CSC3002 2023 spring Assignment 5

Due 23:59, April 23, 2023

Problem 1 (Exercise 14.8, Points: 25)

Problem Description

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 class called `PriorityQueue`, which exports the same methods as the traditional `Queue` class [1] 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.

Requirement Please complete the **TODO** parts in the `enqueue`, `dequeue` and `peek` functions.

In & Out Statement

Your program receives two elements in one line, split by the space: a `string` value to be put into the `PriorityQueue` and the corresponding priority of this element. The priority can be any value that fit the type `double`.You are recommended to test this program using an input file.

- To input a text file:

  ```
  p1.exe < in/p1.txt // for win user
  ./p1 < in/p1.txt // for Mac user
  ```

Functionalities of your implementation will be tested in the `main()` function, and you may see the file *out/p1.txt* for standard output.

About P1

- In all OJ test cases, neither the values nor the priorities will include the space character since it is regarded as a marker of input split.

[1] `Queue` in Stanford C++ library, which is introduced in the lectures.

Problem 2 (Exercise 14.13, Points: 25)

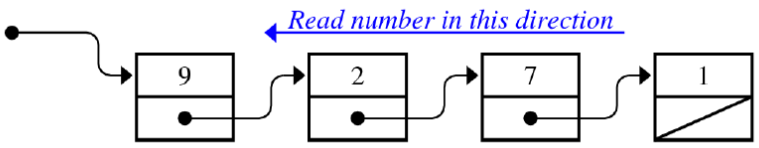
Problem Description

On newer machines, the data type `long` is stored using 64 bits, which means that the largest positive value of type `long` is 9, 223, 372, 036, 854, 775, 807 or 2^{63}˘1. While this number seems enormous, there are applications that require even larger integers. For example, if you were asked to compute the number of possible arrangements for a deck of 52 cards, you would need to calculate 52!, which works out to be

80658175170943878571660636856403766975289505440883277824000000000000

If you are solving problems involving integer values on this scale (which come up often in cryptography, for example), you need a software package that provides ***extended-precision arithmetic***, in which integers are represented in a form that allows them to grow dynamically.

Although there are more efficient techniques for doing so, one strategy for implementing extended-precision arithmetic is to store the individual digits in a **linked list**. In such representations, it is conventional—mostly because doing so makes the arithmetic operators easier to implement—to arrange the list so that the units digit comes first, followed by the tens digit, then the hundreds digit, and so on. Thus, to represent the number 1729 as a linked list, you would arrange the cells in the following order:



Design and implement a class called `BigInt` that uses this representation to implement extended-precision arithmetic, at least for nonnegative values. At a minimum, your `BigInt` class should support the following operations:

- A constructor that creates a `BigInt` from an `int` or from a string of digits. The constructor should prompt "**ERROR: BigInt - illegal format**" if a non-digit character is detected from the input string.
- A destructor that releases the memory.
- A `toString` method that converts a `BigInt` to a string.
- The operators `+` and `*` for addition and multiplication, respectively.

You can implement the arithmetic operators by simulating what you do if you perform these calculations by hand. Addition, for example, requires you to keep track of the carries from one digit position to the next. Multiplication is trickier, but is still straightforward to implement if you find the right recursive decomposition.

Requirement Please complete the aforementioned functions by filling the **TODO** parts in the file *p2bigint.cpp*. Note that this program has no corresponding header file and all the contents are defied in this *.cpp* file. You may add some self-defined functions even inside the `class` section.

In & Out Statement

In this assignment, your program should work when the input and the integers represented by `BidInt` are all nonnegative integer values, while it is not required under negative integer cases. The input of this program is two integers.

- To input a text file:

```
p2.exe < in/p2.txt // for win user
./p2 < in/p2.txt // for Mac user
```

- To input manually:

```
p2.exe // for win user
./p2 // for Mac user
```

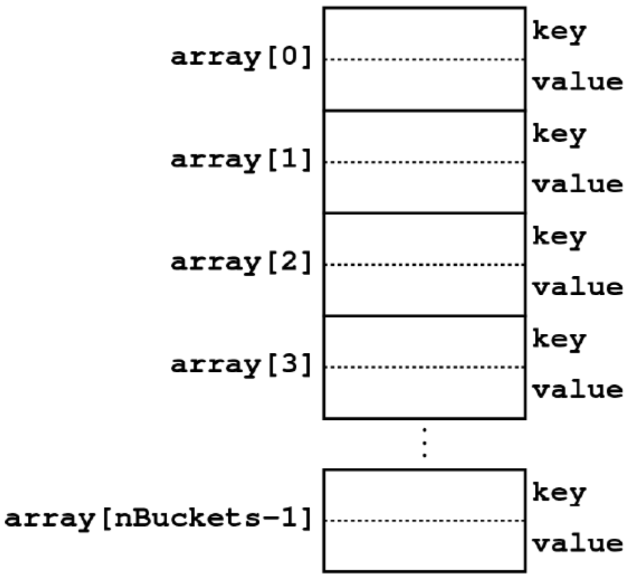
The addition and multiplication will be tested in the `main()` part. Please check the file *out/p2.txt* for the standard output format.

Problem 3 (Points: 50)

Part 1 (Exercise 15.9, Points: 25)

Problem Description

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, 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.

Requirement Please complete the **TODO** parts in the `insertKey` and `findKey` functions of the file *p3stringmap.cpp*. Note that your function should prompt the message "**ERROR: Insufficient space in hash table**" if the space is not enough to insert.

Part 2 (Exercise 15.10, Points: 25)

Problem Description

Extend your solution to exercise 9 so that it expands the array dynamically whenever the load factor exceeds the constant `REHASH_THRESHOLD`, as defined in exercise 5. As in that 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.

Requirement Please complete the **TODO** parts in the `rehash` functions of the file *p3stringmap.cpp*.

In & Out Statement

The input of this program is a set of commands. You may check the functions `helpCommand` and `executeCommand` for detailed functionalities. You are recommended to test this program using an input file. Files *in/p3_1.txt* and *in/p3_2.txt* are provided as the standard input files.

- To input a text file:

```
p3.exe < in/p3_x.txt // for win user
./p3 < in/p3_x.txt // for Mac user
```

You may check the file *out/p3_1.txt* and *out/p3_2.txt* for standard output file.

About this assignment

- You don't need to modify or submit any other files other than `p1PriorityQueue.cpp`, `p2bigint.cpp`, and `p3stringmap.cpp`.
- Submission to OJ platform is required, while submission to Blackboard again is not necessary.
- You can only view your pre-test scores and such scores do NOT equal to your final score. Pre-test cases take up 10-30% cases of the final test (also known as formal test) cases in this assignment. The formal test will be conducted after the assignment. Your final score will ONLY depend on the codes in your latest submission.
- If you are late, 5 points will be deducted from each missing problem immediately after 00:00, and 5 points more every hour afterward until no more points can be deducted.