COMP251: DATA STRUCTURES & ALGORITHMS

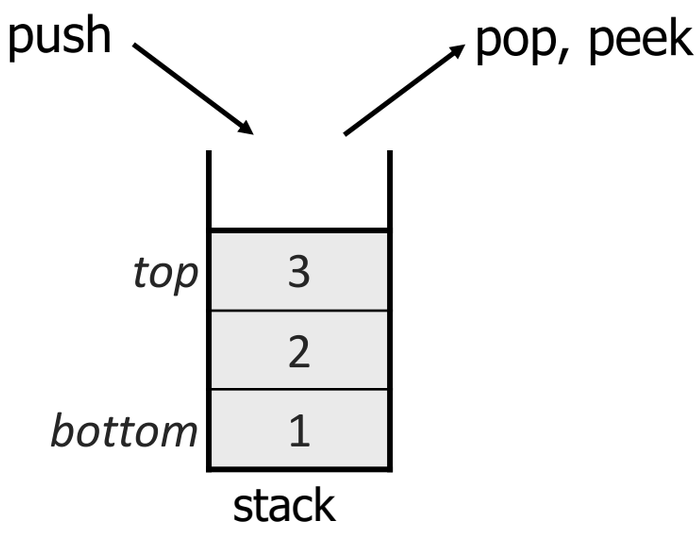
* Some slides from “Algorithms and Data Structures”
by Douglas Wilhelm Harder

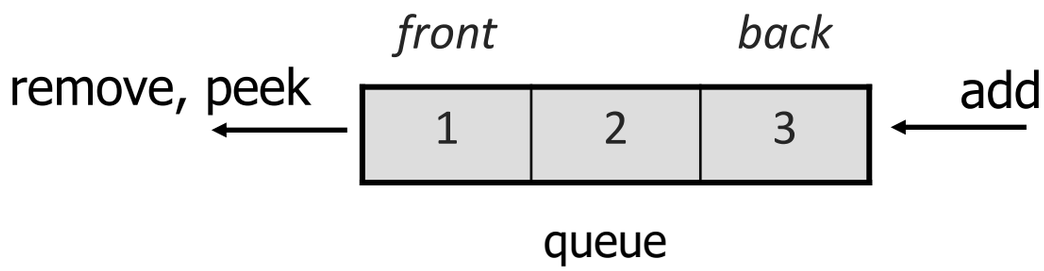
Queue

Stack and Queue

- Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.

- Two specialty collections:
  - **stack:** Retrieves elements in the reverse of the order they were added.
  - **queue:** Retrieves elements in the same order they were added.

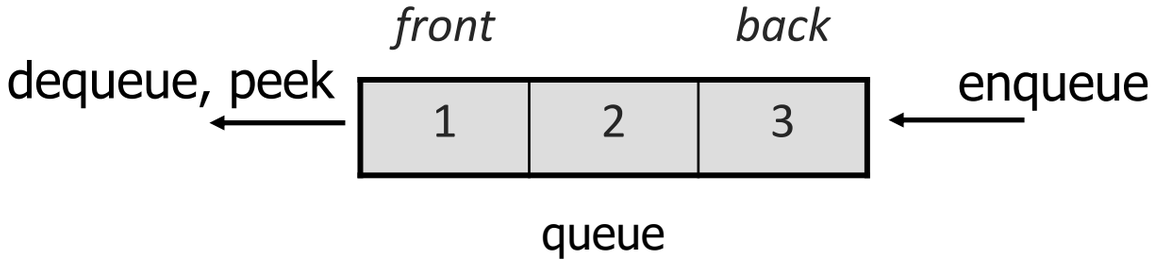




ADT Queue

- The elements are stored in order of insertion, but we do not think of them as having indexes.
- Insertion only add to the end of queue and deletion only examine at the front of queue
- Queues are *first-in–first-out* (FIFO) data structures

`enqueue(`**value**`)`	places given value at the back of queue
`dequeue()`	removes value from front of queue and returns it
`peek()`	returns front value from queue without removing it
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

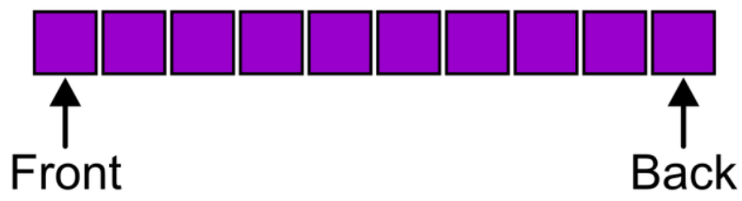


ADT Queue

first-in–first-out (FIFO) ADT

- Graphically, we may view these operations as follows:

General view of a queue:

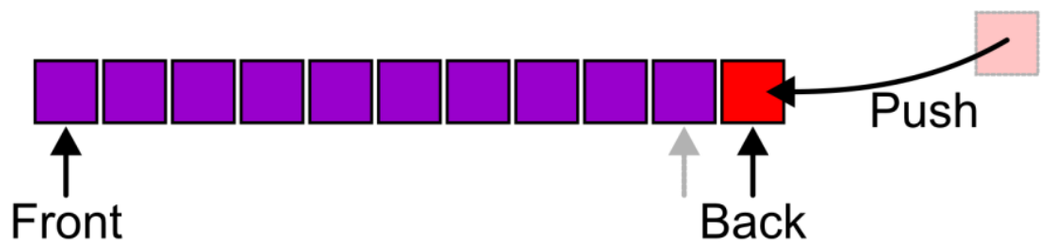


ADT Queue

first-in–first-out (FIFO) ADT

– Graphically, we may view these operations as follows:

Pushing an object to the queue
(it is usually called ***enqueue***)

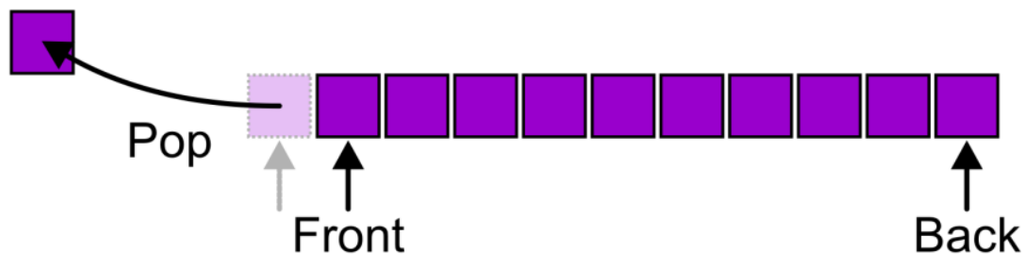


ADT Queue

first-in–first-out (FIFO) ADT

– Graphically, we may view these operations as follows:

Popping from the queue
(it is usually called ***dequeue***)



ADT Queue

There are two exceptions associated with this abstract data type:

– It is an undefined operation to call either `dequeue` or `peek` on an empty `queue`

ADT Queue

The main difference between a stack and a queue:

- A stack is only accessed from one side (the top)
- While a queue is accessed from both ends
  - From the *rear/back for adding items*
  - From the *front for removing items*.

This makes both the array and the linked-list implementation of a queue more complicated than the corresponding stack implementations.

Applications

The most common application is in client-server models

- Multiple clients may be requesting services from one or more servers
- Some clients may have to wait while the servers are busy
- Those clients are placed in a queue and serviced in the order of arrival

Most shared computer services are servers:

- Web, file, ftp, database, mail, printers, ***etc***.

Operating systems use queues to schedule CPU jobs

Implementations

We will look at two implementations of queues:

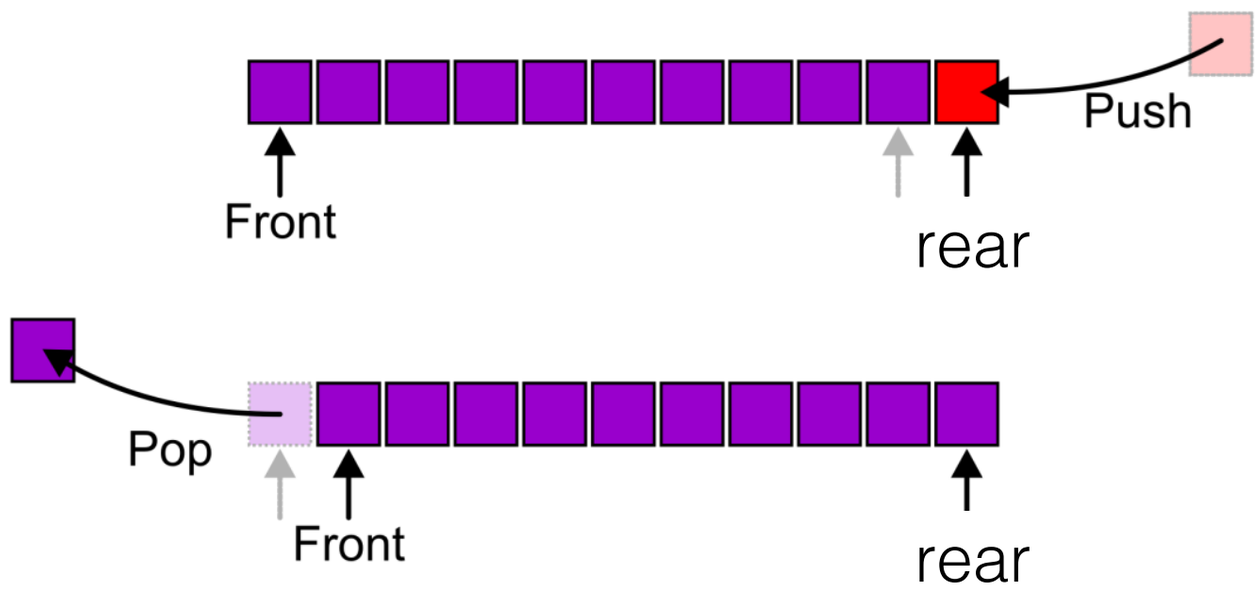
- Singly linked lists
- Circular arrays

Requirements:

- All queue operations must run in $\Theta(1)$ time

Queue: Linked List Implementation

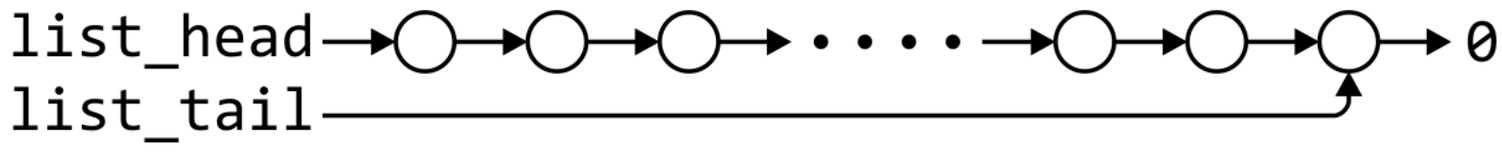
- The first decision in planning the linked-list implementation of the Queue class:
  - Which end of the list will correspond to the front of the queue.
- Recall that items need to be added (enqueued) to the rear of the queue, and removed (dequeued) from the front of the queue.
- Make our choice based on whether it is easier to push/pop (enqueue/dequeue) a node from the front or end of a linked list.



Queue: Linked List Implementation

Removal is only possible at the front with $\Theta(1)$ run time

- Front corresponds to head in the singly linked lists



	Front/1^{st}	End/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

Queue: Linked List Implementation

```
public class SListQueue<AnyType> {
     private SListNode<AnyType> front;     // front of the queue.
     private SListNode<AnyType> rear;     // back/rear of the queue.
     private int size;                    // Number of items in queue.

     public SListQueue() {     // Here's how to create an empty queue.
          front = null;
          rear = null;
          size = 0;
     }
     public int size() { return size; }
     public boolean isEmpty(){ return (size() == 0); }
     public AnyType peek() throws NoSuchElementException {
          if (isEmpty())
               throw new NoSuchElementException();
          return front.item;
     }
     public void enqueue(AnyType x) {
          SListNode<AnyType> newNode = new SListNode<AnyType>(x);
          if (rear == null) {
               rear = newNode;
               front = rear;
          }
          else {
               rear.next = newNode;
               rear = newNode;
          }
          size++;
     }
     public AnyType dequeue() throws NoSuchElementException {
          if (isEmpty())
               throw new NoSuchElementException();
          AnyType item = front.item;
          front = front.next;
          size--;
          return item;
     }
```

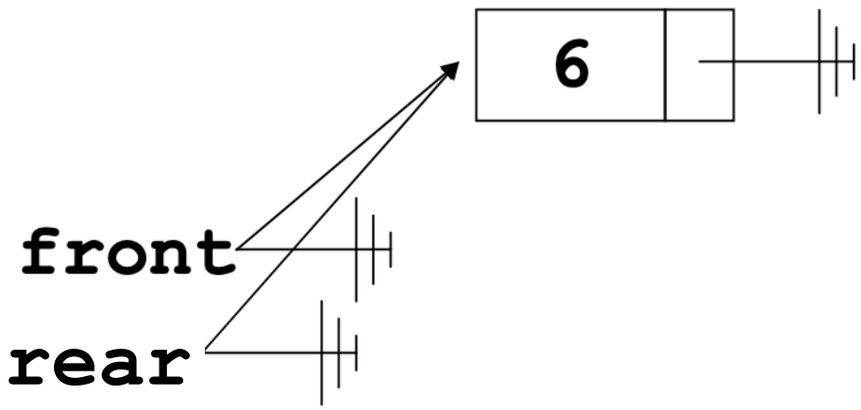
rear corresponds to tail in SList

front corresponds to head in SList

Queue: Linked List Implementation

rear corresponds to tail in SList

front corresponds to head in SList

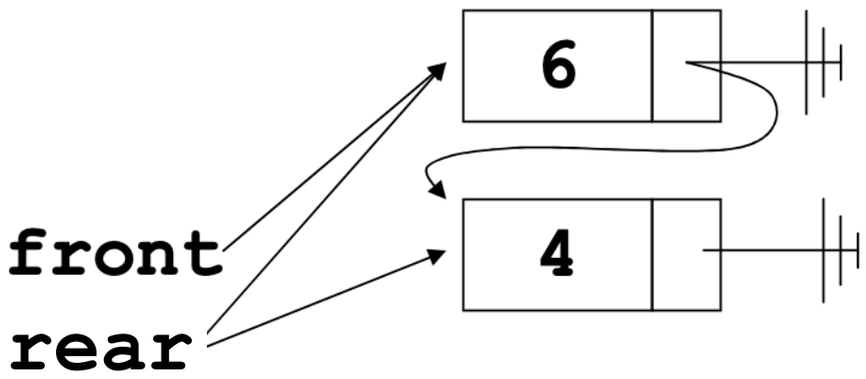



```
//Java Code
Queue q = new Queue();
q.enqueue(6);
```

Queue: Linked List Implementation

rear corresponds to tail in SList

front corresponds to head in SList

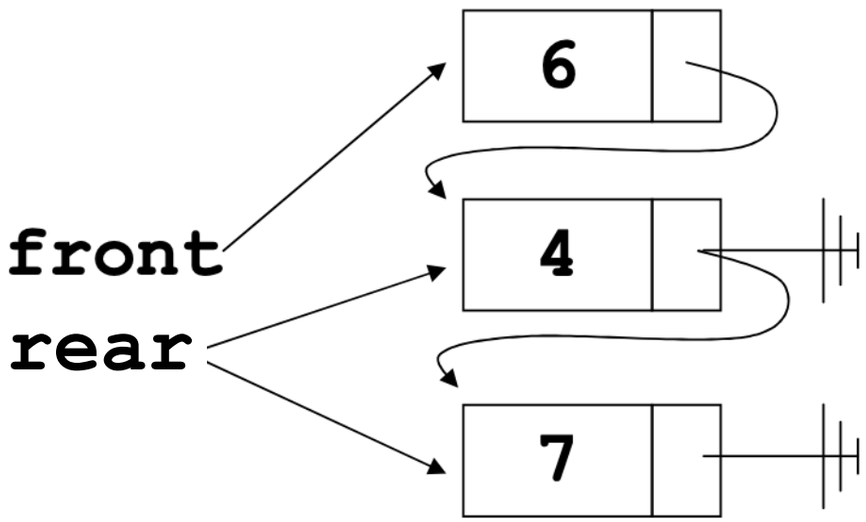



```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
```

Queue: Linked List Implementation

rear corresponds to tail in SList

front corresponds to head in SList

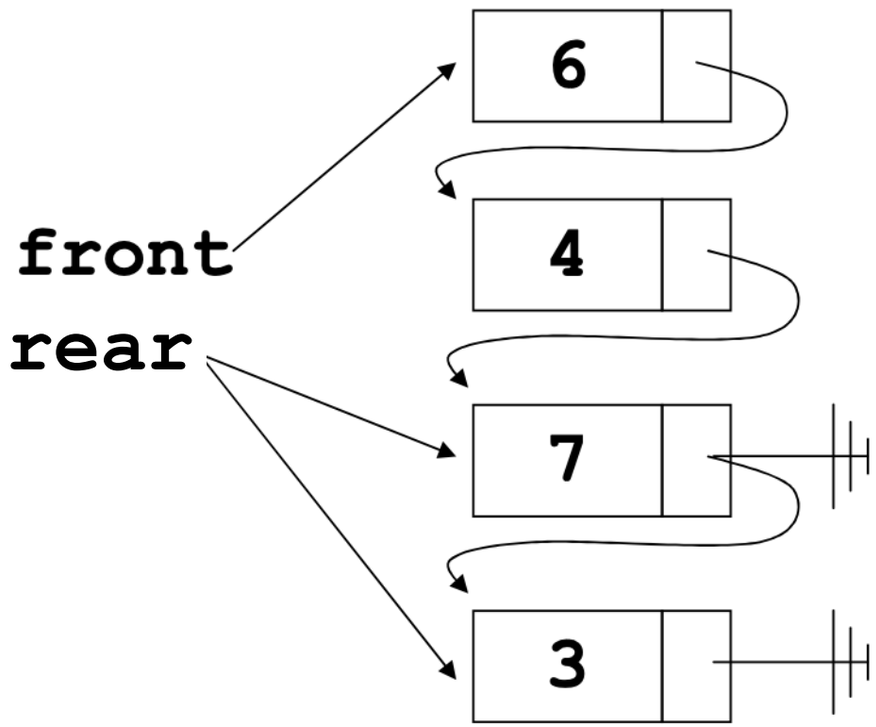



```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
```

Queue: Linked List Implementation

rear corresponds to tail in SList

front corresponds to head in SList

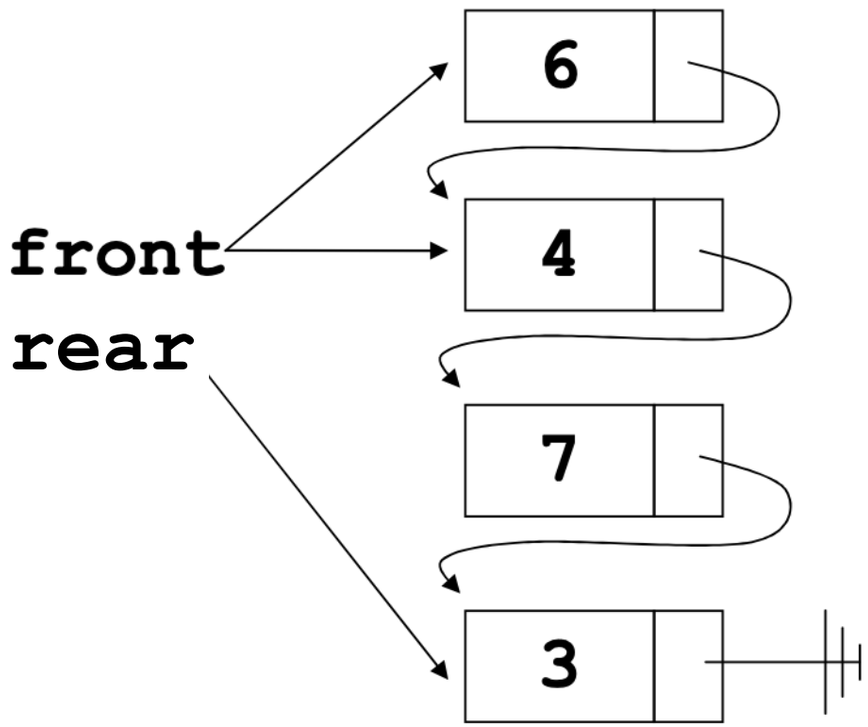



```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
q.enqueue(3);
```

Queue: Linked List Implementation

rear corresponds to tail in SList

front corresponds to head in SList




```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
q.enqueue(3);
q.dequeue();
```

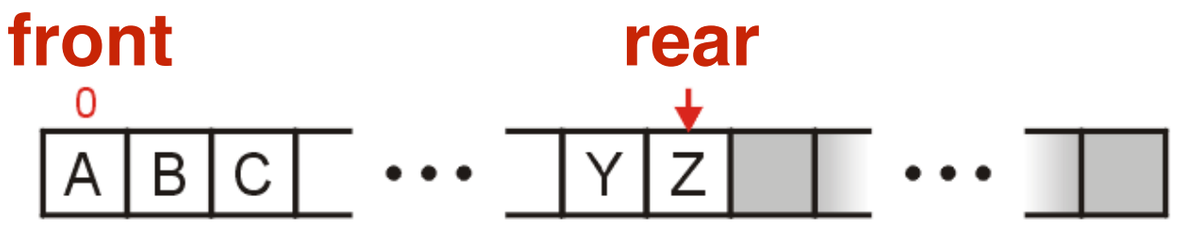
Queue

Array Implementation

Array Implementation

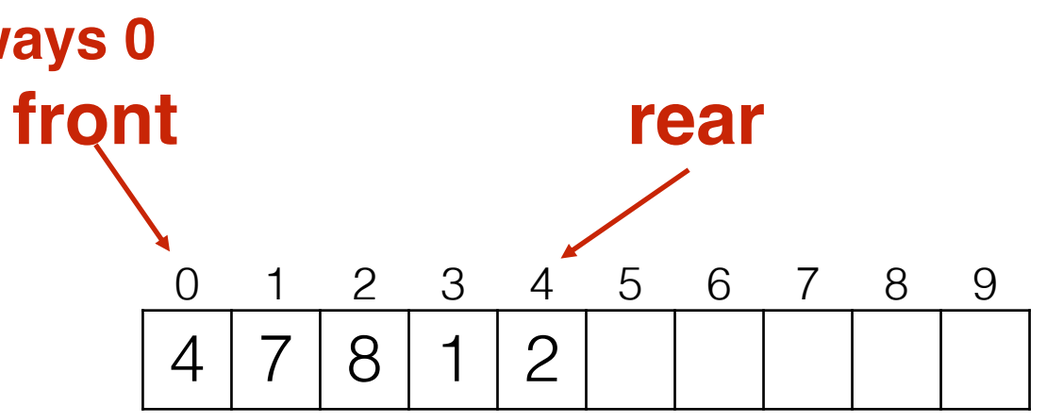
A one-ended array does not allow all operations to occur in $\Theta(1)$ time

- We choose to have front at index 0
  - Then front does not need to be updated (it is always 0)

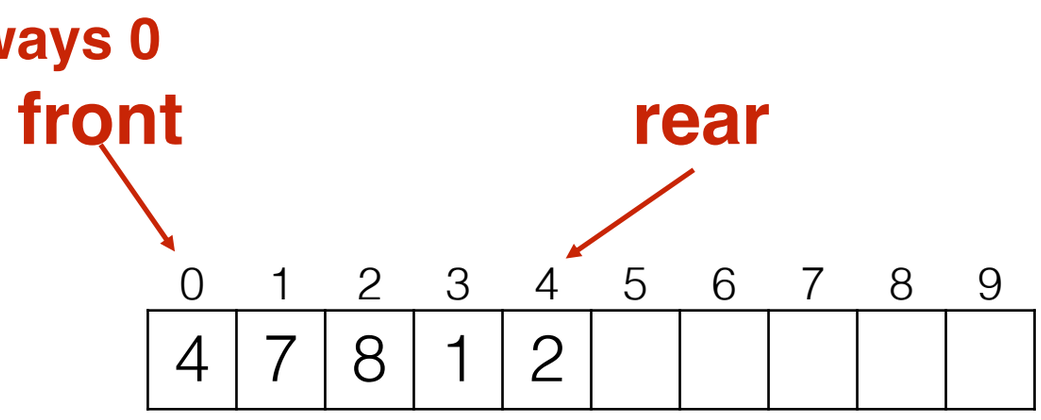


	Front/1^{st}	End/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(n)$	$\Theta(1)$
Erase	$\Theta(n)$	$\Theta(1)$

Array Implementation

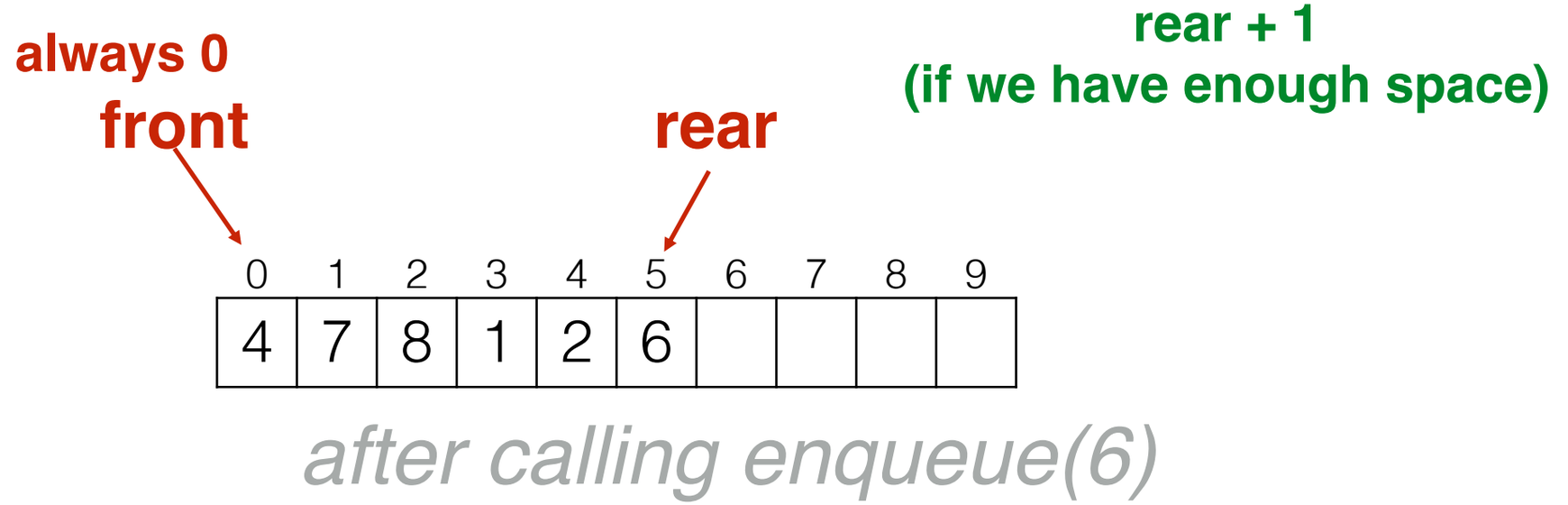


Array Implementation



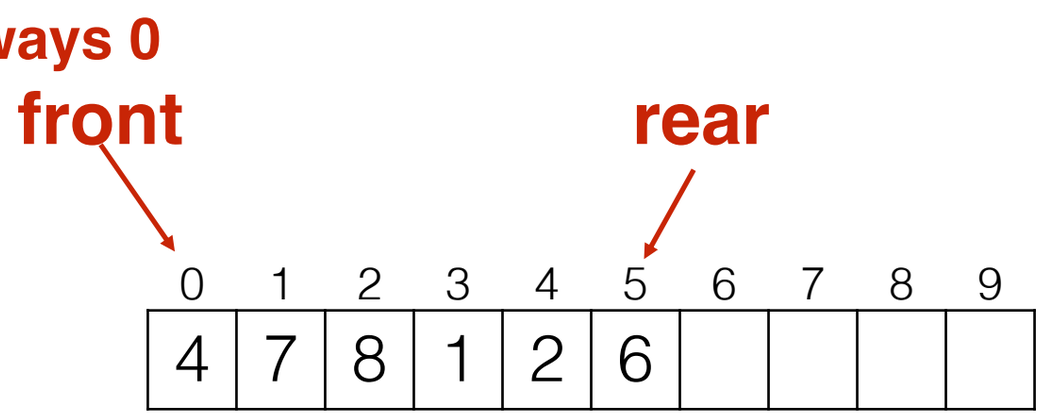
before calling enqueue(6)

Array Implementation



after calling enqueue(6)

Array Implementation

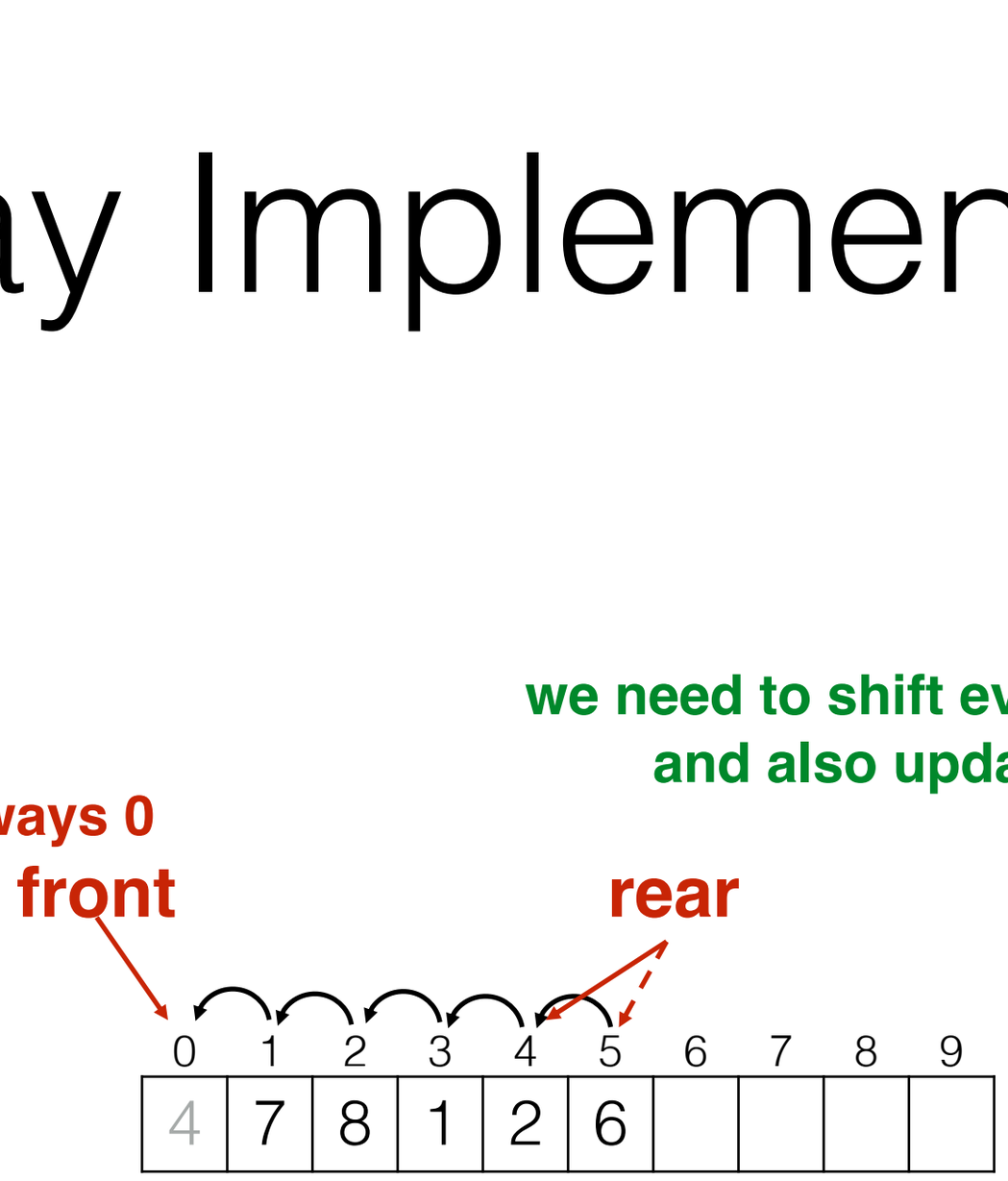


before calling dequeue()

Array Implementation

we need to shift everything to left
and also update the rear

always 0

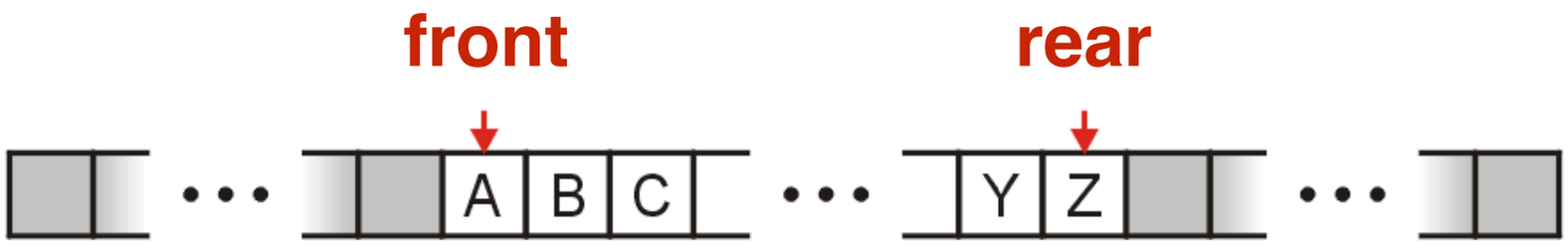


calling dequeue()

do not need to update front
we do not even need to keep
a variable for front, it is always 0
But we need to shift all items
each time we call dequeue.
so it is $\Theta(n)$

Array Implementation

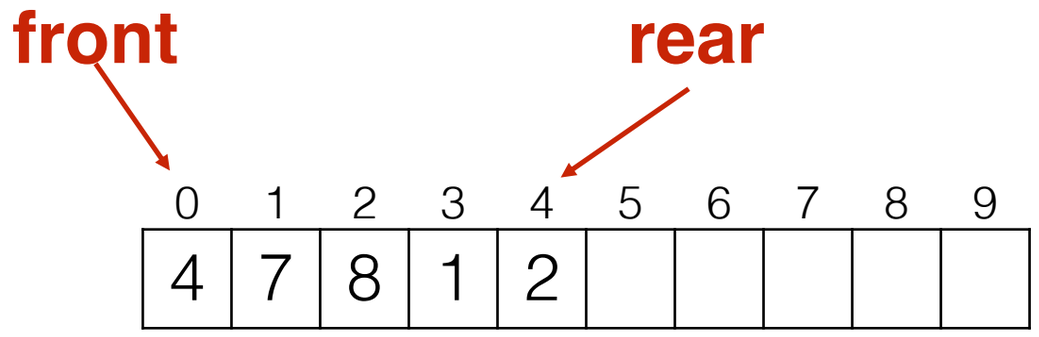
If we wave the constraint on the front, both enqueue and dequeue can be done in $\Theta(1)$ time.



	Front/1^{st}	Back/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Remove	$\Theta(1)$	$\Theta(1)$

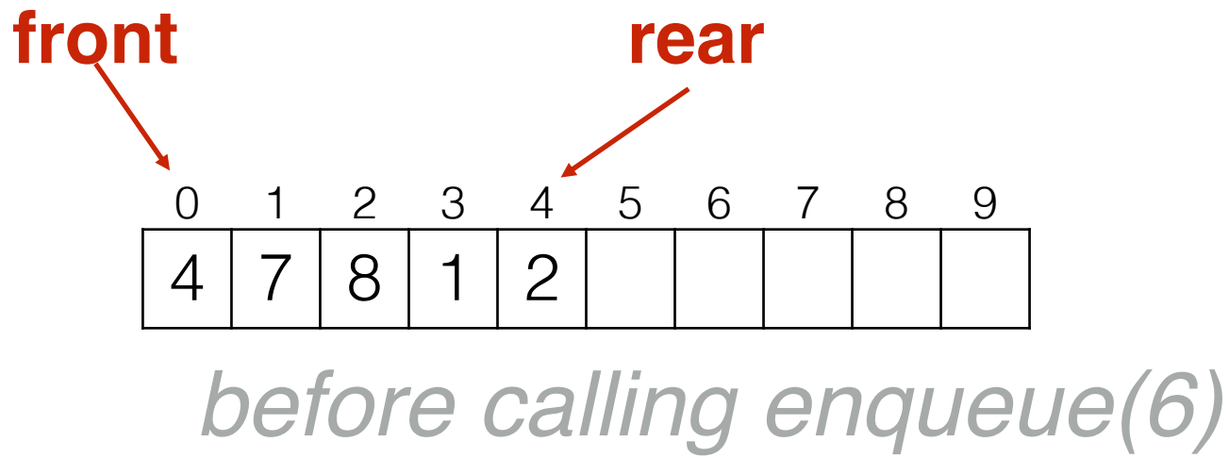
Array Implementation

Double ended array



Array Implementation

Double ended array

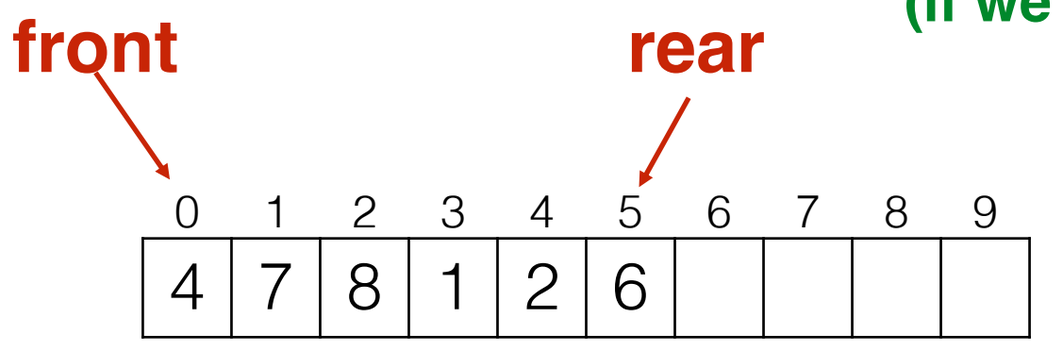


before calling enqueue(6)

Array Implementation

Double ended array

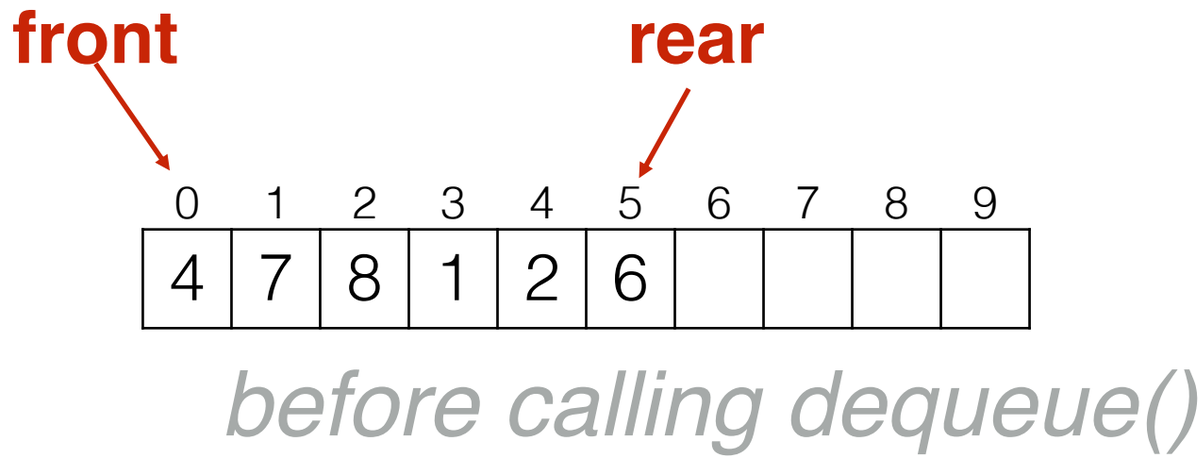
simply update the rear to rear + 1 (if we have enough space)



after calling enqueue(6)

Array Implementation

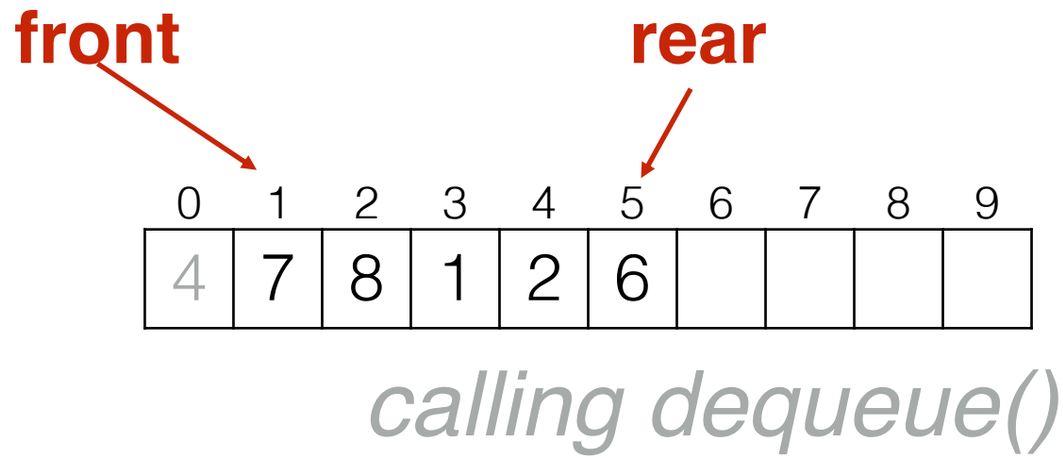
Double ended array



before calling dequeue()

Array Implementation

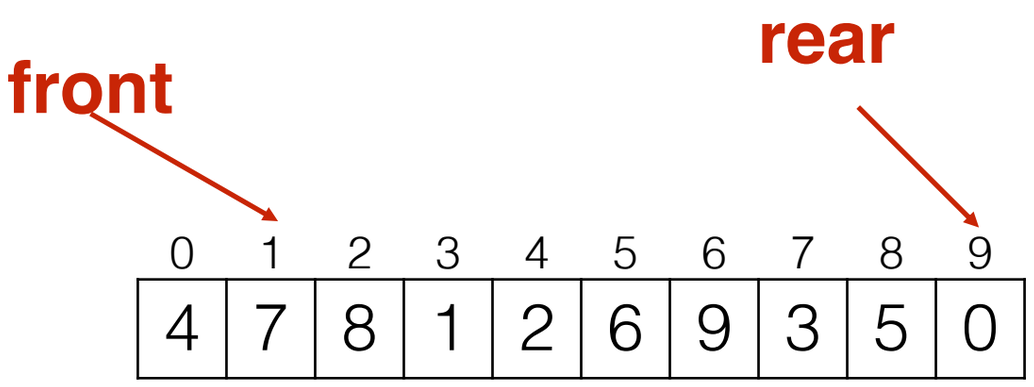
Double ended array



This time we need to update the front, but do not need to shift items. So calling dequeue is $\Theta(1)$.

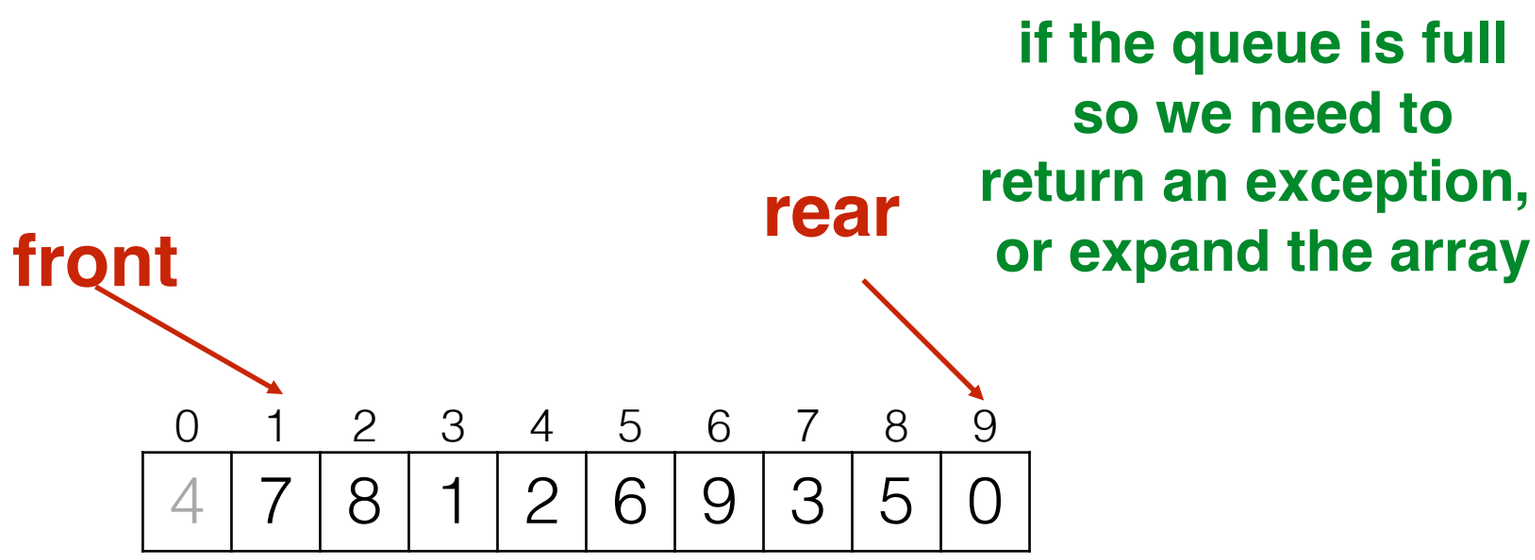
Array Implementation

What if the rear reaches to the end of array (last index)?



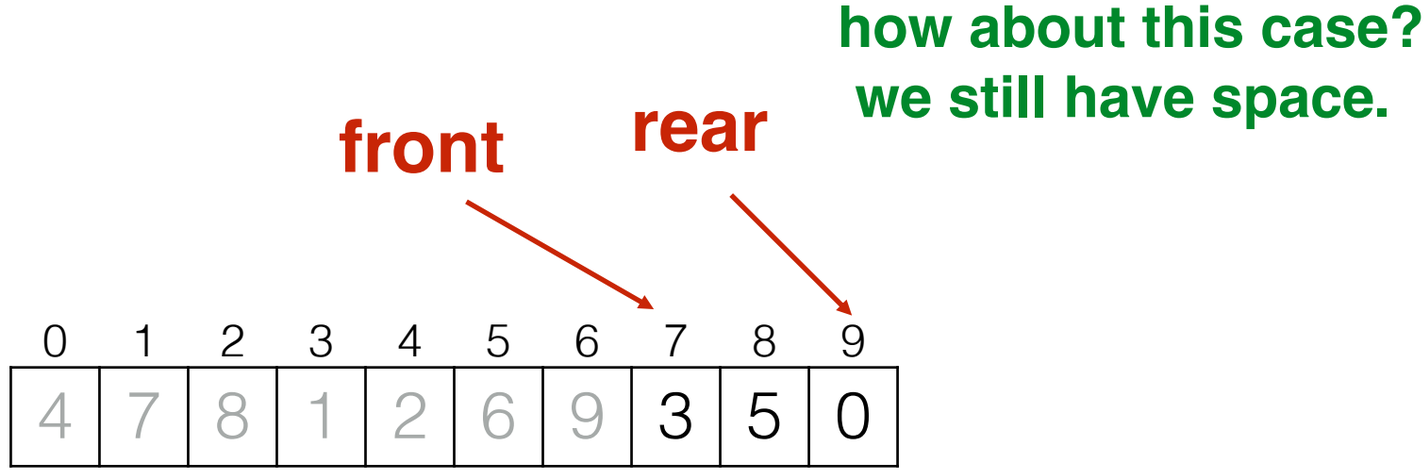
Array Implementation

What if the rear reaches to the end of array (last index)?



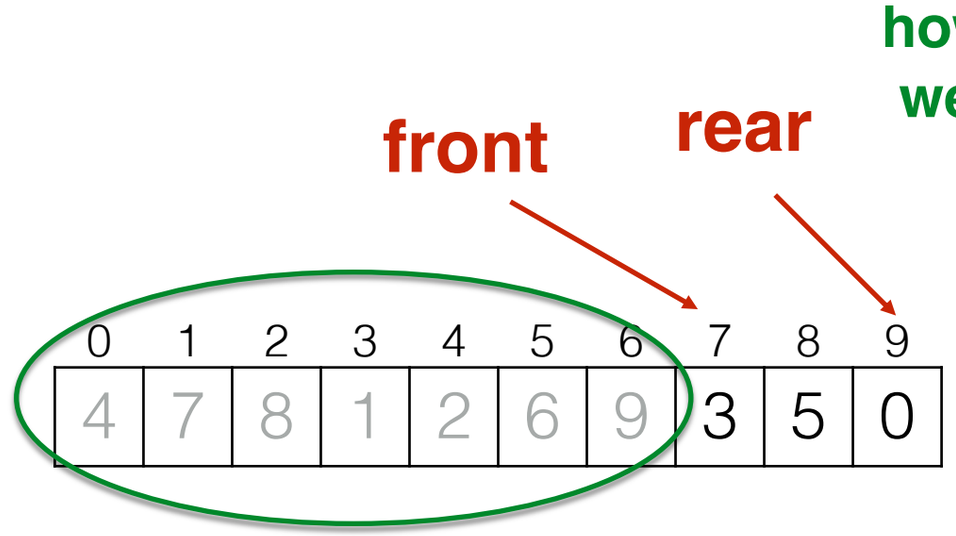
Array Implementation

What if the rear reaches to the end of array (last index)?



Array Implementation

What if the rear reaches to the end of array (last index)?

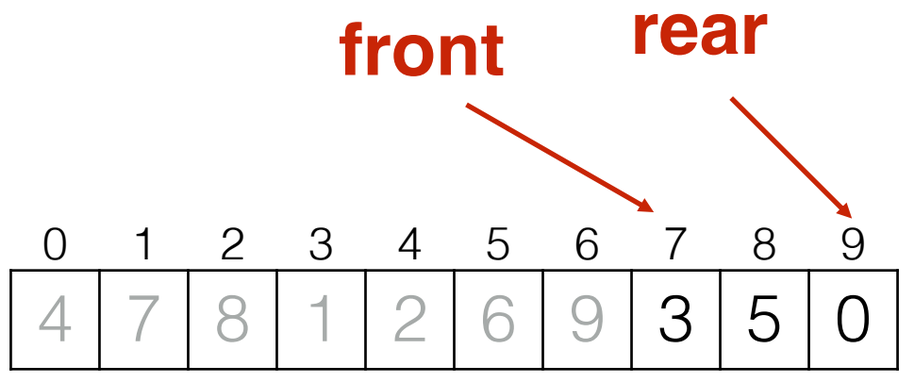


we still have space.
how can we use the
space before front?

Array Implementation

What if the rear reaches to the end of array (last index)?

let's use the available space

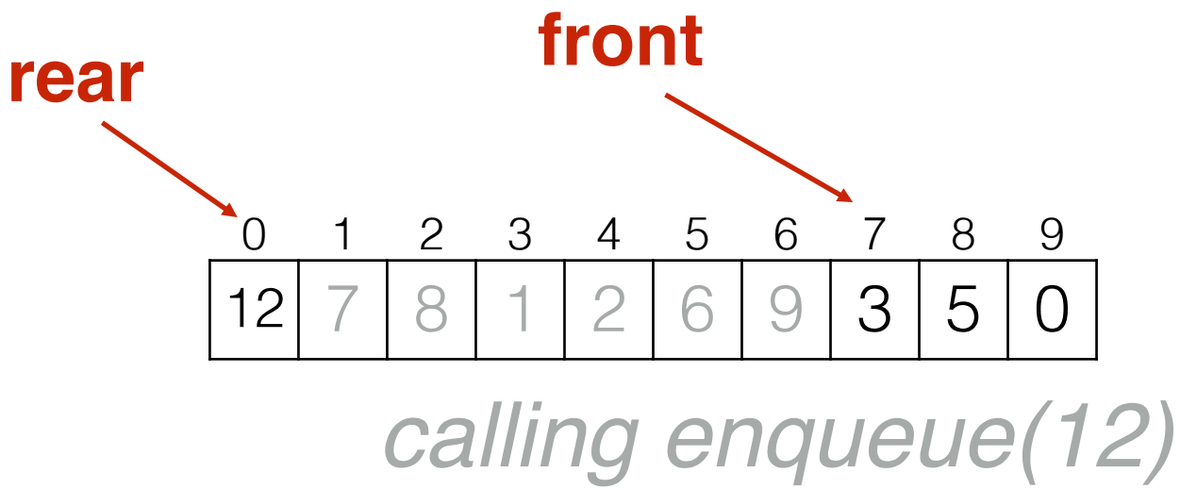


before calling enqueue(12)

Array Implementation

What if the rear reaches to the end of array (last index)?

let's use the available space

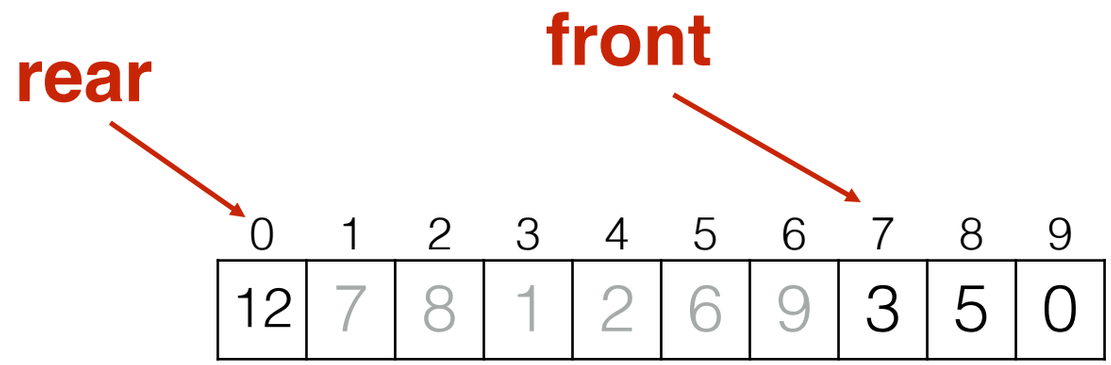


calling enqueue(12)

Array Implementation

What if the rear reaches to the end of array (last index)?

let's use the available space



the rear is reset to 0

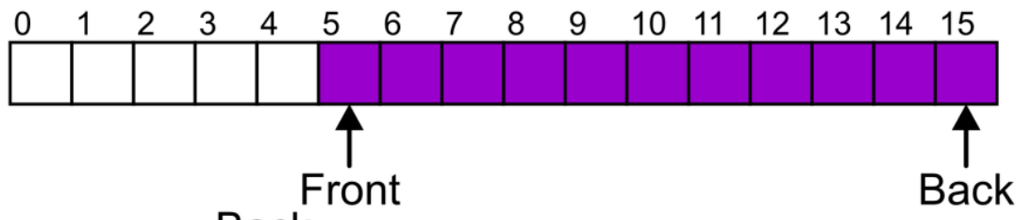
after calling enqueue(12)

Array Implementation

Assume the array is of size 16.

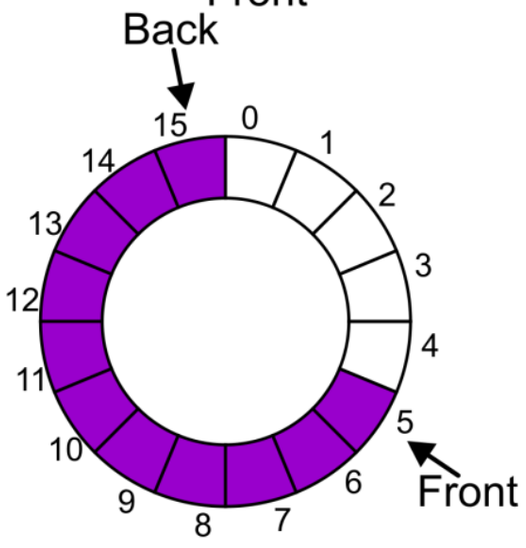
Instead of viewing the array on the range $0, \ldots, 15$, consider the indices being cyclic:

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



This is referred to as a *circular array*

The idea of a circular array is that the end of the array "wraps around" to the start of the array



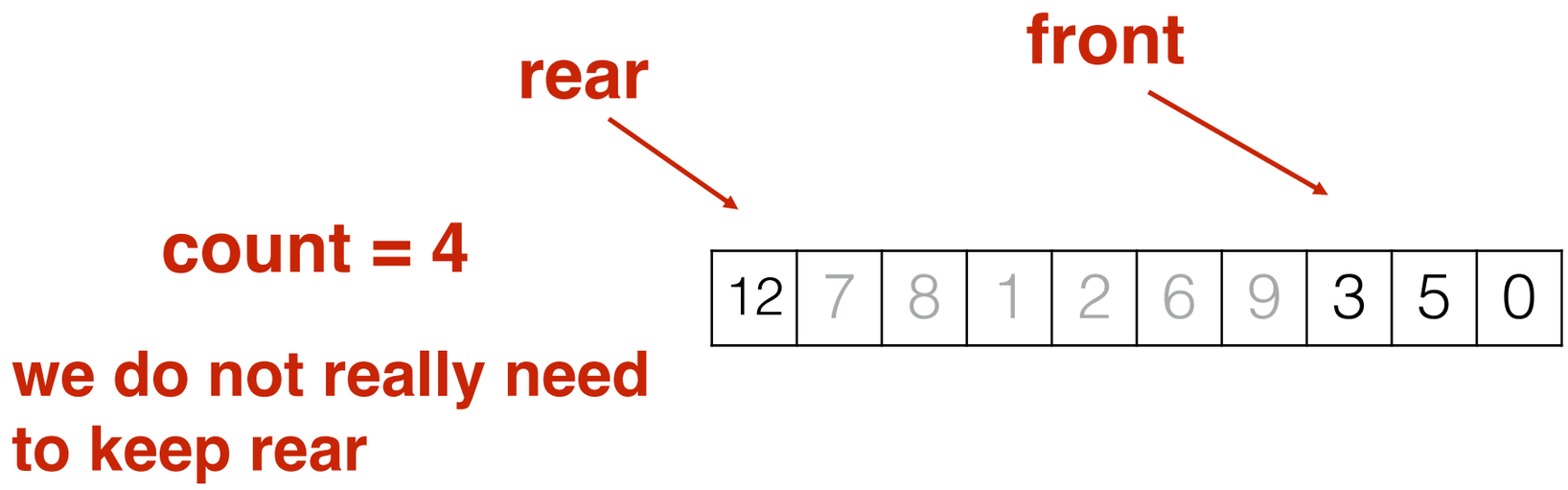
Array Implementation

- The mod operator (%) can be used to calculate reminders
  1%5 is 1 , 2%5 is 2 , 5%5 is 0 , 8%5 is 3
- mod can be used to calculate the front and back positions in a circular array, therefore avoiding comparisons to the array size…,

Array Implementation

- The mod operator (%) can be used to calculate reminders
  1%5 is 1 , 2%5 is 2 , 5%5 is 0 , 8%5 is 3

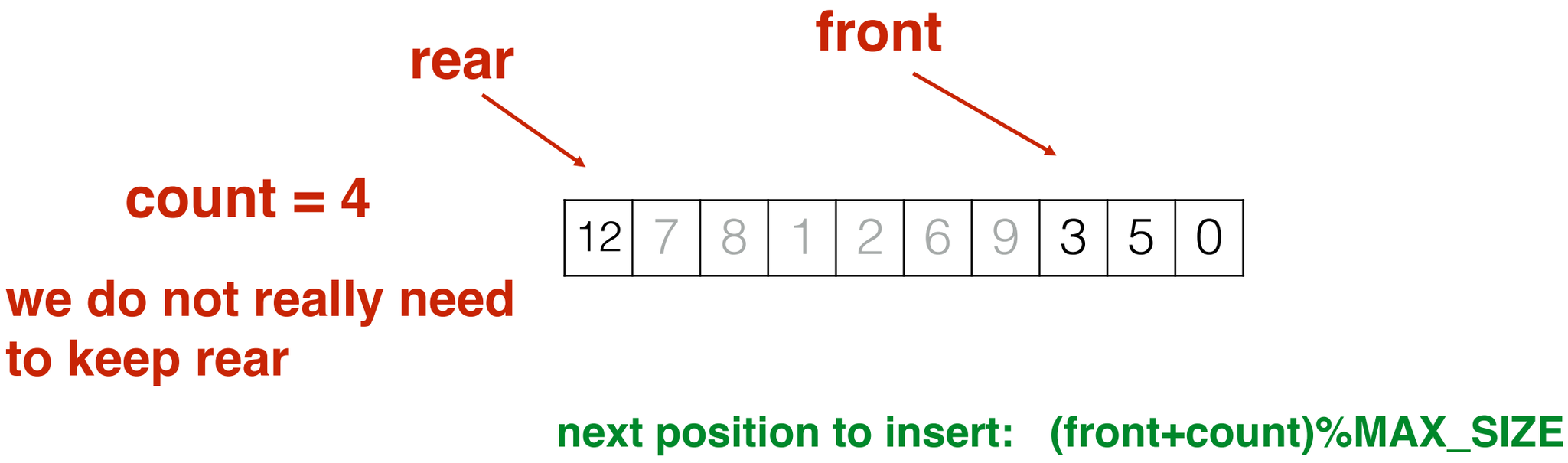
- mod can be used to calculate the front and back positions in a circular array, therefore avoiding comparisons to the array size…,



Array Implementation

- The mod operator (%) can be used to calculate reminders
  1%5 is 1 , 2%5 is 2 , 5%5 is 0 , 8%5 is 3

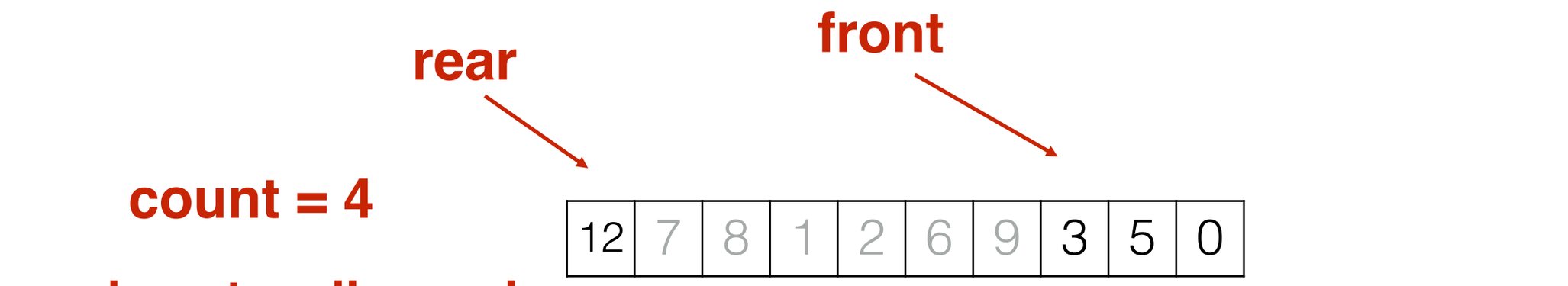
- mod can be used to calculate the front and back positions in a circular array, therefore avoiding comparisons to the array size…,



Array Implementation

- The mod operator (%) can be used to calculate reminders
  1%5 is 1 , 2%5 is 2 , 5%5 is 0 , 8%5 is 3

- mod can be used to calculate the front and back positions in a circular array, therefore avoiding comparisons to the array size…,

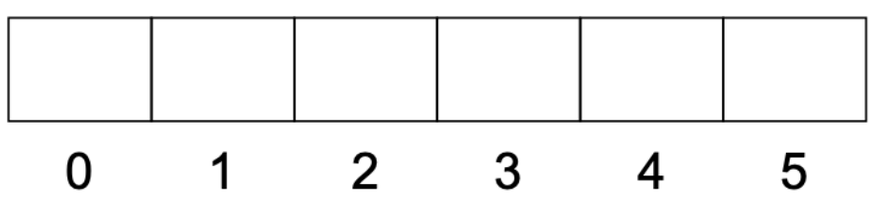


next position to insert: (front+count)%MAX_SIZE

update front after dequeue: (front+1)%MAX_SIZE

Array Implementation

front =	0
count =	0

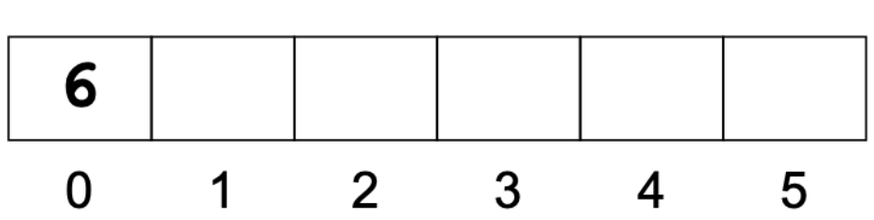



```
//Java Code
Queue q = new Queue();
q.enqueue(6);
```

insert new item at `(front+count)%MAX_SIZE`

Array Implementation

front =	0
count =	1




```
//Java Code
Queue q = new Queue();
q.enqueue(6);
```

insert new item at `(front+count)%MAX_SIZE`

Array Implementation

front =	0
count =	5

6	4	7	3	8	
0	1	2	3	4	5

```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
q.enqueue(3);
q.enqueue(8);
```

Array Implementation

front =	2
count =	4

6	4	7	3	8	9
0	1	2	3	4	5

make front = (0 + 1) % 6 = 1

make front = (1 + 1) % 6 = 2

```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
q.enqueue(3);
q.enqueue(8);
q.dequeue();//front = 1
q.dequeue();//front = 2
q.enqueue(9);
```

Array Implementation

front =	2
count =	5

5		7	3	8	9
0	1	2	3	4	5

insert at (front + count) % 6 = (2 + 4) % 6 = 0

```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
q.enqueue(3);
q.enqueue(8);
q.dequeue();//front = 1
q.dequeue();//front = 2
q.enqueue(9);
q.enqueue(5);
```

Array Implementation

- The mod operator (%) can be used to calculate reminders
  1%5 is 1 , 2%5 is 2 , 5%5 is 0 , 8%5 is 3

- mod can be used to calculate the front and back positions in a circular array, therefore avoiding comparisons to the array size…,

- The back of the queue can be found by:
  - *(front + count) % MAX_SIZE*
  - where count is the number of items in the queue

- After removing an item the front of the queue should be updated as:
  - *(front + 1) % MAX_SIZE*

Exceptions

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

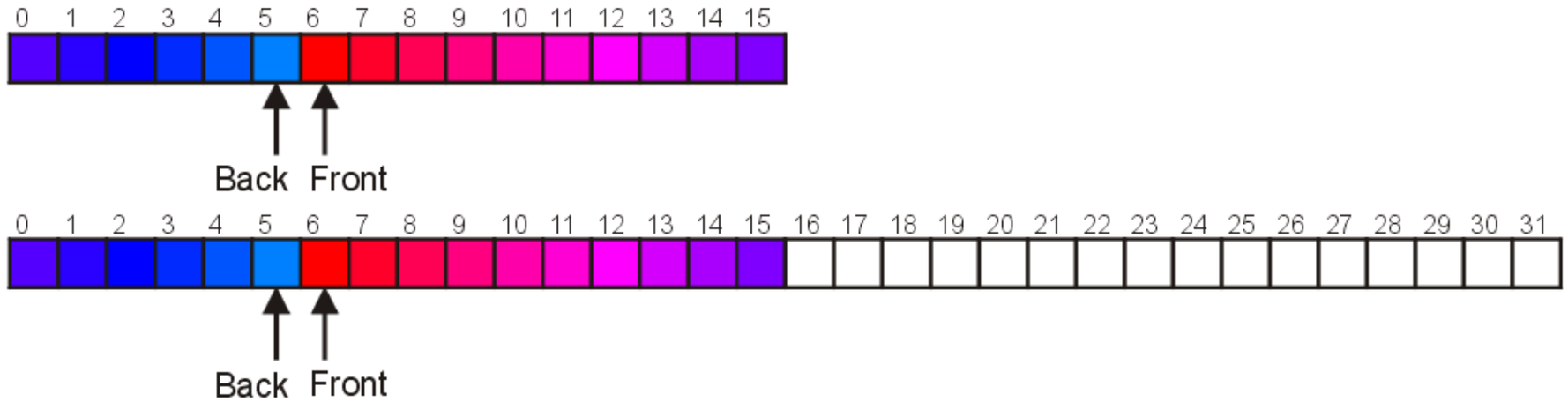
If the array is filled, we can:

- Increase the size of the array
- Throw an exception

Increasing Capacity

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

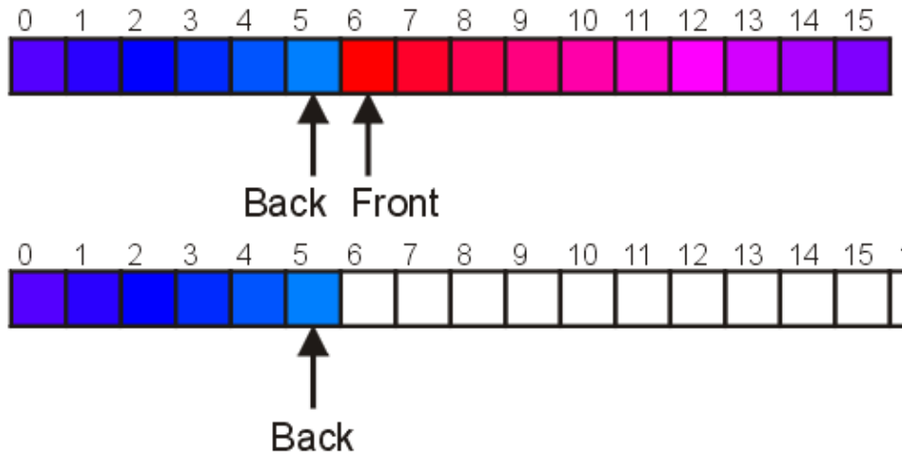
– A direct copy does not work:



Increasing Capacity

There are two solutions:

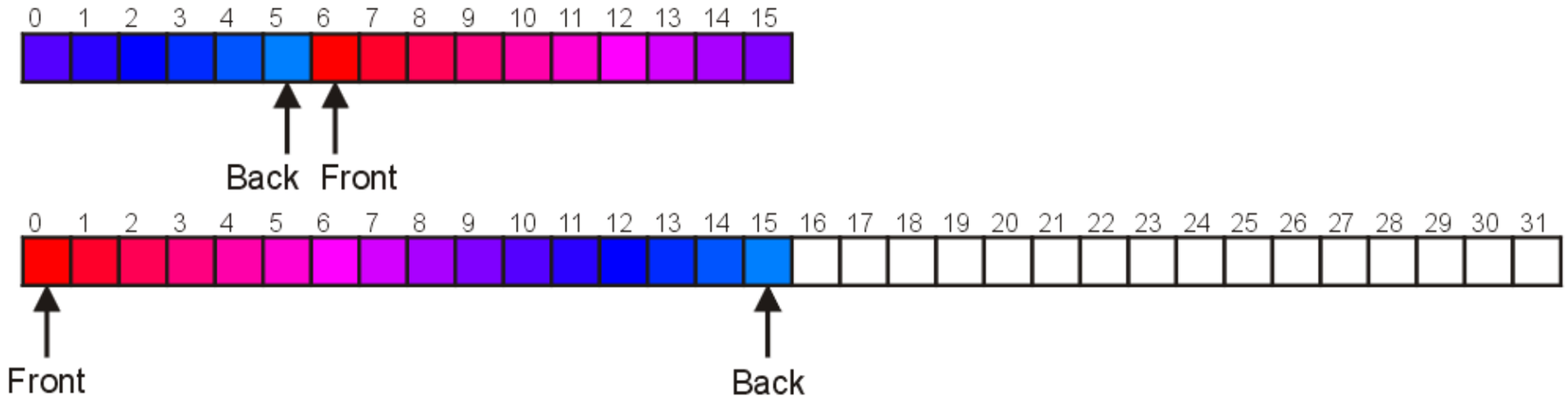
- Move those beyond the front to the end of the array
- The next push would then occur in position 6



Increasing Capacity

An alternate solution is normalization:

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

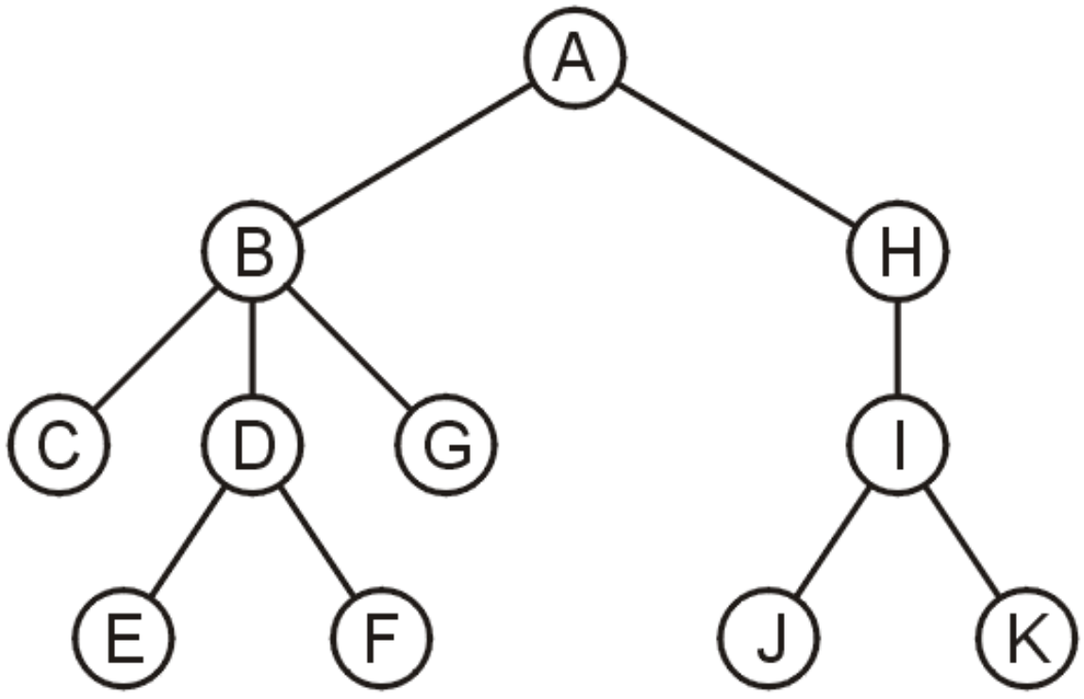


Applications

Application

Performing a breadth-first traversal of a tree

– Consider searching the directory structure (which is an example of a tree structure)

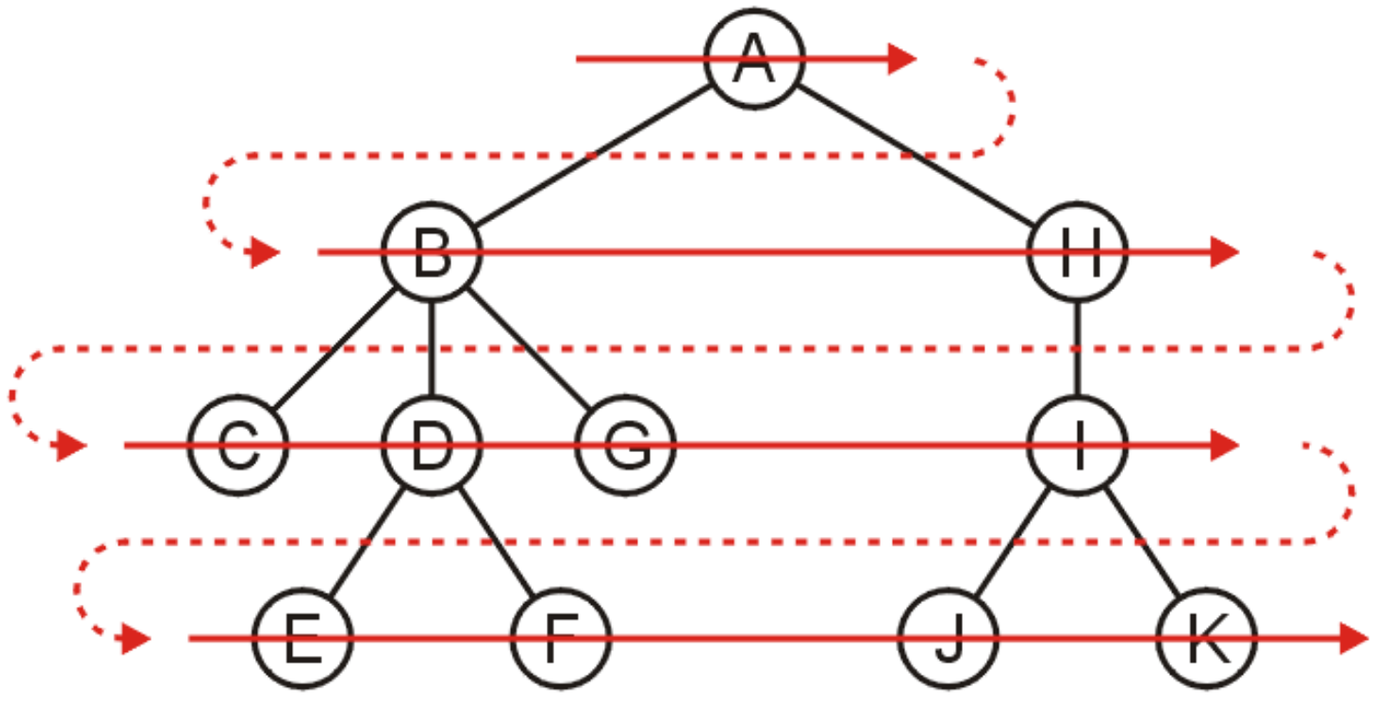


Application

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



Application

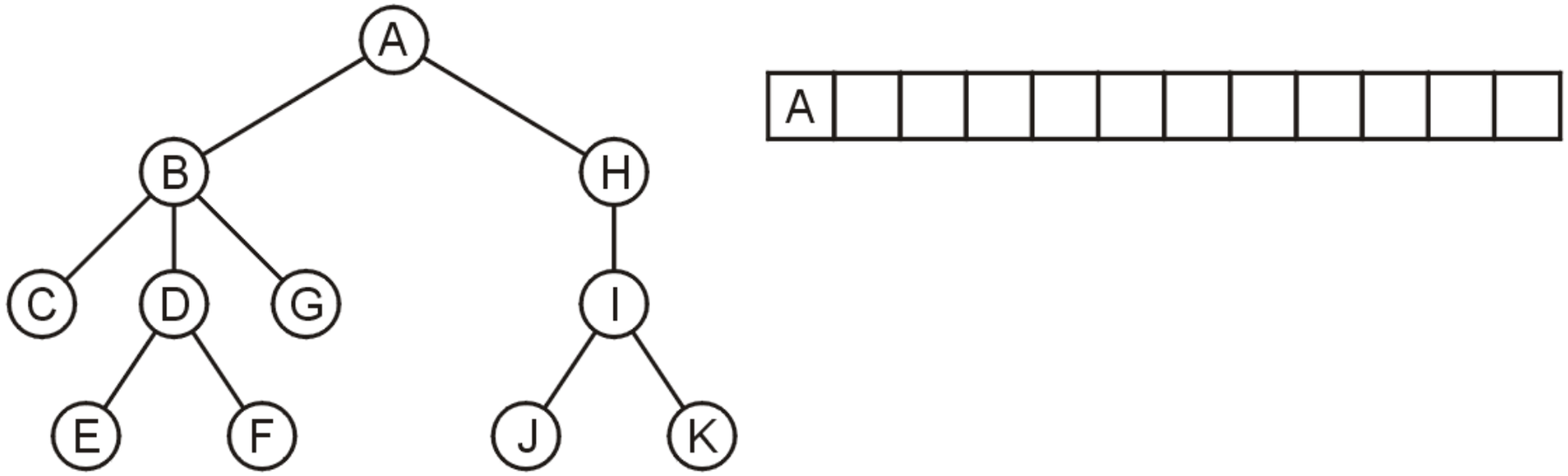
The easiest implementation is:

- Place the root directory into a queue
- While the queue is not empty:
  - dequeue the directory at the front of the queue
  - enqueue 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

Application

Push the root directory A

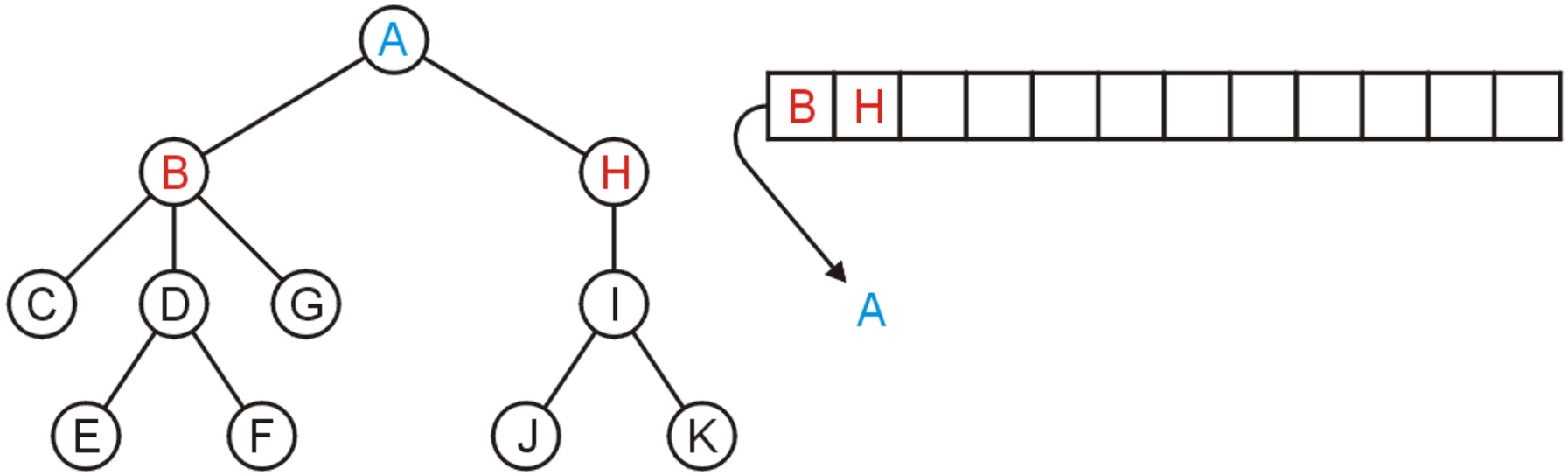


Steps:

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

Application

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

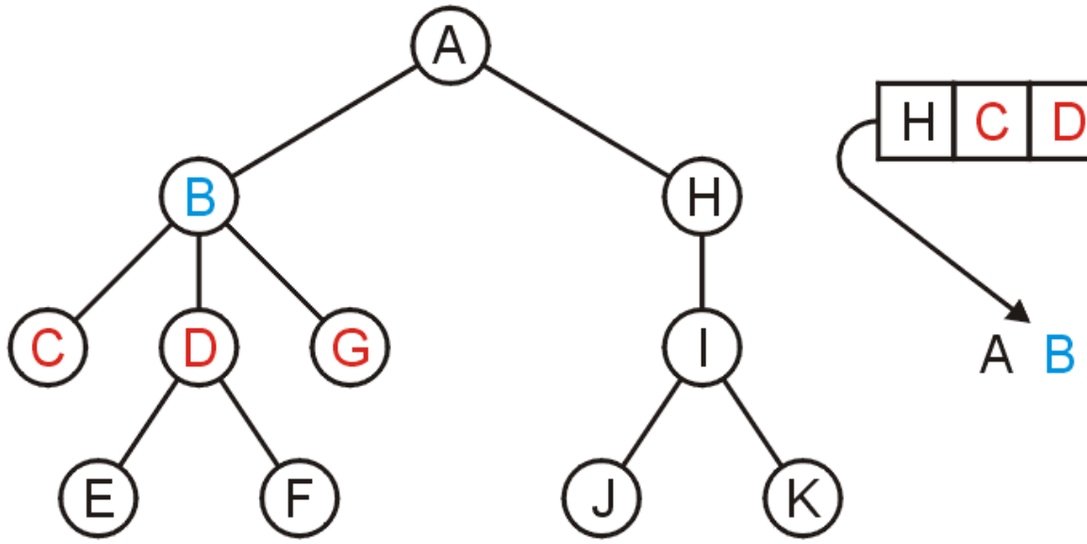


Steps:

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

Application

Pop B and then push C, D, and G

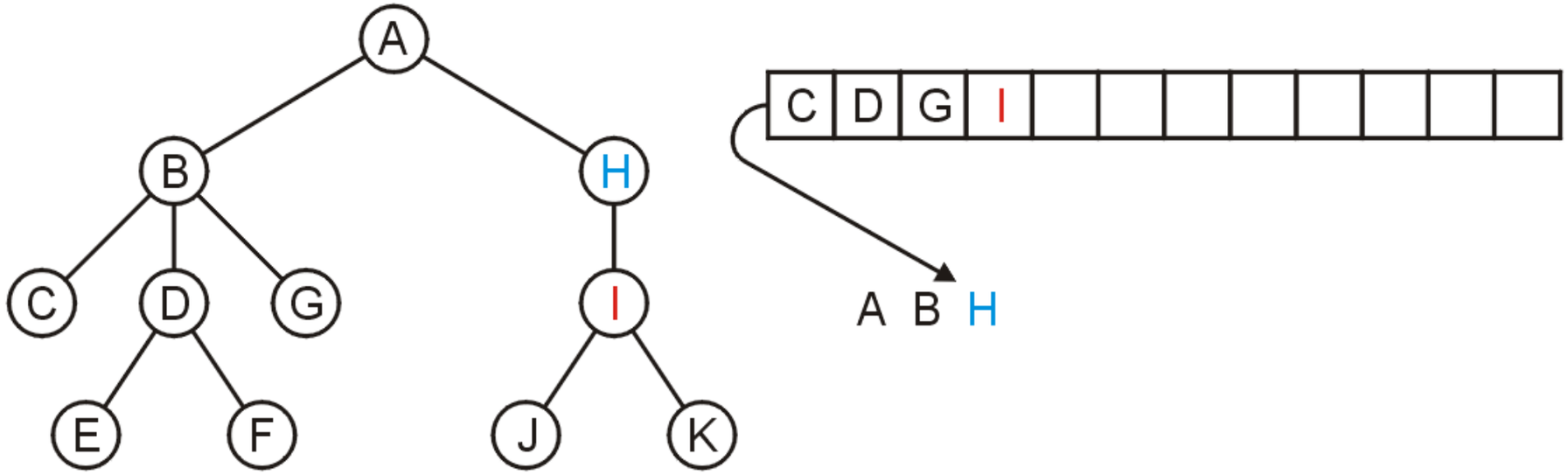


Steps:

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

Application

Pop H and push its one sub-directory I

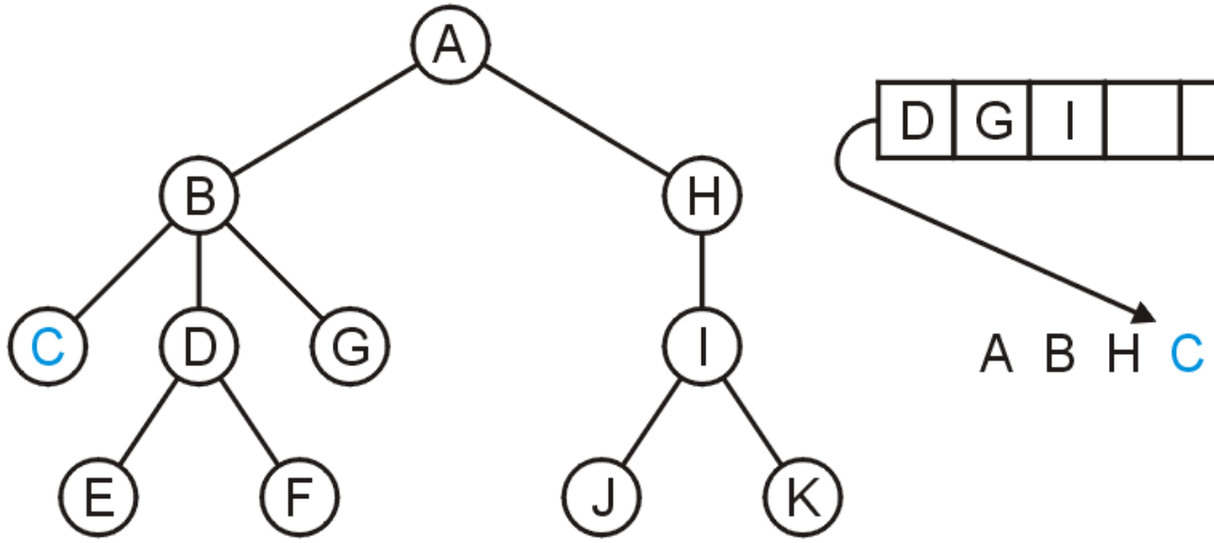


Steps:

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

Application

Pop C: no sub-directories

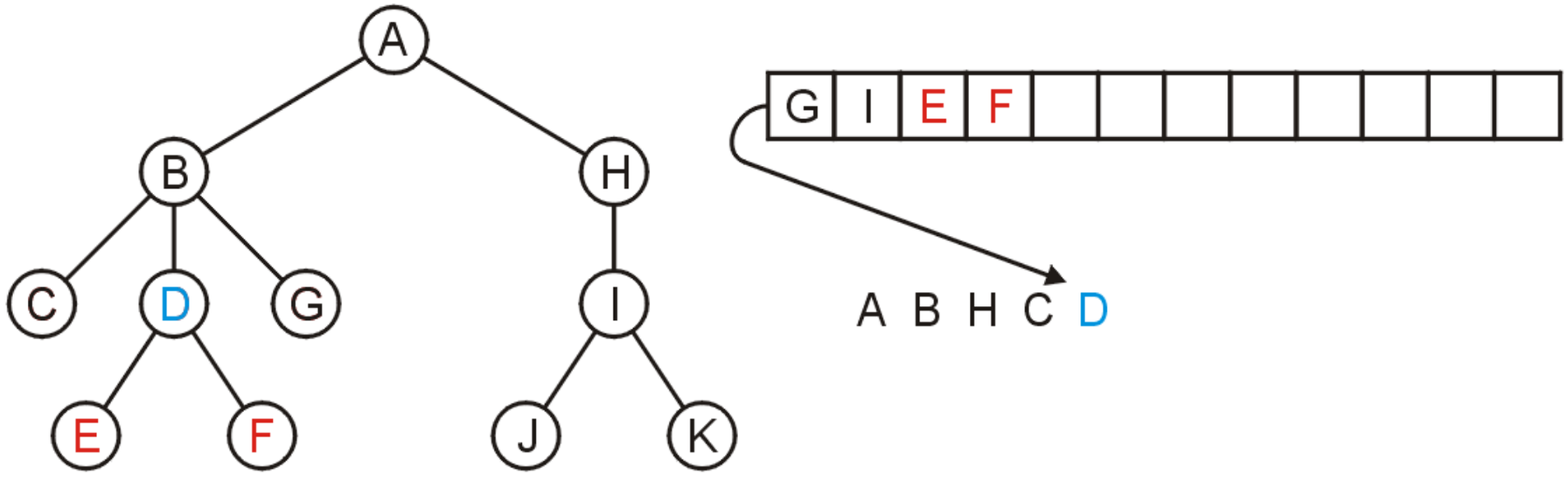


Steps:

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

Application

Pop D and push E and F

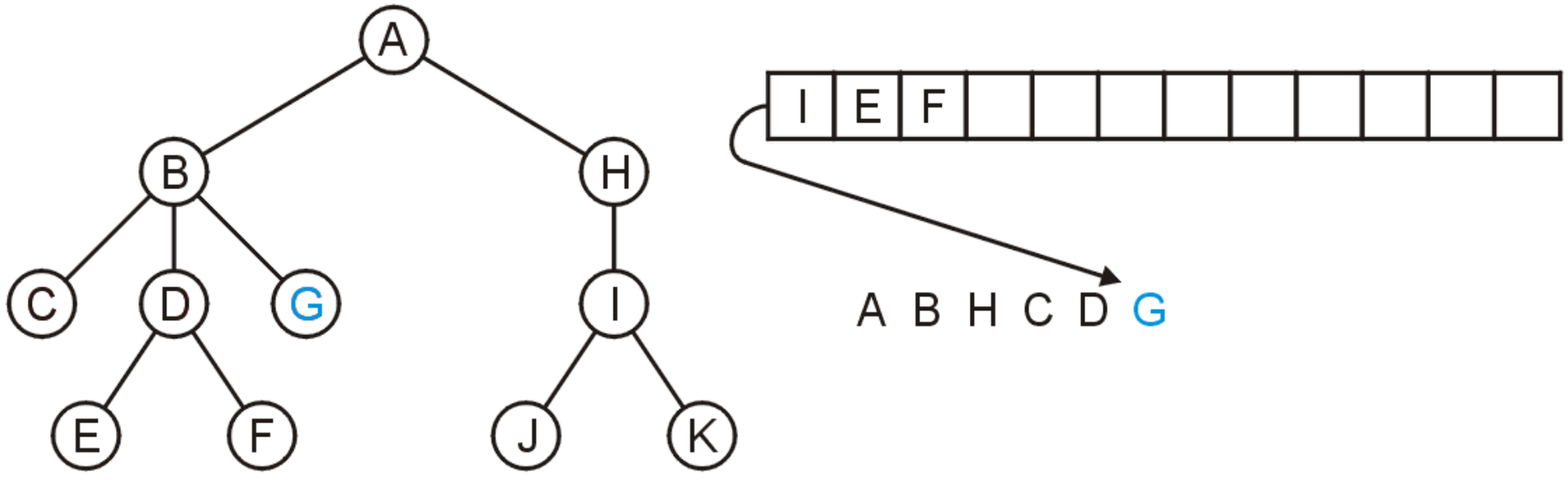


Steps:

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

Application

Pop G

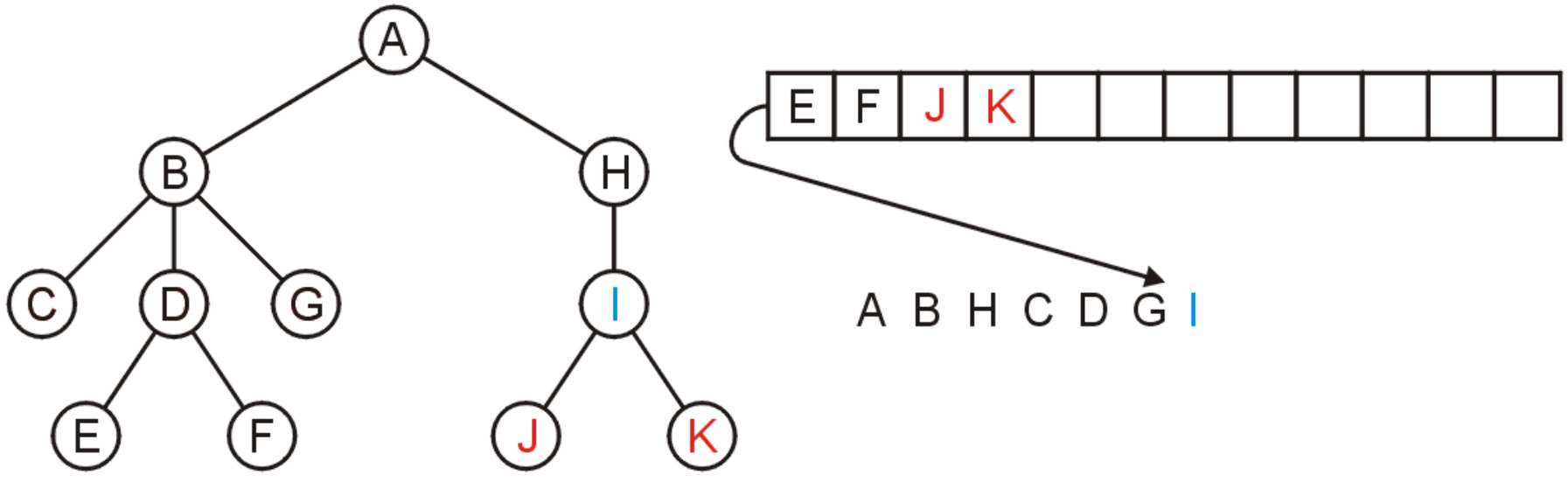


Steps:

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

Application

Pop I and push J and K

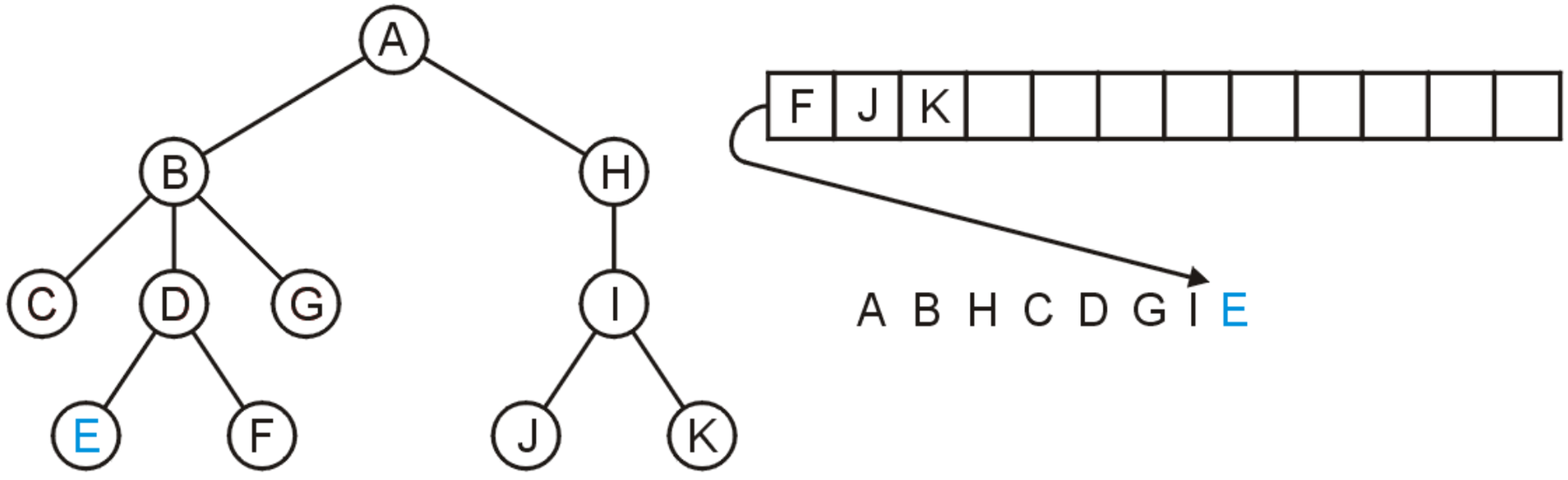


Steps:

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

Application

Pop E

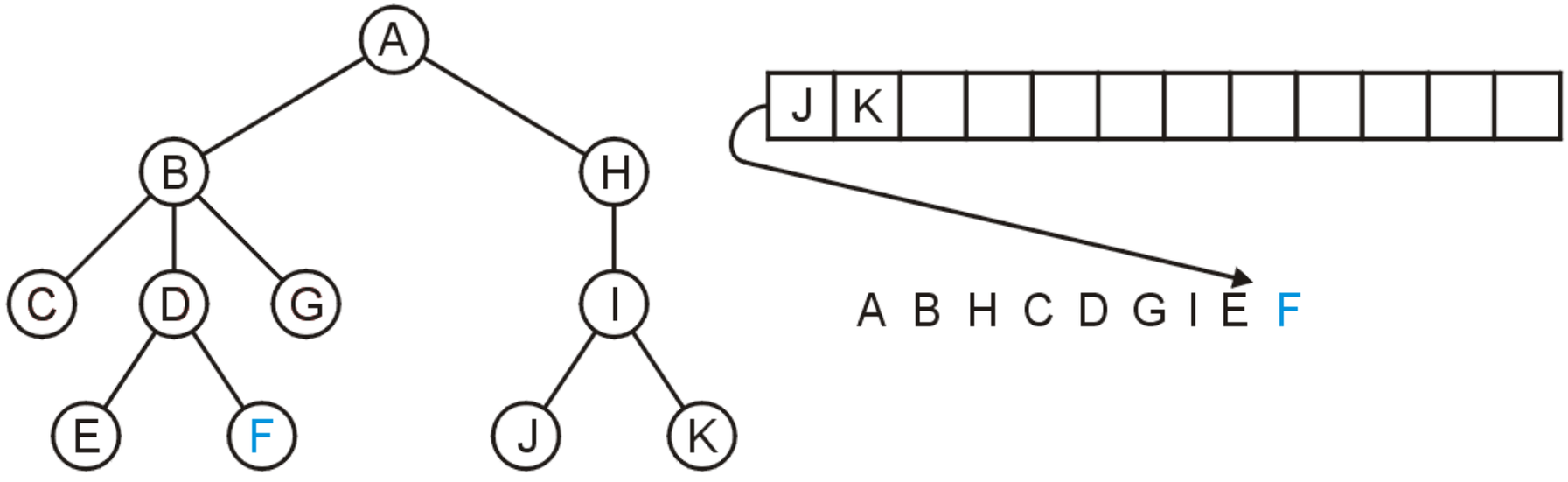


Steps:

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

Application

Pop F

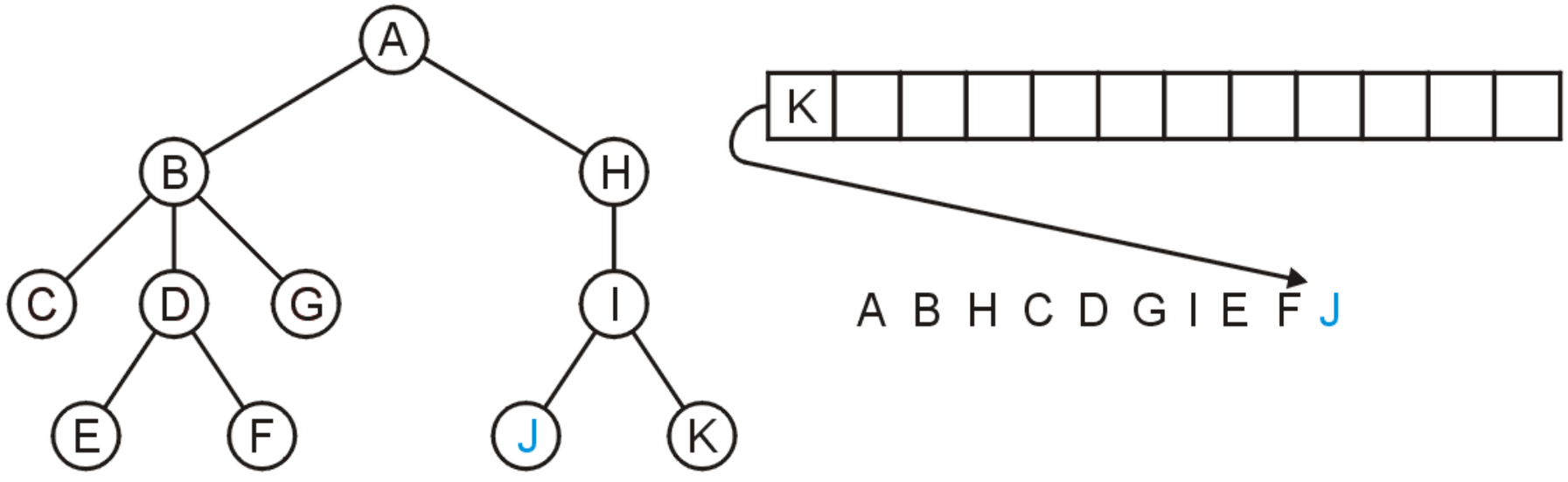


Steps:

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

Application

Pop J

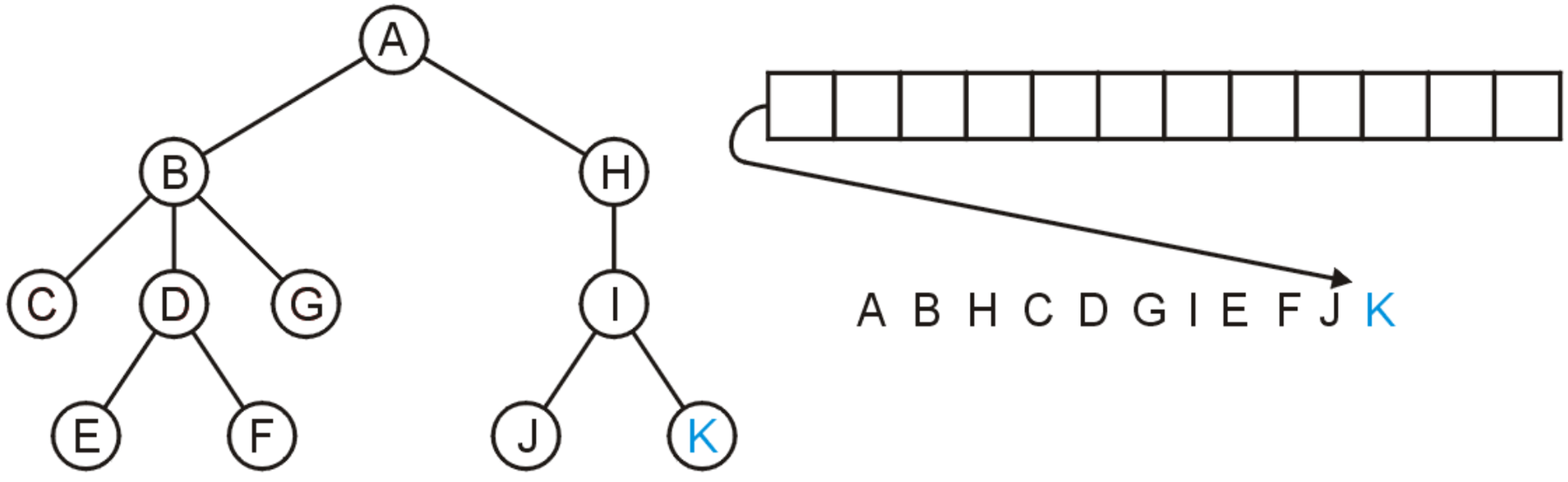


Steps:

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

Application

Pop K and the queue is empty



Steps:

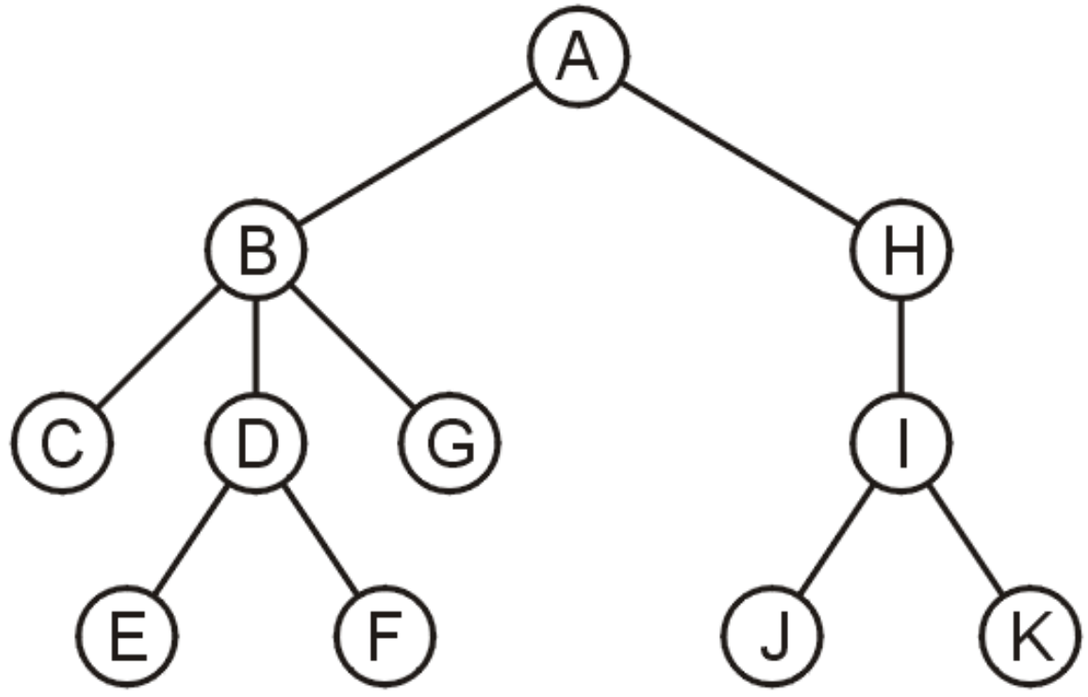
- Place the root directory into a queue
- While the queue is not empty:
  - dequeue the directory at the front of the queue
  - enqueue all of its sub-directories into the queue

Application

The resulting order

A B H C D G I E F J K

is in breadth-first order:



Steps:

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

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