Question 3: Interacting Abstract Data Types -

New ADT - Fraction

You may **not** use functions from any C library that implements fractions

```
typedef struct {
    long long num;
    long long denom;     /* you may also use “unsigned long long denom”, either approach is fine */
} Fraction;
```

- Implement `int set_fraction(Fraction * fract, long long num, long long denom)`
  - Sets the numerator and denominator in the Fraction structure
  - Only the numerator can be negative
    - If the denom parameter is negative, negate the num parameter and store denom in the struct as a positive value
  - The denom parameter cannot be 0
    - If it is zero do not set the num and denom in fract and return FALSE (where FALSE is a #define set to 0)
  - If the num and denom can be successfully set, return TRUE (where TRUE is a #define set to 1)

- Implement `print_fract(Fraction * fract, int mode)`
  - When mode is SIMPLE
    - Print out num/denom
    - If fract has num = 4 and a denom = 3 it should print “4/3” to stdout
  - When mode is MIXED
    - If fract has num = 3 and a denom = 4, print “3/4” to stdout
    - If fract has num = 4 and a denom = 3, print “1 1/3” to stdout
    - If fract has num = 4 and a denom = 1, print “4” to stdout
  - SIMPLE and MIXED are two integers of your choice, using #define in a .h

- Implement `void simplify(Fraction * fract)`
  - This is computed by finding the GCD of the numerator and the denominator and dividing it out from each respectively
    - i.e. for a / b
      - find g = gcd(a, b)
      - the simplified form of a / b is (a/g) / (b/g)
    - e.g. 6/18
      - gcd(9, 12) == 3
      - (9/3) / (12/3) == 3 / 4

note: when you wrote GCD for Q2, qcd(a,b) should equal gcd(b,a)

- Implement `int add_fract(Fraction * result, Fraction * x, Fraction * y)`
  - To compute $a/b + c/d$, i.e. to add two fractions, use the following formula $(ad + bc)/bd$, simplified
  - Check to make sure that the result doesn't overflow/underflow
    - i.e the addition produces a result greater than a long can represent
    - when the result is positive it is called an overflow
    - when negative it is called an underflow
  - to check for an overflow/underflow, understand and use the solution posted in the most popular reply from the following stackoverflow page
    - https://stackoverflow.com/questions/199333/how-do-i-detect-unsigned-integer-multiply-overflow
  - If the addition overflows/underflows, return FALSE and do not update fract
  - Otherwise return TRUE

Extend Map/Reduce/etc. with Filter

- Implement `Sorted_List * filter (Sorted_List * list, filter_fn pointer)`
  - the function creates a **new** sorted list based on the filtered values (added node by node), remember the nodes have to be copied
  - the function filters based on the value, not the key
  - for full marks, implement filter() using recursion
  - store filter() in the same .c file where the other map/reduce/ etc. functions are stored
    - you do not need to submit two different .c files, just one will do
      - filter will just not be exercised when the Q1b is tested

To test question 3

Write a program called `a4q3.c`

- The program must read in a text file that contains a series of commands, one per line, with the name of the text file entered as a command line argument
  - Base this code on the code you used in q1a to implement command entries from a file
  - However, the code will need to be modified as detailed below
- You will need to store fractions in a sorted list using the Sorted_List data type
  - value_type should be of type Fraction
  - key_type should be a double and hold the decimal equivalent of the value stored in the Node
    - e.g. if value stores the fraction 11/4, then key == 2.75
- You will have to have your make file recompile all files that mention or use value_type and key_type variables or Sort_List structs when compiling the program
  - You will need to add #ifdef FRACT to compile using the Fraction typedef definition of value_type
  - E.g. if you stored all your Sort_List ADT functions in a single file called sort_list.c Then for a4q4.c you could have in your make file a command like

    ```
    gcc  -Wall -ansi -DFRACT -c sort_list.c
    ```
- All commands for entry from the input file are listed on the next couple of pages

List of Commands from the Input File

You only need a single Sorted List, like in q1a, and q2, not the array of sorted lists as in q1b

Silent Commands (modifies the list but does not print anything other than the command itself)

- a *n/d*
  - appends to a sorted list
    - with *n* stored in the numerator field of the Fraction held in `node->value`, and *d* stored in the denominator field of the Fraction
    - the decimal value equivalent of the fraction should be stored in the key field

      note: when echoing the command, the fraction is output without simplification and the key is displayed with 3 decimal places
  - example
    - *commands, as stored in the input file*

      ```
      a   5/4
      a   3
      a   4/6
      ```
    - *output* (11 – 1 spaces after the colon)

      ```
      a:          1.250  5/4
      a:          3.000  3
      a:          0.667  4/6
      ```
- p *n/d*
  - same as a except it pushes instead of appends the key-value pair onto the sorted list

Report Commands (prints information, but does not modify the list)

- `print_all` *print_mode*
  - print the sorted list at index *n* in insertion order
  - Using the input from the append examples above

    For the command "`print_all SIMPLE`", the output should be

    ```
    print_all:  Simple Fractions, Insertion Order
         1.250  5/4
         3.000  3/1
         0.667  2/3
    ```
- `print_sort` *print_mode*
  - print the sorted list at index *n* in key sort order
  - Using the input from the append examples above

    For the command "`print_sort MIXED`", the output should be

    ```
    print_sort: Mixed Fraction, Key Sort Order
         0.667  2/3
         1.250  1 1/4
         3.000  3
    ```

- `sum` *print_mode*
  - sums the values of the sorted list into a simplified fraction, which is printed (in insertion order)
  - Using the input from the append examples above
    For two commands

    ```
    sum  SIMPLE
    sum  MIXED
    ```

    the output should be

    ```
    sum:        result = 4 11/12
    sum:        result = 59/12
    ```

  - If the sum enters an overflow situation, the output should be

    ```
    sum:        result = OVERFLOW
    ```

    - This could happen in either the numerator or denominator at any point in the calculation
    - If the numerator is negative, instead of OVERFLOW, it should print UNDERFLOW

    Hint: You will have to change the Fraction struct to indicate if you are in an overflow/underflow situation. Do not change any of the functions in Sort_List.

- `fract` *print_mode*
  - uses the filter function to only keep fractions and ignore whole numbers when producing the new sorted list; then print the filtered list
  - remember to free the new list produced by filter after printing
    (hint: you had to do that for various commands in q1b as well)
  - Using the input from the append examples above
    For the command "`fract MIXED`", the output should be

    ```
    fract:      Mixed Fractions, Insertion Order
         1.250  1 1/4
         0.667  2/3
    ```

- `whole_num` *print_mode*
  - similar to `fract` except it uses the filter function
    to filter out all fractions leaving only the whole numbers in the new list
  - Using the input from the append examples above
    For the command "`whole_num MIXED`", the output should be

    ```
    whole_num:  Mixed Fractions, Insertion Order
         3.000  3
    ```

- `rem_mixed` *print_mode*
  - similar to `fract` except it uses the filter function to keep only the whole numbers and simple fractions (removes the mixed numbers)
    - i.e. leaving out the numbers that, when printed as MIXED,
      have both a whole number and fraction parts, such as `7 2/3`
  - Using the input from the append examples above
    For the command "`rem_mixed MIXED`", the output should be

    ```
    rem_mixed:  Mixed Fractions, Insertion Order
         3.000  3
         0.667  2/3
    ```