ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo

2.1 Containers, Relations, and Abstract Data Types

We use mathematics to model the real world—systems are described by differential equations and we can simulate real systems by solving those differential equations. Similarly, before we begin writing code, it is appropriate to model our data using a similar level of abstraction.

In general, we need to store, access, manipulate, and remove related data in memory. In each case, we must apply engineering analysis to determine those operations that we need to perform and engineering design to achieve those results in the least time and memory.

We will use the term *Abstract Data Type* (or ADT) to model the storage, access, manipulation, and removal of related data. These operations will be divided into two general categories:

1. Queries that determine the properties of the objects that are stored, and
2. Data manipulations that modify the objects being stored.

This is a high-level description which will then be implemented in various programming languages uses the appropriate data structures and algorithms.

2.1.1 The Container ADT or the Abstract Container

The most general Abstract Data Type is that of a *container*. This is a general model of any object that is meant to store and give access to data. The queries and data manipulations that can be performed on a container are quite general in nature:

Table 1. Possible manipulations on a container.

Manipulation	Standard Template Library Equivalent
Create a new container	`Container()`
Copy an existing container and its contents	`Container( Container const & )`
Destroy a container	`~Container()`
Empty a container	`void clear()`
Take the union of (or *merge*) two containers	`void insert( Container const & )`
Find the intersection of two containers	no equivalence

Table 2. Possible queries on a container.

Query	Standard Template Library Equivalent
Is the container empty?	`bool empty() const`
How many objects are in the container?	`int size() const`
What is the maximum capacity of the container?	`int max_size() const`

Table 3. Possible data manipulations and queries on the objects in a container.

Query	Standard Template Library Equivalent
Insert an object into the container	`void insert( Type const & )`
Remove an object from the container	`void erase( Type const & )`
Access or modify an object in the container	`iterator find( Type const & ) const`
Determine the number of copies of an object that are in a container	`int count( Type const & ) const`
Iterate (or step) through the objects currently in the container	`iterator begin() const`

2.1.2 Simple and Associative Containers

A container may store a general collection of objects, or it may store keys each of which are associated with a corresponding record.

For example, a temperature sensor may be reading the temperature of the environment or of a component within a system. This value may be stored in a container waiting for another task to read the values and determine if any action must be taken as a result of changes in the temperature.

Alternatively, Quest stores student records based on the unique key of each student's UW Student ID Number. When an academic advisors access your records, they will enter your UW Student ID Number and Quest will return a record containing all information relevant to your undergraduate career.

The first type of container that stores only objects will be called a *simple container* while the second will be called an *associative container*—it associates the keys it stores with corresponding records.

We will focus on simple containers: any simple container can be modified to create an associative container.

2.1.3 Unique or Duplicate Objects

By design, a container may

1. Require that all objects in the container are unique, or
2. Allow duplicate objects.

In general, data manipulations that are required to deal with duplicate objects requires additional, sometimes subtle, code that, for the most part, does not provide further insight into the algorithms.

We will assume, unless otherwise stated, that all objects within a container are unique.

2.1.4 The Standard Template Library (STL)

The STL has four containers related to the possible requirements in Sections 2.1.2 and 2.13 and these are listed in Table 4.

Table 4. Containers within the Standard Template Library.

	Unique Objects/Keys	Duplicate Objects/Keys
Simple Container of objects	`set<T>`	`multiset<T>`
Associative Containers of key-object pairs	`map<K, T>`	`multimap<K, T>`

An example of the set<T> container is provided here:

```
#include <iostream>
#include <set>
int main() {
    std::set<int> ints;

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

    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;
}
```

2.1.5 Relationships

In general, we may not only want to store data, but we may need to also store relationships between the stored data, for example:

1. A genealogical database must not only store people but must also store family relations, and
2. maps.google.ca must not only store roads but how those roads are connected (they actually store intersections and for each intersections, which other intersections are adjacent to it.

2.1.5.1 No Relationships Required

If it is not necessary to store any relationship between the data in a container, the optimal data structure is often the *hash table*. This will be described in Topic 6. In general, a hash table simplifies the representation of an object (a *hash value*) and then organizes the data based on that hash value.

For example, every person can be represented by the first letter of his or her surname. In an organization with two dozen members where membership frequently changes, it is easiest to set up 24 mailboxes and all mail to, for example, Douglas Harder, is placed in box H. While Douglas may have to sort through mail for Andrew Heunis, this is still much easier than searching through a pile of mail.

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

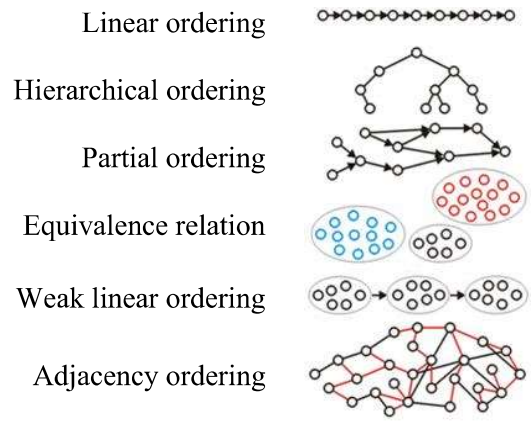
This will work so long as there are not too many individuals with surnames ending in the same letter (*e.g.*, Smith, Singh, Sánchez, Simon, Schmidt, Schneider, *etc.*). A hash table is a programmatic implementation of this idea.

2.1.5.2 Queries on Relationships

In general, however, there are questions that are often asked with respect to a relation:

1. Which objects have been stored in the container the longest?
2. Are these two C++ classes derived from the same base class?
3. Can I take ECE 427 if I fail ECE 250?
4. Do both pixels on an image represent the pavement features?
5. Can I get from here to Roches's Point in under two hours?

Thus, we will look at five different relationships:



These relationships are described using Boolean-valued binary operators:

1. Are two values equal? $x = y$
2. Is one value less than the other? $x < y$
3. Does one value divide the other? $x \mid y$

In other cases, the relationship is more generally described by a Boolean-valued question: "Do two values have the same remainder modulo 16?" or "Do two integers differ by at most one prime factor?"

2.1.5.3 Symmetry, Anti-symmetry and Transitivity

We will use various symbols to represent relationships based on their properties: $x \sim y$, $x \prec y$, $x < y$ and $x \to y$. We will use the symbol as appropriate based on the properties of the relationship:

1. A relationship is *symmetric* if $x \sim y$ if and only if $y \sim x$,
2. A relationship is *anti-symmetric* if at most one of $x \prec y$ or $y \prec x$ is true,
3. For a relationship that is anti-symmetric with two different object x and y, it is necessary that exactly one is true, we will use $x < y$ or $y < x$, and
4. If $x \to y$ gives no relationship as to whether $y \to x$, we will simply use the arrow.

A common property of relationships is *transitivity*: often it is the case that if two relationships hold, a third can be deduced. For example:

> If $x \sim y$ and $y \sim z$, we may deduce that $x \sim z$.
> If $x \prec y$ and $y \prec z$, we may deduce that $x \prec z$.

2.1.6 Examples of Relationships

We will now define the six relationships and consider various examples.

2.1.6.1 Linear Orderings

A linear ordering is any relationship where exactly one of

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

is true and where the relationship is transitive. All linear orderings are therefore anti-symmetric.

It is always possible to create such a sequence for linearly orderings. Such a sequence is said to be *sorted*.

Examples of sets that can be linearly ordered include

1. the integers,
2. real or floating-point numbers,
3. the alphabet, and
4. memory locations.

2.1.6.1.1 Lexicographical Orderings

Consider words in the English language: these are linearly ordered in the dictionary, for example,

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

This gives us a linear ordering induced by the linear ordering of the letters of the alphabet. For this to make sense, it is necessary to append blank characters ␣ after shorter words where ␣ < a. In each case, the order is determined by comparing the letters until the first different letter is found. This letter determines the order of the two words. For example,

aardvark < **aardw**olf	since v < w
aardvark < **ab**acus	since a < b
azygous < **b**aa	since a < b
cat < **cat**ch	since ␣ < c
wilhelm < **will**iam	since h < l

Generalizing this, we could linearly order vectors in the same way:

Define $(x_1, y_1) < (x_2, y_2)$ if either $x_1 < x_2$ or both $x_1 = x_2$ and $y_1 < y_2$.

In this case, (1, 3) < (1, 5) < (2, 1) < (2, 17) < (3, 4).

2.1.6.1.2 Queries and operations on Linear Orderings

Queries that may be made about linear orderings include:

1. What are the first and last objects (what are the *front* and *back*)?
2. What is the k^{th} object?
3. What are all objects that fall on an interval $[a, b]$?
4. Given an object x, what is either the previous smaller or next largest object?

Operations that may be made on linearly ordered data include:

1. Insert an object into a sorted list.
2. Explicitly insert an object either at the front, back, or at the k^{th} location.
3. Sort a collection of objects.

2.1.6.2 Hierarchical Orderings

A hierarchical ordering is one where $x \prec y$ implies that x precedes y in some respect. There is one ultimate object, the *root* of the hierarchy that precedes all other objects, but it is not necessarily possible to compare two arbitrary entries.

As an example are directories in Unix where there is only a single root directory /. Here, $x \prec y$ if x contains y within one of its sub-directories. A hierarchical ordering does not allow the objects to be shown in a list. Instead, we need a tree structure, as is shown in Figure 1.

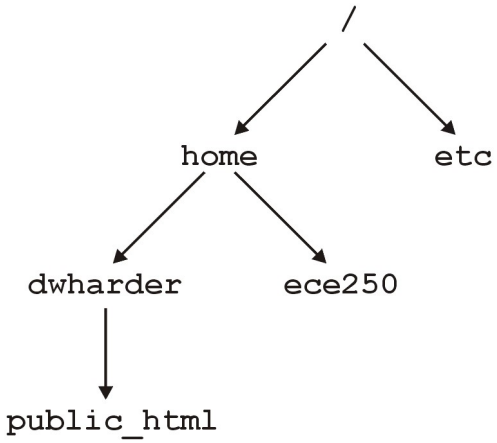


Figure 1. A directory structure in Unix.

Notice in such a tree, there can never be any loops.

2.1.6.2.1 Examples of Hierarchical Orderings

Examples of information organized hierarchically:

1. The organization of directories in a file system,
2. The hierarchy of inheritance in Java and C# with the class `Object` preceding all other classes,
3. The organization of corporations, governmental ministries, and the military, and
4. The scope of variables in a C++ function.

2.1.6.2.2 Properties of Hierarchical Orderings

The properties of a hierarchical relationship include:

1. It is never true that $x \prec x$,
2. If $x \prec y$, $y \not\prec x$,
3. If $x \prec y$ and $y \prec z$, $x \prec z$,
4. There is a root r such that $r \prec x$ for all other x, and
5. If $x \prec z$ and $y \prec z$, either $x \prec y$, $x = y$, or $x \succ y$.

2.1.6.2.3 Queries on Hierarchical Orderings

Queries that may be made on hierarchical orderings include:

1. Given two objects, does one precede the other?
2. Do two objects have the same immediate predecessor?
3. Given two objects, what is the nearest common predecessor?

2.1.6.3 Partial Orderings

A partial ordering has even fewer restrictions that a hierarchical ordering. Consider the relationship $x \prec y$ where x is a task that must be completed before starting another task y. For example, one set of tasks are courses taken by undergraduates. In this case, ECE 150 $\prec$ ECE 155 $\prec$ ECE 250. In this case, ECE 150 is said to be a prerequisite of ECE 155 and ECE 250, and ECE 155 is a prerequisite of ECE 250. Note that flow can only be in one direction—it is not possible to create a loop.

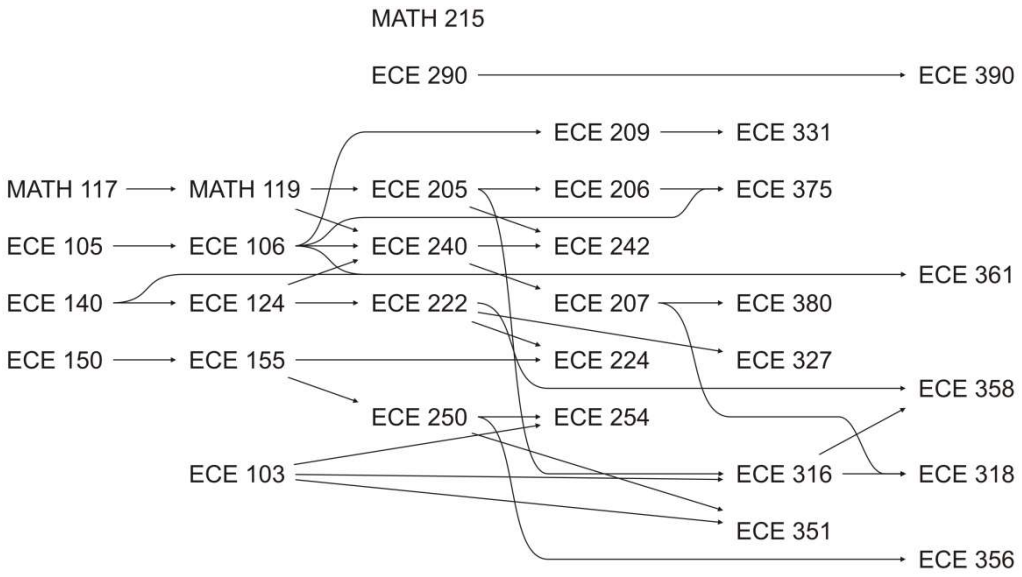


Figure 2. A partial ordering.

It is not possible for x be a prerequisite of y and, for y to be a prerequisite of z, and at the same time z to be a prerequisite of x—otherwise it would be impossible to take any of these courses. At the same time, there may be many courses that have no prerequisites but are prerequisites for other courses. There are many other cases where such situations may exist in real life, as is discussed in the next example.

2.1.6.3.1 Examples of Partial Orderings

Examples of objects that are partially ordered include:

1. The ordering of C++ classes which allows multiple inheritance and does not contain an ultimate class similar to that of `Object` in Java or C#,
2. The ordering of tasks where certain tasks require the completion of others before it is possible to proceed. This includes compilation dependencies (a change in one C++ file may require a second to be recompiled as the second included the first using `#include`).

2.1.6.3.2 Properties of Partial Orderings

The properties of a partial ordering relationship include:

1. It is never true that $x \prec x$,
2. If $x \prec y$, $y \not\prec x$, and
3. If $x \prec y$ and $y \prec z$, $x \prec z$.

In the diagram in Figure 3, a node x precedes another y if there is a path from x to y. Three are two nodes which have no predecessors and three nodes which have no successors.

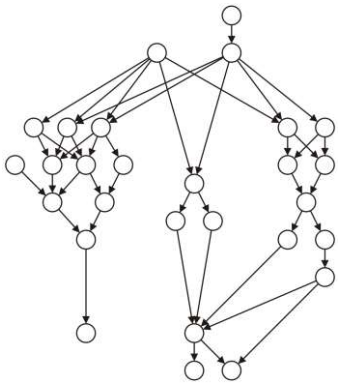


Figure 3. A partially ordered set.

You could think of these as tasks that must be performed: Initially, it is only possible for three tasks to be performed—the three that have no predecessors. You could think of these as courses with prerequisites: three courses have no prerequisites. These courses, however, are prerequisites for others, so taking a course offers new possible courses you can take after having finished the prerequisite.

In addition, there may be multiple paths between two nodes—if such a path exists in a hierarchical ordering, that path is unique. In a partial ordering, there may be multiple paths from one node to another.

2.1.6.3.3 Lattices

A finite lattice L is a partial ordering where there is one objects $\perp$ that precedes all others objects and another object $\top$ that succeeds all other objects. That is, $\perp \preccurlyeq x$ and $x \preccurlyeq \top$ for all elements $x \in L$.

For example, consider the lattice of sets where a set $A \prec B$ if $A \subset B$ as shown in Figure 4.

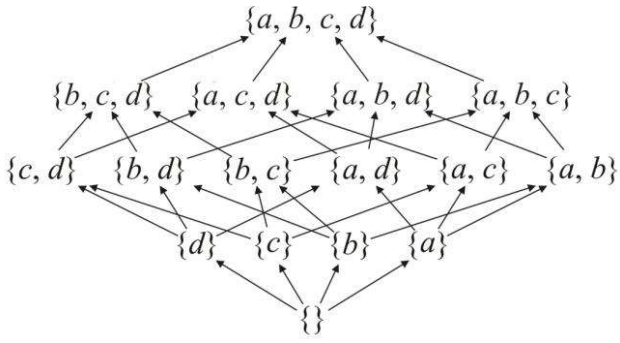


Figure 4. A lattice of sets.

ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo

Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 2.1.
Assistances and comments will be acknowledged.

2.1.6.3.4 Queries on Partial Orderings

Queries that may be made on a set with a partial ordering include:

1. Given two objects, does one precede the other?
2. Which objects have no predecessors?
3. Which objects have no successors?
4. Which objects immediately precede a given object?
5. Which objects immediately succeed a given object?

2.1.6.4 Equivalence Relations

An equivalence relation where two objects are related (*i.e.*, *equivalent*) or not. The easiest example is $x \sim y$ if x and y have the same gender. Based on chromosomes to determine gender, we have approximately four classes of genders, as shown in Figure 5.

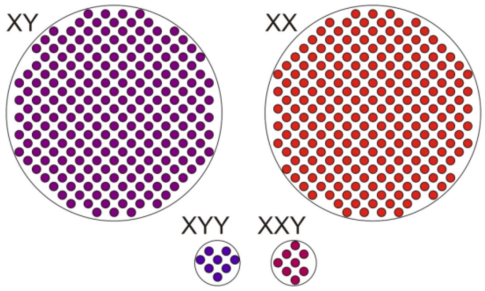


Figure 5. Classes of individuals with the same gender.

2.1.6.4.1 Properties of Equivalence Relations

The properties that make equivalence relations useful are:

1. $x \sim x$ for all x,
2. $x \sim y$ if and only if $y \sim x$, and
3. If $x \sim y$ and $y \sim z$, $x \sim z$.

Any relationship that satisfies these properties allows all objects to be grouped into *equivalence classes* where all objects within a class are related to each other.

2.1.6.4.2 Examples of Equivalence Relations

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 $0 < c < \infty$. Similarly, two circuits made by different manufacturers may be said to be equivalent if they have the same functionality.

2.1.6.4.3 Queries and Data Manipulations of Equivalence Relations

Queries that may be made on equivalence relations include:

1. Are two objects equivalent?
2. What is the number of objects that fall within one equivalence class?
3. What are all the objects in a given equivalence class?

Date manipulations includes the possibility of, given two equivalence classes, merging them into one equivalence class.

2.1.6.5 Weak Orderings

A weak ordering is a linear ordering of equivalence classes. For example, consider ordering people by age: $x < y$ if x is younger than y and $x \sim y$ (equivalent) if they have the same age. This is shown in Figure 6.

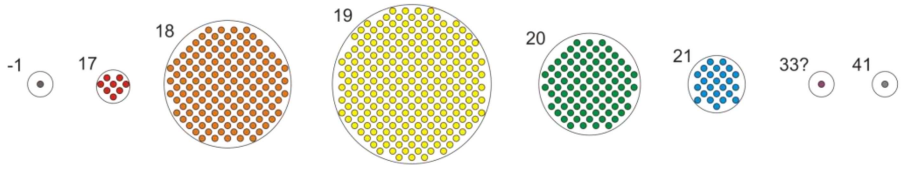


Figure 6. A weak ordering.

2.1.6.5.1 Examples of Weak Orderings

The Standard Template Library (STL) containers `set<T>`, `multiset<T>`, `map<K, T>`, and `multimap<K, T>` use weak orderings. The classes `set` and `map` allow only one object per equivalence class while `multiset` and `multimap` allow multiple objects per equivalence class. There is no guarantee of the ordering of objects that are equivalent.

2.1.6.5.2 Queries and Data Manipulations on Weak Orderings

All the queries and data manipulations that may be applied to either linear orderings or equivalence relations may be applied to weak orderings.

2.1.6.6 Adjacency Relations

Consider the relationship $x \leftrightarrow y$ if x and y are friends. This creates a *graph* of individuals, as is shown in Figure 7.

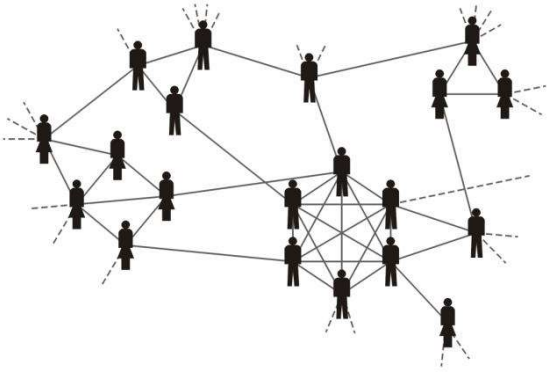


Figure 7. A graph of friendship.

There is no guarantee that friendship is transitive. The mathematical description is to call the individuals *vertices* and those lines that indicate friendships are *edges*. In this case, friendship is symmetric: $x \leftrightarrow y$ if and only if $y \leftrightarrow x$. It is, however, possible to define non-symmetric relationships. Consider, for example, intersections (vertices) and streets connecting those intersections (edges). If a road is one-way, it is possible to legally get from x to y, but not to get from y to x directly. In this case, we write $x \rightarrow y$.

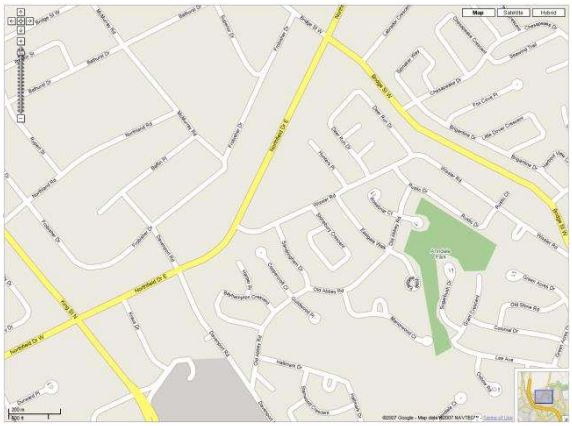


Figure 8. North-east Waterloo from maps.google.ca.

In addition to having a Boolean-valued relationship, it is also possible to have a weight associated with each edge. For example, following Northfield Dr., the length of the road connecting the vertices is shown in Figure 9.

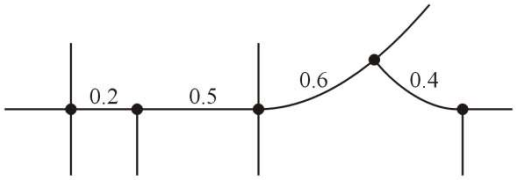


Figure 9. The weights on edges connecting vertices.

2.1.6.6.1 Queries and Manipulations on Adjacency Relations

Queries that may be made include:

1. Are two vertices adjacent?
2. What are all the vertices adjacent to a given vertex?
3. Given two vertices a and b, is there a sequence of vertices $a = x_0, x_1, x_2, ..., x_n = b$ such that $x_k \sim x_{k+1}$? (If so, we say a and b are *connected*.)
4. What are all vertices connected to a given vertex?

You may notice that connected regions of a symmetric adjacency relation form equivalence relations based on $x \sim y$ if x is connected to y.

Common manipulations may be two introduce new relations (for example, following the construction of a road) or remove a relation (if a road has been closed for repair).

2.1.7 Defining Relationships

A relationship may be defined globally or locally and implicitly or explicitly.

2.1.7.1 Globally and Locally Defined Relationships

A relationship is defined globally if any two objects x and y can be immediately compared. For example, the linear ordering of the real numbers is globally defined.

Alternatively, a relationship may be defined pair-wise. For example, in a corporation, a new employee is generally introduced to his or her superior and his or her subordinates. Given two random employees at RIM, it might be difficult determining the exact relationship between them. Such a relationship is said to be locally defined.

2.1.7.2 Implicitly and Explicitly Defined Relationships

A relationship that is based on the properties of the objects themselves is said to be implicitly defined. Alternatively, it is possible for a programmer to explicitly give a set of objects a relationship.

For example, the real numbers are implicitly linearly ordered whereas the characters in this paragraph are explicitly linearly ordered by the programmer. The programmer could choose to use implicit ordering for eervy egilns dorw; eehorvw, hist dlouw be very difficult to read.

2.1.7.3 Relations between Implicit/Explicit and Local/Global Relationships

In general, however, globally defined relationships are usually implicitly defined and locally defined relationships are usually explicitly defined.

2.1.8 Abstract Data Types

We have already discussed the most general Abstract Data Type (ADT), the Abstract Container. Through experience, however, it has become obvious that we are often storing objects that have very specific relationships between the objects and that we are often restricted to those queries or data manipulations that will be required.

For example, often we may need to store data that is implicitly linearly ordered. This is described by the Sorted List ADT. Similarly, like the queues in a bank or a grocery store, the description of a linear ordering where the first person into the queue is also the first person out, is described by a Queue ADT.

As we will see in Topic 6, if we are simply storing objects and are not storing any relationship, it is usually possible to store such objects in a hash table. All queries and relevant data manipulations (insertions and removals) on a hash table can be performed exceptionally quickly. If we need to store a relationship, this will usually require more complex structures, often require more time and more memory.

Thus, whenever you are intending to store, query, and manipulate data, it is necessary to consider questions such as:

1. What are the relationships on the objects?
2. Concerning the objects in the container:
    a. What queries may be made about them?
    b. What data manipulations may be made to them?
3. What operations may be performed on the container as a whole?
4. Concerning the relationships between the objects in the container:
    a. What queries may be made about the relationships?
    b. What manipulations may be performed on those relationships?

This course will describe various ADTs and then consider how these ADTs may be implemented using the C++ programming language using algorithms and data structures.