Binary Search Tree

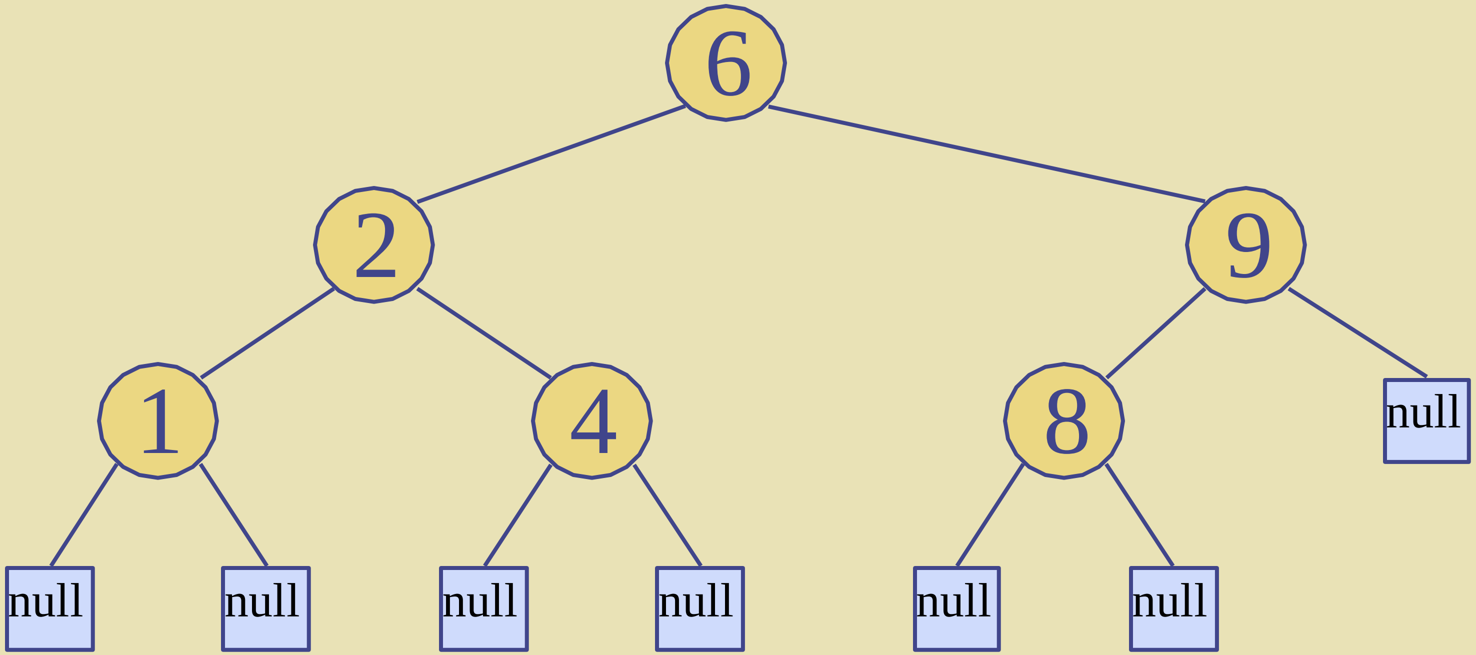
Notes by Yan Yan

Contents

1. Binary Search Tree
2. Binary Search Tree Operations

Binary Search Tree

- Nodes are arranged in an order.
- Assume no duplicates values allowed, then
  - Nodes in the left sub-tree have a value less than (<) the value of the root node.
  - Nodes in the right sub-tree have a value greater than (>) the value of root node.



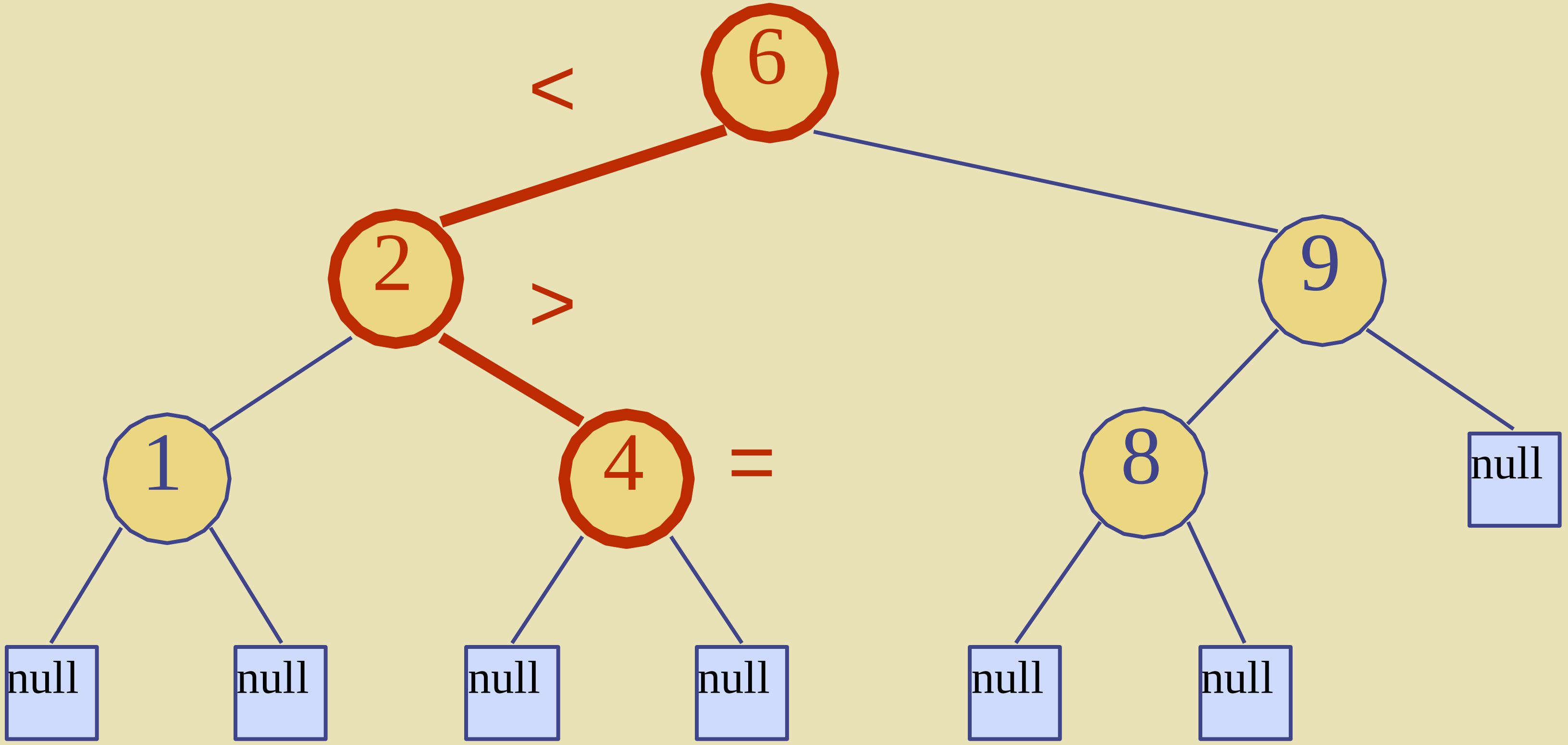
Binary Search Tree -- Search

- To search for value (key = k), we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found, we return NO_SUCH_KEY

```
BSTSearch(tree, key)
   cur = tree⇢root
   while (cur is not null) do
      if (key == cur⇢key) then
         return cur  // Found
      else if (key < cur⇢key) then
         cur = cur⇢left
      else
         cur = cur⇢right
   end while
   return null  // Not found
```

Binary Search Tree -- Search

- Example: search k=4 in this tree. BSTSearch(tree, 4)



Binary Search Tree -- Search

- Exercise: What are the orders of nodes visited, if you search for
    - A. 145
    - B. 900

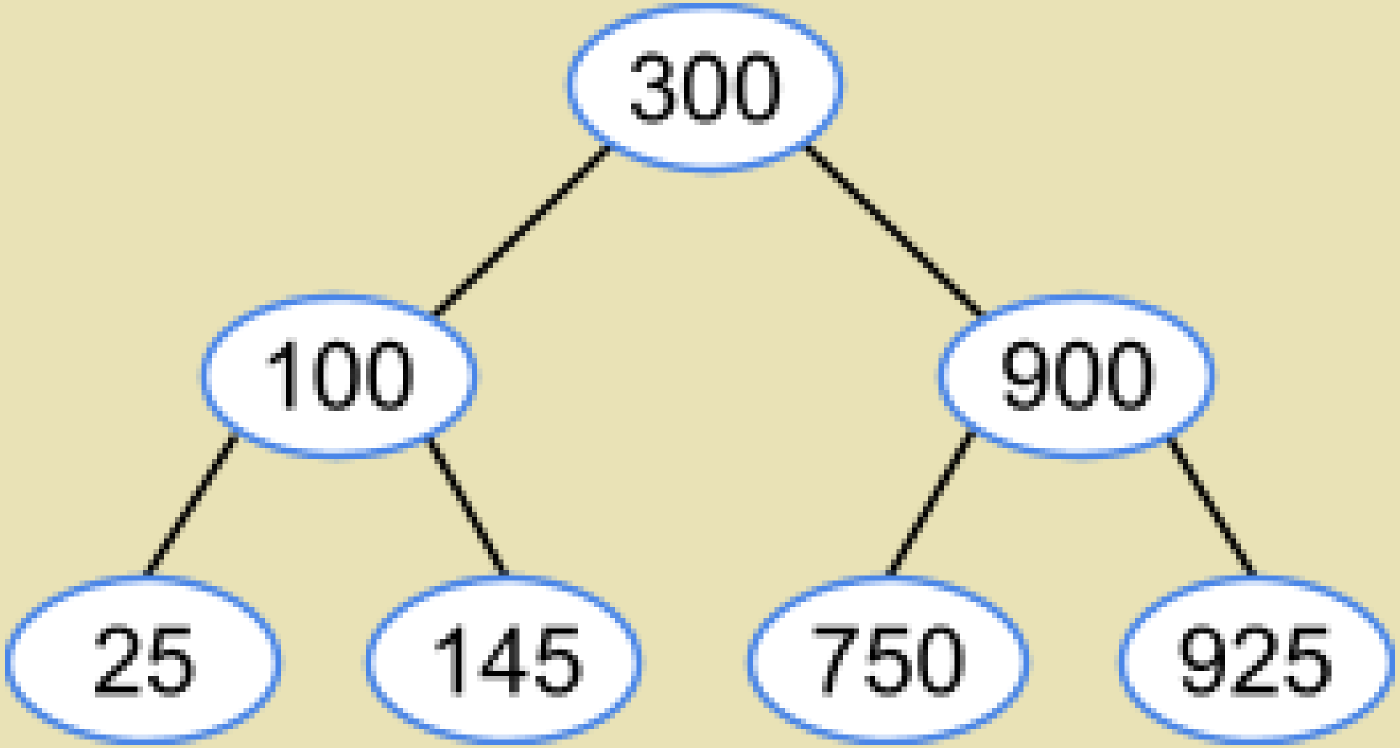


Figure from 7.3 7.3 Binary search trees

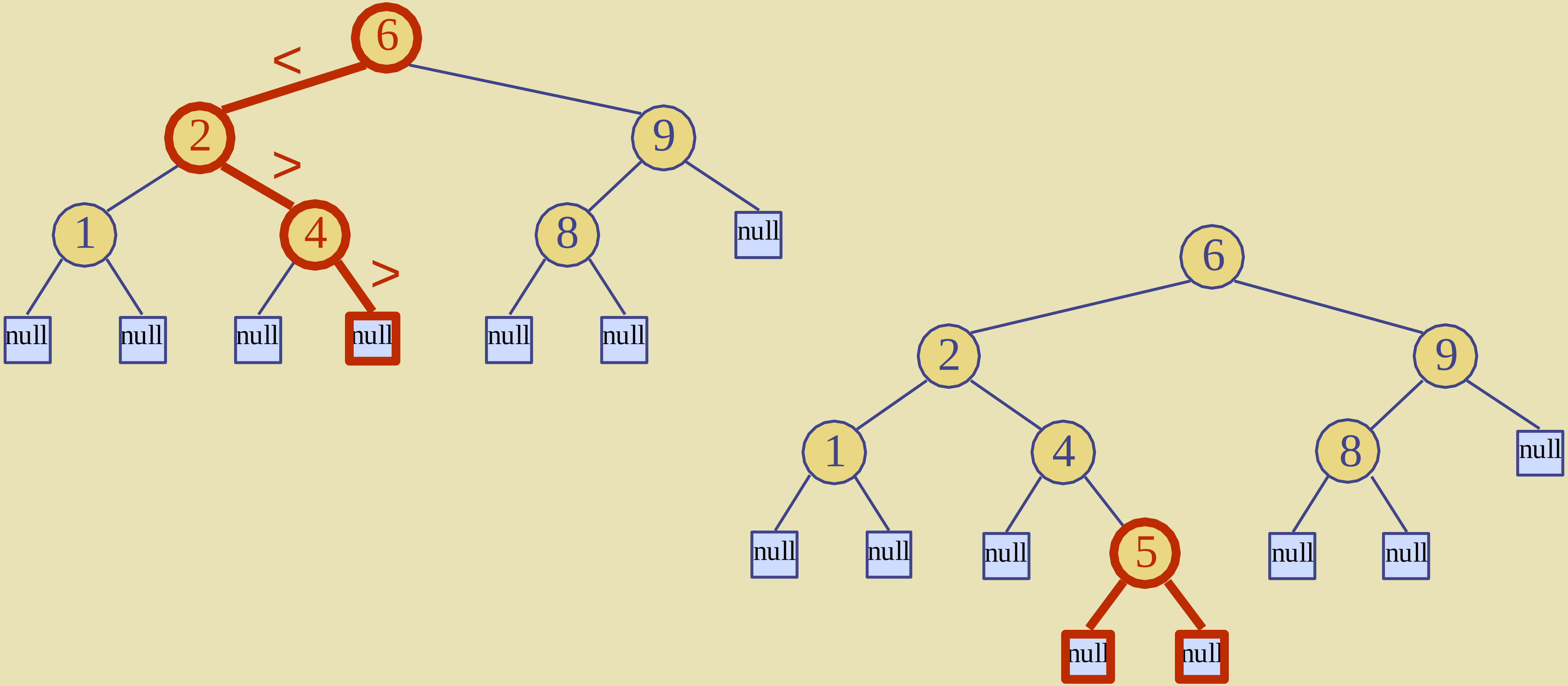
Binary Search Tree -- Insertion

- To perform insertion, we search for key k, and find a suitable location to insert as the leaf.
- Special case: A node inserted into an empty tree will become the tree's root.

```
BSTInsert(tree, node)
  if (tree⇢root is null) then
    tree⇢root = node
  else
    currentNode = tree⇢root
    while (currentNode is not null) do
      if (node⇢key < currentNode⇢key) then
        if (currentNode⇢left is null) then
          currentNode⇢left = node
          currentNode = null
        else
          currentNode = currentNode⇢left
        end if
      else
        if (currentNode⇢right is null) then
          currentNode⇢right = node
          currentNode = null
        else
          currentNode = currentNode⇢right
        end if
      end if
    end while
  end if
```

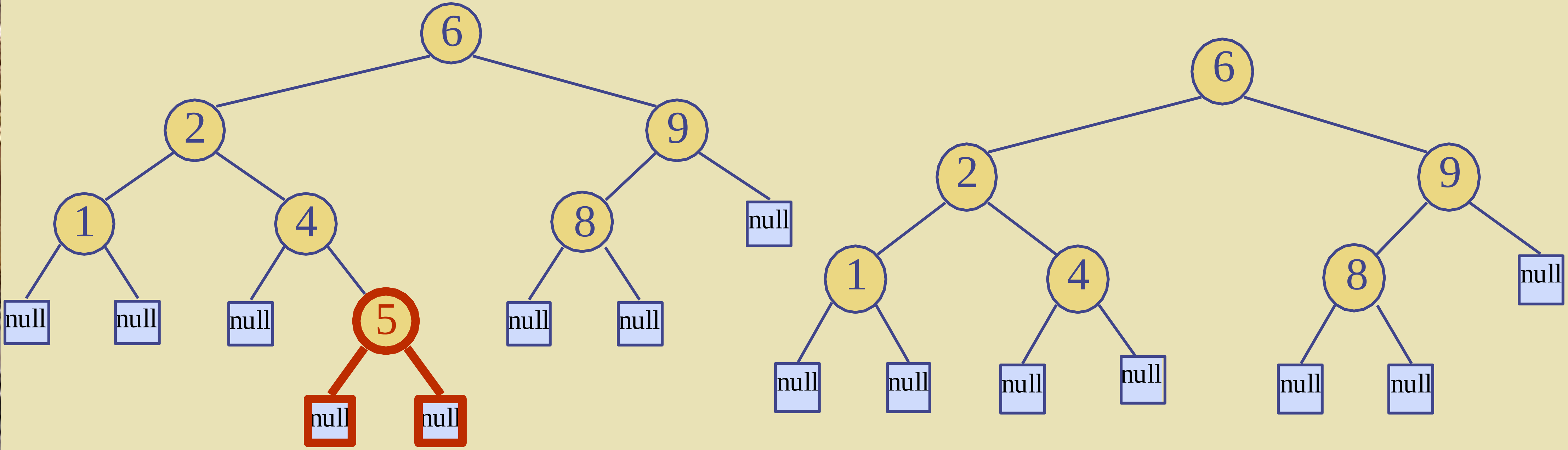
Binary Search Tree -- Insertion

- Example: insert 5 in this tree. BSTInsert(tree, 5)



Binary Search Tree -- Removal

- The algorithm first searches for a matching node just like the search algorithm. If found (call this node X), the algorithm performs one of the following sub-algorithms:
  1. Remove a leaf node: If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with null. Else, if X was the root, the root pointer is assigned with null, and the BST is now empty.

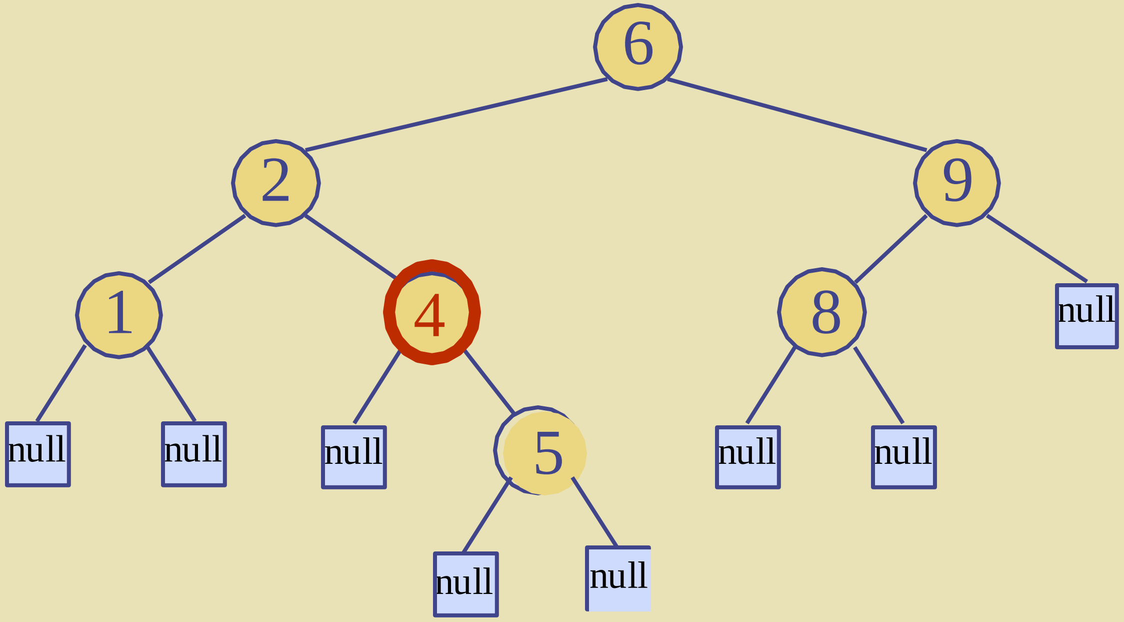


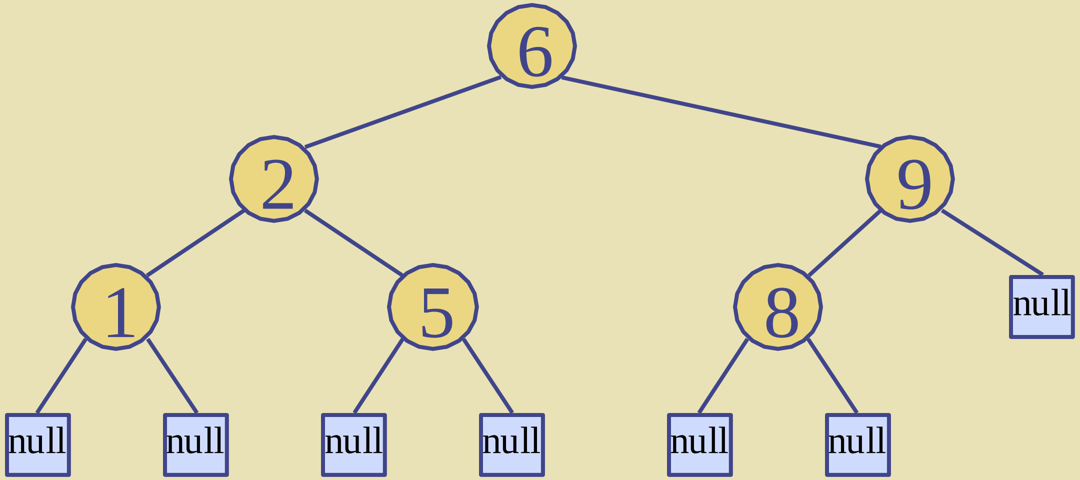
Binary Search Tree -- Removal

- The algorithm first searches for a matching node just like the search algorithm. If found (call this node X), the algorithm performs one of the following sub-algorithms:
  1. Remove a leaf node: If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with null. Else, if X was the root, the root pointer is assigned with null, and the BST is now empty.
  2. Remove an internal node with single child: If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with X's single child. Else, if X was the root, the root pointer is assigned with X's single child.

Binary Search Tree -- Removal

- Remove an internal node with single child: If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with X's single child. Else, if X was the root, the root pointer is assigned with X's single child.





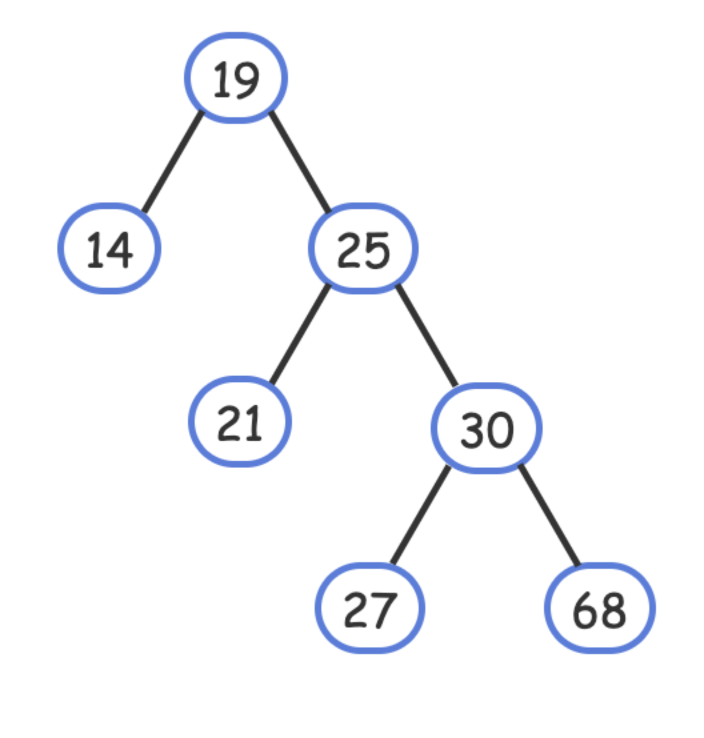
Binary Search Tree -- Removal

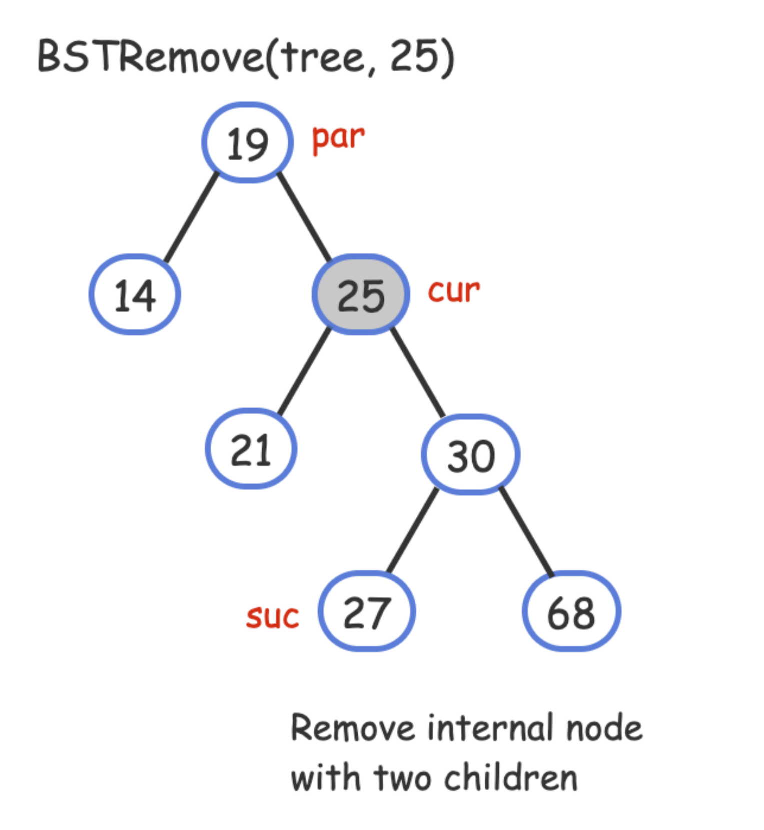
- The algorithm first searches for a matching node just like the search algorithm. If found (call this node X), the algorithm performs one of the following sub-algorithms:
  1. Remove a leaf node: If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with null. Else, if X was the root, the root pointer is assigned with null, and the BST is now empty.
  2. Remove an internal node with single child: If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with X's single child. Else, if X was the root, the root pointer is assigned with X's single child.
  3. Remove an internal node with two children: This case is the hardest. First, the algorithm locates X's successor (the leftmost child of X's right subtree), and copies the successor to X. Then, the algorithm recursively removes the successor from the right subtree.

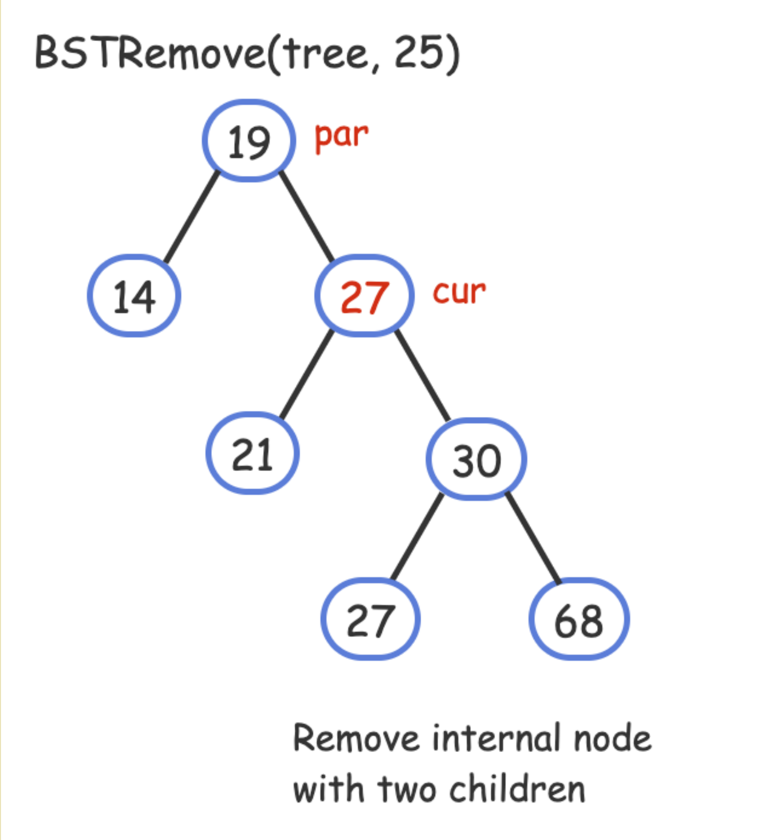
Binary Search Tree -- Removal

- Remove an internal node with two children: This case is the hardest. First, the algorithm locates X's successor (the leftmost child of X's right subtree), and copies the successor to X. Then, the algorithm recursively removes the successor from the right subtree.

- Remove 25



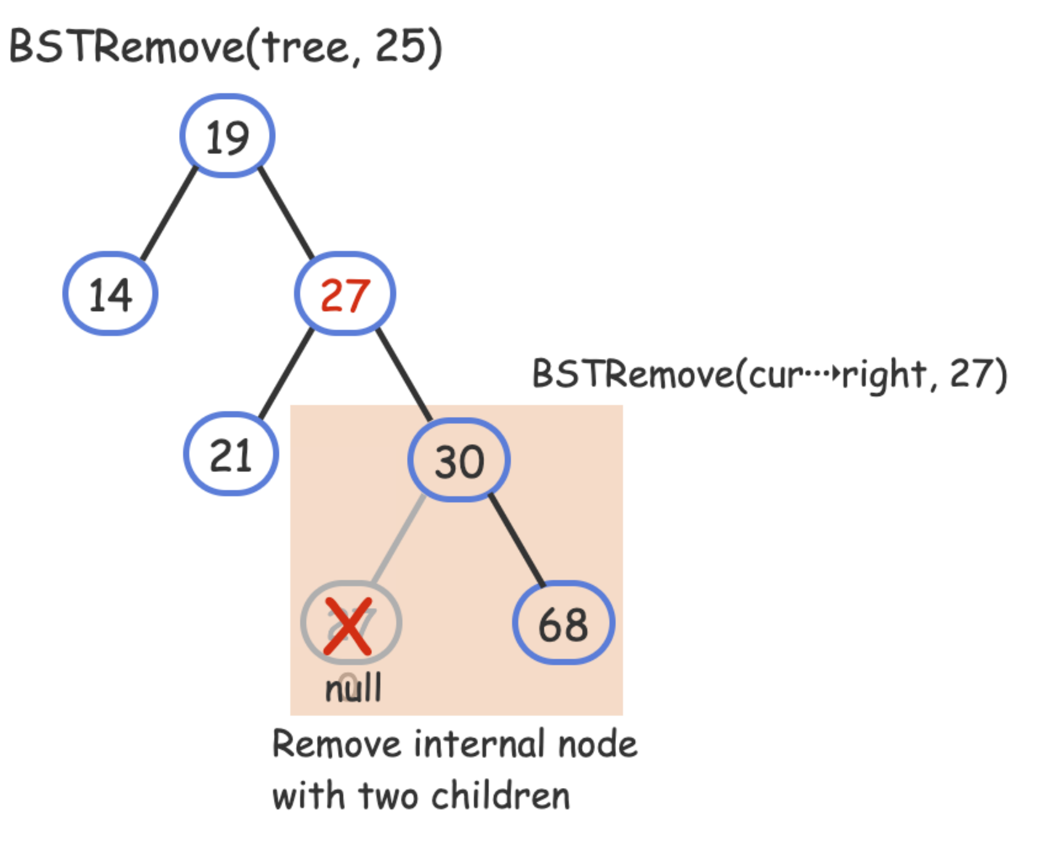




Binary Search Tree -- Removal

- Remove an internal node with two children: This case is the hardest. First, the algorithm locates X's successor (the leftmost child of X's right subtree), and copies the successor to X. Then, the algorithm recursively removes the successor from the right subtree.

- Remove 25



Binary Search Tree -- Removal

- The algorithm first searches for a matching node just like the search algorithm. If found (call this node X), the algorithm performs one of the following sub-algorithms:
  - Remove a leaf node: If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with null. Else, if X was the root, the root pointer is assigned with null, and the BST is now empty.
  - Remove an internal node with single child: If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with X's single child. Else, if X was the root, the root pointer is assigned with X's single child.
  - Remove an internal node with two children: This case is the hardest. First, the algorithm locates X's successor (the leftmost child of X's right subtree), and copies the successor to X. Then, the algorithm recursively removes the successor from the right subtree.

That's about this lecture!