5. (10 points) Evaluate the following postfix expression and write your answer in the specified location. Show the state of the evaluation stack just before the first operation is applied (along with the operation), then just before each additional operation is performed (along with the operation), followed by the final value on the stack. All values are integers. All arithmetic is integer arithmetic.

**Postfix Expression:** 2 3 ^ 4 * 8 7 5 – / +

**Stack**
- 2 3
- 8 4
- 32 8 7 5
- 32 8 2
- 32 4
- 36

**Final value of the expression:** 36

6. (10 points) Given that you have access to the partition method (see Question 7 below), complete the quickSort method — in other words, write the quickSort used by the public quickSort. For the Quick Sort algorithm, here is Sedgewick’s partition method, modified for Comparables and using more self-documenting variable names:

```java
public static void quickSort ( Comparable[] array )
{  quickSort ( array, 0, array.length-1 );  }
private static void quickSort ( Comparable[] array, int lo, int hi )
{  int mid = partition(array, lo, hi);  // Could be while ( lo < hi )
   quickSort ( array, lo, mid-1 );
   quickSort ( array, mid+1, hi );
   // Would be lo = mid + 1;
}
```

For the Quick Sort algorithm, here is Sedgewick’s partition method, modified for Comparables and using more self-documenting variable names:

```java
public static void quickSort ( Comparable[] array )
{  quickSort ( array, 0, array.length-1 );  }
private static void quickSort ( Comparable[] array, int lo, int hi )
{  int mid = partition(array, lo, hi);  // Could be while ( lo < hi )
   quickSort ( array, lo, mid-1 );
   quickSort ( array, mid+1, hi );
   // Would be lo = mid + 1;
}
```

7. (15 points) Show the condition of the array segment given below after this loop has been executed with the indicated initial conditions, before the final swap that puts the pivot into its final position. Be sure to show your work. Otherwise partial credit is IMPOSSIBLE.

<table>
<thead>
<tr>
<th>Subscript</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>7</td>
<td>3</td>
<td>15</td>
<td>13</td>
<td>9</td>
<td>14</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Pivot</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>9</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

**Initial values on entering loop:**
- up: -1 one less than first
- down: 14 last
- pivot: 8 contents of a[14]

**Contents**
- 7 3 4 2 6 1 5 14 9 3 10 12 11 15 8

**Final values when exiting the loop:**
- up: 3
- down: 5
The tear-off sheet at the back of the exam gives the implementation of the circular-array version of a queue that we studied in class. The following program runs using that implementation of a queue.

```java
public class Question8 {  public static void main (String[] args)    {  ArrayQueue theQ = new <Integer> ArrayQueue(5);       Integer val;       System.out.println("Rtn " + val);    } }
```

8. (15 points) Note that the maximum queue size is five (array size of six, thanks to Sedgewick’s code) and care has been taken to avoid reallocation. Trace the execution of this program and show all of the output generated by the program. Then show the internal state of the object `theQ`: the value of `theQ.head`, the value of `theQ.tail`, and the contents of `theQ.q`. Note: show the values of `theQ.head` and `theQ.tail`, not their contents. Also, show only the cells of `theQ.q` that are presently in use (since the get method nulls out the cell when an item is removed). Show all of your work — otherwise partial credit cannot be given if you do not show exactly the correct answer.

```
Output
Workspace for Queue States
__  __  __  __  __  __    head 6  tail 0
Rtn 1
  1  9  2  5  __  __    head 6  tail 4
  __  __  __  __  __  __    head 1  tail 4
  __  __  __  __  __  __    head 1  tail 0
Rtn 9
  __  6  2  5  8  3    head 2  tail 0
  __  __  __  __  __  __    head 2  tail 1
Rtn 2
  __  __  6  5  8  3    head 3  tail 1
Final State
Value of theQ.head: 3
Value of theQ.tail: 1
Contents of theQ.q
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

9. (15 points) Hash table exercise: This hash table will use linear collision processing (or linear probing) for overflow handling — the value is placed in the next available location (wrap around if you are at the end of the array). You will be given a series of non-negative integers, which are to be hashed into this table using the two part hash function given below. You may assume that the values to be hashed will be stored as ints in the hash table and that the individual digits of the values are also available for hashing. You may also assume that the entire hash table has been appropriately initialized with integer values of –1 (the –1 is used to signify a location is unoccupied).

```
int hashTable[] = new int[13];
for (int k = 0; k < hashTable.length; k++)      hashTable[k] = -1; // all values in table initialized to –1
```

Hash Value (integer value) is determined by:
1. If the number is a single digit, use the number as the ExtractedNumber; otherwise concatenate the first and last digits to generated the ExtractedNumber.
2. Applying the following function to the extracted number: HashValue = ExtractedNumber % 13

Using the information given above and your knowledge of this method, fill in the table below showing the complete hash table after the following values have been hashed in the order shown. (All the hash function results have already been done) The first one has been done for you. If a collision occurs at the bottom of the array, wrap around to the top. NOTE 1: Be sure to show the value as well as when it was inserted. NOTE 2: if a value is placed elsewhere as a result of a collision, place a ‘C’ in the rightmost column of the table where that value was placed (not at each location it collides with).

```
<table>
<thead>
<tr>
<th>Input #</th>
<th>Value</th>
<th>Resulting hash value from hash function</th>
<th>hashTable index</th>
<th>hashTable value</th>
<th>When Inserted (1-10)</th>
<th>Collision Column (Place a C where appropriate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>76574</td>
<td>9</td>
<td>0</td>
<td>34934</td>
<td>10</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>24791</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>390876</td>
<td>10</td>
<td>2</td>
<td>402971</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>402971</td>
<td>2</td>
<td>3</td>
<td>23599</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>23599</td>
<td>3</td>
<td>4</td>
<td>1577</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>169540</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>9</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>1577</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>24581</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>76574</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>34934</td>
<td>8</td>
<td>9</td>
<td>390876</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>169540</td>
<td>6</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>8</td>
<td>11</td>
<td>8</td>
<td>8</td>
<td>C</td>
</tr>
</tbody>
</table>
```

Captured 2010-Feb-26 at 15:33