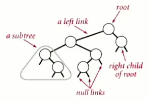
Binary search trees

BST is a binary tree in symmetric order

* Tree is either empty or
* Two disjoint binary trees on the left and right



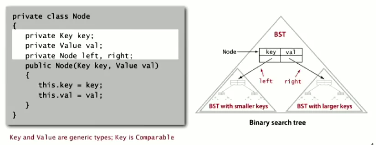
Each node has a key, each key is:

* Larger than all keys on left subtree
* Smaller than all keys on right subtree

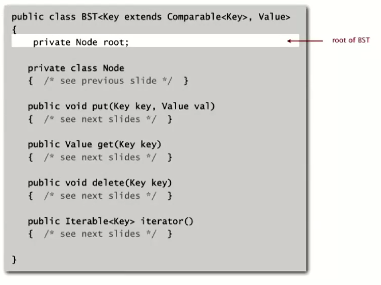
BST is a reference to a root *Node*

A *Node* has four fields:

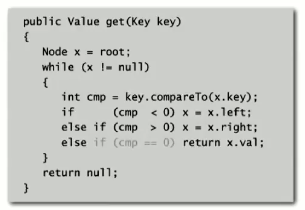
* A key and a value
* A reference to left and right subtrees



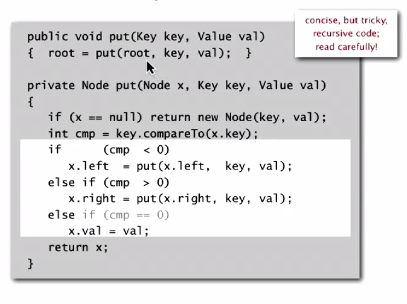
BST implementation



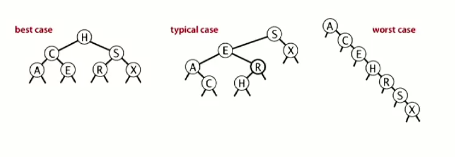
BST Search implementation



BST insert implementation

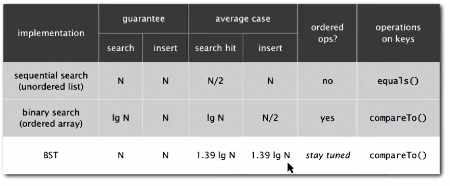


Tree shape depends on order of insertion



BSTs correspond exactly to quicksort partitioning

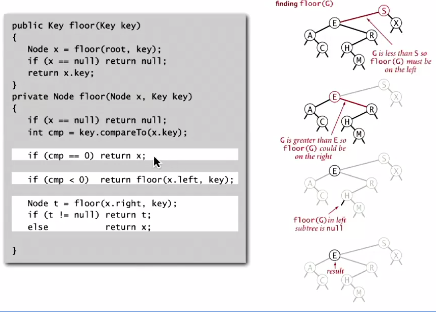
Run time Big O:



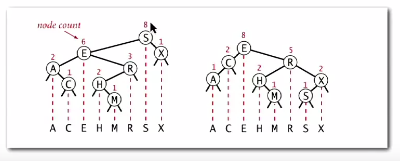
**Ordered operations in BSTs**

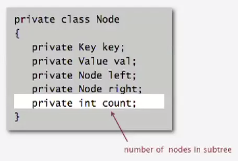
Min/max: Smallest key is all the way to the left from root, largest we move all the way to the right

Floor implementation and explanation below. Similar code for ceiling computation.



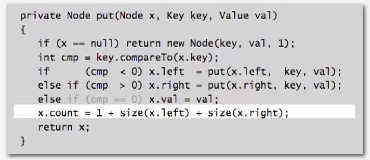
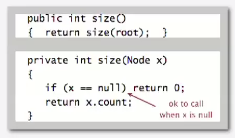
For size, we simply store a count of number of nodes at each subtree (return size() at root)





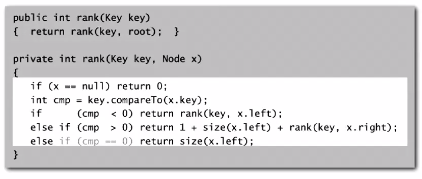
This makes rank() and select() easier to implement

Simple implementation:



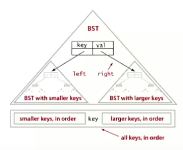
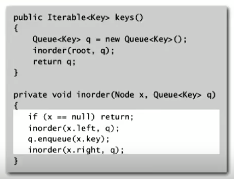
Rank operation is how many keys are less than *k*: return size of left subtree

* If key == current node, then keys less than our keys is == size of the left subtree
* If key < current node, then keys == rank of left subtree
* If key > current node, then keys == 1 + size(x.left) and rank of right

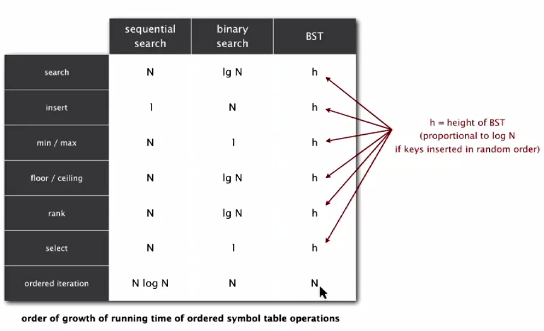


Inorder traversal: [RECURSIVE]

1. Traverse left subtree
2. Enqueue key
3. Traverse right subtree

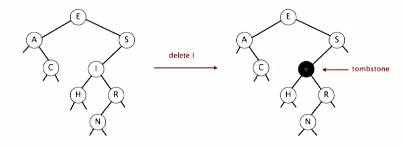


Operation run time for BST:



**Deletion in BSTs**

LAZY: set value to null (leave key for searches)

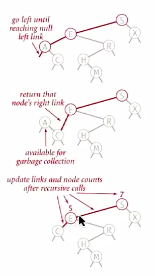


This becomes very inconvenient and will overload memory

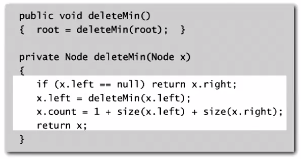
*Deleting a minimum:*

1. Traverse all the way to left until you find null left link
2. Replace that node with right link, then update subtree counts

Delete min visualized:



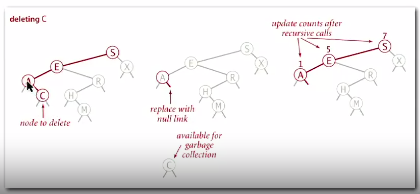
DeleteMin() implementation:



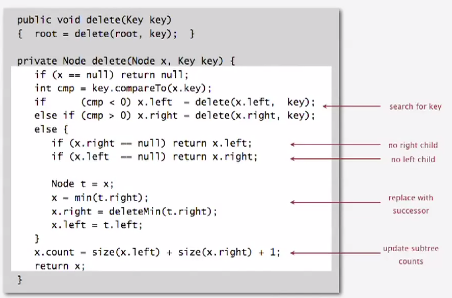
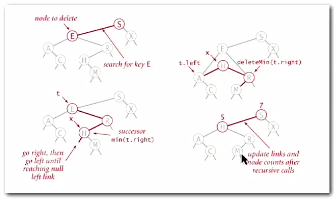
This also works for DeleteMax()

**Hibbard deletion:**

1. Search for node containing key for deletion
2. Options
   1. If node has no children, replace that node’s link with null (update counts)



* 1. If only 1 child, replace node-to-be-del’s link with link to child node
  2. If has 2 children:
     1. Find next smallest node on right subtree of tree (tree of node-to-be-del)
     2. Next smallest node replaces node-to-be-del, connect former smallest node child to former smallest node’s former parent  
        Hibbard deletion implementation  
          
         **ISSUE: Tree becomes asymmetrical!! Height of tree == sqrt of N, not log N (much larger).** Randomly choosing between left and right doesn’t work- still asymmetrical. Long-standing open problem



Deletion increases all other operation times to sqrt of N ☹

Operation runtime comparisons:

