CSE 331 Autumn 2021 HW2

General Rules:

- For logical operators, you may use words (e.g., "or") or any standard symbols (e.g., "V").
- Assume that
    - all numbers are integers
    - integer overflow will never occur
    - integer division rounds toward zero (as in Java) (Example: 5 / 2 = 2)
- Simplify but ***do not weaken*** your assumptions.
- Due Monday, October 18 at 5pm on Gradescope

1. Fill in the proof of correctness for the method evalPoly on the next page. It takes the coefficients of a polynomial in the array A. For example, the polynomial $3 + 2x + 4x^2$ would be represented with A = [3, 2, 4]. It takes a value for x in the second parameter, v. The goal of the method is to return the value of the polynomial at x = v.

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 in the top one. The lines where you might need to provide an explanation are marked "E".

Reason in the direction (forward or backward) indicated by the arrows on each line. Within the body of the loop, you will reason forward from the top and backward from the bottom, with the two meeting in the middle as indicated. Where they meet, explain why the top assertion implies the bottom one.

Notes on the notation used:

- Use "n" to denote the length of A.
- The precondition, which is assumed true at the start of the method, will remain true throughout (since A.length cannot be changed). To save space, we will not bother to write this additional fact (0 < A.length) on every line. (We will do the same for the preconditions in each of the other problems in this assignment as well.)

```
{{ Precondition: 0 < n = A.length }}
float evalPoly(float[] A, float v) {
↓       int i = A.length – 1;
        {{ ____________________________________________________________ }}
↓       int j = 0;
        {{ ____________________________________________________________ }}
↓       float val = A[i];
        {{ ____________________________________________________________ }}

E

        {{ Inv: val = A[i] + A[i+1] v + ... + A[n-1] v^j and i + j = n - 1 }}
        while (j != A.length - 1) {
                {{ ____________________________________________________ }}
↓               j = j + 1;
                {{ ____________________________________________________ }}
↓               i = i – 1;
                {{ ____________________________________________________ }}

E

                {{ ____________________________________________________ }}
↑               val = val * v + A[i];
                {{ ____________________________________________________ }}
↑       }

↓
        {{ ______________________________________________________ }}

E

        {{ Postcondition: val = A[0] + A[1] v + ... + A[n-1] v^(n-1) }}
        return val;
}
```

2. Fill in the missing parts of each implementation of the method removeChar below. It takes as input a string s and a character c, and it returns s with all the c's removed.

In each case, the precondition and postcondition are the same, so each loop is trying to accomplish the same task, but the provided code or the invariant is slightly changed, so the details of how the code will accomplish it should be different.

In each case, ***your code must be correct according to the loop invariant that is provided***. You do not need to turn in your proof of correctness. However, since your code will be graded on its correctness, you should still complete this for yourself.

Additionally, you may not (1) add any additional statements outside of the loop, (2) add any additional loops, or (3) call any methods other than String.charAt, String.length, and StringBuilder.append.

Notes on the notation used:

- To formally argue correctness, we need a formal definition of what the code is supposed to do. The function del_c referenced in the assertions below does that. Informally, $\mathrm{del}_c(s)$ means the string s but with all c's removed.
- Formally, we can define del_c recursively (as in CSE 311) as follows:

$$\begin{array}{ll} \mathrm{del}_c(\text{“”}) = \text{“”} & \\ \mathrm{del}_c(wa) = \mathrm{del}_c(w) & \text{if } a = \mathrm{c} \\ \mathrm{del}_c(wa) = \mathrm{del}_c(w)a & \text{otherwise} \end{array}$$

  Here, *w* represents any string and *a* any single character. Any string is either the empty string “” or can be written either in the form *wa* for some *w* and some *a*.
- The notation “s[i .. j]” means the substring from index i up to and including index j. For example, if s = “abcde”, then s[2 .. 3] = “cd”. If j = i – 1, then this indicates an empty string. In symbols, we have s[i .. i-1] = “”.

a)

```
{{ Precondition: s != null }}
String removeChar(String s, char c) {
        StringBuilder t = new StringBuilder();
        int i =

        {{ Inv: t = del_c(s[0 .. i-1]) }}
        while (________________________) {




                i = i + 1;
        }

        {{ Postcondition: t = del_c(s) }}
        return t.toString();
}
```

b)

```
{{ Precondition: s != null }}
String removeChar(String s, char c) {
        StringBuilder t = new StringBuilder();
        int i =

        {{ Inv: t = del_c(s[0 .. i]) }}
        while (________________________) {




                i = i + 1;
        }

        {{ Postcondition: t = del_c(s) }}
        return t.toString();
}
```

c)

```
{{ Precondition: s != null }}
String removeChar(String s, char c) {
        StringBuilder t = new StringBuilder();
        int i =

        {{ Inv: t = del_c(s[0 .. i]) }}
        while (________________________) {
                i = i + 1;






        }

        {{ Postcondition: t = del_c(s) }}
        return t.toString();
}
```

d)

```
{{ Precondition: s != null }}
String removeChar(String s, char c) {
        StringBuilder t = new StringBuilder();
        int i =

        {{ Inv: t = del_c(s[0 .. i-1]) }}
        while (________________________) {
                i = i + 1;







        }

        {{ Postcondition: t = del_c(s) }}
        return t.toString();
}
```

3. Fill in the proof of correctness for the method on the next page, middleNode, which takes a reference to the front of a 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 in the top one. The lines where you might need to provide an explanation are marked “E”.

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;
        {{ ____________________________________________________________ }}


E


        {{ 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;
            {{ ______________________________________________________ }}

E
            {{ ______________________________________________________ }}

↓       }
        {{ ____________________________________________________________ }}
        int m = 2*n;
        {{ ____________________________________________________________ }}
↓       if (curr == null) {
            {{ ______________________________________________________ }}


E

            {{ ______________________________________________________ }}
```

(code continues on the next page…)

```
↑         } else {
                    {{ ______________________________________________________________ }}


E

                    {{ ______________________________________________________________ }}
↑                   m = m + 1;
                    {{ ______________________________________________________________ }}
↑         }
          {{ m nodes in list and m/2 (note: integer division) before half }}
          return half;
}
```

4. Fill in the missing parts of the method removeDups on the next page. It takes as input two arrays of integers, A and B, both of length at least n. A is also promised to be sorted.

The goal of the method is to copy the first n elements of A except those that are duplicates into the array B. For example, if the input is A = [1, 1, 2, 3, 3] and n = 5, then the method will copy [1, 2, 3] (the elements of A with duplicates removed) into B and return 3 as its length.

In this case, the loop invariant is provided. ***Your code must be correct with this invariant.***

You do not need to turn in your proof of correctness. However, since your code will be graded on its correctness, you should still complete this for yourself.

Additionally, you may not (1) add any additional loops or (2) call any other methods.

Notes on the notation used:

- The notation “set(A)” means the mathematical set containing the elements in A

```
{{ Precondition: 0 < n <= A.length, B.length and A[0] <= A[1] <= ... <= A[n-1] }}
int removeDups(int[] A, int[] B, int n) {








    {{ Inv: Precondition and B[0] < B[1] < ... < B[k-1] and set(B[0 .. k-1]) = set(A[0 .. i-1]) }}
    while (________________________) {














    }









    {{ B[0] < B[1] < ... < B[k-1] and set(B[0 .. k-1]) = set(A[0 .. n-1]) }}
    return k;
}
```