**Exercise 1: Inventory Management System**

1. **Understand the Problem:**

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

* + As the size of the inventory grows, the efficiency of operations (like add, update, delete, and search) becomes critical. Efficient data structures and algorithms ensure that these operations can be performed quickly even as the inventory size increases.
  + Proper data structures help in managing memory usage efficiently.
  + Efficient algorithms reduce the time complexity of operations, leading to faster execution times.

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

* **ArrayList**:
  + Dynamic resizing, allowing the inventory to grow as needed.
  + Fast random access due to index-based access.
  + Disadvantage- Insertion and deletion can be slow (O(n) time complexity) if elements need to be shifted.
* **HashMap**:
  + - Provides average O(1) time complexity for insertion, deletion, and lookup operations.
* Key-value pairing ensures quick access to products using unique product IDs.
* But it does not maintain any order of elements.
* It requires good hashing functions to avoid collisions.

1. **Setup:**
   * Create a new project for the inventory management system.
2. **Implementation:**

**Product.java**

**public** **class** Product {

**private** **int** productId;

**private** String productName;

**private** **int** quantity;

**private** **double** price;

**public** Product(**int** productId, String productName, **int** quantity, **double** price) {

**this**.productId = productId;

**this**.productName = productName;

**this**.quantity = quantity;

**this**.price = price;

}

// Getters and Setters

**public** **int** getProductId() {

**return** productId;

}

**public** **void** setProductId(**int** productId) {

**this**.productId = productId;

}

**public** String getProductName() {

**return** productName;

}

**public** **void** setProductName(String productName) {

**this**.productName = productName;

}

**public** **int** getQuantity() {

**return** quantity;

}

**public** **void** setQuantity(**int** quantity) {

**this**.quantity = quantity;

}

**public** **double** getPrice() {

**return** price;

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

}

}

**Inventory.java**

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.\*;

**public** **class** Inventory {

**private** Map<Integer, Product> products;

**public** Inventory() {

products = **new** HashMap<>();

}

**public** **void** addProduct(Product product) {

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

}

**public** **void** updateProduct(Product product) {

**if** (products.containsKey(product.getProductId())) {

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

} **else** {

System.***out***.println("Product not found!");

}

}

**public** **void** deleteProduct(**int** productId) {

**if** (products.containsKey(productId)) {

products.remove(productId);

} **else** {

System.***out***.println("Product not found!");

}

}

**public** **static** **void** main(String[] args) {

Inventory inventory = **new** Inventory();

Scanner scanner = **new** Scanner(System.***in***);

**while** (**true**) {

System.***out***.println("\nInventory Management System");

System.***out***.println("1. Add Product");

System.***out***.println("2. Update Product");

System.***out***.println("3. Delete Product");

System.***out***.println("4. Exit");

System.***out***.print("Choose an option: ");

**int** choice = scanner.nextInt();

**switch** (choice) {

**case** 1:

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

**int** addId = scanner.nextInt();

scanner.nextLine(); // consume newline

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

String addName = scanner.nextLine();

System.***out***.print("Enter Quantity: ");

**int** addQuantity = scanner.nextInt();

System.***out***.print("Enter Price: ");

**double** addPrice = scanner.nextDouble();

inventory.addProduct(**new** Product(addId, addName, addQuantity, addPrice));

System.***out***.println("Product added successfully.");

**break**;

**case** 2:

System.***out***.print("Enter Product ID to Update: ");

**int** updateId = scanner.nextInt();

scanner.nextLine(); // consume newline

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

String updateName = scanner.nextLine();

System.***out***.print("Enter Updated Quantity: ");

**int** updateQuantity = scanner.nextInt();

System.***out***.print("Enter Updated Price: ");

**double** updatePrice = scanner.nextDouble();

inventory.updateProduct(**new** Product(updateId, updateName, updateQuantity, updatePrice));

System.***out***.println("Product updated successfully.");

**break**;

**case** 3:

System.***out***.print("Enter Product ID to Delete: ");

**int** deleteId = scanner.nextInt();

inventory.deleteProduct(deleteId);

System.***out***.println("Product deleted successfully.");

**break**;

**case** 4:

System.***out***.println("Exiting...");

scanner.close();

**return**;

**default**:

System.***out***.println("Invalid choice. Please try again.");

**break**;

}

}

}

}

**4. Analysis**

**Time Complexity of Each Operation**

1. **Add Product**: O(1) on average because HashMap provides constant time complexity for insertion.
2. **Update Product**: O(1) on average because updating an existing key in a HashMap is also a constant time operation.
3. **Delete Product**: O(1) on average because removing a key-value pair in a HashMap is a constant time operation.

**Optimizing Operations**

* HashMap resizes itself when the number of elements exceeds its capacity, which can be an expensive operation. To mitigate this, it's essential to initialize the HashMap with an appropriate initial capacity if the size of the inventory is known beforehand.

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

1. **Understand Asymptotic Notation:**

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

**Big O notation** is a mathematical notation used to describe the performance characteristics of an algorithm, particularly in terms of time complexity and space complexity. It provides an upper bound on the growth rate of an algorithm's runtime or memory usage as the input size increases. Big O notation abstracts away constants and lower-order terms to focus on the dominant term that affects performance as the input size grows.

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

**1. Linear Search -** Linear Search scans each element of the array or list sequentially to find the target value. It does not require the array to be sorted.

* **Best Case**: The best-case scenario occurs when the target element is the first element of the array. The time complexity is O(1) because the search is completed in a single comparison.
* **Average Case**: In the average case, the target element is somewhere in the middle of the array. On average, it takes about n/2 comparisons to find the target, where n is the number of elements. The time complexity is O(n).
* **Worst Case**: The worst-case scenario occurs when the target element is either not present in the array or is the last element. The algorithm has to scan through all n elements. The time complexity is O(n).

#### 2. ****Binary Search -** Binary Search** requires the array to be sorted. It repeatedly divides the search interval in half, making it efficient for large datasets.

* **Best Case**: The best-case scenario occurs when the target element is exactly in the middle of the array during the first comparison. The time complexity is O(1).
* **Average Case**: In the average case, binary search performs a logarithmic number of comparisons, because each step halves the search space. The time complexity is O(log n), where n is the number of elements.
* **Worst Case**: The worst-case scenario occurs when the target element is not present in the array. Even in this case, binary search still has to divide the array logarithmically, making the time complexity O(log n).

**2.Setup:**

Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.

**Product.java**

**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** productId;

}

**public** String getProductName() {

**return** productName;

}

**public** String getCategory() {

**return** category;

}

@Override

**public** String toString() {

**return** "Product [productId=" + productId + ", productName=" + productName + ", category=" + category + "]";

}

}

**3.Implementation:**

* + Implement linear search and binary search algorithms.
  + Store products in an array for linear search and a sorted array for binary search.

**LinearSearch.java**

**public** **class** LinearSearch {

**public** **static** Product linearSearch(Product[] products, Integer productId, String productName, String category) {

**for** (Product product : products) {

**boolean** matches = **true**;

**if** (productId != **null** && product.getProductId() != productId) {

matches = **false**;

}

**if** (productName != **null** && !product.getProductName().equalsIgnoreCase(productName)) {

matches = **false**;

}

**if** (category != **null** && !product.getCategory().equalsIgnoreCase(category)) {

matches = **false**;

}

**if** (matches) {

**return** product;

}

}

**return** **null**;

}

}

**BinarySearch.java**

**import** java.util.Comparator;

**public** **class** BinarySearch {

**public** **static** Product binarySearch(Product[] products, String productName, Comparator<Product> comparator) {

**int** left = 0;

**int** right = products.length - 1;

**while** (left <= right) {

**int** mid = left + (right - left) / 2;

**int** result = comparator.compare(products[mid], **new** Product(0, productName, ""));

**if** (result == 0) {

**return** products[mid];

}

**if** (result < 0) {

left = mid + 1;

} **else** {

right = mid - 1;

}

}

**return** **null**;

}

}

**Main.java**

**import** java.util.Arrays;

**import** java.util.Comparator;

**import** java.util.Scanner;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Product[] products = {

**new** Product(1, "Laptop", "Electronics"),

**new** Product(2, "Shirt", "Clothing"),

**new** Product(3, "Coffee Maker", "Home Appliances"),

**new** Product(4, "Smartphone", "Electronics"),

**new** Product(5, "Book", "Books")

};

Comparator<Product> productNameComparator = **new** Comparator<Product>() {

@Override

**public** **int** compare(Product p1, Product p2) {

**return** p1.getProductName().compareToIgnoreCase(p2.getProductName());

}

};

Product[] sortedProducts = products.clone();

Arrays.*sort*(sortedProducts, productNameComparator);

Scanner scanner = **new** Scanner(System.***in***);

**boolean** running = **true**;

**while** (running) {

System.***out***.println("Choose the search method (1 for Linear Search, 2 for Binary Search, 0 to Exit): ");

**int** choice = scanner.nextInt();

scanner.nextLine();

**if** (choice == 0) {

running = **false**;

**break**;

}

System.***out***.println("Enter the product ID (or 0 if not searching by ID): ");

Integer productId = scanner.nextInt();

scanner.nextLine();

System.***out***.println("Enter the product name (or leave empty if not searching by name): ");

String productName = scanner.nextLine();

System.***out***.println("Enter the category (or leave empty if not searching by category): ");

String category = scanner.nextLine();

Product result = **null**;

**switch** (choice) {

**case** 1:

// Linear Search

result = LinearSearch.*linearSearch*(products, productId == 0 ? **null** : productId, productName.isEmpty() ? **null** : productName, category.isEmpty() ? **null** : category);

System.***out***.println("Linear Search Result: " + (result != **null** ? result : "Product not found"));

**break**;

**case** 2:

// Binary Search

**if** (!productName.isEmpty()) {

result = BinarySearch.*binarySearch*(sortedProducts, productName, productNameComparator);

System.***out***.println("Binary Search Result: " + (result != **null** ? result : "Product not found"));

} **else** {

System.***out***.println("Binary search requires a non-empty product name.");

}

**break**;

**default**:

System.***out***.println("Invalid choice");

**break**;

}

}

scanner.close();

System.***out***.println("Exiting the program.");

}

}

**4.Analysis:**

* 1. **Compare the time complexity of linear and binary search algorithms.**

Linear Search

* Time Complexity: O(n)
  + Best Case: O(1)— Occurs if the target element is the first element in the list.
  + Average Case: O(n) — On average, the algorithm checks about half of the elements in the list.
  + Worst Case: O(n)— Occurs if the target element is at the end of the list or not present.

Binary Search

* Time Complexity: O(log n)
  + Best Case: O(1) Occurs if the target element is at the middle of the list during the first comparison.
  + Average Case: O(log n) On average, the number of elements to check is reduced exponentially with each comparison.
  + Worst Case: O(log n) Even if the target element is not present, the number of comparisons required is logarithmic in relation to the number of elements.
  1. **Discuss which algorithm is more suitable for your platform and why.**

Binary Search is more efficient, O(log n) for large, sorted datasets and scenarios where searches are frequent.

Linear Search is simpler and may be more practical for small, unsorted, or frequently changing datasets, but it is less efficient for large datasets O(n).

For an e-commerce platform with a large and frequently queried product items, binary search is generally the better choice if the product list can be sorted. For smaller or more dynamic lists, linear search may be more practical.

**Exercise 3: Sorting Customer Orders**

1. **Understand Sorting Algorithms:**

* **Bubble Sort -** Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Time Complexity:

* + Best Case: O(n)- When the list is already sorted .
  + Average Case: O(n^2)
  + Worst Case: O(n^2)
* **Insertion Sort -** Insertion sort is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.
* Time Complexity:
  + Best Case: O(n)— When the list is already sorted.
  + Average Case: O(n^2)
  + Worst Case: O(n^2)
* **Quick Sort -** Quicksort is a sorting algorithm based on the divide and conquer approach where an array is divided into subarrays by selecting a pivot element (element selected from the array). While dividing the array, the pivot element should be positioned in such a way that elements less than the pivot are kept on the left side, and elements greater than the pivot is on the right side of the pivot. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

**Time Complexity**:

* + **Best Case**: O(n log n) — When the pivot divides the array into roughly equal parts.
  + **Average Case**: O(n log n)
  + **Worst Case**: O(n^2)— When the pivot is the smallest or largest element.
* **Merge Sort** - The Merge Sort algorithm is a sorting algorithm that is based on the Divide and Conquers paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner. The merge sort algorithm is an implementation of the divide and conquers technique. Thus, it gets completed in three steps:

Divide: In this step, the array/list divides itself recursively into sub-arrays until the base case is reached.

Conquer: Here, the sub-arrays are sorted using recursion.

Combine: This step makes use of the merge( ) function to combine the sub-arrays into the final sorted array.

**Time Complexity**:

* + **Best Case**: O(n log n)
  + **Average Case**: O(n log n)
  + **Worst Case**: O(n log n)

1. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.

**Order.java**

**public** **class** Order {

**private** **int** orderId;

**private** String customerName;

**private** **double** totalPrice;

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

**this**.orderId = orderId;

**this**.customerName = customerName;

**this**.totalPrice = totalPrice;

}

**public** **int** getOrderId() {

**return** orderId;

}

**public** **void** setOrderId(**int** orderId) {

**this**.orderId = orderId;

}

**public** String getCustomerName() {

**return** customerName;

}

**public** **void** setCustomerName(String customerName) {

**this**.customerName = customerName;

}

**public** **double** getTotalPrice() {

**return** totalPrice;

}

**public** **void** setTotalPrice(**double** totalPrice) {

**this**.totalPrice = totalPrice;

}

@Override

**public** String toString() {

**return** "Order{" +

"orderId=" + orderId +

", customerName='" + customerName + '\'' +

", totalPrice=" + totalPrice +

'}';

}

}

1. **Implementation:**

**BubbleSort.java**

**public** **class** BubbleSort {

**public** **static** **void** bubbleSort(Order[] orders) {

**int** n = orders.length;

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

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

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

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

}

**QuickSort.java**

**public** **class** QuickSort {

**public** **static** **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);

}

}

**private** **static** **int** partition(Order[] orders, **int** low, **int** high) {

**double** pivot = orders[high].getTotalPrice();

**int** i = (low - 1);

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

**if** (orders[j].getTotalPrice() <= 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;

}

}

**Main.java**

**import** java.util.Scanner;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Order[] orders = {

**new** Order(1, "Alice", 250.0),

**new** Order(2, "Bob", 150.0),

**new** Order(3, "Charlie", 300.0),

**new** Order(4, "David", 200.0),

**new** Order(5, "Eve", 100.0)

};

Scanner scanner = **new** Scanner(System.***in***);

**boolean** running = **true**;

**while** (running) {

System.***out***.println("Choose the sorting method (1 for Bubble Sort, 2 for Quick Sort, 0 to Exit): ");

**int** choice = scanner.nextInt();

scanner.nextLine();

Order[] sortedOrders = orders.clone();

**switch** (choice) {

**case** 1:

BubbleSort.*bubbleSort*(sortedOrders);

System.***out***.println("Orders sorted by Bubble Sort:");

**for** (Order order : sortedOrders) {

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

}

**break**;

**case** 2:

QuickSort.*quickSort*(sortedOrders, 0, sortedOrders.length - 1);

System.***out***.println("Orders sorted by Quick Sort:");

**for** (Order order : sortedOrders) {

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

}

**break**;

**case** 0:

running = **false**;

**break**;

**default**:

System.***out***.println("Invalid choice");

**break**;

}

}

scanner.close();

System.***out***.println("Exiting the program.");

}

}

1. **Analysis:**

**Bubble Sort vs. Quick Sort**

* **Bubble Sort**:
  + Time Complexity:
    - Best Case: O(n)— Optimized version for already sorted arrays.
    - Average Case: O(n^2)
    - Worst Case: O(n^2)
  + Simple but inefficient for large datasets due to quadratic time complexity.
* **Quick Sort**:
  + Time Complexity:
    - Best Case: O(n log n)
    - Average Case: O(n log n)
    - Worst Case: O(n^2)
  + Generally faster and more efficient for large datasets, with average-case performance significantly better than Bubble Sort.

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

* **Efficiency**: Quick Sort is typically more efficient than Bubble Sort due to its average-case time complexity.
* **Performance**: Even though both algorithms have O(n^2) worst-case scenarios, Quick Sort is faster in practice due to better average-case performance and smaller constants in its runtime.
* **Scalability**: Quick Sort scales better with larger datasets, making it suitable for real-world applications.

**Exercise 4: Employee Management System**

1. **Understand Array Representation:**

Arrays are a collection of elements stored in contiguous memory locations. The elements are indexed, and each element can be accessed in constant time using its index, making arrays very efficient for read operations.

**Advantages of Arrays:**

**Direct Access:** Accessing an element by its index is very fast (O(1) time complexity).

**Efficient Memory Usage:** Arrays use a fixed amount of memory for a given size, making them memory-efficient for static collections of data.

**Cache-Friendly**: Due to contiguous memory allocation, arrays make better use of CPU caches, leading to faster access times.

1. **Setup:**

**Employee.java**

**public** **class** Employee {

**private** **int** employeeId;

**private** String name;

**private** String position;

**private** **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** **int** getEmployeeId() {

**return** employeeId;

}

**public** **void** setEmployeeId(**int** employeeId) {

**this**.employeeId = employeeId;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getPosition() {

**return** position;

}

**public** **void** setPosition(String position) {

**this**.position = position;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

@Override

**public** String toString() {

**return** "Employee:" +"employeeId=" + employeeId +

" name='" + name + '\'' +

" position='" + position + '\'' +

" salary=" + salary ;

}

}

1. **Implementation:**

**EmployeeManagementSystem.java**

**import** java.util.Scanner;

**public** **class** EmployeeManagementSystem {

**private** Employee[] employees;

**private** **int** size;

**private** **int** capacity;

**public** EmployeeManagementSystem(**int** capacity) {

**this**.capacity = capacity;

**this**.employees = **new** Employee[capacity];

**this**.size = 0;

}

**public** **void** addEmployee(Employee employee) {

**if** (size == capacity) {

System.***out***.println("Array is full, cannot add more employees.");

**return**;

}

employees[size++] = employee;

}

**public** Employee searchEmployee(**int** employeeId) {

**for** (**int** i = 0; i < size; i++) {

**if** (employees[i].getEmployeeId() == employeeId) {

**return** employees[i];

}

}

**return** **null**;

}

**public** **void** traverseEmployees() {

**for** (**int** i = 0; i < size; i++) {

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

}

}

**public** **void** deleteEmployee(**int** employeeId) {

**for** (**int** i = 0; i < size; i++) {

**if** (employees[i].getEmployeeId() == employeeId) {

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

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

}

employees[--size] = **null**;

**return**;

}

}

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

}

**public** **static** **void** main(String[] args) {

EmployeeManagementSystem system = **new** EmployeeManagementSystem(10);

Scanner scanner = **new** Scanner(System.***in***);

**boolean** running = **true**;

**while** (running) {

System.***out***.println("Choose an operation: ");

System.***out***.println("1. Add Employee");

System.***out***.println("2. Search Employee");

System.***out***.println("3. Traverse Employees");

System.***out***.println("4. Delete Employee");

System.***out***.println("0. Exit");

**int** choice = scanner.nextInt();

scanner.nextLine(); // Consume newline character

**switch** (choice) {

**case** 1:

System.***out***.println("Enter Employee ID: ");

**int** id = scanner.nextInt();

scanner.nextLine(); // Consume newline character

System.***out***.println("Enter Employee Name: ");

String name = scanner.nextLine();

System.***out***.println("Enter Employee Position: ");

String position = scanner.nextLine();

System.***out***.println("Enter Employee Salary: ");

**double** salary = scanner.nextDouble();

scanner.nextLine(); // Consume newline character

Employee emp = **new** Employee(id, name, position, salary);

system.addEmployee(emp);

**break**;

**case** 2:

System.***out***.println("Enter Employee ID to search: ");

**int** searchId = scanner.nextInt();

scanner.nextLine(); // Consume newline character

Employee foundEmp = system.searchEmployee(searchId);

System.***out***.println(foundEmp != **null** ? foundEmp : "Employee not found.");

**break**;

**case** 3:

system.traverseEmployees();

**break**;

**case** 4:

System.***out***.println("Enter Employee ID to delete: ");

**int** deleteId = scanner.nextInt();

scanner.nextLine(); // Consume newline character

system.deleteEmployee(deleteId);

**break**;

**case** 0:

running = **false**;

**break**;

**default**:

System.***out***.println("Invalid choice. Please try again.");

**break**;

}

}

scanner.close();

System.***out***.println("Exiting the program.");

}

}

1. **Analysis:**

**Time Complexity:**

* **Add:** O(1) - Adding an employee to the end of the array.
* **Search:** O(n) - Linear search through the array.
* **Traverse:** O(n) - Traversing the entire array.
* **Delete:** O(n) - Linear search to find the employee and shifting elements.

**Limitations of Arrays:**

* **Fixed Size:** Arrays have a fixed size, making them inflexible if the number of employees grows beyond the initial capacity.
* **Inefficient Deletions:** Deleting an element requires shifting elements, which can be inefficient for large arrays.
* **No Dynamic Resizing:** Unlike dynamic data structures (e.g., ArrayList), arrays do not resize automatically.

**Exercise 5: Task Management System**

1. **Understand Linked Lists:**

## **Types of Linked List**

Following are the various types of linked list.

### **Singly Linked Lists**

Singly linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.

### **Doubly Linked Lists**

Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.

### **Circular Linked Lists**

Circular linked lists can exist in both singly linked list and doubly linked list. Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.

1. **Setup:**

**Task.java**

**public** **class** Task {

**private** **int** taskId;

**private** String taskName;

**private** String status;

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

**this**.taskId = taskId;

**this**.taskName = taskName;

**this**.status = status;

}

**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;

}

@Override

**public** String toString() {

**return** "Task" +

"taskId=" + taskId +

", taskName='" + taskName + '\'' +

", status='" + status + '\'' ;

}

}

1. **Implementation:**

**SinglyLinkedList.java**

**public** **class** SinglyLinkedList {

**private** Node head;

**private** **static** **class** Node {

Task task;

Node next;

Node(Task task) {

**this**.task = task;

**this**.next = **null**;

}

}

// Add a new task

**public** **void** addTask(Task task) {

Node newNode = **new** Node(task);

**if** (head == **null**) {

head = newNode;

} **else** {

Node current = head;

**while** (current.next != **null**) {

current = current.next;

}

current.next = newNode;

}

}

// Search for a task by ID

**public** Task searchTask(**int** taskId) {

Node current = head;

**while** (current != **null**) {

**if** (current.task.getTaskId() == taskId) {

**return** current.task;

}

current = current.next;

}

**return** **null**;

}

// Traverse and print all tasks

**public** **void** traverseTasks() {

Node current = head;

**while** (current != **null**) {

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

current = current.next;

}

}

// Delete a task by ID

**public** **void** deleteTask(**int** taskId) {

**if** (head == **null**) **return**;

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

head = head.next;

**return**;

}

Node current = head;

**while** (current.next != **null**) {

**if** (current.next.task.getTaskId() == taskId) {

current.next = current.next.next;

**return**;

}

current = current.next;

}

}

}

**TaskManagementSystem.java**

**import** java.util.Scanner;

**public** **class** TaskManagementSystem {

**public** **static** **void** main(String[] args) {

SinglyLinkedList taskList = **new** SinglyLinkedList();

Scanner scanner = **new** Scanner(System.***in***);

**boolean** running = **true**;

**while** (running) {

System.***out***.println("Choose an operation: ");

System.***out***.println("1. Add Task");

System.***out***.println("2. Search Task");

System.***out***.println("3. Traverse Tasks");

System.***out***.println("4. Delete Task");

System.***out***.println("0. Exit");

**int** choice = scanner.nextInt();

scanner.nextLine(); // Consume newline character

**switch** (choice) {

**case** 1:

System.***out***.println("Enter Task ID: ");

**int** id = scanner.nextInt();

scanner.nextLine(); // Consume newline character

System.***out***.println("Enter Task Name: ");

String name = scanner.nextLine();

System.***out***.println("Enter Task Status: ");

String status = scanner.nextLine();

Task task = **new** Task(id, name, status);

taskList.addTask(task);

**break**;

**case** 2:

System.***out***.println("Enter Task ID to search: ");

**int** searchId = scanner.nextInt();

scanner.nextLine(); // Consume newline character

Task foundTask = taskList.searchTask(searchId);

System.***out***.println(foundTask != **null** ? foundTask : "Task not found.");

**break**;

**case** 3:

taskList.traverseTasks();

**break**;

**case** 4:

System.***out***.println("Enter Task ID to delete: ");

**int** deleteId = scanner.nextInt();

scanner.nextLine(); // Consume newline character

taskList.deleteTask(deleteId);

**break**;

**case** 0:

running = **false**;

**break**;

**default**:

System.***out***.println("Invalid choice. Please try again.");

**break**;

}

}

scanner.close();

System.***out***.println("Exiting the program.");

}

}

1. **Analysis:**

**Time Complexity:**

* **Add:** O(n) - Inserting a new task at the end of the list requires traversal to the end.
* **Search:** O(n) - Linear search through the list.
* **Traverse:** O(n) - Traversing the entire list.
* **Delete:** O(n) - Linear search to find the task and updating links.

**Advantages of Linked Lists over Arrays:**

* **Dynamic Size:** Linked lists can grow or shrink dynamically, eliminating the need for a predefined size.
* **Efficient Insertions/Deletions:** Insertions and deletions do not require shifting elements, making them more efficient for these operations compared to arrays.

**Exercise 6: Library Management System**

1. **Understand Search Algorithms:**

**Linear Search -** Linear Search scans each element of the array or list sequentially to find the target value. It does not require the array to be sorted.

* **Best Case**: The best-case scenario occurs when the target element is the first element of the array. The time complexity is O(1) because the search is completed in a single comparison.
* **Average Case**: In the average case, the target element is somewhere in the middle of the array. On average, it takes about n/2 comparisons to find the target, where n is the number of elements. The time complexity is O(n).
* **Worst Case**: The worst-case scenario occurs when the target element is either not present in the array or is the last element. The algorithm has to scan through all n elements. The time complexity is O(n).

#### ****Binary Search -** Binary Search** requires the array to be sorted. It repeatedly divides the search interval in half, making it efficient for large datasets.

* **Best Case**: The best-case scenario occurs when the target element is exactly in the middle of the array during the first comparison. The time complexity is O(1).
* **Average Case**: In the average case, binary search performs a logarithmic number of comparisons, because each step halves the search space. The time complexity is O(log n), where n is the number of elements.
* **Worst Case**: The worst-case scenario occurs when the target element is not present in the array. Even in this case, binary search still has to divide the array logarithmically, making the time complexity O(log n).

1. **Setup:**

**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** String getTitle() {

**return** title;

}

**public** String getAuthor() {

**return** author;

}

@Override

**public** String toString() {

**return** "Book" +

"bookId=" + bookId +

", title='" + title + '\'' +

", author='" + author + '\'' ;

}

}

1. **Implementation:**

**Library.java**

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** Library {

**private** Book[] books;

**public** Library(Book[] books) {

**this**.books = books;

}

// Linear search by title

**public** Book linearSearchByTitle(String title) {

**for** (Book book : books) {

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

**return** book;

}

}

**return** **null**;

}

**public** Book binarySearchByTitle(String title) {

Comparator<Book> titleComparator = **new** Comparator<Book>() {

@Override

**public** **int** compare(Book b1, Book b2) {

**return** b1.getTitle().compareToIgnoreCase(b2.getTitle());

}

};

Arrays.*sort*(books, titleComparator);

**int** left = 0;

**int** right = books.length - 1;

**while** (left <= right) {

**int** mid = left + (right - left) / 2;

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

**if** (cmp == 0) {

**return** books[mid];

} **else** **if** (cmp < 0) {

left = mid + 1;

} **else** {

right = mid - 1;

}

}

**return** **null**;

}

}

**LibraryManagementSystem.java**

**import** java.util.Scanner;

**public** **class** LibraryManagementSystem {

**public** **static** **void** main(String[] args) {

Book[] books = {

**new** Book(1, "The Great Gatsby", "F. Scott Fitzgerald"),

**new** Book(2, "To Kill a Mockingbird", "Harper Lee"),

**new** Book(3, "1984", "George Orwell"),

**new** Book(4, "Pride and Prejudice", "Jane Austen"),

**new** Book(5, "Moby-Dick", "Herman Melville")

};

Library library = **new** Library(books);

Scanner scanner = **new** Scanner(System.***in***);

**boolean** running = **true**;

**while** (running) {

System.***out***.println("Choose the search method (1 for Linear Search, 2 for Binary Search, 0 to Exit): ");

**int** choice = scanner.nextInt();

scanner.nextLine();

**if** (choice == 0) {

running = **false**;

**break**;

}

System.***out***.println("Enter the book title to search: ");

String title = scanner.nextLine();

Book result = **null**;

**switch** (choice) {

**case** 1:

// Linear Search

result = library.linearSearchByTitle(title);

System.***out***.println("Linear Search Result: " + (result != **null** ? result : "Book not found"));

**break**;

**case** 2:

// Binary Search

result = library.binarySearchByTitle(title);

System.***out***.println("Binary Search Result: " + (result != **null** ? result : "Book not found"));

**break**;

**default**:

System.***out***.println("Invalid choice");

**break**;

}

}

scanner.close();

System.***out***.println("Exiting the program.");

}

}

1. **Analysis:**

**Time Complexity:**

* **Linear Search:** O(n) - Searches each element until the target is found.
* **Binary Search:** O(log n) - Divides the search interval in half iteratively, requiring the list to be sorted.

**When to Use Each Algorithm:**

* **Linear Search:** Use when the dataset is small or unsorted. It is simple to implement and works well for small lists.
* **Binary Search:** Use when the dataset is large and sorted. It is more efficient than linear search for large datasets, reducing the number of comparisons needed to find the target.

**Exercise 7: Financial Forecasting**

1. **Understand Recursive Algorithms:**

**Recursion : -** Recursion is defined as a process which calls itself directly or indirectly and the corresponding function is called a recursive function.

**Properties of Recursion:**

Recursion has some important properties. Some of which are mentioned below:

* The primary property of recursion is the ability to solve a problem by breaking it down into smaller sub-problems, each of which can be solved in the same way.
* A recursive function must have a base case or stopping criteria to avoid infinite recursion.
* Recursion involves calling the same function within itself, which leads to a call stack.
* Recursive functions may be less efficient than iterative solutions in terms of memory and performance.

**Advantages:**

* Simplifies code for problems that have a natural recursive structure.
* Makes the code more readable and easier to understand.

**Disadvantages:**

* Can lead to excessive memory usage and stack overflow if not optimized.
* May be slower due to repeated calculations of the same sub-problems.

1. **Setup:**

**FinancialForecasting.java**

**public** **class** FinancialForecasting {

**public** **static** **double** calculateFutureValue(**double** presentValue, **double** growthRate, **int** periods) {

**if** (periods == 0) {

**return** presentValue;

}

**return** (1 + growthRate) \* *calculateFutureValue*(presentValue, growthRate, periods - 1);

}

}

1. **Implementation:**

**FinancialForecastingTest.java**

**import** java.util.Scanner;

**public** **class** FinancialForecastingTest {

**public** **static** **void** main(String[] args) {

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.println("Enter the present value: ");

**double** presentValue = scanner.nextDouble();

System.***out***.println("Enter the annual growth rate (as a decimal, e.g., 0.05 for 5%): ");

**double** growthRate = scanner.nextDouble();

System.***out***.println("Enter the number of periods (years): ");

**int** periods = scanner.nextInt();

// Calculate future value using recursion

**double** futureValue = FinancialForecasting.*calculateFutureValue*(presentValue, growthRate, periods);

System.***out***.println("The future value is: " + futureValue);

scanner.close();

}

}

1. **Analysis:**

**Time Complexity:**

* The time complexity of the recursive algorithm is O(n) where n is the number of periods. This is because the function calls itself once for each period until it reaches the base case.

**Optimization:**

* **Memoization:** To avoid recalculating the future value for the same period multiple times, we can use memoization. This technique stores the results of expensive function calls and returns the cached result when the same inputs occur again.
* **Iterative Approach:** An iterative solution can be more efficient in terms of space complexity since it doesn't involve the overhead of multiple function calls.