COMP352 W22 Assignment 3

Data Structures and Algorithms (Concordia University)

Department Computer Science and Software Engineering
Concordia University

COMP 352: Data Structures and Algorithms
Winter 2022 –Assignment 3

Due date and time: Thursday March 17th, 2022
by 10:00 AM (MORNING)
NO EXTENSION WILL BE ALLOWED BEYOND THIS TIME!

This assignment has two sections: a **theory section** and a **programming section**. These sections are completely independent from each other except for the due date and time.

Theory section (50 marks) – submit as theory-assignment-3: You must submit the answers to all the questions below. However, only one, or more questions, possibly chosen at random, will be corrected and will be evaluated to the full 50 marks.

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 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

i) Draw the min-heap that results from the **bottom-up** heap construction algorithm on the following list of values:

20, 12, 35, 19, 7, 10, 15, 24, 16, 39, 5, 19, 11, 3, 27.

Starting from the bottom layer, use the values from left to right as specified above. Show immediate steps and the final tree representing the min-heap. Afterwards perform the operation `removeMin` 6 times and show the resulting min-heap after each step.

ii) Create again a min-heap using the list of values from the above part (i) of this question but this time you have to insert these values step by step (i.e. one by one) using the order from left to right (i.e. insert 20, then

insert 12, then 35, etc.) as shown in the above question. Show the tree after each step and the final tree representing the min-heap.

Question 3

Assume a hash table utilizes an array of 13 elements and that collisions are handled by separate chaining. Considering the hash function is defined as: *h(k)=k mod 13*.

i) Draw the contents of the table after inserting elements with the following keys:
32, 147, 265, 195, 207, 180, 21, 16, 189, 202, 91, 94, 162, 75, 37, 77, 81, 48.

ii) What is the maximum number of collisions caused by the above insertions?

Question 4

To reduce the maximum number of collisions in the hash table described in Question 3 above, someone proposed the use of a larger array of 15 elements (that is roughly 15% bigger) and of course modifying the hash function to: *h(k)=k mod 15*. The idea is 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.

Question 5

Draw a single binary tree that gave the following traversals:
Inorder: T N C K V A S M W Q B R L
Postorder: T C N V S A W M B Q L R K

Question 6

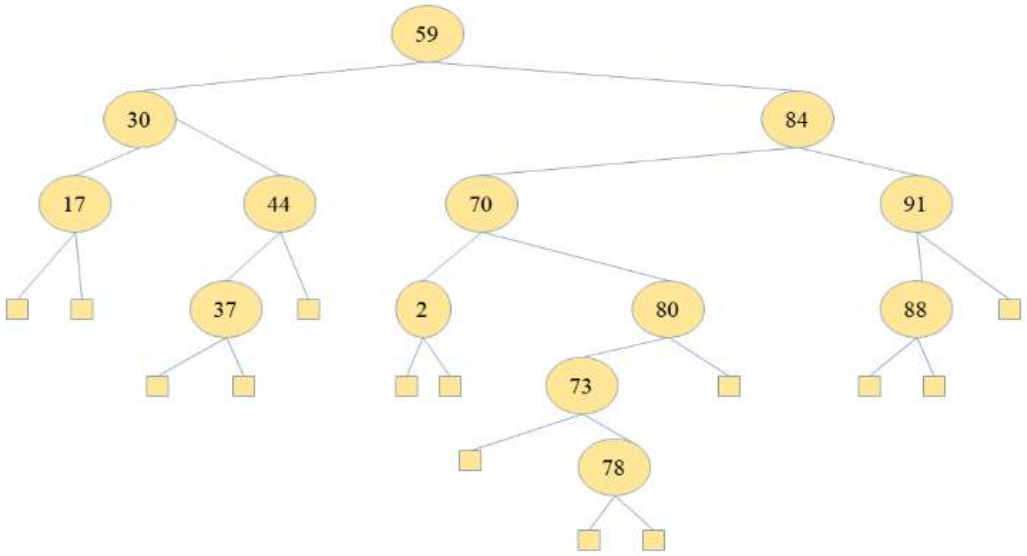
Assume an *open addressing* hash table implementation, where the size of the array is $N = 19$, and that *double hashing* is performed for collision handling. The second hash function is defined as: $d(k) = q - k \bmod q$,
where k is the key being inserted in the table and the prime number q is = 7. Use simple modular operation ($k \bmod N$) for the first hash function.
i) Show the content of the table after performing the following operations, in order:

put(25), put(12), put(42), put(31), put(35), put(39), remove(31), put(48), remove(25), put(18), put(29), put(29), put(35).

ii) What is the size of the longest cluster caused by the above insertions?
iii) What is the number of occurred collisions as a result of the above operations?
iv) What is the current value of the table's *load factor*?

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?

Programming Section (50 marks) submit as programming-assignment-3:

The North American Student Tracking Association (NASTA) maintains and operates multiple lists of students. Some of these student lists are local to villages, towns and remote areas, where the lists contain only a few hundred students each, and possibly less. Others are in large urban centers where lists may contain tens of thousands or more students each.

Each student is identified by a **unique** 8-digit code, called StudentIDentificationCode (SIDC); (e.g. # SIDC: 47203235).

NASTA needs your help to design a clever "student tracking" ADT called **CleverSIDC**. Keys of CleverSIDC entries are long integers of 8 digits, and one can retrieve the keys/values of an CleverSIDC or access a single element by its key. Furthermore, similar to *Sequences*, given a CleverSIDC key, one can access its predecessor or successor (if it exists).

CleverSIDC adapts to their usage and keep the balance between memory and runtime requirements. For instance, if an CleverSIDC 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. CleverSIDC might be almost constant in size or might grow and/or shrink dynamically. Ideally, operations applicable to a single CleverSIDC entry should be *O(1)* but never worse than *O(n)*. Operations applicable to a complete CleverSIDC should not exceed $O(n^2)$.

You have been asked to **design** and **implement** the CleverSIDC 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 **CleverSIDC** must implement the following methods:

- **SetSIDCThreshold (Size)**: where 100 ≤ Size ≤ ~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(CleverSIDC)**: return all keys in CleverSIDC as a **sorted sequence;**
- **add(CleverSIDC,key,value[2])**: add an entry for the given key and value;
- **remove(CleverSIDC,key)**: remove the entry for the given key;
- **getValues(CleverSIDC,key)**: return the values of the given key;
- **nextKey(CleverSIDC,key)**: return the key for the successor of key;
- **prevKey(CleverSIDC,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 at least 4 of the above methods.
2. Write the java code that implements all the above methods.
3. Discuss how both the *time* and *space* complexity change for each of the above methods depending on the underlying structure of your CleverSIDC (i.e. whether it is an array, linked list, etc.)?

[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.

Deliverables

The programming section can be done individually or in groups of two students (maximum!).

You have to submit the following deliverables:

a) A detailed report about your design decisions and specification of your CleverSIDC 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 CleverSIDC by documenting at least 5 different, but representative, data sets. These examples should demonstrate all cases of your CleverSIDC 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.

All these deliverables should be zipped together into one flat (all files are in the root folder) .zip file (NOT tar, NOT gz, NOT rar or any other archiving method) and submitted under Programming Assignment 2 in EAS.

If working in a team, only one member of the team should submit the solution, but it should be clear reading the deliverables if the assignment was done in teams and should clearly identify the team members.

Demo

A demo is needed for this assignment and your lab instructors will communicate the available demo times to you, where you **must** register a time-slot for the demo, and you must prepare your assignment and be ready to demo at the start of your time-slot. If the assignment is done by 2 members, then **both members must** be present for the demo. During your presentation, you are expected to demo the functionality of the application, explain some parts of your implementation, and answer any questions that the lab instructor may ask in relation to the assignment and your work. Different marks may be assigned to the two members of the team if needed. **Demos are mandatory. Failure to demo your assignment will entail a mark of zero for the assignment regardless of your submission. Please read the following carefully!**

- **If you book a demo time, and do not show up, for whatever reason, you will be allowed to reschedule a second demo but a penalty of 50% will be applied.**
- **Failing to demo at the second appointment will result in zero marks and no more chances will be given under any conditions.**