Assignment 3 report

PQ.h

Typename ID: just the type of input

currentSize: current size of PQ

capacity: current max # of elements allowed in PQ, can be extended later

BH: the binary heap part of PQ

AVL: the Avl Tree part of PQ

Functions

printPQ():

- print the content inside the heap and tree

isEmpty():

- check if currentSize is 0
- return true if it is, else return false

deleteMin():

- if PQ is empty, throw underflowexpection{}
- else call the private version of the function
    - BH.extractMin() returns a reference to the actual min_ID stored in BH, I created a pointer that points to what min_ID is referring to so I do not get a copy of the actual min_ID
    - Reconstruct the AVL tree
    - Minus 1 on current size
    - Return what min_ID_ptr is pointing to, which is the actual min_ID

findMin():

- if PQ is empty, throw underflowexpection{}
- else call the private version of the function
    - simply return the element in the 0^{th} index of taskID_list

insert():

- call the private version
    - insert ID and priority into BH, and call makeheap()
    - construct the AVL tree with the newly added ID & priority
    - +1 on currentSize

updatePriority():

- Call the private version
    - Get the taskID_list array from BH.h to check if x is in that array already
    - If it isn't, call the insert_private() function to add it
    - If it is, update it in priority_array
    - Call makeheap();

Size():

- Return currentSize

makeEmpty():

- Call both makeEmpty functions in BinaryHeap.h & AvlTree.h, which erases all data in heap and tree
- Make currentSize = 0

BinaryHeap.h & AvlTree.h

Since we were told that "we could do whatever we want", I decided to simply not bother with making a **vector<avlnode*> pointer_array** as getting avlnode* working was too painful. (we are literally trying to access something we are not supposed to access) Instead, I came up with the system I have, and I will try my best to explain my design.

BinaryHeap.h (my own design based on professor's pseudocode)

currentSize: current size of the heap

capacity: max # of elements the heap can hold

priority_array: the array that stores each ID's priority

taskID_list: the array that stores all IDs

BinaryHeap.h functions:

isEmpty():

- See if currentSize == 0

makeEmpty():

- Call the private version
    - Set currentSize = 0

makeHeap():

- Call the private version
    - Do percolate down starting from the last non-leaf element in priority_array, which is located at floor(currentSize/2)-1, and visit every non-leaf element until root
    - If it's any of its child is smaller than it, swap their positions in both taskID_list & priority_array

Insert():

- Call the private version
    - If arrays are full extend them
    - Insert priority p & task y at the end of priority_array and taskID_list
    - Do percolate up on priority_array, swapping the newly added priority with its parents if it is smaller, do the same for IDs in taskID_list too
    - Update v = p & p = next parent
    - Increase currentSize
    - Return the last inserted index (will be used later)

extractMin_private():

- if PQ is empty, throw underflowexpection{}
- if not empty, call the private version
    - save the 0^{th} element in both priority_array & taskID_list as I need them later
    - reduce currentSize by 1 first
    - replace 0^{th} element in both priority_array & taskID_list with the last element
    - store "temp"s in a garbage spot that cannot be accessed
    - do percolate down like the one in makeheap()
    - return the ID that is stored in the garbage spot

printArrays():

- call the private version
    - simply loop through the two arrays and print their content

get_priorityArray() & get_taskID() :

- return a reference that refers to the actual arrays

AvlTree.h (mostly author's code with some modifications)

- AvlTree's insert order is based on ID's ascii values if IDs are chars, or simply ints if IDs are ints

Task_ID: the ID stored in each node

Heap_index: where the node's ID is stored in heap

AvlTree.h functions:

printTree():

- print the ID & heap_index of each node in tree following an in-order variant

displayTree():

- same function I wrote in assignment 2, except it doesn't display addresses

Whenever something new is inserted in PQ, I first add it to BH. BH's insert function will do percolate up on both the priority_array & taskID_list. By the end of the BH.insert(x,p), both arrays should be perfect, and I make a new tree based on the new arrays, removing the need of a pointer array.

AvlTree.h (author's code but slightly modified)

task_ID: ID of the task

heap_index: where ID is located in the heap