ALGORITHMS AND DATA STRUCTURE – HOMEWORK 3

1. Objective

A) Implement the dictionary ADT of the following dictionary based on stacks. Your implementation should declare and use two stacks.
B) Implement the previous dictionary ADT based on queues. Your implementation should declare and use two queues.

The ADT for a simple dictionary:

```
// The Dictionary abstract class.
template
class Dictionary {
private:
void operator =(const Dictionary&) {}
Dictionary(const Dictionary&) {}
public:
Dictionary() {} // Default constructor
virtual ˜Dictionary() {} // Base destructor
// Reinitialize dictionary
virtual void clear() = 0;
// Insert a record
// k: The key for the record being inserted.
// e: The record being inserted.
virtual void insert(const Key& k, const E& e) = 0;
// Remove and return a record.
// k: The key of the record to be removed.
// Return: A maching record. If multiple records match
// "k", remove an arbitrary one. Return NULL if no record
// with key "k" exists.
virtual E remove(const Key& k) = 0;
// Remove and return an arbitrary record from dictionary.
// Return: The record removed, or NULL if none exists.
virtual E removeAny() = 0;
// Return: A record matching "k" (NULL if none exists).
// If multiple records match, return an arbitrary one.
// k: The key of the record to find
virtual E find(const Key& k) const = 0;
// Return the number of records in the dictionary.
virtual int size() = 0;
};
```

2. Analysis

- In this section, an ADT dictionary is the main goal outcome.

- The ADT dictionary will include some main functions such as insert, delete and find.

- The “stack” implementation must be applied first and then “queue” implementation.

- The use of “template” and “class” could help to pass the variable (element) readily.

- Student can use the example template to expand the idea.

- The basic idea is to create each function for each mission, in which user can import the elements and pass it to the “dictionary”.

3. Result Codes.

These codes can be found at this link:

4. Experience and description

- I have got stuck with these terms (template and class) for a long time because I did not get the idea from your example ADT.

- After completing the midterm, I had some idea to do this task when I felt more confident in implementing ADT functions.

- I have read and done many researches about template and class (also learn from my classmate Mr. An). This results in I got the basic structure of “template class”.

- The member functions in the main “template class” must be widen by associating with “template”

```
+ template<typename Key, typename E>

int MyDict<Key, E>::size(){

  cout << "Size of original stack: " << OriginalStack.size() <<
endl;

  return 0;

}
```

- The view for 2 “stack” or “queue” is to help us easier in “remove” and “find” function. In more detail, when user want to remove an expected element (Key) the program needs to find whether this element exists or not. This leads to comparing the “top” or “front” with the expected “Key”, these one would be “pop” if the condition is not met. Therefore, the “CopiedStack” or “CopiedQueue” will store these “pop” elements. Whenever the “Key” found, the “pop” elements will be “push” again into the original one. (1)

- At the “private” (access specifier), two stacks (or queue) will be assigned:

  + *stack <pair<Key, E>> OriginalStack;*

    stack <pair<Key, E>> CopiedStack;

- Because “stack” and “queue” just store the same element type, so I needed to use “pair” to keep “Key” and “E” together.

- At the insert function, we just need to push the new element to the “end” or the “top” of the list

  + *OriginalStack.push(make_pair(k, e));*

- To remove by “Key”, this “Key” would be found first and then being “pop”. The operation was be described like (1).

```
+ int length = OriginalStack.size();
    int lengthCop = 0;
    for(int i = 0; i < length; ++i){
      if(OriginalStack.top().first == k){
        OriginalStack.pop();
        cout << "Delete completed!!!" << endl;


        for(int j = 0; j < lengthCop; ++j){
          OriginalStack.push(make_pair(CopiedStack.top().first,
CopiedStack.top().second));


          CopiedStack.pop();
        }
        break;
      } else {
        CopiedStack.push(make_pair(OriginalStack.top().first,
OriginalStack.top().second));
        OriginalStack.pop();
        lengthCop++;
      }
```

- Similarly, the rest functions will be operated by this logic.

- *In queue*, with the different mechanism of operation “FIFO”, the element in copied one will not be push into the original one after the “key” found. Now, the rest elements in the OriginalQueue will be push and pop into the Copied until empty. Finally, the program will “swap” the content of both.

```
+ OriginalQueue.swap(CopiedQueue);
```