COSC160: Data Structures: Lists and Queues

Jeremy Bolton, PhD

Assistant Teaching Professor

GEORGETOWN UNIVERSITY

Outline

I. Queues
   I. FIFO Queues
      I. Usage
      II. Implementations
   II. LIFO Queues (Stacks)
      I. Usage
      II. Implementations
   III. Applications / Examples

Queues

- Queue is a linear list where data is added and removed at the “ends”
  - Refers to conceptual behavior (not implementation)
    - Specifically ***when (where)*** data enters the queue, and ***when (where)*** data exits the queue
    - Access is generally limited to the top or the bottom of the queue.
  - Types of queues are differentiated by operations performed (behaviors)

- Common Queues
  - First In First Out (FIFO)
    - Applications: traversals, memory buffers, resource sharing, simulations, …
  - Last In First Out (LIFO)
    - Stack
    - Applications: parsers, Runtime environments, traversals, …

FIFO Queue

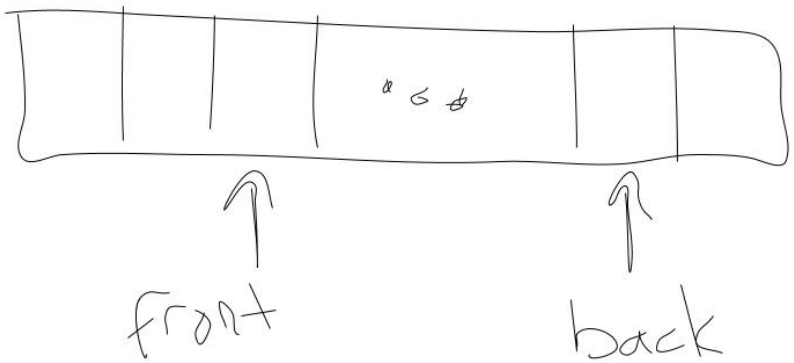
- First In First Out
  - Describes how data is added and removed
    - First come first serve
  - Examples where used:
    - Check-out line at store
    - When entities use a shared resource
    - Memory Buffer
- Operations
  - Insert (item) // inserts item to back of queue
  - Remove () // removes item at front of queue
- Implementation Details
  - Array or chain structure
  - Time Complexity implications

FIFO Queue: Add and Remove

- Enqueue 4 (add to back)
- Enqueue 5 (add to back)
- Dequeue   (remove from front)
- Enqueue 2 (add to back)
- Dequeue (remove from front)
- Dequeue (remove from front)

FIFO Queue: Array Implementation

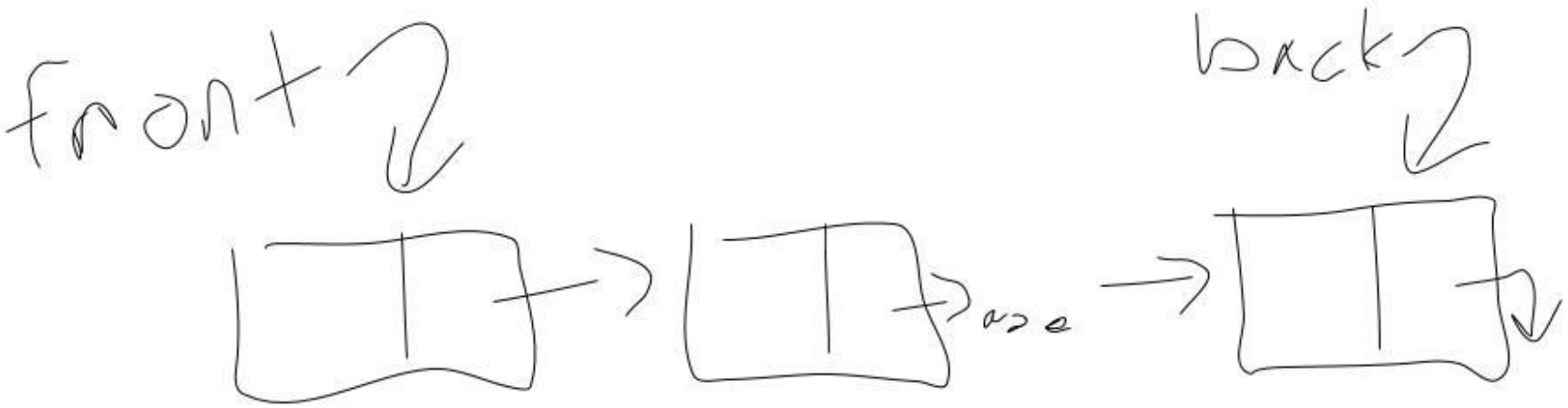
- Considerations
  - Front of Queue
    - Should front of list always be the front of the array
      - Requires shifting after every dequeue
    - Circular, dynamic array is efficient
      - Keep index to front (and back)



- Observations:
  - Since no internal elements in the queue need to be accessed. Both insert (enqueue) and remove (dequeue) operations are efficient.
    - Time Complexity?

FIFO Queue: Linked List Implementation

- Memory Requirements?
- Time?
  - Enqueue
  - Dequeue



LIFO Queue: aka stack

- Last In First Out
  - Describes how data is added and removed
    - Last to be added is first to be removed
  - Examples where used:
    - Push down plates at cafeteria
    - Parsing / grammar analysis
    - Tree or maze traversals

- Operations
  - Insert (to top)
    PUSH(item)
  - Remove (from top)
    POP ()
  - View top
    PEEK()

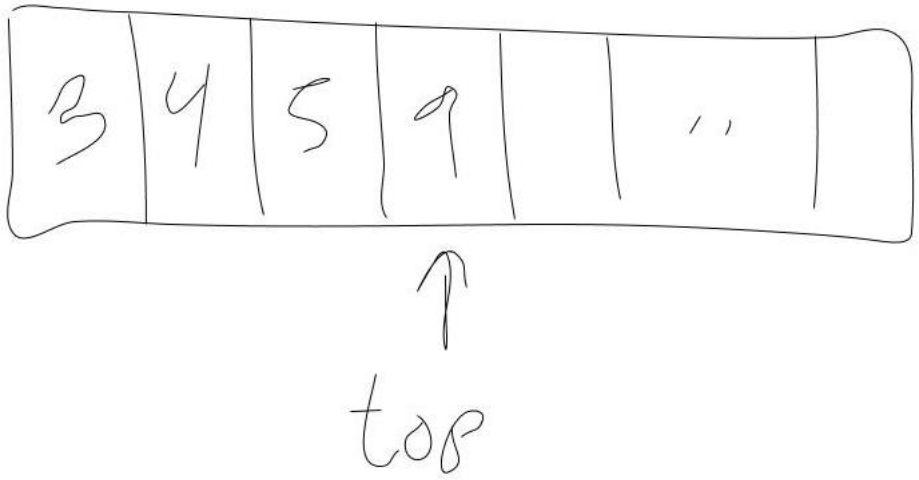
- Implementation Details
  - Array or chain structure
  - Time Complexity implications

LIFO Queue (Stack): Push and Pop

- Push(5)
- Push(10)
- Push(2)
- Pop()
- Push(10)
- Pop()

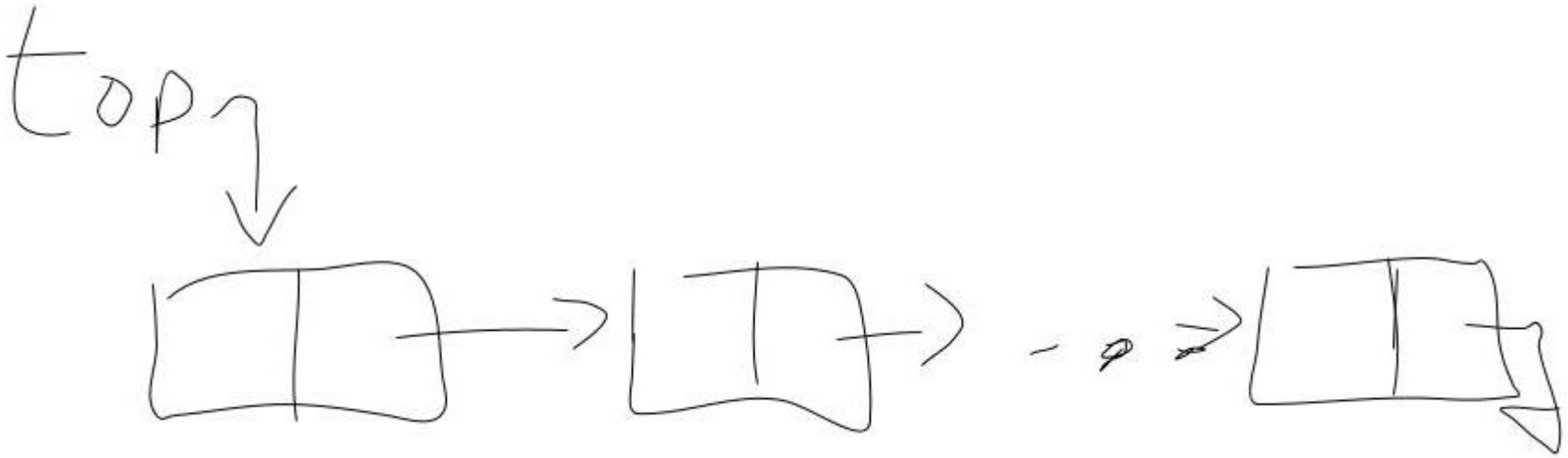
Stack: Array Implementations

- Observations
  - Must allocate size initially
  - No need to keep a base index
  - Maintain a top index



Stack: Linked List Implementations

- Observations
  - Must maintain pointer to top
  - Size changes dynamically

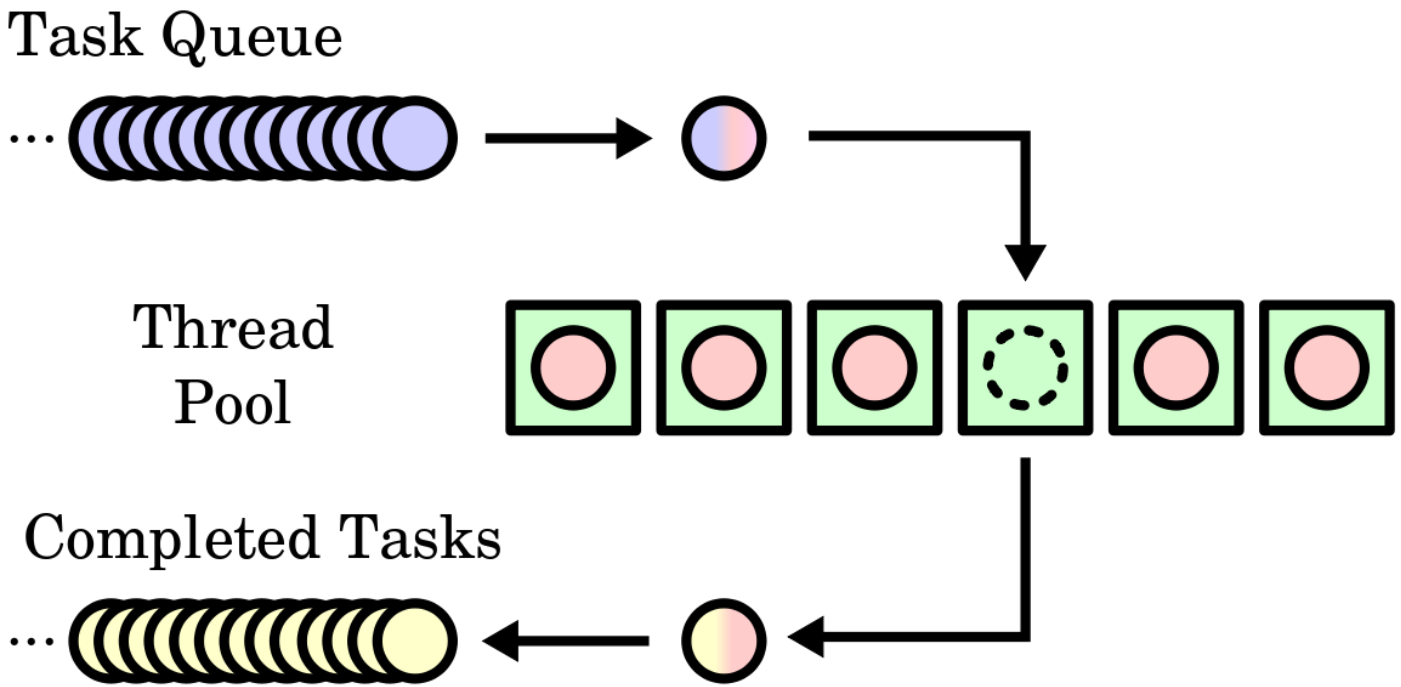


Queue Implementation Summary

- Both Insert and Remove operations for both Queue types are efficient
  - Chaining Implementation permits size to change so that “extra” space is not necessarily allocated. This comes at the cost of one extra pointer per item stored.
  - Array Implementation does not require an extra pointer per item stored, but may use space unnecessarily and may infrequently, require and O(n) reallocation if the Queue is full and an insert is performed.

Application of FIFO Queue

- Shared Resource Queue



Applications of FIFO queues

- Shortest Distance (from a to b) on a grid example

```
FIFO := initialize empty fifo queue
//Square a is at grid location (0,0)
a.distanceThusFar := 0
FIFO.enqueue(a)
while ( ! FIFO.isEmpty()  )
     x := FIFO.dequeue()
     for each neighboring square n of square x
          if square was not previously encountered OR if n.distanceThusFar > x.distanceThusFar + 1
               n.distanceThusFar := x.distanceThusFar + 1
               if n == b // done
                    FIFO.clear()
                    return n.distanceThusFar
               FIFO.enqueue(n); and mark square n as encountered
```

a	1	2	3	4	5
1	1	2	3	4	5
2	2	2	*	4	5
3	3	3	*	5	5
4	4	*	*	6	6
5	5	5	6	7	b

GEORGETOWN UNIVERSITY

Application of Stacks

- Symbol matching (for syntax)
  - EG: Are all the ‘(‘ and ‘{‘ properly matched with ‘)’ and ‘}’

```
function f ( arg )
        {
                while (arg > 0)
                {x = arg --;}
        }
```

- Symbol matching algorithm
  - Scan input code (character array) from left to right

```
for i from 0 to inputLength - 1
     if input[ i ] == `(` OR input[ i ] == `{` , then stack.push(input[ i ])
     else if input[ i ]  == `)` OR input[ i ] == `}` then
          c : = stack.pop()
          if c is `(` and input[ i ] is `)’ OR if c is `{` and input[ i ] is `}’
               //The symbols are matched!, do nothing
          else // the symbols are unmatched
               throw a syntax error

if traversal complete and stack.isEmpty() == false, then
     Throw error, no matching close symbol found
```

GEORGETOWN UNIVERSITY