Lecture 5: Master Theorem, Maps, and Iterators

Data Structures and Algorithms

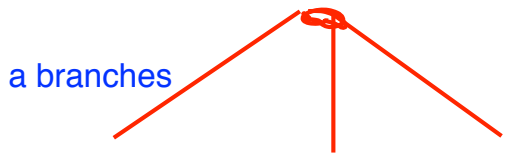
Warmup

Draw a tree for this recurrence, and write equations for the recursive and non-recursive work:

$$T(n) = \begin{cases} d & when\, n \leq 1 \\ aT\left(\frac{n}{b}\right) + n^c & \text{otherwise} \end{cases}$$

Text

ni^c work at each branch;
size at each node : n/b^i
work at each branch = (n^c/b^ic)



i = 0

i = 1

i = 2

recursive work = number branches at level i * work at each branch = a^i(n^c/b^ic) = n^c(a/b^c)^i

Enter base case when n <= 1; so when n/b^i <= 1 —> i = logb n

Base case work = work at single base case * number base cases reached = d * a^(logb (n))

Warmup

$$T(n) = \begin{cases} d & when\ n \leq 1 \\ aT\left(\frac{n}{b}\right) + n^c & otherwise \end{cases}$$

work per base case

work per recursive call

new size relation

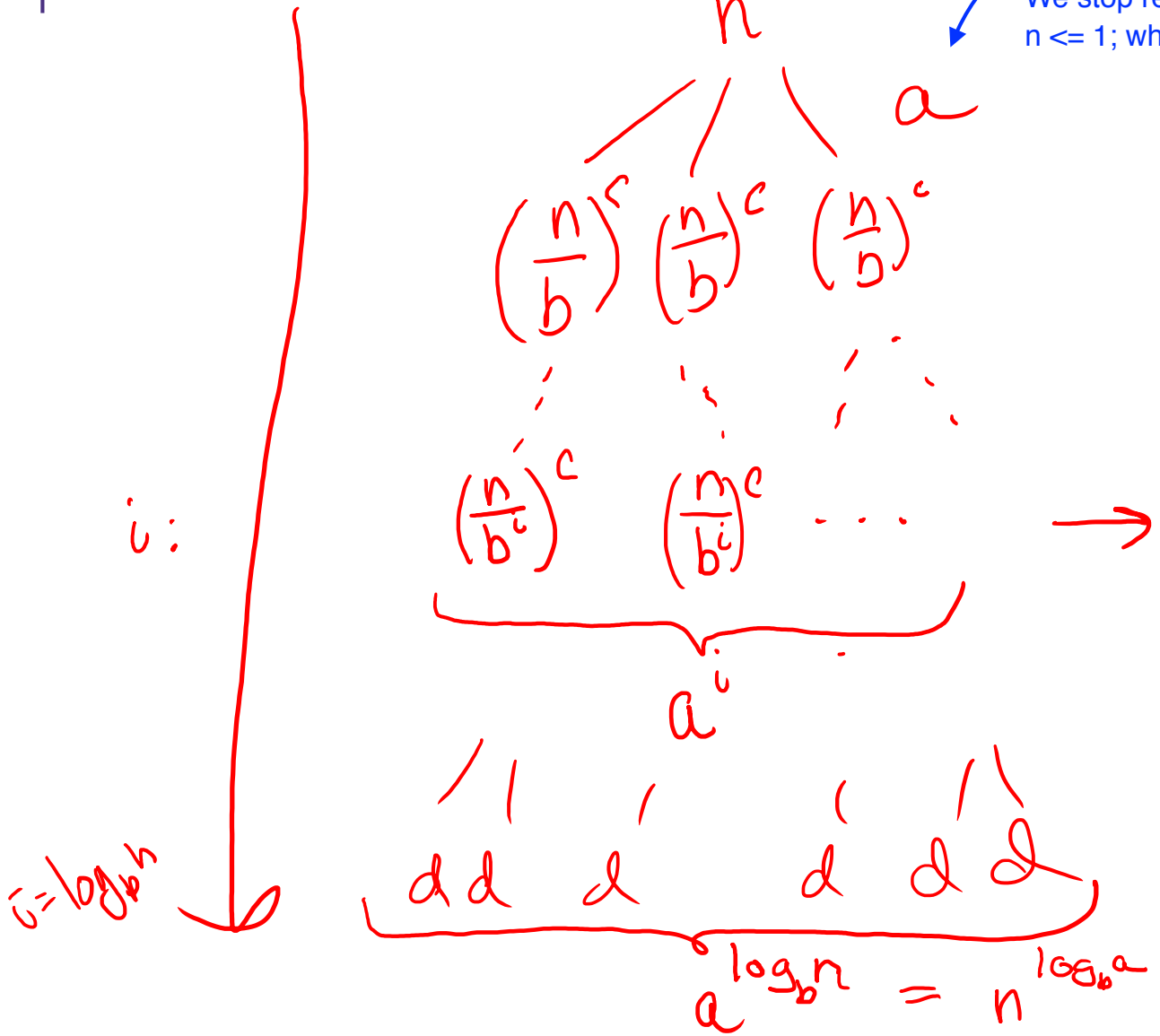
recursive calls made per level

We stop recursing when n <= 1; when does this happen?

$\frac{n}{b^i} = 1$

$n = b^i$

$i = \log_b n$ = # of layers in tree



$a^i \left(\frac{n}{b^i}\right)^c = n^c \left(\frac{a}{b^c}\right)^i$ work per level

work per method call

number of calls made to a given level

$n^c \sum_i \left(\frac{a}{b^c}\right)^i$ + $d\, n^{\log_b a}$

recursive

base case

Master Theorem

Given a recurrence of the following form:

$$T(n) = \begin{cases} d & when\ n \leq \text{some constant} \\ aT\left(\frac{n}{b}\right) + n^c & \text{otherwise} \end{cases}$$

Where a, b, c, and d are all constants.

The big-theta solution always follows this pattern:

a < b^c —> a/b^c < 1 —> work per level is decreasing; "worst case" is first level

If $\log_b a < c$ then $T(n)$ is $\Theta(n^c)$

a= b^c —> a/b^c = 1: work per level is constant, so you do n^c work for logb n levels

If $\log_b a = c$ then $T(n)$ is $\Theta(n^c \log n)$

a > b^c —> a/b^c > 1; work per level is increasing

If $\log_b a > c$ then $T(n)$ is $\Theta\left(n^{\log_b a}\right)$

$\log_b a = c$

$a = b^c$

$\frac{a}{b^c} = 1$

remember:
work per level = n^c (a/b^c)^i

so these ratios tell use whether the work per level is increasing, decreasing, or holding constant

n^c per layer

if $\log_b a = c$

so $n^c \cdot \log n$

Apply Master Theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d \text{ when } n \leq \text{some constant} \\ aT\left(\frac{n}{b}\right) + n^c \text{ otherwise} \end{cases}$$

If $\log_b a < c$ then $T(n)$ is $\Theta(n^c)$

If $\log_b a = c$ then $T(n)$ is $\Theta(n^c \log n)$

If $\log_b a > c$ then $T(n)$ is $\Theta\left(n^{\log_b a}\right)$

$$T(n) = \begin{cases} 1 \text{ when } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n^1 \text{ otherwise} \end{cases}$$

a = 2
b = 2
c = 1
d = 1

$$\log_b a = c \Rightarrow \log_2 2 = 1$$

$$T(n) \text{ is } \Theta(n^c \log_2 n) \Rightarrow \Theta(n^1 \log_2 n)$$

Reflecting on Master Theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d \text{ when } n \leq \text{some constant} \\ aT\left(\frac{n}{b}\right) + n^c \text{ otherwise} \end{cases}$$

If $\log_b a < c$ then $T(n)$ is $\Theta(n^c)$

If $\log_b a = c$ then $T(n)$ is $\Theta(n^c \log n)$

If $\log_b a > c$ then $T(n)$ is $\Theta\left(n^{\log_b a}\right)$

The $\log_b a < c$ case

- Recursive case conquers work more quickly than it divides work
- Most work happens near "top" of tree
- Non recursive work in recursive case dominates growth, n^c term

The $\log_b a = c$ case

- Work is equally distributed across levels of the tree
- Overall work is approximately work at any level x height

The $\log_b a > c$ case

- Recursive case divides work faster than it conquers work
- Most work happens near "bottom" of tree
- Work at base case dominates.

$height \approx \log_b a$ log b (n)?

$branchWork \approx n^c \log_b a$

$leafWork \approx d\left(n^{\log_b a}\right)$ = d(a^ logb (n)))

number of leaves;

Announcements

Pre-Course Survey Due Tonight!

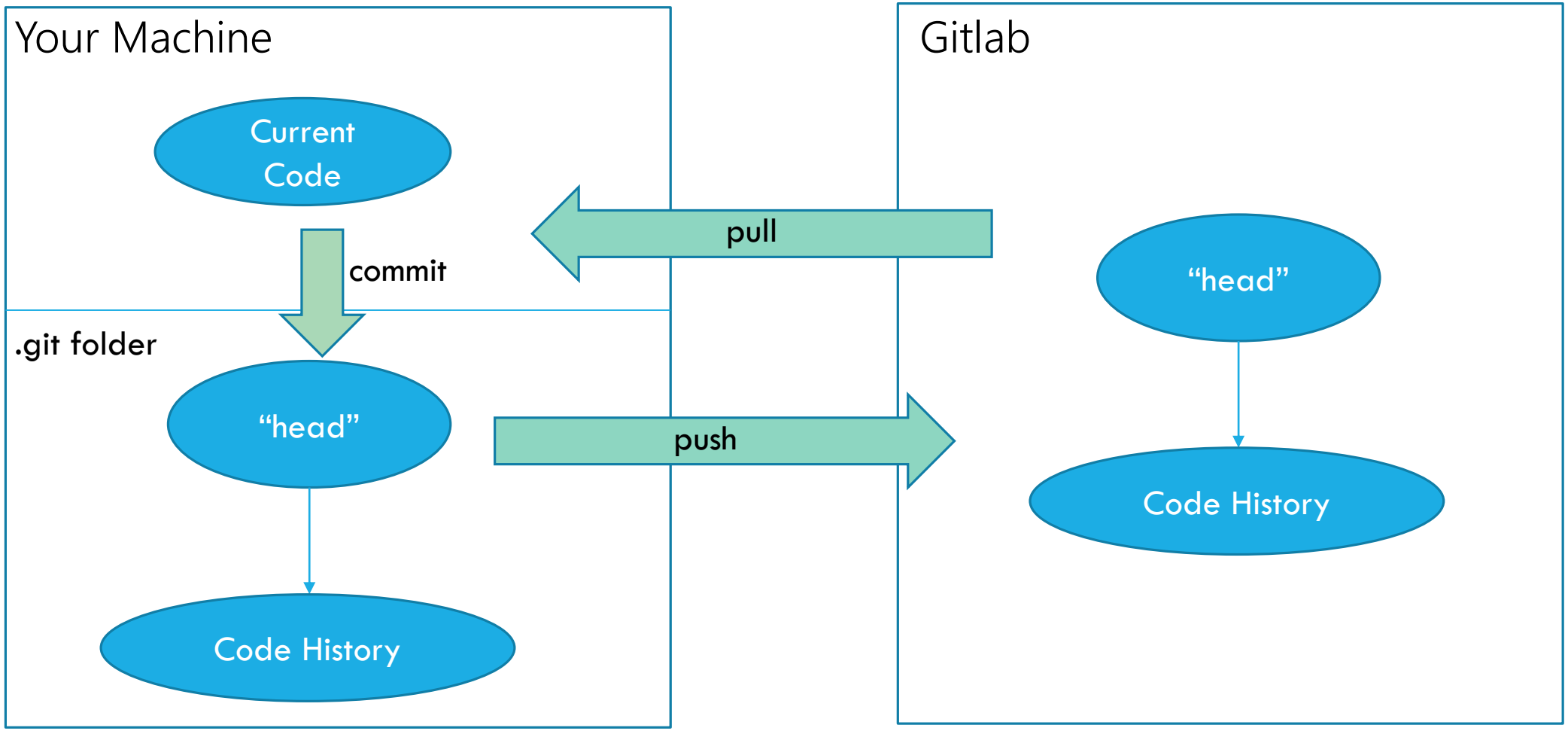
HW1 Due Tonight!

Use cse373-staff@cs.washington.edu if you want to e-mail the staff – faster responses than just e-mailing Ben!

No class Wed. July 4.

Guest lecturer Robbie Webber on Friday, July 6 (I will be out of town Wed. – Sun. with limited internet, so use the staff list for questions)

Git – How it Works



Git – Playing Nicely With Other

Git is designed to work on teams

Workflow:

You: Commit -> Push ->

Partner: Pull

(Swap roles and repeat)

You should be pair programming, so you should not need to deal with merges

If you do run into an issue with merges, talk to a TA and we will teach you more about Git!

Project Turn-In

HW 1 Due Tonight!

Tag with SUBMIT (in all caps)

If there is no SUBMIT tag, we'll use whatever was in the master branch **on Gitlab** as your submission

How to use late days: tag it later. We will use the server's timestamp of the SUBMIT tag to determine late days.

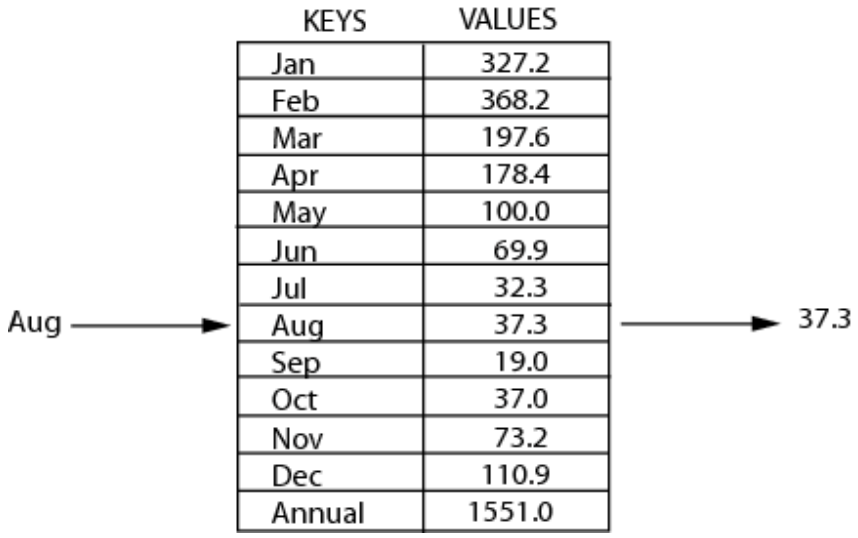
Review: Maps (Dictionaries)

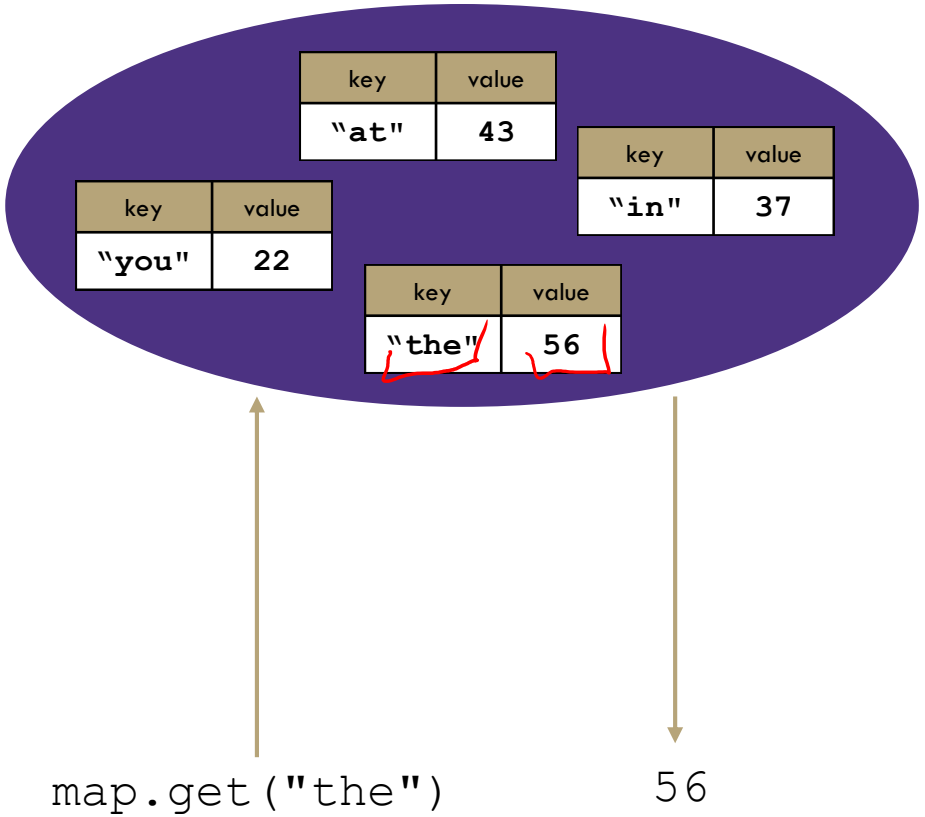
map: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.

- a.k.a. "dictionary", "associative array", "hash"

operations:

- **put**(*key*, *value*): Adds a mapping from a key to a value.
- **get**(*key*): Retrieves the value mapped to the key.
- **remove**(*key*): Removes the given key and its mapped value.





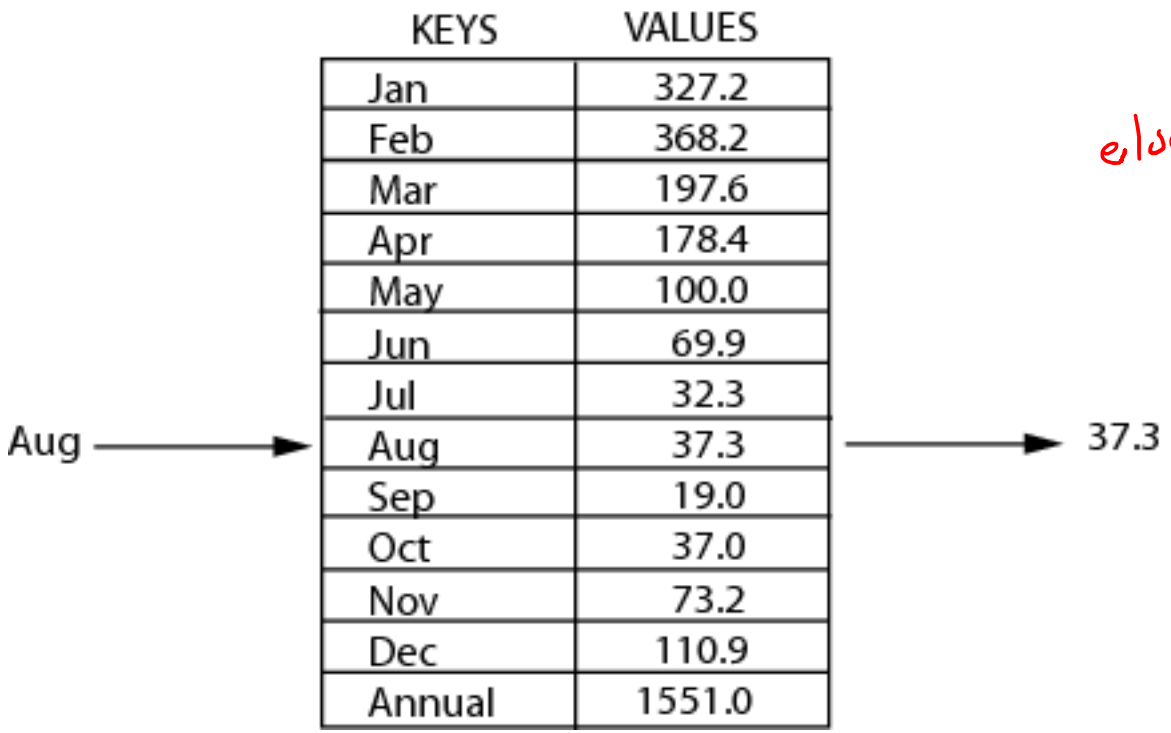
ArrayDictionary

get, put, remove

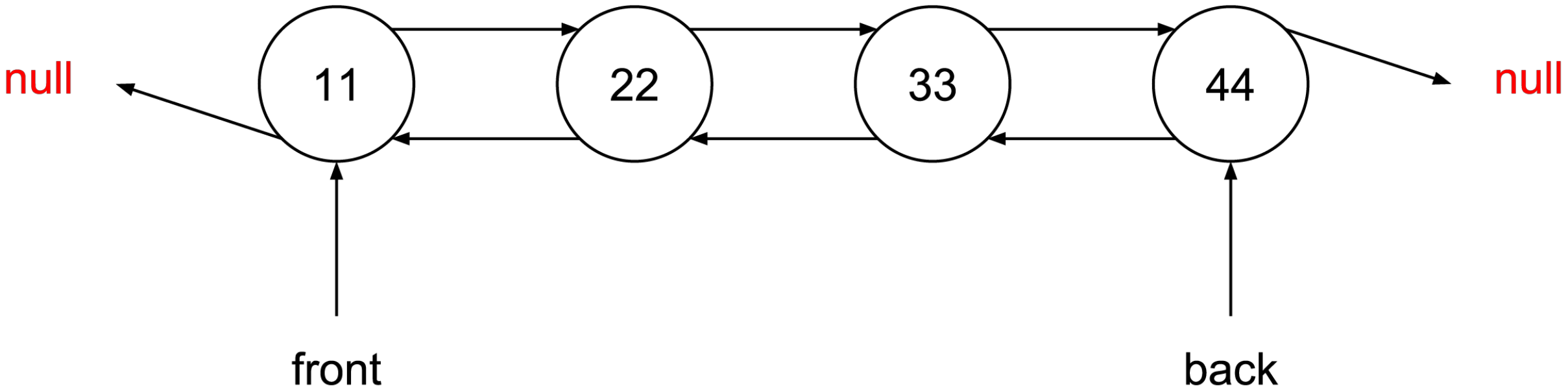
get: iterate through array until find array[i] == key then return value

put: if key in array:
find key by iterating:
update value at key

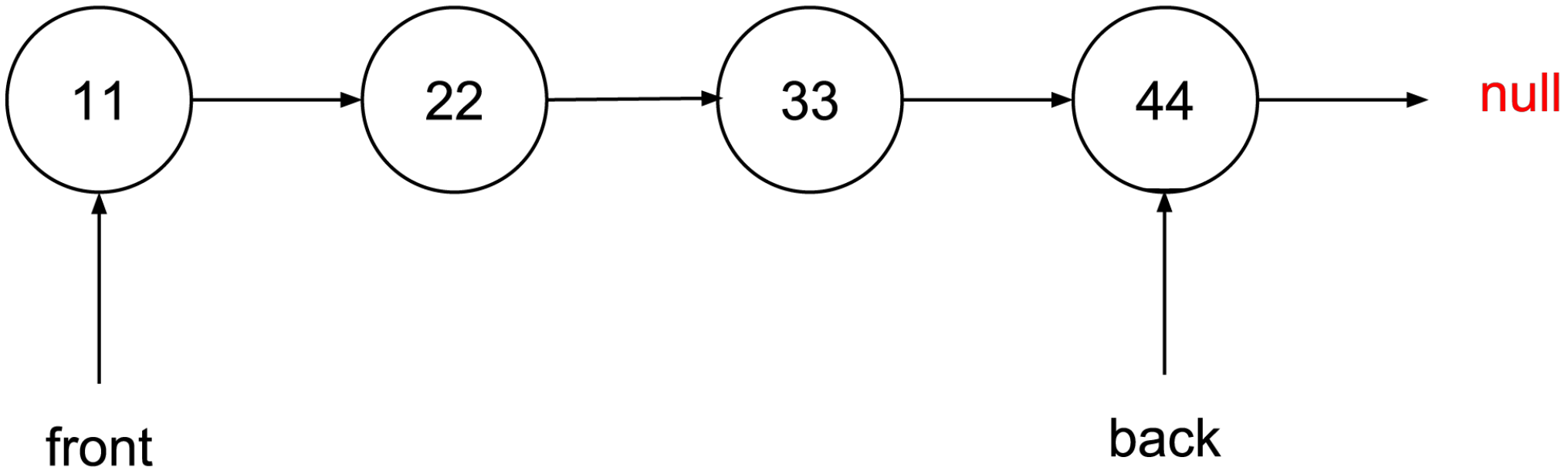
else:
put it at the end
[key, value]



Doubly-Linked List (Deque)



Doubly-Linked List (Deque)



Traversing Data

Array

```
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

List

```
for (int i = 0; i < myList.size(); i++) {
    System.out.println(myList.get(i));
}
```

"for each"

```
for (T item : list) {
    System.out.println(item);
}
```

Iterator!

Iterators

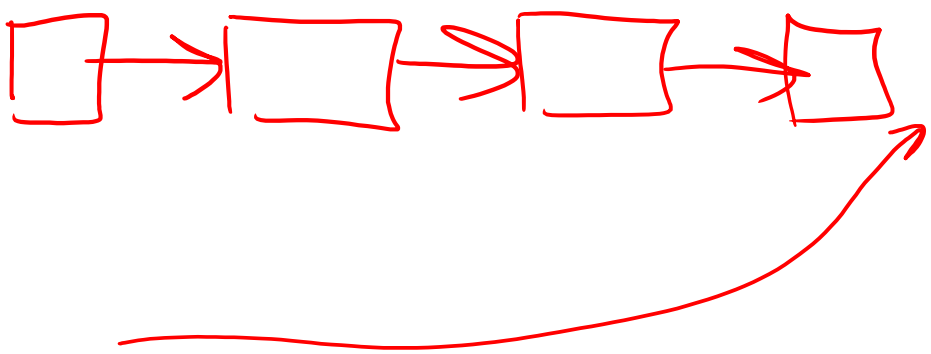
iterator: a Java interface that dictates how a collection of data should be traversed.

Behaviors:

hasNext() – returns true if the iteration has more elements

next() – returns the next element in the iteration

```
while (iterator.hasNext()) {
    T item = iterator.next();
}
```



Iterable

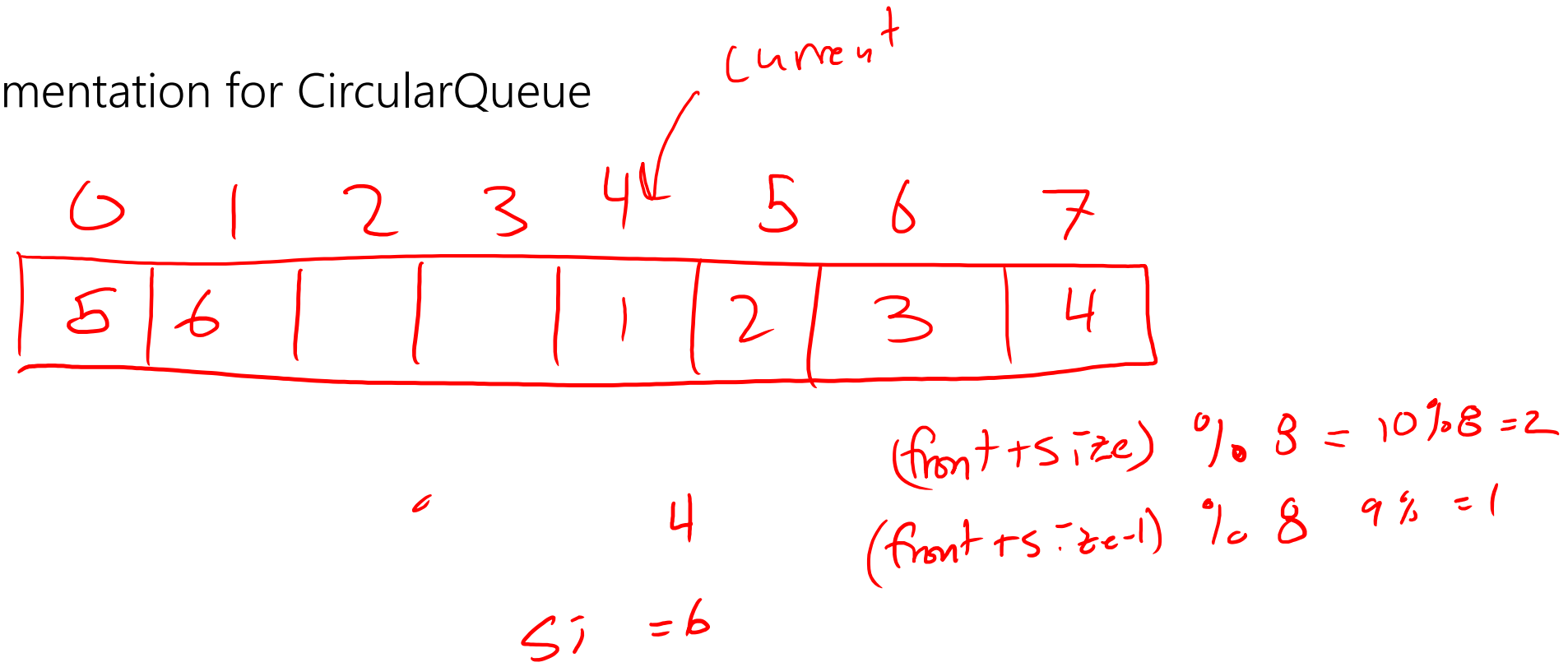
Iterable: a Java interface that lets a class be traversed using iterators (for each, etc).

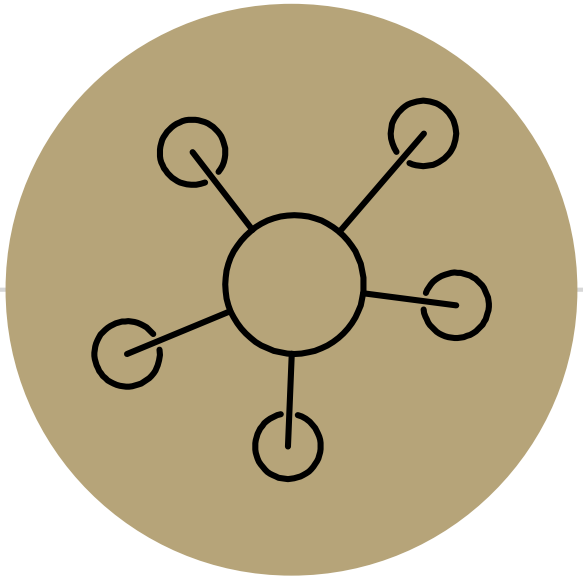
Behaviors:

iterator() – returns an iterator to the class instance

Implementing Iterable

Demo Implementation for CircularQueue





Bonus Slides

Amortized Analysis

Amortization

What's the worst case for inserting into an ArrayList?

- O(n). If the array is full.

Is O(n) a good description of the worst case behavior?

- If you're worried about a single insertion, maybe.
- If you're worried about doing, say, n insertions in a row. NO!

Amortized bounds let us study the behavior of a bunch of consecutive calls.

Amortization

The most common application of amortized bounds is for insertions/deletions and data structure resizing.

Let's see why we always do that doubling strategy.

How long in total does it take to do n insertions?

We might need to double a bunch, but the total resizing work is at most O(n)

And the regular insertions are at most $n \cdot O(1) = O(n)$

So n insertions take $O(n)$ work total

Or amortized $O(1)$ time.

Amortization

Why do we double? Why not increase the size by 10,000 each time we fill up?

How much work is done on resizing to get the size up to n?

Will need to do work on order of current size every 10,000 inserts

$$\sum_{i=0}^{\frac{n}{10000}} 10000i \approx 10{,}000 \cdot \frac{n^2}{10{,}000^2} = O(n^2)$$

The other inserts do $O(n)$ work total.

The amortized cost to insert is $O\left(\frac{n^2}{n}\right) = O(n)$.

Much worse than the $O(1)$ from doubling!