$f(n) \in O(g(n))$

iff

starting point

constant term

$\exists\, \underline{n_0},\ \underline{c}$

s.t.

$f(n) \leq c \cdot g(n),\ n \geq n_0$

$$6n^2 \le 6n^2$$

$$3n^2 + 2n^2 + n^2 \le 6n^2$$

$$3n^2 + 2n^2 + n^2 \ge 3n^2 + 2n + 1, n \ge 1$$

$$1 \le n, \; n \ge 1$$

$$1 \le n^2, \; n \ge 1$$

$$n \le n^2, \; n \ge 1$$

$$3n^2 + 2n + 1 \le \cancel{3n^2 + 2n + n^2} \le 6n^2 \quad , n \ge 1$$

$$3n^2 + 2n + 1 \le 6n^2 \quad , n \ge 1$$

$$3n^2 + 2n + 1 \le c \cdot n^2 \quad , n \ge n_0$$

when $c = 6$, $n_0 = 1$

$$\therefore 3n^2 + 2n + 1 \in O(n^2)$$

$\log n \geq 1 , n \geq 10$

$n \log n \geq n , n \geq 10$

$5n \log n \geq 5n , n \geq 10$

$5n \leq 5n \log n , n \geq 10$

$5n \leq c \cdot n \log n , \quad n \geq n_0$

when $c = 5, n_0 = 10$

$\therefore 5n \in O(n \lg n)$

Stacks and Queues

- Limited ADTs
- Sequential data with restricted access

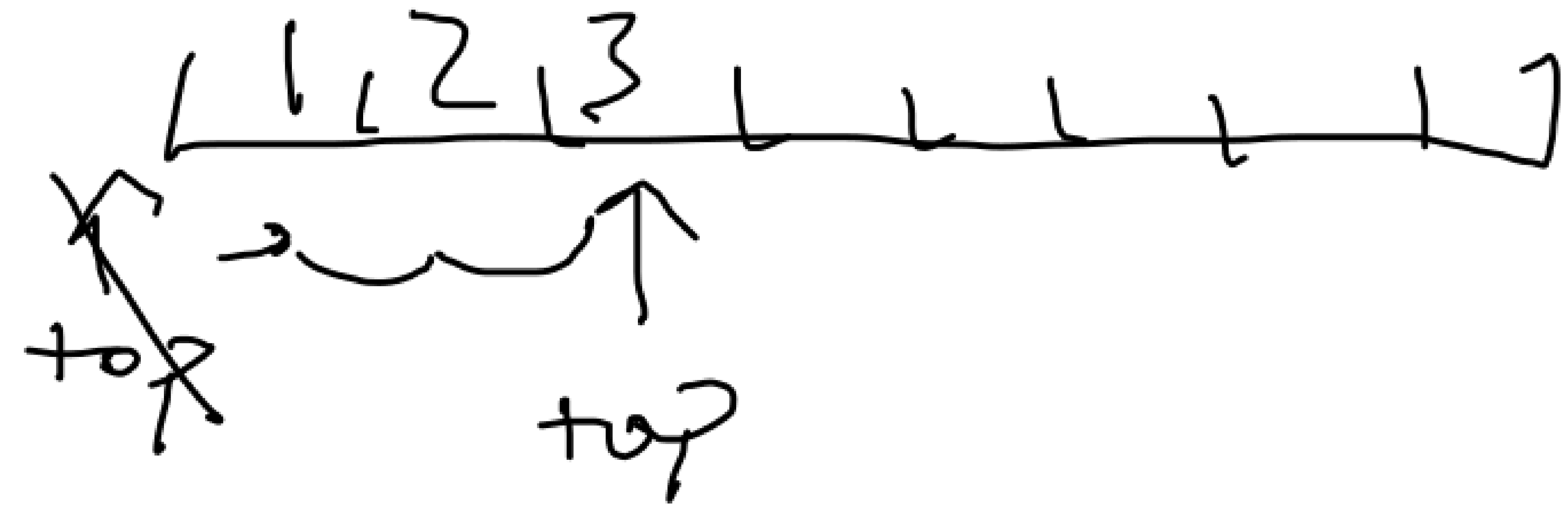
Stacks

- LIFO (last in first out)
- Available methods
  - `void push (Type element)`
  - `Type pop ()`
  - `Type peek ()`
    - sometimes this is `top`
- Common Uses
  - Reversing all elements in a list
  - Adding an undo feature in a program
    - E.g the undo sequence in a word processor
  - A history of visited pages through a web browser
  - Delimiter matching (like matching brackets)
  - Path finding in a maze

Queues

- FIFO (first in first out)
- Available methods
  - `void enqueue (Type element)`
    - Also `add` or `offer`
  - `Type dequque ()`
    - Also `remove` or `poll`
  - `Type front ()`
    - Also `peek` or `element`
- When does one use a queue?
  - When you want to preserve the order of the items
  - Unlike stacks, which reverses the order of the items
- Common Uses

- Round Robin Schedulers
- Waiting lines
- Simulations



Efficiency of Stacks

Implemented with an array:

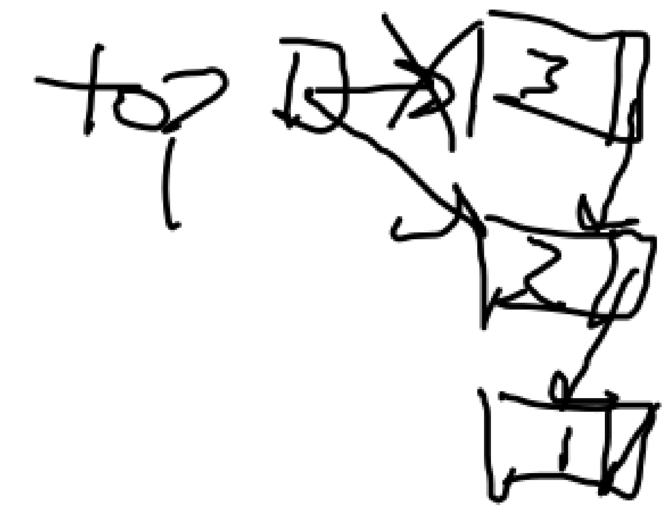
```
public class Stack {

    private int[] stackArray;
    private int top;

    public Stack (int maxSize) {
        stackArray = new int[maxSize];
        top = -1;
    }

    public void push (int item) {
        if (isFull ()) {            ~ 1 op
            // throw exception
        }                                          = 3 operations ∈ O(1)

        top++;            ~ 1
        stackArray[top] = item;   ~ 1
    }

    public int pop () {
        if (isEmpty ()) {      ~ 1 op
            // throw exception
        }
                                      top
      ~ int topItem = stackArray[~~front~~];  ~ 1      = 4 operations ∈ O(1)
        top--;   ~ 1
        return topItem;   ~ 1
    }

    public int peek () {
        if (isEmpty ()) {     ~ 1 op
            // throw exception
        }                                   = 2     ∈ O(1)

        return stackArray[top];   ~ 1
    }
```

```
    public boolean isEmpty () {
        return (top == -1);   ← 1 ) = 1 operations
    }

    public boolean isFull () {
        return (top + 1 >= stackArray.length);   ← 1 ) = 1 operation
    }

    public boolean size () {
        return top + 1;
    }

}
```

Implemented with a linked list:

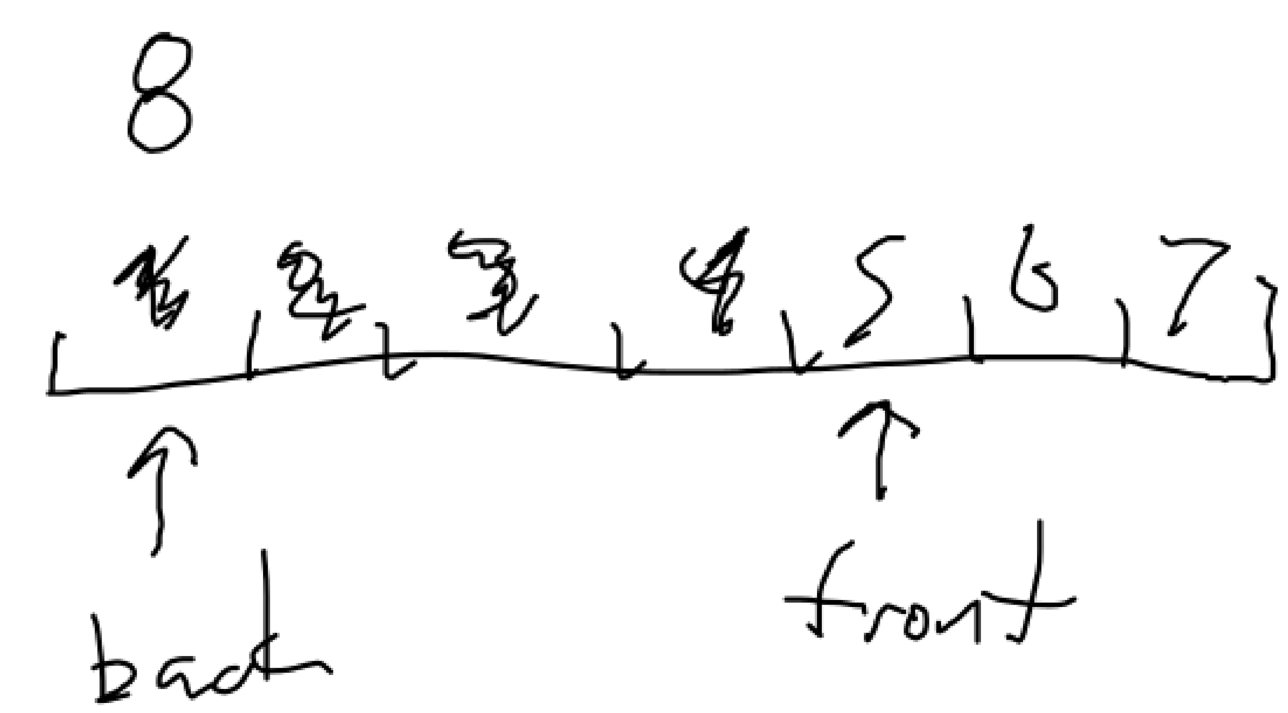
Note: See Linked List Node handout for node implementation.

```java
public class Stack {

    private Node top;
    private int size;

    public Stack () {
        top = null;
        size = 0;
    }

    public void push (int item) {
        Node newNode = new Node(item, top);   -1
        top = newNode;   -1
        size++;   -1
    }                                          = 3 ∈ O(1)

    public int pop () {
        if (isEmpty ()) {   -1
            // throw exception
        }

        int topItem = top.getData();   -1
        top = top.getNext();   -1
        size--;   -1
        return topItem;   -1
    }                                          = 5 ∈ O(1)

    public int peek () {
        if (isEmpty ()) {   -1
            // throw exception
        }

        return top.getData();   -1
    }                                          = 2   O(1)

    public boolean isEmpty () {
        return size == 0;   -1                 = 1 operation
    }

    public boolean size () {
        return size;
    }

}
```



Results:

- Both implementations have the same efficiency for all operations
- Every operation is O(1)
- Only difference is dynamic versus fixed size
  - Dynamic sizes can be implemented in arrays, but then the efficiency isn't gauranteed to be O(1)



Efficiency of Queues

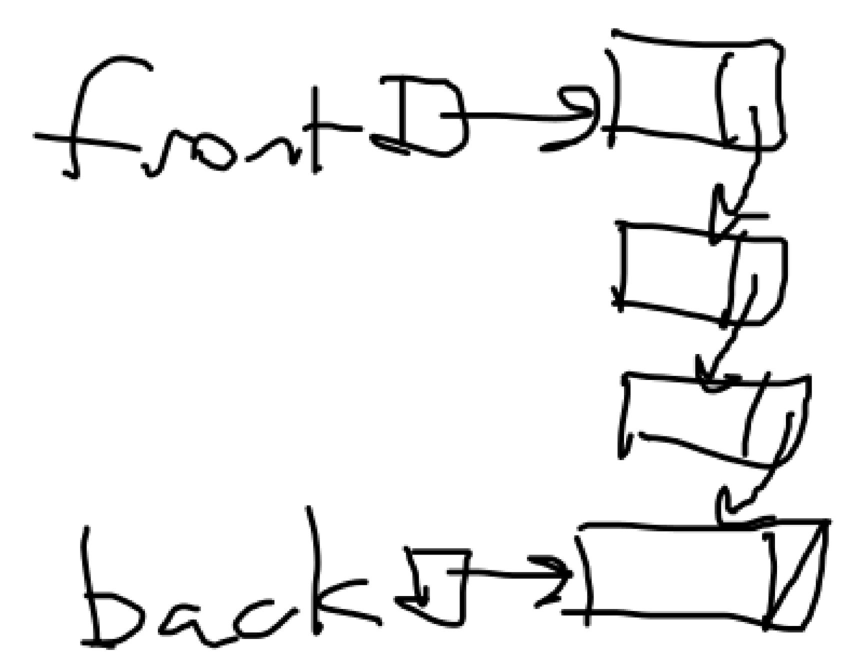
Implemented with an array:

```
public class Queue {

    private int[] queueArray;
    private int front;
    private int back;
    private int size;

    public Queue (int maxSize) {
        queueArray = new int[maxSize];
        front = 0;
        back = -1;
        size = 0;
    }

    public void enqueue (int item) {
        if (isFull()) {     - 1
            // throw exception
        }                                            4 = O(1)

        back = (back + 1) % queueArray.length; - 1
        size++; - 1
        queueArray[back] = item;  - 1
    }

    public int dequeue () {
        if (isEmpty()) {  - 1
            // throw exception
        }

        int frontItem = queueArray[front]; - 1
        front = (front + 1) % queueArray.length; - 1     5 = O(1)
        size--; - 1
```

return frontItem; - 1

```
        return frontItem;
    }

    public int front () {                 )  2  ← O(1)
        if (isEmpty ()) {
            // throw exception
        }

        return queueArray[front];
    }

    public boolean isEmpty () {      )  O(1)
        return size == 0;
    }

    public boolean isFull () {               )  O(1)
        return (top + 1 >= queueArray.length);
                size
    }

    public boolean size () {
        return size;
    }

}
```

Implemented with a linked list:



Note: See Linked List Node handout for node implementation.

```
public class Queue {

    private Node front;
    private Node back;
    private int size;

    public Queue () {
        front = null;
        back = null;
        size = 0;
    }

    public void enqueue (int item) {
        Node newNode = new Node(item, null);  - 1
        if (back != null) {  - 1
            back.setNext(newNode);  - 1
        }
```

+3 op

```
        back = newNode;          — 1
        if (front == null) {     — 1
            front = newNode;     — 1
        }
    }
```

$6 = O(1)$

```
    public int dequeue () {
        if (isEmpty ()) {        — 1
            // throw exception
        }

        int frontItem = front.getData();   — 1
        front = front.getNext();   — 1
        return frontItem;          — 1
    }
```

$4 = O(1)$

```
    public int front () {
        if (isEmpty ()) {
            // throw exception
        }

        return front.getData();
    }
```

$2 = O(1)$

```
    public boolean isEmpty () {
        return size == 0;
    }

    public boolean size () {
        return size;
    }

}
```

Results:

- Same as for stacks
- Conclusion: stacks and queues are "solved"; time to work on more advanced abstract data types.

Map	get	put	remove
Set	contains	add	remove
Generic Steps	find	find + insert	find + delete