Data Structure Course project
Project update
Topic:Inventory Management System

Submitted By,
Gokul G Nair
ASI23CA031
Gokul P
ASI23CA032
Govindlal TL
ASI23CA033
Irine MB
ASI23CA034
Ivan Babu
ASI23CA035

Project Overview

The Inventory management system is designed to stream line the operations such as management of stock level, track product details and such. The system will allow a user to add,update,search,sort,and delete product information. It uses efficient data structures such as linked list in order to store and retrieve data for efficient operations. It uses a Menu based User interface approach to simplify the development.

Key Features of The system,

- Product Registration: Add new products with attributes such as Name,Product ID,Category, price, quantity
- Sorting: The inventory can be sorted on basics such as Name, Product ID , Price
- Product Update: allow user to Update Stock
- Search: The user will be able to search the Inventory using attributes such as Name, product Id, Price, Category
- Data Saving& Loading: The Inventory can be saved to a Text File and that data can be retrieved when restarting the system

Data structures used

- Linked Lists: used to store the Inventory each node will be for each product.
- Pointer Array: Each nodes pointer will be added to the array this allows the use of more efficient searching and sorting.

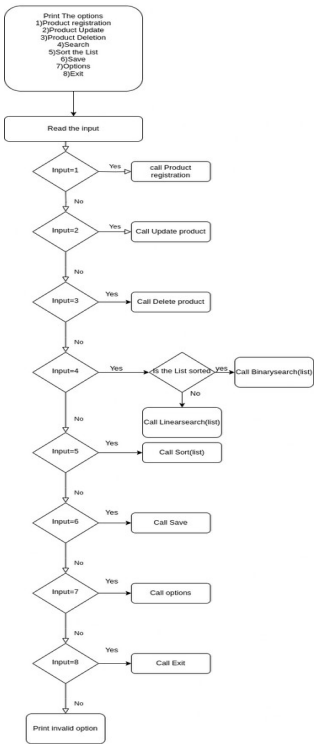
Algorithms Used

- Linear Search: Searching when the system is unsorted
- Binary Search: Searching when the system is sorted
- Merge Sort: For sorting the system
- Bubble Sort:For sorting during insertion

System design

Menu

It will be the User interface of the system. It will contain all the Options it uses switch cases for handling the Operations. The switch will loop until The program is exited



1)Product registration

1) Create a new node
2) Add the required data to that node
3) Link the node with the previous node
4) Add the address of the new node to the array
    1) If auto sorting is enabled use a simplified bubblesort
5) Exit

2)**Product Update**

1) Search for the Product to be updated
2) Update the product
    1) If Sorted attribute is changed re sort the array
3) Exit

3)Product Deletion

1) Search the product to be updated
2) Delete the product
3) Update the array

4)Sorting

1) Select whichever attribute to be sorted
2) Use merge sort to quickly sort the array

5)Searching

1) Select which attribute to be searched
2) Check weather the array is sorted based on the selected attribute
    1) If true run Binary search
    2) else run linear search

6)**Data Saving**

1) Open the save file using file handling
2) Save the data one by one to the file

Bubble sort Enhanced for insertion of an element

It is an algorithm derived from bubble sort in which it starts from rear and any time the elements doesn't need to be swapped it breaks. In this system this can be used for insertion when the elements is already sorted, due to the only misplaced item will be the item we inserted at the end. By implementing this algorithm we can achieve a Worst case of O(n) due to if the smallest element is the element we inserted it would still only take one full pass of the array to be aligned properly.

Algorithm

```
for(i=n;i>-1;i--)
        if(a[i]<a[i-1])
```

```
            swap(a[i],a[i-1])
        else
            break;
```

Time complexity
Worst case - O(n)
Best case - O(1)
Average case – O(n)
Space Complexity- O(1) [In place sorting]

Merge sort

Merge Sort is a divide-and-conquer algorithm. It divides the input array into two halves, recursively sorts the two halves, and finally merges the two sorted halves.

Algorithm

1. If the array has one or zero elements, it is already sorted.
2. Divide the array into two halves.
3. Recursively sort each half.
4. Merge the two halves together, maintaining the sorted order.

Time Complexity:

Best case: O(n log n)

Worst case: O(n log n)

Average case: O(n log n)

Space complexity: O(n)

Linear Search

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)

Worst case: O(n)

Average case: O(n)

Binary Search

Algorithm

```
function 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)

Worst case: O(log n)

Average case: O(log n)

Use case diagram