CSC 311

Lectures on
Data Structures

by

Dr. Marek A. Suchenek ©

Computer Science
CSUDH

Copyrighted material

CSC 311

Lecture 10
ADT Stack and ADT Queue

Definitions, Applications, Implementations, Analysis

Linear Data Structures—Stacks and Queues

LIFO FIFO

using stacks to process nested structures

context-free grammar

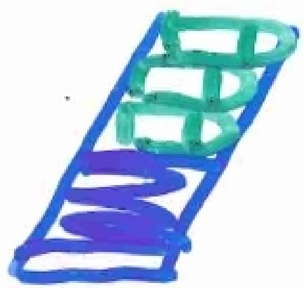
parsing, evaluation, and backtracking

queues, clients, and servers

6.2 Some Background on Stacks

LEARNING OBJECTIVES

1. To learn some of the common terminology.
2. To become informally acquainted with a significant use of stacks in handling recursion, in the theory of push-down automata, and in the theory of parsing, translation, and compiling.



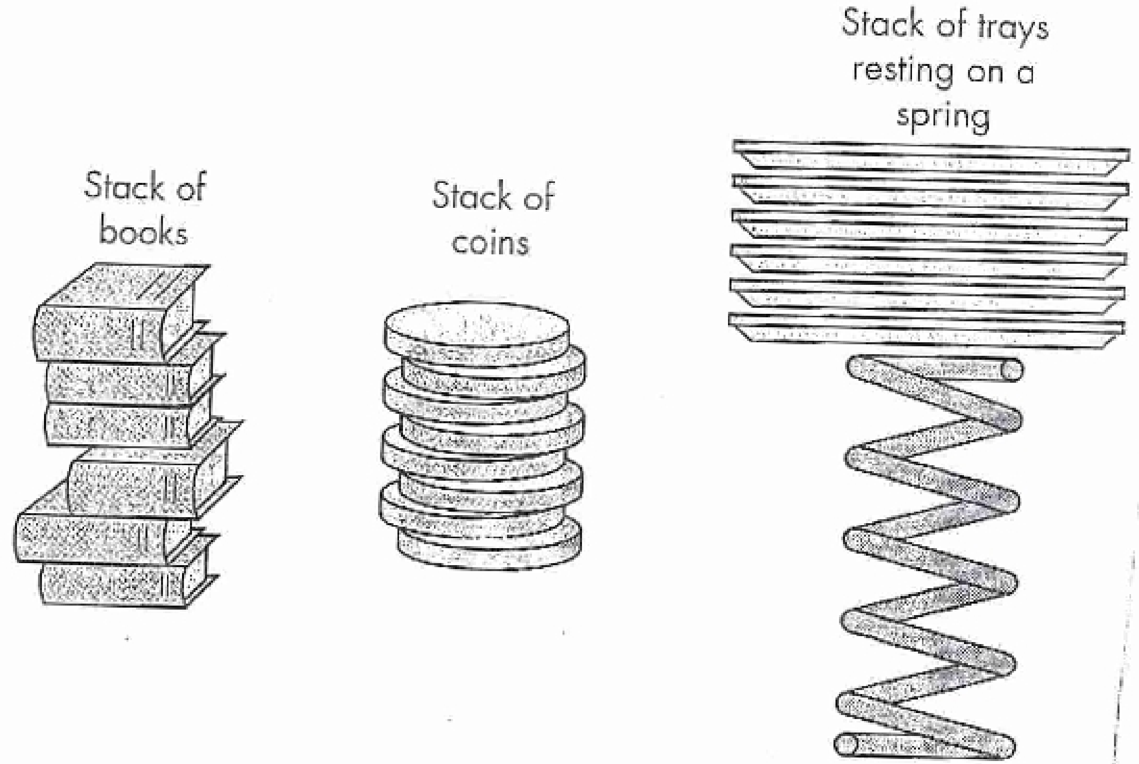


Figure 6.1 Various Kinds of Stacks

Stacks

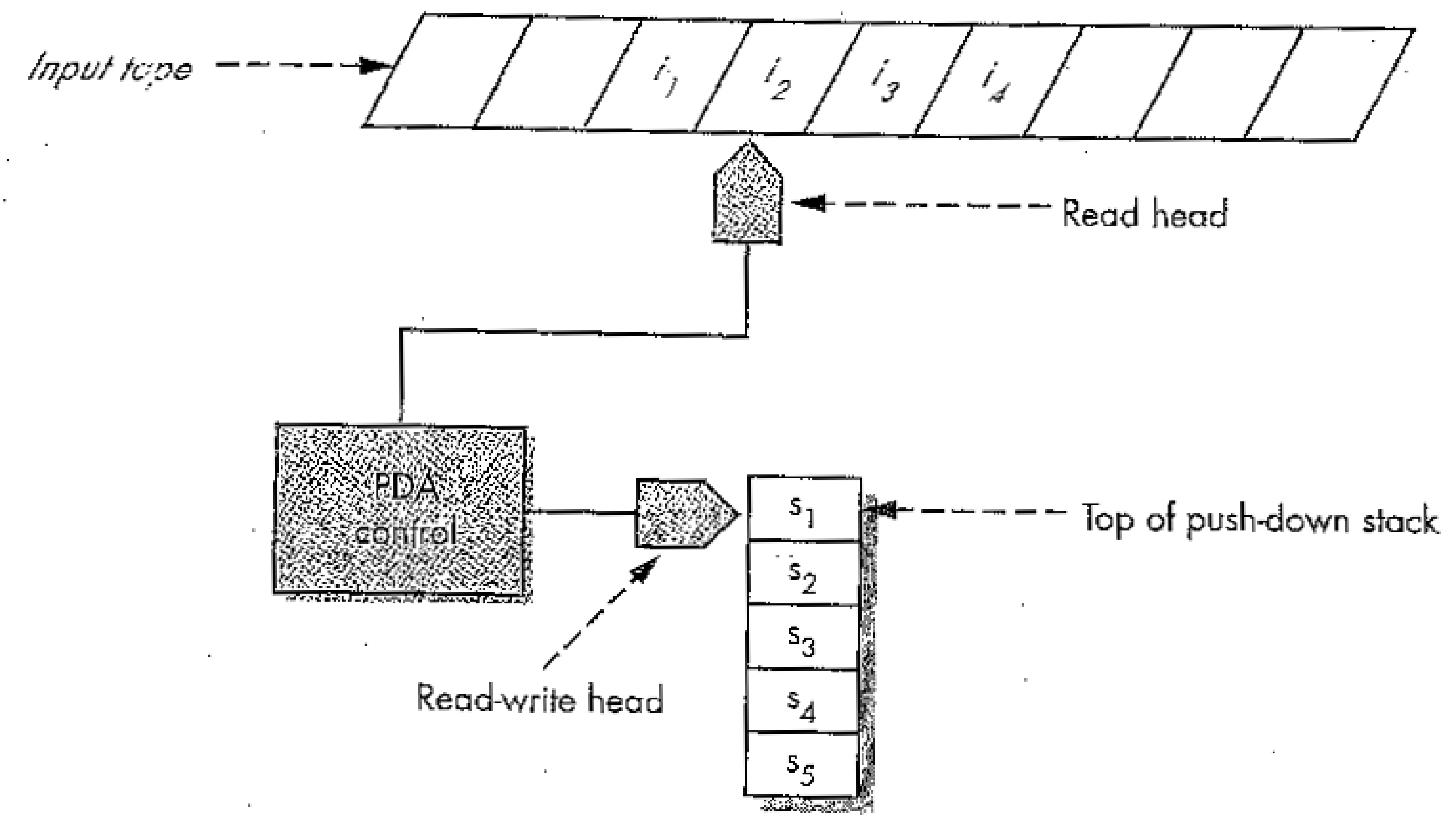


Figure 6.2 Schematic Diagram of a Push-Down Automaton

Stacks

Stacks

A *Stack*, *S*, is a sequence of items on which the following operations are defined:

1. *Construct* an initially empty stack, *S*.
2. Determine whether or not the stack, *S*, is *empty*.
3. *Push* a new item onto the top of the stack, *S*.
4. If *S* is nonempty, *pop* an item from the top of stack, *S*.
5. If *S* is nonempty, *peek* at the top of stack *S* by reading a copy of the top item of *S* without removing it.

Stacks

```
S = new Stack();        // creates an initially empty Stack S

S.empty( );             // a boolean expression that is true if and
                        // only if Stack S contains no items

S.push(X);              // pushes an item X onto the top of Stack S

X = S.pop( );           // removes the top item of S and puts it in X

X = S.peek( );          // puts a copy of the top item of S in X
                        // without removing it from S
```

Figure 6.3 Informal Method Calls in a Stack ADT Interface

Infix Expression	Postfix Expression
(a + b)	a b +
(x − y − z)	x y − z −
(x − y − z)/(u + v)	x y − z − u v + /
$(a^2 + b^2)*(m - n)$	a 2 ^ b 2 ^ + m n − *

Table 6.8 Translations of Expressions into Corresp Postfix Expressions

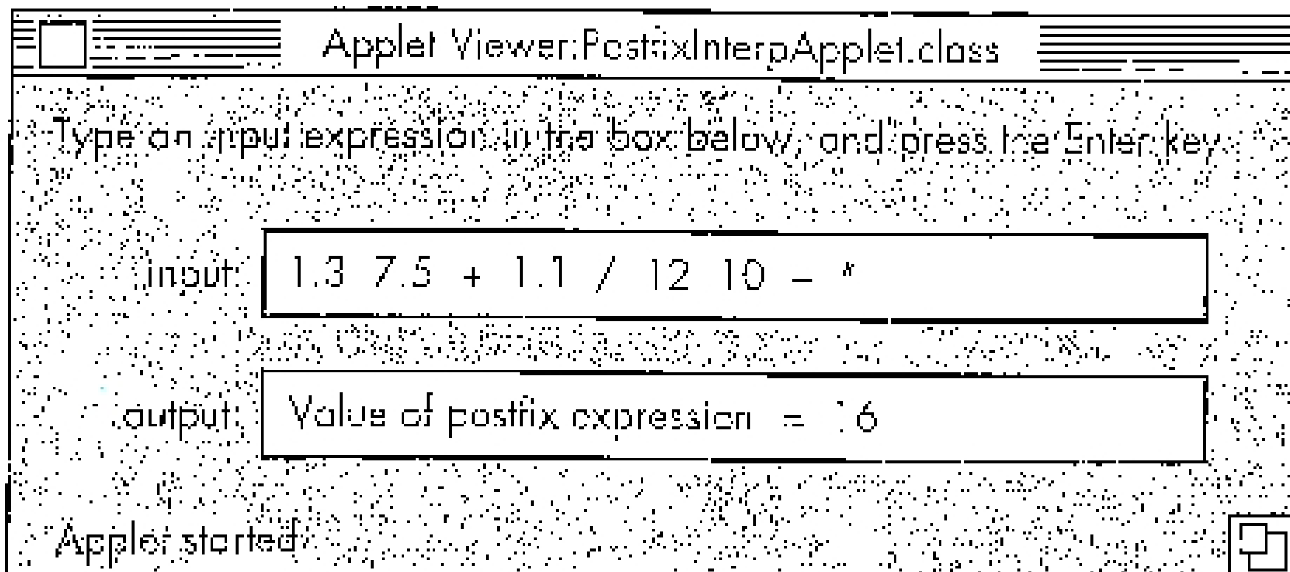


Figure 6.9 Evaluating a Postfix Expression

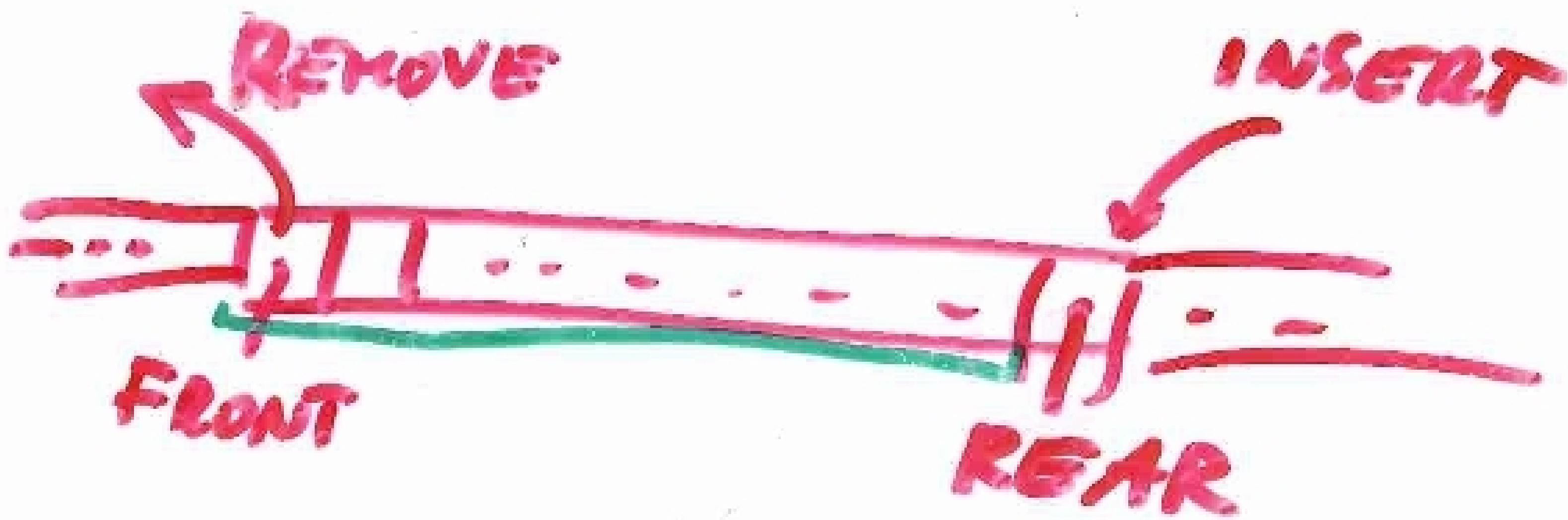
Queues

Queues

A *Queue*, Q, is a sequence of items on which the following operations are defined:

1. *Construct* an initially empty queue, Q.
2. Determine whether or not the queue, Q, is *empty*.
3. *Insert* a new item onto the rear of the queue, Q.
4. Provided Q is nonempty, *remove* an item from the front of Q.

ENQUE
- DEQUE



Queues

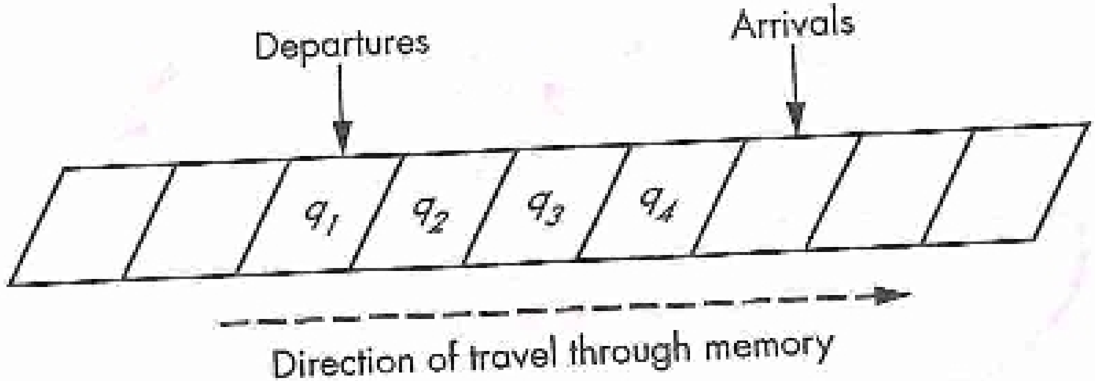
```
Q = new Queue( );          // creates an initially empty Queue Q

Q.empty( );                // a boolean expression that is true if and
                           // only if the Queue Q contains no items
```

enqueue

```
Q.insert(X);               // inserts an item X onto the rear of Queue Q

X = Q.remove( );           // removes an item from the front of Q
                           // and puts it in X
```

dequeue

Figure 6.4 Informal Method Calls for a Queue ADT Interface



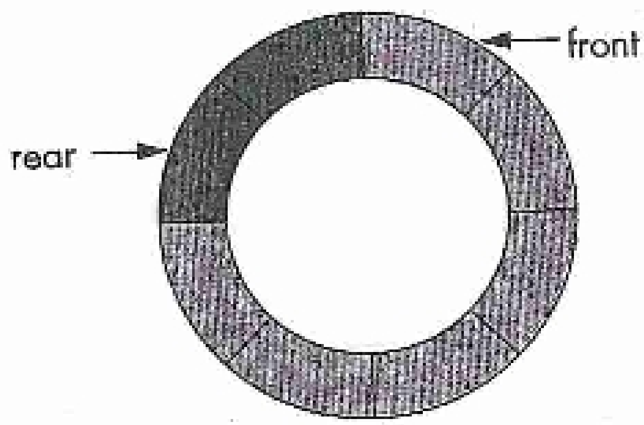
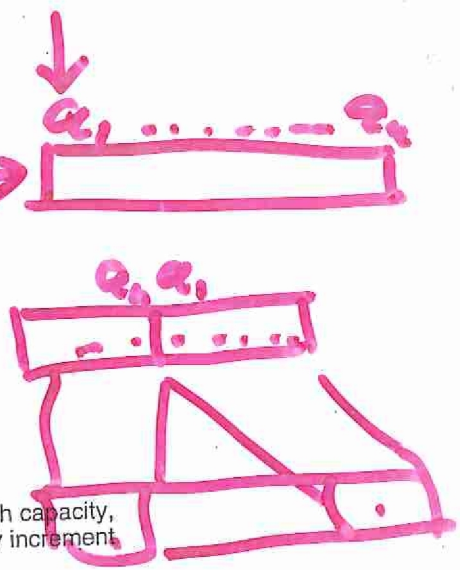


Figure 6.20 Queue on a Circular Track

```
front   =   (front + 1) % n;
rear    =   (rear + 1) % n;
```

Figure 6.21 Incrementing Indexes Using Modular Arithmetic

```
class Queue {

    private int      front;            // the array index of the front item of the queue
    private int      rear;             // the array index for the next item to insert
    private int      count;            // the number of items in the queue
    private int      capacity;         // the number of available array positions
    private int      capacityIncrement; // the amount to increase the capacity
                                        // during array expansion

    private Object[ ] itemArray;        // the array that holds queue items

  /*-----------------*/

    // here, we need the no-arg constructor

    public Queue() {
        front        =   0;
        rear         =   0;
        count        =   0;
        capacity     =  10;
        capacityIncrement = 5;
        itemArray = new Object[capacity];
    }

  /*-----------------*/

    public boolean empty() {
        return (count == 0);
    }

  /*-----------------*/

    public void insert(Object newItem) {

    // if the itemArray does not have enough capacity,
    // expand the itemArray by the capacity increment
      if (count == capacity) {
          capacity += capacityIncrement;
          Object[ ] tempArray = new Object[capacity];
          if (front < rear) {            // if the items are in itemArray[front:rear−1]
              for (int i = front; i < rear; i++) {
                  tempArray[i] = itemArray[i];
              }
          } else {            // otherwise, move the items in two separate sections
              for (int i = 0; i < rear; i++) {                    // section one:
                  tempArray[i] = itemArray[i];               // itemArray[0:rear−1]
              }
              for (int i = front; i < count; i++) {          // and section two:
                                                        // itemArray[front:count−1]
                  tempArray[i+capacityIncrement] = itemArray[i];
              }
              front += capacityIncrement;           // then change front to point to
                                                    // its new position
          }
          itemArray = tempArray;
      }
```



Program 6.22 Circular Queue Representation *(continued)*