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 ≥ 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. 23 14 31 7 17 9 Figure 3.1: Simple unbalanced binary search tree 19 CHAPTER 3. BINARY SEARCH TREE 20 3.1 Insertion As mentioned previously insertion is an O(log n) operation provided that the tree is moderately balanced. 1) algorithm Insert(value) 2) Pre: value has passed custom type checks for type T 3) Post: value has been placed in the correct location in the tree 4) if root = ∅ 5) root ← node(value) 6) else 7) InsertNode(root, value) 8) end if 9) end Insert 1) algorithm InsertNode(current, value) 2) Pre: current is the node to start from 3) Post: value has been placed in the correct location in the tree 4) if value < current.Value 5) if current.Left = ∅ 6) current.Left ← node(value) 7) else 8) InsertNode(current.Left, value) 9) end if 10) else 11) if current.Right = ∅ 12) current.Right ← node(value) 13) else 14) InsertNode(current.Right, value) 15) end if 16) end if 17) end InsertNode The insertion algorithm is split for a good reason. The first algorithm (nonrecursive) 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. CHAPTER 3. BINARY SEARCH TREE 21 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 ≥ 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 = ∅ 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. 1) algorithm Contains(root, value) 2) Pre: root is the root node of the tree, value is what we would like to locate 3) Post: value is either located or not 4) if root = ∅ 5) return false 6) end if 7) if root.Value = value 8) return true 9) else if value < root.Value 10) return Contains(root.Left, value) 11) else 12) return Contains(root.Right, value) 13) end if 14) end Contains CHAPTER 3. BINARY SEARCH TREE 22 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. 23 14 31 7 #1: Leaf Node 9 #2: Right subtree no left subtree #3: Left subtree no right subtree #4: Right subtree and left subtree Figure 3.2: binary search tree deletion cases The Remove algorithm given below relies on two further helper algorithms named F indP arent, and F indNode which are described in §3.4 and §3.5 respectively. CHAPTER 3. BINARY SEARCH TREE 23 1) algorithm Remove(value) 2) Pre: value is the value of the node to remove, root is the root node of the BST 3) Count is the number of items in the BST 3) Post: node with value is removed if found in which case yields true, otherwise false 4) nodeT oRemove ← FindNode(value) 5) if nodeT oRemove = ∅ 6) return false // value not in BST 7) end if 8) parent ← FindParent(value) 9) if Count = 1 10) root ← ∅ // we are removing the only node in the BST 11) else if nodeT oRemove.Left = ∅ and nodeT oRemove.Right = null 12) // case #1 13) if nodeT oRemove.Value < parent.Value 14) parent.Left ← ∅ 15) else 16) parent.Right ← ∅ 17) end if 18) else if nodeT oRemove.Left = ∅ and nodeT oRemove.Right 6= ∅ 19) // case # 2 20) if nodeT oRemove.Value < parent.Value 21) parent.Left ← nodeT oRemove.Right 22) else 23) parent.Right ← nodeT oRemove.Right 24) end if 25) else if nodeT oRemove.Left 6= ∅ and nodeT oRemove.Right = ∅ 26) // case #3 27) if nodeT oRemove.Value < parent.Value 28) parent.Left ← nodeT oRemove.Left 29) else 30) parent.Right ← nodeT oRemove.Left 31) end if 32) else 33) // case #4 34) largestV alue ← nodeT oRemove.Left 35) while largestV alue.Right 6= ∅ 36) // find the largest value in the left subtree of nodeT oRemove 37) largestV alue ← largestV alue.Right 38) end while 39) // set the parents' Right pointer of largestV alue to ∅ 40) FindParent(largestV alue.Value).Right ← ∅ 41) nodeT oRemove.Value ← largestV alue.Value 42) end if 43) Count ← Count −1 44) return true 45) end Remove CHAPTER 3. BINARY SEARCH TREE 24 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. 1) algorithm FindParent(value, root) 2) Pre: value is the value of the node we want to find the parent of 3) root is the root node of the BST and is ! = ∅ 4) Post: a reference to the parent node of value if found; otherwise ∅ 5) if value = root.Value 6) return ∅ 7) end if 8) if value < root.Value 9) if root.Left = ∅ 10) return ∅ 11) else if root.Left.Value = value 12) return root 13) else 14) return FindParent(value, root.Left) 15) end if 16) else 17) if root.Right = ∅ 18) return ∅ 19) else if root.Right.Value = value 20) return root 21) else 22) return FindParent(value, root.Right) 23) end if 24) end if 25) 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. CHAPTER 3. BINARY SEARCH TREE 25 1) algorithm FindNode(root, value) 2) Pre: value is the value of the node we want to find the parent of 3) root is the root node of the BST 4) Post: a reference to the node of value if found; otherwise ∅ 5) if root = ∅ 6) return ∅ 7) end if 8) if root.Value = value 9) return root 10) else if value < root.Value 11) return FindNode(root.Left, value) 12) else 13) return FindNode(root.Right, value) 14) end if 15) 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 f alse. Given FindNode, the easiest way of implementing Contains is to call

FindNode and compare the return value with ∅. 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 F indM in, and F indM ax algorithms is when the Left (F indM in), or Right (F indM ax) node references are ∅ in which case we have reached the last node. 1) algorithm FindMin(root) 2) Pre: root is the root node of the BST 3) root 6= ∅ 4) Post: the smallest value in the BST is located 5) if root.Left = ∅ 6) return root.Value 7) end if 8) FindMin(root.Left) 9) end FindMin CHAPTER 3. BINARY SEARCH TREE 26 1) algorithm FindMax(root) 2) Pre: root is the root node of the BST 3) root 6= ∅ 4) Post: the largest value in the BST is located 5) if root.Right = ∅ 6) return root.Value 7) end if 8) FindMax(root.Right) 9) 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 BinarySearchT ree. 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. 1) algorithm Preorder(root) 2) Pre: root is the root node of the BST 3) Post: the nodes in the BST have been visited in preorder 4) if root 6= ∅ 5) yield root.Value 6) Preorder(root.Left) 7) Preorder(root.Right) 8) end if 9) 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. 1) algorithm Postorder(root) 2) Pre: root is the root node of the BST 3) Post: the nodes in the BST have been visited in postorder 4) if root 6= ∅ 5) Postorder(root.Left) 6) Postorder(root.Right) 7) yield root.Value 8) end if 9) end Postorder CHAPTER 3. BINARY SEARCH TREE 27 23 14 31 7 17 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 (a) (b) (c) (d) (e) (f) 17 17 17 17 17 Figure 3.3: Preorder visit binary search tree example CHAPTER 3. BINARY SEARCH TREE 28 23 14 31 7 17 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 (a) (b) (c) (d) (e) (f) 17 17 17 17 17 Figure 3.4: Postorder visit binary search tree example CHAPTER 3. BINARY SEARCH TREE 29 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. 23 14 31 7 17 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 (a) (b) (c) (d) (e) (f) 17 17 17 17 17 Figure 3.5: Inorder visit binary search tree example 1) algorithm Inorder(root) 2) Pre: root is the root node of the BST 3) Post: the nodes in the BST have been visited in inorder 4) if root 6= ∅ 5) Inorder(root.Left) 6) yield root.Value 7) Inorder(root.Right) 8) end if 9) 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 xi ≤ xi+1∀i. CHAPTER 3. BINARY SEARCH TREE 30 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. 23 14 31 7 17 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 23 14 31 7 9 (a) (b) (c) (d) (e) (f) 17 17 17 17 17 Figure 3.6: Breadth First visit binary search tree example CHAPTER 3. BINARY SEARCH TREE 31 1) algorithm BreadthFirst(root) 2) Pre: root is the root node of the BST 3) Post: the nodes in the BST have been visited in breadth first order 4) q ← queue 5) while root 6= ∅ 6) yield root.Value 7) if root.Left 6= ∅ 8) q.Enqueue(root.Left) 9) end if 10) if root.Right 6= ∅ 11) q.Enqueue(root.Right) 12) end if 13) if !q.IsEmpty() 14) root ← q.Dequeue() 15) else 16) root ← ∅ 17) end if 18) end while 19) end BreadthFirs