****

**Group members: Muhammad Sufyan sp22-bcs-115**

**Muhammad Asad sp22-bcs-050**

**Muhammad Abdullah sp22-bcs-049**

**Project name : Auto Sales Data Analysis**

**SECTION : 3A**

**Submitted to : Dr.Inayat-ur-Rehman**

**Introduction**:

The implementation of a Binary Search Tree (BST) in the Auto Sales data analysis project offers an efficient and organized structure for managing and manipulating the dataset. In this report, we will explore how and why a BST was employed for this specific application.

**Dataset overview:**

The dataset used in this project contains information about auto sales, including order numbers, quantity ordered, sales, status, product type, customer details, and contact information. The dataset is diverse and requires effective data structures to facilitate various operations.

**BST implementation:**

The BST structure was integrated into the project to enable efficient searching, insertion, and removal of orders based on the quantity ordered. The primary motivation for choosing a BST lies in its ability to maintain a sorted order of elements, which aligns well with the requirements of searching and range-based retrieval.

**Insertion Operation:**

The dataset is read from a CSV file, and for each record, a new 'str' structure is created to store the relevant information. The 'insertBST' function is then utilized to insert the 'str' node into the BST based on the quantity ordered. This ensures that the tree maintains a balanced structure, facilitating quicker search operations.

**root = insertBST(root, curr); // Build BST during insertion**

**Search operation and display:**

The 'displayBST' function is designed to perform an in-order traversal of the BST, resulting in a display of the dataset in ascending order of quantity. This operation is crucial for visually assessing the entire dataset.

**displayBST(root);**

**Search within quantity range:**

To display orders falling within a specific quantity range, the 'displayByQuantityRange' function performs an in-order traversal, printing only the nodes that satisfy the specified criteria. This operation efficiently retrieves data within the desired range.

**displayByQuantityRange(root, 10, 20)**

**Order removal operation:**

The 'removeOrder' function is implemented to remove a specific order from the BST based on the order number. This function handles different cases, such as nodes with zero or one child, making the removal process seamless.

**root = removeOrder(root, 5, removedData**

**Bst insertion code:**

**BSTNode\* insertBST(BSTNode\* root, str\* data) {**

**if (root == NULL) {**

**BSTNode\* newNode = new BSTNode;**

**newNode->data = data;**

**newNode->left = newNode->right = NULL;**

**return newNode;**

**}**

**if (data->quantityordered < root->data->quantityordered) {**

**root->left = insertBST(root->left, data);**

**} else {**

**root->right = insertBST(root->right, data);**

**}**

**return root;**

**}**

**Conclusion:**

The integration of a BST into the Auto Sales data analysis project provides a scalable and efficient solution for managing and analyzing the dataset. The BST's inherent properties of maintaining sorted order, efficient search, and balanced structure make it an ideal choice for tasks such as range-based queries and order removal. The project successfully leverages the advantages of BST, enhancing the overall performance and usability of the system.

## **Why Use a Stack?**

### **1. LIFO Principle**

The primary motivation for choosing a stack is its adherence to the Last-In-First-Out (LIFO) principle. In the context of order data management, the LIFO property is well-aligned with real-world scenarios. The latest order placed is often the most relevant for users, and a stack ensures that this order is easily accessible at the top of the stack. This simplifies operations such as viewing the most recent orders and undoing the last entry.

### **2.** **Order History Tracking**

The stack's natural ability to preserve the order of data entries makes it ideal for tracking order history. As new orders are added, they are pushed onto the stack, maintaining a sequential record of order placement. This sequential history is crucial for users who need to understand the chronological evolution of orders, facilitating decision-making and trend analysis.

### **3**. **Undo Functionality**

The LIFO nature of the stack perfectly suits the requirement for undo functionality. Users may make errors while entering orders, and the ability to undo the last entry is a valuable feature. The stack's pop operation allows for the seamless removal of the most recent order, providing a straightforward and efficient means of correction.

### **4. Efficient Memory Management**

Stacks are known for their simplicity and efficiency in memory management. They have a fixed size, simplifying memory allocation and deallocation processes. In the context of order data management, where the emphasis is on maintaining a concise history, the stack's efficient memory management ensures optimal use of resources.

## **Implementation Overview**

The program utilizes a stack of **Order** structures named **order History** to manage and store historical order data. The **insertion** function reads data from a CSV file, creates **Order** objects, and pushes them onto the stack. Functions such as **printOrderHistory**, **addToOrderHistory**, and **undoLastOrderEntry** leverage the stack's properties to display, add, and undo orders efficiently

**void printOrderHistory(stack<Order>& orderHistory) {**

**if (orderHistory.empty()) {**

**cout << "Order history is empty." << endl;**

**return;**

**}**

**cout << "Order History:" << endl;**

**cout << "-----------------------------" << endl;**

**stack<Order> tempStack = orderHistory;**

**while (!tempStack.empty()) {**

**printOrder(tempStack.top());**

**tempStack.pop();**

**}**

**}**

## **Conclusion:**

The implementation of a stack data structure in this program provides an effective and intuitive way to manage order data. The LIFO principle of the stack aligns well with the temporal nature of order placement, making it a suitable choice for tracking order history and supporting functionalities like undoing the last entry. The stack's efficiency and simplicity contribute to the overall robustness of the program.

**Why Use a Priority Queue?**

**Efficient Order Organization**

A priority queue is chosen for its ability to organize orders based on different criteria efficiently. In this implementation, the priority queue allows for the insertion of orders in ascending order based on days left, ascending order based on sales, and descending order based on quantity ordered. This organization enables quick access to orders with specific priorities, enhancing the program's versatility.

**Priority-Based Dequeue Operations**

The priority queue facilitates priority-based dequeue operations, enabling the extraction of orders with the highest or lowest priority. For instance, the program can easily dequeue the order with the fewest days left, the highest sales, or the maximum quantity ordered. This functionality is crucial for tasks like displaying relevant information to users or making decisions based on specific criteria

**Flexible Order Retrieval**

The priority queue allows for flexible order retrieval based on different attributes. The program can easily display orders in ascending order based on days left, ascending order based on sales, or descending order based on quantity ordered. This flexibility caters to diverse user needs, providing a dynamic and adaptable solution for order data analysis

**Implementation Overview**

The program implements a priority queue using a linked list structure to manage orders efficiently. Three separate queues are maintained for different sorting criteria: days left, sales, and quantity ordered. The program enqueues orders into these queues based on the specified criteria and dequeues orders as needed for various operations.

**class PriorityQueue {**

// Implementation of priority queue functions as described in the code // ...};/ Priority queue instance

**PriorityQueue priorityQueue;**

**void enqueueSalesAsc(str \*order)**

**{**

**str \*newNode = new str(\*order);**

**if (frontSalesAsc == nullptr || order->sales < frontSalesAsc->sales)**

**{**

**newNode->next = frontSalesAsc;**

**frontSalesAsc = newNode;**

**}**

**else**

**{**

**str \*curr = frontSalesAsc;**

**while (curr->next != nullptr && order->sales >= curr->next->sales)**

**{**

**curr = curr->next;**

**}**

**newNode->next = curr->next;**

**curr->next = newNode;**

**}**

**}**

**// Function to enqueue an order into the priority queue in descending order based on quantity ordered**

**void enqueueQuantityDesc(str \*order)**

**{**

**str \*newNode = new str(\*order);**

**if (frontQuantityDesc == nullptr || order->quantityordered > frontQuantityDesc->quantityordered)**

**{**

**newNode->next = frontQuantityDesc;**

**frontQuantityDesc = newNode;**

**}**

**else**

**{**

**str \*curr = frontQuantityDesc;**

**while (curr->next != nullptr && order->quantityordered <= curr->next->quantityordered)**

**{**

**curr = curr->next;**

**}**

**newNode->next = curr->next;**

**curr->next = newNode;**

**}**

**}**

**Conclusion**

The use of a priority queue in this program enhances the efficiency of order data management and analysis. The priority queue's ability to organize and retrieve orders based on different criteria provides a versatile solution for displaying relevant information to users. The flexibility offered by the priority queue ensures that the program can adapt to various user requirements, making it a robust tool for order data analysis.

**Why Use Prim's Algorithm?**

**Identifying Significant Connections**

Prim's algorithm is selected for its ability to identify the most significant connections within the dataset. By treating each order as a vertex in the graph and assigning edge weights based on the price differences between orders, the algorithm efficiently identifies the Minimum Spanning Tree. This tree represents the essential connections between orders that collectively encompass the entire dataset.

. **Efficient Graph Construction**

The algorithm's stepwise approach ensures the efficient construction of the Minimum Spanning Tree. It starts with a single vertex and incrementally adds the most cost-effective edges until all vertices are connected. This process minimizes the total weight of the tree, providing a clear representation of the crucial connections between orders.

**Insights into Price Relationships**

Prim's algorithm facilitates the exploration of price relationships between orders. By visualizing the Minimum Spanning Tree, users can gain insights into how prices vary across different orders and identify clusters of orders with similar price characteristics. This analysis can be valuable for pricing strategy optimization and understanding the pricing dynamics within the dataset.

**Implementation Overview**

The program reads order data from a CSV file, constructs a graph based on the price differences between orders, and applies Prim's algorithm to find the Minimum Spanning Tree. The resulting tree is then displayed, showcasing the most significant connections between orders.

**void primMST() {**

**int parent[MAX\_ORDERS];**

**int key[MAX\_ORDERS];**

**bool inMST[MAX\_ORDERS];**

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

**key[i] = INF;**

**inMST[i] = false;**

**}**

**key[0] = 0; // Start with the first order**

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

**int u = minKey(key, inMST);**

**inMST[u] = true;**

**for (int v = 0; v < numOrders; ++v) {**

**if (!inMST[v] && adjacencyMatrix[u][v] < key[v]) {**

**parent[v] = u;**

**key[v] = adjacencyMatrix[u][v];**

**}**

**}**

**}**

**printMST(parent);**

**}**

**Conclusion**

Prim's algorithm proves to be a valuable tool for order data analysis, offering a systematic way to identify and visualize the most significant connections between orders. The Minimum Spanning Tree generated by the algorithm provides insights into the relationships within the dataset, particularly in terms of price differences. This approach enhances the understanding of the dataset's structure and can inform decision-making processes related to pricing and order dependencies

**Sorting Algorithms**

**Bubble Sort**

**Overview**

Bubble Sort is a simple sorting algorithm that iteratively steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

**Implementation**

The **bubbleSort** function is implemented to sort the orders based on the **priceEach** attribute. It performs multiple passes through the order array, swapping elements as needed.

**void bubbleSort(Order orders[], int orderCount) {**

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

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

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

**swap(orders[j], orders[j + 1]);**

**}**

**}**

**}**

**}**

**Evaluation**

Bubble Sort is easy to understand but less efficient for large datasets. Its time complexity is O(n^2), where n is the number of orders.

**2. Merge Sort**

**Overview**

Merge Sort is a divide-and-conquer algorithm that divides the unsorted list into n sub-lists, recursively sorts them, and then merges them to produce a sorted list.

**Implementation**

The **mergeSort** function is implemented to sort the orders using the Merge Sort algorithm. It divides the array into smaller sub-arrays, sorts them, and then merges them back together.

**void mergeSort(Order orders[], int l, int r) {**

**if (l < r) {**

**int m = l + (r - l) / 2;**

**mergeSort(orders, l, m);**

**mergeSort(orders, m + 1, r);**

**merge(orders, l, m, r);**

**}**

**}**

**Evaluation**

Merge Sort is more efficient than Bubble Sort, especially for larger datasets. Its time complexity is O(n log n), making it suitable for handling significant amounts of data.

**3. Selection Sort**

**Overview**

Selection Sort is a simple sorting algorithm that divides the input list into a sorted and an unsorted region, repeatedly selecting the minimum element from the unsorted region and swapping it with the first unsorted element.

**Implementation**

The **selectionSort** function is implemented to sort the orders based on the **priceEach** attribute. It iterates through the array, selects the minimum element, and swaps it with the first unsorted element.

**void selectionSort(Order orders[], int orderCount) {**

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

**int minIndex = i;**

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

**if (orders[j].priceEach < orders[minIndex].priceEach) {**

**minIndex = j;**

**}**

**}**

**swap(orders[i], orders[minIndex]);**

**Evaluation**

Similar to Bubble Sort, Selection Sort has a time complexity of O(n^2). It is straightforward but less efficient for larger datasets compared to Merge Sort.

**4. Hashing (Hash Table)**

**Overview**

Hashing is implemented using a basic Hash Table to efficiently store and retrieve order data. Each order is hashed to determine its position in the table.

**Implementation**

The **HashTable** class is implemented, providing methods to insert orders, handle collisions, print the table, and clear its contents.

**Evaluation**

Hashing provides efficient retrieval of individual orders based on their **orderNumber**. However, the basic implementation may suffer from collisions, and a more sophisticated hashing strategy could be considered for further improvements.

**void insert(const Order& order) {**

**int index = hashFunction(order.orderNumber);**

**if (table[index] == nullptr) {**

**table[index] = new Order(order);**

**} else {**

**// Handle collision as needed**

**// For simplicity, you might want to use a different index**

**}**

**}**

**5. AVL Tree**

**Overview**

An AVL Tree is implemented to maintain a balanced binary search tree of orders, allowing for efficient insertion and retrieval operations.

**Implementation**

The AVL Tree is implemented using the **AVLNode** structure and supporting functions for rotation, insertion, and in-order traversal.

**Evaluation**

The AVL Tree ensures a balanced structure, providing efficient search and insertion operations with a time complexity of O(log n). It is particularly beneficial for dynamic datasets.

**Conclusion**

The implemented methods in the C++ code provide a comprehensive set of tools for order data analysis and manipulation. The choice of sorting algorithms and data

structures depends on the specific requirements and characteristics of the data being processed. Further optimizations and enhancements can be explored based on the nature and scale of the datasets used in real-world scenarios.