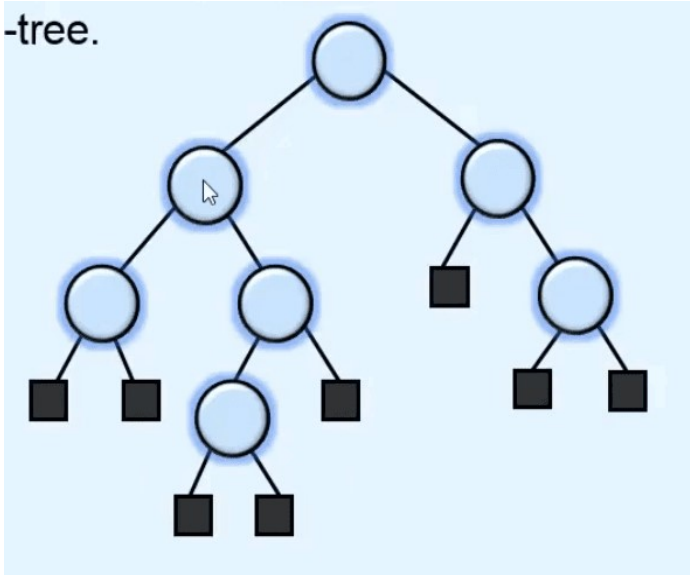
Extended Binary Tree

- Extended binary tree is a type of binary tree in which all the null subtree of the original tree are replaced with special nodes called external nodes whereas other nodes are called internal nodes

- **Properties of External binary tree**

1. The nodes from the original tree are internal nodes and the special nodes are external nodes.
2. All external nodes are leaf nodes and the internal nodes are non-leaf nodes.
3. Every internal node has exactly two children and every external node is a leaf. It displays the result which is a complete binary tree.

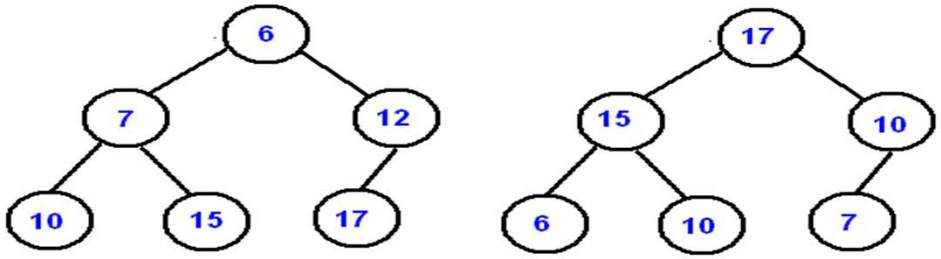


Heaps

A binary heap is a complete or almost complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

The *min-heap property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

The *max-heap property*: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.



Property of Binary Tree:

- Every node has at most two children.

Property of Heap:

- It’s a complete or almost complete binary tree
- Before filling up previous level, it can’t go to the next level.
- Any level fill up from left to right.

Property of Max Heap:

- Root should always be maximum.

Property of Min Heap:

- Root should always be minimum.

Entire tree and subtree will follow the property both for Max Heap and Min Heap.

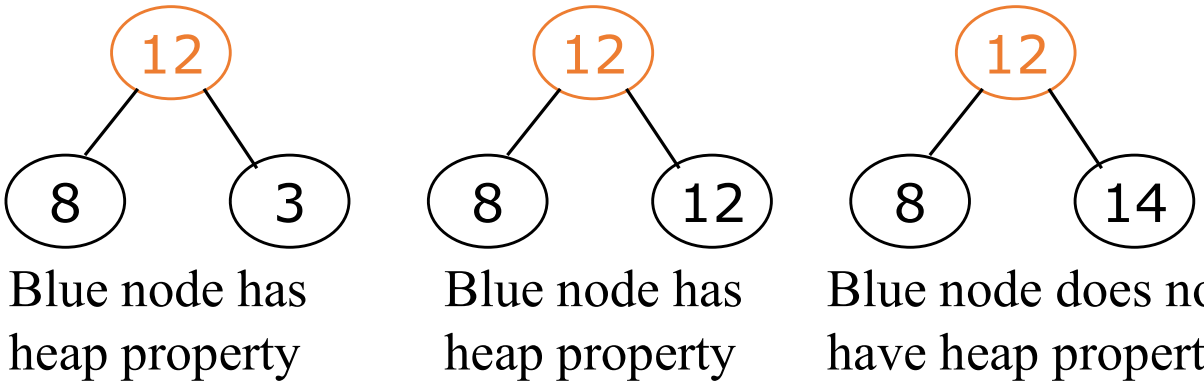
How to implement a Heap?

Though it's a tree, we don't need to use linked list. We can use an array to implement a Heap.

- Root should be the element of first index. Then accordingly fill the array without having a gap.
- Index of parent of a node with index i = [i/2] (Floor Value)
- Index of left child of a node with index i = 2i
- Index of right child of a node with index i = 2i+1

The heap property

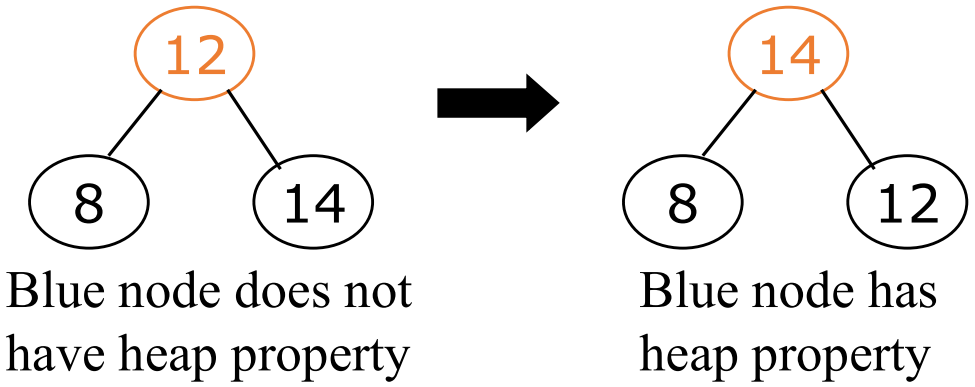
- A node has the heap property if the value in the node is as large as or larger than the values in its children



- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

siftUp

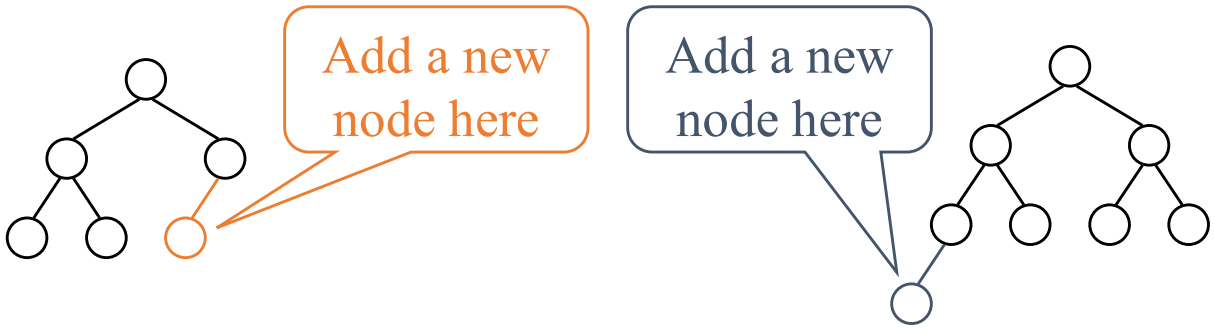
- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



- This is sometimes called sifting up

Constructing a heap I

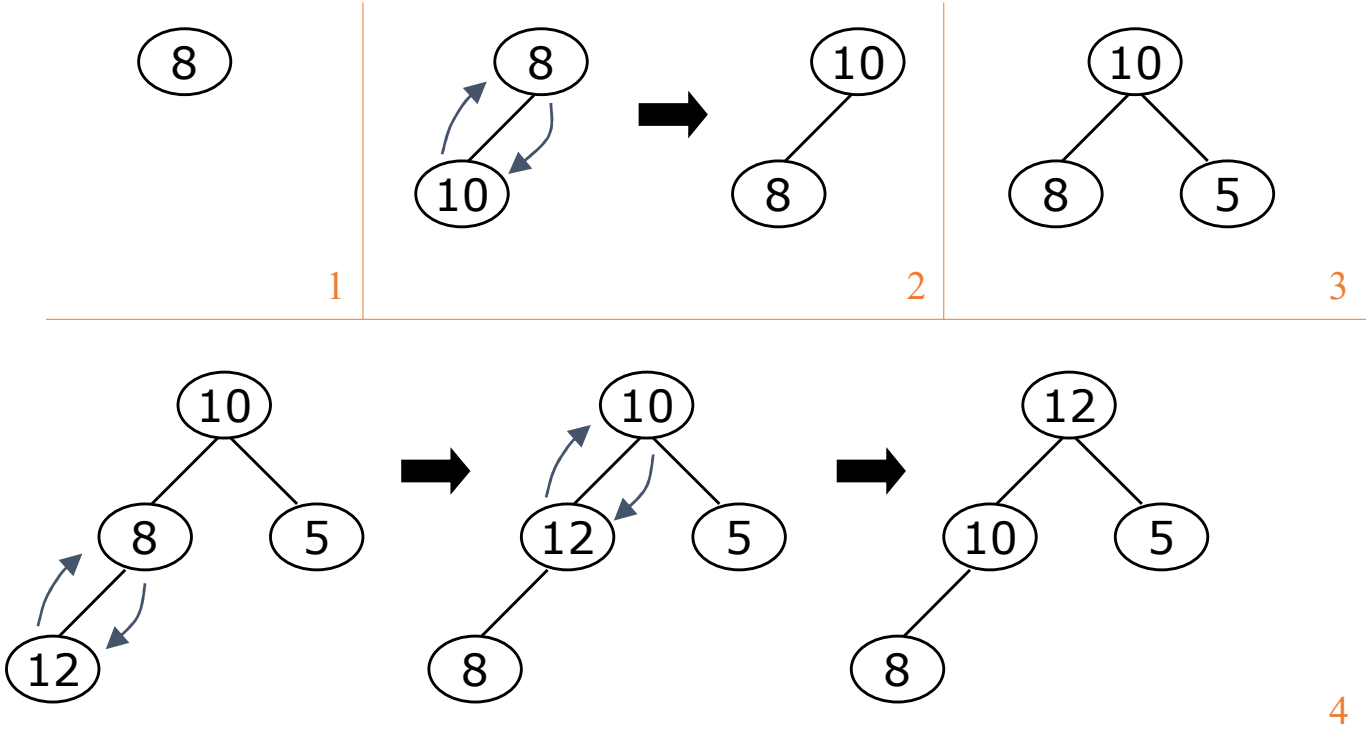
- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- Examples:



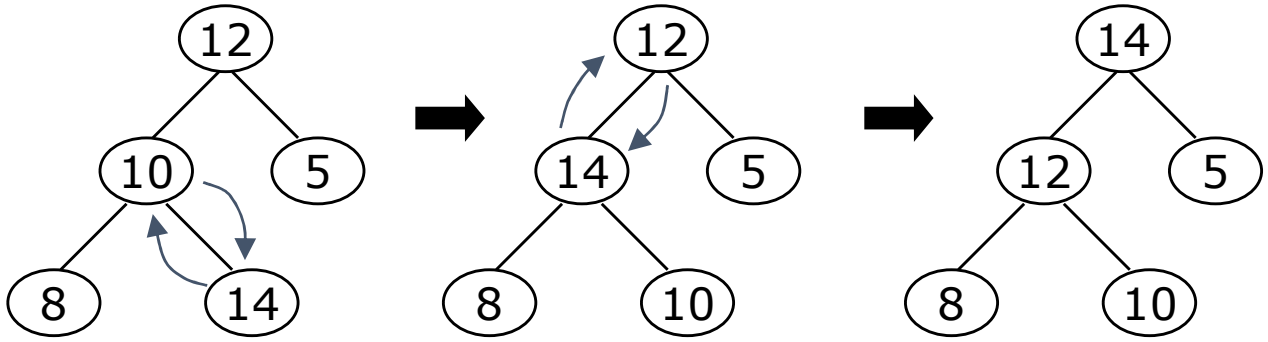
Constructing a heap II

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we sift up
- But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of *its* parent node
- We repeat the sifting up process, moving up in the tree, until either
  - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - We reach the root

Constructing a heap III A[8,10,5,12,14]



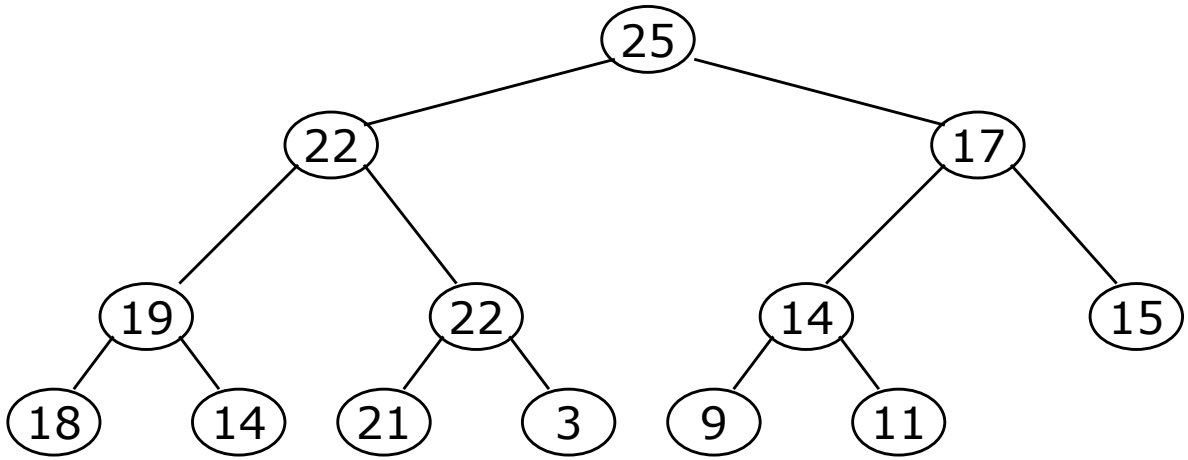
Other children are not affected



- The node containing 8 is not affected because its parent gets larger, not smaller
- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

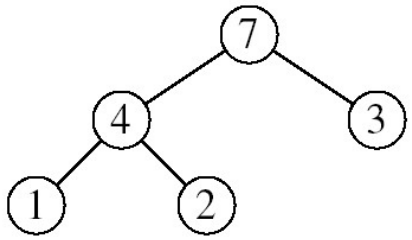
- Here's a sample binary tree after it has been heapified



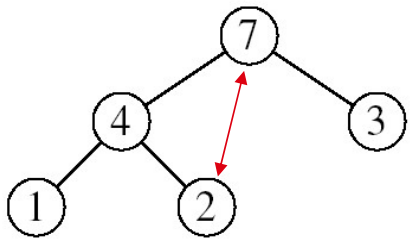
- Notice that heapified does *not* mean sorted
- Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

Heapsort

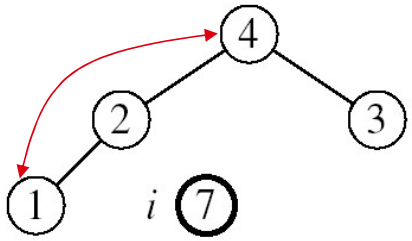
- Goal:
  - Sort an array using heap representations
- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - “Discard” this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains



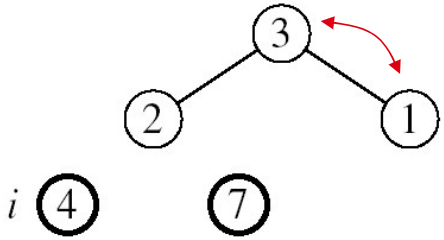
Example: A=[7, 4, 3, 1, 2]



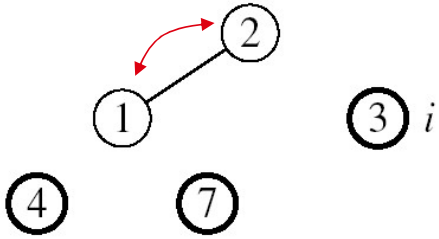
MAX-HEAPIFY(A, 1, 4)



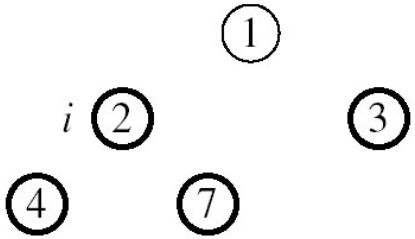
MAX-HEAPIFY(A, 1, 3)

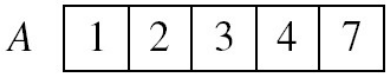


MAX-HEAPIFY(A, 1, 2)



MAX-HEAPIFY(A, 1, 1)





Huffman Algorithm/Huffman coding tree

- Binary tree
  - each leaf contains symbol (character)
  - label edge from node to left child with 0
  - label edge from node to right child with 1

Building a Huffman tree

- Find frequencies of each symbol occurring in message
- Begin with a forest of single node trees
  - each contain symbol and its frequency
- Do recursively
  - select two trees with smallest frequency at the root
  - produce a new binary tree with the selected trees as children and store the sum of their frequencies in the root
- Recursion ends when there is one tree
  - this is the Huffman coding tree

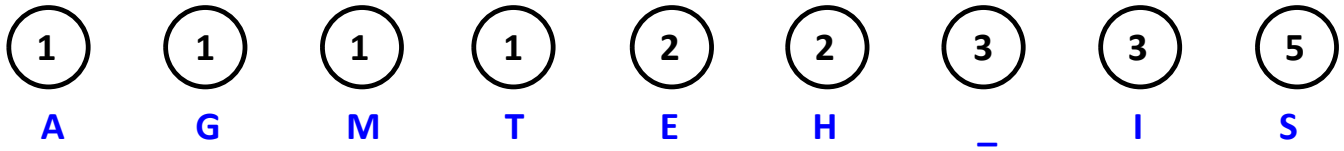
Example

- Build the Huffman coding tree for the message

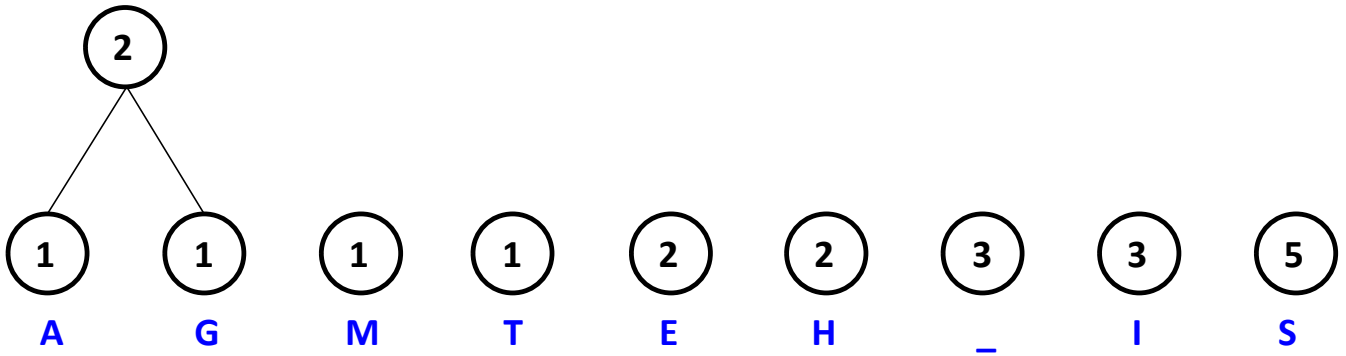
  This is his message

- Character frequencies

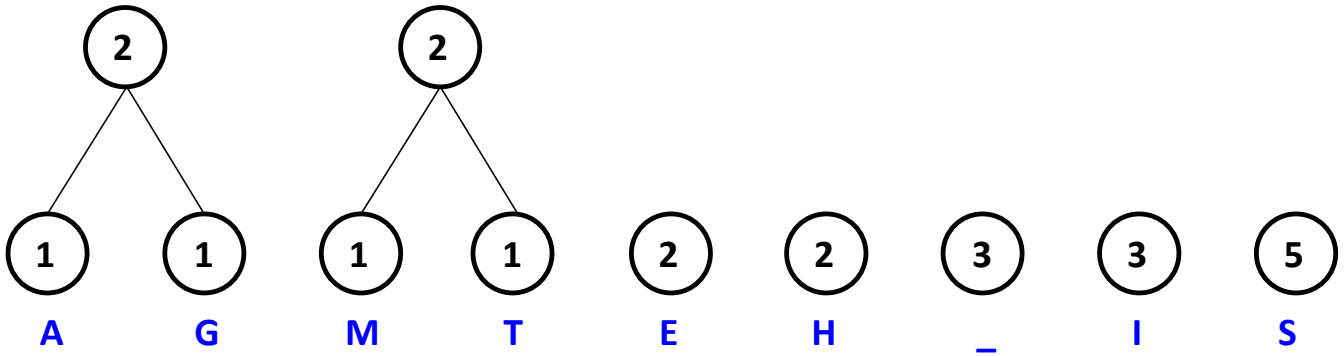
A	G	M	T	E	H	_	I	S
1	1	1	1	2	2	3	3	5



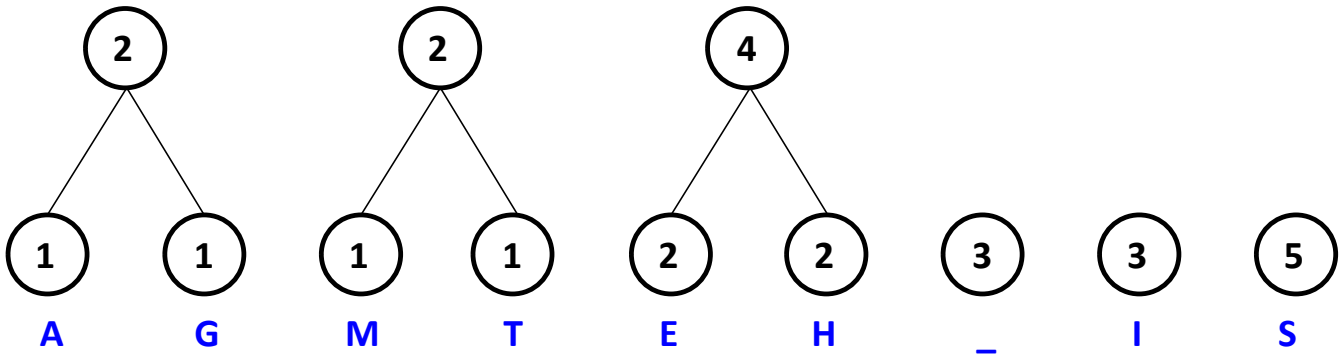
Step 1



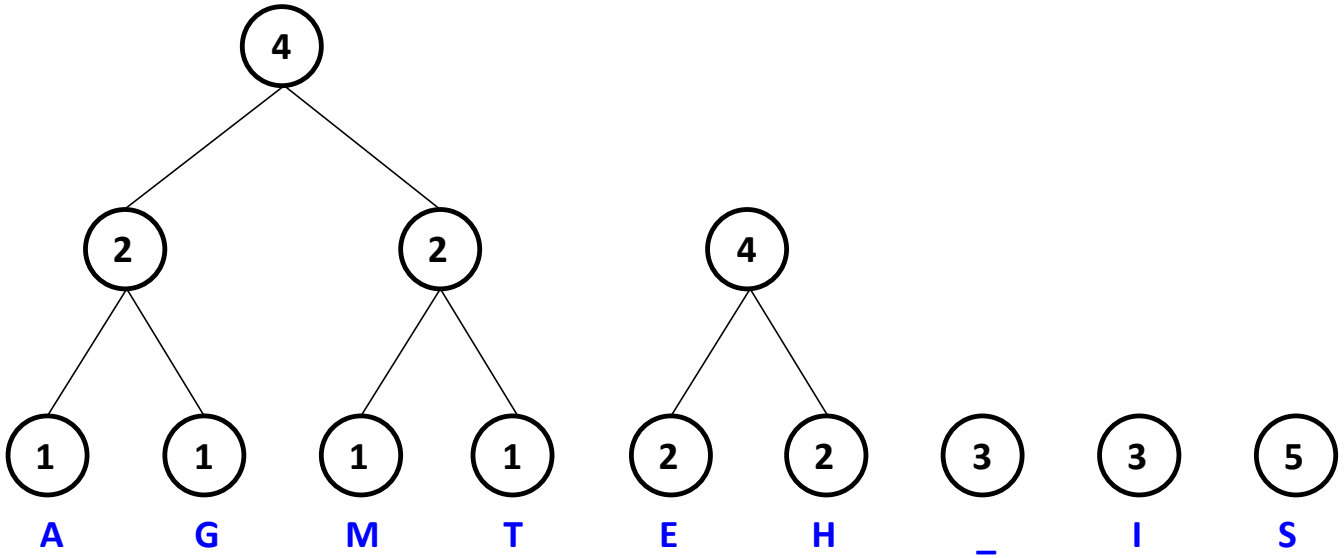
Step 2



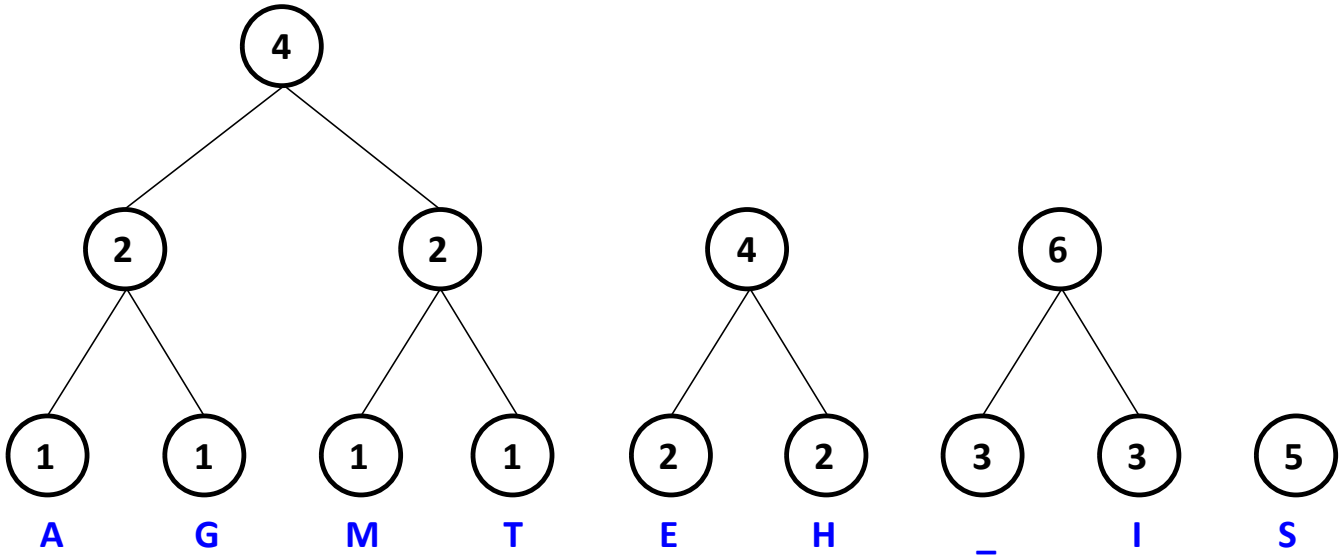
Step 3



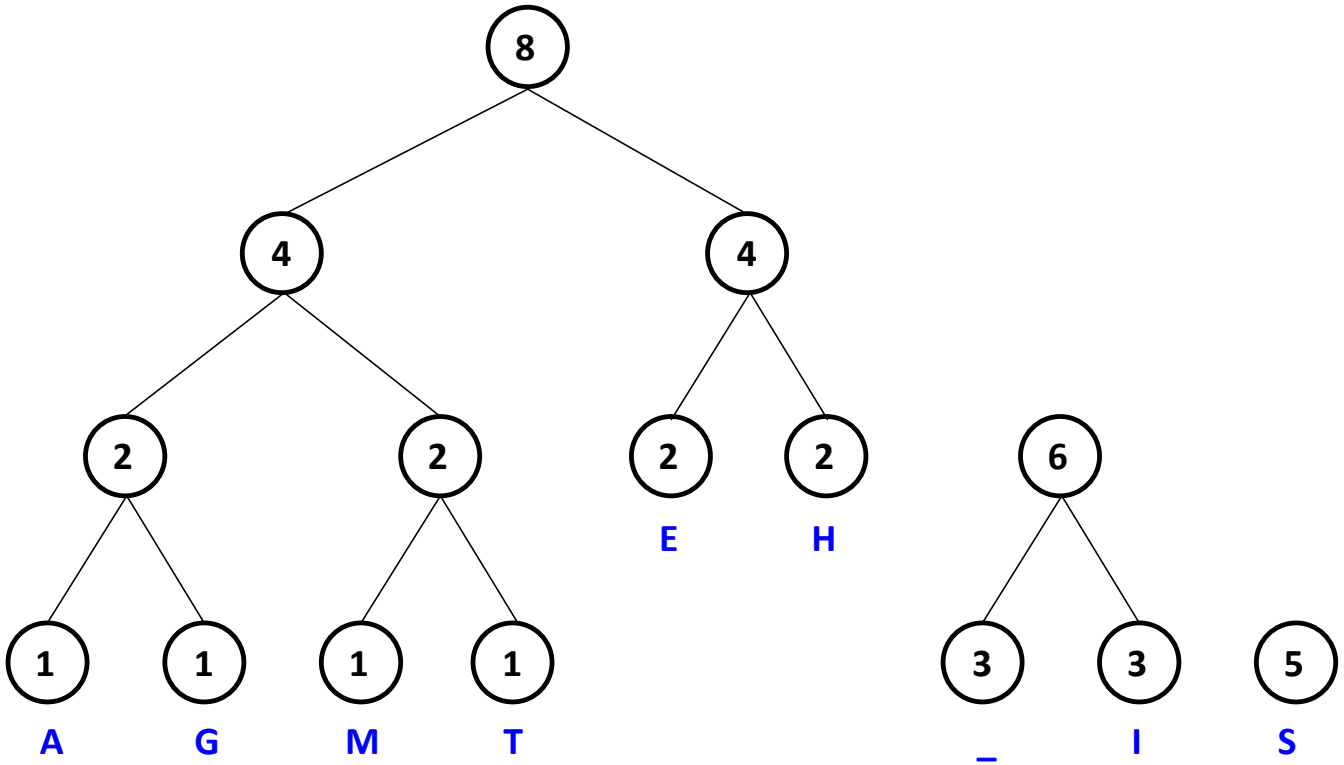
Step 4



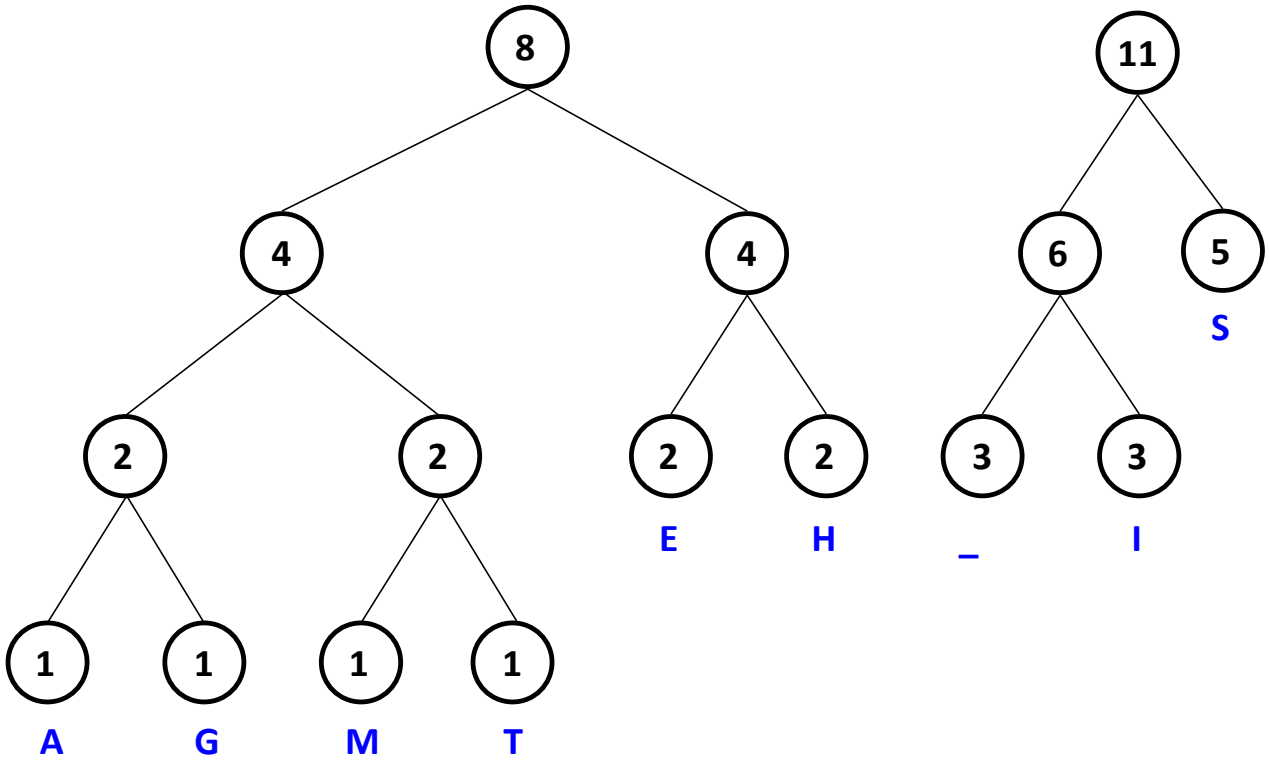
Step 5



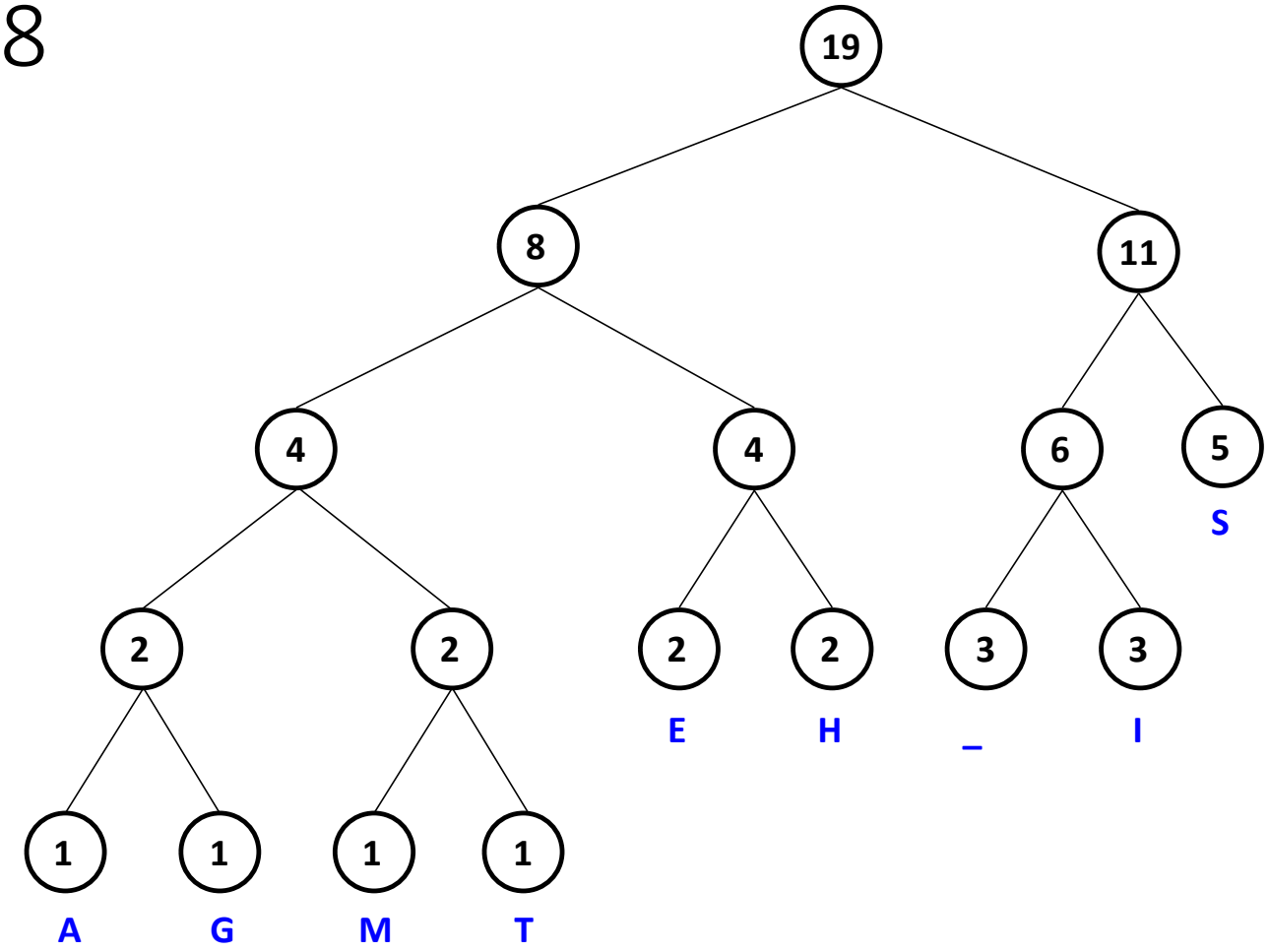
Step 6



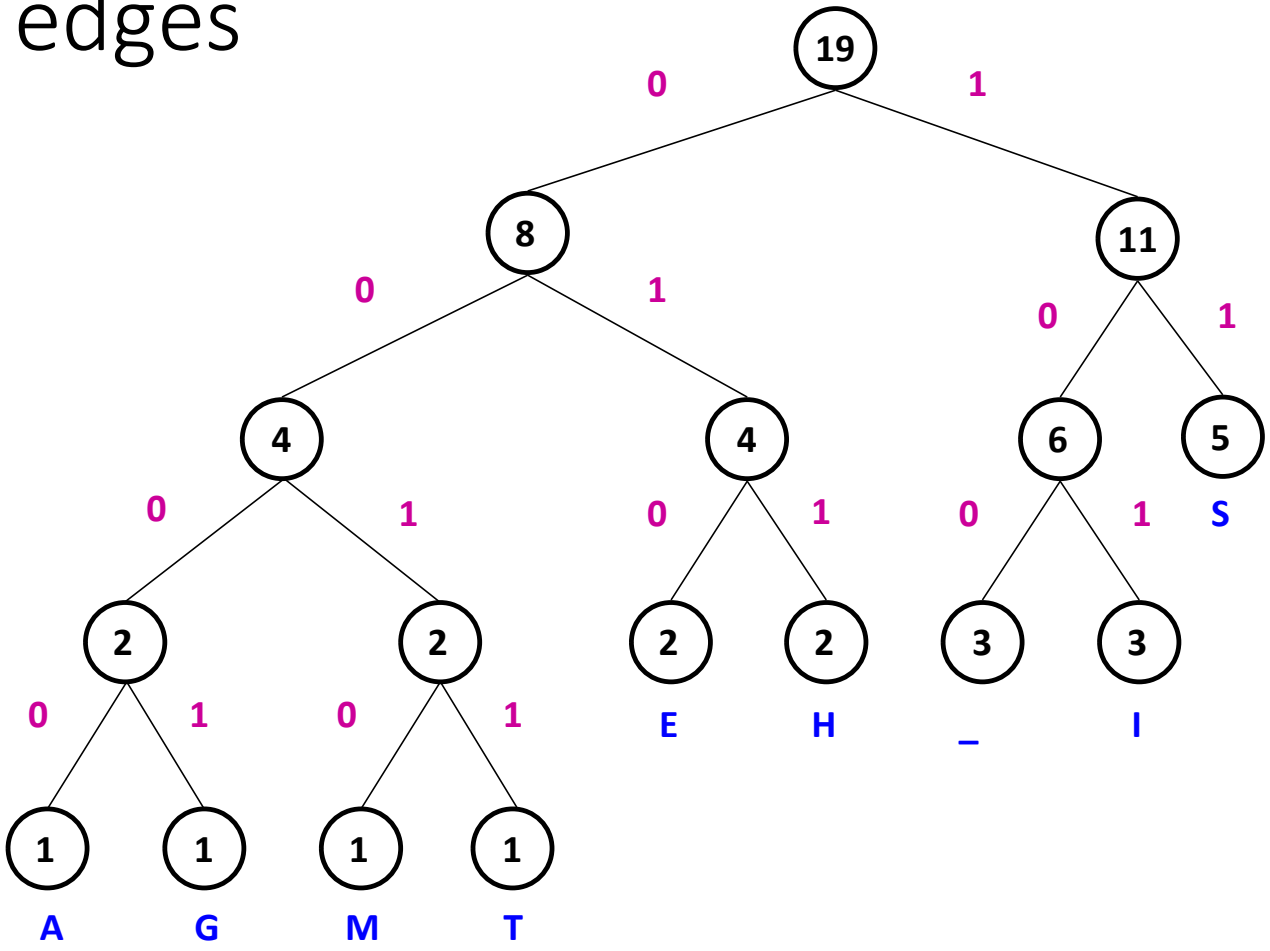
Step 7



Step 8



Label edges



Huffman code & encoded message

This is his message

S	11
E	010
H	011
_	100
I	101
A	0000
G	0001
M	0010
T	0011

00110111011110010111100011101111000010010111100000001010