



## COURSE CODE & NAME DCA1202 – DATA STRUCTURES AND ALGORITHM



1.)

a.) **Question :-** What is a linked list? Discuss the algorithms for insertion and deletion of values in the end of a linked list?

**Answer :-** A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **Linked-List** – A Linked List contains the connection link to the first link called First.

### Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

### Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

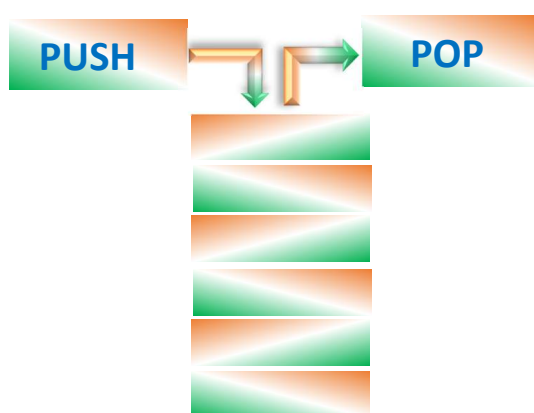
### Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

**b.) Question :- Define stacks and what are the applications of Stack?**

**Answer :-** Stack is an abstract data type and a data structure that follows LIFO (last in first out) strategy. It means the element added last will be removed first. Stack allows two operations push and pop. Push adds an element at the top of the stack and pop removes an element from top of the stack.



**Applications of Stack :-**

- **Expression Evaluation :-**

Stack is used to evaluate prefix, postfix and infix expressions.

**Expression Conversion :-** An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

**Syntax Parsing :-** Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

**Backtracking :-** Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

**Parenthesis Checking :-** Stack is used to check the proper opening and closing of parenthesis.

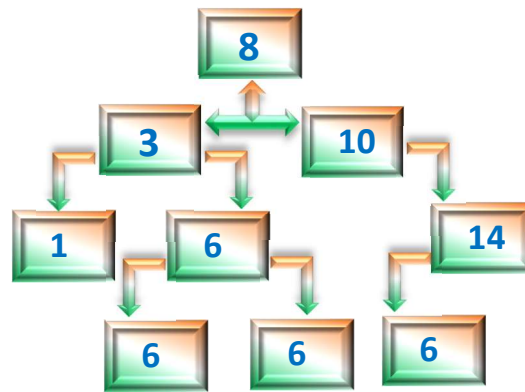
**String Reversal :-** Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

**Function Call** :- Stack is used to keep information about the active functions or subroutines. There are various other applications of stack. I have mentioned some of them. Comment below if you found any mistake in above article.

2.)

**A.) Question :- What are Binary trees? How many types of Binary trees are there, discuss?**

**Answer :-** A binary tree is a special type of tree in which every node or vertex has either no child node or one child node or two child nodes. A binary tree is an important class of a tree data structure in which a node can have at most two children. Child node in a binary tree on the left is termed as 'left child node' and node in the right is termed as the 'right child node.' In the figure mentioned below, the root node 8 has two children 3 and 10; then this two child node again acts as a parent node for 1 and 6 for left parent node 3 and 14 for right parent node 10. Similarly, 6 and 14 has a child node.



A binary tree may also be defined as follows:

- A binary tree is either an empty tree
- Or a binary tree consists of a node called the root node, a left subtree and a right subtree, both of which will act as a binary tree once again
- **Applications of Binary Tree**

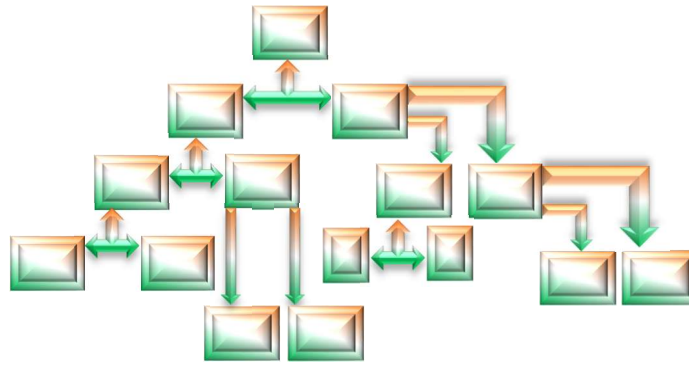
Binary trees are used to represent a nonlinear data structure. There are various forms of Binary trees. Binary trees play a vital role in a software application. One of the most important applications of the Binary tree is in the searching algorithm.

A **general tree** is defined as a nonempty finite set T of elements called nodes such that:

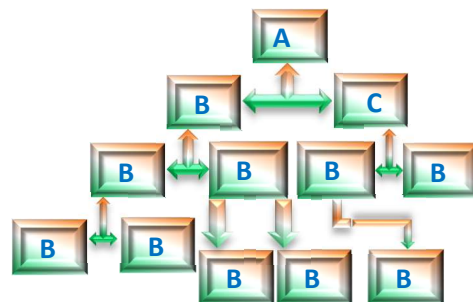
- The tree contains the root element
- The remaining elements of the tree form an ordered collection of zeros and more disjoint trees  $T_1, T_2, T_3, T_4, \dots, T_n$  which are called subtrees.

**Types of Binary Trees are**

- A **full binary tree** which is also called as proper binary tree or 2-tree is a tree in which all the node other than the leaves has exact two children.



- A **complete binary tree** is a binary tree in which at every level, except possibly the last, has to be filled and all nodes are as far left as possible.



- A binary tree can be converted into an **extended binary tree** by adding new nodes to its leaf nodes and to the nodes that have only one child. These new nodes are added in such a way that all the nodes in the resultant tree have either zero or two children. It is also called 2 - tree.
- **The threaded Binary tree** is the tree which is represented using pointers the empty subtrees are set to NULL, i.e. 'left' pointer of the node whose left child is empty subtree is normally set to NULL. These large numbers of pointer sets are used in different ways.

2.)

### B.) Question :- What is a List Structure? Explain Adjacency list and Incidence list?

**Answer :-** it is important to know the concept of a list.

A list is a sequential data structure, ie. a collection of items accessible one after another beginning at the head and ending at the tail. In other words, a **list** is a mutable structure that stores either primitive values ( int, char, etc) or objects. There are two common implementation methods of lists. They are:

#### 1. **Static implementation of list:**

Static implementation stores the elements of list in contiguous memory location. The position of each elements in the list are given by an index from 0 to n-1, where n is the number of elements in the list. In this implementation, memory allocation for the list should be done at the creation time , i.e the size of the list should be known in advance.

There are certain problems with static implementation of lists :

#### # **Insertion and deletion are time consuming and memory consuming.**

The insertion and deletion of an element to and from the last index of the list is not problematic, but insertion and deletion at the specified index is very expensive. for example, inserting at postion 0 ( i.e a new first element) requires first pushing the entire array elements one step right to make room for the new element, whereas deleting element from first index requires shifting entire array elements one step left to cover up the empty space created by deletion.

## 2. Dynamic implementation of list (Linked List):

This implementation allocates the elements of list dynamically (and elements allocated dynamically does not demand contiguous memory location). So in dynamic allocation it is not necessary to know the size of the list in advance. The list elements are not stored in contiguous memory location though they belong to the same list, they are linked to each other by address of next elements.

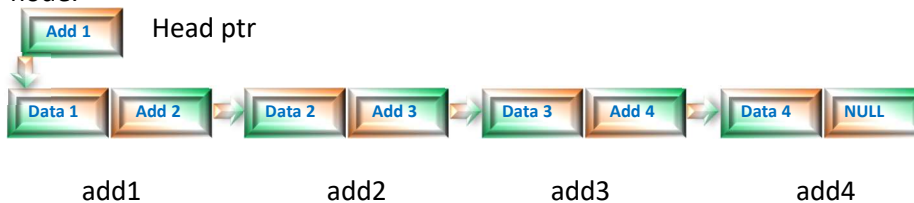
So, **linked list** is a linear data structure in which list elements are dynamically allocated and in which elements point to each other to define a linear relationship. The elements of linked lists are called **nodes**.

Each nodes has minimum 2 fields : Data field (also called info field) and pointer field.

\* **Data field** stores user provided input values.

\* **Pointer field** stores the address of its successor node.

The first node in the linked list is pointed by head pointer which contains the address of the first node.



3.)

**Question :- Discuss the types of directed graphs and matrix representation of Digraph?**

**Answer :-** A graph  $G = (V, E)$  where  $v = \{0, 1, 2, \dots, n-1\}$  can be represented using two dimensional integer array of size  $n \times n$ . `int adj[20][20]` can be used to store a graph with 20 vertices

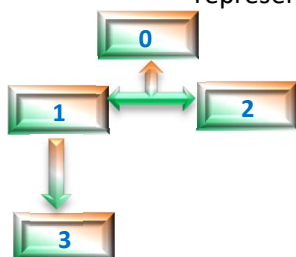
`adj[i][j] = 1`, indicates presence of edge between two vertices  $i$  and  $j$ .

`adj[i][j] = 0`, indicates absence of edge between two vertices  $i$  and  $j$ .

- A graph is represented using square matrix.
- Adjacency matrix of an undirected graph is always a symmetric matrix, i.e. an edge  $(i, j)$  implies the edge  $(j, i)$ .
- Adjacency matrix of a directed graph is never symmetric, `adj[i][j] = 1` indicates a directed edge from vertex  $i$  to vertex  $j$ .

An example of adjacency matrix

representation of an undirected and directed graph is given below:



	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	1	0	0
3	0	1	0	0

# SET-11

4.)

a.) Question :- Explain the algorithms of Bubble sort and Merge sort?

**Answer :-** Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$  where  $n$  is the number of items.

How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes  $O(n^2)$  time so we're keeping it short and precise.

14	33	27	35	10
----	----	----	----	----

Bubble sort starts with very first two elements, comparing them to check which one is greater.

14	33	27	35	10
----	----	----	----	----

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

14	33	27	35	10
----	----	----	----	----

We find that 27 is smaller than 33 and these two values must be swapped.

14	33	27	35	10
----	----	----	----	----

The new array should look like this –

14	33	27	35	10
----	----	----	----	----

Next we compare 33 and 35. We find that both are in already sorted positions.

14	27	33	35	10
----	----	----	----	----

Then we move to the next two values, 35 and 10.

14	33	27	35	10
----	----	----	----	----

We know then that 10 is smaller 35. Hence they are not sorted.

14	33	27	35	10
----	----	----	----	----

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

14	33	27	35	10
----	----	----	----	----

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

14	27	10	33	35
----	----	----	----	----

Notice that after each iteration, at least one value moves at the end.

14	10	27	33	35
----	----	----	----	----

And when there's no swap required, bubble sort learns that an array is completely sorted.

14	10	27	33	35
----	----	----	----	----

4.)

**b.) Question :- What are the characteristics and Building Blocks of an Algorithm?**

**Answer :-** As computer programmers, we are constantly using algorithms, whether it's an existing algorithm for a common problem, like sorting an array, or if it's a completely new algorithm unique to our program. By understanding algorithms, we can make better decisions about which existing algorithms to use and learn how to make new algorithms that are correct and efficient.

An algorithm is made up of three basic building blocks: sequencing, selection, and iteration.

**Sequencing:** An algorithm is a step-by-step process, and the order of those steps are crucial to ensuring the correctness of an algorithm.

Here's an algorithm for translating a word into Pig Latin, like from "pig" to "ig-pay":

1. Append "-".    2. Append first letter    3. Append "ay"    4. Remove first letter

Algorithms can use selection to determine a different set of steps to execute based on a Boolean expression.

Here's an improved algorithm for Pig Latin that handles words that start with vowels, so that "eggs" becomes "eggs-yay" instead of the unpronounceable "ggs-eay":

1. Append "-"    2. Store first letter    3. If first letter is vowel:    a. Append "yay"  
4. Otherwise:    a. Append first letter    b. Append "ay"    c. Remove first letter

Algorithms often use repetition to execute steps a certain number of times or until a certain condition is met.

We can add iteration to the previous algorithm to translate a complete phrase, so that "peanut butter and jelly" becomes "eanut-pay utter-bay and-yay elly-jay":

1. Store list of words  
2. For each word in words:

- a. Append hyphen
- b. If first letter is vowel:
  - i. Append "yay"
- c. Otherwise:
  - i. Append first letter
  - ii. Append "ay"
  - iii. Remove first letter

By combining sequencing, selection, and iteration, we've successfully come up with an algorithm for Pig Latin translation.

5.)

**a.) Question :- How is the Efficiency of an Algorithm measured?**

**Answer :-** Computer resources are limited that should be utilized efficiently. The efficiency of an algorithm is defined as the number of computational resources used by the algorithm. An algorithm must be analyzed to determine its resource usage. The efficiency of an algorithm can be measured based on the usage of different resources.

For maximum efficiency of algorithm we wish to minimize resource usage. The important resources such as time and space complexity cannot be compared directly, so time and space complexity could be considered for an algorithmic efficiency.

**Method for determining Efficiency**

The efficiency of an algorithm depends on how efficiently it uses time and memory space.

The time efficiency of an algorithm is measured by different factors. For example, write a program for a defined algorithm, execute it by using any programming language, and measure the total time it takes to run. The execution time that you measure in this case would depend on a number of factors such as:

- Speed of the machine
- Compiler and other system Software tools
- Operating System
- Programming language used
- Volume of data required

However, to determine how efficiently an algorithm solves a given problem, you would like to determine how the execution time is affected by the nature of the algorithm. Therefore, we need to develop fundamental laws that determine the efficiency of a program in terms of the nature of the underlying algorithm.

**Space-Time tradeoff :-** A space-time or time-memory tradeoff is a way of solving in less time by using more storage space or by solving a given algorithm in very little space by spending more time.

To solve a given programming problem, many different algorithms may be used. Some of these algorithms may be extremely time-efficient and others extremely space-efficient.



Time/space trade off refers to a situation where you can reduce the use of memory at the cost of slower program execution, or reduce the running time at the cost of increased memory usage.

### **Asymptotic Notations**

Asymptotic Notations are languages that uses meaningful statements about time and space complexity. The following three asymptotic notations are mostly used to represent time complexity of algorithms:

#### **(i) Big O**

Big O is often used to describe the worst-case of an algorithm.

#### **(ii) Big $\Omega$**

Big Omega is the reverse Big O, if Big O is used to describe the upper bound (worst - case) of a asymptotic function, Big Omega is used to describe the lower bound (best-case).

#### **(iii) Big $\Theta$**

When an algorithm has a complexity with lower bound = upper bound, say that an algorithm has a complexity  $O(n \log n)$  and  $\Omega(n \log n)$ , it's actually has the complexity  $\Theta(n \log n)$ , which means the running time of that algorithm always falls in  $n \log n$  in the best-case and worst-case.

### **Best, Worst, and Average case Efficiency**

Let us assume a list of  $n$  number of values stored in an array. Suppose if we want to search a particular element in this list, the algorithm that search the key element in the list among  $n$  elements, by comparing the key element with each element in the list sequentially.

The best case would be if the first element in the list matches with the key element to be searched in a list of elements. The efficiency in that case would be expressed as  $O(1)$  because only one comparison is enough.

Similarly, the worst case in this scenario would be if the complete list is searched and the element is found only at the end of the list or is not found in the list. The efficiency of an algorithm in that case would be expressed as  $O(n)$  because  $n$  comparisons required to complete the search.

The average case efficiency of an algorithm can be obtained by finding the average number of comparisons as given below:

Minimum number of comparisons = 1 Maximum number of comparisons =  $n$

If the element not found then maximum

number of comparison =  $n$

Therefore, average number of comparisons =  $(n + 1)/2$

Hence the average case efficiency will be expressed as  $O(n)$ .

5.)

**b.) Question :- What is Divide and conquer strategy? Discuss the binary search algorithm?**

**Answer :-** Consider an array of sorted numbers, with  $n$  elements. The simplest searching algorithm available is the linear sort. In this algorithm, we start from the leftmost element and compare it with the search term; if the search term matches the number on the index we are currently on, then the search operation is successful and the index is returned, but, if the numbers don't match, then we go to the number on the next index and follow the same procedure till the number is found. If the number isn't present, we return that the search was unsuccessful. The time complexity of linear sort is  $O(n)$ . This may hence take enormous time when there are many inputs.

This is when we need a divide and conquer strategy to reduce the time taken by the search procedure. Binary search is one such divide and conquer algorithm to assist with the given problem; note that a sorted array should be used in this case too. This search algorithm recursively divides the array into two sub-arrays that may contain the search term. It discards one of the sub-array by utilising the fact that items are sorted. It continues halving the sub-arrays until it finds the search term or it narrows down the list to a single item. Since binary search discards the sub-array it's pseudo Divide & Conquer algorithm. What makes binary search efficient is the fact that if it doesn't find the search term in each iteration, it just reduces the array/list to it's half for the next iteration. The time complexity of binary search is  $O(\log n)$ , where  $n$  is the number of elements in an array. If the search term is at the centre of the array, it's considered to be the best case since the element is found instantly in a go. Hence the best case complexity will be  $O(1)$ .

Now, consider the above-mentioned time complexities. Complexities like  $O(1)$  and  $O(n)$  are very intuitive to understand:

1.  **$O(1)$**  : refers to an operation where the value/the element is accessed directly
2.  **$O(n)$**  : refers to a (set of) where the element can only be accessed by traversing a set of  $n$  elements, like in linear search.


But what does  **$O(\log n)$**  really mean? It may seem difficult to understand but let's visualize it using a simple example of binary search, while searching for a number in a sorted array which will take the worst-case time complexity:

1.  $n = 16$  (no. of elements in the **sorted array**)

1	3	5	8	12	13	15	16	18	20	22	30	40	50	55	67
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

2. The middle element is selected as the pivot

1	2	3	5	8	12	15	16	18	20	22	30	40	50	55	67
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



3. Since the array is already sorted, and 13 is less than the pivot element, the other half of the array is redundant and hence removed.

1	3	5	8	12	13	15	16
0	1	2	3	4	5	6	7

$16 * \frac{1}{2} = 8$

4. The procedure for finding the pivot (middle) element for every sub-array is repeated.

12	13	5	16
----	----	---	----

$8 * \frac{1}{2} = 4$

12	13
----	----

$4 * \frac{1}{2} = 2$

13
----

$2 * \frac{1}{2} = 1$

5. The searching range is halved after every comparison with the pivot element.

6. The array was divided 4 times to reach the required value in an array of 16 elements. Therefore,

6.)

**Question :- Discuss about All Pair Shortest Paths and Travelling Salesman Problem?**

**Answer :-** Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For  $n$  number of vertices in a graph, there are  $(n - 1)!$  number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph  $G = (V, E)$ , where  $V$  is a set of cities and  $E$  is a set of weighted edges. An edge  $e(u, v)$  represents that vertices  $u$  and  $v$  are connected. Distance between vertex  $u$  and  $v$  is  $d(u, v)$ , which should be non-negative.

Suppose we have started at city  $1$  and after visiting some cities now we are in city  $j$ . Hence, this is a partial tour. We certainly need to know  $j$ , since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities  $S \in \{1, 2, 3, \dots, n\}$  that includes  $1$ , and  $j \in S$ , let  $C(S, j)$  be the length of the shortest path visiting each node in  $S$  exactly once, starting at  $1$  and ending at  $j$ .

When  $|S| > 1$ , we define  $C(S, 1) = \infty$  since the path cannot start and end at  $1$ .

Now, let express  $C(S, j)$  in terms of smaller sub-problems. We need to start at  $1$  and end at  $j$ . We should select the next city in such a way that

$$C(S, j) = \min_{i \in S, i \neq j} [C(S - \{j\}, i) + d(i, j)] \text{ where } i \in S \text{ and } i \neq j$$

Algorithm: Traveling-Salesman-Problem

$C(\{1\}, 1) = 0$

for  $s = 2$  to  $n$  do

for all subsets  $S \in \{1, 2, 3, \dots, n\}$  of size  $s$  and containing 1

$C(S, 1) = \infty$

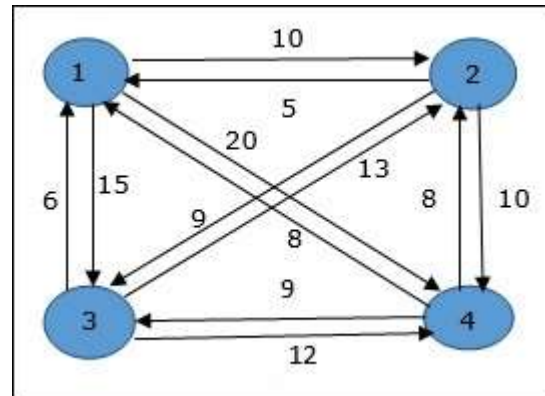
for all  $j \in S$  and  $j \neq 1$

$C(S, j) = \min \{C(S - \{j\}, i) + d(i, j) \text{ for } i \in S \text{ and } i \neq j\}$

Return  $\min_j C(\{1, 2, 3, \dots, n\}, j) + d(j, 1)$

Analysis :- There are at the most  $2n \cdot n \cdot n$  sub-problems and each one takes linear time to solve. Therefore, the total running time is  $O(2n \cdot n^2)O(2n \cdot n^2)$ .

Example :- In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$S = \Phi$

$\text{Cost}(2, \Phi, 1) = d(2, 1) = 5$   $\text{Cost}(2, \Phi, 1) = d(2, 1) = 5$   $\text{Cost}(2, \Phi, 1) = d(2, 1) = 5$   $\text{Cost}(2, \Phi, 1) = d(2, 1) = 5$

$\text{Cost}(3, \Phi, 1) = d(3, 1) = 6$   $\text{Cost}(3, \Phi, 1) = d(3, 1) = 6$   $\text{Cost}(3, \Phi, 1) = d(3, 1) = 6$   $\text{Cost}(3, \Phi, 1) = d(3, 1) = 6$

$\text{Cost}(4, \Phi, 1) = d(4, 1) = 8$   $\text{Cost}(4, \Phi, 1) = d(4, 1) = 8$   $\text{Cost}(4, \Phi, 1) = d(4, 1) = 8$   $\text{Cost}(4, \Phi, 1) = d(4, 1) = 8$

$S = 1$

$\text{Cost}(i, s) = \min \{ \text{Cost}(j, s - \{j\}) + d[i, j] \}$   $\text{Cost}(i, s) = \min \{ \text{Cost}(j, s - \{j\}) + d[i, j] \}$   $\text{Cost}(i, s) = \min \{ \text{Cost}(j, s - \{j\}) + d[i, j] \}$

$\text{Cost}(2, \{3\}, 1) = d[2, 3] + \text{Cost}(3, \Phi, 1) = 9 + 6 = 15$   $\text{cost}(2, \{3\}, 1) = d[2, 3] + \text{cost}(3, \Phi, 1) = 9 + 6 = 15$   $\text{Cost}(2, \{3\}, 1) = d[2, 3] + \text{Cost}(3, \Phi, 1) = 9 + 6 = 15$

$\text{Cost}(2, \{4\}, 1) = d[2, 4] + \text{Cost}(4, \Phi, 1) = 10 + 8 = 18$   $\text{cost}(2, \{4\}, 1) = d[2, 4] + \text{cost}(4, \Phi, 1) = 10 + 8 = 18$   $\text{Cost}(2, \{4\}, 1) = d[2, 4] + \text{Cost}(4, \Phi, 1) = 10 + 8 = 18$

$\text{Cost}(3, \{2\}, 1) = d[3, 2] + \text{Cost}(2, \Phi, 1) = 13 + 5 = 18$   $\text{cost}(3, \{2\}, 1) = d[3, 2] + \text{cost}(2, \Phi, 1) = 13 + 5 = 18$   $\text{Cost}(3, \{2\}, 1) = d[3, 2] + \text{Cost}(2, \Phi, 1) = 13 + 5 = 18$

$$\text{Cost}(3, \{4\}, 1) = d[3, 4] + \text{Cost}(4, \Phi, 1) = 12 + 8 = 20$$

$$\text{Cost}(4, \{3\}, 1) = d[4, 3] + \text{Cost}(3, \Phi, 1) = 9 + 6 = 15$$

$$\text{Cost}(4, \{2\}, 1) = d[4, 2] + \text{Cost}(2, \Phi, 1) = 8 + 5 = 13$$

**S = 2**

$$\text{Cost}(2, \{3, 4\}, 1) = \min \{ d[2, 3] + \text{Cost}(3, \{4\}, 1) = 9 + 20 = 29, d[2, 4] + \text{Cost}(4, \{3\}, 1) = 10 + 15 = 25 \} = 25$$

$$\text{Cost}(3, \{2, 4\}, 1) = \min \{ d[3, 2] + \text{Cost}(2, \{4\}, 1) = 13 + 18 = 31, d[3, 4] + \text{Cost}(4, \{2\}, 1) = 12 + 13 = 25 \} = 25$$

$$\text{Cost}(4, \{2, 3\}, 1) = \min \{ d[4, 2] + \text{Cost}(2, \{3\}, 1) = 8 + 15 = 23, d[4, 3] + \text{Cost}(3, \{2\}, 1) = 9 + 18 = 27 \} = 23$$

**S = 3**

$$\text{Cost}(1, \{2, 3, 4\}, 1) = \min \{ d[1, 2] + \text{Cost}(2, \{3, 4\}, 1) = 10 + 25 = 35, d[1, 3] + \text{Cost}(3, \{2, 4\}, 1) = 15 + 25 = 40, d[1, 4] + \text{Cost}(4, \{2, 3\}, 1) = 20 + 23 = 43 \} = 35$$

The minimum cost path is 35.

Start from cost  $\{1, \{2, 3, 4\}, 1\}$ , we get the minimum value for  $d[1, 2]$ . When  $s = 3$ , select the path from 1 to 2 (cost is 10) then go backwards. When  $s = 2$ , we get the minimum value for  $d[4, 2]$ . Select the path from 2 to 4 (cost is 10) then go backwards.

When  $s = 1$ , we get the minimum value for  $d[4, 3]$ . Selecting path 4 to 3 (cost is 9), then we shall go to then go to  $s = \Phi$  step. We get the minimum value for  $d[3, 1]$  (cost is 6).

