DATA STRUCTURE COURSE PROJECT UPDATE

POS System

SUBMITTED BY:

NANDANA RAMACHANRAN

NAVNEETH SANTHOSH

NIMA FATHIMA

PAVITHRA DEEPU E

PRITHVI P

Project Overview

The **Point of Sale (POS) System** is designed to streamline retail operations by handling sales transactions, managing product information, and keeping track of inventory levels. This system helps store employees efficiently register sales, manage stock etc. The system is built using efficient data structures to ensure quick and accurate operations such as searching, sorting, and updating product data.

Key Features of the POS System:

Product Management:

- **Product Registration:** Allows users to add new products to the system by entering details such as product name, ID, category, price, and stock quantity.
- **Product Update:** Users can update the product information like price changes or stock adjustments.
- **Product Deletion:** Products no longer in use can be removed from the system.
- **Sorting and Searching:** Products can be sorted by name, ID, price, or category, and users can search for products based on these attributes

Data Structures Used:

- **Linked Lists:** Used to store product information and manage the dynamic nature of the inventory.
- **Arrays:** For efficient access to product details during transactions.

Algorithms Used:

- **Binary Search:** For quick product searches when data is sorted.

- **Merge Sort:** For sorting product lists by price, name, or other attributes.
- **Bubble Sort:** For efficient insertion of new products into an already sorted list.

POS System Design:

Menu Interface

The system will feature a menu-based user interface where users can choose different operations such as product management, transaction processing, and reporting. The menu will be implemented using switch-case logic, and the menu will loop until the user exits the system.

1. Product Registration

- Steps:
    1. Create a new node for the product (using a linked list).
    2. Add necessary product attributes like name, ID, price, category, and quantity to the node.
    3. Link the node with the previous product node (if any).
    4. Add the node's pointer to an array for quicker access to products.
    5. If auto-sorting is enabled, run Bubble Sort Enhanced for efficient insertion into a sorted list.

---

2. Product Update

- Steps:
    1. Search for the product to be updated using either Linear Search (if unsorted) or Binary Search (if sorted).
    2. Update the product's details (such as price or quantity).
    3. If the updated attribute requires sorting (e.g., price), re-run Merge Sort to re-sort the array.

---

3. Product Deletion

- Steps:
    1. Search for the product to be deleted.
    2. Delete the product node from the linked list.
    3. Update the pointer array by removing the node reference.
    4. Re-run Merge Sort on the array to maintain the sorted state (if needed).

---

4. Sorting

- Steps:
    1. Allow the user to select the attribute by which the products will be sorted (e.g., price, name, category).
    2. Use Merge Sort to efficiently sort the products based on the selected attribute.

---

5. Searching

- Steps:
    1. Allow the user to select the attribute by which to search (e.g., product ID, name).
    2. Check if the array is sorted by the selected attribute:
        - If sorted, use Binary Search for efficient searching.
        - If unsorted, use Linear Search to find the product.

---

6. Data Saving

- Steps:
    1. Open a file for saving using file handling techniques.

2. Save each product's data (ID, name, price, quantity) into the file line by line.
3. For transactions, save the details of each transaction, including the products purchased, total price, and payment method.

---

7. Merge Sort

- Steps:
    1. Divide the array into two halves.
    2. Recursively sort each half using Merge Sort.
    3. Merge the two sorted halves while maintaining the correct order.
    4. This ensures that the entire product list remains sorted based on the selected attribute.

ALGORITHMS

1. Linear Search

- **Purpose:** Used to find a product in the inventory when the data is **unsorted**.
- **Algorithm:**

```
linearSearch(array, target):
  for i = 0 to length(array) - 1:
    if array[i] == target:
      return i
  return -1
```

- **Time Complexity:**
  - Best Case: **O(1)** (target is the first element)
  - Worst Case: **O(n)** (target is the last element or not found)
  - Average Case: **O(n)**

---

2. Binary Search

- **Purpose:** Used to find a product when the data is **sorted**.
- **Algorithm:**

```
binarySearch(array, target):
  head = 0
  tail = length(array) - 1
  while head <= tail:
    mid = (head + tail) / 2
    if array[mid] == target:
      return mid
```

```
    else if array[mid] > target:
        tail = mid - 1
    else:
        head = mid + 1
  return -1
```

- **Time Complexity:**
  - Best Case: **O(1)** (target is the middle element)
  - Worst Case: **O(log n)** (dividing the array by half each time)
  - Average Case: **O(log n)**

---

3. Merge Sort

- **Purpose:** Used to sort the inventory based on various attributes (e.g., price, product name).
- **Algorithm:**

```
mergeSort(array):
  if length(array) <= 1:
    return array
  mid = length(array) // 2
  left = mergeSort(array[0:mid])
  right = mergeSort(array[mid:length(array)])
  return merge(left, right)

merge(left, right):
  result = []
  while left and right:
```

```
    if left[0] < right[0]:
        result.append(left[0])
        left = left[1:]
    else:
        result.append(right[0])
        right = right[1:]
  return result + left + right
```

- **Time Complexity:**
  - Best Case: **O(n log n)**
  - Worst Case: **O(n log n)**
  - Average Case: **O(n log n)**
- **Space Complexity: O(n)** (requires extra space for merging)

---

4. Bubble Sort (Enhanced for Insertion)

- **Purpose:** Used to insert a new product into a **sorted list**. If the list is already sorted, this modified bubble sort can efficiently place the new item in its correct position.
- **Algorithm:**

```
bubbleSortEnhanced(array, new_element):
  append new_element to the array
  for i = length(array) - 1 to 1:
    if array[i] < array[i-1]:
      swap(array[i], array[i-1])
    else:
      break
```

- **Time Complexity:**
  - Best Case: **O(1)** (if the new element is already in its correct position)
  - Worst Case: **O(n)** (if the new element needs to be moved to the beginning)
  - Average Case: **O(n)**

List of Functions used in switch case:

case 1: registerProduct()

case 2: updateProduct()

case 3: deleteProduct()

case 4: sortProducts()

case 5: searchProduct()

case 6: processTransaction()

case 7: saveData()

case 8: exitSystem()

USECASE DIAGRAM