Binary Search Tree

CS313E - Elements of Software Design

Kia Teymourian

05/11/2022

Agenda

1. Binary Search Trees

2. BST Operation

What is a Binary Search Tree

- A binary search tree is organized as the name is stating in a binary tree structure.
- Each node of the tree is an object that has a key and a payload data (satellite data) and each node contains **3 important attributes or pointers to left node, right node and its parent node**.
- Lef and right nodes are child nodes.

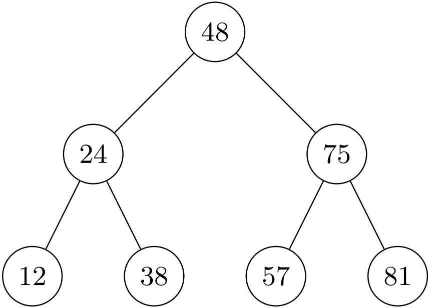

Figure: An Example of a Binary Search Tree.

BST property

- The keys in BST always satisfy the BST property.
- **For any given node in the tree, the keys of the left child node is smaller than the keys of the right child node.**

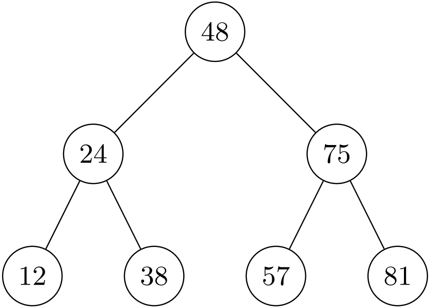

Figure: An Example of a Binary Search Tree.

BST Operations

The search tree data structure supports many dynamic-set operations, including the following operations.

- SEARCH - searches for a given key
- INSERT - Insert a new item to BST
- DELETE - Delete an item from BST
- PREDECESSOR - find the predecessor of a given key
- SUCCESSOR - find the successor of given key
- MINIMUM - find the min of the entire tree
- MAXIMUM - find the max of the entire tree

Print out in Sorted Order

By using the BST property, we can have print out of all tree keys in a sorted order by a simple recursive algorithm.

Algorithm 1 INORDER-TREE-WALK(x)

if $x \neq NIL$ **then**
    INORDER-TREE-WALK $(x.left)$
    print $x.key$
    INORDER-TREE-WALK $(x.right)$
end if

Print out in Sorted Order

By using the BST property, we can have print out of all tree keys in a sorted order by a simple recursive algorithm.

Algorithm 2 INORDER-TREE-WALK(x)

```
if x ≠ NIL then
    INORDER-TREE-WALK (x.left)
    print x.key
    INORDER-TREE-WALK (x.right)
end if
```

What is the run time of the INORDER-TREE-WALK algorithm?

Print out in Sorted Order

By using the BST property, we can have print out of all tree keys in a sorted order by a simple recursive algorithm.

Algorithm 3 INORDER-TREE-WALK(x)

```
if x ≠ NIL then
    INORDER-TREE-WALK (x.left)
    print x.key
    INORDER-TREE-WALK (x.right)
end if
```

What is the run time of the INORDER-TREE-WALK algorithm?
$\boldsymbol{\Theta(n)}$

Print out in Sorted Order

By using the BST property, we can have print out of all tree keys in a sorted order by a simple recursive algorithm.

Algorithm 4 INORDER-TREE-WALK(x)

```
if x ≠ NIL then
    INORDER-TREE-WALK (x.left)
    print x.key
    INORDER-TREE-WALK (x.right)
end if
```

What is the run time of the INORDER-TREE-WALK algorithm?
$\boldsymbol{\Theta(n)}$

- $T(n) \le T(k) + T(n-k+1) + d$
- Assume that node x has a left subtree with k nodes.
- For some constant $d > 0$

Note: See CLRS Theorem 12.1.

BST Search

Algorithm 5 TREE-SEARCH(x, k)

```
if x == NIL or k == x.key then
    return x
end if
if x < x.key then
    return TREE-SEARCH(x.left, k)
else
    return TREE-SEARCH(x.right, k)
end if
```

BST Search

Algorithm 6 TREE-SEARCH(x, k)

---

if $x == NIL$ or $k == x.key$ **then**
    return x
end if
if $x < x.key$ **then**
    return TREE-SEARCH($x.left$, k)
else
    return TREE-SEARCH($x.right$, k)
end if

- ▷ The running time of TREE-SEARCH is $O(h)$, where h is the height of the tree, or $O(log(n))$
- ▷ In worst case $O(n)$

Iterative BST search

We can rewrite the recursive BST search algorithm in an iterative form by "Opening-up and unrolling" the recursion into a while loop.

Algorithm 7 ITERATIVE-SEARCH(x, k)

```
while (x ≠ NIL and k ≠ x.key) do
    if (k < x.key) then
        x = x.left
    else
        x = x.right
    end if
end while
return x
```

- The running time of TREE-SEARCH is $O(h)$, where h is the height of the tree, or $O(log(n))$
- In worst case $O(n)$

Insertion in a BST

The insertion operation may cause that the BST need to reorganize itself to satisfy the BST property.
Givens for the insertion operation:

- A binary tree T
- A node z for which $z.key = v$, $z.left = NIL$ and $z.right = NIL$

Insertion in a BST

Algorithm 8 INSET-SEARCH(T, z)

```
y = NIL                          ▷ trailing pointer y as the parent of x .
x = T.root                       ▷ Take the root of the given Tree T as x
while (x ≠ NIL) do               ▷ As long as x is not null do ...
    y = x
    if (z.key < x.key) then
        x = x.left
    else
        x = x.right
    end if
end while
z.p = y      ▷ z.p is the parent of the z. Here we found a parent for the z to insert
if (y == NIL) then
    T.root = z                   ▷ Our Tree T was an empty tree
else if (z.key < y.key) then
    y.left = z                   ▷ Insert to left
else
    y.right = z                  ▷ Insert to right
end if
```

Deletion of a key from BST

The goal is to delete a given key from BST so that the BST property holds after removal.

3 cases are possible:

- **Case 1 - z has no children.** We remove it by modifying its parent to replace z with NIL as its child.
- **Case 2 - z has only one child.** We would then elevate its childe to take z's position in the tree by modifying z's parent to replace z's child with z.
- **Case 3 - z has two children.** We need to find the z's successor y. In this case y, has to tbe in the z's right subtree so that we can swap position of z with y and remove z. This case is a bit tricky and 4 further sub-cases are possible.

Deletion of a key from BST

Sub-cases of Case-3:

- **Case 3.1. z has no left child.** We replace z by its right child r.
- **Case 3.2. z has no right child.** We replace z by its left child l.
- **Case 3.3. z has two children and right sucessor y has no left child.** z has two children l and y. y has no left child and its right child is z. We remove z and replace it with y
- **Case 3.4. z has two children left child l and right child r, and r has a left child y.** We replace y by its right child x, and we set y to be r's parent. Then we can remove z and replace it with y, and make left child of y be l.

Deletion operation is based on the above 4 cases.

Case 3.1. z has no left child.

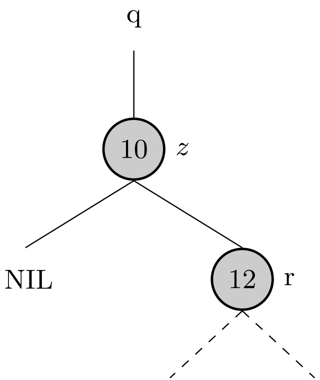

Figure: BST Delete Operation Case 3.1 - Delete node z (10)

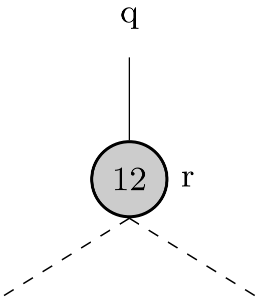

Figure: BST **After** BST Delete Operation Case 3.1

Case 3.2. z has no right child

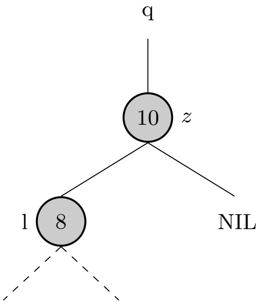

Figure: **Before** - Delete Operation Case 3.2

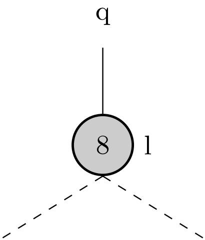

Figure: **After** - Delete operation

Case 3.3. z has two children and right sucessor y has no left child

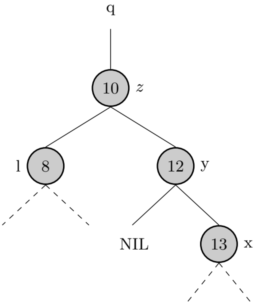

Figure: **Before** - Delete Operation Case 3.3

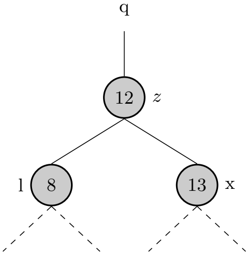

Figure: **After** - Delete operation

Case 3.4. z has two children left child l and right child r, and r has a left child y

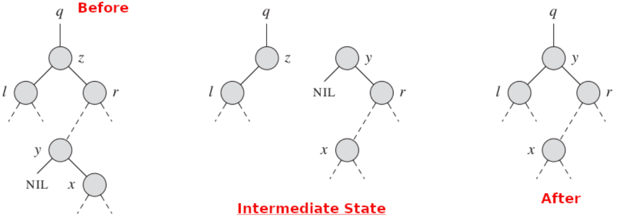

Figure: BST Delete operation Case 3.4

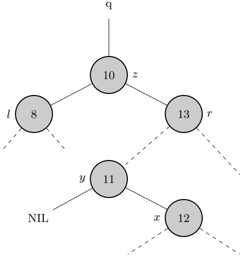

Figure: **Before** - Delete Operation Case 3.4

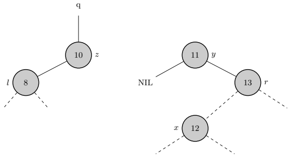

Figure: **Intermediate State**

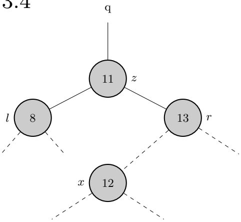

Figure: **After** - Delete Operation

Moving Subtrees

To be able to move subtrees around within the binary search tree, we use a subroutine named TRANSPLANT which replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent then becomes node v's parent, and u's parent will be having u as its child.

Algorithm 9 TRANSPLANT(T, u, v)

$\triangleright$ Tree T, replate subtree u with v

if $u.p == NIL$ **then** $\triangleright$ (If u is the root, replace it with v)

  $T.root = v$

else if $u == u.p.left$ **then** $\triangleright$ If u is the left subtree, replace the left

  $u.p.left = v$

else

  $u.p.right = v$ $\triangleright$ If u is the right subtree, replace the right

end if

if $v \neq NIL$ **then** $\triangleright$ We allow v to be null, if not null set its parent to the parent of u

  $v.p = u.p$

end if

TREE-DELETE(T, z)

Algorithm 10 TREE-DELETE(T, z)

```
                                                ▷ Delete z from Tree T
if z.left == NIL then                           ▷ if we have case 3.1
    TRANSPLANT (T, z, z.right)
else if z.right == NIL then
    TRANSPLANT (T, z, z.left)                   ▷ if we have case 3.2
else
    y = TREE − MINIMUM(z.right)                 ▷ if we have case 3.3 and 3.4
    if y.p ≠ z then
        TRANSPLANT (T, y, y.right)
        y.right = z.right
        y.right.p = y
    end if
    TRANSPLANT (T, z, y)
    y.left = z.left
    y.left.p = y
end if
```

Balacing Tree

A BST can get unbalanced.

Table: BST Run Times

Operation	**Average**	**Worst Case**
Search	$O(log(n))$	$O(n)$
Insert	$O(log(n))$	$O(n)$
Deletion	$O(log(n))$	$O(n)$
Space	$O(n)$	$O(n)$

Can we do it better?

Yes, read more about and AVL-Trees

- **Red**-**Black**-Trees (Search, insert and delete in worst case in $O(log(n))$)
- **AVL-Trees** (Search, insert and delete in worst case in $O(log(n))$)

Additional Examples - 1

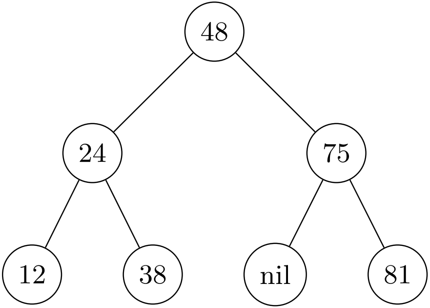

Figure: Insert 90 in the above BST!

What is the output result of the above BST when you insert 90 ?

Additional Examples - 1 Solution

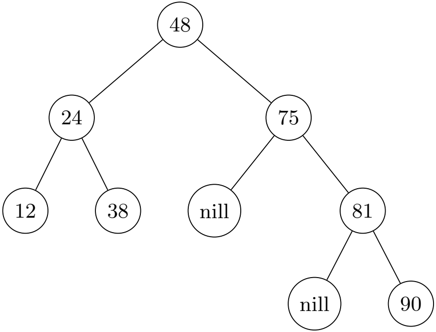

Figure: BST after inserting 90

Additional Examples - 2

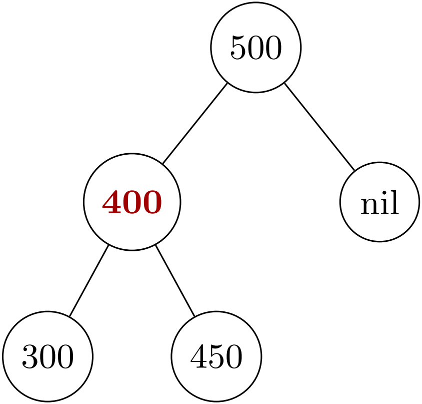

Figure: Remove 400 in the above BST!

What is the output result of the above BST when you remove 400 ?

Additional Examples - 2 Solution

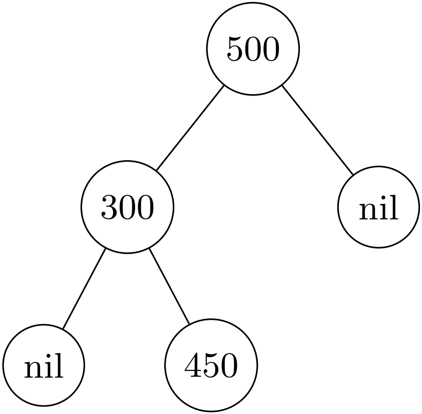

Figure: Resulting BST

Additional Examples - 3

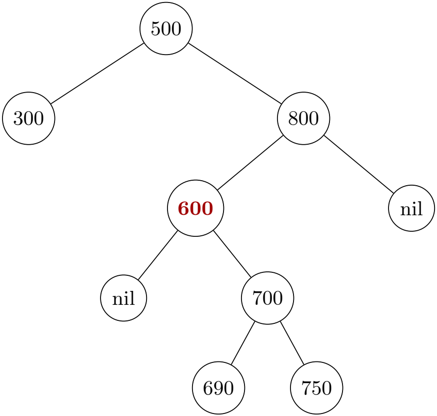

Figure: Remove 600 in the above BST!

What is the output result of the above BST when you remove **400** ?

Additional Examples - 3 Solution

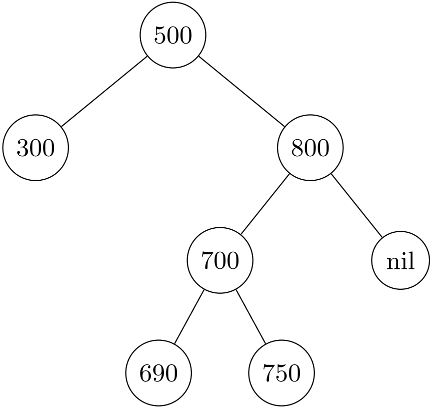

Figure: Resulting BST

Binary Search Tree - Vizualization Tools

- https://www.cs.usfca.edu/~galles/visualization/BST.html
- https://visualgo.net/bn/bst
- http://btv.melezinek.cz/binary-search-tree.html
- https://yongdanielliang.github.io/animation/web/BST.html

Readings from CLRS Book (Introduction to Algorithms, 3rd Edition)

- Chapter 12 Binary Search Trees
- Section 12.1 What is a binary search tree?
- Section 12.2 Querying a binary search tree
- Section 12.3 Insertion and deletion