CSC 3002 (Fall 2021) **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

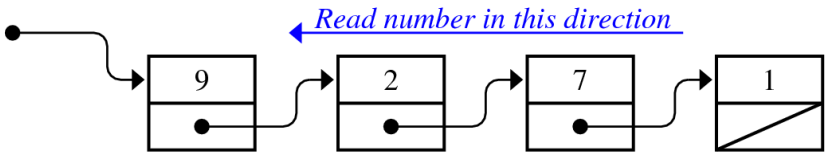
Exercise 14.13:

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¡ª¡ªmostly because doing so makes the arithmetic operators easier to implement¡ª¡ªto 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.
- 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.

Use your **BigInt** class to generate a table showing the value of $n!$ for all values of n between 0 to 52, inclusive.

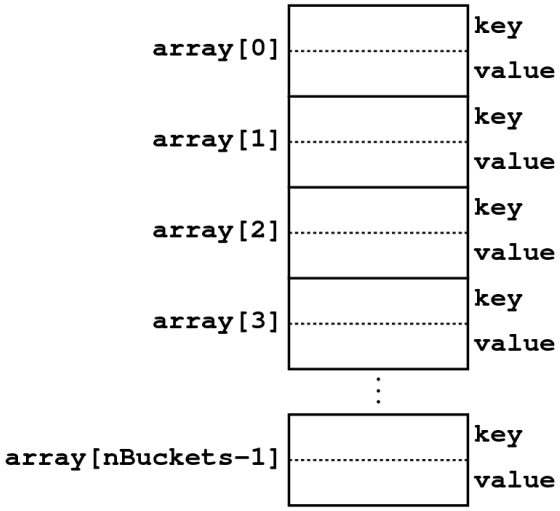
Requirments & Hints:

Please fill in the **TODO** part of **BigInt**, **~BigInt**, **toString**, **operator+** and **operator*** functions in *bigint.cpp*. You can define your own functions in the codes if necessary, but you need to follow the provided code framework.

Problem 3

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¡ªwhen memories were small enough that the cost of introducing extra pointers was taken seriously¡ªhash 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.

Requirements for Assignment

I've provided a project named as *Assignment5.pro*. You should write **TODO** part in each .h or .cpp file according to the problem requirements. The test file is provided under *src* folder. Please pack your whole project files into a single .zip file, name it using your student ID (e.g. if your student ID is 123456, hereby the file should be named as 123456. zip), and then submit the .zip file via BB system.

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.

Please refer to the BB system for the assignment deadline. For each day of late submission, you will obtain late penalty in the assignment marks. If you submit more than 3 days later than the deadline, you will receive 0 in this assignment.

Marking scheme:

- 25% Marks will be given to students who have submitted the program on time.
- 25% Marks will be given to students who wrote the program that meet all the requirements of the questions
- 25% Marks will be given to students who programs that can be compiled without errors.
- 25% Marks will be given to students whose programs produce the correct output if their programs can be compiled.

Reminder: For windows users, please switch you input language to English before interacting in Stanford console. Or, you will get no response.