



DESIGN AND ANALYSIS OF ALGORITHMS

Unit - II

GREEDY ALGORITHM

- General Method
- Fractional Knapsack Problem
- Job sequencing with deadlines
- Optimal storage on tapes
- Optimal merge patterns
- Huffman Codes
- Dijkstra's algorithm.

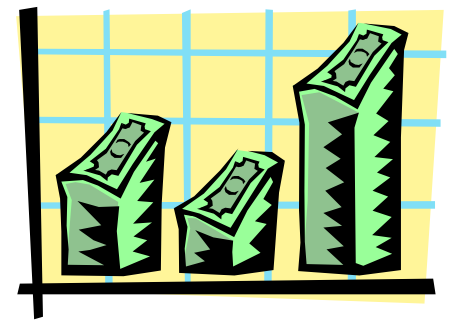
GREEDY METHOD



INTRODUCTION TO GREEDY TECHNIQUE



TECHNIQUE



- **The greedy method** is a general algorithm design paradigm, built on the following elements:
 - **configurations**: different choices, collections, or values to find
 - **objective function**: a score assigned to configurations, which we want to either maximize or minimize.
- It works best when applied to problems with the **greedy-choice** property:
 - a globally-optimal solution can always be found by a series of local improvements from a starting configuration.

GREEDY METHOD INTRODUCTION

- The greedy approach does *not always* lead to an optimal solution.
- The problems that have a greedy solution are said to possess the greedy-choice property.
- The *greedy approach* is also used in the context of hard (*difficult to solve*) problems in order to generate an approximate solution.

GREEDY METHOD INTRODUCTION

the problems that can be solved by the greedy method have n inputs and require us to obtain the subset that satisfies some constraints, this subset would be the solution set.

Any subset that satisfies these constraints is called a feasible solution.

A feasible solution that either maximizes or minimizes a give objective function is called an optimal solution.

The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time.

At each stage decision is made regarding whether a particular input is in an optimal solution.

This is done by considering the inputs in an order determined by some selection procedure.

GREEDY METHOD INTRODUCTION

The selection procedure is based on some optimization measure (which may be the objective function). There may be several different optimization measures possible for a given problem, most of these will result in algorithms that generate suboptimal solutions. This version of greedy technique is called the subset paradigm.

GREEDY METHOD CONTROL ABSTRACTION FOR SUBSET PARADIGM

Algorithm *greedy*(*a*, *n*)

//*a*[1:*n*] contains the *n* inputs

```
{  
    solution:=0.0;  
    for i:=1 to n do  
    {  
        x=Select(a);  
        if Feasible(solution, x) then  
            solution:=Union(solution, x);  
    }  
    return solution;  
}
```

GREEDY METHOD INTRODUCTION

- For problems that do not call for the selection of optimal subset, in the greedy method we make decisions by considering the inputs in some order. Each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is called ordering paradigm.









FRACTIONAL KNAPSACK PROBLEM

- Given: A set S of n items, with each item i having
 - p_i - a positive benefit
 - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most M .
- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
 - In this case, we let x_i denote the amount we take of item i
 - Objective: maximize
$$\sum_{i \in S} p_i x_i$$
 - Constraint:
$$\sum_{i \in S} w_i x_i \leq M$$
 - and $0 \leq x_i \leq 1$, $1 \leq i \leq n$, the profits and weights are positive.



FRACTIONAL KNAPSACK PROBLEM

- Given: A set S of n items, with each item i having
 - p_i - a positive benefit
 - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W .

Items:							Solution:
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml	10 ml	<ul style="list-style-type: none">• 1 ml of 5• 2 ml of 3• 6 ml of 4• 1 ml of 2
Profit:	\$12	\$32	\$40	\$30	\$50		
Profit/Weight: (\$ per ml)	3	4	20	5	50		

FRACTIONAL KNAPSACK ALGORITHM



Algorithm *greedyKnapsack*(m, n)

// $p[1:n]$ and $w[1:n]$ contains profits & weights respectively of the n objects ordered such that $p[i]/w[i] \geq p[i+1]/w[i+1]$

// m is the knapsack size and $x[1:n]$ is the solution vector

```
{
    for  $i:=1$  to  $n$  do  $x[i]:=0.0$ ;
     $u:=m$ ;
    for  $i:=1$  to  $n$  do
    {
        if( $w[i]>u$ ) then break;
         $x[i]:=1.0$ ;
         $u:=u-w[i]$ ;
    }
    if( $i \leq n$ ) then  $x[i]=u/w[i]$ ;
}
```

ANALYSIS

If the items are already sorted into decreasing order of p_i/w_i ,
then the for-loop takes a time in $O(n)$;

Therefore, the total time including the sort is in **$O(n \log n)$** .

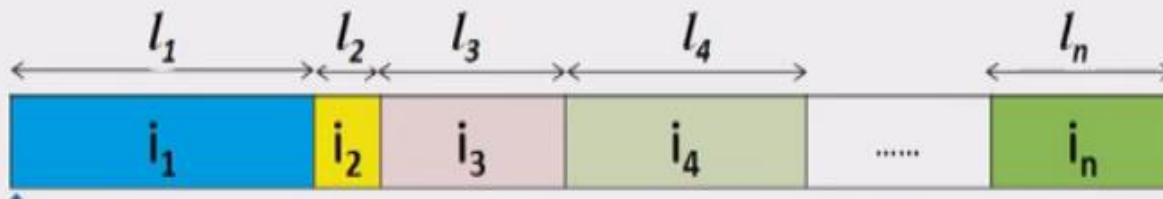
APPLICATIONS:

- Resource allocation with financial constraints
- Construction and scoring of heterogeneous test
- Selection of capital investments.

OPTIMAL STORAGE ON TAPES

- There are n programs that are to be stored on a computer tape of length l .
- Program i has length l_i , $1 \leq i \leq n$
- Let Programs are stored in the order $I = i_1, i_2, \dots, i_n$, Time t_j needed to retrieve program i_j is proportional to

$$\sum_{k=1}^j l_{ik}$$



OPTIMAL STORAGE ON TAPES

If all the programs retrieved often the **Expected or Mean Retrieval Time (MRT)** is

$$MRT = \frac{1}{n} \sum_{j=1}^n t_j$$

Objective is to find permutation for n programs so that Mean Retrieval Time (MRT) is minimized.

Minimizing MRT is equivalent to minimizing

$$d(I) = \sum_{j=1}^n \sum_{k=1}^j l_{ik}$$

EXAMPLE

Let $n=3$ $(l_1, l_2, l_3) = (5, 10, 3)$

For these 3 programs, how many orderings are possible?

Possible solutions ($n!$)

Ordering No.	Program order		
1	1	2	3
2	1	3	2
3	2	1	3
4	2	3	1
5	3	1	2
6	3	2	1

Possible solutions

No.	Program order			d(I)
1	1	2	3	$5 + (5+10) + (5+10+3) = 38$
2	1	3	2	$5 + (5+3) + (5+3+10) = 31$
3	2	1	3	$10 + (10+5) + (10+5+3) = 43$
4	2	3	1	$10 + (10+3) + (10+3+5) = 41$
5	3	1	2	$3 + (3+5) + (3+5+10) = 29$
6	3	2	1	$3 + (3+10) + (3+10+5) = 34$

Possible Solutions

No.	Program order			d(I)	MRT
1	1	2	3	$5 + (5+10) + (5+10+3) = 38$	$38/3 = 12.66$
2	1	3	2	$5 + (5+3) + (5+3+10) = 31$	$31/3 = 10.33$
3	2	1	3	$10 + (10+5) + (10+5+3) = 43$	$43/3 = 14.33$
4	2	3	1	$10 + (10+3) + (10+3+5) = 41$	$41/3 = 13.33$
5	3	1	2	$3 + (3+5) + (3+5+10) = 29$	$29/3 = 9.66$
6	3	2	1	$3 + (3+10) + (3+10+5) = 34$	$34/3 = 11.33$

Find all the permutations of programs and find optimal Solution (minimum MRT).

Drawback : for large value of n , time will be high to find all possible permutations.

Store programs in **increasing order of their lengths** using any sorting order. (Time required for sorting: $n \log n$)

i.e. in given example, increasing order of lengths are : 3,5,10 and hence ordering of programs is 3,1,2.

Theorem: If $l_1 \leq l_2 \leq \dots \leq l_n$ then the ordering $i_j = j$, $1 \leq j \leq n$, minimizes

$$d(I) = \sum_{j=1}^n \sum_{k=1}^j l_{ik}$$

over all possible permutations of i_j .

Proof: Let $I = i_1, i_2, \dots, i_n$ be any permutation of index set $\{1, 2, \dots, n\}$ then

$$d(I) = \sum_{k=1}^n \sum_{j=1}^k l_{ij} = \sum_{k=1}^n (n - k + 1) l_{ik} \quad (\text{eq. 1})$$

If there exists a and b such that $a < b$ and $l_{ia} > l_{ib}$, then interchanging of i_a and i_b results in a permutation I' with

$$d(I') = \left[\sum_{\substack{k=1 \\ k \neq a \\ k \neq b}}^n (n - k + 1) l_{ik} \right] + (n - a + 1) l_{ib} + (n - b + 1) l_{ia} \quad (\text{eq. 2})$$

Subtracting $d(I')$ from $d(I)$, we obtain

$$\begin{aligned}d(I') - d(I) &= (n-a+1) (l_{ia} - l_{ib}) + (n-b+1) (l_{ib} - l_{ia}) \\&= (b-a) (l_{ia} - l_{ib}) \quad (\text{eq.3}) \\&> 0\end{aligned}$$

Hence no permutation that is not in increasing order of l_i can have minimum d .

let $n=4$ and lengths : 12 ,5, 8,4

Find permutation I (increasing order of length) and
 I' (any other permutation)

$$I = (1,2,3,4) (4,5,8,12) \quad I' = (1,2,3,4) (4,8,5,12)$$

$$d(I) = ((4-1+1) * 4) + ((4-2+1) * 5) + ((4-3+1) * 8) + ((4-4+1) * 12) \quad (\text{eq. 1})$$

$$= 59$$

$$a=2 \text{ and } b=3$$

$$d(I') = ((4-1+1) * 4) + ((4-4+1) * 12) + ((4-2+1) * 8) + ((4-3+1) * 5) \quad (\text{eq. 2})$$

$$= 62$$

$$d(I') - d(I) = 62-59 = 3 > 0$$

Using eq. 3, we get

$$d(I') - d(I) = (3-2) (8-5) = 3 > 0$$

TIME COMPLEXITY

- Time complexity is similar to sorting complexity
 $O(n \log n)$

OPTIMAL MERGE PATTERNS

- Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.
- If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as **2-way merge patterns**.
- As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.
- To merge a **p-record file** and a **q-record file** requires possibly **$p + q$** record moves, the obvious choice being, merge the two smallest files together at each step.
- Two-way merge patterns can be represented by binary merge trees. Let us consider a set of **n** sorted files **$\{f_1, f_2, f_3, \dots, f_n\}$** . Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

ALGORITHM

Algorithm: TREE (n)

```
for i := 1 to n - 1 do
    declare new node
    node.leftchild := least (list)
    node.rightchild := least (list)
    node.weight := ((node.leftchild).weight) +
        ((node.rightchild).weight)
    insert (list, node);
return least (list);
```

At the end of this algorithm, the weight of the root node represents the optimal cost.

EXAMPLE:

- Let us consider the given files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.

If merge operations are performed according to the provided sequence, then

$$M_1 = \text{merge } f_1 \text{ and } f_2 \Rightarrow 20 + 30 = 50$$

$$M_2 = \text{merge } M_1 \text{ and } f_3 \Rightarrow 50 + 10 = 60$$

$$M_3 = \text{merge } M_2 \text{ and } f_4 \Rightarrow 60 + 5 = 65$$

$$M_4 = \text{merge } M_3 \text{ and } f_5 \Rightarrow 65 + 30 = 95$$

Hence, the total number of operations is

$$50 + 60 + 65 + 95 = 270$$

- Now, the question arises is there any better solution?

- Sorting the numbers according to their size in an ascending order, we get the following sequence – f_4, f_3, f_1, f_2, f_5

Hence, merge operations can be performed on this sequence

$$M_1 = \text{merge } f_4 \text{ and } f_3 \Rightarrow 5 + 10 = 15$$

$$M_2 = \text{merge } M_1 \text{ and } f_1 \Rightarrow 15 + 20 = 35$$

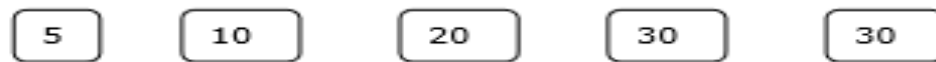
$$M_3 = \text{merge } M_2 \text{ and } f_2 \Rightarrow 35 + 30 = 65$$

$$M_4 = \text{merge } M_3 \text{ and } f_5 \Rightarrow 65 + 30 = 95$$

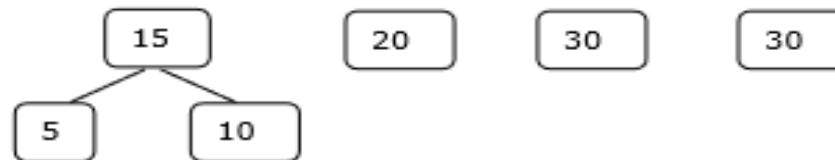
Therefore, the total number of operations is
 $15 + 35 + 65 + 95 = 210$

- Obviously, this is better than the previous one.

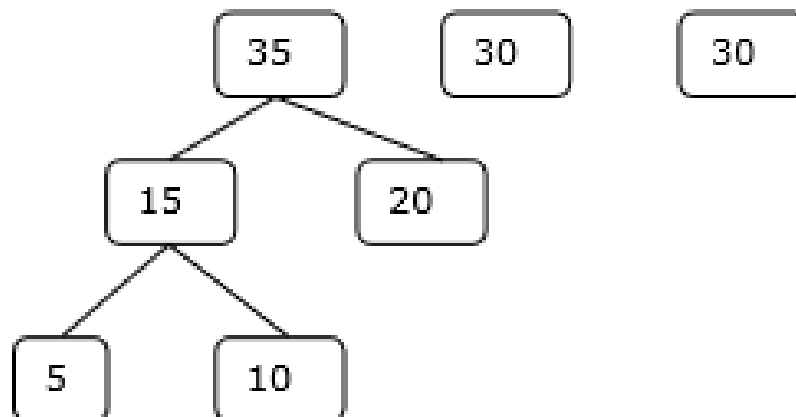
- Initial Set



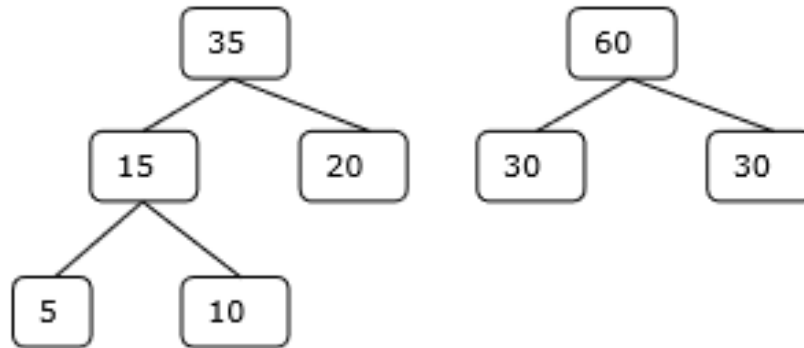
- Step-1



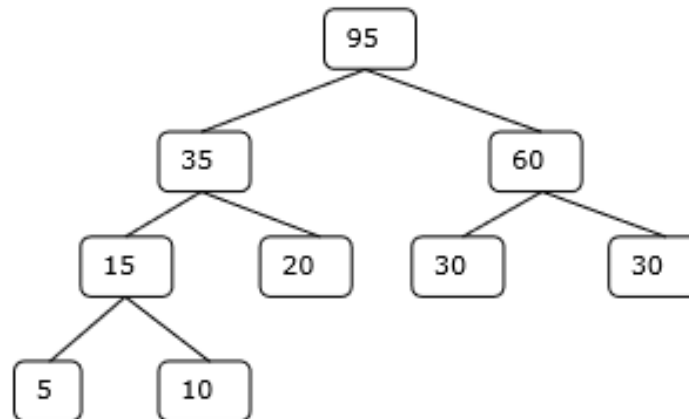
- Step-2



- Step-3



- Step-4



- Hence, the solution takes $15 + 35 + 60 + 95 = 205$ number of comparisons.

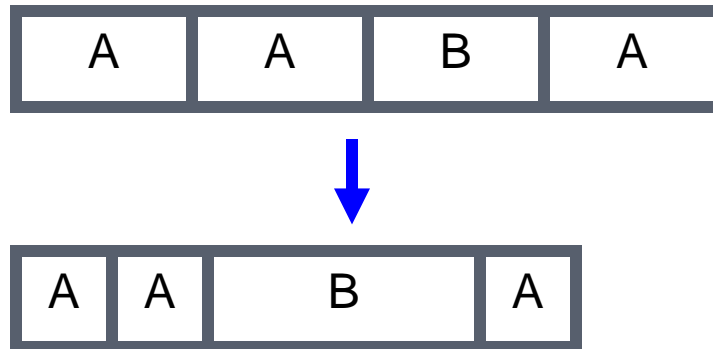
HUFFMAN CODING

○ Approach

- Variable length encoding of symbols
- Exploit statistical frequency of symbols
- Efficient when symbol probabilities vary widely

○ Principle

- Use fewer bits to represent frequent symbols
- Use more bits to represent infrequent symbols



HUFFMAN CODING

- Huffman Coding also known as Huffman Encoding is an algorithm for doing compression and it forms the basic idea behind file compression.

PROPERTIES

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

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

PROPERTIES

- Greedy algorithm
 - Chooses best local solution at each step
 - Combines 2 trees with lowest frequency
- Still yields overall best solution
 - Optimal prefix code
 - Based on statistical frequency
- Better compression possible (depends on data)
 - Using other approaches (e.g., pattern dictionary)

THERE ARE MAINLY TWO MAJOR PARTS IN HUFFMAN CODING

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

STEPS TO BUILD HUFFMAN TREE

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

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

STEPS TO BUILD HUFFMAN TREE

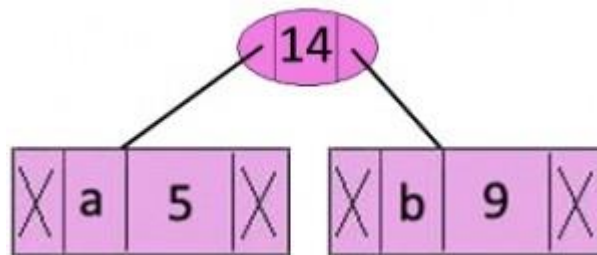
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

EXAMPLE:

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

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

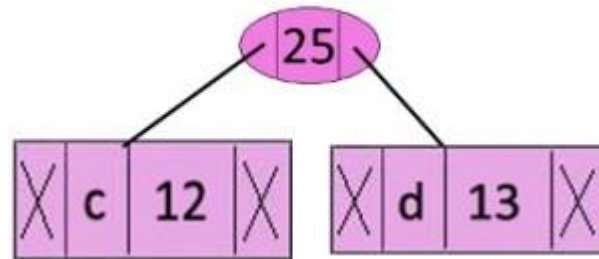
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency $5 + 9 = 14$.



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

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

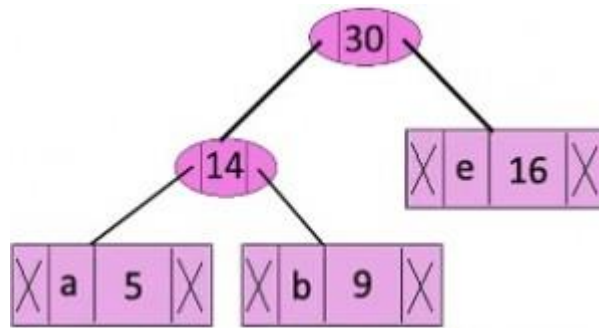
- **Step 3:** Extract two minimum frequency nodes from heap. Add a new internal node with frequency $12 + 13 = 25$



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

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

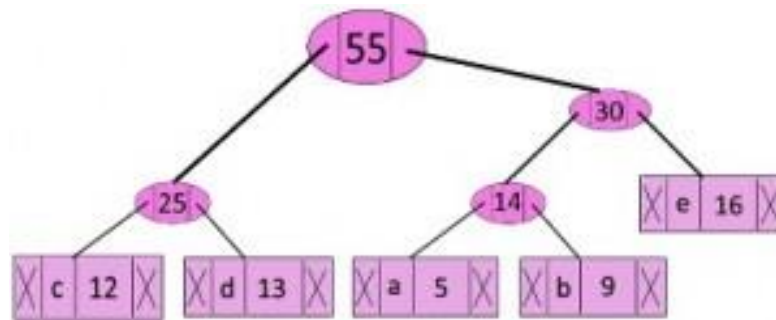
- **Step 4:** Extract two minimum frequency nodes.
Add a new internal node with frequency $14 + 16 = 30$



Now min heap contains 3 nodes.

character	Internal node	Internal node	f
Frequency	25	30	45

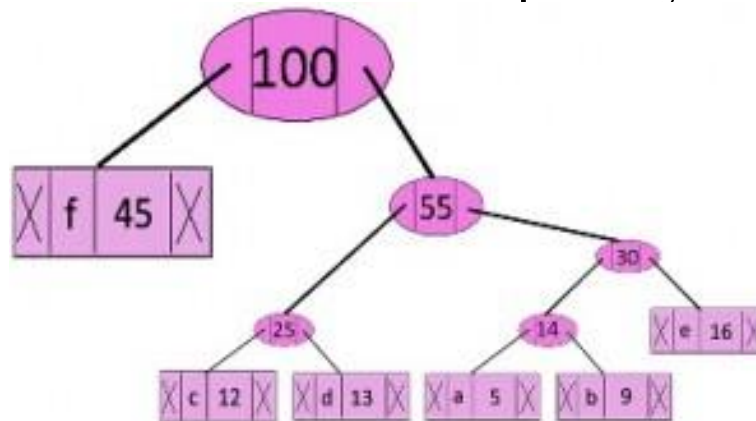
- **Step 5:** Extract two minimum frequency nodes.
Add a new internal node with frequency $25 + 30 = 55$



Now min heap contains 2 nodes.

character	f	Internal node
Frequency	45	55

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



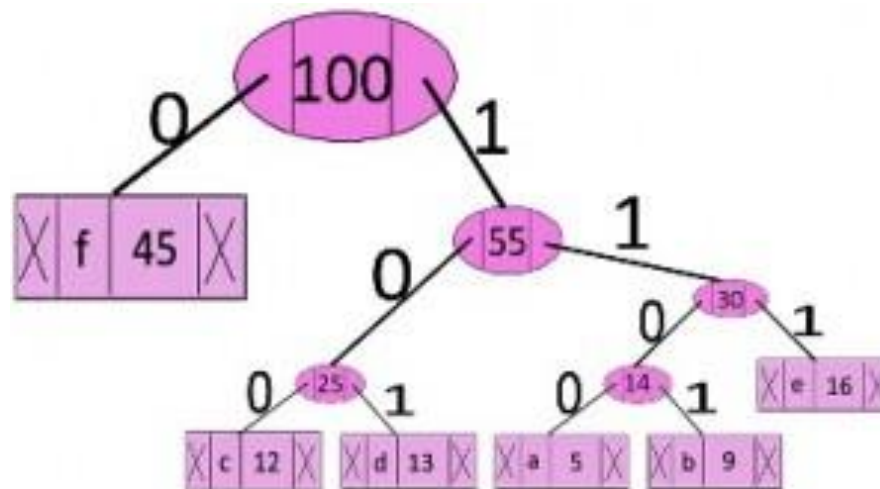
Now min heap contains only one node.

character	Internal node
Frequency	100

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

STEPS TO PRINT CODES FROM HUFFMAN TREE:

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



THE CODES ARE AS FOLLOWS:

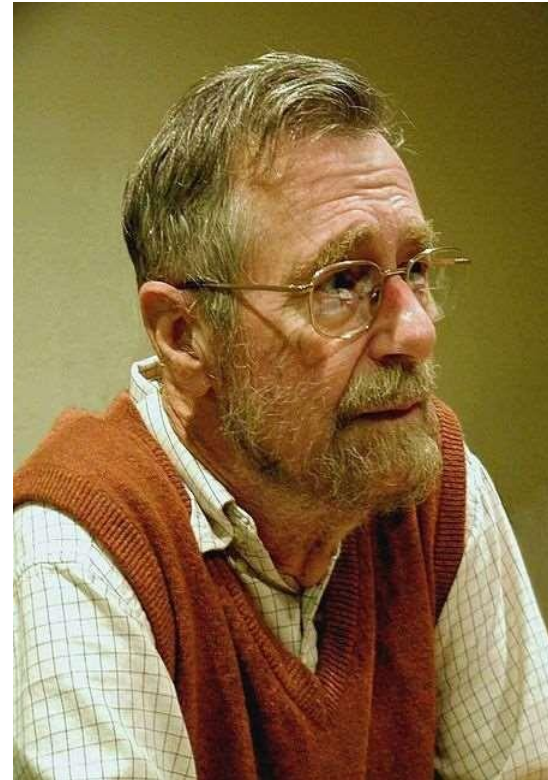
character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

POINTS TO REMEMBER

- It is a lossless data compressing technique generating variable length codes for different symbols.
- It is based on greedy approach which considers frequency/probability of alphabets for generating codes.
- It has complexity of $n \log n$ where n is the number of unique characters.
- The length of the code for a character is inversely proportional to frequency of its occurrence.
- No code is prefix of another code due to which a sequence of code can be unambiguously decoded to characters.

DIJKSTRA'S ALGORITHM

Dijkstra's Algorithm derived
by a Dutch computer scientist
'Edsger Wybe Dijkstra' in 1956
and published in 1959



DIJKSTRA'S ALGORITHM

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph $G=\{E,V\}$ and source vertex, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex to all other vertices

PROCEDURE

This algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.

DIJKSTRA'S ALGORITHM

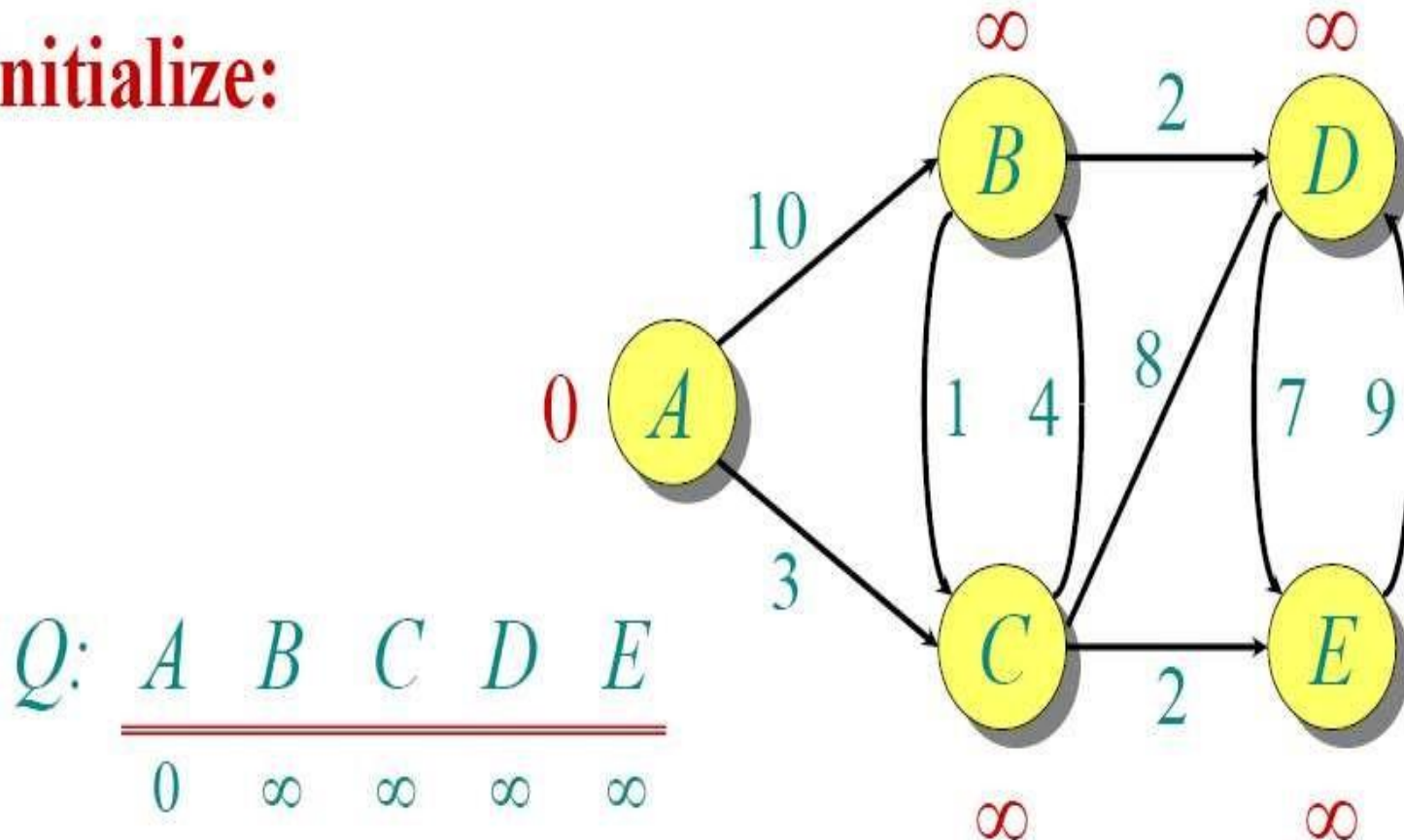
1. Create cost matrix $C[][]$ from adjacency matrix $adj[][]$. $C[i][j]$ is the cost of going from vertex i to vertex j . If there is no edge between vertices i and j then $C[i][j]$ is infinity. Consider infinity with largest number.
2. Array $visited[]$ is initialized to zero.
 for($i=0; i < n; i++$)
 $visited[i]=0$;
3. If the vertex 0 is the source vertex then $visited[0]$ is marked as 1.
4. Create the distance matrix, by storing the cost of vertices from vertex no. 0 to $n-1$ from the source vertex 0.
 for($i=1; i < n; i++$)
 $distance[i]=cost[0][i]$;

Initially, distance of source vertex is taken as 0. i.e. $distance[0]=0$;

5. for($i=1; i < n; i++$)
 - Choose a vertex w , such that $distance[w]$ is minimum and $visited[w]$ is 0. Mark $visited[w]$ as 1.
 - Recalculate the shortest distance of remaining vertices from the source.
 - Only, the vertices not marked as 1 in array $visited[]$ should be considered for recalculation of distance. i.e. for each vertex v
 if($visited[v]==0$)
 $distance[v]=\min(distance[v], distance[w]+cost[w][v])$

EXAMPLE

Initialize:



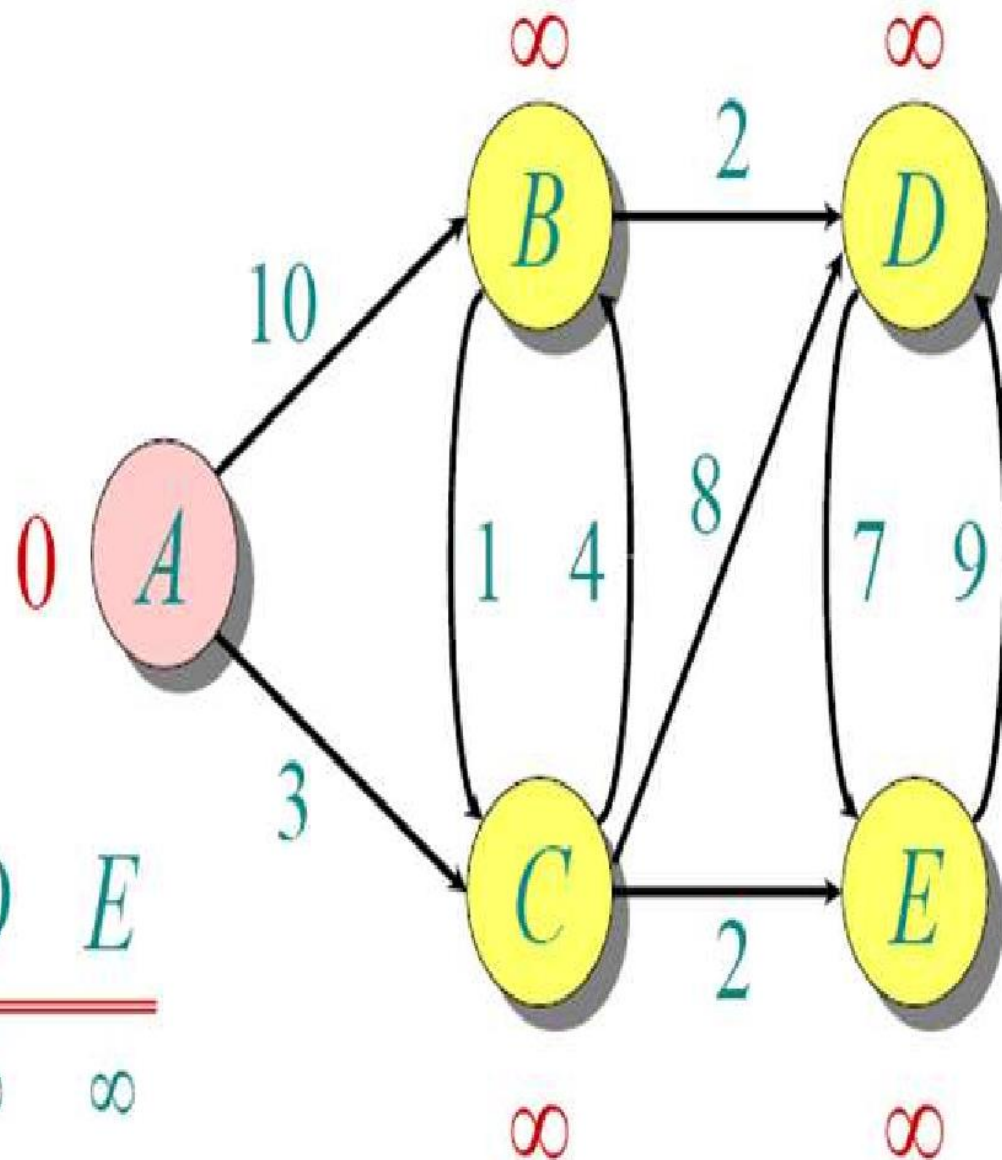
$Q:$

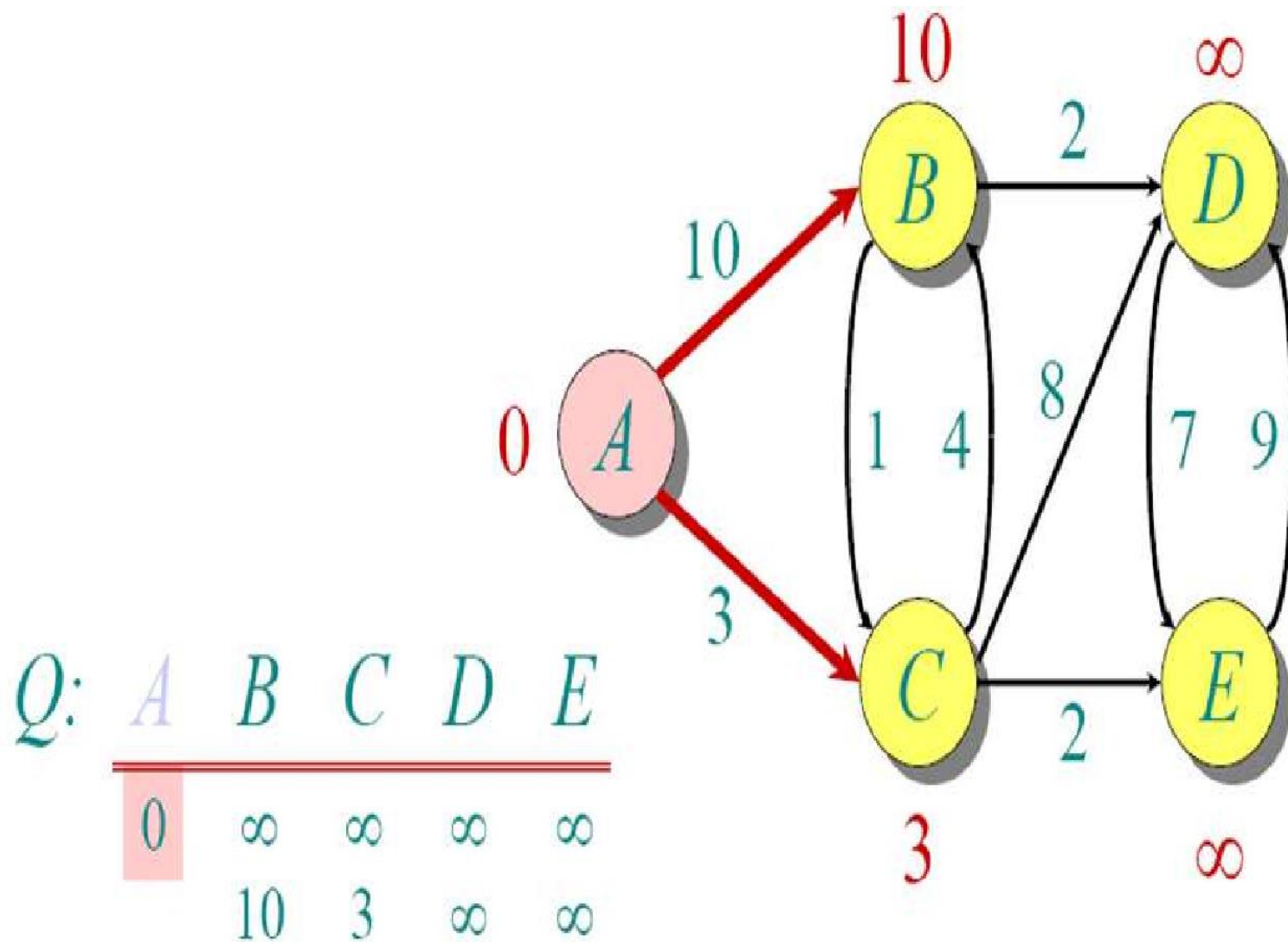
A	B	C	D	E
0	∞	∞	∞	∞

$S: \{\}$

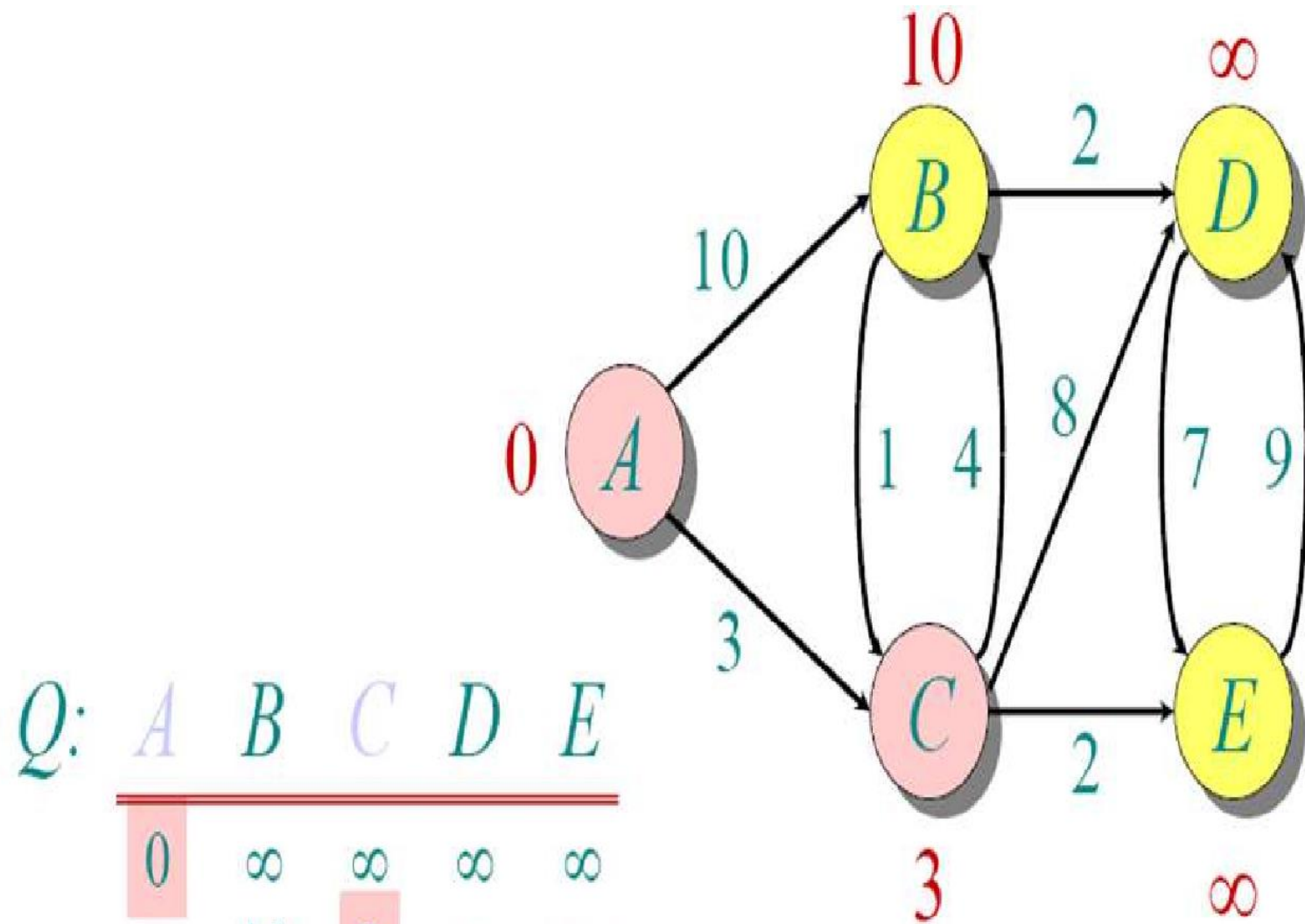
$Q:$

A	B	C	D	E
0	∞	∞	∞	∞

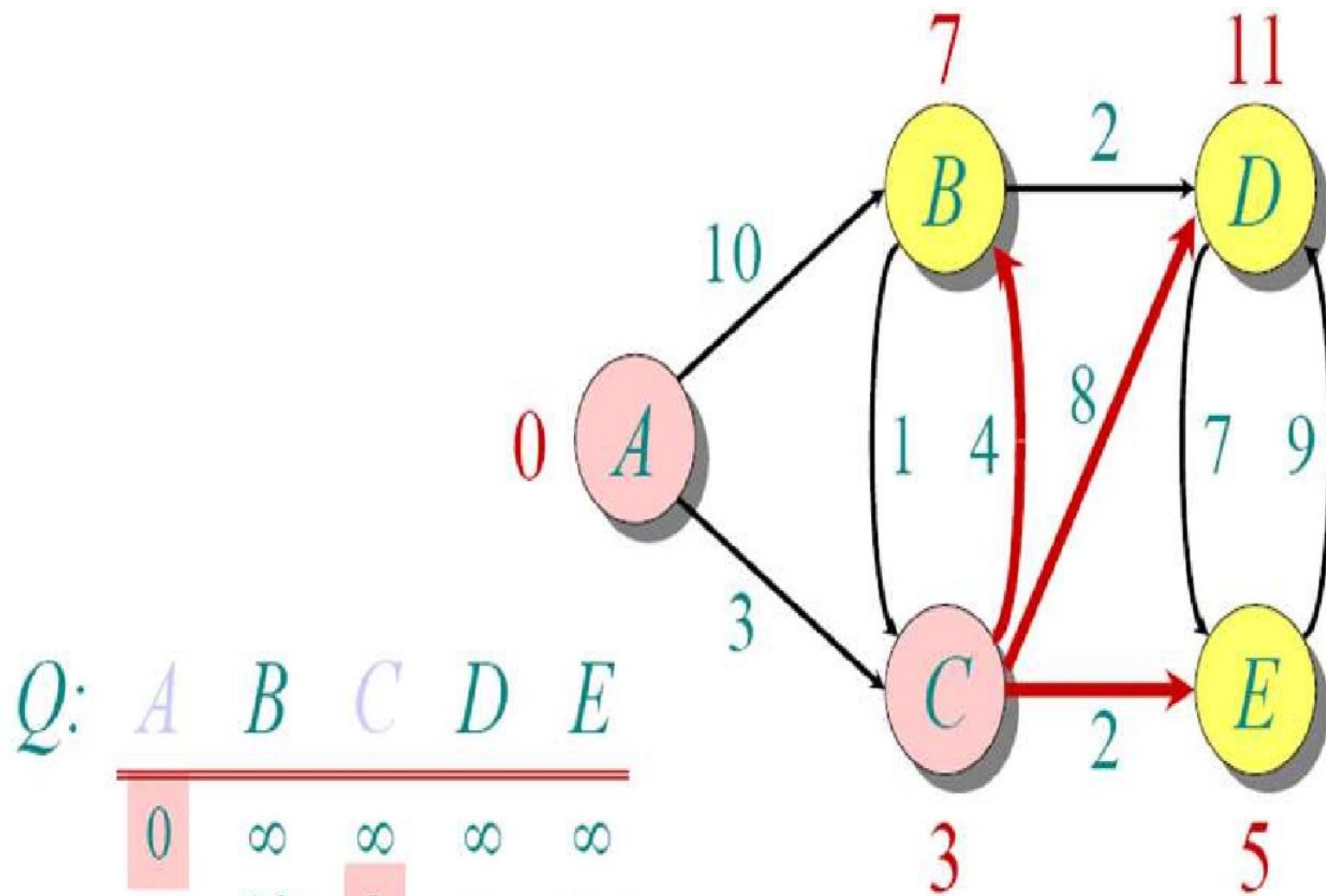




$S: \{A\}$



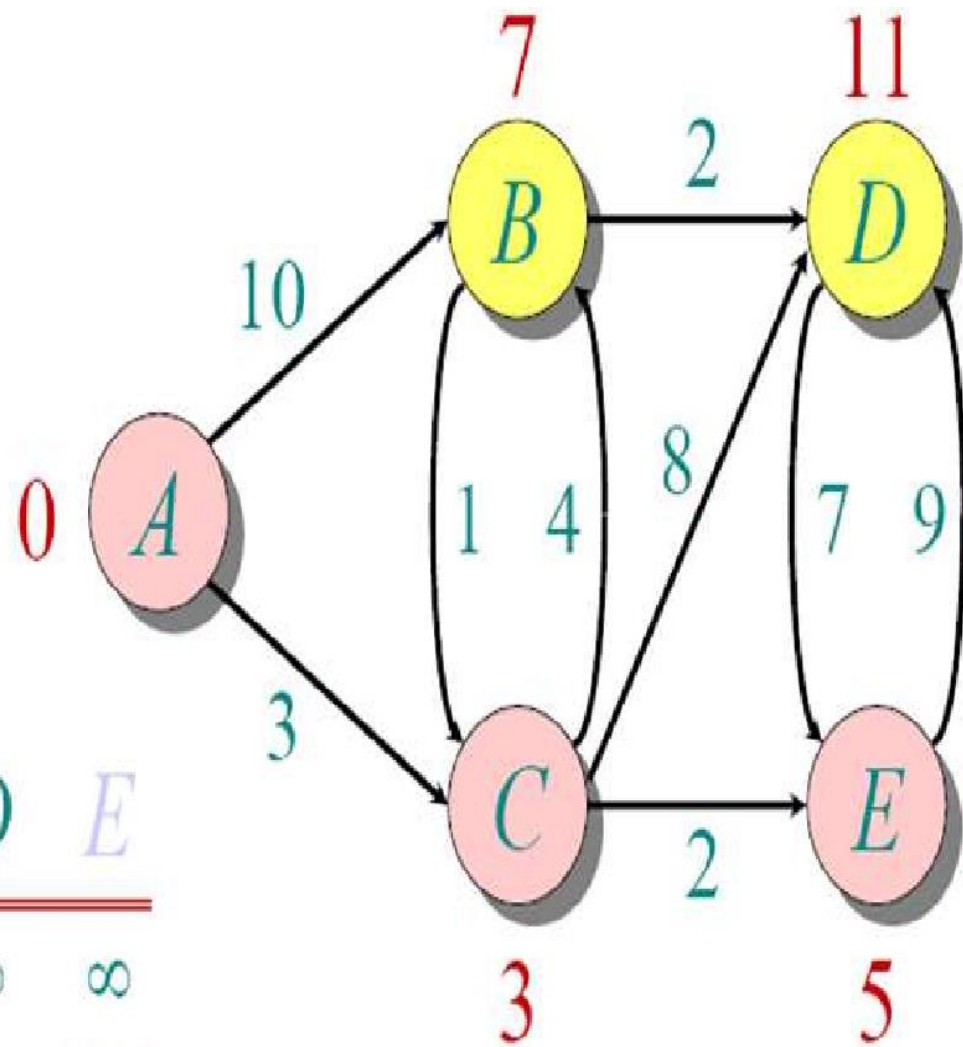
$S: \{A, C\}$



Q:

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5

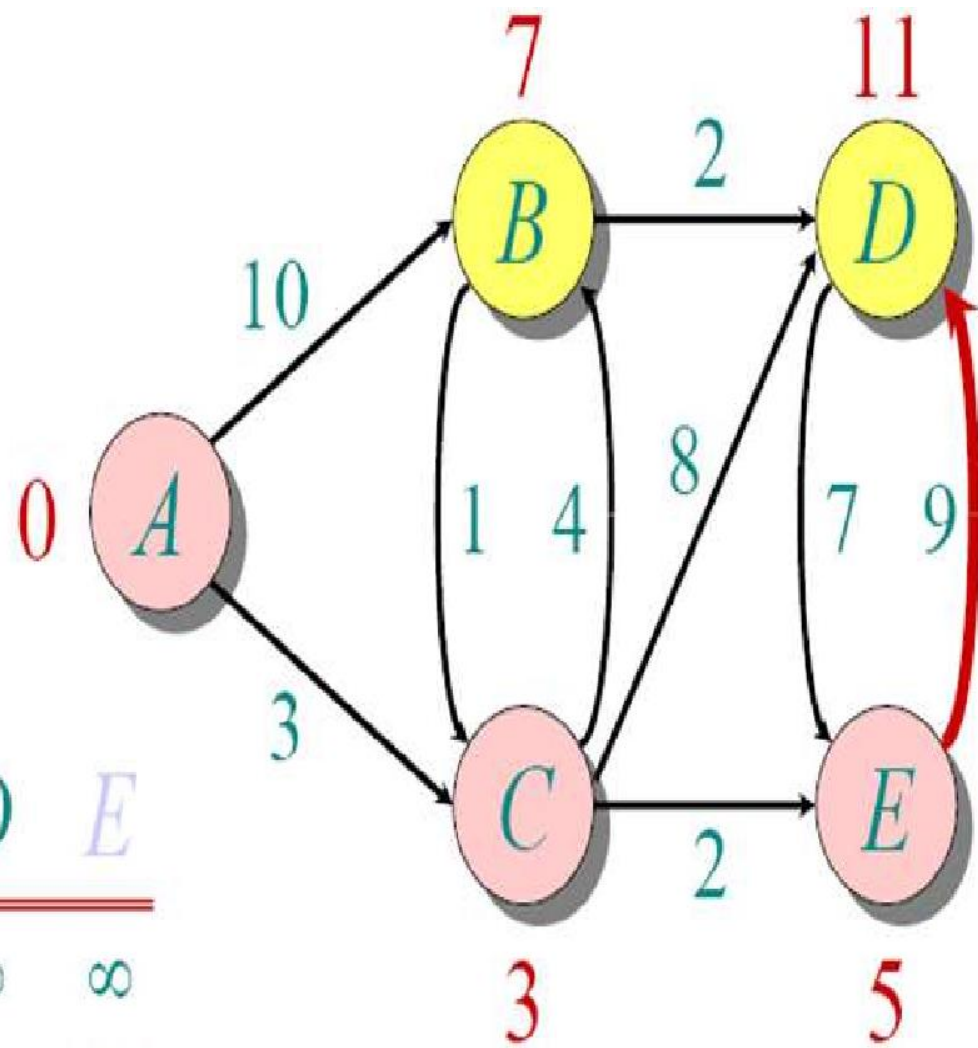
S: {A, C}



$Q:$

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5

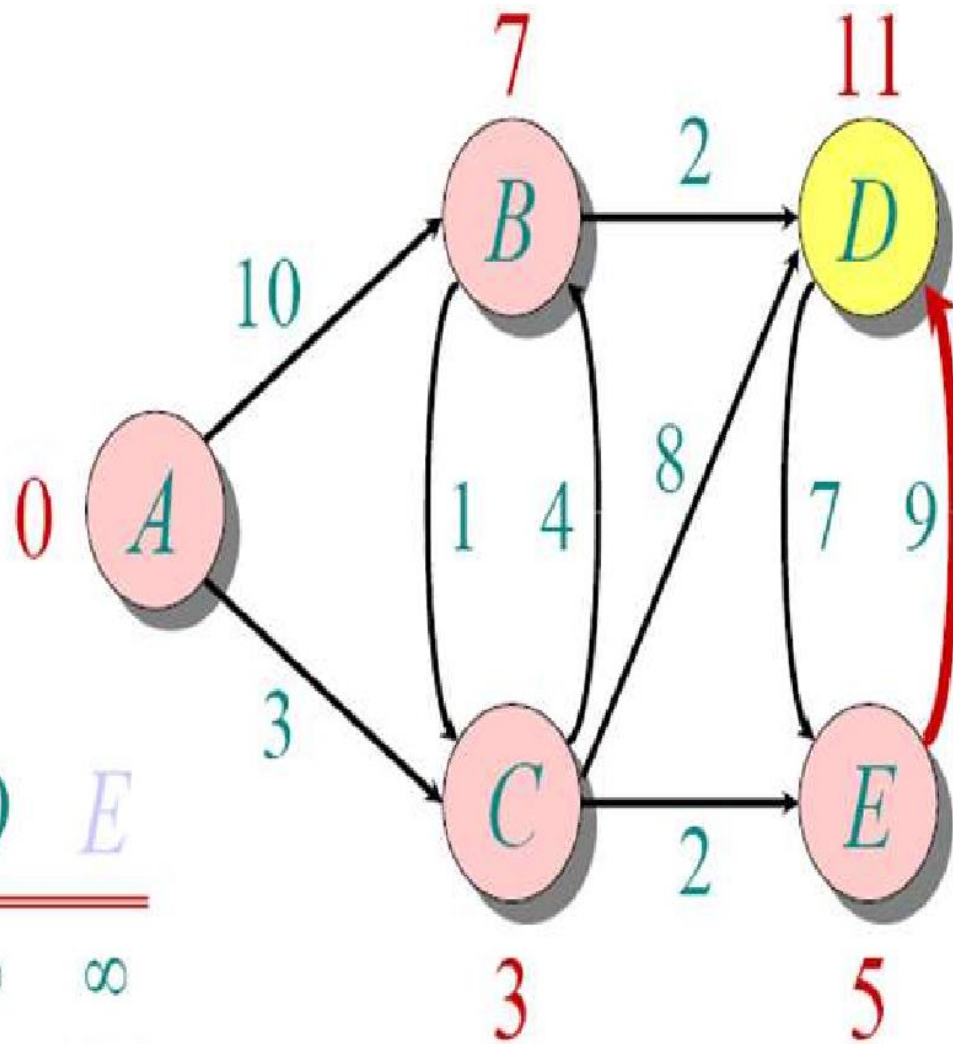
$S: \{A, C, E\}$



$Q:$

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	

$S: \{A, C, E\}$



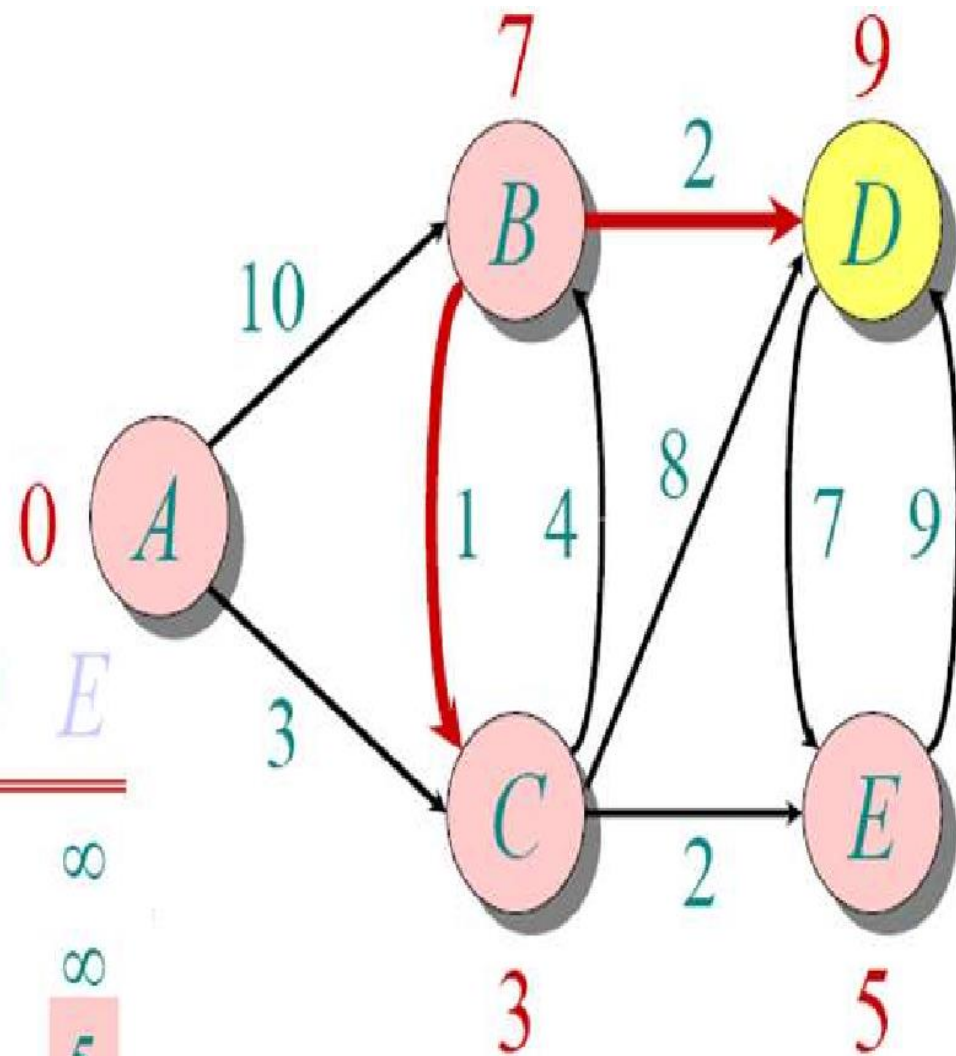
$Q:$

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	

$S: \{A, C, E, B\}$

$Q:$

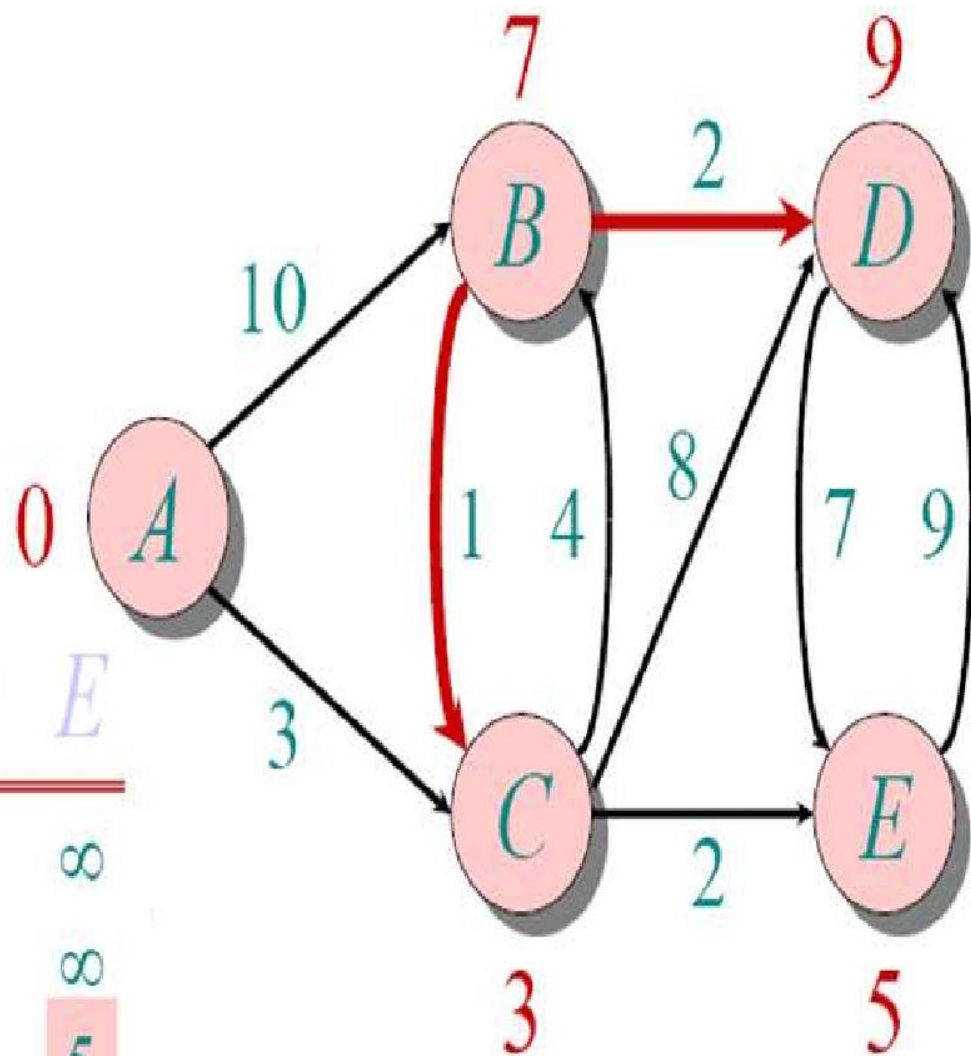
A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	



$S: \{A, C, E, B\}$

$Q:$

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	



$S: \{ A, C, E, B, D \}$

TIME COMPLEXITY: USING LIST

The simplest implementation of the Dijkstra's algorithm stores vertices in an ordinary linked list or array

- Good for dense graphs (many edges)
- $|V|$ vertices and $|E|$ edges
- Initialization $O(|V|)$
- While loop $O(|V|)$
 - Find and remove min distance vertices $O(|V|)$
 - Potentially $|E|$ updates
 - Update costs $O(1)$
- Reconstruct path $O(|E|)$

Total time $O(|V|^2 + |E|) = O(|V|^2)$

TIME COMPLEXITY: PRIORITY QUEUE

For sparse graphs, (i.e. graphs with much less than $|V|^2$ edges) Dijkstra's implemented more efficiently by *priority queue*

- Initialization $O(|V|)$ using $O(|V|)$ buildHeap
- While loop $O(|V|)$
 - Find and remove min distance vertices $O(\log |V|)$ using $O(\log |V|)$ deleteMin
 - Potentially $|E|$ updates
 - Update costs $O(\log |V|)$ using decreaseKey
- Reconstruct path $O(|E|)$

Total time $O(|V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$

- $|V| = O(|E|)$ assuming a connected graph

APPLICATIONS:

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems

