Stacks and Queues

CSE 373
Data Structures
Lecture 6
Readings

• 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
An Important Application of Stacks

- Parsing phase in compilers

$(a+b)*c+d \rightarrow \begin{array}{c}
\text{parse tree} \\
\text{ab+c*d+} \\
\text{traversal of a binary tree in postorder; see Lecture 7}
\end{array}$
Another 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 onto the stack. When the function returns the stack is popped.
  - So, calling a recursive procedure with a depth of $N$ requires $O(N)$ space.
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

- **Array**
  - The k items in the stack are the first k items in the array.
Linked List Implementation

- Stack of blobs
Array Implementation

- Stack of blobs

holder = blob pointer array
size = number in stack
maxsize = max size of stack
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;
}
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.
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”
  › customers waiting on a customer hotline
  › processes waiting to run on the CPU
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
A Sample of Applications of Queues

- File servers: Users needing access to their files on a shared file server machine are given access on a FIFO basis
- 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
Pointer Implementation

Q

Header
Not always there

front
null

rear

front
rear
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

$Q$

$4$

$2$

$12$

$0$ $1$ $2$ $3$ $4$ $5$ $6$ $7$ $8$ $9$ $10$ $11$

front rear

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

$\text{holder} = \text{blob pointer array}$

$\text{size} = \text{number in queue}$

$\text{front} = \text{index of front of queue}$

$\text{maxsize} = \text{max size of queue}$
Wrap Around

\[ \text{rear} = (\text{front} + \text{size}) \mod \text{maxsize} \\
= (10 + 4) \mod 12 = 14 \mod 12 = 2 \]
Enqueue
Enqueue
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!
Deque
Deque

$$\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}$$

$$\begin{array}{cccccccccccc}
3 & 11 & 12 \\
\end{array}$$

$${\text{return}}$$
Try Dequeue

- Define the circular array implementation of Dequeue
Solution to Dequeue

Dequeue(Q : blobqueue pointer) : blob pointer {
  // precondition : queue is not empty //
  p : blob pointer
  p := Q.holder[Q.front];
  Q.front := (Q.front + 1) mod Q.maxsize;
  Q.size := Q.size - 1;
  return p;
}