CS 202 EFFIECIENCY AND GDB WRITE-UP FOR PROGRAM 3

BY TEJAS MENON; ASSIGNMENT BY PROFESSOR KARLA FANT

Working with overloaded operators this program definitely eased higher level classes during implementation, and the extra code required to implement the operators cleaned out repetitive/clunky areas especially while writing the red black tree. Since insertion was a complex process, debugging and interpreting the control structures for the recursive process was simplified by not having to strcmp() for every keyword comparison. Additionally, implementing the keyword class to contain all linear linked list operations meant that the code size and readability across the all_meetings class was significantly improved while simultaneously being able to call super class functions (for the contact, g_of_contacts and meeting classes) using the same operator.

Another structurally efficient process was having very individualized and concentrated classes- each performing their own specific jobs- for e.g. the keyword class performing LLL operations without requiring the all_meetings class to intervene and therefore functionally being synonymous to a node that has its own LLL processes. A similar functional autonomy could have been provided to a node performing red black operations had I more time, and therefore purge the use of an all_meetings class which in essence would have brought about self similarity between classes and encouraged a more OO styled programming.

As a data structure, using a red black tree had several advantages, from being able to search at O(log(n)) time complexity- since a red black tree is simply isometric to a 2-4 but with its children extended to separate nodes. Therefore, this makes a red black at worst twice the height of a 2-4 but without the sequential access speeds of a linear array. Additionally, similar to a 2-4, the recursive ripple issue of the 2-3 is mitigated significantly by way of the actions of right and left rotations. An alphabetically ordered method of displaying the data is another advantage as compared to other large storage solutions such as hash tables. It's practicality can be extended beyond AVL trees additionally, as it reduces the extent of rotations and frequency of processor demanding actions.

The usage of virtual functions were not required this program since the most derived keyword class was essentially the hub and connecting vertices for the LLL and red black tee. Such an occurrence was due to the absence of different types derived from a common base and each class was rather extended by a unique parent.

I did not need much use of any of the debuggers this program as the data structure assigned was relatively simple, with the pathways of execution being very limited- most of the functions were called at construction and the scope of errors were limited to easily rectifiable syntax errors. I did however use Valgrind to correct a few minor character array deallocation errors, which I could pinpoint due to its detailed line number provisions. On one occasion too, I resorted to using gdb to run through specific functions for incremental implementation, and put together working modules of the code very incrementally. The += operator for the keyword class was also initially a root cause of several errors right across the program, which I discovered later was due to un-freed

memory and incorrectly allocated char pointers. Valgrind is I tool I found especially useful off late with large scale data structures as I could test several implementations of a particular structure and determine the most reliable form of implementation. Also a tactic I found that was time-saving was to note all the different line numbers pertaining to different errors –memory or otherwise- and then make changes on the ones appearing too often before figuring out the rest. This allowed me to prioritize my attention depending on the ‘fatality’ of the error and therefore make continual progress.