What criterion should we use to choose an appropriate data structure for an application?

How about "I already understand data structure A, and I don't understand data structure B"? .... umm, no.

Perhaps "There is a built-in module for data structure A, but I would have to code data structure B myself"? .... fail!

Or “I can code A in 5 minutes, but B would take an hour” ... nope, that’s not a good reason.

What could be left? What could be right?

The answer is computational complexity. We will prefer structure A to structure B if A has a lower order of complexity for the operations we need in our particular application.

Before we discuss computational complexity, we need to agree on the type of computer we are dealing with. In particular, we need to clarify which operations can be completed in constant time.
The R.A.M. Model of Computation

We will use the Random Access Machine model, which can be summarized as follows:

- sequential operations only (i.e. no parallel computation)
- all fundamental operations:
  - +, -, *, / and comparisons for integers and floating point numbers
  - comparisons on booleans
  - comparisons and type conversions on characters
  - execution control
  - accessing a memory address
  - assigning a value to a variable
  
  take constant time
- no hardware accelerators (such as caches or virtual memory) are employed

It is important to note that this model implies an upper limit on the number of digits in any number. This is true of virtually all programming languages.

The RAM model does not assume constant time operations on strings. A string is considered to be a data structure consisting of a sequence of characters.

I'm assuming we are all familiar with "big O" complexity classification (since it has been covered in other courses, most notably CISC-203!):

Let $f(n)$ and $g(n)$ be non-negative valued functions on the set of non-negative numbers. If there are constants $c$ and $n_0$ such that $f(n) \leq c \times g(n)$ $\forall n \geq n_0$ then we say $f(n) \in O(g(n))$

The significance of this is that as $n$ gets large, the growth-rate of $f(n)$ is no greater than the growth-rate of $g(n)$. In other words, the growth of $g(n)$ is an upper bound on the growth of $f(n)$. 
There are several complexity classes that we encounter frequently. Here is a table listing the most common ones.

<table>
<thead>
<tr>
<th>Dominant Term</th>
<th>Big-O class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$ (a constant)</td>
<td>$O(1)$</td>
<td>constant time</td>
</tr>
<tr>
<td>$c \cdot \log n$</td>
<td>$O(\log n)$</td>
<td>logarithmic time</td>
</tr>
<tr>
<td>$c \cdot n$</td>
<td>$O(n)$</td>
<td>linear time</td>
</tr>
<tr>
<td>$c \cdot n \cdot \log n$</td>
<td>$O(n \cdot \log n)$</td>
<td>$n \log n$ time</td>
</tr>
<tr>
<td>$c \cdot n^2$</td>
<td>$O(n^2)$</td>
<td>quadratic time</td>
</tr>
<tr>
<td>$c \cdot n^3$</td>
<td>$O(n^3)$</td>
<td>cubic time</td>
</tr>
<tr>
<td>$c \cdot n^k$ Where k is a constant</td>
<td>$O(n^k)$</td>
<td>polynomial time</td>
</tr>
<tr>
<td>$c \cdot k^n$ Where k is a constant &gt; 1</td>
<td>$O(k^n)$</td>
<td>exponential time</td>
</tr>
<tr>
<td>$c \cdot n!$</td>
<td>$O(n!)$</td>
<td>factorial time</td>
</tr>
</tbody>
</table>

**Combinations of Functions**

If $f_1(n) \in O(g_1(n))$, and $f_2(n) \in O(g_2(n))$ ....

then $f_1(n) + f_2(n) \in O(max(g_1(n), g_2(n)))$

and $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$

So far this should all be very familiar. But O classification is just the small first step in the field of computational complexity. There are many other ways of grouping functions together based on the resources (time and/or space) they require. We will consider two more: **Omega** classification and **Theta** classification.
**Omega Classification**

Big O classification gives us an **upper bound** on the growth-rate of a function (that is, \( f(n) \in O(g(n)) \)) tells us that \( f(n) \) grows no faster than \( g(n) \) grows, but it doesn’t tell us anything about a **lower bound** on the growth-rate of \( f(n) \).

Your first reaction to this observation might well be "why would we care about a lower bound on the growth-rate? We use this computational complexity stuff to measure the worst-case running time of an algorithm ... and for worst-case analysis, all we need is an upper bound."

Before we explain why lower-bound analysis is important, we will define exactly what we mean by it and how it works.

**Definition:** Let \( f(n) \) and \( g(n) \) be functions. If there exist constants \( c \) and \( n_0 \) with \( c > 0 \) such that

\[
f(n) \geq c \cdot g(n) \quad \forall n \geq n_0
\]

then \( f(n) \in \Omega(g(n)) \) (\( \Omega \) is the Greek letter “Omega”)

Note that this is almost exactly the same as the definition of Big O except that the "\( \leq c \cdot g(n) \)" has become "\( \geq c \cdot g(n) \)".

As with Big O classification, we can see that \( \Omega(g(n)) \) is actually a class of functions, all of which grow at least as fast as \( g(n) \) grows. We can also see that there is a hierarchy of Omega classes, just as there is a hierarchy of Big O classes. For example, suppose \( f(n) \in \Omega(n^3) \). This means "growth-rate of \( f(n) \)" \( \geq "growth-rate of \( n^3 \)". But since "growth-rate of \( n^3 \)" \( \geq "growth rate of \( n^2 \), we can conclude that "growth rate of \( f(n) \)" \( \geq "growth rate of \( n^2 \), which is equivalent to saying that \( f(n) \in \Omega(n^2) \).

In fact, if \( f(n) \in \Omega(n^k) \), then \( f(n) \in \Omega(n^i) \quad \forall i < k \).

(Note the parallel to Big O: if \( f(n) \in O(n^k) \), then \( f(n) \in O(n^i) \quad \forall i > k \))

When determining the Big O classification for \( f(n) \) we try to find the **smallest** function \( g(n) \) such that \( f(n) \in O(g(n)) \). Conversely, when determining the \( \Omega \) classification for \( f(n) \) we try to find the **largest** function \( g(n) \) such that \( f(n) \in \Omega(g(n)) \).

In class we did a couple of examples. Here’s another:
Let \( f(n) = 0.0001 \times n^2 + (10^6) \times n + 3 \)

We know that \( f(n) \in O(n^2) \). It’s also very easy to see that \( f(n) \in \Omega(n^2) \)… we can let \( c = 0.000001 \) and it is immediately clear that \( f(n) \geq c \times n^2 \quad \forall n \geq 0 \).

Now is it possible that \( f(n) \in \Omega(n^3) \)?

If this were the case, then there would exist a positive constant \( c \) such that

\[
\begin{align*}
f(n) &\geq c \times n^3 \quad \forall n \geq n_0 \\
n &> c \times n^2 - 0.0001 \times n - 10^6
\end{align*}
\]

but we can easily see that this is impossible: even if \( c \) is very small, as \( n \) gets large there will come a point beyond which \( c \times n^2 - 0.0001 \times n - 10^6 \) is \( \geq 1 \) so \( n \times (c \times n^2 - 0.0001 \times n - 10^6) \geq n \), which would give \( 3 \geq n \quad \forall n \geq n_0 \)... which is not possible.

Thus \( f(n) \notin \Omega(n^3) \)

\( \Omega \)This example illustrates a useful fact: if \( f(n) \) is a polynomial, then the Big O class and the \( \Omega \) class for \( f(n) \) are identical.

But this is not always the case. For example, consider this function:

\[
A(n):
\]

\[
\begin{align*}
\text{if } n \% 2 = &\ 0: \\
\quad \text{for } i = 1..n: \\
\quad \quad \text{print } '*' \\
\text{else:} \\
\quad \text{for } i = 1..n^2: \\
\quad \quad \text{print } '*'
\end{align*}
\]

Let \( f(n) \) be the time required to execute \( A(n) \). If you plot \( f(n) \) for \( n = 1, 2, 3, ... \) you will see that it has a zig-zag shape. The tops of the zigs occur when \( n \) is odd, and they grow at the
same rate as $n^2$. It is easy to see that $f(n) \in O(n^2)$. However, the bottoms of the zags, which occur when $n$ is even, do not show this behaviour - they grow at the same rate as $n$.

Referring back to our previous definitions, we are now able to say that $f(n) \in O(n^2)$ and also $f(n) \in \Omega(n)$... and neither of these can be improved: there is no lower O class for f(n), and no higher $\Omega$ class for f(n).

This example demonstrates that an algorithm’s Big O class may be different from its $\Omega$ class.

If it turns out that we can show an algorithm’s complexity is in $O(g(n))$ and in $\Omega(g(n))$, then we get very excited - it means that $g(n)$ gives both an upper and a lower bound on the growth-rate of the time required by the algorithm. Basically it means we know exactly how fast the algorithm's time requirement grows. This is so amazingly wonderful that we give it a special name:

**Theta Classification**

If $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, we say $f(n) \in \Theta(g(n))$. ($\Theta$ is the Greek letter “Theta”)

We also apply $\Theta$ classification to problems ... but that is a story for another day!