OzoBlockly Editor—**Hash Table Program**—Student Guide

by Richard Born, Associate Professor Emeritus, Northern Illinois University

Introduction

A hash table is a data structure that stores data in *(key, value)* pairs. It allows for very fast insertion, searching, and deletion operations, even for very large tables. The key is converted into an index into the table by a process known as hashing. Then the value associated with that key is stored along with the key at the hashed address.

For example, suppose that we have *(student ID, value)* data pairs, where the *student ID* is a unique number assigned to each student and *value* is information about the student. This information may include the student's first and last name, home address, and telephone number, among other information about the student. To keep things simple, we'll assume that the *student ID* is a 3-digit number between 100 and 999, and the *value* is the student's first name. Figure 1 shows the *(student ID, Name)* data pairs that we will use in our study of the hash table concept.

Student ID	253	421	408	678	304	365	804	211	360	906	923	316
Name	Bev	Rich	Sue	Denny	Mark	Carol	Diane	Ed	Steph	Greg	Gloria	Jim

Figure 1

What is Hashing?

Hashing is the process accomplished by a function that takes the key, in this case the student ID, and converts it into an integer index specifying the location in the hash table where the *(key, value)* data is to be stored.

There are a wide variety of hashing functions. We'll use a very common one that finds the remainder when the key is divided by the size of the hash table. Suppose that our table size is 17 and takes the form shown in Figure 2. With a table size of 17, the indexes will run from 0 through 16, as shown to the left of the table. Two separate one-dimensional arrays will be used to store the *Student ID* and *Name*.

Let's take Bev as an example. When her student number 253 is divided by the table size 17, the remainder is 15, so starting with an empty table, her student ID and name would be stored at index 15 in the hash table.

Next, let's suppose we want to add Rich to the table. His student number 421, when divided by the table size 17 gives a remainder of 13. So, Rich's information is stored at index 13.

Next, let's suppose we want to add Sue to the table. Her student number 408, when divided by the table size 17 gives a remainder of 0. So, Sue's information is stored at index 0.

So far so good!

Hash Table

	Student ID	***Name***
0	408	Sue
1		
2		
3		
.		
.		
.		
13	421	Rich
14		
15	253	Bev
16		

Figure 2

Collisions

Next, let's suppose we want to add Denny to the table. His student number 678, when divided by the table size 17 gives a remainder of 15. Oh, oh! Index 15 is already in use by Bev. We have what is known as a ***collision***. There are a variety of ways to handle collisions in hash tables. We will make use of a simple one known as *linear probing*. This technique for handling a collision searches the table in a stepwise fashion beginning with the location following that of the collision. When an empty location is found, the data is stored at that empty location. With this in mind, we see that index 16 is empty, so Denny's data is stored at index 16 in the hash table, as shown in Figure 3.

Next, let's suppose we want to add Steph to the table. Her student number 360, when divided by the table size 17 gives a remainder of 3. So, Steph's information is stored at index 3 in the hash table. No problem.

Next, let's suppose we want to add Diane to the table. Her student number 804, when divided by the table size 17 gives a remainder of 5. So, Diane's information is stored at index 5 in the hash table. Again, no problem.

Hash Table

	Student ID	Name
0	408	Sue
1	304	Mark
2		
3	360	Steph
4		
5	804	Diane
.		
.		
.		
13	421	Rich
14		
15	253	Bev
16	678	Denny

Figure 3

Making the Hash Table Circular

Next, let's suppose we want to add Mark to the table. His student number 304, when divided by the table size 17 gives a remainder of 15. So, we would like to store Mark's data at index 15 if it was available, but it is not. The next index 16 is also taken and we are at the end of the hash table. So, what do we do? One way to solve this problem is to continue the search back at the beginning of the hash table at index 0, searching until you find an empty location. If we do this, then we would store Mark's information at index 1, as shown in Figure 3. In effect, we have made the table *circular*, continuing the search at the top of the table when the end of the table is reached.

Some Comments on Efficiency of Our Hashing Process

A great deal depends on the *randomness* of the hashing function algorithm and on the *density* of the items in the table. If the hashing process separates the input items well, and if the table is far from full, then the search will identify most items after one or a small number of comparisons.

The "fullness" of a hash table is characterized by what is known as the *load factor*. This is the ratio of the number of elements stored in the table to the size of the table. A low load factor, meaning many empty locations, generally leads to better performance when using a linear probe.

A problem with using a linear probe with a hash table is that clumps of data tend to develop. You saw this with the fact that Bev, Denny, and Mark all initially hashed to index 15. This resulted in the clump 15, 16, 0, 1. An alternative method for handling collisions makes use of a technique called *separate chaining*. Each location in the hash table contains a linked list, where items hashing to that key are stored. Our hash table, however, will make use of a linear probe as we have discussed this technique in great detail in the preceding sections of this lesson.

Overview of Our Hash Program

Ozobot Ari or Evi will be the main players in our hashing program. With Evo, all user interactions, including data input and message output, will take place in the *Terminal* located in the bottom right corner of the Ozobot Editor screen. Evo will only display a top light colored green, blue, or red. Green means that the *Add Student* function is being executed. Blue means that the *Search Table* function is being executed, and red means that the *Delete Student* function is being executed. When using Ari, things get much more interesting. This is because all user interactions in the form of data input and message output are accomplished with Ari. You simply hold Ari in your hand, enter student IDs and names on Ari, and Ari responds with appropriate messages related to the current task—no need to look at the Ozobot Editor screen!

The program is designed to keep the user completely aware of what is happening for each of the three functions *Add Student*, *Search Table*, and *Delete Student*. For example, if the user asks to add a student when the table is full, the user will be informed that the table is full. If the user asks to search for a key that isn't in the hash table, the user will be informed of this fact. If the user asks to delete data that is in the table, the user will be given the option to cancel the deletion before it takes place.

In our discussion of the hashing program, a code review in the form of comments will be used to help you understand the fine details of the main program and each of its subroutines. The program contains more than 360 blocks, but its modular design makes it much easier to follow.

This program makes use of blocks from most of the block categories in level 5 of the Ozobot Editor. Do not hesitate to consult the *Reference Guide*, which can be opened on the right side of the Ozobot Editor screen, if you need help with any of the blocks. These blocks include the following categories:

- Evo/Ari
- User I/O
- Text
- Logic
- Loops
- Math
- Lists
- Variables
- Functions

After you have studied the code review and feel confident that you understand the details, the real fun begins. You will add students, search the table, and delete students from the table. There is a helpful aid each time that you add a student to the table. The hash table *index*, *student ID*, and *Name* will be displayed in the *INPUT/OUTPUT* region at the bottom of the Ozobot Editor screen.

The Main Program

The main program creates the hash table consisting of two arrays which are used to store (key, value) entries of studentIDs and student names. It is also designed to handle the three major tasks involved in working with this hash table—adding students, searching the table, and deleting students. Figure 4 shows the blocks making up the main program along with comments that detail the blocks.

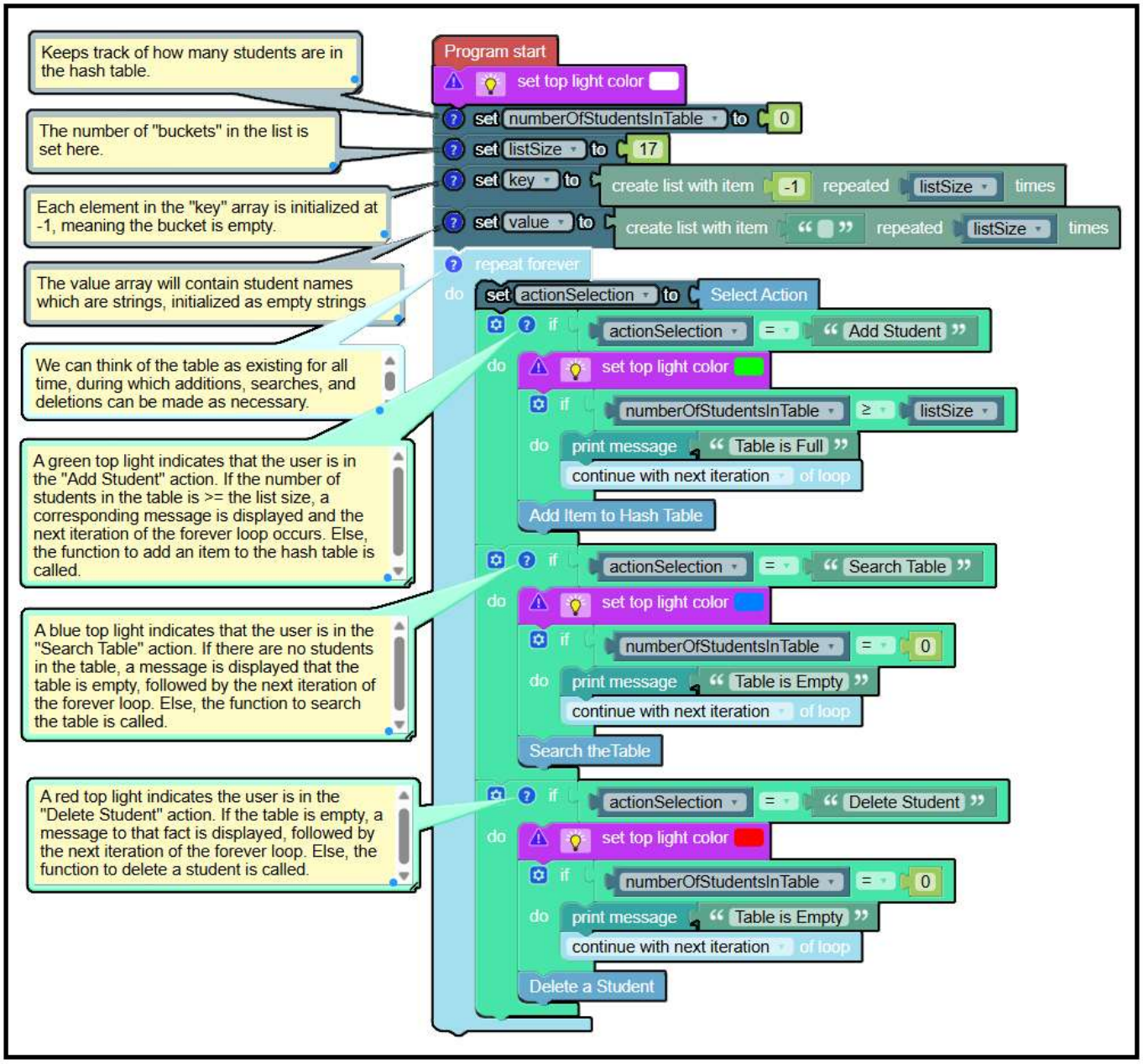

Figure 4

Supporting Functions

There are several functions that support what is accomplished by the main program and the *Add Student*, *Search Table*, and *Delete Student* subroutines. These supporting functions are shown in Figure 5 along with comments that provide detailed information.

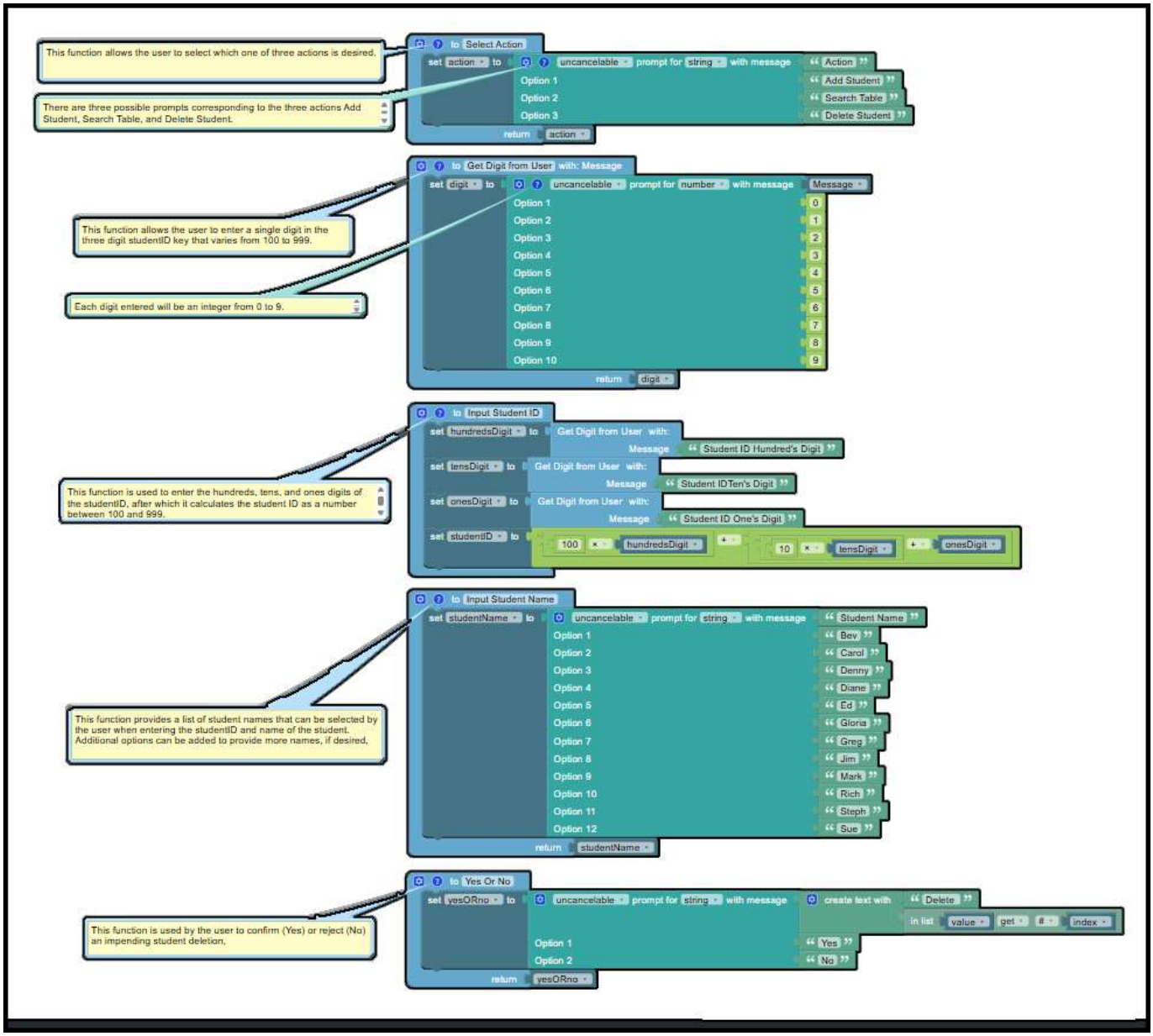

Figure 5

The "Add Student to Hash Table" Function

The function that allows adding items to the hash table is the first of the three major functions associated with hash tables. The entire function is shown in Figure 6, with comments that detail the program logic.

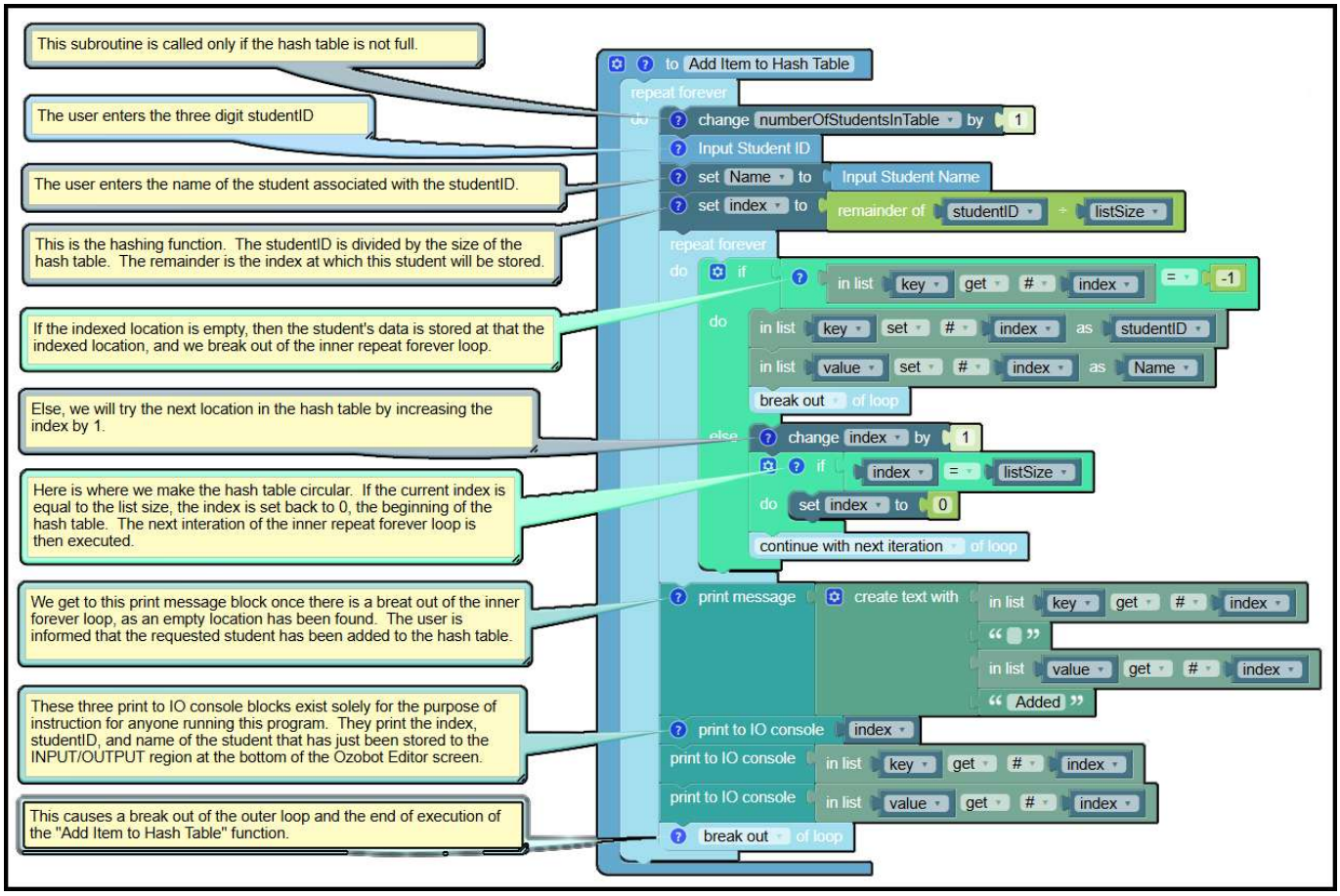

Figure 6

This space is intentionally left blank.

The "Search the Table" Function

The second major function used with hash tables provides the ability to perform a key-based search of the table. The user would enter the key in order to get value information regarding that key. Such a search is generally very fast for a table with a load factor not exceeding about 70%. If the key is not in the table, then the entire table must be searched to determine this fact. The entire function is shown in Figure 7, with comments that detail the program logic.

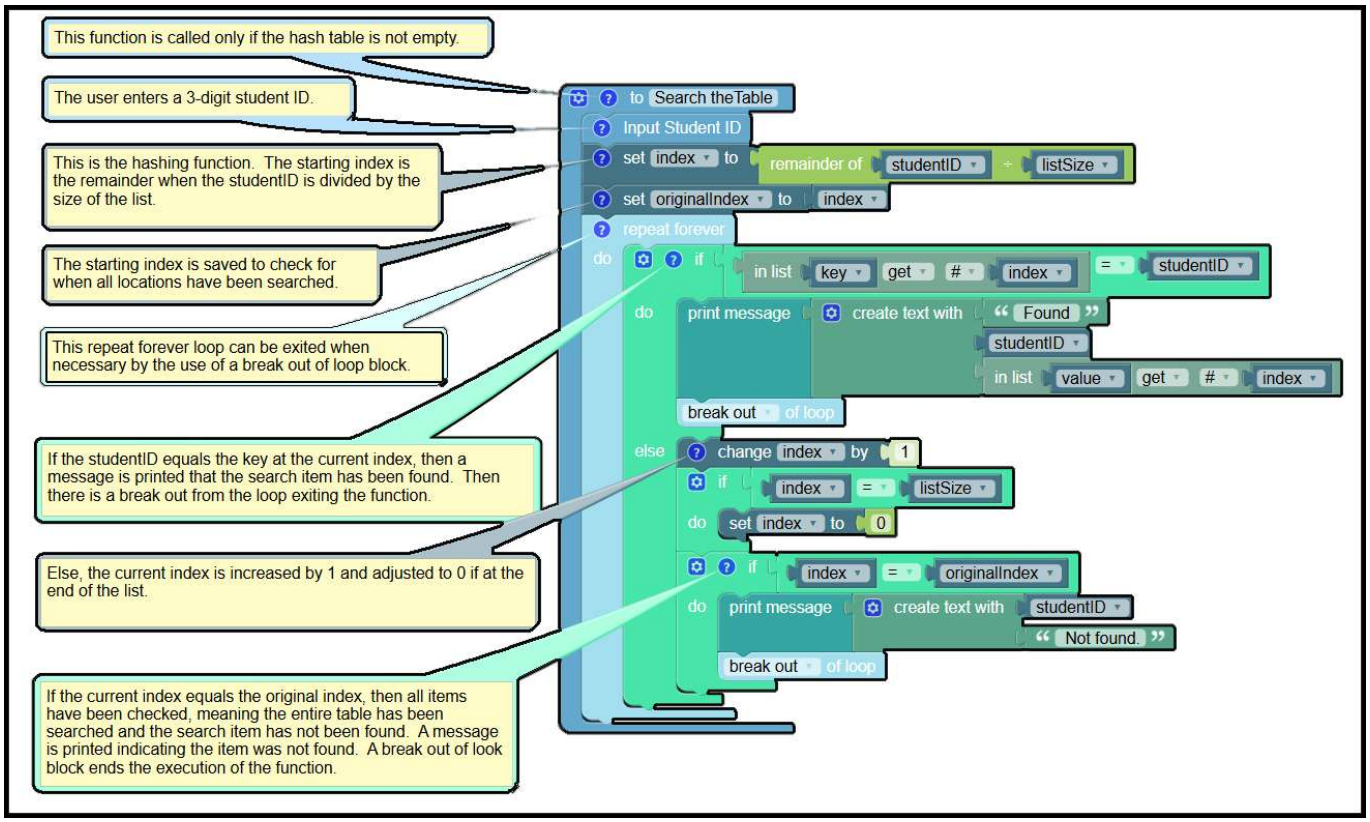

Figure 7

This space is intentionally left blank.

The ***"Delete a Student" Function***

The third and final major function used with hash tables provides the ability to delete an item from the table. The overriding logic is similar to that of the search function, but with special details that must be considered in the deletion process. The entire function is shown in Figure 8, with comments that detail the program logic.

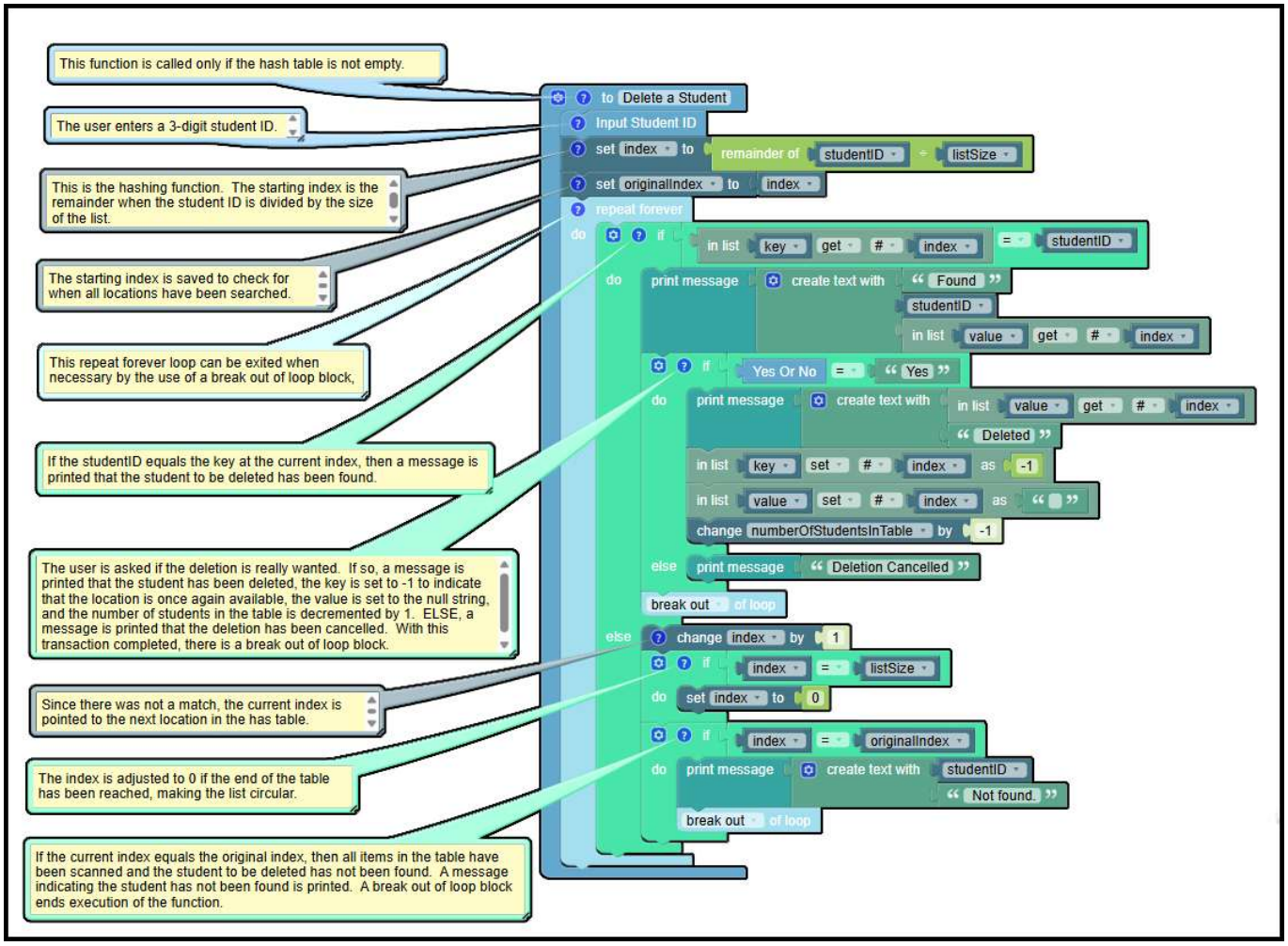

Figure 8

This space is intentionally left blank.

Working with the Hash Table Program

Part 1

1. Open the hash program by entering the share code **um7yi7f**. You will note that the *listSize* variable is set to 17. Leave it at this value and then **RUN** the program.
2. Set up the Ozobot Editor screen so that the left half shows the program blocks, and the right half shows the *INPUT/OUTPUT* region. Do this by clicking on the split screen icon clear that appears at the far right of the *INPUT/OUTPUT* region.
3. Enter the following students in the order given as discussed on pages 1 through 3 of this student guide: *Bev, Rich, Sue, Denny, Steph, Diane, and Mark*. While entering the student data, you will see the *index location* where the data is stored, the *student ID*, and the student *name* displayed in the INPUT/OUTPUT region for each student. When all seven students have been entered, the INPUT/OUTPUT region should appear as shown in Figure 9, in agreement with the discussion on pages 1 through 3 of the student guide. **This region shows data only when it is entered via the function that adds students to the hash table. This region will not show any deletions.** The intent is to let you know the *index into the hash table* when student information is added to the table. The index does not appear on either Ari (or the Terminal if you are using Evo), as the index is intended to be transparent to the user.

Figure 9

4. The table of Figure 1 indicates that the studentID for Greg is 906. **Before** adding Greg to the hash table, manually compute the index location from the hashing technique, *i.e.,* finding the remainder of the studentID 906 when divided by the list size 17. What index do you obtain? Again, before adding Greg to the hash table, what index will Greg's data be stored at and why? Now add Greg to the hash table. At what index was his data stored?
5. What index will Ed' s studentID hash to? Add Ed to the table—were you correct?

6. Now consider adding Gloria to the table. What index will Gloria' s studentID hash to? Where does Gloria's data actually get stored? Explain why.
7. Now consider adding Jim to the table. What index will Jim' s studentID hash to? Does Jim's data get stored at that index in the hash table?
8. Our final student is Carol. What index will Carol's studentID hash to? Does Carol's data get stored at that index in the hash table? Explain why.
9. Now do a couple of searches:
    a. Search for Mark (studentID is 304).
    b. Search for Ed (studentID is 211).
    Both should be Found, as you have stored them in the table.
10. Orient the blocks on the left half of the screen so that the "Search Table" function is in view. Consider searching for a student whose ID is 467, not in the table. What index will this ID hash to? Observe the flashing of blocks as the subroutine is executed? Explain what is going on and why.
11. Assume that Mark (studentID 304) is no longer attending the school. We want to delete his information. Orient the blocks on the left half of the screen so that the "Delete Student" function is in view. Observe the flashing of blocks as the subroutine is executed? Explain what is going on and why. Select "Yes" on the screen to confirm the deletion.
12. Now let us make sure that Mark has been deleted by searching for his studentID 304. Explain what happens and why.
13. What is the current load factor of the hash table?
14. STOP the program.

Part 2

1. With the hash program stopped, change the *listSize* variable to 3. While you would never make such a small hash table in practice, we are doing so here so that you can investigate what happens when trying to add, search, or delete items from a table that is either empty or full. RUN the program with this new *listSize*.
2. With the table currently empty, try searching for Bev (studentID 253). What happens?
3. With the table currently empty, try searching for Jim (studentID 316). What happens?
4. When adding students to this table, what are the possible values for the index based on our hashing technique of finding the remainder when dividing the *studentID* by the *listSize*?
5. Add Carol (365), Greg (906), and Gloria (923) to the hash table. Then try to add Rich (421) to the table. What happens? At what indexes are Carol, Greg, and Gloria data stored, respectively.
6. Try searching for Sue (408). What happens?
7. Try searching for Greg (906). What happens?
8. Delete Gloria (923). What happens?
9. Try to delete Sue (408). What happens?
10. STOP the program.

Part 3

1. With the hash program stopped, change the *listSize* variable to 47.
2. Add all of the students in the table of Figure 1 to the hash table.
3. Were there any collisions?
4. What is the current load factor of the hash table? What do you conclude regarding the relationship between the number of collisions and the size of the hash table?

Learn More About Hash Tables Using Ozobot and a Large Language Model (LLM)

You may be wondering what an LLM is. It is a type of artificial intelligence (AI). According to Wikipedia, "A **large language model** (**LLM**) is a language model trained with self-supervised machine learning on a vast amount of text, designed for natural language processing tasks, especially language generation." LLMs provide some of the basic capabilities of chatbots including the popular ChatGPT. LLMs are also used in *prompt engineering*, as is the case with Ozobot Editor's LLM Bot.

You have already learned how to design a hash table with the add, search, and delete capabilities. But now you will see how to design a simple LLM Bot with the purpose of determining some *hash table applications*. In fact, you will quickly see that *hash table applications* will be the prompt, with the goal of obtaining examples of generated language from this prompt.

As shown in Figure 9, you will need to add either Evo or Ari and an LLM bot using the Ozobot Editor's *Devices* menu selection. The figure shows Ari_1 and LLM_Bot_1 as the selected devices.

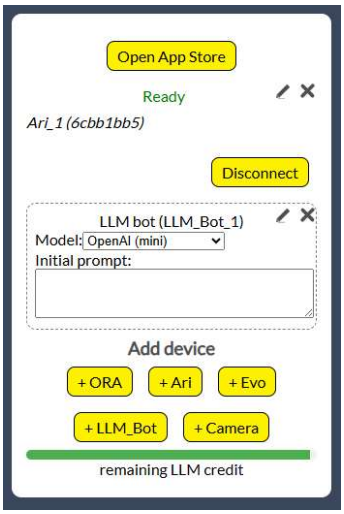

Figure 9

There are several language models you can use, but the *Open AI (mini)* is strongly encouraged. You will also note at the bottom of Figure 9 that you are allowed a certain amount of LLM credit each day. When that green bar turns red, you are running low and when it is gone, you will have to wait until the next day to work with your LLM bot.

Figure 10 shows the program on the left. It makes use of the *Devices* block "with actor … do." It also makes use of the *Exceptions* block "try to do … in case of exception do." The "print to IO console" block has asked the LLM to search the LLM for a string (without the use of a camera) based on the prompt "Hash table applications."

An ***exception*** is an event that occurs during the execution of a program, disrupting the normal flow of a program's logic. According to the Ozobot Editor's *Reference* guide, "the LLM block can fail for a variety of reasons (inappropriate content, LLM error, depleted credits)." For that reason, it is recommended that the LLM block be used within a *Try/Catch* block. In our little program, we have asked that the message LLM ERROR be printed to the IO console.

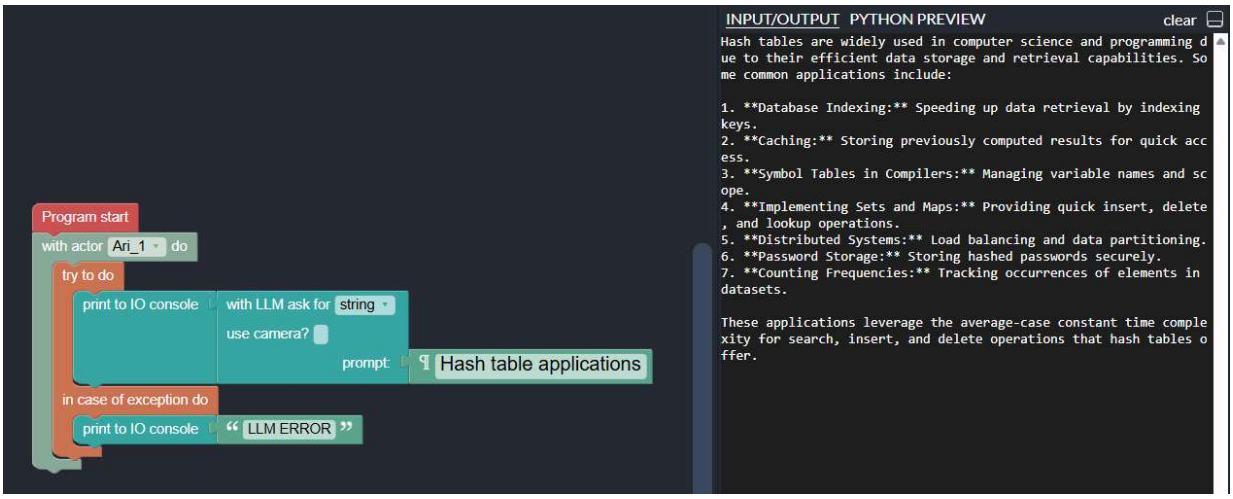

Figure 10

When the program is run, you will see language generated from the prompt "Hash table applications" in the INPUT/OUTPUT region on the right. Do not expect the language generated to be identical to that shown in Figure 10, but comparable results regarding hash table applications can be expected. Remember that the AI (artificial intelligence) LLM model is searching through thousands of documents and randomly selecting those deemed appropriate to the specific search prompt.

For example, when the program was run a second time, the language shown in Figure 11 was generated.

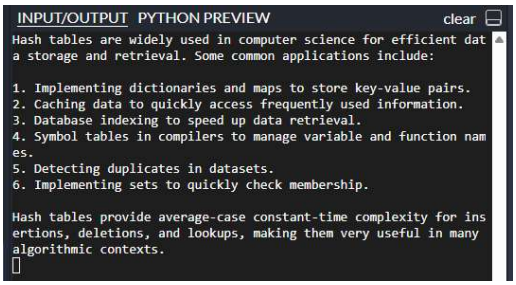

Figure 11

Here are a few more prompts related to hash tables. Try them out. You will be amazed at what you can learn about hash tables and the capabilities of large language models!

- What is the difference between separate chaining and open addressing in hash tables?
- Five true/false questions related to hash tables
- What is a technique to hash a letter to an index in a hash table?
- What are typical sizes for hash tables in real world applications?
- Why do hash table often have a size that is a prime number?
- How are hash tables used in spell checking?
- What color are hash tables?
- How can I describe a hash table to a third-grade student?
- Translate to Swedish: "Hash Tables For Dummies"
- Lesson plan for studying hash tables
- Provide a worksheet on hash tables
- What grade levels typically study hash tables?
- Can hash tables get dusty?
- Can hash tables get cluttered?
- Provide 3 short answer questions related to this paragraph: "A hash table is a data structure that stores data in (key, value) pairs. It allows for very fast insertion, searching, and deletion operations, even for very large tables. The key is converted into an index into the table by a process known as hashing. Then the value associated with that key is stored along with the key at the hashed address."
- True or False: In a well-designed hash table, the load factor affects the performance of operations. *[For this prompt, have the LLM ask for a **Boolean**—a true or false value.]*