Greedy Algorithms

At each step of the solution, pick the best choice given the information currently available (i.e., greedily). This often leads to very efficient solutions to optimization problems. However, not all problems have greedy solutions.

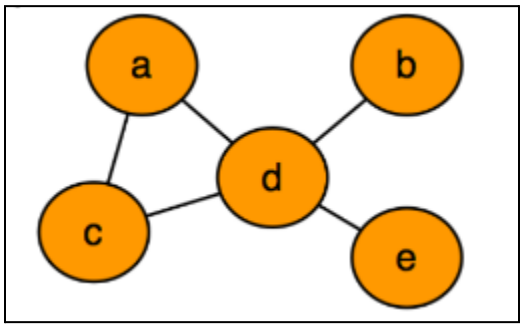
Spanning Tree and MST:

A subgraph T of an undirected graph G = (V, E) is a spanning tree of G if it is

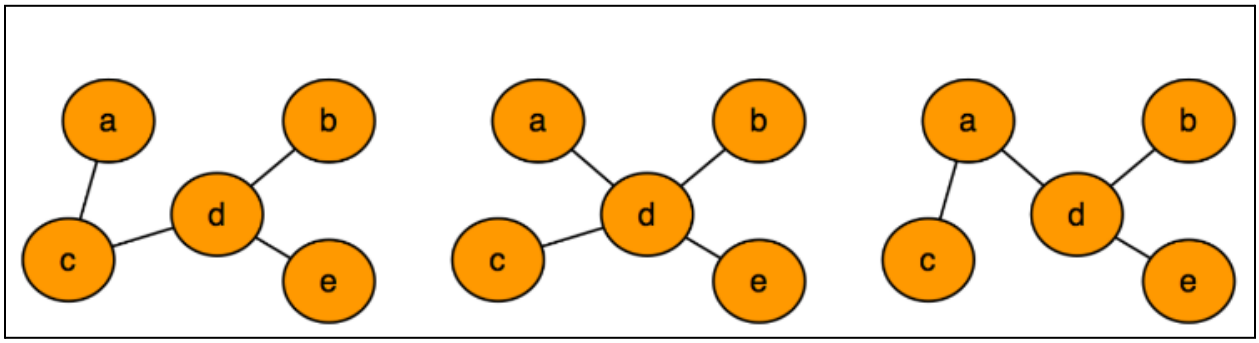
- A Tree (connected, n-1 edges and no cycle)
- Contains every vertex of G.

There can be many possible spanning trees for a graph.

Given the following graph:

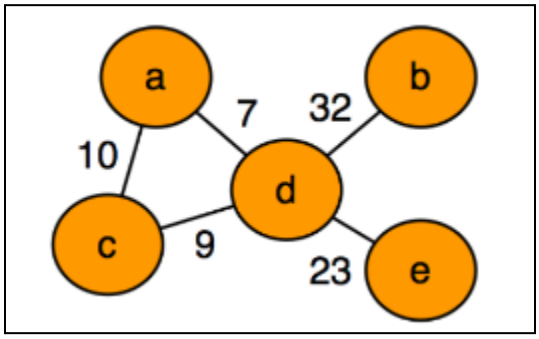


The spanning trees are:

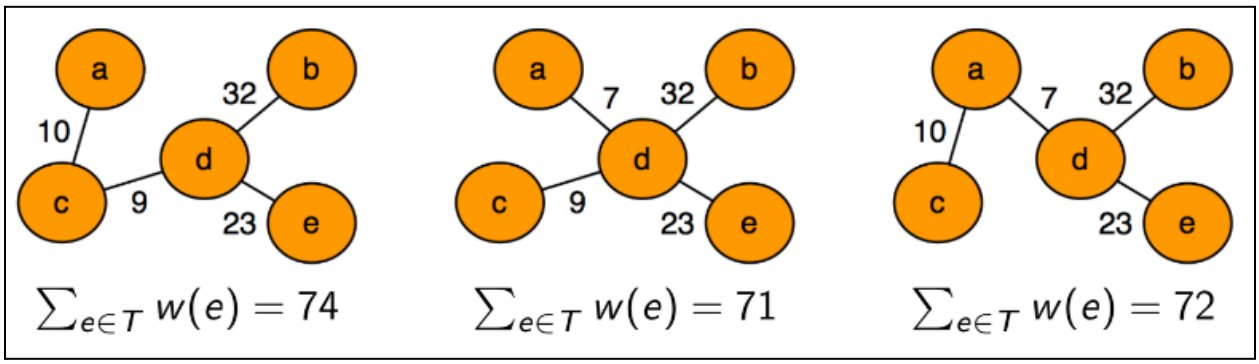


Spanning Trees of Weighted Graph:

Given the following graph:



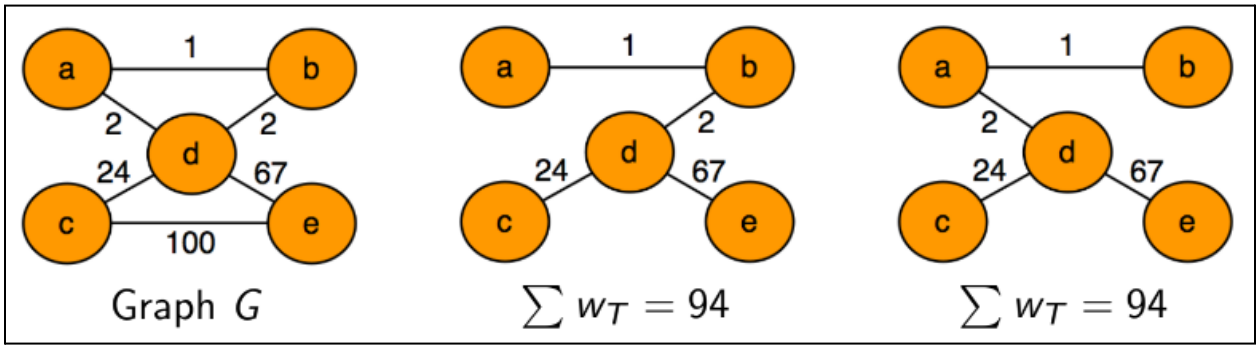
The spanning trees with associated total weights are:



Minimum-cost Spanning Tree (MST):

A minimum spanning tree (MST) is defined as a spanning tree that has the minimum total weight among all the possible spanning trees.

The minimum-cost spanning tree may not be unique.



How to compute MST?

We grow the tree one edge at a time, starting with a graph G' = (V, ∅). Add a new safe edge at each step, ensuring it does not create a cycle. If adding an edge guarantees that the tree after each step is a subset of some MST, the final result will be an MST.

This follows the Greedy approach. There are two popular algorithms through which MST can be computed:

- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm: [Optional]

Prim's algorithm to find the minimum cost-spanning tree uses the greedy approach. The Greedy Choice is to start with a node and pick the smallest weight edge and build the tree until all the vertices are in the tree.

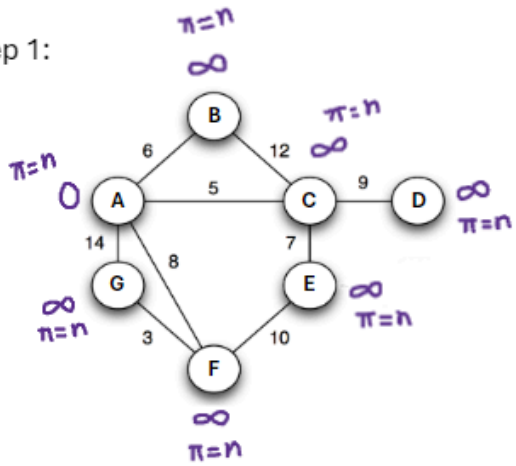
Pseudo Code:

```
MST-PRIM(G, w, r)
 1  for each u ∈ V[G]
 2        do key[u] ← ∞
 3           π[u] ← NIL
 4  key[r] ← 0
 5  Q ← V[G]
 6  while Q ≠ ∅
 7        do u ← EXTRACT-MIN(Q)
 8           for each v ∈ Adj[u]
 9                 do if v ∈ Q and w(u, v) < key[v]
10                       then π[v] ← u
11                            key[v] ← w(u, v)
```

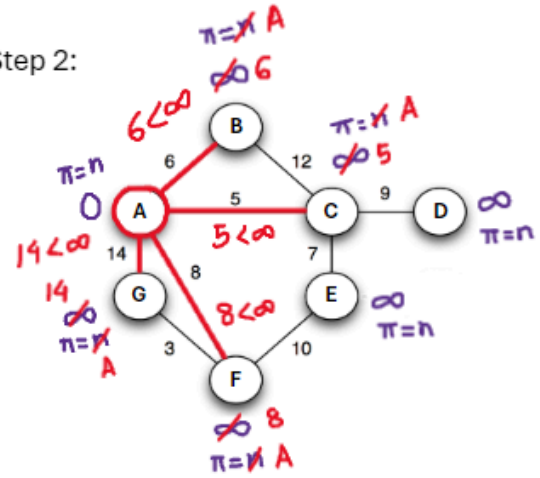
The tree starts from an arbitrary root vertex r and grows until it spans all the vertices in V. We will use a binary min-heap data structure to implement the priority queue.

Simulation:

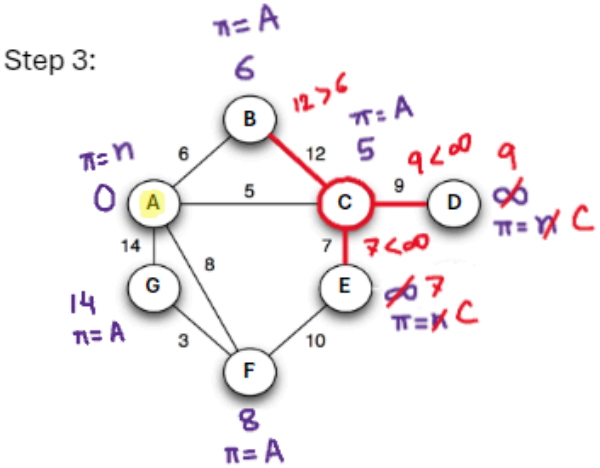
Step 1:



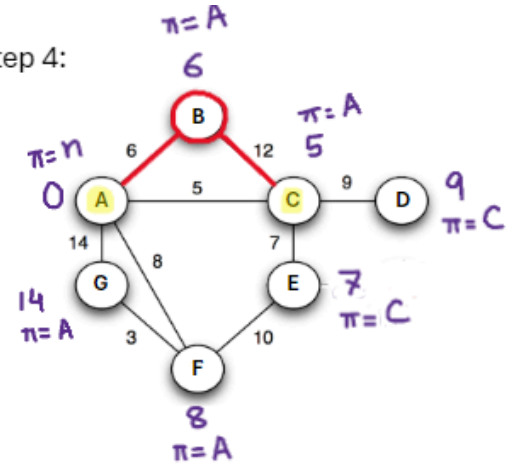
Step 2:



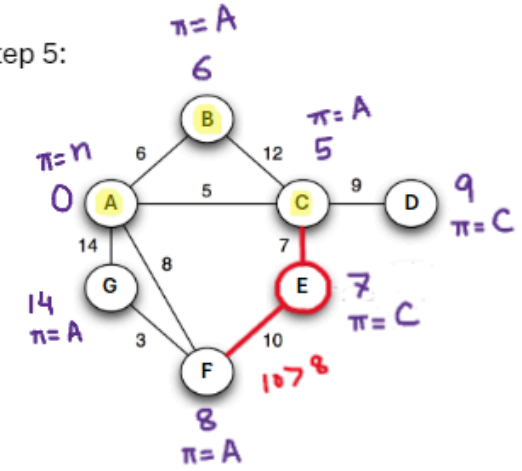
Step 3:



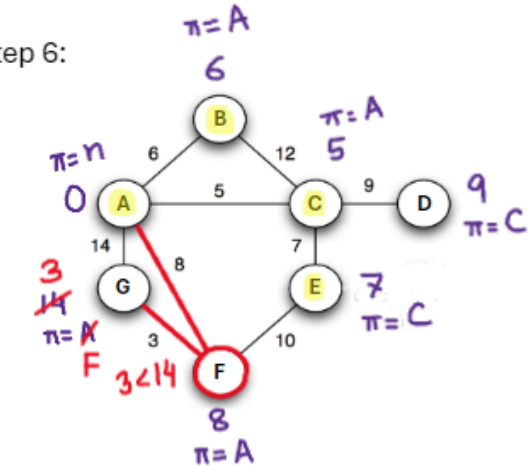
Step 4:



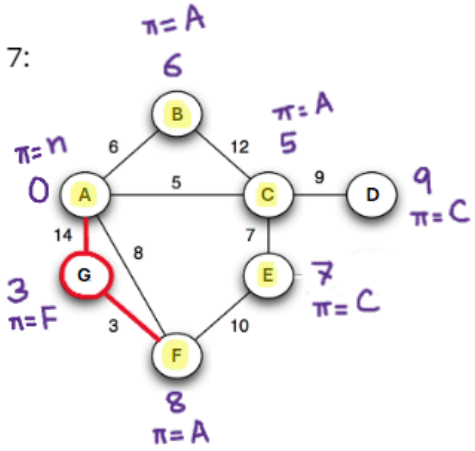
Step 5:



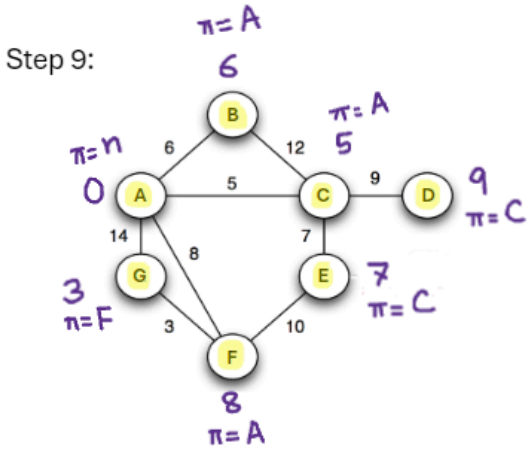
Step 6:



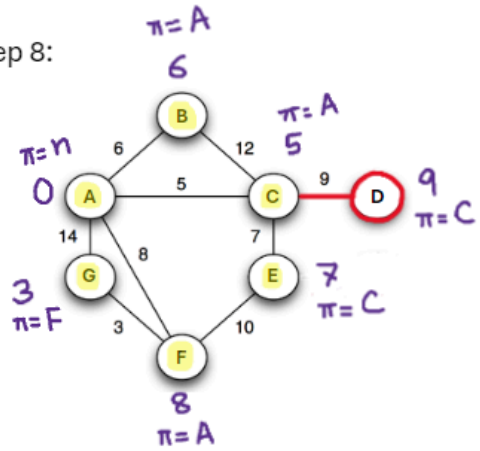
Step 7:



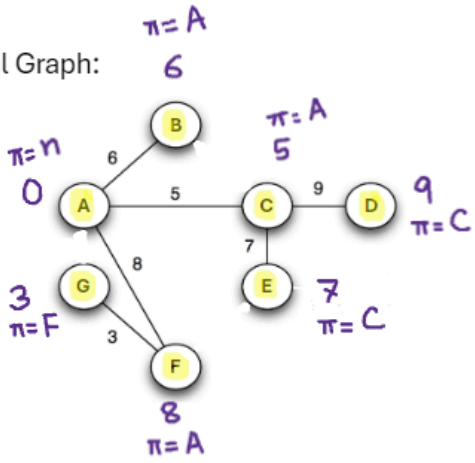
Step 9:



Step 8:



Final Graph:



Time Complexity Analysis:

We will use a binary min-heap data structure to perform Prim's algorithm. A min-heap is a tree-based data structure with the property that every node has a smaller key than its children.

T(findMin) = O(1)
T(removeMin) = O(log(V))
T(updateKey) = O(log(V))

We need to findMin at most V vertices so O(1) * O(V) = O(V)
We need to removeMin at most V vertices so O(logV) * O(V) = O(VlogV)
We need to updateKey at most E edges so O(logV) * O(E) = O(ElogV)
Total = O(V) + O(VlogV) + O(ElogV)

Time Complexity = O((V + E) * log(V))
= O(ElogV) as E is V^2 at worst case

Kruskal's Algorithm:

Kruskal's algorithm is also a greedy approach to find the minimum cost spanning tree. The Greedy Choice is to pick the smallest-weight edge that does not cause a cycle in the MST constructed so far.

Pseudo Code:

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3        do MAKE-SET(v)
4  sort the edges of E into non-decreasing order by weight w
5  for each edge (u, v) ∈ E, taken in non-decreasing order by weight
6        do if FIND-SET(u) ≠ FIND-SET(v)
7              then A ← A ∪ {(u, v)}
8                   UNION(u, v)
9  return A
```

A data structure that stores a non-overlapping or disjoint subset of elements is called a disjoint set data structure. The main operations are:

- MAKE-SET(u): Create a new set containing a single element u.
- FIND-SET(u): Determine the subset containing the element u. This can determine if two elements are in the same subset.
- UNION(u, v): Join two subsets containing u and v into a single subset. Here first we have to check if the two subsets belong to the same set. If not, then we cannot perform union.

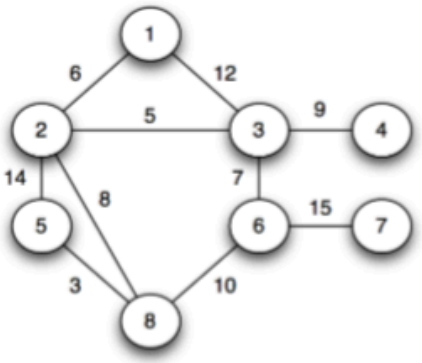
The time complexity of the operations depends on the implementation.

The Disjoint set can be used to check if adding an edge to a graph will create a cycle; hence, it is used in Kruskal's algorithm.

Simulation:

Step 1:

$w(u,v)$	(u,v)
3	(5,8)
5	(2,3)
6	(1,2)
7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)

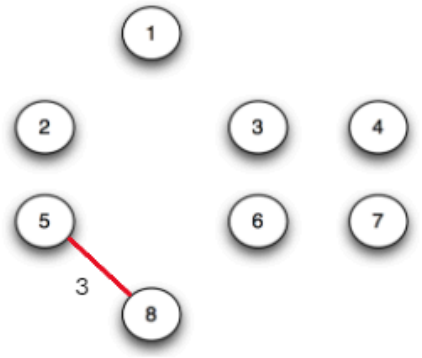


$|V| = 8$
$|E| = 10$
$|T| = 0$

Vertex sets:
{1},{2},{3},{4},{5},{6},{7},{8}

Step 2:

$w(u,v)$	(u,v)
3	(5,8)
5	(2,3)
6	(1,2)
7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)

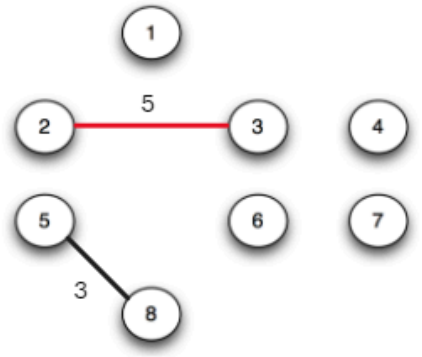


$|V| = 8$
$|E| = 10$
$|T| = 1$

Vertex sets:
{1},{2},{3},{4},{5},{6},{7},{8} ⟹ {1},{2},{3},{4},{5, 8},{6},{7}

Step 3:

$w(u,v)$	(u,v)
✓ 3	(5,8)
5	(2,3)
6	(1,2)
7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)

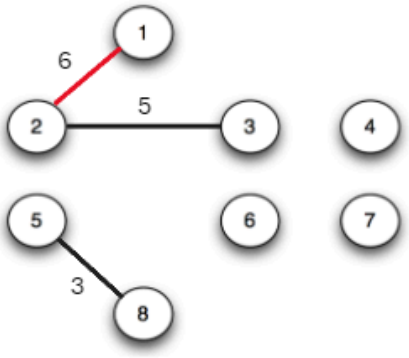


$|V| = 8$
$|E| = 10$
$|T| = 2$

Vertex sets:
{1},{2},{3},{4},{5, 8},{6},{7} ⟹ {1},{2, 3},{4},{5, 8},{6},{7}

Step 4:

$w(u,v)$	(u,v)
✓ 3	(5,8)
✓ 5	(2,3)
6	(1,2)
7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)

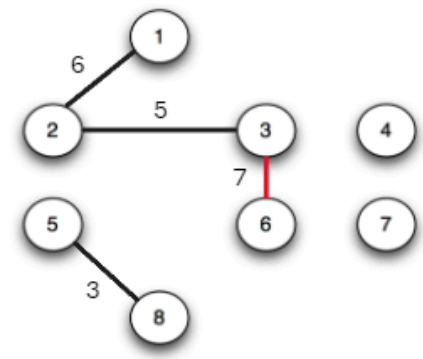


$|V| = 8$
$|E| = 10$
$|T| = 3$

Vertex sets:
{1},{2, 3},{4},{5, 8},{6},{7} ⟹ {1, 2, 3},{4},{5, 8},{6},{7}

Step 5:

$w(u,v)$	(u,v)
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)

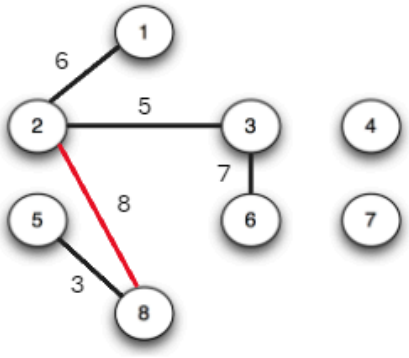


$|V| = 8$
$|E| = 10$
$|T| = 4$

Vertex sets:
{1, 2, 3},{4},{5, 8},{6},{7} ⟹ {1, 2, 3, 6},{4},{5, 8},{7}

Step 6:

$w(u,v)$	(u,v)
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)

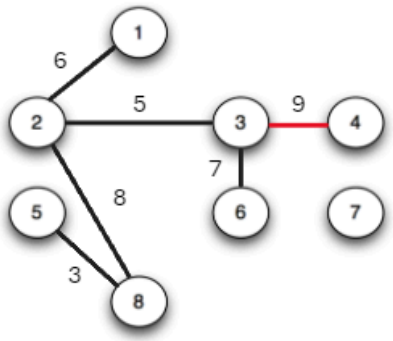


$|V| = 8$
$|E| = 10$
$|T| = 5$

Vertex sets:
{1, 2, 3, 6},{4},{5, 8},{7} ⟹ {1, 2, 3, 5, 6, 8},{4},{7}

Step 7:

$w(u,v)$	(u,v)
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
✓ 8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)

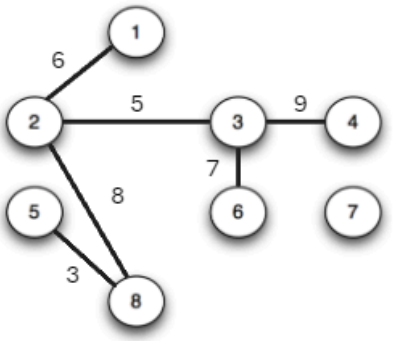


$|V| = 8$
$|E| = 10$
$|T| = 6$

Vertex sets:
{1, 2, 3, 5, 6, 8},{4},{7} ⟹ {1, 2, 3, 4, 5, 6, 8},{7}

Step 8:

$w(u,v)$	(u,v)
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
✓ 8	(2,8)
✓ 9	(3,4)
× 10	(6,8)
× 12	(1,3)
× 14	(2,5)
15	(6,7)

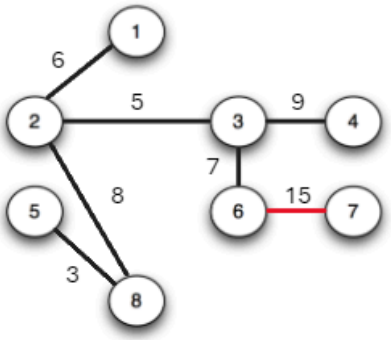


$|V| = 8$
$|E| = 10$
$|T| = 6$

Vertex sets:
{1, 2, 3, 5, 6, 8},{4},{7} ⟹ {1, 2, 3, 4, 5, 6, 8},{7}

Step 9:

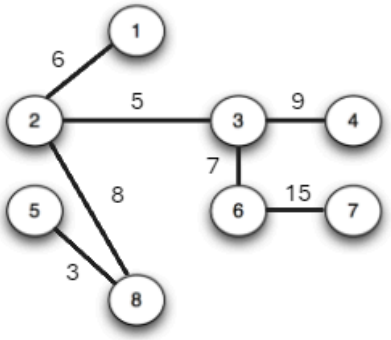
$w(u,v)$	(u,v)
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
✓ 8	(2,8)
✓ 9	(3,4)
× 10	(6,8)
× 12	(1,3)
× 14	(2,5)
15	(6,7)



$|V| = 8$
$|E| = 10$
$|T| = 7$

Vertex sets:
{1, 2, 3, 4, 5, 6, 8},{7} ⟹ {1, 2, 3, 4, 5, 6, 7, 8}

Final Graph:



Time Complexity Analysis:

We will use a disjoint set data structure to perform Prim's algorithm.

T(MAKE-SET) = O(1)
T(FIND-SET) = O(α(V)) where α(V) is very small and can be considered constant
T(UNION) = O(α(V)) where α(V) is very small and can be considered constant

We need to sort E edges by weight so O(ElogE)
We need to MAKE-SET for each vertex so O(1) * O(V) = O(V)
We need to FIND-SET at most E vertices so O(α(V)) * O(E) = O(Eα(V))
We need to UNION at most E vertices so O(α(V)) * O(E) = O(Eα(V))
Total = O(ElogE) + O(Eα(V)) + O(Eα(V))

Time Complexity = O(ElogE)

Prim's Algorithm is particularly efficient when the graph is dense (many edges), as it builds the MST by expanding from a starting node and adding the shortest edge that connects a new node to the growing MST.

Kruskal's Algorithm is often preferred for sparse graphs (fewer edges), as it sorts all the edges and then adds the shortest available edge to the MST, ensuring no cycles are formed.

Huffman Coding:

Huffman Coding is a technique of compressing data to reduce its size without losing any details.

Problem: Not all characters occur with the same frequency. Yet all characters are allocated the same amount of space. According to ASCII, 1 character = 1 byte or 8 bits.

Solution: The idea is to assign variable-length codes to the characters, lengths of the assigned codes are based on the frequencies of corresponding characters. More frequently used characters have shorter codes. By reducing the length of codes for the characters that appear most often, the overall size of the encoded data is minimized.

For example, if the input sentence is:

Eerie eyes seen near lake.

Each character occupies 8 bits. There are a total of 26 characters. Thus, a total of 8*26 = 208 bits is required. We can compress the string to a smaller size using the Huffman Coding technique.

Huffman coding first creates a tree using the frequencies of the characters and then generates code for each character.

Character	Frequency	Character	Frequency
E	1	s	2
e	8	n	2
r	2	a	2
i	1	l	1
space	4	k	1
y	1	.	1

Basic Algorithm:

1. Scan text to be compressed and tally occurrence of all characters.

2. Sort or prioritize characters based on number of occurrences in text.

3. Build Huffman code tree based on prioritized list.

4. Perform a traversal of tree to determine all code words.

5. Scan text again and create new file using the Huffman codes.

Building a Tree:

1. Create binary tree nodes with character and frequency of each character

2. Place nodes in a priority queue (The lower the occurrence, the higher the priority in the queue)

3. While priority queue contains two or more nodes:

    i. Create new node

    ii. Dequeue node and make it left subtree

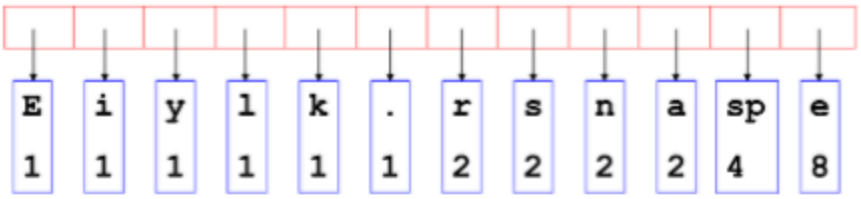
    iii. Dequeue next node and make it right subtree

    iv. Frequency of new node = sum of frequency of left and right children

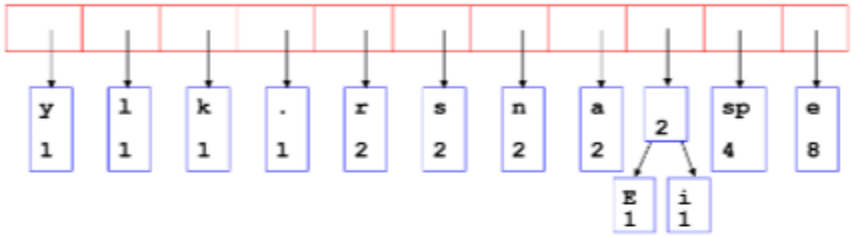
    v. Enqueue new node back into queue

Simulation:

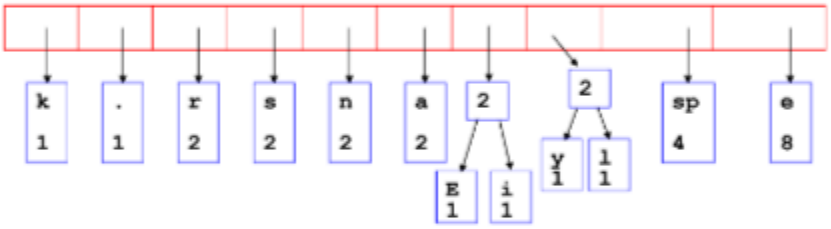
Step 1:



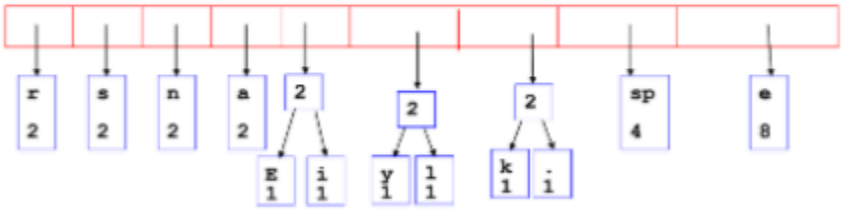
Step 2:



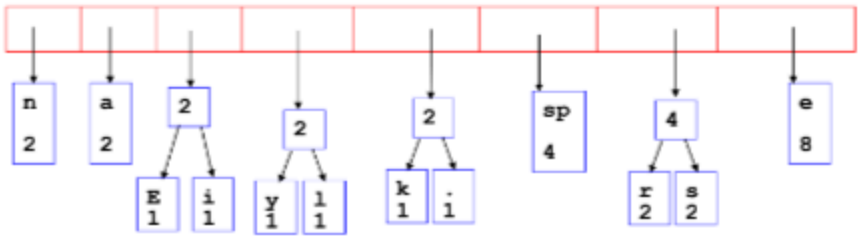
Step 3:



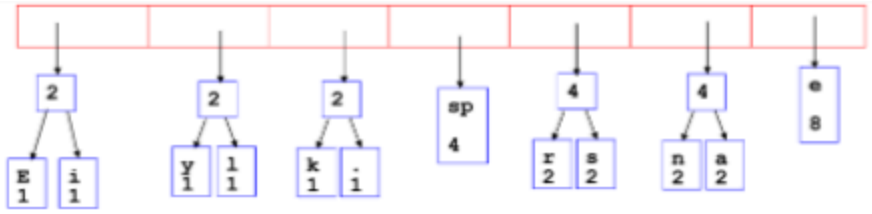
Step 4:



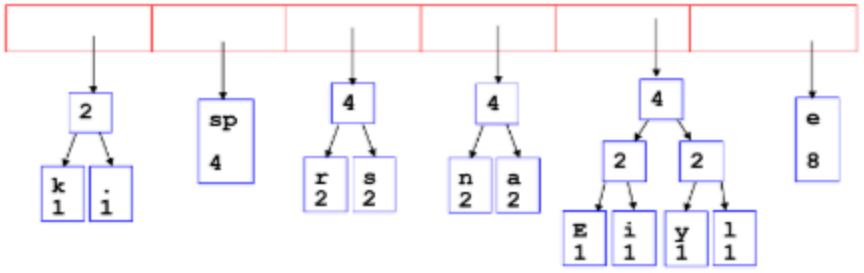
Step 5:



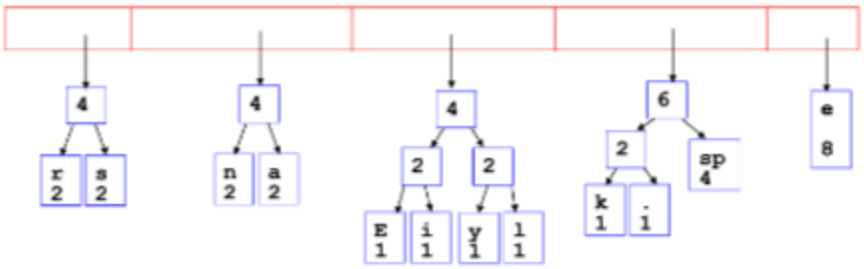
Step 6:



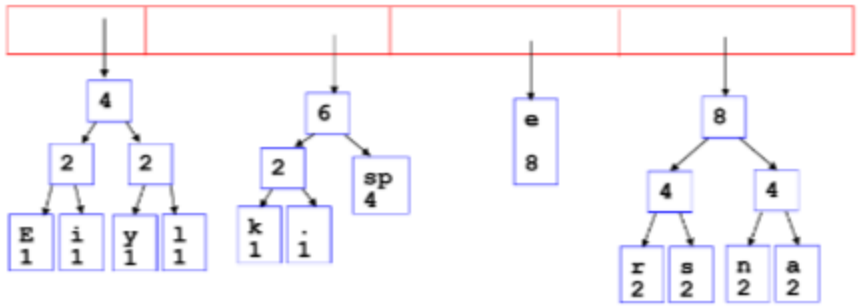
Step 7:



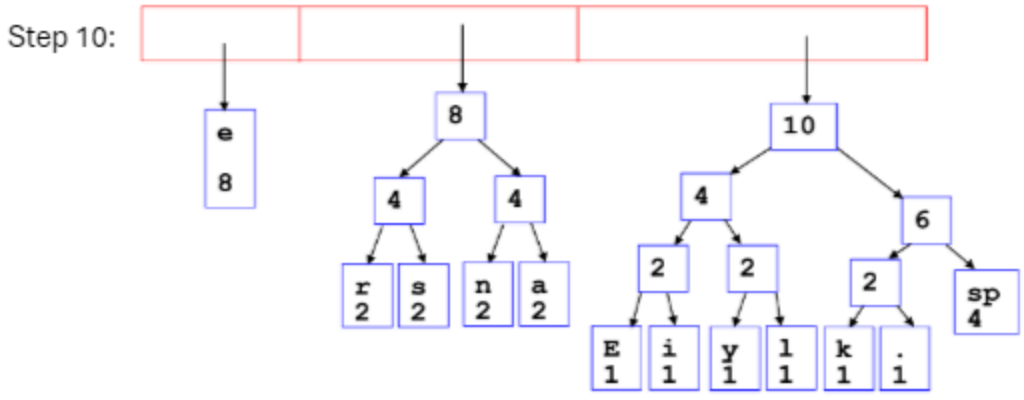
Step 8:



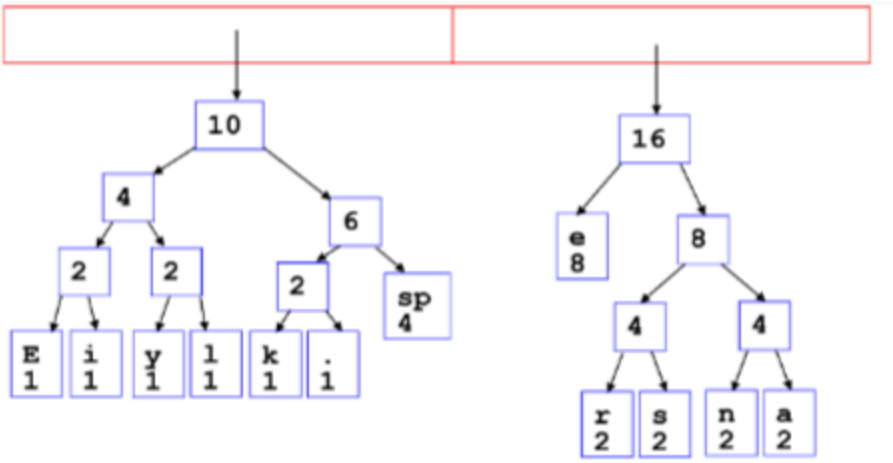
Step 9:



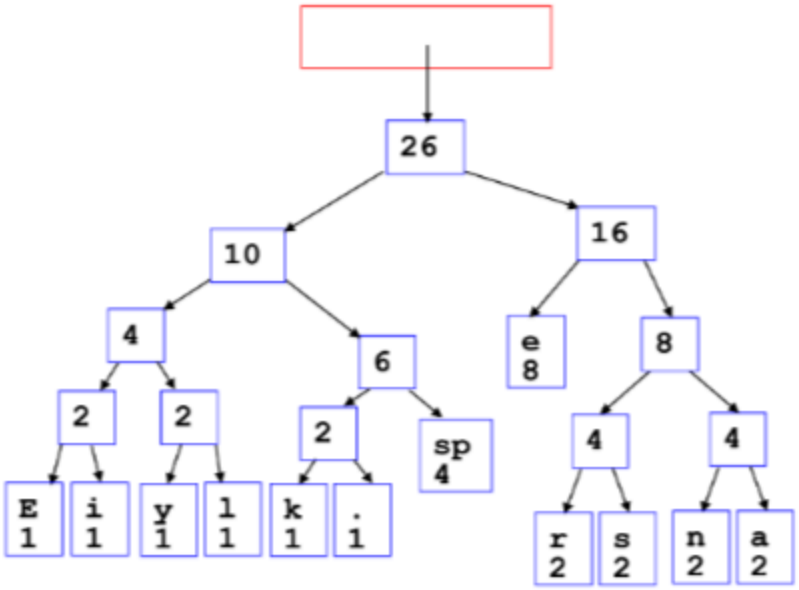
Step 10:



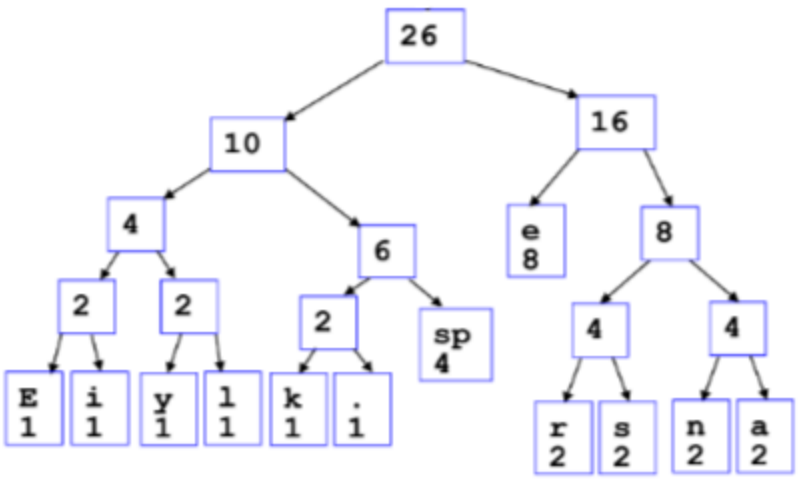
Step 11:



Step 12:

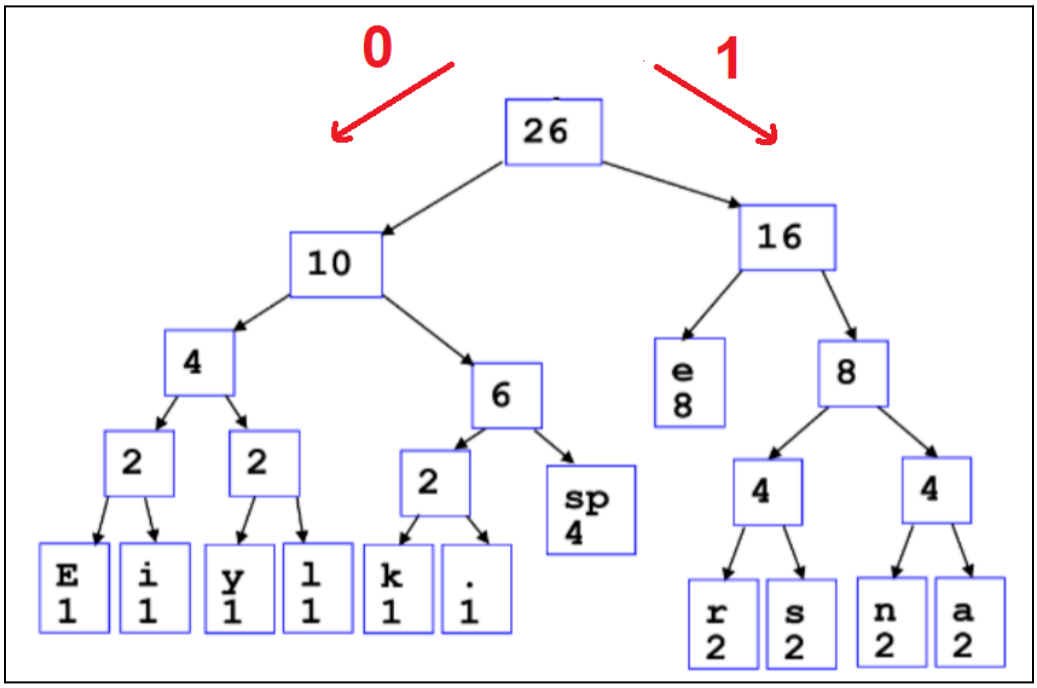


Final Tree:



Encoding:

- Going left is a 0
- Going right is a 1
- Code word is only completed when a leaf node is reached



Character	Code	Character	Code
E	0000	space	011
i	0001	e	10
y	0010	r	1100
l	0011	s	1101
k	0100	n	1110
.	0101	a	1111

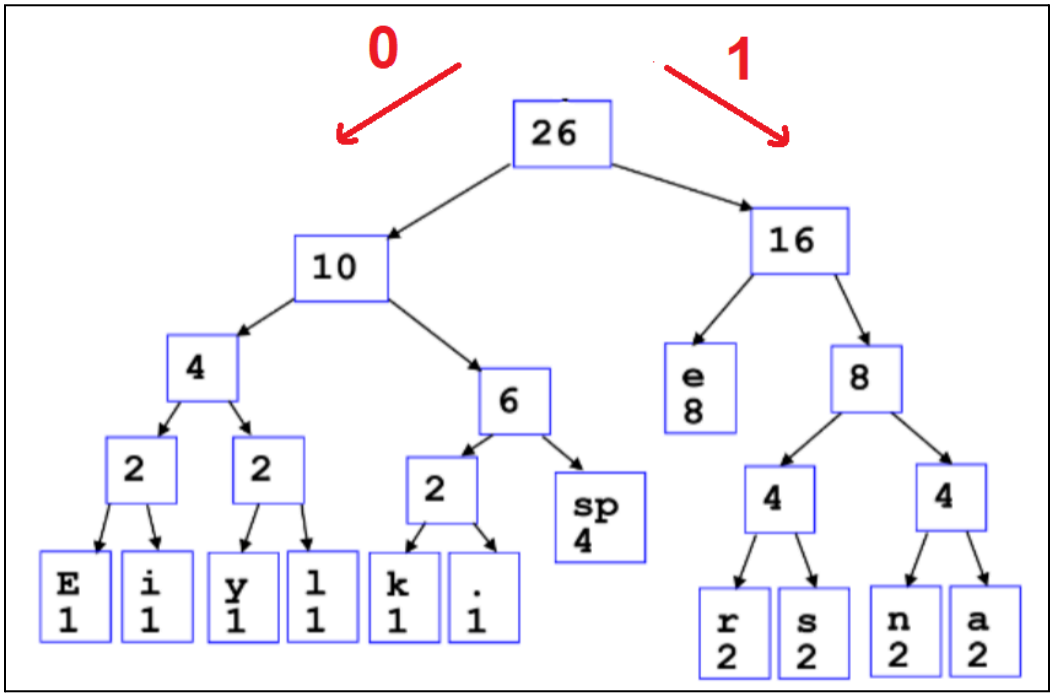
Text: Eerie eyes seen near lake.

Encoded:
00001011000101001110010101101101011011101110111010111110001100111110100101010101

84 bits are required to represent instead of 208 bits (ASCII). Hence, there are savings.

Decoding:

- Once the receiver has a tree it scans the incoming bit stream
- 0 = Go Left
- 1 = Go Right
- A character is recorded when a leaf node is reached



Encoded:
00001011000101001110010101101101011011101110111011101011111000110011111010010101
01

Decoded Text: Eerie eyes seen near lake.

Time Complexity:

Frequency count (m = input size)	O(m) [Linear Search]
Min-heap creation (n = unique characters)	O(n log n) [Swim n nodes]
Build a Huffman tree	O(n log n) [Delete 2 smallest n-1 times]
Generate codes (DFS)	O(n) [Assigning codes for each leaf nodes]
Total	O(m + n log n)