Splay Trees

Peter Chapin

CIS-3012, C++ Programming

Vermont State University

Binary Search Trees

- A binary search tree (BST) is a tree data structure where:
  - Each node contains a data item (of type T)
  - Each node has at most two children (hence, “binary”)
  - There is an *ordering relation which is a strict weak ordering over T*. This relation defines what it means for one item to “come before” another in the desired ordering
  - The data item in the left child “comes before” the data item in the parent node, and the data item in the parent node “comes before” the data item in the right child

Strict Weak Whatnow?

- A **strict weak ordering** has the following properties where `a`, `b` and `c` are values of type T
  - `compare(a, a)` is always false for every `a` (*irreflexivity*)
  - If `compare(a, b)` is true, then `compare(b, a)` is false (*asymmetry*)
  - If `compare(a, b)` is true, and `compare(b, c)` is true, then `compare(a, c)` is true (*transitivity*)
  - It is *weak* in the sense that some pairs of elements are incomparable, meaning that both `compare(a, b)` and `compare(b, a)` are false. *In that case, we say that a and b are equivalent*
  - The equivalence relation is also transitive
  - A subset of values that are equivalent to each other form an *equivalence class*

Examples

- Consider `operator<` as applied to integers
  - It has all the properties of a SWO
  - Each equivalence class contains only a single value, e.g., 4 is equivalent to only 4 and no other integer.
- Consider `operator>` as applied to integers
  - This is also an SWO (although the order is the opposite of `operator<`)
- Consider case-insensitive alphabetical order of strings
  - This is also an SWO. Equivalence classes have more than one member: (“apple”, “APPLE”, “ApPLe”, etc.)
- Consider operator<= as applied to integers. ***This is not an SWO!***

Comes Before

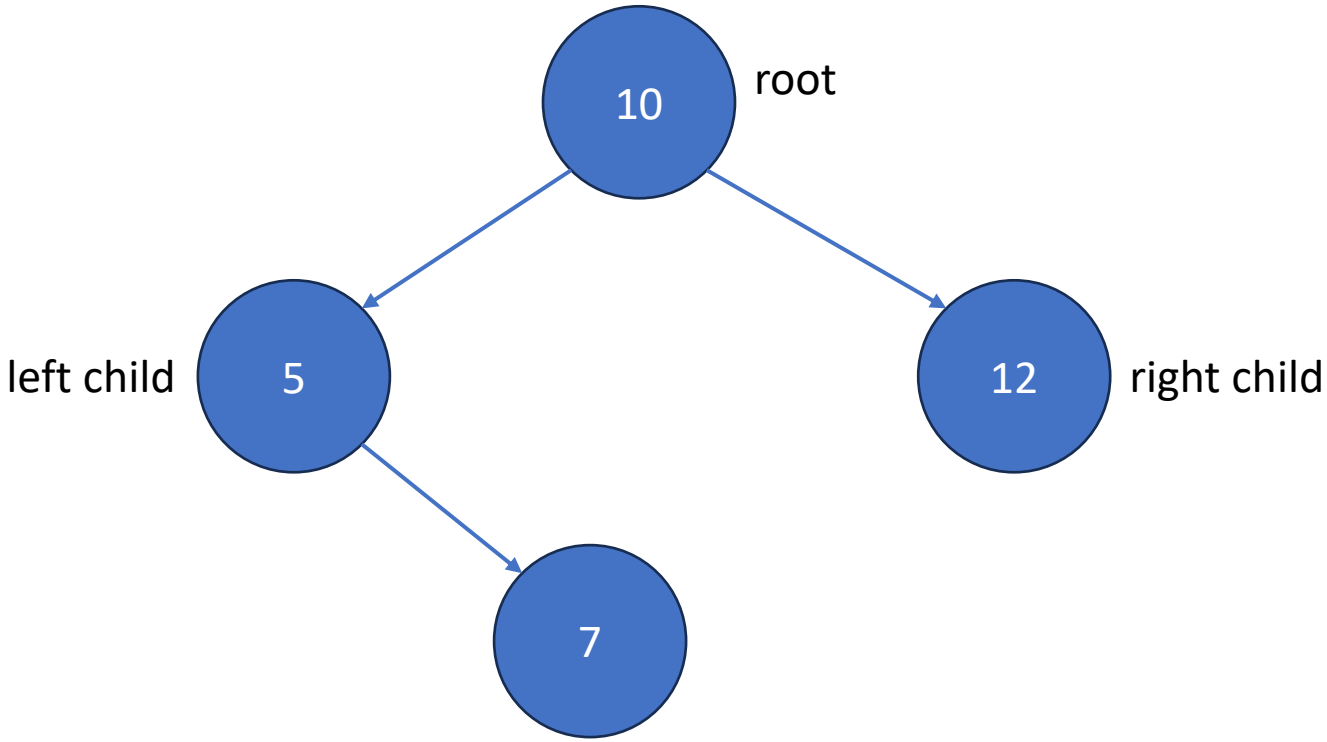
- I like to think of strict weak orderings as “comes before” relations.
  - This emphasizes that it need not be “less than” in the usual sense
  - For example:

```
bool compare_prime_factor_count( unsigned a, unsigned b )
{
    return number_of_prime_factors( a ) < number_of_prime_factors( b );
}
```

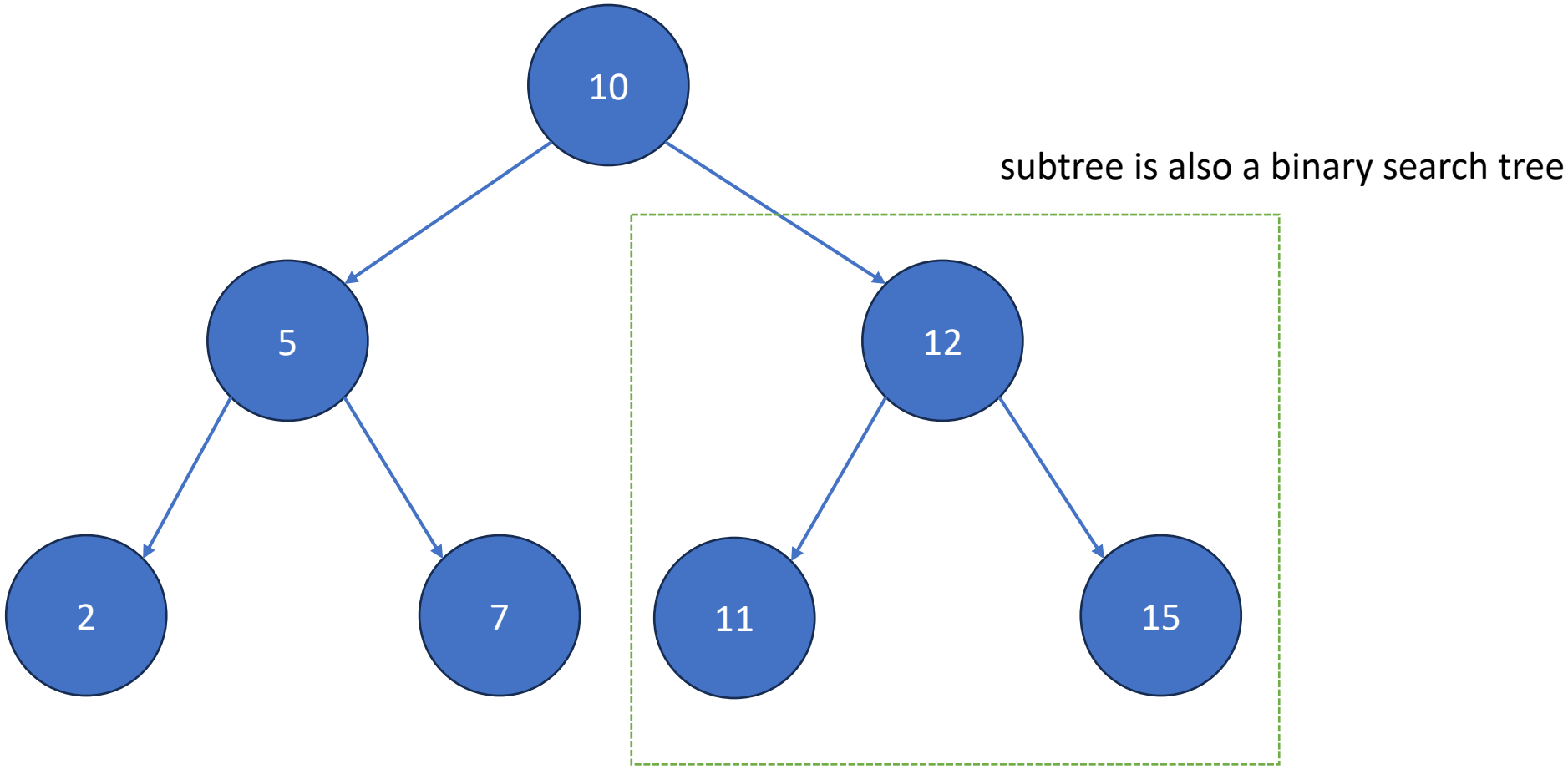
- According to this SWO…
  - 15 comes before 8, and 18 is equivalent to 30.
- Changing < to <= in the function makes this no longer an SWO.

Search Trees and SWOs

- Every binary search tree has an SWO that defines the ordering inside the tree. For example, using ordinary `operator<` on integers:



Recursive

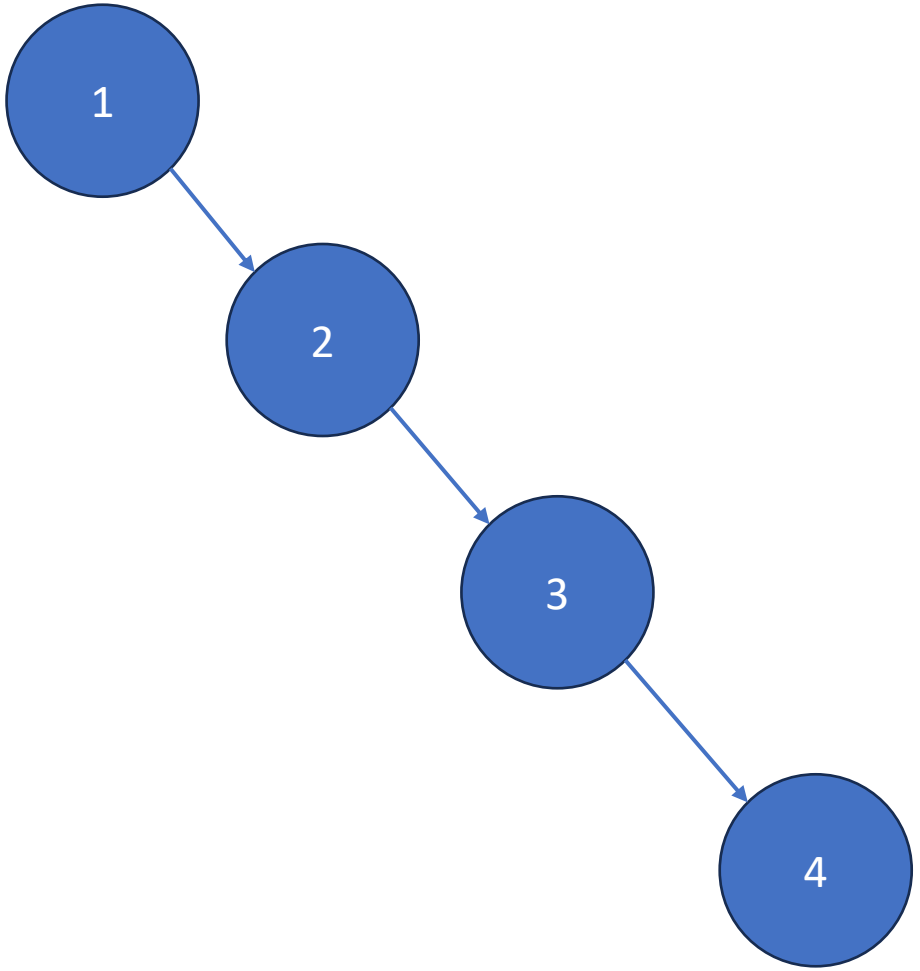


What's Good about BSTs?

- *Fast!*
  - Finding an item runs in *O(log(N))* time!
  - Inserting an item runs in *O(log(N))* time!
  - Erasing an item runs in *O(log(N))* time!
  - Even when *N* is large (billions) the number of comparisons needed is small (dozens)
- **BUT**
  - It is only fast if the tree remains *balanced*

Degenerate Case

- This happens when one inserts in sorted order (*not unusual*)
- Imagine a “tree” like this with a billion nodes
- It is just a linked list
- Look up now takes *O(N)* time



What To Do? Here's An Idea:

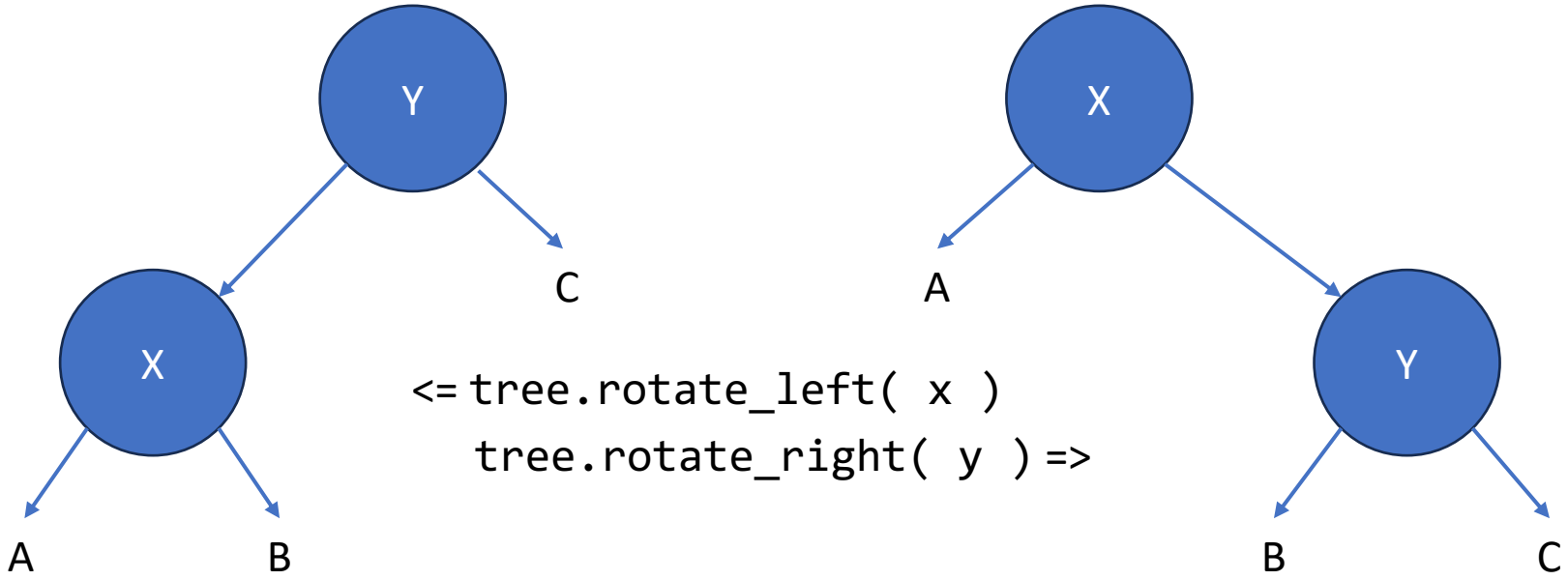
- After each insert operation, do extra work to rebalance the tree!
  - AVL trees: maintain "perfect" balance, but have a lot of overhead
  - Red-Black trees: maintain approximate balance, but with much less overhead (balance is good enough to maintain *O(log(N))* performance.
- Red-Black trees are extremely popular
  - `std::set` and `std::map` use them (most likely)
  - Java's `TreeSet` collection uses them (most likely)
  - Used inside the Linux kernel scheduler
  - Used all over the place!

Another Idea: Splay Trees

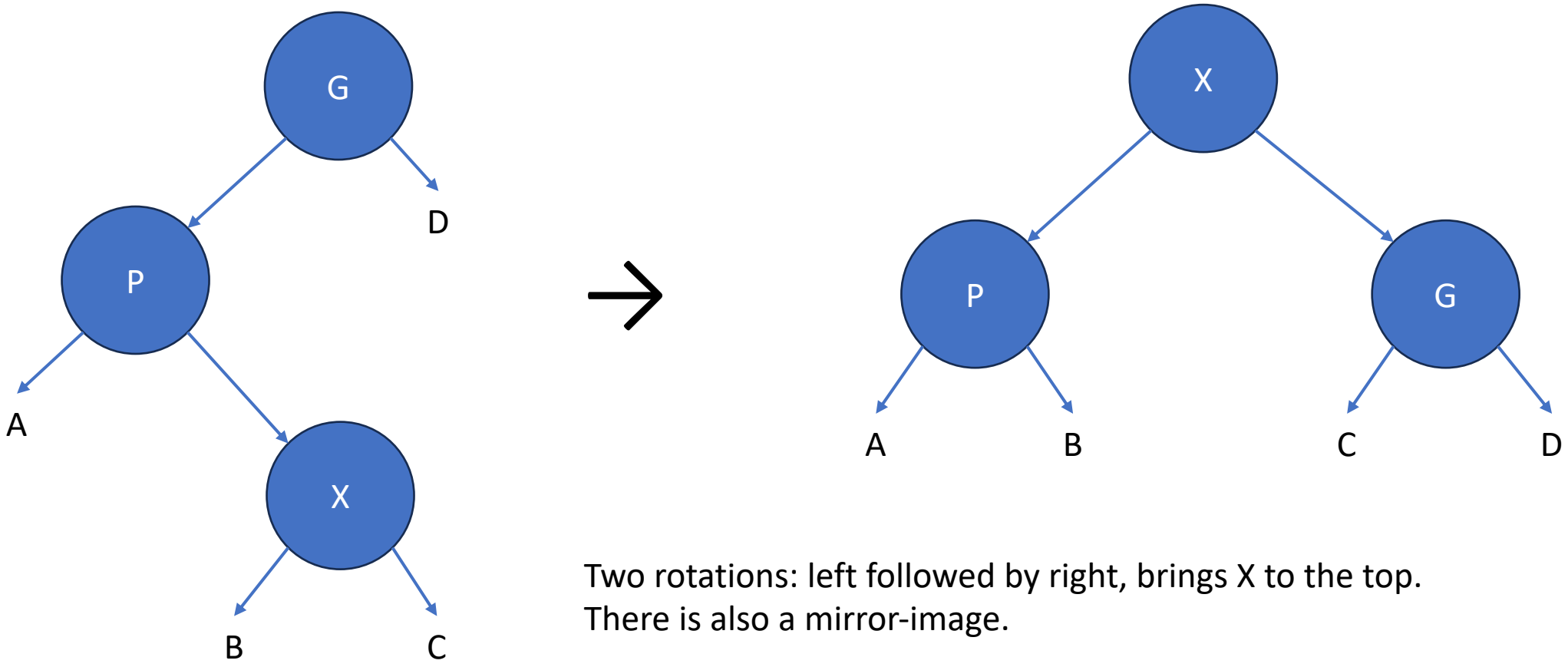
- Not every operation is *O(log(N))*.
  - Some operations are *O(N)*, but…
  - … they happen infrequently…
  - … giving *O(log(N))* on average (“amortized logarithmic time”)
- The trick is to ensure that repeated look up of the same value does not get stuck searching for an item deep in the tree
  - After each find, the item is brought to the root, and the tree is “splayed” to flatten it (somewhat).
  - Looking up a deep item might be slow, but the flattening process will push the tree toward being balanced on average.

Primitive Transformations: Rotations

- Note: BST still valid after rotation!

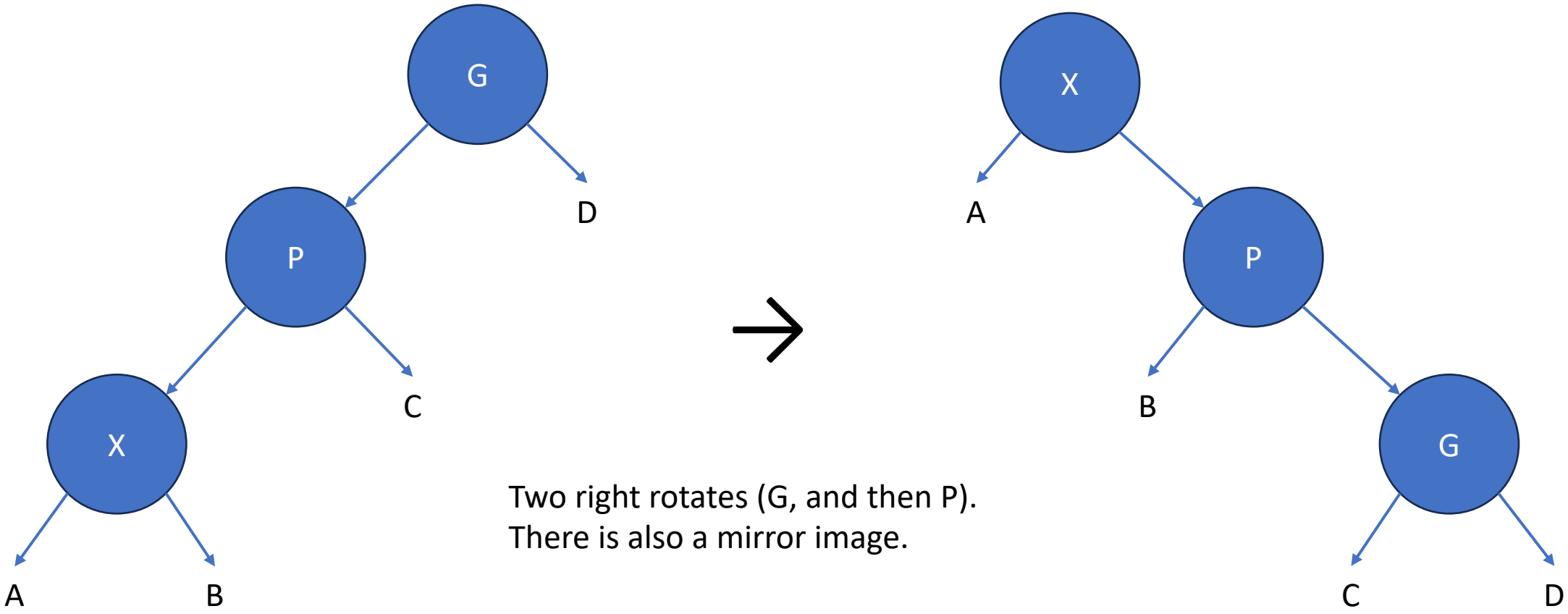


Splay Transformation: Zig-Zag



Two rotations: left followed by right, brings X to the top.
There is also a mirror-image.

Splay Transformation: Zig-Zig



Two right rotates (G, and then P).
There is also a mirror image.

Splay Tree Insert

- Do a normal BST insert
- If the new node is the root, we are done
- If the new node is an immediate child of the root, rotate it to the root
- Otherwise…
  - Work back up the tree (i.e., loop) doing Zig-Zag or Zig-Zig transformations
  - Each such transformation brings the new node up two levels
  - If the new node becomes an immediate child of the root, rotate it and stop
  - If the new node becomes the root itself, stop

Splay Tree Find

- Search for the node in the usual BST way
- Once found, move the node to the root using the splay tree transformations as described on the previous slides

Splay Tree Erase

- *NOT IMPLEMENTED!*
  - We will cover this later

Parent Pointers

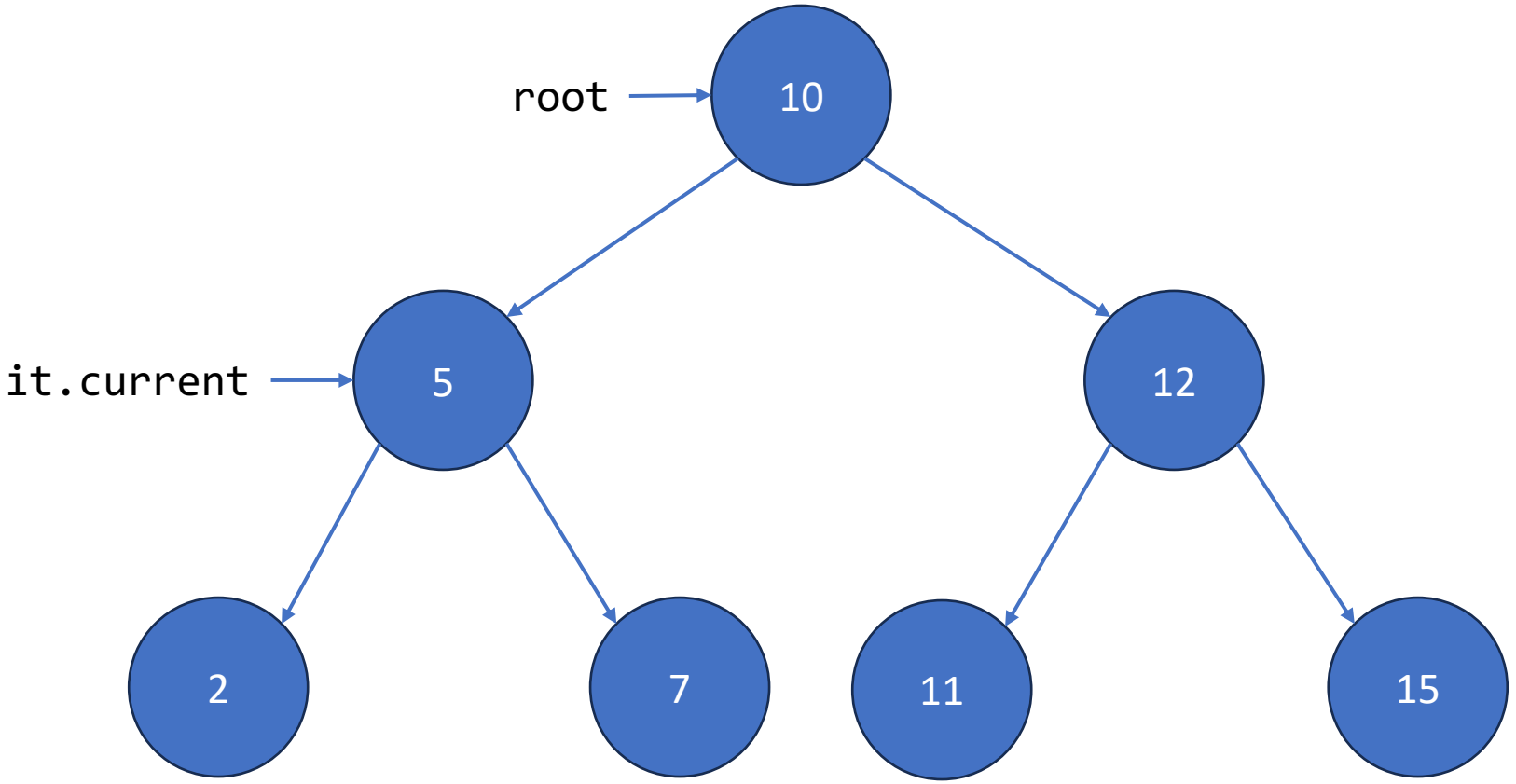
- Splaying requires that each node contain a pointer to its *parent*
  - The root node has a null parent
  - This is so you can find your way “up” the tree toward the root
  - Also, iterators need parent pointers to move through the tree
- There are alternative possibilities. For example:
  - When finding or inserting an item, you can remember the access path you used to get there (in, e.g., a vector of pointers to the nodes)
  - Iterators could contain similar vectors so they can backtrack up the tree
  - This saves space: no need to store a parent pointer (8 bytes?) in each node, but iterators are larger and more complicated

Smart Pointers

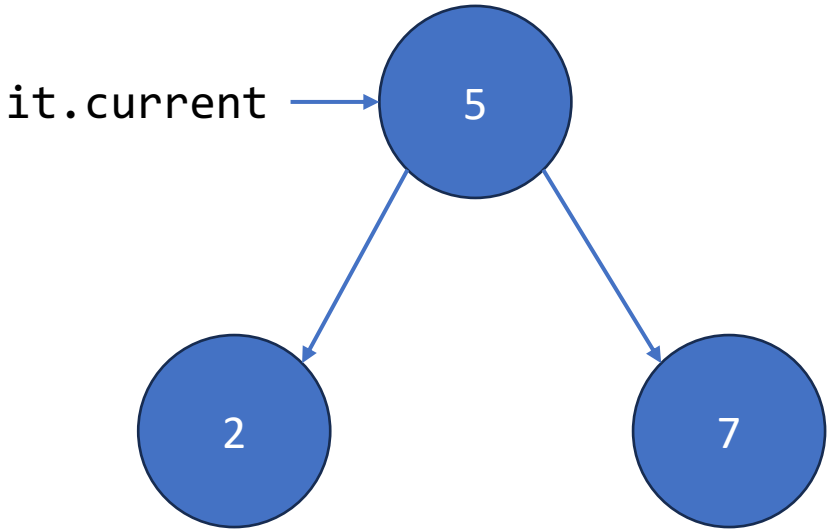
- If the left and right child pointers are *smart,* they can automatically release the tree's memory
- This also allows an iterator to remain valid even when the tree it points into is destroyed (the smart pointer in the iterator prevents destruction of the node it points at)
  - … although moving the iterator in this case will leak memory since only the subtree it points at will be destroyed*
- The child pointers should be `shared_ptr<Node>` so that they can be shared with iterators

* because the parent pointers need to be weak to break cycles

Tree with Iterator

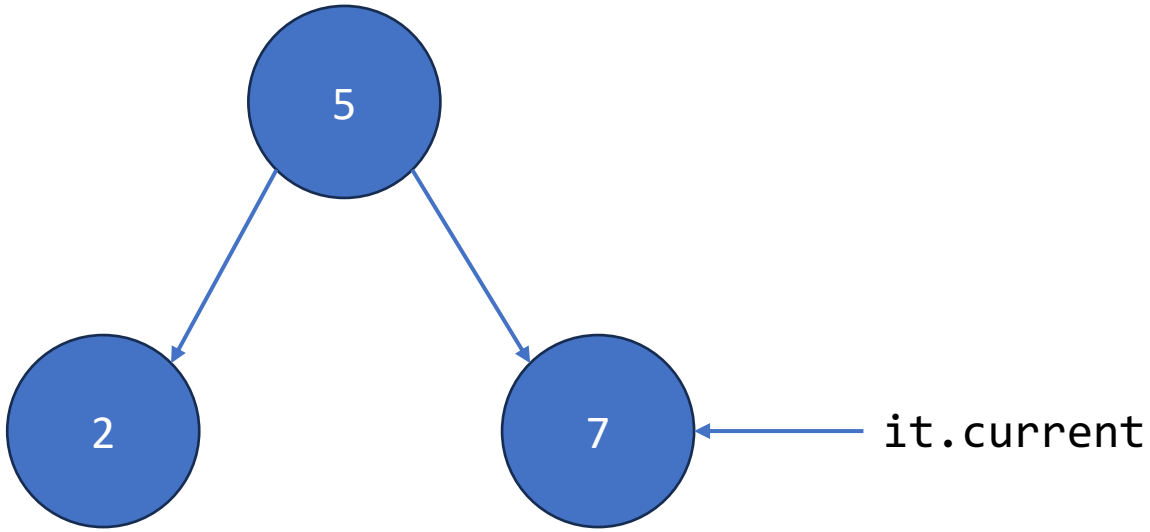


Tree After Root Destroyed

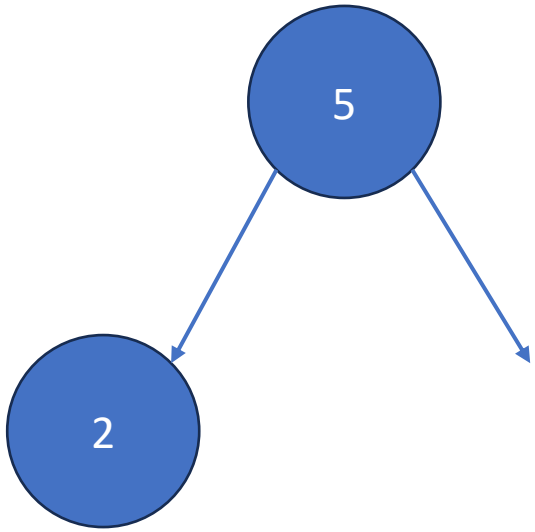


The node “5” continues to exist because there is another smart pointer in the iterator that points at it.

Tree After Iterator Is Incremented



Tree After Iterator Destroyed



Nodes “5” and “2” no longer have references.
Memory leak!

Conclusion: *Iterators remain valid after the tree Is destroyed, but do not move them!*