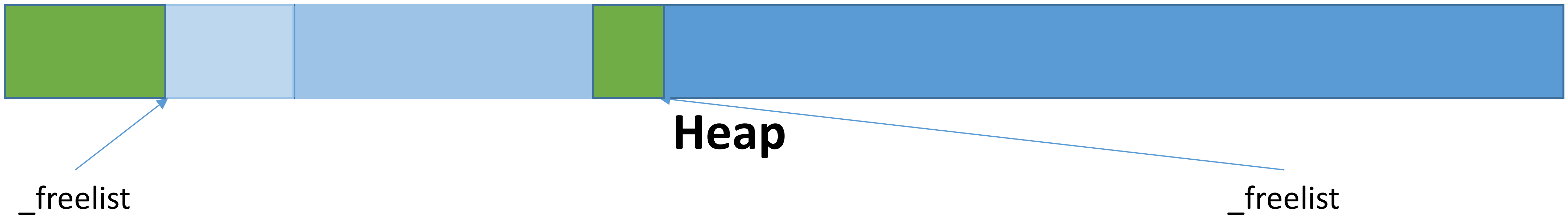
-Xgcpolicy:gencon heap

- Heap is divided into Nursery and Tenure spaces
  - Generally the Nursery is smaller than Tenure
- The Nursery is divided into 2 logical spaces: Allocate and Survivor
  - Objects are allocated in Allocate space*



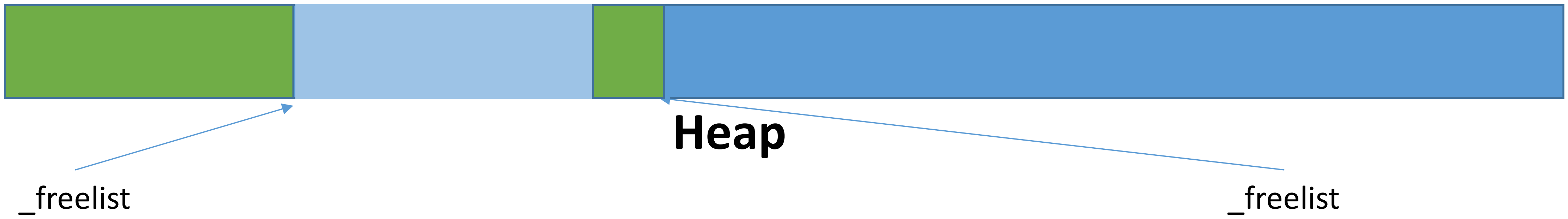
-Xgcpolicy:gencon heap

- Heap is divided into Nursery and Tenure spaces
  - Generally the Nursery is smaller than Tenure
- The Nursery is divided into 2 logical spaces: Allocate and Survivor
  - Objects are allocated in Allocate space*



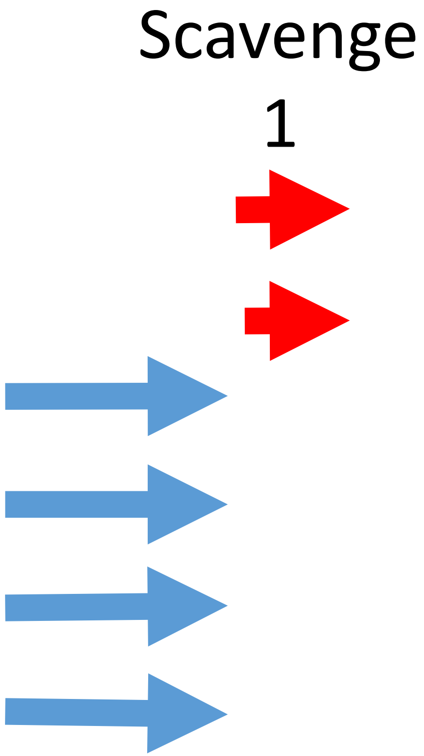
-Xgcpolicy:gencon heap

- Heap is divided into Nursery and Tenure spaces
  - Generally the Nursery is smaller than Tenure
- The Nursery is divided into 2 logical spaces: Allocate and Survivor
  - Objects are allocated in Allocate space*

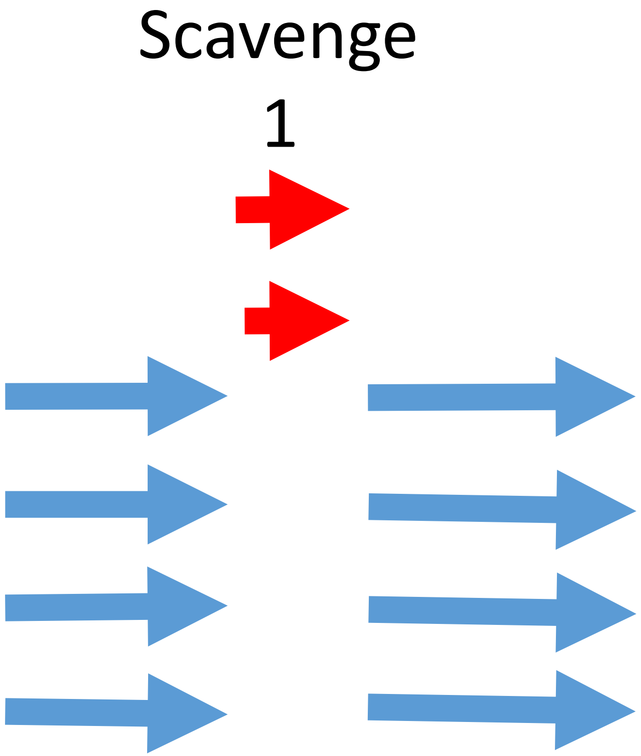


-Xgcpolicy:gencon GC

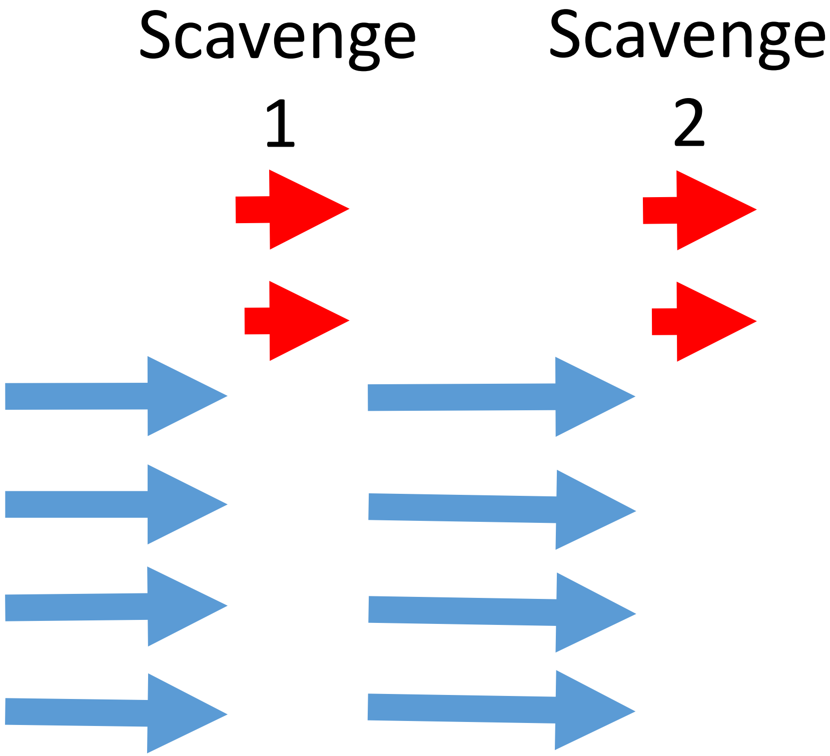
-Xgcpolicy:gencon GC



-Xgcpolicy:gencon GC



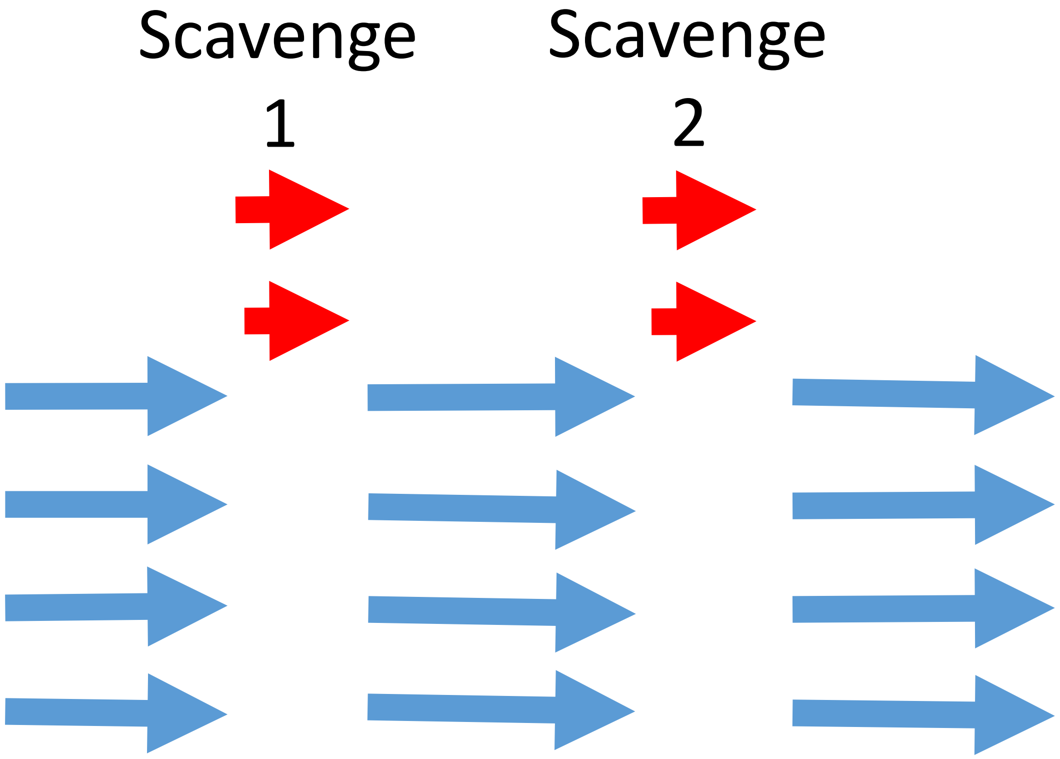
-Xgcpolicy:gencon GC



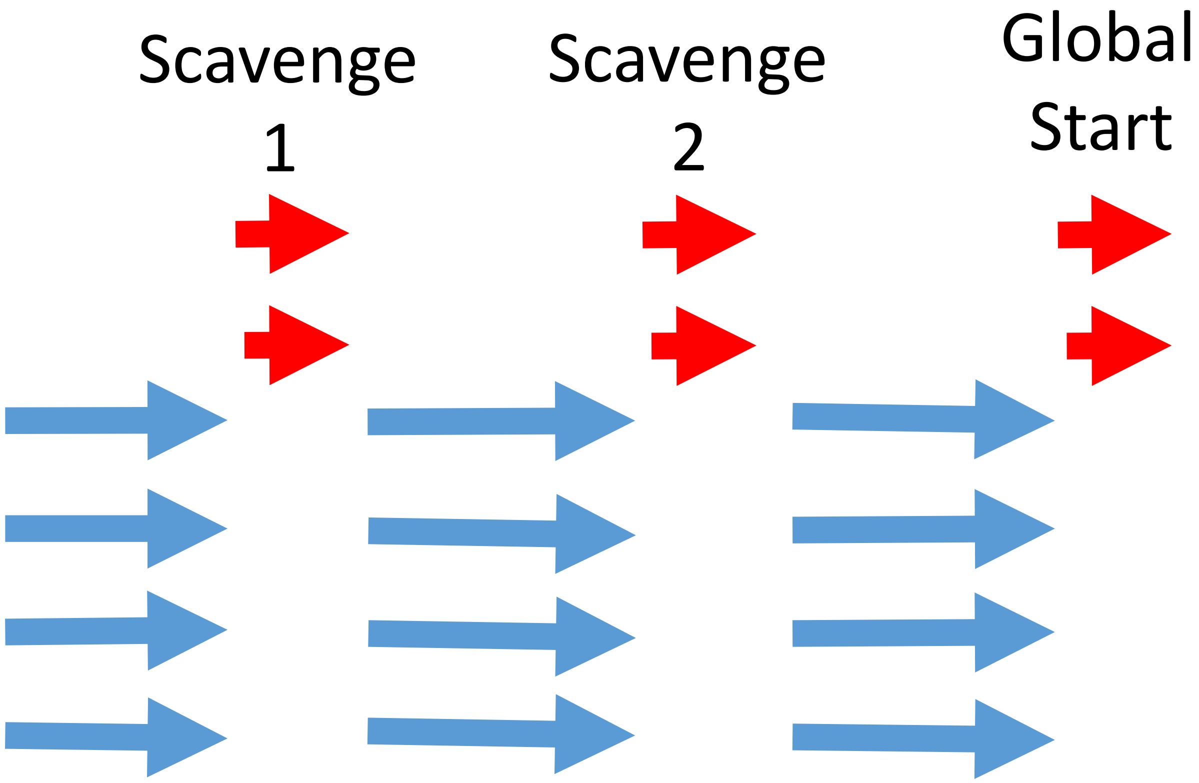
Scavenge 1

Scavenge 2

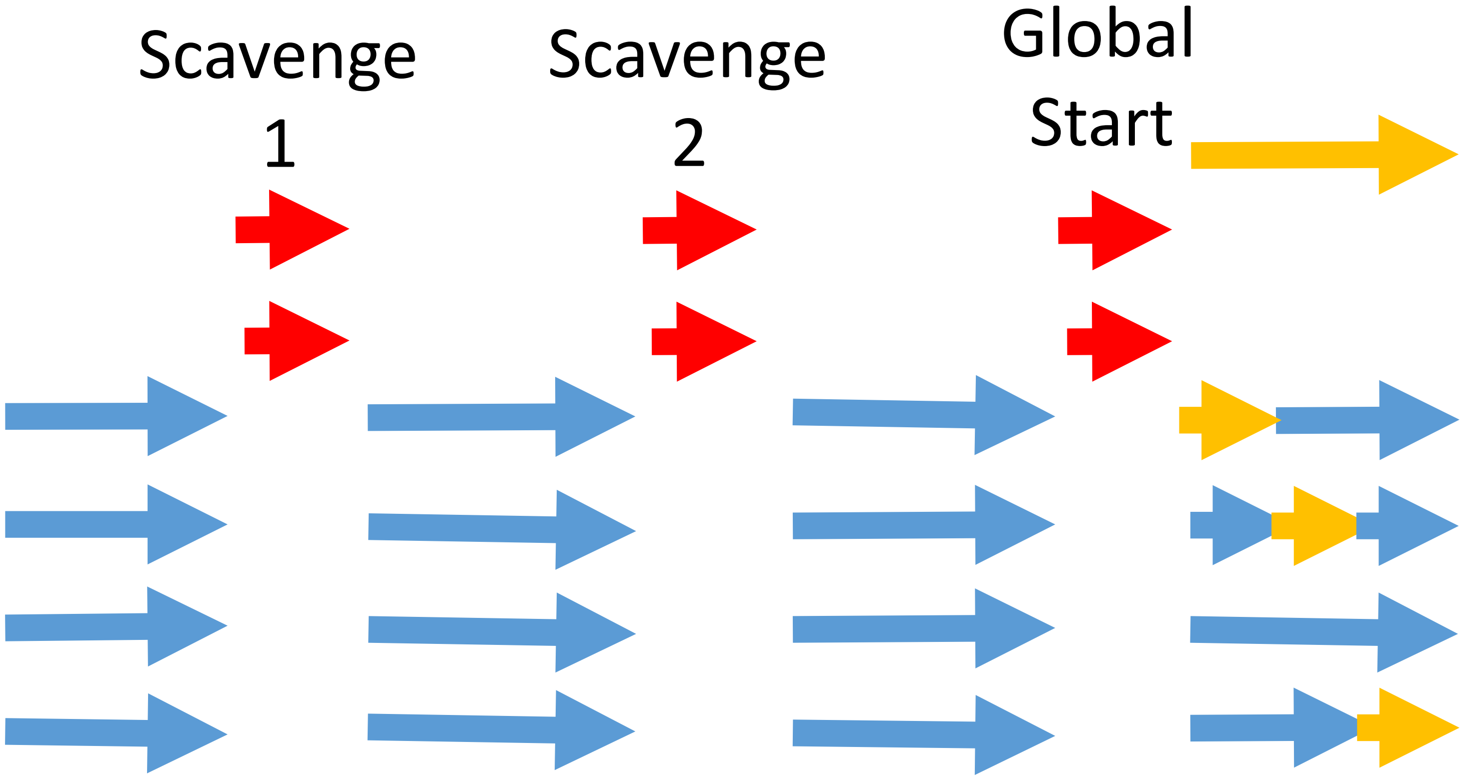
-Xgcpolicy:gencon GC



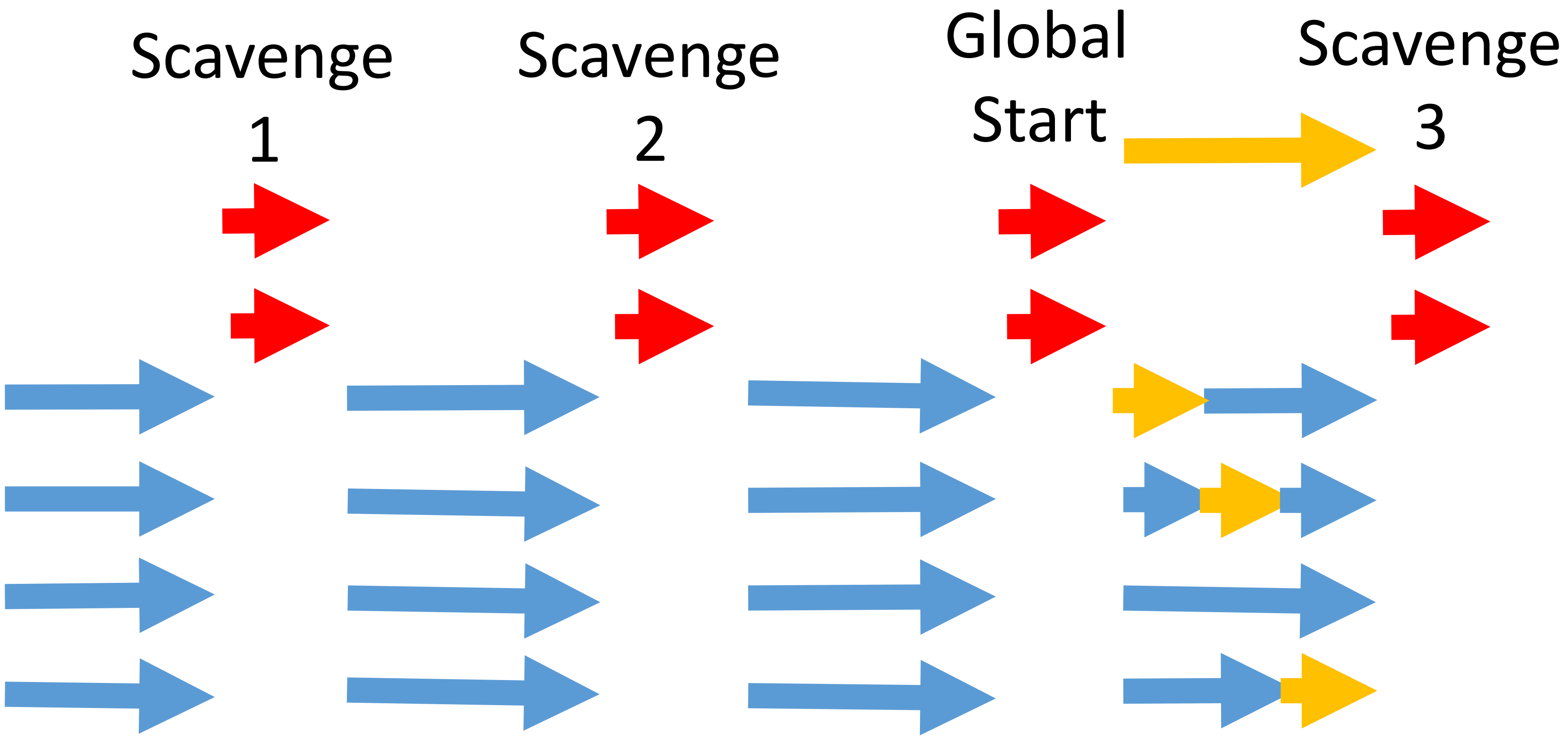
-Xgcpolicy:gencon GC



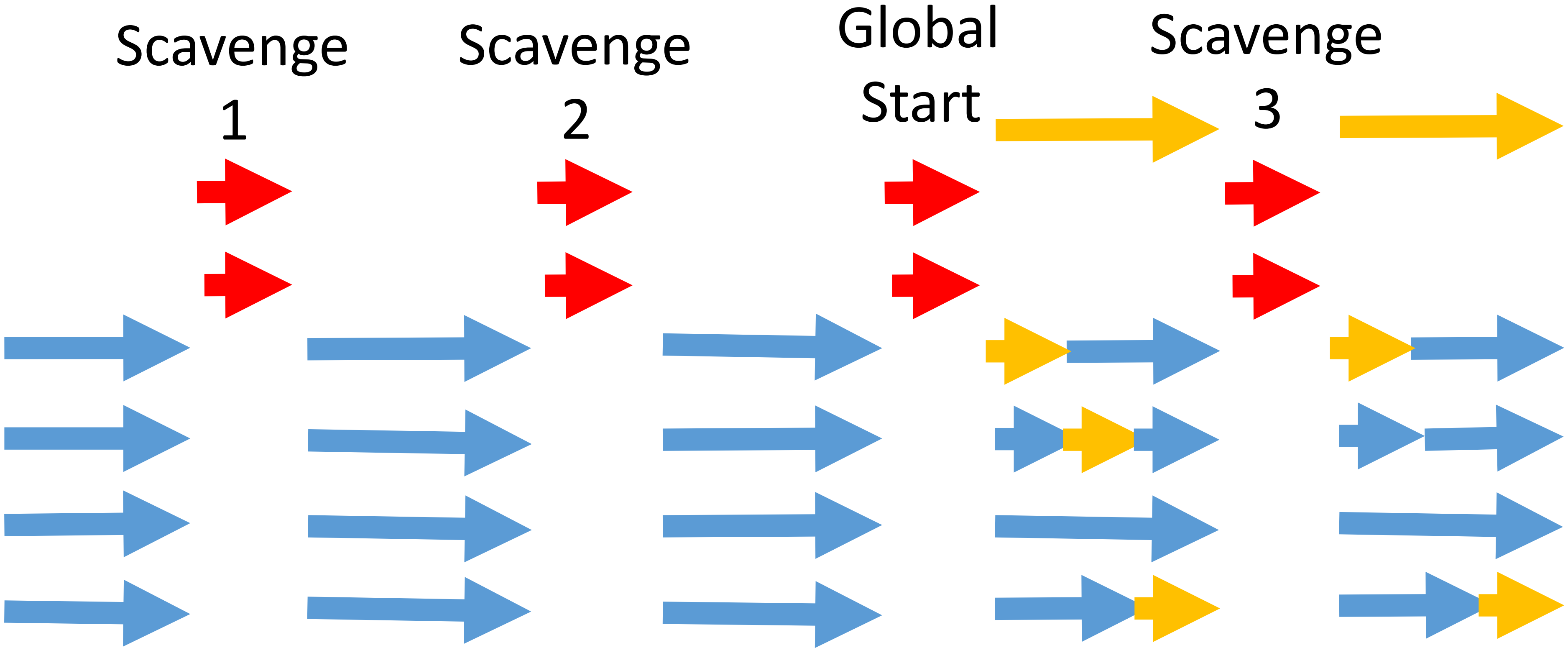
-Xgcpolicy:gencon GC



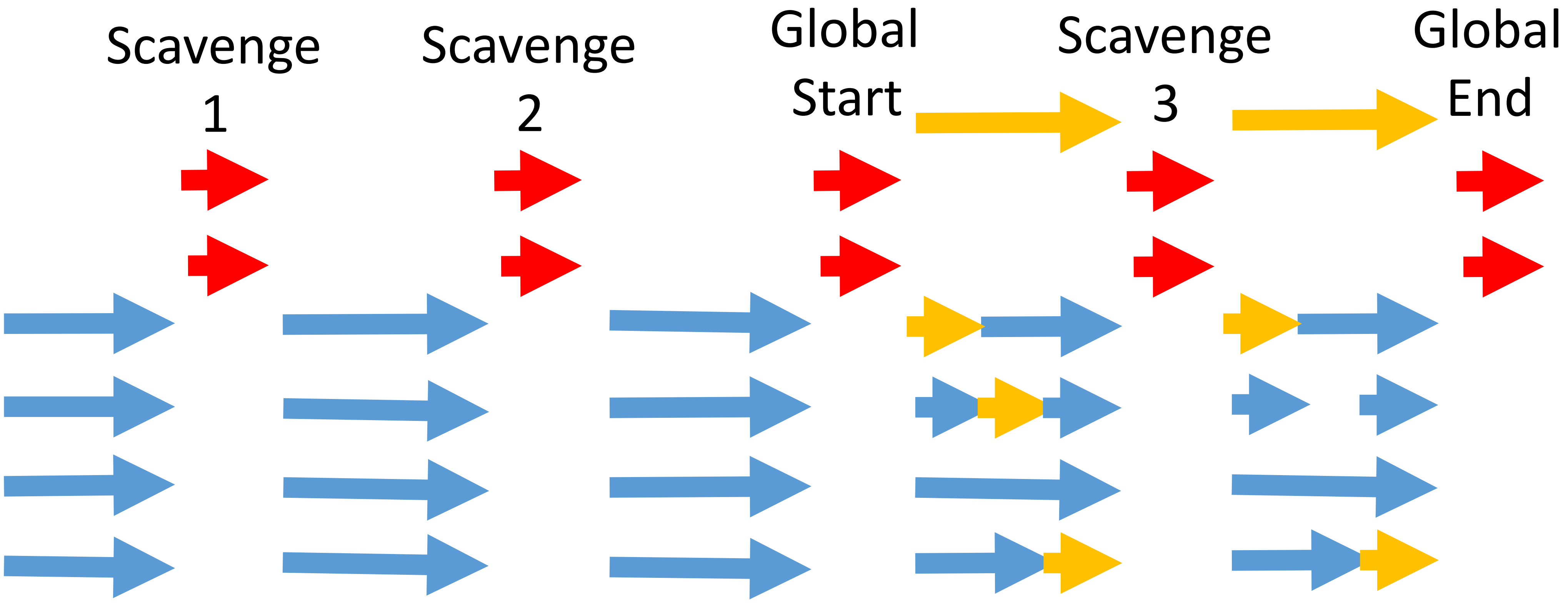
-Xgcpolicy:gencon GC



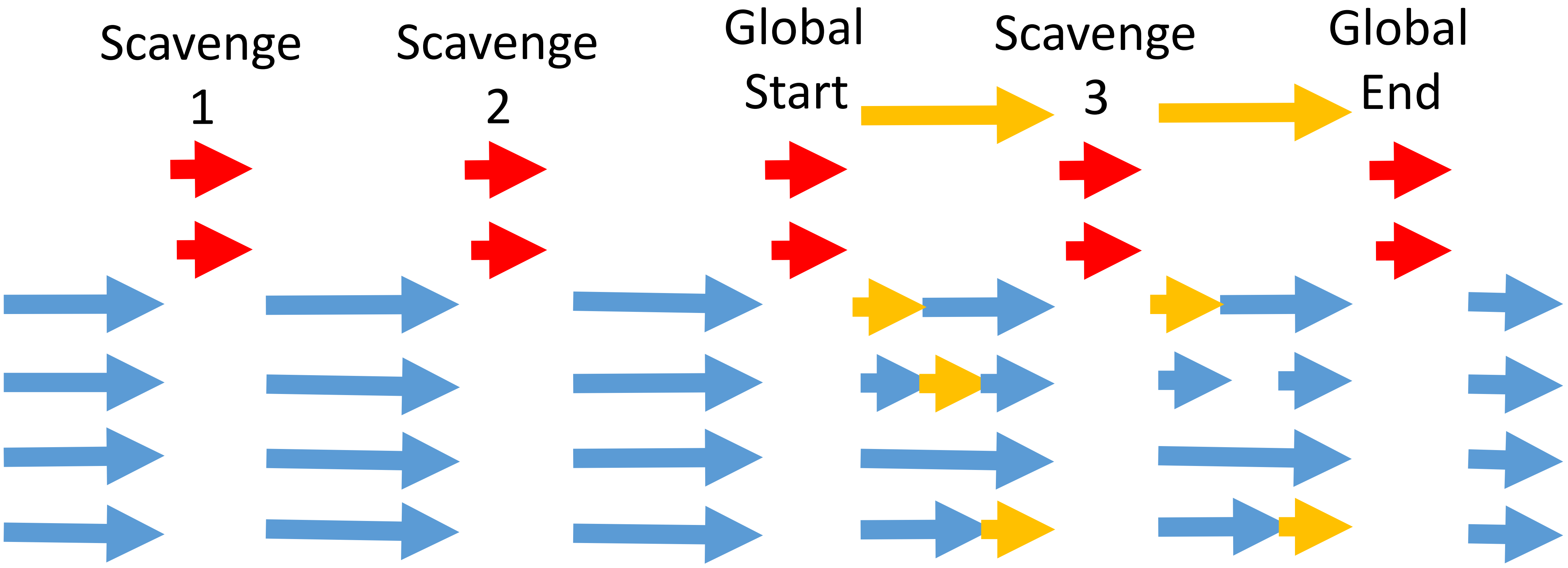
-Xgcpolicy:gencon GC



-Xgcpolicy:gencon GC



-Xgcpolicy:gencon GC



-Xgcpolicy:gencon generational write barrier

- Why is there a write barrier required?

-Xgcpolicy:gencon generational write barrier

- Why is there a write barrier required?

- The GC needs to be able to find objects in the nursery which are only referenced from tenure space

-Xgcpolicy:gencon generational write barrier

- How is the write barrier implemented?

```
private void setField(Object A, Object C) {
   A.field1 = C;
}
```

-Xgcpolicy:gencon generational write barrier

- How is the write barrier implemented?

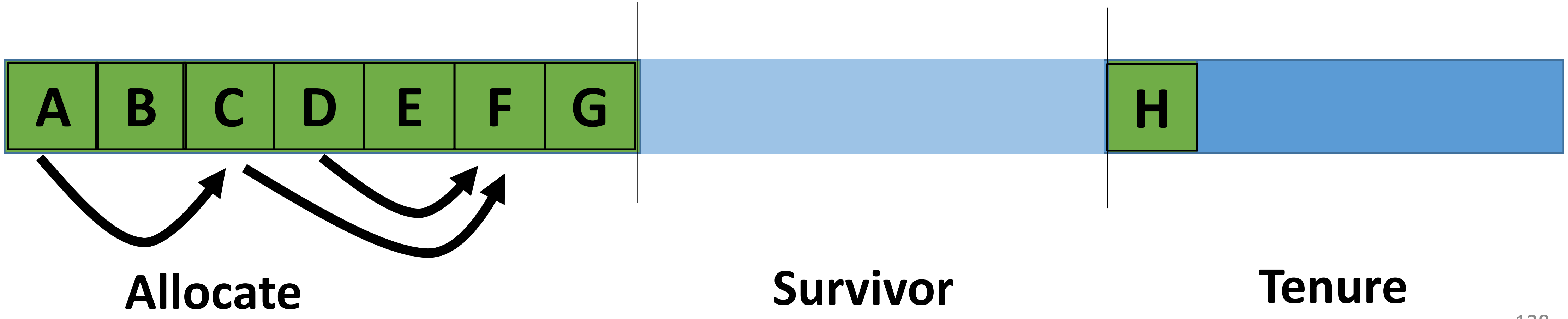
```
private void setField(Object A, Object C) {
    A.field1 = C;
    if (A is tenured) {
       if (C is NOT tenured) {
           remember(A);
       }
    }
}
```

-Xgcpolicy:gencon generational write barrier

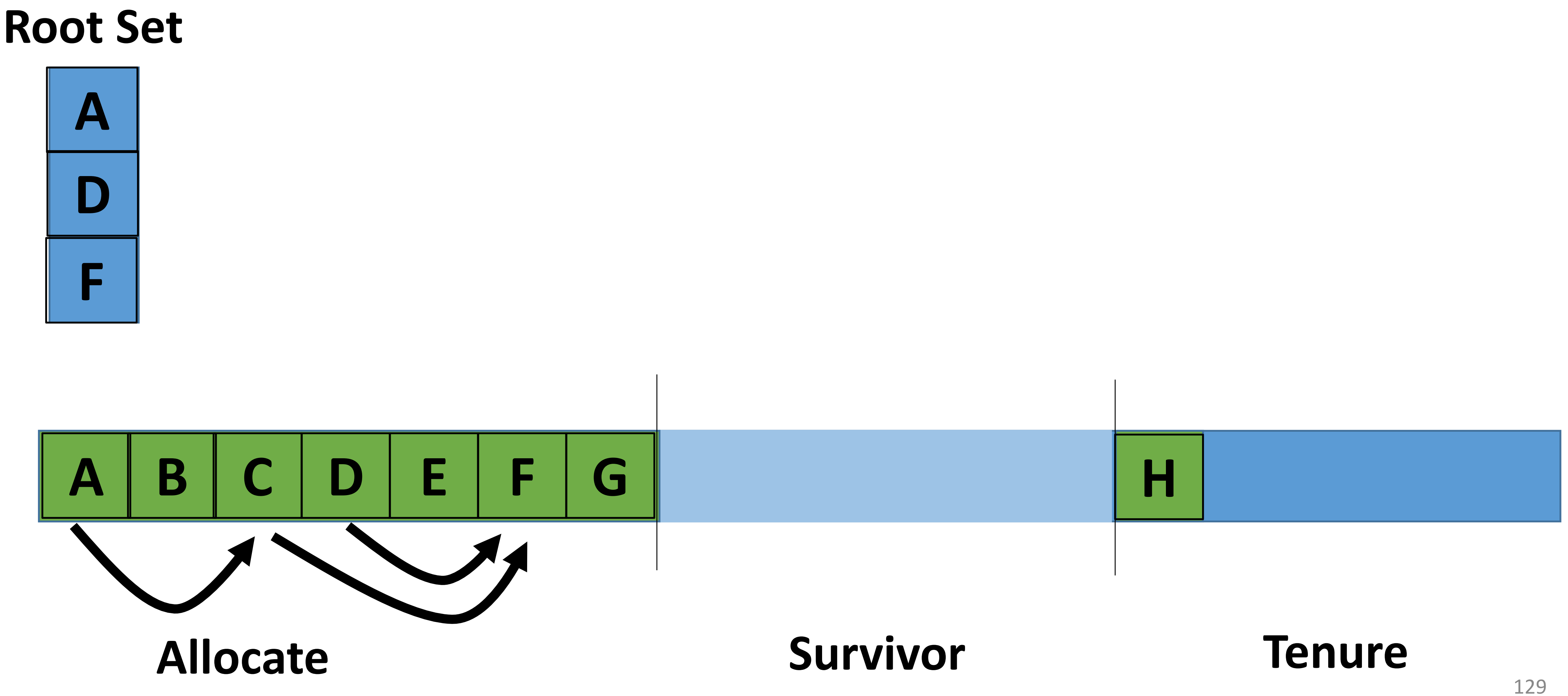
- How is the write barrier implemented?

```
private void setField(Object A, Object C) {
    A.field1 = C;
    if (A is tenured) {
        if (C is NOT tenure) {
            remember(A);
        }
        if (concurrentGCActive) {
            cardTable->dirtyCard(A);
        }
    }
}
```

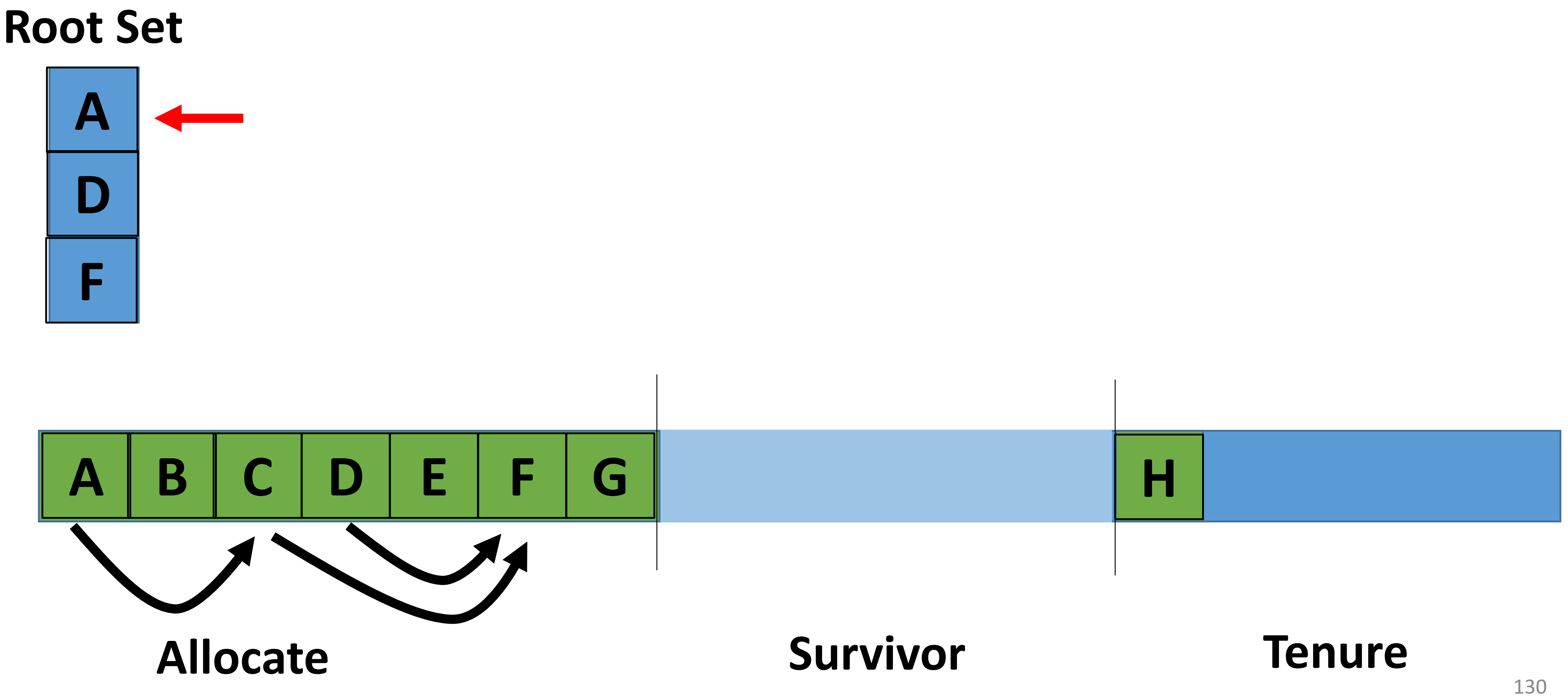
-Xgcpolicy:gencon GC



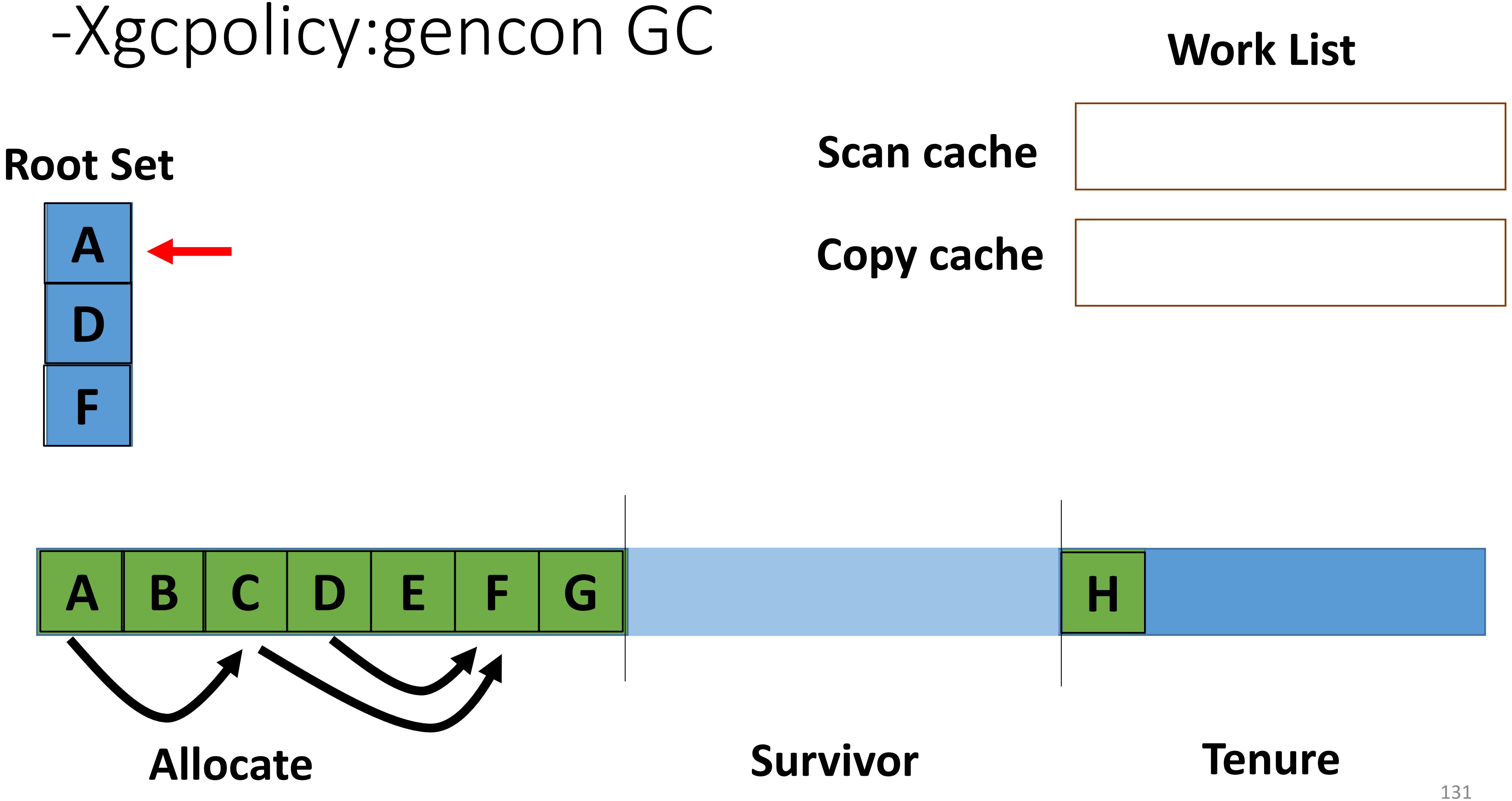
-Xgcpolicy:gencon GC



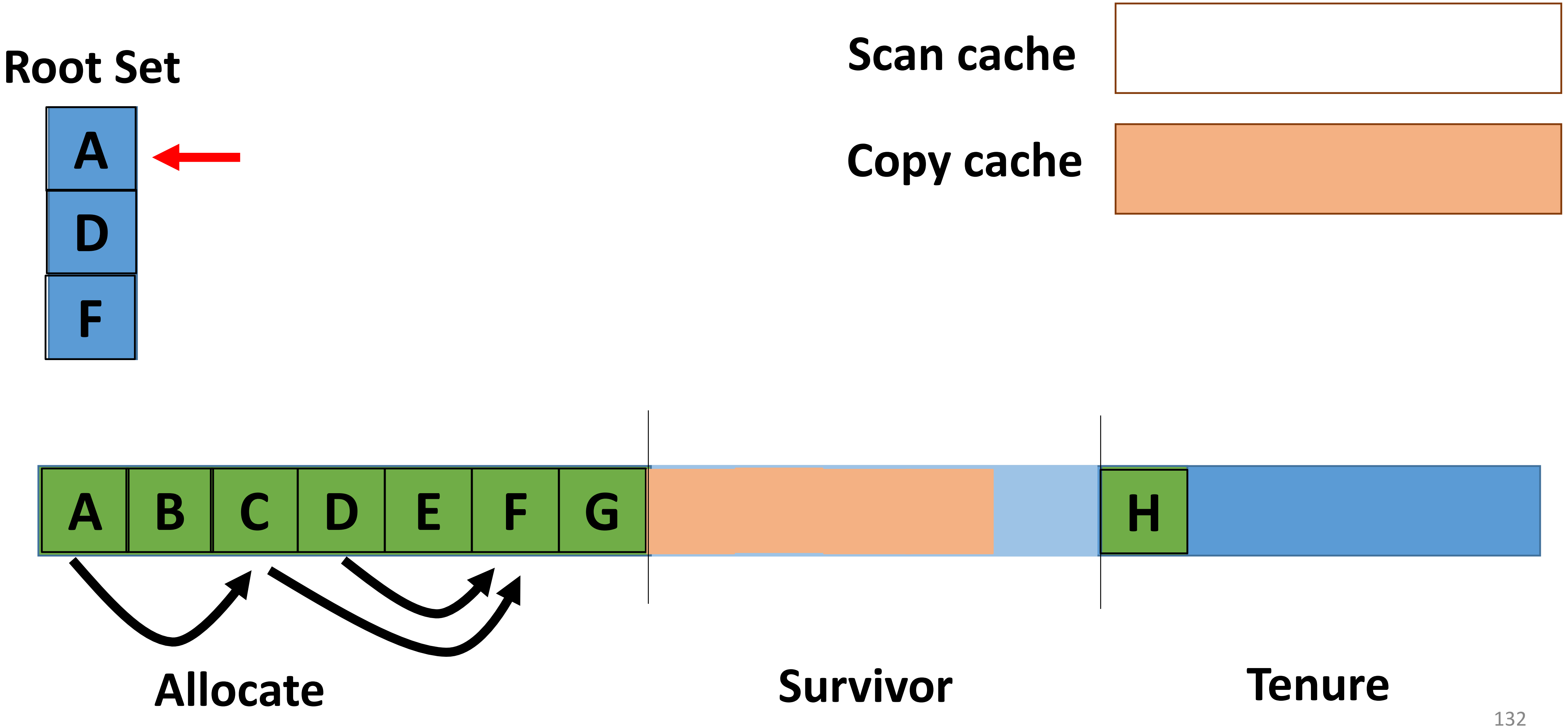
-Xgcpolicy:gencon GC



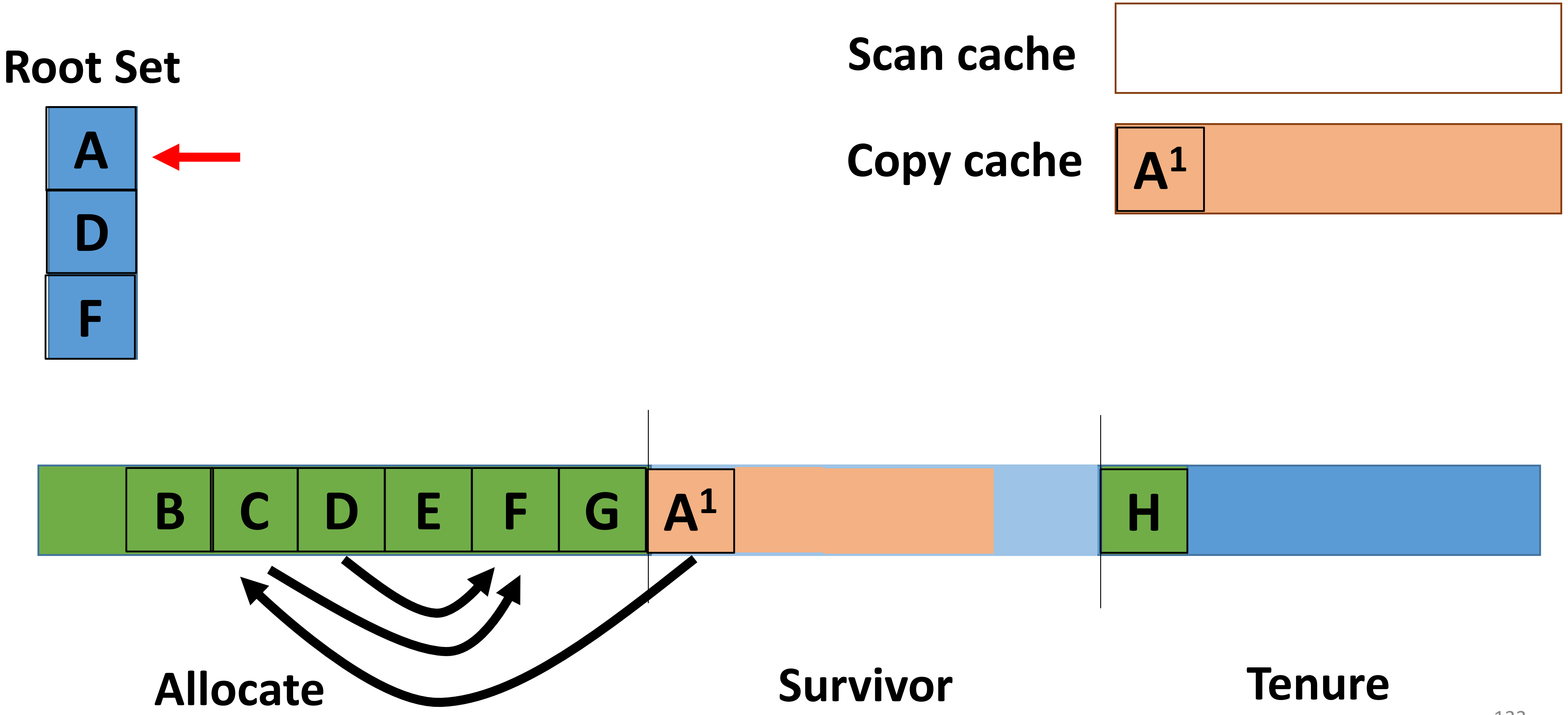
-Xgcpolicy:gencon GC



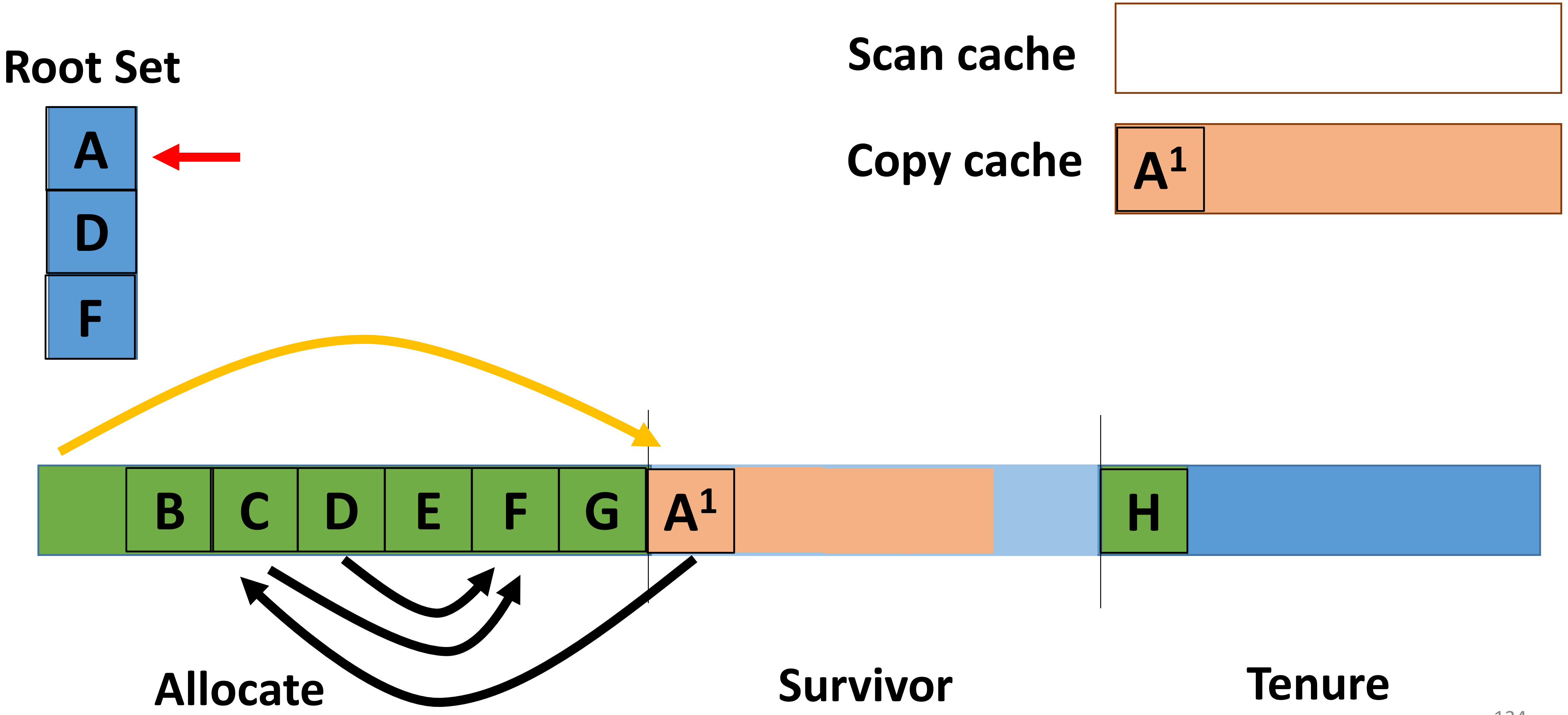
-Xgcpolicy:gencon GC



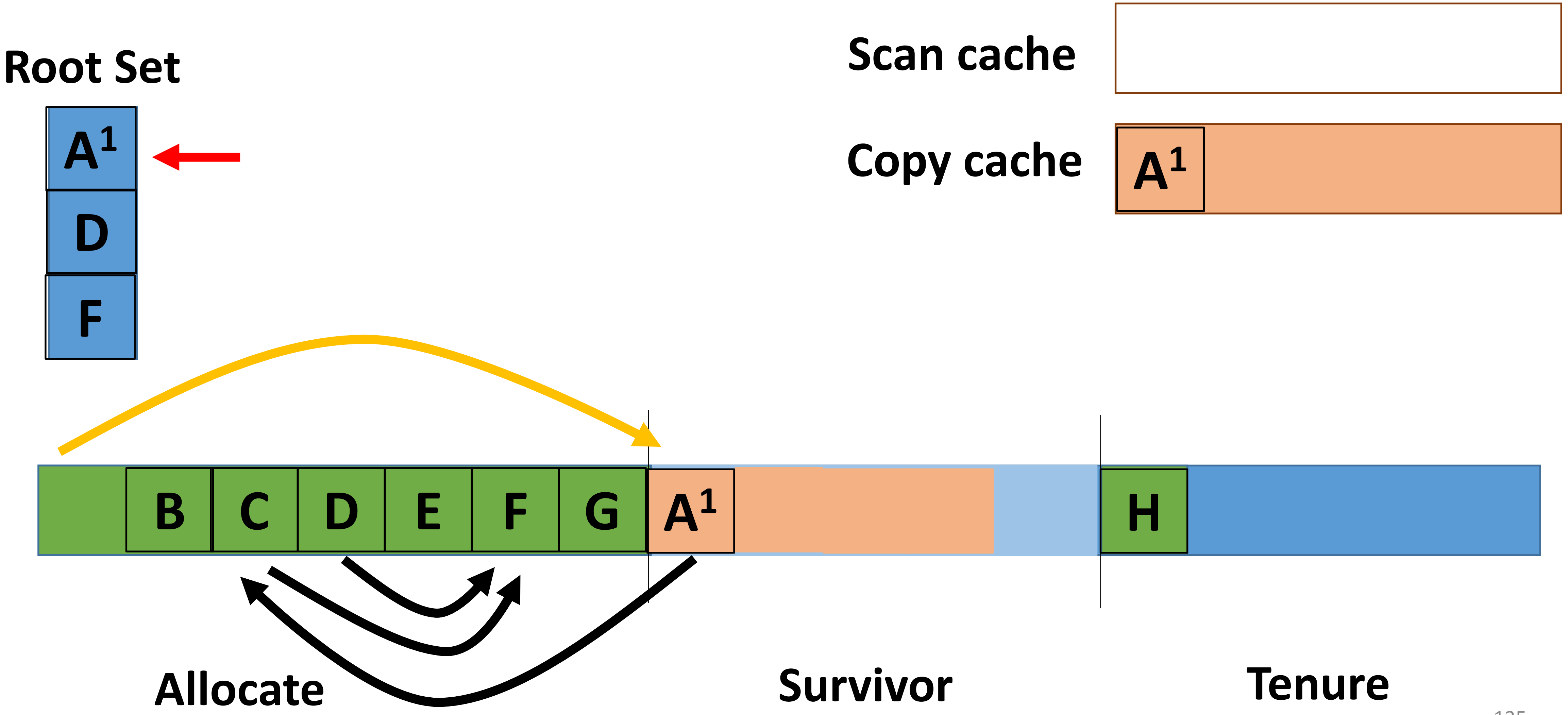
-Xgcpolicy:gencon GC



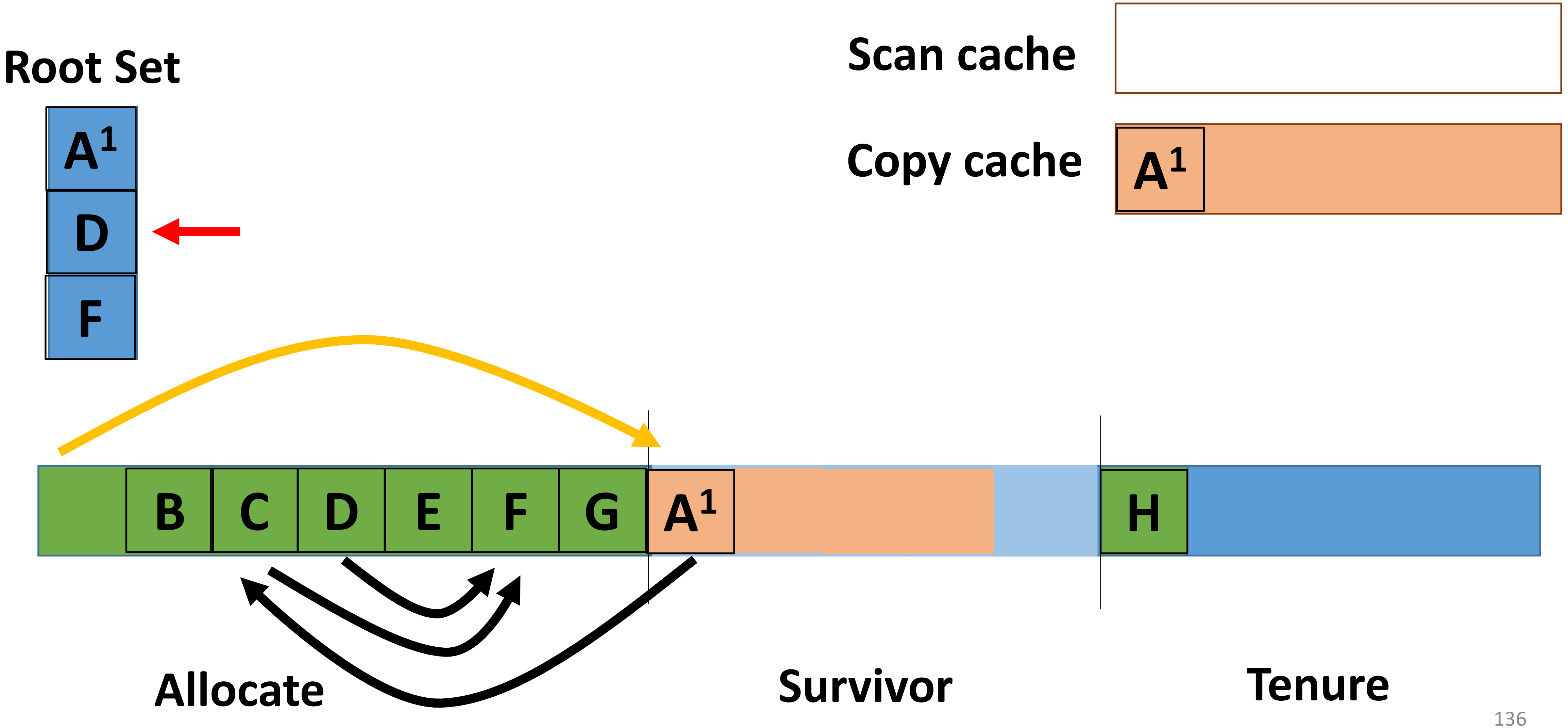
-Xgcpolicy:gencon GC



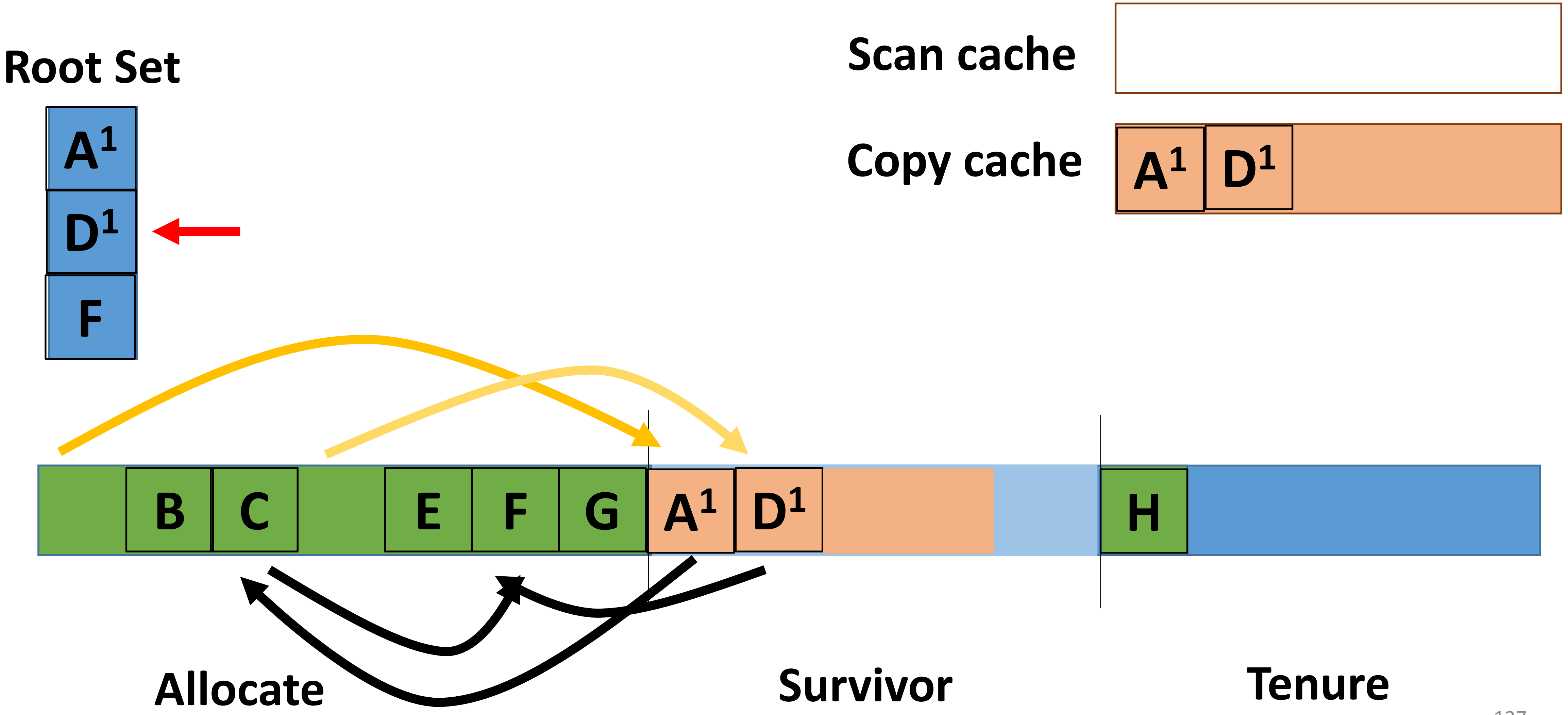
-Xgcpolicy:gencon GC



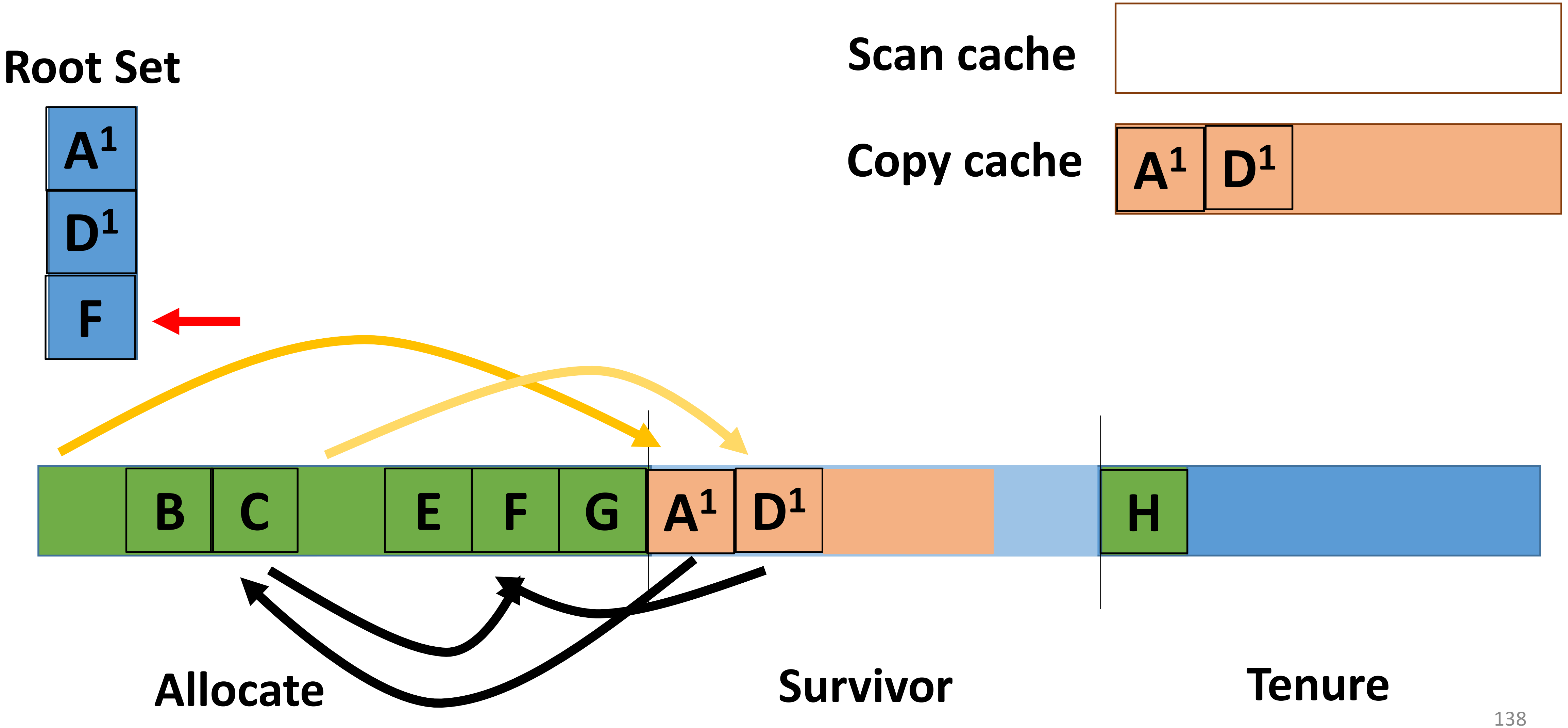
-Xgcpolicy:gencon GC



-Xgcpolicy:gencon GC

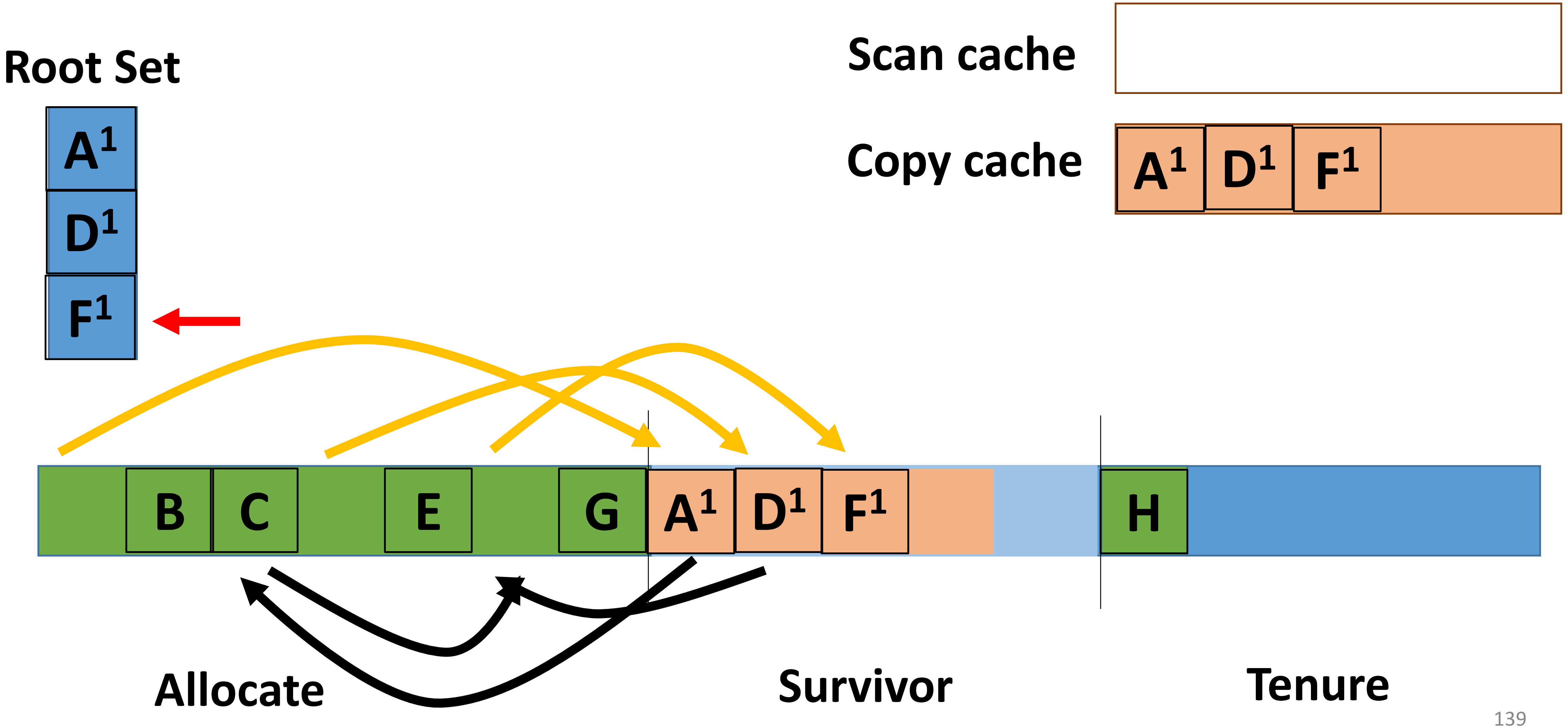


-Xgcpolicy:gencon GC



OpenJ9

-Xgcpolicy:gencon GC

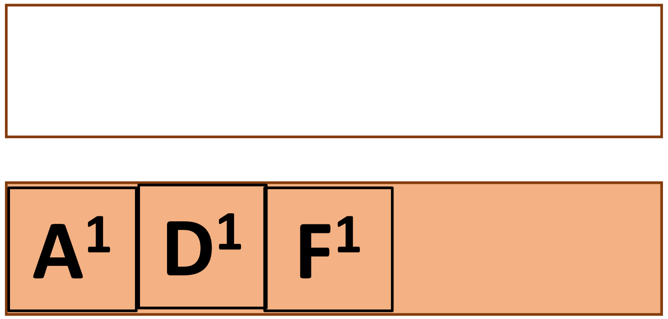


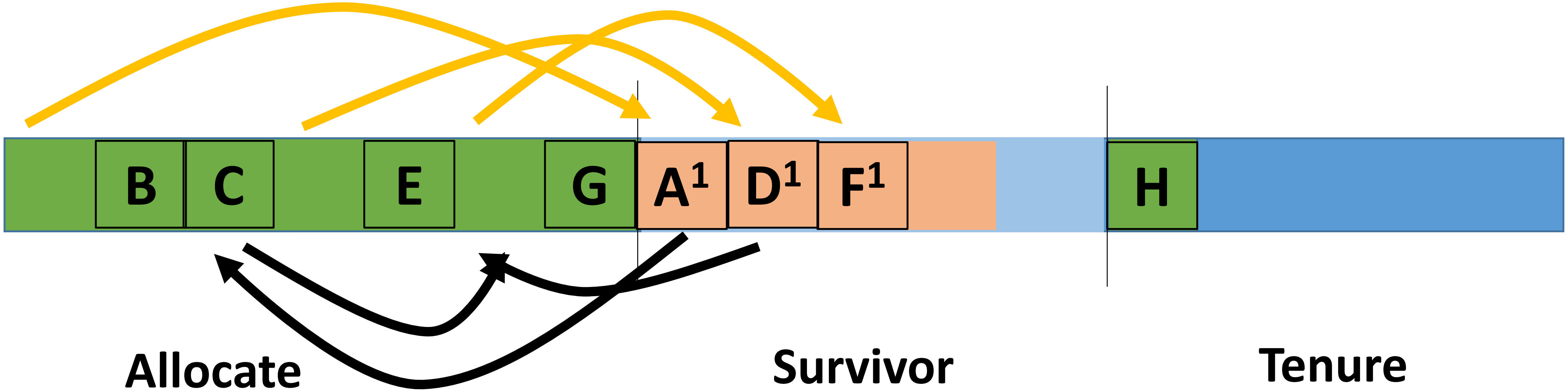
-Xgcpolicy:gencon GC

Work list

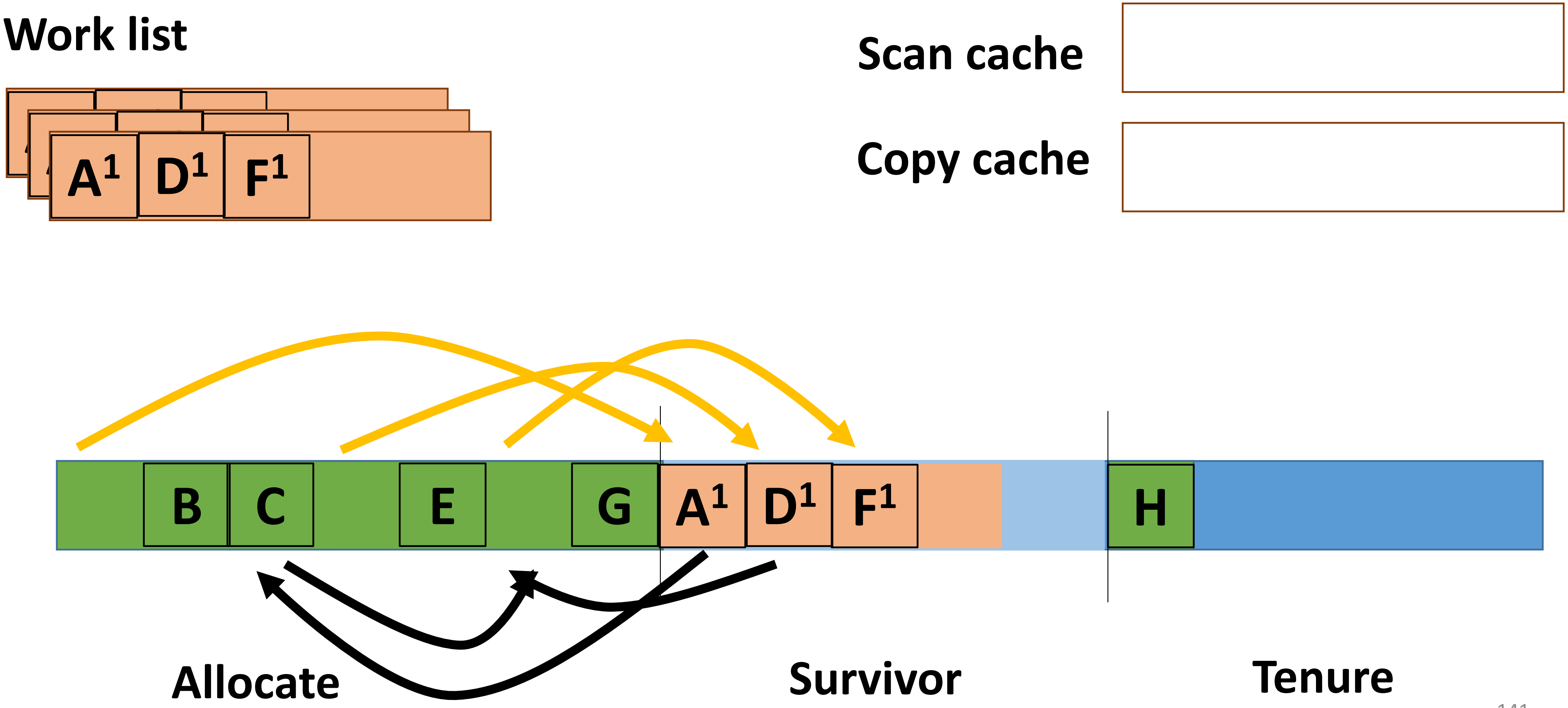
Scan cache

Copy cache

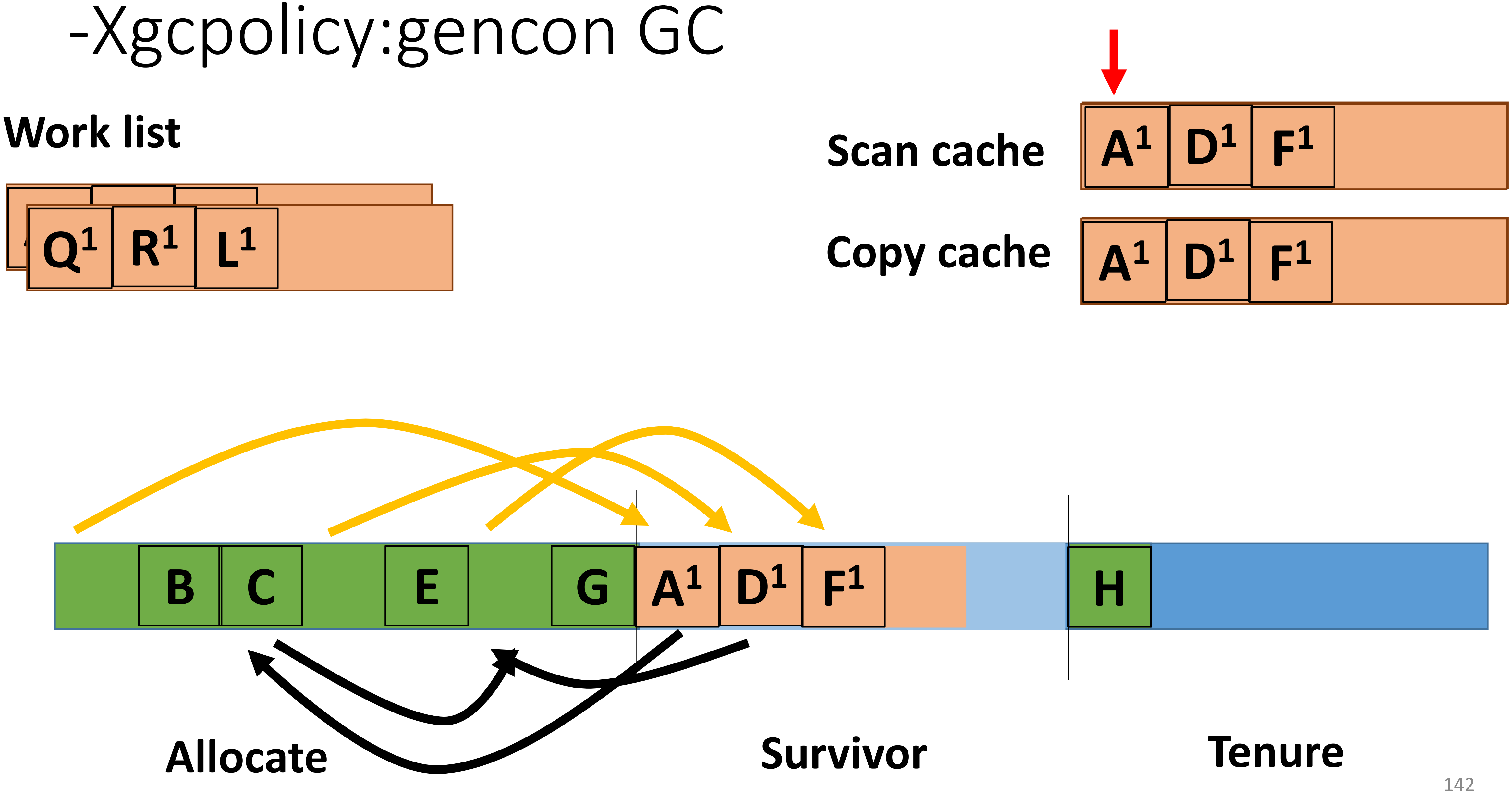




-Xgcpolicy:gencon GC



-Xgcpolicy:gencon GC



Work list

Q^1 R^1 L^1

Scan cache

A^1 D^1 F^1

Copy cache

A^1 D^1 F^1

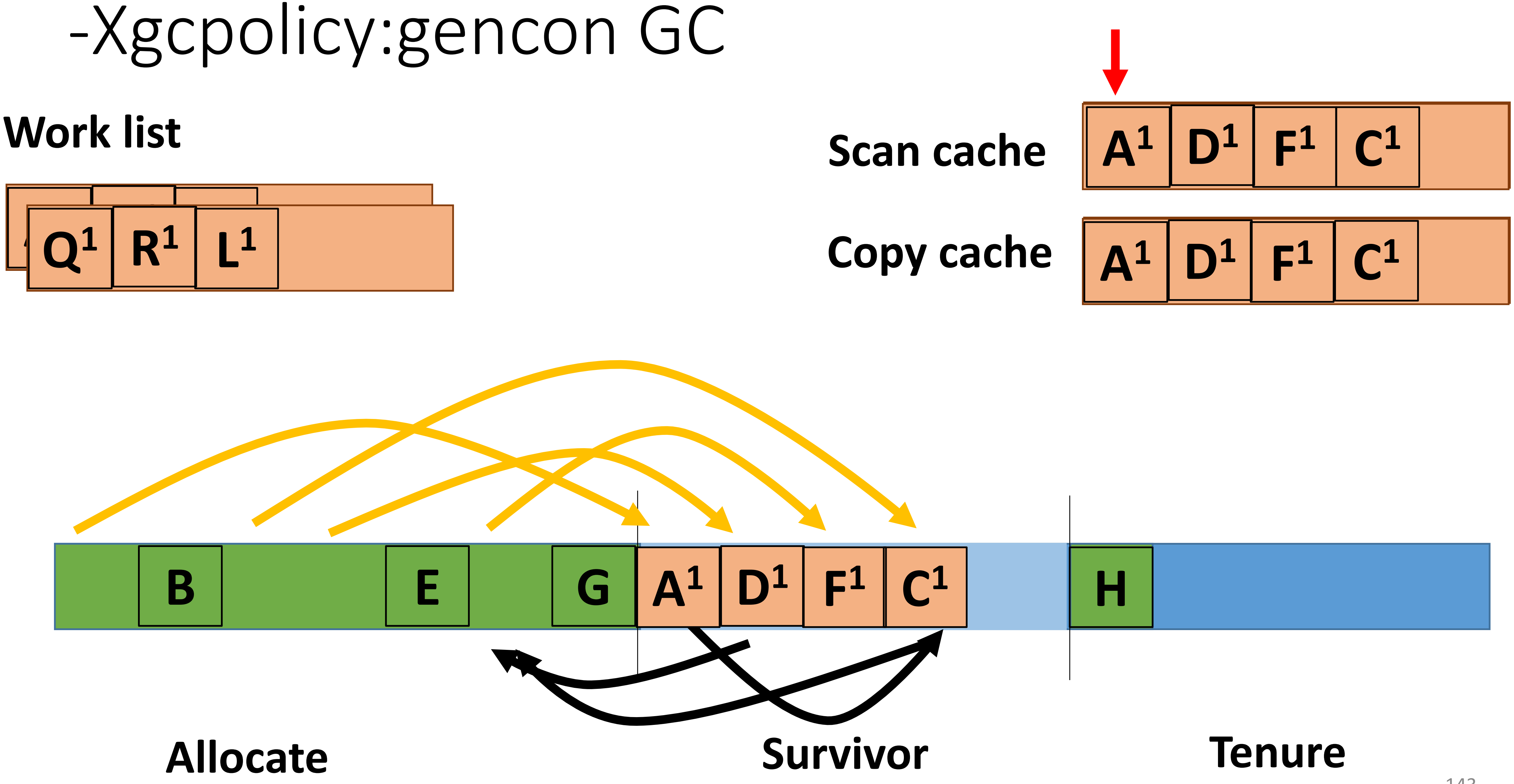
B C E G A^1 D^1 F^1 H

Allocate

Survivor

Tenure

-Xgcpolicy:gencon GC



Work list

Scan cache

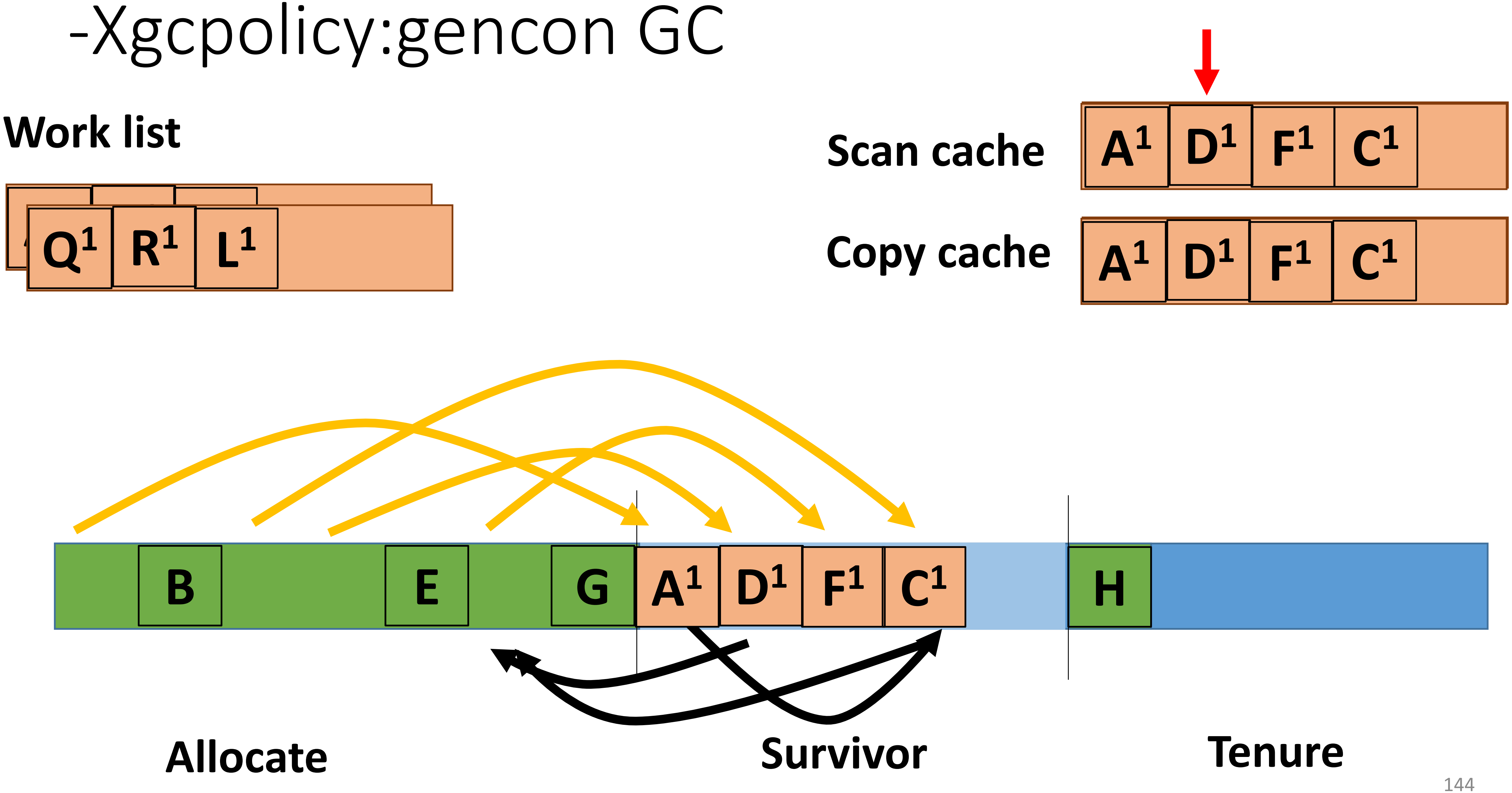
Copy cache

Allocate

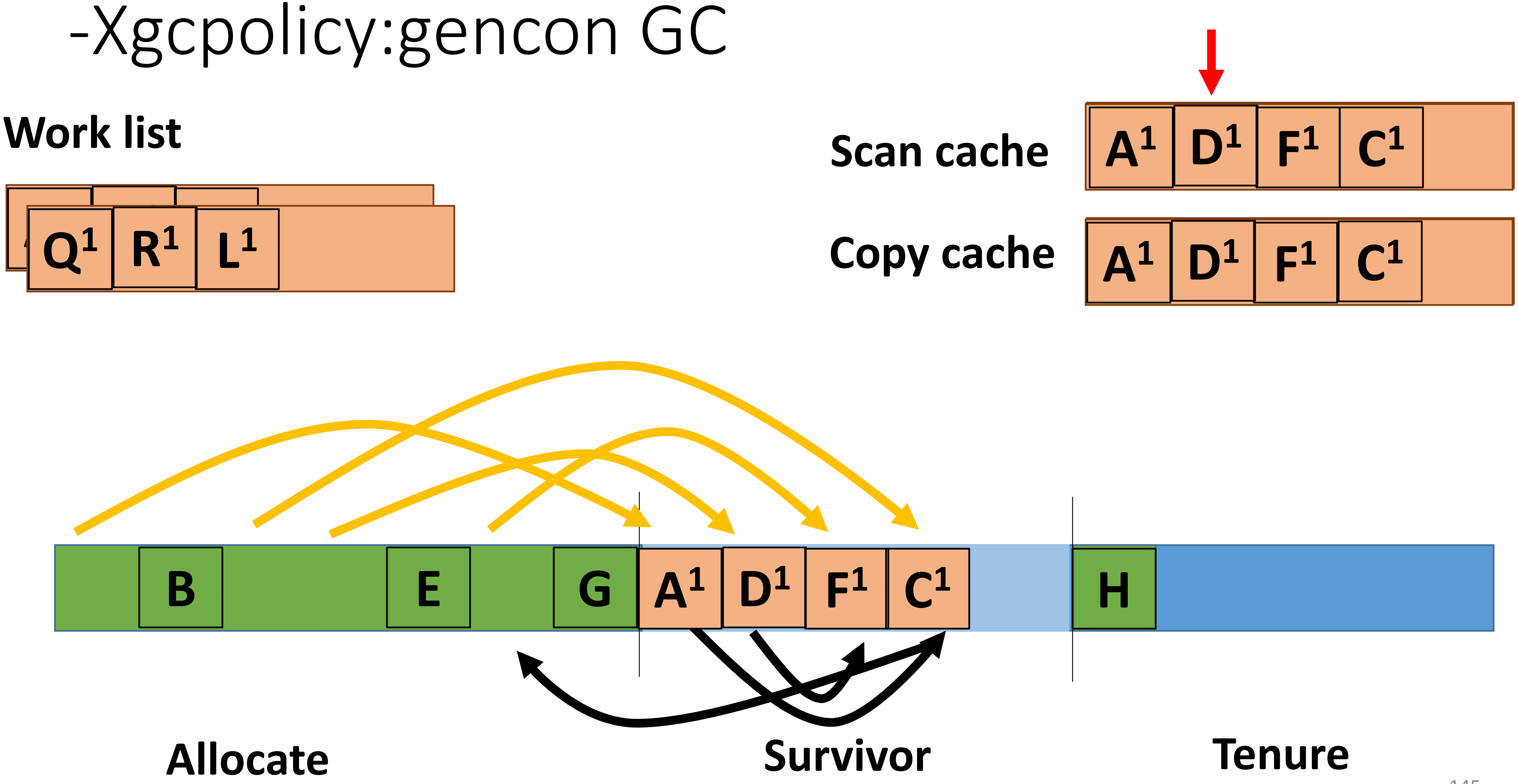
Survivor

Tenure

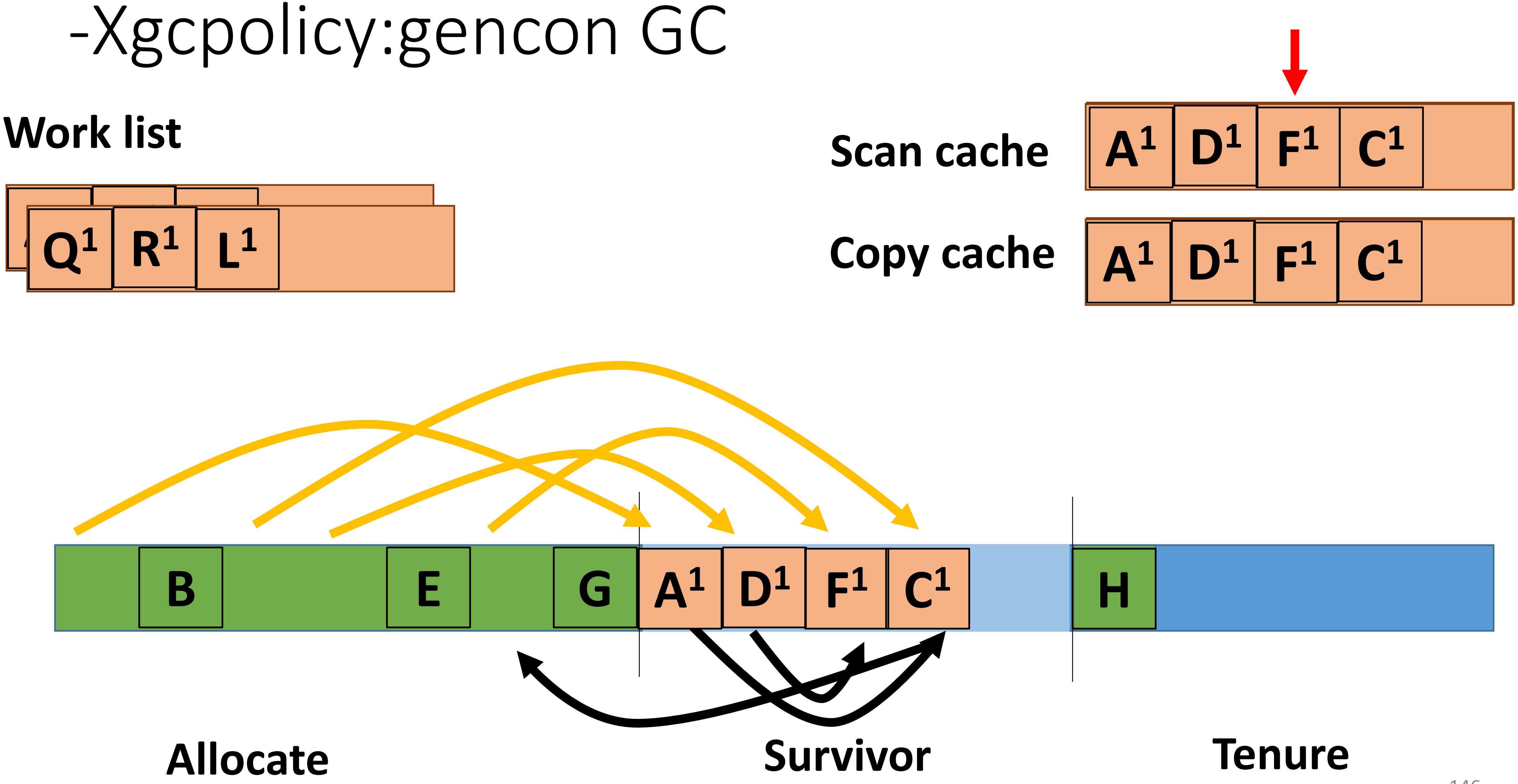
-Xgcpolicy:gencon GC



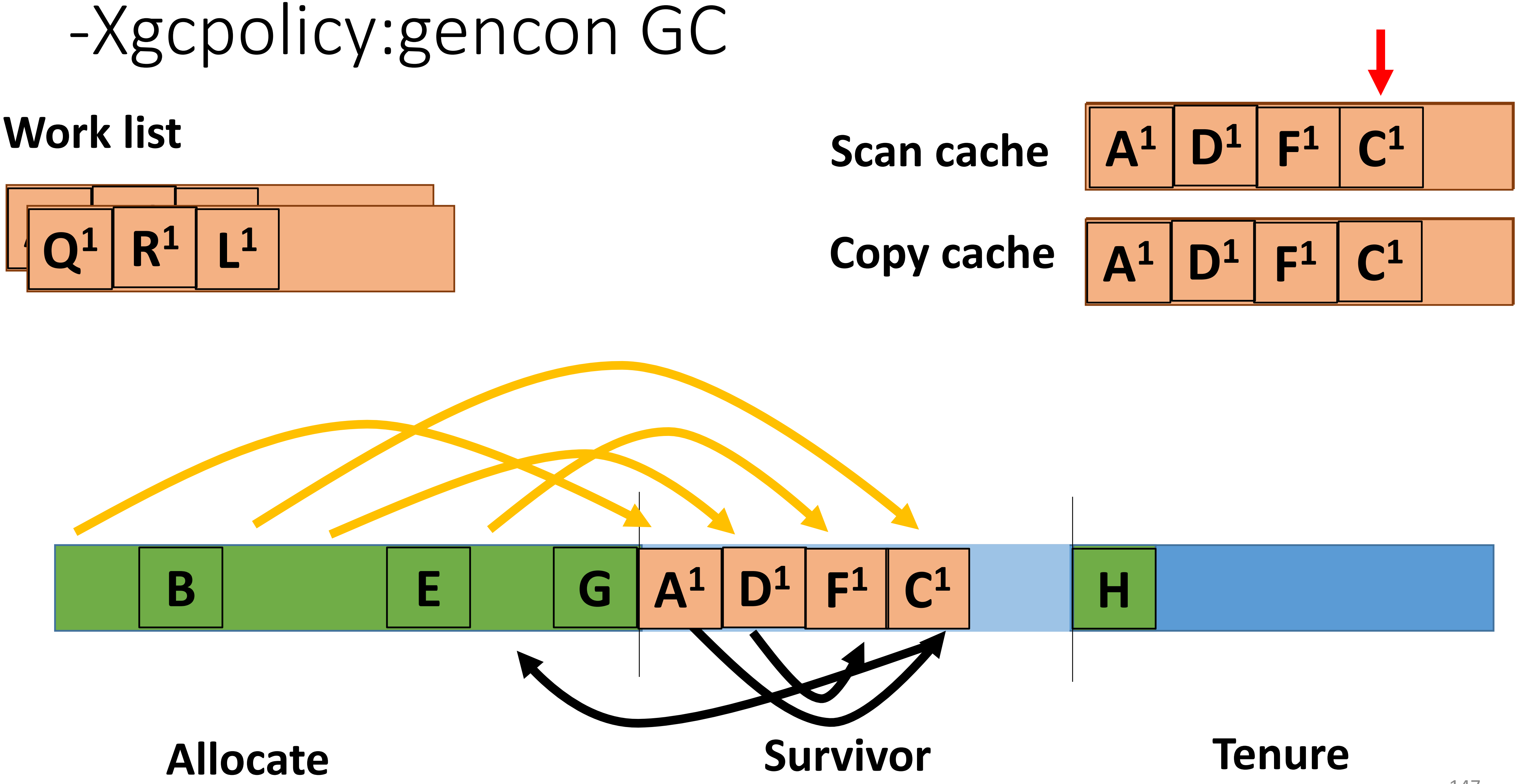
-Xgcpolicy:gencon GC



-Xgcpolicy:gencon GC



-Xgcpolicy:gencon GC



Work list

Q^1 R^1 L^1

Scan cache

A^1 D^1 F^1 C^1

Copy cache

A^1 D^1 F^1 C^1

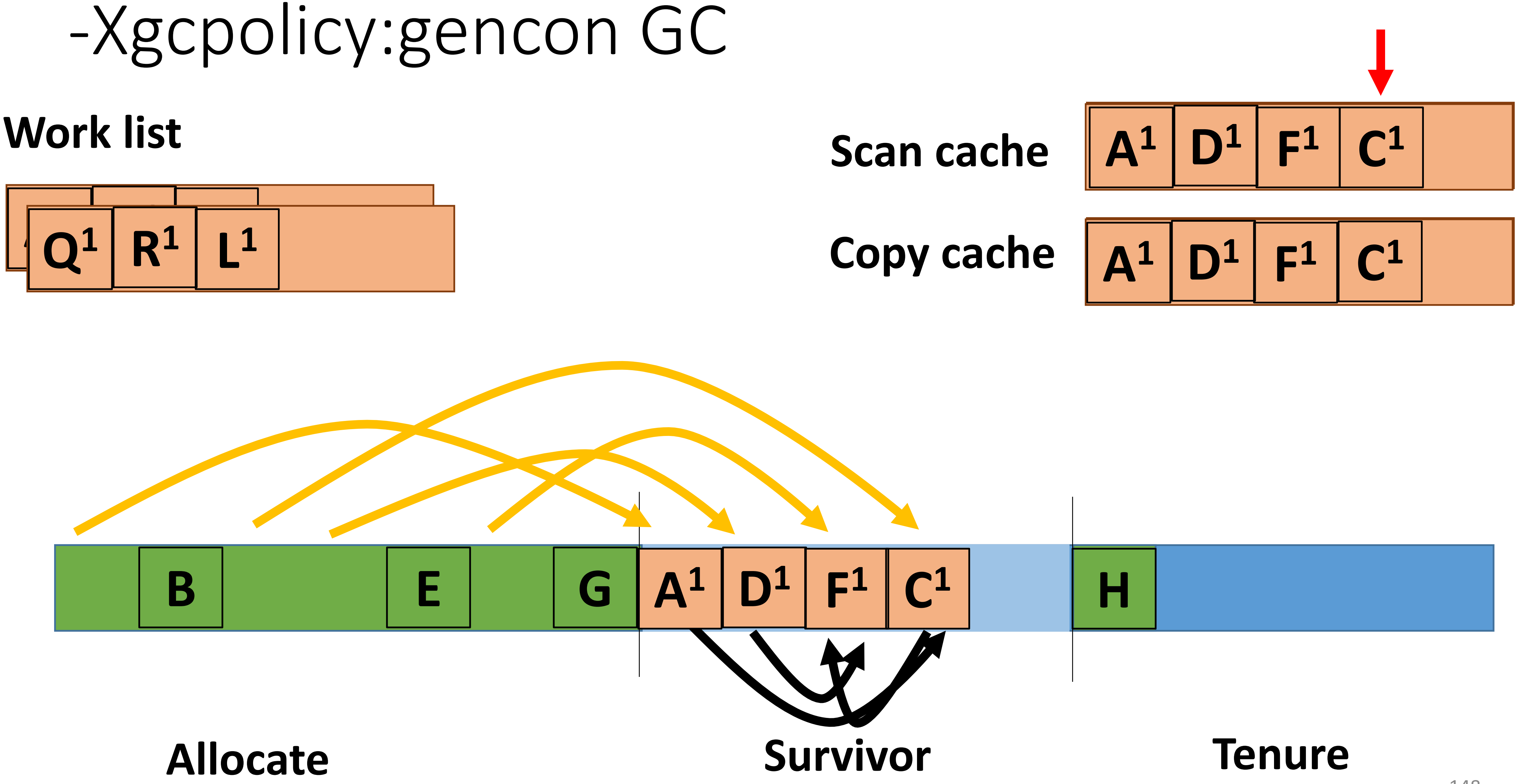
B E G A^1 D^1 F^1 C^1 H

Allocate

Survivor

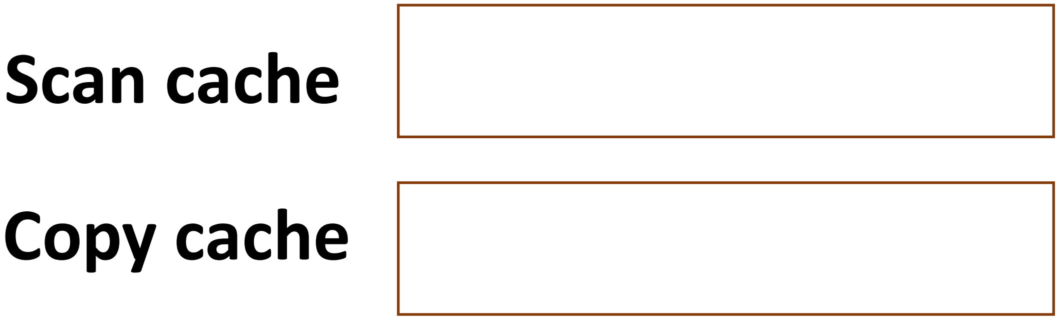
Tenure

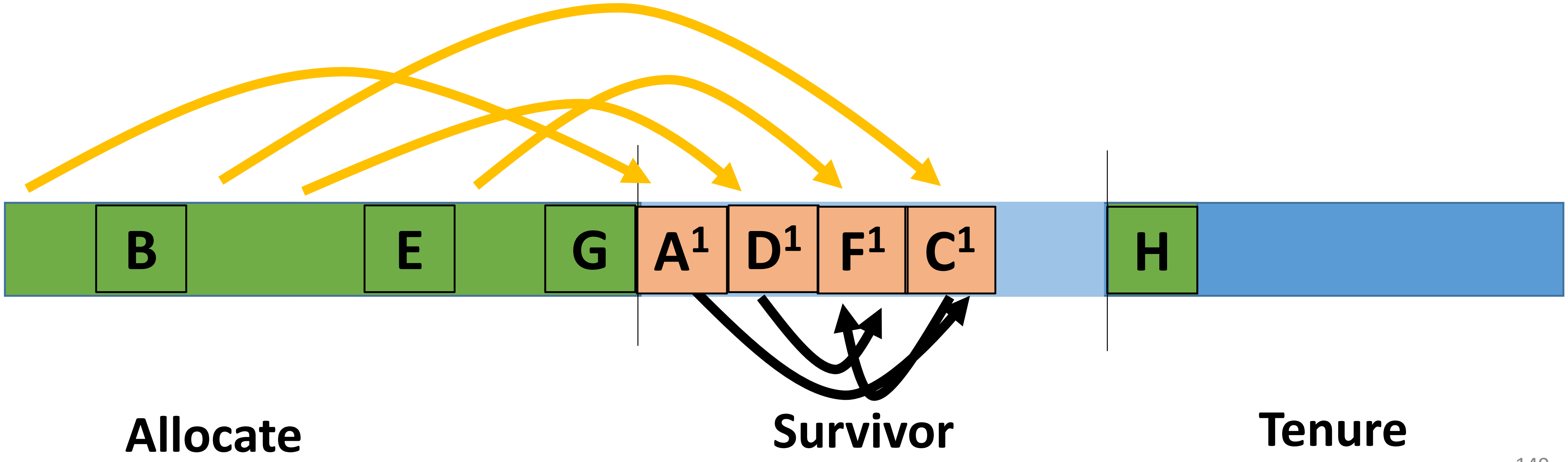
-Xgcpolicy:gencon GC



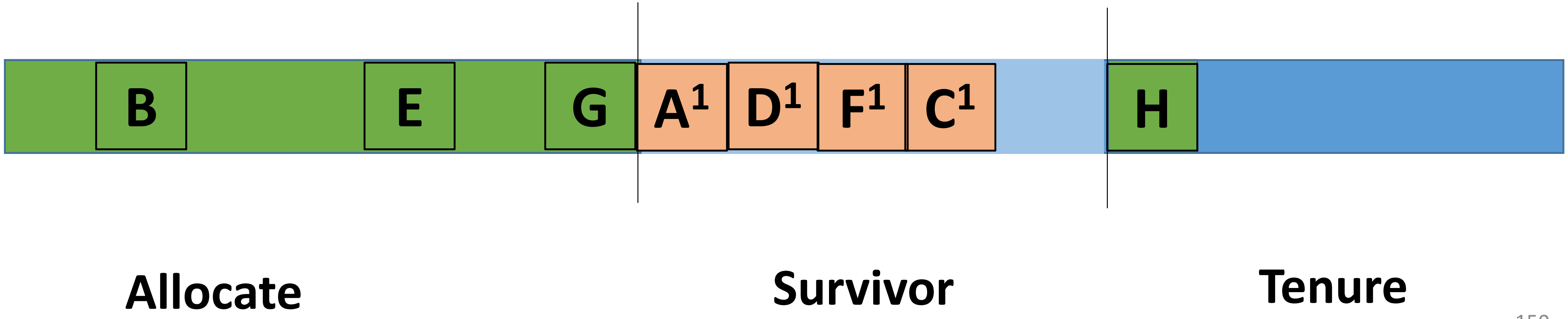
-Xgcpolicy:gencon GC

Work list

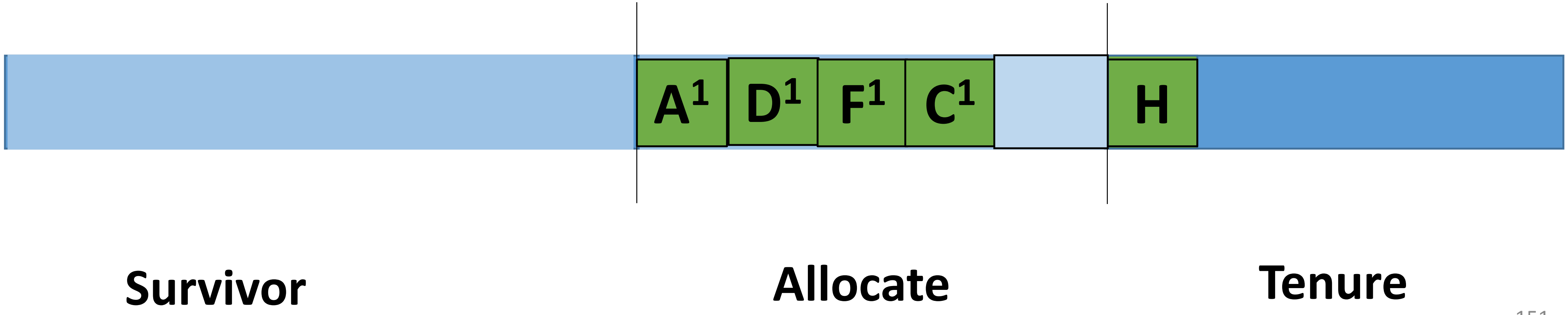




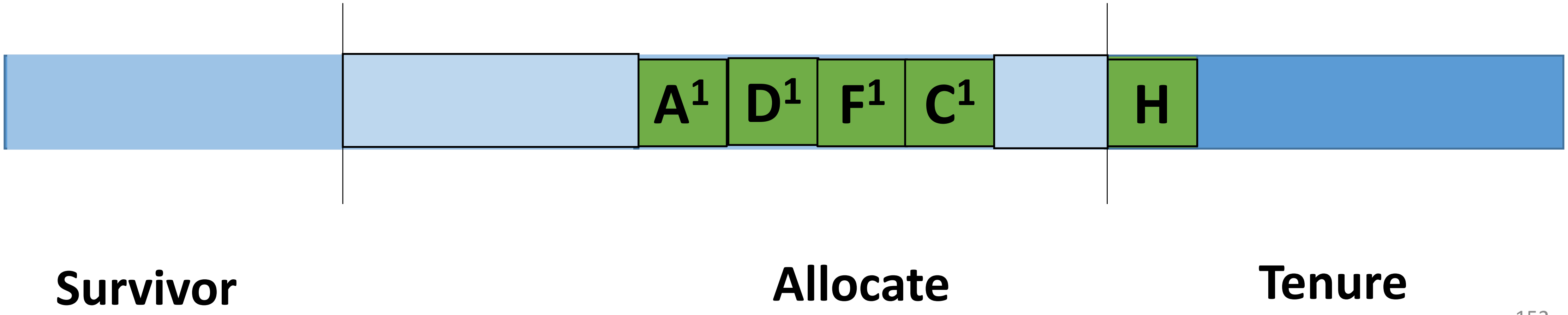
-Xgcpolicy:gencon GC



-Xgcpolicy:gencon GC



-Xgcpolicy:gencon GC

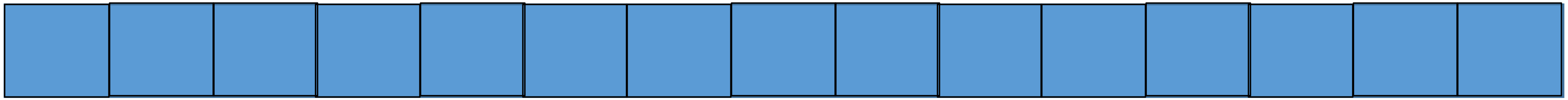


-Xgcpolicy:balanced

- Region based generational collector
  - Partial Garbage Collections (PGC) focus on high ROI regions
  - Goal of no global collections
  - NUMA aware allocation and GC
- Provides a significant reduction in max GC STW pause times
- Introduces a write barrier to track inter region references
- GC native memory overhead for 2 mark maps, work packets, remembered set card table and copy scan caches
- Full parallel global as a fall back if the PGCs can not keep up

-Xgcpolicy:balanced heap

- Heap is divided into a fixed number of regions
  - Region size is always a power of 2
  - Attempts to have between 1000-2000 regions
  - Bigger heap == bigger region size
- Regions are evenly distributed across NUMA nodes



Heap

-Xgcpolicy:balanced heap

- Allocate from Eden regions
  - Eden can be any set of completely free regions
  - Attempts to pick regions from each NUMA node
- Application threads allocate from the NUMA node they are running on

Heap

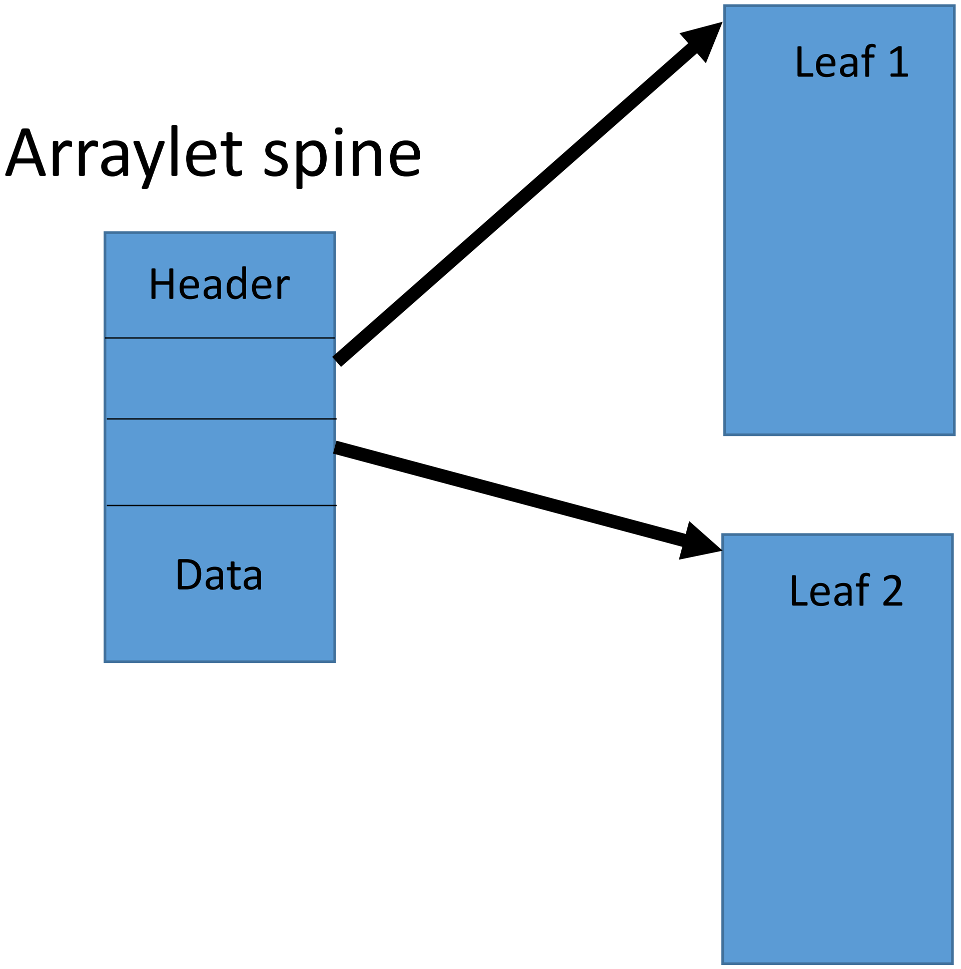
-Xgcpolicy:balanced heap

- No non-array object can be larger than region size
  - Object size > region size == OutOfMemoryError
- Large arrays are allocated as arraylets
  - Arrays less than region size are allocated as normal arrays

-Xgcpolicy:balanced arraylets

- Arraylets are a dis-contiguous representation of arrays
  - Array is create from construct and an arraylet spine and 1 or more arraylet leaves
  - An arraylet spine is allocate like a normal object
  - Each leaf consumes an entire region

-Xgcpolicy:balanced arraylets



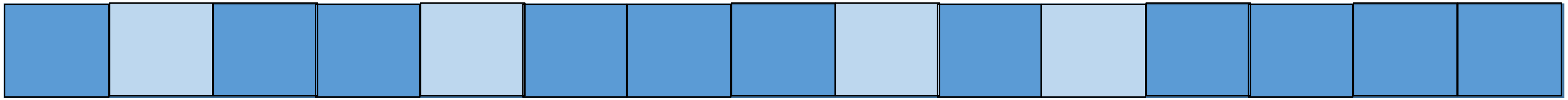
- To access an element you calculate the leaf (index / leaf size)
- Then you calculate the index into that leaf (index % leaf size)

-Xgcpolicy:balanced arraylets

- Array element access is slower due to an extra dereference
- JNI critical sections causes the entire data section of the arraylet to be copied into a native memory buffer

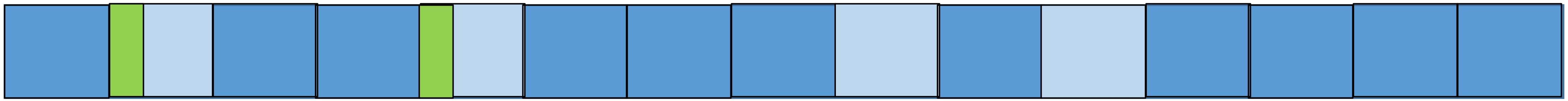
-Xgcpolicy:balanced heap

- What does allocation look like?



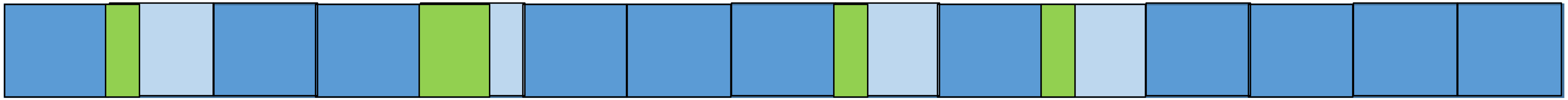
Heap

-Xgcpolicy:balanced heap



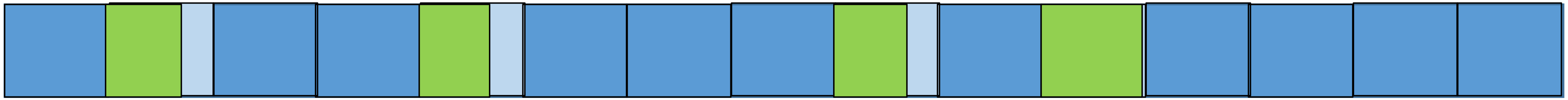
Heap

-Xgcpolicy:balanced heap



Heap

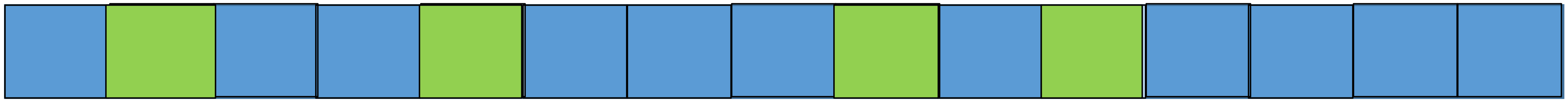
-Xgcpolicy:balanced heap



Heap

-Xgcpolicy:balanced heap

- Time for a PGC

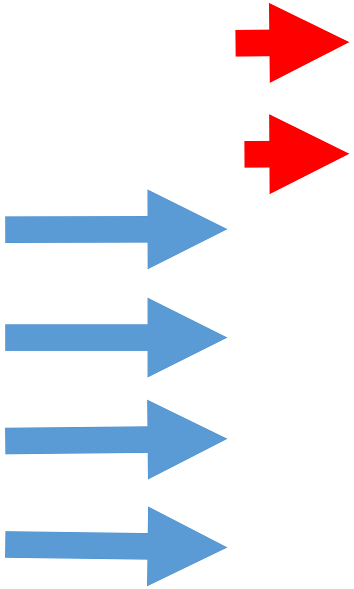


Heap

-Xgcpolicy:balanced GC

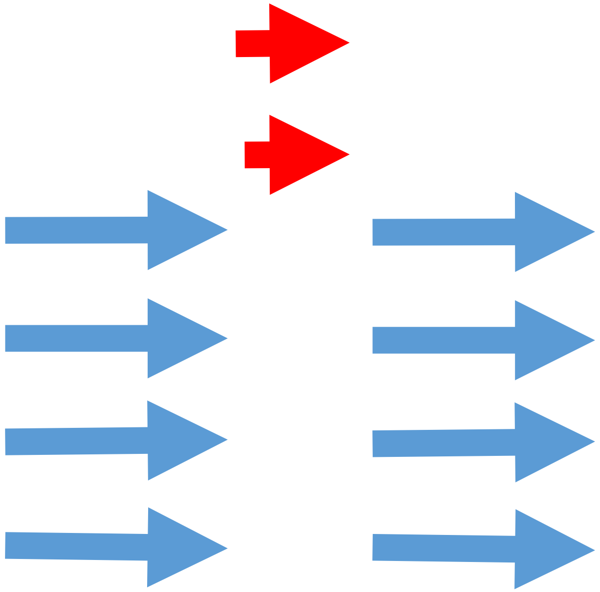
-Xgcpolicy:balanced GC

PGC 1

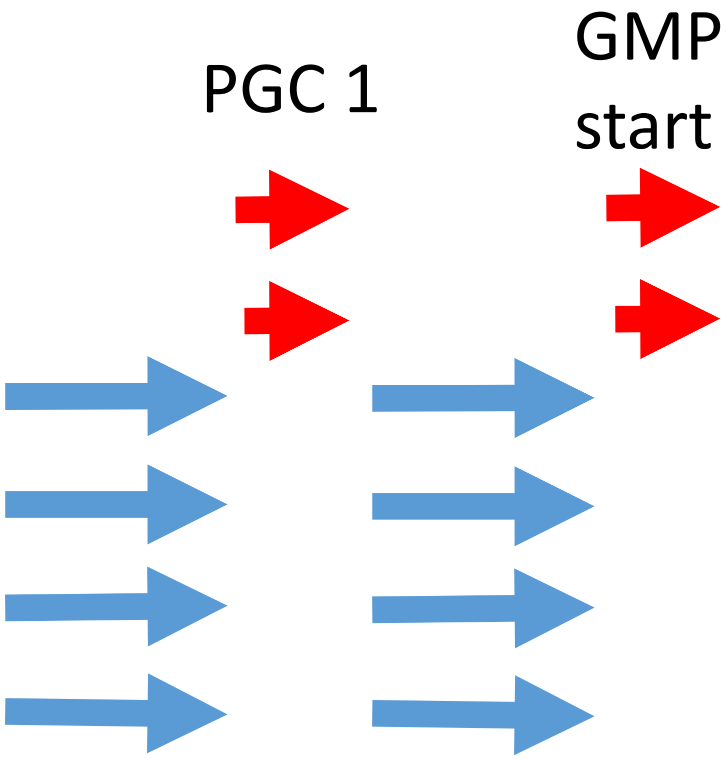


-Xgcpolicy:balanced GC

PGC 1

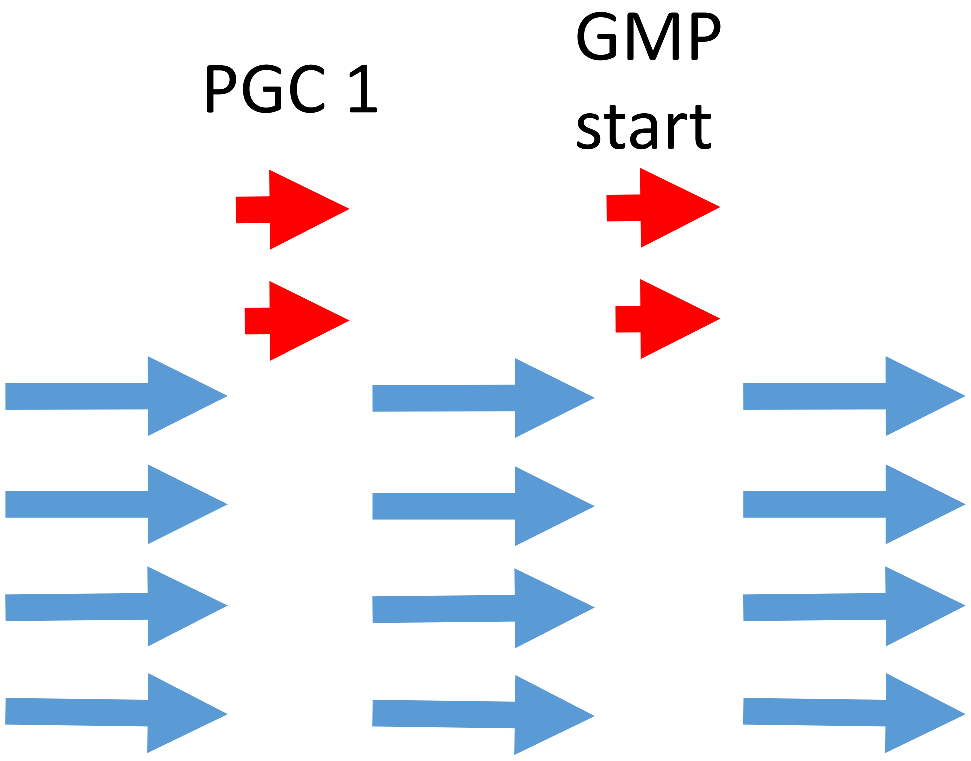


-Xgcpolicy:balanced GC

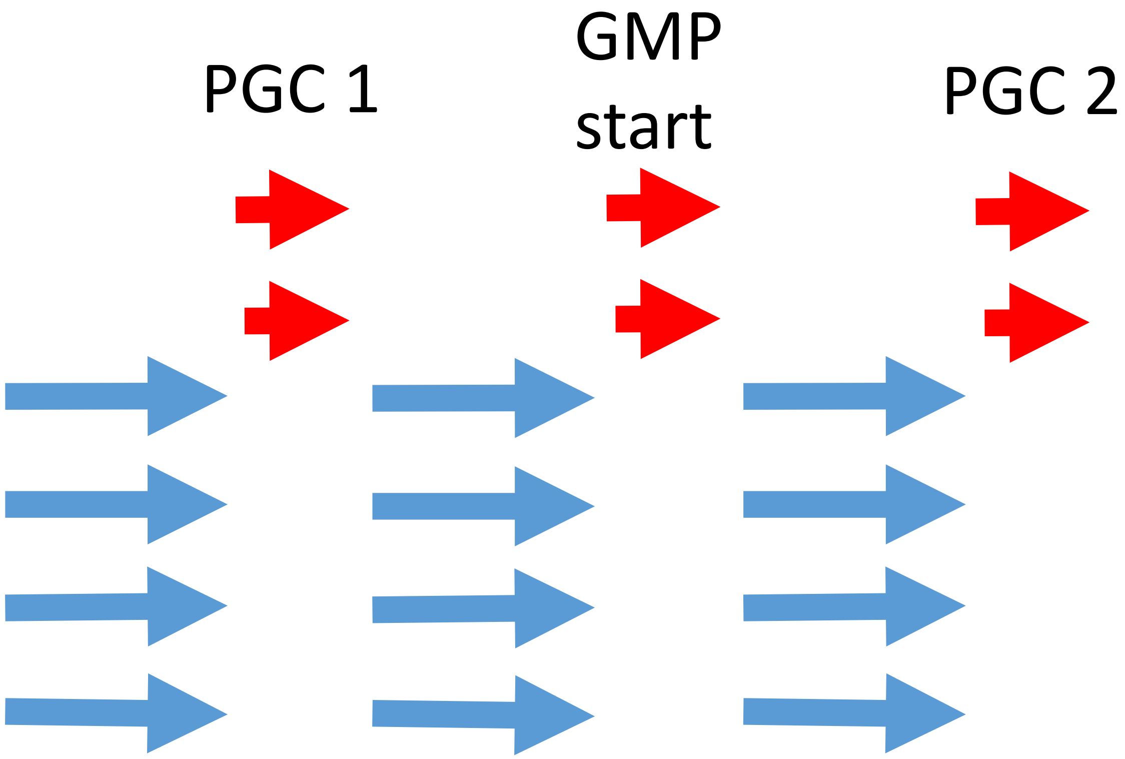


OpenJ9

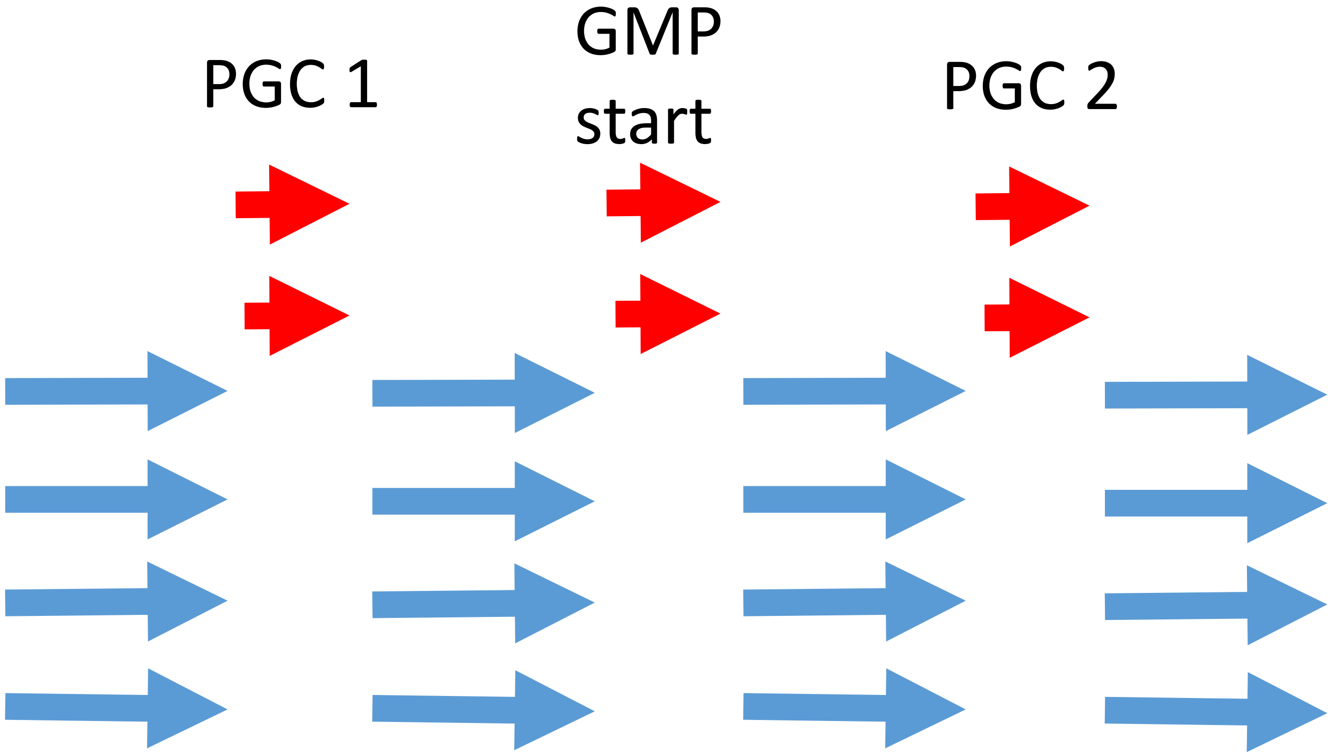
-Xgcpolicy:balanced GC



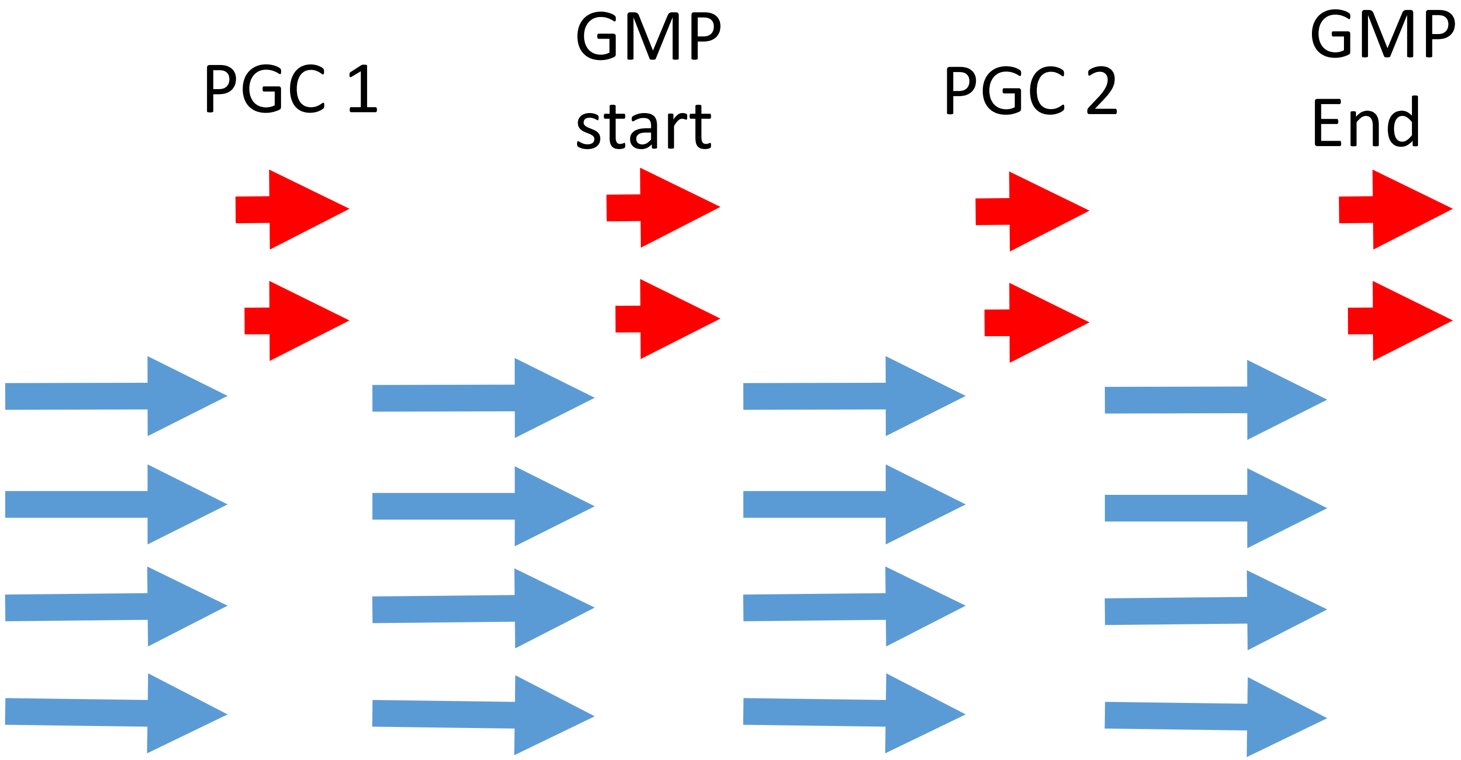
-Xgcpolicy:balanced GC



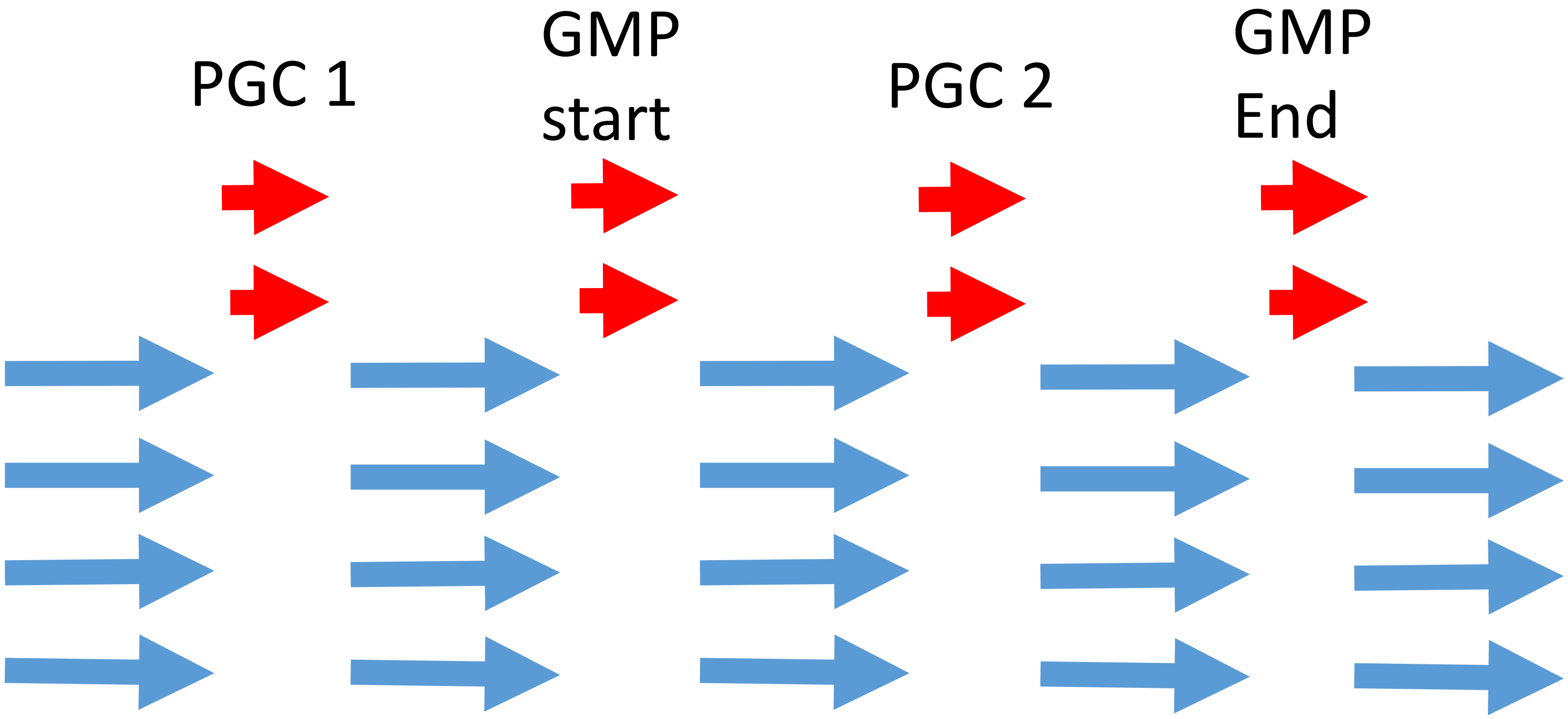
-Xgcpolicy:balanced GC



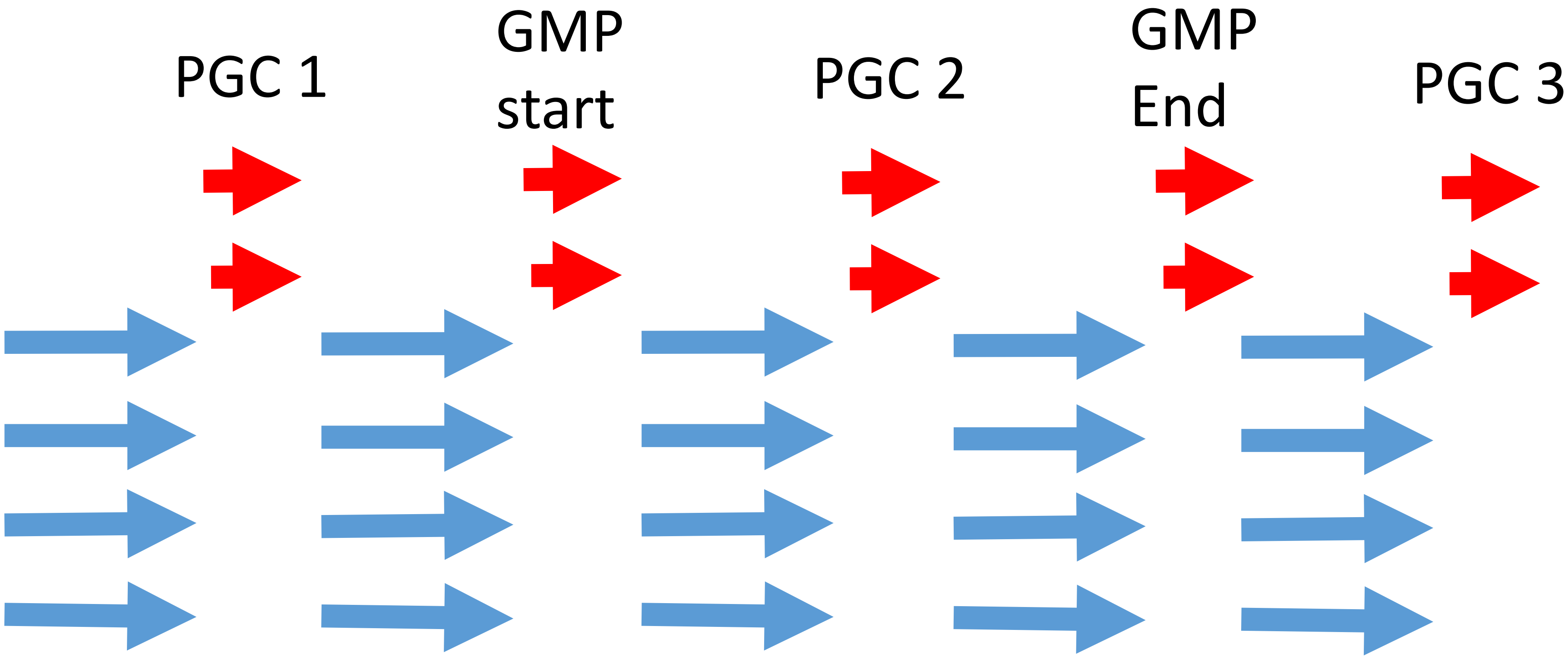
-Xgcpolicy:balanced GC



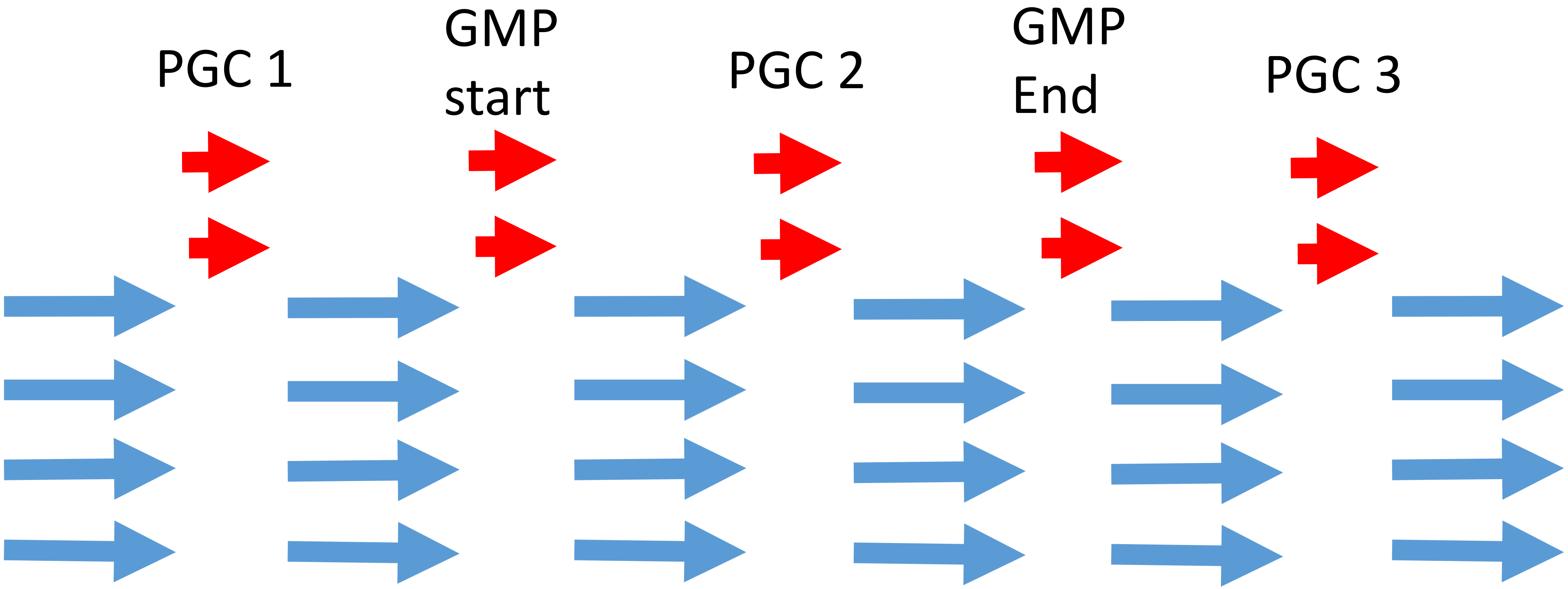
-Xgcpolicy:balanced GC



-Xgcpolicy:balanced GC



-Xgcpolicy:balanced GC

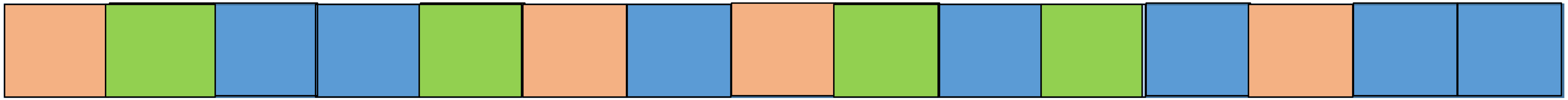


-Xgcpolicy:balanced Global Mark Phase (GMP)

- Does not reclaim any memory!
- Performs a marking phase only
- Scheduled to run in between PGCs
- Builds an accurate mark map of the whole heap
- Mark map is used to predict region ROI for PGC
- Scrubs RSCL

-Xgcpolicy:balanced PGC

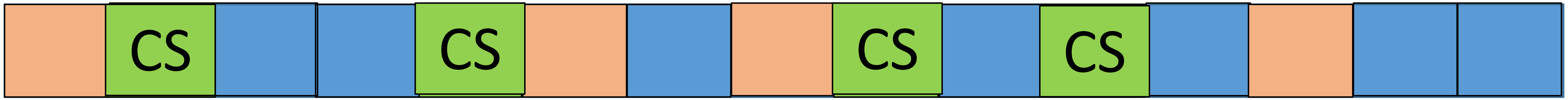
- Select regions for inclusion
  - Always select Eden regions
  - Use GMP mark map to predict regions with low live counts
  - If RSCL has too many entries it is not a good candidate
- RSCL for selected regions becomes a root



Heap

-Xgcpolicy:balanced PGC

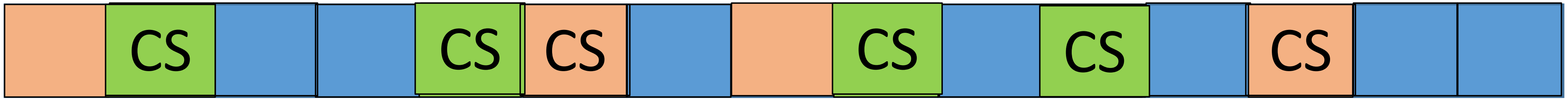
- Select regions for inclusion
  - Always select Eden regions
  - Use GMP mark map to predict regions with low live counts
  - If RSCL has too many entries it is not a good candidate
- RSCL for selected regions becomes a root



Heap

-Xgcpolicy:balanced PGC

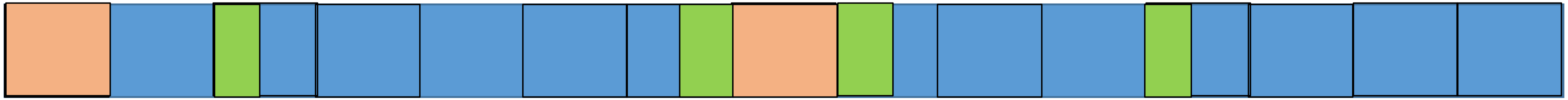
- Select regions for inclusion
  - Always select Eden regions
  - Use GMP mark map to predict regions with low live counts
  - If RSCL has too many entries it is not a good candidate
- RSCL for selected regions becomes a root



Heap

-Xgcpolicy:balanced PGC

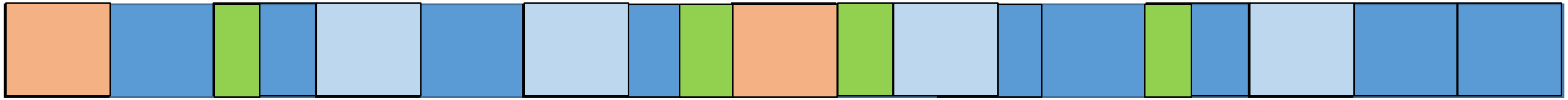
- Perform the copy forward operation



Heap

-Xgcpolicy:balanced PGC

- Perform the copy forward operation
- Pick the next free regions for Eden



Heap

-Xgcpolicy:balanced inter region write barrier

- Why is there a write barrier required?

-Xgcpolicy:balanced inter region write barrier

- Why is there a write barrier required?

- Balanced PGCs can select any region to be included in the collect.
- Similar to the generational barrier the GC needs to know which regions reference a given region

-Xgcpolicy:balanced inter region write barrier

- How is the write barrier implemented?

```
private void setField(Object A, Object C) {
   A.field1 = C;
}
```

-Xgcpolicy:balanced inter region write barrier

- How is the write barrier implemented?

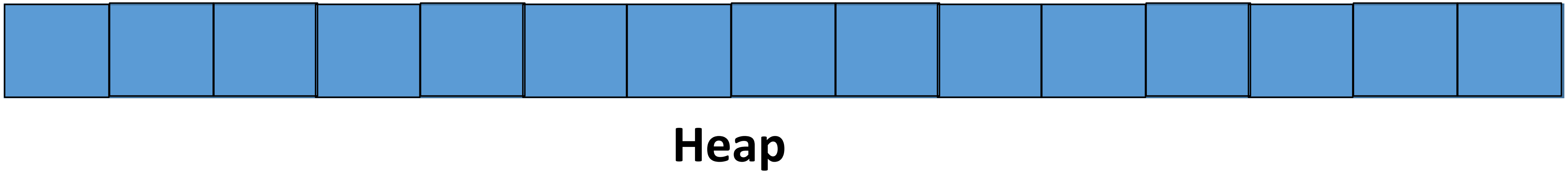
```
private void setField(Object A, Object C) {
    A.field1 = C;
    if (findRegion(A) != findRegion(C)) {
        addRSCLEntryFor(C, A);
    }
}
```

-Xgcpolicy:metronome

- Incremental soft realtime collector
  - Ultra low STW pause times of 3ms
  - Incremental mark and sweep phases with no compactor
  - Application receives 70% of the utilization
  - In any timing windows of 100ms application will run for 70ms
- Provides a significant reduction in max GC STW pause times
- Uses a Snapshot At The Beginning (SATB) write barrier
- High percentage of floating garbage
- GC native memory overhead for a mark map, work packets and a remembered set

-Xgcpolicy:metronome heap

- Heap is divided into fixed sized regions
  - Region size is 64k
  - Bigger heap == more regions



-Xgcpolicy:metronome heap

- Cell based allocation
- Regions are assigned a cell size on first use
  - Cell sizes range from 16 bytes to 2048 bytes
  - Only objects of that cell size can be allocated in a region of a particular cell size

Heap

-Xgcpolicy:metronome heap

- Objects larger than 2048 bytes consume contiguous regions
  - If no contiguous regions a full STW GC is completed before OOM
- Large arrays are allocated as arraylets
  - Arraylet leaves are 2048 bytes
  - Each arraylet leaf region contains 32 arraylet leaves

Heap

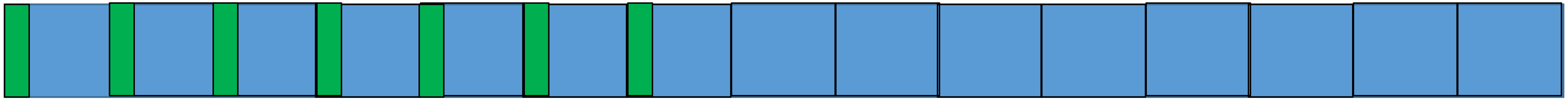
-Xgcpolicy:metronome heap

- What does allocation look like?

Heap

-Xgcpolicy:metronome heap

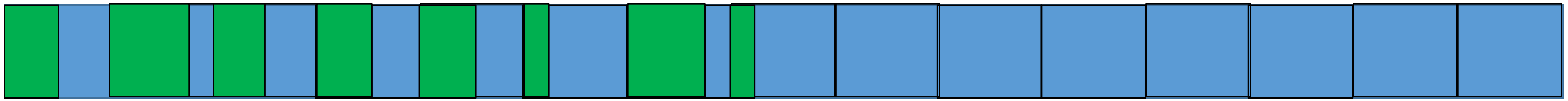
- What does allocation look like?



Heap

-Xgcpolicy:metronome heap

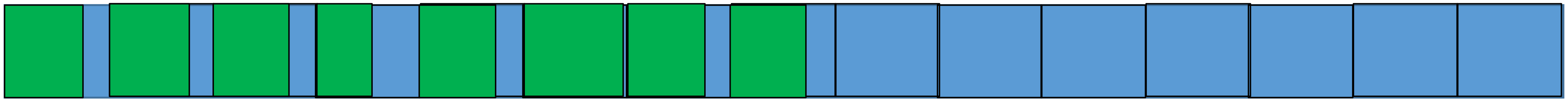
- What does allocation look like?



Heap

-Xgcpolicy:metronome heap

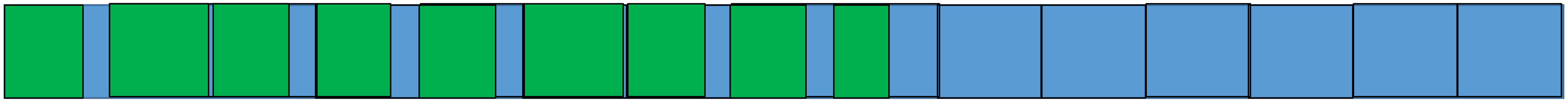
- What does allocation look like?



Heap

-Xgcpolicy:metronome heap

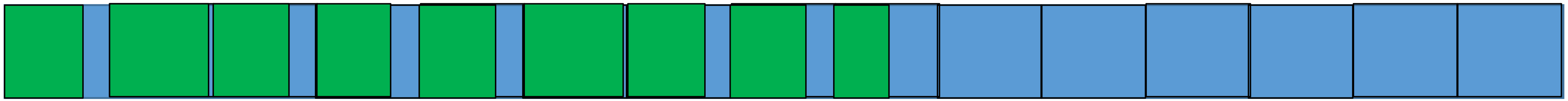
- Time to start a GC before the heap is exhausted
- GCs are triggered with 50% heap free



Heap

-Xgcpolicy:metronome heap

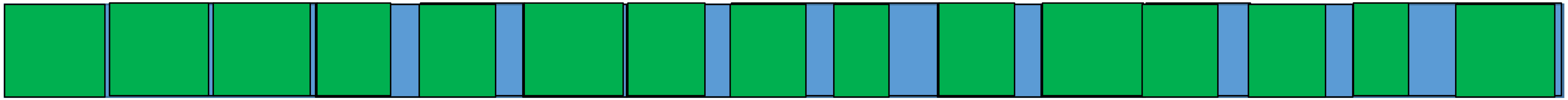
- While the GC is happening heap is still consumed
- All objects allocated during the GC are kept alive for this GC



Heap

-Xgcpolicy:metronome heap

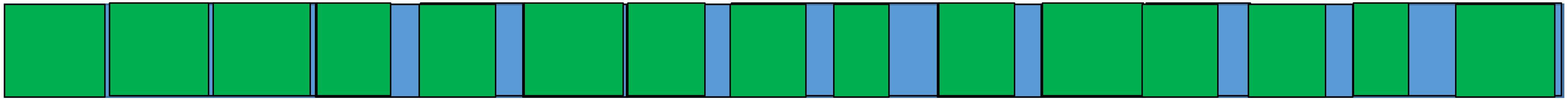
- At marking end the heap basically full



Heap

-Xgcpolicy:metronome heap

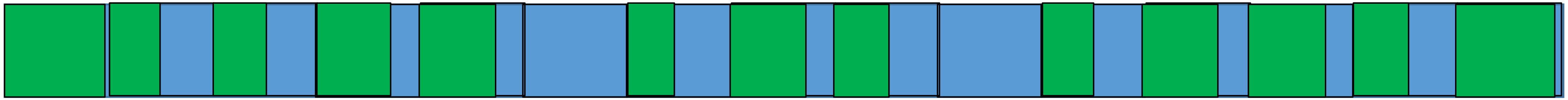
- Sweep phase will start freeing memory



Heap

-Xgcpolicy:metronome heap

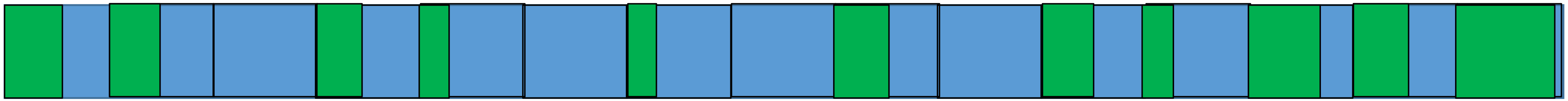
- Sweep phase will start freeing memory



Heap

-Xgcpolicy:metronome heap

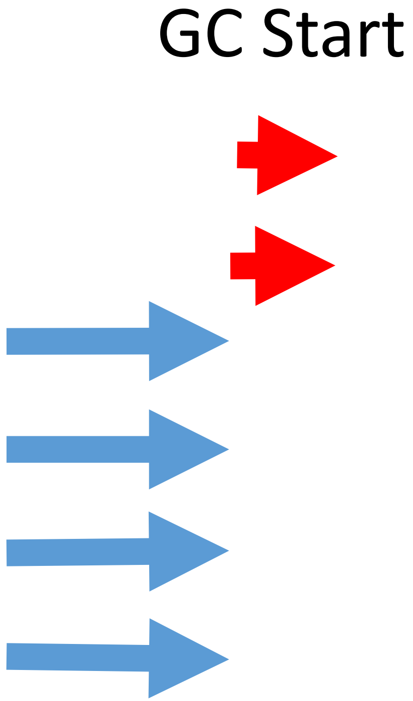
- At sweep complete the application goes back to getting 100% utilization
- If less than 50% of the heap is freed another GC is triggered



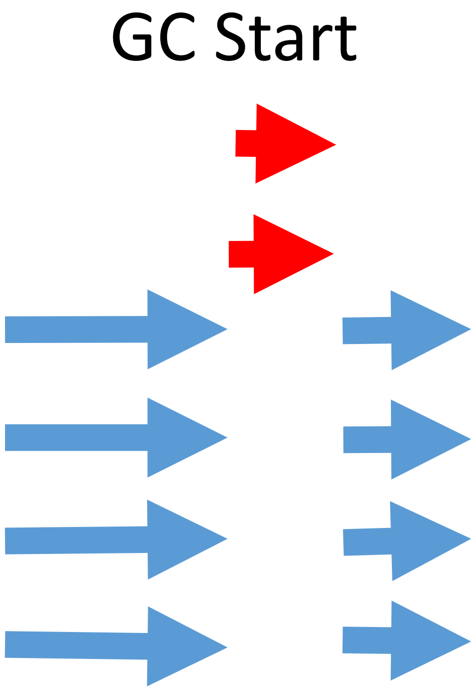
Heap

-Xgcpolicy:metronome GC

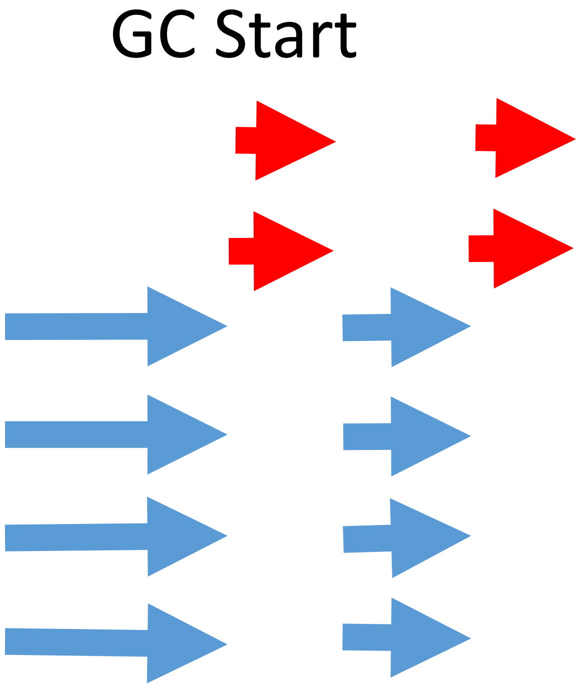
-Xgcpolicy:metronome GC



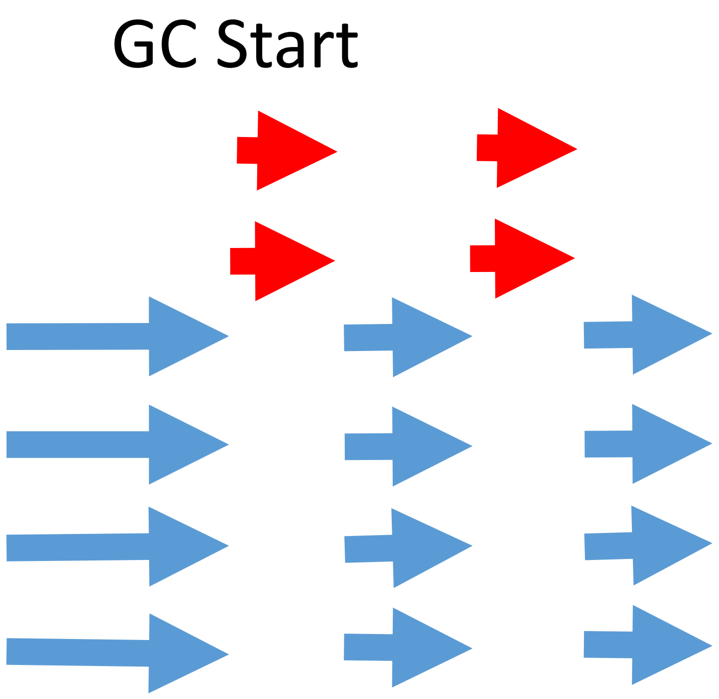
-Xgcpolicy:metronome GC



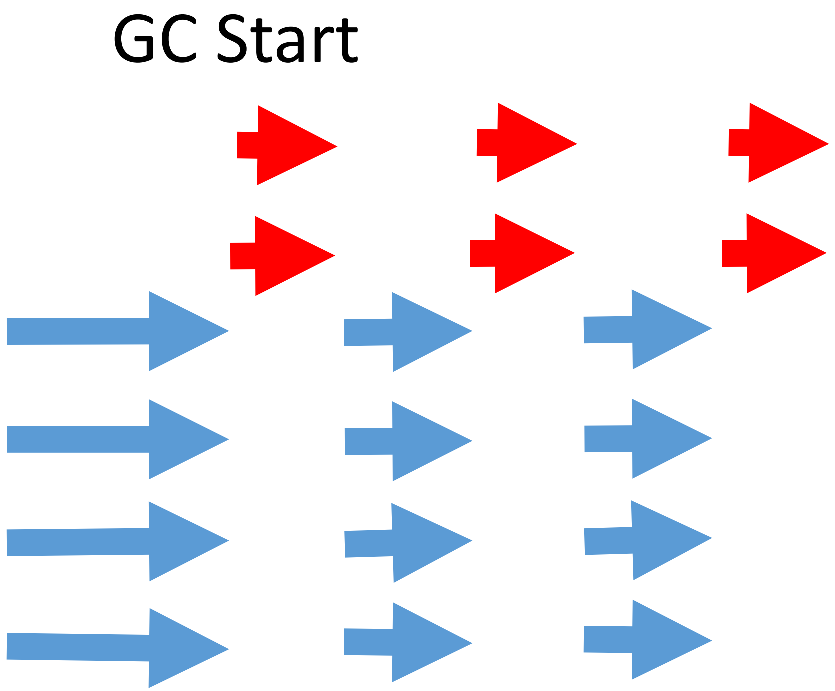
-Xgcpolicy:metronome GC



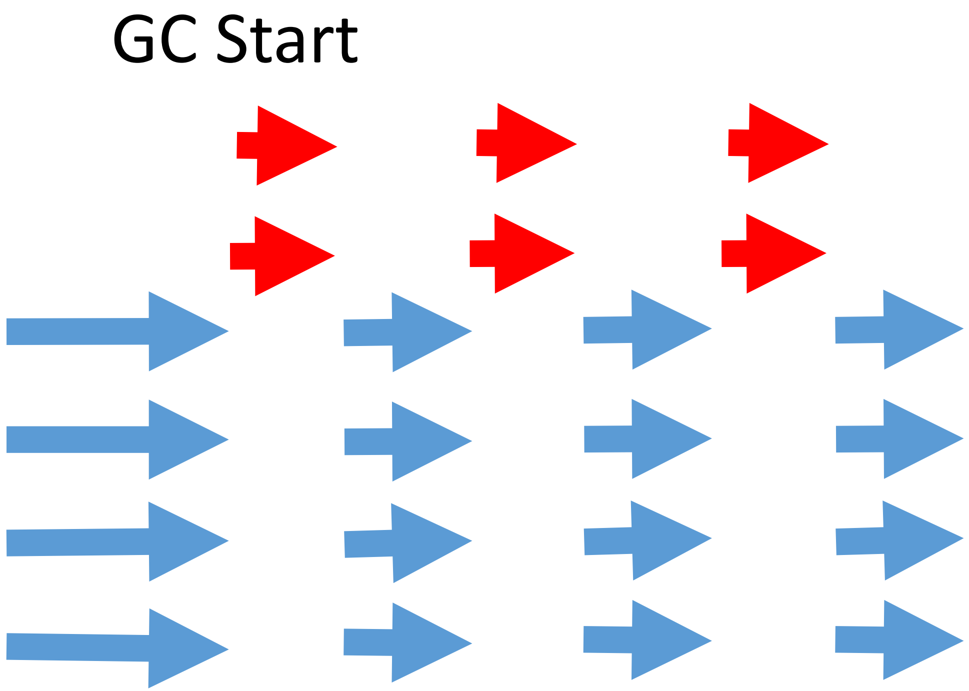
-Xgcpolicy:metronome GC



-Xgcpolicy:metronome GC

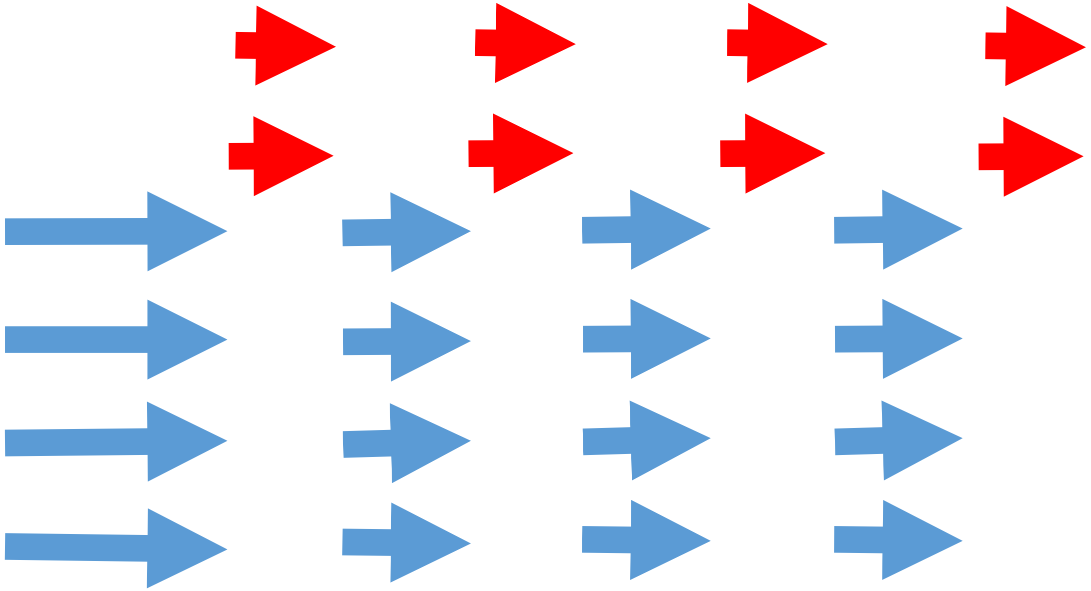


-Xgcpolicy:metronome GC

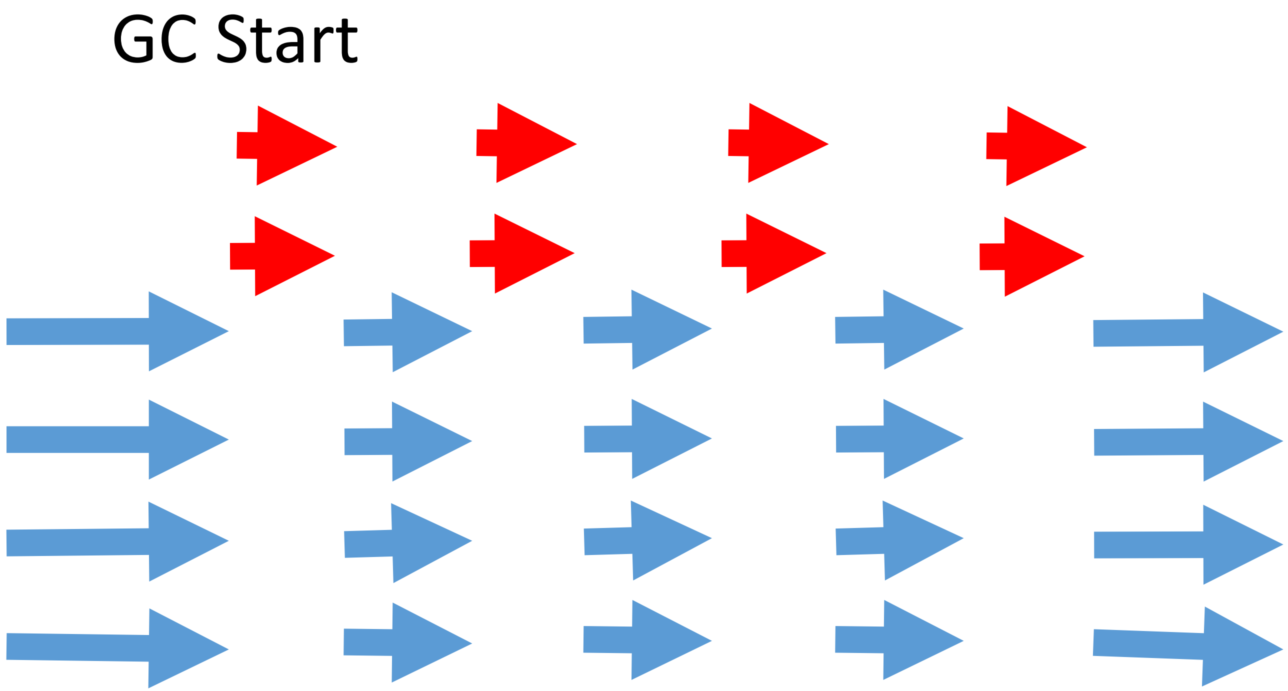


-Xgcpolicy:metronome GC

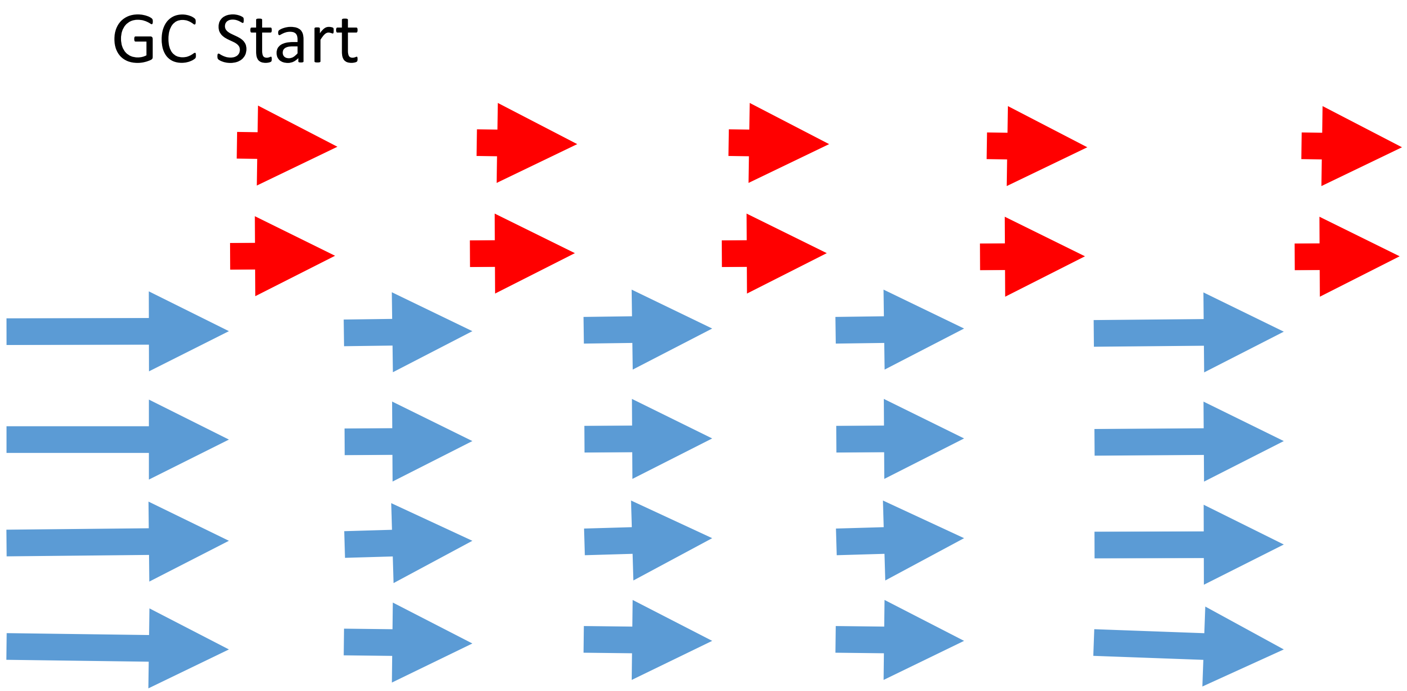
GC Start



-Xgcpolicy:metronome GC

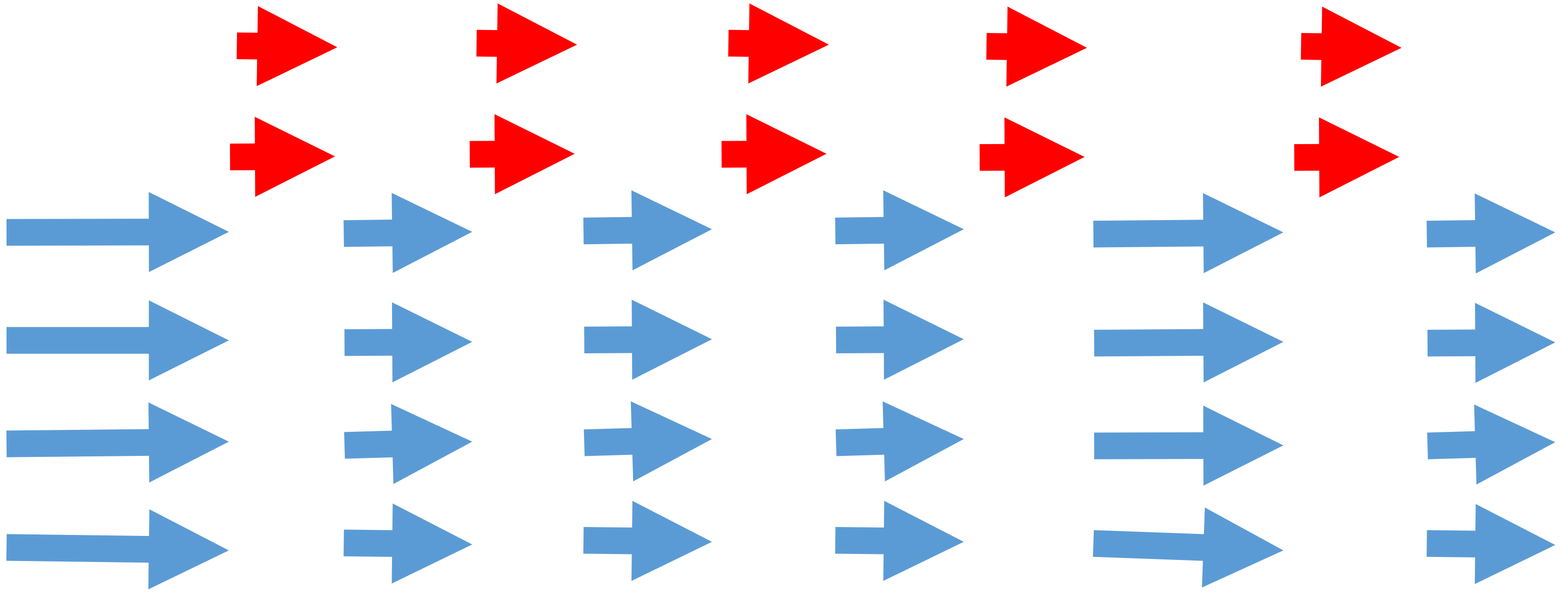


-Xgcpolicy:metronome GC

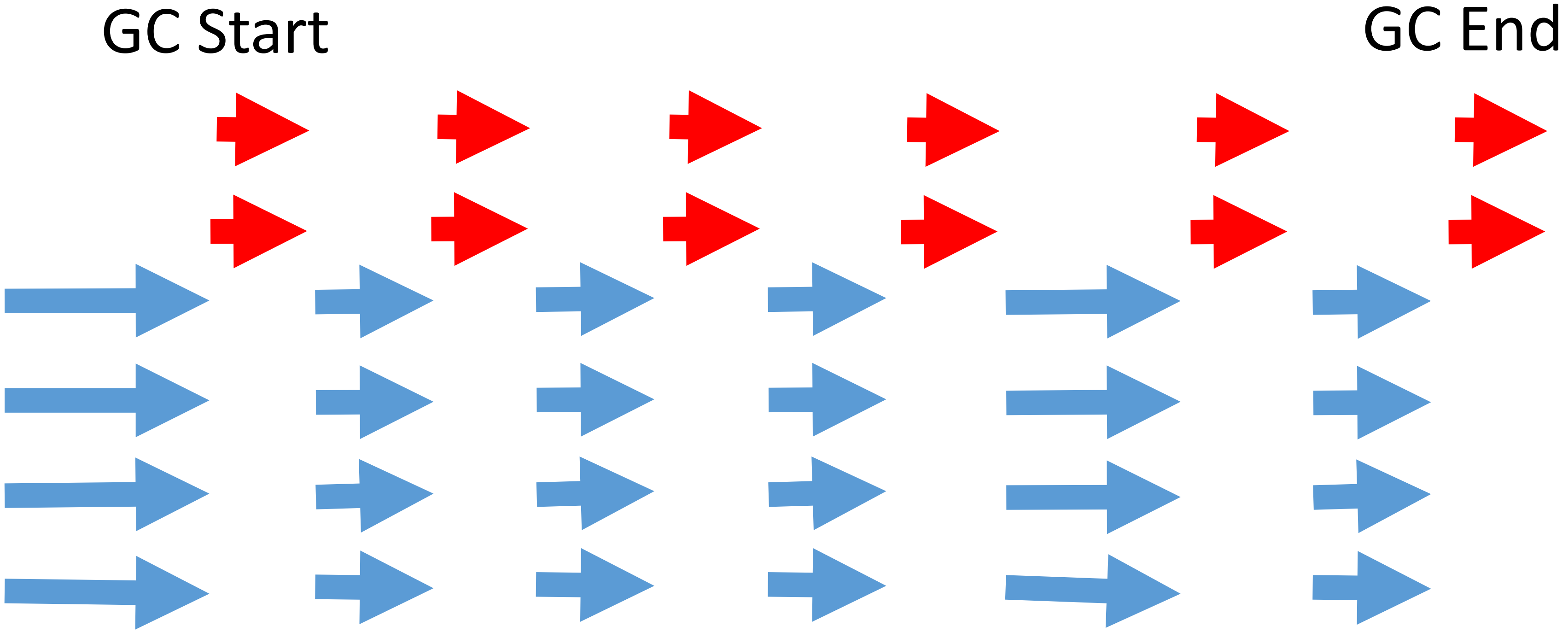


-Xgcpolicy:metronome GC

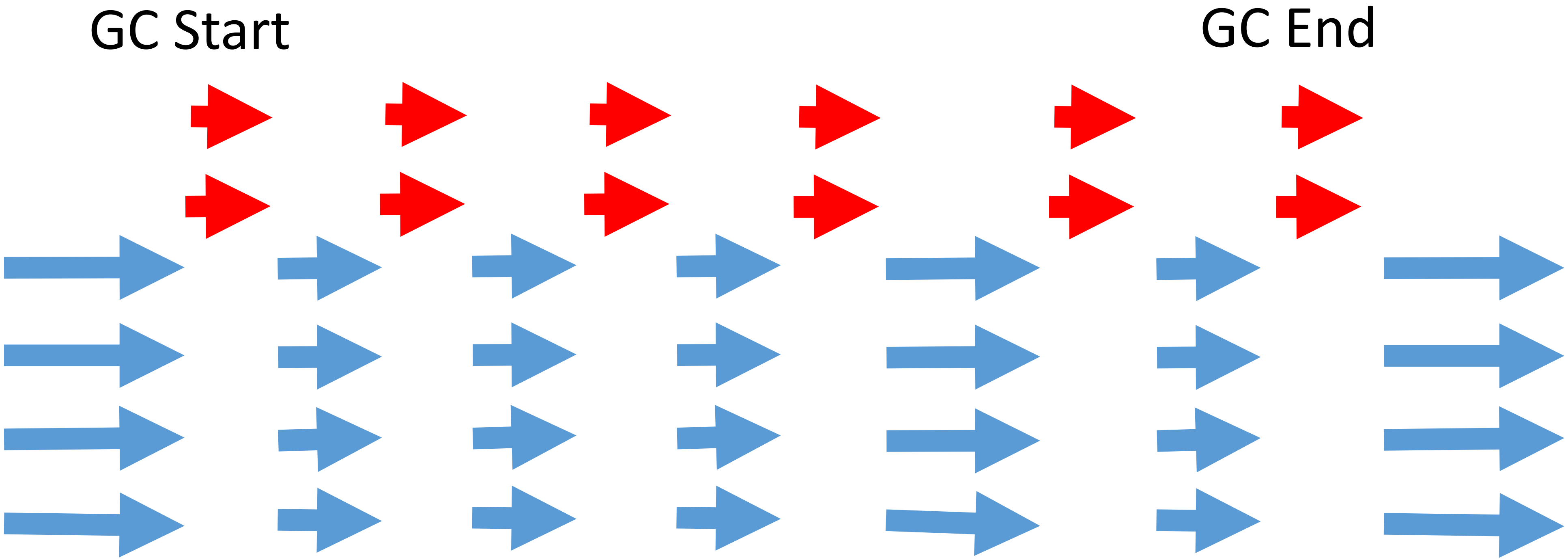
GC Start



-Xgcpolicy:metronome GC



-Xgcpolicy:metronome GC



-Xgcpolicy:metronome SATB write barrier

- A snapshot at the beginning barrier makes the guarantee that all objects alive at the beginning of the GC will be alive at the end
- This causes a lot of floating garbage but ensures correctness for the incremental collector
- The barrier has to be performed before the store

-Xgcpolicy:metronome SATB write barrier

- How is the write barrier implemented?

```
private void setField(Object A, Object C) {
   A.field1 = C;
}
```

-Xgcpolicy:metronome SATB write barrier

- How is the write barrier implemented?

```
private void setField(Object A, Object C) {
    Object temp = A.field1;
    if (barrier isActive) {
        remember(temp):
    }
    A.field1 = C;
}
```

-Xgcpolicy:metronome GC

- Metronome uses the same marking and sweeping as optthruput
- Root scanning, marking and sweeping all have yield points
- The remembered set is added as a new root
- If the pause time has been reached the GC pauses and lets the application run again

GC policy quick guide

GC Policy	Domain	Memory used in addition to heap (% of Xmx)	Throughput
gencon*	Webservers, desktop applications, runs most apps well	9	Highest
balanced	Very large heaps, large NUMA systems	13.5	High
metronome	Soft-realtime systems, trading applications, etc.	4.5	Low-Medium
optthruput	Small heaps with little GC	4.5	Medium
optavgpause	Why not gencon?	6	Low-Medium

Links

- Eclipse OMR

https://www.eclipse.org/omr/

- Eclipse OpenJ9

https://www.eclipse.org/openj9

- AdoptOpenJDK

https://adoptopenjdk.net/?variant=openjdk8-openj9

Questions???