**Exercise 1: Inventory Management System**

**Step-1: Understand the Problem:**

* Explain why data structures and algorithms are essential in handling large inventories.

Large Inventories include a lot of types of products in large quantities. Managing all these products is a tedious task. So, using the right data structures and algorithms to manage inventory is important. Each type of data structure provides us with their own advantages and has its own set of algorithms. Let’s take 4 data structures – ArrayList, LinkedList, TreeMap, HashMap.

ArrayList is the simplest data structure which can be used to manage inventory. Its access time complexity is O(1). But search time complexity is O(n) in case of Linear Search and O(logn) in case of Binary Search. And also insert and delete operations in ArrayList take O(n) time complexity. It stores data in a sequential location in memory which is hard to find in case of large data. We hardly ever use this data structure in case of large data.

LinkedList provides us with a O(1) insert and delete time complexity as it stores data in a non-sequential manner. But search and access times are same as ArrayList O(n). It also doesn’t provide random access time like O(1) for ArrayList. It is useful for efficient memory management but not in the case of large data.

HashMap is the most optimized data structure which provides an access time of O(1) using hashing. It provides search, insert and delete with O(1) time complexity. It is ideal for medium and small data but when it comes to large data it increases overhead due to rehashing and resizing. And also, an inefficient hash function can also cause excess overhead. Thus, it is not preferred for large data.

TreeMap is a red black tree data structure which sorts key value pairs of a map data structure. It sorts and stores key value pairs inserted and provides us an access time complexity of O(logn) using Binary Search algorithm. It is most suited for large data as it provides an optimization in access times. Likewise, it also provides O(logn) time complexity for insert and delete operations.

* Discuss the types of data structures suitable for this problem.

The types of data structures suitable for this problem are TreeMap and HashMap. As said earlier TreeMap is more optimized for large data as it doesn’t cause extra overhead as compared to HashMap. It depends on the size of the data and thus if the data is not so large we choose HashMap and if the data is large enough, then we use TreeMap.

**Code:**

package week1.module2;

import java.util.TreeMap;

class Product1{

private String productId,productName;

private long quantity;

private double price;

// Default Constructor for initializing variables

public Product1(String productId, String productName, long quantity, double price) {

this.productId=productId;

this.productName=productName;

this.quantity=quantity;

this.price=price;

}

// Getter methods

String getProductId() {

return productId;

}

String getproductName() {

return productName;

}

long getQuantity() {

return quantity;

}

double getPrice() {

return price;

}

// Setter methods

void setProductId(String productId) {

this.productId=productId;

}

void setproductName(String productName) {

this.productName=productName;

}

void setQuantity(long quantity) {

this.quantity=quantity;

}

void setPrice(double price) {

this.price=price;

}

}

class Inventory{

// TreeMap to store products with O(logn) access complexity

private TreeMap<String,Product1> treeMap;

public Inventory() {

treeMap=new TreeMap<String, Product1>();

}

// Adding a Product

void add(Product1 product) {

// Checking for duplicate keys

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

System.***out***.println("Product with Product Id:"+product.getProductId()+" already exists");

}

else {

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

System.***out***.println("Added Product with Product Id: "+product.getProductId());

}

}

// Updating a product

void update(Product1 product) {

// Checking if a product exists to update

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

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

System.***out***.println("Updated Product with Product Id: "+product.getProductId());

}

else {

System.***out***.println("Product with Product Id:"+product.getProductId()+" doesn't exist");

}

}

// Deleting a product

void delete(Product1 product) {

// Checking if a product exists to delete

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

treeMap.remove(product.getProductId(),product);

System.***out***.println("Deleted Product with Product Id: "+product.getProductId());

}

else {

System.***out***.println("Product with Product Id:"+product.getProductId()+" doesn't exist");

}

}

}

public class Ex1Main {

public static void main(String[] args) {

Inventory inventory=new Inventory();

// Add Products

Product1 product1=new Product1("61AH33","Samsung S24 Ultra",300,50000);

inventory.add(product1);

Product1 product2=new Product1("61OH33","Oneplus Nord Ce 4",100,26000);

inventory.add(product2);

// Update Products

product1.setPrice(51000);

inventory.update(product1);

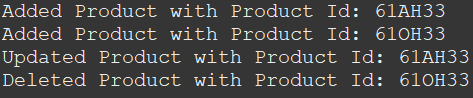
// Delete Products

inventory.delete(product2);

}

}

**Output:**

****

**Step-4 Analysis:**

* Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.

The time complexity of the add operation is O(logn) as it inherently uses put() and containsKey() methods of TreeMap which have complexity O(logn). The time complexity of update operation is O(logn) as it inherently uses put() and containsKey() methods of TreeMap which have complexity O(logn). The time complexity of delete operation is O(logn) as it inherently uses remove() and containsKey() methods of TreeMap which have complexity O(logn).

* Discuss how you can optimize these operations.

We can optimize the methods by removing the containsKey() method from update and delete operations, but it is a check which is necessary as we shouldn’t update by adding an entire new Product or delete a Product which doesn’t exist at all. To overcome this, we can make sure in the frontend part that only existing Products are chosen for Update operation and delete operation. By doing this we can rule out the possibility of updating or deleting Products which don’t actually exist.

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

**Step-1 Understand Asymptotic Notation:**

* Explain Big O notation and how it helps in analyzing algorithms.

Big O Notation is an Asymptotic Notation which represents Worst Case Time Complexity. It helps us understand how much time an algorithm will take to execute at the maximum, thus helping us to rate the efficiency of an algorithm. Knowing the Big O time complexity of algorithms helps us compare an algorithm with other algorithms and choosing the best algorithm for the given application.

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

For Linear Search Algorithm, the best, average and worst-case scenarios are when the element is in the first, in the middle and in the last positions of the array having complexities O(1), O(n) and O(n).

For Binary Search Algorithm, the best, average and worst-case scenarios are when the element is in the middle, when it takes some divisions to find the element and when it takes maximum number of divisions to find the element having complexities O(1), O(logn) and O(logn).

For Binary Search Tree, the best, average and worst-case scenarios are when the element is at root, in between root and leaf nodes and at leaf node having complexities O(1), O(logn) and O(logn).

For Hash Table, the best, average and worst-case scenarios are when there are no collisions, hash function is good and load factor Is low, all elements are hashed to same bucket having complexities O(1), O(1) and O(n).

For Balanced Binary Search Tree, the best, average and worst-case scenarios are when the element is at root, in between root and leaf nodes and at leaf node having complexities O(1), O(logn) and O(logn). It is similar to Binary Search Tree but the height of the tree is minimum possible, so it is better than Binary Search Tree even though both have same complexity.

**Code:**

package week1.module2;

import java.util.ArrayList;

import java.util.Collections;

// Using Comparable Interface to sort products

class Product2 implements Comparable<Product2>{

private String productId,productName,category;

// Constructor to initialize data

public Product2(String productId,String productName, String category) {

this.productId=productId;

this.productName=productName;

this.category=category;

}

// Getter Methods

String getProductId() {

return productId;

}

String getProductName() {

return productName;

}

String getCategory() {

return category;

}

// Setter Methods

void setProductId(String productId) {

this.productId=productId;

}

void setProductName(String productName) {

this.productName=productName;

}

void setCategory(String category) {

this.category=category;

}

// Implementing compareTo() method to sort Products according to productId

@Override

public int compareTo(Product2 product) {

return this.productId.compareTo(product.productId);

}

// To print product and check

public String toString() {

return "Product Details:\nProduct Id: "+productId+"\nProduct Name: "+productName+"\nCategory: "+category;

}

}

class ECommerce{

// ArrayList to store unsorted data and sorted data

//we need not store sorted separately but here its just for 2 separate algorithms

//In real life application we only use 1 algorithm so we don't need 2 separate lists

ArrayList<Product2> list,sortedList;

public ECommerce() {

list=new ArrayList<Product2>();

sortedList=new ArrayList<Product2>();

}

public void add(Product2 product) {

list.add(product);

sortedList.add(product);

Collections.sort(sortedList);

}

// Linear Search

public Product2 linearSearch(String productId) {

for(Product2 product:list) {

if(product.getProductId()==productId) {

System.out.println("Product found by Linear Search Algorithm");

return product;

}

}

System.out.println("Product not found by Linear Search Algorithm");

return null;

}

// Binary Search

public Product2 binarySearch(String productId) {

int low,mid,high;

low=0;

high=sortedList.size()-1;

while(low<=high) {

mid=low+(high-low)/2;

if(sortedList.get(mid).getProductId().compareTo(productId)==0) {

System.out.println("Product found by Binary Search Algorithm");

return sortedList.get(mid);

}

else if(sortedList.get(mid).getProductId().compareTo(productId)<0) {

low=mid+1;

}

else {

high=mid-1;

}

}

System.out.println("Product not found by Binary Search Algorithm");

return null;

}

}

public class Ex2Main {

public static void main(String[] args) {

// Declaring e-commerce to store products

ECommerce eCommerce=new ECommerce();

// Creating Products to search

Product2 p1=new Product2("1A12","Samsung Galaxy S24 Ultra","Android Phone");

Product2 p2=new Product2("1A13","Samsung Galaxy S25 Ultra","Android Phone");

Product2 p3=new Product2("1A14","Oneplus Nord Ce4","Android Phone");

// Adding products to both list and sorted list

eCommerce.add(p3);

eCommerce.add(p1);

eCommerce.add(p2);

// Linear Search Algorithm

Product2 p4=eCommerce.linearSearch("1A13");

System.out.println(p4);

Product2 p5=eCommerce.linearSearch("1A23");

// Binary Search Algorithm

Product2 p6=eCommerce.binarySearch("1A13");

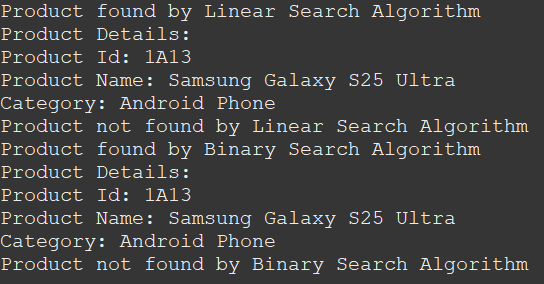
System.out.println(p6);

Product2 p7=eCommerce.binarySearch("1A23");

}

}

**Output:**

****

**Step-4 Analysis:**

* Compare the time complexity of linear and binary search algorithms

The best case in both algorithms is O(1). The average and worst case for Linear Search is O(n) and for Binary Search is O(logn). From the time complexities, we can see that Binary Search is a better algorithm than Linear Search but the problem is that for Binary Search Algorithm to work we need a sorted array. So, to create or maintain a sorted array we may need more time or other algorithms. But in case of large data this overhead and Binary Search growth becomes negligible compared to Linear Search Algorithm’s growth in time taken. Thus Binary Search is more efficient than Linear Search in case of large data. If our application contains small data enough to neglect this change, then we can choose Linear Search itself for simplicity and efficiency.

* Discuss which algorithm is more suitable for your platform and why.

As the given platform is an E-Commerce Platform which has large number of products to maintain and access efficiently, we need an optimized algorithm which works better for large data. Thus, we can use Binary Search Algorithm for optimized and fast performance instead of Linear Search Algorithm. We can implement this using a Sorted Array (which we did now) or a Binary Search Tree. For a Sorted Array, we need to implement Binary Search Algorithm and insert operations, instead we could have also used pre-defined collection TreeMap which implements Red Black Tree which is a Balanced Binary Search Tree. This is the most optimized for our case.

**Exercise 3: Sorting Customer Orders**

**Step-1 Understand Sorting Algorithms:**

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

Bubble sort is a O(n2) sorting algorithm. It is based on consecutive element comparisons. It works by comparing 2 consecutive elements and swapping if the first element is greater than the second. In this way, the largest element is taken to the end of the array. Then the array size is reduced and the same operation is done till last first element. As such the array is sorted by taking large elements to the last of the array.

Example:

Let’s take the following array

1 5 6 4 2

After successive iterations the array is transformed as follows. Right of | are all large elements replaced.

1 5 6 4 2 -> 1 5 6 4 2 -> 1 5 6 4 2 -> 1 5 4 6 2 -> 1 5 4 2|6

1 5 4 2|6 -> 1 5 4 2|6 -> 1 4 5 2|6 -> 1 4 2|5 6

1 4 2|5 6 -> 1 4 2|5 6 -> 1 2|4 5 6

1 2|4 5 6 -> 1|2 4 5 6

1|2 4 5 6-> |1 2 4 5 6

As such after swapping consecutive elements the array is sorted.

Insertion sort is a O(n2) sorting algorithm. It is based on inserting an element in a sorted array. The sorted array starts from the starting element. The next element after sorted array is inserted into the sorted array based on sorting the array completely with the new element. Then the sorted array size increases as a new element is added. As such new elements are added from the next elements of the sorted array thereby increasing the sorted array size and hence sorting the whole array.

Example:

Let’s take the following array

1 5 6 4 2

After successive iterations the array is transformed as follows. Left of | is the sorted part and right of | is the unsorted part.

1|5 6 4 2

1|5 6 4 2 -> 1 5|6 4 2

1 5|6 4 2 -> 1 5 6|4 2

1 5 6|4 2 -> 1 4 5 6|2

1 4 5 6|2 -> 1 2 4 5 6|

As such the sorted array and unsorted array partition moves forward as elements are added and array gets sorted.

Quick Sort is a O(nlogn) sorting algorithm. It is a recursive algorithm which divides the given array into 2 parts and then sorts the individual arrays with the same quick sort algorithm. The division of array into 2 happens by selecting a pivot element (generally first element). All the elements of the array are divided into less than pivot and greater than pivot and placed accordingly. The left array and right array are again applied with quick sort algorithm.

Example:

Let’s take the following array

1 5 6 4 2

After successive iterations the array will change as follows. |a| represents that a is the pivot element and left part and right part are arranged. () represents that after pivot divided leftover arrays.

1 5 6 4 2

|1|5 6 4 2

1(4 2|5|6)

1((2|4|)5(|6|))

1(((2)4)5(6))

1(((|2|)4)5(6))

1(((2)4)5(6))

The array is finally sorted.

Merge Sort is a O(nlogn) sorting algorithm. It works in a divide and conquer manner. It divides the array into 2 halves and applies the same for the divided arrays and so on. After reaching 1 element the arrays are sorted then they are merged in the order they were divided such that the merged array is also sorted. As such all sorted arrays after merging make the final array a sorted array.

Example:

Let’s take an example as follows

1 5 6 4 2

The merge sort algorithm on this array looks like this

[1 5 6 4 2]

/ \

[1 5] [6 4 2]

/ \ / \

[1] [5] [6] [4 2]

/ \

[4] [2]

← Split complete, now start merging →

[1] [5] [6] [4] [2]

\ / \ /

[1 5] [2 4]

| /

| [2 4 6]

| /

[1 5] [2 4 6]

\ /

[1 2 4 5 6] ← Final Sorted Array

**Code:**

package week1.module2;

import java.util.ArrayList;

class Order implements Comparable<Order>{

private String orderId;

private String customerName;

private double totalPrice;

public Order(String orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

// Getter methods

public String getOrderId() {

return orderId;

}

public String getCustomerName() {

return customerName;

}

public double getTotalPrice() {

return totalPrice;

}

// Setter methods

public void setOrderId(String orderId) {

this.orderId = orderId;

}

public void setCustomerName(String customerName) {

this.customerName = customerName;

}

public void setTotalPrice(double totalPrice) {

this.totalPrice = totalPrice;

}

// To String method to print order details

*@Override*

public String toString() {

return "Order{" + "orderId='" + orderId + "'" + ", customerName='" + customerName + "'" + ", totalPrice=" + totalPrice + "}";

}

public int compareTo(Order order) {

if(this.totalPrice>order.totalPrice) {

return 1;

}

else if(this.totalPrice==order.totalPrice) {

return 0;

}

else {

return -1;

}

}

}

interface SortOrders{

void sort();

}

class BubbleSortOrders implements SortOrders {

ArrayList<Order> list;

public BubbleSortOrders() {

list = new ArrayList<Order>();

}

public void addOrder(Order order) {

list.add(order);

}

*@Override*

public void sort() {

int n = list.size();

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

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

if (list.get(j).getTotalPrice() > list.get(j + 1).getTotalPrice()) {

// Swap

Order temp = list.get(j);

list.set(j, list.get(j + 1));

list.set(j + 1, temp);

}

}

}

}

public void displayOrders() {

for (Order o : list) {

System.***out***.println(o);

}

}

}

// Quick sort implementation

class QuickSortOrders implements SortOrders {

ArrayList<Order> list;

public QuickSortOrders() {

list = new ArrayList<Order>();

}

public void addOrder(Order order) {

list.add(order);

}

*@Override*

public void sort() {

quickSort(0, list.size() - 1);

}

private void quickSort(int low, int high) {

if (low < high) {

int pi = partition(low, high);

quickSort(low, pi - 1);

quickSort(pi + 1, high);

}

}

private int partition(int low, int high) {

double pivot = list.get(high).getTotalPrice();

int i = low - 1;

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

if (list.get(j).getTotalPrice() < pivot) {

i++;

Order temp = list.get(i);

list.set(i, list.get(j));

list.set(j, temp);

}

}

Order temp = list.get(i + 1);

list.set(i + 1, list.get(high));

list.set(high, temp);

return i + 1;

}

public void displayOrders() {

for (Order o : list) {

System.***out***.println(o);

}

}

}

public class Ex3SortingCustomerOrders {

public static void main(String args[]) {

// Sample orders

Order o1 = new Order("O101", "Ram", 4200);

Order o2 = new Order("O102", "Raj", 3100);

Order o3 = new Order("O103", "Praneeth", 7800);

Order o4 = new Order("O104", "Akshay", 2500);

// --------- Bubble Sort ----------

BubbleSortOrders bubbleSortOrders = new BubbleSortOrders();

bubbleSortOrders.addOrder(o1);

bubbleSortOrders.addOrder(o2);

bubbleSortOrders.addOrder(o3);

bubbleSortOrders.addOrder(o4);

System.***out***.println("Before Bubble Sort:");

bubbleSortOrders.displayOrders();

bubbleSortOrders.sort();

System.***out***.println("\nAfter Bubble Sort (by totalPrice):");

bubbleSortOrders.displayOrders();

// --------- Quick Sort ----------

QuickSortOrders quickSortOrders = new QuickSortOrders();

quickSortOrders.addOrder(o1);

quickSortOrders.addOrder(o2);

quickSortOrders.addOrder(o3);

quickSortOrders.addOrder(o4);

System.***out***.println("\nBefore Quick Sort:");

quickSortOrders.displayOrders();

quickSortOrders.sort();

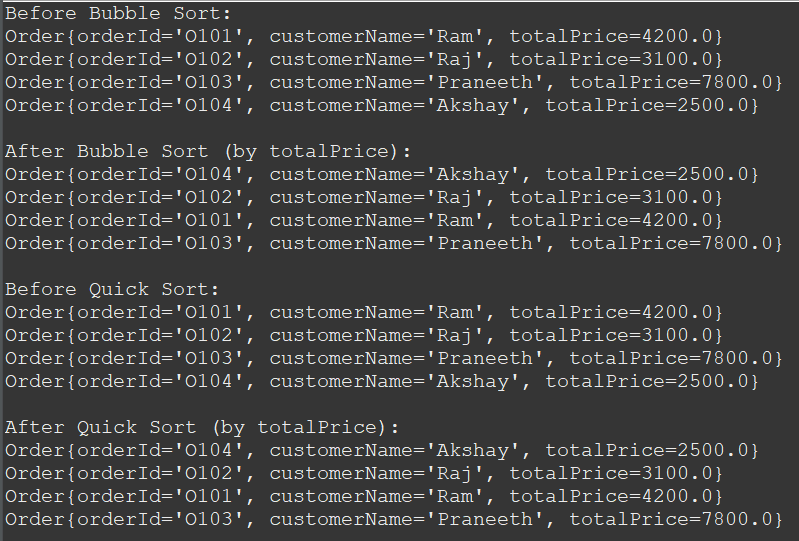
System.***out***.println("\nAfter Quick Sort (by totalPrice):");

quickSortOrders.displayOrders();

}

}

**Output:**



**Step-4 Analysis:**

* + Compare the performance (time complexity) of Bubble Sort and Quick Sort.

Bubble Sort has time complexity O(n2) and Quick Sort has time complexity O(n2). The worst-case complexities of quick sort and bubble sort may be same but the other time complexities are not. For Bubble Sort best case and average case time complexities are O(n2) and for Quick Sort they are O(nlogn). As such only in 1 worst case quick sort works as same as bubble sort. For the rest of the cases, it works better than bubble sort. So, Quick Sort is better than Bubble Sort.

* + Discuss why Quick Sort is generally preferred over Bubble Sort.

Quick Sort has same worst-case time complexity as bubble sort but the best and average case time complexities of quick sort are O(nlogn) which is better than O(n2) of bubble sort. There is only 1 worst case for quick sort in which it yields O(n2) time complexity. As such, that’s why Quick Sort is generally preferred over Bubble Sort.

**Exercise 4: Employee Management System**

**Step-1 Understand Array Representation:**

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

Arrays are represented in memory as continuous blocks of memory locations where each element is stored one after the other. For an example array, it will have following memory locations, assuming that first element address is 1000.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Element | 100 | 200 | 300 | 400 | 500 |
| Memory Address | 1000 | 1004 | 1008 | 1012 | 1016 |

We can access the next element by adding the data type byte size to previous address. In our example, int had 4 bytes. So, if the first element has 1000 address then the second element address will be 1000+1\*4, third element address will be 1000+2\*4 and so on.

This gives many advantages. We can utilize random access with index of the element. As elements are store continuously, modern computers can use caching for fast access. They are simple to understand and are perfect for fixed size collections. They don’t store excess metadata, so there is little overhead. They can be easily traversed with loops using indexes avoiding pointers. These are preferred in critical low-level applications.

**Code:**

package week1.module2;

class Employee {

int employeeId;

String name;

String position;

double salary;

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

this.employeeId=employeeId;

this.name=name;

this.position=position;

this.salary=salary;

}

public void display() {

System.***out***.println("ID: "+employeeId+", Name: "+name+", Position: "+position+", Salary: "+salary);

}

}

class EmployeeManager {

private Employee[] employees;

private int count;

public EmployeeManager(int size) {

employees=new Employee[size];

count=0;

}

// Add employee

public void addEmployee(Employee e) {

if(count<employees.length) {

employees[count++]=e;

System.***out***.println("Employee added.");

}

else {

System.***out***.println("Array full. Cannot add more employees.");

}

}

// Search employee by ID

public Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

// Traverse all employees

public void traverseEmployees() {

if(count==0) {

System.***out***.println("No employees to display.");

return;

}

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

employees[i].display();

}

}

// Delete employee by ID

public void deleteEmployee(int id) {

boolean found=false;

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

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

found=true;

// Shift elements to left

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

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

}

employees[--count]=null;

System.***out***.println("Employee deleted.");

break;

}

}

if(!found) {

System.***out***.println("Employee ID not found.");

}

}

}

public class Ex4EmployeeArray {

public static void main(String[] args) {

EmployeeManager manager=new EmployeeManager(5);

// Adding employees

manager.addEmployee(new Employee(1, "Alice", "Manager", 75000));

manager.addEmployee(new Employee(2, "Bob", "Developer", 55000));

manager.addEmployee(new Employee(3, "Charlie", "Analyst", 50000));

// Traversing

System.***out***.println("\nAll Employees:");

manager.traverseEmployees();

// Searching

System.***out***.println("\nSearching for Employee with ID 2:");

Employee emp = manager.searchEmployee(2);

if(emp != null) {

emp.display();

}

else {

System.***out***.println("Employee not found.");

}

// Deleting

System.***out***.println("\nDeleting Employee with ID 2:");

manager.deleteEmployee(2);

// Traversing again

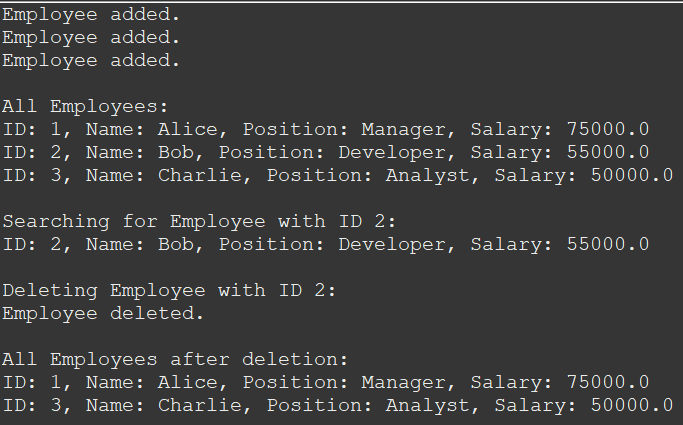
System.***out***.println("\nAll Employees after deletion:");

manager.traverseEmployees();

}

}

**Output:**



**Step-4 Analysis:**

* Analyze the time complexity of each operation (add, search, traverse, delete).

Add operation takes O(1) complexity as it is directly added at the end of the array. There is no traversal included here, that’s why O(1) time complexity. If the array has become full, we can’t add another element.

Seach operation takes O(n) time complexity as we have to traverse through entire array sequentially to find required element. We used Linear Search here to find an element of the array. If the array is sorted, we can also use Binary Search.

Traverse operation takes O(n) time complexity as we have to traverse trough each element and print it to console. We display each employee object details using toString() method.

Delete operation takes O(n) time complexity. IN delete operation first we search for the element required and then shift all elements to the right to the left. As we are accessing each element, this operation has O(n) time complexity.

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

Arrays are used only when we have a fixed size for the collection. Arrays cannot go beyond the size allocated. We can re-allocate extra space only if there is empty memory after the array memory which cannot be valid in every case. So, to overcome the limitation of fixed size we use List collections. These List Collections can be dynamically resized. Arrays can only be used when we need fixed size and cannot be employed in cases where data size dynamically changes over time.

**Exercise 5: Task Management System**

**Step-1 Understand Linked Lists:**

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

Linked List is a type of data structure where elements are randomly placed in entire memory and are accessed by memory addresses. First element address is stored in variable, second element address in first element, third element address in second address and so on.

Singly Linked List is a linked list in which we can traverse only in forward direction. Each element consists of data and only 1 address which is of next element. Thus, we cannot traverse in the other direction at all.

Doubly Linked List is a linked list in which we can traverse in both forward and backward directions. Each element consists of data and 2 addresses – for previous element and next element. This allows us to traverse in both directions offering flexibility in operations than Singly Linked List.

**Code:**

package week1.module2;

class Task {

int taskId;

String taskName;

String status;

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

this.taskId=taskId;

this.taskName=taskName;

this.status=status;

}

public String toString() {

return "Task ID: "+taskId+", Name: "+taskName+", Status: "+status;

}

}

class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task=task;

this.next=null;

}

}

class TaskLinkedList {

private TaskNode head;

// Add task to the end of the list

public void addTask(Task task) {

TaskNode newNode=new TaskNode(task);

if(head==null) {

head=newNode;

}

else{

TaskNode temp=head;

while(temp.next!=null) {

temp=temp.next;

}

temp.next=newNode;

}

System.***out***.println("Task added.");

}

// Search task by taskId

public Task searchTask(int taskId) {

TaskNode temp=head;

while(temp!=null) {

if(temp.task.taskId==taskId) {

return temp.task;

}

temp=temp.next;

}

return null;

}

// Traverse and display all tasks

public void traverseTasks() {

if(head==null) {

System.***out***.println("No tasks available.");

return;

}

TaskNode temp = head;

while(temp!=null) {

System.***out***.println(temp.task);

temp=temp.next;

}

}

// Delete task by taskId

public void deleteTask(int taskId) {

if(head==null) {

System.***out***.println("Task list is empty.");

return;

}

if (head.task.taskId==taskId) {

head=head.next;

System.***out***.println("Task deleted.");

return;

}

TaskNode current=head;

while(current.next!=null&&current.next.task.taskId!=taskId) {

current=current.next;

}

if(current.next==null) {

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

}

else {

current.next=current.next.next;

System.***out***.println("Task deleted.");

}

}

}

public class Ex5TaskManagement {

public static void main(String[] args) {

TaskLinkedList taskList=new TaskLinkedList();

// Adding tasks

taskList.addTask(new Task(1, "Design Database", "Pending"));

taskList.addTask(new Task(2, "Write API", "In Progress"));

taskList.addTask(new Task(3, "Test Application", "Completed"));

// Traversing

System.***out***.println("\nAll Tasks:");

taskList.traverseTasks();

// Searching

System.***out***.println("\nSearching for Task ID 2:");

Task task = taskList.searchTask(2);

if (task != null) {

System.***out***.println(task);

}

else {

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

}

// Deleting

System.***out***.println("\nDeleting Task ID 2:");

taskList.deleteTask(2);

// Traversing after deletion

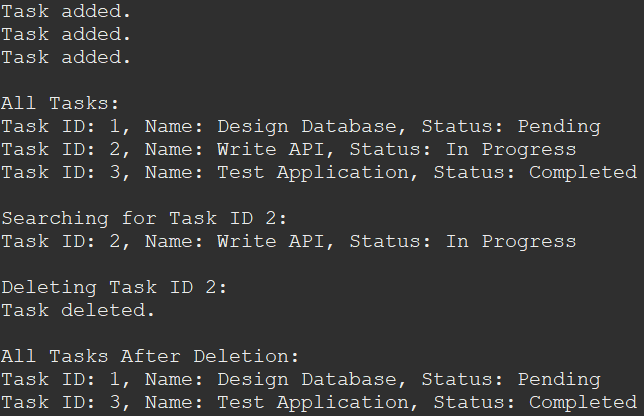
System.***out***.println("\nAll Tasks After Deletion:");

taskList.traverseTasks();

}

}

**Output:**



**Step-4 Analysis:**

* Analyze the time complexity of each operation.

Add Operation takes O(n) time complexity as we have to traverse through all elements to get address of the last node which we need to add a new element. Search operation also takes O(n) complexity as we have to traverse through all elements to find the element required. We cannot even apply Binary Search here as we need random access which is not present in Linked List. Traverse Operation takes O(n) time complexity as we have to traverse across each element individually. Delete operation takes O(n) time as we have to search which element to delete. As such each of the 4 operations take O(n) time complexity.

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

Arrays are fixed in size. Thus, they are not suitable for most applications which require dynamically changing storage. This is overcome by Linked List. As they can store nodes anywhere in the memory, linked lists size can be changed dynamically which is prefect for applications which require dynamic storage. In Arrays after deleting the element we have to shift all the next elements one position left, but in case of linked list we need not do this as we can just change the links between nodes. Hence, it is an optimization.

**Exercise 6: Library Management System**

**Step-1 Understand Search Algorithms:**

* Explain linear search and binary search algorithms.

Linear Search has time complexity O(n). It works in a sequential manner. It goes through each element from the first element and matches each element with search element. If an element matches, the search stops and returns the search element.

Binary Search is optimized than Linear Search. It has O(nlogn) time complexity. It uses divide and conquer approach. For this algorithm to work, the array should be sorted. To search, we first select the middle element. If the middle element is smaller than search element, then we discard left subarray as all elements in it will always be less than search element. As such if the mid element is larger then search element, we discard right subarray form mid element. As such we discard subarrays continuously and reach the search element.

**Code:**

package week1.module2;

import java.util.Arrays;

class Book implements Comparable<Book> {

int bookId;

String title;

String author;

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

this.bookId=bookId;

this.title=title;

this.author=author;

}

public String toString() {

return "Book ID: "+bookId+", Title: "+title+", Author: "+author;

}

*@Override*

public int compareTo(Book other) {

return this.title.compareToIgnoreCase(other.title);

}

}

class BookSearch {

// Linear Search

public static Book linearSearch(Book[] books, String title) {

for(Book book:books) {

if(book.title.equalsIgnoreCase(title)) {

return book;

}

}

return null;

}

// Binary Search (Array is sorted by title)

public static Book binarySearch(Book[] books, String title) {

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

while (low<=high) {

mid=low+(high-low)/2;

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

if(cmp==0) {

return books[mid];

}

else if(cmp<0) {

low=mid+1;

}

else {

high=mid-1;

}

}

return null;

}

}

public class Ex6LibraryManagementSystem {

public static void main(String[] args) {

Book[] books=new Book[5];

books[0]=new Book(101, "Java Basics", "James");

books[1]=new Book(102, "Data Structures", "Mark");

books[2]=new Book(103, "Algorithms", "Nina");

books[3]=new Book(104, "Operating Systems", "Alex");

books[4]=new Book(105, "Networks", "Charlie");

// Linear Search (unsorted)

System.***out***.println("Linear Search for 'Algorithms':");

Book result1 = BookSearch.*linearSearch*(books, "Algorithms");

if (result1 != null) {

System.***out***.println(result1);

}

else System.***out***.println("Book not found.");

// Sort the array before binary search

Arrays.*sort*(books);

// Binary Search (sorted)

System.***out***.println("\nBinary Search for 'Networks':");

Book result2 = BookSearch.*binarySearch*(books, "Networks");

if (result2 != null) {

System.***out***.println(result2);

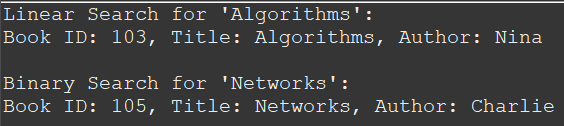
}

else System.***out***.println("Book not found.");

}

}

**Output:**

****

**Step-4 Analysis:**

* Compare the time complexity of linear and binary search.

The best case in both algorithms is O(1). The average and worst case for Linear Search is O(n) and for Binary Search is O(logn). From the time complexities, we can see that Binary Search is a better algorithm than Linear Search but the problem is that for Binary Search Algorithm to work we need a sorted array. So, to create or maintain a sorted array we may need more time or other algorithms. But in case of large data this overhead and Binary Search growth becomes negligible compared to Linear Search Algorithm’s growth in time taken. Thus, Binary Search is more efficient than Linear Search in case of large data. If our application contains small data enough to neglect this change, then we can choose Linear Search itself for simplicity and efficiency.

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

For Binary Search Algorithm to work, we need a sorted array. So, to create or maintain a sorted array we may need more time or other algorithms. But in case of large data this overhead and Binary Search growth becomes negligible compared to Linear Search Algorithm’s growth, in time taken. Thus, Binary Search is more efficient than Linear Search in case of large data. If our application contains small data enough to neglect this change, then we can choose Linear Search itself for simplicity and efficiency.

**Exercise 7: Financial Forecasting**

**Step-1 Understand Recursive Algorithms:**

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

Recursion is a concept in which a function call itself inside the function body. It is useful in cases where the result of a certain function depends on the previous or smaller parameter results of the same function. For example, Fibonacci series is a recursion problem where it’s recursion equation can be written as

Fib(n)=Fib(n-1)+Fib(n-2)

As nth term depends on n-1th and n-2th term, it is a recursion problem.

Recursion allows us to solve a problem where same process has to be repeated again and again and next result depends on previous results. Thus, it simplifies many problems.

**Code:**

package week1.module2;

public class Ex7FinancialForecasting {

// Recursive function to calculate future value

public static double futureValue(double pv, double rate, int years) {

// Base case

if (years==0) {

return pv;

}

// Recursive case

return *futureValue*(pv, rate, years-1)\*(1+rate);

}

public static void main(String[] args) {

double pv=10000; // Initial investment

double rate=0.08; // Annual growth rate (8%)

int years=5; // Forecast for 5 years

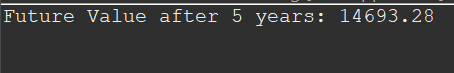
double fv=*futureValue*(pv, rate, years);

System.***out***.printf("Future Value after %d years: %.2f%n", years, fv);

}

}

**Output:**

****

**Step-4 Analysis:**

* Discuss the time complexity of your recursive algorithm.

The time complexity of this algorithm is O(n). It recursively calculates principal value for next year, so that we can calculate gains on the following year and so on. Here n is “years”.

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

There may be repetitive computations which increase time. Recursion calls have more overhead than iterations. We can choose to optimize recursion itself by using memorization or shift to iterations. This allows us to avoid excessive computation.