Comparing *Linear* Container Data Structures

* (all: `template<typename ItemType>*)

<table>
<thead>
<tr>
<th>List</th>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getLength()</code></td>
<td><code>isEmpty()</code></td>
<td><code>isEmpty()</code></td>
</tr>
<tr>
<td><code>insert(where?)</code></td>
<td><code>push(ItemType)</code></td>
<td><code>enqueue(ItemType)</code></td>
</tr>
<tr>
<td><code>remove(which one?)</code></td>
<td><code>pop()</code></td>
<td><code>dequeue()</code></td>
</tr>
<tr>
<td><code>retrive(which one?)</code></td>
<td><code>ItemType peek()</code></td>
<td><code>ItemType peekFront()</code></td>
</tr>
</tbody>
</table>

With respect to *insert/remove/retrieve* in the ADT List interface:

Lists may be **ordered** or **unordered**, and the method prototypes will vary accordingly. Typically:

1. If we design ADT List to maintain an item-based **ordered** list:
   a. *insert* takes object of type `ItemType` and decides where it goes
   b. *remove* and *retrive* take a **Key** (often a field of `ItemType`) and (attempt to) locate the item in the List with the matching **Key**.

2. If we design ADT List for applications in which no fixed item-based ordering is desired, *insert, remove, and retrieve* all take ordinal indices.
## Major *Linear* Container Classes

<table>
<thead>
<tr>
<th></th>
<th>Properties</th>
<th>Applications</th>
<th>Implementations</th>
</tr>
</thead>
</table>
| **List**   | add/delete/retrieve anywhere        | • General ordered or unordered collections of objects (windows in a GUI; names in a roster; objects in a drawing program)  
• Usually we want to access some or all of the elements many times *without* removing them. | • Linked list data structure  
• Array is almost always a poor choice |
| **Stack**  | LIFO: add/delete/retrieve at one end (the “top”) | • We remove objects as we access them  
• Activation records; backtracking algorithms (or anything for which recursion might be appropriate) | • Array (if max size is known at time of instantiation)  
• Linked list data structure |
| **Queue**  | FIFO: add at back; remove/retrieve from front | • We remove objects as we access them  
• Managing access to shared services or objects; used to perform certain types of traversals of advanced data structures  
• Simulations of real shared access situations (e.g., to gather statistics on service times and/or look for bottlenecks) | • Array (if max size is known at time of instantiation)  
• Linked list data structure |