CSC3002 (Fall 2022) Assignment 5

Problem 1

Exercise 14.8:

In the queue abstraction presented in this chapter, new items are always added at the end of the queue and wait their turn in line. For some programming applications, it is useful to extend the simple queue abstraction into a **priority queue**, in which the order of the items is determined by a numeric priority value. When an item is enqueued in a priority queue, it is inserted in the list ahead of any lower priority items. If two items in a queue have the same priority, they are processed in the standard first-in/first-out order.

Using the linked-list implementation of queues as a model, design and implement a **pqueue.h** interface that exports a class called **PriorityQueue**, which exports the same methods as the traditional **Queue** class with the exception of the **enqueue** method, which now takes an additional argument, as follows:

void enqueue(ValueType value, double priority);

The parameter **value** is the same as for the traditional versions of **enqueue**; the priority argument is a numeric value representing the `priority`. As in conventional English usage, smaller integers correspond to higher priorities, so that priority 1 comes before priority 2, and so forth.

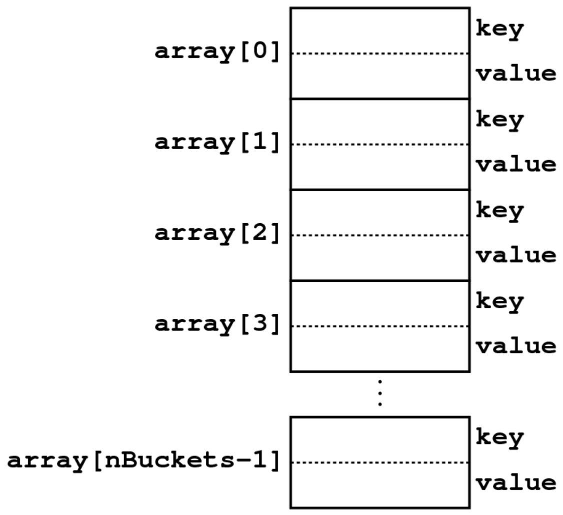
Requirments & Hints:

Please fill in the **TODO** part of **enqueue**, **dequeue** and **peek** functions in *pqueue.h*. You can define your own functions in the codes if necessary, but you need to follow the provided code framework.

Problem 2

1. **Exercise 15.9:**

   Although the bucket-chaining approach described in the text works well in practice, other strategies exist for resolving collisions in hash tables. In the early days of computing when memories were small enough that the cost of introducing extra pointers was taken seriously hash tables often used a more memory-efficient strategy called **open addressing**, in which the key-value pairs are stored directly in the array, like this:



For example, if a key hashes to bucket #2, the open-addressing strategy tries to put that key and its value directly into the entry at **array**[**2**].

The problem with this approach is that **array**[**3**] may already be assigned to another key that hashes to the same bucket. The simplest approach to dealing with collisions of this sort is to store each new key in the first free cell at or after its expected hash position. Thus, if a key hashes to bucket #2, the **put** and **get** functions first try to find or insert that key in **array**[**2**]. If that entry is filled with a different key, however, these functions move on to try **array**[**3**], continuing the process until they find an empty entry or an entry with a matching key. As in the ring-buffer implementation of queues in Chapter 14 of Textbook, if the index advances past the end of the array, it should wrap around back to the beginning. This strategy for resolving collisions is called **linear probing.**

Reimplement the **StringMap** class so that it uses open addressing with linear probing. For this exercise, your implementation should simply signal an error if the client tries to add a key to a hash table that is already full.

Requirments & Hints:

Please fill in the **TODO** part of **findKey** and **insertKey** functions in *stringmap.cpp*. You can define your own functions in the codes if necessary, but you need to follow the provided code framework.

2. **Exercise 15.10:**

   Extend your solution to Problem 3.1 so that it expands the array dynamically. Your implementation should keep track of the load factor for the hash table and perform a rehashing operation if the load factor exceeds the limit indicated by a constant defined as follows:

static const double REHASH_THRESHOLD = 0.7;

In this exercise, you will need to rebuild the entire table because the bucket numbers for the keys change when you assign a new value to **nBuckets**.

Requirments & Hints:

Please fill in the **TODO** part of **rehash** function in *stringmap.cpp*. You can define your own functions in the codes if necessary, but you need to follow the provided code framework.

Problem 3

Exercise 18-3:

Eliminate the recursion from the implementation of **depthFirstSearch** by using a stack to store the unexplored nodes. At the beginning of the algorithm, you simply push the starting node on the stack. Then, until the stack is empty, you repeat the following operations:

1. Pop the topmost node from the stack.
2. Visit that node.
3. Push its neighbors on the stack

Exercise 18.4:

Take your solution from the preceding exercise and replace the stack with a queue. Describe the traversal order implemented by the resulting code.

Requirements for Assignment

You should write **TODO** part in each .h or .cpp file according to the problem requirements.

Please note that, the teaching assistant may ask you to explain the meaning of your program, to ensure that the codes are indeed written by yourself. Please also note that we may check whether your program is too similar to your fellow students' code using BB.

Reminder: Do not wait to submit until the last minute.