ASSIGNMENT NO.:

Title: Using Divide and Conquer Strategies design a cluster / Grid or BBB or Raspberry pi or computer in network to run a function for binary search tree using C / C++ / Java / Python / Scala.

Aim: Using Divide and Conquer Strategies design a function for Binary Search tree using C++/ Java/ Python/ Scale.

Objective:

1) To study algorithmic examples of Divide & Conquer strategies.eg: Binary Search Tree.
2) To design the incorporation of parallelism in Binary Search Tree(BST) approach using MPI standards.
3) To demonstrate the BST approach using MPI across a cluster of machines/BBBs

Theory:

Binary Search Tree

A binary search tree is a rooted binary tree, whose internal nodes each store a key and each have two distinguished sub-trees, denoted left and right. The tree additionally satisfies the binary search tree property, which states that the key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree. The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another. Leaves are commonly represented by a special leaf or nil symbol, a NULL pointer, etc.

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient; they are also easy to code.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays

Operations:

Searching

Searching a binary search tree for a specific key can be a recursive or an iterative process. If the tree is null, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left sub tree. Similarly, if the key is greater than that of the root, we search the right sub tree. This process is repeated until the key is found or the remaining sub tree is null. If the searched key is not found before a null sub tree is reached, then the key is not present in the tree.

On average, binary search trees with *n* nodes have O(log n) height. However, in the worst case, binary search trees can have O(n) height, when the unbalanced tree resembles a linked list (degenerate tree).

Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right sub trees as before. Eventually, we will reach an external node and add the new key-value pair as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the

new node to the left sub tree if its key is less than that of the root, or the right sub tree if its key is greater than or equal to the root.

The part that is rebuilt uses O(log *n*) space in the average case and O(*n*) in the worst case. In either version, this operation requires time proportional to the height of the tree in the worst case, which is O(log *n*) time in the average case over all trees, but O(*n*) time in the worst case.

Deletion

There are three possible cases to consider:

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted *N*.

Do not delete *N*. Instead, choose either its in-order successor node or its in-order predecessor node, *R*. Copy the value of *R* to *N*, then recursively call delete on *R* until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NIL , then its in-order successor is node with least value in its right sub tree, which will have at a maximum of 1 sub tree, so deleting it would fall in one of first 2 cases.

Traversal

Once the binary search tree has been created, its elements can be retrieved in-order by recursively traversing the left sub tree of the root node, accessing the node itself, then recursively traversing the right sub tree of the node, continuing this pattern with each node in the tree as it's recursively accessed. As with all binary trees, one may conduct a pre-order traversal or a post-order traversal, but neither are likely to be useful for binary search trees. An in-order traversal of a binary search tree will always result in a sorted list of node items. Traversal requires O(n) time, since it must visit every node.

Machine Cluster Creation

1. Take 3 PC/BBB(nodes)

2. change the hostnames of the respective nodes using cmd
root@PC#hostname ***node1***
root@PC #hostname ***node2***
root@PC #hostname ***node3***

3. Create a user on each of the nodes
root@node1# ***adduser student***
root@node2# ***adduser student***
root@node3# ***adduser student***

4. check ip address of nodes using ***ifconfig*** (make sure all the PC are in the same network)

5. modify **/etc/hosts** of each node with ***IP-address hostname*** pair of every-node

Eg. 172.16.133.54 node1
172.16.132.12 node2
172.16.133.52 node3

6. now switch to student user
root@node1# su – student
root@node2# su – student
root@node3# su – student

7. Generate **ssh keys**
student@node1$ **ssh-keygen**
student@node2$ **ssh-keygen**
student@node3$ **ssh-keygen**

8. Now from node1 perform following commands
student@node1$ ssh-copy-id node2
student@node1$ ssh-copy-id node3
similarly perform ssh-copy-if from node2 to node1&node3 ***and*** *from node3 to node1&node2*
9. Now check if you are able to ssh passwordlessly
student@node1$ ssh node2
student@node2$

Input: Elements to be added in BST.

Output: Results of various operations on BST,

Algorithm BST:

BST-Search(x, k)

y ← x

while y != nil

do

if key[y] = k then

return y

else

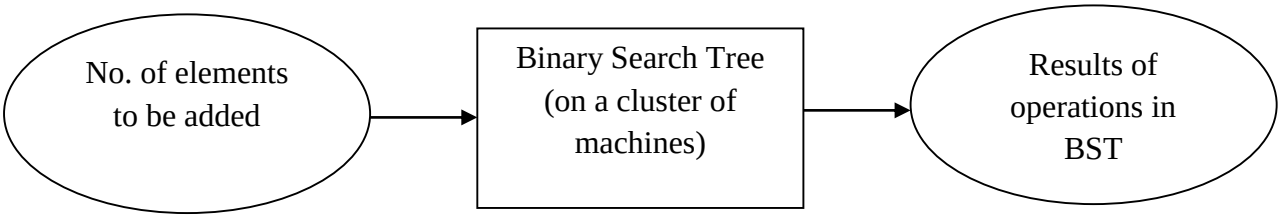
if key[y] < k then

y ← right[y]

else y ← left[y]

return ("NOT FOUND")

Mathematical Model :



Let S be the system such that ,

S={I, O, Fn, Sc, Fc}

I-Input Set

No. of elements to be added

O-Output set

Results of Various Operations performed on BST.

Fn-Function Set

Insertion, Deletion, Searching

Sc – Success Set

Sc1 – Insertion done successfully

Sc2- Correct deletion & searching

Fc – Failure Cases

Fc1 – Elements not inserted / deleted

Fc2 – Search Fails.

Platform: Linux Ubuntu 12.04.

Programming language: C++.

Conclusion: Thus, we have Implemented Binary Search using Divide and Conquer Strategies in C++.