TECNOLÓGICO DE MONTERREY
Campus Guadalajara

Basic database with three Hash Tables that use Binary Search Trees

Professor: Andrés Méndez Vázquez
Data Structures
Jorge Padilla A00570894
Luis Vargas A01630086

Introduction

In this project, we will use hash tables. The reason for this is that for a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data.

Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure, this simplifies the complexity and time to do the operations, making the system more user-friendly.

Hashing uses hash functions with search keys as parameters to generate the address of a data record.

A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records. A hash function, *h*, is a mapping function that maps all the set of search-keys *K* to the address where actual records are placed. It is a function from search keys to bucket addresses.

We will also be working with a couple of binary search trees (BST). These trees will allows us to quickly look up a record by its key. It is important to keep in mind that each search attempt cuts the number of records to search in half.

With that being said, databases typically use some other binary tree-like data structure to perform the indexing. Using a binary tree removes the requirement that the list of keys be sorted before searching.

Our final product will be a basic database management system (DBMS), and the main characteristics the DBMS must and will have are the following:

1. It is capable of inserting the different quantities in the different tables by primary ID.
2. It is capable of deleting quantities by their primary ID at each table.

3. It is capable of selecting names from the Name-Address table and see their total expenses.
4. It is capable of selecting names from the Name-Address table and see their total payments.
5. It is capable of returning the earning after expenses in a similar fashion.
6. It is capable of calculating how similar are different users by expense.

Product perspective

In static hashing, when a search-key value is provided, the hash function always computes the same address. The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. That´s why on the other hand we have what is known as dynamic hashing. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as extended hashing. Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.

Binary trees become truly useful when you balance them. This involves rotating sub-trees through their root node so that the height difference between any two sub-trees is less than or equal to 1. This is typically used in maintaining keys for an index of items.

The BST is a different way of structuring data so that it can still be binary searched, but it's easier to add and remove elements. Instead of just storing the elements contiguously from least to greatest, the data is maintained in many separate pieces, making adding an element a matter of adding a new piece of memory and linking it to existing pieces.

BSTs support everything we can get from a sorted array: efficient search, in-order forward/backwards traversal from any given element, predecessor /successor element search, and max /min queries, with the added benefit of efficient inserts and deletes. With a self-balancing binary search tree (BST), all of the above run in logarithmic time.

Software Implementation

Classes:

1) Database -- The DataBase stores related data for each table and interacts with each other for the specifications of the project
    a) insertAddress -- Inserts name values to the table
    b) getAddress -- Obtains the name values from the table
    c) removeAddress -- Removes the name values from the table
    d) insertPayment -- Inserts invoice values to the table
    e) getPayment -- Obtains the invoice values from the table
    f) removePayment -- Removes the invoice values from the table
    g) insertExpense -- Inserts item values to the table
    h) getExpense -- Obtains the item values from the table
    i) removeExpense -- Remove the item values from the table
    j) displayExpenses -- Display all the expenses a person has made using ExpensesSingleUser
    k) displaysPayments -- Display all the payments a person has made using PaymentsSingleUser
    l) displayEarnings -- Prints the earnings after expense for a person using EarningsSingleUser
    m) getDifference -- Obtains the difference expense of two users
    n) main -- Test the methods

2) Expense -- Node for the entries on the expense's BST
    a) Constructor -- Initializes all the instance variables of the class
    b) toString -- Prints the key and value of the node

3) ExpenseDirectory -- Directory of User's invoice number, item and expense
    a) BSTExpense
        i) BSTNode
            (1) Contructor -- Initializes all the instance variables of the class
            (2) toString -- Prints the key and value of the node
        ii) Constructor -- Initializes all the instance variables of the class
        iii) insert -- Root gets the method
        iv) insert -- Inserts the values
        v) getValue -- Obtains the value from the key
        vi) remove -- Removes values from the key
        vii) selectExpenses -- Creates a string with the item and expense for all the nodes in the tree
        viii) selectGetExpensesItem -- Sum all the expenses from an item for a user
    b) Constructor -- Defines the class
    c) Constructor -- Initializes all the instance variables of the class
    d) hash -- Maps the data
    e) hash -- Digests the data stored in an instance of the class into a single hash
    f) resize -- The table is resized to twice its capacity
    g) add -- Adds expense values
    h) remove -- Removes values from the key
    i) getSize -- Obtains size from the table
    j) getExpense -- Obtains the expense from the item
    k) clear -- Sets the table to null

4) Invoice -- Node for the entries on the invoice's BST
    a) Constructor -- Initializes all the instance variables of the class
    b) toString -- Prints the key and value of the node

5) InvoiceDirectory -- Directory of User's name, invoice number and payment
    a) BSTInvoice
        i) BSTNode
            (1) Constructor -- Initializes all the instance variables of the class
            (2) toString -- Prints the key and value of the node
        ii) Constructor -- Sets the root to null
        iii) insert -- Root gets the method
        iv) insert -- Inserts the values
        v) getValue -- Obtains the value from the key
        vi) remove -- Removes values from the key
        vii) selectExpensesInvoice -- Uses selectExpenses to order the expenses by invoice
        viii) selectPayments -- Creates a string with the invoice number and payment for all the nodes in the tree
        ix) selectGetPayments -- Sum all the payments from an invoice for a user
        x) selectGetExpenses -- Obtains total expenses Using selectGetExpensesItems
    b) Constructor -- Defines the class
    c) Constructor -- Initializes all the instance variables of the class
    d) hash -- Maps the data
    e) hash -- Digests the data stored in an instance of the class into a single hash
    f) resize -- The table is resize twice its capacity
    g) add -- Adds invoice values
    h) remove -- Removes invoice values from the key
    i) getSize -- Obtains size from the table
    j) getInvoice -- Obtains values from the key
    k) clear -- Sets the table to null

6) Name -- Node for the entries on the name's BST
    a) Constructor -- Initializes all the instance variables of the class
    b) toString -- Prints the key and value of the node
    c) insertInvoice -- Inserts invoice values
    d) removeInvoice -- Removes invoice value from invoice number
    e) insertExpense -- Inserts expense values
    f) removeExpense -- Removes expense value from the item

7) NameDirectory -- Directory of User's name and address with it's own BST
    a) BSTName
        i) BSTNode
            (1) Constructor-- Initializes all the instance variables of the class
            (2) toString -- Prints the key and value of the node
        ii) Constructor -- Initializes all the instance variables of the class
        iii) insert -- Root gets the method
        iv) insert -- Inserts the values
        v) getValue -- Obtains the value from the key
        vi) remove -- Removes values from the key
        vii) ExpensesSingleUser -- Uses selectExpensesInvoice to order the expenses by user
        viii) PaymentsSingleUser -- Uses selectPayments to order the payments by user
        ix) EarningsSingleUser -- Earnings result after expense for a person
        x) getTotalPayments -- Obtains total payments using selectGetPayments
        xi) getTotalExpenses -- Obtains total expenses using selectGetExpenses
    b) Constructor -- Defines the class
    c) Constructor -- Initializes all the instance variables of the class
    d) hash -- Maps the data
    e) hash -- Digests the data stored in an instance of the class into a single hash
    f) resize -- The table is resize twice its capacity
    g) add -- Adds name values
    h) remove -- Removes values from the key
    i) getSize -- Obtains size from the table
    j) getAddress -- Obtains the address from the name
    k) clear -- Sets the table to null

Conclusion

This paper has exposed, among other things, some basic ideas, algorithms, and criteria for producing hash functions. It has presented how the existing library hash functions in Java could be improved by delivering a better library hash function. Also, it has created an abstract type for hashing that allows programmers to create new hash values from other primitive hash values and provides them a method to convert hash values to integer values for use in table searching among other validation and authentication techniques which use hashing.

Furthermore, BSTs come in many shapes and the shape of the tree determines the shape of its operations. The height of the BST with *n* nodes can range from a minimum of $O\log_2(n+1)$ to a maximum of *n*.

Overall, we enjoyed the making of this project, although it was definitely not easy. At times (almost all the time) we struggled to much trying to implement a certain method of a certain class. But at the end, it was worth it since we covered very important topics on data structures seen in class this semester. With this project being done, we truly feel that we are ready for the Algorithms class next semester. Let's hope so.

References

Tutorials Point. *DBMS - Hashing*. Recovered from: https://www.tutorialspoint.com/dbms/dbms_hashing.htm

Yarovoi, Eugene. (March 11th). *Binary Search Trees Applications*. Quora. Recovered from: https://www.quora.com/What-are-some-practical-applications-of-binary-search-trees

Parlante, Nick. *Binary Trees*. Stanford CS Library. Recovered from: http://cslibrary.stanford.edu/110/BinaryTrees.html