COMP281 Assignment 3

# 1046 – Sum of Adjacent Numbers

First input is obtained for the gridSize and the number of Adjacent values to check. An integer double pointer variable named ‘grid’ is created through the ‘insMatrix’ function, using malloc and for loops. The function ‘occMatrix’ is called which obtains input from the user, populating the grid. An integer variable named ‘fN’ has it’s value set to zero, then the function ‘gScan’ is called. This scanning function uses three for loops which uses the created grid’s rows, columns, and the user’s offset. An if statement uses the function pointer ‘checkP’ to check which function to use, using fN as the index. fN values from zero to four call xCheck, yCheck, dCheck and eCheck respectively. These check functions make sure that the current value of the offset does not breach the grid’s limits. When the value can be offset safely, a function pointer known as sumP calls the correct function which returns the required number. This is added to a ‘temp’ variable. If temp is higher than the ‘output’ variable, output obtains temp’s value, then temp is reset to 0. Otherwise temp is simply reset. The gScan function is called another three times, changing the value of fN before each iteration so the validation and addition functions change, but the structure of the function remains. The output variable is sent to console once the fourth gScan is complete.

# 1048 – Date Sort ‘n’ Search

The size of the date array is obtained first. insLog creates, instantiates the variables inside each element then returns the finished array. The function getInput stores the user input into the array. A function called procsDate (spoken as ‘process date’) generates a numerical timestamp of the current date element’s values using many switch case statements and is stored in the dateNum variable in the date struct. Qsort is applied to the date array, known in the program as ‘log’; the compare function checks the difference between two timestamps and returns a value which makes the log variable in reverse chronological order. Another date array which stores the search term (known as ‘sTrm’) goes through the same process of being created, obtaining the input and generating a timestamp. Bsearch is then called, using sTrm and log, with the same compare function as was used in qsort. Finally a for loop iterated n (known in the program as ‘total’) plus one times – outputting the sorted date list then outputting yes or no dependant on the result from the bsearch.

# 1049 – Priority Queue

This program does not work, which I believe may be due to a realloc that fails, making the pointer null so the data becomes unrecoverable. Despite this, the theory behind my solution to the problem was to implement a binary heap, making the efficiency ‘O (N Log(n) ).’ First a queue array known as ‘list’ is created. String based input is then accepted – if it is “pop” at this point, then -1 is outputted and an integer variable used as a kill switch is set. If the command was “Insert”, then the addRecord function is called, which increments an integer variable to store the total size of the queue then stores user input at the end of the array. From this point, a while loop using the kill switch variable (known as ‘looper’) as a parameter is ran. This loop scans for a command, then calls addRecord or the function known as ‘popQ.’ popQ checks if the total var. is the same as one. If so, print -1, set the kill switch and then exit the function. Otherwise a for loop goes through the array backwards and calls the srtQ function. srtQ finds the parent of the current element through the use of modulo 2. If the element is odd, indicating it is the second child of a parent node, it checks if the first child (element - 1) is larger. If so, the switchValues function is called to swap the values over, so that the largest child element is always on the left. It then checks if the priority of the child is higher than the parent. If so, switchValues is called to remedy this. srtQ is then called using the parent, so that the function moves up the tree and sorts everything. Once sorted, popQ decrements the total variable, outputs the value stored in index one of the queue array (index zero is left intentionally blank to allow better looking arithmetic with sorting), the data stored in the last index is copied over to index one and finally the array is reallocated, cutting what was the last element out of the array.

# 1050 – Flight Planner

First the total number of cities and the number of connections are obtained. Two connection arrays are then created – one for the bulk of the user input and one for the search terms. Input is then obtained for both arrays using the function filList. srtList is then called, utilising qsort to place the array in ascending order, and the search term’s values are flipped if the second is higher than the first. The chkList function is then called. A for loop finds the first element which contains the same first value as the search term which is stored in the ‘temp’ variable, then another for loop to find when the values are not the same, which is stored in the ‘nextVal’ variable. This provides a range for the rest of the function to work with. A third for loop runs from the nextVal variable, going backwards through the array, and checks if there is a match anywhere with the search term. If so, it sets the result variable to 1, indicating there is a route to the destination. If not, then the first value in the searchTerm is changed to the list[nextVal]’s first value and the function becomes recursive, using the modified values from it’s previous iteration.