ECE430.217 Data Structures

Queues

Weiss Book Chapter 3

Byoungyoung Lee
https://compsec.snu.ac.kr
byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

This topic discusses the concept of a queue:

- Description of an Abstract Queue
- List applications
- Implementation
- Queuing theory
- Standard Template Library
- Descriptions of an Abstract Deque

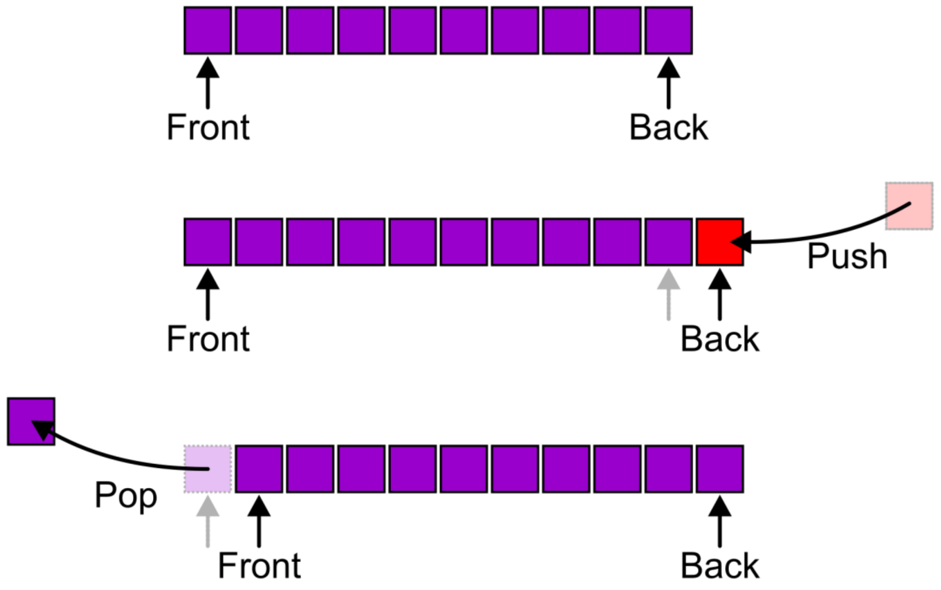
Abstract Queue

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

- The **insert** operation pushes an object to the back of the queue
- The **remove** operation (*popping* from the queue) removes the current *front* of the queue

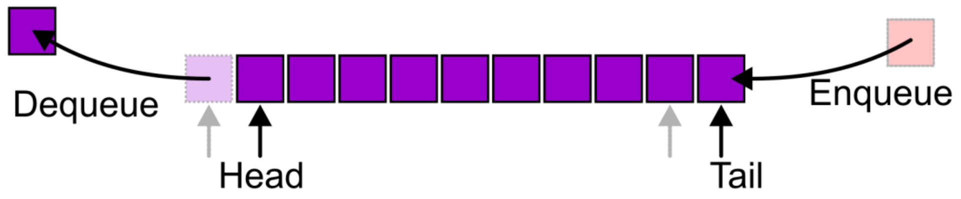
Abstract Queue

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



Abstract Queue

Alternative term: enqueue, dequeue, head, and tail



Abstract Queue

There are two exceptions associated with this abstract queue:

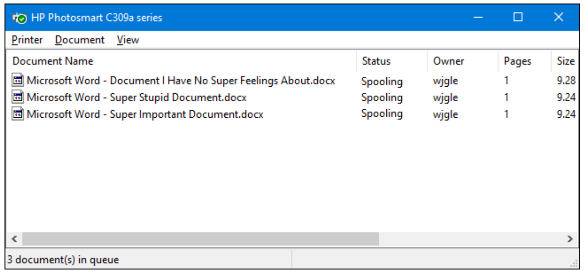
- It is an undefined operation to call either pop or front on an empty queue

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, and computing services use queues



Document Name	Status	Owner	Pages	Size
Microsoft Word - Document I Have No Super Feelings About.docx	Spooling	wjgle	1	9.28
Microsoft Word - Super Stupid Document.docx	Spooling	wjgle	1	9.24
Microsoft Word - Super Important Document.docx	Spooling	wjgle	1	9.24

3 document(s) in queue

Implementations

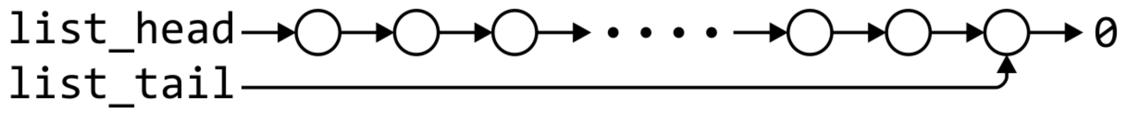
We will look at two implementations of queues:

- Singly linked lists
- Circular arrays

Requirements:

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

Linked-List Implementation of Queue



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

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

Insertion is performed at the back with $\Theta(1)$ run time

Single_list Definition

```
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();
}
```

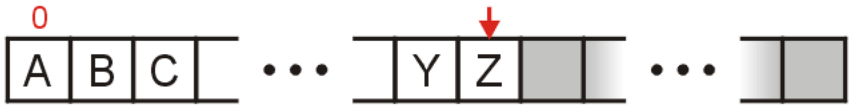
```
template <typename Type>
void Queue<Type>::push( Type const &obj ) {
    list.push_back( obj );
}
```

```
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 of Queue

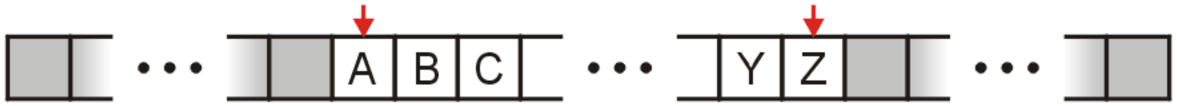
An one-ended array does not allow all operations to occur in $\Theta(1)$ time



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

Array Implementation of Queue

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



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

Array Implementation of Queue

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 member variables
- 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:

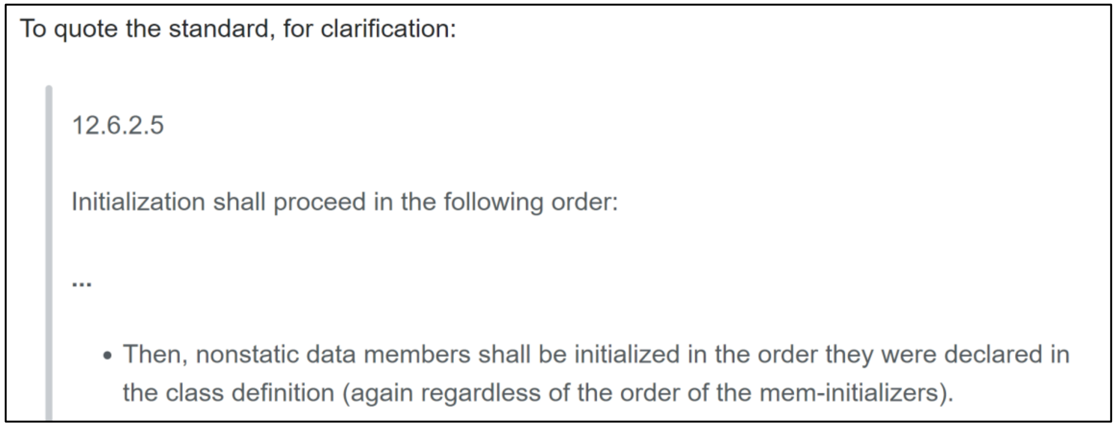
- Initialization is performed in the order specified in the class declaration

```
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();
};
```

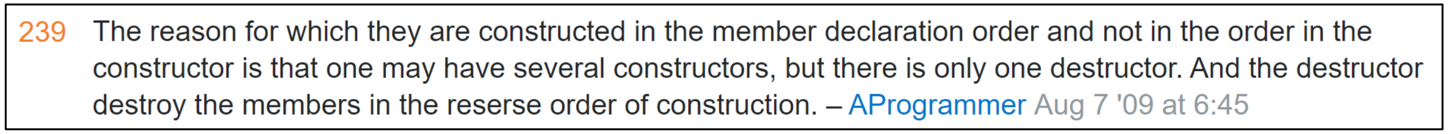
Constructor

For more details,



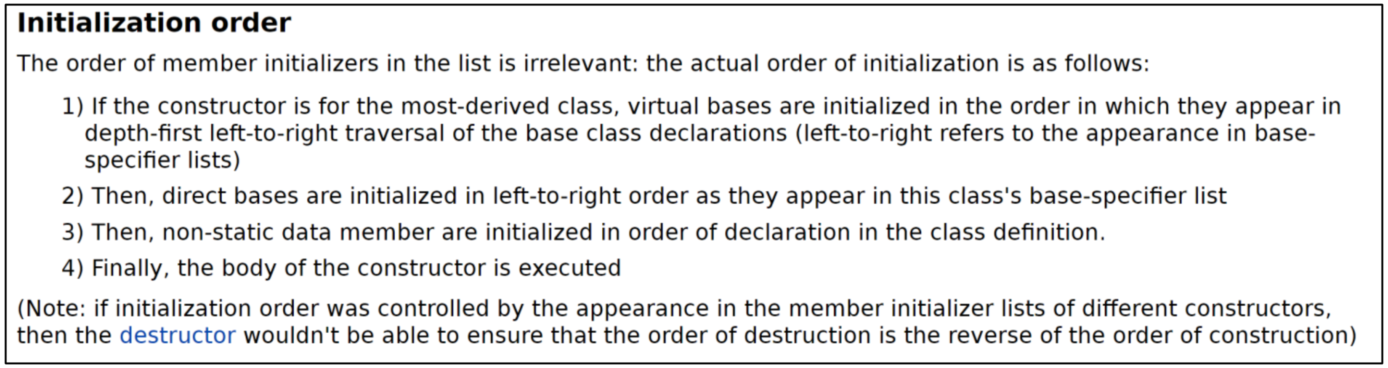
To quote the standard, for clarification:

> 12.6.2.5
>
> Initialization shall proceed in the following order:
>
> ...
>
> - Then, nonstatic data members shall be initialized in the order they were declared in the class definition (again regardless of the order of the mem-initializers).



239 The reason for which they are constructed in the member declaration order and not in the order in the constructor is that one may have several constructors, but there is only one destructor. And the destructor destroy the members in the reserse order of construction. – AProgrammer Aug 7 '09 at 6:45

https://stackoverflow.com/questions/1242830/constructor-initialization-list-evaluation-order



Initialization order

The order of member initializers in the list is irrelevant: the actual order of initialization is as follows:

1) If the constructor is for the most-derived class, virtual bases are initialized in the order in which they appear in depth-first left-to-right traversal of the base class declarations (left-to-right refers to the appearance in base-specifier lists)

2) Then, direct bases are initialized in left-to-right order as they appear in this class's base-specifier list

3) Then, non-static data member are initialized in order of declaration in the class definition.

4) Finally, the body of the constructor is executed

(Note: if initialization order was controlled by the appearance in the member initializer lists of different constructors, then the destructor wouldn't be able to ensure that the order of destruction is the reverse of the order of construction)

https://en.cppreference.com/w/cpp/language/constructor

Destructor

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

Member Functions

It’s simple to implement empty() and front()

```
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 issues:

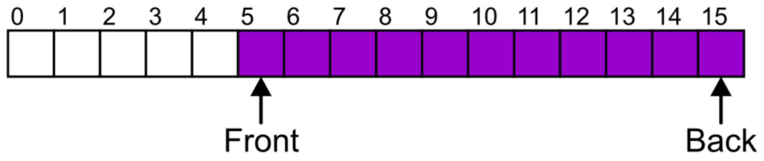
```
template <typename Type>
void Queue<Type>::push( Type const &obj ) {
    if ( queue_size == array_capacity ) {
        throw overflow();
    }

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

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

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

Circular 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

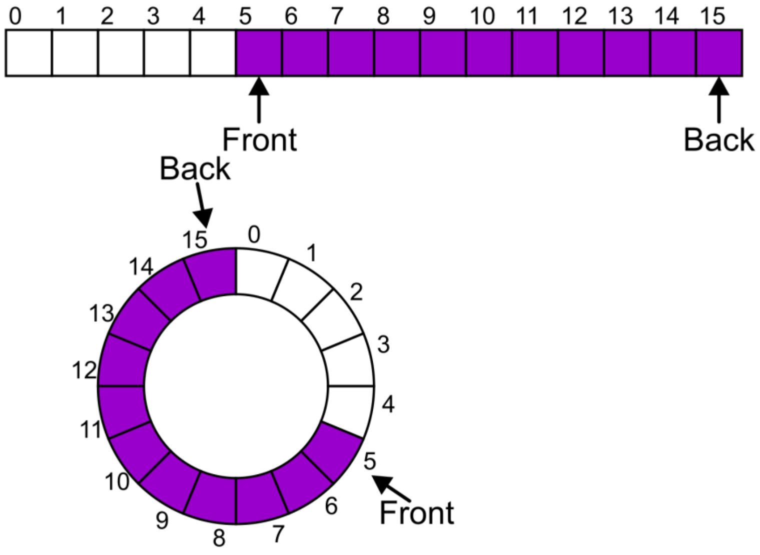
In this case, the array is not full and yet we cannot place any more objects in to the array

Circular Array

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

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

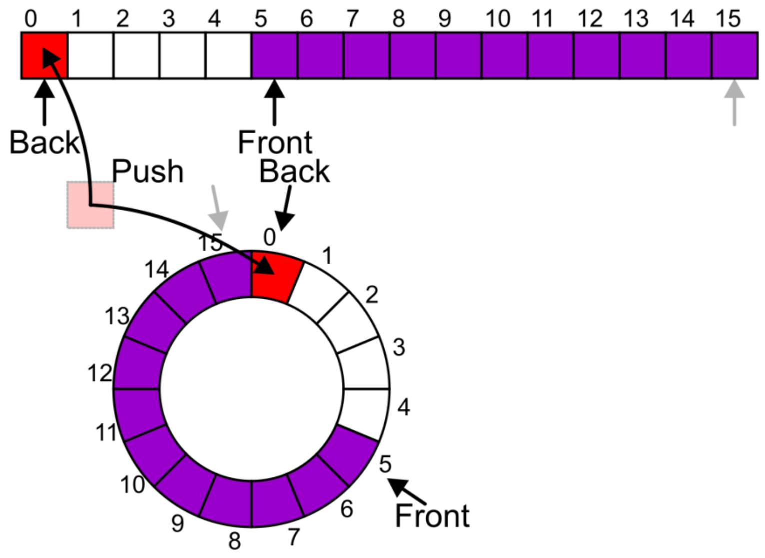
This is referred to as a *circular array*



Circular Array

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

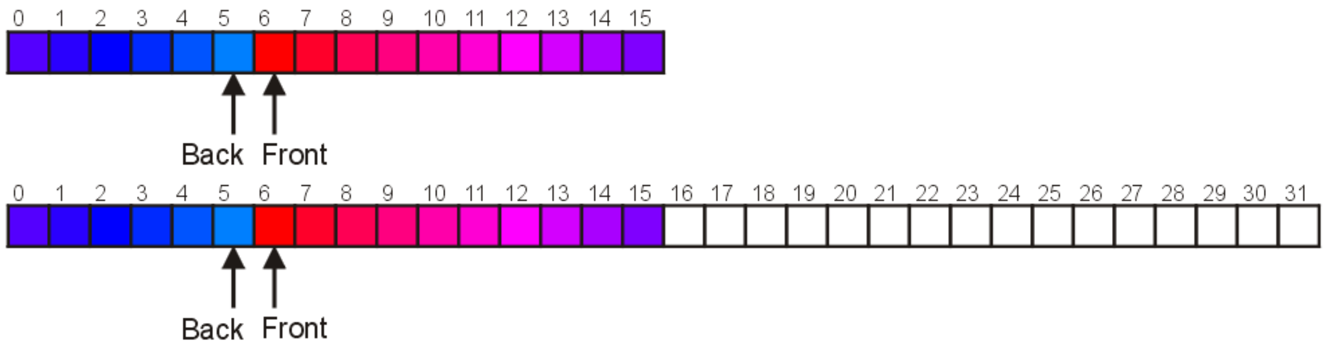
If the array is filled, we have five options:

– 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

Increasing Capacity

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

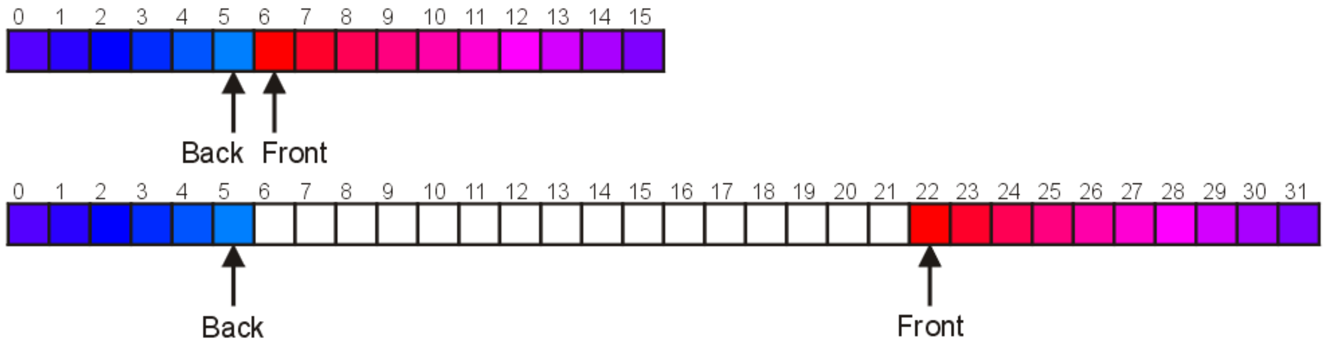
– A direct copy does not work:



Increasing Capacity: Solution #1

The first solution:

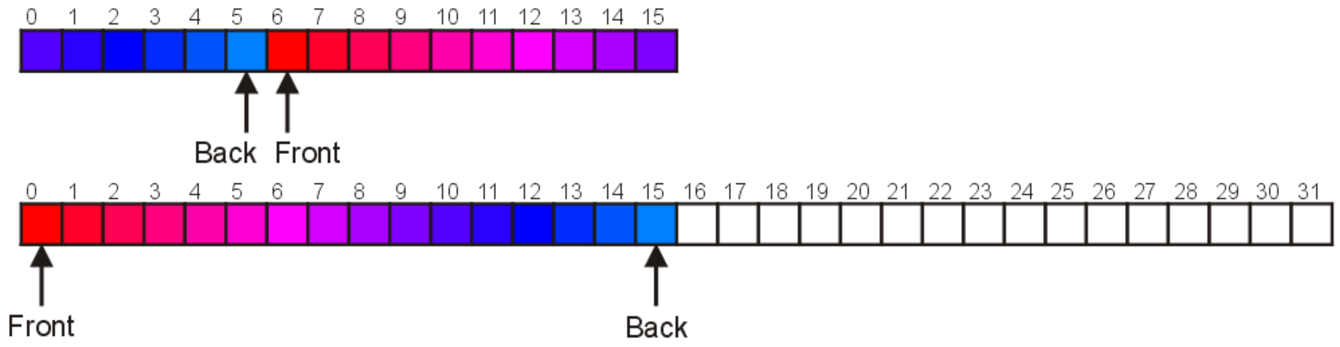
- Move those beyond the front to the end of the array
- The next push would then occur in position 6



Increasing Capacity: Solution #2

Another solution:

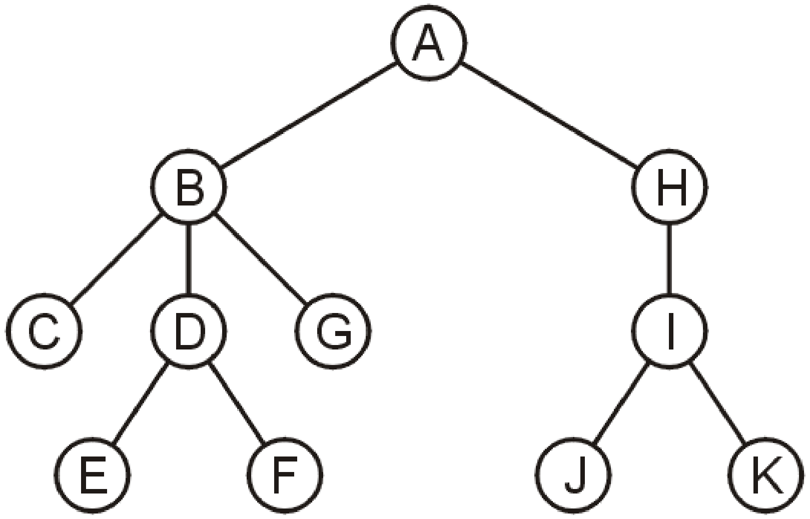
- Map the front back at position 0
- The next push would then occur in position 16



Application: Breath-first traversal

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

- Consider searching the directory structure

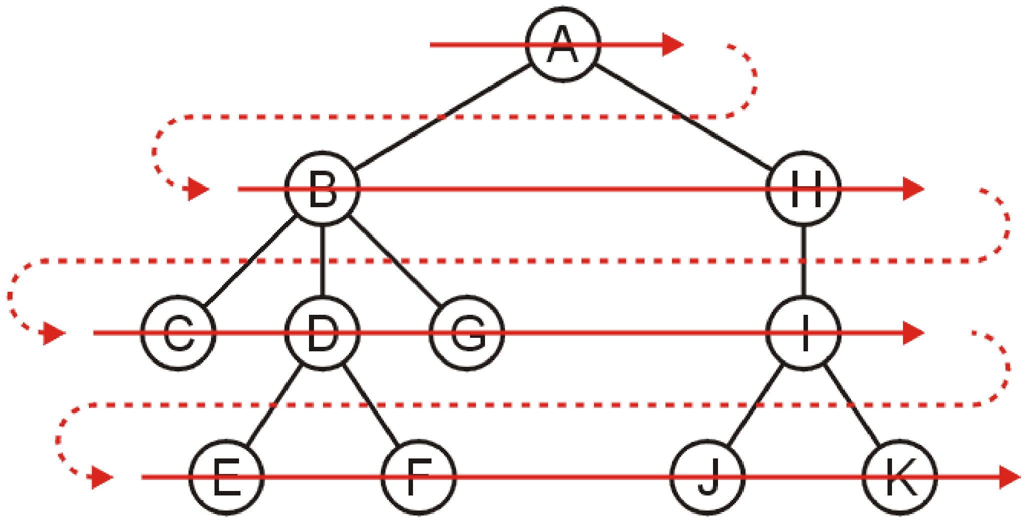


Application: Breath-first traversal

We would rather search the more shallow directories first, and then search into one sub-directory and all of its contents

Such a search strategy is called a *breadth-first traversal*

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



Application: Breath-first traversal

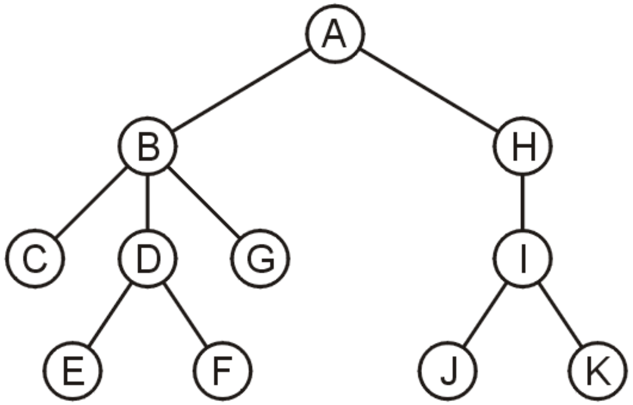
The easiest implementation is:

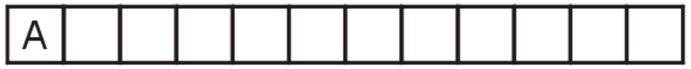
- Place the root directory into a queue
- While the queue is not empty:
  - Pop the directory at the front of the queue
  - Push all of its sub-directories into the queue

The order in which the directory popped from the queue will be in breadth-first order

Application: Breath-first traversal

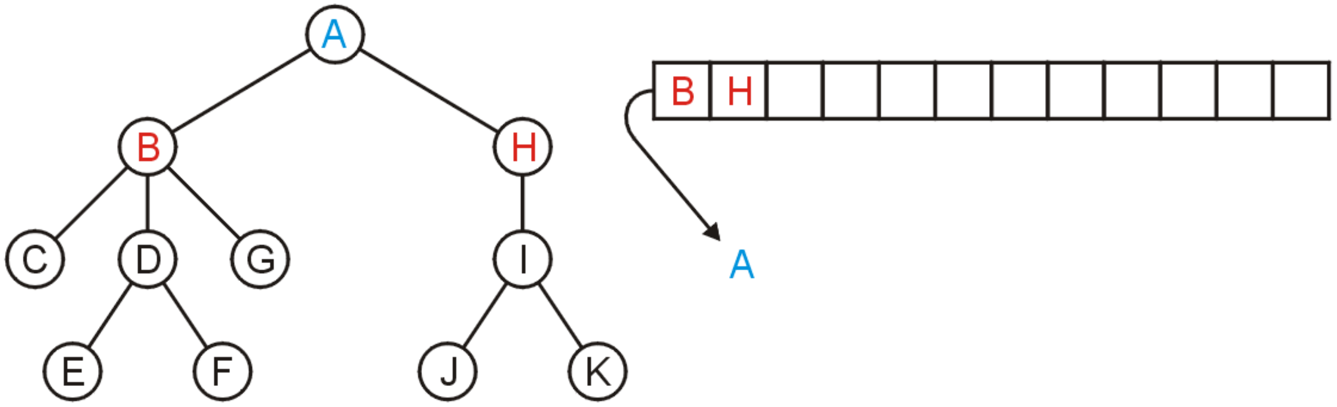
Push the root directory A





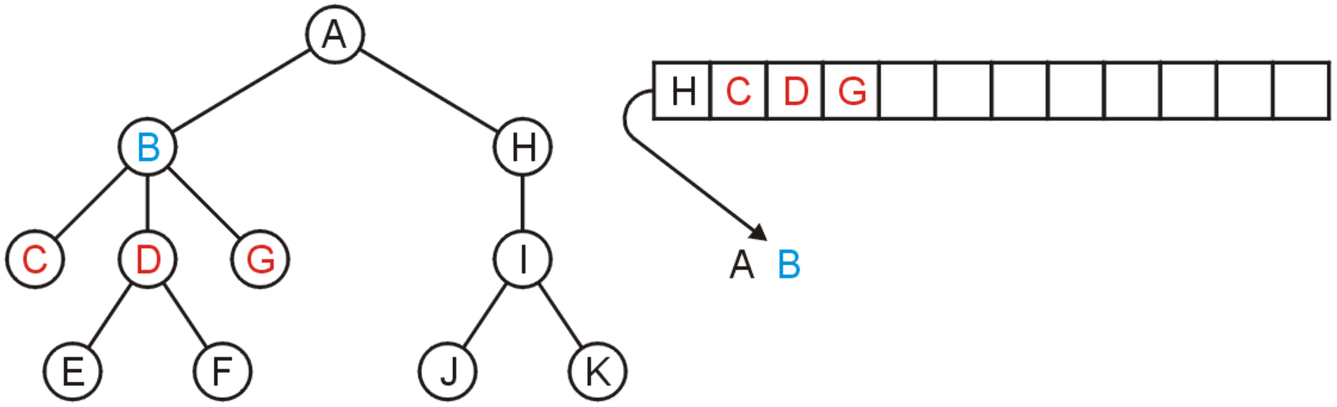
Application: Breath-first traversal

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



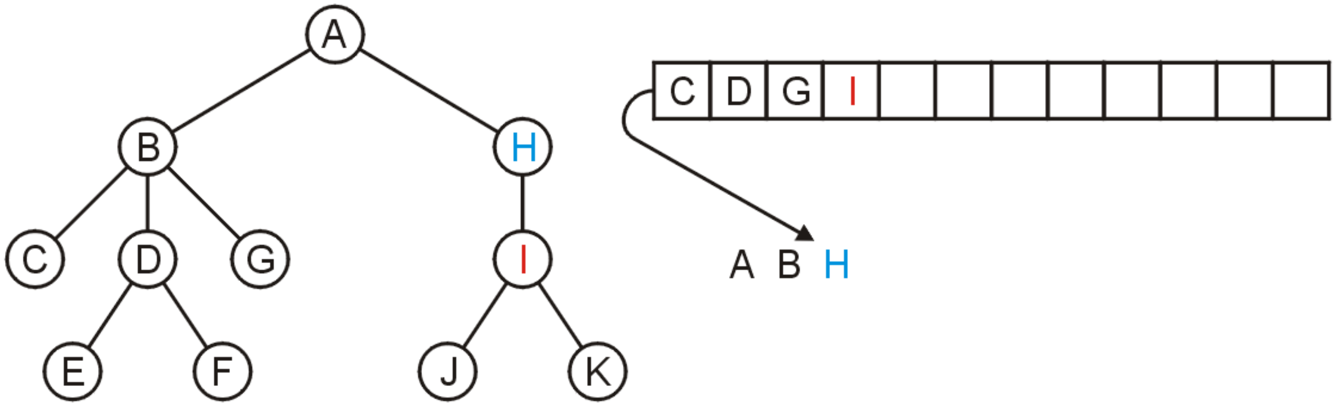
Application: Breath-first traversal

Pop B and push C, D, and G



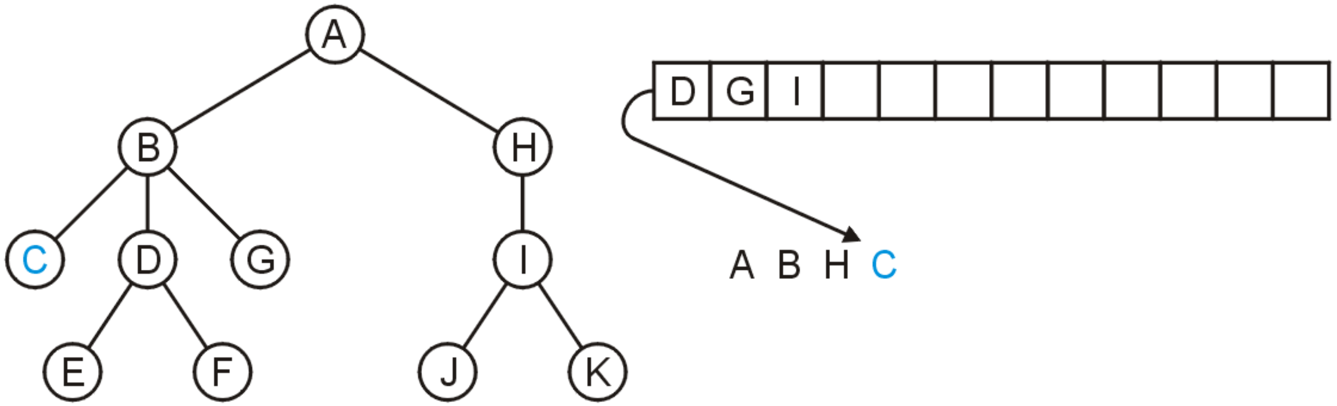
Application: Breath-first traversal

Pop H and push its one sub-directory I



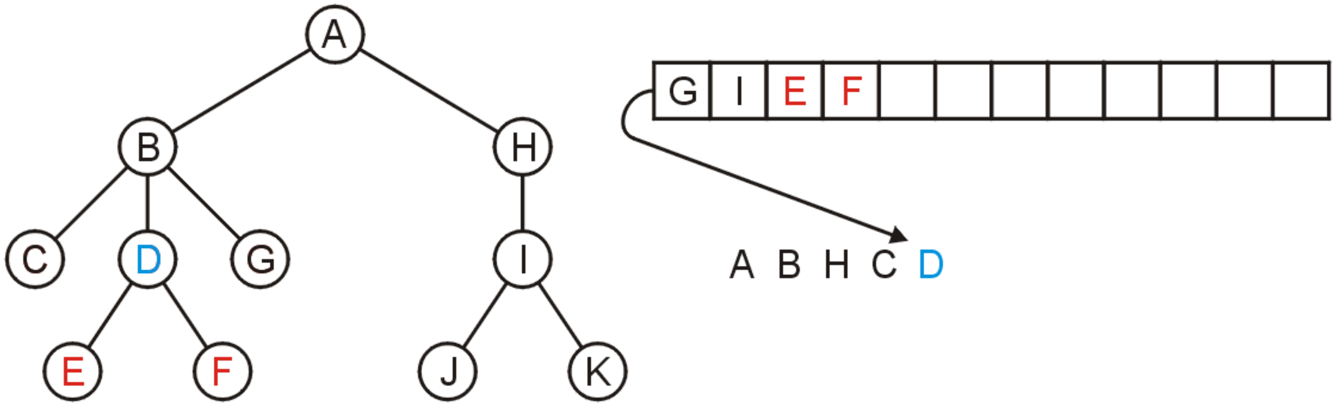
Application: Breath-first traversal

Pop C: no sub-directories



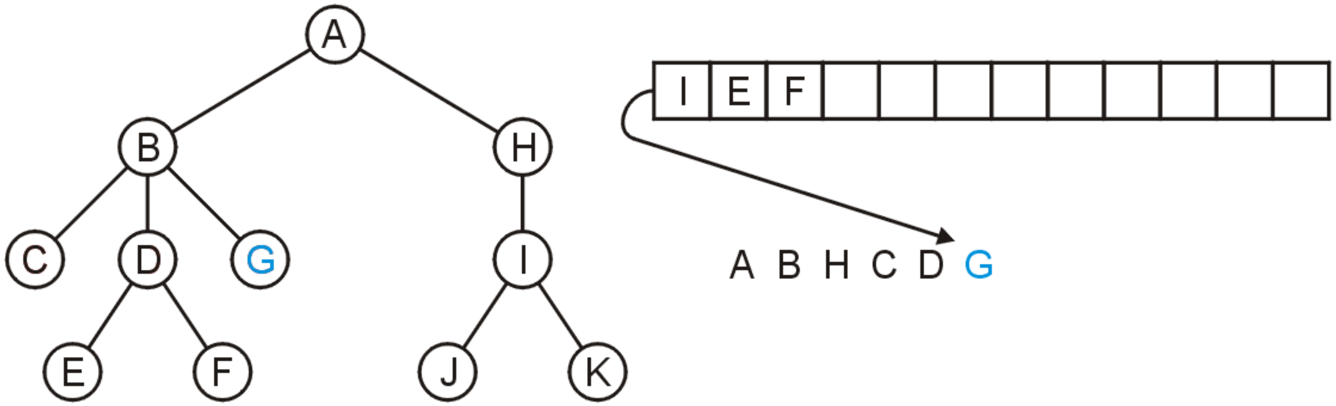
Application: Breath-first traversal

Pop D and push E and F



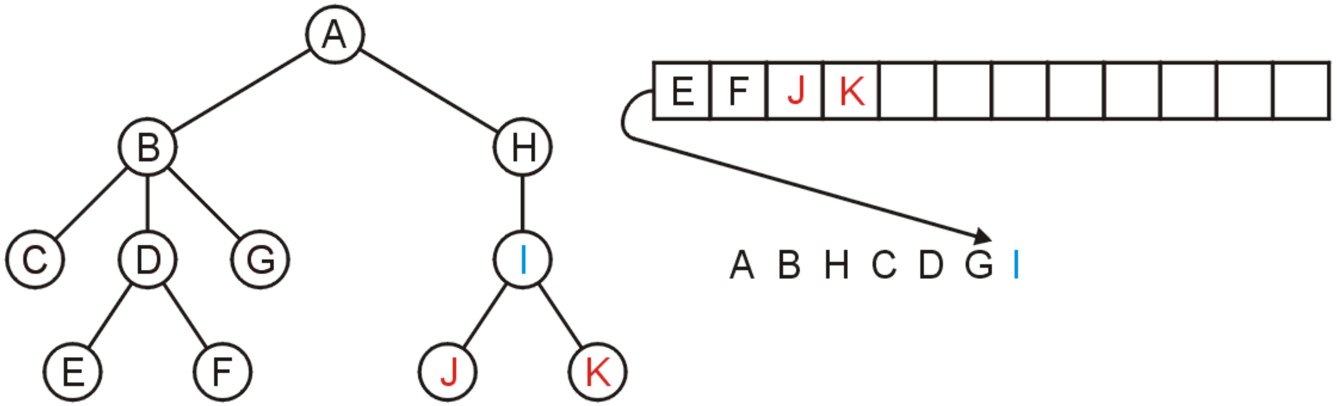
Application: Breath-first traversal

Pop G



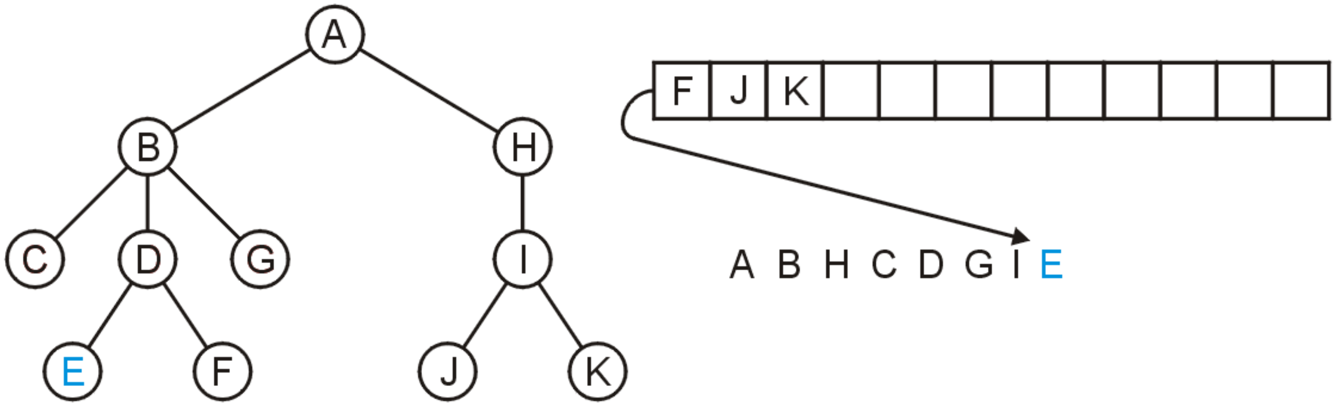
Application: Breath-first traversal

Pop I and push J and K



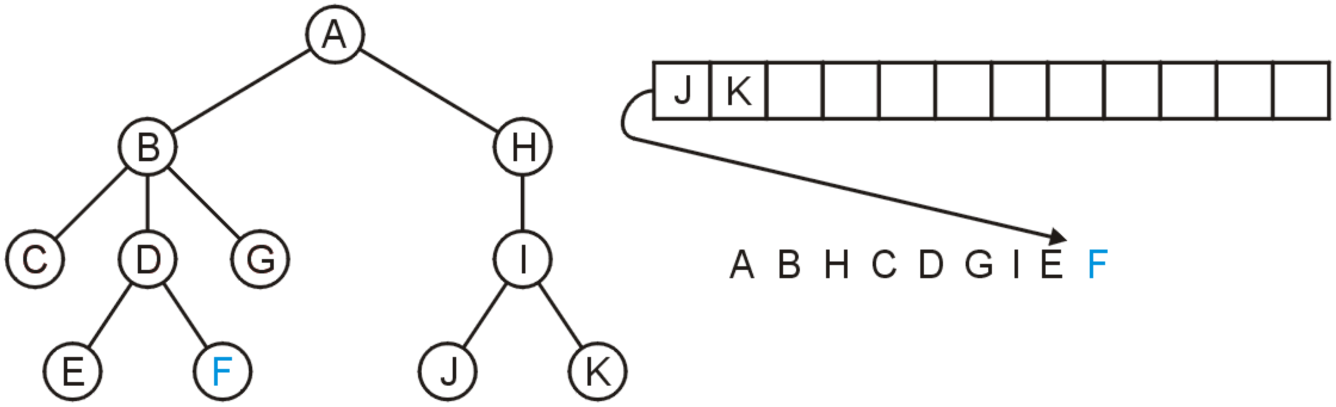
Application: Breath-first traversal

Pop E



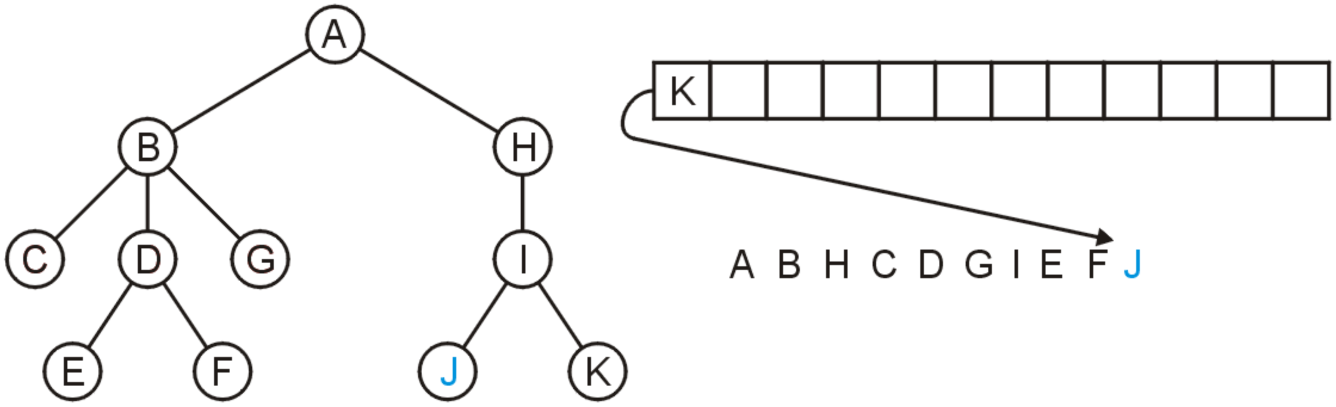
Application: Breath-first traversal

Pop F



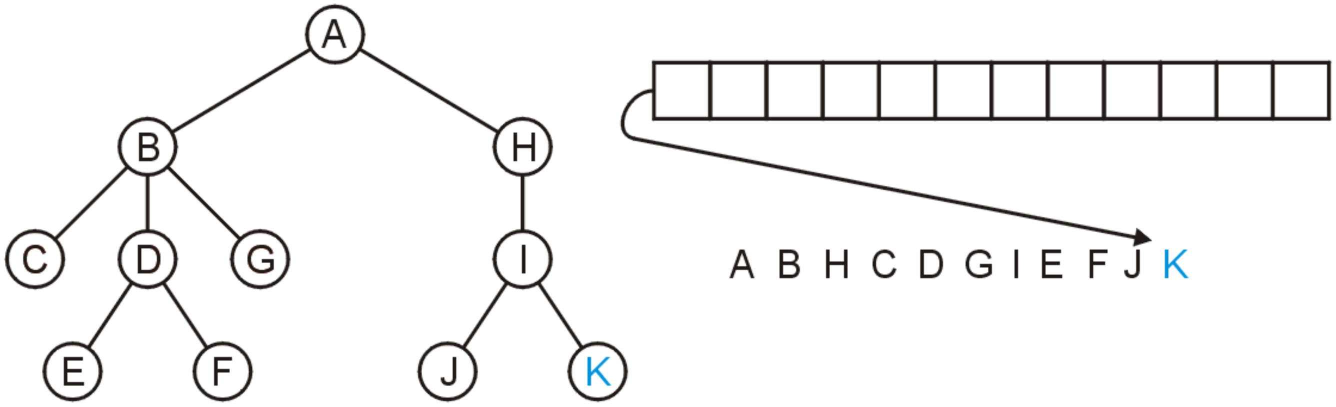
Application: Breath-first traversal

Pop J



Application: Breath-first traversal

Pop K and the queue is empty

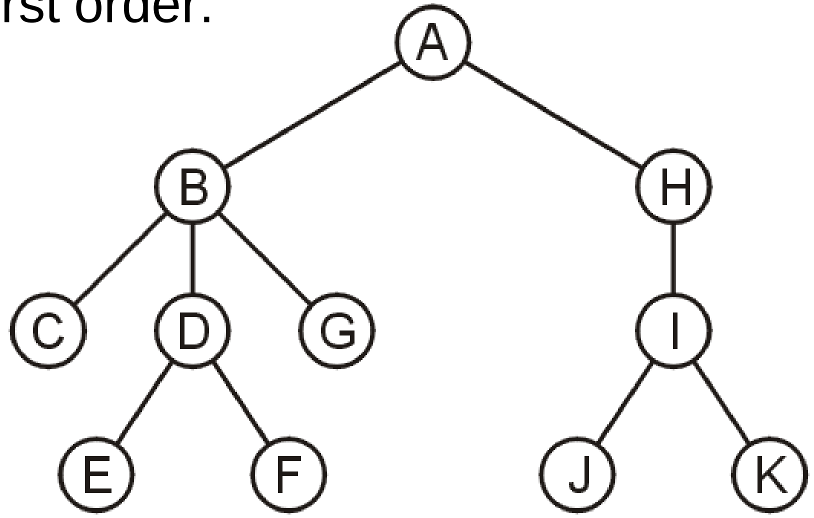


Application: Breath-first traversal

The resulting order

A B H C D G I E F J K

is in breadth-first order:



Application: Breath-first traversal

- **Q. What would happen if we use "stack" instead of "queue"?**
  - Place the root directory into a ~~queue~~ stack
  - While the ~~queue~~ stack is not empty:
    - Pop the directory at the front of the ~~queue~~ stack
    - Push all of its sub-directories into the ~~queue~~ stack (in a reverse order)

Q. What's the order in which the directory popped from the ~~queue~~ stack?

Standard Template Library

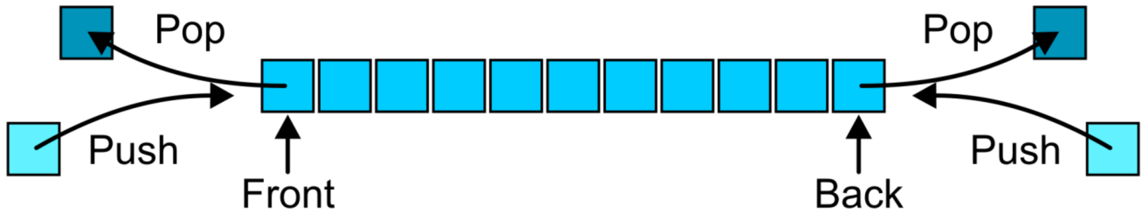
An example of a queue in the STL is:

```
#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:

- Allows insertions at both the front and back of the deque



Implementations of Deque

The implementations are clear:

- Use either a doubly linked list or a circular array

Standard Template Library of Deque

The STL comes with a deque data structure:

```
deque<T>
```

The signatures use stack terminology:

```
T &front();
void push_front(T const &);
void pop_front();
T &back();
void push_back(T const &);
void pop_back();
```

Standard Template Library of Deque

```
$ g++ deque_example.cpp
$ ./a.out
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
$
```

```
#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 );       // 3 5 4 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;
}
```

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

References

Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd Ed., Addison Wesley, 1997, §2.2.1, p.238.

Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, §11.1, p.200.

Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley, §3.6, p.94.

Koffman and Wolfgang, “Objects, Abstraction, Data Strucutes and Design using C++”, John Wiley & Sons, Inc., Ch. 6.

Wikipedia, http://en.wikipedia.org/wiki/Queue_(abstract_data_type)