Company Desc:

Dotlines Bangladesh Ltd.

Address : Uday Tower, Level-12, 57 & 57/A, Gulshan Avenue, Dhaka-1212

Web : www.ssd-tech.io

Business: Incorporated in 2004, SSD-TECH is a Bangladesh-based technology solutions provider which provides custom-made and innovative solutions for telecommunication operators, enterprises, and caters to specific needs of the people of Bangladesh. The company provides Value Added Services platforms to the leading telecom operators of Bangladesh since 2008 and later launched Carnival Internet (home and corporate internet service) in 2015.

SSD-TECH steps into its 15th year in 2019, with big strides in technology-based direct-to-consumer solutions which will elevate and enhance the lives of the people of Bangladesh.

In the past 3 years, recognizing the need of the nation, SSD-TECH successfully transformed itself consciously from a B2B IT services company to a B2C technology-enabled services entity.

According to a valuation by LankaBangla Investments Limited in 2017, SSD-TECH had been valued at US$ 65 million (around BDT 525 crore).

Merge two sorted array

```
#include <iostream>
using namespace std;
void mergeSortedArraysInPlace(int* a, int n, int* b, int m) {
   int i = n - 1, j = m - 1, k = n + m - 1;
  while (i >= 0 && j >= 0) {
      if (a[i] > b[j]) {
         a[k] = a[i];
         i--;
      } else {
         a[k] = b[j];
         j--;
      }
      k--;
   }
   while (j >= 0) {
      a[k] = b[j];
      j--;
      k--;
   }
}

int main() {
   int a[6] = {1, 3, 5, 0, 0, 0};
   int b[3] = {2, 4, 6};
   mergeSortedArraysInPlace(a, 3, b, 3);
   for (int i = 0; i < 6; i++) {
      cout << a[i] << " ";
   }

   return 0;

}
```

This code defines a function mergeSortedArraysInPlace that takes two arrays of integers, a and b, along with their respective sizes n and m, and merges them in place into the array a.

The function uses three integer variables, i, j, and k, to keep track of the current index of each array. The initial value of i is set to n - 1 to point to the last element of the first array, and the initial value of j is set to m - 1 to point to the last element of the second array. The initial value of k is set to n + m - 1, which represents the last index of the merged array.
The function then enters a while loop that continues as long as both i and j are greater than or equal to zero. Within the loop, the function compares the values of the elements at the current indices of a and b, and assigns the larger value to the current index of a[k]. It then decrements i or j accordingly, depending on which array contributed the larger value, and decrements k to move on to the next index in a.

After the loop completes, there may be remaining elements in b that were not merged into a. To handle this case, the function enters a second while loop that continues as long as j is greater than or equal to zero. Within the loop, the function assigns the remaining elements of b to the current indices of a[k], and decrements both j and k to move on to the next index in a.

The main function then initializes two arrays, a and b, with values and sizes specified in the code. It calls the mergeSortedArraysInPlace function with the first three elements of a and all three elements of b. Finally, it prints the resulting merged array a using a for loop that iterates over all six indices of a.

Complexity: O(n+m) for equal size array – O(n2) time complexity
O(1) space complexity

Find the winner of the election

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

int findWinner(int arr[], int n) {
   unordered_map<int, int> mp;
   for (int i = 0; i < n; i++) {
      mp[arr[i]]++;
   }
   int max_votes = 0, winner = -1;
   for (auto it : mp) {
      if (it.second > max_votes) {
         max_votes = it.second;
         winner = it.first;
      }
   }
   return winner;
}

int main() {
   int arr[] = {1, 2, 3, 2, 2, 1, 3, 1, 1};
   int n = sizeof(arr) / sizeof(arr[0]);
   int winner = findWinner(arr, n);
   if (winner != -1) {
      cout << "Winner is candidate #" << winner << endl;
   } else {
      cout << "No winner found!" << endl;
   }
   return 0;
}
```

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

Minimum depth of binary tree
The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

```
class Solution {
public:
    int minDepth(TreeNode* root) {
        if(root == NULL) return 0;
        int l = minDepth(root->left);
        int r = minDepth(root->right);
        if(l == 0 || r == 0){
            return 1+l+r;
        }
        return min(l,r)+1;
    }
};
```

Time Complexity: O(n) ---------- n = number of nodes
Space Complexity: O(h) ---------- h = height of the tree

Linked list cycle found or not found

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode *slow = head;
        ListNode *fast = head;
        while(fast != NULL && fast->next!=NULL){
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast) return true;
        }
        return false;
    }
};
```

Time complexity: O(n)
Space complexity: O(1)

Find the missing integer from a serial array A[1,2…..,N]

Simple Approach

7 - SQL
8 - Database Design
9 – Puzzle
10 – code
11 – singleton design pattern
12 – try catch finally block
13 – thread problem

Palindrome check for a single linked list

Approach: 1. Find the middle element
2. reverse the second half of the list then compare the last and first value of the list if same or not

```
class Solution {
public:
    ListNode* middle(ListNode* head){
        ListNode* slow = head;
        ListNode* fast = head->next;
        while(fast != NULL && fast->next!= NULL){
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }

    ListNode* reverse(ListNode* head){
        if(head == NULL || head->next == NULL) return head;
        else{
            ListNode* newHead = reverse(head->next);
            head->next->next = head;
            head->next = NULL;
            return newHead;
        }
    }

    bool isPalindrome(ListNode* head) {
        if(head == NULL) return true;
        ListNode* mid = middle(head);
        ListNode* last = reverse(mid->next);
        ListNode* curr = head;
        while(last != NULL){
            if(last->val != curr->val) return false;
            curr = curr->next;
            last = last->next;
        }
        return true;
    }
};
```

Time complexity: O(n)
Space complexity: O(1)

OOP

1. Encapsulation – Hide the internal details and show only necessary info

```
using System;
public class BankAccount
{
   private string accountNumber;
   private double balance;

   public BankAccount(string accountNumber, double balance)
   {
      this.accountNumber = accountNumber;
      this.balance = balance;
   }

   public void Deposit(double amount)
   {
      balance += amount;
   }

   public void Withdraw(double amount)
   {
      if (balance >= amount)
      {
         balance -= amount;
      }
      else
      {
         Console.WriteLine("Insufficient balance");
      }
   }

   public double GetBalance()
   {
      return balance;
   }
}

public class Program
{
   public static void Main()
   {
      BankAccount account = new BankAccount("123456789", 1000.00);

      Console.WriteLine("Initial balance: " + account.GetBalance());

      account.Deposit(500.00);

      Console.WriteLine("After deposit: " + account.GetBalance());

      account.Withdraw(2000.00);

      Console.WriteLine("After withdrawal: " + account.GetBalance());
   }
}
```

2. Abstraction – Focuses on essential properties and behavior of an object while hiding non-essential details.

Abstraction involves creating abstract classes or interfaces that define the essential properties and behavior of a group of related objects, without specifying their implementation details.

```
using System;
public abstract class Vehicle
{
   public int MaxSpeed { get; set; }
   public int NumSeats { get; set; }
   public string FuelType { get; set; }

   public abstract void Start();
   public abstract void Stop();
}

public class Car : Vehicle
{
   public override void Start()
   {
      Console.WriteLine("Starting the car");
   }

   public override void Stop()
   {
      Console.WriteLine("Stopping the car");
   }
}

public class Motorcycle : Vehicle
{
   public override void Start()
   {
      Console.WriteLine("Starting the motorcycle");
   }

   public override void Stop()
   {
      Console.WriteLine("Stopping the motorcycle");
   }
}
public class Program{
   public static void Main(){
      Vehicle car = new Car();
      car.MaxSpeed = 200;
      car.NumSeats = 5;
      car.FuelType = "Petrol";
      car.Start();
      car.Stop();

      Vehicle motorcycle = new Motorcycle();
      motorcycle.MaxSpeed = 150;
      motorcycle.NumSeats = 2;
      motorcycle.FuelType = "Gasoline";
      motorcycle.Start();
      motorcycle.Stop();
            }
}
```

3. Polymorphism - The ability of a class to take on multiple forms, and behave differently based on the context in which it is used. It allows objects of different classes to be treated as if they were the same type of object

2 types. A) compile time, B) run-time polymorphism

Compile time (Method Overloading):

```
public class Calculator {
  public int Add(int x, int y) {
    return x + y;
  }
  public float Add(float x, float y) {
    return x + y;
  }
}
```

Run time (Method Overriding):

```
using System;
class vehicle{
public virtual void run(){
Console.WriteLine("vehicle is running!!");
      }
}
 class bike : vehicle{
 public override void run(){
 Console.WriteLine("bike is runnning!!");
     }
 }
class Program{
 static void Main(string []args){
bike b = new bike();
 b.run();
   }
 }
```

4. Inheritance - The ability of a class to inherit attributes and behaviors from its parent class.

Multiple inheritance is not allowed. We can achieve it using Interface.

Access Specifier –

1. **Public – can be accessed from outside the class**

```
public class MyClass {
  public int MyPublicField;
  public void MyPublicMethod() {
    // ...
  }
}

MyClass obj = new MyClass();
obj.MyPublicField = 10;
obj.MyPublicMethod();
```

2. **Private – cannot be accessed outside the declared class**

```
public class MyClass {
  private int myPrivateField;
  private void MyPrivateMethod() {
    // ...
  }
}

MyClass obj = new MyClass();
obj.myPrivateField = 10; // Compile-time error: 'MyClass.myPrivateField' is inaccessible due to its protection level
obj.MyPrivateMethod();   // Compile-time error: 'MyClass.MyPrivateMethod()' is inaccessible due to its protection level
```

3. **Protected – can be accessed in the declared class and in derived class**

```
public class MyBaseClass {
  protected int MyProtectedField;
  protected void MyProtectedMethod() {
    // ...
  }
}

public class MyDerivedClass : MyBaseClass {
  public void MyDerivedMethod() {
    MyProtectedField = 10;
    MyProtectedMethod();
  }
}

MyDerivedClass obj = new MyDerivedClass();
obj.MyProtectedField = 10;  // Compile-time error: 'MyBaseClass.MyProtectedField' is inaccessible due to its protection level
obj.MyProtectedMethod();    // Compile-time error: 'MyBaseClass.MyProtectedMethod()' is inaccessible due to its protection level
obj.MyDerivedMethod();      // This is valid, since MyDerivedMethod can access MyProtectedField and MyProtectedMethod through inheritance
```

Question – When to use Abstract Class?

Abstract classes are useful when we want to create a hierarchy of related classes, where the base class provides some common behavior, but the derived classes can have their own unique behavior as well.

The Singleton design pattern is a creational pattern that ensures that a class has only one instance and provides a global point of access to that instance. Here are some advantages and disadvantages of using the Singleton pattern:

Advantages:

1. Global access: The Singleton pattern provides a global point of access to the instance, allowing objects throughout the system to access the same instance of the class.
2. Single instance: The Singleton pattern ensures that only one instance of the class is created, which can be useful for managing shared resources, such as database connections or thread pools.
3. Lazy initialization: The Singleton pattern allows for lazy initialization, meaning that the instance is only created when it is first needed. This can help to improve performance and reduce memory usage.
4. Thread safety: The Singleton pattern can be implemented in a thread-safe manner, ensuring that multiple threads cannot create multiple instances of the class.

Disadvantages:

1. Global state: The Singleton pattern can introduce global state into a system, which can make it more difficult to reason about and test.
2. Difficulty in testing: Because the Singleton pattern introduces a global state, it can be difficult to test objects that depend on the Singleton instance.
3. Can be overused: The Singleton pattern can be overused, leading to a system that is difficult to maintain and modify.
4. Coupling: The Singleton pattern can introduce tight coupling between objects that depend on the Singleton instance, making it difficult to change the implementation of the Singleton without affecting other parts of the system.

In summary, the Singleton pattern can provide some useful benefits, such as global access to a shared resource and lazy initialization, but it also has some drawbacks, such as introducing global state and tight coupling between objects. When deciding whether to use the Singleton pattern, it's important to consider the specific requirements of your system and weigh the pros and cons carefully.

what is tight coupling?

Tight coupling refers to a situation in software engineering where two or more components or classes in a system are strongly dependent on each other. In other words, tight coupling means that changes to one component or class will have a significant impact on the other components or classes that it is coupled with.

Data Structure

Algorithm (sorting)	Time Complexity
Bubble sort	O(n2)
Insertion sort – stable, inplace	Best case O(n) Worst case O(n2)
Selection sort – not stable, inplace	Best case O(n2) Worst case O(n2)
Quick sort	Best, avg case O(nlogn) Worst case O(n2)
Merge sort	All case O(nlogn)
Heap sort – inplace, unstable (order of same value change after sort)	All case O(nlogn)

Heap tree – Insertion (key by key method) – TC: worst case - O(nlogn) , best case O(1) , Heapify – O(n)
Deletion – TC: worst case - O(nlogn), best case – O(1)

Hashing	TC (Best Case)	TC (Collision) Worst case
Insertion	O(1)	O(n)
Deletion	O(1)	O(n)
Search	O(1)	O(n)

Algorithm	TC
BFS (Queue)	O(V+E)
DFS (Stack)	O(V+E)

Dijkstra	O(E logV)

DFS:

```
void dfs(int u, vector<int> adj[], vector<bool> &visited)
{
   visited[u] = true;
   cout << u << " ";

   for(int v: adj[u])
   {
      if(!visited[v])
      {
         dfs(v, adj, visited);
      }
   }
}

int main()
{
   int n, m; // n = number of vertices, m = number of edges
   cin >> n >> m;
   vector<int> adj[n+1]; // adjacency list
   vector<bool> visited(n+1, false); // visited array

   int u, v;
   for(int i = 0; i < m; i++)
   {
      cin >> u >> v;
      adj[u].push_back(v);
      adj[v].push_back(u); // for undirected graphs
   }
   int start_vertex = 1; // starting vertex for DFS
   dfs(start_vertex, adj, visited); // DFS traversal starting from start_vertex
   return 0;
}
```

BFS:

```
void bfs(int start_vertex, vector<int> adj[], vector<bool> &visited)
{
   queue<int> q; // queue for BFS traversal
   visited[start_vertex] = true;
   q.push(start_vertex); // enqueue the starting vertex
   while(!q.empty())
   {
      int u = q.front();
      cout << u << " ";
      q.pop();
      for(int v : adj[u])
      {
         if(!visited[v])
         {
            visited[v] = true;
            q.push(v); // enqueue the unvisited adjacent vertices
         }
      }
   }
}
```

Data Structure	Insertion	Deletion	Search
Stack(LIFO)	O(1)	O(1)	O(n)
Queue(FIFO)	O(1)	O(1)	O(n)

Implement a stack:

```
#include<iostream>
using namespace std;
const int MAX_SIZE = 100;                    // maximum size of stack
class Stack {
private:
   int arr[MAX_SIZE];                        // array to store stack elements
   int top;                                  // index of top element
public:
   Stack() {                                 // constructor to initialize stack
      top = -1;                              // set top to -1 to indicate empty stack
   }
   bool is_empty() {                         // function to check if stack is empty
      return (top == -1);
   }
   bool is_full() {                          // function to check if stack is full
      return (top == MAX_SIZE - 1);
   }
   void push(int val) {                      // function to add element to top of stack
      if (is_full()) {
         cout << "Error: Stack is full\n";
         return;
      }
      top++;                                 // increment top index
      arr[top] = val;                        // add element to top of stack
   }
   void pop() {                              // function to remove element from top of stack
      if (is_empty()) {
         cout << "Error: Stack is empty\n";
         return;
      }
      top--; // decrement top index
   }
   int peek() { // function to return top element of stack
      if (is_empty()) {
         cout << "Error: Stack is empty\n";
         return -1;
      }
      return arr[top];                       // return top element
   }
};
int main() {
   Stack s;
   s.push(10);
   s.push(20);
   s.push(30);
   cout << s.peek() << endl;                 // prints 30
   s.pop();
   cout << s.peek() << endl;                 // prints 20
   s.pop();
   cout << s.peek() << endl;                 // prints 10
   s.pop();
   s.pop(); // prints "Error: Stack is empty"
}
```

Implement a Queue:

```
#include<iostream>
using namespace std;
const int MAX_SIZE = 100;                                   // maximum size of queue

class Queue {
private:
   int arr[MAX_SIZE];                                       // array to store queue elements
   int front, rear;                                          // indices of front and rear elements
public:
   Queue() {                                                // constructor to initialize queue
      front = -1;                                           // set front to -1 to indicate empty queue
      rear = -1;                                            // set rear to -1 to indicate empty queue
   }
   bool is_empty() {                                        // function to check if queue is empty
      return (front == -1 && rear == -1);
   }
   bool is_full() {                                         // function to check if queue is full
      return (rear == MAX_SIZE - 1);
   }
   void enqueue(int val) {                                  // function to add element to rear of queue
      if (is_full()) {
         cout << "Error: Queue is full\n";
         return;
      }
      if (is_empty()) {                                     // if queue is empty, set front to 0
         front = 0;
      }
      rear++;                                                        // increment rear index
      arr[rear] = val;                                      // add element to rear of queue
   }
   void dequeue() {                                         // function to remove element from front of queue
      if (is_empty()) {
         cout << "Error: Queue is empty\n";
         return;
      }
      if (front == rear) {                                  // if queue has only one element, set front and rear to -1
         front = -1;
         rear = -1;
      } else {
         front++;                                           // increment front index
      }
   }
   int peek() {                                     // function to return front element of queue
      if (is_empty()) {
         cout << "Error: Queue is empty\n";
         return -1;
      }
      return arr[front];                                     // return front element
   }
};

int main() {
   Queue q;
   q.enqueue(10);
   q.enqueue(20);
   q.enqueue(30);
   cout << q.peek() << endl;            // prints 10
   q.dequeue();
   cout << q.peek() << endl;            // prints 20
   q.dequeue();
   cout << q.peek() << endl;            // prints 30
   q.dequeue();
   q.dequeue();                         // prints "Error: Queue is empty"
   return 0;
}
```

Tree	Insertion	Deletion	Search
BST	O(logn)	O(logn)	O(logn)

Diameter of a tree:

```
class Solution {
public:
    int ans=0;
    int helper(TreeNode* node){
        if(node==NULL) return 0;

        int lh = helper(node->left);
        int rh = helper(node->right);
        ans = max(ans, lh+rh);
        return 1+max(lh,rh);
    }


    int diameterOfBinaryTree(TreeNode* root) {
        helper(root);
        return ans;
    }
};
```

TC: O(n)

SQL:

Primary key: A primary key is a field or combination of fields that uniquely identifies each record in a table. It must have a unique value for each record and cannot be null.

Foreign key: A foreign key is a field in one table that refers to the primary key of another table. It establishes a relationship between two tables and is used to enforce referential integrity constraints.

Candidate key: A candidate key is a field or combination of fields that could be used as a primary key. It must have a unique value for each record and cannot be null. However, unlike a primary key, a candidate key is not necessarily chosen as the primary key for a table. A table may have multiple candidate keys, and the primary key is chosen based on which key is most appropriate for the table's needs.

Normalization: Normalization is the process of organizing data in a database so that it meets certain requirements in terms of structure, integrity, and efficiency.

1 NF:
- No multiple records can be there in a single row.
- Each column must be unique

2 NF: (eliminate partial dependency)
- Must be in 1 NF
- All non-key attributes (i.e., attributes that are not part of the primary key) are functionally dependent on the entire primary key.

3 NF:
- All non-key attributes are functionally dependent only on the primary key.
- Must be in 2 NF and no transitive dependency

In SQL, a join is a way to combine data from two or more tables based on a related column between them. Joining tables allows you to retrieve data from multiple tables as if they were a single table.

There are several types of joins in SQL, including:

Inner Join: Returns only the rows where there is a match in both tables based on a specified column.

```
SELECT column1, column2, ...
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

Left Join: Returns all rows from the left table and the matched rows from the right table based on a specified column. If there is no match in the right table, the result will contain NULL values.

```
SELECT column1, column2, ...
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

Right Join: Returns all rows from the right table and the matched rows from the left table based on a specified column. If there is no match in the left table, the result will contain NULL values.

```
SELECT column1, column2, ...
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

Full Outer Join: Returns all rows from both tables and matches them based on a specified column. If there is no match in either table, the result will contain NULL values.

```
SELECT column1, column2, ...
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

Question: Group By and Order By clause of SQL?

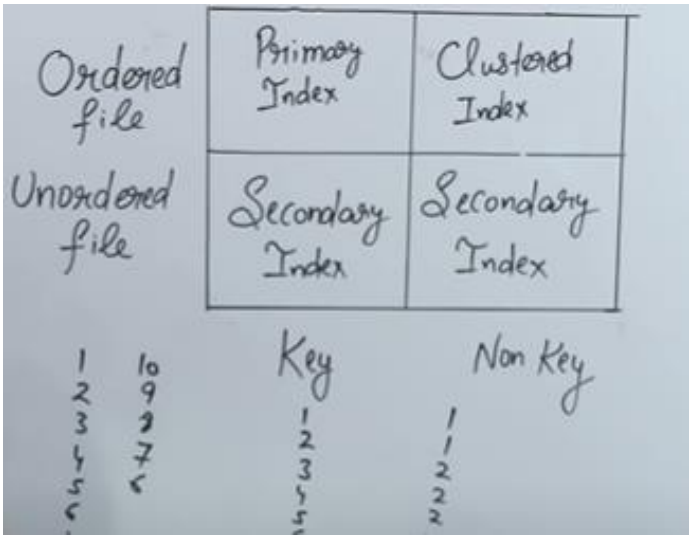
The **GROUP BY** clause is used to group data in a table based on one or more columns. It is commonly used with aggregate functions such as SUM, COUNT, AVG, MIN, and MAX to calculate summary statistics for each group.

The **ORDER BY** clause is used to sort the result set in ascending or descending order based on one or more columns.

 # Query for finding the average salary of the employee of each department in desc order…….

```
SELECT department , AVG(salary) as avg_salary
FROM employee
GROUP BY department
ORDER BY avg_salary in DESC;
```

Indexing in DBMS:



Vue.Js

1) How do you implement two-way data binding in a Vue.js component?

To implement **two-way data binding in a Vue.js** component, you can use the "v-model" directive. This directive binds a value to a form input element, and updates the value whenever the input element's value changes.

Here is an example of how you might use the "v-model" directive in a Vue.js component:

```
<template>
  <div>
   <input v-model="message" type="text">
   <p>{{ message }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: 'Hello world!'
    }
  }
}
</script>
```

2) How to implement computed properties in Vue.js?

Computed properties are properties in a Vue.js component that are calculated based on other properties in the component. They are like methods, but they are cached based on their dependencies, and will only re-evaluate when one of their dependencies changes. This can be more efficient than calling a method multiple times, especially if the method's result is expensive to compute.

Example:

```
<template>
 <div>
   <p>{{ reversedMessage }} </p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: 'Hello world!'
    }
  },
  computed: {
    reversedMessage() {
      return this.message.split('').reverse().join('')
    }
  }
}
</script>
```

Features of Vue.js -

1. **Templates:** Vue.js uses HTML-based templates that allow you to declaratively render the UI based on your data model. The template syntax is designed to be easy to learn and understand, making it simple to build complex UIs without having to write a lot of JavaScript code.
2. **Reactivity:** Vue.js provides a reactive data system that allows you to define data properties that can automatically update the UI when they change. This makes it easy to build dynamic UIs that respond to user input or changes in data over time.
3. **Components:** Vue.js allows you to create reusable components that encapsulate the UI and behavior of a specific part of your application. Components can be easily composed together to build complex UIs, and can be reused across different pages or projects.
4. **Transitions:** Vue.js provides a powerful transition system that allows you to add animation and visual effects to your UI components. The transition system is designed to be easy to use and highly customizable, making it possible to create complex animations with only a few lines of code.
5. **Routing:** Vue.js includes a powerful routing system that allows you to define and navigate between different pages or views within your application. The routing system is designed to be flexible and customizable, making it easy to handle different types of navigation requirements, such as nested routes or dynamic routing based on user input.