8. Stacks and Queues

https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html

Sunday, May 15, 2022 5:52 AM

https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html

- Linear data structure
  - traverse with only 1 element can be reached
    - sequentially one by one
  - push, peak, pop
- higher level vs lower

Stacks (LIFO)

Last in First out

- Browser history (open last closed browser)
- ctrl + z, undo in word

analogy to a stack of plates
- first plate gets removed

Lookup	O(n)
pop	O(1)
push	O(1)
peek	O(1)

Queues (FIFO)

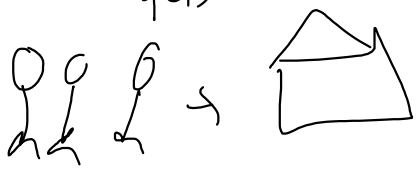
First in First out

- waiting area to buy concert tickets
- reservation

- restaurant check in, first to make ...
- printer app

* lookup O(1)
enqueue O(1) push
dequeue O(1) pop
Peek O(1)

analogy to a line at a restaurant, first person gets served



Stacks vs queues

building w/

arrays • fast b/c memory index is close to each other Linked List • more dynamic size, needs memory for pointer	arrays • bad b/c shifting of indexes Linked List • same as stacks

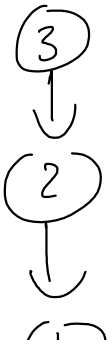
---

Stack Implementation w/ Linked List (LIFO)

Node
data
next

Stacky
push
peek
pop

main
{
Test
}



next peek pop

```
public class Stacky {
   Node top;
   Node bottom;
   int length = 0;

   public Node peek() {
      if (top == null) {
         System.out.println("Stack is Empty");
         return null;
      }
      else {
         System.out.println("Peek " + top.data);
         return top;
      }
   }

   public Node push(int data){
      Node addNode = new Node(data);
      if (this.length == 0) {
         top = addNode;
         bottom = addNode;
      } else {
         // New node will point to previous top, add the new node as the
top
         // Because Stack is LIFO
         addNode.next = top;
         top = addNode;
      }
      length++;
      System.out.println("Push: " + top.data);
      return top;
   }

   public Node pop(){
      if (top == null) {
         System.out.println("Stack is empty");
         return null;
      }
      Node delNode = top;
      top = top.next;
      System.out.println("Popped: " + delNode.data);
      length--;
      return delNode;
   }
}
```

```
public class Node {
   int data;
   Node next;

   Node(int d) {
      data = d;

   }
}
```

```
public class Main {

   public static void main(String[] args) {
      Stacky stack = new Stacky();

      stack.push(5);
      stack.push(10);
      stack.push(20);
      stack.pop();
      stack.peek();
      stack.pop();
      stack.pop();
      System.out.println("length: " + stack.length);
   }
}
```

addNode.next

addNode = top

```
Push: 5
Push: 10
Push: 20
Popped: 20
Peek 10
Popped: 10
Popped: 5
length: 0
```

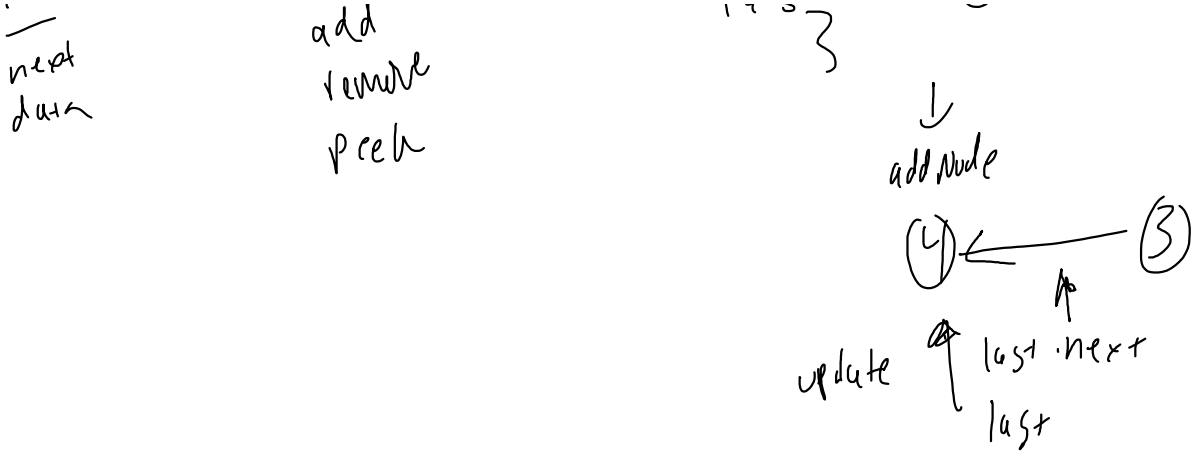
Queue Implementation Linked List (FIFO)

Node
next

Queue
add
remove

main
Test

Last
First
3 ← 2 ← 1




```
public class testQueue {
   Node first;
   Node last;
   int length = 0;

   public Node peek() {
      System.out.println("Peek: " + first.data);
      return first;
   }

   public Node add(int d){
      Node addNode = new Node(d);

      // if queue is empty, this added node should be the first and last
node
      if (first == null) {
         first = addNode;
      }
      // If this isn't the first node ever added, then previous node before
should point to this new node
      if (last != null) {
         last.next = addNode;
      }

      // FIFO, the new node should be in the back of the line
      last = addNode;

      System.out.println("Added node is: " + addNode.data);
      length++;
      return first;
   }

   public Node remove(){
      if (first == null) {
         System.out.println("Queue is empty, nothing to remove");
         return null;
      }

      // Temp placeholder to show removed node
      Node removedNode = first;
```

```
public class Node {
   int data;
   Node next;

   Node(int d) {
      data = d;
   }
}
```

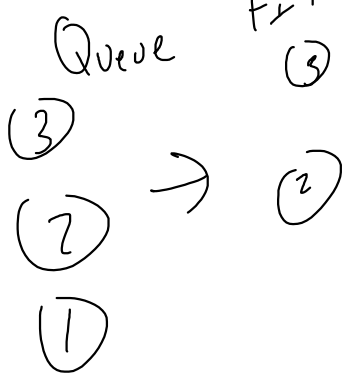
```
 public class Main {

    public static void main(String[] args) {
      testQueue queue = new testQueue();
      queue.add(2);
      queue.add(5);
      queue.add(10);
      queue.remove();
           queue.remove();
           queue.peek();
           queue.remove();
      System.out.println("Length: " +
      queue.length);
    }
 }
```

```
Added node is: 2
Added node is: 5
Added node is: 10
Removed: 2
Removed: 5
Peek: 10
Removed: 10
Length: 0
```

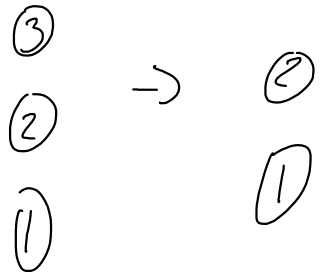
```
    // FIFO, the new first is set next to removedFirst
    first = first.next;

    // If removed node was the last in queue, then set last to null too
    if (first == null) {
      last = null;
    }
    System.out.println("Removed: " + removedNode.data);
    length--;
    return removedNode;
  }
}
```

Implement Queue using Stack Practice

Queue FIFO



Stack LIFO (2 stacks)



Approach:

use one to keep inputs and the other for outputs

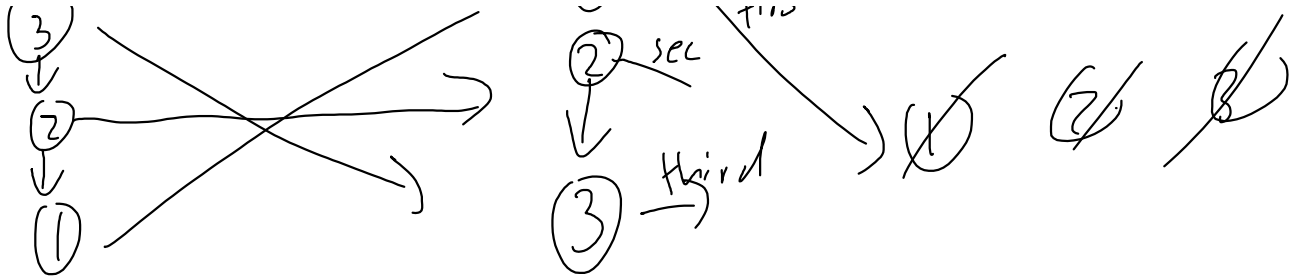
input stack - to push

output stack - pop & peek

empty input stack to output stack when pop is called then pop output stack for FIFO

e.g.

Stack In — pop LIFO — Stack Out — pop LIFO — becomes FIFO



*only refill output stack when it's empty again so you can pop any remaining elements in output stack in FIFO order

```
class MyQueue {

    public MyQueue() {

    }

    // 2 LIFO ----> FIFO
    // One stack handles push, the other handles pop/peek
    // When pop is needed, empty input stack to output stack, and let it
pop out in FIFO order
    Stack<Integer> stackIn = new Stack<Integer>();
    Stack<Integer> stackOut = new Stack<Integer>();
    // Temp declaration in case nothing was ever popped
    int trackTop;

    public void push(int x) {
        if (stackIn.isEmpty()) {
            trackTop = x;
        }
        stackIn.push(x);
    }

    public int pop() {
        // Empty input stack to output stack, it becomes FIFO order
        if (stackOut.isEmpty()) {
        while (!stackIn.isEmpty()){
            stackOut.push(stackIn.pop());
        }
        }

         // Pop the FIFO'd output stack
        return stackOut.pop();
    }

    public int peek() {
        // No values were popped yet, so just return the tracked top
        if (stackOut.isEmpty()) {
            return trackTop;
        }
        return stackOut.peek();
```

O(1) time b/c O(1) for push and O(1) for pop amortized over a lot of

```
            return trackTop;
        }
        return stackOut.peek();
    }

    public boolean empty() {
        return stackIn.isEmpty() && stackOut.isEmpty();
    }
}
```

and O(1) for pop amortized (on avg over a lot of operations)

O(1) Space b/c space was constant

Better Solution Found:

```
class MyQueue {

    Stack<Integer> input = new Stack();
    Stack<Integer> output = new Stack();

    public void push(int x) {
        input.push(x);
    }

    public void pop() {
        peek();
        output.pop();
    }

    public int peek() {
        if (output.empty())
            while (!input.empty())
                output.push(input.pop());
        return output.peek();
    }

    public boolean empty() {
        return input.empty() && output.empty();
    }
}
```

Summary:

Pros

Fast insert & remove

Cons

Slow lookup

Fun peek
ordered