Recursion
• Given an iterative method (using a loop), write a recursive method that produces the same result. Or vice versa.
  ○ Example:

  ■ Loop:
    int factorial (int num)
    {
      int fact=1;
      while(num>0)
      {
        fact*=num;
        num--;
      }
      return fact;
    }

  ■ Recursive:
    int factorial(int num)
    {
      if(num>0)
      {
        int fact =num*
          factorial(num-1);
        return fact;
      }
      else
      {
        return 1;
      }
    }

  ○ Test Yourself:

  ■ Recursive:
    int SumDivBy3(int[] num, int start, int end)
    {
      int sumDiv=0;
      if(start<end)
      {
        if(num[start]%3==0)
        {
          sumDiv+=num[start]+
            SumDivBy3(num,start+1, end);
        }
        else
        {
          sumDiv+=SumDivBy3(num, startIndex)%3!=0)
          {
            sumDiv+=SumDivBy3(num, start+1,end)
          }
        }
      return sumDiv;
    }

  ■ Loop:
    int SumDivBy3(int[] num, int start, int end)
    {
- Given a recursive method, compute the result/output based on certain parameter values.
  - Example:
    ```java
    int triple(int n)
    {
        if (n == 0)
            return 0;
        else
        {
            int total = 3 + triple(n-1);
            System.out.print(total+ " ");
            return total;
        }
    }
    ```
    - Output: triple(5);
      3 6 9 12 15

- Test Yourself:
  - void myMethod( int counter)
    ```java
    {
        if(counter == 0)
            return;
        else
        {
            System.out.print(counter+ " ");
            myMethod(--counter);
            return;
        }
    }
    ```
  - Output: myMethod(5);

- Answer:__________________________

**Linked Lists, Stacks, and Queues**

- Given code using linked lists, stacks, or queues, trace its output
  - Linked List Example

```
LinkedList list = new LinkedList();
list.add(0, "A");
list.add(0, "B");
list.add(1, "C");
// position 1
String small = list.findSmallest();
list.add(0, small);
// position 2
it.removeAllOccurances("A");
// position 3
```

<table>
<thead>
<tr>
<th>List contents at position 1:</th>
<th>List contents at position 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>___________________________</td>
<td>___________________________</td>
</tr>
<tr>
<td>List contents at position 3:</td>
<td></td>
</tr>
<tr>
<td>___________________________</td>
<td></td>
</tr>
</tbody>
</table>
### Stack Example (LIFO - last in, first out)

<table>
<thead>
<tr>
<th>Code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack stack = new Stack();</td>
<td>Stack contents at position 1:</td>
</tr>
<tr>
<td>stack.push(&quot;A&quot;);</td>
<td></td>
</tr>
<tr>
<td>stack.push(&quot;B&quot;);</td>
<td></td>
</tr>
<tr>
<td>stack.push(&quot;C&quot;);</td>
<td></td>
</tr>
<tr>
<td>// position 1</td>
<td></td>
</tr>
<tr>
<td>stack.pop()</td>
<td></td>
</tr>
<tr>
<td>String top = stack.peek();</td>
<td></td>
</tr>
<tr>
<td>stack.push(top);</td>
<td></td>
</tr>
<tr>
<td>// position 2</td>
<td></td>
</tr>
<tr>
<td>stack.pop()</td>
<td></td>
</tr>
<tr>
<td>stack.pop()</td>
<td></td>
</tr>
<tr>
<td>stack.peek()</td>
<td></td>
</tr>
<tr>
<td>// position 3</td>
<td></td>
</tr>
</tbody>
</table>

### Queue Example (FIFO - first in, first out)

<table>
<thead>
<tr>
<th>Code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue queue = new Queue();</td>
<td>Queue contents at position 1:</td>
</tr>
<tr>
<td>queue.add(&quot;A&quot;);</td>
<td></td>
</tr>
<tr>
<td>queue.add(&quot;B&quot;);</td>
<td></td>
</tr>
<tr>
<td>queue.add(&quot;C&quot;);</td>
<td></td>
</tr>
<tr>
<td>// position 1</td>
<td></td>
</tr>
<tr>
<td>String out = queue.remove();</td>
<td></td>
</tr>
<tr>
<td>queue.add(out);</td>
<td></td>
</tr>
<tr>
<td>// position 2</td>
<td></td>
</tr>
<tr>
<td>while(!queue.element().equals(out)) {</td>
<td></td>
</tr>
<tr>
<td>queue.remove();</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>// position 3</td>
<td></td>
</tr>
</tbody>
</table>

- You need to know how these data structures work, how an element can be added or removed
  - Add to Linked List
    - Add to beginning:
    - Add in the middle:
    - Add to end:
- Remove from Linked List
  - Remove from beginning:
  - Remove from middle
  - Remove from end:

- Add to Stack - push(element)
- Remove from Stack - pop()
- Add to Queue - add(element)
- Remove from Queue - remove()

- Review the methods in the LinkedList. You might be asked to write code similar to those ones (possibly different setting).
  - Add, remove, get, size, etc.
- Know the complexity (efficiency) of access and add/remove operations on array lists vs. linked lists

<table>
<thead>
<tr>
<th></th>
<th>Access at index</th>
<th>Add</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linked List</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Know the difference between static data structures and dynamic data structures
  - Example of static data structure: Array
    Why?
  - Example of dynamic data structure: ArrayList
    Why?

**Binary Trees**
- Root: The topmost node in a tree
- Leaves: Any node that does not have child nodes
- Parent: A node that has a child
- Siblings: Two nodes that Share the same parent
In Order: Visit the left child, then the parent and the right child
  ○ Example: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
Pre Order: Visit the parent first and then left and right children
  ○ Test Yourself: Use the same tree above
  ○ Answer: ____________________________.
Post Order: Visit left child, then the right child and then the parent
  ○ Test Yourself: Use the same tree above
  ○ Answer: ____________________________.
(Found at:https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html)

Binary Search Trees
● Definition: For any node x in the tree, the left subtree rooted at x only contains nodes with values
  less than or equal to the value stored at x; and the right subtree rooted at x only contains nodes with
  values greater than the value at x. Also, the left subtree and the right subtree of x must be binary
trees themselves.

Successor: Next Element from In-Order
public static TreeNode findSuccessor(TreeNode node)
{
  if (node == null)
    return null;
  if (node.getRight() != null)
    return findMinimum(node.getRight());

  TreeNode y = node.getParent();
  TreeNode x = node;
  while (y != null && x == y.getRight())
  {
    x = y;
    y = y.getParent();
  }
  return y;
}

Predecessor:
public static TreeNode findPredecessor(TreeNode node)
{
}
Minimum:
```java
public TreeNode findMinimum(TreeNode root) {
    if (root == null)
        return null;
    if (root.getLeft() != null)
        return findMinimum(root.getLeft());
    return root;
}
```

Maximum:
```java
public TreeNode findMaximum(TreeNode root) {
}
```

Example:
```
In Order: -4, 2, 3, 5, 9, 12, 19, 21, 25
- Predecessor of 3 is 2
- Predecessor of 19 is 12
- Predecessor of -4 is null
- Successor of 5 is 9
- Successor of -4 is 2
- Successor of 25 is null
- Minimum: -4
- Maximum: 25
```

Test Yourself:
```
In Order: __________________________
- Predecessor of 16 is ______
- Predecessor of 62 is ______
- Predecessor of 70 is ______
- Successor of 41 is ______
- Successor of 63 is ______
- Successor of 74 is ______
- Minimum: ______
- Maximum: ______
```
- **Delete:**
  - If the node to be removed has 2 children:
    - find a minimum value in the right subtree;
    - replace value of the node to be removed with found minimum. Now, right subtree contains a duplicate!
    - apply remove to the right subtree to remove a duplicate.
  - Example(found at http://www.algolist.net/Data_structures/Binary_search_tree/Removal):

<table>
<thead>
<tr>
<th>Step</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Remove 12 from the below tree:</td>
<td><img src="image1.png" alt="Diagram 1" /></td>
</tr>
<tr>
<td>2. Find minimum element in the right subtree of the node to be removed. In current example it is 19.</td>
<td><img src="image2.png" alt="Diagram 2" /></td>
</tr>
<tr>
<td>3. Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.</td>
<td><img src="image3.png" alt="Diagram 3" /></td>
</tr>
<tr>
<td>4. Remove 19 from the left subtree.</td>
<td><img src="image4.png" alt="Diagram 4" /></td>
</tr>
</tbody>
</table>
Add:
  ○ if a new value is less, than the node's value:
    ■ if a current node has no left child, place for insertion has been found;
    ■ otherwise, handle the left child with the same algorithm.
  ○ if a new value is greater, than the node's value:
    ■ if a current node has no right child, place for insertion has been found;
    ■ otherwise, handle the right child with the same algorithm.
  ○ Test Yourself (Found at http://www.algolist.net/Data_structures/Binary_search_tree/Insertion):

```
Insert 4 Into The Below Tree:
Answer:
```

Heaps
- Definition: A special kind of tree that follows a specific ordering throughout the levels (for example, min or max heap).
- Array Representation:
  ○ Heaps are stored in an array by putting the root in the first index (0), it's 2 children in the next two indices (1, 2), the next four children in the next indices (3,4,5,6), etc.
  ○ Parents: floor((i+1) / 2)
  ○ Right Children: 2i + 2
  ○ Left Children 2i +1
- Insertion:
  ○ Insert the new element at the end of the array, or, at the bottom right most of the heap.
  ○ While the heap property is violated, swap the heap with it parent.
  ○ Example: Insert -2 into the heap
    ■ (found at http://www.algolist.net/Data_structures/Binary_heap/Insertion)
### Heap Representation vs. Array Representation

<table>
<thead>
<tr>
<th>Heap Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Heap Diagram" /></td>
</tr>
<tr>
<td><strong>The -2 will initially be added as the right child of node 6</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image2" alt="Array Diagram" /></td>
</tr>
<tr>
<td><strong>Since this is a min-heap, the arrangement of -2 and 6 violates the min-heap property, and they must be swapped</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3" alt="Heap Diagram" /></td>
</tr>
<tr>
<td><strong>There is a violation of the min-heap property with 1 and -2, so we must swap them. After this swap, we are done.</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4" alt="Array Diagram" /></td>
</tr>
</tbody>
</table>

### Extraction of a Root:
- To remove the root, copy the last element of the array into the 0th position, and decrease the size by 1.
- The heap property may now be violated, so we have to examine its children and swap with the appropriate child until the heap property is not longer violated.

### Heap Sort:

```java
HeapSort(A) {
    buildHeap(A);
    for(i = A.length()-1; i >= 1; i--) {
        swap (A[0], A[i]);
        size = size - 1;
        heapify(A, 0);
    }
}
```
- **Build-Heap Operation:**

  ```java
  buildHeap(A) {
      for(int i = size/2; i >= 0; i--) {
          heapify(A, i);
      }
  }
  ```

- **Complexity:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td></td>
</tr>
<tr>
<td>Extraction of a Root</td>
<td></td>
</tr>
<tr>
<td>Heap Sort including Build-Heap Operation</td>
<td></td>
</tr>
</tbody>
</table>