Binary Search Tree

mardi 25 juin 2024 10:34 AM

TABLE OF CONTENTS

INTRODUCTION

Binary search trees (BSTs) are very simple to understand. We start with a root node with value x, where the left subtree of x contains nodes with values $< x$ and the right subtree contains nodes whose values are $\geq x$. Each node follows the same rules with respect to nodes in their left and right subtrees.

BSTs are of interest because they have operations which are favourably fast: insertion, look up, and deletion can all be done in $O(log\ n)$ time. It is important to note that the $O(log\ n)$ times for these operations can only be attained if the BST is reasonably balanced; for a tree data structure with self balancing properties see AVL tree defined in §7).

INSERTION

```
algorithm Insert(value)
    Pre: value has passed custom type checks for type T
    Post: value has been placed in the correct location in the tree
    if root = ∅
        root ← node(value)
    else
        InsertNode(root, value)
    end if
end Insert
```

```
algorithm InsertNode(current, value)
    Pre: current is the node to start from
    Post: value has been placed in the correct location in the tree
    if value < current.Value
        if current.Left = ∅
            current.Left ← node(value)
        else
            InsertNode(current.Left, value)
        end if
    else
        if current.Right = ∅
            current.Right ← node(value)
        else
            InsertNode(current.Right, value)
        end if
    end if
end InsertNode
```

The insertion algorithm is split for a good reason. The first algorithm (non-recursive) checks a very core base case - whether or not the tree is empty. If the tree is empty then we simply create our root node and finish. In all other cases we invoke the recursive $InsertNode$ algorithm which simply guides us to the first appropriate place in the tree to put $value$. Note that at each stage we perform a binary chop: we either choose to recurse into the left subtree or the right by comparing the new value with that of the current node. For any totally ordered type, no value can simultaneously satisfy the conditions to place it in both subtrees.

Searching

We have talked previously about insertion, we go either left or right with the right subtree containing values that are $\geq x$ where x is the value of the node we are inserting. When searching the rules are made a little more atomic and at any one time we have four cases to consider:

1. the $root = \emptyset$ in which case $value$ is not in the BST; or
2. $root$.Value $= value$ in which case $value$ is in the BST; or
3. $value < root$.Value, we must inspect the left subtree of $root$ for $value$; or
4. $value > root$.Value, we must inspect the right subtree of $root$ for $value$.

```
algorithm Contains(root, value)
    Pre: root is the root node of the tree, value is what we would like to locate
    Post: value is either located or not
    if root = ∅
      return false
    end if
    if root.Value = value
      return true
    else if value < root.Value
      return Contains(root.Left, value)
    else
      return Contains(root.Right, value)
    end if
end Contains
```

Deletion

Removing a node from a BST is fairly straightforward, with four cases to consider:

1. the value to remove is a leaf node; or
2. the value to remove has a right subtree, but no left subtree; or
3. the value to remove has a left subtree, but no right subtree; or
4. the value to remove has both a left and right subtree in which case we promote the largest value in the left subtree.

There is also an implicit fifth case whereby the node to be removed is the only node in the tree. This case is already covered by the first, but should be noted as a possibility nonetheless.

Of course in a BST a value may occur more than once. In such a case the first occurrence of that value in the BST will be removed.

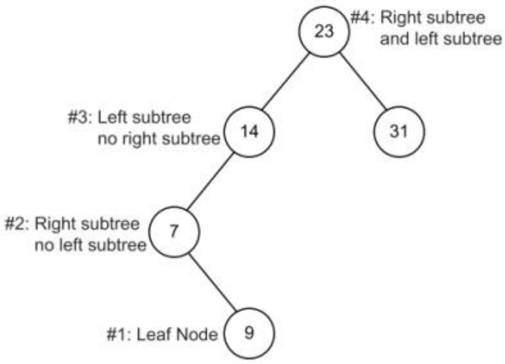


Figure 3.2: binary search tree deletion cases

```
algorithm FindNode(root, value)
    Pre: value is the value of the node we want to find the parent of
         root is the root node of the BST
    Post: a reference to the node of value if found; otherwise ∅
    if root = ∅
      return ∅
    end if
    if root.Value = value
      return root
    else if value < root.Value
      return FindNode(root.Left, value)
    else
      return FindNode(root.Right, value)
    end if
end FindNode
```

```
algorithm FindParent(value, root)
    Pre: value is the value of the node we want to find the parent of
         root is the root node of the BST and is != ∅
    Post: a reference to the parent node of value if found; otherwise ∅
    if value = root.Value
      return ∅
    end if
    if value < root.Value
      if root.Left = ∅
        return ∅
      else if root.Left.Value = value
        return root
      else
        return FindParent(value, root.Left)
      end if
    else
      if root.Right = ∅
        return ∅
      else if root.Right.Value = value
        return root
      else
        return FindParent(value, root.Right)
      end if
    end if
end FindParent
```

Missing root parameter in remove function, because we will need it in FindNode

```
algorithm Remove(value)
    Pre: value is the value of the node to remove, root is the root node of the BST
         Count is the number of items in the BST
    Post: node with value is removed if found in which case yields true, otherwise false
    nodeToRemove ← FindNode(value)
    if nodeToRemove = ∅
      return false // value not in BST
    end if
    parent ← FindParent(value)
    if Count = 1
      root ← ∅ // we are removing the only node in the BST
    else if nodeToRemove.Left = ∅ and nodeToRemove.Right = null
      // case #1
      if nodeToRemove.Value < parent.Value
        parent.Left ← ∅
      else
        parent.Right ← ∅
      end if
    else if nodeToRemove.Left = ∅ and nodeToRemove.Right ≠ ∅
      // case # 2
      if nodeToRemove.Value < parent.Value
        parent.Left ← nodeToRemove.Right
      else
        parent.Right ← nodeToRemove.Right
      end if
    else if nodeToRemove.Left ≠ ∅ and nodeToRemove.Right = ∅
      // case #3
      if nodeToRemove.Value < parent.Value
        parent.Left ← nodeToRemove.Left
      else
        parent.Right ← nodeToRemove.Left
      end if
    else
      // case #4
      largestValue ← nodeToRemove.Left
      while largestValue.Right ≠ ∅
        // find the largest value in the left subtree of nodeToRemove
        largestValue ← largestValue.Right
      end while
      // set the parents' Right pointer of largestValue to ∅
      FindParent(largestValue.Value).Right ← ∅
      nodeToRemove.Value ← largestValue.Value
    end if
    Count ← Count −1
    return true
end Remove
```

Astute readers will have noticed that the *FindNode* algorithm is exactly the same as the *Contains* algorithm (defined in §3.2) with the modification that we are returning a reference to a node not *true* or *false*. Given *FindNode*, the easiest way of implementing *Contains* is to call *FindNode* and compare the return value with $\emptyset$.

Finding the smallest and largest values

```
algorithm FindMin(root)
    Pre: root is the root node of the BST
         root ≠ ∅
    Post: the smallest value in the BST is located
    if root.Left = ∅
      return root.Value
    end if
    FindMin(root.Left)
end FindMin
```

```
algorithm FindMax(root)
    Pre: root is the root node of the BST
         root ≠ ∅
    Post: the largest value in the BST is located
    if root.Right = ∅
      return root.Value
    end if
    FindMax(root.Right)
end FindMax
```

Tree Traversals

Preorder

Pre-order Traversal is done by **visiting the root node first**, then **recursively do a pre-order traversal** **of the left subtree**, **followed by** a r**ecursive pre-order traversal** **of the right subtree**. It's used for creating a copy of the tree, prefix notation of an expression tree, etc.

When using the preorder algorithm, you visit the root first, then traverse the left subtree and finally traverse the right subtree. An example of preorder traversal is shown in Figure 3.3.

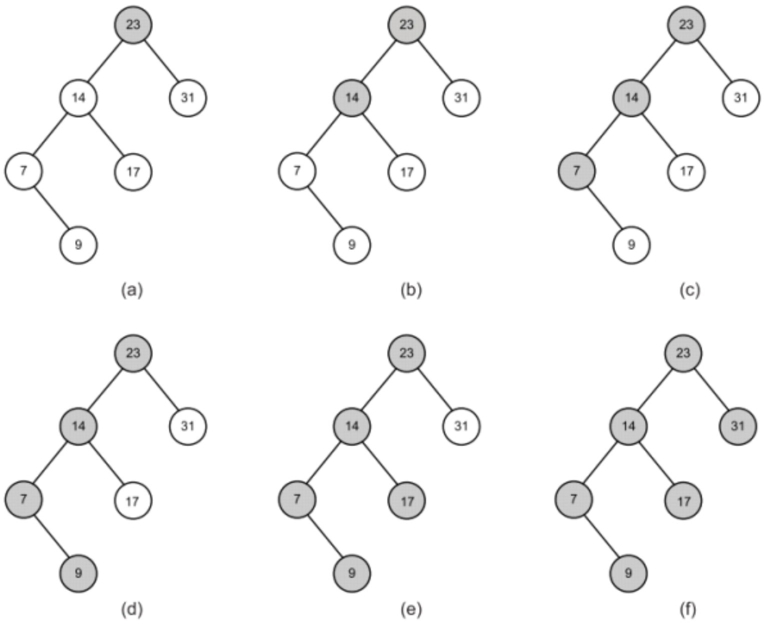


Figure 3.3: Preorder visit binary search tree example

```
algorithm Preorder(root)
    Pre: root is the root node of the BST
    Post: the nodes in the BST have been visited in preorder
    if root ≠ ∅
        yield root.Value
        Preorder(root.Left)
        Preorder(root.Right)
    end if
end Preorder
```

Postorder

the **value of the node is yielded** **after** traversing both subtrees.

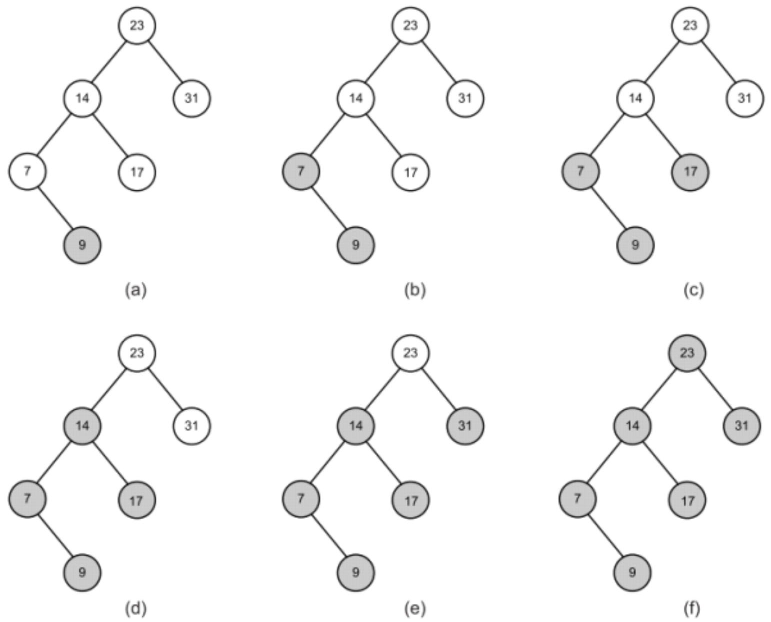
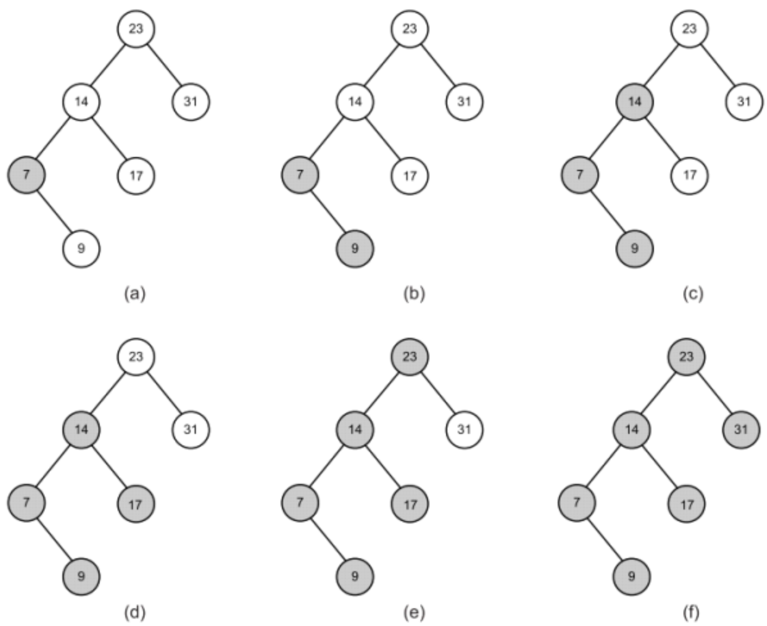


Figure 3.4: Postorder visit binary search tree example

```
algorithm Postorder(root)
   Pre: root is the root node of the BST
   Post: the nodes in the BST have been visited in postorder
   if root ≠ ∅
      Postorder(root.Left)
      Postorder(root.Right)
      yield root.Value
   end if
end Postorder
```

Inorder

the **value of the <u>current node</u> is yielded in between** traversing **the left subtree** and **the right subtree.**



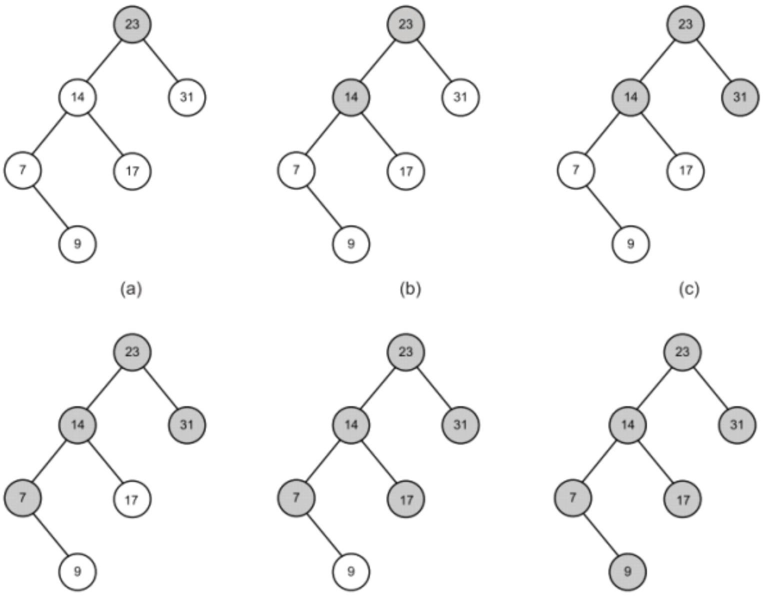
```
algorithm Inorder(root)
    Pre: root is the root node of the BST
    Post: the nodes in the BST have been visited in inorder
    if root ≠ ∅
        Inorder(root.Left)
        yield root.Value
        Inorder(root.Right)
    end if
end Inorder
```

One of the beauties of inorder traversal is that values are yielded in their comparison order. In other words, when traversing a populated BST with the inorder strategy, the yielded sequence would have property $x_i \leq x_{i+1} \forall i$.

Breadth First

Traversing a tree in breadth flrst order **yields the values of** **all nodes of a particular depth** **in the tree** **before any deeper ones.** In other words, given a depth **d** we would visit the values of all nodes at **d in a left to right** fashion, then we would proceed to **d + 1** and so on until we hade no more nodes to visit.

Traditionally breadth first traversal **is implemented using a list** (vector, resizeable array, etc) **to store the values of the nodes visited** in breadth flrst order and then a **queue** to store those **nodes that have yet to be visited**



```
algorithm BreadthFirst(root)
    Pre: root is the root node of the BST
    Post: the nodes in the BST have been visited in breadth first order
    q ← queue
    while root ≠ ∅
        yield root.Value
        if root.Left ≠ ∅
            q.Enqueue(root.Left)
        end if
        if root.Right ≠ ∅
            q.Enqueue(root.Right)
        end if
        if !q.IsEmpty()
            root ← q.Dequeue()
        else
            root ← ∅
        end if
    end while
end BreadthFirst
```

Summary

A binary search tree is a good solution when you need to represent types that are ordered according to some custom rules inherent to that type. With logarithmic insertion, lookup, and deletion it is very effecient. Traversal remains linear, but there are many ways in which you can visit the nodes of a tree. Trees are recursive data structures, so typically you will find that many algorithms that operate on a tree are recursive.

The run times presented in this chapter are based on a pretty big assumption - that the binary search tree's left and right subtrees are reasonably balanced. We can only attain logarithmic run times for the algorithms presented earlier when this is true. A binary search tree does not enforce such a property, and the run times for these operations on a pathologically unbalanced tree become linear: such a tree is effectively just a linked list. Later in §7 we will examine an AVL tree that enforces self-balancing properties to help attain logarithmic run times.