Queue Data Structures

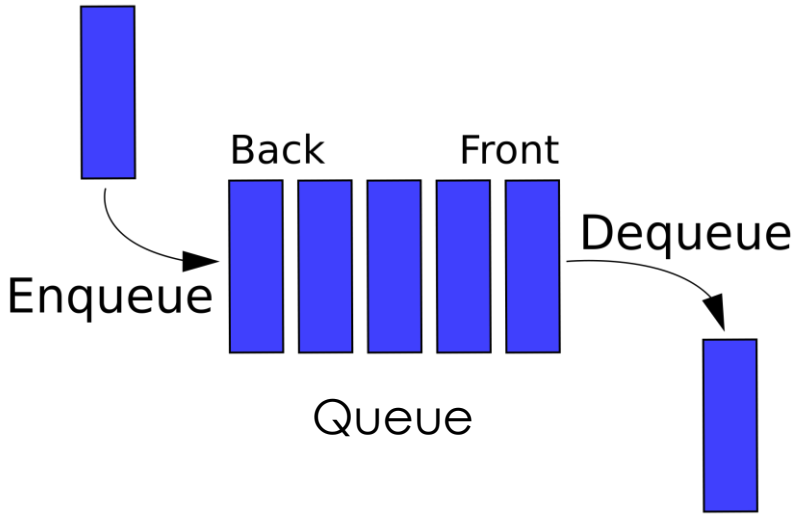
261217 Data Structures for Computer Engineers

Patiwet Wuttisarnwattana, Ph.D.

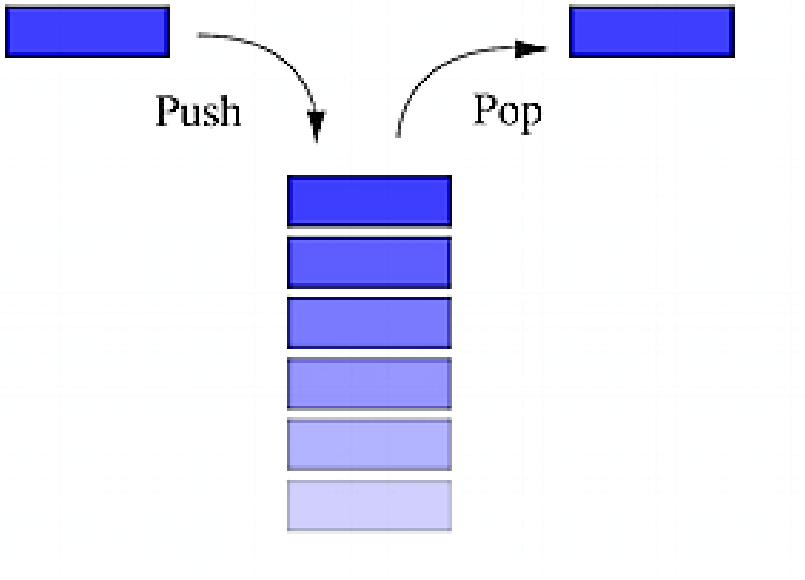
patiwet@eng.cmu.ac.th

Computer Engineering, Chiang Mai University

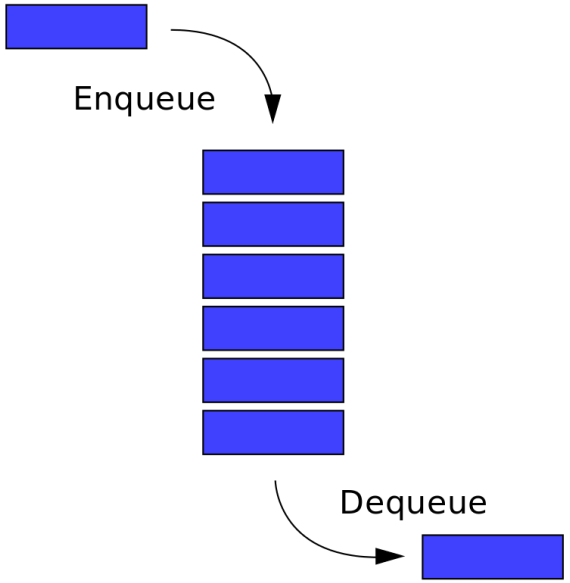
Queue Data Structure



Queue vs Stack



One-ended list



Two-ended list

Which is which?

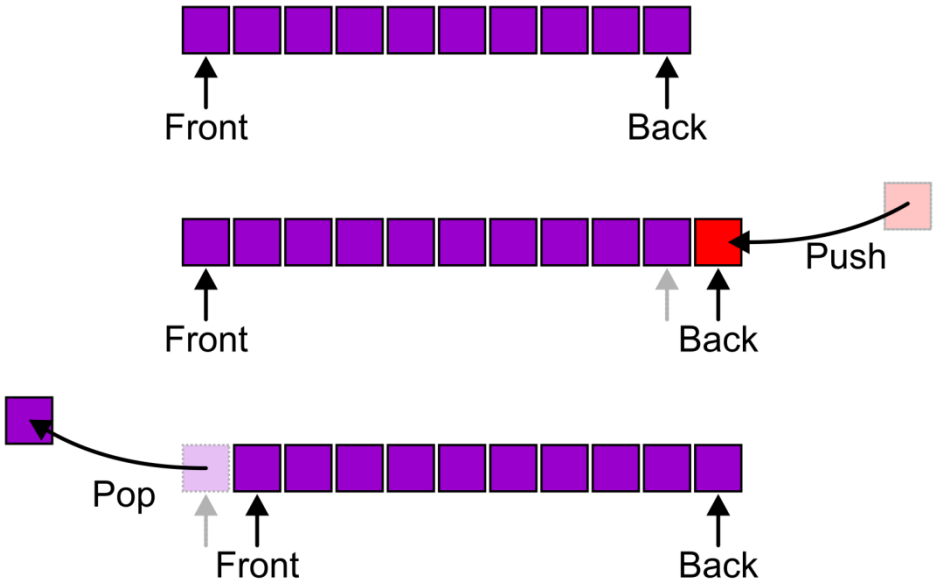
Queue Data Structure

- Linear ordering like Stack
- Queue is two-ended list but Stack is one-ended list
- Insert at one end of the List, remove at the other end
- Queues are **"FIFO" – First In, First Out**
- Primary operations are Enqueue and Dequeue
- A queues ensures "fairness"
  - customers waiting on a customer hotline
  - processes waiting to run on the CPU

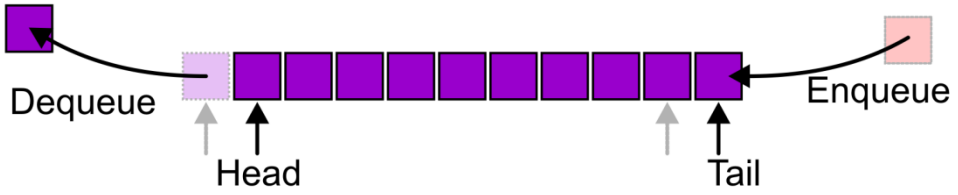
Queue API (Operations)

- *Enqueue*(Key) or *Push*(Key): Add Key at the **end** of the queue (also called "back", "rear", or "tail")
- Key *Dequeue*() or Key *Pop*(): Remove and return a Key from the **front** of the queue (earliest-added key)
- Boolean *IsEmpty*(): Is there any element?

FIFO



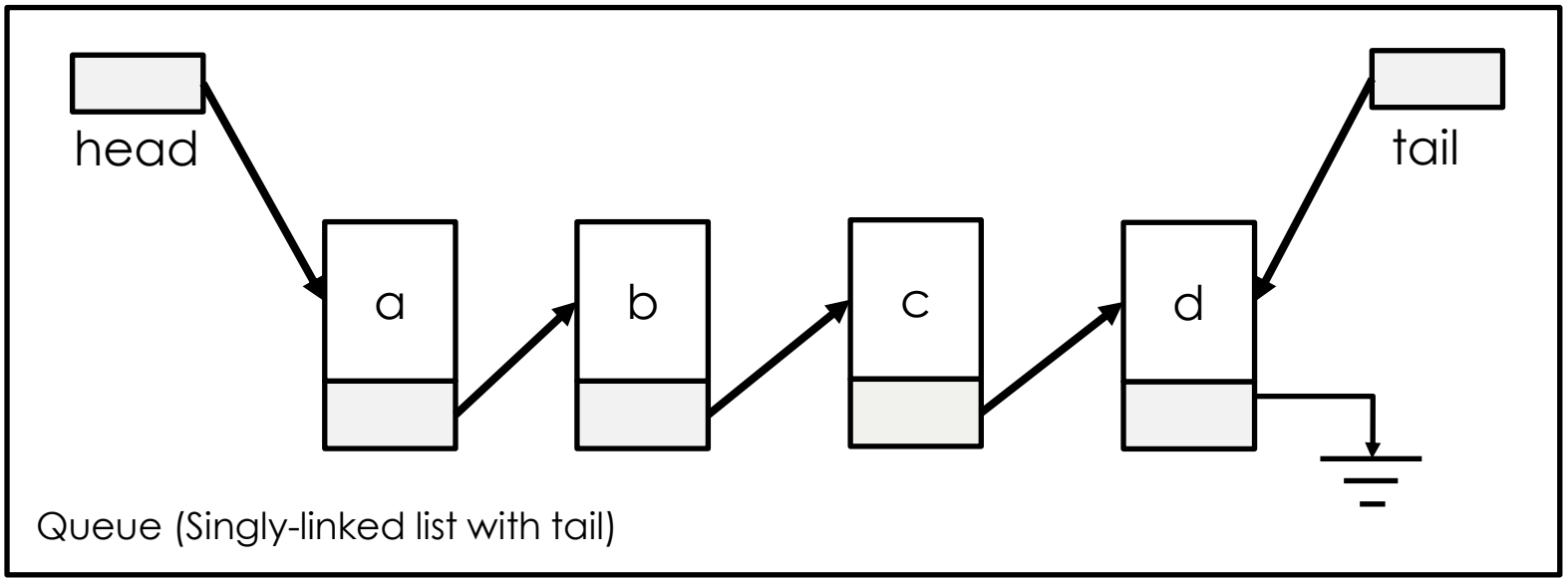
Alternative terms



A sample of CPE Applications of Queue

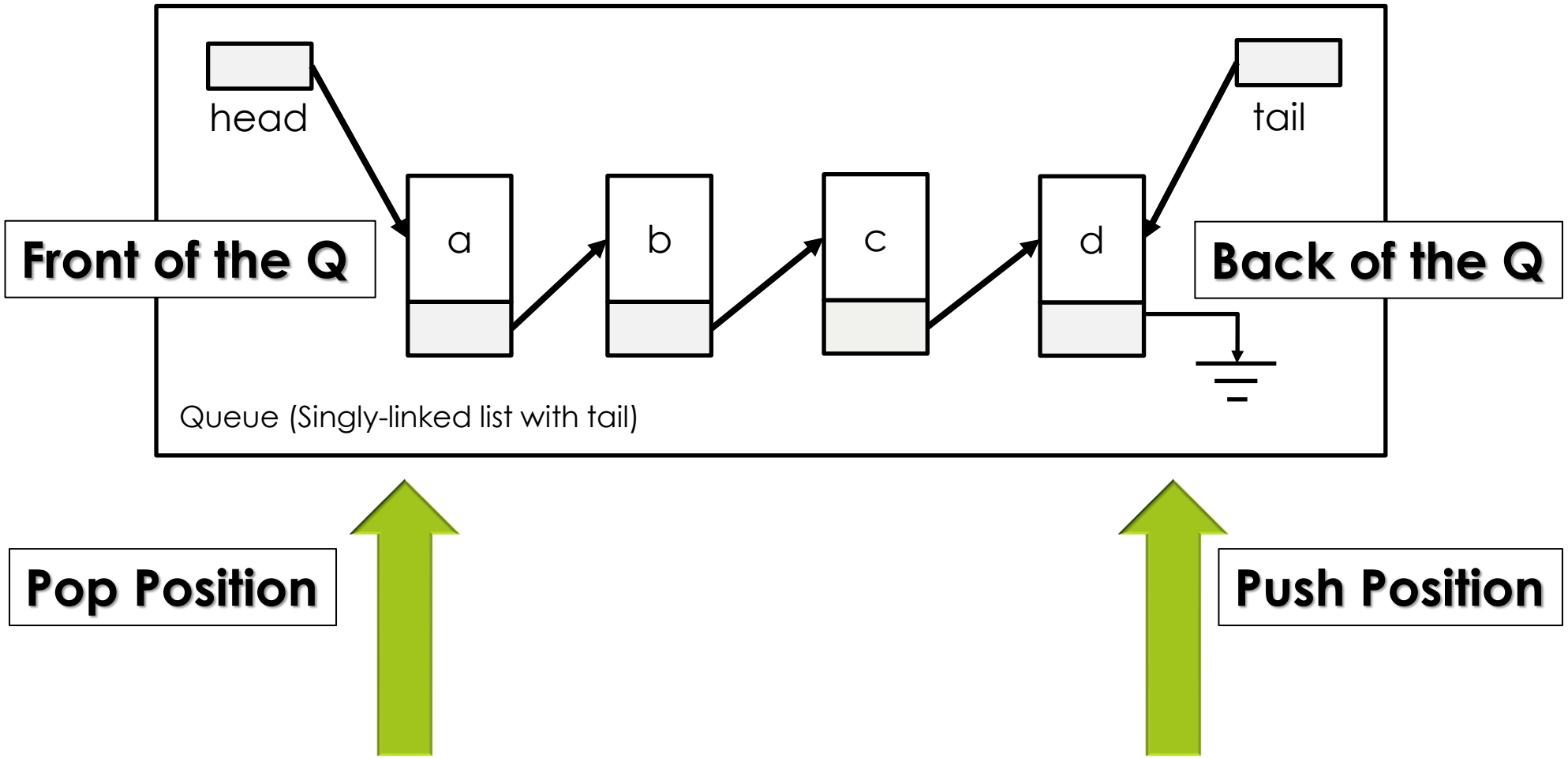
- File servers: Multiples users need to access their files on a shared file server machine are given access on a FIFO basis
- Printer Queue: Printing jobs submitted to a printer are printed in order of arrival
- Phone calls made to customer service hotlines are usually placed in a queue
- Note that Queue works best with jobs with similar priority. If the jobs are with different priority, you should use Priority Queue Data Structure instead.

Queue Implementation with Linked List (Singly-linked list with tail)

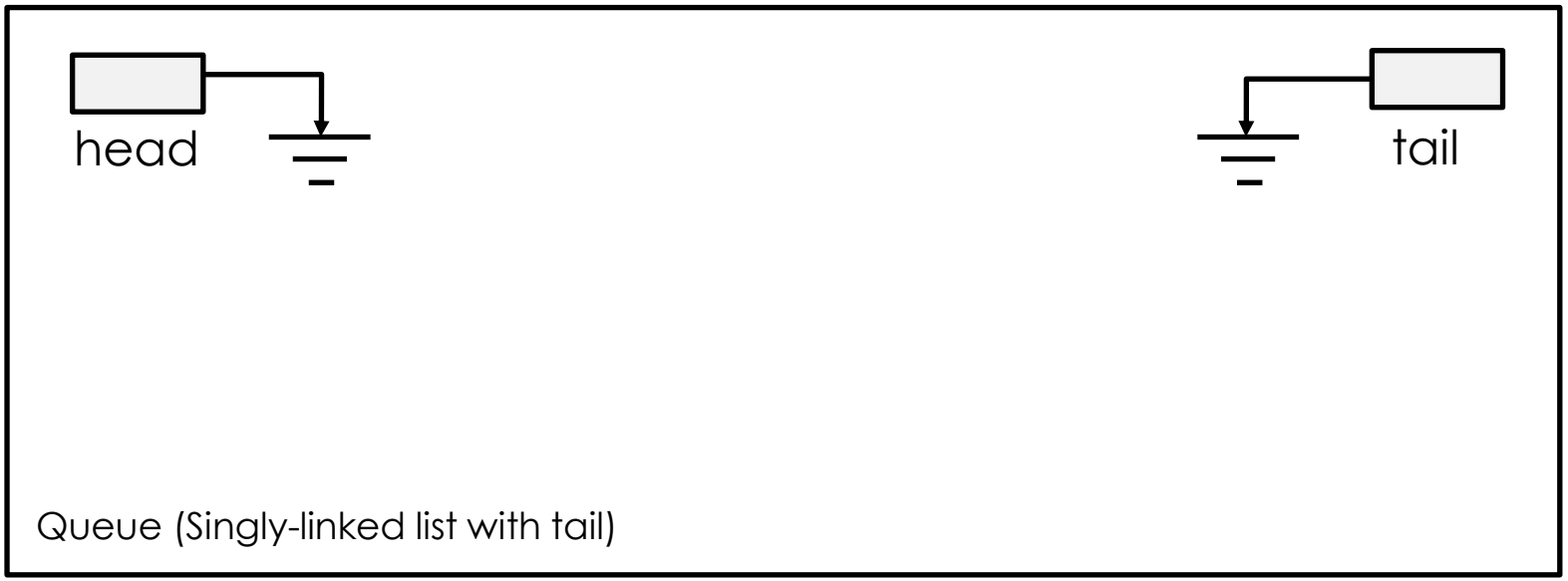


Which side should be the Push position for Enqueue?
Which side should be the Pop position for Dequeue?

Queue Implementation with Linked List (Singly-linked list with tail)

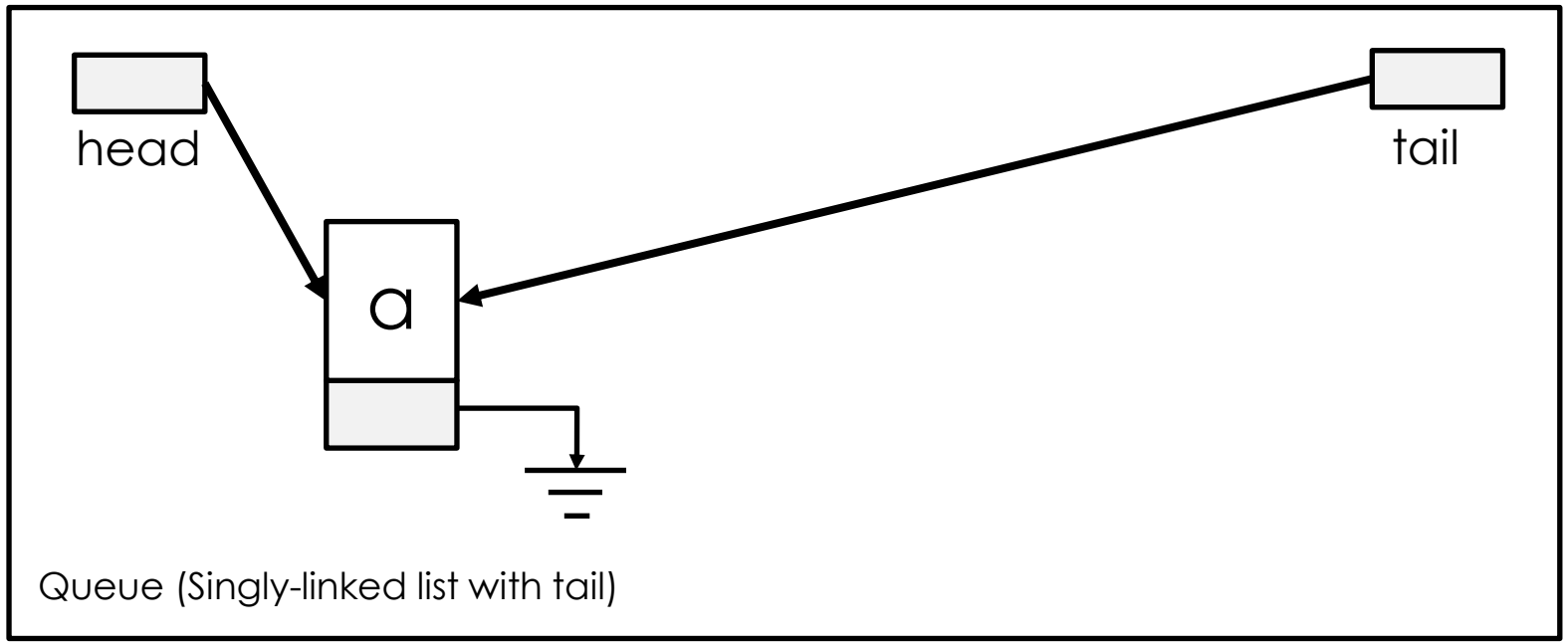


Queue Implementation with Linked List



Enqueue(a)

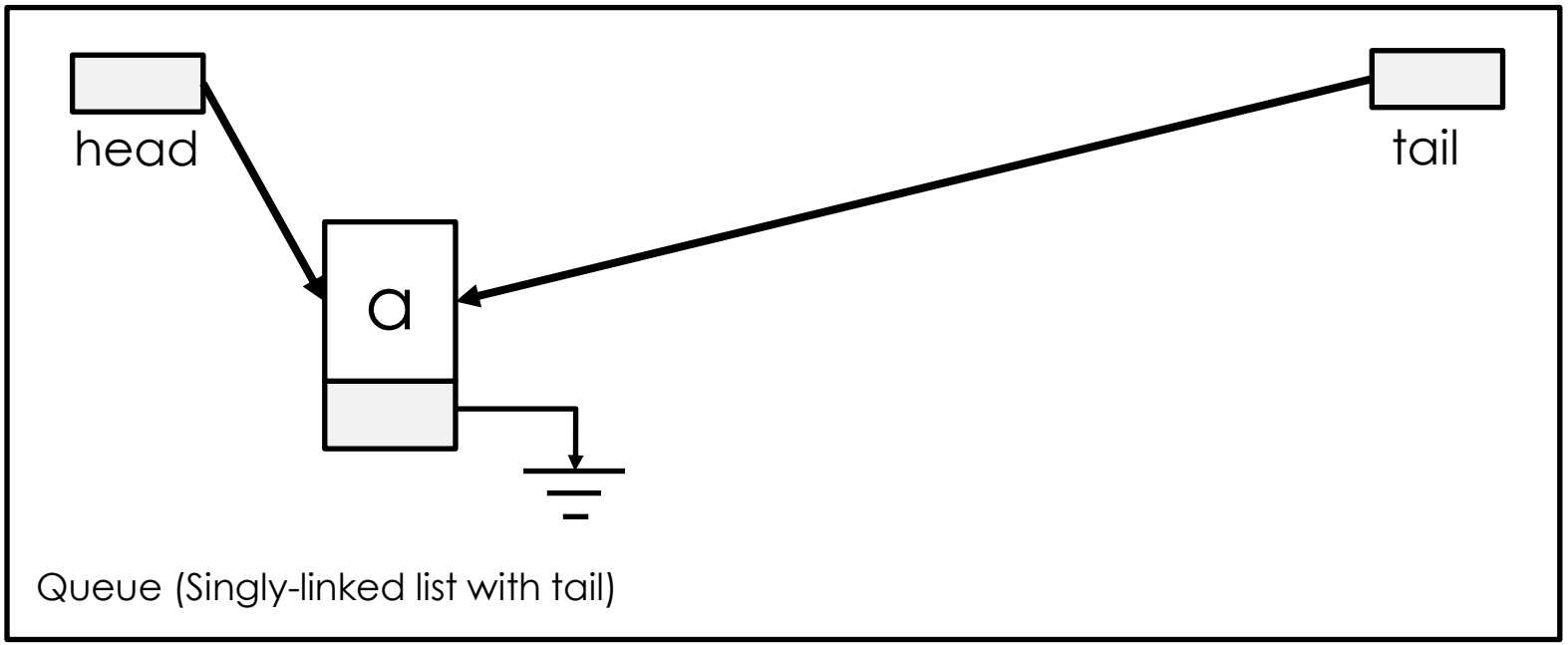
Queue Implementation with Linked List



Queue (Singly-linked list with tail)

Enqueue(a)

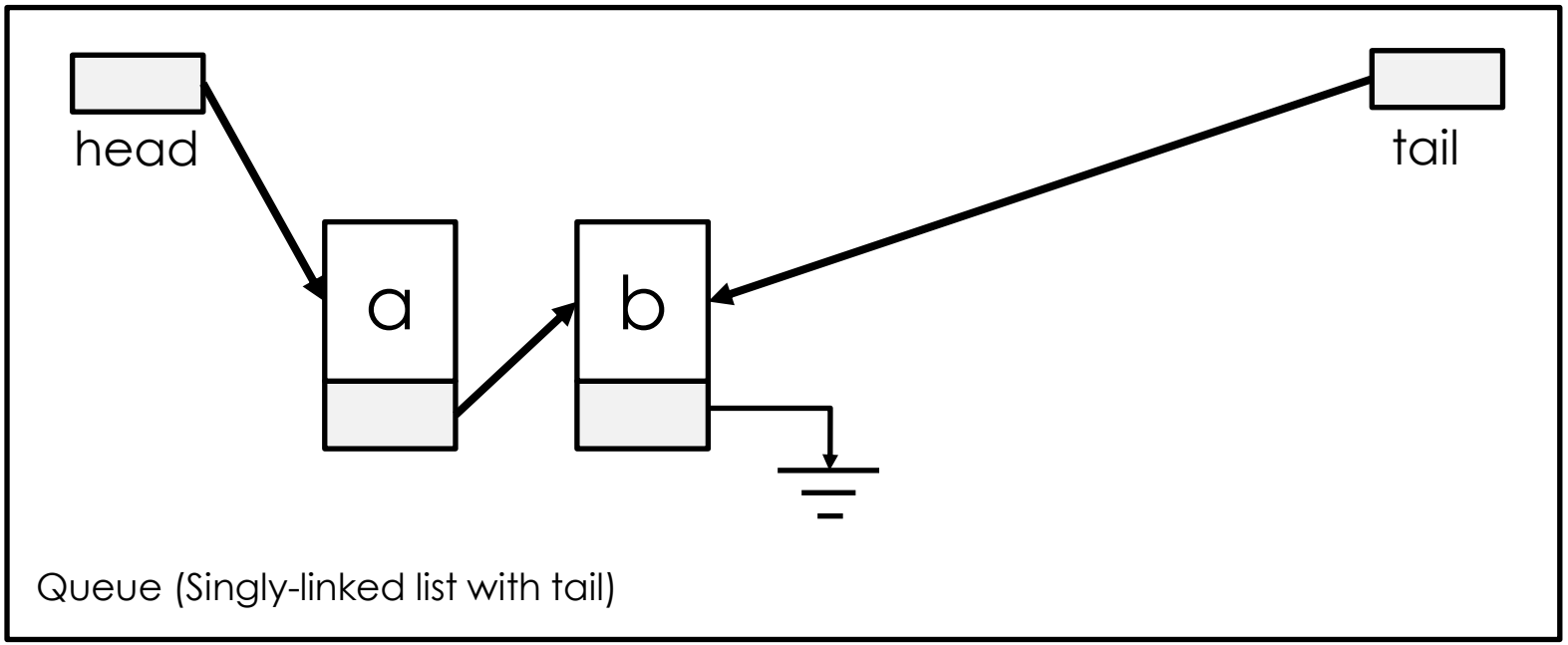
Queue Implementation with Linked List



Queue (Singly-linked list with tail)

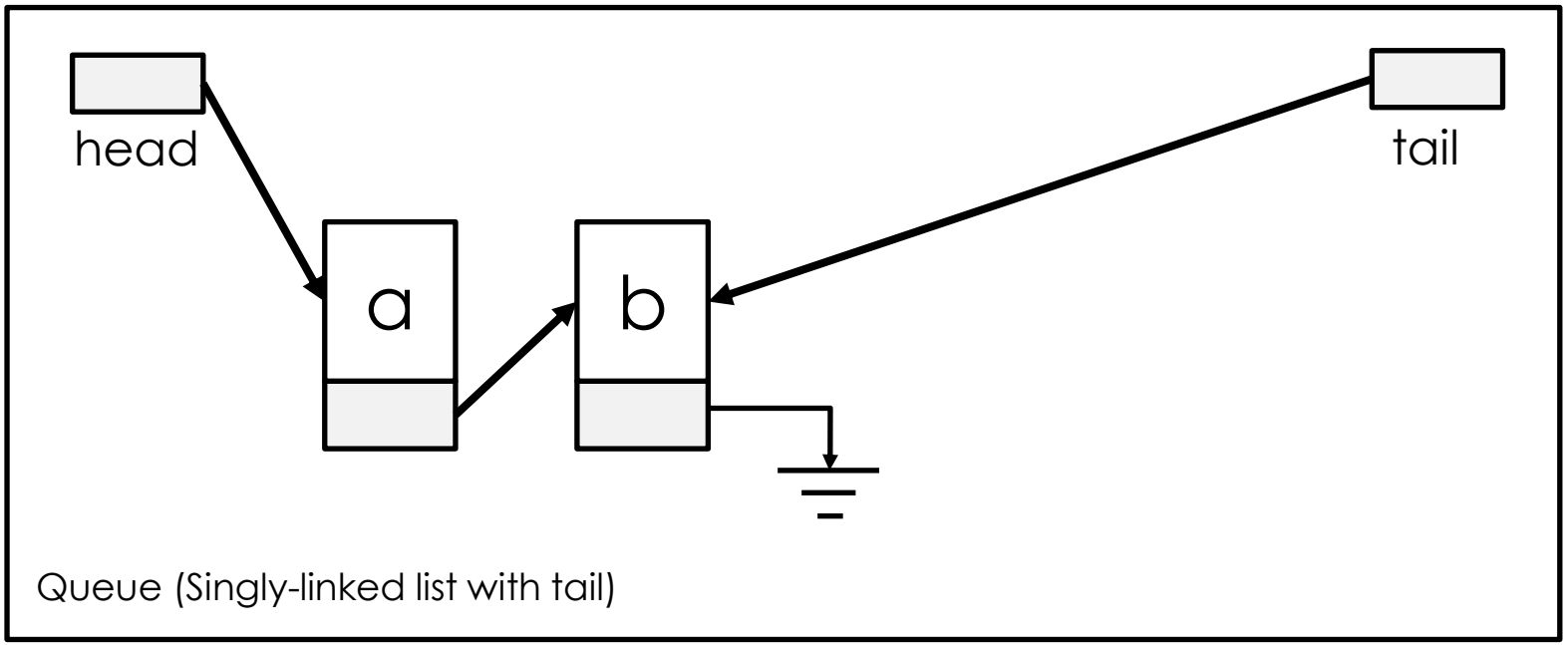
Enqueue(b)

Queue Implementation with Linked List



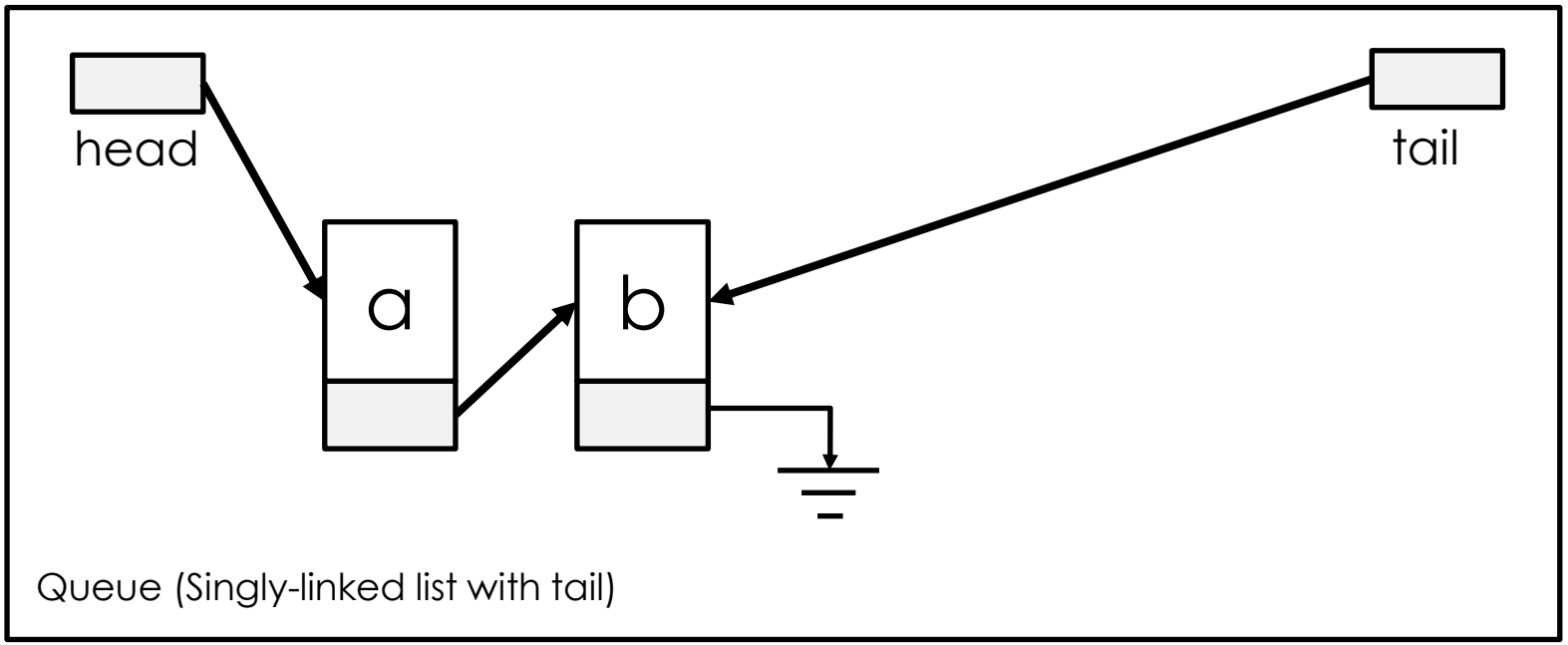
Enqueue(b)

Queue Implementation with Linked List



isEmpty() → false

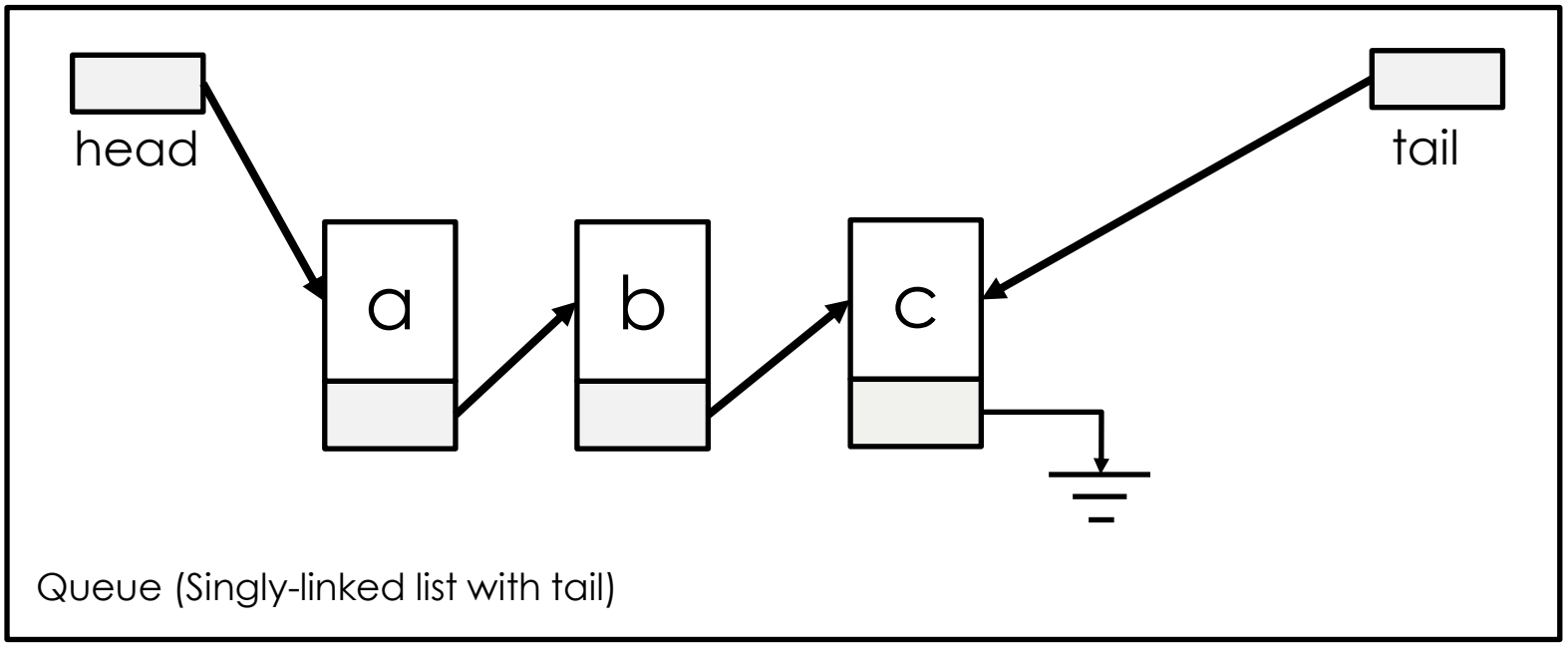
Queue Implementation with Linked List



Queue (Singly-linked list with tail)

Enqueue(c)

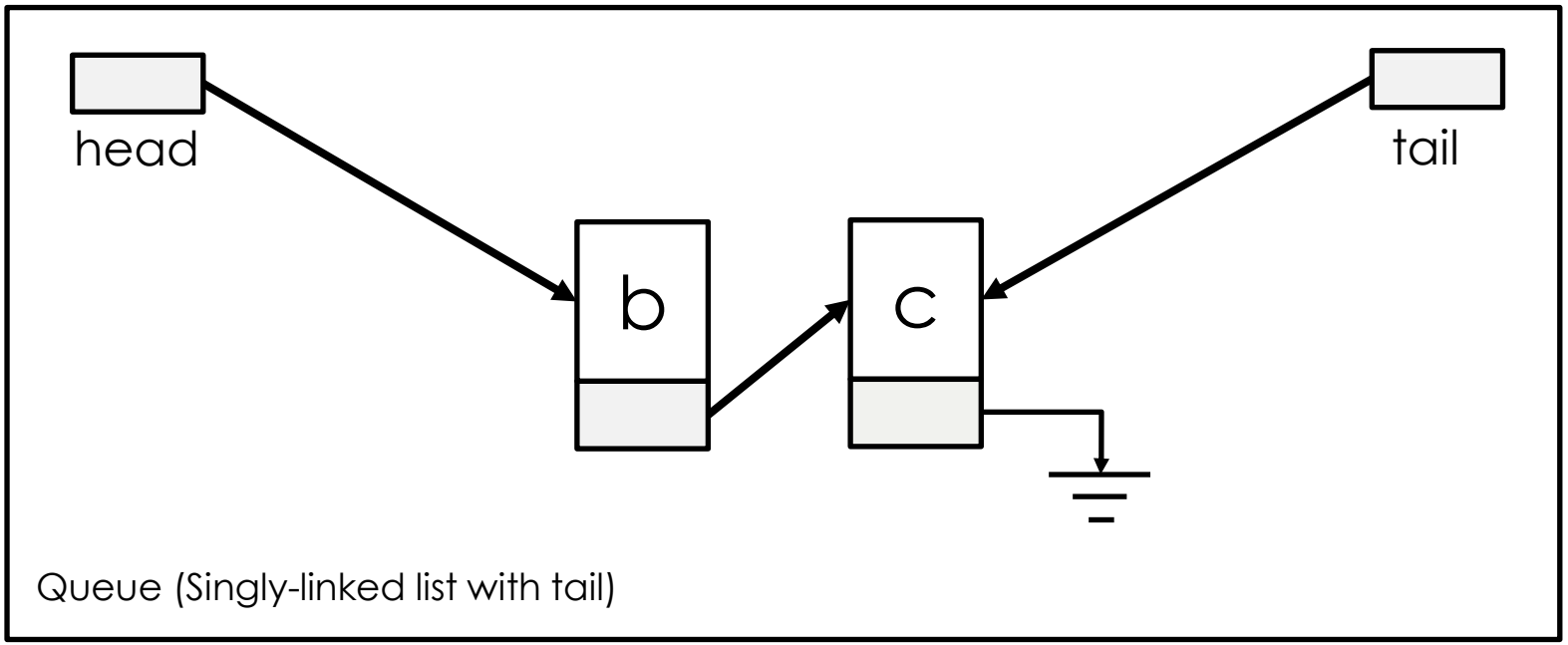
Queue Implementation with Linked List



Queue (Singly-linked list with tail)

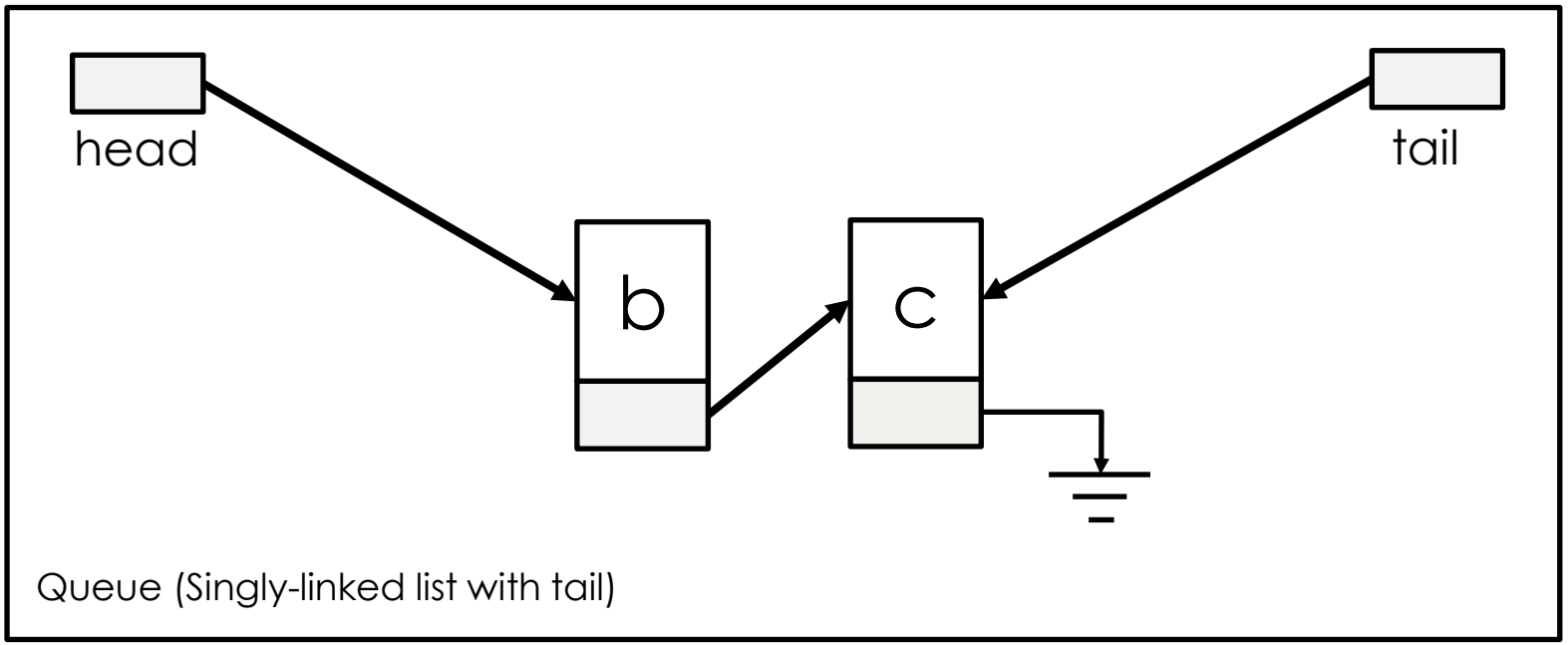
Enqueue(c)

Queue Implementation with Linked List



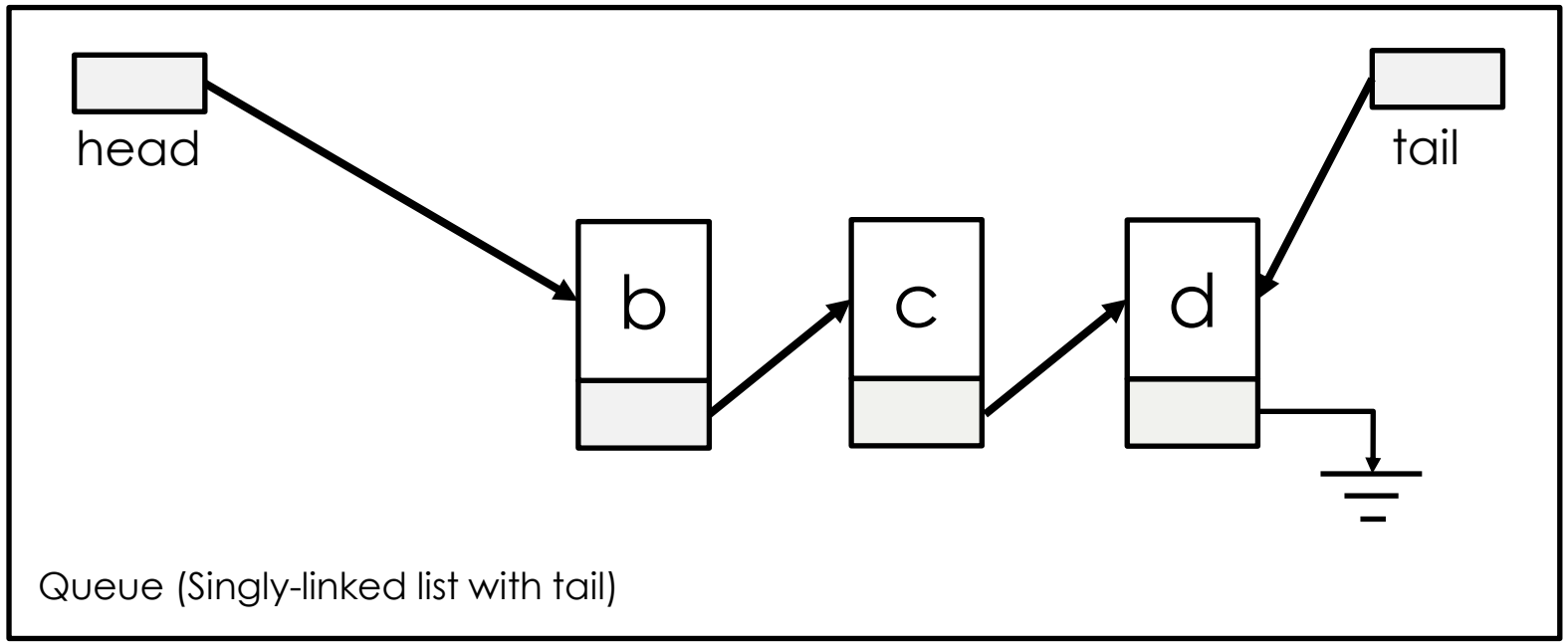
Dequeue() → a

Queue Implementation with Linked List



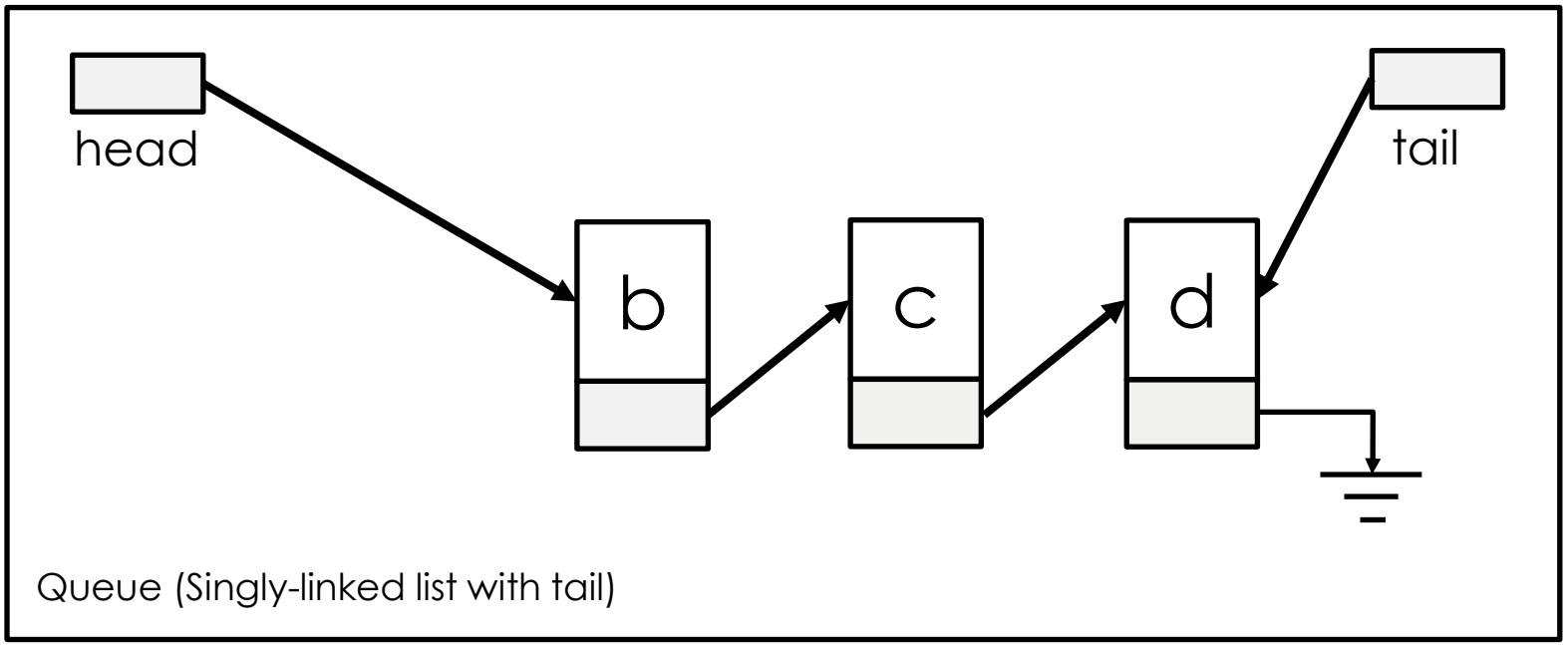
Enqueue(d)

Queue Implementation with Linked List



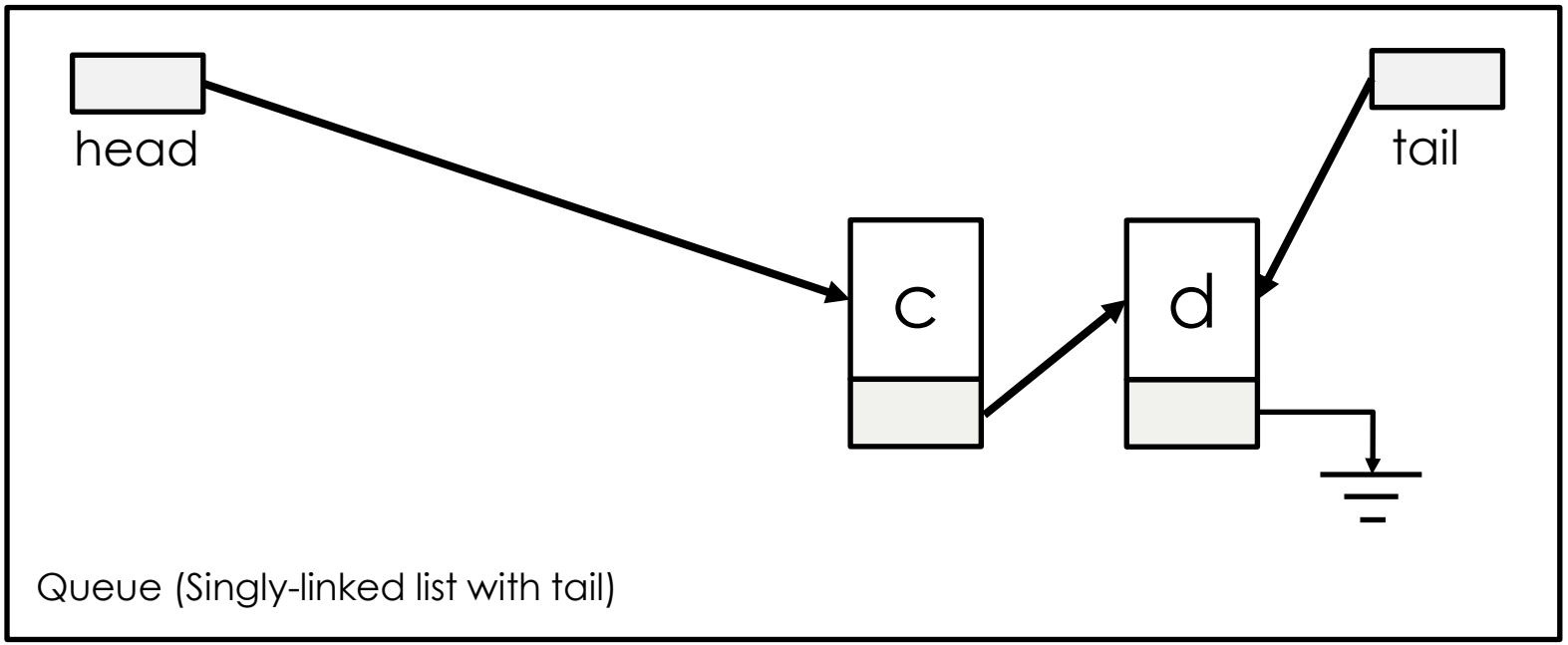
Enqueue(d)

Queue Implementation with Linked List



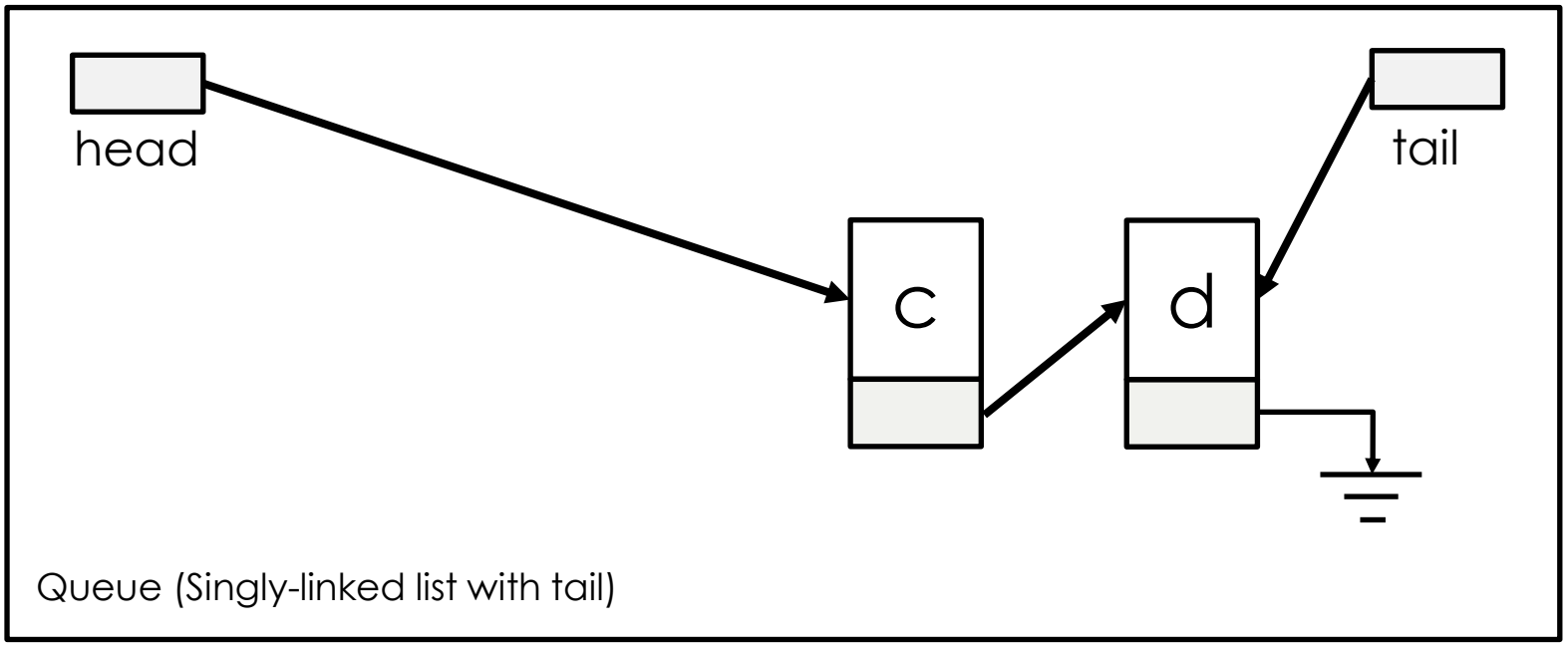
Dequeue()

Queue Implementation with Linked List



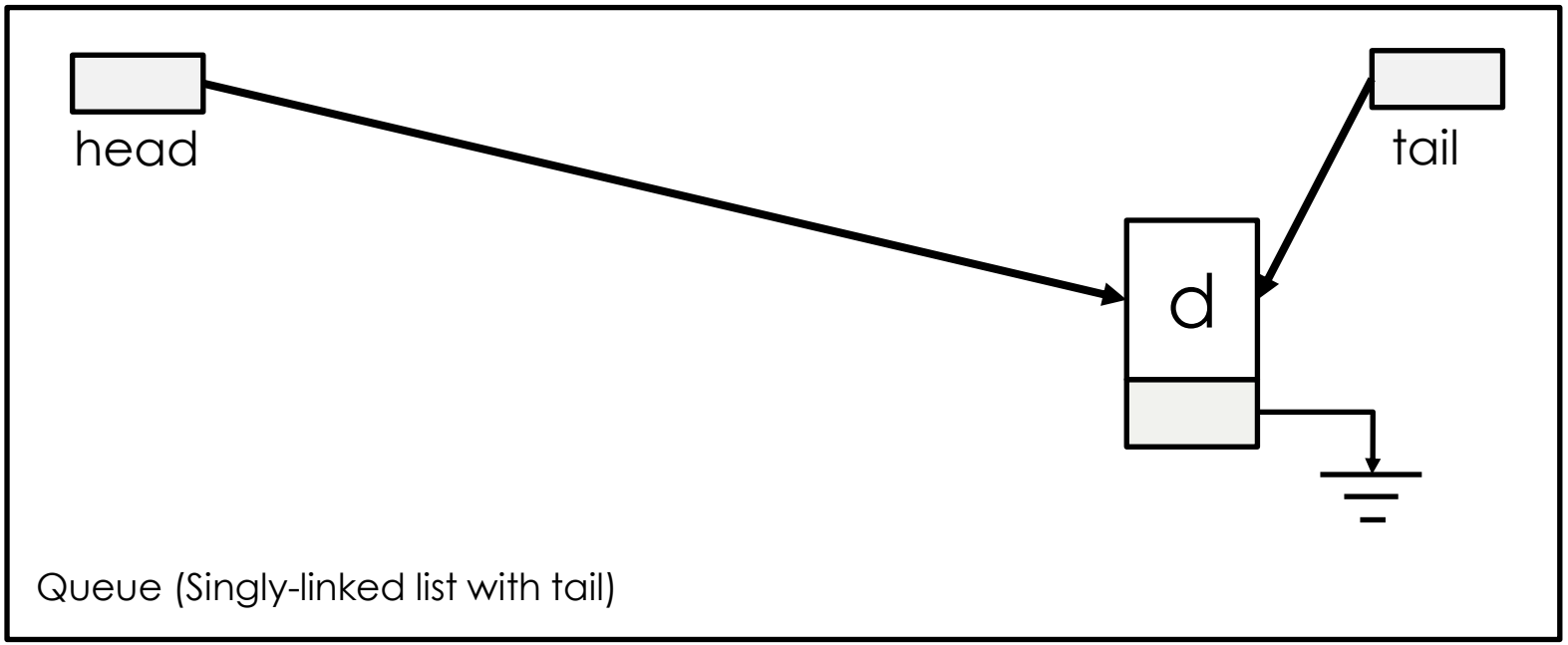
Dequeue() → b

Queue Implementation with Linked List



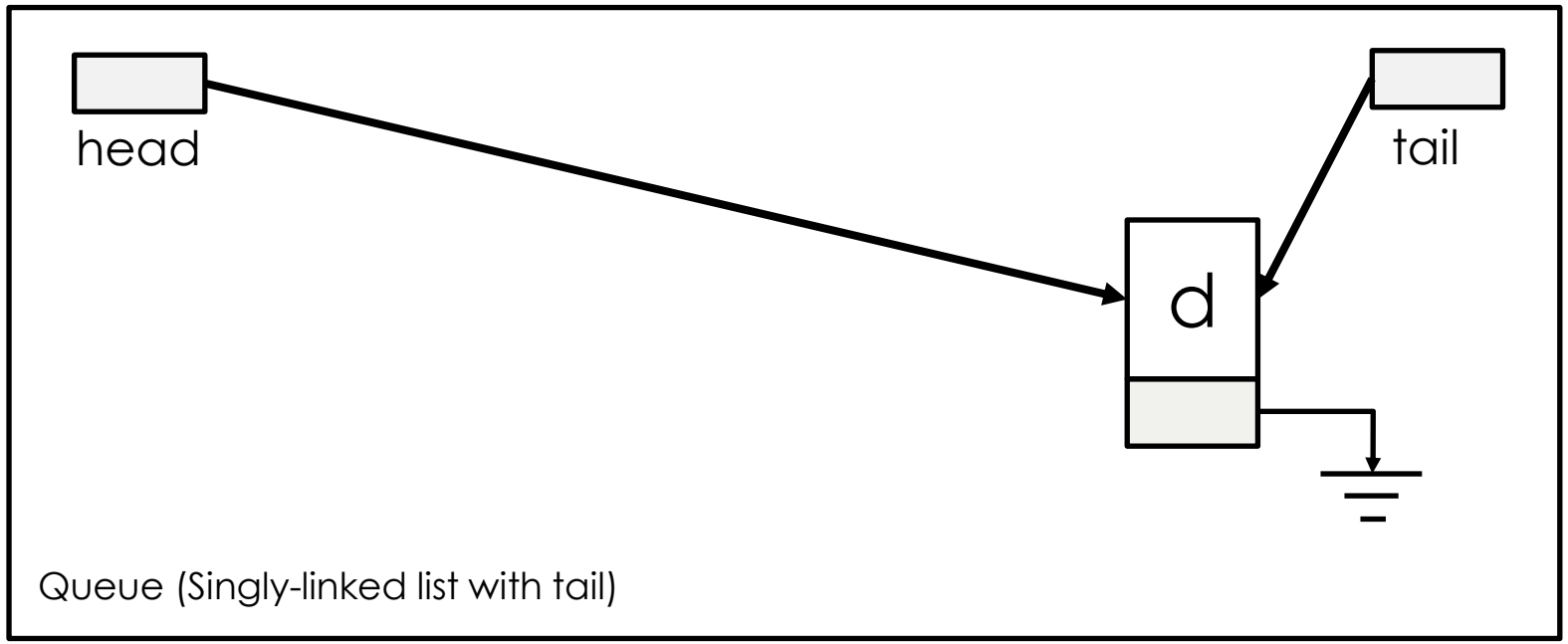
Dequeue()

Queue Implementation with Linked List



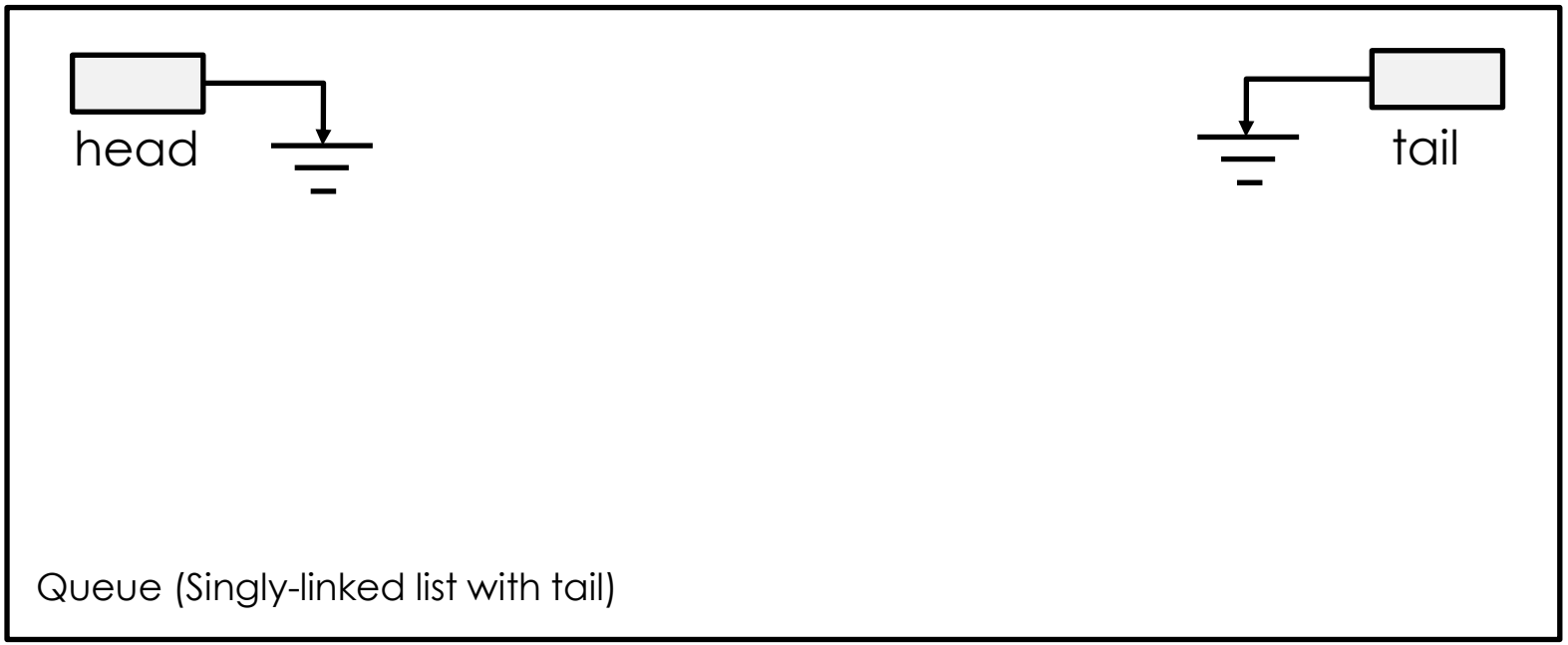
Dequeue() → c

Queue Implementation with Linked List



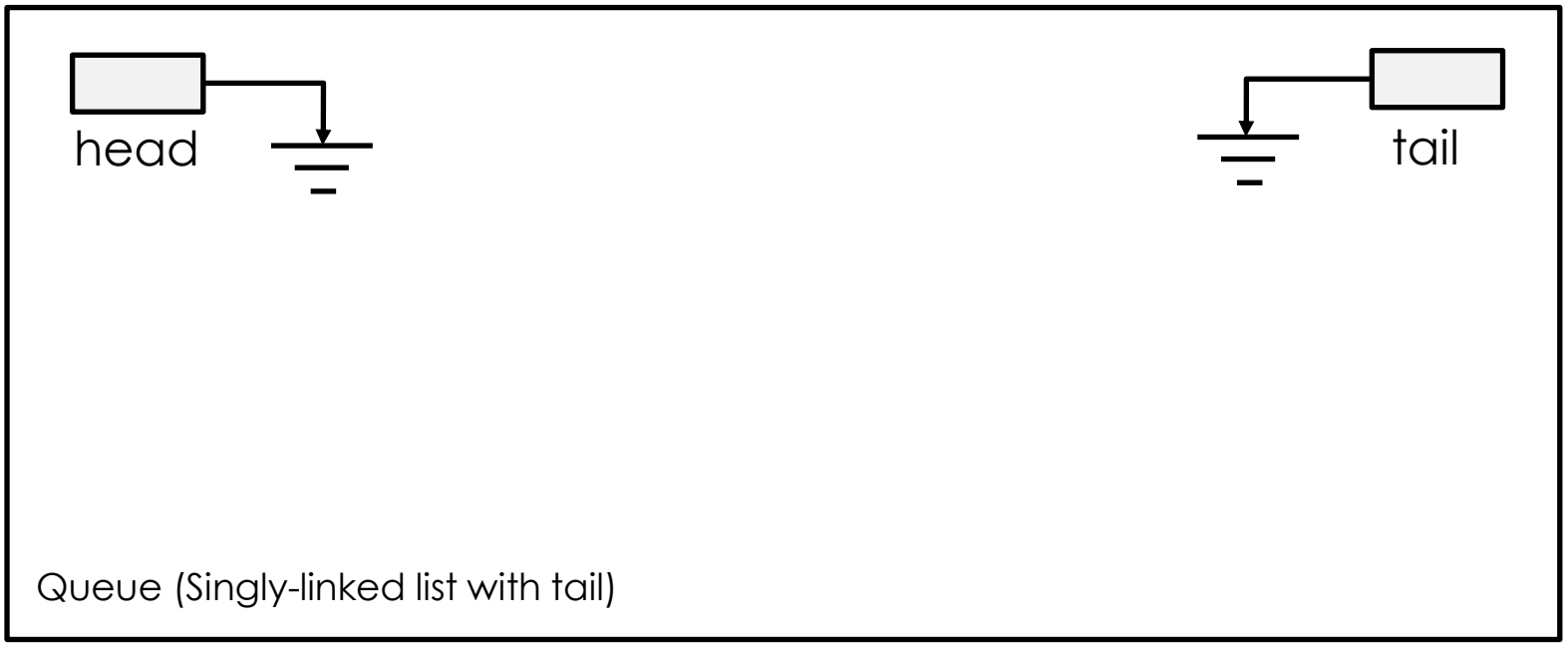
Dequeue()

Queue Implementation with Linked List



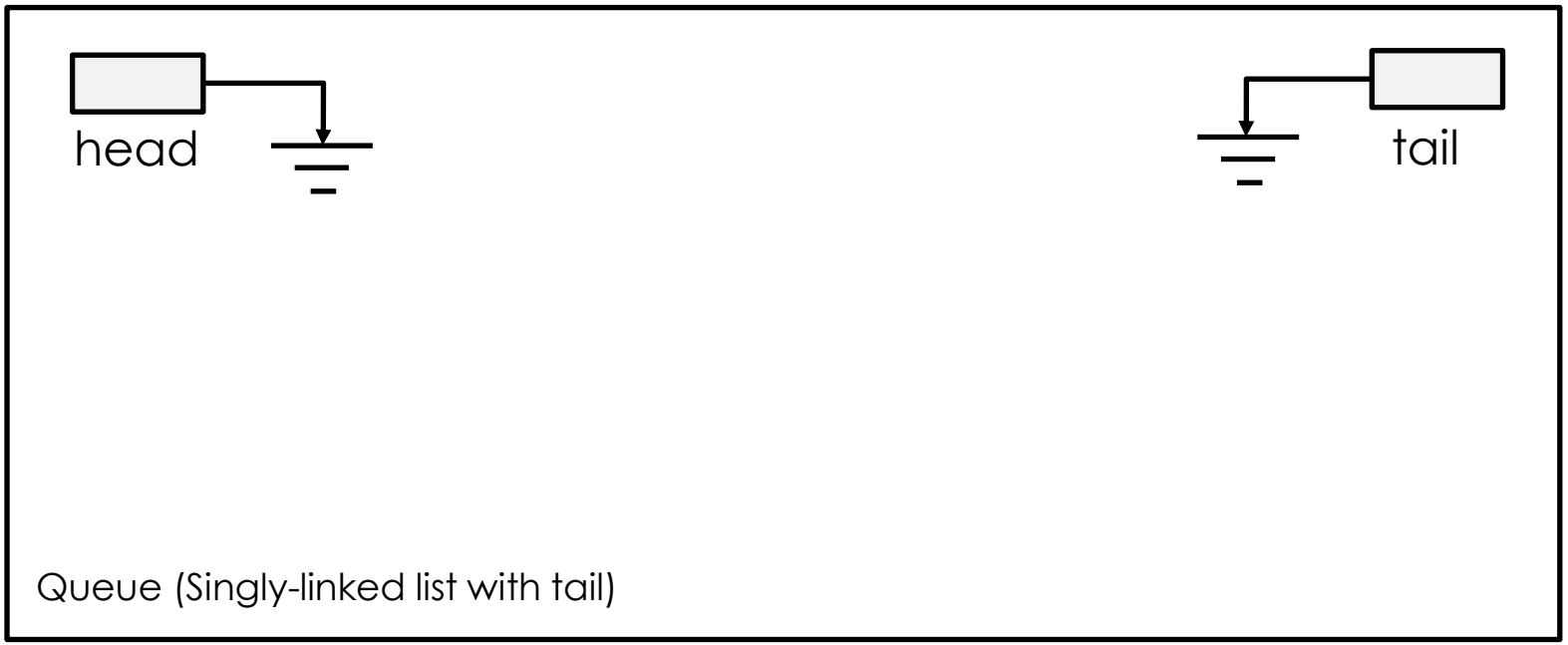
Dequeue() → d

Queue Implementation with Linked List



isEmpty()

Queue Implementation with Linked List



isEmpty() → true

Queue Implementation with Linked List

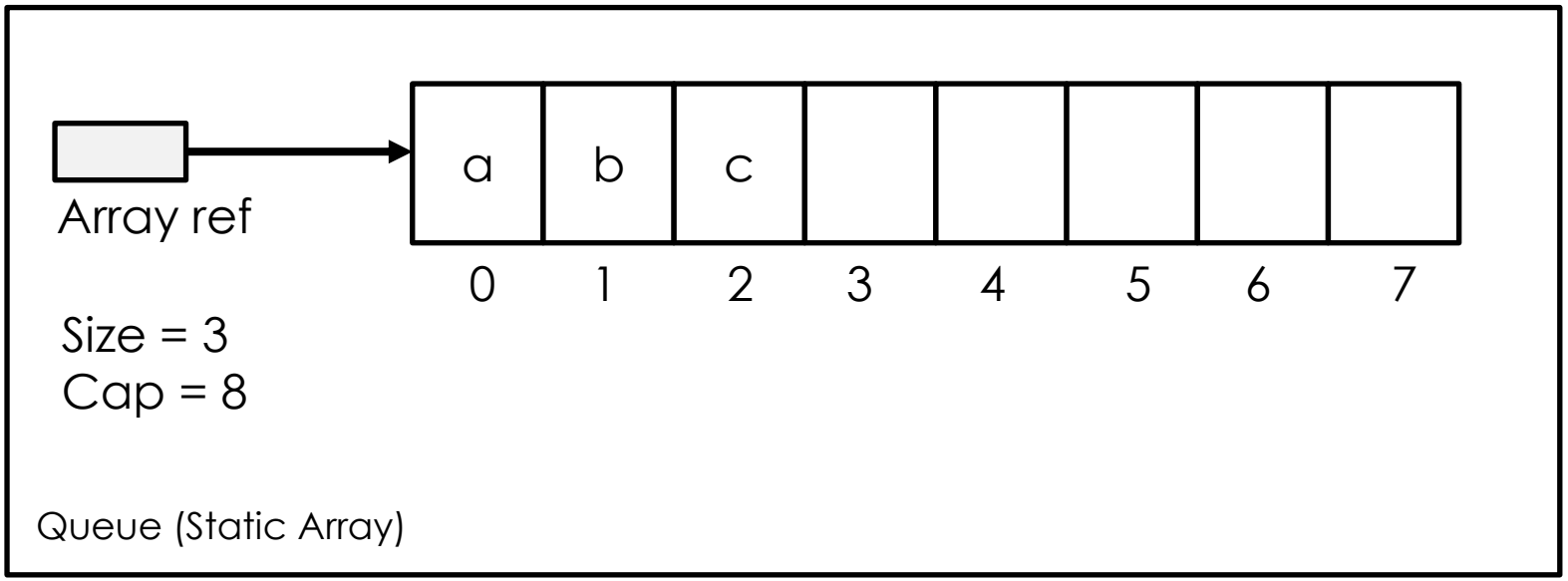
- What is the equivalent operation(s) in the List?

Queue Operations	List Operations	Complexity
Enqueue(Key)	PushBack(Key)	O(1)
Key Dequeue()	Key TopFront(); PopFront();	O(1)
isEmpty()	isEmpty()	O(1)

Thought Questions?

- What if we change the front of the Queue to be the end of the list (still singly-linked list with tail)?
- What would be the complexity for each operation?
  - Enqueue
  - Dequeue
  - isEmpty()
- How about Singly linked list without tail? (2 cases)
- How about Doubly linked list with/without tail? (4 cases)

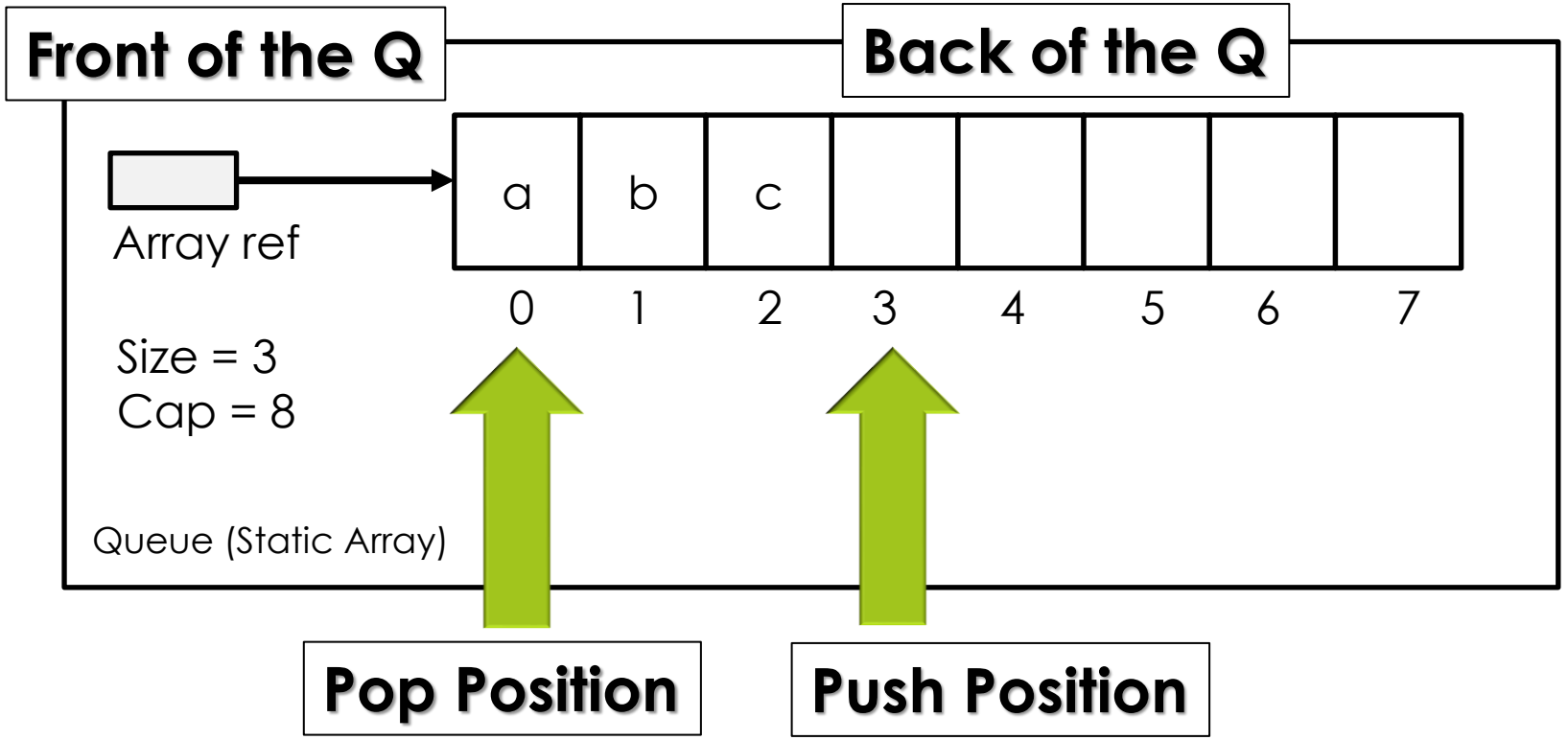
Queue Implementation with Array



Which side should be the Push position for Enqueue?
Which side should be the Pop position for Dequeue?

To prevent O(n), we should not shift anything after popping

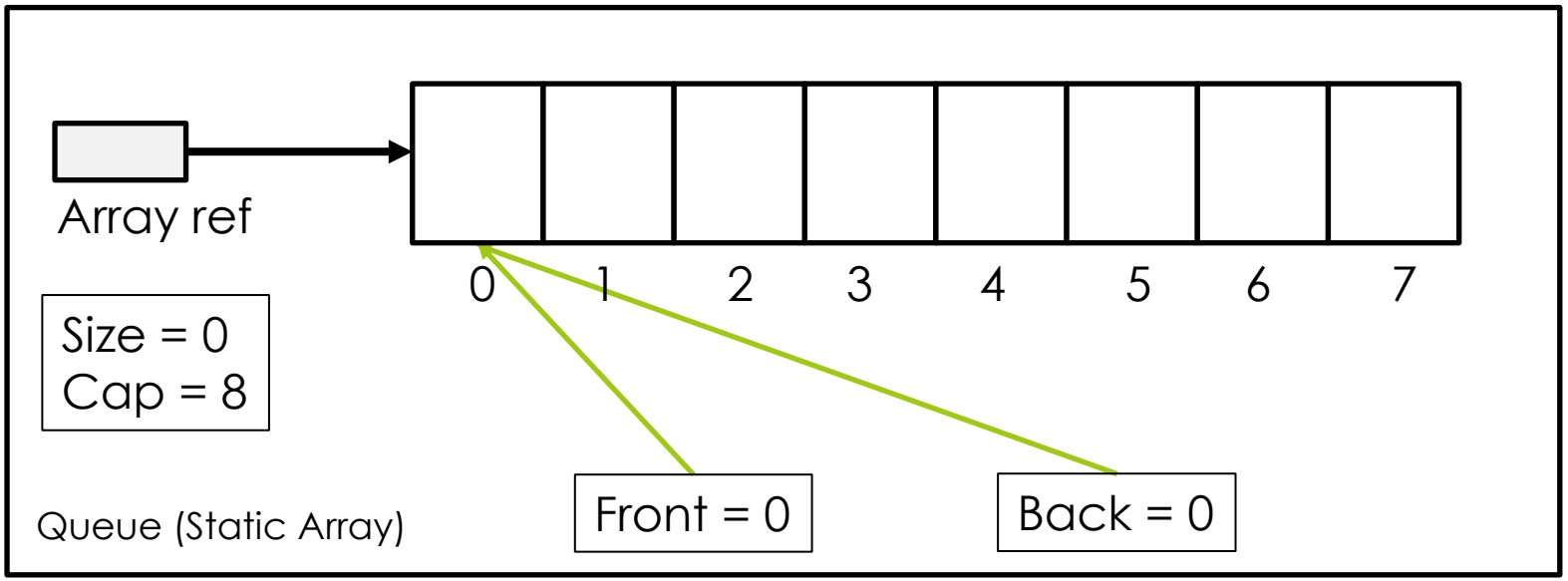
Queue Implementation with Array



To implement Queue using Array, we need two more variables (integer type) →

- Front (indicating pop position)
- Back (indicating push position)

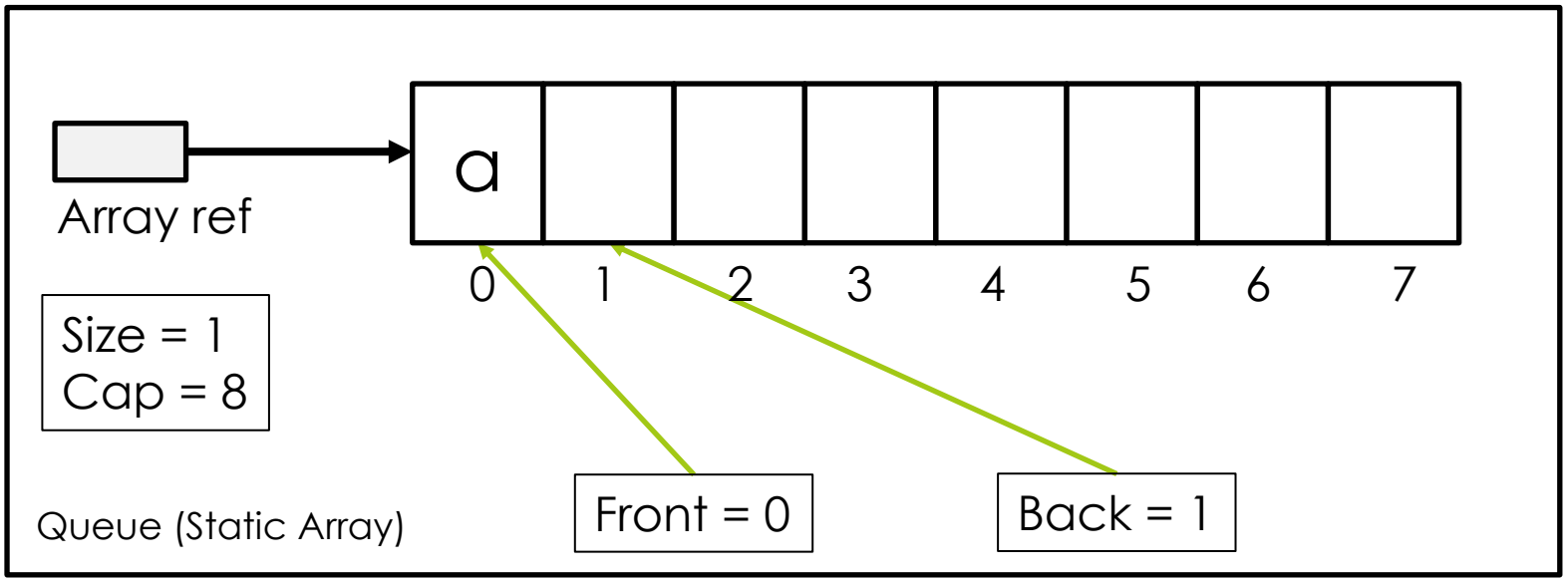
Queue Implementation with Array



Enqueue(a)

What are the corresponding array operations?

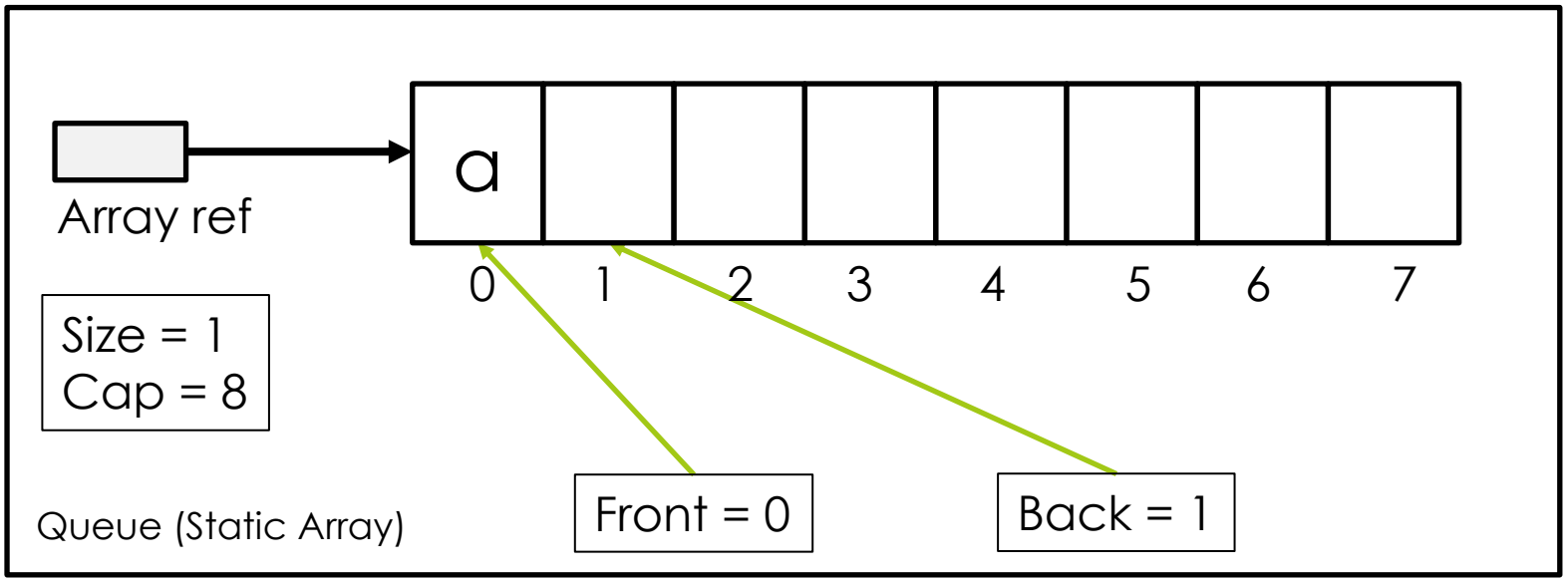
Queue Implementation with Array



Enqueue(a)

Front can be either one block after the last item or the last item itself!
In this lecture, the front variable will be one block after the last item.

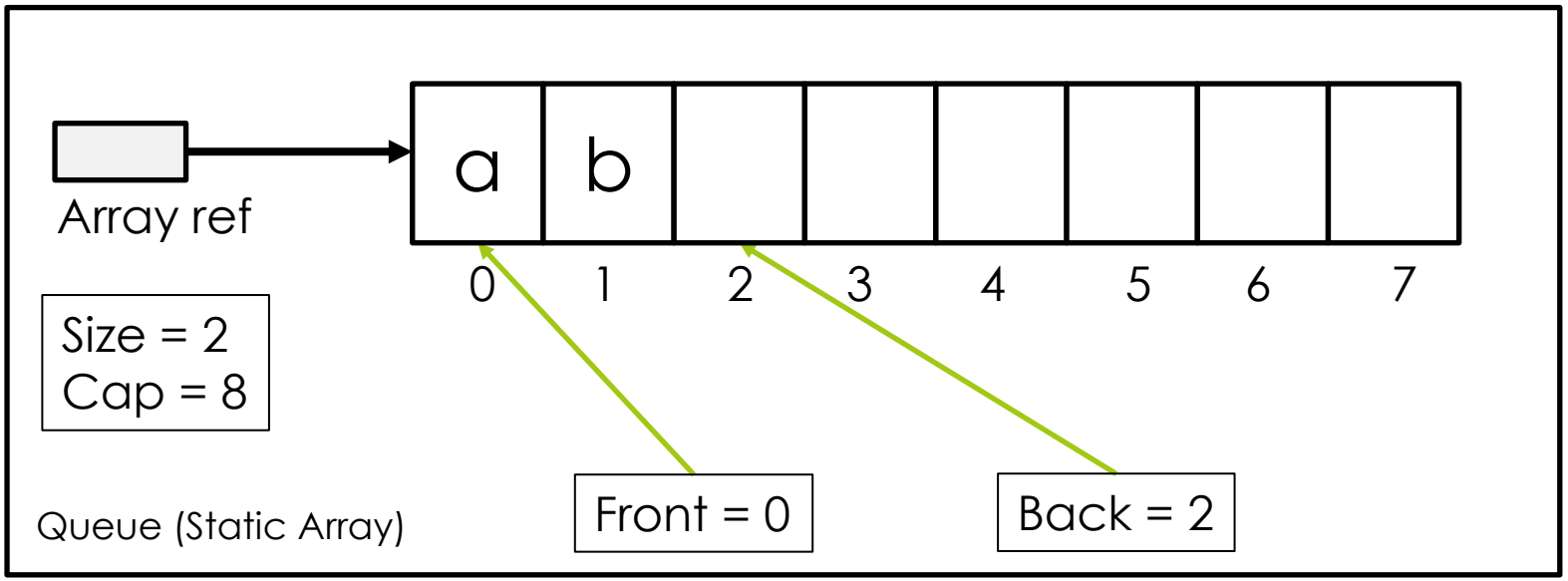
Queue Implementation with Array



Enqueue(b)

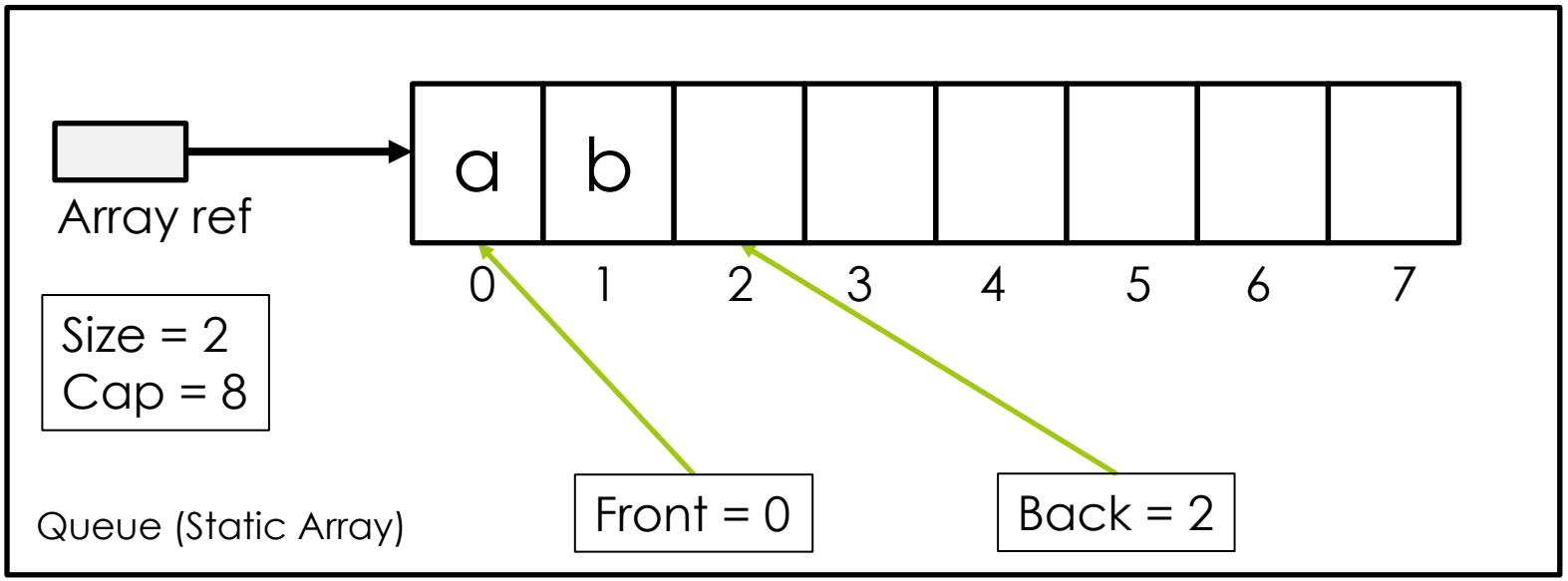
What are the corresponding Array operations?

Queue Implementation with Array



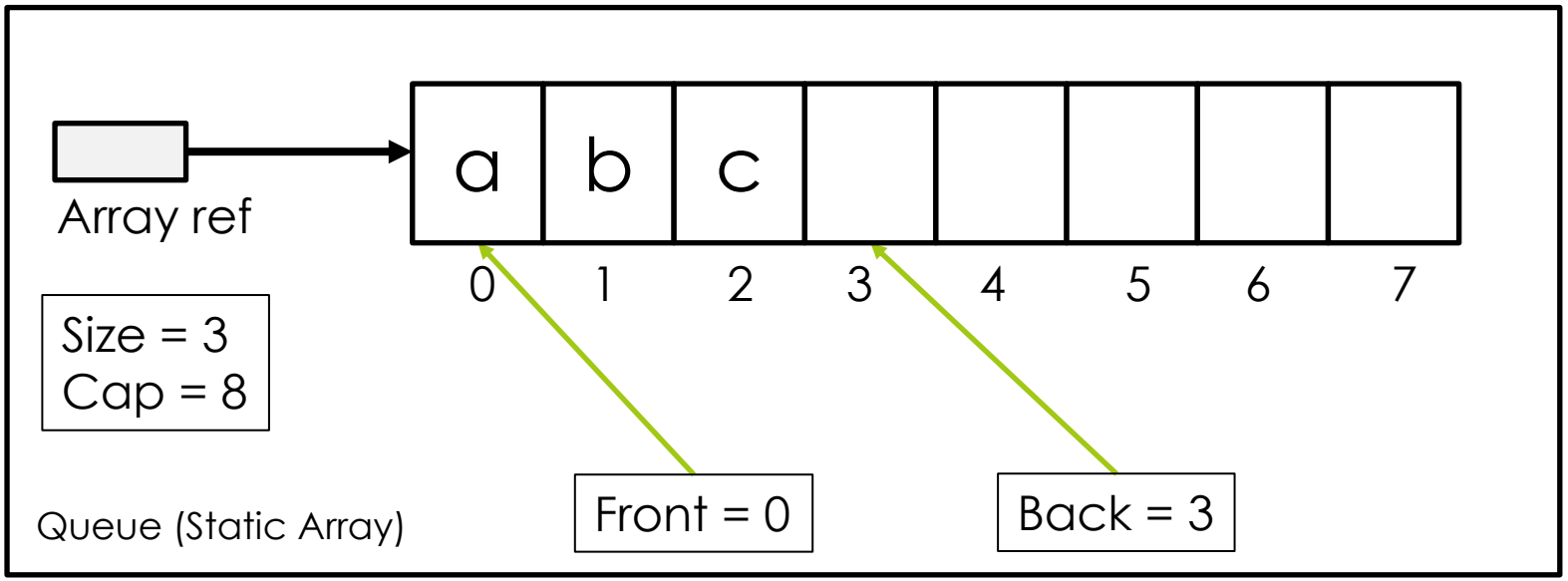
Enqueue(b)

Queue Implementation with Array



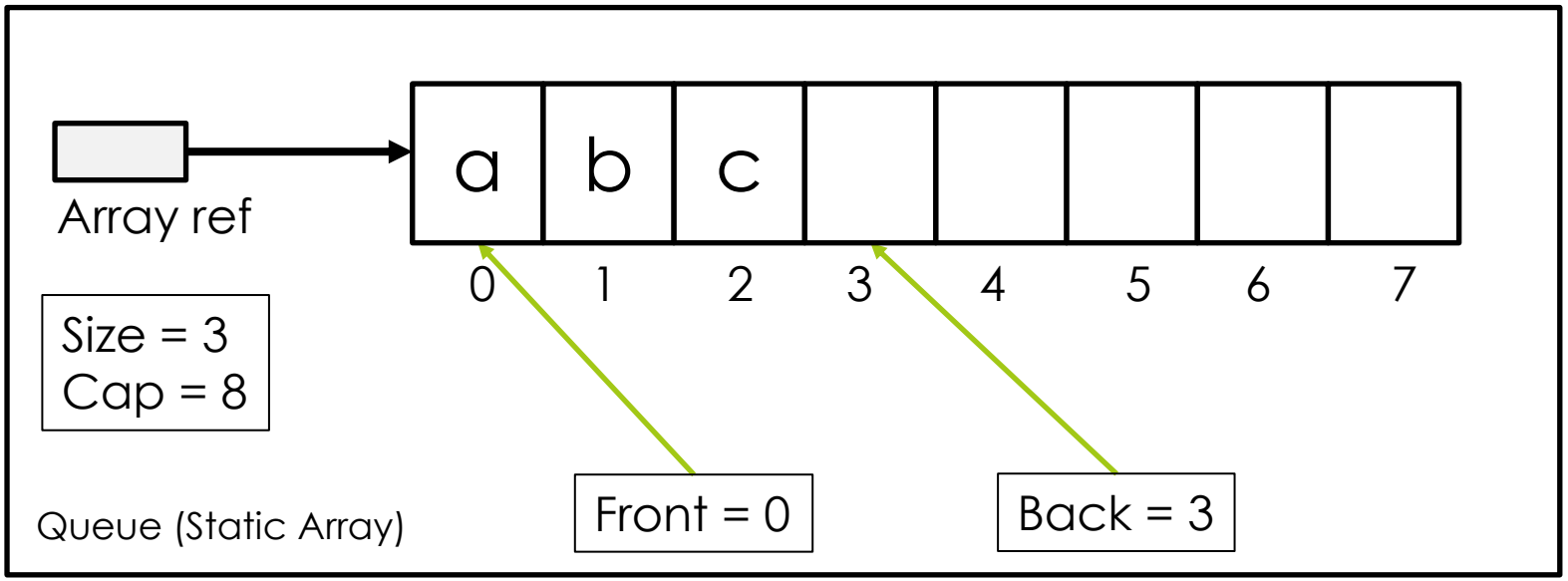
Enqueue(c)

Queue Implementation with Array



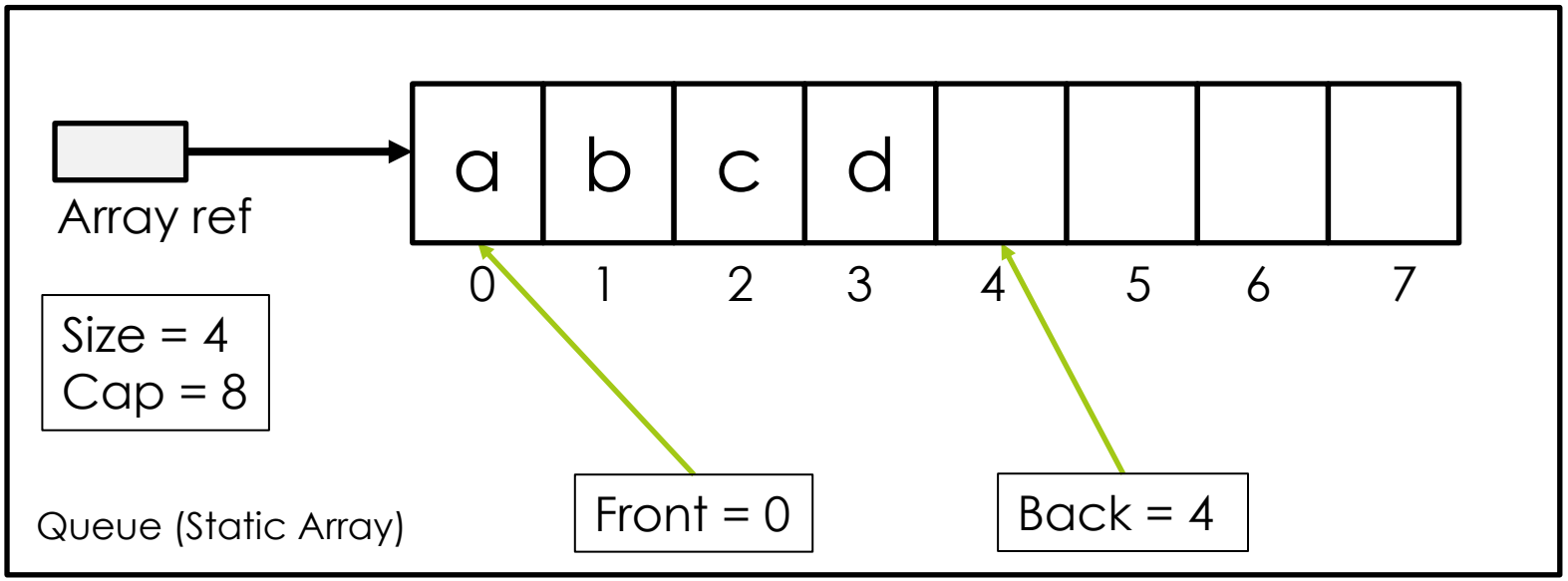
Enqueue(c)

Queue Implementation with Array



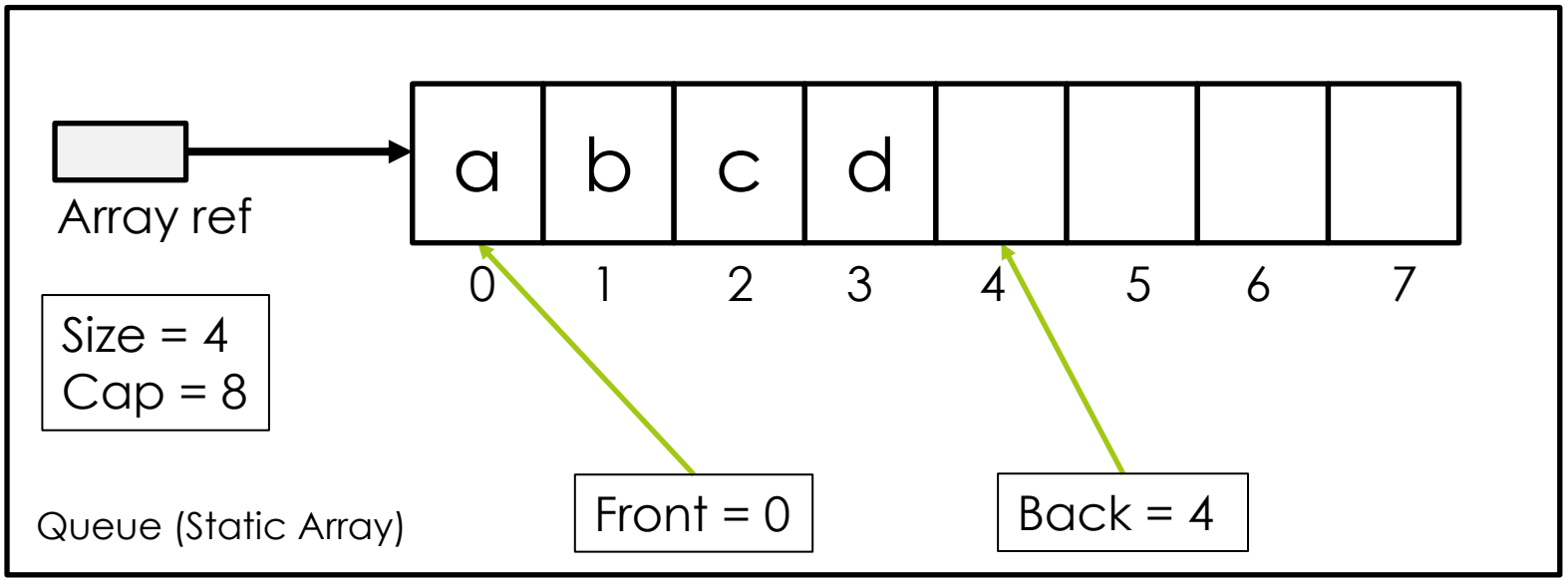
Enqueue(d)

Queue Implementation with Array



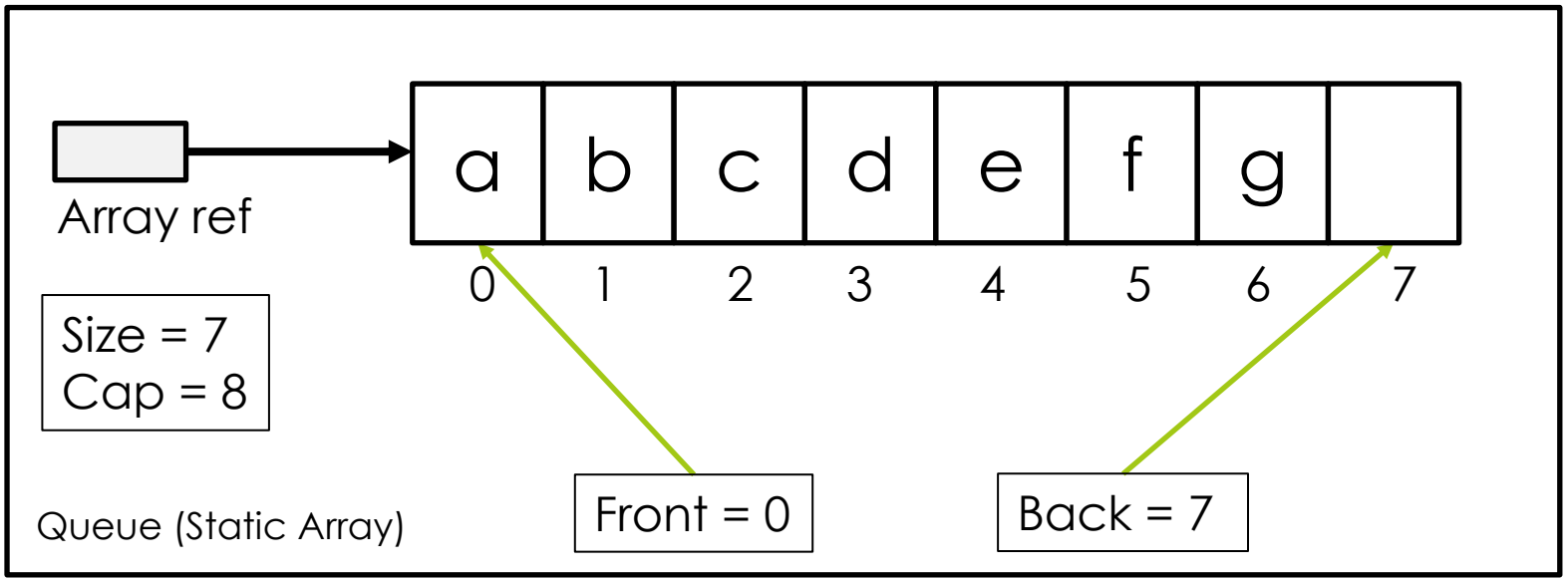
Enqueue(d)

Queue Implementation with Array



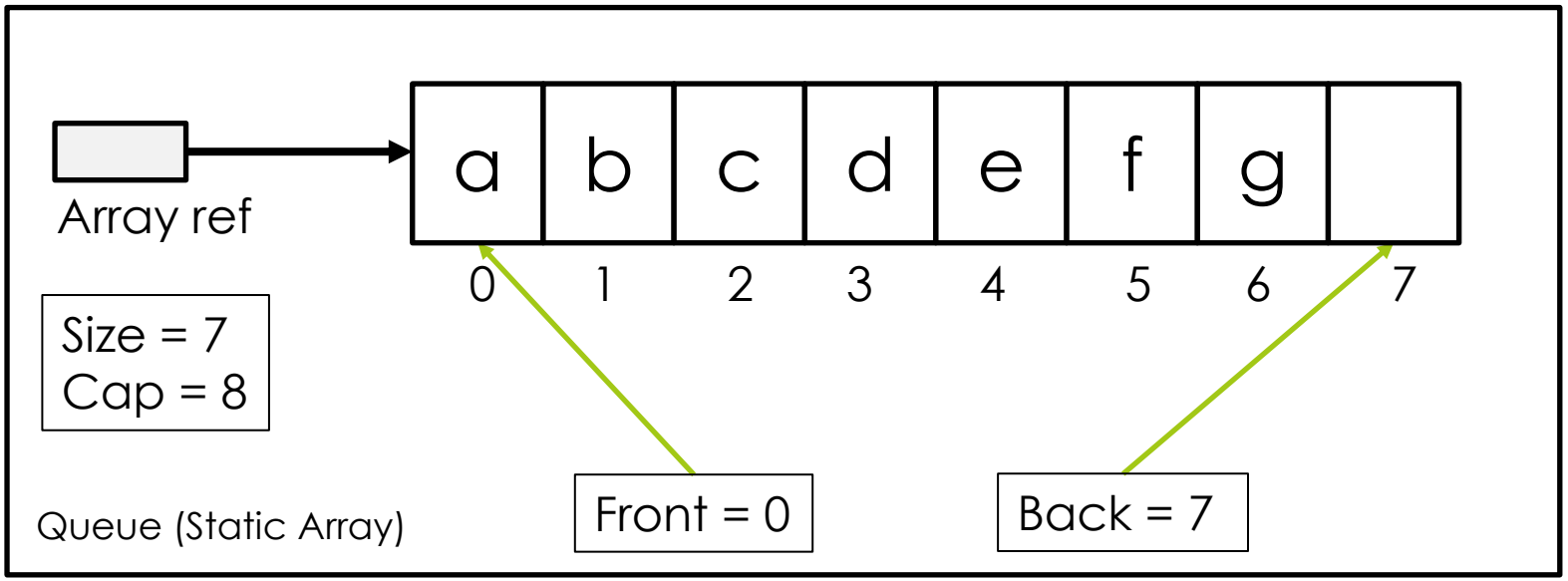
Enqueue(e); Enqueue(f); Enqueue(g);

Queue Implementation with Array



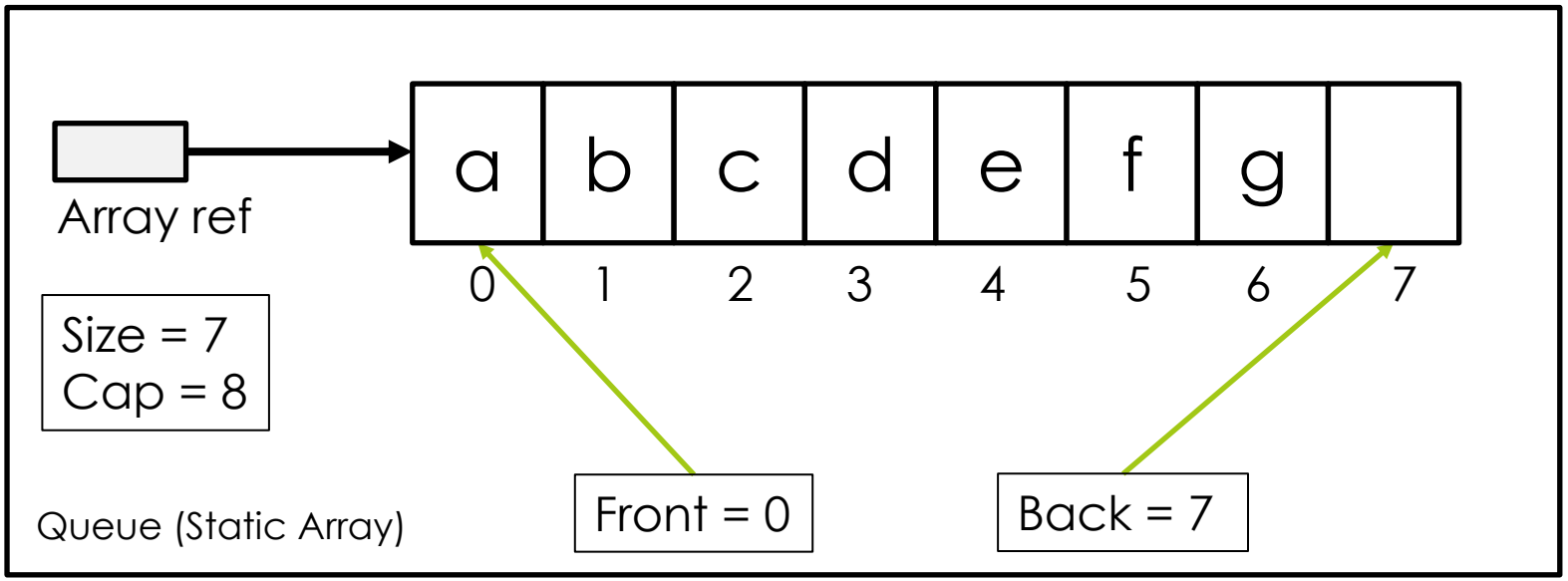
Enqueue(e); Enqueue(f); Enqueue(g);

Queue Implementation with Array



There is one more space, can we do Enqueue(h)?

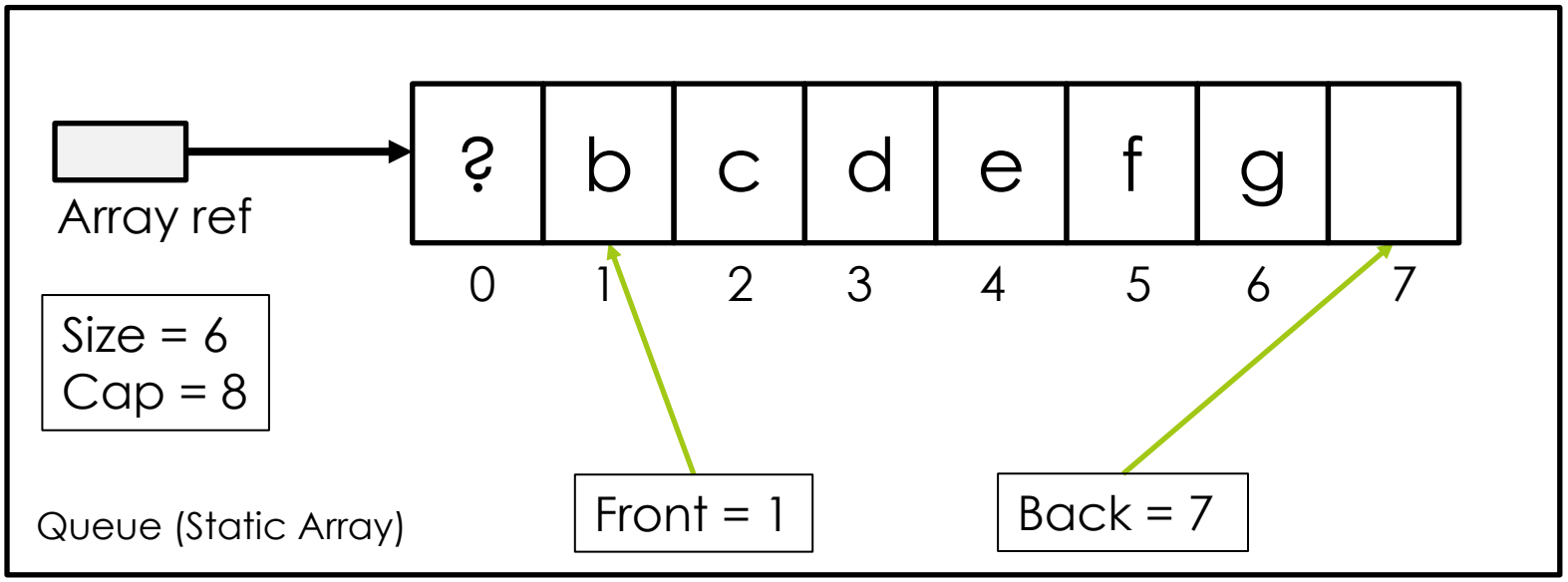
Queue Implementation with Array



Dequeue()

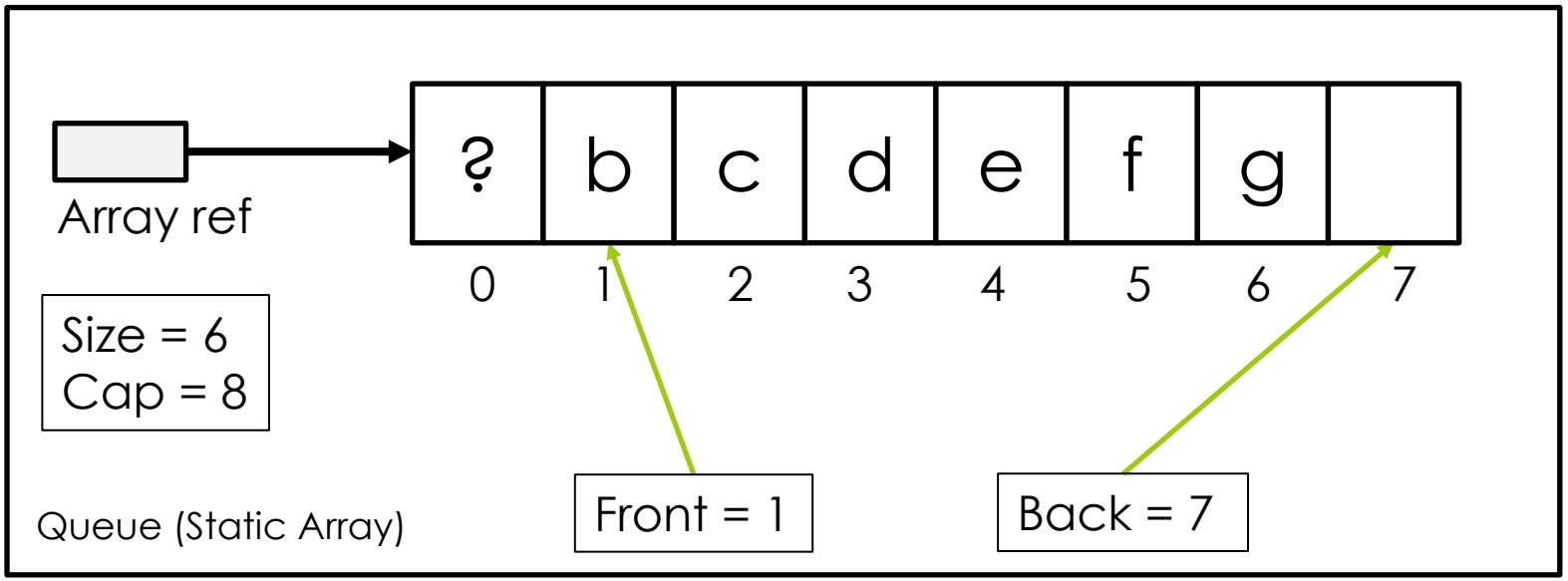
What are the corresponding Array operations?

Queue Implementation with Array



Dequeue() → a

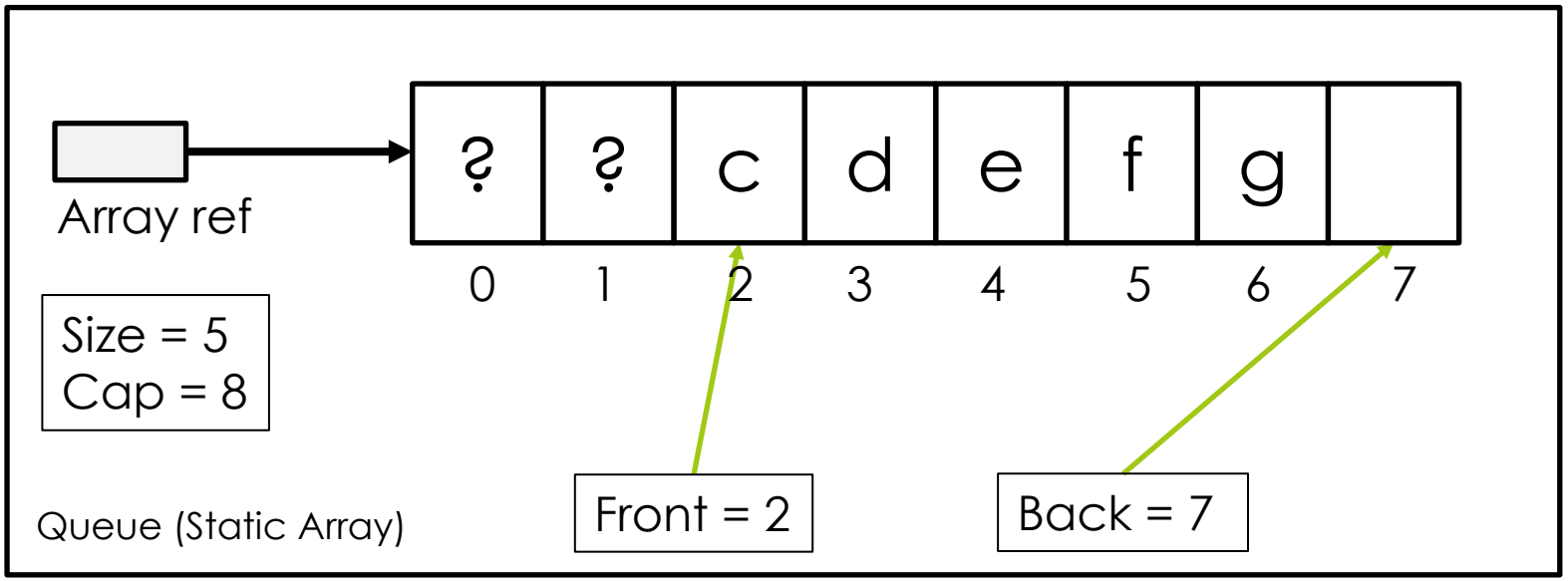
Queue Implementation with Array



Dequeue()

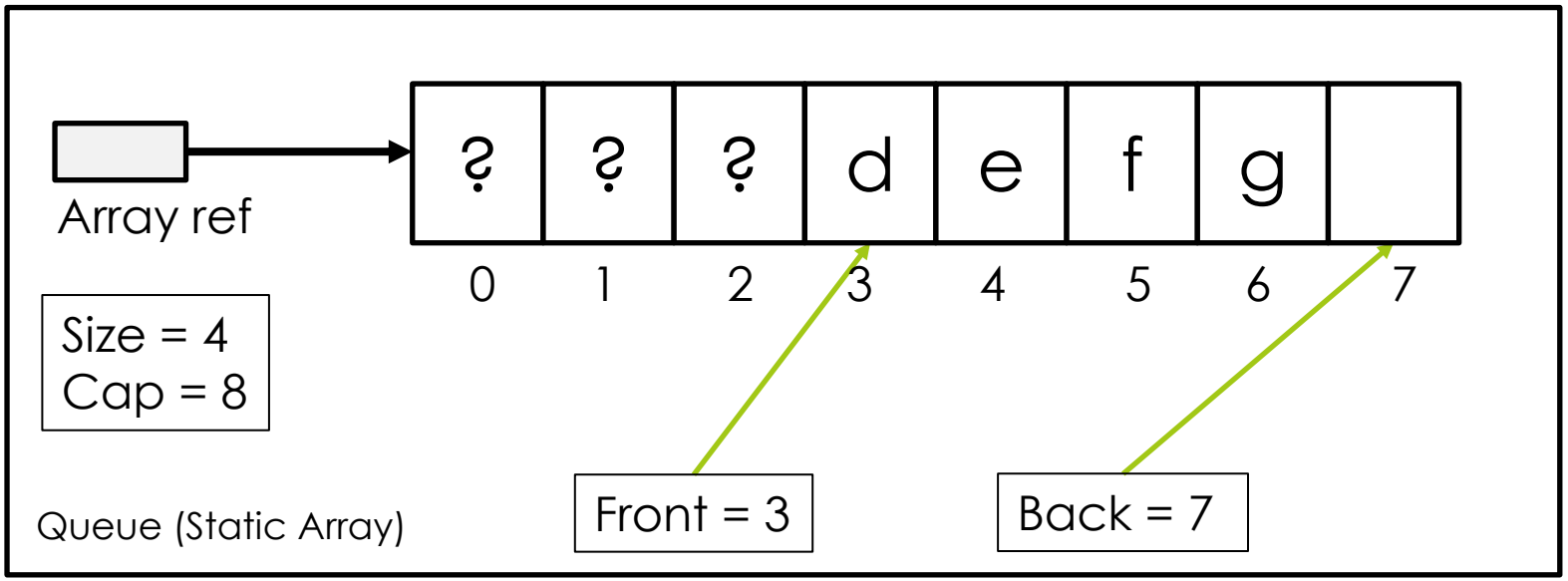
What are the corresponding Array operations?

Queue Implementation with Array



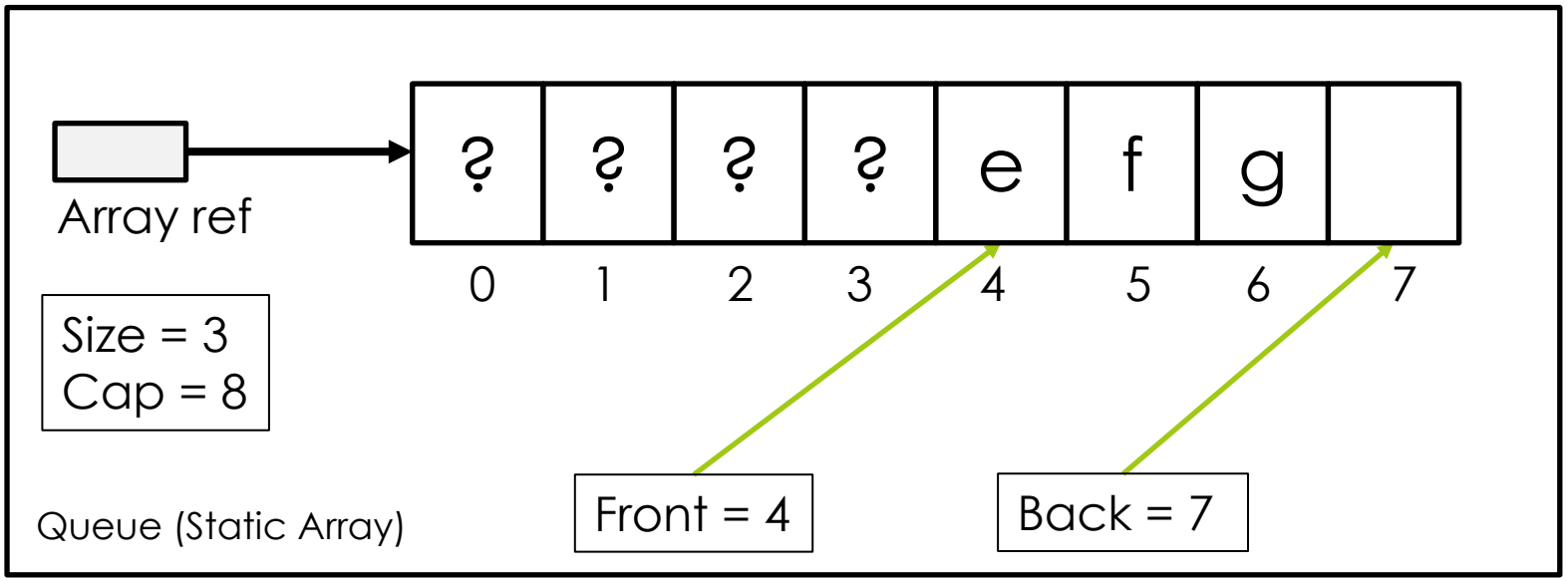
Dequeue() → b

Queue Implementation with Array



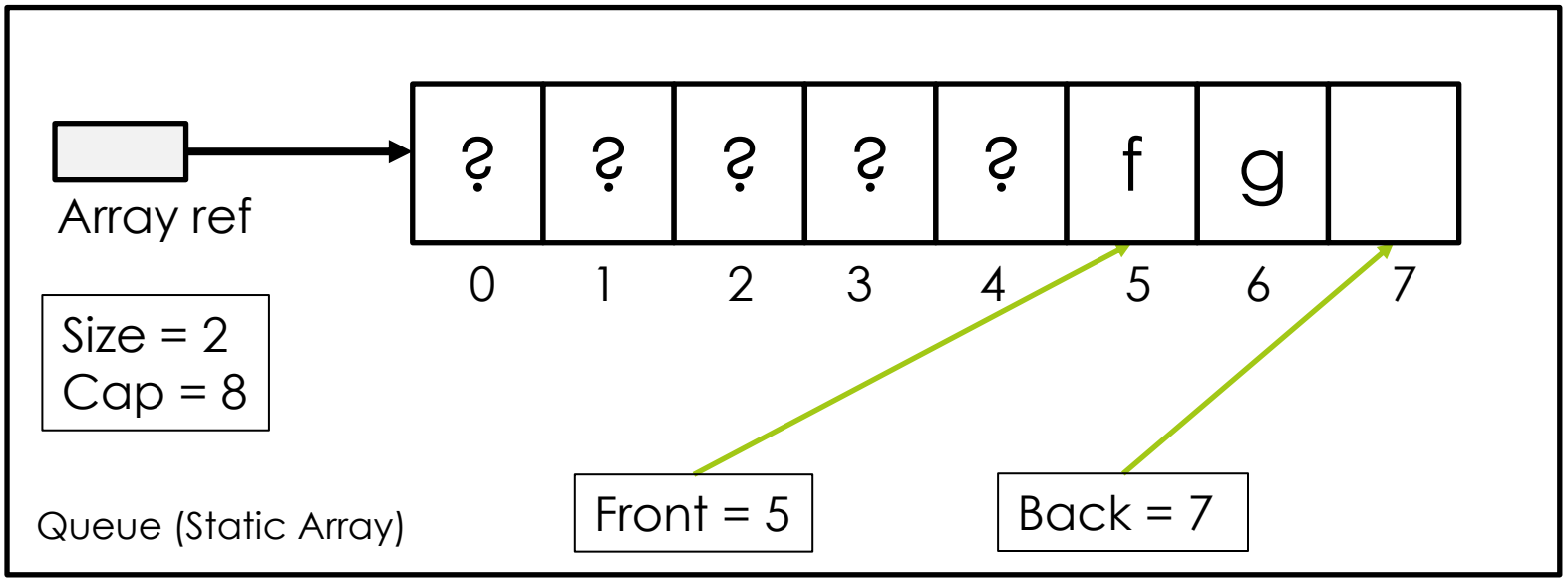
Dequeue() → c

Queue Implementation with Array



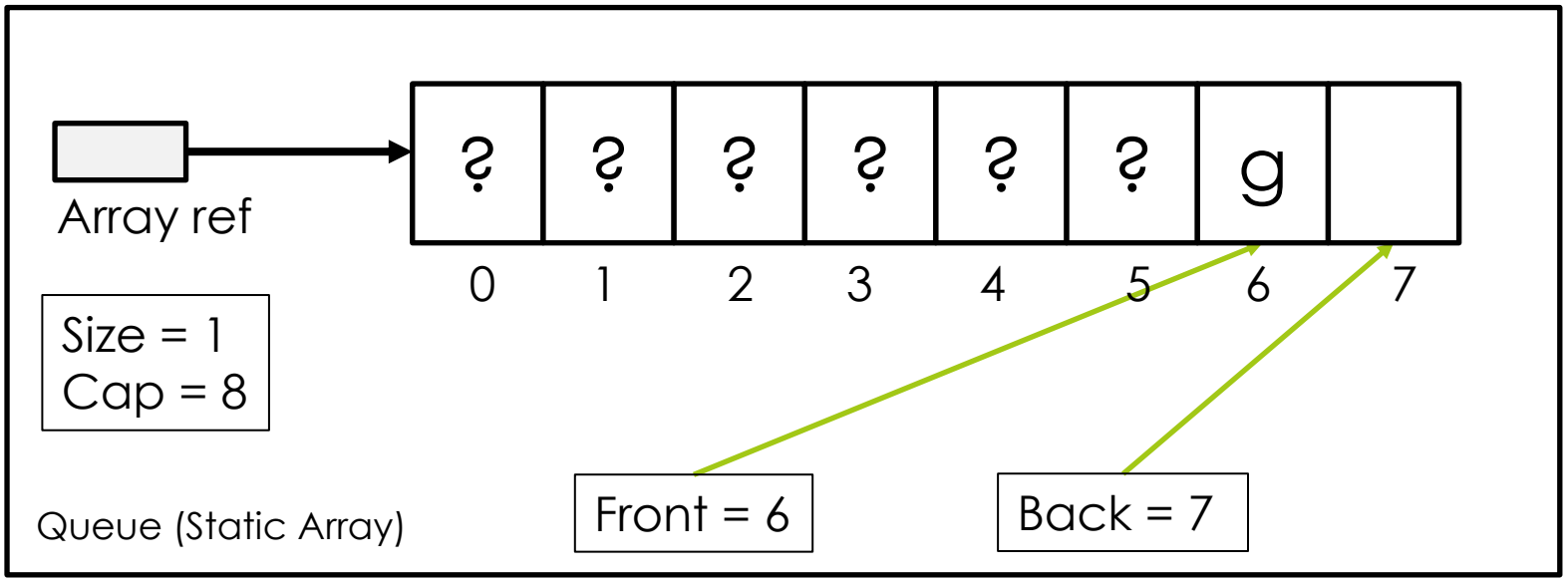
Dequeue() → d

Queue Implementation with Array



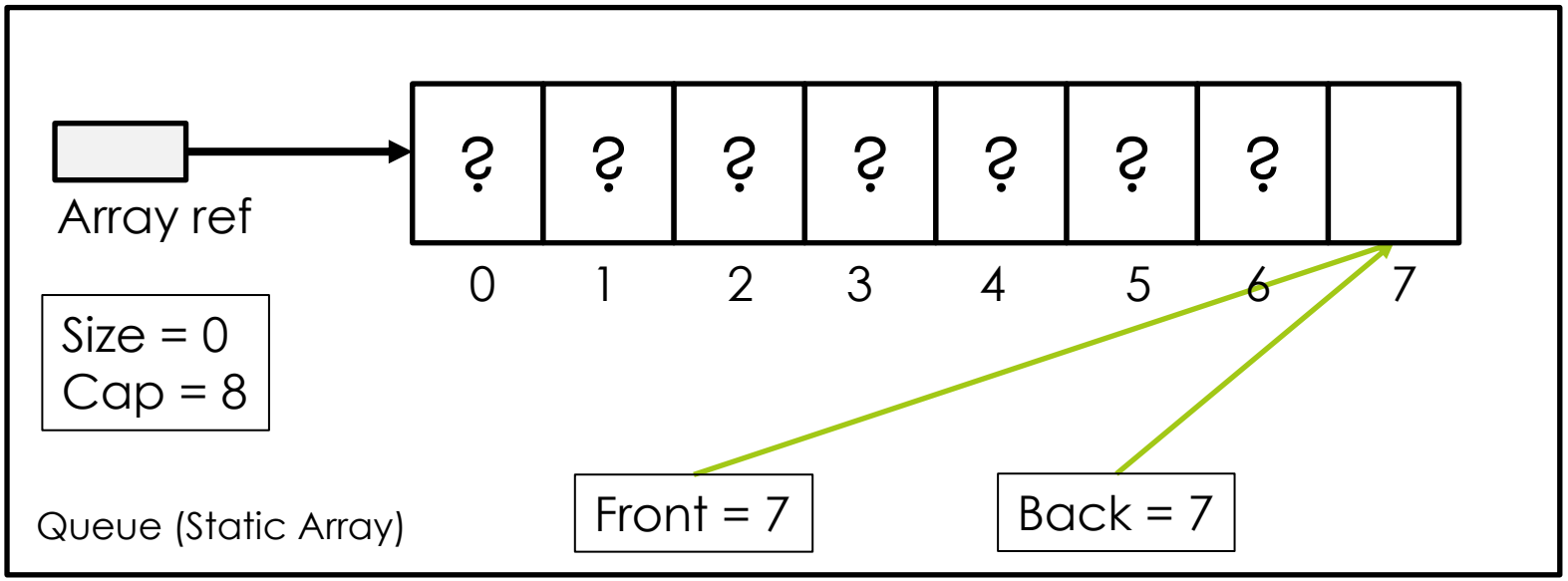
Dequeue() → e

Queue Implementation with Array



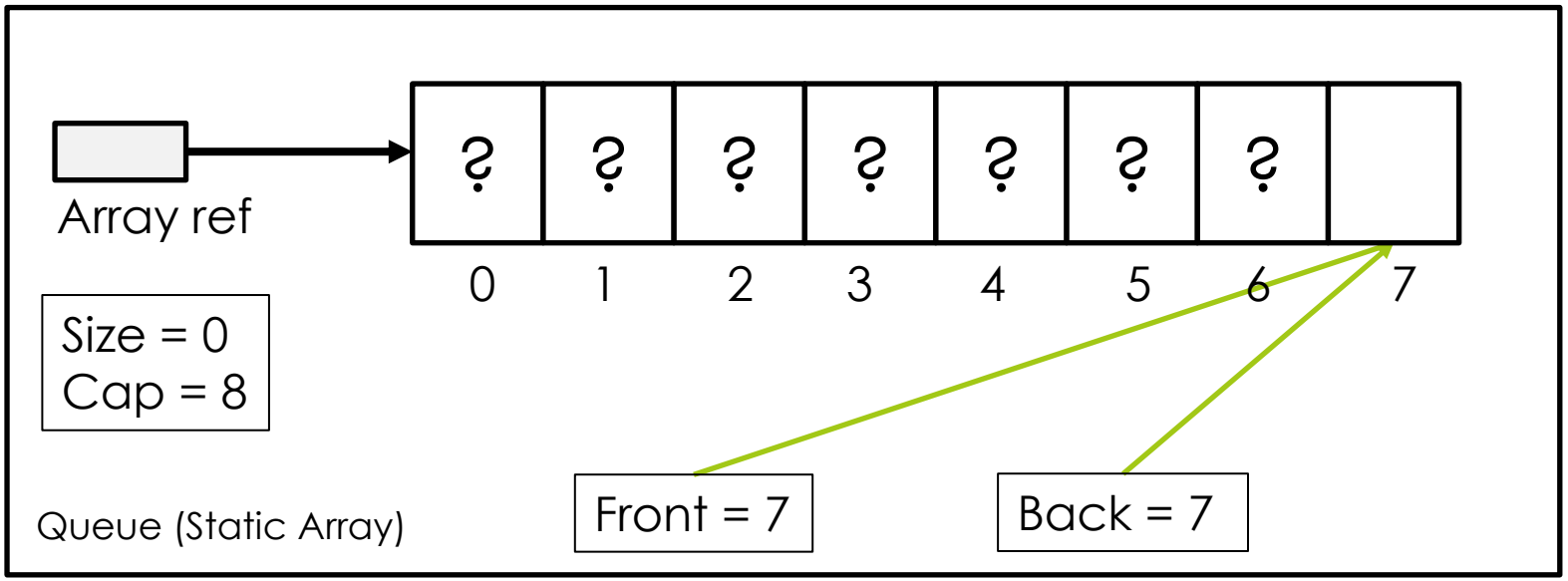
Dequeue() → f

Queue Implementation with Array



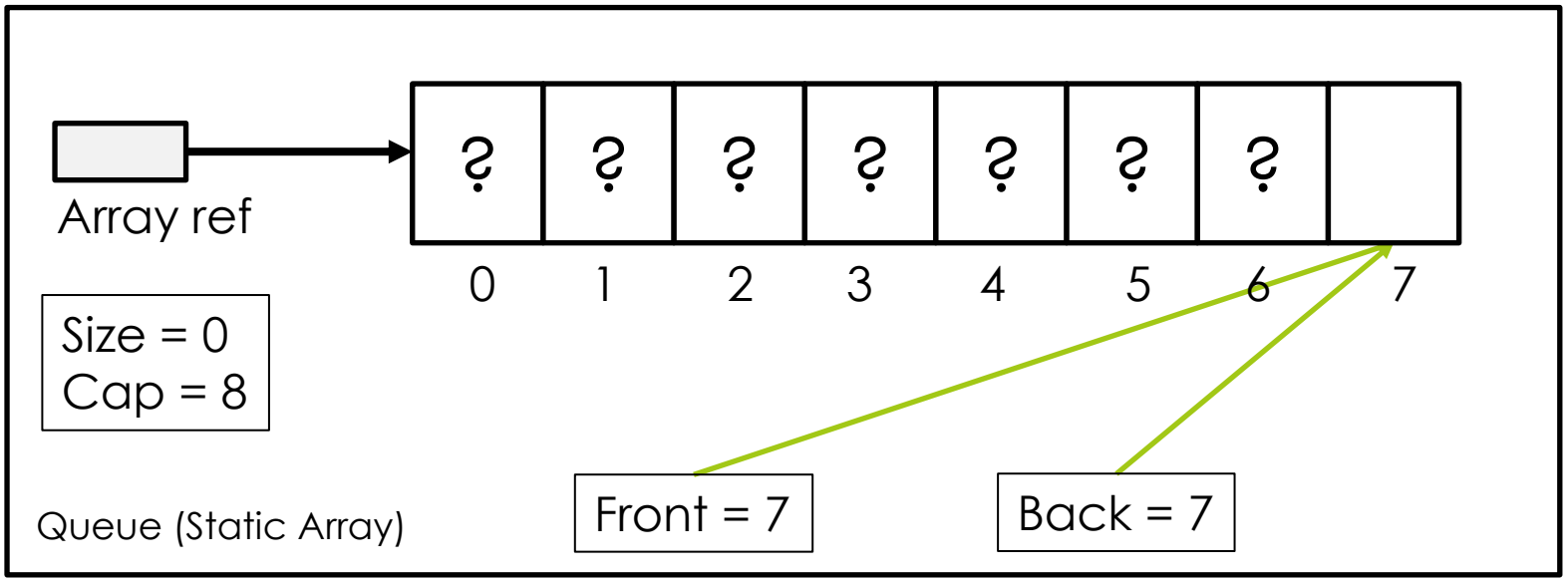
Dequeue() → g

Queue Implementation with Array



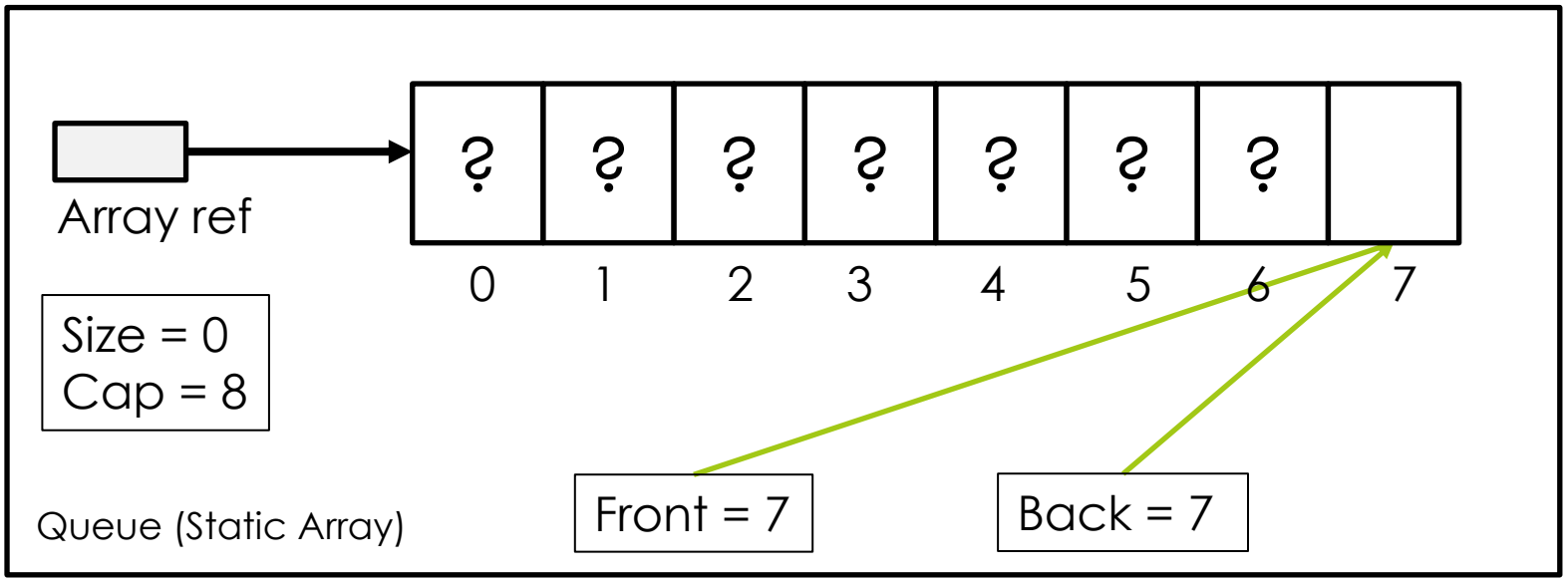
isEmpty() → true

Queue Implementation with Array



Dequeue() → ERROR

Queue Implementation with Array

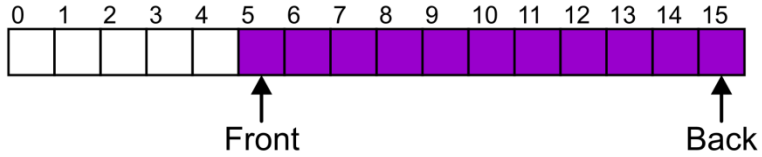


Now the pointers point to the last index, can we still do Enqueue(key)?

Full Queue?

Suppose that:

- The array capacity is 16
- We have performed 16 pushes
- We have performed 5 pops
  - The queue size is now 11



- We perform one further push

In this case, the array is not full and yet we cannot place any more objects in to the array

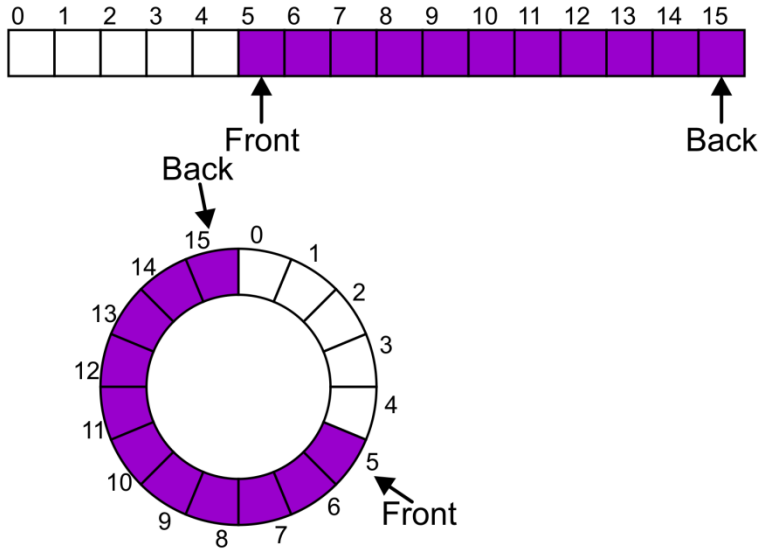
What should we do or just give up?

Circular Array

Instead of viewing the array on the range $0, \ldots, 15$, consider the indices being cyclic:

$$\ldots, 15, 0, 1, \ldots, 15, 0, 1, \ldots, 15, 0, 1, \ldots$$

This is referred to as a *circular array*

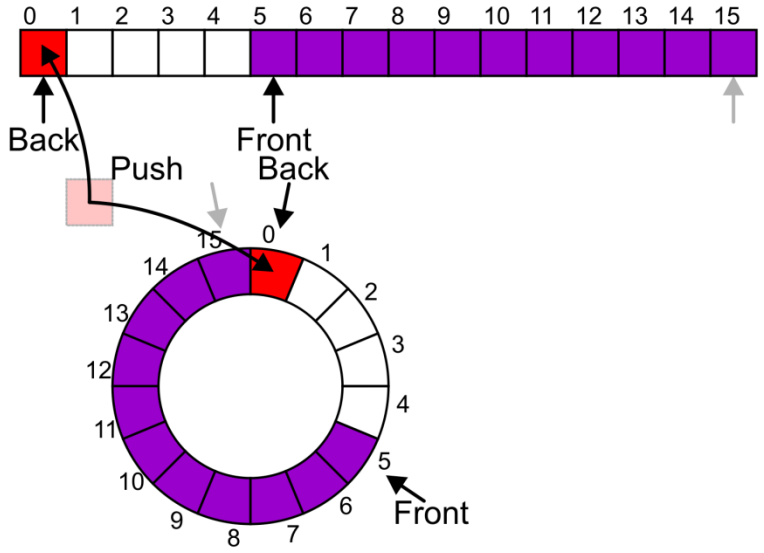


Queue as a Circular Array

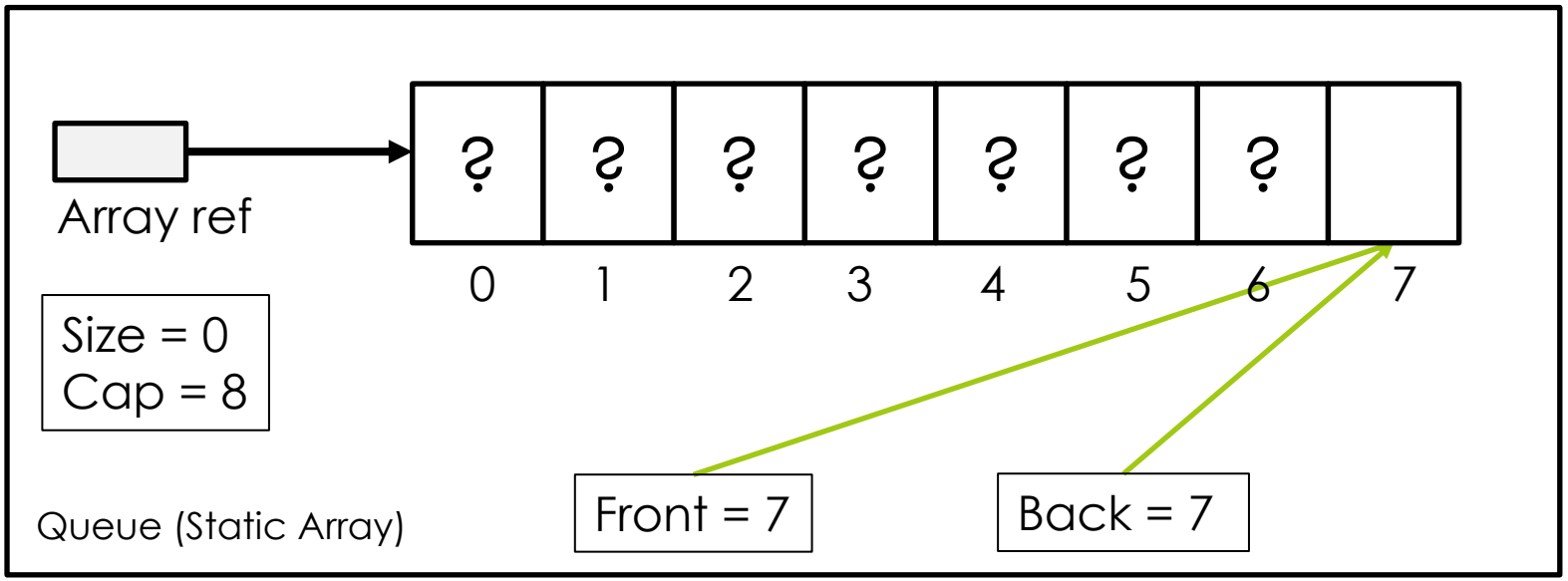
Now, the next push may be performed in the next available location of the circular array:

```
++iback;
if ( iback == capacity() ) {
    iback = 0;
}
```

How to shorten these codes to a single line without a conditional statement (if or :)?



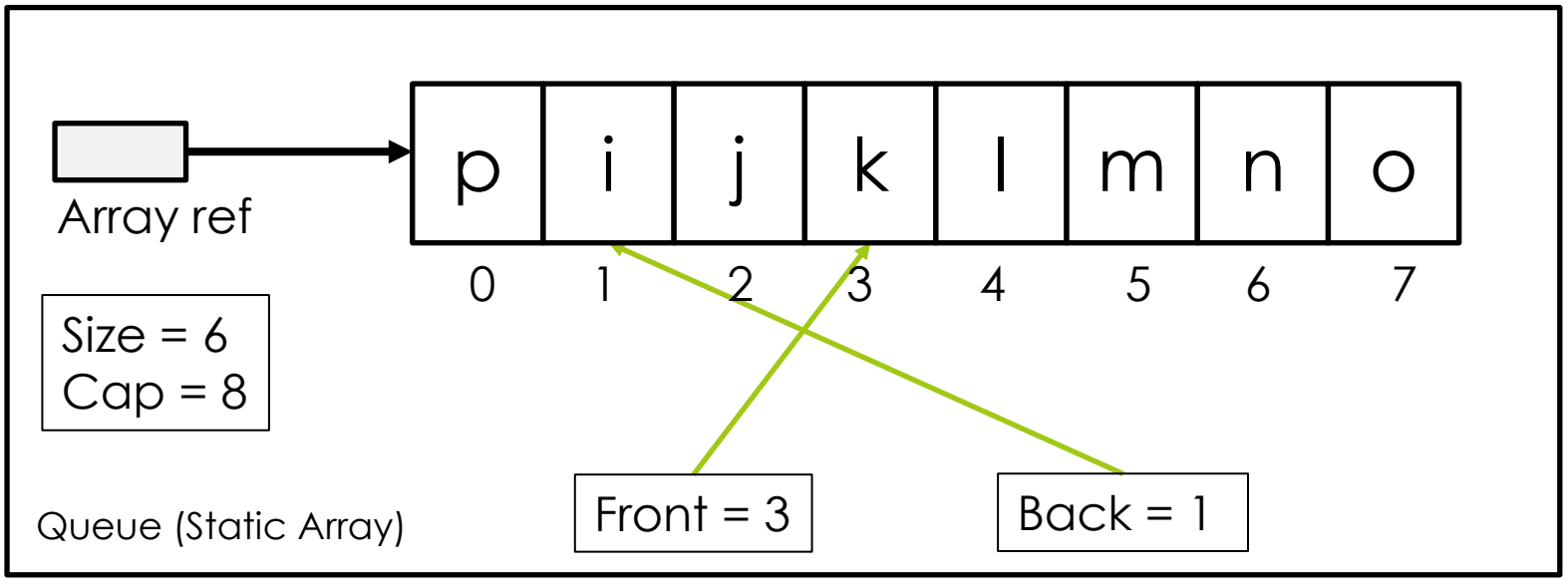
What would be the result of the following operations?



Enqueue(g); Enqueue(h); Enqueue(i); Enqueue(j);
Enqueue(k); Enqueue(l); Enqueue(m);
Pop(); Pop(); Pop(); Pop();
Enqueue(n); Enqueue(o); Enqueue(p);

Size = ?, Cap = ?, Back = ?, Front=?

What would be the result of the following operations?



Is the Queue full?
What are the garbage?

Can you return true in the isFull() if Back == Front?

Exceptions

As with a stack, there are a number of options which can be used if the array is filled

If the array is filled, we have five options:

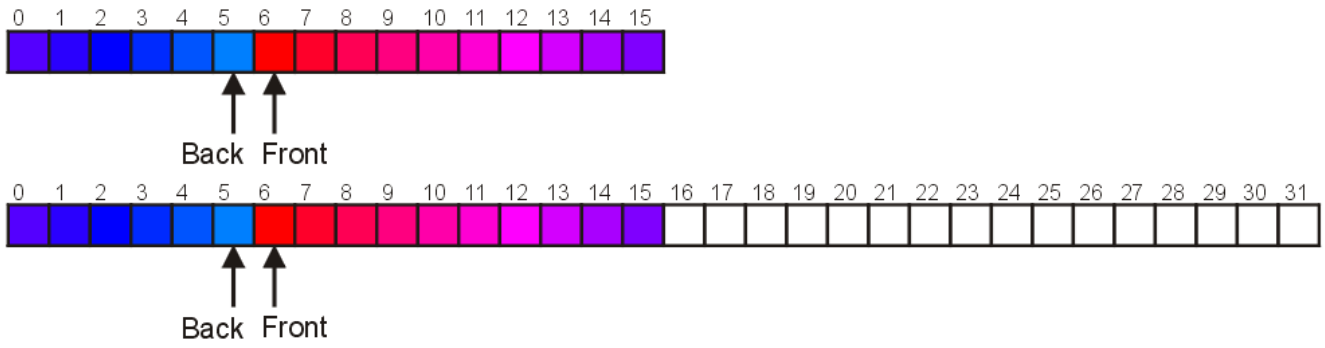
- Increase the size of the array
- Throw an exception
- Ignore the element being pushed
- Put the pushing process to “sleep” until something else pops the front of the queue

Include a member function **`bool full()`**

Increasing Capacity

Unfortunately, if we choose to increase the capacity, this becomes slightly more complex

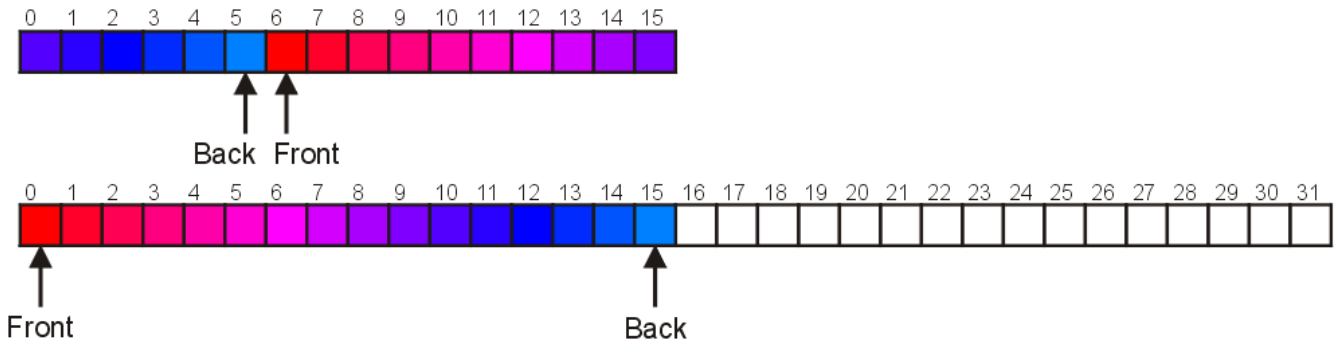
- A direct copy does not work:



Increasing Capacity

An alternate solution is normalization:

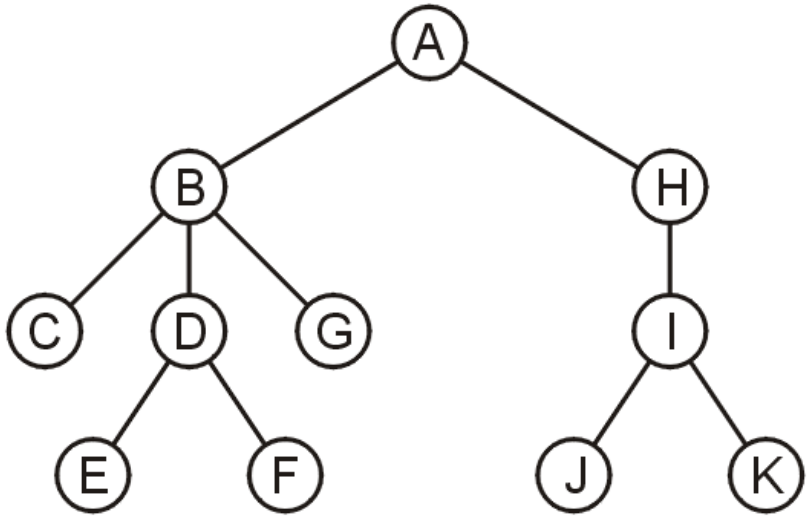
- Map the front back at position 0
- The next push would then occur in position 16



Application

Another application is performing a breadth-first traversal of a directory tree

- Consider searching the directory structure

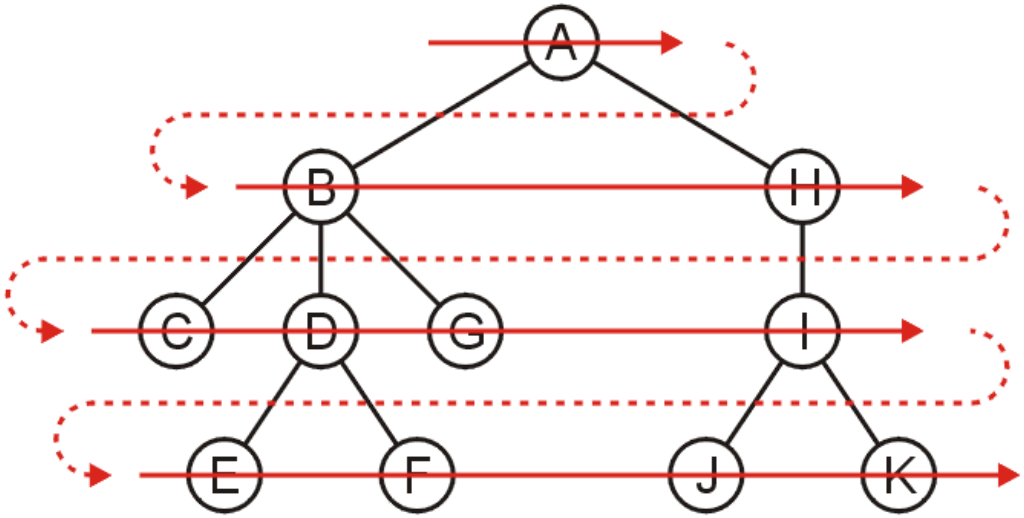


Breadth-first traversal

We would rather search the more shallow directories first then plunge deep into searching one sub-directory and all of its contents

One such search is called a *breadth-first traversal*

- Search all the directories at one level before descending a level



Output: **A**, **B**, **H**, **C**, **D**, **G**, **I**, **E**, **F**, **J**, **K**

Breadth-first traversal

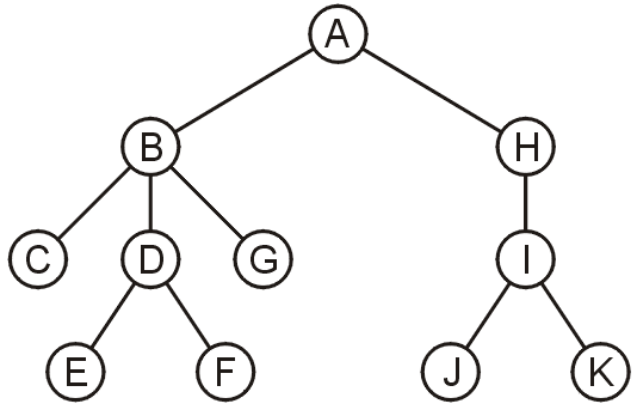
The easiest implementation is:

- Place the root directory into a queue
- While the queue is not empty:
  - Pop the directory at the front of the queue
  - Push all of its sub-directories into the queue

The order in which the directories come out of the queue will be in breadth-first order

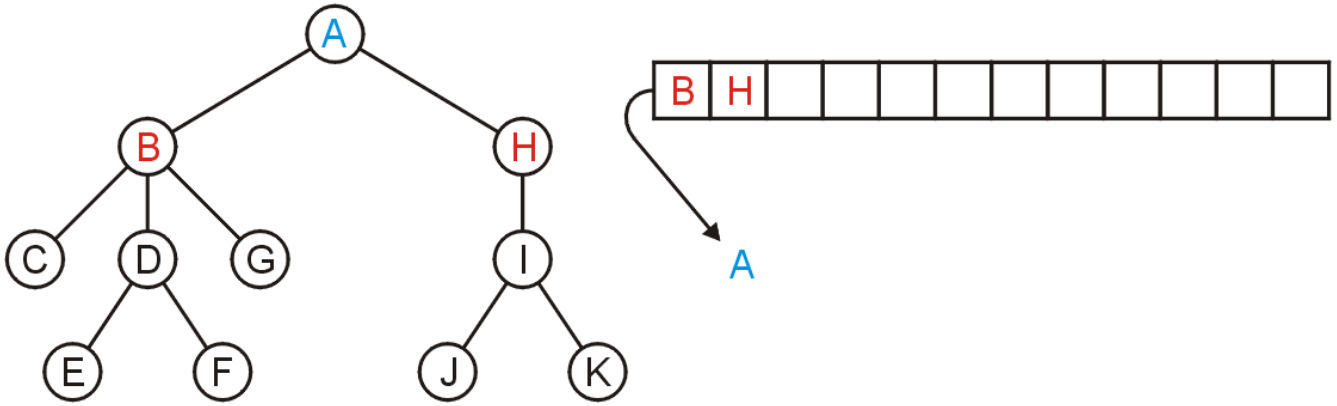
Breadth-first traversal

Push the root directory A



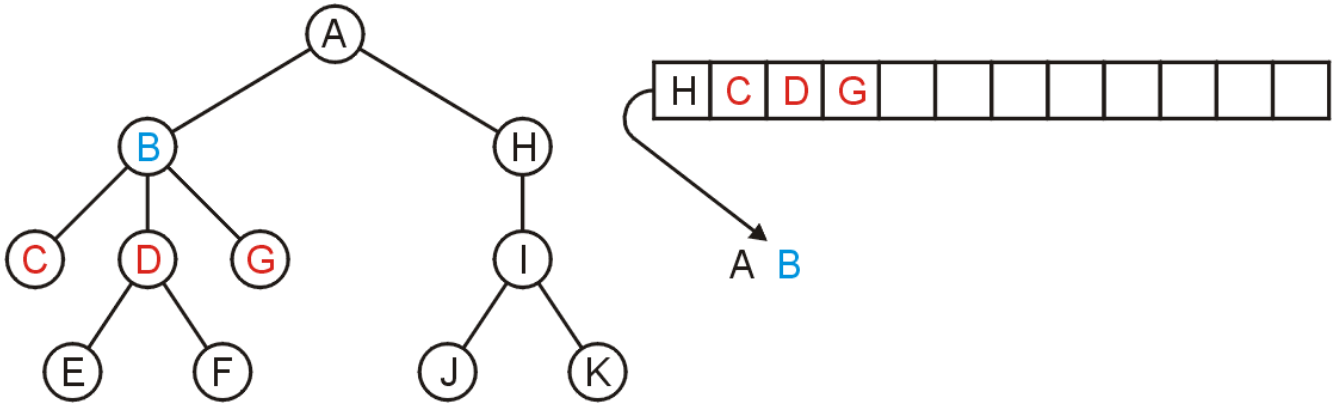
Breadth-first traversal

Pop A and push its two sub-directories: B and H



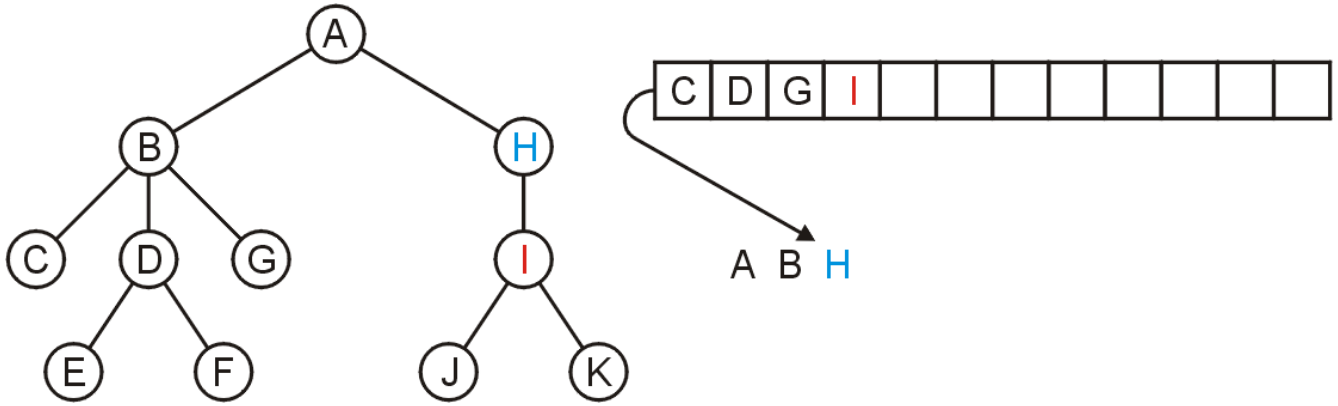
Breadth-first traversal

Pop B and push C, D, and G



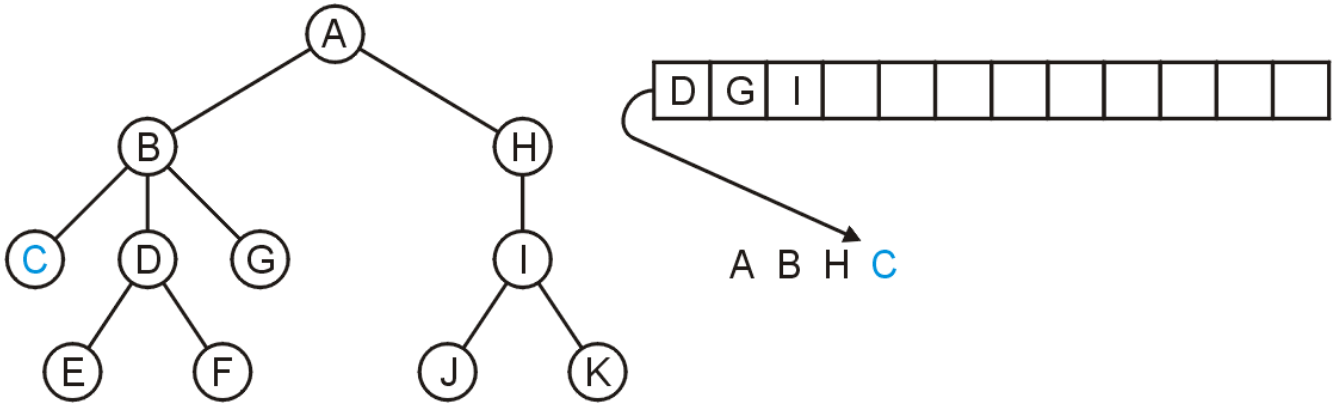
Breadth-first traversal

Pop H and push its one sub-directory I



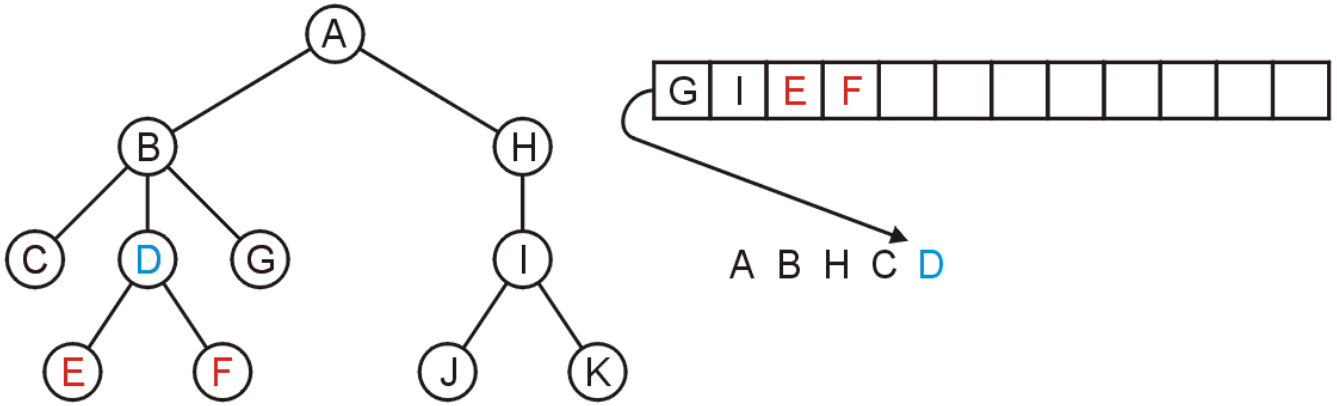
Breadth-first traversal

Pop C: no sub-directories



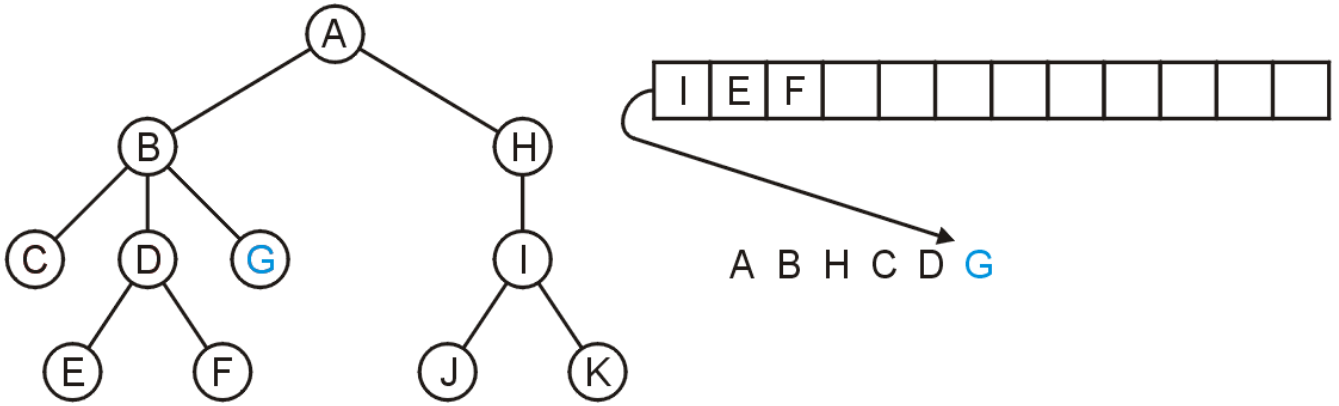
Breadth-first traversal

Pop D and push E and F



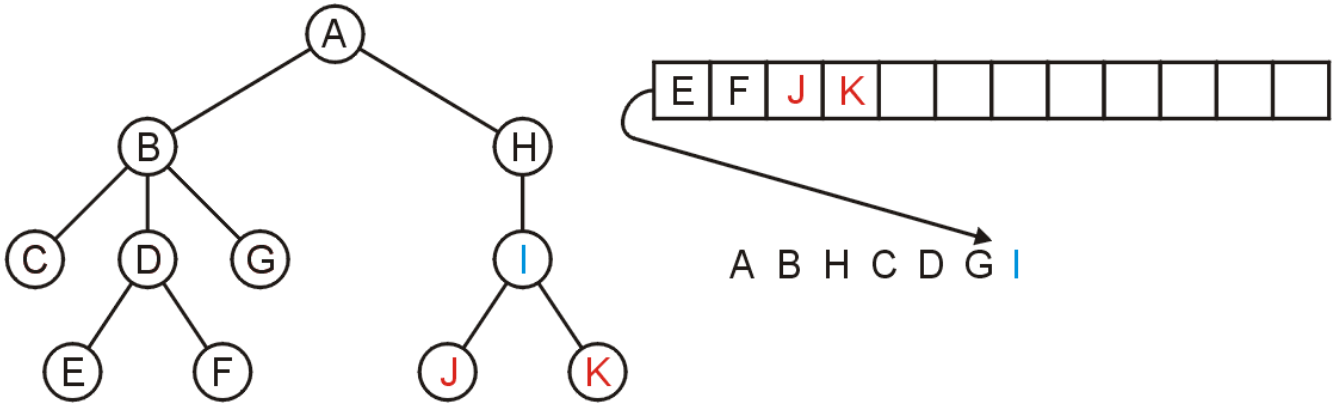
Breadth-first traversal

Pop G



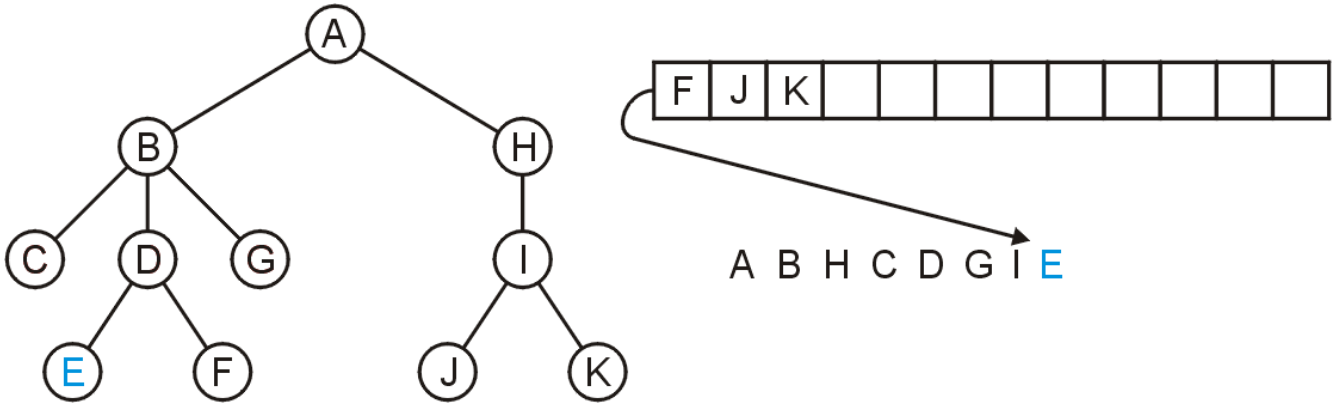
Breadth-first traversal

Pop I and push J and K



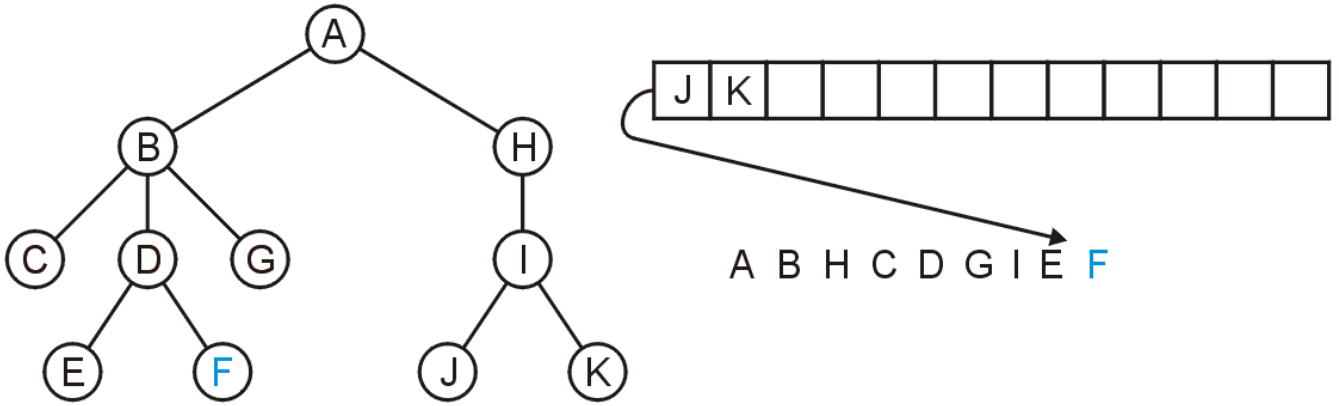
Breadth-first traversal

Pop E



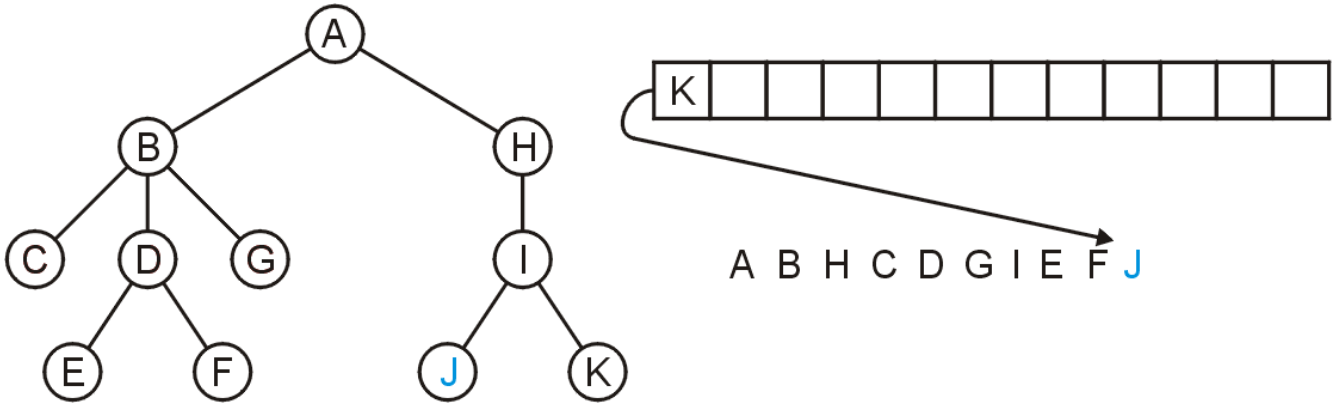
Breadth-first traversal

Pop F



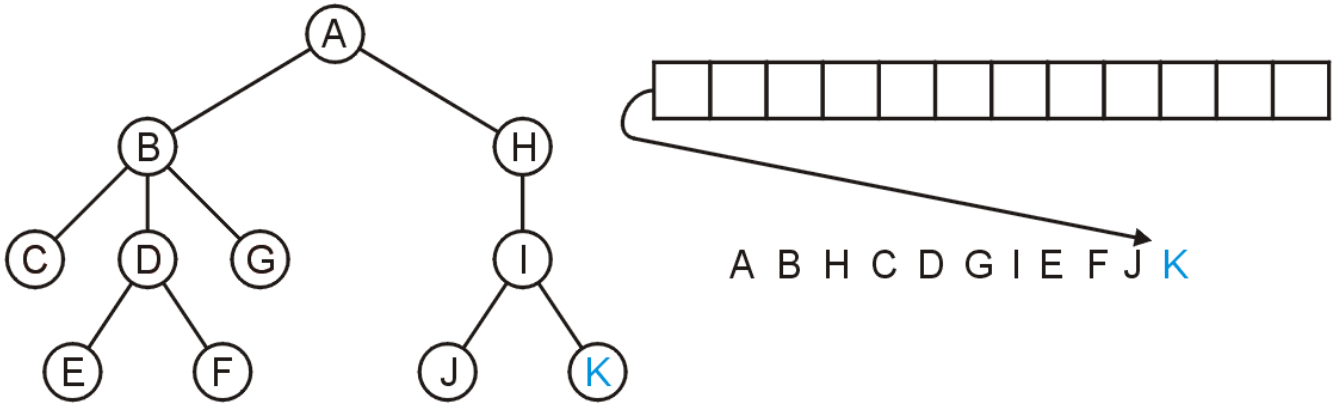
Breadth-first traversal

Pop J



Breadth-first traversal

Pop K and the queue is empty

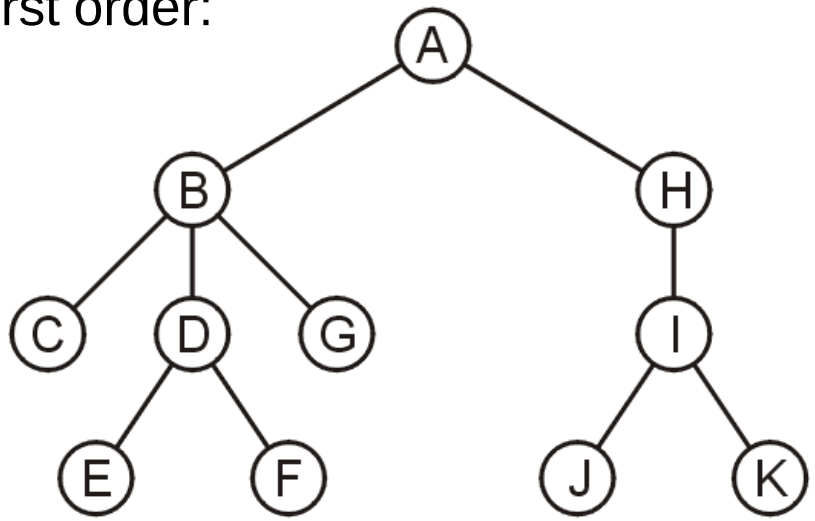


Breadth-first traversal

The resulting order

A B H C D G I E F J K

is in breadth-first order:



Summary

The queue is one of the most common abstract data structures

Understanding how a queue works is trivial

The implementation is only slightly more difficult than that of a stack

Applications include:

- Queuing clients in a client-server model
- Breadth-first traversals of trees