# Project 4

## Group Members

Abhishek Gupta (axg137230) and Manuj Singh (mxs135630)

## Objective

To implement multi-dimensional search over collection of items.

## Problem Statement

Given an item with following attributes:

1. Id: id of an item is unique. It is of long type.
2. Names: set of names associated with an item. It is an array of long type.
3. Price: price of an item. It is of double type.

We need to create a data structure that can support following operations efficiently:

1. Insertion: Inserting an item with given attributes or updating attributes of existing item for a given id.
2. Find item by id: For a given id, get the price of that item.
3. Delete item by id: For a given id, delete the item.
4. Find min price by name: Find the minimum price among all the items containing the given name.
5. Find max price by name: Find the maximum price among all the items containing the given name.
6. Find price range by name: For given range of price [low, high] return the number of items containing the given name.
7. Hike the price of items by id: For given range of ids [low, high] increase price of all items by r %.

## Implementation

To make each of these operation efficient we need to create parallel data structures storing reference of items. Item will be created only once in the memory but will be referenced from multiple data structures. Insertion, deletion and update of item will require re positioning of item reference in various data structures. Next section explain the various data structure used for the purpose.

### Data Structures

Three data structures are used to achieve all the operations efficiently.

#### <Id, item>: (idSearch)

This hash map stores the item references hashed by id. For collision resolution technique used is separate chaining.

#### <Name, >: (nameSearch)

This hash map stores tree containing item references hashed by name. For collision resolution technique used is separate chaining.

#### <Price, item>: (Value in the Hash Map nameSearch)

This tree which is stored as value in the hash map “nameSearch” stores set of items sorted in order of price. Each node of tree contains price as key and a linked list of items as value. Linked list in a node contains referenced of items having same price (handling duplicate keys in the tree).

#### <Id, id>: (keyTree)

This RB tree stores ids of all the items in sorted.

### Operation and its Efficiency

Let’s discuss efficiency of each operation:

#### Insert (id, names, price)

Case 1- Inserting new item: Create a new object of item type with given attributed. Insert the reference into idSearch and nameSearch and just add id to keyTree.

|  |  |
| --- | --- |
| **Data Structure** | **Insertion Complexity** |
| idSearch | O(1): inserting into hash map. |
| nameSearch | O(1): finding location in the hash table and  O(nlogM): inserting item reference into RB tree of size M, mapped to each of item’s name. (n is the number of names in the item) |
| keyTree | O(logN): inserting into RB tree containing ids of all the N items present. |

Case 2- Updating old item for a given id: Get the oldItem. Delete its references from hash map namesSearch corresponding to each name its holding. Update the price and names of the old item and insert this reference into nameSearch for each of its new name.

|  |  |
| --- | --- |
| **Data Structure** | **Insertion Complexity** |
| idSearch | O(1): getting old item reference |
| nameSearch | O(nlogM): deleting item reference from RB tree of size M, mapped to each of its old names. (n is the number of old names in the item)  O(nlogM): inserting item reference into RB tree of size M, mapped to each of item’s new name. (n is the number of new names in the item) |

#### Find (id)

Find the price of an item for given id needs to access only one data structure i.e. idSearch hash map.

|  |  |
| --- | --- |
| **Data Structure** | **Insertion Complexity** |
| idSearch | O(1): getting item reference by id and accessing its stored price |

#### Delete (id)

Deleting an item requires deletion of its references across all the data structures. It access all three data structures.

|  |  |
| --- | --- |
| **Data Structure** | **Insertion Complexity** |
| idSearch | O(1): deleting from the hash map. |
| nameSearch | O(1): finding location in the hash table and  O(nlogM): deleting item reference from RB tree of size M, mapped to each of item’s name. (n is the number of names in the item) |
| keyTree | O(logN): deleting from RB tree containing ids of all the N items present. |

#### FindMinPrice (name)

This operation requires access to only one data structure nameSearch.

|  |  |
| --- | --- |
| **Data Structure** | **Insertion Complexity** |
| nameSearch | O(1): getting the tree mapped to the given name.  O(logM): finding min price in the tree of size M (key in the tree is price and all the item references in this tree will contain given name) |

#### FindMaxPrice (name)

This operation requires access to only one data structure nameSearch.

|  |  |
| --- | --- |
| **Data Structure** | **Insertion Complexity** |
| nameSearch | O(1): getting the tree mapped to the given name.  O(logM): finding max price in the tree of size M (key in the tree is price and all the item references in this tree will contain given name) |

#### FindPriceRange (name, lowPrice, highPrice)

This operation requires access to only one data structure nameSearch.

|  |  |
| --- | --- |
| **Data Structure** | **Insertion Complexity** |
| nameSearch | O(1): getting the tree mapped to the given name.  O(MlogM): finding all the keys between lowPrice and highPrice in the tree of size M. Get the value of the node for each key which is a linkedList, get its size and add to the total. (key in the tree is price and value is linked list of items having same price.) |

#### PriceHike (lowId, highId, rate)

This operation required access to all three data structures.

|  |  |
| --- | --- |
| **Data Structure** | **Insertion Complexity** |
| keyTree | O(N): getting all ids within given range |
| idSearch | O(N): getting item corresponding to each id in the range.  O(N): putting item with updated price. |
| nameSearch | O(1): finding location in the hash table and  O(nlogM): deleting item reference from RB tree of size M, mapped to each of item’s name. (n is the number of names in the item)  O(nlogM): inserting item reference with updated price into RB tree of size M, mapped to each of item’s name. (n is the number of names in the item) |

## References

1. The implementation of Red black tree and hash map using separate chaining is taken from <http://algs4.cs.princeton.edu/home/>