CS 2420: Queue

Dr. Tsung-Wei (TW) Huang

Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

Outline

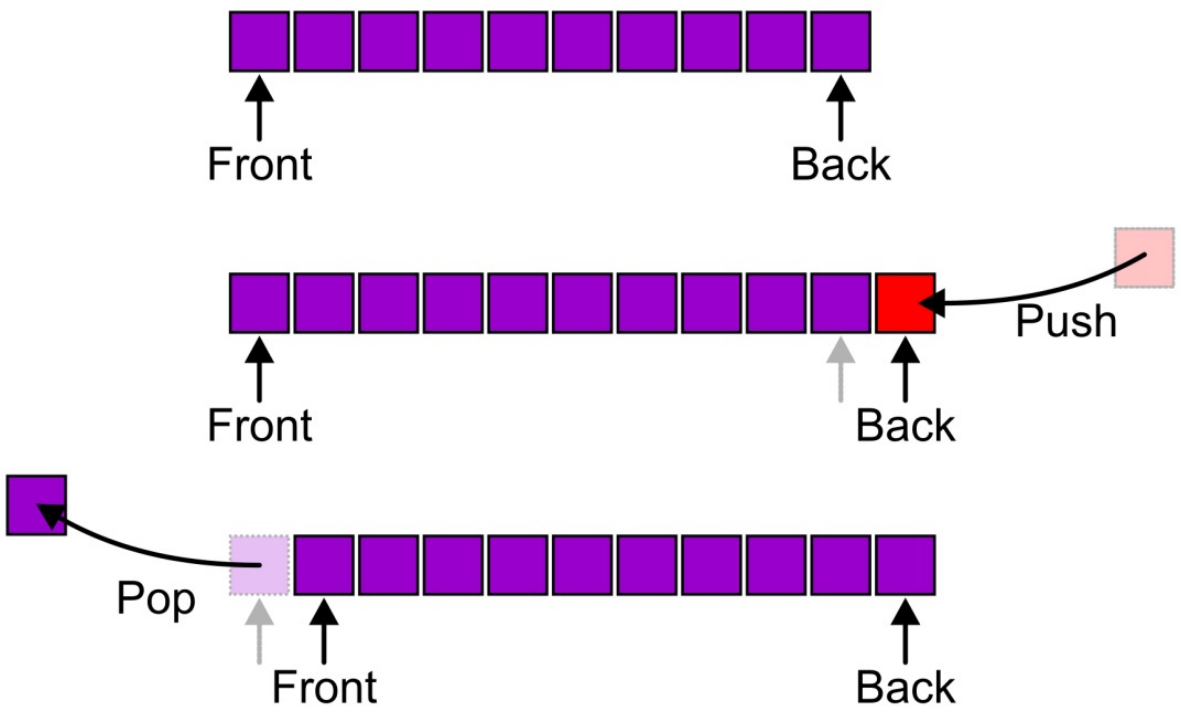
- This topic discusses the concept of a queue:
  - Description of an Abstract Queue
  - List applications
  - Implementation
  - Queuing theory
  - Standard Template Library

Abstract Queue

- An Abstract Queue (Queue ADT) is an abstract data type that emphasizes specific operations:
  - Uses a explicit linear ordering
  - Insertions and removals are performed individually
  - There are no restrictions on objects inserted into (*pushed onto*) the queue—that object is designated the back 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

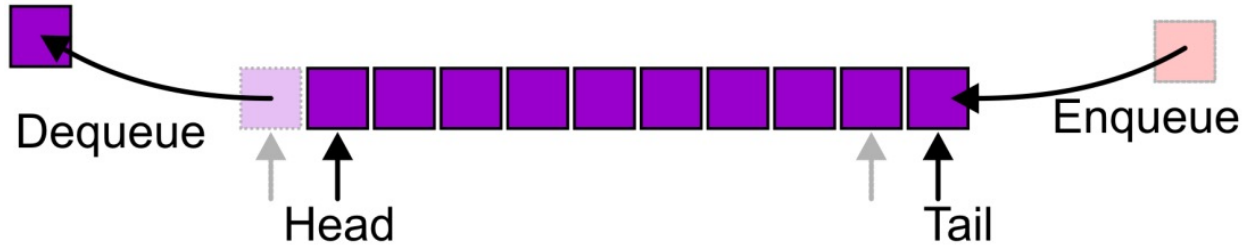
Queue Definition

- Also called a *First-in-first-out* (FIFO) behaviour
  - Graphically, we may view these operations as follows:



Queue Definition

- Alternative terms may be used for the four operations on a queue, including:
  - Dequeue: pops an element out of the queue
  - Enqueue: inserts an element into the queue



- There are two exceptions associated with this abstract data structure:
  - It is an undefined operation to call either pop (dequeue) or front (head) on an empty queue

Queue 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
- The SSH Secure Shell and SFTP are clients
- Most shared computer services are servers:
  - Web, file, ftp, database, mail, printers, WOW, *etc*.

Queue Implementation

We will look at two implementations of queues:

- Singly linked lists
- Circular arrays

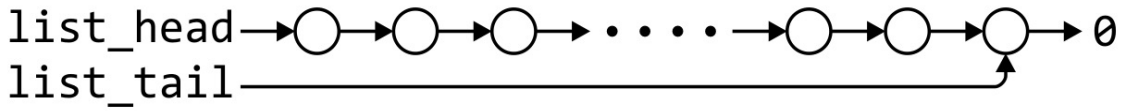
Requirements:

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

Linked-List Implementation

3.3.3.1

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



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

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

Single_list Definition

§3.3.1

The definition of single list class from Project 1 is:

```
template <typename Type>
class Single_list {
          public:
                    int size() const;
                    bool empty() const;
                    Type front() const;
                    Type back() const;
                    Single_node<Type> *head() const;
                    Single_node<Type> *tail() const;
                    int count( Type const & ) const;

                    void push_front( Type const & );
                    void push_back( Type const & );
                    Type pop_front();
                    int erase( Type const & );
};
```

Queue-as-List Class

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

```
template <typename Type>
class Queue{
    private:
        Single_list<Type> list;
    public:
        bool empty() const;
        Type front() const;
        void push( Type const & );
        Type pop();
};
```

Queue-as-List Class

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

```
template <typename Type>
bool Queue<Type>::empty() const {
     return list.empty();
}
```

```
template <typename Type>
void Queue<Type>::push( Type const &obj ) {
    list.push_back( obj );
}
```

```
template <typename Type>
Type Queue<Type>::front() const {
    if ( empty() ) {
        throw underflow();
    }

    return list.front();
}
```

```
template <typename Type>
Type Queue<Type>::pop() {
    if ( empty() ) {
        throw underflow();
    }

    return list.pop_front();
}
```

Array Implementation

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

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

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:
  - In C++, this is done by storing the address of the first entry

```
Type *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;
```

Queue-as-Array Class

- The class definition is similar to that of the Stack:

```
template <typename Type>
class Queue{
    private:
        int queue_size;
        int ifront;
        int iback;
        int array_capacity;
        Type *array;
    public:
        Queue( int = 10 );
        ~Queue();
        bool empty() const;
        Type front() const;
        void push( Type const & );
        Type pop();
};
```

Constructor

- Before we initialize the values, we will state that
  - `iback` is the index of the most-recently pushed object
  - `ifront` is the index of the object at the front of the queue

- To push, we will increment `iback` and place the new item at that location
  - To make sense of this, we will initialize

```
iback = -1;
ifront = 0;
```

  - After the first push, we will increment `iback` to `0`, place the pushed item at that location, and now

Constructor

- Again, we must initialize the values
  - We must allocate memory for the array and initialize the members
  - The call to `new Type[array_capacity]` makes a request to the operating system for `array_capacity` objects

```
#include <algorithm>
// ...
template <typename Type>
Queue<Type>::Queue( int n ):
queue_size( 0 ),
iback( -1 ),
ifront( 0 ),
array_capacity( std::max(1, n) ),
array( new Type[array_capacity] ) {
    // Empty constructor
}
```

Constructor

- Reminder:
  - Initialization is performed in the order specified in the class declaration

```
template <typename Type>
Queue<Type>::Queue( int n ):
queue_size( 0 ),
iback( -1 ),
ifront( 0 ),
array_capacity( std::max(1, n) ),
array( new Type[array_capacity] )
{
    // Empty constructor
}
```

```
template <typename Type>
class Queue {
    private:
        int queue_size;
        int iback;
        int ifront;
        int array_capacity;
        Type *array;
    public:
        Queue( int = 10 );
        ~Queue();
        bool empty() const;
        Type top() const;
        void push( Type const & );
        Type pop();
};
```

Destructor

- The destructor is unchanged from Stack-as-Array:

```
template <typename Type>
Queue<Type>::~Queue() {
    delete [] array;
}
```

Member Functions

- These two functions are similar in behaviour:

```
template <typename Type>
bool Queue<Type>::empty() const {
    return ( queue_size == 0 );
}
```

```
template <typename Type>
Type Queue<Type>::front() const {
    if ( empty() ) {
        throw underflow();
    }

    return array[ifront];
}
```

Member Functions

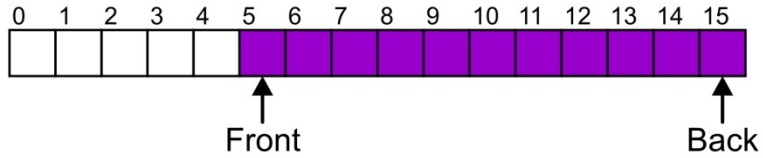
- However, a naïve implementation of push and pop will cause difficulties:

```
template <typename Type>
void Queue<Type>::push( Type const &obj ) {
    if ( queue_size == array_capacity ) {
        throw overflow();
    }

    ++iback;
    array[iback] = obj;
    ++queue_size;
}
```

```
template <typename Type>
Type Queue<Type>::pop() {
    if ( empty() ) {
        throw underflow();
    }

    --queue_size;
    ++ifront;
    return array[ifront - 1];
}
```

Member Functions

- 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

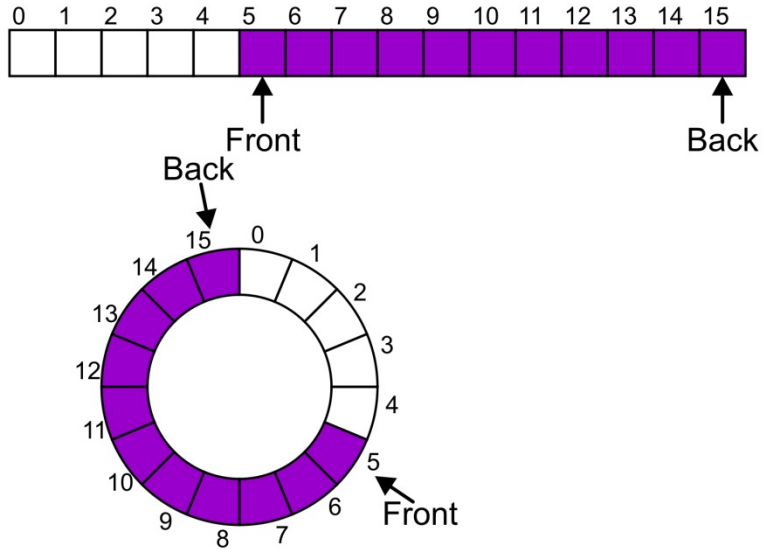
- In this case, the array is not full and yet we cannot place any more objects in to the array

Member Functions

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

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

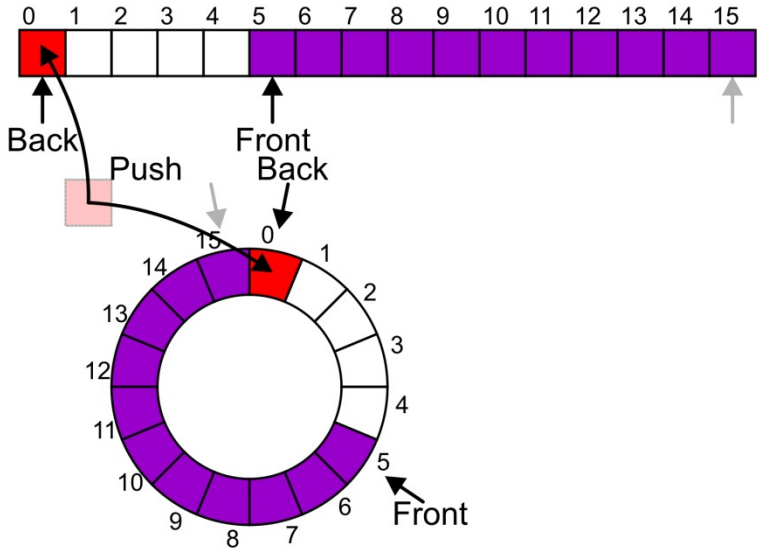
- This is referred to as a *circular array*



Member Functions

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

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



Exceptions

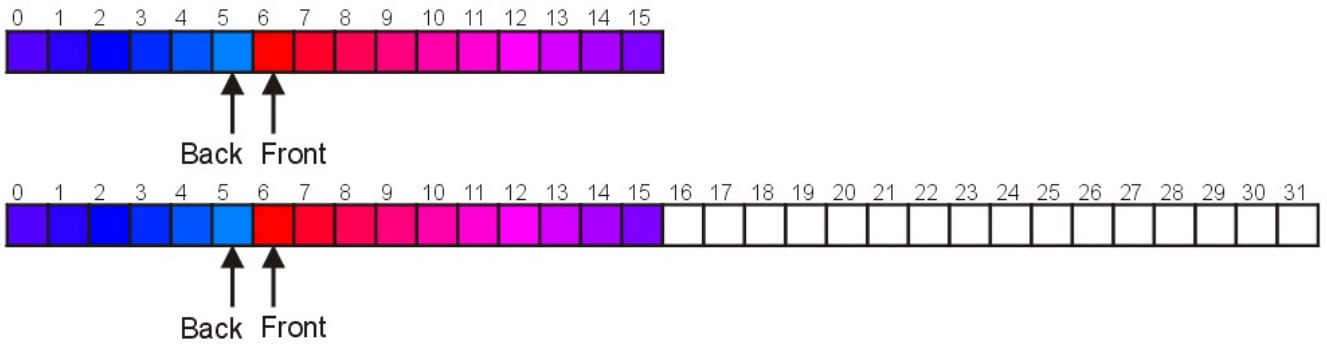
- As with a stack, there are a number of options which can be used if the array is filled

- 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()`**

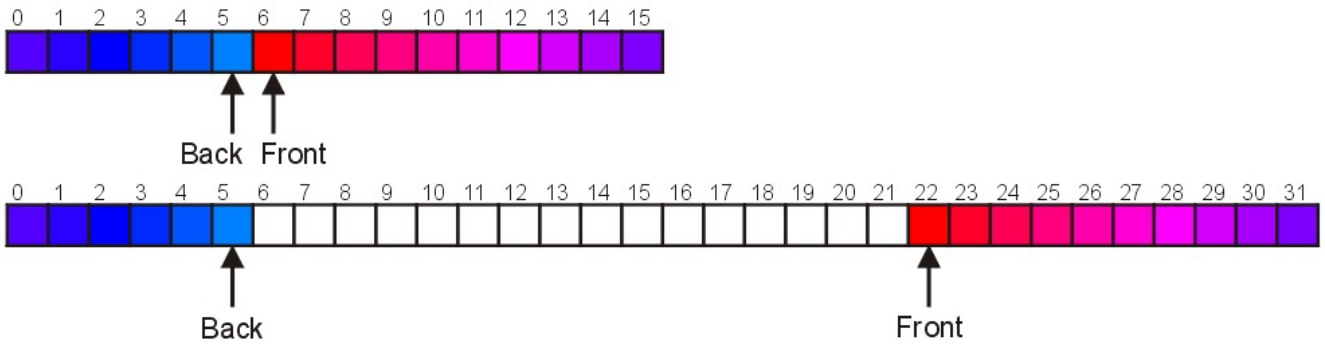
Increasing Capacity

- Unfortunately, if we choose to increase the capacity, this becomes slightly more complex
  - A direct copy does not work:



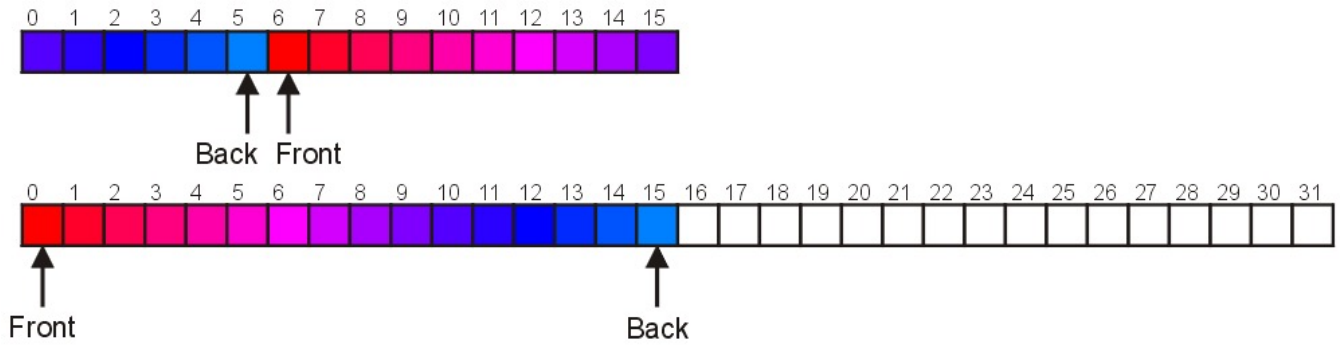
Increasing Capacity

- There are two solutions:
  - Move those beyond the front to the end of the array
  - The next push would then occur in position 6



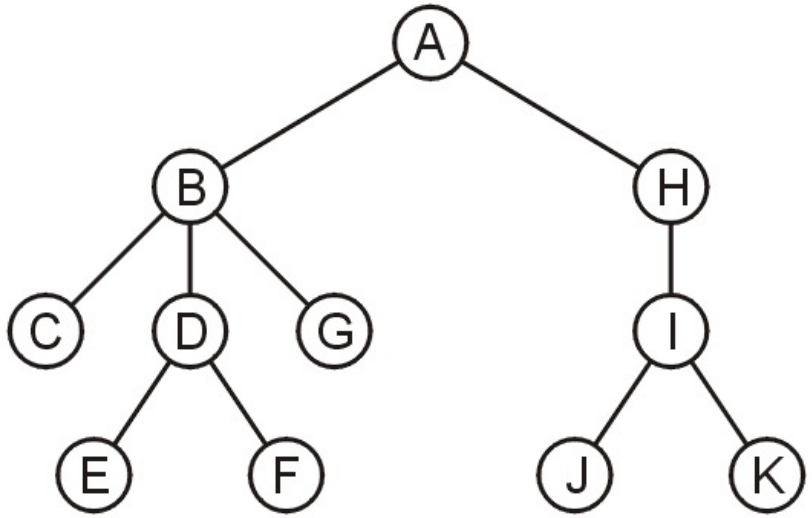
Increasing Capacity

- An alternate solution is normalization:
  - Map the front back at position 0
  - The next push would then occur in position 16



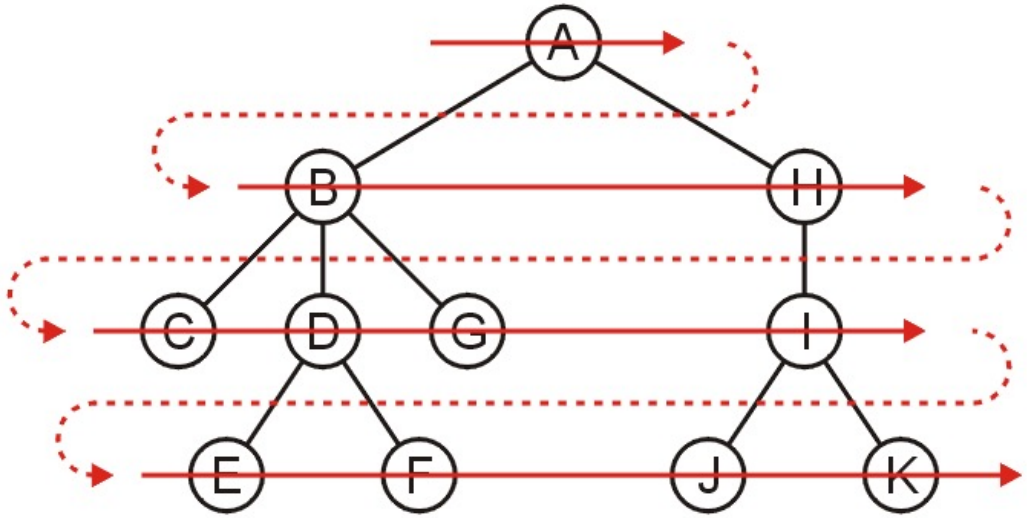
Application

- Another application is performing a breadth-first traversal of a directory tree
  - Consider searching the directory structure



Application

- We would 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



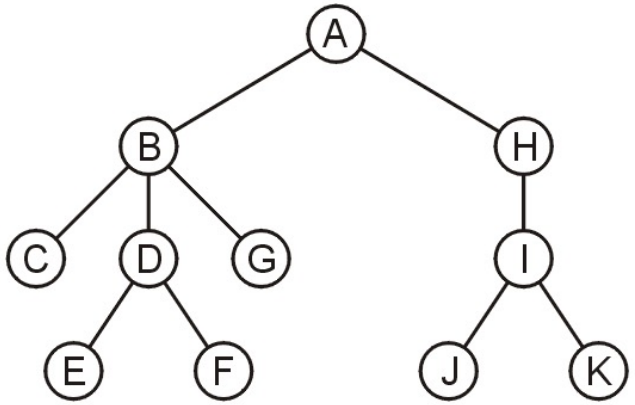
Application

- 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

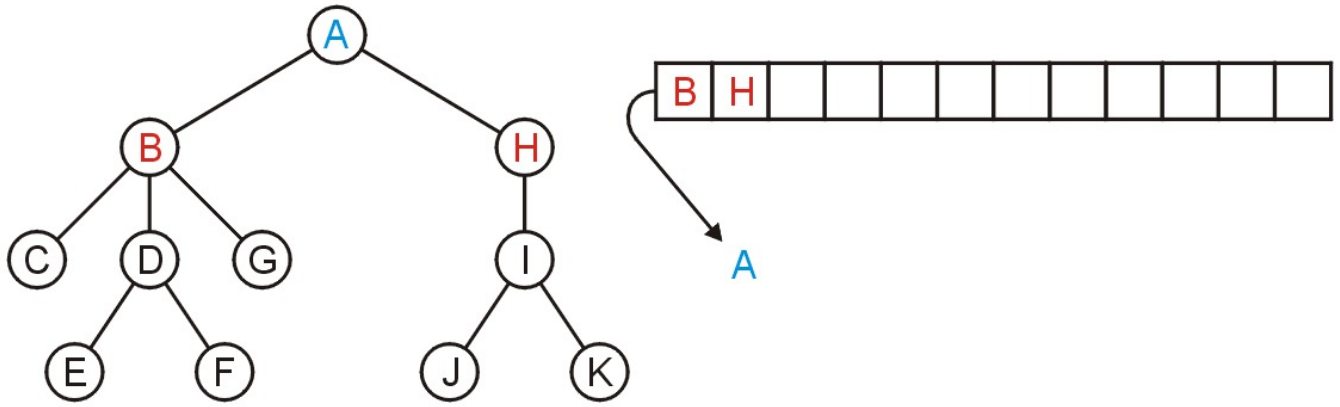
Application

- Push the root directory A



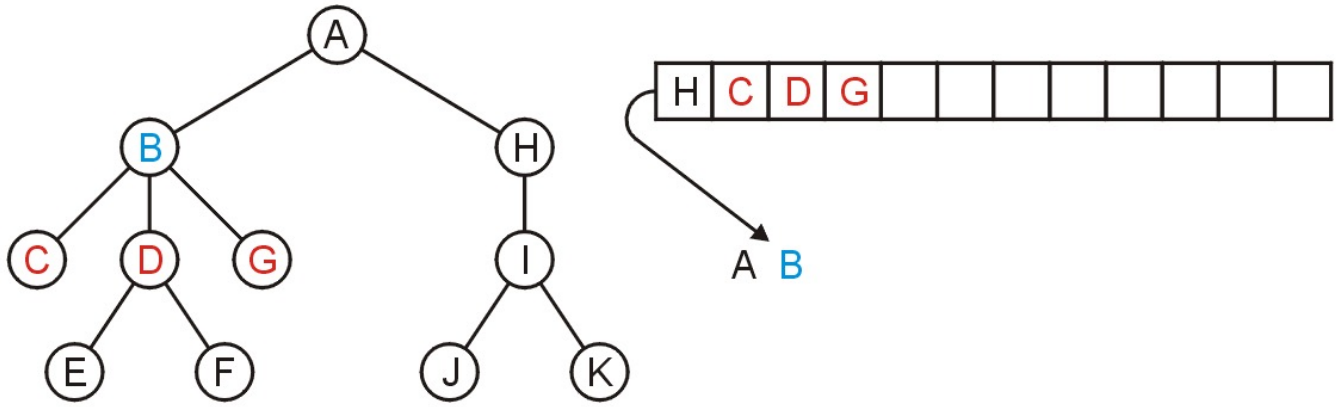
Application

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



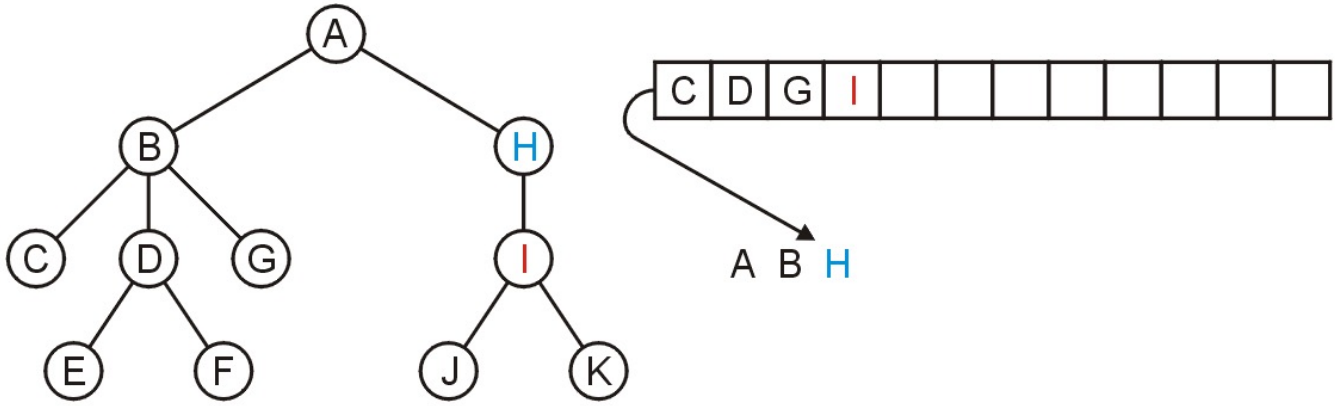
Application

- Pop B and push C, D, and G



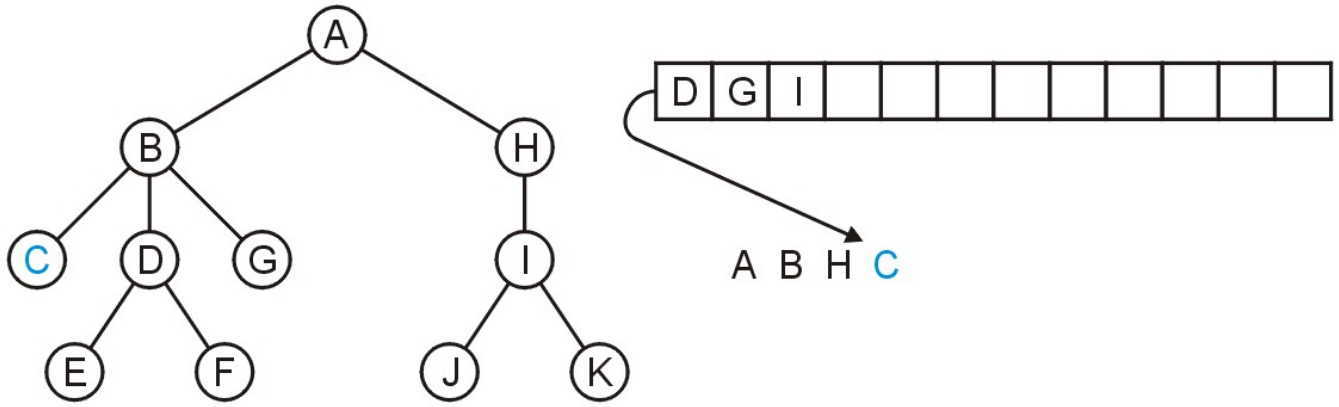
Application

- Pop H and push its one sub-directory I



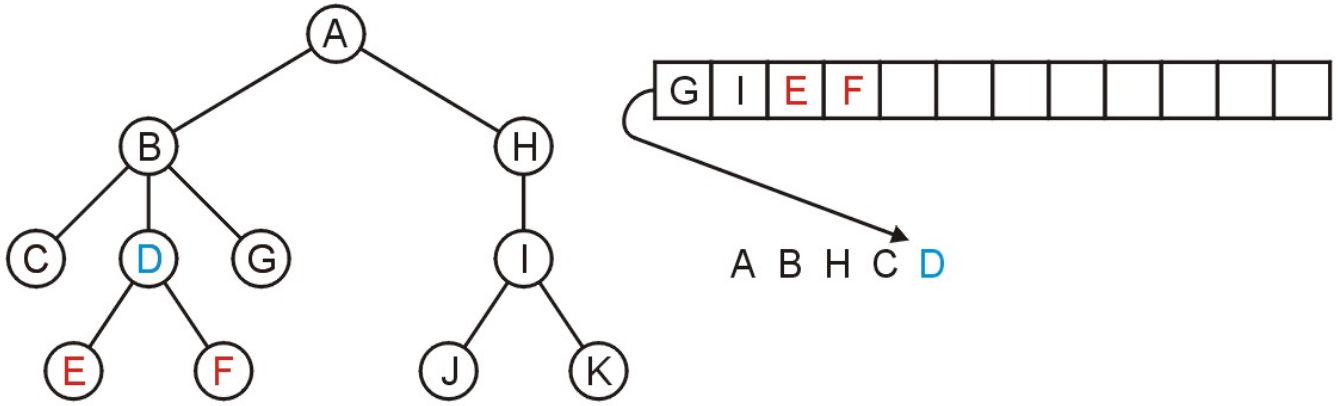
Application

- Pop C: no sub-directories



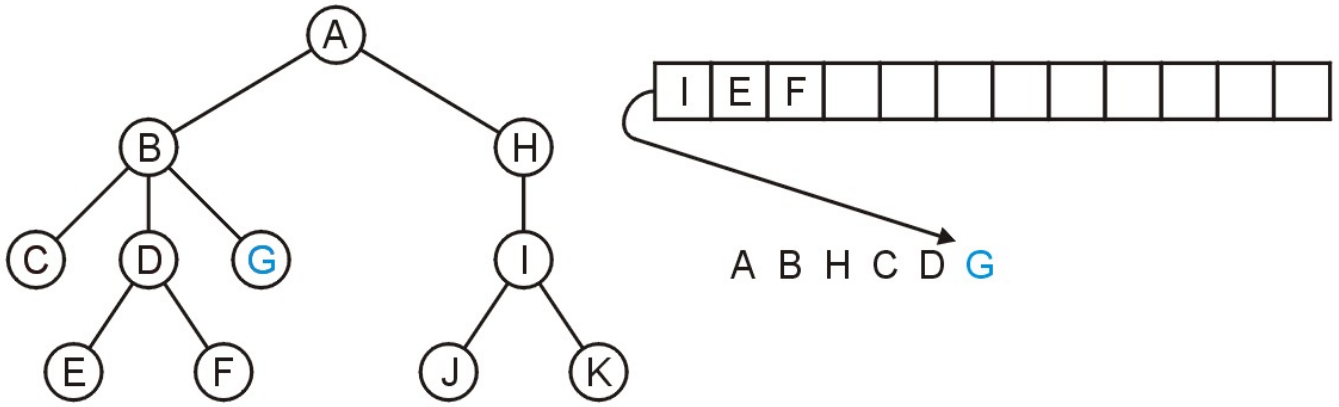
Application

- Pop D and push E and F



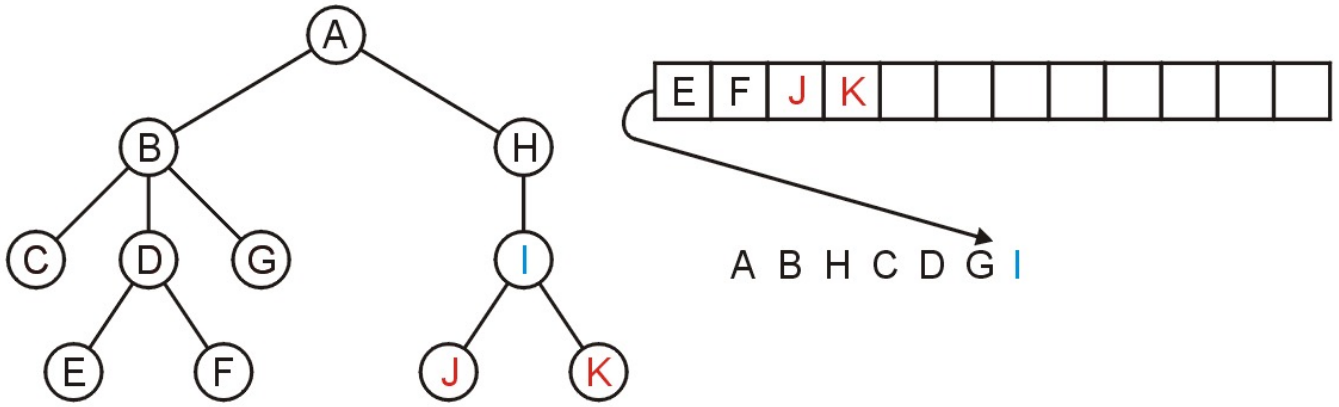
Application

- Pop G



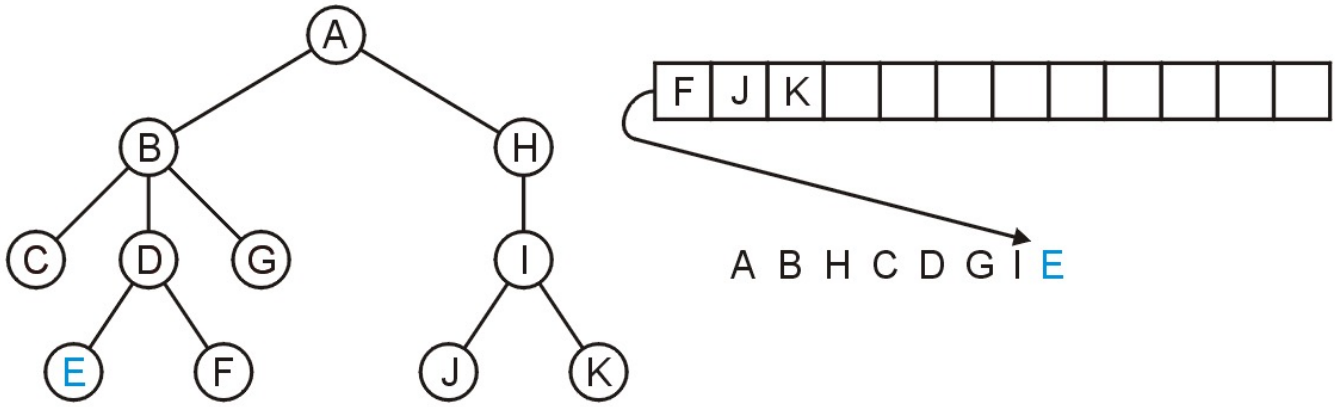
Application

- Pop I and push J and K



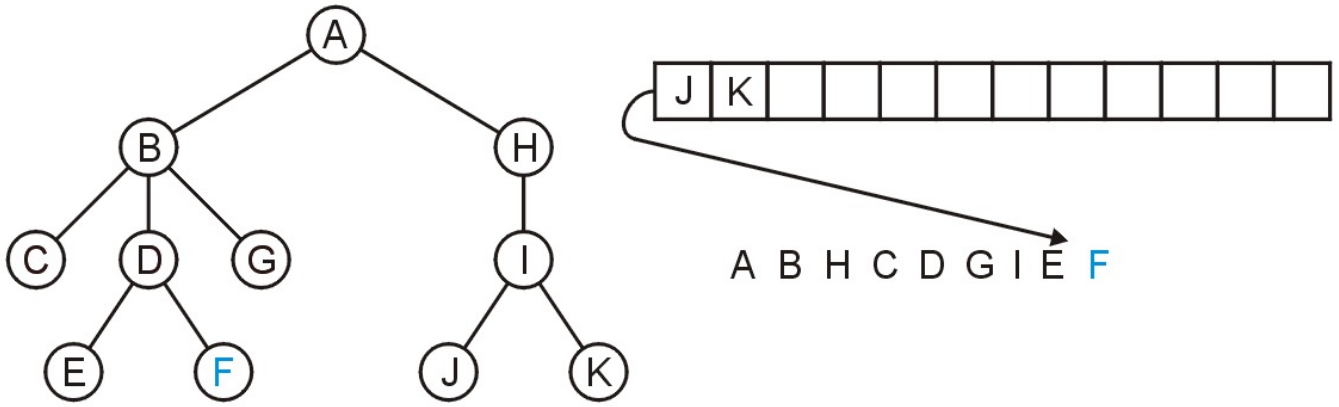
Application

- Pop E



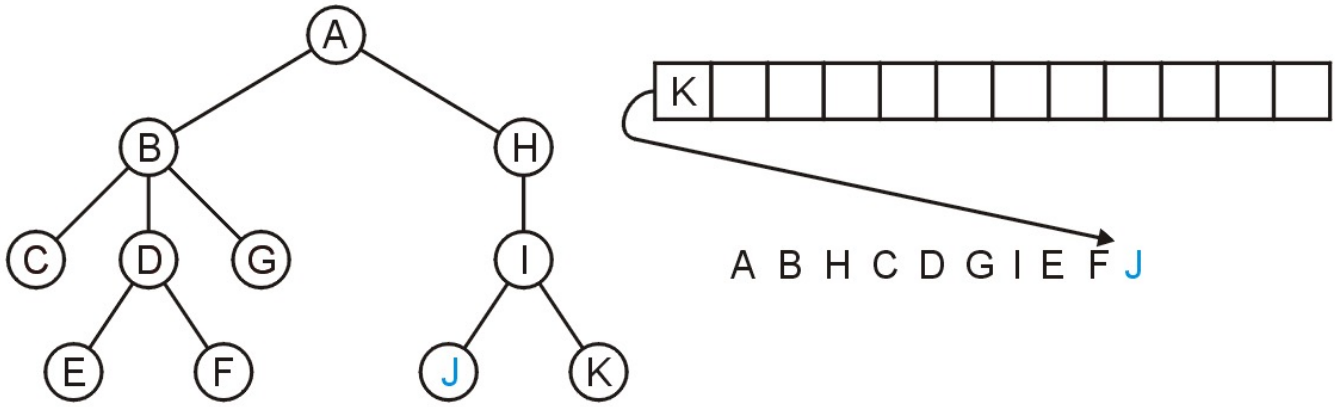
Application

- Pop F



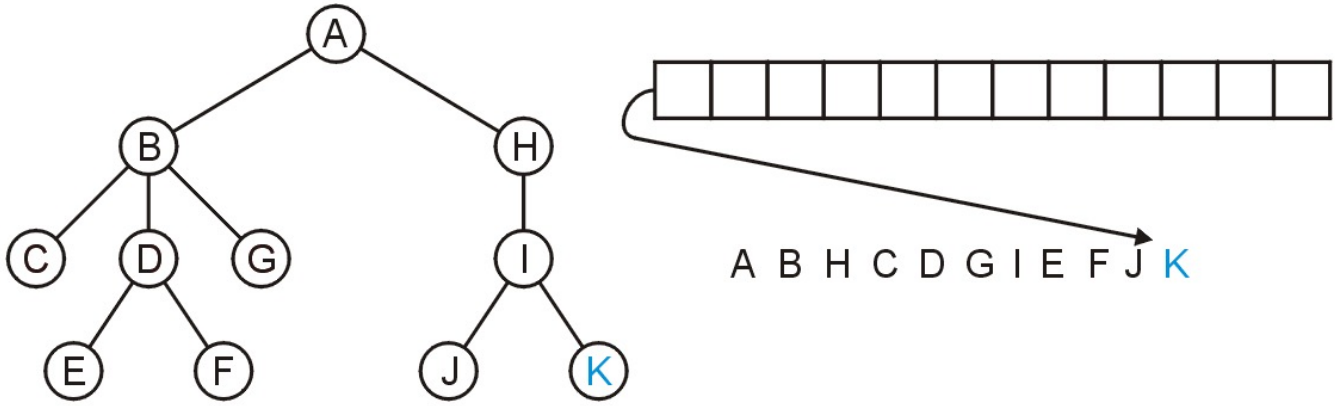
Application

- Pop J



Application

- Pop K and the queue is empty

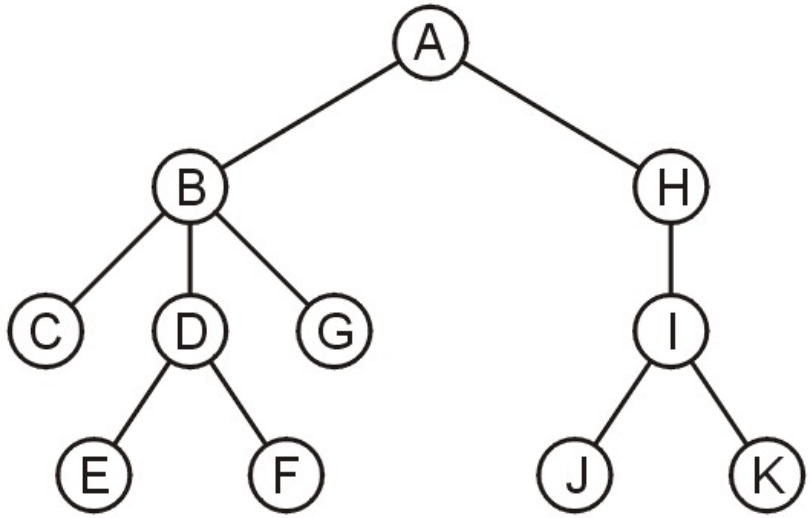


Application

- The resulting order

  A B H C D G I E F J K

  is in breadth-first order:



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

- **Midterm will start next week (10/25-11/1) – We will still have lectures**

LAB: Standard Template Library “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;
}
```