CS 225

#21: AVL Applications

October 15, 2018 · *Wade Fagen-Ulmschneider*

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))**. Both BST, Functionality "same", can be replaced

AVL Trees	Red-Black Trees
Balanced BST	Balanced BST *Functionally equivalent to AVL trees; all key operations runs in O(h) time.*
Max height: 1.44 * lg(n) *Q: Why is our proof 2*lg(n)?*	Max height: 2 * lg(n)
Rotations: → refer to L, R, LR, RL - find: zero rotation - insert: one rotation - remove: O(h) == O(lg(n)) rotations	Rotations: Constant numbers - find: - insert: - remove:

*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)

Advantages	Disadvantages
① Running time O(lg(n)) Better than: Array, Linked List ② Great for specific applications key not exactly known Nearest Neighbors → kd Tree (lg(n)) Range Finding	① Running Time: not O(1) ↳ Hashing solves finding key ~ O(1) ② In-memory Requirement → All data must be in main memory Not for Big Data

Using a Red-Black Tree in C++

C++ provides us a balanced BST as part of the standard library:

```
std::map<K, V> map;
```

The map implements a dictionary ADT. Primary means of access is through the overloaded `operator[]`:

```
V & std::map<K, V>::operator[]( const K & )
```

This function can be used for both insert and find!

Removing an element:

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

Range-based searching:

```
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:

```
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++:

```
DFS dfs(...);
for ( const Point & p : dfs ) {
  std::cout << p << std::endl;
}
```

The exact code you might use will have a generic `ImageTraversal`:

```
ImageTraversal & traversal = /* ... */;
for ( const Point & p : traversal ) {
  std::cout << p << std::endl;
}
```

Running Time of Every Data Structure So Far:

	Unsorted Array	Sorted Array	Unsorted List	Sorted List
Find	O(n)	O(lg(n))	O(n)	O(n)
Insert	O(1)* more space → double size	O(n) → copy ½ n elements	O(1)	Find + O(1)
Remove	O(n)	O(n)	Find + O(1)	Find + O(1)
Traverse	O(n) →			

	Binary Tree	BST h ≤ n	AVL h ~ lg(n)
Find	O(n)		
Insert	O(1) add at root, rest as left/right tree		
Remove	O(1) updating ptr		
Traverse			

Range-based Searches:

Q: Consider points in 1D: $p = \{p_1, p_2, \ldots, p_n\}$.

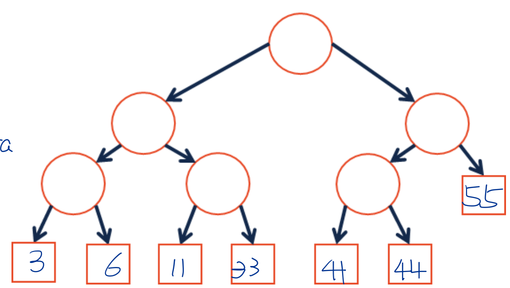
...what points fall in [11, 42]?

O(lg(n)) Time
+ k elements
Best Result

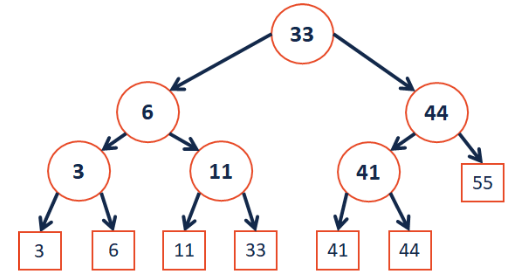
Tree Construction:

① T_L every element ≤ root

② Only leaf nodes contain data



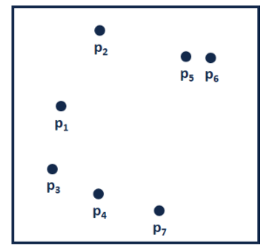
Range-based Searches:



Running Time:

Extending to k-dimensions:

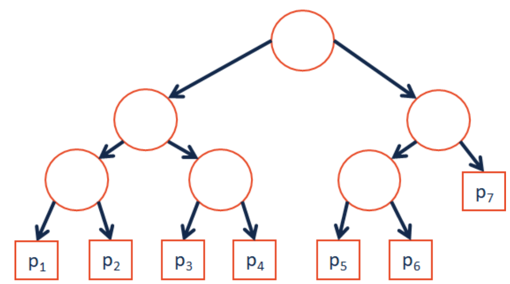
Consider points in 2D: $\mathbf{p} = \{p_1, p_2, \ldots, p_n\}$:



...what points are inside a range (rectangle)?

...what is the nearest point to a query point **q**?

Tree Construction:



CS 225 – Things To Be Doing:

1. Programming Exam B starts in 10 days *(grab your time slot!)*
2. MP4 extra credit +7 due tonight
3. lab_avl released this week; details on Wednesday
4. Daily POTDs are ongoing!