CMSC 341 - Project 1: Circular Linked List - Spring 2021

Due: Tuesday, Mar 02, before 9:00 pm

Addenda

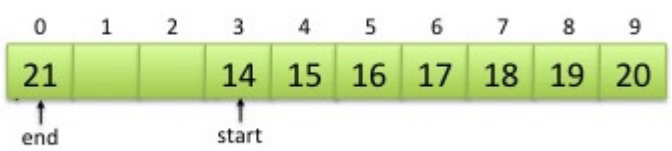
- 02/14/21 - Clarification: the requirements for BufferList::enqueue(...) and BufflerList::dequeue() is asking to catch exceptions thrown from Buffer class. The goal is to perfome some functions if the Buffer object is full or empty. Please note, this can be be done without catching exceptions too. We can simply use the Buffer::empty() and Buffer::full() functions to check on the status of the Buffer object. Both methods of implementation are acceptable.
- 02/14/21 - Clarification: the requirements for the Buffer class indicates that there is no need to initialize the member variable m_next in the Buffer object. The good programming practice tells us that we need to initialize variables. In the implementation of a linked list application, normally the pointers to the next nodes are taken care of in the linked list object. Whether we initialize m_next in the Buffer class or not, we need to manage it in the BufferList class to avoid memory problems.

Objectives

- Implementing a queue data structure.
- Implementing a circular buffer class using arrays.
- Implementing a list container using a circular linked list.

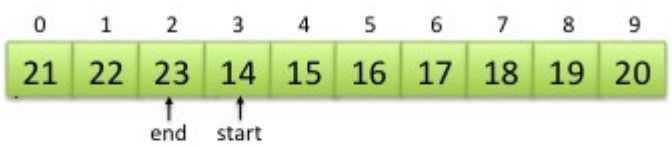
Introduction

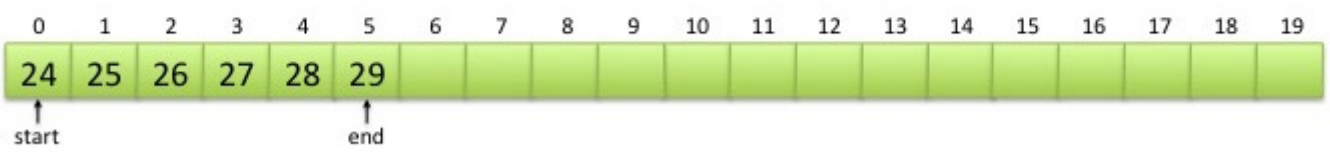
Two common data structures for implementing First-In-First-Out (FIFO) queues are singly linked lists and arrays used as circular buffers. The advantage of using an array is that we can add and remove items from the FIFO queue quickly simply by updating the indices of the start and the end of the queue. As items are added and removed from the array, the indices would "wrap around" and space in the array is reused. In the example below, 14 is at the start of the queue and 21 is at the end. If we added another item to the FIFO queue, it would go in index #1. If we remove an item from the FIFO queue, 14 would be removed.



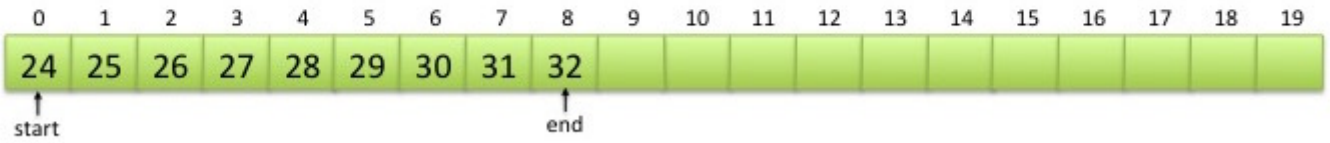
When using an array for a circular buffer, we must make sure that the array is big enough to hold the largest number of items that can be in the FIFO queue simultaneously. Suppose we added two more values 22 and 23 to the circular buffer above. Then, the buffer would be full.

We cannot add another item to the circular buffer. What do we do? The standard trick when we run out of space in an array is to make a new array that holds twice as much data and copy the data over, freeing the old array afterwards. Copying is slow, though. So, instead of copying, we will just add another array for new values, and keep both arrays. The following diagram shows what happens if we add 24, 25, 26, 27, 28 and 29 to the example above.

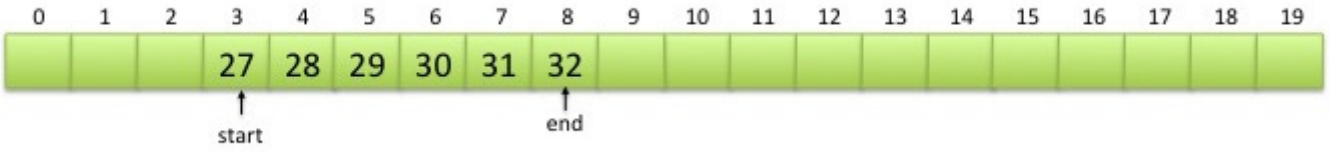




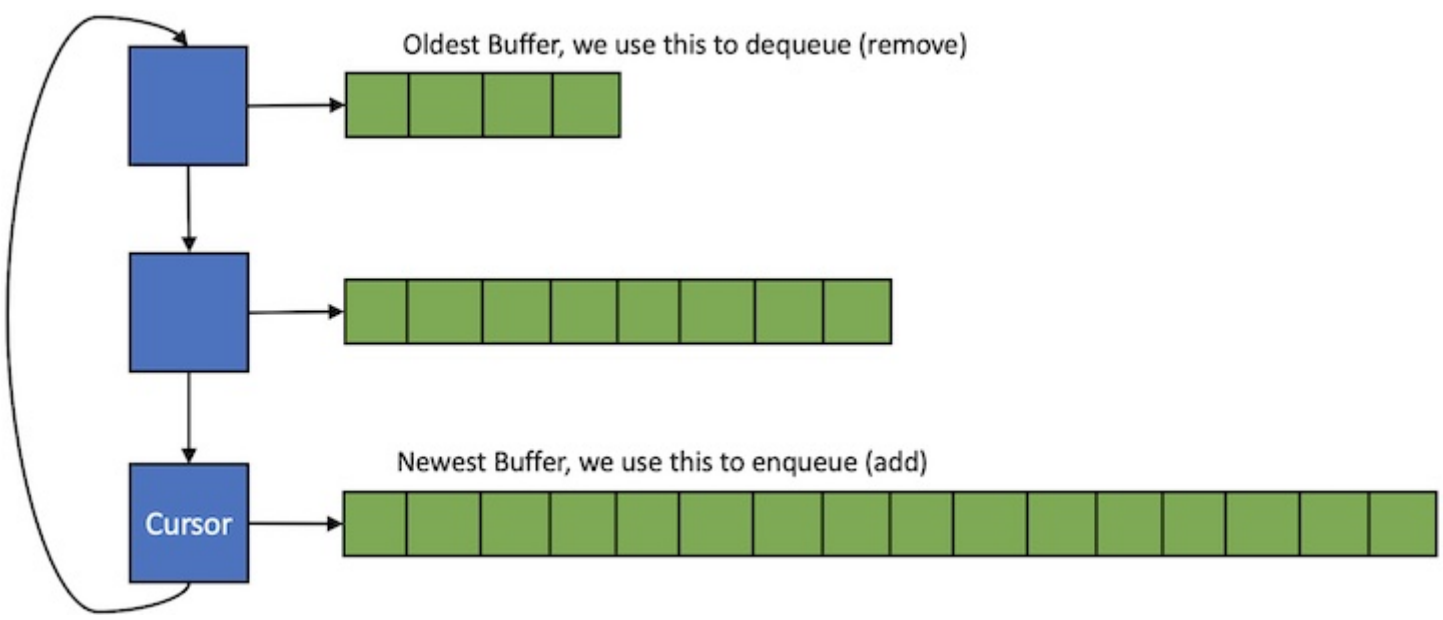
The resulting situation is slightly more complex than a simple expanded single array. We now have to remember that when we remove something from the FIFO queue, we remove it from the start of the old array. but when we add something to the FIFO queue, we add it to the end of the new array. For example, if we added 30, 31 and 32 and removed two items from the queue above, then we would have:



In a simple world, eventually, all of the items in the old array will be removed and we would be left with just the new array. We then fall back to the "normal" circular buffer implemented as a single array. For example, if we removed 11 items from the figure above, we would get:



But what if we add lots of items to the FIFO queue and even the second array fills up? Then, we just create a third array. What if that one becomes full? Then we create another one. And if that one is full? In the general case, we have a bunch of arrays. We keep track of which array is the oldest and which one is the newest. We always add to the newest array and remove from the oldest array. Once the oldest array is emptied, we can deallocate that array, and we remove items from the next oldest array. If the newest array is full, then we create another with twice the capacity. We can use a linked list to keep track of these circular buffers. Every node of the linked list has a pointer to a circular buffer. To keep track of the order of buffers we use a circular linked list. In a circular linked list, we designate a node as cursor, which is always pointing to the newest buffer. The next node of the cursor points to the oldest buffer. The following figure shows the structure.



If we need to allocate a new buffer, we add it as a node after the current cursor and the new node will be the cursor. If we need to deallocate a buffer, in the linked list we remove the oldest node which is the next node of current cursor.

In this PDF file a sample sequence of operations is provided.

Assignment

Your assignment is to implement a queue data structure which consists of a circular linked list and a series of circular array buffers.

For this project, you are provided with the skeleton .h and .cpp files and a sample driver:

- buffer.h - Interface for the Buffer class.
- bufferlist.h - Interface for the BufferList class.
- buffer.cpp - will be completed and submitted.
- bufferlist.cpp - will be completed and submitted.
- driver.cpp - a sample driver program.
- driver.txt - sample output from driver.cpp.

Additionally, you are responsible for thoroughly testing your program. Your test program, `mytest.cpp`, must be submitted along with other files. For grading purposes, your implementation will be tested on input data of varying sizes, including very large data. Your submission will also be checked for memory leaks and memory errors.

Specifications

This project has two classes, i.e. Buffer and BufferList.

Class Buffer

The Buffer class implements a circular array buffer. The member variable m_buffer is a pointer to an array of integer values. The array will be defined by allocating memory dynamically. An object of Buffer class also contains the member variable m_next which is a pointer to another Buffer object. This member variable will be used by the BufferList class for implementation of the linked list.

For the Buffer class, you must implement the following methods in the file `buffer.cpp`:

Buffer(int capacity)	Default constructor allocates memory to an array of int type with the size specified by the parameter capacity. If the specified capacity is less than 1, the constructor sets the capacity to 0. For zero capacity the constructor does not allocate memory. The constructor initializes all member variables. **Note:** there is no need to initialize m_next to nullptr in this class, since this pointer is managed by the linked list class BufferList.
void Buffer::clear()	This function deallocates all memory and reinitializes all member variables. This function can be used by other functions such as destructor or overloaded assignment operator. **Note:** there is no need to initialize m_next to nullptr in this class, since this pointer is managed by the linked list class BufferList.
Buffer::Buffer(const Buffer & rhs)	Copy constructor, creates a deep copy of rhs. A deep copy has its own memory allocated. **Note:** there is no need to initialize m_next to a value in this class, since this pointer is managed by the linked list class BufferList.
const Buffer & Buffer::operator=(const Buffer & rhs)	The overloded assignment operator, creates a deep copy of rhs. It needs to deallocate the current memory if any, and reallocate new memory to create a deep copy. It also needs protection against self-assignment. **Note:** there is no need to initialize m_next to a value in this class, since this pointer is managed by the linked list class BufferList.
void Buffer::enqueue(int data)	This function inserts the data into the array buffer at the proper index. The proper index is the end of the list indicated by the member variable m_end. After every insertion, the function updates the appropriate member variables in the object such as m_end and m_count. If the array is already full, the function throws the exception overflow_error. The exception is defined in the include library <stdexcept>.
int Buffer::dequeue()	This function removes a piece of data from the start of the list in the array. The start index is indicated by the member variable m_start. After removal, the function updates the appropriate member variables in the object such as m_start and m_count. The dequeue function returns the data value. If the buffer is empty the function throws the exception underflow_error. The exception is defined in the include library <stdexcept>.
bool Buffer::full()	This function returns true if the array is full, otherwise it returns false. Full array means there is no more space to store additional data.
bool Buffer::empty()	This function returns true if the array is empty, otherwise it returns false. Empty array means there is no data stored.

Class BufferList

The class BufferList has a pointer to an object of type Buffer. The pointer is stored in the member variable m_cursor. The BufferList is a circular singly linked list. The first node of the linked list has a buffer with the size indicated by the member variable m_minBufCapacity. This value is passed through the constructor. The enqueue function inserts data into this buffer. Once the buffer is full, the enqueue function inserts a new node (Buffer object) into the linked list and starts to insert data in the new buffer. The size of the new buffer is a factor of the size of the buffer in the previous node. We use the constant global variable INCREASE_FACTOR to define the size of the new buffer. For example, if the size of the buffer is 100, and it becomes full, we create a new buffer with the size (INCREASE_FACTOR x 100).

To limit the size of buffer arrays we use the constant global variable MAX_FACTOR. For example, if the minimum size of buffer is 100, the maximum size of a buffer cannot exceed (MAX_FACTOR x 100). As soon as we reach to the maximum size, the next buffer will be created with the minimum size which is defined by the variable member m_minBufCapacity.

To remove data from the queue, the dequeue function removes the data from the start index in the oldest buffer of the linked list. If a buffer becomes empty, the dequeue function removes the buffer from the linked list. If there is only one node in the linked list and its buffer is empty, we do not remove the node. Always there is at least one node in the linked list.

For the BufferList class, you must implement the following methods in the file `bufferlist.cpp`:

BufferList::BufferList(int minBufCapacity)	Constructor creates a BufferList object which contains the first buffer of this object with the size minBufCapacity. If minBufCapacity is less than 1, the constructor creates a buffer with default size DEFAULT_MIN_CAPACITY. The constructor also initializes the member variables of the class.
BufferList::clear()	This function deallocates all memory in the object. Every BufferList object has pointers to Buffer objects, then we also need to deallocate memory for the Buffer objects. To clean a Buffer object we can call clear() function from Buffer.
void BufferList::enqueue(const int& data)	This function inserts data into the queue. It adds the data item at the end index of the newest buffer in the linked list. The newest buffer in the linked list is represented by the member variable m_cursor. We can call Buffer::enqueue(int data) to insert the data item. If the buffer has space, the insertion is performed correctly. If the buffer is full, Buffer::enqueue(int data) throws the overflow_error exception. The exception should be caught here. If an exception is caught, we need to insert a new buffer in the linked list and insert the data item into the new buffer. The new buffer is inserted as the next node of the cursor. Then we advance the cursor, i.e. the new buffer becomes the cursor of the circular linked list.
int BufferList::dequeue()	This function removes data from the queue. It removes the data item from the start index of the oldest buffer in the linked list. The oldest buffer in the linked list is represented by the next node of the member variable m_cursor. We can call Buffer::dequeue() to remove. If the buffer is not empty, the removal is performed correctly. If the buffer is empty, Buffer::dequeue() throws the underflow_error exception. The exception should be caught here. If an exception is caught, we need to remove the node containing the empty buffer from the linked list and remove data from the next node. If there is only one node remaining in the linked list and it is empty, this function throws the underflow_error exception which can be caught by the user program. In a nutshell, if the dequeue operation is successful, this function returns the data item, otherwise it throws the underflow_error exception.
BufferList::BufferList(const BufferList & rhs)	Copy constructor, creates a deep copy of rhs. We can use the copy constructor from Buffer class here to create deep copies of Buffer objects.
const BufferList & BufferList::operator=(const BufferList & rhs)	The assignment operator, creates a deep copy of rhs. We can use the copy constructor from Buffer class here to create deep copies of Buffer objects.

Requirement: The class declarations (`Buffer`) and (`BufferList`) and provided function implementations in `buffer.cpp` and `bufferlist.cpp` may not be modified in any way. No additional libraries may be used, but additional `using` statements are permitted.

Requirement: No STL containers or additional libraries may be used.

Requirement: Your code should not have any memory leaks or memory errors.

Requirement: Follow all coding standards as decribed on the C++ Coding Standards. In particular, indentations and meaningful comments are important.

Testing

Following is a non-exhaustive list of tests to perform on your implementation.

Note: Testing incrementally makes finding bugs easier. Once you finish a function and it is testable, make sure it is working correctly.

Testing Buffer class:

- Test Buffer class for inserting and removing data items for normal cases. (**Note:** We need to test the queue functionality. To test the queue functionality, enqueue and dequeue functions would be tested together. We fill the queue with a certain number of data, and we check whether we can remove the same sequence of data without a problem.)
- Test Buffer class for inserting and removing data items for edge cases. (**Note:** We test the queue functionality with edge sizes for data, i.e. inserting and removing one data item, inserting and removing data with the same size as the buffer size.)
- Test Buffer class for inserting and removing data items for exception cases. (**Note:** We test the dequeue operation on an empty queue, and the enqueue operation on a full queue.)
- Test Buffer class copy constructor for normal and edge cases. We need to check whether a deep copy is created. There are two ways of doing this. We can compare the array pointers, they should not match. Then, We compare all values in the corresponding cells, all values should match. Another way of checking on deep copy is to clear the rhs object, and check whether the current object contains data by calling dequeue function. Either way is acceptable.

Testing BufferList class:

- Test BufferList class for inserting and removing data items for normal cases. (**Note:** We need to test the queue functionality. To test the queue functionality, enqueue and dequeue functions would be tested together. We fill the queue with a certain number of data, and we check whether we can remove the same sequence of data without a problem. To perform this test the data size would be much larger than the data size used to test the same functionality in the Buffer class. We want to make sure that BufferList adds more Buffer objects as required. The example of data size in this test would be 10,000.)
- Test BufferList class for removing data items for exception cases. (**Note:** We test the dequeue operation on an empty queue. Trying to remove from empty queue throws an underflow_error exception. The BufferList class never gets full, then it never throws an overflow_error exception.)
- Test BufferList class copy constructor for normal and edge cases. (**Note:** We can call the test function for Buffer class copy constructor here.)
- Test BufferList class assignment operator. To check whether it created a deep copy, we can clear the rhs object and check that the data is preserved in the current object by calling dequeue.
- Test the running time of BufferList::enqueue(...) grows linearly in relation to the growth of data size. (**Note:** in the sample driver output there is an example that shows when the data size doubles, the running time doubles.)

Memory leaks and errors:

- Run your test program in `valgrind`; check that there are no memory leaks or errors.
  Note: If `valgrind` finds memory errors, compile your code with the -g option to enable debugging support and then re-run `valgrind` with the `-s` and `--track-origins=yes` options. valgrind will show you the lines numbers where the errors are detected and can usually tell you which line is causing the error.
- Never ignore warnings. They are a major source of errors in a program.

What to Submit

You must submit the following files to the `proj1` submit directory:

- `buffer.h`
- `buffer.cpp`
- `bufferlist.h`
- `bufferlist.cpp`
- `mytest.cpp` (**Note:** This file contains the declaration and implementation of your Tester class as well as all your test cases.)

If you followed the instructions in the Project Submission page to set up your directories, you can submit your code using the following command:

```
cp buffer.h buffer.cpp bufferlist.h bufferlist.cpp mytest.cpp ~/cs341proj/proj1/
```

Grading Rubric

The following presents a course rubric. It shows how a submitted project might lose points.

- Conforming to coding standards make about 10% of the grade.
- Correctness and completeness of your test cases (mytest.cpp) make about 15% of the grade.
- Passing tests make about 30% of the grade.

If the submitted project is in a state that receives the deduction for all above items, it will be graded for efforts. The grade will depend on the required efforts to complete such a work.
