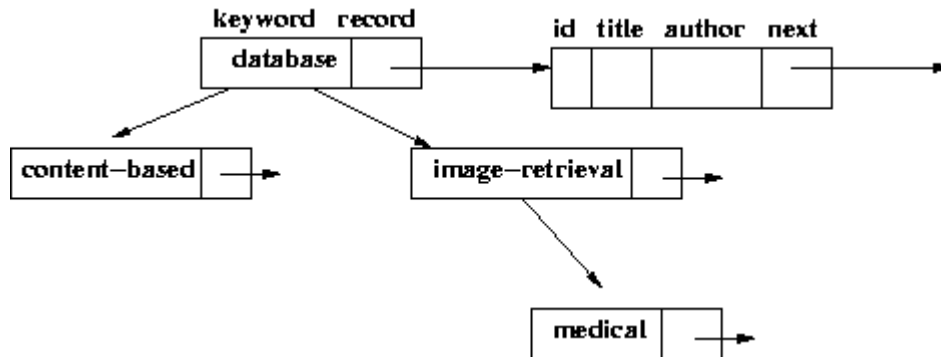# CSE 373 Winter 2015 HW3: Binary Search Trees: Keyword Search

Background

Information retrieval systems allow users to enter keywords and retrieve articles that have those keywords associated with them. For example, once a student named Yi Li wrote a paper called, "Object Class Recognition using Images of Abstract Regions," and included the following keywords: 'object recognition', 'abstract regions', 'mixture models', and 'EM algorithm'. If someone does a search for all articles about the EM algorithm, this paper (and many others) will be retrieved.

Assignment

You are to implement a binary search tree and use it to store and retrieve articles. The tree will be sorted by `keyword`, and each node will contain an unordered linked list of `Record` objects which contain information about each article that corresponds to that `keyword`. This image shows the idea:

Getting started: The necessary files are available at `http://www.cs.washington.edu/education/courses/cse373/15wi/homework/hw03/hw3files.zip`. Included in this archive are:

- A **Data file** which contains records to be read into the data structure.

- `Record.java`: The "Record" class will be the objects stored in the value for each keyword in the tree. This class also has a "next" pointer which provides the structure for the linked list. Objects of this type will be the `value` of each node in your search tree. This code should not be modified.

- `Test.java`: This code performs reading from the data file as well as allowing test operations of your binary search tree. Performing changes to this file can be done to test particular cases, but this is for your benefit, since it will not be collected. The code provided will print the contents of the tree in inorder,

which is alphabetical order. At each node of the tree, it will print the key word and then the titles of all the records in the list that you have created at that node. The test code also performs a few deletions and checks the result to ensure that `delete()` works correctly.

- `bst.java` This file contains the basic shell for the data structure we will ask you to implement in this assignment. All functions that you will be asked to implement are marked with a `\\TODO` comment and are listed below with their expected operation. You will need private functions for implementing these recursively.

  - Node constructor: This function should initialize a record with keyword 'k'. It will not require the other fields to be set because every Node construction will be updated either by directly modifying the children or by performing an `update()` to add a record to its linked list.

  - Node `update(Record r)`: This function should add the Record r to the linked list for a particular keyword. You should add new Records to the front of the list.

  - `insert(String keyword, FileData fd)`: This function includes code that turns the `FileData fd` into a Record `recordToAdd`. The function should insert `recordToAdd` to the node of keyword. If keyword is not in the tree, it should create a node.

  - `contains(String keyword)`: This function should return true if the keyword `keyword` is in the tree.

  - `get_records(String keyword)`: This function returns the linked list of Records associated with a given String `keyword`.

  - `delete(String keyword)`: This function removes the node associated with the input string `keyword`. If no such node exists, the code should do nothing.

Grading

Submission for this assignment only requires `bst.java` to be sent to the class dropbox. The datafile provided will be used for grading, so feel free to edit `test.java` in any way that will make you confident that your binary search tree performs as a binary search tree is supposed to, that is:

1. Each node satisfies the binary search tree property that its key is greater than the key of its left child and less than the key of its right child

2. Insertions and deletions are done correctly and do not violate the binary search tree property.

3. Empty tree situations are handled properly.

4. All titles for a given key word are placed in the list at the node for that key word; they should be inserted at the BEGINNING of the list.

Each of the six functions will be worth 5 points if implemented correctly. Additionally, there will be 5 points awarded for readability of code and comments for a total of 35 points.

Extra Credit: 10 points

Rewrite `BST.java` as a self-balancing AVL tree. This will require coding rotation methods that will keep the tree balanced if an insert will result in an unbalanced tree. Your code does not need to incorporate deletions. A description of the AVL tree and its rotations begins on page 123 of the Weiss textbook. Completing this extra credit will require adding several attributes to the Node and BST tree provided. If you plan to complete the extra credit, please submit the AVL section of your submission as `avl.java` in addition to the `bst.java` required in the regular portion of the homework.