Kristian Snyder  
Data Structures CS2021 Section 001  
Prof. Charles Zimmer, Prof. Karthik Vadambacheri  
Lab 09

Quick lookup of data is always important when writing all kinds of applications, as it is usually one of the largest bottlenecks in execution. Hash tables, when implemented well, will usually outperform many other kinds of data structures that also can be used as a key-value store, making them useful in almost all areas of computing. However, there are several considerations to be taken in implementation of a hash table. One of the most important is deciding how to deal with multiple objects that map to the same index but contain different data, as the user of the structure usually cannot predict when this will occur easily and so must be accounted for. Here, I implemented both linear probing and chaining to manage this issue, with advantages and disadvantages inherent in each. Neither is a perfect solution, but each can outperform each other in different cases, as seen in the results of Task 5.

This issue of handling duplicate indices is one of the most applicable to decisions made in industry, where the effects of different choices made in the name of either ease of use or performance can result in, sometimes, significant differences in speed that can also be contrary to the expected results. As complexity increases in development, the number of variables that go in to determining performance increases quickly. Therefore, minimizing variability based on the data stored in the structure is extremely important, as the data inserted can rarely be predicted or controlled and so can cause a previously efficient but fragile algorithm to collapse under the randomness. This applies in the case of a hash table by creating a hash algorithm that is uniformly distributed under all inputs and choosing a duplication handler that fits for the size of the table and the expected amount that it will be filled.

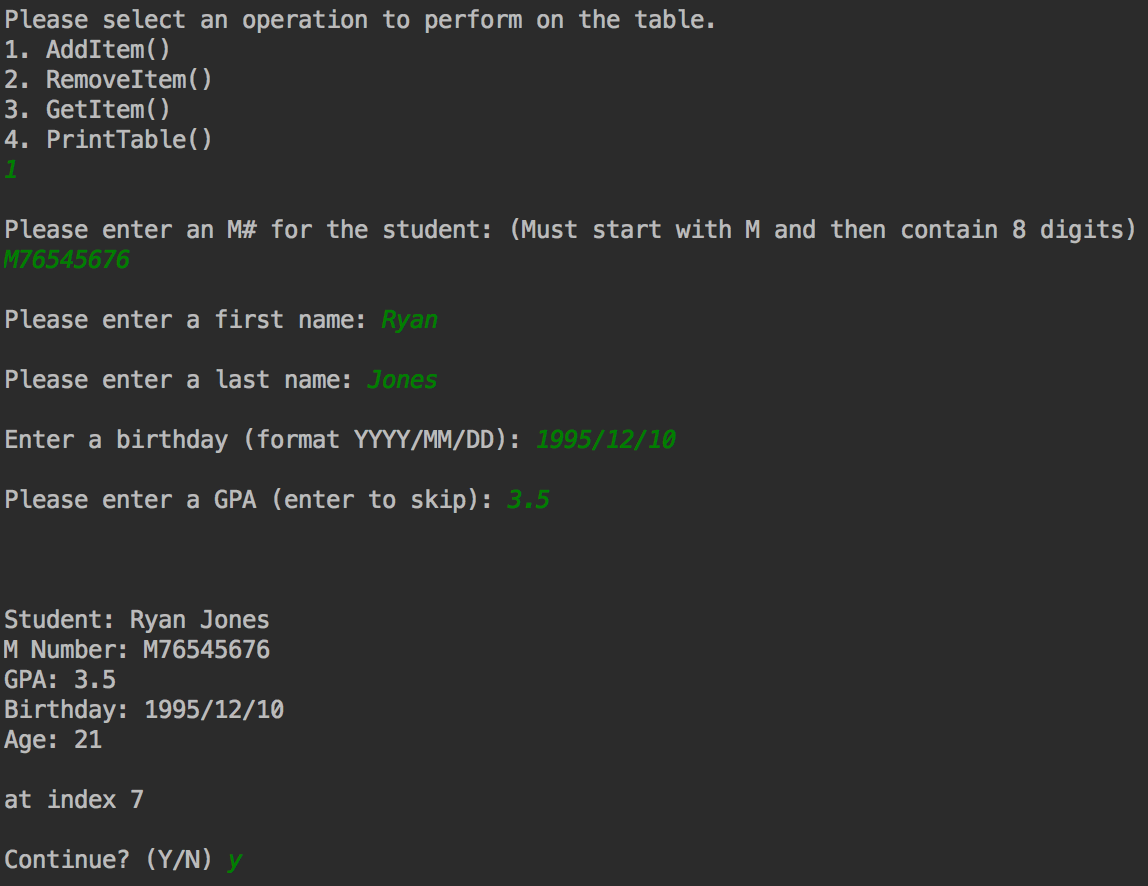
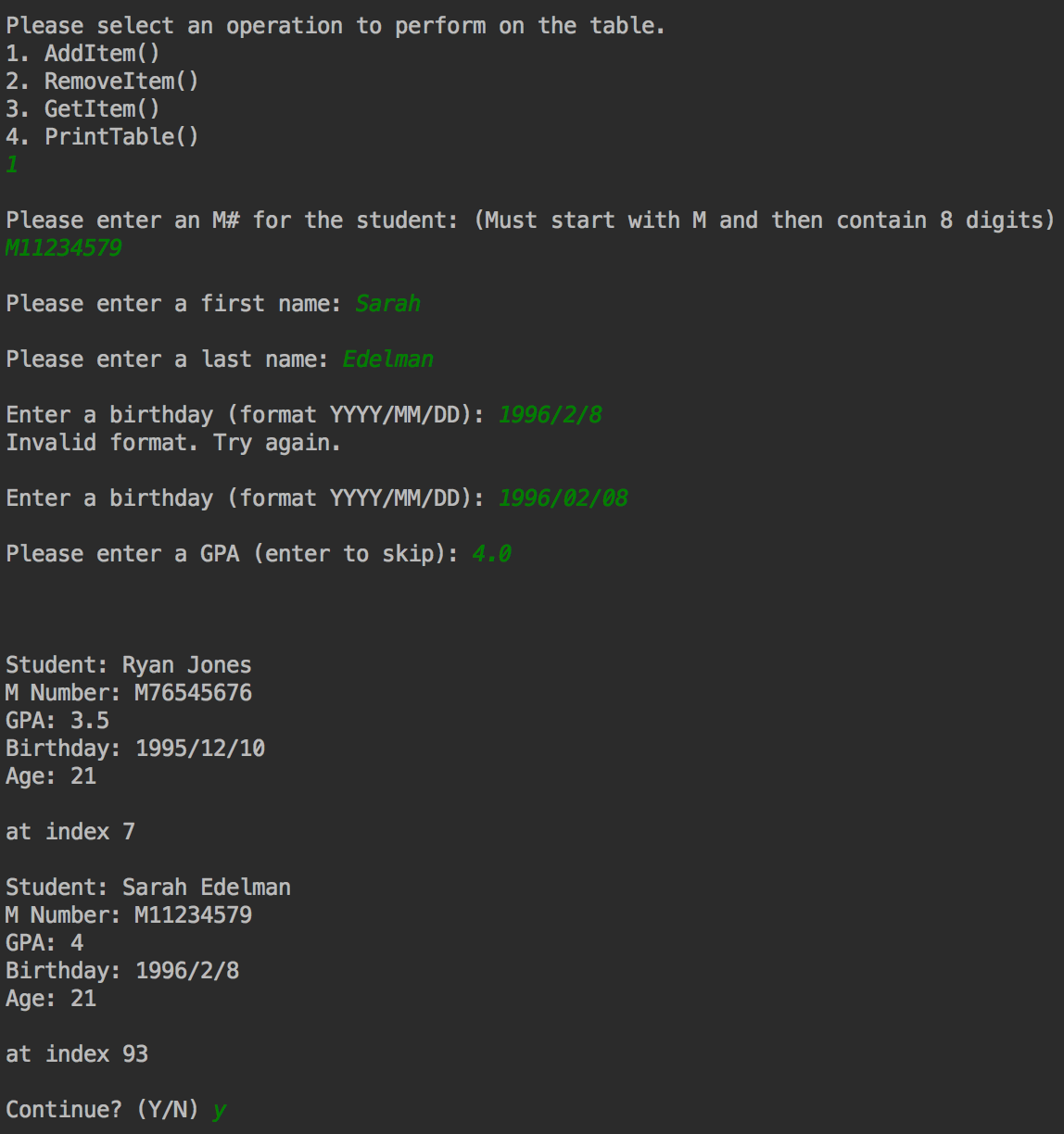
## Task 2

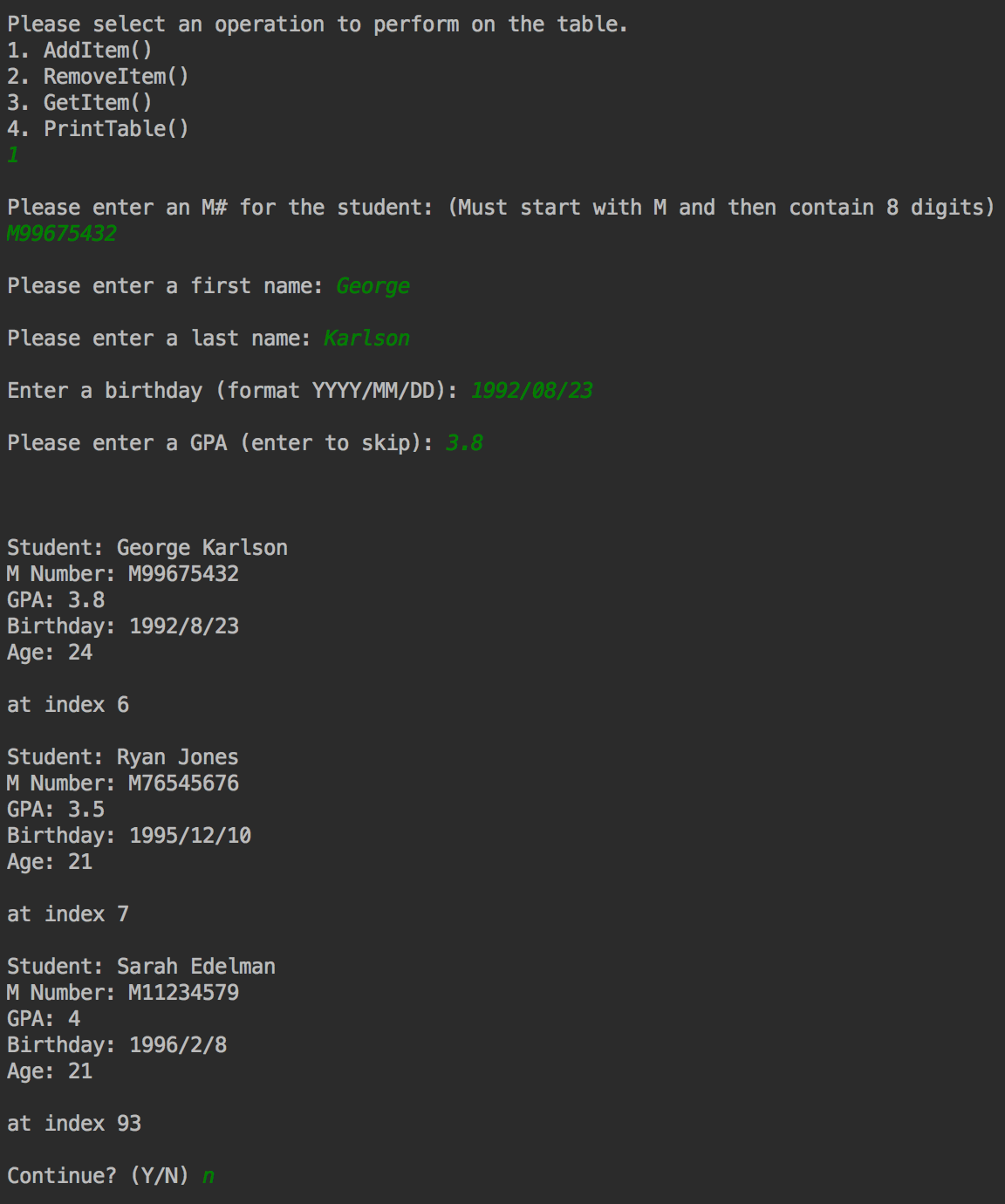
I only had to add the string conversion overload to use the Student class. This was relatively short, since modern C++ has this functionality built in and only required me to return the M# of the student.

## Task 3

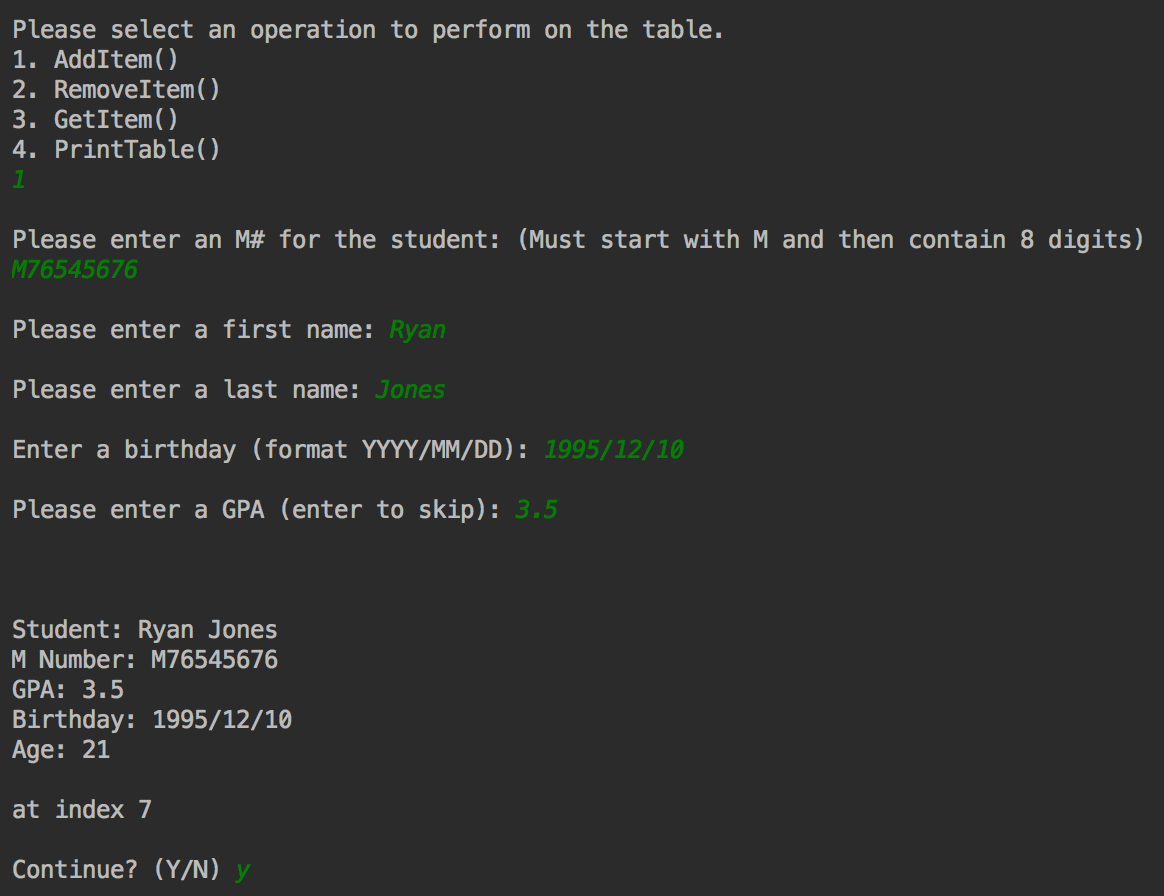
To convert the code for entering students using the user's input and manipulate the hash table using that data, I only had to modify the selections possible and the methods called. All functions for querying the user for student data functioned without modification.

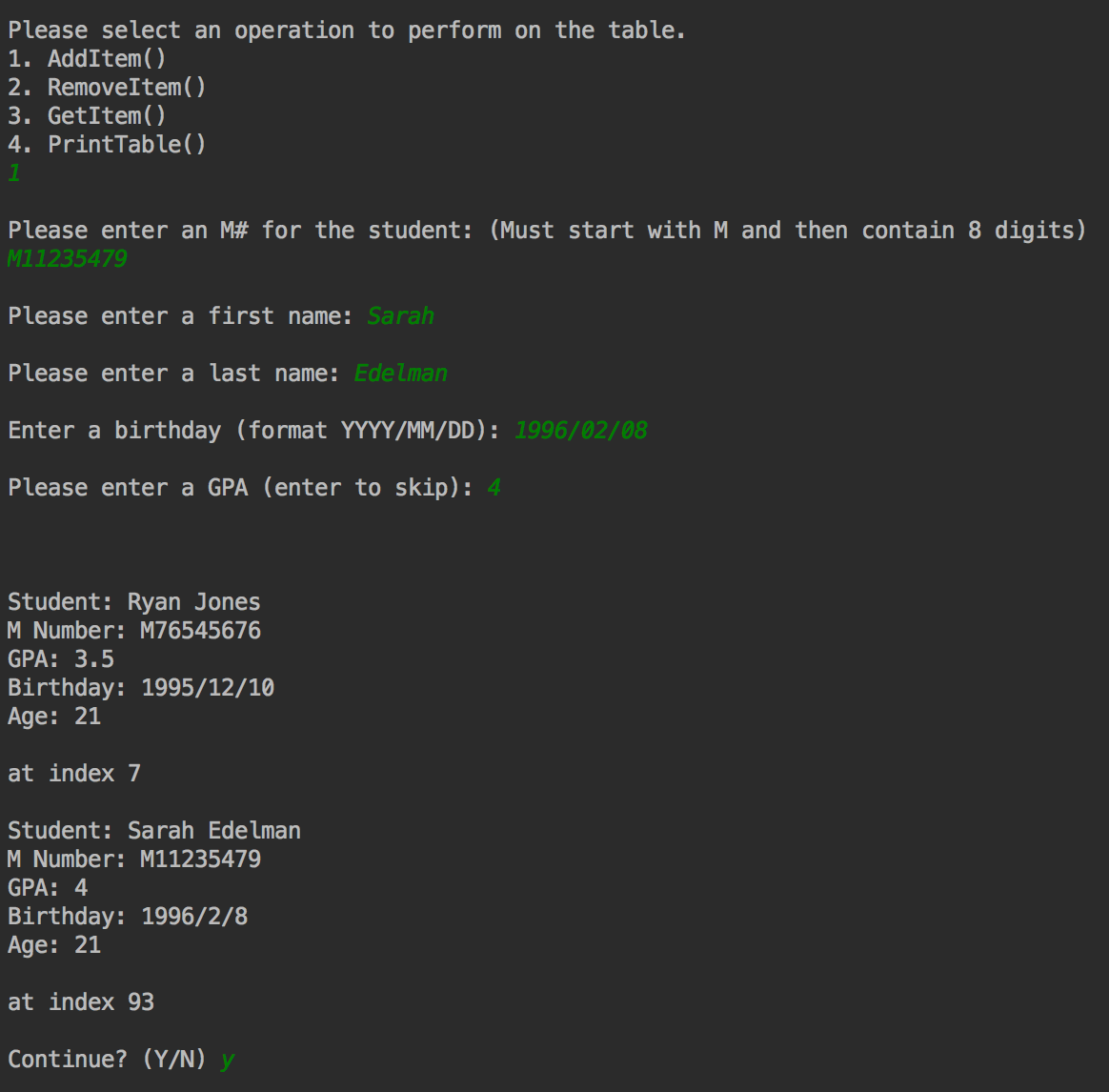
Adding multiple students

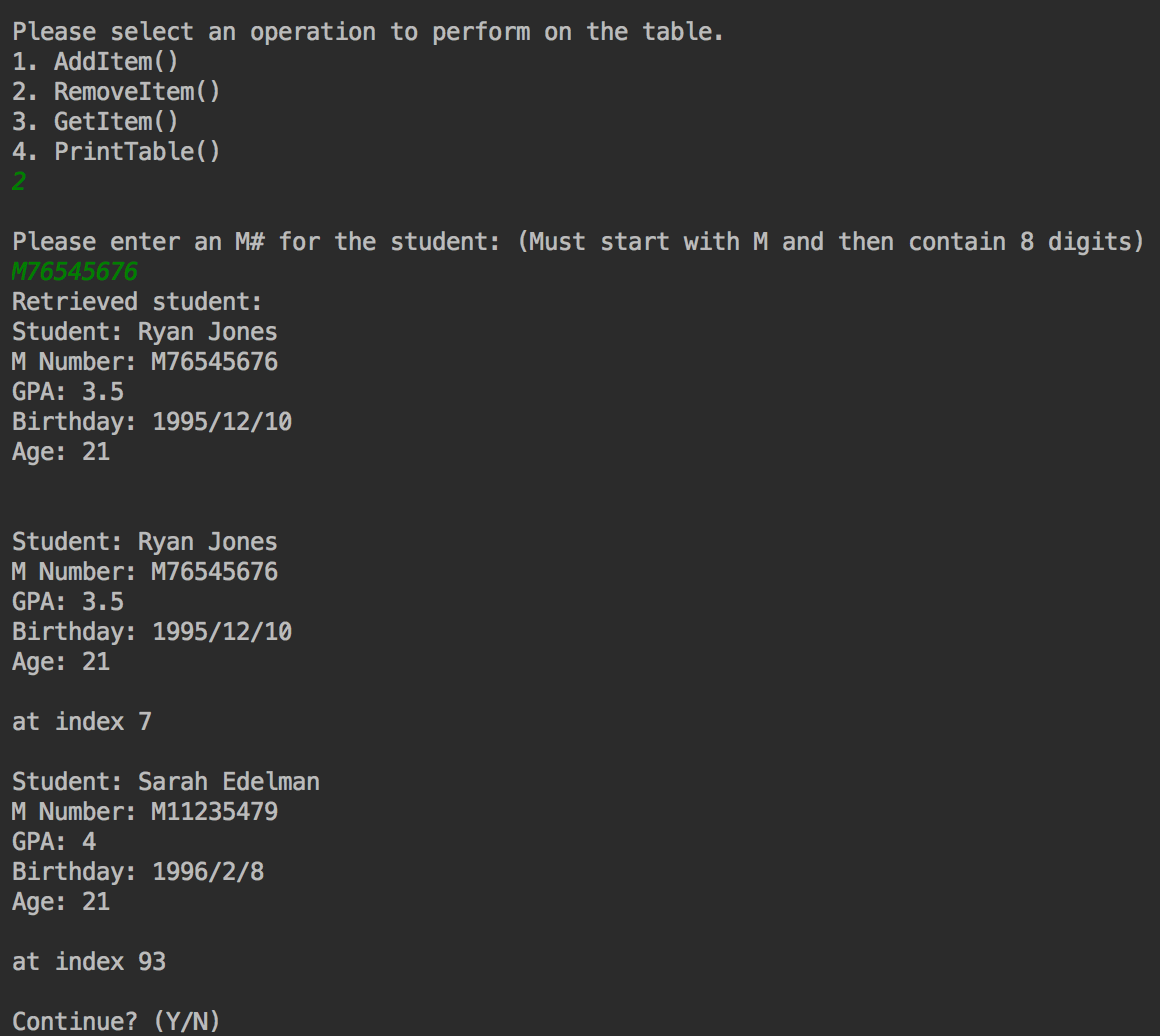
 



*Removing a student*

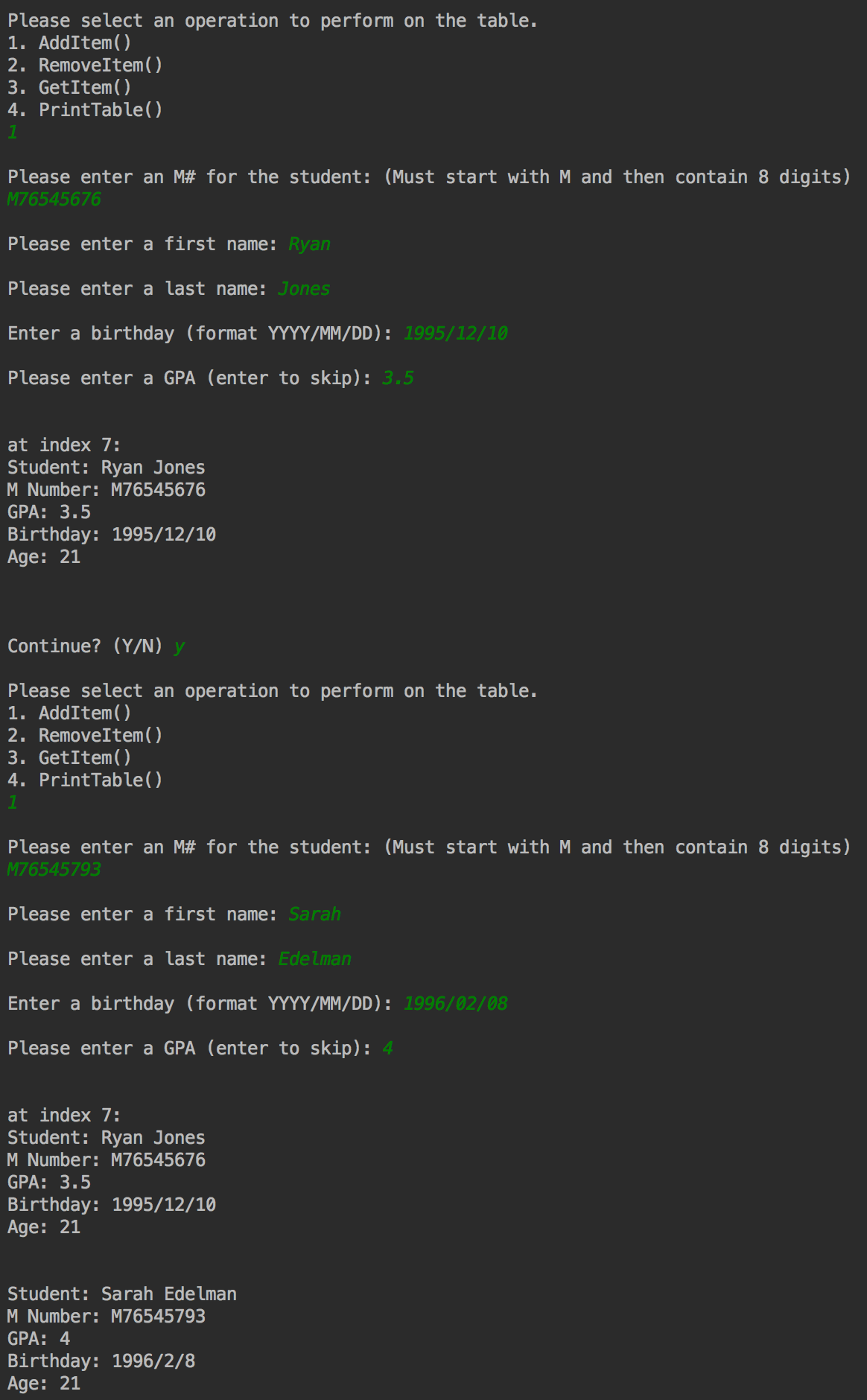






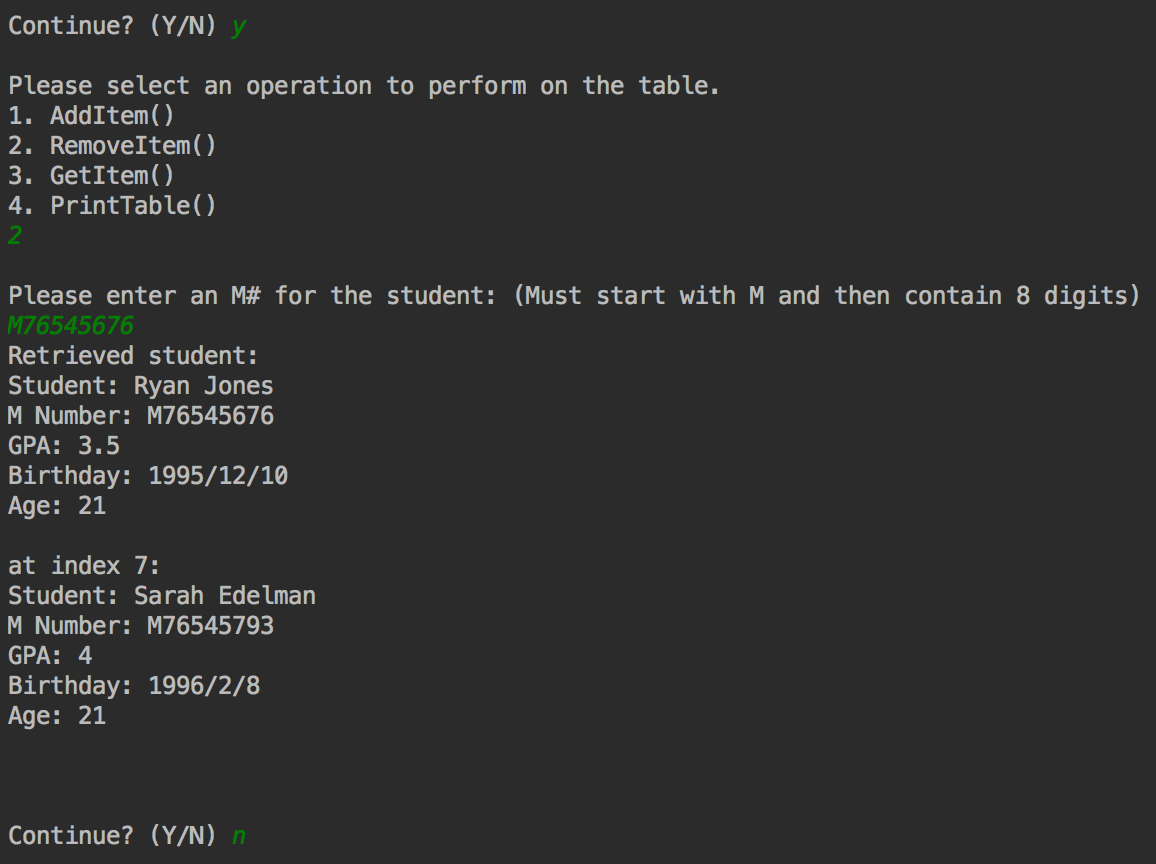
## Task 4

*Adding different students to the same index*

To implement the chained hash table, I decided to replicate instead of import the linked list implementation for additional practice and to directly handle the nodes used with a private class member defining the Node, similarly to how the linear probing table was implemented. Since the hash table returns data, not nodes, I slightly modified portions of the implementation to directly access the data instead of returning the whole node and making it safe for the class user. This also afforded me the opportunity to try and make my past work more efficient.

To test this implementation, I only had to change the hash table instance used.

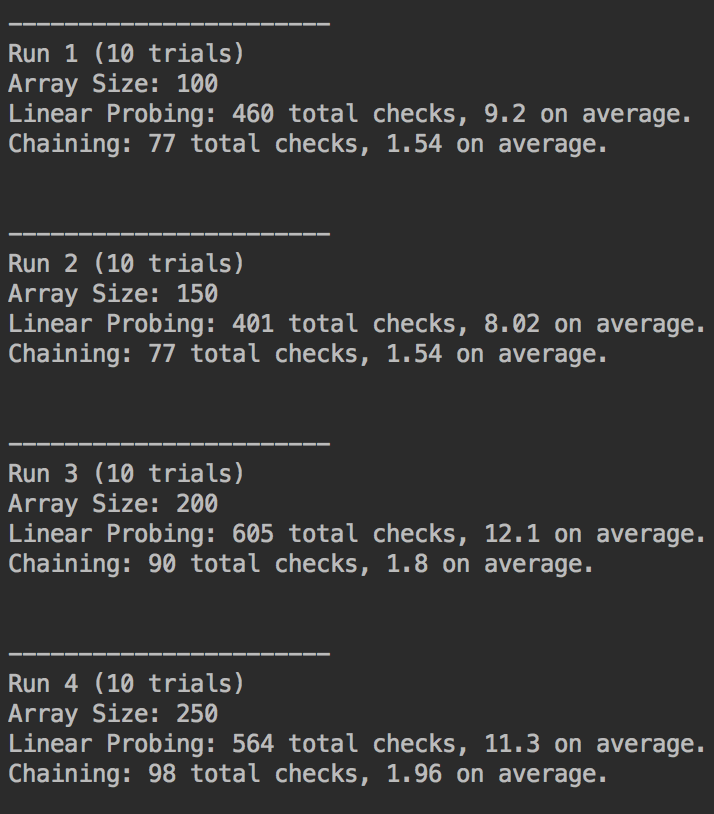
Removing the first student inserted at an index (continues directly from previous test)



## Task 5

Performance results

|  |  |  |
| --- | --- | --- |
| Array size | Linear Probing Accesses | Chaining Accesses |
| 100 | 460 | 77 |
| 150 | 401 | 77 |
| 200 | 605 | 90 |
| 250 | 564 | 98 |

These results were not very surprising to see, considering the hash algorithm used. Almost all of the results for array size 100 were clustered around 1-10 and 90-99, which rapidly decreases performance as almost none of the items are in their expected indices; items added later are shifted even farther from where they should be. While it appears that two M numbers do not frequently map to the same index—the number of accesses for chaining on average is the same as the average number of students per *occupied* index, so fewer than 2 in all array sizes—only a few for each *clustered around a tight range of values* is what results in around **5-6x** the number of checks when probing compared to a chain implementation.

It would be expected to see the number of accesses go down for linear probing as array size increases, since there are, theoretically, more empty spaces to fill with collisions. However, clustering resulted in even more accesses here. (Other runs of this test have resulted in lower accesses as array size increased, but decreases between sizes were rarely more than 10-50 and frequently had *increases instead.*) Chaining stayed relatively constant, since the large amount of clustering resulted in about the same number of items per occupied spot.

With a more uniformly distributed hash algorithm, the indices for the array of students would be far more spread out and the average accesses for linear probing would quickly approach 1-2. Chaining may also see a decrease, but not nearly as pronounced. This performance improvement for linear probing would increase with array size, as more space between elements means that all elements mapping to the same index will most likely be contiguous compared to a mostly filled array, where there may be elements separated from their group by another set that maps to an entirely different index, causing wasted comparisons and bloating the execution time.