Implementing Queues

Eric Roberts  
CS 106B  
February 13, 2013

Outline

• Chapter 14 presents complete implementations for three of the ADTs you’ve been using since the second assignment: stacks, queues, and vectors. We described one implementation of the Stack class in Monday’s lecture.

• The reader presents two different strategies to represent the Stack and Queue classes: one that uses an array to store the elements and one that uses a linked list. For each of these strategies, implementing a Stack turns out to be much easier.

• In today’s lecture, I am going to focus on implementing the Queue class, leaving the details of implementing the Stack and Vector classes to the text.

Methods in the `Queue<ValueType>` Class

```c++
int size();  // Returns the number of values in the queue.

boolean isEmpty();  // Returns true if the queue is empty.

void enqueue(ValueType value);  // Adds a new value to the end of the queue (which is called its tail).

ValueType dequeue();  // Removes and returns the value at the front of the queue (which is called its head).

ValueType peek();  // Returns the value at the head of the queue without removing it.

void clear();  // Removes all values from the queue.
```

The `queue.h` Interface

```c++
template <typename ValueType>
class Queue {
public:
    /*
     * Constructor: Queue
     * Usage: Queue<ValueType> queue;
     * Initializes a new empty queue containing the specified value type.
     */
    Queue();

    /* Destructor: ~Queue
     * Removes any heap storage associated with this queue.
     */
    ~Queue();

    /* Method: size
     * Usage: int nElems = queue.size();
     * Returns the number of elements in this queue.
     */
    int size();

    /* boolean isEmpty()
     * Returns true if the queue is empty.
     */
    boolean isEmpty();

    /* void clear()
     * Removes all values from the queue.
     */
    void clear();

    /* void enqueue(ValueType value)
     * Adds a new value to the end of the queue (which is called its tail).
     */
    void enqueue(ValueType value);

    /* ValueType dequeue()
     * Removes and returns the value at the front of the queue (which is called its head).
     */
    ValueType dequeue();

    /* ValueType peek()
     * Returns the value at the head of the queue without removing it.
     */
    ValueType peek();
};
```
The `queue.h` Interface

```
/* Private section */

Private section goes here.

/**
 * The template feature of C++ works correctly only if the compiler
 * has access to the code without forcing the client
 * to wade through the details.
 * Compiler to have access to the code without forcing the client
 * Using the #include facility of the C++ preprocessor allows the
 * not something that the client needs to see in the interface.
 * the implementation at this point, even though that code is
 * same time. As a result, the compiler must see the code for
 * has access to both the interface and the implementation at the
 * The template feature of C++ works correctly only if the compiler
 */
```

Implement section goes here.

```
#define
```

An Overly Simple Strategy

- The most straightforward way to represent the elements of a queue is to store the elements in an array, exactly as in the `Stack` class from Monday.
- Given this representation, the `enqueue` operation is extremely simple to implement. All you need to do is add the element to the end of the array and increment the element count. That operation runs in \(O(1)\) time.
- The problem with this simplistic approach is that the `dequeue` operation requires removing the element from the beginning of the array. If you’re relying on the same strategy you used in the array-based editor, implementing this operation requires moving all the remaining elements to fill the hole left by the dequeued element. That operation therefore takes \(O(N)\) time.

Fixing the \(O(N)\) Problem

- The key to fixing the problem of having `dequeue` take \(O(N)\) time is to eliminate the need for any data motion by keeping track of two indices: one to mark the head of the queue and another to mark the tail.
- Given these two indices, the `enqueue` operation stores the new element at the position marked by the `tail` index and then increments `tail` so that the next element is enqueued into the next slot. The `dequeue` operation is symmetric. The next value to be dequeued appears at the array position marked by the `head` index. Removing it is then simply a matter of incrementing `head`.
- Unfortunately, this strategy typically ends up filling the array space even when the queue itself contains very few elements, as illustrated on the next slide.

Tracing the Array-Based Queue

At this point, enqueuing the I would require expanding the array, even though the queue contains only four elements.
- The solution to this problem is to let the elements cycle back to the beginning of the array.

Tracing the Array-Based Queue

```
Queue<char> queue;
queue.enqueue('I');
queue.enqueue('H');
queue.enqueue('G');
queue.dequeue();
queue.dequeue();
queue.dequeue();
queue.enqueue('E');
queue.enqueue('D');
queue.dequeue();
queue.enqueue('C');
queue.enqueue('B');
queue.enqueue('A');
```

```
Queue<char> queue;
queue.enqueue('A');
queue.enqueue('B');
queue.enqueue('C');
queue.dequeue() \rightarrow 'A';
queue.enqueue('D');
queue.enqueue('E');
queue.dequeue() \rightarrow 'B';
queue.enqueue('F');
queue.enqueue('G');
queue.dequeue() \rightarrow 'C';
queue.enqueue('H');
queue.enqueue('I');
```

Implementing the Ring-Buffer Strategy

- The data structure described on the preceding slide is called a ring buffer because the end of the array is linked back to the beginning.
- The arithmetic operations necessary to implement the ring buffer strategy are easy to code using modular arithmetic, which is simply normal arithmetic in which all values are reduced to a specific range by dividing each result by some constant (in this case, the capacity of the array) and taking the remainder. In C++, you can use the `%` operator to implement modular arithmetic.
- When you are using the ring-buffer strategy, it is typically necessary to expand the queue when there is still one free element left in the array. If you don’t do so, the simple test for an empty queue—whether `head` is equal to `tail`—fails because that would also be true in a completely full queue.

```
Queue<char> queue;
queue.enqueue('A');
queue.enqueue('B');
queue.enqueue('C');
queue.enqueue('D');
queue.enqueue('E');
queue.enqueue('F');
queue.enqueue('G');
queue.enqueue('H');
queue.enqueue('I');
```

```
Queue<char> queue;
queue.enqueue('A');
queue.enqueue('B');
queue.enqueue('C');
queue.enqueue('D');
queue.enqueue('E');
queue.enqueue('F');
queue.enqueue('G');
queue.enqueue('H');
queue.enqueue('I');
```
Structure for the Array-Based Queue

```cpp
/* Implementation notes */

/* The array-based queue stores the elements in a ring buffer, which consists of a dynamic array and two indices: head and tail. Each index wraps to the beginning if necessary. This design requires that there is always one unused element in the array. If the queue were allowed to fill completely, the head and tail indices would have the same value, and the queue will appear empty. */

/* Private: */
static const int INITIAL_CAPACITY = 10;

/* Instance variables */
ValueType *array;
int capacity;
int head;
int tail;

/* Private method prototypes */
void expandCapacity();

void Queue<ValueType>::~Queue() {
    delete[] array;
}

void Queue<ValueType>::Queue() {
    template <typename ValueType>
    
    capacity = INITIAL_CAPACITY;
    array = new ValueType[capacity];
    head = 0;
    tail = 0;
}

/* Implementation notes: Queue constructor */

/* Implementation notes: dequeue */

int Queue<ValueType>::size() {
    return (tail + capacity - head) % capacity;
}

/* Implementation notes: size */

bool Queue<ValueType>::isEmpty() {
    return head == tail;
}

/* Implementation notes: isEmpty */

void Queue<ValueType>::clear() {
    head = tail = 0;
}

/* Implementation notes: clear */

void Queue<ValueType>::enqueue(ValueType value) {
    array[tail] = value;
    tail = (tail + 1) % capacity;
    if (size() == capacity - 1) expandCapacity();
}

/* Implementation notes: enqueue */

ValueType Queue<ValueType>::peek() {
    if (isEmpty()) error("peek: Attempting to peek at an empty queue");
    ValueType result = array[head];
    head = (head + 1) % capacity;
    return result;
}

/* Implementation notes: peek */

```

Code for the Array-Based Queue

```cpp
/* Implementation notes */

/* The array-based queue stores the elements in a ring buffer, which consists of a dynamic array and two indices: head and tail. Each index wraps to the beginning if necessary. This design requires that there is always one unused element in the array. If the queue were allowed to fill completely, the head and tail indices would have the same value, and the queue will appear empty. */

/* Private: */
static const int INITIAL_CAPACITY = 10;

/* Instance variables */
ValueType *array;
int capacity;
int head;
int tail;

/* Private method prototypes */
void expandCapacity();

void Queue<ValueType>::~Queue() {
    delete[] array;
}

void Queue<ValueType>::Queue() {
    template <typename ValueType>
    
    capacity = INITIAL_CAPACITY;
    array = new ValueType[capacity];
    head = 0;
    tail = 0;
}

/* Implementation notes: Queue constructor */

/* Implementation notes: dequeue */

int Queue<ValueType>::size() {
    return (tail + capacity - head) % capacity;
}

/* Implementation notes: size */

bool Queue<ValueType>::isEmpty() {
    return head == tail;
}

/* Implementation notes: isEmpty */

void Queue<ValueType>::clear() {
    head = tail = 0;
}

/* Implementation notes: clear */

void Queue<ValueType>::enqueue(ValueType value) {
    array[tail] = value;
    tail = (tail + 1) % capacity;
    if (size() == capacity - 1) expandCapacity();
}

/* Implementation notes: enqueue */

ValueType Queue<ValueType>::peek() {
    if (isEmpty()) error("peek: Attempting to peek at an empty queue");
    ValueType result = array[head];
    head = (head + 1) % capacity;
    return result;
}

/* Implementation notes: peek */

```

Code for the Ring-Buffer Queue

```cpp
/* Implementation notes */

/* The ring buffer queue stores the elements in a ring buffer, which consists of a dynamic array and two indices: head and tail. Each index wraps to the beginning if necessary. This design requires that there is always one unused element in the array. If the queue were allowed to fill completely, the head and tail indices would have the same value, and the queue will appear empty. */

/* Private: */
static const int INITIAL_CAPACITY = 10;

/* Instance variables */
ValueType *array;
int capacity;
int head;
int tail;

/* Private method prototypes */
void expandCapacity();

void Queue<ValueType>::~Queue() {
    delete[] array;
}

void Queue<ValueType>::Queue() {
    template <typename ValueType>
    
    capacity = INITIAL_CAPACITY;
    array = new ValueType[capacity];
    head = 0;
    tail = 0;
}

/* Implementation notes: Queue constructor */

/* Implementation notes: dequeue */

int Queue<ValueType>::size() {
    return (tail + capacity - head) % capacity;
}

/* Implementation notes: size */

bool Queue<ValueType>::isEmpty() {
    return head == tail;
}

/* Implementation notes: isEmpty */

void Queue<ValueType>::clear() {
    head = tail = 0;
}

/* Implementation notes: clear */

void Queue<ValueType>::enqueue(ValueType value) {
    array[tail] = value;
    tail = (tail + 1) % capacity;
    if (size() == capacity - 1) expandCapacity();
}

/* Implementation notes: enqueue */

ValueType Queue<ValueType>::peek() {
    if (isEmpty()) error("peek: Attempting to peek at an empty queue");
    ValueType result = array[head];
    head = (head + 1) % capacity;
    return result;
}

/* Implementation notes: peek */

```

Code for the Ring-Buffer Queue

```cpp
/* Implementation notes */

/* The ring buffer queue stores the elements in a ring buffer, which consists of a dynamic array and two indices: head and tail. Each index wraps to the beginning if necessary. This design requires that there is always one unused element in the array. If the queue were allowed to fill completely, the head and tail indices would have the same value, and the queue will appear empty. */

/* Private: */
static const int INITIAL_CAPACITY = 10;

/* Instance variables */
ValueType *array;
int capacity;
int head;
int tail;

/* Private method prototypes */
void expandCapacity();

void Queue<ValueType>::~Queue() {
    delete[] array;
}

void Queue<ValueType>::Queue() {
    template <typename ValueType>
    
    capacity = INITIAL_CAPACITY;
    array = new ValueType[capacity];
    head = 0;
    tail = 0;
}

/* Implementation notes: Queue constructor */

/* Implementation notes: dequeue */

int Queue<ValueType>::size() {
    return (tail + capacity - head) % capacity;
}

/* Implementation notes: size */

bool Queue<ValueType>::isEmpty() {
    return head == tail;
}

/* Implementation notes: isEmpty */

void Queue<ValueType>::clear() {
    head = tail = 0;
}

/* Implementation notes: clear */

void Queue<ValueType>::enqueue(ValueType value) {
    array[tail] = value;
    tail = (tail + 1) % capacity;
    if (size() == capacity - 1) expandCapacity();
}

/* Implementation notes: enqueue */

ValueType Queue<ValueType>::peek() {
    if (isEmpty()) error("peek: Attempting to peek at an empty queue");
    ValueType result = array[head];
    head = (head + 1) % capacity;
    return result;
}

/* Implementation notes: peek */

```
Implementing a Linked-List Queue

- In some ways, the linked-list implementation of a queue is easier to understand than the ring-buffer model, even though it contains pointers.
- In the linked-list version, the private data structure for the Queue class requires two pointers to cells: a head pointer that indicates the first cell in the chain, and a tail pointer that indicates the last cell. Because all insertion happens at the tail of the queue, no dummy cell is required.
- The private data structure must also keep track of the number of elements so that the size method can run in O(1) time.

Structure for the List-Based Queue

- Implementation notes: Queue data structure
  - The list-based queue uses a linked list to store the elements of the queue. To ensure that adding a new element to the tail of the queue is fast, the implementation maintains a pointer to the last cell in the queue as well as the first. If the queue is empty, both the head pointer and the tail pointer are set to NULL.

```
private:
  /* Type for linked list cell */
  struct Cell {
    ValueType data;  /* The data value */
    Cell *link;     /* Link to the next cell */
  };

  /* Instance variables */
  Cell *head;     /* Pointer to the cell at the head */
  Cell *tail;     /* Pointer to the cell at the tail */
  int count;     /* Number of elements in the queue */
```

Code for the Linked-List Queue

```
/* Implementation notes: Queue constructor */
/* Initializes the fields of the object. */
template <typename ValueType>
Queue<ValueType>::Queue() {
  head = tail = NULL;
  count = 0;
}

/* Implementation notes: ~Queue destructor */
/* Frees any memory that is allocated by the implementation. */
/* The clients that use the queue abstraction must ensure that */
/* all of their memory is cleared before calling the destructor. */
template <typename ValueType>
Queue<ValueType>::~Queue() {
  clear();
}

/* Implementation notes: size, isEmpty, clear */
/* These implementations should be self-explanatory */
template <typename ValueType>
int Queue<ValueType>::size() {
  return count;
}

template <typename ValueType>
bool Queue<ValueType>::isEmpty() {
  return count == 0;
}

template <typename ValueType>
void Queue<ValueType>::clear() {
  while (count > 0) {
    dequeue();
  }
}

/* Implementation notes: enqueue */
/* This method allocates a new list cell and chains it in at the tail of the queue. */
/* If the queue is currently empty, the new cell must also become the head pointer in the queue. */
template <typename ValueType>
void Queue<ValueType>::enqueue(ValueType value) {
  Cell *cp = new Cell;
  cp->data = value;
  cp->link = NULL;
  if (head == NULL) {
    head = cp;
  } else {
    tail->link = cp;
  }
  tail = cp;
  count++;
}

/* Implementation notes: dequeue, peek */
/* These methods must check for an empty queue and report an error if there is no first element. */
/* The dequeue method must also check for the case in which the queue becomes empty and set both the head and tail pointers to NULL. */
template <typename ValueType>
ValueType Queue<ValueType>::dequeue() {
  if (isEmpty()) error("dequeue: Attempting to dequeue an empty queue");
  Cell *cp = head;
  ValueType result = cp->data;
  head = cp->link;
  if (head == NULL) tail = NULL;
  count--;
  delete cp;
  return result;
}

template <typename ValueType>
ValueType Queue<ValueType>::peek() {
  if (isEmpty()) error("peek: Attempting to peek at an empty queue");
  return head->data;
}
```