Class Meeting #4

COS 226 — Spring 2018

Mark Braverman

Amortization

- When we design and implement algorithms, we often aim to minimize some resources, such as time and memory.
- Often, we cannot guarantee *worst-case* performance, but instead have to settle for average-case.
- Amortized time T per operation == k operations cost $\leq k \cdot T$ time steps.
- Not to be confused with expected time.

How to think about amortization?

Real-life examples of amortization:

- Maintenance costs
- Big purchases
- Insurance

Example: condo maintenance

- Expenses:
  - Roof: $100K, every 20 years
  - Gardening: $10K/year
  - Elevator: $300K, when it breaks
  - Fire alarm system: $50K, every 10 years.
  - …
- Income: condo fees, stable over time (*if the condo is well-managed): $5K/month

Long-term cost of condo

- Claim: the long-term cost of maintaining the condo is $5K/month
- To establish this claim we only need to show that:
- If we collect $5K/month, we will remain solvent forever.
- Done with careful accounting.
- Amortization “spreads” the $100K roof over many months

Stack with resizable array

Example from section 1.4

- Maintain stack contents in an array.
- If run out of room…

  double the size of the array

```
public void push(Item item) {
    if (n == a.length) resize(2*a.length);    // double size of array if necessary
    a[n++] = item;                            // add item
}
```

Stack with resizable array

Problem:

- May end up wasting a lot of space:

Stack with resizable array

Solution:

- When array becomes less then quarter full, resize it.

why not half??

```
public Item pop() {
    if (isEmpty()) throw new NoSuchElementException("Stack underflow");
    Item item = a[n-1];
    a[n-1] = null;                              // to avoid loitering
    n--;
    // shrink size of array if necessary
    if (n > 0 && n == a.length/4) resize(a.length/2);
    return item;
}
```

Cost analysis

- Want to show that the cost of resizing is constant per operation.
- Cost of resizing from n to 2n is ~2n.
- Cost of resizing from 2n to n is ~n.
- Collect $5 for each push(), pop() operation.
- Pay $2n to resize from n to 2n.
- Pay $n to resize from 2n to n.
- Want: show that we'll remain solvent.
- Then, after m ops, collect at most $5m, and so resizing cost <5m

Observation

- After resizing, the array is of size 2n, and has either n or n+1 elements.
- Resizing up:
  - Resize to 2*n
  - Have n+1 elements

```
public void push(Item item) {
    if (n == a.length) resize(2*a.length);    // double size of array if necessary
    a[n++] = item;                            // add item
}
```

Observation

- After resizing, the array is of size 2n, and has either n or n+1 elements.
- Resizing down:
  - Resize to 2*n
  - Have n elements

```
public Item pop() {
    if (isEmpty()) throw new NoSuchElementException("Stack underflow");
    Item item = a[n-1];
    a[n-1] = null;                              // to avoid loitering
    n--;
    // shrink size of array if necessary
    if (n > 0 && n == a.length/4) resize(a.length/2);
    return item;
}
```

Saving money for next resize

When will next resize happen?

- Next resize up, will require at least $n-1$ operations, and will cost \$$4n$.
- Next resize down, will require at least $n/2$ operations, and will cost \$$n/2$.

Accounting

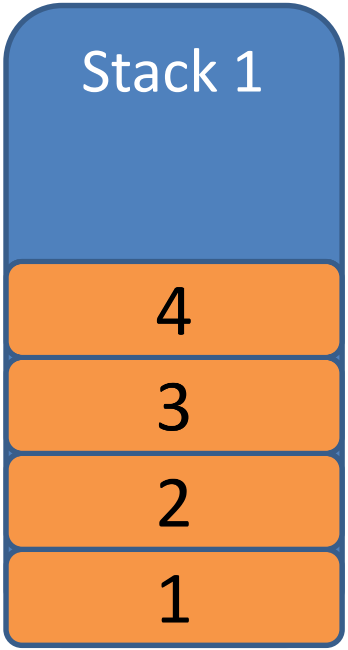
- Case 1: Collect at least $\$5n - 5$, can afford $\$4n$, as long as $n \geq 5$.
- Case 2: Collect at least $\$5n/2$, can afford $\$n/2$.
- Yay!

Algorithm design examples

<u>Problem</u>: given a Stack implementation, implement a queue, subject to the following conditions:

- Use two Stacks
- Amortized constant cost of enqueue() and dequeue()

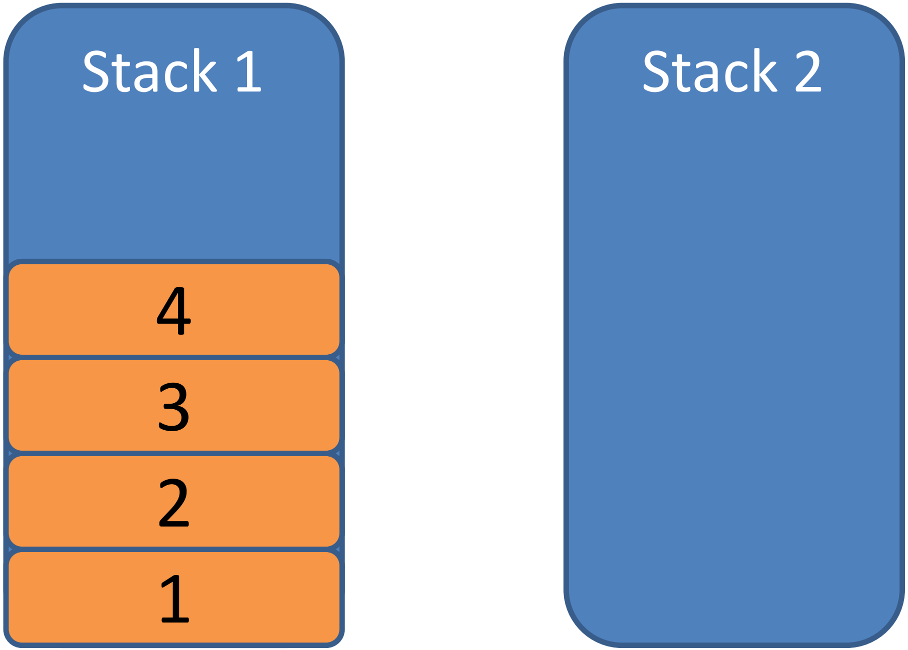
Solution



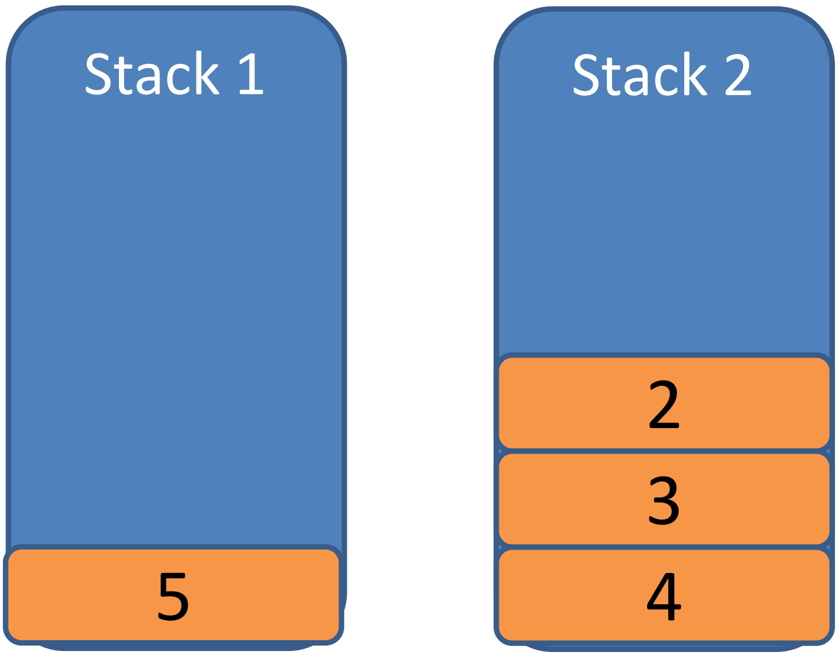
- enqueue(1)
- enqueue(2)
- enqueue(3)
- enqueue(4)
- dequeue()

Solution

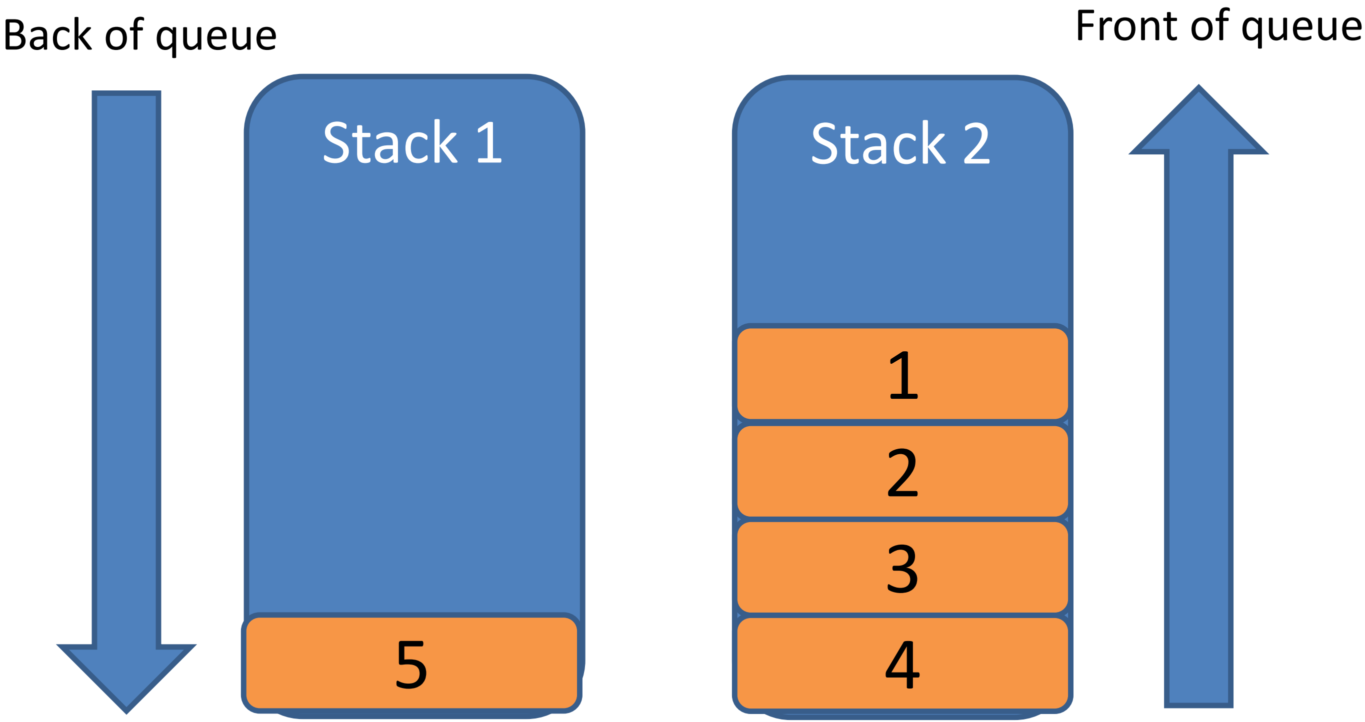
- dequeue()



Solution



- dequeue()
- dequeue()
- enqueue(5)
- dequeue()


Back of queue
Stack 1
5
Stack 2
1
2
3
4
Front of queue

Solution

```
enqueue(x)
        Stack1.push(x)
```

Solution

```
dequeue()
        if (Stack2.isEmpty())
                if(Stack1.isEmpty())
                        return error;
                while(!Stack1.isEmpty)
                        Stack2.push(Stack1.pop());

        return Stack2.pop();
```

Amortized analysis

- enqueue() always has cost 1.
- dequeue() may have an arbitrarily high cost.
- Use amortized analysis.

Amortized analysis

- Use amortized analysis.
- Collect $\$4$ for each enqueued element
- Pay $\$1$ for each push/pop operation
- Each element is addressed at most 4 times (push into Stack1, pop from Stack1, push into Stack2, pop from Stack2)
- The 4 operations are prepaid, therefore will always remain solvent!
- At any point: Cost so far $\leq 4 * \text{number of enqueue( ) calls}$.

The 3SUM problem

- Similar in flavor to binary search.
- Given three lists of numbers A, B, C of length n
- Want to know whether there is an element x in A, y in B, z in C such that x+y=z.

3SUM

Trivial solution

```
for (int x: A)
        for (int y: B)
                for (int z: C)
                        if (x+y==z)
                                return true;
return false;
```

Running time? $\sim n^3$

3SUM

- Many solutions in time $\sim n^2$
- Unknown whether can do better.
- Wouldn’t be completely shocking if can be done in $\sim n^{1.5}$

3SUM

Start by sorting A and B (cost $\sim n \log n$)
Design a procedure **IsInSum**(A,B,z) which, assuming A and B are sorted, returns whether there is x in A and y in B such that x+y=z

IsInSum(A,B,z)

A: ~~1~~, ~~3~~, 7, 12, 18, 22, 26, 31   B: 2, 3, 8, 11, 16, ~~21~~, ~~24~~, ~~32~~

i   j

z=23

32 from B is useless;

24 from B is useless;

1 from A is useless;

….

IsInSum(A,B,z)

```
int i=0;
int j=B.length;
while ((i<A.length)&&(j>0))
{
        if(A[i]+B[j]==z)
                return true;
        if (A[i]+B[j]>z)
                j--;
        else
                i++;
}
return false;
```

Main while() loop runs at most A.length+B.length times, constant cost each.

Total cost linear in n

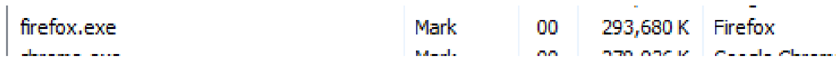
3SUM

```
sort(A)
sort(B)
for (int z: C)
        if (IsInSum(A,B,z))
                return true;
return false;
```

Assignments tips

Avoiding loitering

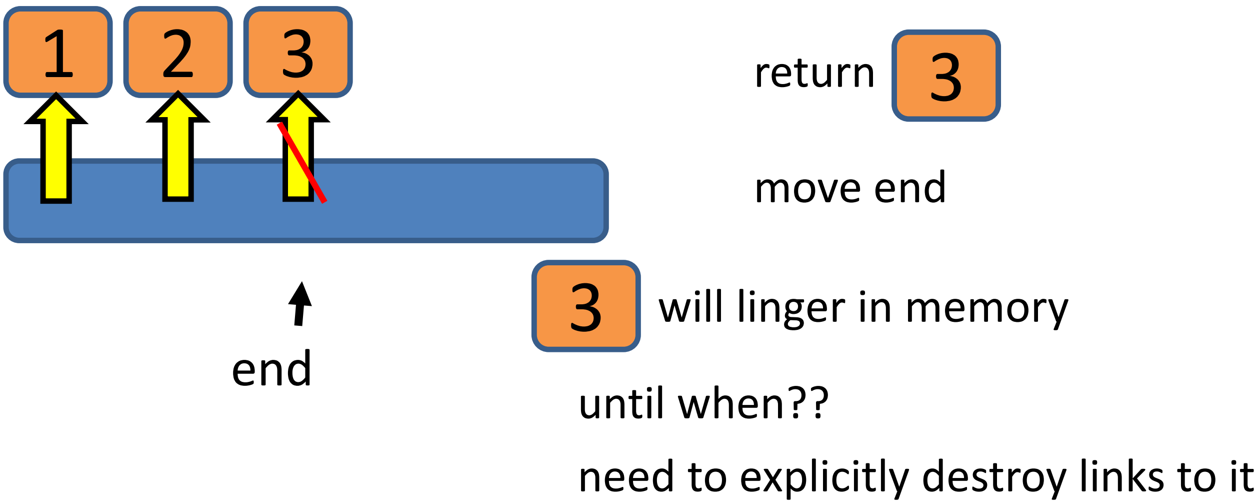
- Loitering:
  - Keeping things in memory after they are no longer needed.



- In Java, garbage collection is automatic.
- “An object is stored as long as someone is pointing at it”

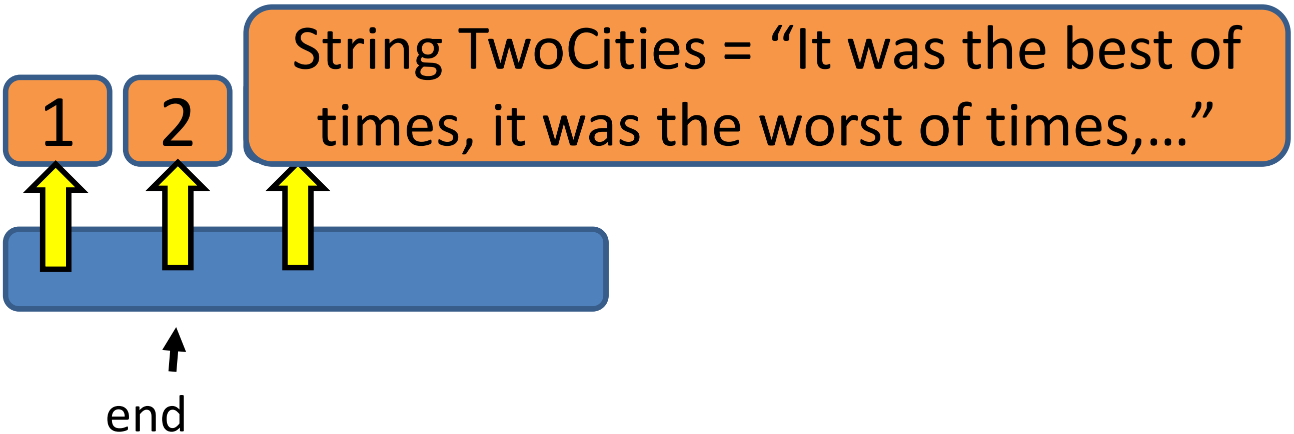
Example: linked list vs resizable array

- When we pop() an element, we may need to actively remove all reference to it.



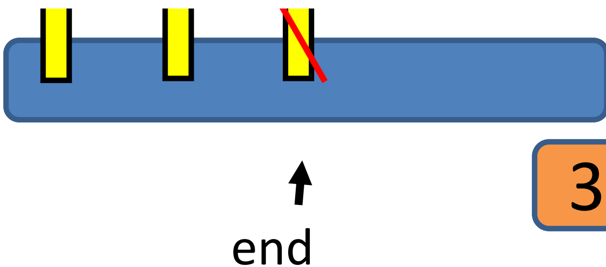
Example: linked list vs resizable array

- Is this a big problem?
- Depends on how big 3 is.



Example: linked list vs resizable array

```java
public Item pop() {
    if (isEmpty()) throw new NoSuchElementException("Stack underflow");
    Item item = a[n-1];
    a[n-1] = null;                              // to avoid loitering
    n--;
    // shrink size of array if necessary
    if (n > 0 && n == a.length/4) resize(a.length/2);
    return item;
}
```



move end

3 will linger in memory

until when??

need to explicitly destroy links to it

Example: linked list vs resizable array

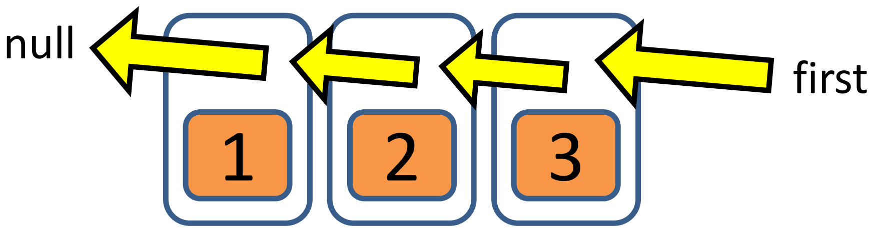
- Removal from linked list implementation of stack

```java
public Item pop() {
    if (isEmpty()) throw new NoSuchElementException("Stack underflow");
    Item item = first.item;          // save item to return
    first = first.next;              // delete first node
    n--;
    return item;                     // return the saved item
}
```

Example: linked list vs resizable array

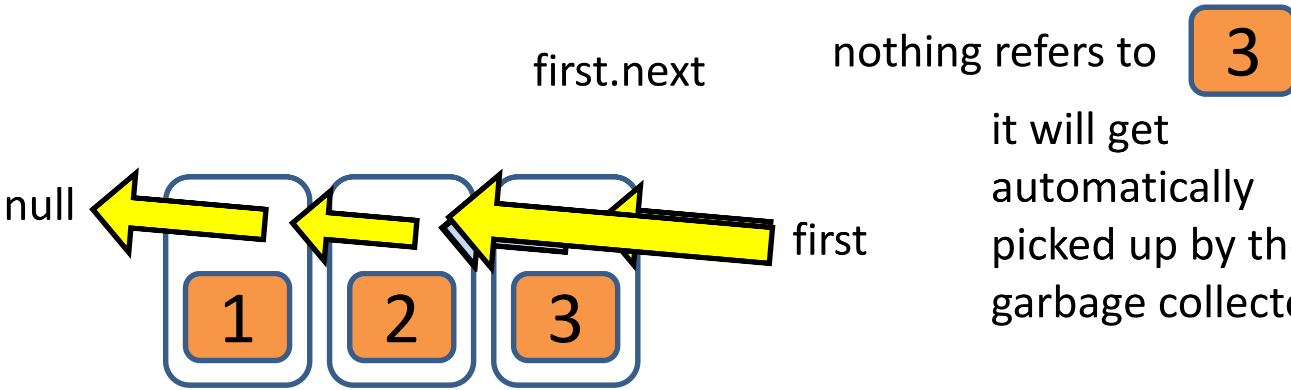
- Removal from linked list implementation of stack

```
private static class Node<Item> {
    private Item item;
    private Node<Item> next;
}
```



Example: linked list vs resizable array

```
public Item pop() {
    if (isEmpty()) throw new NoSuchElementException("Stack underflow");
    Item item = first.item;        // save item to return
    first = first.next;            // delete first node
    n--;
    return item;                   // return the saved item
}
```



it will get automatically picked up by the garbage collector

Random tips

- Consider sentinel nodes in linked implementations.
  - Often simplifies code/reduces bugs.

```
public class DoublyLinkedList<Item> implements Iterable<Item> {
    private int n;         // number of elements on list
    private Node pre;      // sentinel before first item
    private Node post;     // sentinel after last item
```

- Iterator is just another class.
  - You may put code in its constructor.
  - More: in precept tomorrow.