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_type reduce ( Sorted_List *, reduce fn ptr, value_type 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

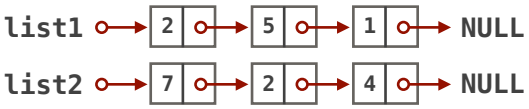
- **`value_type map_reduce ( Sorted_List *list, map fn ptr, reduce fn ptr, value_type init, int order )`**
  - Conceptually, you are first applying map, and then reduce
  - However, both map_fn and reduce_fn should be applied together, node-by-node
    - i.e. do not create a full map list and then apply 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_type * 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_type
  - The values are collected in an array with element type value_type 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_type map_2_reduce( Sorted_List *list1, List_Sort *list2, map fptr, reduce fptr, value_type init, 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



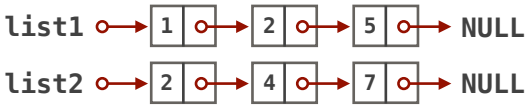

```
array = map_2_array(list1, list2, mult, INSERTED_ORDER)
```


```
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_type` datatype is equal to `int`
  - `key_type` 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_type *, 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_type 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 FAILURE or NULL if the sizes are different (*depending on how you design the function*)
  - 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 a value to indicate failure 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 1: these should be 1 or 2 line programs that take function pointers to functions that are also only a few lines long

note 2: as signatures are not given, you may design the functions any way you choose e.g. what is the return value, what the parameters are, how to get information in and out of the function, etc.

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 key value*
  - append to the nth index of the array of sorted list pointers that you have set up for the program to use (see the 5th point on how to set up the program two pages back)
  - same as the a command but appends the key/value pair into the array of sorted lists
- p|*n key value*
  - same as a|*n* except it pushes instead of appends the key/value pair into the array of sorted lists

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

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

- 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:  list = 2, 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
- sum|*n*
  - sums the values of the sorted list at index *n*
  - For the command "sum|3", the output should be

```
sum:        list = 3, result = 6
```

- square|*n*
  - For the command "square|3", the output should be

```
square:     list = 3
     1.0  1
     2.0  4
     3.0  9
```

  - remember to free any new list produced, after it has been printing
- diff|*n*:*m order*
  - For the command "diff|5:3 INSERTED_ORDER", the output should be

```
diff:       list1 = 5, list2 = 3, Insertion Order
     2
     -1
     4
```

  - remember to free any new list produced, after it has been printing
- 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, Insertion Order, result = 21
```

The assignment continues with Question 2: Recursion