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

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.

Another Application of Stacks

- **Parsing phase** in compilers

\[(a+b)^c+d\]

yields the reverse Polish (postfix) notation:

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

Another Application of Stacks

- The reverse Polish (postfix) notation: 
  \( ab+cd+^\ast \) 

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.

Linked List Implementation

- Stack of blobs

Array Implementation

- Stack of blobs

\begin{itemize}
  \item \text{holder} = \text{blob pointer array}
  \item \text{size} = \text{number in stack}
  \item \text{maxsize} = \text{max size of stack}
\end{itemize}
Push and Pop (array impl.)

```java
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): {
    // preconditions: 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.

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.
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”

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

- 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

Pointer Implementation
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

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

Wrap Around

rear = (front + size) mod maxsize
= (10 + 4) mod 12 = 14 mod 12 = 2

Enqueue

rear = (front + size) mod maxsize
= (10 + 4) mod 12 = 14 mod 12 = 2
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!

Dequeuue

Dequeue

Dequeue
Try Dequeue

• Define the circular array implementation of Dequeue

Solution to Dequeue