Binary Search Tree

Data Structures Made Easy

DUBLIN CITY UNIVERSITY

Contents

1. *Definition* 3
2. *Implementation* 3
3. *Example* 3
4. *Functions* 4
5. *Binary Search Tree Pseudocode* 5
6. *Complexity* 9
7. *Advantages of Binary Search Trees* 9
8. *Disadvantages of Binary Search Trees* 9
9. *References* 10

1. ***Definition***

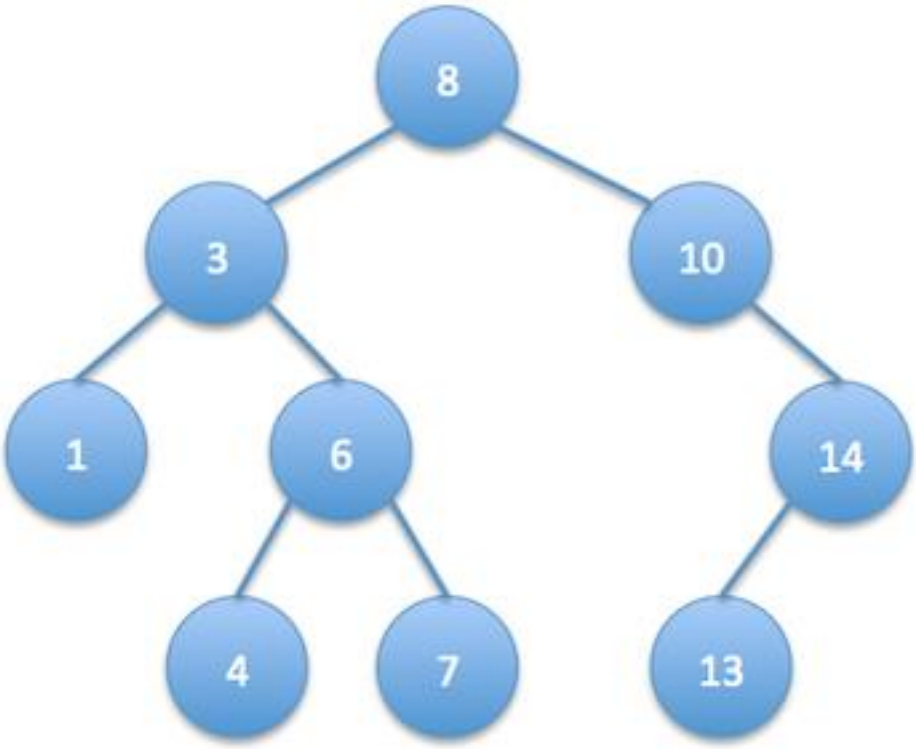
A binary search tree is a binary tree that contains a collection of nodes. Each node possesses a Comparable key and an associated value. The binary search tree satisfies the restriction that the Comparable key within any particular node in the tree is smaller than the keys in all other nodes in that particular node's right subtree and larger than the keys in all other nodes in that particular node's left subtree.

2. ***Implementation***

A binary search tree works as follows:

1. Insert a key value.
2. Determine whether a key value is present within the binary tree.
3. Remove a selected key from the binary tree.
4. Display all of the key value in sorted order.

3. ***Example***



4. ***Functions***

The implementation of a binary search tree involves the following functions. Please note that not all functions are fully required for the implementation of a binary search tree, but are referenced for optimal performance

- **add (T)**
  This function inserts a new node that contains the specified data.

- **add (Node, T)**
  This function inserts a new node that contains the specified data. The insertion begins from the specified node within the tree and continues down the correct path until a location to place the new node has been discovered.

- **remove (T)**
  This function deletes a node that contains the specified data.

- **remove (Node, T)**
  This function deletes a node that contains the specified data. Once the node containing the data has been located, it is extracted from the tree. The merging process commences and the tree is restructured correctly.

- **mergeTrees (Node, T)**
  This function merges the binary search tree's that are rooted at separate node. The function returns the newly formed root node.

- **size ( )**
  (Optional) This function returns the number of items that are contained within the binary search tree.

- **contains (T)**
  (Optional) This function returns a boolean value, depending if there is a node within the binary search tree that contains the specified data.

- **contains (Node, T)**
  (Optional) This function returns a boolean value, depending if the specified node within the binary search tree contains the specified data.

- **preOrderTraversal ( )**
  (Optional) This function traverses the binary tree in a pre-order style. The traversal begins at the root node.

- **preOrderTraversal (Node)**
  (Optional) This function traverses the binary tree in a pre-order style. The traversal begins at the specified node.

- **inOrderTraversal ( )**
  (Optional) This function traverses the binary tree in a in-order style. The traversal begins at the root node.

- **inOrderTraversal (Node)**
  (Optional) This function traverses the binary tree in a in-order style. The traversal begins at the specified node.

- **postOrderTraversal ( )**
  (Optional) This function traverses the binary tree in a post-order style. The traversal begins at the root node.

- **post OrderTraversal (Node)**
  (Optional) This function traverses the binary tree in a post-order style. The traversal begins at the specified node.

5. ***Binary Search Tree Pseudocode***

```
class Node < T >{

        T item
        Node null

        constructor Node
}

function size (){

        return number of items in tree
}

function contains ( Node, T ){

        if Node == null
                return false

        else if Node item equals T
                return true

        else if Node item equals T > 0
                return contains ( Node left, T )
```

```
        else
                return contains ( Node right, T )
}

function contains ( T ){

        return contains ( root, T )
}

function add ( Node, T ){

        if Node == null
                increment number of items in tree
                return new Node with root

        else if Node item compared with T > 0
                add left Node with T
                return Node

        else if T compared with Node item > 0
                add right Node with T
                return Node

        else
                return Node
}

function add ( T ){

        count = number of items in tree
        add root node with T

        if count < number of items in tree
                return true
        else
                return false
}

function remove ( Node, T ){

        if Node == null
                return Node

        else if Node item compared with T > 0
                remove left Node with T
                return Node
```

```
        else if T compared with Node item > 0
                remove right Node with T
                return Node

        else
                decrement number of items in tree
                return mergeTrees ( left Node, right Node )
}

function remove ( T ){

        count = number of items in tree
        remove root node with T

        if count > number of items in tree
                return true
        else
                return false
}

function mergeTrees ( a_Node, b_Node ){

        if b_Node == null
                return a_Node

        else if b_Node left == null
                b_Node left = a_Node
                return b_Node

        else
                p_Node = b_Node left
                parent_Node = b_Node

                while p_Node left != null
                        parent_Node = p_Node
                        p_Node = p_Node left

                parent_Node left = p_Node right
                p_Node left = a_Node
                p_Node right = b_Node

                return p_Node
}
```

```
function preOrderTraversal ( Node ){

        if ( Node != null)
                print out Node item
                preOrderTraversal ( Node left )
                preOrderTraversal ( Node right )

}

function preOrderTraversal ( ){

        preOrderTraversal ( root )
}

function inOrderTraversal ( Node ){

        if ( Node != null)
                inOrderTraversal ( Node left )
                print out Node item
                inOrderTraversal ( Node right )

}

function inOrderTraversal ( ){

        inOrderTraversal ( root )
}

function postOrderTraversal ( Node ){

        if ( Node != null)
                postOrderTraversal ( Node left )
                postOrderTraversal ( Node right )
                print out Node item
}

function postOrderTraversal ( ){

        postOrderTraversal ( root )
}
```

6. ***Complexity***

The optimal time complexity of a binary search tree is O(1). However, this best-case only arises if the first node you search is the node you are intentionally searching for.

The average time complexity is O(log N). This is achieved if the tree is moderately balanced, so that it enables you to discard approximately half the values with each step.

The worst-case time complexity is O(N). This occurs if the tree is poorly balanced and every node is structured on one path from the root node. This disables the ability to discard any values after each step.

7. ***Advantages of Binary Search Trees***

The advantage of a binary search tree is that is computationally fast, in terms of storing keys in nodes. This results in the performance of inserting data, deleting data and balancing the tree being highly efficient.

In relation to implement of data structures, binary search trees are considered to be easier to implement than linked lists. The fundamentality and efficiency of the data structure has made it an ideal selection being used in many applications.

8. ***Disadvantages of Binary Search Trees***

The disadvantage of a binary search tree is when the application is inserting or searching for a particular element, the Comparable key of each visited node must be compared with the Comparable key of the particular element being inserted or searched. There is a possibility that the run time of the application may increase by an unsatisfactory amount.

9. ***References***

Stoimen 2012. *Computer Algorithms: Binary Search Tree* [Online]. Available from: http://www.stoimen.com/blog/2012/06/22/computer-algorithms-binary-search-tree-data-structure

Walker J. 2010 *Binary Search Trees 1* [Online]. Available from: http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_bst1.aspx

Walker J. 2010 *Binary Search Trees 2* [Online]. Available from: http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_bst2.aspx