CSE 250 Recitation

March 5 - 6: Stacks/Queues

Stacks vs Queues

Exercise:
What does the following code print when MysterySequence is a Stack? Queue?

What are the relevant operations for each?

What are their runtimes for different backing **data structures**?

```
MysterySequence seq = new MysterySequence()
seq.addSomething("A")
seq.addSomething("B")
seq.addSomething("C")
seq.addSomething("D")
print(seq.removeSomething())
print(seq.removeSomething())
print(seq.removeSomething())
seq.addSomething("E")
print(seq.removeSomething())
seq.addSomething("F")
print(seq.removeSomething())
seq.addSomething("G")
seq.addSomething("H")
print(seq.removeSomething())
print(seq.removeSomething())
print(seq.removeSomething())
```

Stacks vs Queues

Stack (LIFO)
==========
Prints: DCBEFHGA
Operations: push, pop

LinkedList Impl:
push O(1)
pop O(1)

ArrayList Impl:
push amortized O(1)
pop O(1)

```
MysterySequence seq = new MysterySequence()
seq.addSomething("A")
seq.addSomething("B")
seq.addSomething("C")
seq.addSomething("D")
print(seq.removeSomething())
print(seq.removeSomething())
print(seq.removeSomething())
seq.addSomething("E")
print(seq.removeSomething())
seq.addSomething("F")
print(seq.removeSomething())
seq.addSomething("G")
seq.addSomething("H")
print(seq.removeSomething())
print(seq.removeSomething())
print(seq.removeSomething())
```

Stacks vs Queues

Queue (FIFO)
==========
Prints: ABCDEFGH
Ops: enqueue,dequeue

LinkedList Impl
enqueue O(1)
dequeue O(1)

RingBuffer Impl
enqueue amortized O(1)
dequeue O(1)

```
MysterySequence seq = new MysterySequence()
seq.addSomething("A")
seq.addSomething("B")
seq.addSomething("C")
seq.addSomething("D")
print(seq.removeSomething())
print(seq.removeSomething())
print(seq.removeSomething())
seq.addSomething("E")
print(seq.removeSomething())
seq.addSomething("F")
print(seq.removeSomething())
seq.addSomething("G")
seq.addSomething("H")
print(seq.removeSomething())
print(seq.removeSomething())
print(seq.removeSomething())
```

Queue Exercise #1

Exercise: Give pseudocode for an implementation of the **`Queue`** ADT using a **singly `LinkedList`** that **only has a head reference**.

First: What are the methods of the **`Queue`** ADT?

Queue Exercise #1

Exercise: Give pseudocode for an implementation of the **`Queue`** ADT using a **singly `LinkedList`** that **only has a head reference**.

First: What are the methods of the **`Queue`** ADT? **`enqueue`** **`dequeue`** **`peek`**

`enqueue(elem)`: Adds **`elem`** to back of the queue

`dequeue()`: removes the front of the queue and returns it

`peek()`: returns the element at the front of the queue

Queue Exercise #1

Discussion:

Runtimes?

How could efficiency be improved?

```
Assume LinkedList head pointer is head

enqueue(elem):
    if !head.isPresent():
        head = new LLNode(elem)
    else:
        temp = head
        while temp.next.isPresent():
            temp = temp.next
        temp.next = new LLNode(elem)

dequeue():
    val = head.value
    head = head.next
    return val

peek(): return head.val
```

Queue Exercise #1

Discussion:

Runtimes?
O(n), O(1), O(1)

How could efficiency be improved?
Keep a tail pointer (but we don't **need** it)

```
Assume LinkedList head pointer is head

enqueue(elem):
    if !head.isPresent():
        head = new LLNode(elem)
    else:
        temp = head
        while temp.next.isPresent():
            temp = temp.next
        temp.next = new LLNode(elem)

dequeue():
    val = head.value
    head = head.next
    return val

peek(): return head.val
```

Queue Exercise #2

Exercise: Try to come up with a solution to efficiently implement the Queue ADT but using TWO singly `LinkedList` objects to store the data.

Hints:

- You should be able to get runtimes comparable to the ones from lecture
- Think of the two `LinkedLists` as `Stacks`...
  - How does the LIFO order of `Stacks` compare to FIFO order of `Queues`
  - How can you get FIFO order when using two `Stacks`?
- Don't worry about pseudocode...try drawing it instead

Queue Exercise #2

Discussion:

Runtimes?

Why is this neat?

Assume **Stacks** are named **in** and **out**

```
enqueue(elem):
    in.push(elem)

dequeue():
    if out.isEmpty():
        while !in.isEmpty():
            out.push(in.pop())
    return out.pop()

peek():
    if out.isEmpty():
        while !in.isEmpty():
            out.push(in.pop())
    return out.peek()
```

Queue Exercise #2

Discussion:

Runtimes?

O(1) for enqueue

O(n), Amortized O(1) for dequeue and peek

Why is this neat?

We only need singly linked lists!

```
Assume Stacks are named in and out

enqueue(elem):
    in.push(elem)

dequeue():
    if out.isEmpty():
        while !in.isEmpty():
            out.push(in.pop())
    return out.pop()

peek():
    if out.isEmpty():
        while !in.isEmpty():
            out.push(in.pop())
    return out.peek()
```

Data Structure Design Scenarios

ADTs: Sequence. List, Set, Bag, Queue, Stack
Data structures: Array, Arraylist, Linked List, Ring Buffer

You are designing data structures for a Print Job Manager:

1. Design a data structure to manage print jobs. An arbitrary number of print jobs can be added at any time, and the print jobs must be executed in the order they were received.

2. Each print job has a unique Job ID, and the system frequently checks if a Job ID exists before adding a job. Design a data structure to facilitate these checks.
   Bonus: If we want to be able to cancel jobs based on ID, how can we use this in conjunction with your answer for 1?

3. Users can also set new configuration settings for the printer, and the system should keep track of all the configuration changes made. The user should be able to “Undo” the changes to the configuration. Design a data structure to keep track of the configuration changes.

Discussion: What ADT and data structure would be best suited for each scenario?

Data Structure Design Scenarios

ADTs: Sequence. List, Set, Bag, Queue, Stack
Data structures: Array, Arraylist, Linked List, Ring Buffer

You are designing data structures to manage user data for a large company:

1. In the first application, the program frequently needs to retrieve user records at arbitrary positions (e.g., "get the 5000th element") as fast as possible, but the data set is fairly static. Insertions and deletions happen, but rarely.

2. In the second application, the program needs to support continuous appending of new user records as users sign up for the system. The total number of records is unbounded, and the company wants to avoid any delays during the user sign up process.

3. In the third application, admins need to frequently look users up based on their unique user ID. These searches should be as quick as possible

Discussion: What ADT and data structure would be best suited for each scenario?