Module 9
CS 210
Spring 2012

So far you have implemented a "where" clause in Project 6 to filter rows for various functions. Unfortunately, as written, the clause requires checking of all rows in the table which can be computationally expensive for large table. Obviously if our data were organized in the form of a binary search tree, we could perform the "where" clause much more efficiently. For a table of size ***n,*** we can say that the search has an ***asymptotic upper bound of ln n.***

Unfortunately, there is really no easy way to do this directly, since we would like to be able to add our index at any time and, when we create it, there may be many rows present in our table and their order will be unpredictable. But we can do this by creating another binary file, one for each index. Note that we can have indexes on different fields in each table, so each table can have 0, 1 or more binary file indexes. For each binary file, we will store a binary representation of each node in the binary search tree. Each node can be stored as a series of 4 ***long*** pointers which will point to the following: 1) The actual field/value (or row, as you find most convenient) in the primary table, 2) the parent node in the binary file, and 3), 4) the left and right child nodes in the binary file.

HINT: Methods that I would implement include --

NODE methods: *get, set – parent, leftchild, rightchild, value*
(Typically if you were creating a binary tree in Java, your node classes would contain fields: Parent, LeftChild, RightChild, Value – here we are abstracting the tree from a binary file, so you will be better off replacing the fields with helper methods that locate values in the binary file).

TREE methods – here is a more or less standard set of helper methods for a binary tree (most of these are recursive in nature):

select(Value v, Node root) *// v is the value searching for, root is the root of the subtree in which I am searching*
max (Node root)
min (Node root)
inOrderWalk (Node root)
reverseInOrderWalk(Node root)
successor (Node root) *// these actually only require partial implemented*
predecessor (Node root) *// for this module*
delete(Node toBeDeleted, Node root)
insert(Value to be inserted, Node root)

For this assignment you will implement the command ***DEFINE INDEX.***

1. You will need to track this on subsequent program runs so this information will need to be added to your XML file in an appropriate fashion.
2. You will need to create a binary file to store the binary search tree so that it persists between runs of your program.
3. You will need to add functionality so that inserts, updates and deletes of your primary table will result in appropriate changes to your binary search tree.
4. Because this is a test product, and because the use of a binary search tree is typically not

noted by the user, you will need to add functionality to demonstrate that your binary search tree is working. You will do this by printing out a record of the steps taken by the binary search tree whenever the field is referenced in a where clause. The user can request debug as a command line argument, “debug” given to the java command. This command will become parameter *args[0]* to the main method. Here is an example of how some typical output might look, (depending on the clause, the testing may end when the printout of the data begins) :

```
select emp where sal > 10000;
-testing row 33
-testing row 12
-testing row 44
-testing row 39
-testing row 43
NAME        AGE    SAL         STATE
smith       22     11000       WV
-testing row 41
jones       44     13000       WV
-testing row 42
wilson      99     100000      WV
```