Using the ICS46 Template Library Classes

Introduction:

In this lecture we will discuss five standard data types: Stack, Queue, Priority Queue, Set, and Map. The code (each is a templated classes) for concrete implementations of these data type appears in the courselib you downloaded, in files named like array_queue.hpp. For now, we will focus on the information appearing at the top of these .hpp files: code that declares all the operations applicable to these data types. We will pay special attention to the public methods, operators, and constructors they declare, including the nested Iterator class and the methods and operators specified in it: they allow us to iterate over the information stored in any of these five data types.

By the end of this lecture, we should be able to understand how to write code that USES these templated classes, without having to understand how they are IMPLEMENTED. In fact, throughout the quarter we will see/write different implementations (using different data structures) for these data types, which all exhibit the same external/logical behavior, but whose use of resources (performance, e.g., time/space) varies.

In Friday's lecture, we will focus on how to implement these classes: define their methods, operators, and constructors using an actual data structure. We will study how to use a simple, low-level dynamic array data structure (which can grow) to implement all five data types: specifically we will examine the code for implementing

(a) the constructors/destructor for these templated classes
(b) the standard methods and operators for these templated classes
(c) iterator methods and operators for these templated classes

This code appears in the courselib you downloaded, in files named like array_set.hpp. We will briefly look at the complexity classes of these implementations and expand on this topic later in the quarter when we study more complicated/efficient data structures for these implementation

This sequence of lectures are followed by Programming Assignment #1, which asks you to represent and solve various problems using combinations of these data types. In that assignment, you will focus on understanding/exploiting these data types, using the simple but slow array implementations that I have provided. In Programming Assignments #2, #3, and #4, you will reimplement some of these data types using some of the more sophisticated and efficient data structures that we will study in this course. In Quiz #7 and Programming Assignment #5 (the last one), you will once again use these data types (and their more efficient implementations that you have written) to implement two new data types: Equivalence classes and Graphs.

After we learn more formally about using analysis of algorithms to study the resource use (performance) of different data structures implementing data types, we will have started to cover the three major topics in this course: data types, data structures implementing data types, and efficiency analysis. We will continue to explore their relationships throughout the rest of the course.

Please recognize and take some time/effort to understand the difference between the terms "data type" and "data structure". Getting a good intuitive undestanding of the difference is a major goal of this course, and it does take a bit of time to sort out their different meanings.

Five Data Types/Templated Classes:

In this lecture we will examine the five most important data types in three groups (based on the similarity of their operations) in the following order:
(1) Stack, Queue, Priority Queue
(2) Set
(3) Map (and pair: a simple class used implicitly and explictly with Maps)

After discussing ArrayQueue (as a representative of the first group) we will also discuss iterators for ArrayQueues in detail. Similar iterators are present for all of these data types, independent of what type of information they are iterating over. The behavior of these iterators must follow the same rules, regardless of what data structure we use to implement the data type. Sometimes the rules specify that the order of iteration is undetermined (sets and maps), so we have latitude to implement different orders based on what data structures we use for the implementation (so we can implement simple and efficient ones).

```
-->IMPORTANT:
-->You should have already downloaded the test_all_data_types project folder.
-->If you have not, follow the Sample Programs link from the course home-page
-->index to find it and download it.
-->
-->This project folder provides drivers and GoogleTests for the array
-->implementations (which you downloaded in the courselib folder) of all these
-->data type types. Taking a cue (not a Queue!) from programming Assignment #0,
-->you can easily switch your driver.cpp code to test drive any of the array
--> implementations, or switch to the GoogleTest for any of these data types.
-->
-->Using the drivers, you can experiment with the methods and operators for
-->these data types. You can also write small programs to test your under-
-->standing of their methods and operators. Typically it takes just a small
-->amount of code to do so.
-->
-->I have also provided a project folder (cross_reference) for a program (see
-->the end of this lecture note) that combines some of these data types to
-->solve a problem similar to those you need to write for Programming
-->Assignment #1. Again, follow the Sample Programs link from the course
-->home-page index to find it and download it.
-->
-->Finally, Quiz #1 will test this material in isolated functions, but similar
-->to the larger tasks that appear in Programming Assignment #1.
```

Stacks, Queues, and Priority Queues

The Stack, Queue, and Priority Queue data types are similar, because they are all data types that access their values in a predefined order. That is, when we remove values from these collections (by operations named "pop" and "dequeue"), the order of removal is determined by the data type of the collection. The orders are: Last-In-First-Out (LIFO, for a stack), First-In-First-Out (FIFO, for a queue), and Highest-Priority-First-Out (for a priority queue: we will use a special "gt" function that compares whether or not one value is "greater than" another to determine the ordering). We will often use a priority queue when we need to process (e.g., print) values of another data type in a sorted order: the Priority Queue does the "sorting" for us.

These templated classes are all similar. The biggest difference among them involves naming: adding/removing a value for a Stack uses the names push/pop, whereas the names for the other two classes are enqueue/dequeue. The names of many other "bookkeeping" operations are the same: e.g. size, empty, clear. So, for example, let's take a look at array_queue.hpp.

Notes:
1) All the classes appearing in courselib are put in the ics namespace, to contrast with the std namespace. Remember to use this namespace to qualify the names declared in it.

2) Other classes that implement these data types will have the same destructor, methods, and operations, and similar (or the same) constructors.

You can read the actual C++ code in these templated classes. I am changing the format of the ArrayQueue class a bit here, for simplicity of presentation: e.g., not showing "private" information, which would be different in every different implementation: private information relates to data structures.

```
namespace ics {


template<class T> class ArrayQueue {
  public:
    //Destructor/Constructors
    ~ArrayQueue();

    ArrayQueue          ();
    explicit ArrayQueue (int initialLength);
    ArrayQueue          (const ArrayQueue<T>& to_copy);
    explicit ArrayQueue (const std::initializer_list<T>& il);

    //Iterable class must support "for-each" loop: .begin()/.end()/.size() and prefix ++ on returned
result
    template <class Iterable>
    explicit ArrayQueue (const Iterable& i);


    //Queries
    bool empty      () const;
    int  size       () const;
    T&   peek       () const;
    std::string str () const; //supplies useful debugging information; contrast to operator <<


    //Commands
    int  enqueue (const T& element);
    T    dequeue ();
    void clear   ();

    //Iterable class must support "for-each" loop: .begin()/.end() and prefix ++ on returned result
    template <class Iterable>
    int enqueue_all (const Iterable& i);


    //Operators
    ArrayQueue<T>& operator = (const ArrayQueue<T>& rhs);
    bool operator == (const ArrayQueue<T>& rhs) const;
    bool operator != (const ArrayQueue<T>& rhs) const;

    template<class T2>
    friend std::ostream& operator << (std::ostream& outs, const ArrayQueue<T2>& q);

    Iterator begin () const;
    Iterator end   () const;

...

}
```

Note that this is a templated class; it will store values from the generic type T: e.g., enqueue takes an element of type T and dequeue returns an element of type T. The last constructor and the "enqueue_all" method are further templated by a type named Iterable, which can correctly match any class that implements for-each iteration (begin/end methods and the ++ operator): all five data types support these operations, so all can be arguments to Iterable constructors. For example, we can iterate through a queue and put all its values into a set. If we try to pass to Iterable some object that is defined in a class that doesn't implement these methods, the C++ compiler will indicate an error.

This class also includes/refers to its nested Iterator class (which is also templated by type T). For reference, this class appears as follows (for simplicity, it both declares and defines "operator <<" for Iterator objects).

```
class Iterator {
      public:
        //Private constructor called in begin/end, which are friends of ArrayQueue<T>
```

```
        ~Iterator();
        T          erase();
        std::string str  () const;
        ArrayQueue<T>::Iterator& operator ++ ();
        ArrayQueue<T>::Iterator  operator ++ (int);
        bool operator == (const ArrayQueue<T>::Iterator& rhs) const;
        bool operator != (const ArrayQueue<T>::Iterator& rhs) const;
        T& operator *  () const;
        T* operator -> () const;
        friend std::ostream& operator << (std::ostream& outs, const ArrayQueue<T>::Iterator& i) {
          outs << i.str(); //Use the same meaning as the debugging .str() method
          return outs;
        }
        friend Iterator ArrayQueue<T>::begin () const;
        friend Iterator ArrayQueue<T>::end   () const;

...

}
```

We will talk about how iterators are used after discussing the other methods in ArrayQueue. In fact, iterators are pretty much used identically in all five data types. So, once we know how iterators work in ArrayQueue, we will have a good model for how they work in the other four data types as well. Of course, you can always write/run small programs to test your understanding of these five data types and their iterators.

We will classify methods into two main categories: Commands (also known as "mutators") which change/mutate the state of the data structure implementing the objects they act on; and Queries (also known as "accessors") which examine, but do not change the state of the data structure implementing the objects they act on. Queries always return a result. Commands can be void (e.g., clear) or return some kind of result related to the command (e.g, enqueue, dequeue).

Let's examine each of the Queue methods, operators, and constructors/destructor more closely. To a large degree, the operations here have almost identical meanings for the Stack, Queue, and Priority Queue data types: the biggest difference is in how their removal methods ("pop" vs. "dequeue") work: see the descriptions above, where we characterize Stacks as LIFO, Queues as FIFO, and Priority Queues as HPFO.

Finally, the courselib includes two files named ics_exceptions (both a .hpp and .cpp file). These files declare/define common exceptions that are raised by methods defined in all implementations of these data types: e.g., ics::EmptyError, when we try to pop/dequeue a value from an empty stack/queue or priority queue.

Commands (for Queue: Stack and PriorityQueue are similar):

The "enqueue" method adds (includes) a value into the Queue. We don't need to understand here "how" this is accomplished by the data structure that implements a Queue, but only that it will be correctly accomplished, setting up for "dequeue" to remove the correct value.

  "enqueue" returns an int result specifying the number of values enqueued: for Queues this method always returns 1, because we can add duplicate values into Queues. Contrast this property with Sets, which allows NO DUPLICATES. For Sets, sometimes calling "insert" (the Set method for adding into a Set) will return 0 (the value to be added was already in the Set, so the Set remains unchanged) and sometimes it will return 1 (the value to be added was NOT already in the set, so the set changes to include that value).

The "dequeue" method removes (discards) the "next" value from the Queue, also returning that value: for queues, "next" is the least recently added value (for Stacks remove the most recently added value and for Priority Queues remove the value with the highest priority). The data structure implementing the enqueue and dequeue methods work together to accomplish this requirement, although when using queues in our program, we don't care HOW this external/logical behavior is accomplished, so long as it is accomplished CORRECTLY. Finally, note that the dequeue operation can fail, if there are no values in the Queue to remove (when the empty() query returns true; or the size() query returns 0): in such cases this method indicates its failure by throwing the ics::EmptyError exception.

The "clear" method removes all the values currently the Queue; its return type is void, so it returns nothing (instead, it could return an int specifying the number of values removed; but we will stay with void this quarter; we can call size -see below- before clear to compute this number). Calling the empty() and size() queries after calling "clear" will return a result of true and 0 respectively, regardless of the data structure implementing this data type: that relationship between methods is based on the data type itself, so all implementations must ensure that it is true.

For efficiency purposes, clear in an array with N values doesn't have to do the equivalent of N dequeues. In ArrayQueue, it just sets the used instance variable to 0, indicating there are no values in the array representing the queue. In a LinkedQueue, this operation might need to deallocate all the linked list nodes, taking much longer for large N; or maybe it will just set the head of the linked list to empty and save the linked list nodes for use when other values are enqueued; but what if there are going to be no more enqueues: that space would be wasted. As with many implementations, there are often interesting time/space tradeoffs.

The "enqueue_all" method is further templated by "Iterable", which can be instantiated by every data type that we define (and others too: technically the class must support at least a "begin", "end", prefix increment and dereference operators). It enqueues all the values produced by an Iterator for that data type. It also returns an int result specifying how many values were added to the Queue, which is the number of values the iterator produces. Note that "enqueue_all" may return 0, but only if the iterator parameter produce NO VALUES; if it produces even one value, that value will be added to the Queue so the returned result will be > 0.

We often use this method to copy all the values from one object into another object, often of different types (for the same type, a copy constructor will do the job more efficiently): if q is an ArrayQueue and s is an ArrayStack, and we want to copy successive values from the top of the stack into the queue, we can do so by writing q.enqueue_all(s). The stack remains unchanged because enqueue_all use the iterators that we will discuss in more detail soon to examine all the values in the stack.

Queries (for Queue: Stack and PriorityQueue are similar):

The "empty" method returns whether or not there are any values in the Queue. It is a convenient boolean method equivalent to testing whether size() == 0 (see below). Contrast empty (query) with clear (command).

The "size" method returns the number of values currently in the Queue. Often, but not always, a Queue implementation will store the size of the Queue as a counter. In these implementations, the enqueue method increments this counter by 1 and the dequeue method decrements this counter by 1 (if there is at least one value in the Queue so that dequeue does not throw an exception). So, it doesn't have to repeatedly scan the data structure and count all the values in it to compute the size (but a type can be implemented this way, which costs more time but saves a bit of space by not storing the counter). We use the term "caching" when we store/update a value rather than recomputing it from scratch. Caching is a standard time(faster) for space(uses more memory) tradeoff.

The "peek" method returns the same value as would calling the "dequeue" method (or throws the same exception) but DOES NOT REMOVE that value from the Queue. Recall that queries DO NOT change the state of the data structure implementing the object they act on. So calling "peek" a second time returns the same value as calling "peek" the first time (if no commands are called in between). Although we cannot directly peek at the second value in a queue, we can use iterators to get the equivalent information, but at a higher resource cost.

The "str" method is discussed below, along with the overloaded the << operator: they are closely related but there is an important distinction between them: fundamentally, all Queue implementations must produce identical results for "operator <<" (independent of how they are implemented) but they can -and often do- produce different results for the "str()" method; the difference includes information in "str()" that depends on the implementation used: for example, array and linked list implementations return different information for "str()". Such information is most useful when debugging different implementations of a data type.

Constructors (for Queue: Stack and PriorityQueue are similar):

Every class implementing a Queue will specify a destructor and at least four constructors. There should always be...

(1) A default constructor;
    The constructed object is empty.

(2) A copy constructor;
    The constructed object is a copy of it argument (which is the same type).
    In fact, we could substitute the default constructor followed by calling enqueue_all (which iterates over the argument queue) to achieve the same result, although for more interesting data structures and their implementations, there is often a faster way to "copy" the argument.

(3) An initializer_list constructor (new in C++11);
    The constructed object contains all the values in the initializer_list.

(4) An Iterable constructor, which iterates through any data type, enqueuing all the iterated-over values onto the constructed Queue.
    In fact, we could again substitute the default constructor followed by calling enqueue_all on the iterable to ahcieve the same effect.
    The constructed object contains all the values produced by the iterable.

Note that (3) and (4) are explicit. We can use them to EXPLICITLY construct ArrayQueues or convert initializer_lists and Iterables into ArrayQueues.

An example of (3: initializer_list) is

```
ics::ArrayQueue<int> small_primes({2,3,5,7,11});
```

An example of (4: iterable) is

```
ics::ArraySet<int> primes(small_primes);
```

Note that the Set primes now contains all the values in the Queue small_primes. If we wrote

```
ics::ArrayQueue<int> primes(small_primes);
```

then C++ would use (2: copy-constructor) to accomplish the construction.

Finally, becaue the Iterable constructor for ArraySet is explicit, we could not write the following (which C++ would flag as a complation error).

```
ics::ArraySet<int> primes = small_primes;
```

If the Iterable constructor were not explict, C++ could automatically use it to convert small_primes (iterable, as an ArrayQueue<int>) into an ArraySet<int>, and then perform the = operator. This two-step process would be less efficient.

A templated class may define more constructors, depending on the implementation, but it should always define these four. We will study all these constructors in the next lecture (and one more, special to array implementations), when we study one actual data structure (arrays) that implements these data types.

For Array implementations, there is an additional constructor that specifies the initial length of the array to allocate for storing the data type. In a default constructor, an array of length 0 or 1 is typically used (and the array size is increased when necessary). If we know the approximate number of values the data type will hold, specifying that number here can reduce the time taken to put all the values into the data type because its underlying array will not have to be copied when reallocated with a larger size.

Operators/Miscellaneous (for Queue: Stack and PriorityQueue are similar):

The = operator is overloaded for Queues. So if q1 and q2 are ArrayQueues storing the same type of information, then we can write the statement q1 = q2; now the information q1 originally stored is gone, and q1 contains all the information in q2 (for the == operator, now q1 == q2 should return true).

The == and != operators are overloaded for Queues. The definition for equality between Queues is that they must contain the same values in the same order (Stacks have the same definition; Priority Queues must store the same values and use the same "gt" (greater than function that specifies the ordering): meaning that these values would be dequeued from the PriorityQueues in the same order). Note that assignment (=) of priority queues store the right-hand sides's "gt" function into the (target) left-hand side, so afterward the priority queues will be ==.

The << operator is overloaded for Queues. So, if q is an ArrayQueue we can write std::cout << q << std::endl. We can also use << along with an ostringstream variable to build a string with the textual representation of a Queue inserted. Semantically, the << operator includes just the word "queue" with the queue values in square brackets ("[]"), separated by commas, and followed by the string ":rear", showing where the rear is (with the front implicitly at the other side). For a Queue with the two values "a" (first) and "b" (last), the string "queue[a,b]:rear" would be inserted on the stream.

Related to the overloading of << is the .str() query. It returns a string that includes a variant of the information that << inserts, followed in parentheses by any information that is relevant to class implementing the Queue. So if q is an ArrayQueue storing the example queue shown in the previous paragraph, calling q.str() would return a string like

```
  "ArrayQueue[0:a,1:b,2:,3:](length=4,front=0,rear=2,mod_count=2)"
```

in which private instance variables and their values appear in parentheses, again separated by commas. Likewise, if q is a LinearArrayQueue (see Programming Assignment #0) storing the same values, it would return a string like "LinearArrayQueue[0:a,1:b](length=2,used=2,mod_count=2)".

Regardless of which implementation we use, << inserts the same information for both these Queues: "queue[a,b]:rear".

  Note: the mod_count (modification count) instance variable allows us to implement FAIL-FAST iterators on a Queue that we are iterating over: iterators fail if we mutate the data that they are iterating over. This variable and these concepts are discussed in more detail below, when we discuss iterators.

An Introduction to Using Iterators (on Queues):

The "begin" and "end" methods respectively return an iterator representing a cursor/index TO (a) the first value in the Queue and (b) ONE BEYOND the last value in the Queue. One way to print all the values in an ArrayQueue storing std::string values is to iterate over all the values using the following "for" loop.

```
  for (ics::ArrayQueue<std::string>::Iterator i = q.begin(); i != q.end(); ++i)
    std::cout << *i << std::endl;
```

In this loop i is defined as an iterator initialized to index the first value (at the front of the Queue), incrementing (++i) until i indexes a value that is == to an iterator indexing ONE BEYOND the last value in the Queue. For each index i, *i (using the * operator) returns a reference to the value at that index in the Queue. Examine the Iterator class (shown above) to see that it overloads the ==, !=, ++ (both prefix and postfix), *, ->, and << operators, as well as declaring the "erase" and "str" methods.

The order that the values are iterated over is defined to be the order in which they would be dequeued from the Queue (and popped from a Stack, and dequeued from a Priority Queue, if those data types were iterated over).

In fact, if we compare the code above to the following code fragment

```
  while (!q.empty())
    std::cout << q.dequeue() << std::endl;
```

we will find that both PRINT THE SAME VALUES IN THE SAME ORDER. The difference is that in the first code fragment, q remains UNCHANGED (we iterate over the queue but it still contains all its original values) while in the second code fragment q is now EMPTY (which is the condition that terminates the "while" loop).

Can you explain why the code

```
  for (int i = 0; i < q.size(); ++i)           //Incorrect Code
    std::cout << q.dequeue() << std::endl;     //Incorrect Code
```

FAILS to print all the values in the Queue? Can you explain what it does print? Can you write a simlar for loop that works correctly? If you cannot, write a small program that executes this code and use the results you see to determine these answers.

As another example, we can use the following code fragment to iterate over a Queue of int values, to find the largest one, without modifying the Queue.

```
  int largest = numeric_limits<int>::min(); //All int values are >= this one

  for (ics::ArrayQueue<int>::Iterator i = q.begin(); i != q.end(); ++i)
    if (*i > largest)
      largest = *i;
```

In fact, there is a "for-each" loop in C++ that is even easier to use to iterate over a Queue. Here is a code fragment for the "for-each" loop that solves the same maximum problem, but much more simply.

```
  for (int v : q)
    if (v > largest)
      largest = v;
```

Here, the "for-each" loop implicitly iterate over all values in q, assigning to v each successive value in the Queue: C++ automatically defines an iterator, starting it at the beginning of the queue and going one beyond the end, and storing into v each value iterated over in the Queue: it is a shorter/simpler way to write the standard iterator loop: one also not requiring explicit indexing nor the use of the * operator. If we need to iterate over all the values in a Queue (or Stack or Priority Queue), this is the most elegant way to do so.

But wait, there's more! We can even use the "auto" feature to simplify this loop as follows (substituting auto for int).

```
  for (auto v : q)
    if (v > largest)
       largest = v;
```

That isn't much of an improvement, but we can use auto to simplify the original example, to become

```
  for (auto i = q.begin(); i != q.end(); ++i)
```

Later we will see examples of "for-each" loops iterating over maps, like

```
  for (const ics::pair<std::string,ics::ArraySet<std::string>>& kv : a_map)
    ...
```

which we can simplify using auto to

```
  for (const auto& kv : a_map)
    ...
```

Of course, with "auto" we don't explicitly see in our code the type of value that the "for-each" loop is iterating over. It is sometimes useful information to know (to see explicitly) that relates to how kv is used in the body of the loop. Using auto, C++ automatically deduces the type from the type of "a_map", and we should be able to do so too, but writing this type explicitly saves us the step of deducing it each time we examine the loop. Of course, we could also put the type in a comment to make it clear to anyone reading the code.

Iterating over a Queue allows us to examine every value without changing the Queue's contents....unless we want to. For one example, we can erase selected values. To do that we can call the "erase" method declared for Iterators. Here is a simple example. Suppose that we have a Queue of int and we want to remove all even values. We can do this task with the following code fragment

```
  for (ics::ArrayQueue<int>::Iterator i = q.begin(); i != q.end(); ++i)
    if (*i % 2 == 0)
      i.erase();
```

To call "erase" we need to declare/use an explict Iterator (to call it on; so, we cannot use the "for-each" style of loop for this task). When we call "erase", the Iterator's state becomes such that we cannot call "erase" again until after we increment the Iterator, to get to the next value to erase (done above by ++i in the last part of the for loop); For example, if we wrote

```
  ics::ArrayQueue<std::string>::Iterator i = q.begin();
  i.erase();  //erases first value
  i.erase();  //throws ics::CannotEraseError exception
```

the second call would throw the ics::CannotEraseError exception because the iterator is still referring to a previously erased value (which cannot be erased again). The following code fragment would correctly erase the first two values in the Queue, so long as q.size() is initially >= 2. So, we cannot erase the same value twice or somehow erase a value earlier than the one the current Iterator refers to (in the ICS46 Template Library we restrict iterators to moving forward; some data types in the C++ STL allow iteration both forwards and backwards).

```
  ics::ArrayQueue<std::string>::Iterator i = q.begin();
  i.erase();  //erases first value
  ++i;        //advances iterator to "second" value; could also execute i++;
  i.erase();  //erases second value
```

Calling "erase" also throws a CannotEraseError exception if the Iterator indexes data beyond the end of the queue: e.g., is == to an iterator ONE BEYOND the last value in the Queue. These rules are a bit complicated for beginners, but thely will become more intuitive, and we'll understand them better when we write code USING iterators and again when we have to write code IMPLEMENTING iterators for the more advanced data stuctures that we use to implement data types. In the next lecture we will study in detail how iterators work in the ArraySet class: how they index values in an array.

Note that the following code increments every value in the Queue by 1.

```
  for (int& v : q) //Note type int& for the index, not just int
    ++v;
```

It is equivalent to the loop

```
  for (ics::ArrayQueue<int>::Iterator i = q.begin(); i != q.end(); ++i)
    (*i)++;
```

Note that the * operator here returns a reference to a value in the Queue, not just the value itself, so the value at that reference can be mutated by ++.

For for-each loops that do not modify their argument, it is often more efficient to write them using const and &:

```
  for (const int& v : q)
    if (v > largest)
      largest = v;
```

Here, v references each of the values in q. This form saves copying each value in the queue into v. For ints the cost is not high (about the same as copying the int), but for more complicated values being iterated over, there can be substantial time savings using this form.

There is one more exception related to Iterators. If we are iterating through a Queue and we change the Queue (e.g., call the "enqueue" or "deqeueue" commands -which are mutator methods) in any way OTHER THAN THROUGH THE ERASE ON THE ITERATOR any subsequent use of that Iterator on the queue will throw a ConcurrentModificationError exception.

The basic idea here is that if we change a Queue while we are in the process of iterating over it (with one or more interators), then the meaning of continuing any iteration becomes unclear, so all started iterations refuse to work further. Iterators that do this are known as FAIL-FAST Iterators, as they fail quickly if their underlying Queue is changed (by either a command or performing an erase by another iterator).

That is, if we are iterating over a queue and changing the queue, how will the iteration be affected by enqueues and dequeue. The rules say it would be too hard to specify (and might have to depend on the data structure we are using, which would mean different implementations would produce different results), so it is eaiser to specify that the iterator cannot continue running on a mutated queue: instead it throws an exception.

Note that this problem does not apply to an Iterator itself changing the Queue by calling "erase": it should still be able to continue its iteration (but every other Iterator currently iterating through the Queue -there may be multiple Iterators active for the same Queue, must now fail). This situation is too advanced to cover in detail here, but we will pick up this discussion later in these notes (for Set) and much more in the next lecture when we look at how we implement a templated class with a data structure and how the data structure is iterated over and decides about throwing ConcurrentModificationError exceptions.

PriorityQueue: Special Template and Constructors

PriorityQueues are special, because each MUST be supplied with a "gt" function that determines the relative priority between any two values enqueued into a PriorityQueue. A call to gt(a,b) should return true, if a has a HIGHER priority than b: a is greater than b. Note that we DO NOT need to compute a priority for each value, only to compute whether the priority of one value exceeds another.

  Think about the difficulty in computing an integer priority of strings (with any number of characters): there are an infinite number of different strings, so we would need to compute an infinite number of different integers. But the int type is finite. On the other hand, given two strings, simple algorithms can compute whether or not one has a higher priority than the other (using the standard relational operators on strings). If you don't like the infinite argument, there are 52^N different strings of length N containing only letters; even for a relatively small N, that would exceed the number of int values.

First, each implementation of a queue will define (if no other queue implementation has been included in the code being compiled) a special undefinedgt function. This is done with the following macro.

```
#ifndef undefinedgtdefined
#define undefinedgtdefined
template<class T>
bool undefinedgt (const T& a, const T& b) {return false;}
#endif /* undefinedgtdefined */
```

This undefinedgt function is used as a default in the template/constructors as described below. Obviously we don't ever want to actually use this gt function (maybe it would be better to have it always throw an exception).

There are two opportunities to supply a "gt" function to a PriorityQueue.

1) As part of the template, when the type of the PriorityQueue is instantiated
2) During a call to a constructor

One of these opportunities MUST supply an explicit function pointer (not the "undefinedgt" function that is the default value in both the template and constructors): if NEITHER opportunity is taken, or if BOTH are taken AND the two function pointers are DIFFERENT, the constructor will raise a FunctionTemplateError exception. See below for some examples.

Once a PriorityQueue is constructed, its "gt" function generally cannot change, unless it appears on the (target) left-hand side in an assignment (=) statement, in which case the "gt" of the target becomes the "gt" function of the right-hand side (so the resulting priority queues will be ==).

The template for PriorityQueues looks like (note: tgt represents the TEMPLATE'S gt function and cgt represents the CONSTRUCTOR'S gt function):

```
//Instantiate the templated class supplying tgt(a,b): true, iff a has higher priority than b.
//If tgt is defaulted to undefinedgt in the template, then a constructor must supply cgt.
//If both tgt and cgt are supplied, then they must be the same (by ==) function.
//If neither is supplied, or both are supplied but different, TemplateFunctionError is raised.
//The (unique) non-undefinedgt value supplied by tgt/cgt is stored in the instance variable gt.
template<class T, bool (*tgt)(const T& a, const T& b) = undefinedgt<T>>
   class ArrayPriorityQueue {
```

The constructors look like:

```
ArrayPriorityQueue          (bool (*cgt)(const T& a, const T& b) = undefinedgt<T>);
explicit ArrayPriorityQueue (int initial_length, bool (*cgt)(const T& a, const T& b) =
undefinedgt<T>);
ArrayPriorityQueue          (const ArrayPriorityQueue<T,tgt>& to_copy, bool (*cgt)(const T& a, const
T& b) = undefinedgt<T>);
explicit ArrayPriorityQueue (const std::initializer_list<T>& il, bool (*cgt)(const T& a, const T& b)
= undefinedgt<T>);

template <class Iterable>
explicit ArrayPriorityQueue (const Iterable& i, bool (*cgt)(const T& a, const T& b) =
undefinedgt<T>);
```

For Array implementations, besides the four standard constructors, there is another one that specifies the initial length of the array storing the PriorityQueue. In a default constructor, an array of length 0 is used.

Let's assume that we are using an ArrayPriorityQueue to store ints, and we want the SMALLEST int to be dequeued FIRST. We can first define the "gt" function as

```
  bool gt(const int& a, const int& b) // a gt b (in priority) if a < b
  {return a < b;}                     // smaller values have higher priorities
```

Then, the following three statements produce equivalent ArrayPriorityQueues:

```
  ArrayPriorityQueue<int,gt> x;      //Specify "gt" as template argument
```

```
  ArrayPriorityQueue<int>    x(gt);  //Specify "gt" as constructor argument
  ArrayPriorityQueue<int,gt> x(gt);  //Specify same "gt" in template/constructor
```

All define an empty PriorityQueue whose "gt" function is the one declared above. There are different reasons to prefer different forms, which we will discuss during the quarter. One difference is that the second form can use a lambda for gt, so we could also write it more directly -not declaring the gt function explicitly- as

```
  ArrayPriorityQueue<int> x(
   (bool (*)(const int& a, const int& b))
   {[](const int& a, const int& b) {return a < b;}}
  );
```

Interlude:

The use of a lambda in a "default" constructor for ArrayPriorityQueue requires a special C++ construct: T{e} which tells C++ that e has type T.

The problem is that unlike named functions, lambdas DO NOT have any types in C++. When attempting to use a constructor and supplying just one argument, we must tell C++ to use the default constructor, with the argument specifying its cgt parameter, not the iterable constructor, with the argument specifying its i parameter and defaulting the cgt parameter to nullptr.

The rules for deciding which constructor to use are based on matching types, but lambdas have no types!

So, in the code above for T{e}, T is (bool (*)(const int& a, const int& b)) and e is [](const int& a, const int& b) {return a < b;}.

We do NOT need to use the construct T{e} if we are supplying two arguments to the constructors. For example, if s is an ArraySet<int>, whose values we want to put into a priority queue prioritized by a lambda, we can write just

```
  ArrayPriorityQueue<int> x(s,[](const int& a, const int& b) {return a < b;});
```

not needing the construction T{e} to specify the type of the typeless lambda. C++ will still correctly choose the correct/matching Iterable constructor. Likewise if the first argument is an ArrayPriorityQueue, C++ will use the copy constructor); if it is an initializer list (it will use the initializer list constructor).

Note that we CANNOT use a lambda when specifying the template argument; in the template we must use the name of an explicitly defined function. This is related to the fact that lambdas have no types.

Finally, in

```
  ArrayPriorityQueue<int,gt> x;
  ArrayPriorityQueue<int>    y(gt);
  ArrayPriorityQueue<int,gt> z(gt);
```

the TYPES of x and y are DIFFERENT; the TYPES of x and z are the same. The type is related to how the template is instantiated: the type of x has tgt = gt and the type of y has tgt = undefinedgt<T>. So we cannot write x = y; because there would be a type mismatch. We could use the explicit iterable constructor to translate between them: writing this line instead as

```
x = ArrayPriorityQueue<int,gt>(y);
```

Note that the type of x and z are the same, because the template is instantiated the same way (with tgt = gt); for z, only that same gt function can be used in the constructor. So we could write x = z; as well as z = x;

In fact, if we had a gt_reverse function

```
  bool gt(const int& a, const int& b) // a gt b (in priority) if a > b
  {return a > b;}
```

```
 we could still legally write

  ics::ArrayPriorityQueue<int,gt> x;
  ics::ArrayPriorityQueue<int>    y(gt_reverse);
  x = ics::ArrayPriorityQueue<int,gt>(y);

The last line uses the explicit iterable constructor to create a priority queue
that is organized by gt and contains y's values (which are organized in y by
gt_reverse), so then the assignment statement is legal, although x's gt is not
the the same as y's (y's is gt_reverse). If we wanted all of y's value to be in
x organized by x's gt, we could also write

  x.clear();
  x.enqueue_all(ics::ArrayPriorityQueue<int,gt>(y));

Here we need to use the iterable constructor in the argument.

Trying

  x = ics::ArrayPriorityQueue<int,gt>(y,gt_reverse);

would fail, because the iterable constructor has two different "gt" functions
specified, which is not legal (and will throw a TemplateFunctionError).

-----
Interlude:
  So why have "gt" specified in the template argument at all? The main reason is
  if that type were to be used as the type for a value associated with a key in
  a Map (more details about Maps, read below), we would need to specify the "gt"
  function as part of the template, to be part of its type.

  If we wanted to define an ArrayMap whose keys are std::string and whose
  associated values are ArrayPriorityQueues of ints (so that the smallest int
  value has the highest priority), we would write it as

    bool gt(const int& a, const int& b)
    {returns a < b;}

    typedef ArrayPriorityQueue<int,gt>  intPQ;
    typedef ArrayMap<std::string,intPQ> Map;

    Map m;

  Now, all the ArrayPriorityQueues constructed implicitly for Map values, will
  use this "gt" function specified in the intPQ template. It is part of the
  intPQ type itself, not something that must be supplied when each
  ArrayPriorityQueue is constructed: sometimes the construction is done
  implicitly, as here because it is the value type in the ArrayMap, where we
  have no further control over its "gt" function.

  We will also see this kind of duplication again, when we cover data types
  implemented by the binary search trees and hash table data structures
  (which require a "lt" and "hash" function respectively). And, some programming
  assignments at the end of the quarter will require that we specify a function
  as a template argument (I will remind you to reread this section then).
-----

One more example:

Suppose that we have a std::string x[] = ... storing x_length values and we
want to sort this array of strings, alphabetically. We could use the following
"gt" function and ArrayPriorityQueue.

  bool gt_alphabetic (const std::string& a, cont std::string& b)
  {return a < b;}

Generally, this function must return true when its first argument has a higher
priority than its second argument. Here, it returns true if a comes before b
```

alphabetically, using the standard < on std:string.

To alphabetize the words in array x, we could execute the following code. It (a) puts all the values into the ArrayPriorityQueue and then (b) removes them in alphabetical order, storing them back into the original array.

```
  ics::ArrayPriorityQueue<std::string,gt_alphabetic> pq;

  for (int i=0; i<x_length; ++i)
    pq.enqueue(x[i]);

  for (int i=0; i<x_length; ++i)
    x[i] = pq.dequeue();
```

We could also specify this as

```
  ics::ArrayPriorityQueue<std::string> pq(gt_alphabetic);
```

In C++11, we can also use a lambda for this function, writing the constructor call as

```
  ics::ArrayPriorityQueue<std::string>
     pq(
      (bool (*)(const std::string& a, const std::string& b))
      {[](const std::string& a, const std::string& b){return a < b;}});
```

which uses an anonymous lambda instead of defining and passing a reference to the gt_alphabetic function.

Finally, we CANNOT write

```
  ics::ArrayPriorityQueue<std::string,[](const std::string& a, const std::string& b){returns a < 
b;}) pq;
```

because we CANNOT use lambdas to instantiate templates (but can pass lambdas as arguments to constructors).

Sets

Below is a slightly simplified version of the array_set.hpp file (not including the nested Iterator class, which is the same as the one shown in ArrayQueue, except for the substitution of ArraySet<T> for ArrayQueue<T>). In many ways it is similar to the array_queue.hpp file. Again, below I show only the "public" parts.

Like a Stack, Queue, and Priority Queue, we can add (here "insert") values into a Set and remove (here "erase") specific values from a Set. The primary characteristic of a Set is that it does not contain duplicates (see the "insert" method below for details). Also notice that it declares many relational operators, because there are many different ways to compare sets mathematically (using the proper/normal subset/superset relationships), not just for equality and inequality.

```
namespace ics {


template<class T> class ArraySet {
  public:
    //Destructor/Constructors
    ~ArraySet();

    ArraySet();
    explicit ArraySet(int initialLength);
    ArraySet         (const ArraySet<T>& to_copy);
    ArraySet         (const std::initializer_list<T>& il);

    //Iterable class must support "for-each" loop: .begin()/.end() and prefix ++ on returned result
```

```
    template <class Iterable>
    ArraySet (const Iterable& i);


    //Queries
    bool empty        () const;
    int  size         () const;
    bool contains     (const T& element) const;
    std::string str () const; //supplies useful debugging information; contrast to operator <<

    //Iterable class must support "for-each" loop: .begin()/.end() and prefix ++ on returned result
    template <class Iterable>
    bool contains_all (const Iterable& i) const;


    //Commands
    int  insert (const T& element);
    int  erase  (const T& element);
    void clear  ();

    //Iterable class must support "for" loop: .begin()/.end() and prefix ++ on returned result

    template <class Iterable>
    int insert_all(const Iterable& i);

    template <class Iterable>
    int erase_all(const Iterable& i);

    template<class Iterable>
    int retain_all(const Iterable& i);


    //Operators
    ArraySet<T>& operator = (const ArraySet<T>& rhs);
    bool operator == (const ArraySet<T>& rhs) const;
    bool operator != (const ArraySet<T>& rhs) const;
    bool operator <= (const ArraySet<T>& rhs) const;
    bool operator <  (const ArraySet<T>& rhs) const;
    bool operator >= (const ArraySet<T>& rhs) const;
    bool operator >  (const ArraySet<T>& rhs) const;

    template<class T2>
    friend std::ostream& operator << (std::ostream& outs, const ArraySet<T2>& s);

    Iterator begin () const;
    Iterator end   () const;

...

}
```

Commands (for Set):

The "insert" method adds (includes) a value into the Set. We don't need to understand here "how" this is accomplished by the data structures that implement a Set, but only that it will be correctly accomplished, setting up for the other operations to work correctly on Sets.

"insert" returns an int result specifying the number of values inserted: for a Set, it will return 0 if the value to be added was already in the Set, so in this case the Set remains unchanged; it will return 1 if the value to be added was NOT already in the set, so the set changes to include that value.

The "erase" method removes (discards) the specified value from the Set and returns the number of values it removed: the result will be 0 if the specified value is not in the Set and 1 if it is in the Set. Unlike removal from a Stack, Queue, or Priority Queue, this methods do not throw an exception (trying to erase an absent value just returns 0); also unlike "pop" and "dequeue", we must specify which value to "erase".

The "clear" method removes all the values currently the Set; its return type is void, so it returns nothing (instead, it could return an int specifying the number of values removed; but we will stay with void this quarter; we can call size -see below- before clear to compute this number). Calling the empty() and size() queries after calling "clear" will return a result of true and 0 respectively, regardless of the data structure implementing this data type: that relationship between methods is based on the data type itself, so all implementations must ensure that it is true.

The "insert_all" method is futher templated by "Iterable", which can be instantiated by every data type that we define (and others too: technically the class must support only a "begin", "end", and prefix increment and dereference operators). It inserts all the values produced by an Iterator for that data type. It also returns an int result specifying how many values were added to the Set. Note that "insert_all" may return 0, either if the iterator produce NO VALUES or it produces only values that are ALL ALREADY IN THE Set. The maximum result it can return is the number of values the iterator produces, where each value produced is not already in the Set.

We often use this method to copy all the values from one object into another object, often of different types (for the same type, a copy constructor will do the job more efficiently): if s is an ArraySet and q is an ArrayQueue, and we want to compute whether all the values in the Queue are UNIQUE, we can do so by writing s.insert_all(q) and checking whether its returned result has the same size as the Queue. We can also use the "Iterable" constructor for ArraySet both to define and initialize such a Set: e.g., ArraySet<int> s(q);

Note that we could design insert_all to return the object it was called on instead of the number of new values it had. Then we could test for unique queue values by writing s.insert_all(q).size() == q.size().

The "erase_all" method is futher templated by "Iterable", which can be instantiated by every data type that we define (and others too: technically the class must support only a "begin", "end", and prefix increment and derefence operators). It erases all the values produced by an Iterator for that data type. It also returns an int result specifying how many values were removed from the Set. Note that this "erase" may return 0, either if the iterator produces NO VALUES or it produces only values that are ALL NOT IN THE Set. The maximum result it can return is the size of the Set, if the iterator produces every value in the Set.

The "retain_all" method is futher templated by "Iterable", which can be instantiated by every data type that we define (and others too: technically the class must support only a "begin", "end", and prefix increment and derefernce operators). It erases every value in the Set that is NOT one of the values produced by the iterator for that data type: it retains only those values, which is equivalent to "intersecting" the set with the iterable. It also returns an int result specifying how many values were removed (not retained) from the Set. Note that "retain_all" may return 0 if the iterator produces ALL THE VALUES in the Set. It can return the initial size of the Set, if the iterator parameter produce NO VALUES that are in the Set (this includes an iterator that produces no values). Note that this method doesn't return the number of values retained in the Set, because we can call the "size" method (see below) after "retain_all" to compute that value easily.

Here is a an example: if a Set of int contains the value 1, 2, 3, 4, and 5, and we retain the values 1, 3, 5, and 7 (produced by an iterator), then the resulting Set contains the values 1, 3, 5 and "retain" returns 2, because it removes 2 values; it does not return the number of values retained, 3, which is just the size of the remaining Set.

Queries (for Set):

The "empty" method returns whether or not there are any values in the Set.

It is a convenient boolean method equivalent to testing whether size() == 0 (see below).

The "size" method returns the number of values currently in the Set. Often, but not always, a Set implementation will store the size of the Set as a counter. In these implementations the insert method increments this counter by 1 if the value to insert is not in the Set and the erase method decrements this counter by 1 if the value to erase is in the Set. So, it doesn't have to repeatedly count all the values in the data structure to compute the size (but it can be implemented this way, which costs time but saves space). We use the term "caching" when we store/update a value rather than recomputing it from scratch.

The "contains" method returns whether or not the specified value is stored in the Set.

The "str" method is discussed below, along with the overloaded the << operator: they are closely related but there is an important distinction between them: fundamentally, all Set implementations must produce identical-looking results for "operator <<" (identical-looking, because the values in the Set might appear in any order) but they can (and often do) produce different results for the "str()" method; the difference includes information in "str()" that depends on the implementations. For example, array and linked list implementations return different information for "str()". Such information can be useful when debugging different implementations of a data type.

The "contains_all" method is futher templated by "Iterable", which can be instantiated by every data type that we define (and others too: technically the class must support only a "begin", "end", and prefix increment operator). It returns whether or not ALL the values produced when its start parameter iterates to its end parameter are contained in the Set. If the iterator parameter produces NO VALUES, this function returns true, because it produces NO VALUES that are NOT in the Set.

Operators/Miscellaneous (for Set):

The = operator is overloaded for Set. So if s1 and s2 are ArraySets storing the same type of information, then we can write the statement s1 = s2; now the information s1 originally stored is gone, and s1 contains all the information in s2 (for the == operator, now s1 == s2 should return true).

The == and != operators are overloaded for Sets. The definition for equality between Sets is that they must contain the same values; recall with Sets there really is no mention of in which order the values are. If << for s1 is "set[1,2]" and << for s2 is "set[2,1]" then s1 == s2 returns true because these sets store the same values. Note that s1 == s1 is always true.

The <, <=, >, and >= operators are also overloaded for Sets (but no other data type that we will study). These specify subsets and supersets. Here are the semantics of these operators.

```
  s1 <= s2 if every value is s1 is in s2 (the sets may be == ): subset
  s1 <  s2 if every value is s1 is in s2 (the sets must be !=): proper subset
  s1 >  s2 is the same as s2 < s1                             : proper superset
  s1 >= s2 is the same as s2 <= s1                            : superset
```

As with a Queue, a Set both overloads the << operator and defines a .str() method. A Set storing the values 1 and 2 will insert "set[1,2]" on a stream; but be careful here: the values in a Set have no special order, so this Set might insert "set[2,1]" just as easily. This "unordered" feature is even more important when we learn about iterating over all the values in a Set. If s is an ArraySet storing these values, the .str() method would return a string like "set[1,2](length=2,used=2,mod_count=2)" in which instance variables and their values appear in parentheses, separated by commas.

The "begin" and "end" methods return an iterator representing a cursor/index TO the "first" value in the Set and ONE BEYOND the "last" value in the Set respectively (for a Set of size() == 0, it returns the same iterator for both). As we discussed there really is no order among the values in Sets, but when we iterate over Sets they must produce these values in some order: not only can we not predict the order for different implementations, but different times that we iterate over a Set its values can be produced in a different order! DO NOT MAKE ANY ASSUMPTIONS ABOUT THE ORDER VALUES ARE PRODUCED BY Set ITERATORS: order is NOT an behavioral/logical property of a Set, the way it is for Stacks, Queues, and PriorityQueues.

Constructors (for Set)

Every class implementing a Set will specify a destructor and at least four constructors. There should always be...

(1) A default constructor;
    The constructed object is empty.

(2) A copy constructor;
    The constructed object is a copy of it argument.

(3) An initializer_list constructor (new in C++11);
    The constructed object has all the values in the initializer_list.

(4) An Iterable constructor, which iterates through any data type,
      enqueuing all the iterated over values onto the constructed Set.
      It is similar in form and function to the "enqueue_all" method.
    The constructed object has all the values produced by the iterable.

Note that (3) and (4) are explicit. We can use them to EXPLICITLY construct ArraySets or convert initializer_lists and Iterables into ArraySets.

An example of (3) is

```
  ics::ArraySet<int> primes({2,3,5,7,11});
```

A templated class can define more constructors depending on the implementation but it should always have these four. We will study all these constructors in the next lecture, when we study actual data structures that implement these data type.

For Array implementations, there is a constructor that specifies the initial length of the array storing the Set. In a default constructor, an array of length 0 is used.

Using Iterators (on Sets):

Everything that we know about Queue iterators applies to Set iterators. So I will not reproduce all that coverage here. Note that the following code fragment uses a Set Iterator to remove from Set s all strings whose length exceeds 10.

```
  for (ics::ArraySet<std::string>::Iterator i = s.begin(); i != s.end(); ++i)
    if ((*i).length() > 10)
      i.erase();
```

In fact, we can use the -> to simplify this code a bit, rewriting it as

```
  for (ics::ArraySet<std::string>::Iterator i = s.begin(); i != s.end(); ++i)
    if (i->length() > 10)
      i.erase();
```

The following code is more elegant but it is WRONG. It FAILS to do this job; instead it throws a ConcurrentModificationError exception. Can you explain why? Carefully examine the difference between the calls to the erase method.

```
  for (auto v : s)           //WRONG CODE   for (const auto& v : s) is wrong too
    if (v.length() > 10)     //WRONG CODE
      s.erase(v);            //WRONG CODE   not erase called on s, not iterator
```

Next, let's examine some code that fills a Set with 5 different string values, gotten by prompting the user (see my ics46goody.hpp file for the prompt_string function and a few other very useful programming goodies). What is interesting about this example is that we can write more elegant code if we leverage off a good understanding of Sets and all their methods.

```
  ics::ArraySet<std::string> s;
  int count = 0;
  while (count < 5) {
    std::string attempt = ics::prompt_string("Enter a String");
    if (! s.contains(attempt) ) {
      s.insert(attempt);
      ++count;
    }
  }
```

First, notice that we don't need a local variable to count the number of values in the Set (it has a query method for that). So we can simplify this code by removing all references to count, to be

```
  ics::ArraySet<std::string> s;
  while (s.size() < 5) {
    std::string attempt = ics::prompt_string("Enter a String");
    if (! s.contains(attempt) )
      s.insert(attempt);
  }
```

Second, notice what it if we add a string that is already in the Set, the Set remains unchanged. So, we don't need to test whether it is already contained in Set s before adding it; in fact, for most implementations performing the test takes about the same amount of time as just performing the insert, so just doing just the insert takes 1/2 the time doing a check and then insert. Thus, we can further simplfy this code to be

```
  ics::ArraySet<std::string> s;
  while (s.size() < 5) {
    std::string attempt = ics::prompt_string("Enter a String");
    s.insert(attempt);
  }
```

Finally, we don't really need the variable "attempt" (now that its value is used in just one place), so we can simplify this code to be

```
  ics::ArraySet<std::string> s;
  while (s.size() <5)
    s.insert(ics::prompt_string("Enter a String"));
```

Here is another interesting equivalence.

```
  ics::ArraySet<std::string> s;
  int successful_erases = 0;
  ...
  std::string value = ics::prompt_string("Enter a String to Erase");
  if (s.contains(value)){
    s.erase(value);
    ++successful_erases;
  }
```

Notice that because erase returns an int (1 if it erased a value, 0 if it didn't), we can simplfy this code to be

```
  ics::ArraySet<std::string> s;
  int successful_erases = 0;
  ...
  std::string value = ics::prompt_string("Enter a String to Erase");
  successful_erases += s.erase(value);
```

Of course, because value is now used just once, this could even be reduced to

```
  successful_erases += s.erase( ics::prompt_string("Enter a String to Erase") );
```

but I think that doesn't really simplify things; the "value" name is useful, if technically redundant.

The more you think about and practice using the methods in the Set class (and others), the simpler and more elegant your code will become. When making decisions about whether/what a method should return, the designer should think about what will allow commonly written code to be simplified.

How can we write code to retrieve a random value from a Set of strings and erase that value from the set? We could have required this operation be part of the Set data type, in which case every implementation would have to implement it (but do so efficiently for its data structure). But we can implement it using the current features of the Set data type. We can use a random number generator and an Iterator as follows.

```
  ics::ArraySet<std::string> s;
  ...
  std::string chosen;
  int choose = random() % s.size();
  ics::ArraySet<std::string>::Iterator i = s.begin();
  for (int on = 0; on < choose; ++on)
    ++i;
  std::string chosen = *i;
  s.erase(chosen);  //could also erase using iterator: i.erase();
```

Can you explain how/why this works, say for a Set of 5 values (to be concrete)?

In fact, we could simplify this to work more quickly by just always returning and removing the FIRST value iterated over (since there is no special ordering for Sets, the first value is as good as a random one).

```
  ics::ArraySet<std::string> s;
  ...
  ics::ArraySet<std::string>::Iterator i = s.begin();
  std::string chosen = *i;
  s.erase(chosen);  //could also erase using iterator: i.erase();
```

So, we might need to better understand what we mean by "random".

Maps and Pairs

Maps are the most interesting and useful of the five data types. A map associates "keys' (of some type) with "values" (of some type, which can be the same or different than the key type). Often the key is a simple type (e.g., string) while the value is some more complicated data type (e.g., Set). Each key is "associated with"/"mapped to" one value at any time. Typically once we associate/map a value with a key, we will later use the key to retrieve/get its associated value (and possibly change the value: e.g., if the value is a Set, we may add to or remove something from that Set). We can also erase a key, ask whether a key or value is in a map, and iterate through all mappings (represented by a pair, consisting of a key and its associated value).

A Map in C++ is used like a dict in Python (really is is more like a defaultdict, because accessing a non-existant key automatically creates a new mapping from that key to a value created by the default constructor for the type of the value). Recall the discussion of default constructors for PriorityQueues: one reason why the "gt" function appears as part of the class template is to allow us to specify a default constructor in an ArrayPriorityQueue; the default constructor doesn't require specification of a gt function as an argument.

Below is a slightly simplified version of the array_map.hpp file. Note that it is doubly templated, with both a KEY (specifying the type of the keys in the Map), and a T (specifying the type of the values in the Map); both KEY and T are used when specifying some of the prototypes of the methods: e.g., put takes a key of type KEY and a value of type T; erase takes a key of type KEY and

```
returns a value of type T. Also examine
  typedef ics::pair<KEY,T> Entry;
which specifies that each Entry in a Map is an ics::pair of these two types.

-----------

Just a heads-up here, when discussing Map iterators, we will show how to print
all the keys and their associated values in a Map. For this example lets assume
that ArrayMap m has keys of type std::string and values of type
ics::ArraySet<std::string>.

  for (const ics::pair<std::string,ics::ArraySet<std::string>>& kv : m)
    std::cout << kv.first << "->" << kv.second << std::endl;

Note that iterating through a map means iterating over entries (key/value pairs,
where pair is a class defined in the ics namespace). Using auto we can simplify
this code to just

  for (const auto& kv : a_map)            //faster than for (auto kv : a_map)
    std::cout << kv.first << "->" << kv.second << std::endl;

We will discuss this code more detail after discussing Maps belows.

-----------

------------------------------------------------------------------------------

namespace ics {


template<class KEY,class T> class ArrayMap {
  public:
    typedef ics::pair<KEY,T> Entry;

    //Destructor/Constructors
    ~ArrayMap();

    ArrayMap();
    explicit ArrayMap(int initialLength);
    ArrayMap         (const ArrayMap<KEY,T>& to_copy);
    ArrayMap         (const std::initializer_list<Entry>& il);

    //Iterable class must support "for-each" loop: .begin()/.end() and prefix ++ on returned result
    template <class Iterable>
    ArrayMap (const Iterable& i);


    //Queries
    bool empty      () const;
    int  size       () const;
    bool has_key    (const KEY& key) const;
    bool has_value  (const T& value) const;
    std::string str () const; //supplies useful debugging information; contrast to operator <<


    //Commands
    T    put   (const KEY& key, const T& value);
    T    erase (const KEY& key);
    void clear ();

    //Iterable class must support "for-each" loop: .begin()/.end() and prefix ++ on returned result
    template <class Iterable>
    int put_all(const Iterable& i);


    //Operators

    T&       operator [] (const KEY&);
    const T& operator [] (const KEY&) const;
```

```
    ArrayMap<KEY,T>& operator = (const ArrayMap<KEY,T>& rhs);
    bool operator == (const ArrayMap<KEY,T>& rhs) const;
    bool operator != (const ArrayMap<KEY,T>& rhs) const;

    template<class KEY2,class T2>
    friend std::ostream& operator << (std::ostream& outs, const ArrayMap<KEY2,T2>& m);

    Iterator begin () const;
    Iterator end   () const;

...

}
```

Commands (for Map):

The "put" method maps a key to a value (adds a pair consisting of the key and the value into the Map). Within a Map, such a pair is called an "Entry". If that key was already in the Map, put returns the value that it PREVIOUSLY MAPPED TO; if it wasn't already in the Map, it returns the value it is NOW MAPPED TO (just the second parameter). This method is like "push", "enqueue", "insert" in the other templated classes, but it returns not an int, but the old value that the key mapped to (or the current value if the key is new to the Map). We will see that by overloading the [] operator there is another way to put a key/value pair (or Enry) into Map: but which doesn't return anything; each form has its appropriate uses.

The "erase" method removes (discards) the key and whatever value it maps to in the Map. It also returns the value that the key (now removed) was PREVIOUSLY MAPPED TO. If the key is not in the Map, "erase" throws the KeyError exception.

The "clear" method removes all the entries currently in the Map; its return type is void, so it returns nothing (instead, it could return an int specifying the number of values removed; but we will stay with void this quarter). Calling the empty() and size() queries after calling "clear" will return a result of true and 0 respectively, regardless of the data structure implementing this data type: that relationship between methods is based on the data type itself, so all implementations must ensure that it is true.

The "put_all" method is further templated by "Iterable", which can be instantiated by every data type that we define (and others too: technically the class must support only a "begin", "end", and prefix increment and dereference operators). It puts into the Map all the entries produced by an iterator for that data type (think of it as breaking each entry into its key and associated value part, and doing a "put" with that key and value). It also returns an int result specifying how many entries were put in the Map (regardless of whether they were in the Map before): so really, it returns the number of values that the iterator produces.

Queries (for Map)

The "empty" method returns whether or not there are any values in the Map. It is a convenient boolean method equivalent to testing whether size() == 0 (see below).

The "size" method returns the number of values currently in the Map. Often, but not always, a Map implementation will store the size of the Map as a counter. In these implementations the put method increments this counter by 1 if the key to put is not in the Map and the erase method decrements this counter by 1 if the key to erase is in the Map. So, it doesn't have to count the key/value pairs to compute the size (but it can be implemented this way, which costs time but saves space).

The "has_key" method returns whether or not the specified key is stored in the Map.

The "has_value" method returns whether or not the specified value is associated with some (one or more) keys in the Map.

When we learn more advanced data structures for implementing Maps, we will find that the "has_key" method will typically perform much more quickly than the "has_value" method (and looking up the value associated with a key will perform at the same quick speed). Maps are organized by their keys, e.g., in binary search trees or hash tables, to provide a performance advantage for lookup by key.

The "str" method is discussed below, along with the overloaded the << operator: they are closely related but there is an important distinction between them: fundamentally, all Set implementations must produce identical-looking results for "operator <<" (identical-looking, because the entries in the Map might appear in any order) but they can (and often do) produce different results for the "str()" method; the difference includes information in "str()" that depends on the implementations. For example, array and linked list implementations return different information for "str()". Such information can be useful when debugging different implementations of a data type.

Constructors (for Map)

Every class implementing a Map will specify a destructor and at least four constructors. There should always be...

(1) A default constructor;
    The constructed object is empty.

(2) A copy constructor;
    The constructed object is a copy of it argument.

(3) An initializer_list constructor (new in C++11);
    The constructed object has all the values in the initializer_list.

(4) An Iterable constructor, which iterates through any data type,
      enqueuing all the iterated over values onto the constructed Queue.
      It is similar in form and function to the "enqueue_all" method.
    Constructed object has all the values produced by the iterable

Note that (3) and (4) are explicit. We can use them to EXPLICITLY construct ArrayMaps or convert initializer_lists and Iterables into ArrayMaps.

An example of (3) is

```
typedef ics::pair<std::string,int> Entry;
ics::ArrayMap<std::string,int> small_numbers(
  {Entry("one",1), Entry("two",2), Entry("three",2)});
```

which would print as

```
map[one->1,two->2,three->2]
```

the order might be different for Maps not implemented by arrays; the order is not part of the Map data type (and not part of the Set data type either, but the order is part of the Qeueue, Stack, and PriorityQueue data types).

A templated class can define more constructors depending on the implementation but it should always have these four. We will study all these constructors in the next lecture, when we study actual data structures that implement these data type.

For Array implementations, there is a constructor that specifies the initial length of the array storing the Map. In a default constructor, an array of length 0 is used.

Operators/Miscellaneous (for Map):

The [] operator is overloaded, most importantly to allow us to retrieve the value associated with a key. So in Map m, if k is a value of type KEY we can write m[k] to retrieve the value; if the value is a Set, we could write the statement m[k].insert(...value...); to mutate the Set associated with the key. We DON'T NEED TO RE-PUT the new set in the Map: it is already there, associated with the key k, but now mutated to contain a new value. THINK HARD ABOUT WHAT I JUST WROTE. Reread it.

If key k does not exist in m, then it will be put in m, associated with whatever value is constructed by the default constructor for T (the second class templating a Map). In this case it will return a reference to the new (empty) value associated with k. It is similar to a defaultdict in Python.

Finally, if v is a type T value, writing m[k] = v; updates the Map m equivalently to m.put(k,v); except unlike "put", this assignment statement does not return a value (OK, technically x = y is an expression that does return a value: a reference to the x; so more accurately, it always returns the new value stored in m[k], which is v; one could write (m[k] = v).method call(...);)
For example, executing

```
  ArrayMap<std::string,std::string> m;
  (m["a"] = "b").append("c");
  std::cout << m << std::endl;
```

prints associates the key "a" with the value "b" but then mutates it to "bc" so ultimately prints

```
  map[a->bc];
```

The = operator is overloaded for Maps. So if m1 and m2 are ArrayMap storing the same type of information, then we can write the statement m1 = m2; now the information m1 originally stored is gone, and m1 contains all the information in m2 (for the == operator, now m1 == m2 should return true).

The == and != operators are overloaded for Maps. The definition for equality between Maps is that they must contain the same keys mapped to the same values (said another way, they must contain the same entries). The order in which such maps would print is irrelevant. Note that m == m is always true.

As with other data types, a Map both overloads the << operator and defines a .str() method. A Map storing two entries (Mapping "a" to 1 and "b" to 2) will insert "map[a->1,b->2]" on a stream; but be careful here: like Sets, a Map has no special order, so this Map might insert "map[b->2,a->1]" just as easily. This "unordered" feature is even more important when we learn about iterating over all the entries in a Map. If m is an ArrayMap storing these entries, the .str() method would return a string like
"ArrayMap[a->1,b->2](length=2,used=2,mod_count=2)" in which instance variables and their values appear in parentheses, separated by commas.

The "begin" and "end" methods return an iterator representing a cursor/index TO the "first" entry (of type ics::pair) in the Map and ONE BEYOND the "last" entry in the Map respectively (for a Map of size() == 0, it returns the same iterator for both). As we discussed there really is no order among the entries in Map, but when we iterate over them, these entries must produced in some order: not only can we not predict the order for different implementations, but different times that we iterate over a Map its entries can be produced in a different order! DO NOT MAKE ANY ASSUMPTIONS ABOUT THE ORDER ENTRIES ARE PRODUCED BY Map ITERATORS: order is not a behavioral/logical property of a Map (or Set), the way it is for Stacks, Queues, and PriorityQueues.

Note that we can refer to the public .first and .second instance variables of each ics::pair (which represents an entry). We refer to these instance variables directly, not through accessors.

Simple Uses of Maps

Let us assume for simplicity that we have declared the following typedefs, which we will frequently use then using the ITL (here they have very generic names, in a real program the names should be more specific).

```
  typdef std::string                 Key;
  typdef icc::ArraySet<std::string>  Value;
  typdef ics::pair<Key,Value>        Entry;
  typdef ics::ArrayMap<Key,Value>    Map;

  Map m;
```

We have seen that the following code prints all the key/value associations in a Map, one per line. It iterates over every entry (Key/Value pair, so I often generically use kv for my interation variables) printing it. If you have more specific information about the keys/values, use a more specific name.

```
  for (const Entry& kv : m)
    std::cout << kv.first << "->" << kv.second << std::endl;
```

Again, using auto we can simplify this code to the following: but now that we are using typedefs to name interesting types,the code above is explicit by not so complicated (so auto is not so useful in the case where we have an Entry typedef).

```
  for (const auto& kv : m)
    std::cout << kv.first << "->" << kv.second << std::endl;
```

Because iterators produce key/value entries in no special order, we often need to use the following, more complicated code to print all the key/value associations in a Map, IN ALPHABETICAL ORDER ACCORDING TO k, using the entry_gt function, which we can write as follows. Note that entry_gt(a,b) returns true if the key of a has a higher priority than the key of b, meaning the key of a comes alphabetically BEFORE the key of b.

```
  bool entry_gt (const Entry& a, const Entry& b)
  {return a.first < b.first;}

  ics::ArrayPriorityQueue<Entry> sorted(entry_gt);
  sorted.enqueue_all(m);
```

or rewrite the second part by using entry_gt to instantiate the template

```
  ics::ArrayPriorityQueue<Entry,entry_gt> sorted;
  sorted.enqueue_all(m);
```

Both pairs of statements define an ArrayPriorityQueue named sorted, which is filled with the entries from Map m; they will be removed from the priority queue in an ordered dictated by entry_gt.

Now, we can iterate over the priority queue, resulting in the Map's keys being printed in sorted order: the biggest entry (highest priority), according to entry_gt, is printed first.

```
  for (const Entry& kv : sorted)
    std::cout << kv.first << "->" << kv.second << std::endl;
```

In fact, we can use a special ArrayPriorityQueue constructor (Iterable) to define and fill in this priority queue using just one line of code.

```
  ics::ArrayPriorityQueue<Entry> sorted(m,entry_gt);
```

or

```
  ics::ArrayPriorityQueue<Entry,entry_gt> sorted(m);
```

Although each is a mouthful, both do the job of constructing the Priority Queue named sorted with the appropriate "gt" function, filling it with the entries from the Map named m.

Then, instead of iterating over the Map, we iterate over the Priority Queue
(still producing Entry/pairs), which prints the entries (key/value pairs) in a
sorted order. We will see below different ways to print Maps in sorted order.

In fact, we can write everything a just a for loop, whichi implicitly declares
and uses the priority queue that was named "sorted" above.

```
  for (const Entry& kv :  ics::ArrayPriorityQueue<Entry,entry_gt>(m) )
    std::cout << kv.first << "->" << kv.second << std::endl;
```

Let's next examine a few ways to put/update an association/mapping in the Map
described above. Suppose we have a std::string key k1 and want a std::string v1
to be a value in the Set that k1 maps to. There are two cases to consider:

(1) k1 is a key in the Map (maps to a Set) so we should add v1 to that Set

(2) k1 is not a key in the Map (maps to NO Set), so we should put in an
association/mapping from k to a new Set that contains only v1

This code, or some variant of it, appears in most programs whose most basic
data type is a Map (most of those in Programming Assignment #1).

Here is some code that directly implements this algorithm.

```
  if (m.has_key(k1))            //k1 a key in m with associated set?
    m[k1].insert(v1);           //Yes: add v1 to it associated set
  else{                         //No:
    Value mapped_values;        //  Create empty set (Value is set of string)
    mapped_values.insert(v1);   //  add v1 to that empty set
    m[k1] = mapped_values;      //  associate k1 with this new set, storing v1
  }
```

Interlude

We could also "reverse" the last two lines in the else as

```
    m[k1] = mapped_values;      //  associate k1 with an empty set
    m[k1].insert(v1);           //  add v1 to the empty set
```

We could even write them as a single line, based on what value = evaluates to

```
    (m[k1] = mapped_values).insert(v1);
```

Or even reduce all 3 lines to a single line

```
    (m[k1] = Value()).insert(v1);
```

Why won't the following work?

```
    m[k1] = Value().insert(v1);  //Not equivalent to the code above
```

Notice in this code that we must "search" the Map for a key twice: once for the
call to has_key(), and once for [], depending which if part is executed.

We can simplify this code a bit, by using an special constructor.

```
  if (m.has_key(k1))            //k1 a key in m with associated set?
    m[k1].insert(v1);           //Yes: add v1 to it associated set
  else{                         //No:
    Value mapped_values({v1});  //  Create set containing one value: v1
    m[k1] = mapped_values;      //  associate k1 with this new set storing v
  }
```

But given what [] does for a key not in the Map (putting the key in the Map
associated with a value from type T's default constructor), the following
simpler (almost trivial) code always "searches" the Map once, either inserting
into the Set it finds already associated with that key, or inserting into an
empty Set it creates with the default constructor when it finds the key is not

in the Map.

```
  m[k1].insert(v1)
```

Understand what you want Maps to do and write the simplest code possible (simple often means efficient too, but that depends on implementations). This one line is equivalent to the if statements above.

Mutation in PriorityQueues, Sets, and Maps:

DO NOT ATTEMPT TO MUTATE any VALUES in a PriorityQueue or Set, or any KEYS in a Map. It is perfectly OK and frequently useful (as shown above) to mutate the VALUEs associated with KEYs in a Map.

Some data structures (e.g., binary search tree and hash table) implement these data types by storing values/keys in locations that are based on their state. Changing the state of such a value/key will change the location it SHOULD BE STORED IN, causing the value/key to be stored in the wrong place and/or be lost (unfindable). These issues are not present in Array implementations, so we will revisit them later in the quarter. But since we don't know what implementations we might be using, we should not do these kinds of mutations ever.

So, for example, if you wanted to change value in a Set, you should first remove it, then mutate it, then insert it back into the Set.

```
  ics::ArraySet<std::string> s;
  std::string element = ...;

  s.remove(element);
  element.mutator();
  s.insert(element);
```

Likewise, if you wanted to change a KEY in a Map, you should first remove it, then mutate it, then put it back into the Set associated with its old value.

```
  ics::ArrayMap<std::string,ics::ArraySet<std::string>> m;
  std::string a_key = ...;

  ArraySet<std::string> a_value = m.erase(a_key);  //store current value of key
  a_key.mutator();
  m[a_key] = a_value;
```

We will discuss this issue further, and in greater detail, when we learn about binary search trees. In fact, the same problem occurs when using hashing, so we will discuss this issues more than once later in this quarter.

I have written but will not have time to discuss in depth a cross reference program. Download the project cross_reference from the web: see the Lectures or Weekly Schedule link for today; see if you can quickly create an Eclipse project for this program, which uses courselib. this program uses these data types (templated classes in the ITL: Stack, PriorityQueue, and Map) to solve the following problem. If you understand this code, you are fully ready to solve Programming Assignment #1, but not until.

> Read a file that contains lines of words separated by spaces. Produce a cross reference map containing each word (as a key) and the lines in the file that it appears on (as a value associated with a key: a Stack) such that even if a word appears multiple times on a line, it appears only once in the Stack. (Prove that the values from the bottom to top of the stack must get bigger). Words using different case are considered to be different: "Run" != "run". Then, print the map alphabetically by the words; finally, print the map from most frequently to least frequently occuring words (e.g., sortd by Stack size), such that all the words occurring with the same frequency are printed in alphabetical order.

```
Here is a sample (trivial) input file (notice punctuation was stripped):

See Dick
See Jane
See Spot
See Spot run
Run Spot run

And here are the results in the console of running the program.

Enter file name to analyze[text.txt]:

XRef alphabetically
  Dick --> stack[1]:top
  Jane --> stack[2]:top
  Run --> stack[5]:top
  See --> stack[1,2,3,4]:top
  Spot --> stack[3,4,5]:top
  run --> stack[4,5]:top

XRef by frequency
  See --> stack[1,2,3,4]:top
  Spot --> stack[3,4,5]:top
  run --> stack[4,5]:top
  Dick --> stack[1]:top
  Jane --> stack[2]:top
  Run --> stack[5]:top

Notice that Dick, Jane, and Run all appear once, so in the final print (by
frequency) they appear in alphabetical order.

Here is the program. I'll just briefly explain

1) The typedefs allow us to use simple names in the rest of the code.

2) Notice how I used the following functions from ics46goody (in the courselib):
split in read_xref and safe_open in main. You will find these functions useful
in all parts of Programming Assigment #1.

3) Note that the declaration

     XRefPQ sorted(xref,has_higher_priority);

   declares an XRefPQ filled with entries found by iterating over xref,
   prioritized by the has_higher_priority parameter (discussed below in 4). It
   is a shorthand for the equivalent code

     XRefPQ sorted(has_higher_priority);
     sorted.enqueue_all(xref);

   or even the more verbose

     XRefPQ sorted(has_higher_priority);
     for (const XRefEntry& elem : xref)
       sorted.enqueue(elem);

4) In print_xref, read

      bool (*has_higher_priority)(const XRefEntry& i,const XRefEntry& j))

   as a pointer to a function that determines whether i has a higher priority
   than j (meaning i would be dequeued/iterated-over before j. Generally, this
   function has two const XRefEntry reference parameters and returns a bool.
   Semantically, it returns whether i > j (i has a greater priority than j),
   which is used to prioritize values in the priority queue.  We need to pass
   it an argument that is the name of such a function, or a lambda (which is
   what appears in calls in main). One  example is

      [](const XRefEntry& i,const XRefEntry& j)
```

```
      {return (i.first == j.first ?
                 i.second.size() < j.second.size() : i.first < j.first);}

   It says (see XrefEntry below) if the strings are equal, i > j if its Stack
   size is bigger, otherwise i > j if this string comes early in a dictionary.

   Note that the priority queue local variable in print_xref is instantiated to
   be of type ics::ArrayPriorityQueue<XRefEntry>, whose template does not
   specify a gt function. We specify the gt function by the has_higher_priority
   function. Recall that it is ILLEGAL TO INSTANTIATE a template using such a
   lambds parameter: ics::ArrayPriorityQueue<XRefEntry,has_higher_priority> is
   ILLEGAL.

5) Re: 3) Learn the idiom for using a PriorityQueue to sort the pairs produced
   by a Map in order to print the Map in a sorted order (based on both its
   keys and values).

------------------------------------------------------------------------------

#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include "ics46goody.hpp"
#include "array_stack.hpp"
#include "array_priority_queue.hpp"
#include "array_map.hpp"


//Useful typedefs: meaningful names connected to specific implementations
//  that are used in this program; some of these can be changed to use
//  different implementations (e.g., not Array) when those become available,
//  to improve the performance of the program; we must change #includes too.
//Note this program uses ArrayStack, ArrayPriorityQueue, and ArrayMap
typedef ics::ArrayStack<int>                 LineStack;
typedef ics::pair<std::string,LineStack>     XRefEntry;
typedef ics::ArrayPriorityQueue<XRefEntry>   XRefPQ;    //Must supply gt at construction
typedef ics::ArrayMap<std::string,LineStack> XRef;


//Read an open file of words separated by spaces and return a cross
//  reference (Map) of each word associated with the lines on which it
//  appears; if a word appears multiple times on a line, just record the
//  line number once (this requirement makes stacks the best data type
//  to record line numbers).
XRef read_xref(std::ifstream& file) {
  XRef xref;

  std::string line;
  int line_number = 0;
  while (getline(file,line)) {
    line_number++;
    std::vector<std::string> words = ics::split(line," ");

    for (const std::string& word : words)
      if (!xref.has_key(word) || xref[word].peek() != line_number)
        xref[word].push(line_number);
  }

  file.close();
  return xref;
}


//Print message and all the entries in a cross reference in the order specified
//  by *has_higher_priority: i is printed before j, if has_higher_priority(i,j)
//  returns true.
void print_xref(std::string message,
                const XRef& xref,
```

```
                bool (*has_higher_priority)(const XRefEntry& i,const XRefEntry& j)) {
  std::cout << "\n" << message << std::endl;
  for (const XRefEntry& kv : XRefPQ(xref,has_higher_priority))
    std::cout << "  " << kv.first << " --> " << kv.second << std::endl;
}


////An improved print_xref, printing line numbers ascending, separated by commas.
////Print message and all the entries in a cross reference in the order specified
////  by *has_higher_priority: i is printed before j, if has_higher_priority(i,j)
////  returns true.
//void print_xref(std::string message, XRef& xref,
//                bool (*has_higher_priority)(const XRefEntry& i,const XRefEntry& j)) {
//  std::cout << "\n" << message << std::endl;
//  XRefPQ sorted(xref,has_higher_priority);
//  for (const XRefEntry& kv : sorted) {
//    std::cout << "  " << kv.first << " --> ";
//    LineStack lines;
//    for (const auto& v : kv.second)
//      lines.push(v);
//    int count = 0;
//    for (int line : lines)
//      std::cout << (count++ == 0 ? "  " : ", ") << line;
//    std::cout << std::endl;
//  }
//}


//Prompt the user for a file, create a cross reference of its contents, and print
//  the entries in the cross reference two ways: sorted alphabetically (increasing)
//  by words and sorted by frequency of word use (decreasing); see the two lambdas
//  used to specify the order in which to print the entries.
int main() {
  std::ifstream text_file;
  ics::safe_open(text_file,"Enter file name to analyze","text.txt");

  XRef xref = read_xref(text_file);

  print_xref("XRef alphabetically",xref,
             [](const XRefEntry& i,const XRefEntry& j){return (i.first == j.first ? i.second.size()
< j.second.size() : i.first < j.first);});
  print_xref("XRef by frequency",xref,
             [](const XRefEntry& i,const XRefEntry& j){return (i.second.size() == j.second.size() ?
i.first < j.first : i.second.size() > j.second.size());});

  return 0;
}




For the input above, and the commented out print_xref, it prints

XRef alphabetically
  Dick -->   1
  Jane -->   2
  Run -->   5
  See -->   1, 2, 3, 4
  Spot -->   3, 4, 5
  run -->   4, 5

XRef by frequency
  See -->   1, 2, 3, 4
  Spot -->   3, 4, 5
  run -->   4, 5
  Dick -->   1
  Jane -->   2
  Run -->   5
```

It is still a bit sloppy, because the numbers do not start in an aligned column; that could be fixed by finding iterating throug the map and finding the longest word that is a key and then printing each word in that amount of space.

Can you change this program to convert all strings to lower case? For this file it would produce the following (with the original print_xref).

```
XRef alphabetically
  dick --> stack[1]:top
  jane --> stack[2]:top
  run --> stack[4,5]:top
  see --> stack[1,2,3,4]:top
  spot --> stack[3,4,5]:top

XRef by frequency
  see --> stack[1,2,3,4]:top
  spot --> stack[3,4,5]:top
  run --> stack[4,5]:top
  dick --> stack[1]:top
  jane --> stack[2]:top
```