**The data structure:**

**BinarySearchTreeADT:** We create a new class with one elements of the type “Node” which called root, and an integer that present the size (the number of the elements in the tree).

Each **Node** has a reference to two other nodes which we called them children (leftChild and rightChild), and an integer to present the value.

**Implementation and the complexity:**

**void printByLevels()**

|  |  |  |
| --- | --- | --- |
| if (root != null) | O(1) | O(1) |
| return; |
| Node[] nodesArray = new Node[size]; | O(1) | O(1) |
| int insertIndex = 0; | O(1) | O(1) |
| int getIndex = 0; | O(1) | O(1) |
| nodesArray[insertIndex++] = root; | O(1) | O(1) |
| visitLivel(nodesArray, insertIndex, getIndex, 0); | O(N) see the next table | O(N) |

**complexity =** O(1)+ O(1)+ O(1)+ O(1)+ O(1)+ O(N)= O(N)

**visitLivel(Node[ ] nodesArray, int insertIndex, int getIndex, int level):**

|  |  |  |
| --- | --- | --- |
| System.out.print("Depth "+ level + ":" ); | O(1) | O(1) |
| result = node.value; | O(1) |  |
| int originalInsertIndex = insertIndex; | O(1) |  |
| for (int i= getIndex; i<originalInsertIndex; i++) | N1 Times | N1\*O(1) |
| Node node = nodesArray[getIndex++]; | O(1) |
| System.out.print(" " + node.value); | O(1) |
| if (node.leftChild!=null) | O(1) |
| nodesArray[insertIndex++] = node.leftChild; | O(1) |
| if (node.rightChild!=null) | O(1) |
| nodesArray[insertIndex++] = node.rightChild; | O(1) |
| System.out.println(""); | O(1) |
| if (originalInsertIndex == insertIndex) | O(1) | O(1) |
| return; |  |  |
| level++; | O(1) | O(1) |
| visitLivel(nodesArray, insertIndex, getIndex, level); | T(N2) | T(2N) |

**complexity = N1\*O(1) + T(N2)**

So,

**Total= N1\*O(1) + T(N2)**

**= N1\*O(1) + N2\*O(1) + T(N3)**

**= N1\*O(1) + N2\*O(1) + N3\*O(1)+………+ T(Nh)**

**= N1\*O(1) + N2\*O(1) + N3\*O(1)+………+Nh\*O(1)**

**= (N1+N2+N3+……+Nh)\*O(1)**

**N1**: is the number of the elements in the first level (i.e the root where N1 = 1).

**N2**: is the number of the elements in the second level.  
.  
.  
.  
**Nh** is the number of the elements at the last level.

Then, the total elements-number in the tree:

**N= N1+N2+N3+……+Nh**

=>

**complexity = N O(1) = O(N)**

**Integer mostSimilarValue(Integer value):**

|  |  |  |
| --- | --- | --- |
| Integer result=null; | O(1) | O(1) |
| if (root != null) | O(1) | O(N) (worst-case) O(log N) (best-case) |
| result = findMostSimila(value, root, result) | O(N) (worst-case)  O(log N) (best-case)  see the next table |
| return result; |  |  |

**complexity (worst-case) =** O(1)+ O(N)= O(N)

**complexity (best-case) =** O(1)+ O(log N)= O(log N)

**findMostSimila(Integer value, Node node, Integer result):**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| if (result == null || value == node.value) | O(1) | | | O(1) |
| result = node.value; | O(1) | | |
| ELSE |  | | | O(1) |
| int currentSimilarity = value - node.value; | O(1) | | |
| int lastSimilarity = value - result; | O(1) | | |
| if (currentSimilarity<0) | O(1) | | |
| currentSimilarity = -currentSimilarity; | O(1) | | |
| if (lastSimilarity<0) | O(1) | | |
| lastSimilarity = -lastSimilarity; | O(1) | | |
| if (currentSimilarity<lastSimilarity) | O(1) | O(1) | Max=  O(1) |
| result = node.value; | O(1) |
| else if (currentSimilarity==lastSimilarity && node.value < result) | O(1) | O(1) |
| result = node.value; | O(1) |
| ENDIF |  | | |  |
| if (node.leftChild != null && value < node.value) | O(1) | | | T(leftChild) |
| result = findMostSimila(value, node.leftChild, result); | T(leftChild) | | |
| Else if (node.rightChild != null && value > node.value) | O(1) | | | T(rightChild) |
| result = findMostSimila(value, node.rightChild, result); | T(rightChild) | | |
| ENDIF |  | | |  |
| return result; |  | | |  |

**complexity =** max (O(1)+ O(1))+ max (T(leftChild) + T(rightChild))  
 = O(1)+ T(one of the children)  
So, the visiting and comparing the value for each node = O(1). Then we need to visit one child. Let us say: **N**: is the number of the elements in the tree. **h**: the height of the tree.  
Therefore, we will end with (h+1 visits) at most. As result:

* The worst-case if each node has only one child:  
  **complexity (worst-case) = O(h+1)= O(N).**
* The best-case if each node has two child:  
  **complexity (best-case) = O(h+1) = O(h).  
  but in this case: h = O(log N) =>  
  complexity (best-case) = O(log N)**