Priscilla Chan 912 918 594
Laura Oelsner 912 912 846

ECS36C - HW4 Report

For Phase #1, we decided that our tester should do testing based on type (char, int, double, string) and on number of nodes in the tree, as these are the most variable parameters of our trees. We made three trees called *llrb1*, *llrb2*, and *llrb3*, for our tests. *llrb1* has a map of *int* key type and *string* value type, with some number of nodes that are inserted in order. *llrb2* has a map of *char* key type and *double* value type, with a larger number of nodes that are inserted randomly. *llrb3* has a map of *int* key type and *int* value type (to test for same type in both key and value), with only one node.

With these three trees in both *map_tester.cc and multimap_tester.cc*, we tested for the removal of the min, max, and root nodes, along with the insertion of nodes in different parts of the trees. In *multimap_tester.*cc, we also tested for removal of one or more values. In addition, we tested for special cases such as the removal of non-existent nodes (in both testers), and insertion of same-key nodes (in *map_tester.cc*). These cases lead to expected exceptions that throw the appropriate error message. One thing we noted, however, was that while *map_tester.cc* should not allow the insertion of same-key nodes, *multimap_tester.cc* should not be throwing exceptions for this, as the insertion of same-key nodes leads to an additional value in the list of values.

In Phase #2, we decided that our tree should contain the task *vRuntime* times as the key, with the *Task* itself being the value. Therefore, we created a class *Task*, which holds an *id, startTime, duration, runtime,* and *vRuntime.* For sorting purposes, we used the std::list::sort function along with a lambda function to specify the relative order of tasks. We also created a class *CFS*, which is our tree and has functions *AddTask* and *RunTask*.

Our main program starts by checking if the user entered enough arguments and if the data file can be opened successfully. Next, it creates a list of *Task* pointers called *sortedTasks*, which take in each task from the data file and sorts it first by *startTime* then by task ID. Then, while there are still *Task* pointers in the list (yet to be added to the scheduler because their start time hasn't passed) or while *CFS* is not empty (there are still tasks running), we will continue scheduling and running tasks.

To schedule and run tasks, we first have *tick* and *min_vruntime* be initialized to 0. Then as long as the task's *startTime* is greater than or equal to the tick number, we will add the first task to *CFS* and pop the task from *sortedTasks*. Next, if *CFS* is not empty, we use the function *RunTask*, which gets the minimum of the tree *CFS* and prints out the *tick* number, tree size (denoting number of tasks), and task *id.* If there are multiple tasks that are not complete, and this current task has a *vRuntime* less than or equal to *min_vruntime*, we increase the current task's *runtime* and *vRuntime*, along with increasing the *tick* and printing out the appropriate task information to the screen. Then we add the task back into *CFS*. If the task's *duration* is equal to the *runtime*, then it is done, so we will print out an asterisk and not put the task back into the tree. Back in the main program, it will print out 0 tasks along with no *id* if the tree is empty. Then it will increase the tick number before continuing to schedule and run tasks, until the CFS is empty.