1. Concept and Design :-

Market Research: Identify market needs and trends.

Design: Create sketches, CAD models, and prototypes of the car.

Engineering: Detailed engineering of components, including the engine, chassis, and electronics.

2. Prototype Development :-

Building a Prototype: Construct a prototype for testing.

Testing: Perform tests on performance, safety, and durability.

Refinement: Make necessary adjustments based on test results.

3. Production Planning :-

Supply Chain Setup: Establish contracts with suppliers for parts and materials.

Manufacturing Setup: Prepare assembly lines, machinery, and robotics.

Logistics Planning: Plan for the distribution and delivery of materials.

4. Manufacturing :-

Stamping: Sheet metal is cut and stamped into body parts.

Welding: Body parts are welded together to form the car's frame.

Painting: The car's body is painted and finished.

Assembly: Installation of engine, drivetrain, electronics, interior, and other components.

5. Quality Control :-

Inspection: Inspect each vehicle for defects and ensure it meets quality standards.

Testing: Perform road tests and other evaluations to ensure the car functions as expected.

6. Distribution :-

Shipping: Transport cars to dealerships.

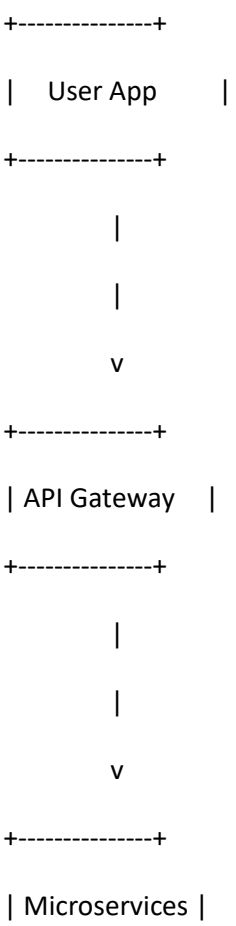
Sales: Cars are made available for sale to customers.

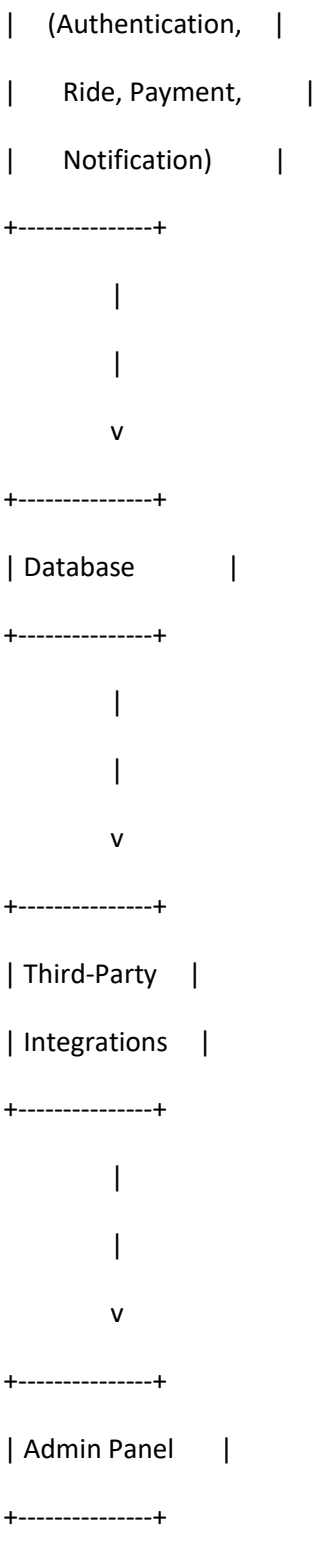
7. Post-Sale Support :-

Maintenance and Repairs: Provide after-sales services, including warranty repairs and regular maintenance.

Customer Feedback: Collect feedback to improve future models.

Here's a high-level architecture diagram:





Note that this is a simplified diagram, and a real-world architecture would involve additional components and complexity.

Title: IRCTC Web Application Architecture using the MVC Pattern

Diagram Structure:

- **IRCTC Browser (Top Element):**
    - Positioned at the top center.
    - Acts as the client interface where users search for trains, check availability, and book tickets.
- **Controller (Left Box):**
    - Positioned below the IRCTC browser and to the left.
    - **Content:**
        - Processing HTTP requests from the IRCTC browser
        - Application-specific logic, such as handling search queries, login authentication, and session management.
        - Data validation to ensure the correctness of user input, such as dates, train numbers, and passenger details.
    - **Arrow directions:**
        - **From IRCTC Browser to Controller:** Represents an HTTP request sent from the browser, such as a ticket search or booking request.
        - **From Controller to Model:** Represents the update request for data processing, such as querying the database for train availability or confirming a booking.
        - **From Controller to View:** Represents form generation and handling user events, like displaying search results or booking confirmations.
- **Model (Bottom Box):**
    - Positioned directly below the IRCTC browser.
    - **Content:**
        - Business logic, such as rules for seat allocation, fare calculations, and cancellation policies.
        - Database access for retrieving train schedules, seat availability, user

accounts, booking history, etc.

- **Arrow directions:**
    - **From Model to Controller:** Represents change notification after data is processed, like updating the booking status.
    - **From Model to View:** Represents refresh request, such as sending updated data (like available seats) to the view.

- **View (Right Box):**
    - Positioned below the IRCTC browser and to the right.
    - **Content:**
        - Dynamic page generation for displaying search results, booking details, and payment confirmations.
        - Forms management, such as handling input forms for passenger details, payment methods, and ticket confirmation.
    - **Arrow directions:**
        - **From View to IRCTC Browser:** Represents the response with dynamically generated pages, such as the ticket booking confirmation page or search results.
        - **From View to Model:** Represents notification for any data refresh requirements, like updating seat availability after a booking

1. Implement a custom dynamic array class that supports basic operations like insertion, deletion, resizing, and clearing.

```
#include <iostream>

#include <stdexcept>

using namespace std;

class DynamicArray {

private:

    int* arr;

    int capacity;
```

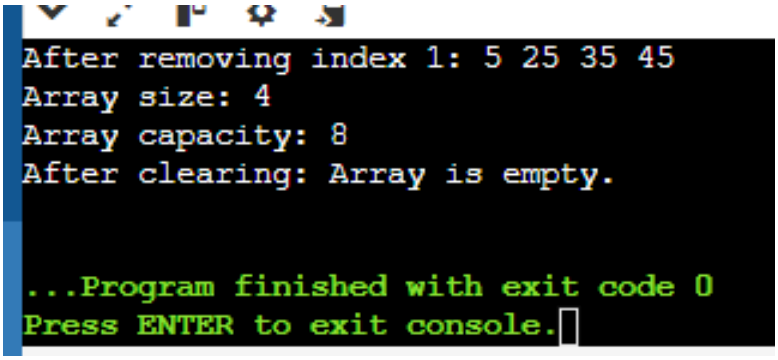
```
    int size;
    void resizeIfNeeded(int new_size) {
        if (new_size > capacity) {
            resize(capacity * 2);
        } else if (new_size <= capacity / 4 && capacity > 4) {
            resize(capacity / 2);          }
    }
    void resize(int new_capacity) {
        int* new_arr = new int[new_capacity];
        for (int i = 0; i < size; ++i) {
            new_arr[i] = arr[i];          }
        delete[] arr;
        arr = new_arr;
        capacity = new_capacity;        }
public:
    DynamicArray(int initial_capacity = 4)
        : capacity(initial_capacity), size(0) {
        if (initial_capacity <= 0) {
            throw invalid_argument("Capacity must be greater than zero.");
        }
        arr = new int[capacity];
    }
    ~DynamicArray() {
        delete[] arr;
    }
    void insert(int value) {
```

```
        resizeIfNeeded(size + 1);
        arr[size++] = value;
}
void remove(int index) {
        if (index < 0 || index >= size) {
                throw out_of_range("Index out of bounds.");
        }
        for (int i = index; i < size - 1; ++i) {
                arr[i] = arr[i + 1];
        }
        --size;
        resizeIfNeeded(size);
}
int get(int index) const {
        if (index < 0 || index >= size) {
                throw out_of_range("Index out of bounds.");
        }
        return arr[index];
}
void clear() {
        delete[] arr;
        size = 0;
        capacity = 4;
        arr = new int[capacity];
}
int getSize() const {
```

```
        return size;
    }
    int getCapacity() const {
        return capacity;
    }
    bool isEmpty() const {
        return size == 0;
    }
    void print() const {
        for (int i = 0; i < size; ++i) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};
int main() {
    DynamicArray da;
    da.insert(5);
    da.insert(15);
    da.insert(25);
    da.insert(35);
    da.insert(45);
    cout << "Array contents: ";
    da.print();
    da.remove(1);
    cout << "After removing index 1: ";
```

```
    da.print();
    cout << "Array size: " << da.getSize() << endl;
    cout << "Array capacity: " << da.getCapacity() << endl;
    da.clear();
    cout << "After clearing: " << (da.isEmpty() ? "Array is empty." : "Array is not empty.") << endl;
    return 0;
}
```

OUTPUT:



2. Create a template-based stack class supporting push, pop, and peek operations. Implement it for different data types like int, float, and std::string.

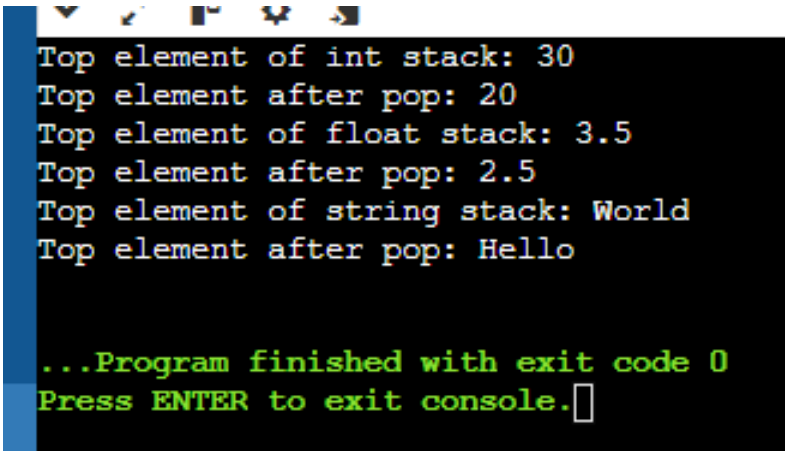
```
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;
template <typename T>
class Stack {
private:
    T* arr;
    int top;
    int capacity;
```

```
    void resize(int new_capacity) {
        T* new_arr = new T[new_capacity];
        for (int i = 0; i < top; ++i) {
            new_arr[i] = arr[i];    }
        delete[] arr;
        arr = new_arr;
        capacity = new_capacity;
    }
public:
    Stack(int initial_capacity = 4)
        : top(0), capacity(initial_capacity) {
        if (initial_capacity <= 0) {
            throw invalid_argument("Capacity must be greater than zero.");          }
        arr = new T[capacity];
    }
    ~Stack() {
        delete[] arr;       }
    void push(const T& value) {
        if (top == capacity) {
            resize(capacity * 2);       }
        arr[top++] = value;
    }
    void pop() {
        if (isEmpty()) {
            throw out_of_range("Stack is empty.");              }
        --top;
    }
    T peek() const {
```

```
        if (isEmpty()) {
            throw out_of_range("Stack is empty.");      }
        return arr[top - 1];
}
    bool isEmpty() const {
        return top == 0;        }
    int size() const {
        return top;      }
    void clear() {
        top = 0;        }
};
int main() {
    Stack<int> intStack;
    intStack.push(10);
    intStack.push(20);
    intStack.push(30);
    cout << "Top element of int stack: " << intStack.peek() << endl;
    intStack.pop();
    cout << "Top element after pop: " << intStack.peek() << endl;
    Stack<float> floatStack;
    floatStack.push(1.5f);
    floatStack.push(2.5f);
    floatStack.push(3.5f);
    cout << "Top element of float stack: " << floatStack.peek() << endl;
    floatStack.pop();
    cout << "Top element after pop: " << floatStack.peek() << endl;
    Stack<string> stringStack;
    stringStack.push("Hello");
```

```
    stringStack.push("World");
    cout << "Top element of string stack: " << stringStack.peek() << endl;
    stringStack.pop();
    cout << "Top element after pop: " << stringStack.peek() << endl;
    return 0;
}
```

OUTPUT:



3. Write a program that reads from a file and handles various exceptions such as file not found, read errors, and unexpected data formats.

```
#include <iostream>
#include <fstream>
#include <string>
#include <stdexcept>

void readFile(const std::string& filename) {
    std::ifstream file;

    // Attempt to open the file
    file.open(filename);
    if (!file.is_open()) {
```

```
        throw std::runtime_error("File not found: " + filename);
    }

    std::string line;
    while (std::getline(file, line)) {
        try {
            // Simulate unexpected data format check
            if (line.empty()) {
                throw std::runtime_error("Unexpected data format: empty line encountered.");
            }
            // Process the line (for demonstration, just print it)
            std::cout << "Read line: " << line << std::endl;
        } catch (const std::runtime_error& e) {
            std::cerr << "Error processing line: " << e.what() << std::endl;
        }
    }

    // Check for any read errors
    if (file.bad()) {
        throw std::runtime_error("Error reading the file: " + filename);
    }

    file.close();
}

int main() {
    const std::string filename = "example.txt";

    try {
        readFile(filename);
```

```
    } catch (const std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

4. Write a unit test suite for the custom dynamic array class using a testing framework like Google Test or CppUnit.

Step 1: Install Google Test

First, make sure you have Google Test installed on your system. You can either download and build it from the source or use a package manager like apt on Ubuntu.

Step 2: Implement the Custom Dynamic Array Class

```
// DynamicArray.h
#ifndef DYNAMIC_ARRAY_H
#define DYNAMIC_ARRAY_H
#include <stdexcept>
template <typename T>
class DynamicArray {
public:
    DynamicArray(size_t capacity = 10)
        : capacity(capacity), size(0), data(new T[capacity]) {}

    ~DynamicArray() {
        delete[] data;
    }
    void add(T value) {
        if (size == capacity) {
            resize();
        }
```

```
        data[size++] = value;
    }
    T get(size_t index) const {
        if (index >= size) {
            throw std::out_of_range("Index out of range");
        }
        return data[index];
    }
    size_t getSize() const {
        return size;
    }
    size_t getCapacity() const {
        return capacity;
    }
private:
    void resize() {
        capacity *= 2;
        T* newData = new T[capacity];
        for (size_t i = 0; i < size; ++i) {
            newData[i] = data[i];
        }
        delete[] data;
        data = newData;
    }
    size_t capacity;
    size_t size;
    T* data;
};
#endif
```

Step 3: Write the Unit Test Suite Using Google Test

```
// DynamicArrayTest.cpp
#include "DynamicArray.h"
#include <gtest/gtest.h>
TEST(DynamicArrayTest, TestAddAndGet) {
    DynamicArray<int> array;
    array.add(10);
    array.add(20);
    array.add(30);
    EXPECT_EQ(array.getSize(), 3);
    EXPECT_EQ(array.get(0), 10);
    EXPECT_EQ(array.get(1), 20);
    EXPECT_EQ(array.get(2), 30);
}
TEST(DynamicArrayTest, TestOutOfRange) {
    DynamicArray<int> array;
    array.add(10);
    EXPECT_THROW(array.get(1), std::out_of_range);
}
TEST(DynamicArrayTest, TestResize) {
    DynamicArray<int> array(2);
    array.add(10);
    array.add(20);
    EXPECT_EQ(array.getCapacity(), 2);
    array.add(30);
    EXPECT_EQ(array.getCapacity(), 4);
}
TEST(DynamicArrayTest, TestInitialCapacity) {
    DynamicArray<int> array(15);
    EXPECT_EQ(array.getCapacity(), 15);
    EXPECT_EQ(array.getSize(), 0);
```

```
}
int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Step 4: Compile and Run the Tests

To compile and run the tests, you need to link against the Google Test libraries.

```
cmake_minimum_required(VERSION 3.10)
project(DynamicArrayTest)
set(CMAKE_CXX_STANDARD 11)
# Add the dynamic array source files
add_library(DynamicArray DynamicArray.h)
# Find Google Test
find_package(GTest REQUIRED)
include_directories(${GTEST_INCLUDE_DIRS})
# Add the test executable
add_executable(DynamicArrayTest DynamicArrayTest.cpp)
# Link the test executable with Google Test and pthread
target_link_libraries(DynamicArrayTest ${GTEST_LIBRARIES} pthread)
```

To compile:

```
mkdir build
cd build
cmake ..
make
```

Then, run the tests:

```
./DynamicArrayTest
```

Write a unit test suite for the custom dynamic array class using a testing framework like Google Test or CppUnit.