Llrb_map.h:

The LLRB_map, compared to the set header, has an additional Value in the Node struct. The functionality of this header file depended on the new Get function, and the modified Insert and Remove functions. Additional errors serve for situations involving null pointers being returned. Meanwhile the logic and code of the other functions remained the same from the original set header file.

The public Get function gives the end user access to a key's corresponding value, without allowing direct access to the private Node. It calls the private Get function which traverses the tree, and returns the Node's value. Meanwhile the Insert and Remove functions had to be modified to accept Value parameters.

llrb_multimap.h:

The LLRB_multimap's class contents mirrors the LLRB_map class with the exception of the Value object containing a Queue of values, rather than just one. This edit resulted in modifying the Insert and Remove functions. A repeated insertion of a key would result in one node containing multiple keys. A removal involving a Node with multiple keys would result in only popping the Value Queue, while leaving the rest of the Node intact.

map_tester.cc & multimap_tester.cc:

The map_tester.cc tests if the Get retrieves values properly, and whether the Insert, and Remove functions populate/depopulate Nodes properly. Exception handling involves the pre-initialization of the tree and out of range inputs. Hence, std::runtime_errors were used in the scenario a tree was uninitialized and std::out_of_range errors were used for out of range inputs. Using the Get function when a tree was initialized would result in a std::runtime error. Meanwhile std::out_of_range errors would result in using the Get, Insert, Remove functions for keys/values that did not exist.

Cfs_sched.cc:

During the beginning, I threw exceptions whether or not a file can be opened or there are too little arguments. A class task is storing the identifier, start time, virtual runtime, and the number of processes or the duration. I used two predicates in order to sort the vector by start time then identifier. In this order, the identifier will get priority in the same start time. I used two vectors and one multimap since I stored all the tasks in one vector. The other vector will be used to hold the current task. The multimap is used to hold the virtual runtime of the task as the key and the task as the value. The outer while loops check if there are any tasks in the two vectors and multimap.

Inside the while loop, there is a for loop that ticks the amount of running tasks dependent on the start time of the tasks in which tasks from the vector are inserted into the multimap. An if statement is nested inside the for loop checking whether or not if the global min is than 1: if it is, then increment the virtual runtime of the task before insertion into the multimap. Another if statement in the while loop checks if there are no running tasks in order to print a special statement. I check the current tasks vector in order to see if anything is running. If there are running tasks, insert the tasks into the current tasks vector and remove from the multimap. Then, set the global min to the next task's virtual runtime. A special if condition has to be performed if the duration is 0 in order to print the identifier with an asterisk. In addition, remove the task from the original vector and the current tasks vector. Otherwise, insert the task back into the multimap and remove from the current tasks vector.