OpenJ9

GC Optimizations You Never Knew Existed

Igor Braga
Jon Oommen

IBM

JPoint 2021

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

Outline

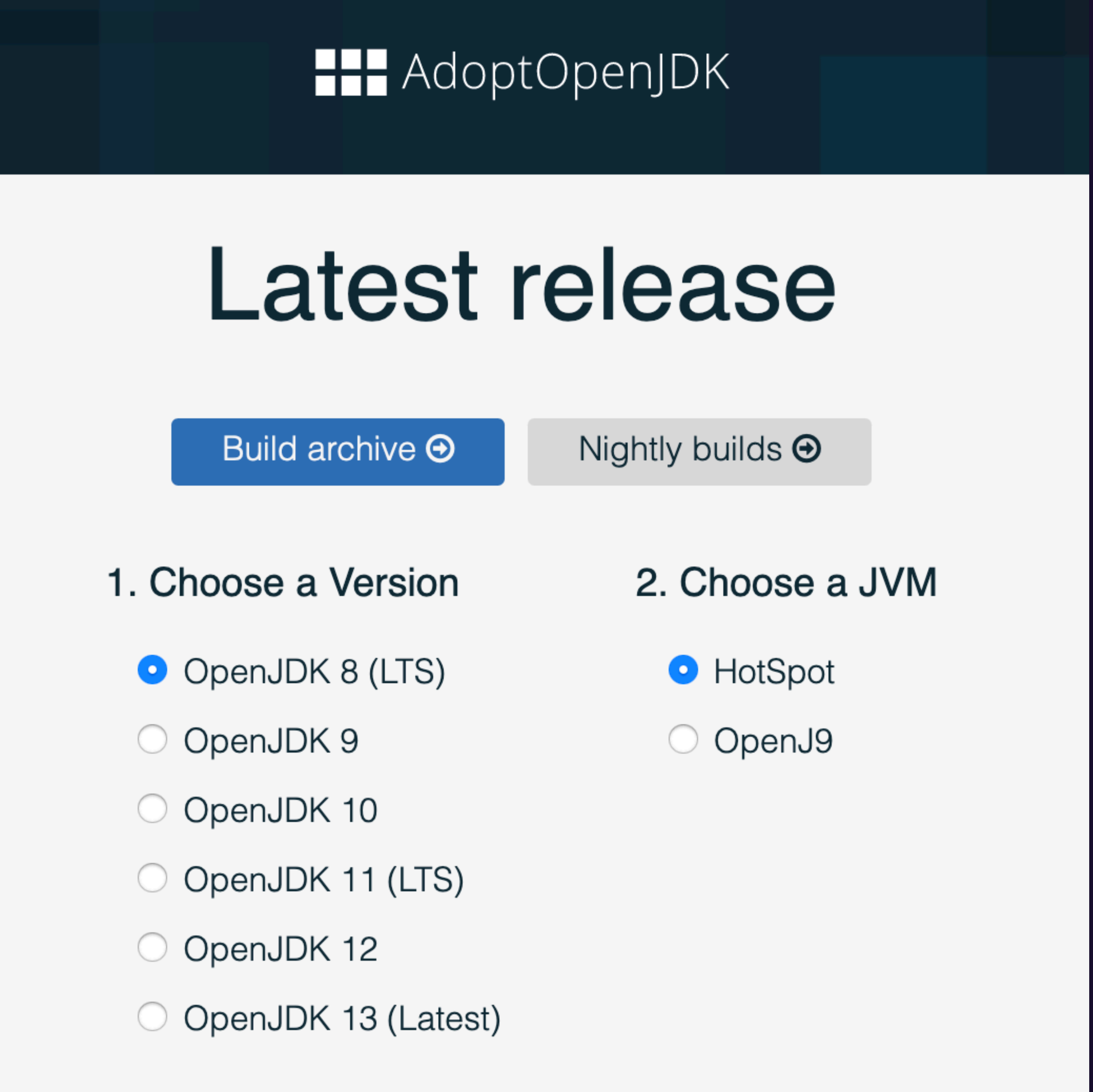
1. Introduction
2. Garbage Collection Algorithms
3. Dynamic Breadth First Scan Ordering
4. Double Map Arraylets
5. Off-heap Management
6. Summary

A Little bit About Igor

1. Software Developer at IBM
2. Masters University of Waterloo
3. Interested in Systems, Compilers, ML/AI
4. Tennis Addict

A Little bit About Jon

1. VM/GC Developer at IBM
2. Studied Systems Engineering at Carleton University
3. Most Interested in ML/AI, Blockchain Technology, and of course, GC
4. Fun Fact: 2nd youngest of 11 children, 5 of whom are Engineers



OpenJ9

The place to get OpenJDK builds

For both

OpenJDK + OpenJ9	or	OpenJDK + Hotspot

https://adoptopenjdk.net

IBM

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

Garbage Collection

Garbage Collection

"Garbage Collection (GC) is a form of automatic memory management. The garbage collector attempts to reclaim memory occupied by objects that are no longer in use by the application."

Garbage Collection

I

Allocation of memory

II

Identification of live data

III

Reclamation of garbage

Garbage Collection

Positives

- Automatic memory management
- Help reduce certain categories of bugs

☺

Negatives

- Require additional resources
- Causes unpredictable pauses
- May introduce runtime costs
- Application has little control of when memory is reclaimed

☹

GC Algorithms [1]

Region based

Mark Sweep

Mark Sweep Compact

Generational

Parallel

Concurrent

Reference counting

Garbage Collection Policies

-Xgcpolicy:

gencon CS – pauseless collector

balanced – region based collector

-Xgcpolicy:gencon

Generational copy collector

Provides a significant reduction in GC STW pause times

Introduces write barrier for the remembered set

Concurrent global marking phase

-Xgcpolicy:gencon Heap

Heap is divided into Nursery and Tenure Spaces

Nursery	Tenure

Heap

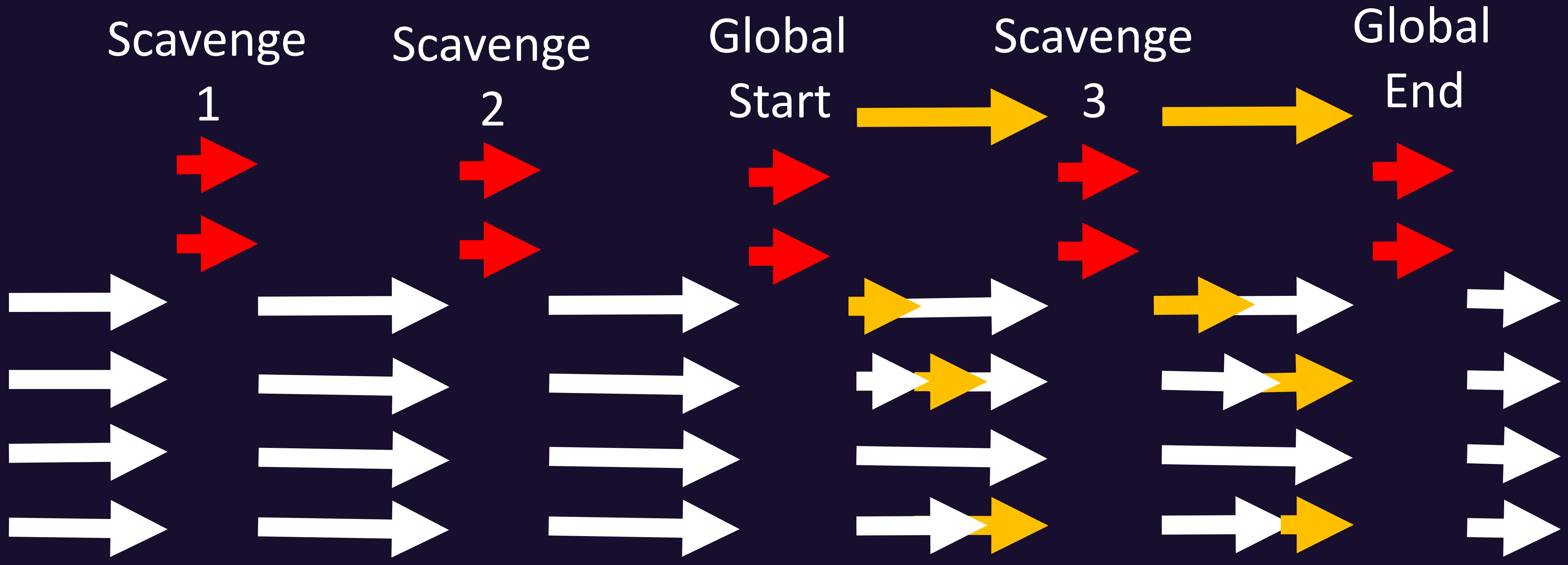
-Xgcpolicy:gencon heap

Heap is divided into Nursery and Tenure Spaces

The Nursery is divided into 2 logical spaces: Allocate and Survivor

Heap

-Xgcpolicy:gencon GC



-Xgcpolicy:gencon GC

Write Barrier

Why do we need a write barrier?

-Xgcpolicy:gencon GC

Write Barrier

Why do we need a write barrier?

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

-Xgcpolicy:gencon GC

Write Barrier

How’s the write barrier implemented?

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

-Xgcpolicy:gencon GC

Write Barrier

How's 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 GC

Write Barrier

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

-Xgcpolicy:gencon GC

Concurrent Scavenger

Generational copy collector

Introduces read Barrier for Concurrent Compact

Pauseless GC

-Xgcpolicy:gencon GC

Concurrent Scavenger

Heap is divided into Nursery and Tenure Spaces

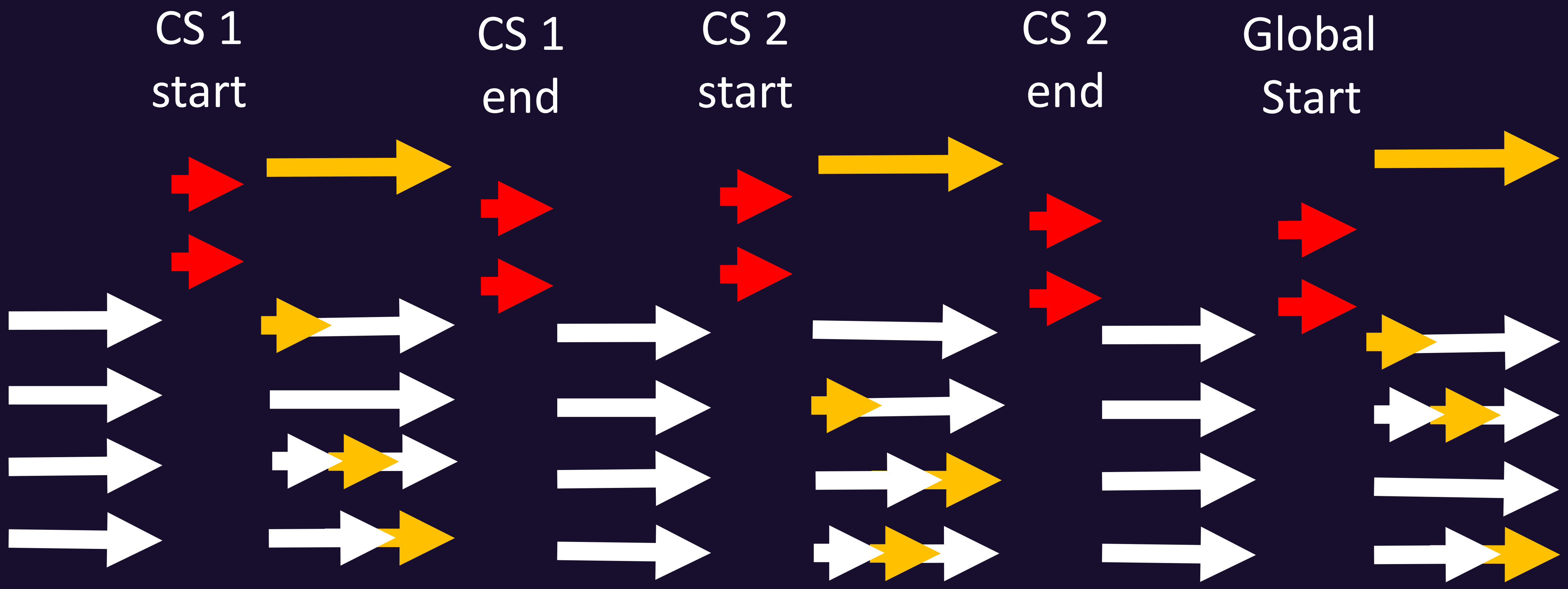
The Nursery is divided into 2 logical spaces: Allocate and Survivor

Allocate	Survivor	Tenure

Heap

-Xgcpolicy:gencon GC

Concurrent Scavenger

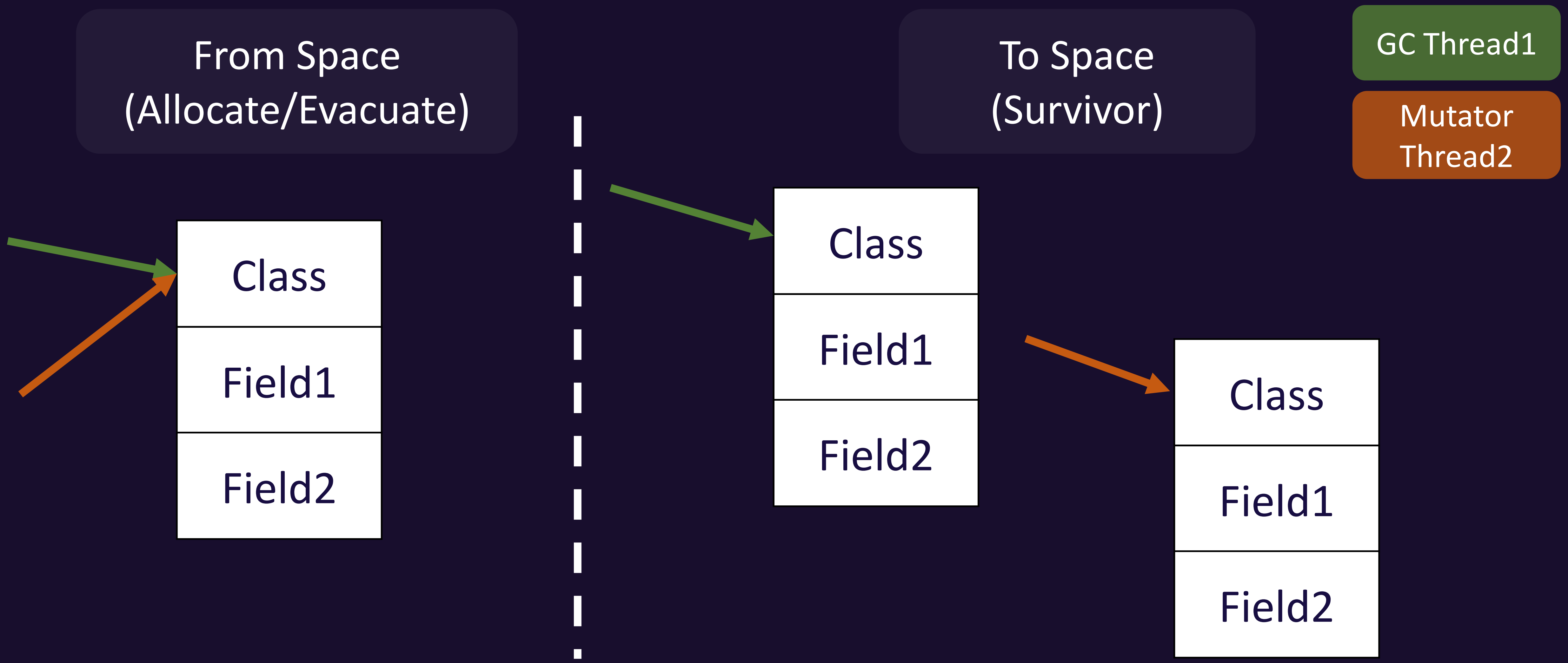


Concurrent Scavenger

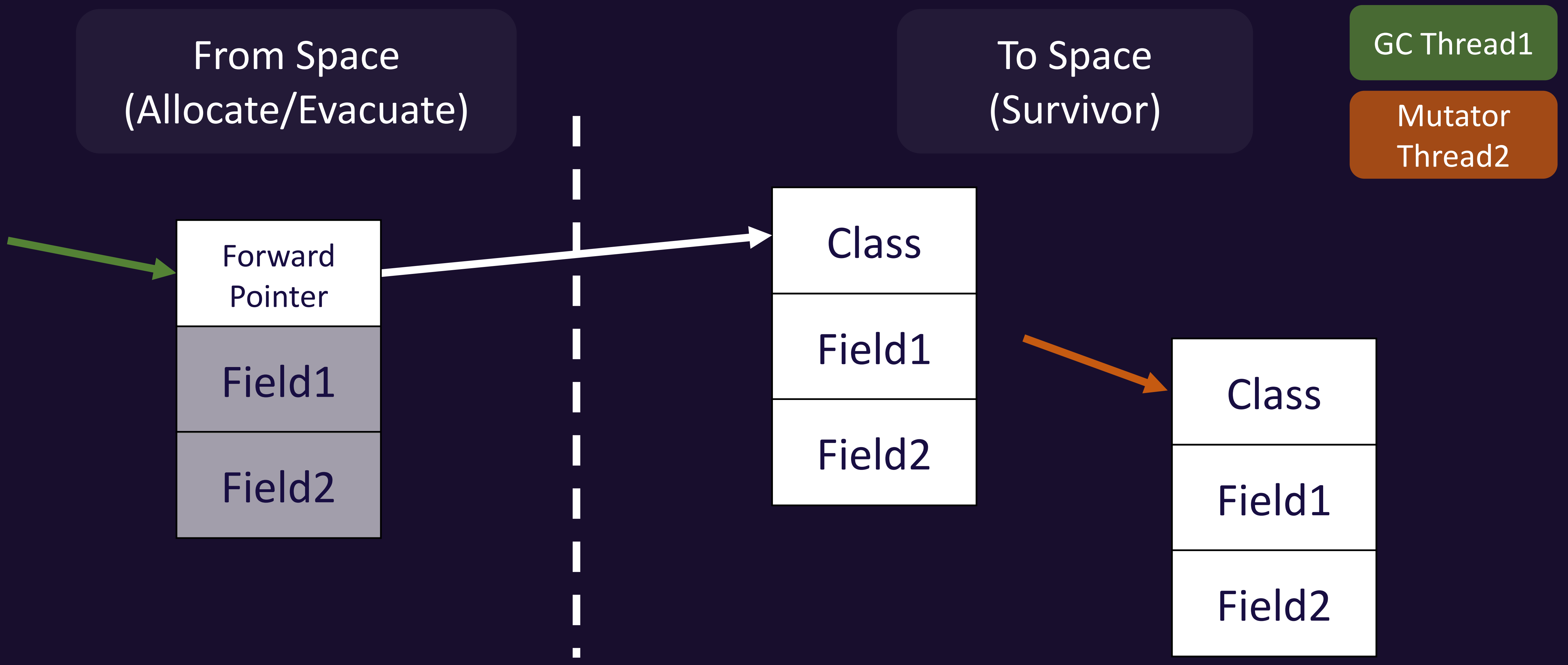
Multiple GC threads trying to move objects

And mutator threads trying to access these same objects

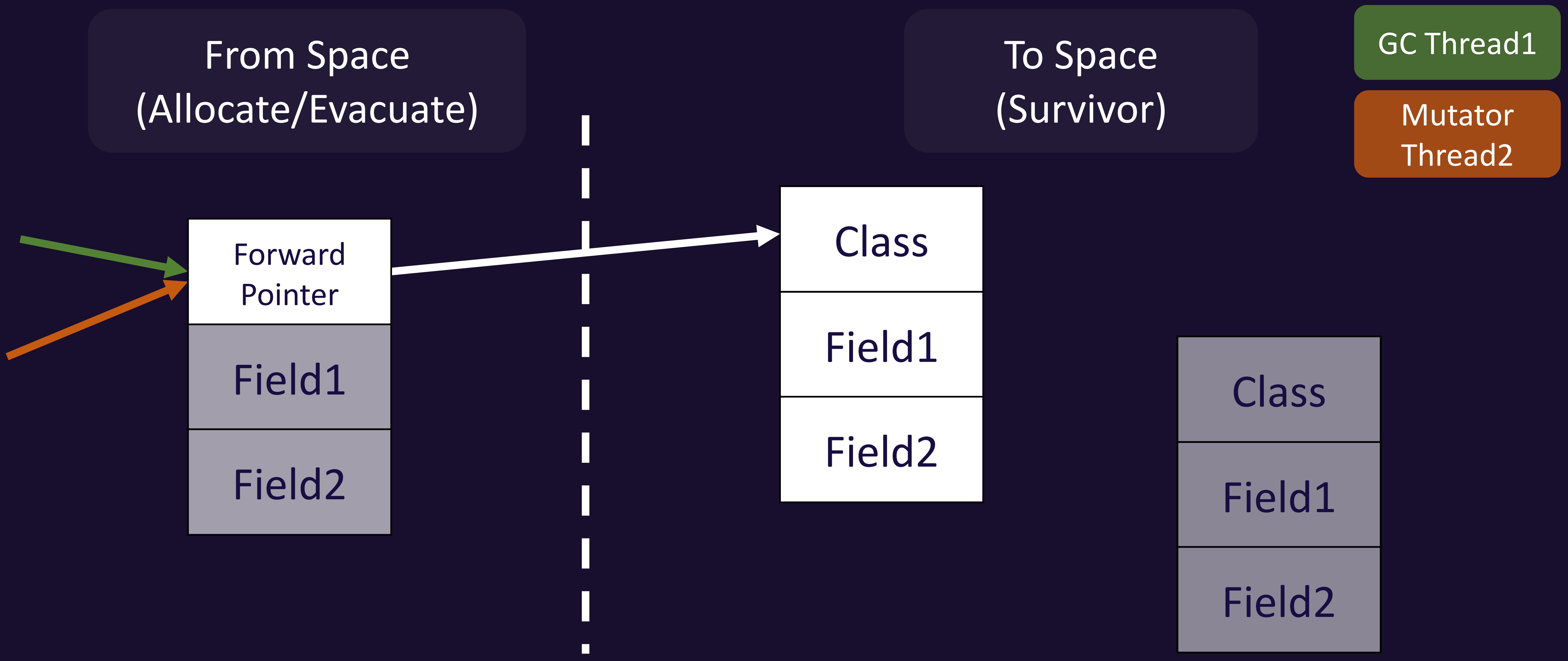
Concurrent Scavanger



Concurrent Scavanger



Concurrent Scavanger



Dynamic Breadth First Scan Ordering

Key Concepts

- **Example 1 – Gencon with Breadth First Scan Ordering**

- **Example 2 – Gencon with Dynamic Breadth First Scan Ordering**

Results & Takeaways

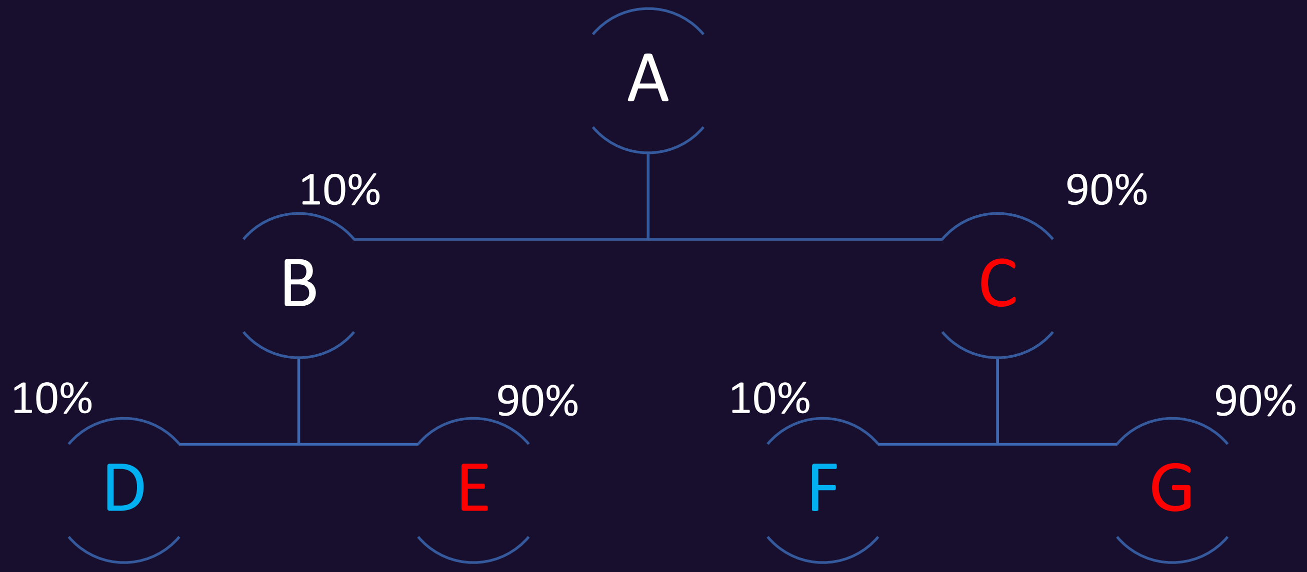
Locality

- 90/10 rule
- Caching
- Cache Prefetching
- Caching Hit to Miss ratio

Hot Fields and Access Patterns

- According to the 90/10 rule – if 90% of time is spent in 10% of code, there is likely some very hot object access patterns and very hot fields
- A hot field is a field that is frequently accessed by an object instance
- A hot access pattern is an object access pattern or path that occurs frequently

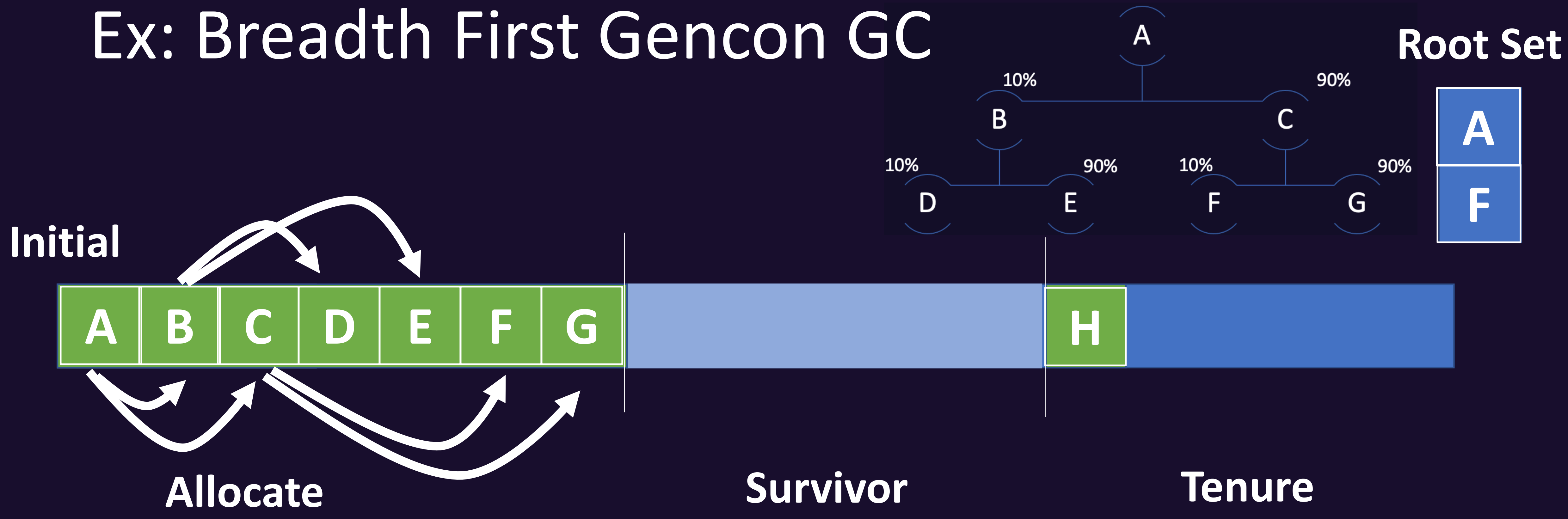
Hot Fields and Access Patterns - Example



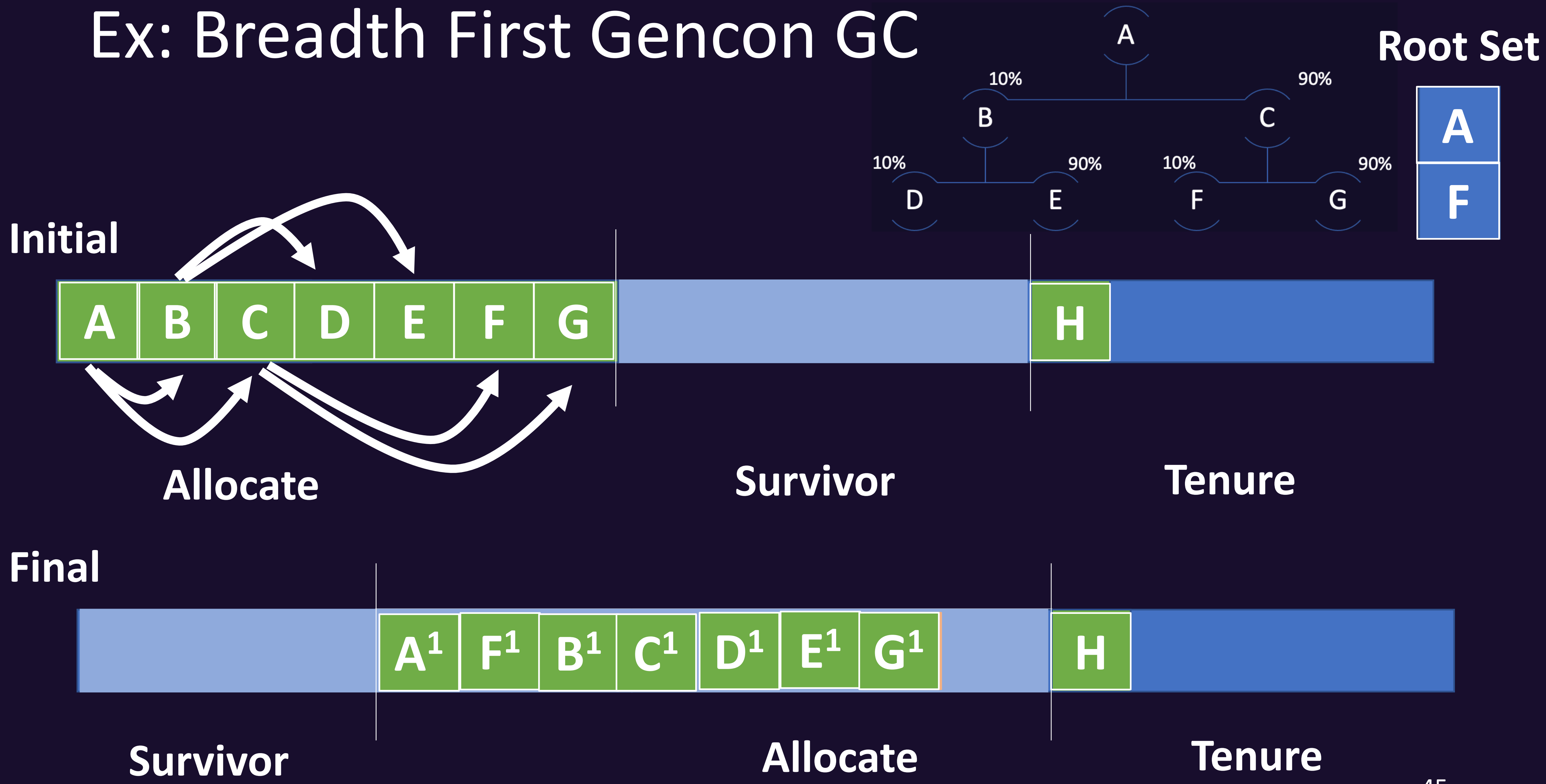
Hot Fields and Access Patterns - Example

Ideally, we would have A, C and G spatially localized in memory, and B and E spatially localized in memory

Ex: Breadth First Gencon GC



Ex: Breadth First Gencon GC



Gencon GC – Breadth First Issues

- With common access patterns of A→ C → G and B → E, the exisiting breadth first scan ordering implementation is clearly not optimal with regards to locality

Goal of Dynamic Breadth First Scan Ordering

- Optimize breadth first scan ordering for improved locality
- Leverage available JIT information for improved locality
- Render locality dependent optimization mechanisms more effective

Relevant Existing Infrastructure

- What is a compiler?
- What is an optimizing compiler?
- What is dynamic compilation?

What is a Compiler?

- A translator
  - Takes code written in one (source) language and produces equivalent code in another (target) language
- Possible source and target languages:
  - Source code to machine code (gcc, clang, etc.)
  - Source code to bytecode (javac)
  - Bytecode to machine code (Testarossa JIT)
  - ... and more

What is an Optimizing Compiler?

- Tries to produce “good” code
- Good (optimized) code should:
  - Execute faster
  - Require less memory
  - Consume less power

What is dynamic compilation?

- Interpreter invokes the compiler *just in time* before a method becomes a performance problem
- The Just-In-Time compiler (*jit*) turns bytecode into much faster native code
- Eclipse OpenJ9's Testarossa JIT compiler is an *optimizing compiler*

Relevant JIT Compiler Information Leveraged

- Applications consists of compilation instances (logical compilation entities – i.e. methods)
- The JIT Compiler is a tiered compilation compiler
- IBM Testarossa compilation levels - cold, warm, hot, very hot, scorching
- Each compilation is divided into “blocks” where the relative hotness of each code block within the compilation gets a normalized block “hotness” value from 1-10000

Relevant JIT Compiler Information Leveraged

- When a field is accessed within a compilation, we can compute an overall ”hotness” value approximation for the field access using:
  - the compilation optimization level of the method
  - the block “hotness” of the block within the compilation where the field was accessed
- This “hotness” value is computed for every field access of every compilation
- For each field of a class, we can aggregate these “hotness” values for all field access’ across all method compilations

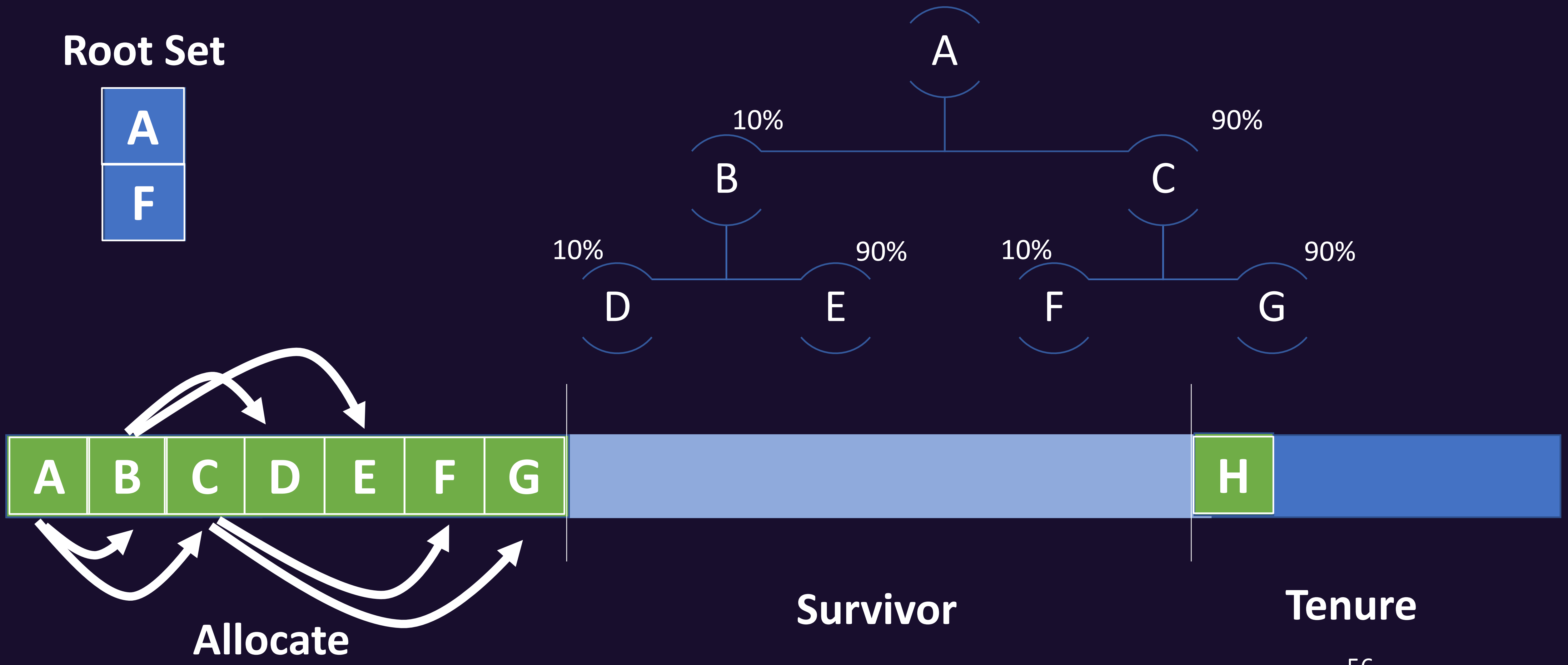
Relevant JIT Compiler Information Leveraged

- Hotness values are aggregated via a hotness aggregation algorithm
- Recursively depth copy the object's two hottest fields directly after an object is copied if hot fields for the object exist
- Assure minimum hotness requirements are met before allowing a field to be depth copied

Simple Field Hotness Calculation Example

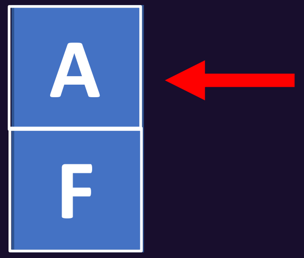
Class String - Field Char []				
Method	**Compilation Level**	**Compilation Level Weighting**	**Block Hotness Within Compilation Where Field is Accessed**	**Hotness Contribution**
A	Hot	10	50	500
B	Scorching	100	40	4000
C	Warm	1	1000	1000
		Current Total Field Hotness		**5500**

Ex: **Dynamic** Breadth First Gencon GC



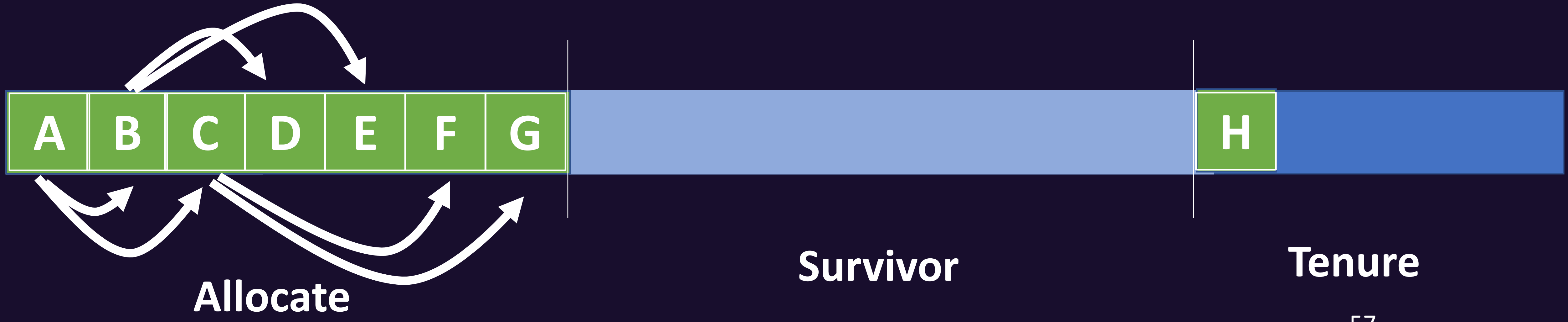
Ex: **Dynamic** Breadth First Gencon GC

Root Set

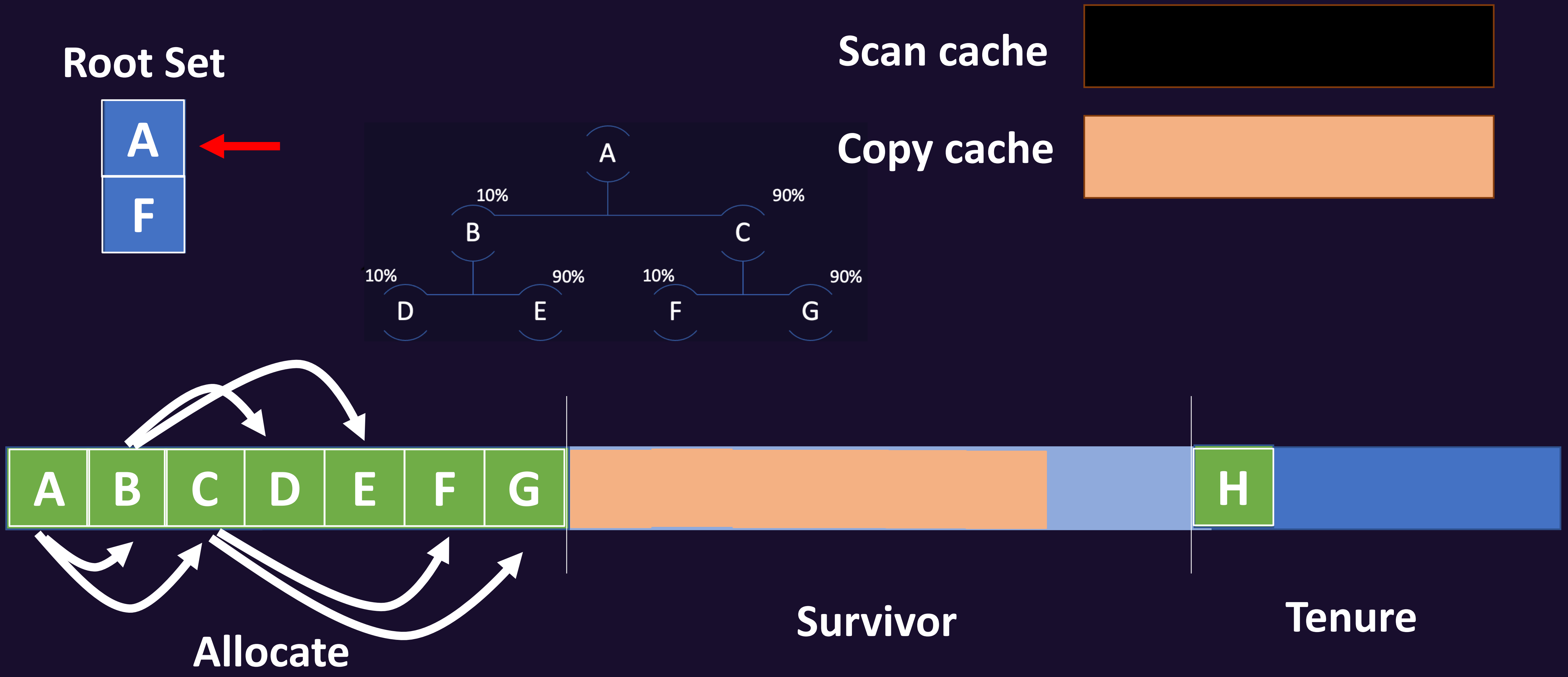


Scan cache

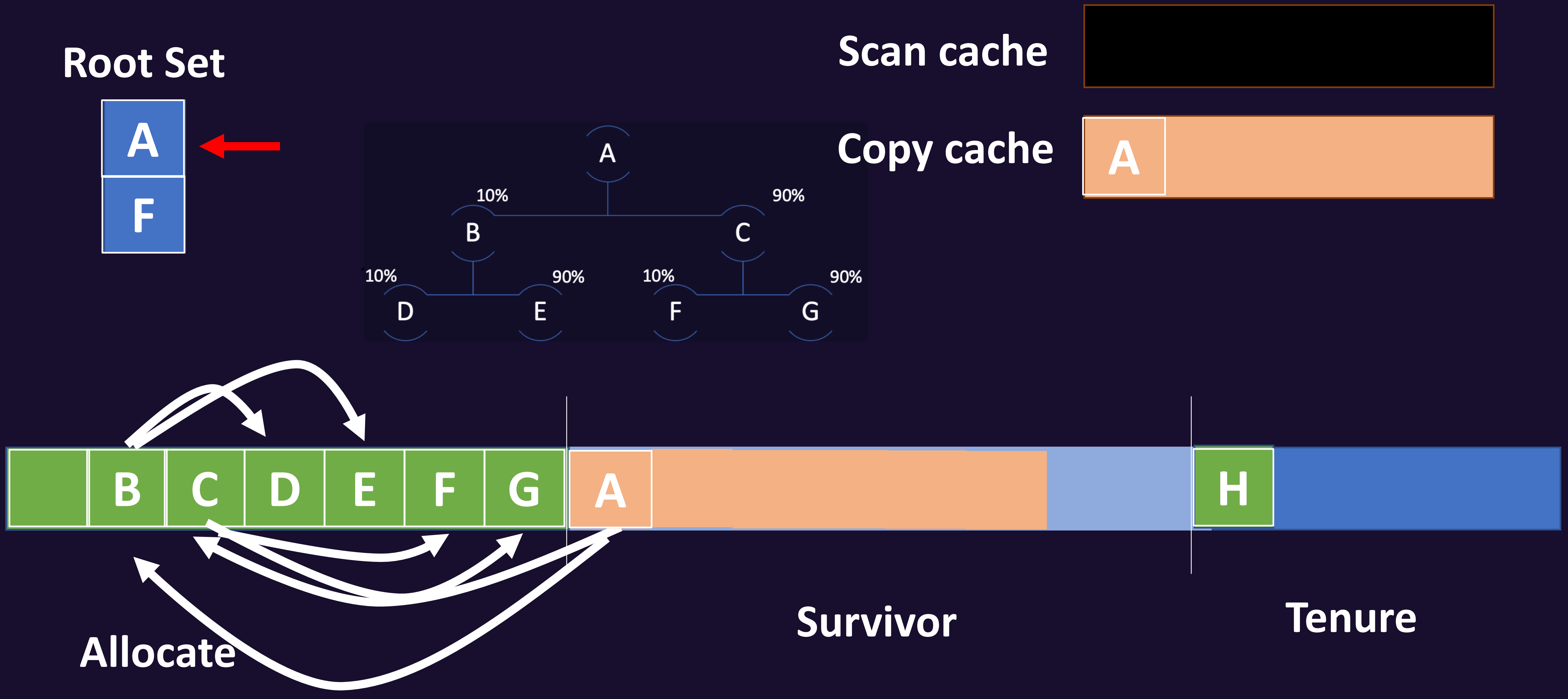
Copy cache



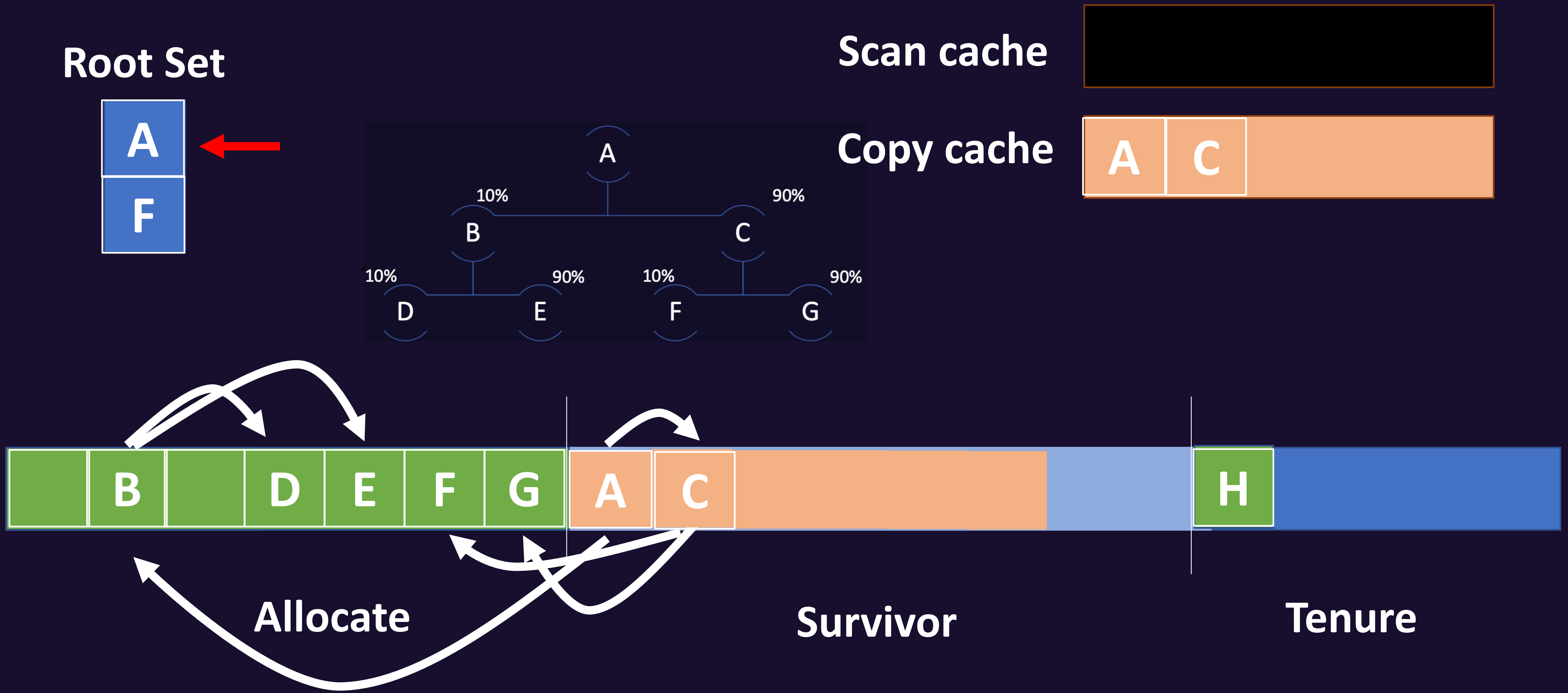
Ex: **Dynamic** Breadth First Gencon GC



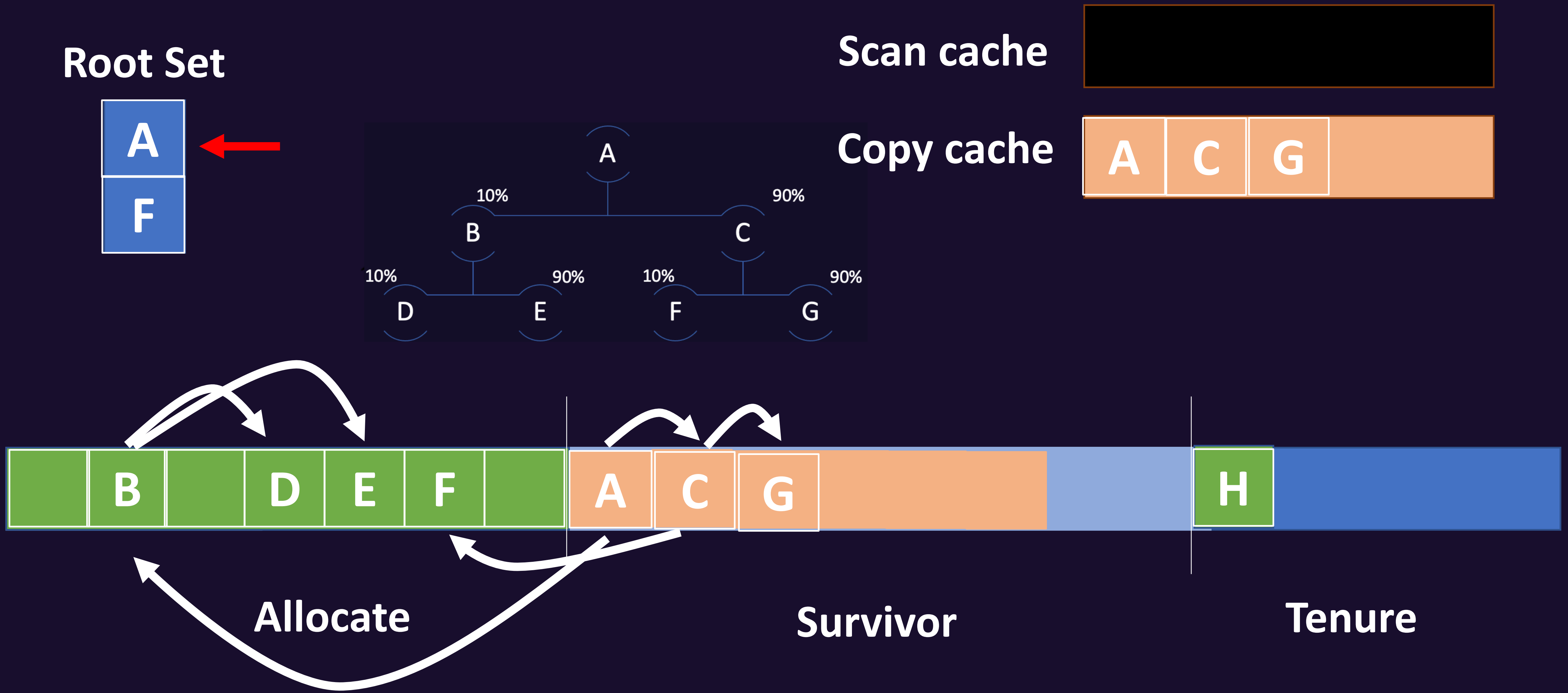
Ex: **Dynamic** Breadth First Gencon GC



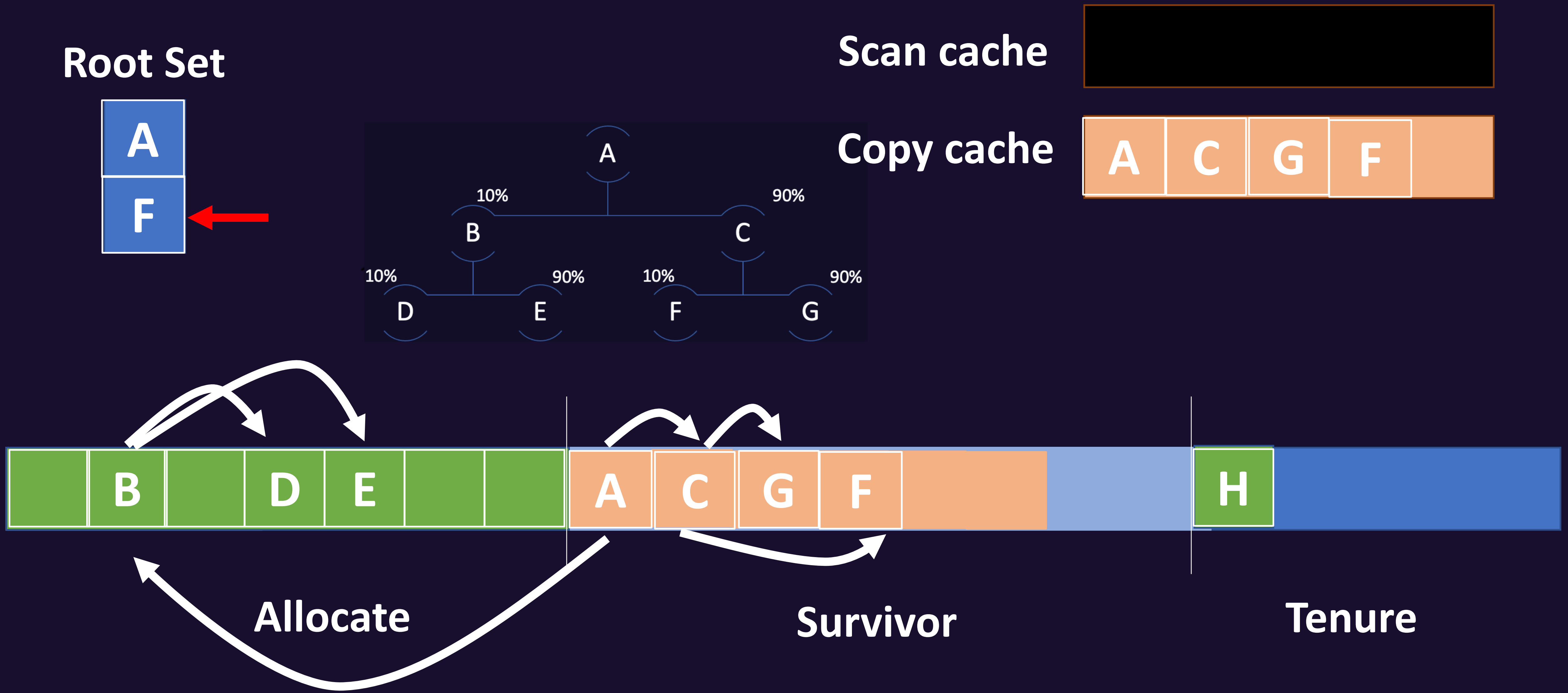
Ex: **Dynamic** Breadth First Gencon GC



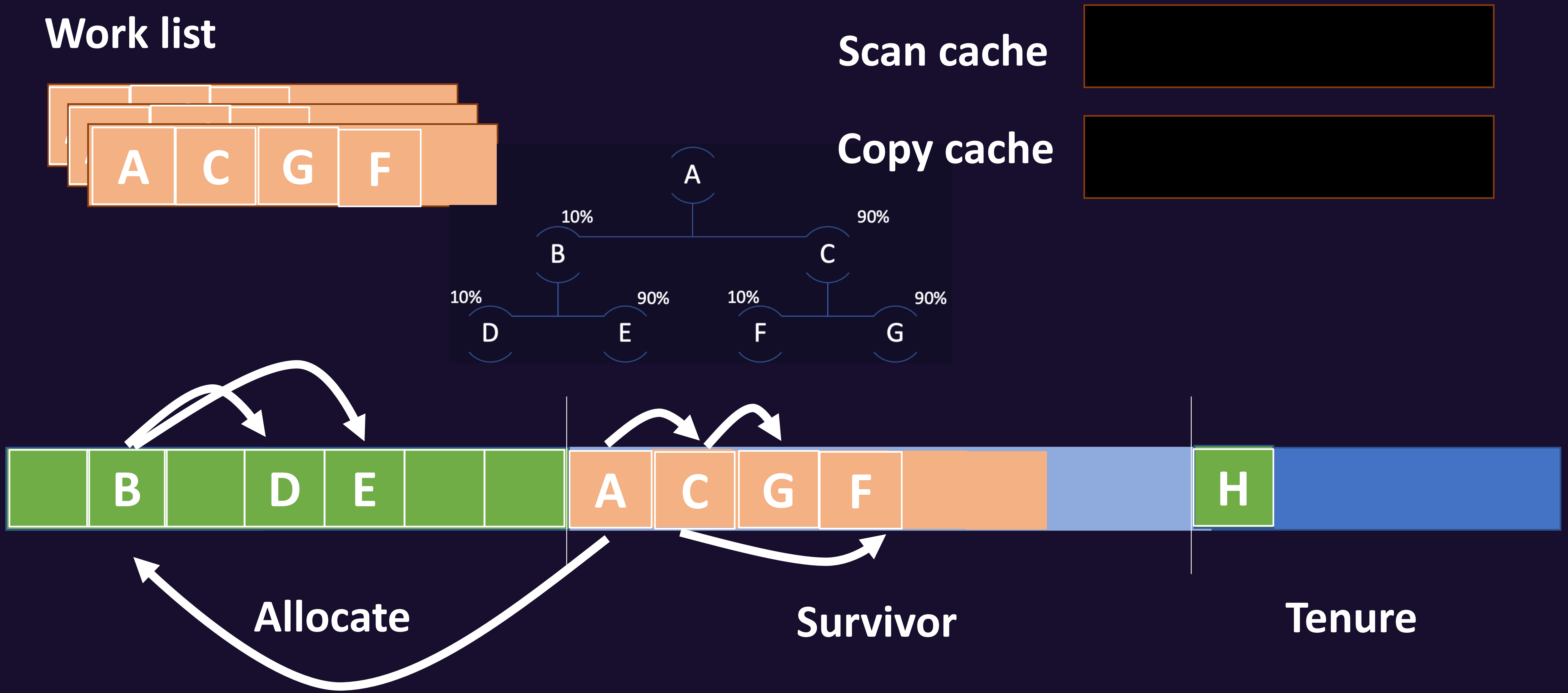
Ex: **Dynamic** Breadth First Gencon GC



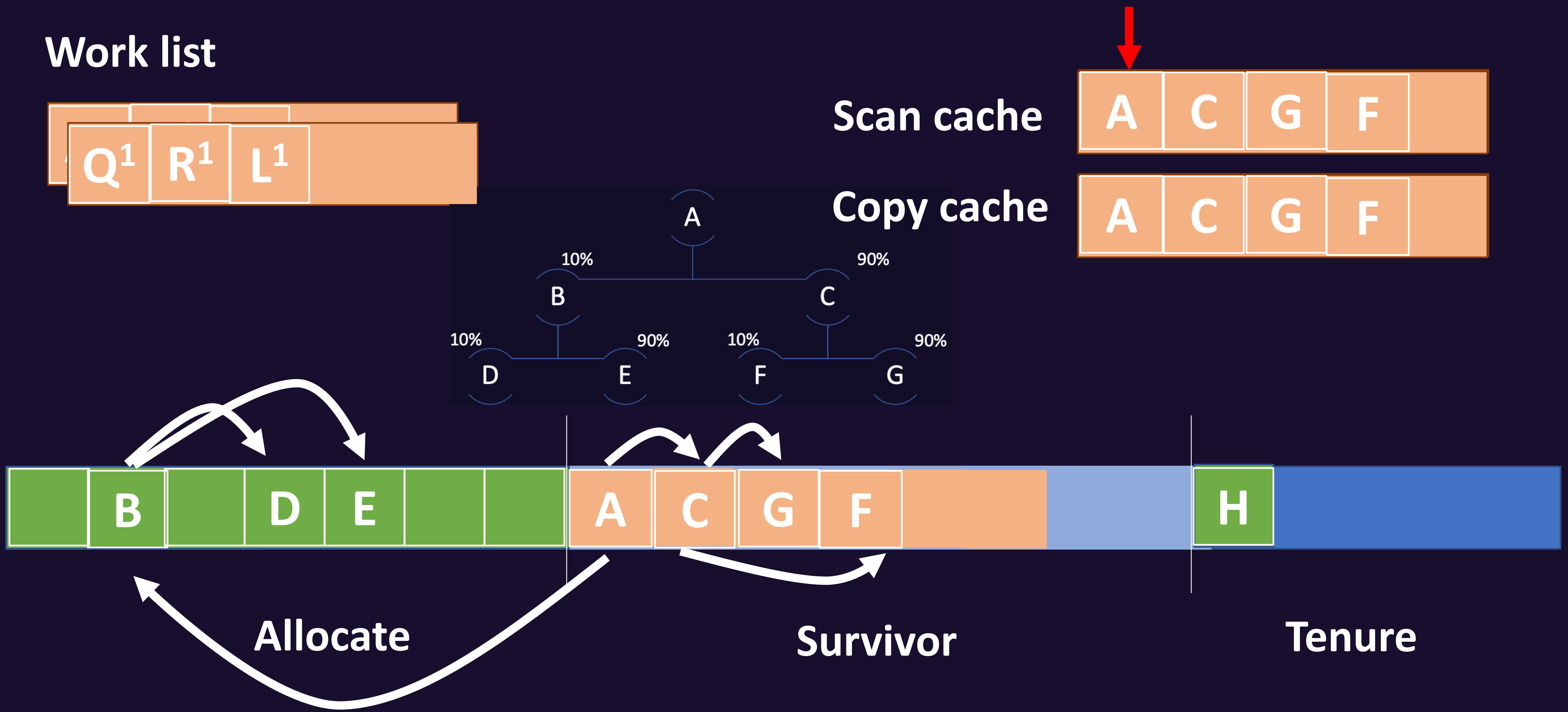
Gencon GC – Ex: **Dynamic** Breadth First



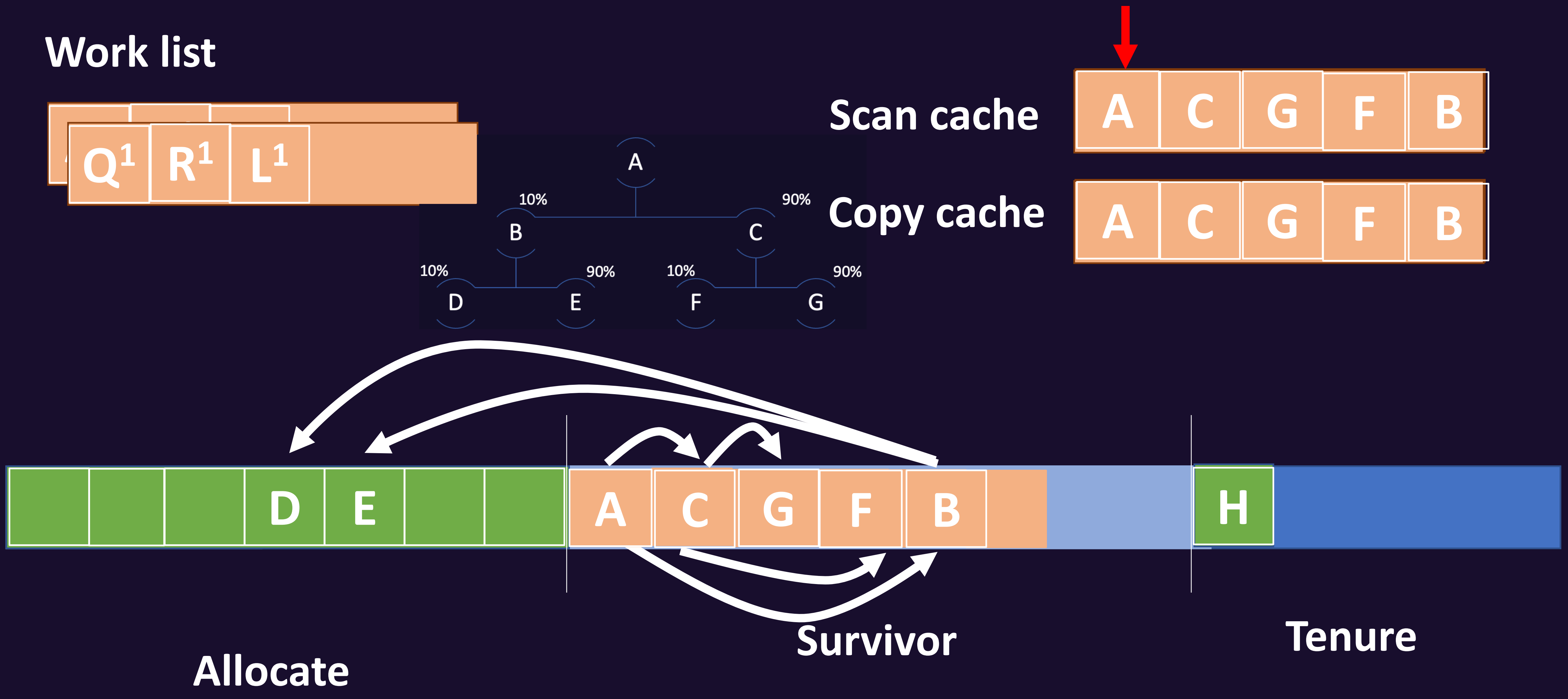
Gencon GC – Ex: **Dynamic** Breadth First



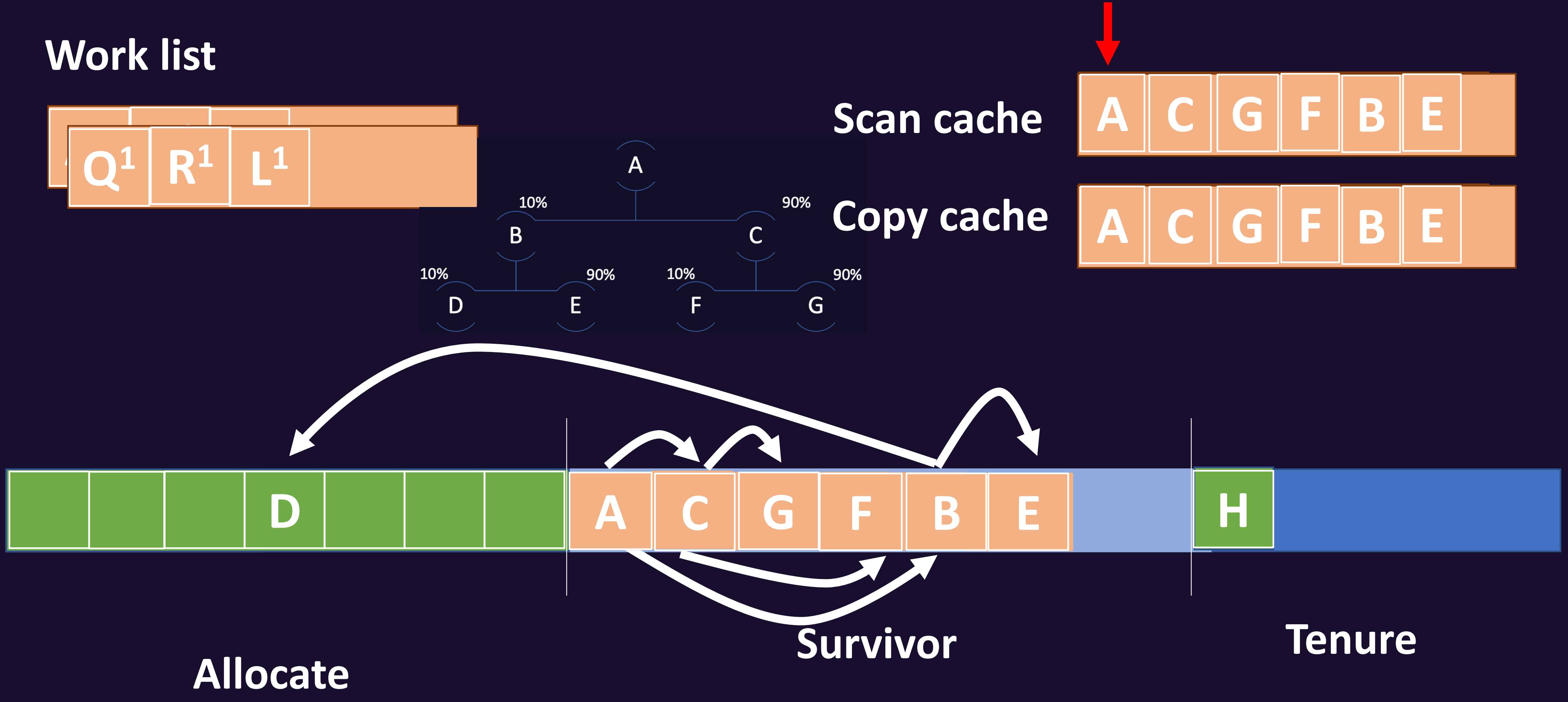
Gencon GC – Ex: **Dynamic** Breadth First



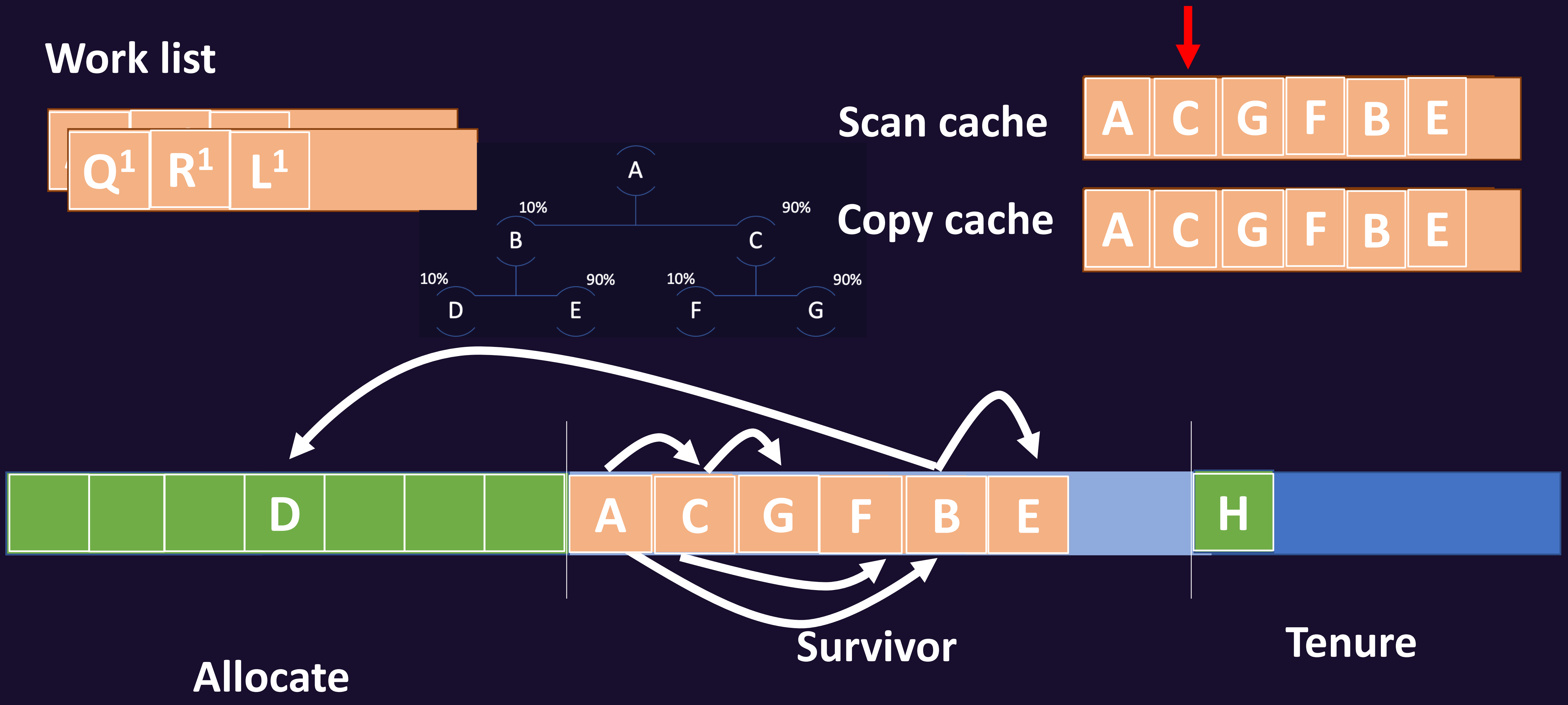
Gencon GC – Ex: **Dynamic** Breadth First



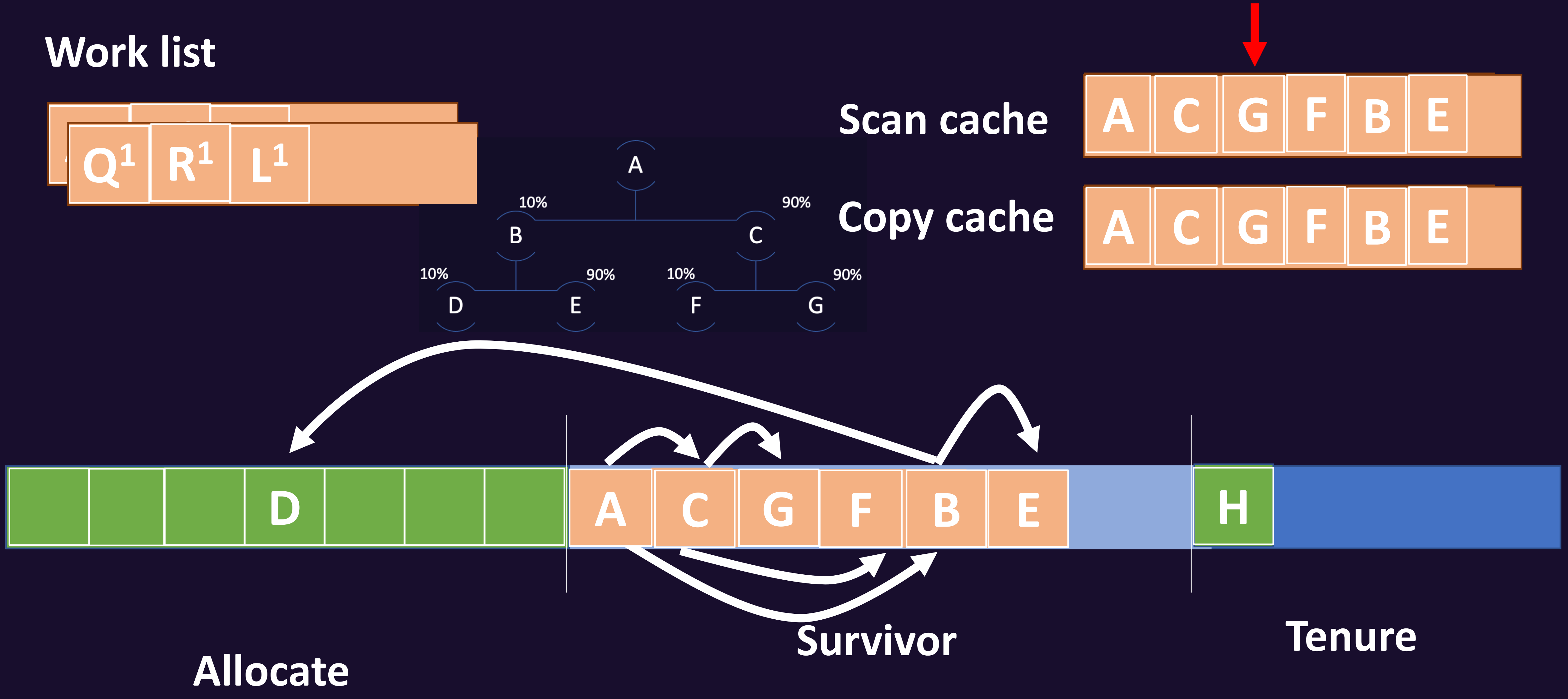
Gencon GC – Ex: **Dynamic** Breadth First



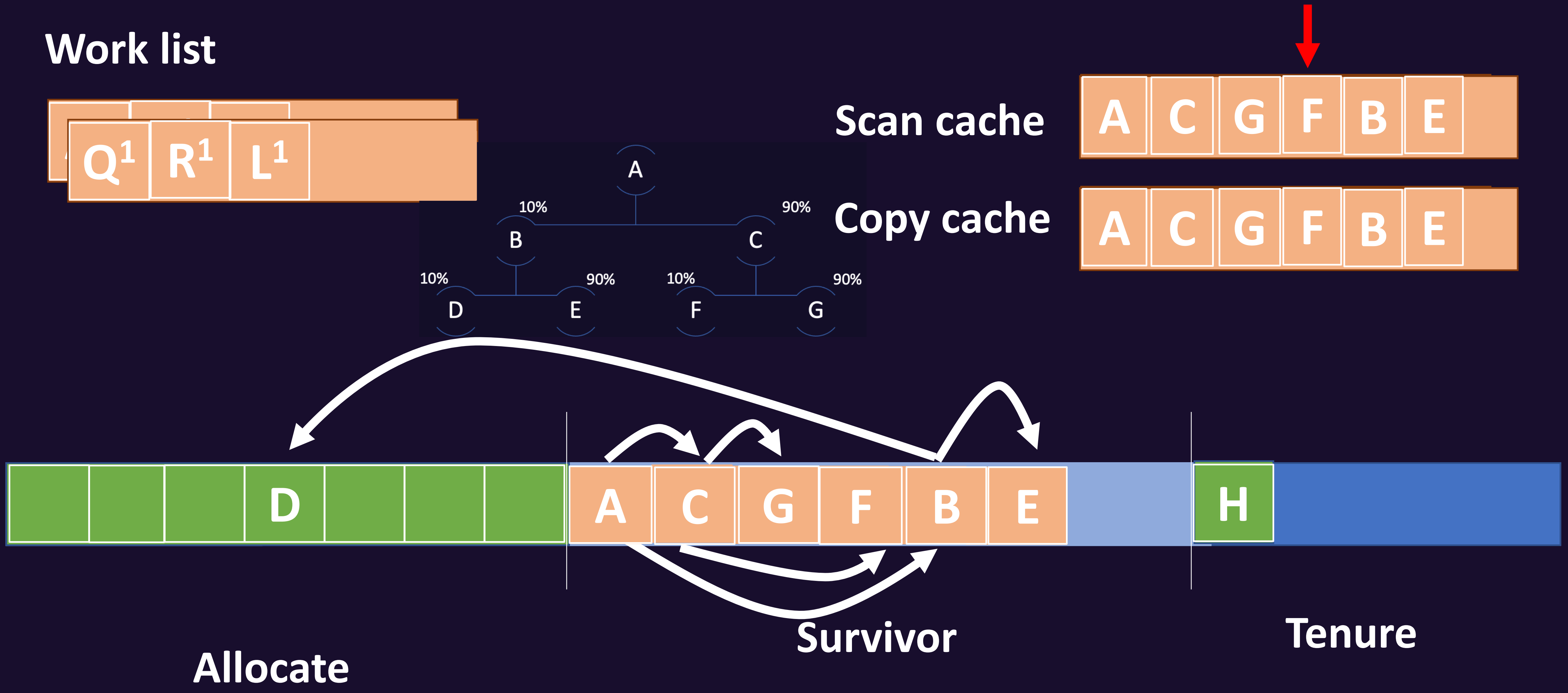
Gencon GC – Ex: **Dynamic** Breadth First



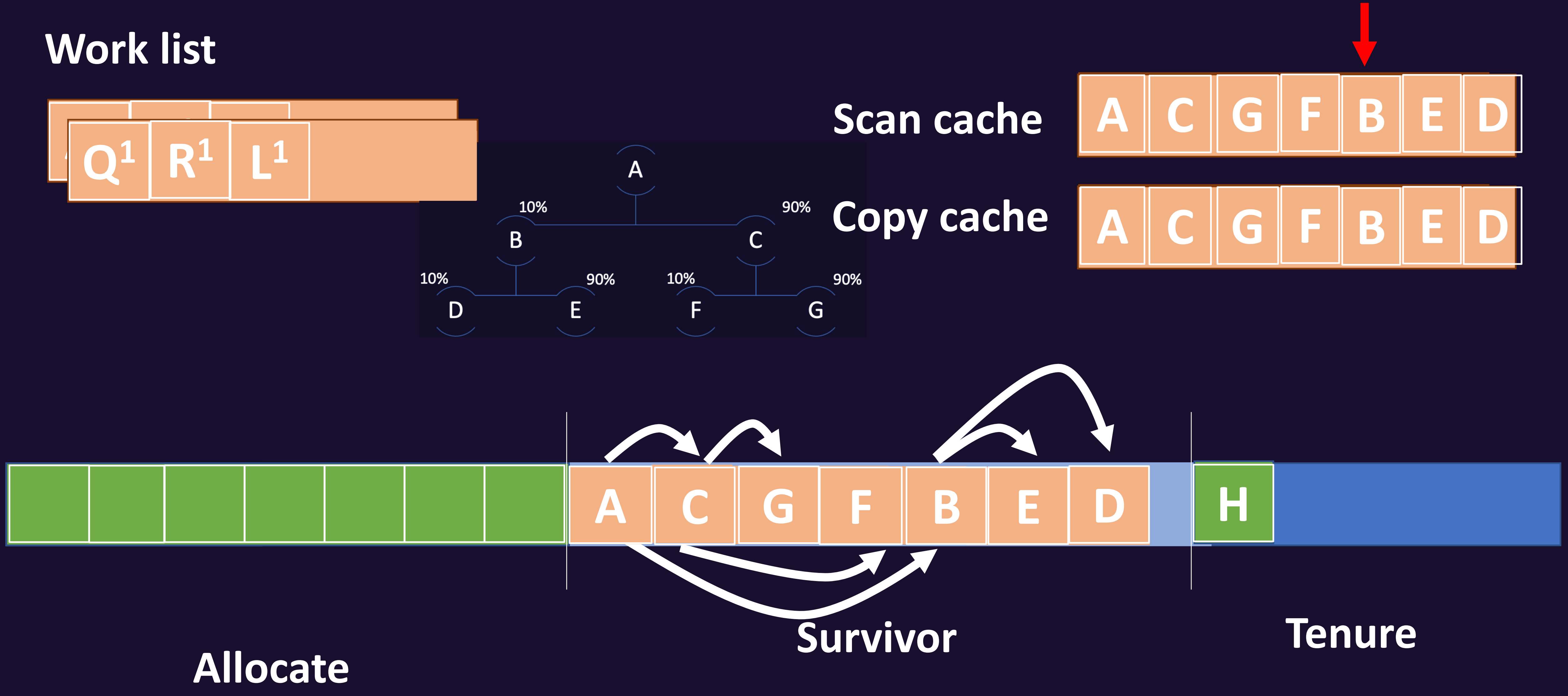
Gencon GC – Ex: **Dynamic** Breadth First



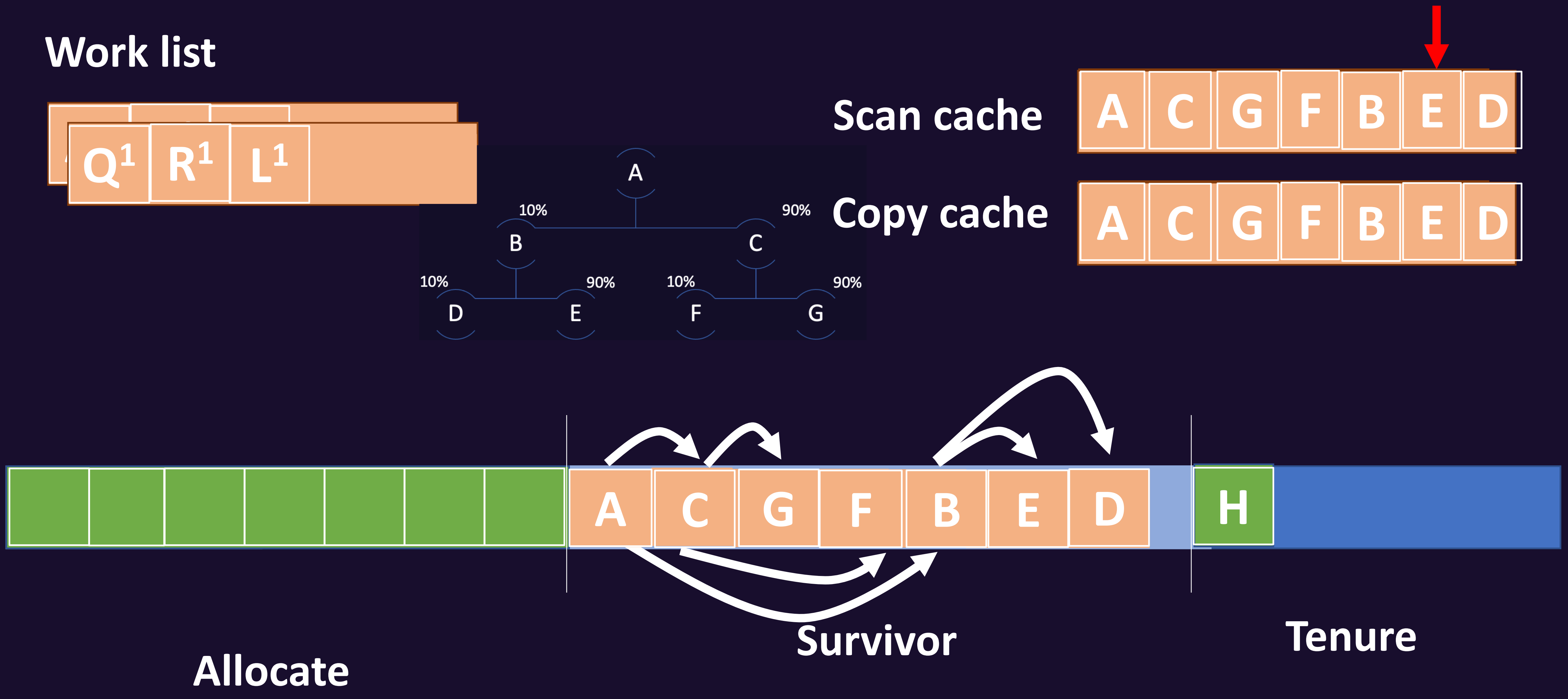
Gencon GC – Ex: **Dynamic** Breadth First



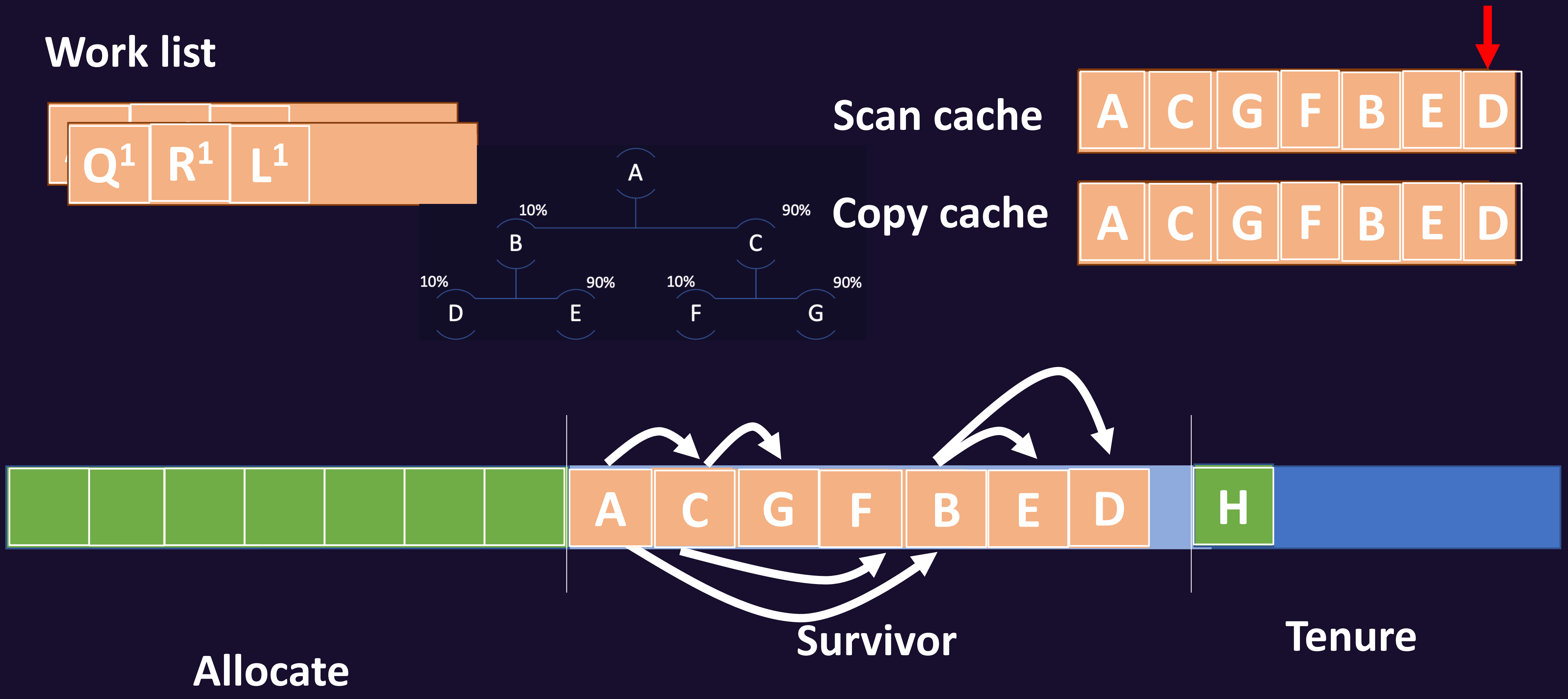
Gencon GC – Ex: **Dynamic** Breadth First



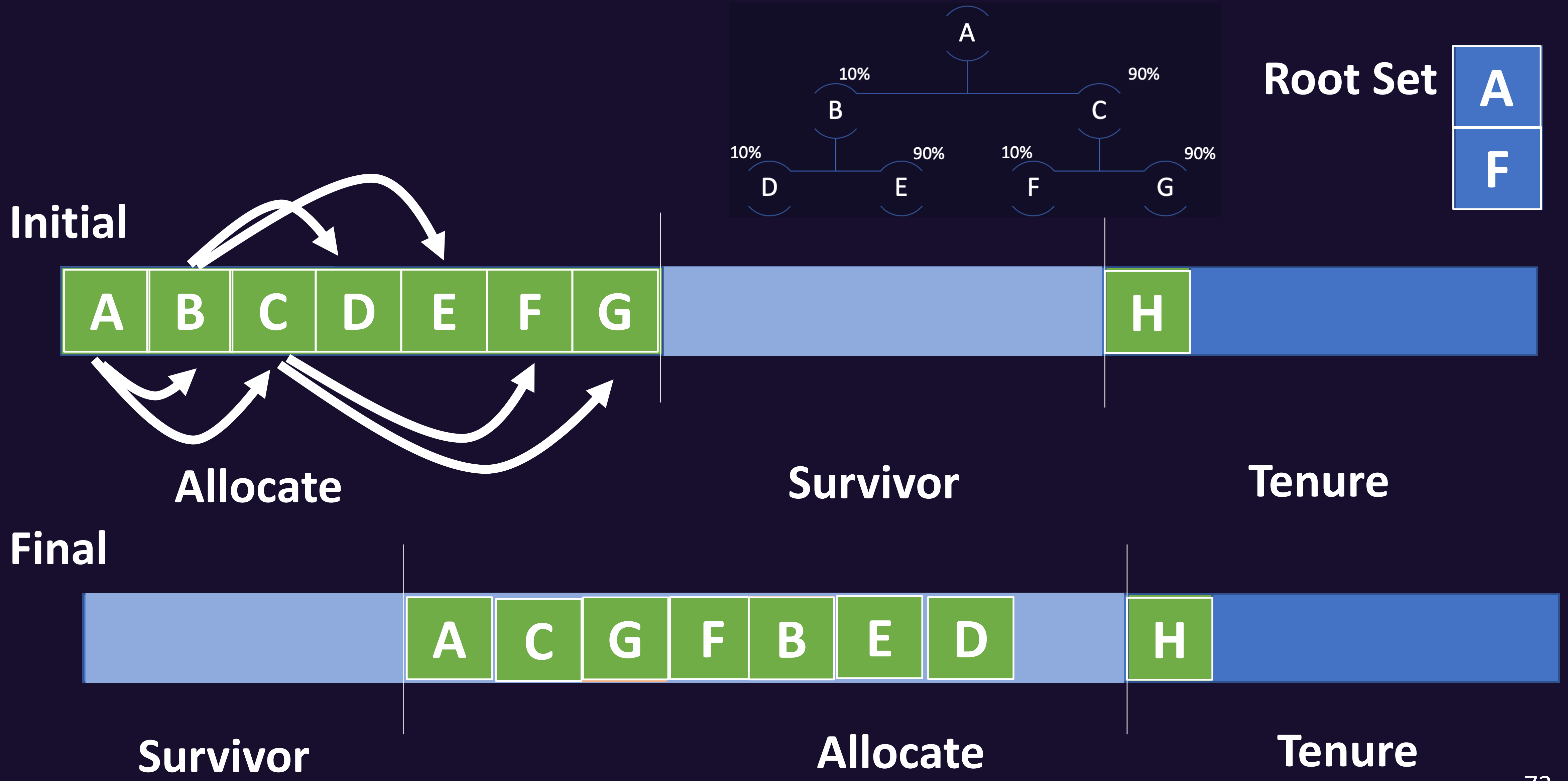
Gencon GC – Ex: **Dynamic** Breadth First



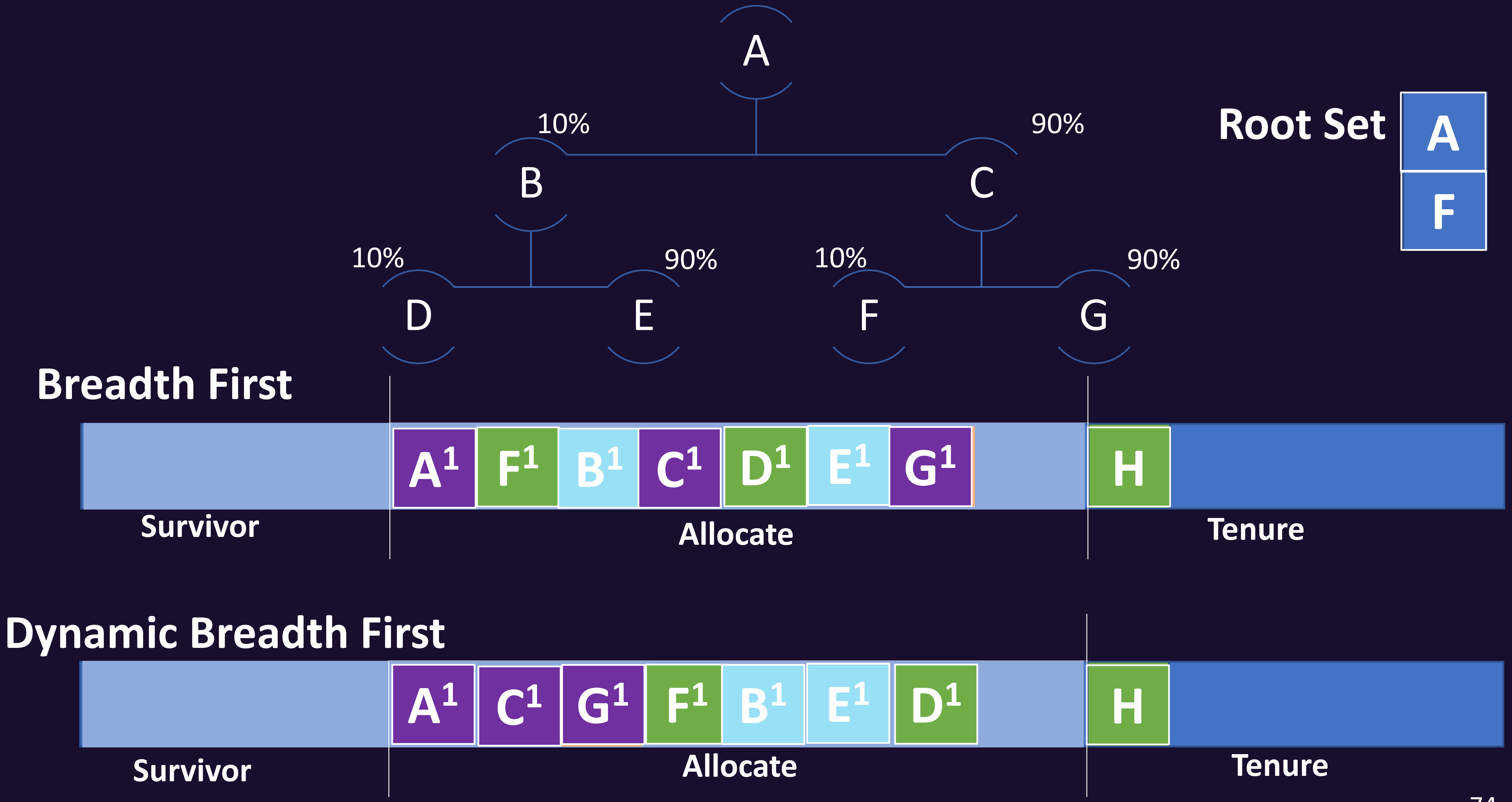
Gencon GC – Ex: **Dynamic** Breadth First



Gencon GC – Ex: Dynamic Breadth First



Breadth First vs. Dynamic Breadth First



Example Takeaways

- Dynamic Breadth First Scan Ordering enables the possibility to have objects accessed frequently spatially localized in memory
- Among other things, Dynamic Breadth First Scan Ordering will likely result in a higher cache hit ratio compared to standard Breadth First Scan Ordering

Results – Breadth First vs Dynamic Breadth First

- 2-8% throughput improvements on various benchmarks
- Negligible difference in application compile time
- 2-3% increase in average application GC pause time
- Future development iterations will be optimized to reduce GC overhead while continuing to improve application throughput efficiency

Dynamic Breadth First Summary

- Leverage existing JIT infrastructure
- Every method is divided into logical blocks where blocks are assigned a normalized hotness value between 1 – 10000
- The overall “hotness” of each field access depends on 2 key factors:
  - The block frequency of the compilation block the field has been reported in
  - The tiered compilation level that the compiler is currently compiling the method at when the field has been reported

-Xgcpolicy:balanced

Region based generational collector

Provides a significant reduction in max GC STW pause times

Introduces a write barrier to track inter region references

Incremental heap defragmentation

-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

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

Heap

-Xgcpolicy:balanced Heap

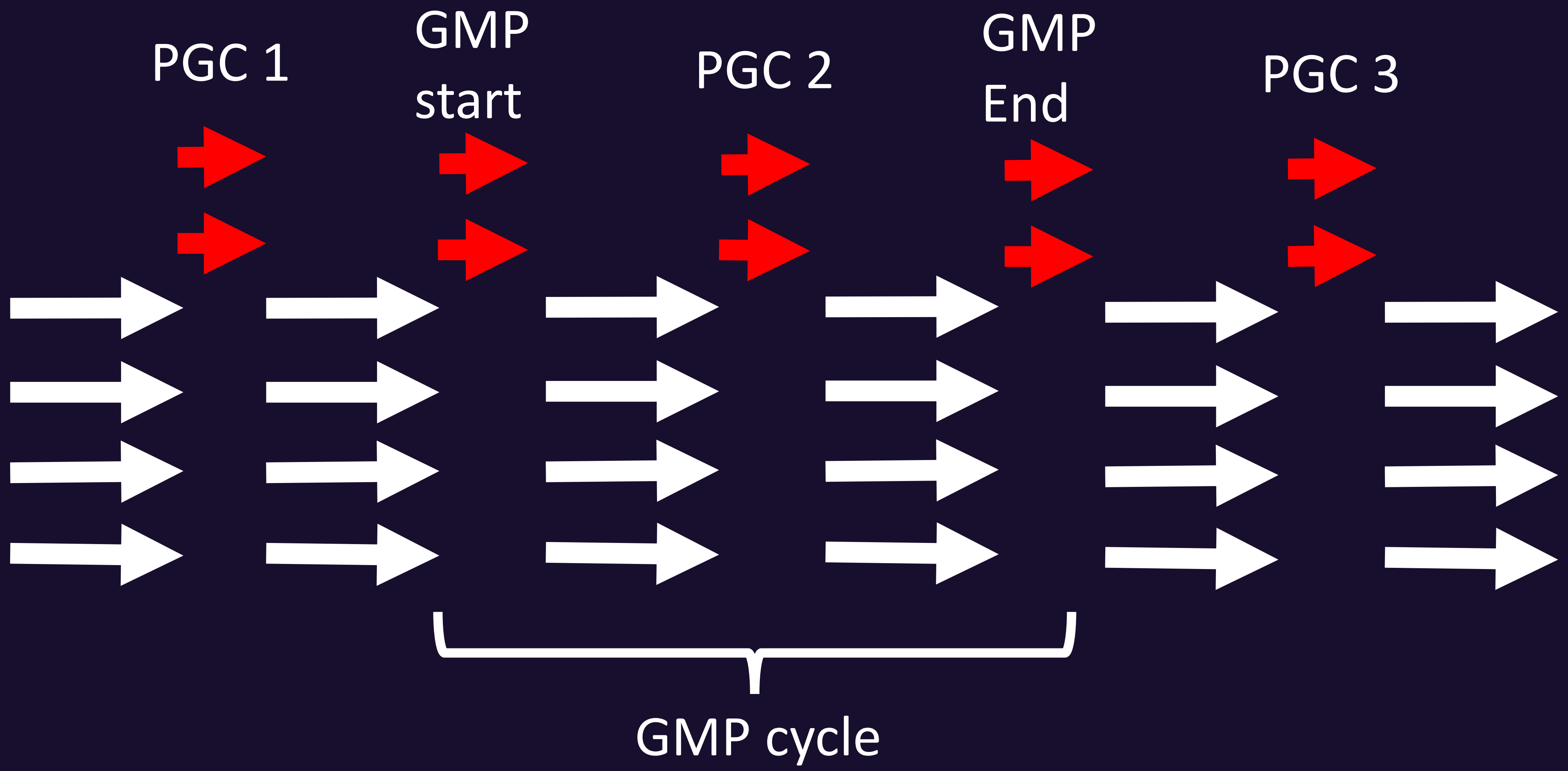
No non-array object can be larger than a region size

- If (object_size > Region_size) throw OutOfMemoryError

Large arrays are allocated as arraylets

- Arrays less than region size are allocated as normal arrays

-Xgcpolicy:balanced GC



-Xgcpolicy:balanced Global Mark Phase (GMP)

Does not reclaim any memory

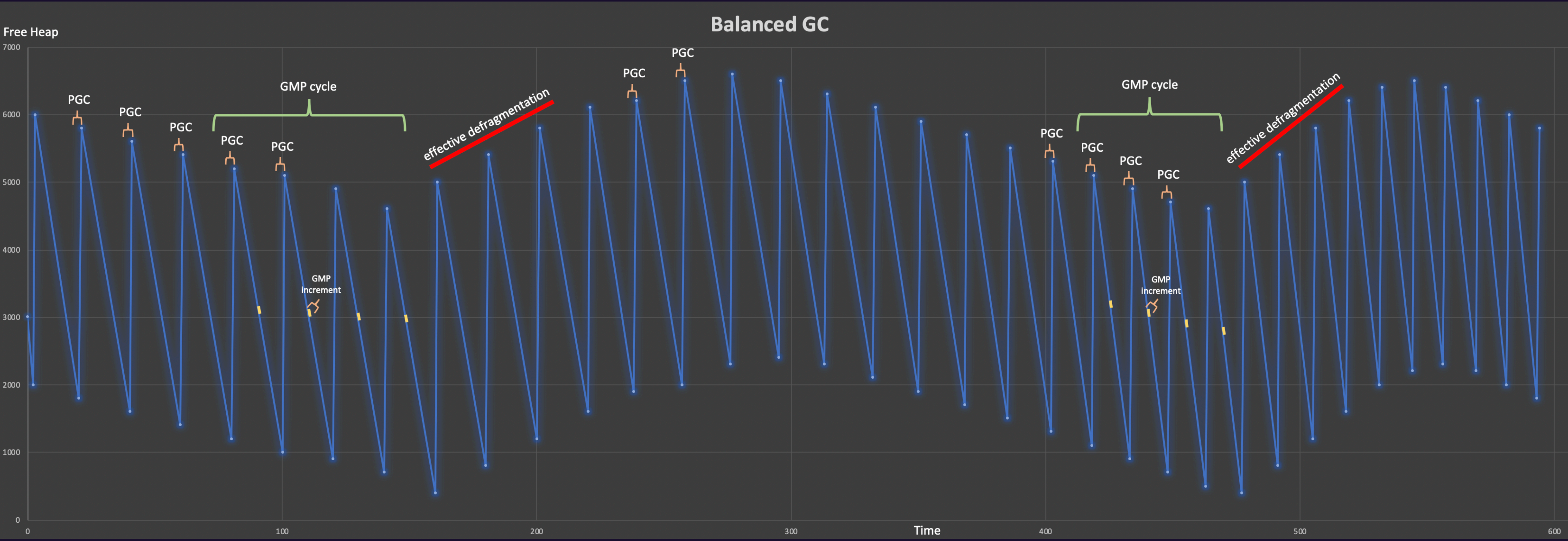
Performs a marking phase only

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

-Xgcpolicy:balanced



-Xgcpolicy:balanced

Write Barrier

Why do we need a write barrier?

Balanced PGCs can select any region to be included in the collect phase

Similar to the generational barrier, the GC needs to know which regions reference a given region

-Xgcpolicy:balanced

Write Barrier

How is the write barrier implemented?

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

-Xgcpolicy:balanced

Write Barrier

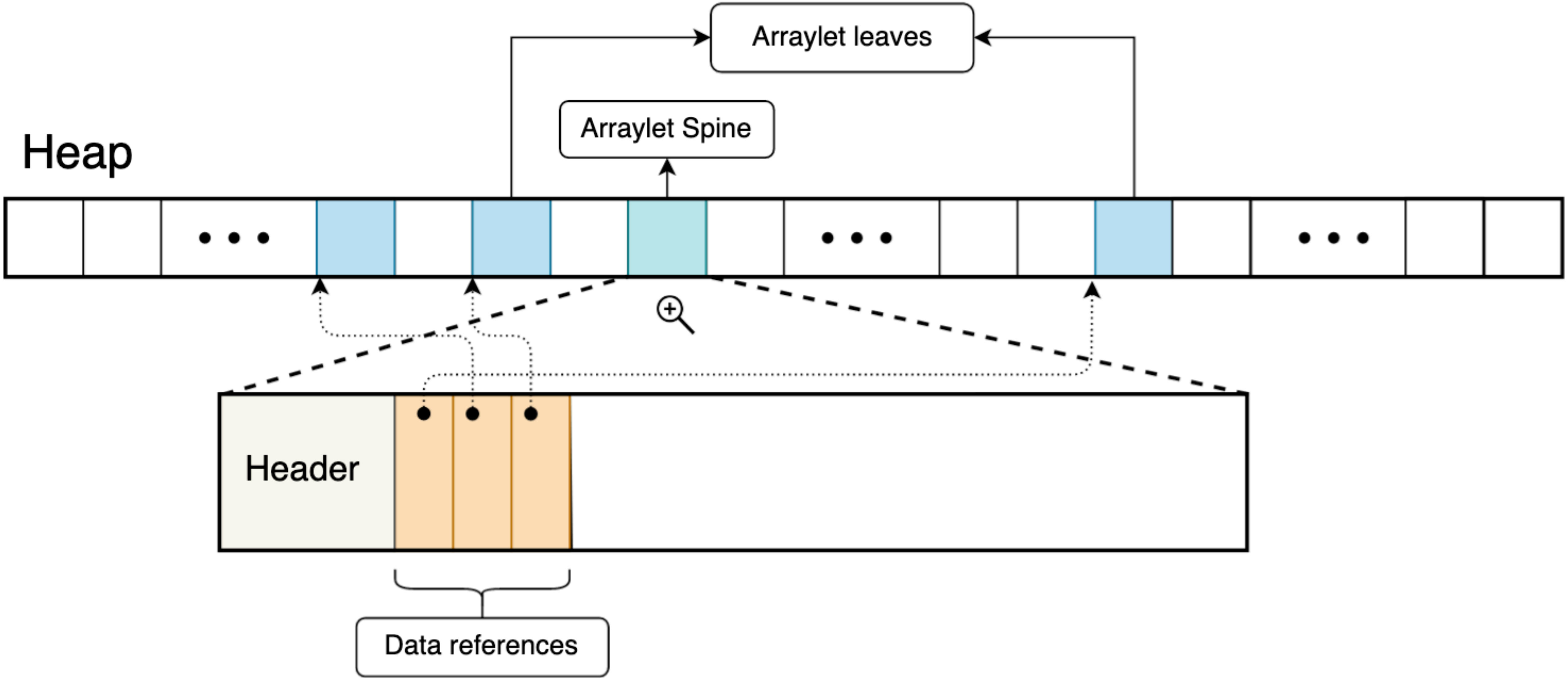
```
private void setField(Object A, Object C) {
  | A.field1 = C;
  | dirtyCard(A);
}
private void checkCards() { // Beginning of PGC
  | for(eachCard)…
  |  | if (findRegion(A) != findRegion(C)) {
  |  |  | addRSCLEntryFor(C, A);
  |  |  }
}
```

Arraylets

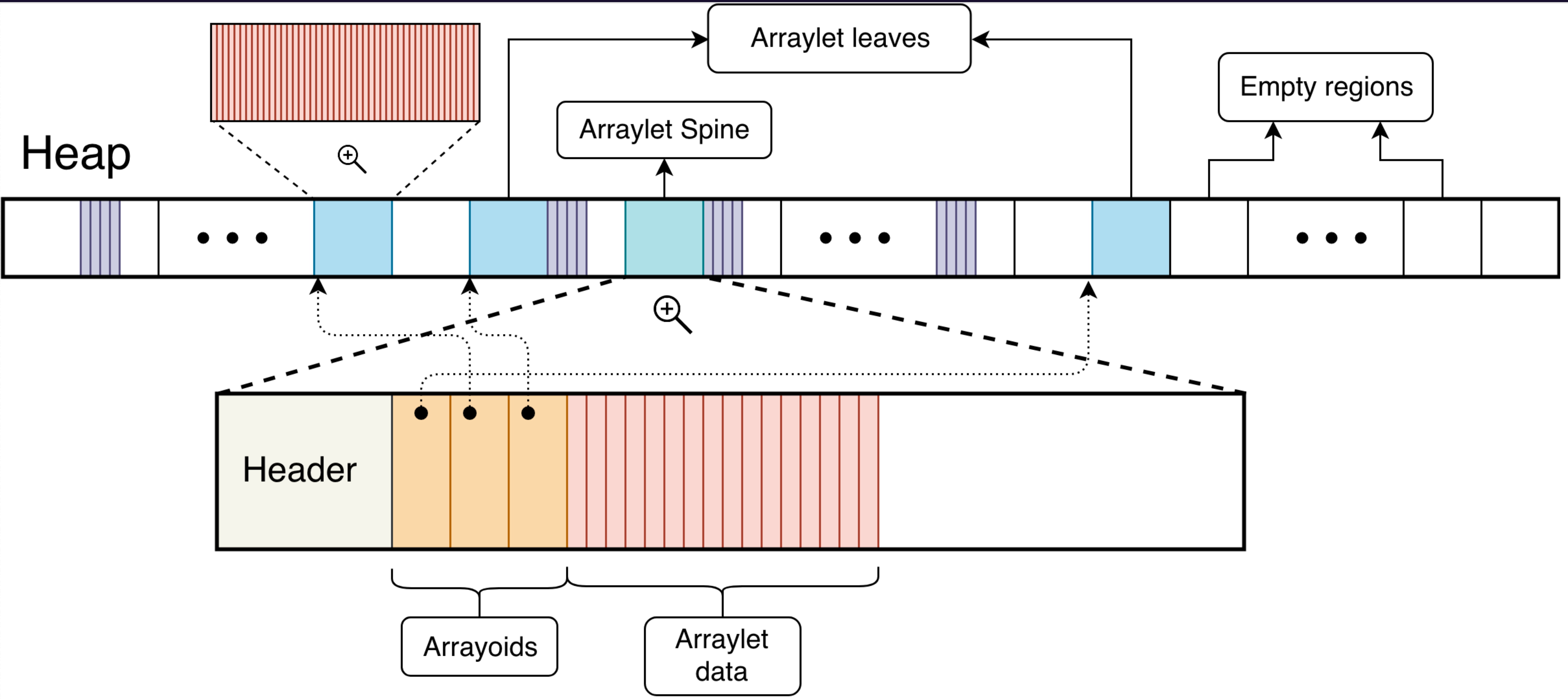
Large Arrays that cannot fit into a single region

- ❖ Array is created from construct comprising of an arraylet spine and 1 or more arraylet leaves
- ❖ An arraylet spine is allocated like a normal object
- ❖ Each leaf consumes an entire region

Arraylets


Arraylet leaves
Arraylet Spine
Heap
Header
Data references

Arraylets


Heap
Arraylet leaves
Arraylet Spine
Empty regions
Header
Arrayoids
Arraylet data

Arraylets

Arraylets were introduced so that arrays were more cleverly stored in the heap for balanced and metronome GC policies.

Some APIs require a contiguous view of an array

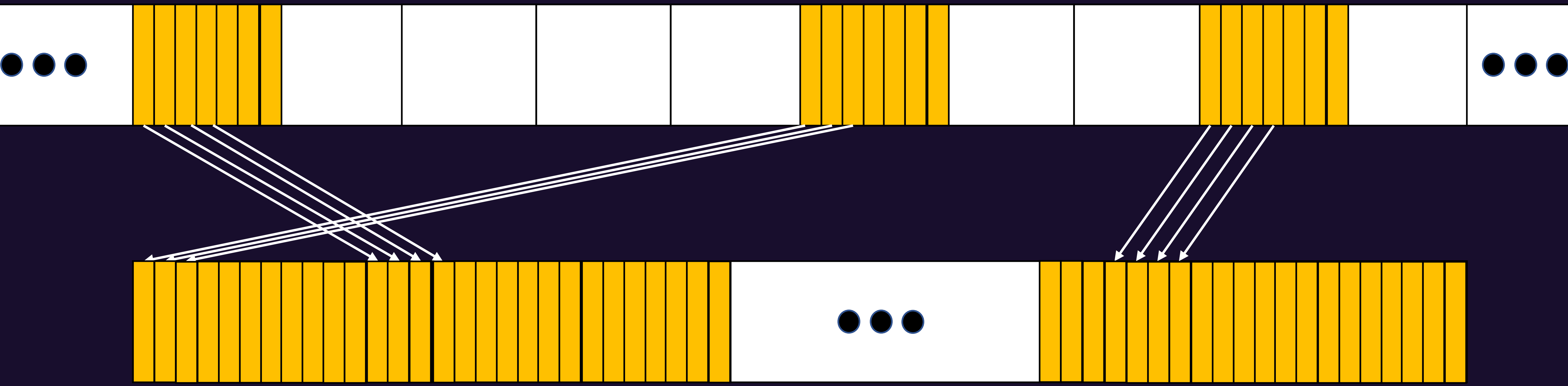
Arraylets

Some APIs require a contiguous view of an array

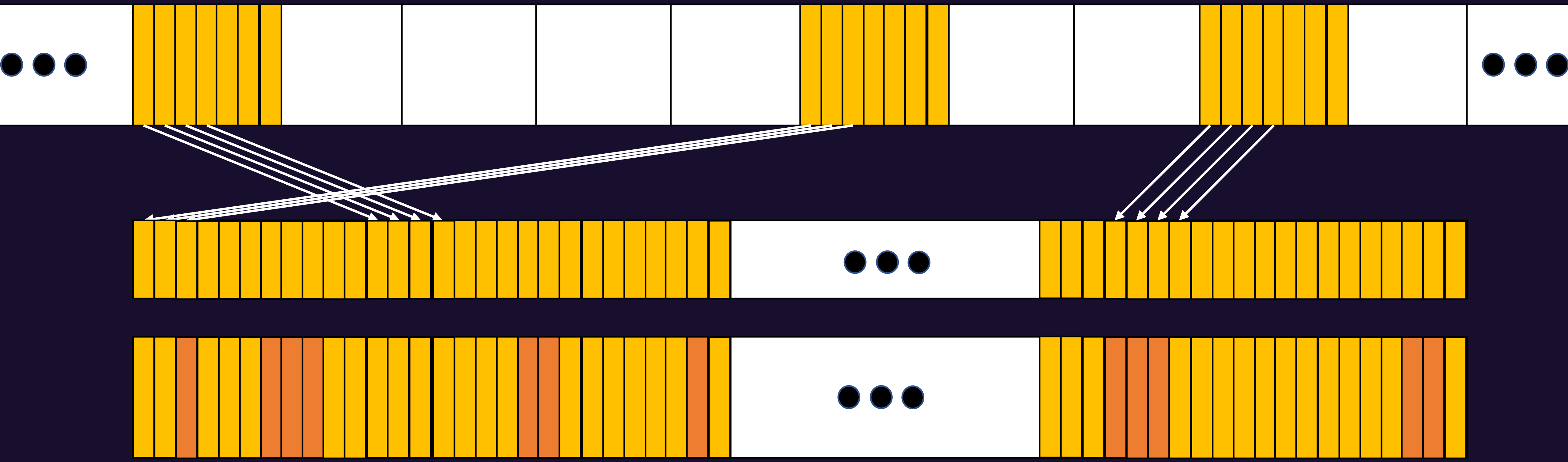
The case of Java Native Interface (JNI) Critical APIs

JNI Critical is used when the programmer wants direct addressability of the object.

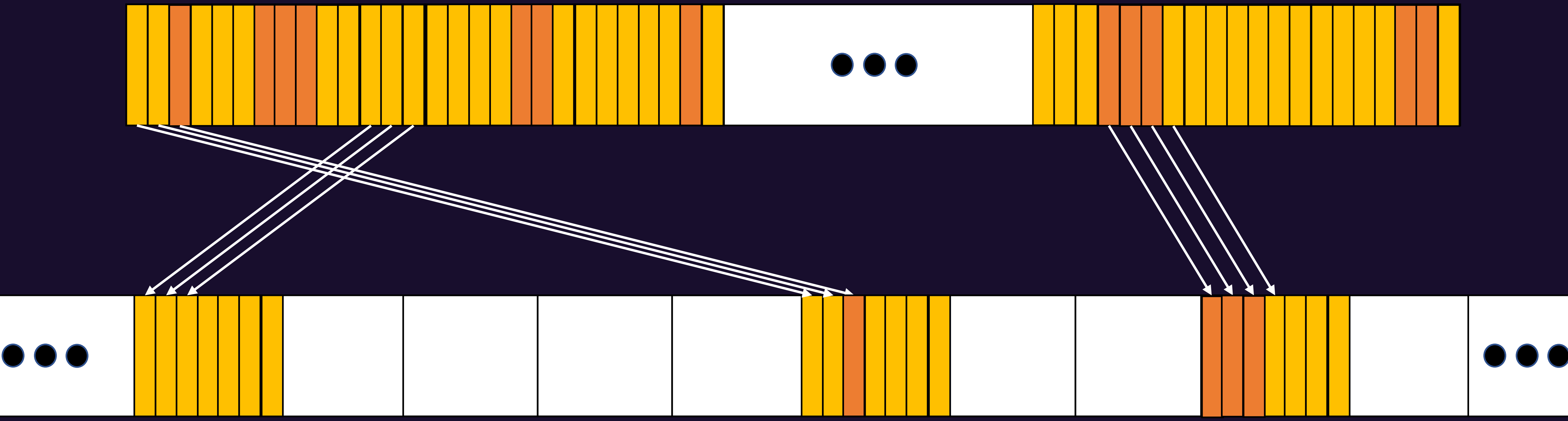
Arraylets



Arraylets



Arraylets



Arraylets

Very expensive!!

Arraylets
Double Mapping

Make large arrays (discontiguous arraylets) look contiguous

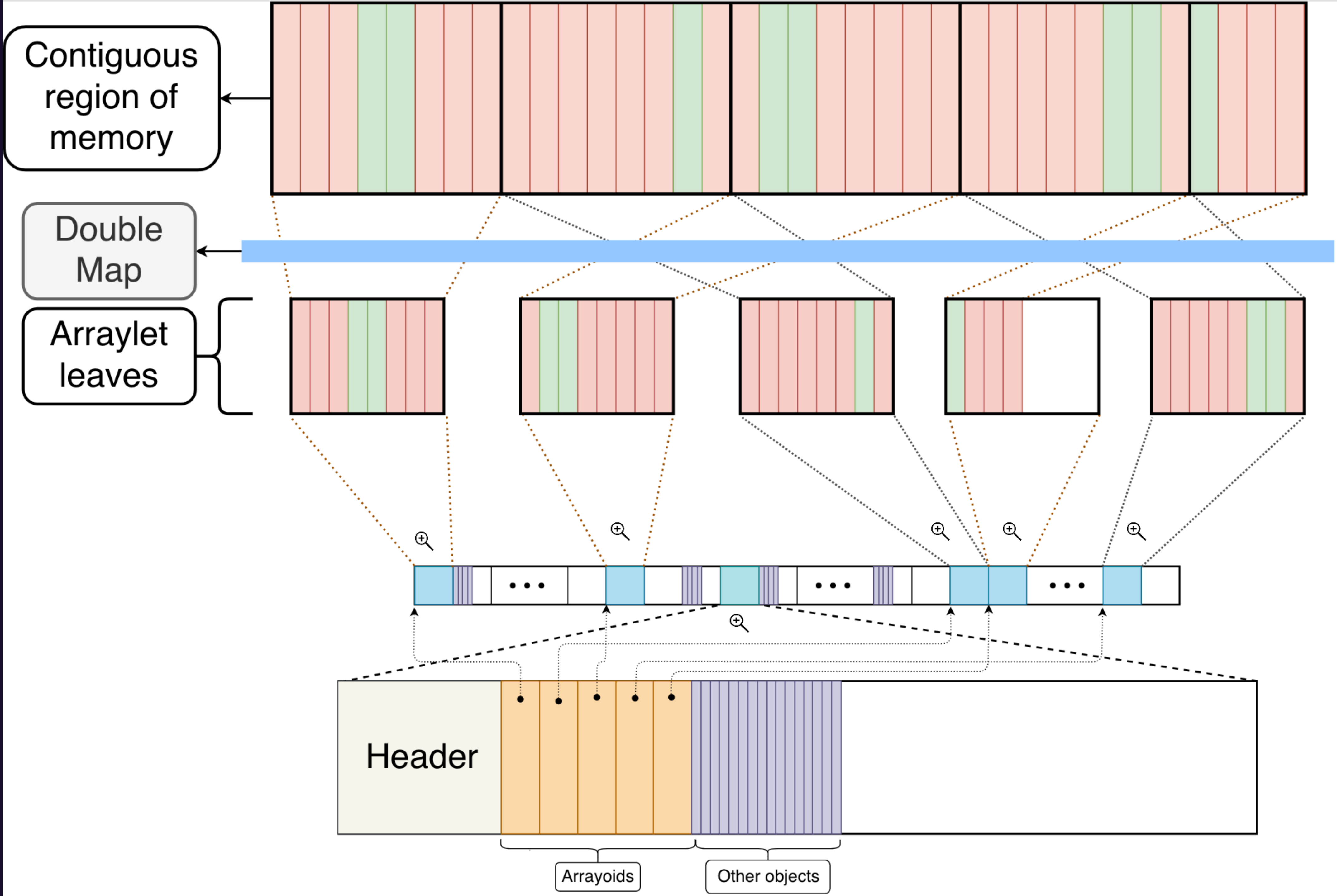
Physical memory is limited

Virtual Memory address space is large in 64 bit systems, 2^{64} in fact compared to 32 bits in 32 bit systems

Arraylets
Double Mapping

Map 2 virtual memory addresses to the same physical memory address

Any modifications to the newly mapped address will reflect the original array data, and vice-versa


Contiguous region of memory
Double Map
Arraylet leaves
Header
Arrayoids
Other objects

Arraylets
Double Mapping

Comparing JNI critical operations, array operations received **30x boost** in speedup

Can We do better?

Double Mapping Arraylets are only available on newer version of Linux

Off-heap management for large objects

Double Mapping Drawbacks

Doable with `shm_open(3)` but:

- It returns a file descriptor (backed by shared memory)
- Linux systems have cap on max sshm_openhared memory

Doable with `memfd_create(2)` but:

- It also returns a file descriptor
- Behaves like regular file backed by RAM
- Only available on newer GLIBC versions

Off-heap Management for Large Objects

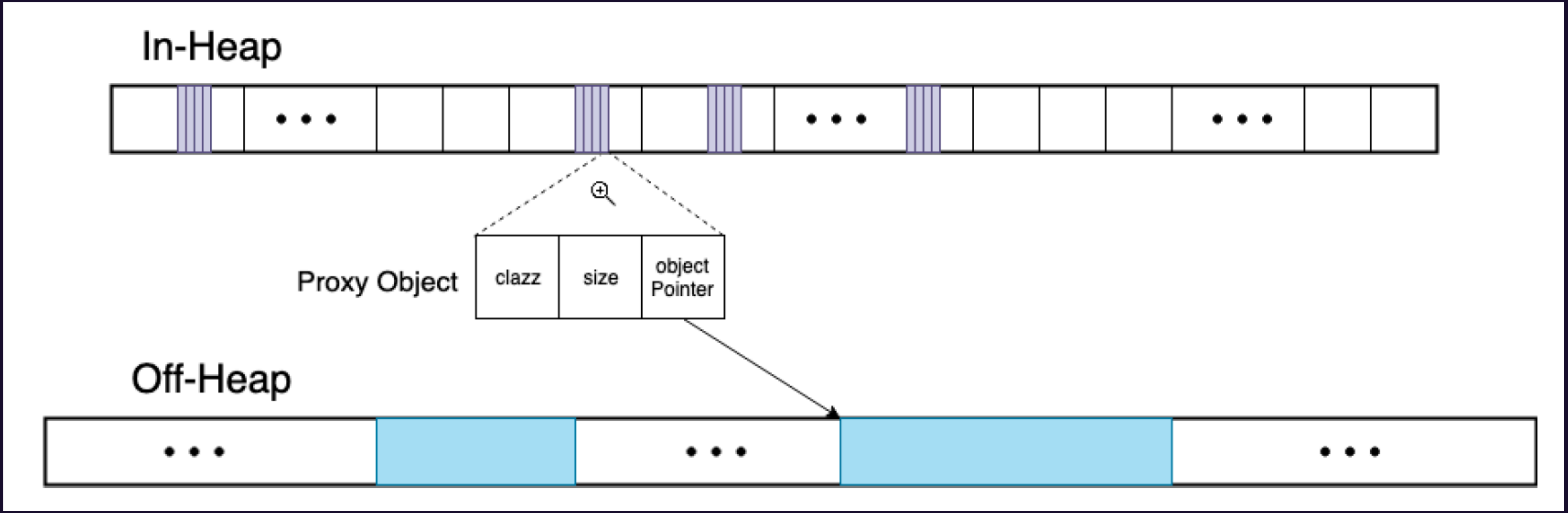
Does not require file descriptors

It also takes advantage of vast virtual memory space

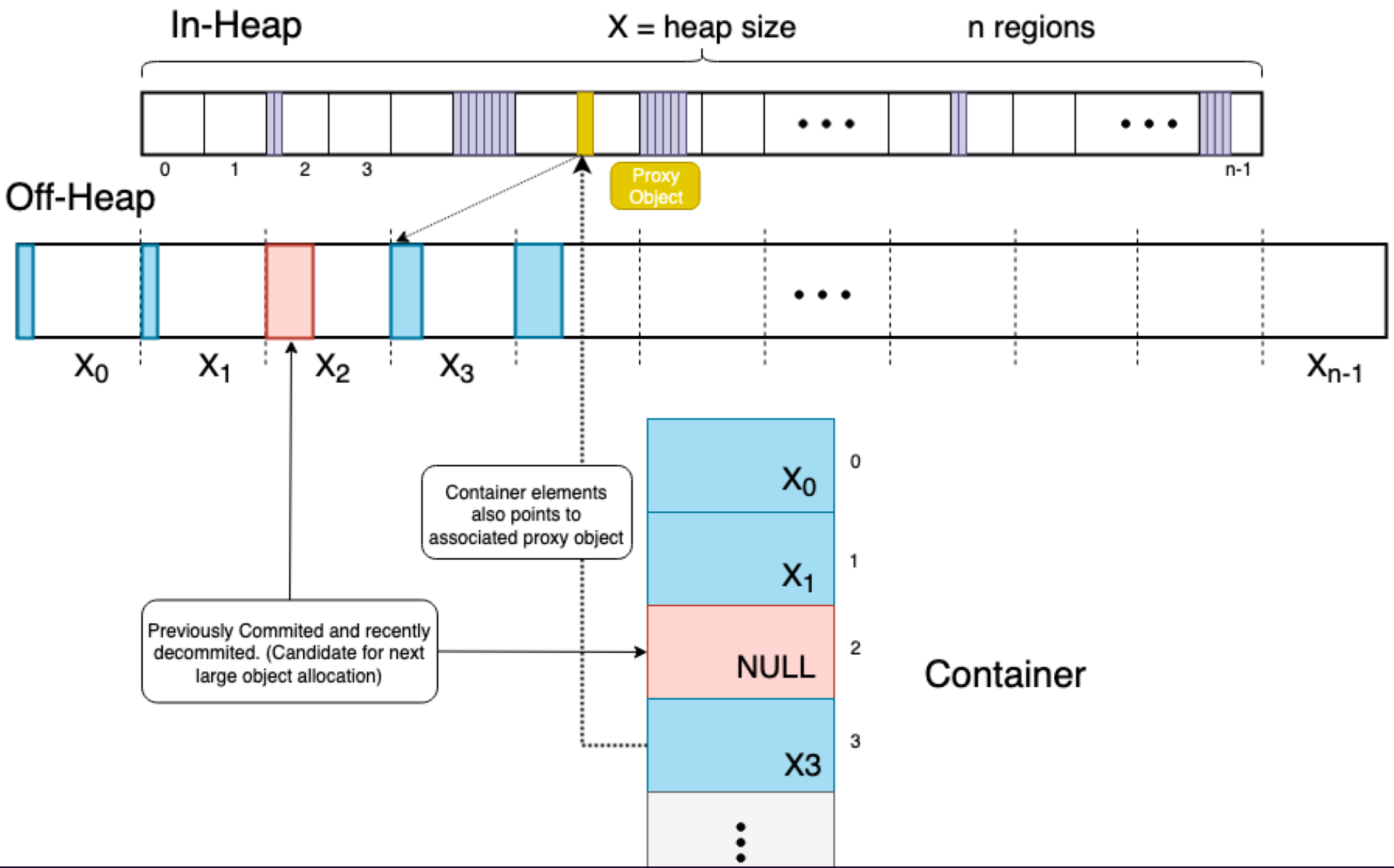
Will only be available in 64bit systems

OpenJ9

Off-heap Management



IBM


In-Heap
X = heap size
n regions
0
1
2
3
n-1
Proxy Object
Off-Heap
X_0
X_1
X_2
X_3
X_{n-1}
Container elements also points to associated proxy object
Previously Commited and recently decommited. (Candidate for next large object allocation)
X_0
X_1
NULL
X3
Container

Off-heap Management

What's the smallest off-heap that we can come up with so that we we'll never have to **compact** it?

Off-heap Management

The smallest object that we'll be storing at off-heap is as big as 2 regions

If we're greedy

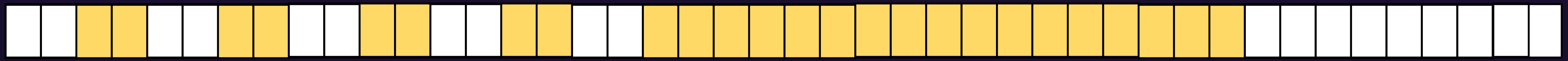
```
off_heap_size = in_heap_size * region_count
off_heap_size = 2TB * 1024 // == 2PB == 2^51 B
```

Off-heap Management

What's the worst possible allocation pattern we can get?

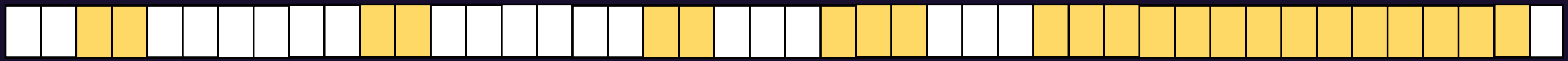
Off-heap Management

What's the worst possible allocation pattern we can get?

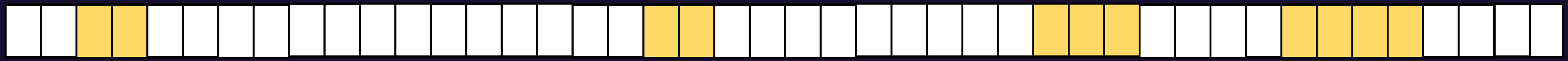


Free half of objects with a pattern of every other object

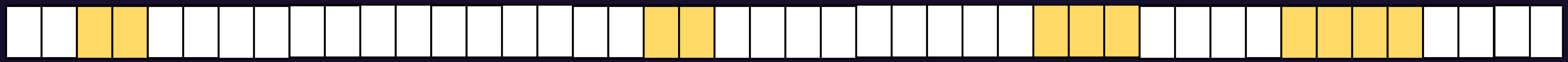
Allocate objects of region size 7



Free half of objects with a pattern of every other object



Off-heap Management



There's a pattern!
Now we can calculate off-heap size with a better upper bound

Off-heap Management

If we're smart

```
off_heap_size = ceil(log2(region_count) * in_heap_size / 2
off_heap_size = ceil(log2(1024) * 2TB / 2 // == 20TB ~ 2^44 B
```

Before

```
off_heap_size = 2TB * 1024 // == 2PB == 2^51 B
```

Off-heap Management

Positives

- Any platform that supports virtual memory can benefit
- Unburdens in-heap from large object allocation
- Off-heap will never need to be compacted
- Does not require file descriptors

Negatives

- Whenever we commit memory at off-heap we must decommit memory at in-heap, and vice-versa
- One extra level of indirection to access array data

GC Policies


Gencon
Balanced
Old/new space | generational | copy collector
Region based | generational | copy collector
CMS
G1 GC
Metronome
Gencon CS
Region based
Read barrier
ZGC
Shenandoah
Azul C4
Common GCs Throughput centric
Pauseless GCs
Pauseless GC with special hardware

Summary

$$\text{throughput} = \frac{GC\ Pause}{x}$$

Perfect STW GC **vs** Perfect Pauseless GC

Higher Throughput

Lower Throughput

Longer pauses

Shorter pauses

Dynamic Breadth First Scan Ordering

Double Mapping

Off-heap Object Management

Links

Eclipse OpenJ9

https://www.eclipse.org/openj9

https://www.eclipse.org/openj9/docs/cmdline_migration

AdoptOpenJDK

https://adoptopenjdk.net

Eclipse OMR

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

JPoint 2021

igorbraga

@igor_h_braga

Questions?

oommen-j

References

R. Jones et al. “The Garbage Collection Handbook”. Chapman & Hall/CRC, 2012