**Data Structures and Algorithms Solutions**

**1.Inventory Management System:**

**Explain why data structures are essential in handling large inventories?**

Data Structures are used to organize the data in an effective manner. We choose data structures according to our requirements. They help in fast access to the data and some of the data structures like set help in storing unique values avoiding duplicate data, on the other hand heap helps in maintaining the data in sorted fashion. In this way each data structure has its own benefits.

**Discuss the type of data structures suitable for this problem?**

We can either use an ArrayList or a Map. According to me choosing a map would be an efficient approach because it helps in fast retrieval of data, where as ArrayList is similar to an array but the only diff is that it can be modified dynamically. Hence in this problem it is better to choose HashMap.

**Product.java:**

public class Product {

    private int productId;

    private String productName;

    private String quality;

    private float price;

    public Product(int pId,String pName,String quality,float price) {

        this.productId=pId;

        this.productName=pName;

        this.quality=quality;

        this.price=price;

    }

    public String getProductName() {

        return this.productName;

    }

    public void setProductName(String pName) {

        this.productName=pName;

    }

     public String getQuality() {

        return this.quality;

    }

    public void setQuality(String quality) {

        this.quality = quality;

    }

     public int getProductId() {

        return this.productId;

    }

    public void setProductId(int pId) {

        this.productId=pId;

    }

     public float getPrice() {

        return this.price;

    }

    public void setPrice(float price) {

        this.price=price;

    }

@Override

    public boolean equals(Object o) {

        if(this==o) return true;

        if(!(o instanceof Product)) return false;

        Product product = (Product)o;

        return productId==product.productId;

    }

    @Override

    public int hashCode() {

        return Integer.hashCode(productId);

    }

    public String toString() {

        return "ProductId: " + this.productId+" productName: "+this.productName+" quality: "+this.quality+" Price: "+this.price+"\n";

    }

}

**InventoryManager.java**

import java.util.Collection;

import java.util.HashMap;

public class InventoryManager {

    private HashMap<Integer,Product> inventory;

    public InventoryManager() {

        inventory = new HashMap<>();

    }

    public void addProduct(Product product) {

        if(inventory.containsKey(product.getProductId())) {

            System.out.println("The product id: "+product.getProductId()+"already exists for product "+product.getProductName());

        }

        else {

            inventory.put(product.getProductId(),product);

        }

    }

    public void removeProduct(int id) {

        if(inventory.containsKey(id)) {

            inventory.remove(id);

        }

        else {

            System.out.println("Key which you want to delete is not found");

        }

    }

    public void updateProduct(int id,Product newProduct) {

        if(!inventory.containsKey(id)) {

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

        }

        else {

            inventory.put(id,newProduct);

        }

    }

    public Collection getAllProducts() {

        return inventory.values();

    }

}

**App.java**

public class App {

    public static void main(String[] args) {

        InventoryManager inventoryM = new InventoryManager();

        DisplayProductList display = new DisplayProductList();

        Product p1 = new Product(1,"Shampoo","good",5.0f);

        Product p2 = new Product(2,"Biscuit","Not bad",10.0f);

        inventoryM.addProduct(p1);

        inventoryM.addProduct(p2);

        display.diplayProducts(inventoryM.getAllProducts());

        inventoryM.removeProduct(2);

        System.out.println("After deleting product pId: 2");

        display.diplayProducts(inventoryM.getAllProducts());

        //updating

        System.out.println("Now updating  product with pid 1 with the contents of p2");

        inventoryM.updateProduct(1, p2);

    }

}

**Output:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Time Complexity:**

addProduct: O(1) [O(n) in worst case]

removeProduct: O(1) [O(n) in worst case]

updateProduct: O(1)

displayProducts: O(n) where n is the number of products.

The time complexity is already optimized because we are using hashmap, if we would have used ArrayList the Time Complexity would be O(n) for all the operations[add, remove, display,update]

**2.E-Commerce platform Search function**

**Explain Big O Notation and how it helps in analyzing algorithms?**

Big O Notation represents the worst case time complexity of an algorithm. By representing the time complexity of an algorithm in a big O Notation we can identify what is the maximum time that any algorithm could take to solve a problem.

**Describe the best, average, and worst-case scenarios for search operations.**

Search operation means we are trying to find a value from a list of values. If the list is completely reverse that is if we are sorting the list in ascending order and it is in non-ascending order at the beginning then it is the -**worst case.**

And if the numbers in the list are randomly placed then it is the – **average case.**

And if the numbers are already in the sorted order before sorting then it is the -**best case scenario.**

**Product.java**

public class Product {

    private int productId;

    private String productName;

    private String category;

    public Product(int productId,String productName, String category) {

        this.productId=productId;

        this.productName=productName;

        this.category=category;

    }

    public int getProductId() {

        return this.productId;

    }

    public void setProductId(int id) {

        this.productId=id;

    }

    public String getProductName() {

        return this.productName;

    }

    public void setProductName(String name) {

        this.productName = name;

    }

    public String getProductCategory() {

        return this.category;

    }

    public void setProductCategory(String category) {

        this.category=category;

    }

    public String toString() {

        return "Product ID: "+productId+" Product name: "+productName+" category: "+category;

    }

}

**Search.java**

public class Search {

    public Product linearSearch(Product[] productArray,int count,int targetProductId) {

        int n = productArray.length;

        for(int i=0;i<count;i++) {

            if(productArray[i].getProductId()==targetProductId) {

                Product res = productArray[i];

                return res;

            }

        }

        return null;

    }

    //binary search

    public Product binarySearch(Product[] productArray,int count,int targetProductId) {

        int left = 0;

        int right = count-1;

        while(left<=right) {

            int mid = (left+right)/2;

            if(productArray[mid].getProductId()==targetProductId) {

                return productArray[mid];

            }

            else if(productArray[mid].getProductId() < targetProductId) {

                left = mid+1;

            }

            else {

                right = mid-1;

            }

        }

        return null;

    }

}

**ProductManager.java**

public class ProductManager {

    static Product[] productArray = new Product[1000];

    int count = 0;

    public void addProduct(Product product) {

        productArray[count]=product;

        count++;

    }

    public Product[] getProduct() {

        return productArray;

    }

    public int getProductCount(){

        return count;

    }

}

**SortProducts.java**

import java.util.Arrays;

import java.util.Comparator;

public class SortProducts {

    public void sort(Product[] array,int count) {

        Arrays.sort(array,0,count,new Comparator<Product>() {

            public int compare(Product p1,Product p2) {

                return Integer.compare(p1.getProductId(), p2.getProductId());

            }

        });

    }

}

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

        ProductManager pm = new ProductManager();

        Product p1 = new Product(1,"Washing Machine","Electronic devices");

        pm.addProduct(p1);

        Product p2 = new Product(2,"Mango","Fruits");

        pm.addProduct(p2);

        Product p3 = new Product(4,"Sneakers","Shoes");

        pm.addProduct(p3);

        Product p4 = new Product(3,"Suit","Dress");

        pm.addProduct(p4);

        System.out.println("Product Array:");

        for(int i=0;i<pm.getProductCount();i++) {

            System.out.println(pm.getProduct()[i]);

        }

        Search search=new Search();

        Product res = search.linearSearch(pm.getProduct(), pm.getProductCount(),p4.getProductId());

        System.out.println("Linear search res when searched for Productid 3");

        System.out.println(res);

        System.out.println("Linear search res when searched for product id that is invalid 101");

        Product res1 = search.linearSearch(pm.getProduct(),pm.getProductCount(),101);

        System.out.println(res1);

        //binary search

        SortProducts sortObj = new SortProducts();

        sortObj.sort(pm.getProduct(),pm.getProductCount());

        System.out.println("Sorted Products array: ");

         for(int i=0;i<pm.getProductCount();i++) {

            System.out.println(pm.getProduct()[i]);

        }

        Product res2 = search.binarySearch(pm.getProduct(),pm.getProductCount(), p3.getProductId());

        System.out.println("Searching using binary search for product id : 4");

        System.out.println(res2);

        System.out.println("Searching for ivalid ProductId such as 1001 using binary search");

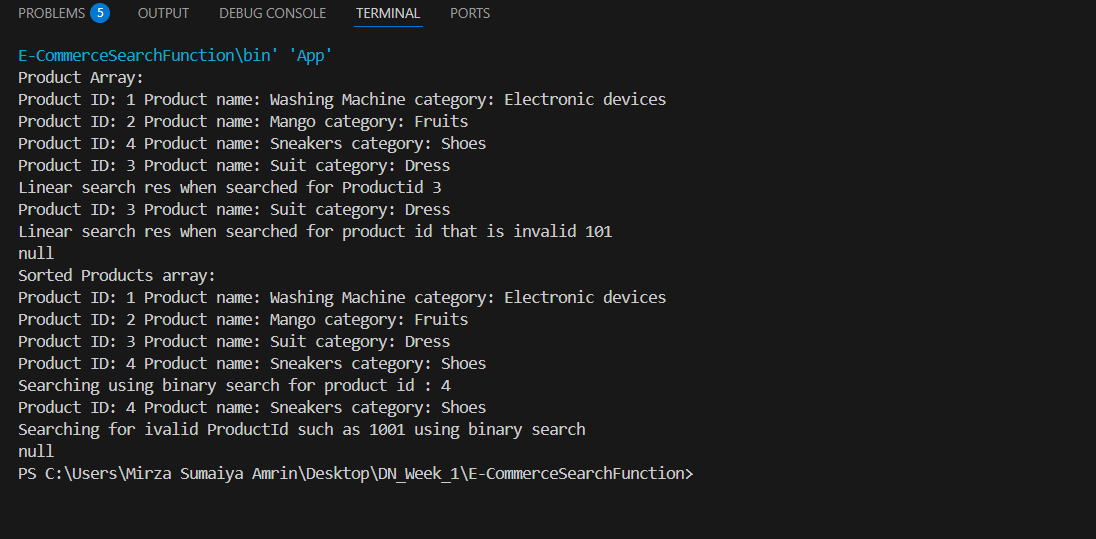
        Product res3 = search.binarySearch(pm.getProduct(),pm.getProductCount(), 1001);

        System.out.println(res3);

    }

}

**Output:**

****

**Time Complexity:**

LinearSearch : O(n)

BinarySearch: O(log n)

Binary Search is more suitable for this task to provide faster responses.

**3.Sorting Customer Orders:**

**Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).**

**Bubble sort:** It is a sorting technique when sorts the array by comparing with the adjacent elements and at the end of one iteration the largest element will be at the last

Time Complexity: O(n^2)

**Insertion sort:** It is sorting technique in which the array is sorted by every time including one element and sorting the array till that element.

Time Complexity: O(n^2)

**Quick Sort:** It is a sorting technique in which one element in the array is selected as pivot and is placed in its correct sorted position and then the same process is repeated from 0 to pivot-1 index and pivot+1 to n where n is length of array.

**Merge Sort:** It is a sorting technique in which the array is divided into 2 halves each time till it gets divided till a single element and then merge the array in such a way that it gets sorted.

**Order.java**

public class Order {

    private int orderId;

    private String customerName;

    private float totalPrice;

    public Order(int orderId,String customerName,float totalPrice) {

        this.orderId=orderId;

        this.customerName=customerName;

        this.totalPrice=totalPrice;

    }

    public int getOrderId() {

        return this.orderId;

    }

    public void setOrderId(int id) {

        this.orderId=id;

    }

    public String getCustomerName() {

        return this.customerName;

    }

    public void setCustomerName(String name) {

        this.customerName=name;

    }

    public void setTotalPrice(float totalPrice) {

        this.totalPrice=totalPrice;

    }

    public float getTotalPrice(){

        return totalPrice;

    }

    public String toString(){

        return "OrderId: "+orderId+" customer name: "+customerName+" Total price: $ "+totalPrice;

    }

}

**ManageOrders.java**

public class ManageOrders {

    static Order[] orderList = new Order[1000];

    static int count = 0;

    public void addOrder(Order o) {

        orderList[count] = o;

        count++;

    }

    public Order[] getOrders() {

        return orderList;

    }

    public int getCount() {

        return count;

    }

}

**SortOrders.java**

public class SortOrders {

    public void bubbleSortOrders(Order[] array, int count) {

    for (int i = 0; i < count - 1; i++) {

        for (int j = 0; j < count - i - 1; j++) {

            if (array[j].getTotalPrice() > array[j + 1].getTotalPrice()) {

                // Swap array[j] and array[j + 1]

                Order temp = array[j];

                array[j] = array[j + 1];

                array[j + 1] = temp;

            }

        }

    }

}

public void quickSortOrders(Order[] array, int count) {

    quickSort(array, 0, count - 1);

}

private void quickSort(Order[] array, int low, int high) {

    if (low < high) {

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

        quickSort(array, low, pi - 1);

        quickSort(array, pi + 1, high);

    }

}

private int partition(Order[] array, int low, int high) {

    float pivot = array[high].getTotalPrice();

    int i = low - 1;

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

        if (array[j].getTotalPrice() <= pivot) {

            i++;

            // Swap array[i] and array[j]

            Order temp = array[i];

            array[i] = array[j];

            array[j] = temp;

        }

    }

    // Swap array[i + 1] and array[high] (pivot)

    Order temp = array[i + 1];

    array[i + 1] = array[high];

    array[high] = temp;

    return i + 1;

}

}

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

        ManageOrders manage = new ManageOrders();

        Order order1= new Order(100,"Sumaiya",500.0f);

        manage.addOrder(order1);

        Order order2= new Order(101,"Sameera",255.0f);

        manage.addOrder(order2);

        for(int i=0;i<manage.getCount();i++) {

            System.out.println(manage.getOrders()[i]);

        }

        System.out.println("Sorted by total price");

        SortOrders sort = new SortOrders();

        System.out.println("Bubble sort: ");

        sort.bubbleSortOrders(manage.getOrders(), manage.getCount());

        for(int i=0;i<manage.getCount();i++) {

            System.out.println(manage.getOrders()[i]);

        }

        System.out.println("Quick Sort: ");

        sort.quickSortOrders(manage.getOrders(), manage.getCount());

        for(int i=0;i<manage.getCount();i++) {

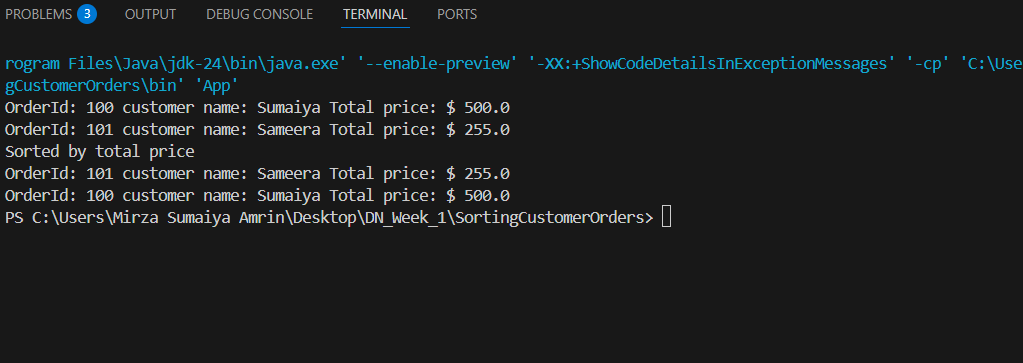
            System.out.println(manage.getOrders()[i]);

        }

    }

}

**Output:**



**Time Complexity:**

**Bubble Sort: O(n^2)**

**Quick Sort: O(n log n)**

**4. Employee Management System:**

**Explain how arrays are represented in memory and their advantages?**

An array is a linear data structure used to store the data in linear fashion. Arrays are static that is the fixed number of elements are only stored into the array. The size of array is fixed. And it is stored in the contiguous manner in the memory. An array does not allows to make dynamic change in size during the execution.

**Discuss the limitations of arrays and when to use them.**

The limitations of array would be inefficient memory usage. Because it requires contiguous storage. We can use arrays when we know the size of data.

**Employee.java**

public class Employee {

    private int employeeId;

    private String name;

    private String position;

    private float salary;

    public Employee(int employeeId,String name,String position, float salary) {

        this.employeeId = employeeId;

        this.name = name;

        this.position = position;

        this.salary = salary;

    }

    public int getEmployeeId() {

        return this.employeeId;

    }

    public void setEmployeeId(int id) {

        this.employeeId=id;

    }

    public String getName() {

        return this.name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public String getPosition() {

        return this.position;

    }

    public void setPosition(String position) {

        this.position=position;

    }

    public float getSalary() {

        return this.salary;

    }

    public void setSalary(float salary) {

        this.salary=salary;

    }

    public String toString() {

        return "Empoyee id: "+employeeId+" name: "+name+" position: "+position+" Salary: "+salary;

    }

}

**ManageEmployee.java**

public class ManageEmployee {

    Employee[] employeeList = new Employee[1000];

    static int count=0;

    public void addEmployee(Employee employee) {

        employeeList[count] = employee;

        count++;

    }

    public void removeEmployee(int id) {

        Employee[] refinedList = new Employee[count];

        int ind = 0;

        boolean found = false;

        for(int i=0;i<count;i++) {

            if(employeeList[i].getEmployeeId()==id) {

                found = true;

                continue;

            }

            refinedList[ind++]=employeeList[i];

        }

        if (found) {

            count--; // Only reduce count if we actually found and removed an employee

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

                employeeList[i] = refinedList[i];

            }

            // Optionally null out the rest of the array

            for (int i = ind; i < employeeList.length; i++) {

                employeeList[i] = null;

            }

        } else {

            System.out.println("Employee with ID " + id + " not found.");

        }

    }

    public Employee[] getEmployees() {

        return employeeList;

    }

}

**SearchEmployee.java**

public class SearchEmployee {

    public Employee searchEmployee(Employee[] employeeList,int count,int id) {

        int left = 0;

        int right = count-1;

        while(left<=right) {

            int mid = (left+right)/2;

            if(employeeList[mid].getEmployeeId()==id) {

                return employeeList[mid];

            }

            else if(employeeList[mid].getEmployeeId()<id) {

                left=mid+1;

            }

            else {

                right=mid-1;

            }

        }

        return null;

    }

}

**TraverseEmployee.java**

public class TraverseEmployee {

    public void traverseEmployeeList(int startInd,int endInd,Employee[] employeeList,int count) {

        if(startInd <0 || startInd>count ) {

            System.out.println("Invalid start index");

            return;

        }

        if(endInd<0 || endInd>count) {

            System.out.println("Invalid end index");

            return;

        }

        if(startInd>endInd) {

            System.out.println("Start index cannot be greater than endIndex");

            return;

        }

        for(int i=startInd;i<=endInd;i++) {

            if(employeeList[i]!=null)

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

            else {

                System.out.println("Employee at "+i+ " is null");

            }

        }

**SortEmployee.java**

import java.util.Arrays;

import java.util.Comparator;

public class SortEmployee {

    public void sortEmployees(Employee[] array,int count) {

        Arrays.sort(array,0,count,new Comparator<Employee>() {

            public int compare(Employee e1,Employee e2) {

                return Integer.compare(e1.getEmployeeId(), e2.getEmployeeId());

            }

        });

    }

}

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

        ManageEmployee manager = new ManageEmployee();

        TraverseEmployee t = new TraverseEmployee();

        Employee e1 = new Employee(1,"Sumaiya","Full Stack Engineer",45000.0f);

        Employee e2 = new Employee(3,"Sarah","Tester",30000.0f);

        Employee e3 = new Employee(4, "Saba", "Software Developer", 100000.0f);

        Employee e4 = new Employee(2,"Afra","Project Manager",150000.0f);

        manager.addEmployee(e1);

        manager.addEmployee(e2);

        manager.addEmployee(e3);

        manager.addEmployee(e4);

        System.out.println("Employee list after adding all Employees");

        t.traverseEmployeeList(0, manager.count-1, manager.getEmployees(), manager.count);

        manager.removeEmployee(e2.getEmployeeId());

        System.out.println();

        System.out.println("Employee list after deleting the employee with employee id 3");

        //traversing

        t.traverseEmployeeList(0, manager.count-1, manager.getEmployees(), manager.count);

        System.out.println();

        SortEmployee sort = new SortEmployee();

        sort.sortEmployees(manager.getEmployees(), manager.count);

        System.out.println("Employee list after sorting");

        t.traverseEmployeeList(0,manager.count-1, manager.getEmployees(), manager.count);

        System.out.println();

        SearchEmployee search = new SearchEmployee();

        Employee res = search.searchEmployee(manager.getEmployees(), manager.count, e3.getEmployeeId());

        System.out.println("Search for employee with id 4");

        System.out.println(res);

    }

}

**Output:**

A computer screen with text on it

AI-generated content may be incorrect.

**Time Complexity:**

Add: O(n)

Search: O(n)

Traverse: O(n)

Delete: O(n)[worst case]

**5. Task Management System:**

**Explain the different types of linked lists (Singly Linked List, Doubly Linked List).**

LinkedList is a linear data structure which will be having nodes linked with each other. Unlike arrays they do not require contiguous memory locations to store the data hence the linked lists are used for better memory management. There are different types of linked list. Some of them are:

1. Singly Linked List:

A singly linked list has a head node and only one link pointing to the next node. We can only traverse in one direction in singly linked list.

1. Doubly linked list:

A doubly linked list has 2 links in each node one pointing to the previous node and the other pointing to next node.

1. Circular Linked List: It is a type of linked list where the last node next pointer will be linked to the first node previous pointer.

**Discuss the advantages of linked lists over arrays for dynamic data.**

Linked lists are dynamic where as arrays are static. We can modify linked lists size at the run time while we can not modify the size of arrays during run time due to their static nature. We use linked lists to store the data which has a variable length.

**Node.java**

public class Node {

    private int taskId;

    private String taskName;

    private String status;

    private Node next;

    public Node(int taskId,String taskName,String status) {

        this.taskId=taskId;

        this.taskName=taskName;

        this.status=status;

        this.next=null;

    }

    public int getTaskId() {

        return taskId;

    }

    public void setTaskId(int taskId) {

        this.taskId = taskId;

    }

    public String getTaskName() {

        return taskName;

    }

    public void setTaskName(String taskName) {

        this.taskName = taskName;

    }

    public String getStatus() {

        return status;

    }

    public void setStatus(String status) {

        this.status = status;

    }

    public Node getNext() {

        return next;

    }

    public void setNext(Node next) {

        this.next = next;

    }

    public String toString() {

        return "TaskId: "+taskId+" Task name: "+taskName+" status "+status;

    }

}

**TaskManager.java**

public class TaskManager {

    Node head;

    public void insert(Node node) {

        if(head==null) {

            head=node;

            return;

        }

        Node current = head;

        while(current.getNext()!=null) {

            current=current.getNext();

        }

        current.setNext(node);

    }

    public Node delete(Node node) {

        if(head == null||node==null) {

            return head;

        }

        if(head == node) {

            head = head.getNext();

            return head;

        }

        Node current = head;

        while(current.getNext()!=null) {

            if(current.getNext() == node) {

                current.setNext(node.getNext());

            }

            current=current.getNext();

        }

        return head;

    }

}

**Traverse.java**

public class Traverse {

    public void traverse(Node head) {

        Node currNode=head;

        if(head == null) {

            System.out.println("TaskList is empty");

        }

        while(currNode!=null) {

            System.out.println(currNode);

            currNode=currNode.getNext();

        }

    }

}

**Sort.java**

public class Sort {

    // Sort the linked list and return the new head

    public Node mergeSort(Node head) {

        if (head == null || head.getNext() == null) {

            return head;

        }

        // Split the list into two halves

        Node middle = getMiddle(head);

        Node nextOfMiddle = middle.getNext();

        middle.setNext(null);

        // Recursively sort the left and right halves

        Node left = mergeSort(head);

        Node right = mergeSort(nextOfMiddle);

        // Merge sorted halves

        return sortedMerge(left, right);

    }

    // Merge two sorted linked lists

    private Node sortedMerge(Node a, Node b) {

        if (a == null) return b;

        if (b == null) return a;

        Node result;

        if (a.getTaskId() <= b.getTaskId()) {

            result = a;

            result.setNext(sortedMerge(a.getNext(), b));

        } else {

            result = b;

            result.setNext(sortedMerge(a, b.getNext()));

        }

        return result;

    }

    // Find the middle of the linked list

    private Node getMiddle(Node head) {

        if (head == null) return head;

        Node slow = head;

        Node fast = head;

        while (fast.getNext() != null && fast.getNext().getNext() != null) {

            slow = slow.getNext();

            fast = fast.getNext().getNext();

        }

        return slow;

    }

}

**Search.java**

public class Search {

    public Node linearSearch(Node head,int targetTaskId) {

        if(head == null) return null;

        Node current = head;

        while(current.getNext()!=null){

            if(current.getTaskId()==targetTaskId) {

                return current;

            }

        }

        return null;

    }

    //binary search

    public Node binarySearch(Node head,int targetTaskId) {

        Node left = head;

        Node right = null;

        while(left!=right) {

            Node mid = getMiddle(left,right);

            if(mid == null) return null;

            if(mid.getTaskId()==targetTaskId) return mid;

            else if(mid.getTaskId()<targetTaskId) left=mid.getNext();

            else right=mid;

        }

        return null;

    }

    public static Node getMiddle(Node left, Node right) {

        if (left == null) return null;

        Node slow = left;

        Node fast = left;

        while (fast != right && fast.getNext() != right) {

            if (fast != right && fast != null) {

                fast = fast.getNext();

                slow = slow.getNext();

            }

        }

        return slow; // This is the middle node between left and right

    }

}

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

        TaskManager tll = new TaskManager();

        Traverse t = new Traverse();

        Sort sort = new Sort();

        Search search = new Search();

        Node n1 = new Node(1, "Reading", "Pending");

        Node n2 = new Node(7, "Cooking", "Completed");

        Node n3 = new Node(4,"Sleeping","Started");

        Node n4 = new Node(3,"Swimming","Completed");

        tll.insert(n1);

        tll.insert(n2);

        tll.insert(n3);

        tll.insert(n4);

        System.out.println("Initial Linked List: ");

        t.traverse(n1);

        System.out.println();

        System.out.println("Removing 1st task");

        Node head = tll.delete(n1);

        t.traverse(head);

        System.out.println();

        System.out.println("Linear search searching for task id 7");

        Node res1 = search.linearSearch(head, 7);

        System.out.println(res1);

        System.out.println();

        head = sort.mergeSort(head);

        System.out.println("Sorted Linked List: ");

        t.traverse(head);

        System.out.println();

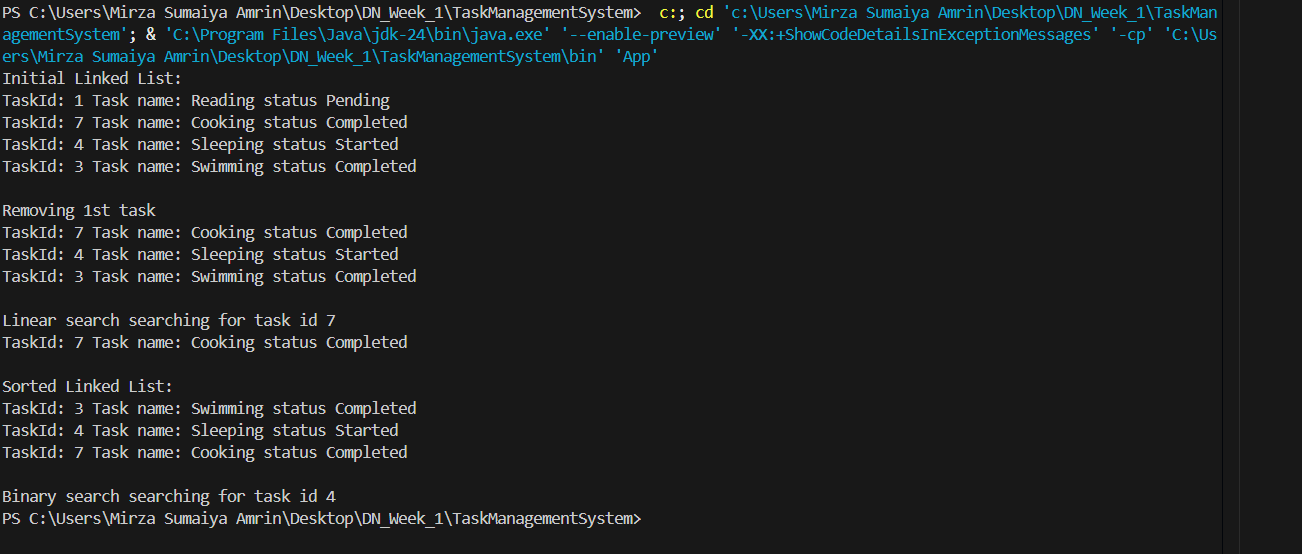
        System.out.println("Binary search searching for task id 4");

        Node res2 = search.binarySearch(head, 4);

    }

}

**Output:**



**Time Complexity:**

Add: O(1)

Search: O(n)

Traverse: O(n)

Delete: O(n)

**6.** Library **Management System:**

**Explain linear search and binary search algorithms.**

Linear Search: It is a technique in which we search the target value in a collection of elements from the starting to the end point of the collection.

Binary Search: It is a technique which is used to search for a target value in a sorted collection. It compares the target with the middle value and take elimination decisions.

**Discuss when to use each algorithm based on the data set size and order.**

We need to use linear search when we have unsorted dataset. And use binary search when we have sorted collection. If we have huge collection which is unsorted we do not move with binary search because it takes O(nLogn) time to sort linear search would be the better approach for this kind of data sets.

**Book.java**

public class Book {

    private int bookId;

    private String title;

    private String author;

    public Book(int bookId, String title, String author) {

        this.bookId=bookId;

        this.title=title;

        this.author=author;

    }

    public int getBookId() {

        return bookId;

    }

    public void setBookId(int bookId) {

        this.bookId = bookId;

    }

    public String getTitle() {

        return title;

    }

    public void setTitle(String title) {

        this.title = title;

    }

    public String getAuthor() {

        return author;

    }

    public void setAuthor(String author) {

        this.author = author;

    }

    public String toString() {

        return "Book id: "+bookId+" title: "+title+" Author: "+author;

    }

}

**BookManager.java**

import java.util.HashMap;

public class BookManager {

    HashMap<String,Book> bookList;

    public BookManager() {

        bookList=new HashMap<>();

    }

    public void addBook(Book b) {

        if(bookList.containsKey(b.getTitle())) {

            System.out.println("The book with title: "+b.getTitle()+" Already exists");

        }

        else {

            bookList.put(b.getTitle(),b);

        }

    }

    public void removeBook(String title) {

        if(bookList.containsKey(title)) {

            bookList.remove(title);

        }

        else {

            System.out.println("The book with title "+title+" is not present");

        }

    }

    public HashMap<String,Book> getAllBooks() {

        return bookList;

    }

}

**Search.java**

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

public class Search {

    public Book linearSeach(HashMap<String,Book> bookList,String targetTitle) {

        for(String title:bookList.keySet()) {

            if(title==targetTitle) {

                return bookList.get(title);

            }

        }

        return null;

    }

    public Book binarySearch(HashMap<String,Book> bookList,String targetTitle) {

        List<String> titles = new ArrayList<>(bookList.keySet());

        int left = 0;

        int right = bookList.size()-1;

        while(left<=right) {

            int mid = (left+right)/2;

            int cmp = targetTitle.compareTo(titles.get(mid));

            if(cmp == 0) {

                return bookList.get(titles.get(mid));

            }

            else if(cmp>0) {

                left = mid+1;

            }

            else {

                right = mid-1;

            }

        }

        return null;

    }

}

**SortBooks.java**

import java.util.\*;

public class SortBooks {

    public HashMap<String,Book> sortBooks(HashMap<String,Book> bookList) {

        TreeMap<String,Book> sortedMap = new TreeMap<>(bookList);

        // Print sorted map

        for (Map.Entry<String,Book> entry : sortedMap.entrySet()) {

            bookList.put(entry.getKey(),entry.getValue());

        }

        return bookList;

    }

}

**App.java**

import java.util.HashMap;

import java.util.Map;

public class App {

    public static void main(String[] args) throws Exception {

        BookManager manage = new BookManager();

        SortBooks sort = new SortBooks();

        Search search = new Search();

        Book b1 = new Book(1001,"Let Us C","Yashwant Kanetkar");

        Book b2 = new Book(1002,"Happy Ending","Hertbert k");

        Book b3 = new Book(1003,"The mischievious rat","John C");

        manage.addBook(b1);

        manage.addBook(b2);

        manage.addBook(b3);

        HashMap<String,Book> bookList = sort.sortBooks(manage.getAllBooks());

        System.out.println("Initial book List: ");

        for(Map.Entry<String,Book> entry : bookList.entrySet()) {

            System.out.println(entry.getKey()+"->"+entry.getValue());

        }

        System.out.println();

        //linear search

        System.out.println("Linear search result when searched for title 'The mischievious rat'");

        Book res1 = search.linearSeach(bookList, "The mischievious rat");

        System.out.println(res1+"\n");

        System.out.println("Binary search result when searched for title 'Happy Ending'");

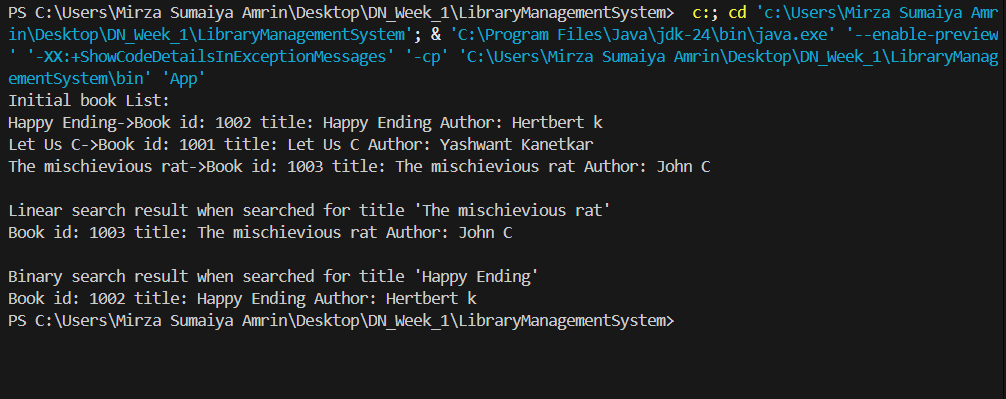
        Book res = search.binarySearch(bookList, "Happy Ending");

        System.out.println(res);

    }

}

**Output:**



**Time Complexity:**

Linear Search: O(n)

Binary Search: O(log n)

**7. Financial Forecasting:**

**Explain the concept of recursion and how it can simplify certain problems.**

Recursion is a process of dividing a huge problem into sub problems and then solving the sub problems. A function calling itself is called as Recursion. A recursive solution has a base condition which specifies when to stop the recursion.

**Explain how to optimize the recursive solution to avoid excessive computation.**

To avoid excessive computation in recursion we use dynamic programming.

Memoization: Stores the precalculated data to avoid extra computations.

Tabulation: Eliminates recursion and tries to solve it using iterative approach.

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

        System.out.println("Predicted Value for year 10: "+ predictValue(10, 10000, 12000));

    }

    public static double predictValue(int year,int initialyear,int secondYear) {

        if(year==0) {

            return initialyear;

        }

        if(year == 1) {

            return secondYear;

        }

        double one = predictValue(year-1, initialyear, secondYear);

        double second = predictValue(year-2, initialyear, secondYear);

        double growth = (one-second)/second;

        double avgGrowth = growth/(year-1);

        return one\*(1+avgGrowth);

    }

}

Output:

A screen shot of a computer code

AI-generated content may be incorrect.

**Time Complexity:**

O(2^n) where n is number of years.