Stacks with an array:

Use a big array for storage.

Store stack elements in bottom-up order at left end of array.
public class ArrayStack<E> {
    ArrayStack() {
        theArray = new E[BIG_NUMBER];
        topOfStack = -1;
    }

    public E top() {
        if (topOfStack == EMPTY_INDEX) {
            return null;
        } else {
            return E[topOfStack];
        }
    }

    public int size() {
        return topOfStack + 1;
    }

    // ...

    private final int EMPTY_INDEX = -1; // (-1) + 1 = 0

    private E[] theArray;
    private int topOfStack;
}
public class ArrayStack<E> {
    // ...

    public void push(E element) {
        // assume for now that the array is sufficiently large
        topOfStack++;
        theArray[topOfStack] = element;
    }

    public E pop() {
        if (topOfStack == EMPTY_INDEX) {
            return null;
        } else {
            E element = theArray[topOfStack];
            theArray[topOfStack] = null;
            topOfStack--;
            return element;
        }
    }

    // ...
}

Fast O(1) time,
Array-based queue.

As you use queue, front and back move right. Use a circular array.
If back or front go beyond end, set to 0.
public class ArrayQueue<E> {
    ArrayQueue() {
        theArray = new E[BIG_NUMBER];
        front = 0;
        back = 0;
        size = 0;
    }

    public void enqueue(E element) {
        // assume for now that the array is sufficiently large
        if (back == theArray.length - 1) {
            back = 0;
        } else {
            back++;
        }
        theArray[back] = element;
        size++;  
    }

    public E dequeue() {
        if (size == 0) {
            return null;
        } else {
            E element = theArray[front];
            theArray[front] = null;
            if (front == theArray.length - 1) {
                front = 0;
            } else {
                front++;
            }
            size--;
            return element;
        }
    }

    public int size() {
        return size;
    }

    private E[] theArray;
    private int front, back;
    private int size;
}
Space: want a large enough array, but not too big.

Ideal: $\Theta(n)$ space amortized $O(1)$ time operations
Push / enqueue:
When adding the (array, length + 1)st element,
n; # elements after op.
Copy everything to new array of length 2n.
When $n$ goes to $\frac{\text{array length}}{3}$, copy to new array of length $2n$. Space is always $\Theta(n)$. Pop/dequeue:
Analysis:

Amortized bound on $A$ times we write an array element.

Taxes: Each operation collect $7$

It op. doesn't make a new array
Use $1$ to pay for op. & put $16$ in treasury.

If we do make a new array cost is $3n$. 
η: # elements since last time we made a new array

It was length $2n$.

We're making a new array, so either
\[ n > 2n \quad \text{or} \quad n < 2^{n/3} \]

\[ \Rightarrow \frac{n}{2} > n \quad \text{or} \quad \frac{3n}{2} < n \]

\[ \Rightarrow \text{we added} \quad \frac{n}{2} \text{ elements} \quad \text{or} \quad \frac{n}{2} \text{ elements} \]

\[ \Rightarrow \text{treasury has} \]

\[ > 6 \cdot \frac{n}{2} = 3n \text{ dollars} \]

so amortized < 7 writes per operation
Splaying:

Two kinds of double rotation

Zig-Zag!
Do these double rotations to bring $x$ to root (or child) of tree.

**Splay on $x$:**

roller-coaster (zig-zig)
of root. + one single rotation is needed to make x the root.
Contains: Find element as usual, splay its node to root.

Insert: Make a new node as normal, then splay new node to root.
Remove: Splay old node \( x \) to root.

\[ \xrightarrow{ } \]

Remove \( x \).

Find leftmost node \( y \) of right subtree.

Splay \( y \) to root.
of right subtree.

\[ \rightarrow \quad \text{Link } y \text{'s left child to root of left subtree} \]
Analysis:
All operations have cost proportional to \# nodes on a splayed path.
So we'll find an amortized bound on cost of a splay (\# nodes on its path)
Assign a non-negative weight $w(x)$ to each node.

Let just use $w(x) = 1$.

$\text{Size}(x) := \text{sum of weights of weights in subtree}$

$\text{rank}(x) := \lceil \log \text{size}(x) \rceil$
\( \Phi_i = \sum \text{rank}(v) \)

\( \Phi = \Theta(n \log n) \)

\( \phi = \Theta(n) \)
Lemma: Amortized cost of a single rotation is \( \leq 1 + 3\text{rank}'(v) - 3\text{rank}(v) \)

Cost of splay before splay is \( \leq 1 + \text{rank}(\text{root}) \)
So...

\[ W(x) = 1 \quad \forall x \]

\[ \Rightarrow \text{size}(x) = \text{nodes in subtree} \]

\[ \Rightarrow \text{rank}(x) \leq \log n \]

\[ \Rightarrow \text{cost} \leq 1 + \log n \]

\[ - \text{rank}(v) \leq 1 + \log n \]

\[ = O(\log n) \]
Suppose each node is accessed $\geq t(x)$ times.

Let $T \geq t(x)$

Amortized cost is $O(\log T - \log (t(x)))$

(If you access some nodes often, amortized cost is less)
**Static optimality theorem**

Proof: Set \( w(x) = f(x) \). Then

\[
\text{cost} \leq 1 + O \left( \log T \right) - \text{rank}(x) - \left( 1 + \log f(x) \right)
\]
Static Finger Theorem

For any node \( s \),

\[
\text{dist}(s, x) = A \text{ nodes between them in sorted order}
\]

\[
\text{amortized cost of a splay is } O(\log \text{dist}(s, x))
\]

Proof: \( w(x) = \frac{1}{\text{dist}^2(s, x)} \)
\[ \sin \theta (\text{root}) = \exp \left( \frac{-2}{\sqrt{\omega}} \right) \]

\[ \approx \frac{1}{3} \]

\[ = O(1) \]

\[ \text{rank}(x) \geq \log \omega (x) \]

\[ \geq \log \text{dist}(x) \]

\[ \cos \theta \leq 1 - O(1) \]

\[ -(-2 \log (\text{dist}(x))) \]