بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

In the name of Allah, Most Gracious, Most Merciful

CSE 4303
Data Structure

Topic: Queues, Deques

Asaduzzaman Herok
Lecturer | CSE | IUT
asaduzzaman34@iut-dhaka.edu

Abstract Queue

A queue is a list from which items are deleted from one end (front) and into which items are inserted at the other end (rear or back)

- It is like line of people waiting to purchase tickets

Queue is referred to as a first-in-first-out (FIFO) data structure

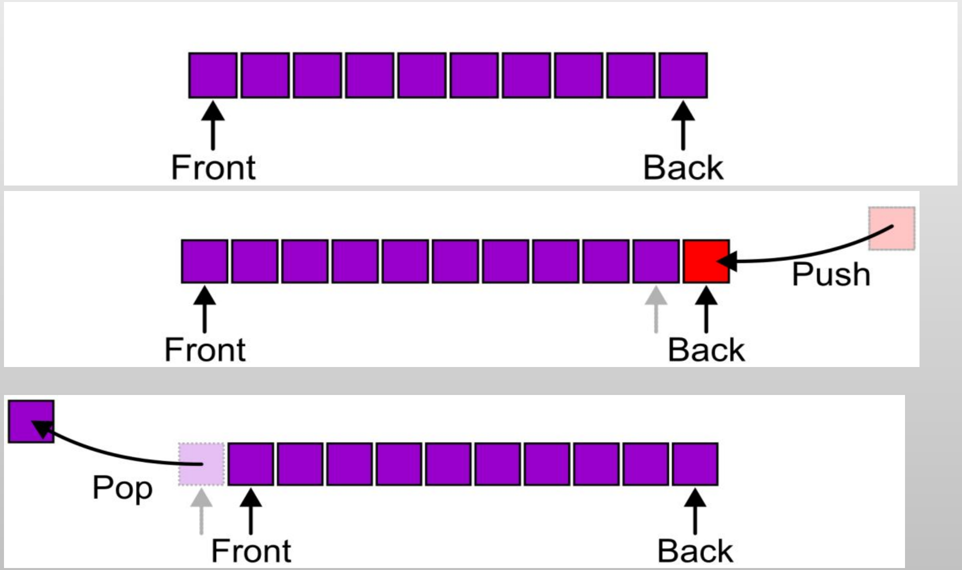
- The first item inserted into a queue is the first item to leave

The object designated as the *front* of the queue is the object which was in the queue the longest

Abstract Queue

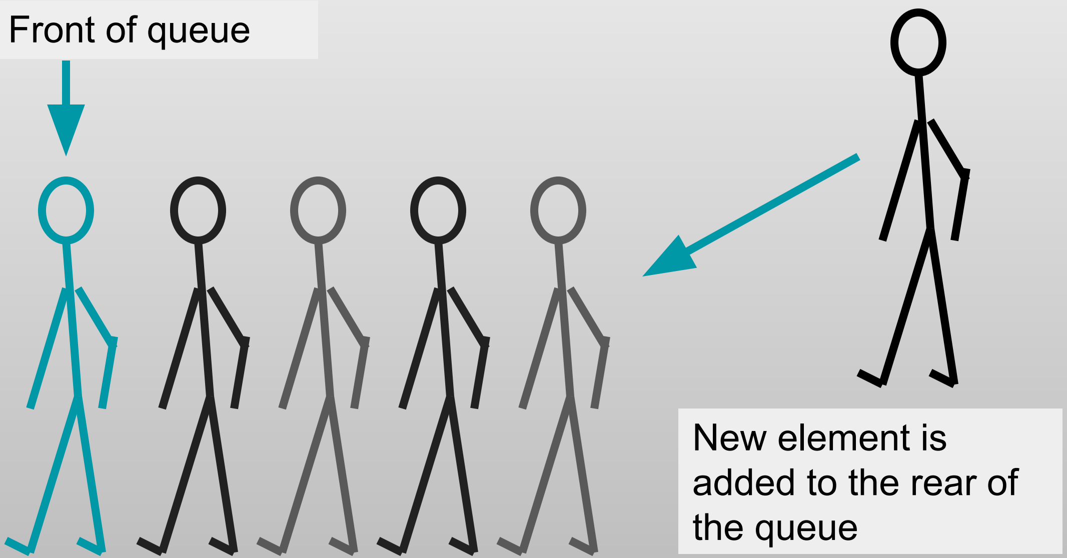
Also called a *first-in–first-out* (FIFO) data structure

- Graphically, we may view these operations as follows:



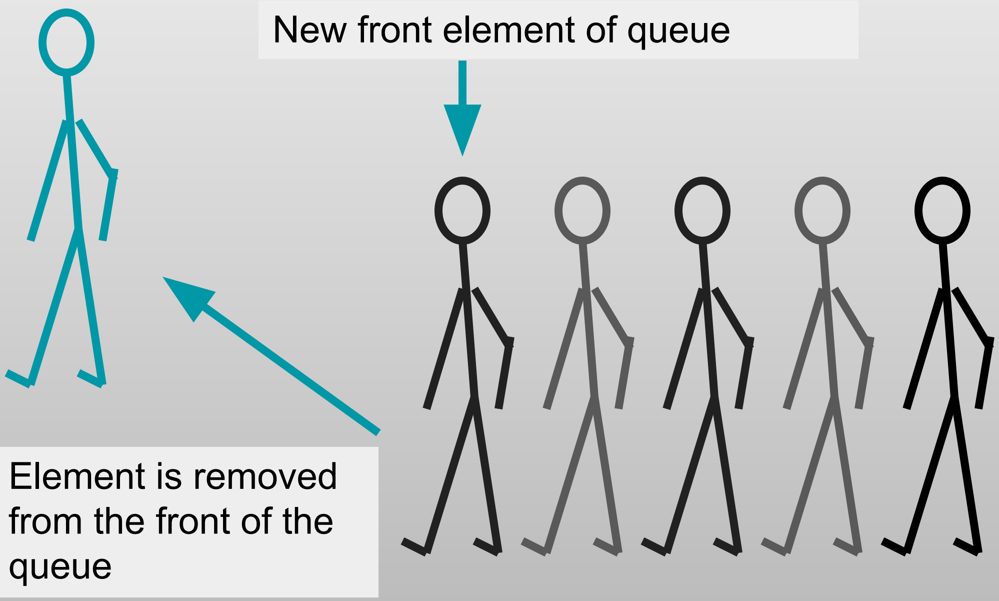
Conceptual View of a Queue

Adding an element



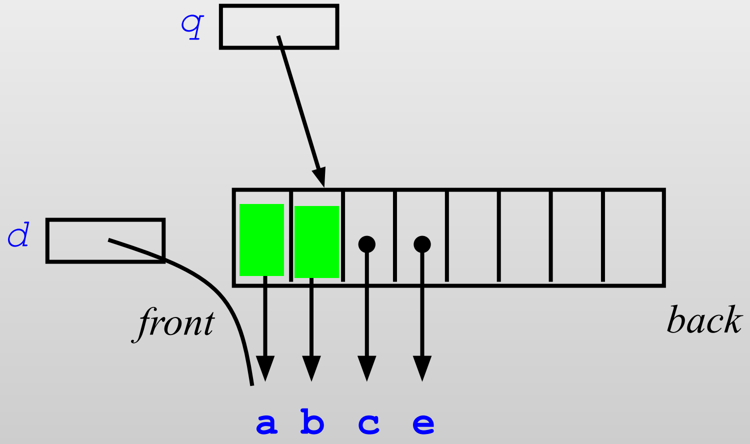
Conceptual View of a Queue

Removing an element


New front element of queue
Element is removed from the front of the queue

Sample Operation

```
Queue  *Q;

enqueue(Q, “a”);

enqueue(Q, “b”);

enqueue(Q, “c”);

d=getFront(Q);

dequeue(Q);

enqueue(Q, “e”);

dequeue(Q);
```



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

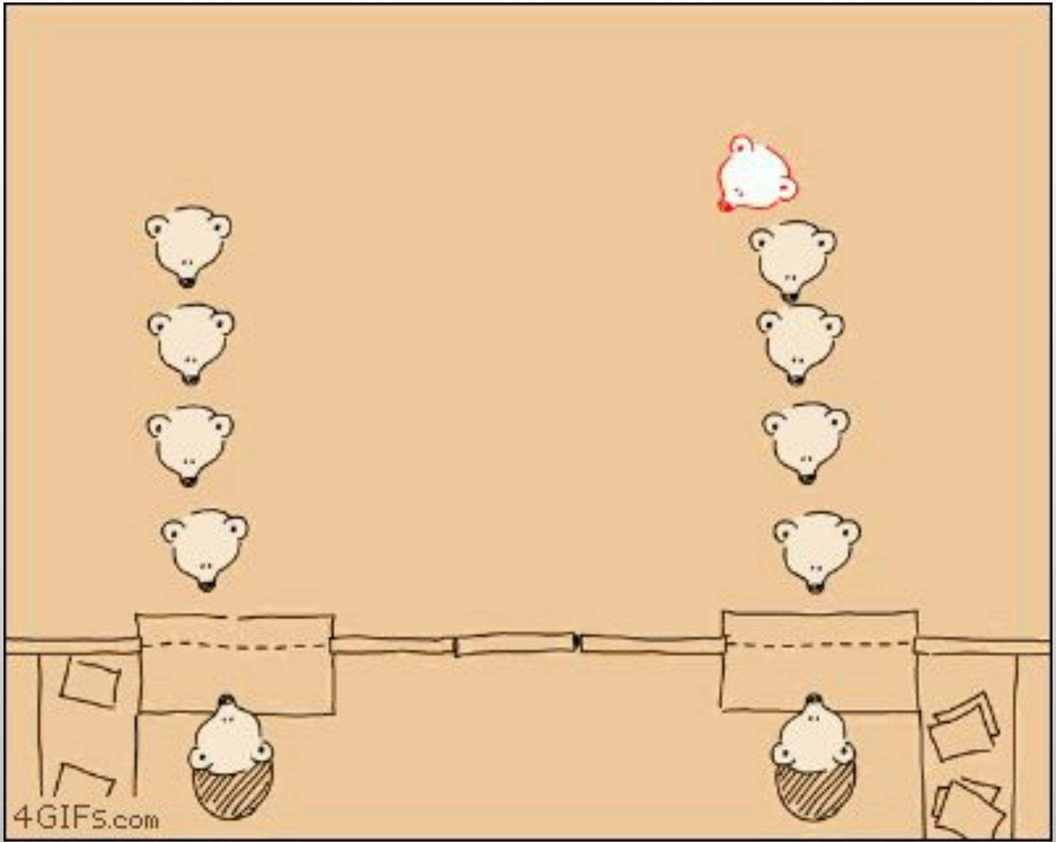
Most shared computer services are servers:

- ❑ Web, file, ftp, database, mail, printers *etc*.
- ❑ Keyboard input buffer
- ❑ GUI event queue (click on buttons, menu items)

Uses of Queues in Computing

In ***simulation studies***, where the goal is to reduce waiting times:

- ➢Optimize the flow of traffic at a traffic light
- ➢Determine number of cashiers to have on duty at a grocery store at different times of day
- ➢Consider an information center
  - ■ Calls arrive at random intervals
  - ■ Placed in queue of incoming calls
  - ■ When agent becomes available, services call at front of queue


4GIFs.com

Using Queues: Coded Messages

A ***Caesar cipher*** is a ***substitution code*** that encodes a message by shifting each letter in a message by a constant amount **k**

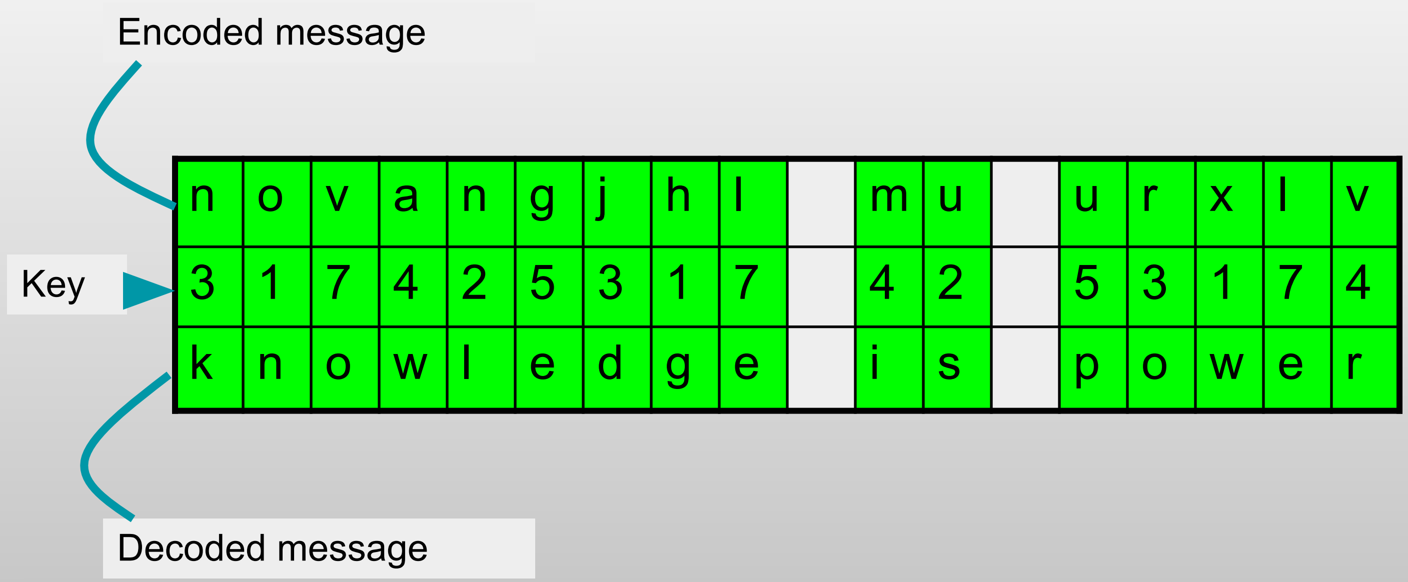
- If **k** is **5**, **a** becomes **f**, **b** becomes **g**, etc.
  - ***Example***: ***n qtaj ofaf***
- Used by Julius Caesar to encode military messages for his generals (around 50 BC)
- This code is fairly easy to break!

Using Queues: Coded Messages

- ***An improvement***: change how much a letter is shifted depending on where the letter is in the message
- A ***repeating key*** is a sequence of integers that determine how much each character is shifted
  - Example: consider the repeating key

    3 1 7 4 2 5
  - The first character in the message is shifted by 3, the next by 1, the next by 7, and so on
  - When the key is exhausted, start over at the beginning of the key

An Encoded Message Using a Repeated Key



n	o	v	a	n	g	j	h	l		m	u		u	r	x	l	v
3	1	7	4	2	5	3	1	7		4	2		5	3	1	7	4
k	n	o	w	l	e	d	g	e		i	s		p	o	w	e	r

Using Queues: Coded Messages

➔ We can use a queue to store the values of the key

- ***dequeue*** a key value when needed
- After using it, ***enqueue*** it back onto the end of the queue
    - So, the queue represents the constantly cycling values in the key

Implementations

We will look at two implementations of queues:

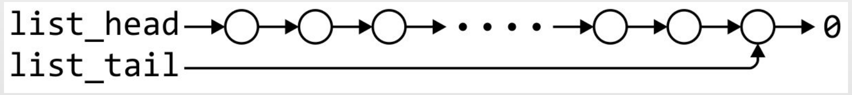
- ➢ Singly linked lists
- ➢ Circular arrays

Requirements:

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

Linked-List Implementation

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

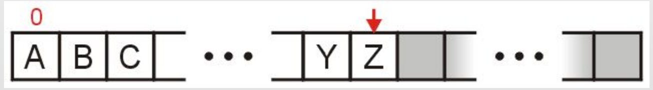


	Front/1st	Back/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

The desired behavior of an Abstract Queue may be reproduced by performing insertions at the back

Array Implementation

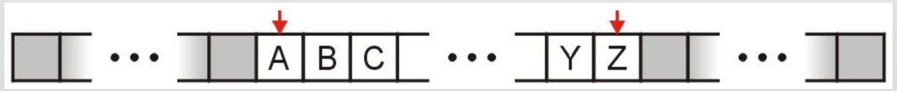
An one-ended array does not allow all operations to occur in $\mathbf{\Theta}(1)$ time



	Front/1st	Back/nth
Find	$\mathbf{\Theta}(1)$	$\mathbf{\Theta}(1)$
Insert	$\mathbf{\Theta}(n)$	$\mathbf{\Theta}(1)$
Erase	$\mathbf{\Theta}(n)$	$\mathbf{\Theta}(1)$

Array Implementation

Using a two-ended array, $\Theta(1)$ are possible by pushing at the back and popping from the front



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

Array Implementation

We need to store an array:

We need additional information, including:

- ➢ The number of objects currently in the queue and the front and back indices

```
int queue_size;
int ifront;        // index of the front entry
int iback;         // index of the back entry
```

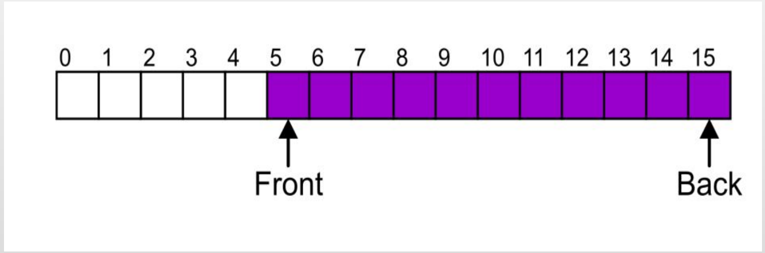
- ➢ The capacity of the array

```
int array_capacity;
```

Problem of One-ended Array

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

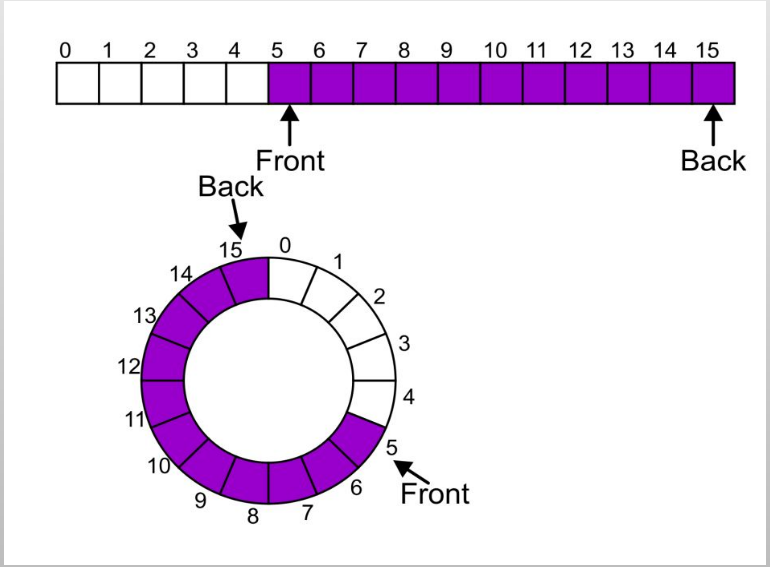
In this case, the array is not full and yet we cannot place any more objects in to the array

Circular Array

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

$$\dots, 15, 0, 1, \dots, 15, 0, 1, \dots, 15, 0, 1, \dots$$

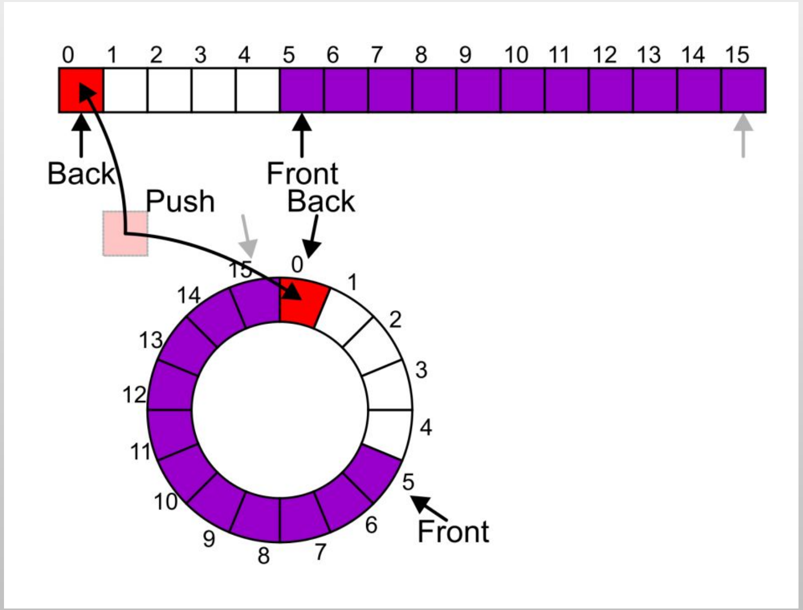
This is referred to as a *circular array*



Circular Array

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

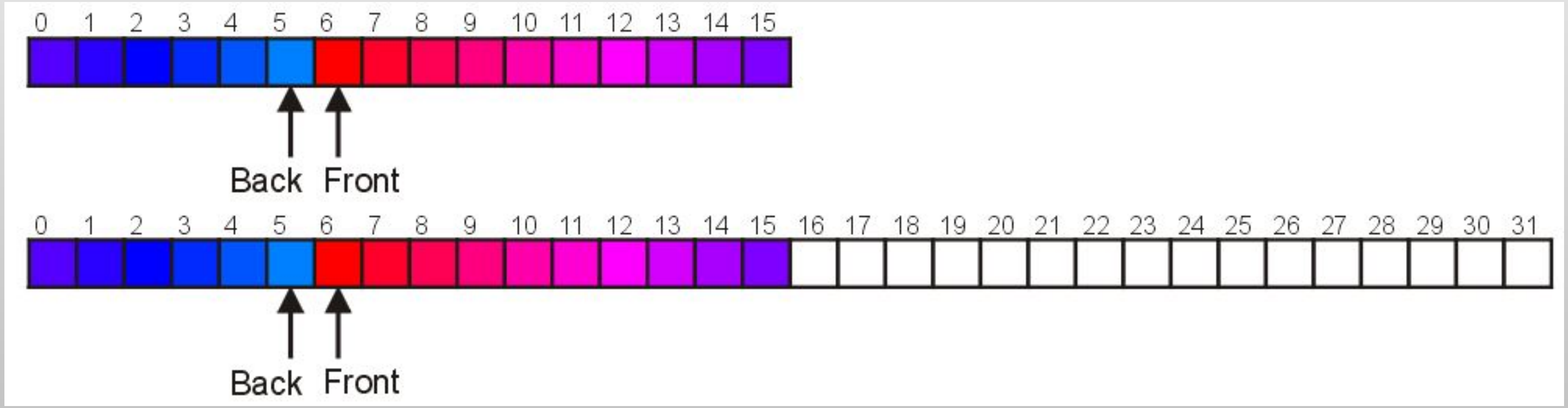
```
++iback;
if ( iback == capacity() ) {
    iback = 0;
}
```



Increasing Capacity

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

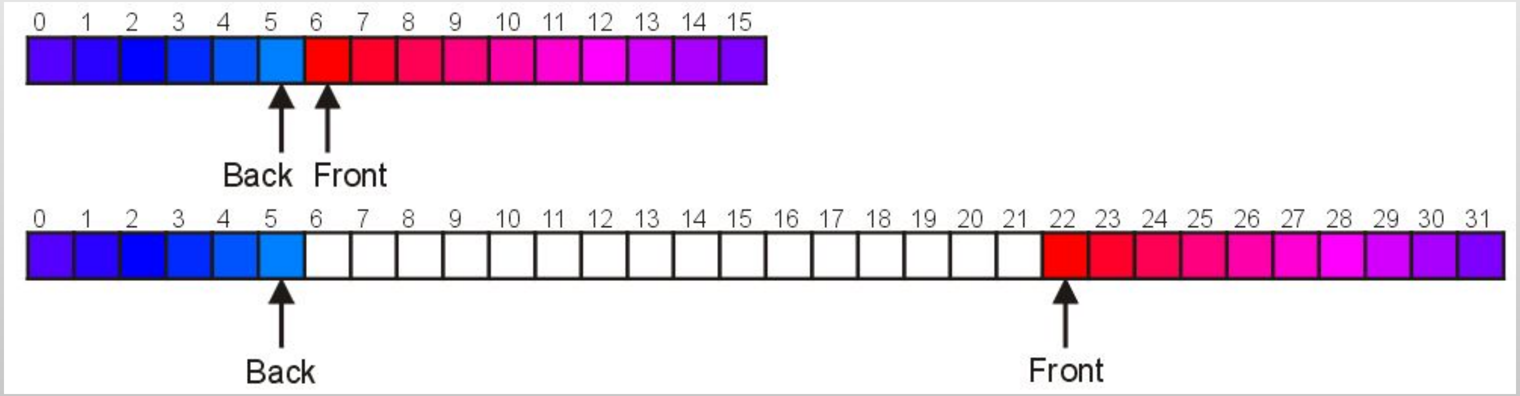
A direct copy does not work:



Increasing Capacity

There are two solutions:

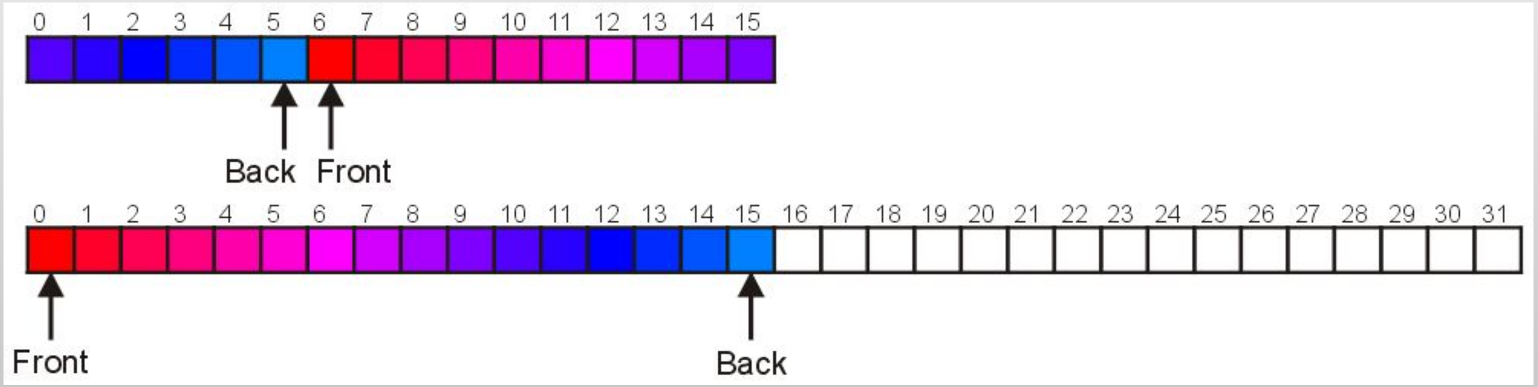
- ➢ Move those beyond the front to the end of the array
- ➢ The next push would then occur in position 6



Increasing Capacity

An alternate solution is normalization:

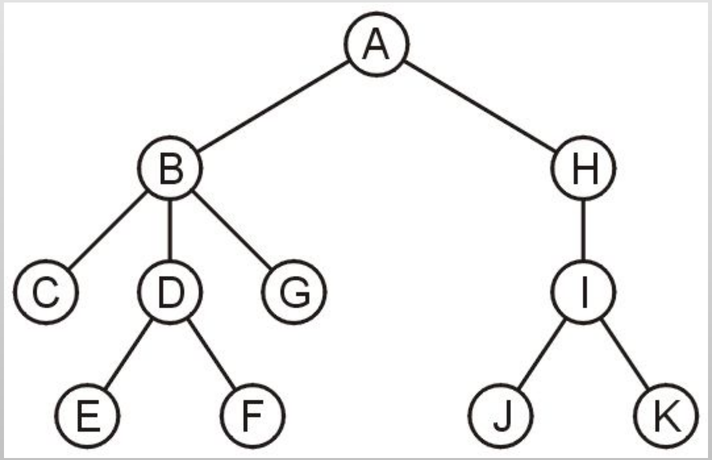
- ➢Map the front back at position 0
- ➢The next push would then occur in position 16



Application

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

- ➢ Consider searching the directory structure

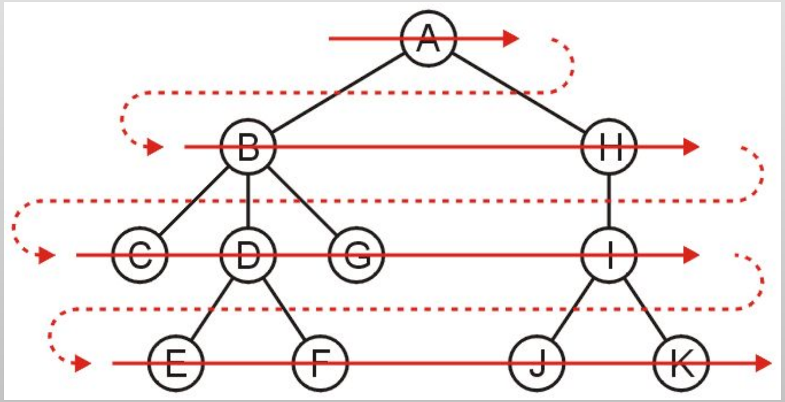


Application

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

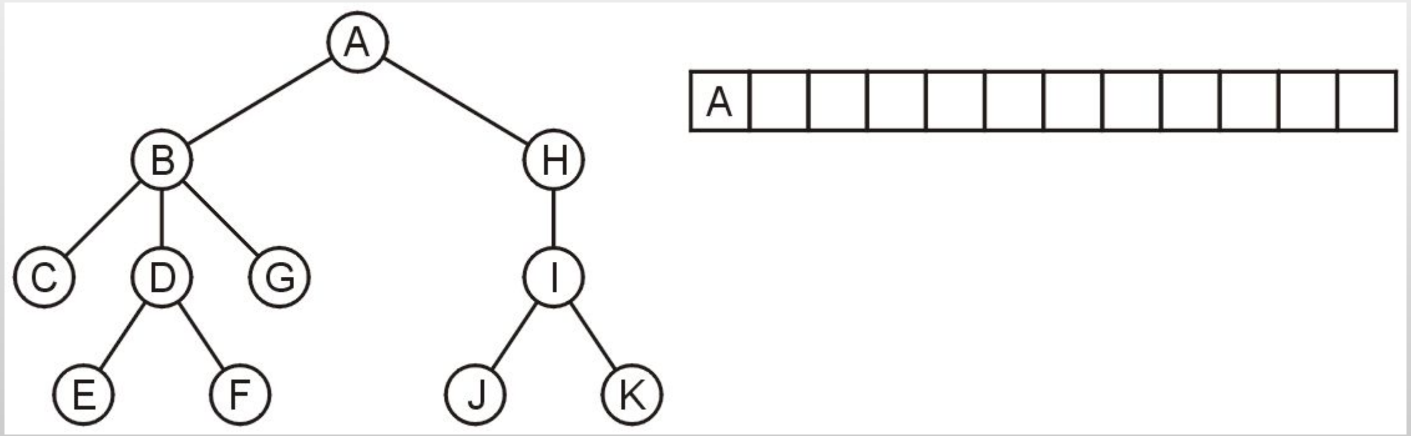
The easiest implementation is:

- Place the root directory into a queue
- While the queue is not empty:
  - 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

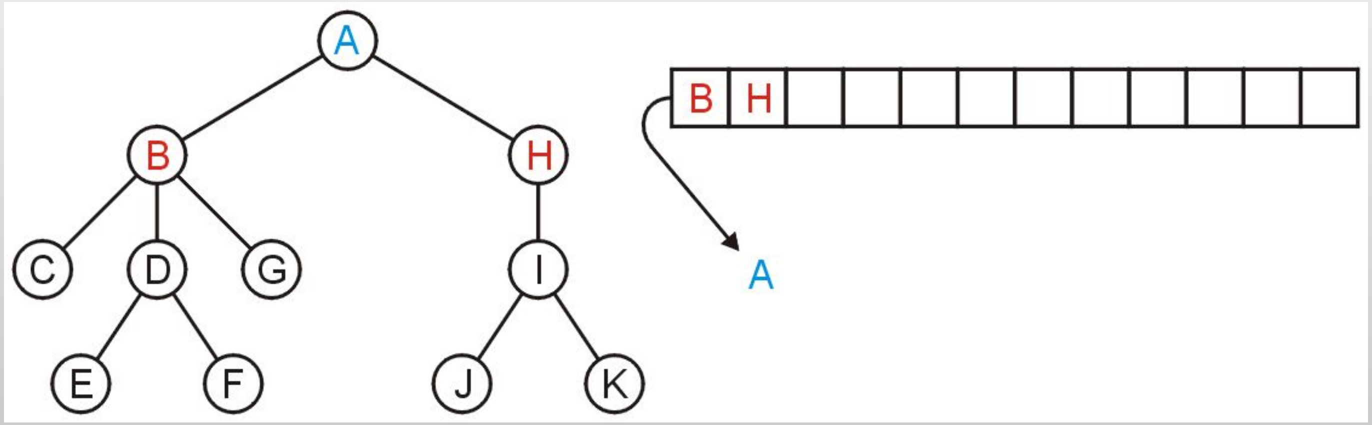
Application

Push the root directory A



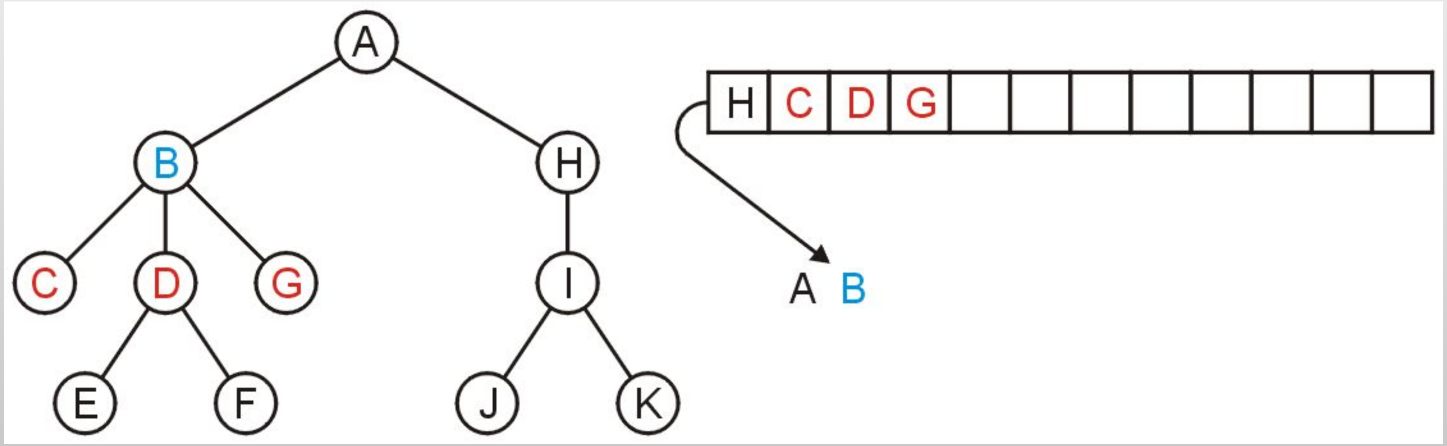
Application

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



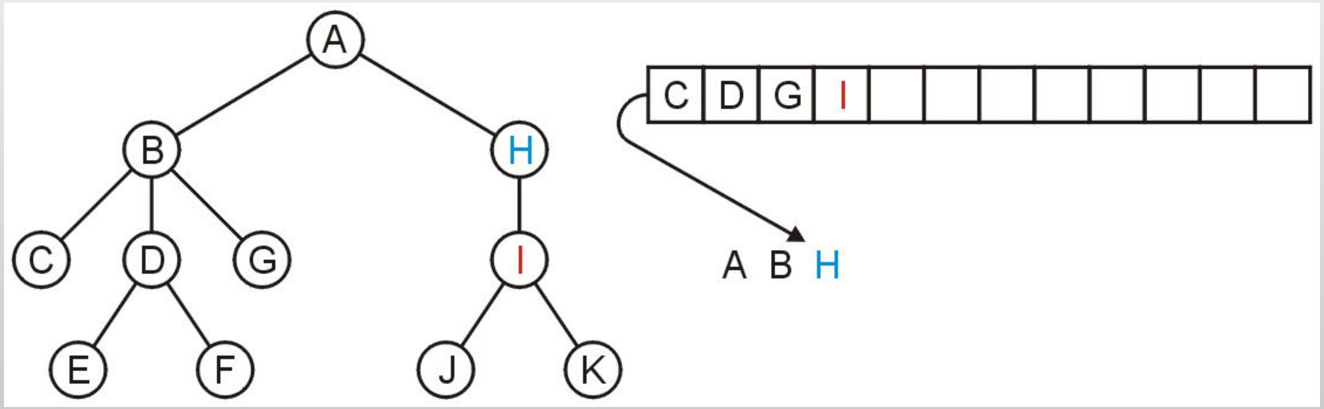
Application

Pop B and push C, D, and G



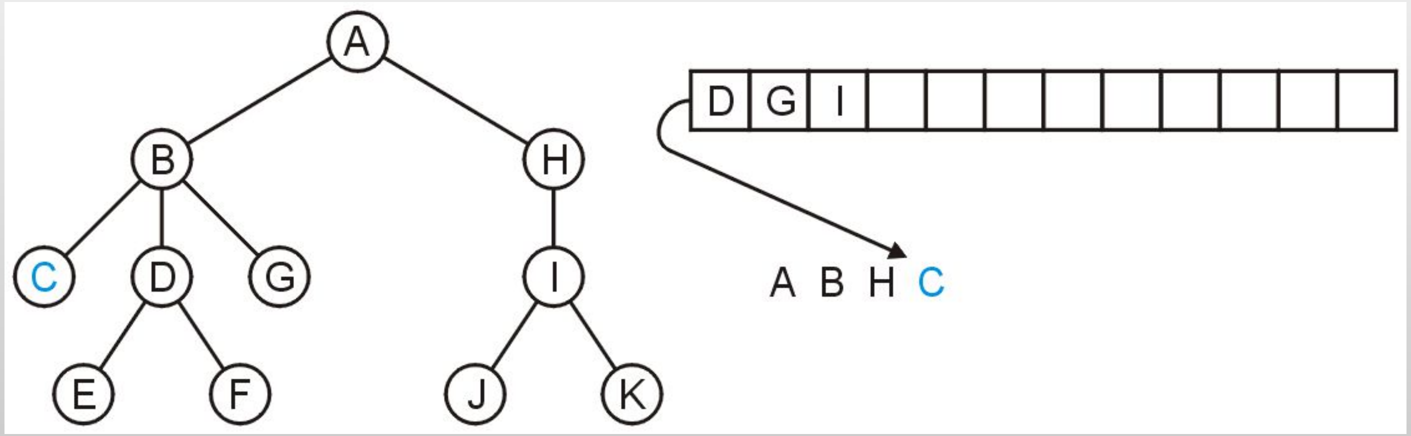
Application

Pop H and push its one sub-directory I



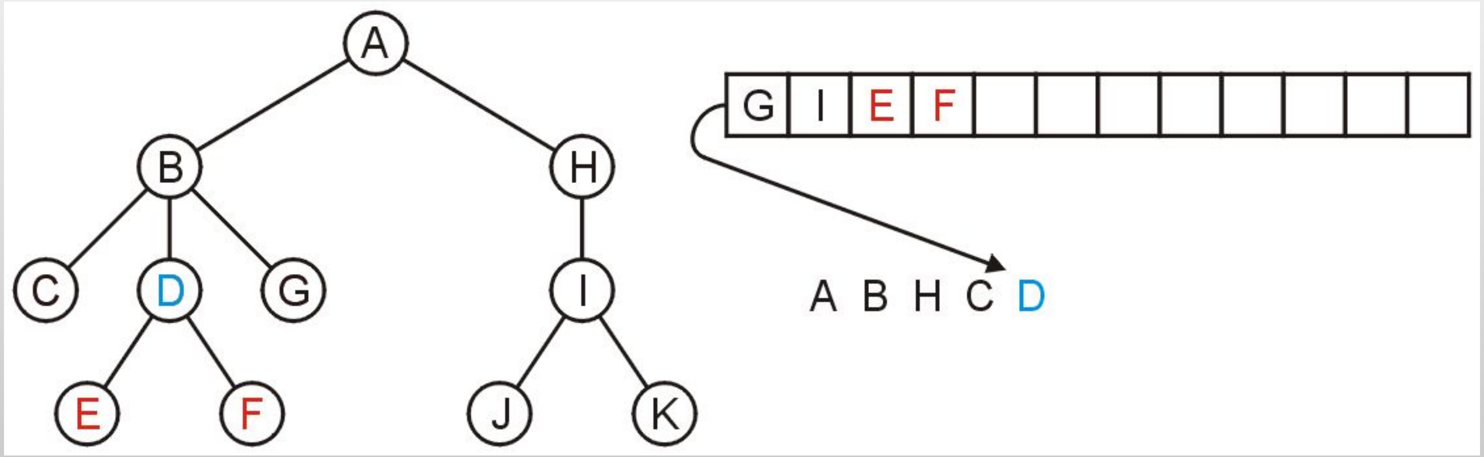
Application

Pop C: no sub-directories



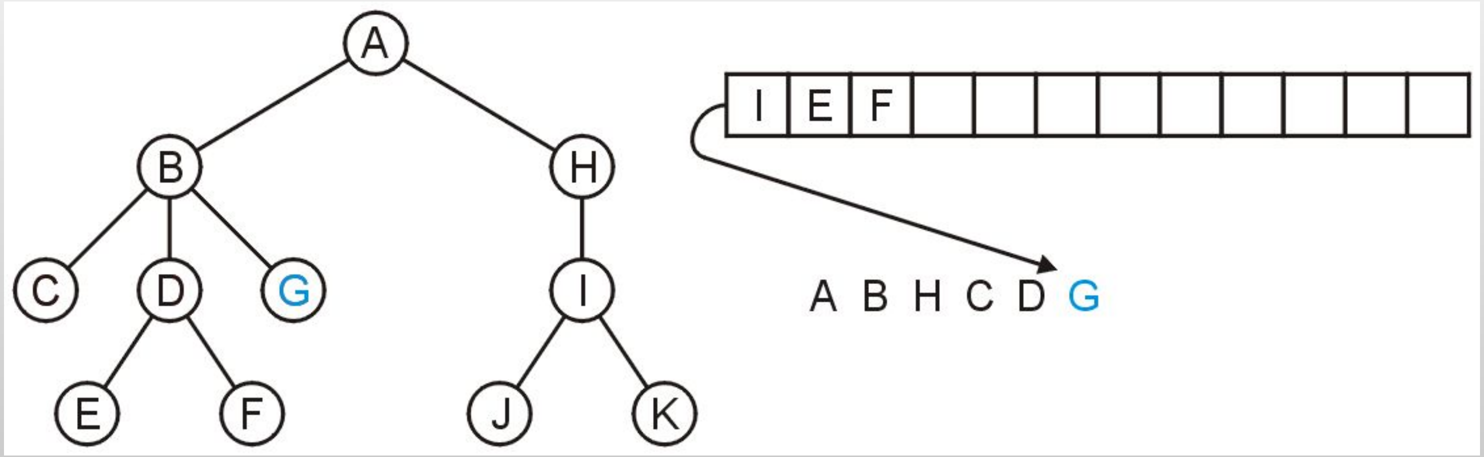
Application

Pop D and push E and F



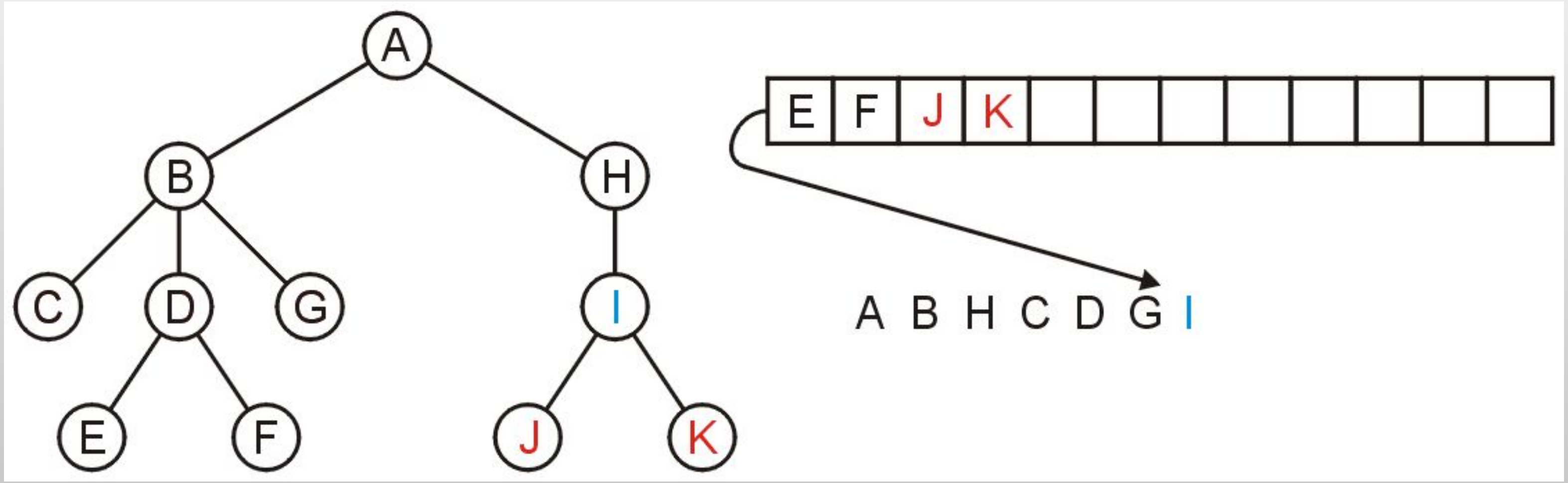
Application

Pop G



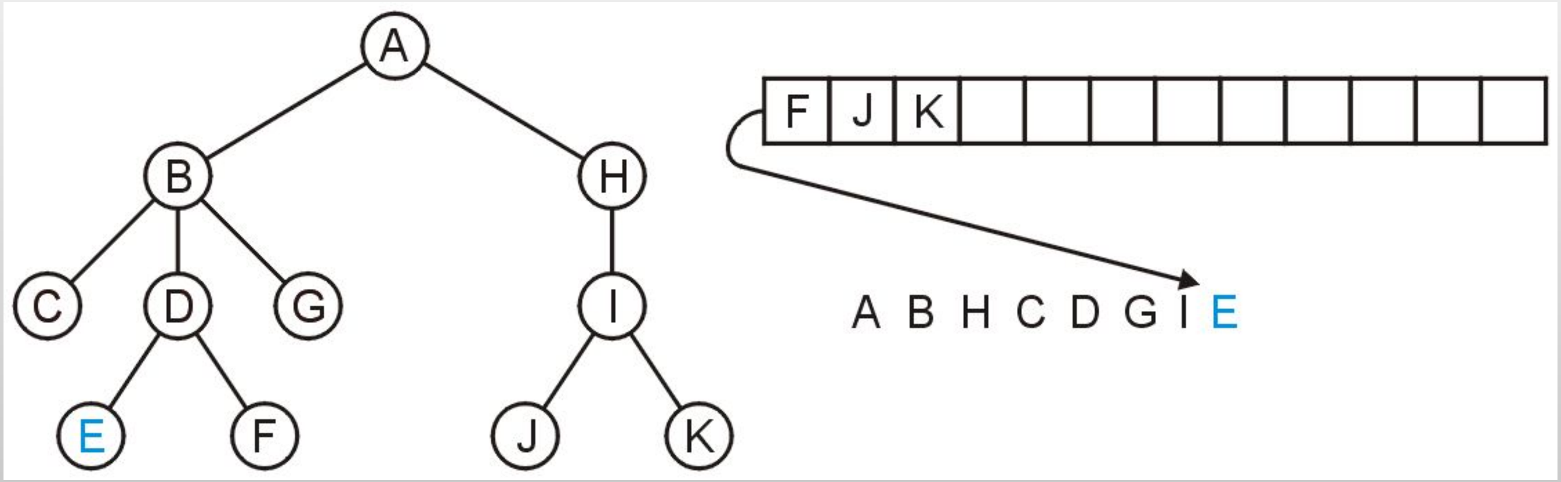
Application

Pop I and push J and K



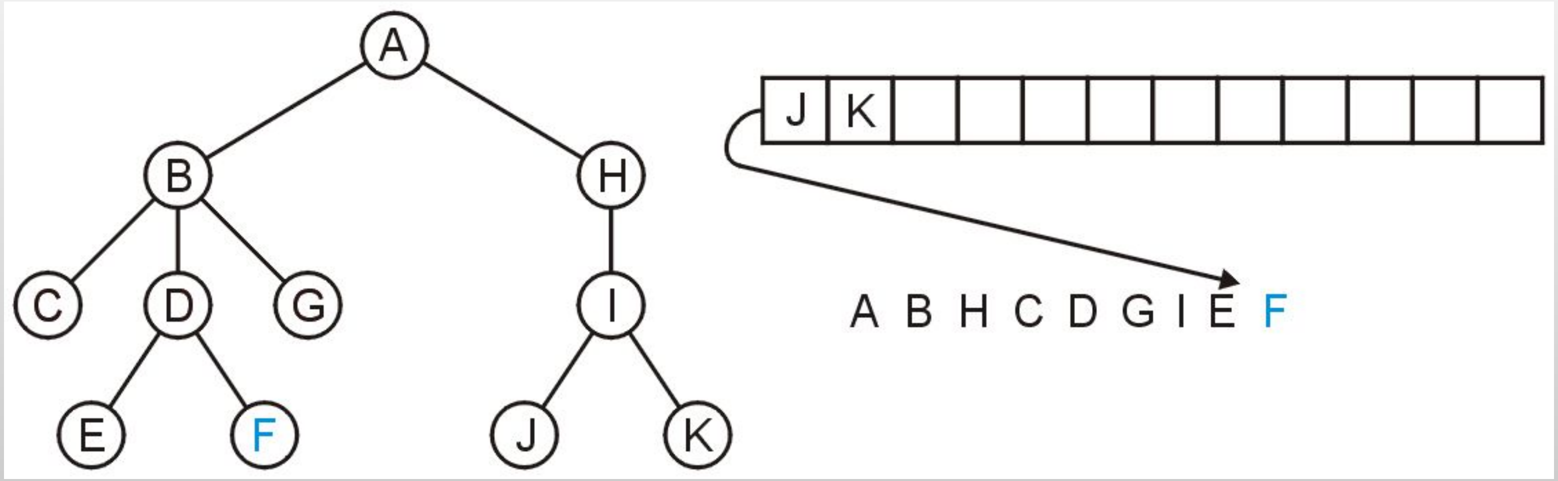
Application

Pop E



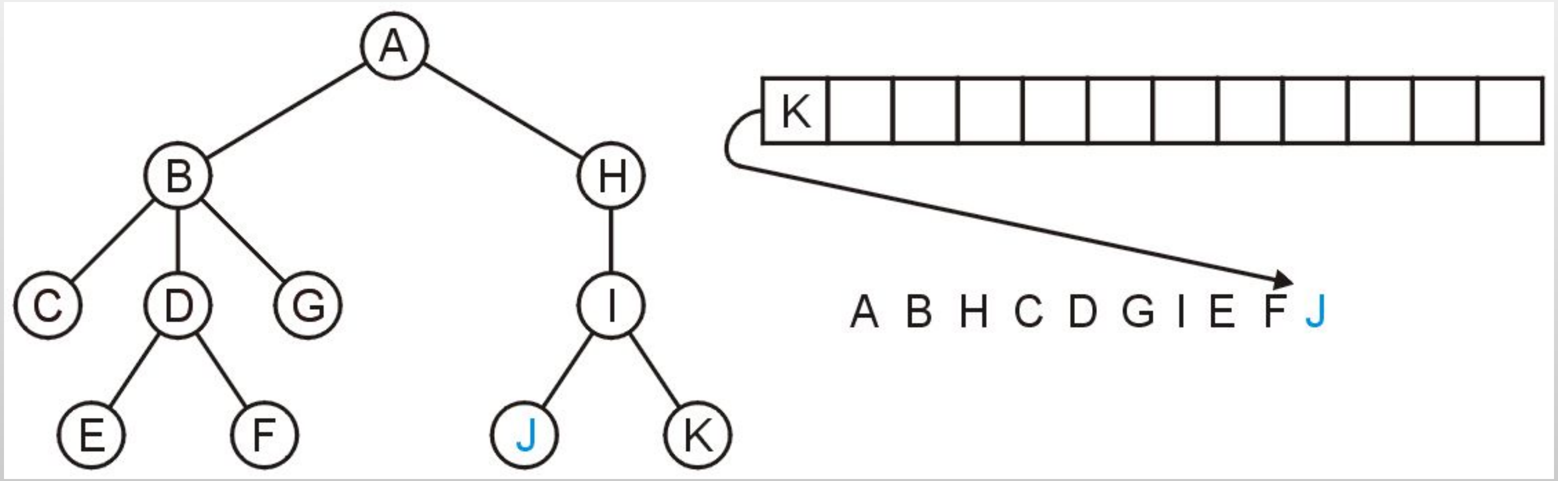
Application

Pop F



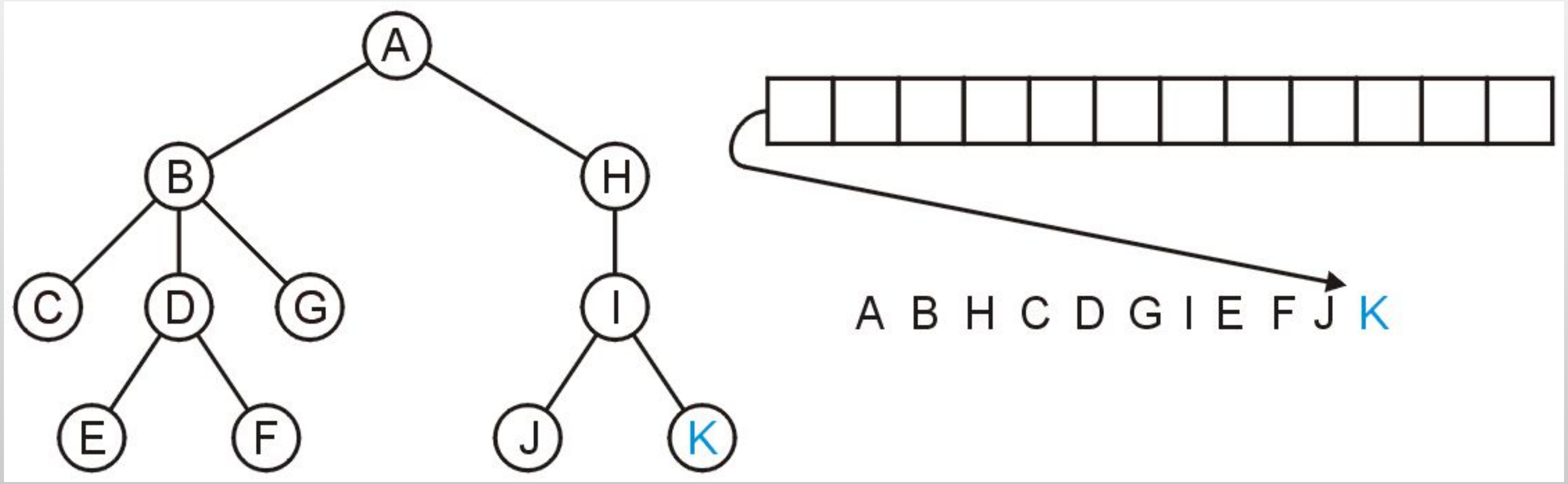
Application

Pop J



Application

Pop K and the queue is empty

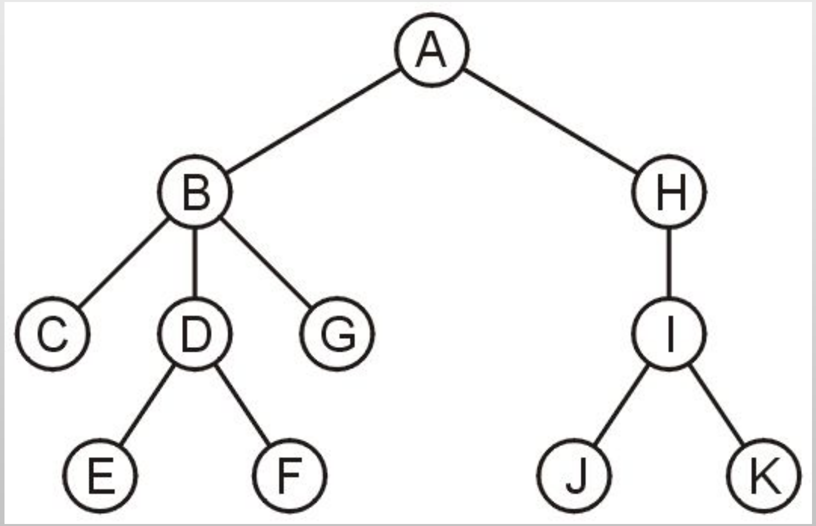


Application

The resulting order

A B H C D G I E F J K

is in breadth-first order:



STL Queue

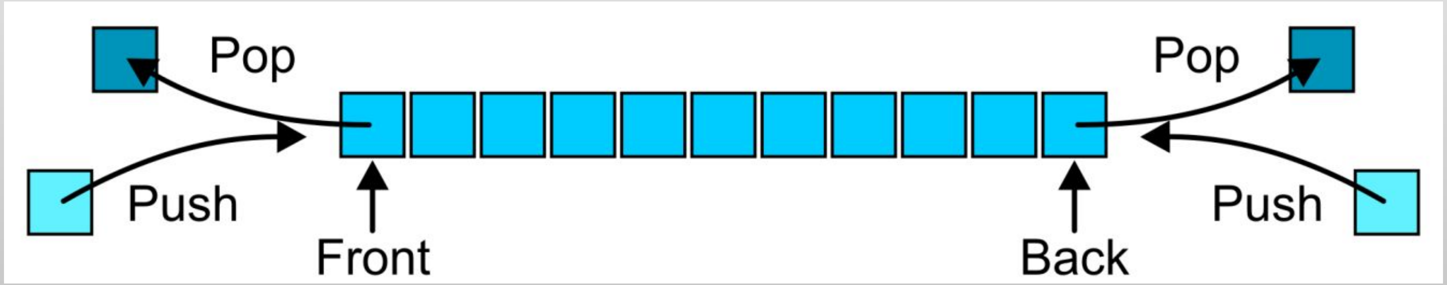
An example of a queue in the STL is:

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
    queue <int> iqueue;
    iqueue.push( 13 );
    iqueue.push( 42 );
    cout << "Head: " << iqueue.front() << endl;
    iqueue.pop();                           // no return value
    cout << "Head: " << iqueue.front() << endl;
    cout << "Size: " << iqueue.size() << endl;
    return 0;
}
```

Abstract Deque

An Abstract Deque (Deque ADT) is an abstract data structure which emphasizes specific operations:

- ➢ Uses an explicit linear ordering
- ➢ Insertions and removals are performed individually
- ➢ Allows insertions at both the front and back of the deque



Abstract Deque

The operations will be called

```
     front      back
push_front      push_back
 pop_front      pop_back
```

The implementations are clear:

- We must use either a doubly linked list or a circular array

Application

Useful as a general-purpose tool:

- ➢ Can be used as either a queue or a stack
- ➢ Palindrome Checker
- ➢ Undo-Redo operation in software applications

Problem solving:

- ➢ Consider solving a maze by adding or removing a constructed path at the front
- ➢ Once the solution is found, iterate from the back for the solution

Standard Template Library

```
#include <iostream>

#include <deque>

using namespace std;

int main() {

    deque<int> ideque;

    ideque.push_front( 5 );

    ideque.push_back( 4 );

    ideque.push_front( 3 );

    ideque.push_back( 6 );

 cout << "Is the deque empty?  " << ideque.empty() << endl;

 cout << "Size of deque:  " << ideque.size() << endl;

 for ( int i = 0; i < 4; ++i ) {

    cout << "Back of the deque:  "<< ideque.back() << endl;

    ideque.pop_back();

 }

 cout << "Is the deque empty?  " << ideque.empty() << endl;

 return 0;

}
```

OUTPUT

```
Is the deque empty?  0
Size of deque:  4
Back of the deque:  6
Back of the deque:  4
Back of the deque:  5
Back of the deque:  3
Is the deque empty?  1
```

Acknowledgement

Rafsanjany Kushol
PhD Student, Dept. of Computing Science,
University of Alberta

Sabbir Ahmed
Assistant Professor
Department of CSE, IUT