Ch. 18: ADTs: Stacks and Queues

CS 2308
Fall 2011

Jill Seaman

Lecture 18

Abstract Data Type

- A data type for which:
  - only the properties of the data and the operations to be performed on the data are specific,
  - not concerned with how the data will be represented or how the operations will be implemented.
- An ADT may be implemented by specific data types or data structures, in many ways and in many programming languages.
- Examples:
  - BookInventory (impl'd using array AND linked list)
  - Stacks and Queues

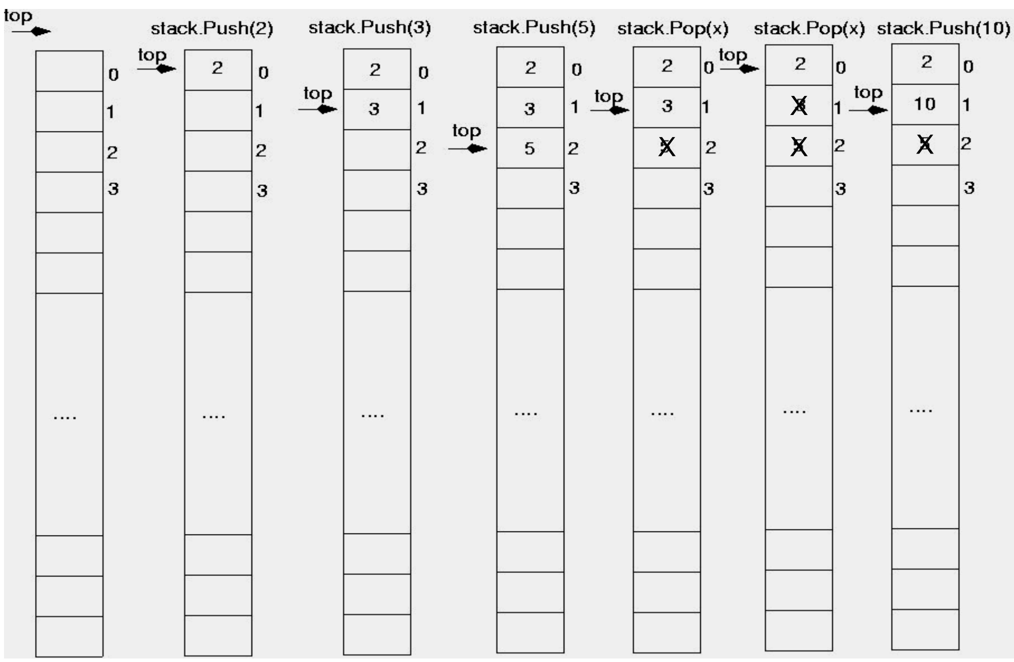
Introduction to the Stack

- 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
- Examples:
  - plates in a cafeteria
  - bangles . . . (bracelets)

Stack Operations

- Operations:
  - push: add a value onto the top of the stack
    - ➡ make sure it's not full first.
  - pop: remove a 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

Stack illustrated



Stack Application: Postfix notation

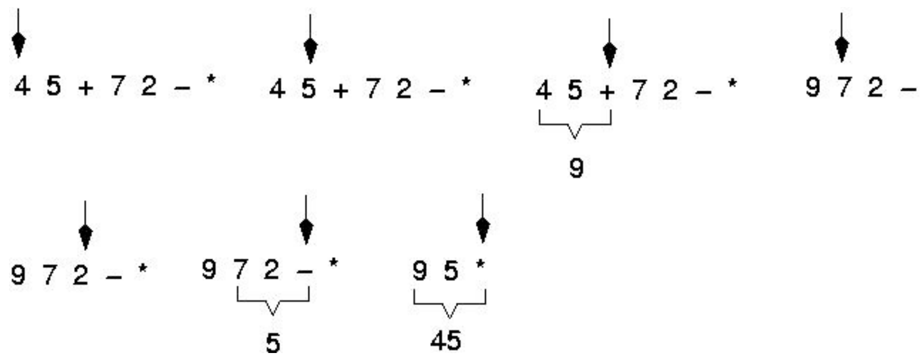
- Postfix notation is another way of writing arithmetic expressions.
- We normally use infix, the operator is between the operands
- In postfix notation, the operator is written after the two operands.

  infix: 2+5   postfix: 2 5 +

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

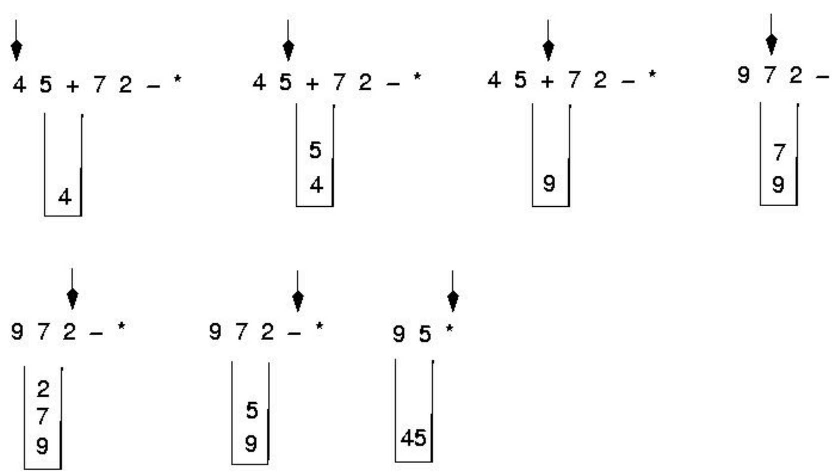
Postfix notation

- 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
    IF item is an operand
      stack.Push(item)
    ELSE
      stack.Pop(operand2)
      stack.Pop(operand1)
      Compute result
      stack.Push(result)
end WHILE

stack.Pop(result)
```


Implementing a Stack Class

- Static stacks:
  - fixed size
  - implemented using arrays
  - uses a dynamically allocated array, but once allocated, the array does not change size
- Dynamic stacks
  - grow in size as needed
  - implemented using linked list

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);
   void pop(int &);
   bool isFull() const;
   bool isEmpty() const;
};
```

const here indicates these functions will not change any of the member variables in the object the functions are called from

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)
{
   if (isFull())
   {
      cout << "The stack is full.\n";
   }
   else
   {
      top++;
      stackArray[top] = num;
   }
}
```


A static stack class: pop

```
//*****************************************************
// Member function pop pops the value at the top     *
// of the stack off, and copies it into the variable *
// passed as an argument.                            *
//*****************************************************

void IntStack::pop(int &num)
{
   if (isEmpty())
   {
      cout << "The stack is empty.\n";
   }
   else
   {
      num = stackArray[top];
      top--;
   }
}
```

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

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

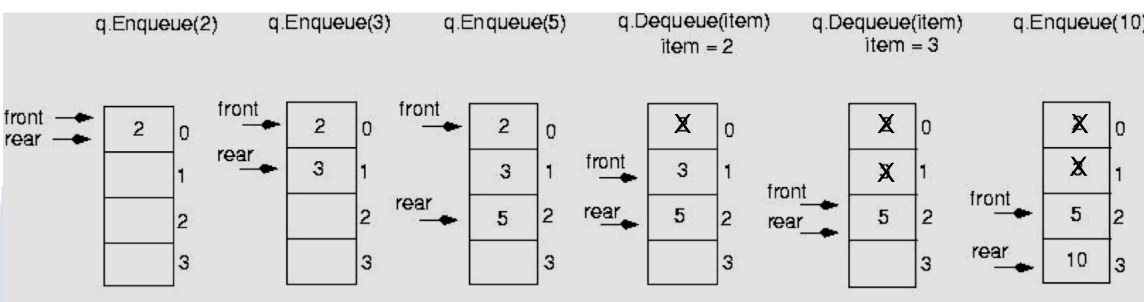

Introduction to the Queue

- 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
- 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

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);
q.dequeue(item); //item is 2
q.dequeue(item); //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 as arrays (static queues) 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? efficiency
- Instead, both front and rear indices move in the array.

Queue implemented

- 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

- 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

- An easy solution for isFull and isEmpty:
  - 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 is 0
- isFull is true when numItems is equal to 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

```
// Specification file for the IntQueue 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:
   // Constructor
   IntQueue(int);

   // Destructor
   ~IntQueue();

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


A static queue class: functions

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

IntQueue::IntQueue(int s)
{
   queueArray = new int[s];
   queueSize = s;
   front = -1;
   rear = -1;
   numItems = 0;
}

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

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

A static queue class: enqueue

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

void IntQueue::enqueue(int num)
{
   if (isFull())
      cout << "The queue is full.\n";
   else
   {
      // 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.                        *
//*****************************************************

void IntQueue::dequeue(int &num)
{
   if (isEmpty())
      cout << "The queue is empty.\n";
   else
   {
      // Move front
      front = (front + 1) % queueSize;
      // Retrieve the front item
      num = queueArray[front];
      // Update item count
      numItems--;
   }
}
```

A static queue class: functions

```
//*****************************************************
// isEmpty returns true if the queue is empty,         *
// otherwise false.                                    *
//*****************************************************

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

//*****************************************************
// isFull returns true if the queue is full, otherwise *
// false.                                              *
//*****************************************************

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