CSC-391: Data Structures

Lecture: 13

Tree-3: Binary Search Tree, Heap Tree & Huffman Coding Tree

CSE-213
(Data Structure)

Lecture on

Binary Search Tree, Heap Tree & Huffman Coding Tree

Md. Jalal Uddin

Lecturer

Department of CSE

City University

Email: jalalruice@gmail.com

No: 01717011128 (Emergency Call)

Department of Computer Science & Engineering (CSE)
City University, Khagan, Birulia, Savar, Dhaka-1216, Bangladesh

CSC-391: Data Structures

Lecture: 13

Tree-3: Binary Search Tree, Heap Tree & Huffman Coding Tree

Lecture-13:
Tree-3: Binary Search Tree, Heap Tree & Huffman Coding Tree

Objectives of this Lecture:

- Discuss three important tree structures:
  - Binary Search Tree (BST)
  - Heap Tree or Heapsort
  - Huffman Coding Tree

Prepared by: K M Akkas Ali, Associate Professor, IIT, JU

Binary Search Tree:

Binary Tree:

- The simplest form of tree is a **binary tree** in which each node has at most two descendants. A binary tree consists of:
  - a *node* (called the **root** node) and
  - left and right *sub-trees*.
  - Both the sub-trees are themselves binary trees

Binary Search Tree (BST):

- Suppose T is a binary tree. Then T is called a binary search tree (or binary sorted tree) if each node N of the tree has the following properties:
  - Each node has a key (or value), and no two nodes have the same key (i.e., all keys are distinct).
  - The value at node N is greater than every value in the left subtree of N.
  - The value at node N is less than every value in the right subtree od N.
  - The left and right subtrees of the root are also binary search trees.

Note:

- There is an analogous definition of a binary search tree which admits duplicates:
  - The value at node is greater than every value in the left subtree of N and is less than or equal to every value in the right subtree of N.

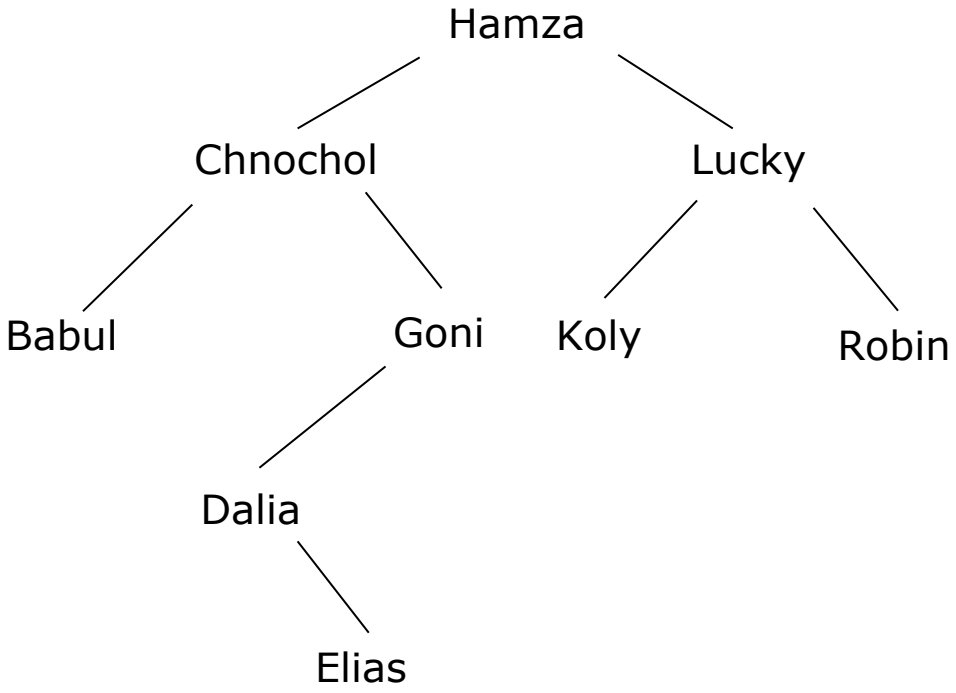
Binary Search Tree:

Advantages of BST:

- A nonempty binary search tree-
  - enables us to search for and to find an element with comparatively less running time than that of sorted linear array or linked list.
  - enables us to insert and delete elements easily.
  - is ideal for implementing dictionaries.

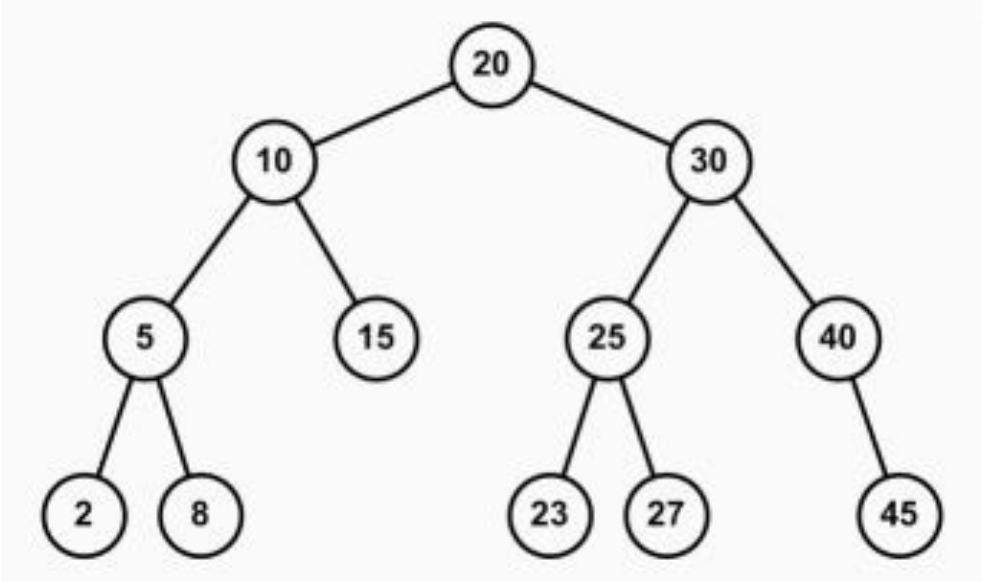
Binary Search Tree:

Examples of BST:



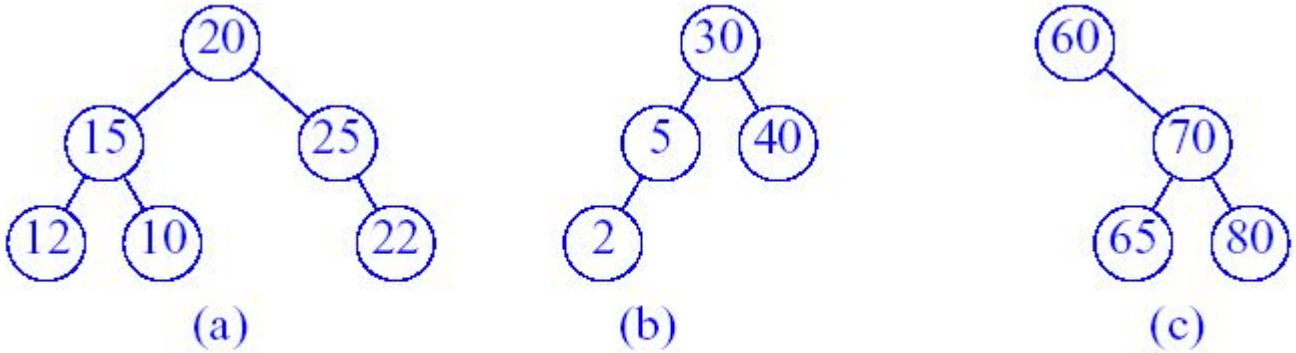
Note:

- If 10 were replaced by 13, then T would still be a binary search tree
- But, if 10 were replaced by 23, then T would not be a binary search tree, since left child of 20 can not be greater than 20.



Binary Search Tree:

Examples of BST:



- Which of the above trees are binary search trees?
  - (b) and (c)
- Why isn't (a) a binary search tree?
  - It violates the property that the value at node N (here 25) is less than or equal to every value in the right.
- If 5 from (b) is replaced by 23, then is T still a binary search tree?
- If 5 from (b) is replaced by 37, then is T still a binary search tree?

Operations on Binary Search Tree:

Following operations are performed on a binary search tree:

(a) Searching for a node
(b) Inserting a node
(c) Deleting a node

Searching in a Binary Search Tree:

- Suppose T is a binary search tree and an ITEM of information is given.
- The following algorithm finds the location of ITEM in the binary search tree.

Step-1: Compare ITEM with the root N of the tree.

i. If ITEM<N, proceed to the left child of N.

ii. If ITEM>N, proceed to the right child of N.

Step-2: Repeat step-1 until one of the following occurs:

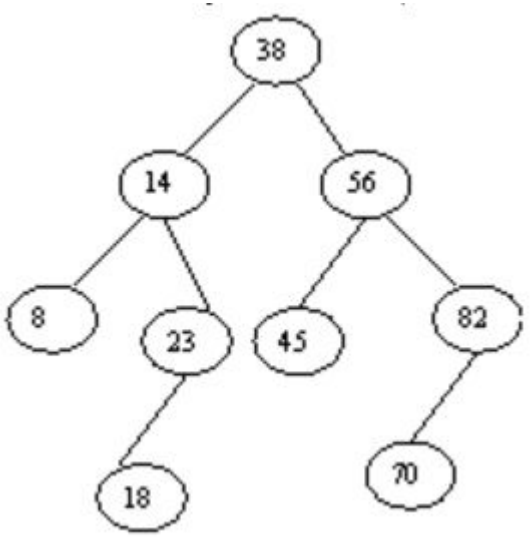
iii. We meet a node N such that ITEM=N. In this case, the search is successful.

iv. We meet an empty subtree, which indicates that the search is unsuccessful.

Searching in a Binary Search Tree:

Example: Searching

- Consider the binary search tree shown below.
- Suppose ITEM=20 is given.
- Simulate the searching algorithm to find whether 20 is in the binary search tree.



Solution:

- By simulating the algorithm, we obtain the following steps:

(1) Compare ITEM=20 with the root, 38, of the tree T. Since 20<38, proceed to the left child of 38, which is 14.

(2) Compare ITEM=20 with 14. Since 20>14, proceed to the right child of 14, which is 23.

(3) Compare ITEM=20 with 23. Since 20<23, proceed to the left child of 23, which is 18.

(4) Compare ITEM=20 with 18. Since 20>18 and 18 does not have a right child, which indicates that the search is unsuccessful.

Inserting Node in a Binary Search Tree:

- An element can be inserted into an appropriate position in a BST.
- Suppose T is a binary search tree and an ITEM of information is given.
- The following algorithm inserts ITEM as a new node in its appropriate place.

Step-1: Compare ITEM with the root N of the tree.

i. If ITEM<N, proceed to the left child of N.

ii. If ITEM>N, proceed to the right child of N.

Step-2: Repeat step-1 until ITEM is compared with a leaf node:

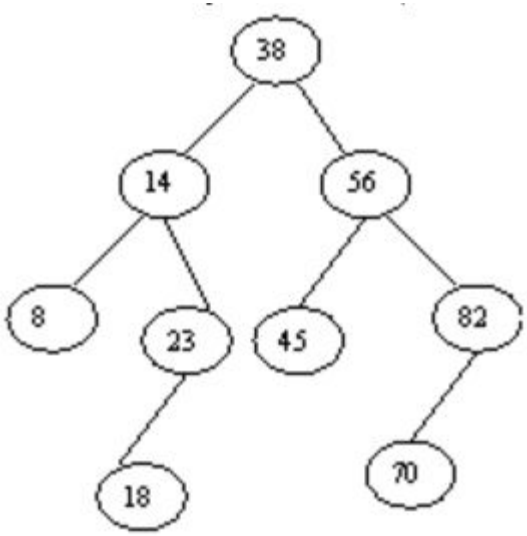
iii. If ITEM is less than the leaf node, then insert it as the left child of the leaf node.

iv. If ITEM is greater than the leaf node, then insert it as the right child of the leaf node.

Inserting Node in a Binary Search Tree:

Example: Inserting

- Consider the binary search tree shown below.
- Suppose ITEM=20 is given.
- Simulate the inserting algorithm to insert 20 as a node in the binary search tree.



Solution:

- By simulating the algorithm, we obtain the following steps:

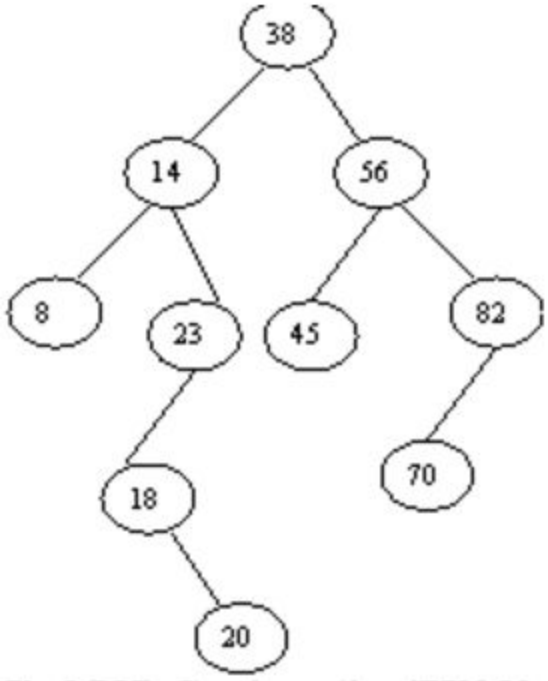
(1) Compare ITEM=20 with the root, 38, of the tree T. Since 20<38, proceed to the left child of 38, which is 14.

(2) Compare ITEM=20 with 14. Since 20>14, proceed to the right child of 14, which is 23.

(3) Compare ITEM=20 with 23. Since 20<23, proceed to the left child of 23, which is 18.

(4) Compare ITEM=20 with 18. Since 20>18 and 18 does not have a right child, insert 20 as the right child of 18.

- Hence we will get the following Binary search Tree after inserting the ITEM=20.



Building a Binary Search Tree:

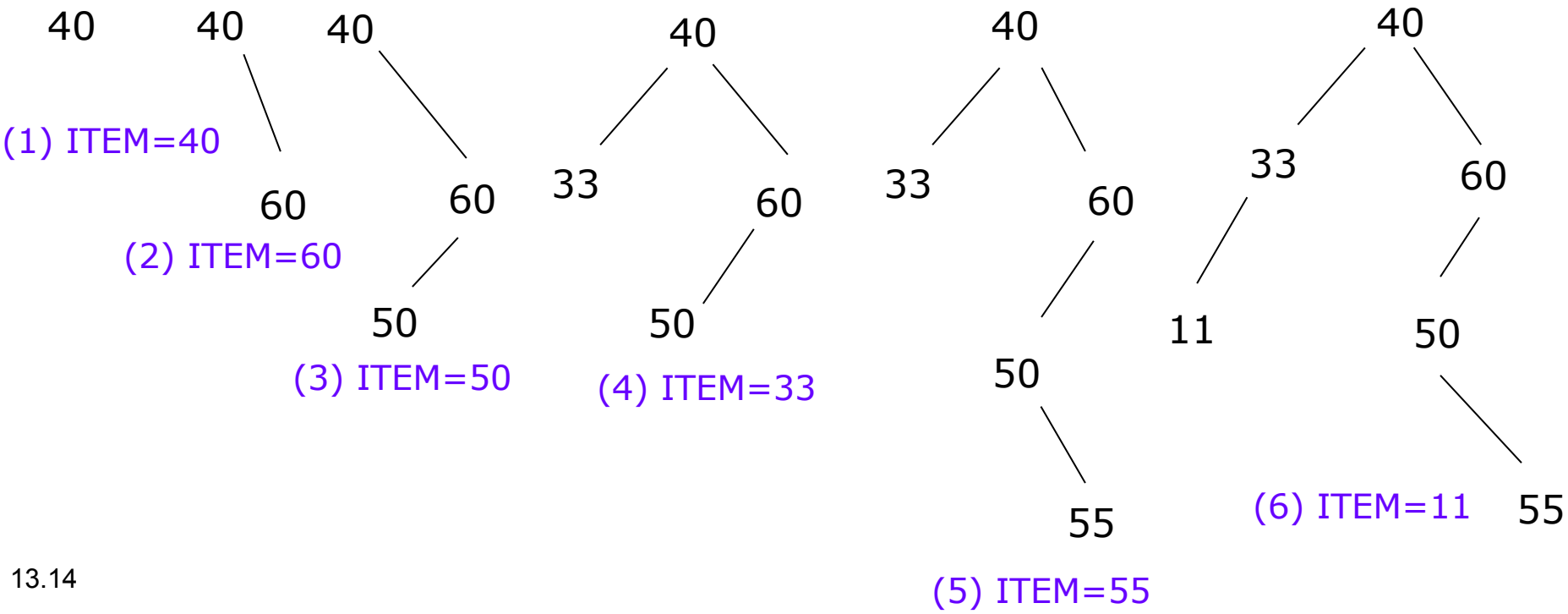
Example: Building a binary search tree

- Suppose the following six numbers are inserted in order into an empty binary search tree:

  40, 60, 50, 33, 55, 11

- Build a binary search tree by showing each stage of the drawing.

Solution:



Deleting in a Binary Search Tree:

- Suppose T is a binary search tree and suppose an ITEM of information is given. The following algorithm deletes ITEM from the tree.
- The algorithm first finds the location of the node N which contains ITEM. The algorithm also finds the location of the parent node of N.
- The way N is deleted from the tree depends primarily on the number of children of node N. There are three cases of the algorithm:

Case-1: node N has no children.

- In this case, N is deleted from T by simply replacing the location of N in the parent node P(N) by the null pointer.

Case-2: node N has exactly one child.

- In this case, N is deleted from T by simply replacing the location of N in the parent node P(N) by the location of the only child of N.

Case-3: node N has two children.

- In this case, let S(N) denote the inorder successor of N. Then N is deleted from T by first deleting S(N) from T (by using case-1 or case-2) and then replacing node N by the node S(N).

Deleting in a Binary Search Tree:

Example-1: Case-1 (N has no children)

- Consider the binary search tree shown in the figure-A below.
- Suppose this tree appears in memory as shown in figure-B.
- Suppose we want to delete node **44** from the tree.
- Simulate the related algorithm to delete the node.

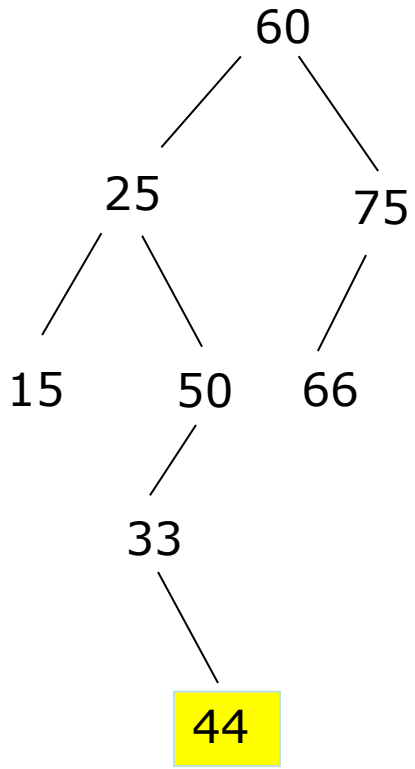


Figure-A: Before deletion

ROOT

3

AVAIL

5

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

Figure-B: Linked representation

Deleting in a Binary Search Tree:

Solution:

- Node 44 in the previous tree has no children.
- Deletion is simply accomplished by assigning NULL to the parent node 33 of 44.
- Figure-C shows the tree after deletion and figure-D shows the updated linked representation.

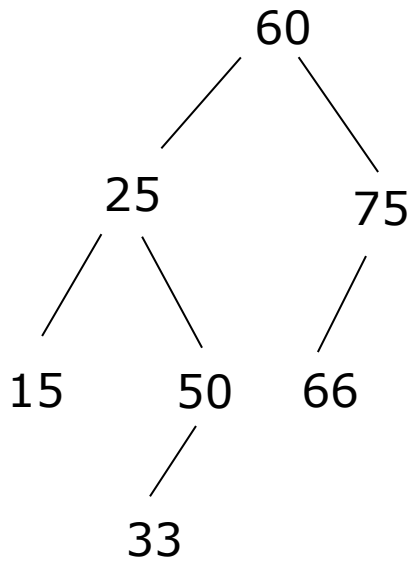


Figure-C: After deletion

ROOT

3

AVAIL

9

	INFO	LEFT	RIGHT
1	33	0	0
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9		5	
10	50	1	0

Figure-D: Updated linked representation

Deleting in a Binary Search Tree:

Example-2: Case-2 (N has exactly one child)

- Consider the binary search tree shown in the figure-A below.
- Suppose this tree appears in memory as shown in figure-B.
- Suppose we want to delete node **75** from the tree.
- Simulate the related algorithm to delete the node.

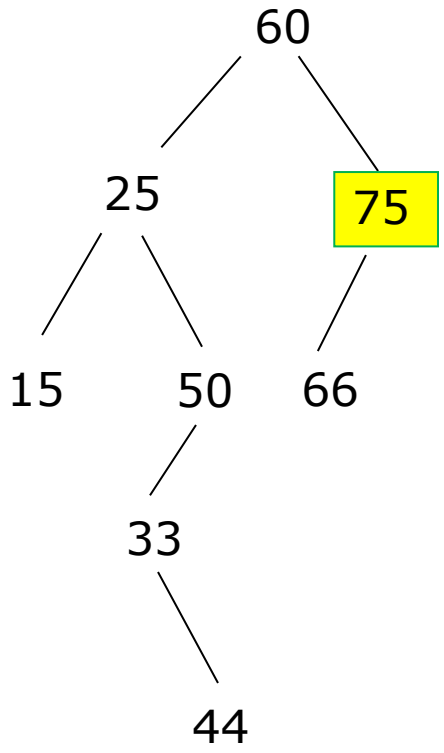


Figure-A: Before deletion

ROOT

3

AVAIL

5

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

Figure-B: Linked representation

Deleting in a Binary Search Tree:

Solution:

- Node 75 in the previous tree has only one child.
- Deletion is simply accomplished by changing the right pointer of the parent node 60 of 75, which originally pointed to 75, so that it now points to node 66, the only child of 75.
- Figure-C shows the tree after deletion and figure-D shows the updated linked representation.

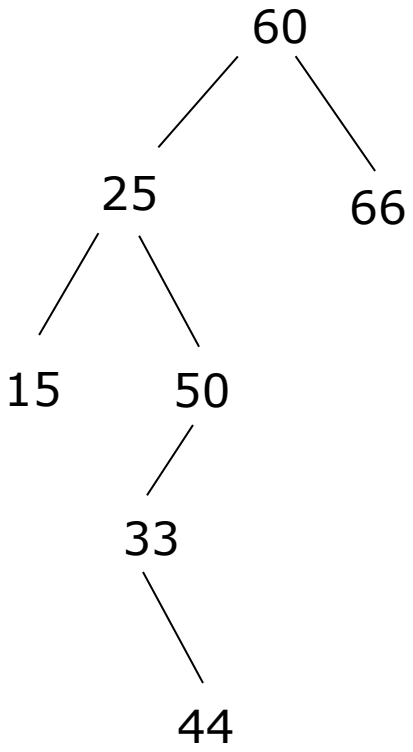


Figure-C: After deletion

ROOT

3

AVAIL

7

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	4
4	66	0	0
5		6	
6		0	
7		5	
8	15	0	0
9	44	0	0
10	50	1	0

Figure-D: Updated linked representation

Deleting in a Binary Search Tree:

Example-3: Case-3 (N has two children)

- Consider the binary search tree shown in the figure-A below.
- Suppose this tree appears in memory as shown in figure-B.
- Suppose we want to delete node **25** from the tree.
- Simulate the related algorithm to delete the node.

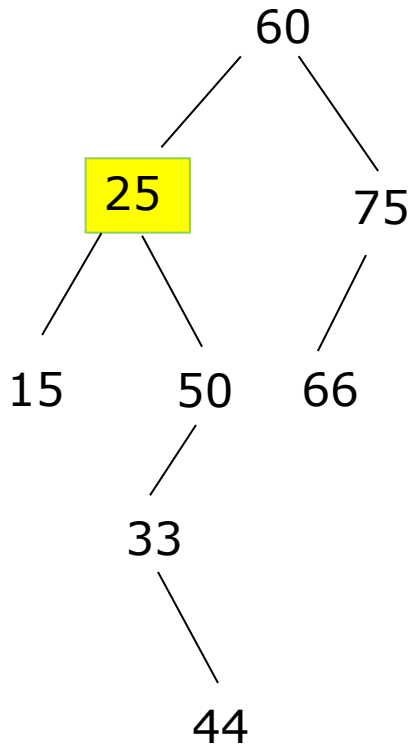


Figure-A: Before deletion

ROOT

3

AVAIL

5

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

Figure-B: Linked representation

Deleting in a Binary Search Tree:

Solution:

- Node 25 in the previous tree has two children. Node 33 is the inorder successor of node 25.
- Deletion is simply accomplished by first deleting 33 from the tree and then replacing node 25 by node 33.
- Figure-C shows the tree after deletion and figure-D shows the updated linked representation.

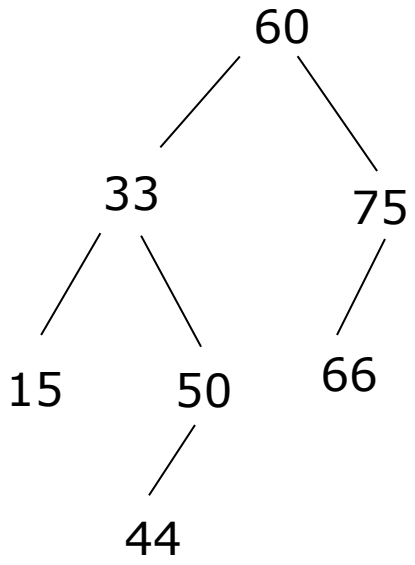


Figure-C: After deletion

ROOT

3

AVAIL

2

	INFO	LEFT	RIGHT
1	33	8	10
2		5	
3	60	1	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	9	0

Figure-D: Updated linked representation

Heap or Heap Tree:

- A heap is a complete binary tree that is ordered from bottom to top, so that a traversal along any leaf-to-root path will visit the keys
  - in ascending order: max heaps **: key(parent) >= key(child)]**
  - in descending order: min heaps **: key(parent) ≤ key(child)**

- Suppose T is a complete binary tree with n elements. Then T is called a heap or a heap tree (or, a max heap), if each node N has the following property:

1. The value at node N is greater than or equal to the value at each of the children of N.
2. Accordingly, The value at node N is greater than or equal to the value at any of the descendents of N.

- A min heap is defined as: The value at node N is less than or equal to the value at any of the children of N.

- Heaps are used-
1. to implement priority queues,
2. to implement the Heap sort algorithm.

Heap or Heap Tree:

Example:

- The complete binary tree shown below is a heap tree.
- Note that the largest element in the heap tree appears at the root of the tree.

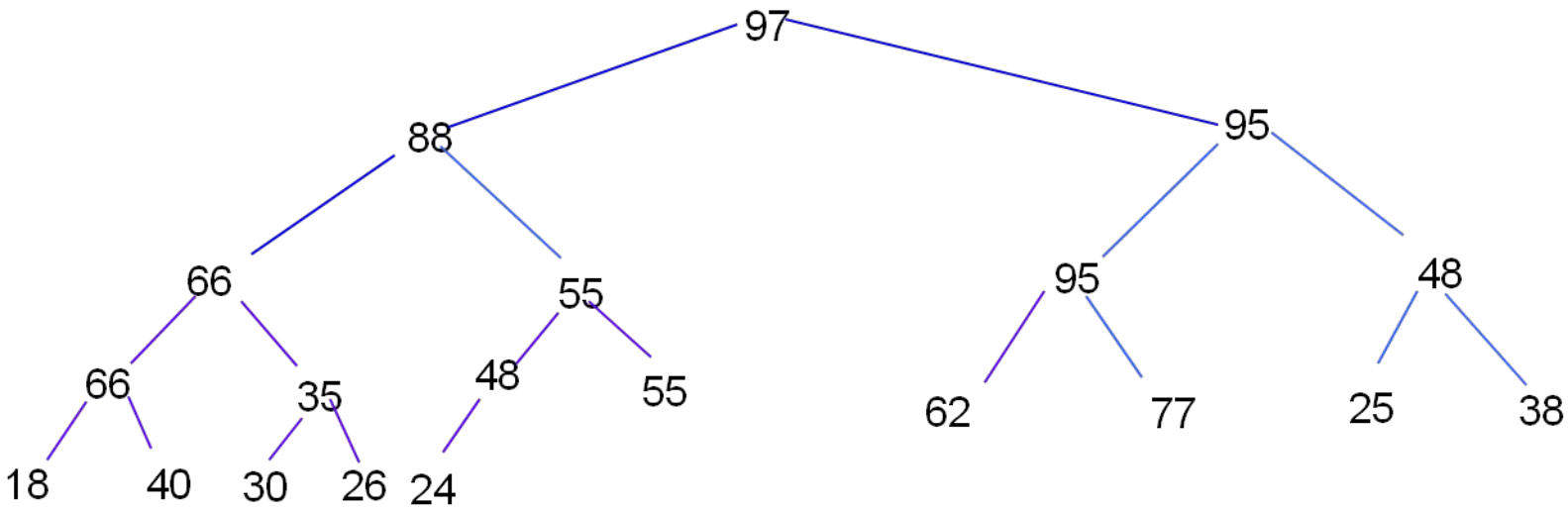


Figure: Heap or Heap tree

Inserting Node into a Heap:

- Suppose H is a heap with N elements, and suppose an ITEM of information is given.
- ITEM can be inserted into the heap as follows:

1. At first adjoin ITEM at the end of H so that H is still a complete tree, but not necessarily a heap.
2. Then let ITEM rise to its 'appropriate place' in H so that H is finally a heap.

Prepared by: K M Akkas Ali, Associate Professor, IIT, JU

Inserting Node into a Heap:

Example:

- Consider the heap tree H shown before. Suppose we want to add ITEM=70 to H.
- 70 can be inserted into the heap as follows:

1. At first adjoin 70 at the end of H so that H is still a complete tree, but not necessarily a heap. This is shown in figure-A below.
2. Then compare 70 with its parent 48. Since 70 is greater than 48, interchange 70 and 48. The path with now look like figure-B.
3. Now compare 70 with its new parent 55. Since 70 is greater than 55, interchange 70 and 55. The path with now look like figure-C.
4. Compare 70 with its parent 88. Since 70 is less than 88, ITEM=70 has risen to its appropriate place in H.

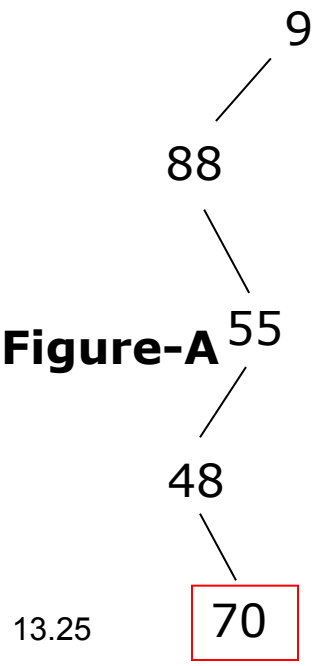


Figure-A

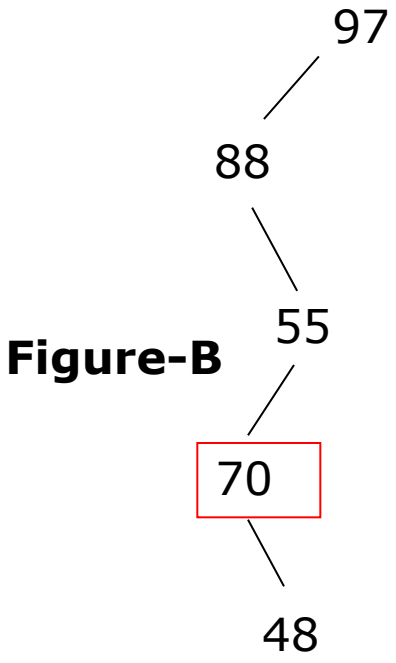


Figure-B

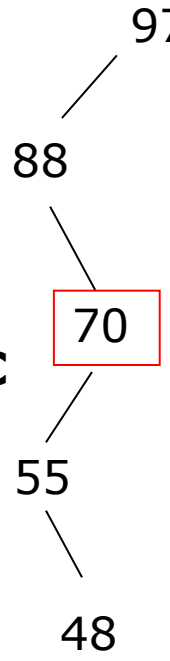


Figure-C

Inserting Node into a Heap:

Example (continue…):

- After inserting ITEM=70, the final heap tree is shown in the figure below.

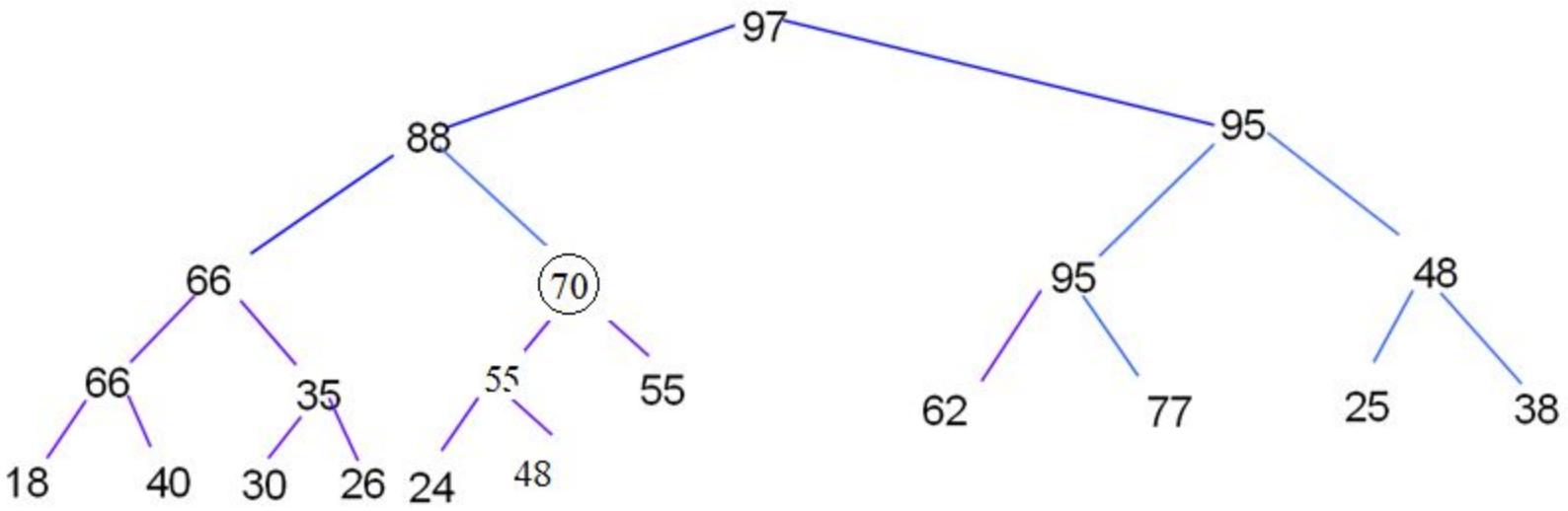


Figure: Heap after inserting ITEM=70

Building a Heap:

Example:

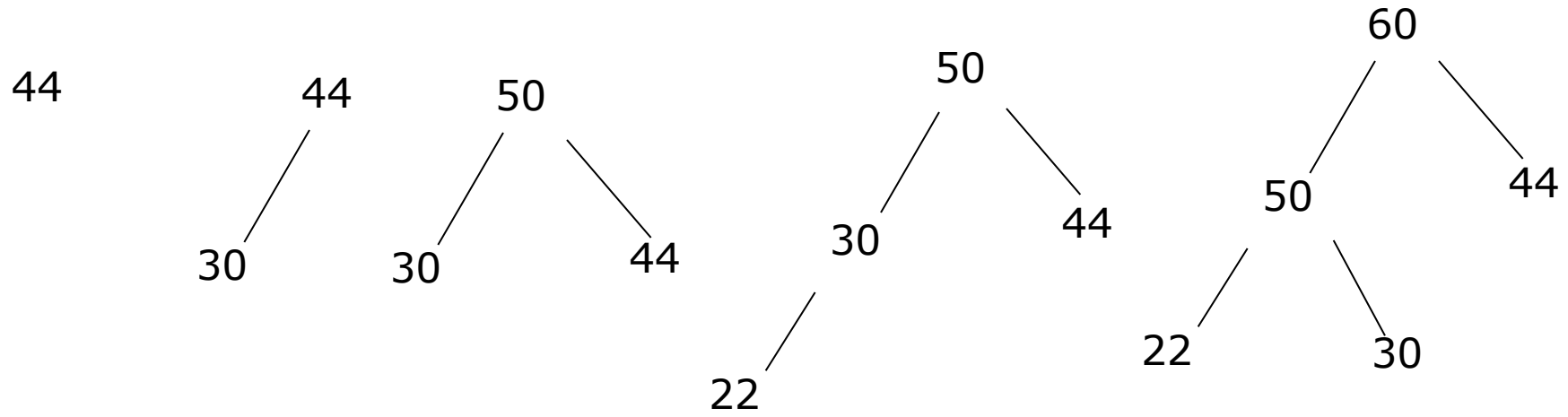
- Consider the following list of numbers:

  44, 30, 50, 22, 60, 55, 77, 55

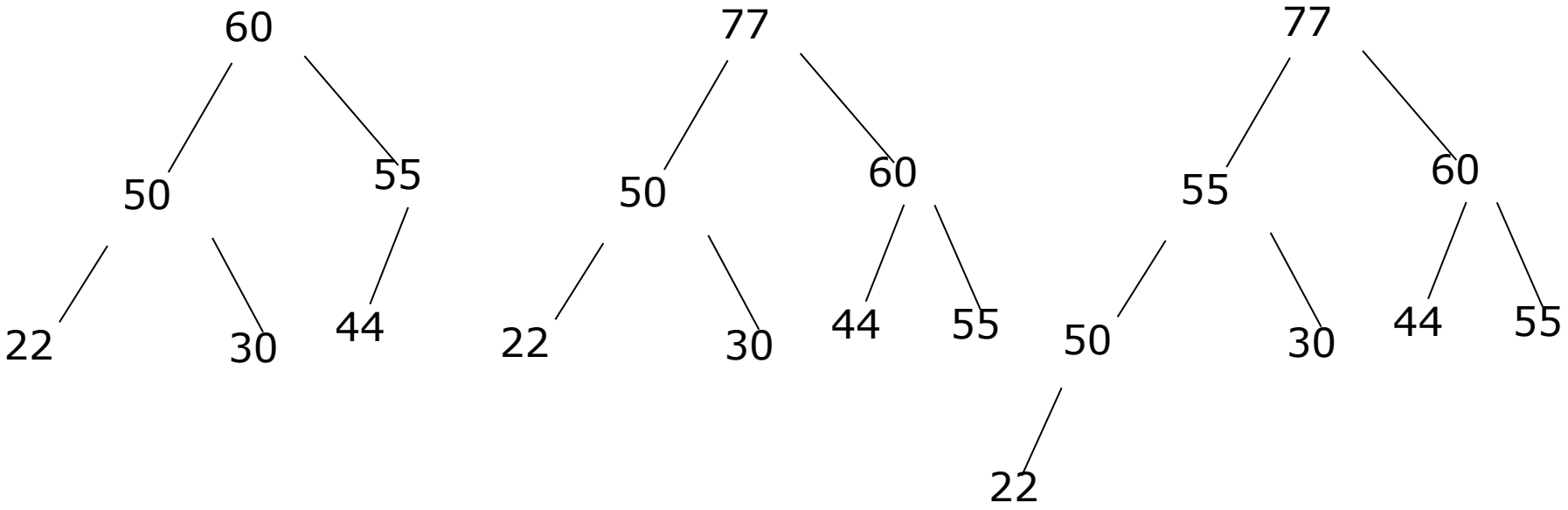
- Build a heap H using the above numbers.

Solution:

- Step-by-step figures are shown below for building the required heap.



Building a Heap:



Path Lengths:

- Let T be an extended binary tree or 2-tree.
- Then each node of T has either 0 or 2 children.
- The nodes with no children are called external nodes, and the nodes with 2 children are called internal nodes.
- Sometimes internal nodes are depicted by circles and external nodes are by squares for distinguishing purpose.
- In any 2-tree, the number of external nodes is 1 more than the number of internal nodes.
- Figure below shows a 2-tree with 6 internal nodes. Then it has 6+1=7 external nodes.

If N_E is the number of external node and N_I is the number of internal node, then

$$N_E = N_I + 1$$

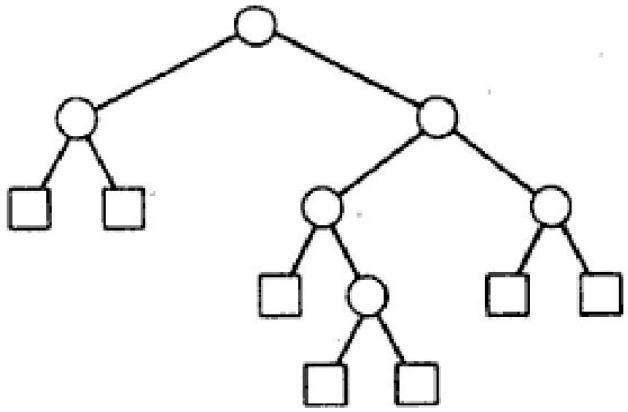


Fig: Extended binary tree or 2-tree

External and Internal Path Lengths:

External path length, L_E:

- It is the sum of all path lengths summed over each path from the root to an external node of a 2-tree. Here, $\mathbf{L_E}$ =2+2+3+4+4+3+3=21

Internal path length, L_I:

- It is the sum of all path length summed over each path from the root to an internal node of a 2-tree. Here, $\mathbf{L_I}$ =0+1+1+2+3+2=9

- Observe that $L_I+2N_I=9+2*6=9+12=21=L_E$ Therefore,

$$\mathbf{L_E = L_I + 2N_I}$$

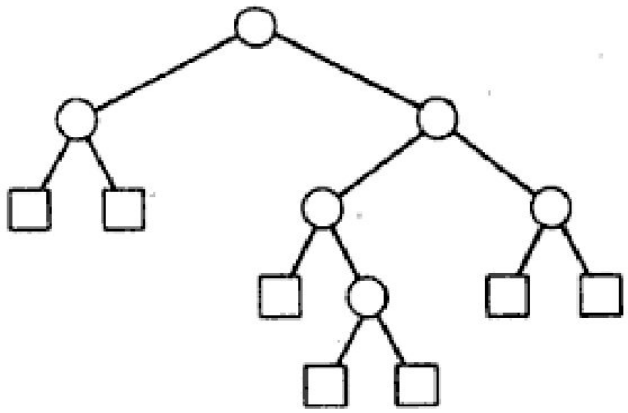


Fig: Extended binary tree or 2-tree

Weighted Path Length P:

- Assume a 2-tree with n external nodes.
- Suppose each of the external nodes is assigned a (nonnegative) weight.
- Then the (external) weighted path length P of the tree is defined to be the sum of the weighted path lengths; i.e.,

$$P = W_1L_1 + W_2L_2 + W_3L_3 + \ldots\ldots + W_nL_n$$

Where W_i and L_i denote respectively the weight and path length of an external node N_i of the 2-tree.

- We now consider the collection of all 2-trees with n external nodes. It is clear that only the complete tree among all 2-trees will have a minimal external path length.
- On the other hand, suppose each of the above 2-trees is given the same n weights for its external nodes. Then it is not clear which of the tree (complete tree or non-complete tree) will give a minimal weighted path length.

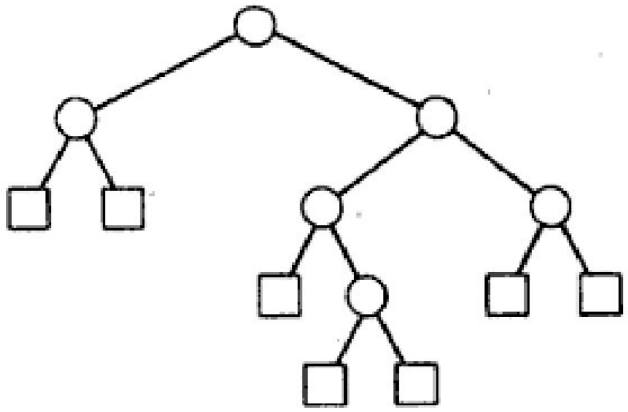


Fig: Extended binary tree or 2-tree

Weighted Path Length P:

Example:

Three 2-trees T_1, T_2 and T_3 with same number of external nodes are shown in the figures below. Weights of the nodes are 2, 3, 5 and 11. Determine the external path length L and (external) weighted path length P for each of the tree.

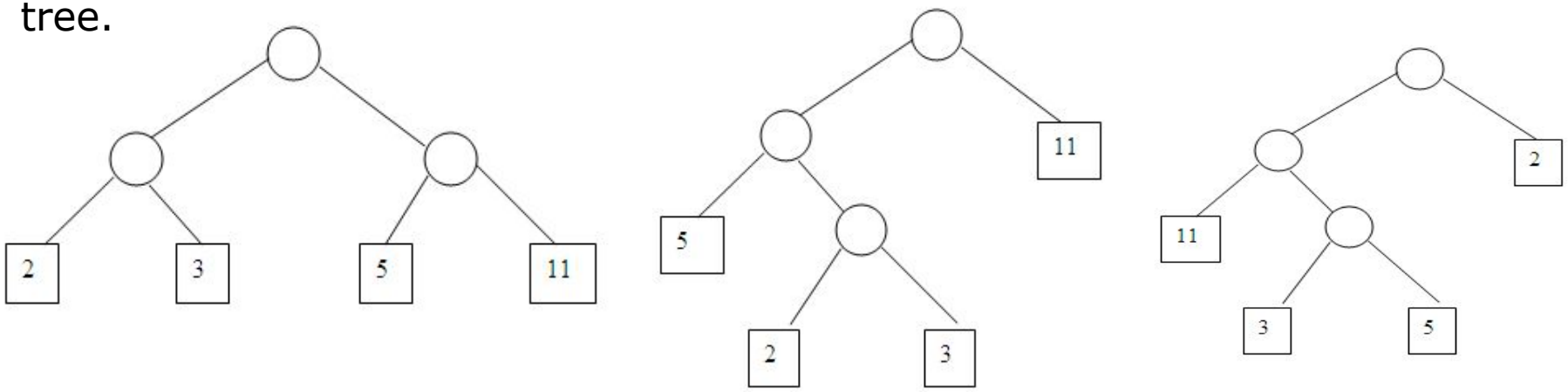


Fig: Three Extended binary tree or 2-tree T_1, T_2 and T_3

The external path length and the weighted path lengths of the above trees are as follows:

$P_1 = 2*2+3*2+5*2+11*2 = 42$ $\quad L_{E1} = 2+2+2+2 = 8$

$P_2 = 2*1+3*3+5*3+11*2 = 48$ $\quad L_{E2} = 2+3+3+1 = 9$

$P_3 = 2*3+3*3+5*2+11*1 = 36$ $\quad L_{E3} = 2+3+3+1 = 9$

- The quantities P_1 and P_3 indicate that the complete tree need not give a minimum path length P, and the quantities P_2 and P_3 indicate that similar trees need not give the same lengths.
- We see that the complete tree has the minimal external path length.
- Among all the 2-trees with n external nodes and given weights, we can find a tree with a minimum-weighted path length using Huffman algorithm.

Huffman Coding:

- Huffman code is a technique for compressing data, which relies on the relative frequency (i.e., the number of occurrences of a symbol) with which different symbols appear in a text.
- The familiar ASCII and UNICODE character-encoding schemes use *fixedlength codes* to represent characters; each character of the ASCII set is represented by an eight-bit pattern, and each UNICODE character is represented by a 16-bit pattern.

- The idea behind Huffman coding is to reduce the space requirement for data by exploiting the fact that some characters are likely to appear more frequently than others.
- In English-language text, for example, the letter 'e' typically occurs far more often than does 'x' or 'z'.

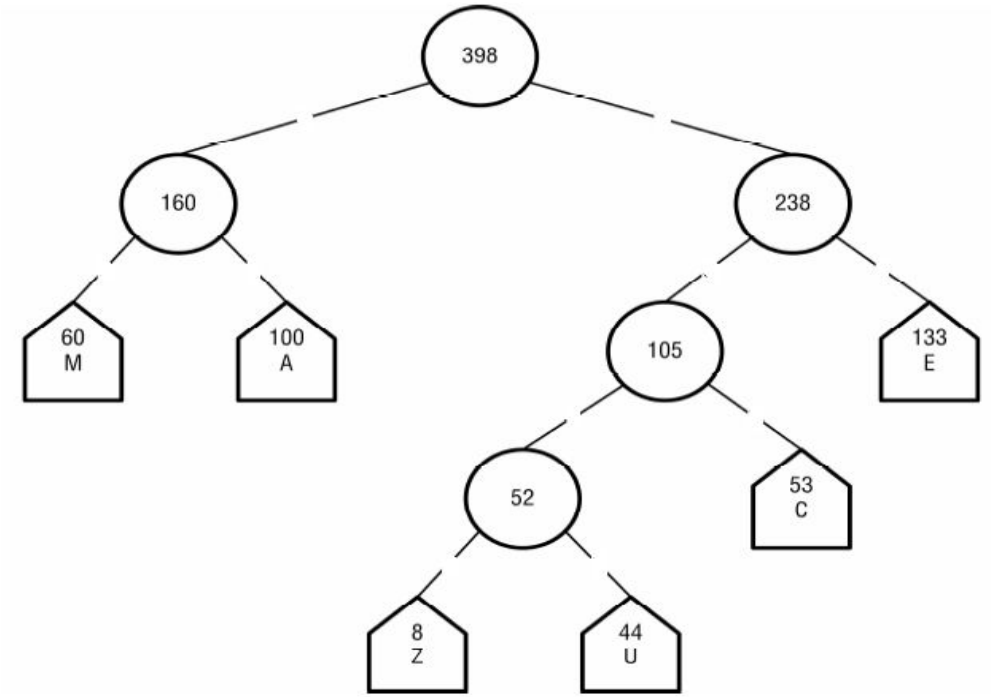
- Huffman coding responds to these different frequencies of occurrence by assigning *variable-length codes* to the various characters:
  - more frequently occurring characters are assigned shorter codes, and
  - letters that occur less frequently are assigned longer codes.

- Huffman coding is often useful for compressing binary data as well as text. The popular MP3 format, for example, uses Huffman coding.

Huffman Tree and Its Properties

- Huffman tree is a binary tree with minimum weighted external path length for a given set of frequencies (weights).

- A Huffman tree is an extended binary tree of integers with two properties:
  1. Each internal node is the sum of its children.
  2. Its weighted external path length is minimal.



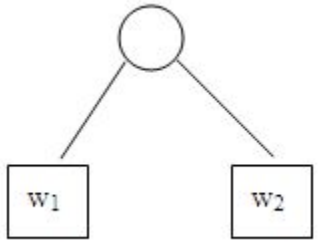
Huffman Algorithm

- We use Huffman algorithm for constructing a Huffman tree and hence Huffman code can be obtained.
- The general problem that we want to solve is as follows:
- Suppose a list of n weights is given:
  $w_1, w_2, w_3, \ldots\ldots w_n$
- Among all the 2-tress with n external nodes and with the given n weights, find a tree with a minimum-weighted path length. Huffman gave an algorithm to find such a tree which is given below.

Suppose w_1 and w_2 are two minimum weights among n given weights w_1, w_2, w_3,, ….., w_n. Huffman algorithm finds a tree which gives a solution for the n-1 weights

$$w_1+w_2,\ w_3,\ w_4, \ldots\ w_n.$$

Then, in the tree, replace the external node w_1+w_2 by the subtree



The new 2-tree is the desired solution.

Constructing a Huffman Coding Tree:

Suppose, the frequency of occurrence of some characters for an English text is shown in the table below.

Let us see how Huffman codes are assigned by using this letter-frequency table.

Here the particular frequencies have been arbitrarily chosen and do not matter; what matters is the frequency of any given letter *relative to* the frequency of any other letter.

Letter	Frequency
A	100
C	53
E	133
M	60
U	44
Z	8

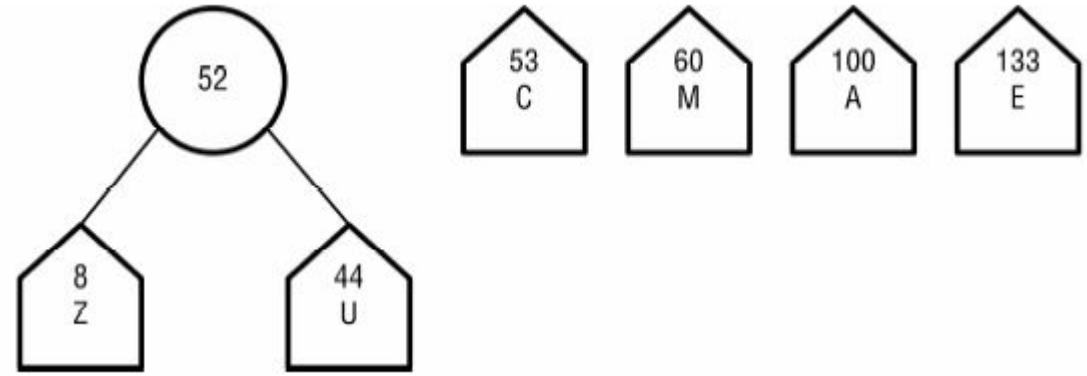
Step-1:

The first step in building a Huffman coding tree is to sort the letters in ascending order by frequency. Each letter will be a leaf in the final tree.

Constructing a Huffman Coding Tree:

Step-2:

Next, the first two letters, those with the lowest frequencies, are removed from the list and become the children of a new internal node. The internal node has no letter, but its frequency is the sum of the frequencies of its children. This new internal node is then placed on the list so as to maintain the list's sorted ordering.

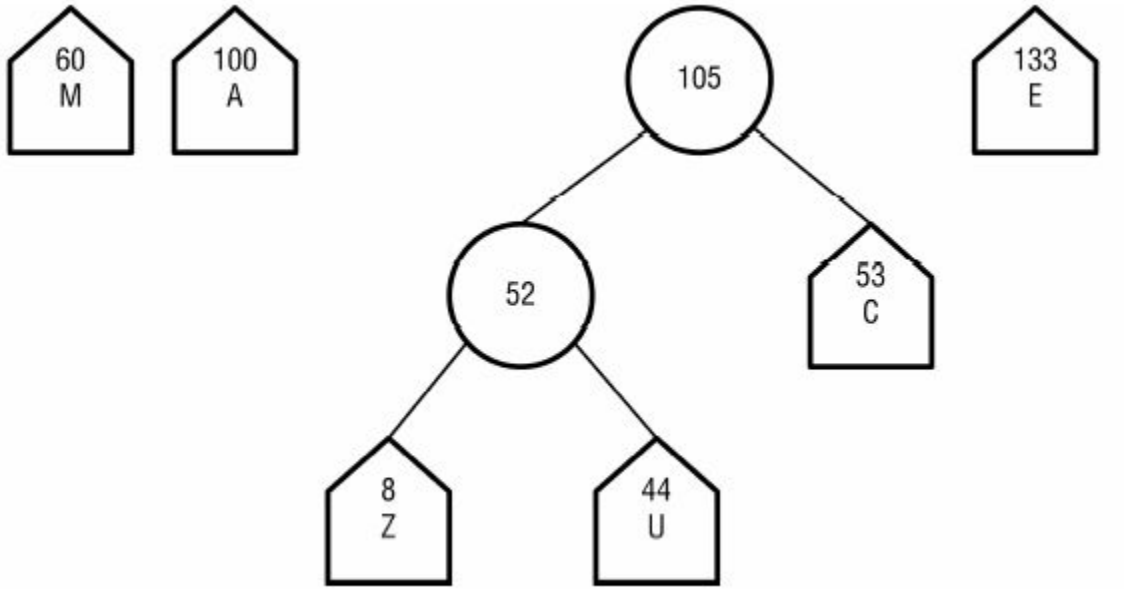


This process continues until the list has only one element, the root of the final Huffman coding tree.

Constructing a Huffman Coding Tree:

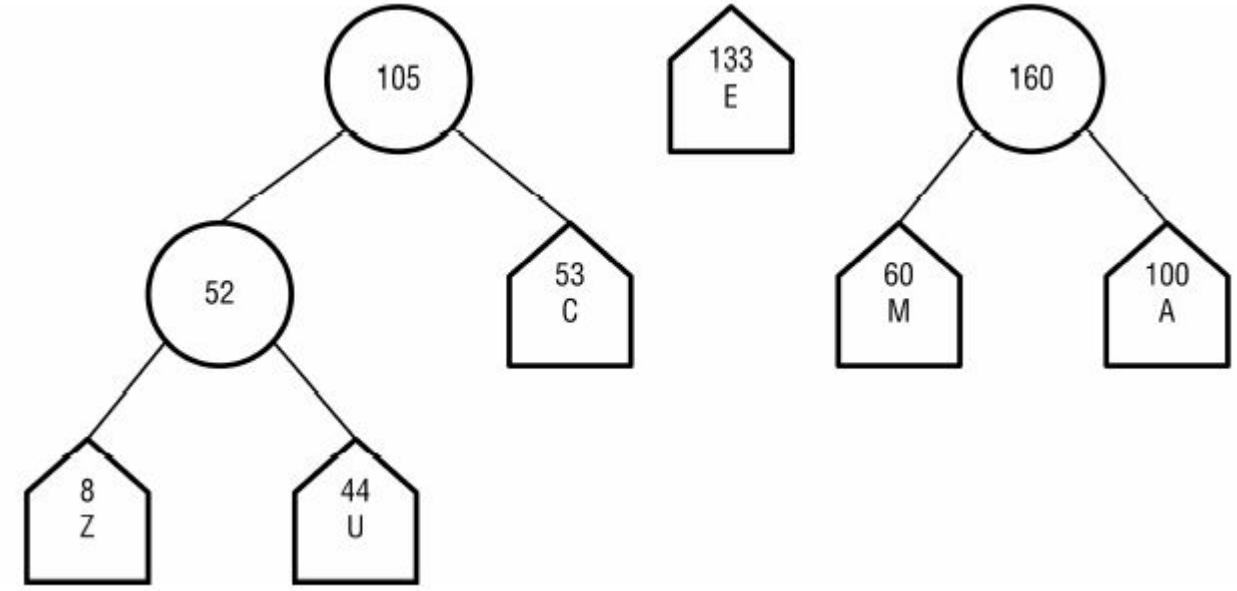
Step-3:

The remaining steps of building our example tree are shown in the following diagrams.



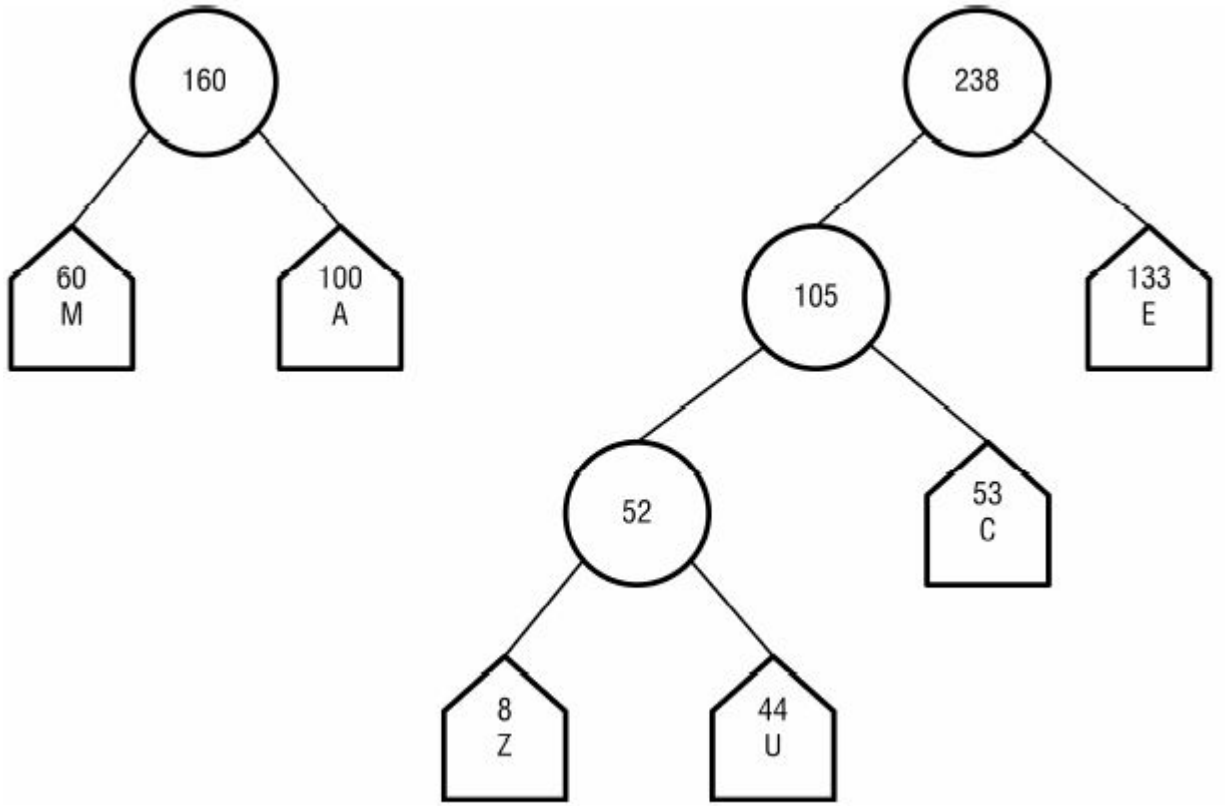
Constructing a Huffman Coding Tree:

Step-4:



Constructing a Huffman Coding Tree:

Step-5:



Constructing a Huffman Coding Tree:

Step-6:

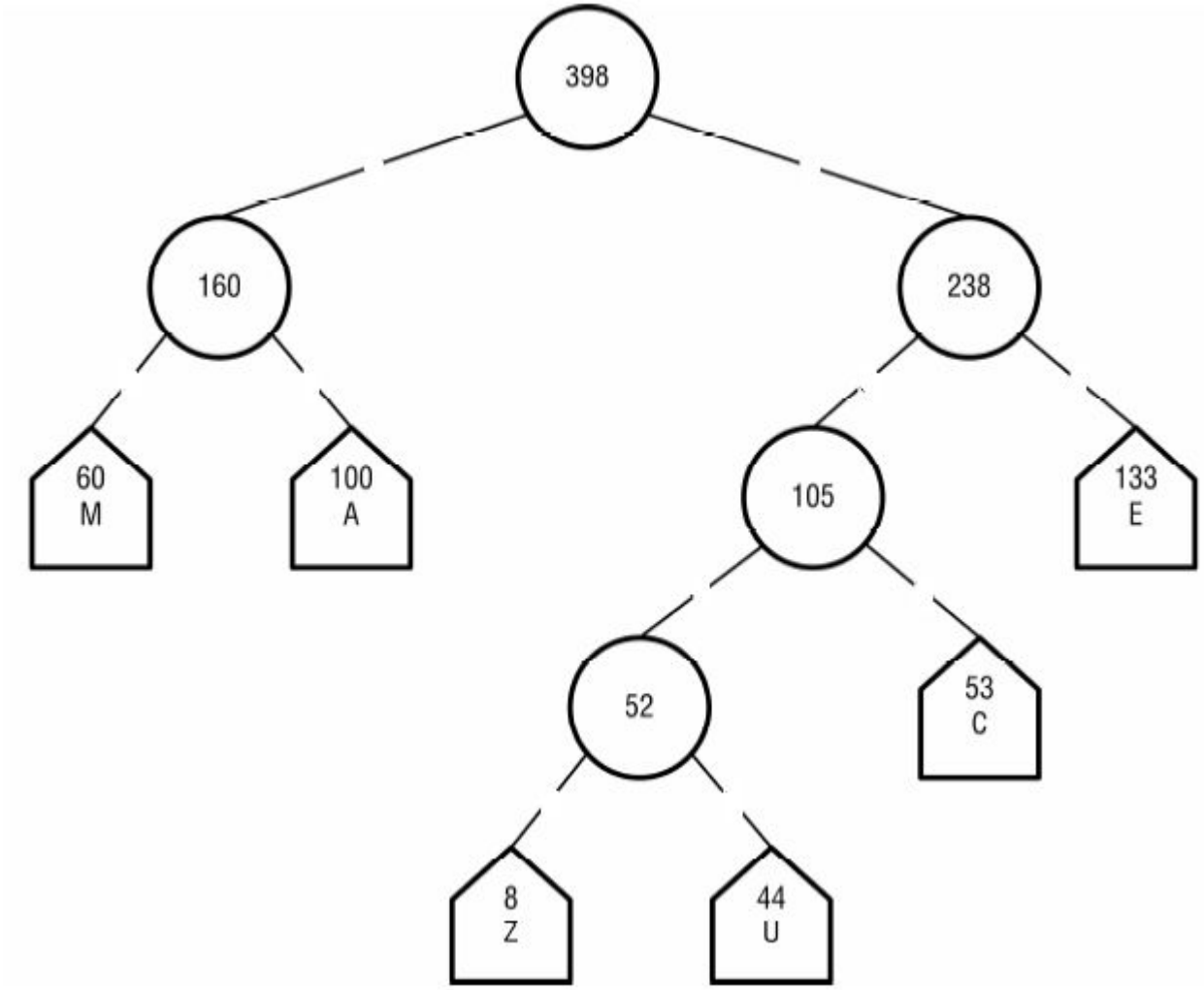


Figure: The Huffman coding tree corresponding to the letter- frequencies

Determining Huffman Code from the Huffman Coding Tree:

- In the Huffman coding tree, label each edge to a left child with a 0, and each edge to a right child with a 1, which is shown below. This labeling of edges is used to determine the Huffman codes.
- The code for a given letter (external node) is the sequence of 0s and 1s encountered on the path from the root to that letter.

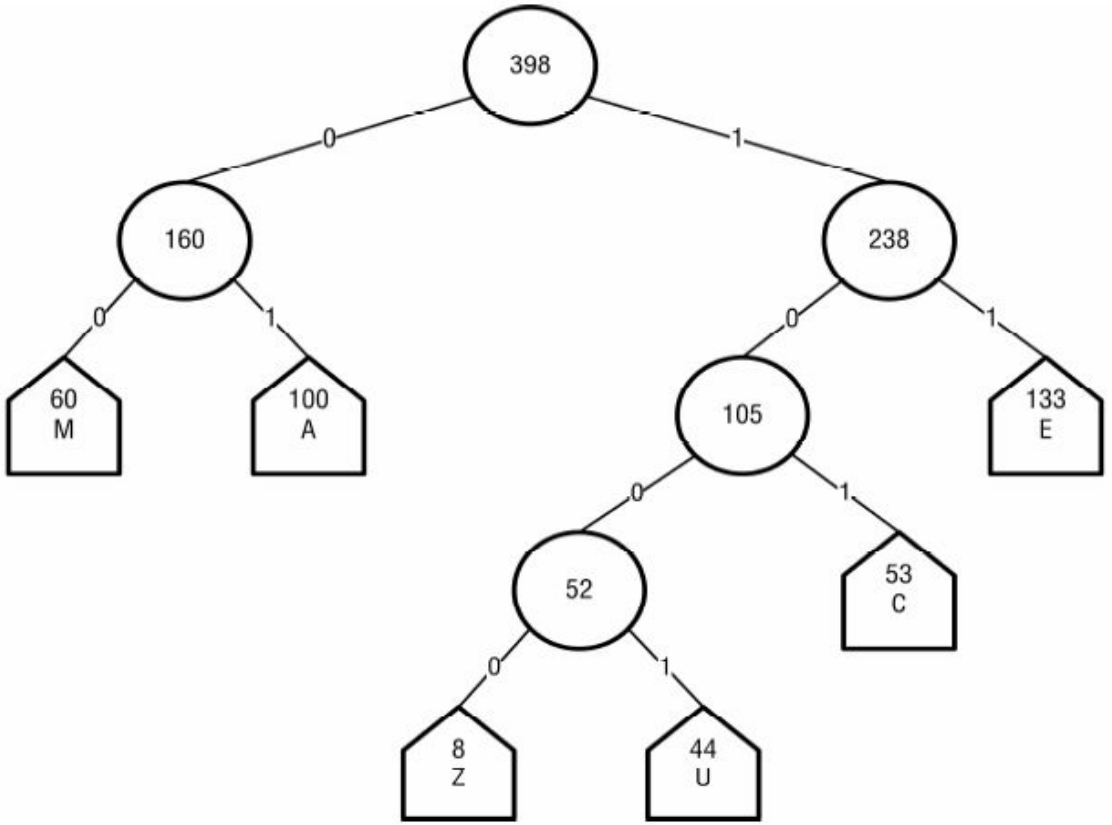


Figure: The Huffman coding tree corresponding to the letter- frequencies

Determining Huffman Code from the Huffman Coding Tree:

- The table below shows the Huffman codes obtained from the Huffman coding tree of the above figure.
- This encoding clearly represents a significant savings compared to an ASCII or UNICODE encoding.

Letter	Frequency	Code	Length
A	100	01	2
C	53	101	3
E	133	11	2
M	60	00	2
U	44	1001	4
Z	8	1000	4

Table: The Huffman codes obtained from the Huffman coding tree

Note:

- The Huffman code has the "prefix" property; i.e., the code of any item is not an initial substring of the code of any other item. This means, there can not be any ambiguity in decoding any message using Huffman code.