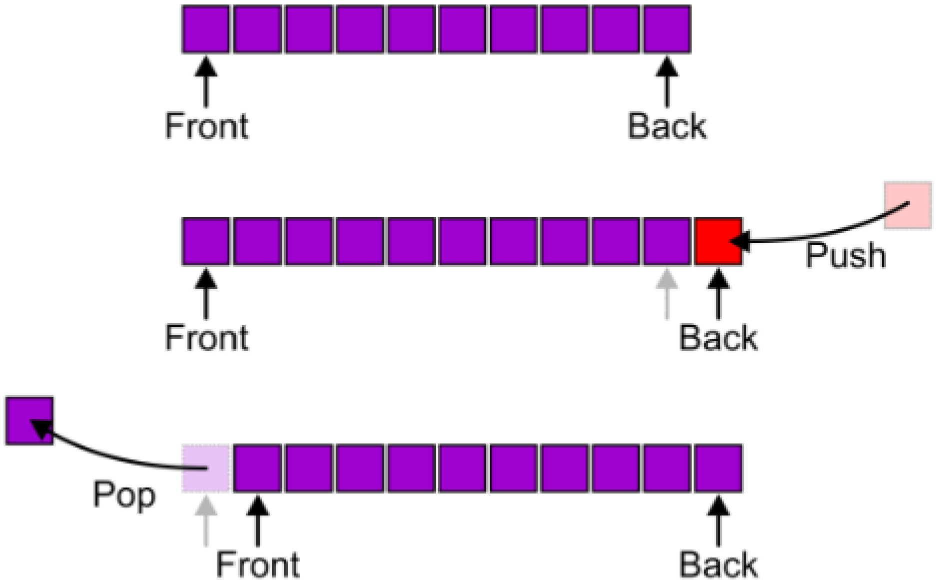
Queues & Deques

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

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



- Alternative terms may be used for the four operations on a queue, including:
    - queue & dequeue
    - head & tail

Applications

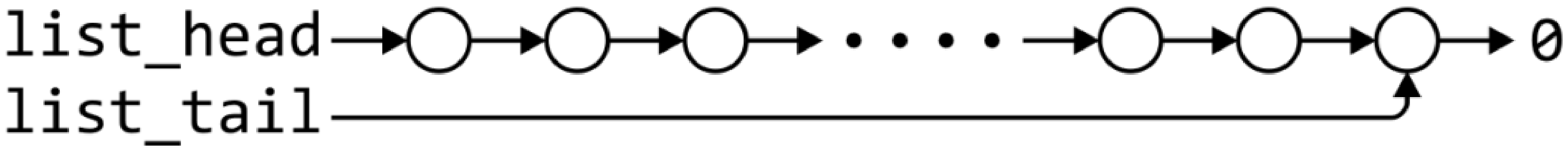
- The most common application is in client-server models
  - Multiple clients may be requesting services from one or more servers
  - Some clients may have to wait while the servers are busy
  - Those clients are placed in a queue and serviced in the order of arrival
- Grocery stores, banks, and airport security use queues
- Most shared computer services are servers:
  - Web, file, ftp, ssh, database, mail, printers, 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

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



	front / 1st node	back / nth node
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

`LinkedList` Definition

`LinkedList` Definition

In [3]:

```
template <typename Type>
class LinkedList {
private:
    SinglyLinkedNode<Type> *list_head;
    SinglyLinkedNode<Type> *list_tail;

public:
    LinkedList();
    ~LinkedList();

    // Accessors
    bool empty() const;
    Type front() const;

    // Mutators
    void push_front( const Type & );
    void push_back( const Type & );
    Type pop_front();
};
```

Queue-as-List Class

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

Queue-as-List Class

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

In [4]:

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

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

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

In [5]:

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

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

In [5]:

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

In [6]:

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

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

In [5]:

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

In [6]:

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

In [7]:

```
template <typename Type>
Type LinkedQueue<Type>::front() const {
    if ( empty() ) {
        throw std::underflow_error("Queue is empty");
    }
    return list.front();
}
```

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

In [5]:

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

In [6]:

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

In [7]:

```
template <typename Type>
Type LinkedQueue<Type>::front() const {
    if ( empty() ) {
        throw std::underflow_error("Queue is empty");
    }
    return list.front();
}
```

In [8]:

```
template <typename Type>
Type LinkedQueue<Type>::pop() {
    if ( empty() ) {
        throw std::underflow_error("Queue is empty");
    }
    return list.pop_front();
}
```

Array Implementation

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

- With asymptotic analysis of array lists, we can now make the following statements:

	front / 1st node	**back / nth node**
find	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(n)$	$\Theta(1)$
erase	$\Theta(n)$	$\Theta(1)$

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

- With asymptotic analysis of two-ended array lists, we can now make the following statements:

	front / 1st node	back / nth node
find	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(1)$	$\Theta(1)$
erase	$\Theta(1)$	$\Theta(1)$

Design

- We need to store an array:
  - In C++, this is done by storing the address of the first entry

```
Type *array;
```

- The number of objects currently in the queue and the front and back indexes
  - The number of objects currently in the stack

```
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 `ArrayStack`

Queue-as-Array Class

- The class definition is similar to that of the `ArrayStack`

In [10]:

```
template <typename Type>
class ArrayQueue {
    private:
        int queue_size;
        int ifront;
        int iback;
        int array_capacity;
        Type *array;
    public:
        ArrayQueue( int = 10 );
        ~ArrayQueue();
        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

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

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

In [12]:

```
#include <algorithm>

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

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

In [12]:

```
#include <algorithm>

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

- **Note:** Initialization is performed in the order specified in the class declaration

Destructor

- The destructor is unchanged from `ArrayStack`

Destructor

- The destructor is unchanged from `ArrayStack`

In [13]:

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

Member Functions

- These two functions are similar in behavior as in `ArrayStack` class

Member Functions

- These two functions are similar in behavior as in `ArrayStack` class

In [14]:

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

Member Functions

- These two functions are similar in behavior as in `ArrayStack` class

In [14]:

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

In [16]:

```
template <typename Type>
Type ArrayQueue<Type>::front() const {
    if ( empty() ) {
        throw std::underflow_error("Queue is empty");
    }
    return array[ifront];
}
```

- However, a naïve implementation of `push` and `pop` may cause difficulties

- However, a naïve implementation of `push` and `pop` may cause difficulties

In [18]:

```
template <typename Type>
void ArrayQueue<Type>::push( Type const &obj ) {
    if ( queue_size == array_capacity ) {
        throw std::overflow_error("Queue is full");
    }
    ++iback;
    array[iback] = obj;
    ++queue_size;
}
```

- However, a naïve implementation of `push` and `pop` may cause difficulties

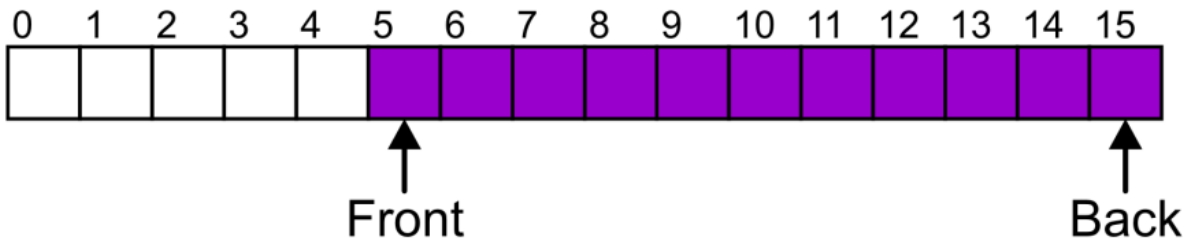
In [18]:

```
template <typename Type>
void ArrayQueue<Type>::push( Type const &obj ) {
    if ( queue_size == array_capacity ) {
        throw std::overflow_error("Queue is full");
    }
    ++iback;
    array[iback] = obj;
    ++queue_size;
}
```

In [19]:

```
template <typename Type>
Type ArrayQueue<Type>::pop() {
    if ( empty() ) {
        throw std::underflow_error("Queue is empty");
    }
    --queue_size;
    ++ifront;
    return array[ifront - 1];
}
```

- 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

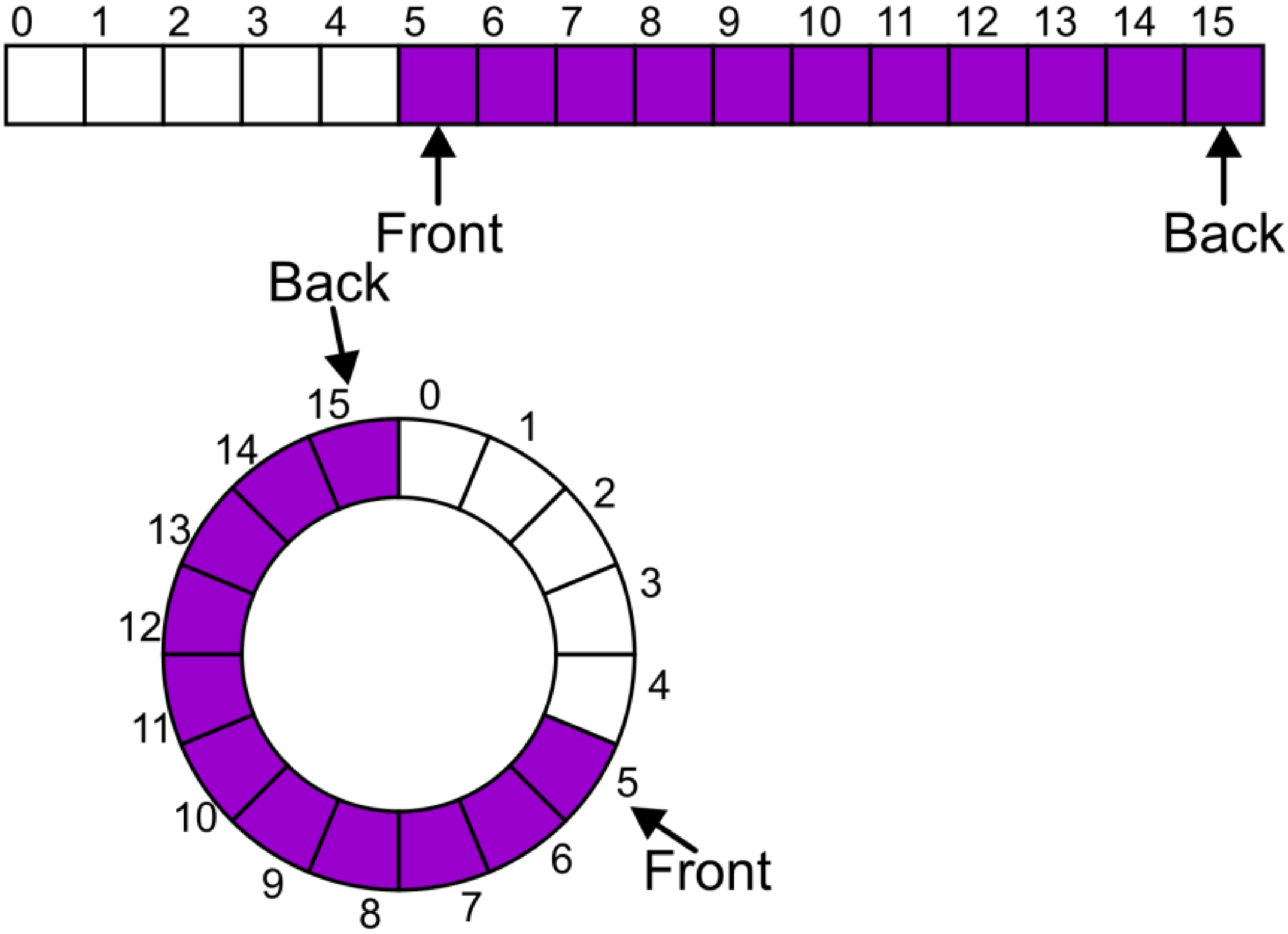
- Instead of viewing the array on the range $0, \ldots, 15$, consider the indexes being cyclic

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

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

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

- This is referred to as a circular array


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Front
Back
Back
15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Front

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

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

- or using modular arithmetic

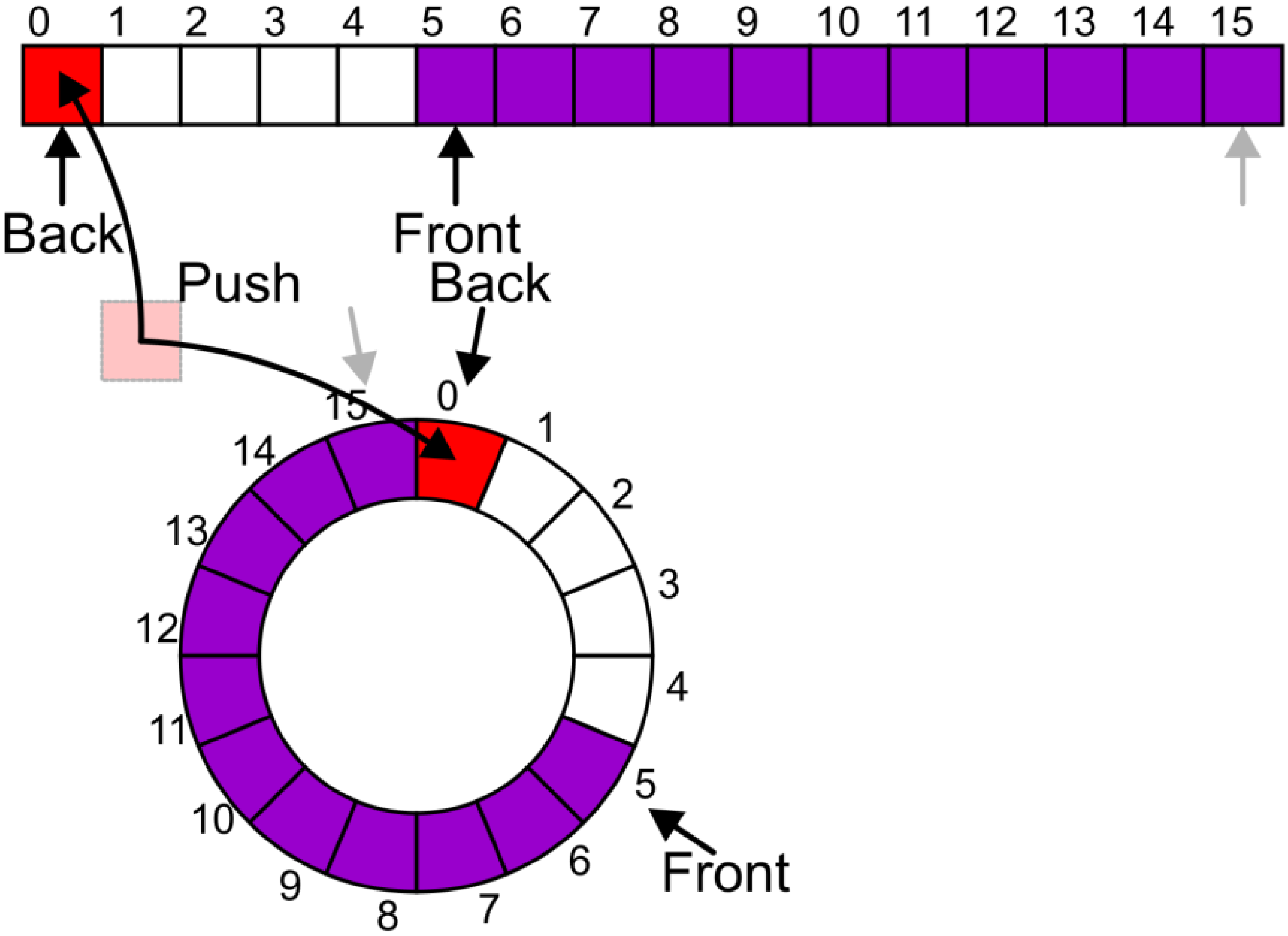
```
iback = (iback + 1) % capacity()
```

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

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

- or using modular arithmetic

```
iback = (iback + 1) % capacity()
```

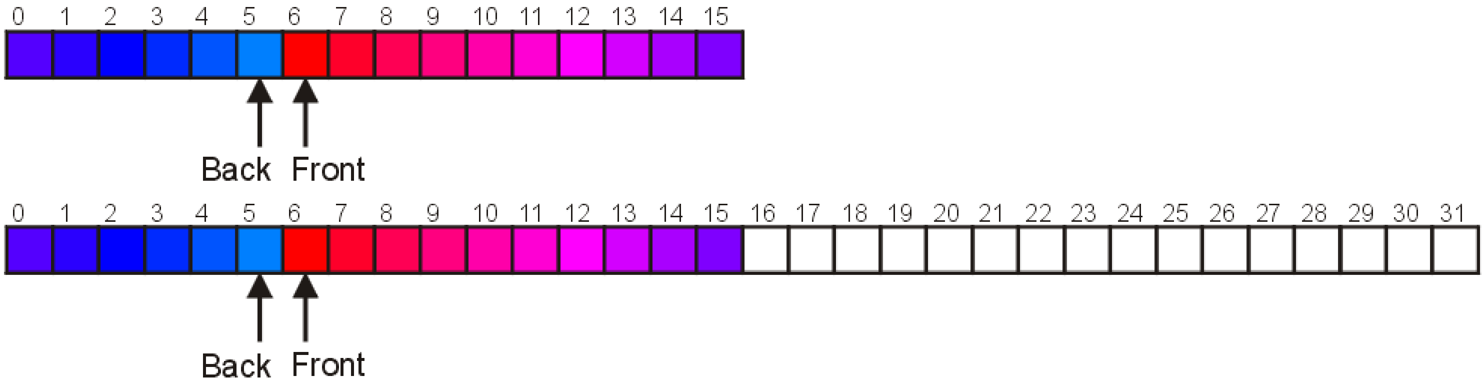

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Back
Front
Push
Back
Front
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Exceptions

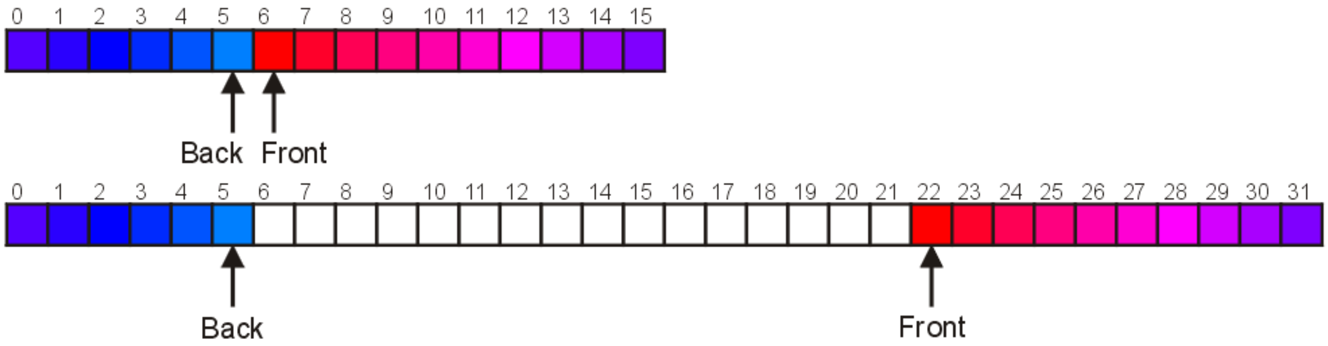
- As with a stack, there are a number of options which can be used if the array is filled
- If the array is filled, we have four options
  - Increase the size of the array
  - Throw an exception
  - Ignore the element being pushed
  - Put the pushing process to "sleep" until something else pops the front of the queue
- Include a member function `bool full() const;`

Increasing Capacity

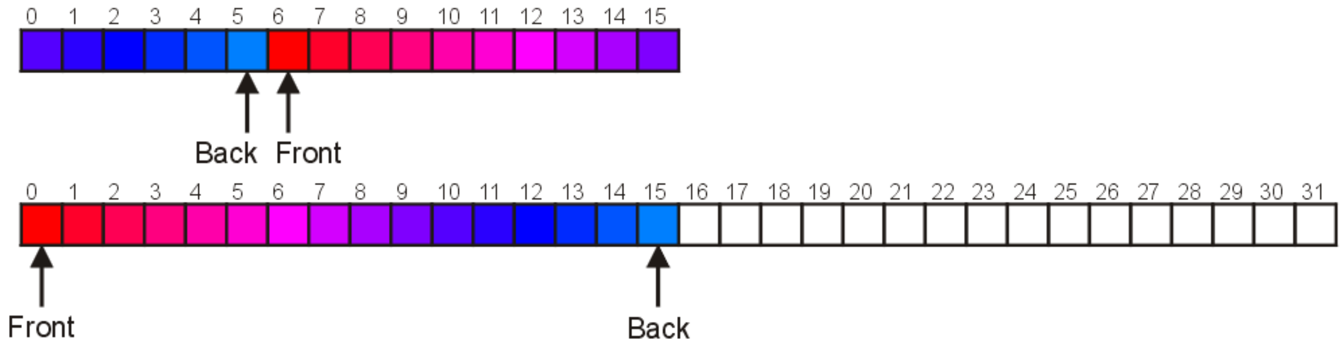
- Unfortunately, if we choose to increase the capacity, this becomes slightly more complex
- A direct copy does not work



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

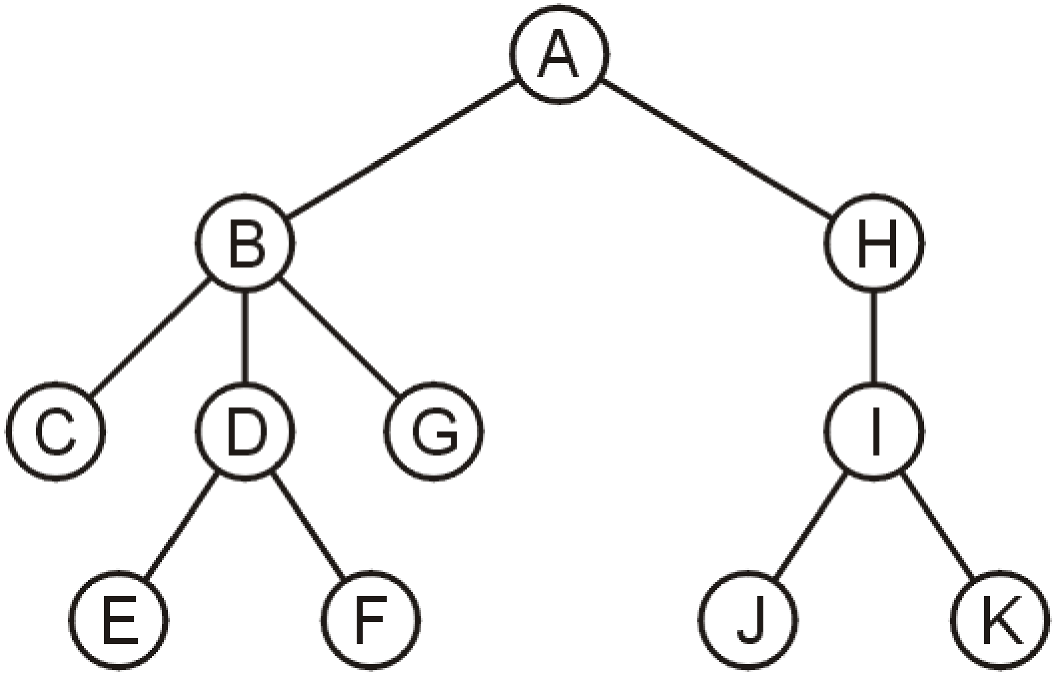


- An alternate solution is normalization
    - Map the front back at position 0
    - The next push would then occur in position 16



Application

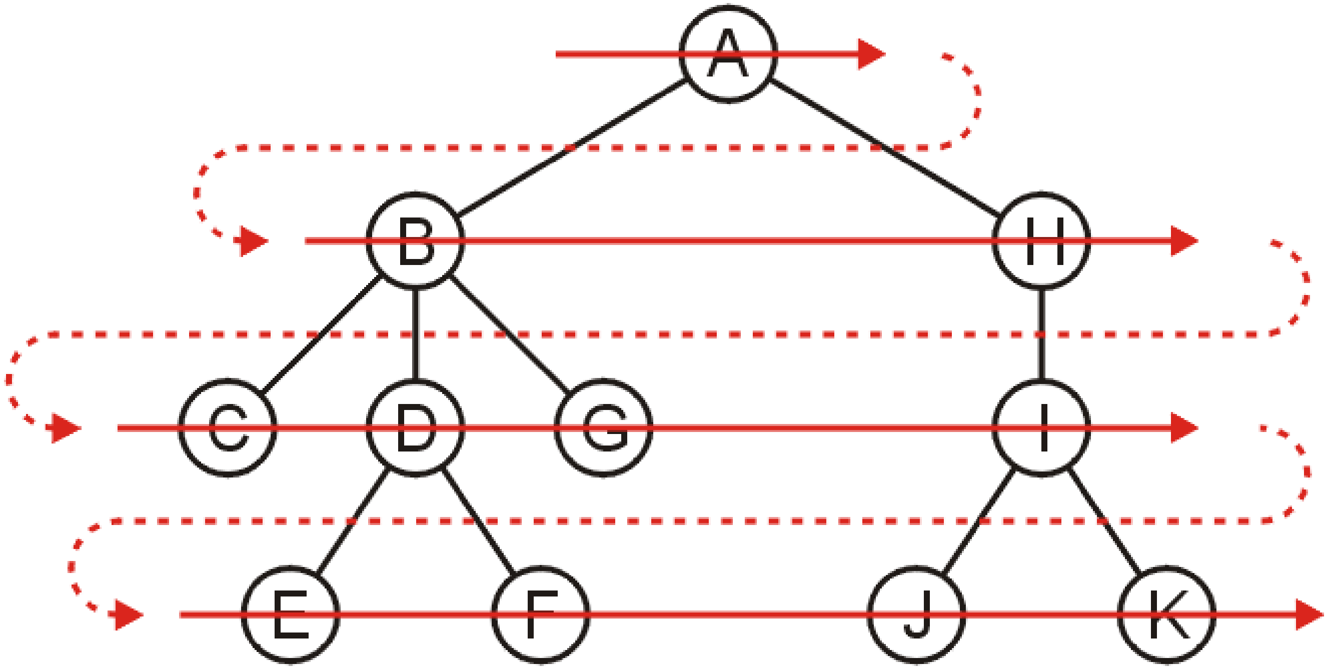
- One of the applications of the queue is performing a **breadth-first traversal** of a directory tree
  - Consider searching the directory structure



- 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 search (BFS)**

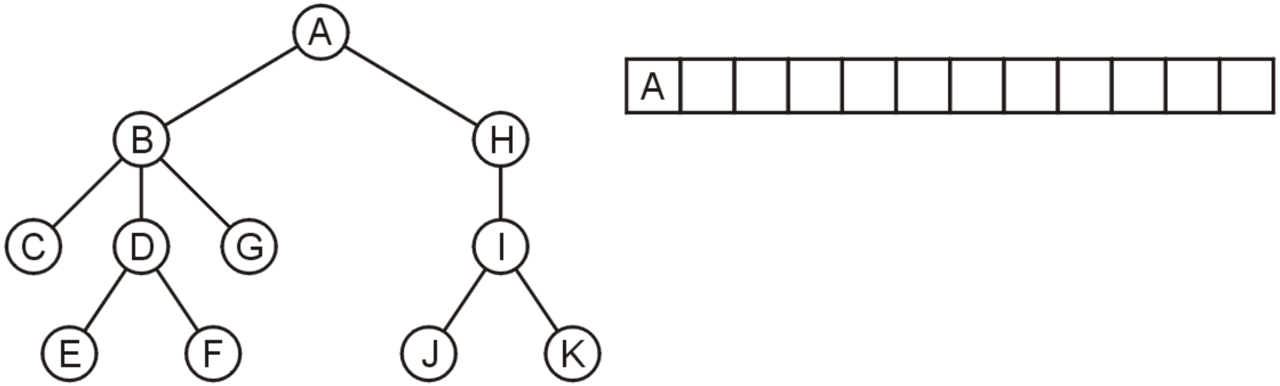
  - Search all the directories at one level before descending a level



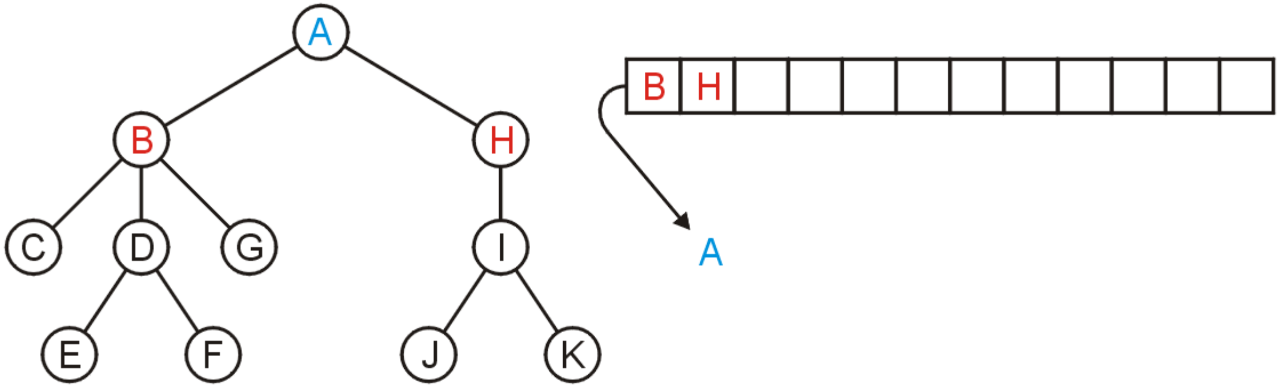
BFS Algorithm

- 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

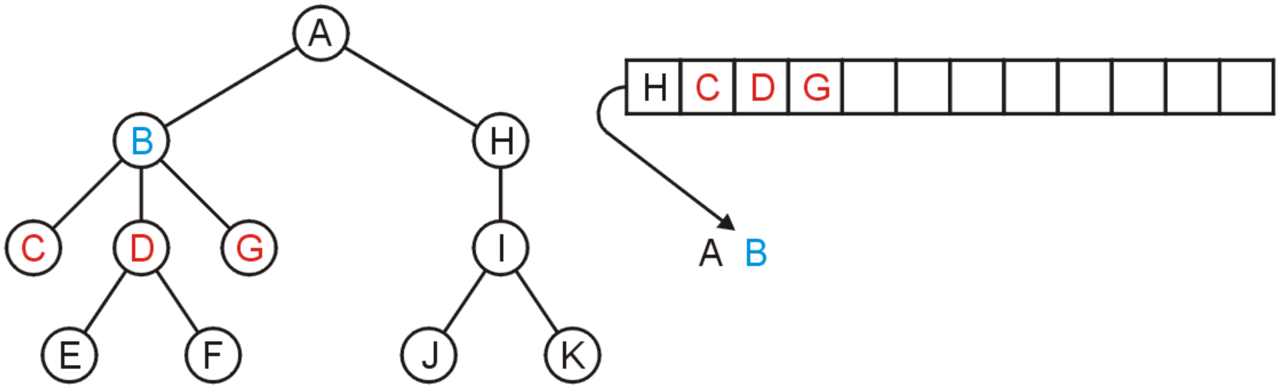
Push the root directory A



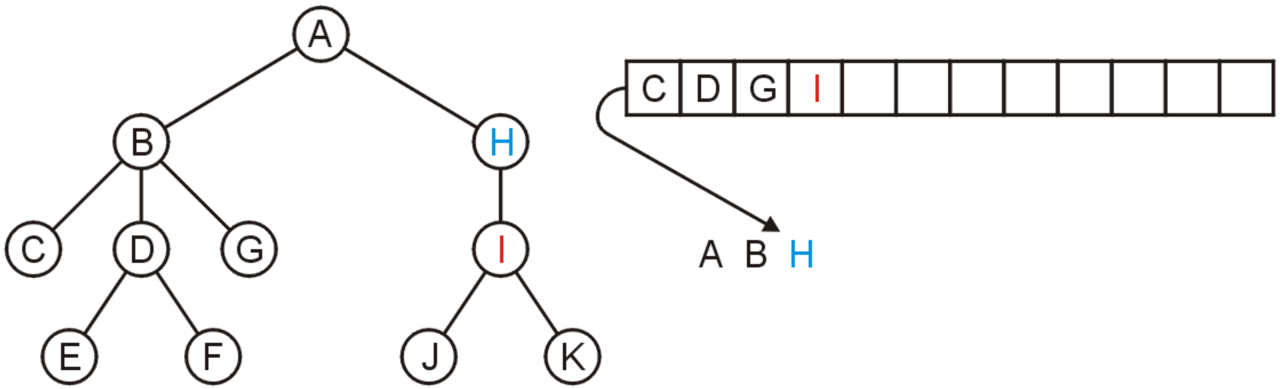
Pop A and push its two sub-directories B and H

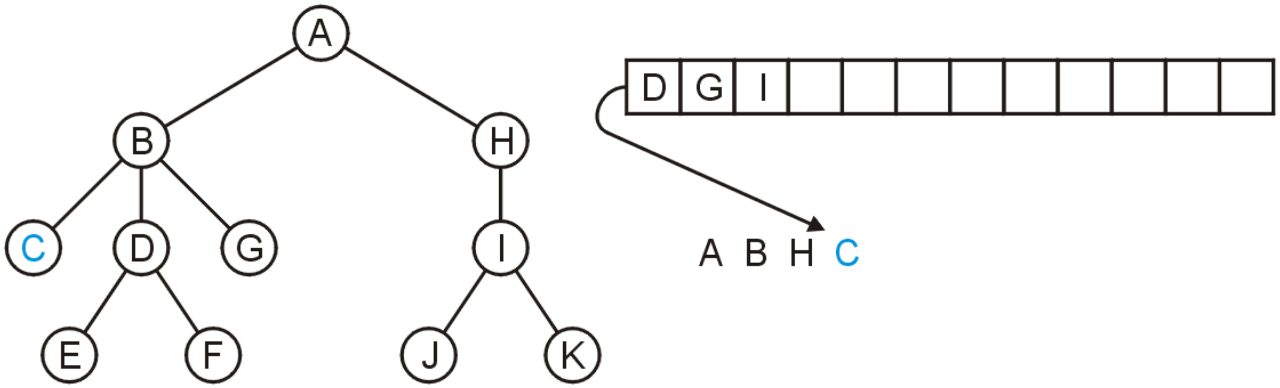
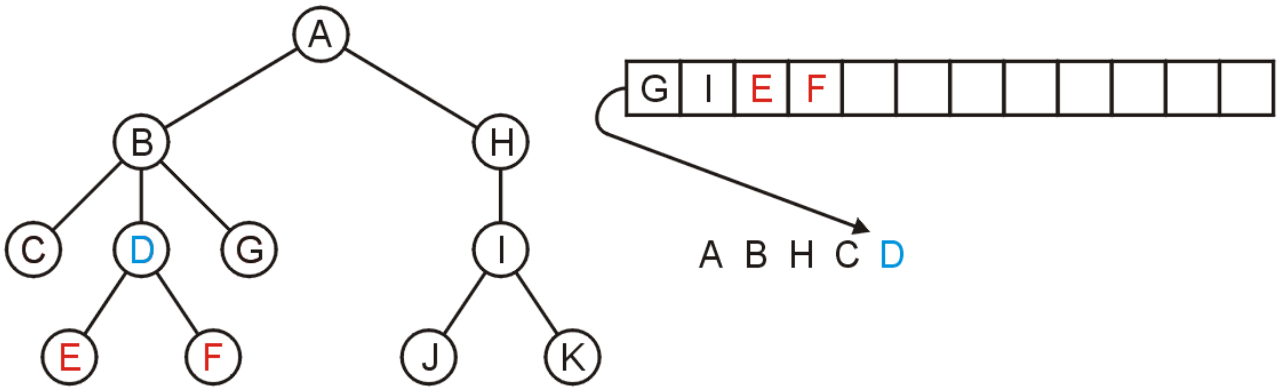


Pop B and push C, D, and G

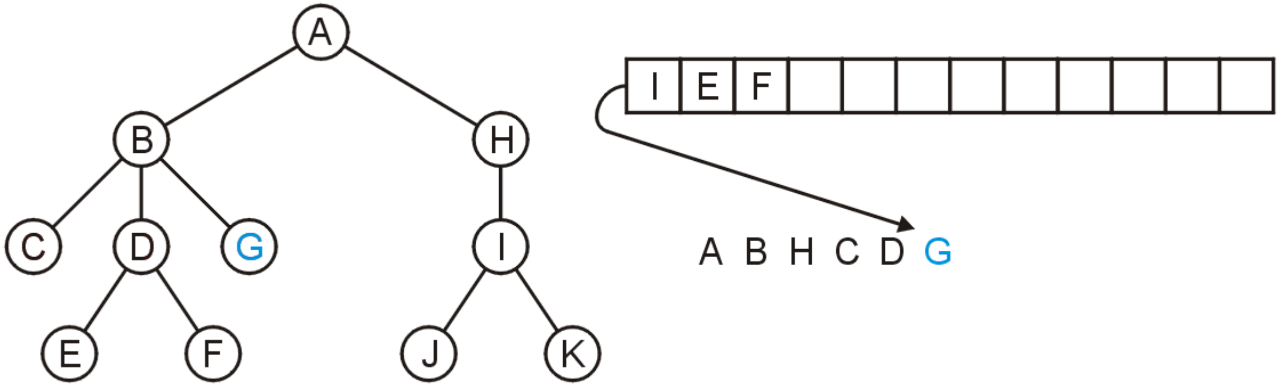


Pop H and push its one sub-directory I

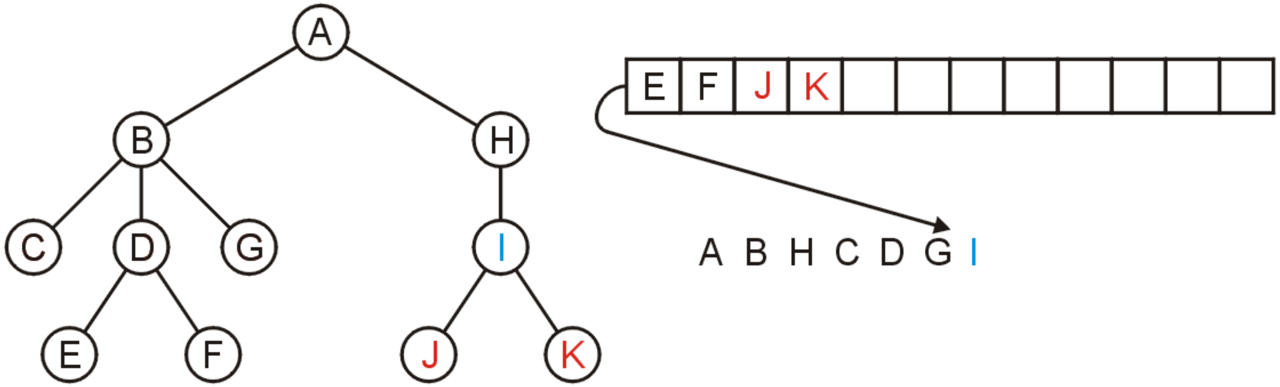


Pop C, no sub-directories	
Pop D and push E and F	

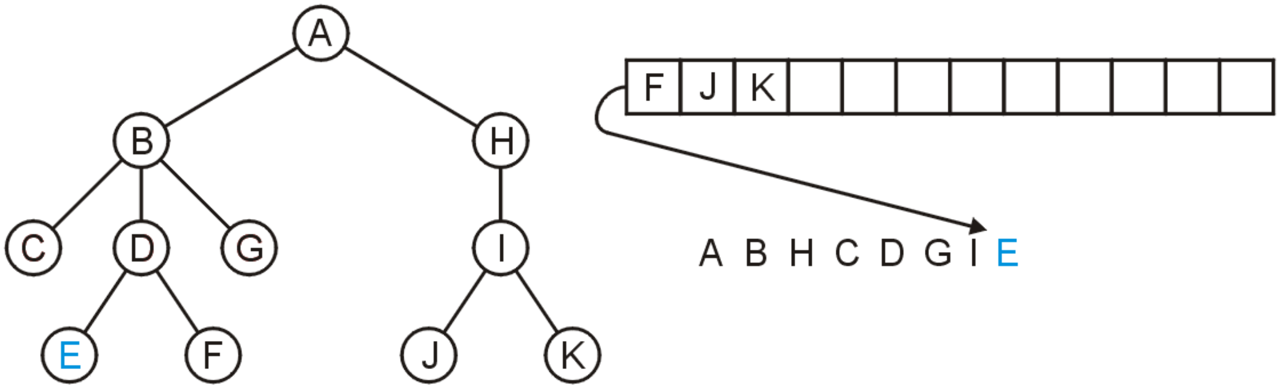
Pop G, no sub-directories



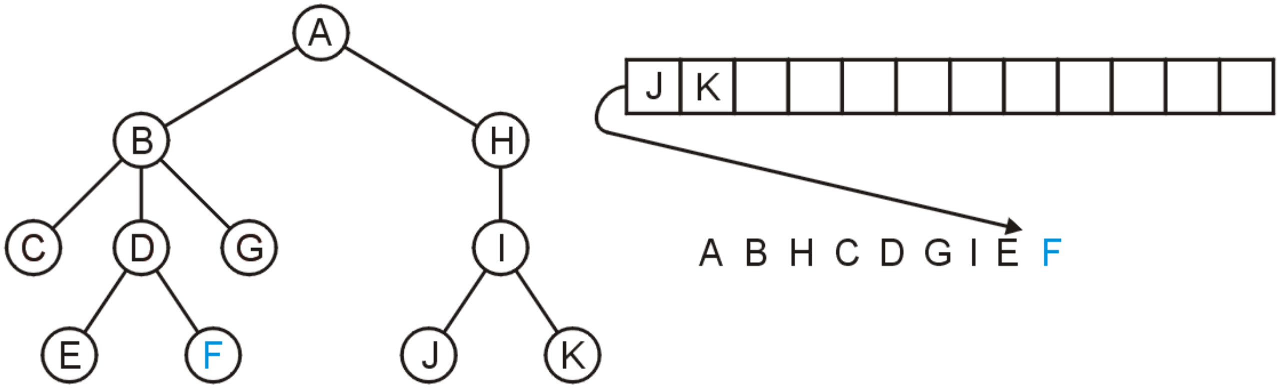
Pop I and push J and K



Pop E, no sub-directories



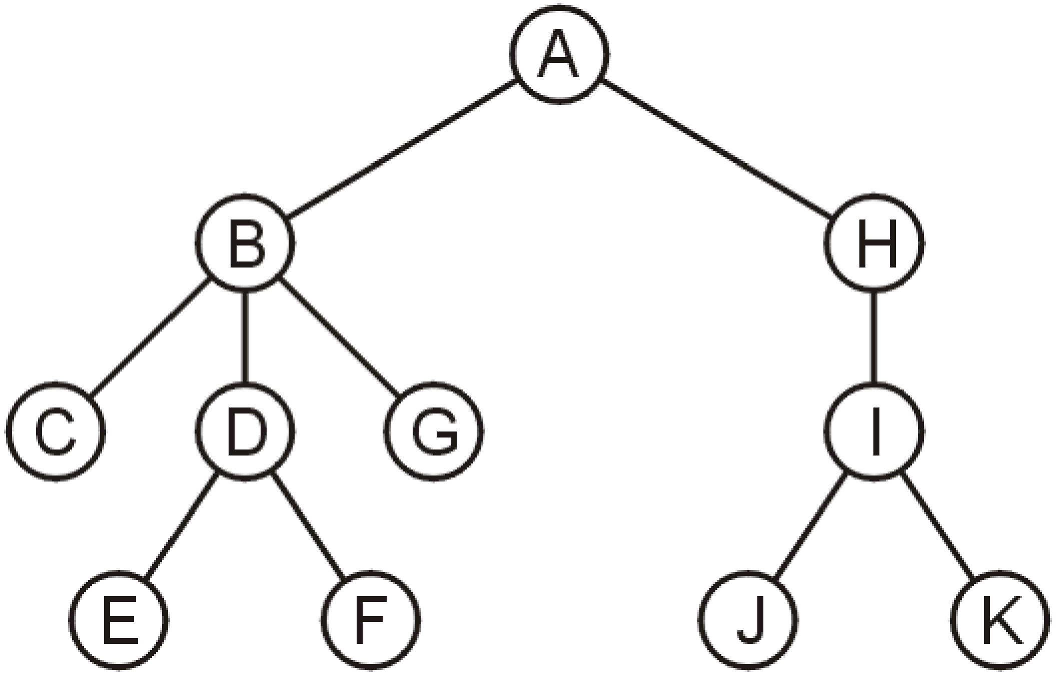
Pop F, no sub-directories



- The resulting order

  `A B H C D G I E F J K`

  is in breadth-first order of



Standard Template Library

- The Standard Template Library (STL) has a wrapper class `queue` which is a container adapter that gives the programmer the functionality of a queue

Standard Template Library

- The Standard Template Library (STL) has a wrapper class `queue` which is a container adapter that gives the programmer the functionality of a queue

In [1]:

```
#include <iostream>
#include <list>
#include <queue>
{
    std::queue< int, std::list<int> > iqueue;

    iqueue.push( 13 );
    iqueue.push( 42 );
    std::cout << "Head: " << iqueue.front() << std::endl;
    iqueue.pop(); // no return value
    std::cout << "Head: " << iqueue.front() << std::endl;
    std::cout << "Size: " << iqueue.size() << std::endl;
}
```

```
Head: 13
Head: 42
Size: 1
```

Queues

- 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

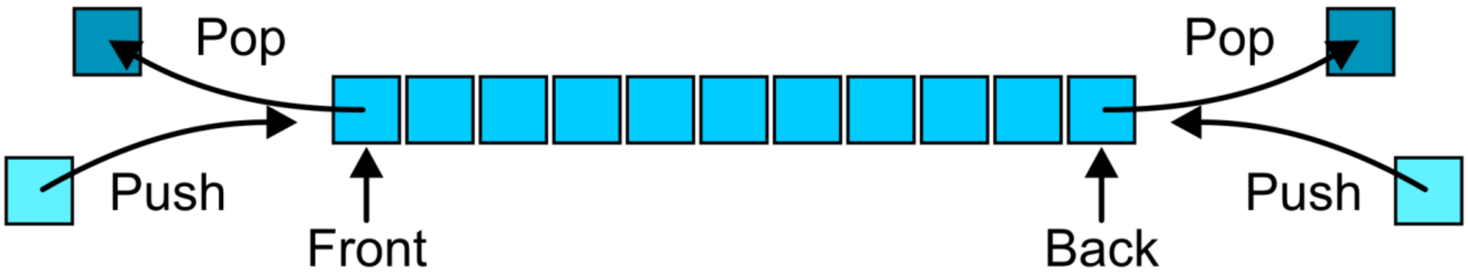
- Applications include:
  - Queuing clients in a client-server model
  - Breadth-first traversals of trees

Deque

Abstract Deque

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

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



Methods

- The operations will be called
    - front
    - back
    - push_front
    - push_back
    - pop_front
    - pop_back
- There are four errors associated with this abstract data type:
    - It is an undefined operation to access or pop from an empty deque

Applications

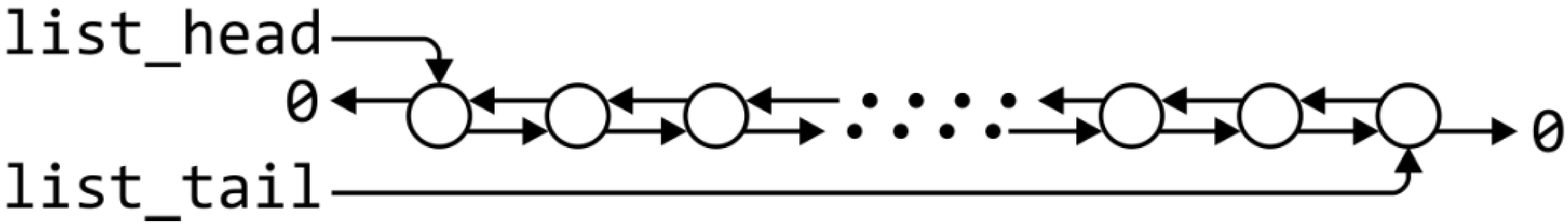
- Useful as a general-purpose tool
  - Can be used as either a queue or a stack
- Problem solving
  - Consider solving a maze by adding or removing a constructed path at the front
  - Once the solution is found, iterate from the back for the solution

Implementations

- The implementations are clear - use
    - a doubly linked list
    - a circular array

Doubly Linked List Implementation

- Every function of the deque ADT runs in $\Theta(1)$ time.
  - The space usage is $O(n)$



	front / 1st node	back / nth node
find	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(1)$	$\Theta(1)$
erase	$\Theta(1)$	$\Theta(1)$

- Most of the member functions for the `LinkedDeque` class are straightforward generalizations of the corresponding functions of the `LinkedQueue` class
  - Simply invoke the appropriate operation from the underlying `DoublyLinkedList` object

Standard Template Library

- The Standard Template Library (STL) has a wrapper class `deque` which is a container adapter that gives the programmer the functionality of a deque

Standard Template Library

- The Standard Template Library (STL) has a wrapper class `deque` which is a container adapter that gives the programmer the functionality of a deque

In [3]:

```
#include <iostream>
#include <deque>
{
    std::deque<int> ideque;

    ideque.push_front( 12 );
    ideque.push_back( 42 );
    ideque.push_front( 11 );
    std::cout << "Head: " << ideque.front() << std::endl;
    std::cout << "Tail: " << ideque.back() << std::endl;
    ideque.pop_front(); // no return value
    std::cout << "Head: " << ideque.front() << std::endl;
    std::cout << "Size: " << ideque.size() << std::endl;
}
```

```
Head: 11
Tail: 42
Head: 12
Size: 2
```