Binary Search Tree (II): Insertion

CSCD 300 – Data Structures

Eastern Washington University

EASTERN WASHINGTON UNIVERSITY
start something big

Goal

We will learn how to insert a new key into an existing binary search tree which could be possibly empty. By using this insert operation, we can construct a binary search tree for a given set of data by inserting the data elements one by one.

Outline

Recall: the BST node class

```
class BST_Node{
   int key; //the key saved in the node. You can have any other fields
            //of any type within a node, depending on your applications.
   BST_Node parent; // reference to the parent node.
   BST_Node left;   // reference to the left child node.
   BST_Node right;  // reference to the right child node.

   BST_Node(int k){  //constructor
      key = k; parent = left = right = null;
   }

   /* supporting operations are here */
}
```

Note

The parent link may not be needed if the set of BST operations in your application do not need the parent links to travel upward inside the tree. However, for general purpose, we include the parent link here.

The binary search tree class

```
class BST{
   BST_Node root;   //the root of the BST

   /* You can add any other member fields needed by your app here */

   /* the constructor */
   BST(){ root = null;}

   /* If the given key k already exists in the BST, return null;
      otherwise, insert k into the BST and return the reference
      to the new node that contains k.
   */
   BST_Node insert(int k){ ... }

   /* Other methods will follow here  */
}
```

Next, let's look at the idea on how to insert a new key into an existing BST and then use that idea to implement the `insert()` member function.

Inserting a new key into a BST

Task

Insert a new key into an existing binary search tree, which might be even empty, so that the resulting tree after the insertion is still a binary search tree.

Key idea

Always insert the new key as a new leaf node, and find the right place to attach this new leaf node.

```
BST_Node BST::insert(int k){
   new_node = new BST_Node(k);

   y = NULL; x = root;
   /* Find the leaf node, where
      the new_node is attached. */
   while(x != null)
      y = x;
      if(k == x.key) //k already exists
          return null;
      else if(k < x.key) x = x.left;
      else x = x.right;

   /* Attach the new node */
   new_node.parent = y;
   if(y == null)  //The tree was empty.
      root = new_node;
   else if(k < y.key)
      y.left = new_node;
   else
      y.right = new_node;

   return new_node;
}
```

The time complexity of the insert operation

Observation

All that we do in the insertion of a new key is a walk from the global root to an existing leaf node. The number of steps in this walk is bounded by the height [a] of the tree. The time cost for the work at each walk step is constant. The time cost for attaching the new node at the end of the walk is also constant. So the total time cost for inserting a new key is bounded by the height of the BST.

[a] The height of the tree is defined as the number of edges in the longest simple path from the global root to a leaf node.

Theorem

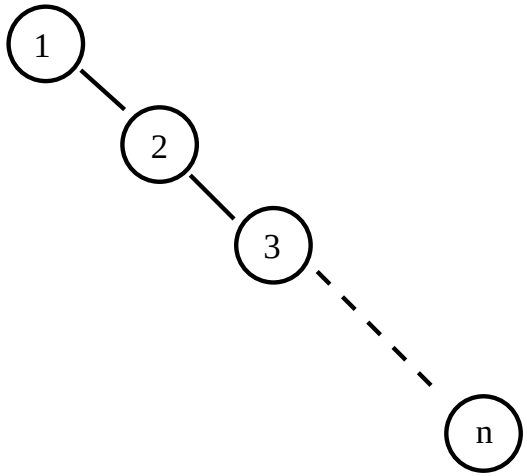
The time cost of inserting a new key into a BST is $O(h)$, where h is the height of the tree.

Question: How large can h be ?

The height of a BST

The worst case

In the worst case, the height of the BST can be as large as $n-1$, where n is the data set size. One example is the BST on the right, which is obtained by inserting the number sequence: $1, 2, \ldots, n$. That means, the `insert` operation can take $O(n)$ time in the worst case.



In practice

However, in practice, a BST constructed from a real-world data set will not have such a skewed shape. If the shape of the BST is quite "balanced" [a], which is true for BSTs constructed from most real-world data sequences, then the height of the BST will be bounded by $O(\log n)$ [b]. That means, the insert operation will take only $O(\log n)$ time, if the tree is quite "balanced".

---

[a] We are not precisely defining the word "balanced" here, but it intuitively means that, if you look at any node in the BST, the number of nodes in the left and right subtrees of that node are not too different.

[b] Why ?

How to maintain the BST "balanced"

Method 1: If people give you the data set in one shot, you can randomly shuffle [a] the data set first by using $O(n)$ time [b], then insert the data elements from the shuffled data set one by one using the `insert` operation that we just discussed.

---

[a] `http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle`

[b] This $O(n)$ extra preprocessing time cost is small amortized over to the n subsequent insert operations.

Method 2: If people give you one new data element only after you've inserted the previous element, Method 1 will not work. There are improved and more complicated versions of BST, which use complicated procedures for inserting a new element and can always guarantee the height of the BST is $O(\log n)$, meaning the tree is guaranteed to be always balanced [a]. We will discuss such balanced BSTs in CSCD320.

---

[a] `https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree`