5. Arrays in most HLLs are static in size (i.e., cannot grow at run-time), so arrays are constructed to hold the “maximum” number of items. For example, an array with 1,000 slots might only contain 3 items:

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 999 \\
\end{array}
\]

a) The physical size of the array is the number of slots in the array. What is the physical size of scores? \(1000\)

b) The logical size of the array is the number of items actually in the array. What is the logical size of scores? \(3\)

c) The load factor is portion of the array being used. What is the load factor of scores? \(3/1000\)

d) What is the \(O()\) for “appending” a new score to the “right end” of the array? \(O(1)\)

e) What is the \(O()\) for adding a new score to the “left end” of the array? \(O(n)\) \(n \leq \text{logical size}\)

f) What is the average \(O()\) for adding a new score to the array? \(\frac{O(n)}{n} = O(1)\)

g) During run-time if an array fills up and we want to add another item, the program can usually:
   - Create a bigger array than the one that filled up
   - Copy all the items from the old array to the bigger array
   - Add the new item
   - Delete the smaller array to free up its memory

   **double size over the old array**

h) What is the \(O()\) of moving to a larger array? \(O(n)\)

6. Consider the following list methods in Python:

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
<th>Average (O()) for myList containing (n) items</th>
</tr>
</thead>
<tbody>
<tr>
<td>index []</td>
<td>itemValue = myList[i]</td>
<td>(O(1))</td>
</tr>
<tr>
<td>append</td>
<td>myList.append(newValue)</td>
<td>(O(1))</td>
</tr>
<tr>
<td>extend</td>
<td>myList.extend(otherList)</td>
<td>(O(m))</td>
</tr>
<tr>
<td>insert</td>
<td>myList.insert(i, item)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>pop</td>
<td>myList.pop( )</td>
<td>(O(1))</td>
</tr>
<tr>
<td>pop(i)</td>
<td>myList.pop(i)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>del</td>
<td>del myList[i]</td>
<td>(O(n))</td>
</tr>
<tr>
<td>remove</td>
<td>myList.remove(item)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>index</td>
<td>myList.index(item)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>iteration</td>
<td>for item in myList:</td>
<td>(O(n))</td>
</tr>
<tr>
<td>reverse</td>
<td>myList.reverse( )</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

Dictionary Operations:

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
<th>Explanation</th>
<th>Average (O()) for (n) keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>get item</td>
<td>myDictionary.get(myKey)</td>
<td>Returns the value associated with myKey; otherwise None</td>
<td>(O(1))</td>
</tr>
<tr>
<td>set item</td>
<td>myDictionary[myKey] = value</td>
<td>Change or add myKey:value pair</td>
<td>(O(1))</td>
</tr>
<tr>
<td>in</td>
<td>myKey in myDictionary</td>
<td>Returns True if myKey is in myDictionary; otherwise False</td>
<td>(O(1))</td>
</tr>
<tr>
<td>del</td>
<td>del myDictionary[myKey]</td>
<td>Deletes the mykey:value pair</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>
1. An “abstract” view of the stack:

Using an array implementation would look something like:

```
0 1 2 3  (max-1)
items: a b c d

top: 2  
max: 100
```

Complete the big-oh notation for the following stack methods assuming an array implementation: ("n" is the # items)

<table>
<thead>
<tr>
<th></th>
<th>push(item)</th>
<th>pop()</th>
<th>peek()</th>
<th>size()</th>
<th>isEmpty()</th>
<th>isFull()</th>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-oh</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

2. Since Python does not have a (directly accessible) built-in array, we can use a list.

```python
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

Since Python uses an array of references (pointers) to list items in their implementation of a list.

a) Complete the big-oh notation for the stack methods assuming this Python list implementation: ("n" is the # items)

<table>
<thead>
<tr>
<th></th>
<th>push(item)</th>
<th>pop()</th>
<th>peek()</th>
<th>size()</th>
<th>isEmpty()</th>
<th><em>init</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-oh</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

b) Which operations should have what preconditions?

`peek, pop - stack is not empty`
3. The text's alternative stack implementation also using a Python list is:

```python
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.insert(0, item)

    def pop(self):
        return self.items.pop(0)

    def peek(self):
        return self.items[0]

    def size(self):
        return len(self.items)
```

Since an array is used to implement a Python list, the alternate Stack implementation using a list:

<table>
<thead>
<tr>
<th>&quot;Abstract&quot; Stack</th>
<th>&quot;alternate&quot; Stack Object</th>
<th>list Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

a) Complete the big-oh notation for the "alternate" Stack methods: ("n" is the # items)

<table>
<thead>
<tr>
<th></th>
<th>push(item)</th>
<th>pop()</th>
<th>peek()</th>
<th>size()</th>
<th>isEmpty()</th>
<th><strong>init</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-oh</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

4. How could we use a stack to check if a word is a palindrome (e.g., radar, toot)?

```
radar
```

```
pushing
```

```
compare
```

5. How could we check to see if we have a balanced string of nested symbols?
1. The `Node` class (in `node.py`) is used to dynamically create storage for a new item added to the stack. The `LinkedList` class (in `linked_stack.py`) uses this `Node` class. Conceptually, a `LinkedList` object would look like:

```
"Abstract"
Stack

   c
   b
   a
   top
   bottom
```

```
class Node:
    def __init__(self, initData):
        self.data = initData
        self.next = None
    def getData(self):
        return self.data
    def getNext(self):
        return self.next
    def setData(self, newData):
        self.data = newData
    def setNext(self, newNext):
        self.next = newNext
```

```
class LinkedList(object):
    
    def __init__(self):
        self.top = None
        self.size = 0
    def push(self, newItem):
        temp = Node(newItem)
        self.size += 1
        if self.top is None:
            self.top = temp
        else:
            current = self.top
            while current.next is not None:
                current = current.next
            current.next = temp
    def pop(self):
        if self.top is None:
            return None
        else:
            item = self.top.data
            self.size -= 1
            self.top = self.top.next
            return item
    def peek(self):
        if self.top is None:
            return None
        return self.top.data
    def size(self):
        return self.size
    def isEmpty(self):
        return self.size == 0
    def __str__(self):
        values = []
        current = self.top
        while current is not None:
            values.append(str(current.data))
            current = current.next
        return '\n'.join(values)
```

a) Complete the `push`, `pop`, and `__str__` methods.

b) Stack methods big-o's?
   (Assume "n" items in stack)

   - `constructor __init__`
   - `push(item)`
   - `pop()`
   - `peek()`
   - `size()`
   - `isEmpty()`
   - `str()`