COP 4530 - Team Assignments

The purpose of this assignment is two-fold. I want to give you experience working with data structures that we did not have time to cover in class. I also want to give you an experience working as a part of a software development team.

Let us both agree from the start: teams are horrible. You could get this work done much easier / quicker / better if I allowed you to just work on it by yourself. However, sadly, that is not the way that the real world works. Instead, we work on teams and it is up to us to draw the best qualities out of the people that we have been thrown together with.

Your team are going to be special. There will be three roles that will have to be assumed by different members of your team. Dr. Anderson does not care who plays what role, just that all of you agree to take on one of the three different roles.

The roles are as follows:

Requirements Generator: This person will not be doing any programming; however, they do have to get their work done before anyone else can start. The requirements generator will read over the task that the team has been assigned. This person is then responsible for creating requirements that will be implemented by the developers. What makes this task challenging is that we are going to assume that the requirements generator does not know how to write Python code. This means that the requirements are going to have to be at a higher level than code. What will the inputs look like? What will the output look like? How does the data have to be processed? These are the types of questions that the team's requirements have to answer.

2 Developers: The developers are the people who will actually write the code for the team. These people will take the requirements and attempt to implement them. If something is unclear or wrong, they will go back to the requirements generator and ask them to make a change. If they have a great idea that impacts the requirements, they will go back to the requirements generator and ask them to make a change. In the end, the code that is produced should do no more or no less than was requested by the team's requirements. This development team will have to agree on how to divide up the programming task.

Tester: The tester is going to be responsible for testing the team's code and making sure that it works in all situations. This means that they can get started once the requirements generator is done with their work. Based on the requirements they can determine what inputs to use, what outputs are expected. When they have the code they should try to overload it, starve it, and give it bad data to see what happens. When the code does not behave correctly, they can go back to the developer and request that changes get made. They can ask the developer to provide them with a special testing API if so needed.

Note: If your team decides that they only need one developer, you can then have either two requirements generators or two testers. However, what each of these people did as part of the project is going to have to be clearly spelled out.

Each team will turn in both the code, input data, and a slide presentation. We will be double checking to make sure that your code works. The slide presentation will be presented in class. The team will have 15 minutes to present their results. Each member of the team will discuss what they did as a part of the project and how they interacted with the other members of their team. How the program works can also be discussed if time allows. Your grade will be based on your team's in-class presentation.

Team Colorado

Linked queue

Requirements:

In this project, you will implement one class:

1. A linked queue class: `Linked_queue`.

Runtime:

The run time of each member function is specified in parentheses at the end of the description.

Class Specifications:

`Linked_queue`

Description

A class which implements a linked queue (in fact, two stacks) which has the specified behaviors. For run-time requirements, the number of elements in the queue is *n*.

Member Variables

The class has four member variables:

- A linked list of pointers to arrays of `Type`: `Single_list<Type *> array_list` (or whatever linked list you created in Project 1),
- An integer storing the queue size, and
- Two integers `ifront` and `itail`.

Each array in the linked list will have a capacity of eight. When the queue is empty, the linked list is empty. As objects are placed into the queue, they are placed into the arrays that will be stored in the linked list. As necessary, additional arrays are added to the linked list. For example, suppose we begin with an empty queue and then push six times and pop once. The result will appear as shown in Figure 1 where the member variables `ifront = 1` and `iback = 5`.

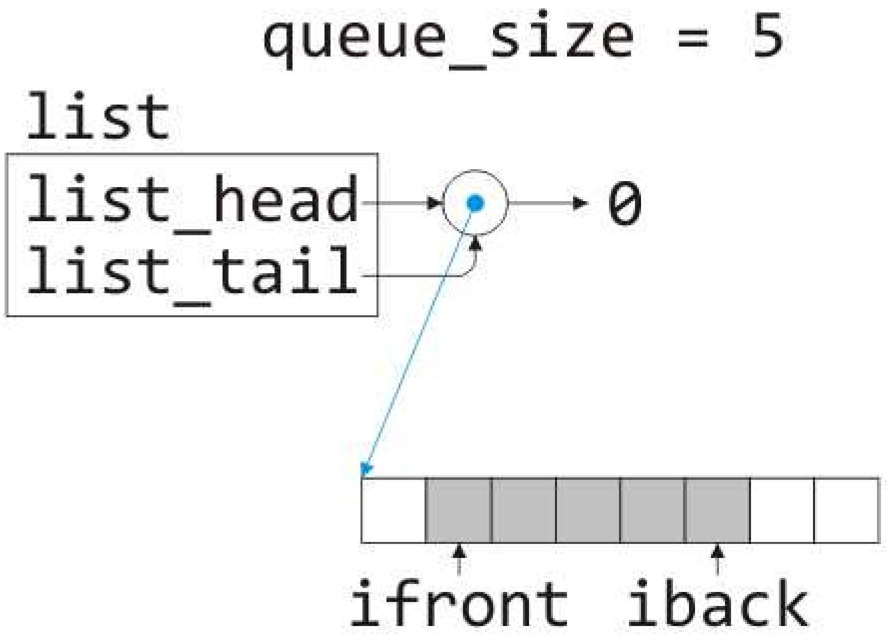

Figure 1. The queue after six pushes and one pop.

Now, suppose that there is a sequence of fourteen pushes: when the array at the back of the linked list is filled, a new array is pushed onto the back of the linked list, and new entries are pushed into the array pointed to by that node. When this array also becomes filled, another array will be pushed onto the linked list. The member variables `ifront` and `iback` keep track of the locations in the first and last entries in their respective arrays. As shown in Figure 2, `ifront = 1` and `itail = 3` and the linked list has three entries.

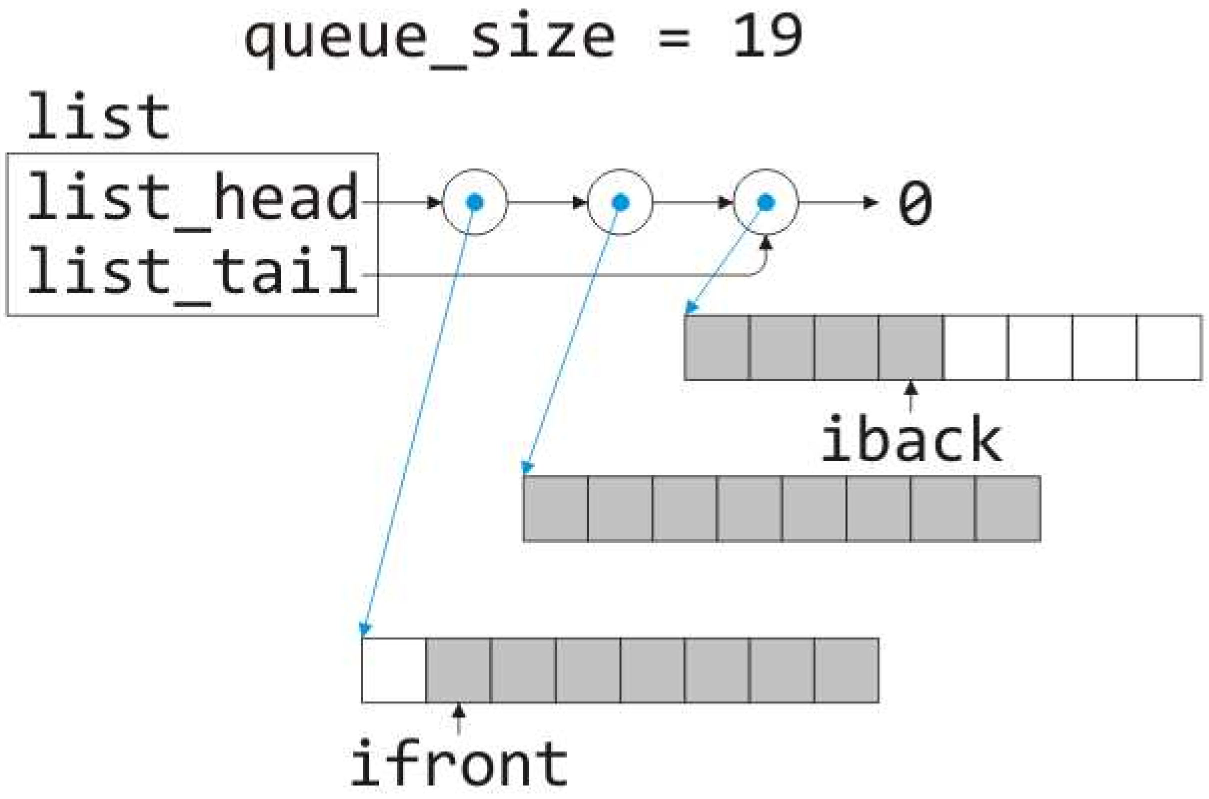

Figure 2. The queue in Figure 1 after another fourteen pushes.

Suppose next there is a sequence of nine pops. After the seventh pop, the node at the front of the linked list would be popped and the memory for the array it points to would be deallocated and `ifront` would be reset to `0`. After two more pops, `ifront = 2`, as shown in Figure 3.

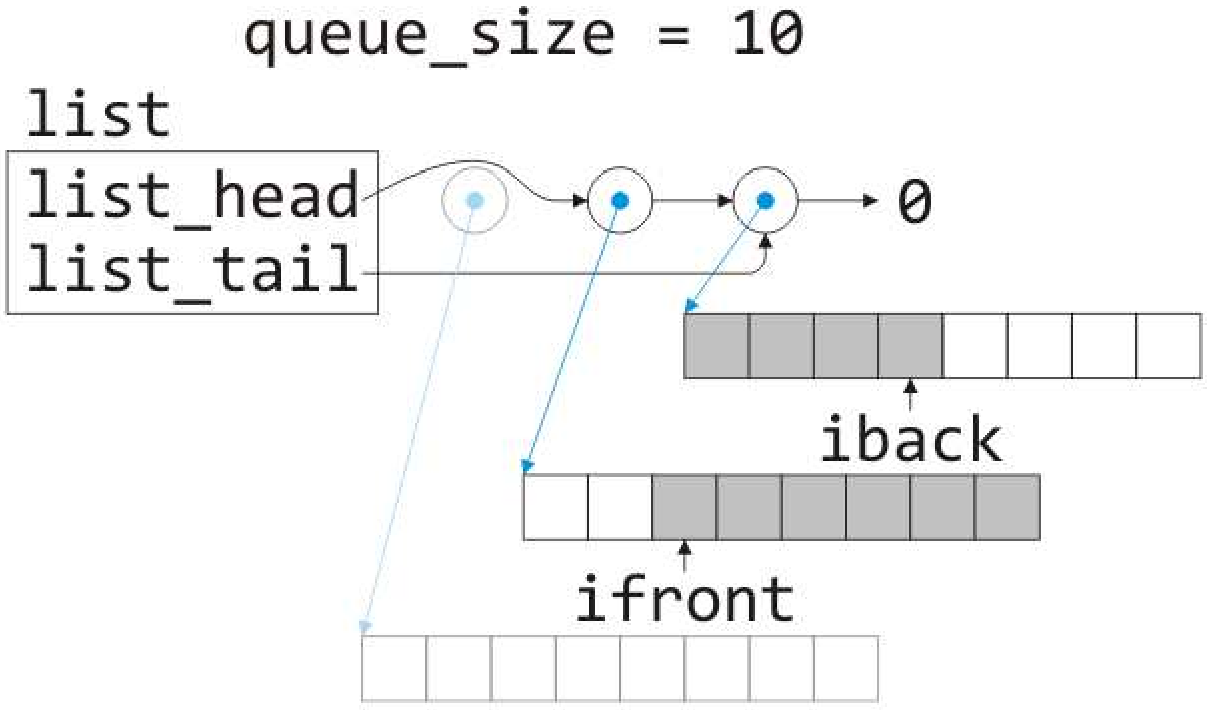

Figure 3. The queue in Figure 2 after nine pops.

Member Functions

Constructors

The default constructor is used.

Copy constructor

Make a complete copy of the linked queue. For each array in the original linked list, a new array must be allocated and the entries copied over.

Destructor

The destructor will have to empty the linked list and deallocate the memory pointed to by the entries of the linked list.

Accessors

This class has three accessors:

`bool empty() const`

Returns true if the queue is empty. (**O**(1))

`int size() const`

Returns the number of objects currently in the queue. (**O**(1)) `int list_size() const`

Returns the number of nodes in the linked list data structure. This must be implemented as provided. (**O**(1))

`Type front() const`

Returns the object at the front of the queue. This member function may throw an `underflow` exception. (**O**(1))

Mutators

This class has four mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void swap( Linked_queue & );`

The swap function swaps all the member variables of this stack with those of the argument. (**O**(*n*))

`Linked_queue &operator=( Linked_queue & );`

The assignment operator makes a copy of the argument and then swaps the member variables of this node with those of the copy. (**O**($n_{\text{lhs}} + n_{\text{rhs}}$))

`void push( Type const & )`

Push the argument onto the back of the queue:

- If the queue is empty, allocate memory for a new array with the required capacity, push the address of that array onto the linked list, set both indices to zero and place the new argument at that location. The size of the queue is now one.
- If the back index already points to the last entry of the array, reset it to zero, allocate memory for a new array with the required capacity, push the address of that array onto the linked list, and insert the argument into the first location.
- Otherwise, increment the back index and place the argument at that location.

Increment the queue size. (**O**(1))

`Type pop()`

Pop the front of the queue and increment the `ifront` index. If the front index equals the aray capacity, reset it to zero and pop the front of the linked list while deallocating the memory allocated to that array. If the queue is emptied, also pop the front of the linked list while deallocated the memory allocated to that array. This member function may throw a `underflow` exception. (**O**(1))