CSE 331 Spring 2020 Homework 2 (part 2)

1. Fill in the proof of correctness for the method on the next page, `countOccurrences`, which returns the number of times a given character occurs within a given string.

   Reason in the direction (forward or backward) indicated by the arrows on each line: forward outside the loop, but mixing forward and backward within the loop body so that the two directions meet inside the body of the "if" and "else" statements.

   In addition to filling in each blank below, you must provide additional explanation whenever two blanks appear right next to each other, with no code in between: in those cases, explain why the top statement *implies* the bottom one. You can skip this explanation if the two statements are identical or if the bottom one simply drops facts included the top one.

   The notation #c(w) indicates the number of 'c' characters appearing in the string w. This function is defined recursively as follows:

   - #c("") = 0 // empty string
   - #c(wc) = #c(w) + 1 // last character is a c
   - #c(wa) = #c(w) // last character is not a c

   For example, #c("cacc") = #c("cac") + 1 = #c("ca") + 2 = #c("c") + 2 = #c("") + 3 = 0 + 3 = 3.

   The notation s[a .. b] refers to the substring from indexes a up through and including index b. For example, if s = "cacc", then s[1 .. 2] = "ac". Special cases:

   - When b < a, there are no indexes in this range, so the result is an empty string. For example, s[0 .. -1] = "" while s[0 .. 0] = "c".
   - When a = 0 and b = s.length – 1, every index is in this range. In other words, we have s[a .. b] = s[0 .. s.length – 1] = s.

```
int countOccurrences(String s, char c) {
↓  int n = s.length();
   {{ ____________________________________________________ }}
↓  int m = 0;
   {{ ____________________________________________________ }}
↓  int i = 0;
   {{ ____________________________________________________ }}


   {{ Inv: m = #c(s[0 .. i-1]) and n = s.length }}
   while (i != n) {
↓
     {{ __________________________________________________ }}
↓    if (s.charAt(i) == c) {
       {{ ________________________________________________ }}


       {{ ________________________________________________ }}
↑      m = m + 1;
       {{ ________________________________________________ }}
↓    } else {
       {{ ________________________________________________ }}


       {{ ________________________________________________ }}
↑      m = m + 0;
       {{ ________________________________________________ }}
     }
     {{ __________________________________________________ }}
↑    i = i + 1;
     {{ __________________________________________________ }}
↑  }

↓
   {{ ____________________________________________________ }}


   {{ m = #c(s) }}
   return m;
}
```

2. Fill in the proof of correctness for the method on the next page, `middleNode`, which takes a reference to the front of linked list and returns a reference to its middle element.

   Reason in the direction (forward or backward) indicated by the arrows on each line: forward inside the loop, and a mix of forward and backward outside the loop.

   In addition to filling in each blank below, you must provide additional explanation whenever two assertions appear right next to each other, with no code in between: in those cases, explain why the top statement *implies* the bottom one. You can skip this explanation if the two statements are identical or if the bottom one simply drops facts included the top one.

   The code makes use of the following Node type, which represents a linked list node.

```
public class Node {
  public int value;
  public Node next;
}
```

   You can think of `null` as being the last node in the list.

   (Note that the variables `n` and `m` are only included to make the analysis clear. They could be removed without changing the behavior of the method.)

```
Node middleNode(Node front) {
↓  int n = 0;
   {{ ________________________________________________ }}
↓  Node curr = front;
   {{ ________________________________________________ }}
↓  Node half = front;
   {{ ________________________________________________ }}

   {{ Inv: n nodes before half and 2n nodes before curr }}
   while ((curr != null) && (curr.next != null)) {
↓
     {{ ______________________________________________ }}
↓    curr = curr.next;
     {{ ______________________________________________ }}
↓    curr = curr.next;
     {{ ______________________________________________ }}
↓    half = half.next;
     {{ ______________________________________________ }}
↓    n = n + 1;
     {{ ______________________________________________ }}
   }
↓
   {{ ________________________________________________ }}
   int m = 2*n;
   {{ ________________________________________________ }}
↓  if (curr == null) {
     {{ ______________________________________________ }}


     {{ ______________________________________________ }}
↑   } else {
     {{ ______________________________________________ }}


     {{ ______________________________________________ }}
↑    m = m + 1;
     {{ ______________________________________________ }}
↑   }

  {{ m nodes in list and m/2 (note: integer division) before half }}
  return half;
}
```

3. Fill in the missing parts of the implementation of the method `joinStrings` on the next page. It takes an array of strings and returns a single string containing all of those individual strings, in the same order, but separated by spaces. (Formal definition below.)

   The invariant for the loop is **already provided**. Your code must be correct with this invariant. You should not add any additional loops.

   You do not need to turn in a proof of correctness, but you should complete one since your code will be graded on correctness.

   The notation join(A) refers to the mathematical function of joining strings as described above. We can define this mathematical function recursively as follows:

   - join([]) = “” // empty array
   - join([x]) = x // one element array
   - join([x1, .., x_{k-1}, x_k]) = join([x1, …, x_{k-1}]]) + “ “ + x_k // at least two elements

   For example, join([“I”, “love”, “CSE 331”]) = join([“I”, “love”]) + “ “ + CSE331” = join([“I”]) + “ “ + “love” + “ “ + “CSE 331” = “I”+ “ “ + “love” + “ “ + “CSE 331” = “I love CSE 331”.

   The notation A[a .. b] refers to the subarray from indexes a up through and including index b. For example, if A = [1, 2, 3, 4], then A[1 .. 2] = [2, 3]. If b < a, then the result is empty, [].

   A `StringBuilder` allows you to create a long string by appending a sequences of pieces, one at a time. When you are done, you call `toString` to turn the sequence of pieces into a single contiguous string. See the official Java documentation for more details.

```
{{ Precondition: A != null, A contains no nulls }}
String joinStrings(String[] A) {
  StringBuilder t = new StringBuilder();



  int i = 1;



  {{ t = join(A[0.. i-1]) }}
  while (                            ) {




    i = i + 1;
  }



  {{ t = join(A) }}
  return t.toString();
}
```

4. Fill in the missing parts of the implementation of the method `decode` on the next page. This method takes a string where every other character, starting with the first, is a digit. It returns a single string where each non-digit character is repeated the number of times indicated by the digit just before them. (See the formal definition below.)

   Your implementation may have more than one loop, but you must **provide the invariant** of each loop and your code must be correct **with those invariants**.

   Formally, decode(s) is the mathematical function defined recursively as follows:

   - decode(“”) = “” // empty string
   - decode(wda) = decode(w) + d copies of a // at least two characters, d and a

   For example, decode(“1a2b3c”) = decode(“1a2b”) + 3 copies of “c” = decode(“1a2b”) + “ccc” = decode(“1a”) + 2 copies of “b” + ccc” = decode(“1a”) + “bbccc” = decode(“”) + 1 copy of “a” + “bbccc” = decode(“”) + “abbccc” = “” + “abbccc” = “abbccc”.

```
{{ Precondition: s != null, s has even length, s has digits at even indexes }}
String decode(String s) {
  StringBuilder t = new StringBuilder();

  {{ t = decode(s) }}
  return t.toString();
}
```