CSCE 221 - Programming Assignment 4 Report **(20 points)**

Due April 14, 2021

First Name: Clayton Last Name: Kristiansen UIN: 328003173 User Name: kristiansenc E-mail address: kristiansenc@tamu.edu

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more in the Aggie Honor System Office http://aggiehonor.tamu.edu/

Type of sources					
People	none				
Web pages (provide URL)	none				
Printed material	none				
Other Sources	none				

I certify that I have listed all the sources that I used to develop the solutions/code to the submitted work.

"*On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work.*"

Your Name (signature) Clayton Kristiansen Date 04-14-2021

1. The description of the assignment problem.

   The assignment was to create 3 different implemenations of a Minimum Priority Queue. These implementations would then be tested for functionality, and evaluated in terms of efficiency.

2. The description of data structures and algorithms used to solve the problem.

   (a) Provide definitions of data structures by using Abstract Data Types (ADTs)

      i. The UnsortedMPQ is a vector based class inheriting from the provided MPQ class. It is a Minimum Priority Queue that provides functionality for setting items aside and recieving them in order of priority.
      ii. The SortedMPQ is a linked list based class inheriting from the provided MPQ class. It is a Minimum Priority Queue that provides functionality for setting items aside and recieving them in order of priority.
      iii. The BinaryHeapMPQ is a vector tree based class inheriting from the provided MPQ class. It is a Minimum Priority Queue that provides functionality for setting items aside and recieving them in order of priority.

   (b) Write about the ADTs implementation in C++ (for all the three MPQs).

      i. The UnsortedMPQ used a vector to store the items in the queue. When insert was called, the item was just inserted at the end. When the min or remove min functions were called, the methods had to iterate through the vector until the minimum value was found.
      ii. The SortedMPQ used a linked list to store the items in the queue. Items were in sorted from highest priority to lowest priority as they were intserted and returned instantly from the front of the list when min or remove min was called.
      iii. The BinaryHeapMPQ used a vector to store the items in the queue. Items were sorted in the binary heap convention as they were inserted to increase efficiency. The vector was arranged such that the first element is always the highest priority.

   (c) Describe algorithms used to solve the problem. For every MPQ (`UnsortedMPQ`, `SortedMPQ` and `BinaryHeapMPQ`), list the MPQ functions (`remove_min()`, `is_empty()`, `min()`, and `insert()`) and provide their descriptions.

      Every MPQ has the same external functionality of: checking if it is empty, inserting an object, returning the highest priority object, and removing the highest priority object.

      i. The UnsortedMPQ
         A. is_empty() simply checks if the size is equal to zero and returns true if so, false if not.
         B. insert() simply calls the push_back method of the vector member to add an object.
         C. min() employs an O(n) search algorithm to find the minimum value, then returns that value.
         D. remove_min() does the same as min(), but also removes the min from the vector.
      ii. The SortedMPQ
         A. is_empty() is the same as above and simply checks if the size is equal to zero and returns true if so, false if not.
         B. insert() uses an iterative algorithm if O(n) to insert each object in the linked list in sorted priority order.
         C. min() returns the first object in the linked list.
         D. remove_min() does the same as min(), but also removes the min from the list.
      iii. The BinaryHeapMPQ
         A. is_empty() is again the same and simply checks if the size is equal to zero and returns true if so, false if not.

B. insert() will insert the new object at the end of the vector member. Then, a function called up_heap() on the new node that continually compares the current node to it's parent. If the current node is less than the parent, they are swapped, and the check begins again for the new parent.

C. min() returns the object at the front of the vector.

D. remove_min() does the same as min(), but also removes the min from the vector. It also moves the node at the end of vector to the front and calls a function called down_heap() on it that will continually swap that "node" with whatever child is the most less than itself until both of its children are greater than itself.

(d) Show the time complexity analysis for the following. Time complexity analysis means providing a **basic runtime function/recurrence relation**, **solution for recurrence relation with steps (wherever needed)** and a **Big-O** Notation:

i. **Best, worst, and average case** of each of the MPQ functions (`remove_min(), is_empty(), min(), and insert()`) for `UnsortedMPQ.` (Note: Some functions may have same runtimes for all the cases. In that case, write the answer only once and mention that the runtime applies to all the cases.).

A. **remove_min()**

Best case (must always iterate through all elements to determine min)

$$runtime : n + 4 \tag{1}$$

$$O(n) \tag{2}$$

Worst case (same as best)

Average case (same as best)

B. **is_empty()**

Best case (simply checks equality of two items

$$runtime : 2 \tag{3}$$

$$O(1) \tag{4}$$

Worst case (same as best)

Average case (same as best)

C. **min()**

Best case (same exact function as remove_min() simply without the removal (which was O(1)))

$$runtime : n + 1 \tag{5}$$

$$O(n) \tag{6}$$

Worst case (same as best)

Average case (same as best)

D. **insert()**

Best case (simply calls push_back() on the vector which is O(1))

$$runtime : 3 \tag{7}$$

$$O(1) \tag{8}$$

Worst case (same as best)

Average case (same as best)

A. Provide an **example for best, worst, and average case** for `UnsortedMPQ`.

In terms of best/average/worst case insertion or removal for UnsortedMPQ, there is no different between input sets. The insert is always O(1) and the min functions must always search through every element to gurantee it is the minumum.

ii. **Best, worst, and average case** of each of the MPQ functions (`remove_min(), is_empty(), min(), and insert()`) for `SortedMPQ.` (Note: Some functions may have same runtimes for all the cases. In that case, write the answer only once and mention that the runtime applies to all the cases).

A. **remove_min()**

Best case (only needs to remove the first element)

$$runtime : 3 \tag{9}$$

$$O(1) \tag{10}$$

Worst case (same as best)

Average case (same as best)

B. **is_empty()**

Best case (simply checks equality of two items

$$runtime : 2 \tag{11}$$

$$O(1) \tag{12}$$

Worst case (same as best)

Average case (same as best)

C. **min()**

Best case (same exact function as remove_min() simply without the removal (which was O(1)))

$$runtime : 2 \tag{13}$$

$$O(1) \tag{14}$$

Worst case (same as best)

Average case (same as best)

D. **insert()**

Best case (element to be inserted is smaller than the first element)

$$runtime : 3 \tag{15}$$

$$O(1) \tag{16}$$

Worst case (element to be inserted is greater than all elements currently in the MPQ)

$$runtime : n + 4 \tag{17}$$

$$O(n) \tag{18}$$

Average case (element to be inserted is the average off all the elements in the MPQ)

$$runtime : n/2 + 4 \tag{19}$$

$$O(n) \tag{20}$$

A. Provide an **example for best, worst, and average case** for `SortedMPQ`.

Best case (for insert): {4, 5, 6, 7, 8} insert 3
Worst case (for insert): {4, 5, 6, 7, 8} insert 9
Average case (for insert): {4, 5, 6, 8, 9, 10} insert 7

iii. **Best, worst, and average case** of each of the MPQ functions (`remove_min(), is_empty(), min(), and insert()`) for `BinaryHeapMPQ.` (Note: Some functions may have same runtimes for all the cases. In that case, write the answer only once and mention that the runtime applies to all the cases).

A. **remove_min()**

Best case (the heap has only one element)

$$runtime : 2 \tag{21}$$

$$O(1) \tag{22}$$

Worst case (rightmost element is largest element)

The number of comparisons required to delete a node in the a heap is proportional to the length of the path the swapped item travels as it moves down. Since the binary tree is dense, this path length is O(log(n)) with n nodes. Therefore, the runtime efficiency delete is O(log(n)). I also implemented this function iteratively.

$$runtime : 3 * log(n) + 4 \tag{23}$$

$$O(log(n)) \tag{24}$$

Average case (rightmost element is average of all elements)

$$runtime : 3 * log(n/2) + 4 \tag{25}$$

$$O(log(n)) \tag{26}$$

B. **is_empty()**

Best case (simply checks equality of two items

$$runtime : 2 \tag{27}$$

$$O(1) \tag{28}$$

Worst case (same as best)

Average case (same as best)

C. **min()**

Best case (same exact function as remove_min() simply without the removal (which was O(1)))

$$runtime : 2 \tag{29}$$

$$O(1) \tag{30}$$

Worst case (same as remove_min())

$$runtime : 3 * log(n) + 4 \tag{31}$$

$$O(log(n)) \tag{32}$$

Average case (same as remove_min())

$$runtime : 3 * log(n/2) + 4 \tag{33}$$

$$O(log(n)) \tag{34}$$

D. **insert()**

Best case (element to be inserted is greater than all other elements)

$$runtime : 3 \tag{35}$$

$$O(1) \tag{36}$$

The number of comparisons required to insert a node in the a heap is proportional to the length of the path the inserted item travels as it moves up. Since the binary tree is dense, this path length is O(log(n)) with n nodes. Therefore, the runtime efficiency delete is O(log(n)). I also implemented this function iteratively.

Worst case (element to be inserted is less than than all elements currently in the MPQ)

$$runtime : 4 * log(n) + 5 \tag{37}$$

$$O(log(n)) \tag{38}$$

Average case (element to be inserted is the average off all the elements in the MPQ)

$$runtime : 4 * log(n/2) + 5 \tag{39}$$

$$O(log(n)) \tag{40}$$

A. Provide an **example for best, worst, and average case** for `BinaryHeapMPQ`.

Best case (for insert): {4, 5, 6, 7, 8} insert 9
Worst case (for insert): {4, 5, 6, 7, 8} insert 3
Average case (for insert): {4, 5, 6, 8, 9, 10} insert 7
Best case (for min): {4}
Worst case (for min): {4, 5, 6, 7, 8}
Average case (for min): {4, 5, 6, 8, 9, 7}

3. A C++ organization and implementation of the problem solution

   (a) Provide a list and description of classes or interfaces used by a program such as classes used to implement the data structures or exceptions.

      i. MPQ - Minimum priority queue, ADT that allows for storage, and access, of items with accordance to their priority.

      ii. BinaryHeap - A heap structure based off a vector that treats the vector like a binary tree where each parent must be ¿ their child.

      iii. UnsortedMPQ - MPQ implemented with vector.

      iv. SortedMPQ - MPQ implemented with linked list.

      v. BinaryHeapMPQ - MPQ implemented with binary heap.

      vi. RuntimeException - Exception thrown when an error is encountered during runtime.

   (b) Provide features of the C++ programming paradigms like Inheritance or Polymorphism in case of object oriented programming, or Templates in the case of generic programming used in your implementation.

      i. Inheritance - All the MPQ classes in this project are inherited from the MPQ class

      ii. Templates - Every class and helper function is templatized to allow each MPQ to handle different types of (already comparable) data.

4. A user guide description how to navigate your program with the instructions how to:

   (a) compile the program: specify the directory and file names, etc.

      Open a Linux terminal to the directory containing the makefile. Type "make all" to compile the program. This creates the "test" file

   (b) run the program: specify the name of an executable file.

      Type "make run" to run the program. This runs the file "test". You may also type "./test"

5. Specifications and description of input and output formats and files

   (a) The type of files: keyboard, text files, etc (if applicable).

   All input and output files are .txt files. User input is done through the keyboard.

   (b) A file input format: when a program requires a sequence of input items, specify the number of items per line or a line termination. Provide a sample of a required input format.

   The .txt files must be formatted with 3 integers (one CPU task) per line. The integers are ID, Length, and Priority. They are to be seperated by spaces with a newline at the end.

   Ex:

   6 2 15
   3 8 3
   2 10 -16
   8 3 -3
   4 7 -16
   9 9 -3

   (c) Discuss possible cases when your program could crash because of incorrect input (a wrong file name, strings instead of a number, or such cases when the program expects 10 items to read and it finds only 9.)

   The program is quite robust, if it encounters an unexpected length it will always read as far as possible before completing its task (it won't crash). It wil throw an exception, however, if operations are attempted to be performed on an empty MPQ, or if an input file does not exist.

6. Provide types of exceptions and their purpose in your program (Answer only to the ones that are applicable for this assignment).

   (a) logical exceptions (such as deletion of an item from an empty container, etc.).

   Logical exceptions are thrown when a requested input file does not exist, or when an empty MPQ is operated on with either min() or remove_min().

   (b) runtime exception (such as division by 0, etc.)

   These exeptions are not currently explicitly handled by the program.

7. Include evidence of your testing by providing screenshots. Screenshots should show execution of the 5 main methods `(unsortedmpq-main.cpp, sortedmpq-main.cpp, main.cpp, cpu-job-main.cpp, binaryheap-mpq-main.cpp).`

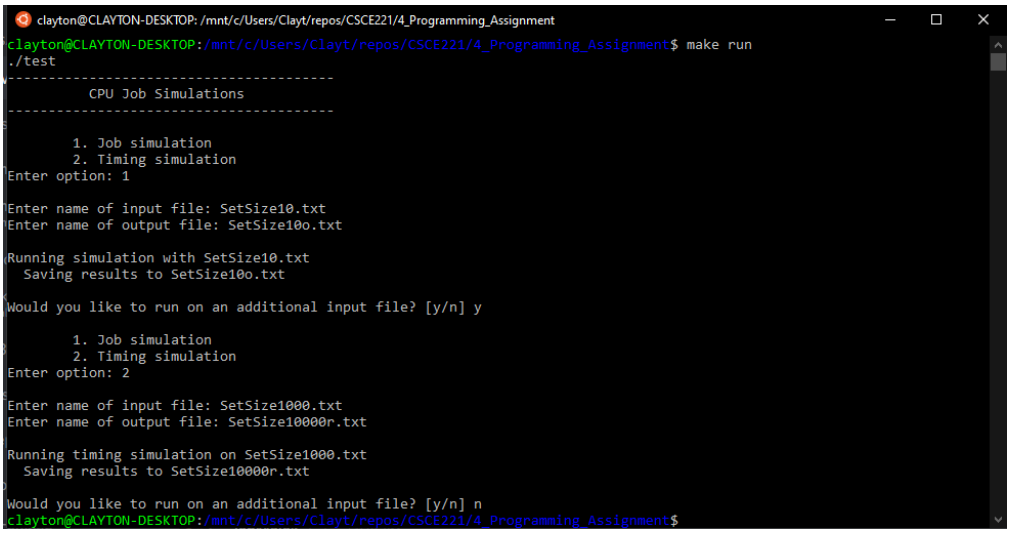

Figure 1: CPU Tests

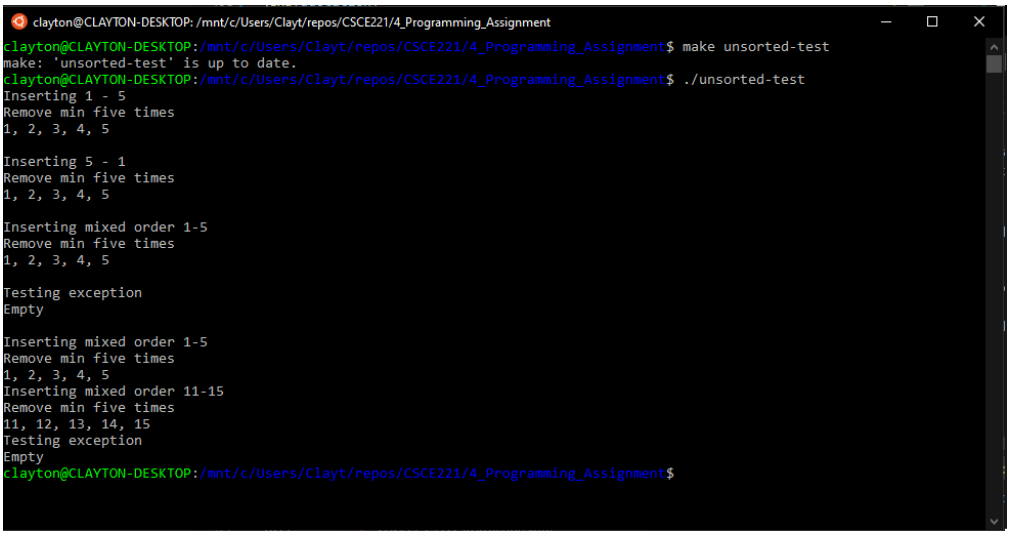

Figure 2: Unsorted Tests

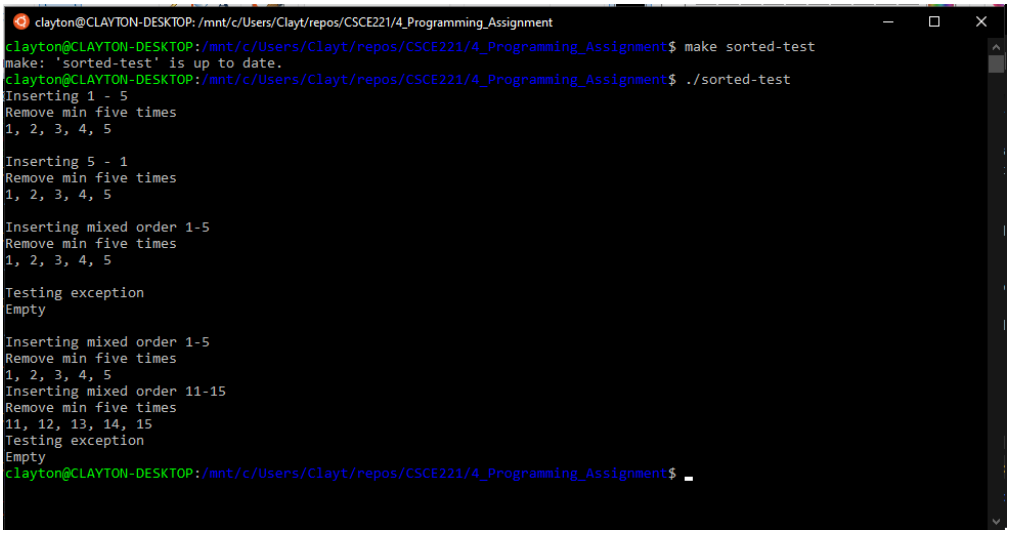

Figure 3: Sorted Tests

Figure 4: BinaryHeap tests

8. Provide graphs and data tables of your CPU timing simulation results. Graph should be plotted for **runtime vs. input size**. The input sizes are 4, 10, 100, and 1,000. To obtain this data, compile and run `main.cpp`. Choose option "2. Timing Simulation". Provide the input file name (SetSize4.txt) and output filename. After execution, you will find the output file in "OutputFiles" folder. The timing for all the three MPQ implementations will be displayed. Fill it in the following table and plot it as a **graph**.

	Runtime		
Input Sizes	Unsorted MPQ	Sorted MPQ	Binary heap MPQ
4 (SetSize4.txt)	0.0094	0.0092	0.0082
10 (SetSize10.txt)	0.0467	0.0102	0.0257
100 (SetSize100.txt)	0.3477	0.1953	0.1092
1000 (SetSize1000.txt)	17.1037	10.6245	1.0095

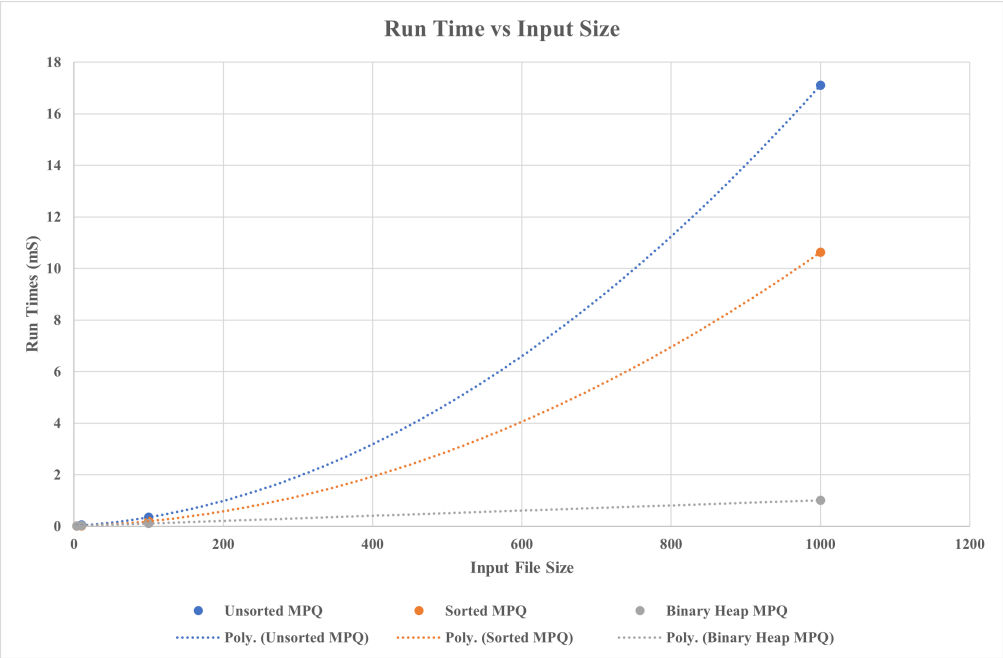

Figure 5: CPU timing simulation results graph