Algorithms in Systems Engineering
ISE 172

Lecture 6

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Chapter 3

Searching a List

- How do we search for an item in a list?
- With an "unordered" list, we have little choice but to search the items one by one.
- The efficiency depends on where the items is in the list.
- If we are always looking for items near the beginning or end of the list, this can be efficient.
- For relatively small lists, this method is efficient enough.
- What about big lists?
- First, let's have a look at the empirical performance of the list implementations we've looked at so far.

Linked List Implementation: Search

```
def __contains__(self, item):
    current = self.head
    while current != None:
        if current.getData() == item:
            return True
        else:
            current = current.getNext()
    return False
```

Note that we search from the end of the list to the beginning.

Comparing List Implementations: Empirical

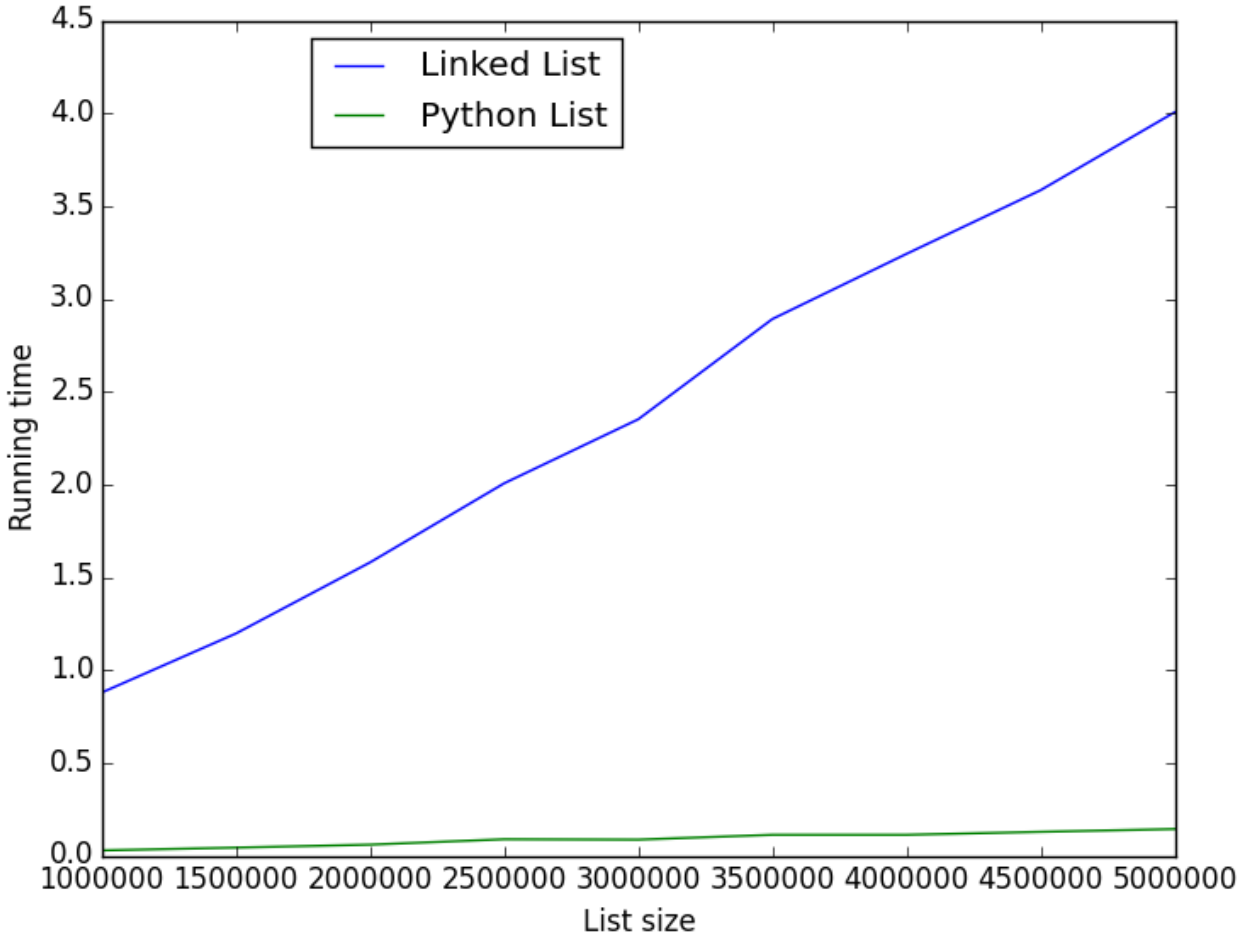

Figure 1: Comparing Linked List to Python List: Search (Worst Case)

Comparing List Implementations: Empirical

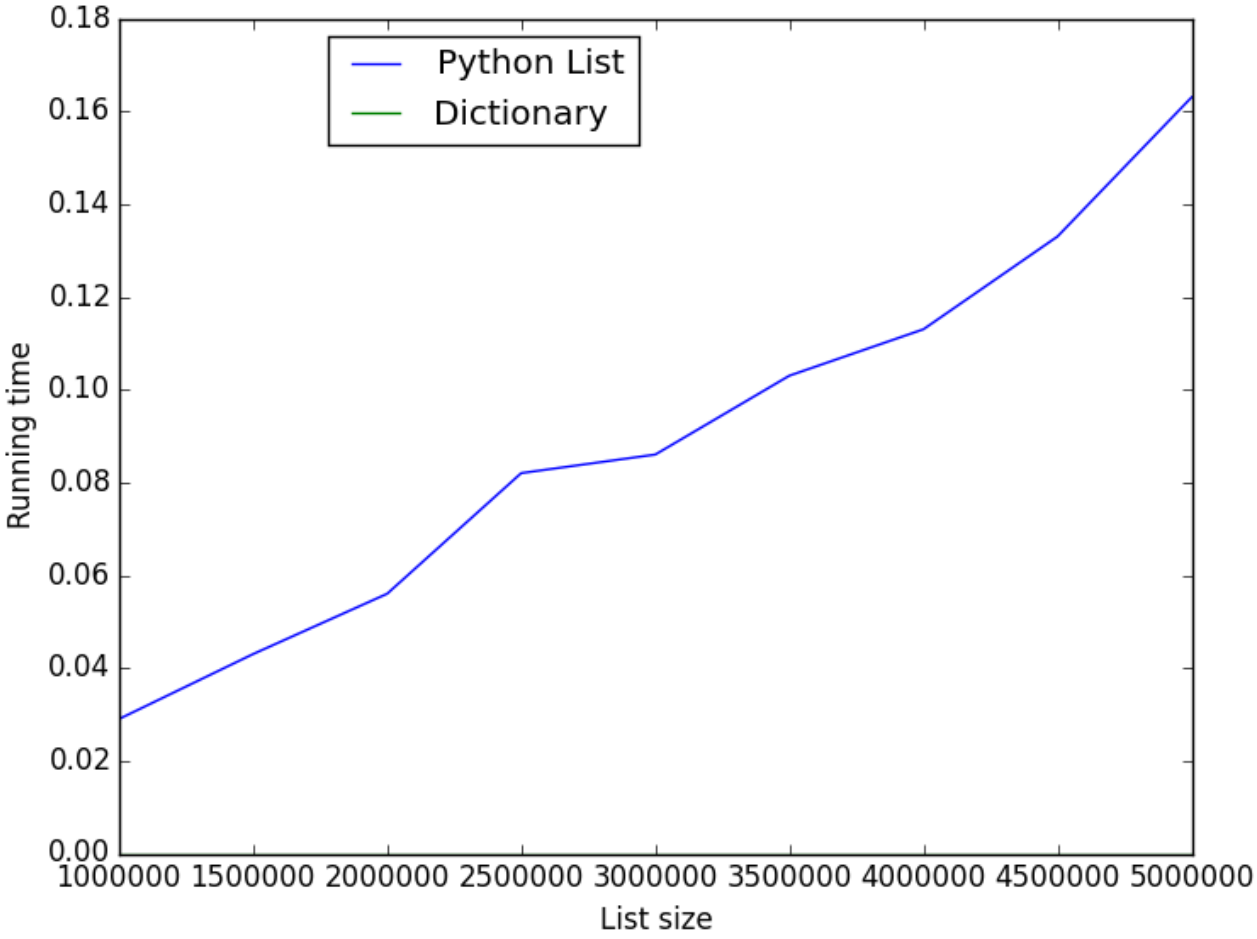

Figure 2: Comparing Python List to Dictionary: Search (Worst Case)

Dictionary Search

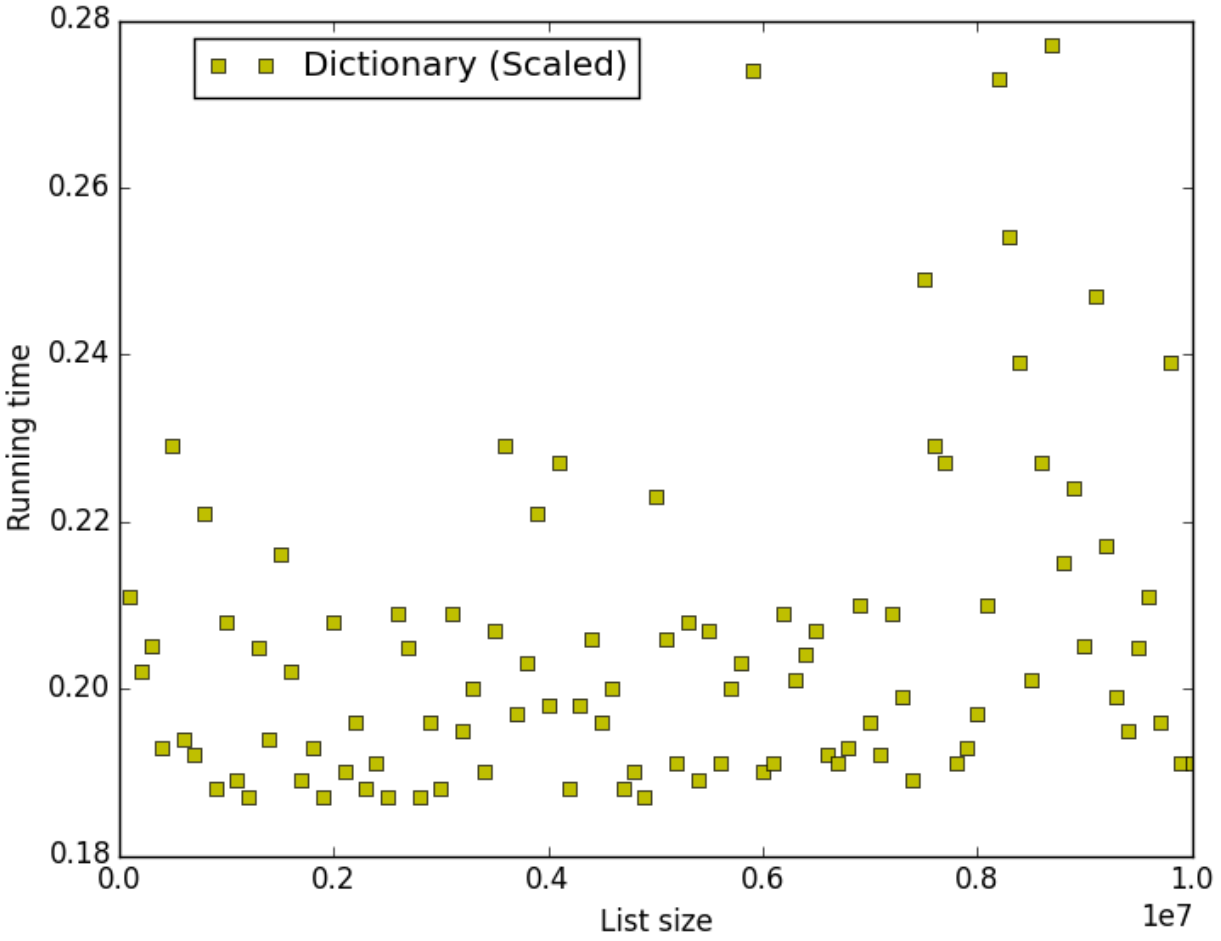

Figure 3: Performance of Dictionary: Search

Improving List Search

Can we improve the time to search a list?

- Idea: Keep the list in order.
- How much does the running time improve?
- What are the problems with this?

An Ordered List Class

```
class OrderedList(object):
    def __init__(self, searchAlgo = 'bisection nonrecursive',
                 priority = 'max', dataStruct = None):
        self.searchAlgo = searchAlgo
        if dataStruct == None:
            self.aList = []
        else:
            self.aList = dataStruct
        self.dataStruct = dataStruct
        self.priority = priority
        if priority == 'max':
            def compare(x, y):
                return x > y
        else:
            def compare(x, y):
                return x < y
        self.compare = compare
```

An Ordered List Class (cont'd)

```
def __contains__(self, item):
    if self.searchAlgo == 'bisection nonrecursive':
        return self.bisection_search(item)
    elif self.searchAlgo == 'bisection recursive':
        return self.bisection_search(item)
    elif self.searchAlgo == 'sequential':
        return self.seq_search(item)
    elif self.searchAlgo == 'bisection recursive with slices':
        l = len(self.aList)
        if l == 0: return False
        if l == 1: return item == self.aList[0]
        else:
            mid = l/2
            current = self[mid]
            if self.compare(current, item):
                return item  in self[:mid-1]
            elif current == item:
                return True
            else:
                return item in self[mid+1:]
    else:
        raise Exception, "Unknown search algorithm"
```

An Ordered List Class (cont'd)

```
def add(self, item):
    if len(self.aList) == 0:
        self.aList.append(item)
    else:
        index = len(self.aList)
        # We use a reverse iterator here, since it is
        # efficient for both Python lists and linked lists
        for i in reversed(self.aList):
            if self.compare(i, item):
                index -= 1
            else:
                break
        if index == len(self.aList):
            self.aList.append(item)
        else:
            self.aList.insert(index, item)
```

An Ordered List Class (cont'd)

```
def bisection_search(self, item, beg = 0, end = None):
    if end == None:
        end = len(self.aList) - 1
    if beg > end:
        return (comparisons, False)
    mid = (beg + end)/2
    current = self.aList[mid]
    if item == current:
        return True
    elif beg < end:
        if self.compare(current, item):
            return self.bisection_search(item, beg, mid - 1)
        else:
            return self.bisection_search(item, mid + 1, end)
    else:
        return (comparisons, False)
```

Sequential and Bisection Search: Empirical Running Time

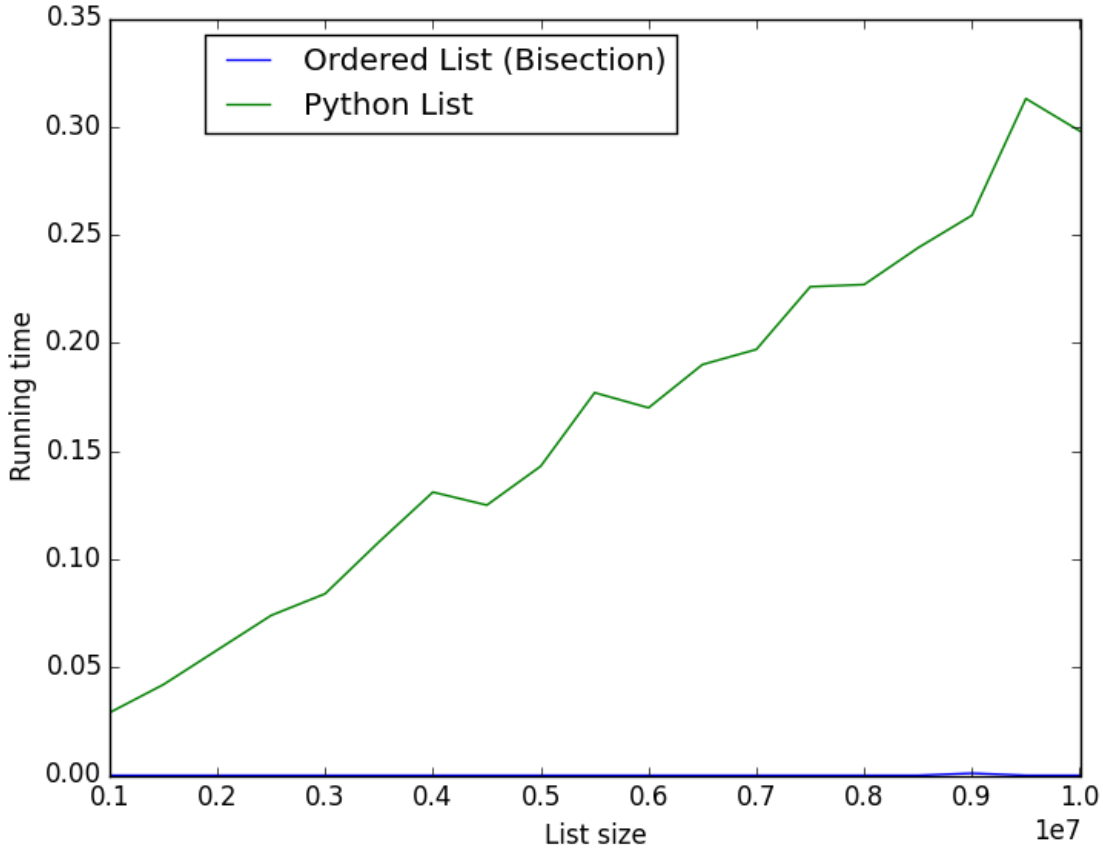

Figure 4: Running time of sequential versus bisection search with Python list (worst case)

Bisection Search: Empirical Running Time

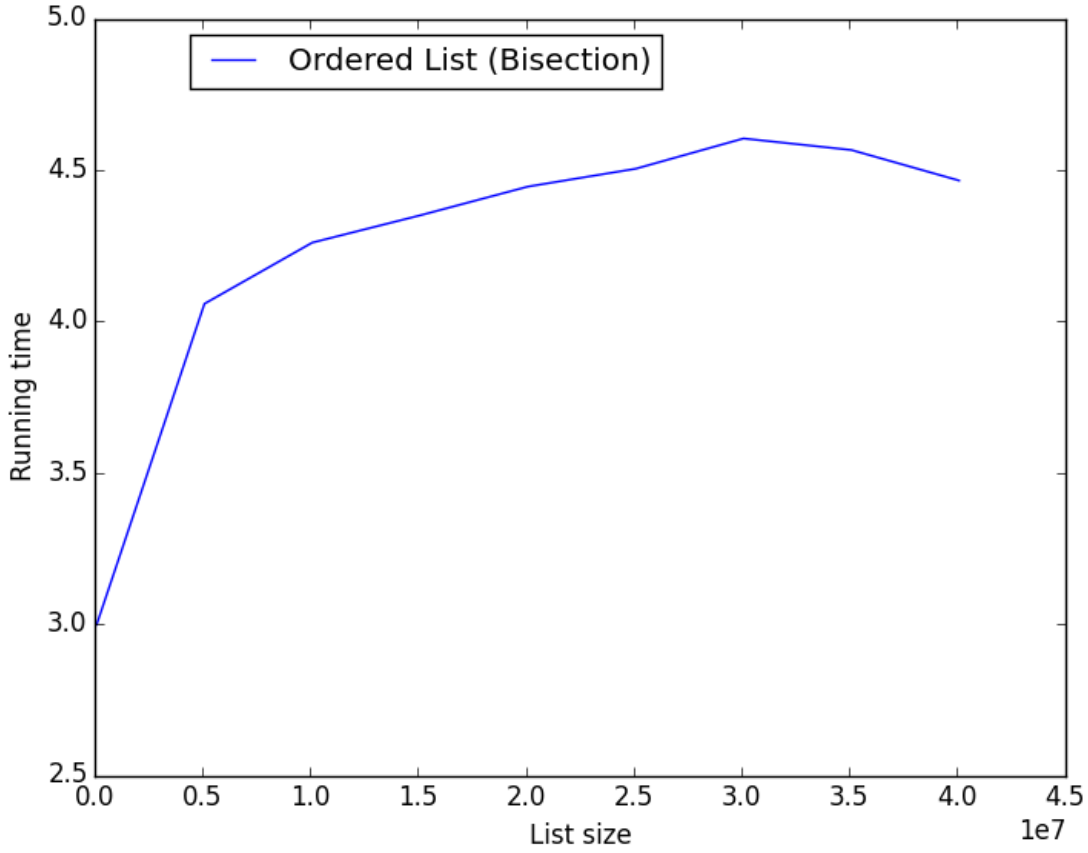

Figure 5: Running time of bissection search with Python list (worst case)

Bisection Search: Running Time with Linked List

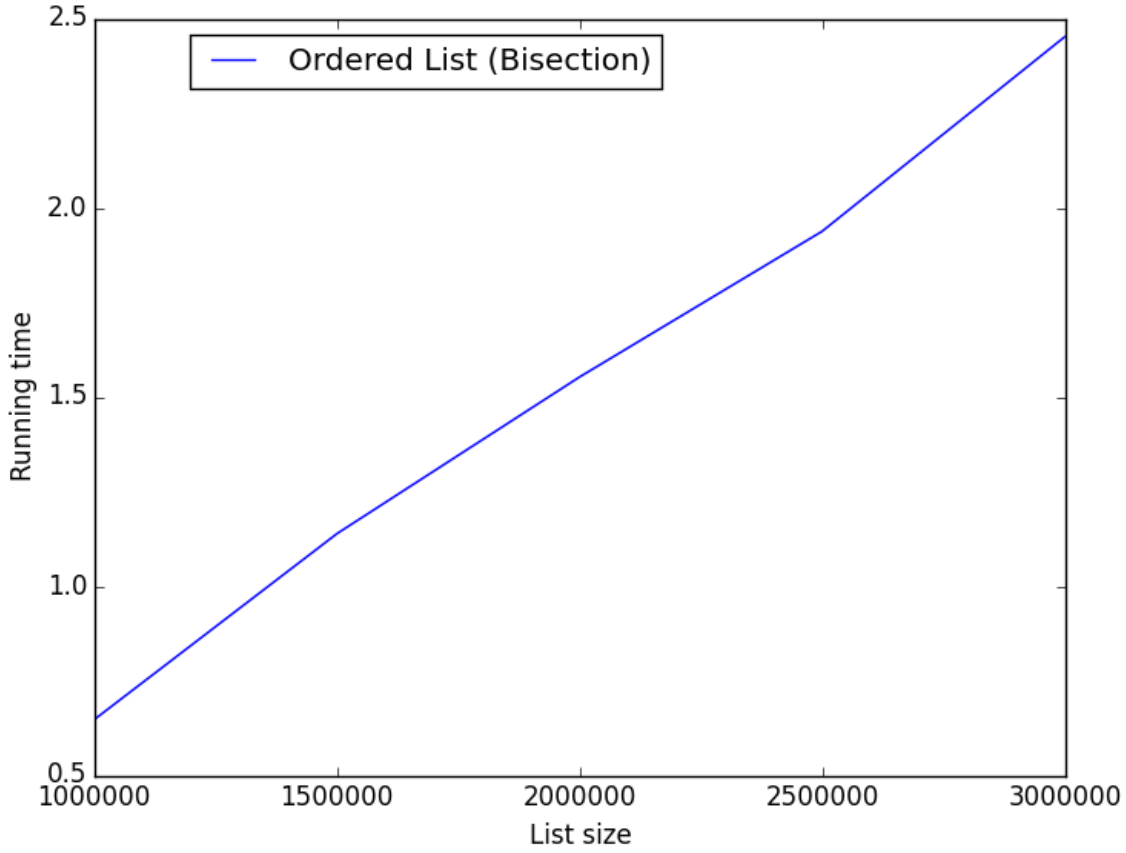

Figure 6: Bisection versus sequential search with a linked list (worst case)

Stacks

- A *stack* is a special kind of list in which items can only be removed in "last-in, first-out" (LIFO) order.
- A *queue* is a list in which items can only be removed in "first-in, first-out" (FIFO) order.
- The basic operations on a stack are
  - `push()`: Put a new item on the stack.
  - `pop()`: Take the most recently added item off the stack.
  - `peek()`: Get a copy of the most recently added item.
  - `isEmpty()`: Determine whether the stack is empty.
  - `remove()`: Remove a particular item from the stack.
- The basic operations on a queue are

Queues

- A *queue* is a list in which items can only be removed in "first-in, first-out" (FIFO) order.
- The basic operations on a queue are
  - `enqueue()`: put a new item in the queue.
  - `dequeue()`: remove the most recently added item from the queue.
  - `peek()`: Get a copy of the most recently added item.
  - `isEmpty()`: Determine whether the stack is empty.
  - `remove()`: Remove a particular item from the stack.

Implementing a Stack ADT

- Because a stack is a special kind of list, a stack can be implemented on top of a list.
- Since we only have to support specific operations, we can more easily choose what underlying list implementation will be most effective.
- How would we go about implementing a stack using a list data structure?
- How would such an implementation work with each of the underlying list implementations we've talked about?
  - Linked List
  - Array

Implementing a Queue/Deque ADT

- Queuesand deques are also special kinds of lists that can be implemented on top of a list ADT.
- Again, we only have to support specific operations, we can more easily choose what underlying list implementation will be most effective.
- How would we go about implementing queues and deques using a list data structure?
- How would such an implementation work with each of the underlying list implementations we've talked about?
  - Linked List
  - Array