**Data Structures and Algorithms II**

**Fall 2020**

Student name: Allister Liu

Student e-mail: liu14@cooper.edu

Assignment: Program #2

Grade: 85

Comments:

You mentioned the change of .h to .hpp in useHeap.cpp; that is fine, of course!

The program works for the provided test case. However, for one of my other test cases, there is incorrect output, and for my largest test case, the program crashes.

I tracked down the incorrect output first. In heap::insert, you should compare Heapsize to capacity, not to capacity-1. This is because the valid indexes range from 1 to capacity. I am fixing that. -5 points

There is still incorrect output for this test case, but it starts later than before, so there is at least one other bug.

I decided to debug the crash next. I ran that case in the debugger. It crashes in deleteMin at this line:

data[1] = data[this->Heapsize--];

At the time of the crash, Heapsize is -2. (This occurs after tens of thousands of commands, near the middle of the test case.)

I found a subtle bug in heap::remove. When you move the item from Heapsize to replace what pn points to, you need to call setPointer to update the pointer for id. You may assume this will get fixed in percolateUp or percolateDown, but if tempkey and newKey are identical, neither gets called! -5 points

I fixed that, and there is still a crash, but it occurs later in the test case now (the same line listed above though). And again, Heapsize is -2 (this might be after the – has been applied). I think that somehow, a previous call to heap::remove must be incorrectly having contains return true, even though the heap is empty.

I think I see a subtle bug in hash::insert. When the key is a duplicate, you should NOT update the void pointer. I commented that out, but it does not change the behavior. (I don't see how it would cause a problem for this program.) I'm not taking off points for this, because it is a reasonable interpretation, but when the insert does not happen due to a duplicate, the void pointer should NOT be updated.

I tracked down an even more subtle bug; the cause of the bug has to do with how different hash functions interact, and that you try to undelete deleted items if they are reinserted (which is fine if you get it just right). -5 points

The bug is in this code:

hashIndex = hash(key);

while(data[hashIndex].isOccupied)

{

hashIndex++;

hashIndex %= capacity;

if(data[hashIndex].key == key){

break;

}

}

If hash returns the location of a deleted item, you go into the loop and increase hashIndex before doing the check, so you miss it. You can then wind up with two versions of the same key; one deleted and one not. But, when you search for the key with contains (or, more generally, findPos), you don't find the non-deleted version, you find the deleted version and return -1 from findPos. I fixed this by moving the check for the key in the code above to BEFORE hashIndex++. After this fix, the program works for all my tests!