UNIT - 3

Transform and Conquer
Greedy Algorithms

UNIT - 3

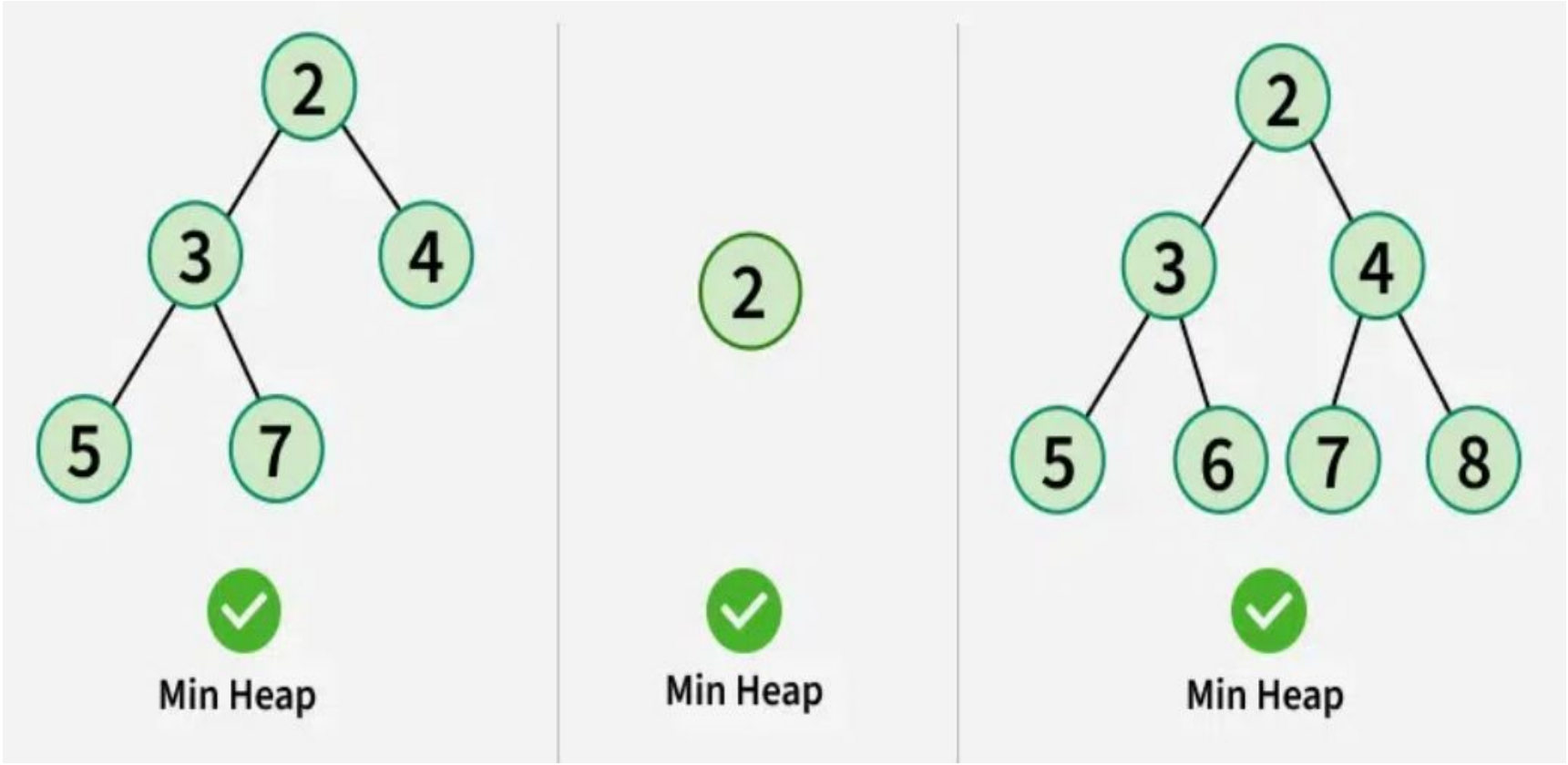
Transform and Conquer: Heaps and Heapsort**. (Textbook - 2, 6.4)**

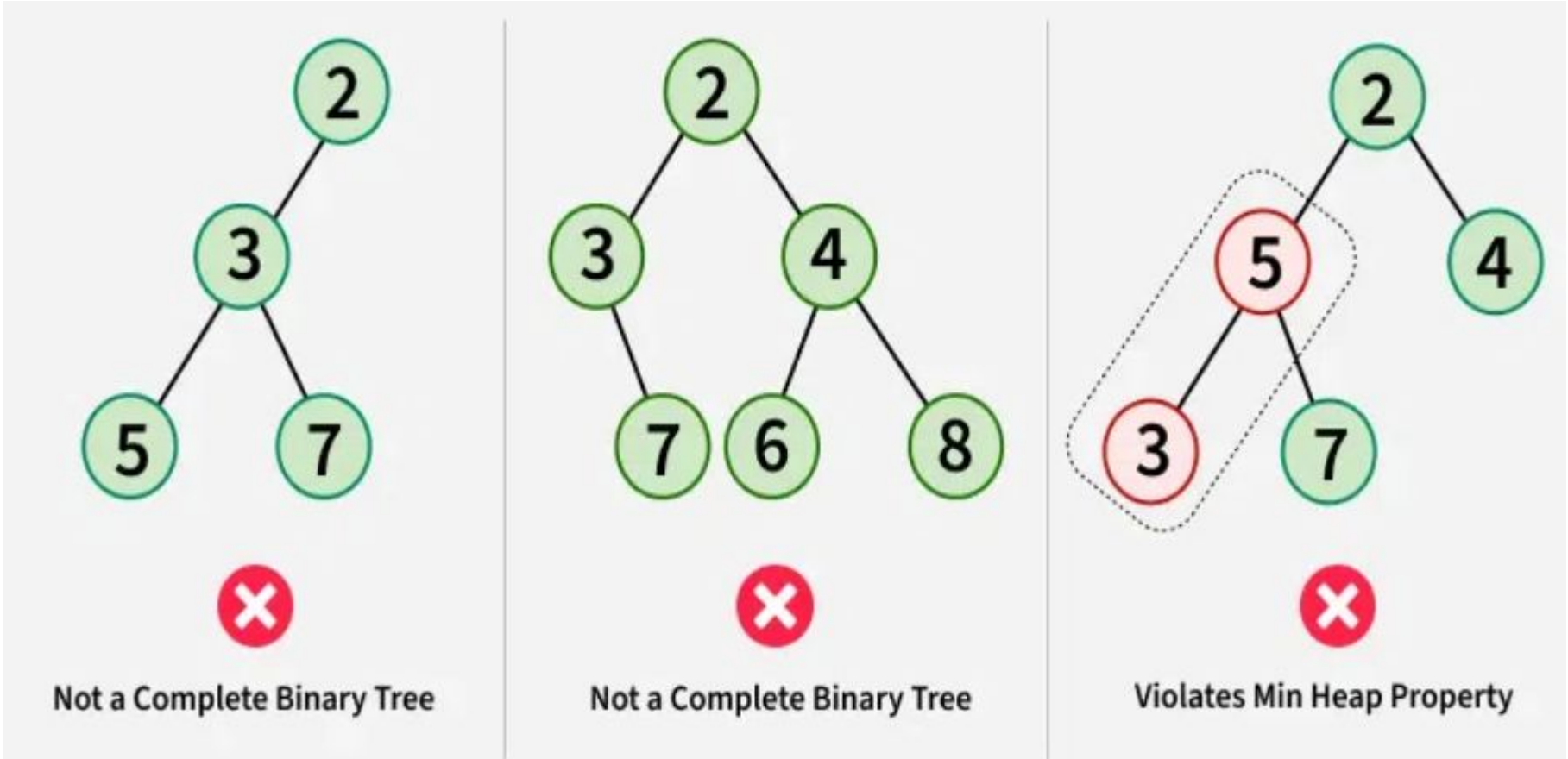
Greedy Algorithms: Interval Scheduling: The Greedy Algorithm Stays Ahead: Designing a Greedy Algorithm, Analyzing the Algorithm, Scheduling to Minimize Lateness: An Exchange Argument: The Problem, Designing the Algorithm, Designing and Analyzing the Algorithm. **(Text book - 1, 4.1,4.2)**

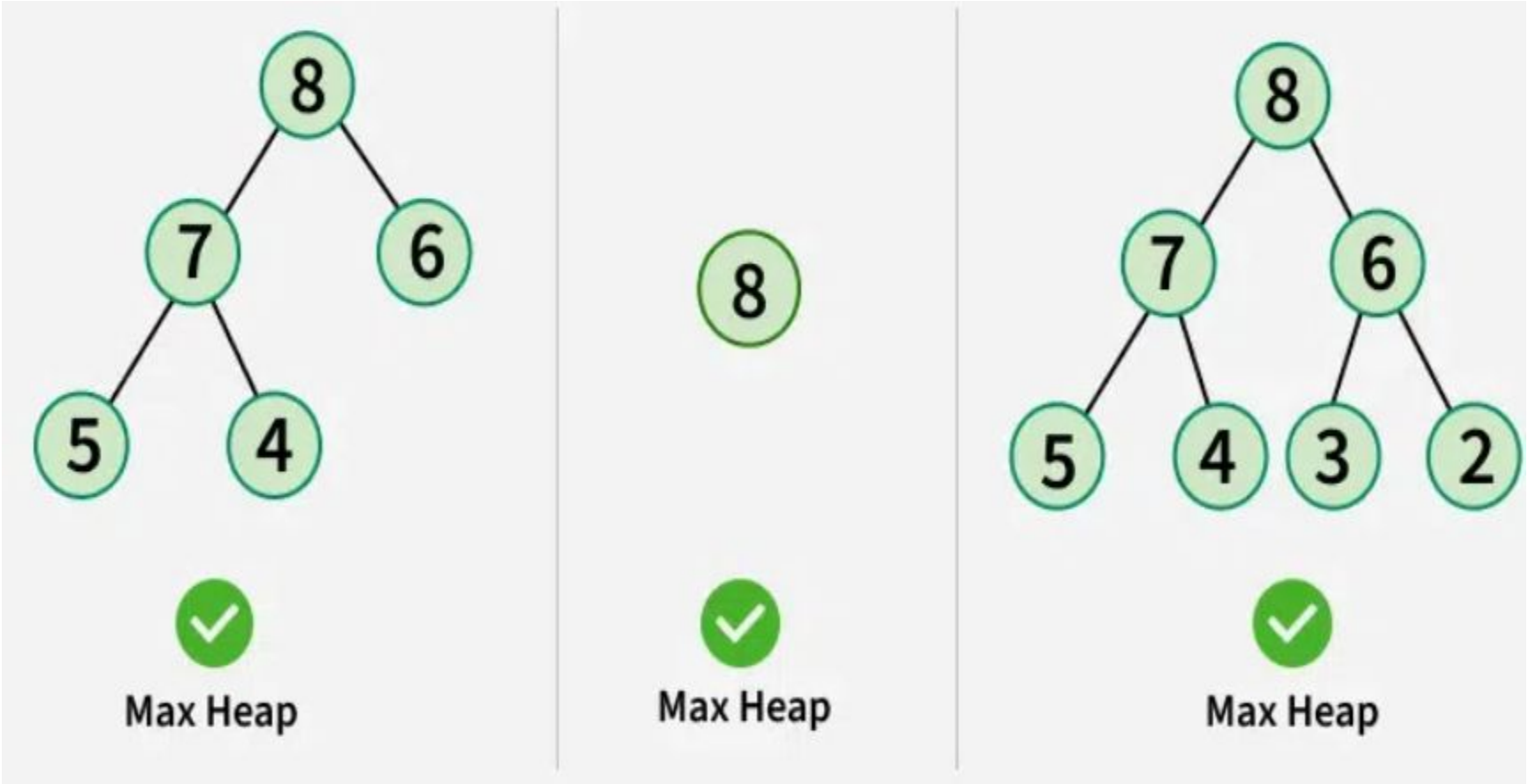
Prim's Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm, Huffman Trees and Codes. **(Text book - 2, 9.1,9.2,9.3,9.4)**

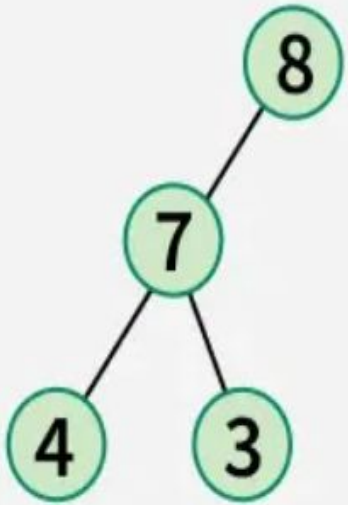
Heaps and Heapsort

- A **Heap** is a complete binary tree data structure that satisfies the heap property
- A **Binary Heap** is a complete binary tree that stores data efficiently, allowing quick access to the maximum or minimum element, depending on the type of heap. It can either be a Min Heap or a Max Heap.

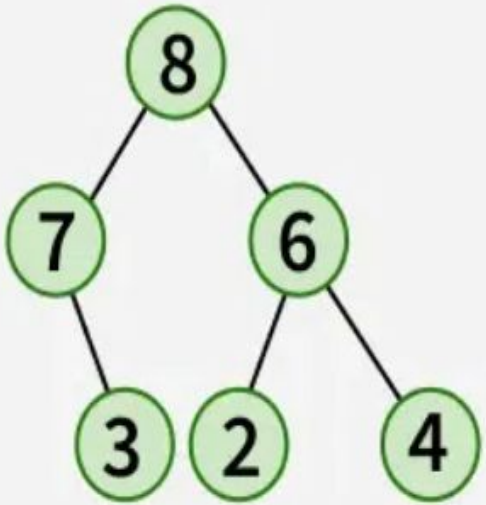

2
3
4
5
7
Min Heap
2
Min Heap
2
3
4
5
6
7
8
Min Heap


2
3
5
7
Not a Complete Binary Tree
2
3
4
7
6
8
Not a Complete Binary Tree
2
5
4
3
7
Violates Min Heap Property

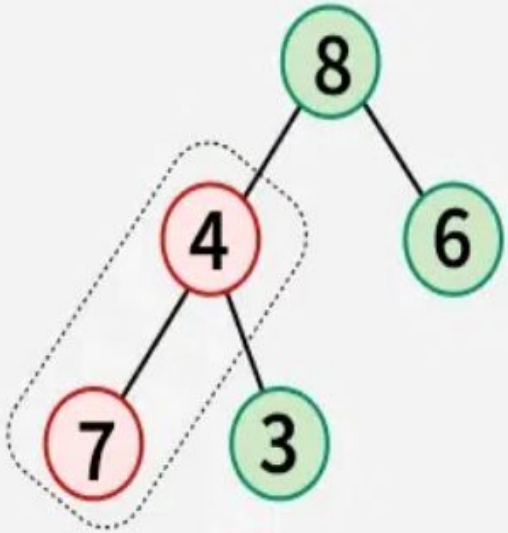

8
7
6
5
4
Max Heap
8
Max Heap
8
7
6
5
4
3
2
Max Heap



Not a Complete Binary Tree



Not a Complete Binary Tree



Violates Max Heap Property

Notion of the Heap

A heap can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

1. The shape property: the binary tree is essentially complete (or simply complete), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.

2. The parental dominance: the key in each node is greater than or equal to the keys in its children.

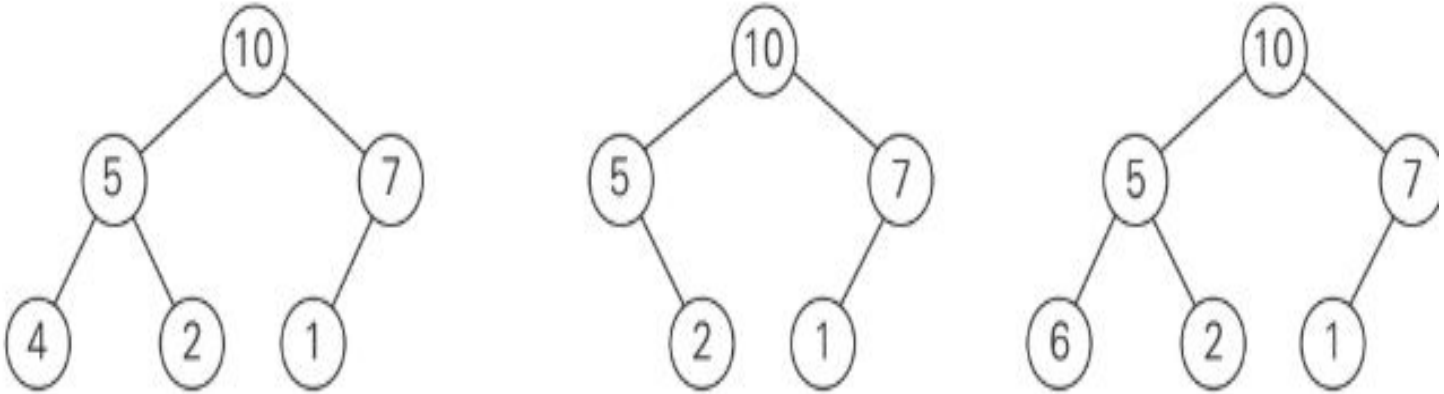


FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

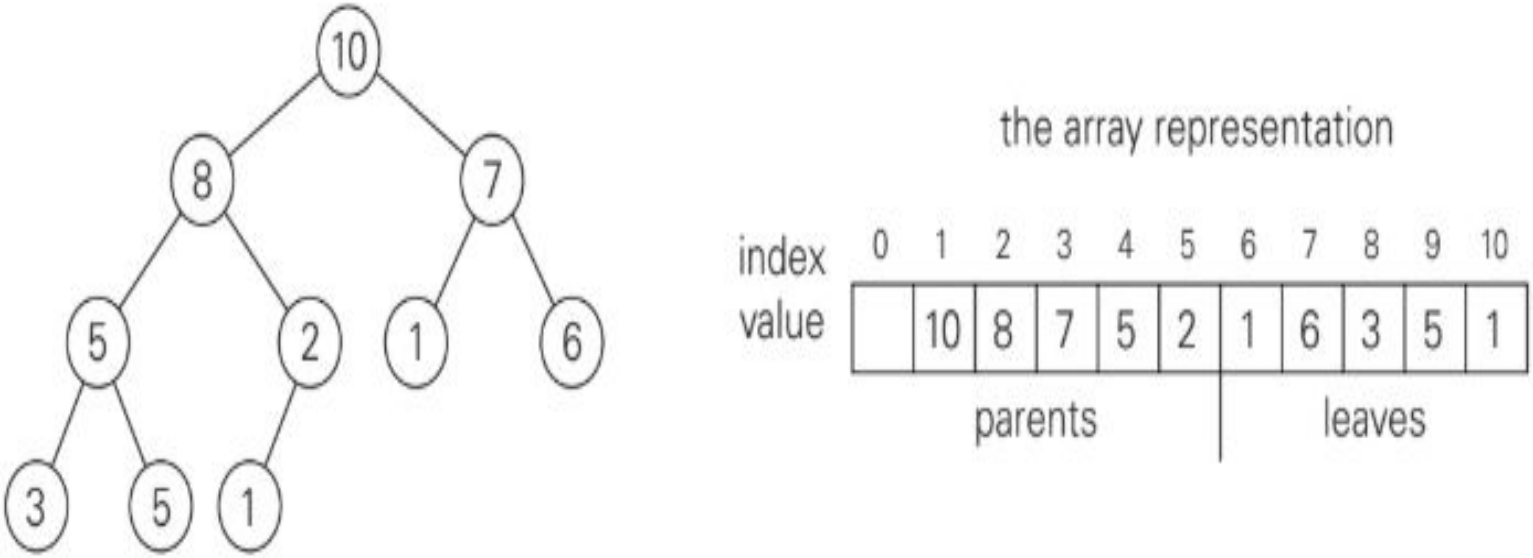


FIGURE 6.10 Heap and its array representation.

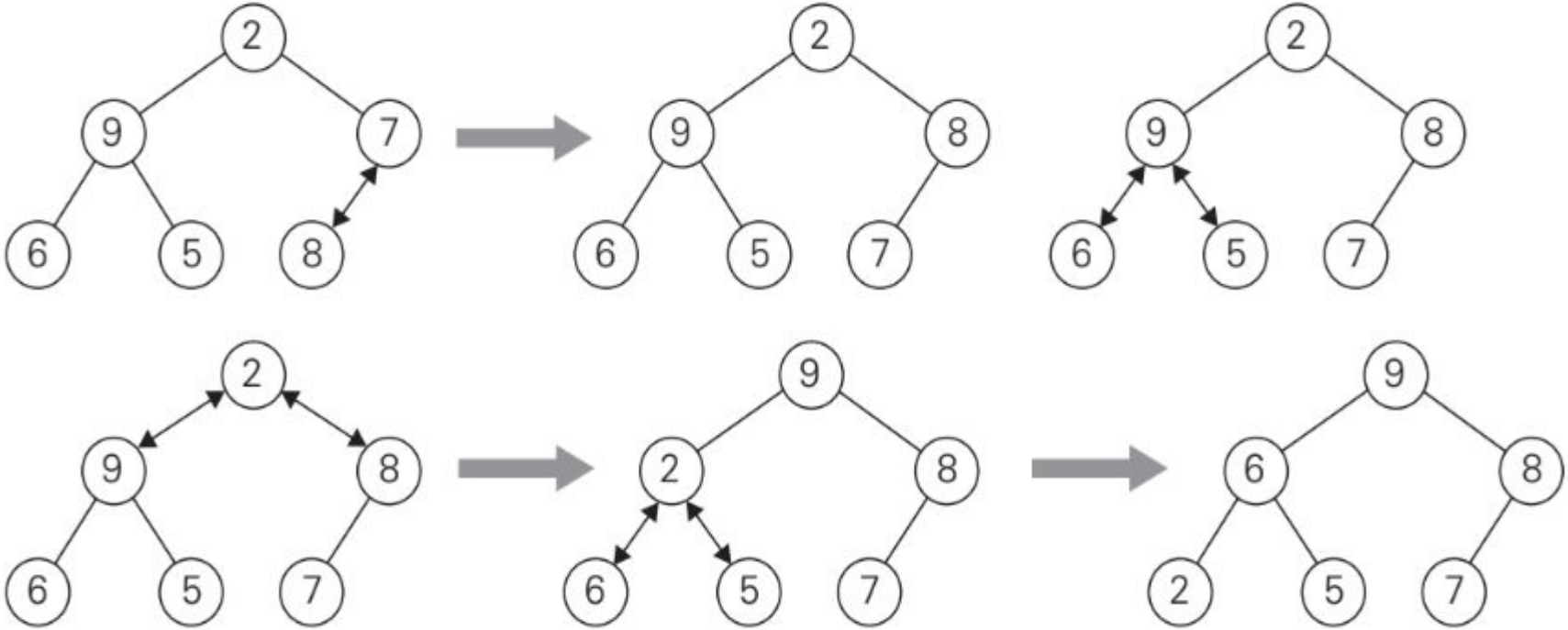


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

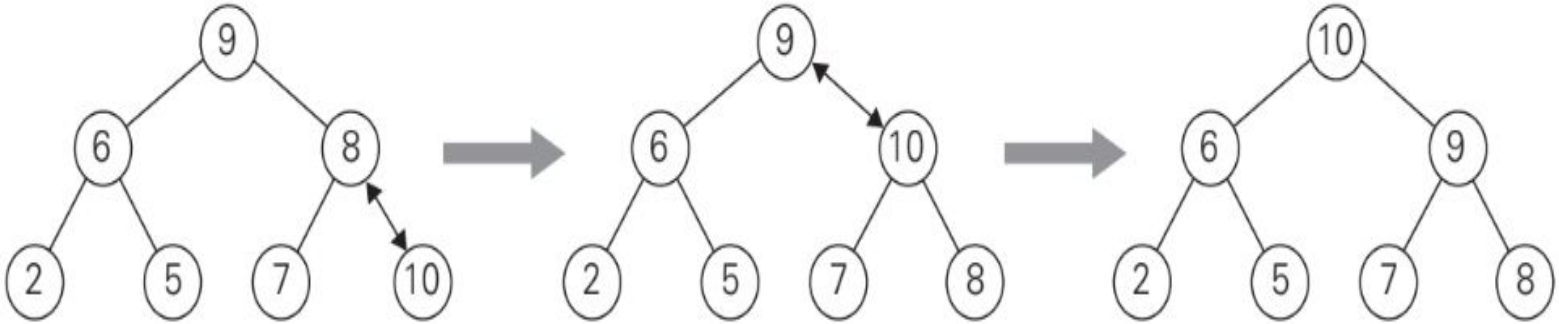


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

Note: Insertion operation cannot require more key comparisons than the heap's height. The time efficiency of insertion is in O(log n).

Maximum Key Deletion from a heap

Step 1: Exchange the root's key with the last key K of the heap.

Step 2: Decrease the heap's size by 1.

Step 3: "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm.

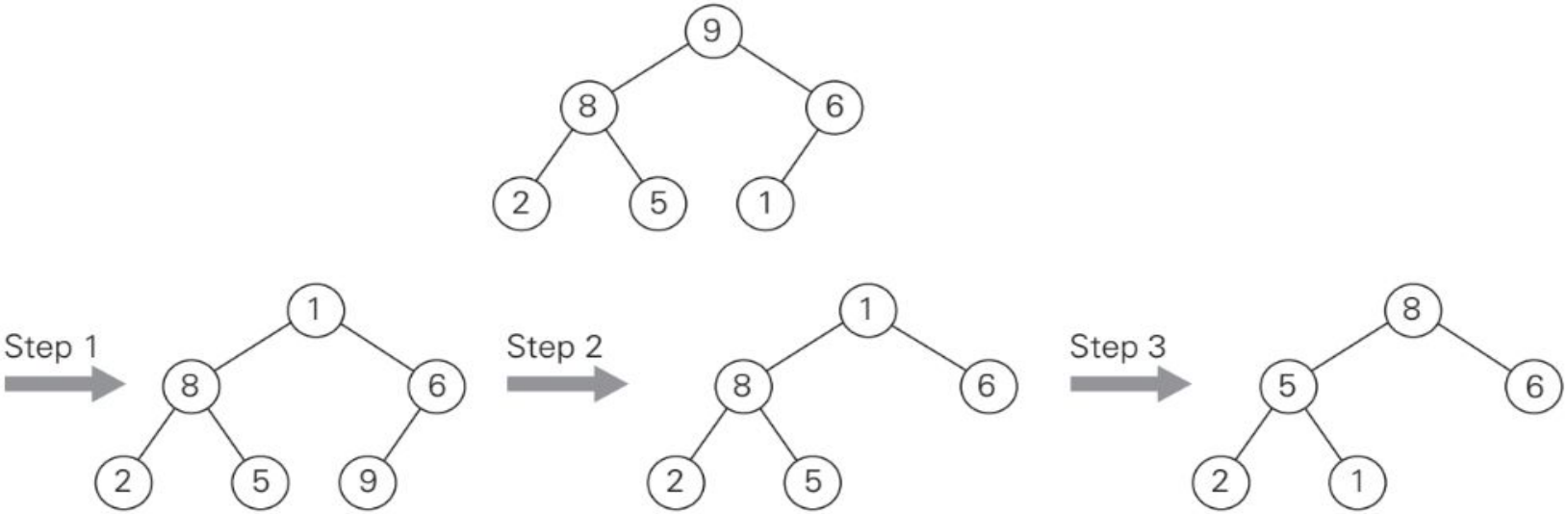


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

Note: Since this cannot require more key comparisons than twice the heap's height, the time efficiency of deletion is in O(log n) as well.

Heapsort

Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation n − 1 times to the remaining heap.

As a result, the array elements are eliminated in decreasing order. But since under the array implementation of heaps an element being deleted is placed last, the resulting array will be exactly the original array sorted in increasing order.

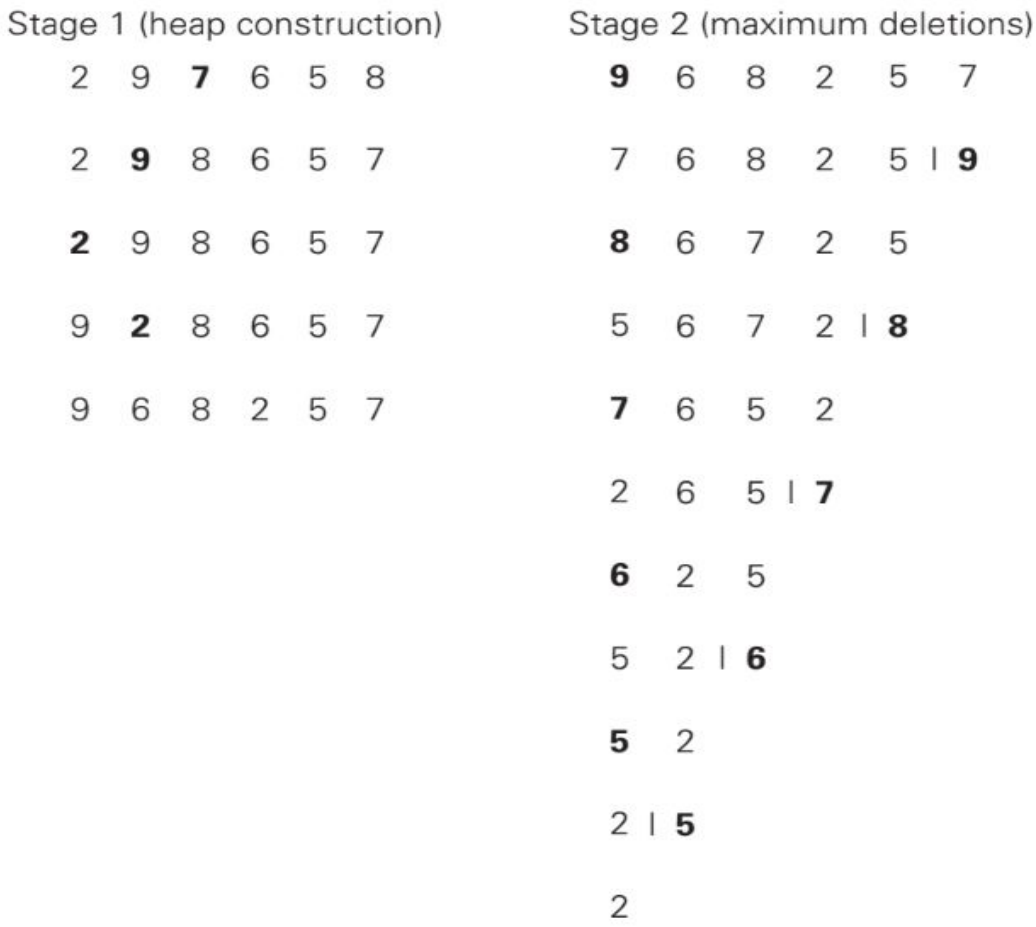
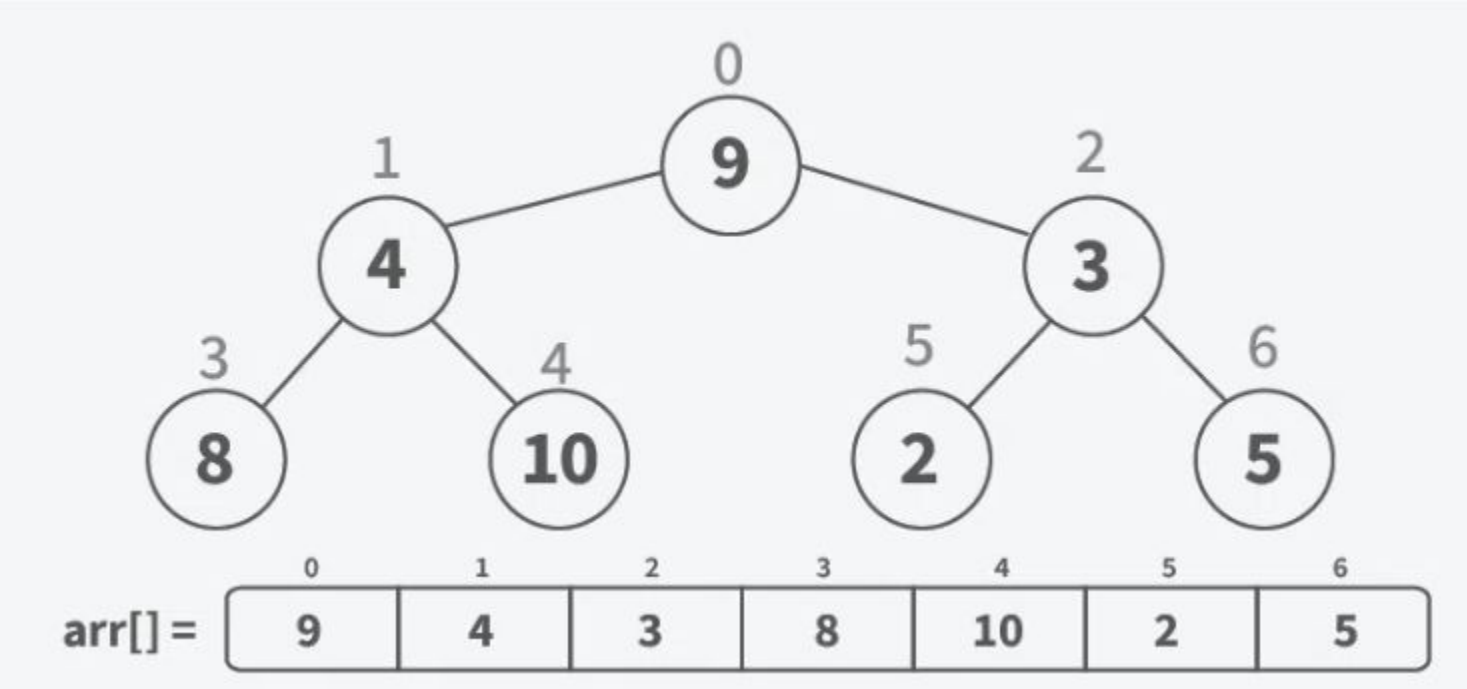


FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

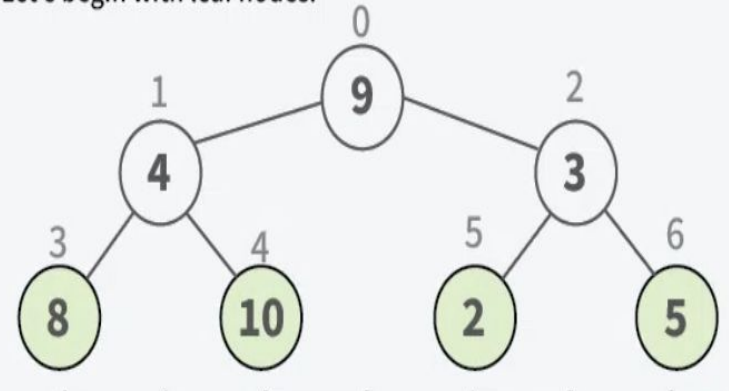
1. Treat the Array as a Complete Binary Tree

We first need to visualize the array as a **complete binary tree**

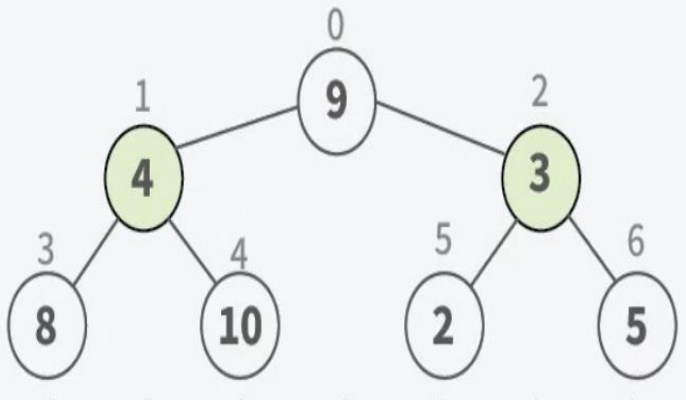


2.Build Heap Method (Max/Min)

01 Step Compare each node with its children, ensuring parent nodes are larger. This causes smaller nodes to bubble down and larger nodes to rise to the top. Let's begin with leaf nodes.

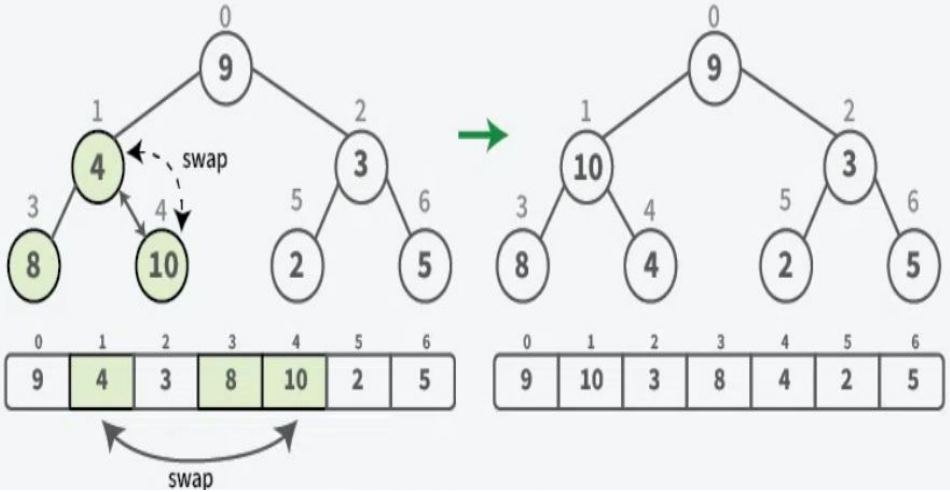


02 Step Let's look in the next upper level (node 4 and 3)



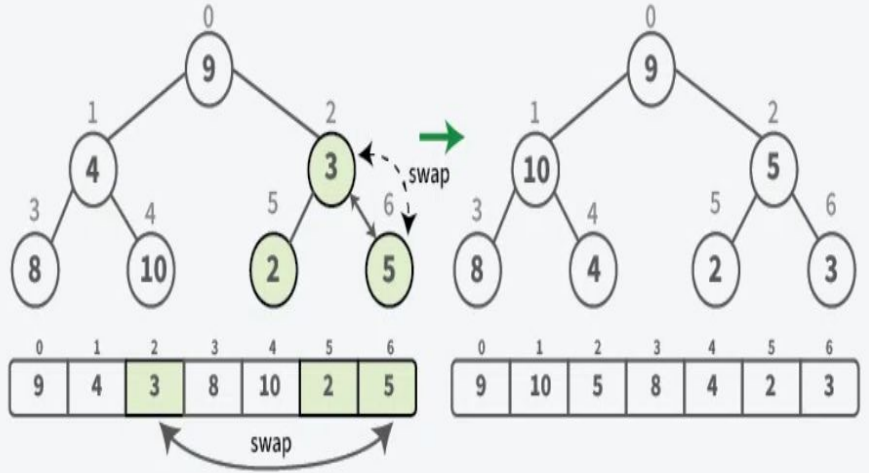
03 Step

Node 4 is smaller than its child node (10), so swap it with the larger child to maintain the property that the parent should be larger than its children.



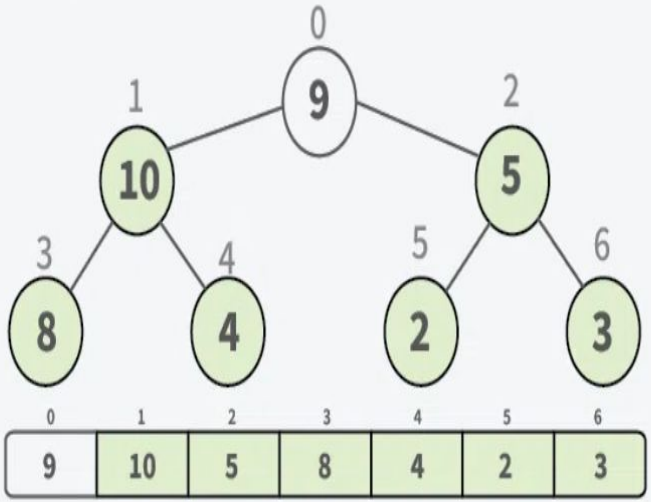
04 Step

Check for the next node (3) in current level. Since, it has a larger child, so swap it to ensure the parent has a larger value.



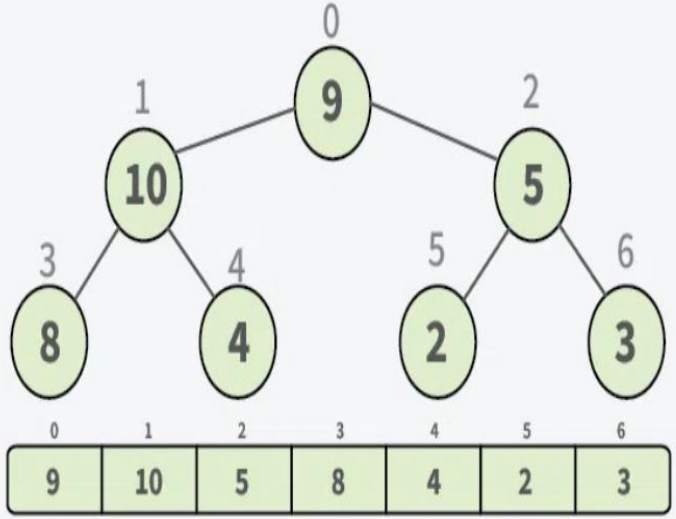
05 Step

After the completion of current level, we have now two smaller valid max heap



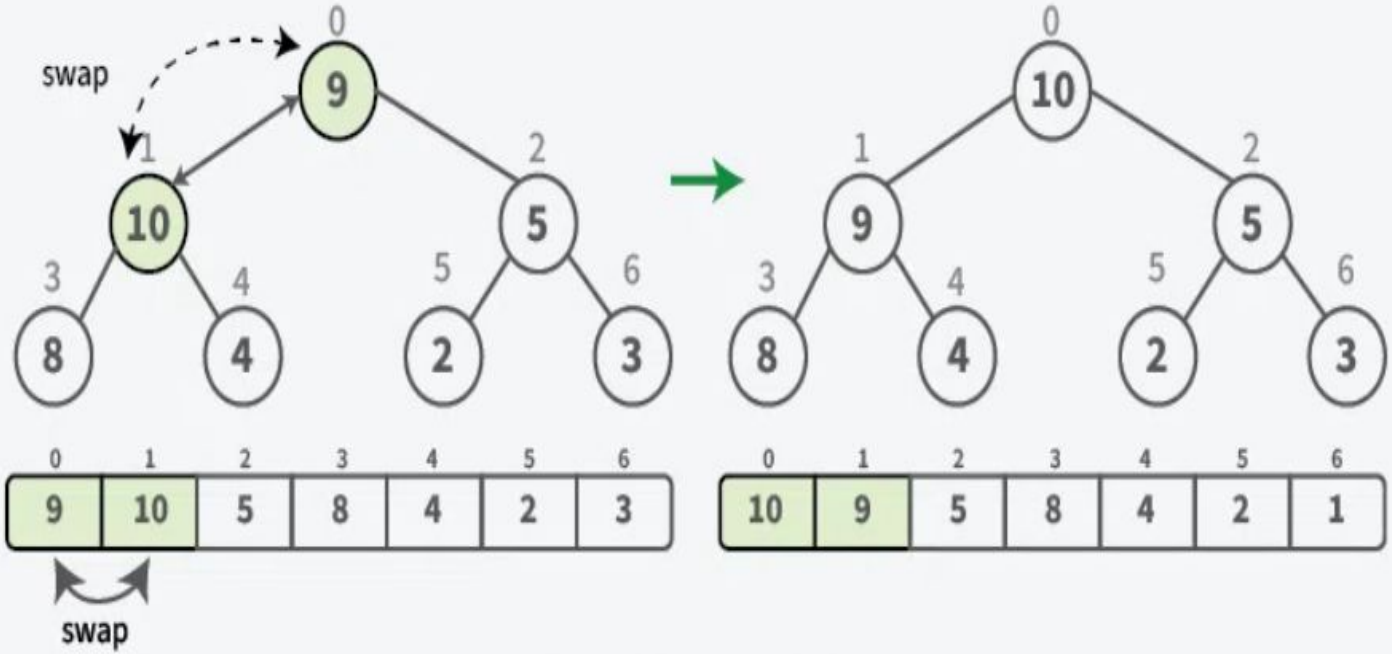
06 Step

Let's move to the next upper level. Here we've node 9 at the root.

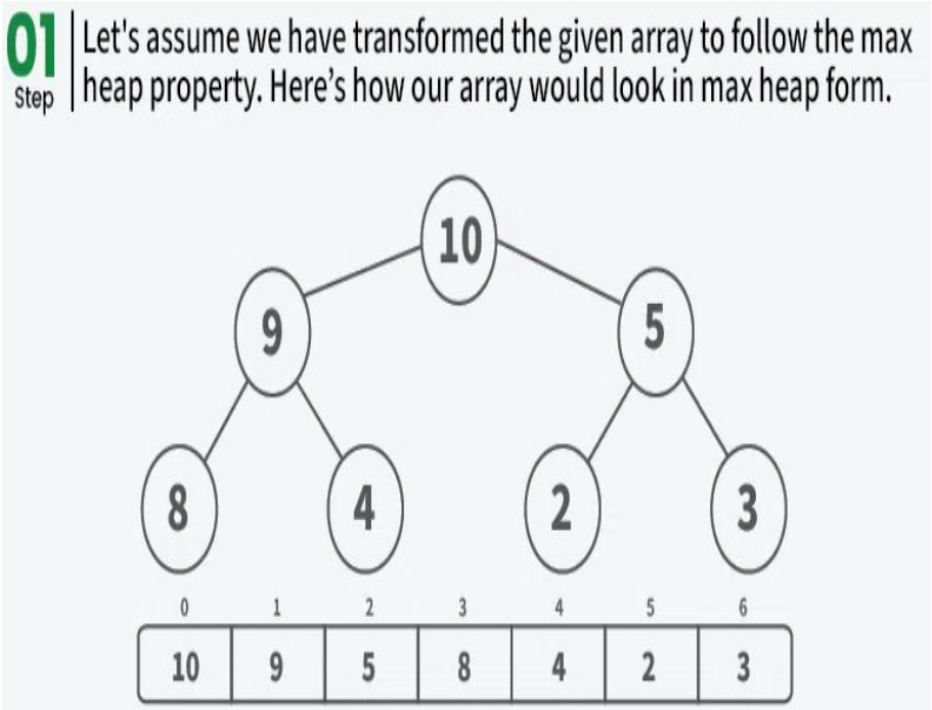


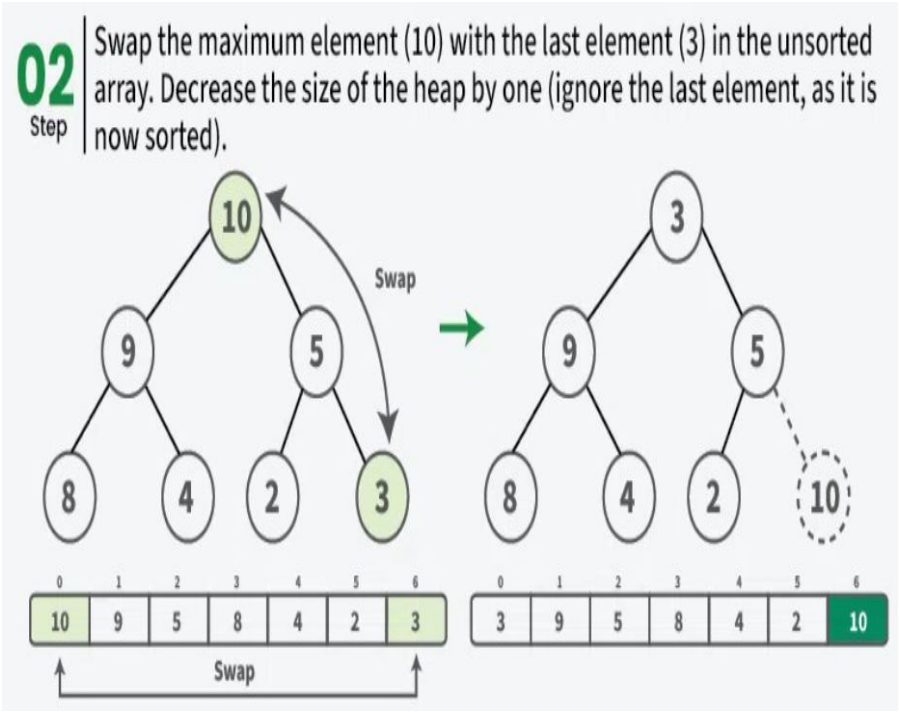
07 Step

Node 9 is smaller than its child node (10), so swap it with the larger child to maintain the property that the parent should be larger than its children.



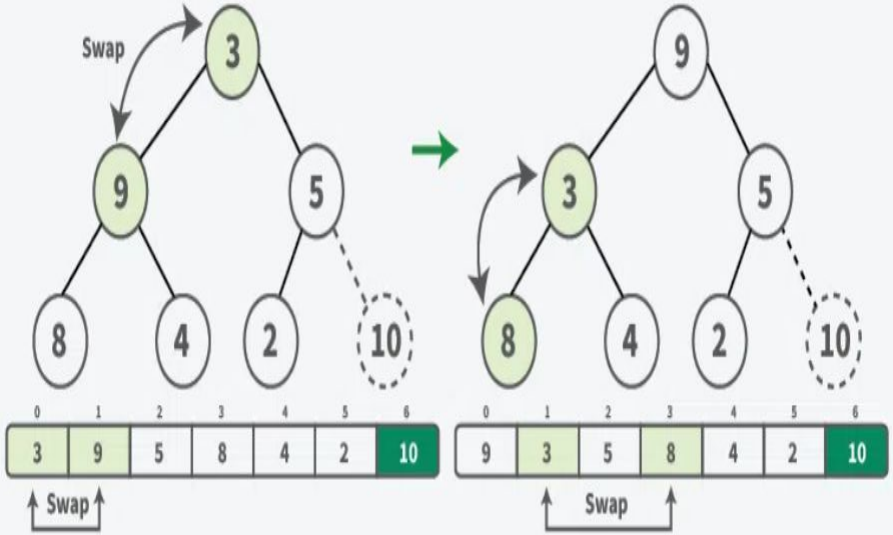
3.Sort array by placing largest element at end of unsorted array





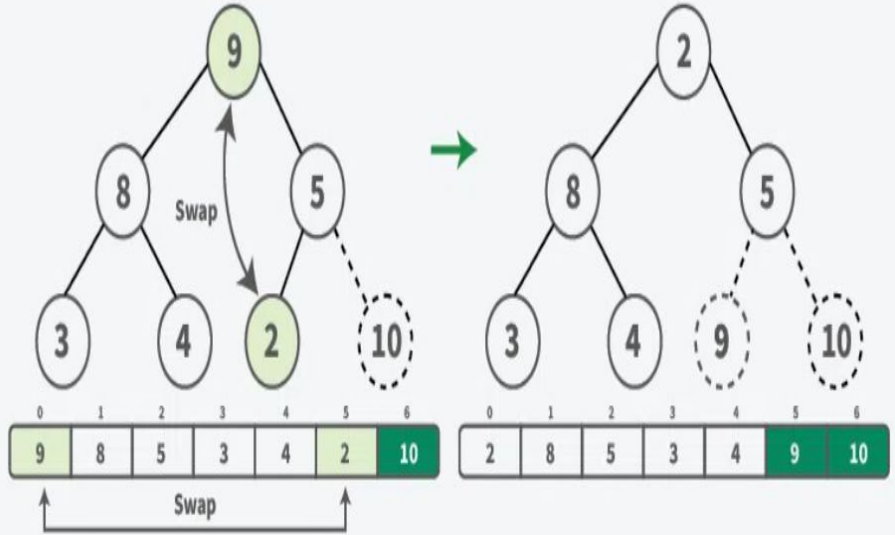
03 Step

Now, root violate the max-heap property, So, heapify it.
Swap node 3 with its largest child (9).
Still node 3 have larger children, so swap it with largest one (node 8).



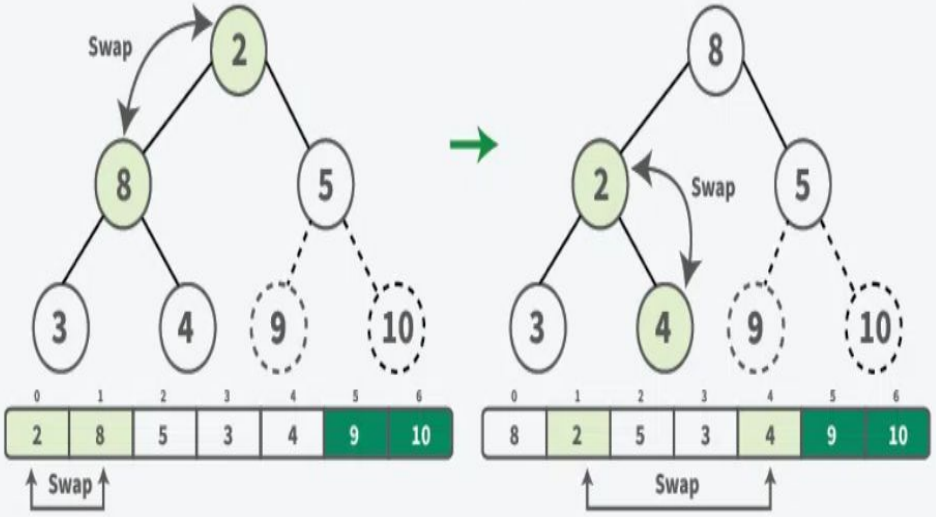
04 Step

Now, we have a max heap. Swap the maximum element (9) with the last element (2) in the unsorted array, then decrease the heap size by one (ignoring the second last element as it's now sorted).



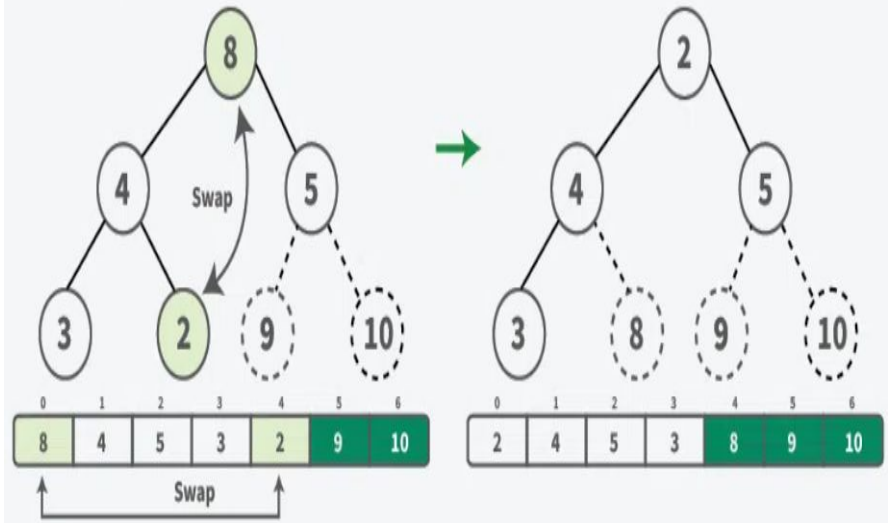
05 Step

Now, root violate the max-heap property, So, heapify it.
Swap node 2 with its largest child (8).
Still node 2 have larger children, so swap it with largest one (node 4)



06 Step

Now, we have a max heap. Swap the maximum element (8) with the last element (2) in the unsorted array, then decrease the heap size by one (ignoring the third last element as it's now sorted).



Time Complexity Analysis of Heap Sort

Time Complexity: O(n log n)

Auxiliary Space: O(log n), due to the recursive call stack. However, auxiliary space can be O(1) for iterative implementation.

Applications of Heaps

- **Heap Sort:** Heap Sort uses Binary Heap to sort an array in O(nLogn) time.
- **Priority Queue**: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() operations in O(log N) time.
- **Graph Algorithms:** The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.

Problem Solving Technique

1. Construct a heap for the list **1, 8, 6, 5, 3, 7, 4** by the **bottom-up algorithm.** Specify the property of building Max Heap. Maximum Key Deletion from a heap.
2. Construct a heap for the list **1, 8, 6, 5, 3, 7, 4** by successive key insertions **(top-down algorithm).**

Greedy Algorithms: Interval Scheduling: The Greedy Algorithm Stays Ahead

- Follows local optimal choice of each stage with intent of finding global optimum
- Feasible solution (Selection)
- Optimal solution (Min. Cost, Max. Profit)

More Info:

https://www.youtube.com/watch?v=XmMSv5xHi9A&list=PLMLFEbzd52KPTn1_ZCjYsNpYOXXzmjfwM&index=2

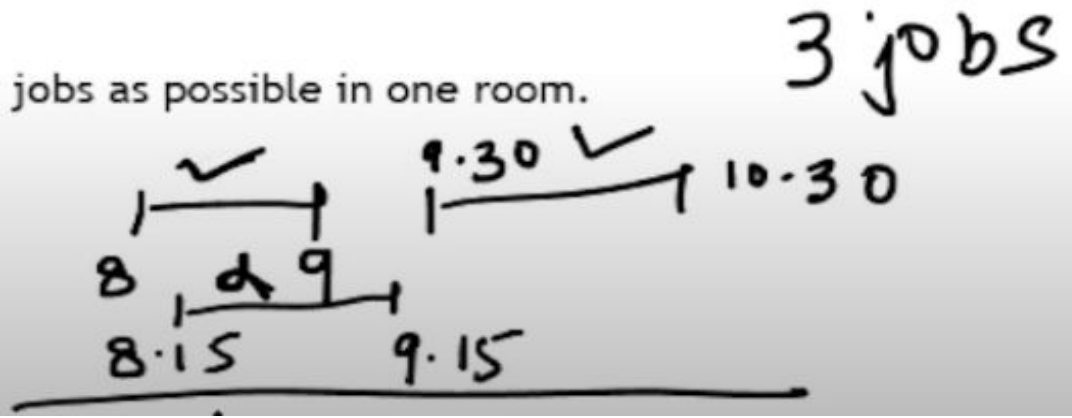
Interval Scheduling Problem

Problem Statement:

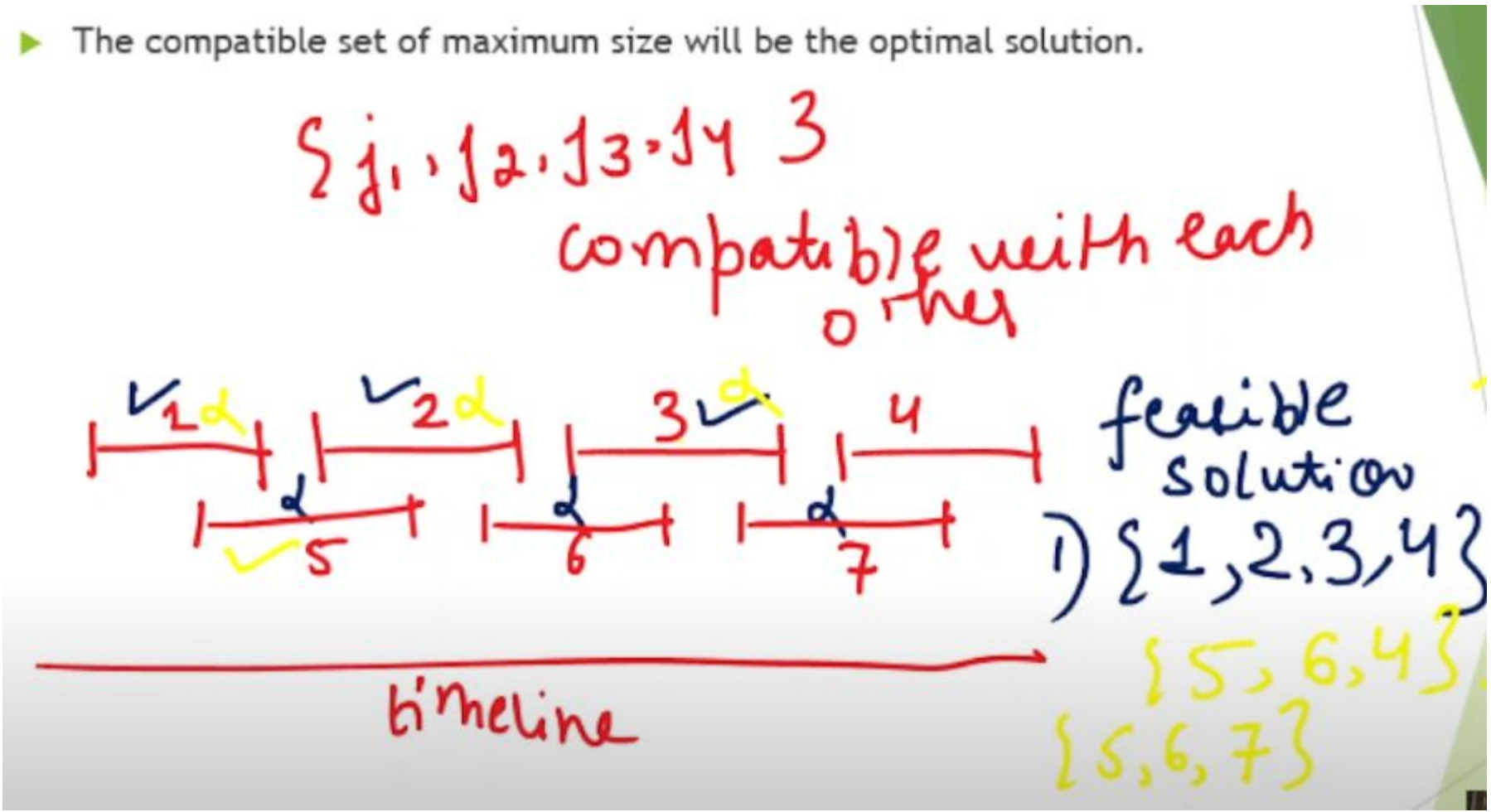
Given a set of jobs or requests {1,2.....n} with respective start time s(i) and finish time f(i). We have to schedule all these jobs in one room.

Constraint : No 2 jobs overlap in time that is they are compatible.

Goal: Schedule as many jobs as possible in one room.



- The compatible set of maximum size will be the optimal solution.



Interval Scheduling Problem helps to illustrate how greedy algorithms work in practice.

Greedy Algorithm Approach:

- Use a simple selection rule to pick the **first request (i_1)**. **Accept i_1** and **reject all overlapping (incompatible)** requests.
- Move on to select the **next compatible request (i_2)**. Repeat this process until **no more compatible requests remain**.

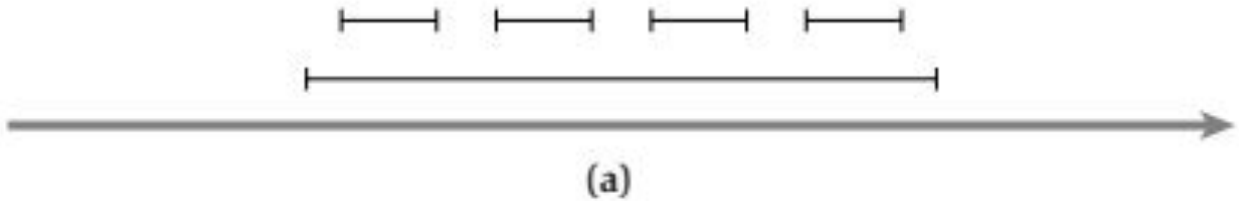
The **core challenge** is to **choose right rule** for selecting requests.

Designing a Greedy Algorithm:

The most obvious rule might be to always select the available request that **starts earliest**—that is, the one with minimal start time s(i). This way our resource starts being used as quickly as possible.

More Info:

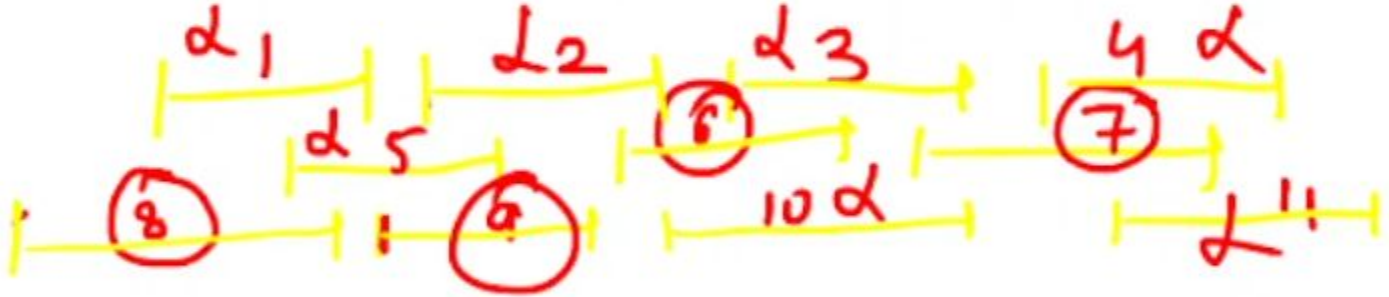
https://www.youtube.com/watch?v=NRmx01NrZl0&list=PLMLFEbzd52KPTn1_ZCjYsNpYOXXzmjfwM&index=3



How do we choose the jobs such that max. jobs are scheduled ?

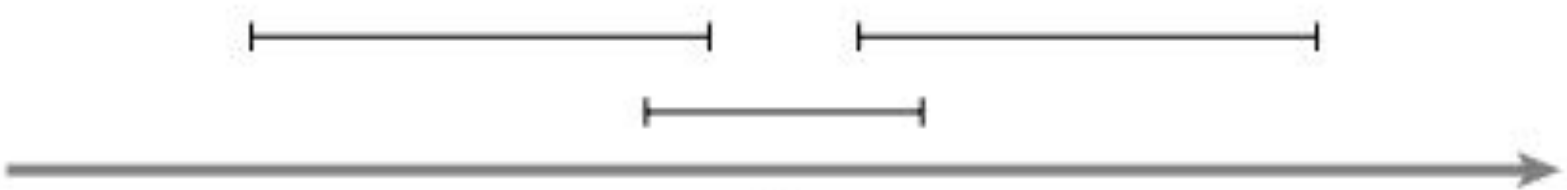
Greedy Strategy 1

Process jobs in the order of their starting time that is earliest start time.



{ 8, 9, 6, 7 }

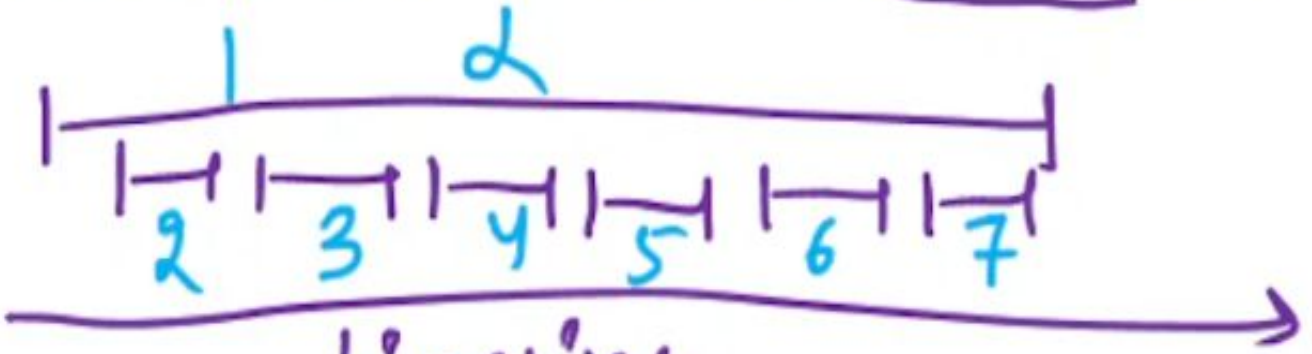
- This might suggest that we should start out by accepting the request that requires the **smallest interval of time**—namely, the request for which f(i) − s(i) is as small as possible.



(b)

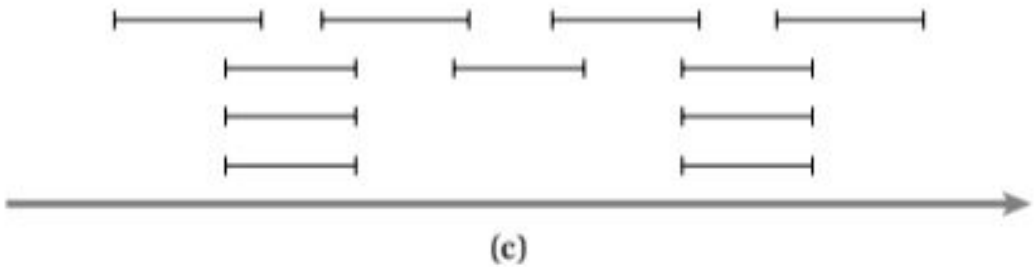
Greedy Strategy 2

Select job that have the smallest processing time that is f(i) - s(i) is least.



{ 2, 3, 4, 5, 6, 7 } → solution

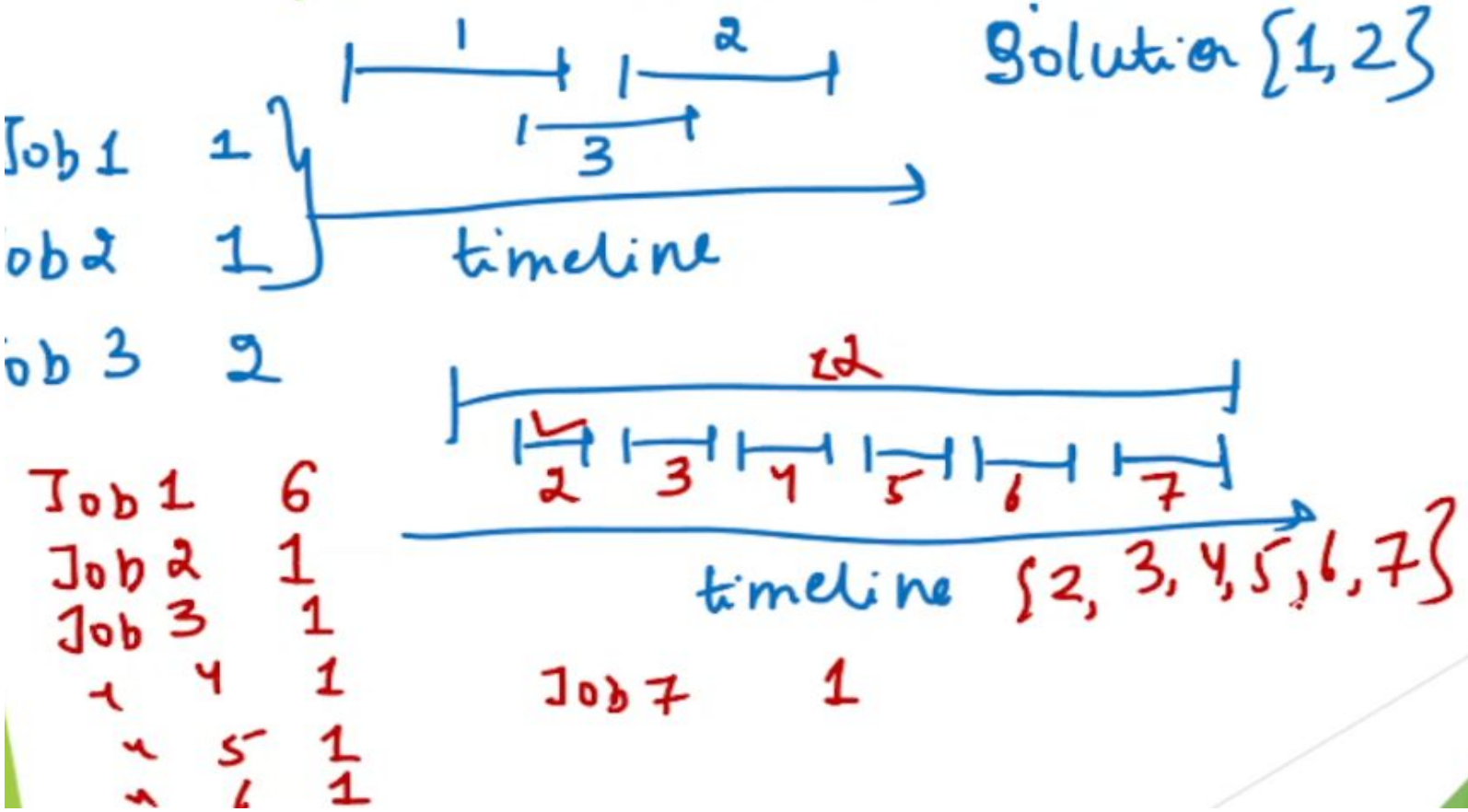
- In the previous greedy rule, our problem was that the second request competes with both the **first and the third**—that is, accepting this request made us reject two other requests. We could design a greedy algorithm that is based on this idea: for each request, we count the number of other requests that are **not compatible,** and **accept the request** that has the fewest number of non compatible requests.

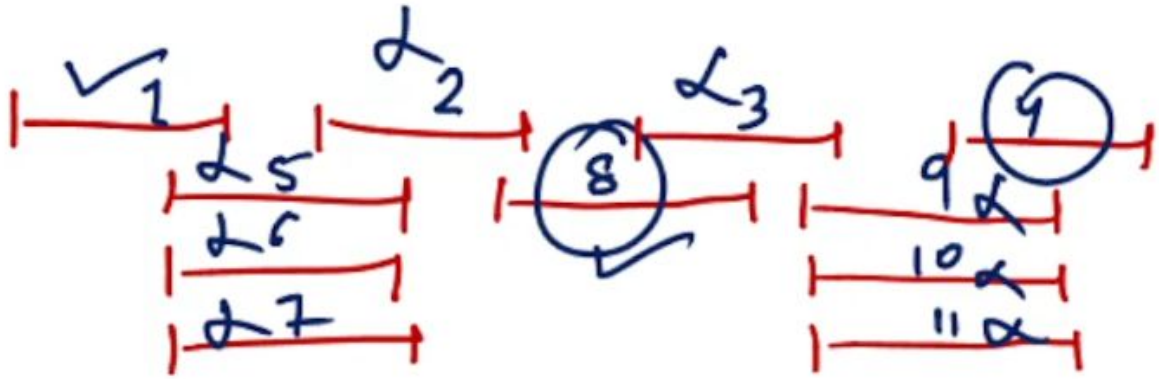


(c)

Greedy Strategy 3

Process jobs in the order that have the fewest conflicts.





timeline

{8, 1, 4}

Job 1 (3)
- 5
- 6
- 7

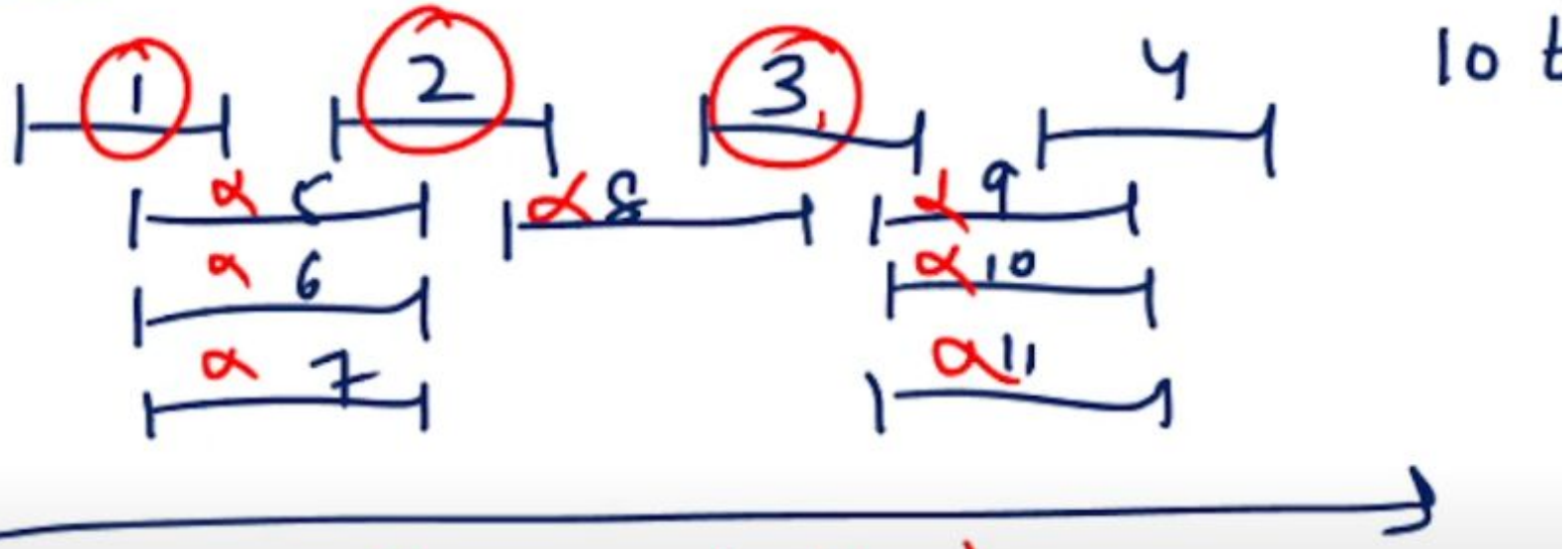
Job 2
Job 8 (2)

Job 3 (2)
Job 9
- 10
- 11

Job 4 (3)

Greedy Strategy 4 → yield optimal result

Process jobs in the order of their finishing times that is the job for which f_i is as least as possible



{1, 2, 3, 4} → solution

Algorithm

Initially let R be the set of all requests, and let A be empty

While R is not yet empty

  Choose a request $i \in R$ that has the smallest finishing time

  Add request i to A

  Delete all requests from R that are not compatible with request i

EndWhile

Return the set A as the set of accepted requests

Analyzing the Algorithm

While this greedy method is quite natural, it is certainly not obvious that it returns an optimal set of intervals. Indeed, it would only be sensible to reserve judgment on its optimality: the ideas that led to the previous non-optimal versions of the greedy method also seemed promising at first.

More Info:

https://www.youtube.com/watch?v=S0mgmfGnEPk&list=PLMLFEbzd52KPTn1_ZCjYsNpYOXXzmjfwM&index=4

Analyzing the Algorithm

Claim 1: A is a compatible set of requests

- Our Algorithm produces a solution A
- Let O be any optimal solution
- A and O may not be identical as there are multiple ways of producing solutions of same size.

Just show that ,

$|A| = |O|$

trivial

feasible

$A \{ j_1, j_2, j_4 \cdots \}$

optimal solution O

$|A| \{ j_1, j_2, j_5 \}$

$|O| \{ j_6, j_7 j_8 \}$

$|A| = |O|$

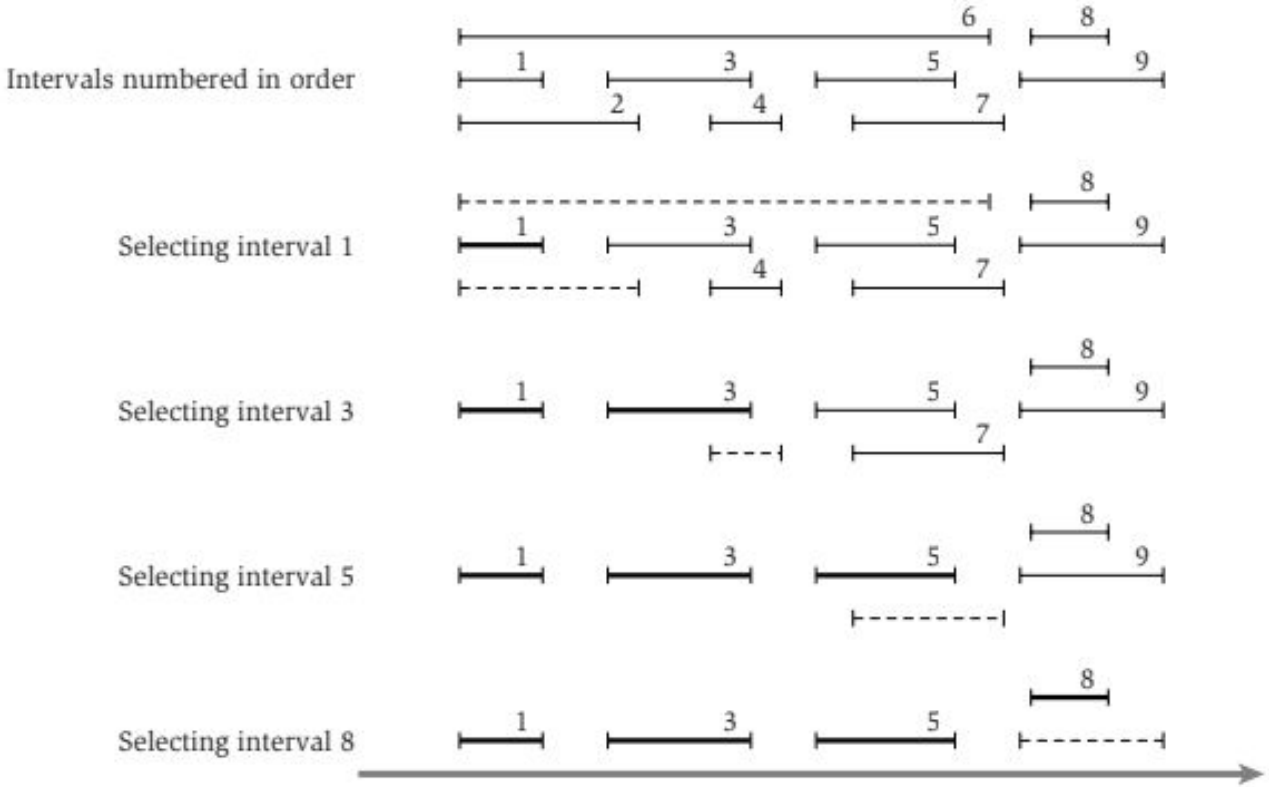


Figure 4.2 Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

Let $A = i_1, i_2 \cdots i_k$ sorted order

$f(i_1) \leq s(i_2)$, $f(i_2) \leq s(i_3)$ ....

Let $O = j_1, j_2 \cdots j_m$

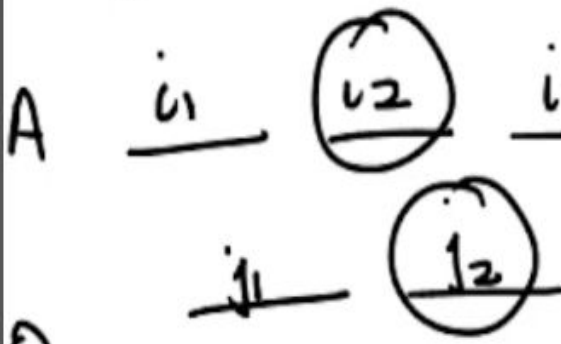
$f(j_1) \leq s(j_2)$, $f(j_2) \leq s(j_3)$

{Assume O contains all the jobs arranged on the basis of their start of finish time}

To show that $\boxed{k = m}$

ie. optimal solution of the same size as our greedy

for all indices $r \leq k$ $f(i_r) \leq f(j_r)$ claim



for each job in the sequence i and j, the corresponding job in A sequence finishes no later than the corresponding job in O sequence.

The greedy solution stays ahead than the solution produced by O.

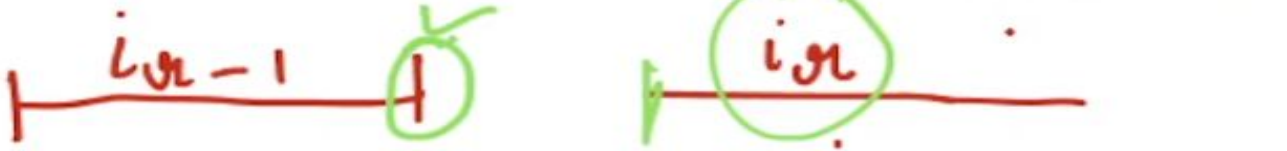
Proof: By induction on r $f(i_r) \leq f(j_r)$

for $r = 1$, $\boxed{f(i_1) \leq f(j_1)}$ A i_1 j_1

is true because among all the jobs the greedy strategy chooses i_1 with earliest finishing time among all the jobs.

$r > 1$: Assume by induction that the statement is true for $r-1$

$$f(i_{r-1}) \leq f(j_{r-1}) \quad -(1)^{r-1}$$

A 

O 

We will proove for r

O consists of compatible intervals

$$f(j_{r-1}) \leq s(j_r) \quad -(2)$$

$f(i_{r-1}) \leq s(j_r)$

So our algorithm elects the interval

which has smallest finishing time even though it was one of the available intervals, our algorithm is selecting it.

It means that

$$f(i_1) \leq f(j_1)$$

This means that greedy solution stays ahead than the optimal This completes the induction.

Claim 3

The greedy algorithm returns an optimal set A.

$A = \{i_1, i_2 \cdots i_k\}$

$O = \{j_1, j_2 \cdots j_m\}$

$m = k$

Proof:

using contradiction,

O contains more jobs

$m > k$

$r \leq R$, $f(i_r) \leq f(j_r)$ — already pro

Let's take $r = R$

$$f(i_R) \leq f(j_R)$$

A $i_1 \cdots \underline{i_{k-1}} \; \underline{i_R}$ Since optimal solution is larg

$\underline{j_R} \quad \underline{j_{R+1}}$ than R,

O

j_{R+1} starts after i_R ends,

j_{R+1} is compatible with i_R

$\therefore$ j_{R+1} is compatible with all

requests from $i_1 \cdots i_k$ R {all jobs}

we deleted all the A $\{i_1, i_2 \cdots i_k\}$

jobs or requests from R

that were not compatible with

$i_1 \cdots i_k$

Set R should have contained the job j_{k+1}

our greedy algorithm stops after

seleiting the job i_k

So our algorithm is supposed to

Stop after R is empty or a contradiction

which means we cannot have any jobs larger than k in our optimal solution. So

$$m = k$$

Scheduling to Minimize Lateness: An Exchange Argument

The Problem: Consider again a situation in which we have a single resource and a set of **n** requests to use resource for an interval of time. Assume that resource is available starting at time **s**. Instead of a start time and finish time, request **i** has a deadline **di**, and it requires a contiguous time interval of length **ti,** but it is willing to be scheduled at any time before **deadline**. Each accepted request must be assigned an interval of time of length **ti**, and different requests must be assigned **non-overlapping intervals**.

- let us denote this interval by **[s(i), f(i)], with f(i) = s(i) + ti.** Unlike the previous problem, then, the algorithm must actually determine a start time for each interval.
- We say that a request i is late if it misses the deadline, that is, if **f(i) > di**. The lateness of such a request **i** is defined to be **li = f(i) − di.** We will say that **li = 0** if request i is not late.
- The goal in our new optimization problem will be to schedule all requests, using non-overlapping intervals, so as to minimize the maximum lateness, **L = maxi li**.

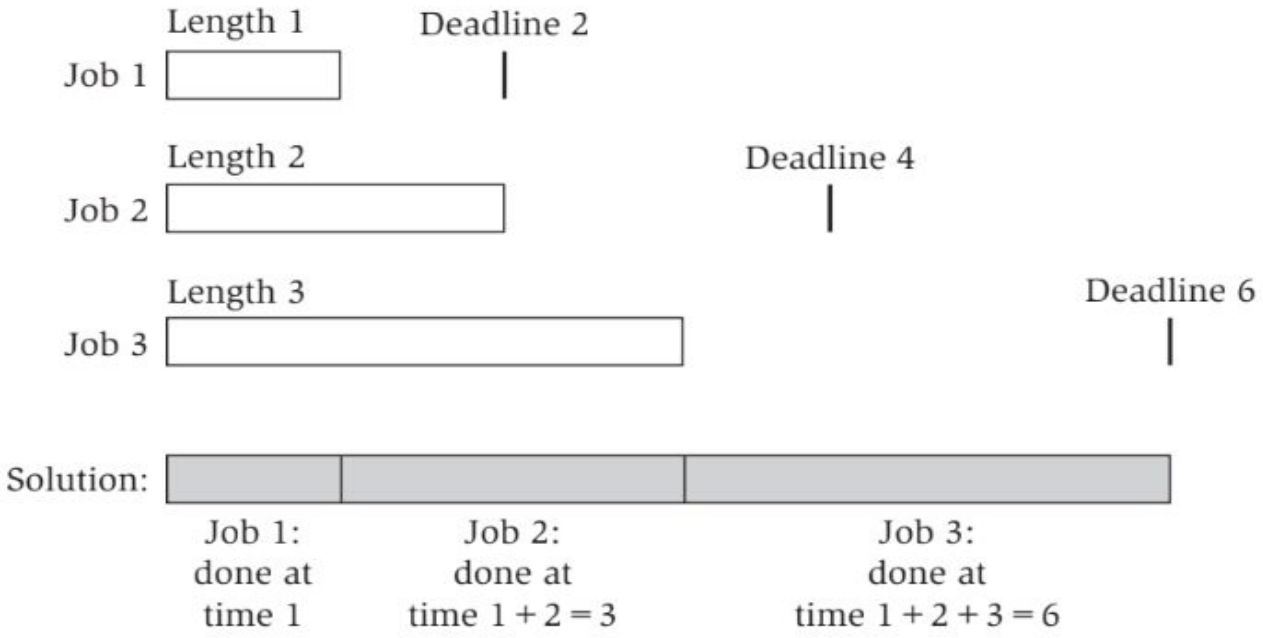


Figure 4.5 A sample instance of scheduling to minimize lateness.

More Info:
https://www.youtube.com/watch?v=80PrGzqW4u8&list=PLMLFEbzd52KPTn1_ZCjYsNpYOXXzmjfwM&index=6

Problem Definition

- Consider a situation where we have single resource and n requests to use this resource.
- Instead of start time and finish time , each request i has a deadline d_i

  t_i -> time taken by job i to complete its task

Constraint : We can schedule the jobs any time before the deadline and scheduling of jobs must be done in such a way that different jobs/requests are assigned non overlapping intervals.

s(j) f(j) given t(j),

$f(j) = s(j) + t(j) > d(j)$

$l(j) = f(j) - d(j)$

request is late

10 s(j)
1 t(j)
11 f(j)

② $f(j) \leq d(j)$ $l(j) = 0$

request is not late

Goal:

- Goal of our optimization problem is to schedule all jobs/requests using non overlapping intervals so as to minimize maximum lateness.

$L = \max_j l_j$

Jobs	t(j)	d(j)	f(j)	l(j)
A	1	2	1	0
B	2	4	3	0
C	3	6	6	0

maximum lateness = 0

Jobs	t(j)	d(j)	f(j)	l(j)
A	1	2	6	6-2=4
B	2	4	5	1
C	3	6	3	0

maximum lateness = 4

Greedy strategy 1

Choose jobs in increasing order of length i.e. t_j

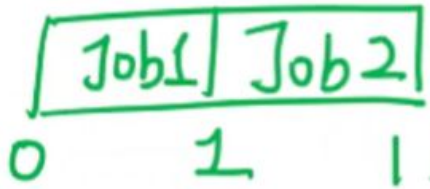
optimal strategy

deadlines

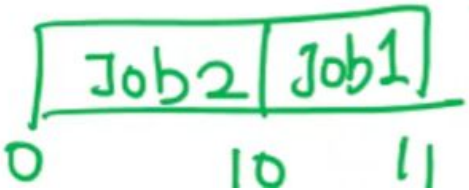
$t_1 = 1$ $d_1 = 100$ $f_1 = 1$ $l_1 = 0$

$t_2 = 10$ $d_2 = 10$ $f_2 = 11$ $l_2 = 1$

maximum lateness = 1



max. lateness = 0



$f_1 = 11$ $l_1 = 0$

$f_2 = 10$ $l_2 = 0$

Greedy strategy 2

Pick the job with the smallest slack time i.e. $d_j - t_j$ is small.

Slack time

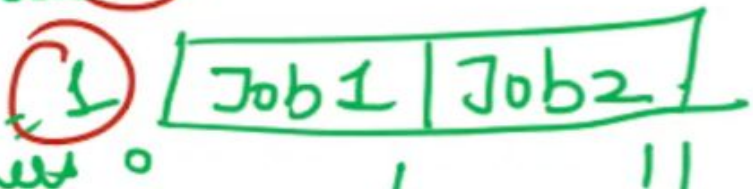
$t_1 = 1 \quad d_1 = 3 \quad S.t_1 = 2$

$t_2 = 10 \quad d_2 = 10 \quad S.t_2 = 0$

$L_1 = 11 - 3 = 8$

$L_2 = 10 \quad 10 = 0$

max lateness = 8



$L_1 = 0$

$L_2 = 1$

Greedy strategy 3

Choose the job with the earliest deadline d_j first.

Algorithm

- order the jobs on the basis of their deadline

$d_1 \leq d_2 \leq \cdots d_n$

S → overall starting time

Job_1 $S \rightarrow S_1$

$S_1 + t_1 = f_1$

Job_2 $S_2 + t_2 \leq f_2$.

:

So on

Schedule each job as soon as the previous job ends

Return the set of scheduled jobs $[(s_1, f_1), (s_2, f_2) \cdots (s_n, f_n)]$

Algorithm

Order the jobs in order of their deadlines

Assume for simplicity of notation that $d_1 \leq \ldots \leq d_n$

Initially, $f = s$

Consider the jobs $i = 1, \ldots, n$ in this order

  Assign job i to the time interval from $s(i) = f$ to $f(i) = f + t_i$

  Let $f = f + t_i$

End

Return the set of scheduled intervals $[s(i), f(i)]$ for $i = 1, \ldots, n$

Analyzing the Algorithm

we first observe that the schedule it produces has no “gaps”—times when the machine is not working yet there are jobs left. The time that passes during a gap will be called idle time: there is work to be done, yet for some reason the machine is sitting idle.

To reason about the optimality of the algorithm

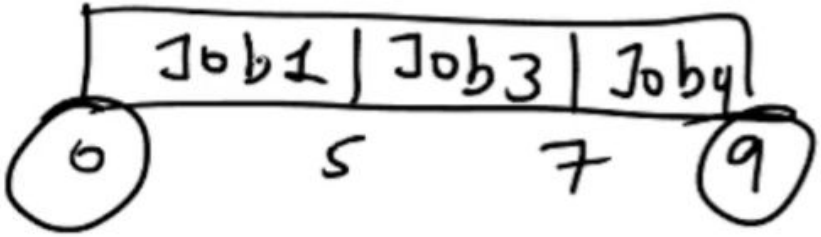
- There is an optimal schedule with **no idle time.**
- All schedules with no inversions and no idle time have the same maximum lateness.
- There is an optimal schedule that has no inversions and no idle time.
- The schedule A produced by the greedy algorithm has optimal maximum lateness L.

Since we are scheduling jobs one after the other , it is clear that schedule has no gaps or idle time

$d_1 < d_2 < \ldots d_n$

minimize maximum lateness

$[(s_1, f_1), (s_2, f_2), \ldots (s_n, f_n)]$



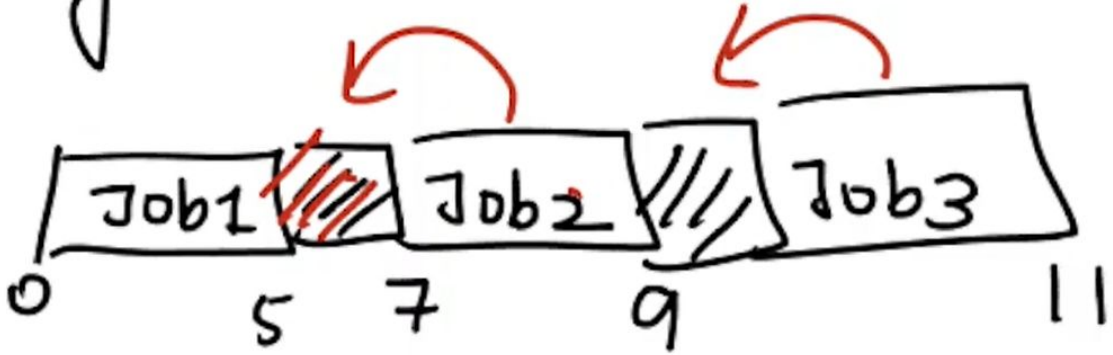
idle time

idle time

our schedule has no gaps

A
greedy

O
optimal



starting time &
reduce maximum lateness

Claim : There is an optimal schedule O with no idle time

Proof: It is very clear that we can shift the jobs earlier because there is no constraint on when jobs should start. The only constraint is on when jobs should finish , so by shifting the jobs earlier to remove idle time we can only reduce lateness.

Hence , optimal schedule has no idle time.

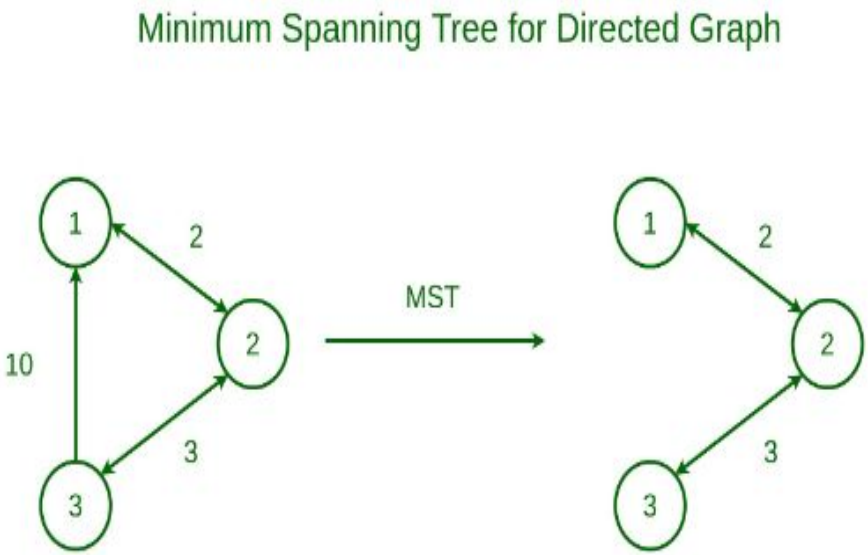
More Info:
https://www.youtube.com/watch?v=b2L3T1OmeXo&list=PLMLFEbzd52KPTn1_ZCjYsNpYOXXzmjfwM&index=7

What is Minimum Spanning Tree (MST)

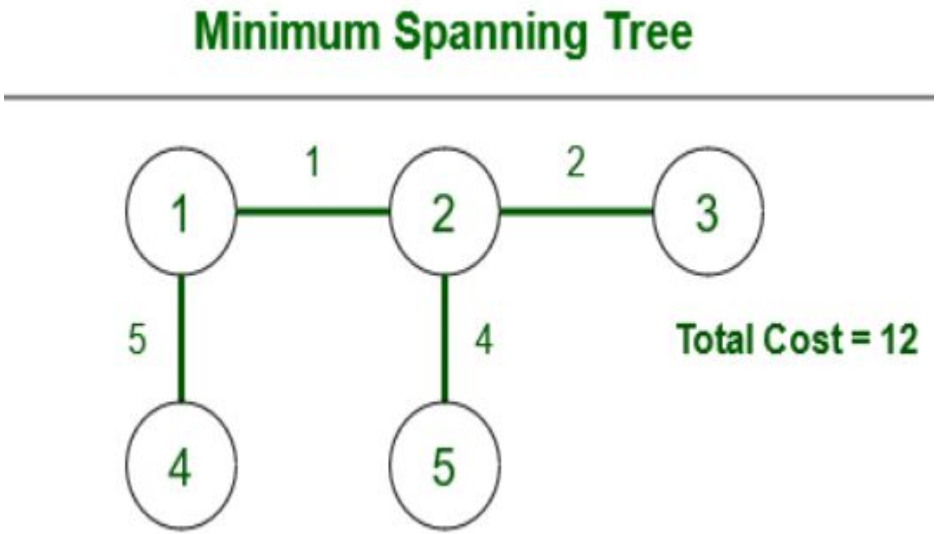
- A **spanning tree** is defined as a tree-like subgraph of a connected, undirected graph that includes all the vertices of the graph.
- A minimum spanning tree **(MST)** is defined as a spanning tree that has the minimum weight among all the possible spanning trees all nodes in a spanning tree are reachable from each other.

Properties of a Spanning Tree

- There is a fixed number of edges in spanning tree which is equal to one less than total number of vertices (**E** = **V-1**).
- The spanning tree should not be **disconnected**.

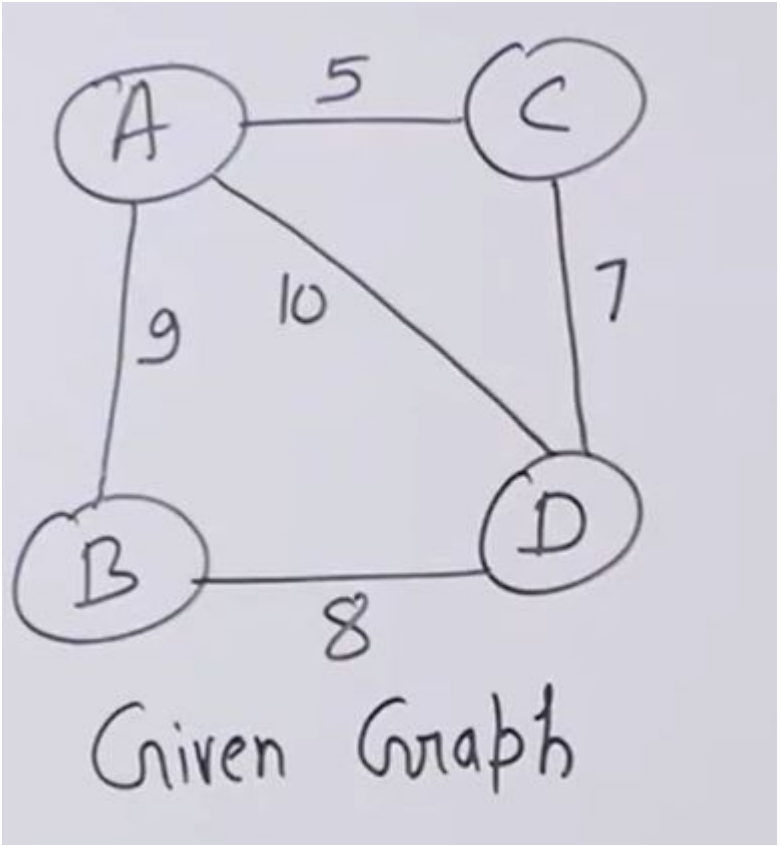


- The spanning tree should be **acyclic**.
- The **total cost** of the spanning tree is defined as sum of the edge weights of all edges of the spanning tree.



Algorithms to find Minimum Spanning Tree

1. Prim's MST Algorithm
2. Kruskal's MST Algorithm



Prim's Algorithm

DEFINITION A spanning tree of an **undirected connected graph** is its **connected acyclic subgraph** that contains all the vertices of the graph. If such a graph has **weights** assigned to its **edges,** a minimum spanning tree is its spanning tree of the smallest weight. The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

Working of the Prim's Algorithm

Step 1: Determine an arbitrary vertex as the starting vertex of MST. We pick 0 in the below diagram.

Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST.

Step 3: Find edges connecting any tree vertex with fringe vertices.

Step 4: Find the minimum among these edges.

Step 5: Add the chosen edge to the MST. Since we consider only the edges that connect fringe vertices with the rest, we never get a cycle.

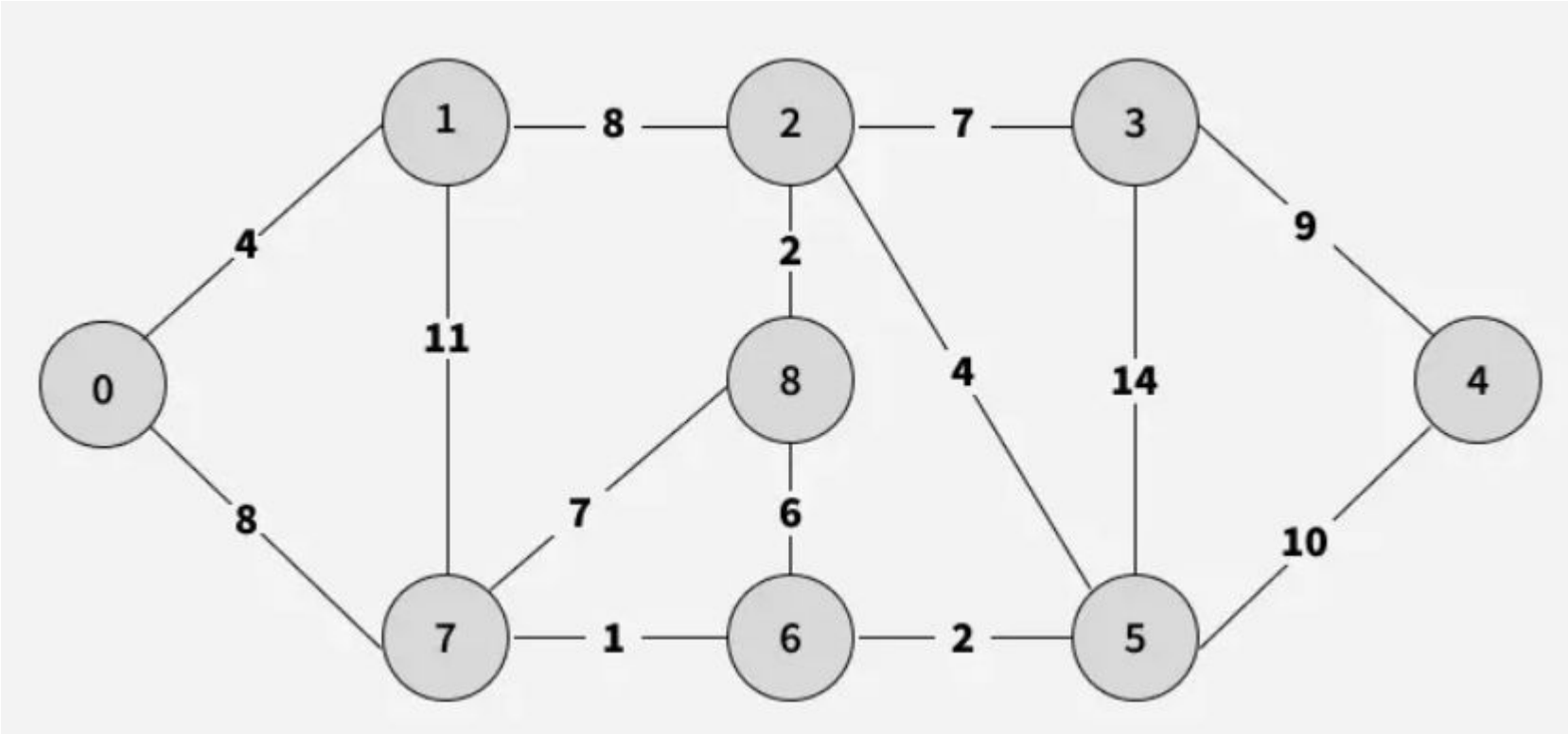
Step 6: Return the MST and exit

Pseudocode

ALGORITHM *Prim(G)*

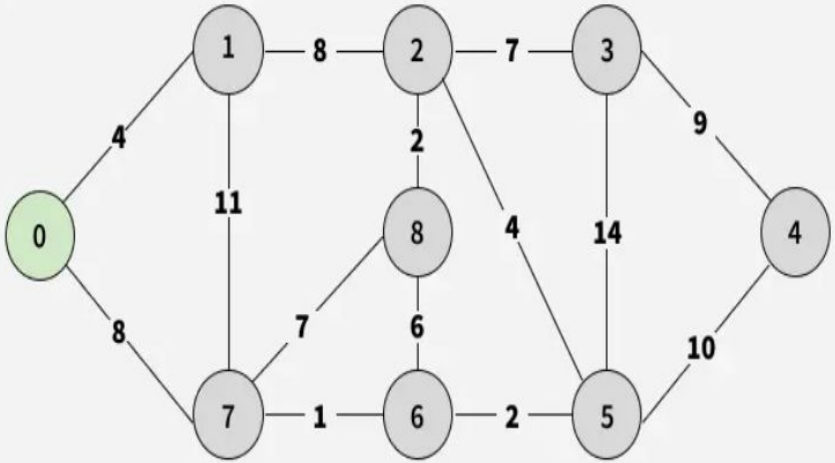
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Problem



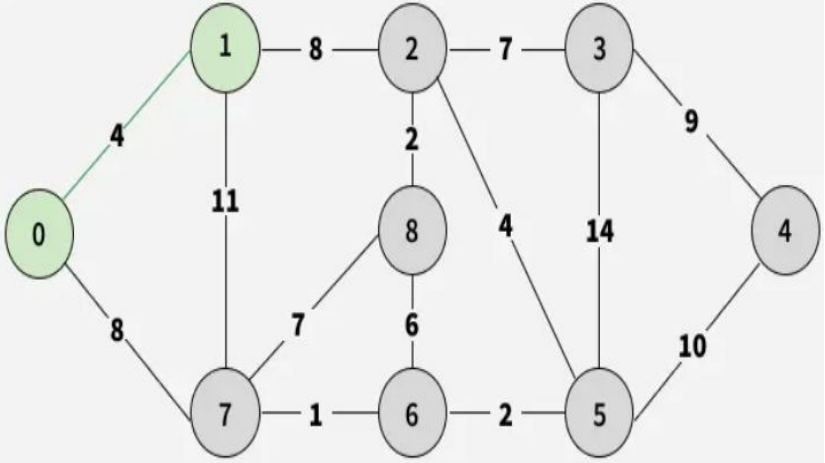
01 Step

Start with edges {0,1} and {0,7}; pick the minimum {0,1} and add vertex 1.



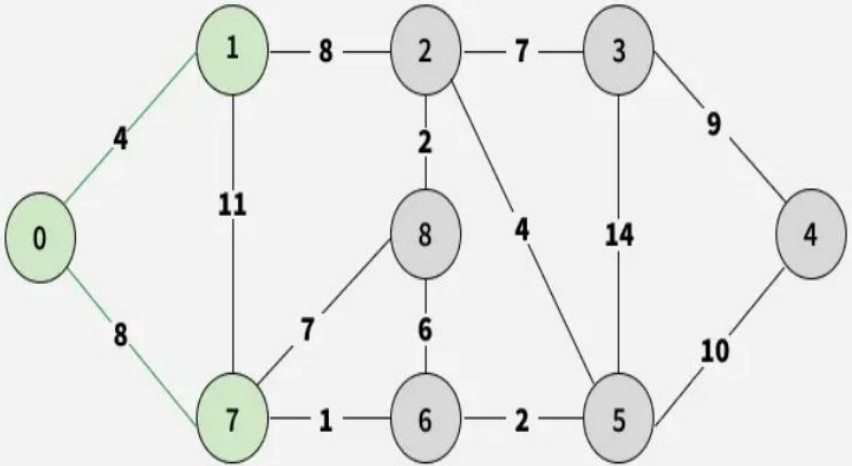
02 Step

Choose between {0,7} and {1,2}; pick {0,7} (or {1,2}), adding vertex 7.



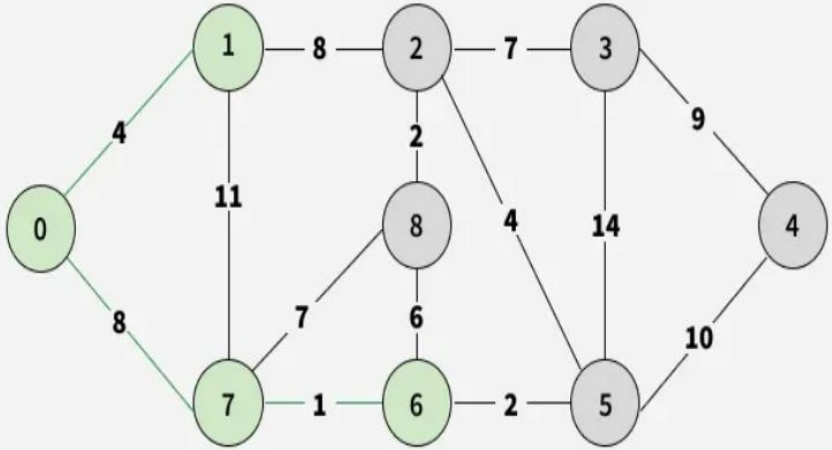
03 Step

Select {7,6} as it has the least weight (1), adding vertex 6.

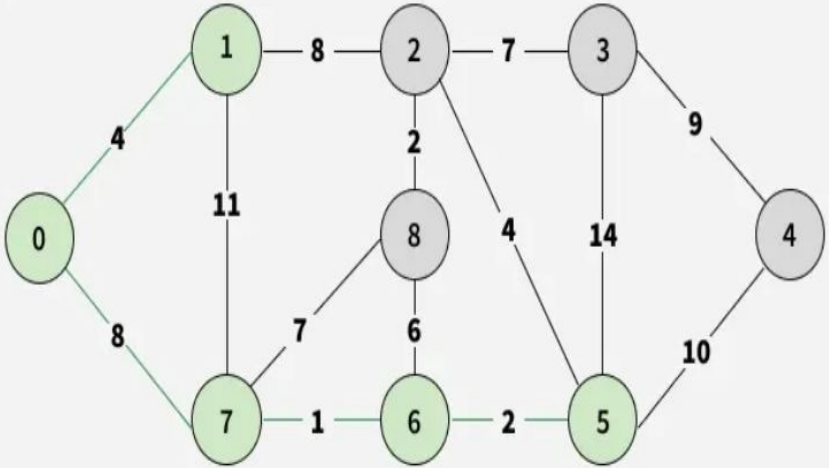


04 Step

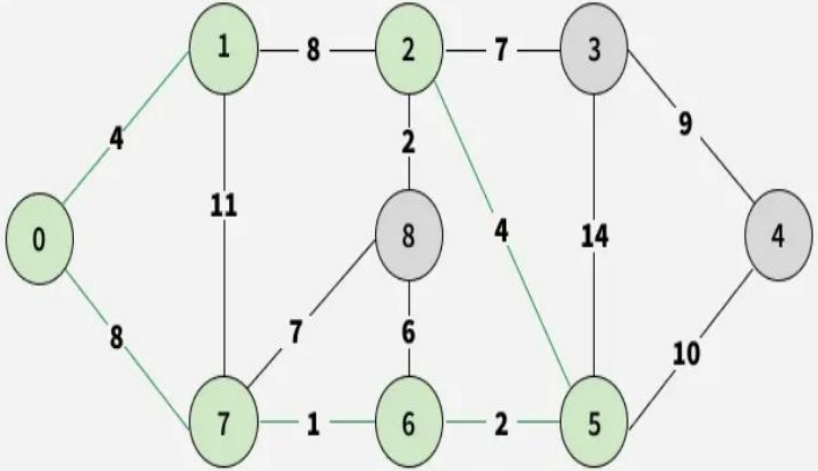
Among {7,8}, {6,8}, and {6,5}, pick {6,5} (weight 2) and add vertex 5.



05 Step | Select {5,2} (weight 4), adding vertex 2.

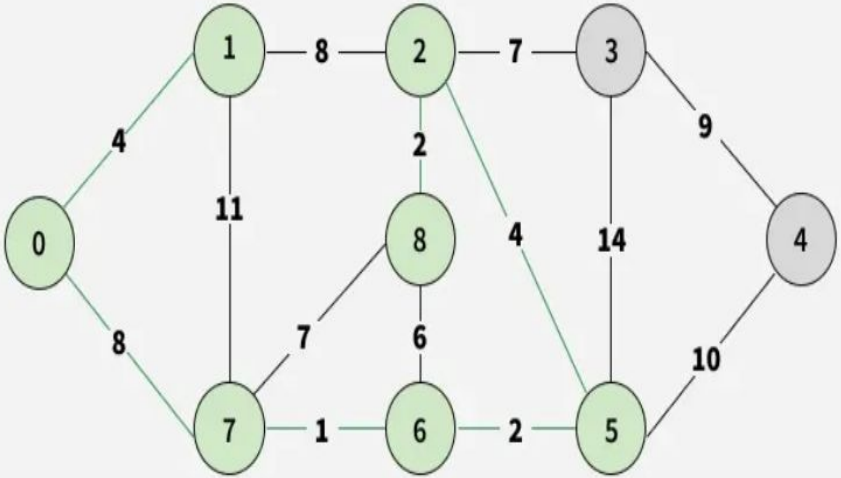


06 Step | Choose {2,8} (weight 2), adding vertex 8.



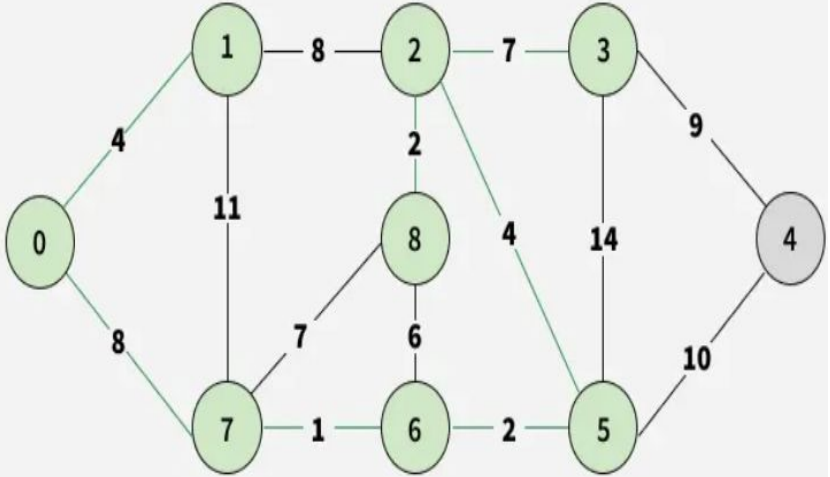
07 Step

Pick {2,3} as the smallest edge, adding vertex 3.



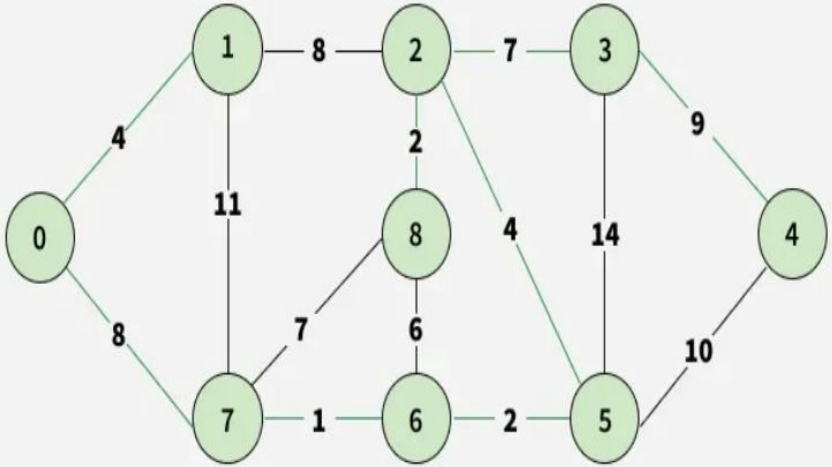
08 Step

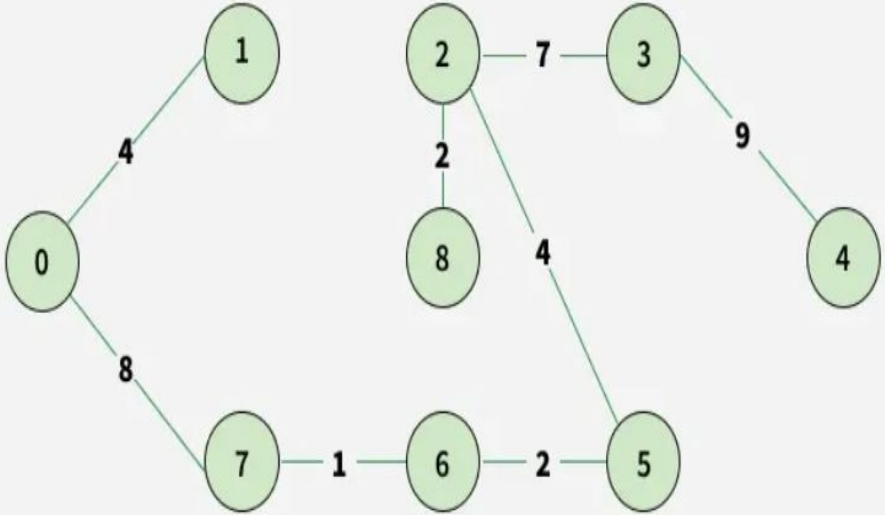
Finally, select {3,4}, adding vertex 4.



09 Step

All vertices are included, so the MST is complete.





Kruskal's Algorithm

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph **G = (V,E)** as an **acyclic subgraph** with **|V | − 1** edges for which the sum of the edge weights is the smallest. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

(1) Sort all edges in increasing order.

(2) Initialize: MST = { }, res = 0

(3) Do following for every edge 'e' while MST size does not become V-1.

(a) If adding e to MST does not cause a cycle

MST = MST ∪ {e}

res = res + e.weight.

(4) Return res.

Working of the Kruskal's Algorithm

STEP 1: Sort all edges in a **non-decreasing order** of weight.

STEP 2: Pick the **smallest edge.** Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this **edge**. Else, discard it.

STEP 3: Repeat step 2 until there are (V-1) edges in the spanning tree

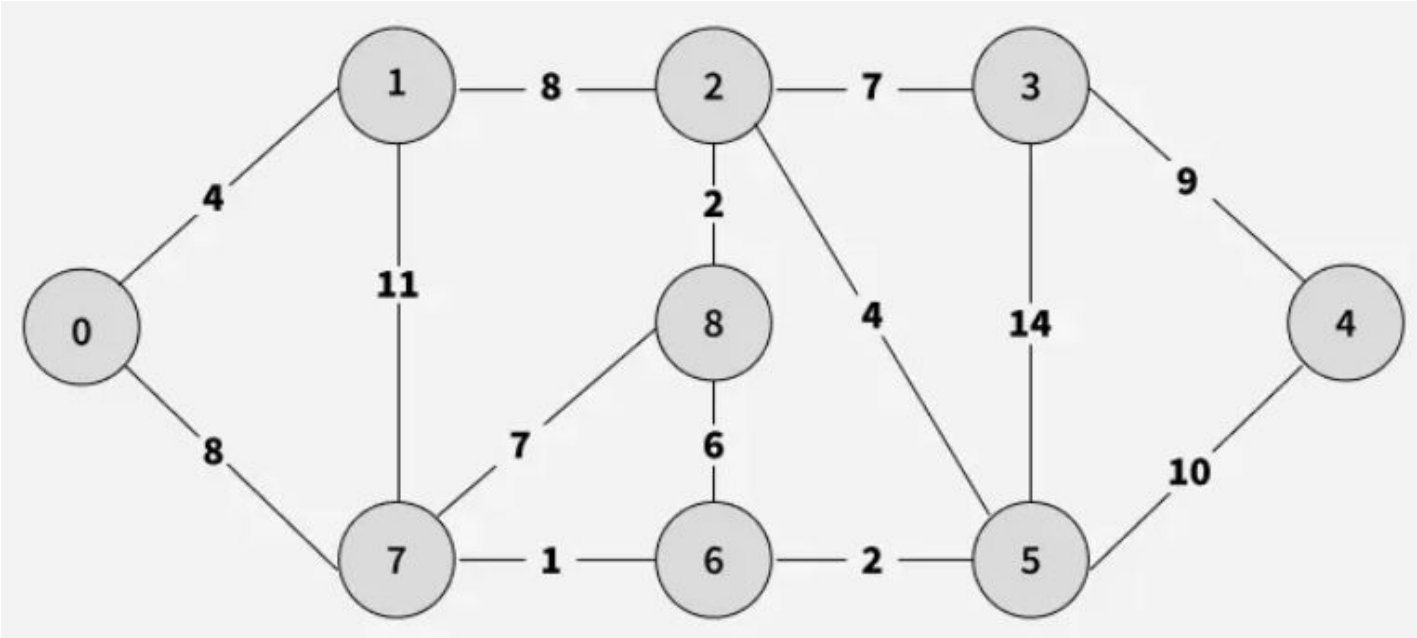
Pseudocode

ALGORITHM *Kruskal*(G)

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

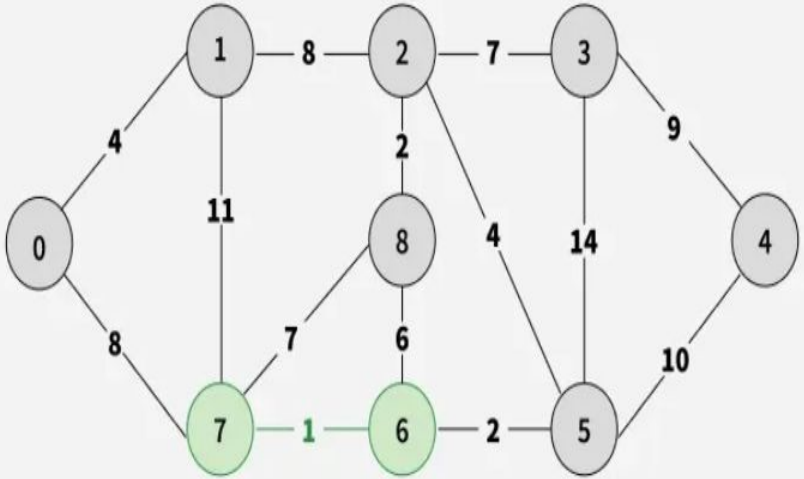
Problem

The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.



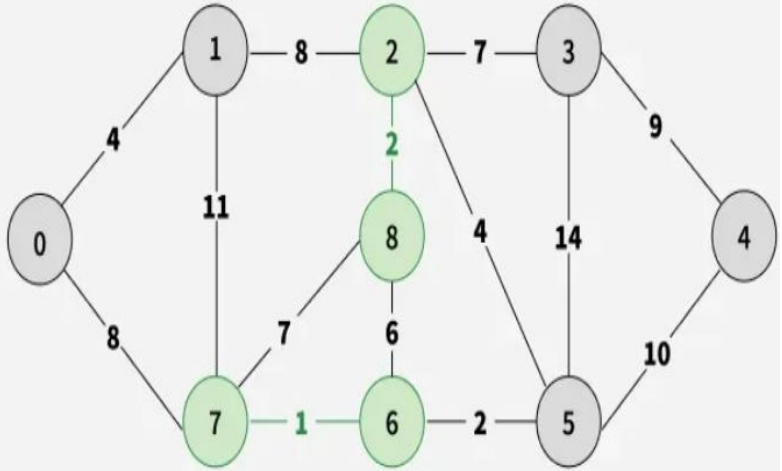
02 Step

Pick all sorted edges one by one.
Pick edge 7-6. No cycle is formed, include it.

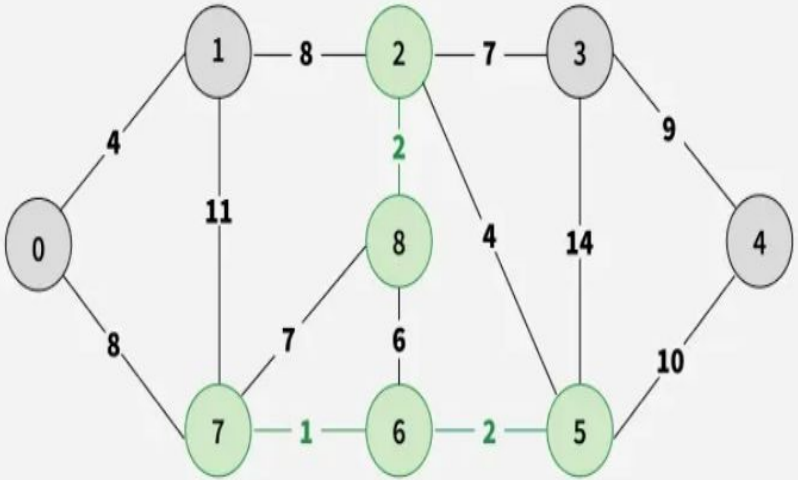


03 Step

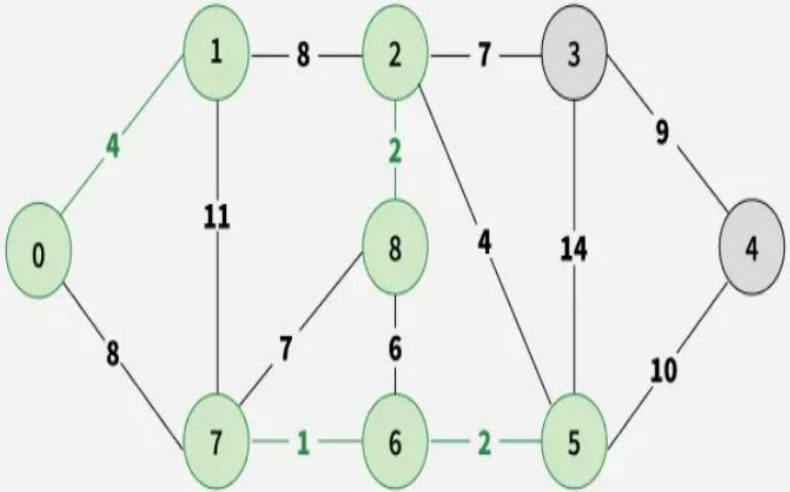
Pick edge 8-2. No cycle is formed, include it.



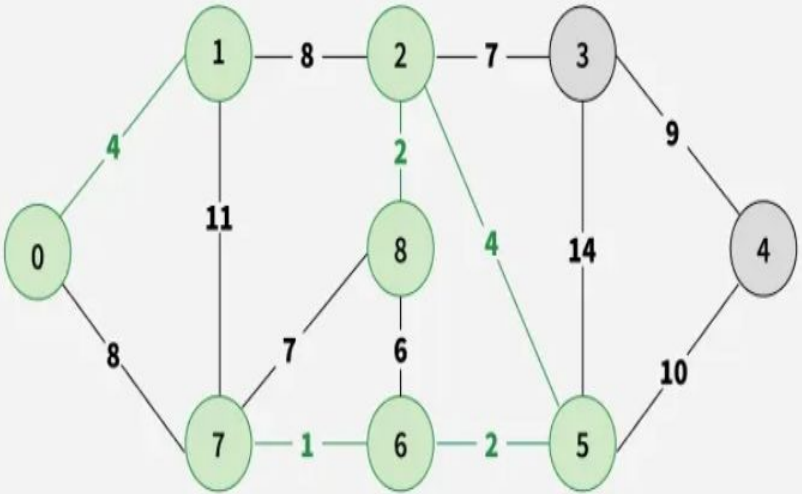
04 Step | Pick edge 6-5. No cycle is formed, include it.



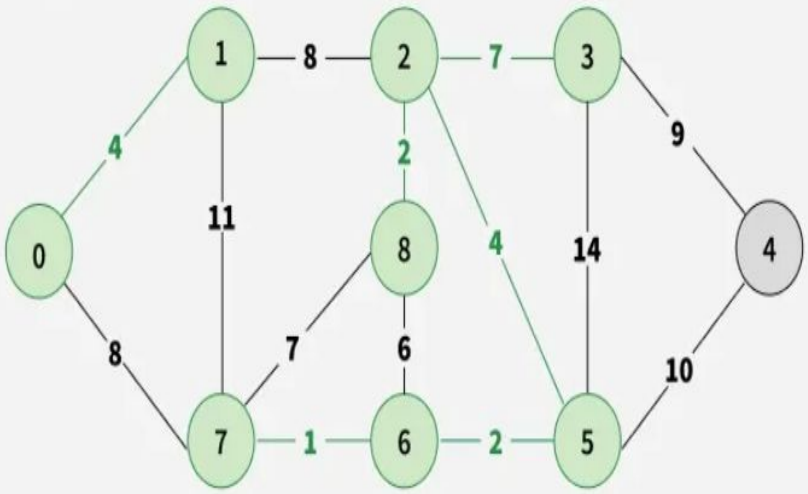
05 Step | Pick edge 0-1. No cycle is formed, include it.



06 Step | Pick edge 2-5. No cycle is formed, include it.

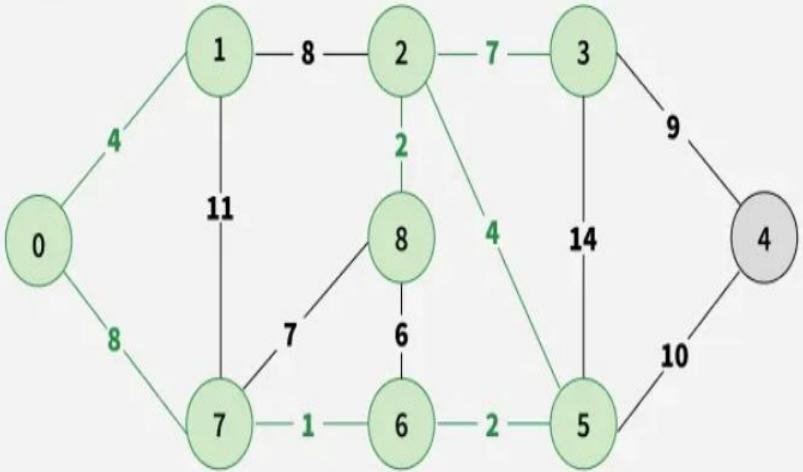


07 Step | Pick edge 8-6. Since including this edge results in the cycle, discard it.
Pick edge 2-3: No cycle is formed, include it.



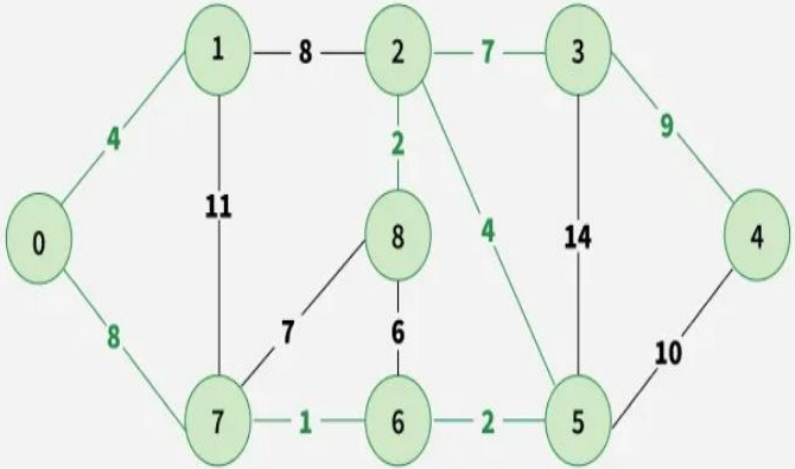
08 Step

Pick edge 7-8. Since including this edge results in the cycle, discard it.
Pick edge 0-7. No cycle is formed, include it.

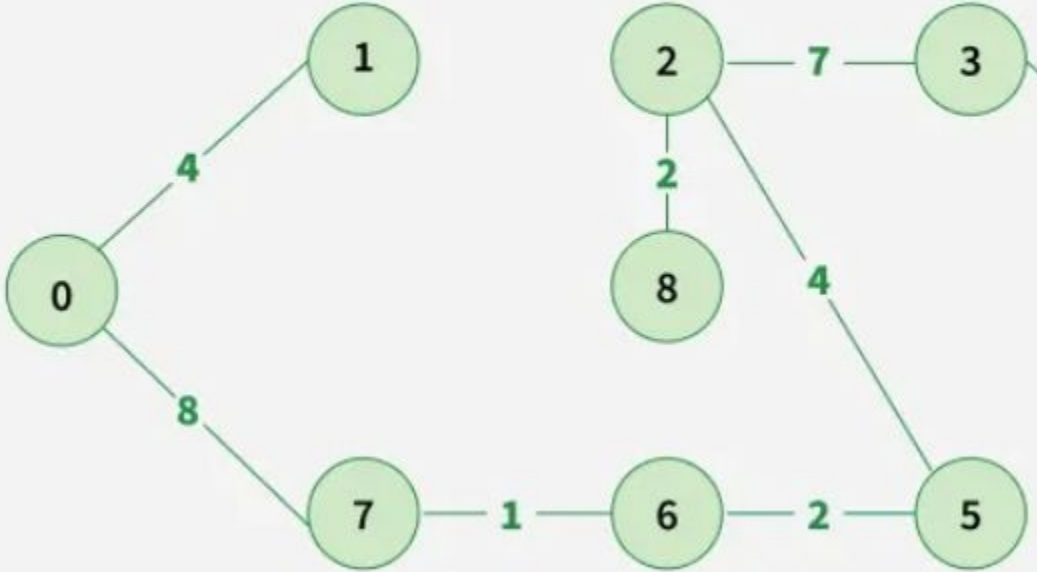


09 Step

Pick edge 1-2. Since including this edge results in the cycle, discard it.
Pick edge 3-4. No cycle is formed, include it.



10 Step | **Final obtained graph minimum spanning tree.**



Since the number of edges included in the MST equals to (V – 1),
So the algorithm stops here.
Total cost of MST = (37)

Dijkstra’s Algorithm

- The best-known algorithm for **single-source shortest-paths** problem, called Dijkstra’s algorithm.
- Find shortest paths to all its other vertices. It is important to stress that we are not interested here in a single shortest path that starts at the source and visits all the other vertices.
- This algorithm is applicable to **undirected and directed graphs** with **nonnegative weights** only.

- The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source.
- It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

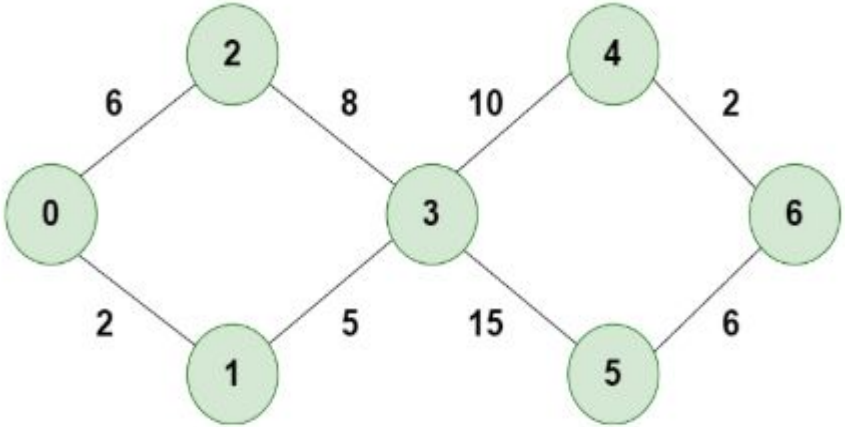
Algorithm for Dijkstra's Algorithm

1. Mark **source node** with current distance of **0** and rest with **infinity.**
2. Set the non-visited node with the smallest current distance as the current node.
3. For each neighbor, **N** of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes are unvisited.

ALGORITHM *Dijkstra*(G, s)

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
//        and its vertex s
//Output: The length d_v of a shortest path from s to v
//          and its penultimate vertex p_v for every vertex v in V
Initialize(Q)  //initialize priority queue to empty
for every vertex v in V
    d_v ← ∞;   p_v ← null
    Insert(Q, v, d_v)  //initialize vertex priority in the priority queue
d_s ← 0;  Decrease(Q, s, d_s)  //update priority of s with d_s
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)  //delete the minimum priority element
    V_T ← V_T ∪ {u*}
    for every vertex u in V − V_T that is adjacent to u* do
        if d_u* + w(u*, u) < d_u
            d_u ← d_u* + w(u*, u);   p_u ← u*
            Decrease(Q, u, d_u)
```

Example

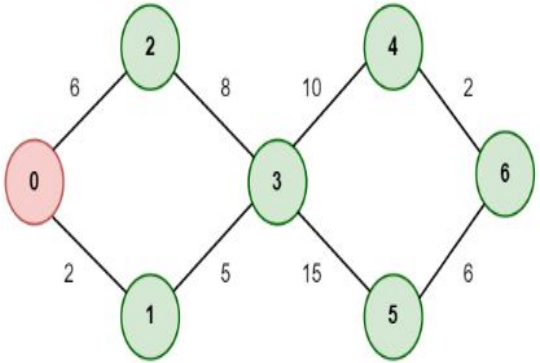


Initially we have:

- The Distance from the source node to itself is 0. In this example the source node is 0.
- The distance from the source node to all other node is unknown so we mark all of them as infinity.

STEP 1

Start from Node 0 and mark Node 0 as Visited and check for adjacent nodes



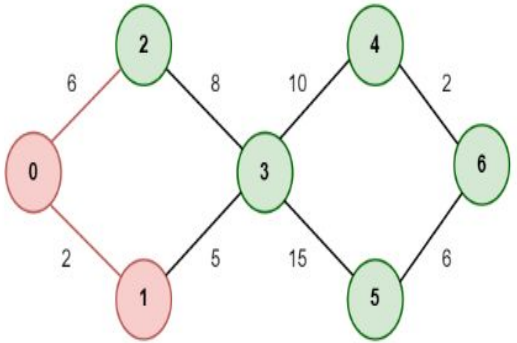
Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✓
1: ∞
2: ∞
3: ∞
4: ∞
5: ∞
6: ∞

Dijkstra's Algorithm

STEP 2

Mark Node 1 as Visited and add the Distance

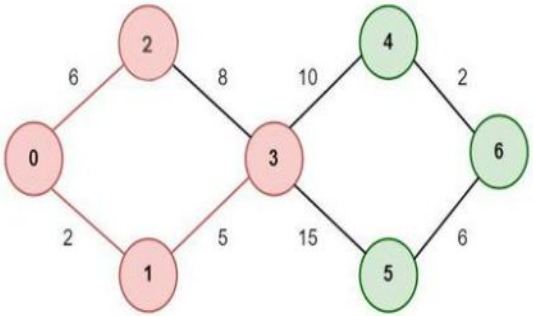


Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✓
1: 2 ✓
2: ∞
3: ∞
4: ∞
5: ∞
6: ∞

Dijkstra's Algorithm

STEP 3 Mark Node 3 as Visited after considering the Optimal path and add the Distance

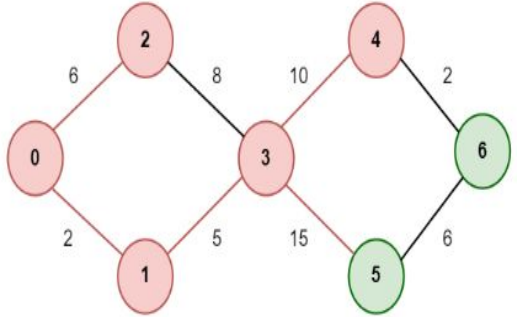


Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✓
1: 2 ✓
2: 6 ✓
3: 7 ✓
4: ∞
5: ∞
6: ∞

Dijkstra's Algorithm

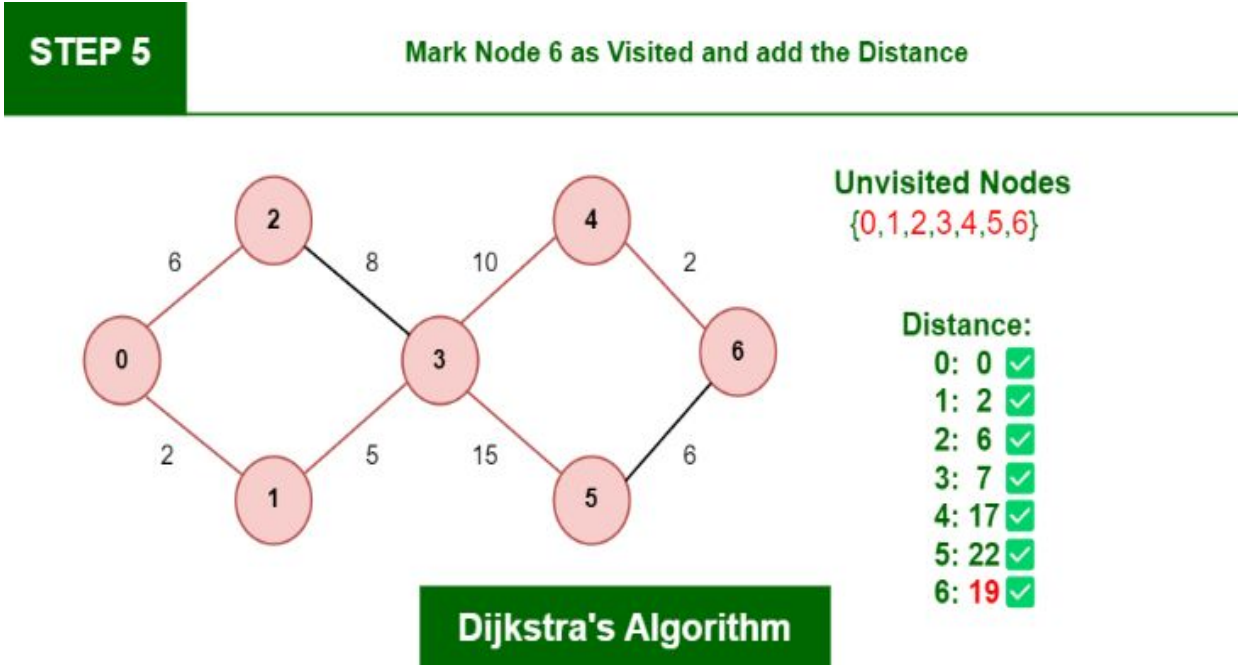
STEP 4 Mark Node 4 as Visited after considering the Optimal path and add the Distance



Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✓
1: 2 ✓
2: 6 ✓
3: 7 ✓
4: 17 ✓
5: ∞
6: ∞

Dijkstra's Algorithm



Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 -> Node 6 = 2 + 5 + 10 + 2 = 19

So, the Shortest Distance from the Source Vertex is 19 which is optimal one

Huffman Trees and Codes

- 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.
- we can use a **fixed-length encoding** that assigns to each symbol a bit string of the same length. This is exactly what the standard **ASCII** code does.

Huffman coding

→ data compression technique [or] Variable length coding Algo

Ex: Computer (data compression)

A B B C D B C C D A A B B E E E B E A B = 20 char

A – 65 – 01000001 = 8 bit

ASCII

20 × 8 = 160 bit

fixed length coding

Huffman's algorithm

Step 1: Initialize **n** one-node trees and label them with the symbols of the **alphabet** given. Record the frequency of each symbol in its tree's root to indicate the **tree's weight.**

Step 2: Repeat the following operation until a single tree is obtained. Find two trees with the **smallest weight.** Make them the left and right subtree of a new tree and record the sum of their weights in the **root of the new tree** as its weight.

Let us understand the algorithm with an example:

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

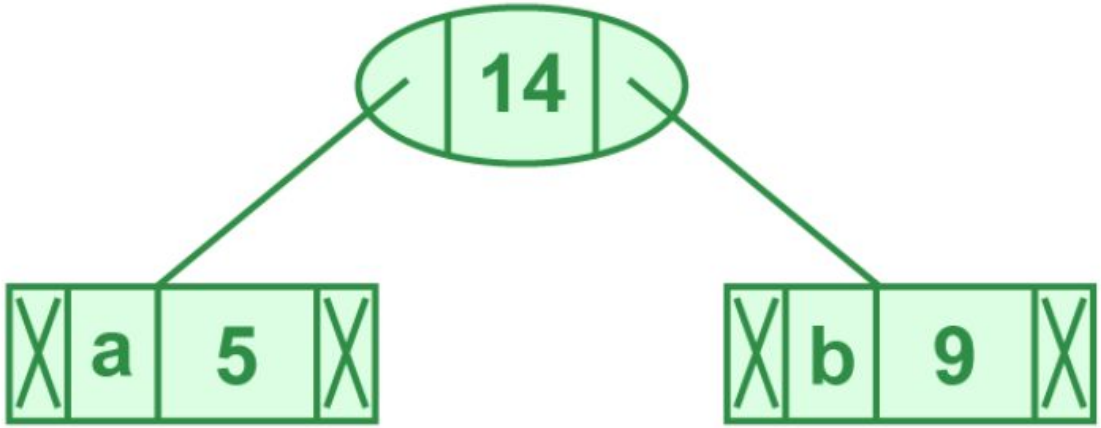


Illustration of step 2

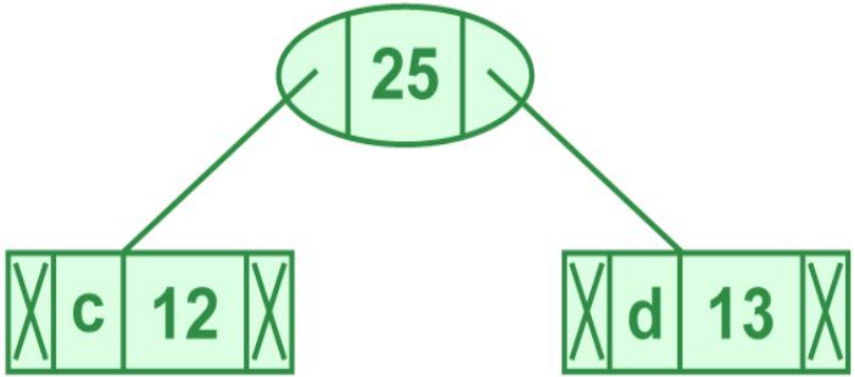


Illustration of step 3

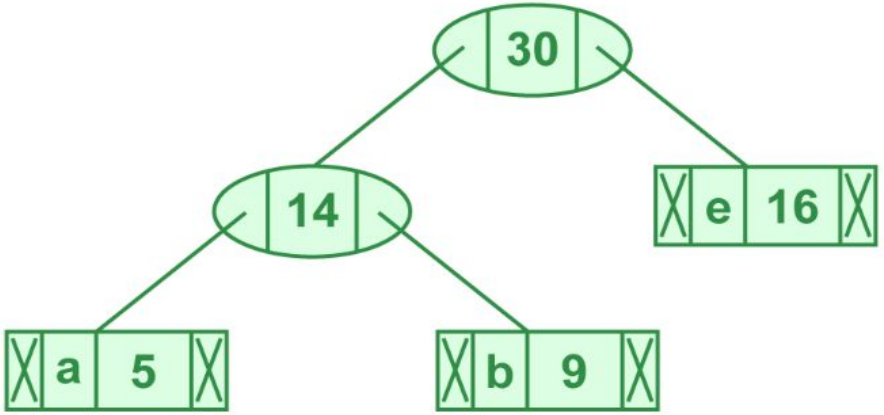


Illustration of step 4

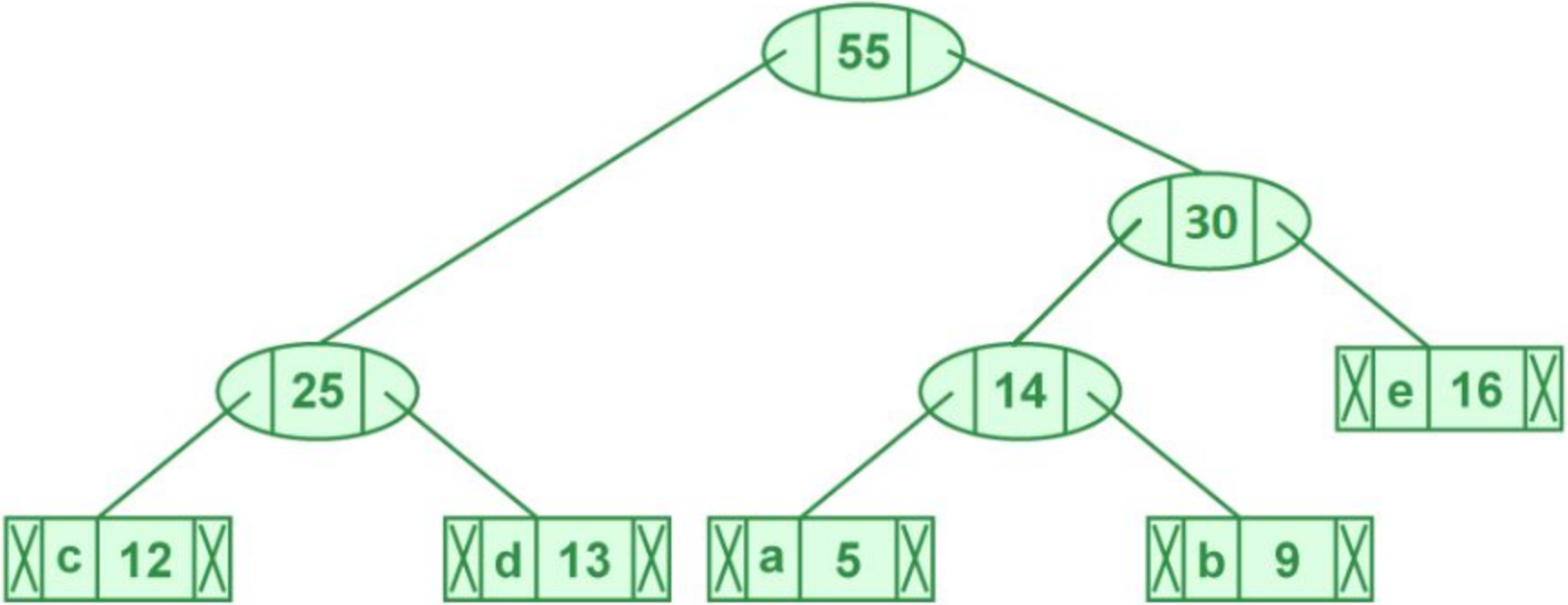


Illustration of step 5

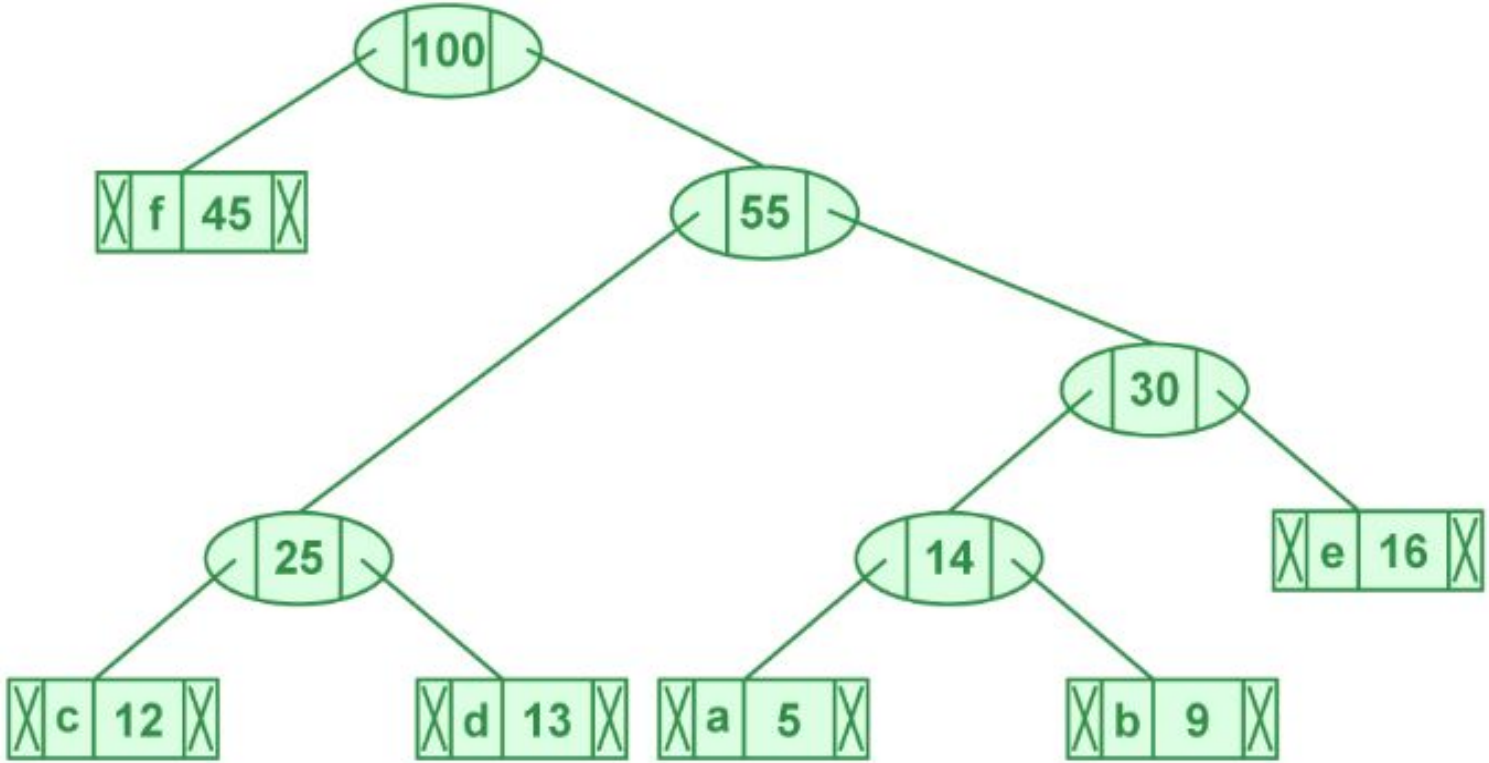
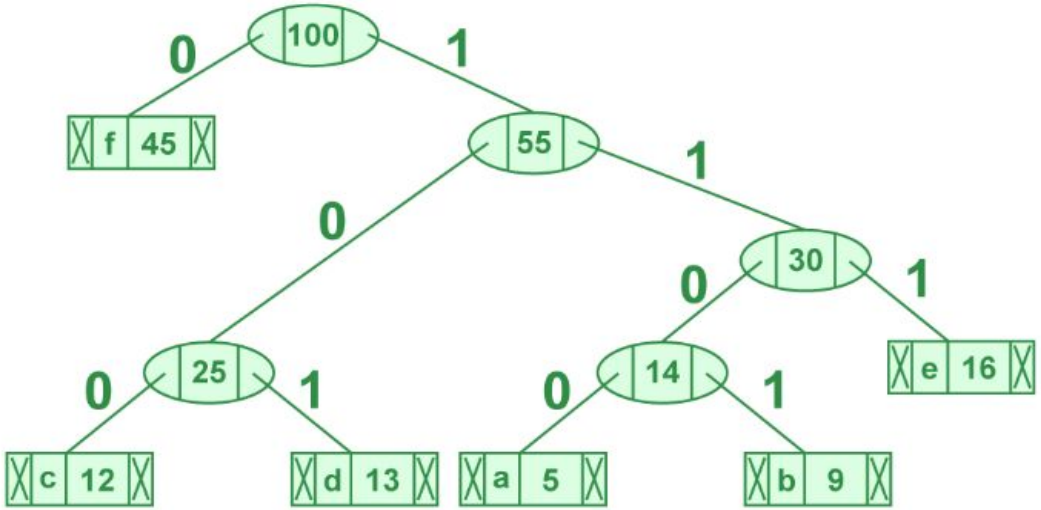


Illustration of step 6



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

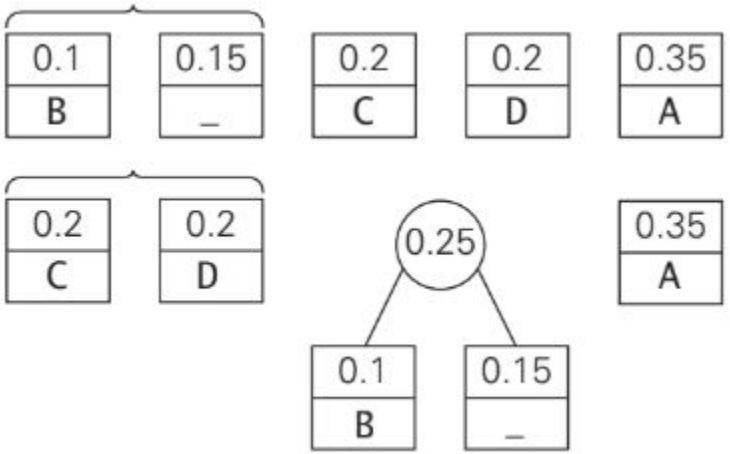
Steps to print codes from Huffman Tree:

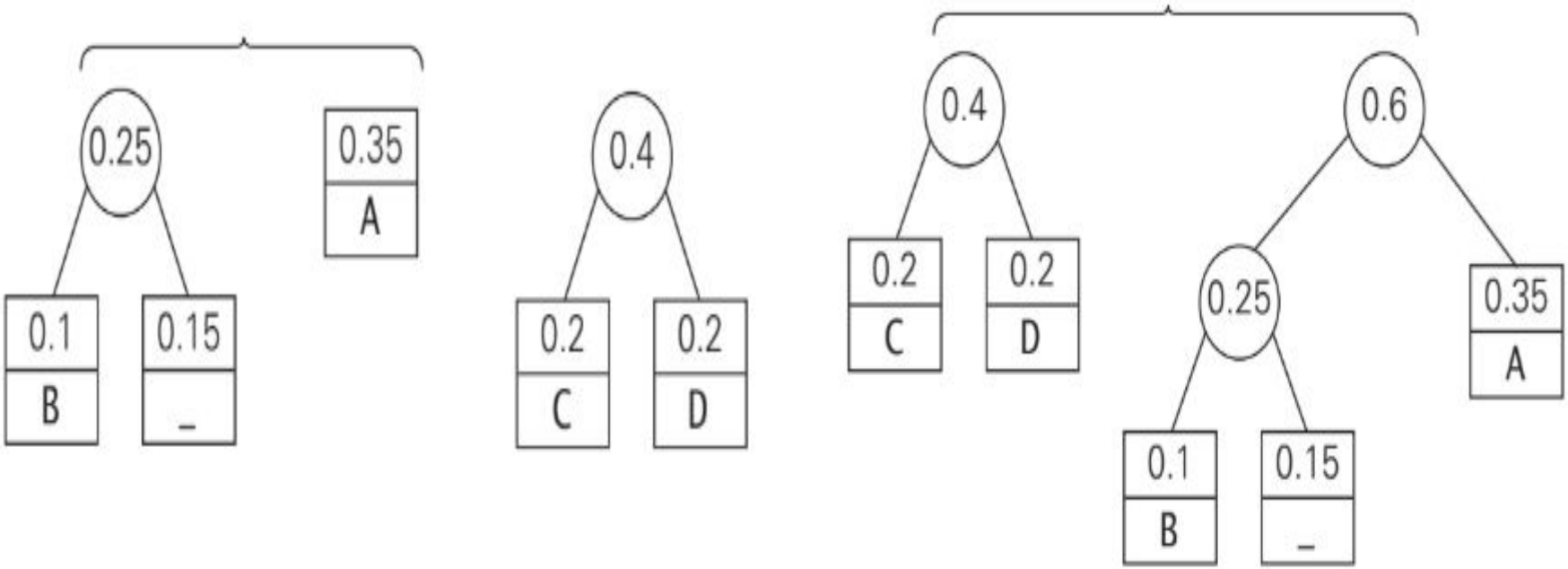
Traverse the tree formed starting from the root. While moving to the left child, **write 0** to the array. While moving to the right child, **write 1** to the array. Print the array when a leaf node is encountered.

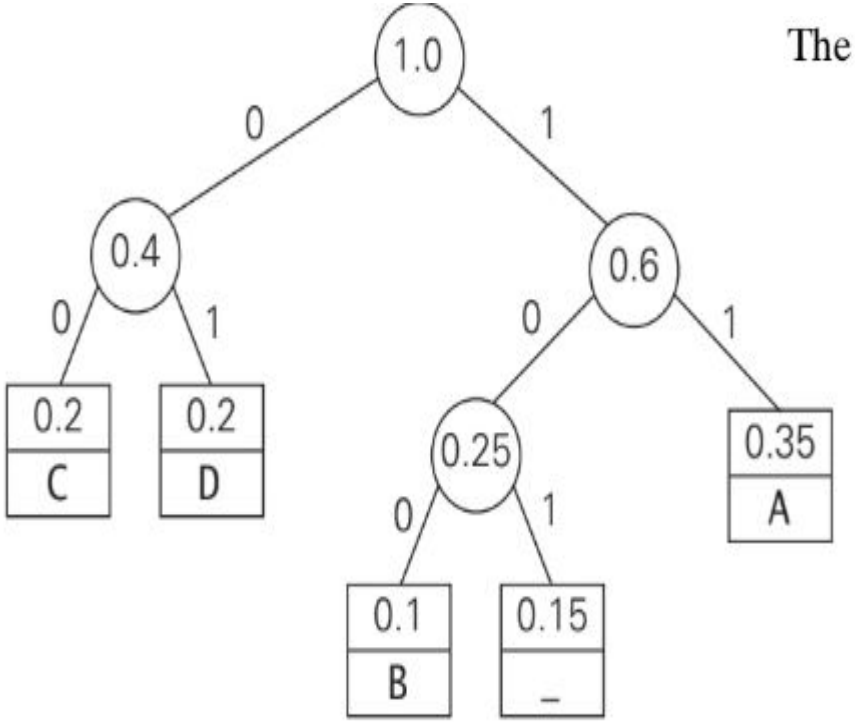
EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is shown in Figure 9.12.




0.25
0.35
A
0.1
B
0.15
_
0.4
0.2
C
0.2
D
0.4
0.6
0.2
C
0.2
D
0.25
0.35
A
0.1
B
0.15
_



The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, DAD is encoded as 011101, and 100110111011101 is decoded as BAD_AD.

With the occurrence frequencies given and the codeword lengths obtained, the average number of bits per symbol in this code is

$$2 \cdot 0.35 + 3 \cdot 0.1 + 2 \cdot 0.2 + 2 \cdot 0.2 + 3 \cdot 0.15 = 2.25.$$