COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

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

Queue

Outline

This topic discusses the concept of a queue:

- Description of an Abstract Queue
- List applications
- Implementation

Abstract Queue

An Abstract Queue (Queue ADT) is an abstract data type that emphasizes specific operations:

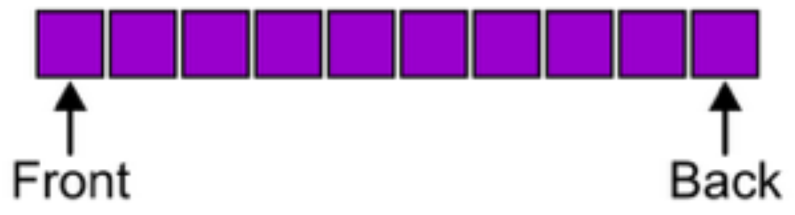
- Uses an explicit linear ordering
- Insertions and removals are performed individually
- There are no restrictions on objects inserted into (***pushed onto***) the queue—that object is inserted at the end of the queue
- The object designated as the ***front*** of the queue is the object which was in the queue the longest
- The remove operation (***popping*** from the queue) removes the current ***front*** of the queue

Abstract Queue

Also called a ***first-in–first-out*** (FIFO) data structure

– Graphically, we may view these operations as follows:

General view of a queue:

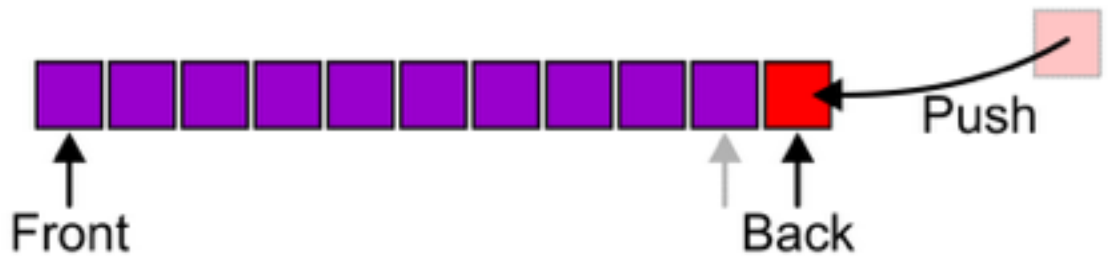


Abstract Queue

Also called a ***first-in–first-out*** (FIFO) data structure

– Graphically, we may view these operations as follows:

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

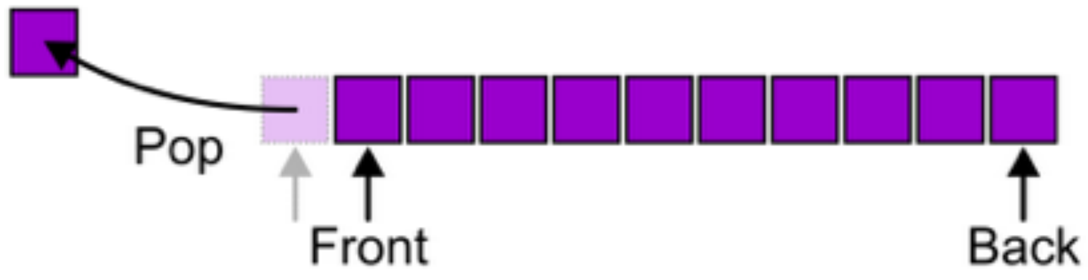


Abstract Queue

Also called a ***first-in–first-out*** (FIFO) data structure

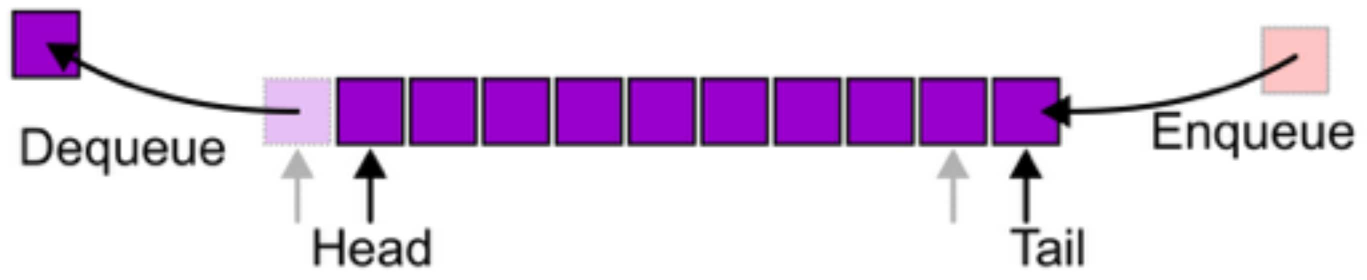
– Graphically, we may view these operations as follows:

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



Abstract Queue

Alternative terms may be used for the four operations on a queue, including:



- `Enqueue` (instead of `push`)
- `dequeue` (instead of `pop`)
- `rear/back` and `front` (returning the item at `head` and `tail`)

Abstract Queue

There are two exceptions associated with this abstract data structure:

- It is an undefined operation to call either `dequeue` or `front` on an empty `queue`

Abstract Queue

The main difference between a stack and a queue:

- A stack is only accessed from 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

Grocery stores, banks, and airport security use queues

Most shared computer services are servers:

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

Implementations

We will look at two implementations of queues:

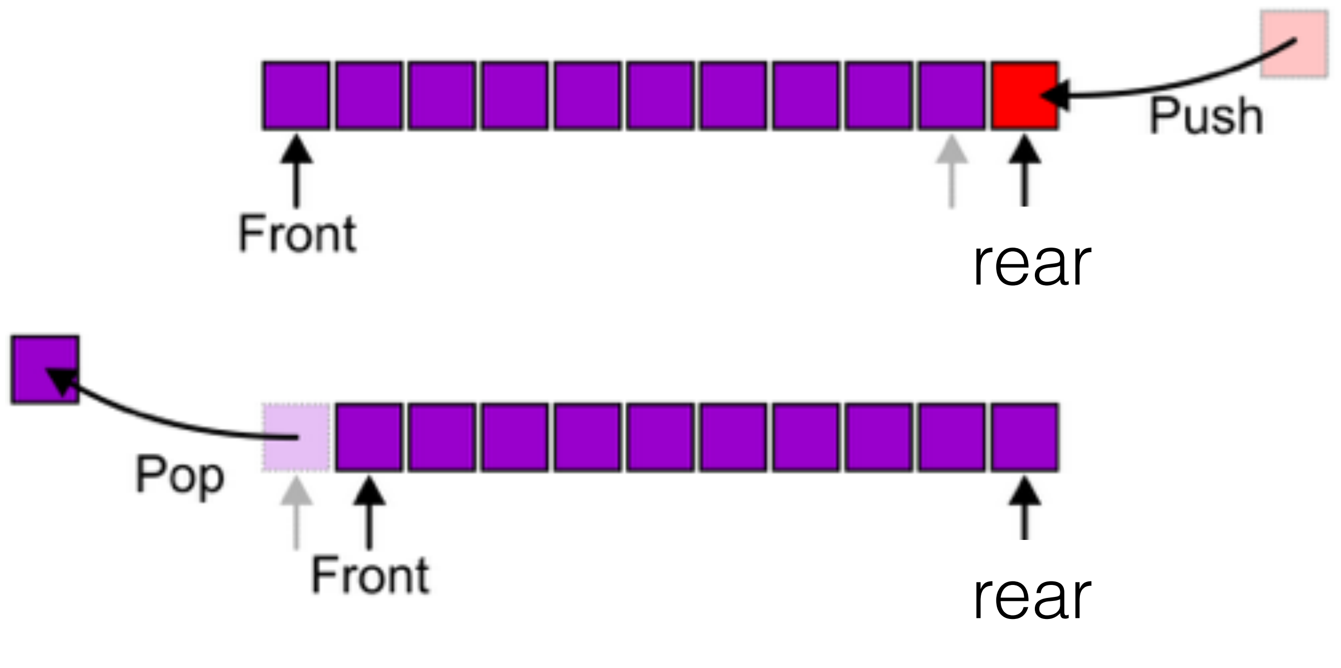
- Singly linked lists
- Circular arrays

Requirements:

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

Linked-List Implementation

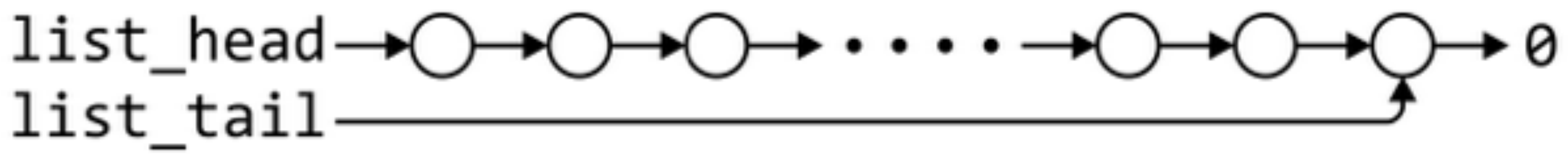
- The first decision in planning the linked-list implementation of the Queue class:
  - Which end of the list will correspond to the front of the queue.
- Recall that items need to be added (enqueued) to the rear of the queue, and removed (dequeued) from the front of the queue.
- Make our choice based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



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/1st	End/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

`SList` Definition

The definition of singly linked list class (`SList`) from lab2 is:

```
public class SList {
   private SListNode head;
   private SListNode tail;
   private int size;
  //methods:
   public boolean isEmpty();
   public int length();
   public void insertFront(Object obj);
   public void insertEnd(Object obj);
   public Object nth(int position);
   public Object removeEnd();
   public Object removeFront();
}
```

Queue-as-List Class

The queue class using a singly linked list has a single private member variable: a singly linked list

```
public LinkListQueue() {
    list = new SList();
}
```

Queue-as-List Class

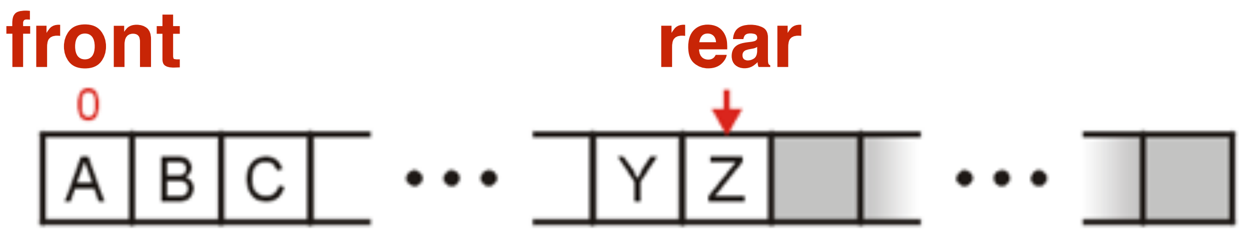
The implementation is similar to that of a Stack-as-List

```
 public void enqueue(Object obj) {
    list.insertEnd(obj);
 }

public Object dequeue() throws NoSuchElementException {
   if (list.isEmpty()) throw new NoSuchElementException("Queue underflow");
   return list.removeFront();
}

 public Object front() throws NoSuchElementException {
    if (list.isEmpty()) throw new NoSuchElementException("Queue underflow");
    return list.nth(1);   //constant time because it is always the first item
}
```

Array Implementation

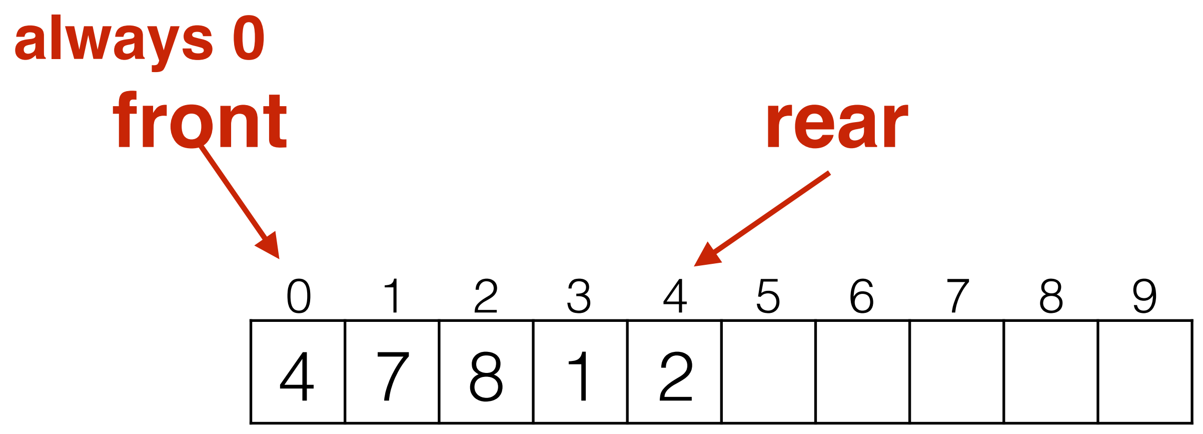
A one-ended array does not allow all operations to occur in $\Theta(1)$ time

- The front should always be index 0

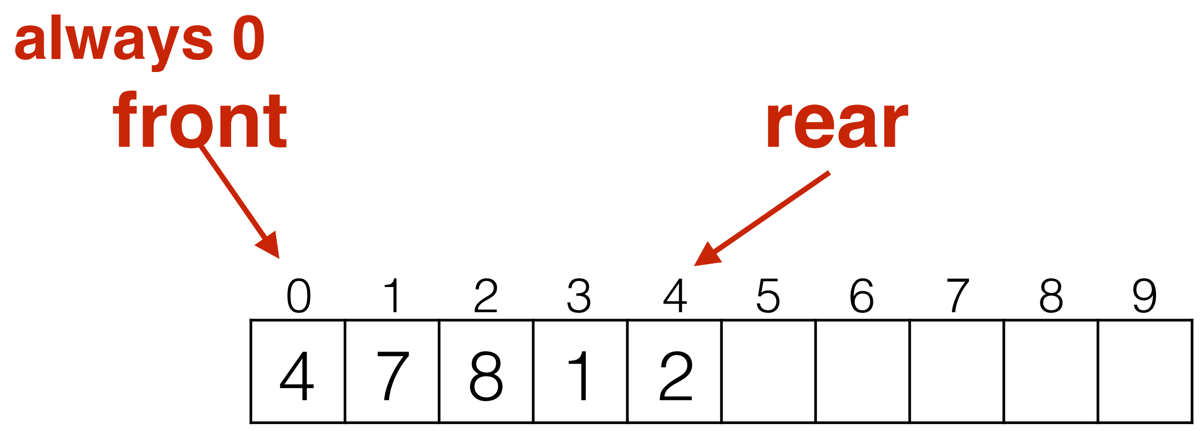


	Front/1st	End/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(n)$	$\Theta(1)$
Erase	$\Theta(n)$	$\Theta(1)$

Array Implementation

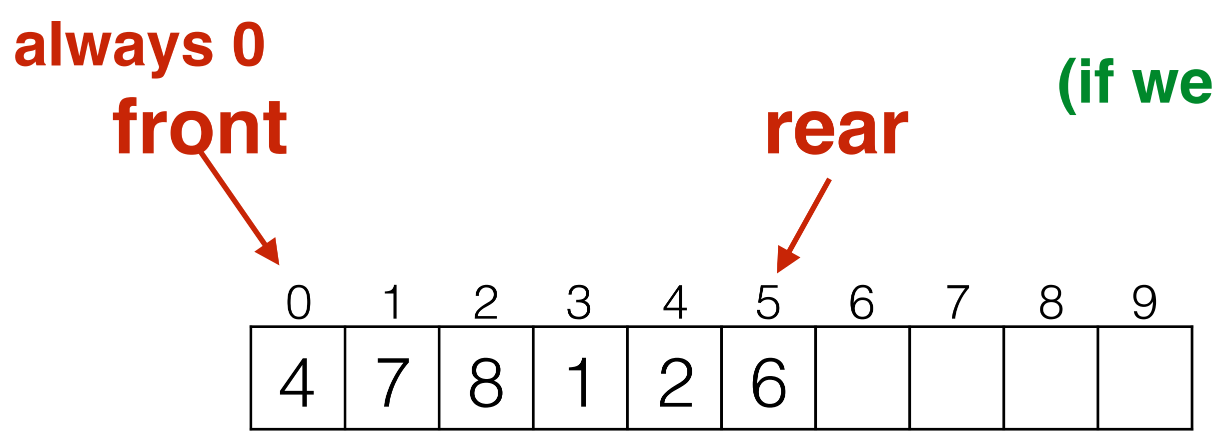


Array Implementation



before calling enqueue(6)

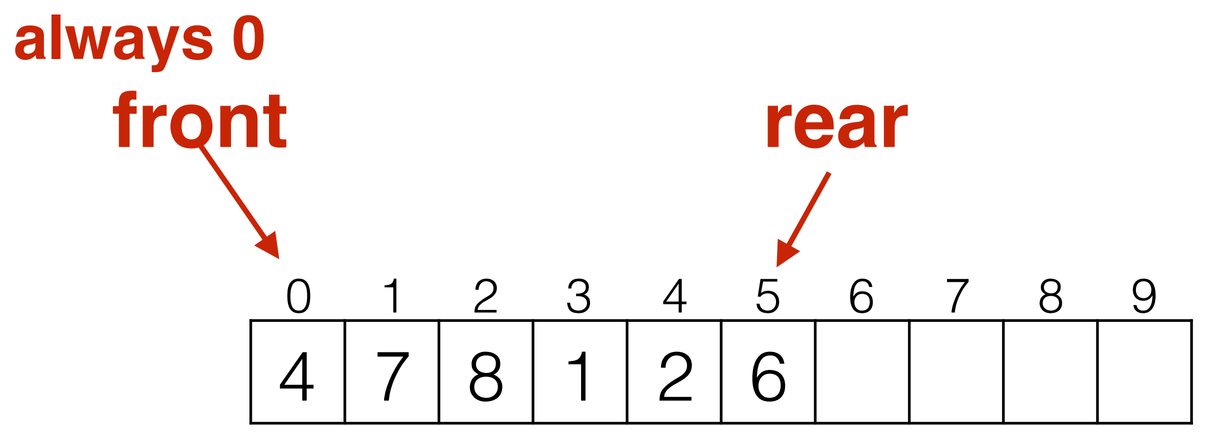
Array Implementation



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

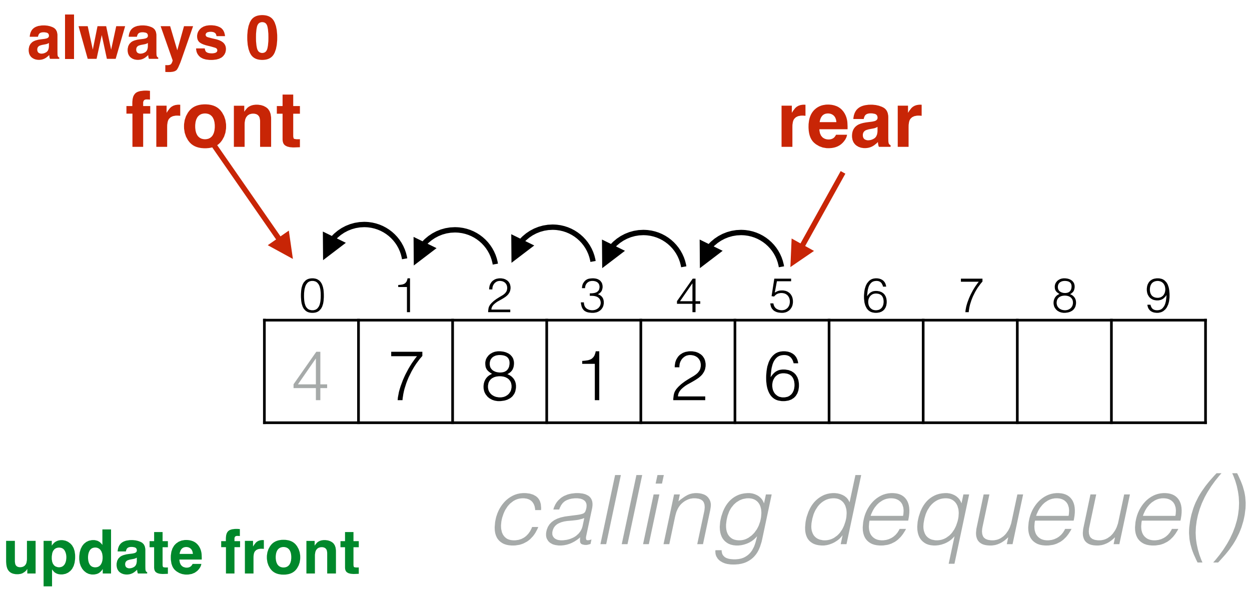
after calling enqueue(6)

Array Implementation



before calling dequeue()

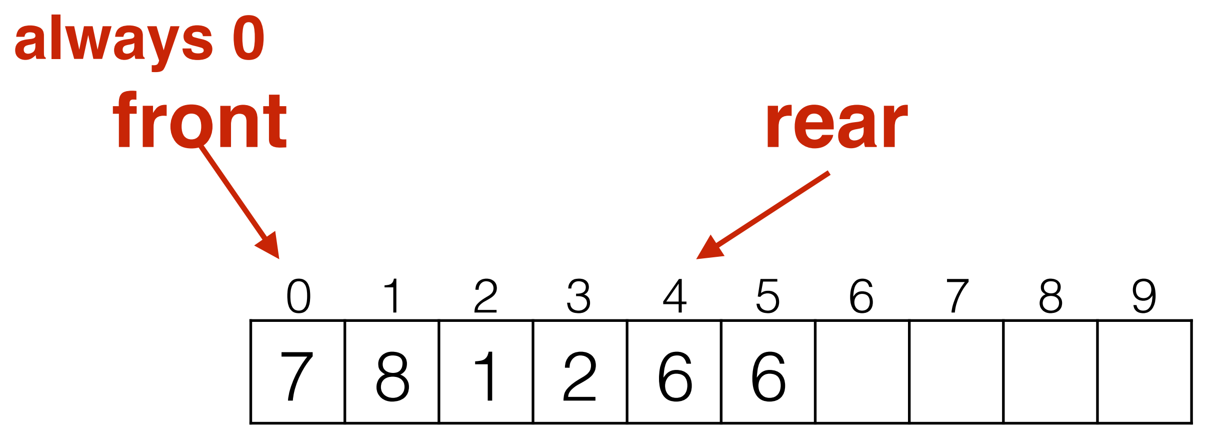
Array Implementation



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 copy every items
each time we call dequeue.
so it is $\Theta(n)$

Array Implementation

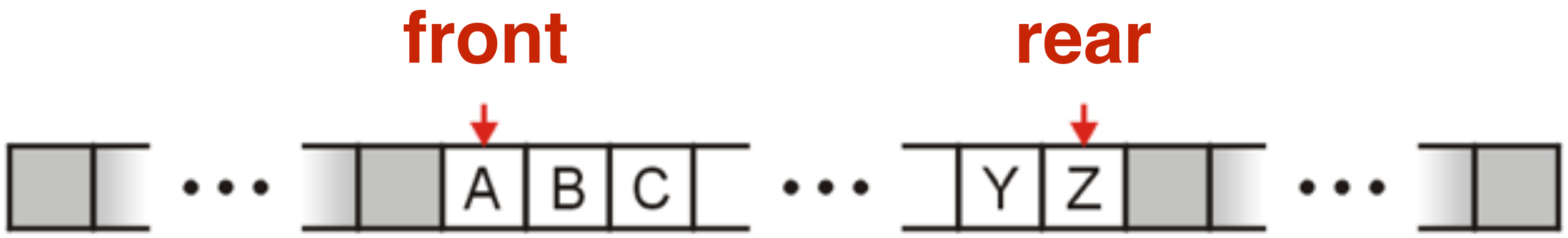
we also need to update the rear



after calling dequeue()

Array Implementation

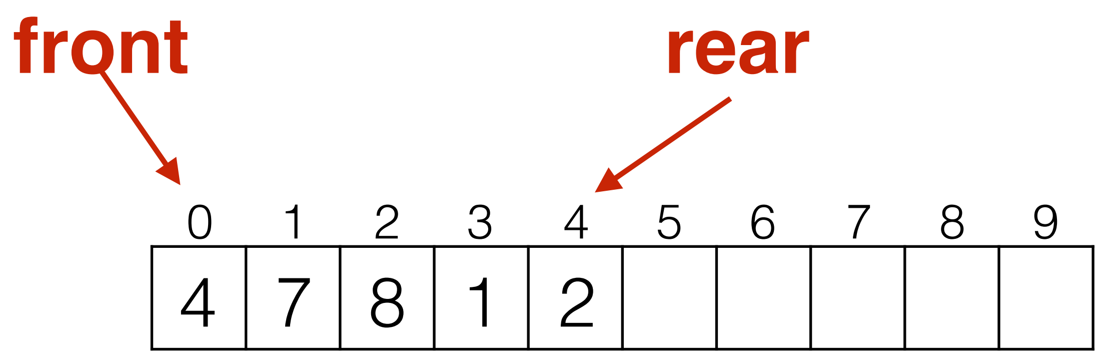
If we wave the constraint on the front, both enqueue and dequeue can be done in $\Theta(1)$ time.



	Front/1st	Back/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Remove	$\Theta(1)$	$\Theta(1)$

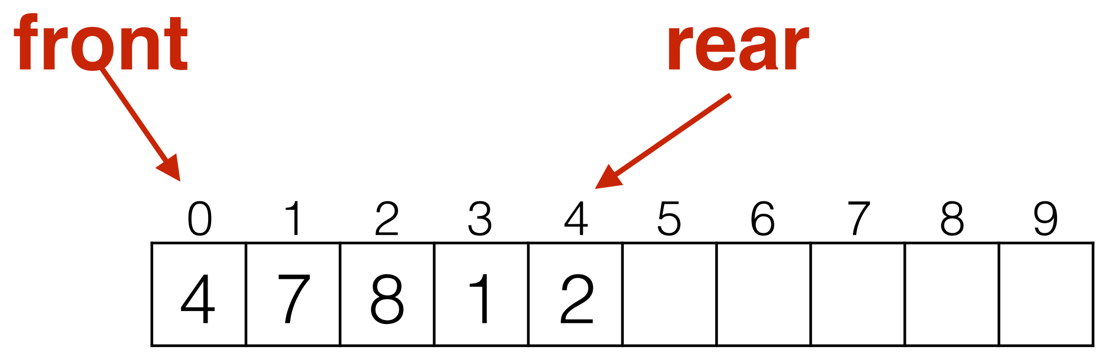
Array Implementation

Double ended array



Array Implementation

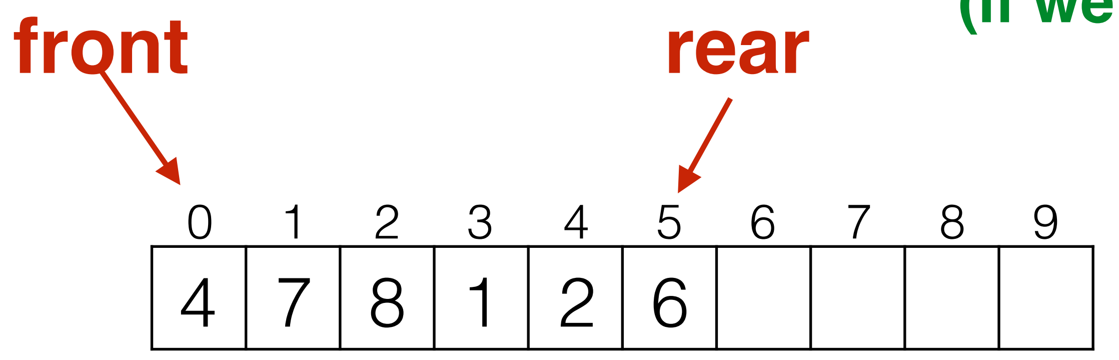
Double ended array



before calling enqueue(6)

Array Implementation

Double ended array

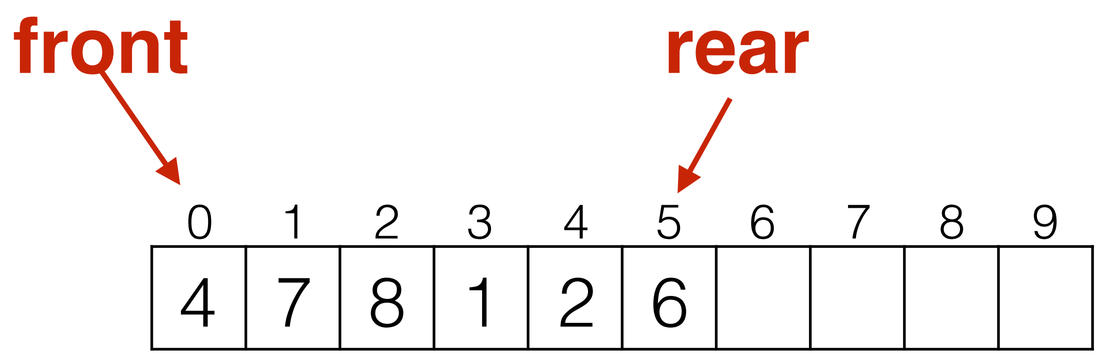


simply update the rear to rear + 1 (if we have enough space)

after calling enqueue(6)

Array Implementation

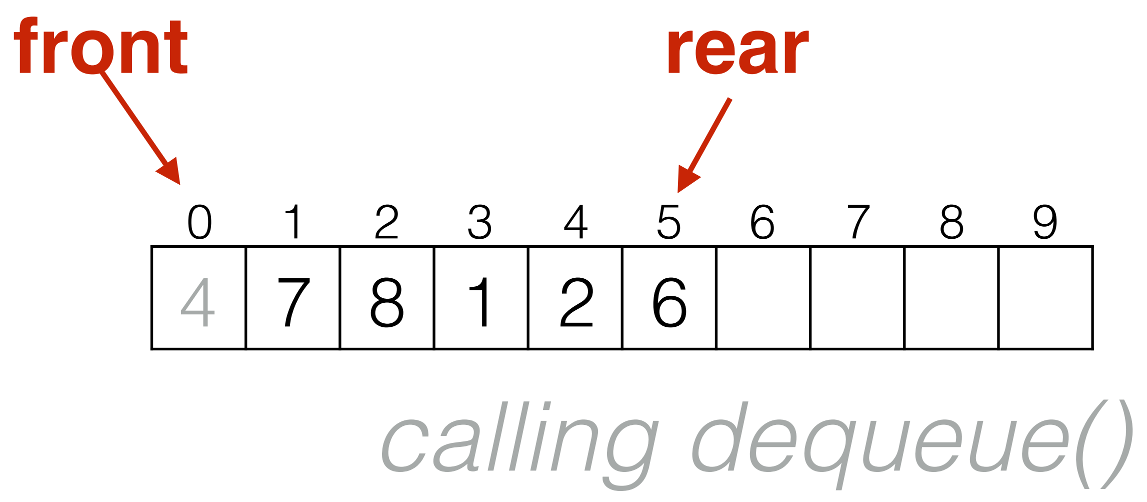
Double ended array



before calling dequeue()

Array Implementation

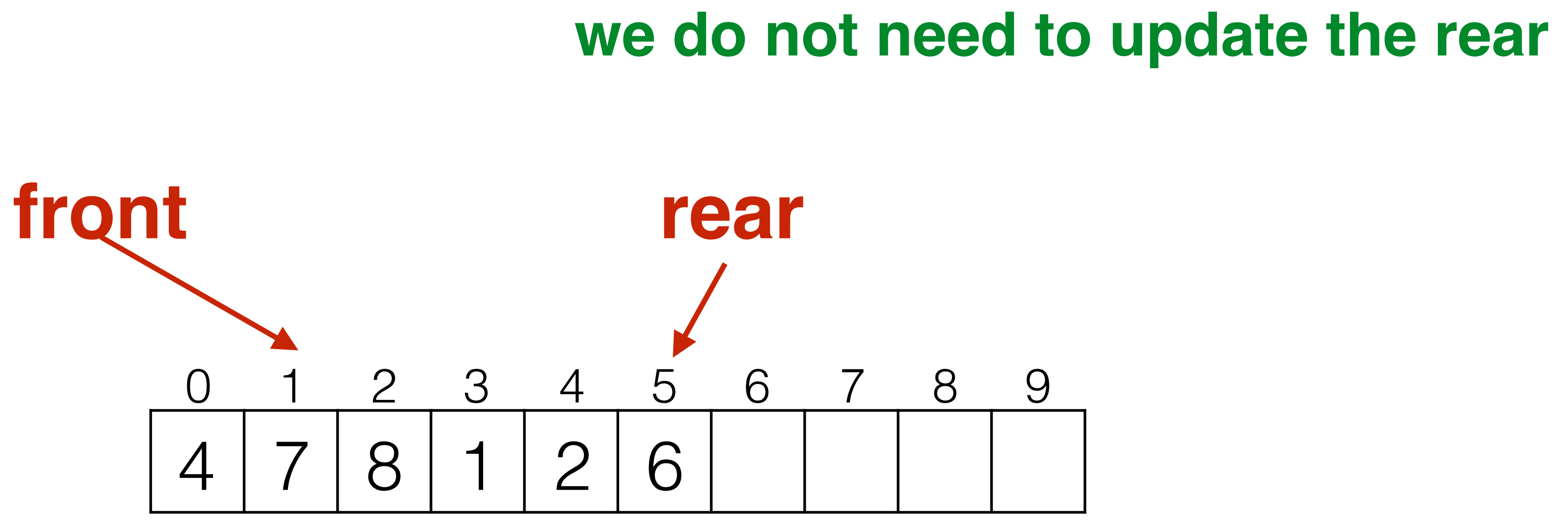
Double ended array



This time we do need to update front, therefore we need a variable too keep track of front. So calling dequeue is $\Theta(n)$.

Array Implementation

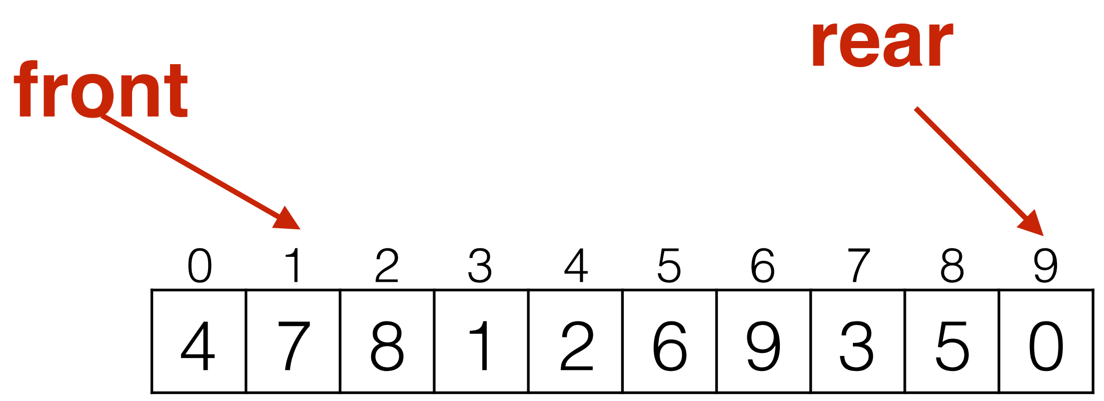
Double ended array



after calling dequeue()

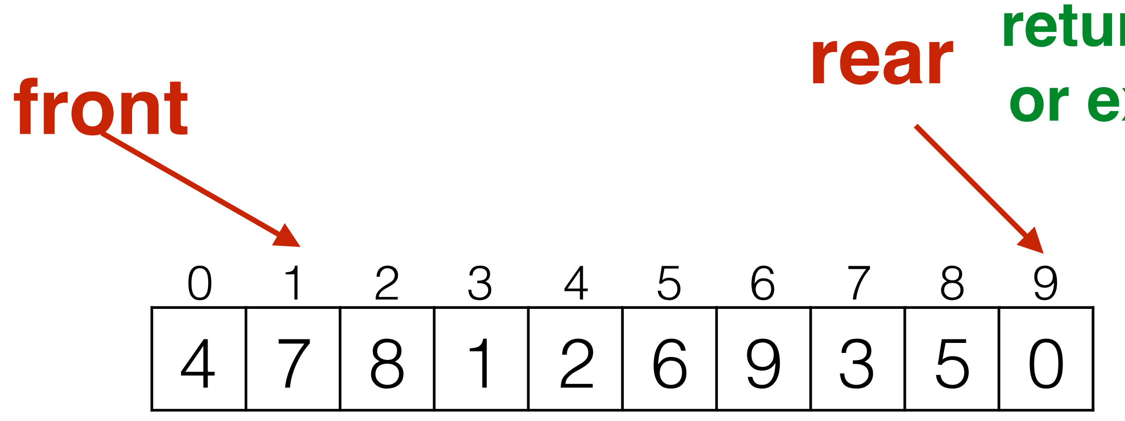
Array Implementation

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



Array Implementation

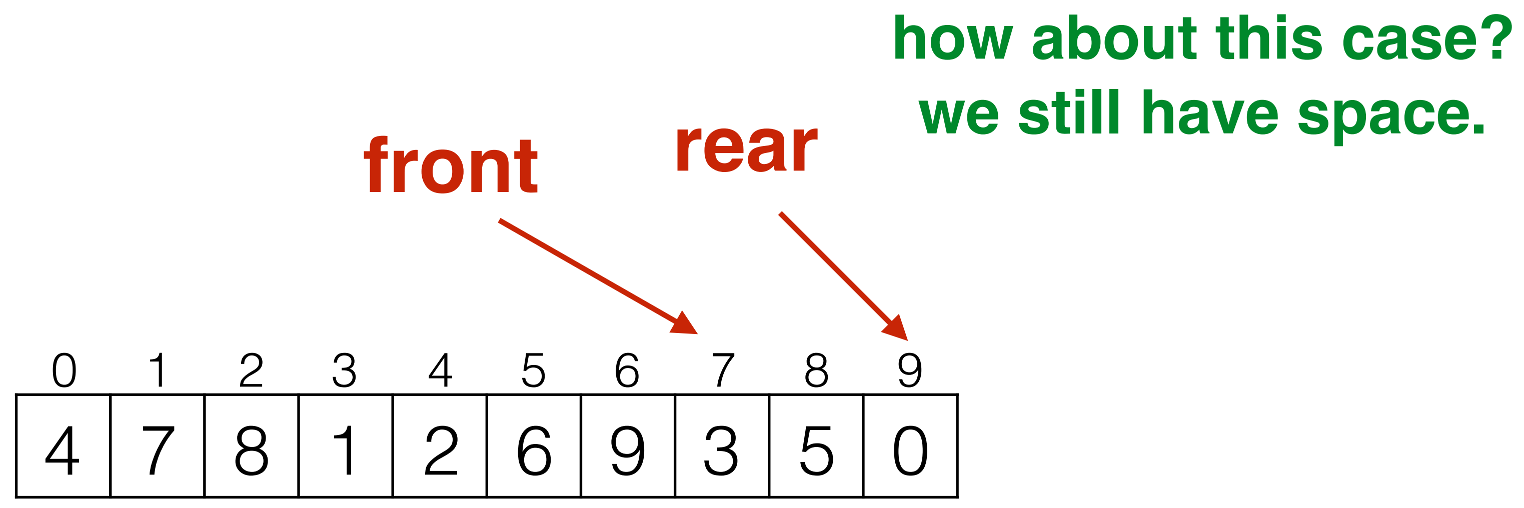
What if we the rear reaches to the end of array (last index)?



if the queue is full so we need consider returning an exception, or expanding the array

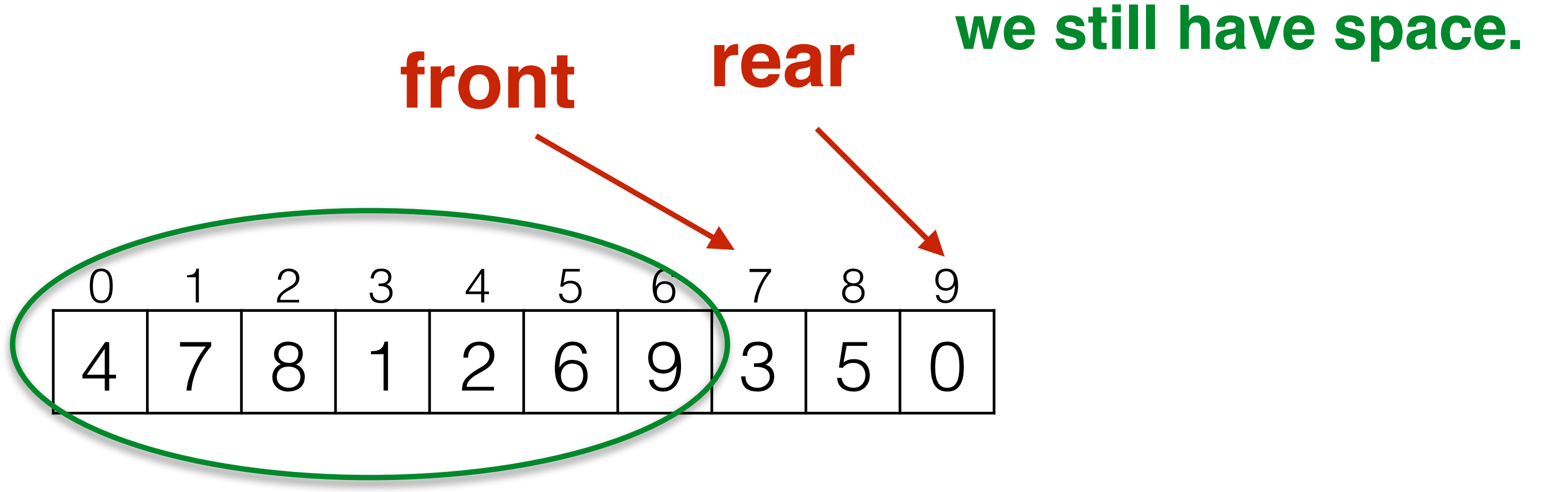
Array Implementation

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



Array Implementation

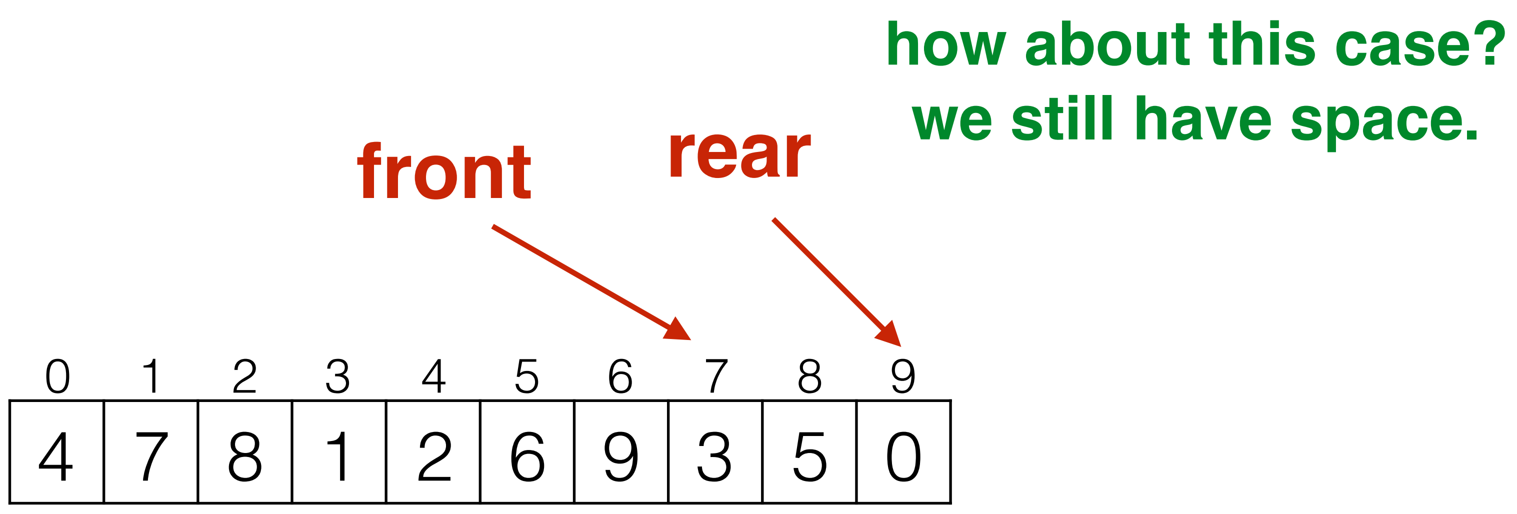
What if we the rear reaches to the end of array (last index)?



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

Array Implementation

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

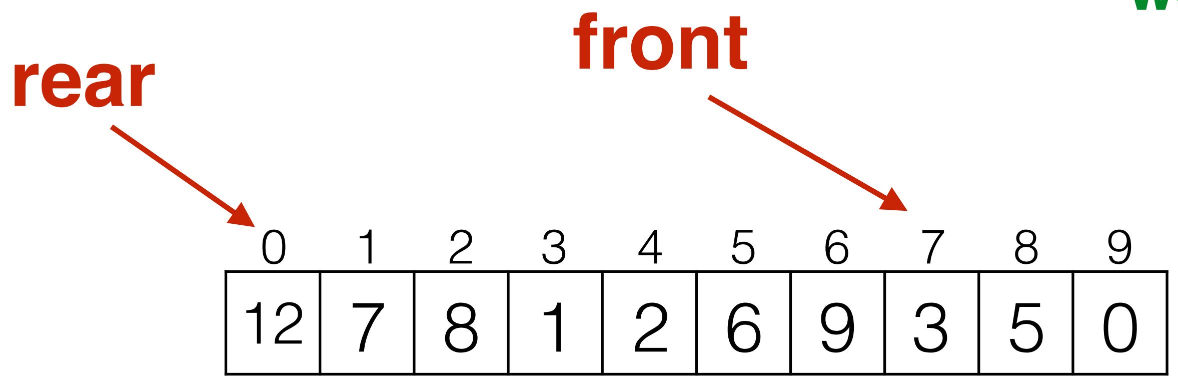


before calling enqueue(12)

Array Implementation

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

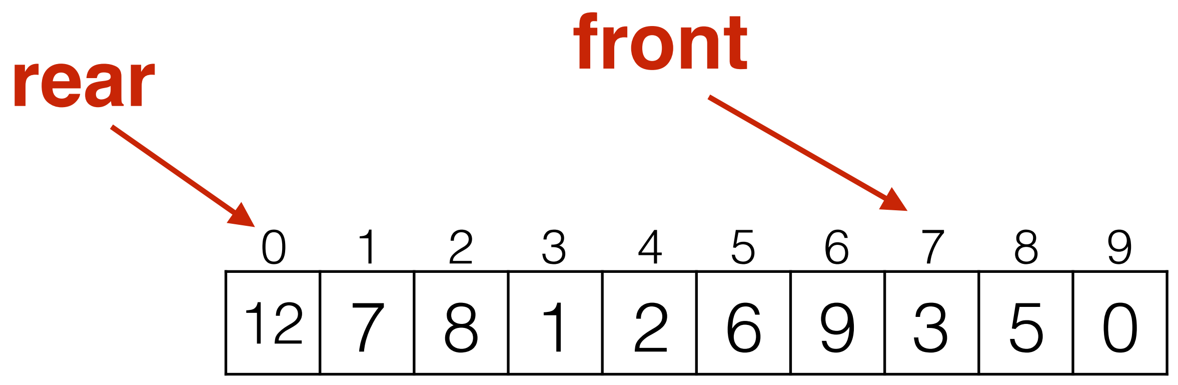
how about this case?
we still have space.



calling enqueue(12)

Array Implementation

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



the rear is reset to 0

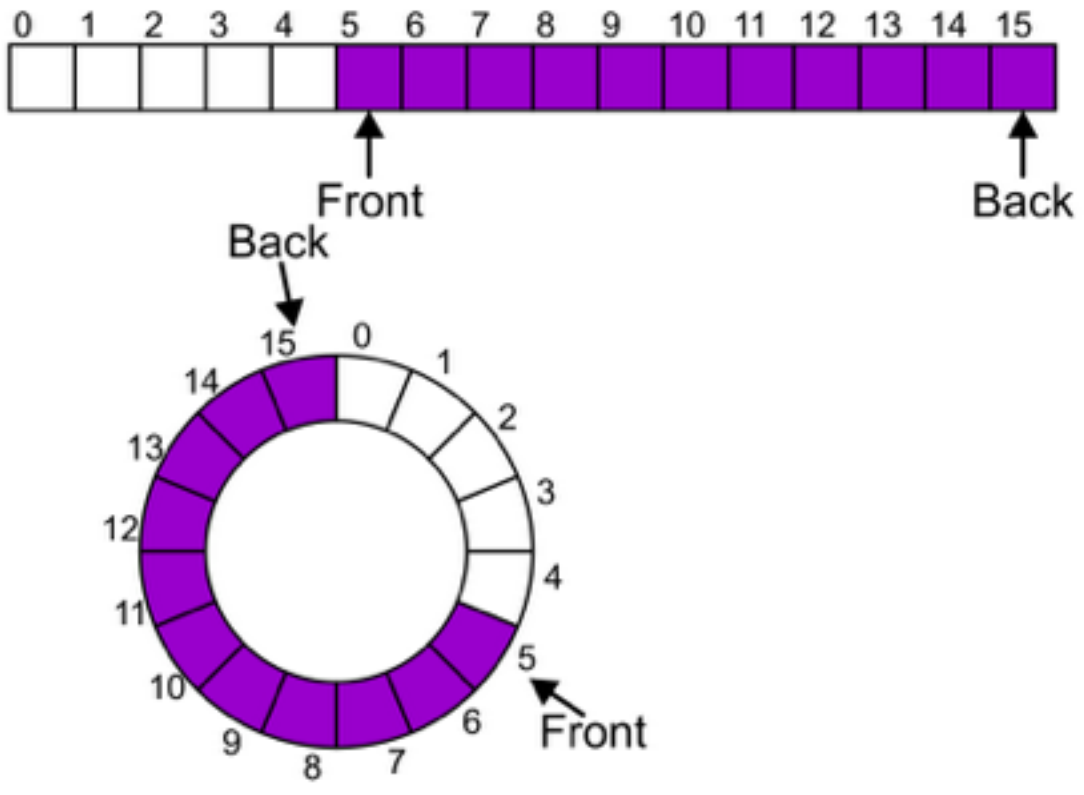
after calling enqueue(12)

Array Implementation

Instead of viewing the array on the range 0, …, 15, consider the indices being cyclic:

…, 15, 0, 1, …, 15, 0, 1, …, 15, 0, 1, …

This is referred to as a *circular array*



Exceptions

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

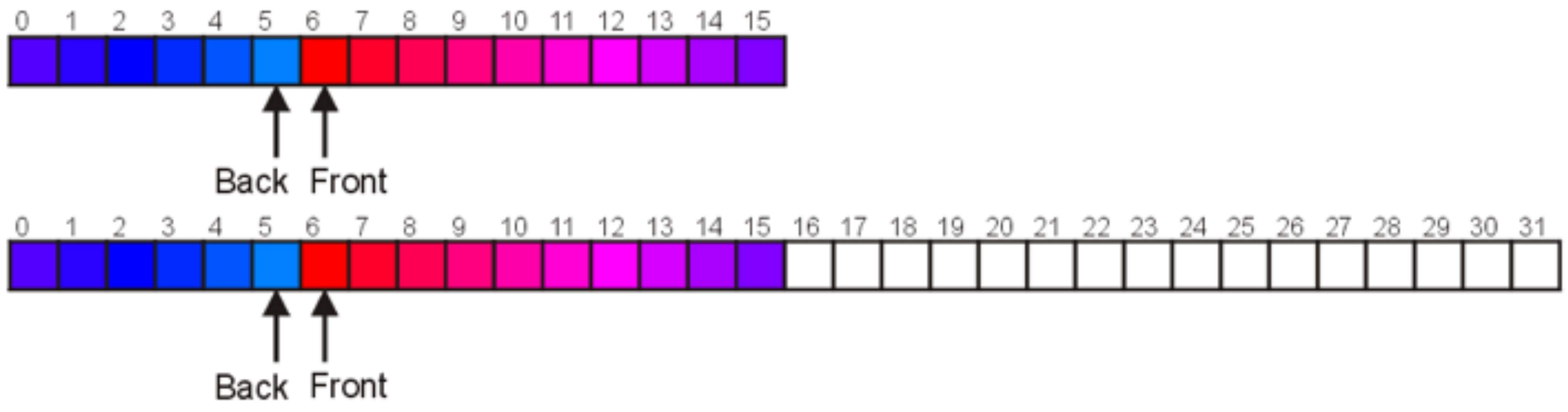
If the array is filled, we have five options:

- Increase the size of the array
- Throw an exception
- Ignore the element being pushed

Increasing Capacity

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

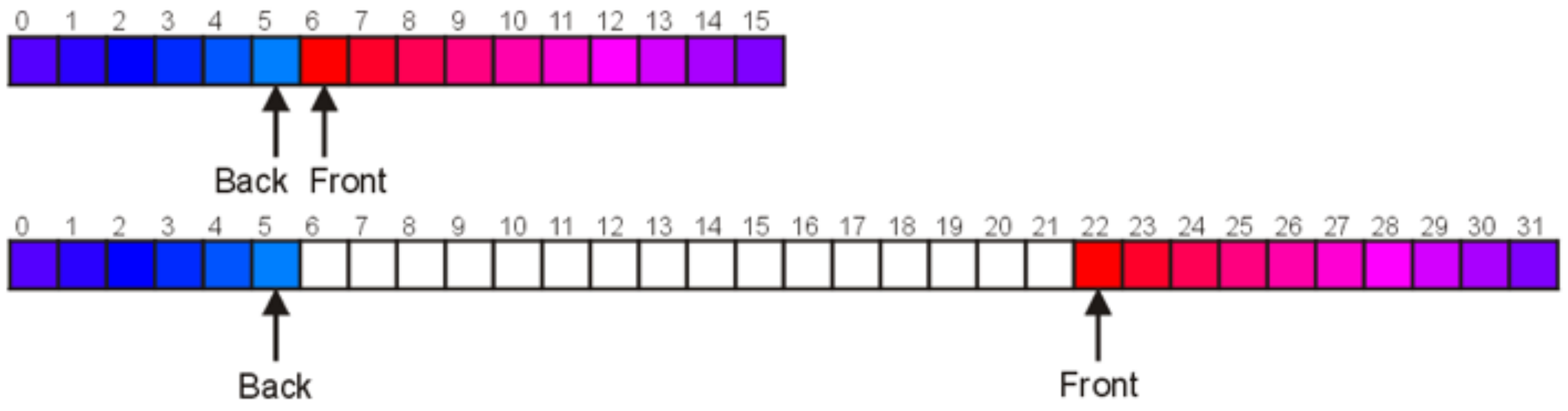
– A direct copy does not work:



Increasing Capacity

There are two solutions:

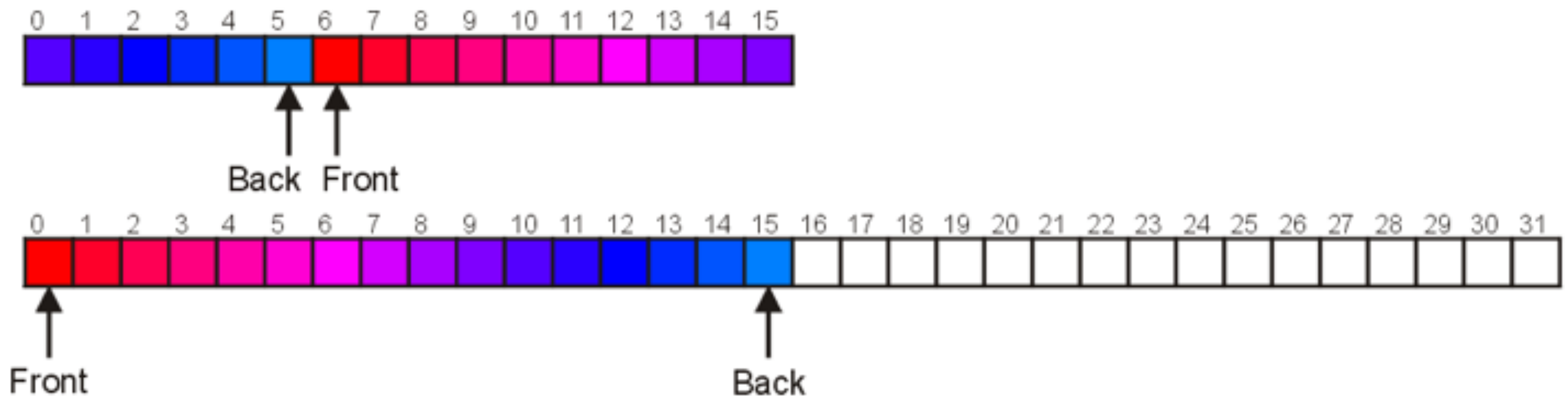
- Move those beyond the front to the end of the array
- The next push would then occur in position 6



Increasing Capacity

An alternate solution is normalization:

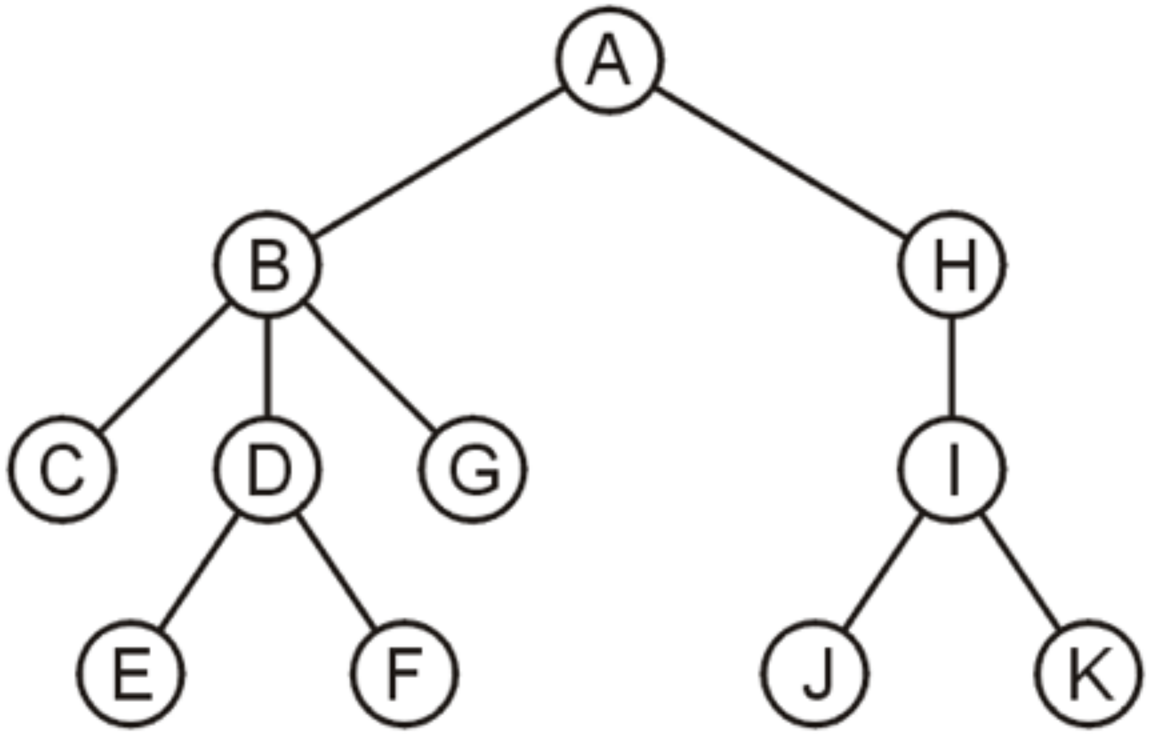
- Map the front back at position 0
- The next push would then occur in position 16



Application

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

- Consider searching the directory structure

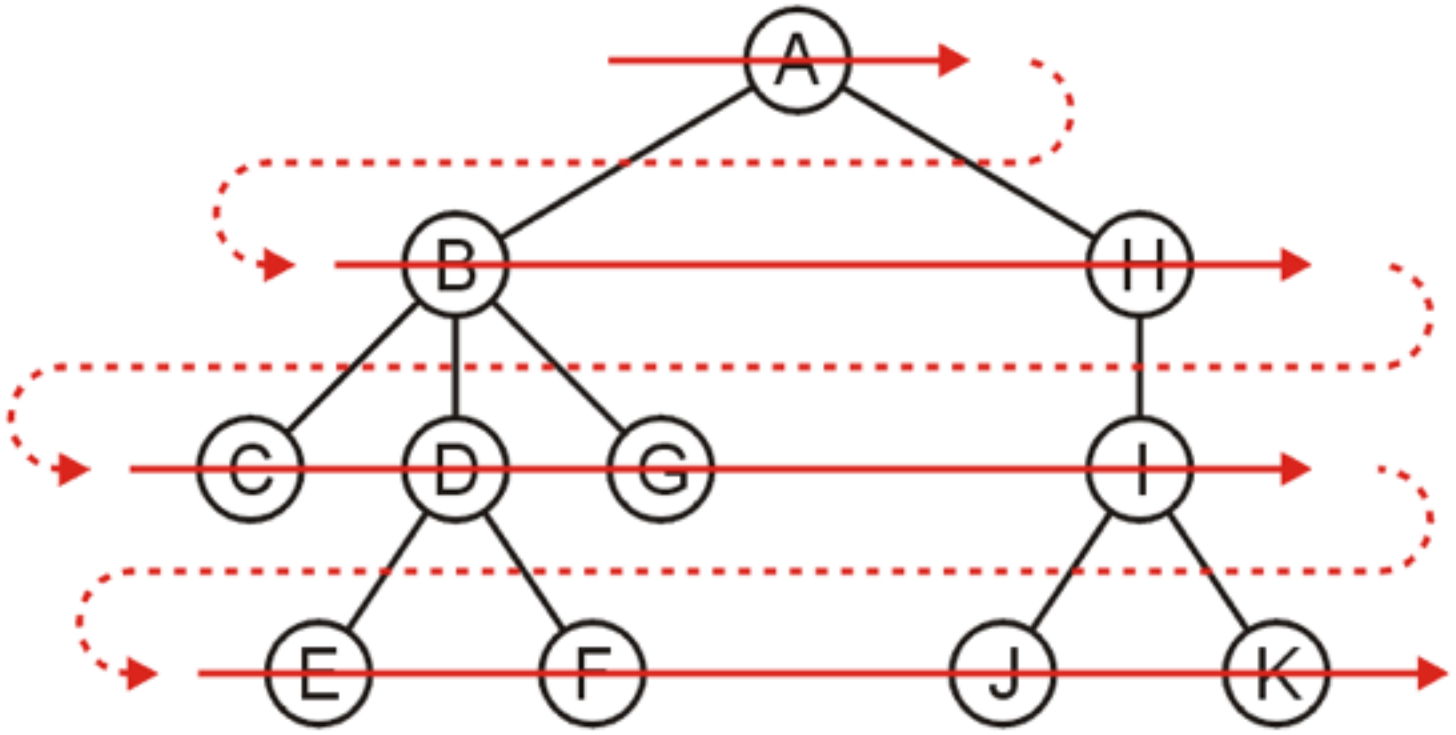


Application

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

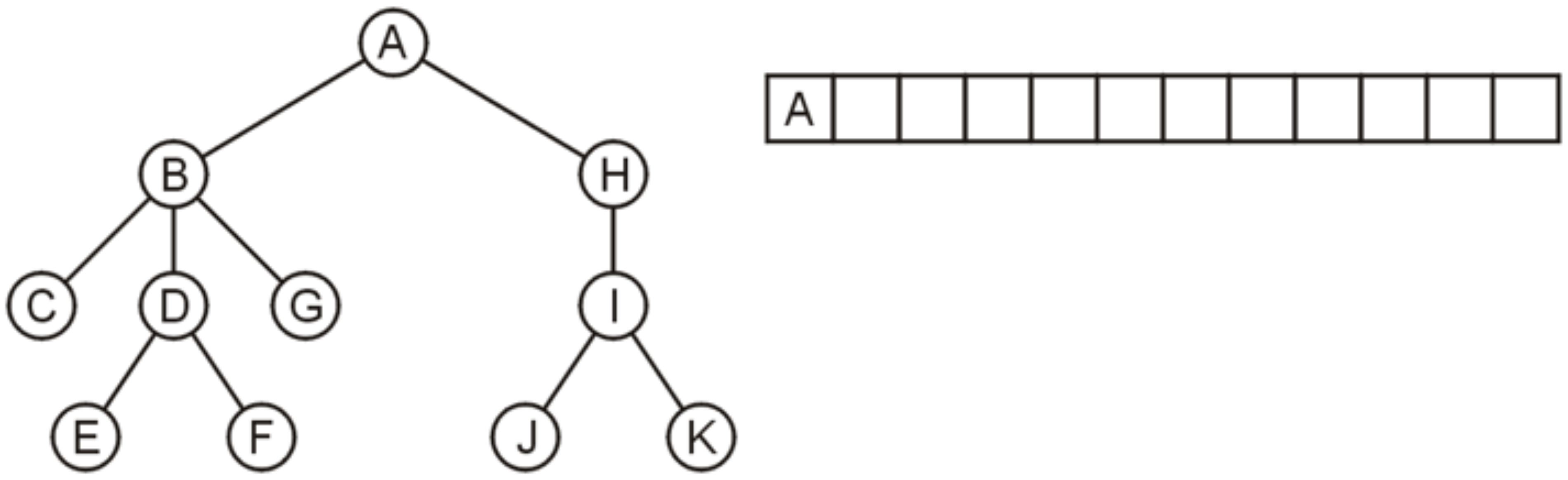
The easiest implementation is:

- 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

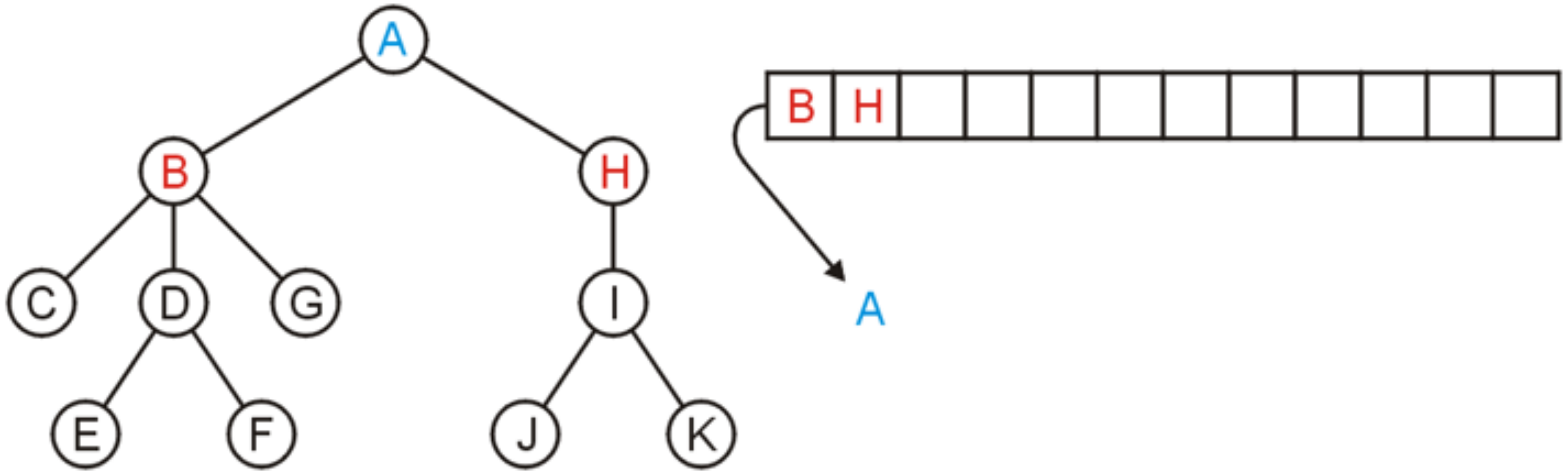
Application

Push the root directory A



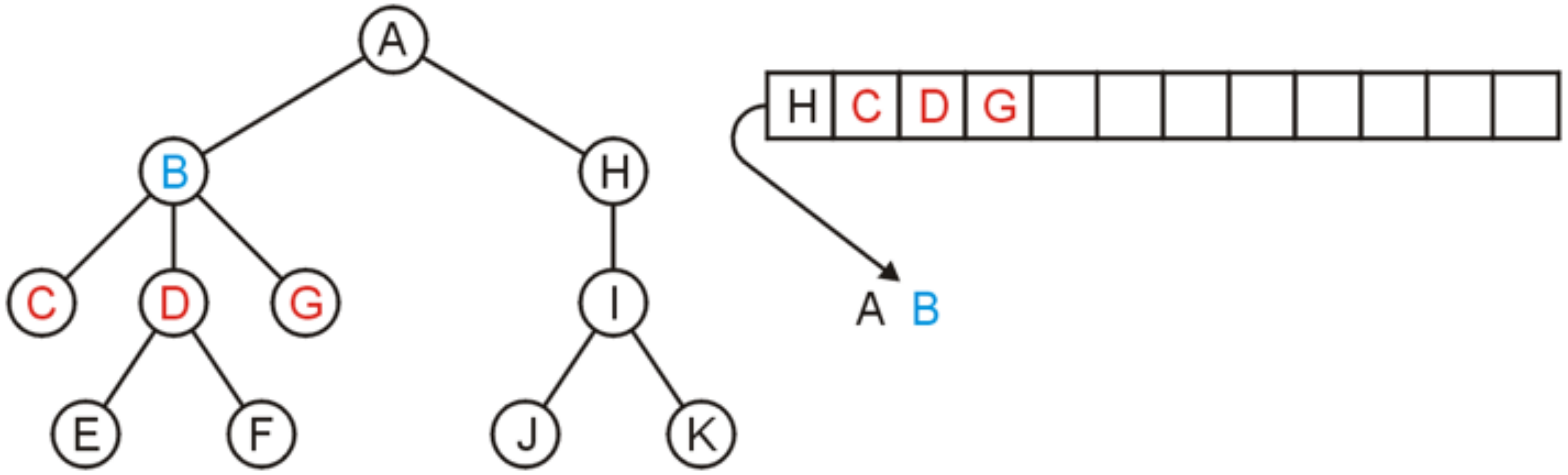
Application

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



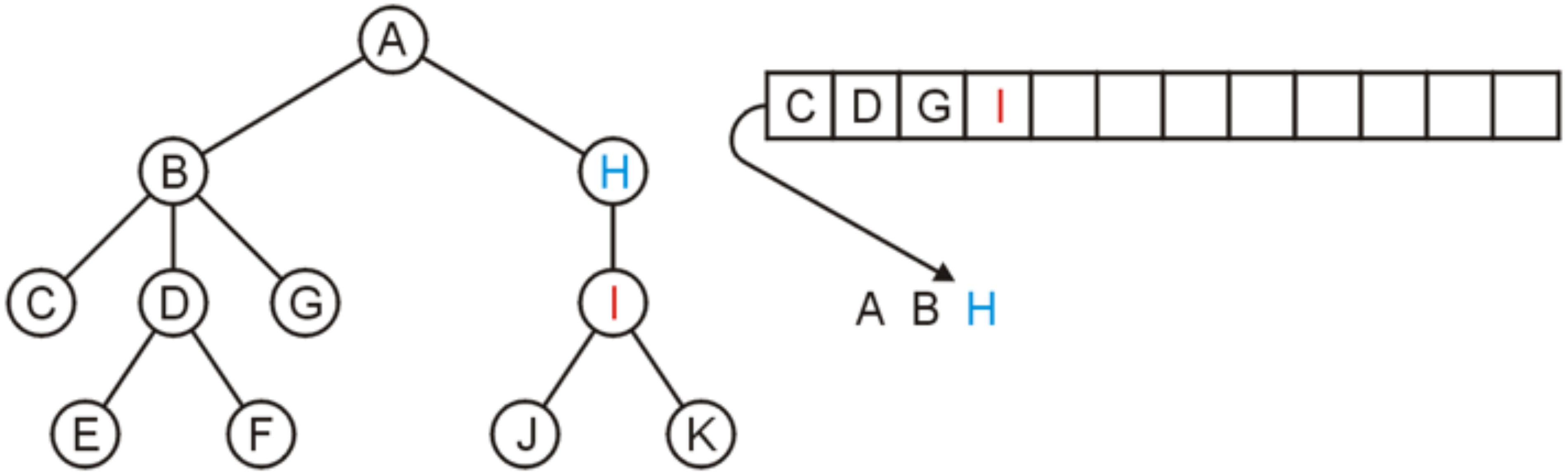
Application

Pop B and push C, D, and G



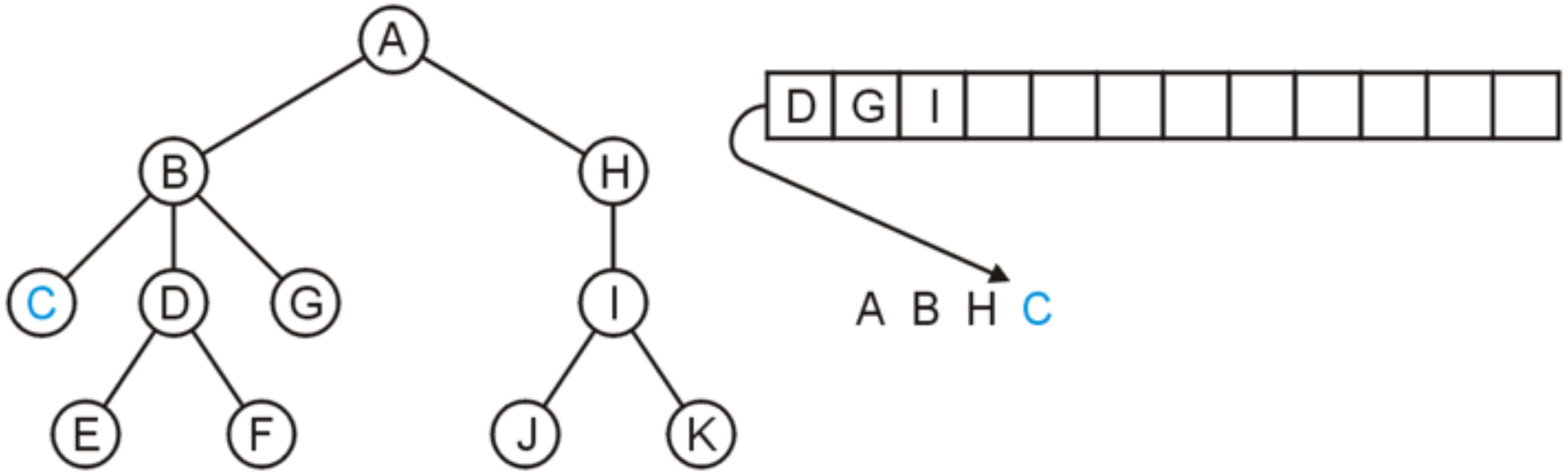
Application

Pop H and push its one sub-directory I



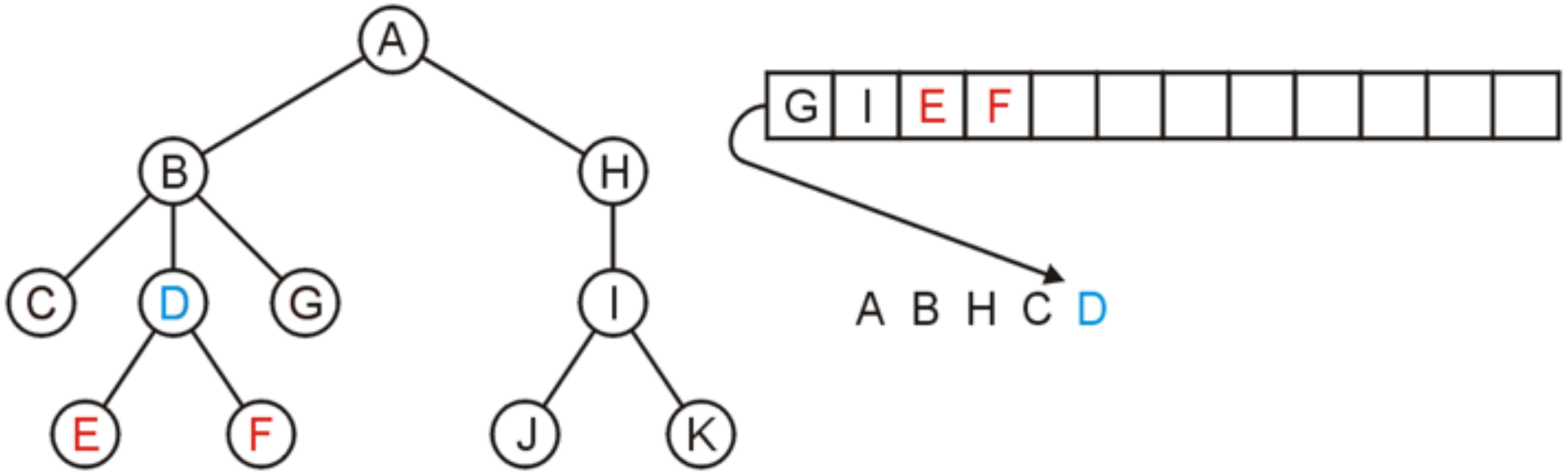
Application

Pop C: no sub-directories



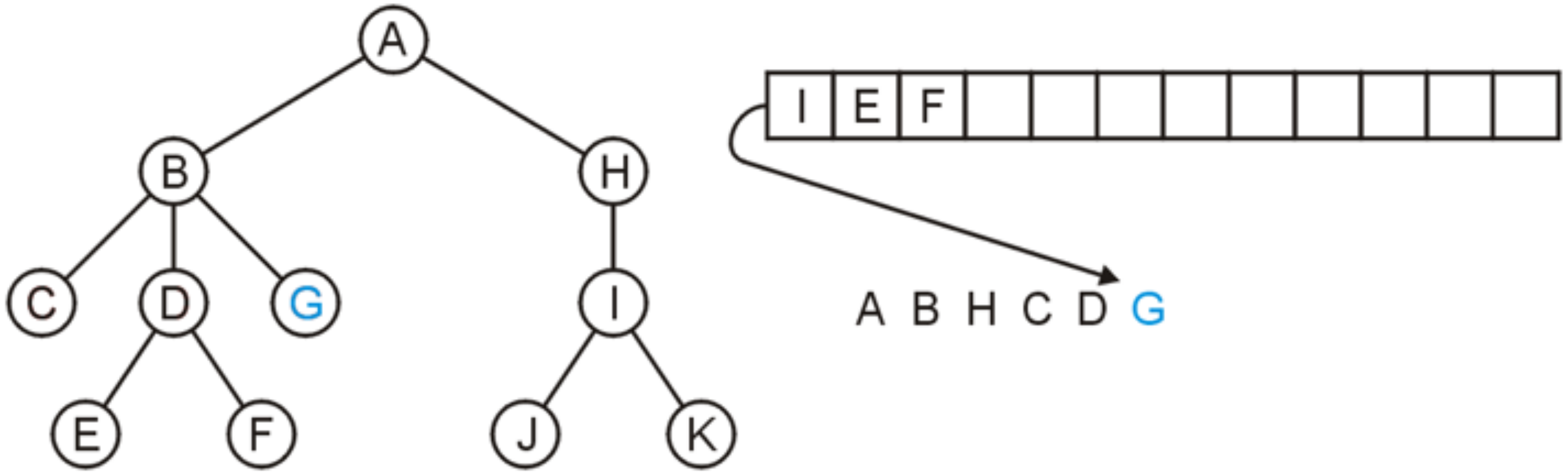
Application

Pop D and push E and F



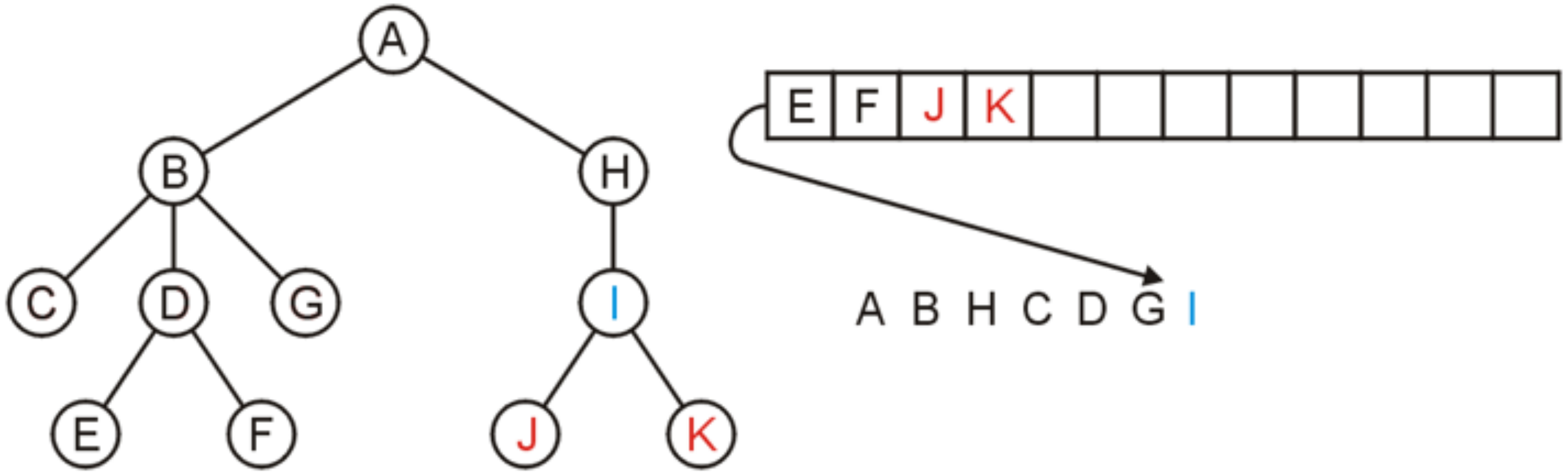
Application

Pop G



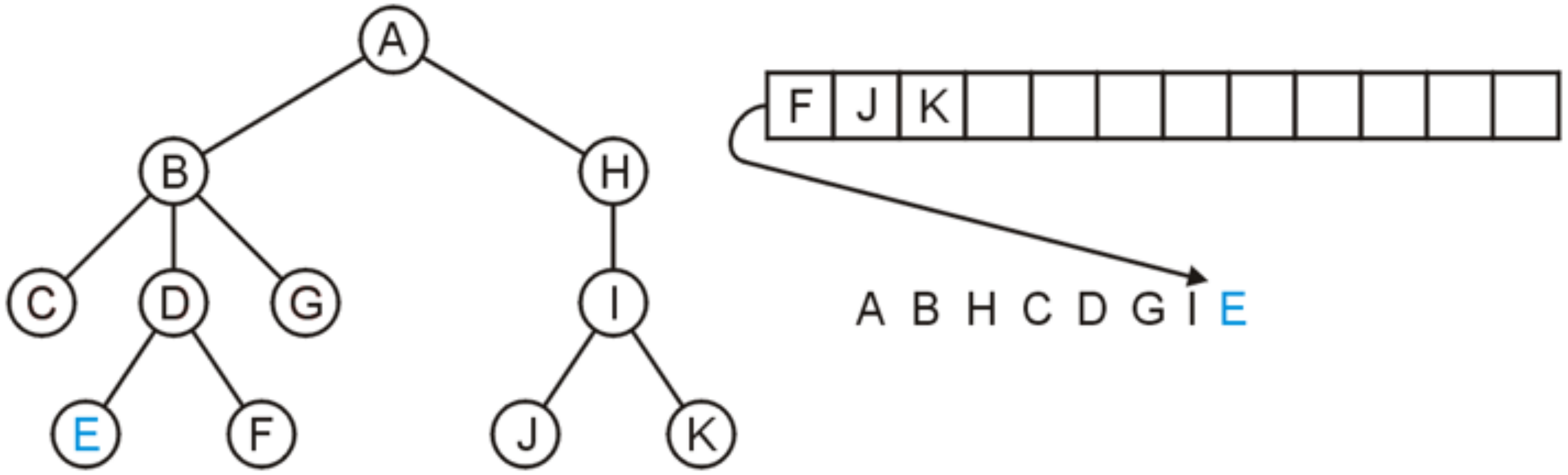
Application

Pop I and push J and K



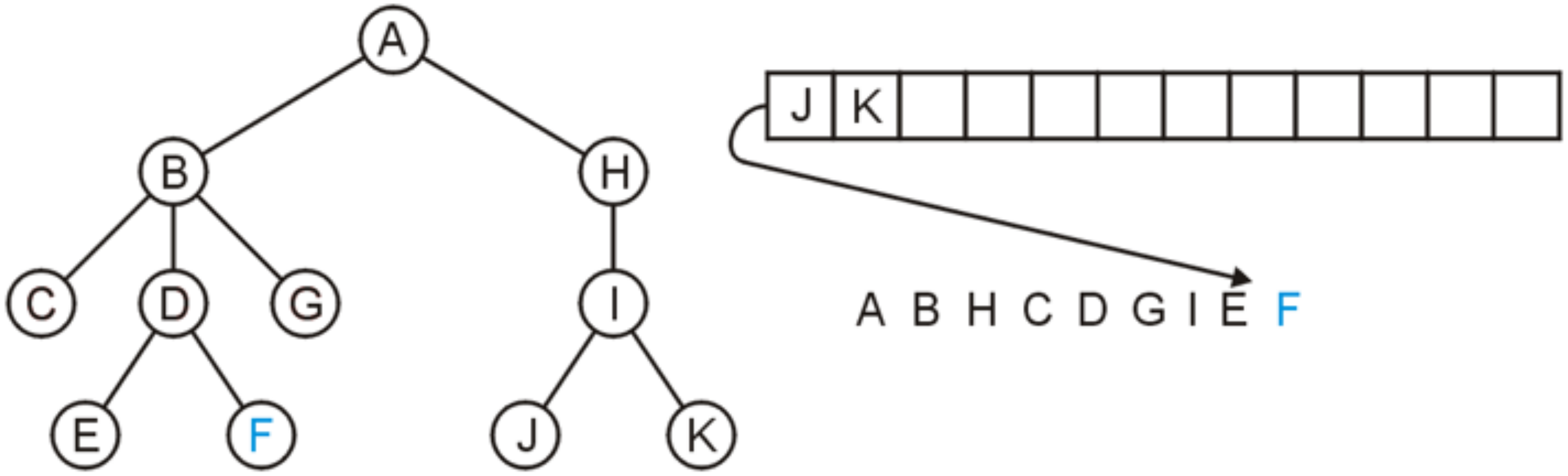
Application

Pop E



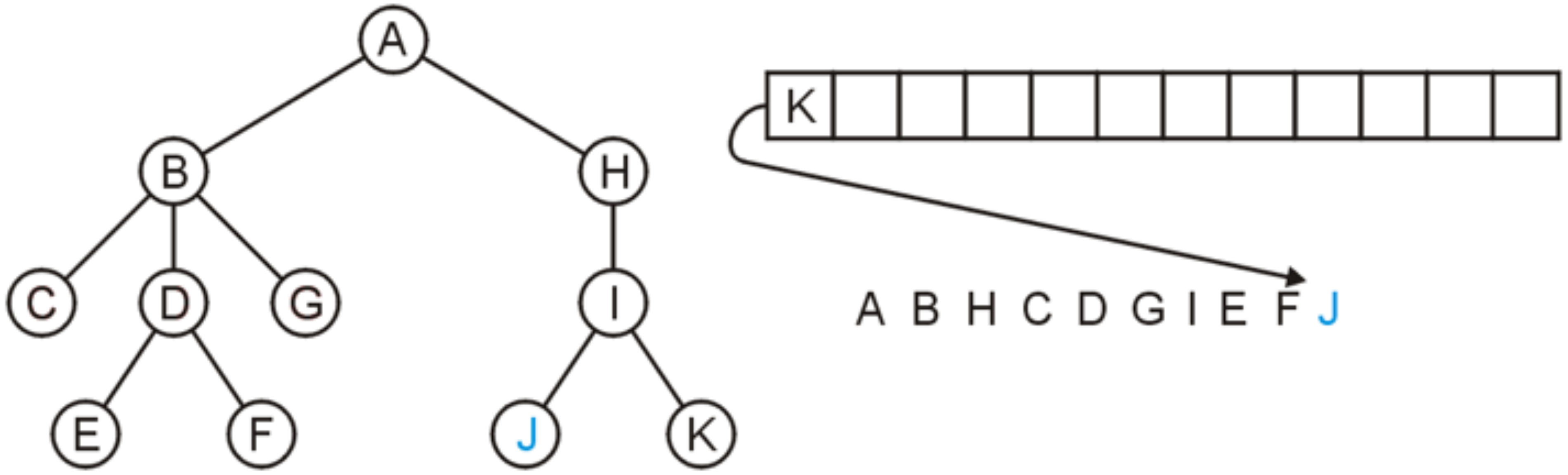
Application

Pop F



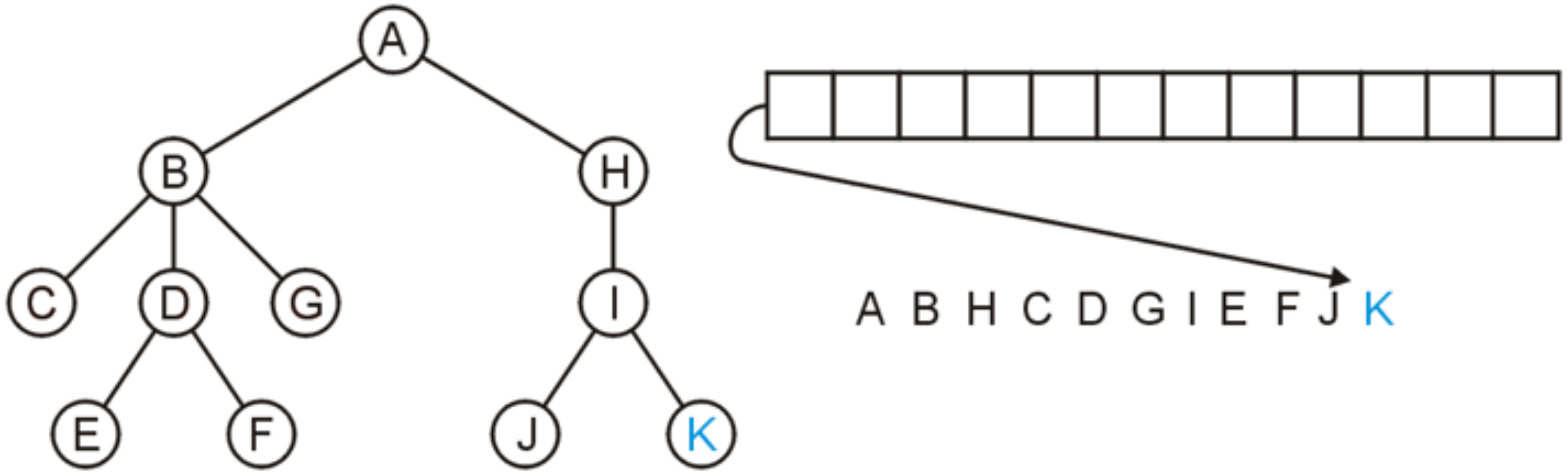
Application

Pop J



Application

Pop K and the queue is empty

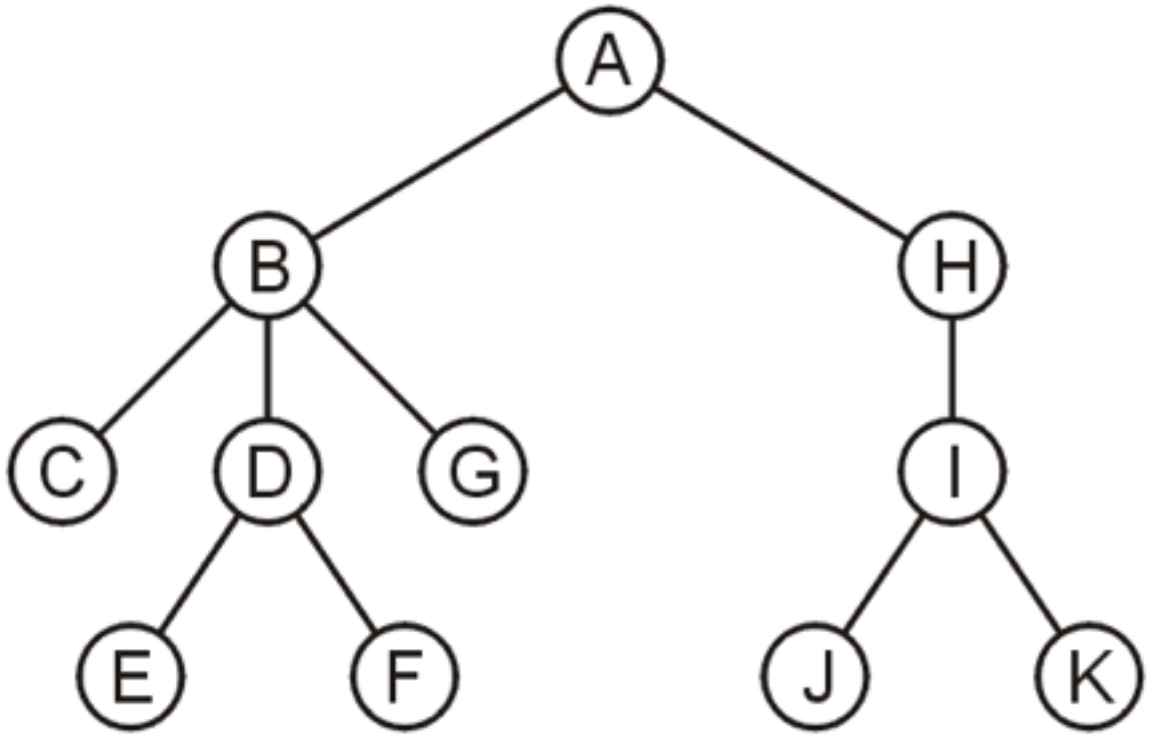


Application

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