6.172
Performance
Engineering
of Software
Systems

Lecture 11

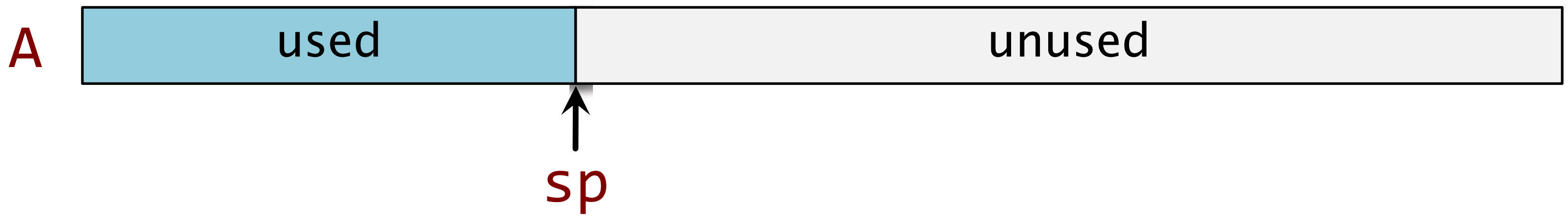
Storage Allocation

Julian Shun

Stacks

Stack Allocation

Array and pointer

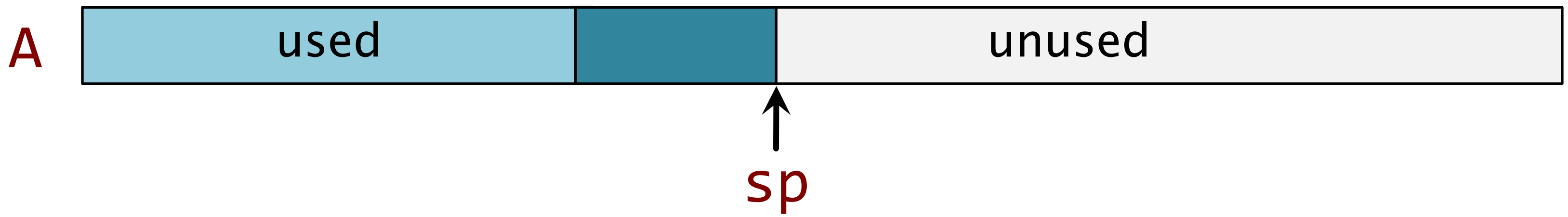


Allocate x bytes

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

Stack Allocation

Array and pointer



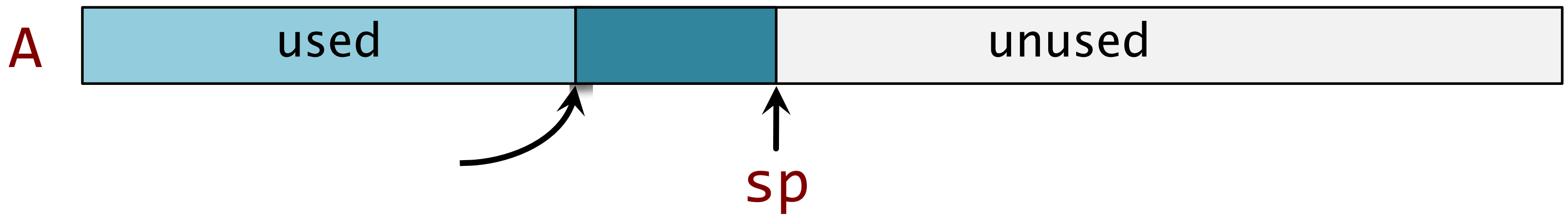
Allocate x bytes

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

Should check for
stack overflow.

Stack Allocation

Array and pointer



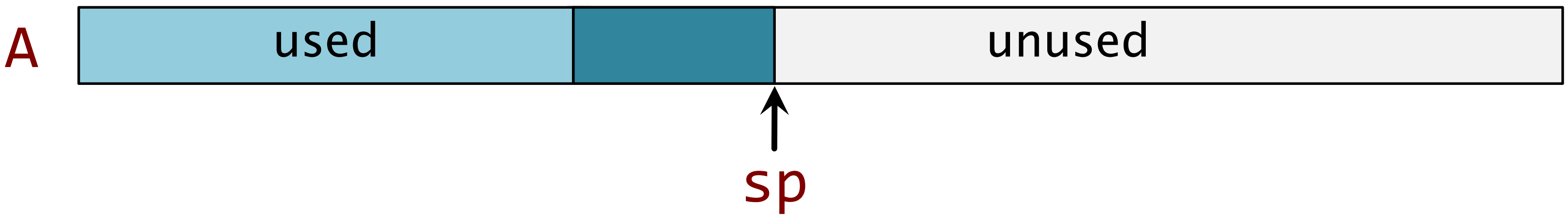
Allocate x bytes

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

Should check for
stack overflow.

Stack Deallocation

Array and pointer



Allocate x bytes

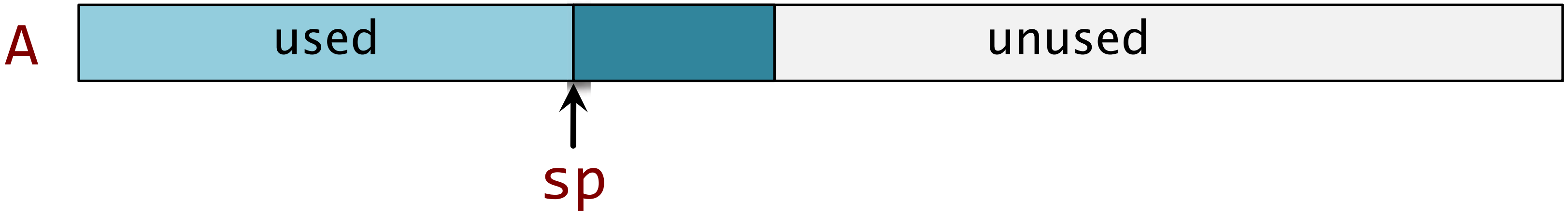
```
sp += x;
return sp - x;
```

Free x bytes

```
sp -= x;
```

Stack Deallocation

Array and pointer



Allocate x bytes

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

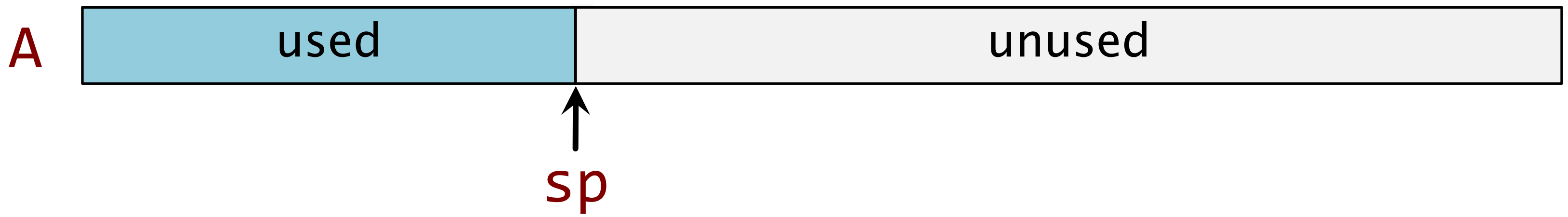
Free x bytes

```
sp -= x;
```

Should check for stack underflow.

Stack Storage

Array and pointer



Allocate x bytes

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

Free x bytes

```
sp -= x;
```

- Allocating and freeing take $\Theta(1)$ 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.

Stacks and Heaps

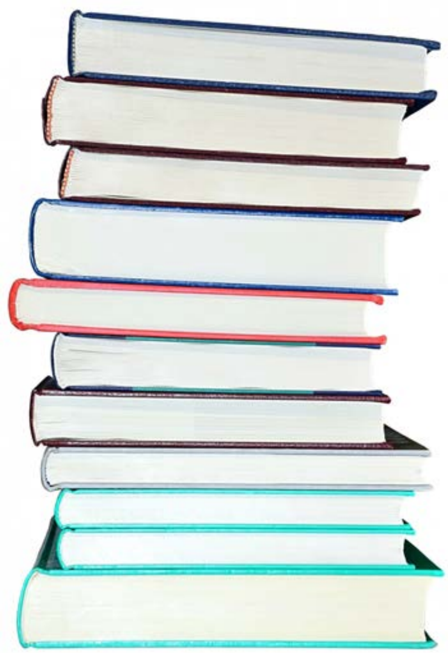


Image is in the public domain.

Stack

Image is in the public domain.

Heap

Fixed-Size Heap Allocation

Heap Allocation*

C provides `malloc()` and `free()`.
C++ provides `new` and `delete`.

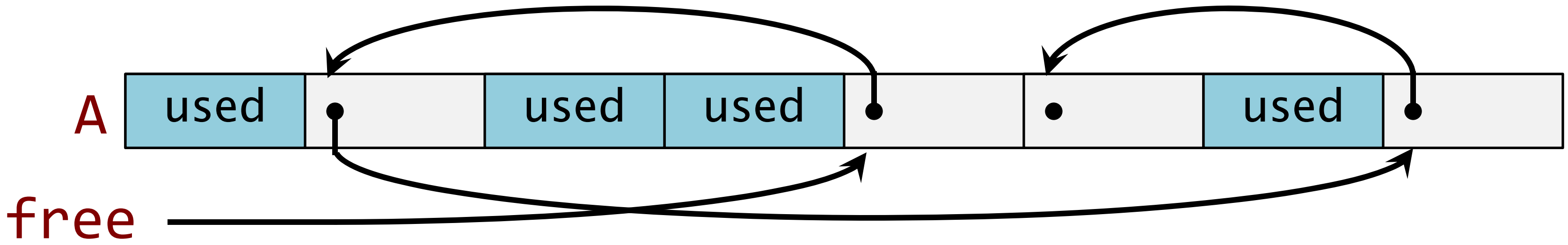
Unlike Java and Python, C and C++ provide no garbage collector. Heap storage allocated by the programmer must be freed explicitly. Failure to do so creates a memory leak. Also, watch for dangling pointers and double freeing.

Memory checkers (e.g., AddressSanitizer, Valgrind) can assist in finding these pernicious bugs.

*Do not confuse with a heap data structure.

Fixed-Size Allocation

Free list



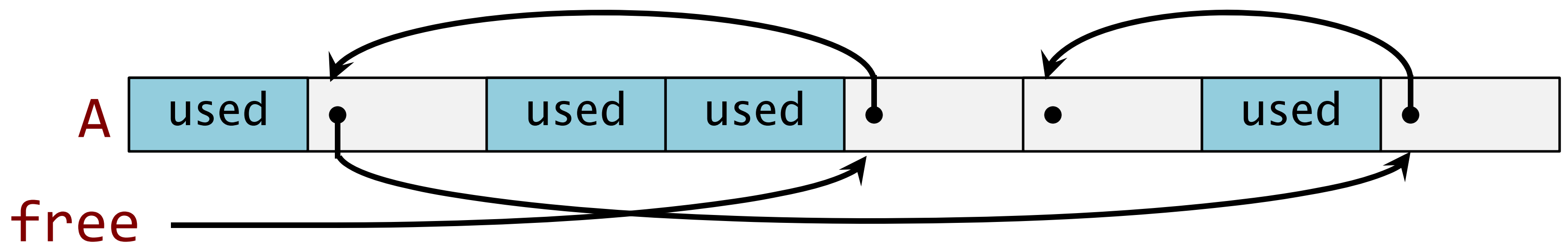
- Every piece of storage has the same size
- Unused storage has a pointer to next unused block

Bitmap mechanism

- Bit for each block saying whether or not it is free
- Bit tricks for allocation

Fixed-Size Allocation

Free list

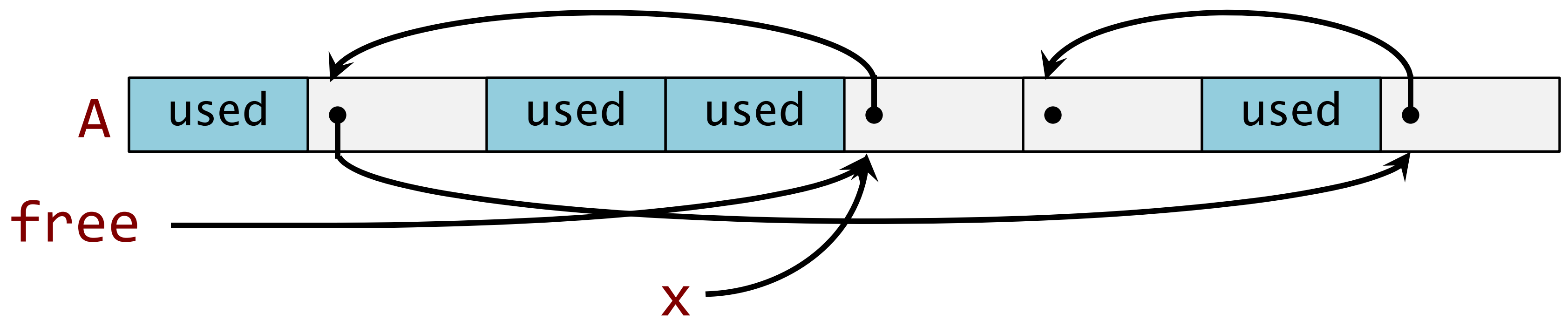


Allocate 1 object

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

Fixed-Size Allocation

Free list

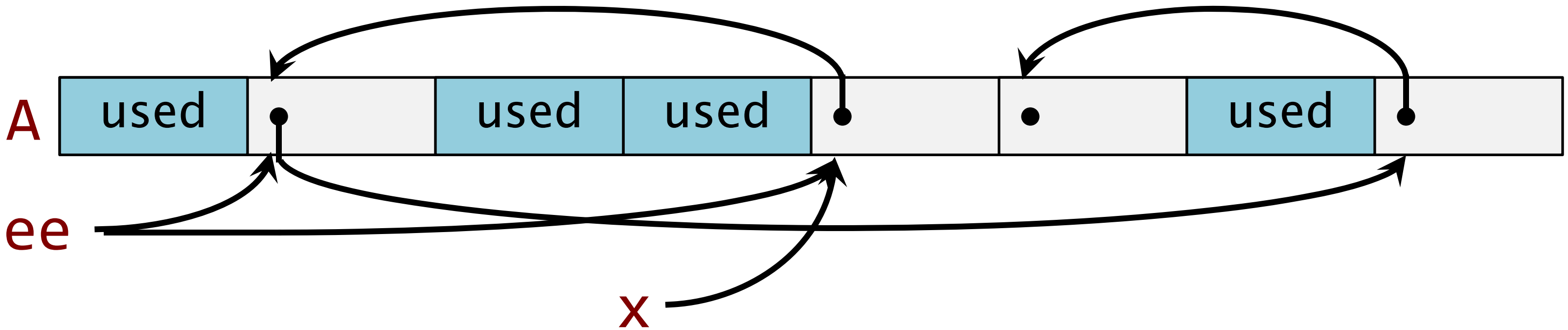


Allocate 1 object

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

Fixed-Size Allocation

Free list



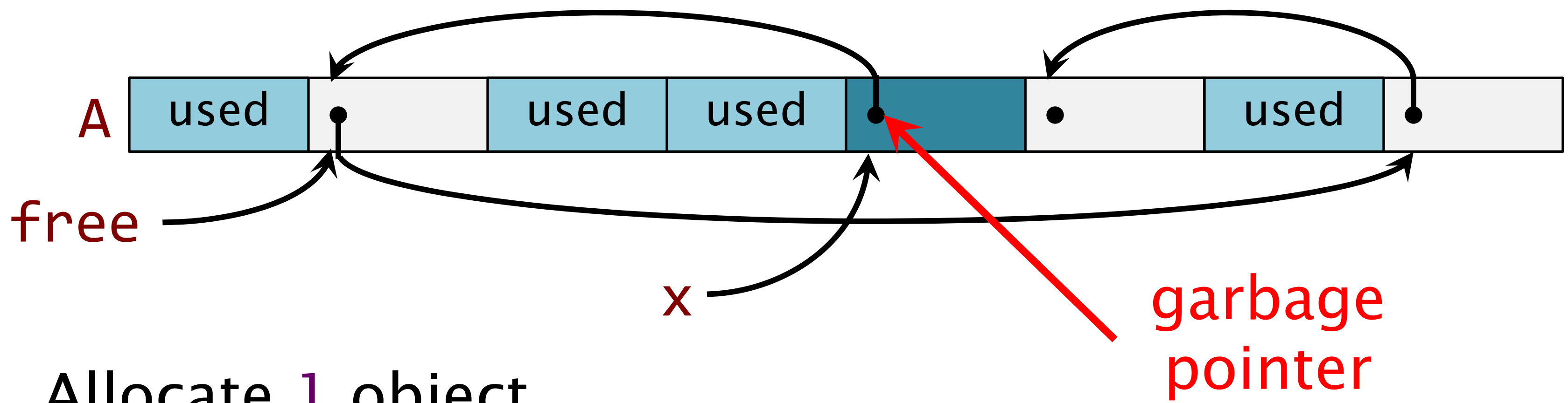
Allocate 1 object

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

Should check
`free != NULL`.

Fixed-Size Allocation

Free list

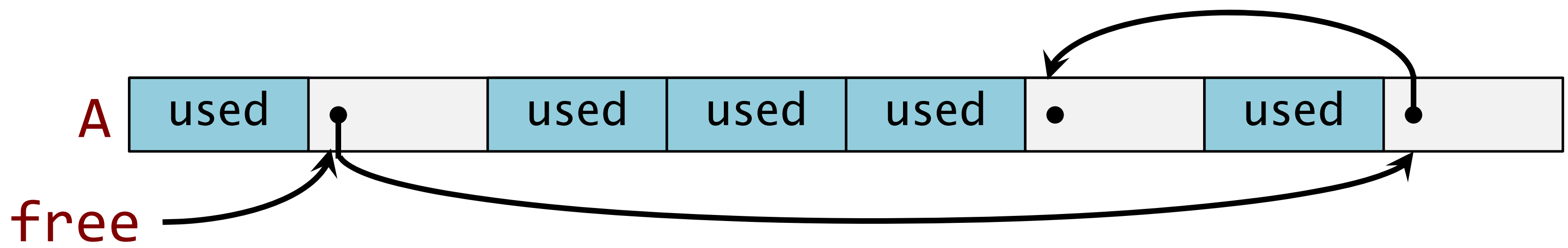


Allocate 1 object

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

Fixed-Size Allocation

Free list

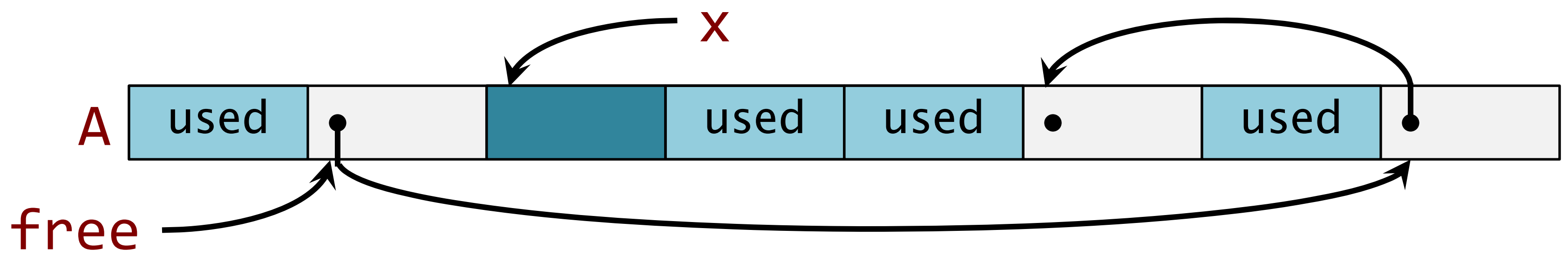


Allocate 1 object

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

Fixed-Size Deallocation

Free list



Allocate 1 object

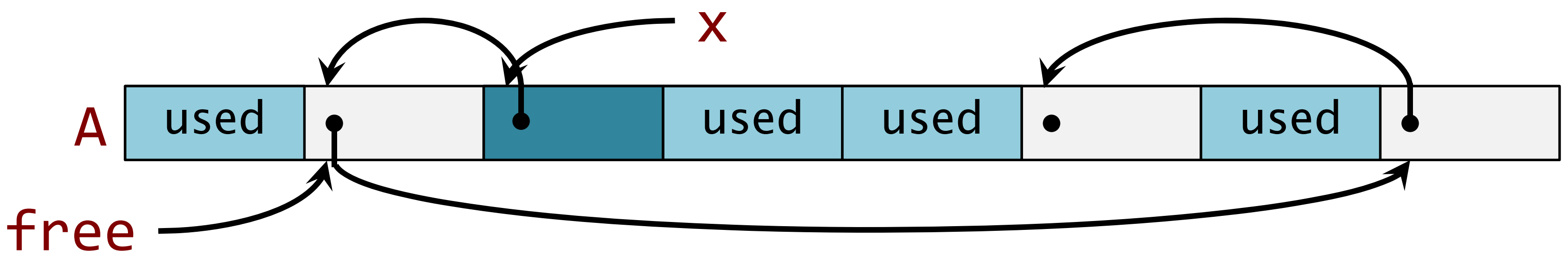
```
x = free;
free = free->next;
return x;
```

free object x

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

Fixed-Size Deallocation

Free list



Allocate 1 object

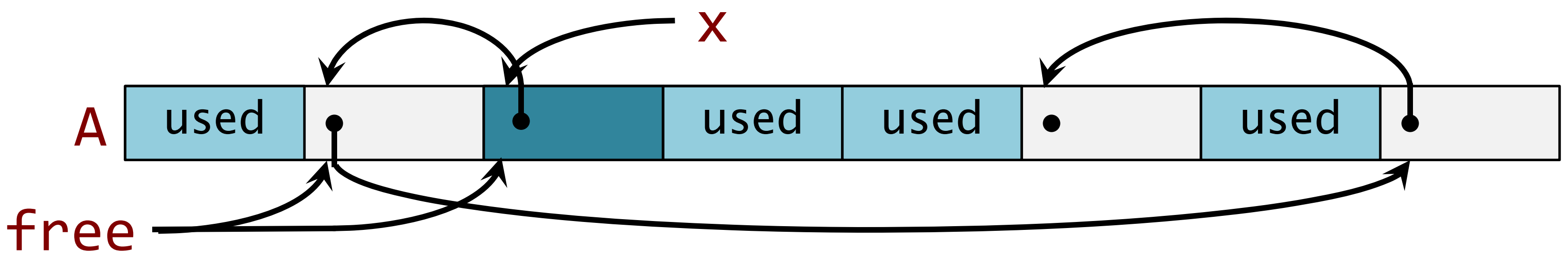
```
x = free;
free = free->next;
return x;
```

free object x

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

Fixed-Size Deallocation

Free list



Allocate 1 object

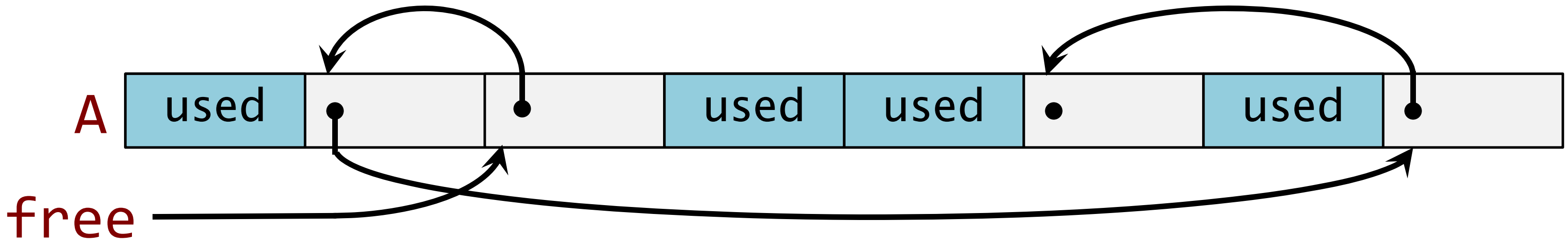
```
x = free;
free = free->next;
return x;
```

free object x

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

Fixed-Size Deallocation

Free list



Allocate 1 object

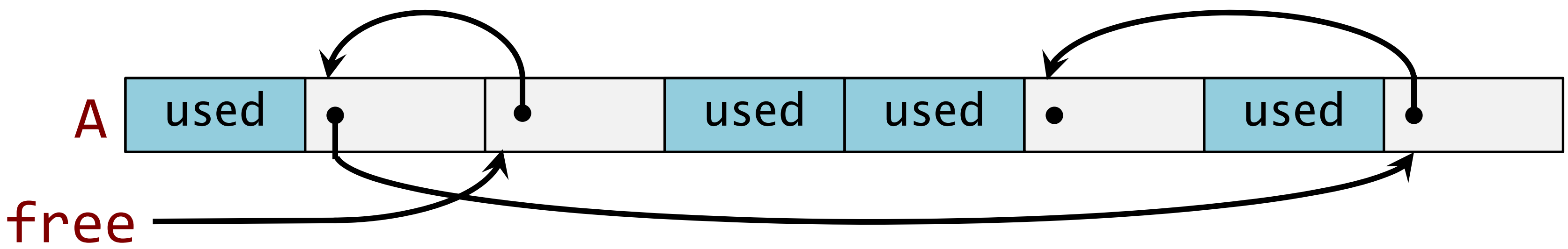
```
x = free;
free = free->next;
return x;
```

free object x

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

Free Lists

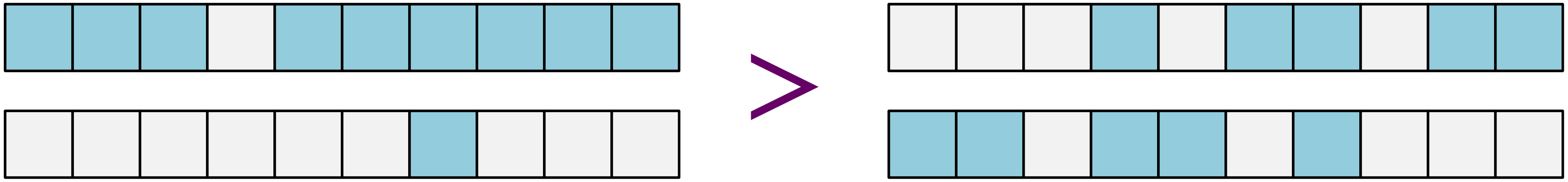
Free list



- Allocating and freeing take $\Theta(1)$ 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.

Mitigating External Fragmentation

- Keep a free list (or bitmap) per disk page.
- Allocate from the free list for the fullest page.
- Free a block of storage 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 affecting program performance.
- 90–10 is better than 50–50:



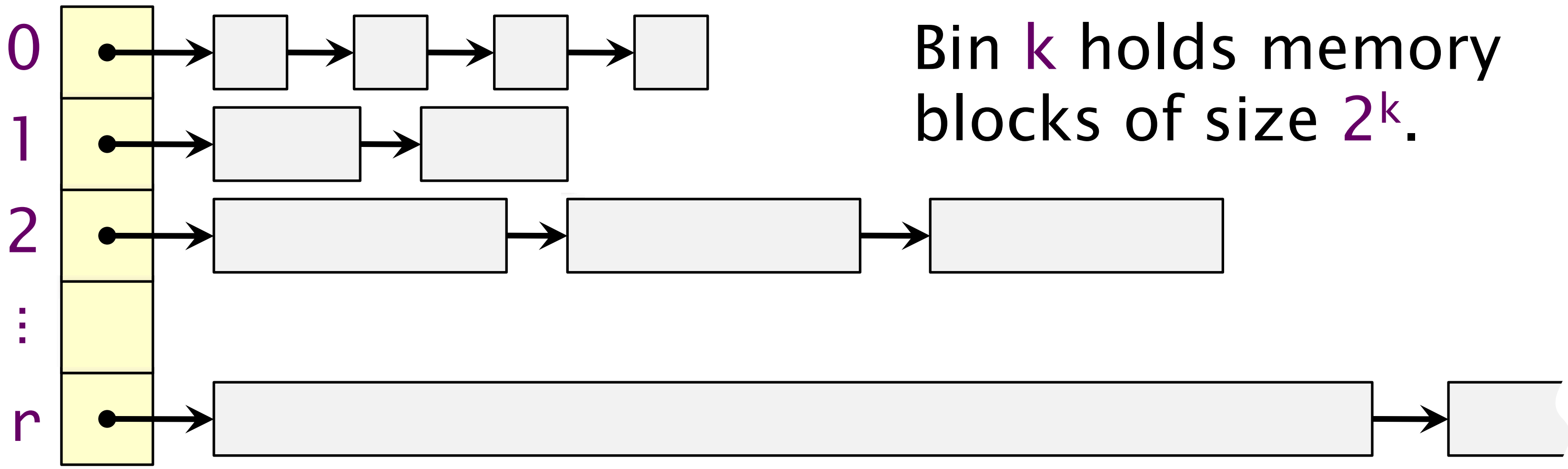
Probability that 2 random accesses hit the same page
$= .9\times.9 + .1\times.1 = .82$ versus $.5\times.5 + .5\times.5 = .5$

VARIABLE-SIZE HEAP ALLOCATION

Variable-Size Allocation

Binned free lists

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

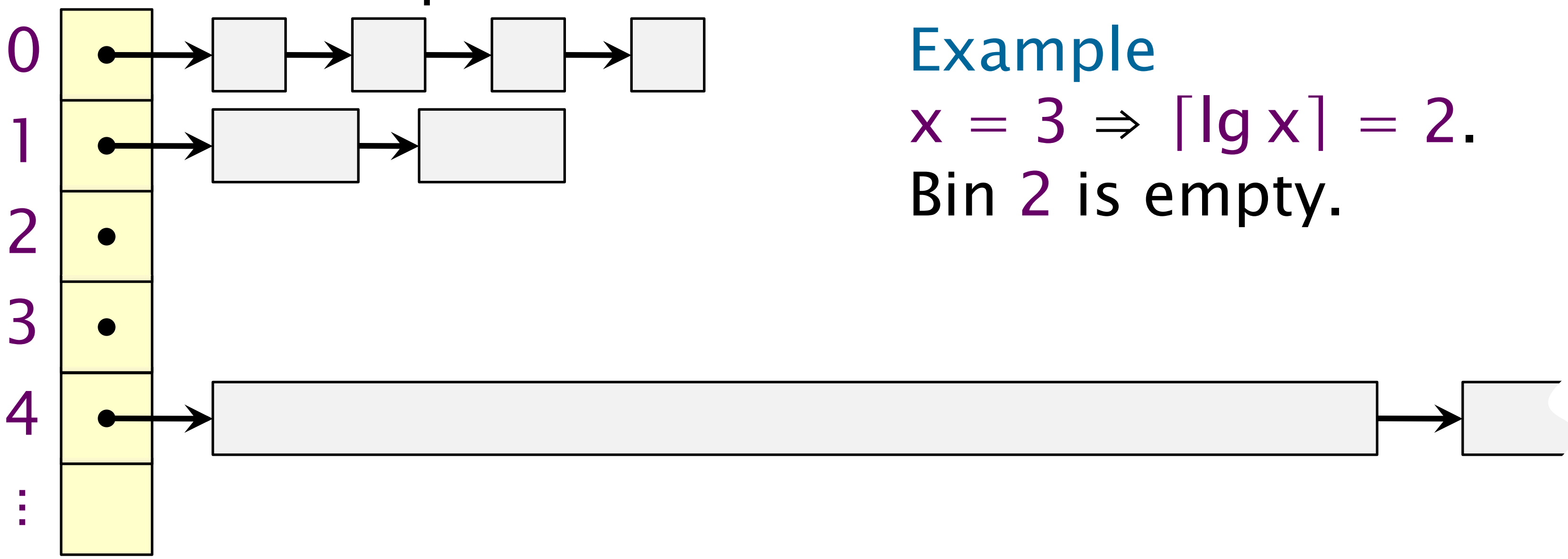


Bin k holds memory blocks of size 2^k.

Allocation for Binned Free Lists

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.

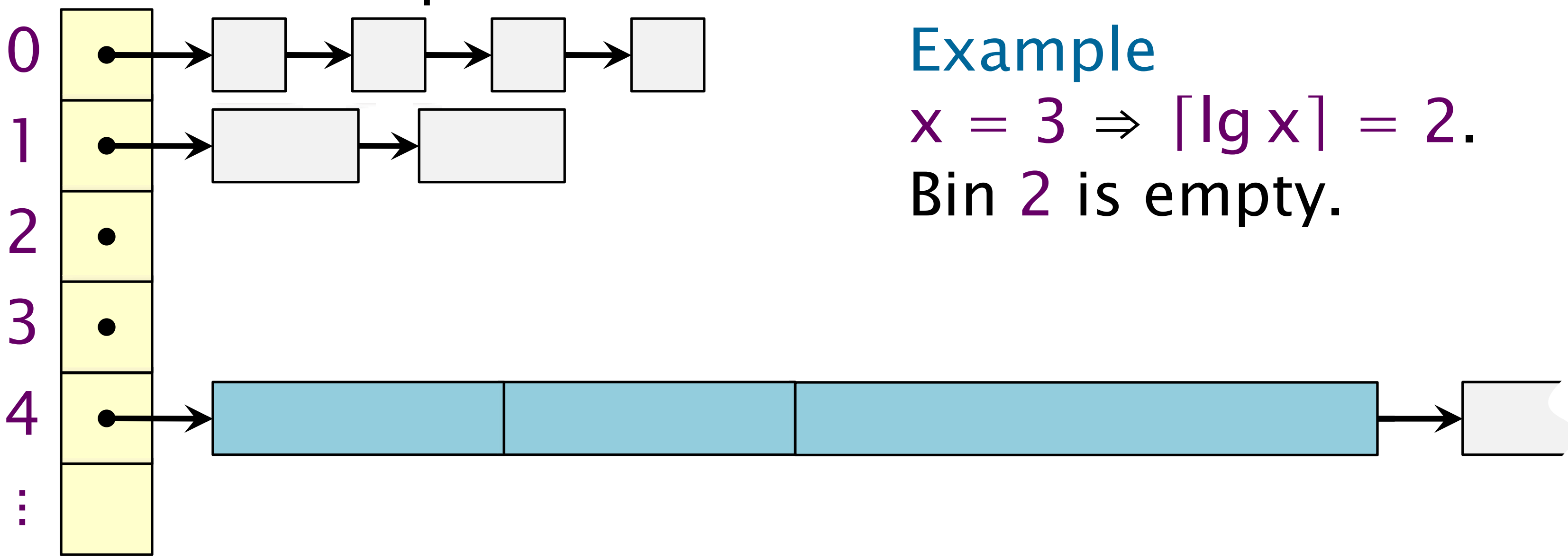


Example
$x = 3 \Rightarrow \lceil \lg x \rceil = 2$.
Bin 2 is empty.

Allocation for Binned Free Lists

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

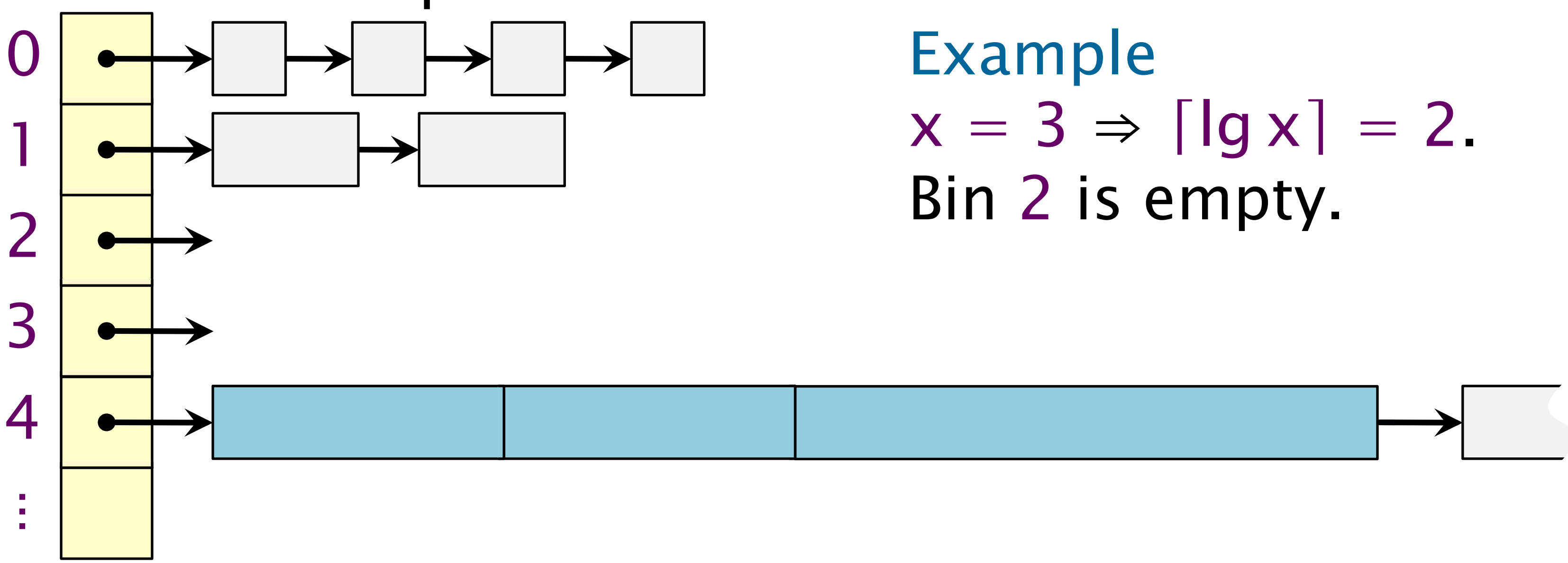
$x = 3 \Rightarrow \lceil \lg x \rceil = 2$.

Bin 2 is empty.

Allocation for Binned Free Lists

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 $2^{k'-1}, 2^{k'-2}, \ldots, 2^k, 2^k$, and distribute the pieces.



Example

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

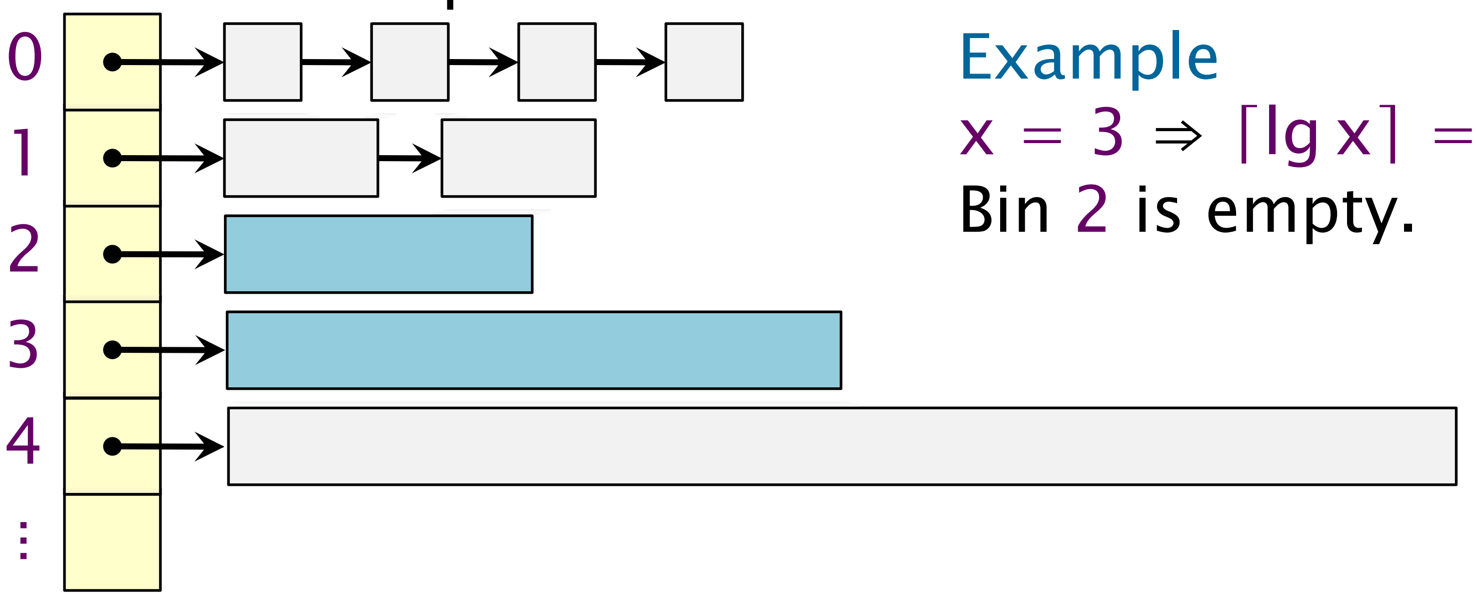
Bin 2 is empty.

return

Allocation for Binned Free Lists

Allocate x bytes

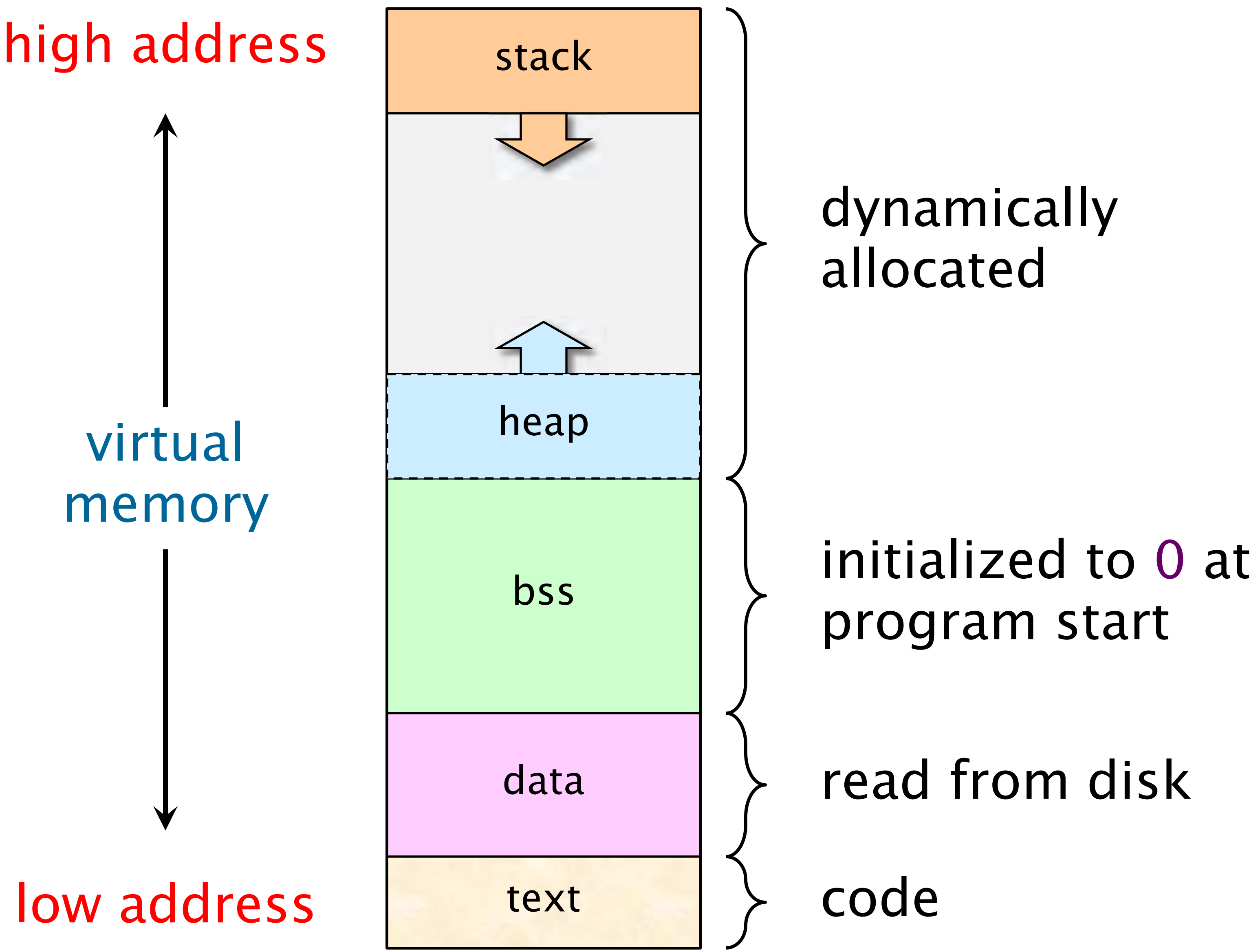
- If bin $k = \lceil \lg x \rceil$ is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin $k' > k$, split it up into blocks of sizes $2^{k'-1}$, $2^{k'-2}$, …, 2^k, 2^k, and distribute the pieces.*



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.

Storage Layout of a Program



How Virtual is Virtual Memory?

Q. Since a 64-bit address space takes over a century to write at a rate of 4 billion bytes per second, we effectively never run out of virtual memory. Why not just allocate out of virtual memory 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 $2^{\lceil \lg x \rceil} \leq 2x$ 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 $\Theta(1)$–competitive with the optimal 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 tends to be deallocated as a stack (LIFO) or in batches.

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 (Java, Python) or do-it-yourself.

Garbage Collection

Terminology

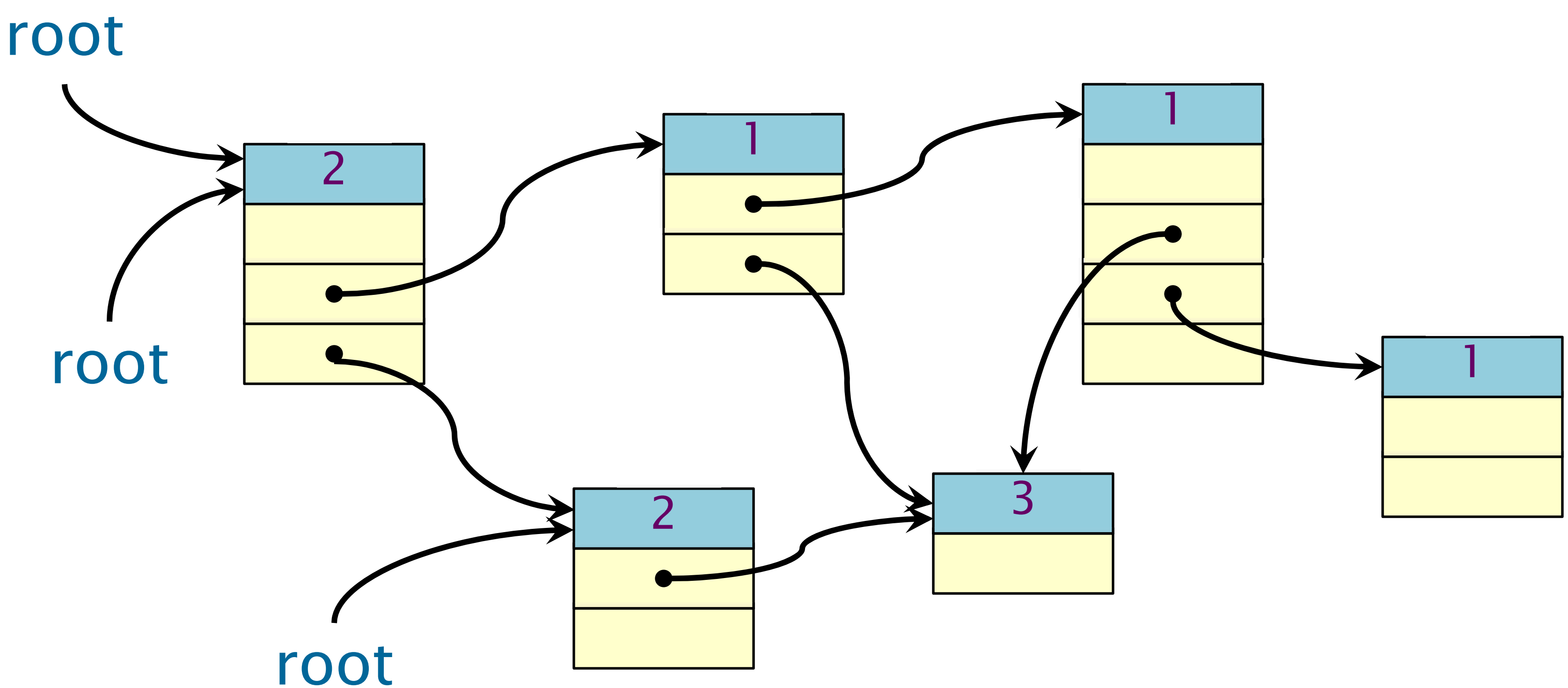
- Roots are objects directly accessible by the program (globals, stack, etc.).
- Live objects are reachable from the roots by following pointers.
- Dead objects are inaccessible and can be recycled.

How can the GC identify pointers?

- Strong typing.
- Prohibit pointer arithmetic (which may slow down some programs).

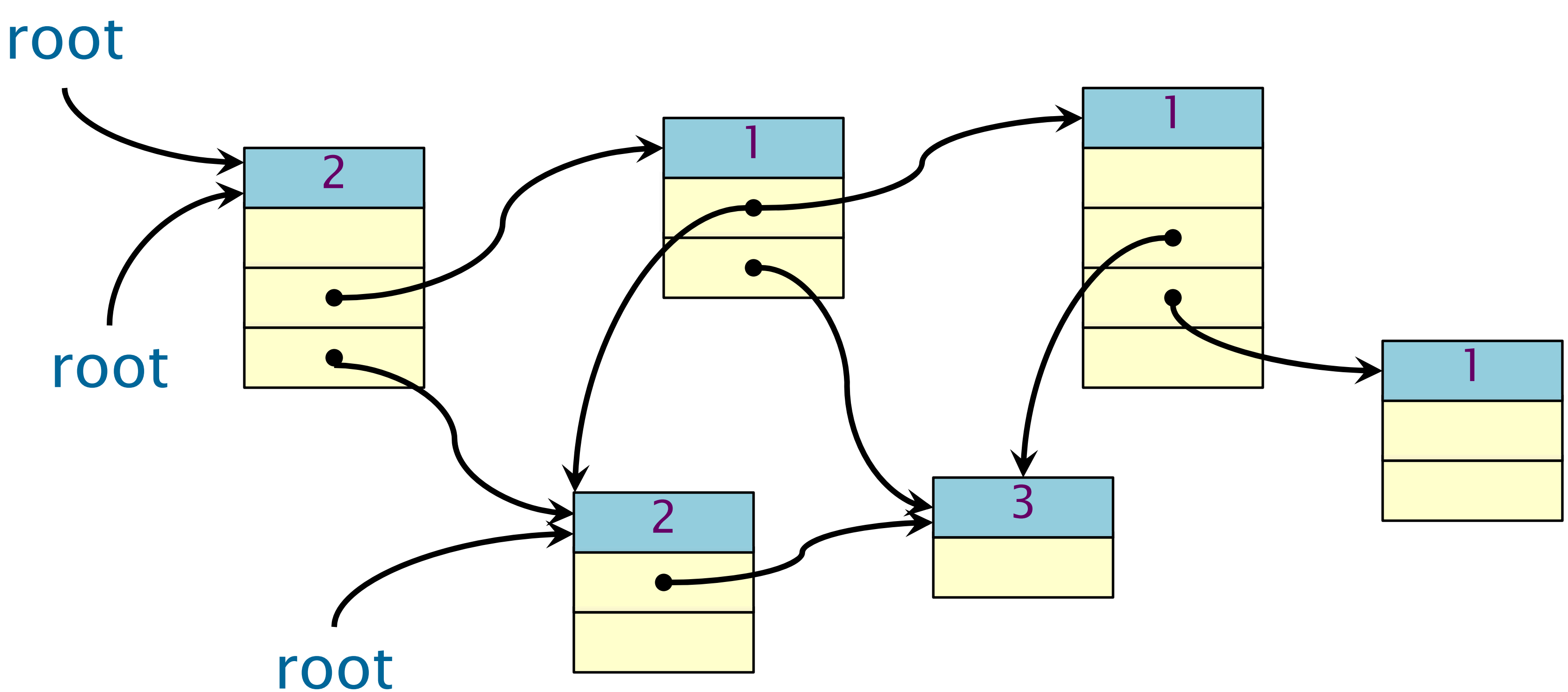
Reference Counting

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



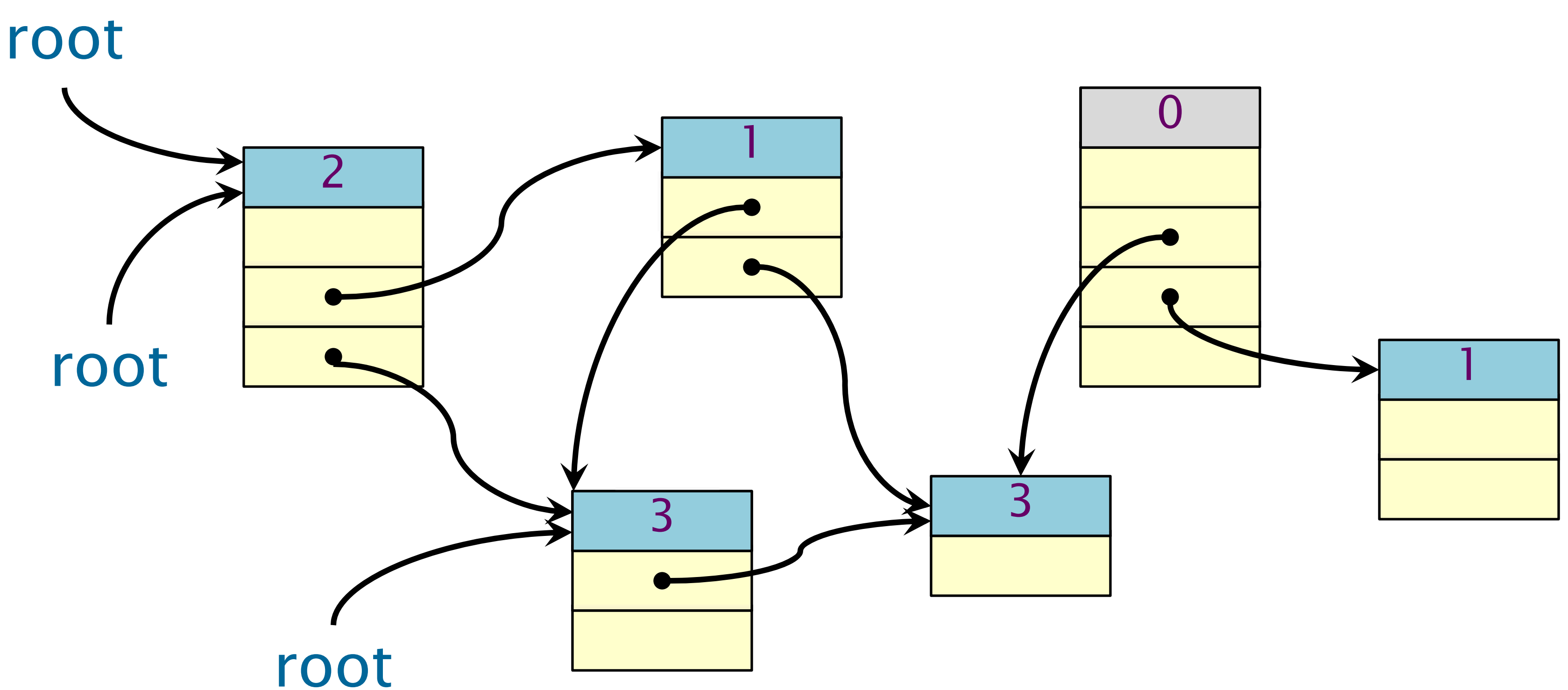
Reference Counting

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



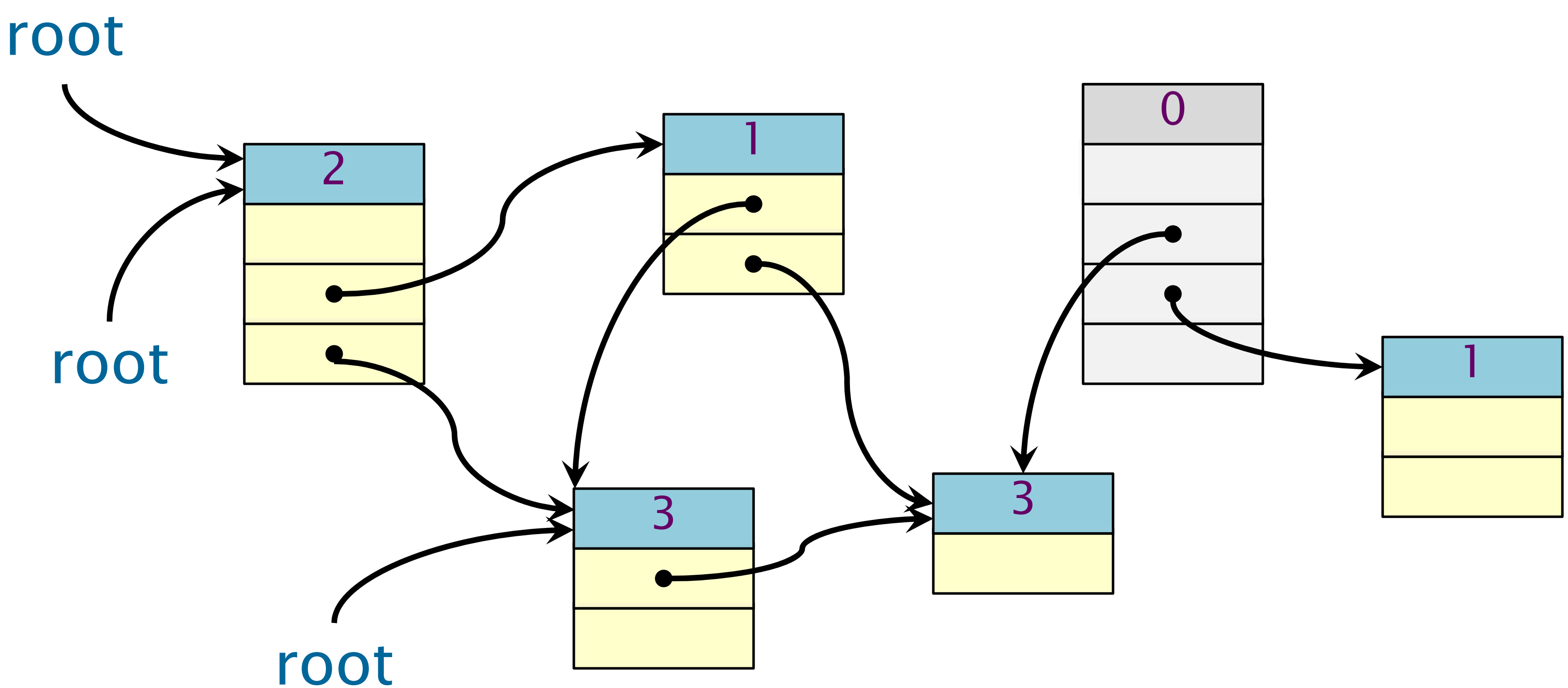
Reference Counting

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



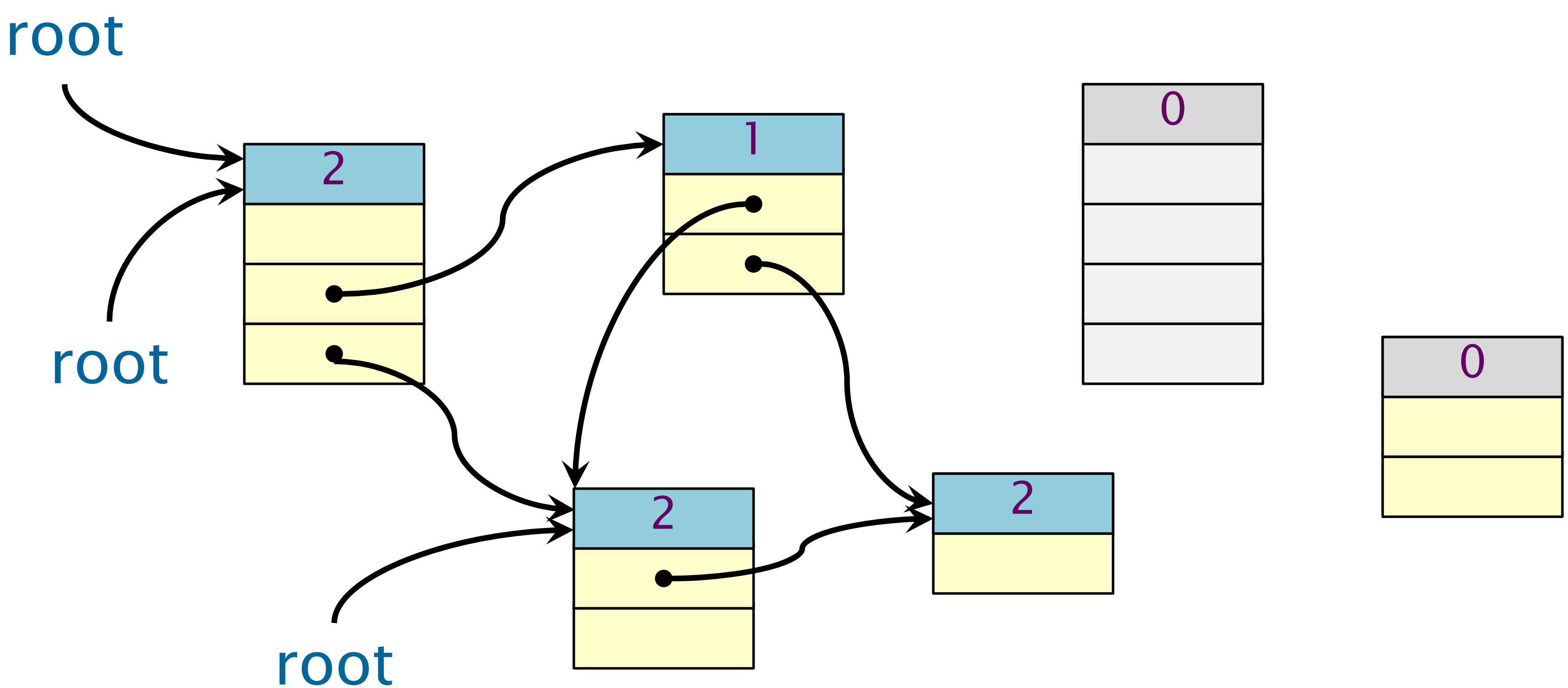
Reference Counting

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



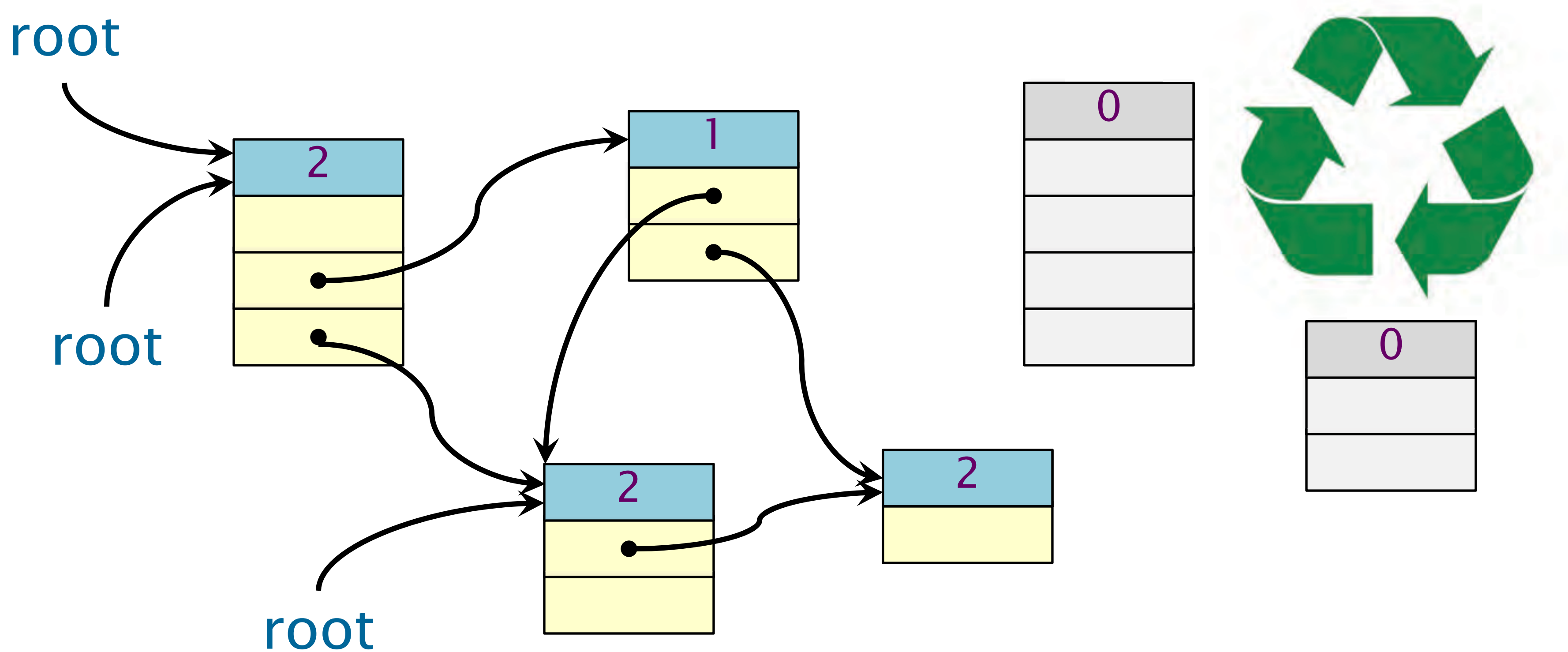
Reference Counting

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



Reference Counting

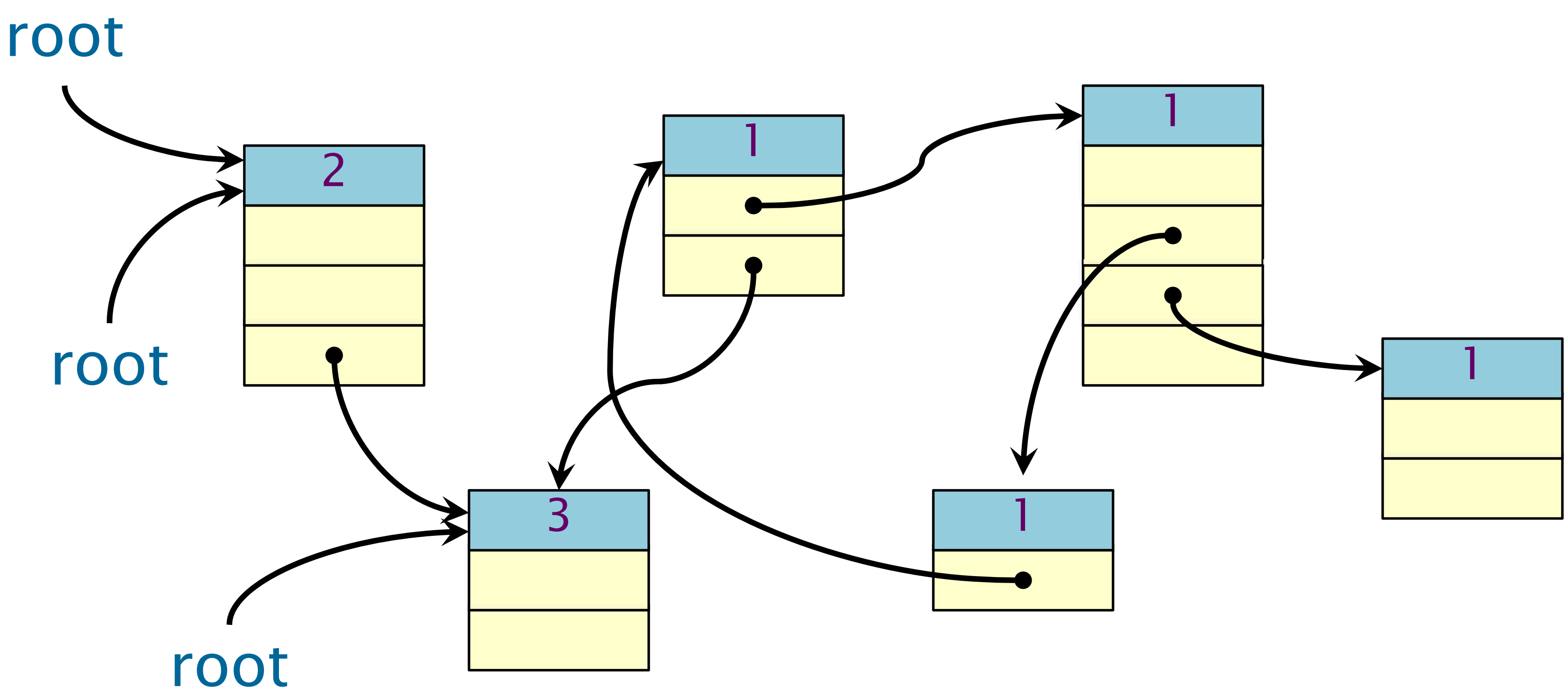
Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.



Limitation of Reference Counting

Problem

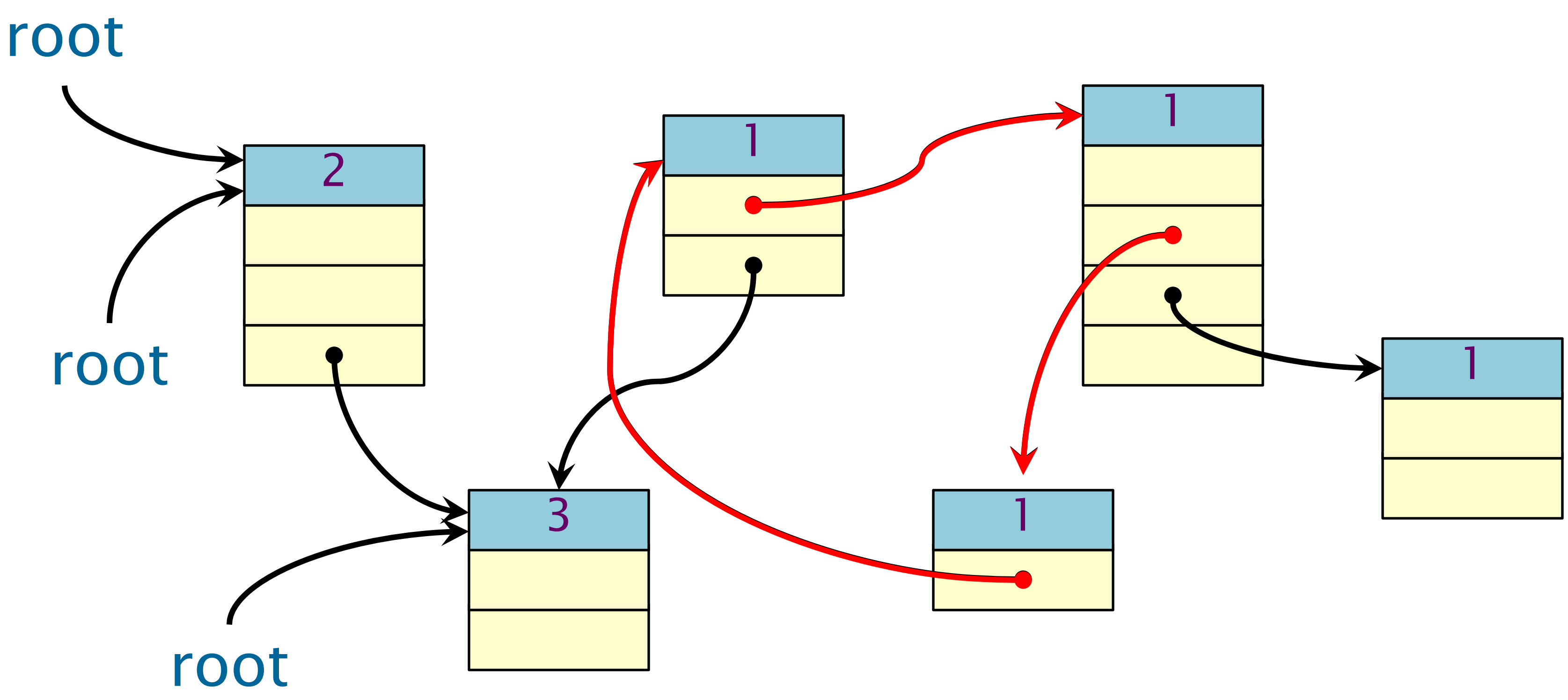
A cycle is never garbage collected!



Limitation of Reference Counting

Problem

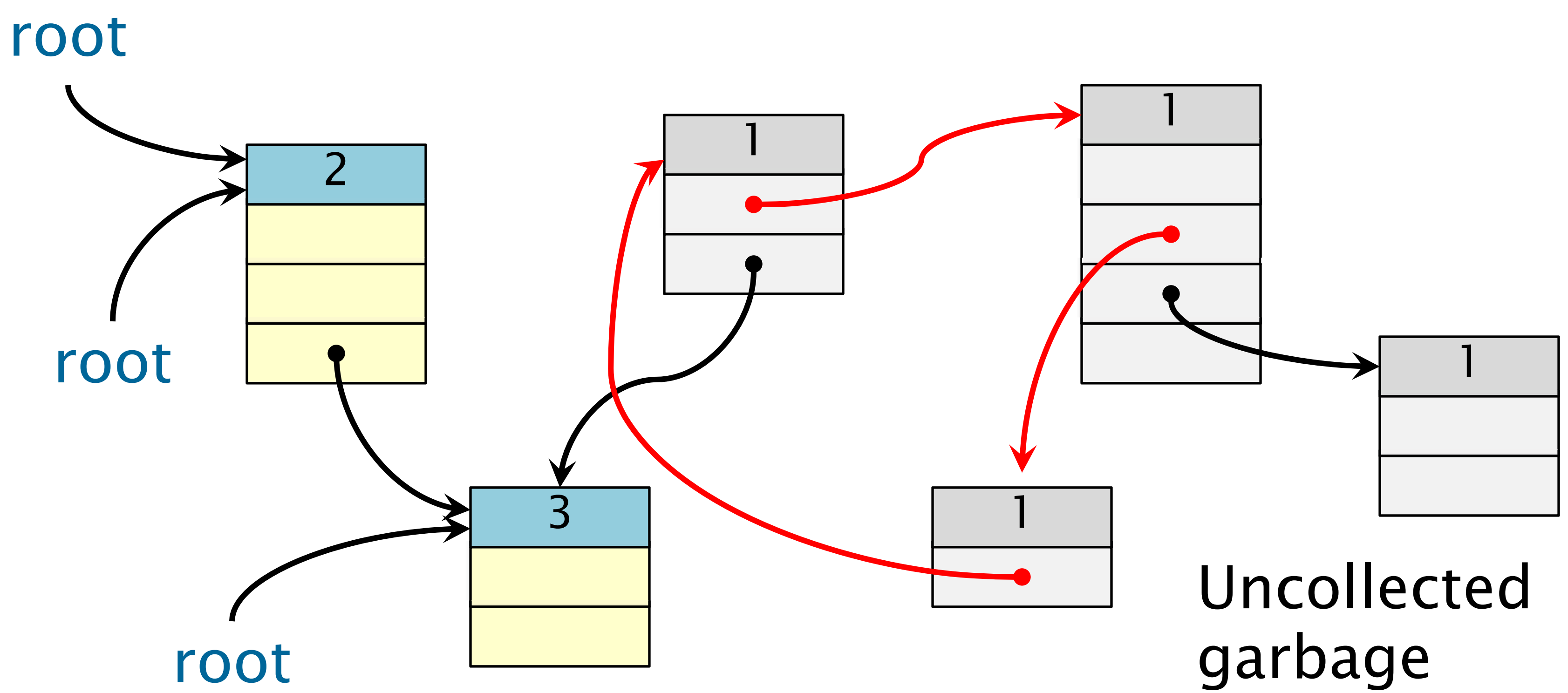
A cycle is never garbage collected!



Limitation of Reference Counting

Problem

A cycle is never garbage collected!



Nevertheless, reference counting works well for acyclic structures.

MARK-AND-SWEEP GARBAGE COLLECTION

Graph Abstraction

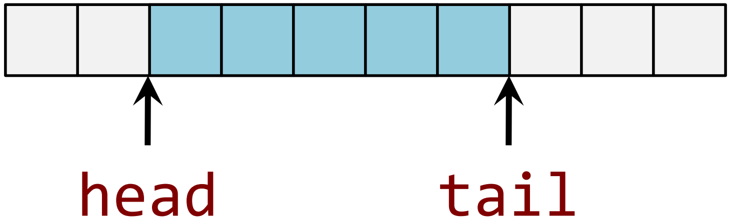
Idea

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.

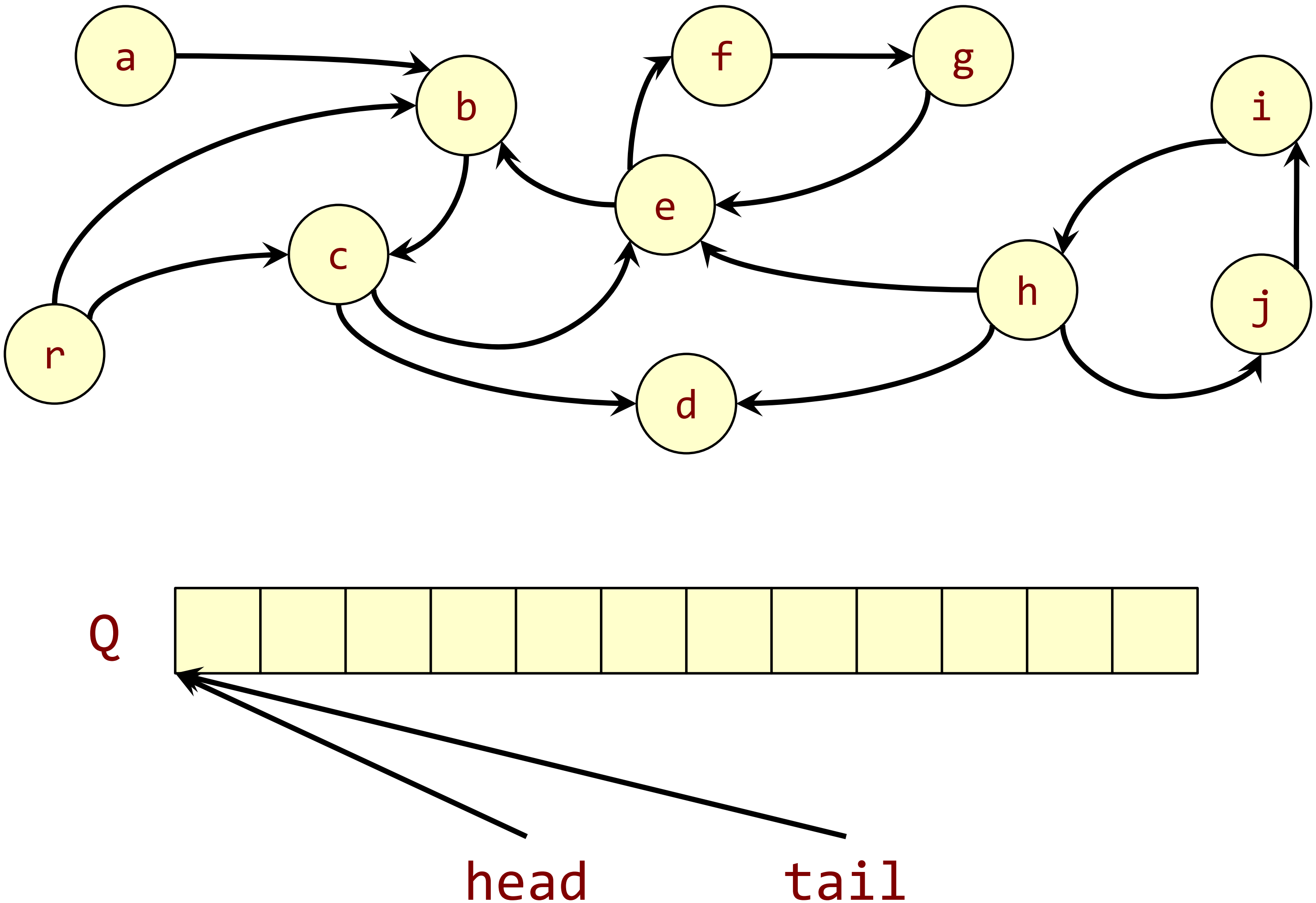
```
for (∀ v∈V) {
  if (root(v)) {
    v.mark = 1;
    enqueue(Q, v);
  } else v.mark = 0;

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

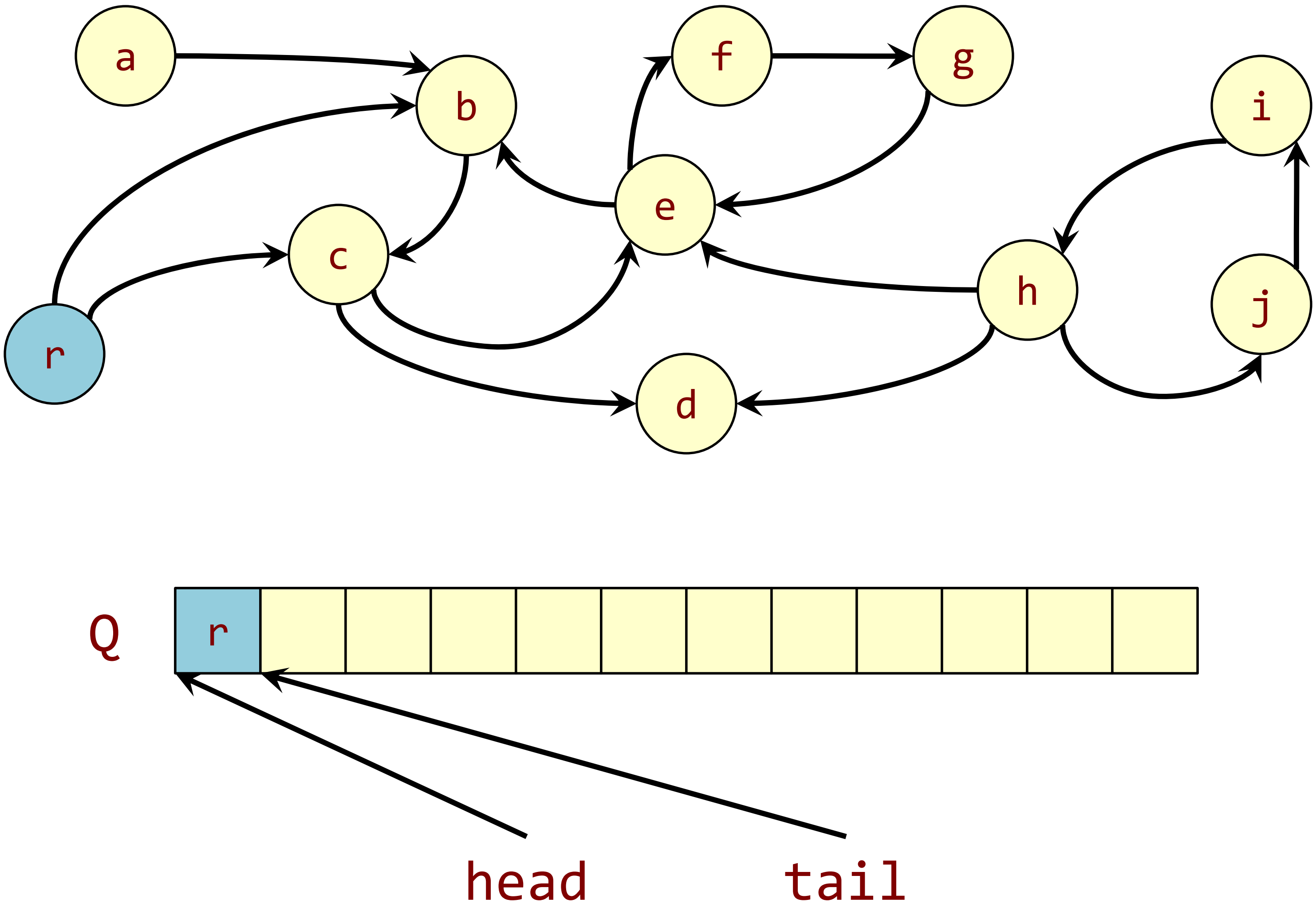
FIFO queue Q



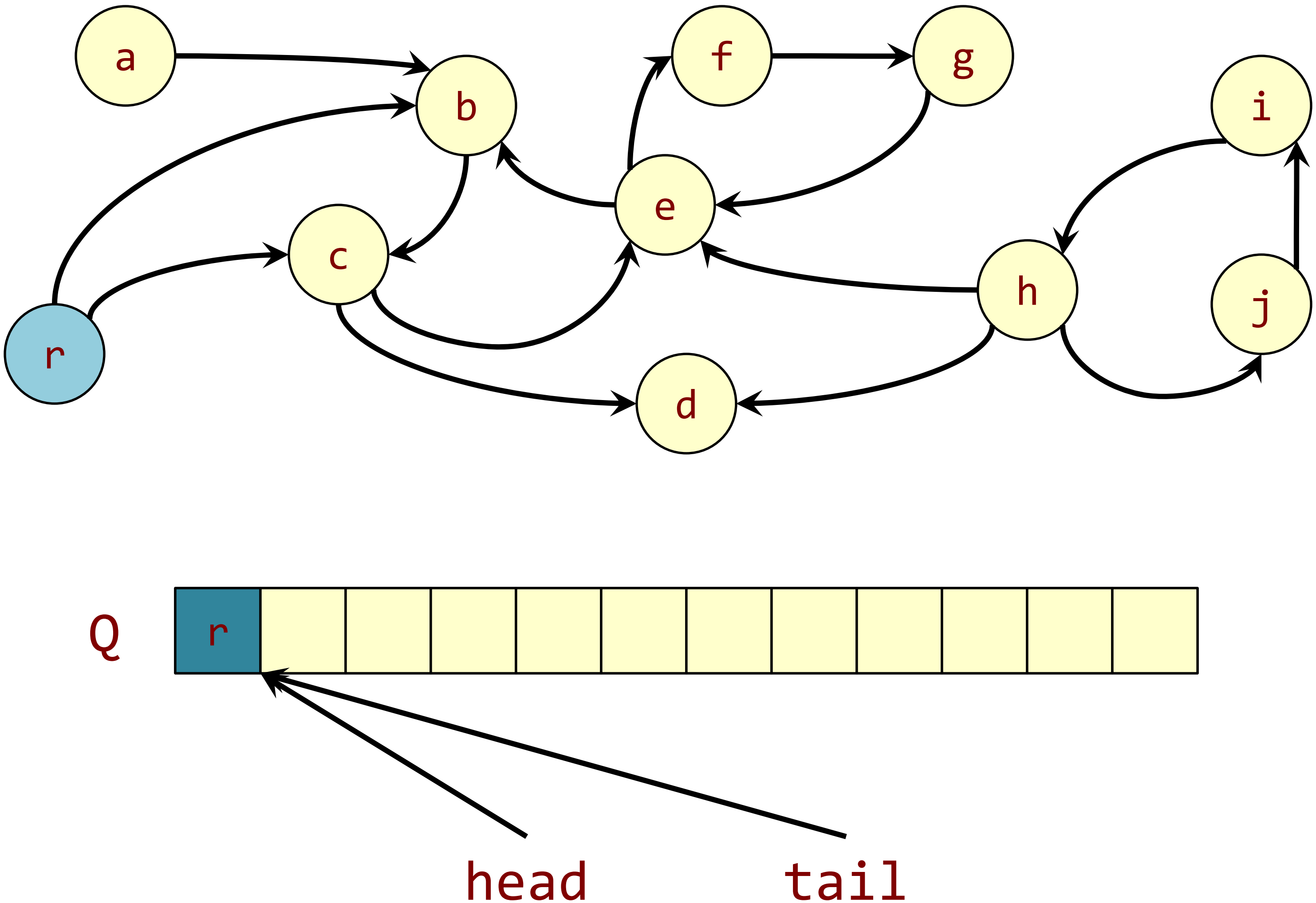
Breadth-First Search



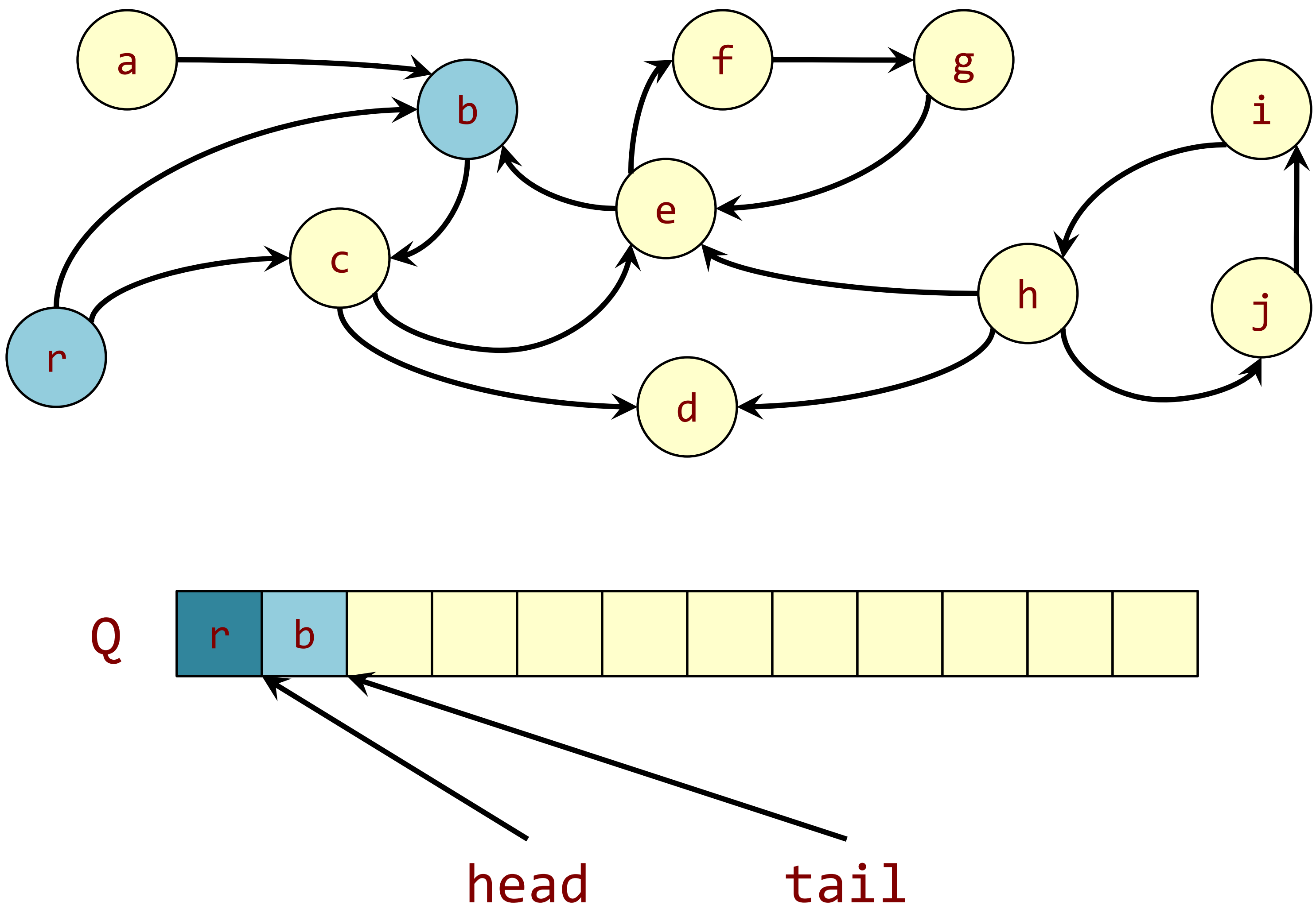
Breadth-First Search



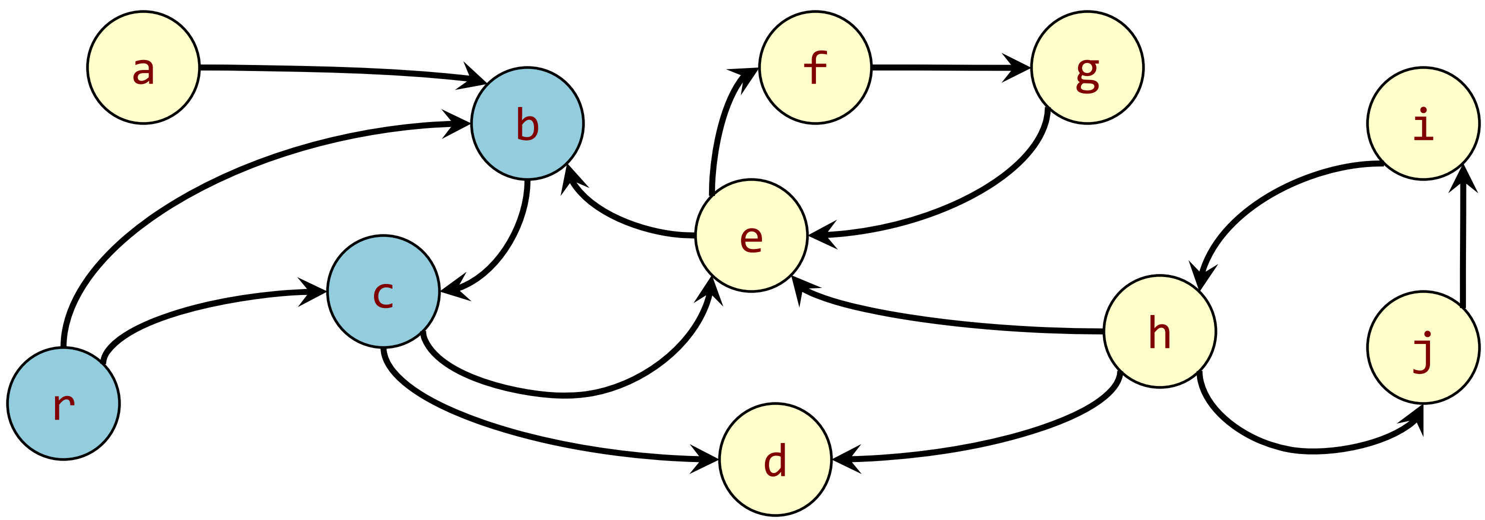
Breadth-First Search

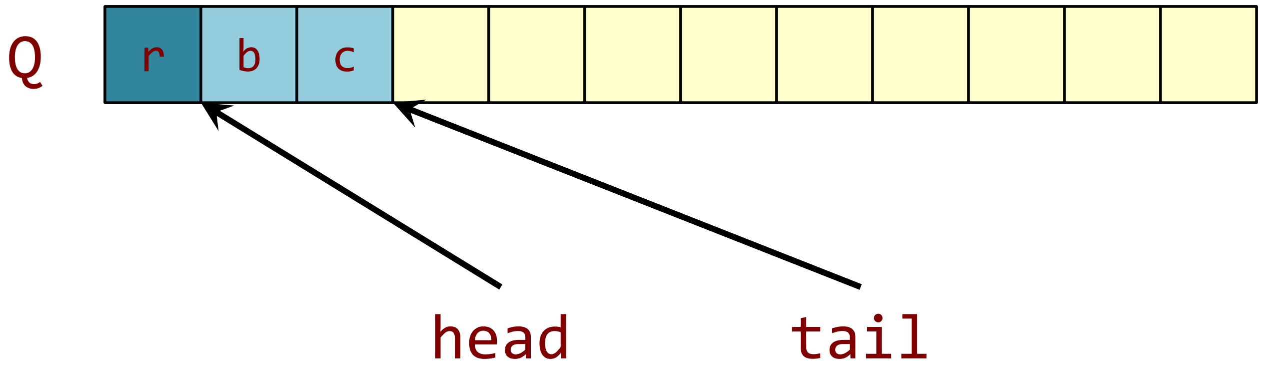


Breadth-First Search

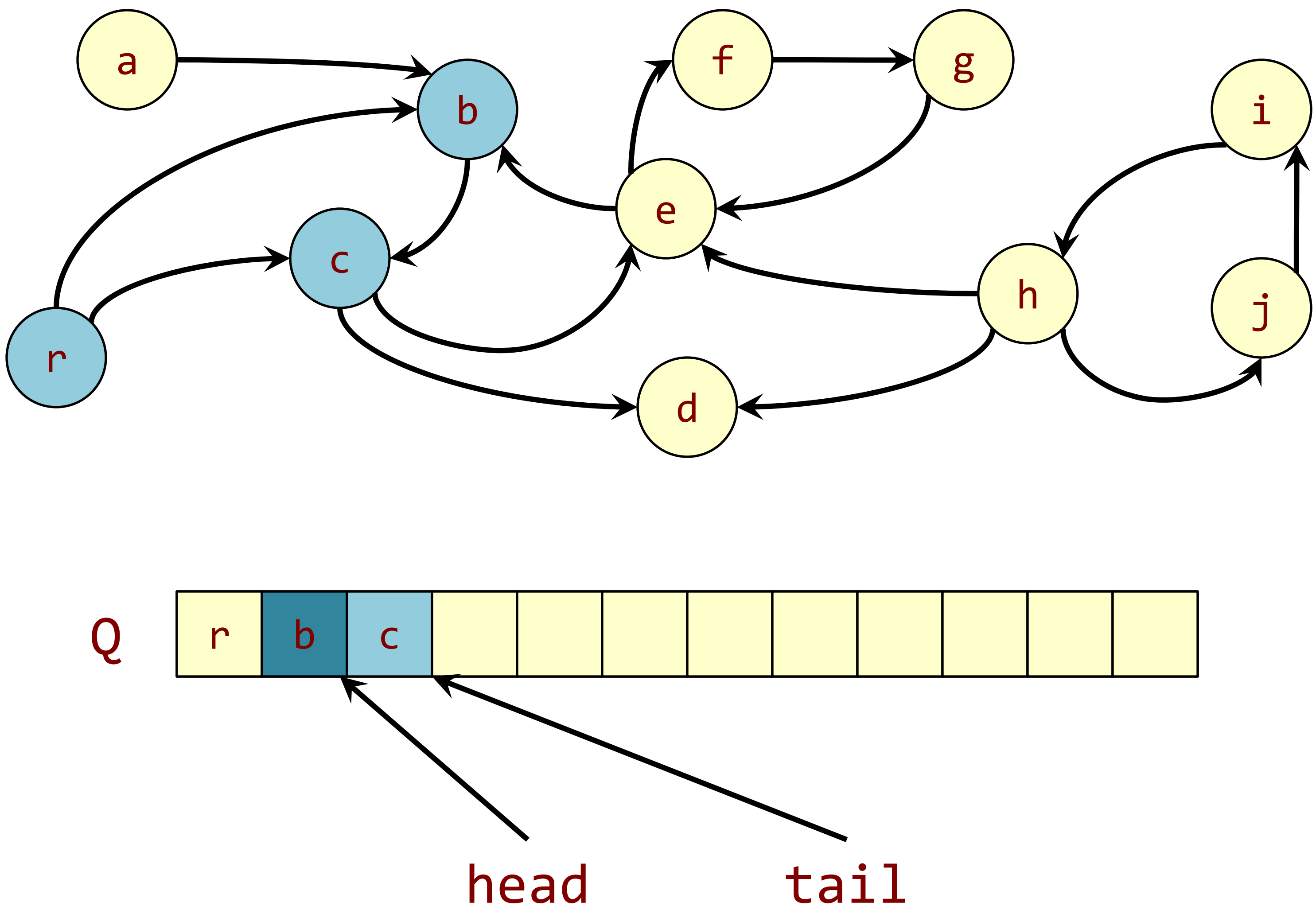


Breadth-First Search

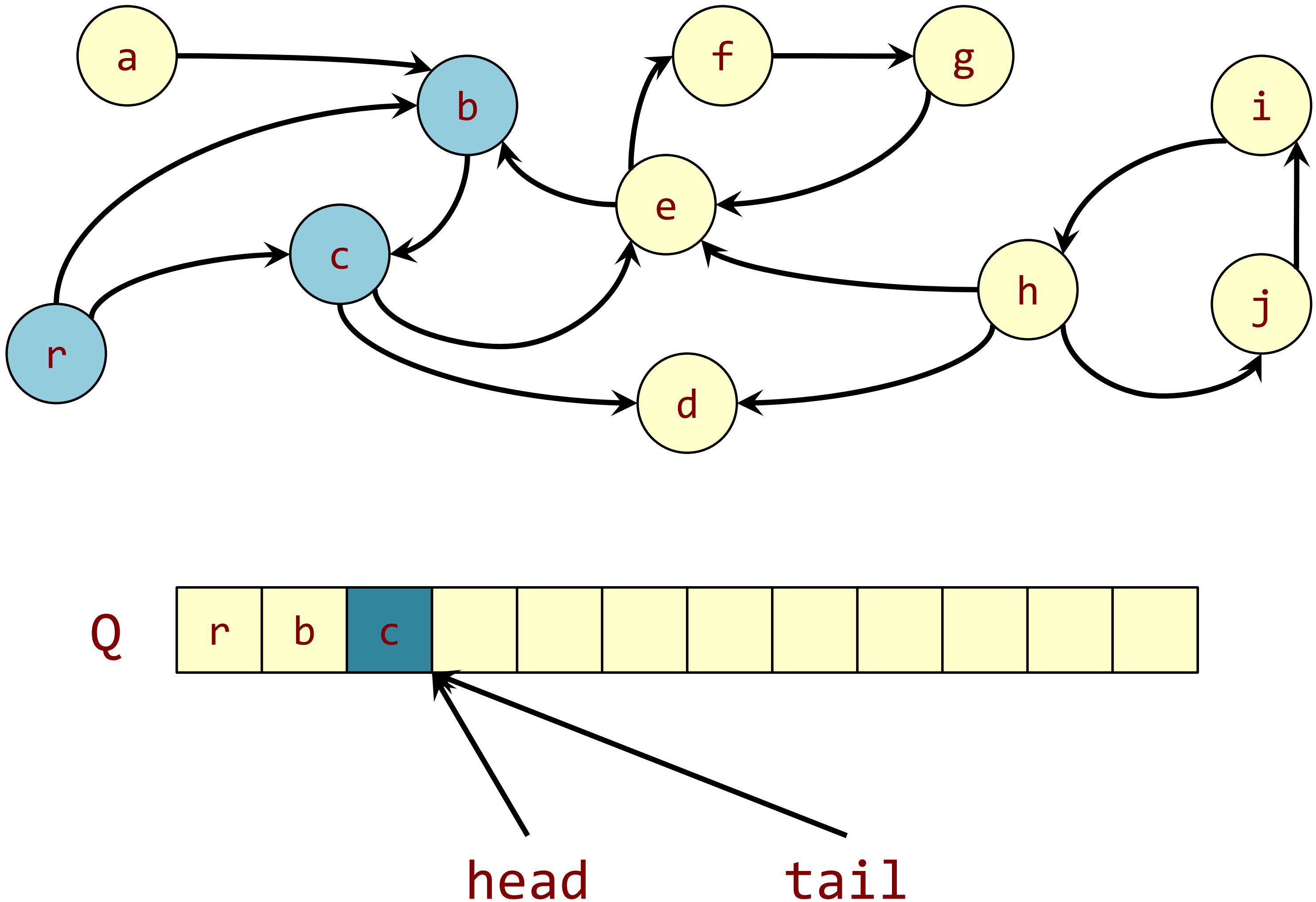




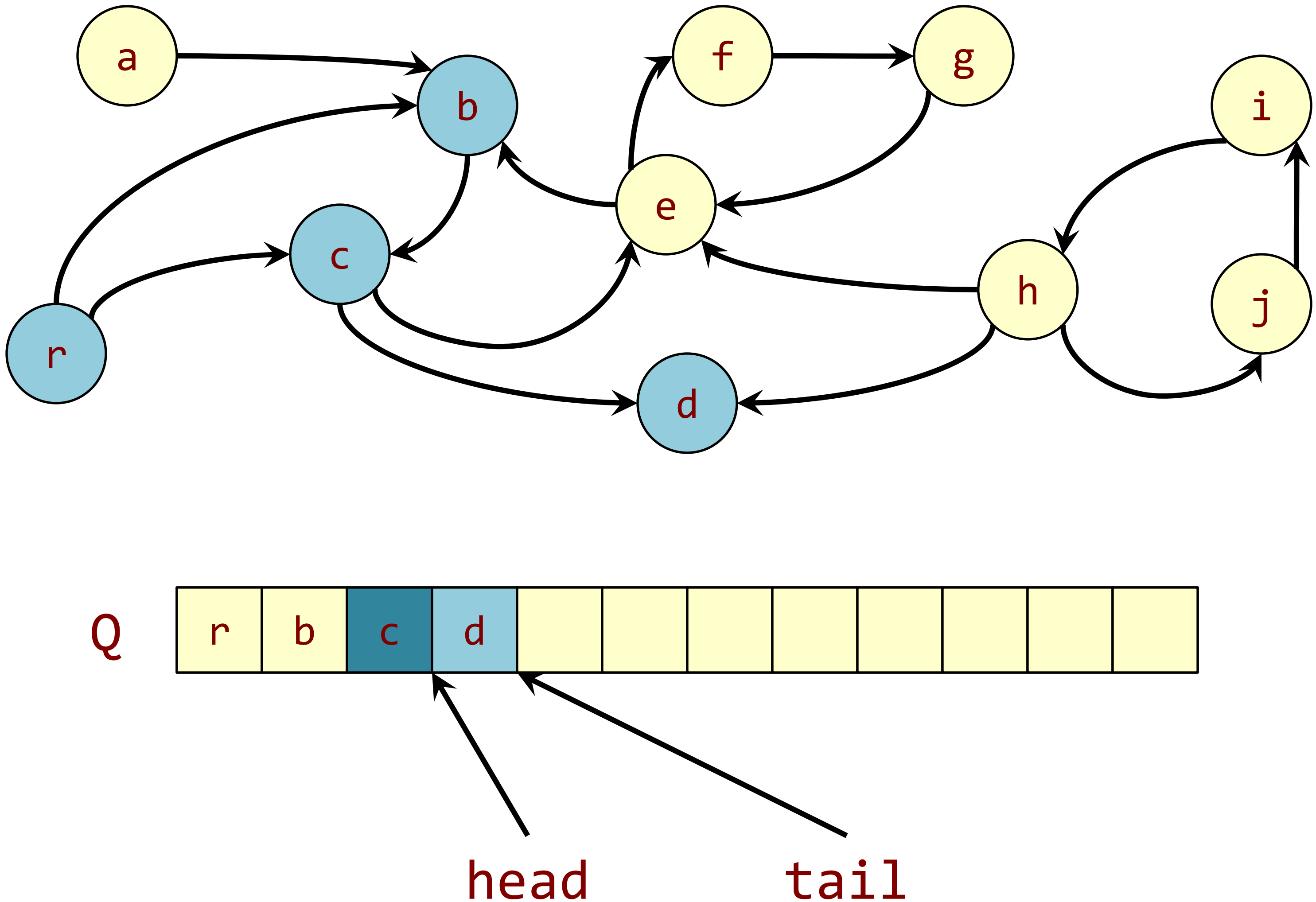
Breadth-First Search



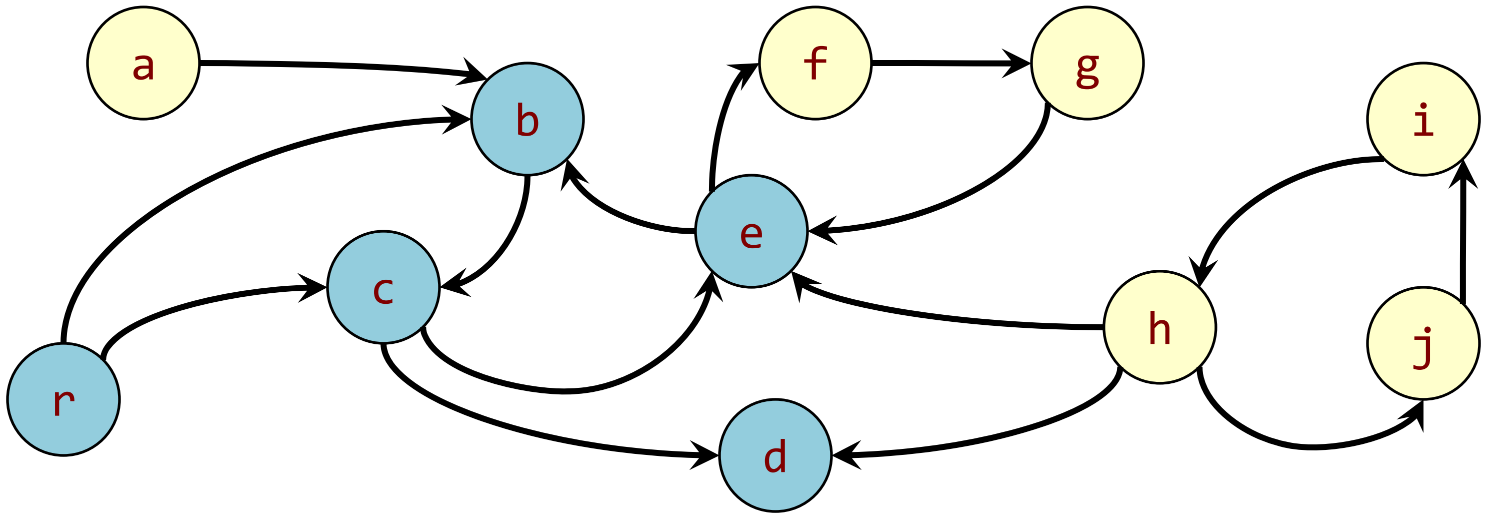
Breadth-First Search

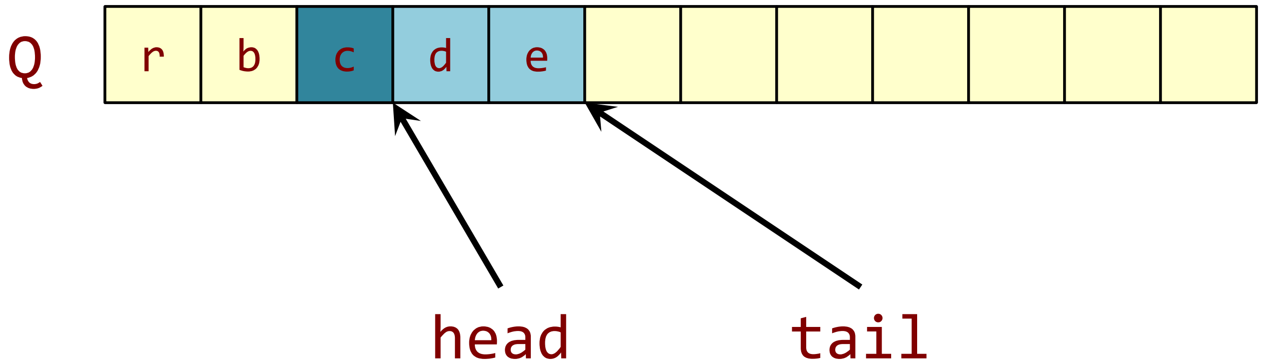


Breadth-First Search

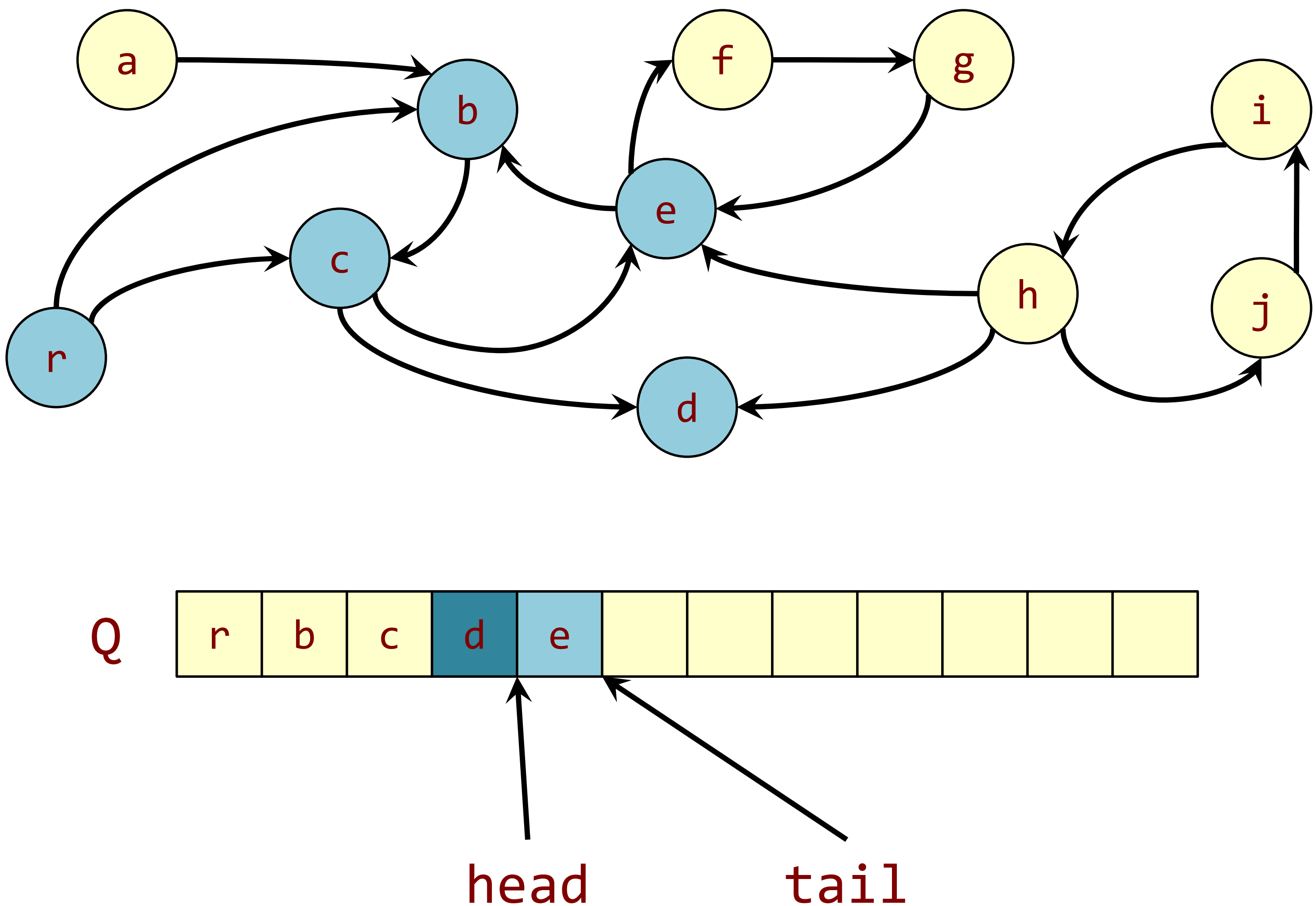


Breadth-First Search

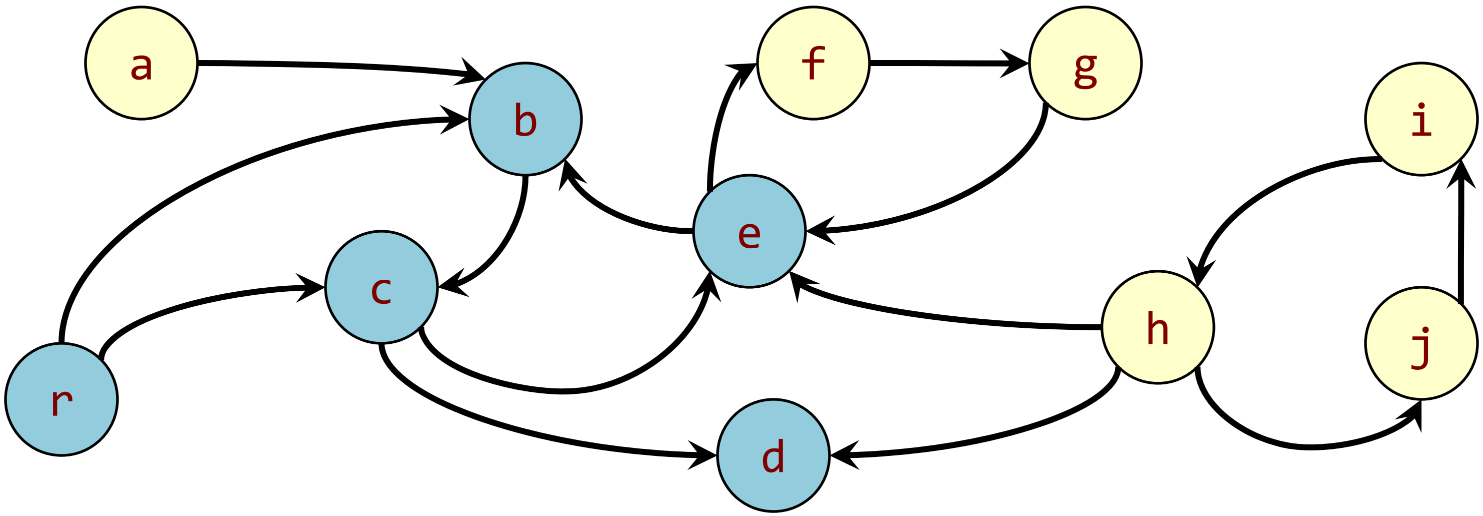


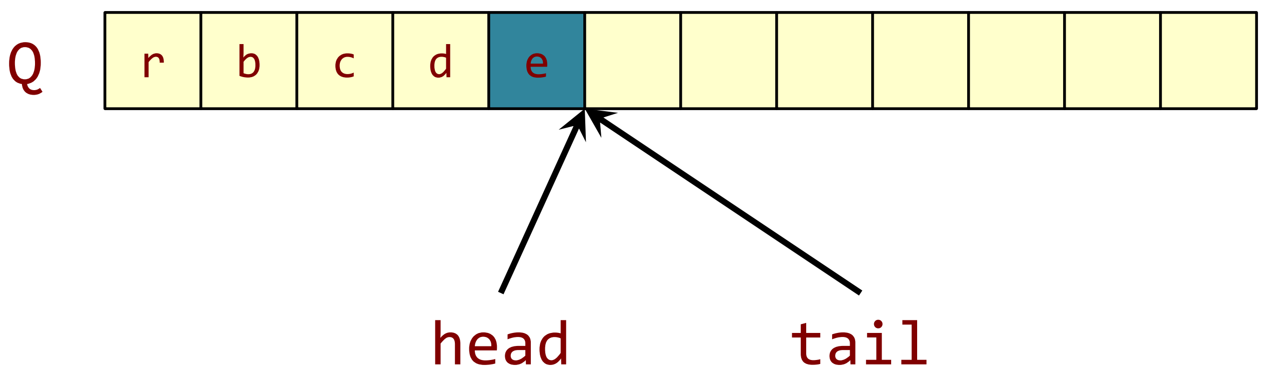


Breadth-First Search

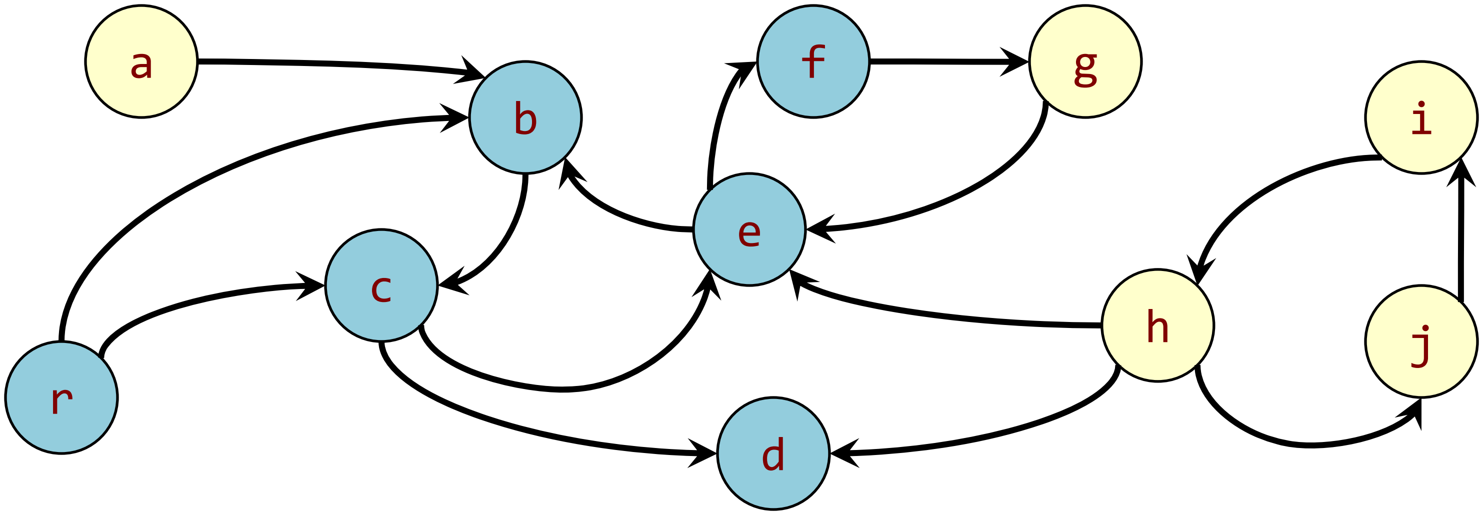


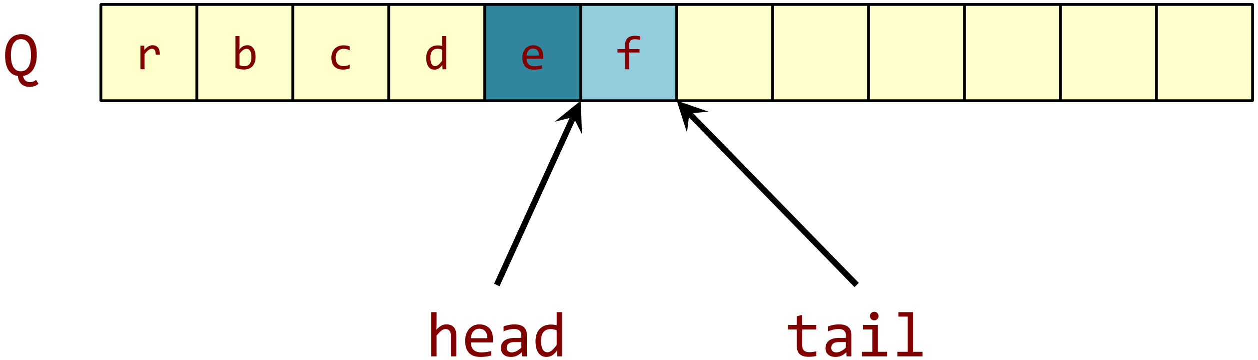
Breadth-First Search



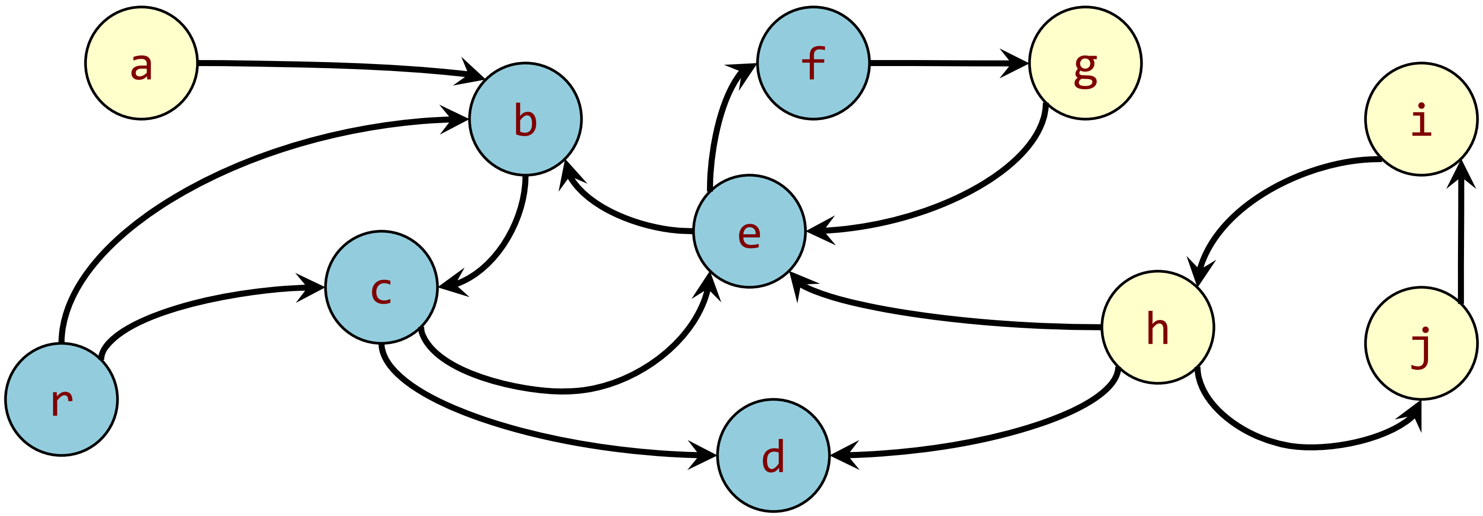


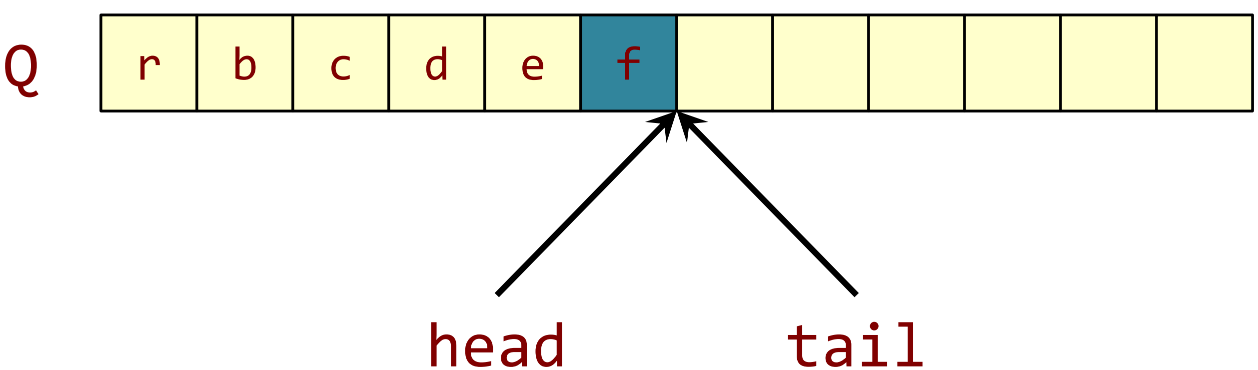
Breadth-First Search



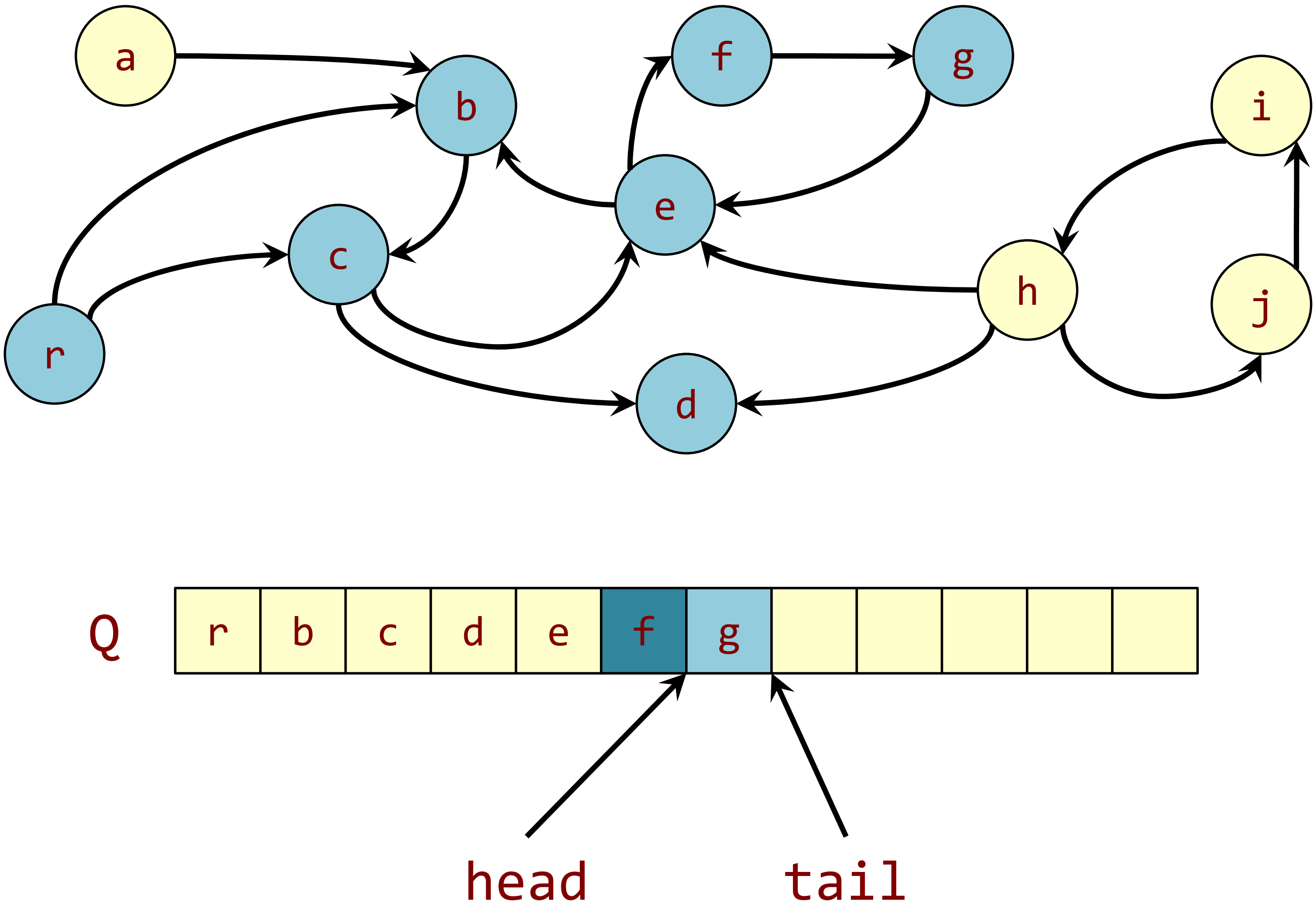


Breadth-First Search

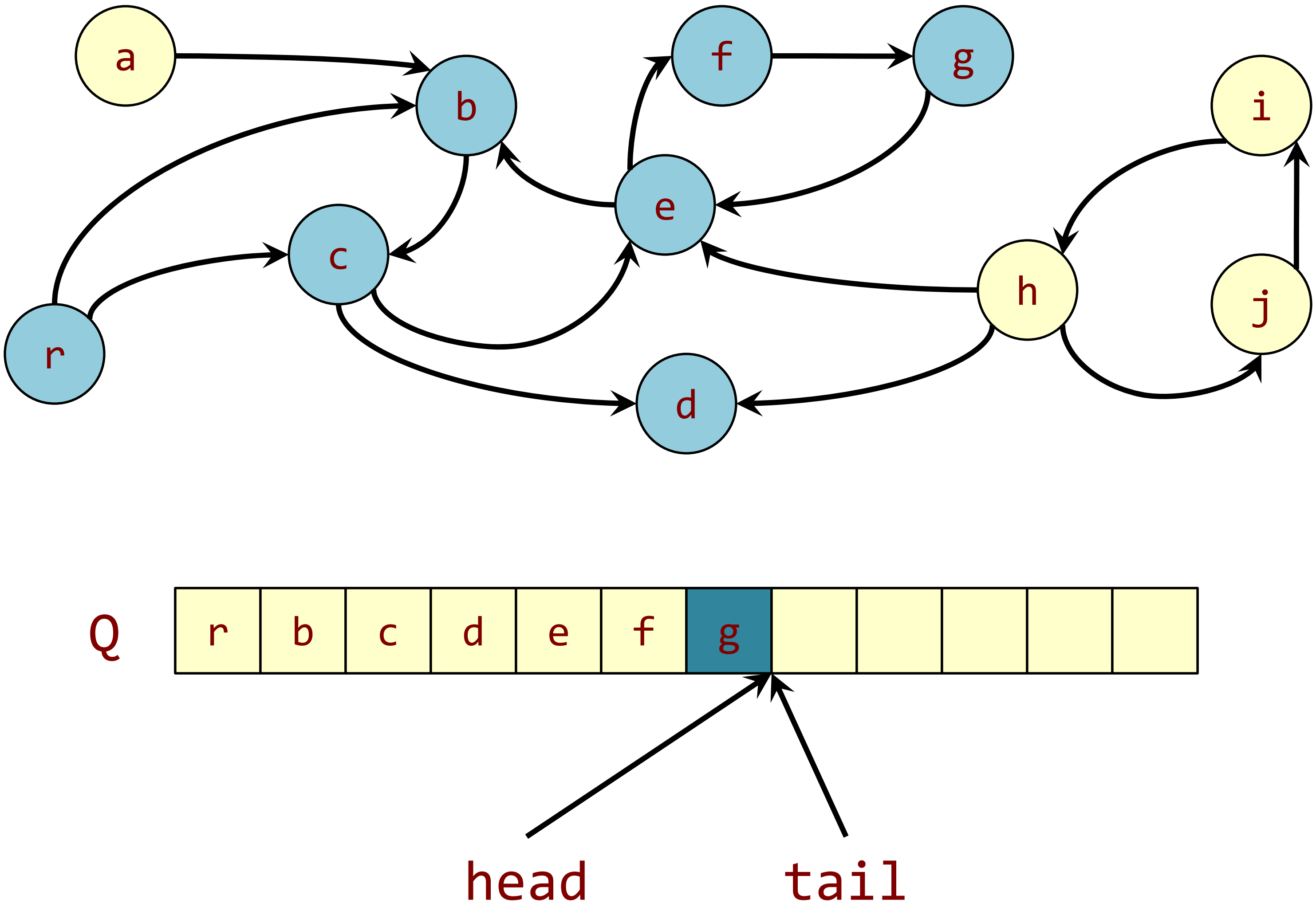




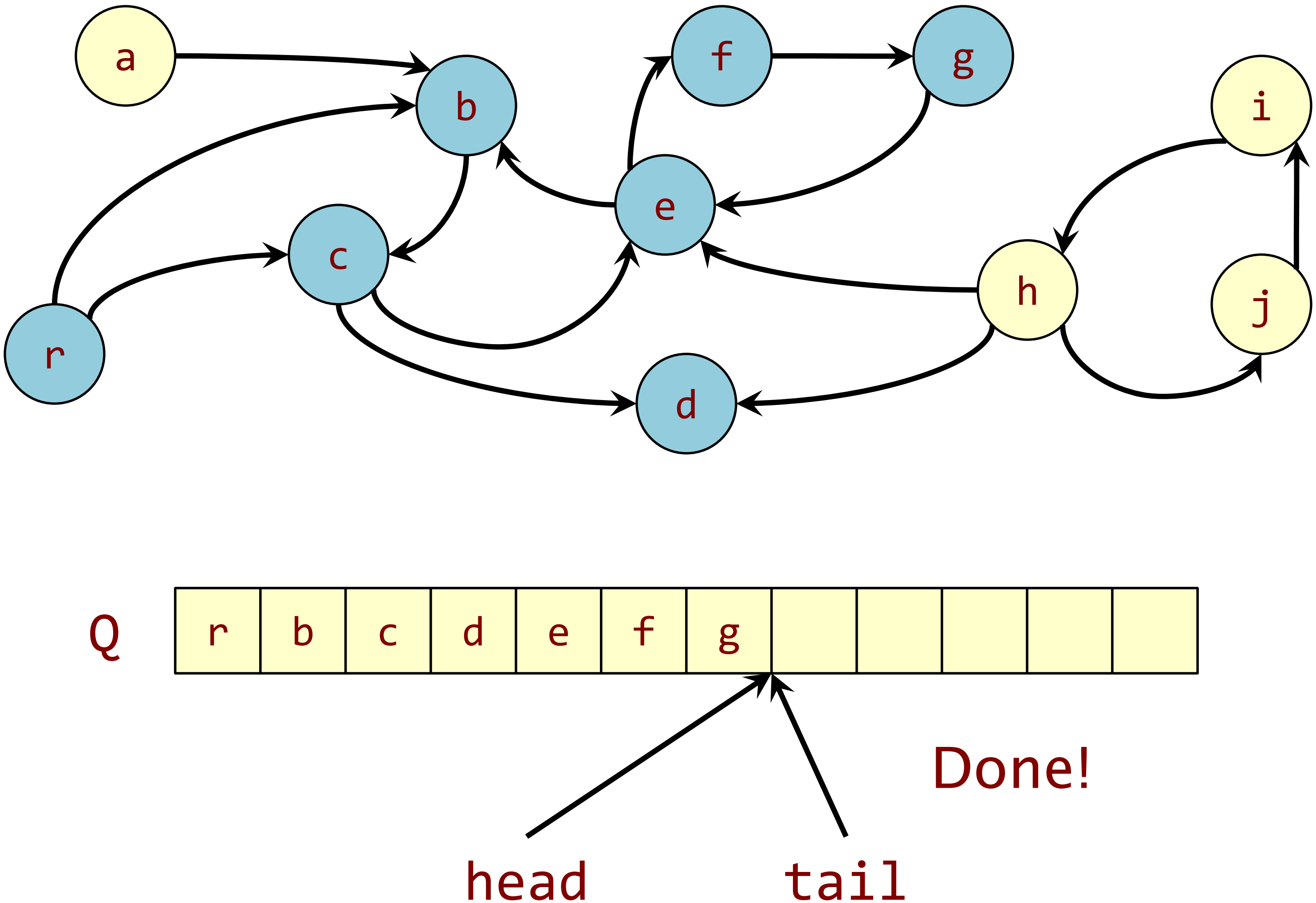
Breadth-First Search



Breadth-First Search



Breadth-First Search



Mark-and-Sweep

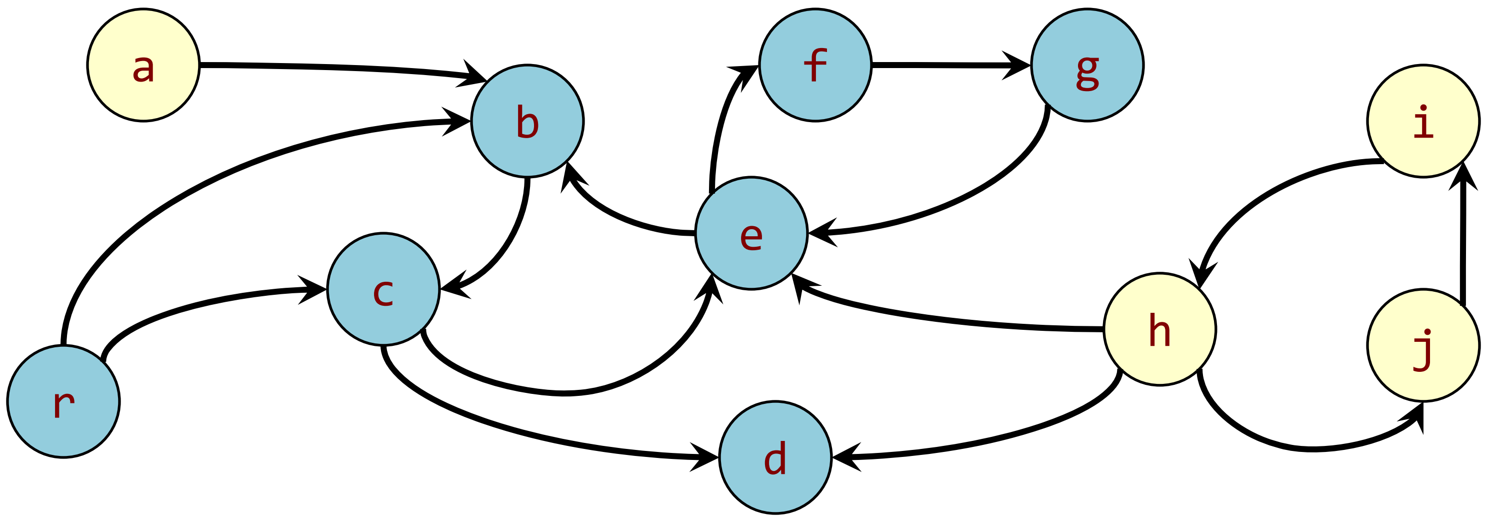
Mark stage: Breadth-first search marked all of the live objects.

Sweep stage: Scan over memory to free unmarked objects.

Mark-and-sweep doesn't deal with fragmentation

Stop-and-Copy Garbage Collection

Breadth-First Search

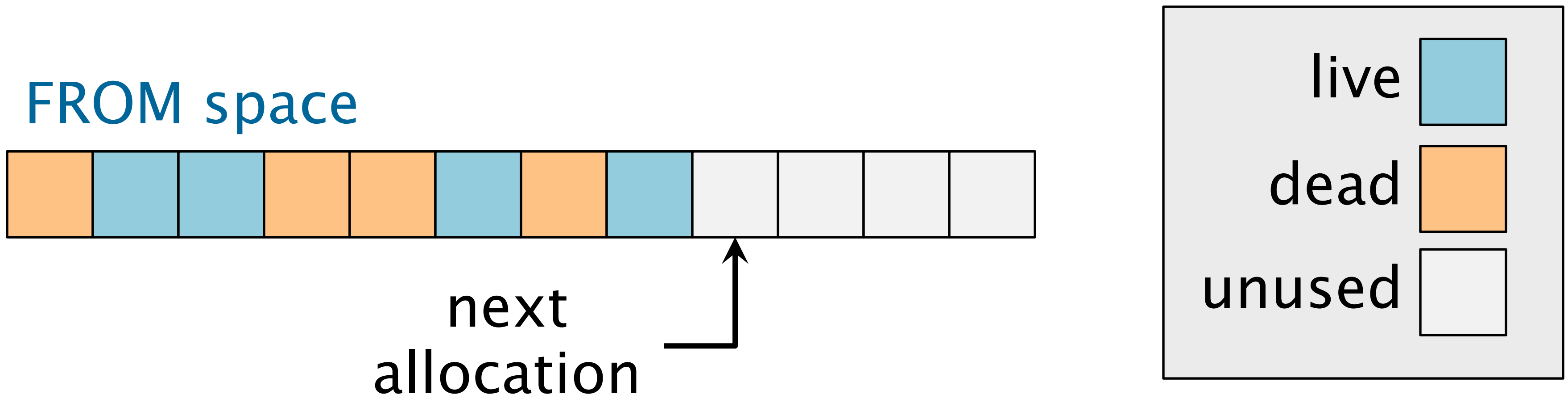


Q	r	b	c	d	e	f	g					

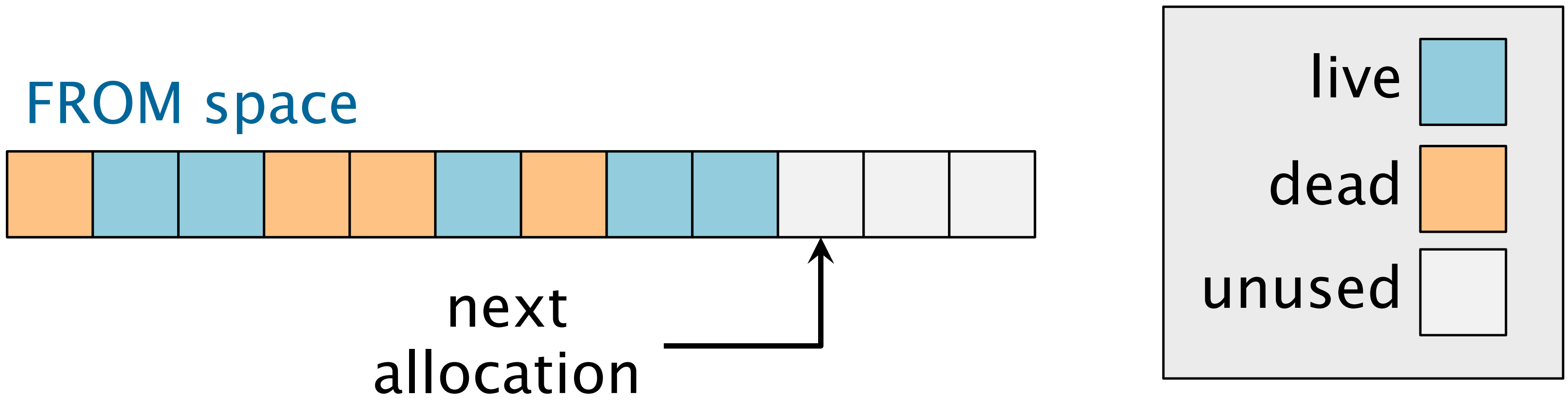
Observation

All live vertices are placed in contiguous storage in Q.

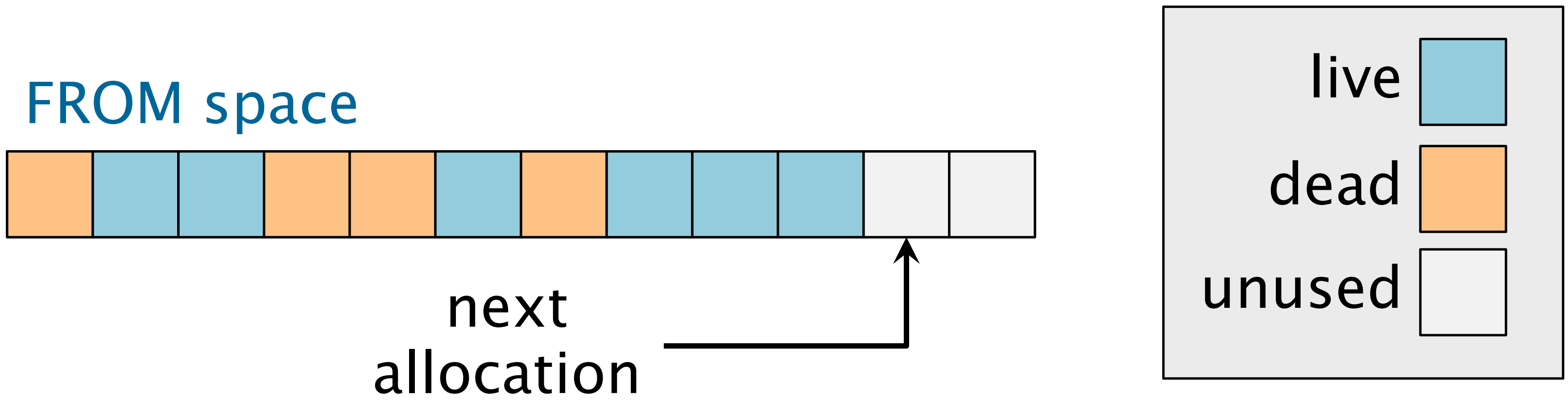
Copying Garbage Collector



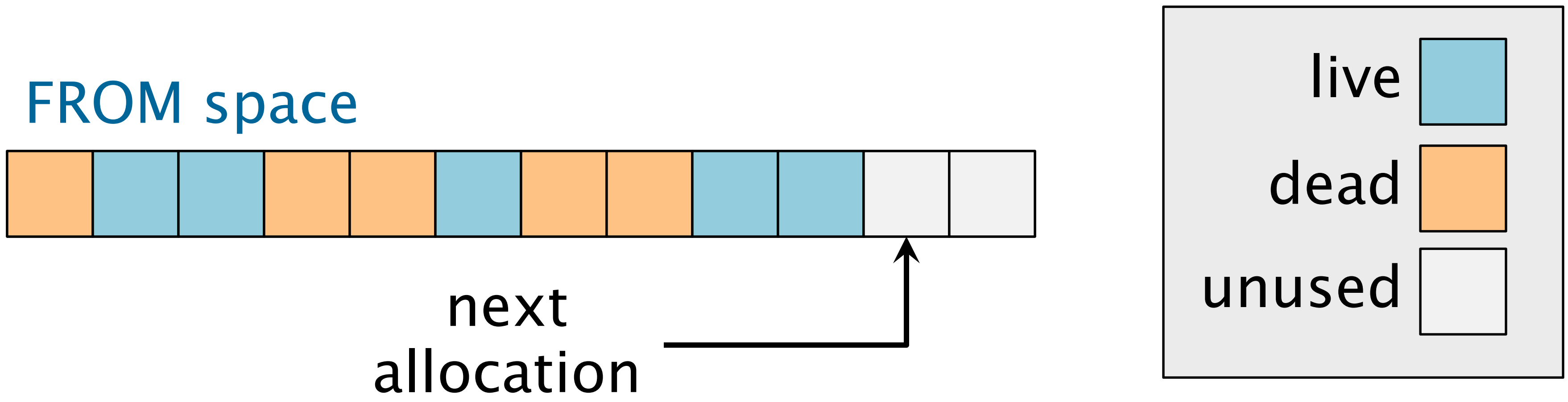
Copying Garbage Collector



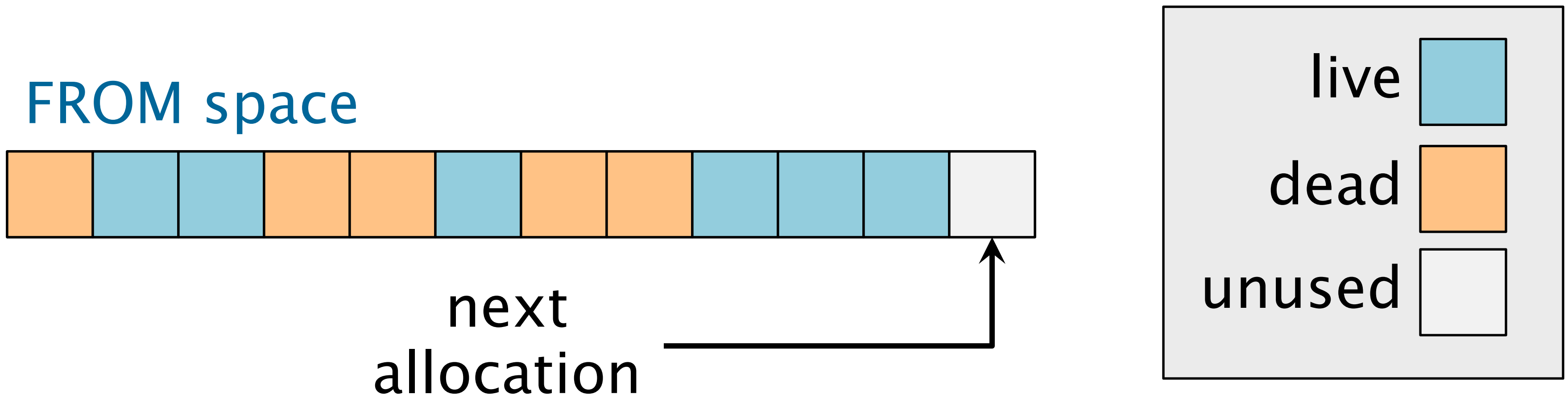
Copying Garbage Collector



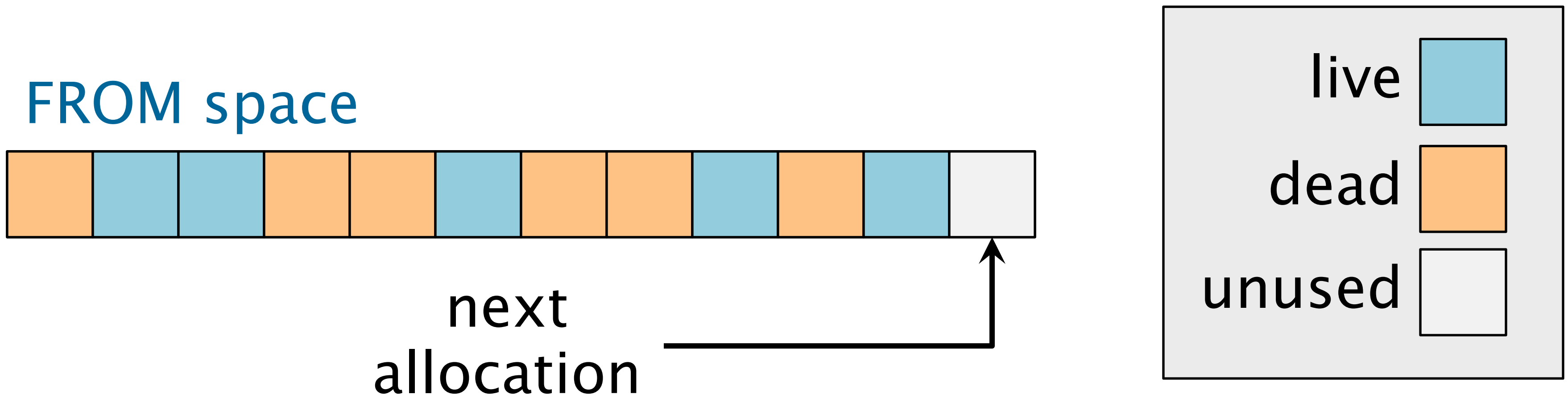
Copying Garbage Collector



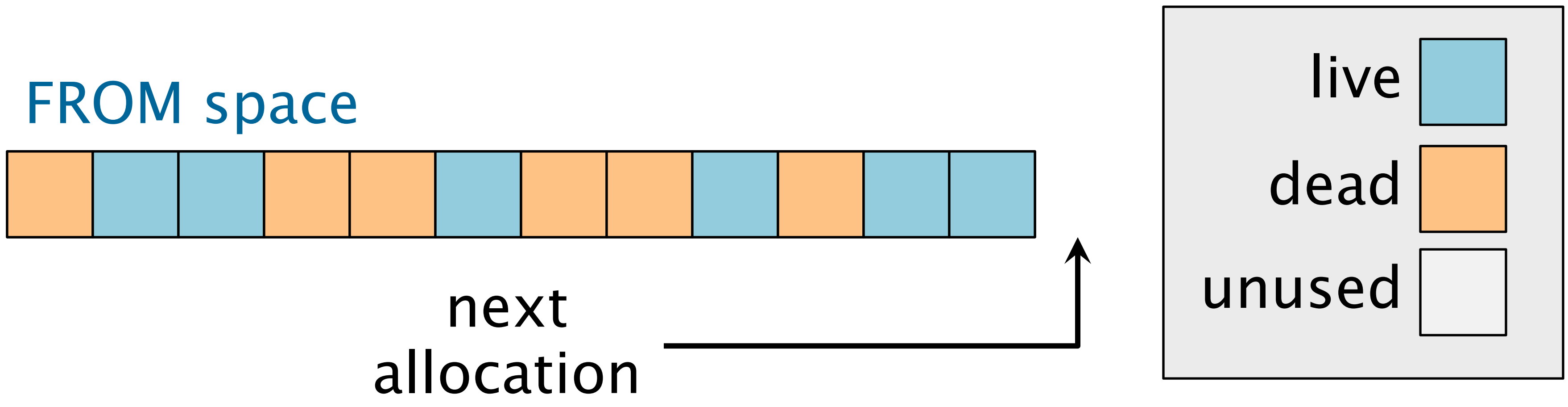
Copying Garbage Collector



Copying Garbage Collector

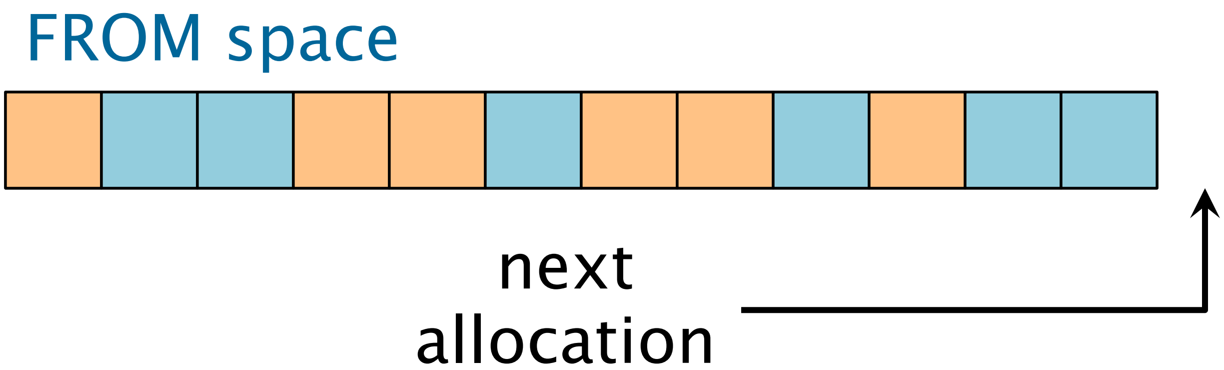


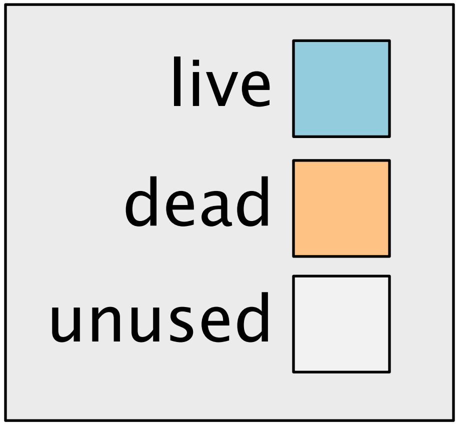
Copying Garbage Collector



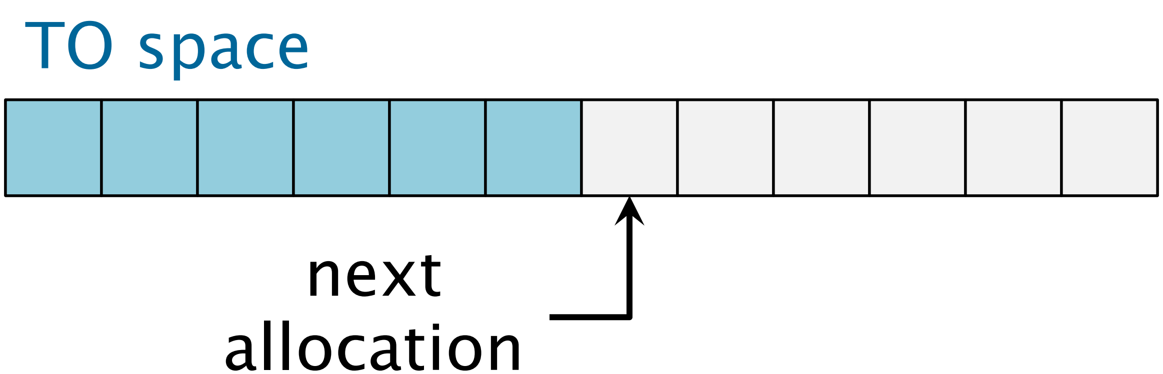
When the FROM space is "full," copy live storage using BFS with the TO space as the FIFO queue.

Copying Garbage Collector





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

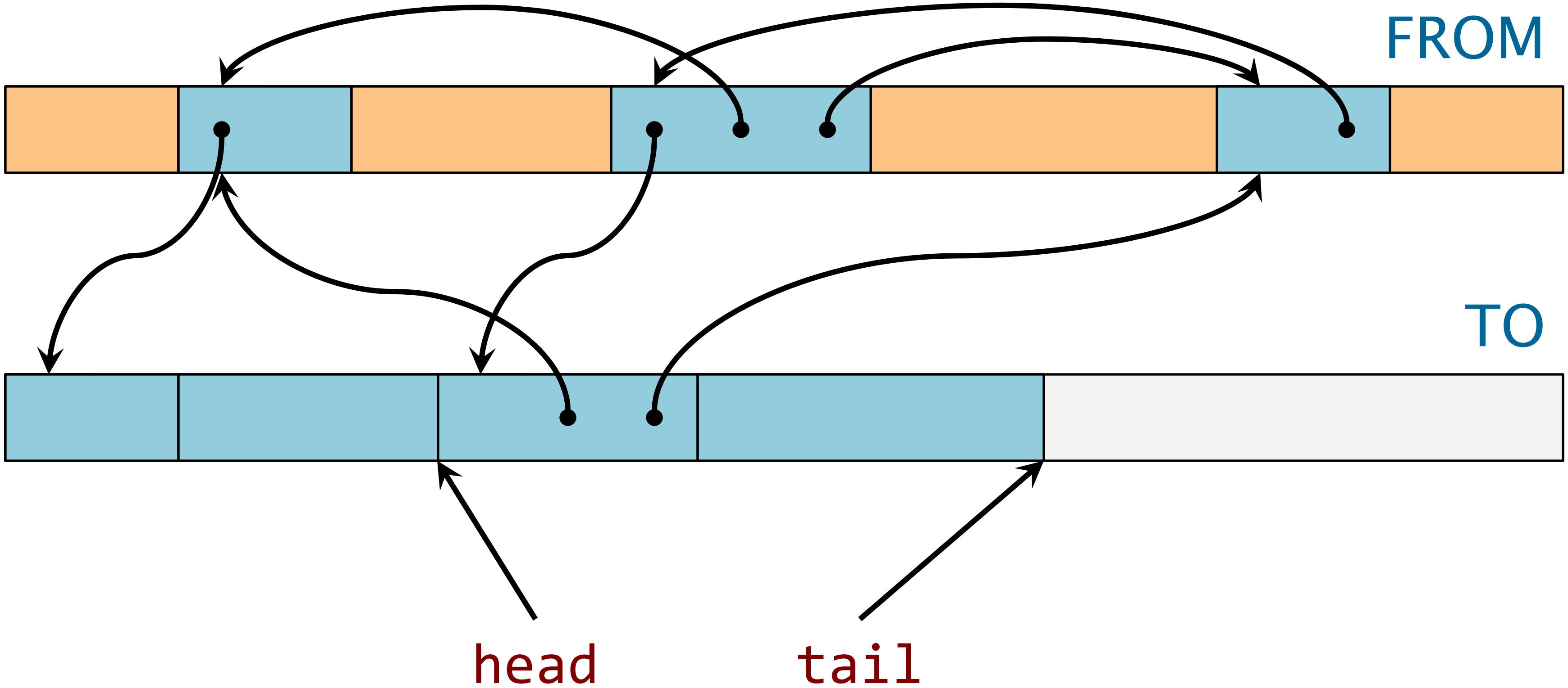


Updating Pointers

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

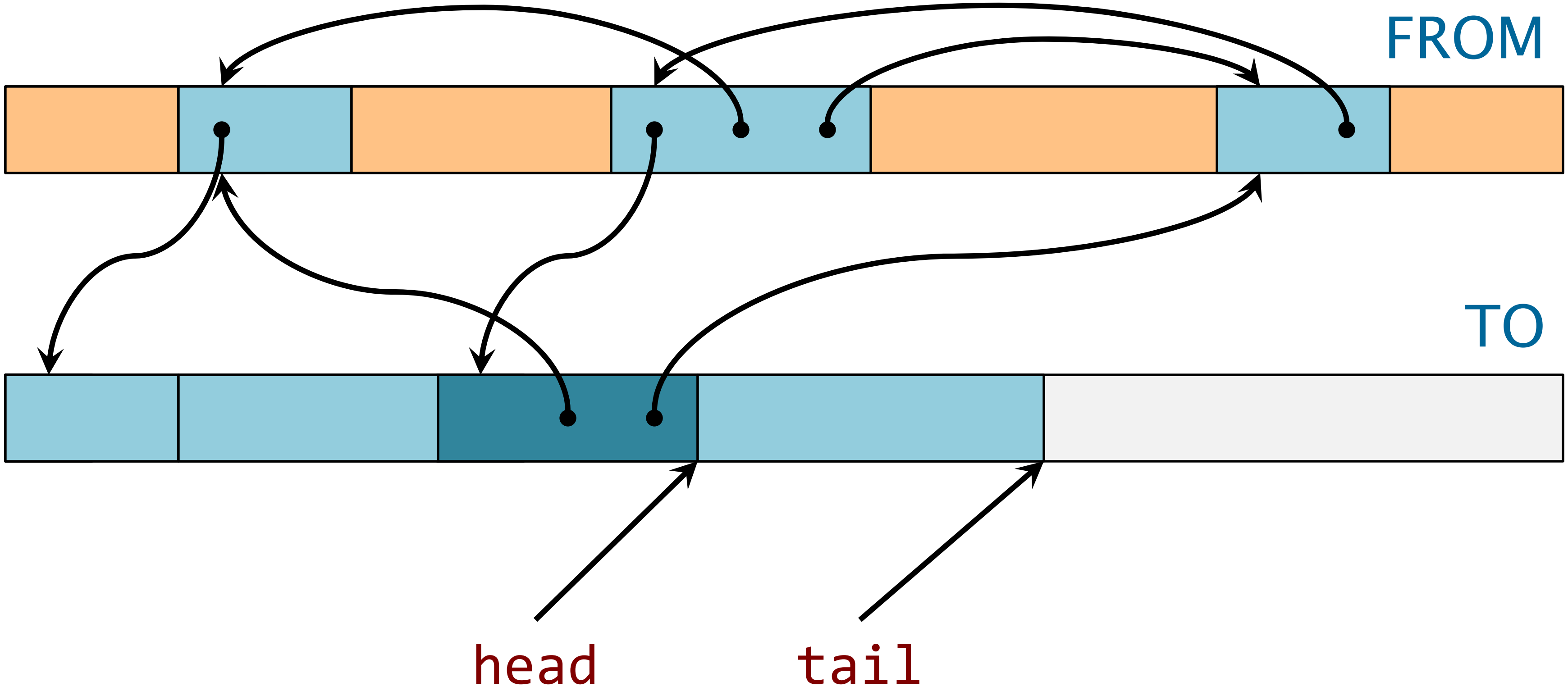
- When an object is copied to the TO space, store a forwarding pointer in the FROM object, which implicitly marks it as moved.
- When an object is removed from the FIFO queue in the TO space, update all its pointers.

Example



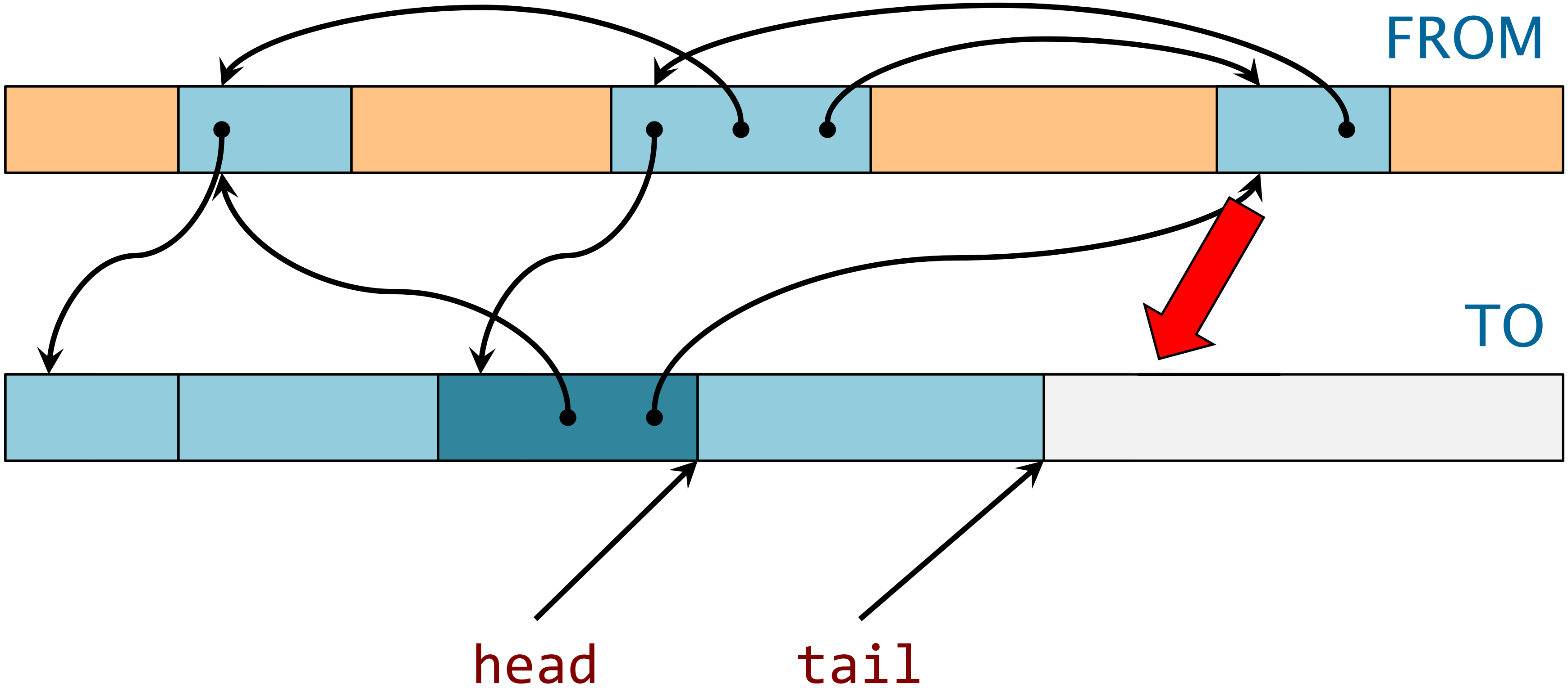
Remove an item from the queue.

Example



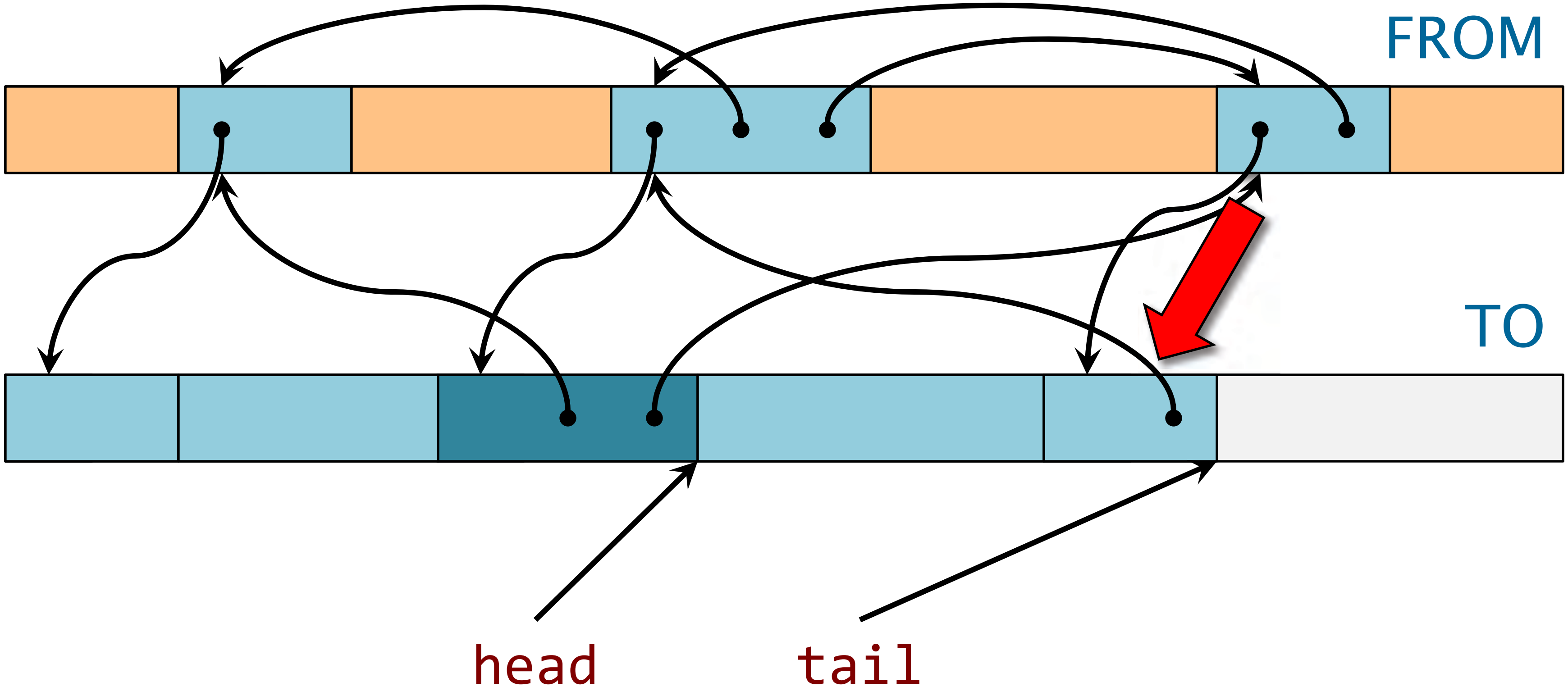
Remove an item from the queue.

Example



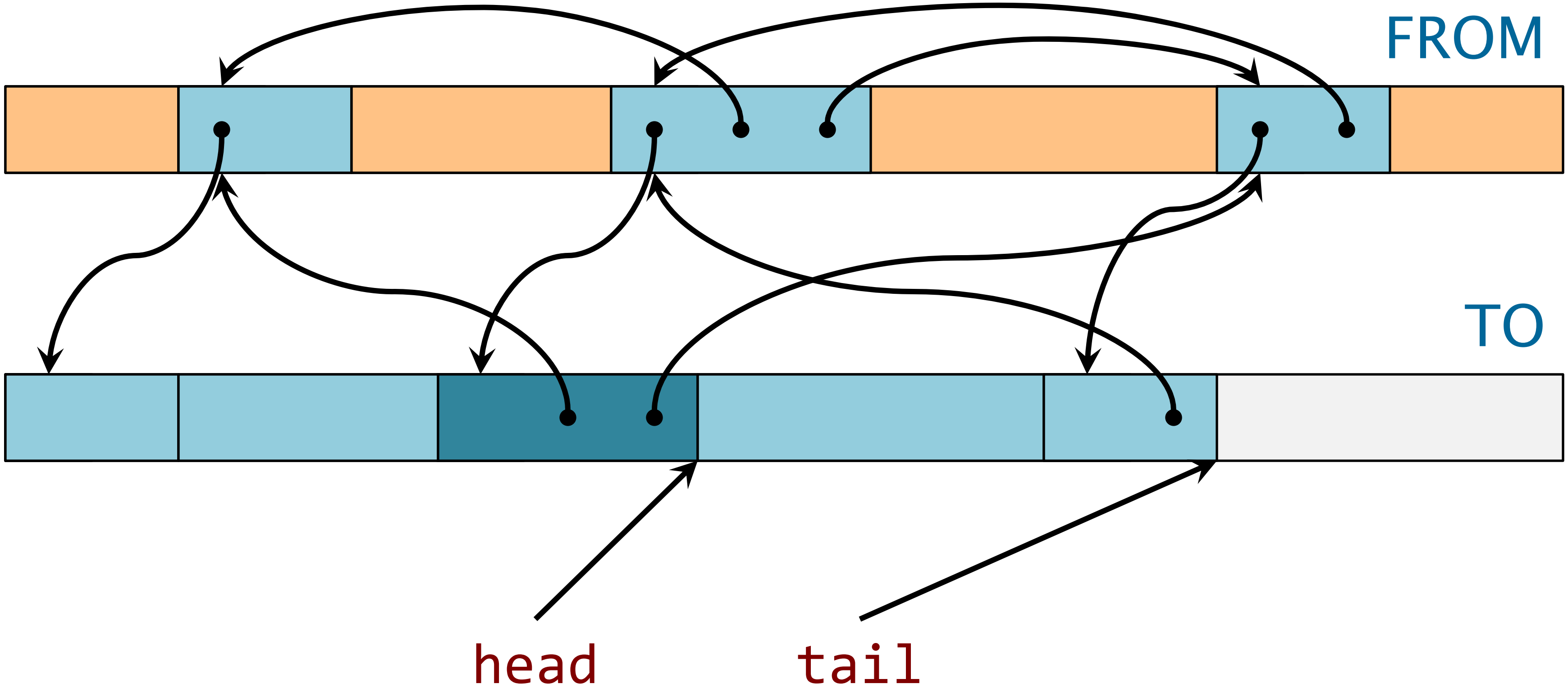
Enqueue adjacent vertices.

Example



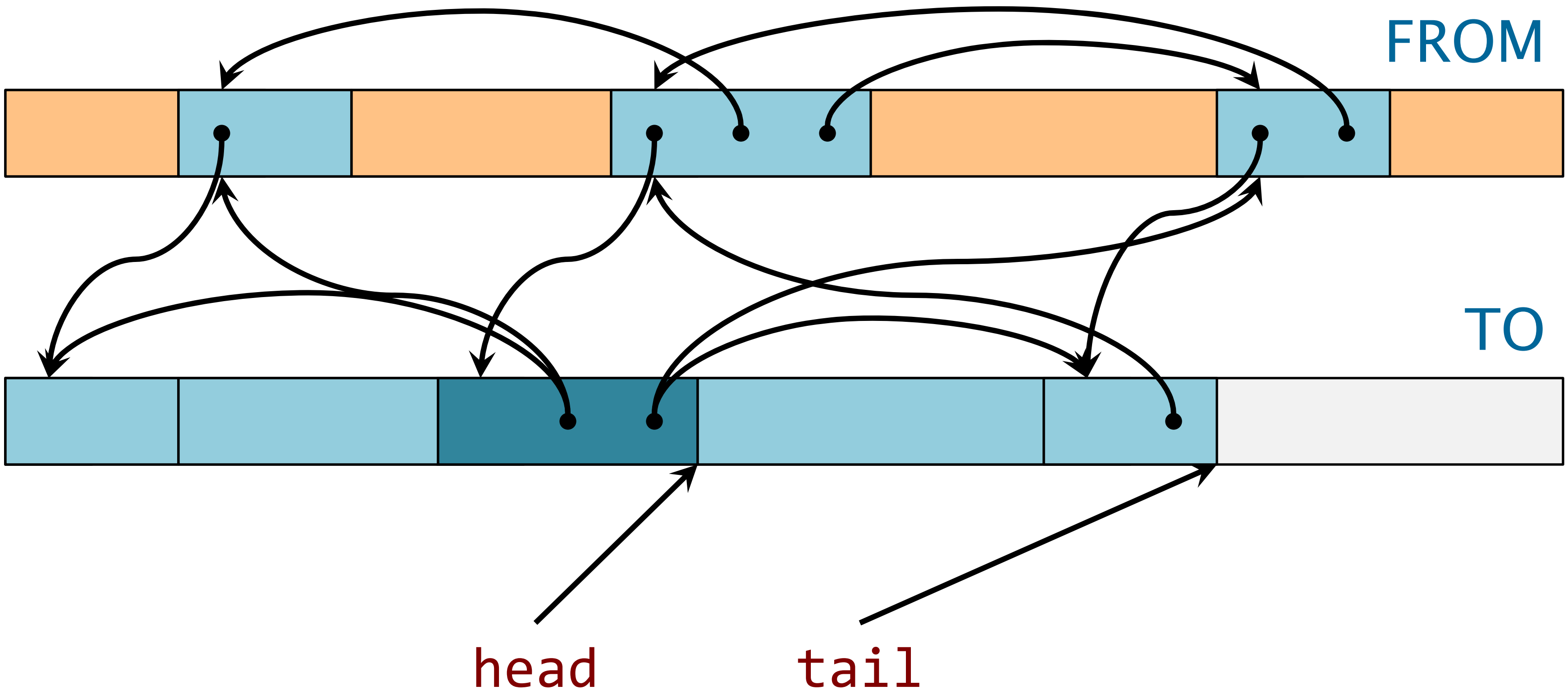
Enqueue adjacent vertices.
Place forwarding pointers in FROM vertices.

Example



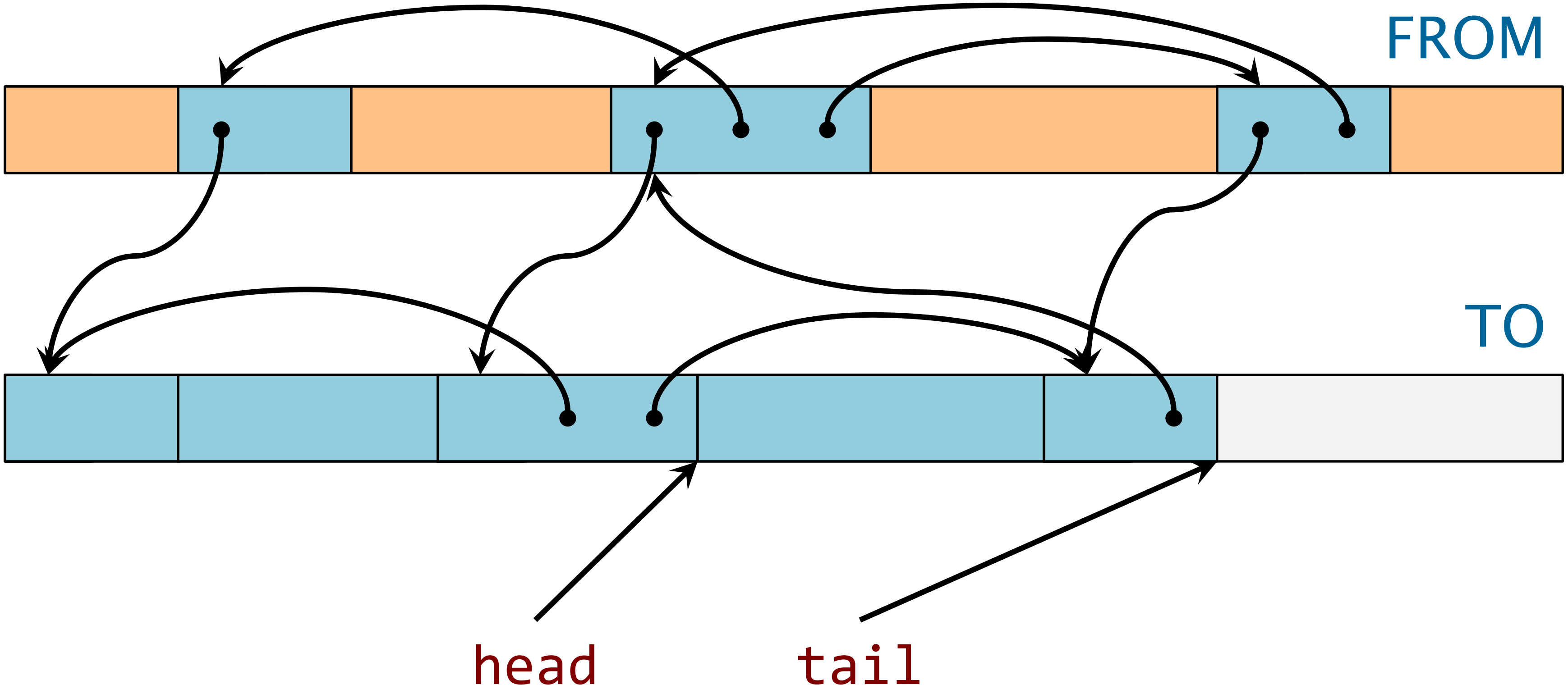
Update the pointers in the removed item to refer to its adjacent items in the TO space.

Example



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

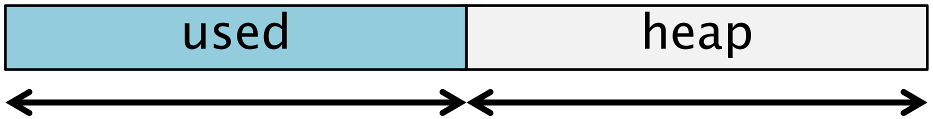
Example



Linear time to copy and update all vertices.

When Is the FROM Space "Full"?

FROM



- 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 then 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.

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.

Summary

- Stack: most basic form of storage and is very efficient when it works
- Heap is the more general form of storage
- Fixed-size allocation using free lists
- Variable-sized allocation using binned free lists
- Garbage collection – reference counting, mark-and-sweep, stop-and-copy
- Internal and external fragmentation
- You will look at storage allocation in Homework 6 and Project 3