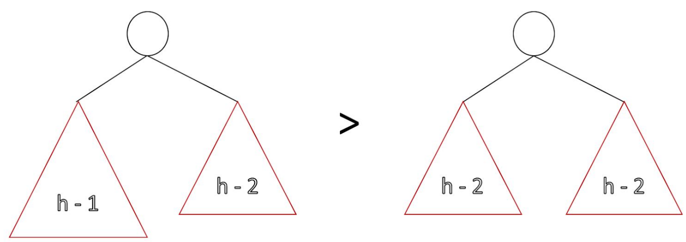
CS 225 Spring 2019 :: TA Lecture Notes
3/4 AVL Application

By Wenjie

- **AVL summary**
  - AVL is a balanced BST.
  - Maximal height is 1.44 * O(lg(n)) = O(lg(n))



  - The most important thing is that the running time is O(lg(n)) for all operations.
  - Number of rotations
    - **Find** → 0
    - **Insert** → up to 1 (L, R, LR, or RL)
    - **Delete** → up to h (O(lg(n)))

  - **Red Black Tree**
    - These are almost the same as AVLs.
    - Maximal height is 2 * lg(n) = O(lg(n)). //AVL is better
    - All operations run in O(lg(n)).
    - Constant time rotations for all operations → for red-black trees there can be a rotation during find(...).

Operation	Worst case running time	AVL Tree max Rotations	Red Black Tree max rotations
find	O(h)=O(lg n)	0	0
insert	O(h)=O(lg n)	1	2
remove	O(h)=O(lg n)	h	3

CS 225 Spring 2019 :: TA Lecture Notes
3/4 AVL Application

By Wenjie

- When we see "AVL" or "Red black tree", we think about **Balanced BST** with O(**log n)** runtime on every operation

- **Advantage of AVL (or Balanced BSTs in general)**
    - Running time for <u>all operations O(lg(n))</u>.
        - Improvement over: Arrays, Linked Lists
    - AVLs are great for specific applications when exact key is unknown:
        - Nearest neighbour search.
        - Range search (return all elements in between range) → O(lg(n)) to find start and end point, and O(k) for traversing those k elements in the range - overall running time is O(lg(n)) + O(k)
            - Nearby/nearest neighbor search: since when we zone into the data in the tree, we are also close to the "nearby" data - so traversal is easier
- **Disadvantages**
    - It is not O(1)
        - List has insertFront and insertBack with O(1) running time
        - lg(n) still grows as n grows.
        - If we have exact key, hash table runs in O(1).
    - All data has to be in the memory
        - All data must be in main memory which is bad for big data. For example, if the tree is on a server, the client must download the entire (left/right) tree for a search...
        - But we have B-trees are used to solve that problem!

- **Standard Map in C++**
    - Balanced tree in C++ is implemented using red-black trees.
    - std::map<K, V> map; → this is a tree map, not a hash map.
    - Find: operator[](const & K);
    - Add: map[42] = "Hello"; → 42 is the key and "Hello" is the value.
    - Delete: map.erase(42);
    - **Range Traversal iterator**
        - There are two functions returning iterators:
            - lower_bound(const & K)

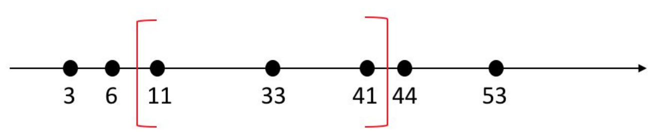
            - upper_bound(const & K)   <- one past that element
        - Stopping condition: lower_bound == upper_bound.
        - Iterators are useful because they allow abstraction and make the syntax very clean.
            - In MP4 example: ImageTraversal & traversal = /* … */
              for (const Point & p : traversal) {…}

- **Summary :: Every Data Structure So far**

Worst runtime	Unsorted Array	Sorted Array	Unsorted List	Sorted List	Binary Tree (unsorted)	BST	AVL
find	O(n)	O(lg n) Binary search	O(n)	O(n)	O(n)	O(h)<=n	O(lg n)
insert	O(1)* InsertEnd and resize properly	O(n) Shifting up to ½ data	O(1) InsertFron t	O(n)	O(1) Insert at root	O(h)<=n	O(lg n)
remove	O(n)	O(n)	O(n)	O(n)	O(n)	O(h)<=n	O(lg n)
traverse	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)

*: amortized runtime

- **Range based search**
    - Consider points p = {p_1, p_2, p_3, p_4, … , p_n}
        - What points fall in range [11, 42].

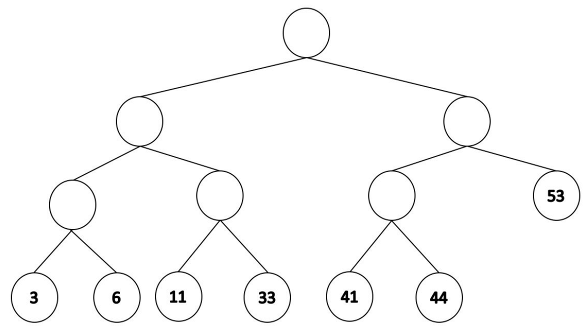


    - Find lowest element, find highest point, and list all elements → lg(n) + k
    - In order to compute this, we build a tree bottom up such that:
        - All nodes in T_L ≤ data.
        - Data is contained only in leaf nodes.

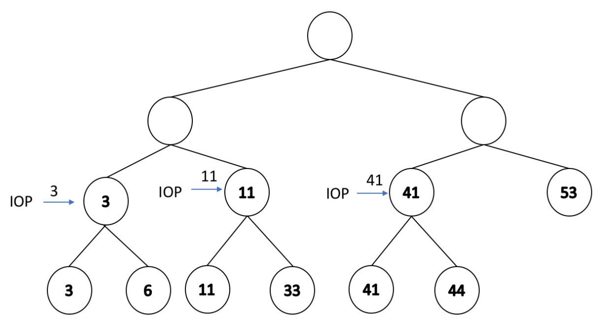
CS 225 Spring 2019 :: TA Lecture Notes
3/4 AVL Application

By Wenjie

Step 1: add all data to the leaves (there are as many leaves as there are data points).



Step 2: Go to the parent and compute IOP from there. The IOP is the value that will be in the parent node.



Step 3: Repeat step 2 as we go up the tree.