Implementing Queues

Eric Roberts
CS 106B
February 11, 2015

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 textbook 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<\*x*> Class

queue.size()

Returns the number of values in the queue.

queue.isEmpty()

Returns true if the queue is empty.

queue.enqueue(value)

Adds a new value to the end of the queue (which is called its tail).

queue.dequeue()

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

queue.peek()

Returns the value at the head of the queue without removing it.

queue.clear()

Removes all values from the queue.

The queue.h Interface

```cpp
#include <queue>

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
     * --------------
     * Deallocates 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();

    /* . . . */

    /* Method: isEmpty
     * Usage: bool isE = queue.isEmpty();
     * . . . */
    bool isEmpty();

    /* Method: clear
     * Usage: void c = queue.clear();
     * . . . */
    void clear();

    /* Method: enqueue
     * Usage: void e = queue.enqueue(value);
     * . . . */
    void enqueue(ValueType value);

    /* Method: dequeue
     * Usage: ValueType d = queue.dequeue();
     * . . . */
    ValueType dequeue();

    /* Method: peek
     * Usage: ValueType p = queue.peek();
     * . . . */
    ValueType peek();

};
```

The queue.h Interface

```cpp
#ifndef _queue_h
#define _queue_h

/* File: queue.h
 * -------------
 * This interface defines a general queue abstraction that uses
 * templates so that it can work with any element type.
 */

#include <assert.h>

/* Template class: Queue<ValueType>
 * --------------------------------
 * This class template models a queue, which is a linear collection
 * of values that resemble a waiting line. Values are added at
 * one end of the queue and removed from the other. The fundamental
 * operations are enqueue (add to the tail of the queue) and dequeue
 * (remove from the head of the queue). Because a queue preserves
 * the order of the elements, the first value enqueued is the first
 * value dequeued. Queues therefore operate in a first-in-first-out
 * (FIFO) order. For maximum generality, the Queue class is defined
 * using a template that allows the client to define a queue that
 * contains any type of value, as in Queue<string> or Queue<taskT>.
 */

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
     * --------------
     * Deallocates 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();

    /* . . . */

    /* Method: isEmpty
     * Usage: bool isE = queue.isEmpty();
     * . . . */
    bool isEmpty();

    /* Method: clear
     * Usage: void c = queue.clear();
     * . . . */
    void clear();

    /* Method: enqueue
     * Usage: void e = queue.enqueue(value);
     * . . . */
    void enqueue(ValueType value);

    /* Method: dequeue
     * Usage: ValueType d = queue.dequeue();
     * . . . */
    ValueType dequeue();

    /* Method: peek
     * Usage: ValueType p = queue.peek();
     * . . . */
    ValueType peek();

};
```
The queue.h Interface

/* Private section */

Private section goes here.

} /* End of Private section */

Implementation section goes here.

#endif /* Private section */

/* 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
*/

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

- Consider what happens if you execute the operations shown.
- Each enqueue operation adds a value at the tail of the queue.
- Each dequeue operation takes
its result from the head.
- If you continue on in this way, what happens when you reach
the end of the array space?

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.
Structure for the Array-Based Queue

/* 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 */
/* index wraps to the beginning if necessary. This design requires */
/* have the same value, and the queue will appear empty. */
/* Private: */
static const int INITIAL_CAPACITY = 10;

/* Instance variables */
ValueType *array; /* A dynamic array of the elements */
int capacity; /* The allocated size of the array */
int head; /* The index of the head element */
int tail; /* The index of the tail element */

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

/* Implementation section */

Code for the Ring-Buffer Queue

/* Implementation section */
/* 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 */
/* index wraps to the beginning if necessary. This design requires */
/* have the same value, and the queue will appear empty. */
/* Private: */
static const int INITIAL_CAPACITY = 10;

/* Instance variables */
ValueType *array; /* A dynamic array of the elements */
int capacity; /* The allocated size of the array */
int head; /* The index of the head element */
int tail; /* The index of the tail element */

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

/* Implementation section */
/* The queue is empty whenever the head and tail indices are */
/* equal. Note that this interpretation means that the queue */
/* always leave one unused space. */
bool Queue<ValueType>::isEmpty() {
    return head == tail;
}

/* Implementation notes: expandCapacity */
/* This private method doubles the capacity of the elements array */
/* whenever it runs out of space. To do so, it must allocate a new */
/* array, copy all the elements from the old array to the new one, */
/* and free the old storage. Note that this implementation also */
/* shifts all the elements back to the beginning of the array. */
void Queue<ValueType>::expandCapacity() {
    int oldCapacity = capacity;
    ValueType *oldArray = array;
    capacity *= 2;
    array = new ValueType[capacity];
    for (int i = 0; i < count; i++) {
        array[(head + i) % oldCapacity] = oldArray[i];
    }
    head = tail = 0;
    delete[] oldArray;
}

/* Implementation notes: size */
/* The size of the queue can be calculated from the head and tail */
/* indices by using modular arithmetic. */
int Queue<ValueType>::size() {
    return (tail + capacity - head) % capacity;
}

/* Implementation notes: isEmpty */
/* The ring buffer is empty whenever the head and tail indices are */
/* equal. Note that this interpretation means that the queue */
/* always leave one unused space. */
bool Queue<ValueType>::isEmpty() {
    return head == tail;
}

/* Implementation notes: enqueue */
/* This method must first check to see whether there is */
/* enough room for the element and expand the array storage */
/* if necessary. */
void Queue<ValueType>::enqueue(ValueType value) {
    if (size() == capacity - 1) expandCapacity();
    array[tail] = value;
    if (tail == capacity - 1) tail = 0;
    else tail = (tail + 1) % capacity;
}

/* Implementation notes: dequeue */
/* This method must first check to see whether there is */
/* enough room in the ring buffer. Note that this implementation also */
/* shifts all the elements back to the beginning of the array. */
ValueType Queue<ValueType>::dequeue() {
    if (isEmpty()) error("dequeue: Attempting to dequeue an empty queue");
    ValueType value = array[head];
    if (head == capacity - 1) head = 0;
    else head = (head + 1) % capacity;
    return value;
}

/* Implementation notes: peek */
/* This method must first check to see whether there is */
/* enough room for the element and expand the array storage */
/* if necessary. */
ValueType Queue<ValueType>::peek() {
    if (isEmpty()) error("peek: Attempting to peek at an empty queue");
    return array[head];
}

/* Implementation notes: clear */
/* The clear method need not take account of where in the */
/* ring buffer any existing data is stored and can simply */
/* reset the head and tail indices back to the beginning. */
void Queue<ValueType>::clear() {
    head = tail = 0;
    for (int i = 0; i < count; i++) {
        array[i] = oldArray[(head + i) % oldCapacity];
    }
}

/* Implementation notes: Queue constructor */
/* The constructor must allocate the array storage for the queue */
/* elements and initialize the fields of the object. */
Queue<ValueType>::Queue() {
    capacity = INITIAL_CAPACITY;
    array = new ValueType[capacity];
    head = 0;
    tail = 0;
}

/* Implementation notes: Queue destructor */
/* The destructor frees any memory that is allocated by the implementation. */
Queue<ValueType>::~Queue() {
    delete[] array;
}

/* Implementation notes: enqueue */
/* This method must first check to see whether there is */
/* enough room in the ring buffer. Note that this implementation also */
/* shifts all the elements back to the beginning of the array. */
void Queue<ValueType>::enqueue(ValueType value) {
    if (size() == capacity - 1) expandCapacity();
    array[tail] = value;
    if (tail == capacity - 1) tail = 0;
    else tail = (tail + 1) % capacity;
}
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

```cpp
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

```cpp
// Implementation notes: Queue constructor
template <typename ValueType>
Queue<ValueType>::Queue() {
  head = tail = NULL;
  count = 0;
}

// Implementation notes: ~Queue destructor
template <typename ValueType>
Queue<ValueType>::~Queue() {
  clear();
}

// Implementation notes: size, isEmpty, clear
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
template <typename ValueType>
void Queue<ValueType>::enqueue(ValueType value) {
  Cell *cp = new Cell;
  cp->data = value;
  cp->link = NULL;
  if (head == NULL) {
    head = cell;
  } else {
    tail->link = cell;
  }
  tail = cell;
  count++;
}

// Implementation notes: dequeue, peek
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 cell;
  return result;
}

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