1.a

ADT

An abstract data type is an **abstraction** of a data structure that provides **only the interface** to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.

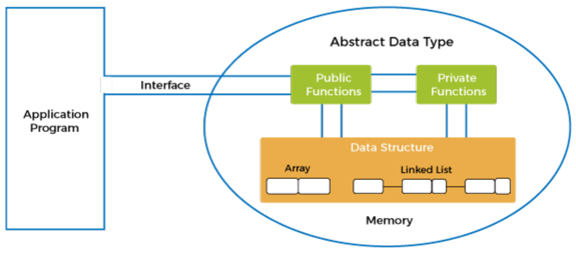
In other words, we can say that abstract data types are the entities that are definitions of data and operations but do not have implementation details. In this case, we know the data that we are storing and the operations that can be performed on the data, but we don't know about the implementation details. The reason for not having implementation details is that every programming language has a different implementation strategy for example; a C data structure is implemented using structures while a C++ data structure is implemented using objects and classes.

Abstract data type model

Before knowing about the abstract data type model, we should know about abstraction and encapsulation.

Abstraction: It is a technique of hiding the internal details from the user and only showing the necessary details to the user.

Encapsulation: It is a technique of combining the data and the member function in a single unit is known as encapsulation.



The above figure shows the ADT model. There are two types of models in the ADT model, i.e., the public function and the private function. The ADT model also contains the data structures that we are using in a program. In this model, first encapsulation is performed, i.e., all the data is wrapped in a single unit, i.e., ADT. Then, the abstraction is performed means showing the operations that can be performed on the data structure and what are the data structures that we are using in a program.The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented. So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Some examples are List ADT, Stack ADT, Queue ADT.

1b. Infix: (P+Q*R/S)+T*U-(V*W+X-Y)

Sr No	Expression	Stack	Postfix
0	(	(	
1	P	(	P
2	+	(+	P
3	Q	(+	PQ
4	*	(+*	PQ
5	R	(+*	PQR
6	/	(+/	PQR*
7	S	(+/	PQR*S
8	)		PQR*S/+
9	+	+	PQR*S/+
10	T	+	PQR*S/+T
11	*	+*	PQR*S/+T
12	U	+*	PQR*S/+TU
13	-	-	PQR*S/+TU*+
14	(	-(	PQR*S/+TU*+
15	V	-(	PQR*S/+TU*+V
16	*	-(*	PQR*S/+TU*+V
17	W	-(*	PQR*S/+TU*+VW
18	+	-(+	PQR*S/+TU*+VW*
19	X	-(+	PQR*S/+TU*+VW*X
20	-	-(-	PQR*S/+TU*+VW*X+
21	Y	-(-	PQR*S/+TU*+VW*X+Y
22	)	-	PQR*S/+TU*+VW*X+Y-
23			PQR*S/+TU*+VW*X+Y--

Postfix: PQR*S/+TU*+VW*X+Y--

2a.

Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller subproblems. It involves solving a problem by reducing it to a simpler version of the same problem.

Factorial of a Positive integer

```
#include<iostream>
using namespace std;

int add(int n);

int main() {
    int n;

    cout << "Enter a positive integer: ";
    cin >> n;
    cout << "Sum =  " << add(n);
    return 0;
}
int add(int n) {
    if(n != 0)
        return n + add(n - 1);
    return 0;
}
```

2b. <u>Insert and Delete in Circular Queue</u>

Enqueue operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

There are two scenarios in *which queue is not full*:

- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which *the element cannot be inserted:*

- When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- front== rear + 1;

Algorithm to insert an element in a circular queue

```
Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT
```

Dequeue Operation

The steps of dequeue operation are given below:

- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

```
Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]

Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]

Step 4: EXIT
```

3b. Insert and Delete at the beginning of the singly linked list

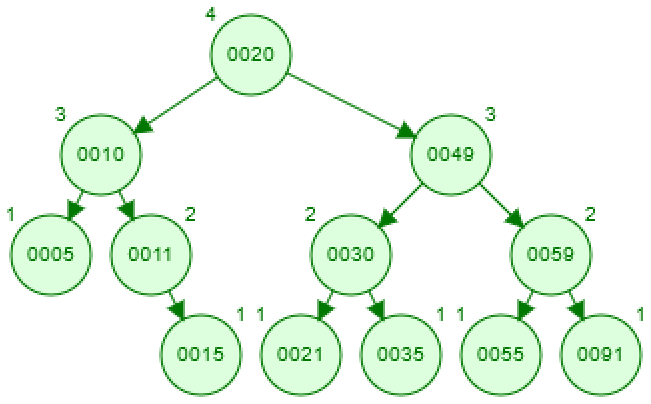
```
void insertAtBeginning(int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = head;
    head = newNode;
}
```

```
void deleteFromBeginning() {
        if (head == nullptr) {
            return;
        }
        Node* temp = head;
        head = head->next;
        delete temp;
    }
```

4a.

AVL Tree

10, 20, 30, 11, 5, 59, 91, 49, 55, 21, 35, 15



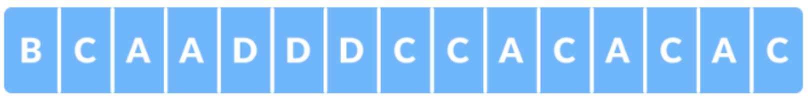
4b. Huffman Algorithm

Huffman coding is an algorithm for compressing data with the aim of reducing its size without losing any of the details. This algorithm was developed by **David Huffman**.

Huffman coding is typically useful for the case where data that we want to compress has frequently occurring characters in it. Huffman coding is a specific implementation of variable-length encoding. It is a well-known algorithm for constructing optimal prefix codes based on variable-length encoding principles.

How it works?

Let assume the string data given below is the data we want to compress -



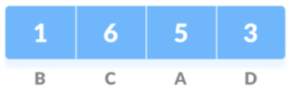
The length of the above string is 15 characters and each character occupies a space of 8 bits. Therefore, a total of 120 bits (8 bits x 15 characters) is required to send this string over a network. We can reduce the size of the string to a smaller extent using Huffman Coding Algorithm.

In this algorithm first we create a tree using the frequencies of characters and then assign a code to each character. The same resulting tree is used for decoding once encoding is done.

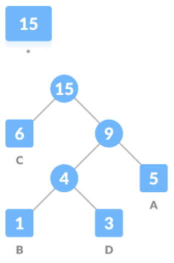
Using the concept of prefix code, this algorithm avoids any ambiguity within the decoding process, i.e. a code assigned to any character shouldn't be present within the prefix of the opposite code.

Steps to Huffman Coding

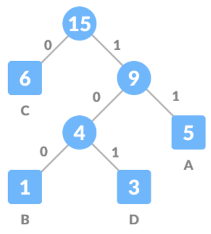
1. First, we calculate the count of occurrences of each character in the string.



Repeat for C -



8. We got our resulting tree, now we assign **0 to the left edge** and **1 to the right edge** of every non-leaf node.



9. Now for generating codes of each character we traverse towards each leaf node representing some character from the root node and form code of it.

All the data we gathered until now is given below in tabular form -

Character	Frequency in string	Assigned Code	Size
B	1	**100**	1 x 3 = 3 bits
D	3	**101**	3 x 3 = 9 bits
A	5	**11**	5 x 2 = 10 bits
C	6	**0**	6 x 1 = 6 bits
4 x 8 = 32 bits	Total = 15 bits		Total = 28 bits

Before compressing the total size of the string was **120 bits.** After compression that size was reduced to **75 bits** (28 bits + 15 bits + 32 bits).

5a. QUICK SORT

The arrangement of data in a preferred order is called sorting in the data structure. By sorting data, it is easier to search through it quickly and easily. The simplest example of sorting is a dictionary. Before the era of the Internet, when you wanted to look up a word in a dictionary, you would do so in alphabetical order. This made it easy.

Implementation of Quick sort

```
#include <iostream>

using namespace std;

int partition (int a[], int start, int end)
{
    int pivot = a[end]; // pivot element
    int i = (start - 1);

    for (int j = start; j <= end - 1; j++)
    {
        // If current element is smaller than the pivot
        if (a[j] < pivot)
        {
            i++; // increment index of smaller element
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i+1];
    a[i+1] = a[end];
    a[end] = t;
    return (i + 1);
}

void quick(int a[], int start, int end) /* a[] = array to be sorted, start = Starting index, end = Ending index */
{
    if (start < end)
    {
        int p = partition(a, start, end);  //p is the partitioning index
        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}

void printArr(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout<<a[i]<< " ";
}
int main()
{
    int a[] = { 23, 8, 28, 13, 18, 26 };
    int n = sizeof(a) / sizeof(a[0]);
    cout<<"Before sorting array elements are - \n";
    printArr(a, n);
    quick(a, 0, n - 1);
    cout<<"\nAfter sorting array elements are - \n";
    printArr(a, n);

    return 0;
}
```

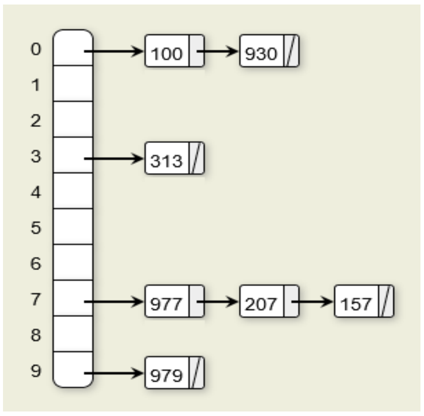
5b. Collision resolution techniques

5.1 Open Hashing

Open Hashing is popularly known as Separate Chaining. This is a technique which is used to implement an array as a linked list known as a chain. It is one of the most used techniques by programmers to handle collisions.

5.1.1 Separate Chaining

When two or more string hash to the same location (hash key), we can append them into a singly linked list (called chain) pointed by the hash key. A hash table then is an array of lists. It is called open hashing because it uses extra memory to resolve collision.



Advantages of Open Hashing:

1. Simple to implement
2. Hash table never fills up; we can always add more elements to the chain.
3. Less sensitive to the function or load factors.
4. It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages of Open Hashing:

1. The cache performance of the Separate Chaining method is poor as the keys are stored using a singly linked list.
2. A lot of storage space is wasted as some parts of the hash table are never used.
3. In the worst case, the search time can become " O (n) ". This happens only if the chain becomes too lengthy.
4. Uses extra space for links.

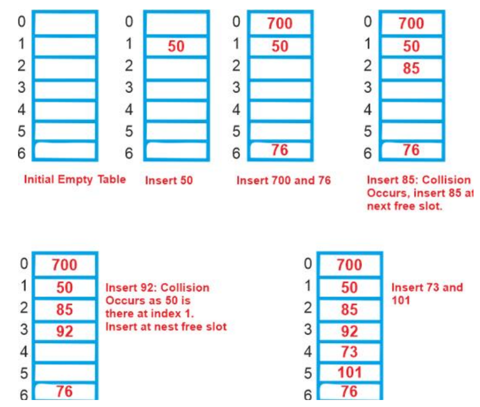
5.2 Closed Hashing

Similar to separate chaining, open addressing is a technique for dealing with collisions. In Open Addressing, the hash table alone houses all of the elements. The size of the table must therefore always be more than or equal to the total number of keys at all times (Note that we can increase table size by copying old data if needed). This strategy is often referred to as **closed hashing**.

5.2.1 Linear Probing

Defination: insert K_i at first free location from (u+i)%m where i = 0 to (m-1)
Another technique of collision resolution is a linear probing. If we cannot insert at index k, we try the next slot k+1. If that one is occupied, we go to k+2, and so on. This is quite simple approach but it requires new thinking about hash tables. Do you always find an empty slot? What do you do when you reach the end of the table?

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85,
92, 73, 101.


0 1 2 3 4 5 6
Initial Empty Table
50
Insert 50
700
50
76
Insert 700 and 76
700
50
85
76
Insert 85: Collision Occurs, insert 85 at next free slot.
700
50
85
92
76
Insert 92: Collision Occurs as 50 is there at index 1. Insert at nest free slot
700
50
85
92
73
101
76
Insert 73 and 101

Clustering

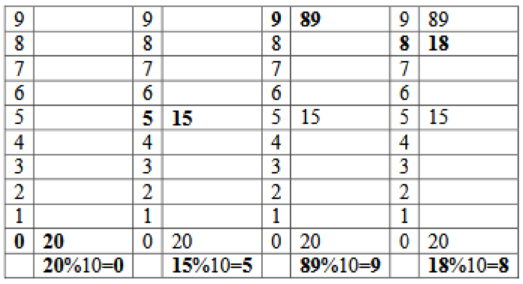
The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

Quadratic Probing

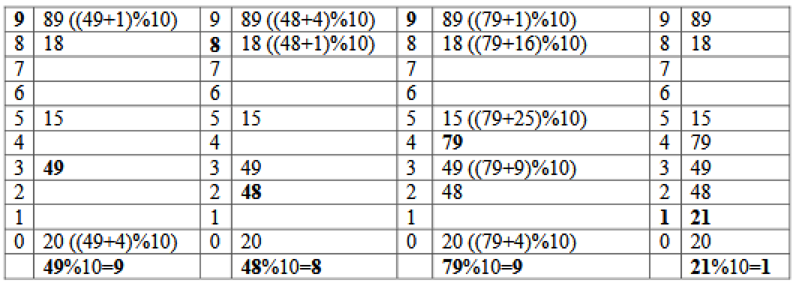
Defination: insert K_i at first free location from $(u+i^2)$%m where i=0 to (m-1). This method makes an attempt to correct the problem of clustering with linear probing. It forces the problem key to move quickly a considerable distance from the initial collision. When a key value hashes and collision occurs for key, this method probes the table location at:

(h (k) + 12) % table-size,
(h (k) + 22) % table-size,
(h (k) + 33) % table-size and so on.
That is, the first rehash adds 1 to the hash value. The second rehash adds 4, the third adds 9 and so on.
It reduces primary clustering but suffers from secondary clustering : hash that hash to some initial slot will probe the same alternative cells.

There is no guarantee to finding an empty location once the table gets more than half full.



9		9		**9**	**89**	9	89
8		8		8		**8**	**18**
7		7		7		7	
6		6		6		6	
5		**5**	**15**	5	15	5	15
4		4		4		4	
3		3		3		3	
2		2		2		2	
1		1		1		1	
0	**20**	0	20	0	20	0	20
	20%10=0		**15%10=5**		**89%10=9**		**18%10=8**



9	89 ((49+1)%10)	9	89 ((48+4)%10)	**9**	89 ((79+1)%10)	9	89
8	18	**8**	18 ((48+1)%10)	8	18 ((79+16)%10)	8	18
7		7		7		7	
6		6		6		6	
5	15	5	15	5	15 ((79+25)%10)	5	15
4		4		4	**79**	4	79
3	**49**	3	49	3	49 ((79+9)%10)	3	49
2		2	**48**	2	48	2	48
1		1		1		**1**	**21**
0	20 ((49+4)%10)	0	20	0	20 ((79+4)%10)	0	20
	49%10=9		**48%10=8**		**79%10=9**		**21%10=1**

5.2.3 Double Hashing

Defination: insert K_i at first free place from (u+v*i)%m where i=0 to (m-1) & u= h_1k%m and v=h_2k%m

Double hashing is a collision resolution technique used in hash tables. It works by using two hash functions to compute two different hash values for a given key. The first hash function is used to compute the initial hash value, and the second hash function is used to compute the step size for the probing sequence.

Double hashing has the ability to have a low collision rate, as it uses two hash functions to compute the hash value and the step size. This means that the probability of a collision occurring is lower than in other collision resolution techniques such as linear probing or quadratic probing.

Advantages of Double hashing

- The advantage of Double hashing is that it is one of the best forms of probing, producing a uniform distribution of records throughout a hash table.
- This technique does not yield any clusters.
- It is one of the effective methods for resolving collisions.

Double hashing can be done using :
(hash1(key) + i * hash2(key)) % TABLE_SIZE
Here hash1() and hash2() are hash functions and TABLE_SIZE
is size of hash table.
(We repeat by increasing i when collision occurs)

Example:

Insert the keys 79, 69, 98, 72, 14, 50 into the **Hash Table of size 13**. Resolve all collisions using Double Hashing where first hash-function is **h_1(k) = k mod 13** and second hash-function is **h_2(k) = 1 + (k mod 11)**

Solution:

Initially, all the hash table locations are empty. We pick our **first key = 79** and apply **h_1** (k) over it,

h_1(79) = 79 % 13 = 1, hence key 79 hashes to 1st location of the hash table. But before placing the key, we need to make sure the 1st1st location of the hash table is empty. In our case it is empty and we can easily place key 79 in there.

Second key = 69, we again apply h1(k) over it, **h_1(69) = 69 % 13 = 4**, since the 4th location is empty we can place 69 there.

Third key = 98, we apply h1(k) over it, **h_1(98) = 98 % 13 = 7**, 7th location is empty so 98 placed there.

Fourth key = 72, **h_1(72) = 72 % 13 = 7**, now this is a collision because the **7th** location is already occupied, we need to resolve this collision using double hashing.

h_{new} = [h_1(72) + i * h_2(72)] % 13
= [7 + 1 * (1 + 72 % 11)] % 13
= 1

Location **1st** is already occupied in the hash-table, hence we again had a collision. Now we again recalculate with i = 2

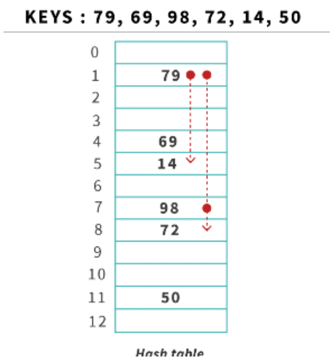
h_{new} = [h_1(72) + i * h_2(72)] % 13
= [7 + 2 * (1 + 72 % 11)] % 13
= 8

Location 8th is empty in hash-table and now we can place key 72 in there.

Fifth key = 14, h_1(14) = 14%13 = 1, now this is again a collision because 1st location is already occupied, we now need to resolve this collision using double hashing.

h_{new} = [h_1(14) + i * h_2(14)] % 13
= [1 + 1 * (1 + 14 % 11)] % 13
= 5

Location 5th is empty and we can easily place key 14 there.



Sixth key = 50, **h_1(50) = 50%13 = 11**, 11th location is already empty and we can place our key 50 there.

Since all the keys are placed in our hash table the double hashing procedure is completed. Finally, our hash table looks like the following,

6a.

Warshall Algorithm

Warshall's algorithm computes the transitive closure of a directed graph. It determines if there is a path from one node to another, considering all possible intermediate nodes. The algorithm maintains an adjacency matrix, initially filled with the graph's adjacency information. It iteratively updates the matrix by checking paths through each node, marking reachability. The process continues until all intermediate nodes are considered. The resulting matrix represents the transitive closure, indicating if there is a path between each pair of nodes.

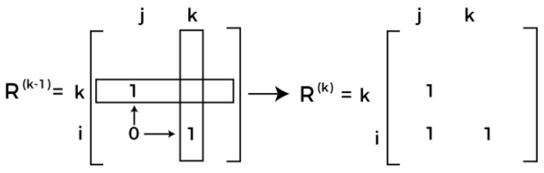
Recurrence relating elements R(k) to elements of R(k-1) can be described as follows:

R(k)[i, j] = R(k-1)[i, j] or (R(k-1)[i, k] and R(k-1)[k, j])

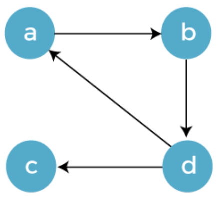
In order to generate R(k) from R(k-1), the following rules will be implemented:

Rule 1: In row i and column j, if the element is 1 in R(k-1), then in R(k), it will also remain 1.

Rule 2: In row i and column j, if the element is 0 in R(k-1), then the element in R(k) will be changed to 1 iff the element in its column j and row k, and the element in row i and column k are both 1's in R(k-1).



Application of Warshall's algorithm to the directed graph



In order to understand this, we will use a graph, which is described as follow:

For this graph R(0) will be looked like this:

$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

Here R(0) shows the adjacency matrix. In R(0), we can see the existence of a path, which has no intermediate vertices. We will get R(1) with the help of a boxed row and column in R(0).

In R(1), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 1, which means just a vertex. It contains a new path from d to b. We will get R(2) with the help of a boxed row and column in R(1).

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

In R(2), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 2, which means a and b. It contains two new paths. We will get R(3) with the help of a boxed row and column in R(2).

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

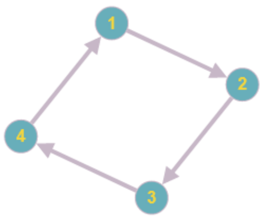
In R(3), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 3, which means a, b and c. It does not contain any new paths. We will get R(4) with the help of a boxed row and column in R(3).

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

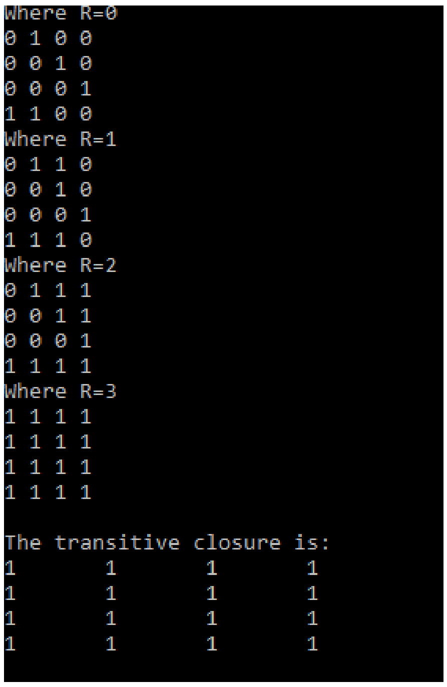
In R(4), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 4, which means a, b, c and d. It contains five new paths.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

i. Digraph



ii.



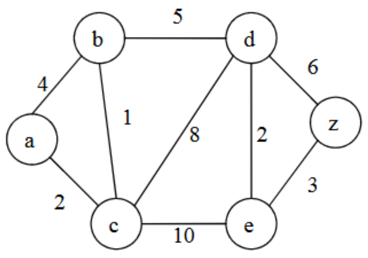
```
Where R=0
0 1 0 0
0 0 1 0
0 0 0 1
1 1 0 0
Where R=1
0 1 1 0
0 0 1 0
0 0 0 1
1 1 1 0
Where R=2
0 1 1 1
0 0 1 1
0 0 0 1
1 1 1 1
Where R=3
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1

The transitive closure is:
1       1       1       1
1       1       1       1
1       1       1       1
1       1       1       1
```

6.b

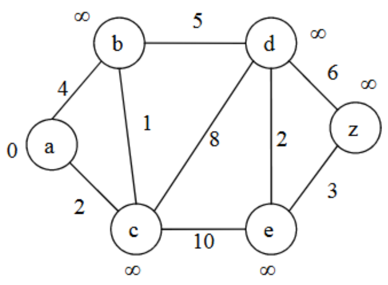
Dijkstra's Algorithm to find the shortest path

```
1. Mark all the vertices as unknown
2. for each vertex u keep a distance $d_u$ from source vertex s to u initially set to infinity except for s which is set to $d_s = 0$
3. repeat these steps until all vertices are known
    i. select a vertex u, which has the smallest $d_u$ among all the unknown vertices
    ii. mark u as visited
    iii. for each vertex v adjacent to u
            if v is unvisited and $d_u + cost(u, v) < d_v$
                update $d_v$ to $d_u + cost(u, v)$
```

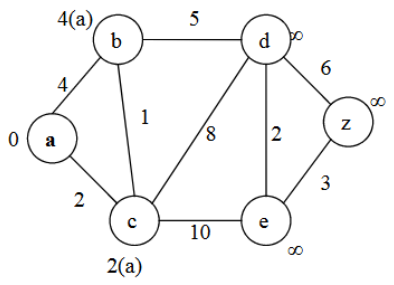
Example:



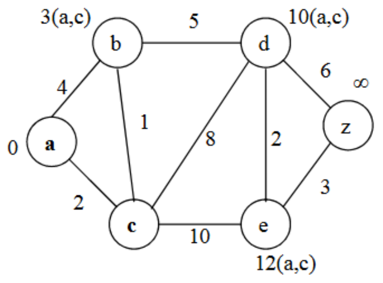
Given graph with weights



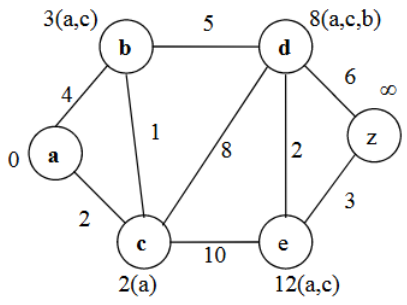
(a)



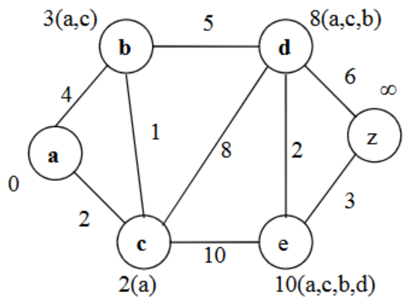
(b)



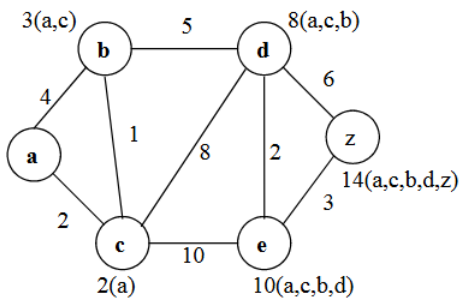
(c)



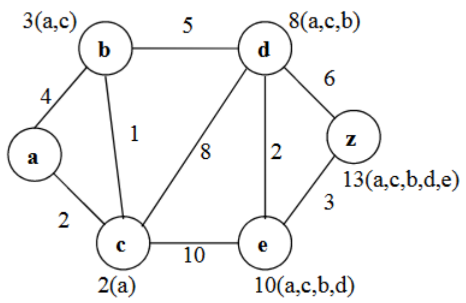
(d)



(e)



(f)



(g)

Fig: Using Dijkstra's Algorithm to find shortest path from a to z

7b. **Binary Search**

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

To do the binary search, first we have to sort the array elements. The logic behind this technique is given below.

1. First find the middle element of the array.
2. Compare the middle element with an item.
3. There are three cases.
    a). If it is a desired element then search is successful,
    b). If it is less than the desired item then search only in the first half of the array.
    c). If it is greater than the desired item, search in the second half of the array.
4. Repeat the same steps until an element is found or search area is exhausted.

In this way, at each step we reduce the length of the list to be searched by half.

Requirements

i. The list must be ordered
ii. Rapid random access is required, so we cannot use binary search for a linked list

Algorithm

```
Binary_Search(a, lower_bound, upper_bound, val)

Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
Step 2: repeat steps 3 and 4 while beg <=end
Step 3: set mid = (beg + end)/2
Step 4: if a[mid] = val
set pos = mid
print pos
go to step 6
else if a[mid] > val
set end = mid - 1
else
set beg = mid + 1
[end of if]
[end of loop]
Step 5: if pos = -1
print "value is not present in the array"
[end of if]
Step 6: exit
```

Binary Search complexity

Now, let's see the time complexity of Binary search in the best case, average case, and worst case. We will also see the space complexity of Binary search.

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

2. Space Complexity

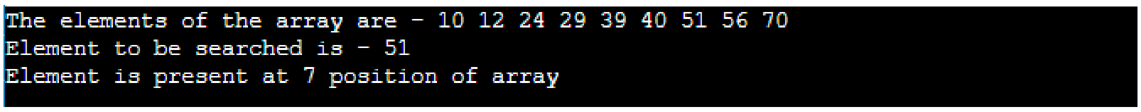
Space Complexity	O(1)

- The space complexity of binary search is O(1).

Implementation of Binary Search

```
#include <iostream>
using namespace std;
int binarySearch(int a[], int beg, int end, int val)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
/* if the item to be searched is present at middle */
        if(a[mid] == val)
        {
            return mid+1;
        }
            /* if the item to be searched is smaller than middle, the
n it can only be in left subarray */
        else if(a[mid] < val)
        {
            return binarySearch(a, mid+1, end, val);
        }
        /* if the item to be searched is greater than middle, then it
 can only be in right subarray */
    else
        {
            return binarySearch(a, beg, mid-1, val);
        }
    }
    return -1;
}
int main() {
  int a[] = {10, 12, 24, 29, 39, 40, 51, 56, 70}; // given array
  int val = 51; // value to be searched
  int n = sizeof(a) / sizeof(a[0]); // size of array
  int res = binarySearch(a, 0, n-1, val); // Store result
  cout<<"The elements of the array are - ";
  for (int i = 0; i < n; i++)
  cout<<a[i]<<" ";
  cout<<"\nElement to be searched is - "<<val;
  if (res == -1)
  cout<<"\nElement is not present in the array";
  else
  cout<<"\nElement is present at "<<res<<" position of array";

  return 0;
}
```

Output



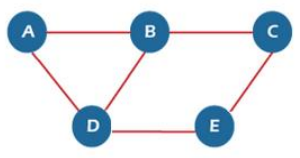
```
The elements of the array are - 10 12 24 29 39 40 51 56 70
Element to be searched is - 51
Element is present at 7 position of array
```

c. Adjacency Matrix

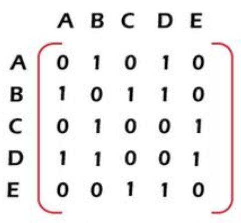
If an Undirected Graph G consists of n vertices then the adjacency matrix of a graph is an n x n matrix A = [a_{ij}] and defined by

$$a_{ij} = \begin{bmatrix} 1, \text{if } \{V_i, V_j\} \text{is an edge i.e., } v_i \text{is adjacent to } v_j \\ 0, \text{if there is no edge between } v_i \text{ and } v_j \end{bmatrix}$$

If there exists an edge between vertex v_i and v_j, where i is a row and j is a column then the value of a_{ij}=1.
If there is no edge between vertex v_i and v_j, then value of a_{ij}=0.

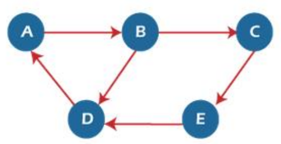


Undirected Graph

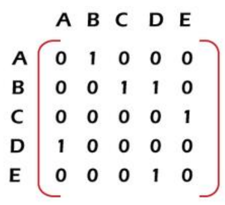


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

Adjacency Matrix

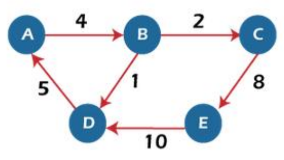


Directed Graph

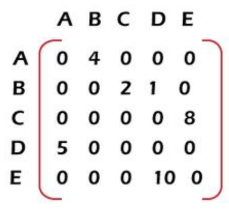


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 1 & 0 \end{array}$$

Adjacency Matrix



weighted Directed Graph



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

Adjacency Matrix