Department of Computer Science and Software Engineering
Concordia University

COMP 352: Data Structure and Algorithms

Summer 2016

Assignment 3

Due date and time: Monday, June 13th, 2016 by 11:50 pm

Important Note:

Since solutions of this assignment will be discussed during the tutorials/PODs of the last week of classes (week of June 12), no deadline extension will be granted and no late assignments will be accepted (not even with a penalty!)

Written Questions (60 marks):

Q.1

a) Draw the min-heap that results from running the bottom-up heap construction algorithm (from Section §9.3. of the textbook and as described in class) on the following list of values:

17 5 20 33 41 30 28 55 17 26 35 19 11 14 60

Starting from the bottom layer, use the values from left to right as specified above. Show intermediate steps and the final tree representing the min-heap.

b) Afterwards perform the operation removeMin() 3 times and show the resulting min-heap after each step.

c) Create again a min-heap using the list of values from part a) of this question but this time you have to insert these values step by step using the order from left to right as shown in part a). Show the tree after each step and the final tree representing the min-heap.

Q.2

Assume a hash table utilizes an array of 11 elements and that collisions are handled by separate chaining. Considering the hash function is defined as: *h(k)=k mod 11*.

a) Draw the contents of the table after inserting elements with the following keys:
41, 121, 203, 172, 98, 113, 27, 12, 87, 202, 91, 85, 162, 72, 30, 74, 81, 49.
b) What is the maximum number of collisions caused by the above insertions?

Q.3

To reduce the maximum number of collisions in the hash table described in Question 2 above, someone proposed the use of a larger array of 13 elements (this is roughly 18% bigger) and of course modifying the hash function to: *h(k)=k mod 13*. The idea was to reduce the *load factor* and hence the number of collisions.
Does this proposal hold any validity to it? If yes, indicate why such modifications would actually reduce the number of collisions. If no, indicate clearly the reasons you believe/think that such proposal is senseless.

Q.4

a) Draw the 13-entry hash table that results from using the hash function *h(i) = (7i + 3) mod 13* to hash the keys 31, 45, 14, 89, 24, 95, 12, 38, 27, 16, and 25 assuming collisions are handled.

b) What is the result of part a) of this question, assuming collisions are handled by linear probing?

c) What is the result of part a) when collisions are handled by double hashing using the secondary hash function *h'(k) = 7 – (k mod 7)*?

Q.5

Consider a hash table of size M = 5. If items with keys k = 17; 10; 20; 13 are inserted in that order, draw the resulting hash table if we resolve collisions using:

a) Separate chaining with *h(k) = (k + 2) mod 5*.
b) Linear probing with *h(k) = (k + 2) mod 5*.
c) Double hashing with *h1(k) = (k + 2) mod 5* and *h2(k) = 1 + (k mod 4)*.

Q.6

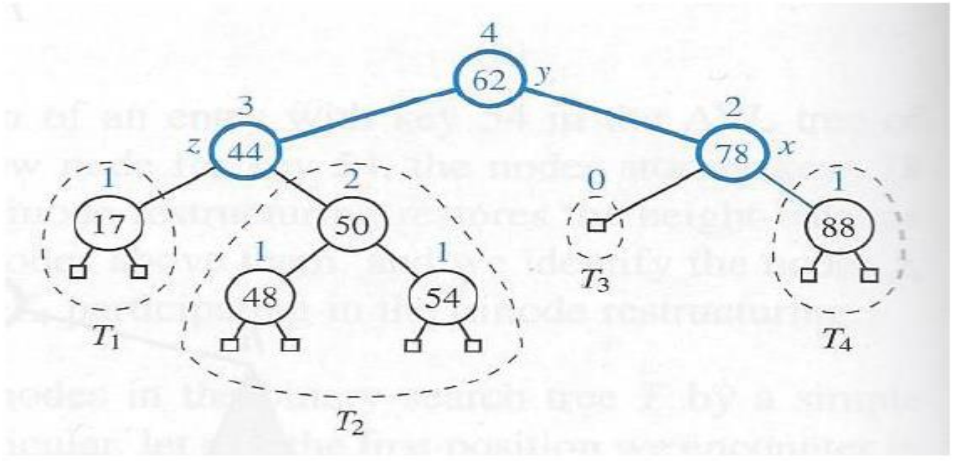
Draw the binary search tree whose elements are inserted in the following order:
50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95

Q.7

One hundred integer elements are chosen at random and inserted into: 1) a sorted linked list, 2) an array based linked list, and 3) a binary search tree. Describe the efficiency of searching for an element in each structure, in terms of Big-O.

Q.8

Note: Part a) and b) of this question are independent.
a) Draw the AVL tree resulting from the insertion of an entry with key 52 in the AVL tree shown below.
b) Draw the AVL tree resulting from the removal of the entry with key 62 in the AVL tree shown below.



<u>Q.9</u>

Describe an efficient algorithm for computing the height of a given AVL tree. Your algorithm should run in time *O(log n)* on an AVL tree of size *n*. In the pseudocode, use the following terminology: `T.left`, `T.right`, and `T.parent` indicate the left child, right child, and parent of a node T and `T.balance` indicates its balance factor (-1, 0, or 1).
For example if T is the root we have `T.parent=nil` and if T is a leaf we have `T.left` and `T.right` equal to nil. The input is the root of the AVL tree. Justify correctness of the algorithm and provide a brief justification of the runtime.

<u>Q.10</u>

Given a sequence *S* of *n* elements on which a total order relation is defined, describe an efficient method for determining whether there are two equal elements in *S*.
What is the running time of your algorithm?

<u>Q.11</u>

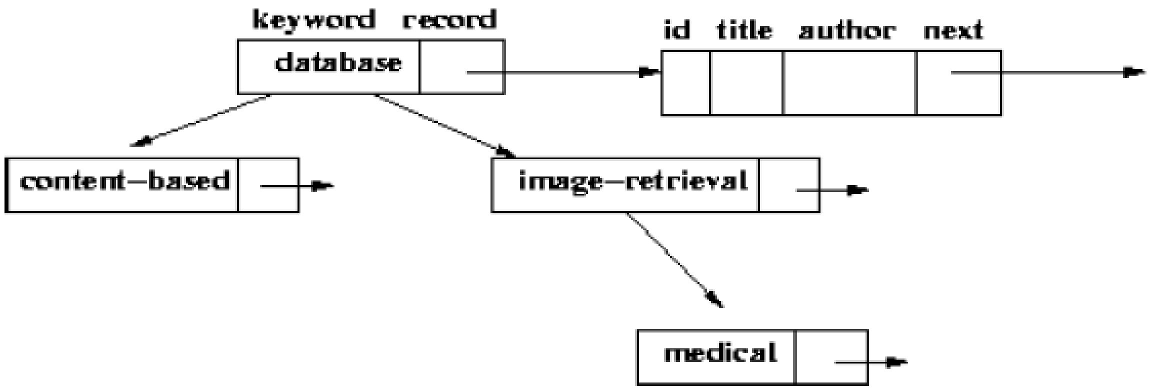
Given the following elements
29 38 74 78 24 75 42 33 21 62 18 77 30 16
Trace the steps when sorting these values into ascending order using:

a) Merge Sort,
b) Quick Sort (using (middle + 1) element as pivot point),
c) Bucket Sort – We know that the numbers are less than 99 and there are 10 buckets,
d) Radix Sort.

Programming Question (40 marks):

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.

PQ1: Your task in this part is 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. The image below shows the idea:



The necessary files below are zipped and available on Moodle:

- 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 of 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 methods 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 methods for implementing these recursively.
  - Node constructor: This method 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 method 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 method includes code that turns the `FileData fd` into a Record `recordToAdd`. The method should insert `recordToAdd` to the node of keyword. If keyword is not in the tree, it should create a node.
- `contains(String keyword)`: This method should return true if the keyword `keyword` is in the tree.
- `get_records(String keyword)`: This method returns the linked list of Records associated with a given String keyword.
- `delete(String keyword)`: This method removes the node associated with the input string `keyword`. If no such node exists, the code should do nothing.

Submission for this programming part only requires `bst.java` to be sent to. The datafile provided will be used for your demo 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.

PQ2: 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. In this question you are required to add several attributes to the Node and BST tree provided in the first question. For this question, you need to submit your `avl.java` file.

Both the written part and the programming part must be done either individually or in a team of two (max) students (no groups are permitted). Submit all your answers to written questions in PDF (no scans of handwriting) or text formats only. Please be concise and brief (less than ¼ of a page for each question) in your answers. For the Java programs, you must submit the source files together with the compiled executables. The solutions to all the questions should be zipped together into one .zip or .tar.gz file and submitted via Moodle under Assignment 3_DropBox.