**Subject: Data Structures and Algorithm Design**

**Assignment 1: Problem Statement 26**

**Group: 166**

The design of the application is split across the required functions and the Main program as explained below. The **error handling** and the **time complexity** is also explained at the bottom

**Hash Table (created in function initializeHash)**

* The hash table is implemented as nested lists treating each applicant record as a list containing name, phone number, member referral and the application status.
* It is chosen to be a nested list instead of a flat list so that in case of hash value/index collision, chaining of records is possible at the same index (e.g. for two names resulting in same hash index, the records will be stored as two sub list within the same main list index).
* Nesting also allows the effective size of the hash table to be dynamic.
* The entries are stored at the main list index position which is represented by the hash code generated by the Hash function (e.g. if hash code is 211, entry is stored at index 211). And thus, hash code itself is not stored in hash table to save memory.
* The size of the hash table is kept at a reasonable limit considering the scenario of a Club House. Also, it is chosen to be a prime number (599) to increase the possibility that the modulo with 599 is more likely to distribute records evenly across the indexes.
* For cases where a data field (e.g. member reference) is not available for a particular record, the indexes of remaining data fields within the list is still maintained for easy search (that is name is always at index 0, phone number at index 1 etc.)

**Hash Function (hashID)**

The hash function is designed keeping following design considerations in mind:

* It is quick to generate the hash key
* The hashing distributes the records evenly across the hash table to minimize collisions
* The hash key (which then serves as the index) falls within the size of the hash table by taking modulo with the size of the hash table. This ensures that the long hash key is reduced to a 3-digit number falling between 0 to 599

The hash function takes the ascii code of each character in the name of the Applicant, multiple it by its position in the string and sum up over all the characters of the name. This long string is then converted to a 3-digit number as explained above

**Function insertAppDetails**

The insertion of individual records is done at the main list index which is represented by the hash key generated by the hashID function using the name of the applicant. Example: if the return value of hashID for a particular name is 201, the record is created at that index.

If there is collision of hash key for two or more records, all the records are appended at the same “main” list index resulting in “Chaining” of records in the order in which they appear.

Before generating the hash key, the applicant name is converted to “title” case and the name itself is also inserted in “title” case to ensure that there is standardization of name format for insertion as well as for search and update later.

**Function updateAppDetails**

The update function converts the provided applicant name to Title case and generates the hash key using hashID function. Since the hash key is same as index of the main list, update function looks for match of the Applicant name at this index. It is important to check for name match as there may be more than 1 applicant details at the same index (It is also recognized that if there are more than 1 applicant entries with exact same name, update function will update all such records. Since name is the only search criteria to be used, there is no way to resolve such duplicates. Using phone number as additional search parameter is not appropriate as the attempt may be to update the phone number itself).

On finding a match for the name at the generated hash key/index, the function check which of the fields are being changed and accordingly writes the appropriate entries to the Output file. If name is found but no change is detected or if the name is not found at all in existing records, the function writes appropriate message to output file as well.

**Function memRef**

This function takes the hash table and the member ID as input and iterates through the complete hash table to look for this member ID at appropriate sub index (memRef is stored at index 2 of each applicant record). For all applicant records matching this reference member ID, an entry is written to output file. If no matching records are found, an appropriate message is written to output file.

**Function appStatus**

This function takes the hash table as input and iterates over it, looks for the 3 application status and sums up each status in a separate variable. The summarized figures are then written to the output file.

**Function destroyHash**

The function takes the hash table and clears it.

**Main Program**

The main program has following steps:

1. **Open “txt” files, Initialize Hash Table and Create Applicant Records**

* Open the input, prompt and output file for read or read/write as appropriate
* Initialize hash table by calling function initializeHash()
* Read each “non-blank” line from input file, strip the trailing newline character, split the string using ‘/’ as separator to separate out various columns of applicant record to be created.
* It also strips any blank spaces around ‘/’ for each entry and search later on.
* The individual records are stored in a temporary list and passed as parameters to function insertAppDetails to create entry in the hash table
* Count total entries made to hash table and create an entry in output file for inserted records

1. **Read prompt file, search for “Update”, “memRef” and “appStatus” tags and take appropriate action**
   * Read “non-blank” lines in prompt file and strip trailing new line character
   * On finding “Update” tag, split string using ‘/’ as separator to separate out columns of the Applicant record. Call function updateAppDetails passing entries from prompt file
   * On finding “memberRef” tag, separate out member ID and call function memRef passing member ID as parameter
   * On finding “appStatus” tag, call function appStatus passing the hash table
2. **Close all files and clear hash table**

**Error Handling**

The program handles following potential erroneous situations due to incorrect data entry:

1. The program works even if there is collision of hash key for 2 different names
2. Skipping any blank lines in input and prompt file to avoid incorrect record insertion or updates.
3. Taking only first 4 columns, after splitting by ‘/’, from input file in each line. This takes care of a situation where 2 records might be merged in single line and avoids incorrect entry to hash table.
4. Works even if one of the fields of the applicant record is not available in input file, i.e., even if phone number, member reference or application status information is missing
5. Writes appropriate messages to output file is the applicant being searched for is not found in hash table or if there is no change detected as per data provided in prompt file
6. Writes appropriate message is no applicant records are found for the member reference ID being given against tag “memberRef” in prompt file

**Time Complexity**

For representing time complexity in asymptotic notation, let’s assume size N to represent the number of applicant records to be inserted, updated or searched. The time complexity of the entire program is then seen as O(N) and is explained below assuming generic constants C and K to represent instructions that don’t grow with problem size N:

**Main Program Time Complexity**:

There are 2 For loops in the main program. The first one is to read each record in input file and then insert record in hash table by calling function insertAppDetails. This For loop will grow linearly with number of records to be inserted. The instructions within each iteration can be executed in a constant time that may vary per applicant name (as explained in function insertAppDetails) but will not grow with number of records. So total time complexity of this part of the main program can be considered as **c1\*O(N)**

The second For loop in main program is to loop through prompt file and call various functions. The For loop itself can be considered to be iterating for some constant c3 times as it will not grow with number of applicants. The function calls inside the For loop will have time complexity as explained below. The worst of this will be of the order Clog(N) + K (for function memRef or appStatus). So total time complexity for this part will be **c2(Clog(N) + K)**

**So the overall program can be considered to be of time complexity: c1\*O(N) + c2(Clog(N) + K) + c3 (where c3 represents constant number of instructions outside For loops). Dropping constants and taking only higher order terms for simplification, the overall time complexity is O(N).** This mainly comes from reading the input file while the time complexity for accessing hash table is sub-linear.

Function initializeHash and destroyHash: Time complexity will be constant as these functions just create and destroy hash table of fixed size, so it can be taken as O(C)

Function hashID: The hash function will execute number of instructions proportionate to the number of characters in the name of an applicant. Each call to the function will be executed in a constant time which doesn’t grow with number of applicants. So time complexity can be taken as O(Ci) where Ci is a constant proportional to the total number of characters in the name of the ith applicant.

Function insertAppDetails: This function calls the hashID function and then appends the record in the hash table at the generated hash index. This will be a constant time complexity for each applicant, say, O(Ci) + K for ith applicant where Ci is time taken to generate hash index/value and K represents the constant number of instructions to insert the record in hash table.

Function updateAppDetails: This function also calls the hashID function and then iterates through all the records in the hash table at the generated hash index. The time complexity of this function will grow with number of applicants in hash table but at a sub-linear rate as the function only needs to run the For loop for records at a particular index only. This would be time complexity Clog(N) + O(Ci) + K (where C represent instructions executed in each iteration of For loop at hash index, O(Ci) represent call to hashID function for ith applicant and K represents remaining instructions outside For loop)

Function memRef: This function needs to loop through all the records of the hash table at all indexes to search for the member ID. For all N > len(hash table), the outer For loop will have a constant number of iterations. The inner For loop that iterates over all records at particular index will grow at a sub-linear rate to the size of the problem N. So total time for this function will be Clog(N) + K, where C represents the constant number of instructions per outer For loop and K represents all remaining instructions. Noticeably this constant C can be potentially much larger than constant C in updateAppDetails but still has an upper limit as per size of hash table above which it will not increase.

Function appStatus: Time complexity of this function will be of similar order as function memRef as they have same For loops.