ECE 250 *Algorithms and Data Structures*

Containers, Relations, and Abstract Data Types

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

WATERLOO
ENGINEERING

Outline

This topic will describe

- The storage of objects in *containers*
- We will focus on linear orderings:
  - Implicitly defined linear orderings (sorted lists)
  - Explicitly defined linear orderings
- We will summarize this information
- We will also look briefly at:
  - Hierarchical orderings
  - Partial orderings
  - Equivalence relations
  - Adjacency relations

Outline

Any form of information processing or communication requires that data must be stored in and accessed from either main or secondary memory

There are two questions we should ask:

- What do we want to do?
- How can we do it?

Abstract Data Types:

- Models of the storage and access of information

Data Structures and Algorithms:

- The concrete methods for organizing and accessing data in the computer

2.1.1

Containers

The most general Abstract Data Type (ADT) is that of a *container*

- The Container ADT

A container describes structures that store and give access to objects

The queries and operations of interest may be defined on:

- The container as an entity, or
- The objects stored within a container

2.1.1

Operations on a Container

The operations we may wish to perform on a container are:

- Create a new container
- Copy or destroy an existing container
- Empty a container
- Query how many objects are in a container
- Query what is the maximum number of objects a container can hold
- Given two containers:
  - Find the union (merge), or
  - Find the intersection

2.1.1

Operations on a Container

Many of these operations on containers are in the Standard Template Library

Constructor	`Container()`
Copy Constructor	`Container( Container const & )`
Destructor	`~Container()`
Empty it	`void clear()`
How many objects are in it?	`int size() const`
Is it empty?	`bool empty() const`
How many objects can it hold?	`int max_size() const`
Merge with another container	`void insert( Container const & )`

2.1.1 Operations on Objects Stored in a Container

Given a container, we may wish to:

- Insert an object into a container
- Access or modify an object in a container
- Remove an object from the container
- Query if an object is in the container
  - If applicable, count how many copies of an object are in a container
- Iterate (step through) the objects in a container

2.1.1 Operations on Objects Stored in a Container

Many of these operations are also common to the Standard Template Library

Insert an object	`void insert( Type const & )`
Erase an object	`void erase( Type const & )`
Find or access an object	`iterator find( Type const & )`
Count the number of copies	`int count( Type const & )`
Iterate through the objects in a container	`iterator begin() const`

2.1.2

Simple and Associative Containers

We may split containers into two general classifications:

Simple Containers

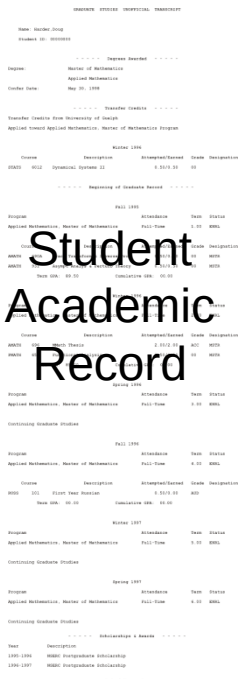
Containers that store individual objects

Associative Containers

Containers that store keys but also store records associated with those keys

Temperature readings → **Circular Array**

UW Student ID Number → **QUEST Academic Server** → Student Academic Record



2.1.2

Simple and Associative Containers

Any of the Abstract Data Types we will discuss can be implemented as either a simple container or an associative container

We will focus on simple containers in this course

- Any container we discuss can be modified to store key-record pairs

2.1.3

Unique or Duplicate Objects

Another design requirement may be to either:

- Require that all objects in the container are unique, or
- Allow duplicate objects

Many of the containers we will look at will assume uniqueness unless otherwise stated

- Dealing with duplicate objects is often just additional, and sometimes subtle, code

2.1.4

Standard Template Library

We will begin by introducing four containers from the C++ Standard Template Library (STL)

	Unique Objects/Keys	Duplicate Objects/Keys
Simple Container	`set<Type>`	`multiset<Type>`
Associative Container	`map<Key_type, Type>`	`multimap<Key_type, Type>`

2.1.4

The STL set Container

```
#include <iostream>
#include <set>

int main() {
     std::set<int> ints;

    for ( int i = -100; i <= 100; ++i ) {
        ints.insert( i*i );         // Ignores duplicates:  (-3)*(-3) == 3*3
    }                               // Inserts 101 values: 0, 1, 4, 9, ..., 10000

    std::cout << "Size of 'is': " << ints.size() << std::endl;  // Prints 101

    ints.erase( 50 );                  // Does nothing
    ints.erase( 9 );                   // Removes 9
    std::cout << "Size of 'is': " << ints.size() << std::endl;  // Prints 100

    return 0;
}
```

Operations

In any application, the actual required operations may be only a subset of the possible operations

- In the design of any solution, it is important to consider both current and future requirements
- Under certain conditions, by reducing specific operations, the speed can be increased or the memory decreased

Relationships

However, we may want to store not only objects, but relationships between the objects

- Consequently, we may have additional operations based on the relationships
- Consider a genealogical database
  - We don't only want to store the people, but we want to also make queries about the relationships between the people

Relationships

If we are not storing relationships, there is a data structure that is always the same speed no matter how many objects are stored

- A *hash table* takes the same time to find an object whether there are 10 or one billion objects in the container
- It requires approximately 30 % more memory than is occupied by the objects being stored
- Example:
  - Assume a department has 12 staff that are frequently changing
  - Rather than having a mailbox for each person, have 24 mailboxes and place mail into the *bin* corresponding to the person's last name

A	**E**	**I**	**M**	**R**	**V**
B	**F**	**J**	**M**	**S**	**W**
C	**G**	**K**	**N**	**T**	**XY**
D	**H**	**L**	**PQ**	**U**	**Z**

  - This works fine as long as there is not too much duplication

2.1.5.2

Relationships

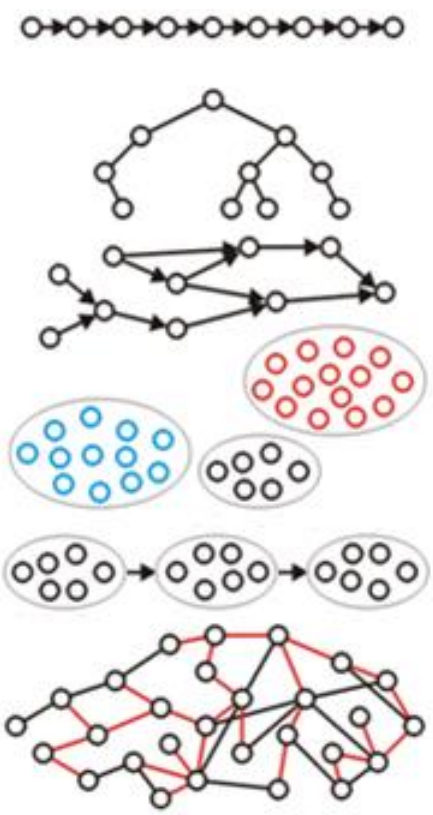
Most interesting problems, however, have questions associated with relationships:

- Which object has been stored the longest?
- Are these two classes derived from the same base class?
- Can I take ECE 427 if I failed ECE 250?
- Do both these pixels represent pavement on this image?
- Can I get from here to Roches's Point in under two hours?

Relationships

We will look at six relationships:

- Linear orderings
- Hierarchical orderings
- Partial orderings
- Equivalence relations
- Weak orderings
- Adjacency relations



2.1.5.2

Relationships

Relationships are often Boolean-valued binary operations

Example: given two integers:

- Are they equal? $x = y$
- Is one less than the other? $x < y$
- Does the first divide the second? $x \mid y$
- Do they have the same remainder modulo 16? $x \underset{16}{\equiv} y$
- Do two integers differ by at most one prime factor?

2.1.5.3

Classification of Relationships

The relationships we look at can usually be classified by one of two properties based on *symmetry*:

Symmetric	$x \sim y$ if and only if $y \sim x$	Ali is the same age as Bailey
Anti-symmetric	at most one of $x < y$ or $y < x$ can be true	Ali is shorter than Bailey ECE 150 is a prereq of ECE 250
Reflexive	$x \sim x$ for all x	Ali is the same age as Ali
Anti-reflexive	$x \not< x$ for all x	Ali is not shorter than Ali

Another common property is *transitivity*:

- If $x \sim y$ and $y \sim z$, it must be true that $x \sim z$:

  If Ali is the same age as Bailey and Bailey is the same age as Casey, it follows that Ali is the same age as Casey.

- If $x < y$ and $y < z$, it must be true that $x < z$:

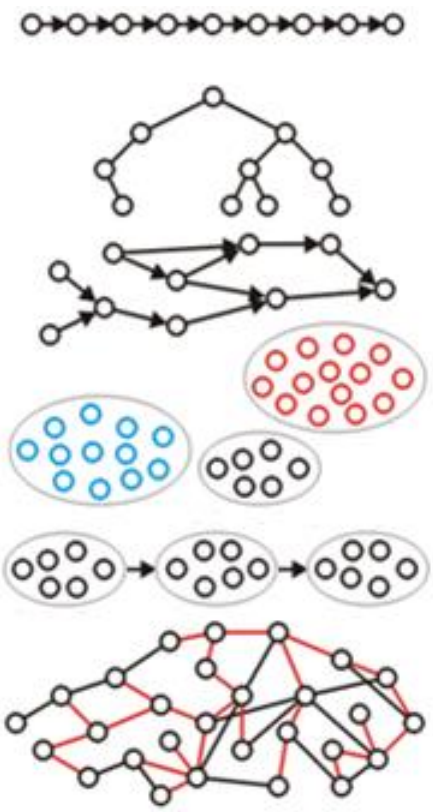
  If Ali is shorter than Bailey and Bailey is shorter than Casey, it follows that Ali is shorter than Casey.

2.1.6

Definitions and examples

We will now define and consider examples of these relationships

- Linear orderings
- Hierarchical orderings
- Partial orderings
- Equivalence relations
- Weak orderings
- Adjacency relations



Linear Orderings

A linear ordering is any relationship where any two objects x and y that can be compared, exactly one of:

$$x < y \text{ , } x = y \text{, or } y < x$$

is true and where the relation is transitive

- Such a relation is therefore anti-symmetric
- Any collection can be *sorted* according to this relation

Examples of sets which are linearly ordered include:

- Integers 1, 2, 3, 4, 5, 6, ...
- Real numbers 1.2, 1.2001, 1.24, 1.35, 2.78, ...
- The alphabet a, b, c, d, e, f, ..., x, y, z
- Memory **`0x00000000, 0x00000001, ..., 0xFFFFFFFF`**

We could store linearly ordered sets using arrays or linked lists

2.1.6.1.1

Lexicographical Orderings

Another linearly ordered set is the set of English words:

a, aardvark, aardwolf, ..., abacus, …, baa, …, bdellium, ..., zygote

The order is induced by the linear ordering on a through z

The order is determined by comparing the first letters which differ:

aardvark	< aardwolf	since	v < w
aadvark	< abacus	since	a < b
azygous	< baa	since	a < b
cat	< catch	since	b < c
wilhelm	< william	since	h < l

2.1.6.1.1

Lexicographical Orderings

Such an order can also be used to linearly order vectors:

– Say that $(x_1, y_1) < (x_2, y_2)$ if either:

$$x_1 < x_2 \text{ or both } x_1 = x_2 \text{ and } y_1 < y_2$$

For example,

$(3, 4) < (5, 1)$ **`cd < ea`**

$(3, 4) < (3, 8)$ **`cd < ch`**

2.1.6.1.2

Operations on Linear Orderings

Queries that may be asked about linear orderings:

- What are the first and last objects (the *front* and the *back*)?
- What is the k^{th} object?
- What are all objects on a given interval $[a, b]$
- Given a reference to one object in the container:
  - What are the previous and next objects?

Operations that may be performed as a result:

- Insert an object into a sorted list
- Insert an object at either the front, the back, or into the k^{th} position
- Sort a collection of objects

2.1.6.2

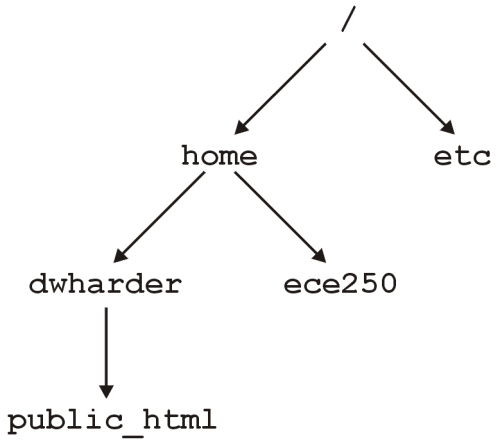
Hierarchical Orderings

The next relation we will look at is a hierarchical ordering

Consider directories in a file system:

$x \prec y$ if x contains y within one of its subdirectories

– In Unix, there is a single root directory /

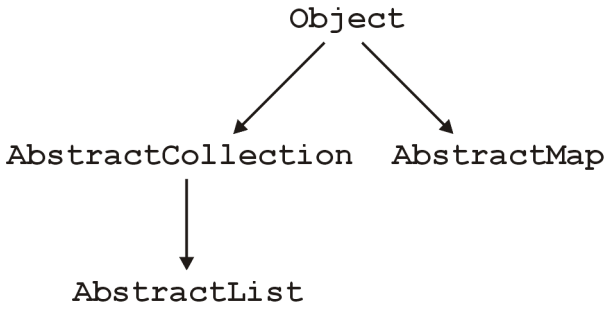


Such structures allow us to organize information

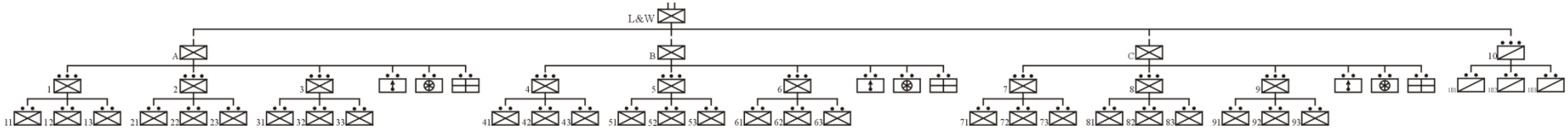
Hierarchical Orderings

Other examples:

Classes in Java and C#




```
int f() {
    int a;
    {
        int b;
        {
            int c;
        }
    }

    {
        int d;
    }
    return a;
}
```



2.1.6.2.2

Hierarchical Orderings

If $x \prec y$, we say that x precedes y or x is a predecessor of y

Even though all of these relationships may appear very different, they all exhibit the same properties:

- For all x, $x \not\prec x$ (anti-reflexive)
- If $x \prec y$ then $y \not\prec x$
- If $x \prec y$ and $y \prec z$, it follows that $x \prec z$
- There is a root r such that $r \prec x$ for all other x
- If $x \prec z$ and $y \prec z$, it follows that either $x \prec y$, $x = y$ or $x \succ y$

2.1.6.2.3 Operations on Hierarchical Orders

If the hierarchical order is explicitly defined (the usual case), given two objects in the container, we may ask:

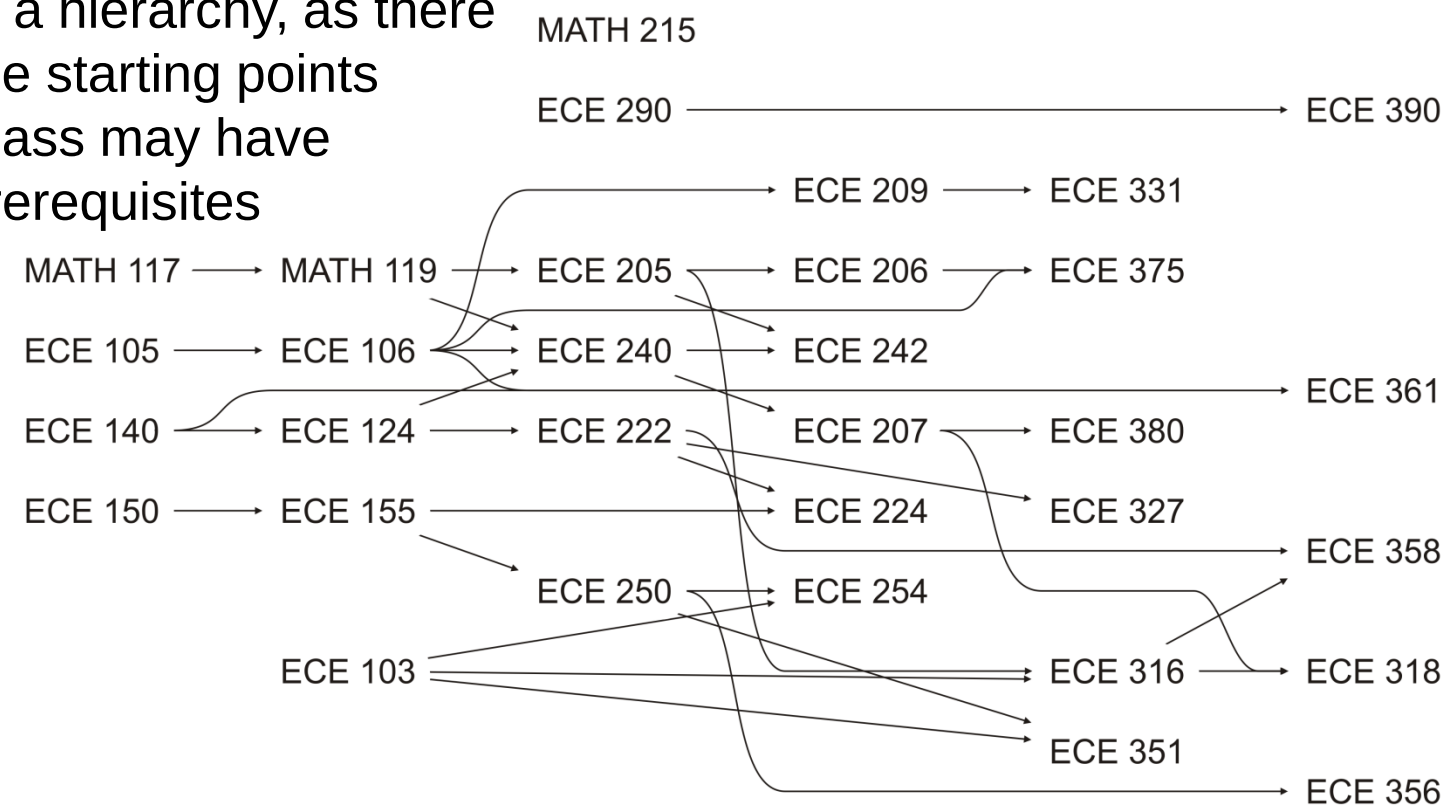
- Does one object precede the other?
- Are both objects at the same depth?
- What is the nearest common predecessor?

Partial Orderings

The next relationship we will look at is

$x \prec y$ if x is a prerequisite of y

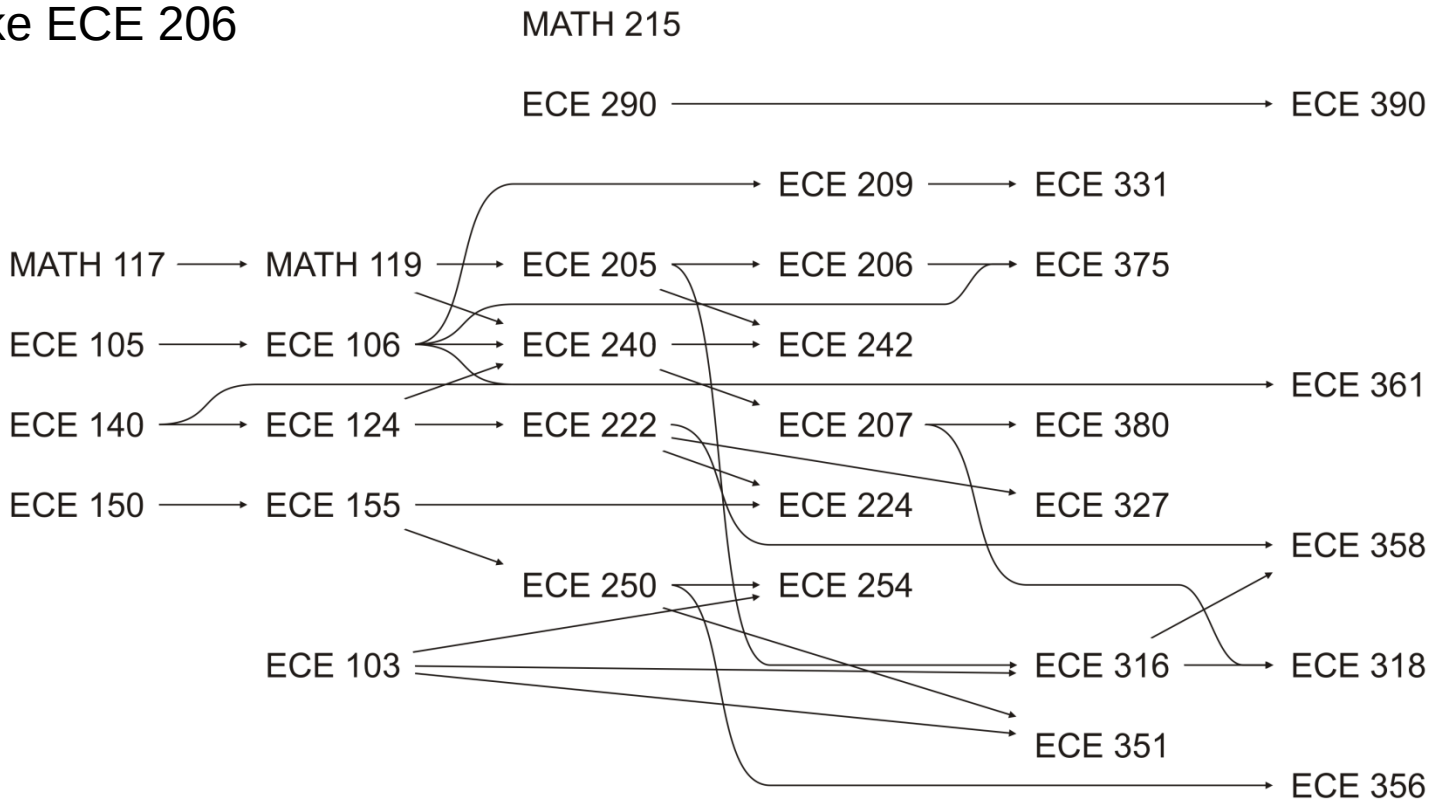
This is not a hierarchy, as there are multiple starting points and one class may have multiple prerequisites



Partial Orderings

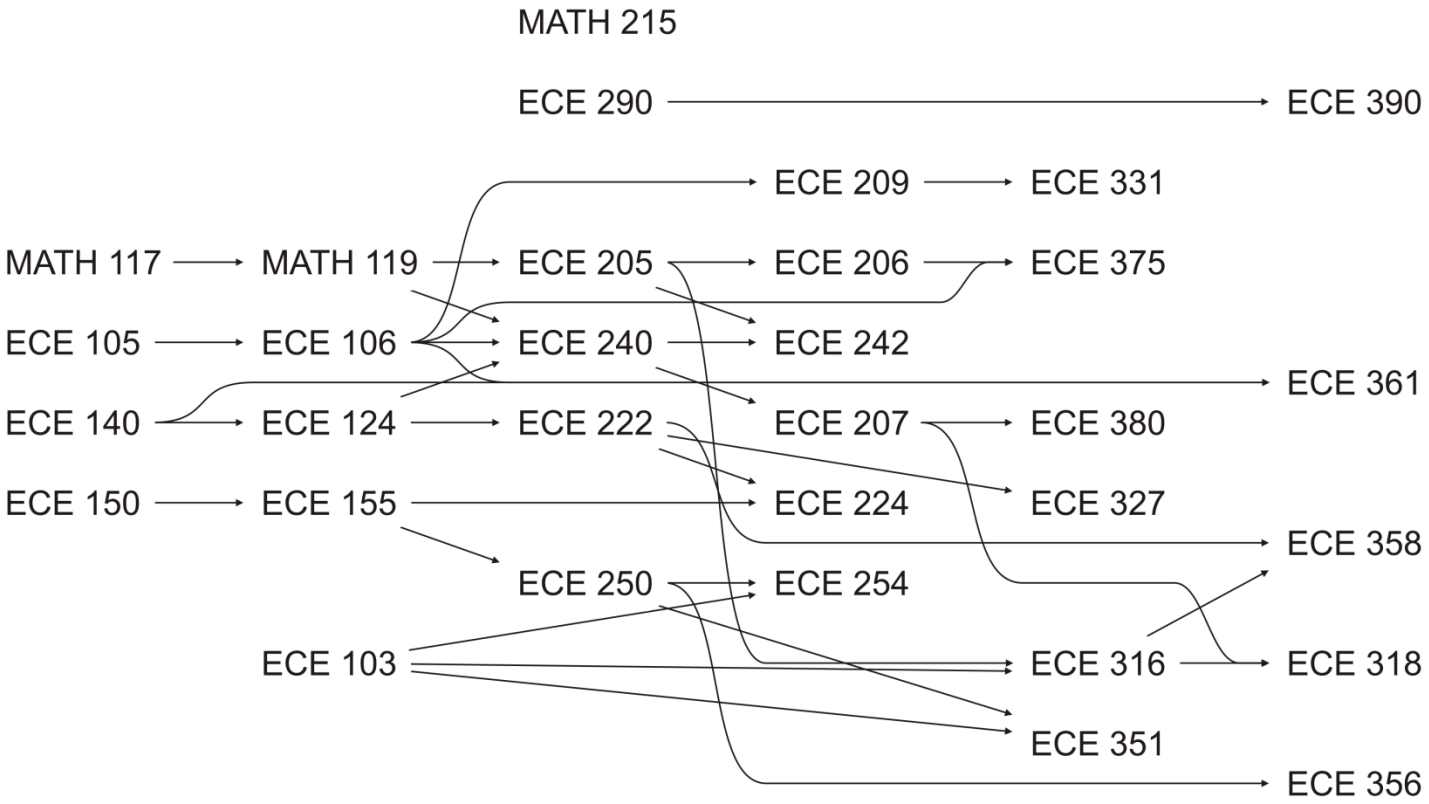
Arrows are necessary to indicate the direction:

- Having completed ECE 140, you can now take ECE 124 and ECE 361
- If you want to take ECE 375 *Electromagnetic fields and waves*, you must take ECE 206



Partial Orderings

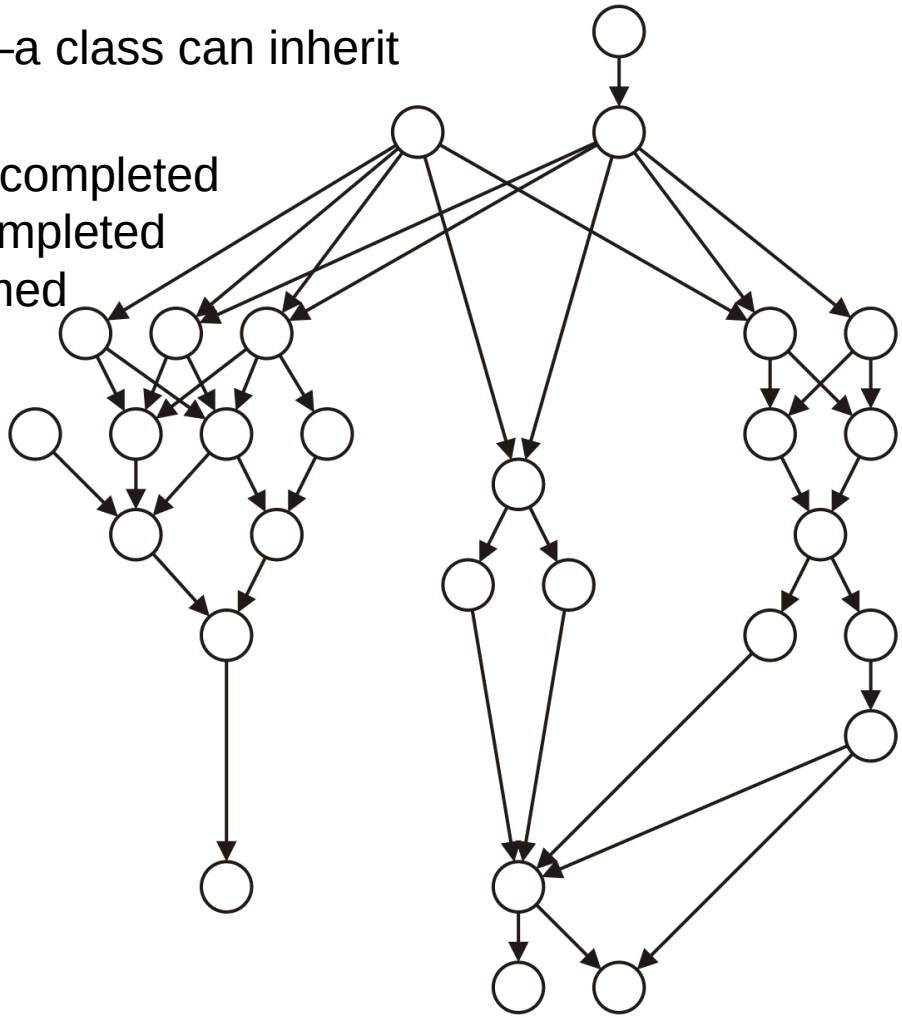
There are no loops—otherwise, you could not take any courses in the loop…



Partial Orderings

Examples:

- C++ classes (multiple inheritance–a class can inherit from more than one class), and
- A number of tasks which must be completed where particular tasks must be completed before other tasks may be performed
  - Compilation dependencies



2.1.6.3.2

Partial Orderings

All partial orderings are anti-reflexive, anti-symmetric and transitive

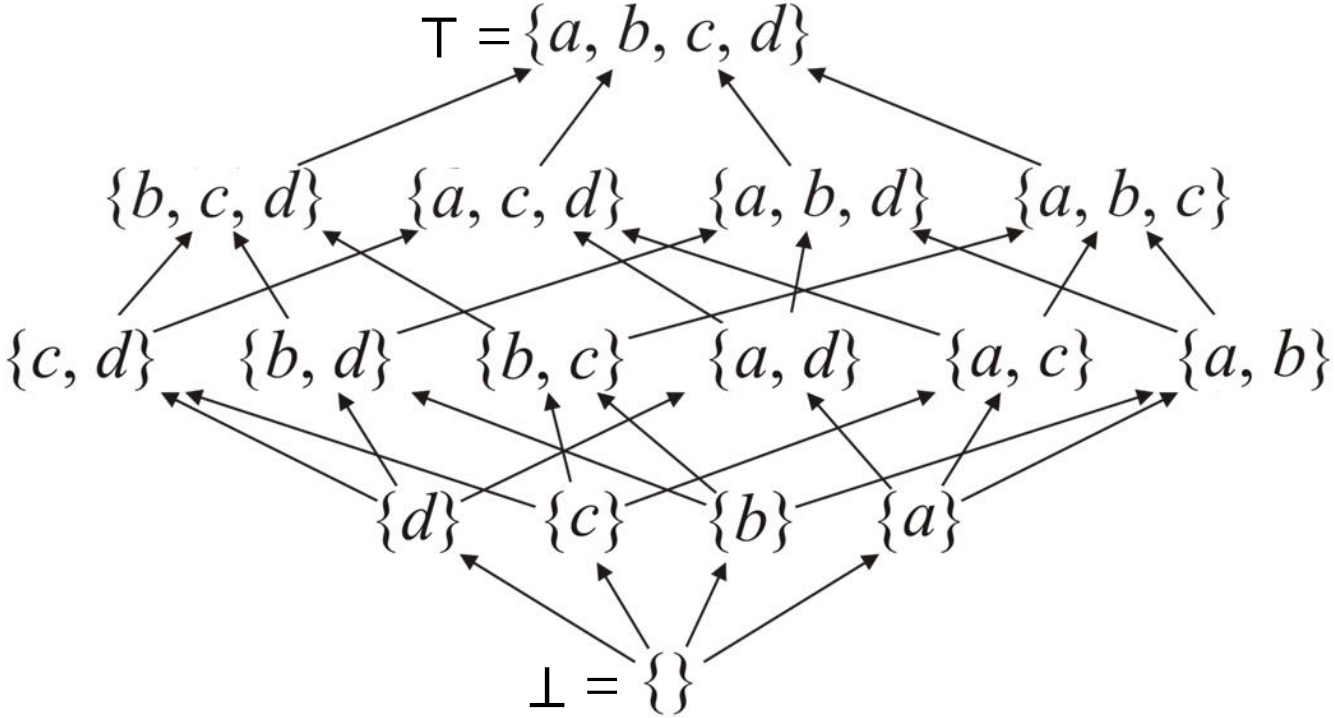
You will note that these are the first three rules of a hierarchical ordering

- What is lost?
- We are not guaranteed that there is a unique root
- There may be multiple different paths between two objects

Lattice

A finite set L that is partially ordered is called a *lattice* if there are unique elements $\top$ and $\bot$ such that $\bot \preccurlyeq x$ and $x \preccurlyeq \top$ for all elements $x \in L$

- Here, $A \prec B$ if $A \subset B$
- Graphically, $A \prec B$ if there is a path from A to B



2.1.6.3.4

Operations on Partial Orderings

Partial orders are similar to hierarchical orders; consequently, some operations are similar:

- Given two objects, does one precede the other?
- Which objects have no predecessors?
  - Not unique (unlike a hierarchy)
- Which objects immediate precede an object?
  - A hierarchical order has only one immediate predecessor
- Which objects immediately succeed an object?

2.1.6.4.1

Equivalence Relations

Consider the relationship

$x \sim y$ if x and y are of the same gender

Here we have another set of properties:

- This relationship is symmetric and transitive, but it is also *reflexive*:

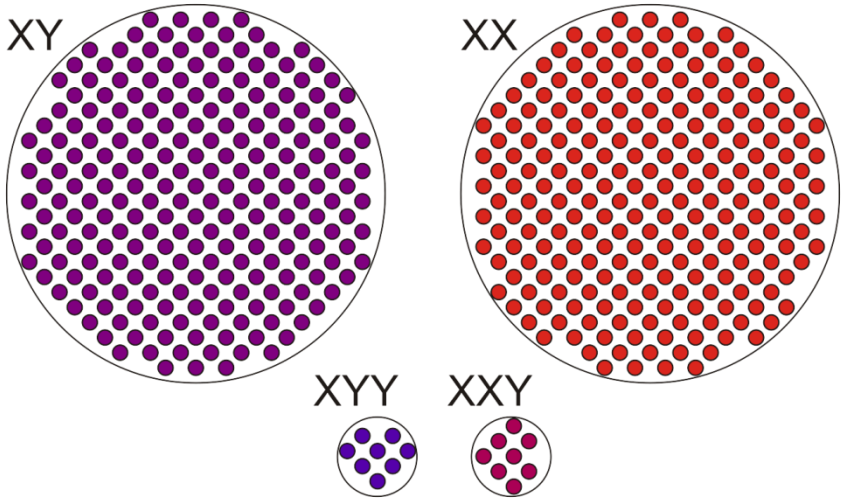
$x \sim x$ for all x

2.1.6.4.1

Equivalence Relations

One nice property of equivalence relations is that you can create *equivalence classes* of objects where each object in the class are related

- If you consider genetics, there are four general *equivalence classes*



2.1.6.4.2

Equivalence Relations

Mathematically, we could say that two functions $f(x)$ and $g(x)$ are *equivalent* if

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = c$$

for some value of c that is $0 < c < \infty$

Two circuits, one from Motorola and the other from IBM, may be said to be *equivalent* if they perform the same operations

2.1.6.4.3 Operations on Equivalence Relations

Given an equivalence relation:

- Are two objects related?
- Iterate through all objects related to one particular object
- Count the number of objects related to one particular object
- Given two objects x and y which are not currently related, make them related (union)
  - Not so easy: everything related to x must now be related to everything related to y

2.1.6.5

Weak Orderings

Finally, we will look at the relationship

$x \sim y$ if x and y are the same age

and

$x < y$ if x is younger than y

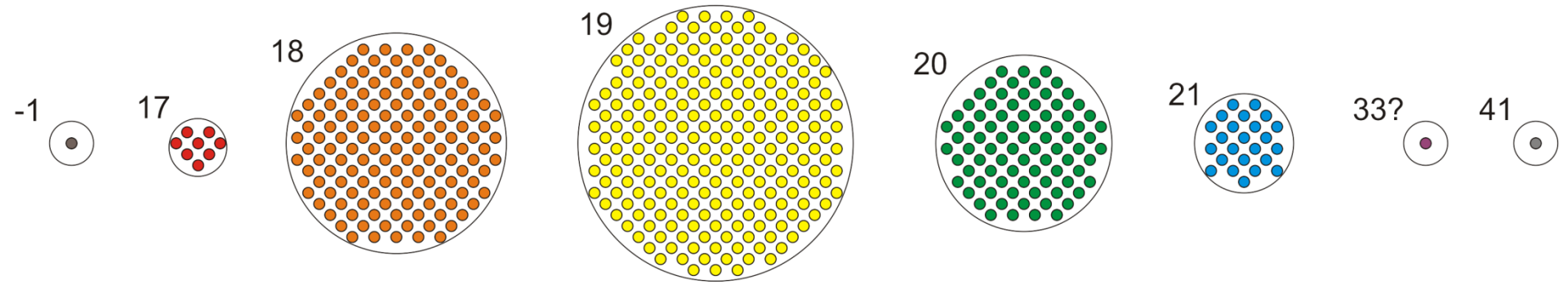
A weak ordering is a linear ordering of equivalence classes

If x is the same age or younger than y, we would say $x \lesssim y$

2.1.6.5

Weak Orderings

One nice property of equivalence relations is that you can create groups of objects where each object in the group has the same properties



Weak Orderings

The four containers,

```
set<T>  multiset<T>  map<K, S>  multimap<K, S>
```

expect that the objects/keys may be compared using the relational operators and that relational operator must satisfy a weak ordering

The set/map will store only one object/key per equivalence class

The multiset/multimap will store multiple objects in each equivalence class

- The containers do not guarantee the same internal ordering for objects that are equivalent

2.1.6.5.2

Operations on Weak Orderings

The operations on weak orderings are the same as the operations on linear orderings and equivalence classes, however:

- There may be multiple smallest or largest objects
- The *next* or *previous* object may be equivalent

2.1.6.6

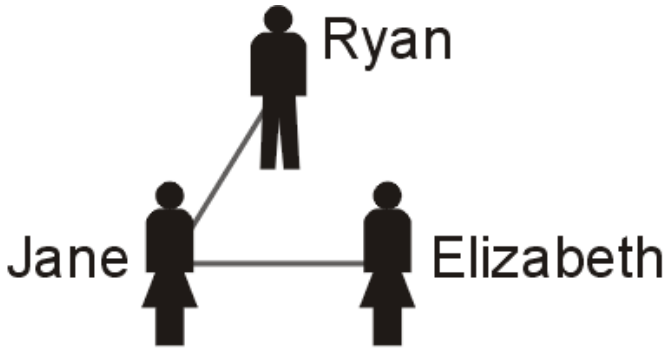
Adjacency Relations

The last relationship we will look at is

$x \leftrightarrow y$ if x and y are friends

Like a tree, we will display such a relationship by displaying a line connecting two individuals if they are friends (a *graph*)

E.g., Jane and Ryan are friends, Elizabeth and Jane are friends, but Elizabeth thinks Ryan is a little odd...

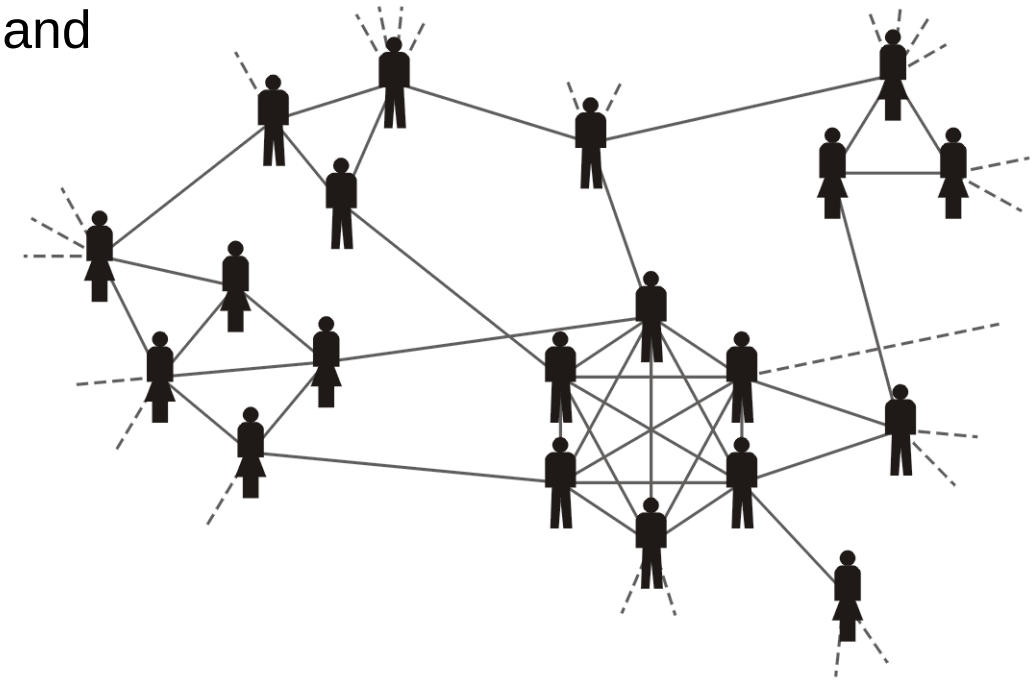


Adjacency Relations

Such a relationship is termed an *adjacency relationship*

- Two individuals who are related are also said to be *adjacent* to each other

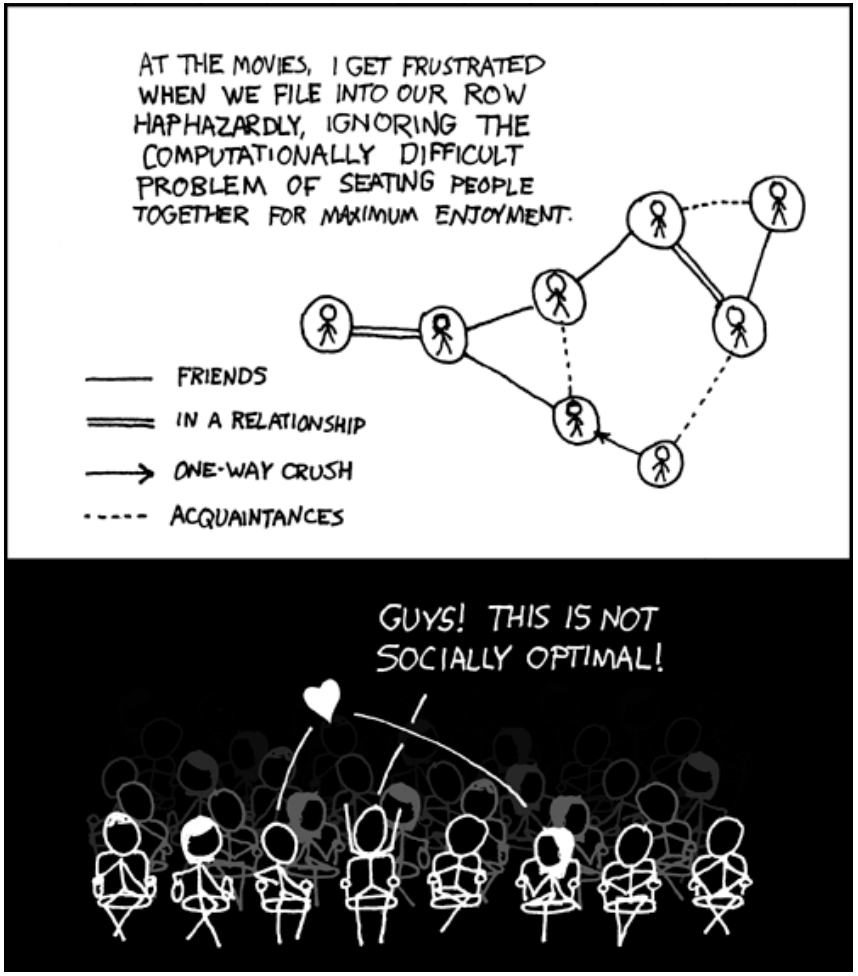
Here we see a hockey team and some of their friends



2.1.6.6

Adjacency Relations

Alternatively, the graph may be more complex



http://xkcd.com/173/

2.1.6.6

Adjacency Relations

In some cases, you do not have global relationships, but rather, you are simply aware of neighbouring, or adjacent, nodes

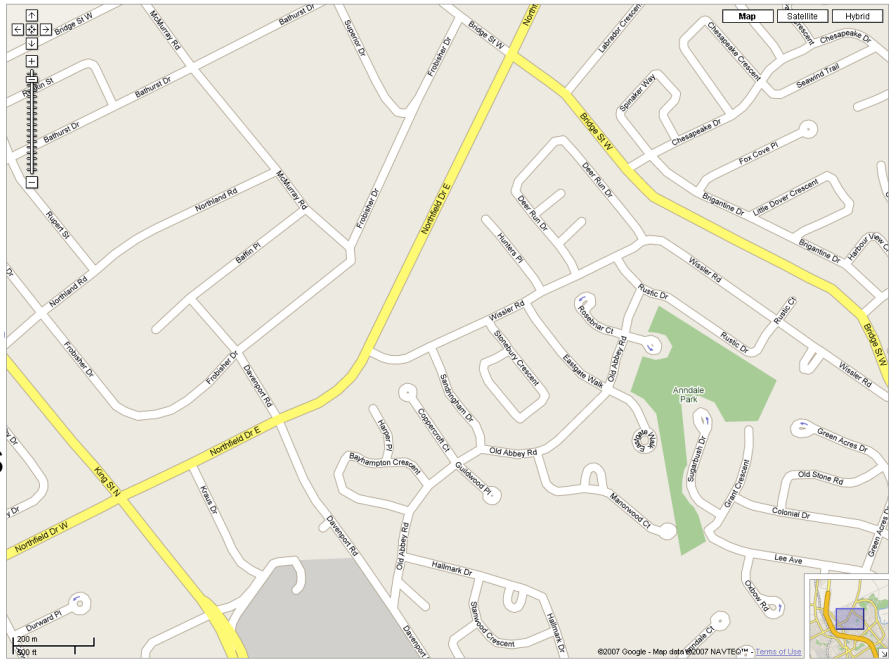
Such a relationship defines a graph, where:

- Nodes are termed vertices
- Edges denote adjacencies

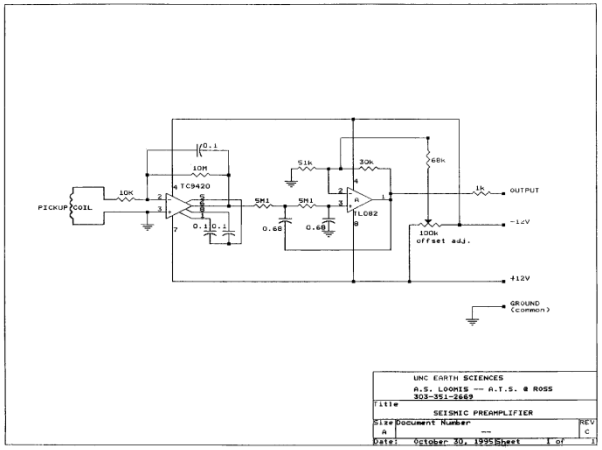
Adjacency Relations

Two examples:

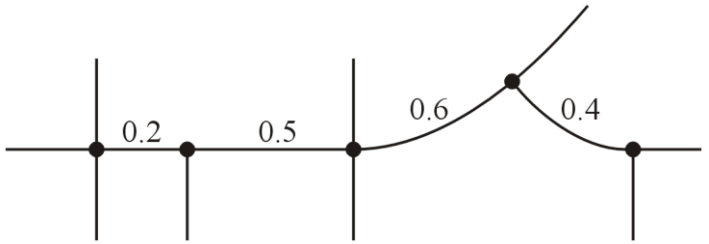
- City streets
  - intersections are vertices
  - streets are edges
- Circuits
  - circuit elements are vertices
  - connections are edges



http://maps.google.ca/



http://esci.unco.edu/resource/circuit.htm



2.1.6.6.1

Operations on Adjacency Relations

Given an adjacency relation:

- Are two objects adjacent?
- Iterate through all objects adjacent to one object
- Given two objects a and b, is there a sequence of objects

$$a = x_0, x_1, x_2, x_3, \ldots, x_n = b$$

such that x_k is adjacent to x_{k+1}?

I.e., are the objects *connected*?

Summary of Relations

We have now seen six relationships:

- Linear orderings
- Hierarchical orderings
- Partial orderings
- Equivalence relations
- Weak orderings
- Adjacency relations

All of these are relationships that exist on the objects we may wish to store, access, and query

2.1.7

Defining Relations

Any relationship may be either implicitly defined or explicitly imposed

- Integers are implicitly ordered based on their relative values
- Age groups are defined by the properties of the individuals

Defining Relations

Any relationship may be either implicitly defined or explicitly imposed

- A hierarchy in a company is explicitly defined
- The order of the letters on this slide are explicitly imposed by the author
- Pixels are defined as pavement based on explicitly imposed rules based on colour and surrounding pixels

2.1.7.1

Defining Relations

Relationships may be defined globally or locally

- Any two integers may be compared without reference to other integers
- Any two sets can be compared to determine if one is a subset of the other

2.1.7.2

Defining Relations

Relationships may be defined globally or locally

- Prerequisites are defined locally:

  ECE 150 is a prerequisite of ECE 155

  ECE 155 is a prerequisite of ECE 250

  From these, we may deduce that ECE 150 is a prerequisite of ECE 250
- Relationships in a company are defined locally:
  - Person X reports directly to person Y
  - Person Z is the president (or root) of the hierarchy
- Street grids and circuits are defined locally:
  - These two intersections are connected by a road
  - These two circuit elements are connected by a wire

2.1.7.3

Defining Relations

In general,

- Explicitly imposed relationships are usually defined locally
- Implicitly defined relationships can usually be determined globally

Abstract Data Types

In engineering, we tend to see certain patterns that occur over and over in applications

In these circumstances, we first name these patterns and then proceed to define certain standard solutions or implementations

In software in storing objects and relationships in containers, there are reoccurring containers of objects and associated relationships where the actual queries and operations are restricted

- We model such containers by *Abstract Data Types* or ADTs

2.1.8

Abstract Data Types

Any time you are intending to store objects, you must ask:

- What are the relationships on the objects?
- What queries will be made about the objects in the container?
- What operations will be performed on the objects in the container?
- What operations may be performed on the container as a whole?
- What queries will be made about the relationships between the objects in the container?
- What operations may be made on the relationships themselves between the objects in the container?

2.1.8

Abstract Data Types

Throughout this lecture, we will describe various ADTs and then look at various data structures that will allow us to efficiently implement the required queries and operations defined by the ADT

2.1.8

Abstract Data Types

Another ADT is the *Sorted List ADT*

- A container that stores linearly ordered objects where we may want insert, access, or erase objects at arbitrary locations

You may immediately think that we could using either an array or a linked list to implement the Sorted List ADT; however, we will see that that is usually **very** inefficient

- They are so inefficient that if, by the end of the class, if the first thing you think of is using an array or linked list to store sorted objects, I've done something wrong…

What's next?

We have discussed containers, relationships, and ADTs

- What is it we want to store and access
- What queries and operations are we interested in

The next question is, how do we implement these efficiently on a computer?

The next step is to look at *data structures*

- These are particular methods of storing and relating data on the computer
- One data structure may be appropriate as an implementation for numerous ADTs
- It may not be possible to find a data structure that allows optimal implementations for all queries and operations of a particular ADT

Summary

In this topic, we have covered:

- The Container ADT as a basic model of organizing data
  - Queries and operations on containers
  - Simple and associative containers
  - Unique or duplicate objects
- Relationships between data
  - Linear ordering
    - Lexicographical ordering
  - Hierarchical ordering
  - Partial ordering
  - Equivalence relation
  - Weak ordering
  - Adjacency relation
- In each case, we considered relationship-specific queries and operations
- Abstract Data Types as a model for organizing information

References

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

ECE 250 *Algorithms and Data Structures*

Lists

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

WATERLOO
ENGINEERING

3.1

Definition

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

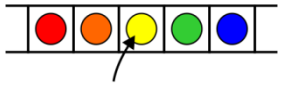
We will look at the most common operations that are usually

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

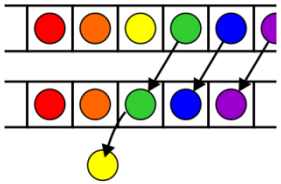
Operations

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

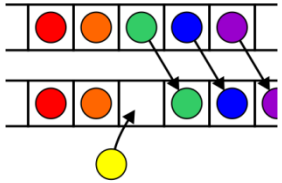
Access to the object



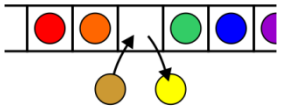
Erasing an object



Insertion of a new object



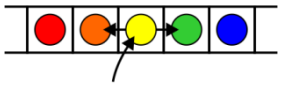
Replacement of the object



3.1.1

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

3.1.2

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 $\Theta(1)$, $O(n)$ or $\Theta(n)$

3.1.3

Linked lists

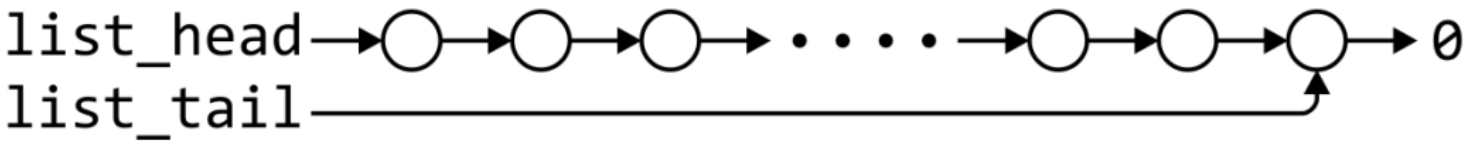
We will consider these for

- 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

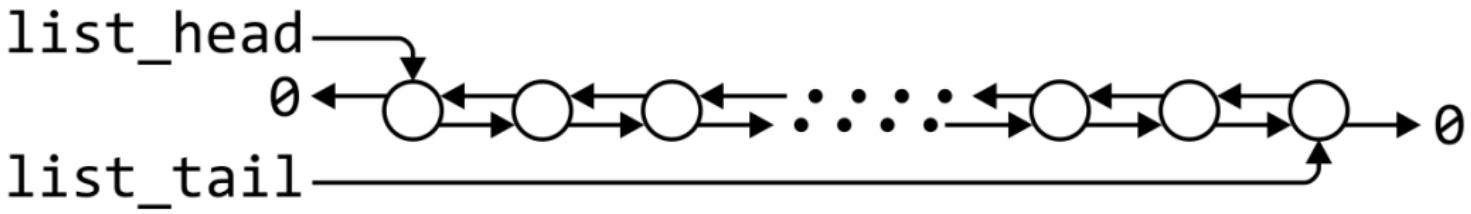


3.1.3.2

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

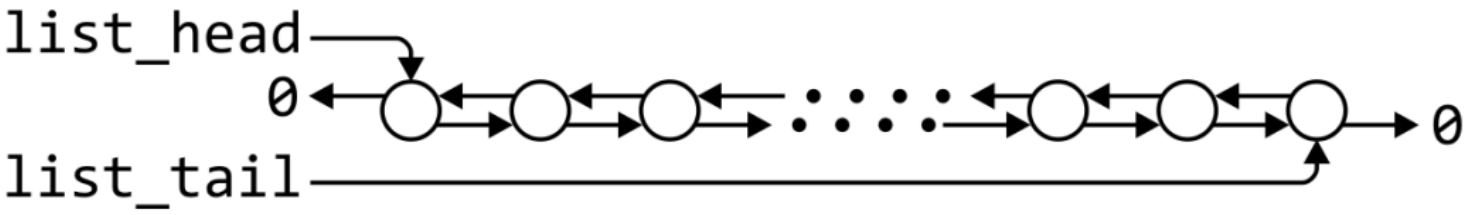


3.1.3.2

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	$\Theta(1)$



3.1.3.3

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

3.1.4

Arrays

We will consider these operations for arrays, including:

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

3.1.4

Standard arrays

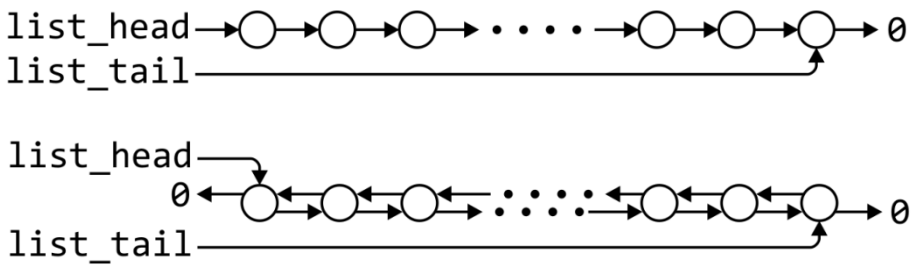
We will consider these operations for arrays, including:

- 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

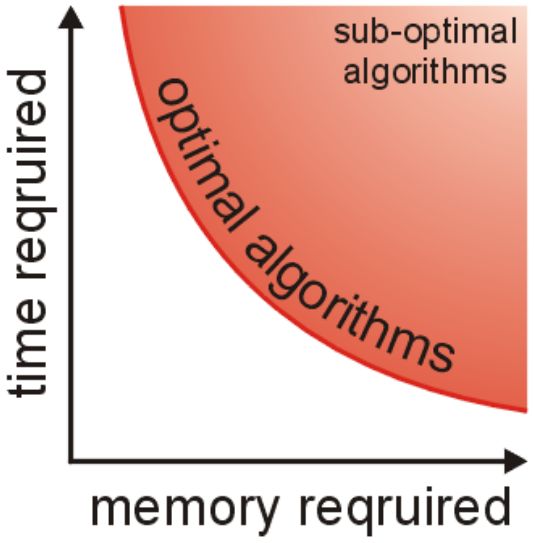
Memory usage versus run times

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

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
$
```

ECE 250 *Algorithms and Data Structures*

Stacks

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

WATERLOO
ENGINEERING

3.2.1

Abstract Stack

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

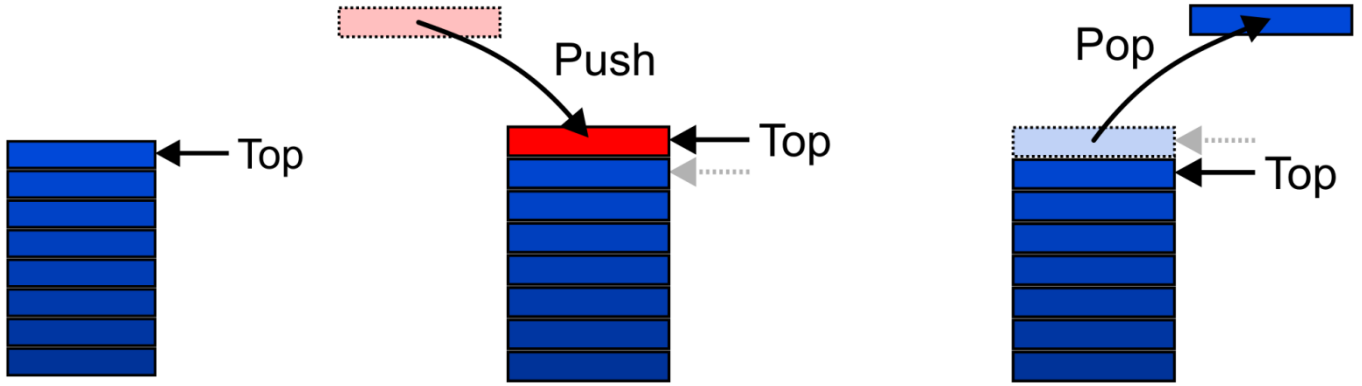
- Uses a explicit linear ordering
- Insertions and removals are performed individually
- Inserted objects are *pushed onto* the stack
- The *top* of the stack is the most recently object pushed onto the stack
- When an object is *popped* from the stack, the current *top* is erased

3.2.1

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

3.2.2

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

3.2.2

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

3.2.3

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

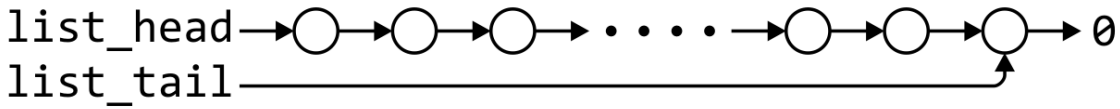
We will look at

- Singly linked lists
- One-ended arrays

3.2.3.1

Linked-List Implementation

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



	Front/1st	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

3.2.3.1

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 & );
};
```

3.2.3.1

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();
};
```

3.2.3.1

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();
};
```

3.2.3.1

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 );
}
```

3.2.3.1

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();
}
```

3.2.3.2

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

3.2.3.2

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;
```

3.2.3.2

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();
};
```

3.2.3.2

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
}
```

3.2.3.2

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();
};
```

3.2.3.2

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;
}
```

3.2.3.2

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;
}
```

3.2.3.2

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];
}
```

3.2.3.2

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];
}
```

3.2.3.2

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;
}
```

3.2.3.2

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;`

3.2.4

Array Capacity

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

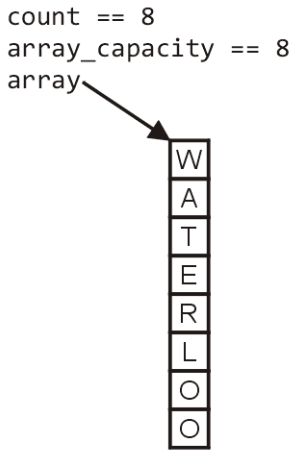
If we increase the array capacity, the question is:

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

3.2.4

Array Capacity

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

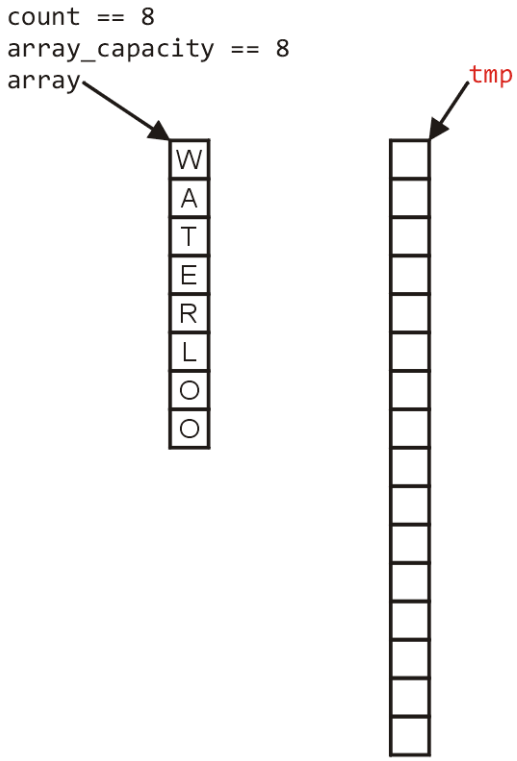


3.2.4

Array Capacity

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

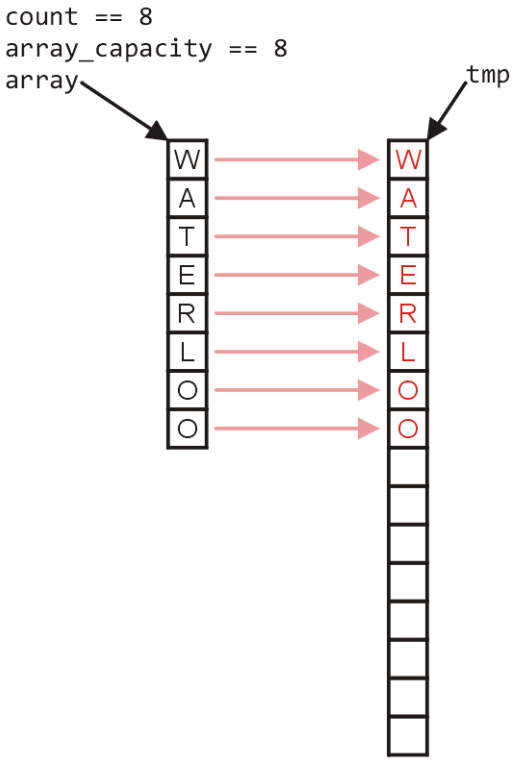
- We must have access to this so we must store the address returned by new in a local variable, say `tmp`



3.2.4

Array Capacity

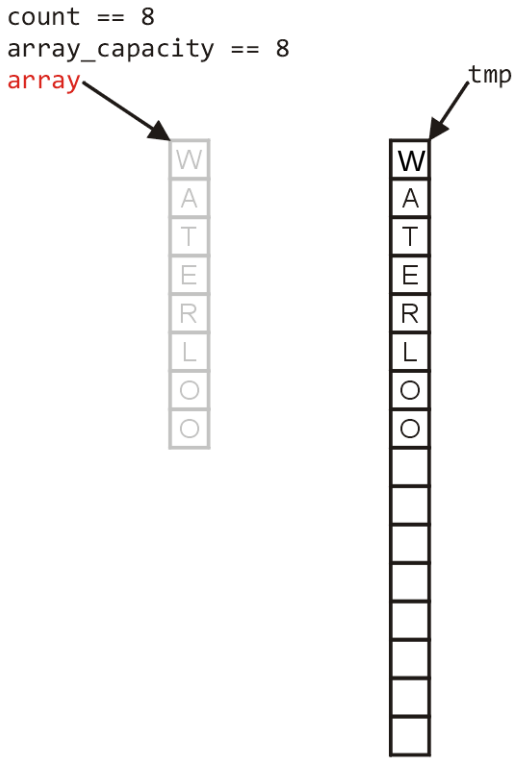
Next, the values must be copied over



3.2.4

Array Capacity

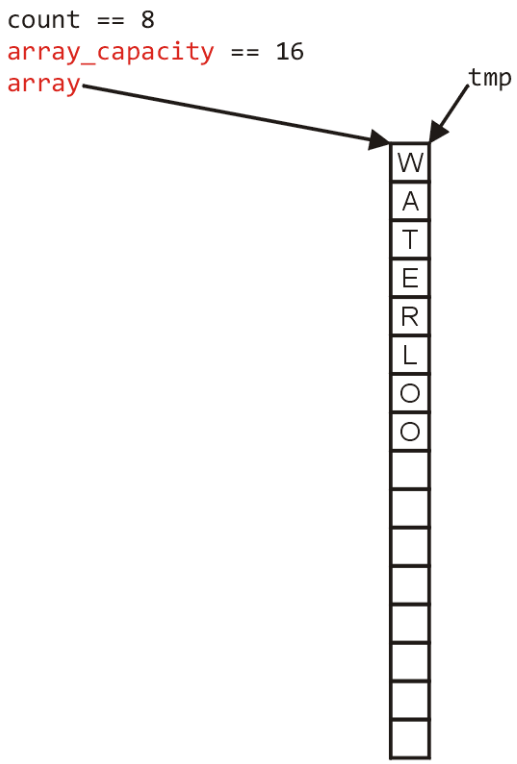
The memory for the original array must be deallocated



3.2.4

Array Capacity

Finally, the appropriate member variables must be reassigned

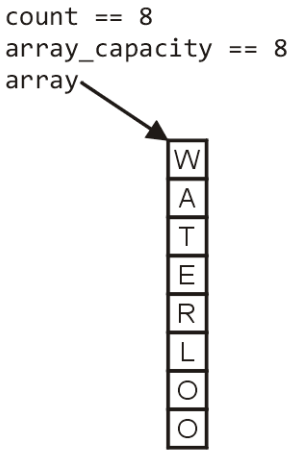


3.2.4

Array Capacity

The implementation:

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

}
```

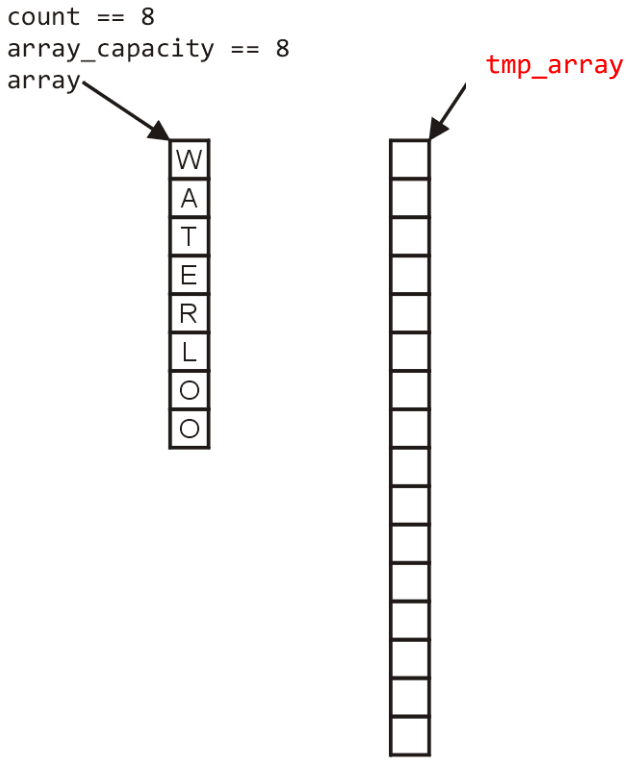


3.2.4

Array Capacity

The implementation:

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




}
```

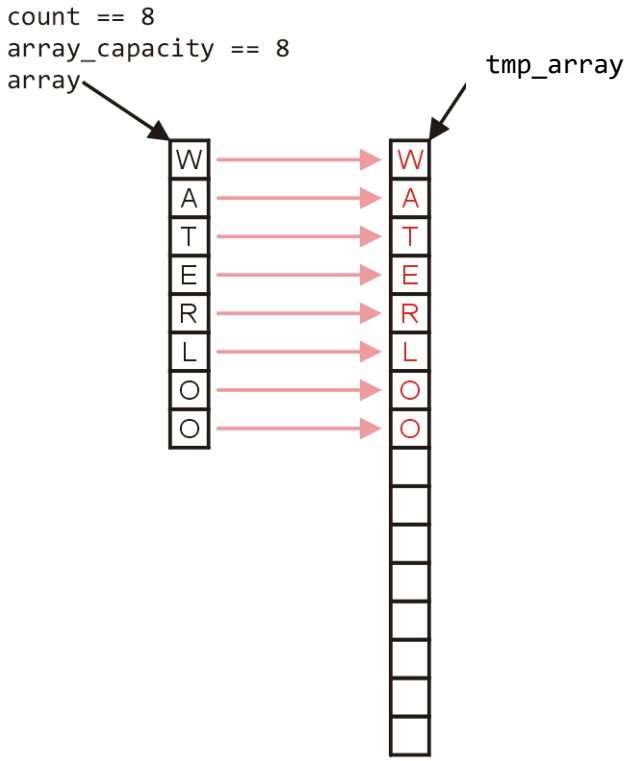


3.2.4

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];
    }



}
```

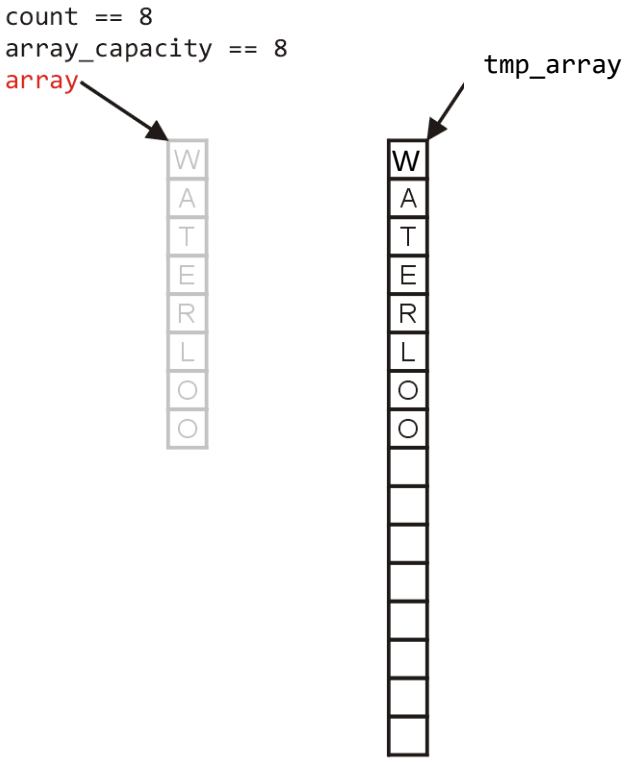


3.2.4

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;



}
```

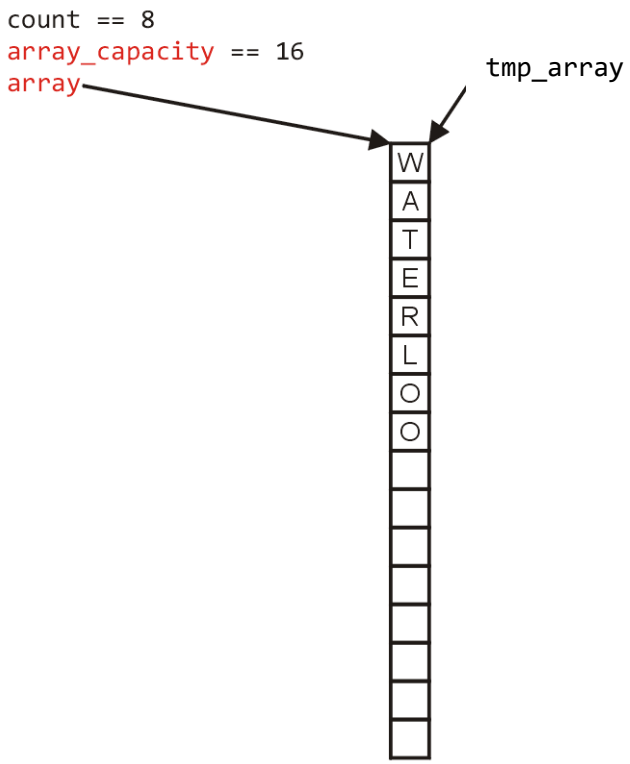


3.2.4

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;
}
```



3.2.4

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

3.2.5.5

Standard Template Library

The Standard Template Library (STL) has a *wrapper* class stack with the following declaration:

```
template <typename T>
class stack {
    public:
        stack();                        // not quite true...
        bool empty() const;
        int size() const;
        const T & top() const;
        void push( const T & );
        void pop();
};
```

3.2.6

Standard Template Library

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
    stack<int> istack;

    istack.push( 13 );
    istack.push( 42 );
    cout << "Top: " << istack.top() << endl;
    istack.pop();                                  // no return value
    cout << "Top: " << istack.top() << endl;
    cout << "Size: " << istack.size() << endl;

    return 0;
}
```

3.2.6

Standard Template Library

The reason that the `stack` class is termed a wrapper is because it uses a different container class to actually store the elements

The `stack` class simply presents the *stack interface* with appropriately named member functions:

- `push`, `pop` , and `top`

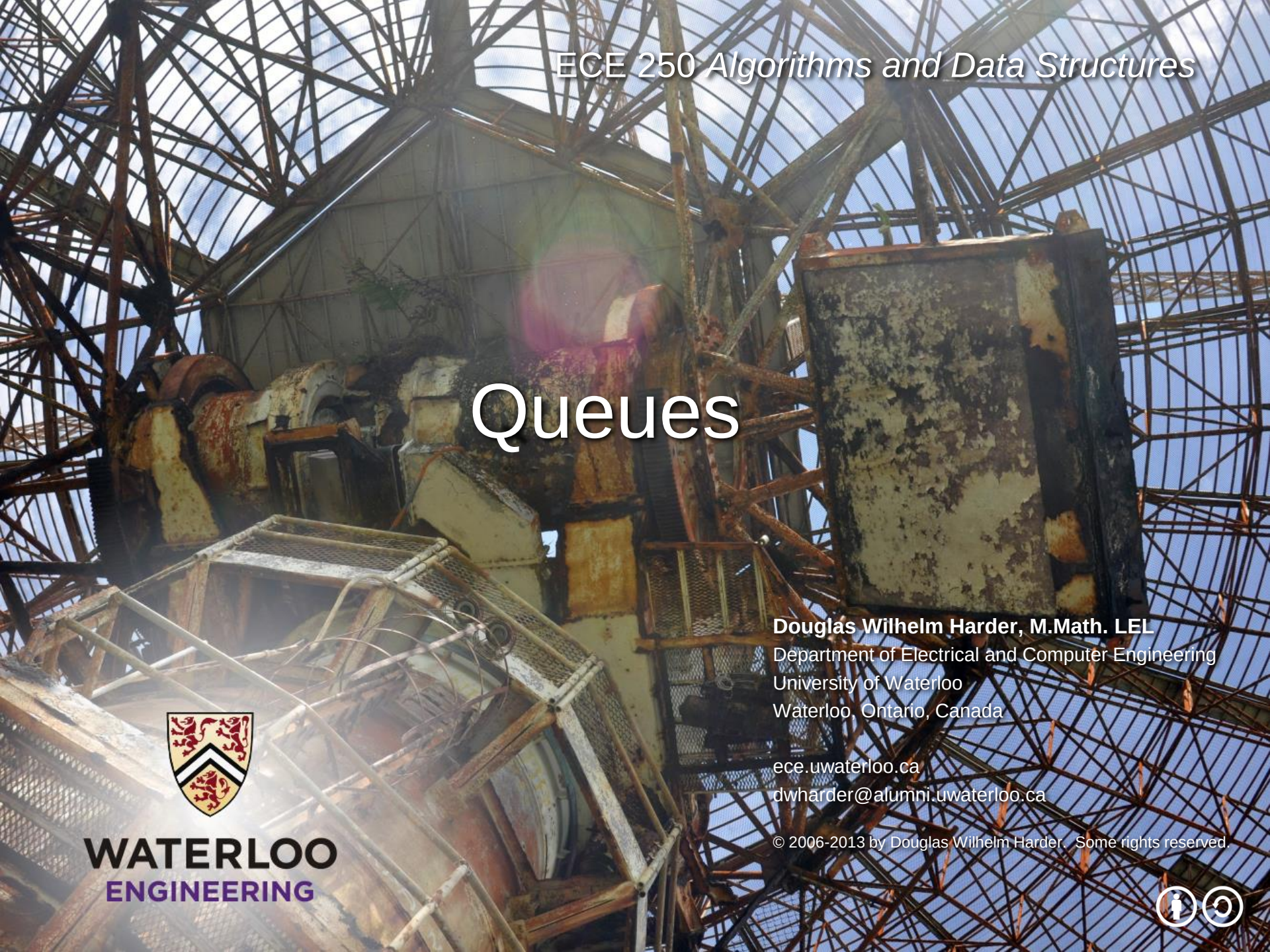
Stacks

The stack is the simplest of all ADTs

- Understanding how a stack works is trivial

The application of a stack, however, is not in the implementation, but rather:

- Where possible, create a design which allows the use of a stack



ECE 250 *Algorithms and Data Structures*

Queues

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

WATERLOO
ENGINEERING

3.3

Abstract Queue

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

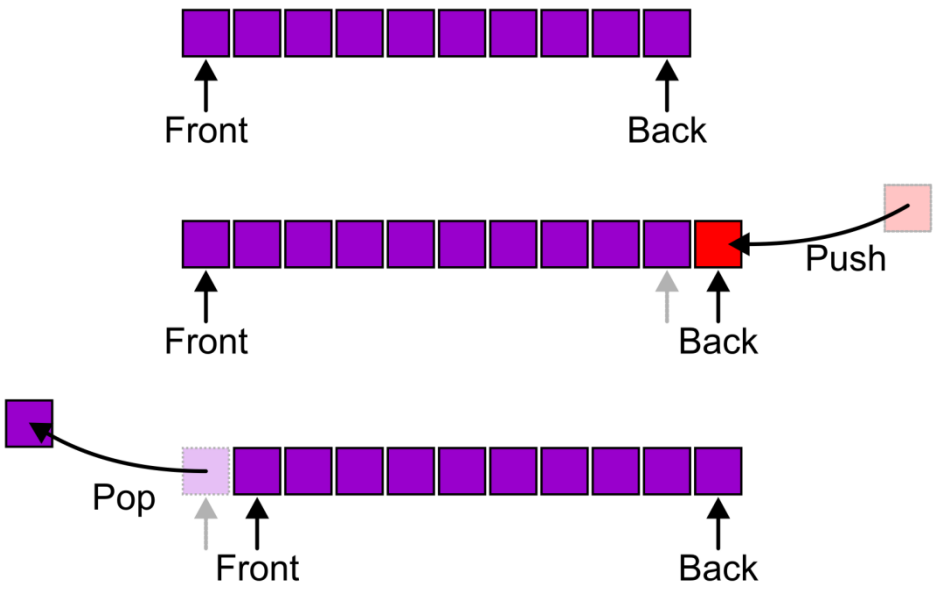
- Uses a explicit linear ordering
- Insertions and removals are performed individually
- There are no restrictions on objects inserted into (*pushed onto*) the queue—that object is designated the back of the queue
- The object designated as the *front* of the queue is the object which was in the queue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

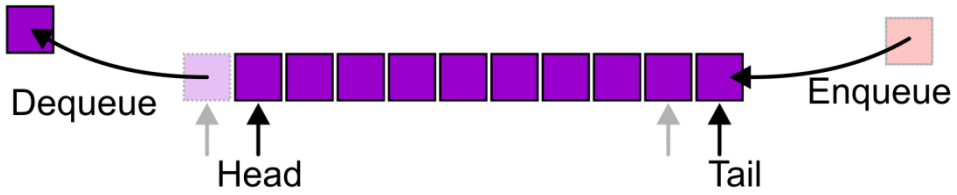
– Graphically, we may view these operations as follows:



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*.

3.3.3

Implementations

We will look at two implementations of queues:

- Singly linked lists
- Circular arrays

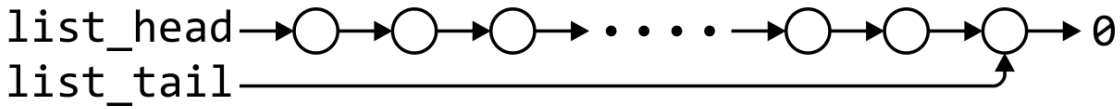
Requirements:

- 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/1st	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 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;
}
```

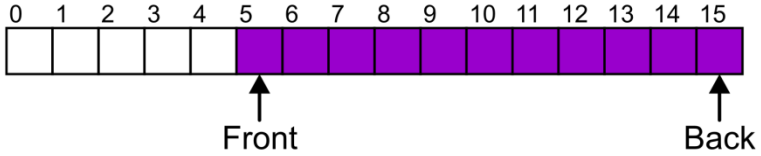
```
template <typename Type>
Type Queue<Type>::pop() {
    if ( empty() ) {
        throw underflow();
    }

    --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

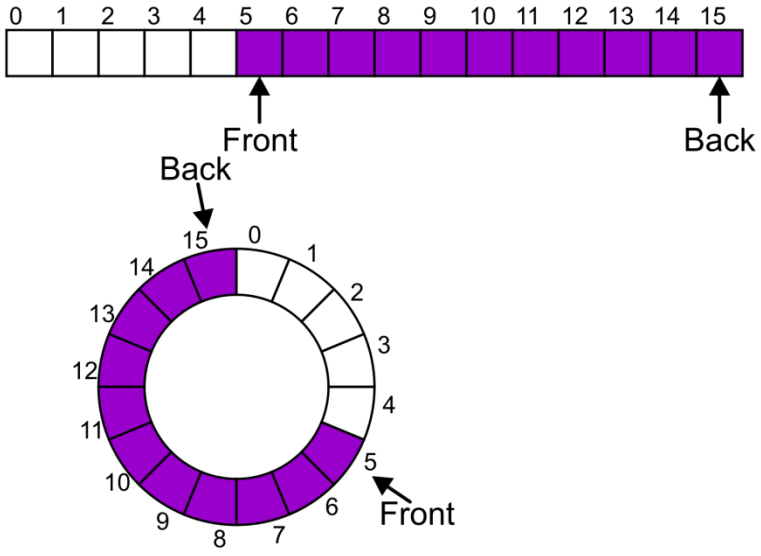
3.3.3.2

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*

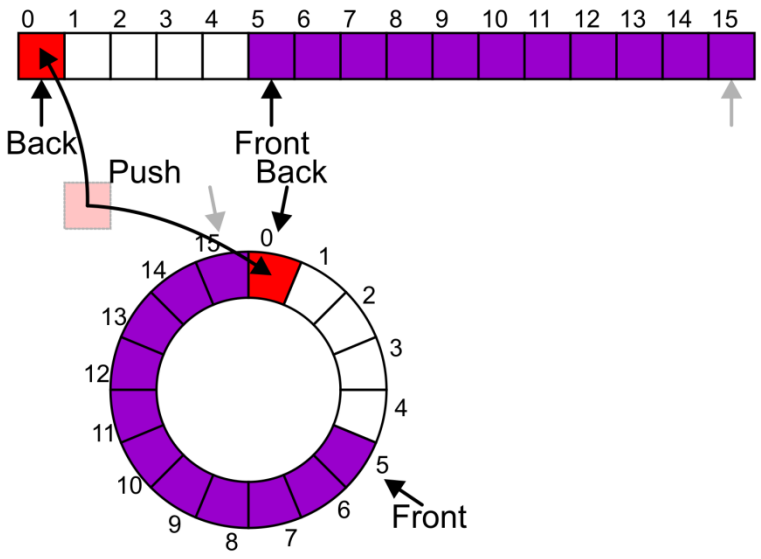


3.3.3.2

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

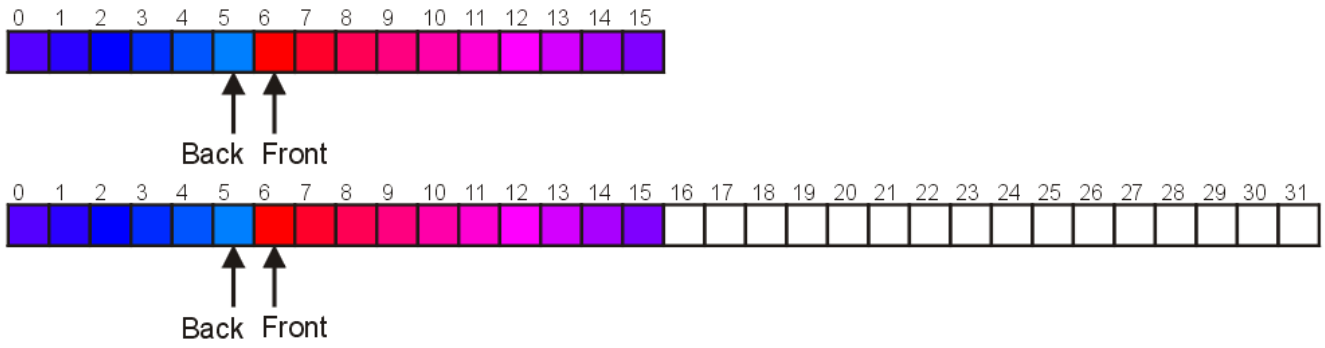
Include a member function **`bool full()`**

3.3.4

Increasing Capacity

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

- A direct copy does not work:

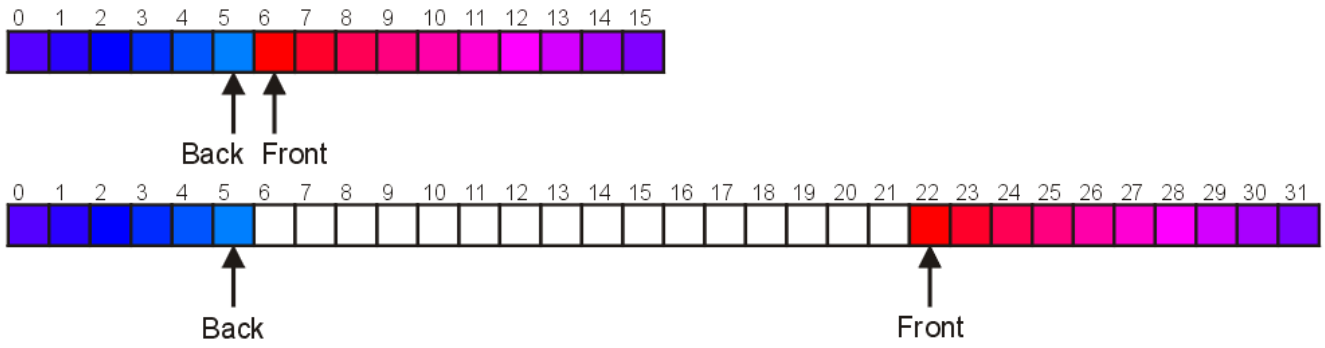


3.3.4

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

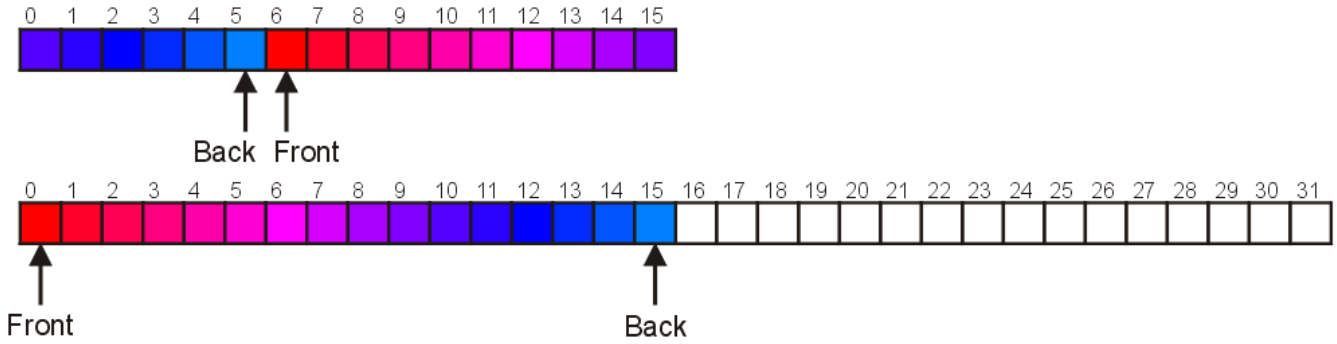


3.3.4

Increasing Capacity

An alternate solution is normalization:

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

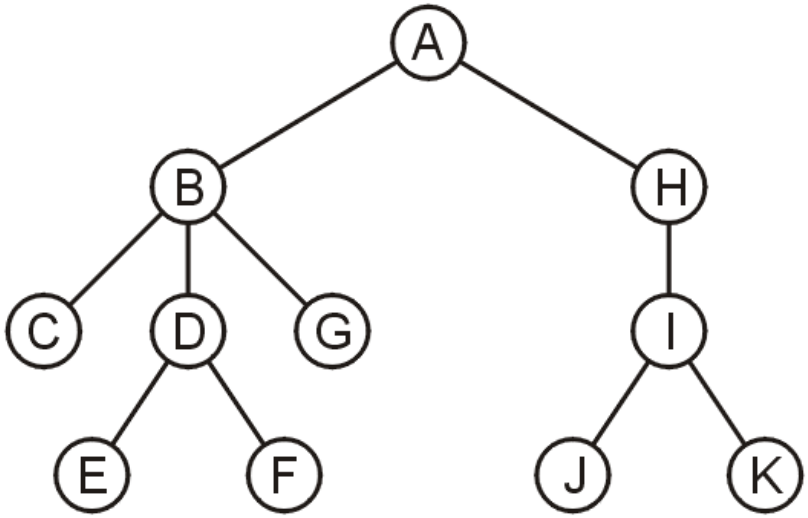


3.3.5

Application

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

- Consider searching the directory structure



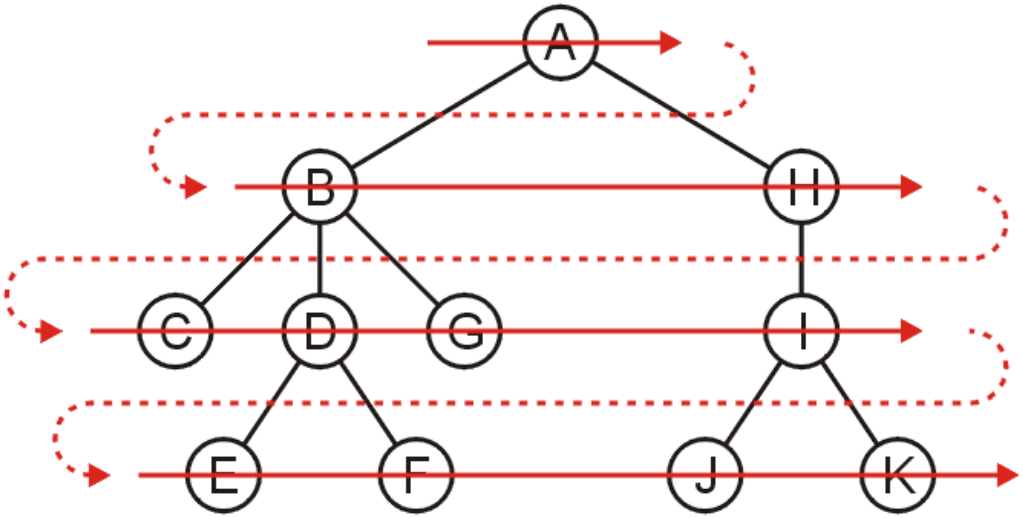
3.3.5

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:

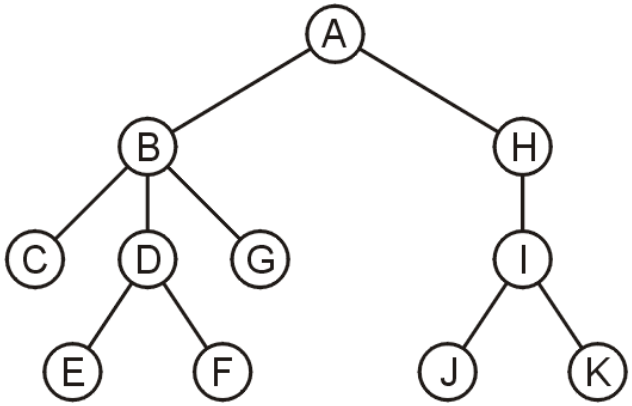
- Place the root directory into a queue
- While the queue is not empty:
  - Pop the directory at the front of the queue
  - Push all of its sub-directories into the queue

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

3.3.5

Application

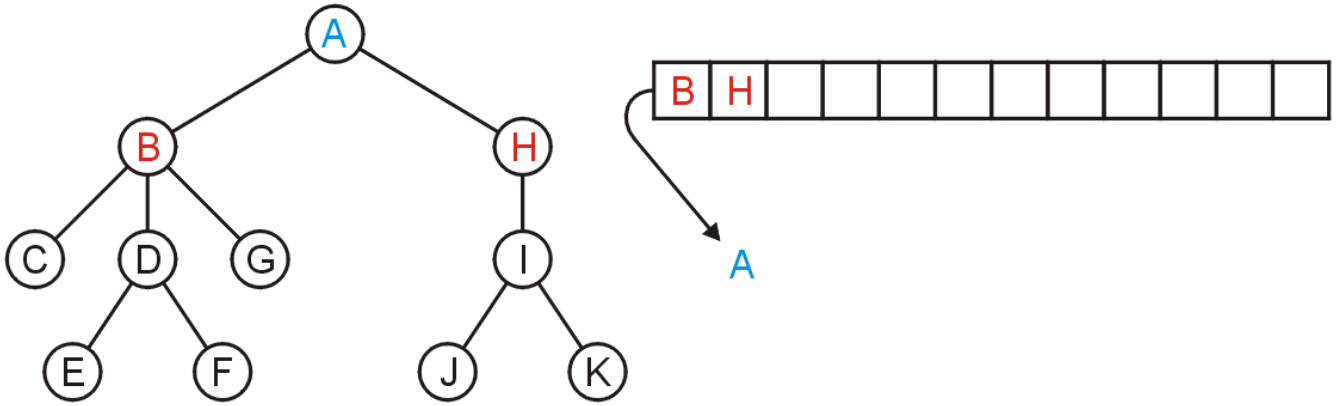
Push the root directory A



3.3.5

Application

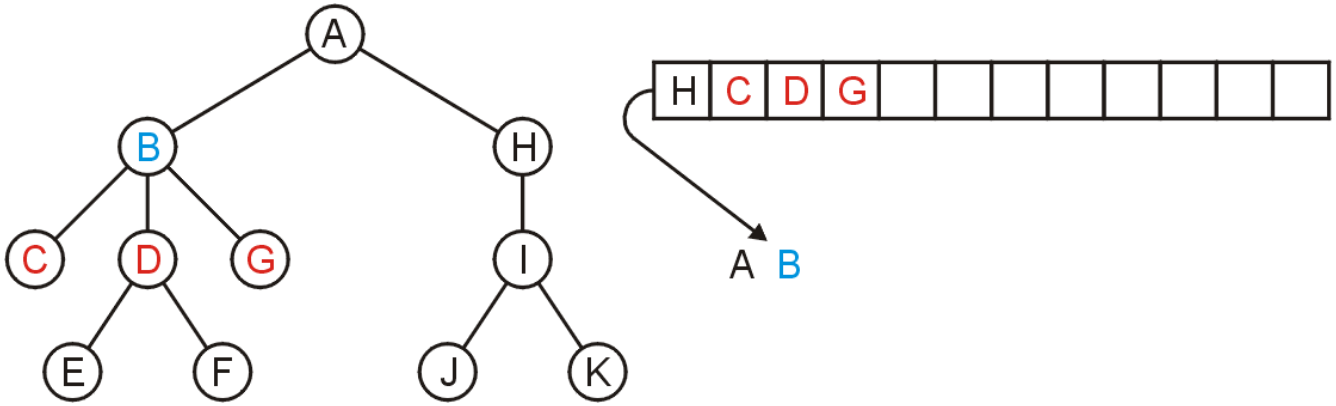
Pop A and push its two sub-directories: B and H



3.3.5

Application

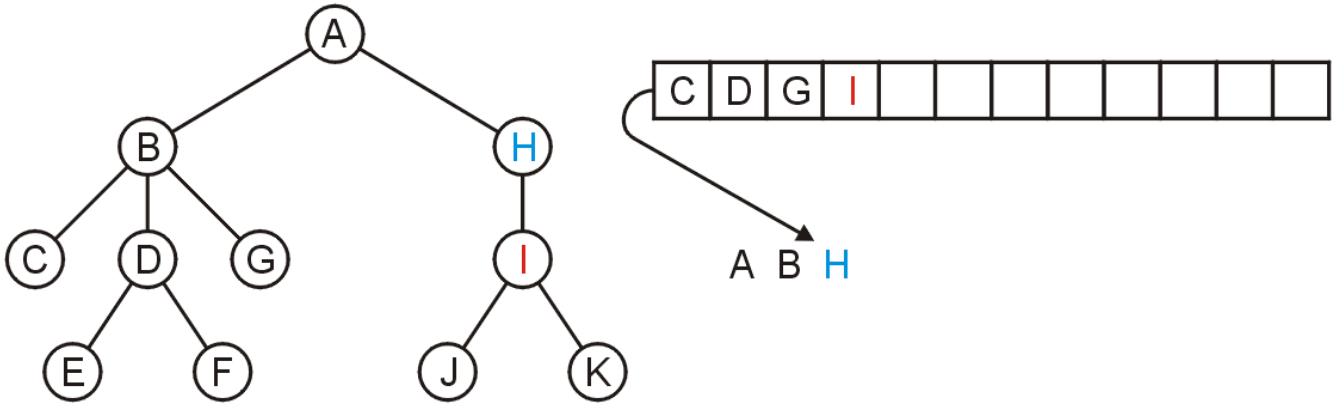
Pop B and push C, D, and G



3.3.5

Application

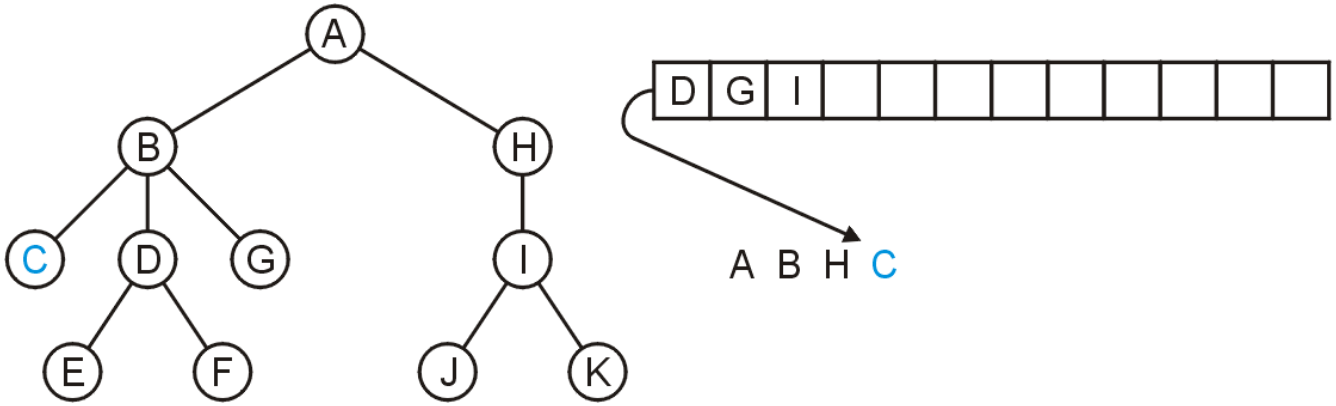
Pop H and push its one sub-directory I



3.3.5

Application

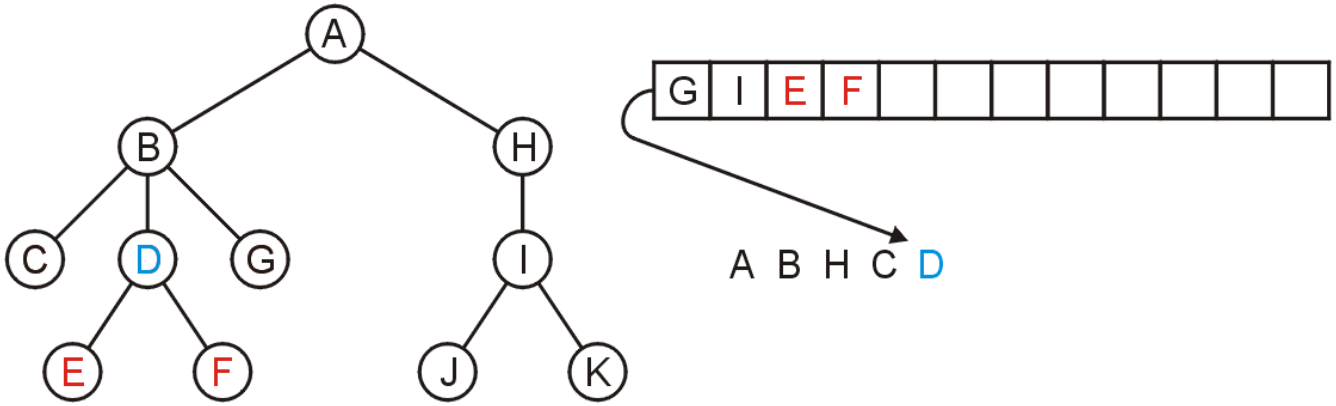
Pop C: no sub-directories



3.3.5

Application

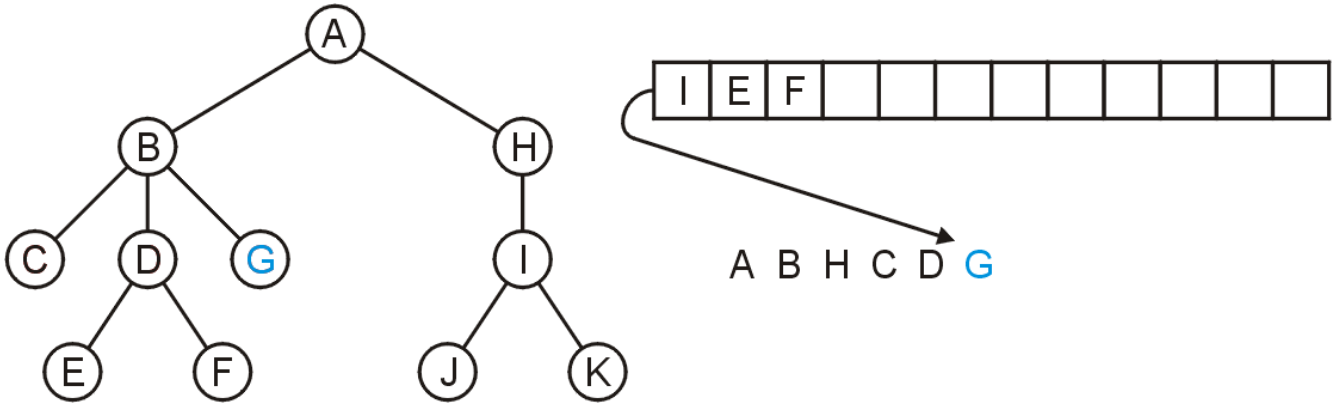
Pop D and push E and F



3.3.5

Application

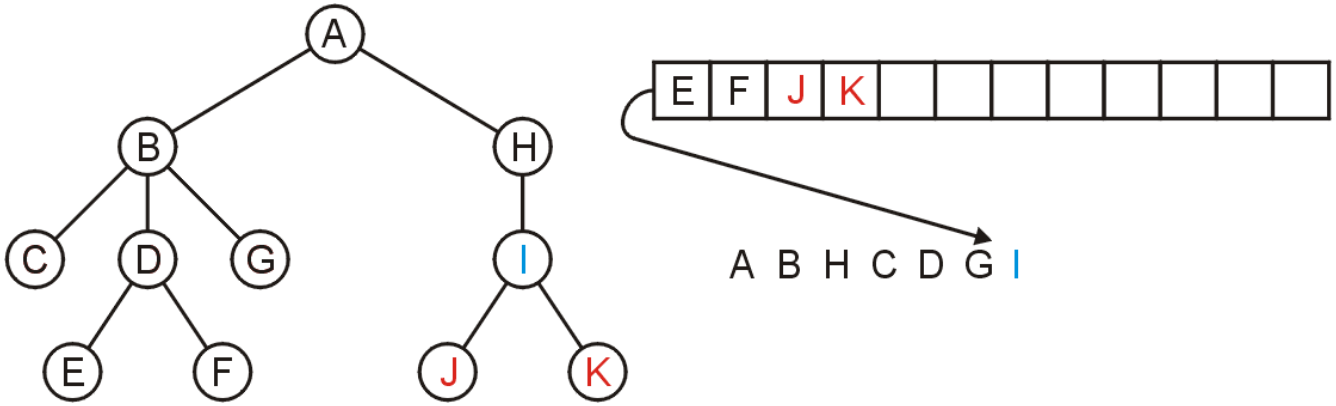
Pop G



3.3.5

Application

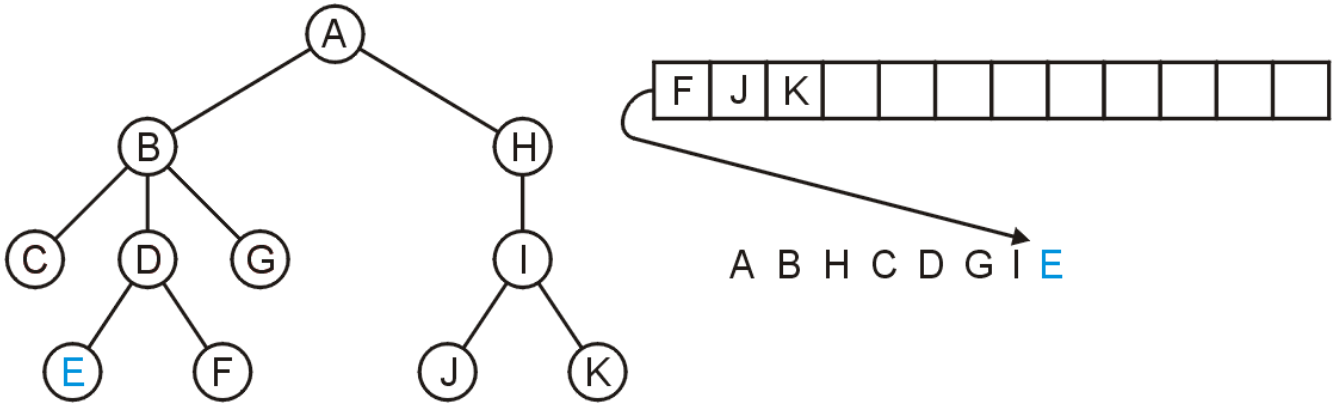
Pop I and push J and K



3.3.5

Application

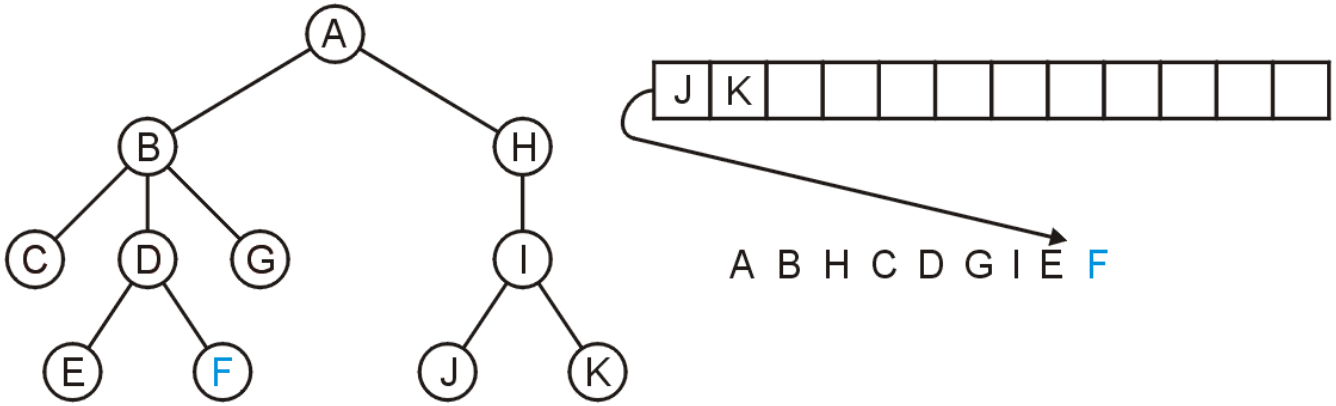
Pop E



3.3.5

Application

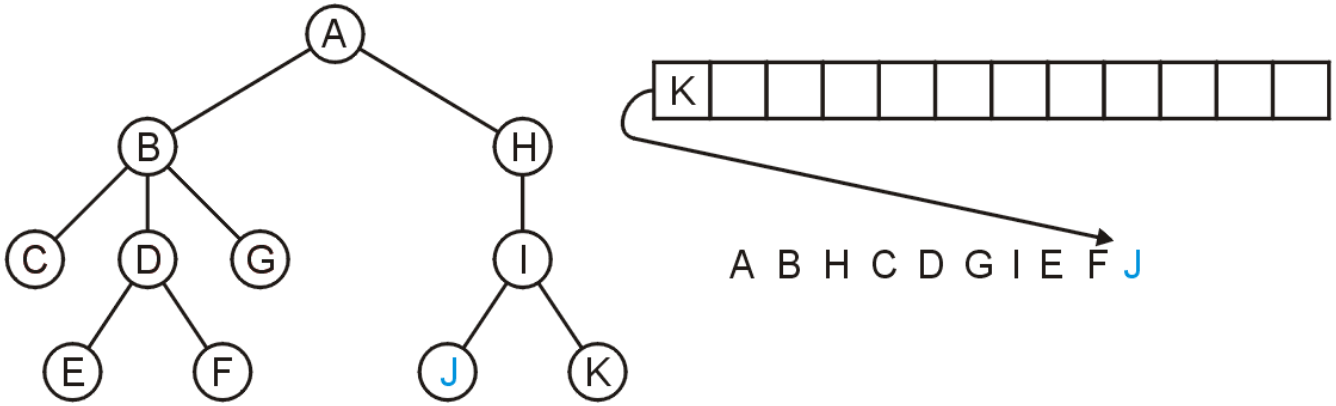
Pop F



3.3.5

Application

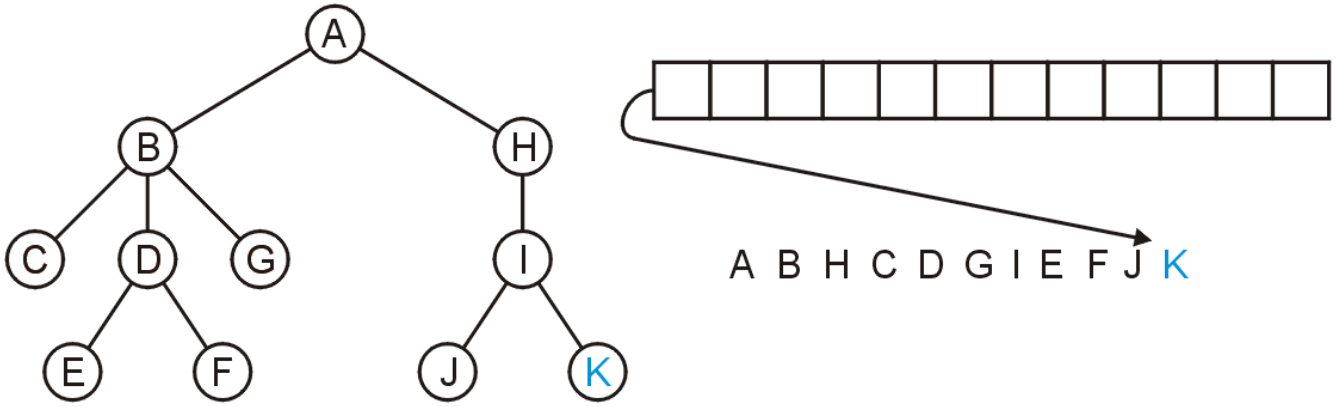
Pop J



3.3.5

Application

Pop K and the queue is empty



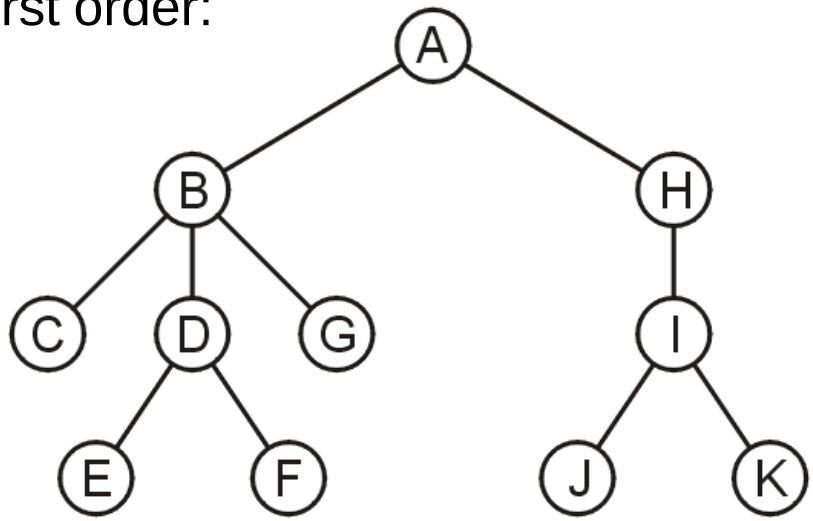
3.3.5

Application

The resulting order

A B H C D G I E F J K

is in breadth-first order:



3.3.6

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

ECE 250 *Algorithms and Data Structures*

An introduction to hash tables

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

WATERLOO
ENGINEERING

Outline

Discuss storing unrelated/unordered data

- IP addresses and domain names

Consider conversions between these two forms

Introduce the idea of hashing:

- Reducing $\mathbf{O}(\ln(n))$ operations to $\mathbf{O}(1)$

Consider some of the weaknesses

Supporting Example

Suppose we have a system which is associated with approximately 150 error conditions where

- Each of which is identified by an 8-bit number from 0 to 255, and
- When an identifier is received, a corresponding error-handling function must be called

We could create an array of 150 function pointers and to then call the appropriate function….

9.1.1.1

Supporting Example

```
#include <iostream>

void a() {
    std::cout
        << "Calling 'void a()'"
        << std::endl;
}

void b() {
    std::cout
        << "Calling 'void b()'"
        << std::endl;
}
```

```
int main() {
    void (*function_array[150])();
    unsigned char error_id[150];

    function_array[0] = a;
    error_id[0] = 3;
    function_array[1] = b;
    error_id[1] = 8;

    function_array[0]();
    function_array[1]();

    return 0;
}
```

Output:

```
% ./a.out
Calling 'void a()'
Calling 'void b()'
```

9.1.1.1

Supporting Example

Unfortunately, this is slow—we would have to do some form of binary search in order to determine which of the 150 slots corresponds to, for example, error-condition identifier `id = 198`

This would require approximately 6 comparisons per error condition

If there was a possibility of dynamically adding new error conditions or removing defunct conditions, this would substantially increase the effort required…

9.1.1.2

Supporting Example

A better solution:

- Create an array of size 256
- Assign those entries corresponding to valid error conditions

```
int main() {
    void (*function_array[256])();
    for ( int i = 0; i < 256; ++i ) {
        function_array[i] = nullptr;
    }

    function_array[3] = a;
    function_array[8] = b;

    function_array[3]();
    function_array[8]();

    return 0;
}
```

Question:

- Is the increased speed worth the allocation of additional memory?

9.1.3

Keys

Our goal:

Store data so that all operations are $\Theta(1)$ time

Requirement:

The memory requirement should be $\Theta(n)$

In our supporting example, the corresponding function can be called in $\Theta(1)$ time and the array is less than twice the optimal size

9.1.3

Keys

In our example, we:

- Created an array of size 256
- Store each of 150 objects in one of the 256 entries
- The error code indicated which bin the corresponding function pointer was stored

In general, we would like to:

- Create an array of size M
- Store each of n objects in one of the M bins
- Have some means of determining the bin in which an object is stored

9.1.3.1

The hashing problem

The process of mapping an object or a number onto an integer in a given range is called *hashing*

Problem: multiple objects may hash to the same value

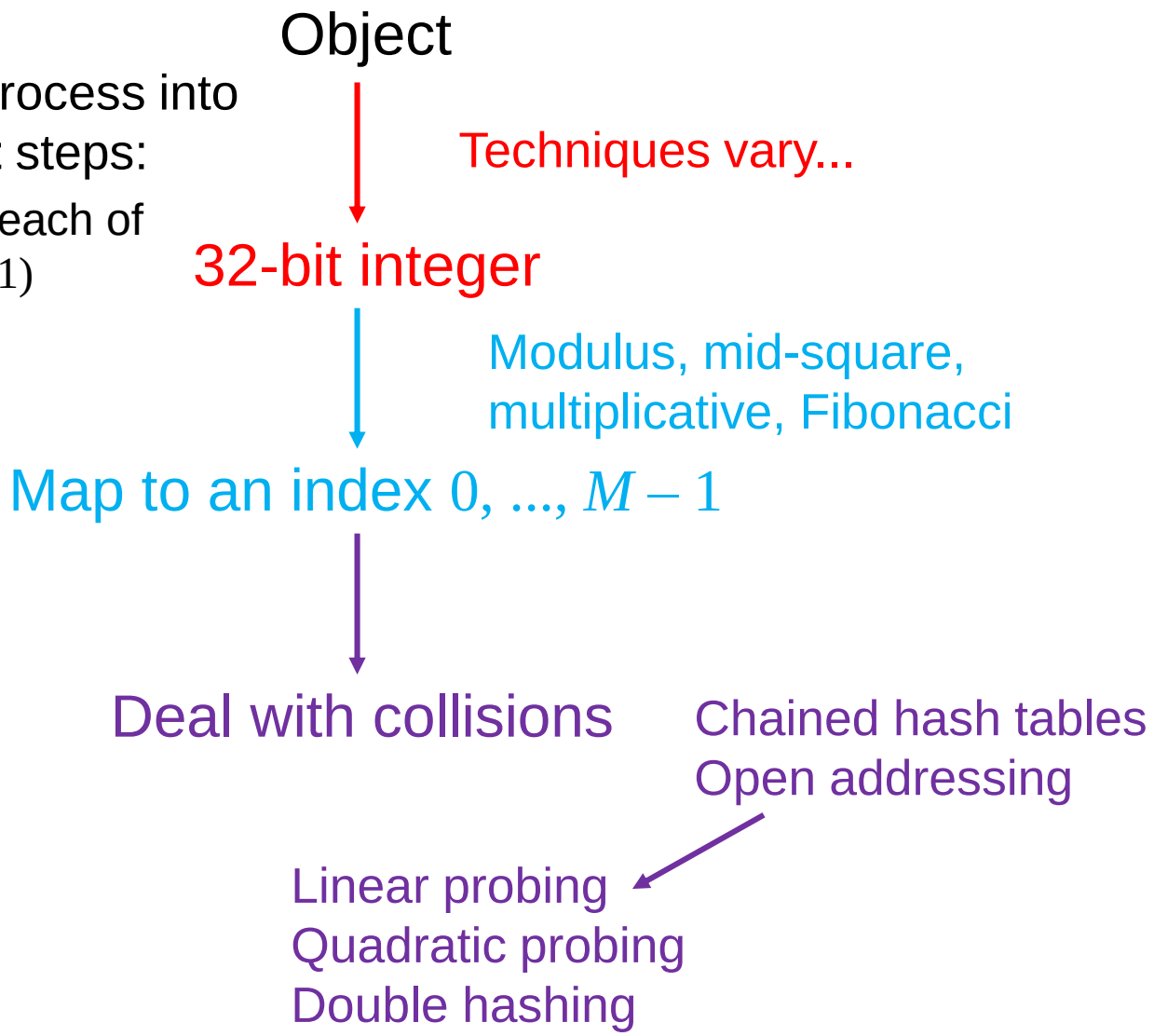
- Such an event is termed a *collision*

Hash tables use a hash function together with a mechanism for dealing with collisions

The hash process

We will break the process into three **independent** steps:

- We will try to get each of these down to $\Theta(1)$



ECE 250 *Algorithms and Data Structures*

Hash functions

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

WATERLOO
ENGINEERING

Outline

In this talk, we will discuss

- Finding 32-bit hash values using:
  - Predetermined hash values
    - Auto-incremented hash values
    - Address-based hash values
  - Arithmetic hash values
- Example: strings

Definitions

What is a hash of an object?

From Merriam-Webster:

a restatement of something that is already known

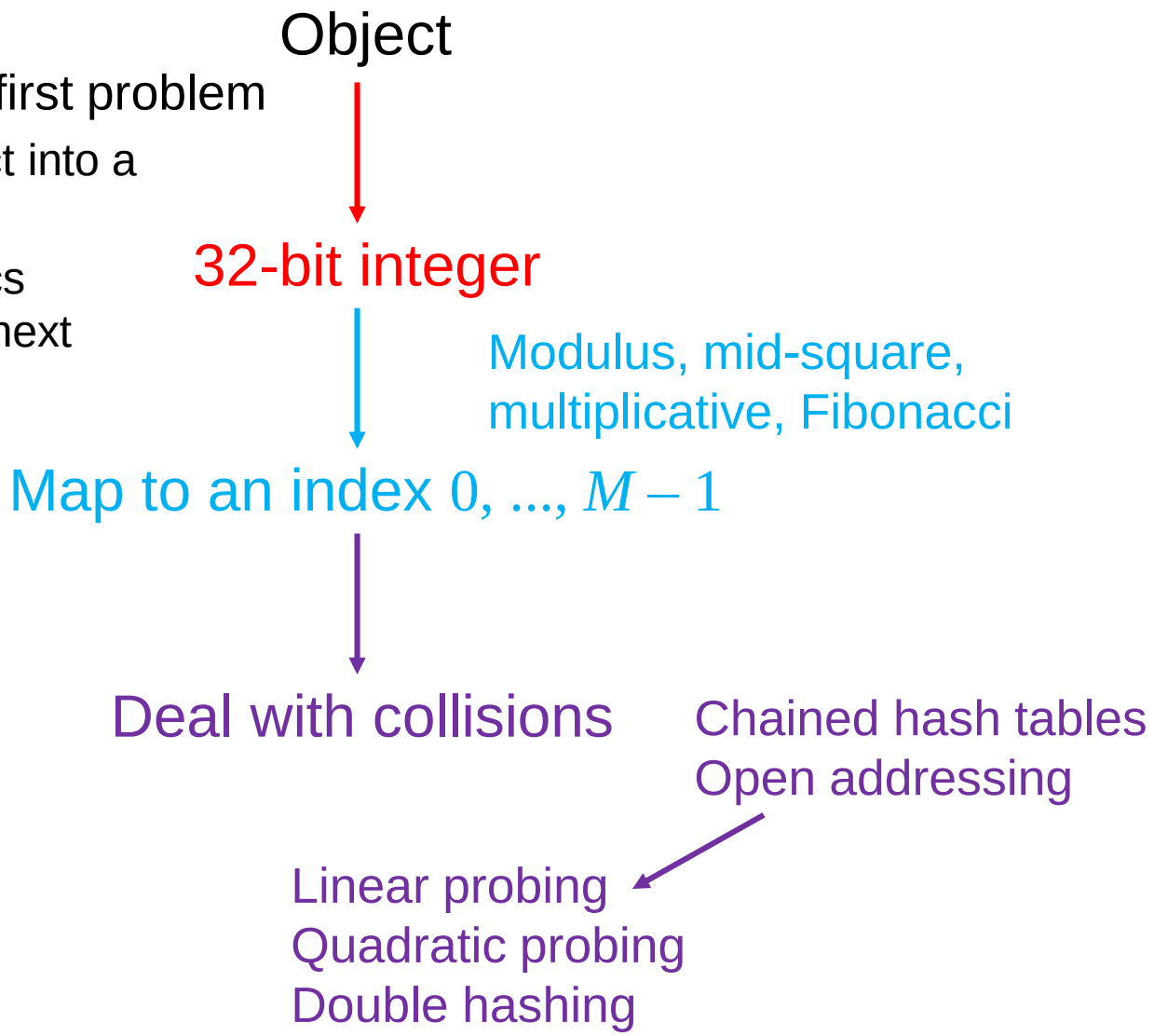
The ultimate goal is to map onto an integer range

`0, 1, 2, ..., M – 1`

The hash process

We will look at the first problem

- Hashing an object into a 32-bit integer
- Subsequent topics will examine the next steps



9.2.2

Properties

Necessary properties of such a hash function h are:

1a. Should be fast: ideally $\Theta(1)$

1b. The hash value must be *deterministic*

- It must always return the same 32-bit integer each time

1c. Equal objects hash to equal values

- $x = y \Rightarrow h(x) = h(y)$

1d. If two objects are randomly chosen, there should be only a one-in-2^{32} chance that they have the same hash value

Types of hash functions

We will look at two classes of hash functions

- Predetermined hash functions (explicit)
- Arithmetic hash functions (implicit)

9.2.4

Predetermined hash functions

The easiest solution is to give each object a unique number

```
class Class_name {
    private:
        unsigned int hash_value;  // int:           -2^31, ..., 2^31 - 1
                                  // unsigned int:    0, ..., 2^32 - 1
    public:
        Class_name();
        unsigned int hash() const;
};
```

```
Class_name::Class_name() {
    hash_value = ???;
}
```

```
unsigned int Class_name::hash() const {
    return hash_value;
}
```

9.2.4

Predetermined hash functions

For example, an auto-incremented static member variable

```
class Class_name {
    private:
        unsigned int hash_value;
        static unsigned int hash_count;
    public:
        Class_name();
        unsigned int hash() const;
};

unsigned int Class_name::hash_count = 0;
```

```
Class_name::Class_name() {
    hash_value = hash_count;
    ++hash_count;
}

unsigned int Class_name::hash() const {
    return hash_value;
}
```

9.2.4

Predetermined hash functions

Examples: All UW co-op student have two hash values:

- UW Student ID Number
- Social Insurance Number

Any 9-digit-decimal integer yields a 32-bit integer

$$\lg(10^9) = 29.897$$

9.2.4

Predetermined hash functions

If we only need the hash value while the object exists in memory, use the address:

```
unsigned int Class_name::hash() const {
   return reinterpret_cast<unsigned int>( this );
}
```

This fails if an object may be stored in secondary memory

- It will have a different address the next time it is loaded

9.2.4.1

Predetermined hash functions

Predetermined hash values give each object a unique hash value

This is not always appropriate:

- Objects which are conceptually equal:

```
Rational x(1, 2);
Rational y(3, 6);
```

- Strings with the same characters:

```
string str1 = "Hello world!";
string str2 = "Hello world!";
```

These hash values must depend on the member variables

- Usually this uses arithmetic functions

9.2.5

Arithmetic Hash Values

An arithmetic hash value is a deterministic function that is calculated from the relevant member variables of an object

We will look at arithmetic hash functions for:

- Rational numbers, and
- Strings

9.2.5.1

Rational number class

What if we just add the numerator and denominator?

```
class Rational {
    private:
        int numer, denom;
    public:
        Rational( int, int );
};

unsigned int Rational::hash() const {
    return static_cast<unsigned int>( numer ) +
        static_cast<unsigned int>( denom );
}
```

9.2.5.1

Rational number class

We could improve on this: multiply the denominator by a large prime:

```
class Rational {
    private:
        int numer, denom;
    public:
        Rational( int, int );
};

unsigned int Rational::hash() const {
    return static_cast<unsigned int>( numer ) +
        429496751*static_cast<unsigned int>( denom );
}
```

9.2.5.1

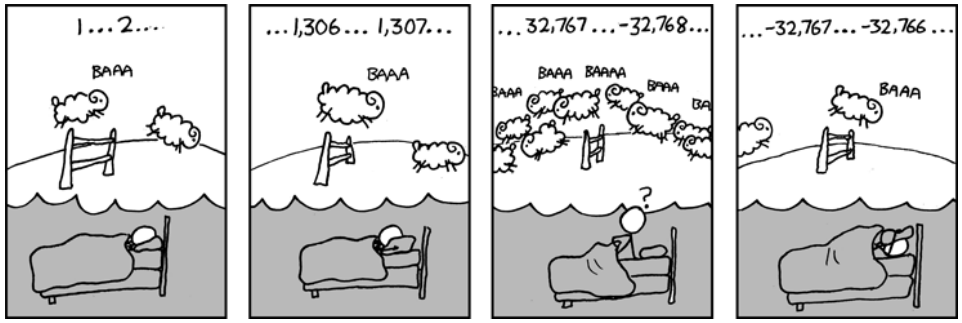
Rational number class

For example, the output of

```
int main() {
    cout << Rational(  0,   1 ).hash() << endl;
    cout << Rational(  1,   2 ).hash() << endl;
    cout << Rational(  2,   3 ).hash() << endl;
    cout << Rational( 99, 100 ).hash() << endl;

    return 0;
}
```

is

```
429496751
858993503
1288490255
2239
```



http://xkcd.com/571/

Recall that arithmetic operations wrap on overflow

9.2.5.1

Rational number class

This hash function does not generate unique values

- The following pairs have the same hash values:

0/1	1327433019/800977868
1/2	534326814/1480277007
2/3	820039962/1486995867

- Finding rational numbers with matching hash values is very difficult:
- Finding these required the generation of $1\,500\,000\,000$ random rational numbers
- It is fast: $\Theta(1)$
- It does produce an even distribution

9.2.5.1

Rational number class

Problem:

- The rational numbers 1/2 and 2/4 have different values
- The output of

```
int main() {
    cout << Rational( 1, 2 ).hash();
    cout << Rational( 2, 4 ).hash();
    return 0;
}
```

is

```
858993503
1717987006
```

9.2.5.1

Rational number class

Solution: divide through by the greatest common divisor

```
Rational::Rational( int a, int b ):numer(a), denom(b) {
    int divisor = gcd( numer, denom );
    numer /= divisor;
    denom /= divisor;
}
```

```
int gcd( int a, int b) {
    while( true ) {
        if ( a == 0 ) {
            return (b >= 0) ? b : -b;
        }

        b %= a;

        if ( b == 0 ) {
            return (a >= 0) ? a : -a;
        }
        a %= b;
    }
}
```

9.2.5.1

Rational number class

Problem:

- The rational numbers $\frac{1}{2}$ and $\frac{-1}{-2}$ have different values
- The output of

```
int main() {
    cout << Rational(  1,  2 ).hash();
    cout << Rational( -1, -2 ).hash();
    return 0;
}
```

is

```
858993503
3435973793
```

9.2.5.1

Rational number class

Solution: define a normal form

- Require that the denominator is positive

```
Rational::Rational( int a, int b ):numer(a), denom(b) {
    int divisor = gcd( numer, denom );
    divisor = (denom >= 0) ? divisor : -divisor;
    numer /= divisor;
    denom /= divisor;
}
```

9.2.5.3

String class

Two strings are equal if all the characters are equal and in the identical order

A string is simply an array of bytes:

- Each byte stores a value from 0 to 255

Any hash function must be a function of these bytes

String class

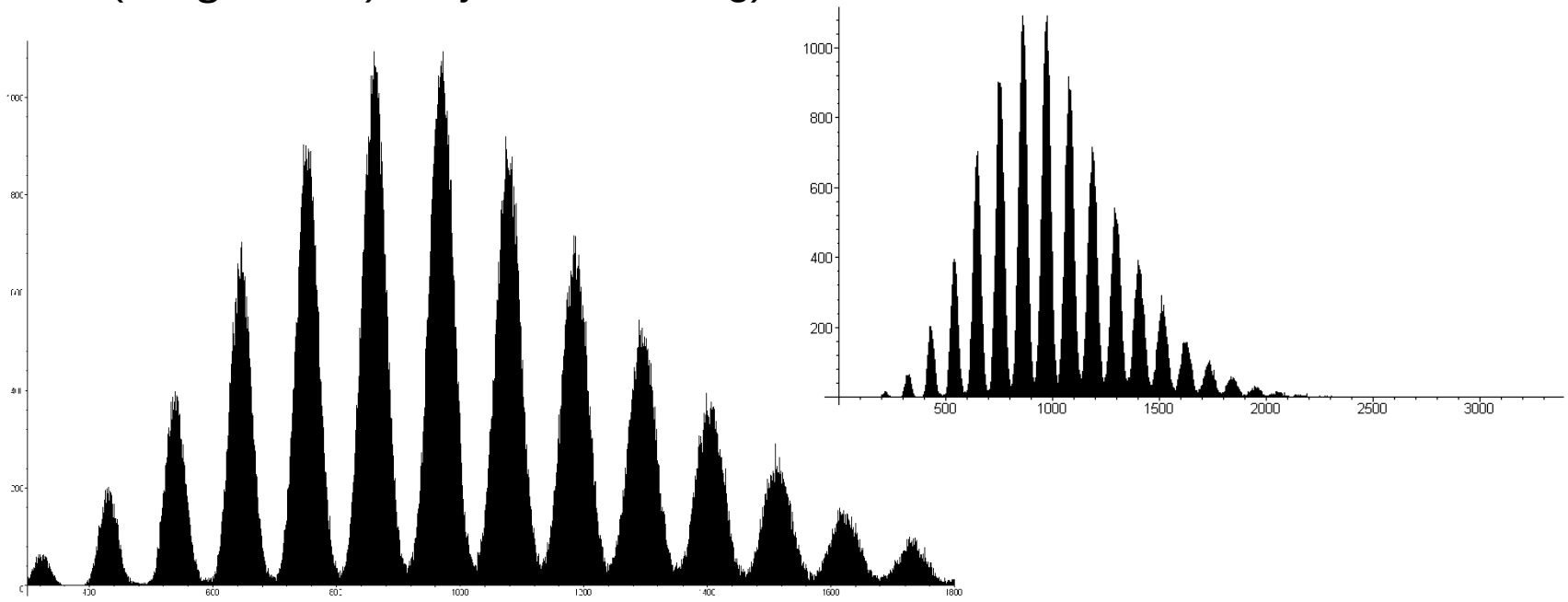
We could, for example, just add the characters:

```
unsigned int hash( const string &str ) {
    unsigned int hash_vaalue = 0;

    for ( int k = 0; k < str.length(); ++k ) {
        hash_value += str[k];
    }

    return hash_value;
}
```

String class

Not very good:

- Slow run time: $\Theta(n)$
- Words with the same characters hash to the same code:
  - `"form"` and `"from"`
- A poor distribution, *e.g.*, all words in Moby™ Words II by Grady Ward (`single.txt`) Project Gutenberg):



String class

Let the individual characters represent the coefficients of a polynomial in x:

$$p(x) = c_0 x^{n-1} + c_1 x^{n-2} + \cdots + c_{n-3} x^2 + c_{n-2} x + c_{n-1}$$

Use Horner's rule to evaluate this polynomial at a prime number, *e.g.*, $x = 12347$:

```
unsigned int hash( string const &str ) {
    unsigned int hash_value = 0;

    for ( int k = 0; k < str.length(); ++k ) {
        hash_value = 12347*hash_value + str[k];
    }

    return hash_value;
}
```

9.2.5.3.3

Arithmetic hash functions

In general, any member variables that are used to uniquely define an object may be used as coefficients in such a polynomial

- The salary hopefully changes over time…

```
class Person {
    string surname;
    string *given_names;
    unsigned char num_given_names;
    unsigned short birth_year;
    unsigned char birth_month;
    unsigned char birth_day;
    unsigned int salary;
    // ...
};
```

Summary

We have seen how a number of objects can be mapped onto a 32-bit integer

We considered

- Predetermined hash functions
  - Auto-incremented variables
  - Addresses
- Hash functions calculated using arithmetic

Next: map a 32-bit integer onto a smaller range $0, 1, ..., M-1$

ECE 250 *Algorithms and Data Structures*

Abstract Priority Queues

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

WATERLOO
ENGINEERING

7.1.1

Definition

With queues

- The order may be summarized by *first in, first out*

If each object is associated with a priority, we may wish to pop that object which has highest priority

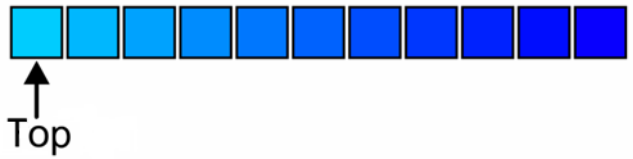
With each pushed object, we will associate a nonnegative integer (0, 1, 2, ...) where:

- The value 0 has the *highest* priority, and
- The higher the number, the lower the priority

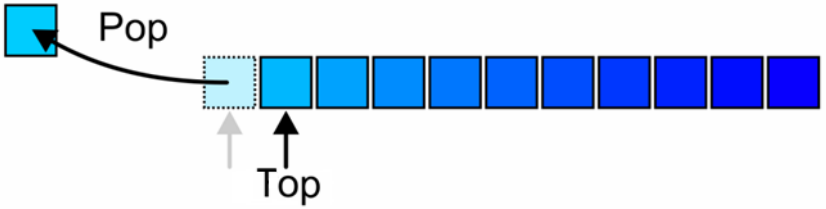
7.1.2

Operations

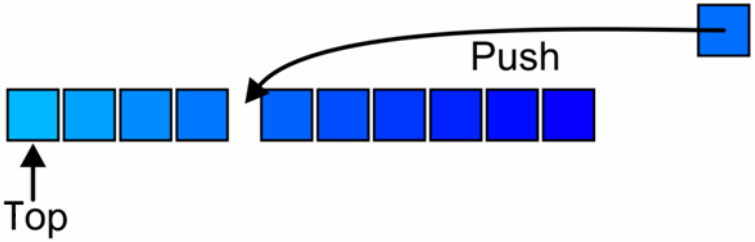
The top of a priority queue is the object with highest priority



Popping from a priority queue removes the current highest priority object:



Push places a new object into the appropriate place



7.1.3

Lexicographical Priority

Priority may also depend on multiple variables:

- Two values specify a priority: (a, b)
- A pair (a, b) has higher priority than (c, d) if:
  - $a < c$, or
  - $a = c$ and $b < d$

For example,

- (5, 19), (13, 1), (13, 24), and (15, 0) all have *higher* priority than (15, 7)