**Data Structures and Algorithms**

**Exercise 1: Inventory Management System**

**Scenario:** You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

import java.util.\*;

class Product {

int productId;

String productName;

int quantity;

double price;

Product(int id, String name, int qty, double price) {

this.productId = id;

this.productName = name;

this.quantity = qty;

this.price = price;

}

public String toString() {

return productId + " - " + productName + " - Qty: " + quantity + " - Price: ₹" + price;

}

}

public class InventorySystem {

static HashMap<Integer, Product> inventory = new HashMap<>();

static Scanner sc = new Scanner(System.in);

public static void main(String[] args) {

while (true) {

System.out.println("\n1. Add\n2. Update\n3. Delete\n4. View\n5. Exit");

int choice = sc.nextInt();

switch (choice) {

case 1 -> addProduct();

case 2 -> updateProduct();

case 3 -> deleteProduct();

case 4 -> viewProducts();

case 5 -> System.exit(0);

}

}

}

static void addProduct() {

System.out.print("Enter ID, Name, Quantity, Price: ");

int id = sc.nextInt();

String name = sc.next();

int qty = sc.nextInt();

double price = sc.nextDouble();

inventory.put(id, new Product(id, name, qty, price));

}

static void updateProduct() {

System.out.print("Enter ID to update: ");

int id = sc.nextInt();

if (inventory.containsKey(id)) {

System.out.print("Enter new Name, Quantity, Price: ");

String name = sc.next();

int qty = sc.nextInt();

double price = sc.nextDouble();

inventory.put(id, new Product(id, name, qty, price));

} else System.out.println("Product not found.");

}

static void deleteProduct() {

System.out.print("Enter ID to delete: ");

int id = sc.nextInt();

inventory.remove(id);

}

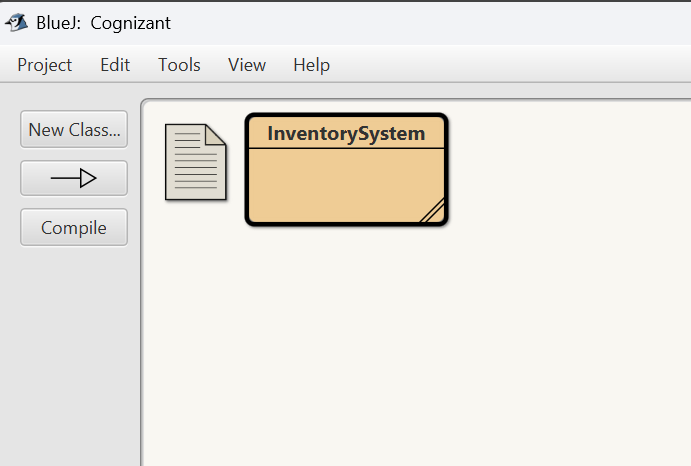
static void viewProducts() {

for (Product p : inventory.values())

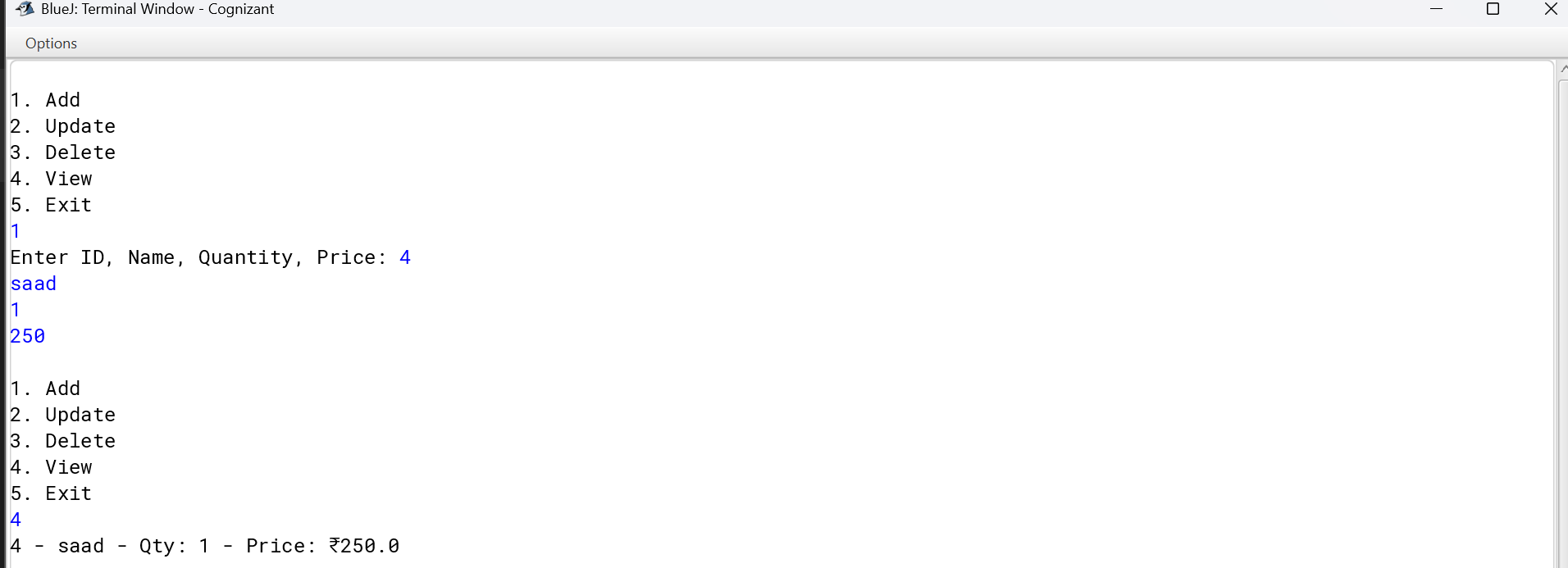
System.out.println(p);

}

}



**Output:**





**Time Complexity (using HashMap):**

* **Add**: O(1)
* **Update**: O(1) (search by key)
* **Delete**: O(1)

**Optimization:**

* Use HashMap<productId, Product> for constant time operations.

**Exercise 2: E-commerce Platform Search Function**

**Scenario**: You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

import java.util.\*;

class Product {

int productId;

String productName;

String category;

Product(int id, String name, String category) {

this.productId = id;

this.productName = name;

this.category = category;

}

public String toString() {

return productId + " - " + productName + " - " + category;

}

}

public class ProductSearch {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Product[] products = {

new Product(1, "Phone", "Electronics"),

new Product(2, "Shoes", "Fashion"),

new Product(3, "Laptop", "Electronics"),

new Product(4, "Book", "Education"),

new Product(5, "Watch", "Fashion")

};

Arrays.sort(products, Comparator.comparing(p -> p.productName));

System.out.print("Enter product name to search: ");

String searchName = sc.next();

System.out.println("Using Linear Search:");

boolean found = false;

for (Product p : products) {

if (p.productName.equalsIgnoreCase(searchName)) {

System.out.println(p);

found = true;

}

}

if (!found) System.out.println("Not Found");

System.out.println("Using Binary Search:");

int low = 0, high = products.length - 1;

found = false;

while (low <= high) {

int mid = (low + high) / 2;

int cmp = products[mid].productName.compareToIgnoreCase(searchName);

if (cmp == 0) {

System.out.println(products[mid]);

found = true;

break;

} else if (cmp < 0) low = mid + 1;

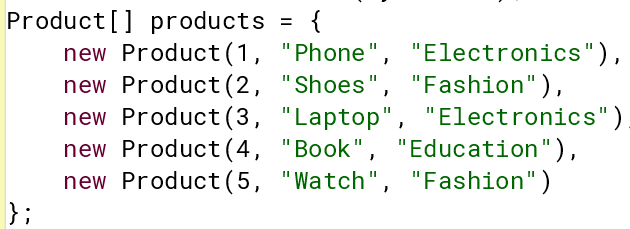
else high = mid - 1;

}

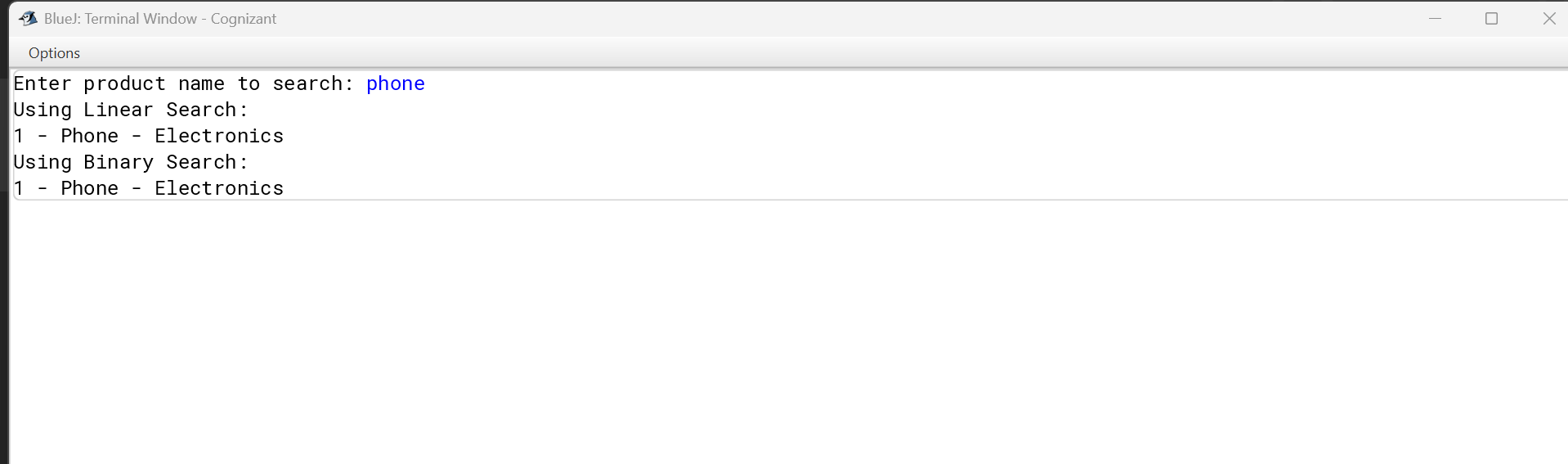
if (!found) System.out.println("Not Found");

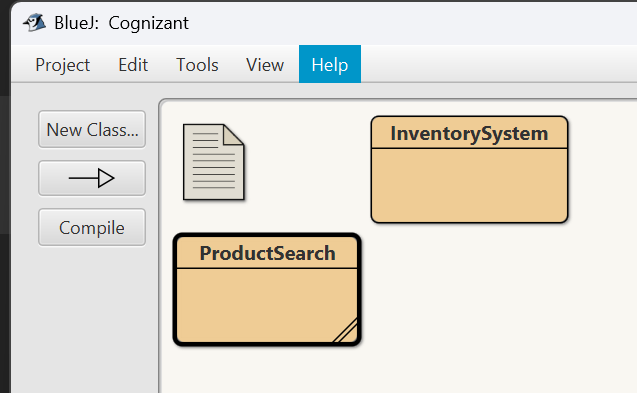
}

}



**Output:**





**Best/Average/Worst Cases:**

* **Linear Search**:
  + Best: O(1) (first element)
  + Avg: O(n)
  + Worst: O(n)
* **Binary Search**:
  + Best: O(1)
  + Avg/Worst: O(log n)

**Time Complexities:**

* **Linear Search**: O(n)
* **Binary Search**: O(log n)

**Best Algorithm:**

* Use **Binary Search** for large, sorted datasets.
* Use **Linear Search** for small/unsorted data or infrequent searches

**Exercise 3: Sorting Customer Orders**

**Scenario:** You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

import java.util.Scanner;

class Order {

int orderId;

String customerName;

double totalPrice;

Order(int id, String name, double price) {

this.orderId = id;

this.customerName = name;

this.totalPrice = price;

}

public String toString() {

return orderId + " - " + customerName + " - ₹" + totalPrice;

}

}

public class OrderSorting {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter number of orders: ");

int n = sc.nextInt();

Order[] orders = new Order[n];

for (int i = 0; i < n; i++) {

System.out.println("Enter Order ID, Customer Name, and Total Price:");

int id = sc.nextInt();

String name = sc.next();

double price = sc.nextDouble();

orders[i] = new Order(id, name, price);

}

System.out.println("\nSorted using Bubble Sort:");

bubbleSort(orders.clone());

System.out.println("\nSorted using Quick Sort:");

quickSort(orders, 0, n - 1);

for (Order o : orders) System.out.println(o);

}

static void bubbleSort(Order[] arr) {

for (int i = 0; i < arr.length - 1; i++)

for (int j = 0; j < arr.length - i - 1; j++)

if (arr[j].totalPrice > arr[j + 1].totalPrice) {

Order temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

for (Order o : arr) System.out.println(o);

}

static void quickSort(Order[] arr, int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

static int partition(Order[] arr, int low, int high) {

double pivot = arr[high].totalPrice;

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j].totalPrice < pivot) {

i++;

Order temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

Order temp = arr[i + 1];

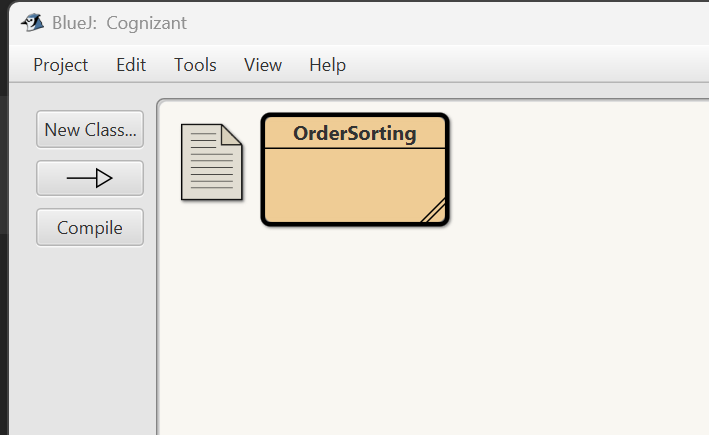
arr[i + 1] = arr[high];

arr[high] = temp;

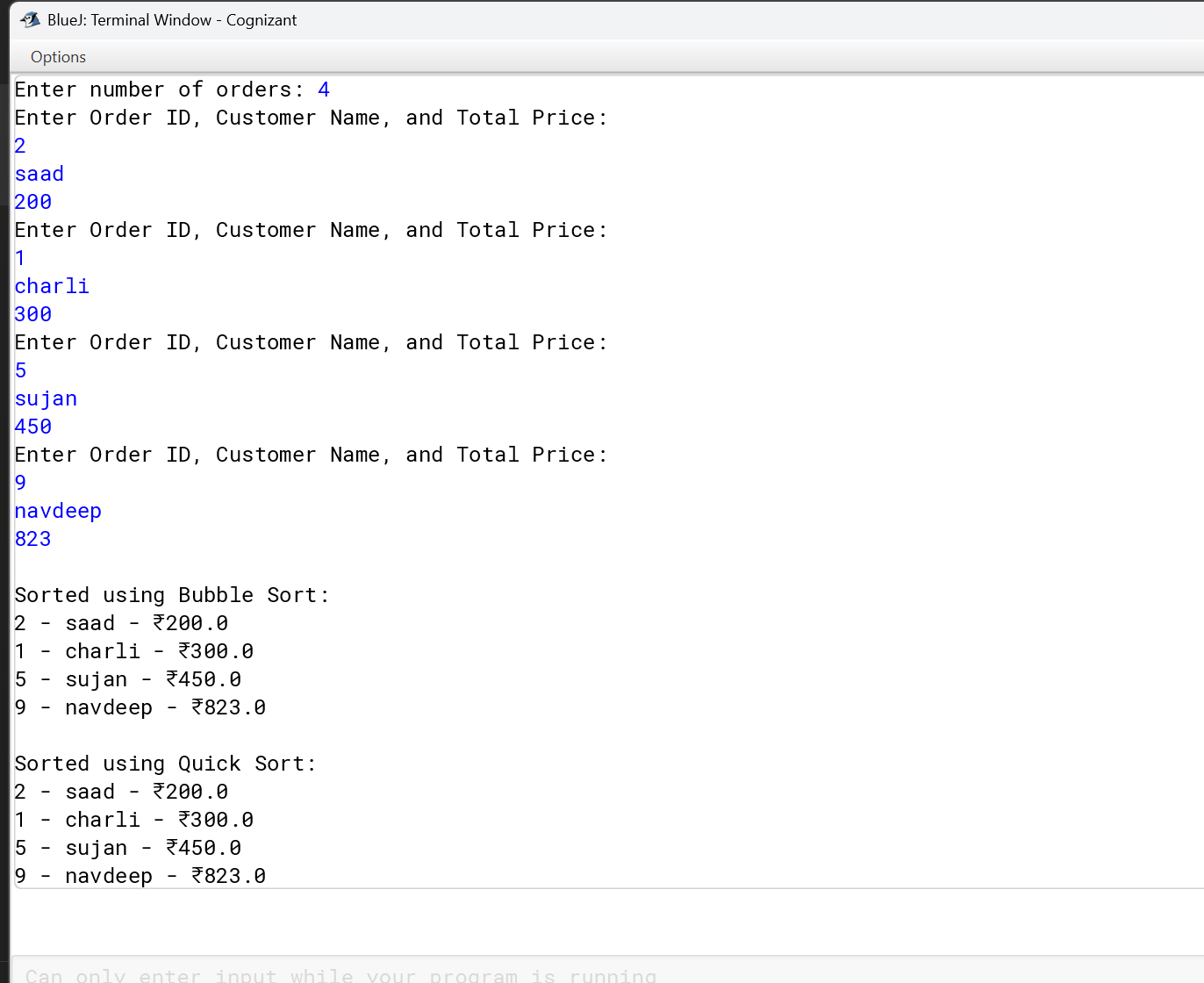
return i + 1;

}

}



**Output:**



**Comparison:**

* **Quick Sort** is generally **preferred** due to its average-case efficiency.
* **Bubble Sort** is only for educational or very small datasets.

**Exercise 4: Employee Management System**

**Scenario:** You are developing an employee management system for a company. Efficiently managing employee records is crucial.

import java.util.Scanner;

class Employee {

int employeeId;

String name;

String position;

double salary;

Employee(int id, String name, String pos, double sal) {

this.employeeId = id;

this.name = name;

this.position = pos;

this.salary = sal;

}

public String toString() {

return employeeId + " - " + name + " - " + position + " - ₹" + salary;

}

}

public class EmployeeSystem {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Employee[] employees = new Employee[100];

int count = 0;

while (true) {

System.out.println("\n1. Add\n2. Search\n3. View All\n4. Delete\n5. Exit");

int choice = sc.nextInt();

switch (choice) {

case 1 -> {

System.out.print("Enter ID, Name, Position, Salary: ");

int id = sc.nextInt();

String name = sc.next();

String pos = sc.next();

double sal = sc.nextDouble();

employees[count++] = new Employee(id, name, pos, sal);

}

case 2 -> {

System.out.print("Enter ID to search: ");

int id = sc.nextInt();

boolean found = false;

for (int i = 0; i < count; i++) {

if (employees[i].employeeId == id) {

System.out.println(employees[i]);

found = true;

break;

}

}

if (!found) System.out.println("Employee not found");

}

case 3 -> {

for (int i = 0; i < count; i++)

System.out.println(employees[i]);

}

case 4 -> {

System.out.print("Enter ID to delete: ");

int id = sc.nextInt();

for (int i = 0; i < count; i++) {

if (employees[i].employeeId == id) {

for (int j = i; j < count - 1; j++)

employees[j] = employees[j + 1];

count--;

System.out.println("Deleted.");

break;

}

}

}

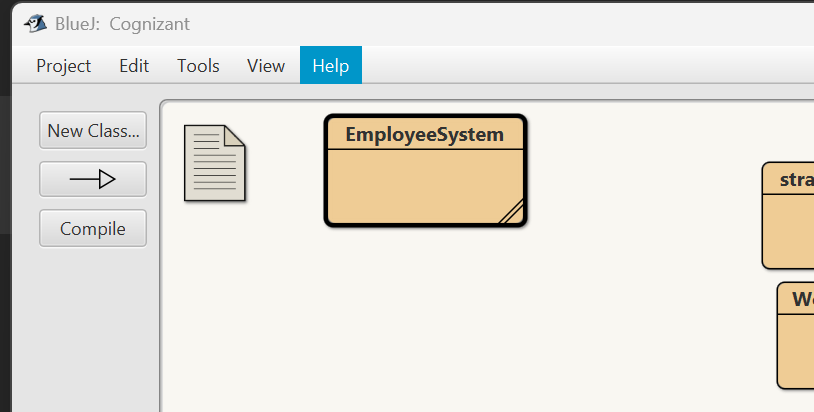
case 5 -> System.exit(0);

}

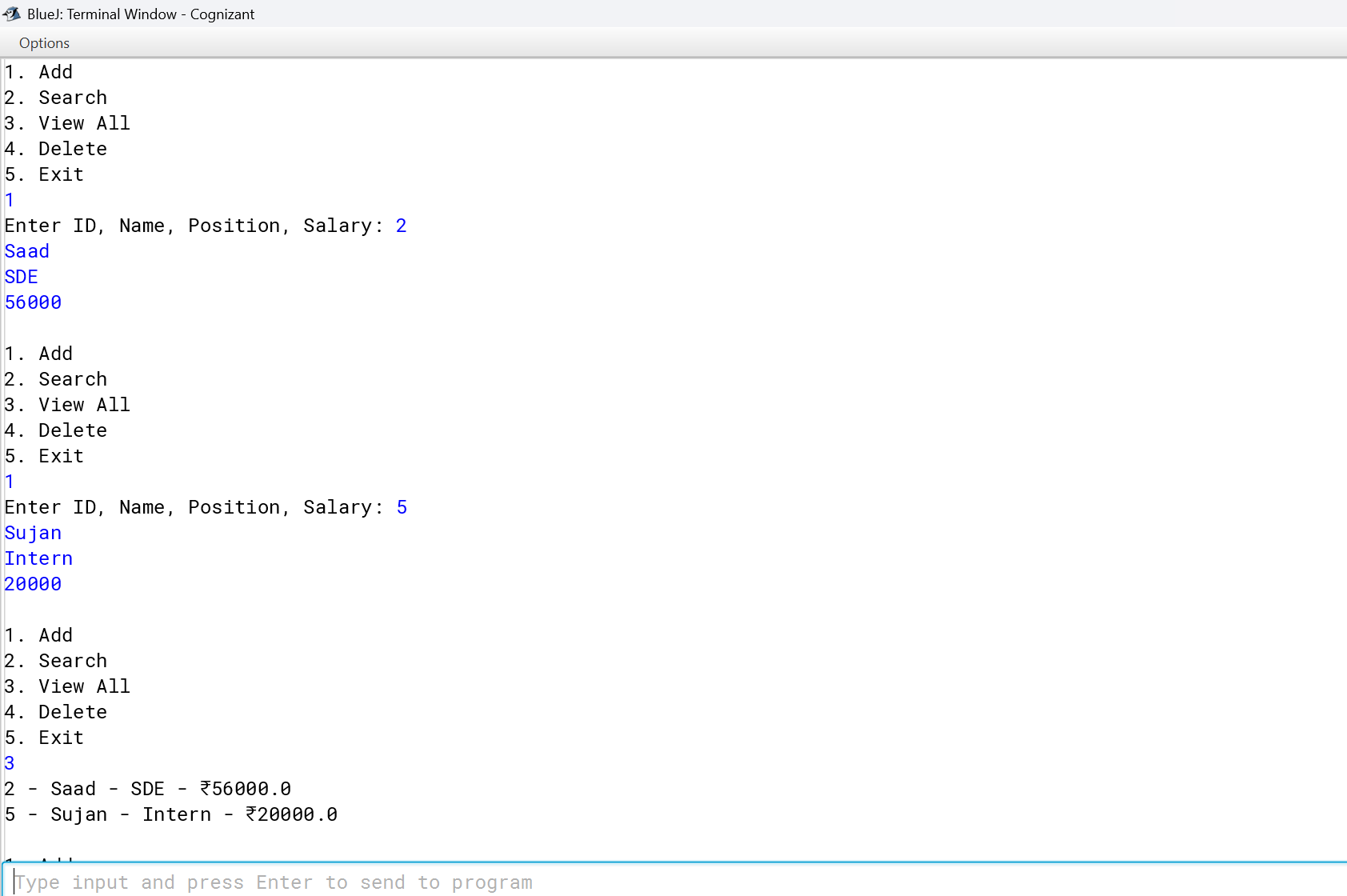
}

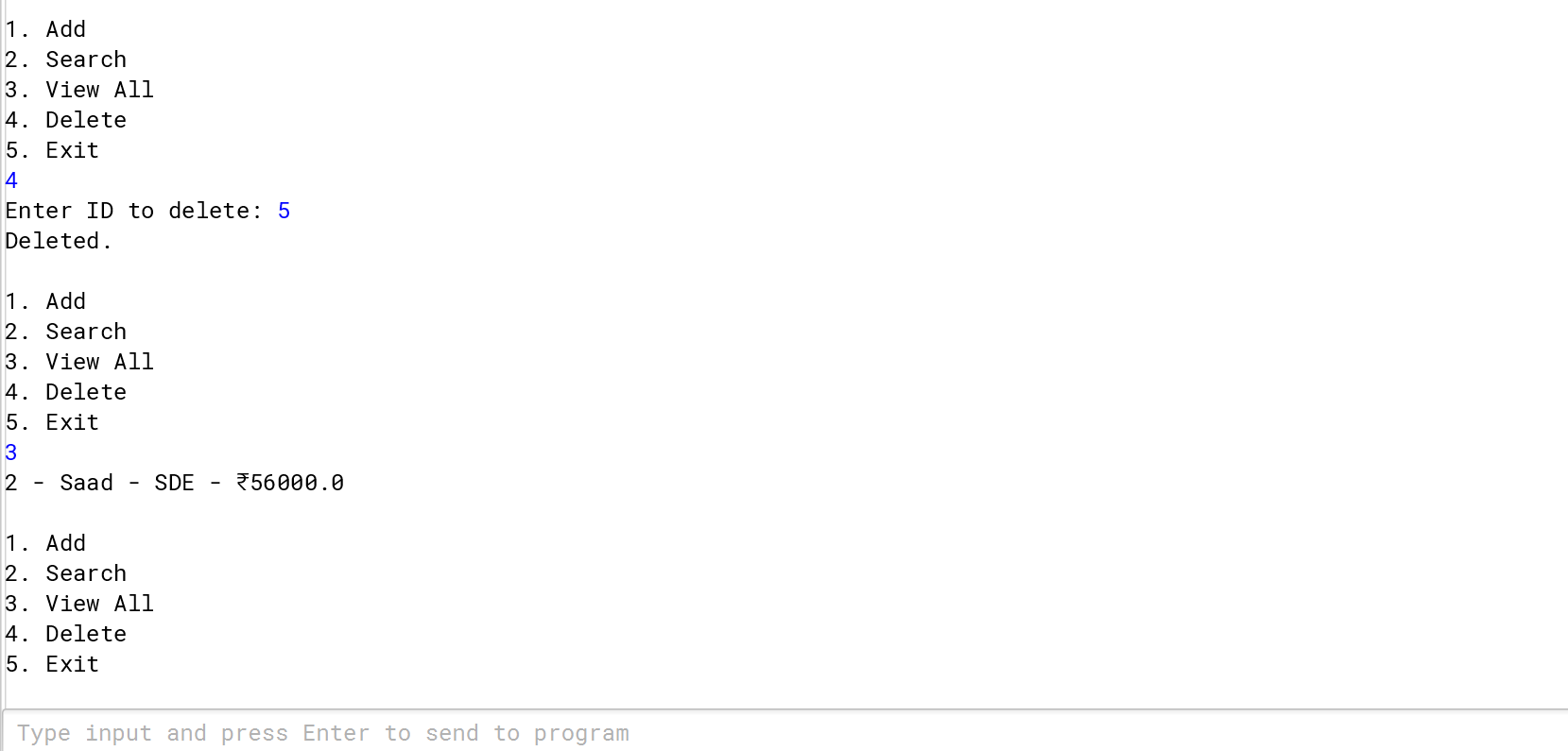
}

}



**Output:**





**Time Complexities:**

|  |  |
| --- | --- |
| Add | O(1) , O(n) (if resizing) |
| Search | O(n) (linear search) |
| Traverse | O(n) |
| Delete | O(n) (shifting required) |
|  |  |

**Limitations:**

* Fixed size unless resized.
* Costly deletions/insertions.
* Best when the number of elements is known and mostly static.

**Exercise 5: Task Management System**

**Scenario:** You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

import java.util.Scanner;

class Task {

int taskId;

String taskName;

String status;

Task next;

Task(int id, String name, String status) {

this.taskId = id;

this.taskName = name;

this.status = status;

this.next = null;

}

public String toString() {

return taskId + " - " + taskName + " - " + status;

}

}

public class TaskManager {

static Task head = null;

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

while (true) {

System.out.println("\n1. Add\n2. Search\n3. View\n4. Delete\n5. Exit");

int ch = sc.nextInt();

switch (ch) {

case 1 -> {

System.out.print("Enter ID, Name, Status: ");

int id = sc.nextInt();

String name = sc.next();

String status = sc.next();

Task newTask = new Task(id, name, status);

if (head == null) head = newTask;

else {

Task temp = head;

while (temp.next != null)

temp = temp.next;

temp.next = newTask;

}

}

case 2 -> {

System.out.print("Enter ID to search: ");

int id = sc.nextInt();

Task temp = head;

boolean found = false;

while (temp != null) {

if (temp.taskId == id) {

System.out.println(temp);

found = true;

break;

}

temp = temp.next;

}

if (!found) System.out.println("Task not found.");

}

case 3 -> {

Task temp = head;

while (temp != null) {

System.out.println(temp);

temp = temp.next;

}

}

case 4 -> {

System.out.print("Enter ID to delete: ");

int id = sc.nextInt();

if (head == null) return;

if (head.taskId == id) {

head = head.next;

System.out.println("Deleted.");

continue;

}

Task prev = head;

Task curr = head.next;

while (curr != null && curr.taskId != id) {

prev = curr;

curr = curr.next;

}

if (curr != null) {

prev.next = curr.next;

System.out.println("Deleted.");

} else {

System.out.println("Task not found.");

}

}

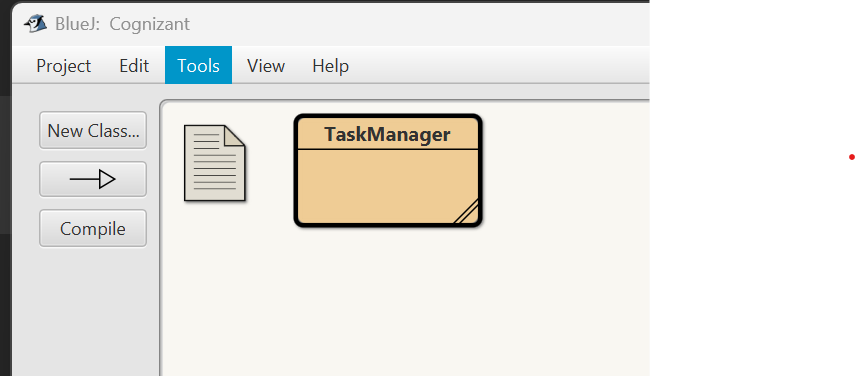
case 5 -> System.exit(0);

}

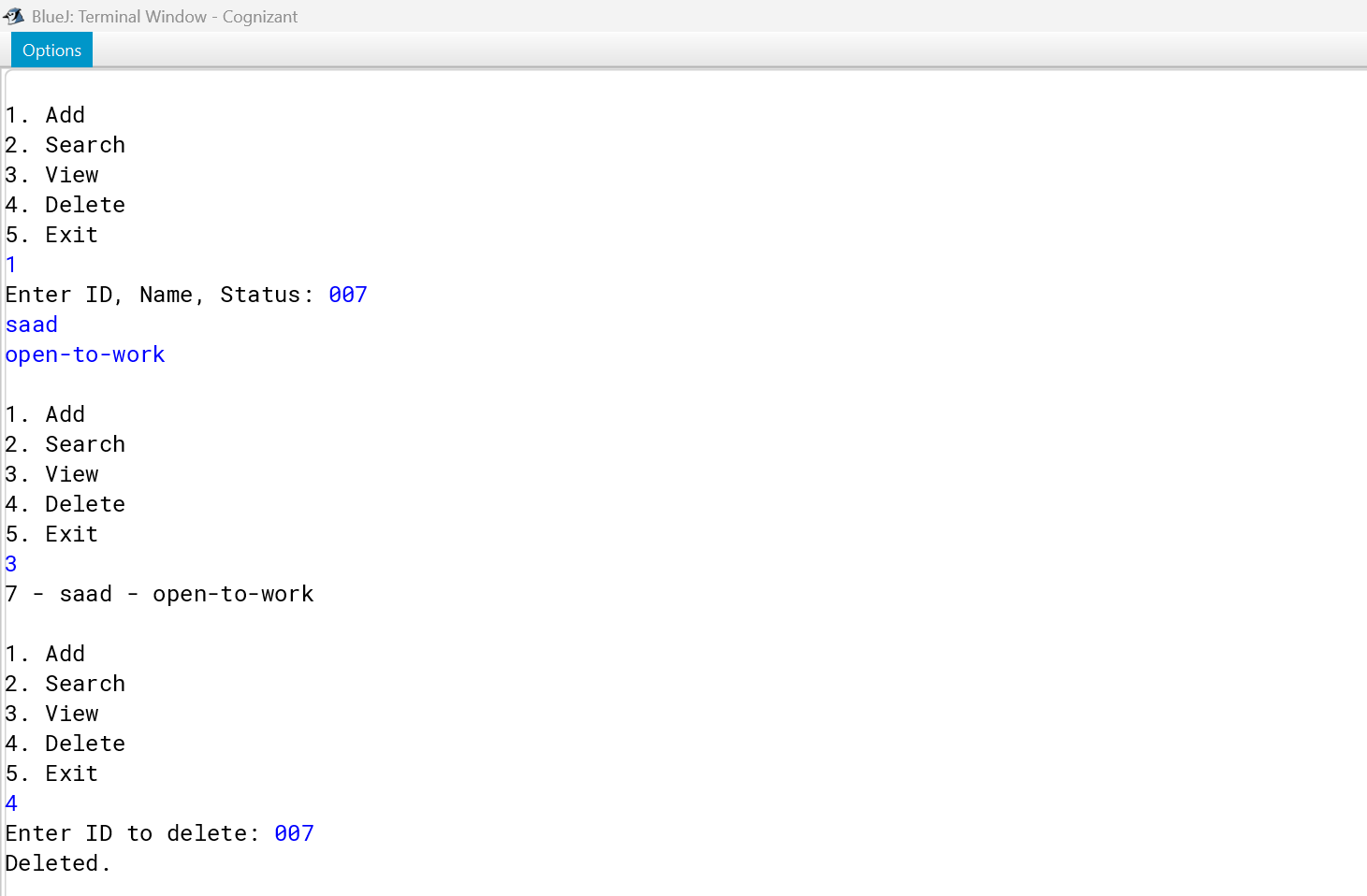
}

}

}



**Output:**



| **Operation** | **Time Complexity** |
| --- | --- |
| Add | O(1) (at head) or O(n) (at tail ) |
| Search | O(n) |
| Traverse | O(n) |
| Delete | O(n) (find + adjust links) |
|  |  |

**Advantages over Arrays:**

* Dynamic size.
* Efficient insertions/deletions.
* No need to shift elements like arrays

**Exercise 6: Library Management System**

**Scenario:** You are developing a library management system where users can search for books by title or author.

import java.util.\*;

class Book {

int bookId;

String title;

String author;

Book(int id, String title, String author) {

this.bookId = id;

this.title = title;

this.author = author;

}

public String toString() {

return bookId + " - " + title + " by " + author;

}

}

public class LibrarySystem {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter number of books: ");

int n = sc.nextInt();

Book[] books = new Book[n];

for (int i = 0; i < n; i++) {

System.out.print("Enter ID, Title, Author: ");

int id = sc.nextInt();

String title = sc.next();

String author = sc.next();

books[i] = new Book(id, title, author);

}

Arrays.sort(books, Comparator.comparing(b -> b.title));

System.out.print("Enter title to search: ");

String searchTitle = sc.next();

System.out.println("Linear Search:");

for (Book b : books) {

if (b.title.equalsIgnoreCase(searchTitle))

System.out.println(b);

}

System.out.println("Binary Search:");

int low = 0, high = books.length - 1;

boolean found = false;

while (low <= high) {

int mid = (low + high) / 2;

int cmp = books[mid].title.compareToIgnoreCase(searchTitle);

if (cmp == 0) {

System.out.println(books[mid]);

found = true;

break;

} else if (cmp < 0) low = mid + 1;

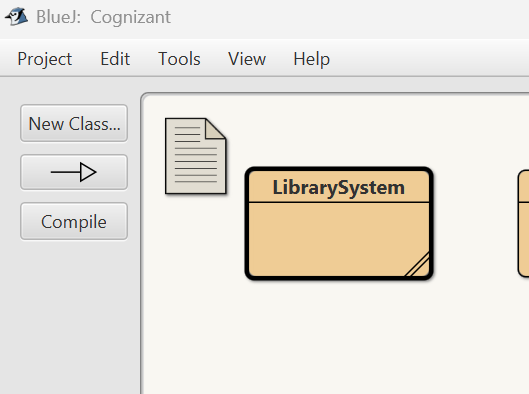
else high = mid - 1;

}

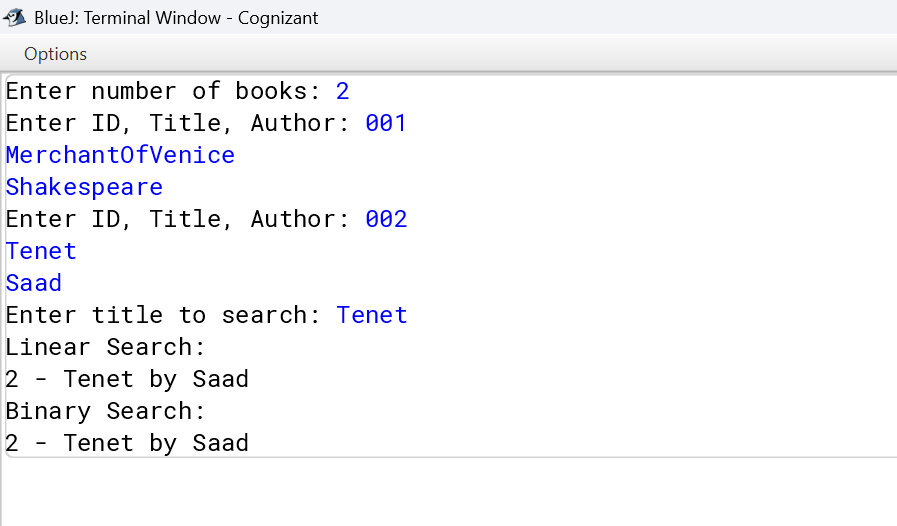
if (!found) System.out.println("Book not found.");

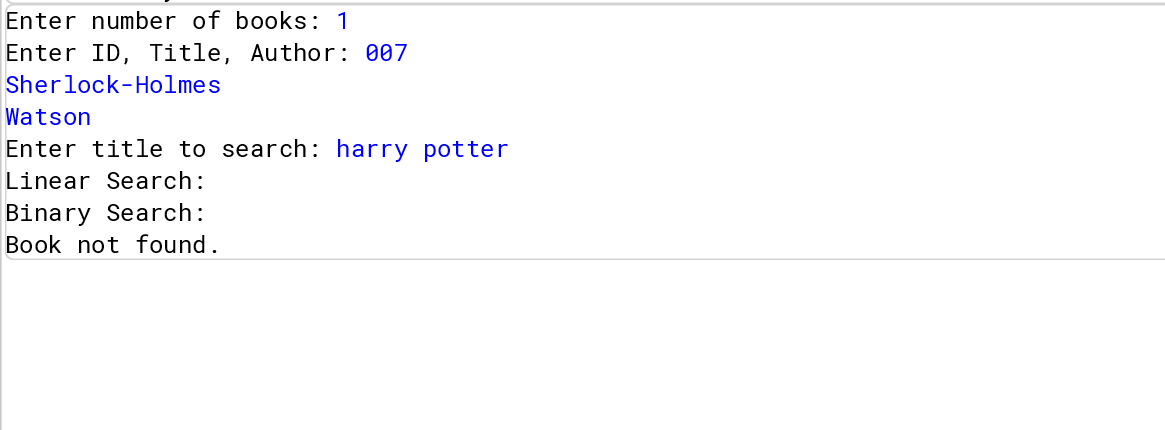
}

}



**Output:**





| **Search Type** | **Best** | **Avg** | **Worst** |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

**When to Use:**

* **Linear Search**: Unsorted or small datasets.
* **Binary Search**: Large and sorted datasets.

**Exercise 7: Financial Forecasting**

**Scenario:** You are developing a financial forecasting tool that predicts future values based on past data.

import java.util.Scanner;

public class FinancialForecasting {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter initial amount: ");

double amount = sc.nextDouble();

System.out.print("Enter growth rate (%): ");

double rate = sc.nextDouble();

System.out.print("Enter number of years: ");

int years = sc.nextInt();

double futureValue = forecast(amount, rate, years);

System.out.printf("Future value after %d years: ₹%.2f\n", years, futureValue);

}

static double forecast(double amount, double rate, int years) {

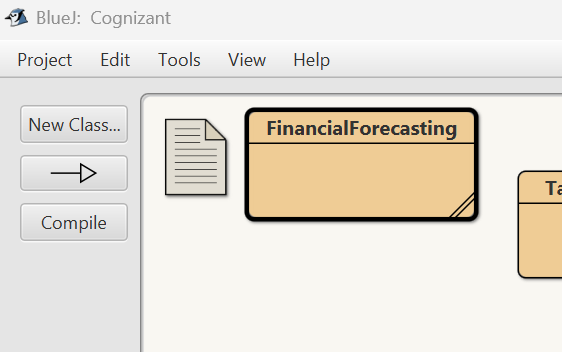
if (years == 0)

return amount;

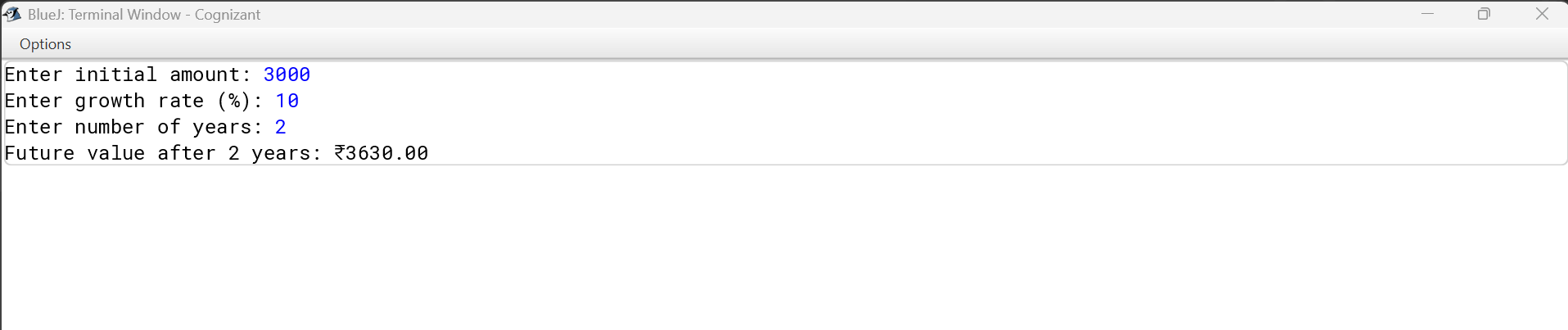
return forecast(amount \* (1 + rate / 100), rate, years - 1);

}

}



**Output:**



**Time Complexity:**

* **Recursive solution**: Often exponential, e.g., O(2^n) for naive Fibonacci.
* Optimized via **memoization** or converting to **iterative**: O(n)

**Optimization:**

* Use **Dynamic Programming** to cache results.
* Avoids redundant computations in recursive calls.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Thank You\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**