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, \(|\text{height(left)} - \text{height(right)}| \leq 1\).

AVL = Adelson-Velsky and Landis

**Interlude: Exploring the balance invariant**

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

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

What happens if we insert two elements. What happens?

**AVL tree invariants review**

Question: is this a valid AVL tree?

```
       7
      /\  \\
     4  10
    / \  /  \\
   3  5 9 12
  /  \ /  /  \\
 2  6 8 11 13
      /  \\
      14
```

Question: is this also an AVL tree?

```
       6
      /  \\
     2   8
  /  \\
 1  7 12
  /  /  \\
 3  5 10 13
  /  /  \\
 9 11
```
AVL tree invariants review

Question: ...and what about now?

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)
- 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:

Practice

Practice: insert 16, and fix the tree:

Step 1: insert 16
Practice

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

```
  15
   /
  8  22
 /   /
4 10 19 24
/   /   /
3  6 17 20
```

Practice

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

```
  15
   /
  8  19
 /   /
4 10 22
/   /   /
3  6 16 20
```

A second case...

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

```
insert 1 and 3
     1
      /
     3
```
```
insert 2
   3
  / 
2 1
```
```
rotate left
```

Tree is still unbalanced!

The two AVL cases

The “line” case

```
  1
 /
 2
```

The “kink” case

```
  1
 /
 2
```

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!)
1
/
3
```
```
double-rotate: convert to line
2
/
3
```
```
double-rotate: fix tree
2
/
3
```
```
1
/
3
```
The kink case: rotation 1

Initial tree (Unbalanced)

- Fix the inner "b" subtree:

After fixing the "b" subtree

Fix the outer "a" subtree:

Practice

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

- insert a
- insert b
- insert e

Practice

- rotate left on a
- insert c
- insert d

Practice

- double rotation on e, part 1
- double rotation on e, part 2

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?

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**

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

```java
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 variations**

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

**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

**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?
  
  \[
  \frac{k + ck}{k} = 1 + c
  \]

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?

▶ 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 \( \frac{T(n)}{n} \).

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?