CSE 373: AVL trees

Michael Lee

Friday, Jan 19, 2018

Warmup:

- What is an *invariant*?
- What are the AVL tree invariants, exactly?

Discuss with your neighbor.

AVL Trees: Invariants

Core idea: add extra **invariant** to BSTs that enforce balance.

AVL Tree Invariants

An AVL tree has the following invariants:

- **The "structure" invariant:**
  All nodes have 0, 1, or 2 children.
- **The "BST" invariant:**
  For all nodes, all keys in the *left* subtree are smaller;
  all keys in the *right* subtree are larger
- **The "balance" invariant:**
  For all nodes, $\text{abs}\,(\text{height}\,(\text{left})) - \text{height}\,(\text{height}\,(\text{right})) \leq 1$.

AVL = **A**delson-**V**elsky and **L**andis

Interlude: Exploring the balance invariant

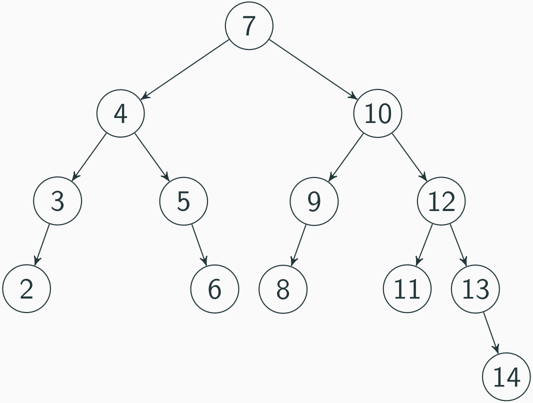
Question: why $\text{abs}\,(\text{height}\,(\text{left})) - \text{height}\,(\text{height}\,(\text{right})) \leq 1$?

Why not $\text{height}\,(\text{left}) = \text{height}\,(\text{right})$?

What happens if we insert two elements. What happens?

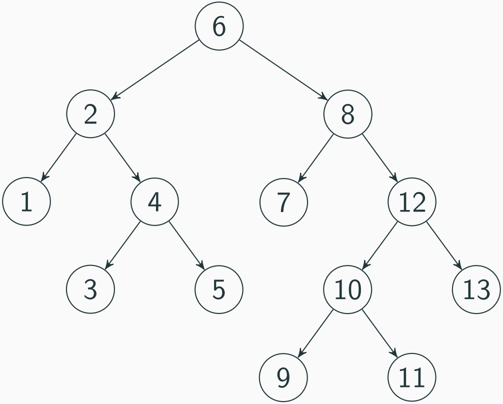
AVL tree invariants review

Question: is this a valid AVL tree?



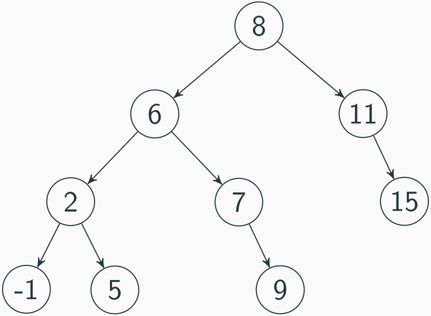
AVL tree invariants review

Question: is this also an AVL tree?



AVL tree invariants review

Question: ...and what about now?



Implementing an AVL dictionary

How do we implement an AVL dictionary?

- **get:** Same as BST!

Implementing an AVL dictionary

How do we implement an AVL dictionary?

- **get:** Same as BST!
- **containsKey:** Same as BST!

Implementing an AVL dictionary

How do we implement an AVL dictionary?

- **get:** Same as BST!
- **containsKey:** Same as BST!
- **put:** ???
- **remove:** ???

A basic example

Suppose we insert 1, 2, and 3. What happens?

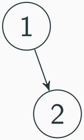
```
insert(1)
```

A basic example

Suppose we insert 1, 2, and 3. What happens?

`insert(1)`

`insert(2)`

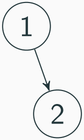


A basic example

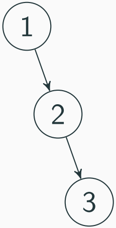
Suppose we insert 1, 2, and 3. What happens?

`insert(1)`

`insert(2)`

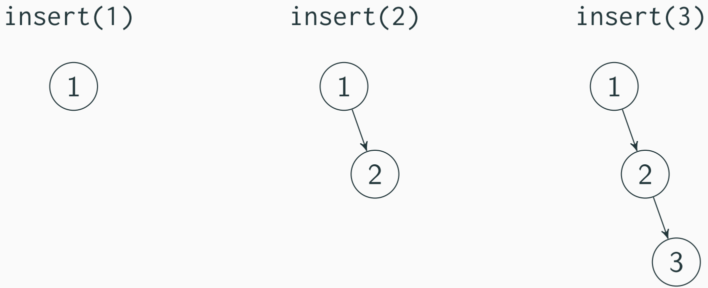


`insert(3)`



A basic example

Suppose we insert 1, 2, and 3. What happens?

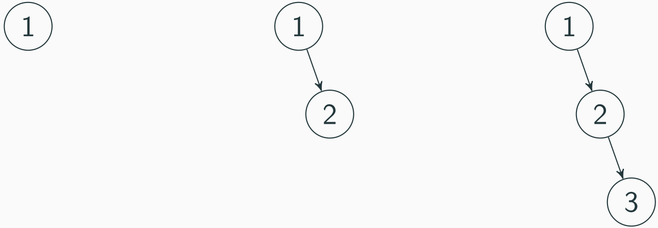


What do we do now? Hint: there's only one possible solution.

A basic example

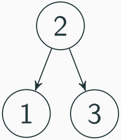
Suppose we insert 1, 2, and 3. What happens?

`insert(1)` `insert(2)` `insert(3)`



What do we do now? Hint: there's only one possible solution.

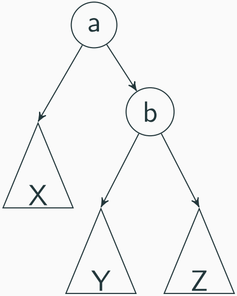
Rotate.



AVL rotation

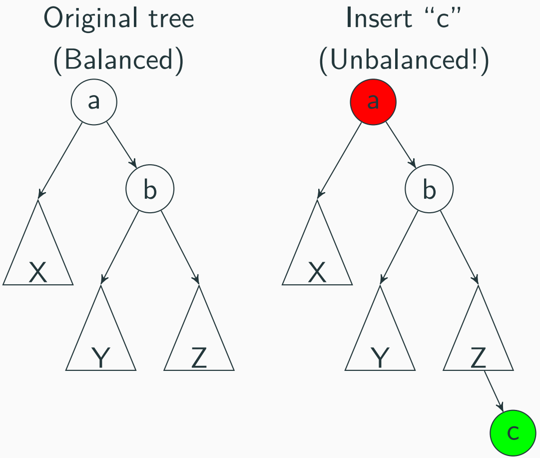
An algorithm for "insert"/"put", in pictures:

Original tree
(Balanced)



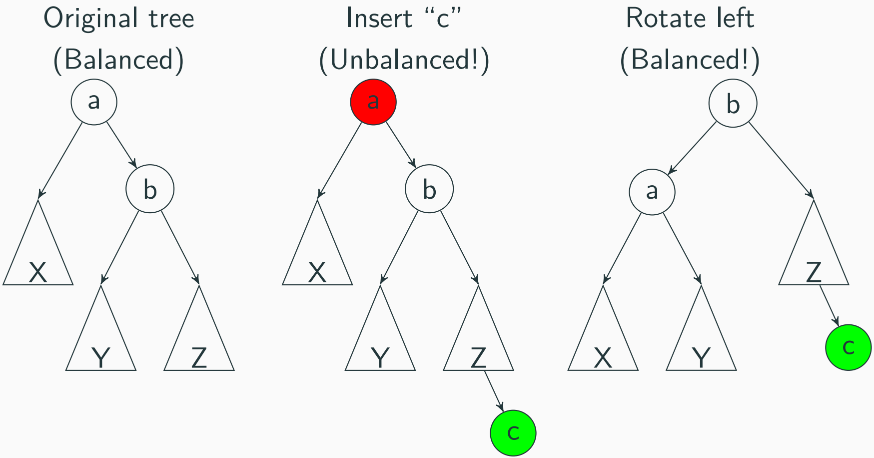
AVL rotation

An algorithm for "insert"/"put", in pictures:



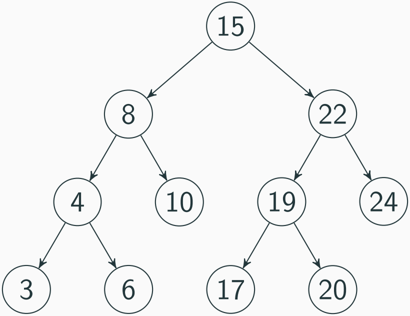
AVL rotation

An algorithm for "insert"/"put", in pictures:



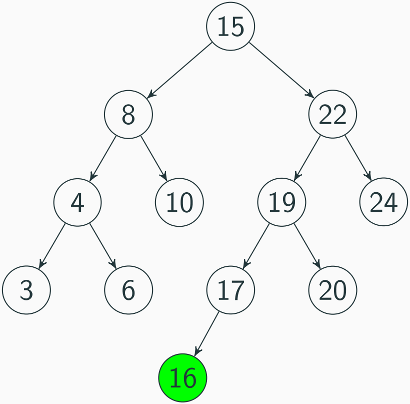
Practice

Practice: insert 16, and fix the tree:



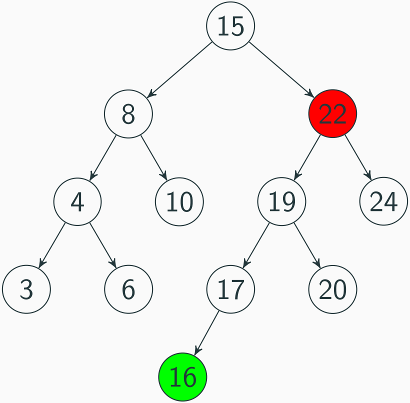
Practice

Step 1: insert 16



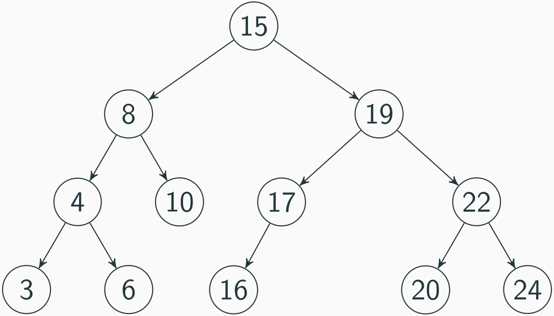
Practice

Step 2: Start from the inserted node and move back up to the root. Find the first unbalanced subtree.



Practice

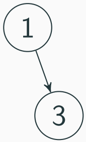
Step 3: Rotate left or right to fix. (Here, we rotate right).



A second case...

Now, try this. Insert 1, 3, then 2. What's the issue?

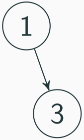
```
insert 1 and 3
```



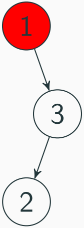
A second case...

Now, try this. Insert 1, 3, then 2. What's the issue?

`insert 1 and 3`

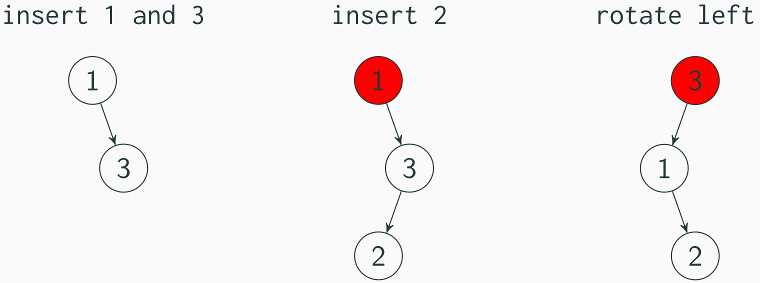


`insert 2`



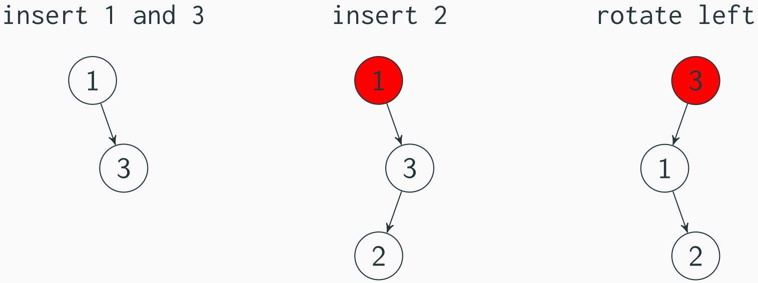
A second case...

Now, try this. Insert 1, 3, then 2. What's the issue?



A second case...

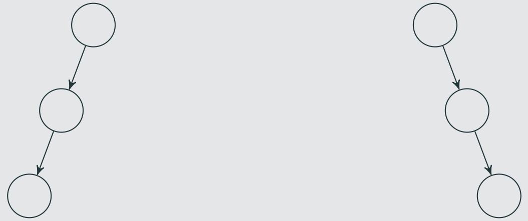
Now, try this. Insert 1, 3, then 2. What's the issue?



Tree is still unbalanced!

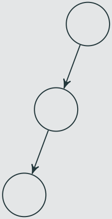
The two AVL cases

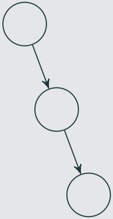
The “line” case



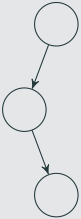
The two AVL cases

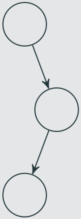
The “line” case





The “kink” case





Handling the "kink" case

Insight: Handling the kink case is hard. Can we somehow convert the kink case into the line case?

Handling the “kink” case

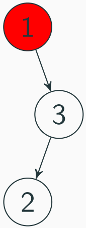
Insight: Handling the kink case is hard. Can we somehow convert the kink case into the line case?

Solution: Yes, use two rotations!

Let's try again

A second attempt...

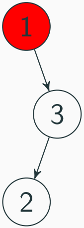
```
insert 1, 3, 2
(unbalanced!)
```



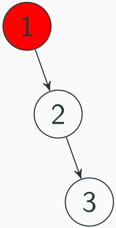
Let's try again

A second attempt...

`insert 1, 3, 2`
`(unbalanced!)`

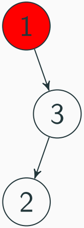


`double-rotate:`
`convert to line`

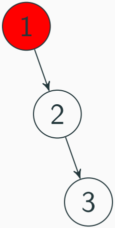


A second attempt...

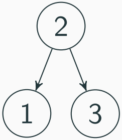
`insert 1, 3, 2`
`(unbalanced!)`



`double-rotate:`
`convert to line`

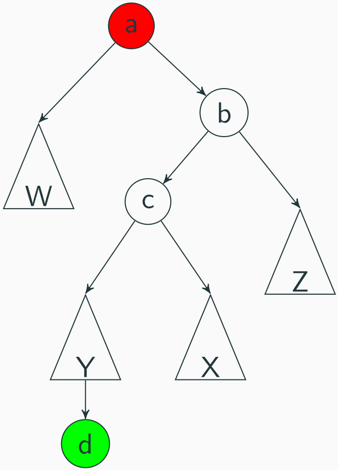


`double-rotate:`
`fix tree`



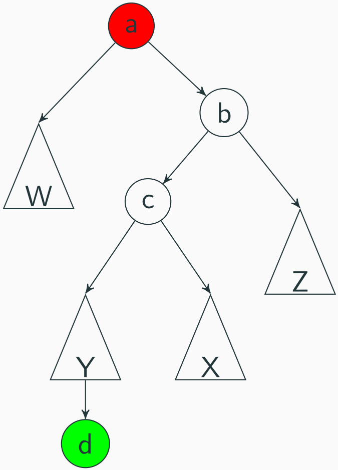
The kink case: rotation 1

Initial tree
(Unbalanced)

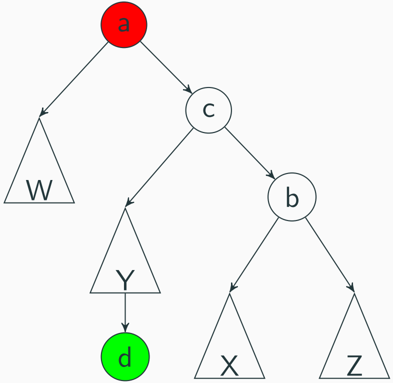


The kink case: rotation 1

Initial tree
(Unbalanced)

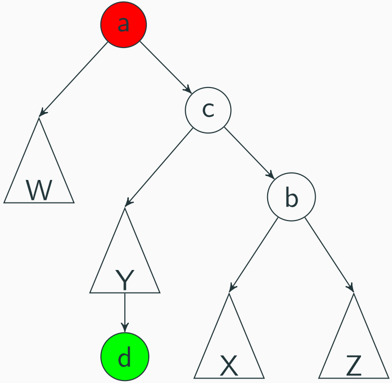


Fix the inner "b" subtree:



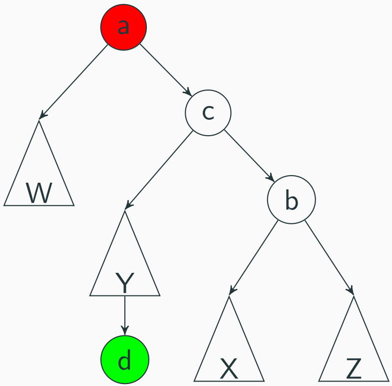
The kink case: rotation 2

After fixing the “b” subtree

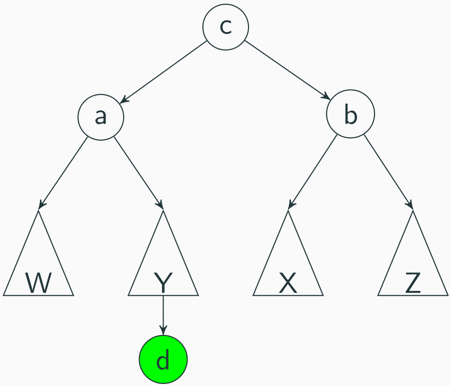


The kink case: rotation 2

After fixing the “b” subtree



Fix the outer “a” subtree:



Practice

Try inserting a, b, e, c, d into an AVL tree.

Practice

Try inserting a, b, e, c, d into an AVL tree.

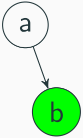
```
insert a
```

Practice

Try inserting a, b, e, c, d into an AVL tree.

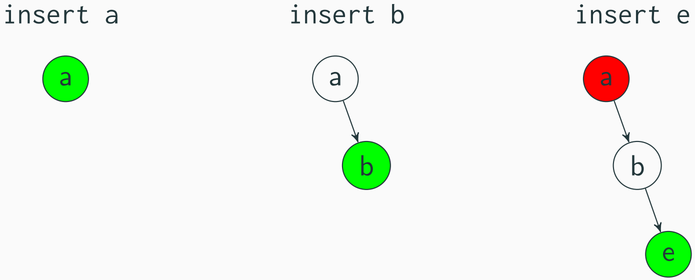
`insert a`

`insert b`



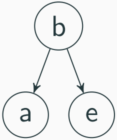
Practice

Try inserting a, b, e, c, d into an AVL tree.



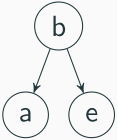
Practice

`rotate left on a`

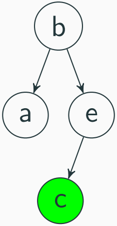


Practice

`rotate left on a`

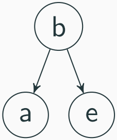


`insert c`

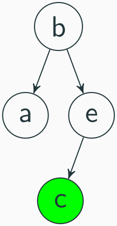


Practice

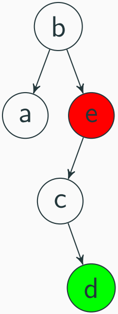
`rotate left on a`



`insert c`

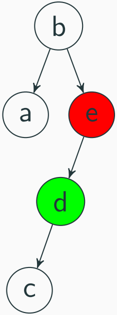


`insert d`



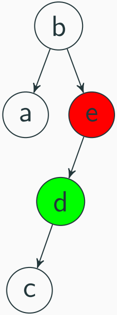
Practice

```
double rotation on e,
       part 1
```

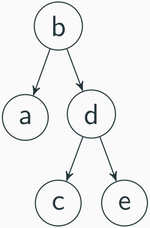


Practice

double rotation on e,
part 1



double rotation on e,
part 2



In summary...

In summary...

Implementing AVL operations

- **get:** Same as BST!
- **containsKey:** Same as BST!
- **put:** Do BST insert, move up tree, perform single or double rotations to balance tree
- **remove:** Either lazy-delete or use similar method to insert

A note on implementation

We sometimes need to rotate left, rotate right, double-rotate left, or double-rotate right.

Do we need to implement 4 methods?

A note on implementation

We sometimes need to rotate left, rotate right, double-rotate left, or double-rotate right.

Do we need to implement 4 methods?

No: can reduce redundancy by having an *array* of children instead of using left or right fields. This lets us refer to children by index so we only have to write two methods: rotate, and double-rotate.

(E.g. we can have “rotate” accept two ints: the index to the “bigger” subtree, and the index to the “smaller” subtree)

Analyzing Arraylist add

And now, for a completely unrelated topic...

Analyzing ArrayList add

Exercise: model the worst-case runtime of ArrayList's add method in terms of n, the number of items inside the list:

```
public void add(T item) {
    if (array is full) {
        resize and copy
    }
    this.array[this.size] = item;
    this.size += 1;
}
```

Analyzing ArrayList add

Exercise: model the worst-case runtime of ArrayList's add method in terms of n, the number of items inside the list:

```
public void add(T item) {
    if (array is full) {
        resize and copy
    }
    this.array[this.size] = item;
    this.size += 1;
}
```

Answer: $T(n) = \begin{cases} c & \text{when the array is not full} \\ n + c & \text{when the array is full} \end{cases}$

So, in the **WORST** possible case, what's the runtime? .

Analyzing ArrayList add

Exercise: model the worst-case runtime of ArrayList's add method in terms of n, the number of items inside the list:

```
public void add(T item) {
    if (array is full) {
        resize and copy
    }
    this.array[this.size] = item;
    this.size += 1;
}
```

Answer: $T(n) = \begin{cases} c & \text{when the array is not full} \\ n + c & \text{when the array is full} \end{cases}$

So, in the **WORST** possible case, what's the runtime? $\Theta(n)$.

Analyzing ArrayList add

Question: what's the runtime **on average**?

Analyzing ArrayList add

Question: what's the runtime **on average**?

Core idea: cost of resizing is *amortized* over the subsequent calls

Analyzing ArrayList add

Question: what's the runtime **on average**?

Core idea: cost of resizing is *amortized* over the subsequent calls

Metaphors:

- When you pay rent, that large cost is *amortized* over the following month
- When you buy an expensive machine, that large cost is *amortized* and pays itself back over the next several years

Analyzing ArrayList's add

Our recurrence: $T(n) = \begin{cases} c & \text{when the array is not full} \\ n + c & \text{when the array is full} \end{cases}$

Scenario:

Let's suppose the array initially has size k. Let's also suppose the array initially is at capacity.

Analyzing ArrayList's add

Our recurrence: $T(n) = \begin{cases} c & \text{when the array is not full} \\ n + c & \text{when the array is full} \end{cases}$

Scenario:

Let's suppose the array initially has size k. Let's also suppose the array initially is at capacity.

- How much work do we need to do to resize once then fill back up to capacity?
- What is the *average* amount of work done?

Analyzing ArrayList's add

Our recurrence: $T(n) = \begin{cases} c & \text{when the array is not full} \\ n + c & \text{when the array is full} \end{cases}$

Scenario:

Let's suppose the array initially has size k. Let's also suppose the array initially is at capacity.

- How much work do we need to do to resize once then fill back up to capacity?

  $1 \cdot (k + c) + (k - 1) \cdot c = k + ck$.
  Note: since array was full, $n = k$ in first resize

- What is the *average* amount of work done?

  $$\frac{k + ck}{k} = 1 + c$$

Analyzing ArrayList's add variations

Now, what if instead of resizing by doubling, what if we increased the capacity by 100 each time?

Analyzing ArrayList's add variations

Now, what if instead of resizing by doubling, what if we increased the capacity by 100 each time?

- Assuming we're full, how much work do we do in total to resize once then fill back up to capacity?
- What is the *average* amount of work done?

Analyzing ArrayList's add variations

Now, what if instead of resizing by doubling, what if we increased the capacity by 100 each time?

- Assuming we're full, how much work do we do in total to resize once then fill back up to capacity?

  $1 \cdot (k + c) + 99 \cdot c = k + 100c$

- What is the *average* amount of work done?

  $\frac{k + 100c}{100} = \frac{k}{100} + c$

Analyzing ArrayList's add variations

Now, what if instead of resizing by doubling, what if we increased the capacity by 100 each time?

- Assuming we're full, how much work do we do in total to resize once then fill back up to capacity?

  $1 \cdot (k + c) + 99 \cdot c = k + 100c$

- What is the *average* amount of work done?

  $$\frac{k + 100c}{100} = \frac{k}{100} + c$$

  What is k? k is the value of n each time we resize. If we plot this, we'll get a step-wise function that grows linearly!

Analyzing ArrayList's add variations

Now, what if instead of resizing by doubling, what if we increased the capacity by 100 each time?

- Assuming we're full, how much work do we do in total to resize once then fill back up to capacity?

  $1 \cdot (k + c) + 99 \cdot c = k + 100c$

- What is the *average* amount of work done?

  $$\frac{k + 100c}{100} = \frac{k}{100} + c$$

  What is k? k is the value of n each time we resize. If we plot this, we'll get a step-wise function that grows linearly!

  So, add would be in $\Theta(n)$.

Analyzing ArrayList's add variations

Now, what if instead of resizing by doubling, we triple?

Analyzing ArrayList's add variations

Now, what if instead of resizing by doubling, we triple?

- Assuming we're full, how much work do we do in total to resize once then fill back up to capacity?
- What is the *average* amount of work done?

Analyzing ArrayList's add variations

Now, what if instead of resizing by doubling, we triple?

- Assuming we're full, how much work do we do in total to resize once then fill back up to capacity?

  $1 \cdot (k + c) + (2k - 1) \cdot c = k + 2kc$

- What is the *average* amount of work done?

  $\frac{k + 2kc}{2k} = \frac{1}{2} + c$

Analyzing ArrayList's add variations

Now, what if instead of resizing by doubling, we triple?

- Assuming we're full, how much work do we do in total to resize once then fill back up to capacity?

  $1 \cdot (k + c) + (2k - 1) \cdot c = k + 2kc$

- What is the *average* amount of work done?

  $\frac{k + 2kc}{2k} = \frac{1}{2} + c$

  So, add would be in $\Theta(1)$.

Amortized analysis

This is called *amortized analysis*. The technique we discussed:

- **Aggregate analysis:**
  Show a series of n operations has an upper-bound of $T(n)$. The average cost is then $\frac{T(n)}{n}$.

Amortized analysis

This is called *amortized analysis*. The technique we discussed:

- **Aggregate analysis:**

  Show a series of n operations has an upper-bound of $T(n)$. The average cost is then $\frac{T(n)}{n}$.

Other common techniques (not covered in this class):

- **The accounting method:**

  Assign each operation an "amortized cost", which may differ from actual cost. If amortized cost $>$ actual cost, incur credit. Credit is later used to pay for operations where amortized cost $<$ actual cost.
- **The potential method:**

  The data structure has "potential energy", different operations alter that energy.

  Hooray, physics metaphors?