CS 310: Stack and Queue (Part II)

Connor Baker

February 14, 2019

Review: Stack/Queue

- Restricted operations give us good worst cases

    - $O(1)$ for all supported operations
    - $O(n)$ space

- Simple data structures

    - Focus on limited operations
    - Can be made out of primitive data structures (arrays, linked lists, etc.)

- Good for representing time-related data

    - Call stack
    - Packet queues

Big-O Comparison

- Stack

Implementation	push()	pop()	top()	isEmpty()	size
Array	1*	1	1	1	1
Linked List	1	1	1	1	1

*Amortized analysis

- Queue

Implementation	enqueue()	dequeue()	getFront()	isEmpty()	size
Array	1*	1	1	1	1
Linked List	1	1	1	1	1

*Amortized analysis

Warm-up

- What does this method do?

```java
class Node {
    int data;
    Node next;
}
void method(Node c) {
```

```java
    Stack<Node> stack = new Stack<>();
    while (c != null) {
        stack.push(c);
        c = c.next;
    }
    while (stack.size() > 0) {
        System.out.println(stack.pop().data);
    }
}
```

Review: Queues

- FIFO

- Supported operations:

    - `enqueue(x)`: insert at the tail
    - `dequeue()`: remove from head
    - `getFront()`: return head contents
    - `size()`: returns the size of the queue
    - `isEmpty()`

- Applications:

    - Simulate a process with a FIFO order
    - Scheduling queue of a CPU or disk or printer
    - Serve as a buffer for file I/O, network communications, etc.

Priority Queues

- Much of the time tasks that we use a queue for have different priorities

    - It is convention that the lower the priority, the better
    - Symmetric code if higher is better
    - Dequeue the ones with the "best" priority first

- Common priority queue operations

    - `void insert(T x, int p)`: insert x with priority p
    - `T findMin()`: return the object with the "best" priority
    - `deleteMin()`: remove the object with the "best" priority

Practice

- How can we implement a priority queue with the data structures that we've discussed so far?

    - How can we implement the operations associated with a priority queue (like `add(x)`, `findMin()`, and `deleteMin()`)?
        * What would the $O(n)$ of those operations be?

- Candidates: dynamic arrays and linked lists

Unsorted List

- `add()` and `enqueue()`: same as normal queue

  - Append to the end
  - Which end depends on the underlying data structure

- Dequeue the best priority

  - Search for the one with the best priority and remove
  - Shift if needed

- Can be implemented with either dynamic array or linked list

Sorted List

- Idea: keep items sorted based on their priorities

- Perform sorted insertion

  - Which end keeps the best priority?
    * With a dynamic array, probably the end
    * With a linked list, the head

- Dequeue the best priority from wherever is appropriate

Multiple Queues

- Have one queue per priority level

  - Fixed number of priorites (like high/medium/low)

- `enqueue(T x, Queue p)`

  - Add to the end of the queue corresponding to `p`

- `dequeue()` and `peek()`

  - Search for a non-empty queue with the best priority

Priority Queue Design

Data Structure	`enque()`	`peek()`*	`dequeue()`*	`Notes()`
Unsorted List	$O(1)$	$O(n)$	$O(n)$	best priority at any location
Sorted Array	$O(n)$	$O(1)$	$O(1)$	best priority at high index
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$	min at head or tail
Multiple Queues	$O(1)$	$O(m)$	$O(m)$	

*Using the best priority

- n: number of items in queue

- m: number of priority levels

Priority Queue

- There are other ways that we can implement priority queues:

  - Binary search trees
  - Heaps
  - And others, all of which we'll look at later in the semester

Summary

- Stacks and queues

  - Try implementing them
  - Project 2

- Next lecture: Trees, recursion

  - Reading: Chapter 18, Chapter 7

Extra: Interview Questions

- Assume that you only have a stack data structure available; how do you implement a queue? (Hint: you need two stacks.)

- How would you use queues to implement a stack?

- Design a special stack which has the following $O(1)$ operations: (there is no space requirement)

  - `push()`
  - `pop()`
  - `min()` (returns the smallest value in the stack

- Describe an algorithm to sort a stack in ascending order using only a second stack and a temporary variable (Hint: Tower of Hanoi)

  - Assume normal stack implementation with only `push()`, `pop()`, `peek()`, and `isEmpty()`