Splay Trees

CMSC 420

Splay trees were not covered in Spring 2019, yet we are still posting those slides for your knowledge.

"Splaying"

splay

/splā/

verb

1. thrust or spread (things, especially limbs or fingers) out and apart.
   "her hands were splayed across his broad shoulders"

noun

1. a widening or outward tapering of something, in particular.
2. a surface making an oblique angle with another, such as the splayed side of a window or embrasure.

adjective

1. turned outward or widened.
   "the girls were sitting splay-legged"

Translations, word origin, and more definitions

Core idea of splay trees

- Similar to that of amortized complexity.
- Not guaranteed to be balanced at any point in time!
- So any given search, insertion, deletion not guaranteed to be efficient ($\mathcal{O}(\log_2 n)$)!
- These trees make use of a very powerful idea in Computer Architecture, known as temporal locality

Principle of locality

- Temporal locality: Variables that are used at some time t in your program are likely to be used again quite soon.
  - So keep them in registers or cache longer than you do others!
  - This implements a Most-Recently-Used (MRU) heuristic.
  - Example: looping indices (i, j) are dereferenced and subsequently added to very soon and very often!

Principle of locality

- Spatial locality: Variables whose memory addresses are relatively close to the address of the currently manipulated variable are likely to be used soon.
  - So, once again, pull them to the faster levels of our memory hierarchy!
  - Examples: Arrays, class fields,…
- Recall: `RandomAccessBag`

Locality and splay trees

- Splay trees try to emulate this idea by keeping the key that was operated on as well as its neighbors in the tree close to the root after the operation!
  - In the case of a deletion, only the neighborhood will be pulled close to the root.

Locality and splay trees

- Splay trees try to emulate this idea by keeping the key that was operated on as well as its neighbors in the tree close to the root after the operation!
  - In the case of a deletion, only the neighborhood will be pulled close to the root.
- That way, and given the assumption of locality, future operations on either the element itself or its neighborhood will be completed in sub-logarithmic time!
  - So we might pay a lot in certain cases ($\mathcal{O}(n)$) but in the near future we will even end up with time to spare!

Wait… what do you mean “pulled close”?

- Via successive rotations!
  - Recall that those **preserve the BST property.**
- We’ll see some examples right now.
- Remember: Splay Trees are not guaranteed to be balanced at any given point in time!
  - In fact, in all but the most trivial examples, they will not be.

“Splaying” a node

- We will define a new operation called **splaying.**
- **Splaying** is like searching with steroids.
- It searches for the node first, very much like classic BST searching.
- Then, two options exist:
    1. Either we will find the node
    2. Or we won’t, but we **will** have found its parent node!
        - Spatial locality would then say: *“Make access to this parent node and maybe his close neighborhood easier for an application, since he is likely to be needed soon”*!

The "steroids" part

- Irrespective of which node we reached, our splaying operation will then start pulling the node upwards towards the root of our tree!
  - It will accomplish this **through slightly altered rotations**.
  - We will see how this works through some examples.
- In those examples, we have an existing BST on the root of which we call our splay routine: `splay(key, root)`
  - We **do not** insert or remove nodes from the tree: we are just searching for a key, with the stated intent of lifting **it** or **its parent** to the root.

Splay Tree rotations

- Since our goal is to pull the splayed key as close to the root as we can, we will try to do this faster than simple sequences of left and right rotations.
- Maintain three pointers:
  1. C, representing the child node
  2. P, representing the parent node
  3. G, representing the grandparent node
- **SIX** different kinds of rotations:
  1. Left (a simple AVL-like left rotation)
  2. Right (a simple AVL-like left rotation)

Splay Tree rotations

- Since our goal is to pull the splayed key as close to the root as we can, we will try to do this faster than simple sequences of left and right rotations.
- Maintain three pointers:
  1. C, representing the child node
  2. P, representing the parent node
  3. G, representing the grandparent node
- **SIX** different kinds of rotations:
  1. Left (a simple AVL-like left rotation)
  2. Right (a simple AVL-like left rotation)
  3. **Left-Right ("ZIG-ZAG", like AVL LR rotation)**
  4. **Right-Left ("ZAG-ZIG", like AVL RL rotation)**

Splay Tree rotations

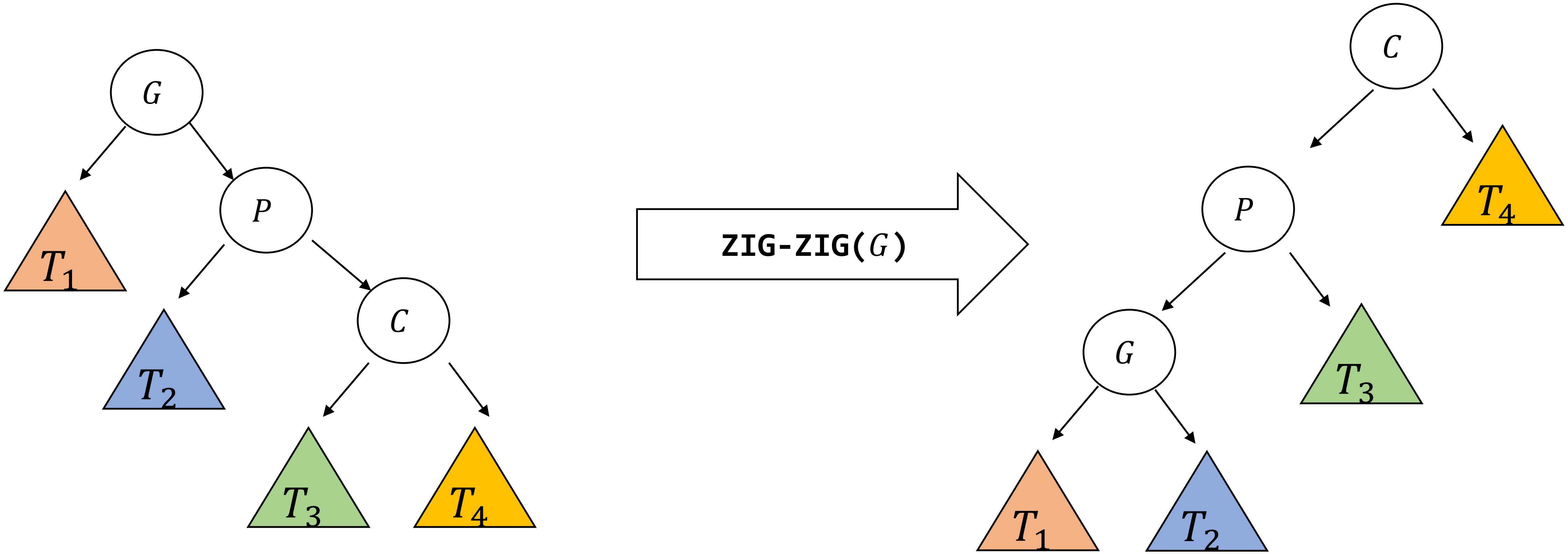
- Since our goal is to pull the splayed key as close to the root as we can, we will try to do this faster than simple sequences of left and right rotations.
- Maintain three pointers:
    1. C, representing the child node
    2. P, representing the parent node
    3. G, representing the grandparent node
- **SIX** different kinds of rotations:
    1. Left (a simple AVL-like left rotation)
    2. Right (a simple AVL-like left rotation)
    3. **Left-Right ("ZIG-ZAG", like AVL LR rotation)**
    4. **Right-Left ("ZAG-ZIG", like AVL RL rotation)**
    5. **Left-Left ("ZIG-ZIG", does <u>not</u> happen in an AVL tree)**
    6. **Right-Right ("ZAG-ZAG", does <u>not</u> happen in an AVL tree)**

Splay Tree rotations

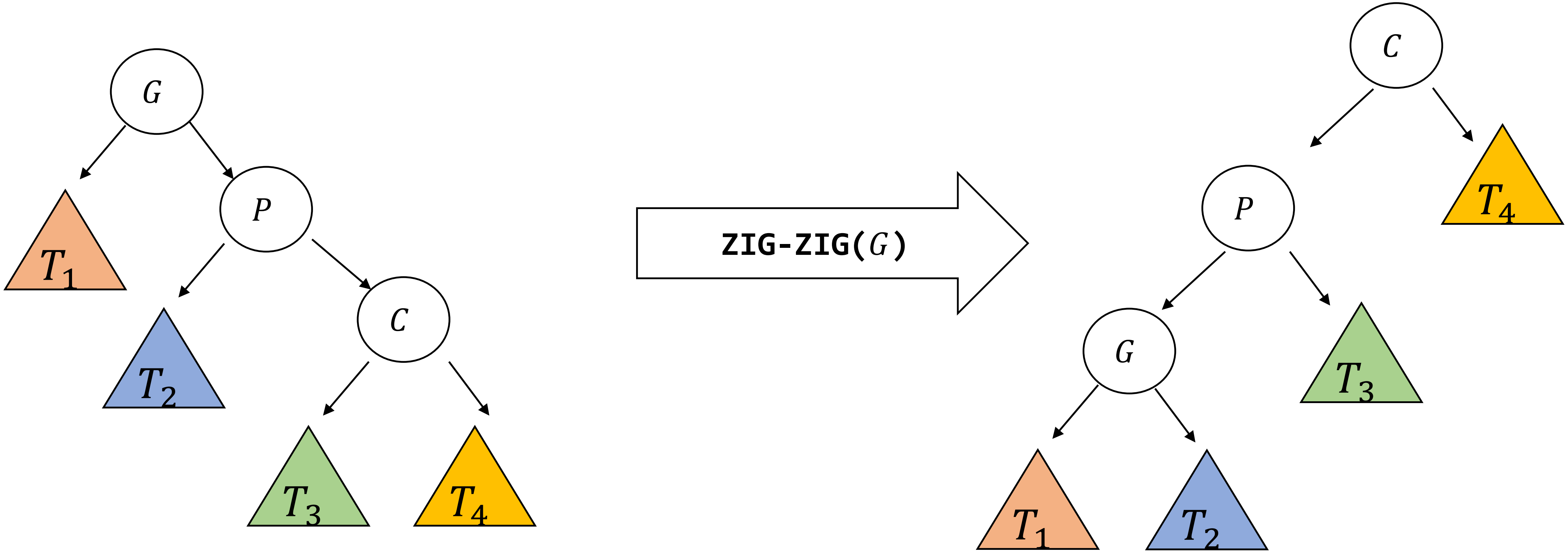
AVL Terminology	Splay Terminology
L	“ZIG”
R	“ZAG”
LR	“ZIGZAG”
RL	“ZAGZIG”
LL *(never used in AVL Trees)*	“ZIGZIG”
RR *(never used in AVL Trees)*	“ZAGZAG”

- It’s **not** important that you remember that “Zig” means left and “Zag” means right. **Just remember that there exist two patterns**: **one that looks like pulling on a rope** (zig-zig, zag-zag) and **one that looks like a ”zig-zag” pattern** (zig-zag, zag-zig).
  - In fact, Schaffer calls both “ZAGZAG” and “ZIGZIG” “ZIGZIG”, and the other two he calls “ZIGZAG”!
- The rotations that look like “zig-zags” are **exactly the RL and LR rotations we have seen in AVL trees!**

ZIG-ZIG

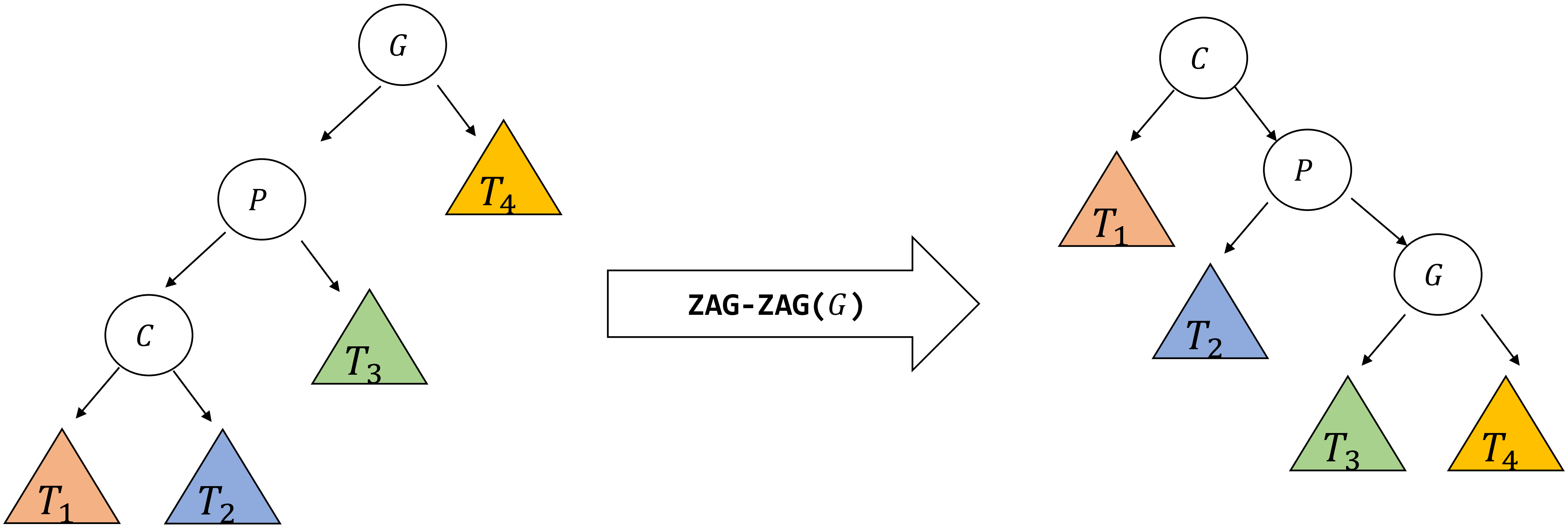


ZIG-ZIG

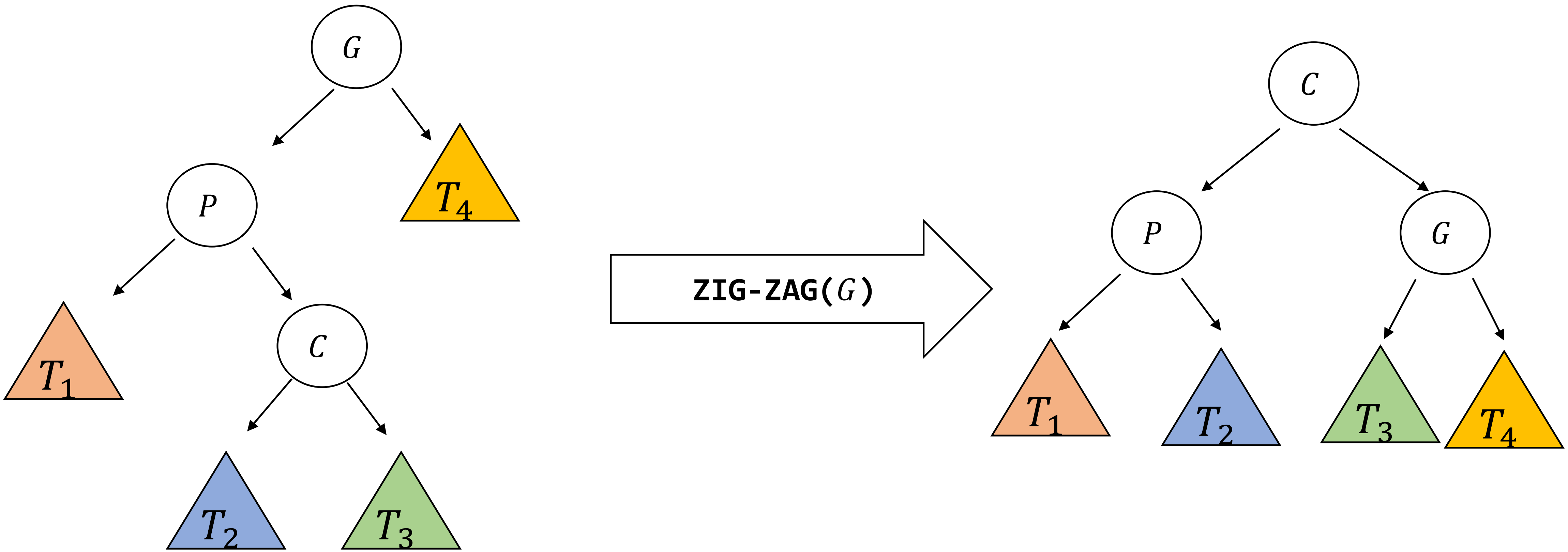


Make sure the subtree re-distributions make sense to you!

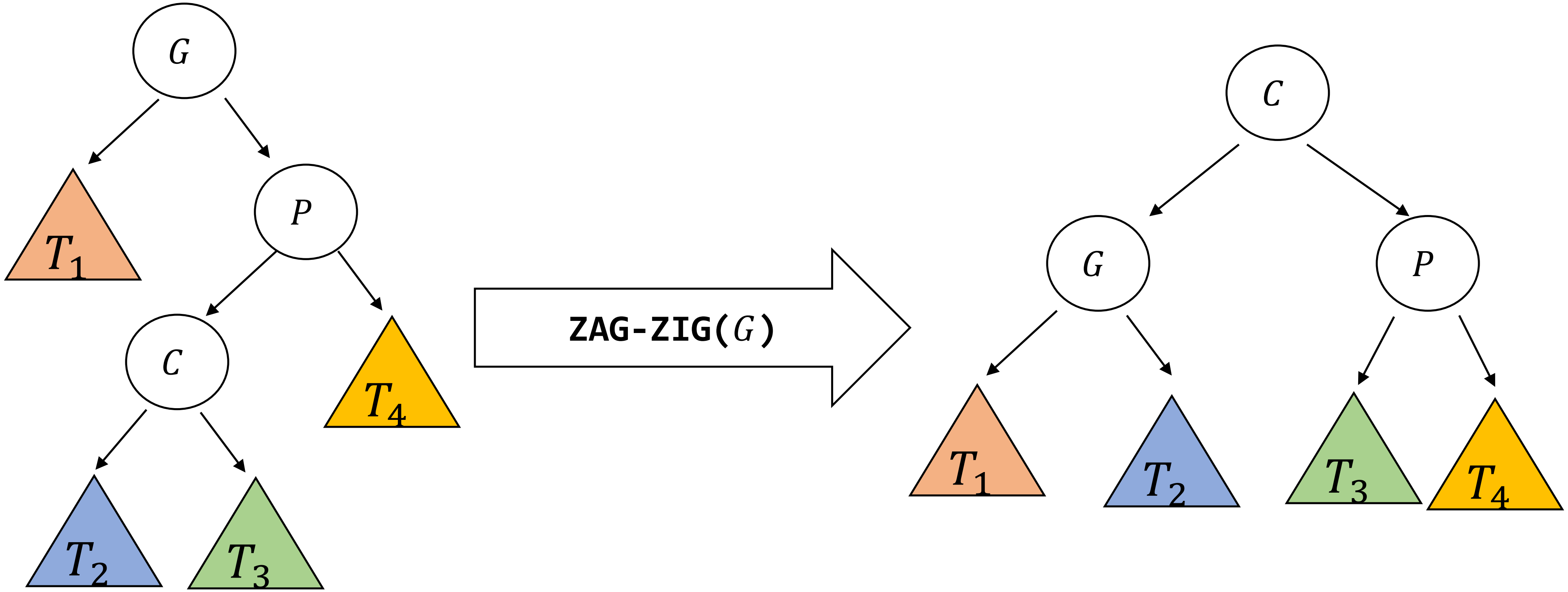
ZAG-ZAG



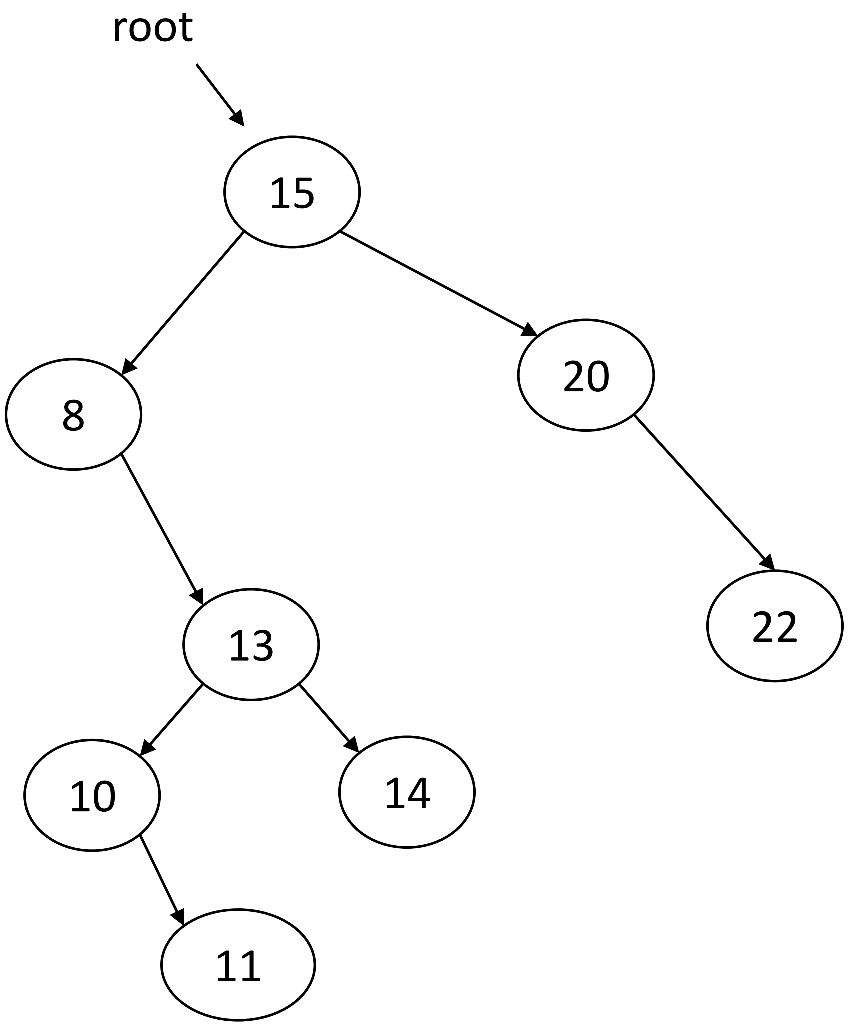
ZIG-ZAG



ZAG-ZIG



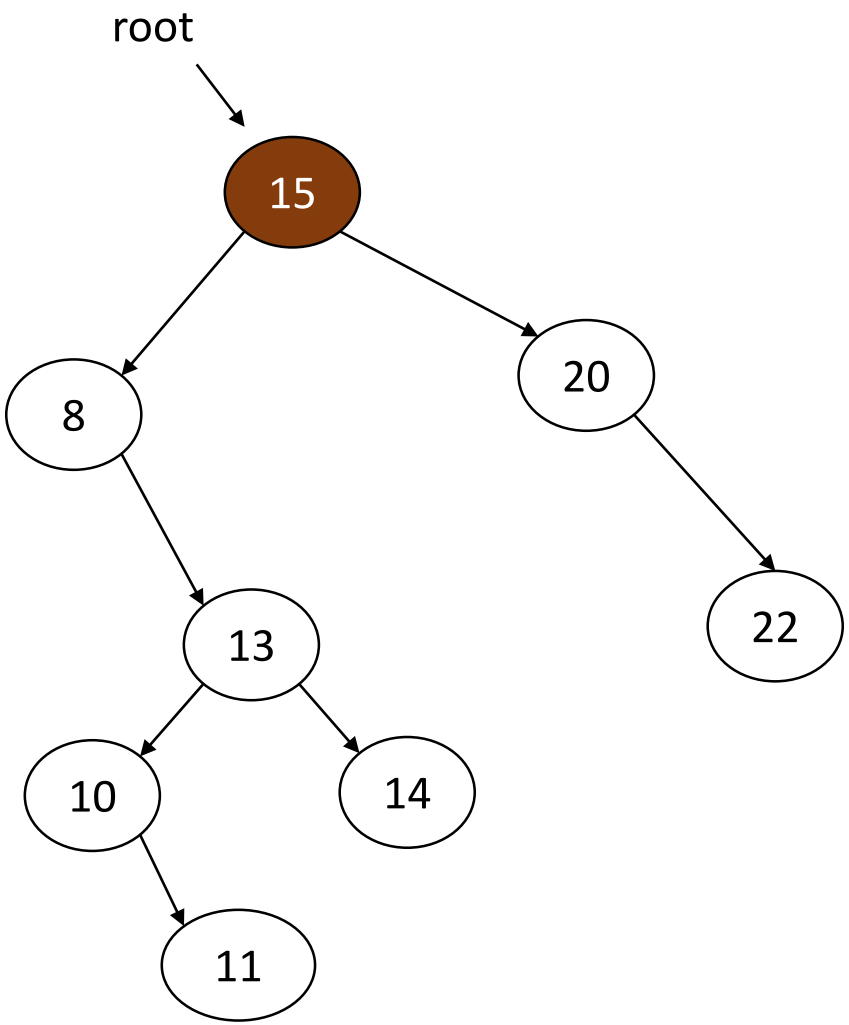
Splaying example



Splaying example

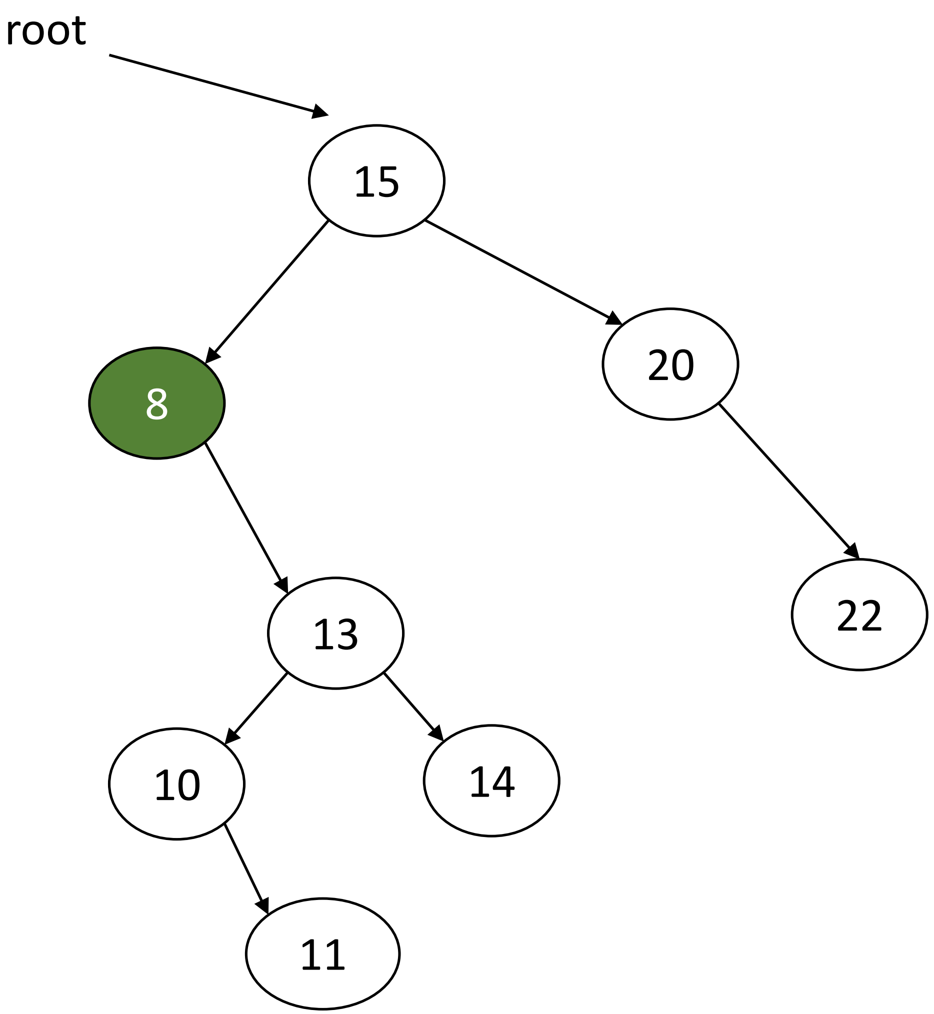
`splay(15, root)`

15 is already at the root, so there is nothing to do (best case scenario for search!)



Splaying example

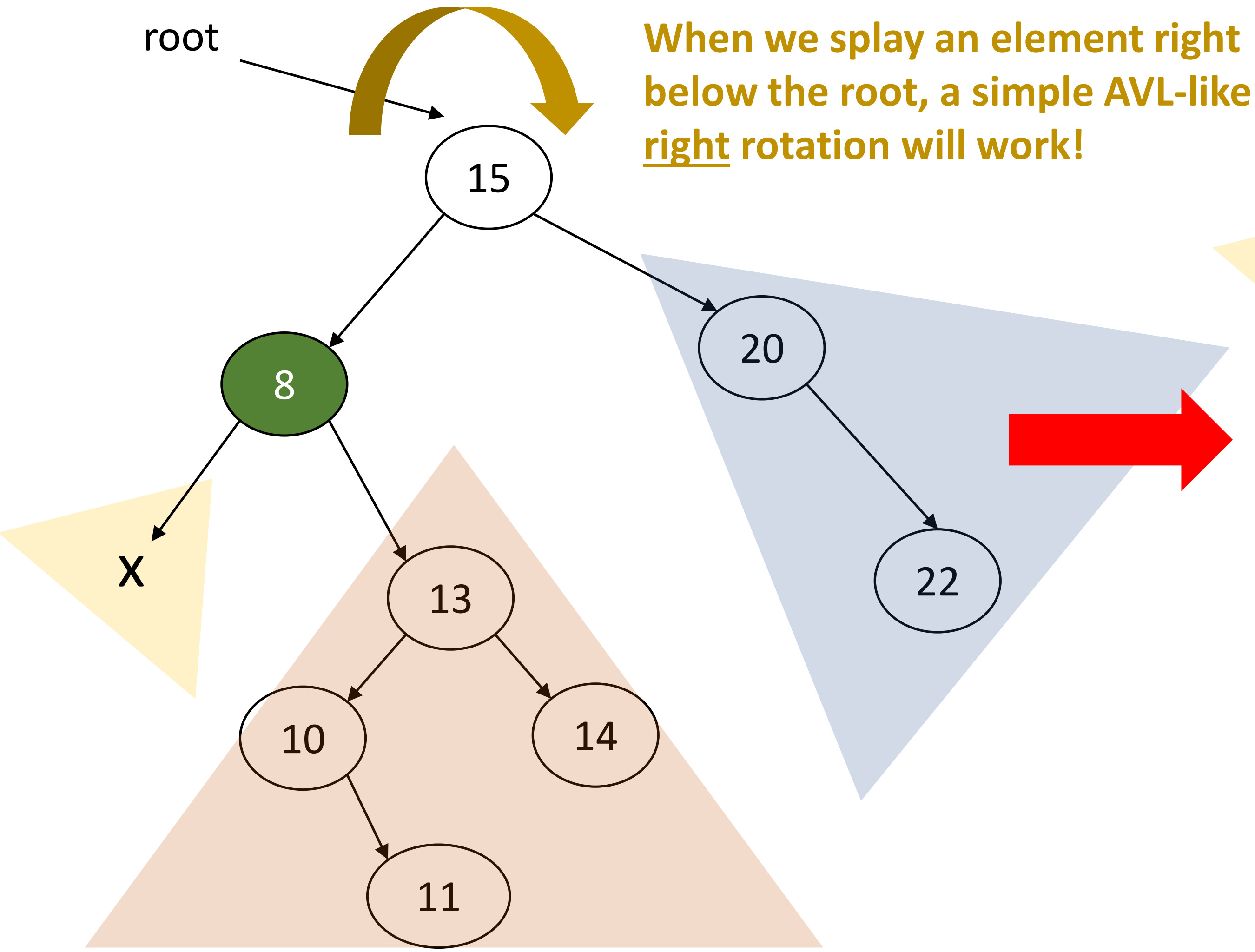
`splay(8, root)`

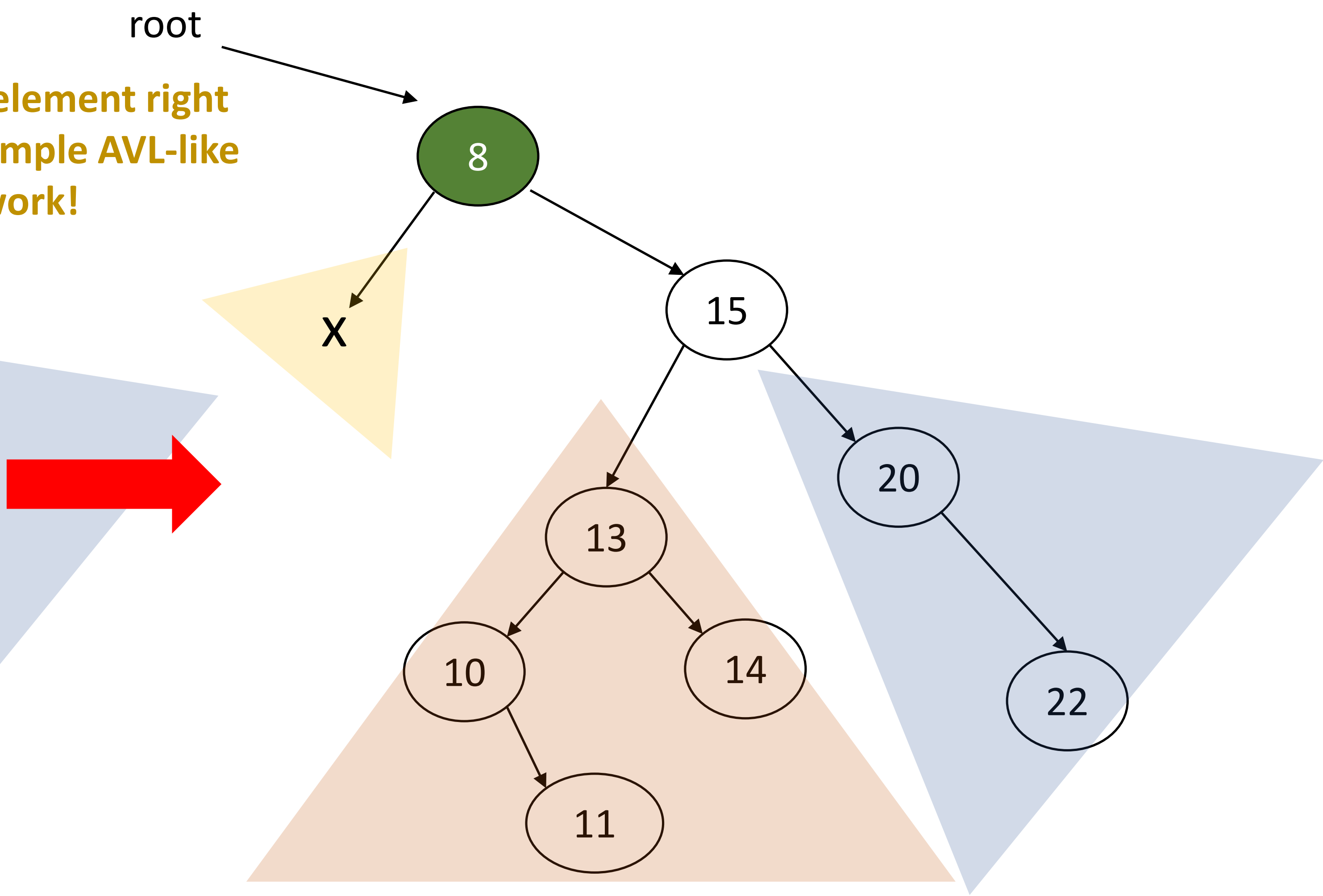


Splaying example

`splay(8, root)`

When we splay an element right below the root, a simple AVL-like <u>right</u> rotation will work!

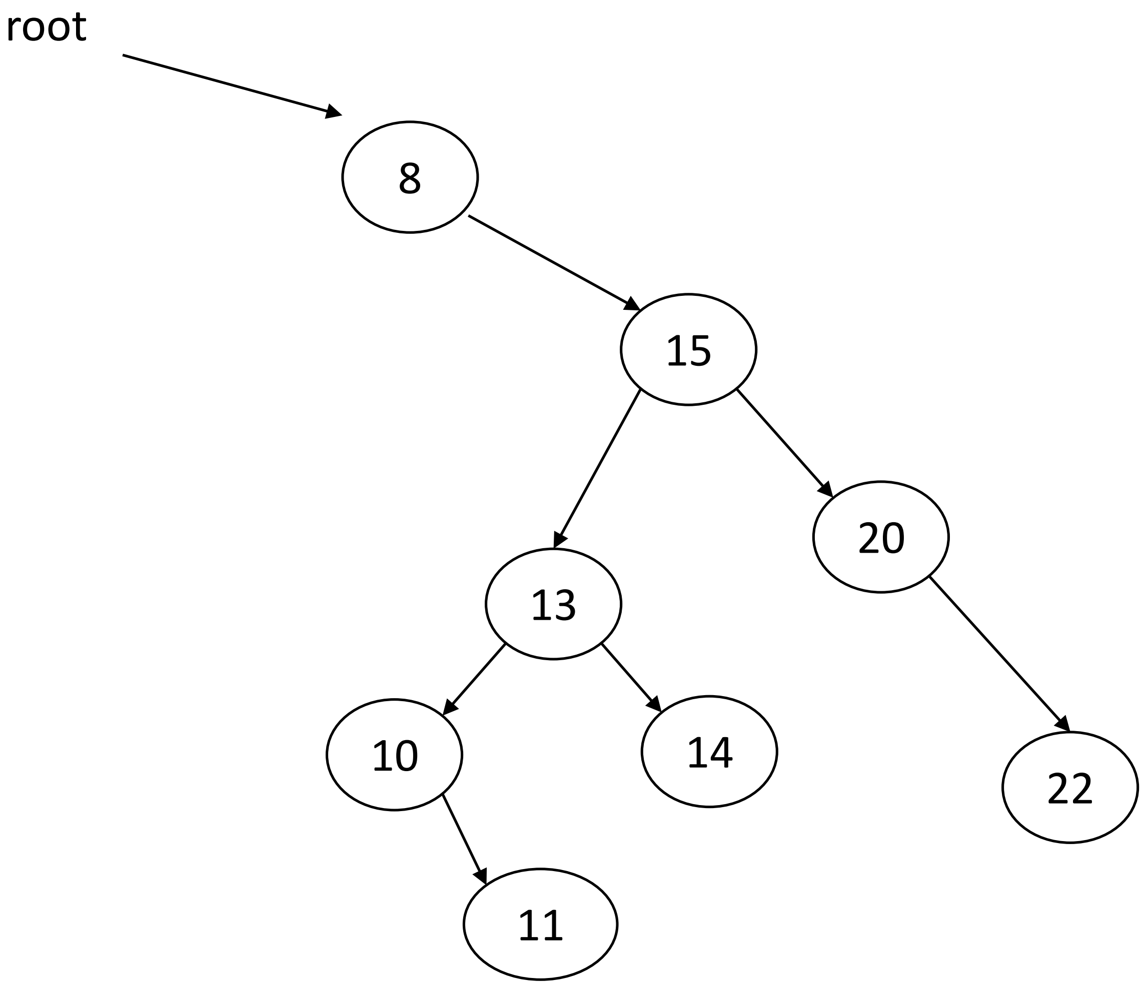




Splaying example

`splay(12, root)`

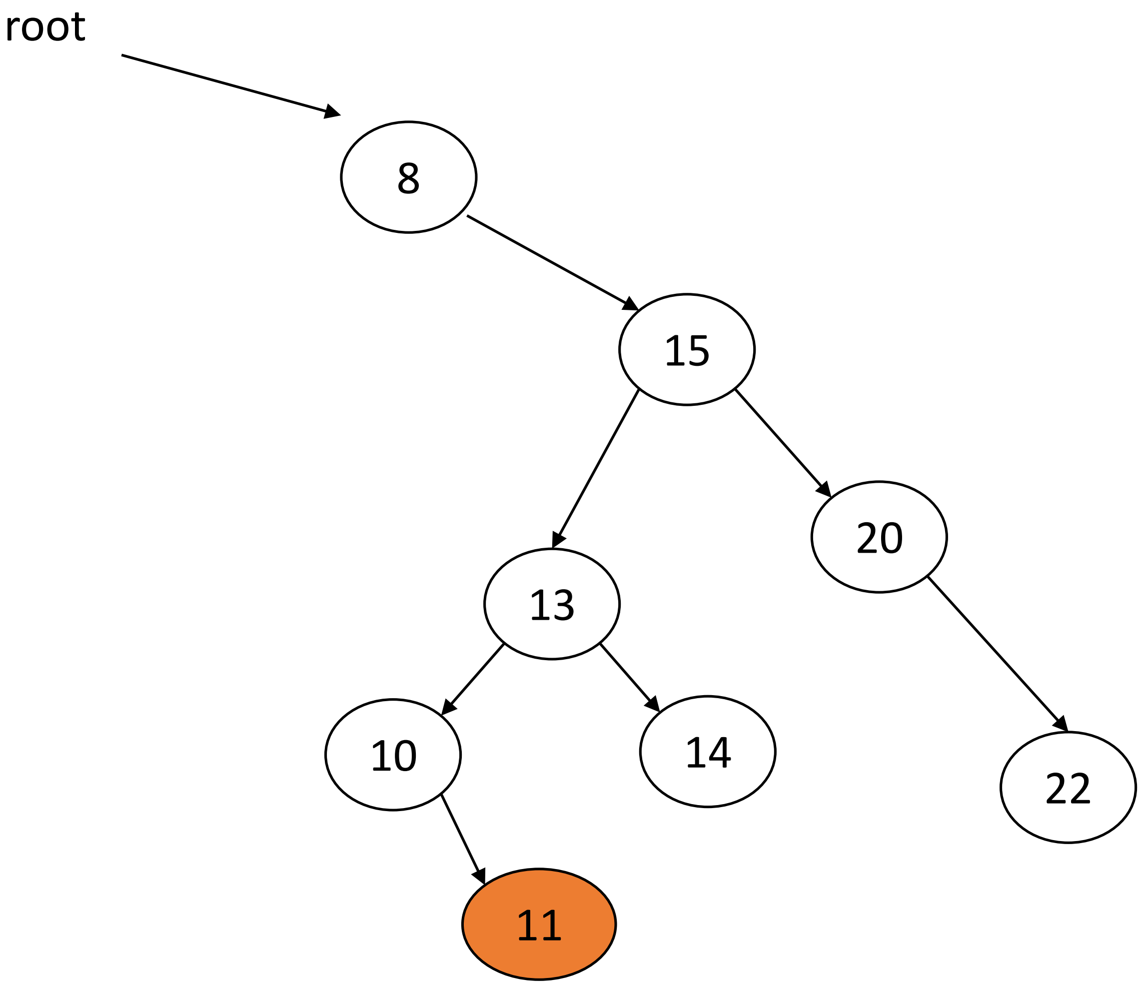
- **Note that 12 is not in the tree!**



Splaying example

`splay(12, root)`

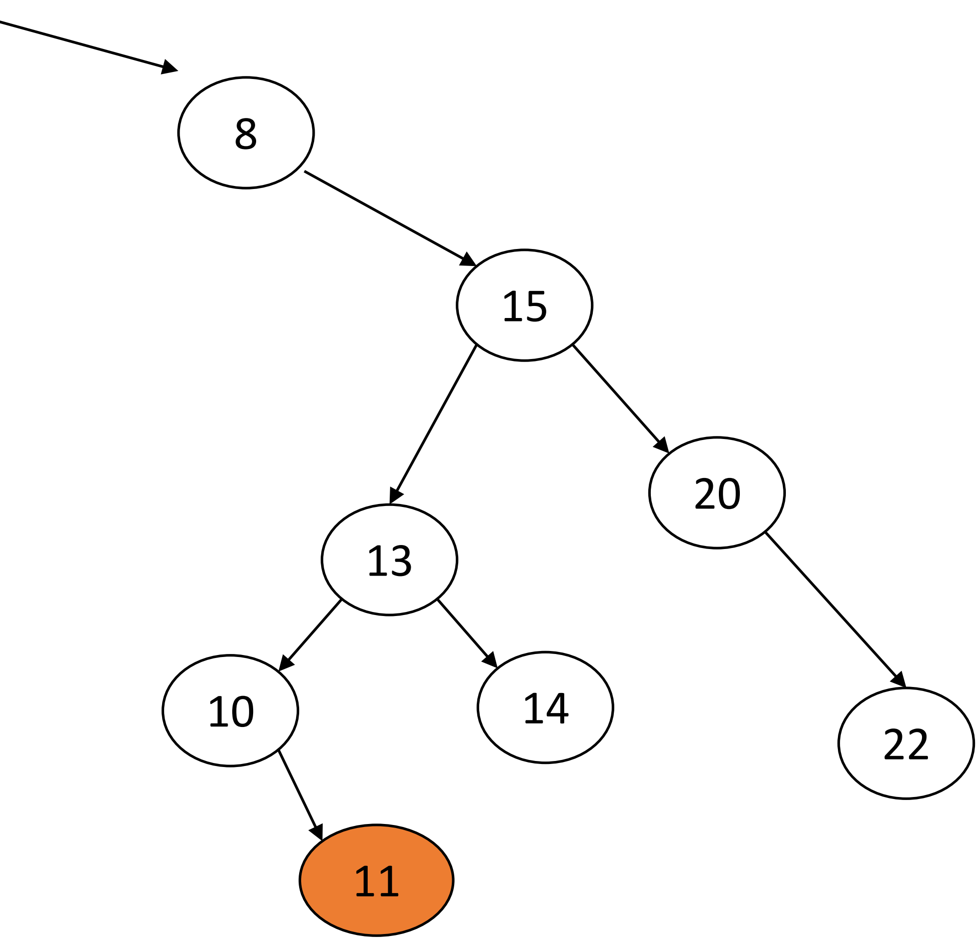
- **Note that 12 is not in the tree!**
- The descending part of the splaying routine will reach 11.



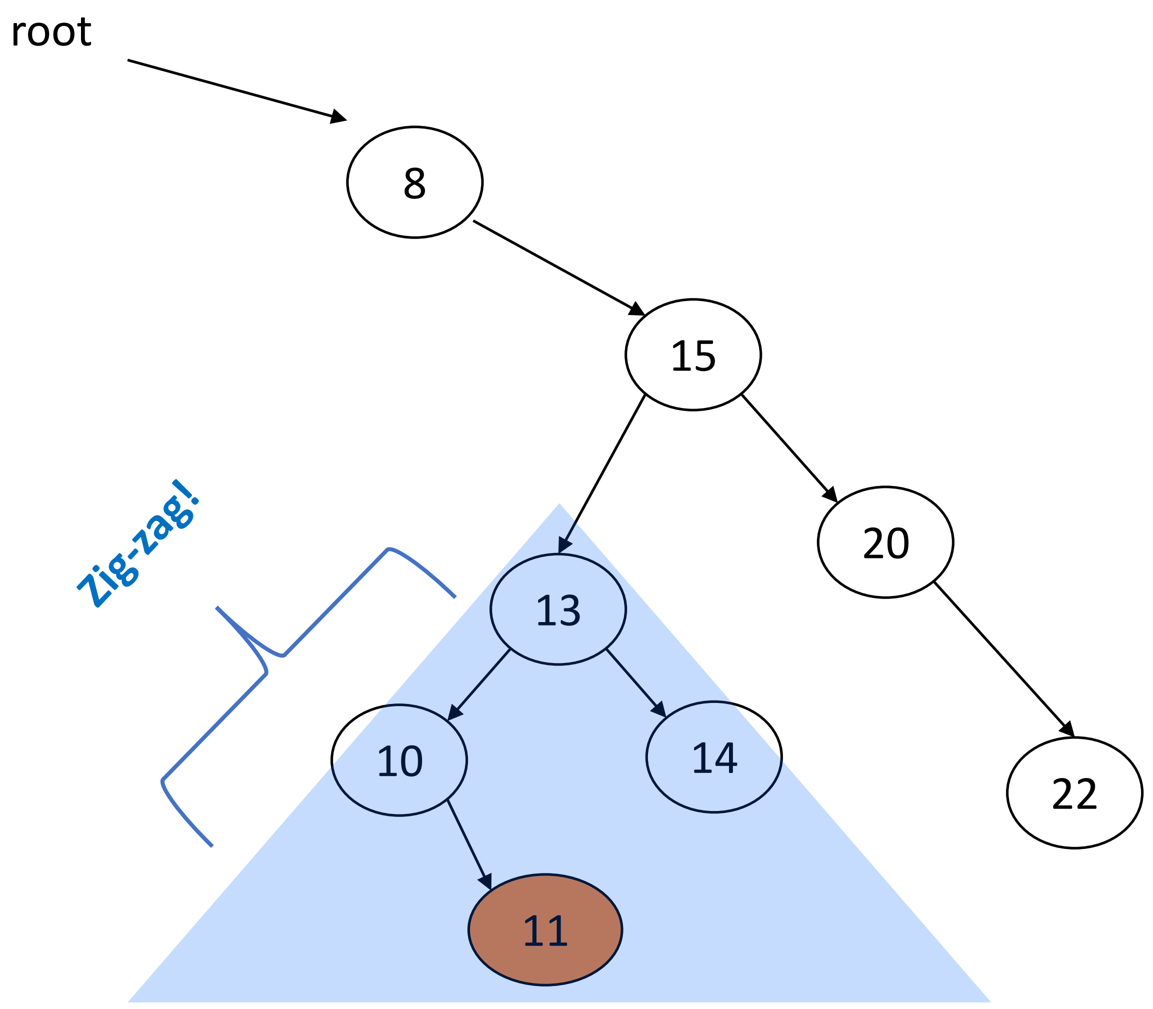
Splaying example

`splay(12, root)`

- **Note that 12 is not in the tree!**
- The descending part of the splaying routine will reach 11.
- **Ascending part (steroids)**: Bring 11 to the root!



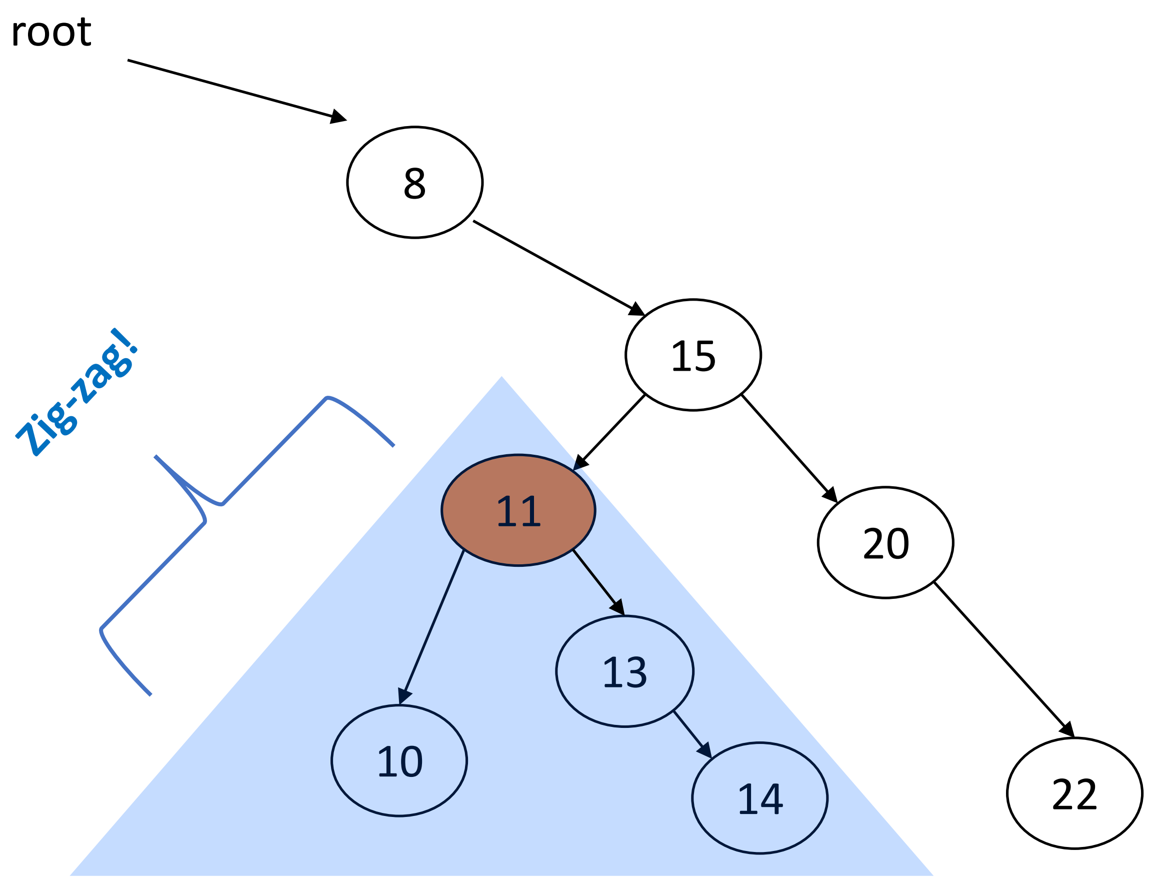
Bringing 11 to the root…



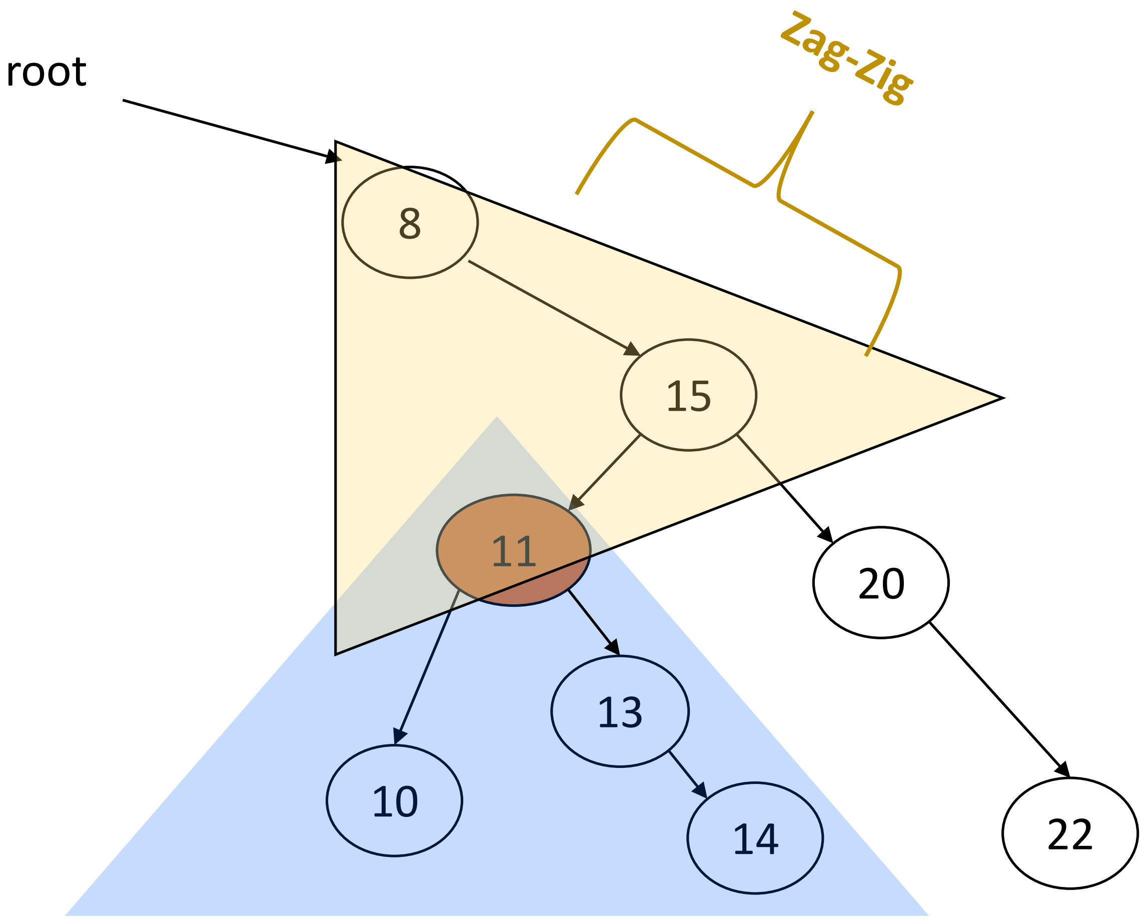
AVL Terminology	Splay Terminology
L	“ZIG”
R	“ZAG”
LR	“ZIGZAG”
RL	“ZAGZIG”
LL *(never used in AVL Trees)*	“ZIGZIG”
RR *(never used in AVL Trees)*	“ZAGZAG”

Bringing 11 to the root…

AVL Terminology	Splay Terminology
L	"ZIG"
R	"ZAG"
LR	"ZIGZAG"
RL	"ZAGZIG"
LL *(never used in AVL Trees)*	"ZIGZIG"
RR *(never used in AVL Trees)*	"ZAGZAG"

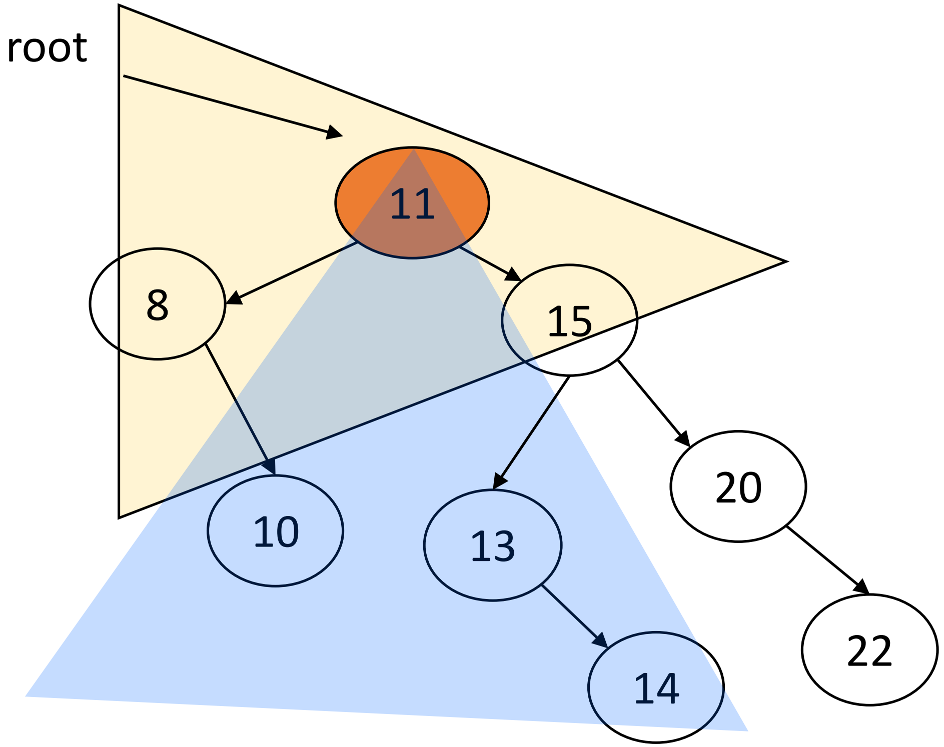


Bringing 11 to the root...



AVL Terminology	Splay Terminology
L	"ZIG"
R	"ZAG"
LR	"ZIGZAG"
RL	"ZAGZIG"
LL *(never used in AVL Trees)*	"ZIGZIG"
RR *(never used in AVL Trees)*	"ZAGZAG"

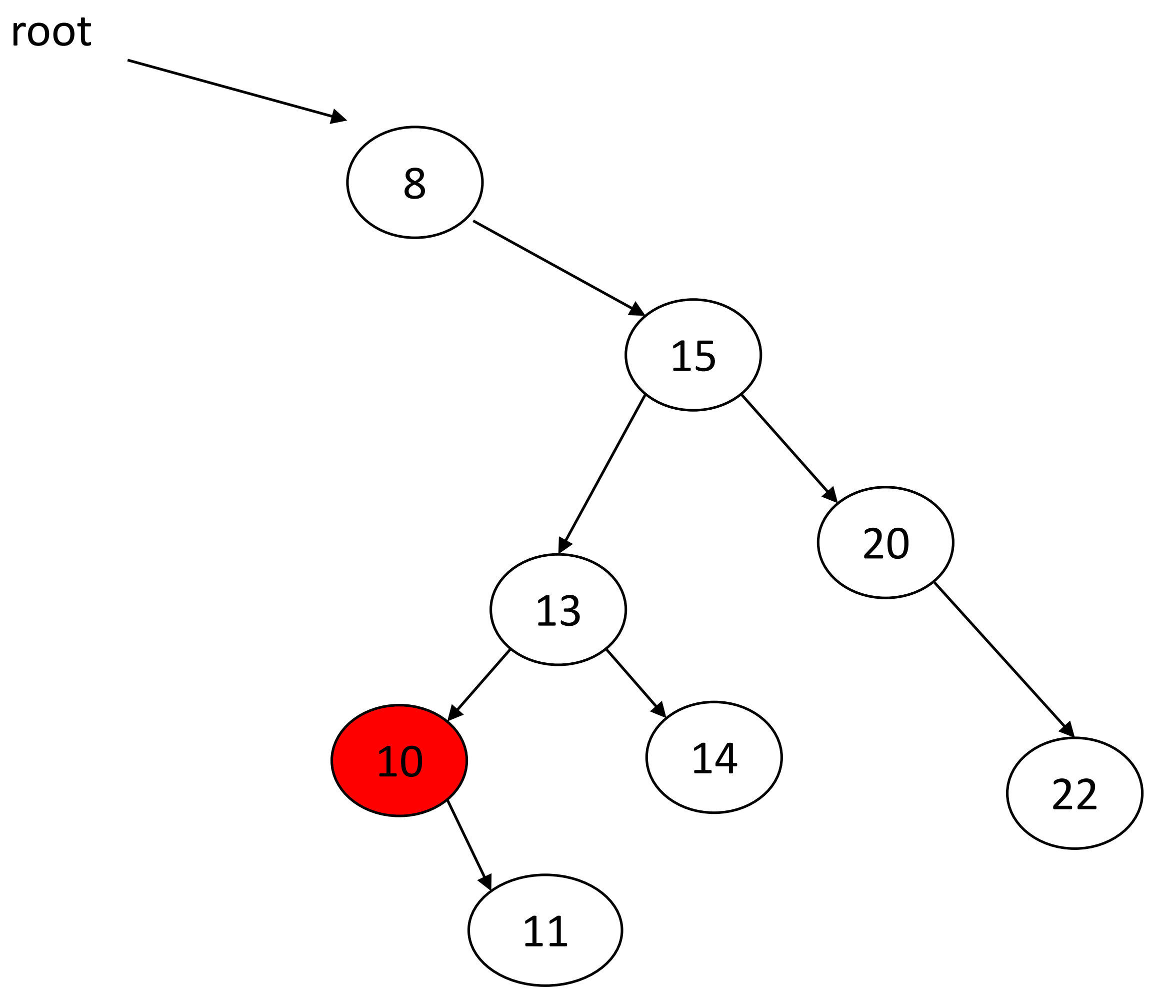
Bringing 11 to the root…



Notice how 10, 13 and 14, all close neighbors of 11 before we splayed it, are now one level closer to the root?

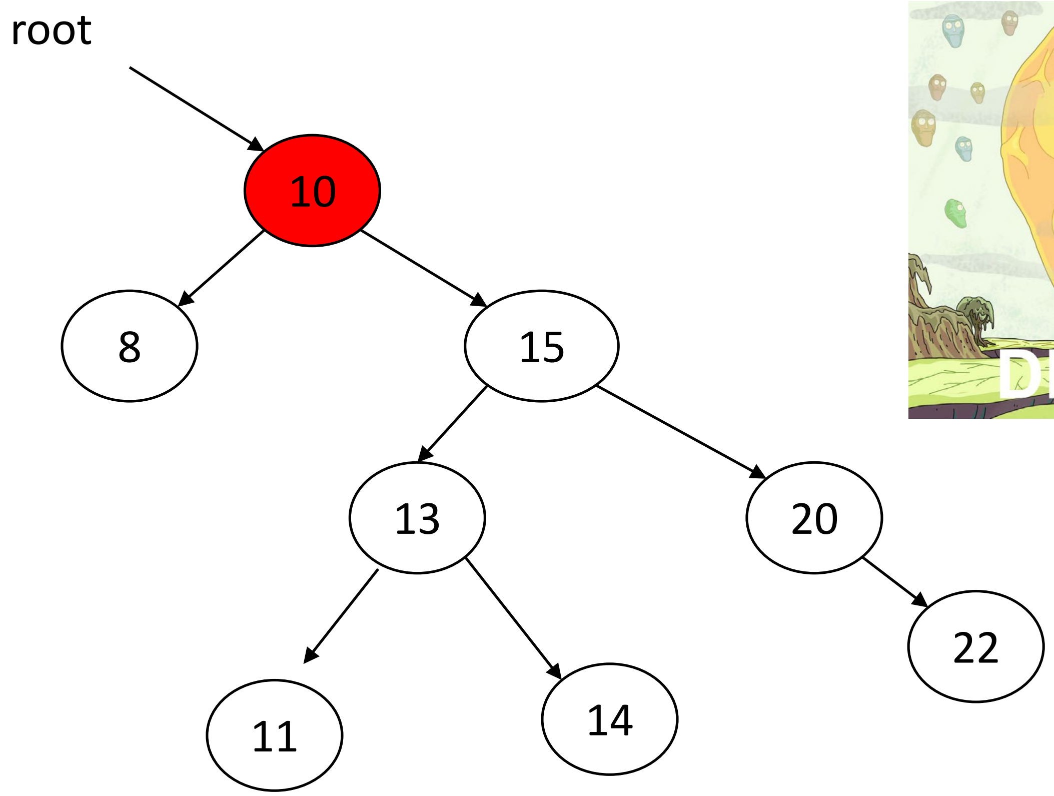
Splaying example #2

```
splay(10, root)
```



Splaying example #2

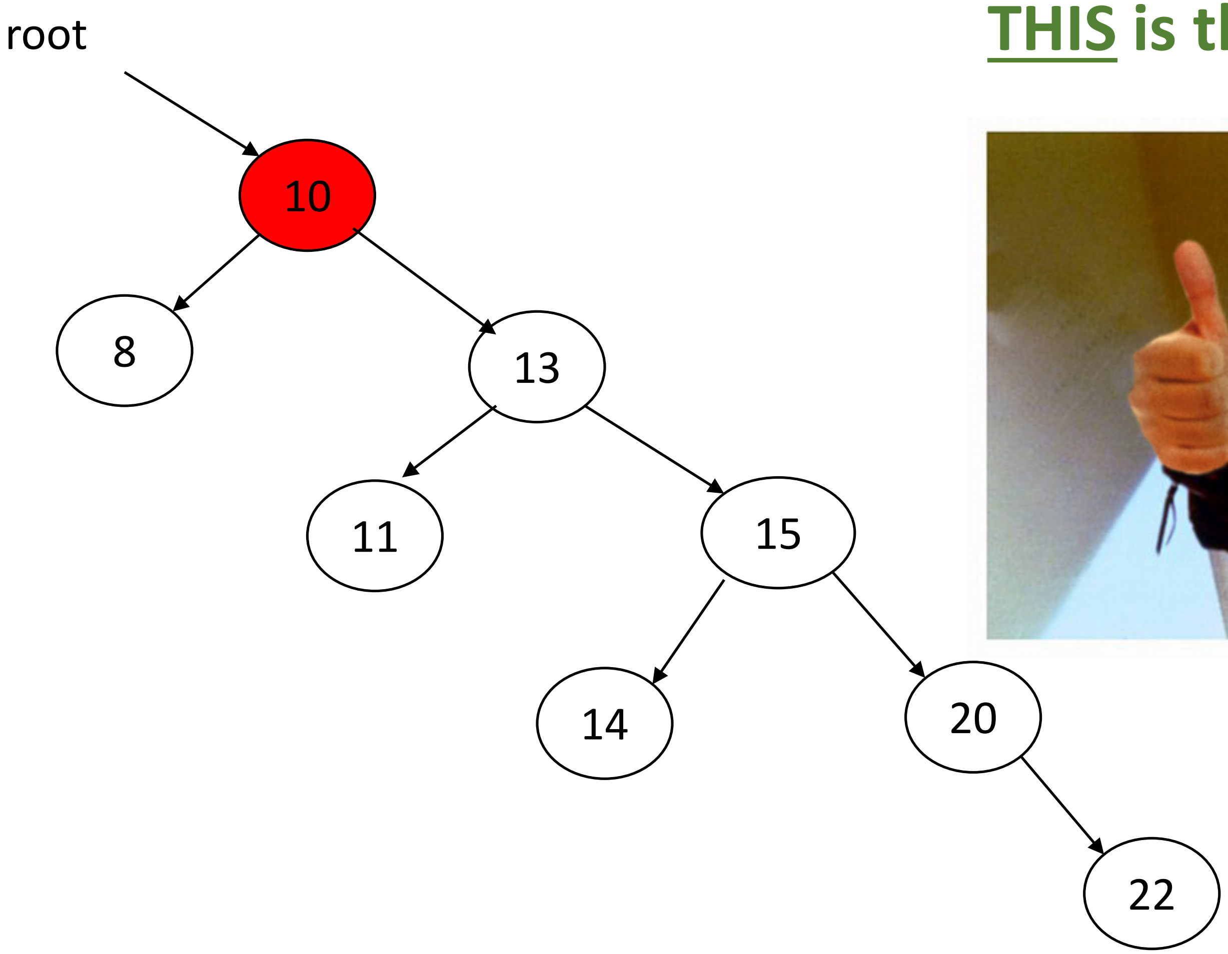
`splay(10, root)`



ALERT: If you came up with this tree, instead of ZIGZIG / ZIG-ZAG / ZAG- ZIG / ZAG – ZAG rotations, you performed AVL-Style rotations!

Thought experiment

`splay(10, root)`



<u>THIS</u> is the correct tree!

Searching a splay tree

- Assuming `splay(Key key, Node node)` implemented, develop a routine `search(Key key)` now, in your notes!
- Your search method should return `null` if the key is not there and the key itself if it is.
- Assume your splay tree class begins like this:

Searching a splay tree

- Assuming `splay(Key key, Node node)` implemented, develop a routine `search(Key key)` now, in your notes!
- Your search method should return `null` if the key is not there and the key itself if it is.
- Assume your splay tree class begins like this:

```
public class SplayTree<Key extends Comparable<Key>> {
        private class Node {
                Key key;
                Node left, right;
        }
        private Node root;
        private Node splay(Key key, Node node){/* This is assumed implemented */}
```

Search routine

```
public Key search(Key key){
	if(root == null)
		return null;
	root = splay(key, root);
	if(root.key.compareTo(key) == 0)
		return root.key;
	else
		return null;
}
```

Search routine

```
public Key search(Key key){
	if(root == null)
		return null;
	root = splay(key, root);
	if(root.key.compareTo(key) == 0)
		return root.key;
	else
		return null;
}
```

search() is a breeze given splay() ☺

Insertion in a splay tree

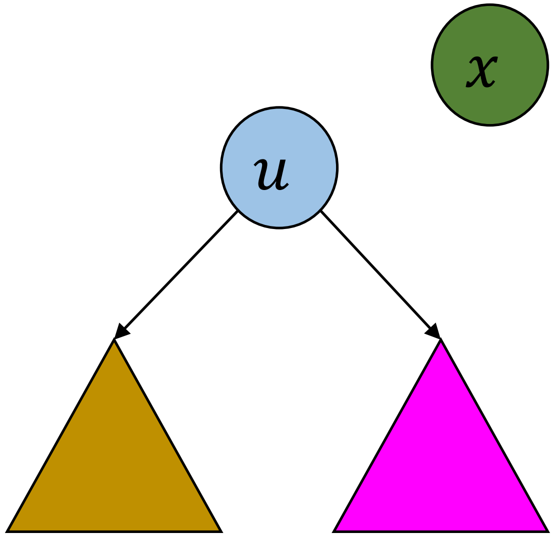
- Let the element to be inserted be called x.
- By temporal locality, we expect to need to search for or delete x soon.
- So we would like fast access to x, i.e for it to be at the root after insertion.
- For insertions, it doesn't matter whether you insert first and splay second or vice versa.
  - The introductory paper (Sleator and Tarjan 1985) **splays first, and so will we.**

Insertion

- Suppose that we don't allow duplicates in our splay tree
  - Consistent with using the structure as a **dictionary (database of <K, V> pairs)!** ☺
- Then, here's how insertion will work:
  1. Splay the root node with the key to be inserted.
  2. Compare new root's key with key to be inserted.
     - If they're equal, do nothing, since we don't allow duplicates.
     - If the new root's key is the biggest key in the set smaller than our new key *(inorder predecessor),* then name this case #1.
     - If the new root's key is the smallest key in the set bigger than our new key (*inorder successor),* then name this case #2.

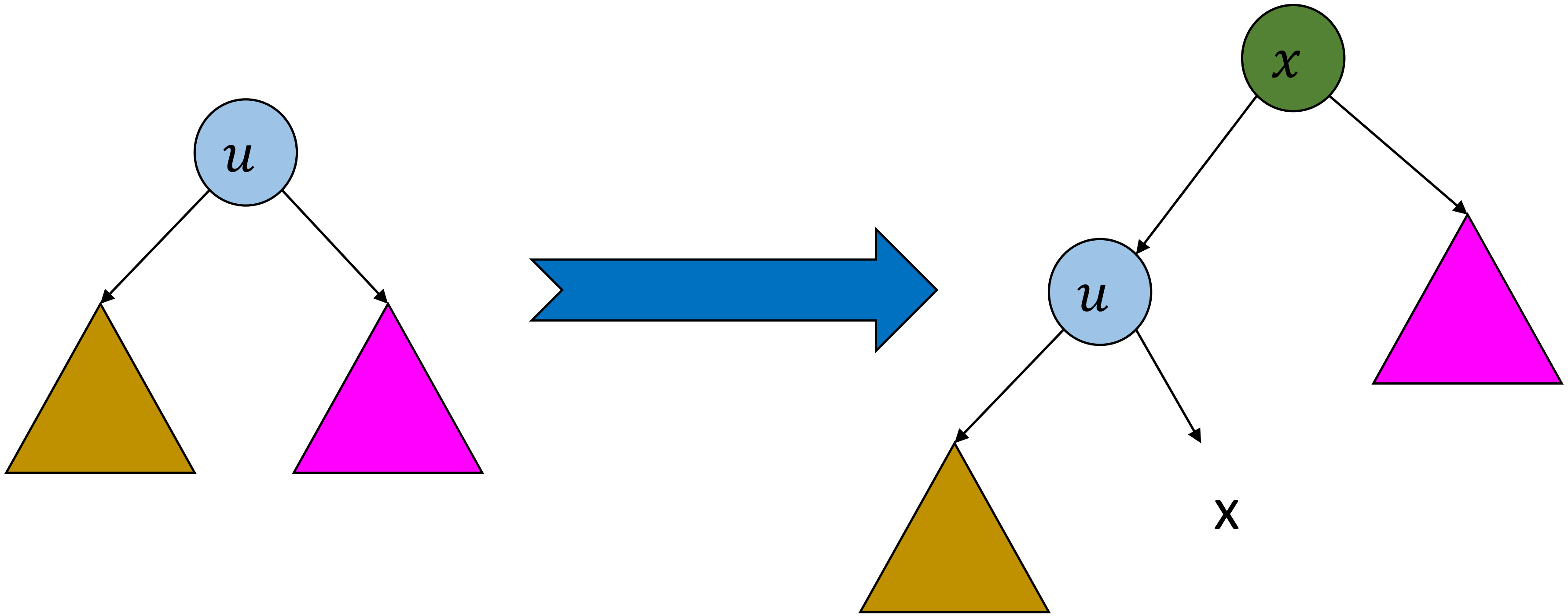
Case #1: Inorder Predecessor

- We wanted to insert key x, so first we splay the root with key x.
- Let the node of the new root be called u. We assume $u < x$, which by the nature of the splaying operation means that u is x's inorder predecessor.



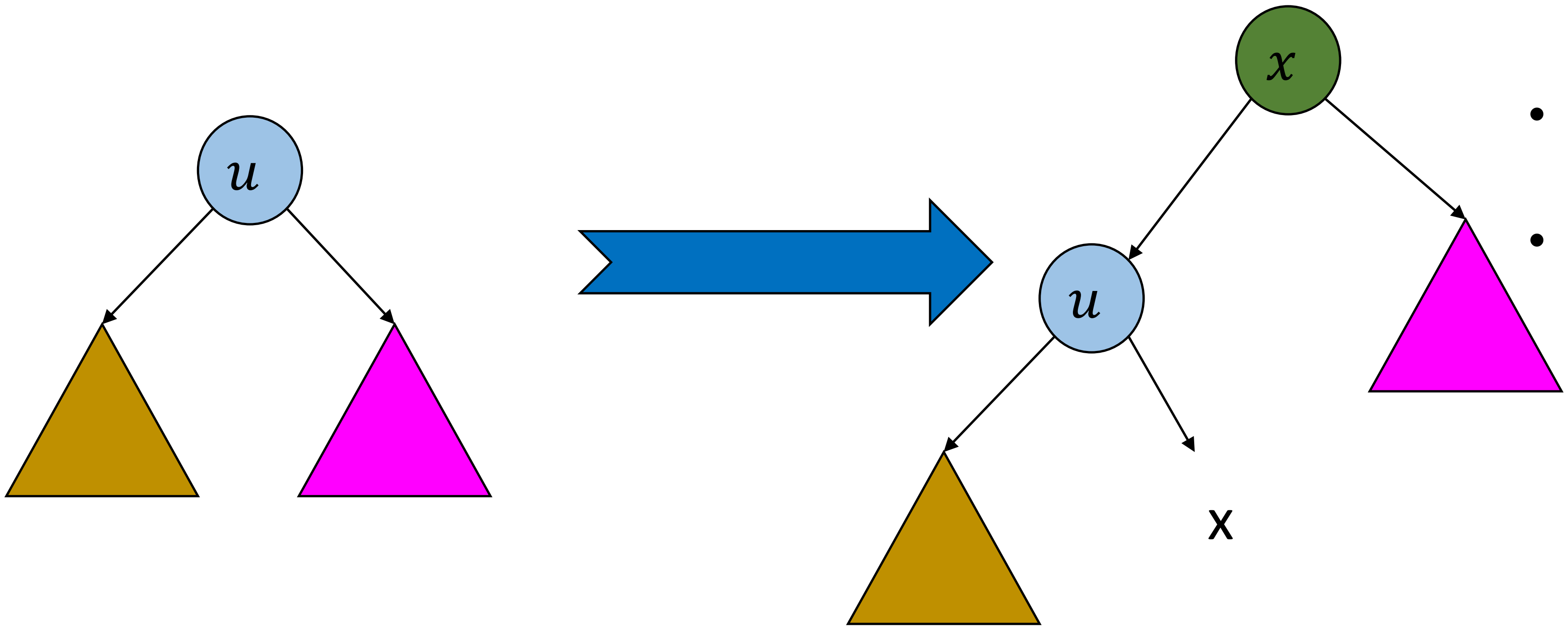
Case #1: Inorder Predecessor

- We wanted to insert key x, so first we splay the root with key x.
- Let the node of the new root be called u. We assume $u < x$, which by the nature of the splaying operation means that u is x's inorder predecessor.



Case #1: New root's key is Inorder Predecessor

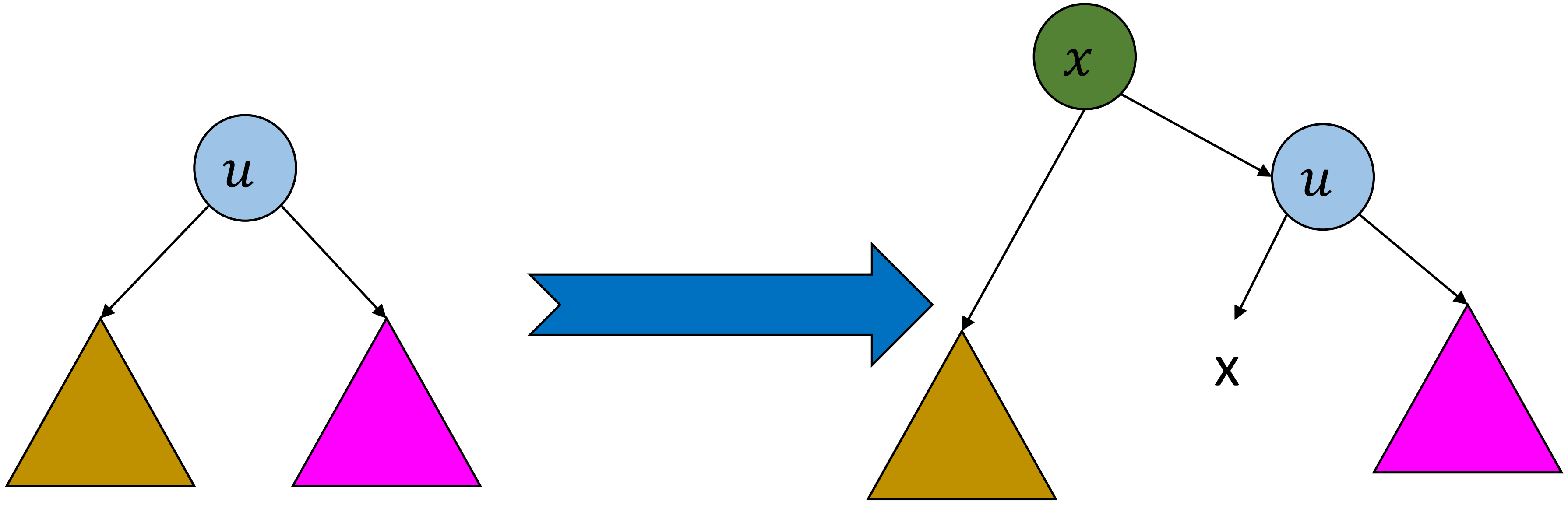
- We wanted to insert key x, so first we splay the root with key x.
- Let the node of the new root be called u. We assume $u < x$, which by the nature of the splaying operation means that u is x's inorder predecessor.



- We satisfy the BST property.
- Newly inserted key immediately available: leveraging locality!

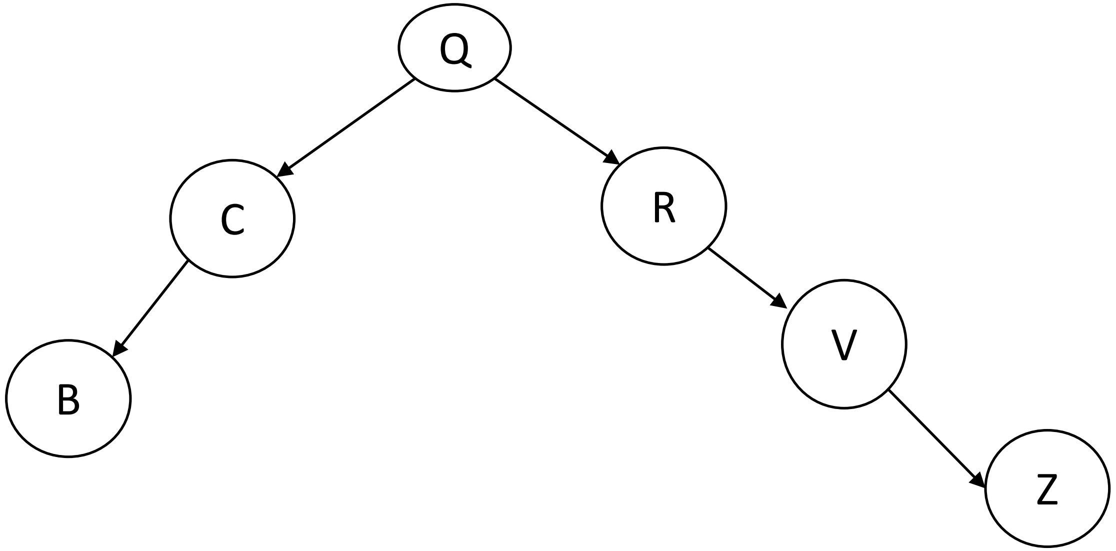
Case #2: New root's key is inorder successor

- Symmetric case ($x < u$):



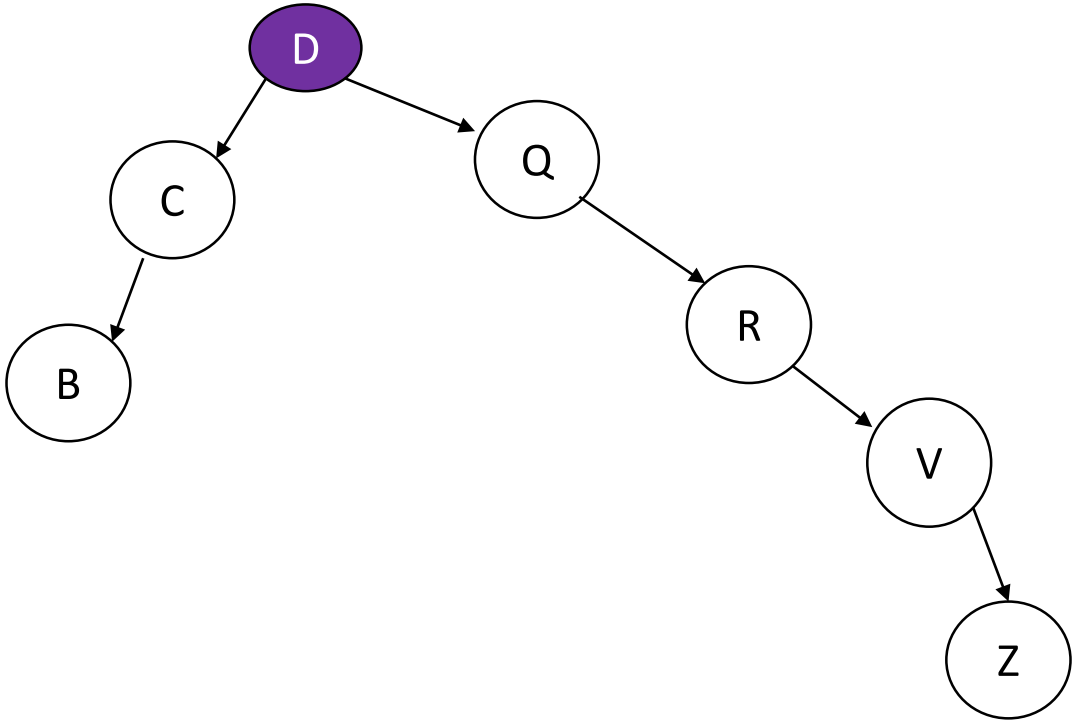
Examples

- Insert ‘D’ in the following splay tree.



Examples

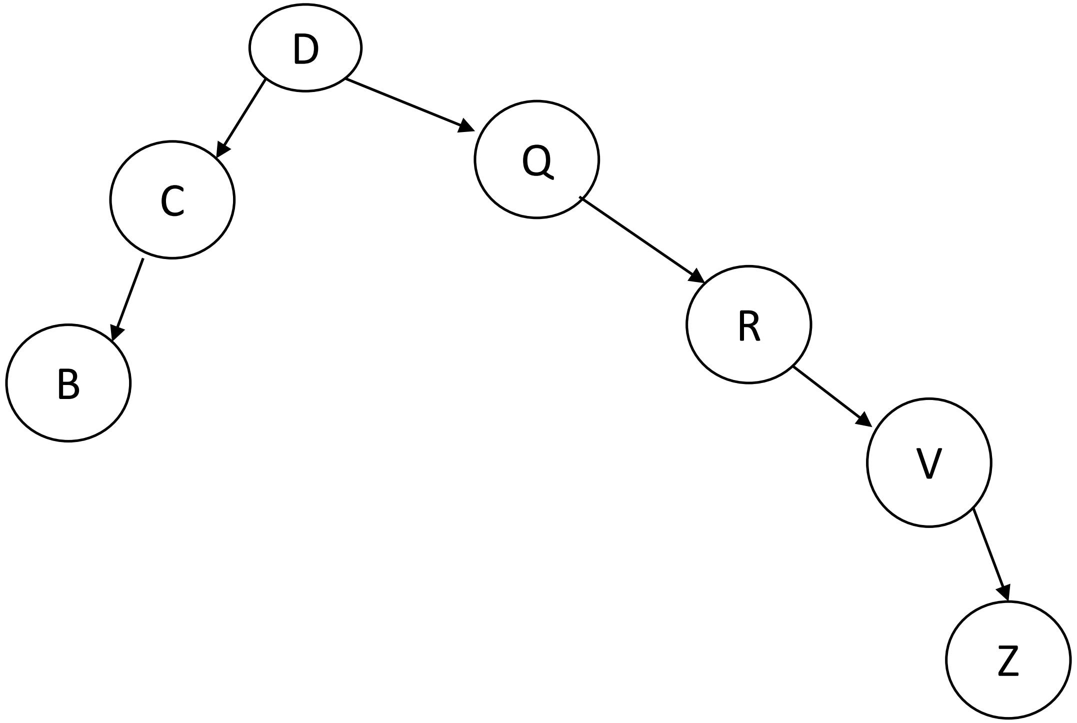
- Insert ‘D’ in the following splay tree.



1. D is not in the tree.
2. Last node visited is ‘C’
3. Splay the root with key ‘C’
4. Insert ‘D’ as a new root that is the inorder **successor** of ‘C’

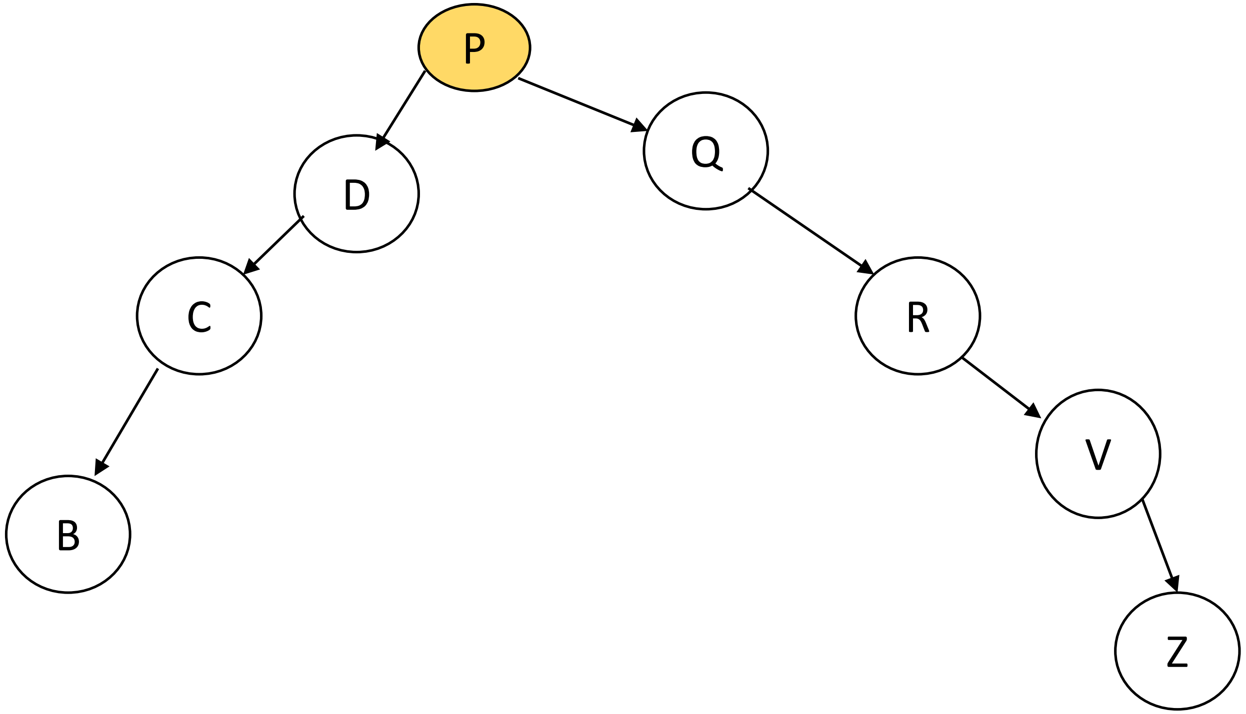
Practice

- Now go ahead and insert ‘P’.



Practice

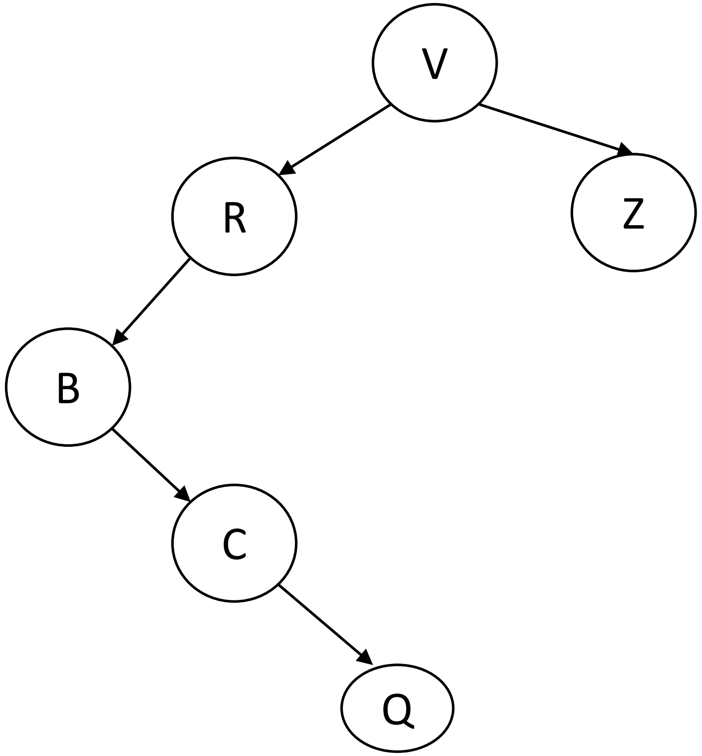
- Now go ahead and insert 'P'.



1. P is not in the tree.
2. Last node visited is 'Q'
3. Splay the root with key 'P'
4. Insert 'P' as a new root that is the inorder **predecessor** of 'Q'

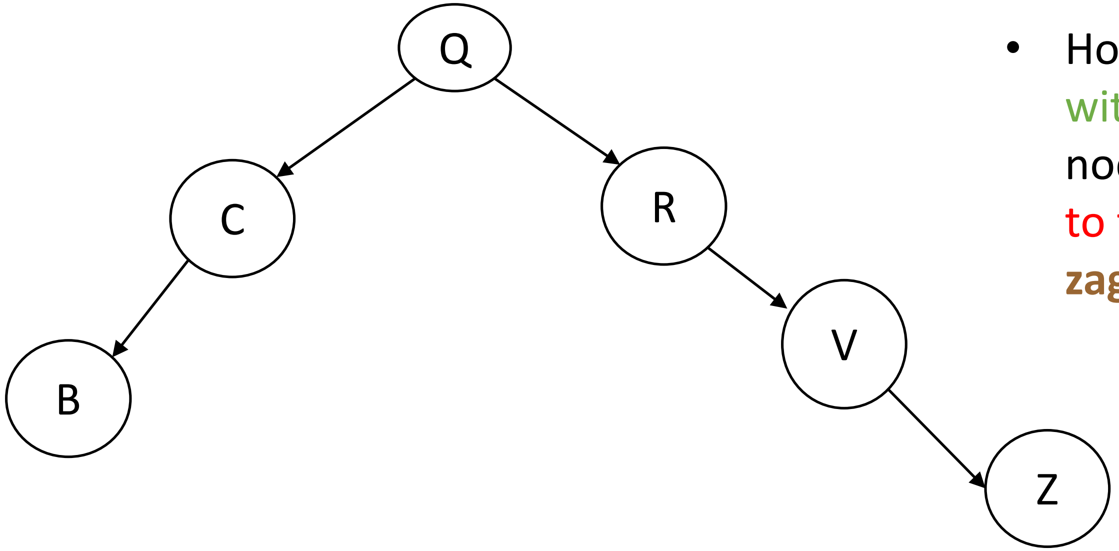
Examples

- Insert ‘Q’ in the following splay tree:



Examples

- Insert ‘Q’ in the following splay tree:



- Q is already in the tree, so no actual insertion will take place!
- However, we will splay the root with key ‘Q’, causing the leaf node that contains ‘Q’ to ascend to the root (**1 zig-zig and 1 zag-zag**)

Deletion

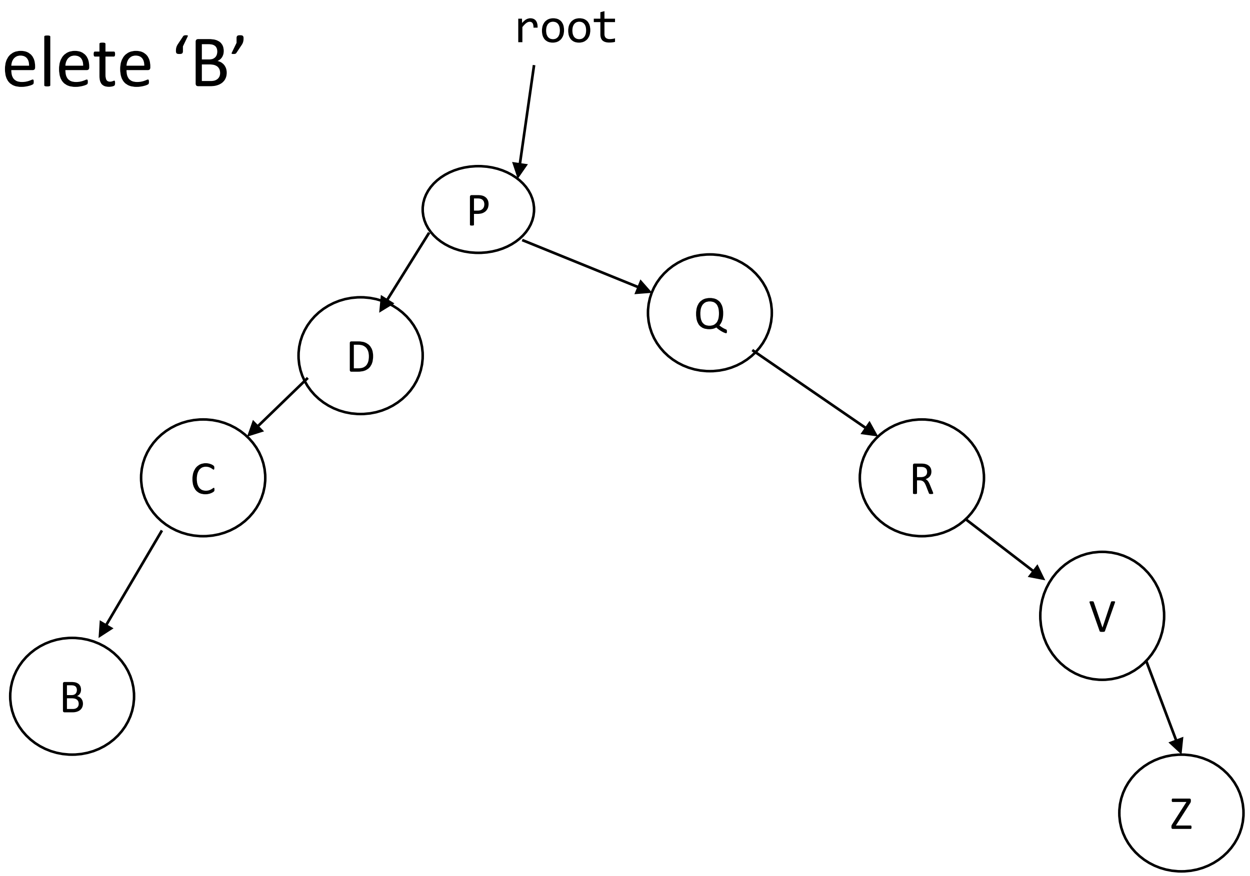
- Deletion is similar to addition
- Splay trees have, the easiest (and most efficient) deletion process among **all** dynamic binary trees that we will discuss!

Deletion

- Process (roughly)
  1. Splay the root with the key to be deleted
  2. If the root doesn't contain the key to be deleted, nothing to do. We did, however, significantly improve access to an inorder predecessor or successor.
  3. If the root **does** actually contain the key, split between cases:
     a) (Easy case): If the left child is null, we splayed the root with the minimum key in the tree. **Just replace the root with its right child.**
     b) (Hard case): If not, splay the *(non-null)* left child of the root ***with the key to be deleted once again***! This will bring the key's inorder predecessor as the left child of the old root, and we can then replace the root with that particular child!

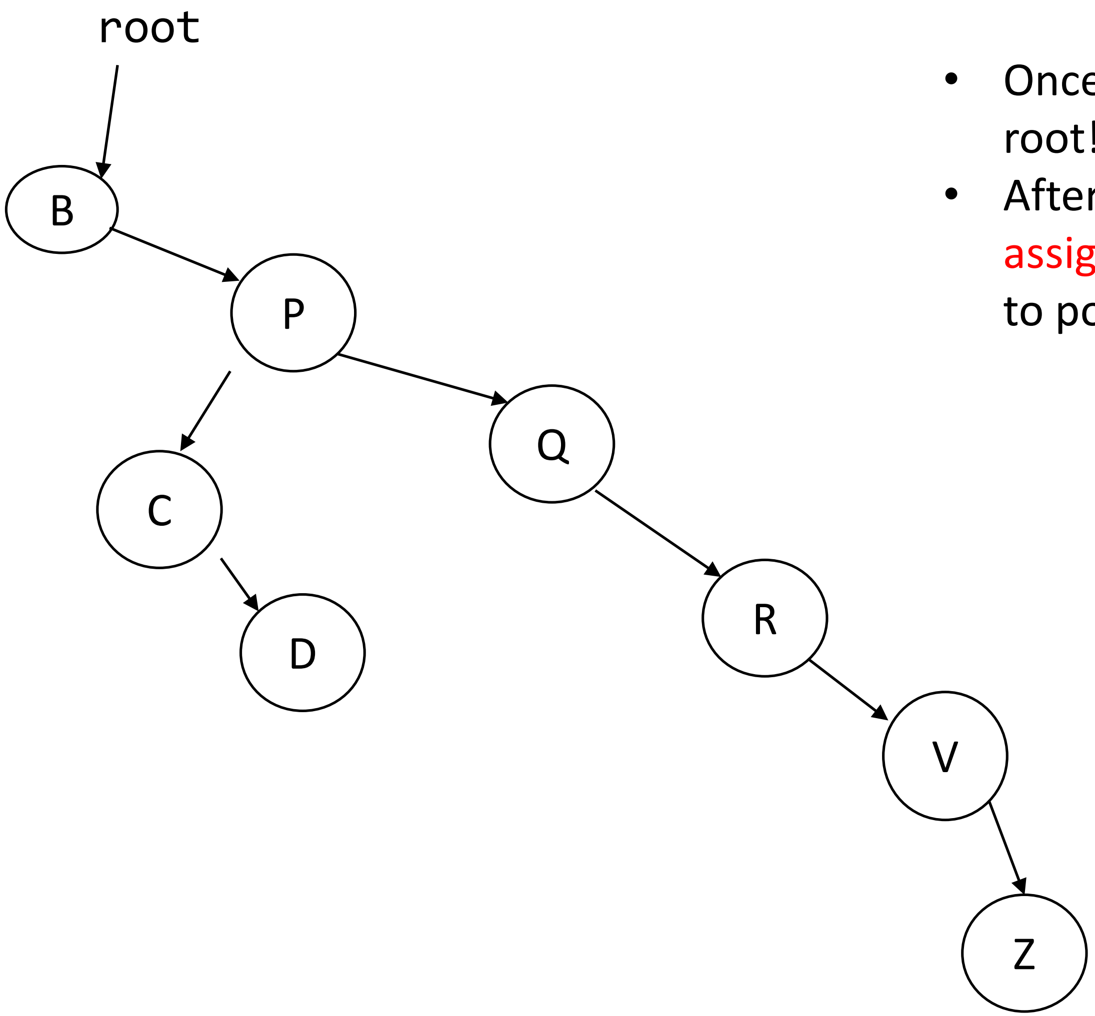
Deletion examples

- Let’s delete ‘B’



Deletion examples

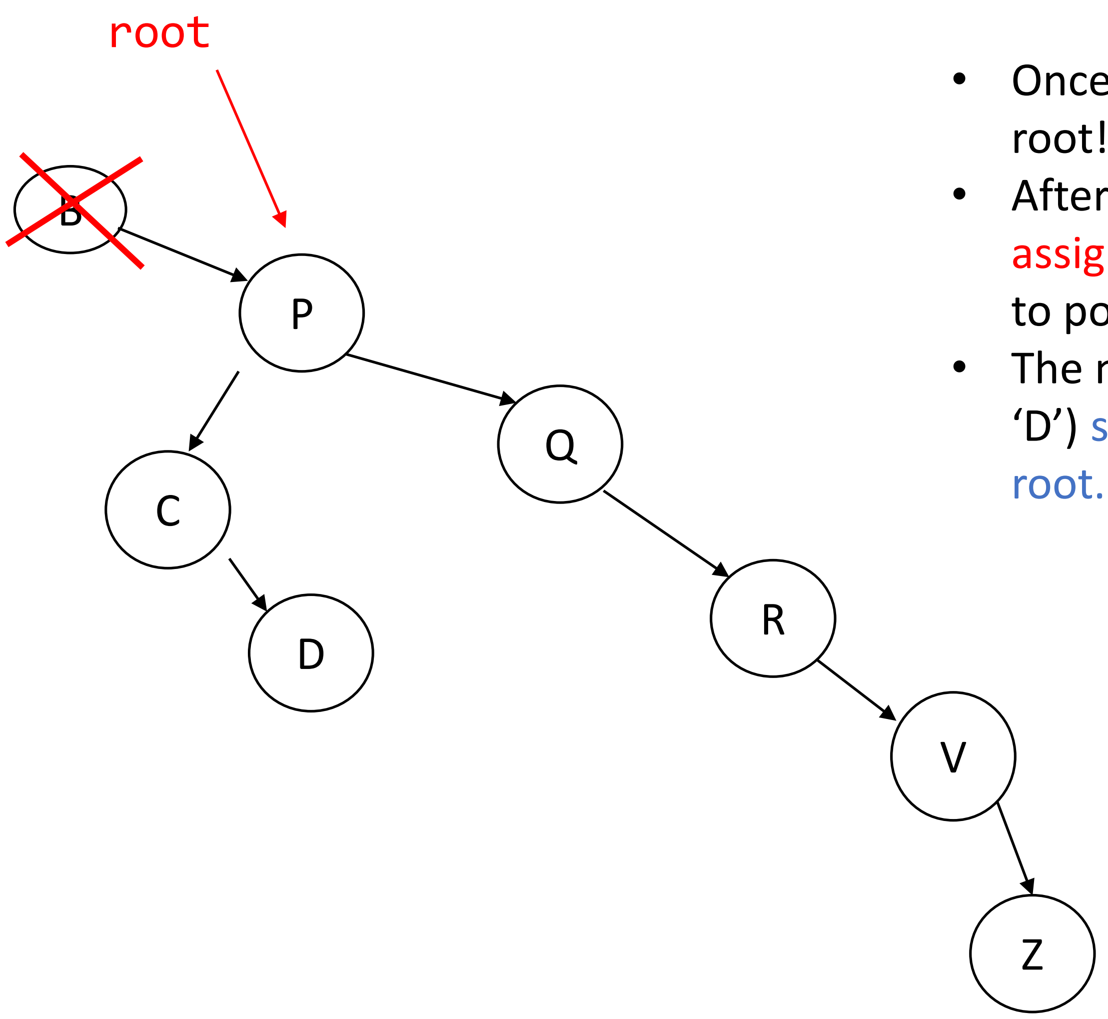
- Let’s delete ‘B’



- Once splayed, ‘B’ will be at the root!
- After that, it’s a simple pointer assignment to change the root to point to it.

Deletion examples

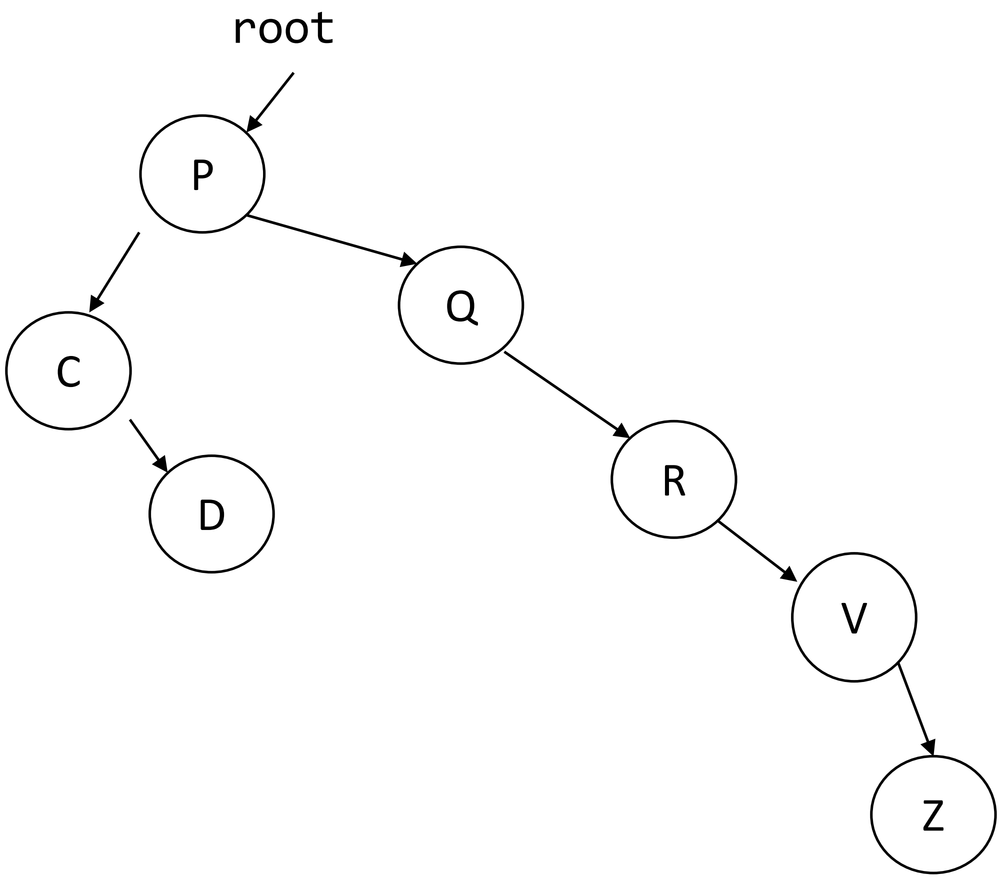
- Let’s delete ‘B’



- Once splayed, ‘B’ will be at the root!
- After that, it’s a simple pointer assignment to change the root to point to its right child, ‘P’.
- The neighborhood of ‘B’ (‘C’, ‘D’) still remain close to the root.

Deletion examples

- Let’s now delete ‘R’!



Theory behind splay trees

- Let $m \geq 1$ and $o_1, o_2, o_3, \ldots, o_m$ be a sequence of m operations on a splay tree.
  - By "operations", we mean either insertions, searches or deletions! ☺
  - For example, when $m = 7$, we could have…
    - 4 insertions, 2 searches, 1 deletion
    - Or 2 deletions *(who said the tree had to be initially empty?)*, 3 insertions and 2 searches!

Theory behind splay trees

- Let also N be the ***maximum*** number of nodes the tree had during those m operations.
  - If the operations were all deletions, N would be whatever count we began with.
  - If there are 2 deletions, 2 insertions and 3 searches, same thing.
  - If there are only insertions, N is our current node count.

Theory behind splay trees

- Then, the total time for completing all of those m operations is:

$$\mathcal{O}(m \cdot \log_2 N)$$

Theory behind splay trees

- Then, the total time for completing all of those m operations is:

$$\mathcal{O}(m \cdot \log_2 N)$$

- From this we can deduce that the amortized cost of a search, insertion or deletion in a threaded tree is…

Amortized Constant	Amortized Linear	Amortized Logarithmic	Something else (what?)

Theory behind splay trees

- Then, the total time for completing all of those m operations is:

$$\mathcal{O}(m \cdot \log_2 N)$$

- From this we can deduce that the amortized cost of a search, insertion or deletion in a threaded tree is…

$$\frac{c \cdot \cancel{m} \cdot \log_2 N}{\cancel{m}} = c \cdot \log_2 N$$

$c > 0$ is the constant from the defn. of $\mathcal{O}(\log_2 N)$.

Amortized Constant	Amortized Linear	Amortized Logarithmic	Something else (what?)

Theory behind splay trees

- Then, the total time for completing all of those m operations is:

$$\mathcal{O}(m \cdot \log_2 N)$$

- From this we can deduce that the amortized cost of a search, insertion or deletion in a threaded tree is…

$$\frac{c \cdot \cancel{m} \cdot \log_2 N}{\cancel{m}} = \mathrm{c} \cdot \log_2 N$$

And here we see what we pay, on average: *the base 2 logarithm of the* ***maximum*** *amount of nodes ever present in the sequence!*

Amortized Constant	Amortized Linear	Amortized Logarithmic	Something else (what?)

Take-home message

Splay Trees		AVL Trees	
+	-	+	-
Do not need to store balance / height info	Not guaranteed to be balanced, leading to some super-logarithmic operation times	Guaranteed $\mathcal{O}(\log_{\phi} n)$ performance of all operations.	Spend **a lot of time** *(almost after every operation!)* on rotations, which are themselves expensive operations
Exploit temporal and spatial locality to **often** achieve **sub-logarithmic** operation time	Cannot easily adjust existing BST routines into Splay Trees; have to re-write them.	Easy to add balancing functionality to existing BST code (even by overriding)	Need to store at least 4 bits per node to preserve balance information (plus the unit cost of updating it)
Easy, clean implementation	**Expensive searches**	… … …	Deletions somewhat complex
Efficient Deletions	… … …		

… … …