

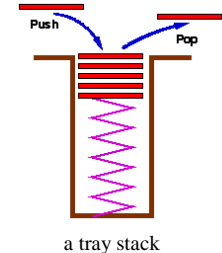
Stacks and Queues

CSE 326 Data Structures Unit 3

Reading: Sections 3.3 and 3.4

Stack ADT

- A list for which Insert and Delete are allowed only at one end of the list (the *top*)
 - › the implementation defines which end is the "top"
 - › LIFO – Last in, First out
- **Push**: Insert element at top
- **Pop**: Remove and return top element (aka **TopAndPop**)
- **IsEmpty**: test for emptiness



2

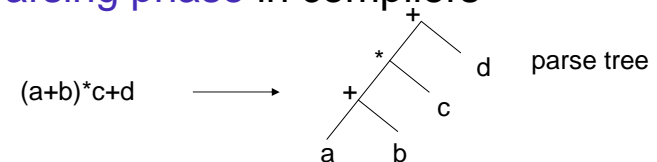
An Important Application of Stacks

- **Call stack** in run time systems
 - › When a function (method, procedure) is called, the work area (local variables, copies of parameters, return location in code) for the new function is pushed on to the stack. When the function returns the stack is popped.
 - › The order we need the data back is 'LIFO'
 - › This explains why calling a recursive procedure with a depth of N requires $O(N)$ space.

3

Another Application of Stacks

- **Parsing phase** in compilers

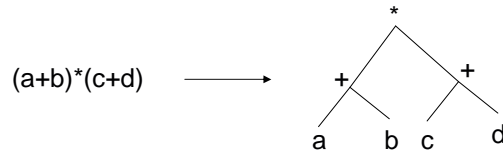


yields the reverse Polish (postfix) notation:

$ab+c*d+$ (traversal of a binary tree in postorder; to be learnt...)

4

Another Application of Stacks



- The reverse Polish (postfix) notation:
 $ab+cd+*$

5

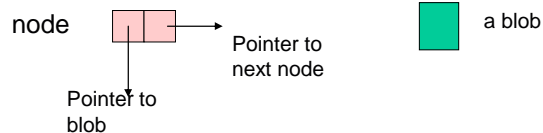
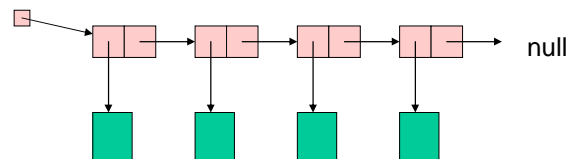
Two Basic Implementations of Stacks

- Linked List**
 - Push** is InsertFront
 - Pop** is DeleteFront (Top is “access” the element at the top of the stack)
 - IsEmpty is test for null (or null after the header if there’s one)
- Array**
 - The k items in the stack are the first k items in the array.

6

Linked List Implementation

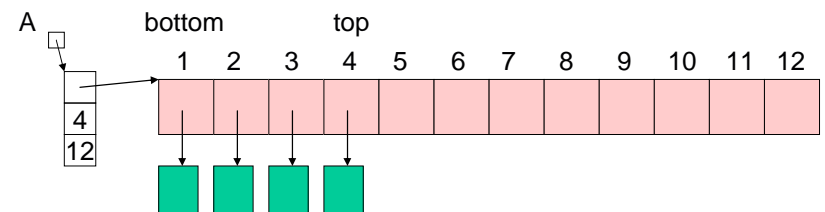
- Stack of blobs






7

Array Implementation

- Stack of blobs



 holder = blob pointer array
 size = number in stack
 maxsize = max size of stack

8

Push and Pop (array impl.)

```
IsEmpty(A : blobstack pointer) : boolean {
    return A.size = 0
}
IsFull(A : blobstack pointer) : boolean {
    return A.size = A.maxsize;
}
Pop(A : blobstack pointer) : blob pointer {
    // Precondition: A is not empty //
    A.size := A.size - 1;
    return A.holder[A.size + 1];
}
Push(A : blobstack pointer, p : blob pointer): {
    // precondition: A is not full//
    A.size := A.size + 1;
    A.holder[A.size] := p;
}
```

9

Linked Lists vs Array

- **Linked list implementation**
 - + flexible – size of stack can be anything
 - + constant time per operation
 - Call to memory allocator can be costly
- **Array Implementation**
 - + Memory preallocated
 - + constant time per operation.
 - Not all allocated memory is used
 - Overflow possible - Resizing can be used but some ops will be more than constant time.

10

Exercise : Find Min

Propose a data structure that supports the stack 'push' and 'pop' operations and a third operation 'find_min', which returns the smallest element in the data structure.

All three operations in $O(1)$ worst case.

11

Exercise : Find Min

12

Queue

- Insert at one end of List, remove at the other end
- Queues are “FIFO” – first in, first out
- Primary operations are **Enqueue** and **Dequeue**
- A queue ensures “fairness”

13

Queue ADT

- Operations:
 - › **Enqueue** - add an entry at the end of the queue (also called “rear” or “tail”)
 - › **Dequeue** - remove the entry from the front of the queue
 - › **IsEmpty**
 - › **IsFull** may be needed

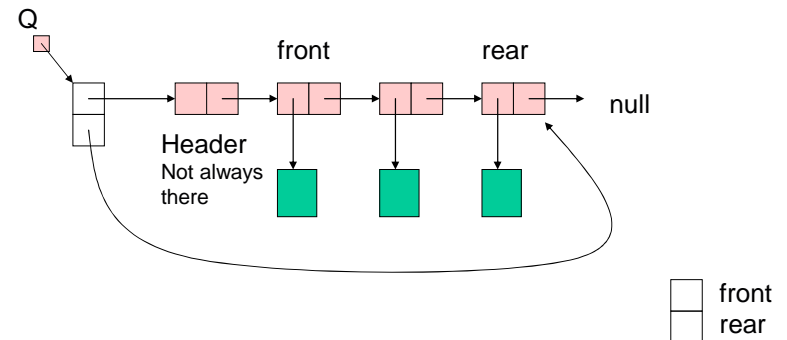
14

A Sample of Applications of Queues

- **Printer Queue**: Jobs submitted to a printer are printed in order of arrival
- **Phone calls made to customer service hotlines** are usually placed in a queue
- **File servers**: Users needing access to their files on a shared file server machine are given access on a FIFO basis

15

Pointer Implementation



16

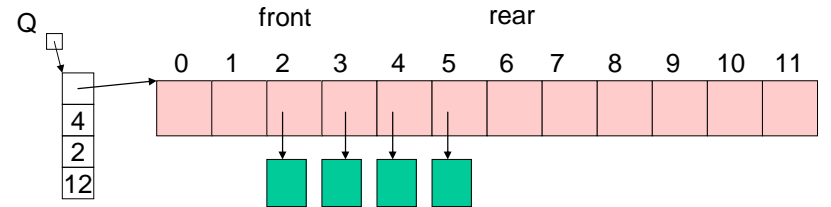
List Implementation

```

IsEmpty(Q : blobqueue pointer) : boolean {
    return Q.front = Q.rear
}
Dequeue(Q : blobqueue pointer) : blob pointer {
    // Precondition: Q is not empty //
    B : blob pointer;
    B := Q.front.next;
    Q.front.next := Q.front.next.next;
    return B;
}
Enqueue(Q : blobqueue pointer, p : blob pointer): {
    Q.rear.next := new node;
    Q.rear := Q.rear.next;
    Q.rear.value := p;
}
    
```

Array Implementation

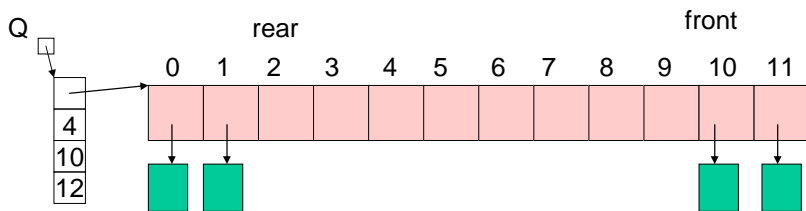
• Circular array



rear = (front + size) mod maxsize
 (the index of the entry after the last occupied one)

holder = blob pointer array
 size = number in queue
 front = index of front of queue
 maxsize = max size of queue

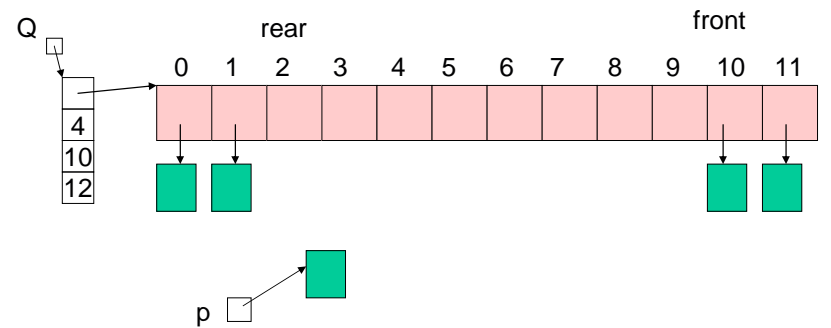
Wrap Around



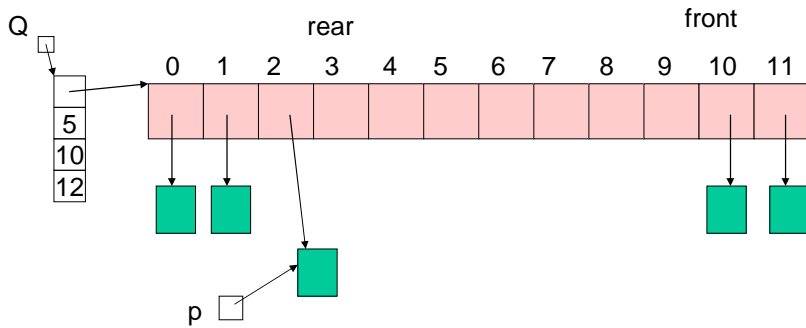
$$\text{rear} = (\text{front} + \text{size}) \bmod \text{maxsize}$$

$$= (10 + 4) \bmod 12 = 14 \bmod 12 = 2$$

Enqueue



Enqueue



21

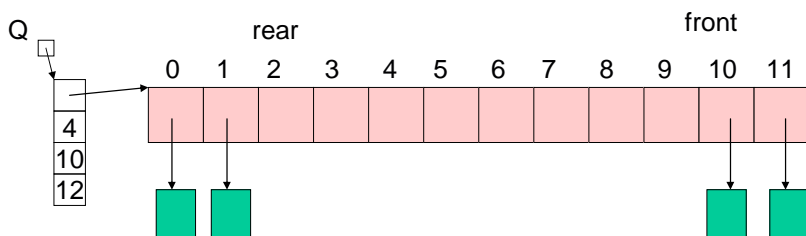
Enqueue

```
Enqueue(Q : blobqueue pointer, p : blob pointer) : {  
  // precondition : queue is not full //  
  Q.holder[(Q.front + Q.size) mod Q.maxsize] := p;  
  Q.size := Q.size + 1;  
}
```

Constant time!

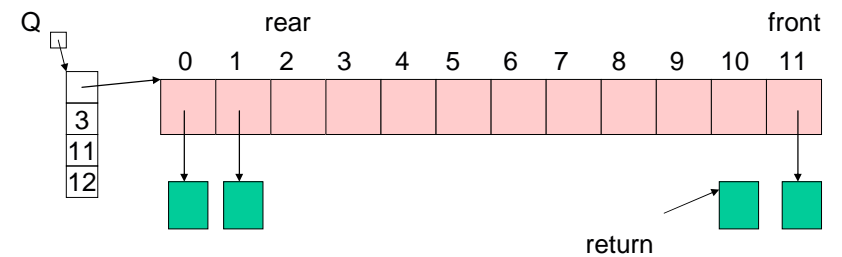
22

Dequeue



23

Dequeue



24

Try Dequeue

- Define the circular array implementation of Dequeue

25

Solution to Dequeue

26