Greedy Approach

Greedy approach always makes the choice that looks best at the moment. That means it makes locally optimal choice in the hope that this choice will lead to global optimal solution.

Greedy algorithm does not yield optimal solution always. However, for some problems, greedy approach gives optimal solution.

Problems that can be solved using greedy approach

- Minimum Spanning Tree: (Prim, Kruskal)
- Activity Selection Problem
- Fractional Knapsack problem
- Huffman Code
- Single source shortest path problems(Dijkstra alg, Bellman-ford)

Difference Between Greedy Approach and Dynamic Programming

Dynamic programming is efficient in terms of time, but it requires more work for a simple problem	Greedy approach is efficient for a particular set of problem.
Dynamic programming makes global optimal choice	Greedy approach makes locally optimal choice
Dynamic programming grows in bottom up approach	Greedy follows top down approach

Minimum Spanning Tree(MST)

Spanning tree of a graph(G) is a sub graph(S) which is a tree, containing all the vertices of graph G. A graph may have several spanning tree. The spanning tree, whose sum of cost of all the edges are minimum is called minimum spanning tree.

GenericMST(G,W)

1. S = Ø

2. **while** S does not form a spanning tree

3. find an edge (u,v) that is **safe** for S

4. S = S U (u,v)

5. return S

Safe edge : The edge, which is light weight and by adding it in partial MST ,it does not form a cycle. This is called safe edge.

Kruskal's Algorithm for MST

In this algorithm, the spanning tree (S) starts from a forest. the edges are selected in such a manner that it contains smallest weight and upon adding it to the partial spanning tree set S, it does not form cycle.

Kruskal's algorithm uses disjoint set forest data structure to find minimum spanning tree of a graph.

Three operations of Disjoint data structure

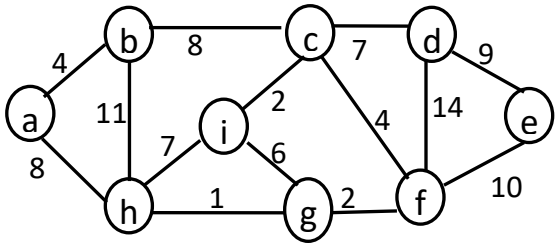
- MAKESET(x) : create new set with x as the only member(hence the representative)
- FINDSET(x) : returns the representative element from the set containing x.
- UNION(x,y) : Unites two set containing x and y.

```
MST_Kruskal(G,W)

// This algorithm finds a minimum cost of spanning tree
//(S) from given graph G(U,V) which is represented by
//the weighted matrix W.

1. S = Ø
2. for each vertex u in V[G]
3.    MAKESET(u)
4. sort all the edges in E in non-decreasing order by weights
5. for each edge (u,v) in E        // taken in order
6.    if FINDSET(u) ≠ FINDSET(v)
7.         S = S U (u,v)
8.         UNION(u,v)
```

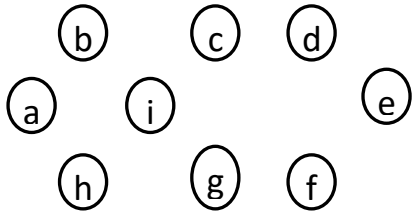
Problem: *Using Kruskal's algorithm, Find the cost of MST in following graph.*



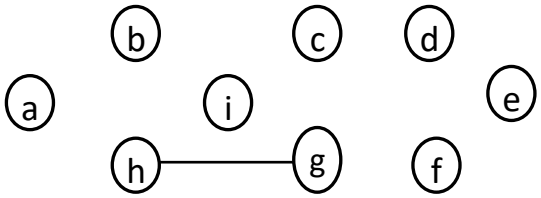
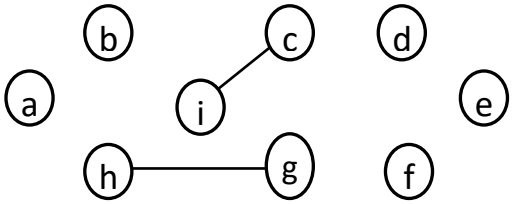
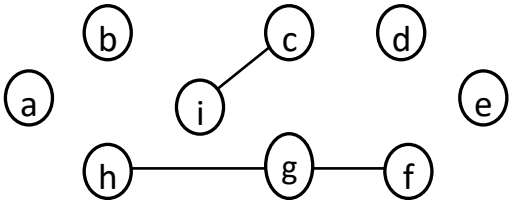
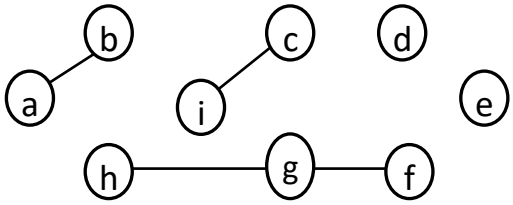
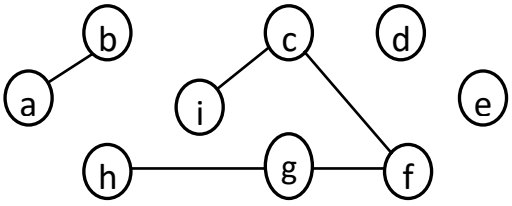
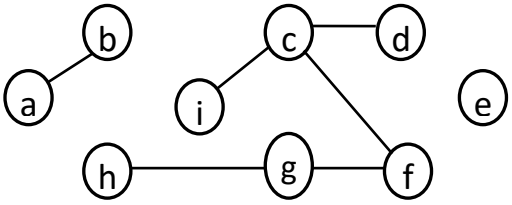
Solution

Initial Configuration: Create a forest i,e one set for each node

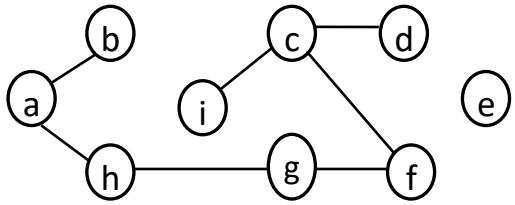
{a} {b} {c} {d} {e} {f} {g} {h} {i}



Sorted E ={ (g,h), (c,i), (f,g), (a,b), (c,f), (g,i),(c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f) }

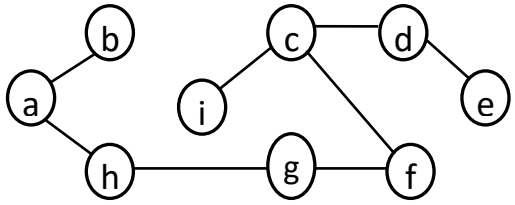
Edge selected	Disjoint set forest	Partial MST
(g,h)	{a} {b} {c} {d} {e} {f} {g,h} {i}	
(c,i)	{a} {b} {c,i} {d} {e} {f} {g,h}	
(f,g)	{a} {b} {c,i} {d} {e} {f,g,h}	
(a,b)	{a,b} {c,i} {d} {e} {f,g,h}	
(c,f)	{a,b} {c,f,g,h,i} {d} {e}	
(g,i)	Not Safe	
(c,d)	{a, b} {c, d, f, g,h,i} {e}	
(h,i)	Not Safe	

(a,h) {a, b, c, d, f, g, h, i} {e}



(b,c) Not Safe

(d,e) {a, b, c, d, f, g, h, i, e}



(e,f) Not Safe
(b,h) Not Safe
(d,f) Not Safe

Now the obtained tree is a minimum cost of spanning tree. The edges in spanning tree set S are { (g,h), (c,i), (f,g), (a,b), (c,f), (c,d), (a,h),(d,e) }. To find the cost of MST, add the weight of all the edges in the spanning tree.

Cost = 1 + 2 +2 + 4 + 4 + 7 + 8 + 9 = 37.

Analysis of Kruskal's Algorithm

Each makeset() takes θ(1) time. There are |V| number of call s to makeset() in the loop in line number 2.So altogether it takes θ(V) times

Line number 4 sorts the set E edges , takes θ(ElogE) times.

In loop from line number 5 to 8, there are 2E number of findset() operation and |V|-1 union operations. Using efficient implementation of disjoint data structure, both findset and union operation altogether takes O(E α(V)) , where α is the very slowly growing function.Upper bound of α(V)) can be expressed by O(logV).

So total running time of kruskal's algorithm is O(V) + O(ElogE) + O(ElogV) = O(ElogE). As logE is O(logV), the running time can also be expressed as O(ElogV)

Prim's Algorithm for MST

In this algorithm, edges in set S always form a single tree. The tree starts from any arbitrary root vertex and grows by adding a lightweight edge at each step, until the tree spans all the vertices.

It uses two data structures key and p.

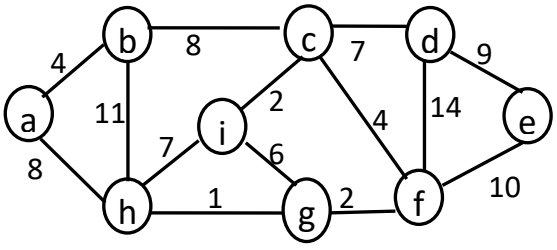
- key[u] contains the minimum weight to reach vertex u
- p[u] contains the predecessor vertex of vertex u.

```
MST_PRIM(G,W,r)
```

//the graph G=(U,V) is a connected graph with weighted matrix W and a root //vertex r. The algorithm finds a spanning tree that has minimum cost.

```
1. for each vertex u ∈ V[G]
2.      key[u] =  ∞
3.      p[u]= NIL
4. key[r] = 0
5. Q = V[G] //Insert all vertices in  Priority Queue Q
6. while Q ≠ϕ
7.      u=EXTRACT_MIN(Q)//Delete a vertex with smallest key from Q
8.      for each vertex v ∈ Adj[u]
9.          if v ∈ Q and W(u,v) < key[v]
10.             key[v] = W[u,v]
11.             p[v] = u
```

Problem: Using Kruskal's algorithm, Find the cost of MST in following graph.

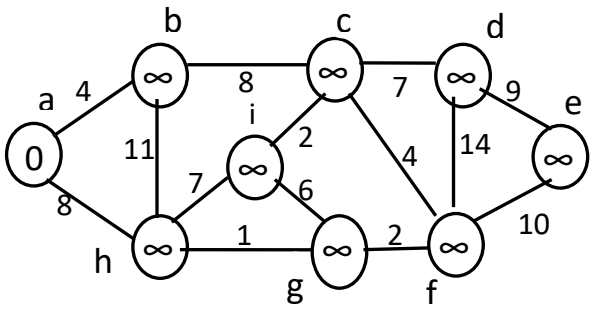


Solution:

Initial configuration (line number 1 to 5)

	a	b	c	d	e	f	g	h	i
Key	0	∞	∞	∞	∞	∞	∞	∞	∞
p	N	N	N	N	N	N	N	N	N

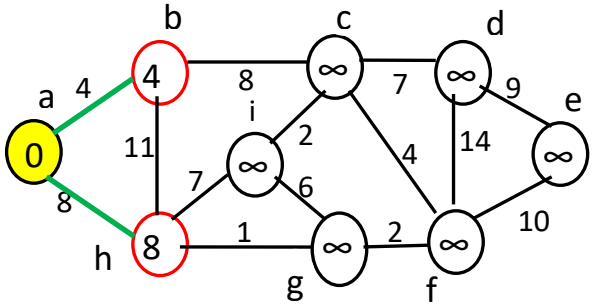
Queue: { a, b, c, d, e, f, g, h, i}



step-1:Extract a, Update b,h

	a	b	c	d	e	f	g	h	i
Key	0	4	∞	∞	∞	∞	∞	8	∞
p	N	a	N	N	N	N	N	a	N

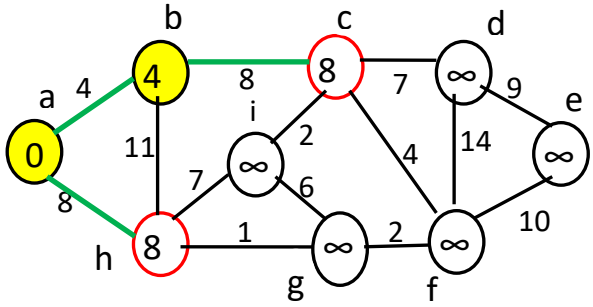
Queue: { b, c, d, e, f, g, h, i}



Step-2:Extract b, Update c,h

	a	b	c	d	e	f	g	h	i
Key	0	4	8	∞	∞	∞	∞	8	∞
p	N	a	b	N	N	N	N	a	N

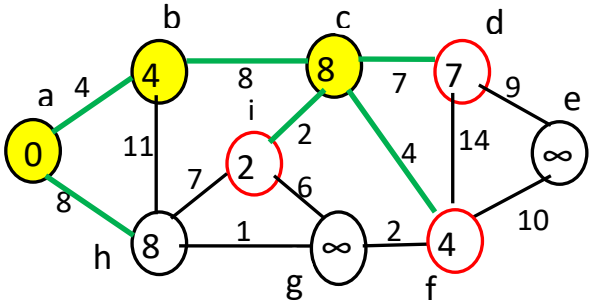
Queue: { c, d, e, f, g, h, i}



Step-3:Extract c, Update d,f,i

	a	b	c	d	e	f	g	h	i
Key	0	4	8	7	∞	4	∞	8	2
p	N	a	b	c	N	c	N	a	c

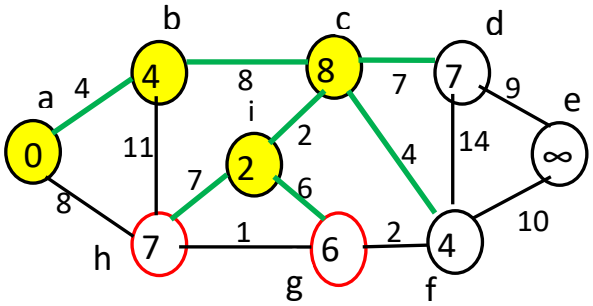
Queue: { d, e, f, g, h, i}



Step-4: Extract i, Update g,h

	a	b	c	d	e	f	g	h	i
Key	0	4	8	7	∞	4	6	7	2
p	N	a	b	c	N	c	i	i	c

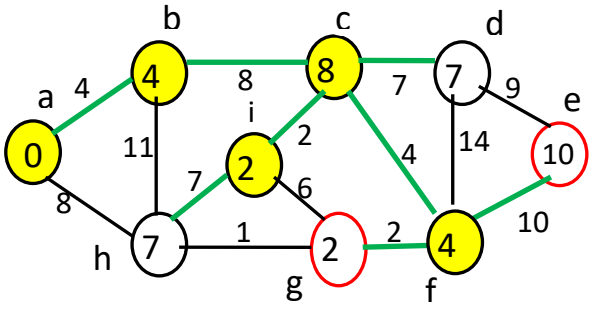
Queue: { d, e, f, g, h}



Step-5: Extract f , Update e,g

	a	b	c	d	e	f	g	h	i
Key	0	4	8	7	10	4	2	7	2
p	N	a	b	c	f	c	f	i	c

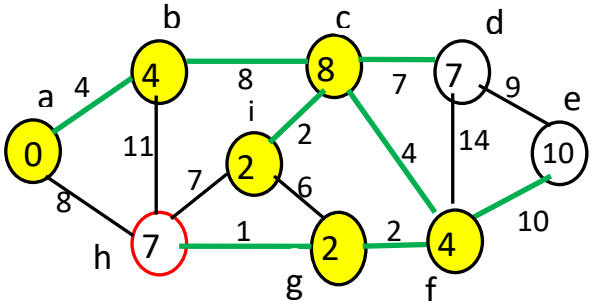
Queue: { d,e, g, h}



Step-6: Extract g , Update h

	a	b	c	d	e	f	g	h	i
Key	0	4	8	7	10	4	2	1	2
p	N	a	b	c	f	c	f	g	c

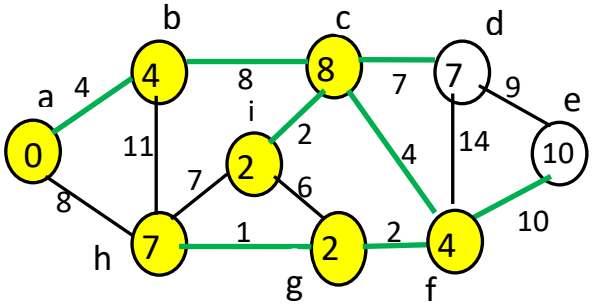
Queue: { d,e, h}



Step-7: Extract h , Update nothing

	a	b	c	d	e	f	g	h	i
Key	0	4	8	7	10	4	2	1	2
p	N	a	b	c	f	c	f	g	c

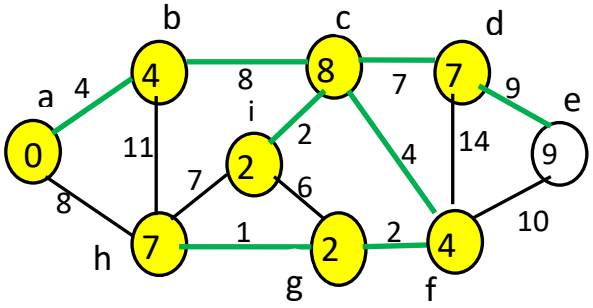
Queue: { d, e}



Step-8: Extract d , Update e

	a	b	c	d	e	f	g	h	i
Key	0	4	8	7	9	4	2	1	2
p	N	a	b	c	d	c	f	g	c

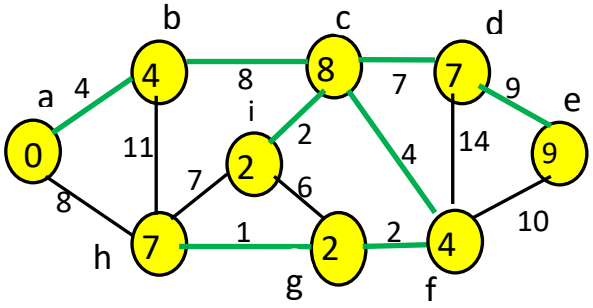
Queue: { e }

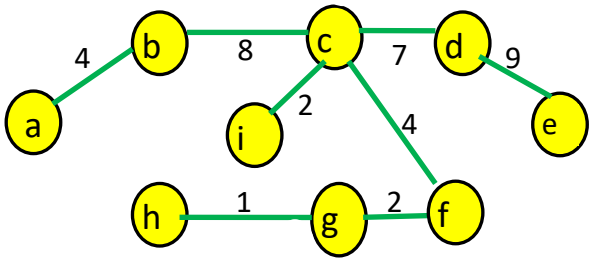


Step-8: Extract e , Update Nothing

	a	b	c	d	e	f	g	h	i
Key	0	4	8	7	9	4	2	1	2
p	N	a	b	c	d	c	f	g	c

Queue: { Nil }





Cost =4 + 8 + 7 + 4 +2 + 9 + 2 + 1

= 37

Analysis:

Line number 1 to 4 takes O(V) time to set the key and P array.

To insert |V| vertices in a min-priority Queue Q in line number 5 takes O(V) times.

Assuming Queue is implemented in Min-heap, each Extract_min() in line number 7 takes O(logV) tomes. So for a total of |V| such call it takes O(VlogV) times.

Loop in line number 8 is executed O(E) times altogether. Within this loop,Line number -11 updates the key[v] by decreasing the key of vertices in set V. As each decrease_key operation requires O(logV) times,and it is executed O(E) number of times, altogether it takes O(ElogV) times.

So total running time of Prim's Algorithm –

O(V) + O(VlogV) + O(ElogV) = O(ElogV)

Single Source shortest path problems

Given a directed weighted graph, shortest path problem finds a path from between source vertex and destination vertex, in which sum of weights of all the edges is minimum.

The objective of **Single Source shortest path problem** is to find a shortest path from a source vertex to every other vertex in the graph.

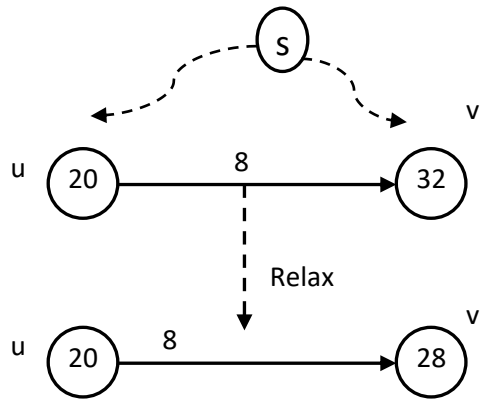
Dijkstra's Algorithm

Dijkstra's algorithm solves a single source shortest path problem on a weighted graph, where the edge weights are non-negative.

Dijkstra algorithm uses Following data structure for its implantation

- A Min Priority Queue
- Distance vector(D) , where D[u] is the distance from source to vertex 'u'
- Predecessor vector(P) , where P[u] predecessor vertex of vertex 'u'

Edge Relaxation: Relaxing an edge (u,v) is testing whether we can improve the shortest path from vertex 'u' to vertex 'v' found so far and updating the distance vector and predecessor vector.



```
Relax(u,v,W)
```

// Relax the distance from vertex u to v, W is a weighted matrix.

```
1. if d[u] + w(u,v) < d[v]
2.      d[v] = d[u] + w(u,v)
3.      p[v] = u
```

It runs in O(1) time , but it depends on implementation

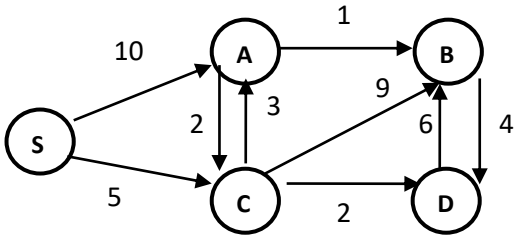
Algorithm: DIJKSTRA(G,W,s)

//Solves single source shortest path problem on a graph G with Weighted matrix

//W and a source vertex s

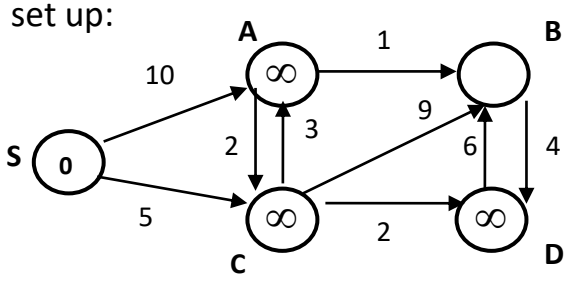
```
1. for each vertex v ∈V [G]
2.      d[v] = ∞
3.      p[v] = Nil
4. d[s] = 0
5. Q = V[G]          // Insert all vertices in  Priority Queue Q
6. while Q ≠Ø
7.      u= EXTRACT_MIN(Q)//Delete a vertex with smallest d value from Q
8.      for each vertex v ∈ Adj[u]
9.           RELAX(u,v,W)
10.return
```

Problem: Apply Dijkstra's algorithm in following graph , assuming source vertex 's'



Solution:

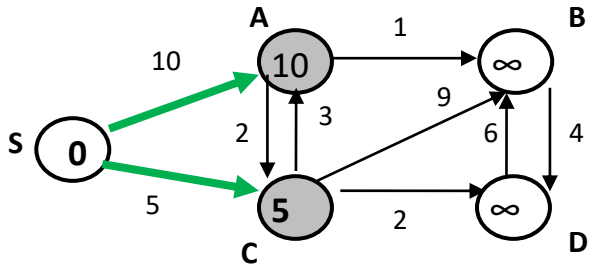
Initial set up:



V	S	A	B	C	D
d	0	∞	∞	∞	∞
p	Nil	Nil	Nil	Nil	Nil

QUEUE : **S,A,B,C,D**

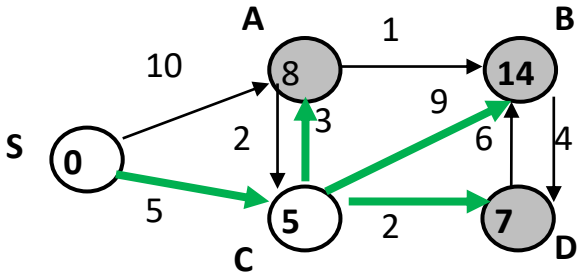
Step-1: Extract S, Relax edge (S,A) and (S,C)



V	S	A	B	C	D
d	0	10	∞	5	∞
p	Nil	S	Nil	S	Nil

QUEUE : **A,B,C,D**

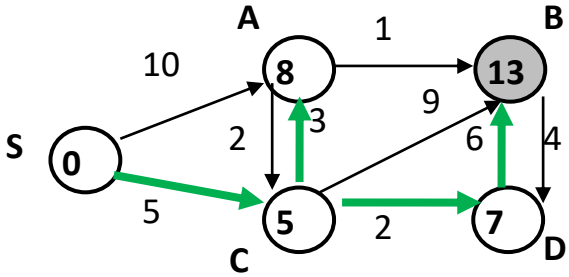
Step-2: Extract C, Relax edge (C,A) ,(C,B) and (C,D)



V	S	A	B	C	D
d	0	8	14	5	7
p	Nil	C	C	S	C

QUEUE : **A, B, D**

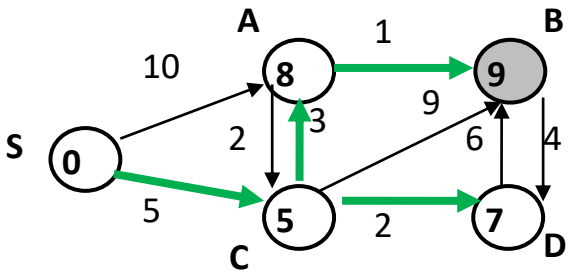
Step-3: Extract D, Relax edge (D,B)



Vertex	S	A	B	C	D
d	0	8	13	5	7
p	Nil	C	D	S	C

QUEUE : **A, B**

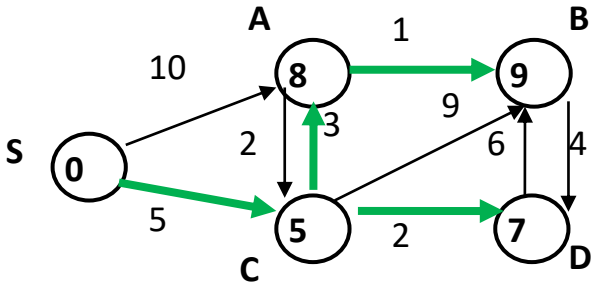
Step-4: Extract A, Relax edge (A,B), (A,C)



V	S	A	B	C	D
d	0	8	9	5	7
p	Nil	C	A	S	C

QUEUE : **B**

Step-5: Extract B, Relax edge (B,D)



V	S	A	B	C	D
d	0	8	9	5	7
p	Nil	C	A	S	C

QUEUE : Nil

Analysis:

The loop in line number 1 takes O(V) times.

Inside the while loop, Extract_Min(Q) is called |V|times. Each Extract_Min takes O(logV) times. So a total of O(VlogV) times is required for all extract_Min operations.

The relax() operation internally calls to the decrease_key() operation to decrease the 'd' value of the vertices. Each decrease_key operation on a Min-Priority Queue takes O(logV) times. As there are O(E) calls to relax(), it takes a total of O(ElogV) time altogether. Therefore, the running time of the algorithm is –

$$O(V) + O(VlogV) + O(ElogV) = O(ElogV)$$

Bellman-Ford

This algorithm solves single source shortest path problem in which there is one or more negative weight edge.

Give a weighted directed graph G=(V,E) , a source 's' and a weighted vector W, the algorithm returns a Boolean value indicating whether or not there is a -ve weight cycle reachable fro source. If there is a –ve weight cycle exists, the algorithm return 0 indicating 'No solution exist'. Else, the algorithm produces the shortest path.

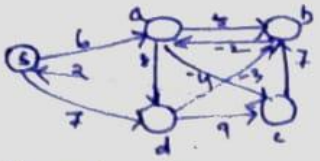
Algorithm: BELLMAN-FORD(G,W,s)

//Solves single source shortest path problem on a graph G with Weighted matrix //W and a source vertex s

```
1.  for each vertex v ∈ V [G]
2.          d[v] = ∞
3.          p[v] = Nil
4.  d[s] = 0
5.  for i =1 to |V[G]| -1
6.          for each edge(u,v)  ∈ E[G]
7.                  RELAX(u,v,W)
8.  for each edge(u,v)  ∈ E[G]
9.          if    d[v] > d[u] + W(u,v)
10.                 return False
11. return True
```

Analysis : Loop in line 1-3takes O(V) times. For loop in line-5 has a inner loop that runs O(E) times for all the vertices in V., which contributes O(VE) times. Loop in line-8 takes O(E) times. S the totak running time = O(V) + O(VE) +O(E) = O(VE).

Problem

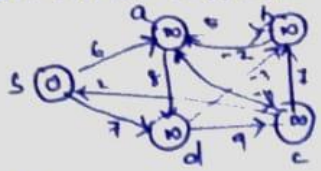


Let source = s

Let edges are taken in following order

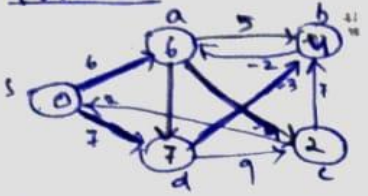
(sa)(sd), (ab)(ad), (ba)(cs), (cb), (db)(dc)

Shortest Path estimates –



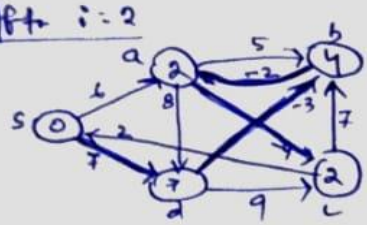
	s	a	b	c	d
d	0	∞	∞	∞	∞
p	NIL	NIL	NIL	NIL	NIL

After i = 1



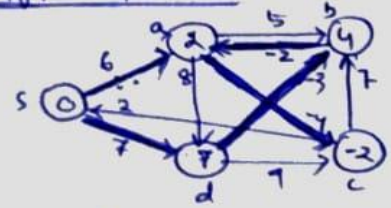
	s	a	b	c	d
d	0	6	4	2	7
p	-	s	d	a	s

After i = 2



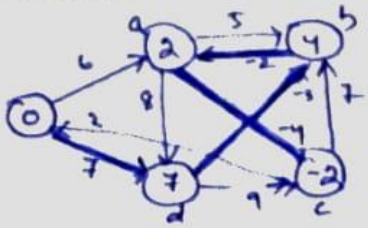
	s	a	b	c	d
d	0	2	4	2	7
p	-	b	d	a	s

After step i = 3



	s	a	b	c	d
d	0	2	4	2	7
p	-	b	d	a	s

After step (i = 4)



	s	a	b	c	d
d	0	2	4	2	7
p	-	b	d	a	s

Huffman Code

Huffman code is a technique used for data compression i,e the reduction in size of data or file.

Huffman's greedy algorithm used a table of occurrences of characters to build an optimal way of representing each character as binary strings. Use of Huffman code always creates ambiguity free data compression as the generated code is a prefix code.

Prefix code: the code in which no codeword is a prefix of some other codeword is called prefix code. It is highly required in order to avoid any ambiguity in differentiating the characters.

Two types of Huffman code

- **Fixed length code:** It uses a fixed number of bits to represent all the characters in the file. It is less efficient.
- **Variable length code**: It is an optimal approach, where frequently occurred characters are assigned shorter length codeword and infrequent characters are assigned longer codeword. These will reduce the size of the data file to a great extent.

Decoding schemes:

Huffman code is generated by constricting a binary tree. Leaves represent the characters with their frequencies. Non-leaf nodes represent the sum of the frequencies of its children.

The code words are obtained from leaves. The codeword for a character is interpreted by combining the symbols from the root to that character, where '0' indicates the left edge and '1' indicates the right edge.

Problem: *Following table represents the characters and their frequency of occurrences. Construct the Huffman codes. Also, find the compression ratio.*

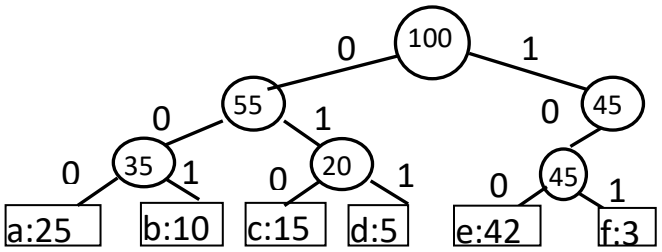
Character	a	b	c	d	e	f
Frequency (in thousands)	25	10	15	5	42	3

Solution:

Size of the original file = number of character × bit length of each character

= 1,00,000 × 8 = 8,00,000

Construction of Huffman code using Fixed length code



After collecting the symbols the codeword for each character are given below.

Character	Original code	Fixed length Huffman code
a	01100001	000
b	01100010	001
c	01100011	010
d	01100100	011
e	01100101	100
f	01100110	101

Length of each codeword is 3. So size of entire file = 1,00,000 × 3 = 3,00,000

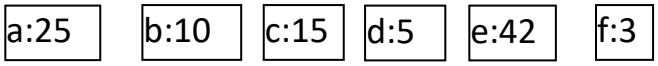
The compression ratio is 8,00,000/3,00,000 = 2.67

Construction of Huffman code using variable length code

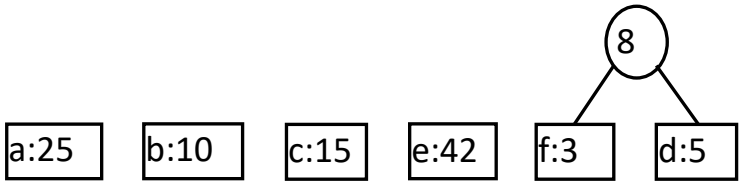
Put all the characters in a min-priority Queue. At each steps

- extract two least frequent nodes
- create a new node, associate two extracted nodes as left child and right child.
- Set the frequency of parent node as the sum of frequency of its children

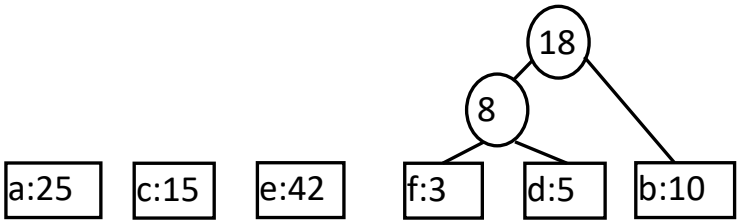
- Put the parent node in Queue.



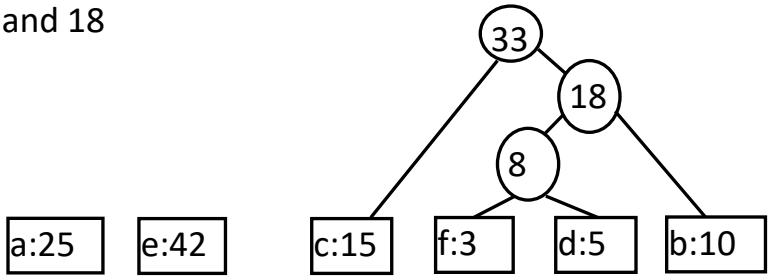
Step-1 : Extract f and d



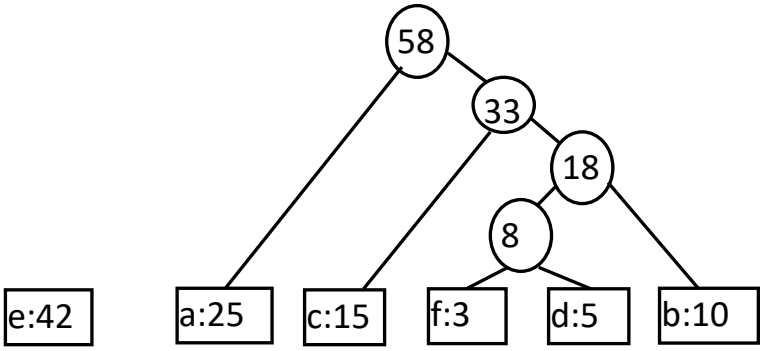
Step-2 : Extract 8 and b



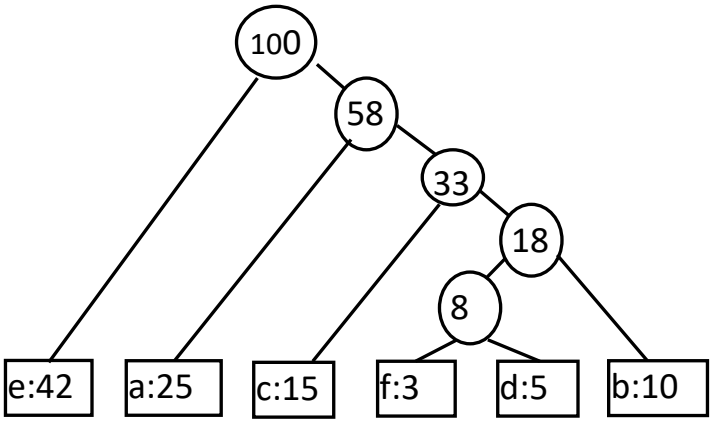
Step-3 : Extract c and 18



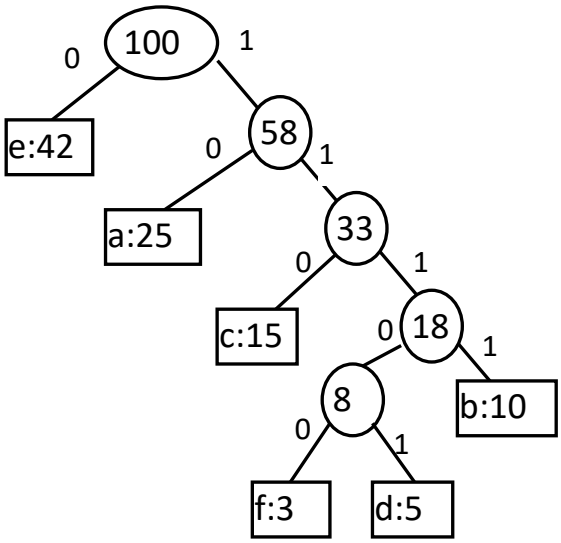
Step-4 : Extract a and 33



Step-5 : Extract e and 58



Re drawing the tree below



After collecting the symbols the codeword for each character are given below.

Character	Original code	Variable length Huffman code
a	01100001	10
b	01100010	1111
c	01100011	110
d	01100100	11101
e	01100101	0
f	01100110	11100

Length of each codeword is 3. So size of entire file =

1000×(25×2 + 10 × 4 + 15 ×3+ 5× 5+ 42 × 1+ 3 × 5)= 2,17,000

The compression ratio is 8,00,000/2,17,000 = 3.69

Huffman(C)

1. n = |C|
2. Q = C // Put all characters in C in Min-priority Queue Q
3. **for** i=1 to n - 1
4. create a node Z
5. left[Z] = x =Extract_Min(Q)
6. right[Z] = y =Extract_Min(Q)
7. f[z] = f[x] + f[y]
8. Insert(Q,z)
9. return Extract_Min(Q)

Analysis:

Time to insert n characters in Queue takes O(n) times.

Extract_Min() removes and return an element from the priority Queue.

Each Extract_Min() ins inside the loop takes O(logn) times. As the loop runs n-1 times. So time spent in the loop is O(nlogn) time.

Therefore running time of the algorithm is O(n) + O(nlogn) = O(nlogn)

Knapsack Problem

There are n objects $< x1, x2, ... xn >$ of known weights $< w1, w2, ... wn >$ with values $< v1, v2, ... vn >$ and a knapsack of capacity M. The objective of knapsack problem is to obtain a filled knapsack that maximizes the value.

As **a fractional knapsack problem**, the fraction of items can be taken, i,e 0≤xi≤1.

Problem: Solve fractional knapsack problem with following information.

n = 3, M = 20,

Item	X1	X2	X3
Weight	18	15	10
Value	25	24	15

Solution:

Arrange the items in most valuable to least valuable order. (i,e value/weight)

Item	X2	X3	X1	
Weight(W)	15	10	18	
Value(V)	24	15	25	
V/W	1.6	1.5	1.4	

Action	X1	X2	X3	∑ViXi (0)	∑WiXi (0)	Remaining M=20
Take x2	0	1	0	24	15	5
Take 5/10 of x3	0	1	5/10	24+7.5 =31.5	15+5 =20	0

Algorithm:

```
Knapsack_Greedy(n,M,W,V)
1. Arrange the items in ascending order by Vi/Wi.
2. For i = 1 to n
3.     X[i] = 0
4. total = 0
5. for i= 1 to n
6.     if W[i] <=M
7.         X[i] = 1
8.         M = M – W[i]
9.         total = total + V[i]
10.    else,
11.        X[i] = M/W[i]
12.        M = 0
13.        total = total + V[i]×M/W[i]
14. return X , total
```