CSCE 221 Assignment 5 Cover Page

First Name　　　　　　Last Name　　　　　　UIN

User Name　　　　　　E-mail address

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 on Aggie Honor System Office website: `http://aggiehonor.tamu.edu/`

Type of sources				
People				
Web pages (provide URL)				
Printed material				
Other Sources				

I certify that I have listed all the sources that I used to develop the solutions/codes 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　　　　　　Date

CSCE 221 – Prelab for the Final Project, Fall 2013 (100 points)

Due: April 8, 2014 at 11:59 pm

Hardcopy Report: Return to your TA in lab

Problem description (80 points)

1. Implement minimum priority queue (MPQ), once as an unsorted/sorted array or a linked list and the other time as a minimum binary heap.

   (a) For example, given a source city and other cities connected by flights, each city has a corresponding distance milage from the source city. We want to find the city with minimum distance.

   (b) A table of city-distance data

City	City ID	Distance
SFO	0	2467
LAX	1	3288
ORD	2	621
DFW	3	1423
JFK	4	84
BOS	5	371
PVD	6	328
BWI	7	0
MIA	8	946

   (c) An illustration of unsorted array MPQ:

MPQ array index	(Distance, City ID)	MPQ array element address
0	(0, 7)	0x000010
1	(2467, 0)	0x000030
2	(3288, 1)	0x000050
3	(621, 2)	0x000070
4	(1423, 3)	0x000090
5	(84, 4)	0x0000b0
6	(371, 5)	0x0000d0
7	(328, 6)	0x0000f0
8	(946, 8)	0x000110

2. Implement both data structures with locator pattern. A locator serves as a re-direction linker to an entry in the main data structure.

   (a) Assume each data entry has a corresponding ID, and users have a need to locate a data entry in the data sturcture by its ID in an efficient way.

      i. For example, with the above MPQ, we are able to find the city with minimum distance, but we also have a need to update distances efficiently when the flight stops between the cities are changed. That is, given a city ID, we want to access and update the (distance, city ID) entry directly without going through the whole MPQ array or linked list and check each city ID.

   (b) A locator object typically stores the address or the array index of a data entry which is then utilized to access the data entry directly. The design of a locator highly depends on the main data structure. A locator class should be defined along with the main data structure.

      i. For example, the locator object corresponding to the city 0 stores MPQ array index '1' or the MPQ item address '0x000030'

(c) **When a data entry is created and stored in the main data structure, a corresponding locator object should be created.**

(d) We may create another search structure for locator addresses to enable quick access to locators. The search structure maps a ID to a locator address. The structure can be a random-access array, hash table or search tree.

i. Because the city IDs are contiguous positive integers, we can just create a random-access array with city ID as index and locator address as element. The locator search array looks like:

Locator search array index (i.e. City ID)	Locator address
0	0x606030
1	0x606050
2	0x606070
3	0x606090
4	0x6060b0
5	0x6060d0
6	0x6060f0
7	0x606010
8	0x606110

ii. Corresponding locators which store MPQ array index:

0x606010	0x606030	0x606050	0x606070	0x606090
0	1	2	3	4
0x6060b0	**0x6060d0**	**0x6060f0**	**0x606110**	
5	6	7	8	

iii. The process of locating a data entry by ID:

A. Find the locator address by ID.

B. Access the locator and retrieve the the address or array index stored.

C. Use the address or array index in the locator to access data entry in the main data structure.

(e) The main data structure should be re-defined to incorporate pointers to locators. The reason is that **data entries can be moved around inside the main data structure. We need to update the content of corresponding locators so that the locators remain valid.**

i. The re-defined MPQ unsorted array to store a locator address along with each data entry:

MPQ array index	(Distance, City ID, Locator address)
0	(0, 7, 0x606010)
1	(2467, 0, 0x606030)
2	(3288, 1, 0x606050)
3	(621, 2, 0x606070)
4	(1423, 3, 0x606090)
5	(84, 4, 0x6060b0)
6	(371, 5, 0x6060d0)
7	(328, 6, 0x6060f0)
8	(946, 8, 0x606110)

ii. Corresponding locators which store MPQ array index:

0x606010	0x606030	0x606050	0x606070	0x606090
0	1	2	3	4
0x6060b0	**0x6060d0**	**0x6060f0**	**0x606110**	
5	6	7	8	

iii. The process of updating a locator:

A. When a data entry is moved around inside the main data structure, retrieve the locator address stored along with that data entry

B. Access the corresponding locator by address

C. Update the content of the locator, which is the data entry address or the array index, to the latest value

(f) If data are stored in a vector, locator objects should not store memory addresses for those data, as vector elements' addresses are changed when the vector expands.

3. Consider the following interface for minimum priority queue:

   - `insertItem(k,x,&loc)` inserts a key `k` and element `x;` The address of a locator `loc` is passed and updated after insertion.
     NOTE: First an empty locator should be created. The address of locator is then passed to the function and also stored in the locator search array.
   - `min()` returns the locator to a MPQ item (key and element) with the smallest key (but it does not remove it)
   - `remove(loc)` removes the item with locator `loc` and updates the minimum priority queue in no more than $O(logn)$ time
     NOTE: When this function is called, the locator object should be destroyed and the locator address should be erased from the search structure.
   - `isEmpty()` tests whether the priority queue is empty
   - `replaceKey(loc, k)` updates the item pointed by locator `loc` with key `k`. This should be done in no more than $O(logn)$ time
   - `createPriorityQueue(kArray, xArray, locAddrArr, n)` builds a minimum priority queue from input arrays of size n which include keys, elements and locator addresses. This should be done in $O(n)$ time
   - `getKey(loc)`returns the key pointed by locator `loc`
   - `getElem(loc)`returns the element pointed by locator `loc`

   HINT: For every moving of an array element in the MPQ, make sure you update the corresponding locator of the element.

4. To estimate complexity, count the number of comparisons occurred on the MPQ elements in the `remove(loc)` and `replaceKey(loc, k)` functions, for both implementations. For heap, make sure you count the number of comparisons when the structure is adjusted, such as walk-up or walk-down.

5. In the test program `Main.cpp`, complete the following operations:

   (a) Creates a MPQ and a locator search structure

   (b) Populate the data structure with city-distance data using the function `createPriorityQueue`. You do not need to follow the element order in the example.

      i. Print the data structure on the screen
      ii. Print the locator search structure on the screen and the pointed (key, element) values

   (c) Performs a sort on the cities based on distance using the MPQ functions

      i. Print the (key, element) on the screen in sorted order
      ii. Print number of comparisons at each operation on the screen if applicable

   (d) Populate the data structure again using the function `insert` and the city-distance data

      i. Print the data structure on the screen
      ii. Print the locator search structure on the screen and the pointed (key, element) values

   (e) Ask user which city to replace distance. After updating the data structure,

      i. Print number of comparisons on the screen
      ii. Print the data structure on the screen

iii. Print the locator search structure on the screen and the pointed (key, element) values

Report (20 points)

In addition to the regular programming assignment, your report should include answers to the following problems:

- Describe the algorithms of Priority-Queue ADT for the both implementations
- Discuss about number of comparisons for `remove(loc)` and `decreaseKey(loc, k)`for the both implementations
- Provide running time complexity of Priority-Queue ADT for the both representations (array/linked list and binary heap) express in terms of the big-O notation.
- Compare the running time complexity for the same operations in both implementations of Priority-Queue ADT.
- Does using the locator pattern have an impact on the complexity of the priority queue operations for the both representations?
- Write about three real-life applications where you can use a minimum priority queue.