AVL – Proof of Runtime
On Friday, we proved an upper-bound on the height of an AVL tree is $2^\lg(n)$ or $O(\lg(n))$.

### AVL Trees vs. Red-Black Trees

<table>
<thead>
<tr>
<th>AVL Trees</th>
<th>Red-Black Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balanced BST</td>
<td>Balanced BST</td>
</tr>
<tr>
<td>Max height: $1.44 \times \lg(n)$</td>
<td>Max height: $2 \times \lg(n)$</td>
</tr>
<tr>
<td>Q: Why is our proof $2^\lg(n)$?</td>
<td>Rotations:</td>
</tr>
<tr>
<td>- find:</td>
<td>- find:</td>
</tr>
<tr>
<td>- insert:</td>
<td>- insert:</td>
</tr>
<tr>
<td>- remove:</td>
<td>- remove:</td>
</tr>
</tbody>
</table>

In CS 225, we learned AVL trees because they’re intuitive and I’m certain we could have derived them ourselves given enough time. A red-black tree is simply another form of a balanced BST that is also commonly used.

### Summary of Balanced BSTs:
(Includes both AVL and Red-Black Trees)

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using a Red-Black Tree in C++
C++ provides us a balanced BST as part of the standard library:
```cpp
std::map<K, V> map;
```

The map implements a dictionary ADT. Primary means of access is through the overloaded `operator[]`:
```cpp
V & std::map<K, V>::operator[]( const K & );
```

This function can be used for both insert and find!

Removing an element:
```cpp
void std::map<K, V>::erase( const K & );
```

Range-based searching:
```cpp
iterator std::map<K, V>::lower_bound( const K & );
iterator std::map<K, V>::upper_bound( const K & );
```

### Iterators and MP4
Three weeks ago, you saw that you can use an iterator to loop through data:
```cpp
DFS dfs(...);
for ( ImageTraversal::Iterator it = dfs.begin(); it != dfs.end(); ++it ) {
    std::cout << (*it) << std::endl;
}
```

You will use iterators extensively in MP4, creating them in Part 1 and then utilizing them in Part 2. Given the iterator, you can use the for-each syntax available to you in C++:
```cpp
DFS dfs(...);
for ( const Point & p : dfs ) {
    std::cout << p << std::endl;
}
```

The exact code you might use will have a generic `ImageTraversal`:
Running Time of Every Data Structure So Far:

<table>
<thead>
<tr>
<th></th>
<th>Unsorted Array</th>
<th>Sorted Array</th>
<th>Unsorted List</th>
<th>Sorted List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traverse</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consider Instagram profile data:

<table>
<thead>
<tr>
<th>How many profiles?</th>
<th>How much data /profile?</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVL Tree</td>
<td>BTree</td>
</tr>
</tbody>
</table>

Tree Height

**BTree Motivations**

Knowing that we have long seek times for data, we want to build a data structure with two (related) properties:

1. 

2. 

**BTree**

Big-O assumes uniform time for all operations, but this isn't always true.

However, seeking data from the cloud may take 100ms+.

...an O(lg(n)) AVL tree no longer looks great:

**Goal**: Build a tree that uses ______________ /node!

...optimize the algorithm for your platform!

**CS 225 – Things To Be Doing:**

1. Final Project Teams due March 26th!
2. mp_mosaic due on March 29th!
3. lab_trees due on March 28th!
4. Daily POTDs are ongoing!