Q1) Consider the following array [1, 3, 5, 8, 9, 2, 6, 7, 6]
What is minimum number of jump required to reach the end of the array?

---

solwe have an array of element [1,3,5,8,9,2,6,7,6]
here each element represent maximum no. of jumps, let suppose we are on index 0 having value 1 so it can jump up to next 1 index { here next one index is only 1 which have value 3}. **1JUMP.**
now my pointer is on 3 ,i can jump any one from next 3 index which can give the high value {next three index will be 2,3,4} having value {5,8,9} so i choose 9, now my pointer is on index 4. **1 JUMP.**

AT pointer 4 value 9 i can jump up to next 9 index, but before to reach next 9 index, i complete traverse my array and reach at the end. so. after CONCLUDING this jump also. i can reach end to the array. 1 **JUMP**

SO TOTAL 3 JUMP WE RACH AT THE END OF ARRAY. { REMEMBER THESE ARE THE MIN JUMPS NOW YOU CAN ALSON FIND MAX JUMPS,CHOOSING THE MIN ELEMENT FROM THE ARRAY},

Q2)Solve the following instance of 0/1 knapsack problem by applying Dynamic programming n = 3 w = (3, 5, 7) p(3, 7, 12), M = 4?
Consider the instance of 0/1 knapsack problem n = 3, m = 20, p = (25, 24, 15), w = (18, 15, 10) using dynamic programming.
Q3) Find the largest common subsequence for the following string using Dynamic Programming ?
X = [A, B, C, D, B, A, C, D, F]

Draw Binary Tree from given traversal
Inorder : 4,2,5,1,6,7,3,8
Post order : 4,5,2,6,7,8,3,1

Construct a binary tree from inorder and postorder traversals

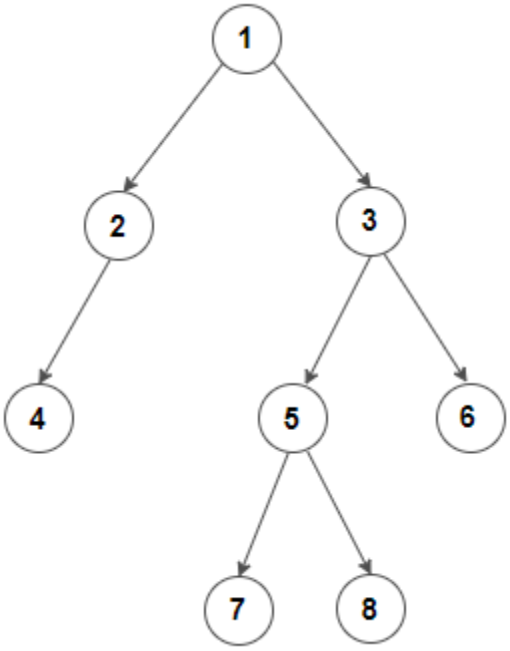
Write an efficient algorithm to construct a binary tree from the given **inorder** and **postorder** traversals.

For example,

Input:

```
Inorder    Traversal    :    {    4,    2,    1,    7,    5,    8,    3,    6    }
Postorder    Traversal    :    {    4,    2,    7,    8,    5,    6,    3,    1    }
```

Output: Below binary tree



Practice this problem

The idea is to start with the root node, which would be the last item in the postorder sequence, and find the boundary of its left and right subtree in the inorder sequence. To find the boundary, search for the index of the root node in the inorder sequence. All keys before the root node in the inorder sequence become part of the left subtree, and all keys after the root node become part of the right subtree. Repeat this recursively for all nodes in the tree and construct the tree in the process.

To illustrate, consider the following inorder and postorder sequence:

```
Inorder    :    {    4,    2,    1,    7,    5,    8,    3,    6    }
Postorder : { 4, 2, 7, 8, 5, 6, 3, 1 }
```

Root would be the last element in the postorder sequence, i.e., `1`. Next, locate the index of the root node in the inorder sequence. Now since `1` is the root node, all nodes before `1` in the inorder sequence must be included in the left subtree of the root node, i.e., `{4, 2}` and all the nodes after `1` must be included in the right subtree, i.e., `{7, 5, 8, 3, 6}`. Now the problem is reduced to building the left and right subtrees and linking them to the root node.

Left subtree:

```
Inorder   : {4, 2}
Postorder : {4, 2}
```

```
Right subtree:

Inorder   : {7, 5, 8, 3, 6}
Postorder : {7, 8, 5, 6, 3}
```

- # Tower of Hanoi

1. It is a classic problem where you try to move all the disks from one peg to another peg using only three pegs.

2. Initially, all of the disks are stacked on top of each other with larger disks under the smaller disks.

3. You may move the disks to any of three pegs as you attempt to relocate all of the disks, but you cannot place the larger disks over smaller disks and only one disk can be transferred at a time.

This problem can be easily solved by Divide & Conquer algorithm

Tower of Hanoi using Recursion:

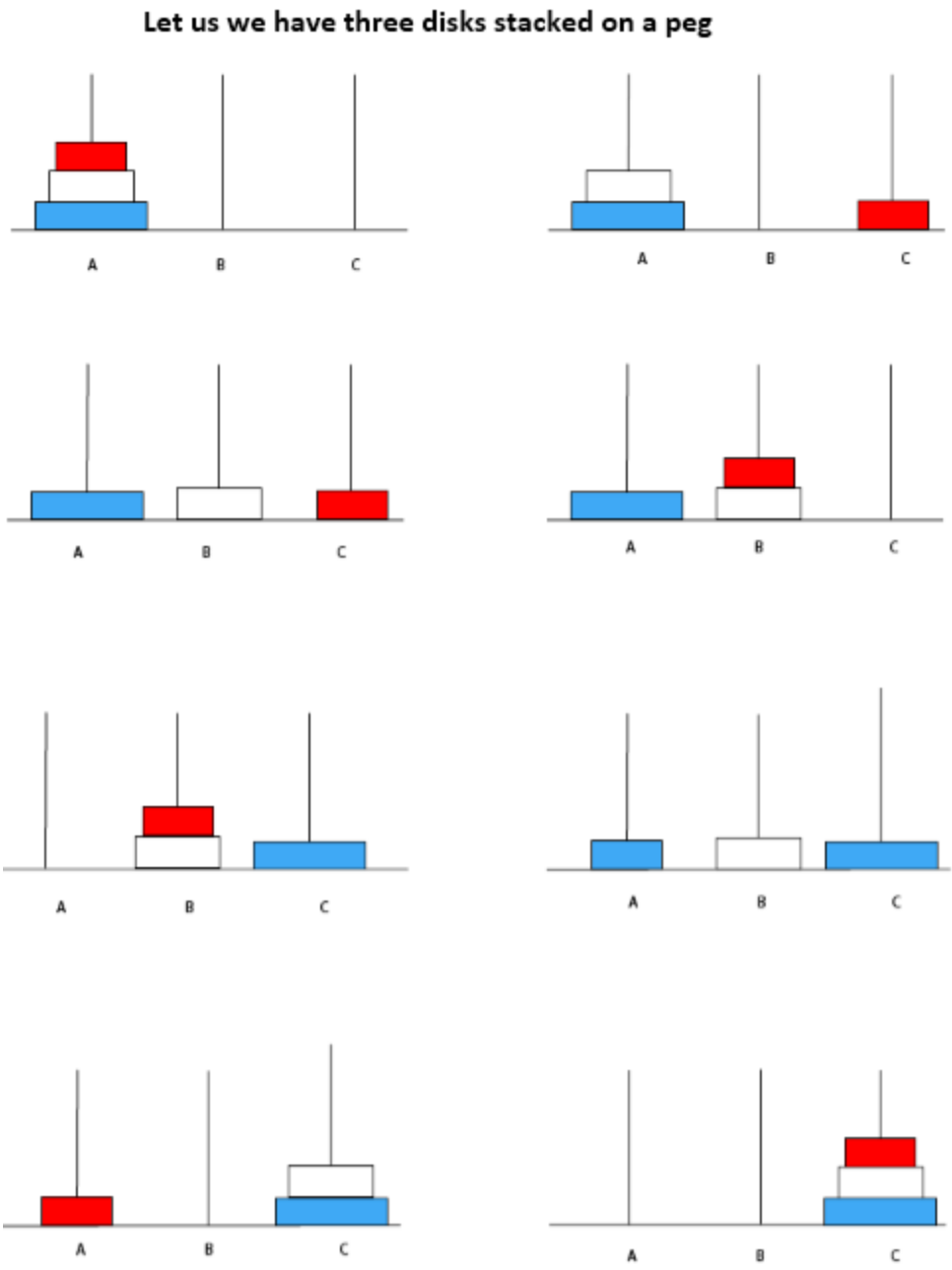
The idea is to use the helper node to reach the destination using recursion. Below is the pattern for this problem:

- *Shift 'N-1' disks from 'A' to 'B', using C.*
- *Shift last disk from 'A' to 'C'.*
- *Shift 'N-1' disks from 'B' to 'C', using A.*

Follow the steps below to solve the problem:

- Create a function **towerOfHanoi** where pass the **N** (current number of disk), **from_rod**, **to_rod**, **aux_rod.**
- Make a function call for N – 1 th disk.
- Then print the current the disk along with **from_rod** and **to_rod**
- Again make a function call for N – 1 th disk.

Below is the implementation of the above approach.



In the above 7 step all the disks from peg A will be transferred to C given Condition:

1. Only one disk will be shifted at a time.
2. Smaller disk can be placed on larger disk.

Let T (n) be the total time taken to move n disks from peg A to peg C

1. Moving n-1 disks from the first peg to the second peg. This can be done in T (n-1) steps.
2. Moving larger disks from the first peg to the third peg will require first one step.
3. Recursively moving n-1 disks from the second peg to the third peg will require again T (n-1) step.

So, total time taken T (n) = T (n-1)+1+ T(n-1)

Relation formula for Tower of Hanoi is:

$$T(n) = 2T(n-1) + 1$$

Note: Stopping Condition: T (1) =1

Because at last there will be one disk which will have to move from one peg to another.

T (n) = 2T (n-1) +1.....eq1

Put n = n-1 in eq 1

T (n-1) = 2T (n-2) +1.....eq2

Putting 2 eq in 1 eq

T (n) = 2(2T (n-2) +1) +1

T (n) = 2^2T (n-2) +2+1.....eq3

Put n = n-2 in eq 1

T (n-2) = 2T (n-3) +1.....eq4

Putting 4 eq in 3 eq

$T(n) = 2^2[2T(n-3)+1] + 2 + 1$

$T(n) = 2^3T(n-3) + 2^2 + 2 + 1$.....eq5

From 1 eq, 3 eq, 5 eq

We get,

$$T(n) = 2^{i}T(n-i) + 2^{i-1} + 2^{i-2} + \ldots\ldots\ldots\ldots + 2^{0}$$

Now n-i=1 from stopping condition

And T (n-i) =1

n-1 =i ← A equation

Now, $T(n) = 2^{i}(1) + 2^{i-1} + 2^{i-2} + \ldots\ldots\ldots\ldots\ldots\ldots\ldots + 2^{0}$

It is a Geometric Progression Series with common ratio, r=2
First term, $a=1(2^{0})$

Sum of n terms in G.P= $S_n = \frac{a(1-r^{n})}{1-r}$

So $T(n) = \frac{1(1-2^{i+1})}{(1-2)}$

$T(n) = \frac{2^{i+1}-1}{2-1} = 2^{i+1} - 1$

$T(n) = 2^{i+1} - 1$

Because exponents are from 0 to 1

n=i+1

From A

$$T(n) = 2^{n-1+1} - 1$$

$T(n) = 2^{n} - 1$.....B Equation

B equation is the required complexity of technique tower of Hanoi when we have to move n disks from one peg to another.

$T(3) = 2^{3} - 1$
$= 8 - 1 =$ **7 Ans**

[As in concept we have proved that there will be 7 steps now proved by general equation]

Program of Tower of Hanoi:

<script>

// javascript recursive function to

```
// solve tower of hanoi puzzle
function towerOfHanoi(n, from_rod, to_rod, aux_rod)
{
        if (n == 0)
        {
            return;
        }
        towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
        document.write("Move disk " + n + " from rod " + from_rod +
        " to rod " + to_rod+"<br/>");
        towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
    }

    // Driver code
    var N = 3;

    // A, B and C are names of rods
    towerOfHanoi(N, 'A', 'C', 'B');

// This code is contributed by gauravrajput1
</script>
```

Output:

```
Enter the number of disks: 3
The sequence of moves involved in the Tower of Hanoi is
Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg
```

METHOD NAME	EQUATION	STOPPING CONDITION	COMPLEXITIES
1.Max&Min	$T(n) = 2T\left(\frac{n}{2}\right)+2$	T(2)=1	$T(n) = \frac{3N}{2} - 2$
2.Binary Search	$T(n) = T\left(\frac{n}{2}\right) + 1$	T(1)=1	T(n)=log n
3.Merge Sort	$T(n)=2T\left(\frac{n}{2}\right)+n$	T(1)=0	T(n)=nlog n
4. Tower of Hanoi	T(n)=2T(n-1)+1	T(1)=1	$T(n)=2^n-1$

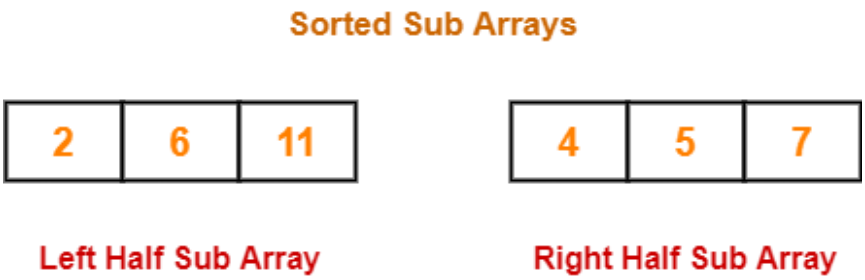
- **<u>Merge Sort-</u>**

- Merge sort is a famous sorting algorithm.
- It uses a divide and conquer paradigm for sorting.
- It divides the problem into sub problems and solves them individually.
- It then combines the results of sub problems to get the solution of the original problem.

How Merge Sort Works?

Before learning how merge sort works, let us learn about the merge procedure of merge sort algorithm.
The merge procedure of merge sort algorithm is used to merge two sorted arrays into a third array in sorted order.

Consider we want to merge the following two sorted sub arrays into a third array in sorted order-



The merge procedure of merge sort algorithm is given below-

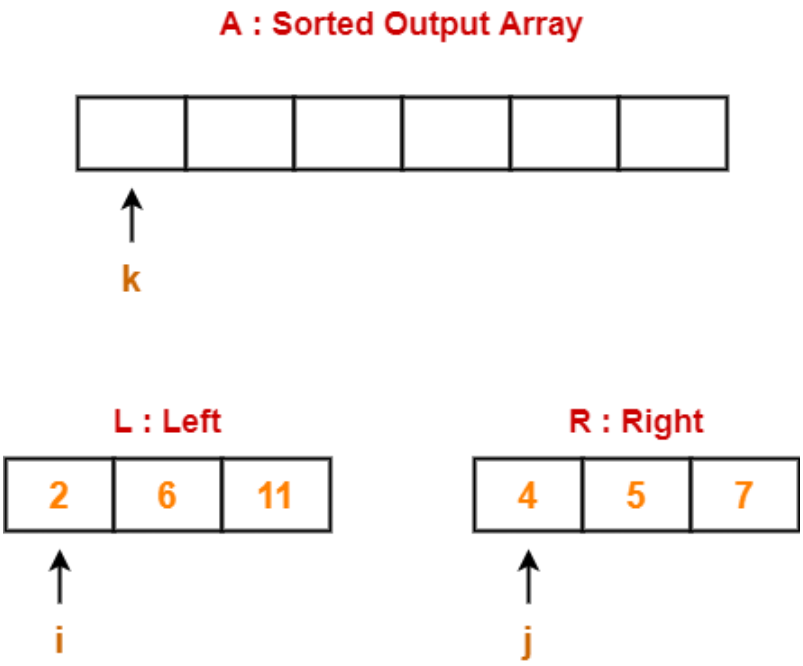
```
// L : Left Sub Array , R : Right Sub Array , A : Array
merge(L, R, A)
{
nL = length(L) // Size of Left Sub Array
nR = length(R) // Size of Right Sub Array
i = j = k = 0
while(i<nL && j<nR)
{
/* When both i and j are valid i.e. when both the sub arrays have elements to insert in A */
if(L[i] <= R[j])
{
A[k] = L[i]
k = k+1
i = i+1
}
else
{
A[k] = R[j]
k = k+1
```

```
j = j+1
}
}
// Adding Remaining elements from left sub array to array A
while(i<nL)
{
A[k] = L[i]
i = i+1
k = k+1
}
// Adding Remaining elements from right sub array to array A
while(j<nR)
{
A[k] = R[j]
j = j+1
k = k+1
}
}
```

The above merge procedure of merge sort algorithm is explained in the following steps-

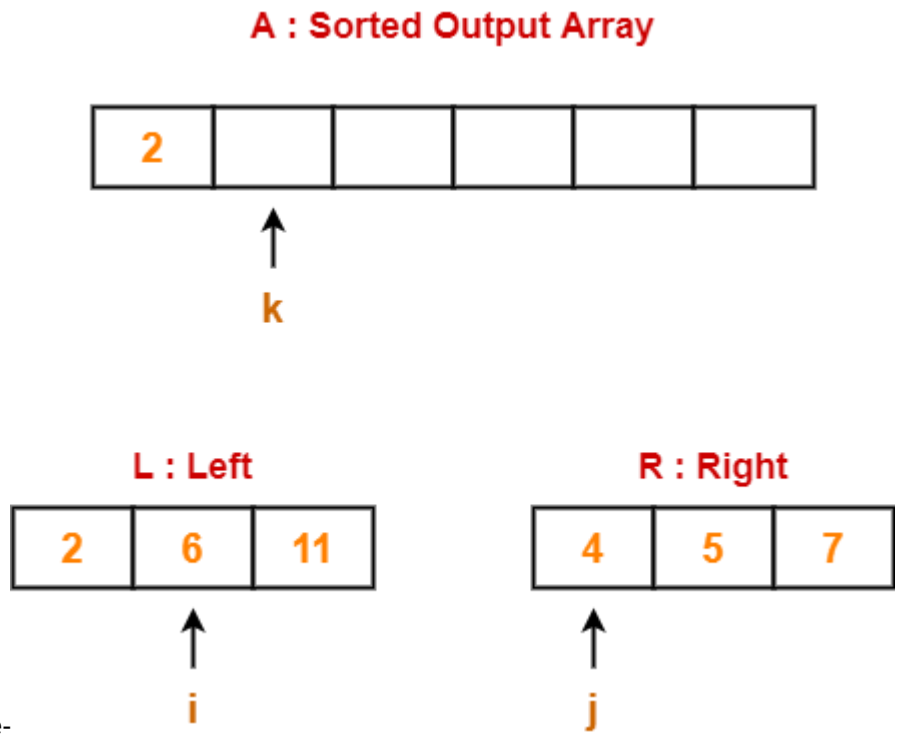
Step-01:

- Create two variables i and j for left and right sub arrays.
- Create variable k for sorted output array.



Step-02:

- We have i = 0, j = 0, k = 0.
- Since L[0] < R[0], so we perform A[0] = L[0] i.e. we copy the first element from left sub array to our sorted output array.
- Then, we increment i and k by 1.

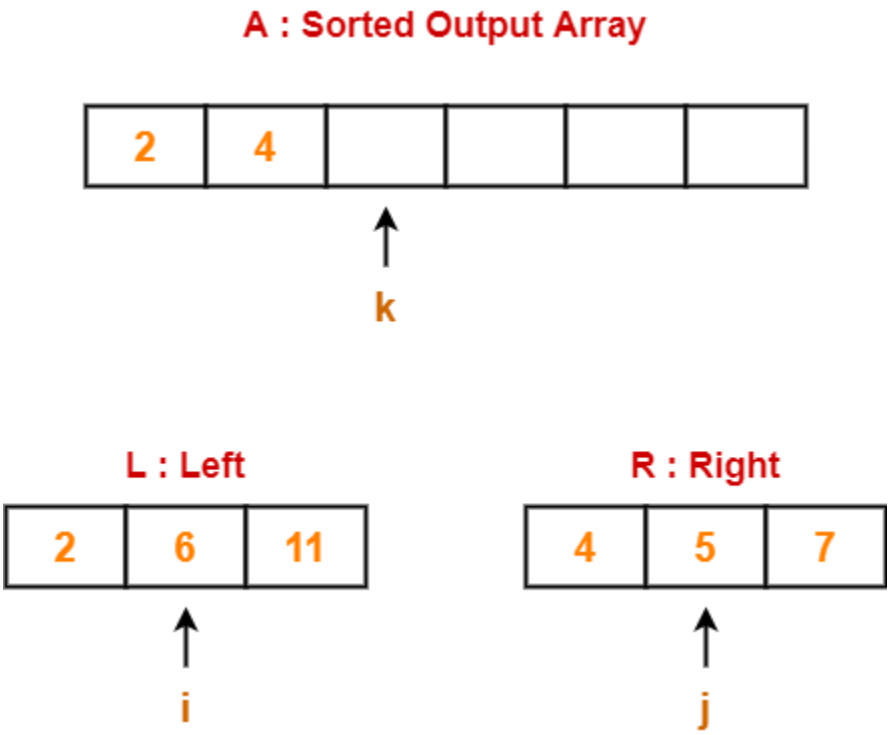


Then, we have-

Step-03:

- We have i = 1, j = 0, k = 1.
- Since L[1] > R[0], so we perform A[1] = R[0] i.e. we copy the first element from right sub array to our sorted output array.
- Then, we increment j and k by 1.

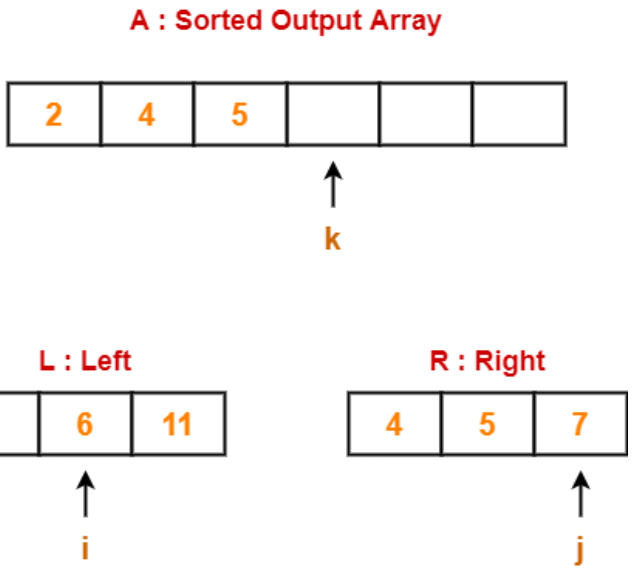
Then, we have-



Step-04:

- We have i = 1, j = 1, k = 2.
- Since L[1] > R[1], so we perform A[2] = R[1].
- Then, we increment j and k by 1.

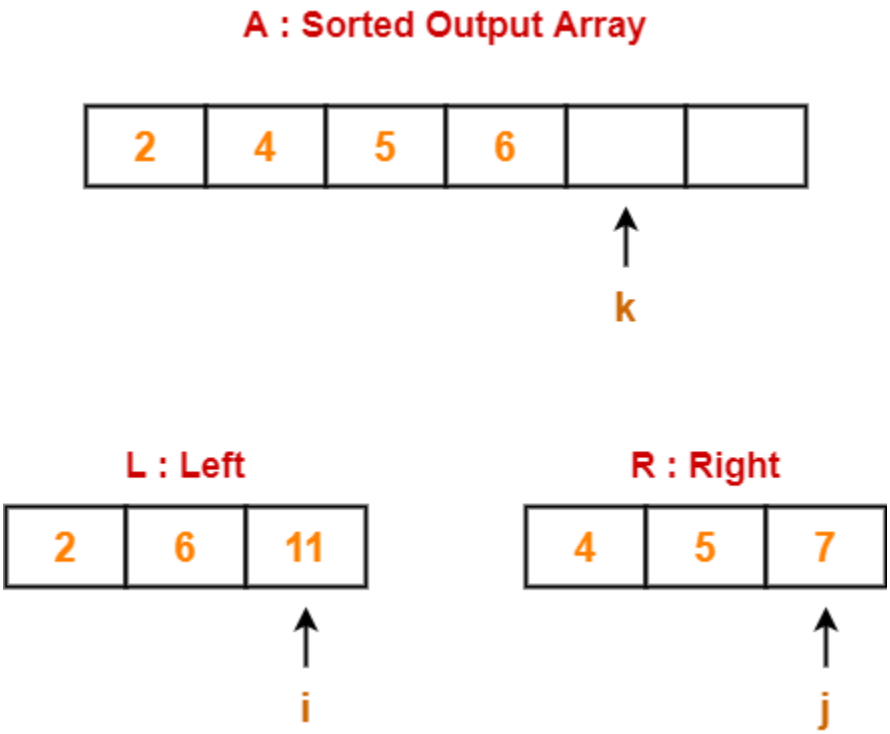
Then, we have-



Step-05:

- We have i = 1, j = 2, k = 3.
- Since L[1] < R[2], so we perform A[3] = L[1].
- Then, we increment i and k by 1.

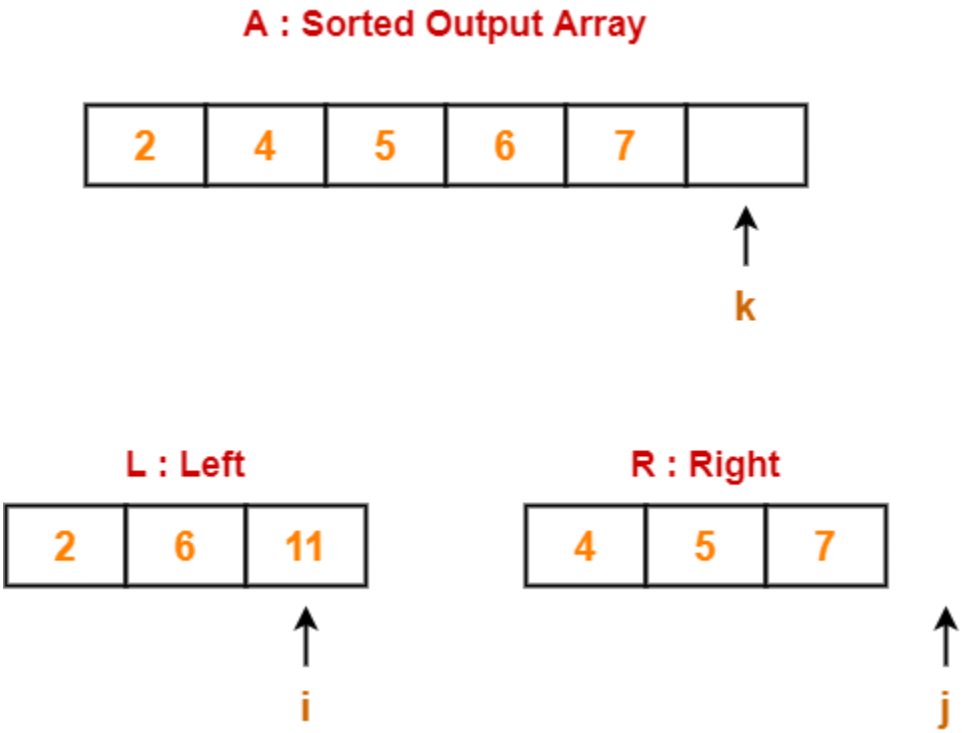
Then, we have-



Step-06:

- We have i = 2, j = 2, k = 4.
- Since L[2] > R[2], so we perform A[4] = R[2].
- Then, we increment j and k by 1.

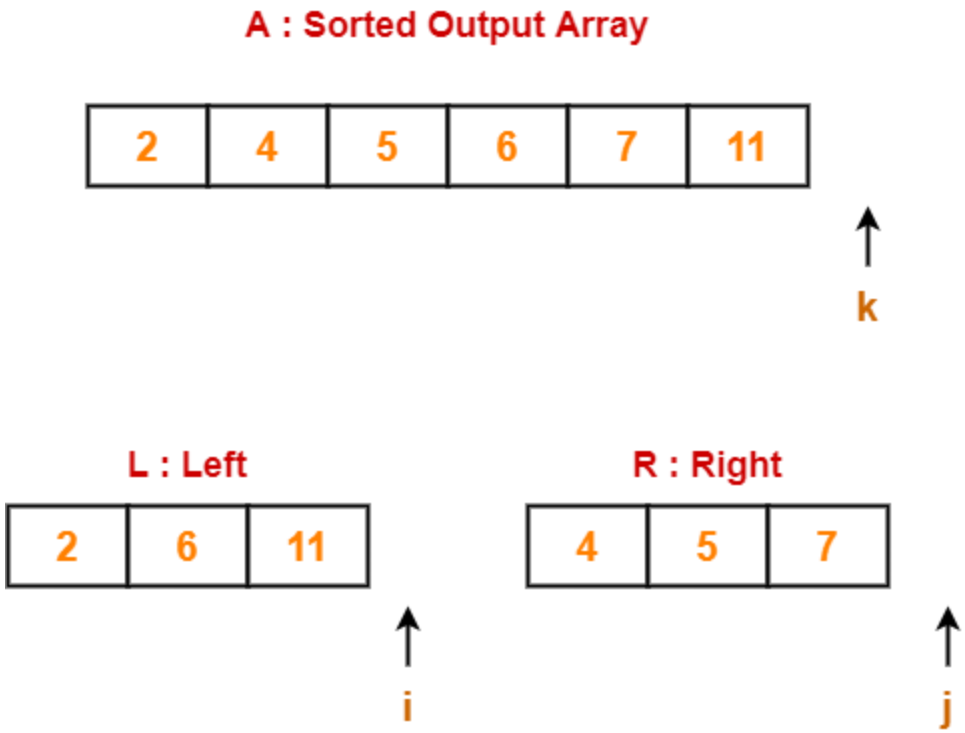
Then, we have-



Step-07:

- Clearly, all the elements from right sub array have been added to the sorted output array.
- So, we exit the first while loop with the condition while(i<nL && j<nR) since now j>nR.
- Then, we add remaining elements from the left sub array to the sorted output array using next while loop.

Finally, our sorted output array is-



Basically,

- After finishing elements from any of the sub arrays, we can add the remaining elements from the other sub array to our sorted output array as it is.
- This is because left and right sub arrays are already sorted.

Time Complexity

The above mentioned merge procedure takes Θ(n) time.

This is because we are just filling an array of size n from left & right sub arrays by incrementing i and j at most Θ(n) times.

Merge Sort Algorithm-

Merge Sort Algorithm works in the following steps-

- It divides the given unsorted array into two halves- left and right sub arrays.
- The sub arrays are divided recursively.
- This division continues until the size of each sub array becomes 1.
- After each sub array contains only a single element, each sub array is sorted trivially.
- Then, the above discussed merge procedure is called.
- The merge procedure combines these trivially sorted arrays to produce a final sorted array.

The division procedure of merge sort algorithm which uses recursion is given below-

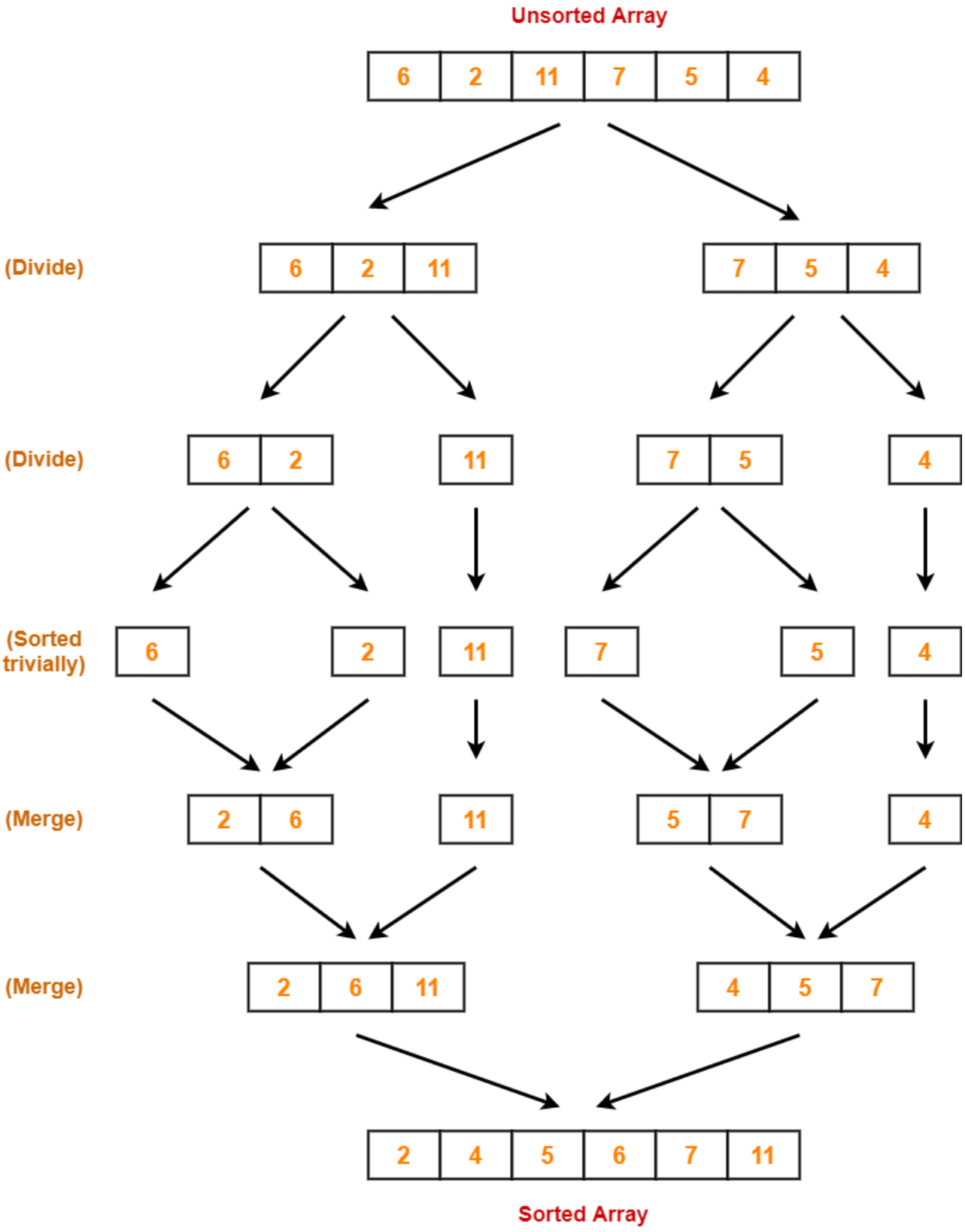
// A : Array that needs to be sorted

```
MergeSort(A)
{
n = length(A)
if n<2 return
mid = n/2
left = new_array_of_size(mid) // Creating temporary array for left
right = new_array_of_size(n-mid) // and right sub arrays
for(int i=0 ; i<=mid-1 ; ++i)
{
left[i] = A[i] // Copying elements from A to left
}
for(int i=mid ; i<=n-1 ; ++i)
{
right[i-mid] = A[i] // Copying elements from A to right
}
MergeSort(left) // Recursively solving for left sub array
MergeSort(right) // Recursively solving for right sub array
merge(left, right, A) // Merging two sorted left/right sub array to final array
}
```

Merge Sort Example-

Consider the following elements have to be sorted in ascending order-
6, 2, 11, 7, 5, 4

The merge sort algorithm works as-



Time Complexity Analysis-

In merge sort, we divide the array into two (nearly) equal halves and solve them recursively using merge sort only.

So, we have-

$$T\left(\frac{n_L}{2}\right) + T\left(\frac{n_R}{2}\right) = 2T\left(\frac{n}{2}\right)$$

n_L = Left Half

n_R = Right Half

$n_L \approx n_R$

Finally, we merge these two sub arrays using merge procedure which takes Θ(n) time as explained above.

If T(n) is the time required by merge sort for sorting an array of size n, then the recurrence relation for time complexity of merge sort is-

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Recurrence Relation

On solving this recurrence relation, we get T(n) = Θ(nlogn).

Thus, time complexity of merge sort algorithm is T(n) = Θ(nlogn).

Space Complexity Analysis-

- Merge sort uses additional memory for left and right sub arrays.
- Hence, total Θ(n) extra memory is needed.

Properties-

Some of the important properties of merge sort algorithm are-

- Merge sort uses a divide and conquer paradigm for sorting.
- Merge sort is a recursive sorting algorithm.
- Merge sort is a stable sorting algorithm.
- Merge sort is not an in-place sorting algorithm.
- The time complexity of merge sort algorithm is Θ(nlogn).
- The space complexity of merge sort algorithm is Θ(n)

Merge Sort-

- Merge sort is a famous sorting algorithm.
- It uses a divide and conquer paradigm for sorting.
- It divides the problem into sub problems and solves them individually.
- It then combines the results of sub problems to get the solution of the original problem.

Minimum number of jumps to reach the end using Recursion

Start from the first element and recursively call for all the elements reachable from the first element. The minimum number of jumps to reach end from first can be calculated using the minimum value from the recursive calls.
minJumps(start, end) = Min (minJumps(k, end)) *for all k reachable from start.*
Follow the steps mentioned below to implement the idea:

- Create a recursive function.
- In each recursive call get all the reachable nodes from that index.
    - For each of the index call the recursive function.
    - Find the minimum number of jumps to reach the end from current index.
- Return the minimum number of jumps from the recursive call.

Below is the Implementation of the above approach:

```
// JavaScript program to find Minimum
// number of jumps to reach end

// Function to return the minimum number
// of jumps to reach arr[h] from arr[l]
function minJumps(arr, n)
{

        // Base case: when source and
        // destination are same
        if (n == 1)
                return 0;

        // Traverse through all the points
        // reachable from arr[l]
        // Recursively, get the minimum number
        // of jumps needed to reach arr[h] from
```

```
        // these reachable points
        let res = Number.MAX_VALUE;
        for (let i = n - 2; i >= 0; i--) {
                if (i + arr[i] >= n - 1) {
                        let sub_res = minJumps(arr, i + 1);
                        if (sub_res != Number.MAX_VALUE)
                                res = Math.min(res, sub_res + 1);
                }
        }

        return res;
}


// Driver Code

        let arr = [ 1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9 ];
        let n = arr.length;
        document.write("Minimum number of jumps to");
        document.write(" reach end is " + minJumps(arr, n));
```

• Prim's Algorithm

In this article, we will discuss the prim's algorithm. Along with the algorithm, we will also see the complexity, working, example, and implementation of prim's algorithm.

Before starting the main topic, we should discuss the basic and important terms such as spanning tree and minimum spanning tree.

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. Backward Skip 10sPlay VideoForward Skip 10s

Now, let's start the main topic.

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

How does the prim's algorithm work?

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

- First, we have to initialize an MST with the randomly chosen vertex.
- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed.

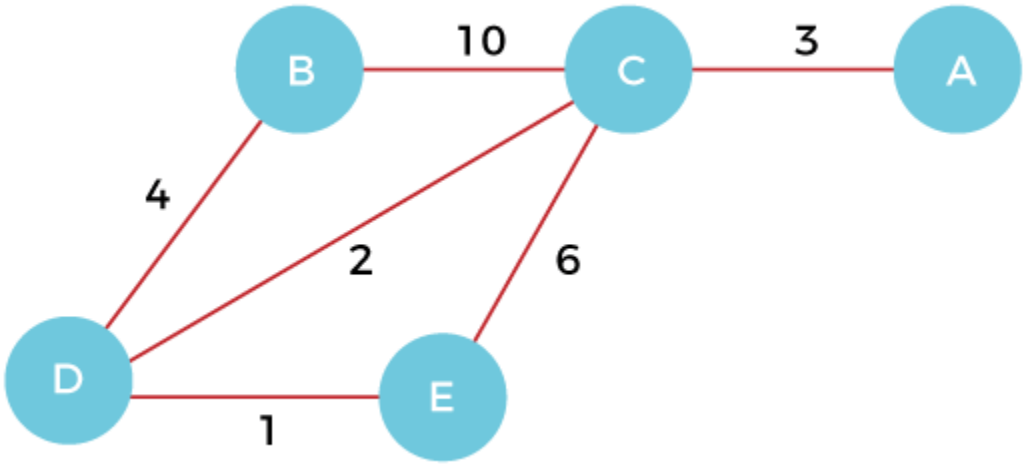
The applications of prim's algorithm are -

- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

Example of prim's algorithm

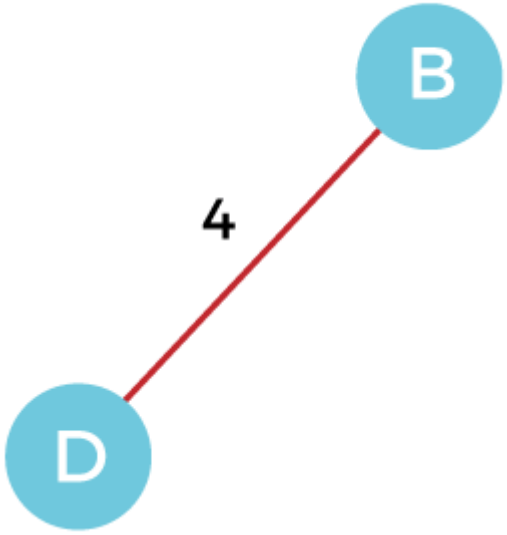
Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

Suppose, a weighted graph is -

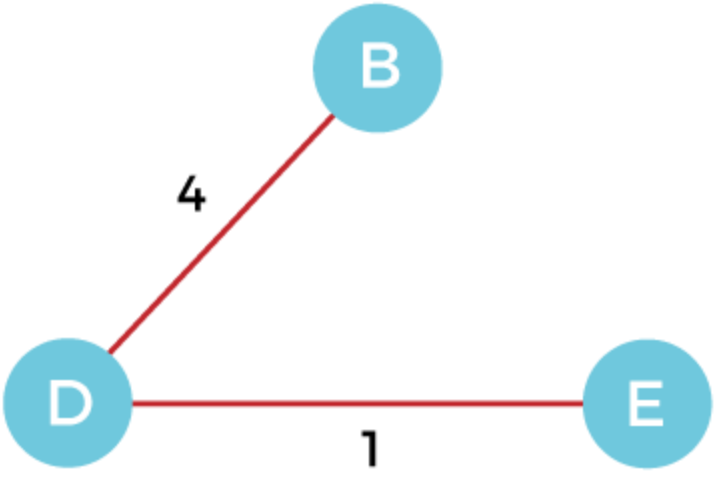


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

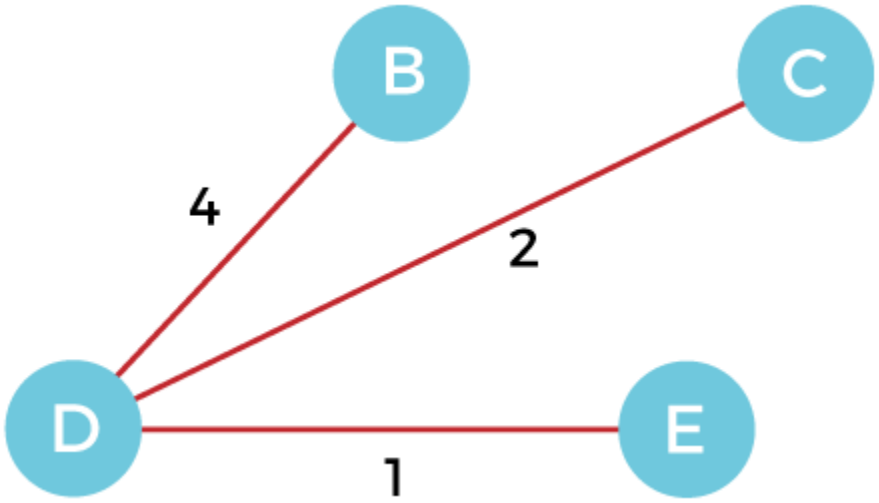
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



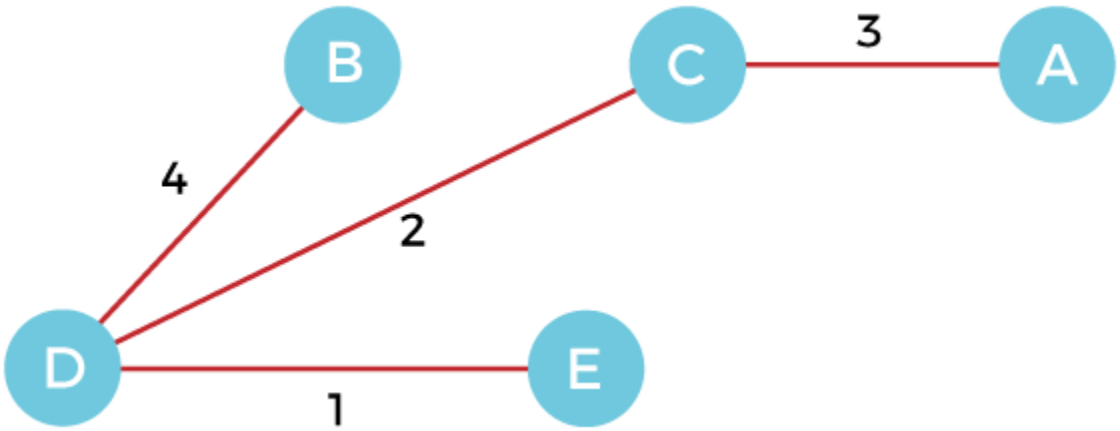
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Algorithm

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices
3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T
5. [END OF LOOP]
6. Step 5: EXIT

Complexity of Prim's algorithm

Now, let's see the time complexity of Prim's algorithm. The running time of the prim's algorithm depends upon using the data structure for the graph and the ordering of edges. Below table shows some choices -

- **Time Complexity**

Data structure used for the minimum edge weight	Time Complexity
Adjacency matrix, linear searching	$O(\|V\|^2)$
Adjacency list and binary heap	O(|E| log |V|)
Adjacency list and Fibonacci heap	O(|E|+ |V| log |V|)

Prim's algorithm can be simply implemented by using the adjacency matrix or adjacency list graph representation, and to add the edge with the minimum weight requires the linearly searching of an array of weights. It requires $O(|V|^2)$ running time. It can be improved further by using the implementation of heap to find the minimum weight edges in the inner loop of the algorithm.

The time complexity of the prim's algorithm is O(E logV) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

Implementation of Prim's algorithm

Now, let's see the implementation of prim's algorithm.

Program: Write a program to implement prim's algorithm in C language.

```
1.  #include <stdio.h>
2.  #include <limits.h>
3.  #define vertices 5  /*Define the number of vertices in the graph*/
4.  /* create minimum_key() method for finding the vertex that has minimum key-
    value and that is not added in MST yet */
5.  int minimum_key(int k[], int mst[])
6.  {
7.      int minimum  = INT_MAX, min,i;
8.
9.      /*iterate over all vertices to find the vertex with minimum key-value*/
10.     for (i = 0; i < vertices; i++)
```

```
11.         if (mst[i] == 0 && k[i] < minimum )
12.             minimum = k[i], min = i;
13.     return min;
14.  }
15.  /* create prim() method for constructing and printing the MST.
16.  The g[vertices][vertices] is an adjacency matrix that defines the graph for MST.*/
17.  void prim(int g[vertices][vertices])
18.  {
19.     /* create array of size equal to total number of vertices for storing the MST*/
20.     int parent[vertices];
21.     /* create k[vertices] array for selecting an edge having minimum weight*/
22.     int k[vertices];
23.     int mst[vertices];
24.     int i, count,edge,v; /*Here 'v' is the vertex*/
25.     for (i = 0; i < vertices; i++)
26.     {
27.         k[i] = INT_MAX;
28.         mst[i] = 0;
29.     }
30.     k[0] = 0; /*It select as first vertex*/
31.     parent[0] = -1;   /* set first value of parent[] array to -1 to make it root of MST*/
32.     for (count = 0; count < vertices-1; count++)
33.     {
34.         /*select the vertex having minimum key and that is not added in the MST yet from the set of vertices*/
35.         edge = minimum_key(k, mst);
36.         mst[edge] = 1;
37.         for (v = 0; v < vertices; v++)
38.         {
39.             if (g[edge][v] && mst[v] == 0 && g[edge][v] <  k[v])
40.             {
41.                 parent[v]  = edge, k[v] = g[edge][v];
42.             }
43.         }
44.      }
```

```
45.     /*Print the constructed Minimum spanning tree*/
46.     printf("\n Edge \t  Weight\n");
47.     for (i = 1; i < vertices; i++)
48.     printf(" %d <-> %d    %d \n", parent[i], i, g[i][parent[i]]);
49.
50. }
51. int main()
52. {
53.    int g[vertices][vertices] = {{0, 0, 3, 0, 0},
54.                                 {0, 0, 10, 4, 0},
55.                                 {3, 10, 0, 2, 6},
56.                                 {0, 4, 2, 0, 1},
57.                                 {0, 0, 6, 1, 0},
58.                                 };
59.    prim(g);
60.    return 0;
61. }
```

Output

```
Edge      Weight
3 <-> 1     4
0 <-> 2     3
2 <-> 3     2
3 <-> 4     1
```

Kruskal's algorithm is a greedy algorithm that works as follows –

1. It Creates a set of all edges in the graph.

2. While the above set is not empty and not all vertices are covered,
   - It removes the minimum weight edge from this set
   - It checks if this edge is forming a cycle or just connecting 2 trees. If it forms a cycle, we discard this edge, else we add it to our tree.

3. When the above processing is complete, we have a minimum spanning tree.

In order to implement this algorithm, we need 2 more data structures.

First, we need a priority queue that we can use to keep the edges in a sorted order and get our required edge on each iteration.

Next, we need a disjoint set data structure. A disjoint-set data structure (also called a union-find data structure or merge–find set) is a data structure that tracks a set of elements partitioned into a number of disjoint (non-overlapping) subsets. Whenever we add a new node to a tree, we will check if they are already connected. If yes, then we have a cycle. If no, we will make a union of both vertices of the edge. This will add them to the same subset.

Let us look at the implementation of UnionFind or DisjointSet data structure &minsu;

Example

```
class UnionFind {
   constructor(elements) {
      // Number of disconnected components
      this.count = elements.length;

      // Keep Track of connected components
      this.parent = {};

      // Initialize the data structure such that all
      // elements have themselves as parents
      elements.forEach(e => (this.parent[e] = e));
   }

   union(a, b) {
      let rootA = this.find(a);
      let rootB = this.find(b);

      // Roots are same so these are already connected.
      if (rootA === rootB) return;

      // Always make the element with smaller root the parent.
      if (rootA < rootB) {
         if (this.parent[b] != b) this.union(this.parent[b], a);
         this.parent[b] = this.parent[a];
      } else {
         if (this.parent[a] != a) this.union(this.parent[a], b);
         this.parent[a] = this.parent[b];
      }
   }

   // Returns final parent of a node
   find(a) {
      while (this.parent[a] !== a) {
         a = this.parent[a];
      }
```

```
      return a;
   }

   // Checks connectivity of the 2 nodes
   connected(a, b) {
      return this.find(a) === this.find(b);
   }
}
```

You can test this using –

Example

```
let uf = new UnionFind(["A", "B", "C", "D", "E"]);
uf.union("A", "B"); uf.union("A", "C");
uf.union("C", "D");

console.log(uf.connected("B", "E"));
console.log(uf.connected("B", "D"));
```

Output

This will give the output –

false

true

Now let us look at the implementation of Kruskal's algorithm using this data structure –

Example

```
kruskalsMST() {
   // Initialize graph that'll contain the MST
   const MST = new Graph();
   this.nodes.forEach(node => MST.addNode(node));
   if (this.nodes.length === 0) {
      return MST;
   }

   // Create a Priority Queue
   edgeQueue = new PriorityQueue(this.nodes.length * this.nodes.length);

   // Add all edges to the Queue:
   for (let node in this.edges) {
      this.edges[node].forEach(edge => {
         edgeQueue.enqueue([node, edge.node], edge.weight);
      });
   }
```

```
   let uf = new UnionFind(this.nodes);

   // Loop until either we explore all nodes or queue is empty
   while (!edgeQueue.isEmpty()) {
      // Get the edge data using destructuring
      let nextEdge = edgeQueue.dequeue();
      let nodes = nextEdge.data;
      let weight = nextEdge.priority;

      if (!uf.connected(nodes[0], nodes[1])) {
         MST.addEdge(nodes[0], nodes[1], weight);
         uf.union(nodes[0], nodes[1]);
      }
   }
   return MST;
}
```

You can test this using –

Example

```
let g = new Graph();
g.addNode("A");
g.addNode("B");
g.addNode("C");
g.addNode("D");
g.addNode("E");
g.addNode("F");
g.addNode("G");

g.addEdge("A", "C", 100);
g.addEdge("A", "B", 3);
g.addEdge("A", "D", 4);
g.addEdge("C", "D", 3);
g.addEdge("D", "E", 8);
g.addEdge("E", "F", 10);
g.addEdge("B", "G", 9);
g.addEdge("E", "G", 50);

g.kruskalsMST().display();
```

Output

This will give the output –

A->B, D

B->A, G

C->D

D->C, A, E

E->D, F

F->E

G->B

Kruskal's Algorithm

In this article, we will discuss Kruskal's algorithm. Here, we will also see the complexity, working, example, and implementation of the Kruskal's algorithm.

But before moving directly towards the algorithm, we should first understand the basic terms such as spanning tree and minimum spanning tree.

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Now, let's start with the main topic.

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

How does Kruskal's algorithm work?

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- First, sort all the edges from low weight to high.
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

The applications of Kruskal's algorithm are -

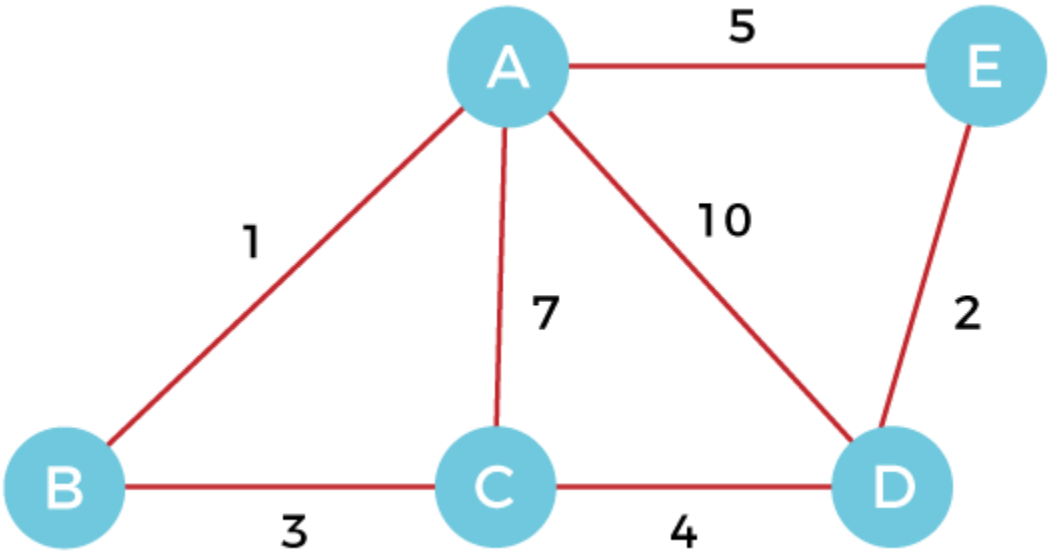
- Kruskal's algorithm can be used to layout electrical wiring among cities.
- It can be used to lay down LAN connections.

Example of Kruskal's algorithm

Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.

Suppose a weighted graph is -

Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2



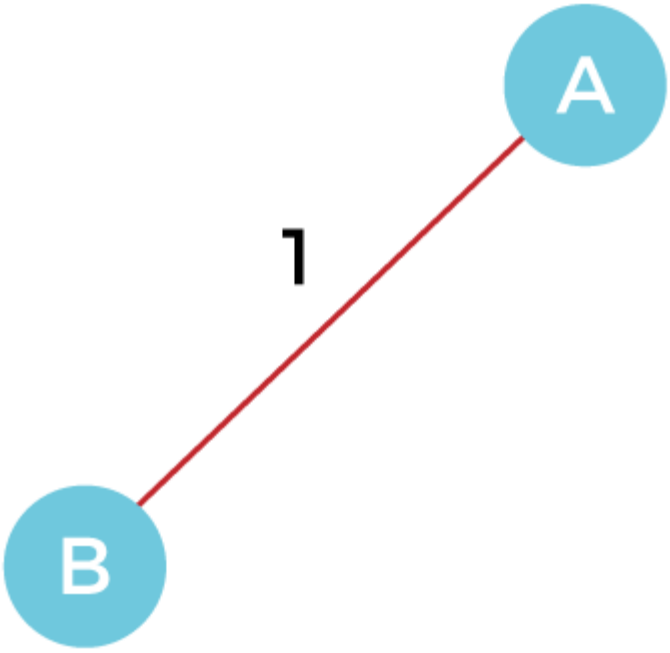
The weight of the edges of the above graph is given in the below table -

Now, sort the edges given above in the ascending order of their weights.

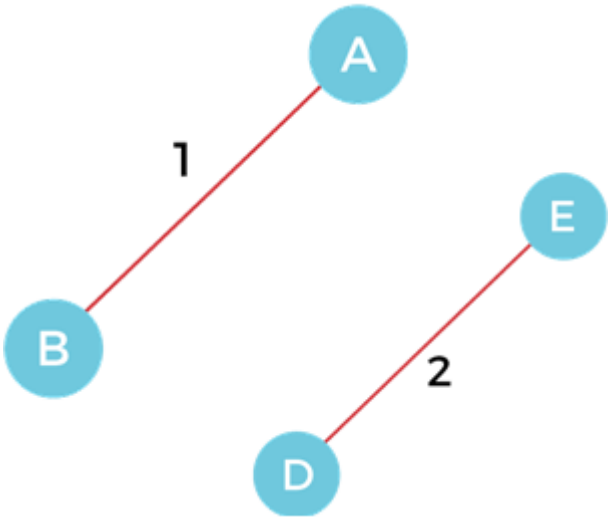
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Now, let's start constructing the minimum spanning tree.

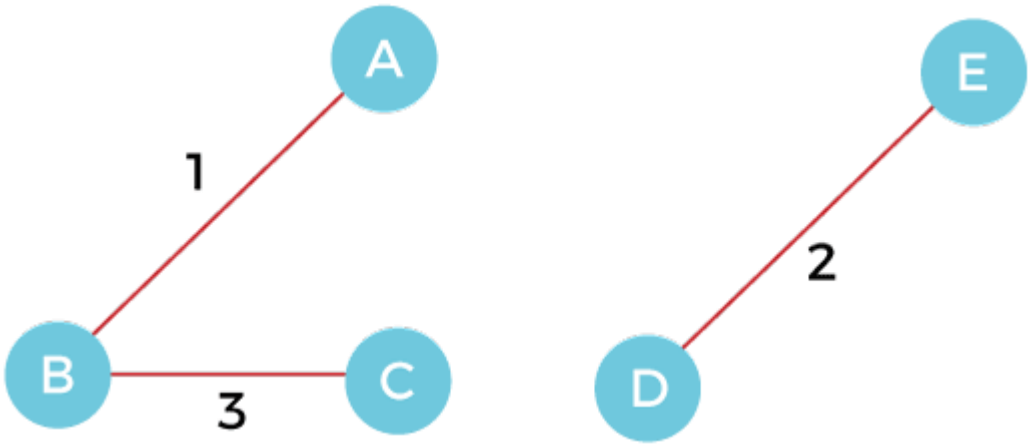
Step 1 - First, add the edge **AB** with weight **1** to the MST.



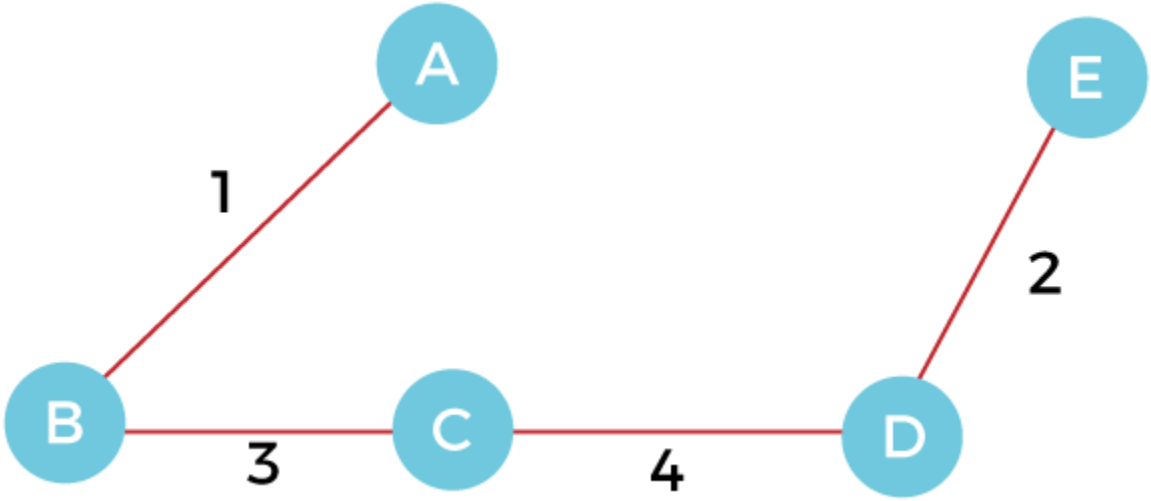
Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



Step 3 - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.



Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.

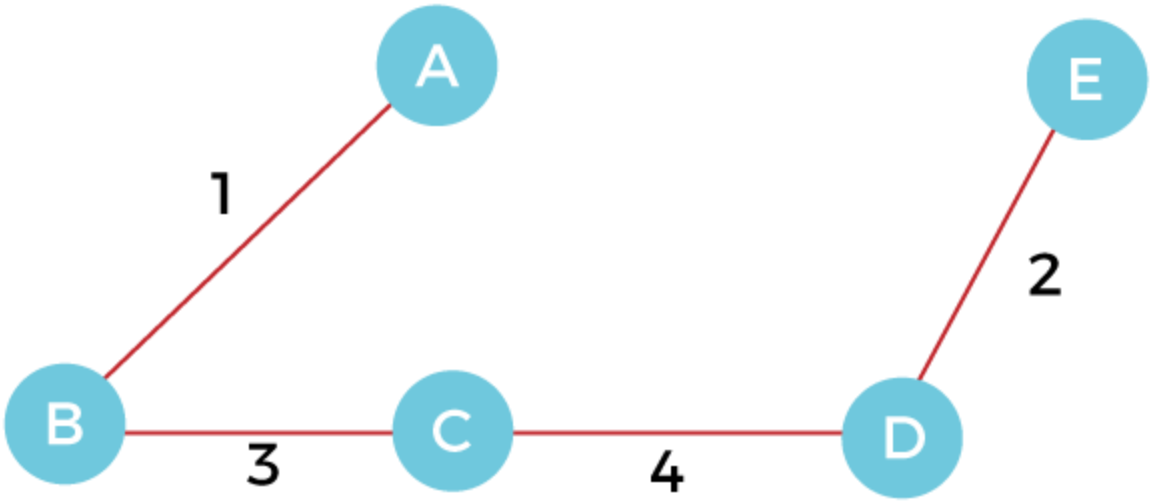


Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

Step 6 - Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

Step 7 - Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

Algorithm

1. Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.
2. Step 2: Create a set E that contains all the edges of the graph.
3. Step 3: Repeat Steps 4 and 5 **while** E is NOT EMPTY and F is not spanning
4. Step 4: Remove an edge from E with minimum weight
5. Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F
6. (**for** combining two trees into one tree).
7. ELSE
8. Discard the edge
9. Step 6: END

Complexity of Kruskal's algorithm

Now, let's see the time complexity of Kruskal's algorithm.

- **Time** **Complexity**

  The time complexity of Kruskal's algorithm is O(E logE) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

Implementation of Kruskal's algorithm

Now, let's see the implementation of kruskal's algorithm.

Program: Write a program to implement kruskal's algorithm in C++.

```
#include <iostream>
#include <algorithm>
using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];
void init()
{
    for(int i = 0;i < MAX;++i)
        id[i] = i;
}
int root(int x)
{
    while(id[x] != x)
    {
        id[x] = id[id[x]];
        x = id[x];
    }
    return x;
}
void union1(int x, int y)
{
    int p = root(x);
    int q = root(y);
    id[p] = id[q];
}
long long kruskal(pair<long long, pair<int, int> > p[])
{
    int x, y;
    long long cost, minimumCost = 0;
    for(int i = 0;i < edges;++i)
    {
        x = p[i].second.first;
```

```
34.         y = p[i].second.second;
35.         cost = p[i].first;
36.         if(root(x) != root(y))
37.         {
38.             minimumCost += cost;
39.             union1(x, y);
40.         }
41.     }
42.     return minimumCost;
43. }
44. int main()
45. {
46.     int x, y;
47.     long long weight, cost, minimumCost;
48.     init();
49.     cout <<"Enter Nodes and edges";
50.     cin >> nodes >> edges;
51.     for(int i = 0;i < edges;++i)
52.     {
53.         cout<<"Enter the value of X, Y and edges";
54.     cin >> x >> y >> weight;
55.         p[i] = make_pair(weight, make_pair(x, y));
56.     }
57.     sort(p, p + edges);
58.     minimumCost = kruskal(p);
59.     cout <<"Minimum cost is "<< minimumCost << endl;
60.     return 0;
61. }
```

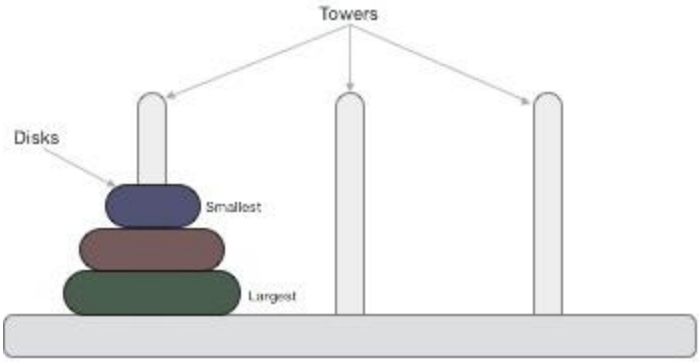
Output

```
Enter Nodes and edges 5 7
Enter the value of X, Y and edges 1 2 1
Enter the value of X, Y and edges 1 3 7
Enter the value of X, Y and edges 1 4 10
Enter the value of X, Y and edges 1 5 5
Enter the value of X, Y and edges 2 3 3
Enter the value of X, Y and edges 3 4 4
Enter the value of X, Y and edges 4 5 2
Minimum cost is 10
```

So, that's all about the article. Hope the article will be helpful and informative to you.

Data Structure & Algorithms - Tower of Hanoi

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −



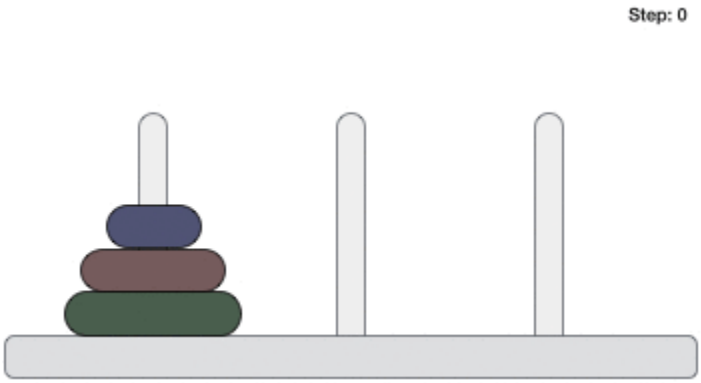
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



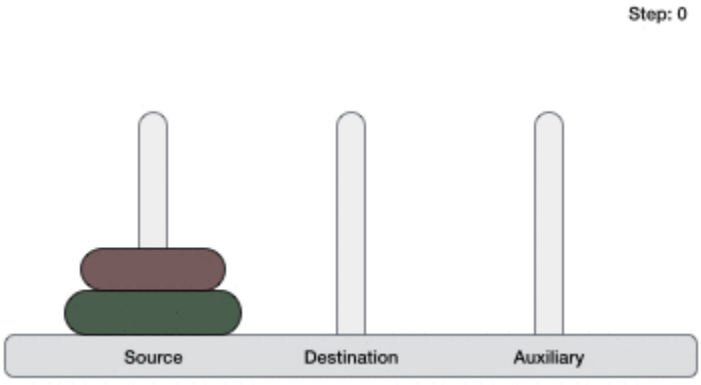
Tower of Hanoi puzzle with n disks can be solved in minimum 2^n-1 steps. This presentation shows that a puzzle with 3 disks has taken $2^3 - 1 = 7$ steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks –

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are –

Step 1 – Move n-1 disks from **source** to **aux**

Step 2 – Move n^{th} disk from **source** to **dest**

Step 3 – Move n-1 disks from **aux** to **dest**

A recursive algorithm for Tower of Hanoi can be driven as follows –

```
START
Procedure Hanoi(disk, source, dest, aux)

  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)    // Step 1
    move disk from source to dest         // Step 2
    Hanoi(disk - 1, aux, dest, source)    // Step 3
  END IF

END Procedure
STOP
```

Trapping Rain Water

Given an array of **N** non-negative integers **arr[]** representing an elevation map where the width of each bar is **1**, compute how much water it is able to trap after raining.

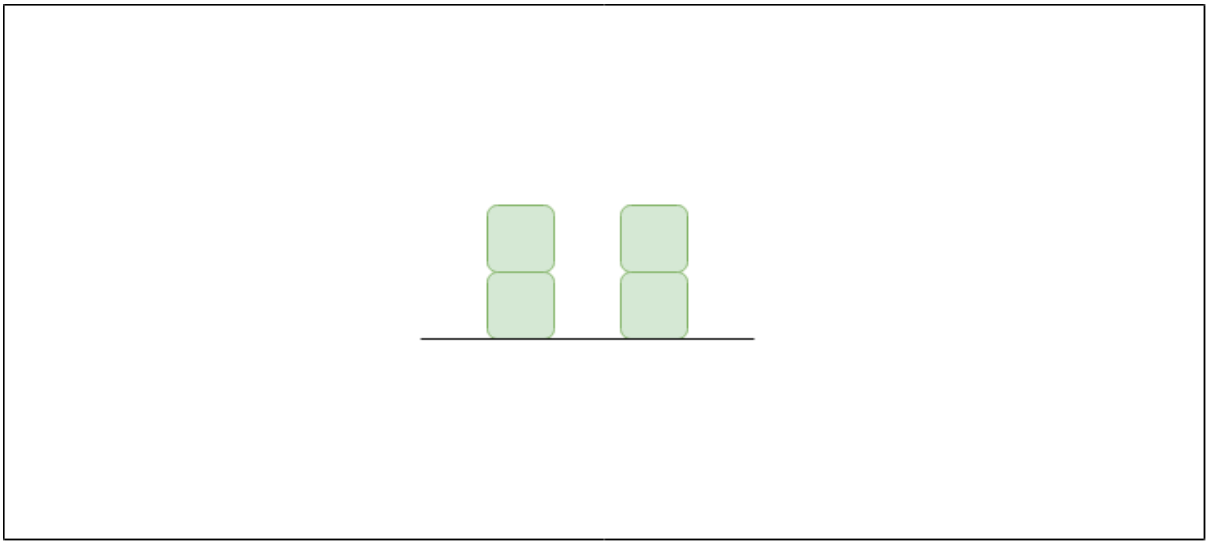
Examples:

Input: *arr[] = {2, 0, 2}*

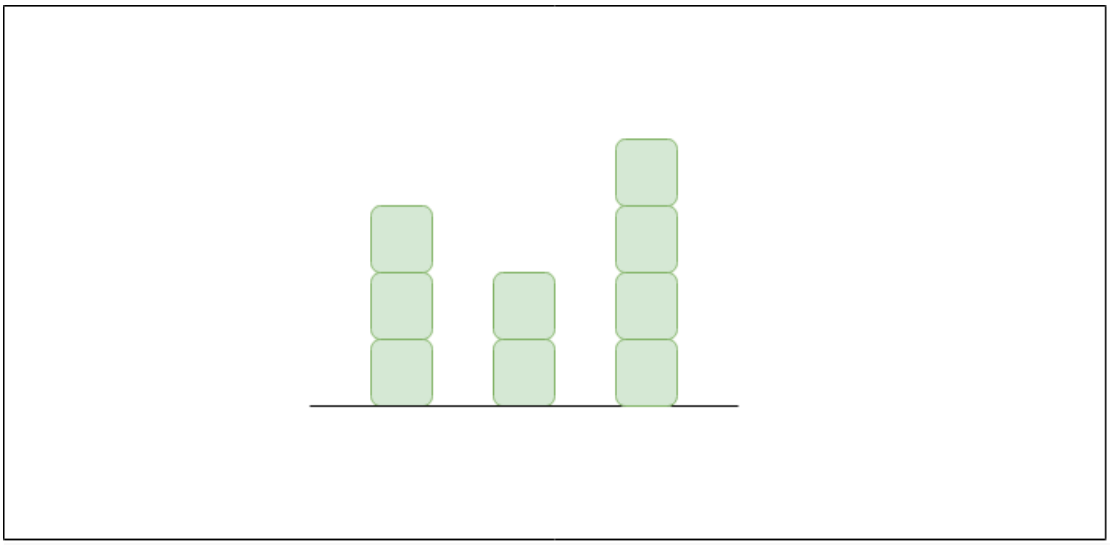
Output: *2*

Explanation: *The structure is like below.*

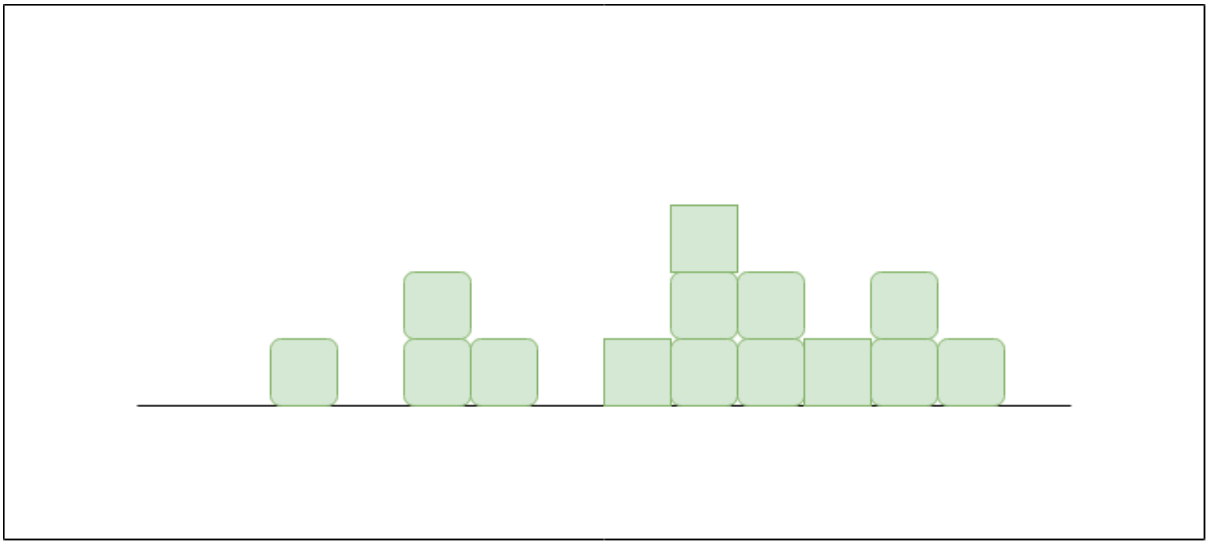
We can trap 2 units of water in the middle gap.



Input: *arr[] = {3, 0, 2, 0, 4}*
Output: *7*
Explanation: *Structure is like below. We can trap "3 units" of water between 3 and 2, "1 unit" on top of bar 2 and "3 units" between 2 and 4.*



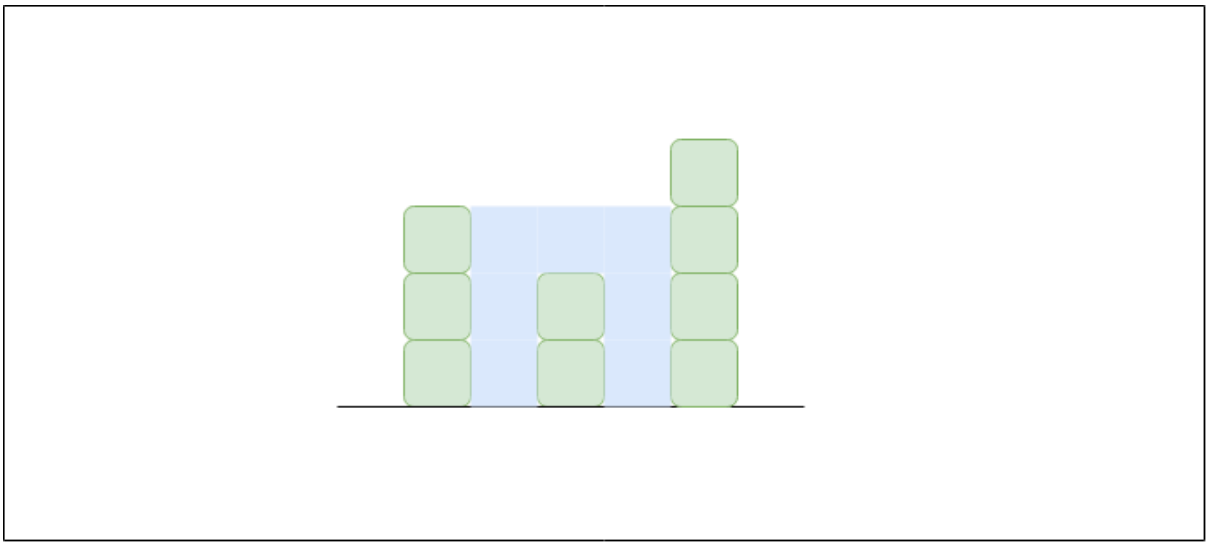
Input: *arr[] = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1}*
Output: *6*
Explanation: *The structure is like below. Trap "1 unit" between first 1 and 2, "4 units" between first 2 and 3 and "1 unit" between second last 1 and last 2*



Intuition: The basic intuition of the problem is as follows:

- *An element of the array can store water if there are higher bars on the left and the right.*
- *The amount of water to be stored in every position can be found by finding the heights of bars on the left and right sides.*
- *The total amount of water stored is the summation of the water stored in each index.*

For example** – Consider the array **arr[] = {3, 0, 2, 0, 4}**. Three units of water can be stored in two indexes 1 and 3, and one unit of water at index 2. Water stored in each index = 0 + 3 + 1 + 3 + 0 = **7



Traverse every array element and find the highest bars on the left and right sides. Take the smaller of two heights. The difference between the smaller height and the height of the current element is the amount of water that can be stored in this array element.

Follow the steps mentioned below to implement the idea:

- Traverse the array from start to end:
    - For every element:
        - Traverse the array from start to that index and find the maximum height *(a)* and
        - Traverse the array from the current index to the end, and find the maximum height *(b)*.
- The amount of water that will be stored in this column is min(a,b) – array[i], add this value to the total amount of water stored
- Print the total amount of water stored.

Below is the implementation of the above approach.

```
<script>

    // Javascript implementation of the approach

    // Function to return the maximum
    // water that can be stored
    function maxWater(arr, n)
    {

        // To store the maximum water
        // that can be stored
        let res = 0;

        // For every element of the array
        // except first and last element
        for(let i = 1; i < n - 1; i++)
        {

            // Find maximum element on its left
            let left = arr[i];
            for(let j = 0; j < i; j++)
            {
                left = Math.max(left, arr[j]);
            }

            // Find maximum element on its right
            let right = arr[i];
            for(let j = i + 1; j < n; j++)
            {
                right = Math.max(right, arr[j]);
            }

            // Update maximum water value
            res += Math.min(left, right) - arr[i];
        }
        return res;
    }

    let arr = [ 0, 1, 0, 2, 1, 0,
                1, 3, 2, 1, 2, 1 ];
    let n = arr.length;
```

```
        document.write(maxWater(arr,n));

</script>
```

Linear Search Algorithm:

Step 1: First, read the search element (Target element) in the array.
Step 2: In the second step compare the search element with the first element in the array.
Step 3: If both are matched, display “Target element is found” and terminate the Linear Search function.
Step 4: If both are not matched, compare the search element with the next element in the array.
Step 5: In this step, repeat steps 3 and 4 until the search (Target) element is compared with the last element of the array.
Step 6 – If the last element in the list does not match, the Linear Search Function will be terminated, and the message “Element is not found” will be displayed.

Given an array arr[] of N elements, the task is to write a function to search a given element x in arr[].

Examples:

Input: arr[] = {10, 30, 80, 45, 60, 50,110, 100, 130, 170}, x = 110;
Output: 6
Explanation: Element x is present at index 6

Input: arr[] = {10, 30, 80, 30, 60, 50,120, 100, 150, 170}, x = 175;
Output: -1
Explanation: Element x is not present in arr[].

Follow the below idea to solve the problem:

Iterate from 0 to N-1 and compare the value of every index with x if they match return index

Follow the given steps to solve the problem:

1. Set the first element of the array as the current element.
2. If the current element is the target element, return its index.
2. If the current element is not the target element and if there are more elements in the array, set the current element to the next element and repeat step 2.
3.If the current element is not the target element and there are no more elements in the array, return -1 to indicate that the element was not found.

```
<script>

// Javascript code to linearly search x in arr[]. If x
// is present then return its location, otherwise
// return -1

function search(arr, n, x)
{
	let i;
	for (i = 0; i < n; i++)
		if (arr[i] == x)
			return i;
	return -1;
}


// Driver code

	let arr = [ 2, 3, 4, 10, 40 ];
	let x = 10;
	let n = arr.length;

	// Function call
	let result = search(arr, n, x);
	(result == -1)
		? document.write("Element is not present in array")
		: document.write("Element is present at index " + result);

================================================================================

// JavaScript Recursive Code For Linear Search

let linearsearch = (arr, size, key) => {
if (size == 0) {
	return -1;
}
else if (arr[size - 1] == key)
{

	// Return the index of found key.
	return size - 1;
}
else
{
	let ans = linearsearch(arr, size - 1, key);
	return ans;
}
};
```

```
// Driver Code
let main = () => {
let arr = [5, 15, 6, 9, 4];
let key = 4;

let ans = linearsearch(arr, 5, key);
if (ans == -1) {
        console.log(`The element ${key} is not found.`);
} else {
        console.log(
        `The element ${key} is found at ${ans} index of the given array.`
        );
}
return 0;
};

main();
```

Longest Common Subsequence

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

If `S1` and `S2` are the two given sequences then, `Z` is the common subsequence of `S1` and `S2` if `Z` is a subsequence of both `S1` and `S2`. Furthermore, `Z` must be a **strictly increasing sequence** of the indices of both `S1` and `S2`.

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in `Z`.

If

```
S1 = {B, C, D, A, A, C, D}
```

Then, `{A, D, B}` cannot be a subsequence of `S1` as the order of the elements is not the same (ie. not strictly increasing sequence).

Let us understand LCS with an example.

If

```
S1 = {B, C, D, A, A, C, D}
S2 = {A, C, D, B, A, C}
```

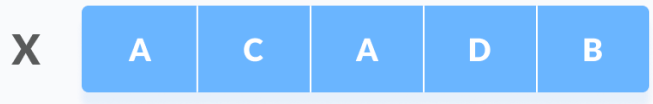
Then, common subsequences are `{B, C}`, `{C, D, A, C}`, `{D, A, C}`, `{A, A, C}`, `{A, C}`, `{C, D}`, `...`

Among these subsequences, `{C, D, A, C}` is the longest common subsequence. We are going to find this longest common subsequence using dynamic programming.

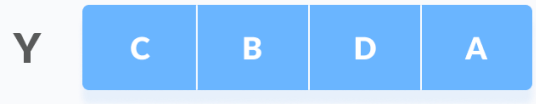
Before proceeding further, if you do not already know about dynamic programming, please go through dynamic programming.

Using Dynamic Programming to find the LCS

Let us take two sequences:



The first sequence

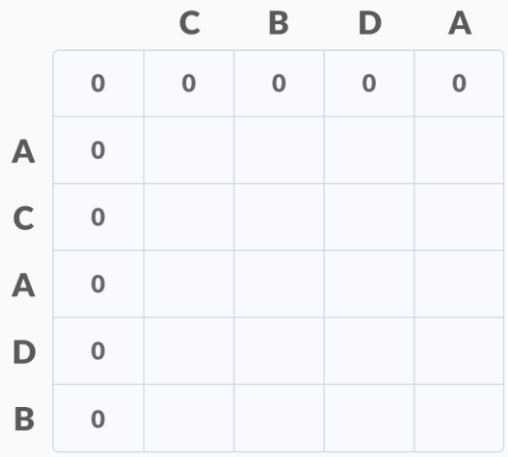


Second Sequence

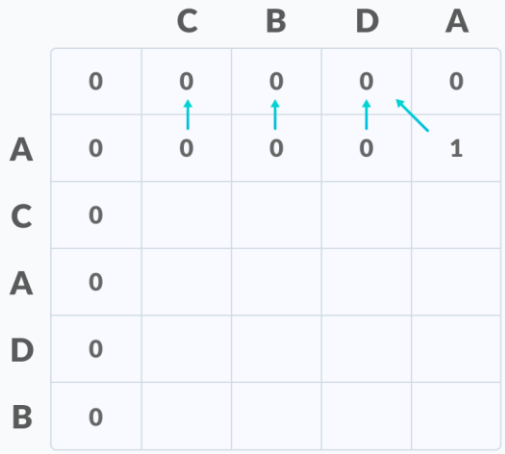
The following steps are followed for finding the longest common subsequence.

1. Create a table of dimension `n+1*m+1` where n and m are the lengths of `X` and `Y` respectively. The first row and the first column are filled with zeros.

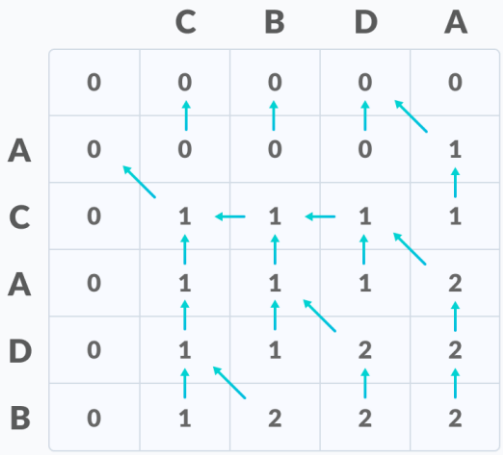
   Initialise a table



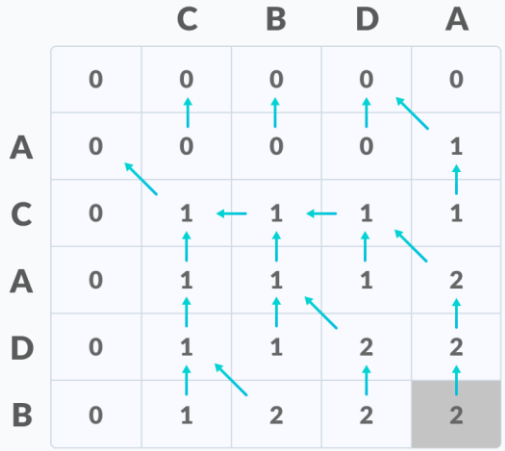
2. Fill each cell of the table using the following logic.
3. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.



Fill the values

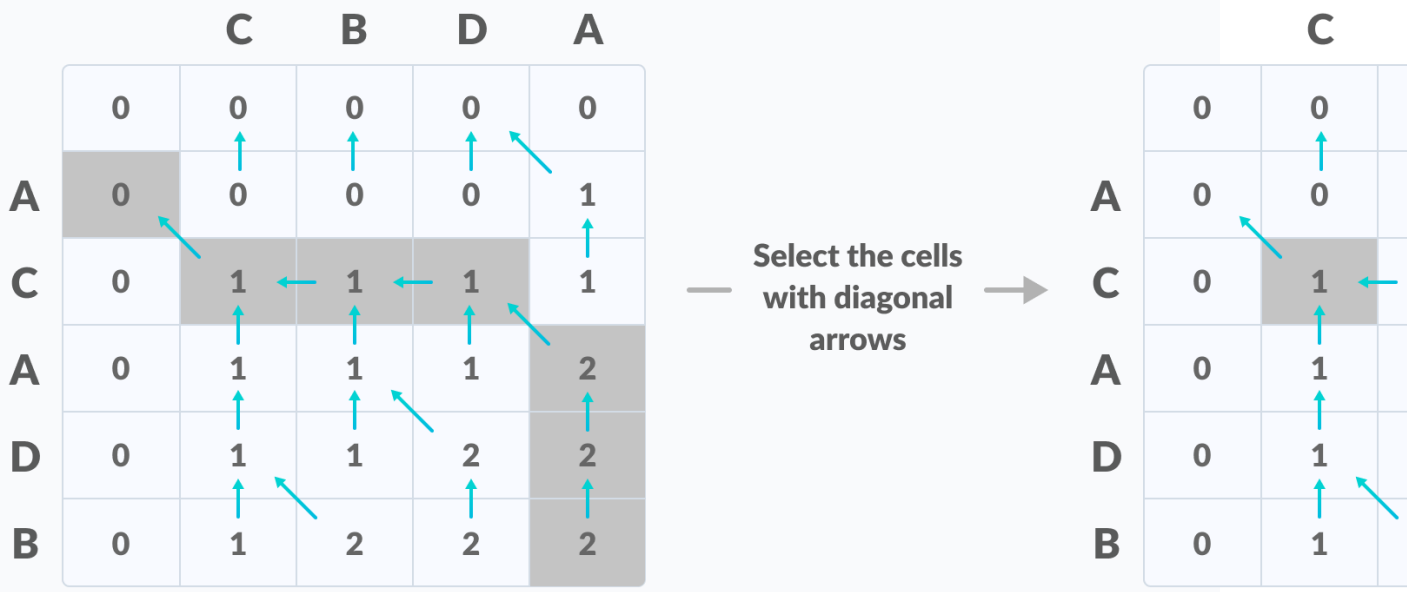


5. **Step 2** is repeated until the table is filled. Fill all the values
6. The value in the last row and the last column is the length of the longest common subsequence.



The bottom right corner is the length of the LCS

7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.



Create a path according to the arrows

Thus, the longest common subsequence is `CA`.

LCS

How is a dynamic programming algorithm more efficient than the recursive algorithm while solving an LCS problem?

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of `X` and the elements of `Y` are stored in a table so that they can be used in future computations.

So, the time taken by a dynamic approach is the time taken to fill the table (ie. O(mn)). Whereas, the recursion algorithm has the complexity of $2^{max(m, n)}$.

Longest Common Subsequence Algorithm

```
X and Y be two given sequences
Initialize a table LCS of dimension X.length * Y.length
X.label = X
Y.label = Y
LCS[0][] = 0
LCS[][0] = 0
Start from LCS[1][1]
Compare X[i] and Y[j]
    If X[i] = Y[j]
        LCS[i][j] = 1 + LCS[i-1, j-1]
        Point an arrow to LCS[i][j]
    Else
        LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
        Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

Python, Java and C/C++ Examples

Python
Java
C
C++

```
# The longest common subsequence in Python



# Function to find lcs_algo
def lcs_algo(S1, S2, m, n):
    L = [[0 for x in range(n+1)] for x in range(m+1)]

    # Building the mtrix in bottom-up way
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0:
                L[i][j] = 0
            elif S1[i-1] == S2[j-1]:
                L[i][j] = L[i-1][j-1] + 1
            else:
                L[i][j] = max(L[i-1][j], L[i][j-1])
```

```
    index = L[m][n]

    lcs_algo = [""] * (index+1)
    lcs_algo[index] = ""

    i = m
    j = n
    while i > 0 and j > 0:

        if S1[i-1] == S2[j-1]:
            lcs_algo[index-1] = S1[i-1]
            i -= 1
            j -= 1
            index -= 1

        elif L[i-1][j] > L[i][j-1]:
            i -= 1
        else:
            j -= 1

    # Printing the sub sequences
    print("S1 : " + S1 + "\nS2 : " + S2)
    print("LCS: " + "".join(lcs_algo))


S1 = "ACADB"
S2 = "CBDA"
m = len(S1)
n = len(S2)
lcs_algo(S1, S2, m, n)
```

Longest Common Subsequence Applications

1. in compressing genome resequencing data
2. to authenticate users within their mobile phone through in-air signatures

- ## 0/1 Knapsack Problem

Unbounded Knapsack (Repetition of items allowed)

Given a knapsack weight **W** and a set of **n** items with certain value val_i and weight wt_i, we need to calculate the maximum amount that could make up this quantity exactly. This is different from classical Knapsack problem, here we are allowed to use unlimited number of instances of an item.

Examples:

```
Input : W = 100
       val[]  = {1, 30}
       wt[] = {1, 50}
Output : 100
There are many ways to fill knapsack.
1) 2 instances of 50 unit weight item.
2) 100 instances of 1 unit weight item.
3) 1 instance of 50 unit weight item and 50
   instances of 1 unit weight items.
We get maximum value with option 2.

Input : W = 8
       val[] = {10, 40, 50, 70}
       wt[]  = {1, 3, 4, 5}
Output : 110
We get maximum value with one unit of
weight 5 and one unit of weight 3.
```

Knapsack with Duplicate Items

Recursive Approach:

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

Optimal Sub-structure: To consider all subsets of items, there can be two cases for every item.

Case 1: The item is included in the optimal subset.

Case 2: The item is not included in the optimal set.

Therefore, the maximum value that can be obtained from 'n' items is the max of the following two values.

Maximum value obtained by n-1 items and W weight (excluding nth item).

Value of nth item plus maximum value obtained by n(because of infinite supply) items and W minus the weight of the nth item (including nth item).

If the weight of 'nth' item is greater than 'W', then the nth item cannot be included and Case 1 is the only possibility.

Below is the implementation of the above approach:

```
/* A Naive recursive implementation of
unbounded Knapsack problem */

// Returns the maximum value that
// can be put in a knapsack of capacity W
function unboundedKnapsack(W, wt, val, idx)
{

    // Base Case
    // if we are at idx 0.
    if (idx == 0) {
    return Math.floor(W / wt[0]) * val[0];
```

```
        }

        // There are two cases either take element or not take.
        // If not take then
        var notTake = 0 + unboundedKnapsack(W, wt, val, idx - 1);

        // if take then weight = W-wt[idx] and index will remain
        // same.
        var take = Number.MIN_VALUE;
        if (wt[idx] <= W) {
        take = val[idx] + unboundedKnapsack(W - wt[idx], wt, val, idx);
        }
        return Math.max(take, notTake);
}

// Driver code
var W = 100;
var val = [10, 30, 20];
var wt = [5, 10, 15];
var n = val.length;

console.log(unboundedKnapsack(W, wt, val, n - 1));

// This code is contributed by satwiksuman.
```

You are given weights and values of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. Note that we have only one quantity of each item.
In other words, given two integer arrays val[0..N-1] and wt[0..N-1] which represent values and weights associated with N items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or dont pick it (0-1 property).

Example 1:

Input:
N = 3
W = 4
values[] = {1,2,3}
weight[] = {4,5,1}
Output: 3
Explanation: Choose the last item that weighs 1 unit and holds a value of 3.
Example 2:

Input:
N = 3
W = 3
values[] = {1,2,3}
weight[] = {4,5,6}
Output: 0
Explanation: Every item has a weight exceeding the knapsack's capacity (3).
Your Task:
Complete the function knapSack() which takes maximum capacity W, weight array wt[], value array val[], and the number of items n as a parameter and returns the maximum possible value you can get.

Expected Time Complexity: O(N*W).
Expected Auxiliary Space: O(N*W)

Constraints:
1 ≤ N ≤ 1000
1 ≤ W ≤ 1000
1 ≤ wt[i] ≤ 1000
1 ≤ v[i] ≤ 1000

The Knight's tour problem

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that "works". Problems that are typically solved using the backtracking technique have the following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once. A Naive solution for these problems is to try all configurations and output a configuration that follows given problem constraints. Backtracking works incrementally and is an optimization over the Naive solution where all possible configurations are generated and tried.
For example, consider the following Knight's Tour problem.

Problem Statement:
Given a N*N board with the Knight placed on the first block of an empty board. Moving according to the rules of chess knight must visit each square exactly once. Print the order of each cell in which they are visited.

Example:

```
Input :
N = 8
Output:
0  59  38  33  30  17   8  63
37  34  31  60   9  62  29  16
58   1  36  39  32  27  18   7
35  48  41  26  61  10  15  28
42  57   2  49  40  23   6  19
47  50  45  54  25  20  11  14
56  43  52   3  22  13  24   5
51  46  55  44  53   4  21  12
```

The path followed by Knight to cover all the cells
Following is a chessboard with 8 x 8 cells. Numbers in cells indicate the move number of Knight.

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

Let us first discuss the Naive algorithm for this problem and then the Backtracking algorithm.

Naive Algorithm for Knight's tour

The Naive Algorithm is to generate all tours one by one and check if the generated tour satisfies the constraints.

```
while there are untried tours
{
   generate the next tour
   if this tour covers all squares
   {
      print this path;
   }
}
```

Backtracking works in an incremental way to attack problems. Typically, we start from an empty solution vector and one by one add items (Meaning of item varies from problem to problem. In the context of Knight's tour problem, an item is a Knight's move). When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives works out then we go to the previous stage and remove the item added in the previous stage. If we reach the initial stage back then we say that no solution exists. If adding an item doesn't violate constraints then we recursively add items one by one. If the solution vector becomes complete then we print the solution.

Backtracking Algorithm for Knight's tour

Following is the Backtracking algorithm for Knight's tour problem.

```
If all squares are visited
    print the solution
Else
   a) Add one of the next moves to solution vector and recursively
   check if this move leads to a solution. (A Knight can make maximum
   eight moves. We choose one of the 8 moves in this step).
   b) If the move chosen in the above step doesn't lead to a solution
   then remove this move from the solution vector and try other
   alternative moves.
   c) If none of the alternatives work then return false (Returning false
   will remove the previously added item in recursion and if false is
   returned by the initial call of recursion then "no solution exists" )
```

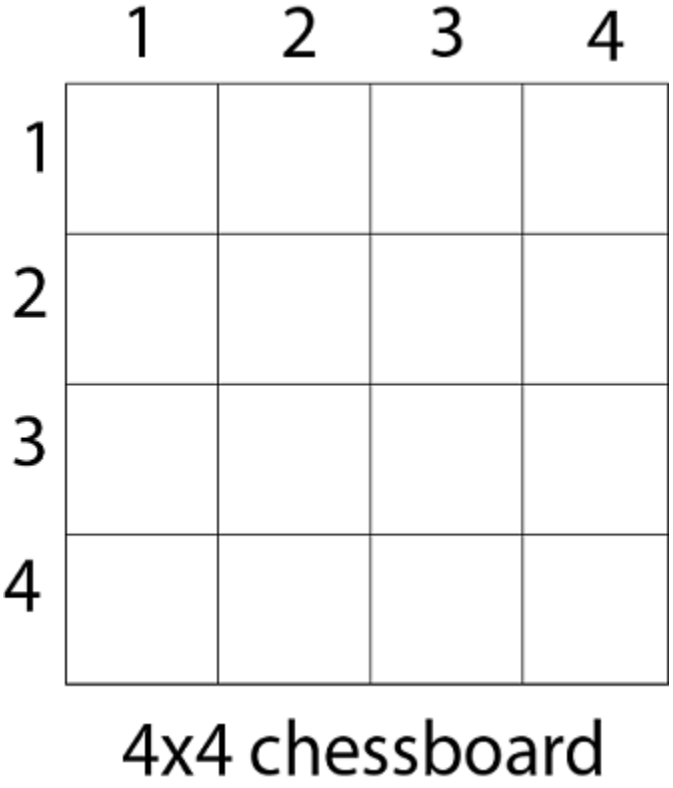
Following are implementations for Knight's tour problem. It prints one of the possible solutions in 2D matrix form. Basically, the output is a 2D 8*8 matrix with numbers from 0 to 63 and these numbers show steps made by Knight.

N-Queens Problem

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

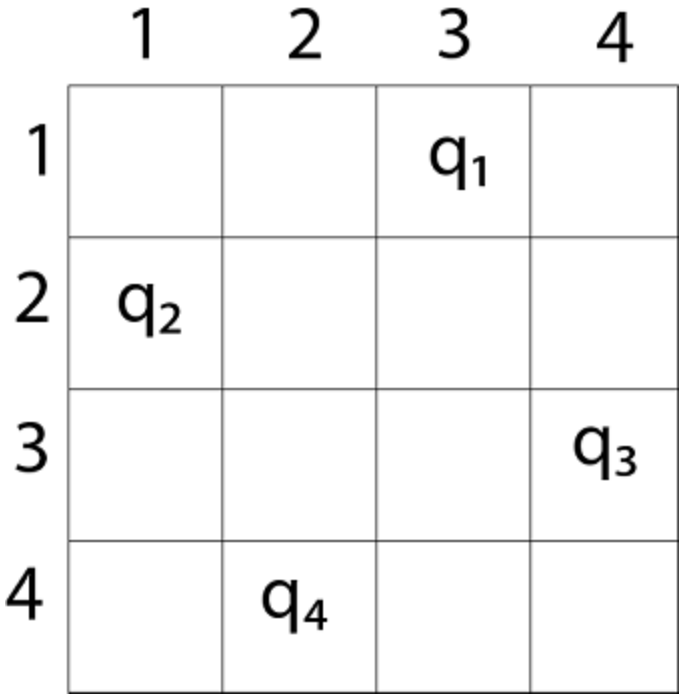
Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.



4x4 chessboard

Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.



The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

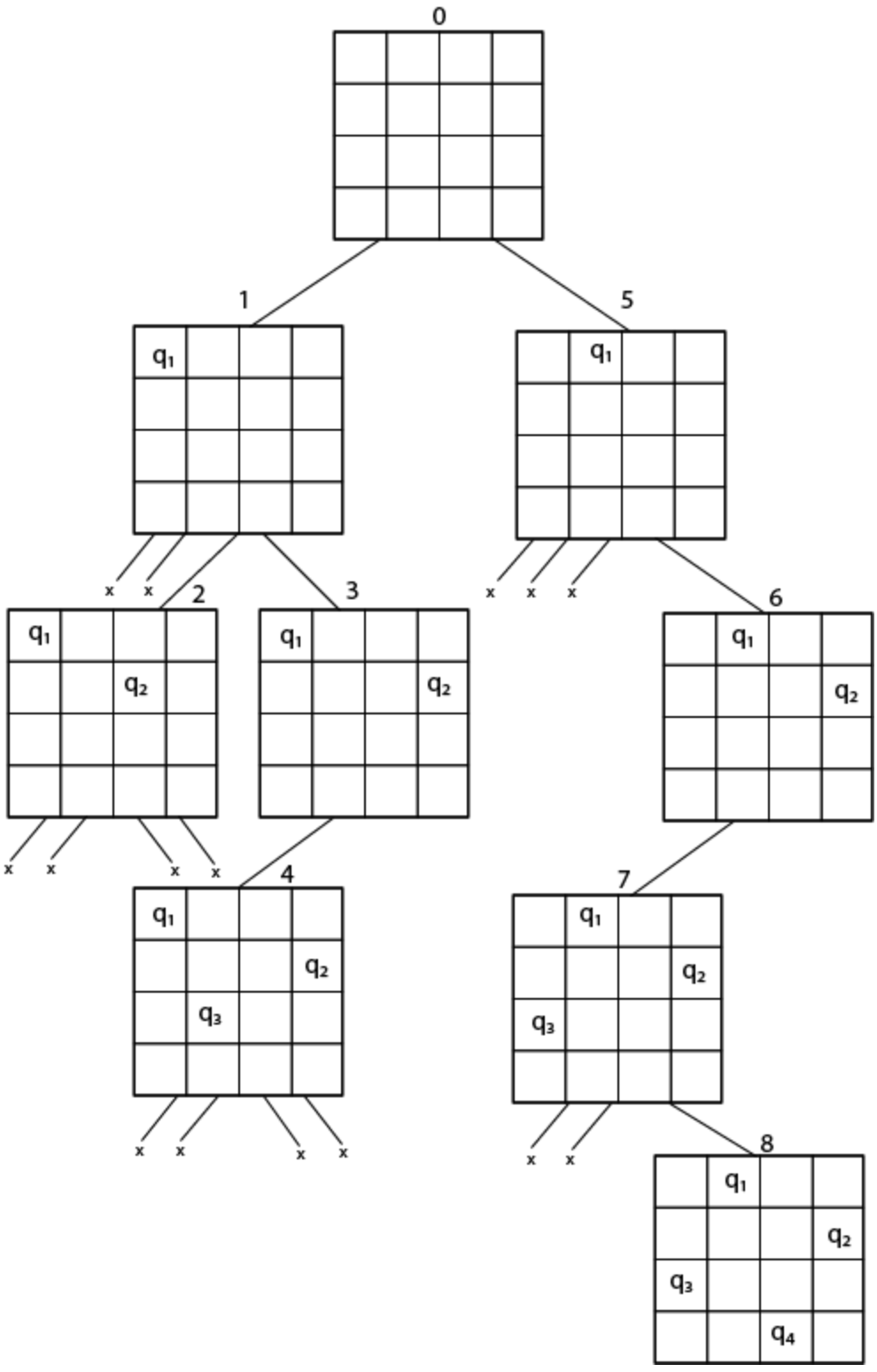
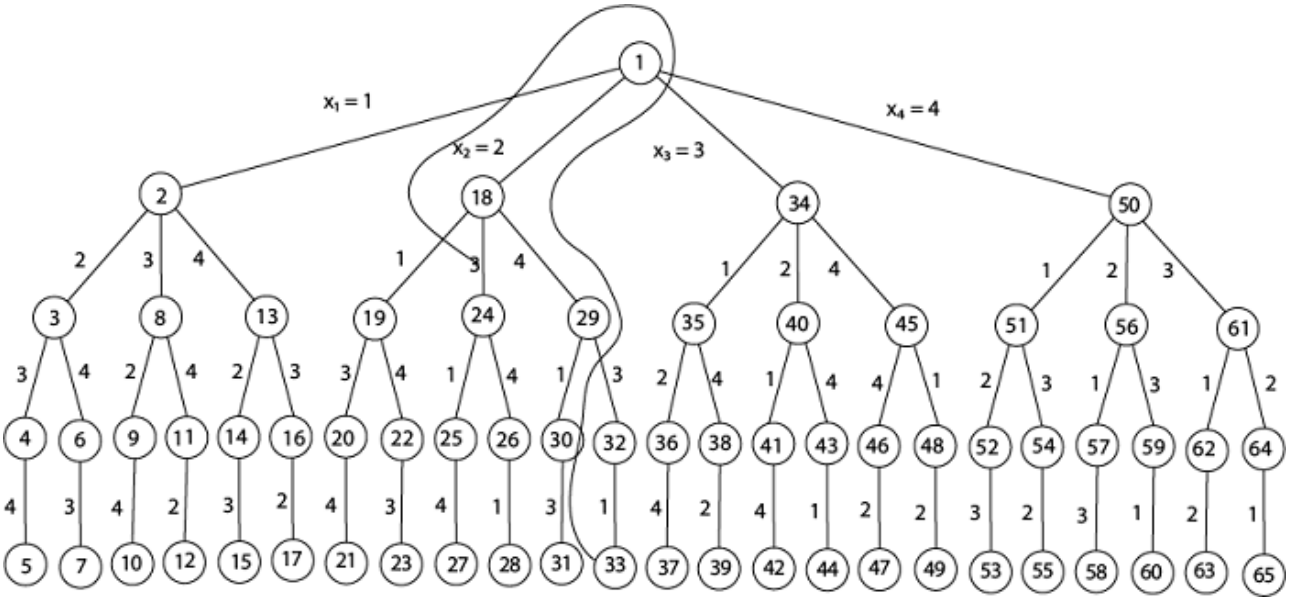


Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



4 - Queens solution space with nodes numbered in DFS

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in fig:

	1	2	3	4	5	6	7	8
1				q_1				
2						q_2		
3								q_3
4		q_4						
5							q_5	
6	q_6							
7			q_7					
8					q_8			

1. Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l).
3. Then they are on same diagonal only **if** (i - j) = k - l or i + j = k + l.

4. The first equation implies that j - l = i - k.
5. The second equation implies that j - l = k - i.
6. Therefore, two queens lie on the duplicate diagonal **if** and only **if** |j-l|=|i-k|

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2,....x_{k-1} and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to then n- queens problem.

```
1.  Place (k, i)
2.    {
3.      For j ← 1 to k - 1
4.       do if (x [j] = i)
5.        or (Abs x [j]) - i) = (Abs (j - k))
6.     then return false;
7.      return true;
8.  }
```

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.

x [] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

```
1.  N - Queens (k, n)
2.  {
3.    For i ← 1 to n
4.        do if Place (k, i) then
5.    {
6.      x [k] ← i;
7.      if (k ==n) then
8.        write (x [1....n));
9.      else
10.     N - Queens (k + 1, n);
11.   }
12. }
```

Power Set: Power set **P(S)** of a set **S** is the set of all subsets of **S**. For example S = {a, b, c} then P(s) = {{}, {a}, {b}, {c}, {a,b}, {a, c}, {b, c}, {a, b, c}}.
If **S** has **n** elements in it then **P(s)** will have 2^n elements

Example:

Set = [a,b,c]
power_set_size = pow(2, 3) = 8
Run for binary counter = 000 to 111

Value of Counter Subset
000 -> Empty set
001 -> a
010 -> b
011 -> ab
100 -> c
101 -> ac
110 -> bc
111 -> abc

Recommended Problem
Power Set

Algorithm:
Input: Set[], set_size
1. Get the size of power set
powet_set_size = pow(2, set_size)
2 Loop for counter from 0 to pow_set_size
(a) Loop for i = 0 to set_size
(i) If ith bit in counter is set
Print ith element from set for this subset
(b) Print separator for subsets i.e., newline

Method 1:
For a given set[] S, the power set can be found by generating all binary numbers between **0** and 2^n**-1**, where **n** is the size of the set.
For example, for the set **S {*x, y, z*}**, generate all binary numbers from **0** to 2^3**-1** and for each generated number, the corresponding set can be found by considering set bits in the number.
Below is the implementation of the above approach.

```
// javascript program for power setpublic
    function printPowerSet(set, set_size)
    {
        /*
        * set_size of power set of a set with set_size n is (2**n -1)
        */
        var pow_set_size = parseInt(Math.pow(2, set_size));
        var counter, j;
        /*
        * Run from counter 000..0 to 111..1
        */
        for (counter = 0; counter < pow_set_size; counter++)
        {
            for (j = 0; j < set_size; j++)
            {
                        /*
```

```
                 * Check if jth bit in the counter is set If set then print jth element from set
                 */
                if ((counter & (1 << j)) > 0)
                    document.write(set[j]);
            }
            document.write("<br/>");
        }
    }

    // Driver program to test printPowerSet
        let set = [ 'a', 'b', 'c' ];
        printPowerSet(set, 3);
```

Learn Data Structures & Algorithms with GeeksforGeeks

Output

```
a
b
ab
c
ac
bc
abc
```

Time **Complexity:** $O(n2^n)$

Auxiliary Space: O(1)

Method 2: (sorted by cardinality)

- In auxiliary array of bool set all elements to 0. That represent an empty set. Set first element of auxiliary array to 1 and generate all permutations to produce all subsets with one element. Then set the second element to 1 which will produce all subsets with two elements, repeat until all

```
// C++ program for the above approach
#include <bits/stdc++.h>
using namespace std;
  // Function to print all the power set
void printPowerSet(char set[], int n)
{
    bool *contain = new bool[n]{0};
          // Empty subset
    cout << "" << endl;
    for(int i = 0; i < n; i++)
    {
        contain[i] = 1;
        // All permutation
        do
        {
            for(int j = 0; j < n; j++)
                if(contain[j])
                    cout << set[j];
            cout << endl;
        } while(prev_permutation(contain, contain + n));
    }
}
  /*Driver code*/
int main()
{
```

```
    char set[] = {'a','b','c'};
    printPowerSet(set, 3);
    return 0;
}
```

Output
a
b
c
ab
ac
bc
abc
Time **Complexity:** $O(n2^n)$
Auxiliary Space: O(n)
Method 3:
We can use backtrack here, we have two choices first consider that element then don't consider that element.
Below is the implementation of the above approach.

```
#include <bits/stdc++.h>
using namespace std;

void findPowerSet(char* s, vector<char> &res, int n){
        if (n == 0) {
            for (auto i: res)
              cout << i;
            cout << "\n";
            return;

        }
         res.push_back(s[n - 1]);
         findPowerSet(s, res, n - 1);
         res.pop_back();
         findPowerSet(s, res, n - 1);
    }

void printPowerSet(char* s, int n){
    vector<char> ans;
    findPowerSet(s, ans, n);
}

int main()
{
    char set[] = { 'a', 'b', 'c' };
    printPowerSet(set, 3);
    return 0;
}
```

Output
cba
cb
ca
c
ba

```
b
a
Time                                                    Complexity: O(2^n)
Auxiliary Space: O(n)
```

Heap and Hash : An Overview

Heap:

1. Introduction to Heap:

Heap is a tree-based data structure, and a **complete binary tree** is used for the creation and implementation of a heap.

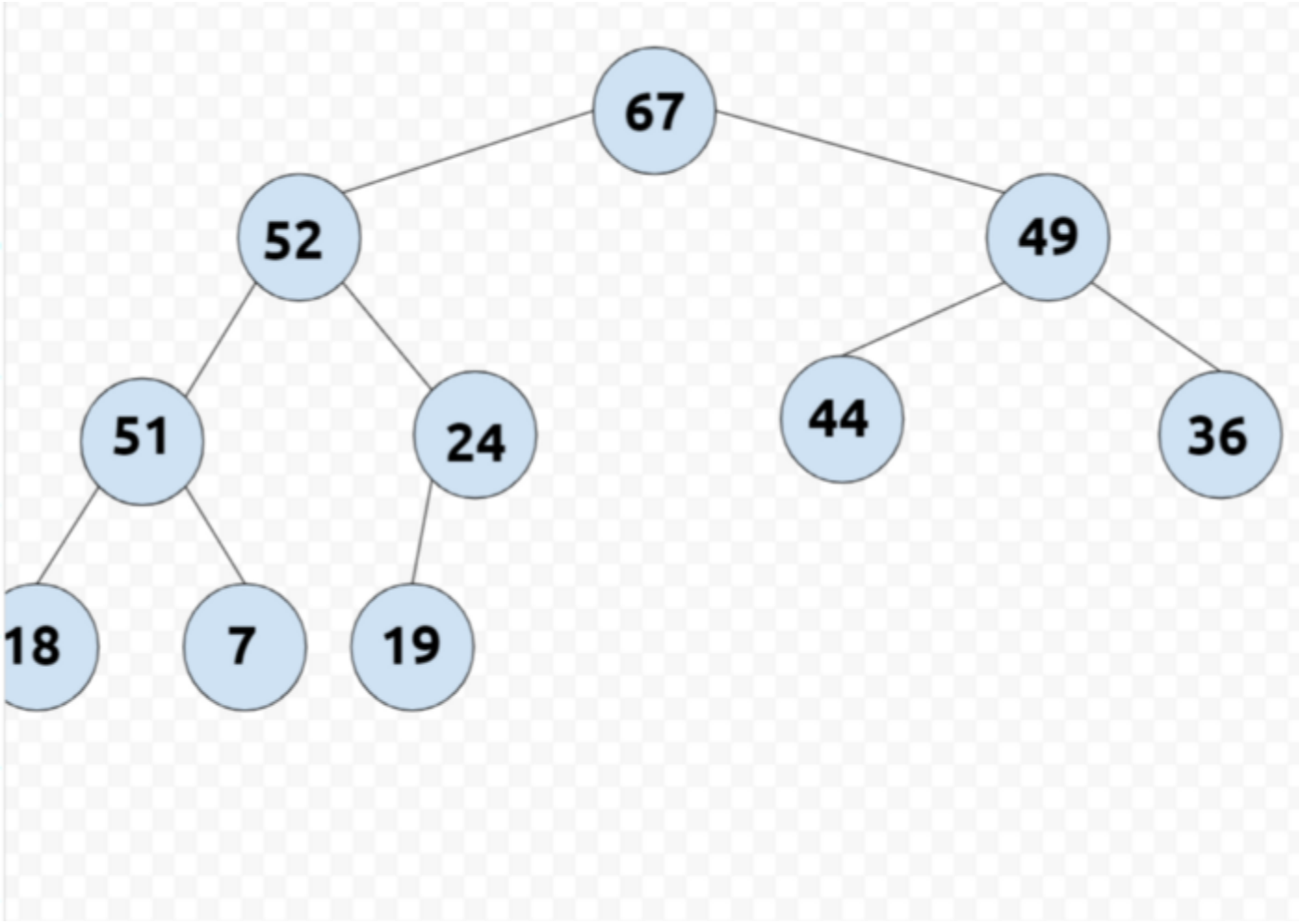
2. Properties of a Heap:

- Heap is a complete binary tree.
- The root node is located in **H[0].**
- **H[(i-1)/2]** returns the parent node.
- **H[(2*i)+1]** returns the left child node.
- **H[(2*i)+2]** returns the right child node.

3. Types of Heap:

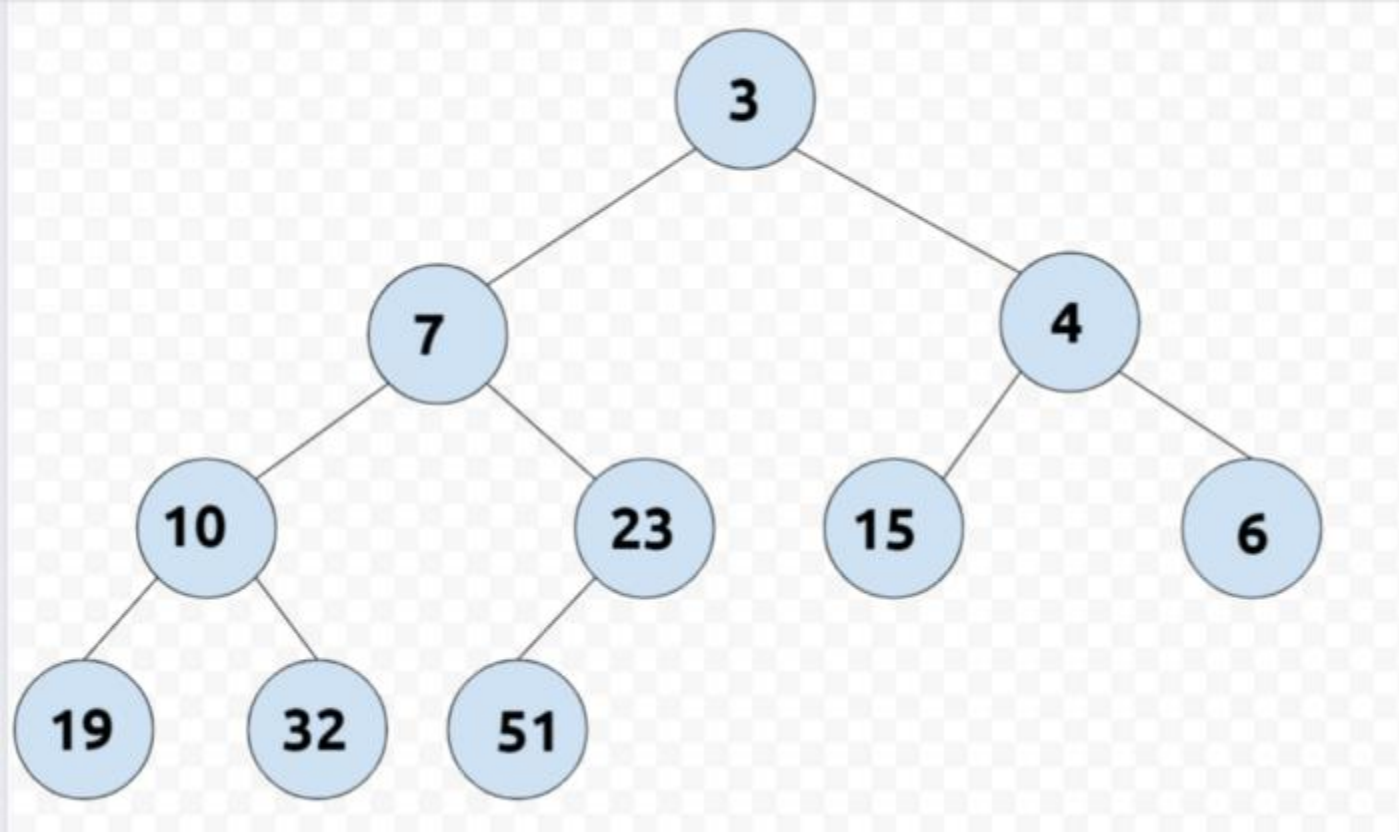
Mainly there are two types of Heap namely:

Max-Heap: A type of heap where the value of the parent node's values is always greater than its children. This property must be recursively true for all sub-trees in that Binary Tree. The root node consists of the highest value of the heap.



Max- Heap

Min-Heap: A type of heap where the value of the parent node's values is always lesser than its children. This property must be recursively true for all subtrees in that Binary Tree. The root node consists of the minimum value of the heap.



Min — Heap

4. Insertion in a Heap

The insertion into a heap is an easy process. While inserting new elements into heap we need to maintain the heap property. The following is the algorithm for insertion:

1. Size of the heap must be increased by 1 to accommodate the new element.
2. Insert the element at the end of the heap, so that the resulting heap so formed, is a complete binary tree.
3. Compare the value of the element with its parent and swap **if** the value of the parent is less than the element in case of Max Heap. Value of the parent is more than the element in case of Min Heap.
4. Continue comparing and swapping until the above condition fails.

5. Deletion from a Heap

1. The standard protocol that has to be followed while deleting elements from heap is that the element present in the root node can only be deleted.
2. We must shift the entire tree upwards.
3. Delete the root node from the tree and copy the last element of the heap to the root.

4. Compare the current root node with its children and if: Root is greater than any of its children, replace the root with least value of the children in case of Min Heap. Root is lesser than any of its children , replace the root with the greatest value of the children in case of Max Heap.
5. Continue comparing and swapping until the above condition fails.

6. Heap Sort:

The technique followed in heap sort is deletion of elements in a heap.

- In case of Max Heap, the deleted nodes will be of decreasing order as discussed above, therefore the resultant array so formed will have elements in decreasing order.
- In case of Min Heap, the deleted nodes will be of increasing order as discussed above, therefore the resultant array so formed will have elements in increasing order.

Hashing Techniques:

Hashing is a technique or process of mapping keys, values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used

Types of Hash Function

1. **Ideal Hash Function:** h(x) = x where x is the element to be mapped.
2. **Standard Modulus Function:** h(x) = x%10 where x is the element to be mapped.
3. **Custom Random Function:** h(x) = <DEFINE_YOUR_FUNCTION>

What are Collisions?

A state in a hash table when more than one key is mapped at a specific index. A hash table allows to store only one element per index. A newly inserted key mapped to an already occupied index results in a collision.

Resolve collisions in Hashing:

There are two methods to resolve collisions while mapping elements into a hash table. They are:

1. Open Hashing:

To resolve collisions, the computer might use extra storage space along with the hash table size. It uses an array of linked-lists to resolve collisions.

In open hashing the element to be mapped is not stored in a hash table, rather it's stored in a linked list external to the hash table memory. The word open refers to freedom of mapping values in an external data structure without using the hash memory table.

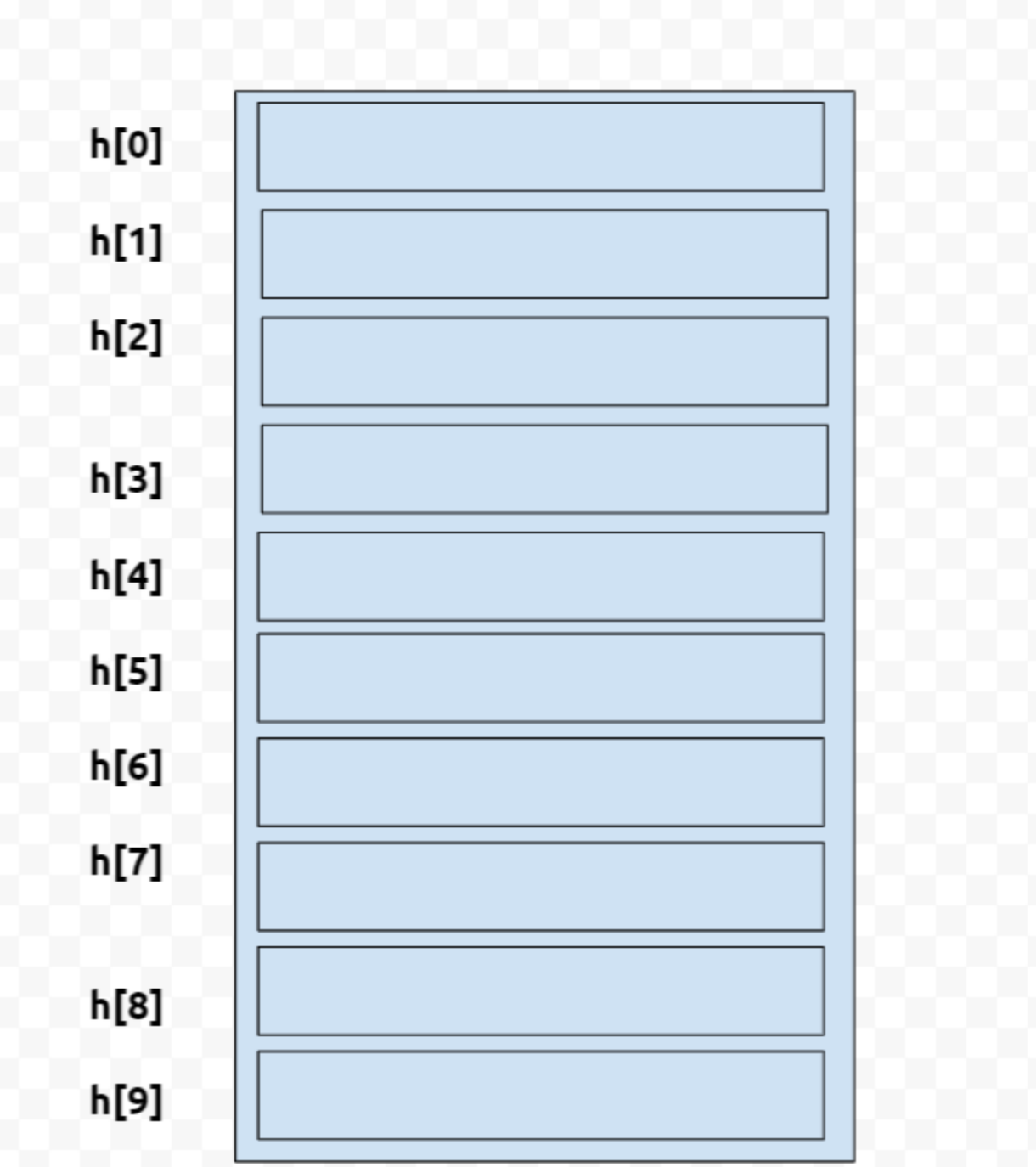
Open hashing uses a technique of chaining to resolve collisions. All key-value pairs mapping to the same index will be stored in the linked list of that index.

Ex- Consider the following elements:

9, 24, 37, 58, 99, 114, 109, 64, 88

Hash Function: h(x) = x%10.

Initial Hash Table:



A Hash Table Representation

Function Values:

h(9) = 9%10 = 9

h(24) = 24 %10 = 4

h(37) = 37%10 = 7

h(99) = 99%10 =9

h(58) = 58%10=8

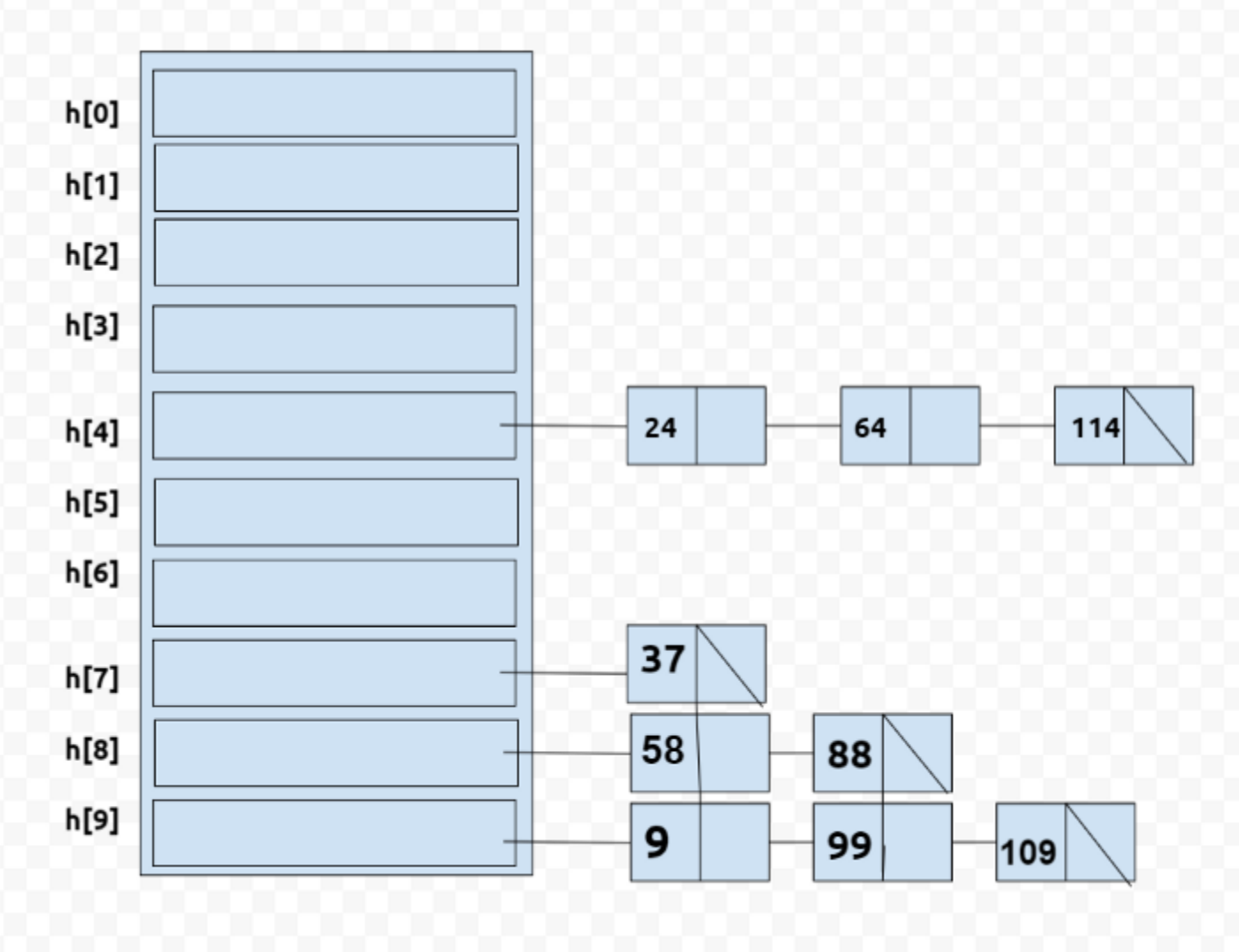
h(114) = 114 %10 =4

h(109) = 109%10=9

h(64) = 64 % 10 =4

h(88) = 88 %10 =8

Final Table after inserting values:



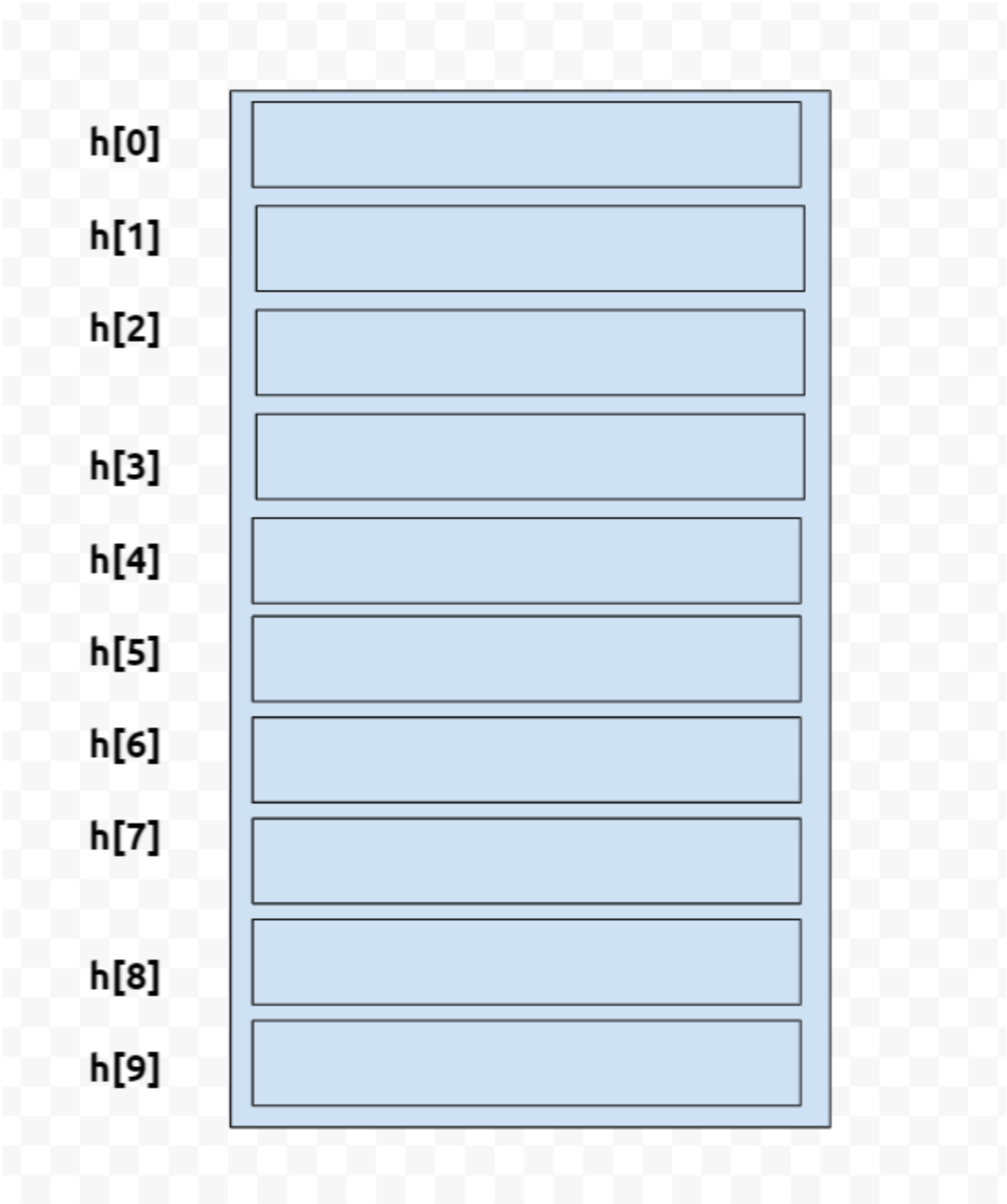
An example for Chaining

2. Closed Hashing:

1. **Linear Probing:**

Consider an example: 36, 20, 25, 43, 65, 74, 33, 87

Initial Hash Table:



A Hash Table Representation

Modified Hash Function:

- Initially we use the standard modulus function to insert or map values into the hash table.
- When a collision occurs while mapping in the table we use the modified hash function iteratively until an empty index or slot is found.
- **h'(x) = [h(x) + f(i)] %10** where f(i) = i; i = 0, 1, 2, 3, 4, 5…

Hash Functional Values: 36, 20, 25, 43, 65, 74, 33, 99

h'(36)=6

h'(20)=0

h'(25)=5

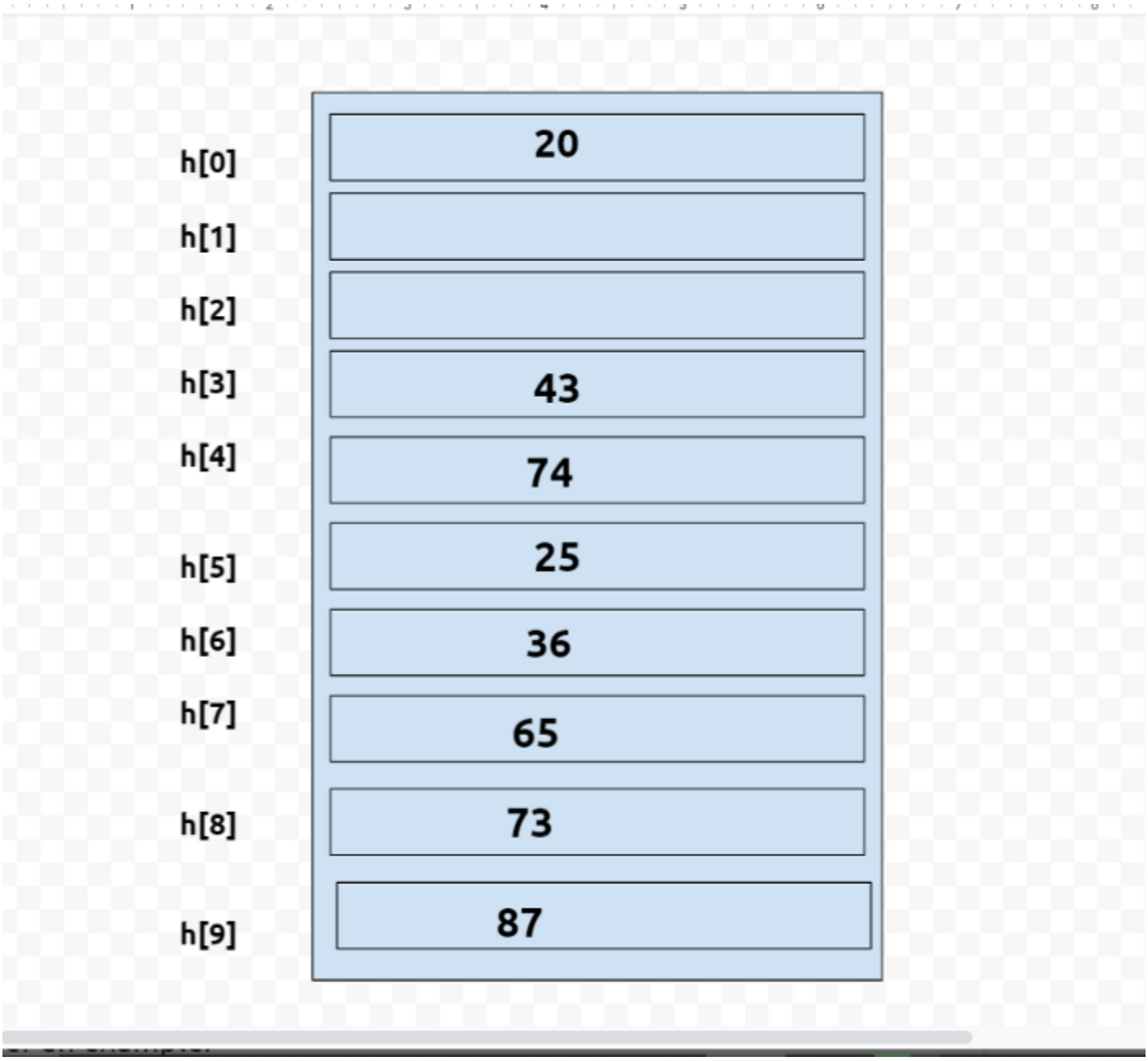
h'(43)=3

h'(65)=5 (C)

h'(74)=4

h'(33)=3(C) h'(87)=[C]

Table after insertion:



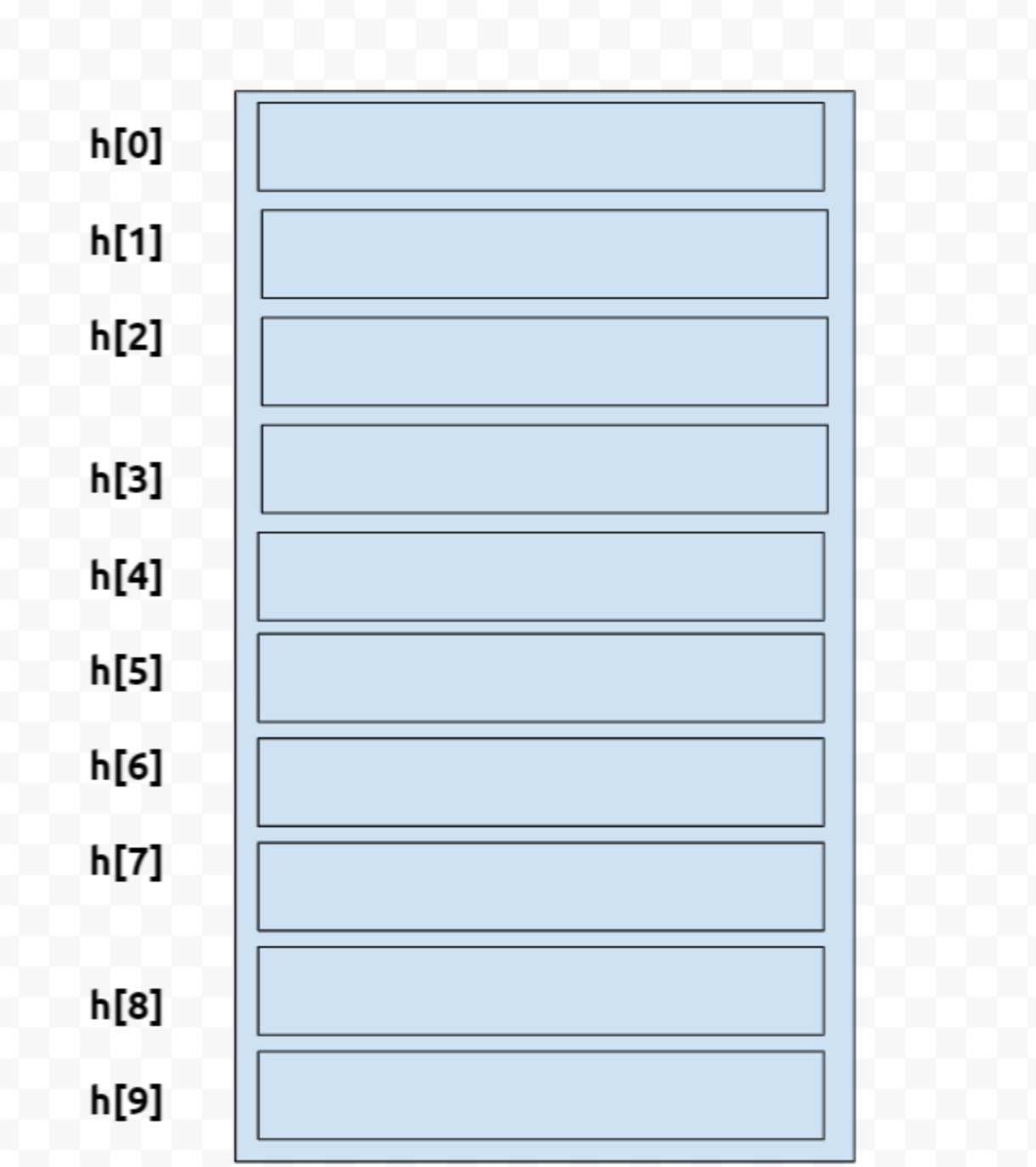
Linear Probing

2. Quadratic Probing:

Consider an example:

36, 20, 25, 43, 65, 74, 33,99

Initial Hash Table:



A Hash Table Representation

Modified Hash Function:

- Initially we use the standard modulus function to insert or map values into the hash table.

- When a collision occurs while mapping in the table we use the modified hash function iteratively until an empty index or slot is found.
- **h'(x) = [h(x) + f(i)] %10** where $f(i) = i^2$; i = 0, 1, 2, 3, 4, 5…

Hash Functional Values:

h'(36)= 6

h'(20)= 0

h'(25)= 5

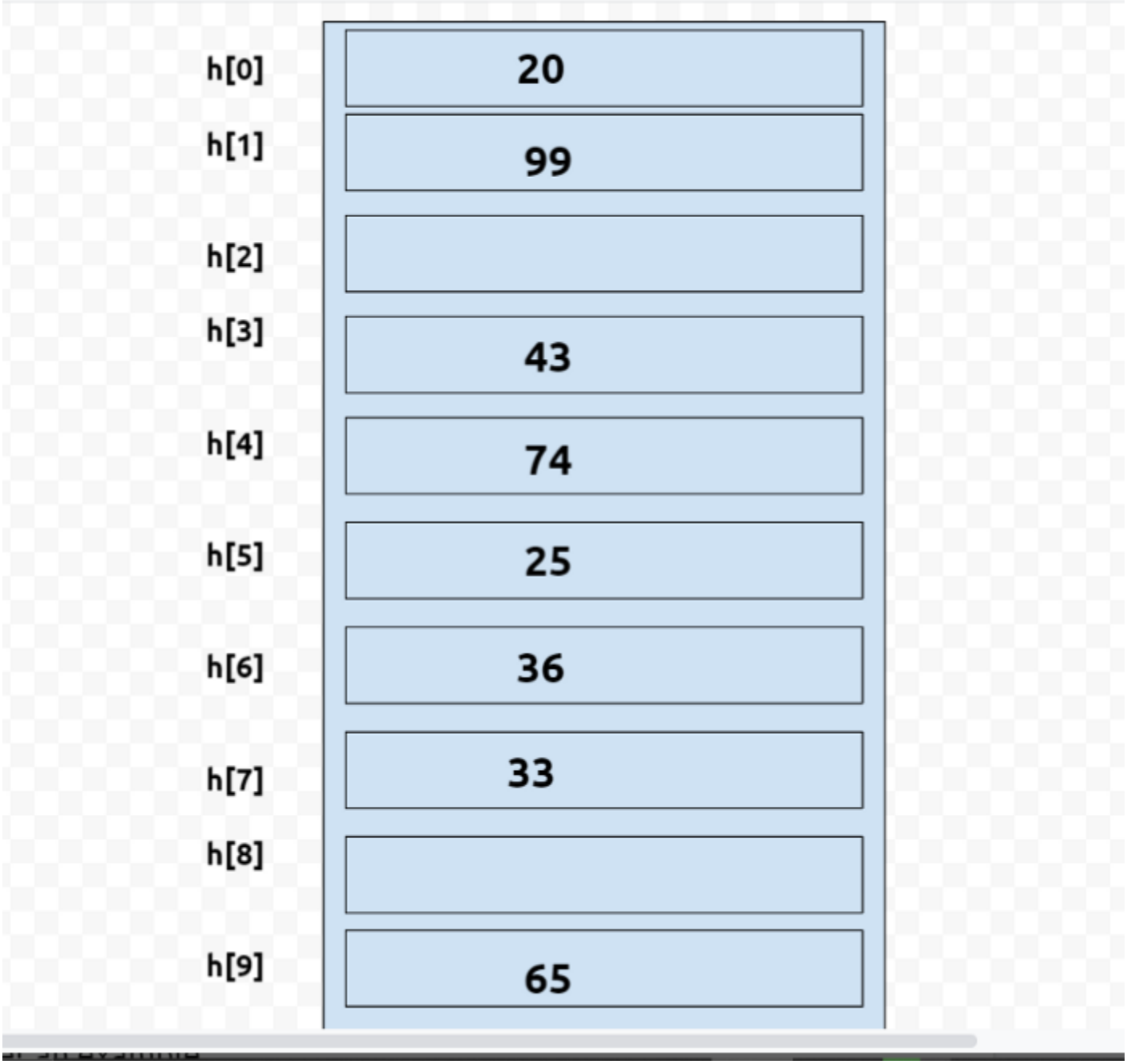
h'(43)= 3

h'(65)= 9 (C)

h'(74)= 4

h'(33)= 7(C) h'(99) = 1 (C)

Table after insertion:



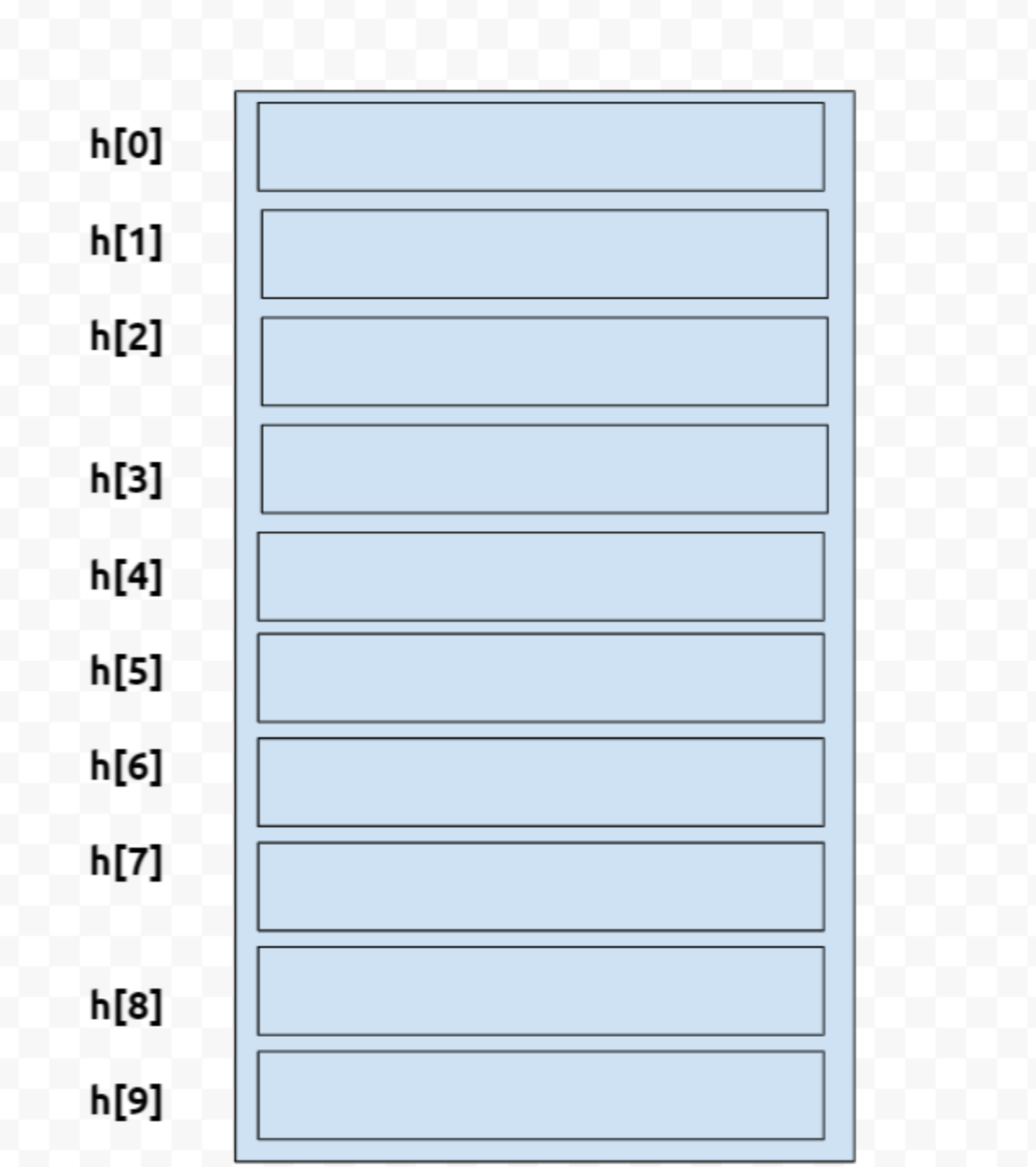
Quadratic Probing

3. Double Hashing:

Consider an example:

5, 25, 15, 35,95

Initial Hash Table:



A Hash Table Representation

Modified Hash Function:

- Initially we use the standard modulus function to insert or map values into the hash table.

- When a collision occurs while mapping in the table we use the modified hash function iteratively until an empty index or slot is found.
- h1(x) = x % 10
- h2(x) = R — (x % R) where R = closest prime number less than size of hash table.
- **h'(x) = [h1(x) + i*h2(x)] %10** where f(i) = i; i = 0, 1, 2, 3, 4, 5…

Hash Functional Values:

h'(5)=5

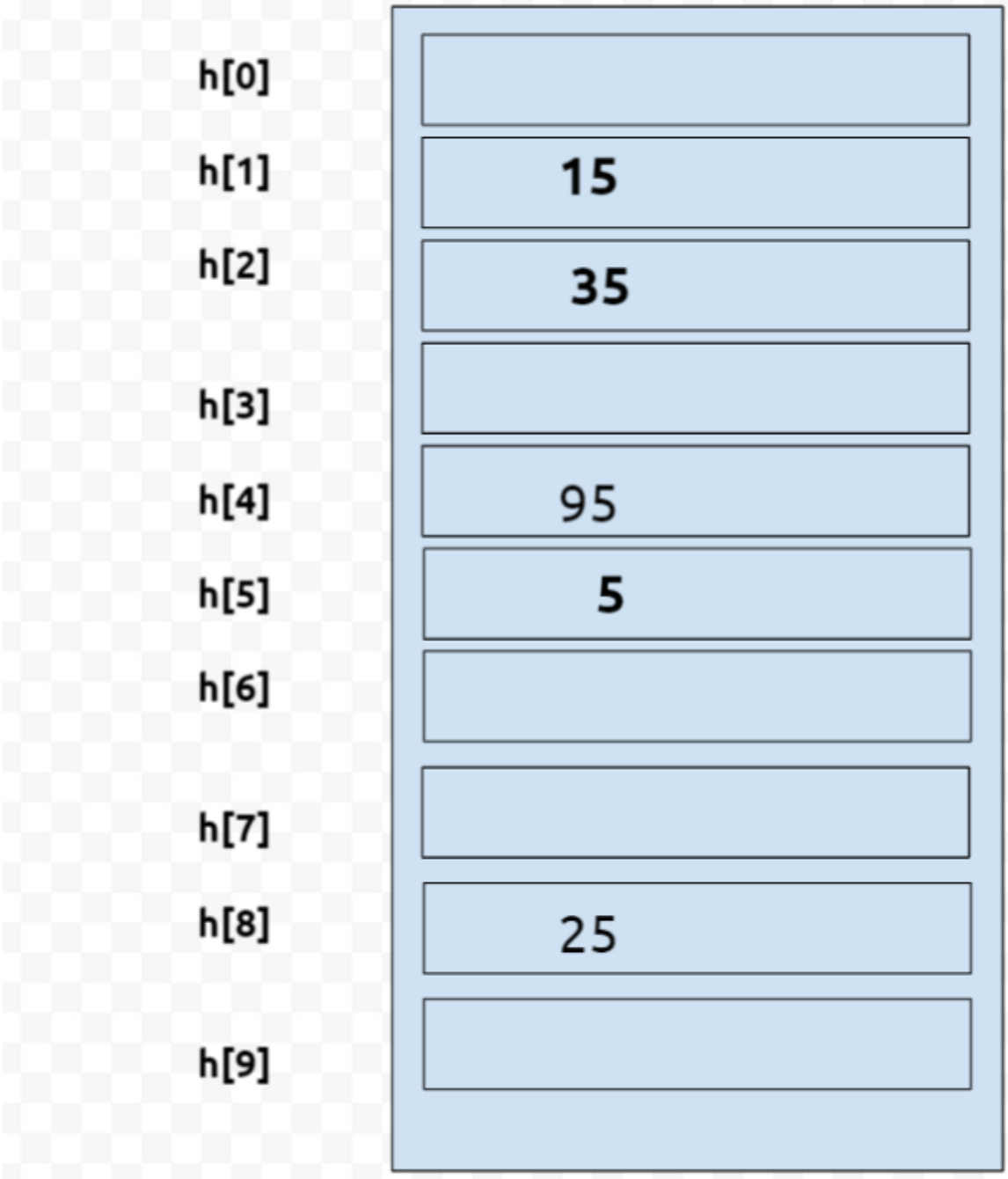
h'(25)=8

h'(15)=1

h'(35)=2

h'(95)=4

Table after insertion:



Double Hashing

Knapsack Problem :

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either ***0*** or ***1***, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example-1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	**A**	**B**	**C**	**D**
Profit	24	18	18	10
Weight	24	10	10	7

Without considering the profit per unit weight (p_i/w_i), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute maximum profit among all the elements.

After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is ***24***. Whereas, the optimal solution can be achieved by selecting items, ***B*** and C, where the total profit is 18 + 18 = 36.

==========

Example-2

Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio p_i/w_i. Let us consider that the capacity of the knapsack is W = 60 and the items are as shown in the following table.

Item	**A**	**B**	**C**
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6

Using the Greedy approach, first item ***A*** is selected. Then, the next item ***B*** is chosen. Hence, the total profit is **100 + 280 = 380**. However, the optimal solution of this instance can be achieved by selecting items, ***B*** and ***C***, where the total profit is **280 + 120 = 400**.

Hence, it can be concluded that Greedy approach may not give an optimal solution.

To solve 0-1 Knapsack, Dynamic Programming approach is required.

Problem Statement

A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are ***n*** items and weight of **i**th item is w_i and the profit of selecting this item is p_i. What items should the thief take?

Dynamic-Programming Approach

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution ***S*** is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maximum weight **w**.

The algorithm takes the following inputs

- The maximum weight **W**
- The number of items **n**
- The two sequences $\mathbf{v} = \langle v_1, v_2, \ldots, v_n \rangle$ and $\mathbf{w} = \langle w_1, w_2, \ldots, w_n \rangle$

Dynamic-0-1-knapsack (v, w, n, W)

```
for w = 0 to W do
  c[0, w] = 0
for i = 1 to n do
  c[i, 0] = 0
  for w = 1 to W do
    if wi ≤ w then
      if vi + c[i-1, w-wi] then
        c[i, w] = vi + c[i-1, w-wi]
      else c[i, w] = c[i-1, w]
    else
      c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, w-W]**.

Analysis

This algorithm takes θ(*n*, *w*) times as table *c* has (*n* + 1).(*w* + 1) entries, where each entry requires θ(1) time to compute.

Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Problem Scenario

A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are **n** items in the store
- Weight of i^{th} item $w_i>0>0$
- Profit for i^{th} item $p_i>0>0$ and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of i^{th} item.

$$0 \leqslant x_i \leqslant 1$$

The i^{th} item contributes the weight $x_i.w_i$ to the total weight in the knapsack and profit $x_i.p_i$ to the total profit.

Hence, the objective of this algorithm is to

$$\text{Maximize } \sum_{n=1\ n} (x_i.pi)$$

subject to constraint,

$$\sum_{n=1n}(x_i.wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{n=1n}(x_i.wi)=W$$

In this context, first we need to sort those items according to the value of p_iw_i so that $p_{i+1}w_{i+1}+1+1 \leq p_iw_i$. Here, ***x*** is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
  do x[i] = 0
weight = 0
for i = 1 to n
  if weight + w[i] ≤ W then
    x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
    weight = W
```

```
    break
return x
```

Analysis

If the provided items are already sorted into a decreasing order of $p_i w_i$, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Example

Let us consider that the capacity of the knapsack ***W* = 60** and the list of provided items are shown in the following table −

Item	**A**	**B**	**C**	**D**
Profit	280	100	120	120
Weight	40	10	20	24
Ratio ($p_i w_i$)	7	10	6	5

As the provided items are not sorted based on $p_i w_i$. After sorting, the items are as shown in the following table.

Item	**B**	**A**	**C**	**D**
Profit	100	280	120	120
Weight	10	40	20	24
Ratio ($p_i w_i$)(	10	7	6	5

Solution

After sorting all the items according to $p_i w_i$**.** First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Job Sequencing with Deadline

Problem Statement

In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

Solution :

Let us consider, a set of n given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

It may happen that all of the given jobs may not be completed within their deadlines.

Assume, deadline of ith job Ji is di and the profit received from this job is pi. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Thus, D(i)>0

for $1 \leqslant i \leqslant n$

Initially, these jobs are ordered according to profit, i.e. $p1 \geqslant p2 \geqslant p3 \geqslant ... \geqslant pn$

Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)

```
D(0) := J(0) := 0

k := 1

J(1) := 1   // means first job is selected

for i = 2 … n do

   r := k

   while D(J(r)) > D(i) and D(J(r)) ≠ r do

      r := r – 1

   if D(J(r)) ≤ D(i) and D(i) > r then

      for l = k … r + 1 by -1 do

         J(l + 1) := J(l)

         J(r + 1) := i

         k := k + 1
```

Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is O(n2)

.Example

Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	J1	J2	J3	J4	J5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	J2J1	J4	J3	J5	
Deadline	1	2	2	3	1
Profit 100	60	40	20	20	

From this set of jobs, first we select J2, as it can be completed within its deadline and contributes maximum profit.

Next, J1 is selected as it gives more profit compared to J4.

In the next clock, J4 cannot be selected as its deadline is over, hence J3 is selected as it executes within its deadline.

The job J5 is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (J2, J1, J3), which are being executed within their deadline and gives maximum profit.

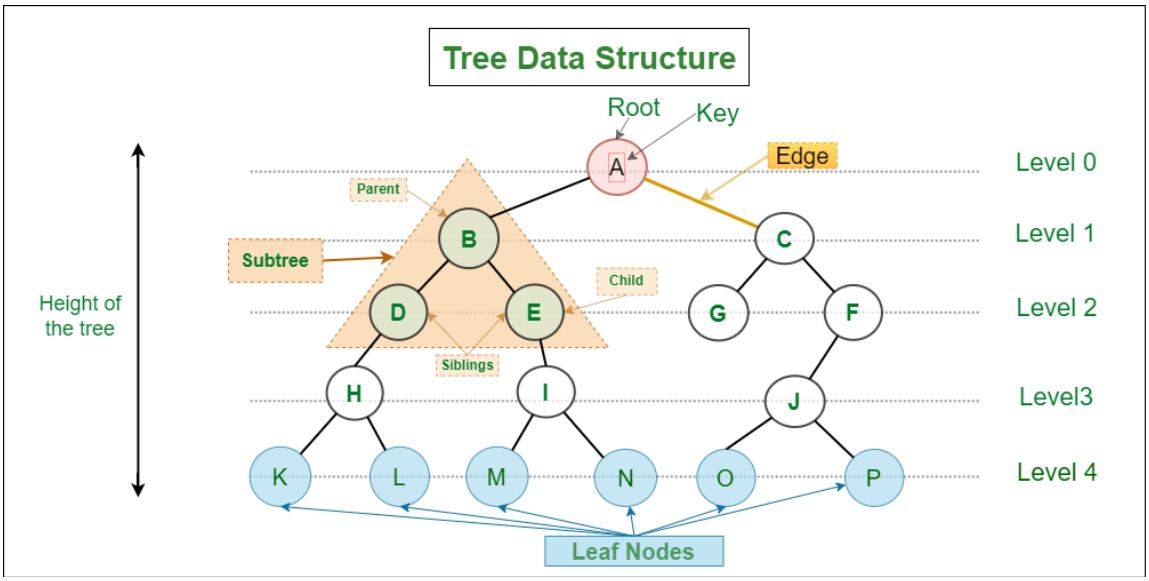
Total profit of this sequence is **100 + 60 + 20 = 180.**

Introduction to Binary Tree

A **tree** is a popular data structure that is non-linear in nature. Unlike other data structures like an array, stack, queue, and linked list which are linear in nature, a tree represents a hierarchical structure. The ordering information of a tree is not important. A tree contains nodes and 2 pointers. These two pointers are the left child and the right child of the parent node. Let us understand the terms of tree in detail.

- **Root:** The root of a tree is the topmost node of the tree that has no parent node. There is only one root node in every tree.

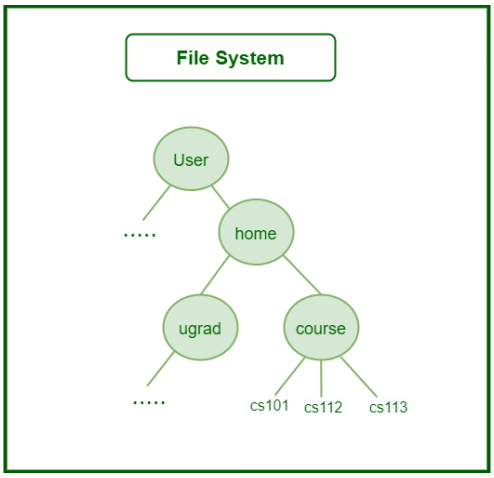
- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node.
- **Sibling:** Children of the same parent node are called siblings.
- **Edge:** Edge acts as a link between the parent node and the child node.
- **Leaf:** A node that has no child is known as the leaf node. It is the last node of the tree. There can be multiple leaf nodes in a tree.
- **Subtree:** The subtree of a node is the tree considering that particular node as the root node.
- **Depth:** The depth of the node is the distance from the root node to that particular node.
- **Height:** The height of the node is the distance from that node to the deepest node of that subtree.
- **Height of tree:** The Height of the tree is the maximum height of any node. This is the same as the height of the root node.
- **Level:** A level is the number of parent nodes corresponding to a given node of the tree.
- **Degree of node:** The degree of a node is the number of its children.
- **NULL:** The number of NULL nodes in a binary tree is (N+1), where N is the number of nodes in a binary tree.



Introduction to Binary Tree – Data Structure and Algorithm Tutorials

Why to use Tree Data Structure?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:



File System

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on the number of nodes as nodes are linked using pointers.

The main applications of tree data structure:

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Form of multi-stage decision-making (see business chess).
7. Trees can be used to represent the structure of a sentence, and can be used in parsing algorithms to analyze the grammar of a sentence.
8. Trees can be used to represent the decision-making process of computer-controlled characters in games, such as in decision trees.
9. Huffman coding uses a tree to represent the frequency of characters in a text, which can be used for data compression.
10. Trees are used to represent the syntax of a programming language, and can be used in compiler design to check the syntax of a program and generate machine code.

What is a Binary Tree?

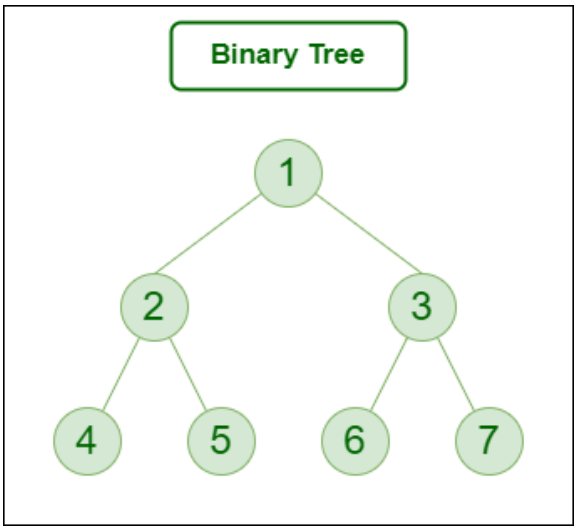
A binary tree is a tree data structure in which each node can have at most two children, which are referred to as the left child and the right child. The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves. A binary tree can be visualized as a hierarchical structure with the root at the top and the leaves at the bottom.

Binary trees have many applications in computer science, including data storage and retrieval, expression evaluation, network routing, and game AI. They can also be used to implement various algorithms such as searching, sorting, and graph algorithms.

Representation of Binary Tree:

Each node in the tree contains the following:

- Data
- Pointer to the left child
- Pointer to the right child




```
/* Class containing left and right child of current
node and key value*/

class Node
{
                constructor(item)
                {
                        this.key = item;
                        this.left = this.right = null;
                }
}
```

Basic Operations On Binary Tree:

- Inserting an element.
- Removing an element.
- Searching for an element.
- Deletion for an element.
- Traversing an element. There are four (mainly three) types of traversals in a binary tree which will be discussed ahead.

Auxiliary Operations On Binary Tree:

- Finding the height of the tree
- Find the level of the tree
- Finding the size of the entire tree.

Applications of Binary Tree:

- In compilers, Expression Trees are used which is an application of binary trees.
- Huffman coding trees are used in data compression algorithms.
- Priority Queue is another application of binary tree that is used for searching maximum or minimum in O(1) time complexity.
- Represent hierarchical data.
- Used in editing software like Microsoft Excel and spreadsheets.
- Useful for indexing segmented at the database is useful in storing cache in the system,
- Syntax trees are used for most famous compilers for programming like GCC, and AOCL to perform arithmetic operations.
- For implementing priority queues.
- Used to find elements in less time (binary search tree)
- Used to enable fast memory allocation in computers.
- Used to perform encoding and decoding operations.
- Binary trees can be used to organize and retrieve information from large datasets, such as in inverted index and k-d trees.
- Binary trees can be used to represent the decision-making process of computer-controlled characters in games, such as in decision trees.
- Binary trees can be used to implement searching algorithms, such as in binary search trees which can be used to quickly find an element in a sorted list.
- Binary trees can be used to implement sorting algorithms, such as in heap sort which uses a binary heap to sort elements efficiently.

Binary Tree Traversals:

Tree Traversal algorithms can be classified broadly into two categories:

- Depth-First Search (DFS) Algorithms
- Breadth-First Search (BFS) Algorithms

Tree Traversal using Depth-First Search (DFS) algorithm can be further classified into three categories:

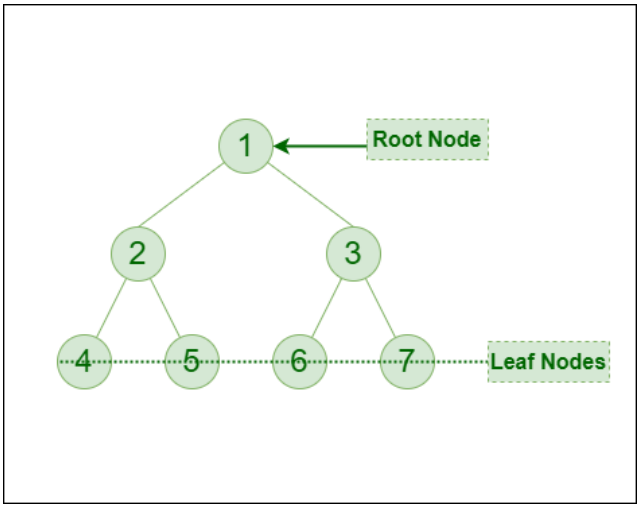
- **Preorder Traversal (current-left-right):** Visit the current node before visiting any nodes inside the left or right subtrees. Here, the traversal is root – left child – right child. It means that the root node is traversed first then its left child and finally the right child.
- **Inorder Traversal (left-current-right):** Visit the current node after visiting all nodes inside the left subtree but before visiting any node within the right subtree. Here, the traversal is left child – root – right child. It means that the left child is traversed first then its root node and finally the right child.

- **Postorder Traversal (left-right-current):** Visit the current node after visiting all the nodes of the left and right subtrees. Here, the traversal is left child – right child – root. It means that the left child has traversed first then the right child and finally its root node.

Tree Traversal using Breadth-First Search (BFS) algorithm can be further classified into one category:

- **Level Order Traversal:** Visit nodes level-by-level and left-to-right fashion at the same level. Here, the traversal is level-wise. It means that the most left child has traversed first and then the other children of the same level from left to right have traversed.

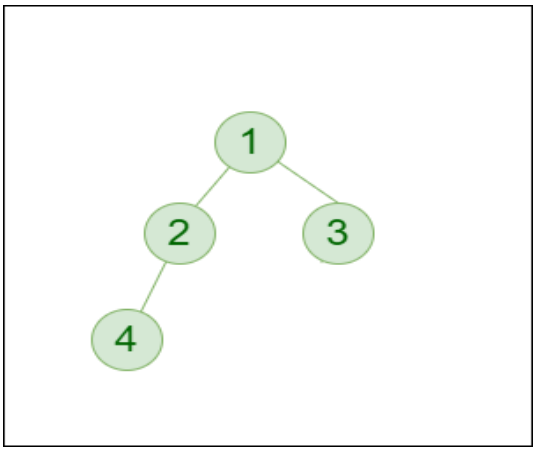
Let us traverse the following tree with all four traversal methods:



Pre-order Traversal of the above tree: 1-2-4-5-3-6-7
In-order Traversal of the above tree: 4-2-5-1-6-3-7
Post-order Traversal of the above tree: 4-5-2-6-7-3-1
Level-order Traversal of the above tree: 1-2-3-4-5-6-7

Implementation of Binary Tree:

Let us create a simple tree with 4 nodes. The created tree would be as follows.



/* Class containing left and right child of current

node and key value*/
class Node {

constructor(val) {
this.key = val;
this.left = null;
this.right = null;
}

}

// A javascript program to introduce Binary Tree
// Root of Binary Tree

var root = null;

/*create root*/

root = new Node(1);

/* following is the tree after above statement
     1
    / \
null null */

root.left = new Node(2);

root.right = new Node(3);

```
/* 2 and 3 become left and right children of 1
    1
   / \
   2 3
/ \ / \
null null null null */

root.left.left = new Node(4);

/* 4 becomes left child of 2
1
/ \
2 3
/ \ / \
4 null null null
/ \
null null
*/
```

Properties of Binary Search Tree :

1. The maximum number of nodes at level 'l' of a binary tree is 2^l:
Note: Here level is the number of nodes on the path from the root to the node (including root and node). The level of the root is 0
This can be proved by induction:

For root, l = 0, number of nodes = 2^0 = 1
Assume that the maximum number of nodes on level 'l' is 2^l
*Since in a Binary tree every node has at most 2 children, the next level would have twice nodes, i.e. 2 * 2^l*

2. The Maximum number of nodes in a binary tree of height 'h' is 2^h – 1:
Note: Here the height of a tree is the maximum number of nodes on the root-to-leaf path. The height of a tree with a single node is considered as 1
This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So the maximum number of nodes in a binary tree of height h is 1 + 2 + 4 + .. + 2^{h-1}. This is a simple geometric series with h terms and the sum of this series is 2^h– 1.
In some books, the height of the root is considered as 0. In this convention, the above formula becomes 2^{h+1} – 1

3. In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is $Log_2(N+1)$:

Each level should have at least one element, so the height cannot be more than N. A binary tree of height 'h' can have a maximum of $2^h - 1$ nodes (previous property). So the number of nodes will be less than or equal to this maximum value

$N <= 2^h - 1$

$2^h >= N+1$

$log_2(2^h) >= log_2(N+1)$ (Taking log both sides)

$h log_2 2 >= log_2(N+1)$ (h is an integer)

$h >= | log_2(N+1) |$

So the minimum height possible is $| log_2(N+1) |$

4. A Binary Tree with L leaves has at least $| Log_2 L |$+ 1 levels:

A Binary tree has the maximum number of leaves (and a minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is valid for the number of leaves L

$L <= 2^{l-1}$ [From Point 1] [Note: Here, consider level of root node as 1]

$l = | Log_2 L | + 1$

where l is the minimum number of levels

5. In a Binary tree where every node has 0 or 2 children, the number of leaf nodes is always one more than nodes with two children:

$L = T + 1$

Where L = Number of leaf nodes

T = Number of internal nodes with two children

Proof:

No. of leaf nodes (L) i.e. total elements present at the bottom of tree = 2^{h-1} (h is height of tree)

No. of internal nodes = {total no. of nodes} – {leaf nodes} = $\{ 2^h - 1 \} - \{2^{h-1}\} = 2^{h-1} (2-1) - 1 = 2^{h-1} - 1$

So , $L = 2^{h-1}$

$T = 2^{h-1} - 1$

Therefore $L = T + 1$

Hence proved

6. In a non-empty binary tree, if n is the total number of nodes and e is the total number of edges, then e = n-1:

Every node in a binary tree has exactly one parent with the exception of the root node. So if n is the total number of nodes then n-1 nodes have exactly one parent. There is only one edge between any child and its parent. So the total number of edges is n-1.

Types of Binary Tree based on the number of children:

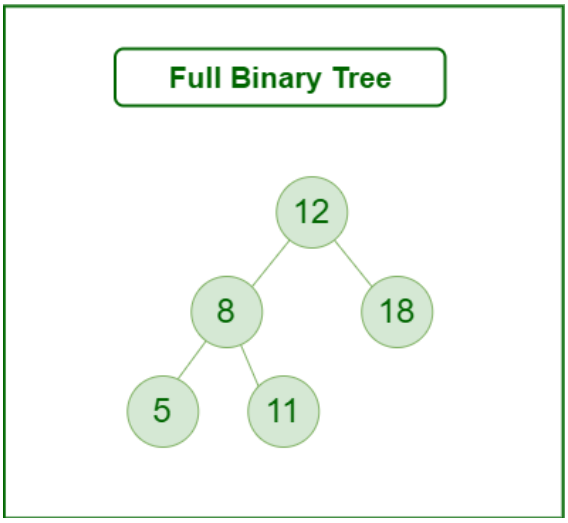
Following are the types of Binary Tree based on the number of children:

1. Full Binary Tree
2. Degenerate Binary Tree
3. Skewed Binary Trees

1. Full Binary Tree

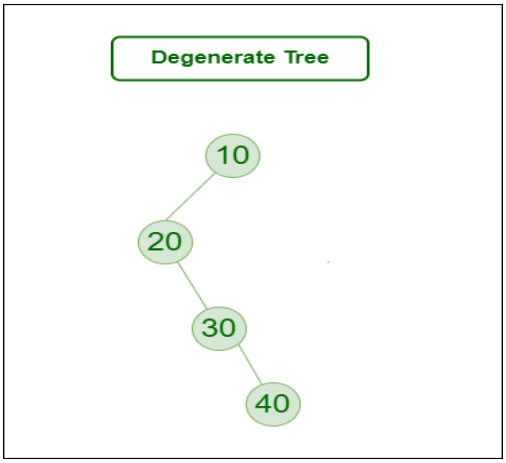
A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. It is also known as a proper binary tree.



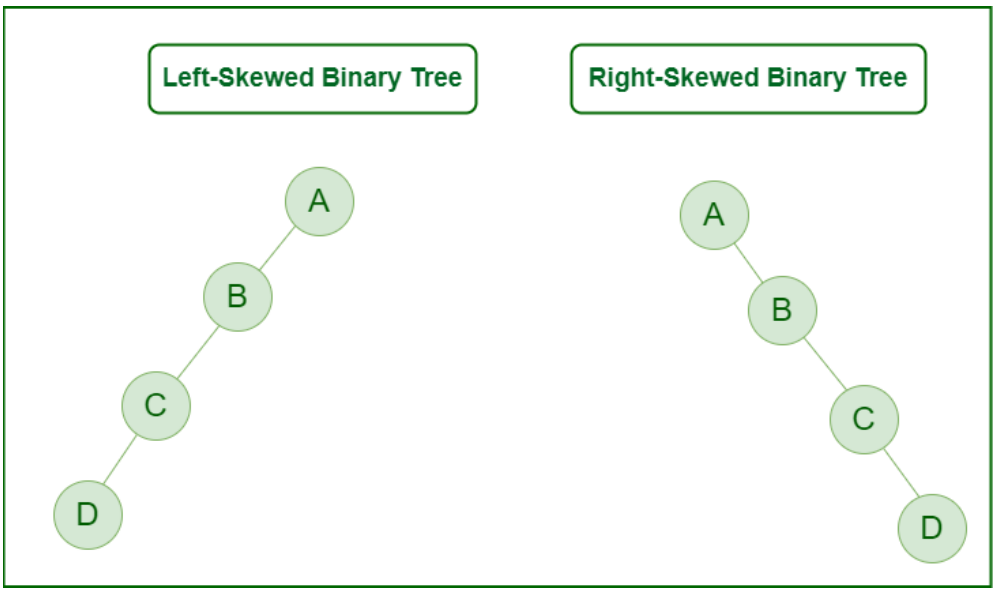
2. Degenerate (or pathological) tree

A Tree where every internal node has one child. Such trees are performance-wise same as linked list. A degenerate or pathological tree is a tree having a single child either left or right.



3. Skewed Binary Tree

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.



Types of Binary Tree On the basis of the completion of levels:

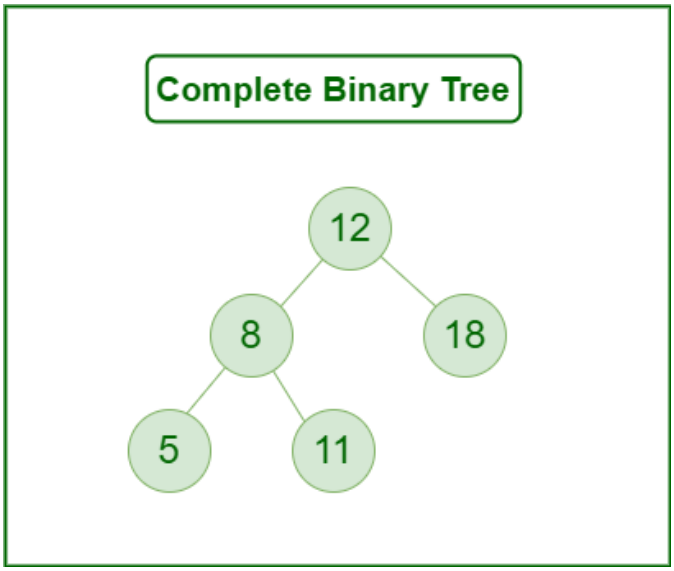
1. Complete Binary Tree
2. Perfect Binary Tree
3. Balanced Binary Tree

1. Complete Binary Tree

A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

A complete binary tree is just like a full binary tree, but with two major differences:

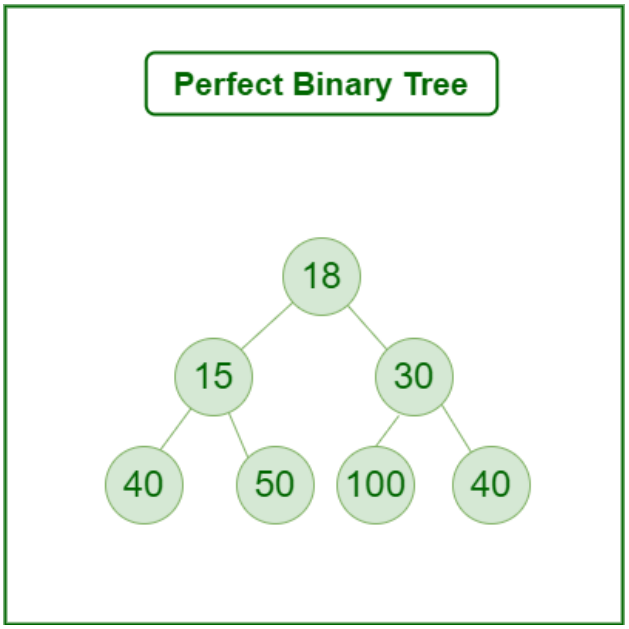
- Every level must be completely filled
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



2. Perfect Binary Tree

A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.
The following are examples of Perfect Binary Trees.

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

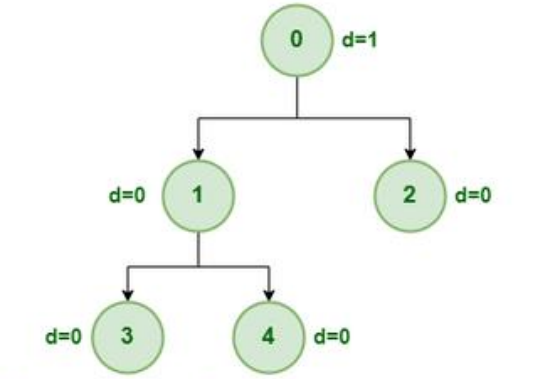


In a Perfect Binary Tree, the number of leaf nodes is the number of internal nodes plus 1

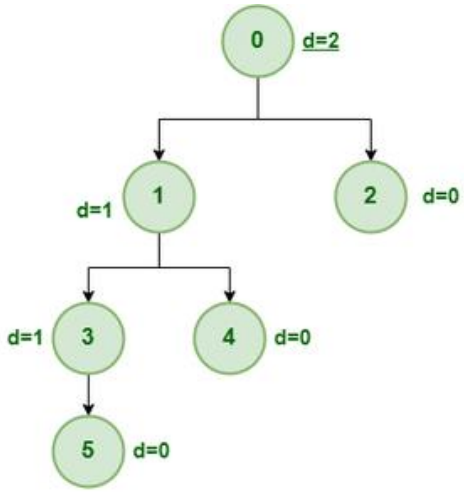
L = I + 1 Where L = Number of leaf nodes, I = Number of internal nodes.

A Perfect Binary Tree of height h (where the height of the binary tree is the number of edges in the longest path from the root node to any leaf node in the tree, height of root node is 0) has $2^{h+1} – 1$ node.
An example of a Perfect binary tree is ancestors in the family. Keep a person at root, parents as children, parents of parents as their children.

3. Balanced Binary Tree
A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes. For Example, the AVL tree maintains O(Log n) height by making sure that the difference between the heights of the left and right subtrees is at most 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths is the same and that there are no adjacent red nodes. Balanced Binary Search trees are performance-wise good as they provide O(log n) time for search, insert and delete.



Balanced Binary Tree with depth at each level indicated
Depth of a node = |height of left child - height of right child|



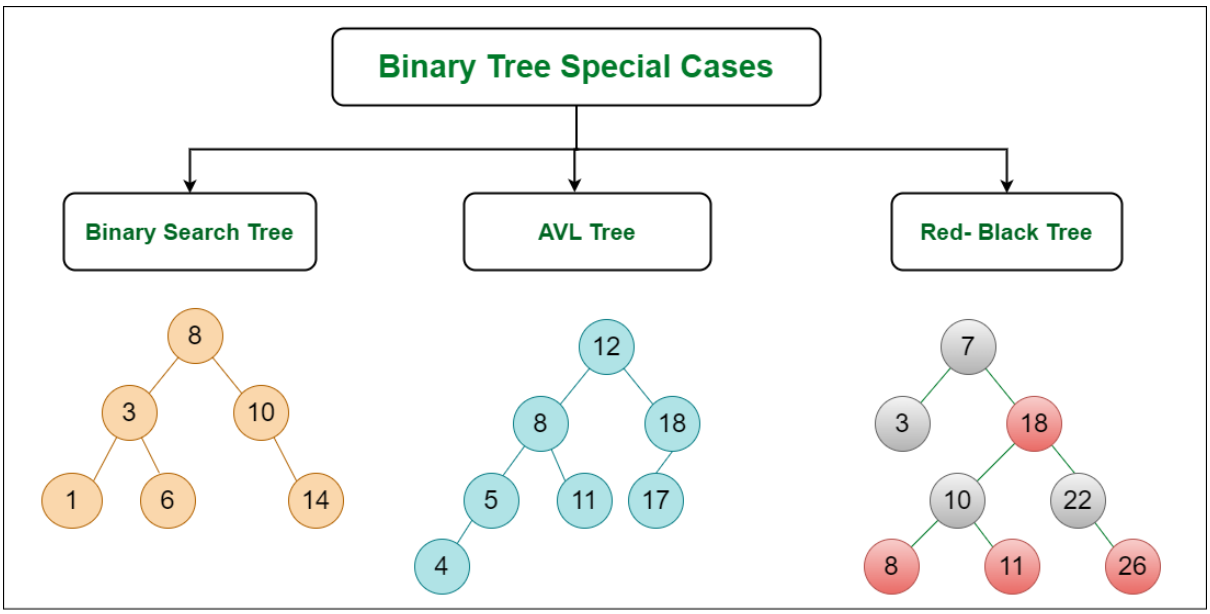
Unbalanced Binary Tree with depth at each level indicated
Depth of a node = |height of left child - height of right child|

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1. In the figure above, the root node having a value 0 is unbalanced with a depth of 2 units.

Some Special Types of Trees:

On the basis of node values, the Binary Tree can be classified into the following special types:
1. Binary Search Tree
2. AVL Tree
3. Red Black Tree
4. B Tree
5. B+ Tree
6. Segment Tree

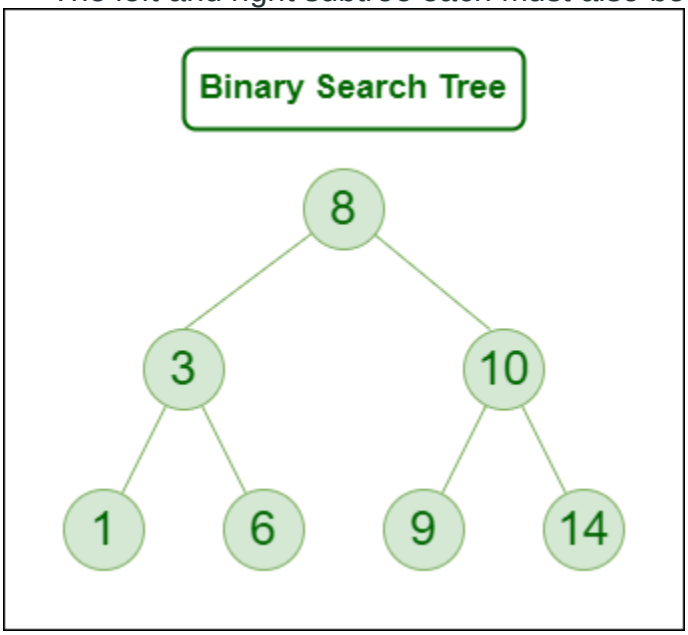
Below Image Shows Important Special cases of binary Trees:



1. Binary Search Tree

Binary Search Tree is a node-based binary tree data structure that has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

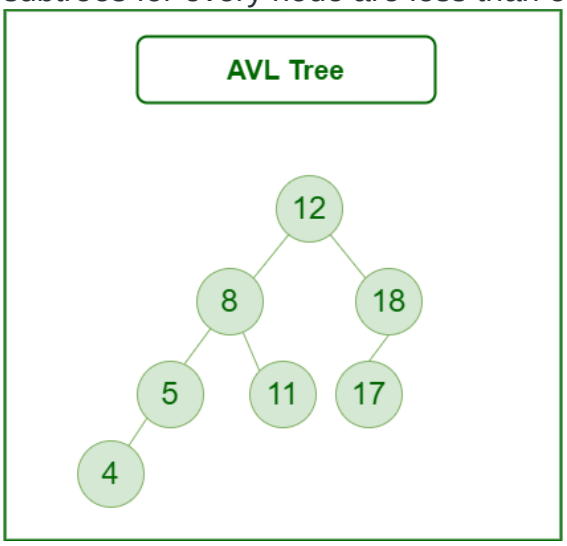


2. AVL Tree

AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

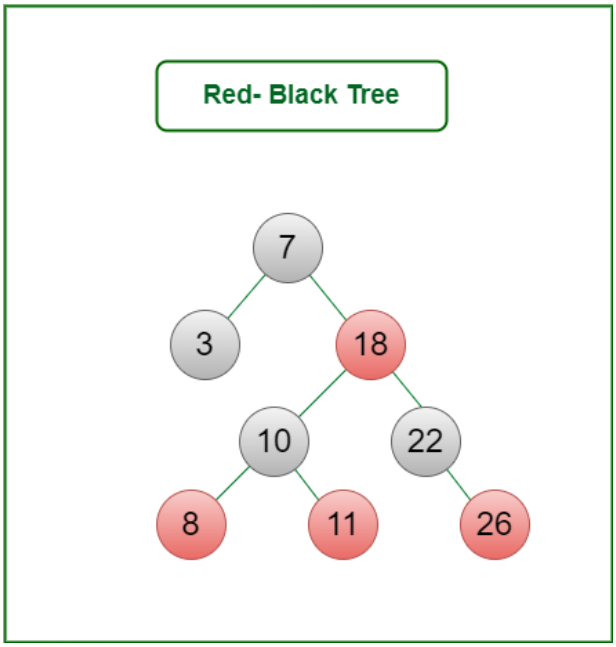
Example of AVL Tree shown below:

The below tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1



3. Red Black Tree

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. This tree was invented in 1972 by Rudolf Bayer.



4. B – Tree

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in the main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in the main memory. When the number of keys is high, the data is read from the disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting the maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, etc.

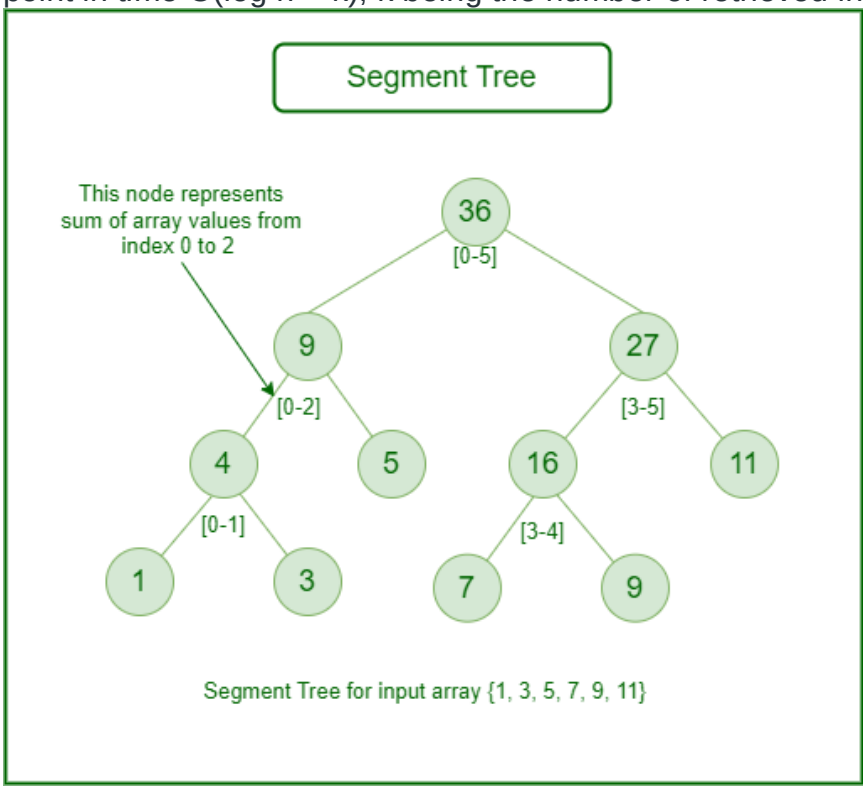
5. B+ Tree

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in the main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in the main memory. When the number of keys is high, the data is read from the disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk

accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting the maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, etc.

6. Segment Tree

In computer science, a **Segment Tree**, also known as a statistic tree, is a tree data structure used for storing information about intervals, or segments. It allows querying which of the stored segments contain a given point. It is, in principle, a static structure; that is, it's a structure that cannot be modified once it's built. A similar data structure is the interval tree.
A **segment tree** for a set I of n intervals uses O(n log n) storage and can be built in O(n log n) time. Segment trees support searching for all the intervals that contain a query point in time O(log n + k), k being the number of retrieved intervals or segments.



Knapsack Problem-

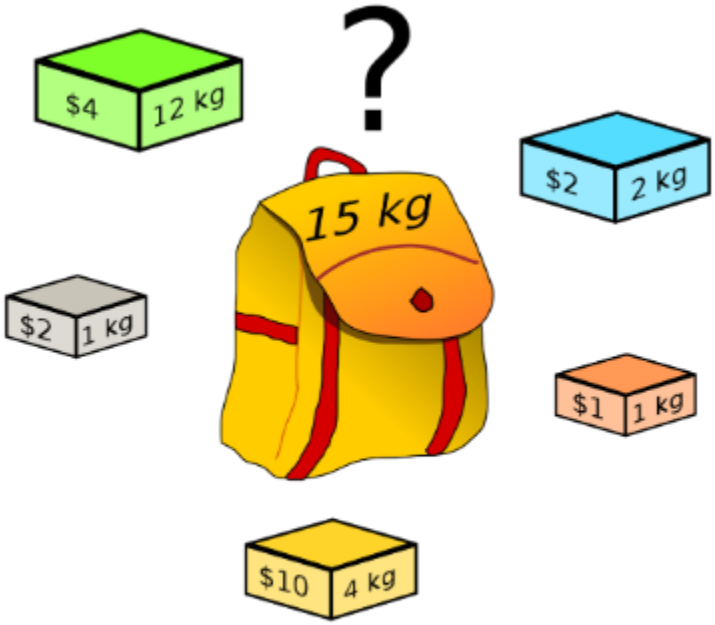
You are given the following-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

The problem states-

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.



Knapsack Problem

Knapsack Problem Variants-

Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

In this article, we will discuss about Fractional Knapsack Problem.

Fractional Knapsack Problem-

In Fractional Knapsack Problem,

- As the name suggests, items are divisible here.
- We can even put the fraction of any item into the knapsack if taking the complete item is not possible.
- It is solved using Greedy Method.

Fractional Knapsack Problem Using Greedy Method-

Fractional knapsack problem is solved using greedy method in the following steps-

Step-01:

For each item, compute its value / weight ratio.

Step-02:

Arrange all the items in decreasing order of their value / weight ratio.

Step-03:

Start putting the items into the knapsack beginning from the item with the highest ratio.

Put as many items as you can into the knapsack.

Time Complexity-

- The main time taking step is the sorting of all items in decreasing order of their value / weight ratio.
- If the items are already arranged in the required order, then while loop takes O(n) time.
- The average time complexity of **Quick Sort** is O(nlogn).
- Therefore, total time taken including the sort is O(nlogn).

Problem-

For the given set of items and knapsack capacity = 60 kg, find the optimal solution for the fractional knapsack problem making use of greedy approach.

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

OR

Find the optimal solution for the fractional knapsack problem making use of greedy approach. Consider-

n = 5

w = 60 kg

(w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25)

(b1, b2, b3, b4, b5) = (30, 40, 45, 77, 90)

OR

A thief enters a house for robbing it. He can carry a maximal weight of 60 kg into his bag. There are 5 items in the house with the following weights and values. What items should thief take if he can even take the fraction of any item with him?

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

Solution-

Step-01:

Compute the value / weight ratio for each item-

Items	Weight	Value	Ratio

1	5	30	6
2	10	40	4
3	15	45	3
4	22	77	3.5
5	25	90	3.6

Step-02:

Sort all the items in decreasing order of their value / weight ratio-

I1 I2 I5 I4 I3

(6) (4) (3.6) (3.5) (3)

Step-03:

Start filling the knapsack by putting the items into it one by one.

Knapsack Weight	**Items in Knapsack**	**Cost**

60	Ø	0
55	I1	30
45	I1, I2	70
20	I1, I2, I5	160

Now,

- Knapsack weight left to be filled is 20 kg but item-4 has a weight of 22 kg.
- Since in fractional knapsack problem, even the fraction of any item can be taken.
- So, knapsack will contain the following items-

< I1 , I2 , I5 , (20/22) I4 >

Total cost of the knapsack

= 160 + (20/27) x 77

= 160 + 70

= 230 units

Important Note-

Had the problem been a 0/1 knapsack problem, knapsack would contain the following items-

< I1 , I2 , I5 >

The knapsack's total cost would be 160 units.

==========

Job Sequencing With Deadlines-

The sequencing of jobs on a single processor with deadline constraints is called as Job Sequencing with Deadlines.

Here-

- You are given a set of jobs.
- Each job has a defined deadline and some profit associated with it.
- The profit of a job is given only when that job is completed within its deadline.
- Only one processor is available for processing all the jobs.
- Processor takes one unit of time to complete a job.

The problem states-

"How can the total profit be maximized if only one job can be completed at a time?"

Approach to Solution-

- A feasible solution would be a subset of jobs where each job of the subset gets completed within its deadline.
- Value of the feasible solution would be the sum of profit of all the jobs contained in the subset.
- An optimal solution of the problem would be a feasible solution which gives the maximum profit.

Greedy Algorithm-

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.

The greedy algorithm described below always gives an optimal solution to the job sequencing problem-

Step-01:

- Sort all the given jobs in decreasing order of their profit.

Step-02:

- Check the value of maximum deadline.
- Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

Step-03:

- Pick up the jobs one by one.
- Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

PRACTICE PROBLEM BASED ON JOB SEQUENCING WITH DEADLINES-

Problem-

Given the jobs, their deadlines and associated profits as shown-

Jobs	**J1**	**J2**	**J3**	**J4**	**J5**	**J6**
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Answer the following questions-

1. Write the optimal schedule that gives maximum profit.
2. Are all the jobs completed in the optimal schedule?
3. What is the maximum earned profit?

Solution-

Step-01:

Sort all the given jobs in decreasing order of their profit-

Jobs	**J4**	**J1**	**J3**	**J2**	**J5**	**J6**
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

Value of maximum deadline = 5.

So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-

Gantt Chart

Now,

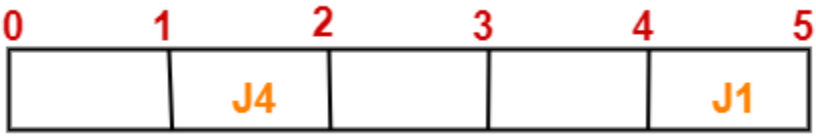
- We take each job one by one in the order they appear in Step-01.
- We place the job on Gantt chart as far as possible from 0.

Step-03:

- We take job J4.
- Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-

Step-04:

- We take job J1.
- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



Step-05:

- We take job J3.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-

Step-06:

- We take job J2.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3.
- Since the second and third cells are already filled, so we place job J2 in the first cell as-

Step-07:

- Now, we take job J5.
- Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-

Now,

- The only job left is job J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 can not be completed.

Now, the given questions may be answered as-

Part-01:

The optimal schedule is-

J2 , J4 , J3 , J5 , J1

This is the required order in which the jobs must be completed in order to obtain the maximum profit.

Part-02:

- All the jobs are not completed in optimal schedule.
- This is because job J6 could not be completed within its deadline.

Part-03:

Maximum earned profit

= Sum of profit of all the jobs in optimal schedule

= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1

= 180 + 300 + 190 + 120 + 200

= 990 units

==========

Huffman Coding-

- Huffman Coding is a famous Greedy Algorithm.
- It is used for the lossless compression of data.
- It uses variable length encoding.
- It assigns variable length code to all the characters.
- The code length of a character depends on how frequently it occurs in the given text.
- The character which occurs most frequently gets the smallest code.
- The character which occurs least frequently gets the largest code.
- It is also known as **Huffman Encoding**.

Prefix Rule-

- Huffman Coding implements a rule known as a prefix rule.
- This is to prevent the ambiguities while decoding.
- It ensures that the code assigned to any character is not a prefix of the code assigned to any other character.

Major Steps in Huffman Coding-

There are two major steps in Huffman Coding-

1. Building a Huffman Tree from the input characters.
2. Assigning code to the characters by traversing the Huffman Tree.

Huffman Tree-

The steps involved in the construction of Huffman Tree are as follows-

Step-01:

- Create a leaf node for each character of the text.

- Leaf node of a character contains the occurring frequency of that character.

Step-02:

- Arrange all the nodes in increasing order of their frequency value.

Step-03:

Considering the first two nodes having minimum frequency,

- Create a new internal node.
- The frequency of this new node is the sum of frequency of those two nodes.
- Make the first node as a left child and the other node as a right child of the newly created node.

Step-04:

- Keep repeating Step-02 and Step-03 until all the nodes form a single tree.
- The tree finally obtained is the desired Huffman Tree.

Time Complexity-

The time complexity analysis of Huffman Coding is as follows-

- extractMin( ) is called 2 x (n-1) times if there are n nodes.
- As extractMin( ) calls minHeapify( ), it takes O(logn) time.

Thus, Overall time complexity of Huffman Coding becomes **O(nlogn)**.

Here, n is the number of unique characters in the given text.

Important Formulas-

The following 2 formulas are important to solve the problems based on Huffman Coding-

Formula-01:

$$\text{Average code length per character} = \frac{\Sigma\,(\text{frequency}_i \times \text{code length}_i)}{\Sigma\,\text{frequency}_i}$$

$$= \Sigma\,(\text{probability}_i \times \text{code length}_i)$$

Formula-02:

Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

= $\sum$ (frequency_i x Code length_i)

PRACTICE PROBLEM BASED ON HUFFMAN CODING-

Problem-

A file contains the following characters with the frequencies as shown. If Huffman Coding is used for data compression, determine-

1. Huffman Code for each character
2. Average code length
3. Length of Huffman encoded message (in bits)

Characters	Frequencies
a	10
e	15
i	12

o	3
u	4
s	13
t	1

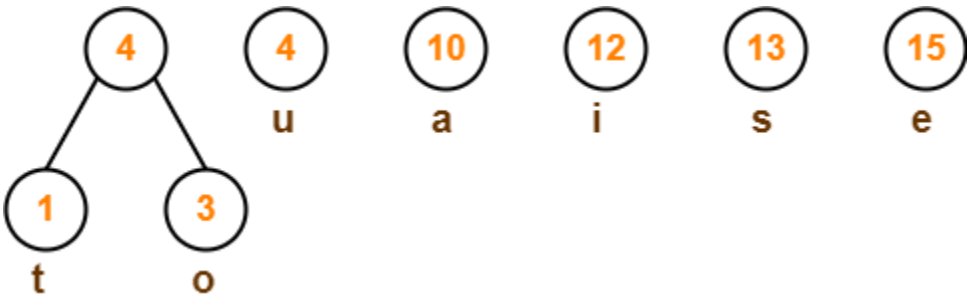
Solution-

First let us construct the Huffman Tree.

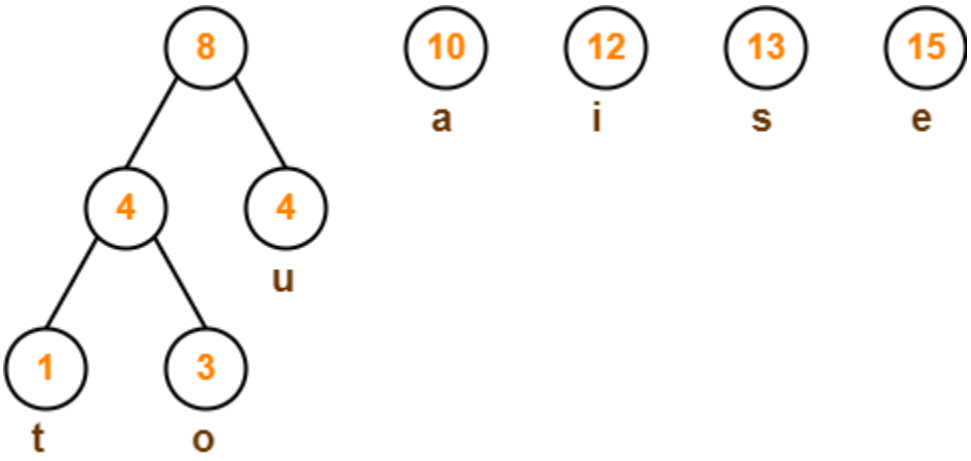
Huffman Tree is constructed in the following steps-

Step-01:

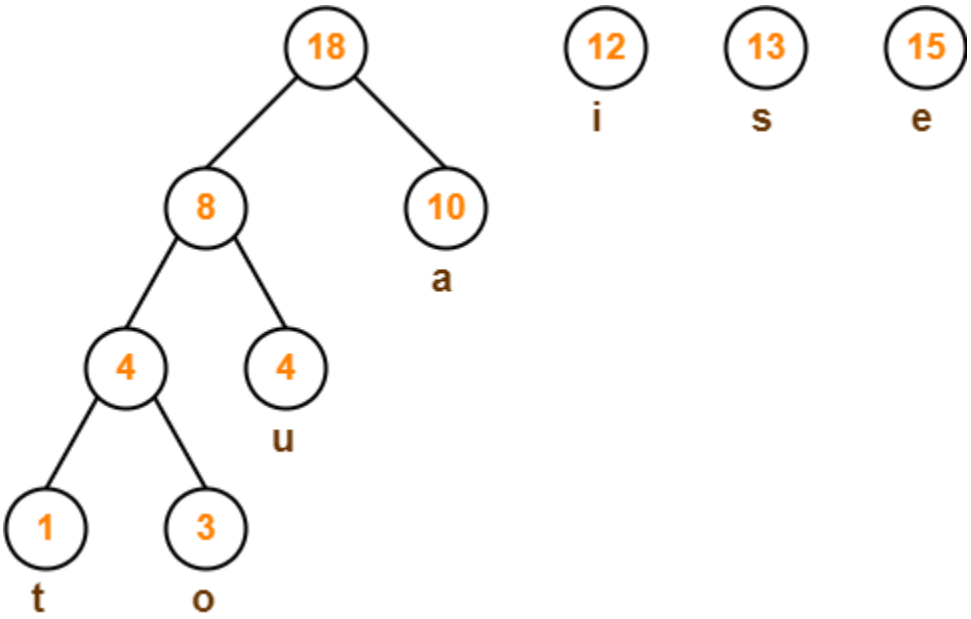
Step-02:



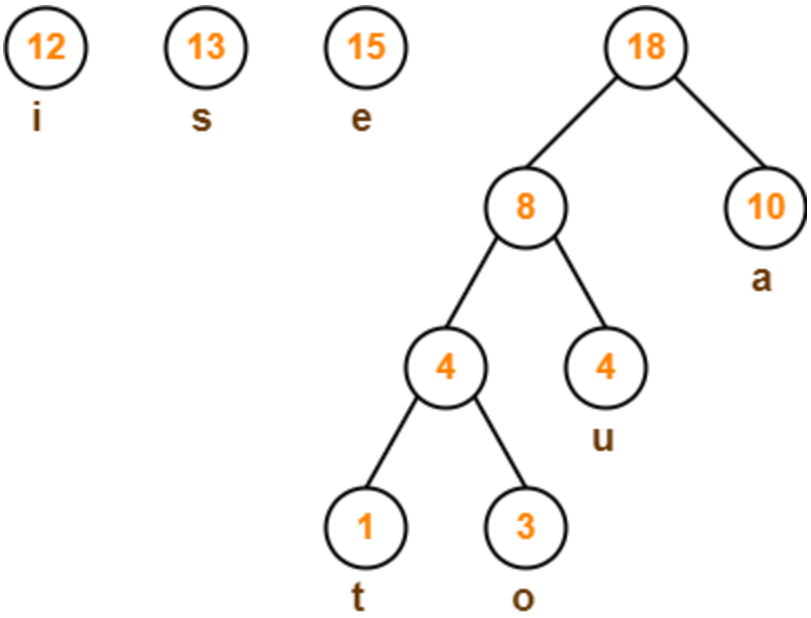
Step-03:

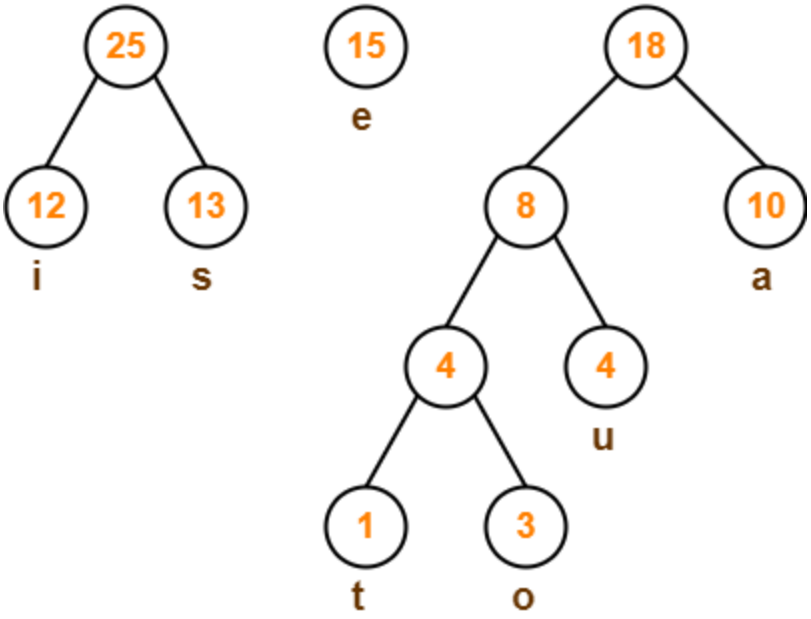


Step-04:

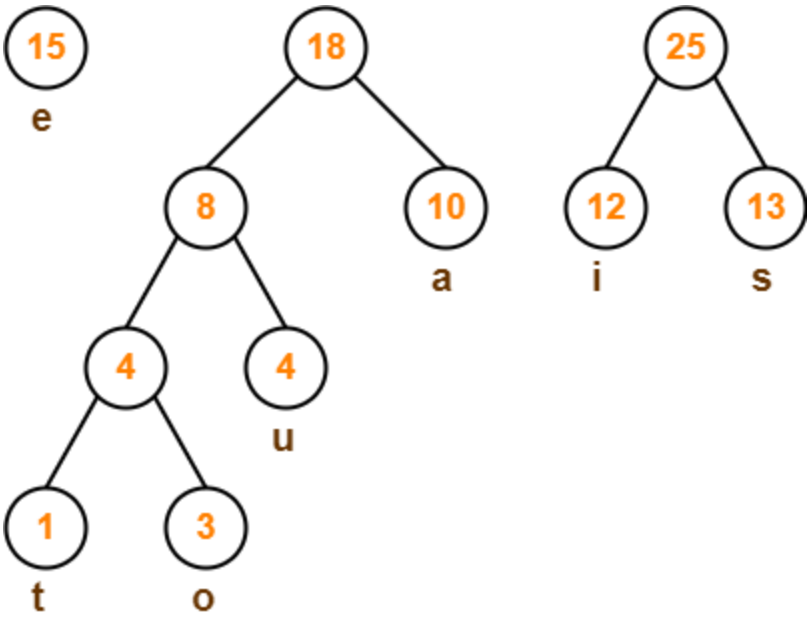


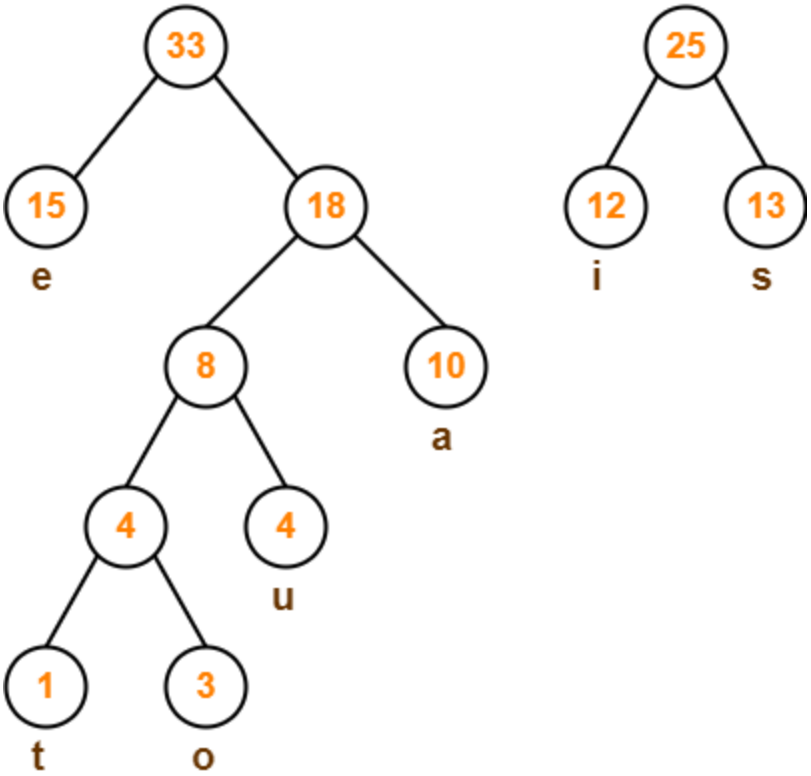
Step-05:



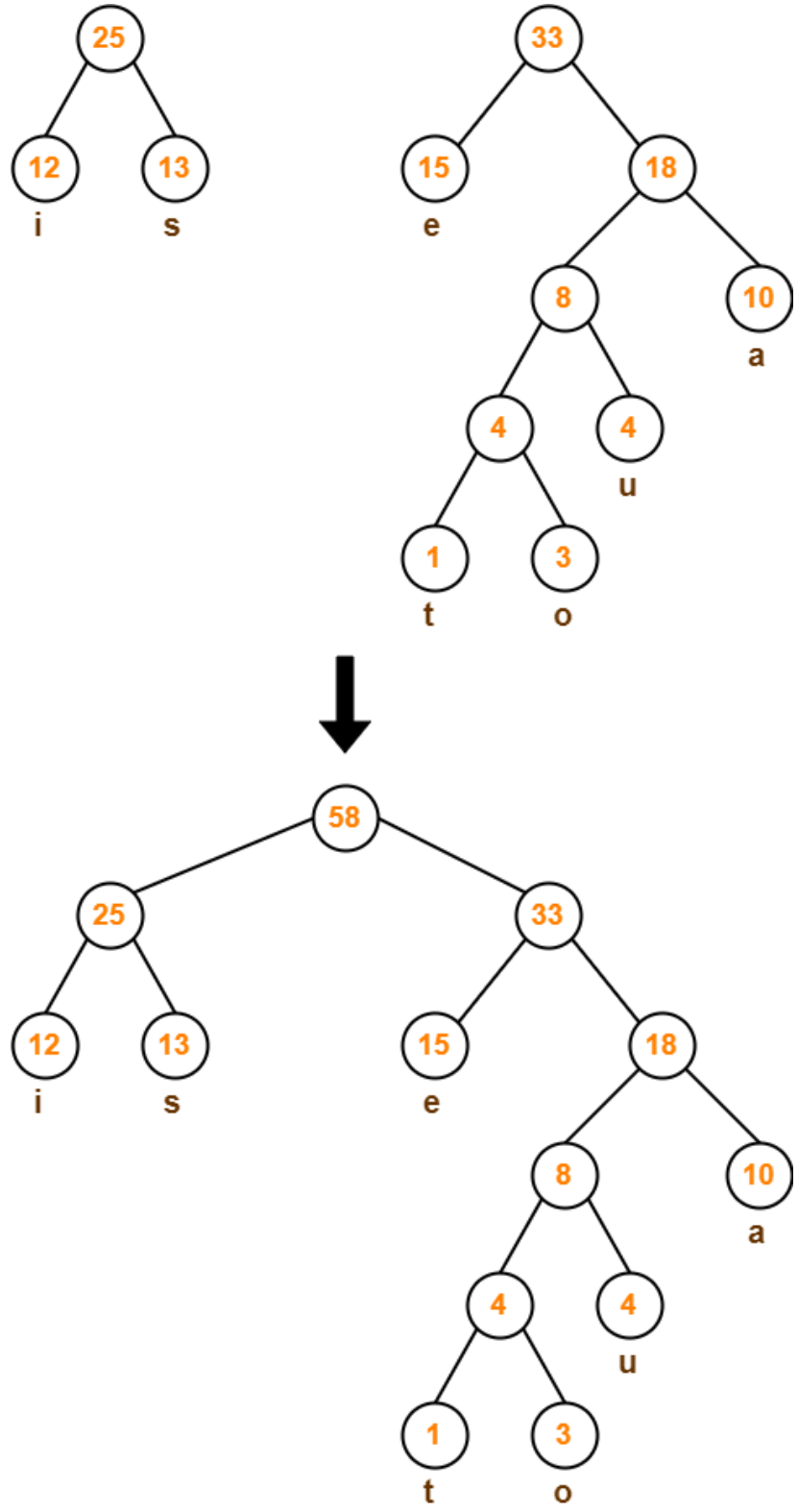


Step-06:


15
e
18
8
10
a
4
4
u
1
3
t
o
25
12
13
i
s


33
15
e
18
8
10
a
4
4
u
1
3
t
o
25
12
13
i
s

Step-07:



Huffman Tree

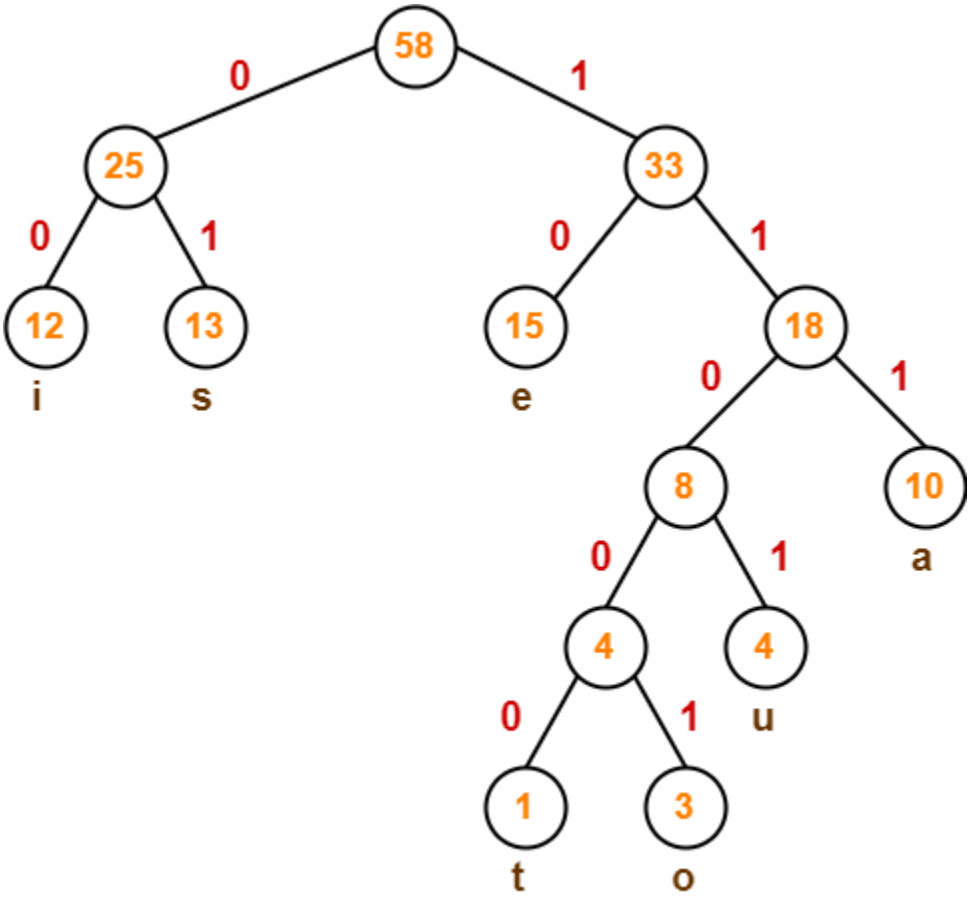
Now,

- We assign weight to all the edges of the constructed Huffman Tree.
- Let us assign weight ‘0’ to the left edges and weight ‘1’ to the right edges.

Rule

- If you assign weight ‘0’ to the left edges, then assign weight ‘1’ to the right edges.
- If you assign weight ‘1’ to the left edges, then assign weight ‘0’ to the right edges.
- Any of the above two conventions may be followed.
- But follow the same convention at the time of decoding that is adopted at the time of encoding.

After assigning weight to all the edges, the modified Huffman Tree is-



Huffman Tree

Now, let us answer each part of the given problem one by one-

1. Huffman Code For Characters-

To write Huffman Code for any character, traverse the Huffman Tree from root node to the leaf node of that character.

Following this rule, the Huffman Code for each character is-

- a = 111
- e = 10
- i = 00
- o = 11001
- u = 1101
- s = 01
- t = 11000

From here, we can observe-

- Characters occurring less frequently in the text are assigned the larger code.
- Characters occurring more frequently in the text are assigned the smaller code.

2. Average Code Length-

Using formula-01, we have-

Average code length

$= \sum (\text{frequency}_i \times \text{code length}_i) / \sum (\text{frequency}_i)$

= { (10 x 3) + (15 x 2) + (12 x 2) + (3 x 5) + (4 x 4) + (13 x 2) + (1 x 5) } / (10 + 15 + 12 + 3 + 4 + 13 + 1)

= 2.52

3. Length of Huffman Encoded Message-

Using formula-02, we have-

Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

= 58 x 2.52

= 146.16

≅ 147 bits

Prim's Algorithm-

- Prim's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Prim's algorithm, the given graph must be weighted, connected and undirected.

Prim's Algorithm Implementation-

The implementation of Prim's Algorithm is explained in the following steps-

Step-01:

- Randomly choose any vertex.
- The vertex connecting to the edge having least weight is usually selected.

Step-02:

- Find all the edges that connect the tree to new vertices.
- Find the least weight edge among those edges and include it in the existing tree.
- If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

Step-03:

- Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.

Prim's Algorithm Time Complexity-

Worst case time complexity of Prim's Algorithm is-

- O(ElogV) using binary heap
- O(E + VlogV) using Fibonacci heap

Time Complexity Analysis

- If adjacency list is used to represent the graph, then using breadth first search, all the vertices can be traversed in O(V + E) time.
- We traverse all the vertices of graph using breadth first search and use a min heap for storing the vertices not yet included in the MST.
- To get the minimum weight edge, we use min heap as a priority queue.
- Min heap operations like extracting minimum element and decreasing key value takes O(logV) time.

So, overall time complexity

= O(E + V) x O(logV)

= O((E + V)logV)

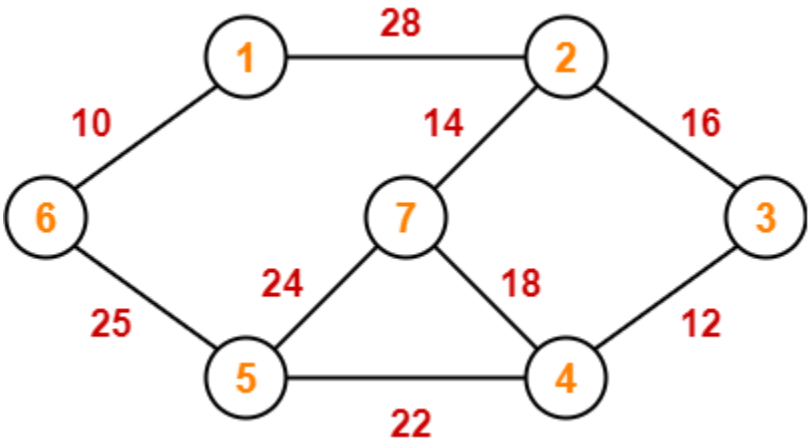
= O(ElogV)

This time complexity can be improved and reduced to O(E + VlogV) using Fibonacci heap.

PRACTICE PROBLEMS BASED ON PRIM'S ALGORITHM-

Problem-01:

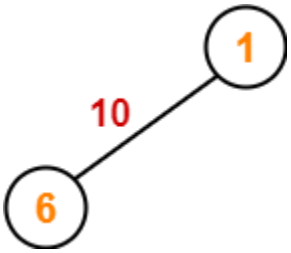
Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-



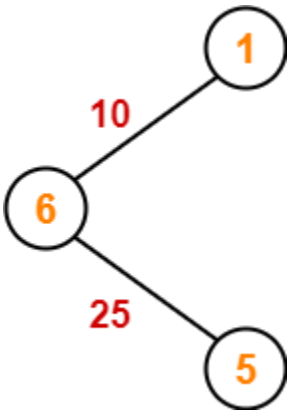
Solution-

The above discussed steps are followed to find the minimum cost spanning tree using Prim's Algorithm-

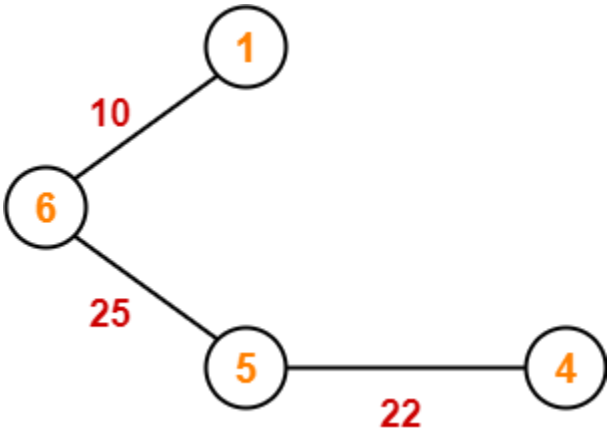
Step-01:



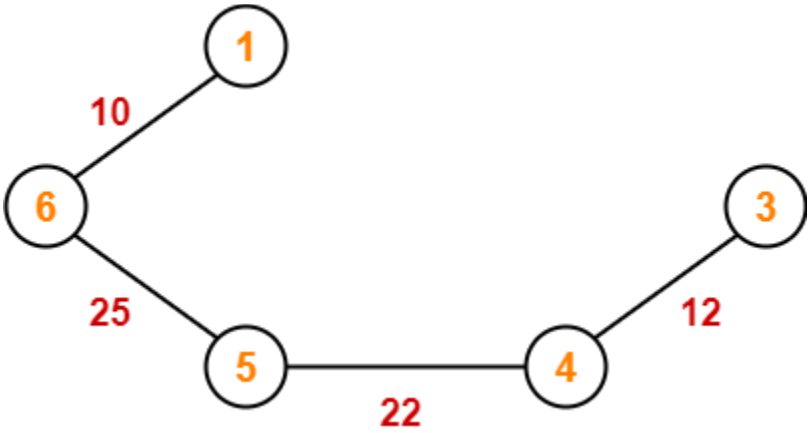
Step-02:



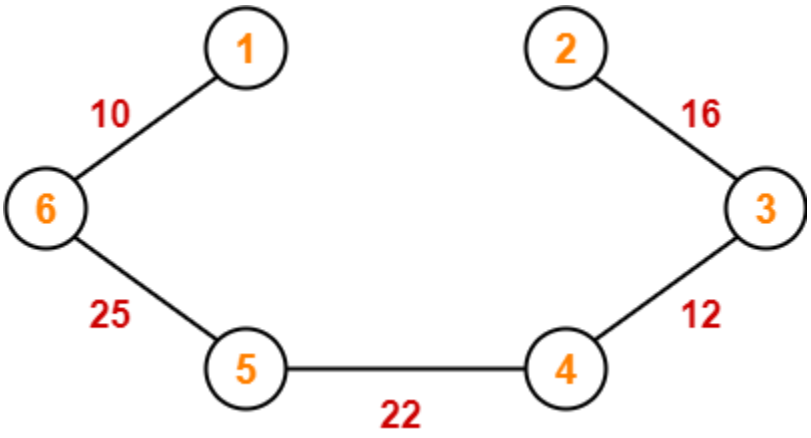
Step-03:



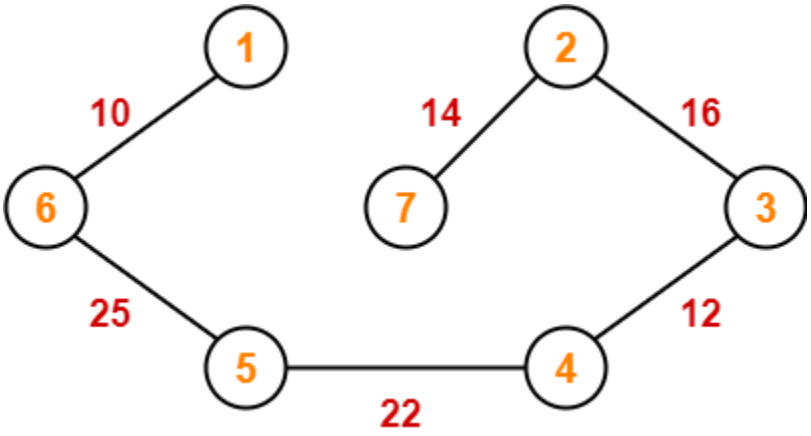
Step-04:



Step-05:



Step-06:



Since all the vertices have been included in the MST, so we stop.

Now, Cost of Minimum Spanning Tree

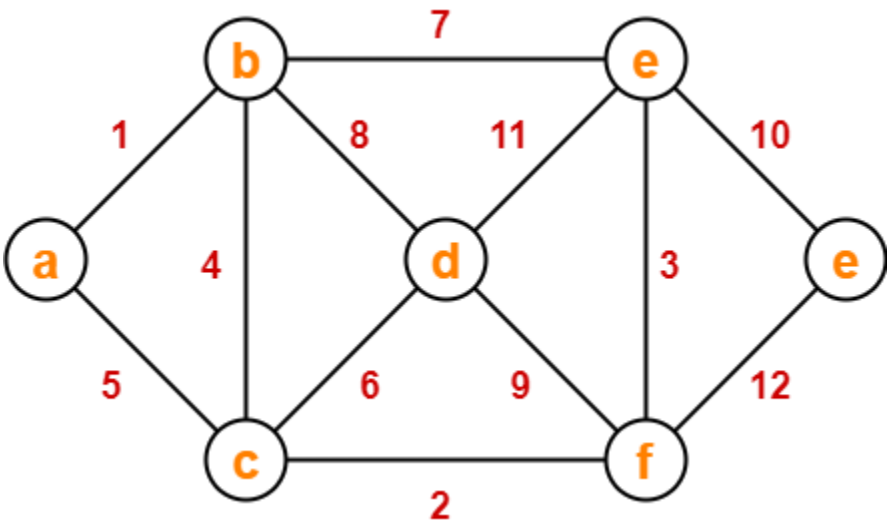
= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

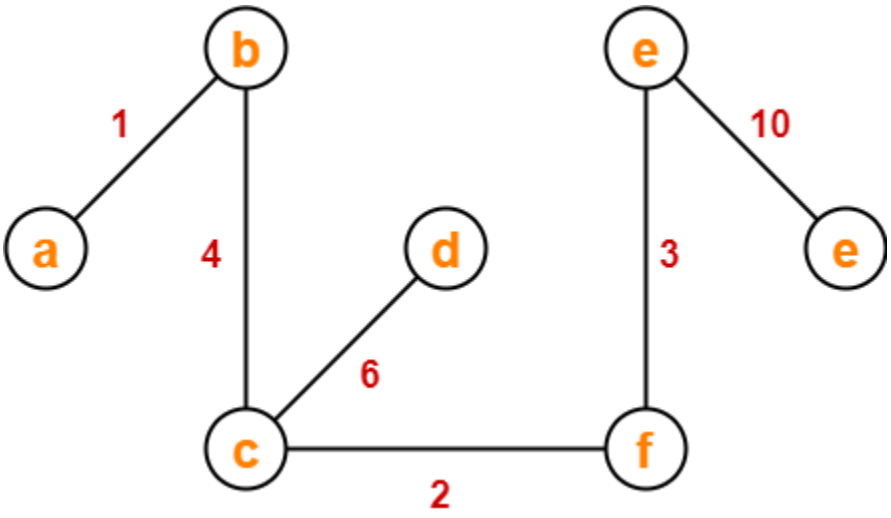
Problem-02:

Using Prim's Algorithm, find the cost of minimum spanning tree (MST) of the given graph-



Solution-

The minimum spanning tree obtained by the application of Prim's Algorithm on the given graph is as shown below-



Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

= 1 + 4 + 2 + 6 + 3 + 10

= 26 units

Kruskal's Algorithm-

- Kruskal's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Kruskal's algorithm, the given graph must be weighted, connected and undirected.

Kruskal's Algorithm Implementation-

The implementation of Kruskal's Algorithm is explained in the following steps-

Step-01:

- Sort all the edges from low weight to high weight.

Step-02:

- Take the edge with the lowest weight and use it to connect the vertices of graph.
- If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.

Step-03:

- Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

Thumb Rule to Remember

The above steps may be reduced to the following thumb rule-

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

Kruskal's Algorithm Time Complexity-

Worst case time complexity of Kruskal's Algorithm

= O(ElogV) or O(ElogE)

Analysis-

- The edges are maintained as min heap.
- The next edge can be obtained in O(logE) time if graph has E edges.
- Reconstruction of heap takes O(E) time.
- So, Kruskal's Algorithm takes O(ElogE) time.
- The value of E can be at most $O(V^2)$.
- So, O(logV) and O(logE) are same.

Special Case-

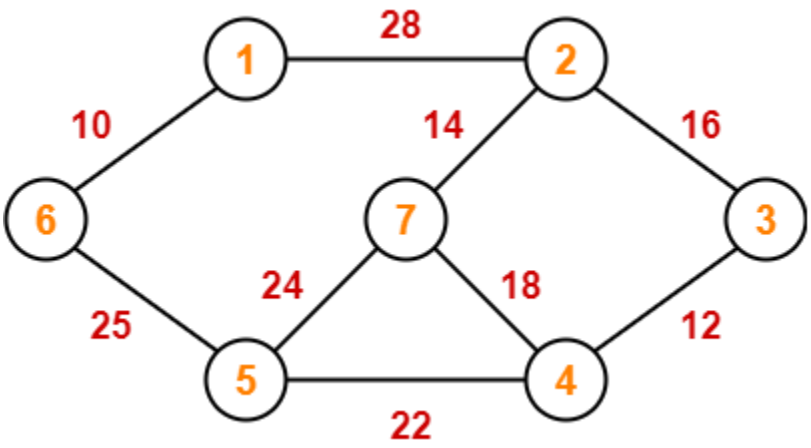
- If the edges are already sorted, then there is no need to construct min heap.
- So, deletion from min heap time is saved.
- In this case, time complexity of Kruskal's Algorithm = O(E + V)

Also Read- Prim's Algorithm

PRACTICE PROBLEMS BASED ON KRUSKAL'S ALGORITHM-

Problem-01:

Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm-

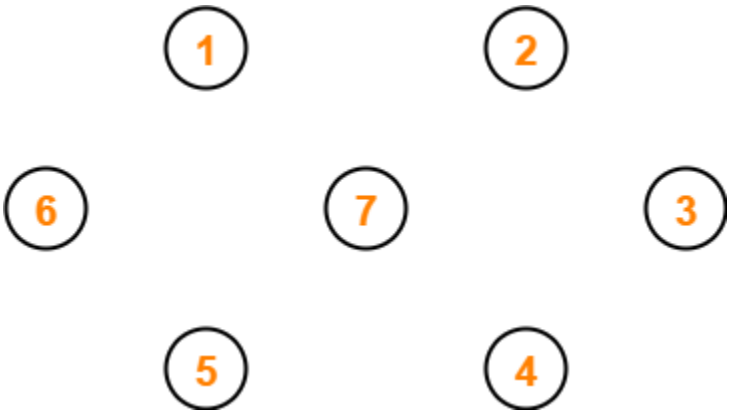


Solution-

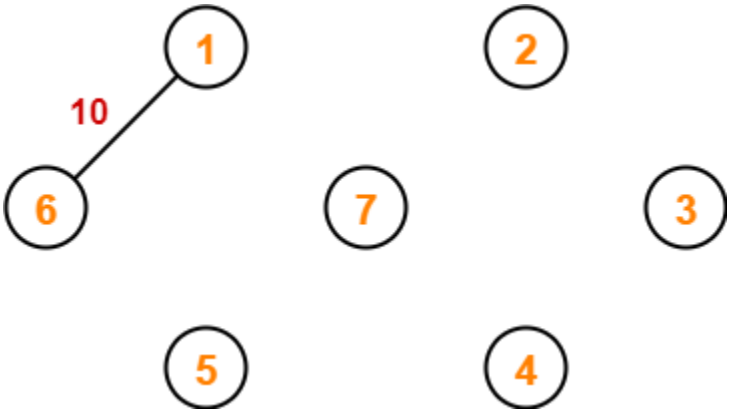
To construct MST using Kruskal's Algorithm,

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

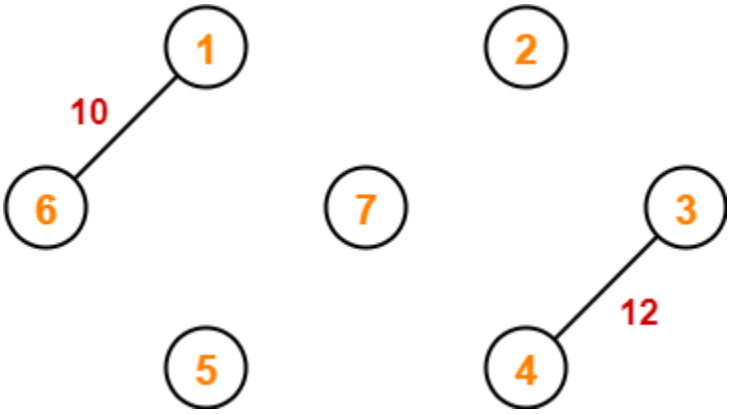
Step-01:



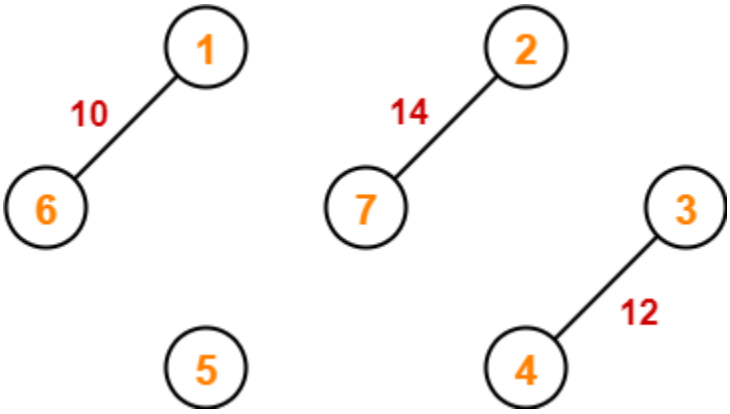
Step-02:



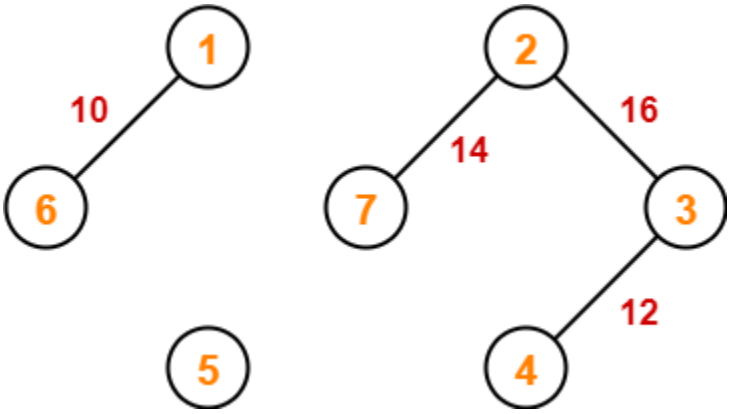
Step-03:



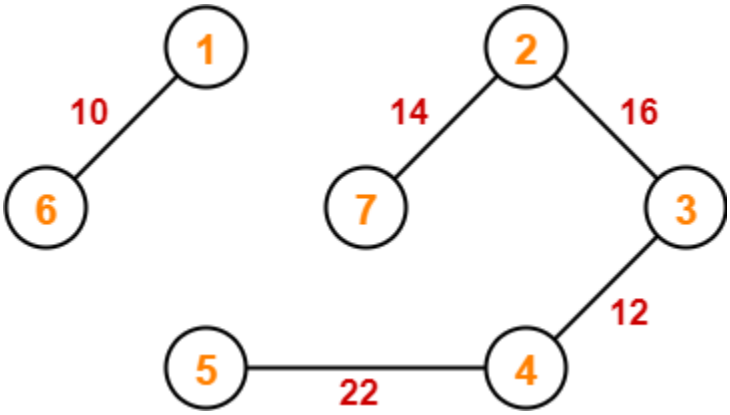
Step-04:



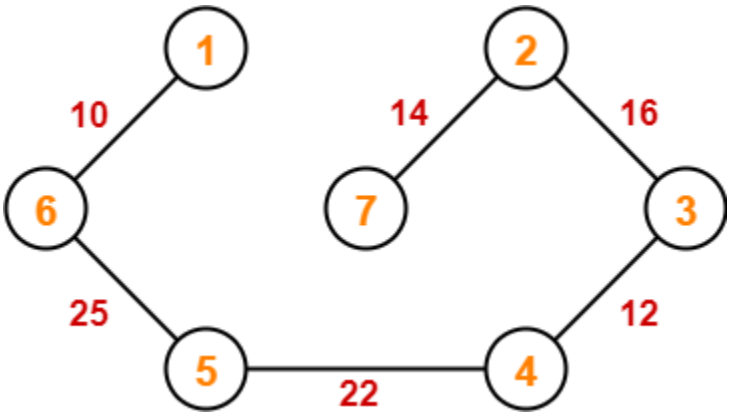
Step-05:



Step-06:



Step-07:



Since all the vertices have been connected / included in the MST, so we stop.

Weight of the MST

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

==========

IMP Points and Concepts:

Prim's and Kruskal's Algorithms-

Before you go through this article, make sure that you have gone through the previous articles on **Prim's Algorithm** & **Kruskal's Algorithm**.

We have discussed-

- Prim's and Kruskal's Algorithm are the famous greedy algorithms.
- They are used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply these algorithms, the given graph must be weighted, connected and undirected.

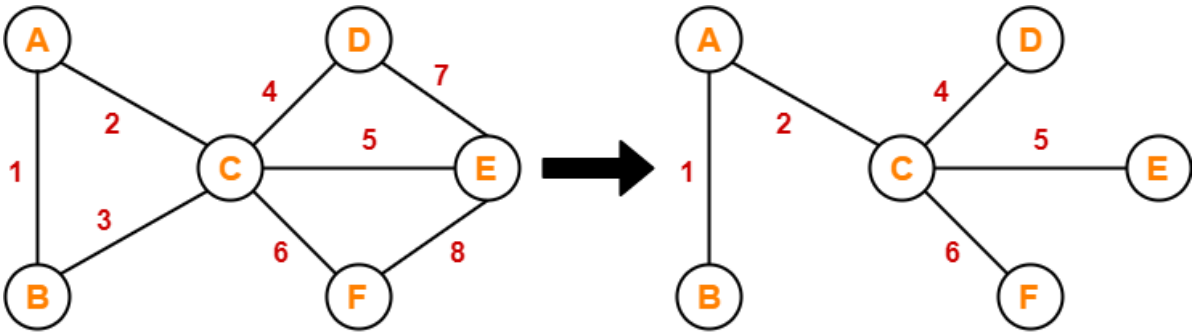
Some important concepts based on them are-

Concept-01:

If all the edge weights are distinct, then both the algorithms are guaranteed to find the same MST.

Example-

Consider the following example-



Given Graph

Minimum Spanning Tree (MST)

(Cost = 18 units)

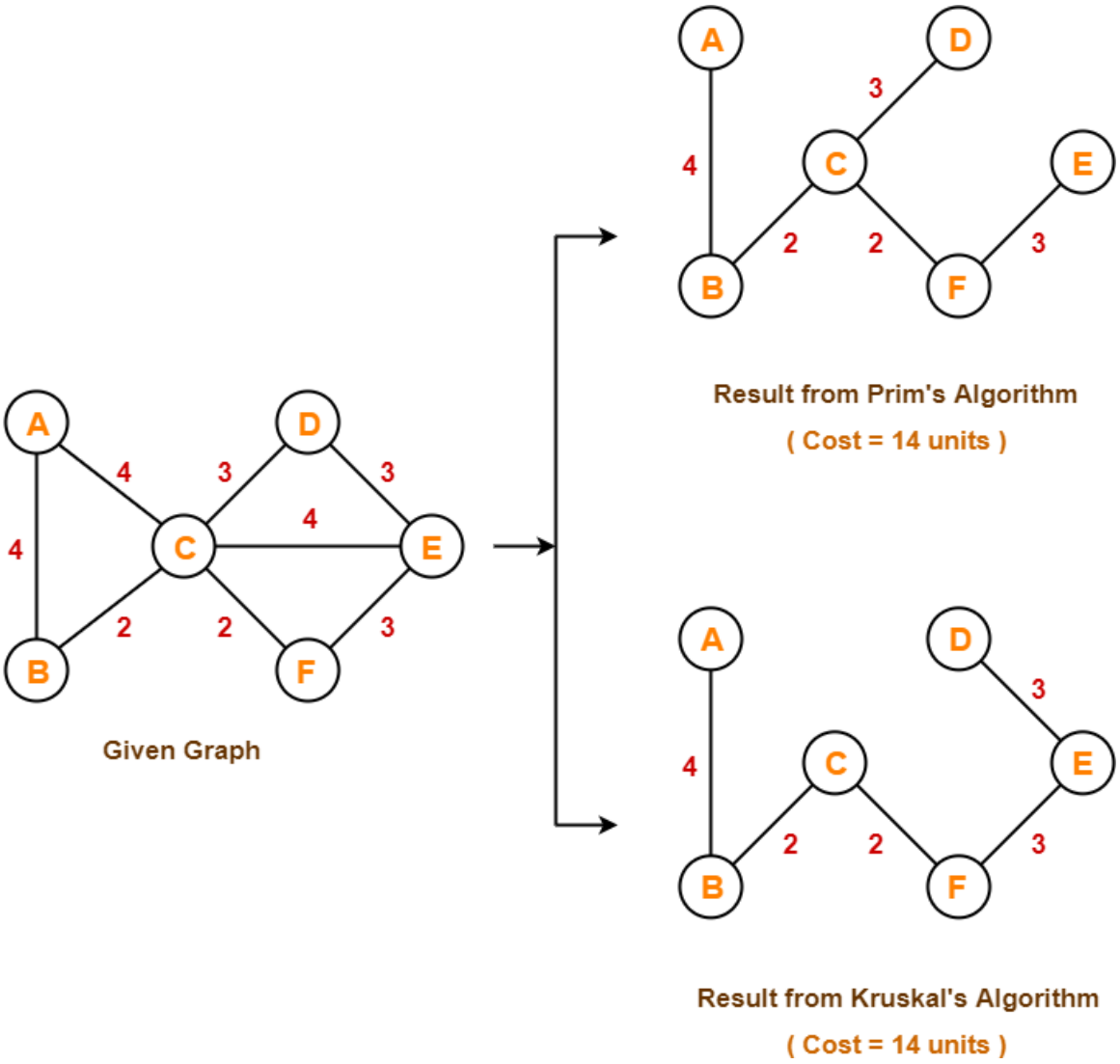
Here, both the algorithms on the above given graph produces the same MST as shown.

Concept-02:

- If all the edge weights are not distinct, then both the algorithms may not always produce the same MST.
- However, cost of both the MST_s would always be same in both the cases.

Example-

Consider the following example-



Here, both the algorithms on the above given graph produces different MST_s as shown but the cost is same in both the cases.

Concept-03:

Kruskal's Algorithm is preferred when-

- The graph is sparse.
- There are less number of edges in the graph like E = O(V)
- The edges are already sorted or can be sorted in linear time.

Prim's Algorithm is preferred when-

- The graph is dense.
- There are large number of edges in the graph like $E = O(V^2)$.

Concept-04:

Difference between Prim's Algorithm and Kruskal's Algorithm-

Prim's Algorithm	Kruskal's Algorithm
The tree that we are making or growing always remains connected.	The tree that we are making or growing usually remains disconnected.
Prim's Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest.
Prim's Algorithm is faster for dense graphs.	Kruskal's Algorithm is faster for sparse graphs.

Brute Force Algorithms

- A brute force algorithm solves a problem through exhaustion: it goes through all possible choices until a solution is found.
- The time complexity of a brute force algorithm is often proportional to the input size.

```
# pseudocode that prints all divisors of n by brute force

define printDivisors, n
    for all numbers from 1 to n
        if the number is a divisor of n
```

```
            print the number
```

Searching for smallest or largest value using linear search

Linear search can be used to search for the smallest or largest value in an unsorted list rather than searching for a match. It can do so by keeping track of the largest (or smallest) value and updating as necessary as the algorithm iterates through the dataset.

```
Create a variable called max_value_index
Set max_value_index to the index of the first element of the search list
  For each element in the search list
    if element is greater than the element at max_value_index
      Set max_value_index equal to the index of the element
return max_value_index
```

Linear Search best case

For a list that contains `n` items, the best case for a linear search is when the target value is equal to the first element of the list. In such cases, only one comparison is needed. Therefore, the best case performance is O(1).

Linear Search Complexity

Linear search runs in linear time and makes a maximum of `n` comparisons, where `n` is the length of the list. Hence, the computational complexity for linear search is O(N).

The running time increases, at most, linearly with the size of the items present in the list.

Linear Search expressed as a Function

A linear search can be expressed as a function that compares each item of the passed dataset with the target value until a match is found.

The given pseudocode block demonstrates a function that performs a linear search. The relevant index is returned if the target is found and -1 with a message that a value is not found if it is not.

```
For each element in the array
  if element equal target value then
    return its index
  if element is not found, return
    “Value Not Found” message
```

Return value of a linear search

A function that performs a linear search can return a message of success and the index of the matched value if the search can successfully match the target with an element of the dataset. In the event of a failure, a message as well as `-1` is returned as well.

```
For each element in the array
  if element equal target value then
    print success message
    return its index
  if element is not found
    print Value not found message
    return -1
```

Modification of linear search function

A linear search can be modified so that all instances in which the target is found are returned. This change can be made by not 'breaking' when a match is found.

```
For each element in the searchList
  if element equal target value then
    Add its index to a list of occurrences
if the list of occurrences is empty
  raise ValueError
otherwise
  return the list occurrences
```

Linear search

Linear search sequentially checks each element of a given list for the target value until a match is found. If no match is found, a linear search would perform the search on all of the items in the list.

For instance, if there are `n` number of items in a list, and the target value resides in the `n-5`th position, a linear search will check `n-5` items total.

Linear search as a part of complex searching problems

Despite being a very simple search algorithm, linear search can be used as a subroutine for many complex searching problems. Hence, it is convenient to implement linear search as a function so that it can be reused.

Linear Search Best and Worst Cases

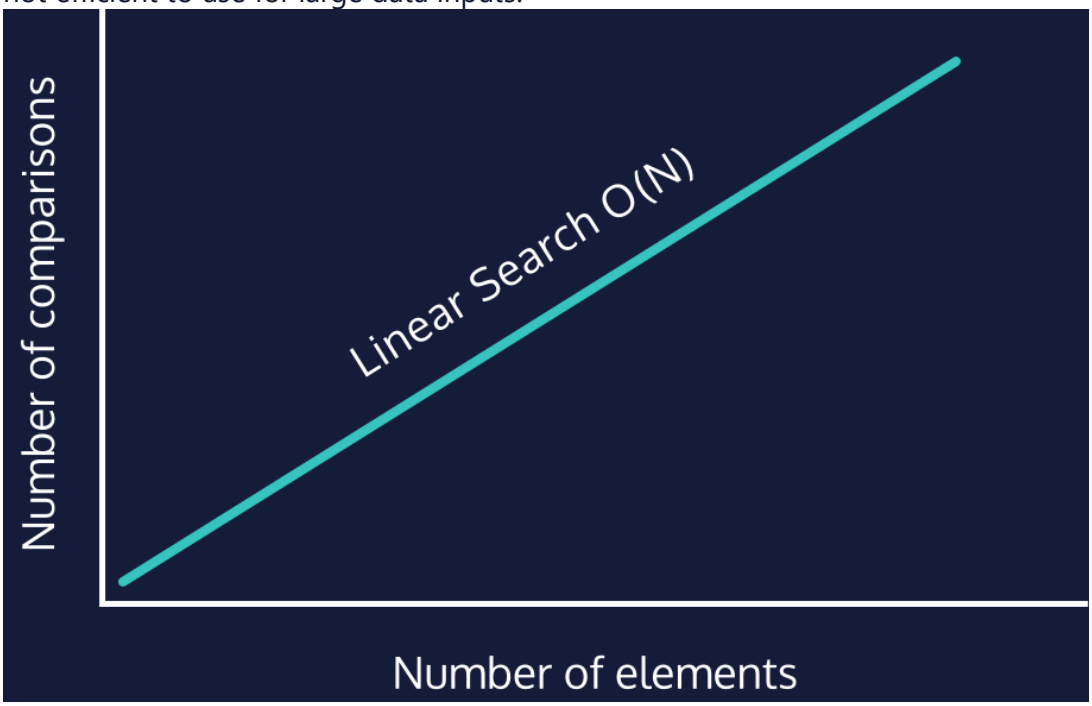
The best-case performance for the Linear Search algorithm is when the search item appears at the beginning of the list and is O(1). The worst-case performance is when the search item appears at the end of the list or not at all. This would require N comparisons, hence, the worse case is O(N).

Linear Search Average Runtime

The Linear Search Algorithm performance runtime varies according to the item being searched. On average, this algorithm has a Big-O runtime of O(N), even though the average number of comparisons for a search that runs only halfway through the list is N/2.

Linear Search Runtime

The Linear Search algorithm has a Big-O (worst case) runtime of O(N). This means that as the input size increases, the speed of the performance decreases linearly. This makes the algorithm not efficient to use for large data inputs.



Throwing Exception in Linear Search

The linear search function may throw a `ValueError` with a message when the target value is not found in the search list. Calling the linear search function inside a `try` block is recommended to catch the `ValueError` exception in the `except` block.

```
def linear_search(lst, match):
  for idx in range(len(lst)):
    if lst[idx] == match:
      return idx
    else:
      raise ValueError("{0} not in list".format(match))
```

```
recipe = ["nori", "tuna", "soy sauce", "sushi rice"]
ingredient = "avocado"
try:
  print(linear_search(recipe, ingredient))
except ValueError as msg:
  print("{0}".format(msg))
```

Find Maximum Value in Linear Search

The Linear Search function can be enhanced to find and return the maximum value in a list of numeric elements. This is done by maintaining a variable that is compared to every element and updated when its value is smaller than the current element.

```
def find_maximum(lst):
  max = None
  for el in lst:
    if max == None or el > max:
      max = el
  return max
test_scores = [88, 93, 75, 100, 80, 67, 71, 92, 90, 83]
print(find_maximum(test_scores)) # returns 100
```

Linear Search Multiple Matches

A linear search function may have more than one match from the input list. Instead of returning just one index to the matched element, we return a list of indices. Every time we encounter a match, we add the index to the list.

```
def linear_search(lst, match):
  matches = []
  for idx in range(len(lst)):
    if lst[idx] == match:
      matches.append(idx)
  if matches:
    return matches
  else:
    raise ValueError("{0} not in list".format(match))
```

```
scores = [55, 65, 32, 40, 55]
print(linear_search(scores, 55))
```

Raise Error in Linear Search

A Linear Search function accepts two parameters:

```
1) input list to search from
2) target element to search for in the input list
```

If the target element is found in the list, the function returns the element index. If it is not found, the function raises an error. When implementing in Python, use the `raise` keyword with `ValueError()`.

```
def linear_search(lst, match):
  for idx in range(len(lst)):
    if lst[idx] == match:
      return idx
```

What is Queue?

*A **Queue** is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.*

We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.

Queue Representation:
Like stacks, Queues can also be represented in an array: In this representation, the Queue is implemented using the array. Variables used in this case are

- **Queue:** the name of the array storing queue elements.
- **Front**: the index where the first element is stored in the array representing the queue.
- **Rear:** the index where the last element is stored in an array representing the queue.

Queue Data Structure

FIFO Principle of Queue:

- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).
- Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **front** of the queue(sometimes, **head** of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the **rear** (or the **tail**) of the queue. See the below figure.

Characteristics of Queue:

- Queue can handle multiple data.
- We can access both ends.
- They are fast and flexible.

Some common applications of Queue data structure :

1. **Task Scheduling**: Queues can be used to schedule tasks based on priority or the order in which they were received.
2. **Resource Allocation:** Queues can be used to manage and allocate resources, such as printers or CPU processing time.
3. **Batch Processing**: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.
4. **Message Buffering**: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.
5. **Event Handling**: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.
6. **Traffic Management**: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.

7. **Operating systems:** Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.
8. **Network protocols:** Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.
9. **Printer queues :**In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.
10. **Web servers:** Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.
11. **Breadth-first search algorithm:** The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.

Useful Applications of Queue

- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at the same rate as sent) between two processes. Examples include IO Buffers, pipes, etc.

Applications of Queue in Operating systems:

- Semaphores
- FCFS (first come first serve) scheduling, example: FIFO queue
- Spooling in printers
- Buffer for devices like keyboard
- CPU Scheduling
- Memory management

Applications of Queue in Networks:

- Queues in routers/ switches
- Mail Queues
- **Variations:** (Deque, Priority Queue, Doubly Ended Priority Queue)

Some other applications of Queue:

- Applied as waiting lists for a single shared resource like CPU, Disk, and Printer.
- Applied as buffers on MP3 players and portable CD players.
- Applied on Operating system to handle the interruption.
- Applied to add a song at the end or to play from the front.
- Applied on WhatsApp when we send messages to our friends and they don't have an internet connection then these messages are queued on the server of WhatsApp.
- Traffic software (Each light gets on one by one after every time of interval of time.)

Issues in applications of Queue :

1. **Queue overflow:** If a queue has a fixed size, it can become full, leading to a queue overflow. This can happen if elements are added to the queue faster than they are removed. To prevent overflow, some implementations use dynamic resizing or circular buffers.

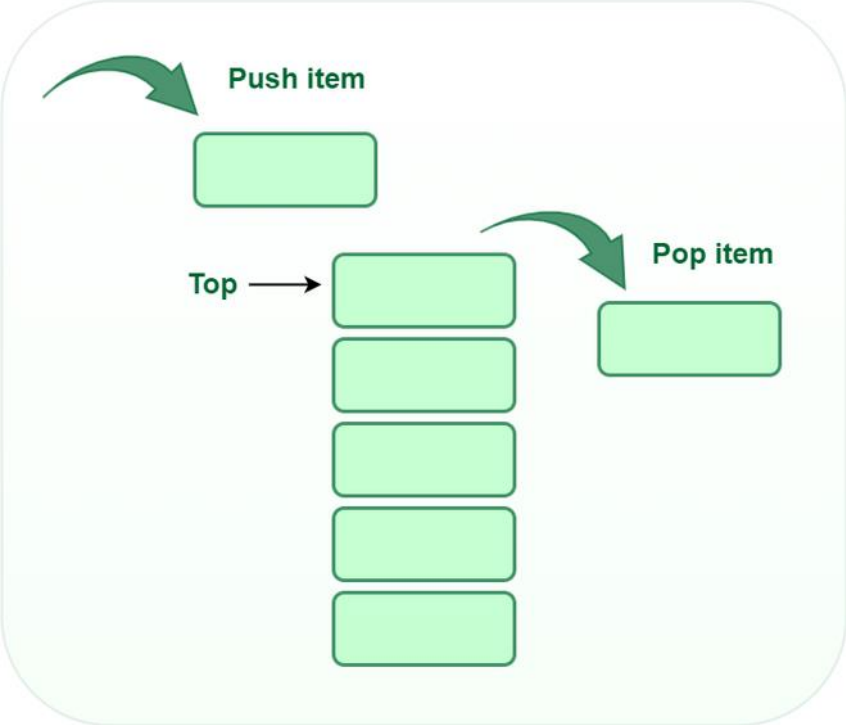
2. **Queue underflow:** If a queue is empty and an attempt is made to remove an element, this can lead to a queue underflow. This can happen if elements are removed from the queue faster than they are added. To prevent underflow, some implementations use sentinel values or null pointers to represent an empty queue.
3. **Blocking queues:** In some applications, a queue may become blocked if it is full or empty. This can cause delays in processing or deadlock. To address this, some implementations use bounded queues or non-blocking queues.
4. **Priority inversion:** In some applications, a higher priority element can get stuck behind a lower priority element in the queue. This can lead to priority inversion and result in reduced performance. To prevent this, some implementations use priority queues or multiple queues with different priorities.
5. **Synchronization issues:** In concurrent applications, multiple threads may access the same queue simultaneously. This can lead to synchronization issues like race conditions, deadlocks, and livelocks. To address this, some implementations use locking mechanisms like mutexes or semaphores.
6. **Memory management:** In some implementations, a queue may allocate and deallocate memory frequently, leading to memory fragmentation and reduced performance. To address this, some implementations use memory pools or pre-allocated buffers.

What is Stack?

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.

In order to make manipulations in a stack, there are certain operations provided to us for Stack, which include:

- **push()** to insert an element into the stack
- **pop()** to remove an element from the stack
- **top()** Returns the top element of the stack.
- **isEmpty()** returns true if the stack is empty else false.
- **size()** returns the size of the stack.



Application of Stack Data Structure:

- **Function calls and recursion:** When a function is called, the current state of the program is pushed onto the stack. When the function returns, the state is popped from the stack to resume the previous function's execution.
- **Undo/Redo operations:** The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To undo the action, the top element of the stack is popped, and the reverse operation is performed.
- **Expression evaluation:** Stack data structure is used to evaluate expressions in infix, postfix, and prefix notations. Operators and operands are pushed onto the stack, and operations are performed based on the stack's top elements.
- **Browser history:** Web browsers use stacks to keep track of the web pages you visit. Each time you visit a new page, the URL is pushed onto the stack, and when you hit the back button, the previous URL is popped from the stack.
- **Balanced Parentheses:** Stack data structure is used to check if parentheses are balanced or not. An opening parenthesis is pushed onto the stack, and a closing parenthesis is popped from the stack. If the stack is empty at the end of the expression, the parentheses are balanced.
- **Backtracking Algorithms:** The backtracking algorithm uses stacks to keep track of the states of the problem-solving process. The current state is pushed onto the stack, and when the algorithm backtracks, the previous state is popped from the stack.

Application of Stack in real life:

- CD/DVD stand.

- Stack of books in a book shop.
- Call center systems.
- Undo and Redo mechanism in text editors.
- The history of a web browser is stored in the form of a stack.
- Call logs, E-mails, and Google photos in any gallery are also stored in form of a stack.
- YouTube downloads and Notifications are also shown in LIFO format(the latest appears first).
- Allocation of memory by an operating system while executing a process.

Advantages of Stack:

- **Easy implementation:** Stack data structure is easy to implement using arrays or linked lists, and its operations are simple to understand and implement.
- **Efficient memory utilization**: Stack uses a contiguous block of memory, making it more efficient in memory utilization as compared to other data structures.
- **Fast access time:** Stack data structure provides fast access time for adding and removing elements as the elements are added and removed from the top of the stack.
- **Helps in function calls:** Stack data structure is used to store function calls and their states, which helps in the efficient implementation of recursive function calls.
- **Supports backtracking:** Stack data structure supports backtracking algorithms, which are used in problem-solving to explore all possible solutions by storing the previous states.
- **Used in Compiler Design:** Stack data structure is used in compiler design for parsing and syntax analysis of programming languages.
- **Enables undo/redo operations**: Stack data structure is used to enable undo and redo operations in various applications like text editors, graphic design tools, and software development environments.

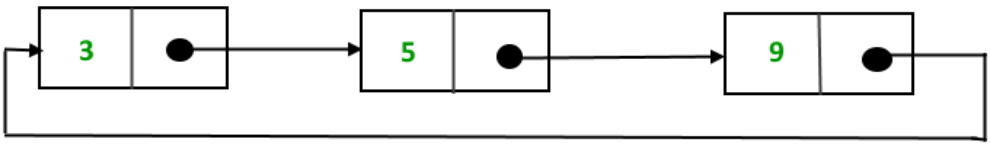
Disadvantages of Stack:

- **Limited capacity:** Stack data structure has a limited capacity as it can only hold a fixed number of elements. If the stack becomes full, adding new elements may result in stack overflow, leading to the loss of data.
- **No random access:** Stack data structure does not allow for random access to its elements, and it only allows for adding and removing elements from the top of the stack. To access an element in the middle of the stack, all the elements above it must be removed.
- **Memory management:** Stack data structure uses a contiguous block of memory, which can result in memory fragmentation if elements are added and removed frequently.
- **Not suitable for certain applications:** Stack data structure is not suitable for applications that require accessing elements in the middle of the stack, like searching or sorting algorithms.

- **Stack overflow and underflow**: Stack data structure can result in stack overflow if too many elements are pushed onto the stack, and it can result in stack underflow if too many elements are popped from the stack.
- **Recursive function calls limitations:** While stack data structure supports recursive function calls, too many recursive function calls can lead to stack overflow, resulting in the termination of the program.

Circular linked list

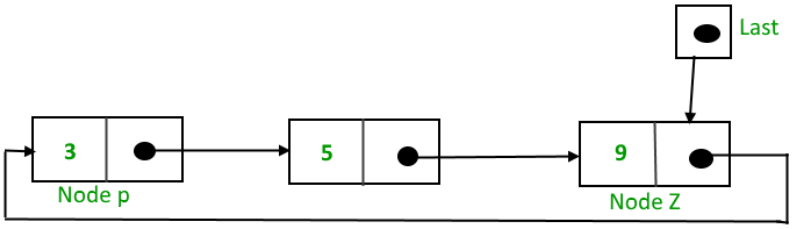
In a singly linked list, for accessing any node of the linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of a singly linked list. In a singly linked list, the next part (pointer to the next node) of the last node is NULL. If we utilize this link to point to the first node, then we can reach the preceding nodes.



.

Implementation of circular linked list:

To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then last -> next will point to the first node.



The pointer last points to node Z and last -> next points to node P.

Why have we taken a pointer that points to the last node instead of the first node?

For the insertion of a node at the beginning, we need to traverse the whole list. Also, for insertion at the end, the whole list has to be traversed. If instead of the ***start*** pointer, we take a pointer to the last node, then in both cases there won't be any need to traverse the whole list. So insertion at the beginning or at the end takes constant time, irrespective of the length of the list.

Insertion in a circular linked list:

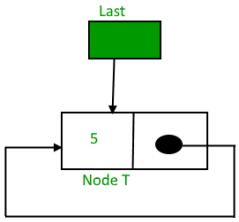
A node can be added in three ways:

- Insertion in an empty list
- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

Insertion in an empty List:

Initially, when the list is empty, the *last* pointer will be NULL.

After inserting node T,



After insertion, T is the last node, so the pointer *last* points to node T. And Node T is the first and the last node, so T points to itself.

```
// Javascript program for the above operation
function addToEmpty(last , data)
{
        // This function is only for empty list
        if (last != null)
        return last;

        // Creating a node dynamically.
        var temp = new Node();
```

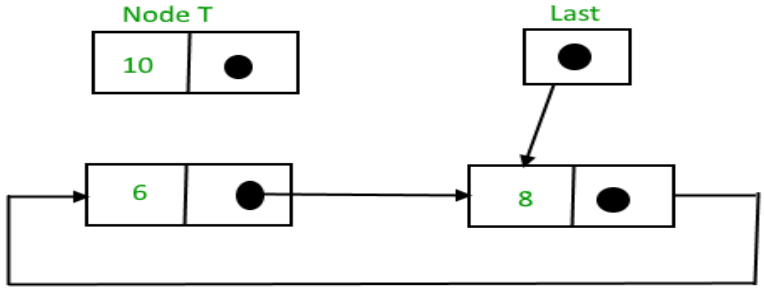
```
        // Assigning the data.
        temp.data = data;
        last = temp;
        // Note : list was empty. We link single node
        // to itself.
        temp.next = last;

        return last;
    }

    // This code contributed by umadevi9616
```

Time Complexity: O(1), As we have to perform constant number of operations.
Auxiliary Space: O(1), As constant extra space is used.

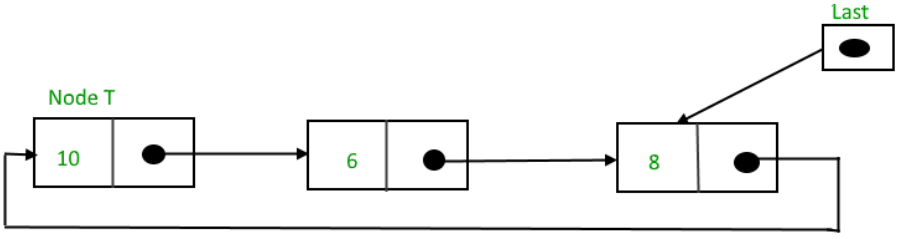
Insertion at the beginning of the list

To insert a node at the beginning of the list, follow these steps:

- Create a node, say T
- Make T -> next = last -> next
- last -> next = T



After insertion,



Below is the implementation of the above operation:
// Javascript program for the above operation

```
function addBegin(last , data)
```

```
{
        if (last == null)
        return addToEmpty(last, data);

        // Creating a node dynamically.
        var temp = new Node();

        // Assigning the data.
        temp.data = data;

        // Adjusting the links.
        temp.next = last.next;
        last.next = temp;
        return last;
}
```

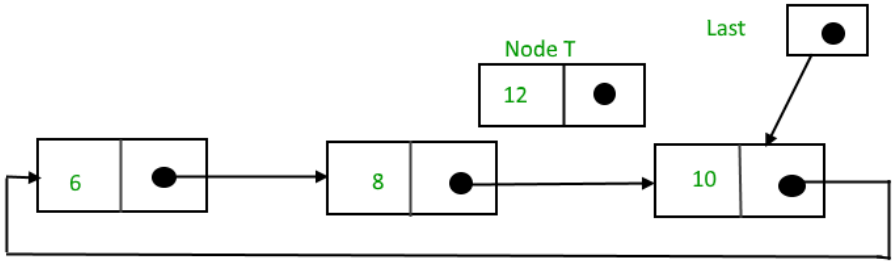
Time Complexity: O(1)
Auxiliary Space: O(1)

Insertion in between the nodes

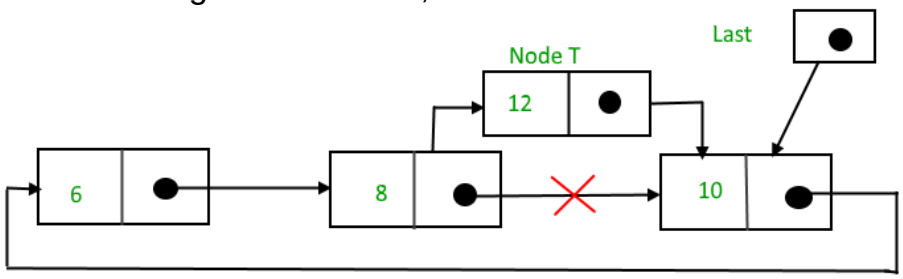
To insert a node in between the two nodes, follow these steps:

- Create a node, say T.
- Search for the node after which T needs to be inserted, say that node is P.
- Make T -> next = P -> next;
- P -> next = T.

Suppose 12 needs to be inserted after the node that has the value 8,



After searching and insertion,



Below is the implementation of the above operation:

```
// Javascript program for the above operation

function addAfter(last, data, item) {
        if (last == null) return null;

        var temp, p;
        p = last.next;
        do {
        if (p.data == item) {
                temp = new Node();
                temp.data = data;
                temp.next = p.next;
                p.next = temp;

                if (p == last) last = temp;
                return last;
        }
        p = p.next;
        } while (p != last.next);

        document.write(item + " not present in the list. <br>");
        return last;
}
```

Time Complexity: O(N)
Auxiliary Space: O(1)

Below is a complete program that uses all of the above methods to create a circular singly linked list.

```
class Node {
        constructor() {
                this.data = 0;
                this.next = null;
        }
}
```

```
function addToEmpty(last, data) {
        // This function is only for empty list
        if (last != null)
            return last;

        // Creating a node dynamically.
        var temp = new Node();

        // Assigning the data.
        temp.data = data;
        last = temp;

        // Creating the link.
        last.next = last;
        return last;
}

function addBegin(last, data) {
        if (last == null)
        return addToEmpty(last, data);

        var temp = new Node();
        temp.data = data;
        temp.next = last.next;
        last.next = temp;
        return last;
}

function addEnd(last, data) {
        if (last == null)
         return addToEmpty(last, data);

        var temp = new Node();
        temp.data = data;
        temp.next = last.next;
```

```
        last.next = temp;
        last = temp;
        return last;
}

function addAfter(last, data, item) {
        if (last == null)
        return null;

        var temp, p;
        p = last.next;
        do {
                if (p.data == item) {
                        temp = new Node();
                        temp.data = data;
                        temp.next = p.next;
                        p.next = temp;
                        if (p == last) last = temp;
                        return last;
                }
                p = p.next;
        } while (p != last.next);
        document.write(item + " not present in the list. <br>");
        return last;
}

function traverse(last) {
        var p;
        // If list is empty, return.
        if (last == null) {
                document.write("List is empty.<br>");
                return;
        }
        // Pointing to first Node of the list.
        p = last.next;
```

```
        // Traversing the list.
        do {
            document.write(p.data + " ");
            p = p.next;
        } while (p != last.next);
    }

    // Driver code
    var last = null;
    last = addToEmpty(last, 6);

    last = addBegin(last, 4);
    last = addBegin(last, 2);
    last = addEnd(last, 8);
    last = addEnd(last, 12);
    last = addAfter(last, 10, 8);
    traverse(last);
```

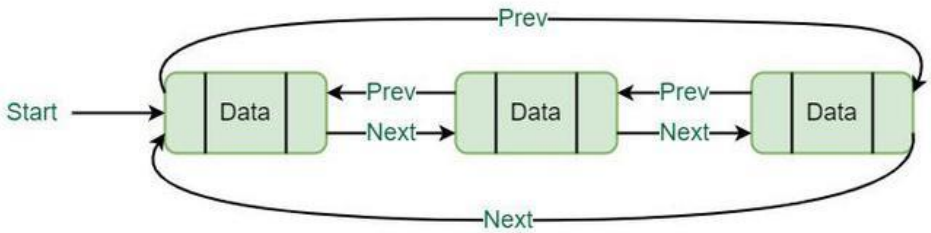
Time Complexity: O(N)
Auxiliary Space: O(1), as we are not using any extra space.

==========

Doubly Circular Linked List :

Insertion in Doubly Circular Linked List

Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.



// JavaScript program to illustrate inserting a Node in
// a Circular Doubly Linked list in begging, end and middle

```
var start = null;
```

// Structure of a Node

```
class Node {
   constructor() {
        this.data = 0;
        this.next = null;
        this.prev = null;
  }
}
```

// Function to insert at the end

```
function insertEnd(value) {
var new_node;
```

// If the list is empty, create a single node
// circular and doubly list

```
if (start == null) {
        new_node = new Node();
        new_node.data = value;
        new_node.next = new_node.prev = new_node;
        start = new_node;
        return;
}

// If list is not empty

/* Find last node */
var last = start.prev;

// Create Node dynamically
new_node = new Node();
new_node.data = value;

// Start is going to be next of new_node
new_node.next = start;

// Make new node previous of start
start.prev = new_node;

// Make last previous of new node
new_node.prev = last;

// Make new node next of old last
last.next = new_node;
}

// Function to insert Node at the beginning
// of the List,

function insertBegin(value) {
// Pointer points to last Node
```

```
var last = start.prev;

var new_node = new Node();
new_node.data = value; // Inserting the data

// setting up previous and next of new node
new_node.next = start;
new_node.prev = last;

// Update next and previous pointers of start
// and last.
last.next = start.prev = new_node;

// Update start pointer
start = new_node;
}

// Function to insert node with value as value1.
// The new node is inserted after the node with
// with value2

function insertAfter(value1, value2) {
var new_node = new Node();
new_node.data = value1; // Inserting the data

// Find node having value2 and next node of it
var temp = start;
while (temp.data != value2) temp = temp.next;
var next = temp.next;

// insert new_node between temp and next.
temp.next = new_node;
new_node.prev = temp;
new_node.next = next;
next.prev = new_node;
```

```
}

function display() {
var temp = start;

document.write("<br>Traversal in forward direction <br>");
while (temp.next != start) {
        document.write(temp.data + " ");
        temp = temp.next;
}
document.write(temp.data);

document.write("<br>Traversal in reverse direction <br>");
var last = start.prev;
temp = last;
while (temp.prev != last) {
        document.write(temp.data + " ");
        temp = temp.prev;
}
document.write(temp.data);
}

/* Driver code*/
/* Start with the empty list */
var start = null;

// Insert 5. So linked list becomes 5.null
insertEnd(5);

// Insert 4 at the beginning. So linked
// list becomes 4.5
insertBegin(4);

// Insert 7 at the end. So linked list
// becomes 4.5.7
```

```
insertEnd(7);

// Insert 8 at the end. So linked list
// becomes 4.5.7.8
insertEnd(8);

// Insert 6, after 5. So linked list
// becomes 4.5.6.7.8
insertAfter(6, 5);

document.write("Created circular doubly linked list is: ");
display();
```

==========

Queue Data Structure :

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
2. Queue is referred to be as First In First Out list.
3. For example, people waiting in line for a rail ticket form a queue.

Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.

2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.
6.

Complexity:

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Types of Queue

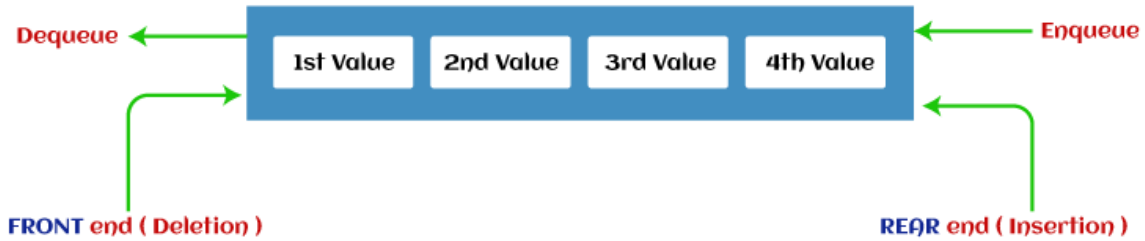
In this article, we will discuss the types of queue. But before moving towards the types, we should first discuss the brief introduction of the queue.

What is a Queue?

Queue is the data structure that is similar to the queue in the real world. A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Queue can also be defined as the list or collection in which the insertion is done from one end known as the **rear end** or the **tail** of the queue, whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the last person enters in the queue gets the ticket at last. Similar approach is followed in the queue in data structure.

The representation of the queue is shown in the below image –



Now, let's move towards the types of queue.

Types of Queue

There are four different types of queue that are listed as follows -

- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Let's discuss each of the type of queue.

Simple Queue or Linear Queue

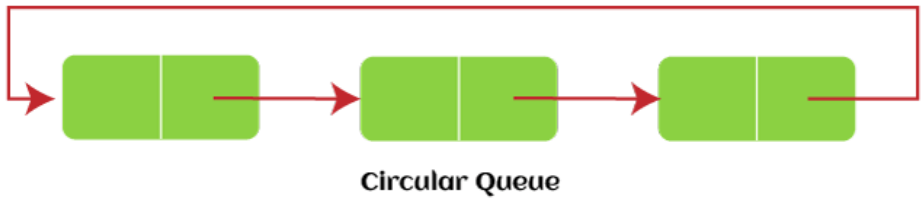
In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.

The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

To know more about the queue in data structure, you can click the link - https://www.javatpoint.com/data-structure-queue

Circular Queue

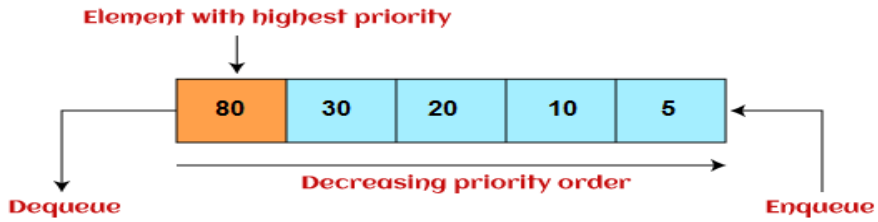
In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image –



The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.
To know more about the circular queue, you can click the link - https://www.javatpoint.com/circular-queue

Priority Queue

It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image -



Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.
There are two types of priority queue that are discussed as follows -

- **Ascending priority queue -** In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an

array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.

- **Descending priority queue -** In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

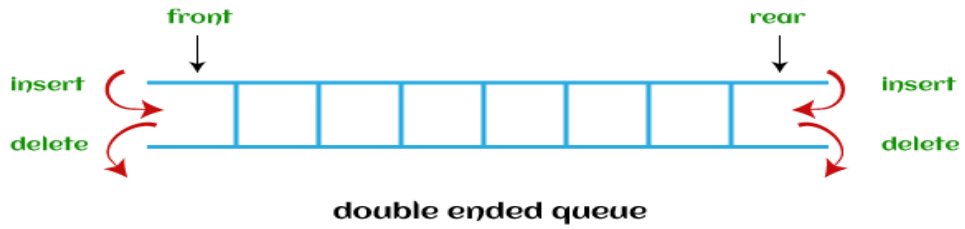
To learn more about the priority queue, you can click the link - https://www.javatpoint.com/ds-priority-queue

Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

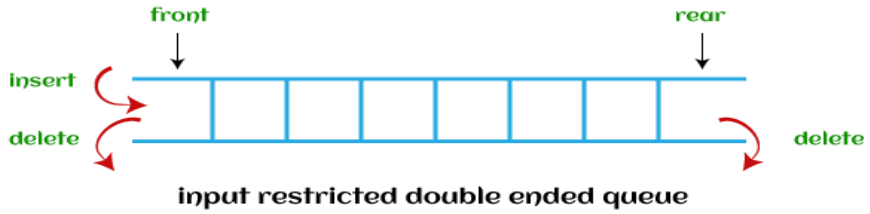
The representation of the deque is shown in the below image –



To know more about the deque, you can click the link - https://www.javatpoint.com/ds-deque

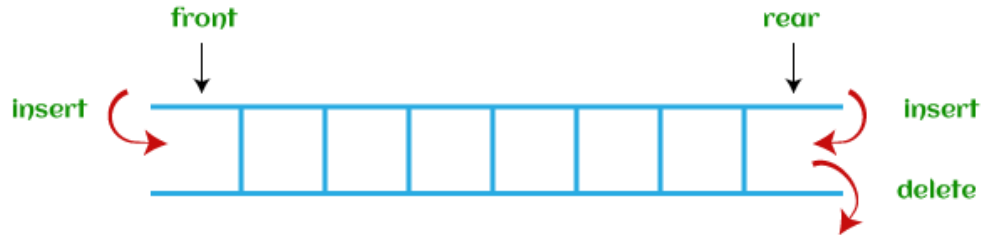
There are two types of deque that are discussed as follows -

- **Input restricted deque -** As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

- **Output restricted deque -** As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Now, let's see the operations performed on the queue.

Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

- **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
- **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

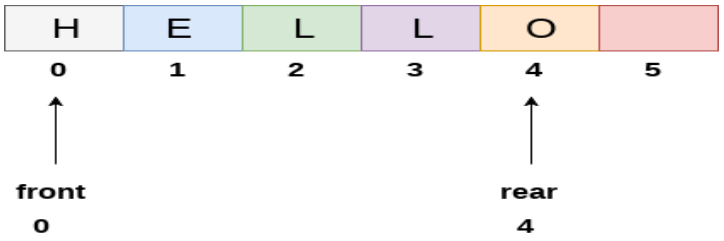
Now, let's see the ways to implement the queue.

Ways to implement the queue

There are two ways of implementing the Queue:

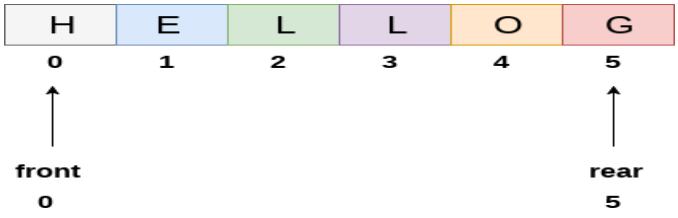
- **Implementation using array:** The sequential allocation in a Queue can be implemented using an array. **Implementation using Linked list:** The linked list allocation in a Queue can be implemented using a linked list.

Array representation of Queue
We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



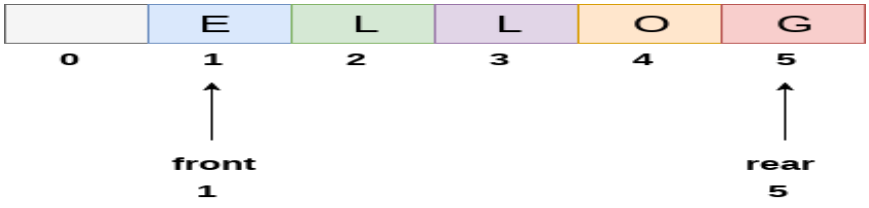
Queue

The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue
Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.
If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.
Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm:

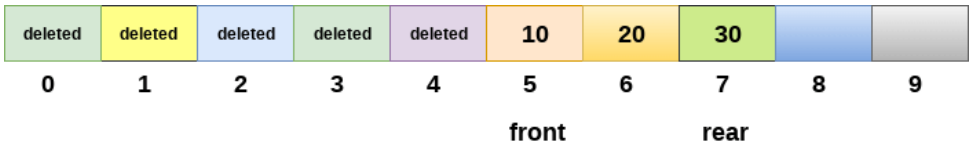
- **Step 1:** IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

Algorithm to delete an element from the queue:
If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.
Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.
Algorithm

- **Step 1:** IF FRONT = -1 or FRONT > REAR
  Write UNDERFLOW
  ELSE
  SET VAL = QUEUE[FRONT]
  SET FRONT = FRONT + 1
  [END OF IF]
- **Step 2:** EXIT

Drawback of array implementation
Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



limitation of array representation of queue

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

- **Deciding the array size**

On of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.