Stacks and Queues

CSE 373
Data Structures
Readings

• Reading Chapter 5
We’ll cover

• Stack ADT
  › Array and linked list implementations
• Queue ADT
  › Circular array and linked list implementations
• Double-Ended Queues
  › Deque implementations
Stack ADT

• A list for which Insert and Delete are allowed only at one end of the list (the top)
  › LIFO – Last in, First out
• isEmpty(); size()
• Push: Insert element at top
• Pop: Remove and return top
• Top (aka peek): return top
• “built-in” class in java.util
Many important applications of Stacks for example

- Parsing phase in compilers

\[
\begin{array}{c}
\quad + \\
\quad + \\
a \quad b \\
\end{array}
\]

yields the reverse Polish (postfix) notation:

\[ab+c*d+\] (traversal of a binary tree in postorder; see forthcoming lecture)
A bit of history

- Polish notation (or prefix notation) introduced by Polish mathematician Jan Lukasiewicz (1878-1956).
- Reverse polish notation (postfix notation) should be called “Zciweisakul”
- Question: What data structure would you use to write a program to go from “lukasiewicz” to “zciweisakul”? 
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 on to 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

• **Array**
  › The k items in the stack are the first k items in the array
  › Push is InsertLast, Pop is DeleteLast, Top is access to the last element of the array

• **Linked List**
  › Push is InsertFront, Pop is DeleteFront, Top is “access” the first element
  › IsEmpty is test for null
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 List Implementation

• Stack of blobs

![Diagram of a linked list stack with nodes and pointers]
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.
ADT Queue

- Insert at one end of List, remove at the other end
- Queues are “FIFO” – first in, first out
- 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 (also called “head” of the queue)
  › **IsEmpty**; **size**
  › **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
Linked list Implementation

Requires a pointer to the front (or head) and a pointer to the rear (or tail). Why do we choose to enqueue at the tail and dequeue at the head?
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; // the value of the element is a blob
  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

front

rear

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

rear = (front + size) mod maxsize

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

Q

4

10

12

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

rear

front

p
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!
Dequeue
Deque queue
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;
}
Double-ended Queue (aka deque)

- List ADT that allows insertions and deletions at both ends
- isempty; size
- Addfirst; addlast; removefirst; removelast
- So best implementations are:
Deque implementations

• Circular array
  › Beware of full and empty conditions
• Doubly linked list