BITS Pilani
Pilani | Dubai | Goa | Hyderabad

TIME AND SPACE COMPLEXITY ANALYSIS

By Team 3:

- **Amit Raj**
- **Avantika Mishra**
- **Ashutosh Karmakar**
- **Hritika Sanjay Dhasal**
- **Sritik Dash**

INSERT

Let us start by performing the time complexity analysis for “Insert” function.

1. **Hashing Operations:** The function performs hashing operations to determine the index where the `Account` object should be inserted in the hash table.
   Time Complexity: O(1) for both the `hash1` and `hash2` operations, as they involve constant-time lookups and calculations.

2. **Load Factor Check:** The function checks the load factor of the hash table to determine if rehashing is required.
   Time Complexity: O(1), as it involves a simple load factor calculation and comparison.

3. **Insertion in Hash Table**: The function inserts the `Account` object into the hash table, handling collisions if necessary.
   Time Complexity: O(1) on average for each insertion operation, assuming a reasonably balanced hash table and a limited number of collisions.

4. **Rehashing (if required):** If the load factor exceeds the default threshold, the function performs rehashing by extending the hash table and redistributing the elements. The time complexity of the insert function depends on the number of collisions, the rehashing process, and the linear probing to find an empty slot or an existing account with the same PAN value.
   Time Complexity:
   In the worst case, when the hash table is highly occupied and no empty slots are available, the linear probing process may require traversing the entire hash table, resulting in a time complexity of O(size).
   The rehashing process occurs when the load factor exceeds the default load factor. The extend_rehash method is called, which performs the following steps:

- Copies the existing elements from the old hash table to a temporary array.
- Doubles the size of the hash table.
- Inserts the elements from the temporary array into the new hash table.
- The rehashing process has a time complexity of O(size), as it involves iterating over the elements in the old hash table and inserting them into the new hash table.

Overall, the time complexity of the `insert` function is O(1) on average for each insertion operation, assuming a reasonably balanced hash table and a limited number of collisions. Therefore, considering both the linear probing and rehashing processes, when rehashing is required, the time complexity of the insert function can be approximated as O(size) in the worst-case where “size” is the number of elements in the hash table.

As for the space complexity, the function doesn't use any additional space that scales with the input size. It mainly relies on the existing data structures and doesn't create any significant additional space. The rehashing process involves temporarily storing the elements in a separate array during the resizing of the hash table. Therefore, it requires additional space proportional to the size of the old hash table.

In terms of space complexity, the insert function has a worst-case space complexity of O(size), where size represents the size of the old hash table during rehashing. Therefore, the space complexity of the insert function is constant, except during the rehashing process, where it can be O(size) due to temporarily storing elements in a separate array.

It's important to note that this analysis assumes that the complexity of other functions and data structures used within the `insert` function, such as the hash table, is constant or has already been analysed separately.

SEARCH

Let us now move on to the “SearchDetails” function to analyse its time and space complexity. To do that, let's break it down step by step:

1. **Loop Iteration**: The function iterates over each “String” object in the `find_pans` list.
   Time Complexity: O(n), where n is the number of String objects in the ` find_pans ` list.

2. **Hashing Operations:** Within each iteration, the function performs hashing operations to locate the account details.
   Time Complexity: O(1) on average for each hashing operation, assuming a constant number of collisions.

Time and Space Complexity Analysis for the `if` Block:

The if block executes when the desired account is found at the computed hash index (h2) in the hash table.

3. The time complexity of accessing an element in the hash table is considered O(1) on average. Since accessing elements in the if block involves direct access to the hash table, the time complexity is O(1).

4. **Print PAN Found:** The function formats and prints the account details.
   Time Complexity: O(1) for each account details printing operation.
   Space Complexity: Constant

Space Complexity for “if” block: The space complexity is constant as no additional space is used within the if block.

Time and Space Complexity Analysis for the `else` Block:

5. The else block executes when the desired account is not found at the computed hash index and requires further probing.
   Time Complexity: In the worst case, the else block performs quadratic probing until it finds the desired account or reaches an empty slot. The worst-case time complexity of the quadratic probing loop is O(size), where size is the size of the hash table.
   Space Complexity: The space complexity is constant as no additional space is used within the else block.

Overall, the worst-case **time complexity** of the `searchDetails` function is O(n * size) because, in the worst case, each account may require probing through the entire hash table of size “size”, where n is the number of “String” objects in the “find_pans” list.

As for **space complexity**, the function doesn't use any additional space that scales with the input size. It mainly relies on the existing data structures and doesn't create any significant additional space. Therefore, the space complexity of the `searchDetails` function can be considered O(1), indicating constant space usage.

It is worth noting that the actual time and space complexity may vary depending on the specific inputs and implementation details. The analysis provided here is based on the structure and logic of the given code.

OVERALL ANALYSIS

Keeping all the methods in mind, let's analyse the overall time and space complexity of the entire program:

1. **Initialization:** The program initializes the hash table and other data structures.

   Time Complexity: O(ftable_size), where ftable_size is the size of the firstHash array. It requires initializing each element in the array, resulting in a linear time complexity.

   Space Complexity:
   The space complexity of the FirstLevelHash class: The FirstLevelHash class consists of an array hashTable of size size, which requires O(size) space.
   The space complexity of the Hash class: The Hash class contains an array firstLevelHash of size ftable_size, which requires O(ftable_size) space.

   The fourthChar array has a constant size and requires O(1) space.

2. **Insertion of Accounts:** The program inserts Account objects into the hash table.

   Time Complexity:
   The insert function involves computing the hash values, checking load factors, performing linear probing, and potentially triggering the rehashing process. In the worst case, when the hash table is highly occupied and linear probing is required to find an empty slot, the time complexity can be approximated as O(size).

   The rehashing process, which occurs when the load factor exceeds the default load factor, also has a time complexity of O(size), as it involves copying elements from the old hash table to a new one.

   Therefore, the worst-case time complexity of the insert function is O(size).

   Space Complexity: O(size), as it stores elements in the hash table.

3. **Searching for Account Details:** The program searches for Account details based on given criteria.

   Time Complexity:

   The searchDetails function searches for account details in the hash table and prints them. It iterates over the find_pans list and performs hash calculations and linear probing to find the corresponding accounts. In the worst case, when the accounts are scattered across different slots in the hash table, the time complexity can be approximated as O(n * size), where n is the number of accounts and size is the size of the hash table. The printPanFound method, called within the searchDetails function, has a constant time complexity of O(1).

   Therefore, the worst-case time complexity of the searchDetails function is O(n * size).

   Space Complexity: O(1), as it doesn't use any additional space that scales with the input size.

4. **Rehashing (if required)**: If the load factor exceeds the default threshold, the program performs rehashing by extending the hash table and redistributing the elements.

   Time Complexity: The extend_rehash function involves resizing the hash table and rehashing the elements. It copies the existing elements from the old hash table to a temporary array, which has a time complexity of O(size). The new hash table size is doubled, so the rehashing process has a time complexity of O(size). Therefore, the time complexity of the extend_rehash function is O(size).

   Space Complexity: It temporarily stores the elements in a separate array during the resizing process, requiring O(size) space for the old hash table.

5. **hashTablePrint function**:

   Time Complexity: The hashtablePrint function iterates over the hash table and prints the account details. It iterates over each slot in each hash table, resulting in a time complexity of O(ftable_size * size). Therefore, the time complexity of the hashtablePrint function is O(ftable_size * size).

   Space Complexity: The space complexity of the function is primarily determined by the Formatter object fmt and the printed output. The Formatter object requires a small amount of memory, which is negligible compared to the overall data size. The printed output requires additional space proportional to the number of elements in the hash table. Overall, the space complexity can be approximated as O(ftable_size * size) due to the printed output.

Considering these functions are the main operations in the code, the overall time complexity can be approximated as the maximum among them, which is O(n * size) where n is the number of accounts in this case.

Overall, the space complexity of the entire code can be approximated as O(ftable_size * size) + O(size) + O(ftable_size) + O(1) which simplifies to O(ftable_size * size) for the hash tables and other data structures, along with the constant space required for variables and calculations.

It's important to note that these complexity analyses assume that the hash table and other data structures used in the program have constant-time operations and do not introduce additional complexity. Additionally, the complexity can vary depending on factors such as the input size, distribution of data, and specific implementation details.