ADTs: Stacks and Queues

CS 3358
Spring 2015

Jill Seaman

All Sections of chapter 16 (or Gaddis chapter 18)

Introduction to the Stack

- Stack: a data structure that holds a collection of elements of the same type.
  - The elements are accessed according to LIFO order: last in, first out
  - No random access to other elements
- Examples:
  - plates in a cafeteria
  - bangles . . .

Stack Operations

- Operations:
  - push: add a value onto the top of the stack
    - make sure it's not full first.
  - pop: remove (and return) the value from the top of the stack
    - make sure it's not empty first.
  - isFull: true if the stack is currently full, i.e.,has no more space to hold additional elements
  - isEmpty: true if the stack currently contains no elements
- These operations should take constant time: O(1).

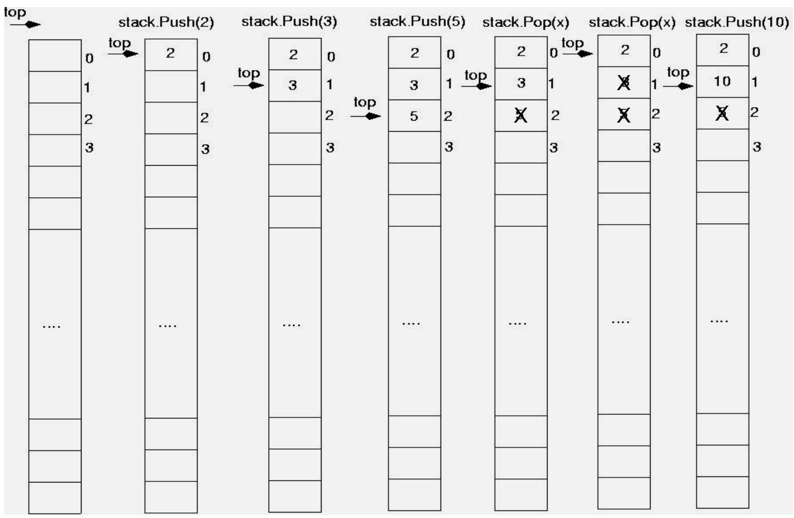
Stack Operations

- Operations:
  - makeEmpty: removes all the elements
- This is allowed to take longer than constant time.

Stack Terms

- Stack overflow:
  - trying to push an item onto a full stack
- Stack underflow.
  - trying to pop an item from an empty stack

Stack illustrated



Stack Applications

- Matching brackets in a text file
  Do the brackets all match?

```
if (x==list.getCurrent())
{ z[i] = x; count++; }
}
```

- What else?

Implementing a Stack Class

- Array implementations:
  - fixed arrays: size doesn’t change
  - dynamic arrays: can resize as needed in push
- Linked List
  - grow and shrink in size as needed
- Templates
  - any of the above can be implemented using templates

A static stack class

```
class IntStack
{
private:
   const int STACKSIZE = 100;  // The stack size
   int stackArray[STACKSIZE];  // The stack array
   int top;          // Index to the top of the stack

public:
   // Constructor
   IntStack();

   // Stack operations
   void push(int);
   int pop();
   bool isFull() const;
   bool isEmpty() const;
   void makeEmpty();
};
```

A static stack class: functions

```
//***********************************************
// Constructor                                  *
// This constructor creates an empty stack.     *
//***********************************************

IntStack::IntStack()
{
   top = -1;                    // empty
}


//no need to initialize the static array stackArray
```

A static stack class: push

```
//*************************************************
// Member function push pushes the argument onto  *
// the stack.                                     *
//*************************************************

void IntStack::push(int num)
{
   assert(!isFull());

   top++;
   stackArray[top] = num;
}
```

A static stack class: pop

```
//****************************************************
// Member function pop pops the value at the top     *
// of the stack off, and returns it.                 *
//****************************************************

int IntStack::pop()
{
   assert(!isEmpty());

   int num = stackArray[top];
   top--;
   return num;
}
```

A static stack class: functions

```
//***************************************************
// Member function isFull returns true if the stack *
// is full, or false otherwise.                     *
//***************************************************

bool IntStack::isFull() const
{
   return (top == stackSize - 1);
}

//****************************************************
// Member function isEmpty returns true if the stack *
// is empty, or false otherwise.                     *
//****************************************************

bool IntStack::isEmpty() const
{
   return (top == -1);
}
```

A static stack class: makeEmpty

```
//****************************************************
// Member function makeEmpty makes the stack an      *
// empty stack.                                      *
//****************************************************

void IntStack::makeEmpty()
{
   top = -1;
}
```

A Dynamic Stack Class

- stack_3358_LL.h
  - On the class website
  - Singly-linked-list implementation
  - Templated (all code in *.h file)
  - Push and pop from the head of the list

Introduction to the Queue

- Queue: a data structure that holds a collection of elements of the same type.
  - The elements are accessed according to FIFO order: first in, first out
  - No random access to other elements
- Examples:
  - people in line at a theatre box office
  - restocking perishable inventory

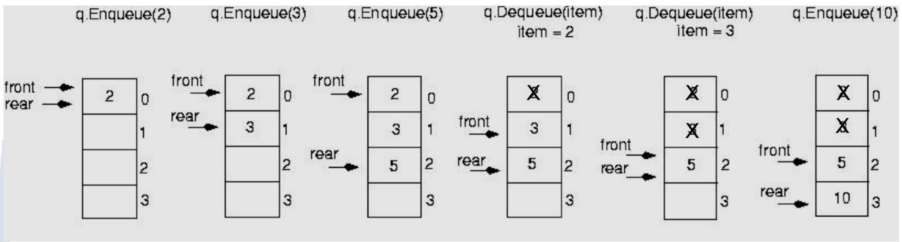
Queue Operations

- Operations:
  - enqueue: add a value onto the rear of the queue (the end of the line)
    - make sure it's not full first.
  - dequeue: remove a value from the front of the queue (the front of the line) "Next!"
    - make sure it's not empty first.
  - isFull: true if the queue is currently full, i.e.,has no more space to hold additional elements
  - isEmpty: true if the queue currently contains no elements
- These operations should take constant time: O(1)

Queue Operations

- Operations:
  - makeEmpty: removes all the elements
- This is allowed to take longer than constant time.

Queue illustrated



Note: front and rear are variables used by the implementation to carry out the operations

```
int item;
q.enqueue(2):
q.enqueue(3);
q.enqueue(5);
item = q.dequeue(); //item is 2
item = q.dequeue(); //item is 3
q.enqueue(10);
```

Queue Applications

- The best examples of applications of queues involve managing multiple processes.
- For example, imagine the print queue for a computer lab.
- Any computer can add a new print job to the queue (enqueue).
- The printer performs the dequeue operation and starts printing that job.
- While it is printing, more jobs are added to the Q
- When the printer finishes, it pulls the next job from the Q, continuing until the Q is empty

Implementing a Queue Class

- Just like stacks, queues can be implemented using arrays (fixed size, or resizing dynamic arrays) or linked lists (dynamic queues), and may be implemented using templates.
- The previous illustration assumed we were using an array to implement the queue
- When an item was dequeued, the items were NOT shifted up to fill the slot vacated by dequeued item
  - why not?
- Instead, both front and rear indices move in the array.

Queue implemented

problem: end of the array

- When front and rear indices move in the array:
  - problem: rear hits end of array quickly
  - solution: wrap index around to front of array

					7	9	6
					front		rear

3					7	9	6
rear					front		

3	4				7	9	6
	rear				front		

Implementing a Queue Class

- To “wrap” the rear index back to the front of the array, you can use this code to increment rear during enqueue:

```
if (rear == queueSize-1)
   rear = 0;
else
   rear = rear+1;
```

- The following code is equivalent, but shorter (assuming 0 <= rear < queueSize):

```
rear = (rear + 1) % queueSize;
```

- Do the same for advancing the front index.

Implementing a Queue Class

- When is it full?

3	4				7	9	6
	rear				front		

```
q.enqueue(5);
q.enqueue(2);
q.enqueue(1);
```

Note: enqueue increments rear

3	4	5	2	1	7	9	6
				rear	front		

- It’s full:

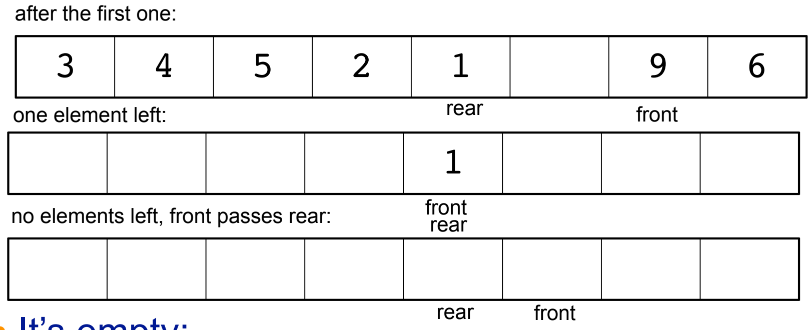
(rear+1)%queueSize==front

Implementing a Queue Class

- When is it empty?

```
int x;
for (int i=0; i<queueSize;i++)
   x = q.dequeue();
```

Note: dequeue increments front



- It's empty:

(rear+1)%queueSize==front

Implementing a Queue Class

- When is it full?   (rear+1)%queueSize==front
- When is it empty? (rear+1)%queueSize==front
- How do we define isFull and isEmpty?
  - Use a counter variable, numItems, to keep track of the total number of items in the queue.
- enqueue: numItems++
- dequeue: numItems--
- isEmpty is true when numItems == 0
- isFull is true when numItems == queueSize

A static queue class

```
class IntQueue
{
private:
   const int QUEUESIZE = 100;  // capacity of the queue
   int queueArray[QUEUESIZE];  // The queue array
   int front;          // Subscript of the queue front
   int rear;           // Subscript of the queue rear
   int numItems;       // Number of items in the queue
public:
   // Constructor
   IntQueue();

   // Queue operations
   void enqueue(int);
   int dequeue();
   bool isEmpty() const;
   bool isFull() const;
   void makeEmpty();
};
```

A static queue class: functions

```
//*********************************************************
// Creates an empty queue of a specified size.           *
//*********************************************************

IntQueue::IntQueue()
{
   front = 0;                // set up bookkeeping
   rear = -1;
   numItems = 0;
}
```

A static queue class: enqueue

```
//******************************************************
// Enqueue inserts a value at the rear of the queue.   *
//******************************************************

void IntQueue::enqueue(int num)
{
    assert(!isFull());

    // Calculate the new rear position
    rear = (rear + 1) % queueSize;

    // Insert new item
    queueArray[rear] = num;

    // Update item count
    numItems++;
}
```

A static queue class: dequeue

```
//******************************************************
// Dequeue removes the value at the front of the       *
// queue and returns the value.                        *
//******************************************************

int IntQueue::dequeue()
{
    assert(!isEmpty());

    //save the result to return
    int result = queueArray[front];

    // Advance front
    front = (front + 1) % queueSize;

    // Update item count
    numItems--;

    // Return the front item
    return result;
}
```

A static queue class: functions

```
//******************************************************
// isEmpty returns true if the queue is empty          *
//******************************************************

bool IntQueue::isEmpty() const {
   return (numItems == 0);
}

//******************************************************
// isFull returns true if the queue is full            *
//******************************************************

bool IntQueue::isFull() const {
   return (numItems == queueSize);
}

//******************************************************
// makeEmpty makes the stack empty                     *
//******************************************************

void IntQueue::makeEmpty() {
   front =  0;
   rear =  - 1;
   numItems = 0;
}
```

A Dynamic Queue Class

- queue_3358_LL.h
  - On the class website
  - Singly-linked-list implementation
  - Templated (all code in *.h file)
  - Requires pointers to both ends of the list

Array vs Linked List implementations

- Both are very fast (O(1)).
- Array may be faster (no dynamic allocation)
- Static arrays:
  - must anticipate maximum size
  - wasted space: entire array is allocated, even if using small portion
- Dynamic arrays (resize when full):
  - resizing takes time (copying all the elements)
  - resizing requires memory that is three times what is needed to store the elements at that time

---

Array vs Linked List implementations

- Linked List:
  - code is actually simpler than array with resizing, especially for queues.
  - space used by elements is always proportional to number of elements (only wasted space is for the pointers)
- Summary:
  - array implementation is probably better for small objects.
  - linked list is probably better for large objects if space is scarce or copying is expensive (resizing)