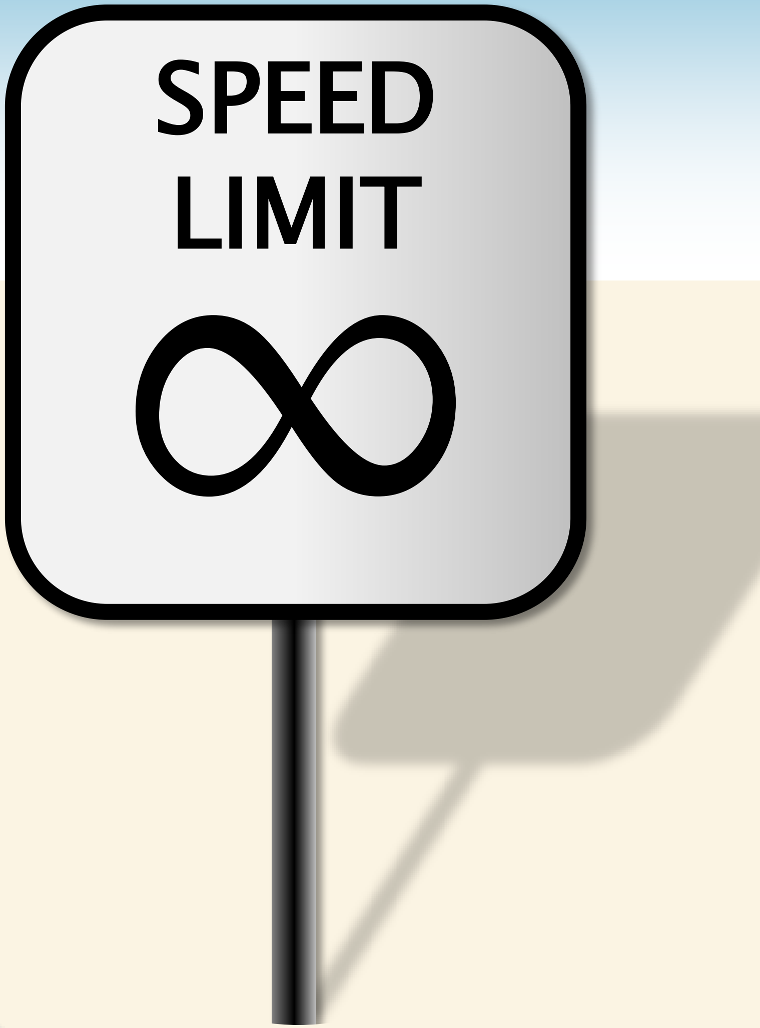
Performance
Engineering of
Software Systems



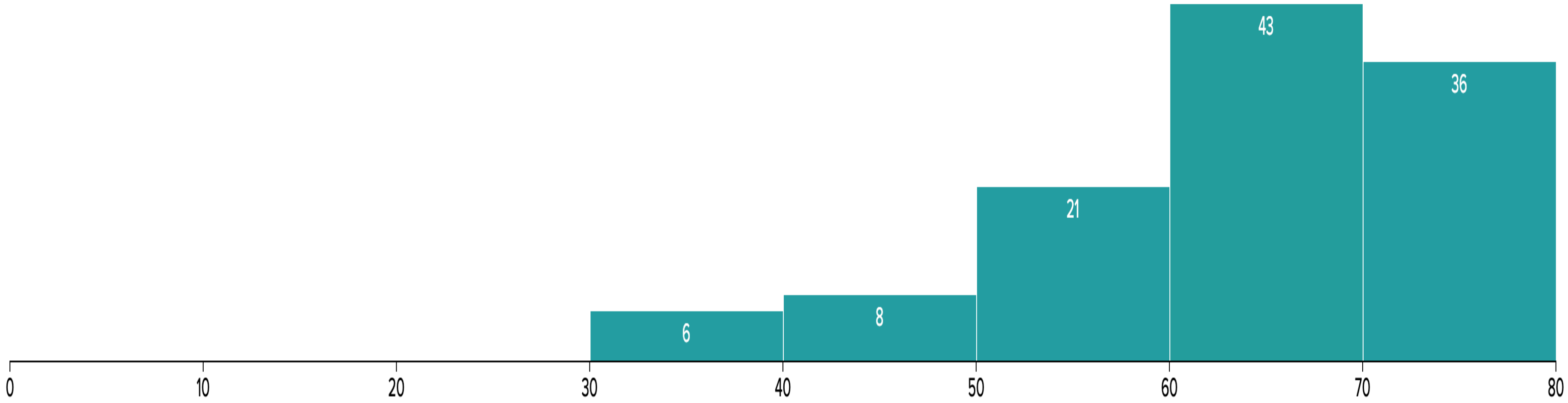
LECTURE 12

Storage Allocation

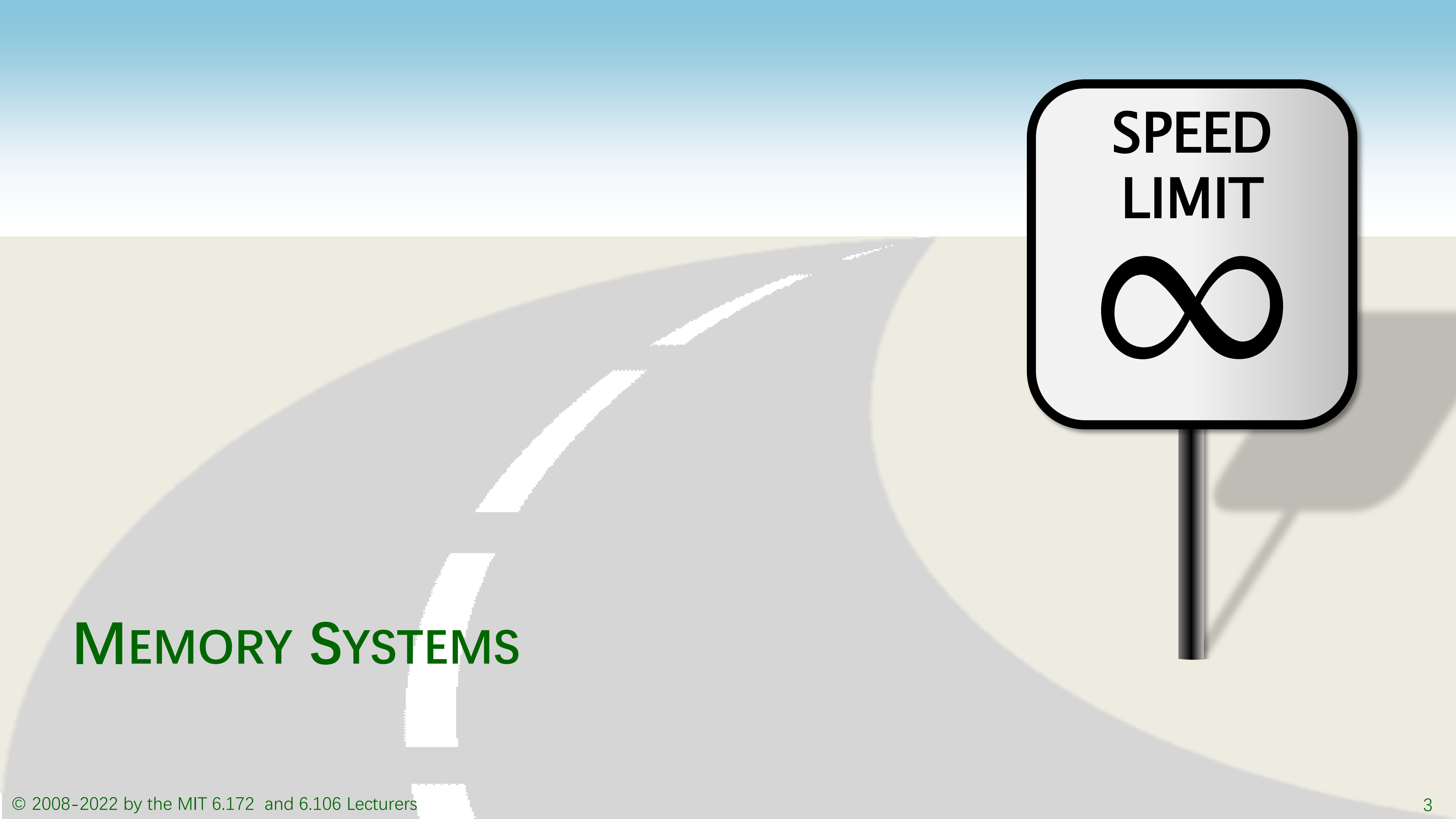
Saman Amarasinghe

October 25, 2022

Quiz 1



MEDIAN	MAXIMUM	MEAN	STD DEV
65.25	80.0	62.91	11.08



MEMORY SYSTEMS

The Memory System

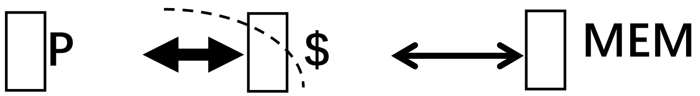
The Principle of Locality:

- Program access a relatively small portion of the address space at any instant of time.

Two Different Types of Locality:

- Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
- Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight-line code, array access)

Last 30 years, HW relied on locality for memory performance


P $ MEM

Levels of the Memory Hierarchy

Capacity
Access Time
Cost

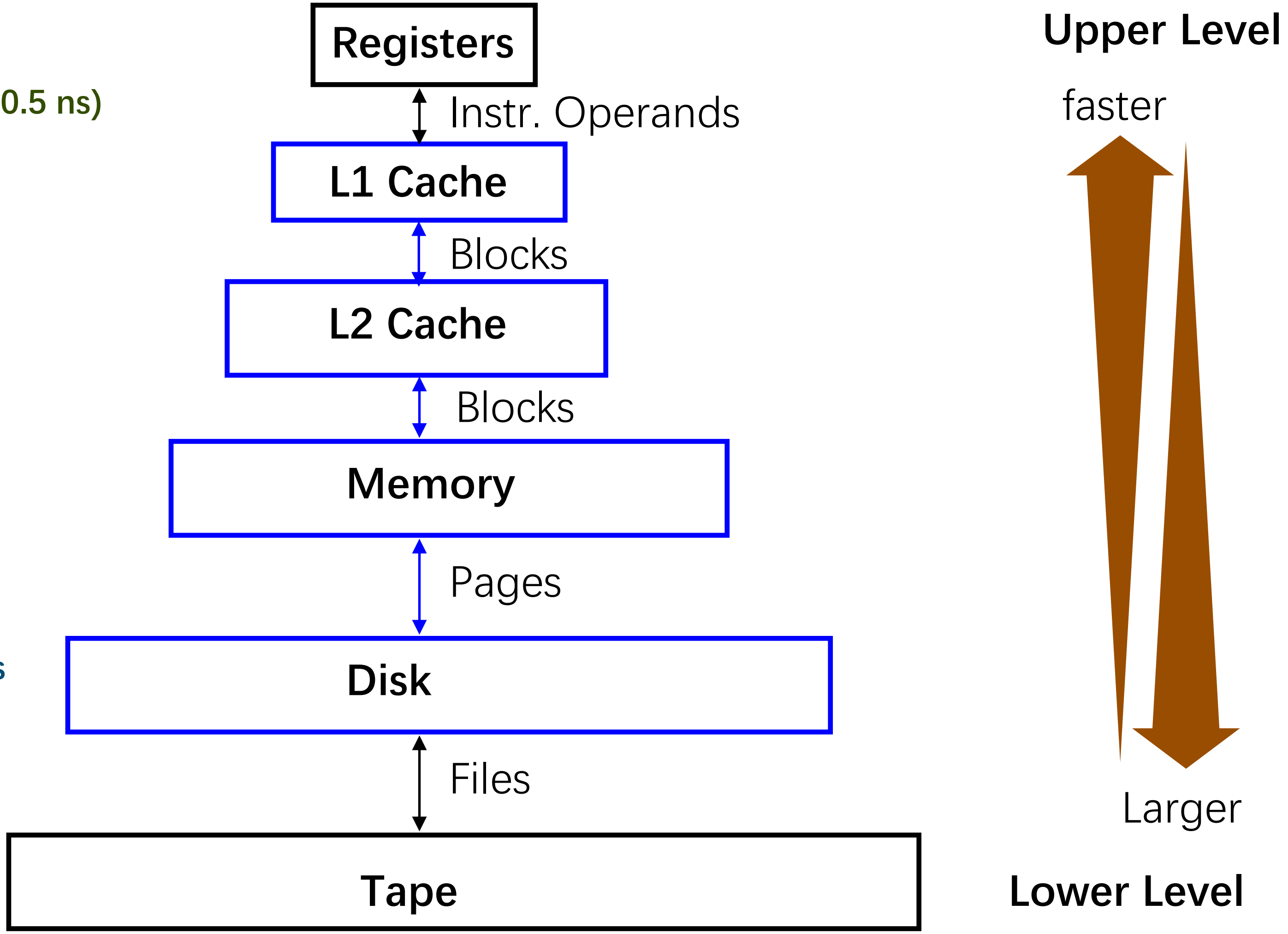
CPU Registers
100s Bytes
300 – 500 ps (0.3-0.5 ns)

L1 and L2 Cache
10s-100s K Bytes
~1 ns - ~10 ns
$1000s/ GByte

Main Memory
G Bytes
80ns- 200ns
~ $100/ GByte

Disk
10s T Bytes, 10 ms
(10,000,000 ns)
~ $1 / GByte

Tape
infinite
sec-min
~$1 / GByte



Cache Issues

Cold Miss

- The first time the data is available
- Prefetching may be able to reduce the cost

Capacity Miss

- The previous access has been evicted because too much data touched in between
- “Working Set” too large
- Reorganize the data access so reuse occurs before getting evicted.
- Prefetch otherwise

Conflict Miss

- Multiple data items mapped to the same location. Evicted even before cache is full
- Rearrange data and/or pad arrays
- Associativity helps

True Sharing Miss

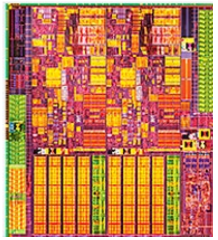
- Thread in another processor wanted the data, it got moved to the other cache
- Minimize sharing/locks

False Sharing Miss

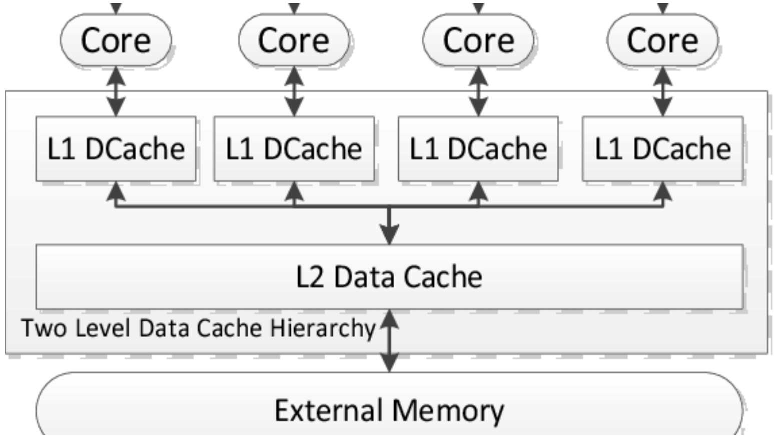
- Other processor used different data in the same cache line. So the line got moved
- Pad data and make sure structures such as locks don’t get into the same cache line

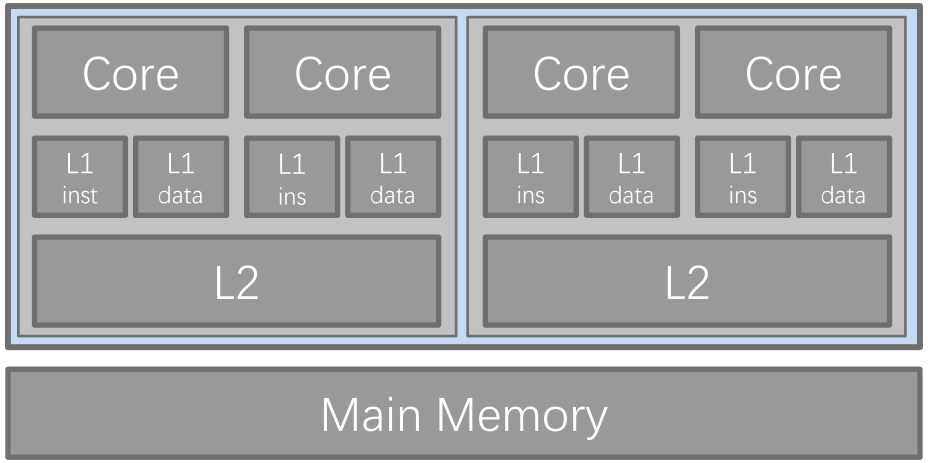
Memory Sub-system

Intel Core 2 Quad Processor



2006





L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L2 Cache			
Size	Line Size	Latency	Associativty
6 MB	64 bytes	14 cycles	24-way

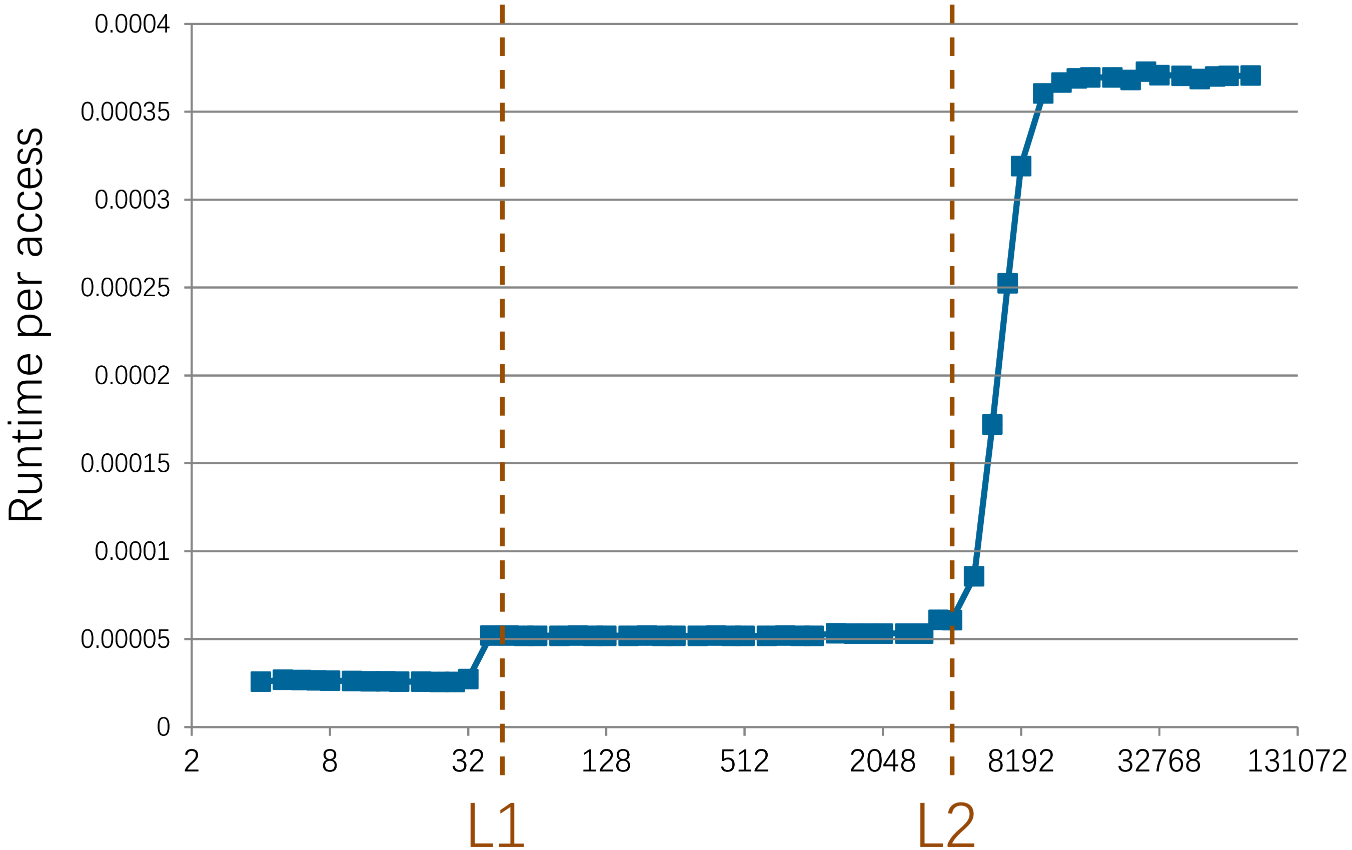
Intel Core 2 Quad Processor

```
for(rep=0; rep < REP; rep++)
        for(a=0; a <  N ; a++)
                A[a] = A[a] + 1;
```

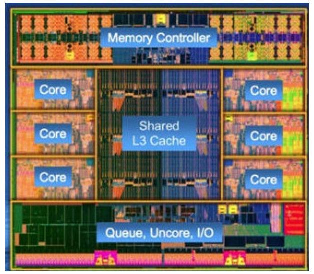
Intel Core 2 Quad Processor

```
for(rep=0; rep < REP; rep++)
        for(a=0; a <  N ; a++)
                A[a] = A[a] + 1;
```

Capacity misses if larger than the cache at each level

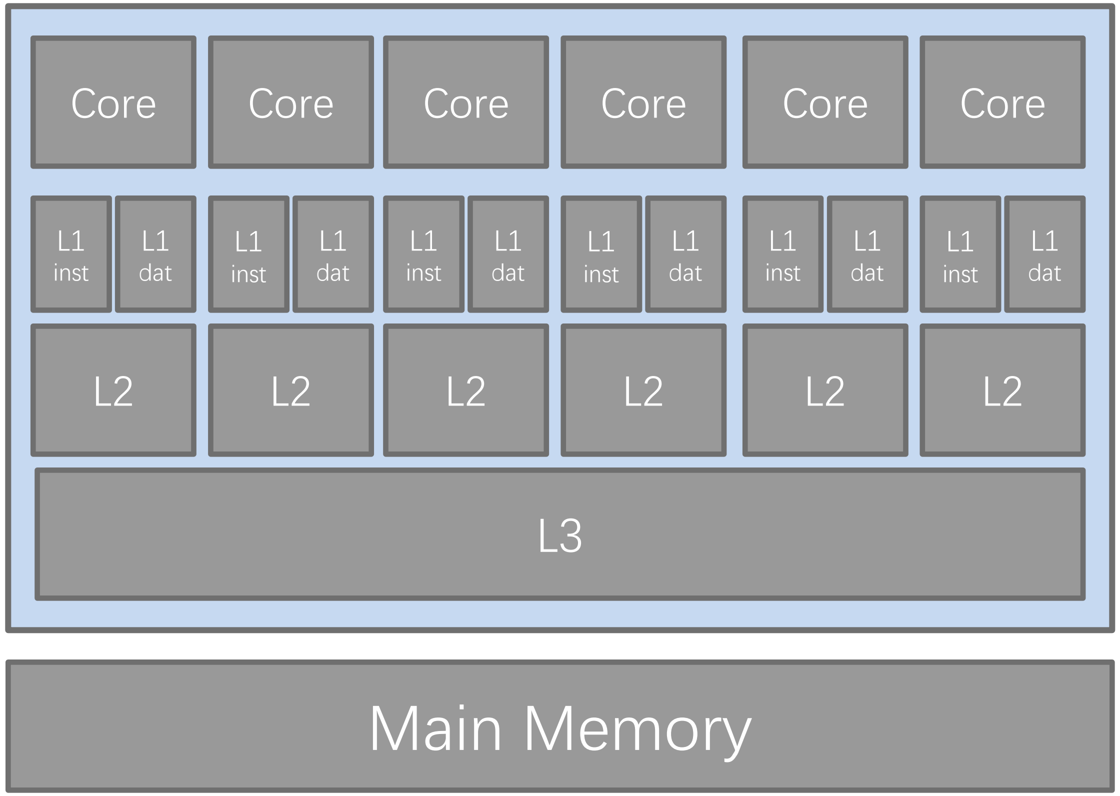


Intel® Nehalem™ Microarchitecture – Mem. Sub-system



2008

Intel 6 Core Processor


Core Core Core Core Core Core

L1 inst L1 dat L1 inst L1 dat L1 inst L1 dat L1 inst L1 dat L1 inst L1 dat L1 inst L1 dat

L2 L2 L2 L2 L2 L2

L3

Main Memory

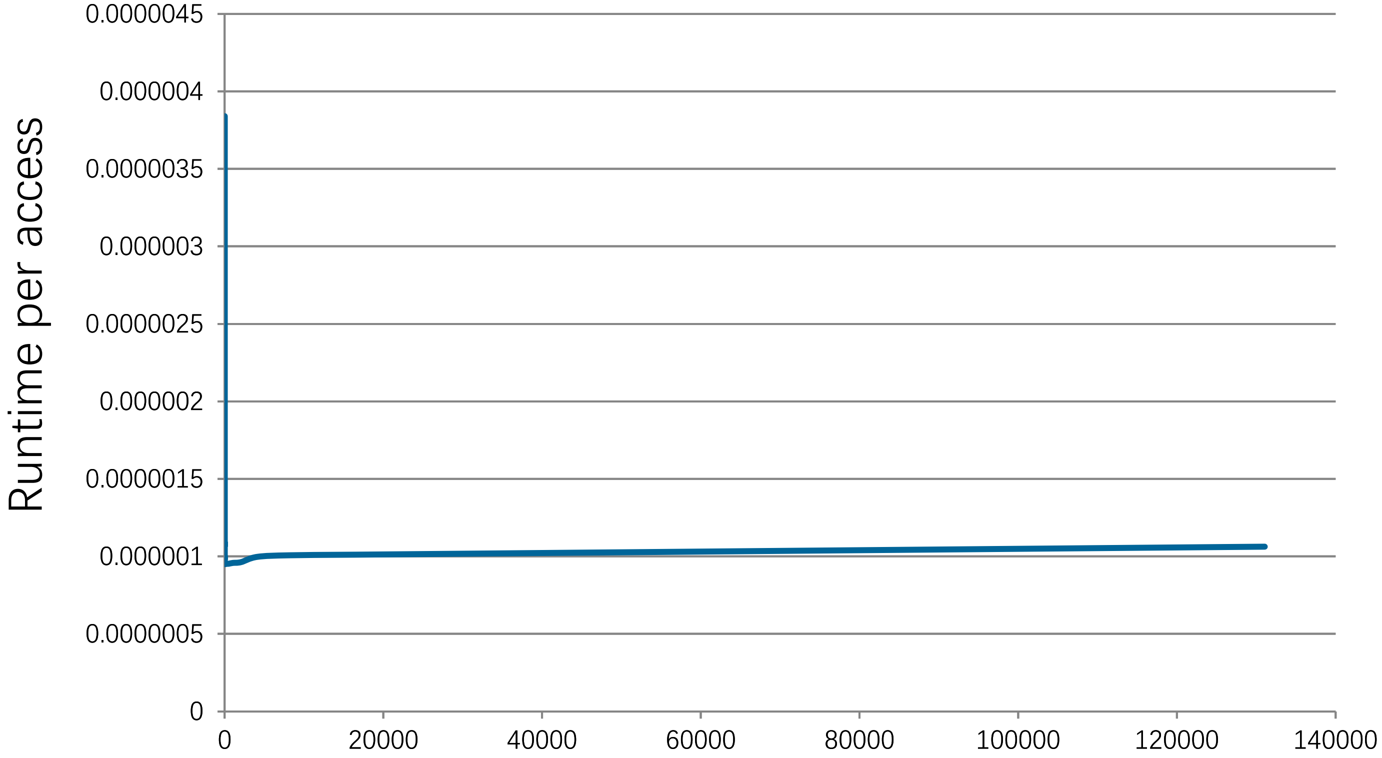
L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	4 ns	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	4 ns	4-way
L2 Cache			
Size	Line Size	Latency	Associativty
128 KB	64 bytes	10 ns	8-way
L3 Cache			
Size	Line Size	Latency	Associativty
8 MB	64 bytes	50 ns	16-way
Main Memory			
Size	Line Size	Latency	Associativty
	64 bytes	75 ns	

Intel® Nehalem™ Processor

```
for(rep=0; rep < REP; rep++)
        for(a=0; a <  N ; a++)
                A[a] = A[a] + 1;
```

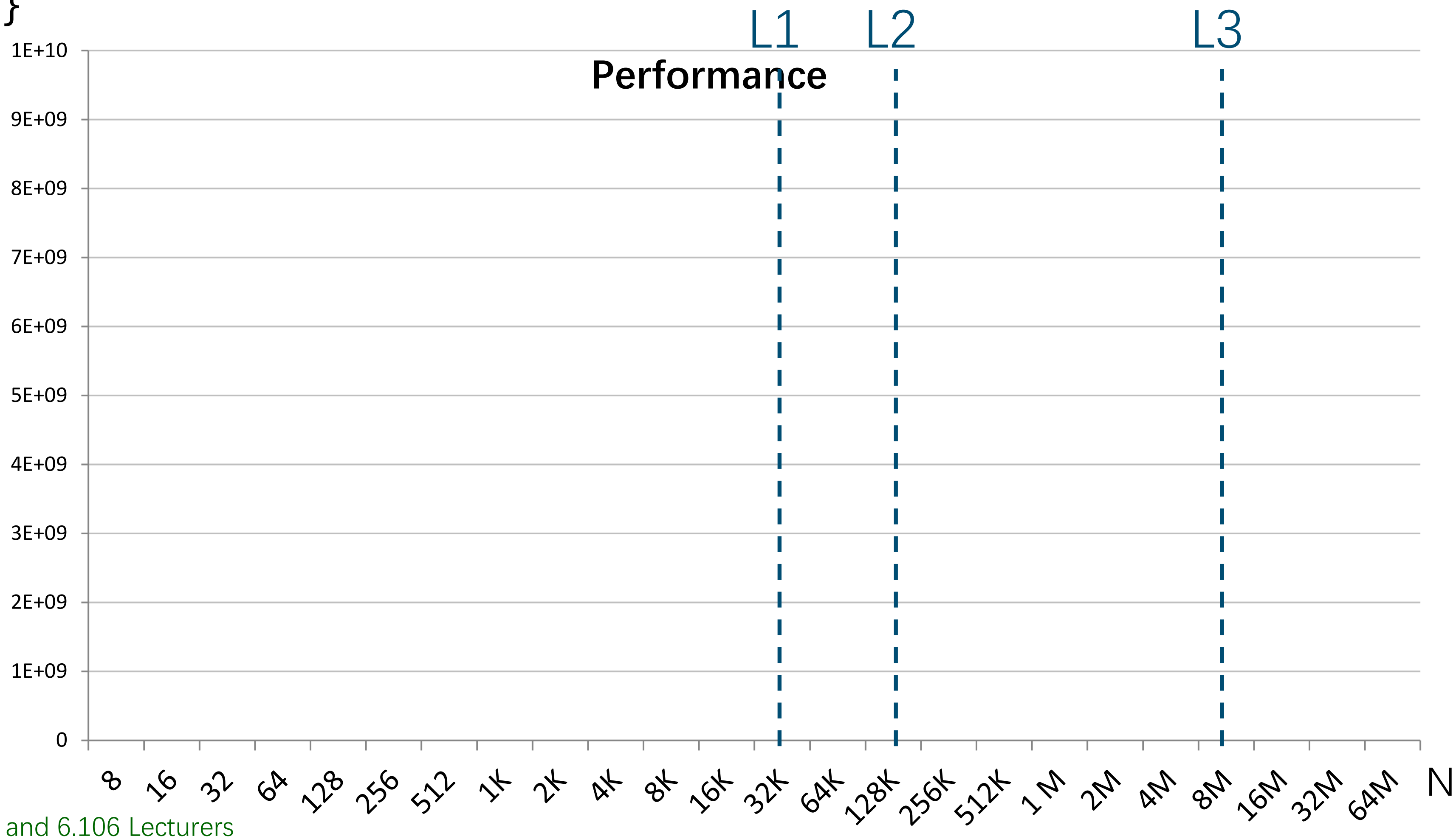
Amazing prefetcher

Single core cannot saturate the memory system



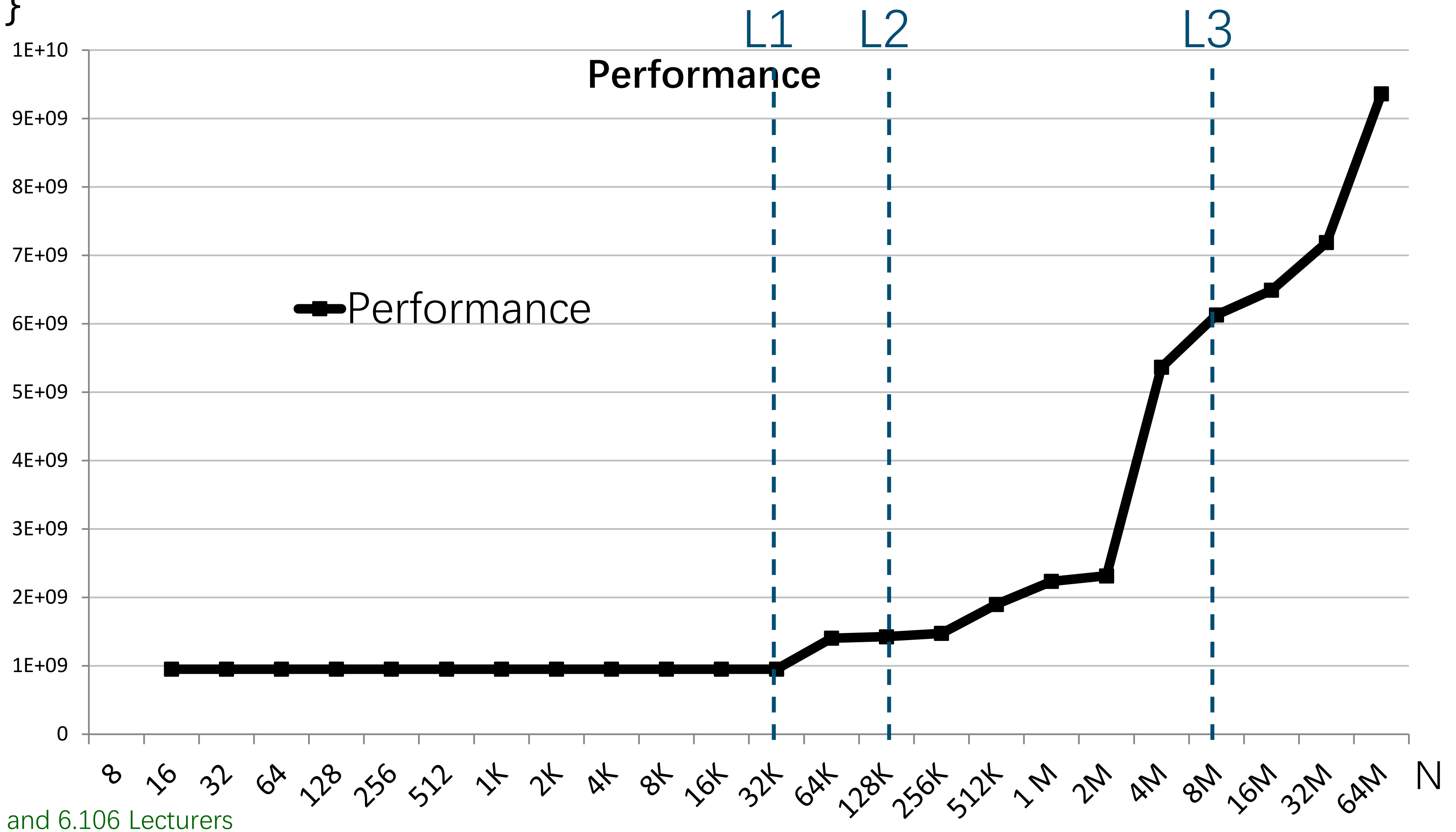
Intel® Nehalem™ Processor

```
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
      addr = ((rep + 523)*253573) & mask;
      A[addr] = A[addr] + 1;
}
```



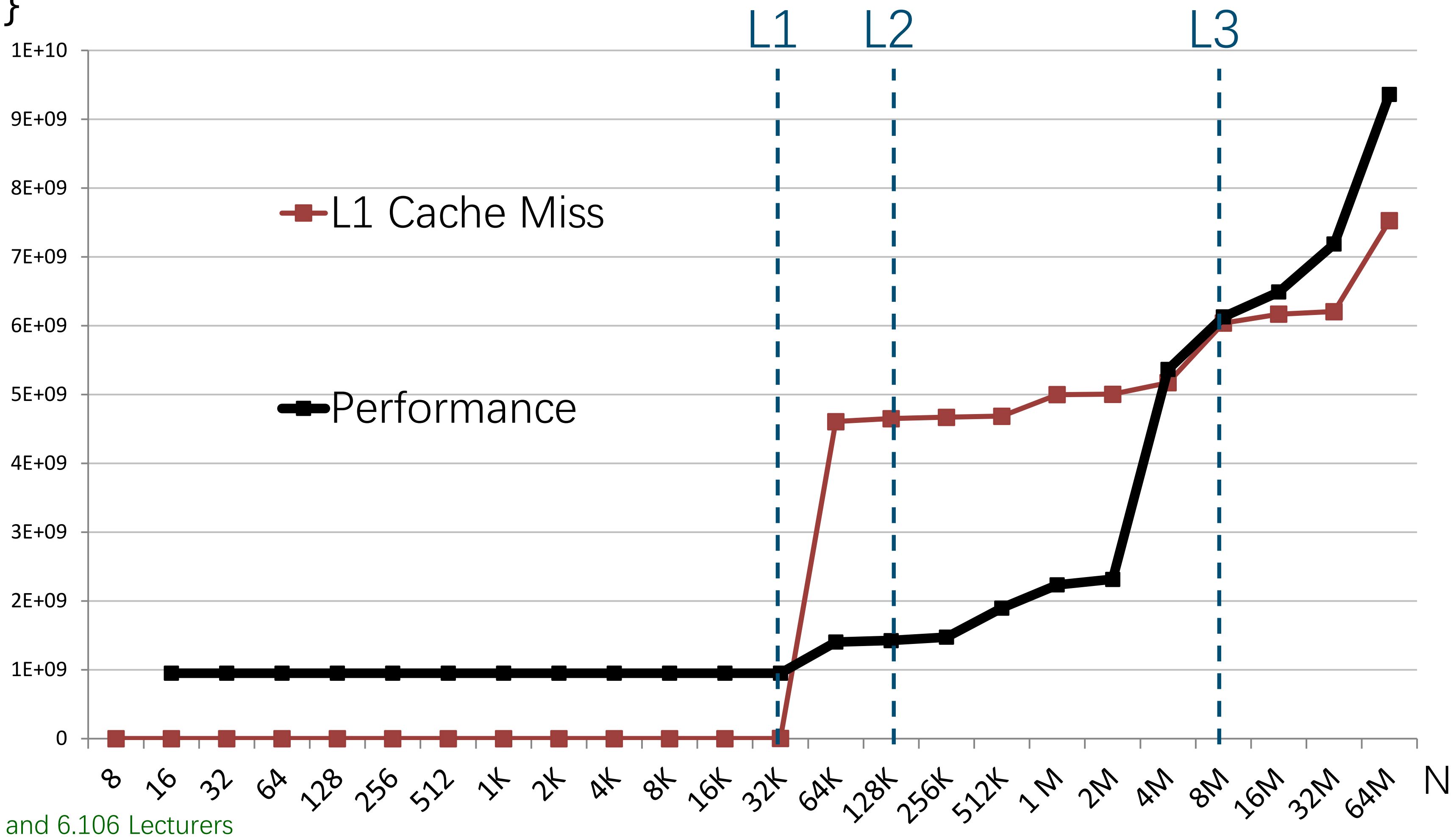
Intel® Nehalem™ Processor

```
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
      addr = ((rep + 523)*253573) & mask;
      A[addr] = A[addr] + 1;
}
```



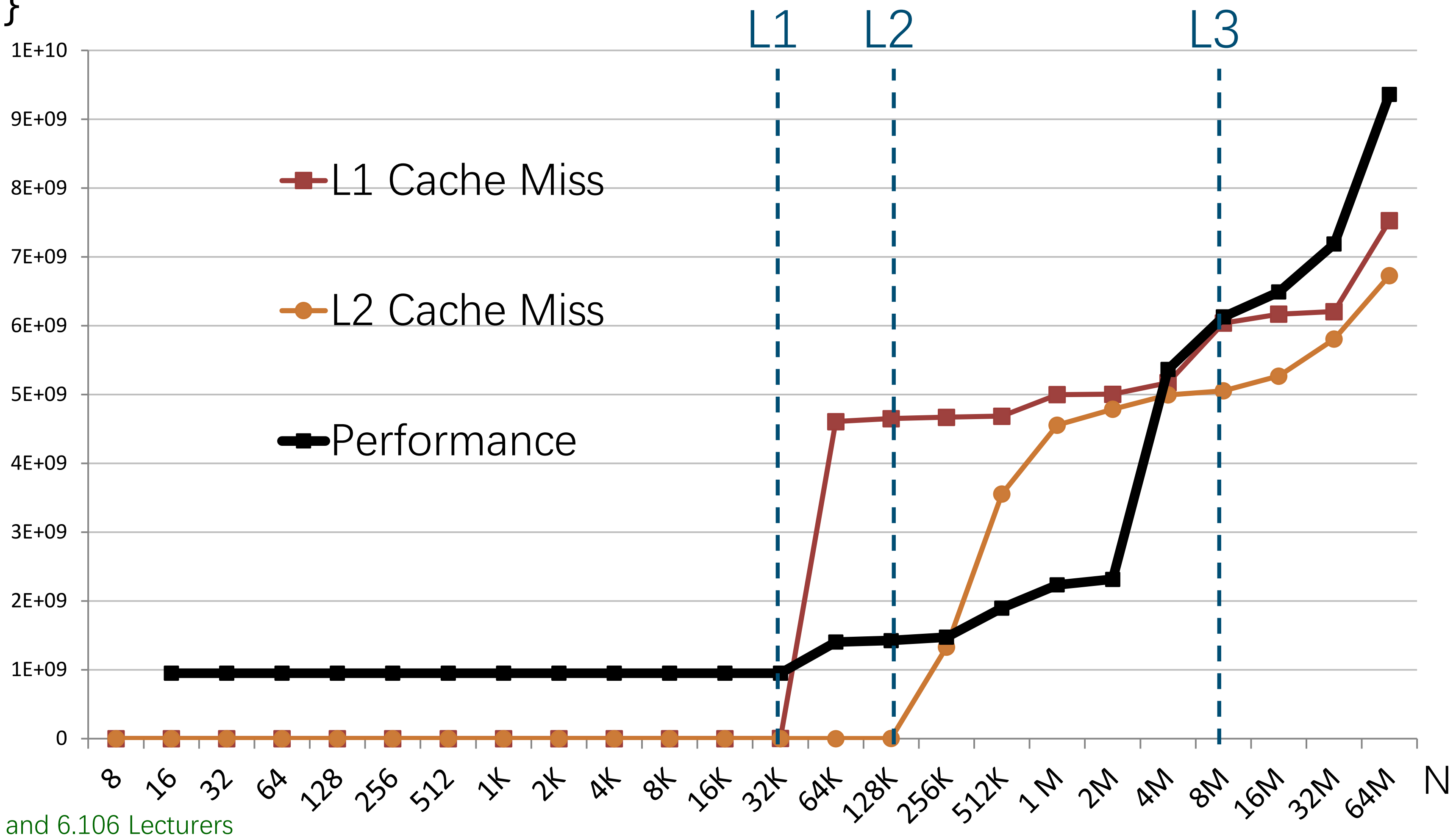
Intel® Nehalem™ Processor

```
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
      addr = ((rep + 523)*253573) & mask;
      A[addr] = A[addr] + 1;
}
```



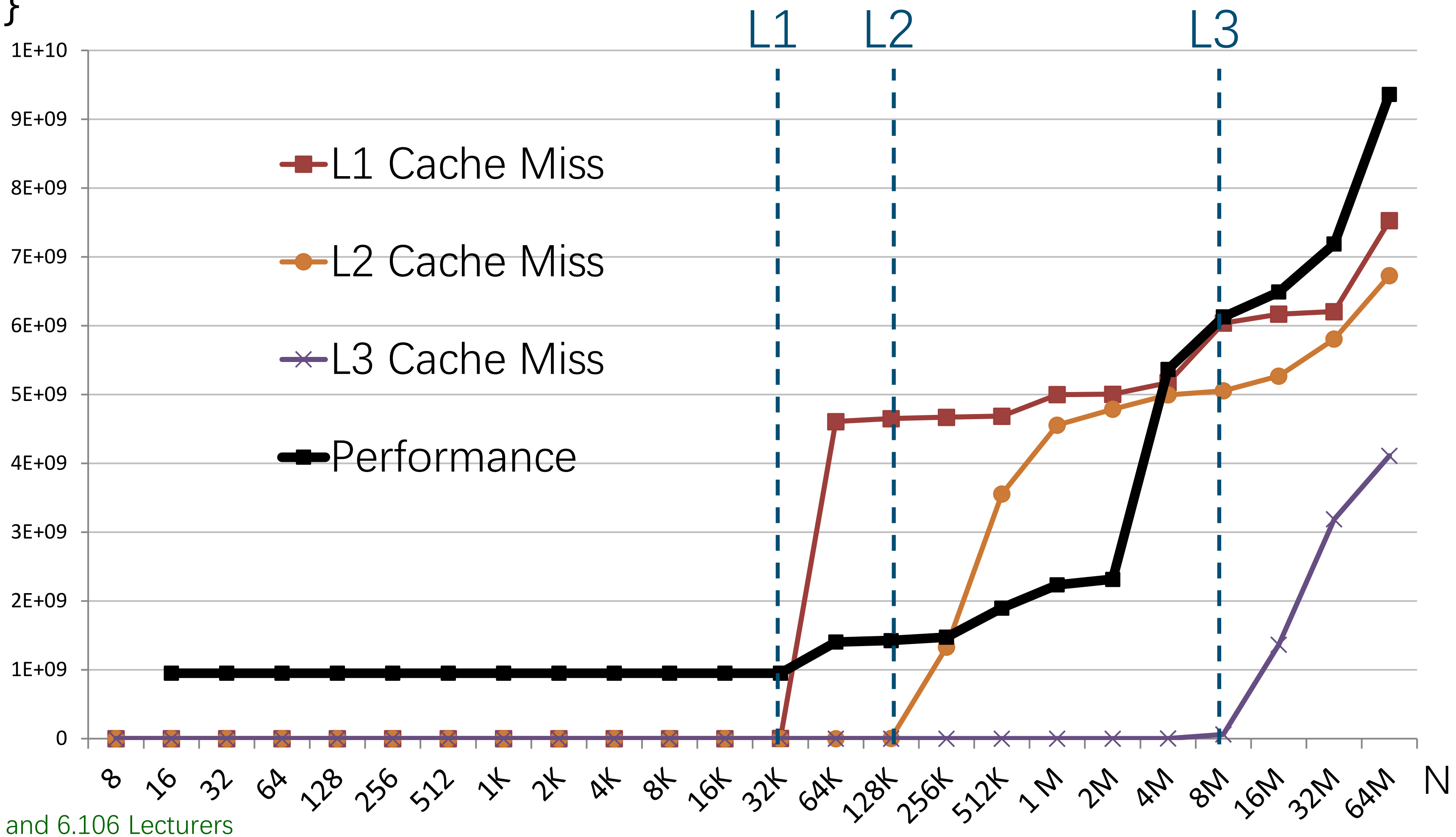
Intel® Nehalem™ Processor

```
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
      addr = ((rep + 523)*253573) & mask;
      A[addr] = A[addr] + 1;
}
```



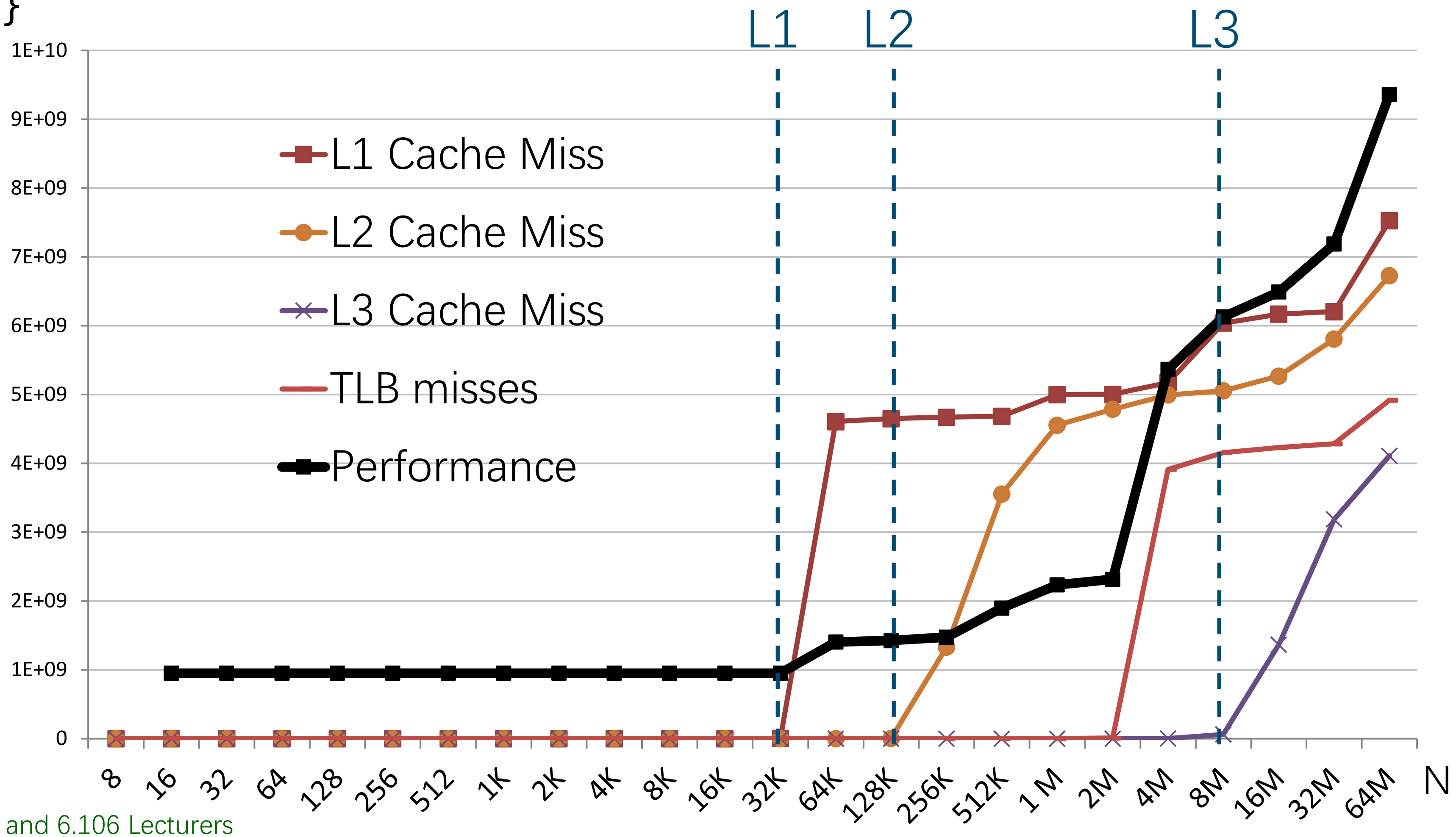
Intel® Nehalem™ Processor

```
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
      addr = ((rep + 523)*253573) & mask;
      A[addr] = A[addr] + 1;
}
```



Intel® Nehalem™ Processor

```
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
      addr = ((rep + 523)*253573) & mask;
      A[addr] = A[addr] + 1;
}
```



Virtual Memory System

You access virtual memory, your computer has physical memory & disk

- 2^{64} virtual memory
- Limited physical memory
- All allocated memory backed up on disk

Virtual2physical mapped by pages

- X86: 4KB small, 2MB large, and 1GB huge pages

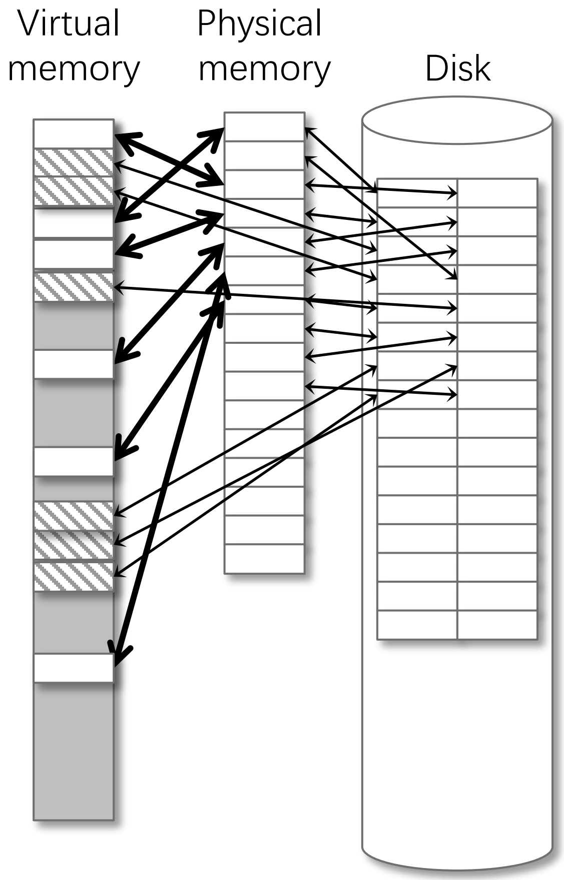
OS Manages Virtual memory

- Allocates virtual pages, maps them to physical
- Backs pages on disk and bring them in and out
- provides a page table to the hardware

Hardware caches page table entries in the TLB

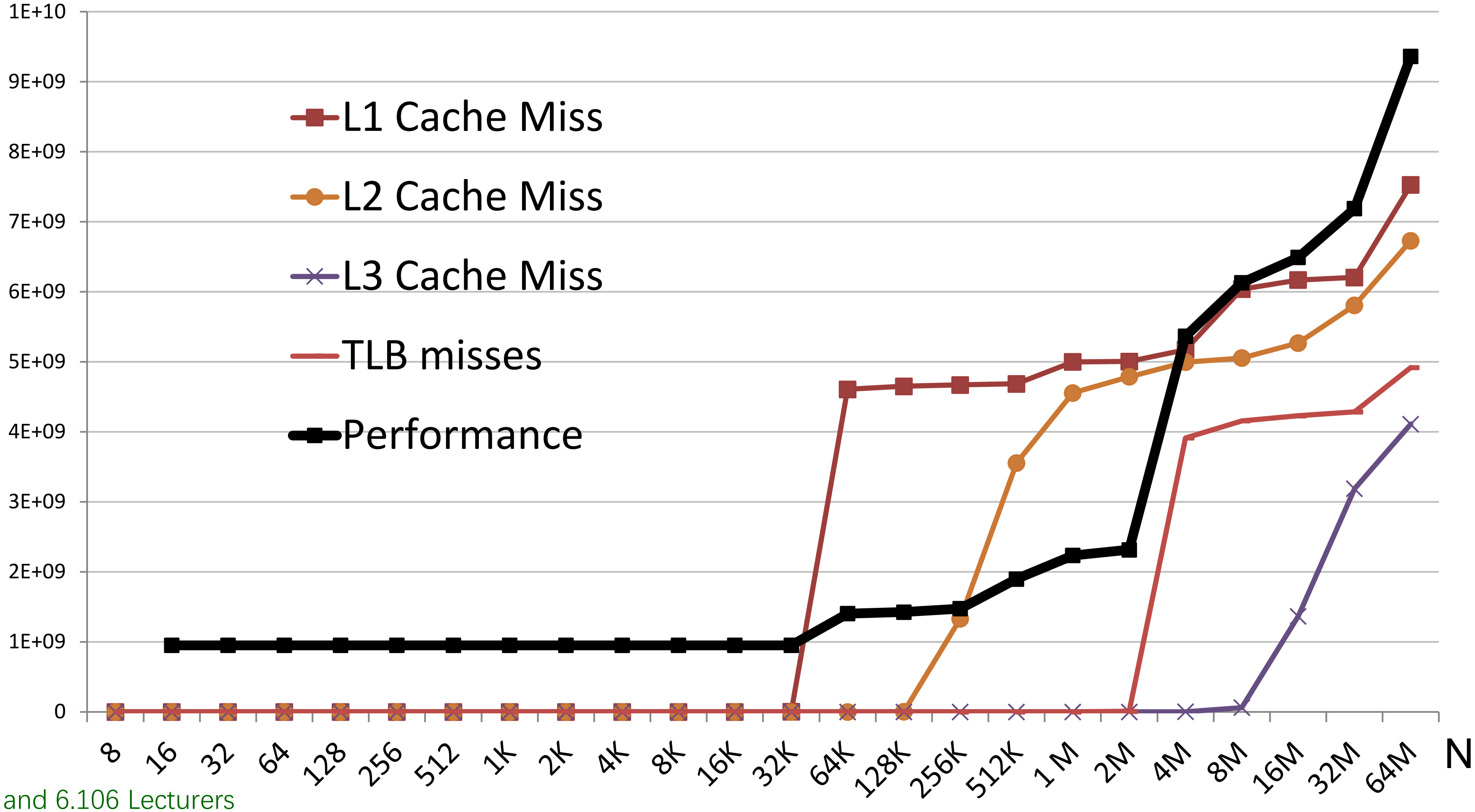
When you access a memory location

- If that page is mapped to physical memory and the mapping is cached in TLB → aok (~1 cycle)
- If mapping is not in TLB → TLB miss. (~100 cycles)
  - The HW gets the mapping from the page table and caches it in TLB
- If page is not mapped → Page fault. (~1,000,000 cycles)
  - The OS has to get involved in bringing in the page to physical memory from disk and updating the page table



Intel® Nehalem™ Processor

```
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
      addr = ((rep + 523)*253573) & mask;
      A[addr] = A[addr] + 1;
}
```



My Nehalem TLB Story

- Page size was set to 4 KB
- Number of TLB entries is 512

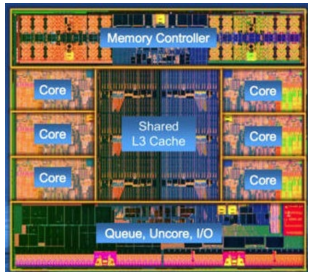
- So, total memory that can be mapped by TLB is 2 MB
- L3 cache is 8 MB!

- TLB misses before L3 cache misses!

Evolution of TLBs

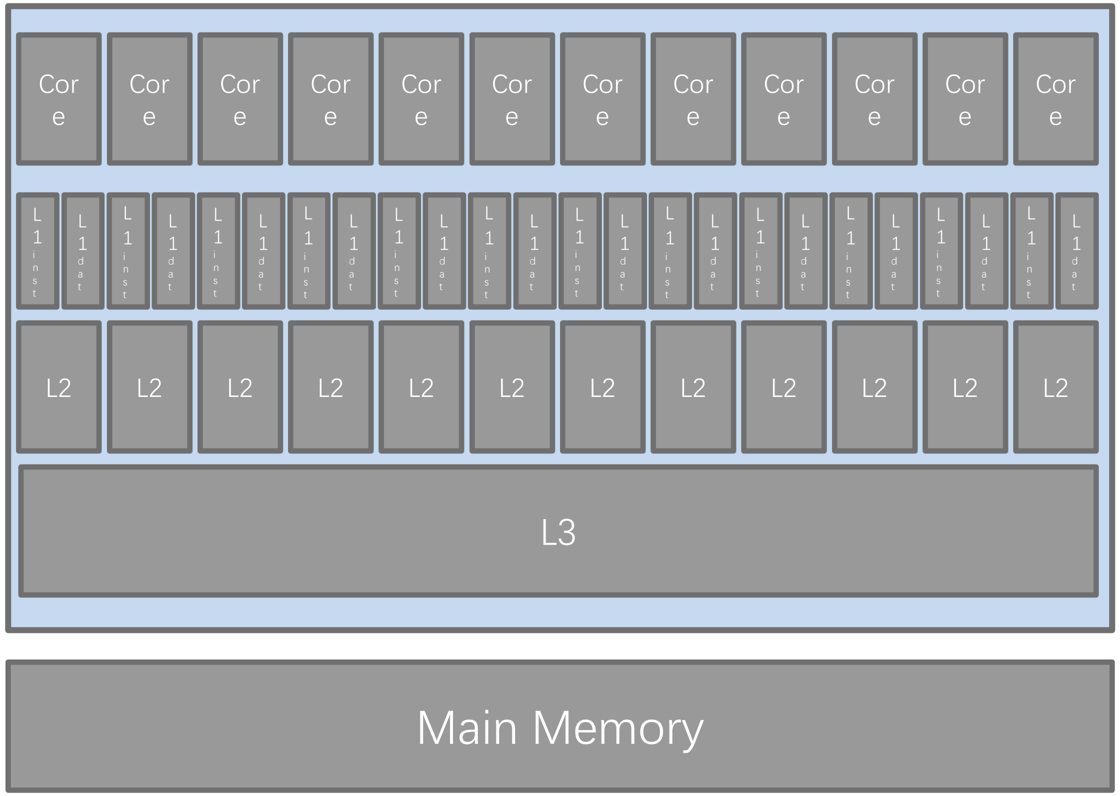
Year	2000	2008	2019
Processor	Pentium 4	Nehalem	Ice Lake
Max L1 TLB size	64	64	128
Max L2 TLB size		512	2048
Page sizes	4KB, 2MB	4KB, 2MB, 1GB	4KB, 2MB, 1GB

Intel® IvyBridge™v2 E5-2692 – Memory Sub-system



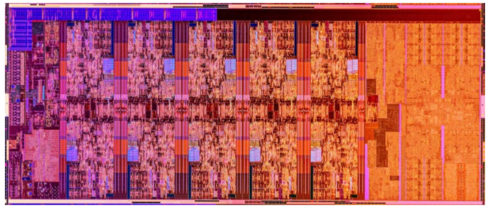
2012

Intel 12 Core Processor



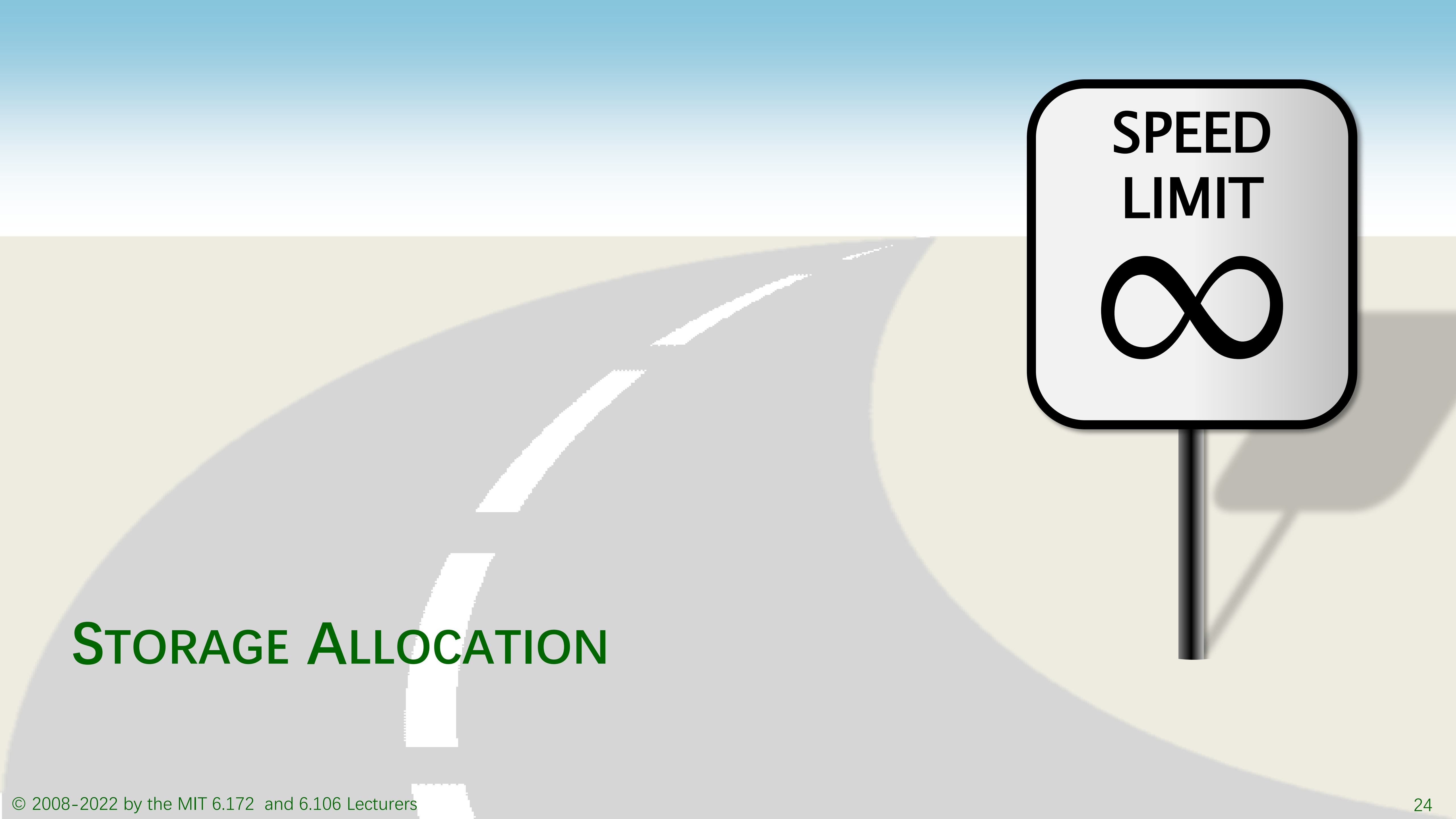
L1 Data Cache			
Size	Line Size	Latency	Associativity
32 KB	64 bytes	4 ns	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativity
32 KB	64 bytes	4 ns	8-way
L2 Cache			
Size	Line Size	Latency	Associativity
256 KB	64 bytes	12 ns	8-way
L3 Cache			
Size	Line Size	Latency	Associativity
8 MB	64 bytes	50 ns	16-way
Main Memory			
Size	Line Size	Latency	Associativity
	64 bytes	85 ns	

Intel Sunny Cove/Ice Lake – Memory Sub-system



2019

L1 Data Cache			
Size	Line Size	Latency	Associativity
48 KB	64 bytes	5 ns	12-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativity
32 KB	64 bytes	5 ns	8-way
L2 Cache			
Size	Line Size	Latency	Associativity
512 KB/core	64 bytes	14 ns	8-way
L3 Cache			
Size	Line Size	Latency	Associativity
8 MB	64 bytes	39-45 ns	16-way



STORAGE ALLOCATION

Dynamic Storage Allocation

Kinds of storage management

Stack

Heap

Garbage-Collected

Stack Allocation

Stack discipline

- LIFO (last in, first out).
- The object that was most recently allocated (pushed) is the next to be freed (popped).

C call stack

- Stores the local variables for function instantiations.
- A frame is pushed onto the stack when the function is called.
- The frame is popped when the function returns.

A cafeteria plate dispenser obeys a stack discipline.

Heap Allocation*

- Memory space available to the programmer that can be allocated and deallocated without constraint.
  - C provides `malloc()` and `free()`.
  - C++ provides `new` and `delete`.
- Heap storage must be freed explicitly.
- Failure to do so creates a memory leak.
- Watch out for dangling pointers (pointers to freed memory) and double freeing (freeing memory that has already been freed).
- Memory checkers (e.g., AddressSanitizer, Valgrind) can assist in finding these pernicious bugs. Use them!

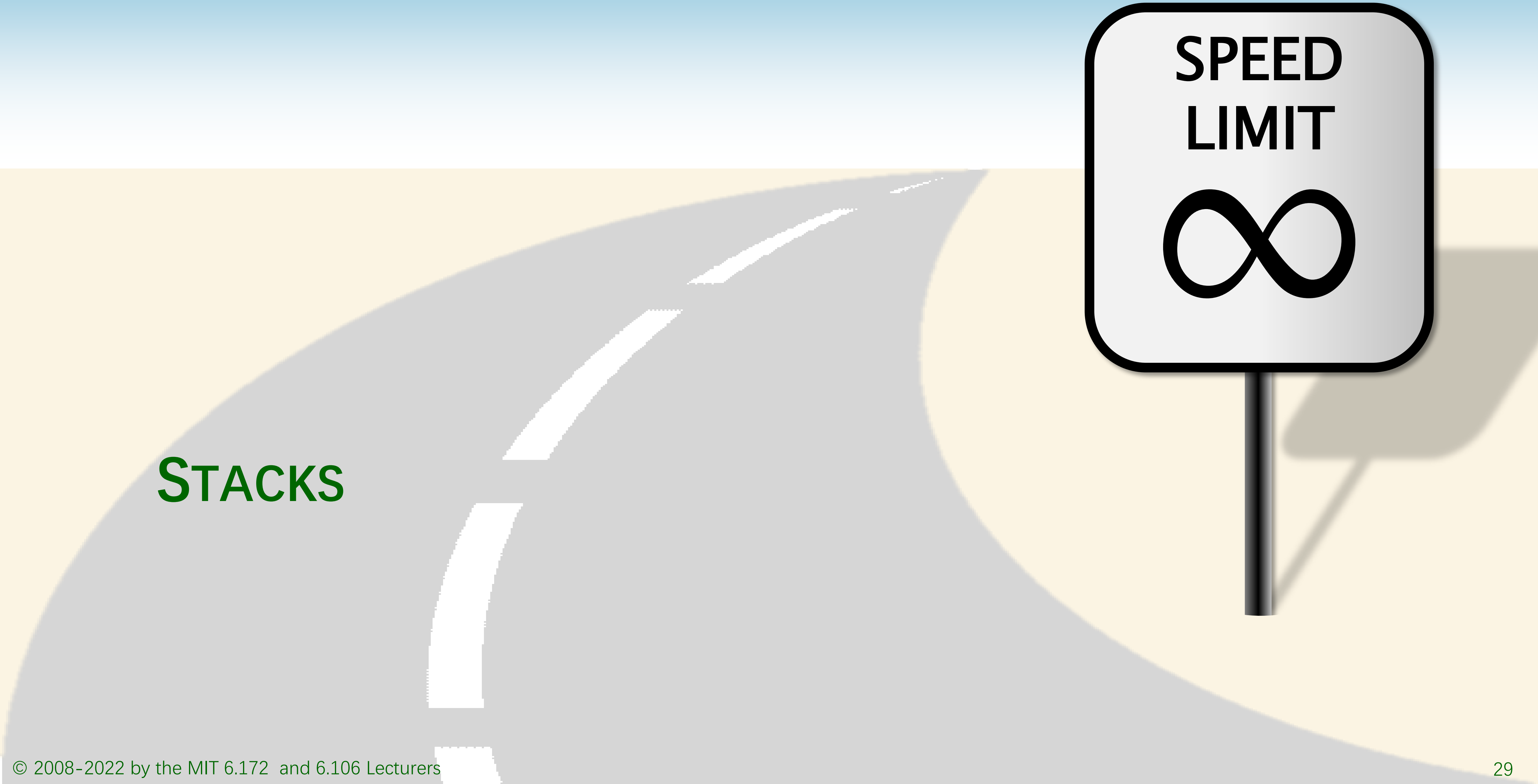
---

*Do not confuse with a heap data structure.

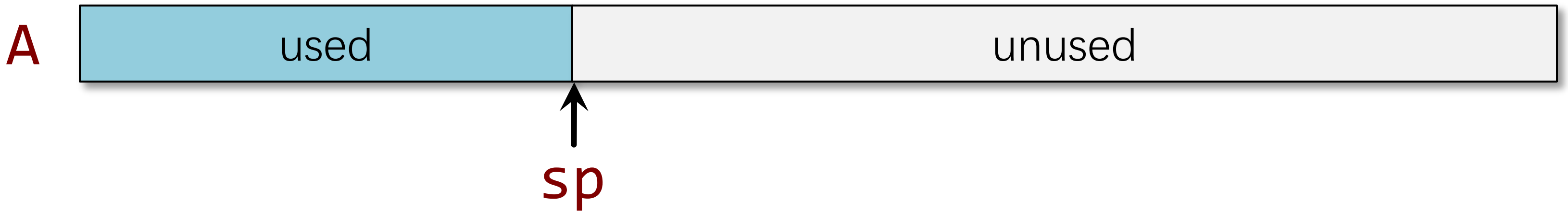
Garbage Collection

- Unlike heap storage, garbage-collected storage need not be freed explicitly, greatly aiding in programmer productivity.
- Available in most higher-level languages (e.g., Python, Java, Julia).
- The garbage collector looks for storage that the program can no longer access and reclaims it.
- The garbage collector can pause the executing program, run in real time, or operate concurrently.
- Garbage collection is usually slower than `malloc()` and `free()`, because allocated storage is rarely in the L1-cache.

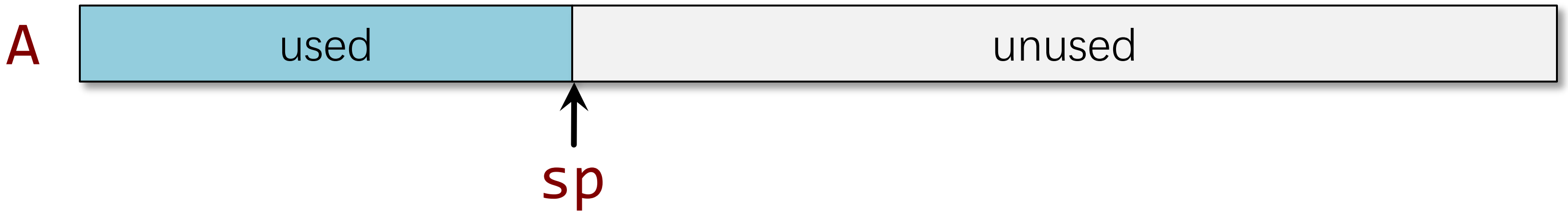
STACKS



Array and Pointer



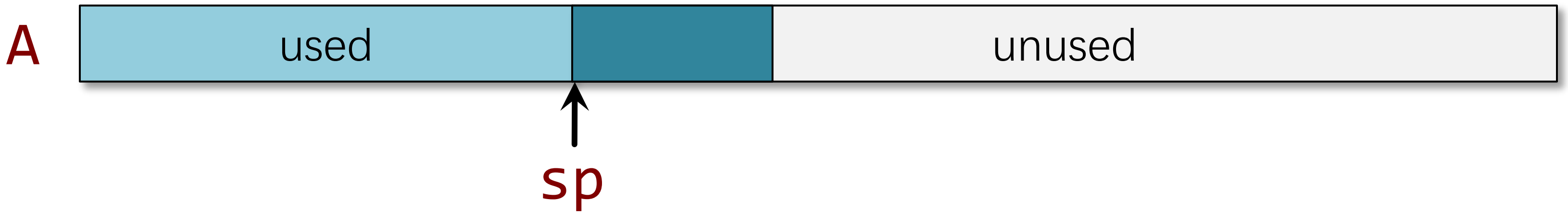
Array and Pointer: Allocating



Allocate x bytes

```
sp += x;
return sp – x;
```

Array and Pointer: Allocating

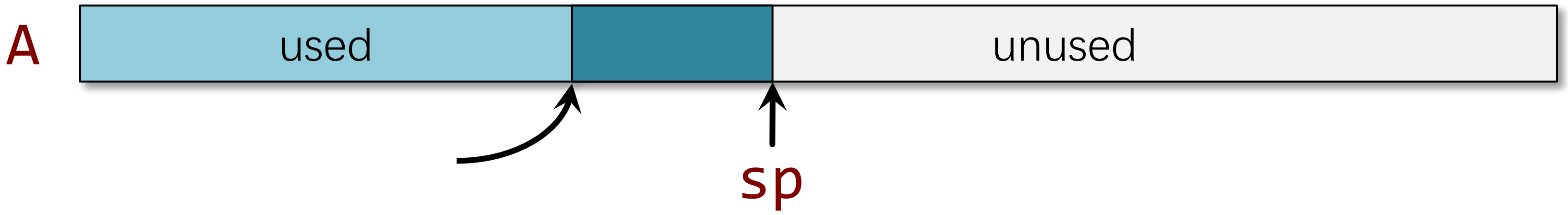


Allocate x bytes

```
sp += x;
return sp – x;
```

Should check for stack overflow.

Array and Pointer: Allocating

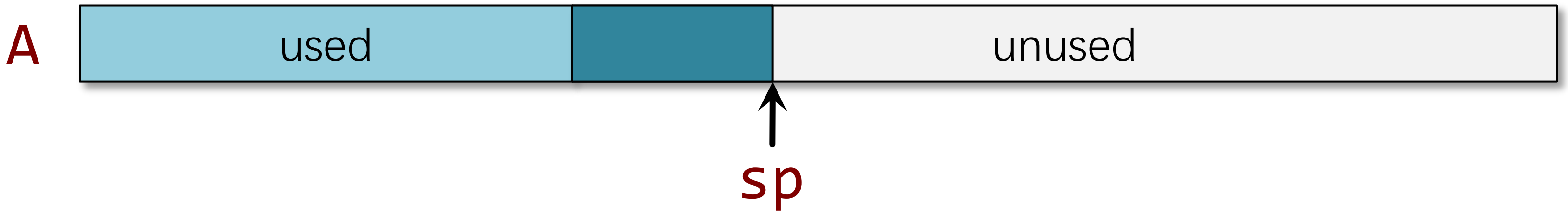


Allocate x bytes

```
sp += x;
return sp - x;
```

No math if stack grows downward, but it doesn't really matter, because integer arithmetic is fast, and the processor core has many ALU's.

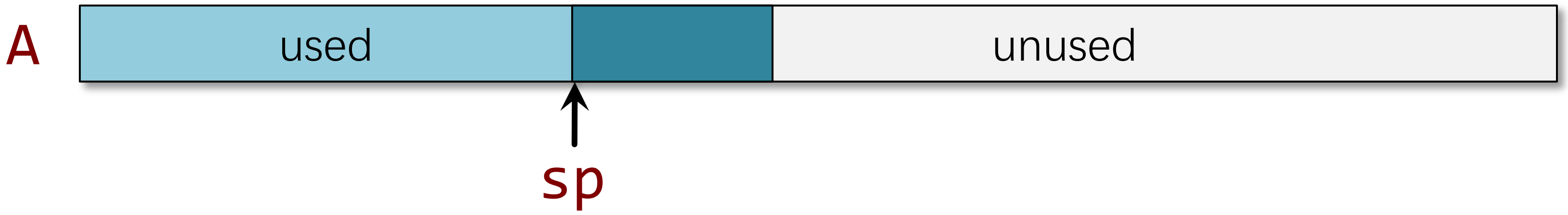
Array and Pointer: Deallocating



Free x bytes

```
sp -= x;
```

Array and Pointer: Deallocating

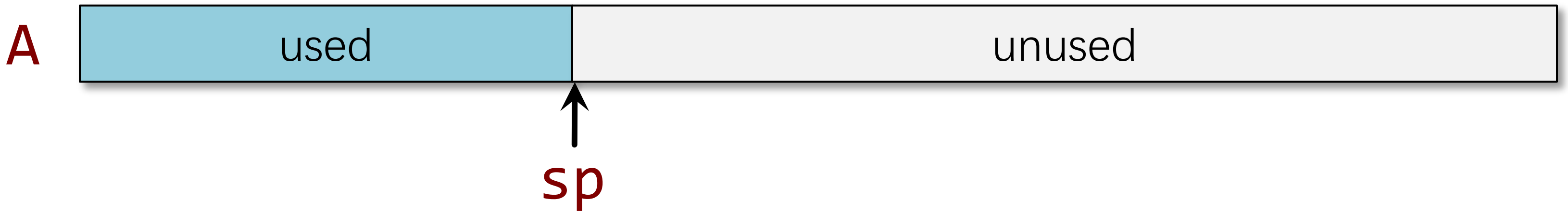


Free x bytes

```
sp -= x;
```

Should check for stack underflow.

Summary of Stacks



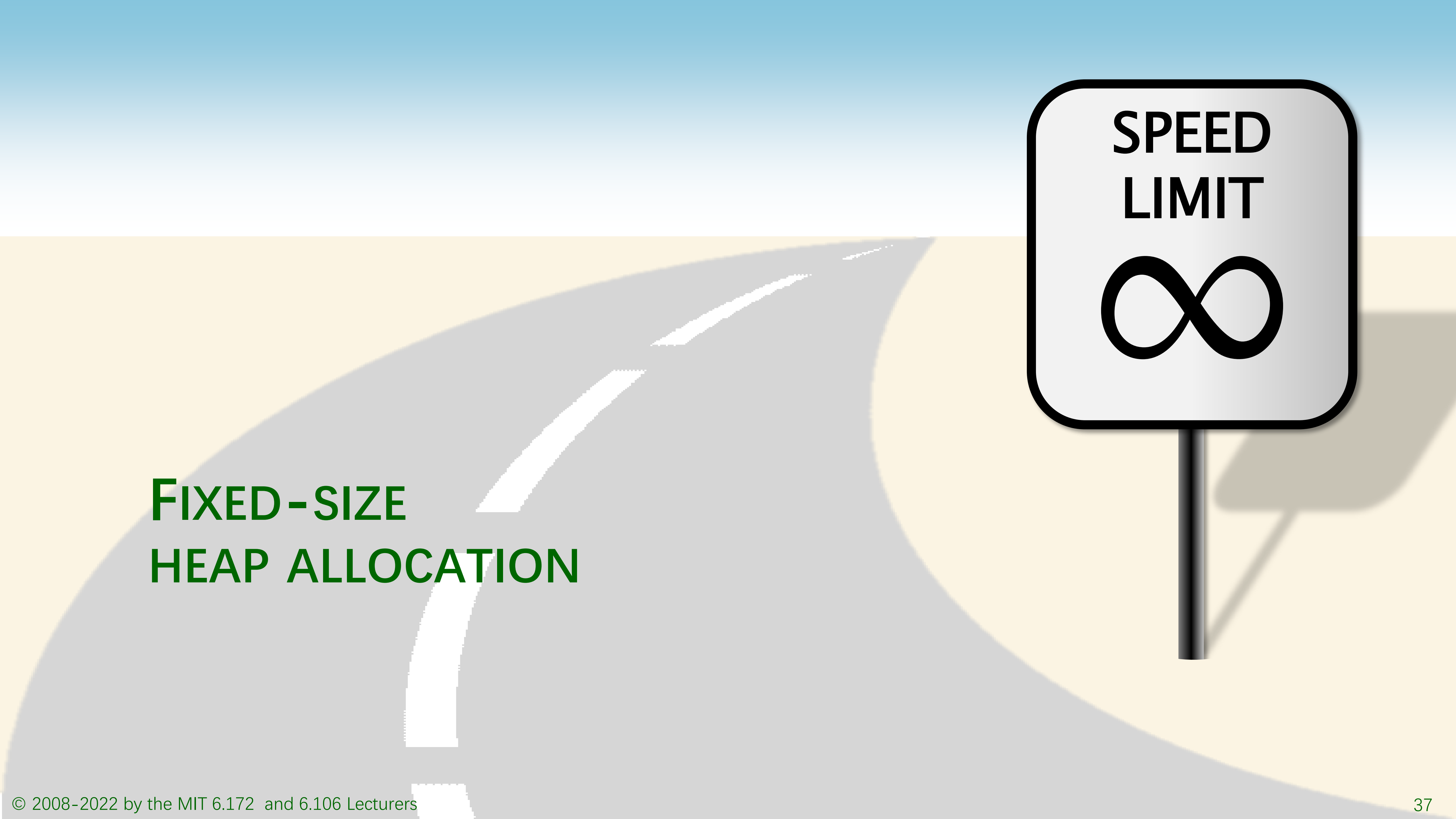
Allocate x bytes

```
sp += x;
return sp – x;
```

Free x bytes

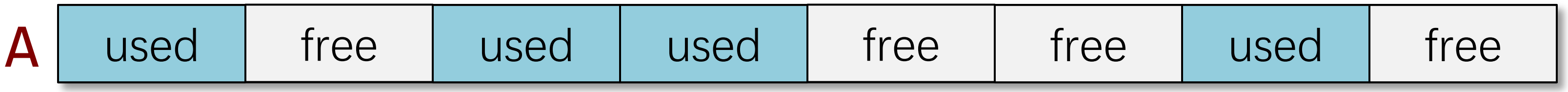
```
sp -= x;
```

- Allocating and freeing take Θ(1) time.
- Must free consistent with stack discipline.
- Limited applicability, but great when it works!
- One can allocate on the call stack using `alloca()`, but this function is deprecated, and the compiler is more efficient with fixed-size frames.



FIXED-SIZE HEAP ALLOCATION

Bitmap Allocator

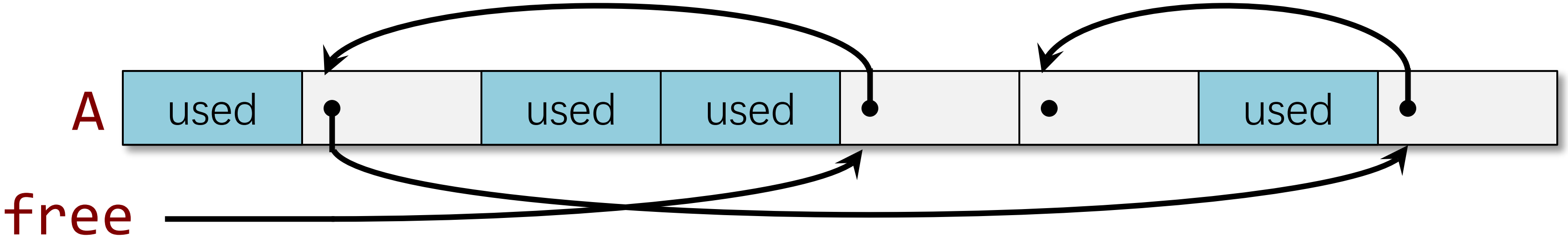


A	used	free	used	used	free	free	used	free

`bitmap: 01001101`

- Use a bitmap to keep track of which blocks of `A` are free and which are used.
- Block sizes can be arbitrarily small.
- Bit tricks can help speed the search for a free block — e.g., `bitmap & (-bitmap)` — but the approach is fundamentally not scalable (linear-time search).
- A multilayer hierarchy can sometimes be helpful: e.g., a bitmap per page and a bitmap for pages.

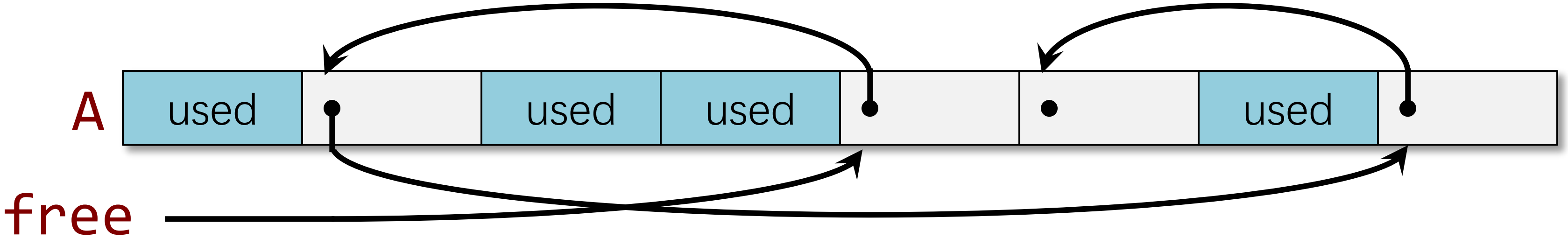
Free List



- Every piece of storage has the same size.
- Each unused storage block contains a pointer to the next unused block.
  - The block size must be at least as big as a pointer.

```
struct freelist_item {
  void *next;
}
```

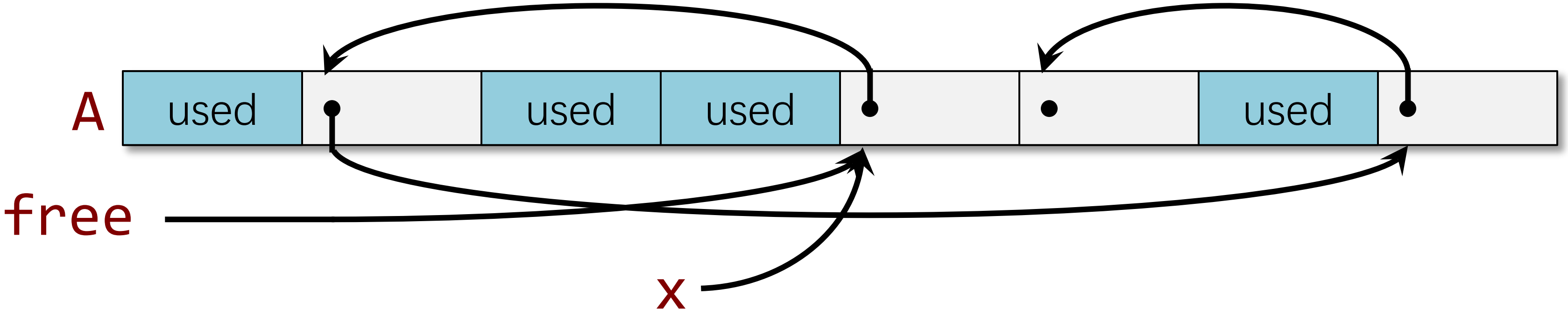
Free-List: Allocating



Allocate 1 object

```
x = free;
free = free->next;
return x;
```

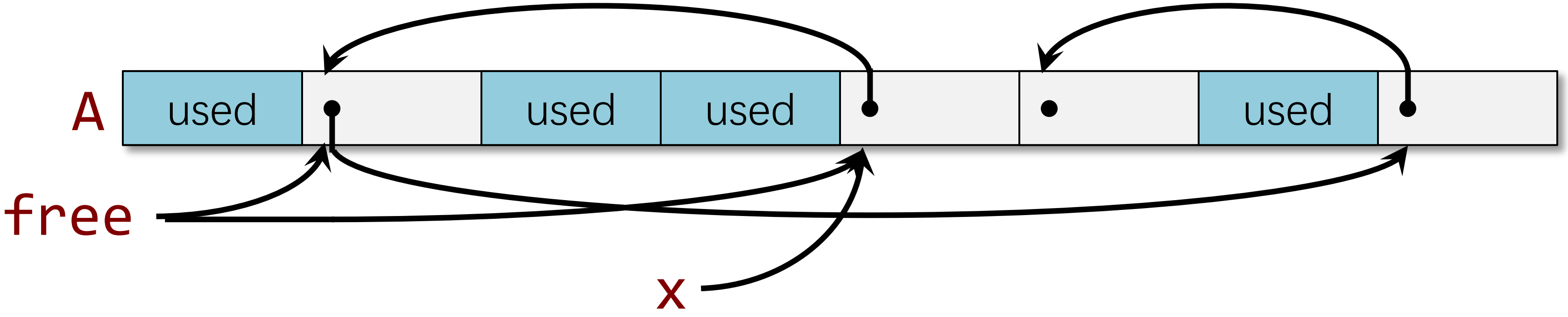
Free-List: Allocating



Allocate 1 object

```
x = free;
free = free->next;
return x;
```

Free-List: Allocating

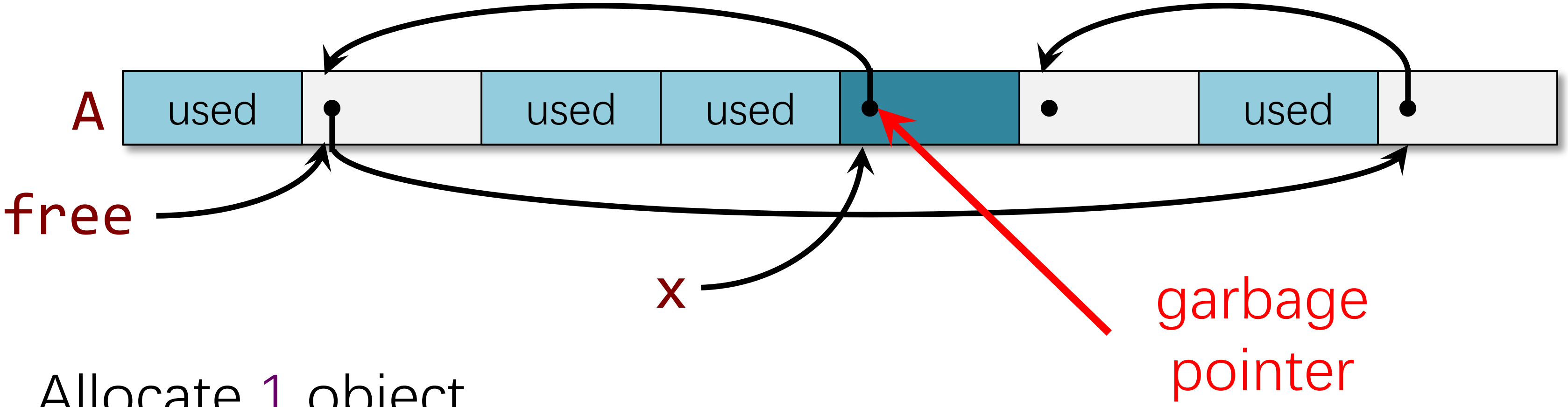


Allocate 1 object

```
x = free;
free = free->next;
return x;
```

Should check `free != NULL`.

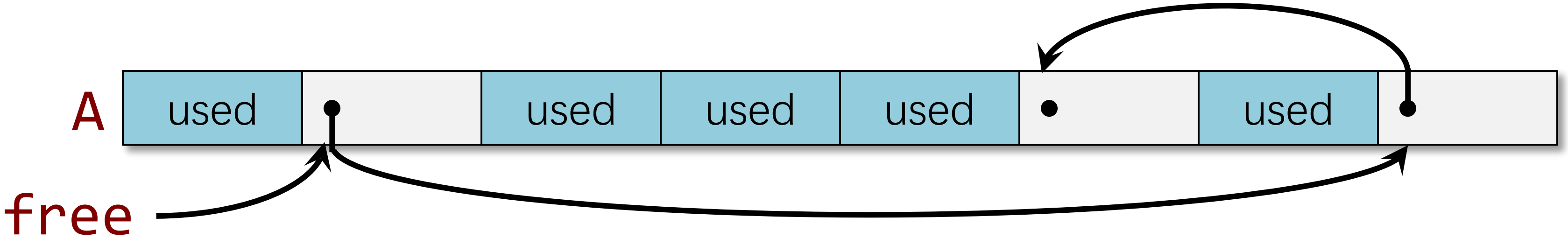
Free-List: Allocating



Allocate 1 object

```
x = free;
free = free->next;
return x;
```

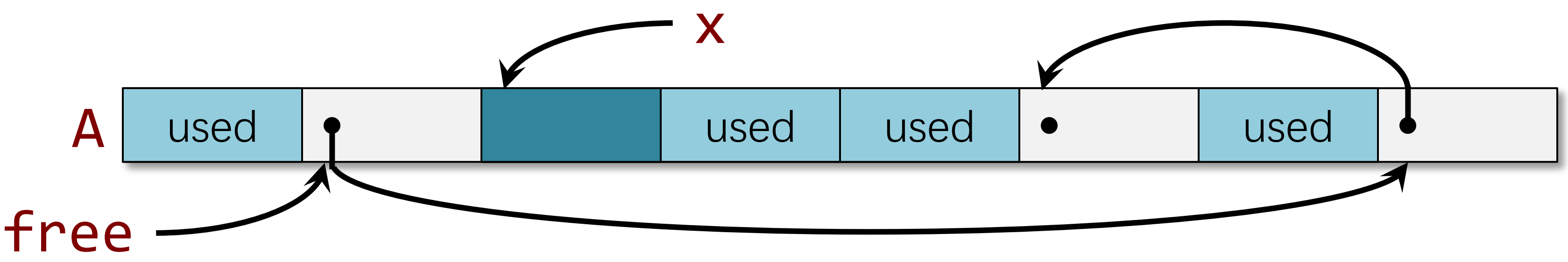
Free-List: Allocating



Allocate 1 object

```
x = free;
free = free->next;
return x;
```

Free-List: Deallocating



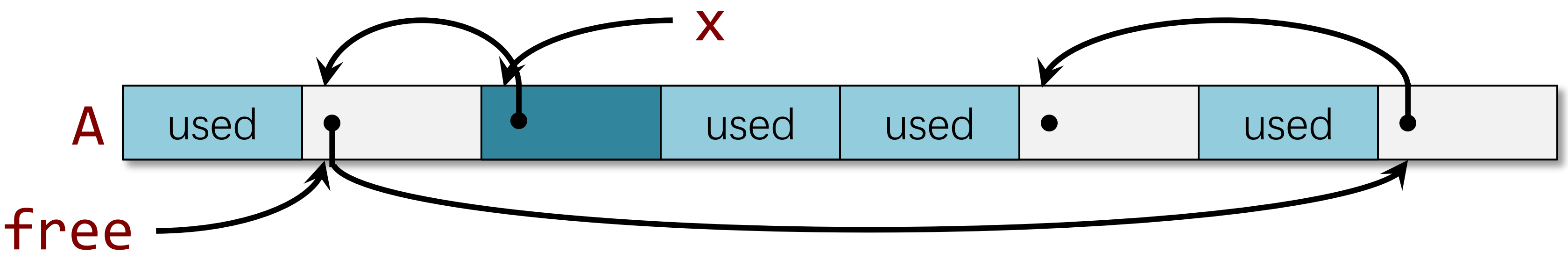
Allocate 1 object

```
x = free;
free = free->next;
return x;
```

free object **x**

```
x->next = free;
free = x;
```

Free-List: Deallocating



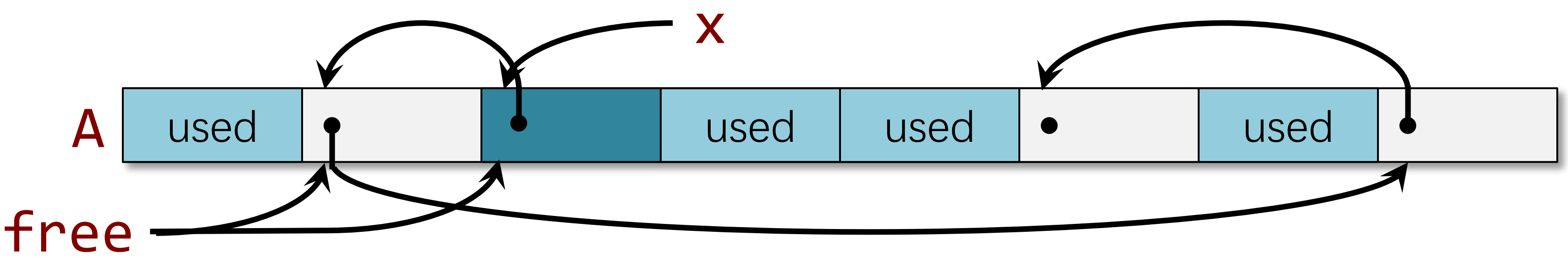
Allocate 1 object

```
x = free;
free = free->next;
return x;
```

free object x

```
x->next = free;
free = x;
```

Free-List: Deallocating



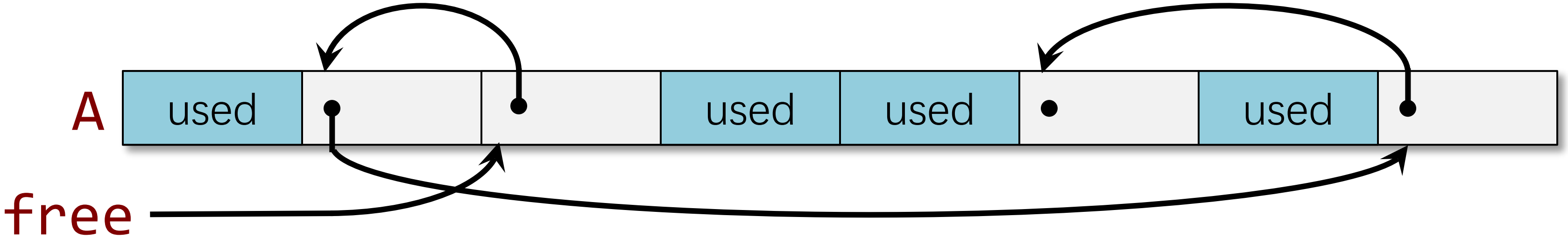
Allocate 1 object

```
x = free;
free = free->next;
return x;
```

free object x

```
x->next = free;
free = x;
```

Free-List: Deallocating



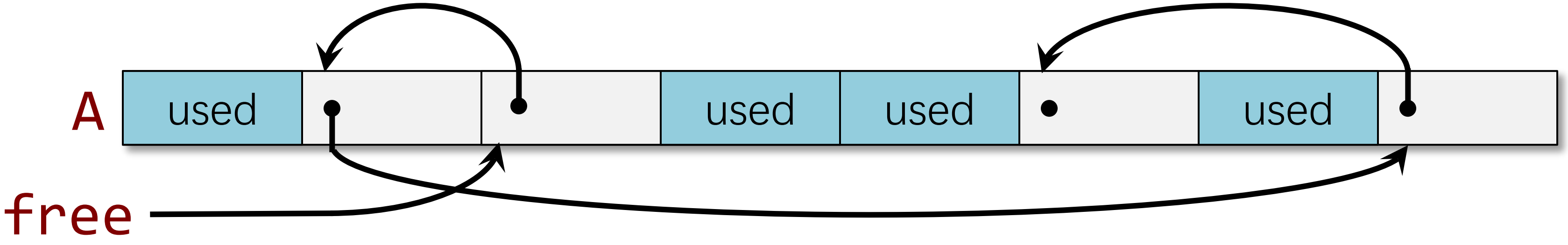
Allocate 1 object

```
x = free;
free = free->next;
return x;
```

free object x

```
x->next = free;
free = x;
```

Summary of Free Lists



- Allocating and freeing take Θ(1) time.
- Good temporal locality.
- Poor spatial locality due to external fragmentation — blocks distributed across virtual memory — which can increase the size of the page table and cause disk thrashing.
- The translation lookaside buffer (TLB) can also be a problem.

Fragmentation

Internal Fragmentation

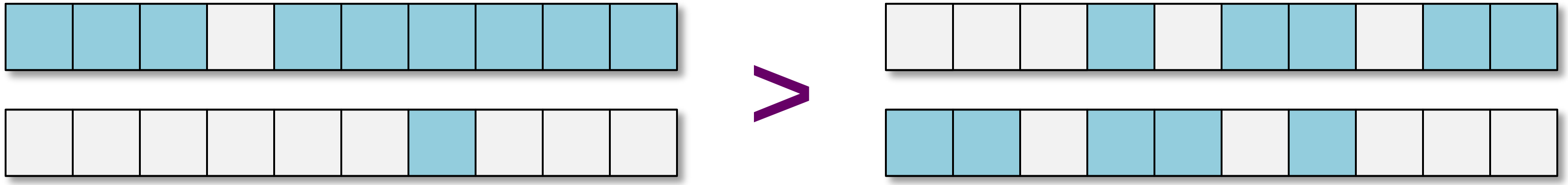
- When blocks larger than what was required are given.
  - i.e. ask for block of size 2^k+1 will get a block of size 2^{k+1}
- Worst case: No blocks of asking size left, but a lot of unused space in allocated blocks

External Fragmentation

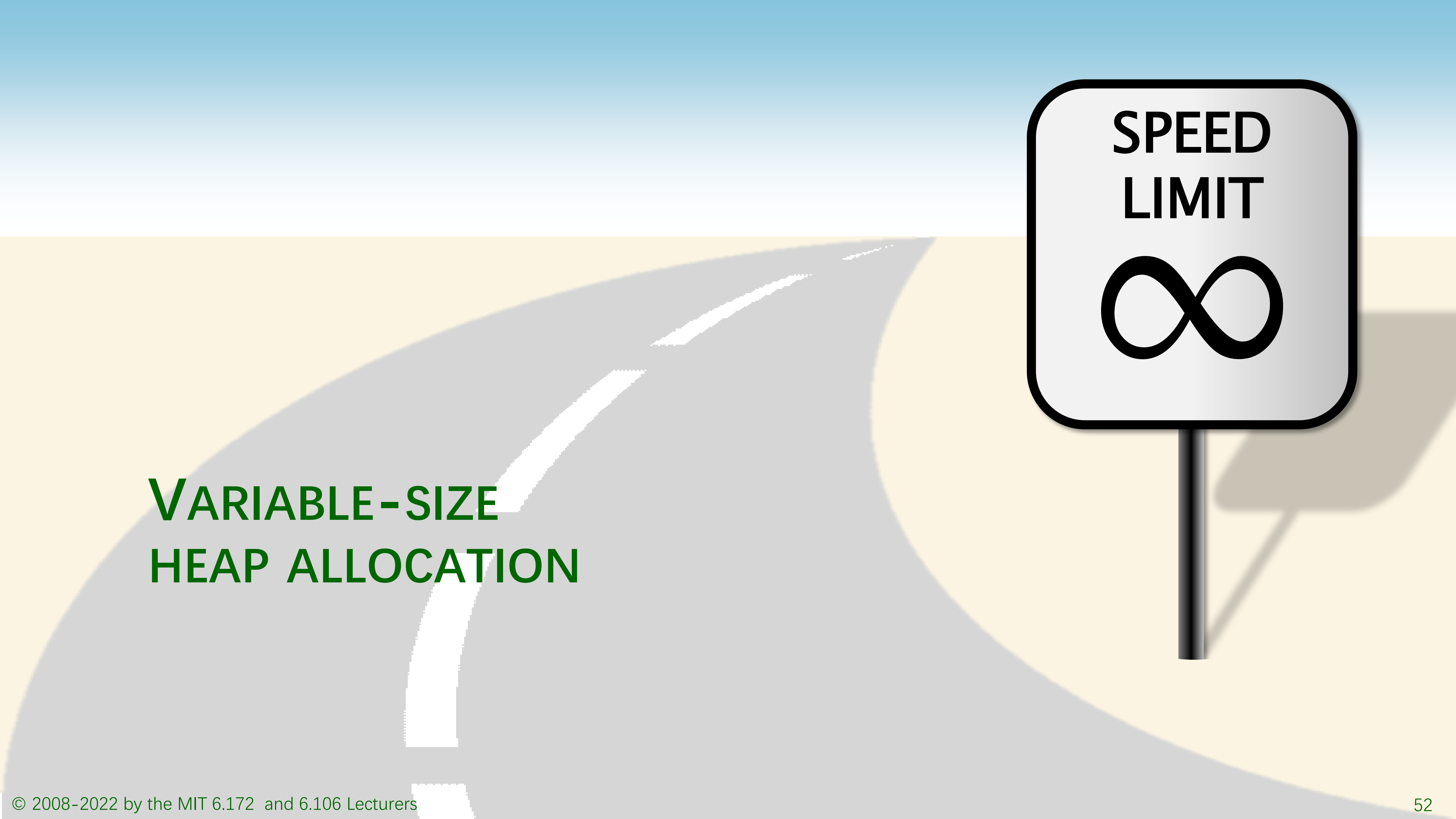
- A free blocks and allocated blocks interspersed.
- Bad spatial locality
- Worst case: no block of a given size, while there are a lot of smaller free blocks, but no contiguous blocks to coalesce.

Mitigating External Fragmentation

- Keep a free list (or bitmap) per disk page.
- Allocate from the free list for the fullest unfull page.
- To free a block of storage, add it to the free list for the page on which the block resides.
- If a page becomes empty (only free-list items), the virtual-memory system can page it out without substantial impact on program performance.
- 90-10 beats 50-50:



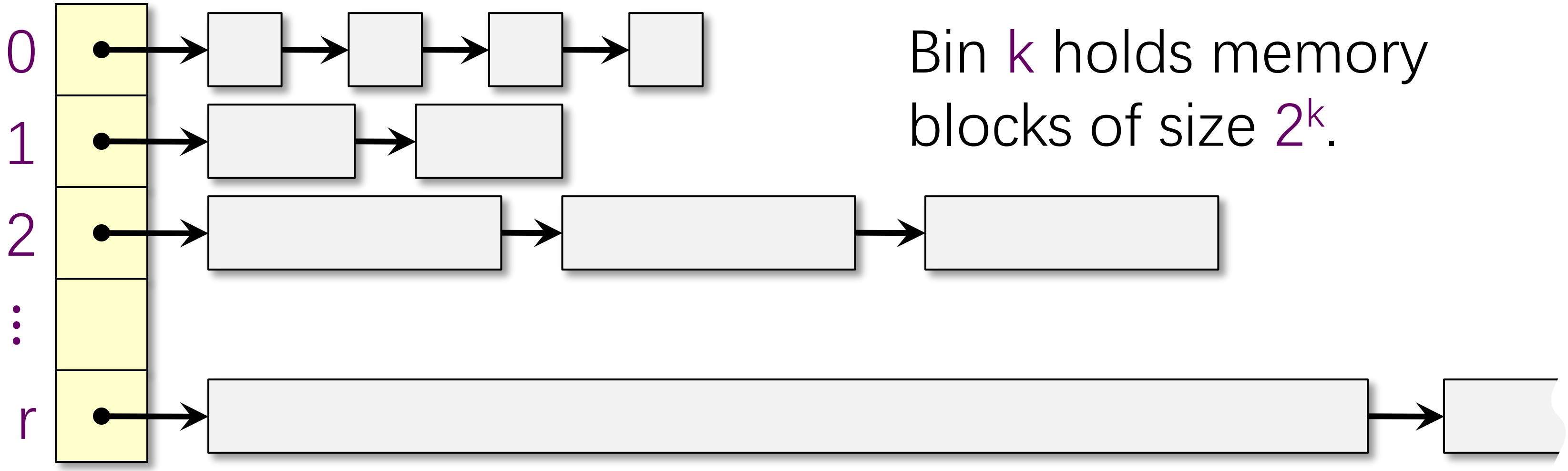
Probability that 2 random accesses hit the same page
$= .9 \times .9 + .1 \times .1 = .82$ versus $.5 \times .5 + .5 \times .5 = .5$



VARIABLE-SIZE HEAP ALLOCATION

Binned Free Lists

- Leverage the efficiency of free lists.
- Accept a bounded amount of internal fragmentation.

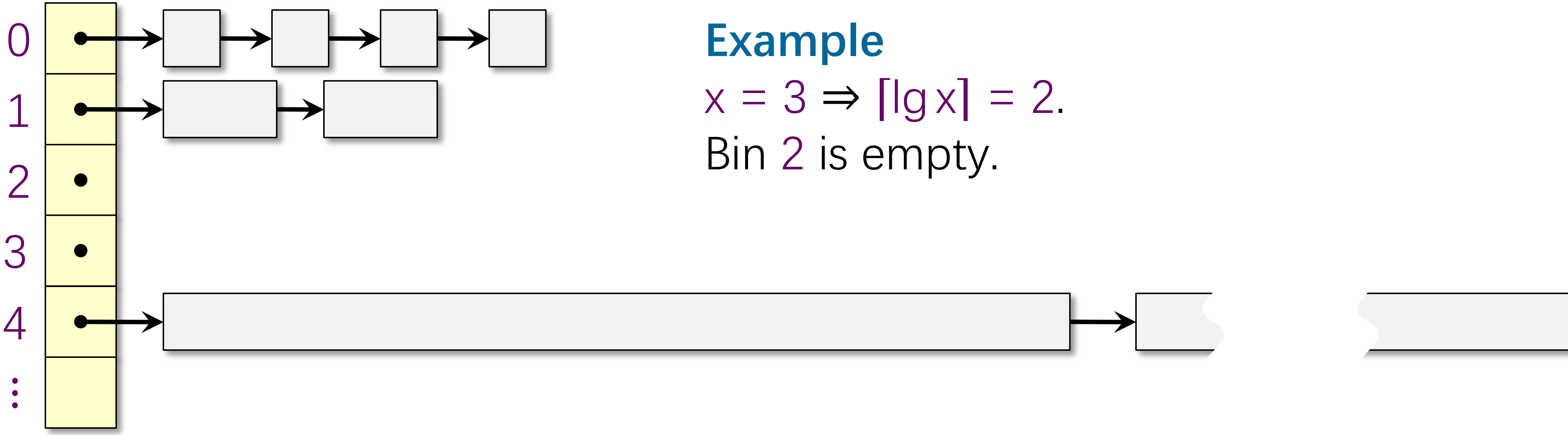


Bin k holds memory blocks of size 2^k.

Binned Free Lists: Allocating

Allocate
x bytes

- If bin $k = \lceil \lg x \rceil$ is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin $k' > k$, split it up into blocks of sizes $2^{k'-1}, 2^{k'-2}, \cdots, 2^k, 2^k$, and distribute the pieces.



Example

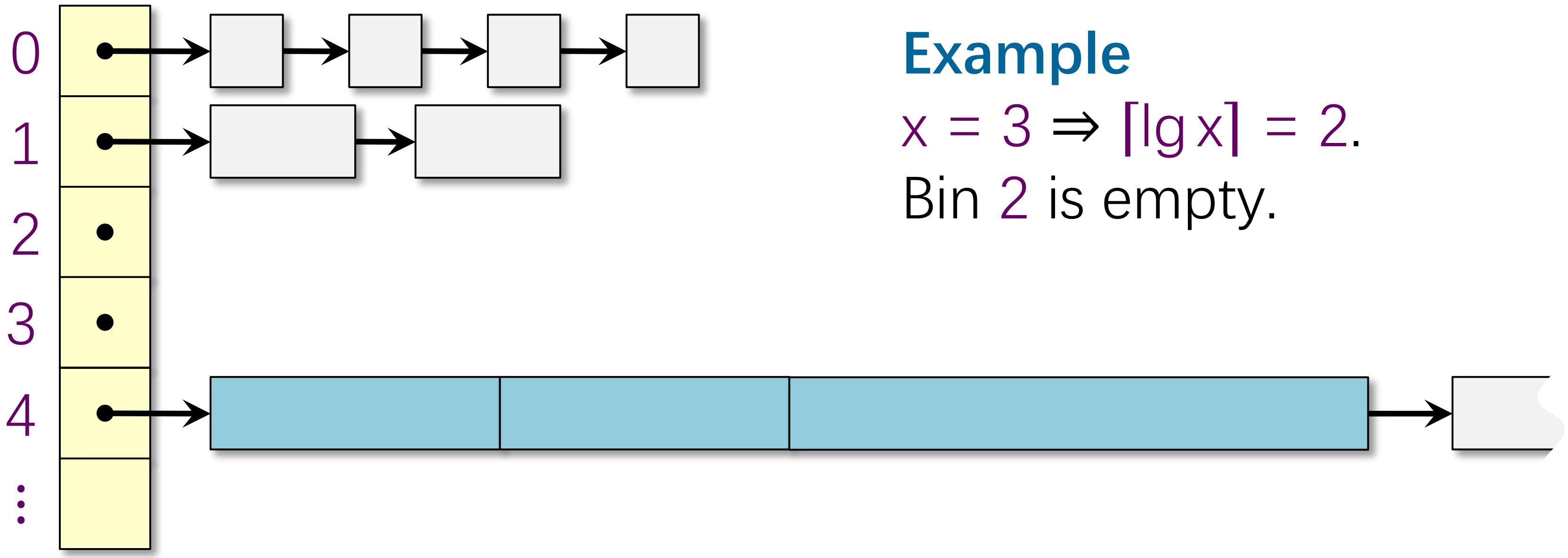
$x = 3 \Rightarrow \lceil \lg x \rceil = 2$.

Bin 2 is empty.

Binned Free Lists: Allocating

Allocate x bytes

- If bin $k = \lceil \lg x \rceil$ is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin $k' > k$, split it up into blocks of sizes $2^{k'-1}, 2^{k'-2}, \cdots, 2^k, 2^k$, and distribute the pieces.



Example

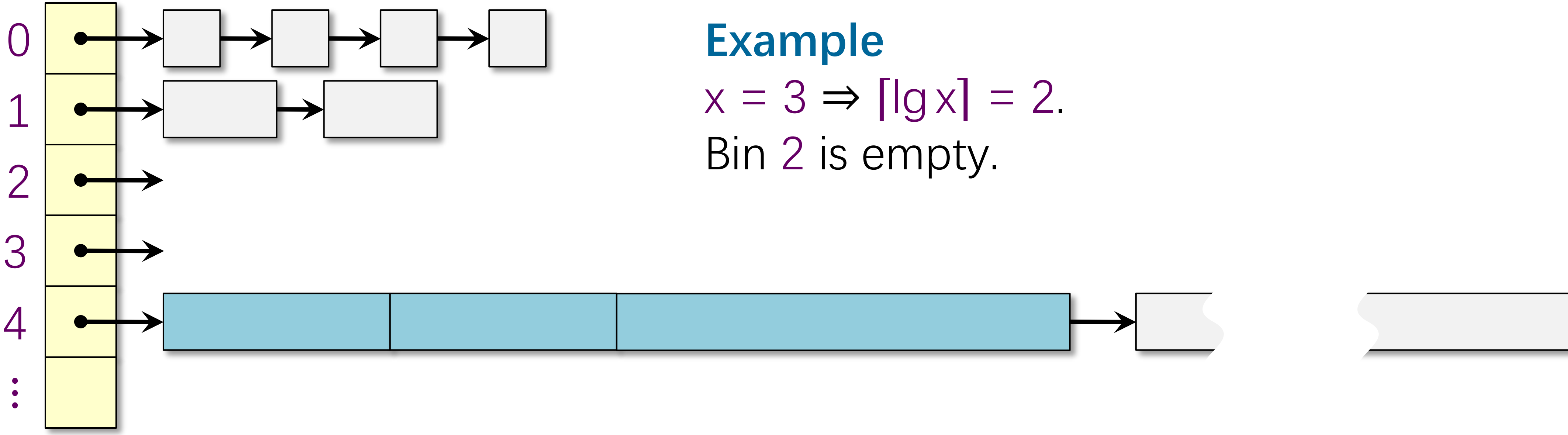
$x = 3 \Rightarrow \lceil \lg x \rceil = 2$.

Bin 2 is empty.

Binned Free Lists: Allocating

Allocate
x bytes

- If bin $k = \lceil \lg x \rceil$ is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin $k' > k$, split it up into blocks of sizes $2^{k'-1}, 2^{k'-2}, \cdots, 2^k, 2^k$, and distribute the pieces.



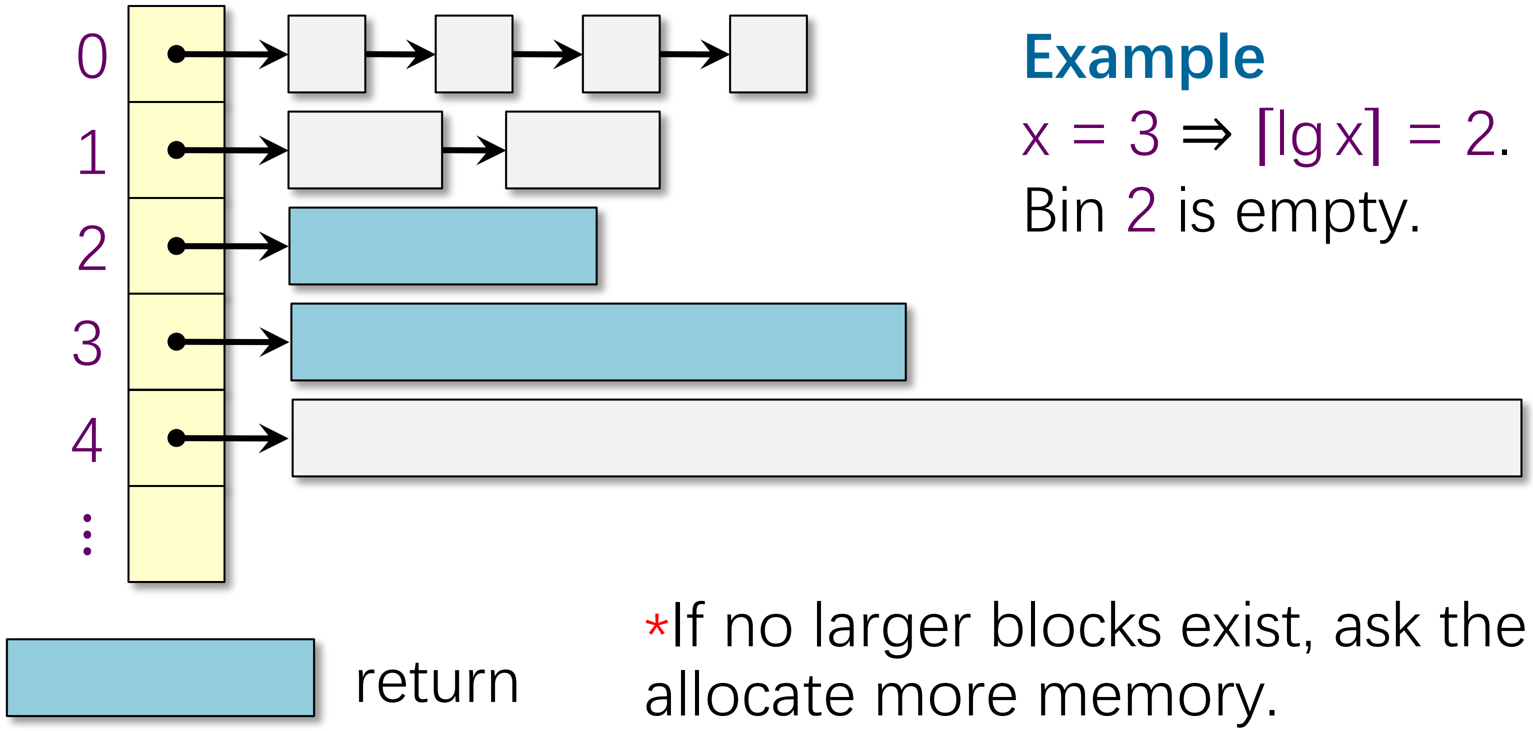
Example
$x = 3 \Rightarrow \lceil \lg x \rceil = 2$.
Bin 2 is empty.

return

Binned Free Lists: Allocating

Allocate
x bytes

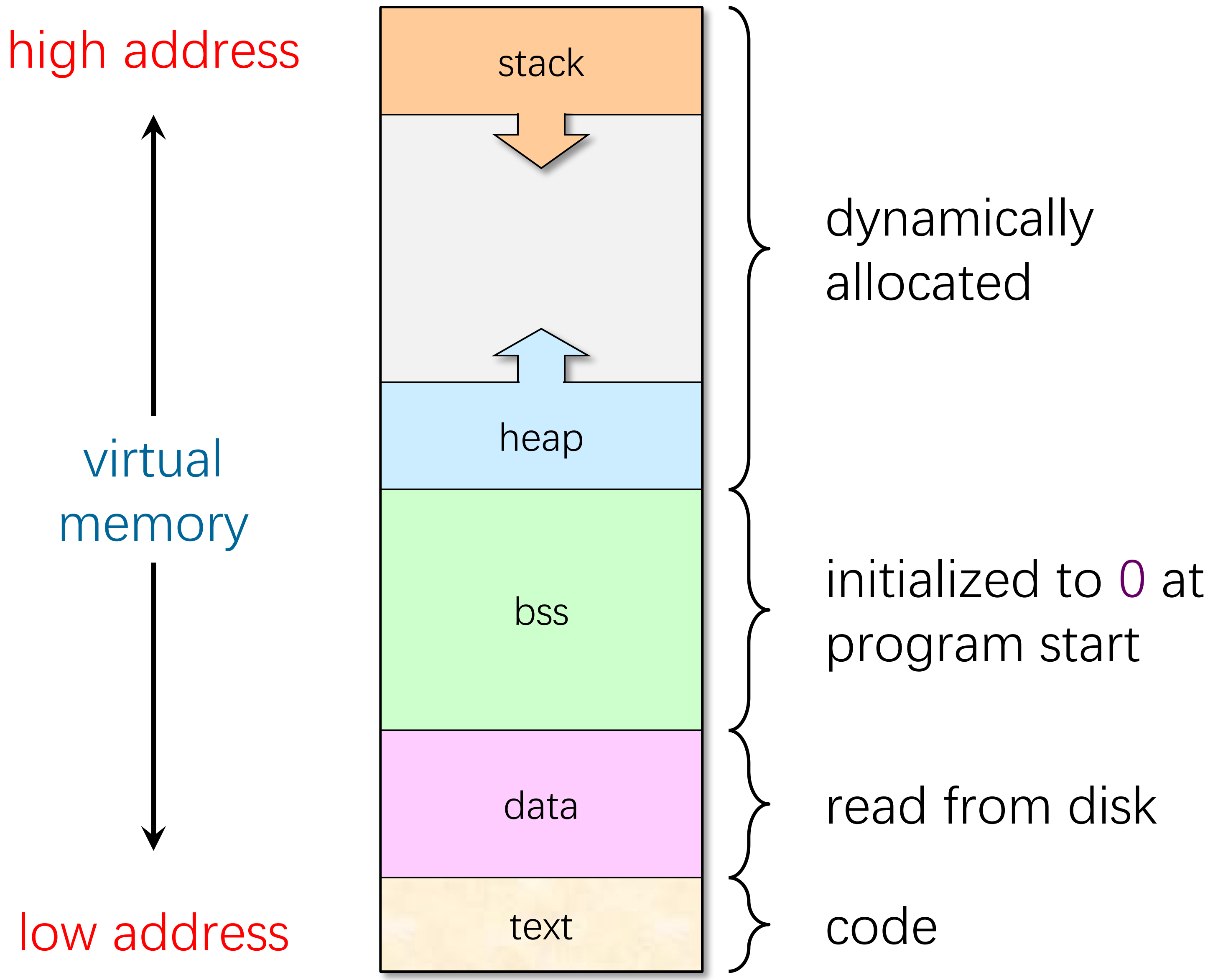
- If bin $k = \lceil \lg x \rceil$ is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin $k' > k$, split it up into blocks of sizes $2^{k'-1}$, $2^{k'-2}$, ⋯, 2^k, 2^k, and distribute the pieces.*



Example
$x = 3 \Rightarrow \lceil \lg x \rceil = 2$.
Bin 2 is empty.

*If no larger blocks exist, ask the OS to allocate more memory.

Program Segments



high address

virtual memory

low address

stack

heap

bss

data

text

dynamically allocated

initialized to 0 at program start

read from disk

code

How Virtual is Virtual Memory?

Q. Since a 64-bit address space takes over 8 years to write at a rate of 64 gigabytes per second (GDDR6 technology), we effectively never run out of virtual memory. So, why not just allocate increasing VM addresses and never free?

A. External fragmentation would be horrendous! The performance of the page table would degrade tremendously leading to disk thrashing, since all nonzero memory must be backed up on disk in page-sized blocks.

Goal of storage allocators

Use as little virtual memory as possible, and try to keep the used portions relatively compact.

Analysis of Binned Free Lists

Theorem. Suppose that the maximum amount of heap memory in use at any time by a program is M. If the heap is managed by a BFL allocator, the amount of virtual memory consumed by heap storage is $O(M \lg M)$.

Proof. An allocation request for a block of size x consumes $2^{\lceil \lg x \rceil} \leqslant 2x$ storage. Thus, the amount of virtual memory devoted to blocks of size 2^k is at most $2M$. Since there are at most $\lg M$ free lists, the theorem holds. ■

⇒ In fact, BFL is 6-competitive with the optimal omniscient allocator (assuming no coalescing).

Coalescing

Binned free lists can sometimes be heuristically improved by splicing together adjacent small blocks into a larger block.

- Clever schemes exist for finding adjacent blocks efficiently — e.g., the "buddy" system — but the overhead is still greater than simple BFL.
- No good theoretical bounds exist that prove the effectiveness of coalescing.
- Coalescing seems to reduce fragmentation in practice, because heap storage often obeys a stack discipline or tends to be deallocated in batches.

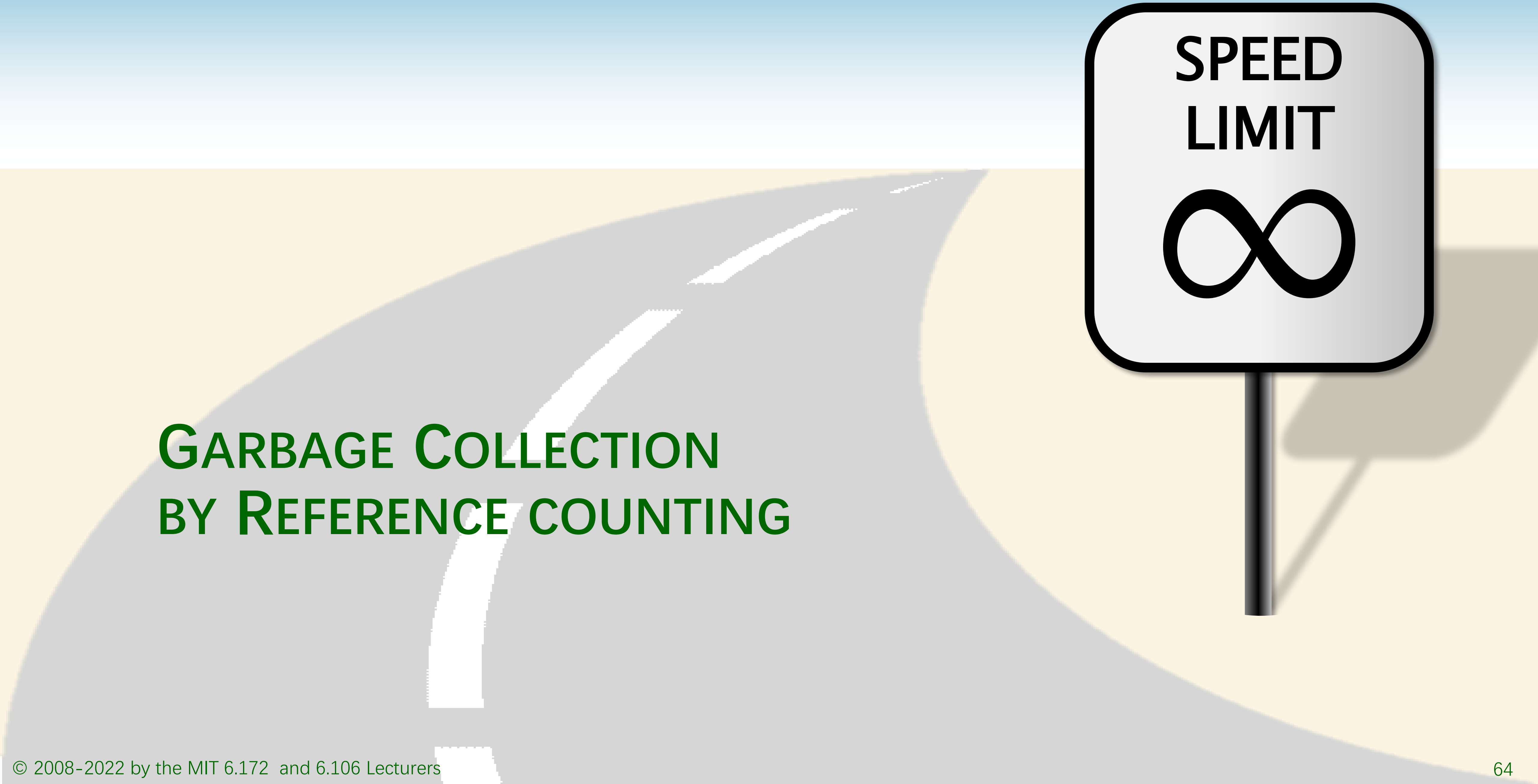
Tradeoff in Page Sizes

Can use either 4KB vs 2MB vs 1GB

- 4K: Little internal fragmentation, But TLB can get overwhelmed
  - 4KB $*$ 2048 = 8MB before running out of TLB entries
- 2MB:···
- 1GB: Efficient use of TLB, but can result in a lot of internal fragmentation
  - Good for applications that have a very large memory footprint
  - 1GB $*$ 1024 = 1TB before running out of TLB entries

Summary

	Manual
Ease of Use	Bad
Throughput	Good
Latency	Good
External Fragmentation	Bad
Example	C `malloc/free`



GARBAGE COLLECTION BY REFERENCE COUNTING

Garbage Collectors

Idea

- Free the programmer from freeing objects.
- A garbage collector identifies and recycles the objects that the program can no longer access.
- GC can be built-in (Python, Java, Julia) or do-it-yourself.

Garbage Collection

Terminology

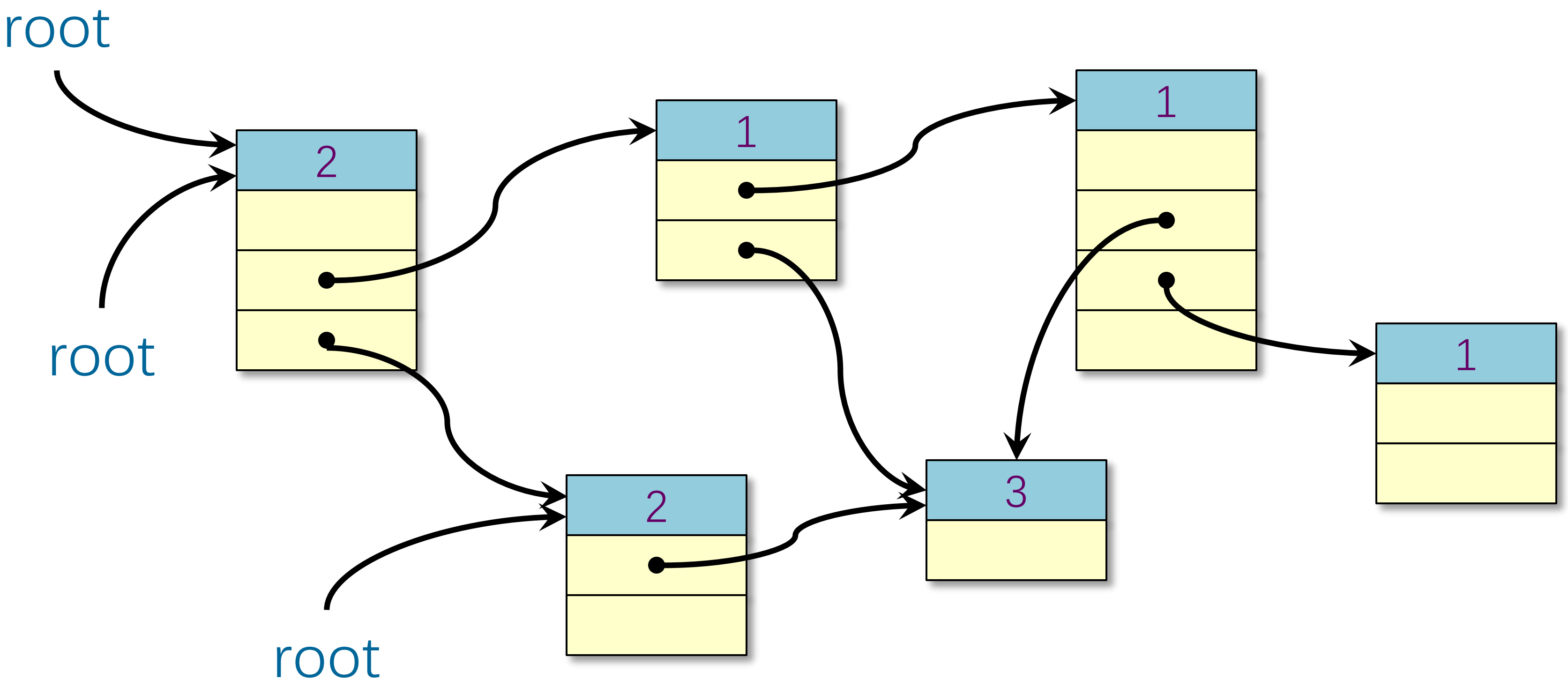
- Roots are objects directly accessible by the program (globals, stack, etc.).
- Live objects are reachable from the roots by following pointers.
- Dead objects are inaccessible and can be recycled.

How can the GC identify pointers?

- Strong typing — types are known at compile time (or at runtime with JIT).
- Prohibit pointer arithmetic (which may slow down some programs).

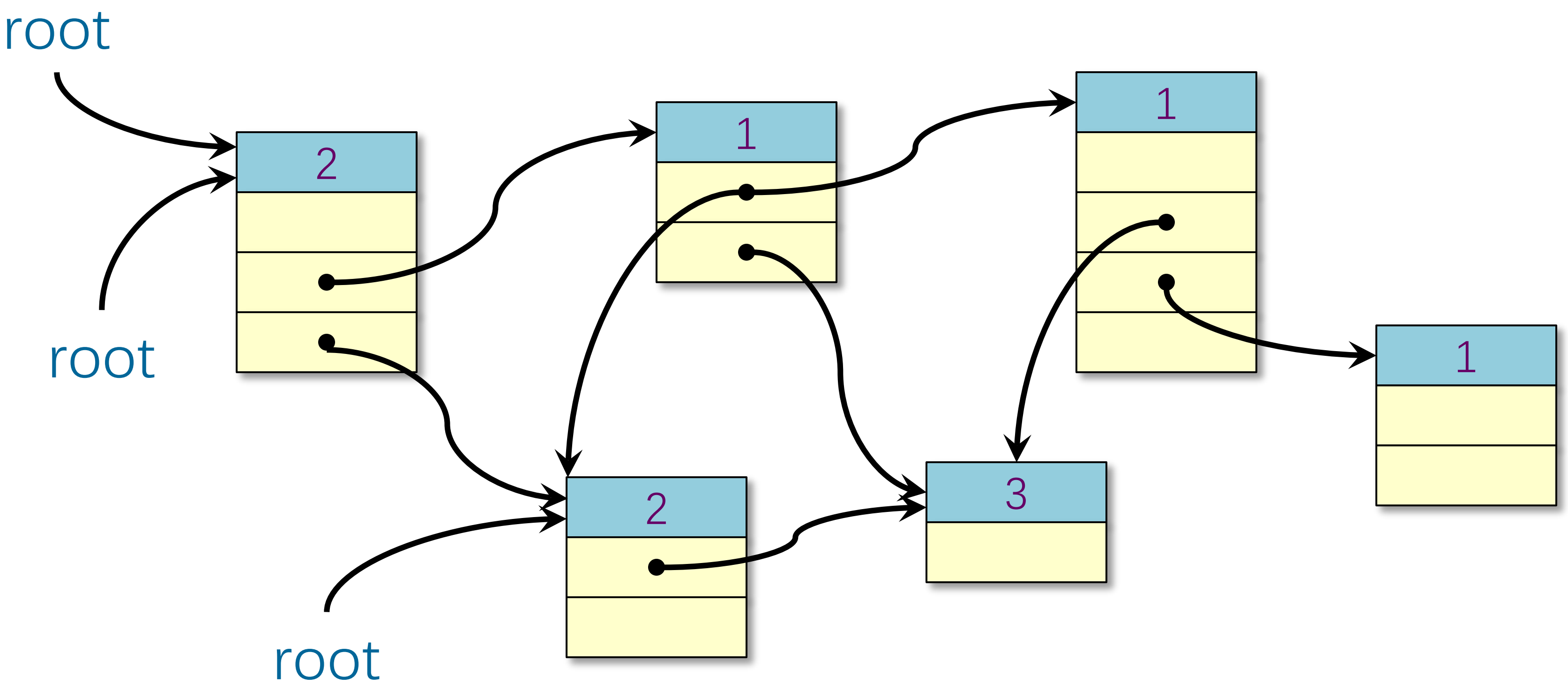
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.



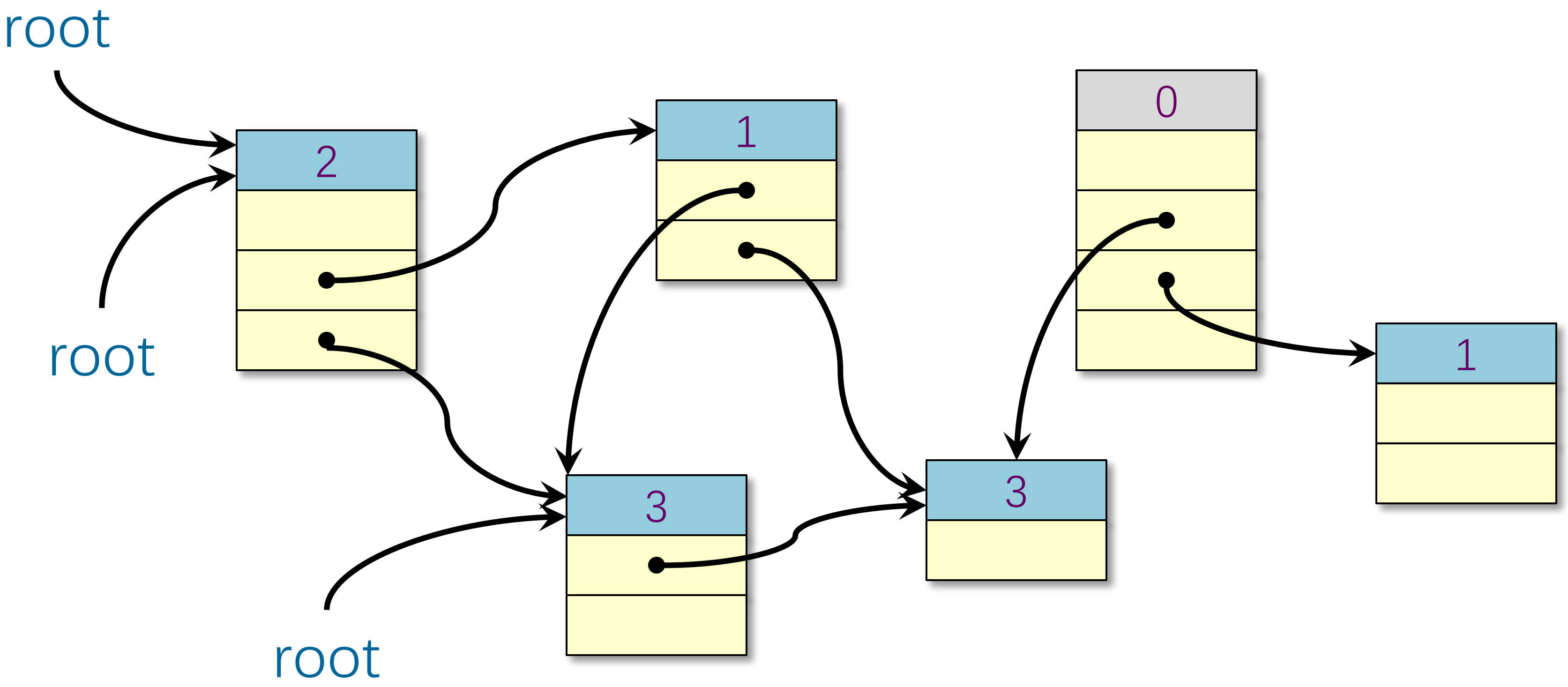
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.



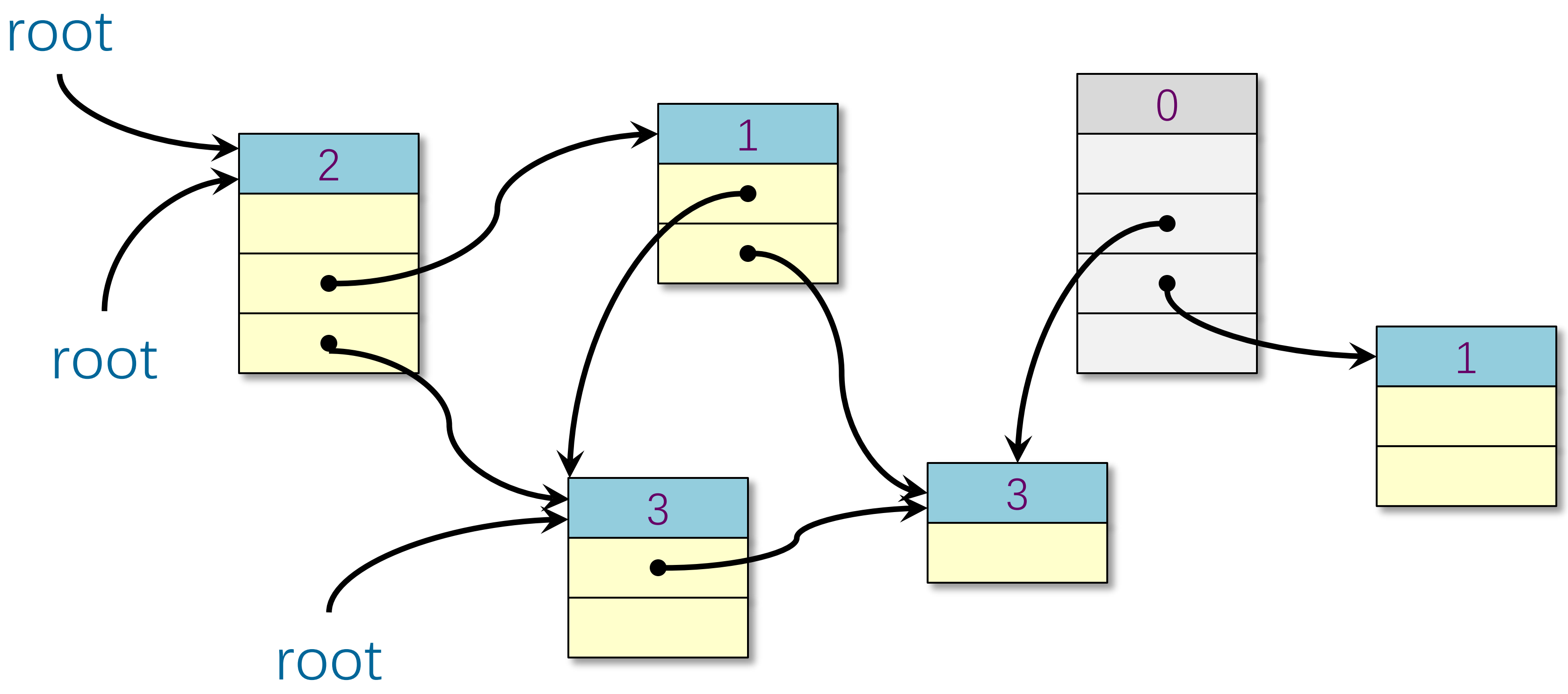
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.



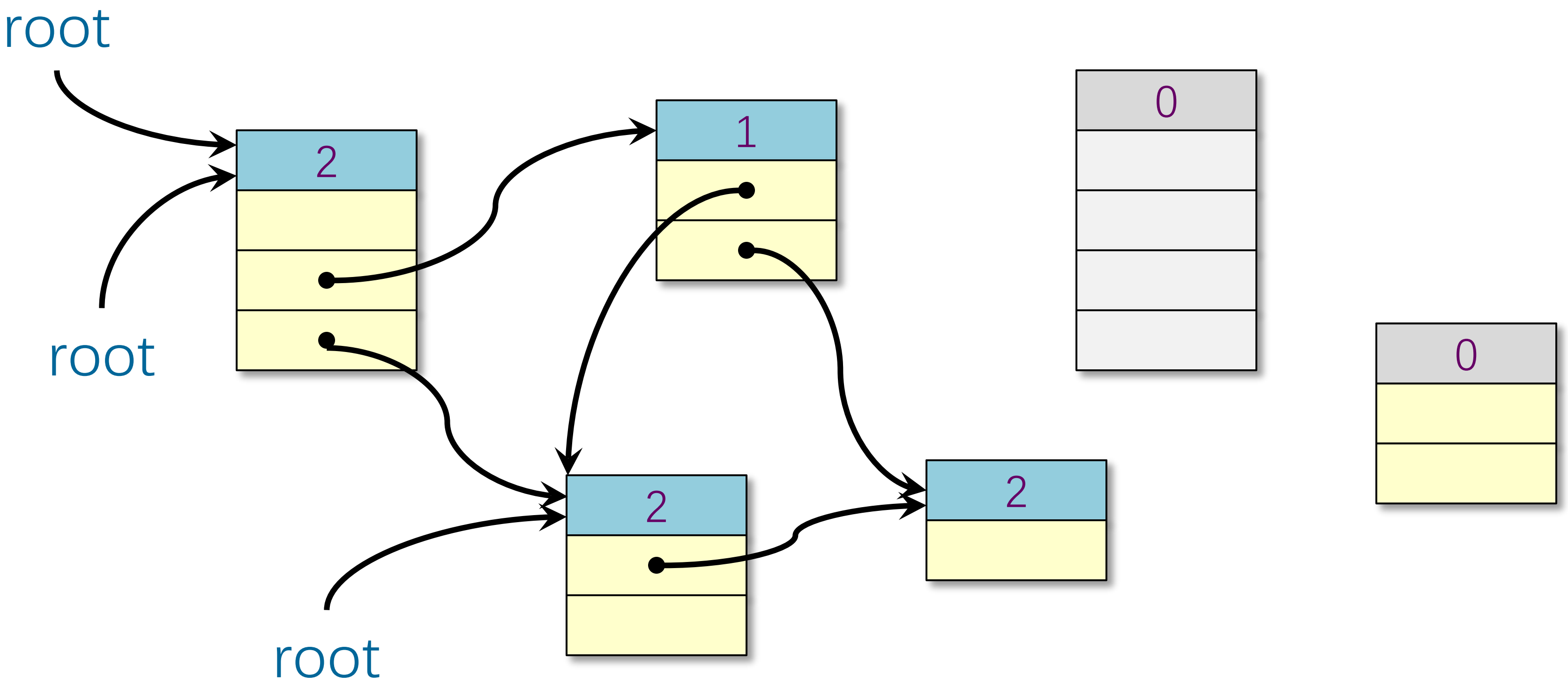
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.



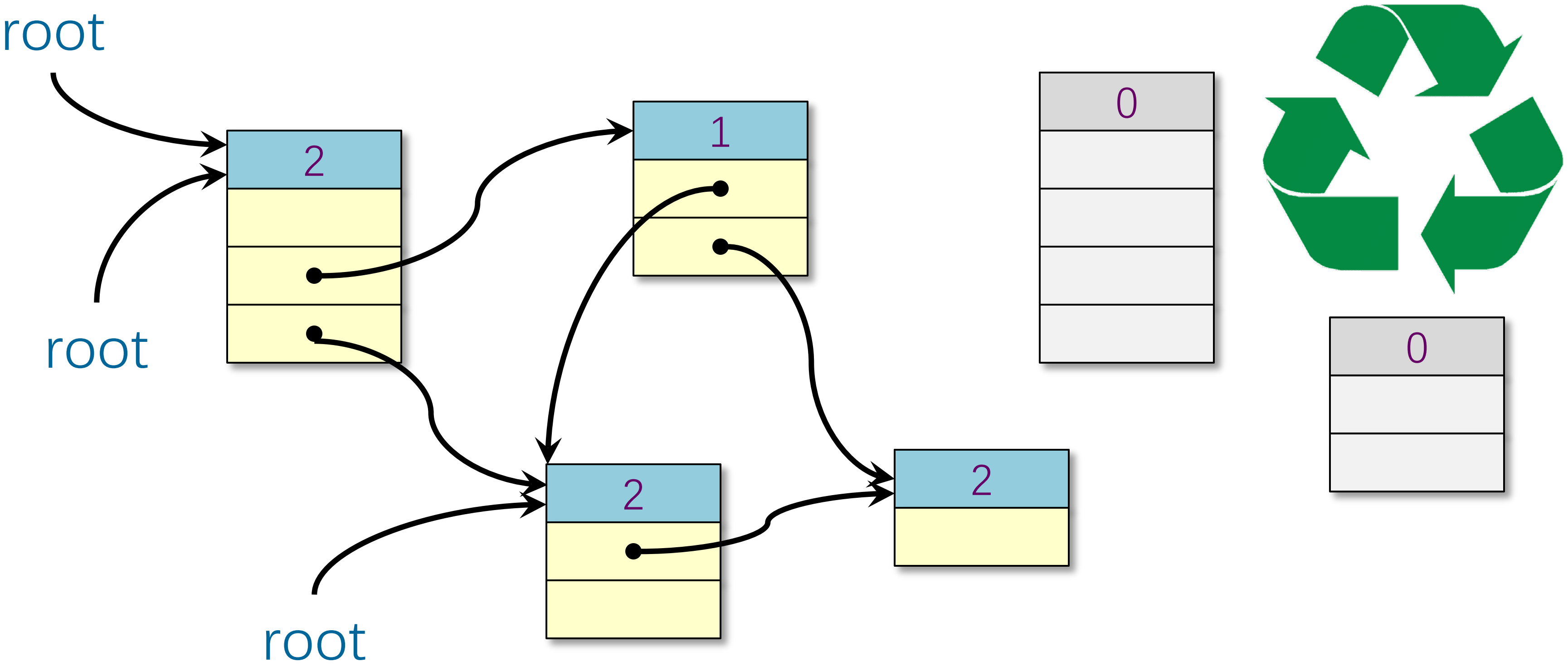
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.



Reference Counting

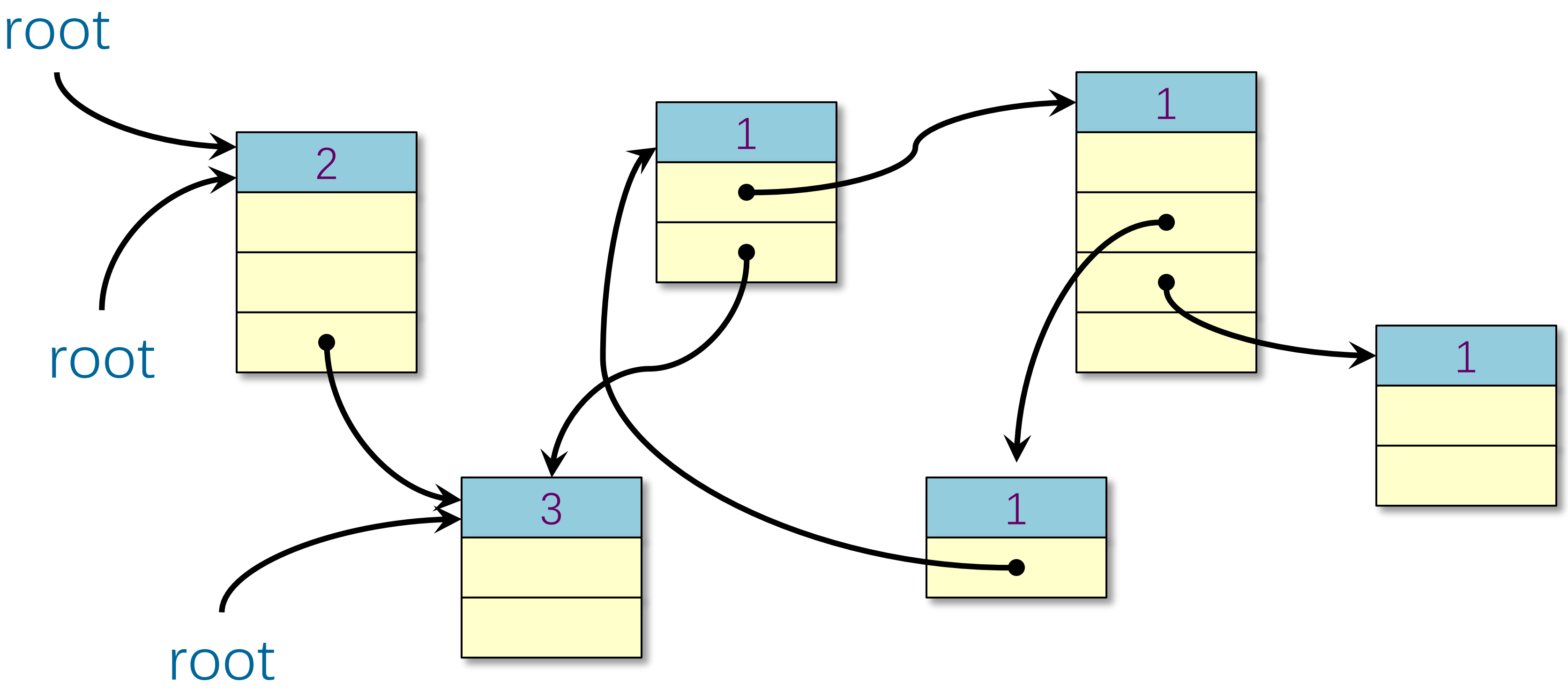
Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.



Limitation of Reference Counting

Problem

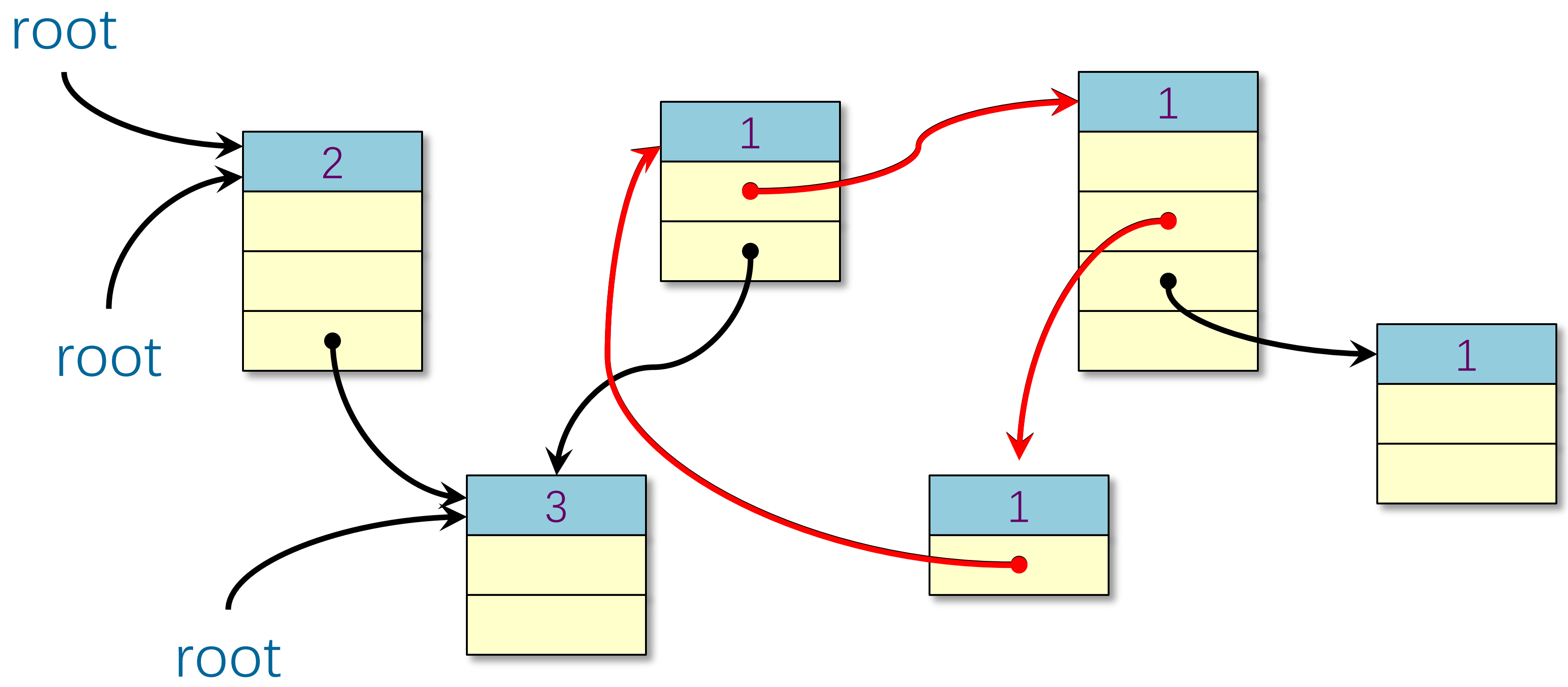
A cycle is never garbage collected!



Limitation of Reference Counting

Problem

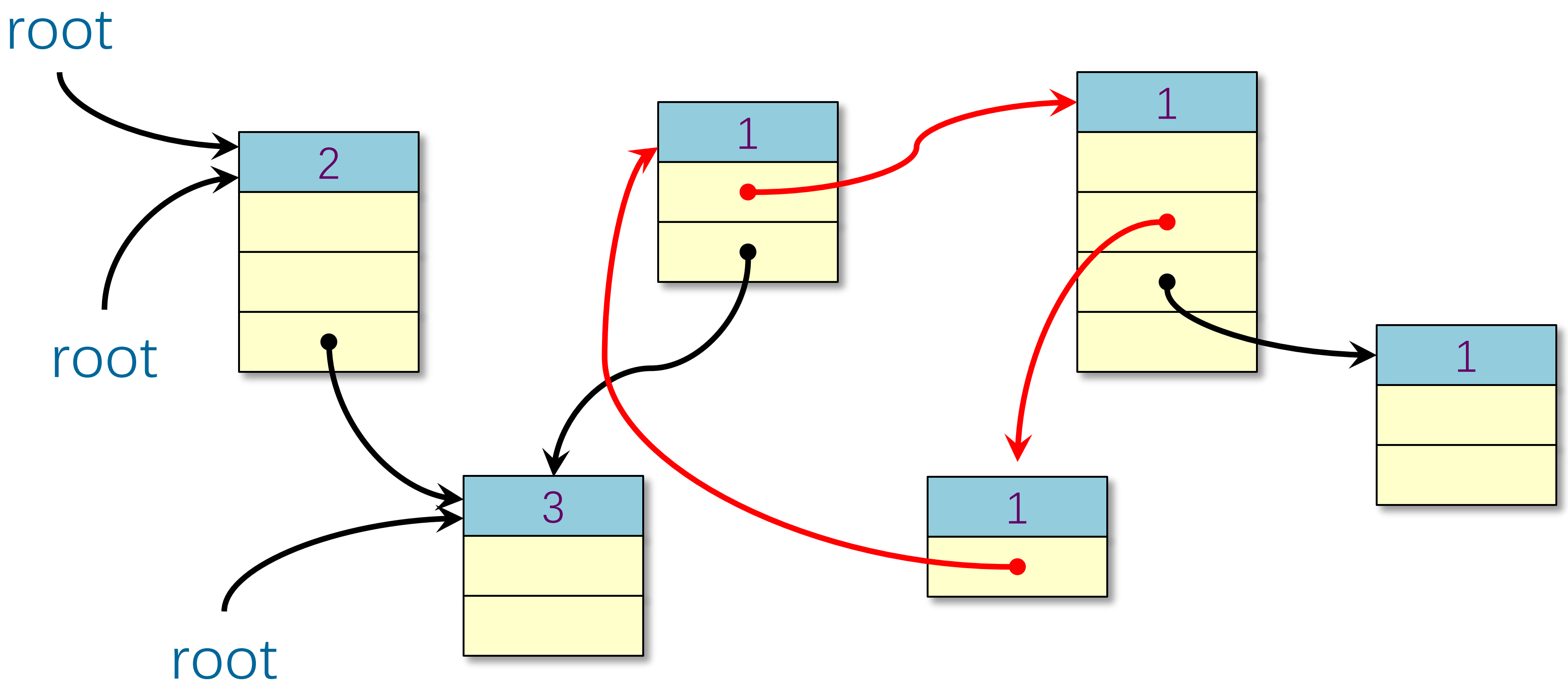
A cycle is never garbage collected!



Limitation of Reference Counting

Problem

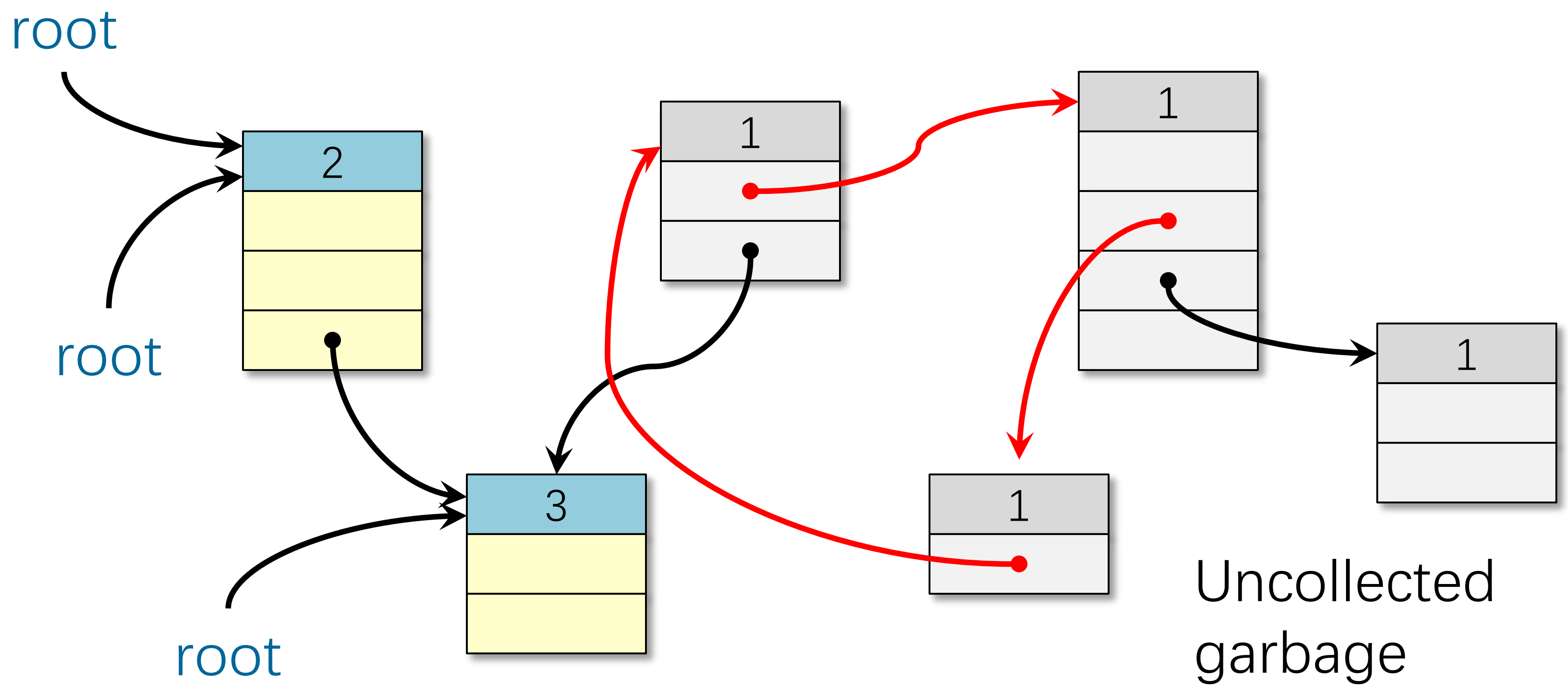
A cycle is never garbage collected!



Limitation of Reference Counting

Problem

A cycle is never garbage collected!

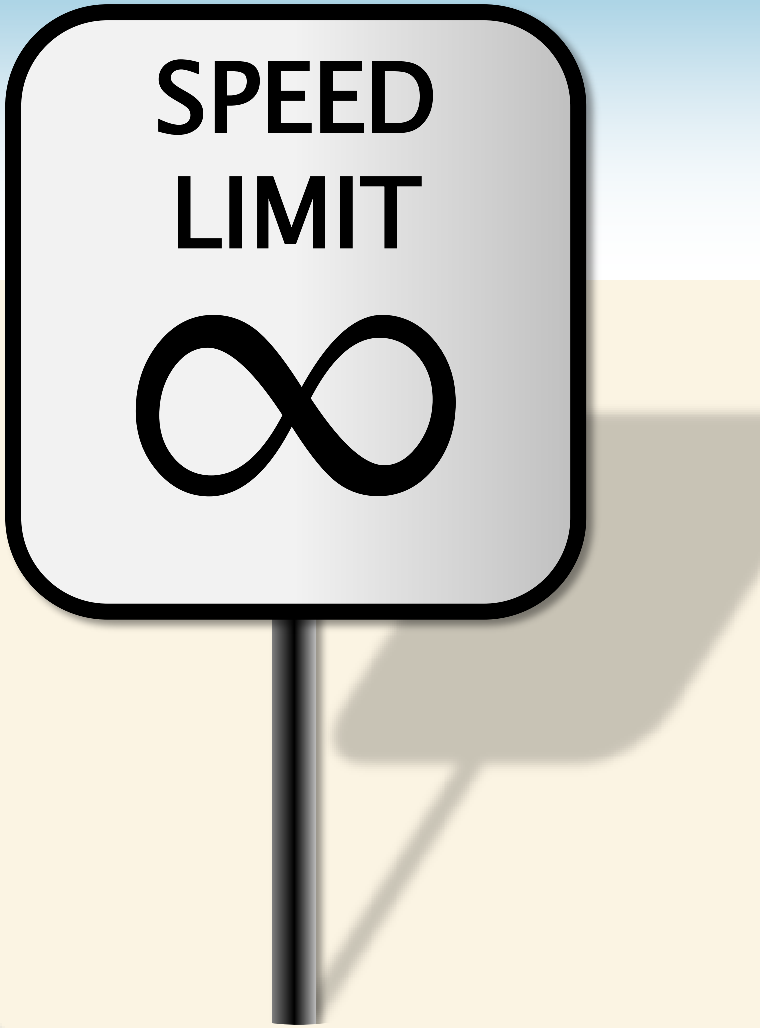


Uncollected garbage stinks!

Nevertheless, reference counting works well for acyclic structures.

Summary

	Manual	Reference Counting
Ease of Use	Bad	Medium
Throughput	Good	Medium
Latency	Good	Good
External Fragmentation	Bad	Bad
Example	C `malloc/free`	C++ `std::shared_ptr`



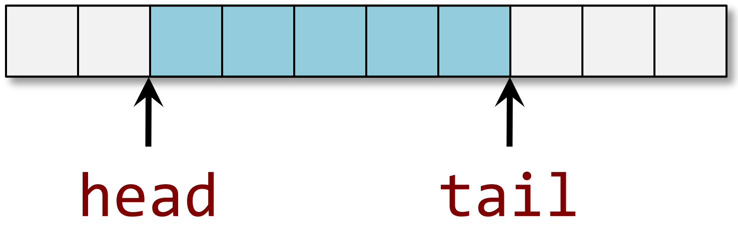
MARK-AND-SWEEP GARBAGE COLLECTION

Graph Abstraction

Idea

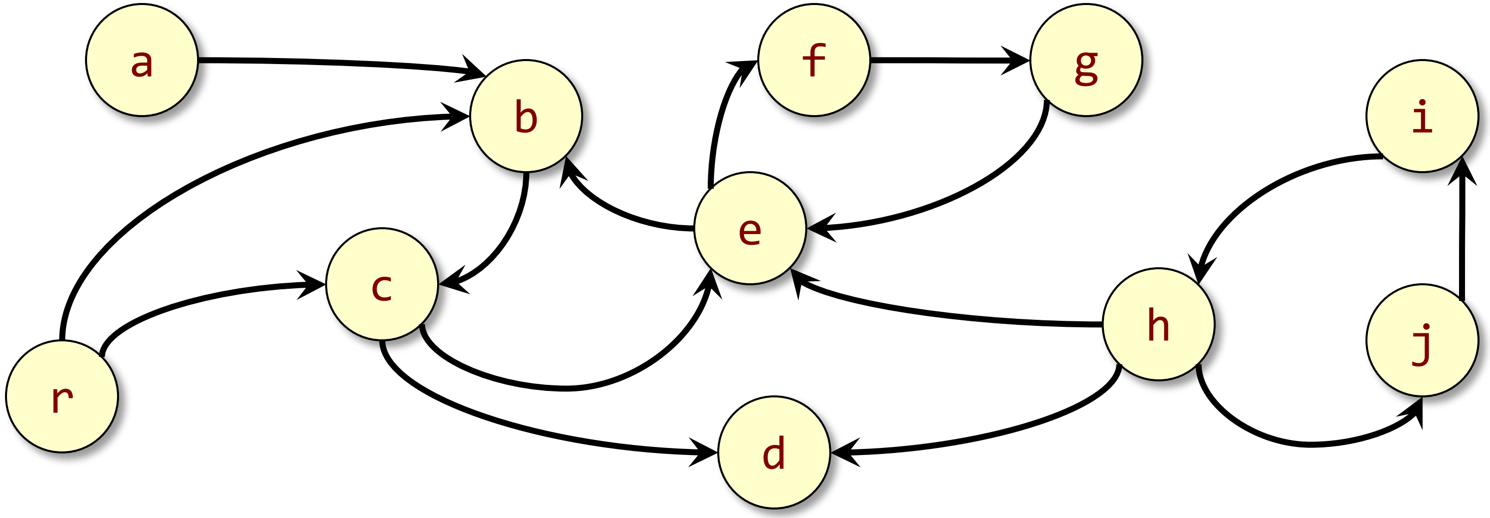
Objects and pointers form a directed graph $G = (V, E)$. Live objects are reachable from the roots. Use breadth-first search to find the live objects.

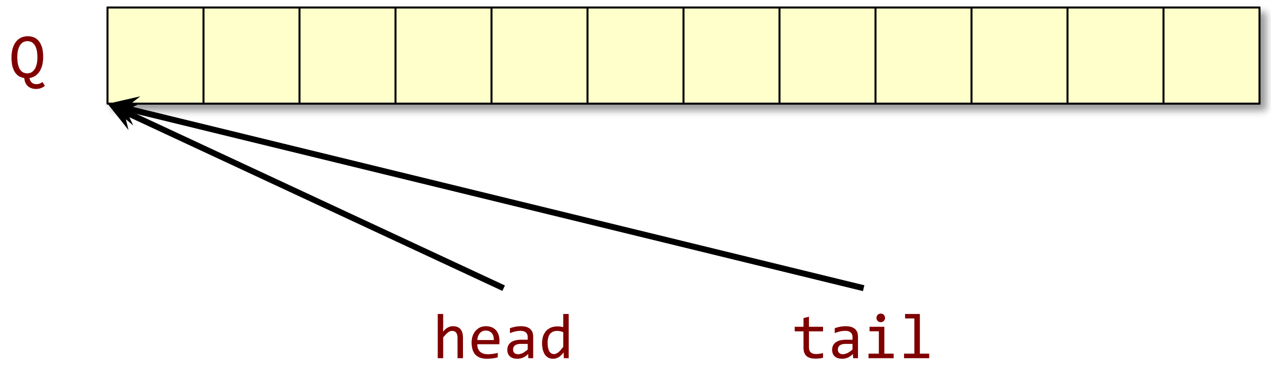
FIFO queue **Q**



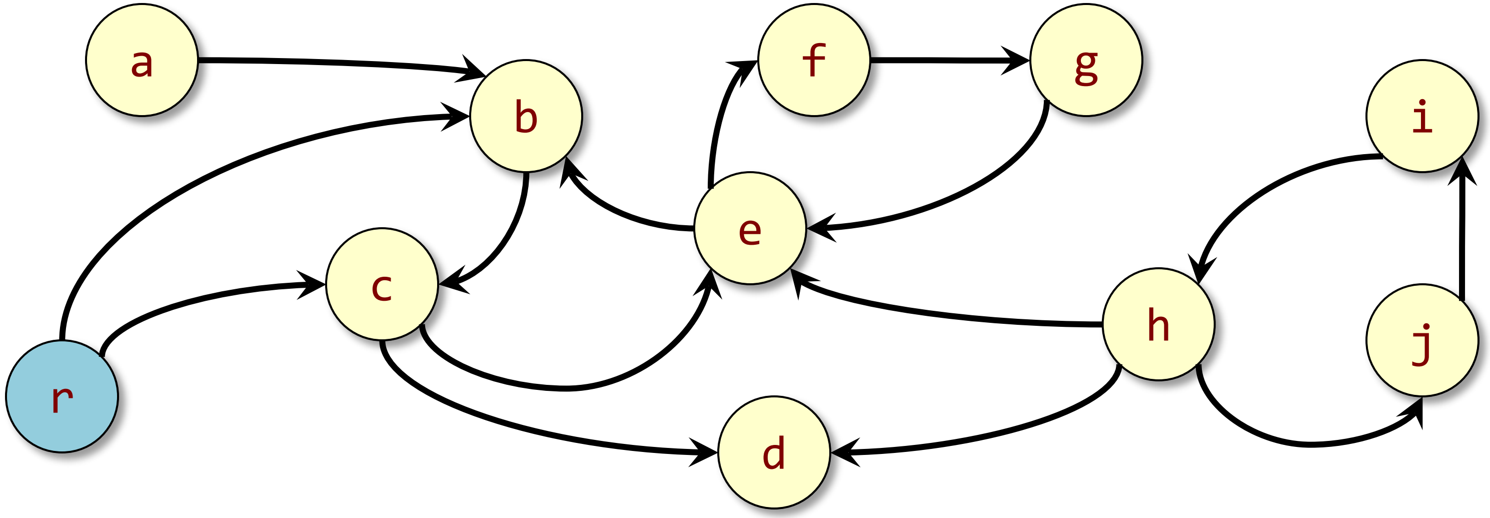
```
for (v ∈ V) {
  if (root(v)) {
    v.mark = 1;
    enqueue(Q, v);
  } else v.mark = 0;

while (Q != ∅) {
  u = dequeue(Q);
  for (v ∈ V such that (u,v) ∈ E)
{
    if (v.mark == 0) {
      v.mark = 1;
      enqueue(Q, v);
    }
  }
}
```

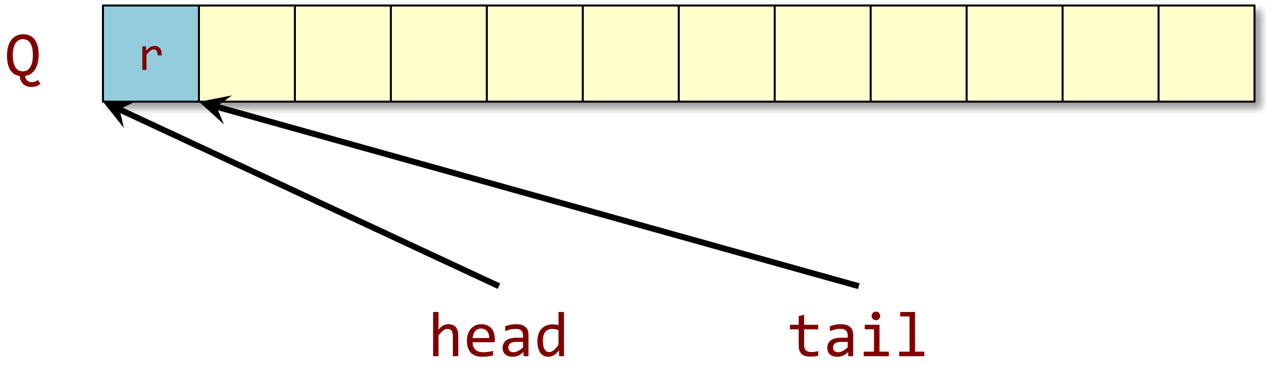
Breadth-First Search



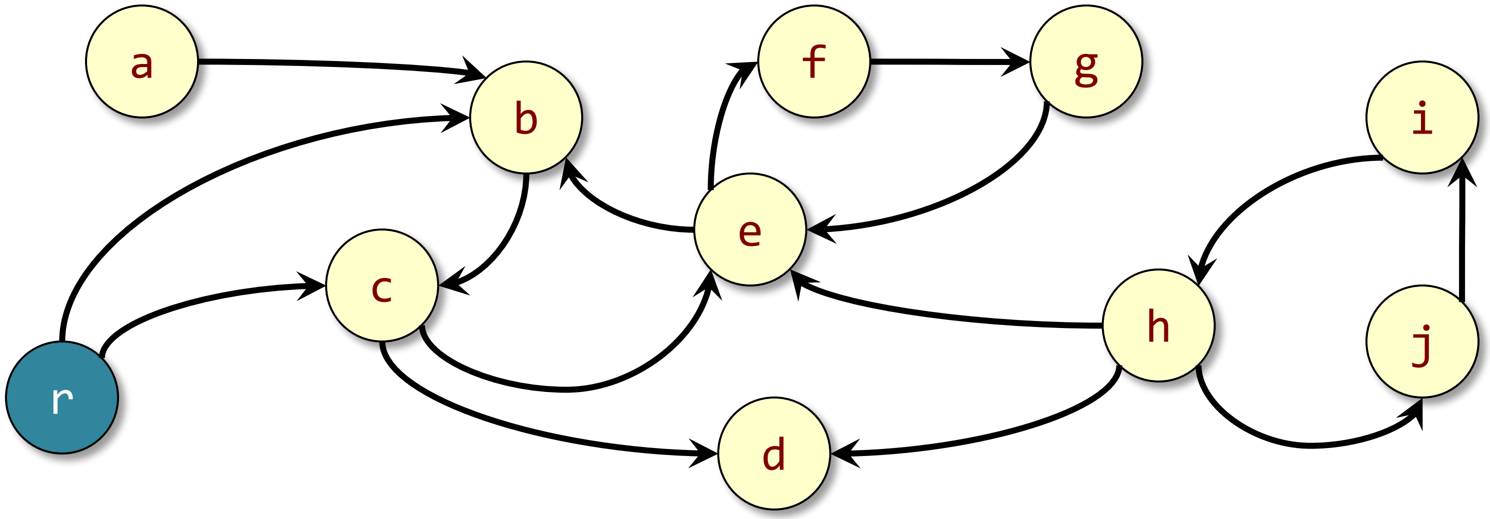


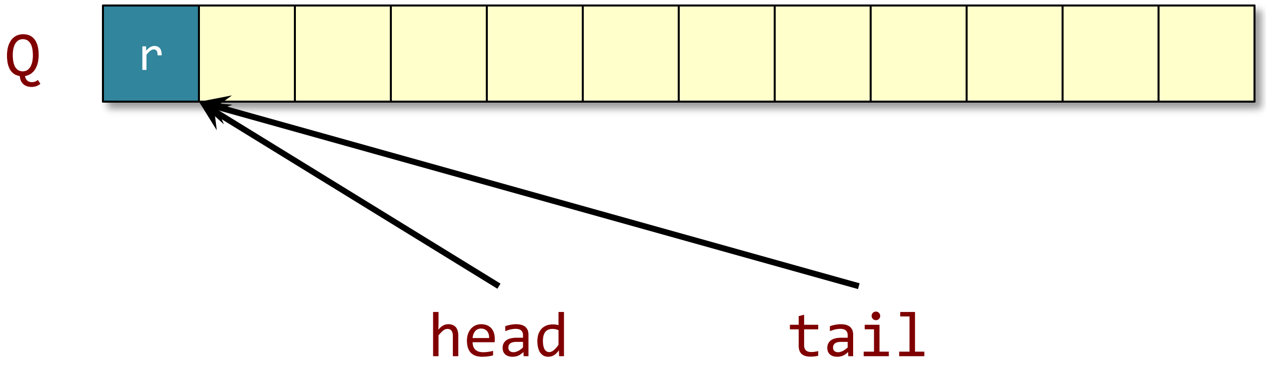
Breadth-First Search



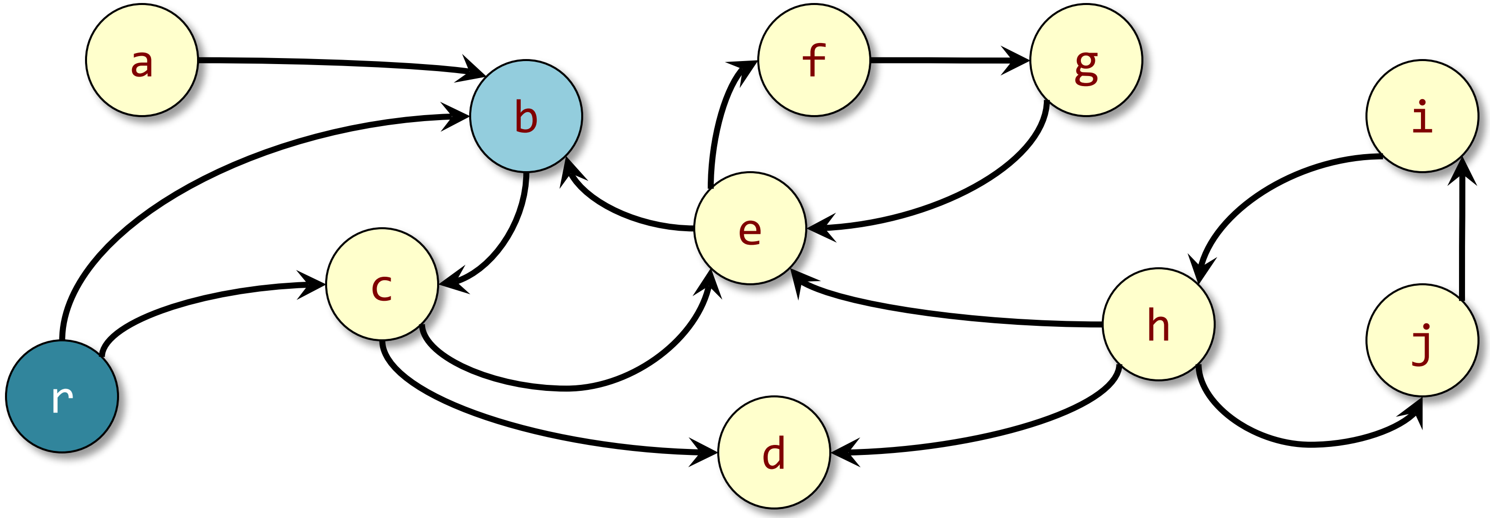


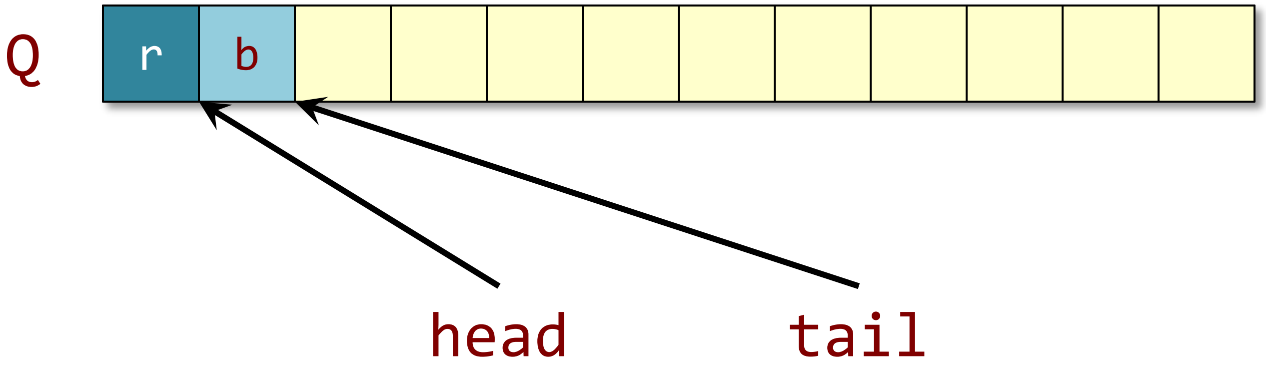
Breadth-First Search



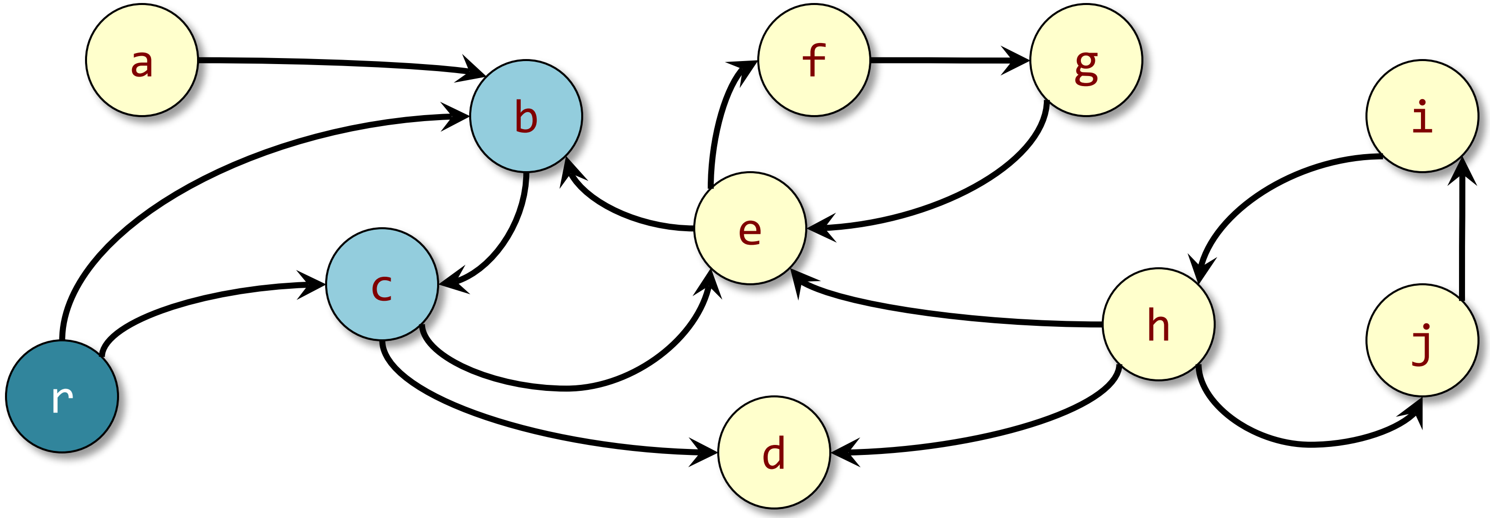


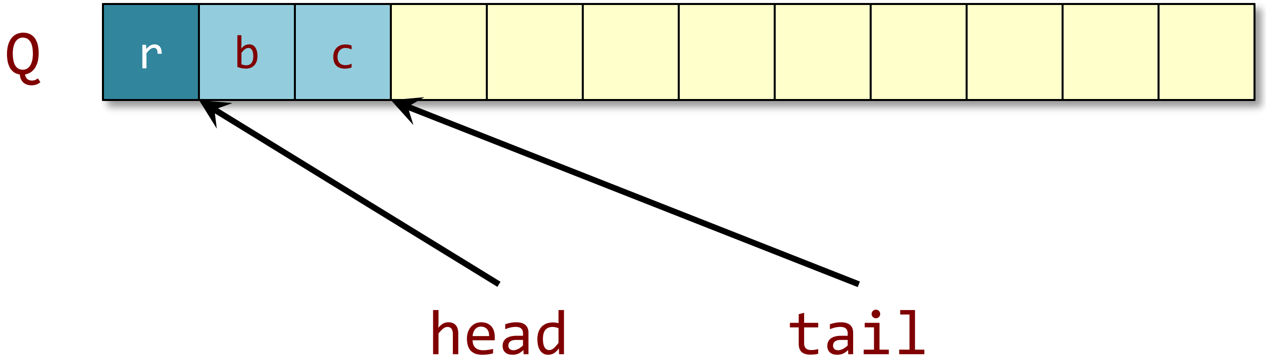
Breadth-First Search



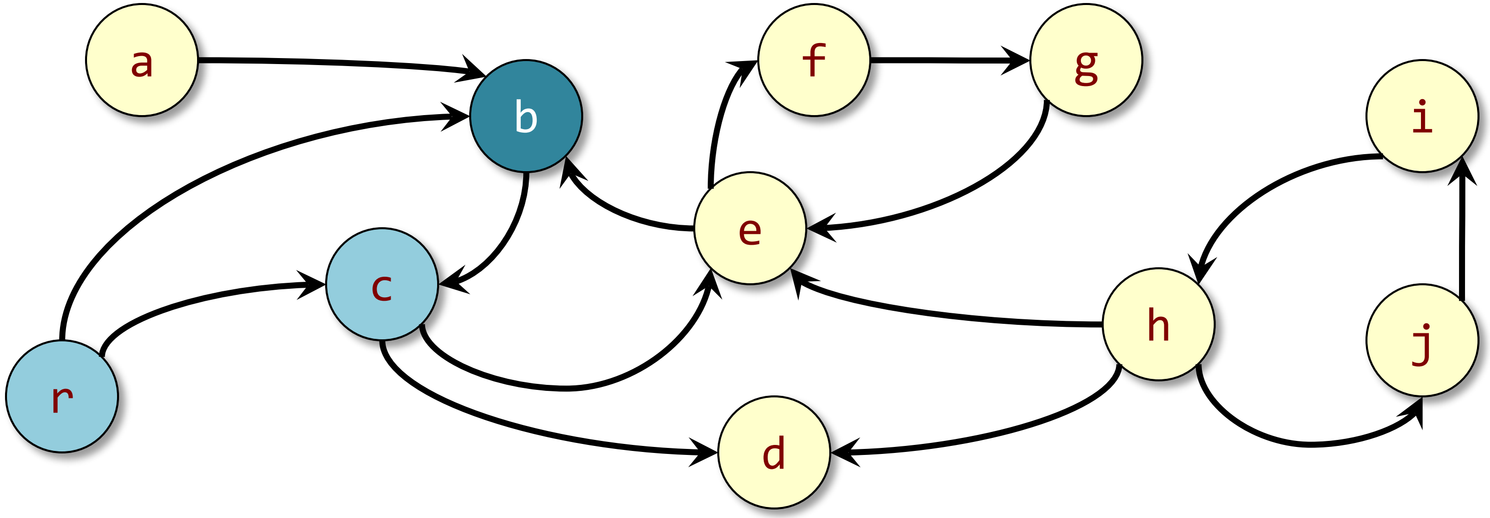


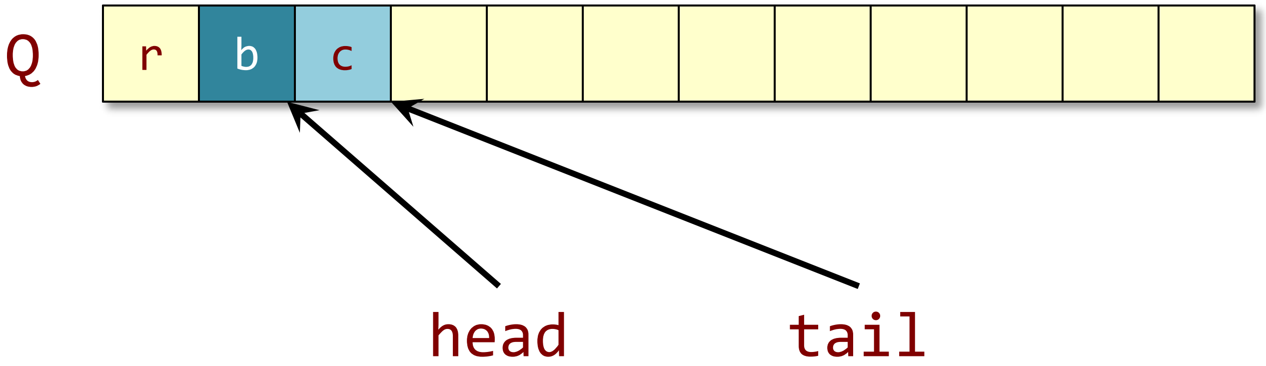
Breadth-First Search



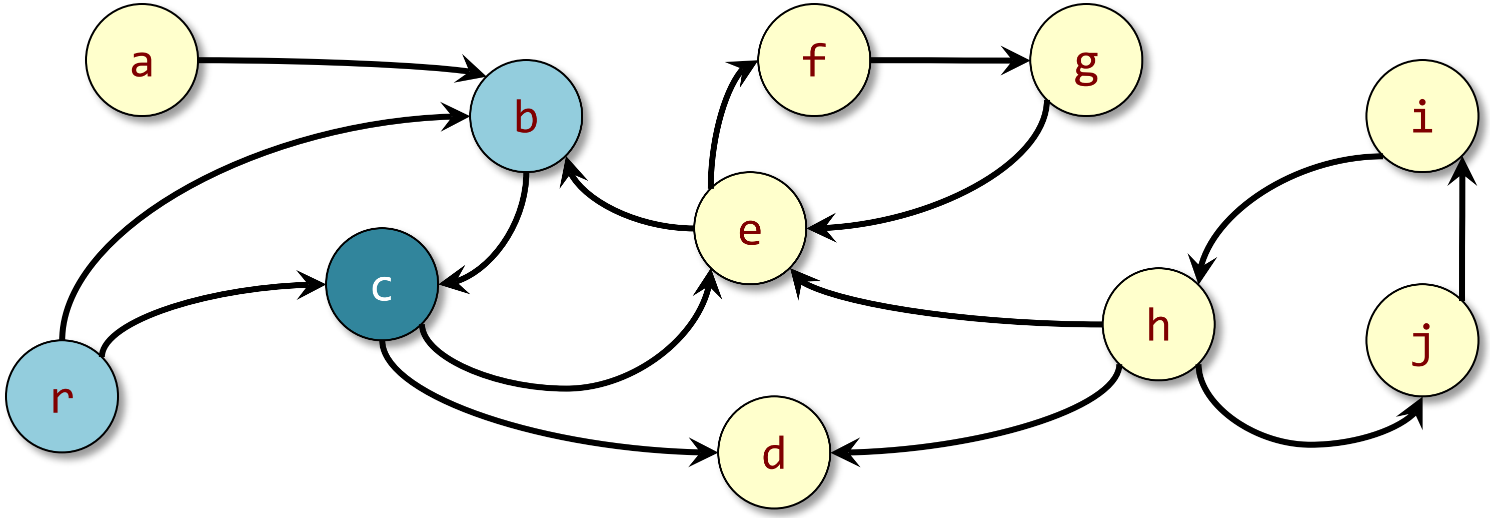


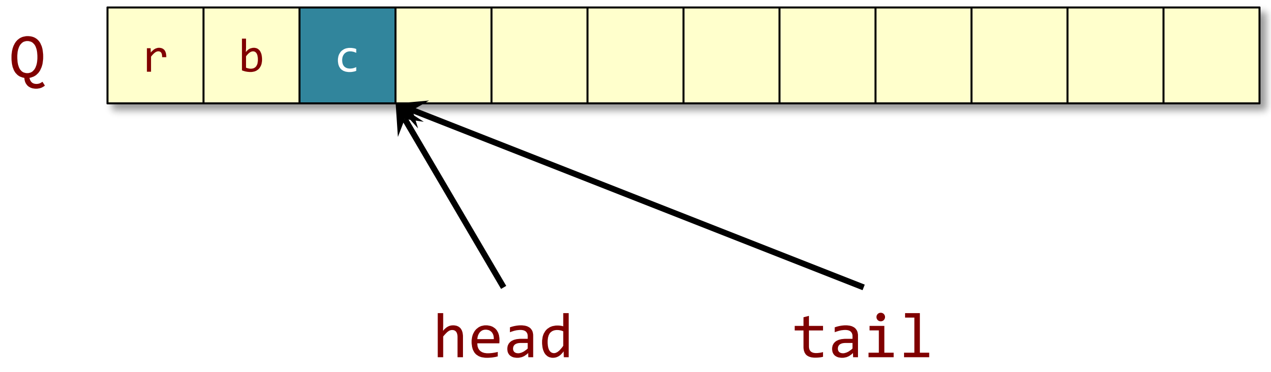
Breadth-First Search



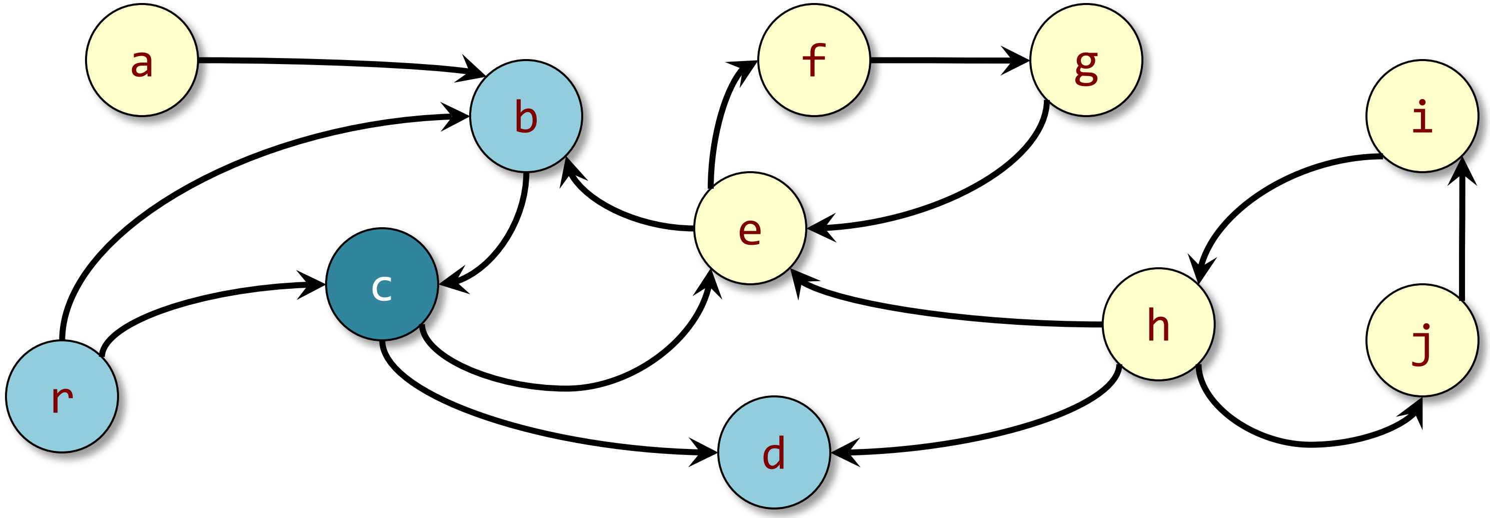


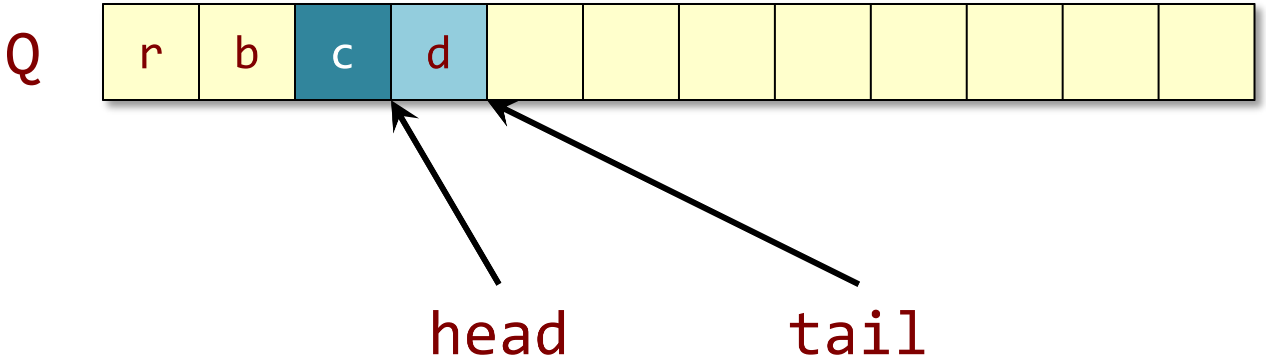
Breadth-First Search



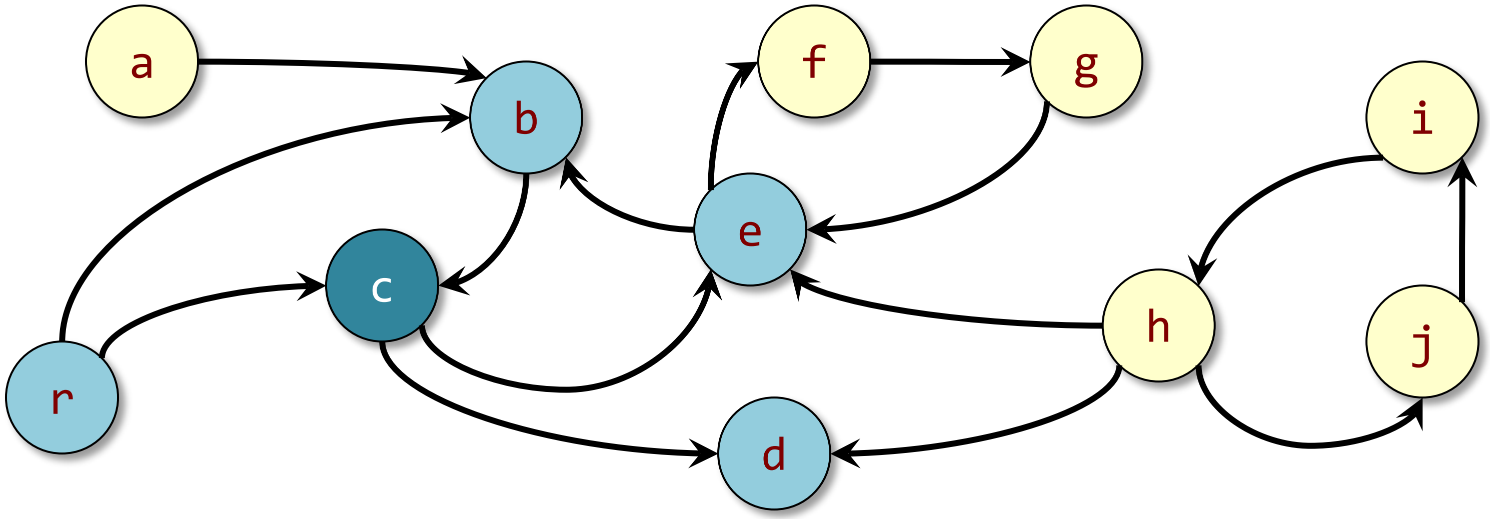


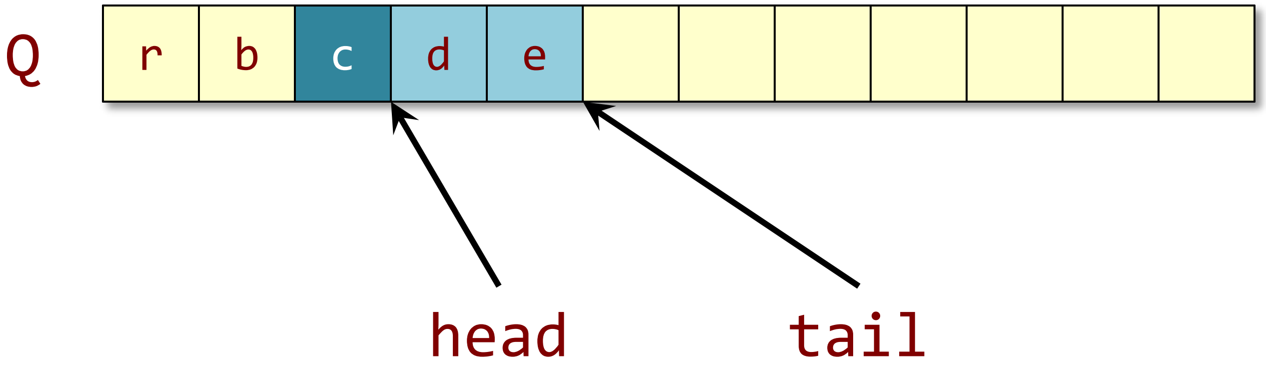
Breadth-First Search



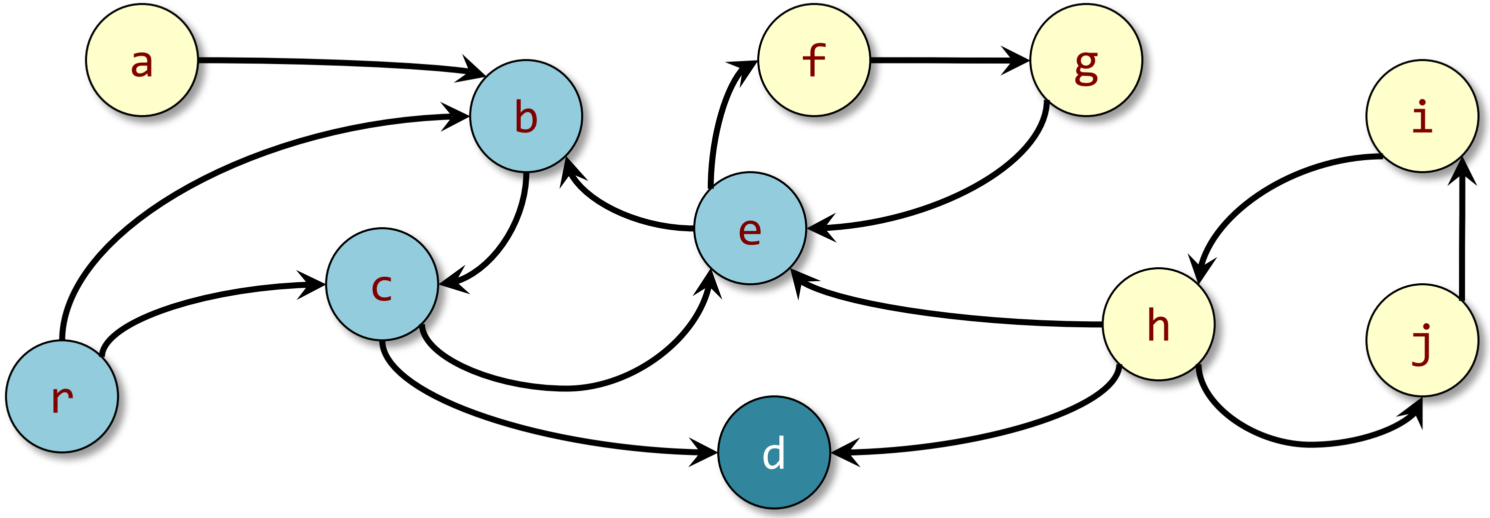


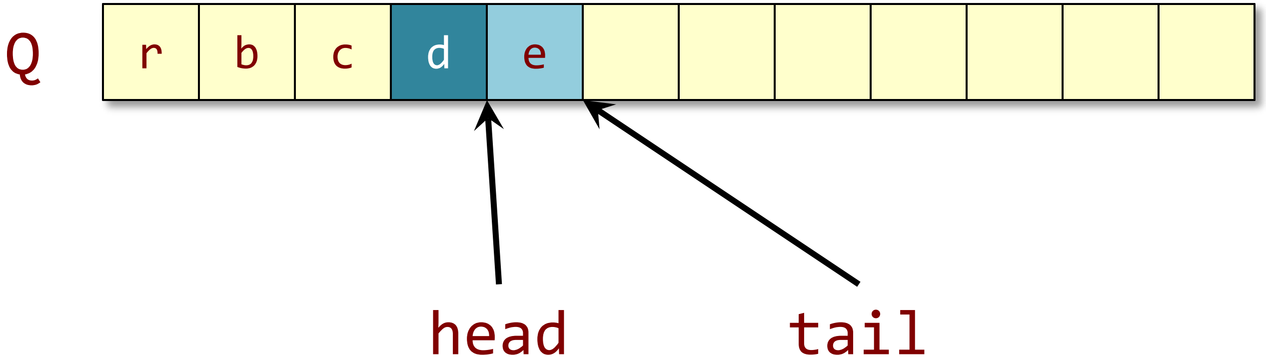
Breadth-First Search



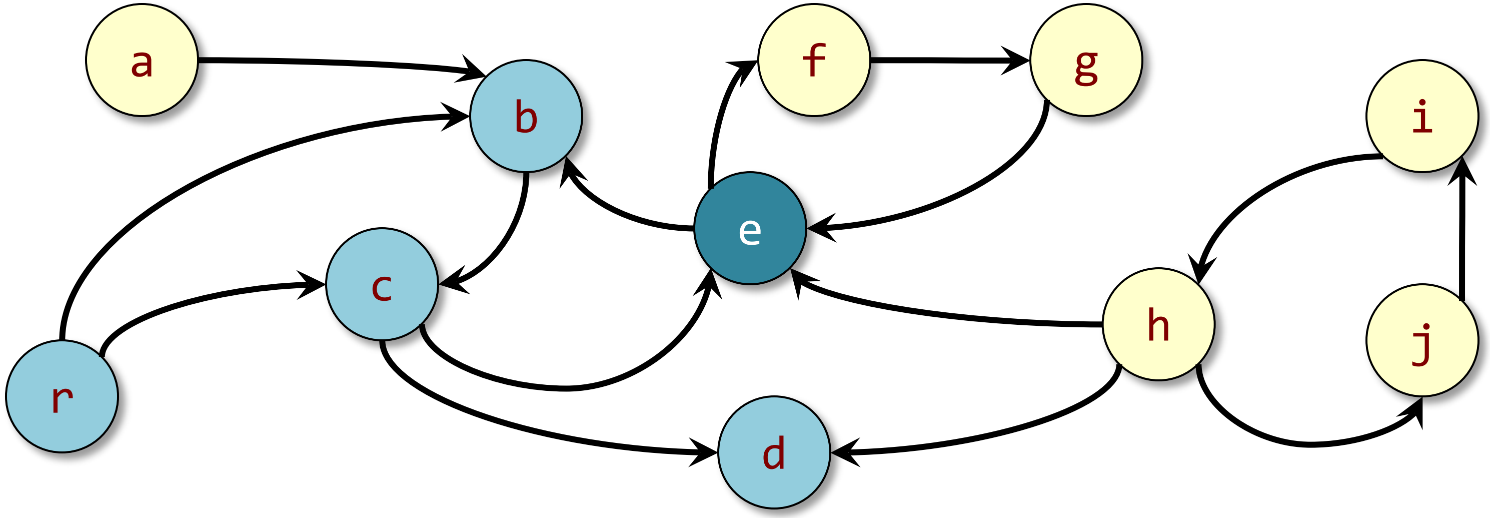


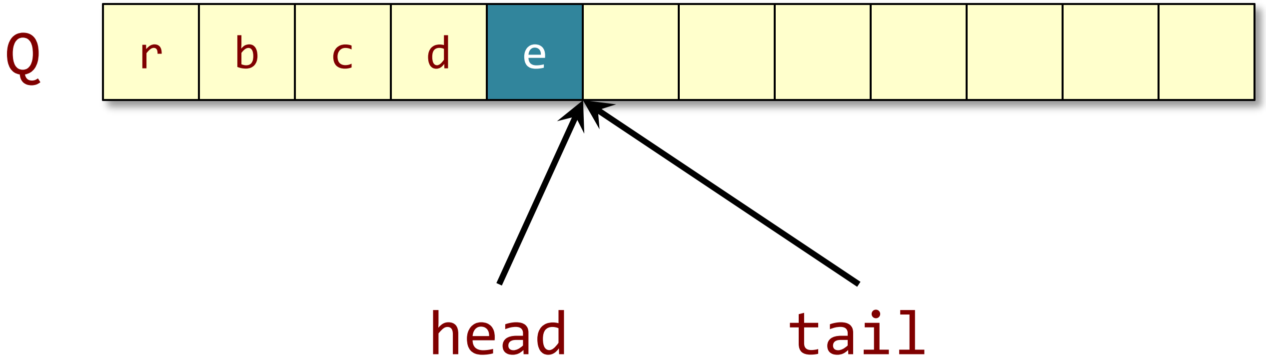
Breadth-First Search



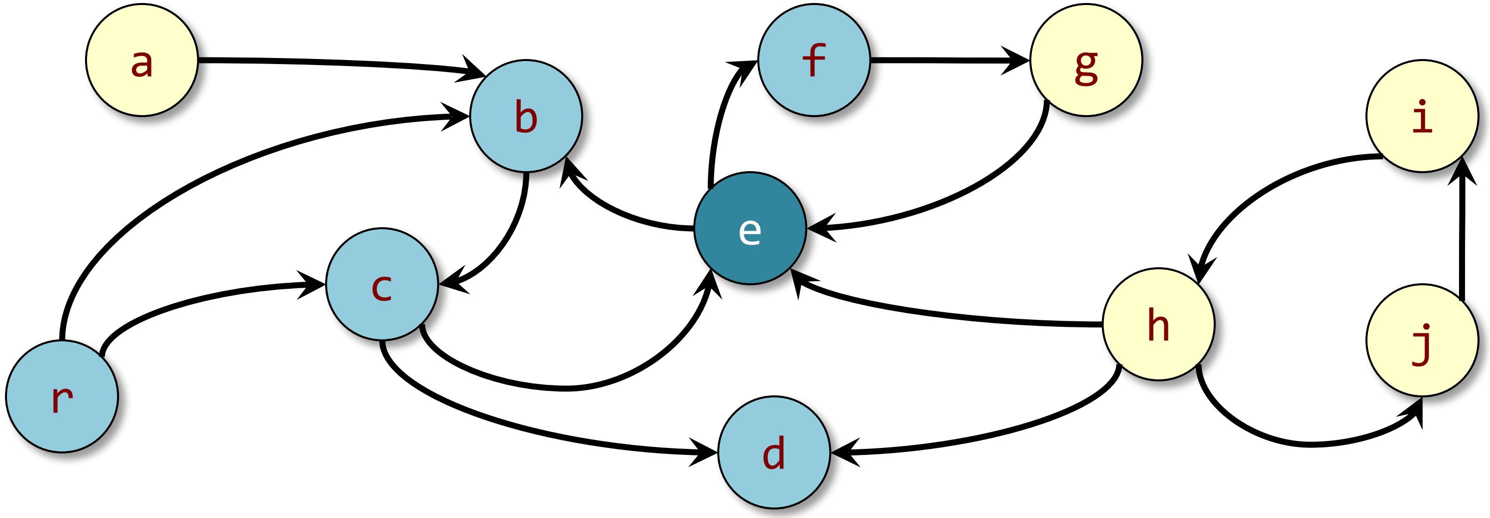


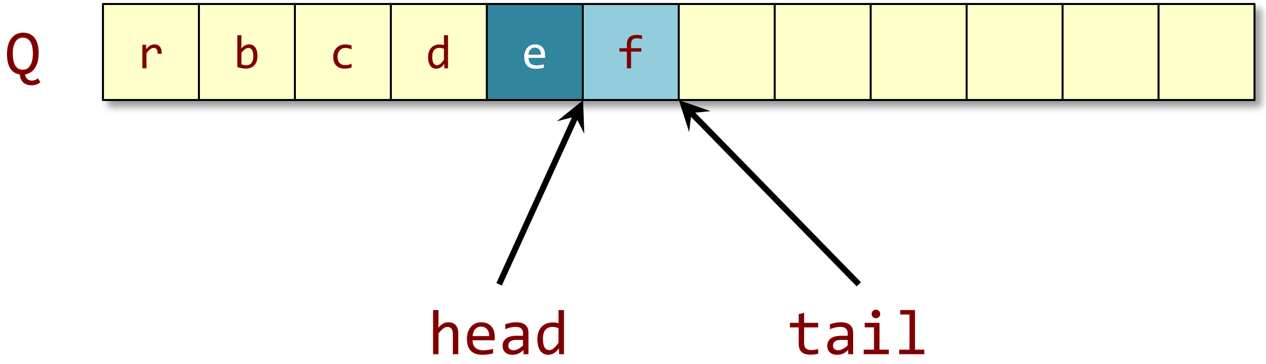
Breadth-First Search



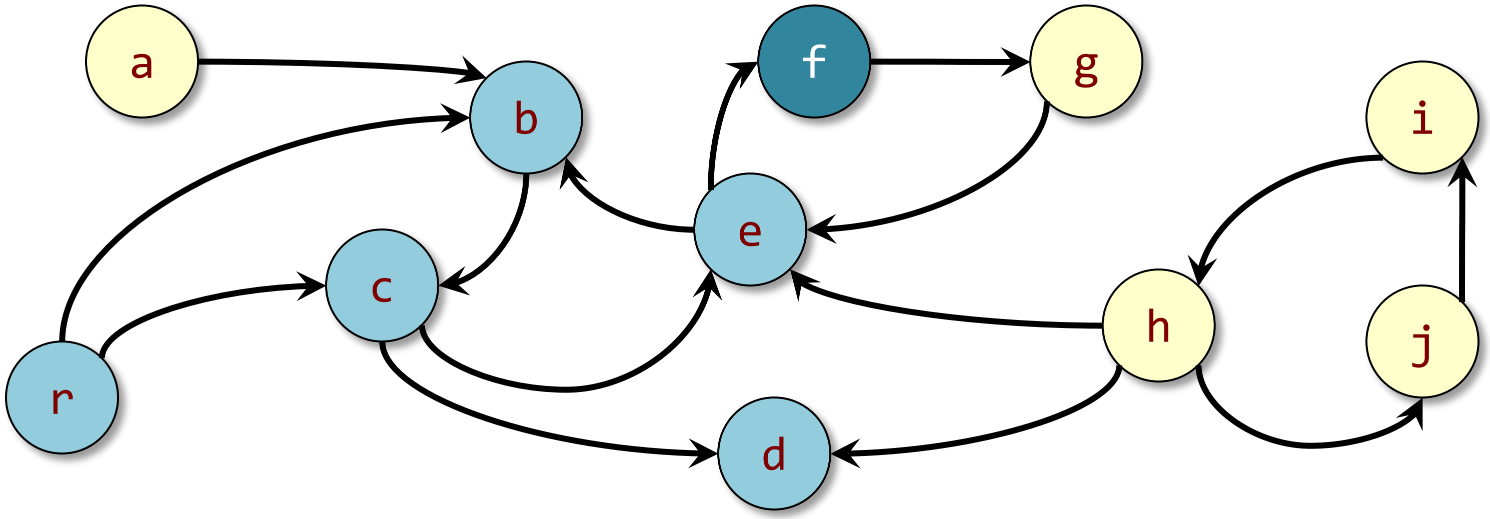


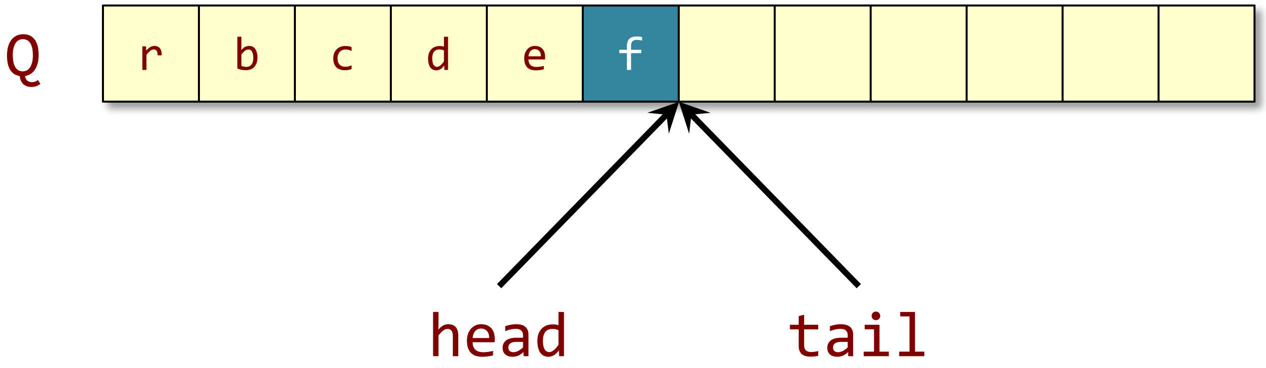
Breadth-First Search



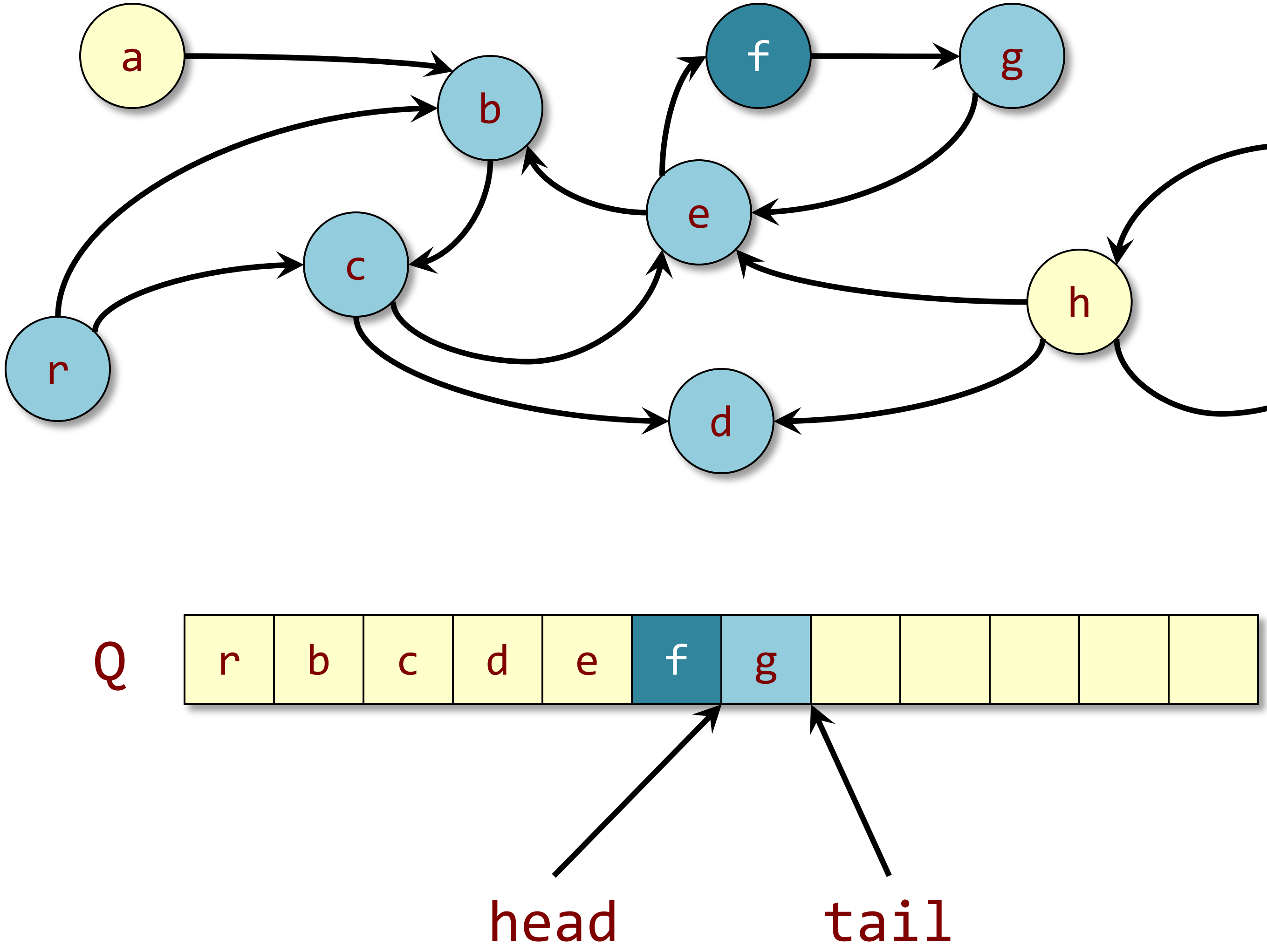


Breadth-First Search

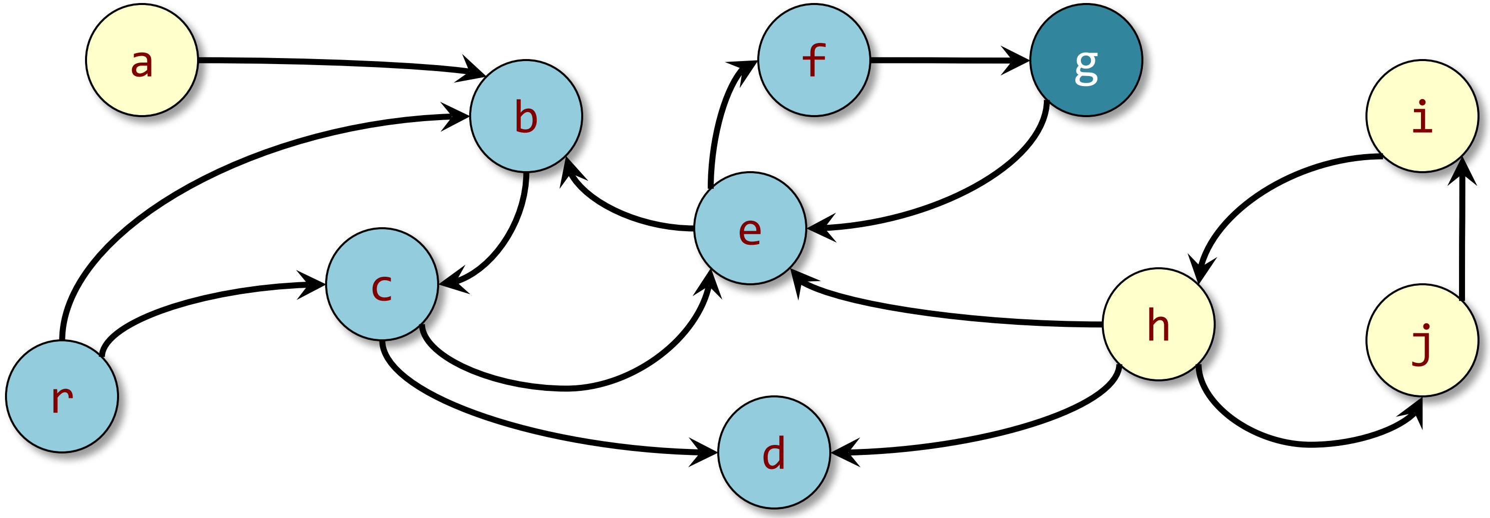


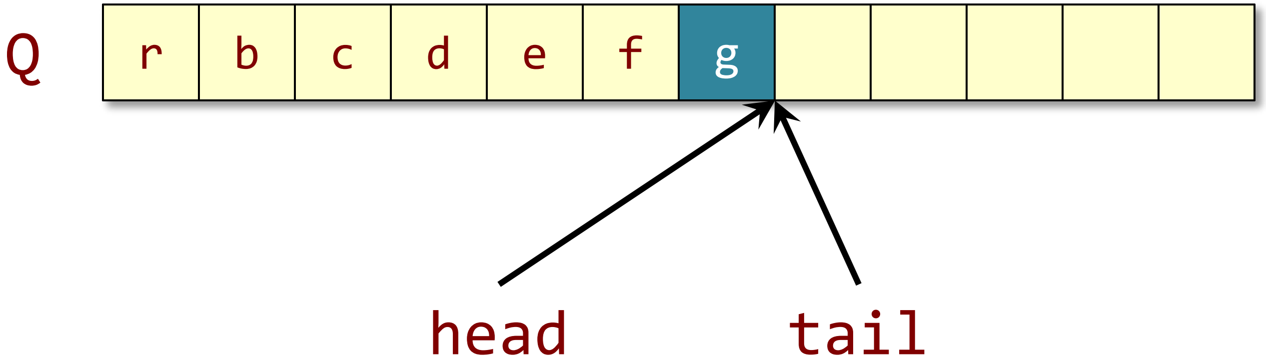


Breadth-First Search



Breadth-First Search





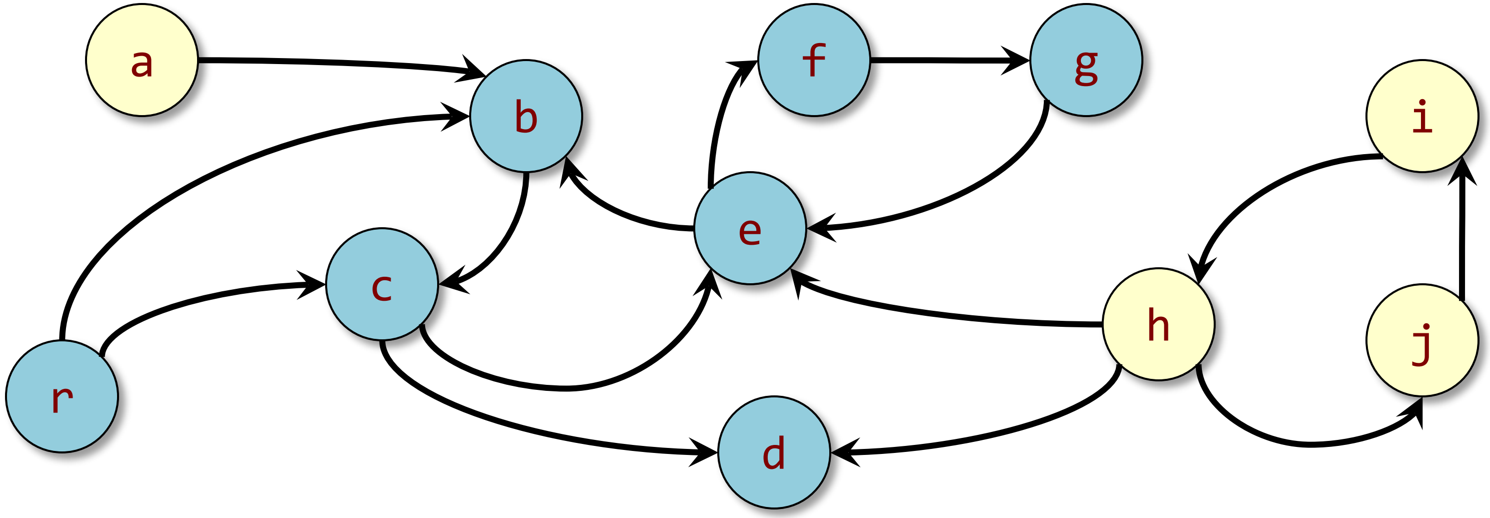
Q

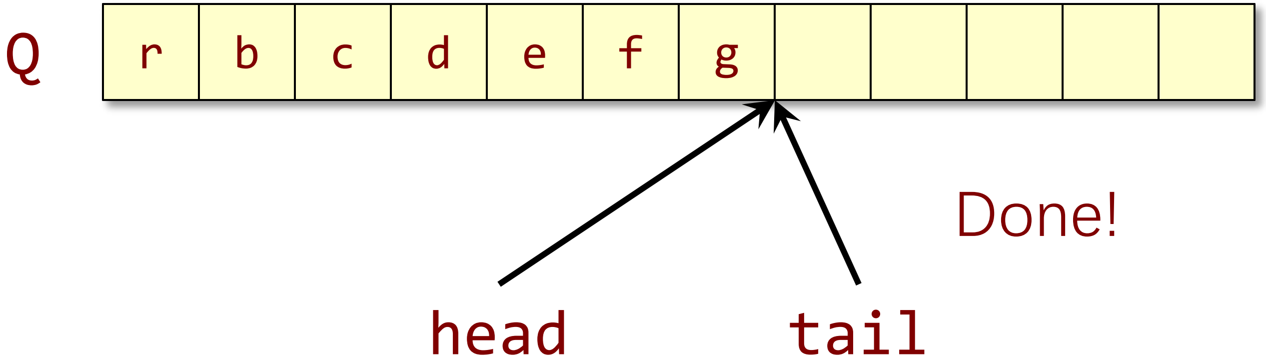
r	b	c	d	e	f	g					

head

tail

Breadth-First Search





Done!

Mark-and-Sweep

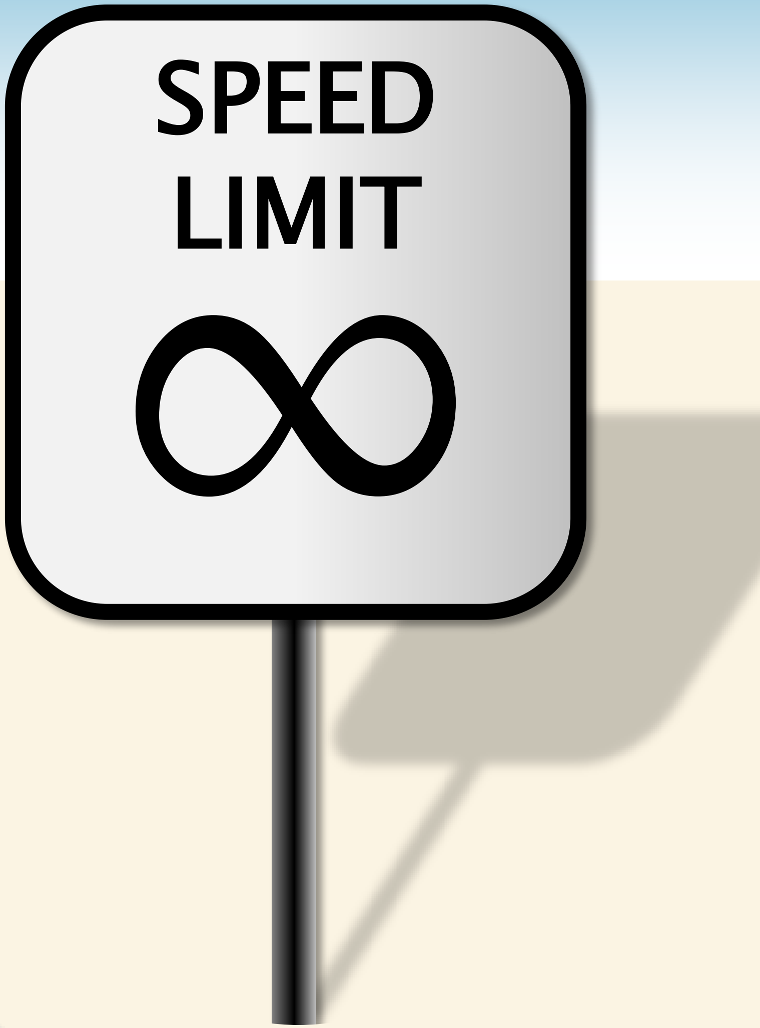
Mark stage: Breadth-first search marked all of the live objects.

Sweep stage: Scan over memory to free unmarked objects.

Mark-and-sweep doesn’t deal with fragmentation

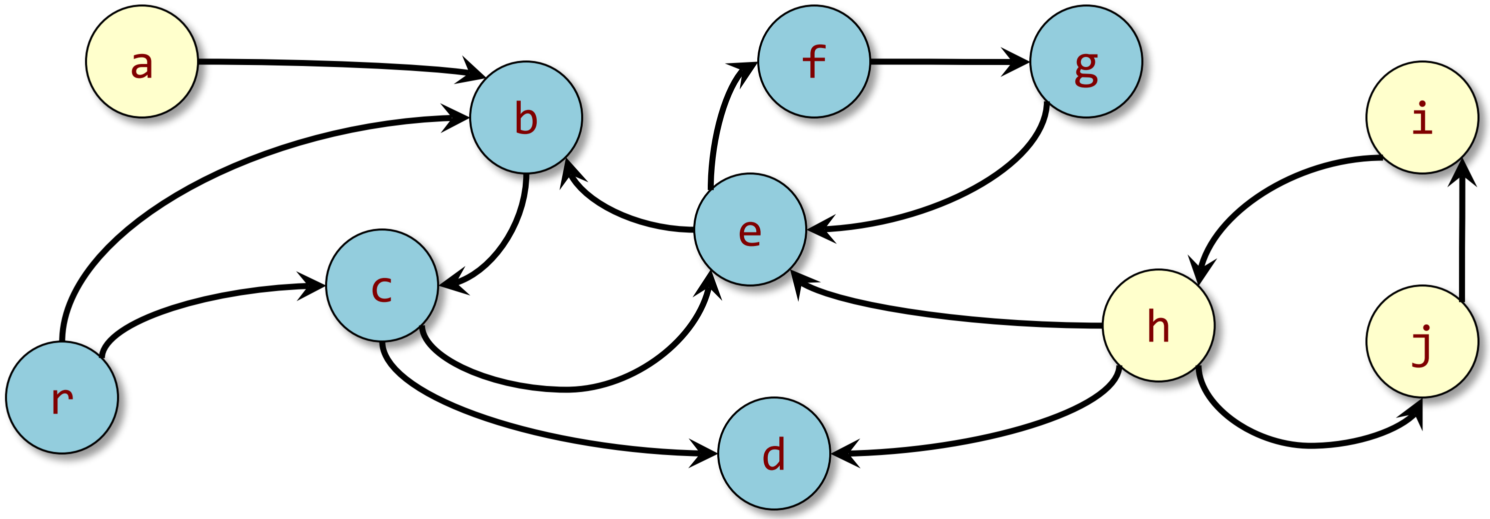
Summary

	Manual	Reference Counting	Mark and Sweep
Ease of Use	Bad	Medium	Good
Throughput	Good	Medium	Medium
Latency	Good	Good	Bad
External Fragmentation	Bad	Bad	Bad
Example	C `malloc/free`	C++ `std::shared_ptr`	Java



Stop-and-Copy Garbage Collection

Breadth-First Search



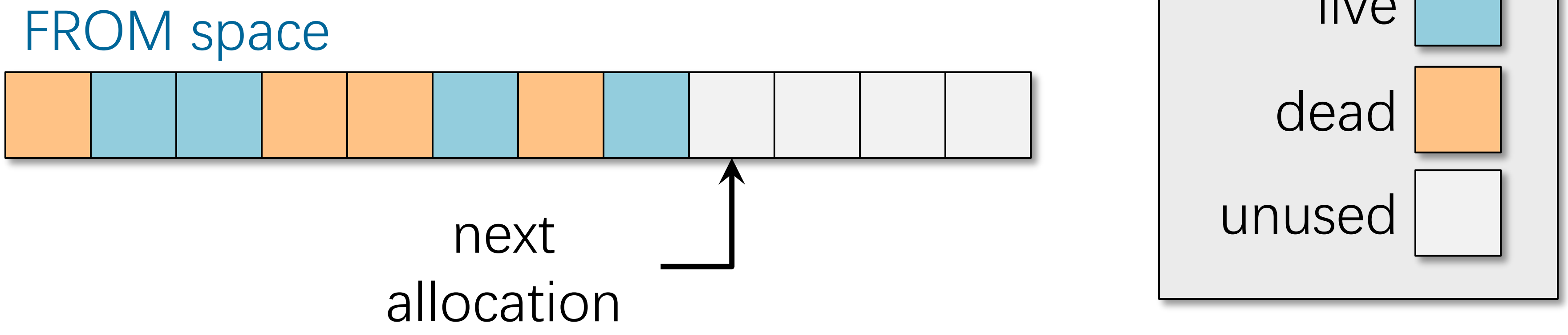
Q

r	b	c	d	e	f	g					

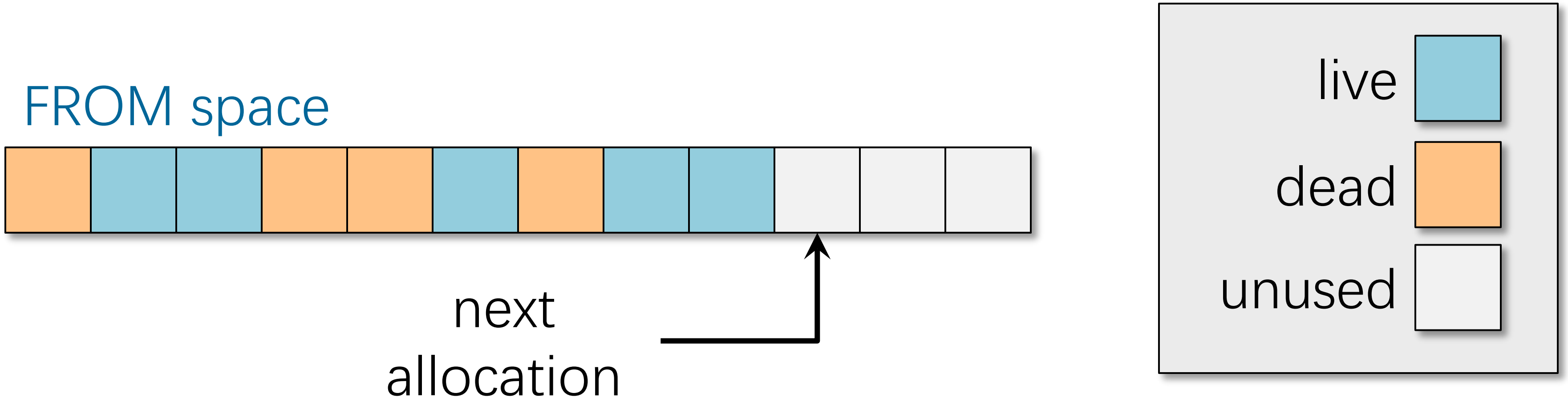
Observation

All live vertices are placed in contiguous storage in Q.

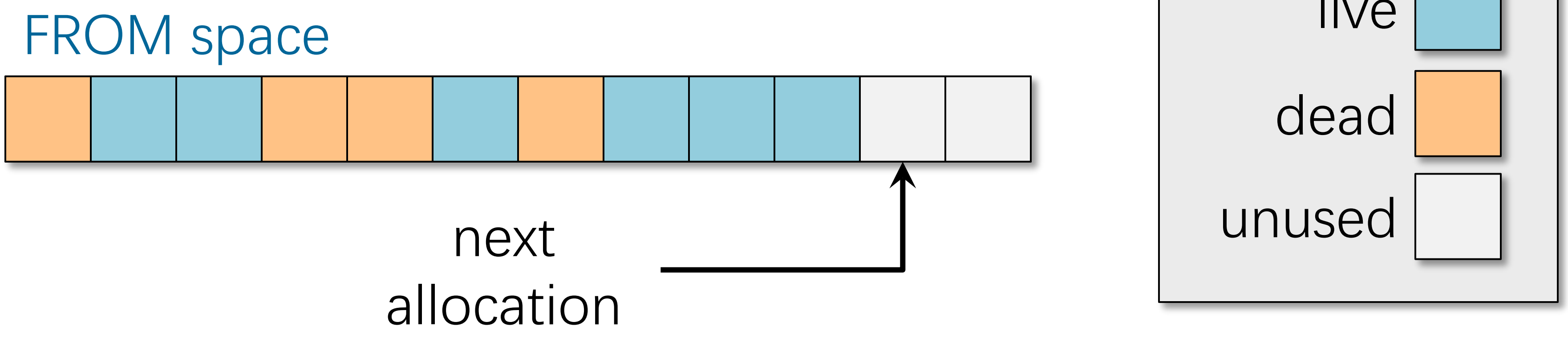
Copying Garbage Collector



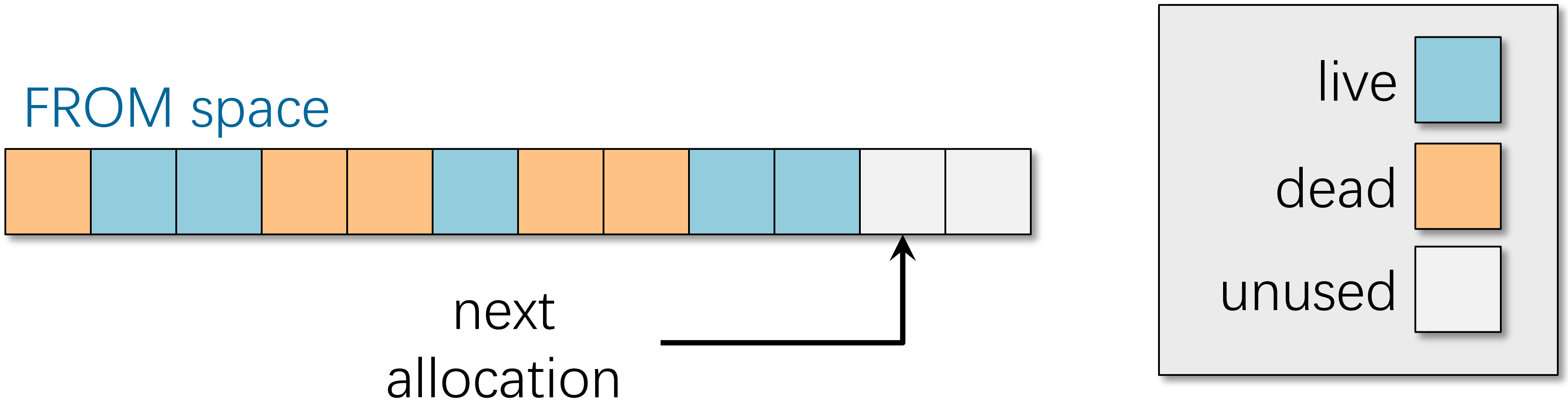
Copying Garbage Collector



Copying Garbage Collector



Copying Garbage Collector



FROM space

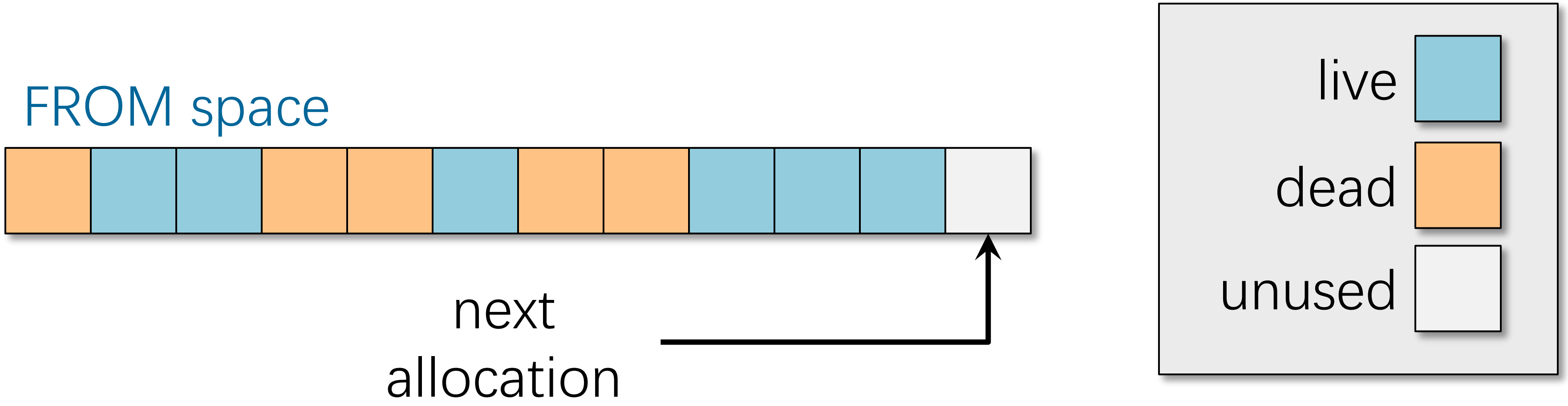
next allocation

live

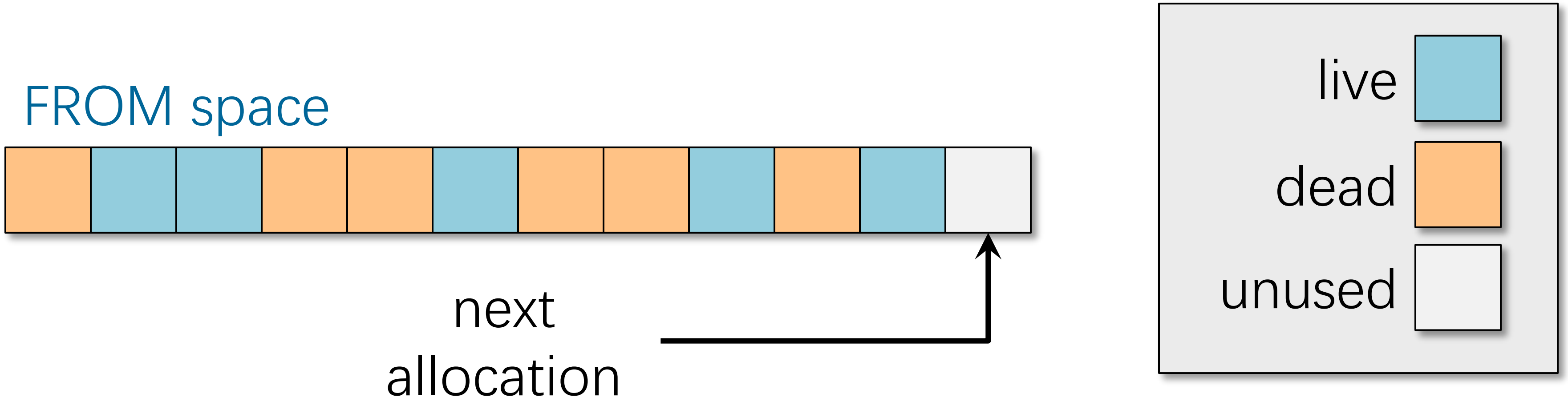
dead

unused

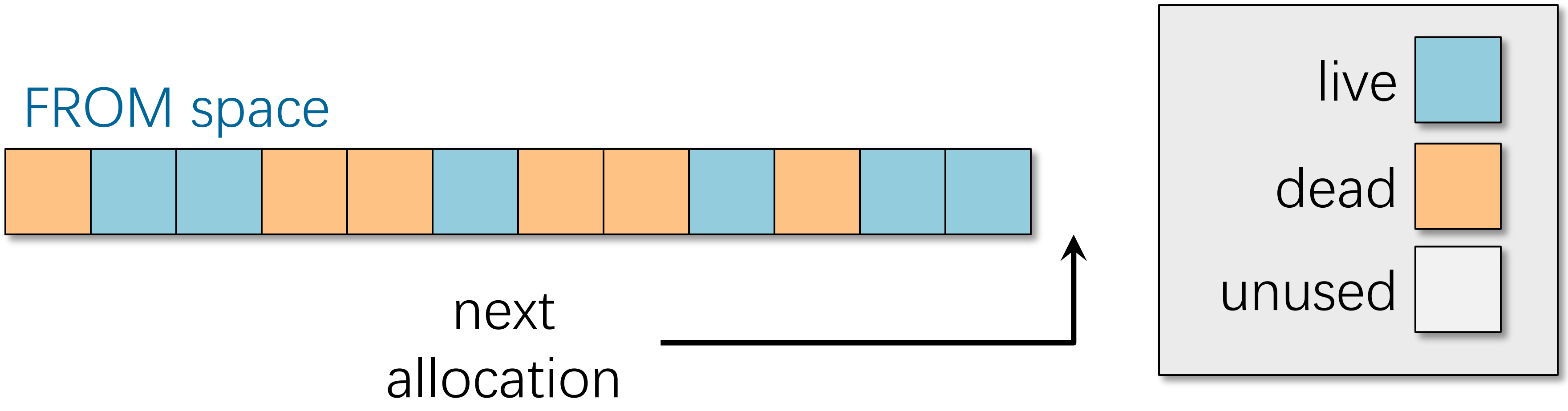
Copying Garbage Collector



Copying Garbage Collector

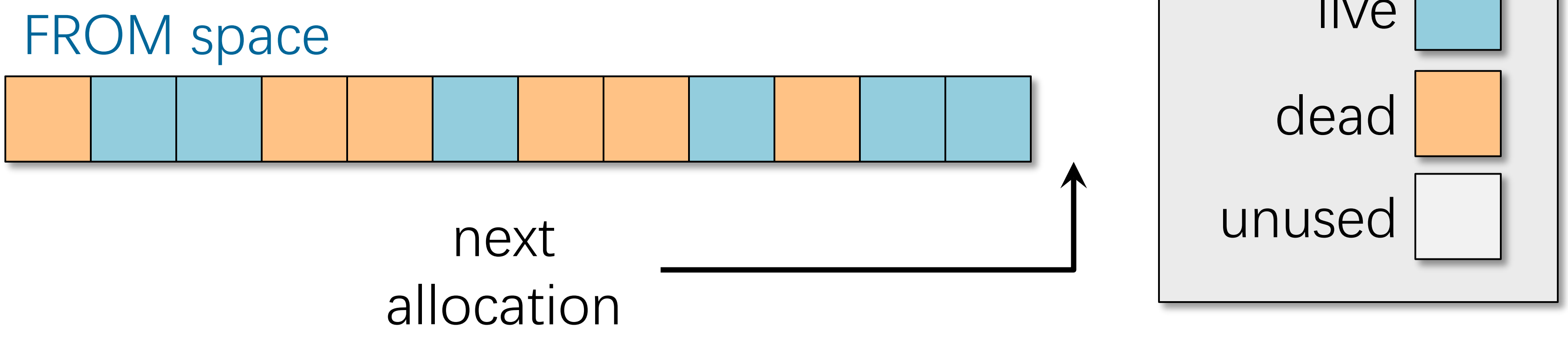


Copying Garbage Collector



When the FROM space is “full,” copy live storage using BFS with the TO space as the FIFO queue.

Copying Garbage Collector



FROM space

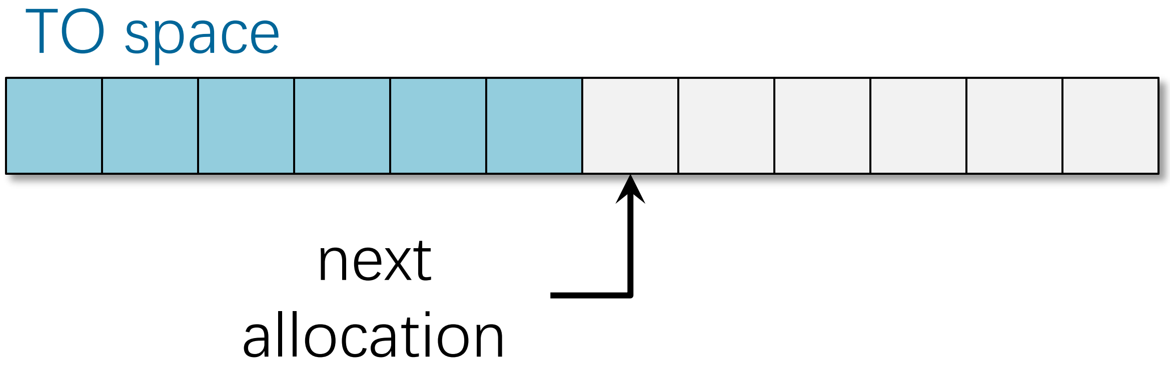
next allocation

live

dead

unused

When the FROM space is "full," copy live storage using BFS with the TO space as the FIFO queue.



TO space

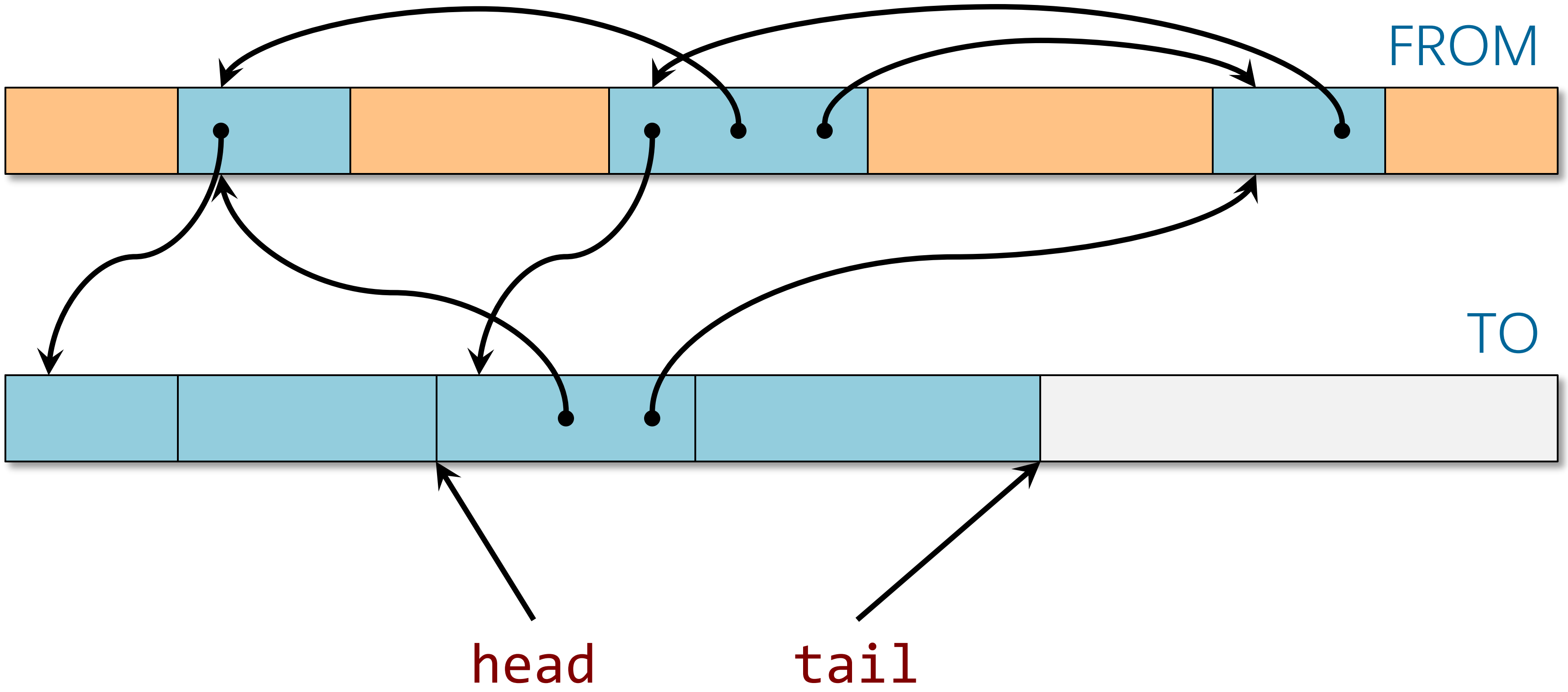
next allocation

Updating Pointers

Since the FROM address of an object is not generally equal to the TO address of the object, pointers must be updated.

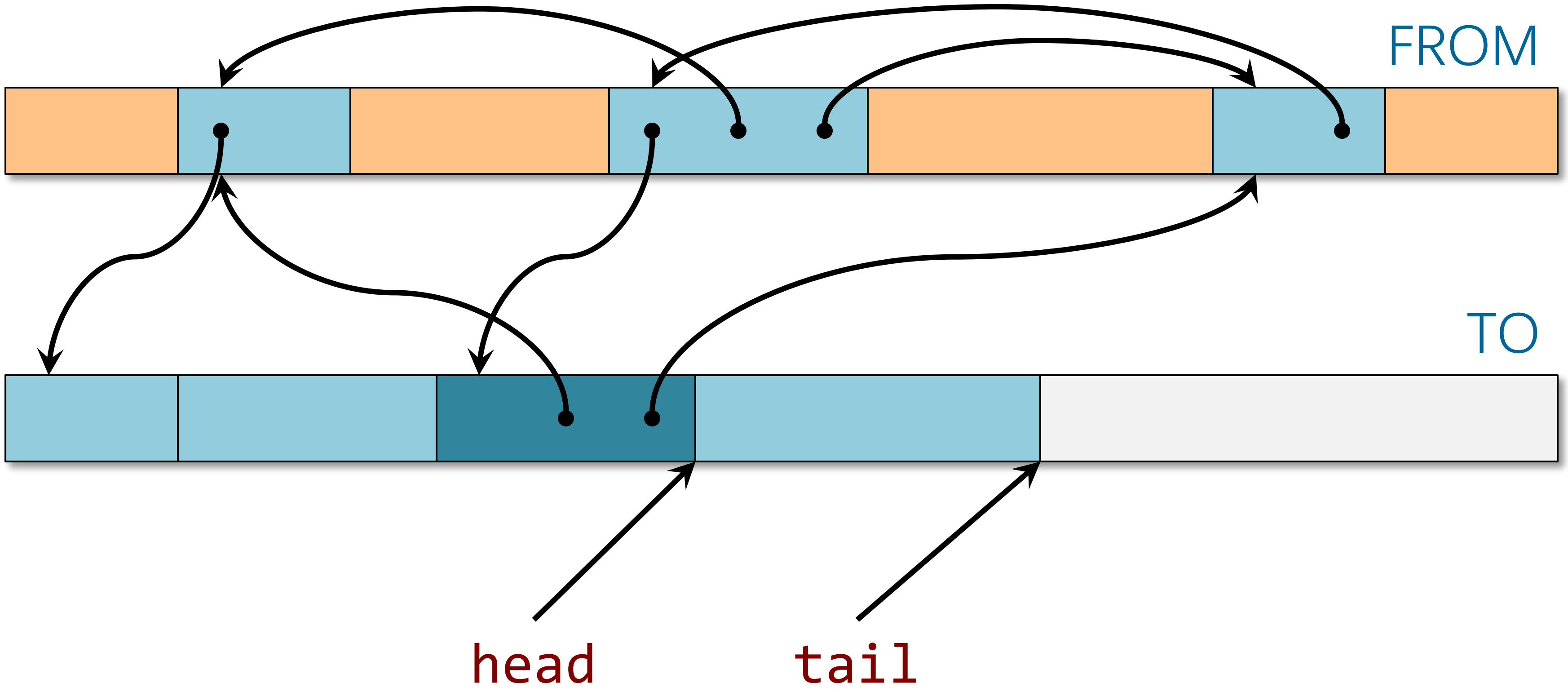
- When an object is copied to the TO space, store a forwarding pointer in the FROM object, which implicitly marks it as moved.
- When an object is removed from the FIFO queue in the TO space, update all its pointers.

Example



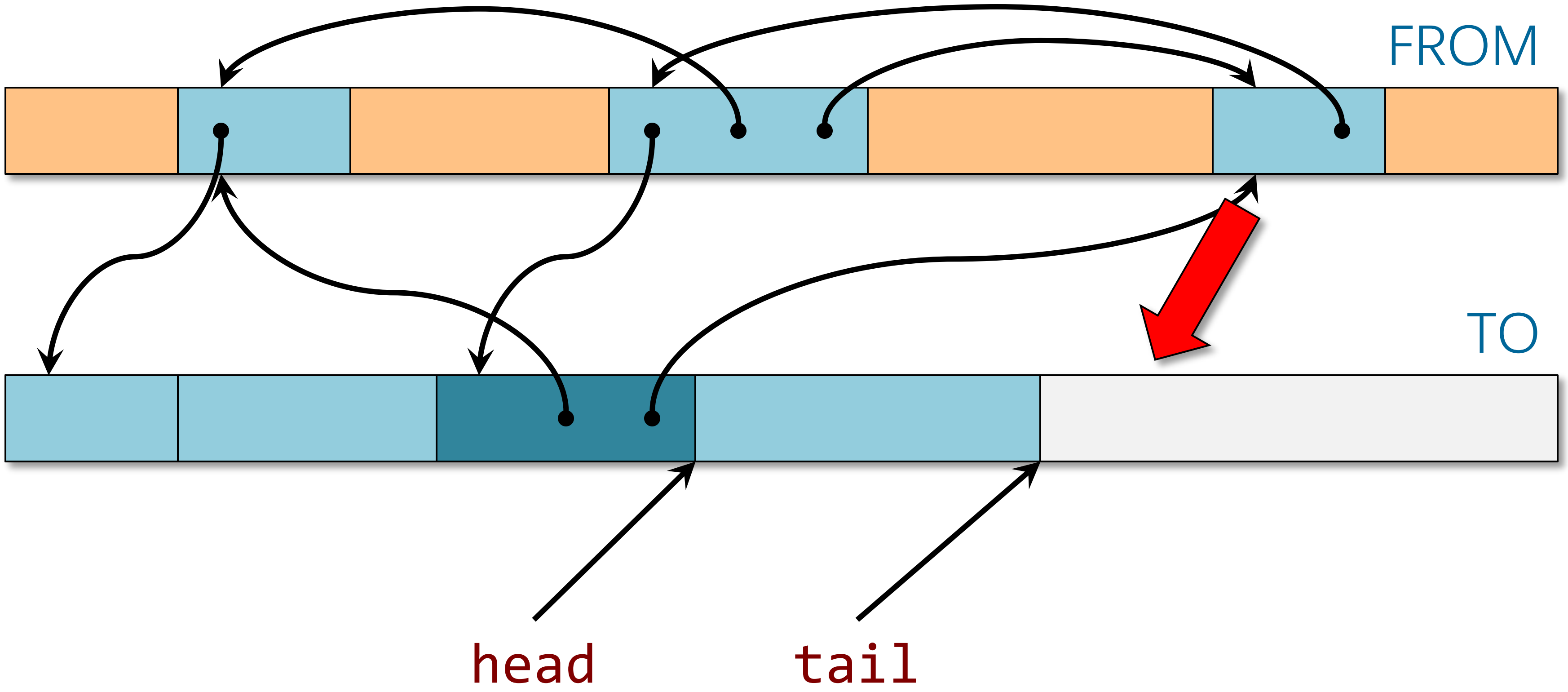
Remove an item from the queue.

Example



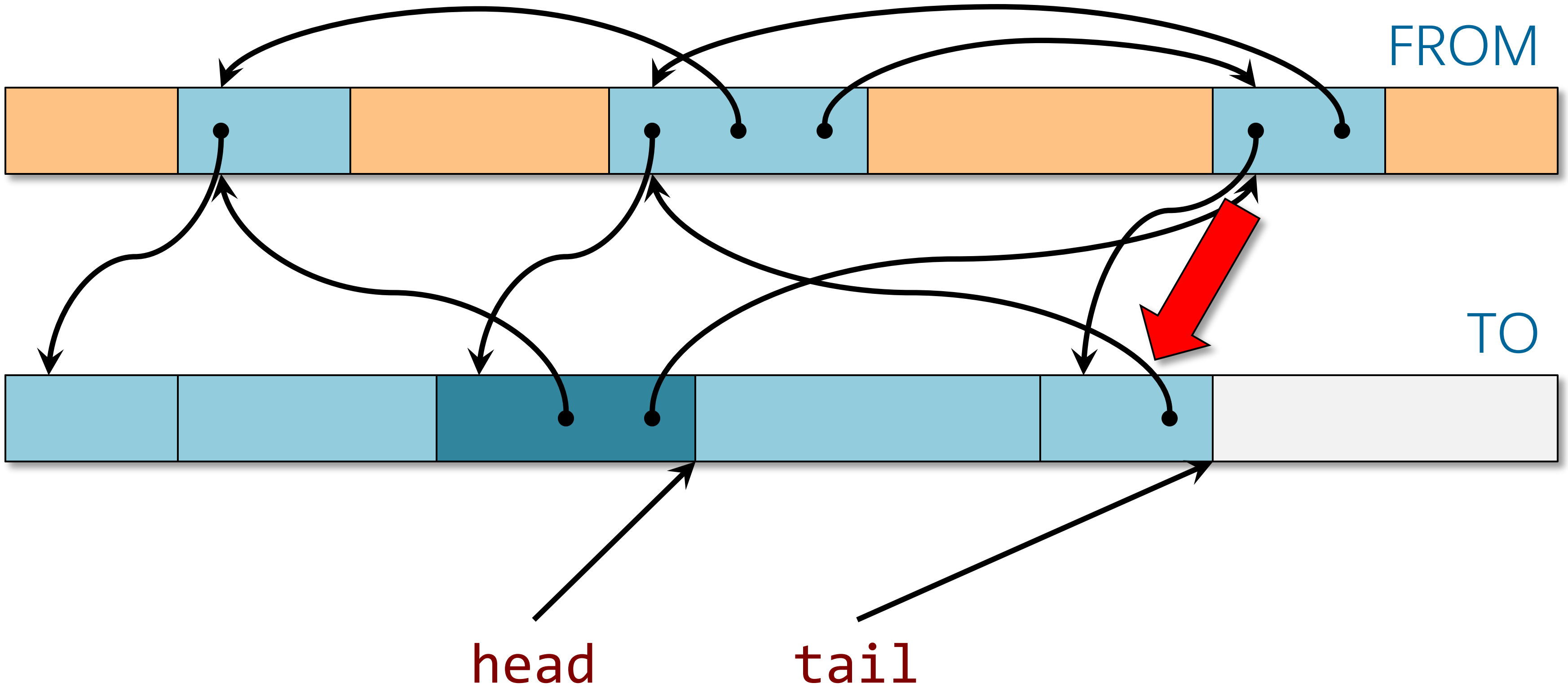
Remove an item from the queue.

Example



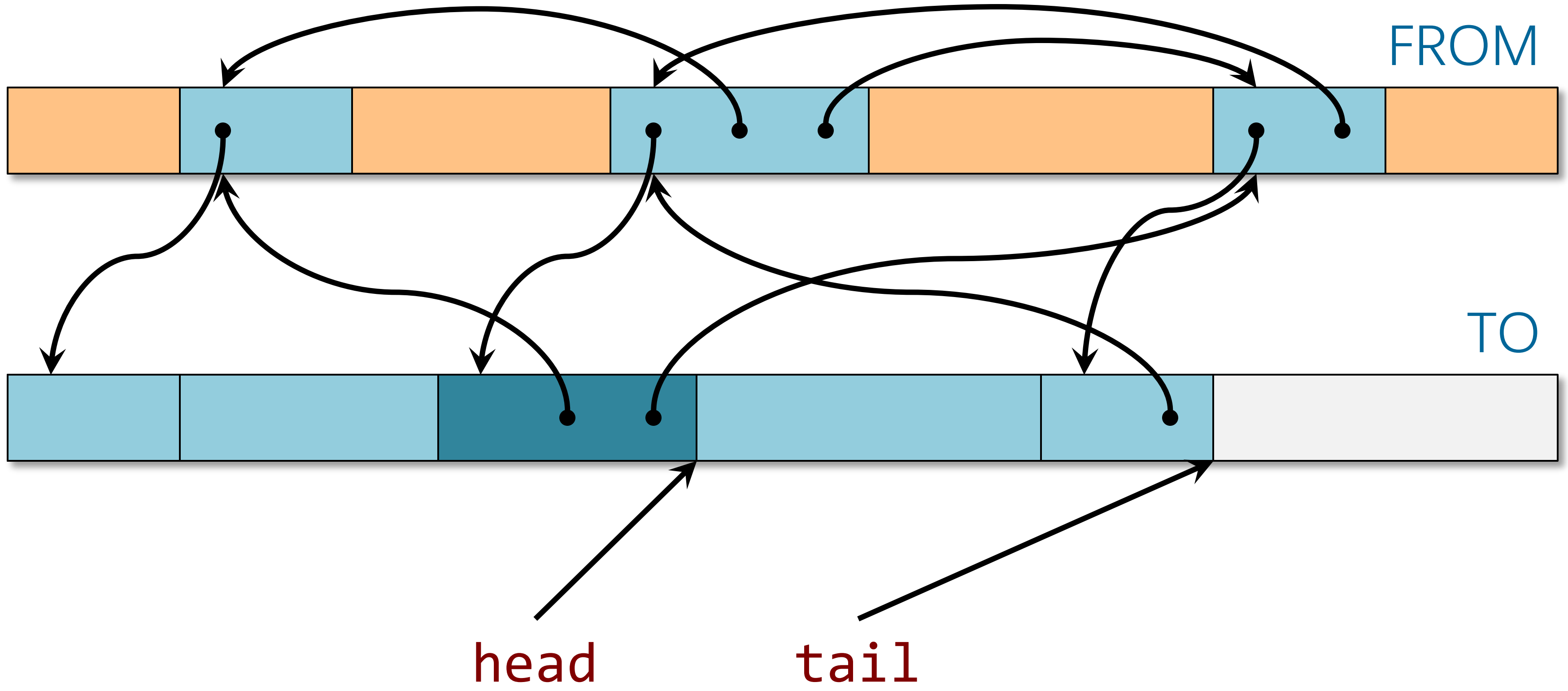
Enqueue adjacent vertices.

Example



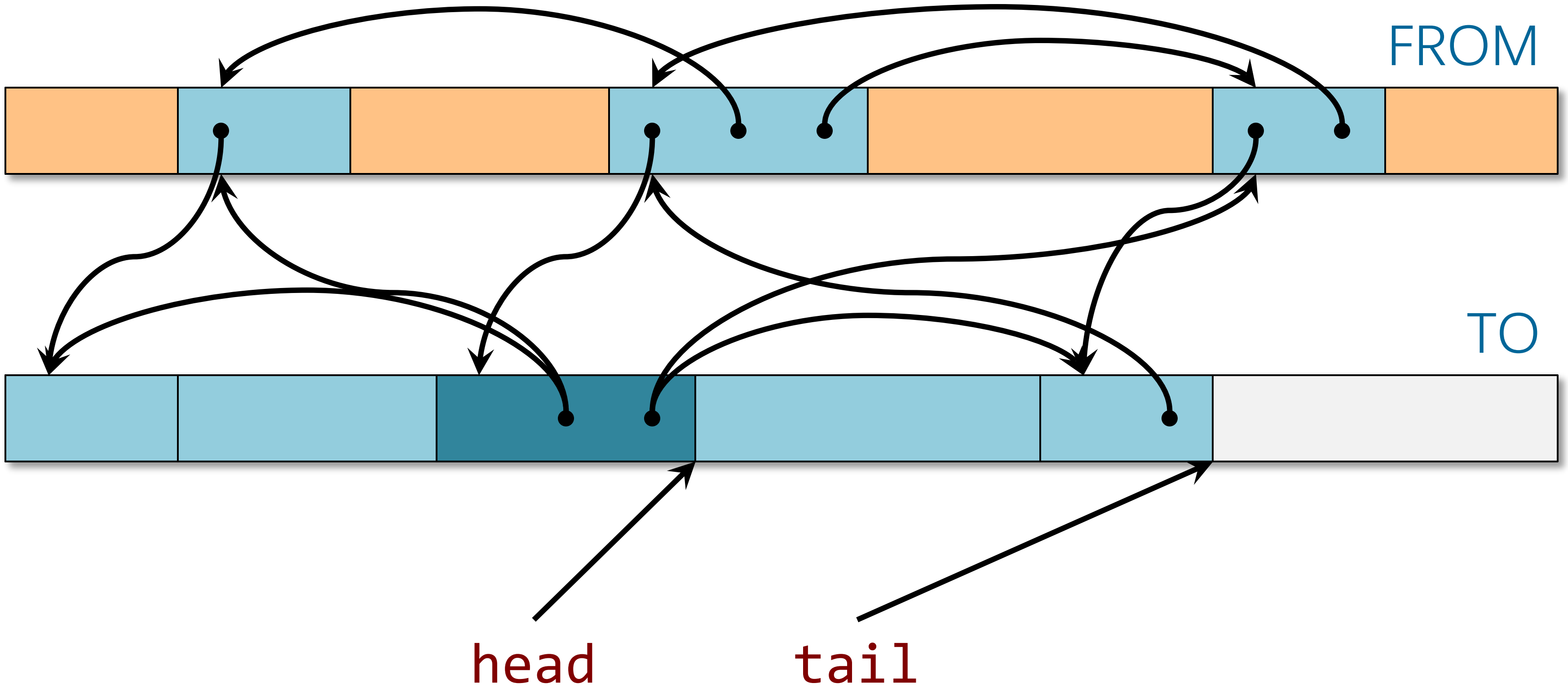
Enqueue adjacent vertices.
Place forwarding pointers in FROM vertices.

Example



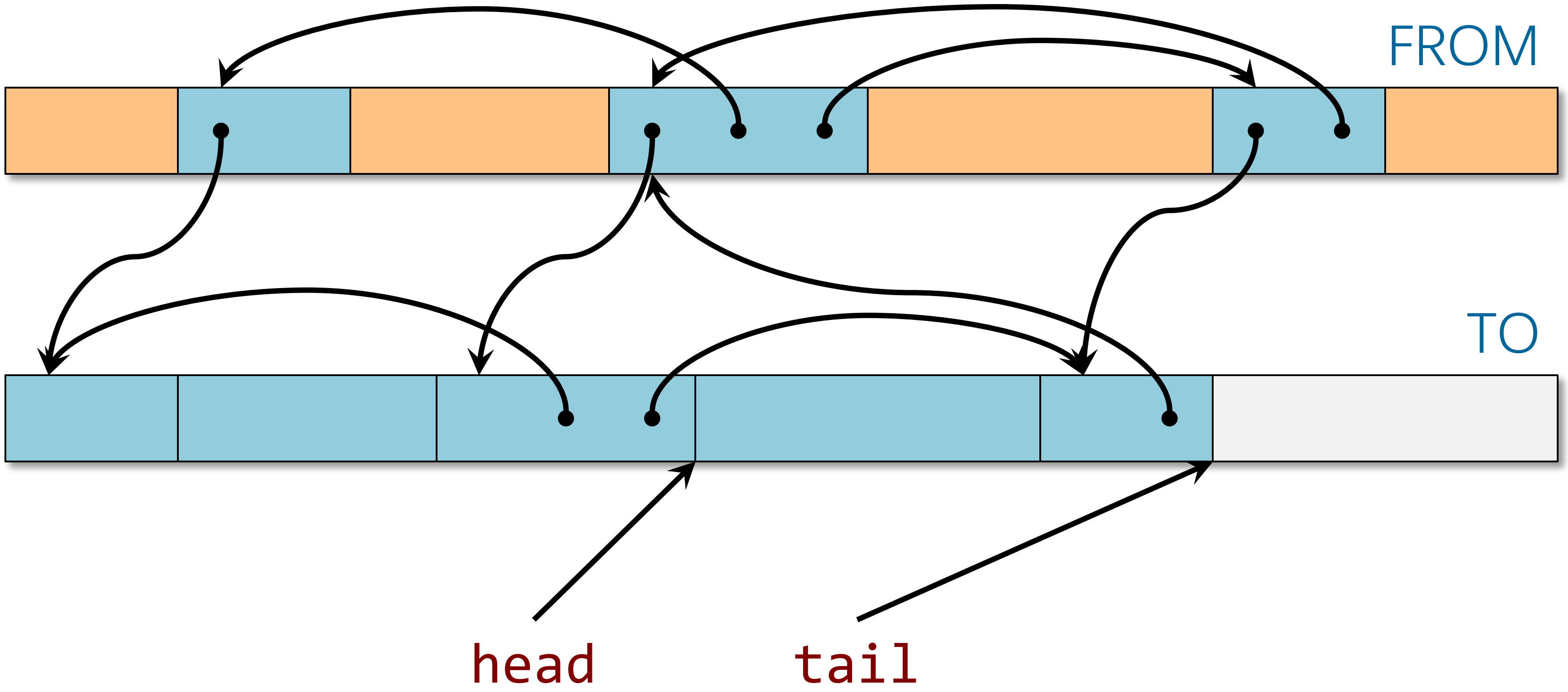
Update the pointers in the removed item to refer to its adjacent items in the TO space.

Example



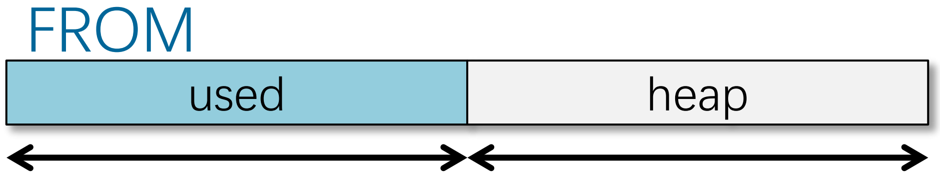
Update the pointers in the removed item to refer to its adjacent items in the TO space.

Example



Linear time to copy and update all vertices.

When Is the FROM Space “Full”?



- Request new heap space equal to the used space, and consider the FROM space to be “full” when this heap space has been allocated.
- The cost of garbage collection is proportional to the size of the new heap space ⇒ amortized $O(1)$ overhead, assuming that the user program touches all the memory allocated.
- Moreover, the VM space required is $O(1)$ times optimal by locating the FROM and TO spaces in different regions of VM where they cannot interfere with each other.

Summary

	Manual	Reference Counting	Mark and Sweep	Stop and Copy
Ease of Use	Bad	Medium	Good	Good
Throughput	Good	Medium	Medium	Bad
Latency	Good	Good	Bad	Bad
External Fragmentation	Bad	Bad	Bad	Good
Example	C `malloc/free`	C++ `std::shared_ptr`	Java	C#

Dynamic Storage Allocation

Lots more is known and unknown about dynamic storage allocation. Strategies include

- buddy system,
- variants of mark-and-sweep,
- generational garbage collection,
- real-time garbage collection,
- multithreaded storage allocation,
- parallel garbage collection,
- etc.