Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement (a good) API!

Q. How to grow and shrink array?

First try.

- `push()`: increase size of array `s[]` by 1.
- `pop()`: decrease size of array `s[]` by 1.

Too expensive.

- Need to copy all item to a new array.
- Inserting first N items takes time proportional to $1 + 2 + \ldots + N \sim N^2 / 2$.

infeasible for large N

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-array implementation

"repeated doubling"

Q. How to grow array?

A. If array is full, create a new array of twice the size, and copy items.

```
public ResizingArrayStackOfStrings()
{  s = new String[1];  }
public void push(String item)
{
   if (N == s.length) resize(2 * s.length);
   s[N++] = item;
}
private void resize(int capacity)
{
   String[] copy = new String[capacity];
   for (int i = 0; i < N; i++)
      copy[i] = s[i];
   s = copy;
}
```

Q. Inserting first N items takes time proportional to N (not N^2).

Why?

Stack: resizing-array implementation

A. Inserting N items takes into a resizable array takes…

1 + 2 + 4 + 8 …. + 2^log(N) operations

= geometric series

= 2(1-2^N)/(1-2) + 1

For N = 16, log N = 4

Sum is

2 ((1-16)/-1) + 1 = 31

For N = 64, log N = 6

Sum is

2 ((1-64)/-1) + 1 = 127

So cost is at most 2N-1 which is O(N)

Does not need to be "double", so long as a constant factor: e.g., 2/1, 3/2, 9/8, etc…

Thus, double the array size when a resize is needed on average amortizes the cost of insertion to O(1) per insertion!

Amortized algorithm analysis is a cool concept you should look up!

Stack: resizing-array implementation

Q. How to shrink array?

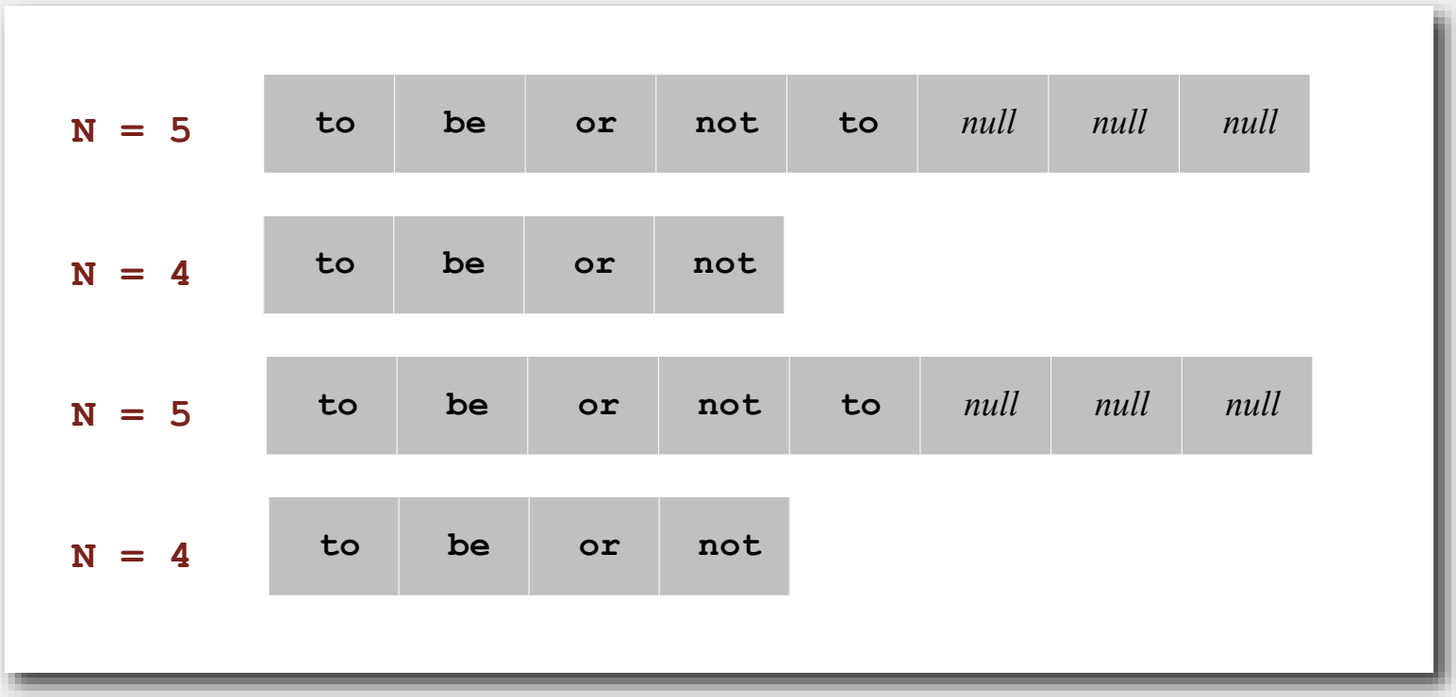
First try.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is one-half full.

Too expensive in worst case.

- Consider push-pop-push-pop-... sequence when array is full.
- Each operation takes time proportional to N.

"thrashing"



Stack: resizing-array implementation

Q. How to shrink array?

More efficient solution.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is one-quarter full.

```
public String pop()
{
     String item = s[--N];
     s[N] = null;
     if (N > 0 && N == s.length/4)
         resize(s.length/2);
     return item;
}
```

Invariant. Array is between 25% and 100% full.

Stack resizing-array implementation: performance

Amortized analysis. Average running time per operation over a worst-case sequence of operations.

Proposition. Starting from an empty stack, any sequence of M push and pop operations takes time proportional to M.

	best	worst	amortized
construct	1	1	1
push	1	N	1
pop	1	N	1
size	1	1	1

doubling and halving operations

order of growth of running time for resizing stack with N items

Stack implementations: resizing array vs. linked list

Tradeoffs. Can implement a stack with either a resizing array or a linked list; client can use interchangeably. Which one is better?

Linked-list implementation.

- Every operation takes constant time in the worst case.
- Uses extra time and space to deal with the links.

Resizing-array implementation.

- Every operation takes constant amortized time.
- Less wasted space.