The program should NOT have any errors and memory leaks!

Some useful command:
To check error: gcc -Wall
To check memory leak: valgrind

Question 1a: Advanced Linked Lists

```
typedef struct NODE {
    value_t value;
    key_t key;
    struct NODE * next;
    struct NODE * sort;
} Node;
```

or

Use this one if you can

[*bonus 10% for Q1 – doubly linked lists*]

```
typedef struct NODE {
    value_t value;
    key_t key;
    struct NODE * next;
    struct NODE * sort;
    struct NODE * prev;
    struct NODE * prev_sorted;
} Node;
```

In this linked list:

- The datatype for the value being stored is called value_t
- The datatype for the key being stored is called key_t
- As in lab 4, `next` links to the node in the order it was added to the list (either at the head or the tail)
    - This will be referred to as *insertion order*
- Similar to lab 4, `sort` links to the node where the key is greater or equal to its key
    - i.e. the list is kept in ascending order by key
    - This will be referred to as *key sort order*
    - Note: unlike lab 4, there is only one key

Create a Sorted List abstract data type

- Has two heads (`head` *for insertion order*, `head_sort` *for key sort order*)
- Has two tails (`tail` *for insertion order*, `tail_sort` *for key sort order*)
- *If you are* ***<u>not</u>*** *using the bonus Node struct*
    - Has two "previous tail" (the node pointing to the tail node) links `prev_tail` and `prev_tail_sorted`
- Has an `int` field called size that stored the node count (the number of elements in the list)
- The datatype should be called Sorted_List

Note: technically you will be implementing only be a subset of the Sorted List ADT as you will not be asked to implement all functions of the full ADT

Functions to be implemented

All functions, except where noted, return SUCCESS if the function can complete or FAIL if not

- `int size (Sorted_List *)`
- `int push ( Sorted_List *, value_t , key_t )`
  - add the node to the head of the list
  - the node must also be inserted in **ascending** sort order by key, using the sort link
- `int append ( Sorted_List * , value_t , key_t )`
  - similar to push, except the node gets added to `tail`
- `int remove_first ( Sorted_List * , value_t * , key_t  *)`
  - removes the node from the head of the list
  - returns the value and key of the removed node (and frees the node)
  - remember to update the sort order links
    - if not using doubly linked lists, you will need to find the previous sorted node to change its sort order link
- `int remove_last ( Sorted_List * , value_t * , key_t * )`
  - similar to remove_first, except it removes the node from the tail
- `int remove_smallest_key ( Sorted_List * , value_t * , key_t * )`
  - removes the node with the smallest key
  - returns the value and key of the removed node (and frees the node)
  - remember to update the insertion order links
    - if not using doubly linked lists, you will need to find the previous insert order node to change its insertion order link
- `int remove_largest_key ( Sorted_List * , value_t * , key_t * )`
  - similar to remove_smallest_key, except it removes the node with the largest key
- `void empty_list ( Sorted_List *)`
  - empties the contents of the list
- `void destroy_list ( Sorted_List *)`
  - empties the contents of the list, as well as freeing the list itself

To test the Sorted List ADT

Write two programs called `a4q1a_char.c` *and* `a4q1a_int.c`

- `Data types used`
  - `a4q1a_int.c`
    - has its `value_t` datatype set equal to `int`
    - has it `key_t` datatype set equal to `double`
  - `a4q1a_char.c`
    - has its `value_t` datatype set equal to `char[80]`
      - i.e. it can take strings up to 79 characters in length
    - has its `key_t` datatype set equal to `int`
      - its value is set equal to the length of the string
- Both programs read in a text file that contains a series of commands, one per line (i.e each ending with a newline)
  - The name of the text file should be entered as a command line argument
    - If there is no file name, read from `stdin`
      - this can use IO redirect, i.e. `a4q1a_int < filename.txt`
    - If using keyboard input, exit using `^d`
- All commands are echoed to `stdout`, followed by a colon :,
  - After that the results of the command follows,
    - usually on the same line following 11 – strlen(cmd name) spaces or on the next line when noted

    Note: Silent commands do not have the colon : after the command, but rather after the command name
- Remember to free the sorted list at the end of the program (use `destroy_list`)

General Note: The two programs should be almost identical, with the following differences

- The file input will be slightly different depending on the data type and nature of the input data
- Your will have to write similar, but not identical `void print_list_all ( Sorted_List * )` and `void print_list_sort ( Sorted_List * )` functions
  - These functions print out the lists according to their respective sort orders
  - See the report commands section below for details (the `print_all` and `print_sort` commands)
- You will have to have your make file recompile all files that mention or use value_t and key_t variables or Sort_List structs when compiling the two programs
  - To do this you will need to use condition compilation (see Week1 lecture notes)
  - In specific, use #ifdef CHAR to compile using the char[80] typedef definition of value_t and #ifdef INT to compile using the int typedef definition of value_t
  - E.g. if you stored all your Sort_List ADT functions in a single file called sort_list.c Then for a4q1a_char.c you could have in your make file a command like

    gcc -Wall -ansi -DCHAR -c sort_list.c

List of Commands

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

- a = append
  - `a4q1a_int.c`
    - input line: `a  key  value`
    - example
      - *input*
        ```
        a   3.27   1427
        a   0.94   984
        a   7.21   346
        ```
      - *output* (11 – 1 spaces after the colon)
        ```
        a:          3.27  1427
        a:          0.94  984
        a:          7.21  346
        ```
  - `a4q1a_char.c`
    - input line: `a  value`
    - example
      - *input*
        ```
        a   The sun did not shine.
        a   It was too wet to play.
        a   So we sat in the house
        a   All that cold, cold, wet day.
        ```
        Note: skip the white space between the command 'a' and the input string
      - The key values for the above are 22, 23, 22, 29
      - *output* (11 – 1 spaces after the colon)
        ```
        a:          The sun did not shine.
        a:          It was too wet to play.
        a:          So we sat in the house
        a:          All that cold, cold, wet day.
        ```
- p = push
  - same as a except it pushes instead of appends the key/value pair onto the list

Verbose Commands (modifies the list and then reports to stdout)

- `rem_first` = remove first node of the list by insertion order
  - also prints the element's key-value pair,
    with two spaces between the key and the value
  - Example for `a4q1a_int.c` assuming the first list element key is 2.465 and value is 212
    `rem_first:  2.465  212`
  - Note the two spaces after rem_first:
    - "rem_first" is 9 characters in length,
      so the number of spaces following should be 11 – 9 = 2
- `rem_last` = remove last node of the list by insertion order and print the element's key-value pair
- `rem_small` = remove the node with the smallest key and print the element's key-value pair
- `rem_large` = remove the node with the largest key and print the element's key-value pair
- `empty` = empty the list
  - the output of this command should be
    `empty:  size = 0`

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

- `size` = size of sorted linked list
  - if there are 21 nodes in the list it prints
    size: List size = 21
- `print_all` = print list in insertion order
  - The type of order is printed on the same line as the command
  - The list starts printing on the next line, one element per line
  - Each element is prefaced by 5 spaces, then the key, then 2 spaces, then the value
  - Example using the input from the append examples
    - a4q1a_int.c
      ```
      print_all:  Insertion Order
           3.27  1427
           0.94  984
           7.21  346
      ```
    - a4q1a_char.c
      ```
      print_all:  Insertion Order
           22  The sun did not shine.
           23  It was too wet to play.
           22  So we sat in the house
           29  All that cold, cold, wet day.
      ```

- `print_sort` = print list in key sort order
  - The output is the same as with print_all except the order of the lines are in key sort order and the command line will read Key Sort Order
  - Example using the input from the append examples
    - a4q1a_int.c
      ```
      print_all:  Key Sort Order
           0.94  984
           3.27  1427
           7.21  346
      ```
    - a4q1a_char.c
      ```
      print_all:  Key Sort Order
           22  The sun did not shine.
           22  So we sat in the house
           23  It was too wet to play.
           29  All that cold, cold, wet day.
      ```

Question 1b: List ADT and Function Pointers

- Implement List_Sort * map (List_Sort *, *function pointer*)
  - map only applies to the values, not the sort keys
  - however, make sure that the new list produced has the same key values and links (both next and sort)
- Implement void with (List_Sort *, *function pointer* , int order)
  - however, make sure that the new list produced has the same key values and links (both next and sort)
- Implement value_t reduce (List_Sort *, *reduction function pointer*, int order)
  - Like map, reduce only applies to values, not keys
  - Unlike map reduce takes an extra final parameter: `int order`
    - order can be either SORT_ORDER or INSERT_ORDER
    - you may decide what values these will take as long as they are int values and defined in the appropriate .h file
    - while the order the reduction occurs in does not matter for sum or prod, it could with other reduction functions
- Implement value_t map_reduce (List_Sort *list, *map fn pointer*, *reduce fn pointer*, int order)
  - First applies map then reduce
  - Unlike map all of these functions take an extra final parameter: `int order`
    - order can be either SORT_ORDER or INSERT_ORDER
    - you may decide what values these will take as long as they are int values and defined in the appropriate .h file
- Implement value_t * map_2_array (List_Sort *list1, List_Sort *list2, *function pointer*, int order)
  - map_2_array takes two lists and applies a function (passed in as a function pointer) that takes two values (from the nodes at the same position in their respective lists) and returns a value of type value_t
  - The values are collected in an array with element type value_t in the same order as traversed by the linked list, depending on whether SORT_ORDER or INSERT_ORDER was choisen
  - the function then returns the above array
  - note: unlike map, order of traversal matters with map_2_array as different nodes will be paired together depending on the order
- Implement value_t map_2_reduce(List_Sort *list1, List_Sort* list2, *map fn ptr*, *reduce fn ptr*, int order)
  - First applies map then reduce

- Use either reduce and/or map_reduce to implement
  - void print_list_all (Sorted_List *)
    - Prints the list in insertion order
      - the first line should read “Insertion Order”
      - then the list is printed out, one node per line as `(key, value)`
  - void print_list_sorted (Sorted_List *)
    - Prints the list in key sort order
      - the first line should read “Key Sort Order”
      - the rest should print the same is in print_list_all, except in key sort order
  - value_t sum (Sorted_List * , value_t init)
    - see lecture notes
  - int count (Sorted_List * , *function pointer*)
    - Takes a list and function pointer
    - the function pointer should be passed a function that takes a single parameter of type VALUE and produces either TRUE or FALSE
    - Returns the count of <u>all</u> of the list values that returned TRUE

Question 2: Recursion

- Implement the following functions recursively
  - Count down from n to 0
  - Count up from 0 to 2n by 2
  - nth, nth_sorted
  - [bonus] remove_nth, remove_nth_sorted

- Greatest Common Divisor
  - Read up on the Euclidean Algorithm in Wikipedia
    - First Section: https://en.wikipedia.org/wiki/Euclidean_algorithm
    - Procedure: https://en.wikipedia.org/wiki/Euclidean_algorithm#Procedure
    - Example: https://en.wikipedia.org/wiki/Euclidean_algorithm#Worked_example
    - Using mod: https://en.wikipedia.org/wiki/Euclidean_algorithm#Euclidean_division
    - Implementations: https://en.wikipedia.org/wiki/Euclidean_algorithm#Implementations
  - Use the final implementation, which is recursive and implement it in C using long integer parameters i.e. long gcd(long, long)
  - The Wikipedia recursive implementation (and so your implementation) is naturally in a "tail recursive" form
    - explain why it is tail recursive in the readme
    - make sure your make uses the appropriate gcc flag to run tail recursive code efficiently

Question Fraction ADT

```
typedef struct {
    long num;
    unsigned long denom;
} Fraction;
```

- Implement int fraction(Fraction * fract, int num, int 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)
  - If fract has a numerator of 3 and a denominator of 4 it should print "3/4" to stdout
  - If fract has a numerator of 4 and a denominator of 3 it should print "1 1/3" to stdout
  - If fract has a numerator of 4 and a denominator of 1 it should print "4" to stdout
- Implement simplify fraction using GCD

    - For a / b
        - find g = gcd(a, b)
        - the simplified form of a / b is
          (a/g) / (b/g)
    - e.g. 6/12
        - gcd(9, 12) = 3
        - (9/3) / (12/3) = 3 / 4
- Implement int add_fract(Fraction * result, Fraction * x, Fraction * y)
    - To compute add_fract(a/b, c/d) use the following formula
      (a * d + b * c) / (b * d) simplified
    - Check to make sure that the result doesn't overflow/underflow
      (i.e the addition and/or multiplication produces a result greater than a long can represent)
        - If it overflows return FALSE and do not update fract
        - Otherwise return TRUE
- Implement mult_ fract
    - To compute mult_fract(a/b, c/d) use the following formula
      (a * c) / (b * d) simplified
    - Check to make sure that the result doesn't overflow/underflow
      (i.e the addition and/or multiplication produces a result greater than a long can represent)
        - If it overflows return FALSE and do not update fract
        - Otherwise return TRUE
- Also implement sub_ fract (which subtracts fractions) and div_ fract (which divides fractions) in a similar way
- Implement void rand_fract(Fraction * x, int a, int b), which should store a Fraction with
    - Numerator randomly chosen between [-a, a]
    - Denominator randomly chosen between [1, b]
- Create a function called rfract(int n, int a, int b) that produces an array that stores n random Fraction values created by rand_fraction(a,b)
- Print the fractions in sorted order

Question 1b: List ADT and Function Pointers

Using the same data type Sorted_List as in q1a,

Implement

- **`Sorted_List * map ( Sorted_List *, `*fn ptr*`)`**
  - map only applies to the values, not the sort keys
  - however, make sure that the new list produced has the same key values and links (both next and sort)

- **`value_t reduce ( Sorted_List *, `*reduce fn ptr*`, value_t init, int order )`**
  - like map, reduce only applies to values, not keys
  - however, reduce also takes an extra parameter, `int order`
    - order is either INSERTED_ORDER or SORTED_ORDER and determines which set of links to follow while reducing: `next` or `sort` respectively
    - note: while the order of the reduction does not matter when using add or mult as the reducing function, it could with other reduction functions

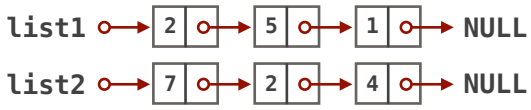
- **`value_t map_reduce (Sorted_List *list, `*map fn ptr*`, `*reduce fn ptr*`, value_t init, int order )`**
  - Conceptually, you are first applying map, and then reduce
  - However, both map_fn and reduce_fn should be applied together, node-by-node
    - i.e. do not create a full map list and then apply reduce to it
    - instead apply the map function to list's node, then store the result in the `reduce` accumulator
  - This allows you to avoid creating and freeing the memory that would be used if an intermediate map list were to have been created and only then reduced

- **`value_t * map_2_array ( Sorted_List *list1, List_Sort *list2, `*fn ptr*`, int order)`**
  - map_2_array takes two lists and applies a function, passed in as a function pointer, that takes two values (from the nodes at the same position in their respective lists) and returns a value of type value_t
  - The values are collected in an array with element type value_t in the same order as traversed along the links chosen (i.e. next if INSERTED_ORDER was chosen or sort if SORTED_ORDER was)
  - the function then returns the above array

    note: unlike map, order of traversal matters with map_2_array as different nodes will be paired together depending on the order

- **`value_t map_2_reduce( Sorted_List *list1, List_Sort *list2, `*map fptr*`, `*reduce fptr*`, int order)`**
  - Similar to map_reduce,
  - However, node by node, it should
    - apply the map function to the value of list1's node as its first argument and the value of list2's node as its second argument
    - then store the result in reduce's accumulator

Below list1 and list2 only depict the value and next fields



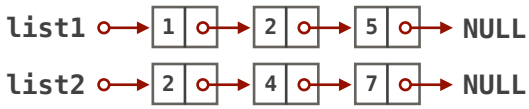

```
array = map_2_array(list1, list2, mult, INSERTED_ORDER)
```

```
ans = map_2_reduce(list1, list2, mult, add, 0, INSERTED_ORDER)
ans == 28  ➡  TRUE
```

where add and mult are functions that take two int arguments
as seen in the reduce example in the lecture notes

Figure 1: Example of map_2array and map_2_reduce using INSERTED_ORDER

list1 and list2 are the same as before,
but now depict the key and sort fields where key was set equal to value




```
array = map_2_array(list1, list2, mult, SORTED_ORDER)
```

```
ans = map_2_reduce(list1, list2, mult, add, 0, SORTED_ORDER)
ans == 45  ➡  TRUE
```

Figure 2: Example of map_2array and map_2_reduce using SORTED_ORDER

To test within, map, reduce, etc.

Write a program called `a4q1b.c`

- `Data types` used
  - `value_t` datatype is equal to `int`
  - `key_t` datatype is equal to `double`
  - same as a4q1a_int.c, so compile files containing Sorted_List functions using -DINT
- The program reads 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 extended to allow for new commands that are detailed below
- Again, all commands are echoed to `stdout`, followed by a colon : in the same format as used with q1
- Include a void print_array(value_t *, int size)
  - this prints out values in the array produced by map_2_array
  - the values in the array should be printed one per line, with five spaces preceding the value
- Make sure you have declared an array that can hold up to 10 Sorted_List pointers
  - Do not confuse this with the array produced by map_2_array, this array holds Sorted_List pointers **not** value_t values.
- Remember to free all sorted lists and nodes (and arrays that may contain them) at the end of the program

To implement the new commands, write the following functions

use either map, reduce, map_reduce, map_2_array, or map_2_reduce when implementing

- **sum**
  - sums the values of a list and returns the sum
  - see lecture notes
- **diff**
  - takes two sorted lists of the same size
    - returns NULL if the sizes are different
  - produces an array whose values are the differences of the values in the sorted lists args
  - you should also take a third argument that can be set to SORTED_ORDER or INSERTED_ORDER in order to follow the appropriate links
- **square**
  - takes a sorted list and produces a new sorted list whose value at a node equals the original value (not key) squared
  - the keys and links should be copied unchanged
- **sum_of_sq_diff**
  - takes two sorted lists of the same size
    - returns NULL if the sizes are different
  - produces a value computed as follows:
    - at each node position, take the difference between the values
    - square the resulting difference
    - sum across all nodes into a single result
  - you should also take a third argument that can be set to SORTED_ORDER or INSERTED_ORDER in order to follow the appropriate links

note: these should be 1 or 2 line programs that take function pointers to functions that are also only a few lines long

List of Commands

All commands from q1a should be made available as well as the following

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

- a|*n*
  - append to the nth index of the array of sorted list pointers
  - same as the a command but appends into relevant sorted list in the array of sorted lists
- p|*n*
  - same as a|*n* except it pushes instead of appends the key/value pair onto the appropriate list

Verbose Commands (modifies the list and then reports to stdout)

For all examples, the sorted list at index 3 holds the values <1, 2, 3> where the key equals the value
the sorted list at index 5 holds the values <3, 1, 7> where the key equals the value

- sum|*n*
  - sums the values of the sorted list at index *n*
  - For the command "sum|3 SORTED_ORDER", the output should be
    ```
    sum:        list = 3, result = 6
    ```
- square|*n*
  - For the command "square|3", the output should be
    ```
    square:     list = 3
         1
         4
         9
    ```
- diff|*n*:*m order*
  - For the command "diff|5:3 INSERTED_ORDER", the output should be
    ```
    diff:       list1 = 5, list2 = 3
         2
         -1
         4
    ```
- sum_sq_d|*n*:*m order*
  - For the command "sum_sq_d |5:3 INSERTED_ORDER", the output should be
    ```
    sum_sq_d:   list = 5, list2 = 3, result = 21
    ```

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

- print_all|*n*
  - print the sorted list at index *n* in insertion order
  - Using the input from the append examples in q1a that had been stored at index 2
    For the command "print_all|2", the output should be
    ```
    print_all:  Insertion Order
         3.27  1427
         0.94  984
         7.21  346
    ```
- print_sort|*n*
  - print the sorted list at index *n* in key sort order