**Binary Search Trees**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **search** | **insert** | **delete** |
| arrayList |  |  |  |
| linkedList |  |  |  |
| BST |  |  |  |

If the data is stored in order (already sorted), what data structure is the most efficient? Be able to explain why.

The Binary Search Tree (BST) data structure keeps its items sorted and also allows O(log n) access to each item. By definition, in a BST every node is greater than its left child and less than its right child. Every comparison thus throws away half the remaining data. We have seen that behavior before when we performed a binary search in an array. That property means BSTs can access any item in O(log n). Keeping the tree in order is also fast, because finding the right place is O(log n), followed by linking and unlinking some nodes.

One problem to keep in mind is that the BST's performance depends on the shape of the tree. Obviously, finding an item in a linear tree is O(n). If the data in a BST changes often, programmers may need to pause and rebuild the tree so that it is balanced. Perhaps they can run the rebalancing program after business hours. Or they could use a self-balancing tree, such as an AVL tree, a red-black tree, or an AA tree.

1. General algorithm for building a BST: Compare the root to the new item. If the item is equal or less, go left, else go right. When you reach the end, insert the item, either left or right. The shape of each tree depends on the order that the data is given to you. Draw the BST formed by the letters in this string: AMERICAN

2. Build a BST from this data: AACEIMNR

3. Build a BST from this data: MAENIRAC

4. Looking at the trees above, and generalizing, what is the best case scenario for building a balanced binary tree? When the data is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. In the best case, each of the n elements will require about log n comparisons, resulting in O(\_\_\_\_\_\_\_ ) time for building a BST. In the worst case, the case of building a linear tree, the resulting Big-O performance is O(\_\_\_\_) time, because there are n elements, but each requires n comparisons.

5. It is easy to find the minimum value in a BST. Hint: look at several BSTs. Where are the min values always located? Use an *iterative* method. Now write the code.

**private** Object min(TreeNode t)

{

6. Now do a similar task *recursively*. Return the maximum value in a BST. Use recursion!

**private** Object max(TreeNode t)

{

7. How would you search a BST for a target? For example, search this tree for "4".

*Iterative* algorithm: create a temporary pointer p at the root. While p is not null, if the p's value equals the target, return *true*. If the target is less than the p's value, go left, otherwise go right. If the target is not found, return *false*.

**private** **boolean** contains(TreeNode t, Comparable target)

{

8. Now search *recursively*. Here is the algorithm: if the tree is empty, return *false*. If the target is less than the current node value, return the left subtree. If the target is greater, return the right subtree. Otherwise, return *true*.

**private boolean** contains(TreeNode t, Comparable target)

{

9. The next task is to build a BST. For example, add a "2" in this tree. Then add a "7". You might come up with this *iterative* algorithm to build a BST: if the root is null, create the new node. Else, create two pointers, p and q. Advance the "front" pointer q-- either left or right--until it is null, at which point the "second" pointer p points to the node at which the new data will be attached. Then attach it, either left or right as needed.

**private** TreeNode add(TreeNode t, Comparable item)

{

10. Now think *recursively* to build a BST. For example, add a "2" in the tree above. Then add a "7". Recursive algorithm: If the node is null, create the new node. Else, if the item is less, set the left node while recurring to the left. Else, if the item is greater, set the right node while recurring to the right.

**private** TreeNode insert(TreeNode t, Comparable item)

{

**Assignment**

Your BST, which stores Strings, will implement this interface:

interface BSTinterface  
 {  
 public int size();

public TreeNode getRoot();  
 public boolean contains(String obj);  
 public void add(String obj); //does not balance  
 //public void addBalanced(String obj); //AVL  
 //public void remove(String obj); //BST with remove  
 //public void removeBalanced(String obj); //extra lab   
 public String min();  
 public String max();  
 public String display();  
 public String toString();  
 }

Your BST should have two private fields and one default constructor. BST\_Driver , which is written for you, tests each method above. Submit BST.java to Codepost.

**Sample Run**

|  |  |  |
| --- | --- | --- |
| Root is TreeNode@5b6f7412 R  N M  I  E  C  A  A Input target: I found: I Min = A Max = R In Order: A A C E I M N R  --------------------- | Root is TreeNode@1e80bfe8  R  N  M  I  E  C A  A Input target: I found: I Min = A Max = R In Order: A A C E I M N R  --------------------- | Root is TreeNode@66a29884  R  N  M  I  E  C A  A Input target: i i not found. Min = A Max = R In Order: A A C E I M N R  --------------------- |

**Extension Labs for BST**

11. Given the inorder ("ABCDEFGHI") and postorder ("ABEIHGFD") of a BST of single letters, display the preorder printing ("?????????"). You are NOT allowed to create the tree. Instead, you must manipulate the strings. (Therefore, this is not actually a tree lab.) In the BST class, implement the following method:

public String preorderExtension(String inorder, String postorder)

**Optional Labs for the Trees Unit**

12. Write a BST to a text file, then re-create the BST by reading back from the file. No shell.

13. Implement a SpellCheck tree. The assignment is in folder “09 SpellCheck Tree.” There is no shell.

14. Implement a Stern-Brocot Tree. The assignment is in folder “10 Stern-Brocot Tree.” There is no shell.