Department of Computer Science and
Software Engineering
Concordia University

COMP 352: Data Structure and Algorithms
Fall 2020

Assignment # 4

Due date and time: December 5th , 2020
by 10:00 AM (MORNING)
The due date is sharp and strict. NO EXTENSION WILL BE ALLOWED BEYOND THIS TIME!

Part 1: Written Questions (50 marks):

Assignment MUST be typed. However, you are allowed to hand-draw any images and incorporate them into your typed submission. Hand-drawn images must be very clear; otherwise they will be discarded by the markers. All text must be typed and not hand-written.

Question 1

Assume the utilization of *linear probing* for hash-tables. To enhance the complexity of the operations performed on the table, a special *AVAILABLE* object is used. Assuming that all keys are positive integers, the following two techniques were suggested in order to enhance complexity:

i) In case an entry is removed, instead of marking its location as AVAILABLE, indicate the key as the negative value of the removed key (i.e. if the removed key was 16, indicate the key as -16). Searching for an entry with the removed key would then terminate once a negative value of the key is found (instead of continuing to search if AVAILABLE is used).

ii) Instead of using AVAILABLE, find a key in the table that should have been placed in the location of the removed entry, then place that key (the entire entry of course) in that location (instead of setting the location as AVAILABLE). The motive is to find the key faster since it is now in its hashed location. This would also avoid the dependence on the AVAILABLE object.

Will either of these proposal have an advantage of the achieved complexity? You should analyze both time-complexity and space-complexity. Additionally, will any of these approaches result in misbehaviors (in terms of functionalities)? If so, explain clearly through illustrative examples.

Question 2

Show the steps that a radix sort takes when sorting the following array of 3-tuple Integer keys (notice that each digit in the following values represent a key):

783 992 472 182 264 543 356 295 692 491 947

Question 3

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

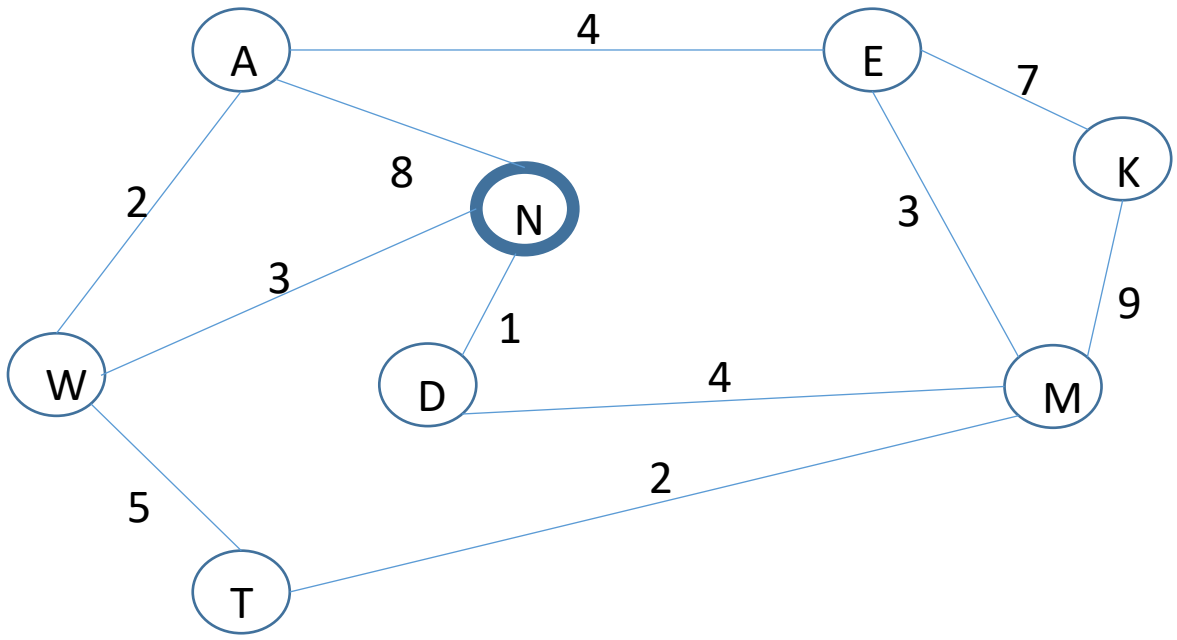
Question 4

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.

Question 5

Using ***Dijksra's Alogrithm***, find the shortest path of the following graph form node N to each of the other nodes.



Question 6

Given the following elements:

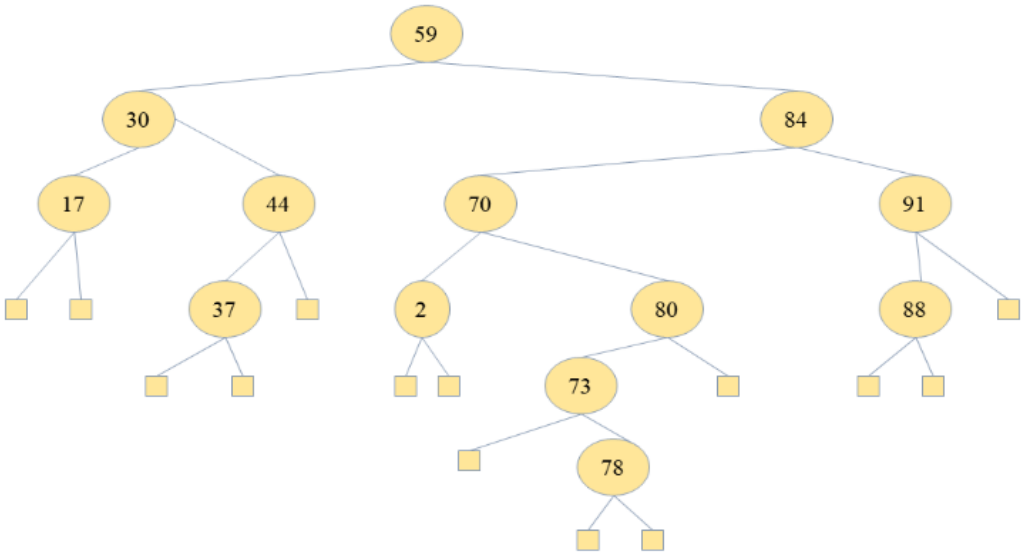
29 38 74 78 24 75 42 33 21 62 18 77 30 16

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.

Question 7

Given the following tree, which is assumed to be an AVL tree:



i) Are there any errors with the tree as shown? If so, indicate what the error(s) are, correct these error(s), show the corrected AVL tree, then proceed to the following questions (Questions ii to iv) and start with the tree that you have just corrected. If no errors are there in the above tree, indicate why the tree is correctly an AVL tree, then proceed to the following questions (Questions ii to iv) and continue working on the tree as shown above.

ii) Show the AVL tree after ***put(74)*** operation is performed. What is the complexity of this operation?

iii) Show the AVL tree after ***remove(70)*** is performed. What is the complexity of this operation?

iv) Show the AVL tree after ***remove(91)*** is performed. Show the progress of your work step-by-step. What is the complexity of this operation?

Part 2: Programming Questions (50 marks):

The Canadian Student Tracking Agency (CSTA) maintains and operates on multiple lists of *n* students. Each student is identified by a **unique**, code called StudentIDentificationCode (SIDC), of 8 digits (e.g. # SIDC: 47203235), some of the lists are local for villages, towns and remote areas, where *n* counts only to few hundred students, and possibly less. Others are at the urban cities or provincial levels, where *n* counts to tens of thousands or more. CSTA needs your help to design an intelligent "student tracking" ADT called **IntelligentSIDC**. Keys of IntelligentSIDC entries are long integers of 8 digits, and one can retrieve the keys/values of an IntelligentSIDC or access a single element by its key. Furthermore, similar to *Sequences*, given a IntelligentSIDC key, one can access its predecessor or successor (if it exists).

IntelligentSIDC adapts to their usage and keep the balance between memory and runtime requirements. For instance, if an IntelligentSIDC contains only a small number of entries (e.g., few hundreds), it might use less memory overhead but slower (sorting) algorithms. On the other hand, if the number of contained entries is large (greater than 1000 or even in the range of tens of thousands of elements), it might have a higher memory requirement but faster (sorting) algorithms. IntelligentSIDC might be almost constant in size or might grow and/or shrink dynamically. Ideally, operations applicable to a single IntelligentSIDC entry should be *O(1)* but never worse than *O(n)*. Operations applicable to a complete IntelligentSIDC should not exceed $O(n^2)$.

You have been asked to **design** and **implement** the IntelligentSIDC ADT, which automatically adapts to the dynamic content that it operates on. In other words, it accepts the size (total number of students, *n*, identified by their 8 digits SIDC number as a key) as a parameter and uses an appropriate (set of) data structure(s), or other data types, from the one(s) studied in class in order to perform the operations below efficiently[1]. You are NOT allowed however to use any of the built-in data types (that is, you must implement whatever you need, for instance, linked lists, expandable arrays, hash tables, etc. yourself).

The **IntelligentSIDC** must implement the following methods:

- **SetSIDCThreshold (Size)**: where $100 \leq \text{Size} \leq \sim 500{,}000$ is an integer number that defines the size of the list. This size is very important as it will determine what data types or data structures will be used (i.e. a Tree, Hash Table, AVL tree, binary tree, sequence, etc.);
- **generate()**: randomly generates new non-existing key of 8 digits;
- **allKeys(IntelligentSIDC)**: return all keys in IntelligentSIDC as a **sorted sequence;**
- **add(IntelligentSIDC,key,value[2])**: add an entry for the given key and value;
- **remove(IntelligentSIDC,key)**: remove the entry for the given key;
- **getValues(IntelligentSIDC,key)**: return the values of the given key;
- **nextKey(IntelligentSIDC,key)**: return the key for the successor of key;
- **prevKey(IntelligentSIDC,key)**: return the key for the predecessor of key;
- **rangeKey(key1, key2)**: returns the number of keys that are within the specified range of the two keys *key1* and *key2*.

1. Write the pseudo code for each of the methods above.
2. Write the java code that implements the above methods.

---

[1] The lower the memory and runtime requirements of the ADT and its operations, the better will be your marks.

[2] Value here could be any info of the student. You can use a single string composed of Family Name, First Name, and DOB.

3. Discuss how both the *time* and *space* complexity change for each of the methods above depending on the underlying structure of your IntelligentSIDC (i.e. whether it is an array, linked list, etc.)?

You have to submit the following deliverables:

a) A detailed report about your design decisions and specification of your IntelligentSIDC ADT including a rationale and comments about assumptions and semantics.

b) Well-formatted and documented Java source code and the corresponding class files with the implemented algorithms.

c) Demonstrate the functionality of your IntelligentSIDC by documenting at least 5 different, but representative, data sets. These examples should demonstrate all cases of your IntelligentSIDC ADT functionality (e.**g., all operations of your ADT for different sizes).** You have to additionally test your implementation with benchmark files that are posted along with the assignment.

Submission Guidelines

- **The written part must be done individually (no groups are permitted). The programming part can be done in groups of two students (maximum).**
- **For the written questions, submit all your answers in PDF (no scans of handwriting; this will result in your answer being discarded) or text formats only. Please be concise and brief (less than ¼ of a page for each question) in your answers. Submit the assignment under Assignment 4 directory in EAS or the correct Dropbox/folder in Moodle (depending on your section).**
- **For the Java programs, you must submit the source files together with the compiled files. The solutions to all the questions should be zipped together into one .zip or .tar.gz file and submitted via Moodle/EAS under Programming 4 directory or under the correct Dropbox/folder. You must upload at most one file (even if working in a team; please read below). In specific, here is what you need to do:**
- 1) Createte **one** zip file, containing the necessary files (.java and .html). Please name your file following this convention:
- If the work is done by 1 student: Your file should be called *a#_studentID*, where # is the number of the assignment *studentID* is your student ID number.
- If the work is done by 2 students: The zip file should be called *a#_studentID1_studentID2*, where # is the number of the assignment *studentID1* and *studentID2* are the student ID numbers of each student.
- 2) If working in a group, only one of the team members can submit the programming part. Do not upload 2 copies.

Very Important: Again, the assignment must be submitted in the right folder of the assignments. Depending on your section, you will either upload to Moodle or EAS (your instructor will indicate which one to use). **Assignments uploaded to an incorrect folder will not be marked and result in a zero mark. No resubmissions will be allowed.**

⇨ Additionally, for the programming part of the assignment, a demo is required (please refer to the courser outline for full details). The marker will inform you about the demo times. **Please notice that failing to demo your assignment will result in zero mark regardless of your submission.** If working in a team, both members of the team must be present during the demo.