Chapter 3

Binary Search Tree

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).

In the following examples you can assume, unless used as a parameter alias that *root* is a reference to the root node of the tree.

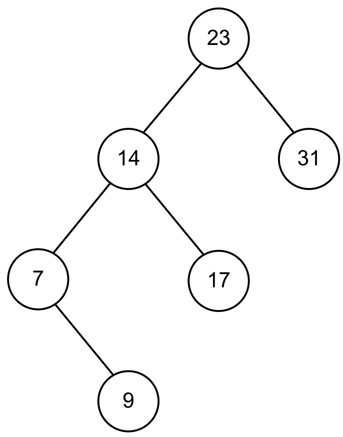

Figure 3.1: Simple unbalanced binary search tree

3.1 Insertion

As mentioned previously insertion is an $O(log\ n)$ operation provided that the tree is moderately balanced.

```
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.

3.2 Searching

Searching a BST is even simpler than insertion. The pseudocode is self-explanatory but we will look briefly at the premise of the algorithm nonetheless.

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
```

3.3 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.

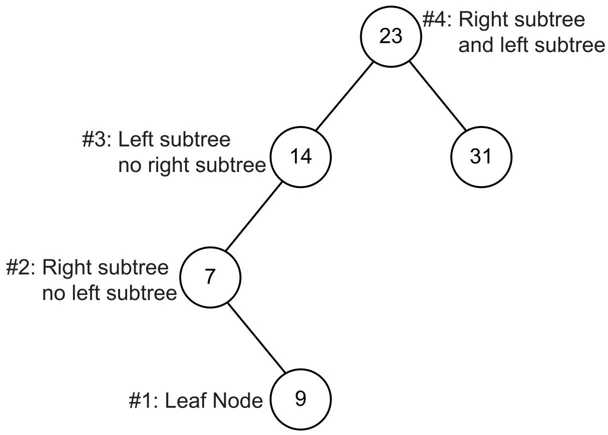

Figure 3.2: binary search tree deletion cases

The *Remove* algorithm given below relies on two further helper algorithms named *FindParent*, and *FindNode* which are described in §3.4 and §3.5 respectively.

```
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
```

3.4 Finding the parent of a given node

The purpose of this algorithm is simple - to return a reference (or pointer) to the parent node of the one with the given value. We have found that such an algorithm is very useful, especially when performing extensive tree transformations.

```
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
```

A special case in the above algorithm is when the specified value does not exist in the BST, in which case we return ∅. Callers to this algorithm must take account of this possibility unless they are already certain that a node with the specified value exists.

3.5 Attaining a reference to a node

This algorithm is very similar to §3.4, but instead of returning a reference to the parent of the node with the specified value, it returns a reference to the node itself. Again, ∅ is returned if the value isn't found.

```
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
```

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 $\emptyset$.

3.6 Finding the smallest and largest values in the binary search tree

To find the smallest value in a BST you simply traverse the nodes in the left subtree of the BST always going left upon each encounter with a node, terminating when you find a node with no left subtree. The opposite is the case when finding the largest value in the BST. Both algorithms are incredibly simple, and are listed simply for completeness.

The base case in both *FindMin*, and *FindMax* algorithms is when the Left (*FindMin*), or Right (*FindMax*) node references are $\emptyset$ in which case we have reached the last node.

```
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
```

3.7 Tree Traversals

There are various strategies which can be employed to traverse the items in a tree; the choice of strategy depends on which node visitation order you require. In this section we will touch on the traversals that DSA provides on all data structures that derive from *BinarySearchTree*.

3.7.1 Preorder

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.

```
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
```

3.7.2 Postorder

This algorithm is very similar to that described in §3.7.1, however the value of the node is yielded after traversing both subtrees. An example of postorder traversal is shown in Figure 3.4.

```
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
```

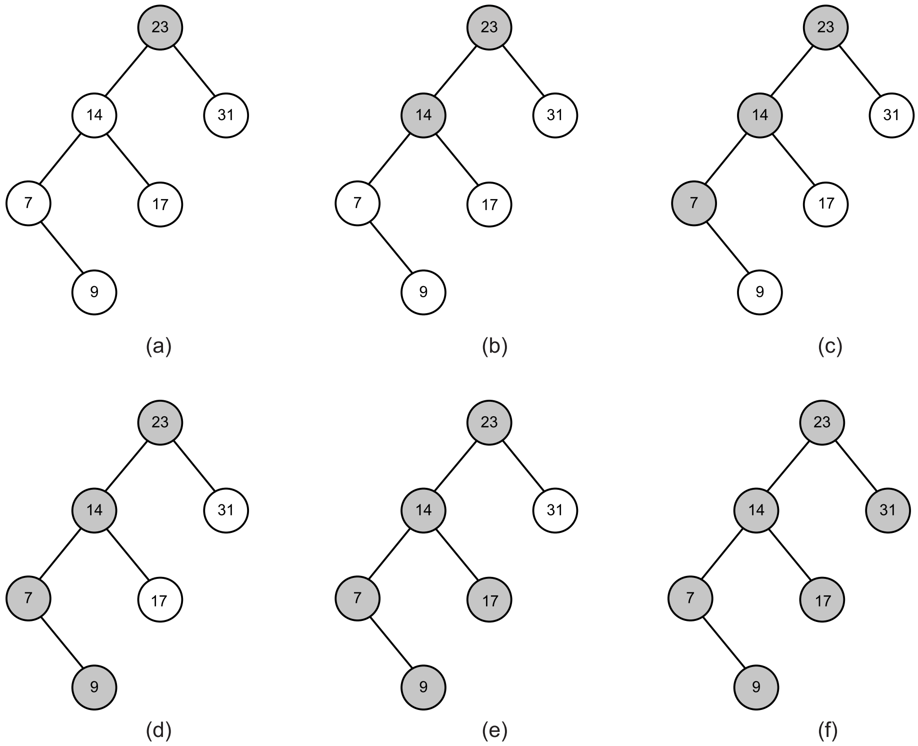

Figure 3.3: Preorder visit binary search tree example

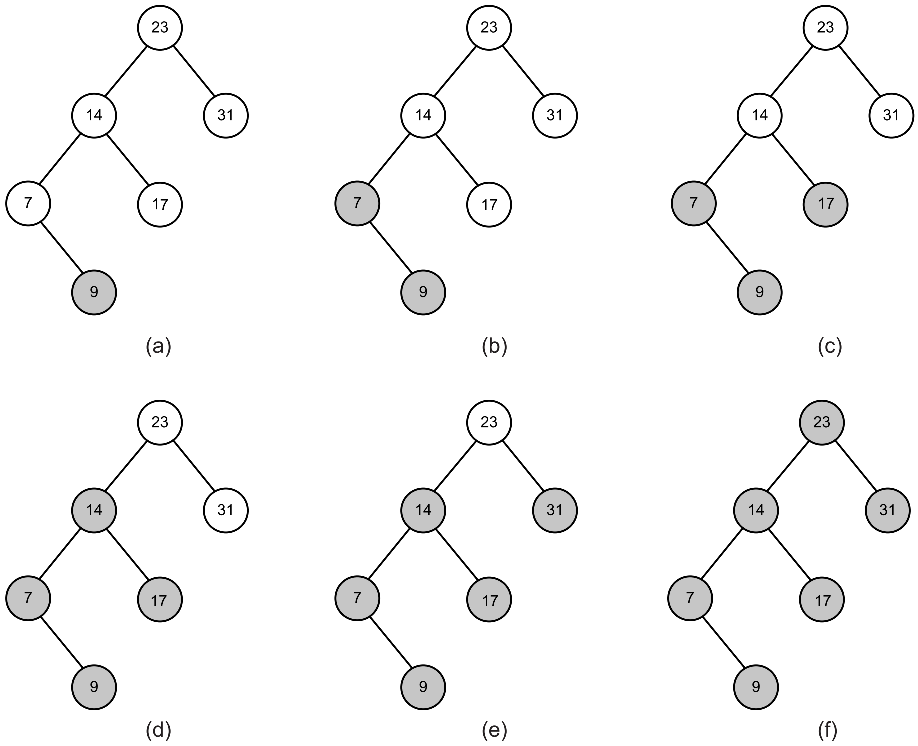

Figure 3.4: Postorder visit binary search tree example

3.7.3 Inorder

Another variation of the algorithms defined in §3.7.1 and §3.7.2 is that of inorder traversal where the value of the current node is yielded in between traversing the left subtree and the right subtree. An example of inorder traversal is shown in Figure 3.5.

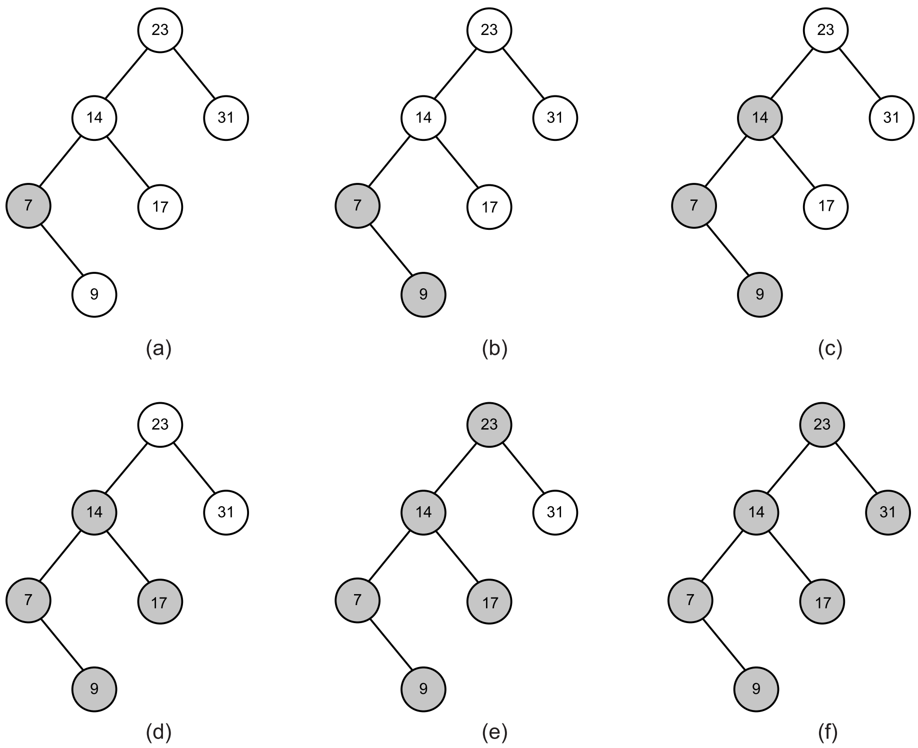

Figure 3.5: Inorder visit binary search tree example

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* $\neq \emptyset$
        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 $x_i \leq x_{i+1} \forall i$.

3.7.4 Breadth First

Traversing a tree in breadth first 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. An example of breadth first traversal is shown in Figure 3.6.

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

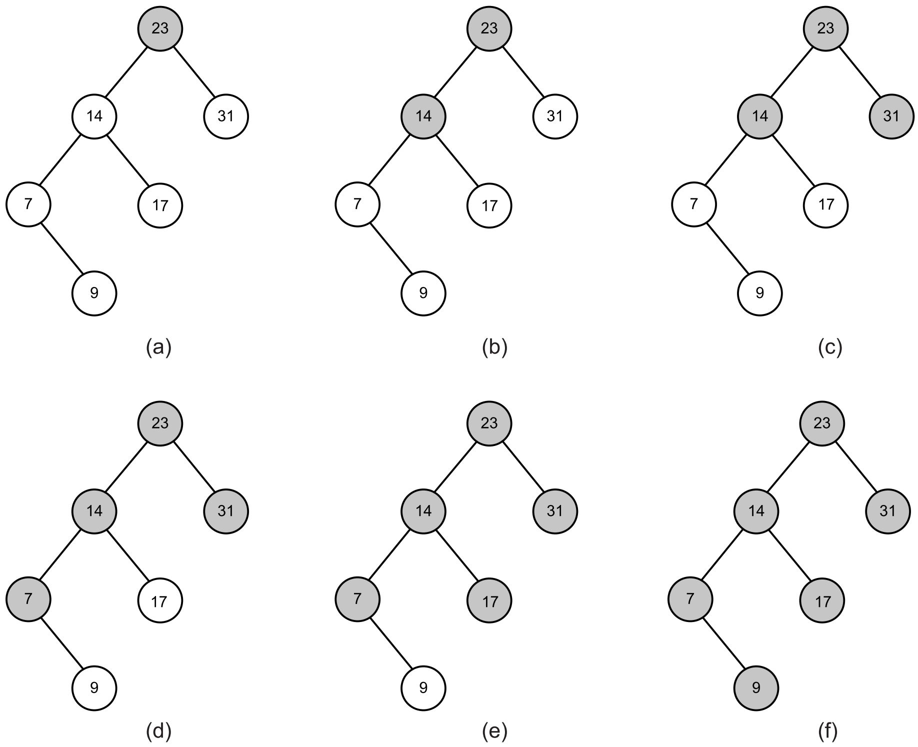

Figure 3.6: Breadth First visit binary search tree example

```
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
```

3.8 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.