ADTs: Stacks and Queues

CS 3358
Summer I 2012

Jill Seaman

Introduction to the Stack

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

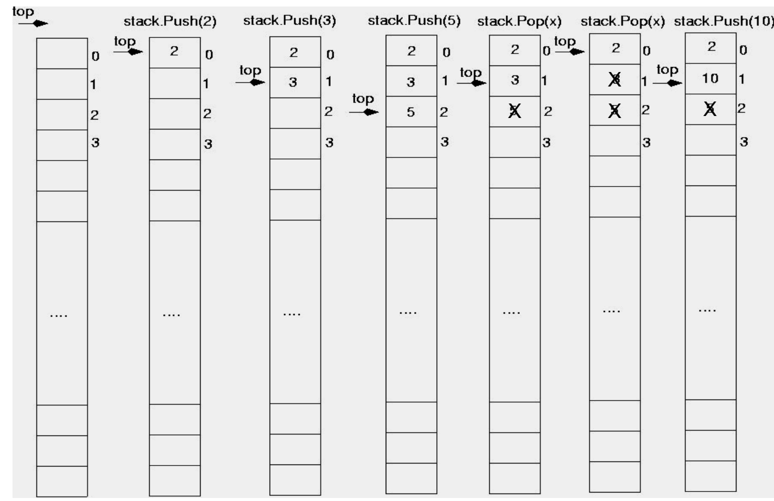
Stack Operations

- Operations:
  - <u>makeEmpty</u>: removes all the elements
- This may 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 Application: Postfix notation

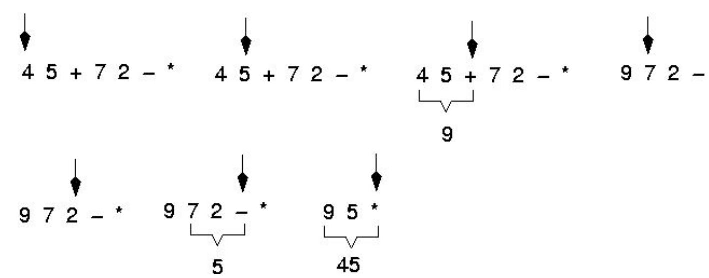
- Postfix notation is another way of writing arithmetic expressions.
- We normally use infix: the operator is between the operands
- In postfix notation, the operator is written after the two operands.

infix: 2+5   postfix: 2 5 +

- Expressions are evaluated from left to right.
- Precedence rules and parentheses are never needed!!

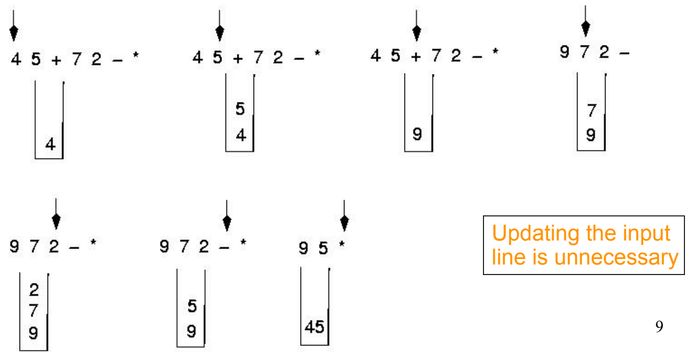
Postfix notation

- evaluation from left to right
- replace evaluated expression with result



Postfix notation: using a stack

- evaluation from left to right: push operands
- for operator: pop two values, perform operation, and push the result



Evaluate Postfix Expression algorithm

- Using a stack:

```
WHILE more input items exist
   get next item from input
   IF item is an operand
     stack.Push(item)
   ELSE
     operand2 = stack.Pop()
     operand1 = stack.Pop()
     Compute result using item as operator
     stack.Push(result)
end WHILE

result = stack.Pop()
```


Implementing a Stack Class

- Array implementation:
  - fixed size or use dynamic arrays
  - fixed arrays: size doesn't change
  - dynamic arrays: can resize as needed in pop
- Linked List
  - grow and shrink in size as needed

A static stack class

```
class IntStack
{
private:
   int *stackArray;  // Pointer to the stack array
   int stackSize;    // The stack size (will not change)
   int top;          // Index to the top of the stack

public:
   // Constructor
   IntStack(int);

   // Destructor
   ~IntStack();

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

This implementation uses a dynamic stack, but it never resizes it once initialized. But it is capable of making a "fixed size" stack of any size.

A static stack class: functions

```
//***********************************************
// Constructor                                  *
// This constructor creates an empty stack. The *
// size parameter is the size of the stack.     *
//***********************************************

IntStack::IntStack(int size)
{
   stackArray = new int[size];  // dynamic alloc
   stackSize = size;            // save for reference
   top = -1;                    // empty
}


//***********************************************
// Destructor                                   *
//***********************************************

IntStack::~IntStack()
{
   delete [] 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
  - print jobs sent to a printer

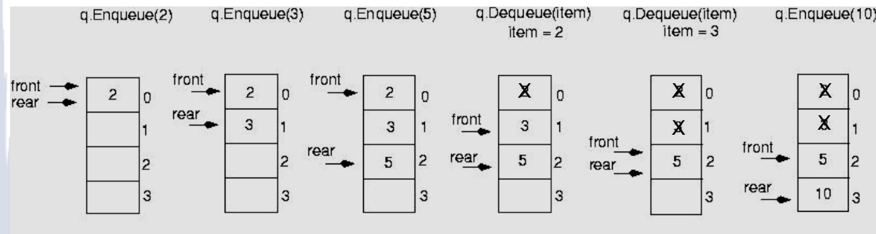
Queue Operations

- 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 may 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 applications of queues involve 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

Queue implemented

- Just like stacks, queues can be implemented using arrays (fixed size, or resizing dynamic arrays) or linked lists (dynamic queues).
- 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		

Queue implemented

solution: wraparound

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

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

- This code is equivalent to the following

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

- Do the same for advancing front index.

- Now, how do we know if the queue is empty or full?

Queue implemented

problem: detecting full and empty queues

- When rear==front, is it full, or size==1?
  - Some implementations offset front or rear by 1.
- An easy solution:
  - Use a counter variable 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

Queue implemented

- In the implementation that follows:
- the queue is a dynamically allocated array, whose size does not change
- front and rear are initialized to -1.
- If the queue is not empty:
  - rear is the index of the last item that was enqueued.
  - front+1 is the index of the next item to be dequeued.
- numItems: how many items are in the queue
- queueSize: the size of the array

A static queue class

```
class IntQueue
{
private:
   int *queueArray;   // Points to the queue array
   int queueSize;     // The queue size
   int front;         // Subscript of the queue front
   int rear;          // Subscript of the queue rear
   int numItems;      // Number of items in the queue
public:
   // Constructors/Destructor
   IntQueue(int);
   IntQueue(const IntQueue &);
   ~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(int s)
{
   queueArray = new int[s]; // dynamic alloc
   queueSize = s;           // save for reference
   front = -1;              // set up bookkeeping
   rear = -1;
   numItems = 0;
}

//********************************************************
// Destructor                                            *
//********************************************************

IntQueue::~IntQueue()
{
   delete [] queueArray;
}
```


A static queue class: functions

```
//********************************************************
// Copy constructor                                      *
//********************************************************

IntQueue::IntQueue(const IntQueue &obj)
{
   // Allocate the queue array.
   queueArray = new int[obj.queueSize];

   // Copy the other object's attributes.
   queueSize = obj.queueSize;
   front = obj.front;
   rear = obj.rear;
   numItems = obj.numItems;

   // Copy the other object's entire queue array.
   for (int count = 0; count < obj.queueSize; count++)
      queueArray[count] = obj.queueArray[count];
}
```

How could the copying be made more efficient?

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 copies t into num.                        *
//******************************************************

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

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

    // Update item count
    numItems--;

    // Retrieve the front item
    return queueArray[front];
}
```


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 =  - 1;
   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)
  - 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)