Assignment A-1

Aim:

To study and implement a parallel Binary Search Tree.

Problem Statement:

Using Divide and Conquer Strategies design a cluster/Grid of BBB or Rasberi pi or Computers in network to run a function for Binary Search Tree using C /C++/ Java/Python/ Scala

Theory:

Binary Search Tree:

Binary Search tree is a binary tree in which each internal node x stores an element such that the element stored in the left subtree of x are less than or equal to x and elements stored in the right subtree of x are greater than or equal to x. This is called binary-search-tree property.

The basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with node n, such operations runs in O(lg n) worst-case time. If the tree is a linear chain of n nodes, however, the same operations takes O(n) worst-case time. The height of the Binary Search Tree equals the number of links from the root node to the deepest node.

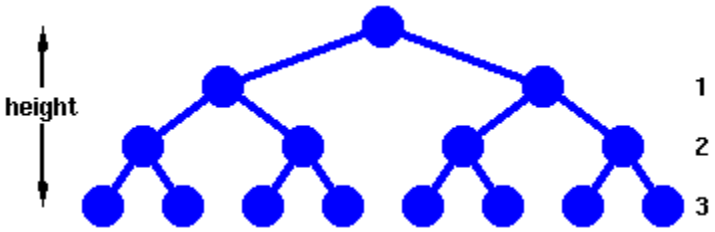


Figure 1: Binary Search Tree

Implementation of Binary Search Tree:

Binary Search Tree can be implemented as a linked data structure in which each node is an object with three pointer fields. The three pointer fields left, right and p point to the nodes corresponding to the left child, right child and the parent respectively NIL in any pointer field signifies that there exists no corresponding child or parent. The root node is the only node in the BTS structure with NIL in its p field.

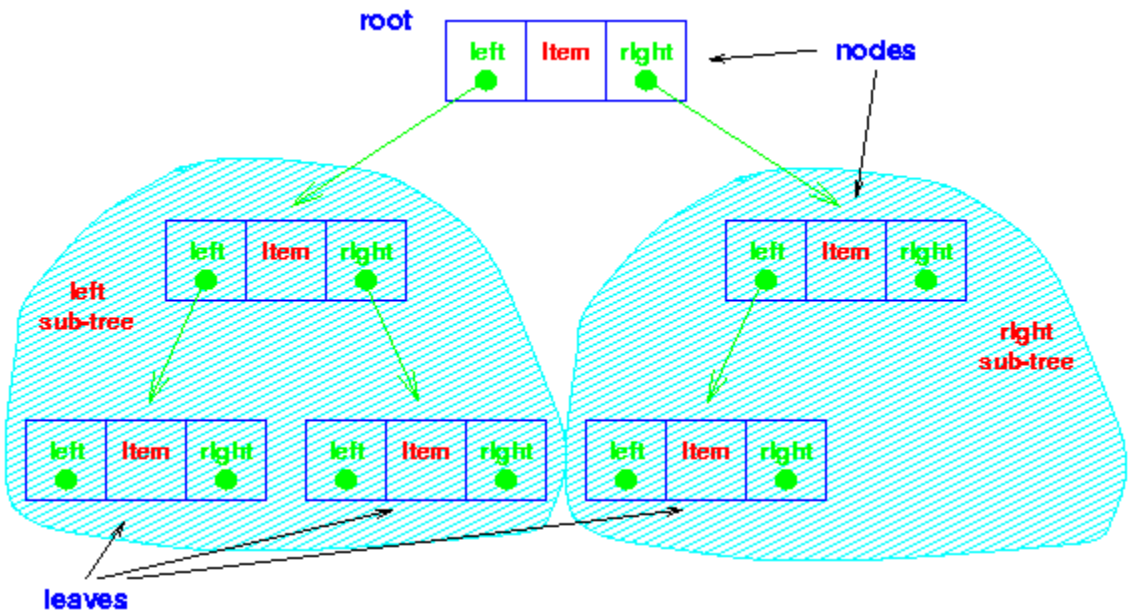


Figure 2: Binary Search Tree Implementation

Parallelism and Scaling:

The current setup works well enough on small amounts of data, but at some point data sets can grow sufficiently large that a single computer cannot hold all of the data at once, or the tree becomes so large that it takes an unreasonable amount of time to complete a traversal. To overcome these issues, parallel computing, or parallelism, can be employed. In parallel computing, a problem is divided up and each of multiple processors performs a part of the problem. The processors work at the same time, in parallel. In a tree, each node/subtree is independent. As a result, we can split up a large tree into 2, 4, 8, or more subtrees and hold subtree on each processor. Then, the only duplicated data that must be kept on all processors is the tiny tip of the tree that is the parent of all of the individual subtrees. Mathematically speaking, for a tree divided among n processors (where n is a power of two), the processors only need to hold n − 1 nodes in common – no matter how big the tree itself is. A picture of this is shown below in Figure. In this picture and throughout the module, a processor's rank is an identifier to keep it distinct from the other processors.

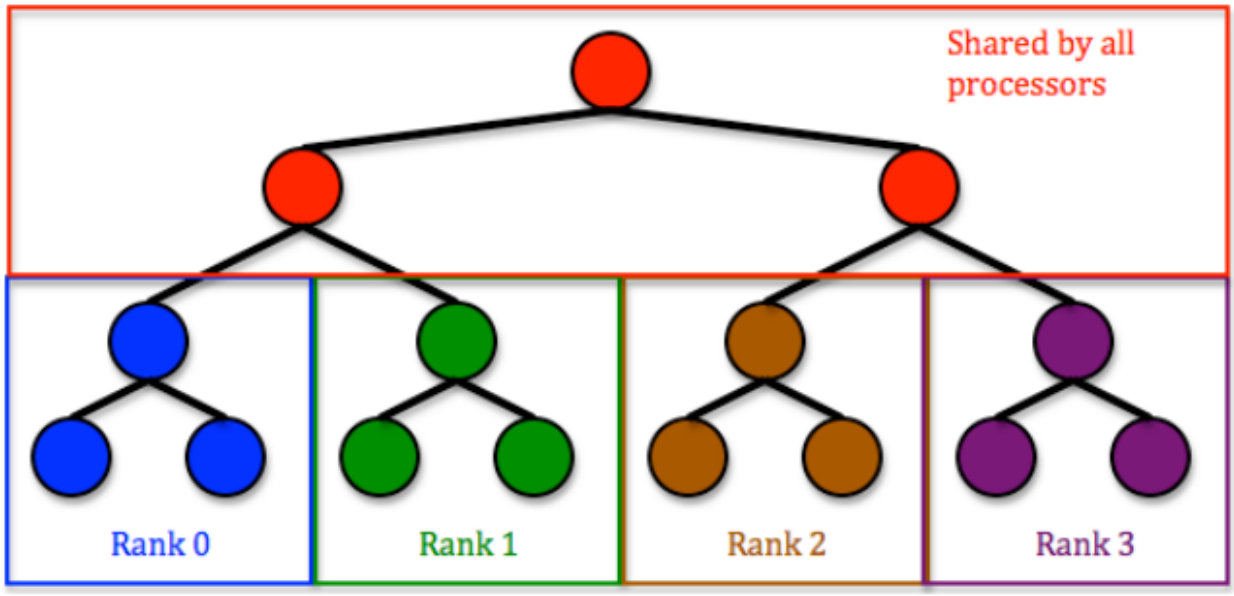


Figure 3: A tree of height 4 split among 4 processors

Algorithm:

Insert:

```
Insert(Node root, Key k)
1 if (root == null) // insertion position found
2 return new Node(k);
3 if (k <= root.key) // proceed to the left branch
4 root.left = Insert(root.left, k);
5 else // k > root.key, i.e. proceed to the right branch
6 root.right = Insert(root.right, k);
```

Search:

```
BST-Search(x, k)
1: y <- x
2: while y != nil do
3: if key[y] = k then return y
4: else if key[y] < k then y <- right[y]
5: else y <- left[y]
6: return ("NOT FOUND")
```

Inorder-Walk:

```
Inorder-Walk(x)
1: if x = nil then return
2: Inorder-Walk(left[x])
3: Print key[x]
4: Inorder-Walk(right[x])
```

Conclusion:

Thus, we have studied and implemented a parallel Binary Search Tree program.

Mathematical Modeling:

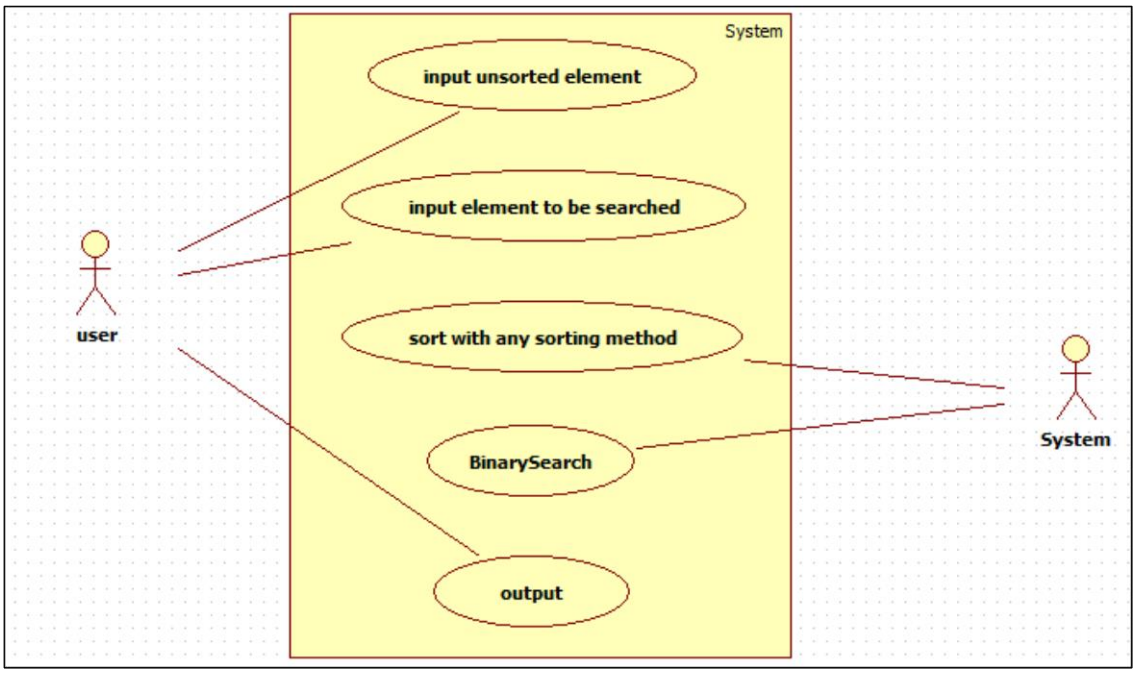
Let S be the system that represents the parallel Binary Search Tree program.
Initially,
S = { Ø }
Let,
S = { I, O, F }
Where,
I – Represents Input set
O – Represents Output set
F – Represents Function set
Input set – I:
Elements of the Binary Search Tree.
Output set – O:
Searched element and the In order traversal.
Function set – F:
F = { F1, F2, F3}
F1 – Represents the function to insert element into Binary Search Tree.
F1(Ri, Ei) –> { E1, E2 ..... En } U { Ei } = { E1, E2 ..... En, Ei }
Where,
· Ri: Root of the ith processor.
· Ei: ith element of the Binary Search Tree.

F2 – Represents the function to print Inorder traversal.
F2(Ri) –> { E1, E2 ..... En }
Where,
· Ri: Root of the ith processor.
· Ei: ith element of the Binary Search Tree.

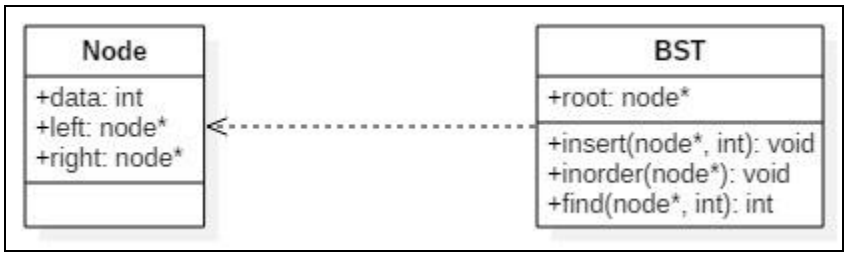
F3 – Represents the function to find element from the Binary Search Tree.
F3(Ri, K) –> { K }
Where,
· Ri: Root of the ith processor.
· K: Search element.
Finally,
S ={ I, O, F }

UMLS:

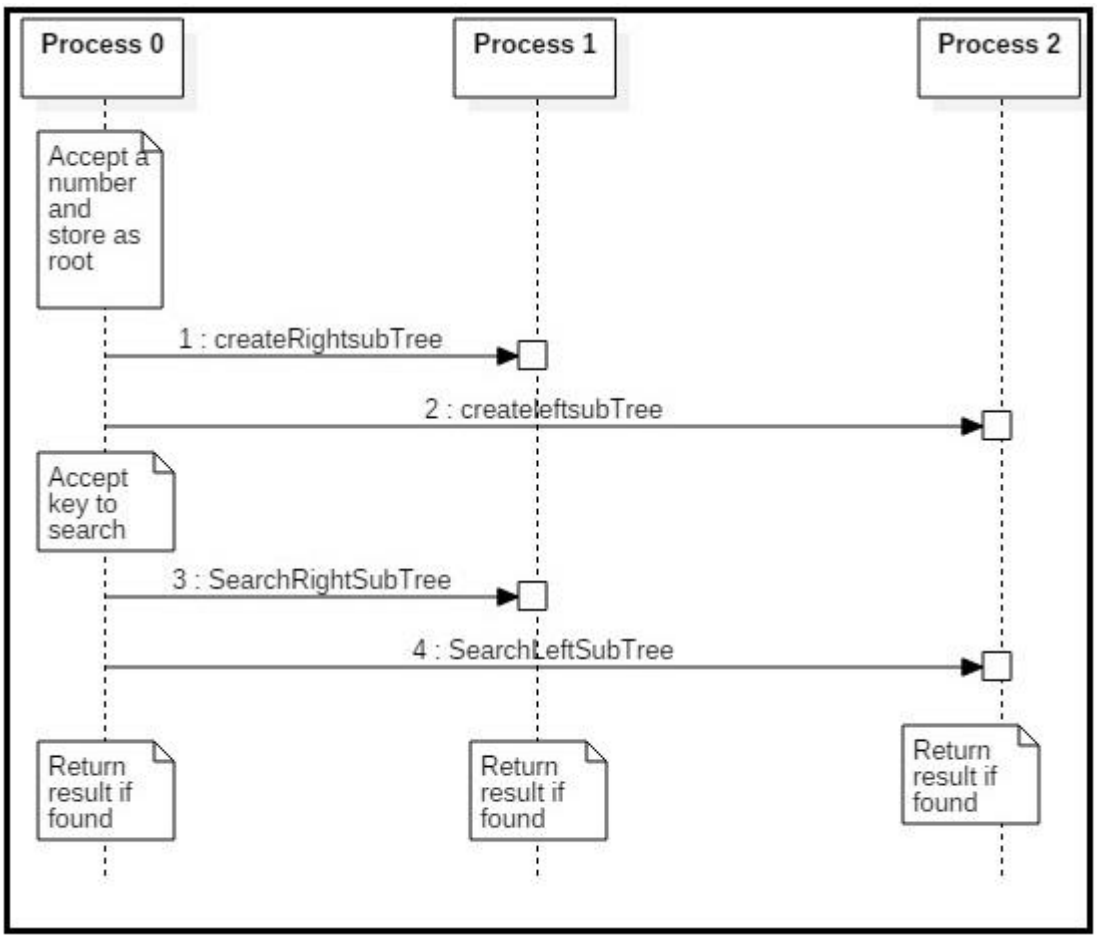
Use Case Diagram



Class Diagram



Sequence Diagram



Relative Efficiency:

Class	Search algorithm
Worst case performance	O(log n)
Best case performance	O(1)
Average case performance	O(log n)
Worst case space complexity	O(1)