Data structures – wet 2 (Dry part)

We are using the following data structures, which appeared in classes:

1. **HashTable:**
   This is a dynamic array of binary search trees, which store the (key,data) pairs.
   It uses a given hasher object, which supports two functionalities:
   - Calculates a hasher function, f, in O(1) time, where f is a function from the set of all possible keys to the set of natural numbers between 0 and m (where m is θ(number of data objects in hash)) .
   - Modifies f's range in O(1) time in case of resizing of the dynamic array.

Initialized in time complexity O(n)

insert(key, data): Inserts (key,data)

Worst case time complexity: O(log(numOfElements))
Operates at average time complexity: O(1)

remove(key): Removes the (key,data) associated with key

Worst case time complexity: O(log(numOfElements))
Average time complexity: O(1)

search(key): Returns the data related to the given key

Worst case time complexity: O(log(numOfElements))
Average time complexity: O(1)

The HashTable contains an array of size m (m = θ(n)) (n is number of elements).
Each non-empty cell has an AVLTree of elements. Every element appears in only one tree. An empty cell contains NULL pointer. Sum of memory of all the trees is O(n).
So overall memory complexity is O(m+n) = O(n)=O(m) (since m = θ(n)).

2. **<u>UnionFind:</u>**

   Saves n elements numbered between 0 and n-1. And up to n sets with the same numbering scheme. After initialization, every element is in a set of the same number.
   Initialized at time complexity O(n)
   find(i):
   Returns the name of the set element i belongs to.
   Worst case time complexity: O(log(n))
   Amortized time complexity: O(log*(n))
   union(p,q):
   Unites the groups associated with “p” and “q”. Returns the new group.
   P and q stop to exist, p U q exists instead.
   Worst case time complexity: O(log(n))
   Amortized time complexity: O(log*(n))

   UnionFind holds a constant amount of arrays of size n. So the memory complexity is O(n).
   FindNum is a recursive method of the unionFind object. The recursion depth is O(log(n)) with constant memory per iteration.
   Other methods take a constant memory on the heap.
   So the overall memory complexity is O(n).

3. **<u>AVL tree(with getMax functionality):</u>**

   Initialized at time complexity O(1)

Insert(key,data)	worst case time complexity O(log(n))
Remove(key)	worst case time complexity O(log(n))
Find(key)	worst case time complexity O(log(n))
getMax	worst case time complexity O(1)

Assuming number of elements = n, this data structure holds n nodes, and a constant amount of information for the tree information. So this data structure takes O(n) memory.

Data structures summary:

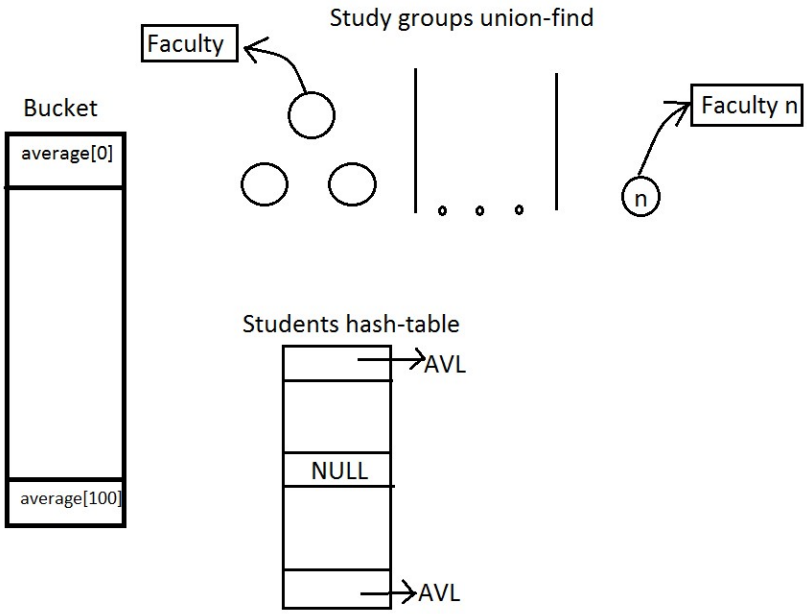
- A hashTable of students.
  Uses a hashing function object, which hashes according to the student's ID.
  Every student points to its study group.
- UnionFind of study groups.
  A faculty is represented as a "reverse tree" of study groups. The root has a pointer to its faculty, which stores information such as smartest student (student with highest average, and if more than one exist than the student with smallest ID).
- "Bucket" of students averages.
  This is an array of size 101. Array[i] holds the number of students that have an average of i.
- AVL tree of students, ordered by ID.

Note:

In the unionFind, a faculty is represented as a "reverse tree" of study groups.
Access to faculty data is done via the root of said tree.
The smallest faculty is united to the largest faculty. Because of that, the new root is the root of largest faculty (not necessarily the root of faculty1).
This is why, when joining 2 faculties, we update the faculty info of the new root to "faculty1". Done at constant time. Faculty2 is then marked as a non-existing faculty. (It doesn't exist anymore, but its study groups belong to faculty 1). Done at constant time.

When finding the faculty of a study group, we will use the usual amortized log*(n) functionality of unionFind to find the root of the reverse tree, but then return the faculty pointed to by the root studyGroup.

From here on out we will refer to the corrected versions of the find and union functionalities of the unionFind as find and union.



Time complexity:

Void* Init(init n)

Initializes hashTable and unionFind at O(n).
Initializes the bucket (sets all elements to "0") at O(n).
Initializes an empty students AVL at O(1).

Overall worst case time complexity for Init: O(n)

StatusType AddStudent(void* DS, int studentID, int average)

Bucket[average]++	O(1)
Insert student to students AVL	worst case: O(log(numOfStudents))
insert student to students hashTable	worst case: O(log(numOfStudents))

Overall worst case time complexity for AddStudent: O(log(numOfStudents))

StatusType AssignStudent (void* DS, int studentID, int studyGroup)

Find student in students tree	worst case: O(log(numOfStudents))

If student is already in the studyGroup, or in another study group, an appropriate error code will be returned(or the function will end if student is already in the study group). This check is performed at constant time, because the student saves a pointer to its study group.

Access studyGroup in unionFind	O(1)
Update study group in student	O(1)
Faculty = unionFind.find(studyGroup)	amortized: O(log*(numOfStudyGroups))

Update smartest student (if inserted student is smartest than the current smartest student).

Overall armortised time complexity for AssignStudent:
O(log(numOfStudents)) + : O(log*(numOfStudyGroups))

StatusType JoinFaculties(void* DS, int studyGroup1, int studyGroup2)

Faculty1 = unionFind.find(studyGroup1)	amortized: O(log*(numOfStudyGroups))
Faculty2 = unionFind.find(studyGroup2)	amortized: O(log*(numOfStudyGroups))
update the "smartestStudent" of faculty1	O(1)
unionFind.Union(Faculty1, Faculty2)	amortized: O(log*(numOfStudyGroups))

Overall amortized time complexity for JoinFaculties: O(log*(numOfStudyGroups))

StatusType GetFaculty(void* DS, int studentID, int* faculty)

Find student in students hashTable	average time complexity: O(1)
Get studyGroup of student	O(1)
faculty = unionFind.Find(studyGroup)	amortized: O(log*(numOfStudyGroups))

Overall average amortized time complexity for GetFaculty: O(log*(numOfStudyGroups))

StatusType UnifyFacultiesByStudents(void* DS, int studentID1, int studentID2)

Faculty1 = GetFaculty(ID1)	average amortized: O(log*(numOfStudyGroups))
Faculty2 = GetFaculty(ID2)	average amortized: O(log*(numOfStudyGroups))
JoinFaculties(Faculty1, Faculty2)	amortized: O(log*(numOfStudyGroups))

Overall average amortized time complexity for UnifyFacultiesByStudents:
O(log*(numOfStudyGroups))

StatusType UpgradeStudyGroup(void* DS, int studyGroup, int factor)

Iterate over all students in students AVL. If a student belongs to the given studyGroup, his average will be updated respectively.
A "studyGroupBestStudent" value is being updated during iteration.
A pointer to the studyGroup is also saved (A field of each student).
This iteration is done at worst case time complexity O(numOfStudents).

Faculty = unionFind.(studyGroup) worst case: O(log(numOfStudyGroups))
Update the "facultyBestStudent" of the faculty if studyGroupBestStudent > facultyBestStudent (while ">" is according to the comparison of studentAverage and studentID).

Overall worst case time complexity for UpgradeStudyGroup:
O(numOfStudents + log(numOfStudyGroups))

StatusType GetSmartestStudent(void* DS, int facultyID, int* student)

Access the faculty information via faculties array.
If it wasn't a faculty (out of range of 1-n) or currently this study group is contained in another faculty (because of a previous "JoinFaculties" command) or no smartest student exists, than it will be reported with the appropriate return value.
Else – the smartestStudent is a parameter of the faculty. It is accessed at constant time.

Overall worst case time complexity for GetSmartestStudent: O(1)

StatusType GetNumOfStudentsInRange(void* DS, int fromAvg, int toAvg, int* num)

Iterate over the "bucket" array, from bucket[fromAvg] until bucket[toAvg], and count the number of students. At worst case, the iteration will be over 101 cells (0 to 100).

Overall worst case time complexity for GetNumOfStudentsInRange: O(1)

void Quit(void DS)**

- Students hashTable:
  Array of size m (numberOfStudents = O(m)). Each non-empty cell has an AVL of students. Every student appears in only one tree. An empty cell contains NULL pointer.
  We iterate over the array and deallocate the trees.
  Sum of time for deallocating all the trees is O(numberOfStudents).
  So deallocating the hashTable this is done at time complexity of O(numberOfStudents + m).
  Because numberOfStudents = O(m), the overall time complexity is O(m).

- UnionFind of study groups:
  Union find holds a constant amount of arrays of size n. Deallocating an array takes O(1).
  Thus, deallocating the union find takes O(1).

- "Grades Bucket" - array of students averages:
  This is an array of 101 integers. The memory is deallocated at a single command, in a constant time.

- Students AVL (ordered by ID):
  Deallocating a tree of size k means deallocating k nodes, and the info of the tree (constant size).
  Overall, this is done at a time complexity of O(numberOfStudents).

So deallocating the whole technion system is performed at time complexity of :
O(numberOfStudents + numberOfStudyGroups).

Memory complexity:

We used OOP in C++ for writing this program. Each function (of the interface with the user) takes a constant amount of memory on the stack for local variables and calls a constant amount of methods of the data structures in the system:

- Students hashTable has a memory complexity O(numberOfStudents).

- UnionFind of study groups has a memory complexity O(numberOfStudyGroups).

- "Grades Bucket" - array of students averages:
  This is an array of 101 integers => O(1) memory complexity.

- Students AVL (ordered by ID) has a memory complexity O(numberOfStudents).

Overall, the memory complexity of the system is O(numberOfStudents + numberOfStudyGroups).