Design and Analysis of Algorithm

UNIT – 4: Greedy Algorithm

Greedy Algorithm:

The greedy method is one of the strategies like Divide and conquer used to solve the problems. This method is used for solving optimization problems. An optimization problem is a problem that demands either maximum or minimum results. Let's understand through some terms.

The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

Characteristic components of greedy algorithm:

1. **The feasible solution**: A subset of given inputs that satisfies all specified constraints of a problem is known as a "feasible solution".

2. **Optimal solution:** The feasible solution that achieves the desired value is called an "optimal solution". In other words, the feasible solution that either minimizes or maximizes the objective function specified in a problem is known as an "optimal solution".

3. **Feasibility check:** It investigates whether the selected input fulfils all constraints mentioned in a problem or not. If it fulfils all the constraints then it is added to a set of feasible solutions; otherwise, it is rejected.

4. **Optimality check:** It investigates whether a selected input produces either a minimum or maximum value of the objective function by fulfilling all the specified constraints. If an element in a solution set produces the desired value, then it is added to a sell of optimal solutions.

5. **Optimal substructure property:** The globally optimal solution to a problem includes the optimal sub solutions within it.

Activity Selection Problem:

The Activity Selection Problem is an optimization problem which deals with the selection of non-conflicting activities that needs to be executed by a single person or machine in a given time frame.

Each activity is marked by a start and finish time. Greedy technique is used for finding the solution since this is an optimization problem.

Let's consider that you have n activities with their start and finish times, the objective is to find solution set having **maximum number of non-conflicting activities** that can be executed in a single time frame, assuming that only one person or machine is available for execution.

Some **points to note** here:

- It might not be possible to complete all the activities, since their timings can collapse.

- Two activities, say **i** and **j**, are said to be non-conflicting if si >= fj or sj >= fi where si and sj denote the starting time of activities **i** and **j** respectively, and fi and fj refer to the finishing time of the activities **i** and **j** respectively.

- **Greedy approach** can be used to find the solution since we want to maximize the count of activities that can be executed. This approach will greedily choose an activity with earliest finish time at every step, thus yielding an optimal solution.

Input Data for the Algorithm:
- act[] array containing all the activities.
- s[] array containing the starting time of all the activities.
- f[] array containing the finishing time of all the activities.

Output Data from the Algorithm:
- sol[] array referring to the solution set containing the maximum number of non-conflicting activities.

Steps for Activity Selection Problem:

Following are the steps we will be following to solve the activity selection problem,

Step 1: Sort the given activities in ascending order according to their finishing time.

Step 2: Select the first activity from sorted array act[] and add it to sol[] array.

Step 3: Repeat steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of previously selected activity, then add it to the sol[] array.

Step 5: Select the next activity in act[] array.

Step 6: Print the sol[] array.

Activity Selection Problem Example:

Let's try to trace the steps of above algorithm using an example:

In the table below, we have 6 activities with corresponding start and end time, the objective is to compute an execution schedule having maximum number of non-conflicting activities:

Start Time (s)	Finish Time (f)	Activity Name
5	9	a1
1	2	a2
3	4	a3
0	6	a4
5	7	a5
8	9	a6

A possible **solution** would be:

Step 1: Sort the given activities in ascending order according to their finishing time. The table after we have sorted it:

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6

Step 2: Select the first activity from sorted array act[] and add it to the sol[] array, thus **sol = {a2}**.

Step 3: Repeat the steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of the previously selected activity, then add it to sol[].

Step 5: Select the next activity in act[]

For the data given in the above table,

A. Select activity **a3**. Since the start time of **a3** is greater than the finish time of **a2** (i.e. s(a3) > f(a2)), we add **a3** to the solution set. Thus **sol = {a2, a3}**.

B. Select **a4**. Since s(a4) < f(a3), it is not added to the solution set.

C. Select **a5**. Since s(a5) > f(a3), **a5** gets added to solution set. Thus **sol = {a2, a3, a5}**

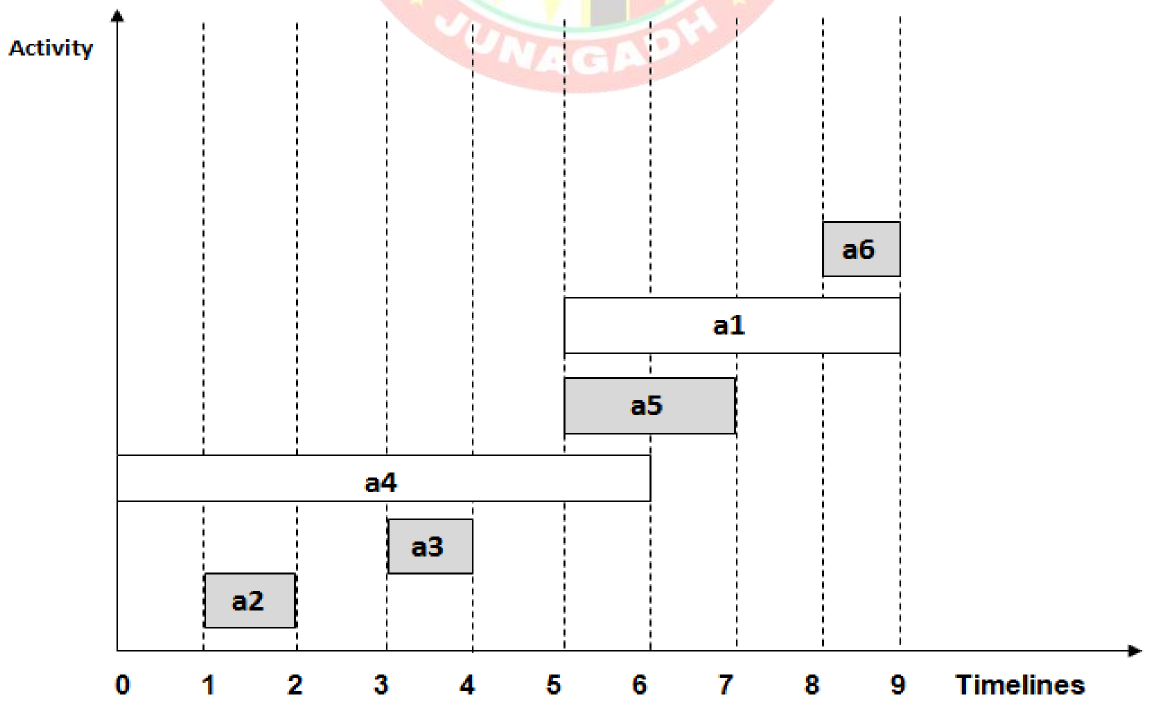
D. Select **a1**. Since s(a1) < f(a5), **a1** is not added to the solution set.

E. Select **a6**. **a6** is added to the solution set since s(a6) > f(a5). Thus **sol = {a2, a3, a5, a6}**.

Step 6: At last, print the array sol[]

Hence, the execution schedule of maximum number of non-conflicting activities will be:

(1,2)
(3,4)
(5,7)
(8,9)



In the above diagram, the selected activities have been highlighted in grey.

Elements of Greedy Strategy:

A greedy strategy is an algorithmic paradigm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

In other words, a greedy algorithm makes the best choice at each step without worrying about the future consequences. While greedy strategies may not always guarantee an optimal solution, they are often efficient and easy to implement. Here are some key elements of a greedy strategy:

Greedy Choice Property:

A global optimum can be reached by selecting a local optimum at each step.

Optimal Substructure:

The problem can be divided into subproblems, and the optimal solution to the overall problem can be constructed from the optimal solutions to its subproblems.

Greedy Algorithm:

The algorithm iteratively makes a series of choices, each optimized for the current step, without reconsidering those choices in the future steps.

Selection of Locally Optimal Choices:

At each step, the algorithm makes the best choice available without considering the overall problem.

Does Not Always Guarantee an Optimal Solution:

Greedy algorithms do not always guarantee the optimal solution for every problem. However, they often provide a solution that is close to the optimum.

No Backtracking:

Once a decision is made, it is not reconsidered. Greedy algorithms do not backtrack or undo previous decisions.

Efficiency:

Greedy algorithms are often efficient and have a low time complexity. They are suitable for problems where a locally optimal choice leads to a globally optimal solution.

Sorting or Priority Queue:

Many greedy algorithms involve sorting elements or maintaining a priority queue to efficiently make locally optimal choices.

Proof of Correctness:

To show that a greedy algorithm produces an optimal solution, a proof of correctness is often required, demonstrating that the greedy choice at each step leads to an optimal solution globally.

Examples of problems that can be solved using a greedy strategy include:

Activity Selection Problem

Fractional Knapsack Problem

Huffman Coding

Shortest Path Algorithm

Prim's Minimum Spanning Tree Algorithm

It's important to note that while greedy strategies are powerful and efficient in many cases, they are not suitable for all types of problems. Some problems may require more complex algorithms, such as dynamic programming or backtracking, to find the optimal solution.

Kruskal's (Prim's) Algorithm (Minimum Spanning Trees):

The Kruskal's Algorithm is used to find the minimum cost of a spanning tree. A spanning tree is a connected graph using all the vertices in which there are no loops. In other words, we can say that there is a path from any vertex to any other vertex but no loops.

What is Minimum Cost Spanning Tree?

The minimum spanning tree is a spanning tree that has the smallest total edge weight. The Kruskal algorithm is an algorithm that takes the graph as input and finds the edges from the graph, which forms a tree that includes every vertex of a graph.

Working of Kruskal Algorithm:

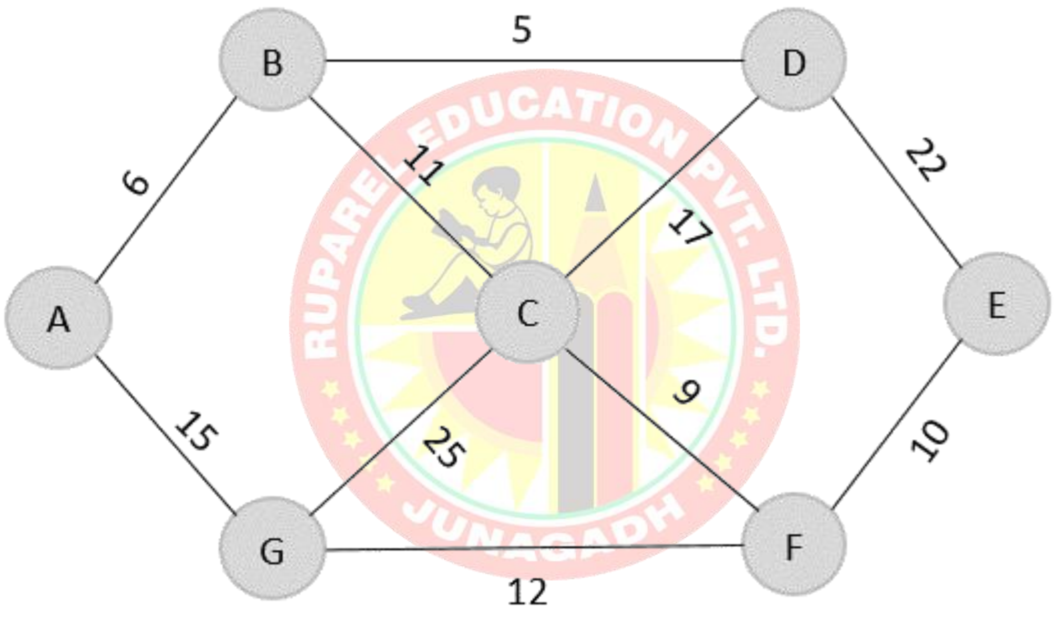
The working of the Kruskal algorithm starts from the edges, which has the lowest weight and keeps adding the edges until we reach the goal.

The following are the steps used to implement the Kruskal algorithm:

- First, sort the edges in the ascending order of their edge weights.
- Consider the edge which is having the lowest weight and add it in the spanning tree. If adding any edge in a spanning tree creates a cycle, then reject that edge.
- Keep adding the edges until we reach the end vertex.

Let's understand through an example:

Construct a minimum spanning tree using Kruskal's algorithm for the graph given below –

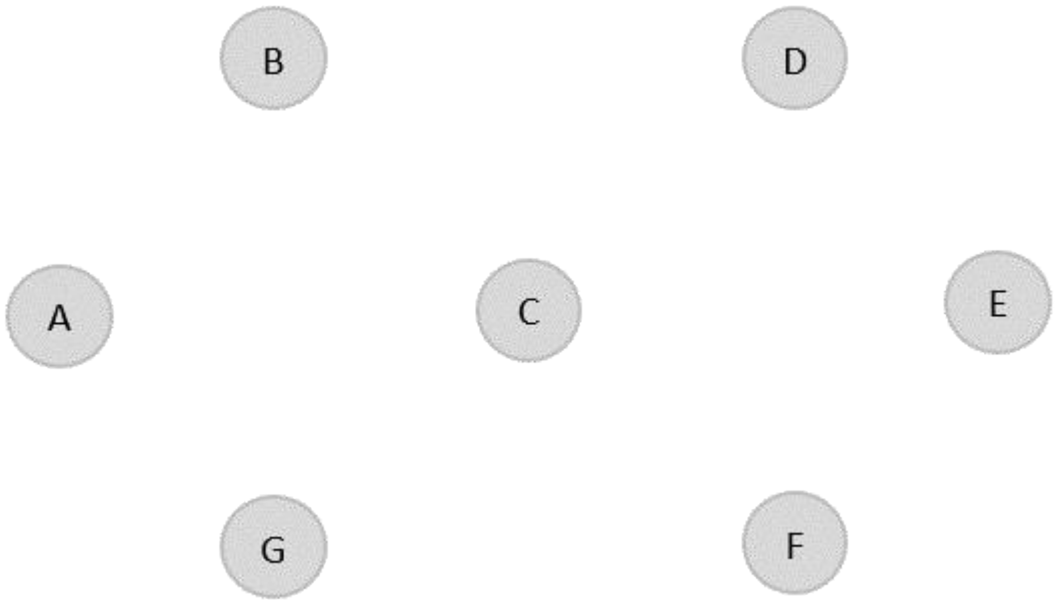


Solution:

As the first step, sort all the edges in the given graph in an ascending order and store the values in an array.

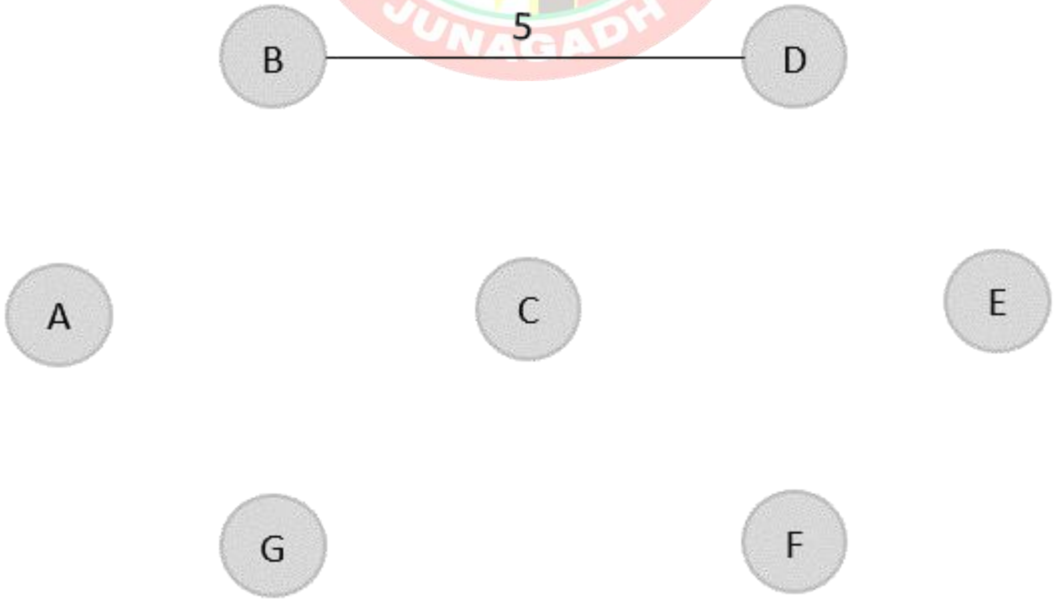
Edge	B→D	A→B	C→F	F→E	B→C	G→F	A→G	C→D	D→E	C→G
Cost	5	6	9	10	11	12	15	17	22	25

Then, construct a forest of the given graph on a single plane.

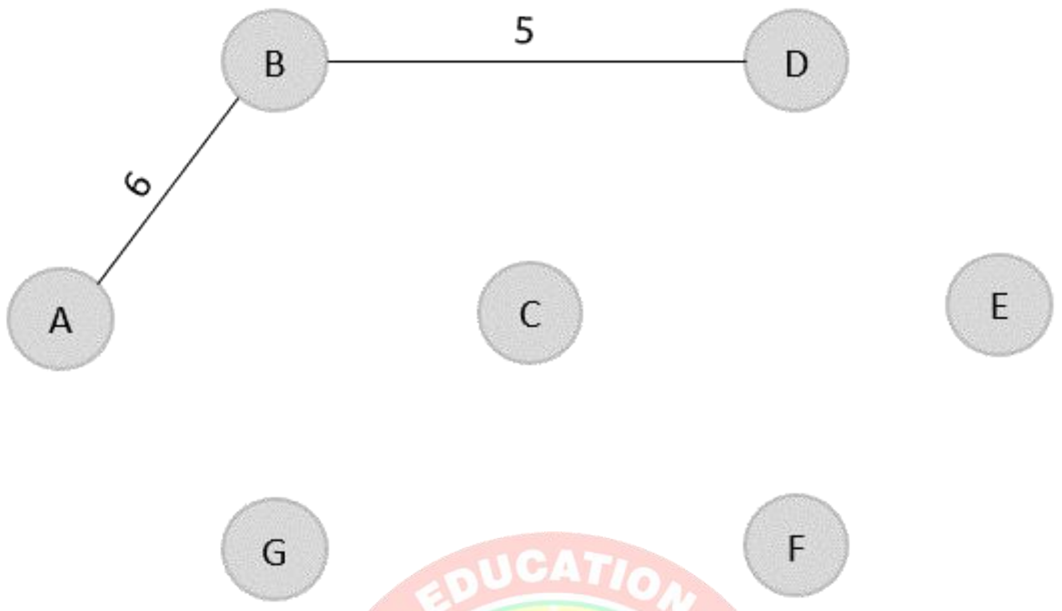


From the list of sorted edge costs, select the least cost edge and add it onto the forest in output graph.

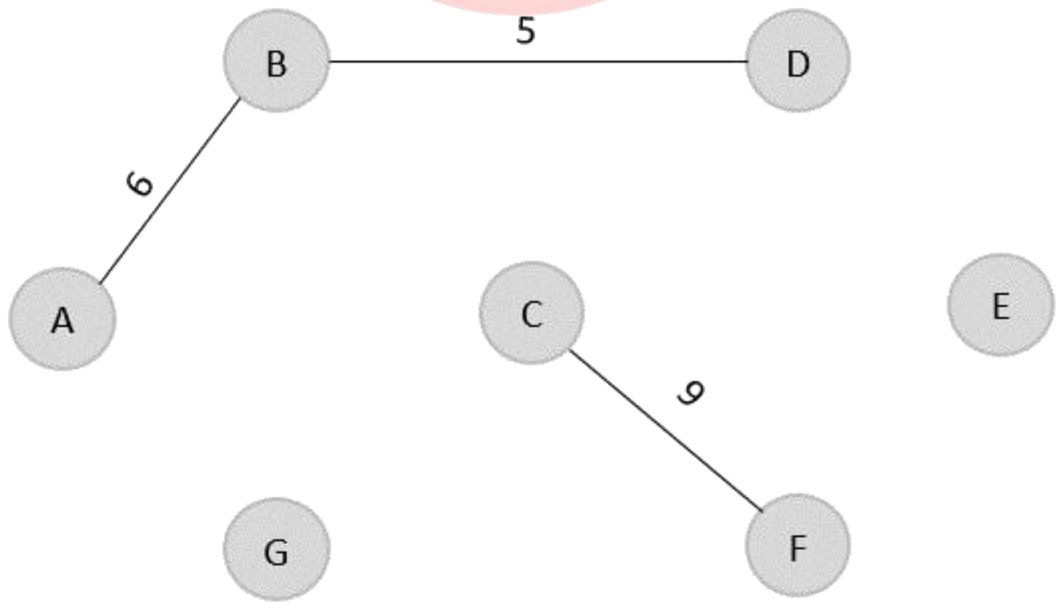
B → D = 5
Minimum cost = 5
Visited array, v = {B, D}



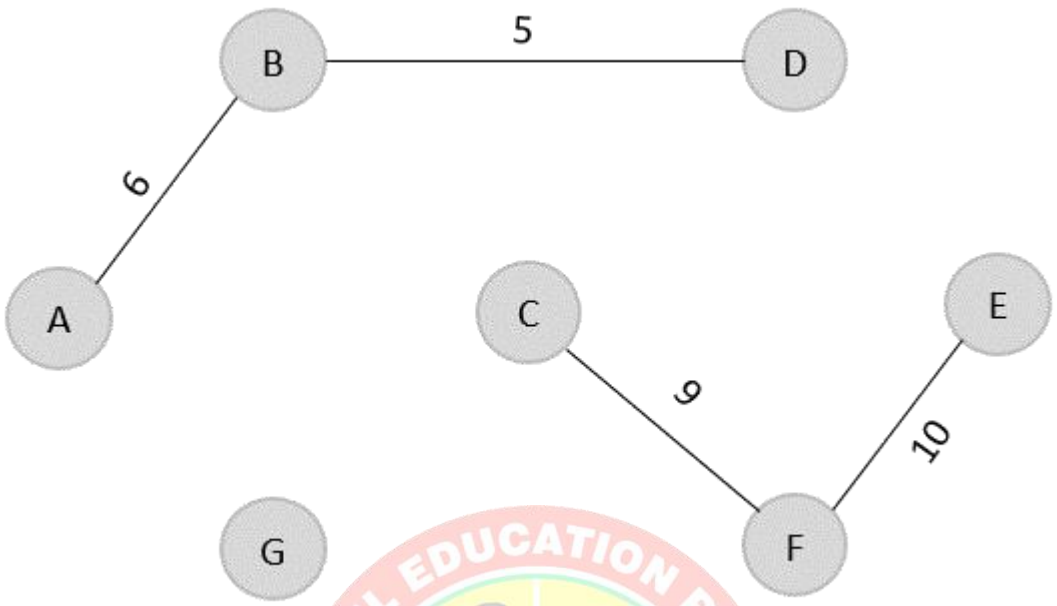
Similarly, the next least cost edge is B → A = 6; so we add it onto the output graph.
Minimum cost = 5 + 6 = 11
Visited array, v = {B, D, A}



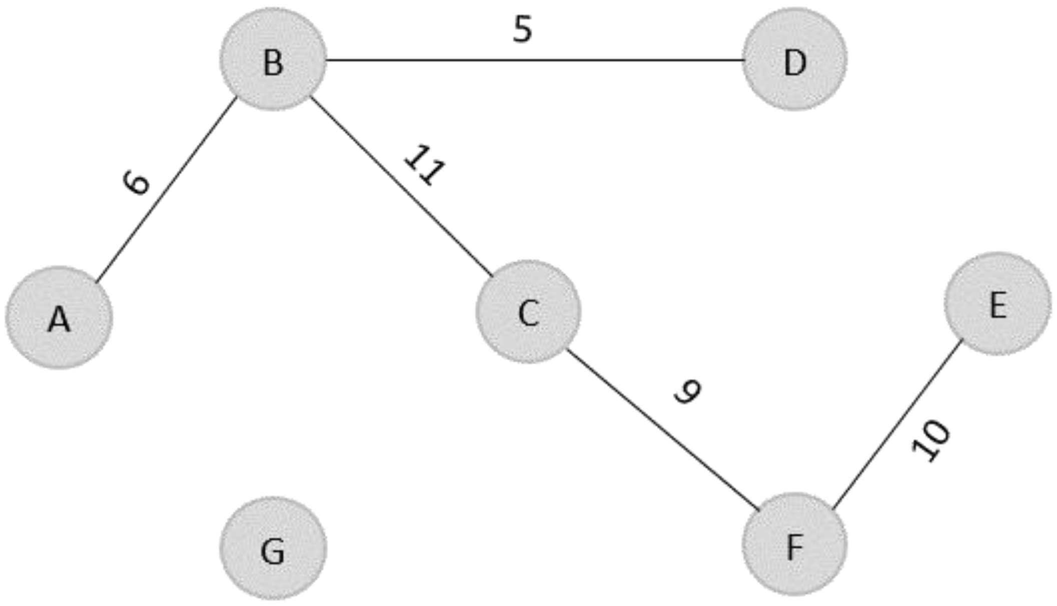
The next least cost edge is C → F = 9; add it onto the output graph.
Minimum Cost = 5 + 6 + 9 = 20
Visited array, v = {B, D, A, C, F}



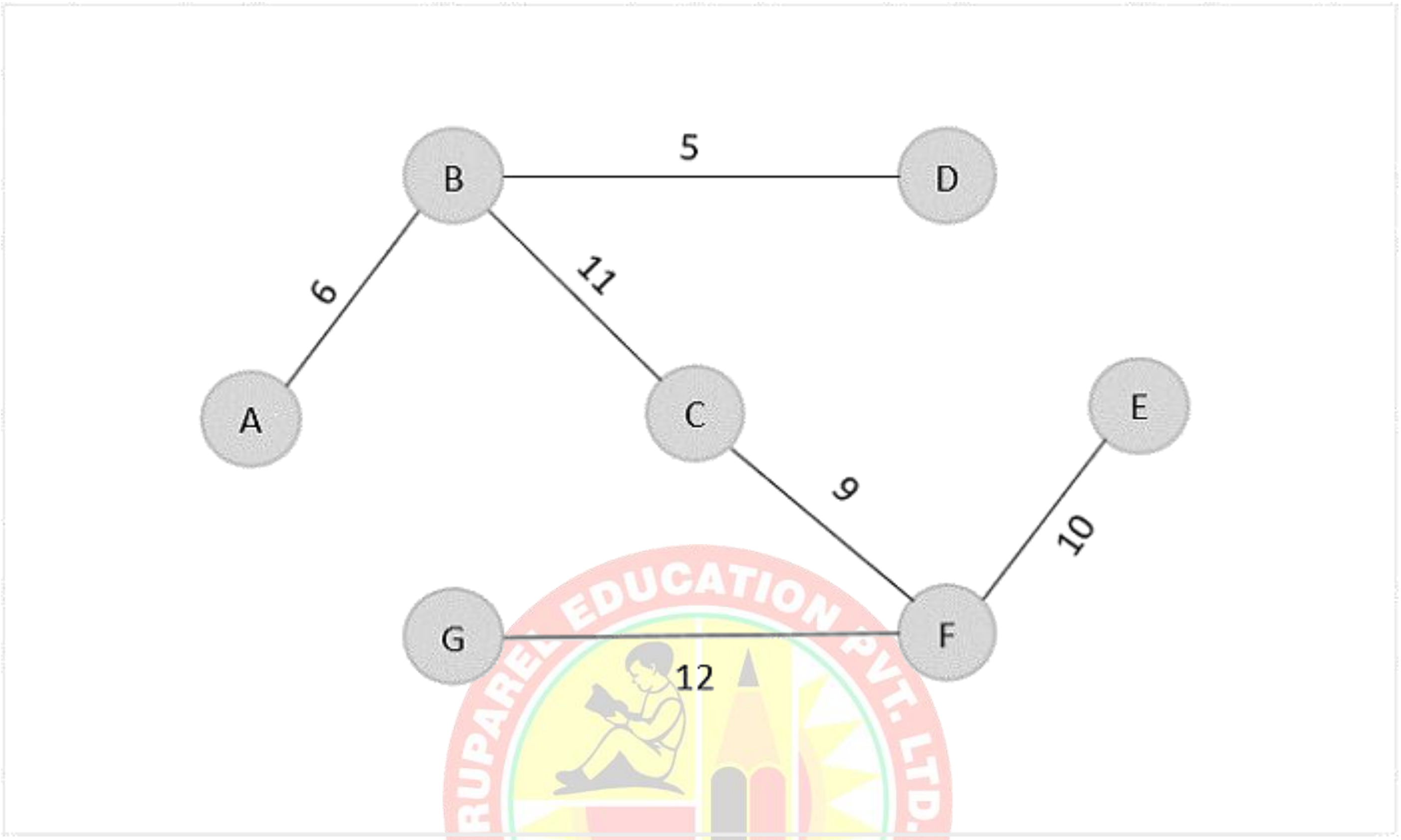
The next edge to be added onto the output graph is F → E = 10.
Minimum Cost = 5 + 6 + 9 + 10 = 30
Visited array, v = {B, D, A, C, F, E}



The next edge from the least cost array is B → C = 11, hence we add it in the output graph.
Minimum cost = 5 + 6 + 9 + 10 + 11 = 41
Visited array, v = {B, D, A, C, F, E}



The last edge from the least cost array to be added in the output graph is F → G = 12.
Minimum cost = 5 + 6 + 9 + 10 + 11 + 12 = 53
Visited array, v = {B, D, A, C, F, E, G}



The obtained result is the minimum spanning tree of the given graph with cost = **53.**

Dijkstra's Algorithm:

The following tutorial will teach us about Dijkstra's Shortest Path Algorithm. We will understand the working of Dijkstra's Algorithm with a stepwise graphical explanation.

A Brief Introduction to Graphs:

Graphs are non-linear data structures representing the "connections" between the elements.

These elements are known as the **Vertices**, and the lines or arcs that connect any two vertices in the graph are known as the **Edges**.

More formally, a Graph comprises **a set of Vertices (V)** and **a set of Edges (E)**.

The Graph is denoted by **G(V, E)**.

Components of a Graph:

1. **Vertices:** Vertices are the basic units of the graph used to represent real-life objects, persons, or entities. Sometimes, vertices are also known as Nodes.

2. **Edges:** Edges are drawn or used to connect two vertices of the graph. Sometimes, edges are also known as Arcs.

The following figure shows a graphical representation of a Graph:

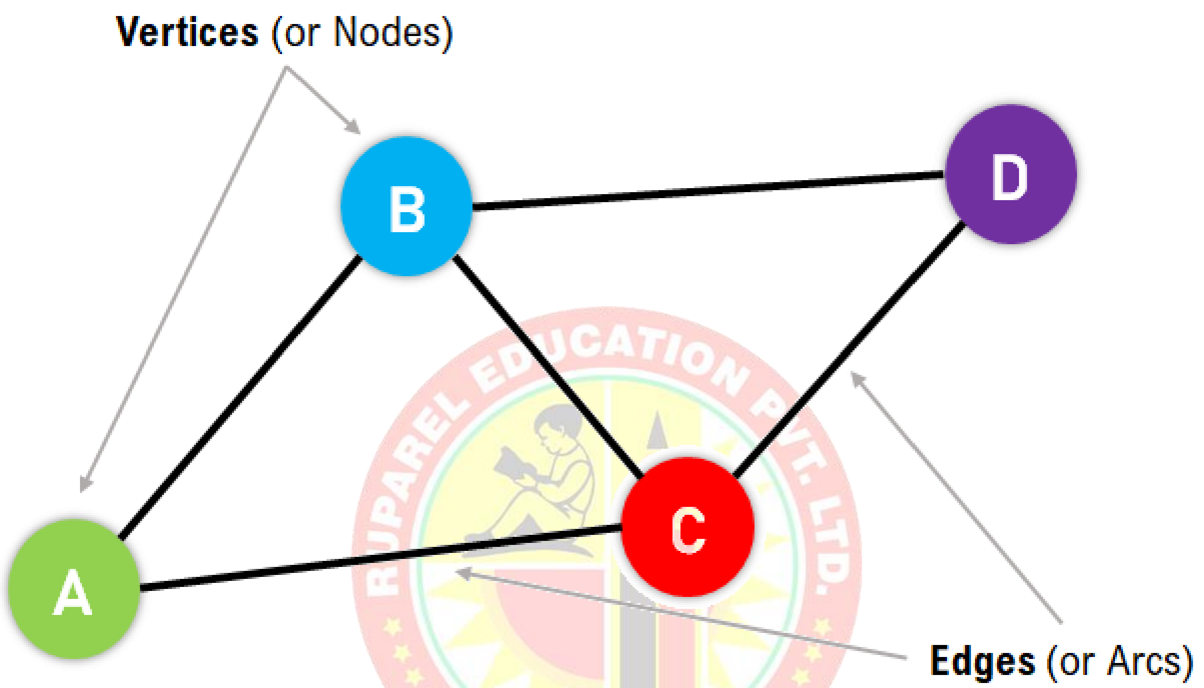


Figure 1: Graphical Representation of a Graph

In the above figure, the Vertices/Nodes are denoted with Coloured Circles, and the Edges are denoted with the lines connecting the nodes.

Applications of the Graphs:

Graphs are used to solve many real-life problems. Graphs are utilized to represent the networks. These networks may include telephone or circuit networks or paths in a city.

For example,
we could use Graphs to design a transportation network model where the vertices display the facilities that send or receive the products, and the edges represent roads or paths connecting them. The following is a pictorial representation of the same:

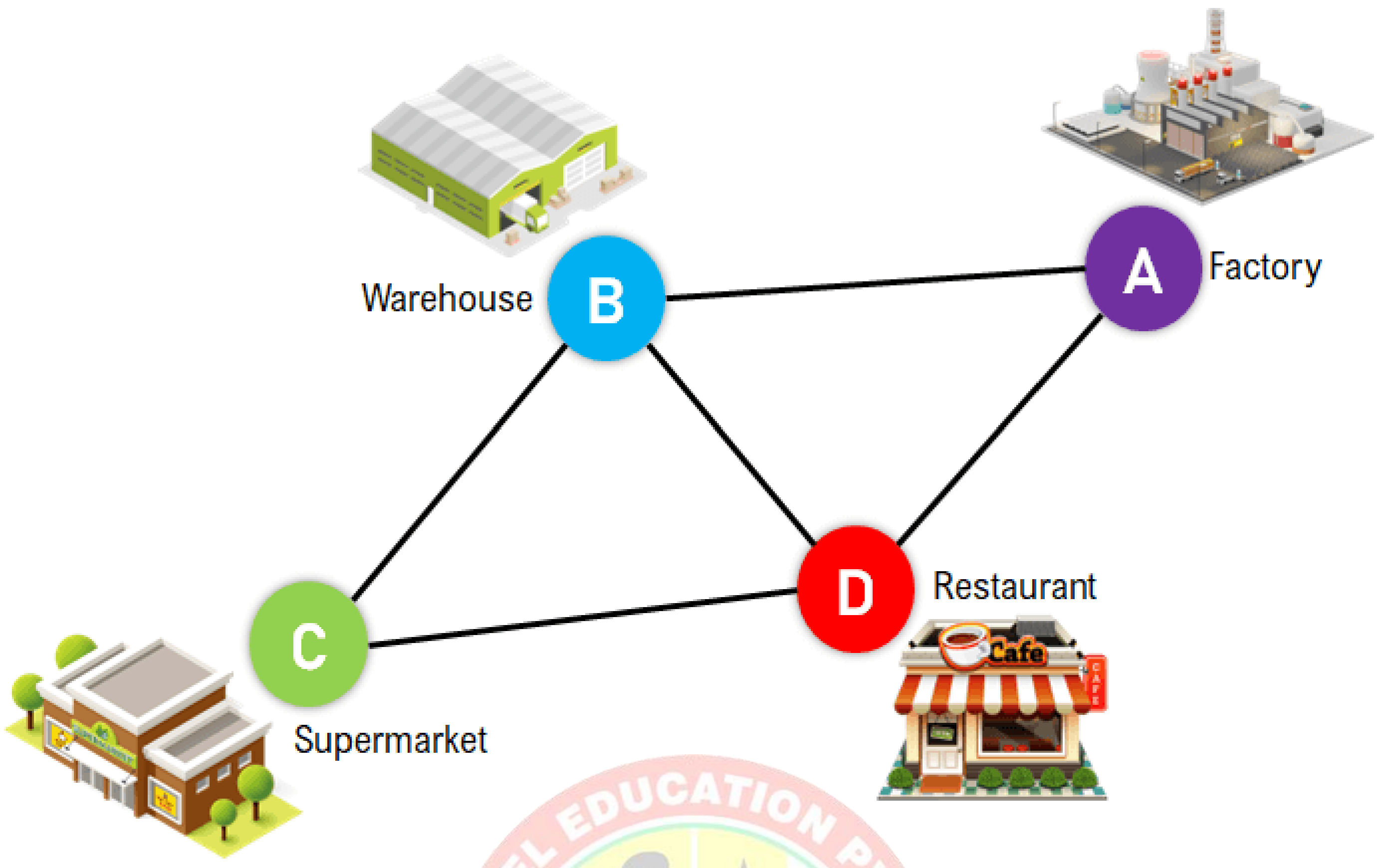


Figure 2: Pictorial Representation of Transportation Network

Graphs are also utilized in different Social Media Platforms like LinkedIn, Facebook, Twitter, and more.

For example, Platforms like Facebook use Graphs to store the data of their users where every person is indicated with a vertex, and each of them is a structure containing information like Person ID, Name, Gender, Address, etc.

Types of Graphs:

The Graphs can be categorized into two types:

1. **Undirected Graph**
2. **Directed Graph**

Undirected Graph:

A Graph with edges that do not have a direction is termed an Undirected Graph. The edges of this graph imply a two-way relationship in which each edge can be traversed in both directions. The following figure displays a simple undirected graph.

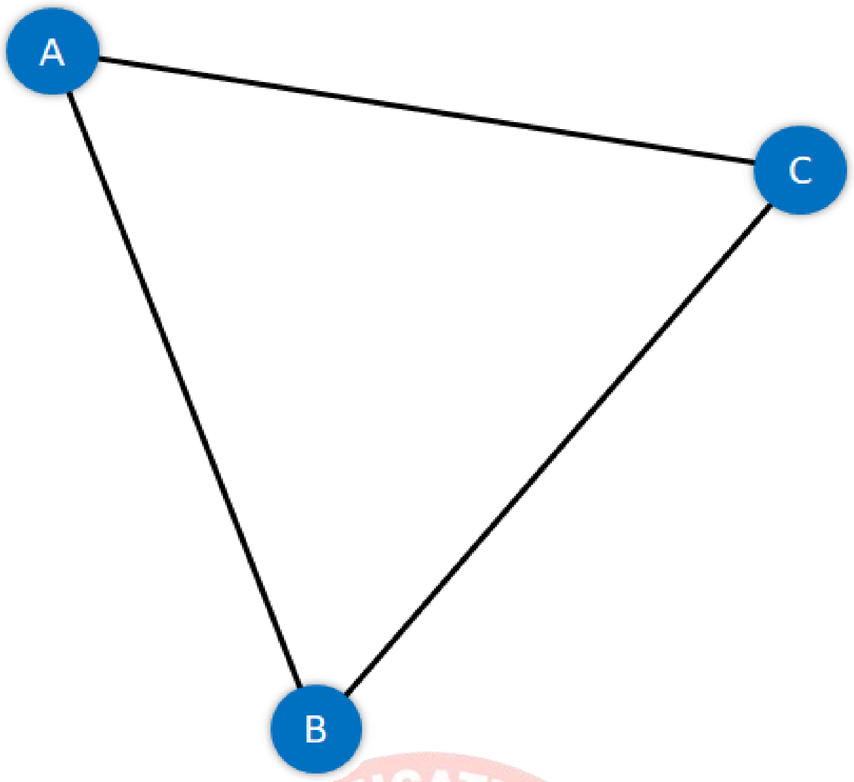


Figure 3: A Simple Undirected Graph

Directed Graph:

A Graph with edges with direction is termed a Directed Graph. The edges of this graph imply a one-way relationship in which each edge can only be traversed in a single direction. The following figure displays a simple directed graph.

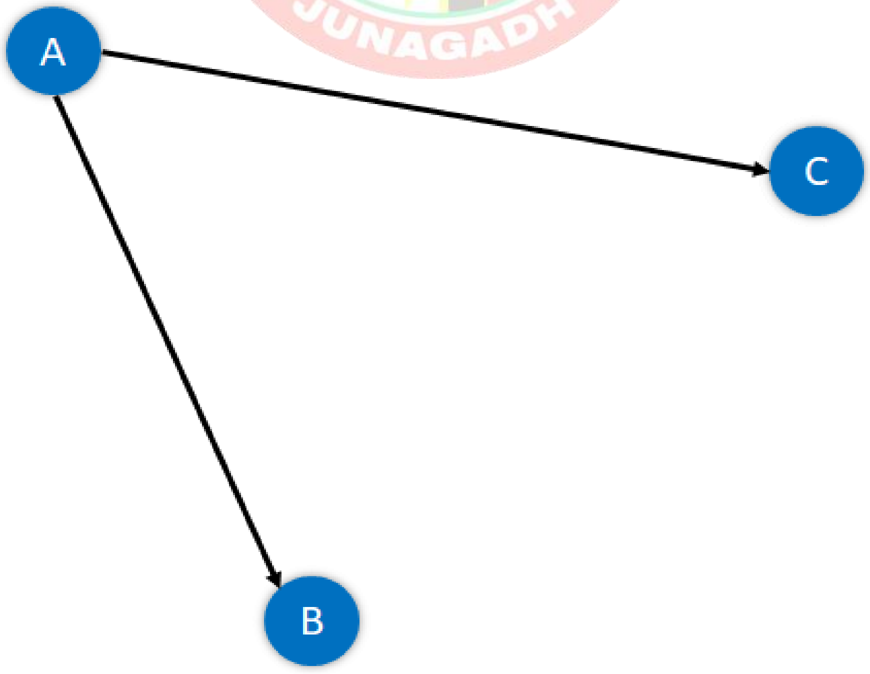


Figure 4: A Simple Directed Graph.

The absolute length, position, or orientation of the edges in a graph illustration characteristically does not have meaning. In other words, we can visualize the same graph in different ways by rearranging the vertices or distorting the edges if the underlying structure of the graph does not alter.

What are Weighted Graphs?

A Graph is said to be Weighted if each edge is assigned a 'weight'. The weight of an edge can denote distance, time, or anything that models the 'connection' between the pair of vertices it connects.

For instance, we can observe a blue number next to each edge in the following figure of the Weighted Graph. This number is utilized to signify the weight of the corresponding edge.

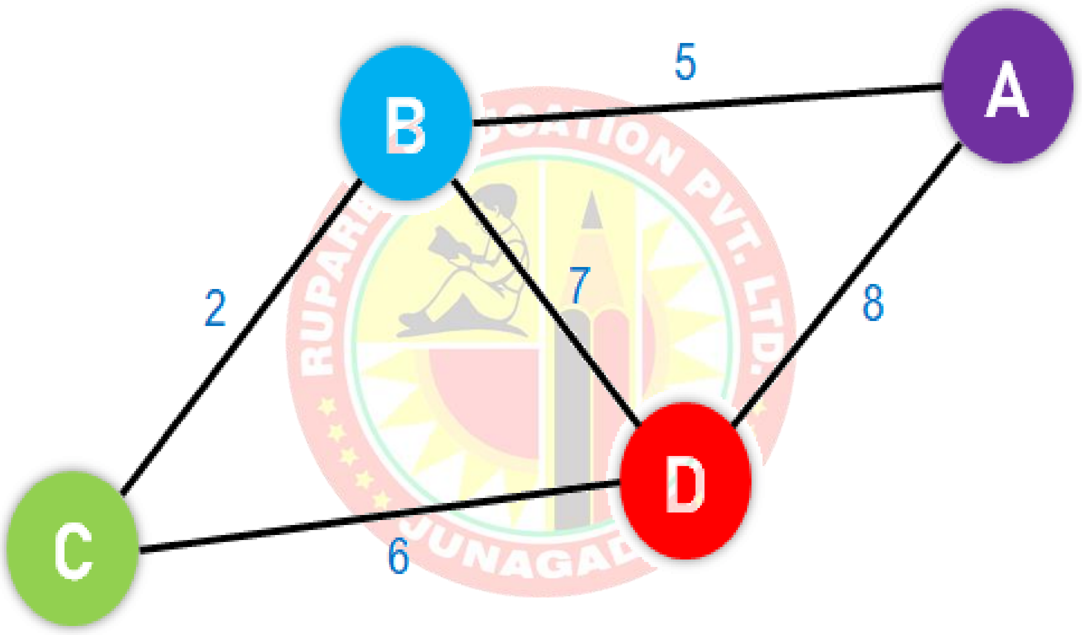


Figure 5: An Example of a Weighted Graph

Job Scheduling using Greedy Algorithm:

Job scheduling is the problem of scheduling jobs out of a set of N jobs on a single processor which maximizes profit as much as possible. Consider N jobs, each taking unit time for execution. Each job is having some profit and deadline associated with it. Profit earned only if the job is completed on or before its deadline. Otherwise, we have to pay a profit as a penalty. Each job has deadline $d_i \geq 1$ and profit $p_i \geq 0$. At a time, only one job can be active on the processor.

The job is feasible only if it can be finished on or before its deadline. A feasible solution is a subset of N jobs such that each job can be completed on or before its deadline. An optimal solution is a solution with maximum profit. The simple and inefficient solution is to generate all subsets of the given set of jobs and find the feasible set that maximizes the profit.

Our goal is to find a feasible schedule S which maximizes the profit of scheduled job. The goal can be achieved as follow: Sort all jobs in decreasing order of profit. Start with the empty schedule, select one job at a time and if it is feasible then schedule it in the *latest possible slot*.

Complexity Analysis of Job Scheduling:

Simple greedy algorithm spends most of the time looking for the latest slot a job can use. On average, N jobs search N/2 slots. This would take $O(N^2)$ time.

Examples:

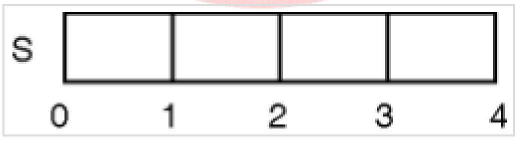
Problem: Solve the following job scheduling with deadlines problem using the greedy method. Number of jobs N = 4. Profits associated with Jobs: $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$. Deadlines associated with jobs $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Solution:

Sort all jobs in descending order of profit.

So, P = (100, 27, 15, 10), J = (J_1, J_4, J_3, J_2) and D = (2, 1, 2, 1).

We shall select one by one job from the list of sorted jobs, and check if it satisfies the deadline. If so, schedule the job in the latest free slot. If no such slot is found, skip the current job and process the next one. Initially,



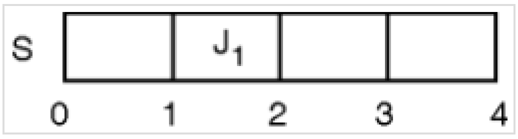
Profit of scheduled jobs, SP = 0

Iteration 1:

Deadline for job J_1 is 2.

Slot 2 (t = 1 to t = 2) is free, so schedule it in slot 2.

Solution set S= $\{J_1\}$, and Profit SP = {100}

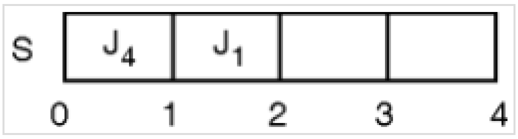


Iteration 2:

Deadline for job J_4 is 1.

Slot 1 (t = 0 to t = 1) is free, so schedule it in slot 1.

Solution set S = {J_1,J_4}, and Profit SP = {100, 27}



Iteration 3:

Job J_3 is not feasible because first two slots are already occupied and if we schedule J_3 any time later t = 2, it cannot be finished before its deadline 2.

So job J_3 is discarded,

Solution set S = {J_1,J_4}, and Profit SP = {100, 27}

Iteration 4:

Job J_2 is not feasible because first two slots are already occupied and if we schedule J_2 any time later t = 2, it cannot be finished before its deadline 1.

So job J_2 is discarded,

Solution set S = {J_1,J_4}, and Profit SP = {100, 27}

With the greedy approach, we will be able to schedule two jobs {J_1, J_4}, which gives a profit of 100 + 27 = 127 units.

problem: n = 7, profits (p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (3, 5, 20, 18, 1, 6, 30) and deadlines (d_1, d_2, d_3, d_4, d_5, d_6, d_7) = (1, 3, 4, 3, 2, 1, 2). Schedule the jobs in such a way to get maximum profit.

Solution:

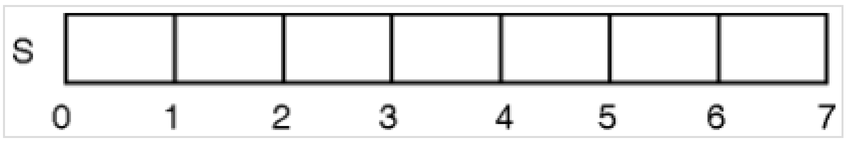
Given that,

Jobs	j_1	j_2	j_3	j_4	j_5	j_6	j_7
Profit	3	5	20	18	1	6	30
Deadline	1	3	4	3	2	1	2

Sort all jobs in descending order of profit.

So, P = (30, 20, 18, 6, 5, 3, 1), J = (J_7, J_3, J_4, J_6, J_2, J_1, J_5) and D = (2, 4, 3, 1, 3, 1, 2).

We shall select one by one job from the list of sorted jobs J, and check if it satisfies the deadline. If so, schedule the job in the latest free slot. If no such slot is found, skip the current job and process the next one. Initially,



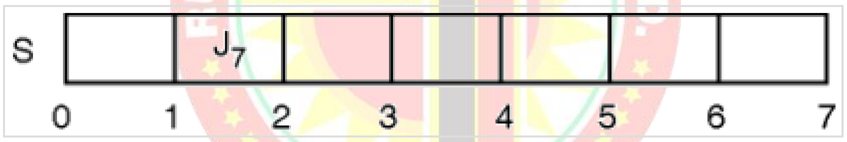
Profit of scheduled jobs, SP = 0

Iteration 1:

Deadline for job J_7 is 2.

Slot 2 (t = 1 to t = 2) is free, so schedule it in slot 2.

Solution set S = {J_7}, and Profit SP = {30}

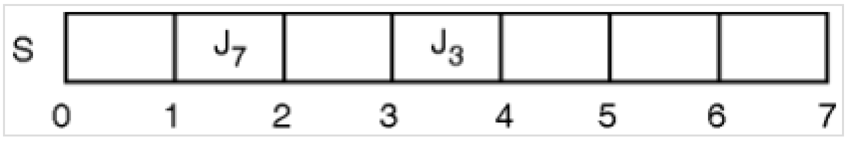


Iteration 2:

Deadline for job J_3 is 4.

Slot 4 (t = 3 to t = 4) is free, so schedule it in slot 4.

Solution set S = {J_7, J_3}, and Profit SP = {30, 20}

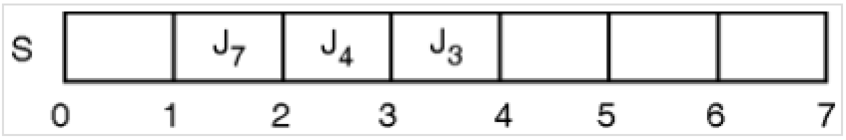


Iteration 3:

Deadline for job J_4 is 3.

Slot 3 (t = 2 to t = 3) is free, so schedule it in slot 3.

Solution set S = {J_7, J_3, J_4}, and Profit SP = {30, 20, 18}

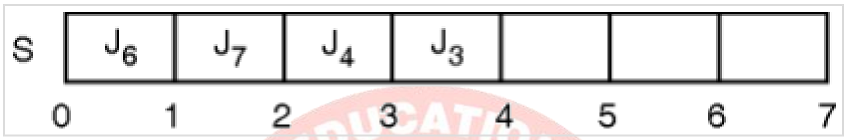


Iteration 4:

Deadline for job J_6 is 1.

Slot 1 (t = 0 to t = 1) is free, so schedule it in slot 1.

Solution set S = {J_7, J_3, J_4, J_6}, and Profit SP = {30, 20, 18, 6}



First, all four slots are occupied and none of the remaining jobs has deadline lesser than 4.

So none of the remaining jobs can be scheduled.

Thus, with the greedy approach, we will be able to schedule four jobs {J7, J3, J4, J6}, which give a profit of (30 + 20 + 18 + 6) = 74 units.

Huffman Coding:

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Let us understand prefix codes with a counter example.

Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

See this for applications of Huffman Coding.

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree:

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.
   Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.

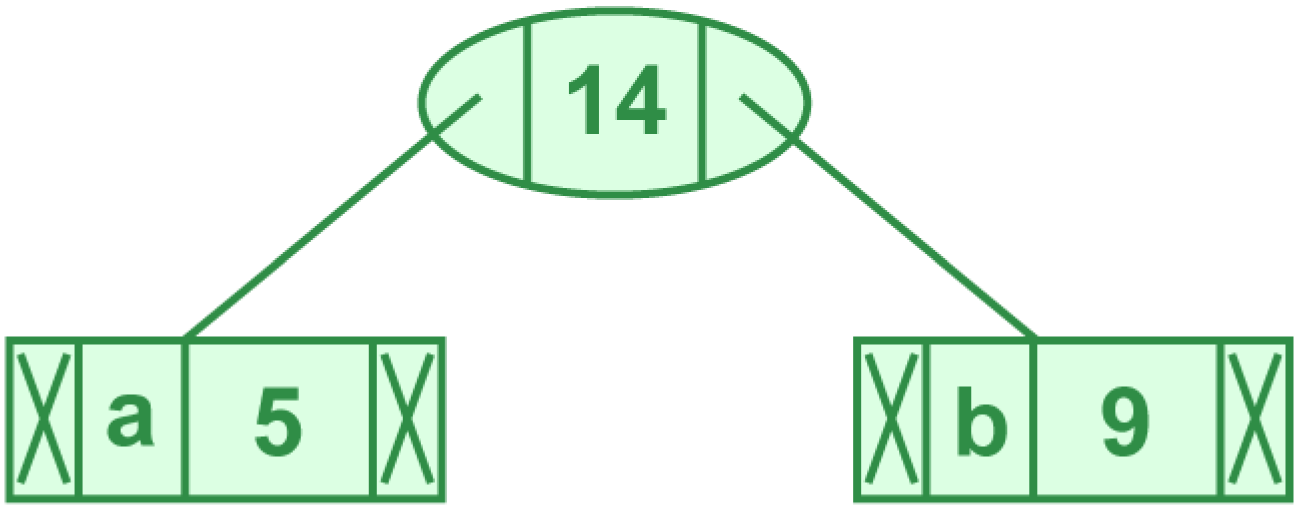


Illustration of step 2

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25

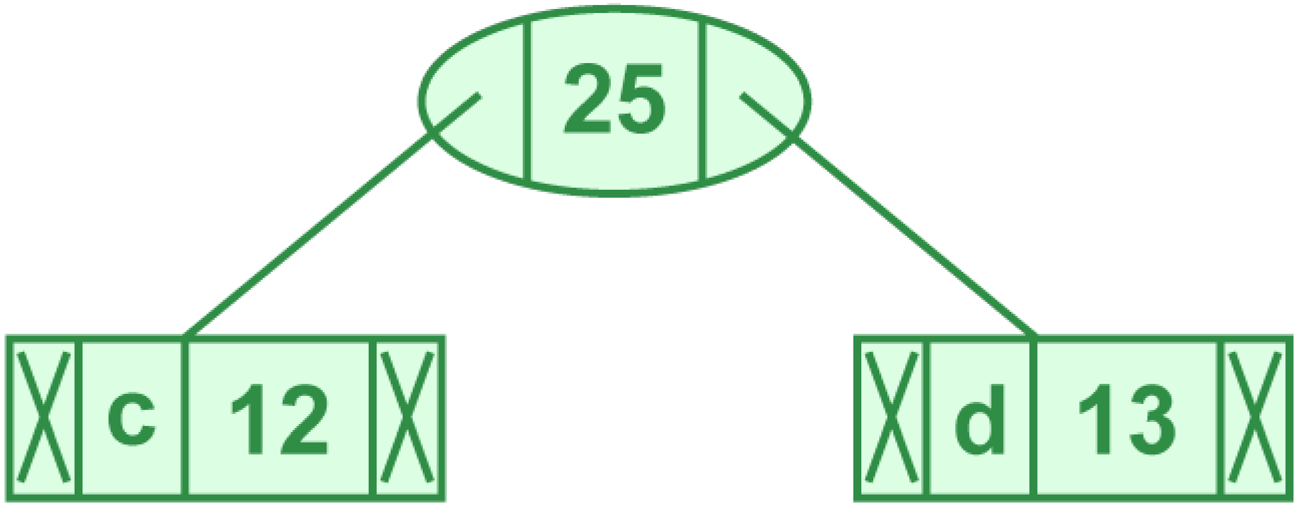


Illustration of step 3

Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30

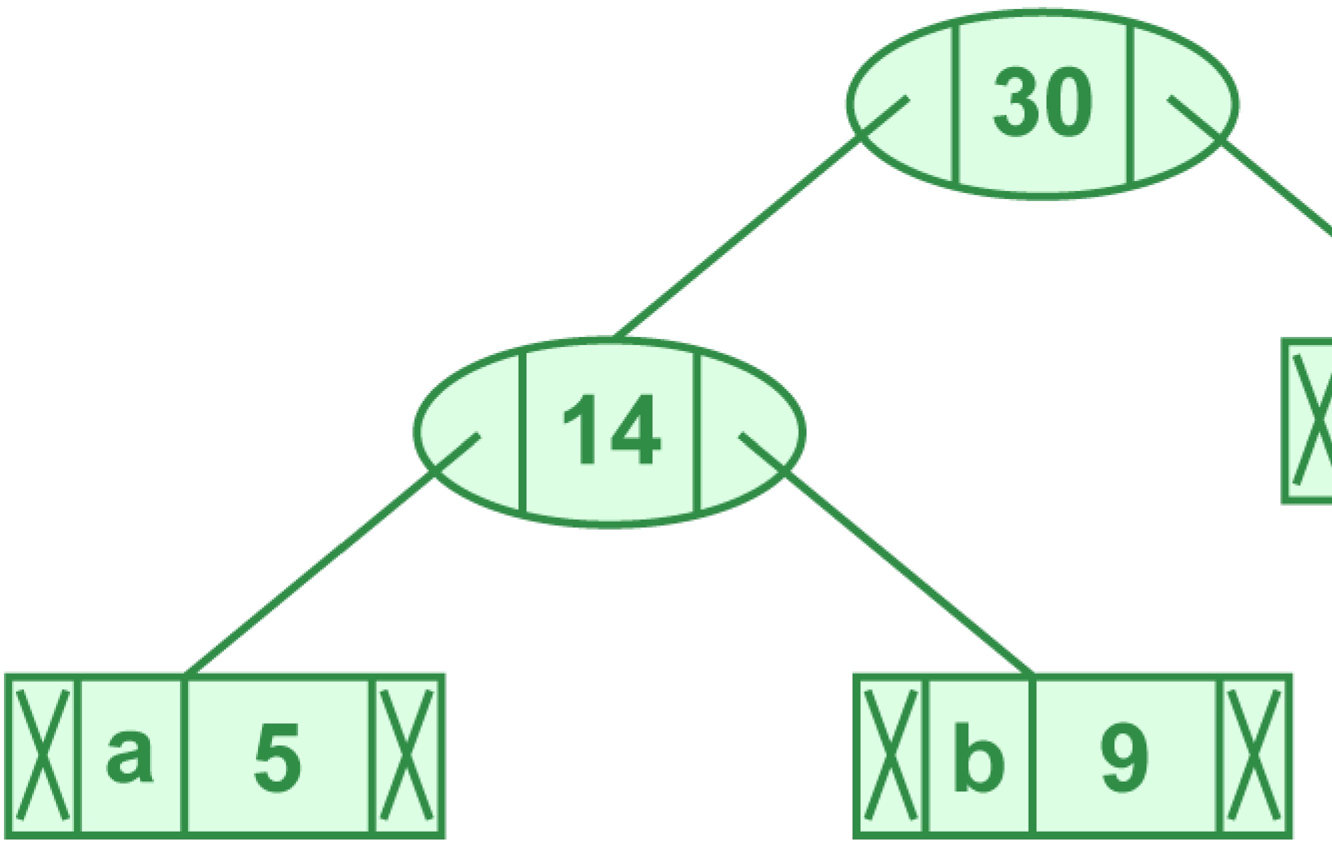


Illustration of step 4

Now min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30
f	45

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55

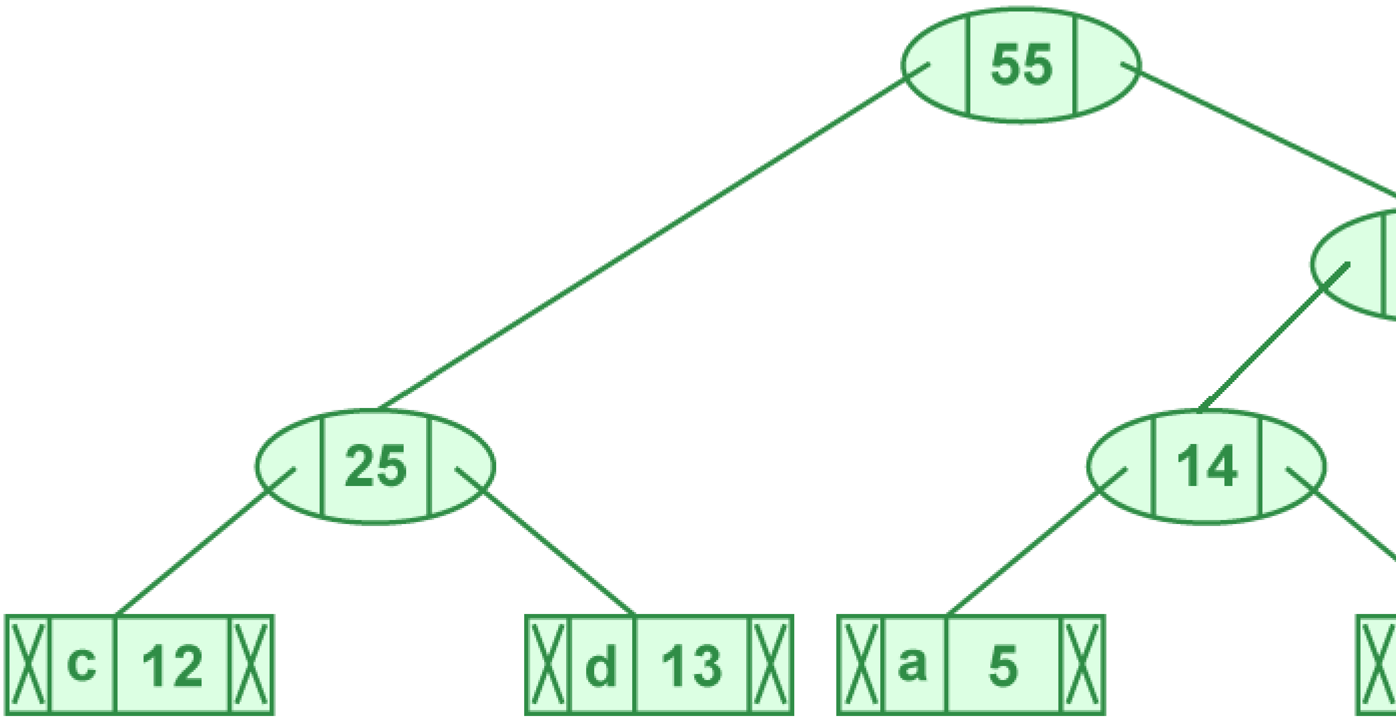


Illustration of step 5

Now min heap contains 2 nodes.

character	Frequency
f	45
Internal Node	55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100

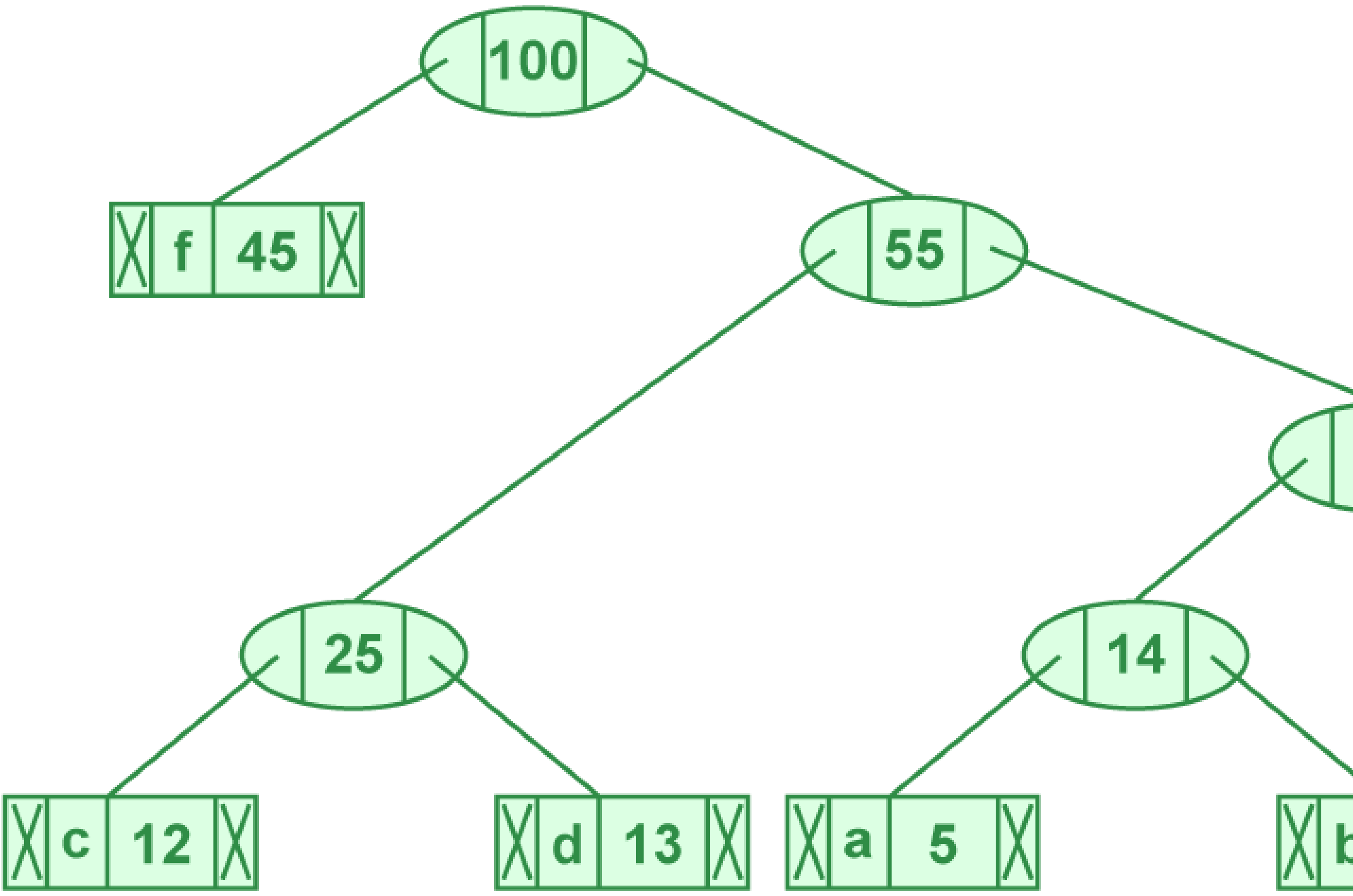
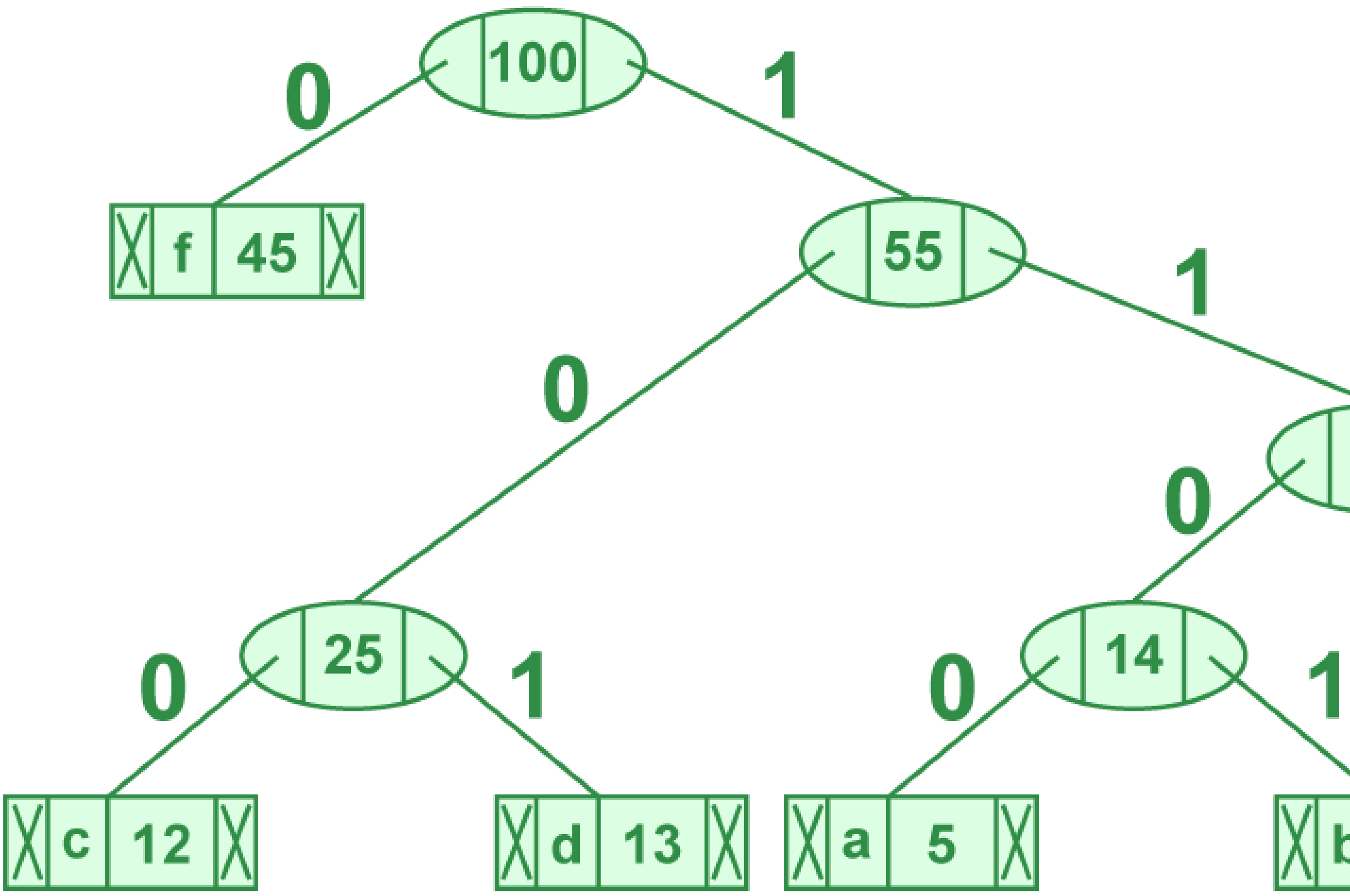


Illustration of step 6
Now min heap contains only one node.

character	Frequency
Internal Node	100

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



Steps to print code from HuffmanTree

The codes are as follows:

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

Output

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Applications of Huffman Coding:

1. They are used for transmitting fax and text.
2. They are used by conventional compression formats like PKZIP, GZIP, etc.
3. Multimedia codecs like JPEG, PNG, and MP3 use Huffman encoding(to be more precise the prefix codes).

It is useful in cases where there is a series of frequently occurring characters.