Huffman’s Algorithm

```
// new TreeNode definition //redefine TreeNode

   Object symbol;
   Object probability;
   TreeNode leftchild, rightchild;
```


Huffman’s Algorithm

- //TreeNode class

```
public Object getWeight();
   // return the weight field
public void setWeight();
   //set the weight field to a new weight
```


Huffman’s Algorithm

```
class Tree //

public Tree();  //default constructor

public Tree(KeyedItem rootItem, Float
  weight);
}

public void insert(KeyedItem newItem, Float
  weight) {
}
```


Huffman’s Algorithm

```
class Tree //

public TreeNode getRoot();   //returns the root
public Float getWeight();   // returns weight
public KeyedItem getSymbol(); //returns symbol
public void setRoot(TreeNode tNode); // sets the
   current tree’s root to tNode
public Float getLeftWeight(); // return weight of left
   reference
public Float getRightWeight(); // return weight of
   right reference
```



```
public void printTree();   // prints the tree
   (inorder) with formatting

publid void leafNodeTraversal();
  //traverses the tree and computes the
  code map and prints the symbols with
  their codes

publid void leafNodeTraversal() {
  Path code = new Path();
  LeafNodeHelper(root, code);
}
```


Huffman’s Algorithm

```
public void LeafNodeHelper(TreeNode tNode, Path code)
  {
   if( the tree is not empty) {
     if(the node is a leaf node) {
          print the symbol and the code }
    else {
         add 0 to path and traverse left subtree
          remove a bit from the code
          add 1 to path and traverse right subtree
          remove a bit from the code }
}}
```

```
public void LeafNodeHelper(TreeNode tNode, Path code) {
     if(tNode != null)   {
if( ( tNode.getLeft() == null) && (tNode.getRight()==null))
          {



     }
```



```
else
         {
          code.AddToPath(0);


          code.AddToPath(1);



         }
       }
    }
```



```
public class Path {
final static int MAXPATHLENGTH = 20;
int PathComponents[] = new int[MAXPATHLENGTH];
int PathLength;

public void Path()  {  //initialise PathComponents and
   PathLength



}
```



```
public void AddToPath(int direction) {

 if(PathLength < MAXPATHLENGTH)
   {


  }
 else
   System.out.println("Error maximum pathlength
   reached\n");
}
```



```
public void RemoveFromPath() {
  if (PathLength > 0)

  else
     System.out.println("Error no path exists \n");
}
```



```
public void PrintPath() {
 for(int i=0;i<PathLength;i++)


}
}  //end Path
```

public class Forest {

final static int MaxNumberofTrees = 100;
Tree TreeArray[] = new Tree[MaxNumberofTrees];
static int Forest_Size;

public Forest(); //initialise the TreeArray to hold empty trees


```
public Forest( ) {

}
```



```
public void InitialiseForest(
  KeyedItem [] symbols, Float []
  weights, int index)  {
```

for *index* number of trees in the forest

insert a symbol and a weight from the arrays ***symbols*** and ***weights*** in the root node of each tree

increment the Forest_size

}


```
public void InitialiseForest(KeyedItem []
  symbols, Float [] weights, int index) {

}
```



```
public Float RootWeight(int index)
 {
Float newItem =
  TreeArray[index].getWeight();
     return newItem;

// returns the weight of the root node of
  the tree at index index of the tree array

 }
```



```
public void JoinTrees(int index1, int
  index2, KeyedItem symbol)
  {
   Tree tempTree = new Tree();

   Float w1,w2,w3;
   w1 =   //retrieve weight at Tree indexed by index1
   w2 = //retrieve weight at Tree indexed  by index2
   w3 = w1 + w2
```

```
public void JoinTrees(int index1, int
  index2, KeyedItem symbol)
  {
  Tree tempTree = new Tree();
   Float w1,w2,w3;
   w1 = RootWeight(index1);
   w2 = RootWeight(index2);
   w3 =
```


tempTree.insert(symbol, w3);

//set right reference of tempTree to contents of TreeArray[index1]

//set left reference of tempTree to contents of TreeArray[index2]

//Store tempTree at TreeArray[index1] and delete (makeEmpty) TreeArray[index2]

}


```
tempTree.insert(symbol, w3);
```

//set right reference of tempTree to contents of TreeArray[index1].

//set left reference of tempTree to contents of TreeArray[index2]

//Store tempTree at TreeArray[index1] - delete TreeArray[index2]

}


```
public void ConstructTree(KeyedItem symbol) {

while(Forest_Size >1)  {
```

- Find the smallest and secondsmallest weights and record the index of these TreeArray elements as *index1* and *index2*
- Call method JoinTrees(index1,index2,symbol);
- Decrement the Forest_size;
- Print the symbols and their codes with **`LeafNodeTraversal();`**

}


```
public void ConstructTree(KeyedItem symbol)
  {
   while(Forest_Size >1)
  {
Float smallestweight = new Float(100);
Float secondsmallestweight = new Float(100);
int index1 = 0, index2 = 0;
```



```
for(int i = 0; i< MaxNumberofTrees; i++)
   {
if( TreeArray[i].getRoot() != null)
    {


if(smallestweight > weighting)
        {



} //end if
```

```
else  if(secondsmallestweight >
  weighting)
 {



}  //end else
 }  //end if
} //end for
```



```
//Join the trees and insert symbol
//Decrement Forest_Size;
  }

for(i=0;i>MaxNumberofTrees;i++)
{
if( tree at index i is not empty null)
 {


}
}
```