Application Programming Interface (API)

One of the goals of this course is to discuss ways to organize and structure code. The API is a mechanism that helps us organize the code. It is a group of function specifications that are meant to provide some service or services to programs, clients of the API.

Data encapsulation refers to hiding the data of the implementation of the API functions so well that the client can’t get at it by normal programming methods. The clients have to use the API functions to get their work done. Notice that the “client” refers to a piece of code here, not to a person. The client code is a class or a function that requires the use of functions or utilities accessed through the API:

Example: C string library functions: strcpy, strcmp. Some of you may have seen the actual code for these C library functions. If you have, you probably noticed that they don’t look at all like what you would expect if you were to implement these functions in a homework assignment, for example. As a matter of fact, they are hardly comprehensible. But that’s OK, since they are library functions. The client code that uses them should not know or care how they are implemented, as long as they do what they are supposed to do. In any case we are not supposed to have access to the actual implementation of these functions. We only call the code through the C string library, which is the API.

Notice: There are no classes in C, only structures, and all fields are public. But, encapsulation can still be achieved through pointers to structures, that does not allow direct access to the contents of the structure.

Example: The Flips.java example from section 1.2 in S&W. It flips a coin, and counts the number of heads/tails. It is a client code for Counter.java, which implements everything you need for a
counter. As a matter of fact, any getter or setter function is an API function, since it allows us access to class members which are supposed to be private, so direct access by other classes is impossible.

**Example – bank account:**

Construct an account:

```java
BankAcnt ba = new BankAcnt('JJ', 234, 100);
// Modify balance:
ba.withdraw(20); ba.deposit(10);
// Access balance:
ba.balance; // Incorrect!!
ba.getBalance(); // Correct!!
```

The API is the method specification. An API lies between two bodies of code, the client code and the class implementation. The Java `interface` keyword allows us to define interfaces (APIs) for clear implementation:

```java
public interface Account {
    // withdraw amt from this BA
    int withdraw(int amount);
    // deposit amt to this BA
    void deposit(int amount);
    // return balance for this BA
    int getBalance();
}
```

Any class that has balance going up and down can _implement_ this interface:

```java
public class BankAccount implements Account {
    // constructor - create a BA
    public BankAccount(String nm, int _id, int bal) {...}
    // Account API functions
    public int withdraw(int amt) {...}
    public void deposit(int amt) {...}
    public int getBalance() {...}
    // Fields - all private
    private int id;
    private String name;
    private int balance;

    // client code example:
    public class TestBankAccount {
        static void main(String [] args)
        {
            BankAccount ba = BankAccount('JJ', 234, 100);
            ba.withdraw(20);
            ba.deposit(10);
        }
    }
```
Advantages of Encapsulation:

- Clear statement of functionality in use – The API functions are what is intended/provided by the class.
- Partition of responsibility/code: Important if many programmers work on the same code (which is often the case).
- Can share General Purpose objects across many apps, which saves coding. At worst we have to make minor improvements later. Say we find a better way to implement strcmp. Client code that uses strcmp should not know or care about it, nor should it make any changes.
- Safety of contents – the private fields can be accessed by the class code only.
- Debugability. No mystery changes to data – we can set breakpoint at the functions that change data.

Collections in Java

A collection is a container of objects. There are many types of collections in Java, and you have already seen some of them in CS210. The main collections we will work with are List, Stacks, Queues, Sets, Maps and Priority Queues. (stacks and queues, priority queues, and to an extent lists and sets, were covered by the pre-requisites). There are other collections we will not discuss in this course, but you are welcome to look at the Java documentation.

S&W does not provide a comprehensive cover for the Java Collections, but I will give the necessary background, and you can always refer to the Java documentation. The Java Collection Interface lays down the foundation. Every collection implements the Collection interface to allow different features – A collection may be ordered or unordered, it may or may not allow duplicates, etc. The decision on what collection to use depends on the application, and it is very important to use the right one.

// Collection interface; the root of all 1.6 collections.
public interface Collection<AnyType> extends Iterable<AnyType>, java.io.Serializable
{
    int size(); // How many items are in this collection.
    boolean isEmpty(); // Is this collection empty?
    boolean contains( Object x ); // Is X in this collection?
    boolean add( AnyType x ); // Adds x to collection.
    boolean remove( Object x ); // Removes x from collection.
    void clear(); // Change collection size to zero.
    // Obtains an Iterator object to traverse collection.
    Iterator<AnyType> iterator( );
    // Obtains a primitive array view of the collection.
    Object [] toArray();
    // Obtains a primitive array view of the collection.
    <OtherType> OtherType[] toArray(OtherType [] arr );
}

Notes

- Note how all the elements for a Collection are Objects, since a type parameter can only take on Object types.
• The only non-Object types are int, double, char, etc., the primitive types.
• However, each of these has a corresponding “wrapper” Object type: Integer, Double, etc., and autoboxing makes it easy to use these collections. We will see examples later on.

A Simple Client Program Example  The following simple program reads strings from the input, one line at a time, and inserts them into a List. The main program gets the List and prints it out. We can see some examples of Collection functions.

```java
public class ReadStringsWithArrayList {
    // Read an unlimited number of String;
    public static ArrayList getStrings() {
        Scanner in = new Scanner(System.in);
        ArrayList<String> array = new ArrayList<String>();
        // create a collection
        String oneLine;
        System.out.println("Enter any number of strings, one per line; ");
        System.out.println("Terminate with empty line: ");
        try {
            while (in.hasNextLine())
                oneLine = in.nextLine();
            array.add(oneLine); // add to collection
        } catch (IOException e) {
            System.out.println("Unexpected IO Exception has shortened amount read");
        }
        System.out.println("Done reading");
        return array;
    }

    public static void main(String[] args) {
        ArrayList<String> array = getStrings();
        for (int i = 0; i < array.size(); i++)
            System.out.println(array.get(i));
    }
}
```

It is possible to use Iterator for the printing:

```java
Iterator<String> itr = array.iterator();
// any Collection has an iterator
while (itr.hasNext())
    System.out.println(itr.next());
```

Encapsulation of Collection Objects

• In Java, the collection object is fully encapsulated.
• We can’t see the collection itself.
• We are allowed to get references to the objects in the collection.
• Each of them should be individually encapsulated.

Iterators Traverse Collections

Iterators are the mechanism by which we traverse or travel through a collection. This is the basic iterator interface.

```java
// Iterator interface
public interface Iterator<AnyType> extends java.util.Iterator<AnyType>
{
    // Are there any items not iterated over
    boolean hasNext();
    // Obtains the next (as yet unseen) item in the collection
    AnyType next();
    // Remove the last item returned by next.
    // Can only be called once after next
    void remove();
}
```

You can think of an iterator as a pointer by which we access elements in a Collection, since the items themselves are encapsulated and we do not have direct access.

Collection Types: The List Interface

You worked with lists before. This is a short reminder. A List is an ordered sequence of elements: \( a_0, a_1, a_2, ..., a_{n-1} \).

```java
// List interface.
public interface List<AnyType> extends Collection<AnyType>
{
    // Returns the item at position idx
    AnyType get( int idx );

    // Changes the item at position idx.
    AnyType set( int idx, AnyType newVal );

    // Obtains a ListIterator object used to traverse the collection bi-directionally.
    ListIterator<AnyType> listIterator( int pos );
}
```

```java
// ListIterator interface for List interface.
public interface ListIterator<AnyType> extends Iterator<AnyType>
{
    // Tests if there are more items in the collection when iterating in reverse.
    boolean hasPrevious();

    // Obtains the previous item in the collection when traversing in reverse.
    AnyType previous();
}
```
AnyType previous( );

   // Remove the last item returned by next or previous.
   // Can only be called once after next or previous.
   void remove( );
}

The two most important classes that implement the List interface are LinkedList and ArrayList. There are more that we will not discuss in this course. They have different performance for large lists. Both have extra methods over and above the List interface.

Mental Model of a List:
Here is an example of a 4 member list. This is a linked list. You can think of an ArrayList as basically an array.

```
A0 A1 A2 A3 NULL
```

We can use get(0), ..., get(3) and access any particular object ref. We can set(0, b) and replace the object at 0 with b.

Questions:
- What happens if we set(4, b)?
- To grow the list we need to use add(Object x), but where does it go??

Adding new elements is fast because the LinkedList has a pointer to the end-of-list.

This is how the iterator works:
- A ListIterator starting from 0, has a next method that returns element 0 on first call, element 1 on second, etc.
- Should test with hasNext before doing a next.
- If hasNext returns false, the iterator is at end of list (EOL).
- It starts at beginning of list, so there are 5 different iterator states for 4 elements:

```
< A0 > < A1 > < A2 > < A3 >
```

↑ original iterator, before element 0

↑ after first next, returning A0 (just before element 1)

↑ after 2nd next, returning A1

↑ after 3rd next, returning A2

↑ after last element

So, you want to think of an iterator as sitting between two elements, so to speak:

- At each point in time, an iterator is positioned just after the element that would be returned by previous(), and just before the element that would be returned by next().
- A ListIterator can go both ways.
- When we talk about numerical position in a list, it’s normally about the position of an element, not directly the iterator.
It can be somewhat confusing, since we refer to an index of an element and the iterator is between two elements, but looking at the figures and practicing a bit should clear the confusion.

Just remember: With the listIterator(int pos) method, the pos determines an element position and the method returns an iterator positioned prior to that element, or at EOL if pos == size of list. listIterator(0) gives an iterator positioned before the very first element, etc. A special case is that extra position after all the elements: this is attained by using N as an arg, the number of elements in the list.

**Remember: an iterator has N+1 possible positions for N elements.**

**Question:** What happens if next returns A1, then another next returns A2, and then a previous is done – is A1 returned?

**Answer:** No! We've just gone past A2 one way, and now we go back across it again, so A2 gets returned again.

**List Client Code:** This example shows several ways in which an iterator can go up and down a list.

```java
class TestArrayList
{
    public static void main( String [ ] args )
    {
        ArrayList<Integer> lst = new ArrayList<Integer>( );
lst.add( 2 );
lst.add( 4 );
        ListIterator<Integer> itr1 = lst.listIterator( 0 );
        System.out.print( "Forward: ");
        while( itr1.hasNext( ) )
            System.out.print( itr1.next( ) + " ");
        System.out.println( );
        System.out.print( "Backward: ");
        while( itr1.hasPrevious( ) )
            System.out.print( itr1.previous( ) + " ");
        System.out.println( );
        System.out.print( "Backward: ");
        ListIterator<Integer> itr2 = lst.listIterator( lst.size( ) );
        while( itr2.hasPrevious( ) )
            System.out.print( itr2.previous( ) + " ");
        System.out.println( );
        System.out.print( "Forward: ");
        for( Integer x : lst )
            System.out.print( x + " ");
        System.out.println( );
    }
}
```

Here the ListIterator<Integer> goes all the way down the list to EOL, then back along the list, so the turn-around occurs at EOL. Another ListIterator<Integer> starts from “lst.size()”, which would be 4 for our list. This is an artificial element number denoting the EOL position of the iterator. Again, there are n+1 different iterator states for n elements, and these are numbered from 0 to n. This is an important point which we must be clear about when adding or removing objects.
Iterator Remove: An iterator, as we have seen before, is basically a pointer that allows you to retrieve objects from a Collection, but also to remove them. In order to be sure we’re removing the right element, it is important to understand how iterators actually work.

- An iterator sits between elements.
- When calling remove, which nearby element gets removed?
- The object removed is the last one returned by next or previous, and only one remove per movement-action is allowed.
- What happens if you next, remove, and then next again?
- You access the element just after the removed element. Because we’ve moved past the deleted element already, the iterator position is clear.
- If you next, remove, previous, you should get the previous-to-removed. And so on, using the model above.

Iterator Remove Example: We want to remove all objects from a list that are equal to given object x. Say our objects are of class EOrder. We iterate down the list and call remove(x) whenever we encounter x (this is linear).

Example: Remove Duplicates With Two Iterators Using two iterators to remove duplicates from a List doesn’t work! We try to use two iterators: o1 runs the outer loop, o2 runs the inner loop.

- Algorithm: scan LinkedList list with itr, from list.iterator()
- For each itr-position, with element o1, initialize a ListIterator (o2) at that position.
- Scan rest of list with o2, removing elements that equal o1.
- The exception is ConcurrentModificationException.

The problem is that the two iterators try to modify the same list concurrently. The problem starts only after the first modification, when the second iterator tries to access the modified list.

List<Order> list = new LinkedList<Order>(); // or ArrayList
// add some elements to the list: Foo, Bar, Bar, Bam
Iterator<Order> itr = list.iterator();
int position = 0;
System.out.println("about to do next() in outer loop... list is" +list);
Order o1 = itr.next(); // throws ConcurrentModificationException here, after first remove
System.out.println("working on " + o1);
ListIterator<Order> listItr = list.listIterator(position + 1);
while (listItr.hasNext()) {
    Order o2 = listItr.next();
    System.out.println("inner loop o2 =" + o2);
    if (o1.equals(o2)) {
        System.out.println("removing o2 =" + o2);
        listItr.remove();
    }
}
Output:
List with Foo, Bar, Bar, Bam:
about to do next() in outer loop... list is[Foo, Bar, Bar, Bam]
working on Foo
inner loop o2 =Bar
inner loop o2 =Bar
inner loop o2 =Bam
about to do next() in outer loop... list is[Foo, Bar, Bar, Bam]
working on Bar
inner loop o2 =Bar
removing o2 =Bar
inner loop o2 =Bam
about to do next() in outer loop... list is[Foo, Bar, Bam]
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
at java.util.AbstractList$Itr.next(Unknown Source)
at ListTest.main(ListTest.java:20)

Safely Removing Duplicates  We don’t need two iterators. Drop the outer iterator and just save its value into a variable. Use the inner loop to traverse the rest of the list with an iterator, removing all elements equal to that variable.
What to do with huge lists, when using get and/or remove in inner loop means \( O(n^2) \) or worse?

- In this case we can use HashSet \( h = \text{new HashSet(list)} \); Set means no dups, \( O(n) \). We can then put the result back in a list.
- Another way: toArray, then sort, then pick off unique values, \( O(n \log n) \)

Performance of LinkedList vs. ArrayList  So, what type of list should we use? The answer, as always, depends on the application.

- For ArrayList of size \( n \)
  - Get, set are very fast, \( O(1) \)
  - Append-type add is fast most of the time. If it involves array expansion, it is expensive, \( O(n) \).
  - Delete is expensive unless if it is at the end.
- For LinkedList of size \( n \)
  - Get, set depends on the index position
  - get(1) is done by two next’s down the list from the beginning of the list, and get(n-2) is done by two previous next’s from the end of the list
  - Most expensive is get(n/2)
  - Delete/add is easy once the right spot in the list is located. Remove in an iterator is \( O(1) \), but the larger task may involve \( O(n) \) next’s to get the iterator positioned.

So, I would say, as a rule of thumb – if your list requires mostly insertions and deletions but not many searches, use a LinkedList. If the size of the list is more or less static and you do many get’s by index – use an ArrayList.
**Stacks and Queues** A Stack is a specialized List where can only insert (push), retrieve (top), and delete (pop) elements at one end. Stacks and Queues were covered in CS210. We will not talk about them directly in CS310, but we will introduce Priority Queues later on.

![Stack Diagram](image)

A Queue is a specialized list where insert at one end, retrieve and delete at the other.

![Queue Diagram](image)

**Sets**

A set is a mathematical concept. Basically, it is a bunch of elements, with no duplicates and no order. A set can also be empty, or infinite in size. The elements don’t have to be numbers.

**Examples:** A = \{1, 5, 3, 96\}, or B = \{17, 5, 1, 96\}, C = \{“Mary”, “contrary”, “quite”\}. This is incorrect - \{“Mary”, “contrary”, “quite”, “Mary”\} (there is a duplicate).

In Java, the Set interface is the Collection interface. The API isn’t sensitive to the lack of duplicates, only the implementation. The implementations in the JDK that we will discuss here are the TreeSet and HashSet. They check for duplicates by using the equals method of the elements. Additionally, HashSet uses hashCode() and TreeSet uses compareTo(), to implement the specific types of sets.

**TreeSet Example:** A TreeSet keeps the elements sorted. Note that a pure set is supposed to be without order, and here we are seeing order imposed by the TreeSet. It’s an extra feature, so the TreeSet gives us a sorted Set. We can just ignore the order if we want. The TreeSet gives a high-performance implementation, almost competitive with HashSet.

```java
public static void main(String[] args) {
    Set<String> s = new TreeSet<String>();
    // For reverse order: TreeSet<String>(Collections.reverseOrder());
    s.add("joe");
    s.add("bob");
}
```
Sets of JDK Element Type

Sets of type String, Integer, etc. are very easy to use, JDK classes all have appropriate equals, hashCode, and compareTo. They implement Comparable\(<\text{E}>\), which is necessary for TreeSet.

Example: Simple set app using element type String.

```java
public static void main(String[] args) {
    Set<String> s = new HashSet<String>();
    s.add("joe");
    s.add("bob");
    s.add("hal");
    printCollection(s);
}
```

If we add “hal” again, there will be no difference in resulting Set.

Sets of User Defined Objects

If we use our own class for the element type, we have to make sure equals, and hashCode or compareTo are in good enough shape to work properly when called by HashSet or TreeSet on the element objects.

Consistency requirements:

- equals and hashCode must be consistent, so that if \(a.equals(b)\), then \(a.hashCode() == b.hashCode()\). The opposite is not necessarily true, as we will see when discussing Hash Tables later on.
-quals and compareTo must be consistent, so that if \(a.equals(b)\), then \(a.compareTo(b) == 0\). This does have to go in both directions.

Equals Example – Potentially Problematic: Here are two classes, a base and a derived class. Look at the equals function. As a reminder, this function defines what constitutes equality between two instances of a class. Without it, the objects are compared by reference.

```java
class BaseClass {
    public BaseClass(int i) { x = i; }
    public boolean equals(Object rhs) {
        if(rhs == null || ! rhs instanceof BaseClass )
            return false;
        return x == ((BaseClass) rhs).x;
    }
    private int x;
}

class DerivedClass extends BaseClass
```
This example is problematic, because the derived class is also an instance of the base class, and so the base class equals may return true if it gets an instance of the derived class with the same x. Unless that’s what you wanted all along, it is incorrect. If we use the following

```java
if (getClass() != rhs.getClass()) return false;
```

it will return false when comparing the base and derived class.

**Question:** is BaseClass in good shape to make HashSet<BaseClass>?

**Answer:** No, it overrides equals, but not hashCode. If it overrode neither, it would be OK, but once equals is overridden, you need to override hashCode too, and consistently. See above.

**Fix:** either drop equals(), or add:

```java
public int hashCode() {return Integer(x).hashCode();}
```

**Question:** is BaseClass in good shape to make “new TreeSet<BaseClass>()”?

**Answer:** No, it doesn’t implement Comparable<BaseClass>.

**Fix:** add “implements Comparable<BaseClass>”, and method compareTo:

```java
public int compareTo(BaseClass b) { return Integer(x).compareTo(b.x); }
```

**Student Example:** This is an example of a simple student class. The comparison is done by student id. I will call it the "unique identifier" throughout the course, since this is what distinguish one instance from another, as far as the code is concerned. Notice that two students with the same name but different id’s will be considered different students under this definition.

```java
class SimpleStudent implements Comparable<SimpleStudent> {
    public SimpleStudent(String n, int i) { name = n; id = i; }
    public boolean equals(Object rhs) {
        if (rhs == null || getClass() != rhs.getClass())
            return false;
        SimpleStudent other = (SimpleStudent) rhs;
        return id.equals(other.id);
    }
    public int compareTo(SimpleStudent other) {
        return id.compareTo(other.id);
    }
    public int hashCode() {
        return id.hashCode();
    }
}
```
equals/hashCode in User Defined Sets: As before, we can drop equals and hashCode from the class implementation. It will allow students with the same name and compare two objects by reference only. This way, two students with the same id, say (“Bob, 1”) and (“Alice, 1”), are different elements, so the set will contain two elements. If we use the equals that is coded above, the two objects are .equals each other, and the set will contain only one element even though they have two different names. Remember that compareTo should be consistent with equals, and hashCode should be the same for two .equals objects.

Usually, when you define your own class, you will have just one field which is the unique identifier, and it will usually be a built-in data type with its own equals, hash code and compareTo. In this case you’re all set, but sometimes you will need more than one field. For example (from section 1.2 of S&W):

```java
public final class PhoneNumber {
    private final int area, exch, ext;

    public PhoneNumber(int area, int exch, int ext) {
        this.area = area; this.exch = exch; this.ext = ext;
    }

    public boolean equals(Object other) {
        if (other == this) return true;
        if (other == null) return false;
        if (other.getClass() != this.getClass()) return false;
        PhoneNumber that = (PhoneNumber) other;
        return (this.area == that.area) && (this.exch == that.exch) && (this.ext == that.ext);
    }

    public int hashCode() {
        return 31 * (area + 31 * exch) + ext;
    }
}
```

The example uses all three parts of the phone number of hash code and equals.

Maps

Definition 1 Maps

- Given two Sets, Domain and Range, with a relation from one to another.
- Like a math function, each domain element has associated with it exactly one range element.
- Two arrows can land on the same range element, but one domain element cannot have two arrows out of it. See Figure 1.

Notice that a map is basically the same as a mathematical function from one set to another (the Set here is as in the mathematical definition, no repeats, no particular order). Remember from Calc 1 that a function from one set (domain) to another (range) should cover every element in the domain. That is – every domain element has a range element associated with it. The opposite is not necessarily true (unless it’s an onto function). Also – a range element may have several domain elements.
pointing to it (unless it’s a one-to-one function). The action of following the arrow is often known as a “lookup” action. For example, employee records are looked up by social-security no. and/or by employee name. Social security numbers or employee names are the Domain, Employee objects are the Range. In programming, Maps are lookup tables. We are mapping integers to employee objects or Strings to employee objects. Mapping creates a pair of \( <\text{DomainType},\text{RangeType}> \).

The DomainType is called a \textit{key}, the RangeType is a \textit{value}. To remember it as follows: The key is what you have, the domain is what you’re looking for when you “open” the particular entry with your key. When you look up an employee record by SSN, you should know the SSN, which is your key, to pull the entire record from the database.

A simple example: Converting grades from numbers to letters:

\[
\begin{align*}
92 & \rightarrow \text{’}A\text{’} \\
79 & \rightarrow \text{’}B\text{’} \\
68 & \rightarrow \text{’}C\text{’}
\end{align*}
\]

Here the DomainType is an integer and the RangeType is string (could be a character but we want to account for all letter grades like "A-" etc). Each of these lines can be called a “key/value pair”, or just “pair”. \((92, \text{’}A\text{’})\) is a pair of the grade 92 (the key) and the string "A" (the value) The whole mapping is the set of these 3 pairs.

\( M = \{ (92, \text{’}A\text{’}), (79, \text{’}B\text{’}), (68, \text{’}C\text{’}) \} \) – a map as a set of pairs, or “associations”

A mapping is a collection like other collections we are studying, lists, stack, queues, and sets. However, in Java a Map has its own interface separate from Collection, since the mapping makes it quite different from the other collections. Note that not every collection of pairs makes a proper map: \( M \) qualifies as a map only if the collection of keys has no duplicates, i.e., constitutes a Set. The collection of values can have repetitions, so it is not a Set, just a collection

The Map Interface

```java
public interface Map<KeyType,ValueType> extends Serializable {

```

Figure 1: This is an example of a Map. Notice that every element from the domain has an element from the range associated with it.
Returns the number of keys in this map.

```java
int size();
```

Tests if this map is empty.

```java
boolean isEmpty();
```

Tests if this map contains a given key.

```java
boolean containsKey(KeyType key);
```

Returns the value in the map associated with the key.

```java
ValueType get(KeyType key);
```

Adds the key value pair to the map, overriding the original value if the key
was already present.

```java
ValueType put(KeyType key, ValueType value);
```

Remove the key and its value from the map.

```java
ValueType remove(KeyType key);
```

Removes all key value pairs from the map.

```java
void clear();
```

Returns the keys in the map.

```java
Set<KeyType> keySet();
```

Returns the values in the map. There may be duplicates.

```java
Collection<ValueType> values();
```

Return a set of Map.Entry objects corresponding to
```
Set<Entry<KeyType,ValueType>> entrySet();
```
Maps vs. Sets  Say we had a Set of Student records and we wanted to look up a student (by name or student ID). We can look up a student record very fast, but there is no efficient way to pull out an entire record in a Set! We could iterate over the whole Set, and find the match, but that’s $O(N)$, and we want something faster. The Map interface allows us to do that.

Priority Queue

Often data elements are processed in a specific order, not necessarily the order in which they were inserted. Examples include:

- In OS process scheduling, a process comes with a priority, and high priority processes are executed earlier than low priority ones
- In network routing, high priority traffic (VoIP, IPTV) are delivered before other traffic

This means that we have to attach a number, a "priority" to each element, and we need to dynamically maintain the pending elements so that high priority elements will be processed before low priority ones, regardless of the order in which they were introduced. For example, if we are maintaining a server with a scheduling process:

- New jobs keep coming in
- When a server becomes available, the highest-priority job is removed from PQ and serviced

By Convention: the priority is a non-negative number, and a smaller value has a higher priority. Priority Queues support three operations:

1. Insert(Q, item, priority)
2. FindMinimum(Q)
3. DeleteMinimum(Q)

Implementation by different data structures, worst case runtime:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unsorted array</th>
<th>Sorted array</th>
<th>Linked list</th>
<th>Balanced tree</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>FindMin</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DeleteMin</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

The trick for arrays and lists: in addition to the data structure, keep a pointer that points to the current minimum. Priority queues are usually implemented using balanced trees or heaps. We will see many examples throughout the course.