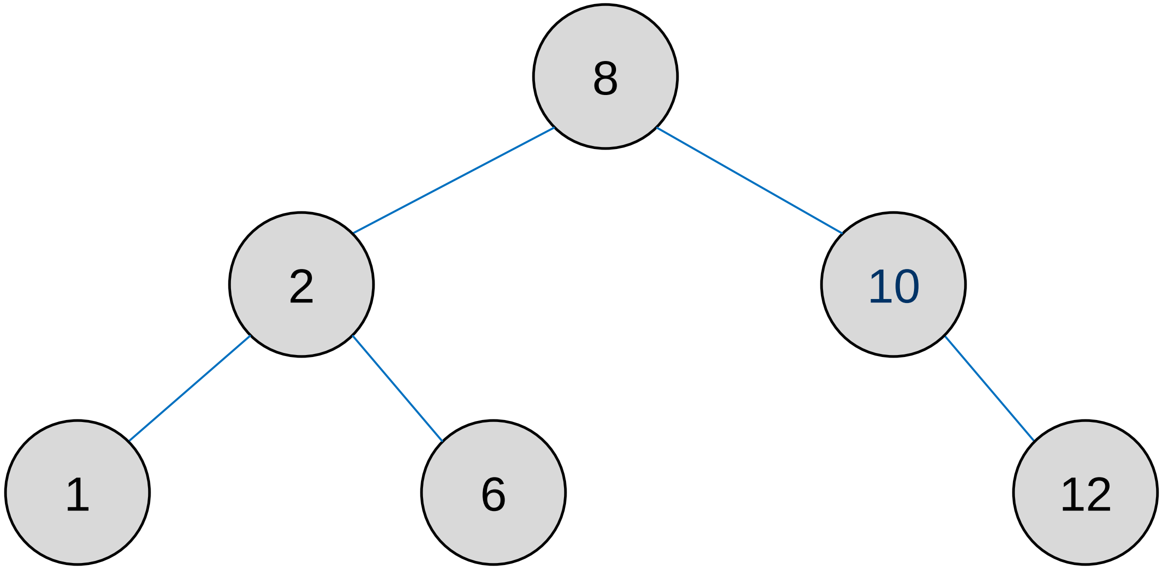
Insertion, Retrieval, and Deletion with Binary Search Trees

Inserting elements into a BST

- To insert a new element into a BST:
  - If the tree is empty, make the new element into the root
  - If the tree is not empty, compare the new element against the root
    - If the new element matches the root, do nothing (assuming we want to keep all our BST elements unique)
    - If the new element is less than the root, recursively insert the element into the root's left subtree
    - If the new element is greater than the root, recursively insert the element into the root's right subtree

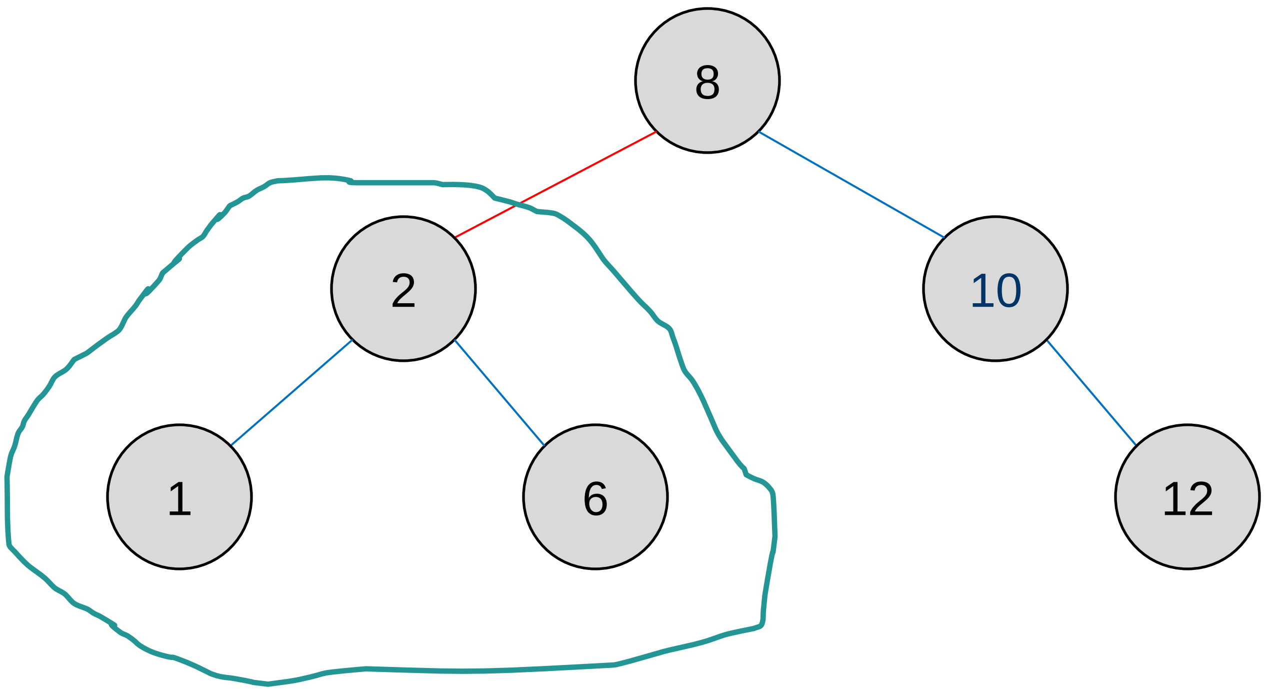
Inserting elements into a BST

Example: Insert 5 into the following BST:



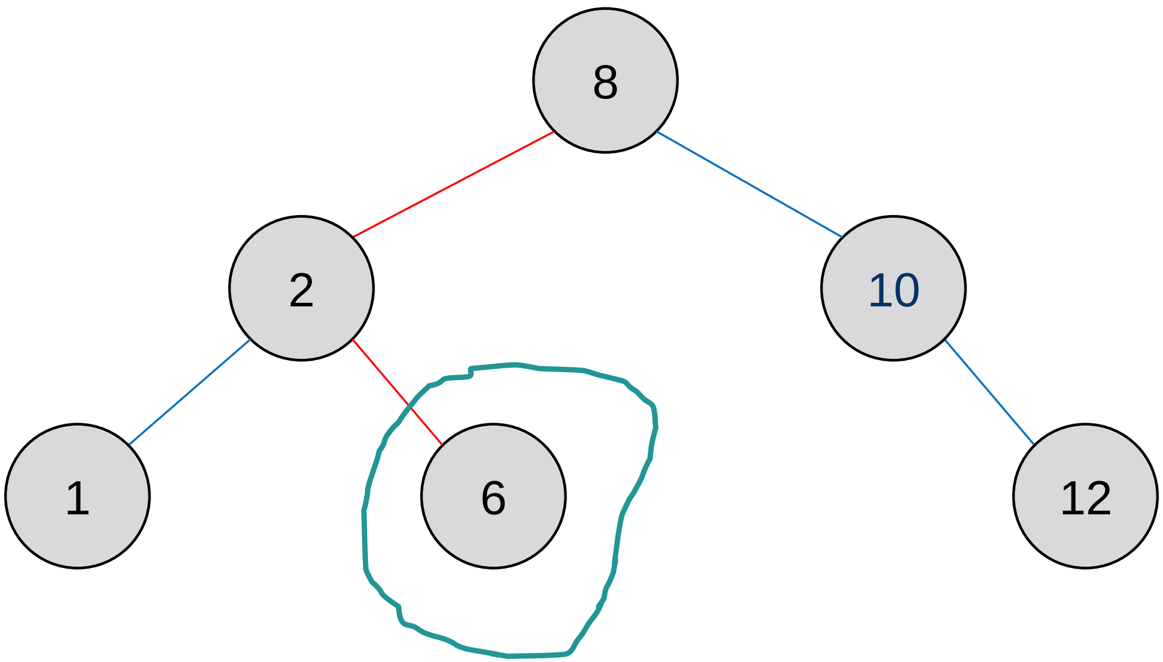
Inserting elements into a BST

Compare 5 against the root (8). Since 5 is less, recursively insert 5 into 8’s left subtree (circled below)



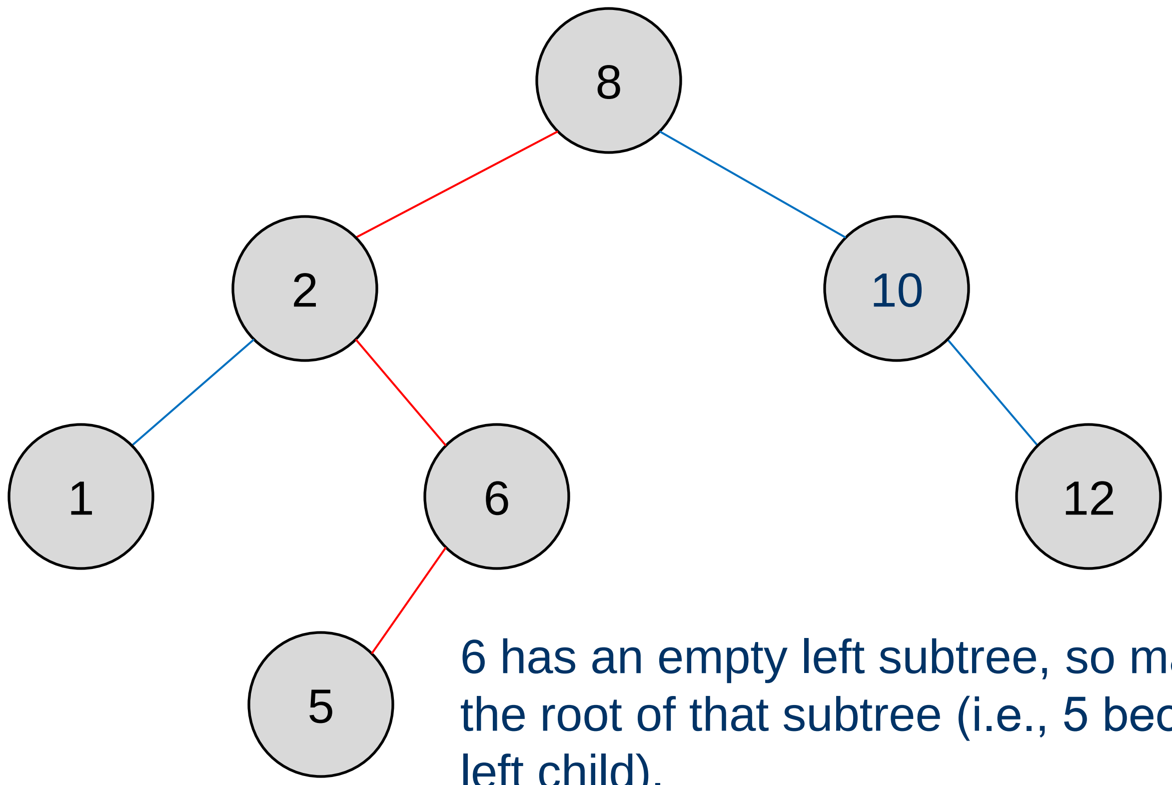
Inserting elements into a BST

Compare 5 against the subtree's root (2). Since 5 is greater, recursively insert 5 into 2's right subtree (circled below)



Inserting elements into a BST

Compare 5 against the subtree's root (6). Since 5 is less, recursively insert 5 into 6's left subtree.



6 has an empty left subtree, so make 5 into the root of that subtree (i.e., 5 becomes 6's left child).

Retrieving elements from a BST

- Retrieving (finding) an element follows a very similar procedure to adding an element:
  - If the tree is empty, return **null** to indicate the element was not found
  - If the tree is not empty, check the element to retrieve against the root
    - If the element matches the root, return the element
    - If the element is less than the root, recursively retrieve the element from the left subtree
    - If the element is greater than the root, recursively retrieve the element from the right subtree

Deleting elements from a BST

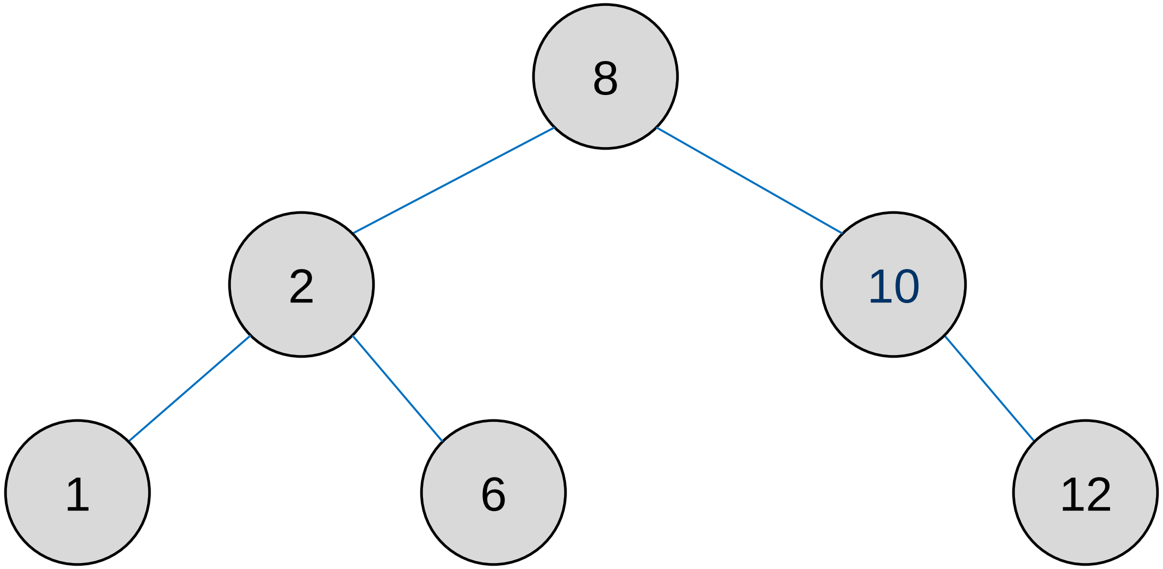
- To delete an element from a BST:
  - First find that element in the BST, using the previously discussed algorithm for retrieval
  - Once the element to delete is found, there are three cases to consider:
    - Case 1: The element to delete is a leaf node (has no children)
    - Case 2: The element to delete has one child
    - Case 3: The element to delete has two children

Deleting elements from a BST

- Case 1: Simply remove the node from the tree
- Case 2: Remove the node from the tree, replacing it with its only child
- Case 3:
  - Find the **in-order predecessor** of the node to remove (the maximum element from that node's left subtree)
  - Replace the node to remove with the in-order predecessor
  - Delete the in-order predecessor from the tree
- Note that case 3 can also work by using the **in-order successor** (the minimum element from the node's right subtree)

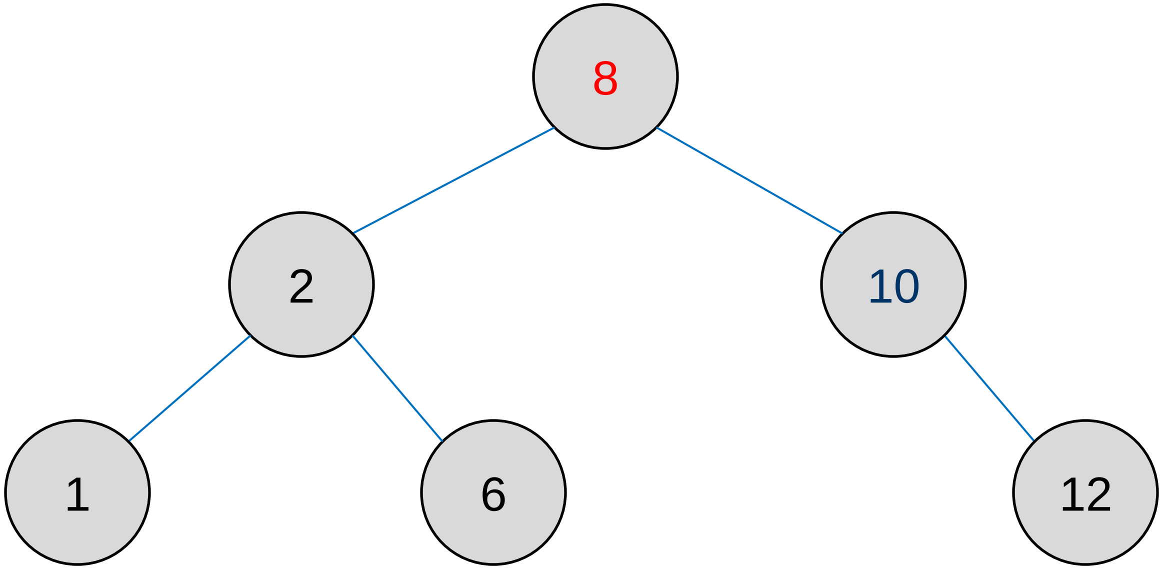
Deleting elements from a BST

Example: Delete 8 from the following BST:



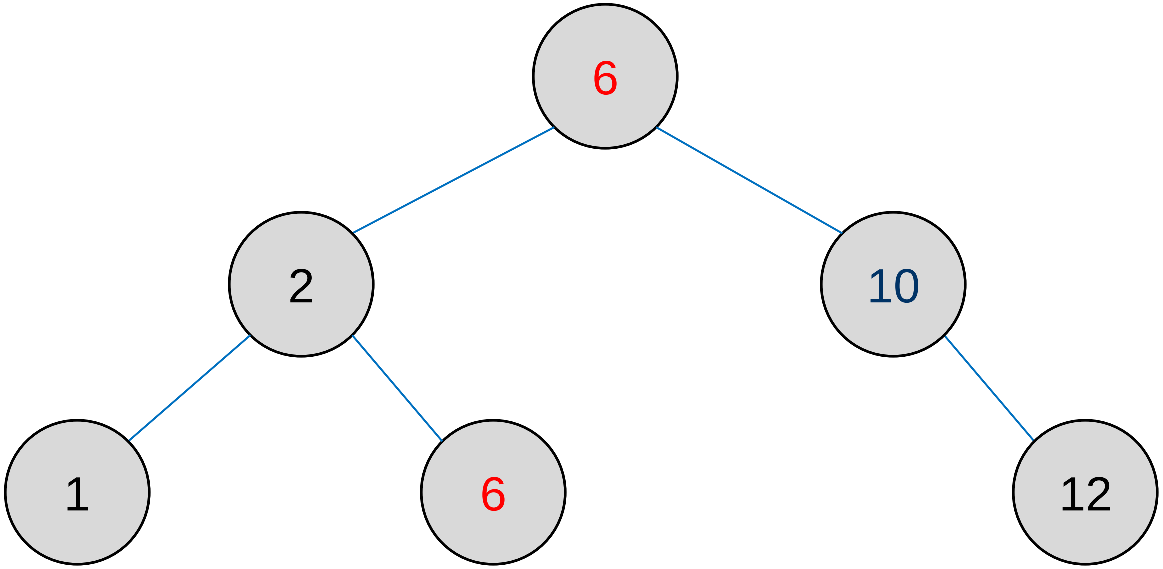
Deleting elements from a BST

First we find 8. That's pretty easy here, since 8 is the root! 8 has two children, so we must use Case 3.



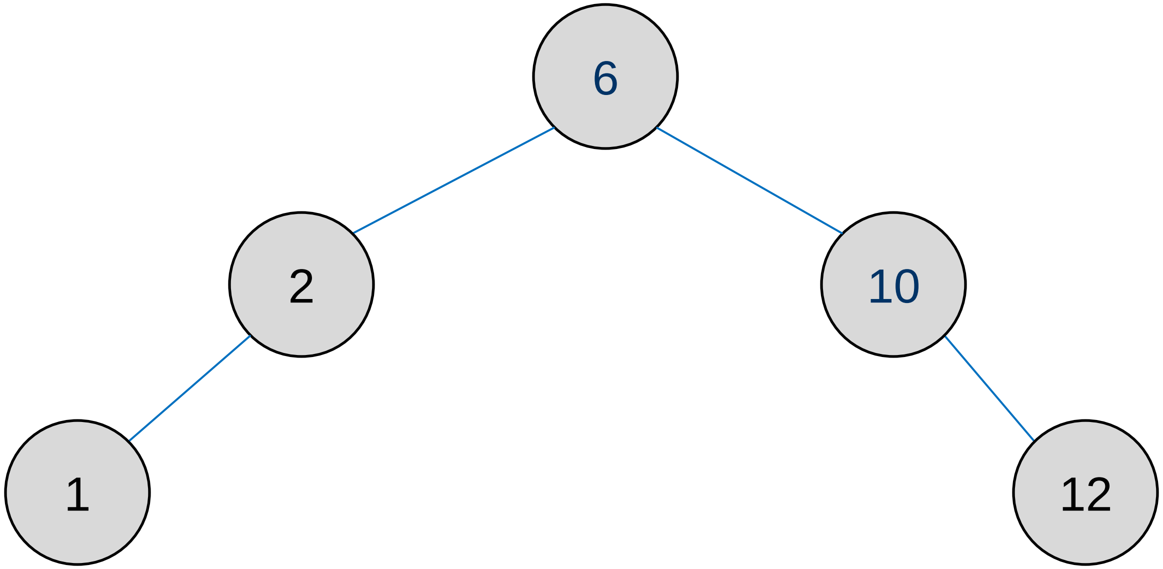
Deleting elements from a BST

Find 8's in-order predecessor (the maximum element from 8's left subtree), which is 6. Replace 8 with 6.



Deleting elements from a BST

Remove the in-order predecessor from the tree. The BST property is preserved!



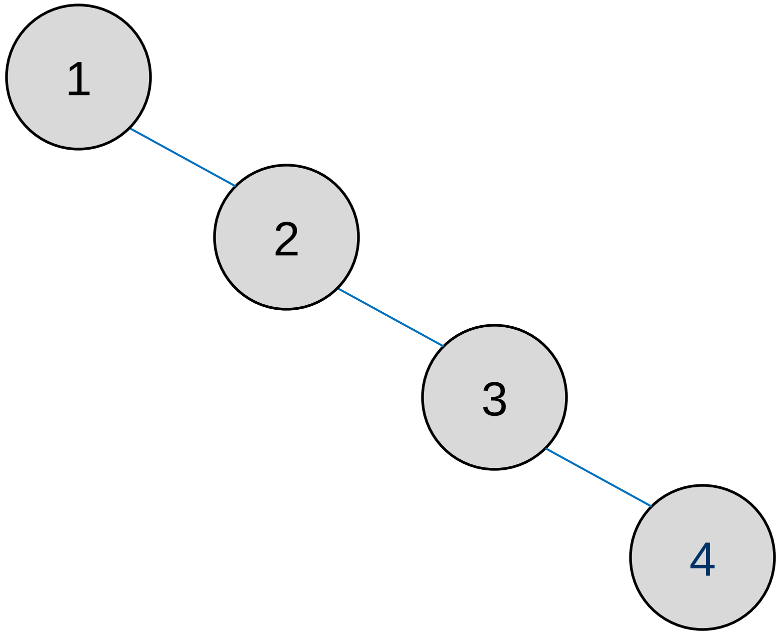
Note that the in-order predecessor will never have two children (if it did, its right child would be greater, and hence it wouldn't be the maximum element in that subtree). So removing the in-order predecessor is guaranteed to be easy (Case 1 or 2 of deletion).

Analysis of BST operations

- Insertion, retrieval, and deletion are all $O(\log n)$ operations, as long as the BST is well-balanced
  - Each time we decide which direction to go from a node, we are eliminating half of the remaining nodes in the tree
  - Remember that an algorithm that halves its input size each time it runs is usually $O(\log n)$

Analysis of BST operations

- But what if the tree isn't well-balanced? Consider the BST that is formed by adding the elements 1, 2, 3, 4 in that order:



Analysis of BST operations

- This is pretty much just a linked list!
  - Insertion, retrieval, and deletion all become $O(n)$ operations since we potentially need to look through all the nodes
- A BST has average-case performance of $O(\log n)$ for insertion, retrieval, and deletion, but the worst-case performance is $O(n)$
- TL;DR - Maintaining balance in a BST is important!
- Later this semester (Ch. 9) we'll discuss several techniques for ensuring that a tree always keeps itself in balance