

# CSE 332

## Data Structures and Parallelism

# Dictionarys & Trees



# Outline

1 Dictionaries & Sets

2 Vanilla BSTs

## Where We've Been So Far

- Stack (Get LIFO)
- Queue (Get FIFO)
- Priority Queue (Get By Priority)

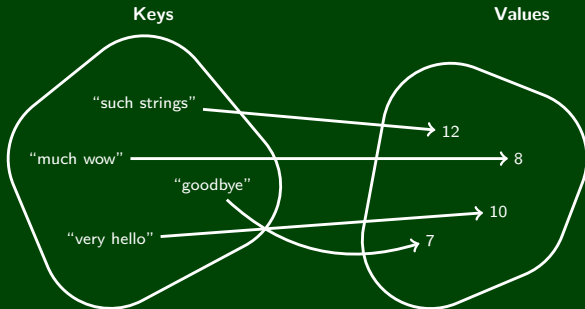
Today, we begin discussing **Maps**. This ADT is hugely important.

# A New ADT: “Dictionaries” (Also called “Maps”)

2

## Dictionary ADT

Data	Set of (key, value) pairs
insert( <b>key</b> , <b>val</b> )	Places ( <b>key</b> , <b>val</b> ) in map (overwrites existing <b>val</b> entry)
find( <b>key</b> )	Returns the <b>val</b> currently associated to <b>key</b>
delete( <b>key</b> )	deletes any pair relating <b>key</b> from the map



find("such strings") → 12

Dictionaries are the **more general** structure, but, in terms of implementation, they're nearly identical.

In a Set, we store the key directly, but conceptually, there's nothing different in storing an **Item**:

```
1 class Item {  
2     Data key;  
3     Data value;  
4 }
```

The Set ADT usually has our favorite operations: intersection, union, etc.

Notice that union, intersection, etc. **still make sense on maps!**

As always, depending on our usage, we might choose to add/delete things from our ADT.

Bottom Line: If we have a set implementation, we also have a valid dictionary implementation (and vice versa)!

It turns out dictionaries are super useful. They're a natural generalization of arrays. Instead of storing data at an index, we store data at **anything**.

- Networks: router tables
- Operating Systems: page tables
- Compilers: symbol tables
- Databases: dictionaries with other nice properties
- Search: inverted indexes, phone directories, ...
- Biology: genome maps

For each of the following potential implementations, what is the worst case runtime for `insert`, `find`, `delete`?

- Unsorted Array
- Unsorted Linked List
- Sorted Linked List
- Sorted Array List



For each of the following potential implementations, what is the worst case runtime for insert, find, delete?

- Unsorted Array

**Insert** by searching for existence and inserting which is  $\mathcal{O}(n)$

**Find** by linear search which is  $\mathcal{O}(n)$

**Delete** by linear search AND shift which is  $\mathcal{O}(n)$

- Unsorted Linked List

**Insert** by searching for existence and inserting which is  $\mathcal{O}(n)$

**Find** by linear search which is  $\mathcal{O}(n)$

**Delete** by linear search AND shift which is  $\mathcal{O}(n)$

- Sorted Linked List

**Insert** by searching for existence and inserting which is  $\mathcal{O}(n)$

**Find** by linear search which is  $\mathcal{O}(n)$

**Delete** by linear search AND shift which is  $\mathcal{O}(n)$

- Sorted Array List

**Insert** by binary search AND shift which is  $\mathcal{O}(n)$

**Find** by binary search which is  $\mathcal{O}(\lg n)$

**Delete** by binary search AND shift which is  $\mathcal{O}(n)$

It turns out there are **many** different ways to do much better.

**But they all have their own trade-offs!**

So, we'll study many of them:

- “Vanilla BSTs” – today (vanilla because they're “plain”)
- “Balanced BSTs” – there are many types: we'll study **AVL Trees**
- “B-Trees” – another strategy for **a lot of data**
- “Hashtables” – a completely different strategy (lack data ordering)

**Binary Search** is great! It's the only thing that was even sort of fast in that table. But `insert` and `delete` are really bad into a sorted array. Store the data in a structure where **most of the data isn't accessed**.

Interestingly, this is **very similar** to what made heaps useful!

To put it another way, by storing the data in an **array**, we're paying for the constant-time access that we're never even using!

It's **okay** that it takes more time to access certain elements.

... as long as it's **never** too bad.

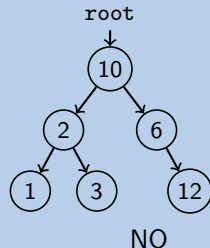
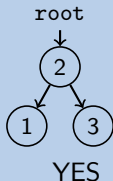
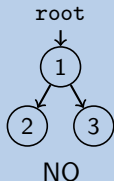
## Definition (Vanilla BST)

A binary tree is a **BST** when an **in-order traversal of the tree** yields a sorted list.

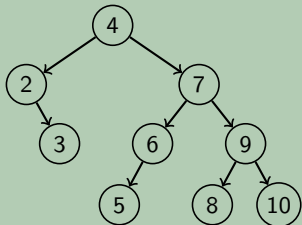
To put it another way, a binary tree is a **BST** when:

- All data “to the left of” a node is less than it
- All data “to the right of” a node is greater than it
- All sub-trees of the binary tree are also BSTs

Example (Which of the following are BSTs?)



BST Properties



**Structure Property:**

0, 1, or 2 children

**BST Property:**

Keys in Left Subtree are smaller

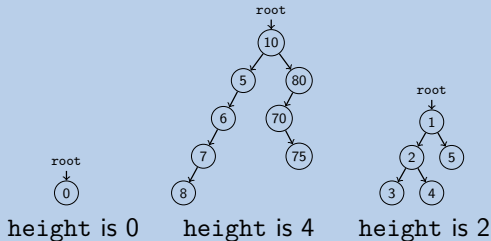
Keys in Right Subtree are larger

## Definition (Height)

The **height** of a binary tree is the length of the longest **path** from the root to a leaf.

- Height of an empty tree? **-1**
- Height of  $\otimes$ ? **0**

## height



```
1 private int height(Node current) {  
2     if (current == null) { return -1; }  
3     return 1 + Math.max(height(current.left), height(current.right));  
4 }
```

## Height

```
1 private int height(Node current) {  
2     if (current == null) { return -1; }  
3     return 1 + Math.max(height(current.left), height(current.right));  
4 }
```

Given that a tree has height  $h$ ...

- What is the maximum number of **leaves**?
- What is the maximum number of **nodes**?
- What is the minimum number of **leaves**?
- What is the minimum number of **nodes**?

**That's a big spread!**

This confirms what we already know: `height` in a tree has a big impact on runtime.

## Height

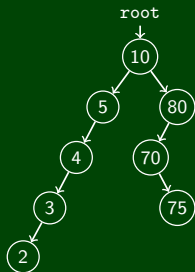
```
1 private int height(Node current) {  
2     if (current == null) { return -1; }  
3     return 1 + Math.max(height(current.left), height(current.right));  
4 }
```

Given that a tree has height  $h$ ...

- What is the maximum number of **leaves**?  $2^h$
- What is the maximum number of **nodes**?  $2^{h+1} - 1$
- What is the minimum number of **leaves**? 1
- What is the minimum number of **nodes**?  $h + 1$

**That's a big spread!**

This confirms what we already know: `height` in a tree has a big impact on runtime.



What about other finds?

- findMin?
- findMax?
- deleteMin?

## Recursive find

```

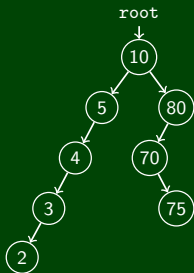
1 Data find(Key key, Node curr) {
2   if (curr == null) { return null; }
3   if (key < curr.key) {
4     return find(key, curr.left);
5   }
6   if (key > curr.key) {
7     return find(key, curr.right);
8   }
9   return curr.data;
10 }
  
```

## Iterative find

```

1 Data find(Key key) {
2   Node curr = root;
3   while (curr != null && curr.key != key) {
4     if (key < curr.key) {
5       curr = curr.left;
6     }
7     else (key > curr.key) {
8       curr = curr.right;
9     }
10  }
11  if (curr == null) { return null; }
12  return curr.data;
13 }
  
```



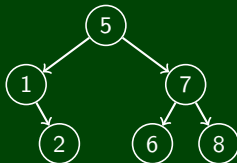


### insert

- find
- create a new node

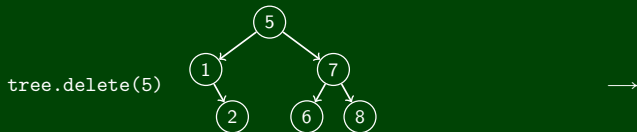
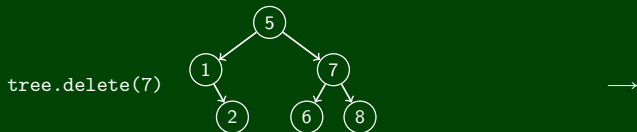
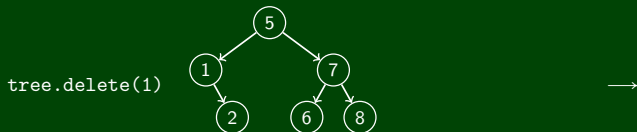
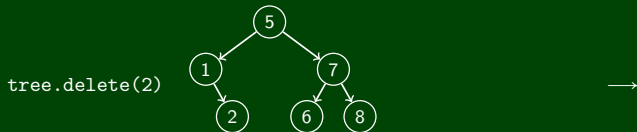
How about delete?

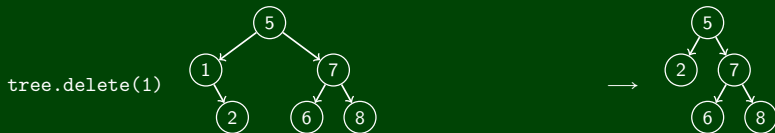
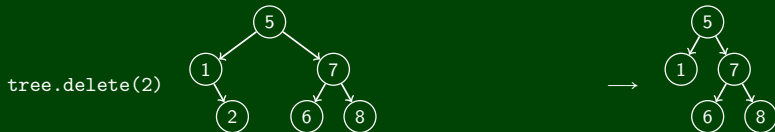
Consider the following tree:

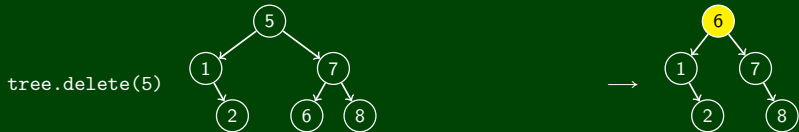
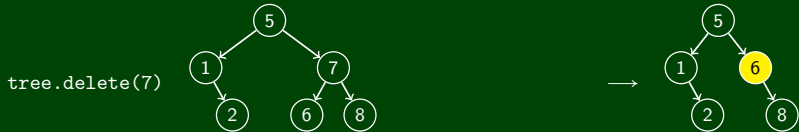
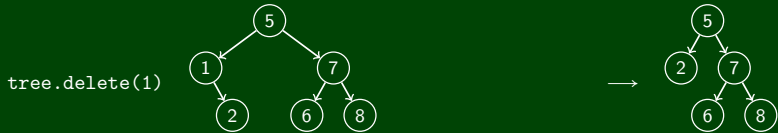
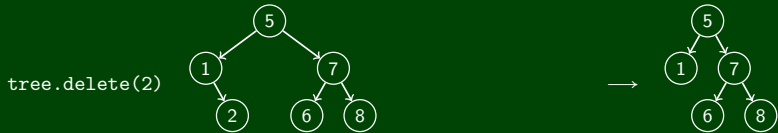


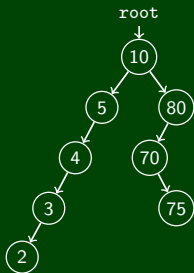
Let's try the following removals:

- `tree.delete(2)`
- `tree.delete(1)`
- `tree.delete(7)`
- `tree.delete(5)`









### delete( $x$ )

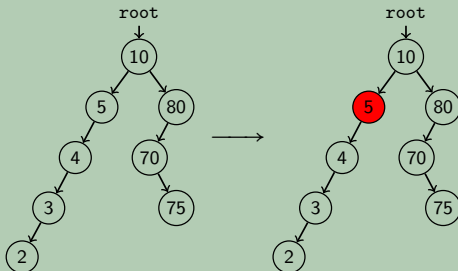
- Case 1:  $x$  is a leaf
  - Just delete  $x$
- Case 2:  $x$  has one child
  - Replace  $x$  with its child
- Case 3:  $x$  has two children
  - Replace  $x$  with the **successor** or **predecessor** of  $x$

The tricky case is when  $x$  has two children. If we think of the BST in sorted array form, to get the successor, we `findMin(right subtree)` (or predecessor is `findMax(left subtree)`)

Instead of doing this complicated algorithm, here's an idea:

Mark the node as “deleted” instead of doing anything

```
lazyDelete(5)
```



Then, insert and find change slightly, but the whole thing is much simpler.

This “lazy deletion” is a very useful strategy!

## Pseudocode

```
1 void buildTree(int[] input) {  
2     for (int i = 0; i < input.length; i++) {  
3         insert(input[i]);  
4     }  
5 }
```

What's the best case? The worst case?

The worst case is a sorted input which is  $O(n^2)$ . Ouch.

## The Good News

**On average**, we get  $O(\lg n)$  height (see textbook for proof). But we want it to **always** be  $O(\lg n)$  height. . .

## The Solution

Add restrictions on the height of the tree. Somehow, the tree should “fix itself” so it never has too large a height.

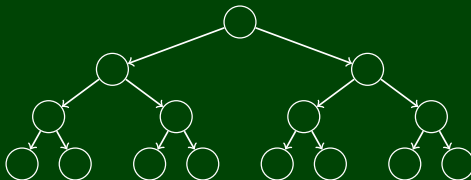
We call this condition a **Balance Condition**.





## Ideas?

- Left and right subtrees ~~of the root~~ **recursively** have the same number of nodes
- Left and right subtrees ~~of the root~~ **recursively** have the same **height**



These ideas suffer from the same problem:

They're way too strong. Only **perfect** trees satisfy them.

Left and right subtrees **recursively** have heights differing by at most one.

Definition (balance)

$$\text{balance}(n) = \text{abs}(\text{height}(n.\text{left}) - \text{height}(n.\text{right}))$$

Definition (AVL Balance Property)

An AVL tree is balanced when:

$$\text{For every node } n, \text{balance}(n) \leq 1$$

- This ensures a small depth (we'll prove this next time)
- It's relatively easy to maintain (we'll see this next time)