Resume

Chapter 3

Binary Search Tree

Binary Search Tree (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.

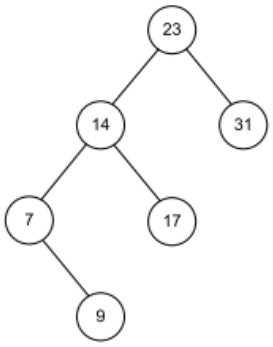


Figure 3.1: Simple unbalanced binary search tree

3.1 Insertion

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

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
```

3.2 Searching

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
```

3.3 Delection

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.

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.

```
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= Ø
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 Ø
40) FindParent(largestV alue.Value).Right ← Ø
41) nodeT oRemove.Value ← largestV alue.Value
42) end if
43) Count ← Count −1
44) return true
45) end Remove

3.4 Finding The Parent Of A Given Node

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 ! = Ø
4) Post: a reference to the parent node of value if found; otherwise Ø
5) if value = root.Value
6) return Ø
7) end if
8) if value < root.Value
9) if root.Left = Ø
10) return Ø
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 = Ø
18) return Ø
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

3.5 Attaining A Reference To A Node

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 Ø
5) if root = Ø
6) return Ø
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 FindNod
```

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

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 Ø 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= Ø
4) Post: the smallest value in the BST is located
5) if root.Left = Ø
6) return root.Value
7) end if
8) FindMin(root.Left)
9) end FindMin
```

```
1) algorithm FindMax(root)
2) Pre: root is the root node of the BST
3) root 6= Ø
4) Post: the largest value in the BST is located
5) if root.Right = Ø
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.

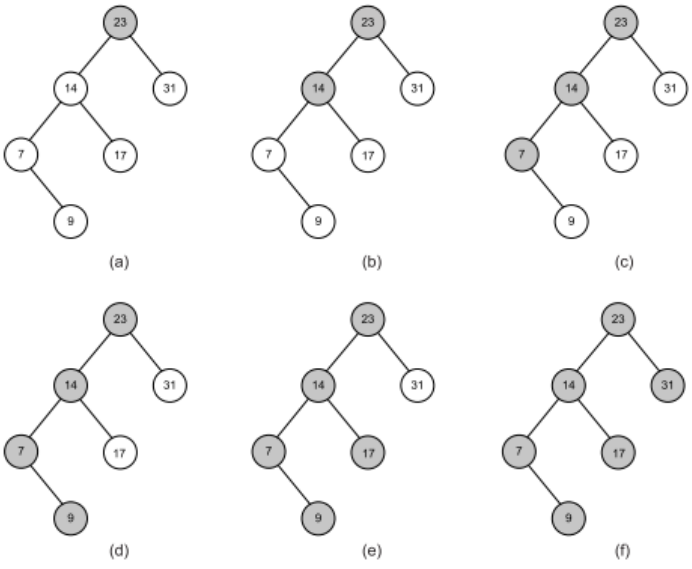
3.7.1 Preorder

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

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

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



3.7.3 Breadth First

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= Ø

6) yield root.Value

7) if root.Left 6= Ø

8) q.Enqueue(root.Left)

9) end if

10) if root.Right 6= Ø

11) q.Enqueue(root.Right)

12) end if

13) if !q.IsEmpty()

14) root ← q.Dequeue()

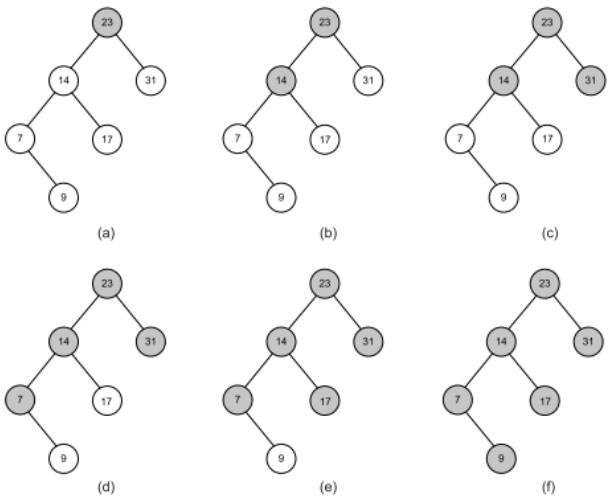
15) else

16) root ← Ø

17) end if

18) end while

19) 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.

IMPLEMENTASI :

Contoh program Penelusuran node (level order) :

```
static void theLevelOrder(Node2P node)
{
QueueArray temp = new QueueArray();
temp.inisialisasi(15);
temp.enqueue(node);
while(temp.jumlah_item >0)
if(node.previous !=null)
temp.enqueue(node.previous);
if(node.next !=null)
temp.enqueue(node.next);
System.out.print(node.data + " ");
if(!temp.isEmpty())
temp.dequeue();
node = temp.peekQueue();
}
```