CS 6301.016 : Implementation of advanced DS&A

Project 6

Extending Data Structures : Multi-Dimensional Search

Arjun Gopal
10/19/2014

Project Specification

Multi-dimensional search: Consider the web site of a seller like Amazon. They carry tens of thousands of products, and each product has many attributes (Name, Size, Description, Keywords, Manufacturer, Price, etc.). The search engine allows users to specify attributes of products that they are seeking, and shows products that have most of those attributes. To make search efficient, the data is organized using appropriate data structures, such as balanced trees. But, if products are organized by Name, how can search by price implemented efficiently? The solution, called indexing in databases, is to create a new set of references to the objects for each search field, and organize them to implement search operations on that field efficiently. As the objects change, these access structures have to be kept consistent.

In this project, each object has 3 attributes: id (long int), name (one or more long ints), and price (dollars and cents). The following operations are supported:

a. Insert(id,price,name): insert a new item. If an entry with the same id already exists, its name and price are replaced by the new values. If name is empty, then just the price is updated. Returns 1 if the item is new, and 0 otherwise.

b. Find(id): return price of item with given id (or 0, if not found).

c. Delete(id): delete item from storage. Returns the sum of the long ints that are in the name of the item deleted (or 0, if such an id did not exist).

d. FindMinPrice(n): given a long int n, find items whose name contains n (exact match with one of the long ints in the item's name), and return lowest price of those items.

e. FindMaxPrice(n): given a long int n, find items whose name contains n, and return highest price of those items.

f. FindPriceRange(n,low,high): given a long int n, find the number of items whose name contains n, and their prices fall within the given range, [low, high].

g. PriceHike(l,h,r): increase the price of every product, whose id is in the range [l,h], by r% Discard any fractional pennies in the new prices of items. Returns the sum of the net increases of the prices.

Implement the operations using data structures that are best suited for the problem. You may download any implementations of standard data structures from the web, and modify the code as needed.

Input specification:
Initially, the store is empty, and there are no items. The input contains a sequence of lines (use test sets with millions of lines). Lines starting with "#" are comments. Other lines have one operation per line: name of the operation, followed by parameters needed for

that operation (separated by spaces). Lines with Insert operation will have a "0" at the end, that is not part of the name. The output is a single number, which is the sum of the following values obtained by the algorithm as it processes the input.

```
Sample input:
Insert 22 19.97 475 1238 9742 0
# New item with id=22, price="$19.97", name="475 1238 9742"
# Return: 1
#
Insert 12 96.92 44 109 0
# Second item with id=12, price="96.92", name="44 109"
# Return: 1
#
Insert 37 47.44 109 475 694 88 0
# Another item with id=37, price="47.44", name="109 475 694 88"
# Return: 1
#
PriceHike 10 22 10
# 10% price increase for id=12 and id=22
# New price of 12: 106.61, Old price = 96.92.  Net increase = 9.69
# New price of 22: 21.96.  Old price = 19.97.  Net increase = 1.99
# Return: 11.68  (sum of 9.69 and 1.99)
#
FindMaxPrice 475
# Return: 47.44 (id of items considered: 22, 37)
#
Delete 37
# Return: 1366 (=109+475+694+88)
#
FindMaxPrice 475
# Return: 21.96 (id of items considered: 22)
#


Output:
1450.08
```

Introduction

The Aim of the project is to combine and extend the data structures learned in class to solve a real world problem. May be by using a single tree structure to store all the data will help to achieve the result but it will not be efficient from the performance aspect. The main idea was to use multiple data structures to achieve efficient solution with less redundant data.

Classes

1. RedBlackBST
    a. Red Black Tree Implementation from Princeton. References Provided
2. SeperateChainingHashST
    a. Hash Table Implementation from Princeton. References Provided
3. ProductNode
    a. Representation of the Product with its ID, name and Price.
4. NameNode
    a. NameNode consists of the name and all the references to the associated Product
5. Project6
    a. The Main class with all the operations.

Implementation Details

The project is implemented in Java by extending the red black trees and hash data structure

1. Each Product is represented as an object "ProductNode" and stored in a single Red Black Tree named "idTree". The "ProductNode" has member variables which stores the name (name list) of the product, its price and its id. The Red Black tree uses top down approach for the operation and balances itself giving a good performance.
2. Each sub name is represented in "NameNode", which is basically the object that represents the name and associated "ProductNodes", in a list. The NameNodes are saved in a hash table which implements the Separate Chaining hashing.
3. When a ProductNode is inserted into the main red black tree, the name associated with this product will be hashed , NameNodes are created and inserted to the hash table.
4. There is one more hash table maintained to store all the references of the product for easy access during find, insert and delete operations.
5. The RedblackTree code base which was downloaded from the Princeton Code Project, was modified to support the operations like price hike and optimized further to do fast calculations.

Now let's see how the operations are handled using the above data structure scheme.

1. *INSERT*
    a. The input values are read from the standard input.

b. A fast search on the hash table for the Product Reference is done and if a node is present, then it can be a price update or name and price updates. In that case, we need not search for the Product in the main tree. Update can be done with the reference we got from the hash table.
c. Take all the names and create NameNodes (which stores the sub names and the Product associated with them) and put them into the hash table for names.

2. *FIND*
    a. Just check the hash table for the product, indexed by ID and update the sum with its price.
3. *DELETE*
    a. We can check in the hash table which stores the product references and if it is there, then we can delete from the tree.
4. *FINDMIN*
    a. We will check in the hash table for names and will do a linear search on the items there to find the min price
5. *FINDMAX*
    a. We will check in the hash table for names and will do a linear search on the items there to find the max price
6. *FINDPRICERANGE*
    a. We will check in the hash table for names and will do a linear search on the items there to find the price range

7. *PRICEHIKE*
    a. Traverse the tree for items within the given range and update the price of each item after calculating the hike. The code of redblacktree was modified to do an efficient range search.

In summary we have

1. *A single Red Black Tree - which stores all the Products*
2. *A hash table indexed by Id and which stores the references of nodes in the main Red Black Tree. This provides a faster access to the product with the ID. This is optional and may not be required in small test cases.*
3. *A hash table indexed by names and which stores a NameNode which have all the references to the associated Products in the main RedBlack Tree.*

Platform

The problem was coded on a 64 bit windows 8 machine. Dual Core I7 Processor. 8 GB Main Memory and 3.1 GHz Processor Speed.

IO Format

The program can be compiled and executed as follows.

```
>javac AXG145630_Project6.java
```

```
>java AXG145630_Project6
```

To terminate the program user has to give 101

Execution Statistics

These execution statistics are based on the 5 input files provided by professor

Sl No	Input	Execution Time (ms)
1	P6-in 1.txt	0
2	P6-in 2.txt	23
3	P6-in 3-1k.txt	115
4	P6-in 4-5k.txt	234
5	P6-in 5-ck.txt	821

Table 1 : Execution Statistics

Conclusion

The aim of the project was to understand extending the data structures, combining them to solve real world problems. A red black tree implementation was combined with hash data structure to solve the problem efficiently. There was no duplicate data stored and handled all the edge cases. It took 2 days for the implementation and one day for the testing.

References

1. http://algs4.cs.princeton.edu/code/ - For red black tree and separate chaining hashing.