Binary Search Trees

(BST)

Binary Tree

- Binary Tree:
  - New nodes can be placed at any part of the binary tree
  - For finding a node we have to traverse the tree
  - For deleting a node, we have to traverse the tree to find and delete the node

  The above processes are not efficient for a general binary tree

- In a Binary tree:
  - contain() method is O(n) on average, because there is no order to the nodes, and so each node might need to be checked
  - what would the contains() algorithm look like?
    - check the node, and then recursively check both the up and down children

Binary Search Tree (BST) / Sorted Tree

- A Binary Search Tree is still a binary tree
- Nodes are placed/arranged in a binary search tree according to their values
- We have an algorithm for adding nodes to a BST
- We have an algorithm for deleting a node from a BST
- We have an algorithm for searching a BST

Binary Search Tree (BST) / Sorted Tree

New nodes are added to a BST such that:

- For each node:
  - The up child has a value that is greater than the node’s value
  - The down child has a value that is less than the node’s value

- Accordingly, at each node:
  - The values in the node’s up subtree are greater than the node’s value
  - The values in the node’s down subtree are less than the node’s value
- Generally, duplicates values are not allowed in BSTs

Performance of operations on a BST

The performance of operations on a BST depends on the structure of a BST.

For a balanced BST adding, deleting, searching take place in O(log n)

What can you store in a BST?

Nodes of a BSTs should have an order. Therefore, a BST can only store values that are comparable:

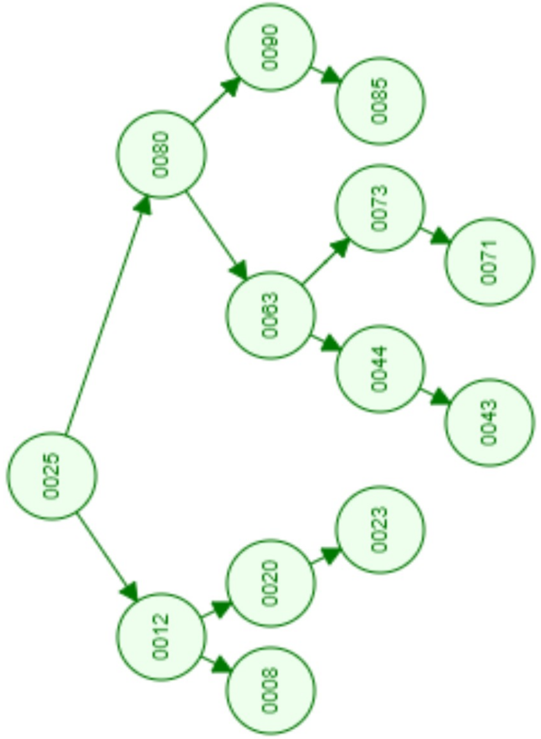
- Numbers
- Characters
- Strings

The following type of variables cannot be stored in BSTs:

- Booleans

Interesting properties of a BST

- In a BST, the in-order traversal will always give the numbers in sorted order.
- The smallest value in a BST can be found by starting at the root and repeatedly moving down until you reach null.
- The largest value in a BST can be found by starting at the root and repeatedly moving up until you reach null.



BST algorithms that we develop:

- add()
- contain()
- delete()

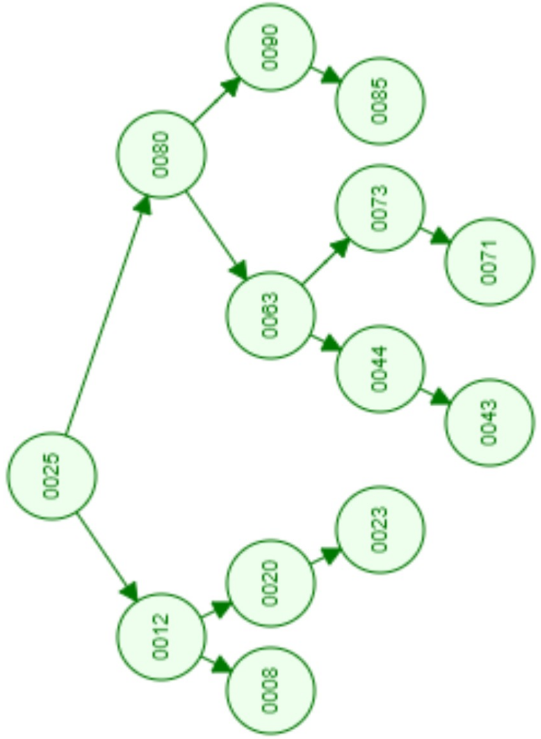
add() and delete() methods must be implemented in a way that BST remains organized/ordered.

contain()

- In a binary search tree:
    - the order of the nodes leads to more efficient searching
    - searching is *potentially* O(log n)
    - what would the contains() algorithm look like?
        - check the node, and then recursively check only one descending subtree depending on the node’s

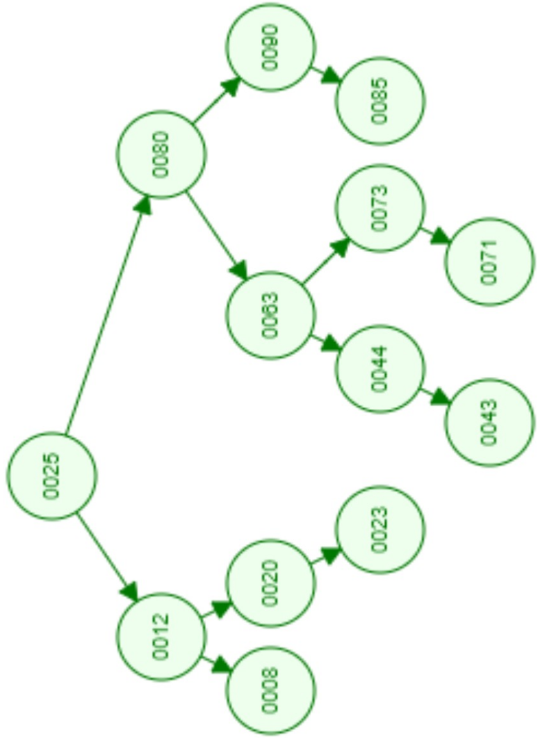
contains(44) in a BST

- Start at the root:
  - Root is 25
  - 44 > 25
  - We check the up subtree
  - We reach the node 80
  - 44 < 80
  - We check the down subtree
  - We reach the node 63
  - 44<63
  - We check the down subtree
  - We reach the node 44
  - We return True



contains(15) in a BST

- Start at the root:
  - Eventually, the algorithm will reach the node 20
  - 15 < 20
  - we should check the down subtree of 20
  - But the down subtree is null
  - We return False



add() method

- Adding new nodes to a BST nodes must be done in a way that keeps the order of the nodes

- For BST nodes, the add() algorithm is very similar to the contain() algorithm in the way you move through the tree
    - add() method keeps moving down or up based on the values. Once a null space is found actual insertion of the new node in the BST takes place.

delete() method

- Deleting nodes from a BST must be done in a way that keeps the order of the nodes

Deleting nodes

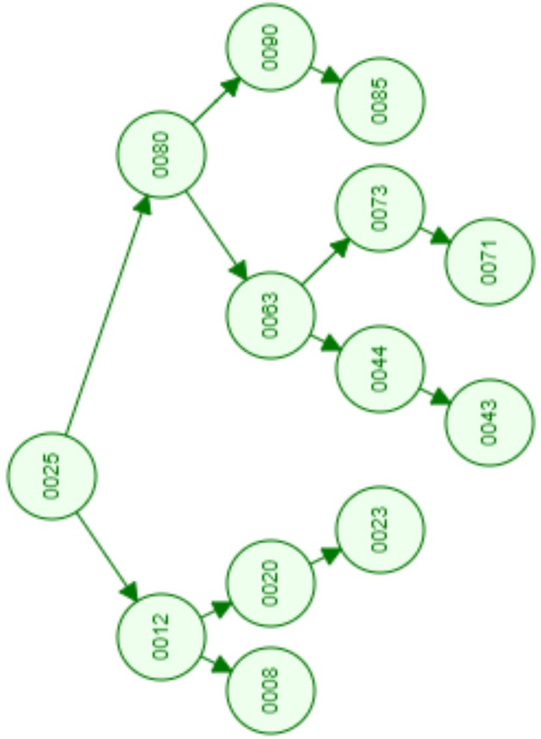
- Three main cases to consider:
    - Removing a node that has <u>0</u> child nodes
    - Removing a node that has <u>1</u> child node
    - Removing a node that has <u>2</u> child nodes

delete(43)

- 43 has no children

44 is the parent of 43

We set the down child of 44 node to be null



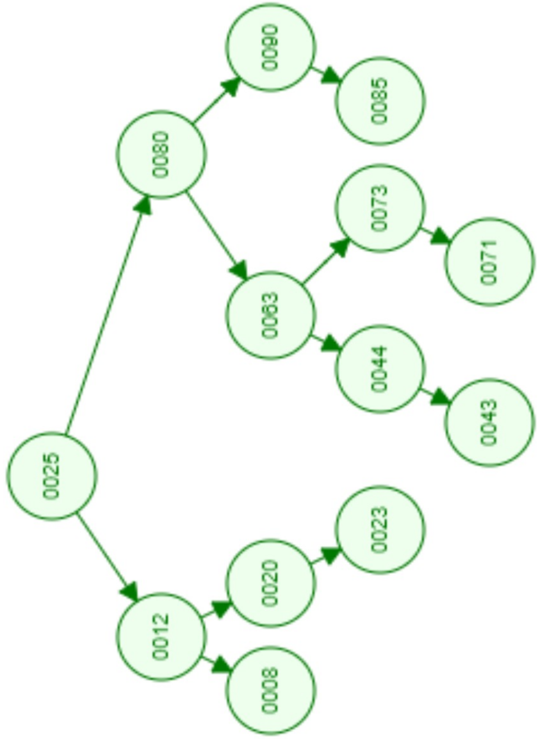
delete(44)

- 44 has 1 child

43 is the child of 44

The parent of 44 is 63

43 is set as the down child of 63



delete(80)

- 80 has 2 children

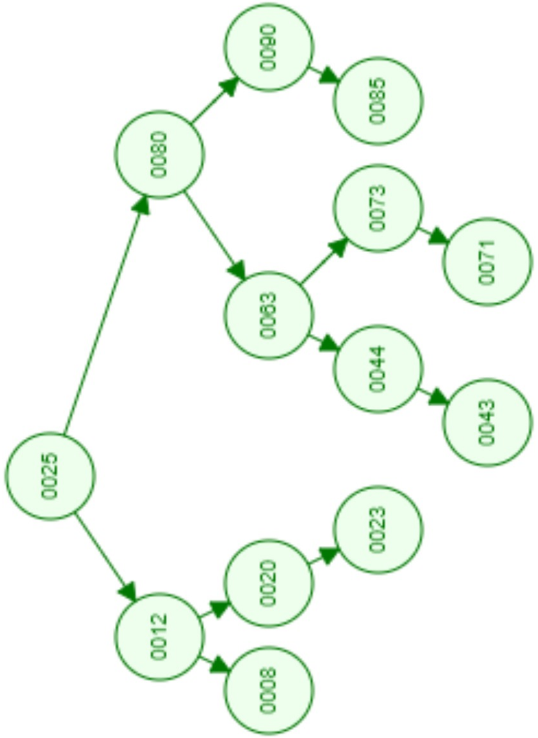
The nodes in the up subtree all have a value greater than 80

Among these nodes 85 has the smallest value

We copy 85 into 80 node

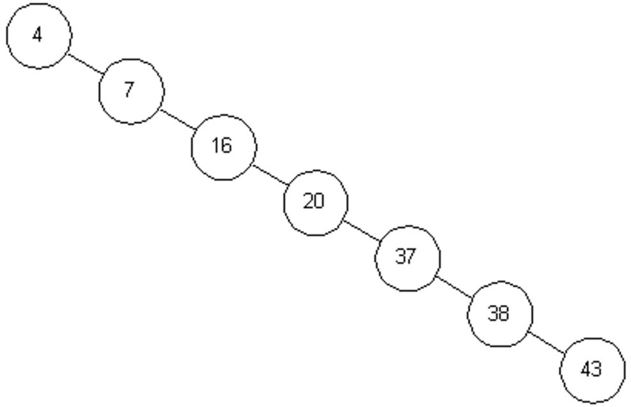
80 is deleted and ordered of nodes is preserved

85 is duplicate. We proceed with deleting the duplicate 85 node

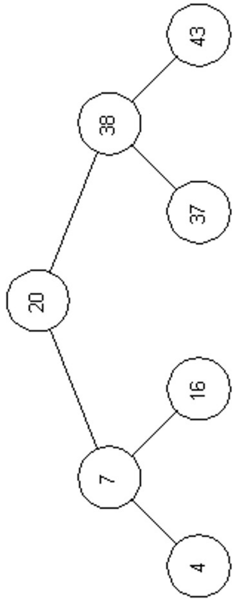


Efficiency in a BST with n nodes depends on shape:

- If the BST is **unbalanced**, then add(), delete(), contain() tend toward **O(n)** on average



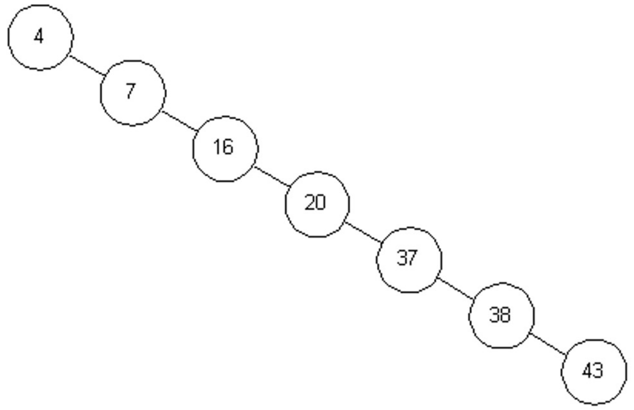
- If the BST is **balanced**, then add(), delete(), contain() tend toward **O(log n)** on average

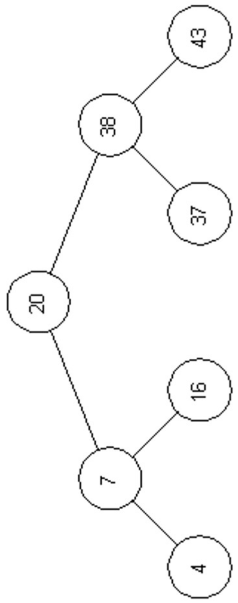


https://www.eecs.umich.edu/courses/eecs380/ALG/niemann/s_bin.htm

Efficiency in a BST with length h:

- We often express BST efficiency in terms of length. The efficiency of add(), delete() and contain() for all BSTs on average is O(h).





https://www.eecs.umich.edu/courses/eecs380/ALG/niemann/s_bin.htm

Balanced BSTs are better than unbalanced...

With regard to efficiency of add(), contain() and delete() methods, balanced BSTs are better than unbalanced ones. So, often, efforts are made to keep BSTs balanced.