Lists, stacks and queues

Lists

We will now look at our first abstract data structure

- Relation: explicit linear ordering
- Operations
- Implementations of an abstract list with:
  - Linked lists
  - Arrays
- Memory requirements
- Strings as a special case
- The STL `vector` class

Definition

An Abstract List (or List ADT) is linearly ordered data where the programmer explicitly defines the ordering

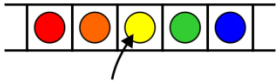
We will look at the most common operations that are usually

- The most obvious implementation is to use either an array or linked list
- These are, however, not always the most optimal

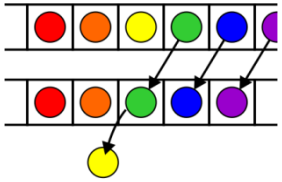
Operations

Operations at the k^{th} entry of the list include:

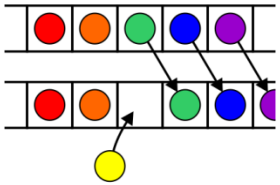
Access to the object



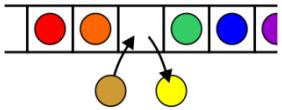
Erasing an object



Insertion of a new object

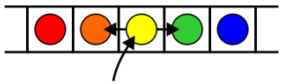


Replacement of the object



Operations

Given access to the k^{th} object, gain access to either the previous or next object



Given two abstract lists, we may want to

- Concatenate the two lists
- Determine if one is a sub-list of the other

Locations and run times

The most obvious data structures for implementing an abstract list are arrays and linked lists

- We will review the run time operations on these structures

We will consider the amount of time required to perform actions such as finding, inserting new entries before or after, or erasing entries at

- the first location (the *front*)
- an arbitrary (k^{th}) location
- the last location (the *back* or n^{th})

The run times will be $\Theta(1)$, $\mathrm{O}(n)$ or $\Theta(n)$

Linked lists

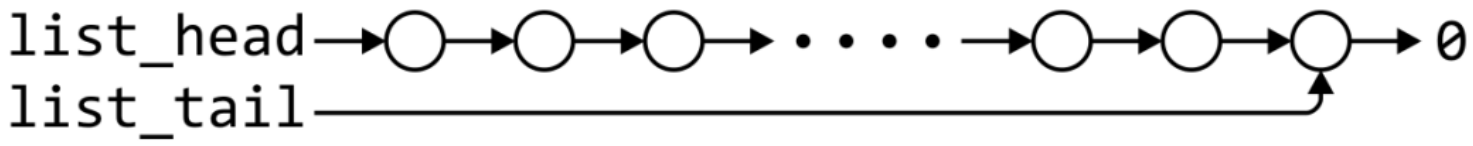
We will consider these for

- Singly linked lists
- Doubly linked lists

Singly linked list

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$O(n)$	$\Theta(n)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$O(n)$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

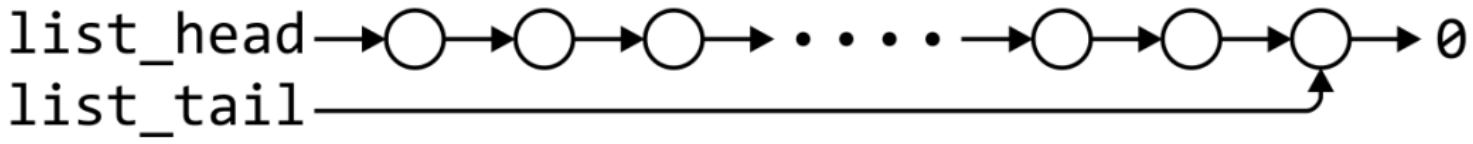
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Singly linked list

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

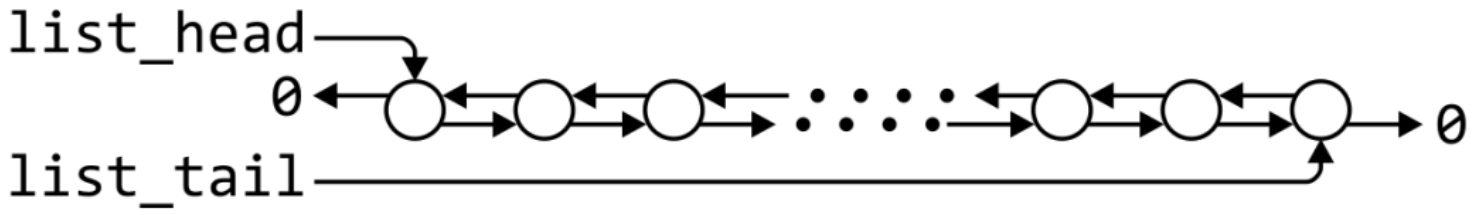
By replacing the value in the node in question, we can speed things up – useful for interviews



Doubly linked lists

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

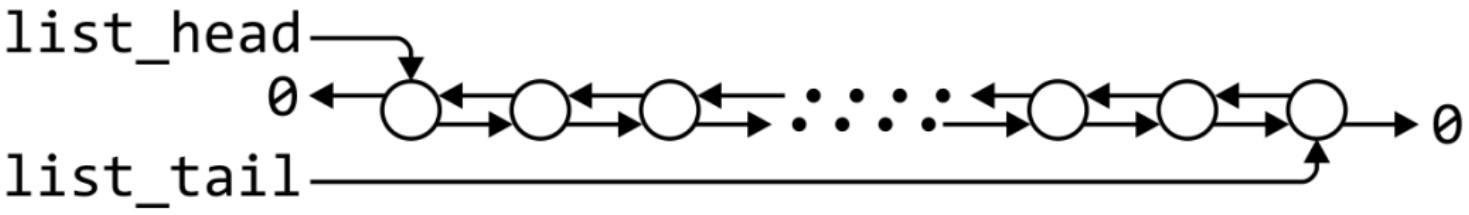
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Doubly linked lists

Accessing the k^{th} entry is $O(n)$

	k^{th} node
Insert Before	$\Theta(1)$
Insert After	$\Theta(1)$
Replace	$\Theta(1)$
Erase	$\Theta(1)$
Next	$\Theta(1)$
Previous	$\Theta(1)$



Other operations on linked lists

Other operations on linked lists include:

- Allocation and deallocating the memory requires $\Theta(n)$ time
- Concatenating two linked lists can be done in $\Theta(1)$
  - This requires a tail pointer

Arrays

We will consider these operations for arrays, including:

- Standard or one-ended arrays
- Two-ended arrays

Standard arrays

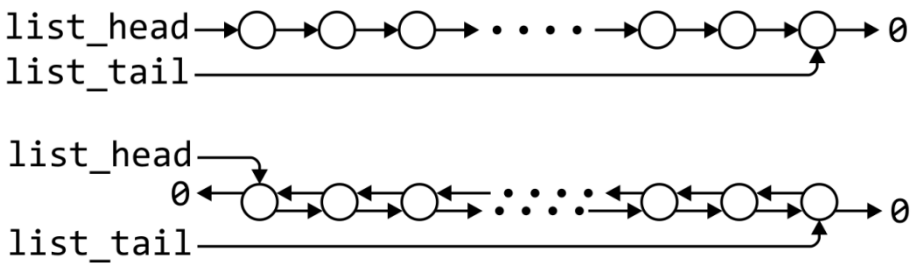
We will consider these operations for arrays, including:

- Standard or one-ended arrays
- Two-ended arrays

Run times

	Accessing the k^{th} entry	Insert or erase at the Front	Insert or erase at the k^{th} entry	Insert or erase at the Back
Singly linked lists	$O(n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$ or $\Theta(n)$
Doubly linked lists				$\Theta(1)$
Arrays	$\Theta(1)$	$\Theta(n)$	$O(n)$	$\Theta(1)$
Two-ended arrays		$\Theta(1)$		

* Assume we have a pointer to this node



Data Structures

In general, we will only use these basic data structures if we can restrict ourselves to operations that execute in $\Theta(1)$ time, as the only alternative is $\mathbf{O}(n)$ or $\Theta(n)$

Interview question: in a singly linked list, can you speed up the two $\mathbf{O}(n)$ operations of

- Inserting before an arbitrary node?
- Erasing any node that is not the last node?

If you can replace the contents of a node, the answer is “yes”

- Replace the contents of the current node with the new entry and insert after the current node
- Copy the contents of the next node into the current node and erase the next node

Memory usage versus run times

All of these data structures require $\Theta(n)$ memory

- Using a two-ended array requires one more member variable, $\Theta(1)$, in order to significantly speed up certain operations
- Using a doubly linked list, however, required $\Theta(n)$ additional memory to speed up other operations

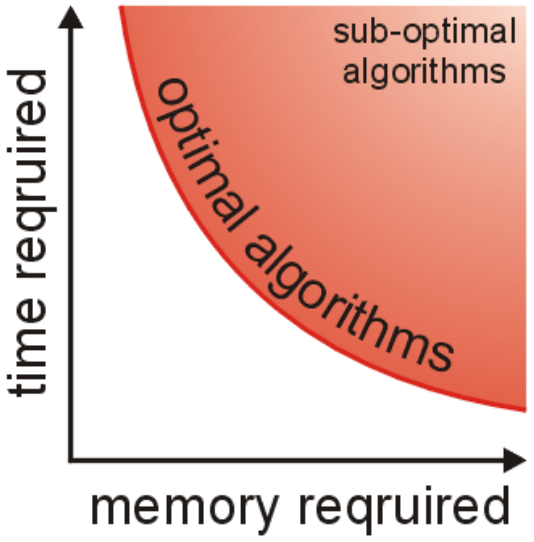
Memory usage versus run times

As well as determining run times, we are also interested in memory usage

In general, there is an interesting relationship between memory and time efficiency

For a data structure/algorithm:

- Improving the run time usually requires more memory
- Reducing the required memory usually requires more run time



Memory usage versus run times

Warning: programmers often mistake this to suggest that given any solution to a problem, any solution which may be faster must require more memory

This guideline not true in general: there may be different data structures and/or algorithms which are both faster and require less memory

- This requires thought and research

The `sizeof` Operator

In order to determine memory usage, we must know the memory usage of the various built-in data types and classes

- The `sizeof` operator in C++ returns the number of bytes occupied by a data type
- This value is determined at compile time
  - It is **not** a function

The `sizeof` Operator

```
#include <iostream>
using namespace std;

int main() {
     cout << "bool     " << sizeof( bool )     << endl;
     cout << "char     " << sizeof( char )     << endl;
     cout << "short    " << sizeof( short )    << endl;
     cout << "int      " << sizeof( int )      << endl;
     cout << "char *   " << sizeof( char * )   << endl;
     cout << "int *    " << sizeof( int * )    << endl;
     cout << "double   " << sizeof( double )   << endl;
     cout << "int[10]  " << sizeof( int[10] )  << endl;

     return 0;
}
```

```
{eceunix:1} ./a.out # output
bool     1
char     1
short    2
int      4
char *   4
int *    4
double   8
int[10]  40
{eceunix:2}
```

Abstract Strings

A specialization of an Abstract List is an Abstract String:

- The entries are restricted to *characters* from a finite *alphabet*
- This includes regular strings “Hello world!”

The restriction using an alphabet emphasizes specific operations that would seldom be used otherwise

- Substrings, matching substrings, string concatenations

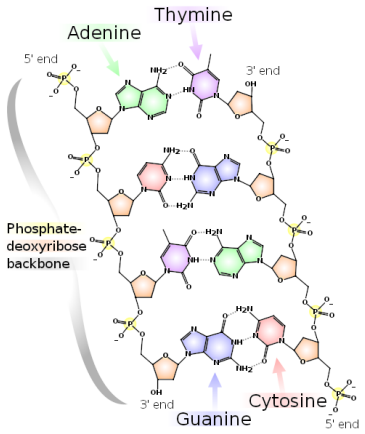
It also allows more efficient implementations

- String searching/matching algorithms
- Regular expressions

Abstract Strings

Strings also include DNA

- The alphabet has 4 *characters*: A, C, G, and T
- These are the nucleobases:
  adenine, cytosine, guanine, and thymine



Bioinformatics today uses many of the algorithms traditionally restricted to computer science:

- Dan Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge, 1997
  `http://books.google.ca/books?id=STGlsyqtjYMC`
- References:
  `http://en.wikipedia.org/wiki/DNA`
  `http://en.wikipedia.org/wiki/Bioinformatics`

Standard Template Library

In this course, you must understand each data structure and their associated algorithms

- In industry, you will use other implementations of these structures

The C++ Standard Template Library (STL) has an implementation of the `vector` data structure

- Excellent reference:

  http://www.cplusplus.com/reference/stl/vector/

Standard Template Library

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> v( 10, 0 );

    cout << "Is the vector empty?  " << v.empty() << endl;
    cout << "Size of vector:  "      << v.size()  << endl;

    v[0] = 42;
    v[9] = 91;

    for ( int k = 0; k < 10; ++k ) {
        cout << "v[" << k << "] = " << v[k] << endl;
    }

    return 0;
}
```

```
$ g++ vec.cpp
$ ./a.out
Is the vector empty?  0
Size of vector:  10
v[0] = 42
v[1] = 0
v[2] = 0
v[3] = 0
v[4] = 0
v[5] = 0
v[6] = 0
v[7] = 0
v[8] = 0
v[9] = 91
$
```

Summary

In this topic, we have introduced Abstract Lists

- Explicit linear orderings
- Implementable with arrays or linked lists
  - Each has their limitations
  - Introduced modifications to reduce run times down to $\Theta(1)$
- Discussed memory usage and the `sizeof` operator
- Looked at the String ADT
- Looked at the `vector` class in the STL

Outline

This topic discusses the concept of a stack:

- Description of an Abstract Stack
- List applications
- Implementation
- Example applications
  - Parsing: XHTML, C++
  - Function calls
  - Reverse-Polish calculators
  - Robert's Rules
- Standard Template Library

Abstract Stack

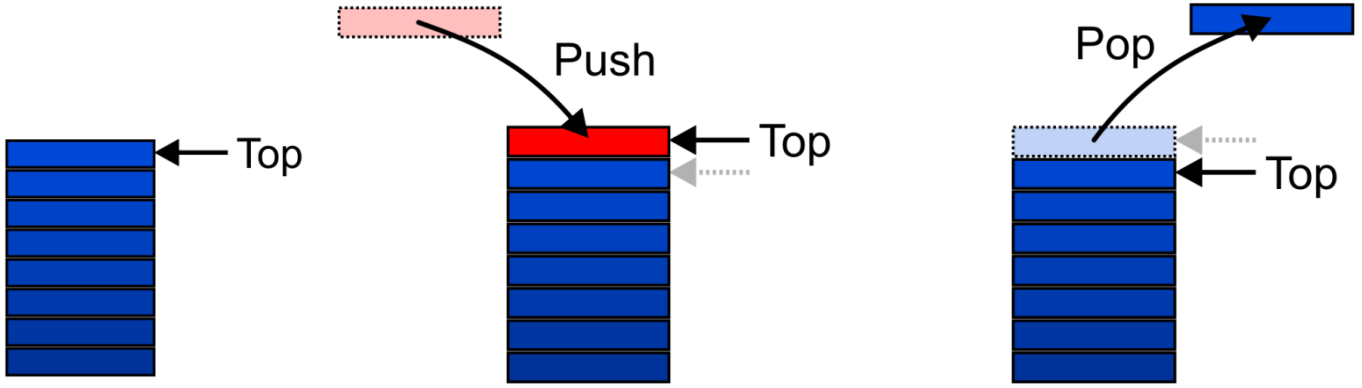
An Abstract Stack (Stack ADT) is an abstract data type which emphasizes specific operations:

- Uses a explicit linear ordering
- Insertions and removals are performed individually
- Inserted objects are *pushed onto* the stack
- The *top* of the stack is the most recently object pushed onto the stack
- When an object is *popped* from the stack, the current *top* is erased

Abstract Stack

Also called a *last-in–first-out* (LIFO) behaviour

– Graphically, we may view these operations as follows:



There are two exceptions associated with abstract stacks:

– It is an undefined operation to call either pop or top on an empty stack

Applications

Numerous applications:

- Parsing code:
  - Matching parenthesis
  - XML (e.g., XHTML)
- Tracking function calls
- Dealing with undo/redo operations
- Reverse-Polish calculators
- Assembly language

The stack is a very simple data structure

- Given any problem, if it is possible to use a stack, this significantly simplifies the solution

Stack: Applications

Problem solving:

- Solving one problem may lead to subsequent problems
- These problems may result in further problems
- As problems are solved, your focus shifts back to the problem which lead to the solved problem

Notice that function calls behave similarly:

- A function is a collection of code which solves a problem

Reference: Donald Knuth

Implementations

We will look at two implementations of stacks:

The optimal asymptotic run time of any algorithm is $\Theta(1)$

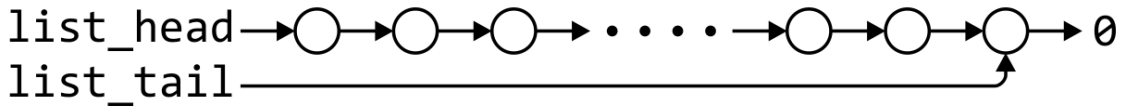
- The run time of the algorithm is independent of the number of objects being stored in the container
- We will always attempt to achieve this lower bound

We will look at

- Singly linked lists
- One-ended arrays

Linked-List Implementation

Operations at the front of a singly linked list are all $\Theta(1)$



	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 Stack may be reproduced by performing all operations at the front

Single_list Definition

The definition of single list class from Project 1 is:

```
template <typename Type>
class Single_list {
    public:
       Single_list();
        ~Single_list();

        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 & );
};
```

Stack-as-List Class

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

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

Stack-as-List Class

A constructor and destructor is not needed

- Because `list` is declared, the compiler will call the constructor of the `Single_list` class when the `Stack` is constructed

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

Stack-as-List Class

The empty and push functions just call the appropriate functions of the `Single_list` class

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

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

Stack-as-List Class

The top and pop functions, however, must check the boundary case:

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

    return list.front();
}
```

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

    return list.pop_front();
}
```

Array Implementation

For one-ended arrays, all operations at the back are $\Theta(1)$

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

Destructor

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 stack

```
int stack_size;
```

- The capacity of the array

```
int array_capacity;
```

Stack-as-Array Class

We need to store an array:

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

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

Constructor

The class is only storing the address of the array

- 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>
Stack<Type>::Stack( int n ):
stack_size( 0 ),
array_capacity( std::max( 1, n ) ),
array( new Type[array_capacity] ) {
    // Empty constructor
}
```

Constructor

Warning: in C++, the variables are initialized in the order in which they are defined:

```
template <typename Type>
Stack<Type>::Stack( int n ):
stack_size( 0 ),
array_capacity( std::max( 1, n ) ),
array( new Type[array_capacity] ) {
    // Empty constructor
}
```

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

Destructor

The call to new in the constructor requested memory from the operating system

- The destructor must return that memory to the operating system:

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

Empty

The stack is empty if the stack size is zero:

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

The following is unnecessarily tedious:

- The == operator evaluates to either `true` or `false`

```
if ( stack_size == 0 ) {
    return true;
} else {
    return false;
}
```

Top

If there are n objects in the stack, the last is located at index $n-1$

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

    return array[stack_size - 1];
}
```

Pop

Removing an object simply involves reducing the size

- It is invalid to assign the last entry to “0”
- By decreasing the size, the previous top of the stack is now at the location `stack_size`

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

    --stack_size;
    return array[stack_size];
}
```

Push

Pushing an object onto the stack can only be performed if the array is not full

```
template <typename Type>
void Stack<Type>::push( Type const &obj ) {
    if ( stack_size == array_capacity ) {
        throw overflow();  // Best solution?????
    }

    array[stack_size] = obj;
    ++stack_size;
}
```

Exceptions

The case where the array is full is not an exception defined in the Abstract Stack

If the array is filled, we have five options:

- Increase the size of the array
- Throw an exception
- Ignore the element being pushed
- Replace the current top of the stack
- Put the pushing process to “sleep” until something else removes the top of the stack

Include a member function `bool full() const;`

Array Capacity

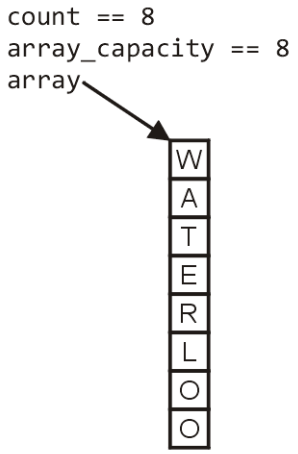
If dynamic memory is available, the best option is to increase the array capacity

If we increase the array capacity, the question is:

- How much?
- By a constant? `array_capacity += c;`
- By a multiple? `array_capacity *= c;`

Array Capacity

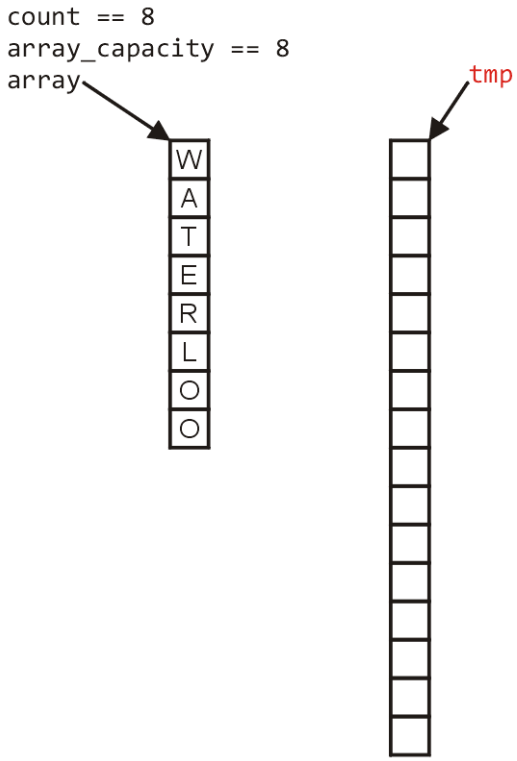
First, let us visualize what must occur to allocate new memory



Array Capacity

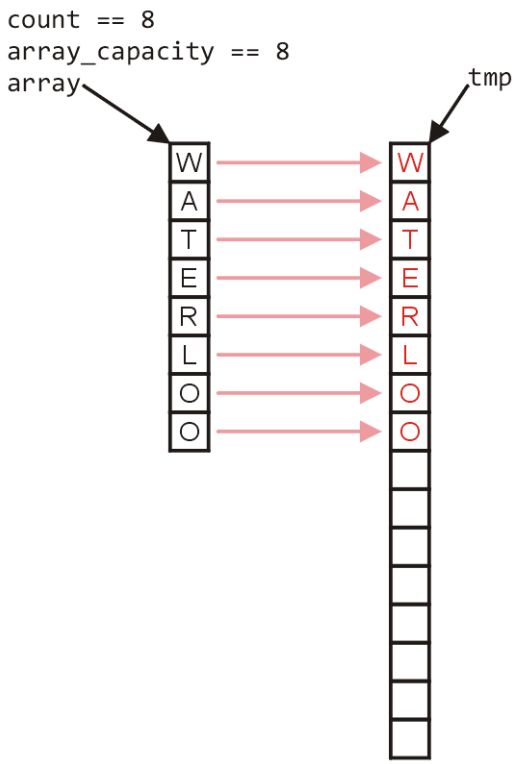
First, this requires a call to `new Type[N]` where *N* is the new capacity

- We must have access to this so we must store the address returned by new in a local variable, say `tmp`



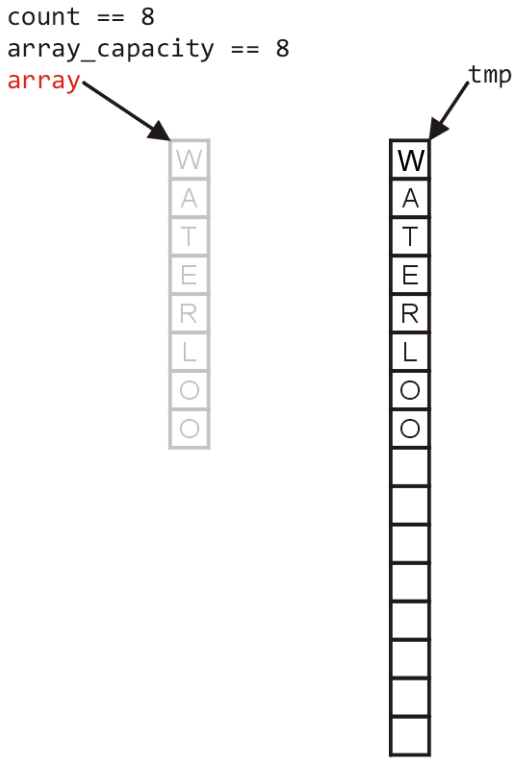
Array Capacity

Next, the values must be copied over



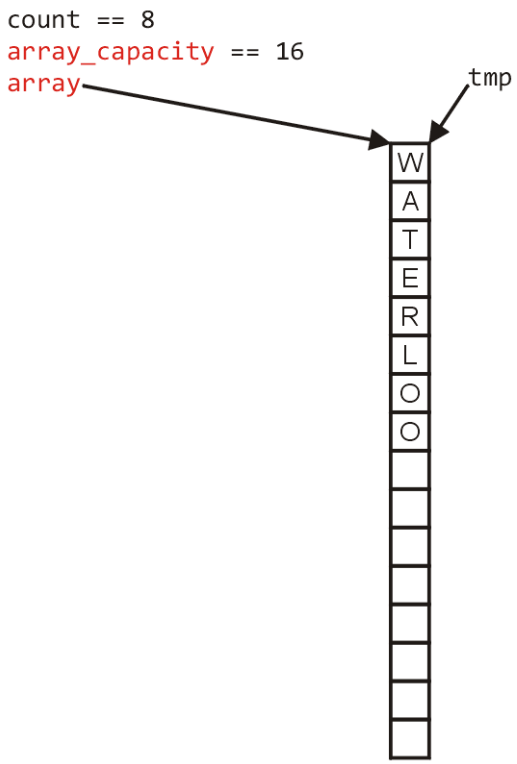
Array Capacity

The memory for the original array must be deallocated



Array Capacity

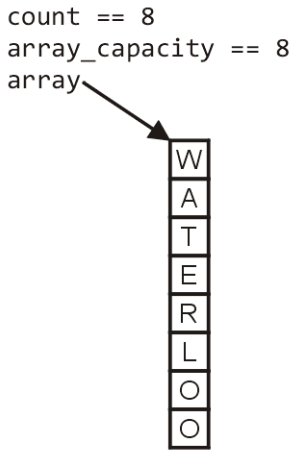
Finally, the appropriate member variables must be reassigned



Array Capacity

The implementation:

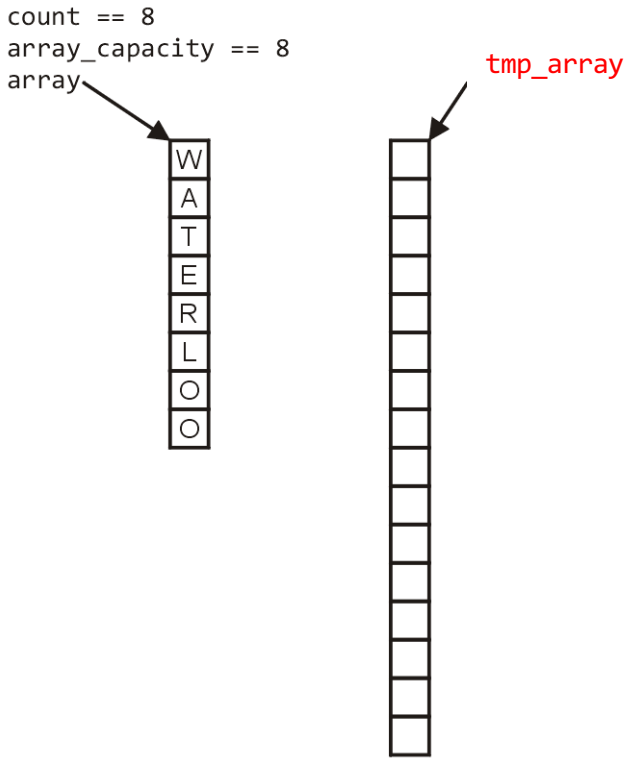
```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];

}
```



Array Capacity

The implementation:

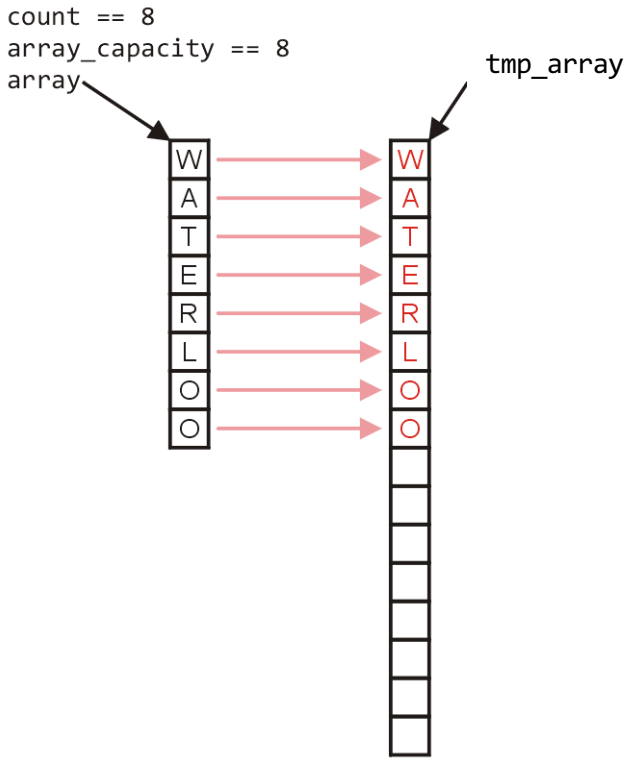
```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];




}
```



Array Capacity

The implementation:

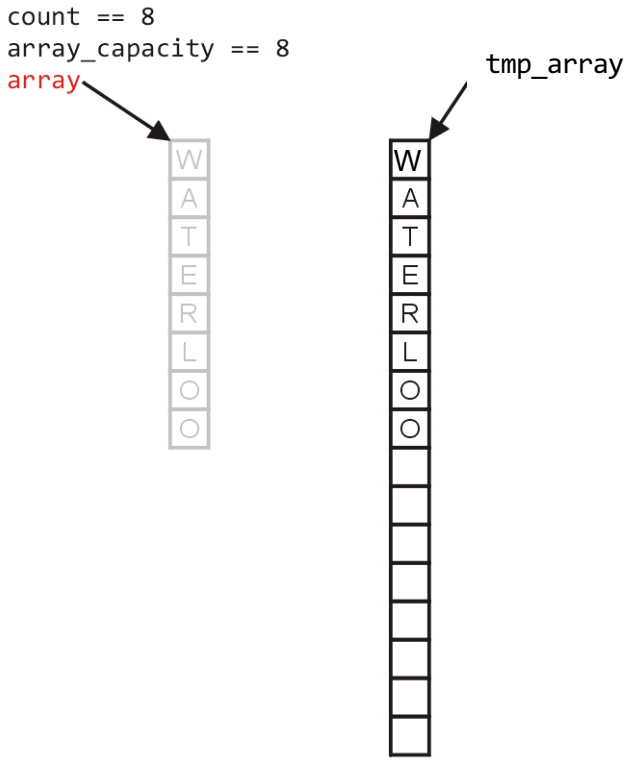
```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];

    for ( int i = 0; i < array_capacity; ++i ) {
        tmp_array[i] = array[i];
    }


}
```



Array Capacity

The implementation:

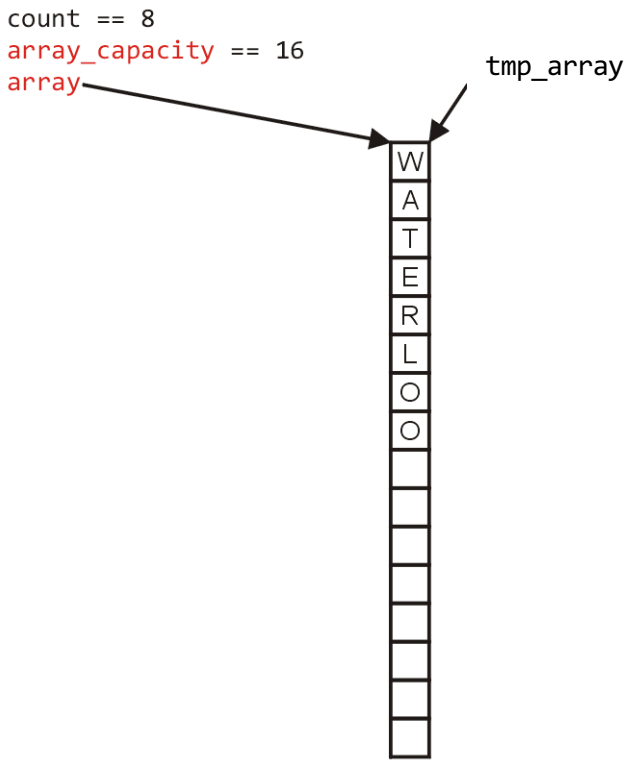
```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];

    for ( int i = 0; i < array_capacity; ++i ) {
        tmp_array[i] = array[i];
    }

    delete [] array;


}
```



Array Capacity

The implementation:

```
void double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];

    for ( int i = 0; i < array_capacity; ++i ) {
        tmp_array[i] = array[i];
    }

    delete [] array;
    array = tmp_array;

    array_capacity *= 2;
}
```



Array Capacity

Back to the original question:

- How much do we change the capacity?
- Add a constant?
- Multiply by a constant?

First, we recognize that any time that we `push` onto a full stack, this requires n copies and the run time is $\Theta(n)$

Therefore, `push` is usually $\Theta(1)$ except when new memory is required

Array Capacity

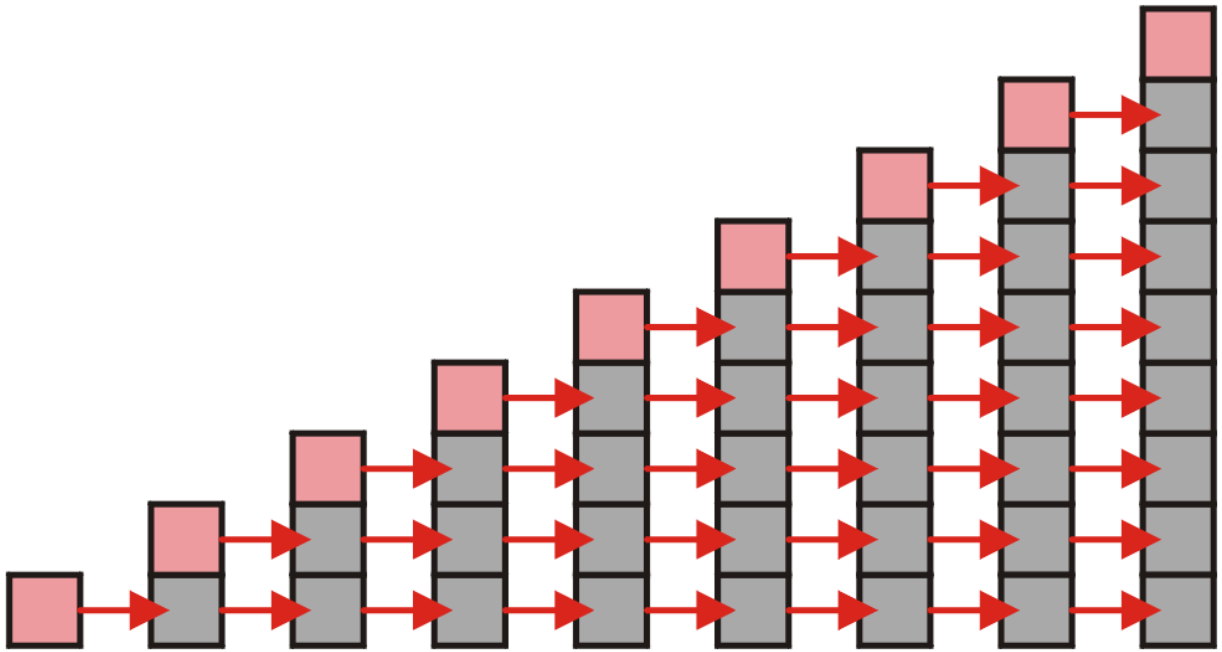
To state the average run time, we will introduce the concept of amortized time:

- If n operations requires $\Theta(f(n))$, we will say that an individual operation has an amortized run time of $\Theta(f(n)/n)$
- Therefore, if inserting n objects requires:
  - $\Theta(n^2)$ copies, the amortized time is $\Theta(n)$
  - $\Theta(n)$ copies, the amortized time is $\Theta(1)$

Array Capacity

Let us consider the case of increasing the capacity by 1 each time the array is full

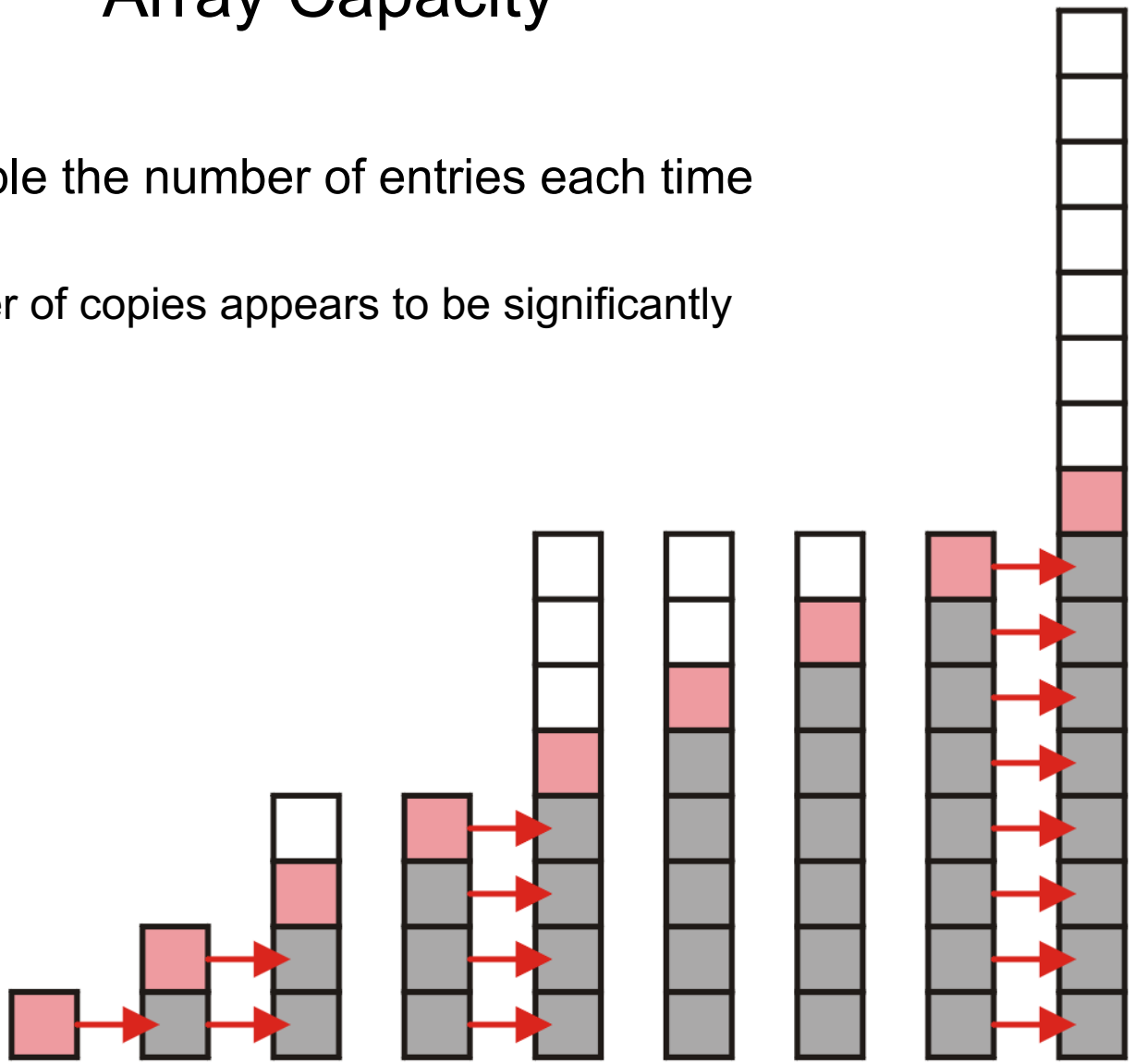
- With each insertion when the array is full, this requires all entries to be copied



Array Capacity

Suppose we double the number of entries each time the array is full

- Now the number of copies appears to be significantly fewer



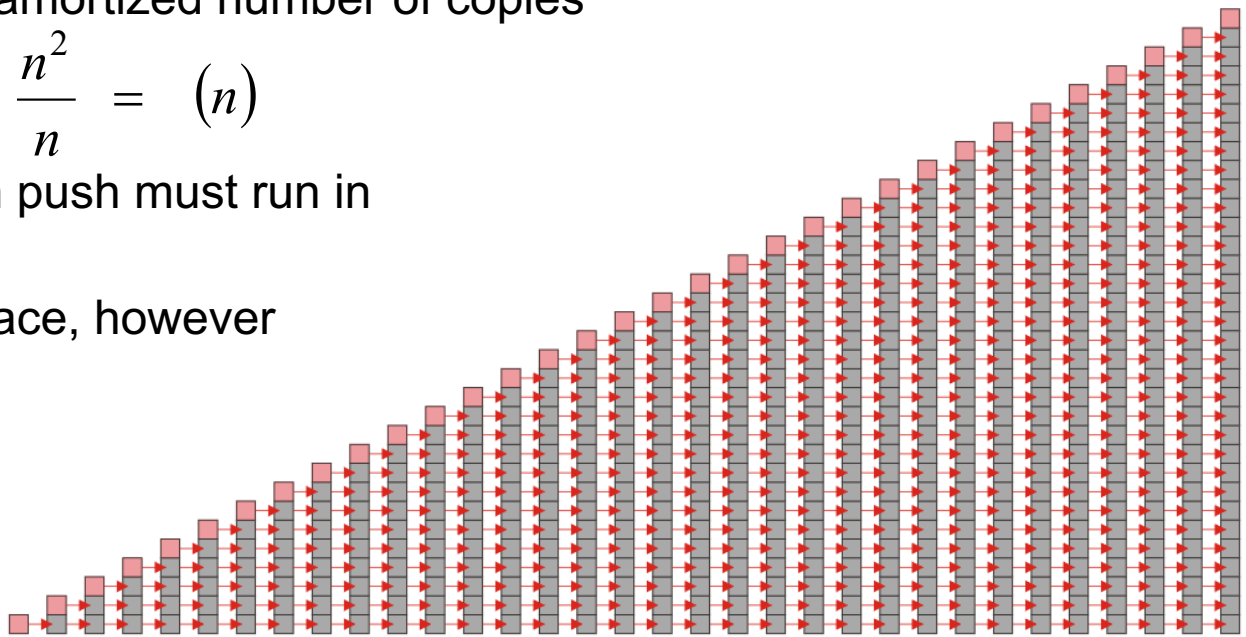
Array Capacity

Suppose we insert k objects

- The pushing of the k^{th} object on the stack requires k copies
- The total number of copies is now given by:

$$\sum_{k=1}^{n}(k \quad 1) = \sum_{k=1}^{n} k \qquad n = \frac{n(n+1)}{2} \qquad n = \frac{n(n \quad 1)}{2} = \quad \left(n^2\right)$$

- Therefore, the amortized number of copies is given by $\dfrac{n^2}{n} \quad = \quad (n)$
- Therefore each push must run in $\Theta(n)$ time
- The wasted space, however is $\Theta(1)$



Array Capacity

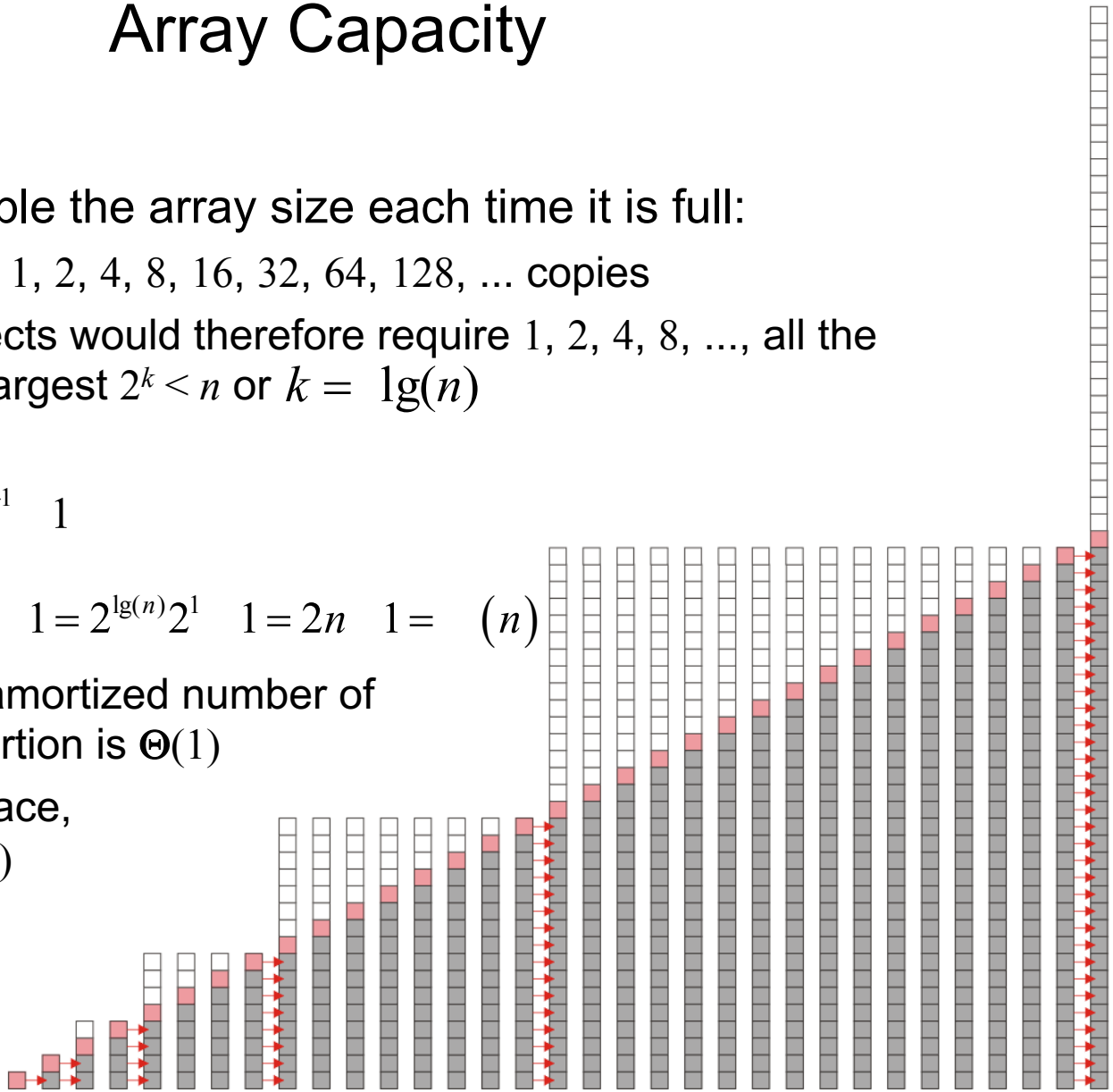
Suppose we double the array size each time it is full:

- We must make $1, 2, 4, 8, 16, 32, 64, 128, \ldots$ copies
- Inserting n objects would therefore require $1, 2, 4, 8, \ldots$, all the way up to the largest $2^k < n$ or $k = \lg(n)$

$$\sum_{k=0}^{\lg(n)} 2^k = 2^{\lg(n)+1} - 1$$

$$2^{\lg(n)+1} - 1 = 2^{\lg(n)} 2^1 - 1 = 2n - 1 = \Theta(n)$$

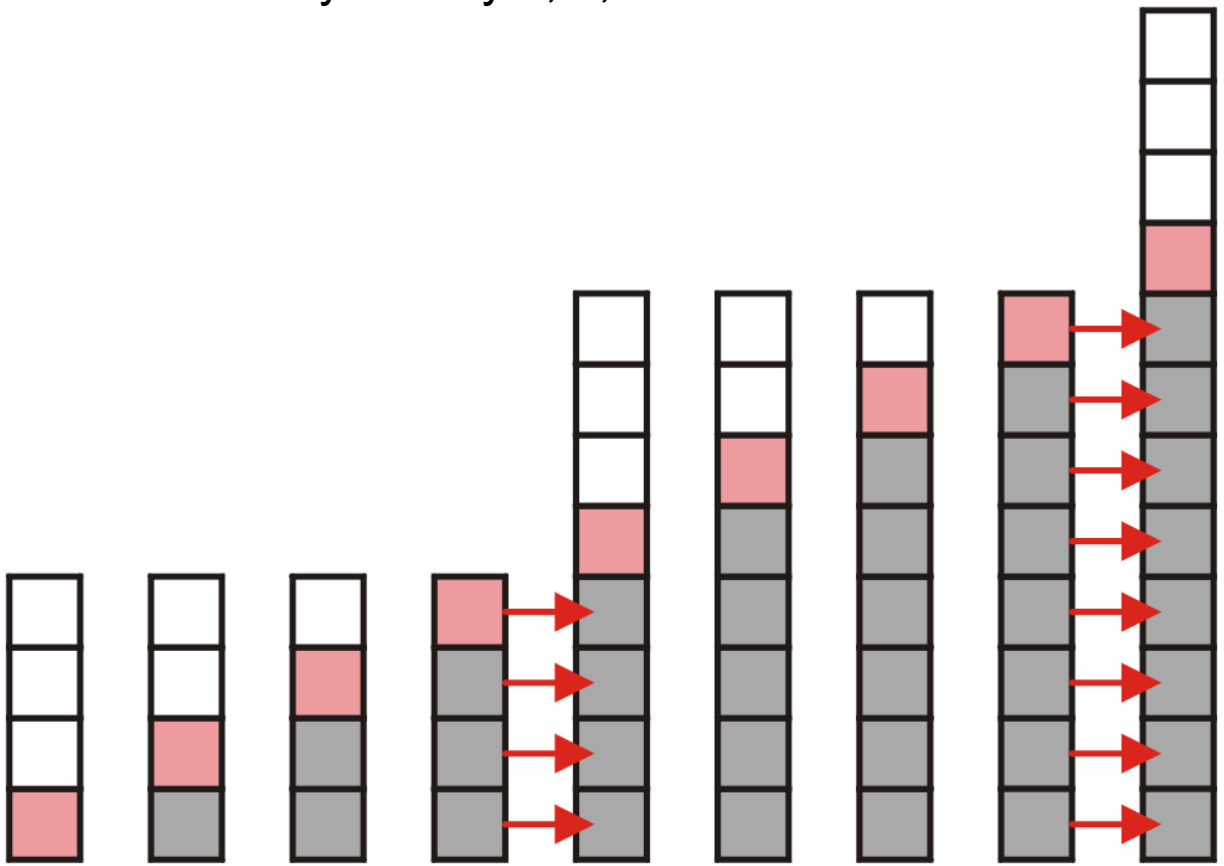
- Therefore the amortized number of copies per insertion is $\Theta(1)$
- The wasted space, however is $\mathbf{O}(n)$



Array Capacity

What if we increase the array size by a larger constant?

– For example, increase the array size by 4, 8, 100?

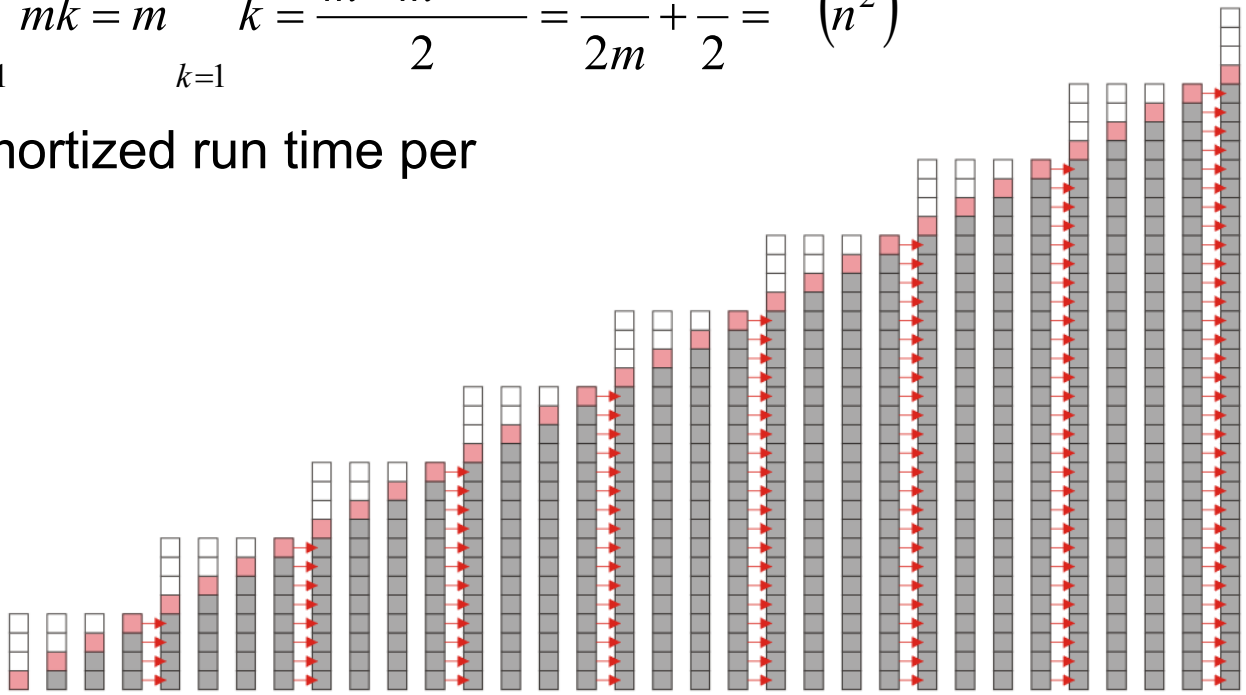


Array Capacity

Here we view the number of copies required when increasing the array size by 4; however, in general, suppose we increase it by a **constant** value m

$$\sum_{k=1}^{n/m} mk = m \sum_{k=1}^{n/m} k = \frac{\frac{n}{m}\left(\frac{n}{m}+1\right)}{2} = \frac{n^2}{2m} + \frac{n}{2} = \Theta\left(n^2\right)$$

Therefore, the amortized run time per insertion is $\Theta(n)$



Array Capacity

Note the difference in worst-case amortized scenarios:

	Copies per Insertion	Unused Memory
Increase by 1	$n-1$	0
Increase by m	n/m	$m-1$
Increase by a factor of 2	1	n
Increase by a factor of $r>1$	$1/(r-1)$	$(r-1)n$

The web site

http://www.ece.uwaterloo.ca/~ece250/Algorithms/Array_resizing/

discusses the consequences of various values of r

Application: Parsing

Most parsing uses stacks

Examples includes:

- Matching tags in XHTML
- In C++, matching
  - parentheses `( ... )`
  - brackets, and `[ ... ]`
  - braces `{ ... }`

Parsing XHTML

The first example will demonstrate parsing XHTML

We will show how stacks may be used to parse an XHTML document

You will use XHTML (and more generally XML and other markup languages) in the workplace

Parsing XHTML

A *markup language* is a means of annotating a document to given context to the text

- The annotations give information about the structure or presentation of the text

The best known example is HTML, or HyperText Markup Language

- We will look at XHTML

Parsing XHTML

XHTML is made of nested

- *opening tags*, e.g., `<some_identifier>`, and
- matching *closing tags*, e.g., `</some_identifier>`

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

Parsing XHTML

Nesting indicates that any closing tag must match the most recent opening tag

Strategy for parsing XHTML:

- read though the XHTML linearly
- place the opening tags in a stack
- when a closing tag is encountered, check that it matches what is on top of the stack and

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```

<html>			

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```

<html>	<head>		

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```

<html>	<head>	<title>	

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```

<html>	<head>	<title>	

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```

<html>	<head>		

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```

<html>	<body>		

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	<i>

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	<i>

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	

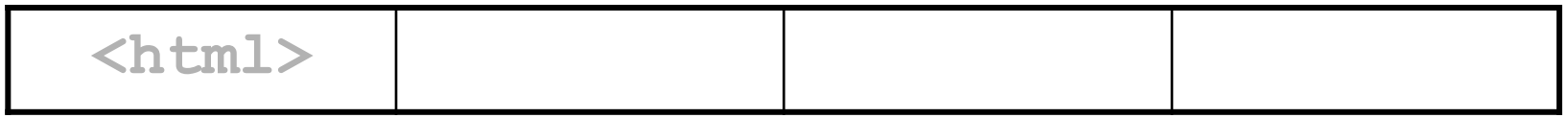
Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```

<html>	<body>		

Parsing XHTML

```
<html>
   <head><title>Hello</title></head>
   <body><p>This appears in the
   <i>browser</i>.</p></body>
</html>
```



Parsing XHTML

We are finished parsing, and the stack is empty

Possible errors:

- a closing tag which does not match the opening tag on top of the stack
- a closing tag when the stack is empty
- the stack is not empty at the end of the document

HTML

Old HTML required neither closing tags nor nesting

```
<html>
  <head><title>Hello</title></head>
  <body><p>This is a list of topics:
  <ol>                          <!-- para ends with start of list -->
    <li><i>veni                              <!-- implied </li> -->
    <li>vidi                               <!-- italics continues -->
    <li>vici</i>
  </ol>                  <!-- end-of-file implies </body></html> -->
```

Parsers were therefore specific to HTML

- Results: ambiguities and inconsistencies

XML

XHTML is an implementation of XML

XML defines a class of general-purpose *eXtensible Markup Languages* designed for sharing information between systems

The same rules apply for any flavour of XML:

- opening and closing tags must match and be nested

Parsing C++

The next example shows how stacks may be used in parsing C++

Those taking ECE 351 *Compilers* will use this

For other students, it should help understand, in part:

- how a compiler works, and
- why programming languages have the structure they do

Parsing C++

Like opening and closing tags, C++ parentheses, brackets, and braces must be similarly nested:

```
void initialize( int *array, int n ) {
    for ( int i = 0; i < n; ++i ) {
        array[i] = 0;
    }
}
```

http://xkcd.com/859/

Parsing C++

For C++, the errors are similar to that for XHTML, however:

- many XHTML parsers usually attempt to “correct” errors (e.g., insert missing tags)
- C++ compilers will simply issue a parse error:

```
{eceunix:1} cat example1.cpp
#include <vector>
int main() {
        std::vector<int> v(100];
        return 0;
}

{eceunix:2} g++ example1.cpp
example1.cpp: In function 'int main()':
example1.cpp:3: error: expected ')' before ']' token
```

Parsing C++

For C++, the errors are similar to that for XHTML, however:

- many XHTML parsers usually attempt to “correct” errors (*e.g.*, insert missing tags)
- C++ compilers will simply issue a parse error:

```
{eceunix:1} cat example2.cpp
#include <vector>
int main() {
        std::vector<int> v(100);
        v[0] = 3];
        return 0;
}
{eceunix:2} g++ example2.cpp
example2.cpp: In function 'int main()':
example2.cpp:4: error: expected ';' before ']' token
```

Function Calls

This next example discusses function calls

In ECE 222 *Digital Computers*, you will see how stacks are implemented in hardware on all CPUs to facilitate function calling

The simple features of a stack indicate why almost all programming languages are based on function calls

Function Calls

Function calls are similar to problem solving presented earlier:

- you write a function to solve a problem
- the function may require sub-problems to be solved, hence, it may call another function
- once a function is finished, it returns to the function which called it

Function Calls

You will notice that the when a function returns, execution and the return value is passed back to the last function which was called

This is again, the last-in—first-out property

- Covered in much greater detail in ECE 222

Today's CPUs have hardware specifically designed to facilitate function calling

Reverse-Polish Notation

Normally, mathematics is written using what we call *in-fix* notation:

$$(3 + 4) \times 5 - 6$$

The operator is placed between to operands

One weakness: parentheses are required

$$(3 + 4) \times 5 - 6 = 29$$

$$3 + 4 \times 5 - 6 = 17$$

$$3 + 4 \times (5 - 6) = -1$$

$$(3 + 4) \times (5 - 6) = -7$$

Reverse-Polish Notation

Alternatively, we can place the operands first, followed by the operator:

$$(3 + 4) \times 5 - 6$$

$$3\ 4\ +\ 5\ \times\ 6\ -$$

Parsing reads left-to-right and performs any operation on the last two operands:

$$3\ 4\ +\ 5\ \times\ 6\ -$$

$$7\quad 5\ \times\ 6\ -$$

$$35\quad 6\ -$$

$$29$$

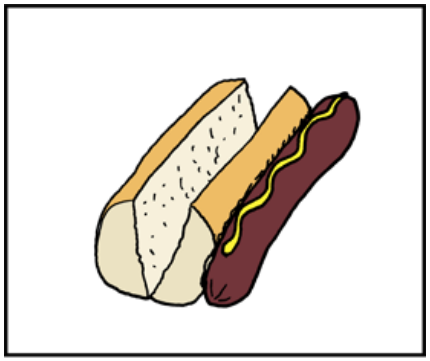
Reverse-Polish Notation

This is called *reverse-Polish* notation after the mathematician Jan Łukasiewicz

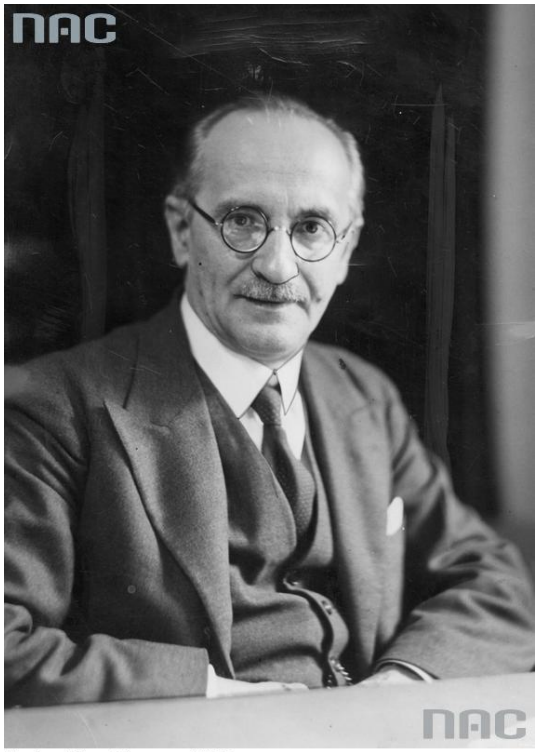
- As you will see in ECE 222, this forms the basis of the recursive stack used on all processors

He also made significant contributions to logic and other fields

- Including humour…



http://xkcd.com/645/



http://www.audiovis.nac.gov.pl/

Reverse-Polish Notation

Other examples:

$$
\begin{array}{cccccccc}
3 & 4 & 5 & \times & + & 6 & - & \\
3 & & 20 & & + & 6 & - & \\
 & & 23 & & & 6 & - & \\
 & & & & & 17 & & \\
3 & 4 & 5 & 6 & - & \times & + & \\
3 & 4 & & -1 & & \times & + & \\
3 & & & -4 & & & + & \\
 & & & -1 & & & &
\end{array}
$$

Reverse-Polish Notation

Benefits:

- No ambiguity and no brackets are required
- It is the same process used by a computer to perform computations:
  - operands must be loaded into registers before operations can be performed on them
- Reverse-Polish can be processed using stacks

Reverse-Polish Notation

Reverse-Polish notation is used with some programming languages

- *e.g.*, postscript, pdf, and HP calculators

Similar to the thought process required for writing assembly language code

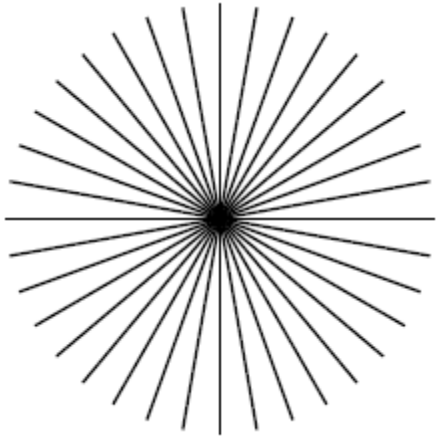
- you cannot perform an operation until you have all of the operands loaded into registers

```
MOVE.L #$2A, D1       ; Load  42 into Register D1
MOVE.L #$100, D2      ; Load 256 into Register D2
ADD D2, D1            ; Add D2 into D1
```

Reverse-Polish Notation

A quick example of postscript:

```
0 10 360 {                  % Go from 0 to 360 degrees in 10-degree steps
    newpath                 % Start a new path
    gsave                   % Keep rotations temporary
        144 144 moveto
        rotate              % Rotate by degrees on stack from 'for'
        72 0 rlineto
        stroke
    grestore                % Get back the unrotated state
} for % Iterate over angles
```



http://www.tailrecursive.org/postscript/examples/rotate.html

Reverse-Polish Notation

The easiest way to parse reverse-Polish notation is to use an operand stack:

- operands are processed by pushing them onto the stack
- when processing an operator:
  - pop the last two items off the operand stack,
  - perform the operation, and
  - push the result back onto the stack

Reverse-Polish Notation

Evaluate the following reverse-Polish expression using a stack:

$$1\ \ 2\ \ 3\ +\ 4\ \ 5\ \ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ \ 9\ \times\ +$$

Reverse-Polish Notation

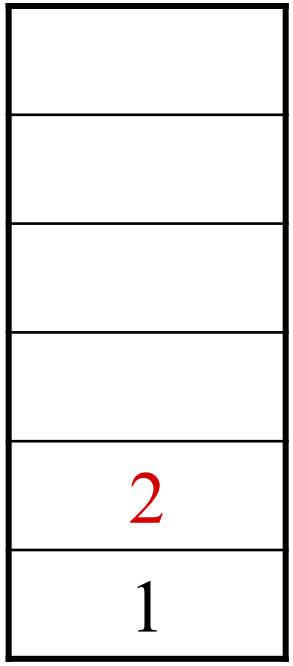
Push 1 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

Reverse-Polish Notation

Push 1 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Queues

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:

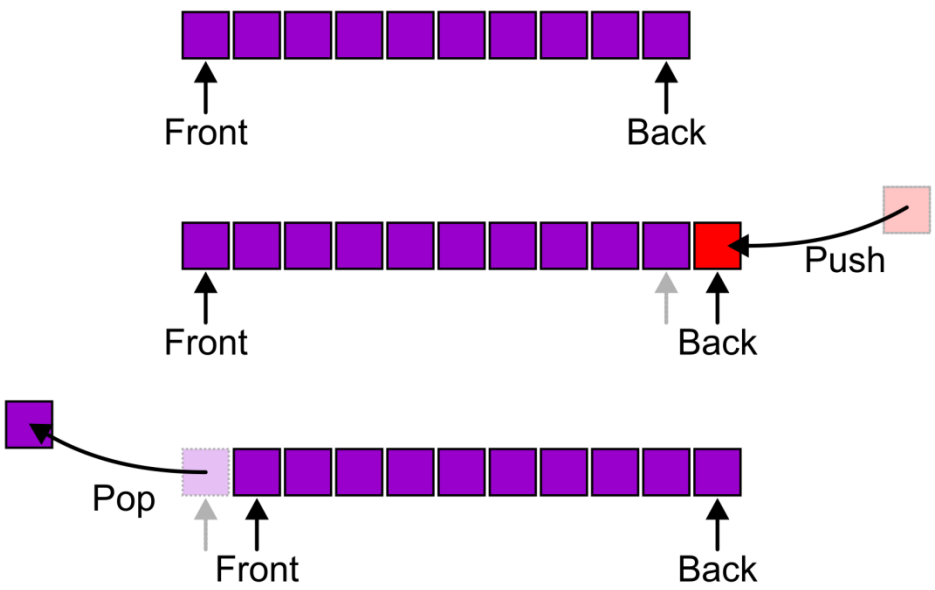
- Uses a explicit linear ordering
- Insertions and removals are performed individually
- There are no restrictions on objects inserted into (*pushed onto*) the queue—that object is designated the back of the queue
- The object designated as the *front* of the queue is the object which was in the queue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

3.3.1

Abstract Queue

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

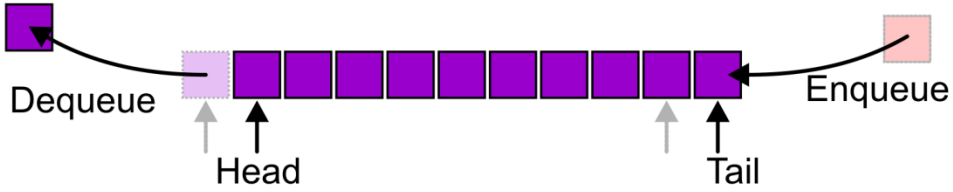
– Graphically, we may view these operations as follows:



3.3.1

Abstract Queue

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



3.3.1

Abstract 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

3.3.2

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

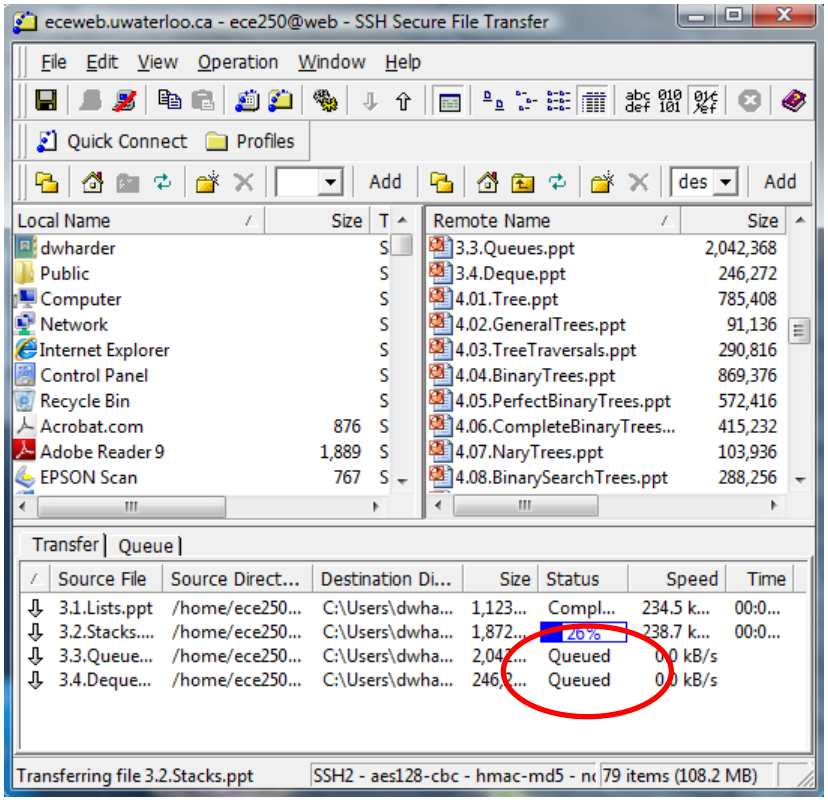
Most shared computer services are servers:

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

3.3.2

Applications

For example, in downloading these presentations from the ECE 250 web server, those requests not currently being downloaded are marked as “Queued”



3.3.3

Implementations

We will look at two implementations of queues:

- Singly linked lists
- Circular arrays

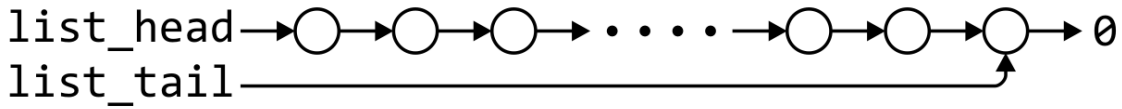
Requirements:

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

3.3.3.1

Linked-List Implementation

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

3.3.3.1

Single_list Definition

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 & );
};
```

3.3.3.1

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();
};
```

3.3.3.1

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();
}
```

3.3.3.2

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	$\Theta(n)$	$\Theta(1)$

3.3.3.2

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	$\Theta(1)$	$\Theta(1)$

3.3.3.2

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;
```

3.3.3.2

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();
};
```

3.3.3.2

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

3.3.3.2

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
}
```

3.3.3.2

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();
};
```

3.3.3.2

Destructor

The destructor is unchanged from Stack-as-Array:

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

3.3.3.2

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];
}
```

3.3.3.2

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;
}
```

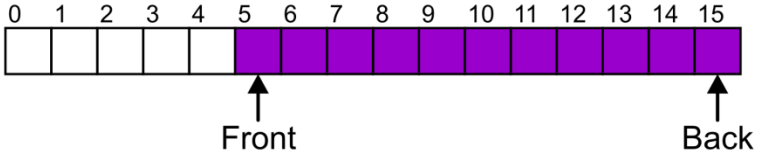
```
template <typename Type>
Type Queue<Type>::pop() {
    if ( empty() ) {
        throw underflow();
    }

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

3.3.3.2

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

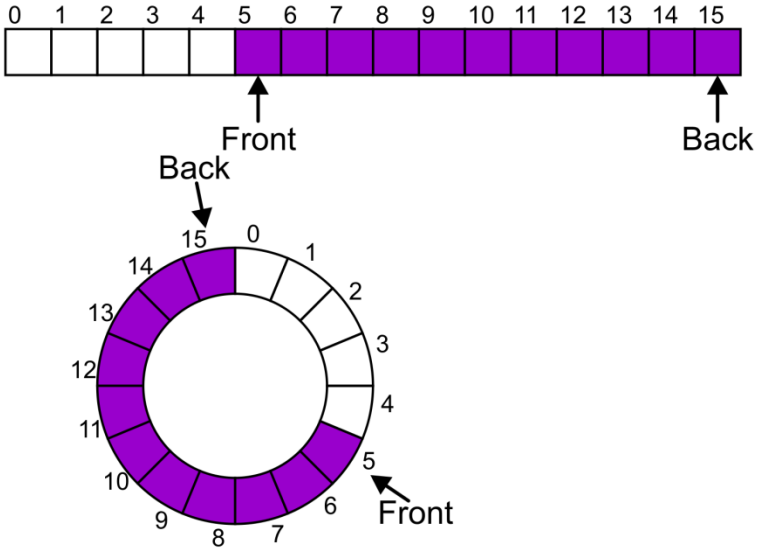
3.3.3.2

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*

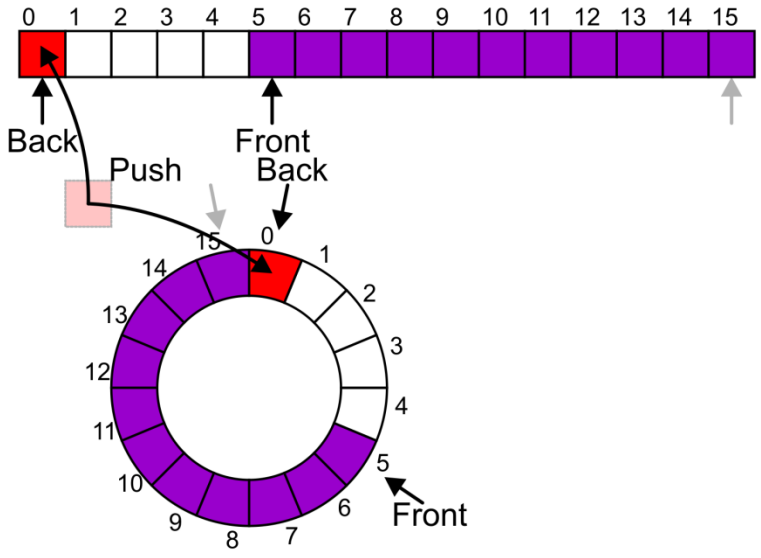


3.3.3.2

Member Functions

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

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



3.3.3.2

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

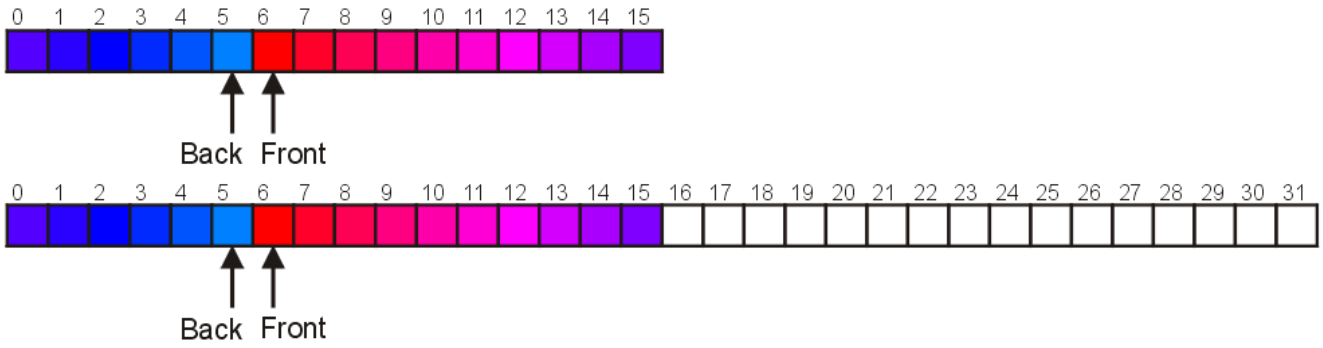
Include a member function **`bool full()`**

3.3.4

Increasing Capacity

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

- A direct copy does not work:

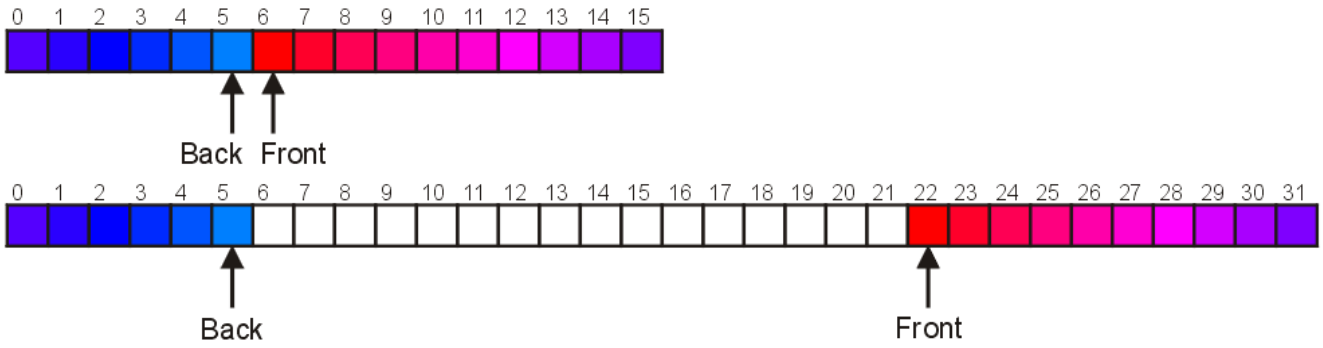


3.3.4

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

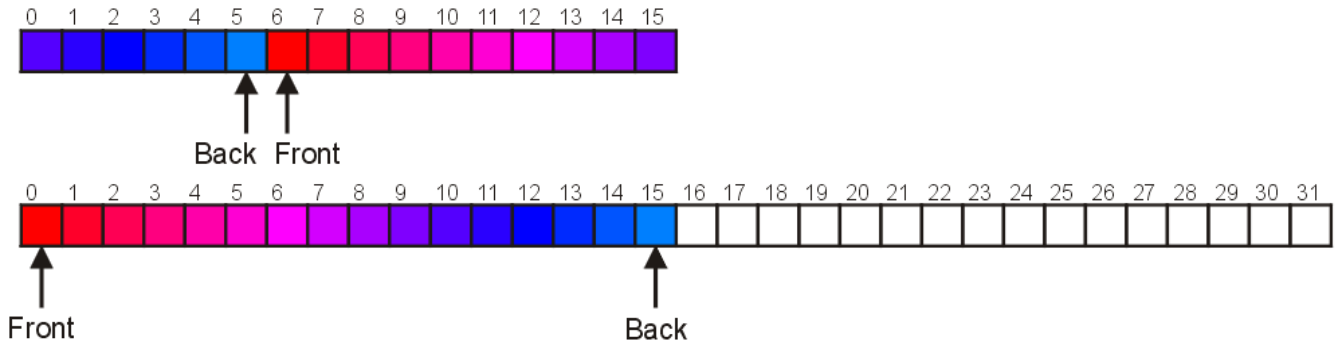


3.3.4

Increasing Capacity

An alternate solution is normalization:

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

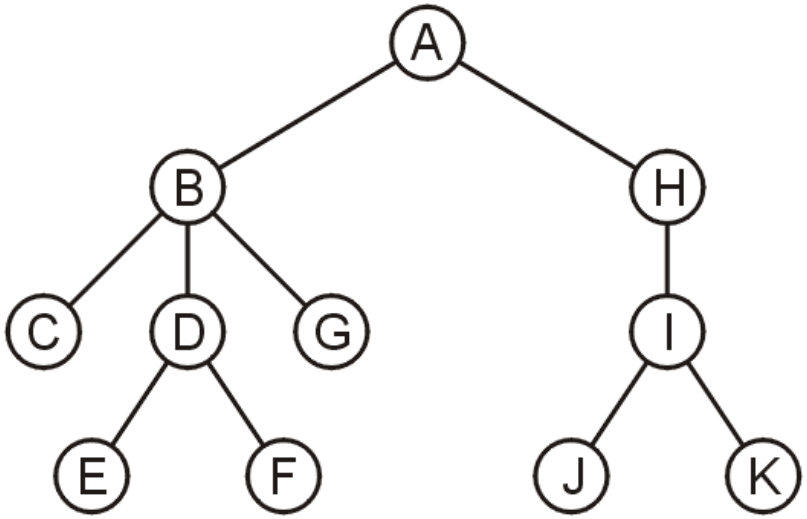


3.3.5

Application

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

- Consider searching the directory structure



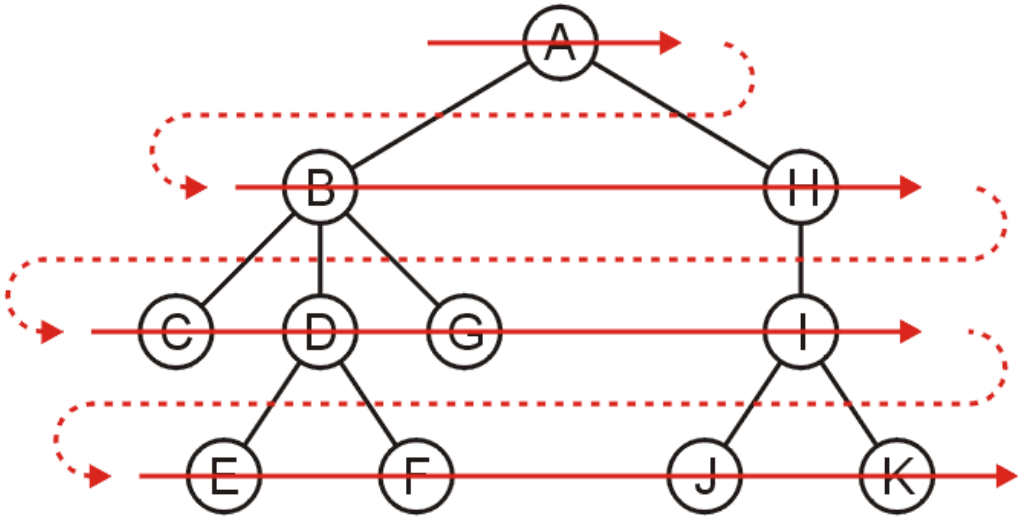
3.3.5

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



3.3.5

Application

The easiest implementation is:

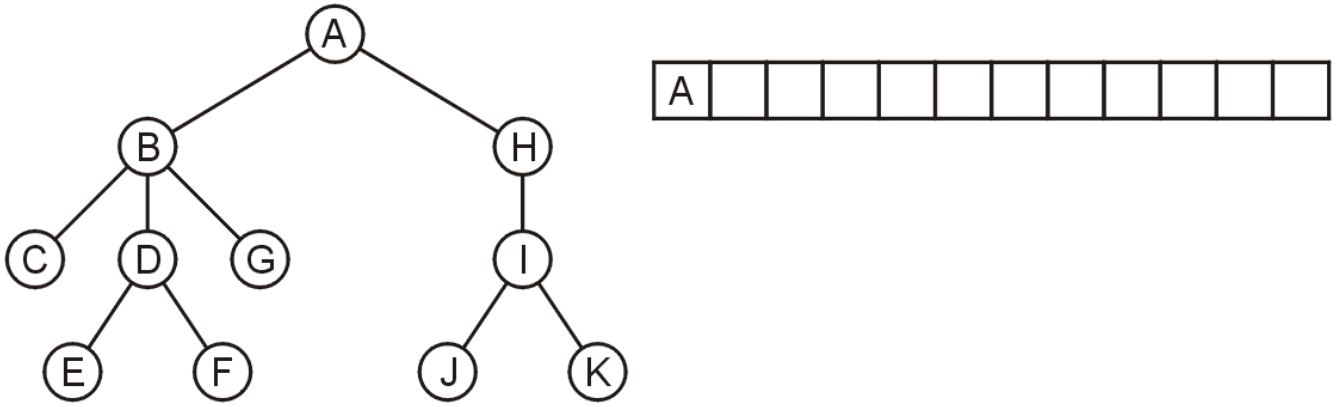
- Place the root directory into a queue
- While the queue is not empty:
  - Pop the directory at the front of the queue
  - Push all of its sub-directories into the queue

The order in which the directories come out of the queue will be in breadth-first order

3.3.5

Application

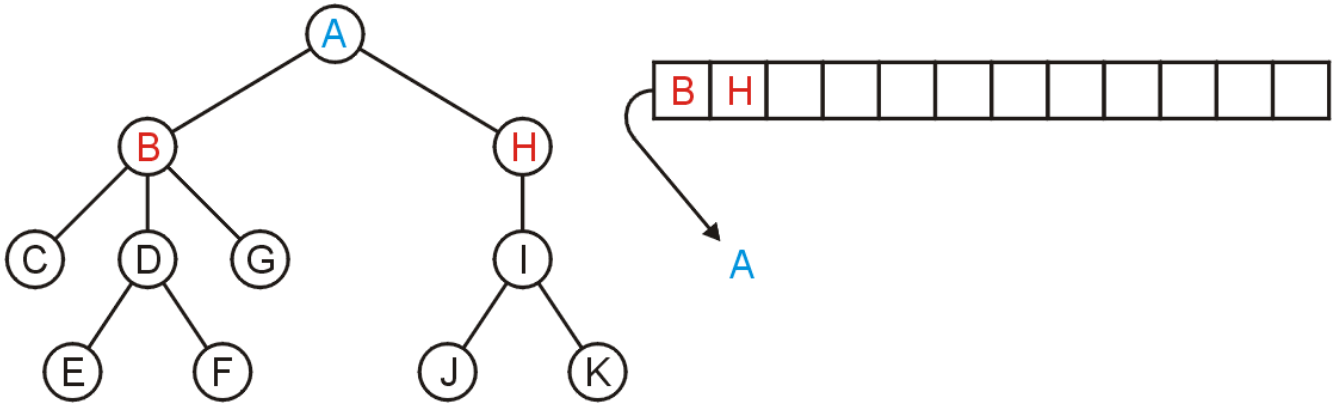
Push the root directory A



3.3.5

Application

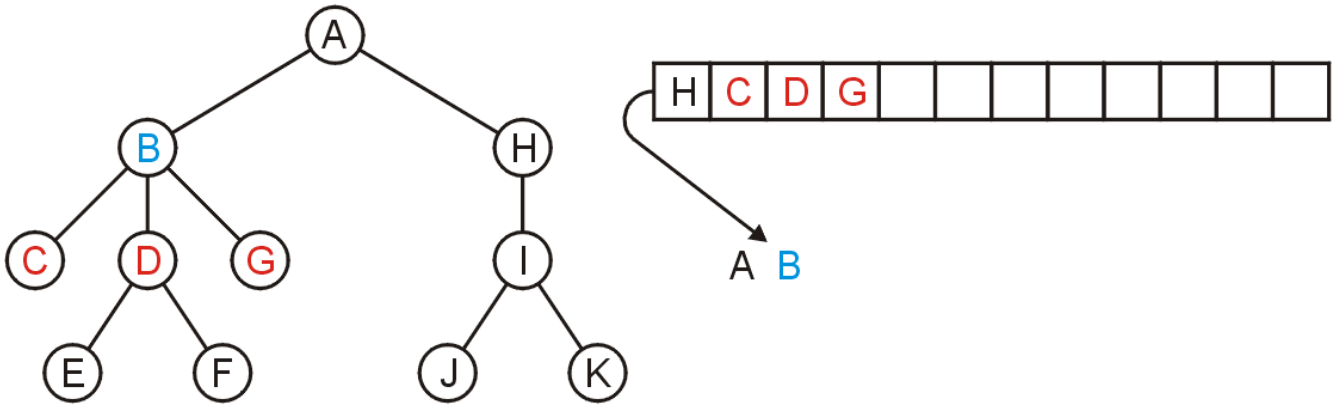
Pop A and push its two sub-directories: B and H



3.3.5

Application

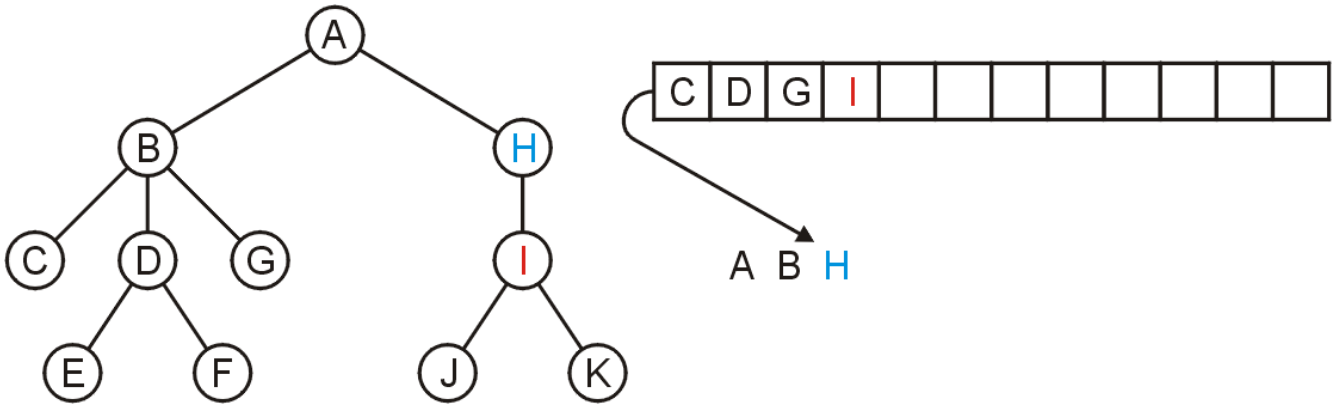
Pop B and push C, D, and G



3.3.5

Application

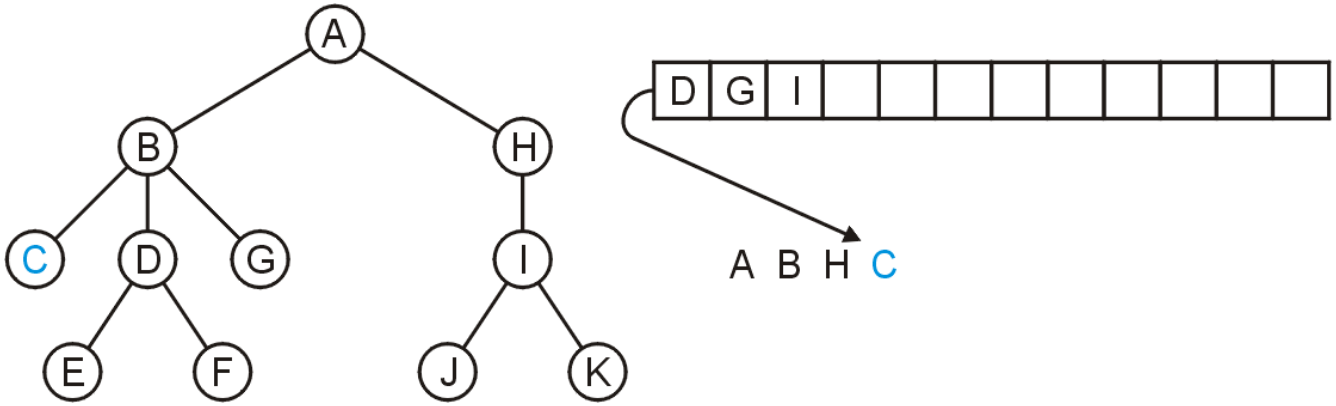
Pop H and push its one sub-directory I



3.3.5

Application

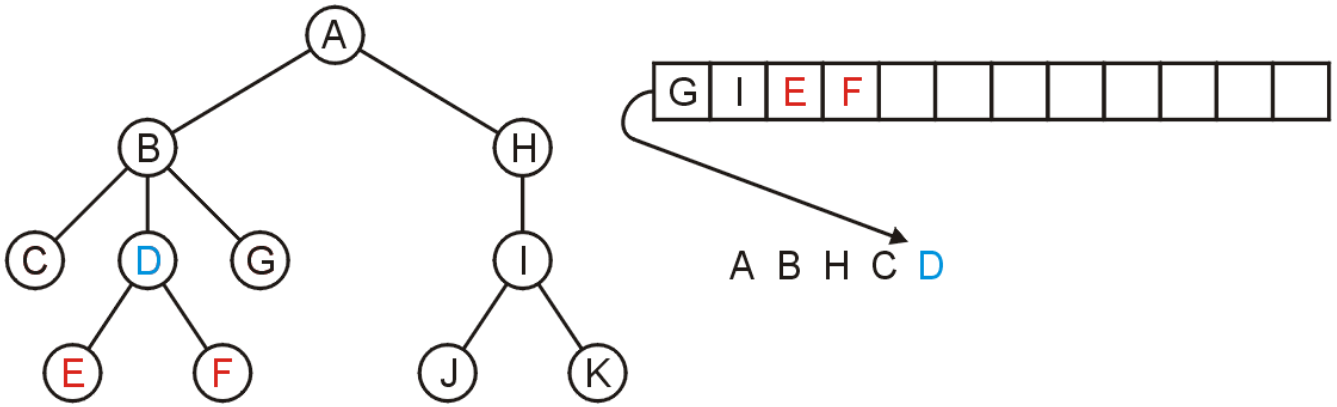
Pop C: no sub-directories



3.3.5

Application

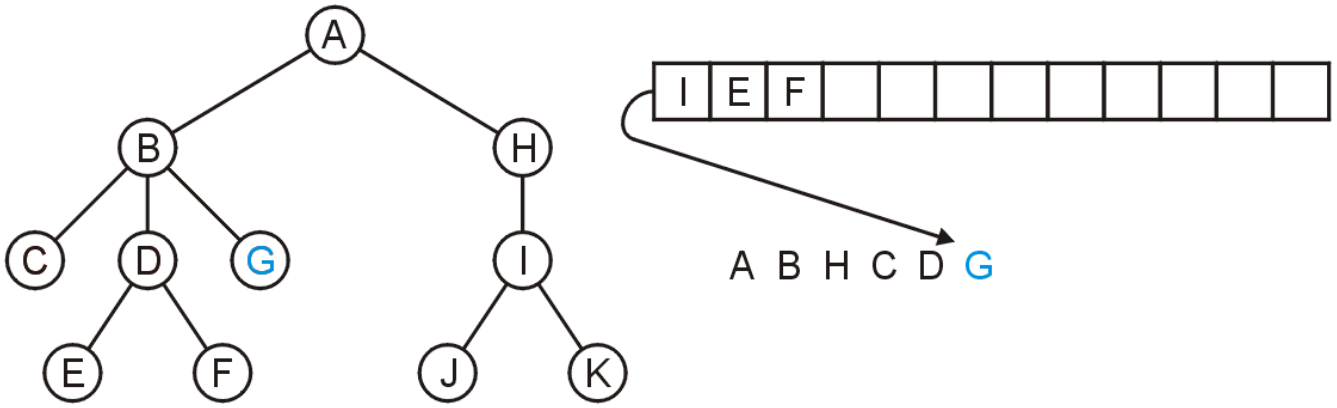
Pop D and push E and F



3.3.5

Application

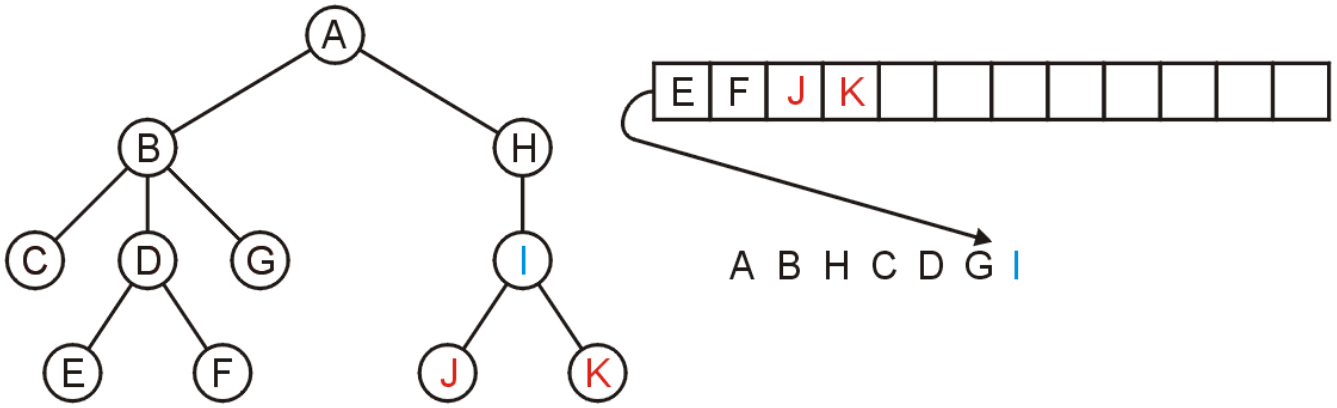
Pop G



3.3.5

Application

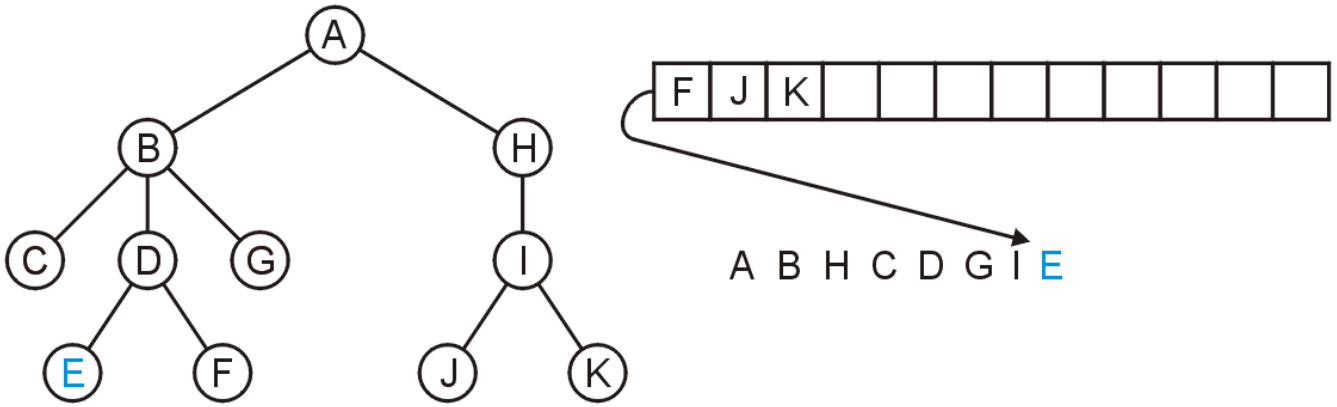
Pop I and push J and K



3.3.5

Application

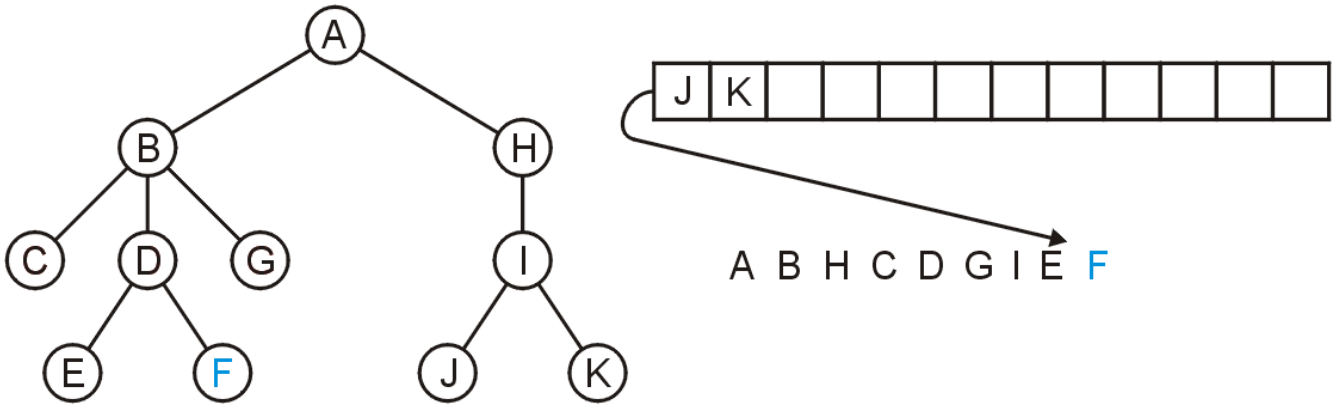
Pop E



3.3.5

Application

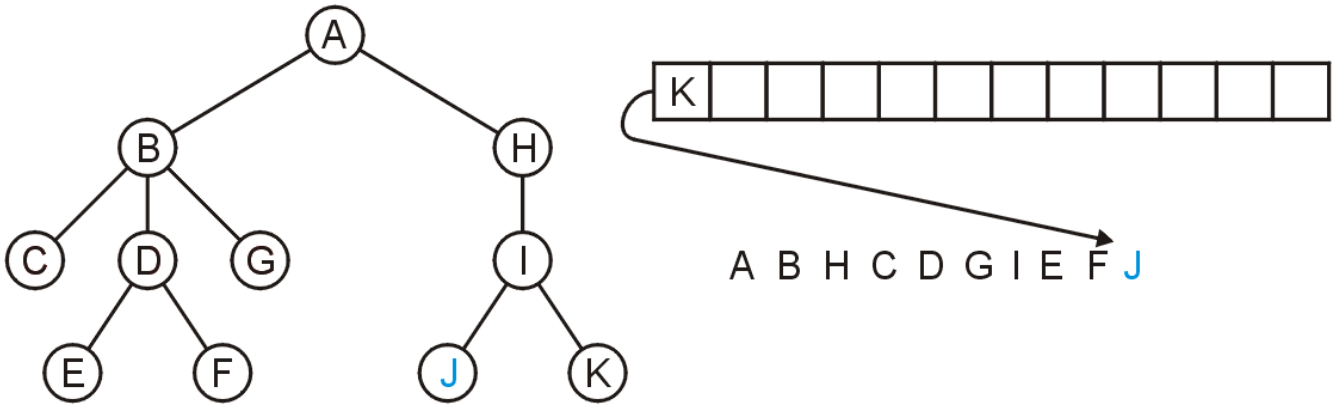
Pop F



3.3.5

Application

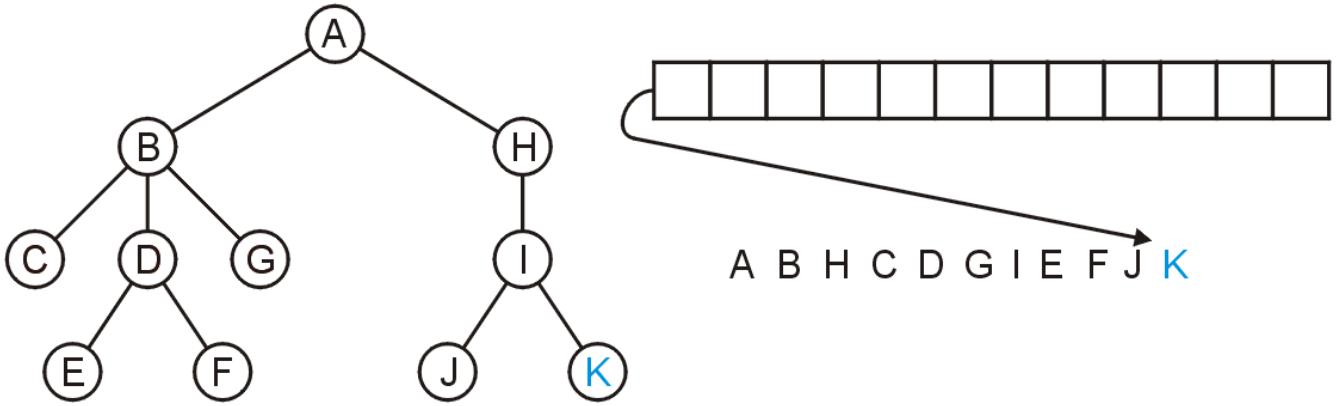
Pop J



3.3.5

Application

Pop K and the queue is empty

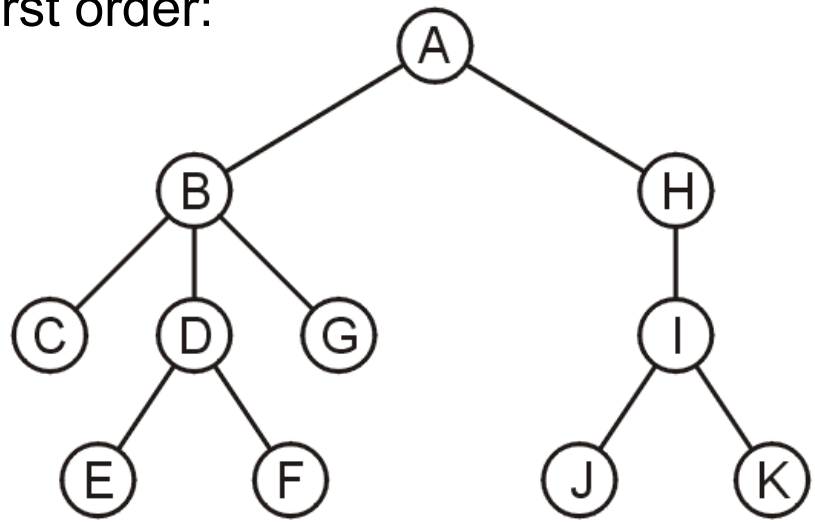


Application

The resulting order

A B H C D G I E F J K

is in breadth-first order:



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;
}
```

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