# Week 2 Assignment – Inventory Management System

## Inventory Management System

## This project could be used to manage a large inventory with various categories, quantities, and location tracking, efficiently storing items where most storage slots are empty.

## The system will use a multi-dimensional array to represent storage locations and a sparse array to handle specific inventory items with non-zero quantities.

## By employing sparse representation, we minimize memory usage for an inventory system where most items have zero stock across many locations.

## Key Features

## Inventory Storage:

## Use a multi-dimensional array to represent storage bins, aisles, and warehouse locations.

## Implement a sparse array to store non-zero quantities only, optimizing memory usage.

## Functionalities:

## Add and Remove Inventory: Add or remove items in specific locations.

## Check Stock: Get the quantity of a specific item at a given location.

## Display Inventory: Display all items, including their location and quantity.

## Transfer Items: Transfer items from one location to another.

## Low-Stock Alerts: Flag items with stock levels below a specified threshold.

## User Interface:

## Provide a command-line interface for interaction, such as adding items, checking stock, transferring items, and displaying the entire inventory.

**Solution**:

***Solution 1*:** The solution for an **Inventory Management System** uses multi-dimensional arrays for warehouse locations and a **Dictionary of Keys (DoK)** format for sparse arrays in C++.

The DoK format is often used in sparse matrices because it allows easy insertion, deletion, and access by storing only non-zero elements in a dictionary-like structure.

In this system:

* A **3D array** represents the warehouse (racks, bins, aisles).
* The **DoK (Dictionary of Keys)** format is implemented with a map for sparse storage, only keeping track of items with non-zero quantities.

**Project Overview**

**Inventory Storage**:

* Use a **3D array** for physical structure: racks, bins, and aisles.
* Implement a **sparse representation using DoK** (a map) to store non-zero quantities for each item efficiently.

**Functionality**:

* **Add or Update Item**: Add an item to a specific warehouse location.
* **Get Item Quantity**: Retrieve quantity of an item at a specific location.
* **Display Inventory**: Display all items with non-zero quantities, their locations, and quantities.
* **Remove Item**: Remove or reduce the quantity of an item.
* **Transfer Items**: Move items from one location to another.

**Data Structures**:

* tuple<int, int, int> for location keys (rack, bin, aisle).
* map< tuple<int, int, int>, Item> as the DoK sparse array, storing only non-zero quantities.

#include <iostream>

#include <map>

#include <tuple>

#include <string>

using namespace std;

struct Item {

string name;

int quantity;

Item(string n = "", int q = 0) : name(n), quantity(q) {}

};

class Inventory {

private:

int racks;

int bins;

int aisles;

// DoK representation for sparse storage

map<tuple<int, int, int>, Item> sparseInventory;

public:

// Constructor to initialize dimensions

Inventory(int r, int b, int a) : racks(r), bins(b), aisles(a) {}

// Method to add or update an item

void addItem(int rack, int bin, int aisle, const string& name, int quantity) {

if (quantity != 0) {

sparseInventory[{rack, bin, aisle}] = Item(name, quantity);

} else {

sparseInventory.erase({rack, bin, aisle});

}

}

// Method to get the quantity of an item at a specific location

int getQuantity(int rack, int bin, int aisle) {

auto it = sparseInventory.find({rack, bin, aisle});

if (it != sparseInventory.end()) {

return it->second.quantity;

}

return 0; // Zero quantity for non-existent items

}

// Remove an item or reduce its quantity at a location

void removeItem(int rack, int bin, int aisle, int quantity) {

auto it = sparseInventory.find({rack, bin, aisle});

if (it != sparseInventory.end()) {

it->second.quantity -= quantity;

if (it->second.quantity <= 0) {

sparseInventory.erase(it);

}

}

}

// Display all non-zero items in the inventory

void displayInventory() {

cout << "Inventory List:"<<endl;

for (const auto& entry : sparseInventory) {

auto [rack, bin, aisle] = entry.first;

const Item& item = entry.second;

cout << "Location (" << rack << ", " << bin << ", " << aisle << "): "

<< item.name << " - Quantity: " << item.quantity << endl;

}

}

// Transfer items between locations

void transferItem(int fromRack, int fromBin, int fromAisle,

int toRack, int toBin, int toAisle, int quantity) {

int currentQuantity = getQuantity(fromRack, fromBin, fromAisle);

if (currentQuantity >= quantity) {

Item item = sparseInventory[{fromRack, fromBin, fromAisle}];

removeItem(fromRack, fromBin, fromAisle, quantity);

addItem(toRack, toBin, toAisle, item.name, quantity);

cout << "Transferred " << quantity << " of " << item.name

<< " from (" << fromRack << ", " << fromAisle << ", " << fromBin

<< ") to (" << toRack << ", " << toBin << ", " << toAisle << ")"<<endl;

} else {

cout << "Not enough quantity to transfer"<<endl;

}

}

};

// Helper function to populate inventory for testing

void populateInventory(Inventory& inv) {

inv.addItem(0, 1, 2, "Screws", 150);

inv.addItem(0, 1, 3, "Nails", 200);

inv.addItem(1, 2, 0, "Bolts", 75);

}

int main() {

int racks = 2, bins = 3, aisles = 4;

Inventory inv(racks, bins, aisles);

populateInventory(inv);

inv.displayInventory();

cout << "Adding 100 Washers at (1, 2, 1):"<<endl;

inv.addItem(1, 2, 1, "Washers", 100);

inv.displayInventory();

cout << endl<<"Transferring 50 Screws from (0, 1, 2) to (1, 2, 1):"<<endl;

inv.transferItem(0, 1, 2, 1, 2, 1, 50);

inv.displayInventory();

return 0;

}

**Explanation of Code**

**Item Structure**:

* + Item stores each inventory item’s name and quantity.

**Inventory Class:**

* **Attributes**: racks, bins, and aisles specify the warehouse structure. sparseInventory uses a DoK (Dictionary of Keys) structure as a map to store non-zero items by location.
* **addItem**: Adds or updates an item at a specified location. If quantity is zero, the item is removed from sparseInventory.
* **getQuantity**: Fetches the quantity of an item at a given location.
* **removeItem**: Decreases item quantity at a location. If quantity reaches zero, the item is removed.
* **displayInventory**: Displays all non-zero items, showing locations and quantities.
* **transferItem**: Moves items from one location to another. If insufficient quantity, it prints an error message.

**populateInventory**:

* Populates the inventory with a few items for testing purposes.

**Key Points**

**Efficient Memory Usage**:

* The DoK format (using std::map) is memory efficient, storing only non-zero quantities and skipping empty slots.

**Flexible Storage**:

* The use of tuple<int, int, int> as keys allows us to flexibly address multiple storage dimensions.

**Enhanced Readability**:

* The displayInventory method only shows non-zero items, making it easier to track and review inventory levels.

This project highlights efficient use of memory through sparse representation, effective inventory management, and a robust application of advanced C++ data structures.

***Solution 2***: **Inventory Management System** using a **vector of vectors of vectors** (vector< vector< vector<Item>>>) instead of a map. This approach works well for scenarios where the warehouse has mostly non-sparse inventory, and memory isn’t as much of a concern as it is with the sparse representation.

The warehouse is modelled as a **3D vector** *vector< vector< vector<Item>>>* with:

* Racks
* Bins
* Aisles

Each location in the 3D array can store an Item.

#include <iostream>

#include <vector>

#include <string>

using namespace std;

struct Item {

string name;

int quantity;

Item(string n = "", int q = 0) : name(n), quantity(q) {}

};

class Inventory {

private:

int racks;

int bins;

int aisles;

// 3D vector to represent the warehouse

vector< vector<vector<Item>>> warehouse;

public:

// Constructor to initialize dimensions

Inventory(int r, int b, int a) : racks(r), bins(b), aisles(a) {

warehouse.resize(racks, vector<vector<Item>>(

bins, vector<Item>(aisles, Item())));

}

// Method to add or update an item

void addItem(int rack, int bin, int aisle, const string& name, int quantity) {

if (rack >= racks || aisle >= aisles || bin >= bins || rack < 0 || bin < 0 || aisle < 0) {

cout << "Invalid location specified."<<endl;

return;

}

warehouse[rack][bin][aisle] = Item(name, quantity);

}

// Method to get the quantity of an item at a specific location

int getQuantity(int rack, int bin, int aisle) {

if (rack >= racks || bin >= bins || aisle >= aisles || rack < 0 || bin < 0 || aisle < 0) {

cout << "Invalid location specified."<<endl;

return 0;

}

return warehouse[rack][bin][aisle].quantity;

}

// Remove an item or reduce its quantity at a location

void removeItem(int rack, int bin, int aisle, int quantity) {

if (rack >= racks || bin >= bins || aisle >= aisles || rack < 0 || bin < 0 || aisle < 0) {

cout << "Invalid location specified."<<endl;

return;

}

Item& item = warehouse[rack][bin][aisle];

item.quantity -= quantity;

if (item.quantity <= 0) {

item = Item(); // Reset the item

}

}

// Display all non-empty items in the inventory

void displayInventory() {

cout << "Inventory List:\n";

for (int r = 0; r < racks; ++r) {

for (int b = 0; b < bins; ++b) {

for (int a = 0; a < aisles; ++a) {

const Item& item = warehouse[r][b][a];

if (item.quantity > 0) {

cout << "Location (" << r << ", " << b << ", " << a << "): "

<< item.name << " - Quantity: " << item.quantity << endl;

}

}

}

}

}

// Transfer items between locations

void transferItem(int fromRack, int fromBin, int fromAisle,

int toRack, int toBin, int toAisle, int quantity) {

if (getQuantity(fromRack, fromBin, fromAisle) >= quantity)

{

Item& fromItem = warehouse[fromRack][fromAisle][fromBin];

Item& toItem = warehouse[toRack][toBin][toAisle];

if (toItem.name.empty() || toItem.name == fromItem.name) {

// Add quantity to destination

toItem.name = fromItem.name;

toItem.quantity += quantity;

// Subtract quantity from source

fromItem.quantity -= quantity;

if (fromItem.quantity == 0) {

fromItem = Item(); // Reset source

}

cout << "Transferred " << quantity << " of " << toItem.name

<< " from (" << fromRack << ", " << fromBin << ", " << fromAisle

<< ") to (" << toRack << ", " << toBin << ", " << toAisle << ")"<<endl;

} else {

cout << "Destination bin contains a different item."<<endl;

}

} else {

cout << "Not enough quantity to transfer."<<endl;

}

}

};

// Helper function to populate inventory for testing

void populateInventory(Inventory& inv) {

inv.addItem(0, 1, 2, "Screws", 150);

inv.addItem(0, 1, 3, "Nails", 200);

inv.addItem(1, 2, 0, "Bolts", 75);

}

int main() {

int racks = 2, bins = 3, aisles = 4;

Inventory inv(racks, bins, aisles);

populateInventory(inv);

inv.displayInventory();

cout << "\nAdding 100 Washers at (1, 2, 1):"<<endl;

inv.addItem(1, 2, 1, "Washers", 100);

inv.displayInventory();

cout << "\nTransferring 50 Screws from (0, 1, 2) to (1, 2, 1): "<<endl;

inv.transferItem(0, 1, 2, 1, 2, 1, 50);

inv.displayInventory();

return 0;

}

**Explanation of Code**

1. **Data Structure**:

* A 3D vector vector<vector< vector<Item>>> warehouse represents the inventory.
* Each Item at a location is directly accessible using warehouse[rack][bin][aisle].

1. **Add/Update Item:**

* The addItem method updates the 3D vector at the specified location.

1. **Transfer Items:**

* Ensures items can be moved between bins without overwriting unrelated items.

1. **Display Inventory:**

* Loops through the 3D array to find and print all non-zero inventory entries.

1. **Remove Item:**

* Resets an Item to default (Item("", 0)) when its quantity reaches zero.

**Key Differences**

**Memory Usage**:

* The vector approach allocates memory for all slots (even if unused), unlike sparse arrays.
* Works best for scenarios with dense inventory data.

**Complexity**:

* Access to items is faster with direct indexing.

**Use Cases**:

* Suitable for systems where most storage slots are utilized. Sparse representation is better for large, mostly empty warehouses.

***Solution 3***: Using structures.

// Online C++ compiler to run C++ program online

#include <iostream>

#include <string>

#include <map>

using namespace std;

//Structure for inventory items

struct Item

{

int itemId;

string itemName;

int quantity;

double price;

Item(int \_id = -1, string \_name = "", int \_quantity = 0,

double \_price = 0.0) : itemId(\_id), itemName(\_name), quantity(\_quantity), price(\_price)

{}

};

//Structure for Rack which contains items

//Contains a map<int, Item> for storing multiple items

struct Rack{

map<int, Item> items; //Map of items by ID within each rack

};

//Struct Bin contains a map<int, Rack> for storing multiple Racks

struct Bin{

map<int, Rack> racks; //Map of racks by rack number within each bin

};

//Struct Aisle contains a map<int, Bin> for storing multiple Bins

struct Aisle{

map<int, Bin> bins; //map of bins by bin number within each aisle

};

//Struct Warehouse contains a map<int, Aisle> for storing multiple Aisles

struct Warehouse{

string wName;

map<int, Aisle> aisles; //map of aisles by aisle number within each warehouse

};

//Inventory class using sparse arrays having add items, remove items, check stock, //display inventory, transfer items, low stock alert functionality

class InventorySystem

{

//Map to store items at specified positions

//map<pair<int, int>, Item> inventory;

map<int, Warehouse> warehouses; //Map to store each warehouse by its id

int lowStockThreshhold = 10;

public:

//Add / update item in a warehouse, aisle, bin, rack

void addUpdateItem(int wareId, int aisle, int bin, int rack, int id, string name, int qty, double price)

{

//pair<int, int> position = make\_pair(row, col);

warehouses[wareId].aisles[aisle].bins[bin].racks[rack].items[id] = Item(id, name, qty, price);

//inventory[position] = Item(id, name, qty, price);

cout<<"Item added / updated in warehouse ("<<wareId <<", Aisle "<<aisle<<", Bin "<<bin <<", Rack "<<rack<<endl;

checkLowStock(wareId, aisle, bin, rack, id);

}

//Remove item from specific warehouse, aisle, bin, rack

void removeItem(int wareId, int aisle, int bin, int rack, int id)

{

if(warehouses[wareId].aisles[aisle].bins[bin].racks[rack].items.erase(id))

{

cout<<"Item removed from warehouse ("<<wareId <<", Aisle "<<aisle<<", Bin "<<bin <<", Rack "<<rack<<endl;

}

else

{

cout<<"No item found in warehouse ("<<wareId <<", Aisle "<<aisle<<", Bin "<<bin <<", Rack "<<rack<<endl;

}

}

//Check stock for an item at specified location

void checkStock(int wareId, int aisle, int bin, int rack, int id) const

{

auto wareIt = warehouses.find(wareId);

if(wareIt == warehouses.end())

{

cout<<"Warehouse "<<wareId <<" does not exist"<<endl;

return;

}

const Warehouse& ware = wareIt->second;

//Check if aisle exist in warehouse

const auto& aisleIt = ware.aisles.find(aisle);

if(aisleIt == ware.aisles.end())

{

cout<<"Aisle "<<aisle<<" does not exist in Warehouse "<<wareId<<endl;

return;

}

//check if bin exists in aisle

const auto& binIt = aisleIt->second.bins.find(bin);

if(binIt == aisleIt->second.bins.end())

{

cout<<"Bin "<<bin<<" does not exist in Aisle "<<aisle<<endl;

return;

}

//check if rack exists in bin

const auto& rackIt = binIt->second.racks.find(rack);

if(rackIt == binIt->second.racks.end())

{

cout<<"Rack "<<rack<<" does not exist in Bin "<<bin<<endl;

return;

}

const auto& itemIt = rackIt->second.items.find(id);

if(itemIt == rackIt->second.items.end())

{

const Item& item = itemIt->second;

cout<<"Stock in warehouse "<<wareId<<", Aisle "<<aisle<<", Bin "<<bin<<", Rack "<<rack<<": "<<item.itemName<<" (ID: "<<item.itemId<<"), Quantity: "<<item.quantity<<endl;

}

else

{

cout<<"No items found in warehouse "<<wareId<<", Aisle "<<aisle<<", Bin "<<bin<<", Rack "<<rack<<endl;

}

}

//Display all items in inventory

void displayInventory(int wareId) const

{

auto wareIt = warehouses.find(wareId);

if(wareIt == warehouses.end())

{

cout<<"Warehouse "<<wareId<<" doesnot exist"<<endl;

return;

}

const Warehouse& ware = wareIt->second;

cout<<"Inventory for warehouse "<<wareId<<" ("<<ware.wName<<")"<<endl;

for(const auto& [aisleId, aisle] : ware.aisles)

{

for(const auto& [binId, bin]:aisle.bins)

{

for(const auto& [rackId, rack]: bin.racks)

{

for(const auto& [itemId, item]:rack.items)

{

cout<<"Aisle "<<aisleId<<", Bin "<<binId<<", Rack "<<rackId<<": ";

cout<<"Item id: "<<item.itemId <<", Name"<<item.itemName;

cout<<", Quantity "<<item.quantity<<", Price "<<item.price<<endl;

}

}

}

}

}

//Low stock alert

void checkLowStock(int wareId, int aisle, int bin, int rack, int id)

{

auto wareIt = warehouses.find(wareId);

if(wareIt != warehouses.end())

{

const auto& itemIt = wareIt->second.aisles[aisle].bins[bin].racks[rack].items.find(id);

if(itemIt != wareIt->second.aisles[aisle].bins[bin].racks[rack].items.end() &&

itemIt->second.quantity < lowStockThreshhold)

{

cout<<"Low Stock Alert: "<<itemIt->second.itemName<<" in warehouse "<<wareId<<", Aisle "<<aisle<<", Bin "<<bin<<", Rack "<<rack<<", Quantity: "<<itemIt->second.quantity<< endl;

}

}

}

//Set low stock threshold

void setLowStockThreshhold(int threshhold)

{

lowStockThreshhold = threshhold;

}

};

int main()

{

InventorySystem invent;

invent.setLowStockThreshhold(10);

//addupdateitem(WareId, Aisle id, bin id, rack id, item id, item name, quantity, price)

invent.addUpdateItem(1, 2, 0, 1, 1003, "Macbook Pro", 5, 123000.00);

invent.addUpdateItem(1, 3, 0, 1, 1004, "Apple Smart Watch 5", 20, 47000.00);

invent.addUpdateItem(1, 3, 0, 2, 1005, "Apple iPhone 15", 50, 85500.00);

//Display inventory present in warehouse

invent.displayInventory(1);

//int wareId, int aisle, int bin, int rack, int itemid

invent.removeItem(1, 3, 0, 1, 1003);

//int wareId, int aisle, int bin, int rack, int itemid

invent.checkLowStock(1, 2, 0, 1, 1003);

return 0;

}