Trevor Stanley
5/6/18

Data Structures Final Project

Purpose

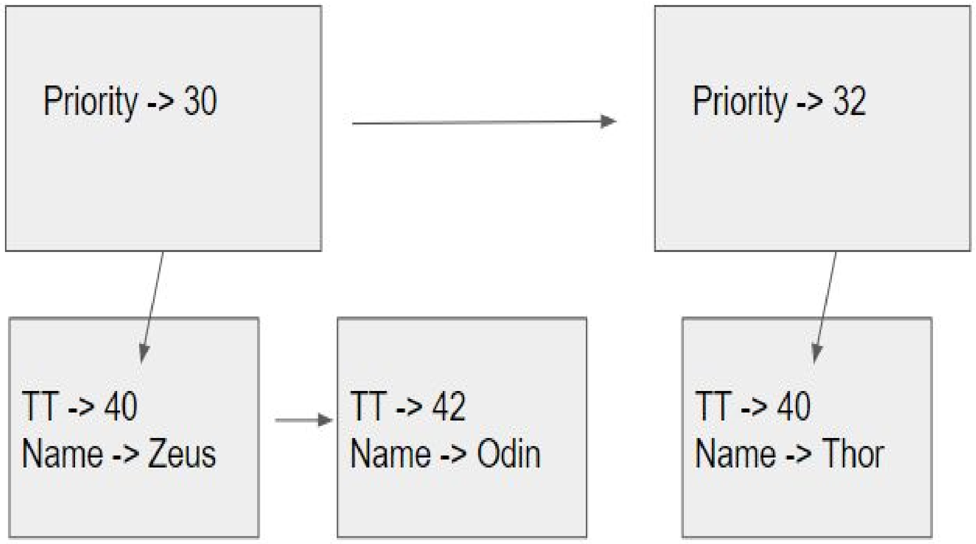
The overarching purpose of this project has been incredibly multifaceted; not only has this project put the lowly programmer through undue stress, but has allowed them to thoroughly demonstrate their knowledge and ability of utilizing linked lists, heaps, and priority queue in the standard library. Through these various implementations, an intimate understanding of the differences between these data structures arose. Further building upon this understanding is an analysis of the different runtimes that these implementations take. This will be further expanded upon in subsequent paragraphs from both a conceptual level (e.g. what is the theoretical big O notation for the particular implementation and subset function) and via analyses of the actual run time for these implementations (e.g. an internal clock in the code and fun mathematical analyses of these results). From a larger picture perspective, this project is actually very cool as computation time of programs is a very important and relevant thing in the real world. Knowing how to optimize code for the most efficient runtime is incredibly useful and saves us time in the end...which is the most precious thing after all!

I. Procedure

Three main data structures were employed in this project: Linked Lists, Heaps, and Priority Queue through the Standard Library. Details about these implementations follow under their respective sections. While many includes are present at the top of each file, not all of these were ultimately used. Chrono was ultimately used for the runtime analysis, which was done in milliseconds and will be described in more detail under the "Data/Runtime Analysis" section. Big O notation will be covered there as well.

-Linked Lists

Perhaps my favorite data structure, the linked list implementation had a higher degree of flexibility in its deployment than the others. Following the advice of grand meister archwizard arbiter Prashil, the great, I pursued a method that used a linked list within a linked list. Put simply, and as visualized in the figure to the right, there are essentially two "layers". The first layer is merely a linked list of priority times, while the second layer contains names and treatment times. Firstly, within main the CSV file is passed to the



function “setup”. The file is opened, the first line discarded via a getline, and then a while loop gets each line while for the entire file. These lines are then parsed using stringstream, the values stored in variable placeholders, and then these are passed to the “addNode” function. AddNode will check if mainHead (a public variable in the class definition) is null; if it is null, then a new node is created and assigned the priority time that was passed in. Simultaneously, a new node is also created for the layer 2 linked list and the name and treatment time are assigned. Each node from layer one has a “head2” pointer, initialized to null, that acts is accessed for the creation of the second layer. If it is null, as is the case when “mainHead” is null, it is assigned the name and treatment time (TT). Thus, a second layer linked list will always exist if a node with the priority value exists. In cases where neither heads are null, it is first checked if the priority time of the incoming node is equal to that of any existing node in the first layer of the LL. If so, then “insertInSecondary” is called; this essentially iterates through the existing layer 2 LL. If the treatment time passed into the function is less than an existing node, it will be inserted before that node, otherwise, it will be inserted at the end. Finally, there is a “deleteN” function that takes in an integer value. While the integer is greater than 0, the second layer list will be iterated through and print out the name and treatment time until empty. Once the pointer to the next node of the head of the second layer LL is null, the “head2” pointer is deleted and the next node of the first layer LL is moved on to.

-MinHeap

Similar to the linked list implementation, the minHeap implementation begins almost identically with the CSV file passed to the “setup” function. Within setup, the data is parsed in the same way, stored in their respective variables (name, prior, and treat), and these are passed to the “push” function. Push will add nodes at the end of the heap array and then use swap to “heapify” the array into the proper order based on if the “child” is smaller or larger than the “parent”. The comparison utilizes the overloaded operator which compares each element of the node (e.g. treatment time and priority time), returning true or false based on the relation we are looking for (more details on this in comments within code/hpp file). The “print” function then prints out the first element of the array, replaces this with the last element, decrements the total size of the array, and resorts the array back to proper or with the “printSort” function. Essentially, the printSort function does the opposite of the push function as it pushes the new element at the front of the array “down”, whereas the push function pushed the smallest element “up”. Please see comments in code for more details on these functions.

-STL Priority Queue

Utilizing the STL priority queue was much easier compared with the other implementations. Similar to the minHeap implementation, and overloaded operator was used. Specifically, when the instantiation of “pq” as a priority_queue of type patient (the struct node containing name, priority time, and treatment time), “compare” is invoked such that the proper “heapified” order is maintained when pushing or poping in a similar fashion to the minHeap implementation, just with more functions going on “under the hood”. Nearly identical code is used for extracting data from the CSV and parsing the data.

Data

The data is stored in a CSV file. The first line is the title/header for the respective columns of data: name, priority time, and treatment time. Priority time is what is prioritized as this is the time until birth. Treatment time is less important as it is not the time until birth but rather the amount of time to treat someone (perhaps the time to give birth). If two women's priority times are equal, then the one with the lower treatment time will take precedence in the ordering of the list/heap array. The file is opened, and getline is used to get rid of the first line of the file as we do not need/want this information. A while loop is then used to get lines from the file until the lines end. Stringstream is then used on the given line in a file to parse the line based on commas and stored this into variables. Priority and treatment time are initially strings, and as such, stoi must be used on them to convert to integers.

Results/Runtime Analysis

High level explanations of the implementations were discussed above, and further details can be found in comments within the code. Regarding the runtime of different implementations, the below graphs show that the Linked List build took the longest. This makes sense conceptually, as the larger number of items being processed would mean more nodes to search through before performing insertion or deletion. Linked lists have a worst case scenario of O(n) for both accessing and searching, with O(1) for insertion or deletion. Deletion/dequeuing the linked list was much faster as deletion/dequeuing was always happening from the front of the lists. This contrasts with the minHeap implementation, which has a worst case of O(n(log(n))). Regarding the STL priority queue, the top() function by which printing is done is O(1), while insertion and removal is the same as minHeap, or O(n(log(n))). From my runtime analysis, it STL building and dequeuing were slightly slower than my minHeap.