Ngoc Nguyen NITK17K - 1700062

Homework 3

Question

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.

Github link

https://github.com/NgocNguyenGit/Algorithms-and-Data-Structures

Objectives

The objective of the Dictionary ADT provided in the question is to store and retrieve data, here I assumed two types of data- name/city name (string type) and ID/postcode (integer type). From my own analysis, the implementations based on stack and queue support the most in how the Dictionary insert and remove data. Therefore, this homework will only focus on these two operations, excluding the finding records.

The first implementation is based on stacks, in which the program is supposed to perform the **push** (insert) and **pop** (remove) operations from only one end. It follows the "LIFO" rule, which stands for "Last-In, First-Out" where the stacks remove elements in reverse order of their arrival.

In this project, I named the top elements as **front**, which can be accessible and are **pushed** onto the stack. On the other hand, an element is **popped** from the stack in order to be removed.

```
void Dictionary::push(string key, int element)
{
    rear++;
    this->key[rear] = key;
    this->element[rear] = element;
}

void Dictionary::pop()
{
    cout << "Name: " << this->key[rear] <<" – ID: "<<this->element[rear]<<" popped." << endl;
    rear--;
}
```

*Figure 1. **push** and **pop** functions*

In this Homework, I simply implemented the Dictionary ADT with no input from the user. There are two stacks were used in the Homework, which are **key** and **element**. Regarding multiple stacks implementation, it is supposed to utilize the one-way growth of the array-based stack than linked stack.

The second implementation should be based on queues and in this Homework, there are two queues **key** and **element** were used. It is quite similar to stack as a list-like structure. The rule of this implementation is the queue elements may only be inserted at the back by **enqueue** operation and removed from the front by **dequeue** operation. This means queue is a "FIFO" list, which stands for "First-In, First-Out."

```
void Dictionary::enqueue(string key, int element){
    rear++;
    this->key[rear] = key;
    this->element[rear] = element;
}
void Dictionary::dequeue(){
    front++;
    cout << "City name: " << this->key[front] <<" – Postcode: "<<this->element[front]<< "
        popped." << endl;
}
```

*Figure 2. **enqueue** and **dequeue** functions*

Figure 3 is an example screen of main function and the result in the interface.

```
11
12  int main(int argc, const char * argv[]) {
13      cout<<"Stack implementation:"<<endl;
14      Dictionary s;
15      s.push("Marisa Bradshaw",119745);
16      s.push("Zak Ortega", 313592);
17      s.push("Elisa Guerrero", 933665);
18      s.push("Danika Smyth", 158972);
19      s.push("Shanae Guest", 234521);
20      s.push("Morris Farmer", 734133);
21      s.pop();
22      s.pop();
23      s.pop();
24      cout<<endl;
25      cout<<"Queue implementation:"<<endl;
26      Dictionary q;
27      q.enqueue("Helsinki", 10200);
28      q.enqueue("Kokkola", 67100);
29      q.enqueue("Joensuu", 80100);
30      q.enqueue("Oulu", 90100);
31      q.enqueue("Rovaniemi", 99100);
32      q.enqueue("Kemi", 95200);
33      q.dequeue();
34      q.dequeue();
35      q.dequeue();
36
37      return 0;
```

```
Stack implementation:
Name: Morris Farmer – ID: 734133 popped.
Name: Shanae Guest – ID: 234521 popped.
Name: Danika Smyth – ID: 158972 popped.

Queue implementation:
City name: Helsinki – Postcode: 10200 popped.
City name: Kokkola – Postcode: 67100 popped.
City name: Joensuu – Postcode: 80100 popped.
Program ended with exit code: 0
```

Figure 3. Output of result