**Exercise 1: Inventory Management System**

1. **Understanding Data Structures**
2. we need data structures and algorithms to store and manage a large number of products efficiently. They help in fast searching, adding, updating, and deleting items. Without proper DS, system will be slow and messy.
3. Suitable Data Structures:

* ArrayList : Good for small data, but slow for searching/updating by ID.
* HashMap : Fast in lookup, update, and delete.

So I’ll go with HashMap.

**Code:**

import java.util.\*;

class Product

{

HashMap<Integer,String> items=new HashMap<Integer,String>();

Scanner s= new Scanner(System.in);

public void insertion()

{

int pid;

String pname;

System.out.print("Enter the Product Id:");

pid=s.nextInt();

s.nextLine();

if(items.containsKey(pid))

System.out.println("Entered the Already Existing Id!!!");

else

{

System.out.print("Enter the Product Name:");

pname=s.nextLine();

items.put(pid,pname);

System.out.println("Succesfully Stored the Product!!!");

}

}

public void deletion()

{

int pid;

System.out.print("Enter the Product ID:");

pid=s.nextInt();

s.nextLine();

if(items.containsKey(pid))

{

items.remove(pid);

System.out.println("Successfully Deleted the Product!!!");

}

else

{

System.out.println("Entered a Product Id Which is not Present!!!");

}

}

public void displaying()

{

if(items.size()==0)

{

System.out.println("The Warehouse is Empty with no Products!!!");

}

else

{

System.out.print("Produts present are:");

System.out.println(items);

}

}

public void updating()

{

int pid;

System.out.print("Enter the Product ID:");

pid=s.nextInt();

s.nextLine();

System.out.print("Enter the Product Name that is to be updated:");

String n=s.nextLine();

items.put(pid,n);

System.out.println("Succesfully Updated the Product!!!");

}

}

class Inventory

{

Inventory()

{

System.out.println("This is an Inventory Warehouse System!!");

Scanner s= new Scanner(System.in);

Product p=new Product();

while(true)

{

System.out.println("Enter 1 for Adding the New Product");

System.out.println("Enter 2 for Deleting the Product");

System.out.println("Enter 3 for Updating the Product");

System.out.println("Enter 4 for Displaying all the Products in the Warehouse");

System.out.println("Enter 5 for stopping the System");

System.out.print("Enter Your Choice:");

int k=s.nextInt();

switch(k)

{

case 1:p.insertion();

break;

case 2:p.deletion();

break;

case 3:p.updating();

break;

case 4:p.displaying();

break;

case 5:System.exit(0);

default:System.out.println("Please Enter Valid Choice!!!");

}

}

}

}

public class Main {

public static void main(String[] args)

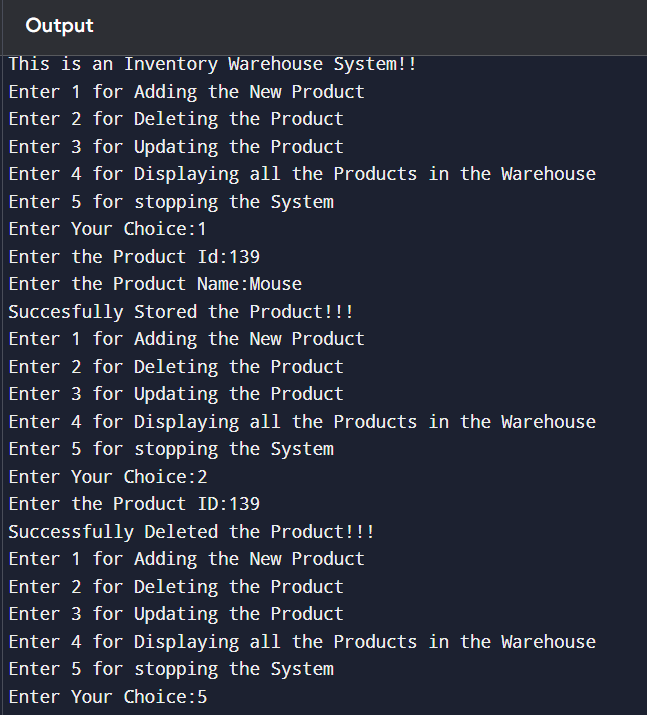
{

Inventory i=new Inventory();

}

}

**Output:**

****

**(iv) Analysis**

* Time Complexities :
  + Add: O(1)
  + Update: O(1)
  + Delete: O(1)

**Optimizations:**

1)Avoid Duplicate IDs : Before adding, check if the productId already exists. This avoids data issues.

2)If implemented in ArrayList recommended to use HashMap since it is much faster than using ArrayList where you search linearly (O(n)).

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

**(i)Big O Notation:**

Big O notation is used to describe the time complexity or space complexity of an algorithm in the worst-case scenario. It provides an upper bound on the growth rate of an algorithm as the input size increases. It helps in analyzing the scalability and efficiency of algorithms independent of machine or compiler variations.

* For example, if an algorithm has a time complexity of O(n), it means the time taken grows linearly with the size of input n.
* If it’s O(log n), the time grows logarithmically, which is faster.

Best, Average, and Worst Case:

* Best Case: Minimum time taken by the algorithm for the smallest input scenario. E.g., in linear search, if the element is the first, time is O(1).
* Average Case: Expected time taken, assuming the input is random. For linear search, this would be O(n/2) → simplified to O(n).
* Worst Case: Maximum time taken by the algorithm. In linear search, if the element is not present or is the last, time is O(n).

**Code:**

import java.util.\*;

class Shop

{

int[] pid;

String[] pname;

String[] pcategory;

int[] spid;

String[] spname;

String[] scategory;

int l;

Scanner s=new Scanner(System.in);

public void read()

{

System.out.print("Enter the Number of Products:");

l=s.nextInt();

pid=new int[l];

spid=new int[l];

pname=new String[l];

spname=new String[l];

pcategory=new String[l];

scategory=new String[l];

System.out.println("Enter the Details of the Products!!!");

for(int x=0;x<l;x++)

{

int j=x+1;

System.out.print("Enter the Item-"+j+" Product ID:");

pid[x]=s.nextInt();

s.nextLine();

System.out.print("Enter the Item-"+j+" Product Name:");

pname[x]=s.nextLine();

System.out.print("Enter the Item-"+j+" Product Category:");

pcategory[x]=s.nextLine();

spid=pid;

spname=pname;

scategory=pcategory;

}

}

public void linearsearch(int id)

{

int flag=0;

for(int x=0;x<l;x++)

{

if(pid[x]==id)

{

System.out.println("The Product has Found!!!");

System.out.println("Product Id is:"+pid[x]+" and Name is:"+pname[x]+" and Category is:"+pcategory[x]);

flag=1;

break;

}

}

if(flag==0)

System.out.println("The Product has Not Found!!!");

}

public void sort()

{

String t2;

int t1;

for(int x=0;x<l;x++)

{

for(int y=0;y<l;y++)

{

if(spid[x]<spid[y])

{

t1=spid[x];

spid[x]=spid[y];

spid[y]=t1;

t2=spname[x];

spname[x]=spname[y];

spname[y]=t2;

t2=scategory[x];

scategory[x]=scategory[y];

scategory[y]=t2;

}

}

}

}

public void binarysearch(int id)

{

int mid,flag=0;

int le=0;

int ri=l-1;

while(le<=ri)

{

mid=(le+ri)/2;

if(spid[mid]==id)

{

System.out.println("The Product has Found!!!");

System.out.println("Product Id is:"+spid[mid]+" and Name is:"+spname[mid]+" and Category is:"+scategory[mid]);

flag=1;

break;

}

else if(spid[mid]<id)

le=mid+1;

else

ri=mid-1;

}

if(flag==0)

{

System.out.println("The Product has Not Found!!!");

}

}

}

public class Main {

public static void main(String[] args)

{

Shop i=new Shop();

i.read();

i.sort();

int id;

Scanner s=new Scanner(System.in);

while(true)

{

System.out.println("Enter 1 to Search using Linear Search");

System.out.println("Enter 2 to Search using Binary Search");

System.out.print("Enter the Choice:");

int k=s.nextInt();

s.nextLine();

switch(k)

{

case 1:System.out.print("Enter the Product Id for Search using Linear Search:");

id=s.nextInt();

s.nextLine();

i.linearsearch(id);

break;

case 2:System.out.print("Enter the Product Id for Search using Binary Search:");

id=s.nextInt();

s.nextLine();

i.binarysearch(id);

break;

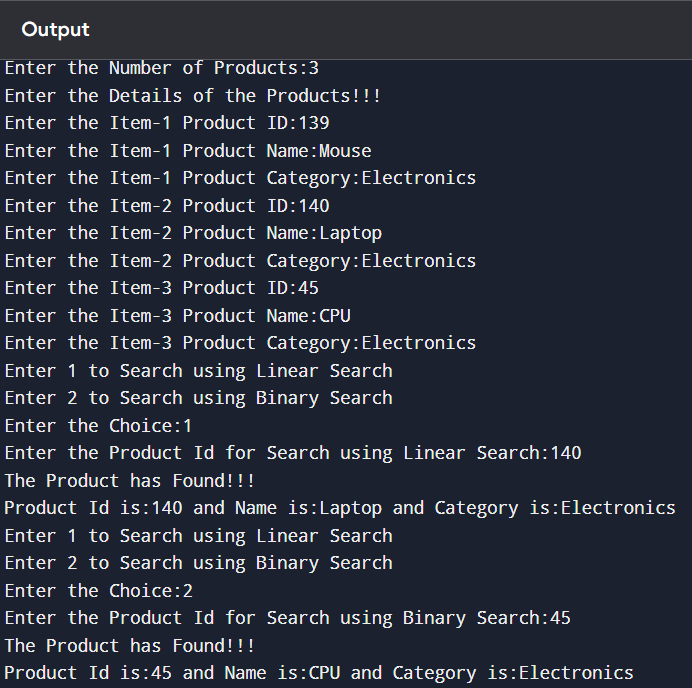
}

}

}

}

**Output:**

****

**(iv)** **Analysis :**

Time Complexity Comparison:

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** |
| --- | --- | --- | --- | --- |
| **Linear Search** | O(1) | O(n) | O(n) | O(1) |
| **Binary Search** | O(1) | O(log n) | O(log n) | O(1) |

**Which Algorithm is More Suitable?**

* **Linear Search** is simple and doesn’t require sorted data. It is ideal for small datasets or unsorted data. However, it's inefficient for large lists.
* **Binary Search** is **much faster** (logarithmic time) for large datasets but requires data to be sorted. Sorting takes O(n log n) time, but if we’re frequently searching and infrequently updating the list, sorting once is worth it.

**Exercise 3: Sorting Customer Orders**

**Code:**

**(i)Understanding Sorting Techniques :**

* **Bubble Sort:**
  + Compares two elements and swaps them if needed.
  + Repeats till everyting is sorted.
  + **Slow** for big data (O(n²))
  + Easy to write.
* **Insertion Sort:**
  + Picks elements one by one and puts them in the right place.
  + Good for small datasets.
  + Also O(n²).
* **Quick Sort:**
  + Picks a pivot, splits data into smaller and bigger parts, and sorts recursively.
  + Very fast for big data.
  + Avg time = **O(n log n)**
* **Merge Sort:**
  + Splits data in half, sorts both halves, and merges them.
  + Time = O(n log n), but uses more memory.

**(ii) SetUp :**

public class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

}

**(iii) Implementation :**

1. Bubble Sort

public void bubbleSort(Order[] orders) {

int n = orders.length;

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

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

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

// Swap

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

1. Quick Sort

public void quickSort(Order[] orders, int low, int high) {

if (low < high) {

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

quickSort(orders, low, pi - 1);

quickSort(orders, pi + 1, high);

}

}

public int partition(Order[] orders, int low, int high) {

double pivot = orders[high].totalPrice;

int i = low - 1;

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

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

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

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

orders[high] = temp;

return i + 1;

}

**(iv) Analysis**

* **Bubble Sort:**
  + Time Complexity: **O(n²)**
  + Bad for large data. Too many comparisons.
* **Quick Sort:**
  + Avg Time: **O(n log n)**
  + Fast and efficient.

**Why Quick Sort is better than Bubble Sort?**

1. Bubble Sort is simple, but **too slow**.
2. Quick Sort is much **faster**, especially with large orders.
3. That’s why we prefer Quick Sort in real-world problems.

**Exercise 4: Employee Management System**

**(i) Understanding Array Representation**

* **How arrays are stored:**  
  Arrays are stored in **continuous memory** blocks. Each element is placed next to the other.
* **Advantages:**
  + Fast access using index (O(1)).
  + Simple to use.
  + Good for fixed-size data.

**(ii) SetUp:**

public 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;

}

}

**(iii) Implementation :**

public class EmployeeSystem {

Employee[] employees = new Employee[100];

int count = 0;

public void addEmployee(Employee e) {

if (count < employees.length) {

employees[count] = e;

count++;

}

}

public Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

public void displayEmployees() {

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

System.out.println(employees[i].name + " - " + employees[i].position);

}

}

public void deleteEmployee(int id) {

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

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

// shift elements left

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

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

}

employees[count - 1] = null;

count--;

break;

}

}

}

}

**(iv) Analysis**

* **Add:** O(1) → adding at end
* **Search:** O(n) → need to check one by one
* **Traverse:** O(n)
* **Delete:** O(n) → shifting needed

**Limitations of Arrays**

* Fixed size (can't grow automatically).
* Insertion/deletion in middle is slow.
* Not good for big dynamic data.

**When to Use Arrays**

* When size is known and fixed.
* When you need fast index access.
* For small/medium size datasets.

**Exercise 5: Task Management System**

**Scenario:**

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

**(i) Understanding Linked Lists**

* **Singly Linked List:**  
  Each node has data + next pointer.  
  Can go forward only.
* **Doubly Linked List:**  
  Each node has data + next + prev pointer.  
  Can go forward and backward.  
  Takes more memory but more flexible.

**(ii) SetUp**

public 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;

}

}

**(ii) Implementation**

class Node {

Task task;

Node next;

public Node(Task task) {

this.task = task;

this.next = null;

}

}

public class TaskManager {

Node head = null;

public void addTask(Task task) {

Node newNode = new Node(task);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

public Task searchTask(int id) {

Node temp = head;

while (temp != null) {

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

return temp.task;

}

temp = temp.next;

}

return null;

}

public void displayTasks() {

Node temp = head;

while (temp != null) {

System.out.println(temp.task.taskName + " - " + temp.task.status);

temp = temp.next;

}

}

public void deleteTask(int id) {

if (head == null) return;

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

head = head.next;

return;

}

Node temp = head;

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

temp = temp.next;

}

if (temp.next != null) {

temp.next = temp.next.next;

}

}

}

**(iv) Analysis**

* **Add:** O(n) – adding at end
* **Search:** O(n) – need to traverse
* **Traverse:** O(n)
* **Delete:** O(n) – find node, then unlink

**Advantages of Linked List over Array**

* Dynamic size – can grow/shrink as needed
* Easy insertion/deletion – no shifting needed
* Memory efficient, if size keeps changing
* No fixed limit 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.

**(i) Understanding Search Algorithms**

* **Linear Search:**
  + Go through the list one by one.
  + Works on unsorted data.
  + Simple, but slow.
  + Time: O(n)
* **Binary Search:**
  + Works on sorted list only.
  + Divide and search (like cutting in half).
  + Much faster.
  + Time: O(log n)

**(ii) SetUp**

public class Book {

int bookId;

String title;

String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

}

**(iii) Implementation**

* 1. Linear Search

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

for (Book book : books) {

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

return book;

}

}

return null;

}

* 1. Binary Search

import java.util.Arrays;

import java.util.Comparator;

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

int left = 0;

int right = books.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) {

return books[mid];

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

// Sorting the array before binary search

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

**(iv) Analysis**

* **Linear Search:**
  + Time: O(n)
  + Works on any list (sorted or not).
  + Good for small data.
* **Binary Search:**
  + Time: O(log n)
  + Needs sorted data.
  + Super fast for large lists.

**When to Use What?**

* Use **linear search** if:
  + List is small
  + Or unsorted
  + Or only occasional search
* Use **binary search** if:
  + List is big and sorted
  + You search a lot
  + Want better speed

**Exercise 7: Financial Forecasting**

**(i) Recursion**

Recursion is used in this program to:

* Calculate the **mean** of values in an array (mean)
* Compute the **numerator** and **denominator** for the linear regression formula
* Avoid loops and express the logic in a more functional, repetitive structure

**Example – Recursive Mean Calculation:**

java

CopyEdit

public double mean(int g[], int index) {

if (index == l) return 0;

else return g[index] + mean(g, index + 1);

}

This computes the sum recursively. The actual mean is calculated by dividing this sum by the total number of elements l.

**(ii) Setup**

This project forecasts electricity bills for future months based on past bill data using **simple linear regression**.

**Regression Formula:**

csharp

CopyEdit

Predicted Value (y) = b0 + b1 \* x

Where:

- x is the input (month number),

- y is the output (bill amount),

- b1 = slope = Σ((xi - x̄) \* (yi - ȳ)) / Σ((xi - x̄)^2)

- b0 = intercept = ȳ - b1 \* x̄

The program reads past electricity bills month-wise and predicts the bill for any future month using this regression formula.

**Code:**

import java.util.\*;

class Bill

{

int[] amounts;

int[] months;

Scanner s=new Scanner(System.in);

int l;

public void read()

{

System.out.println("Enter the Data Present!!!");

System.out.print("Enter the Number of Months data Available:");

l=s.nextInt();

s.nextLine();

amounts=new int[l];

months=new int[l];

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

{

System.out.print("Enter the Month Number:");

months[i]=s.nextInt();

s.nextLine();

System.out.print("Enter the Bill Amount for the Month:");

amounts[i]=s.nextInt();

s.nextLine();

}

}

public double mean(int g[],int index)

{

if(index==l)

return 0;

else

return g[index]+mean(g,index+1);

}

public double num(int g[],int h[],int index,double xmean,double ymean)

{

if(index==l)

return 0;

else

return (g[index]-xmean)\*(h[index]-ymean)+num(g,h,index+1,xmean,ymean);

}

public double din(int g[],int index,double mean)

{

if(index==l)

return 0;

else

return ((g[index]-mean)\*(g[index]-mean))+din(g,index+1,mean);

}

public double predict(int n)

{

double xmean=mean(months,0)/l;

double ymean=mean(amounts,0)/l;

double b1=num(months,amounts,0,xmean,ymean)/din(months,0,xmean);

double b0=ymean-(b1\*xmean);

if(b0+(b1\*n)<150) //If Predicted AMount is Less than the Minimum Charge(150) then return 150

return 150;

return b0+(b1\*n);

}

}

public class Main {

public static void main(String[] args) {

Bill b=new Bill();

b.read();

Scanner s=new Scanner(System.in);

System.out.print("Enter the Month Number to Predict Bill Amount:");

int n=s.nextInt();

s.nextLine();

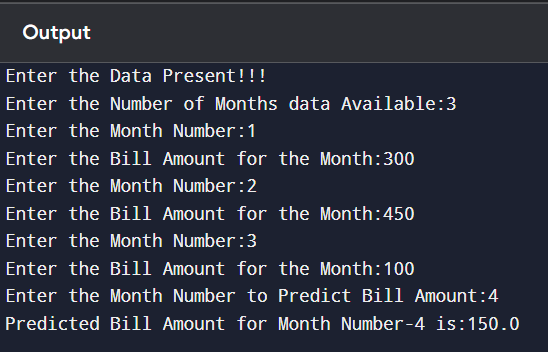
System.out.print("Predicted Bill Amount for Month Number-"+n+" is:");

System.out.println(b.predict(n));

}

}

**Output:**

****