Hash Tables
Open Addressing

Justification

Sorted arrays are fast to search but require sorting or the use of an insertion function to keep them in order. Unsorted arrays are easy to maintain, but expensive to search.

Another solution is to use the data itself to organize the array. For example, in the array below each location starts out false and as items are added to the array the locations are changed to true. This allows adding a new value, removing a value, or finding if a value is present – each in a single step.

Bool	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

The pseudocode algorithms for adding, finding, and removing an array are shown here.

```
addValue(value)
        theArray[value] = true

findValue(value)
        return theArray[value]

removeValue(value)
        theArray[value] = false
```

This works quite well, until the number of values being stored is excessive. For example, to keep track of students in a class at Lane using L numbers, it would require a table of 700,000 locations to handle all L numbers starting with L006.

This suggests a possible solution. Somehow create an index using the value. This is the basic idea behind the use of hash tables. Create a valid index by using some algorithm on the value. Such an algorithm is called a hash function. Since values are stored in the table at locations derived by this function, and the function depends on the value rather than the number of items currently in the table, the *performance of an ideal hash table is O(1).*

Simple Hash Table

One simple hash function for integers is to use the remainder function based on the size of the array.

```
hash(value)
```

```
return value % arraySize
```

This returns an index from 0 to arraySize-1, so works for indexing directly into an array.

For the next example, consider setting all array locations to -1 before starting. This does mean that the hash table only can hold positive integers, but it demonstrates the concepts. With such an array and the hash function shown above, the methods for adding, finding, and removing a function can be expressed in pseudocode as shown here.

```
addValue(value)
        theArray[hash(value)] = value

findValue(value)
        return theArray[hash(value)] == value

removeValue(value)
        theArray[hash(value)] = -1
```

The following table shows the result of adding 5, 23, 17 to the array, which has a size of 19:

Value	-1	-1	-1	-1	**23**	**5**	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	**17**	-1
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

All looks fine. But consider what happens when trying to add the number 24. Unfortunately, 24 % 19 == 5, which is the same as 5 % 19. If the new value is just stored, then there is a collision and data is lost. The solution is to come up with more complex algorithms for maintaining the data in the table.

Open Addressing with Linear Probing

This style of hash table is called **open addressing** since the hash function causes values to be stored directly in the array at the proper location. Another approach is separate chaining which is described in a different document.

The solution to the collision is to add some way of finding the next empty location to the add, find, and remove functions. The first one that will be covered is **linear probing**. In linear probing, if the first location is not available, the next one is checked and so on. If the index reaches the end of the array, then it wraps around to start at zero.

This is illustrated in the following table. When adding 24 to the array, the hash function returns the index 5. Since this is occupied, it increments by one and checks the contents of location 6. This location contains -1, so it is possible to store 24 in the array there as shown below. Then the value 4 is stored, but the locations at indexes 4, 5, and 6 are filled, so it must be stored at location 7.

Value	-1	-1	-1	-1	**23**	**5**	**24**	**4**	-1	-1	-1	-1	-1	-1	-1	-1	-1	**17**	-1
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

This works fine until the array becomes too full. As the array fills up, the performance decreases from the optimal O(1) to O(N) as it approaches a linear search. The optimal loading of the array is somewhere below 60%. If the array becomes totally filled, then searching for an empty spot is an infinite loop and fails.

Another problem is when removing an item, it is not possible to just store a -1 there. Consider what would happen if the 5 was removed by changing it to -1. Then a search for 4 would fail incorrectly.

Value	-1	-1	-1	-1	**23**	-1	**24**	**4**	-1	-1	-1	-1	-1	-1	-1	-1	-1	**17**	-1
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

The solution to this problem is to define a second constant to indicate a removed value. For example, EMPTY = -1, DELETED = -2. The resulting array would appear as shown below.

Value	-1	-1	-1	-1	**23**	-2	**24**	**4**	-1	-1	-1	-1	-1	-1	-1	-1	-1	**17**	-1
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Now a search for 4 steps across the DELETED value at index 5 and properly returns true. The pseudocode for adding, finding, and removing is shown below.

Open Addressing Pseudo Code

In this code, theTable contains the items, its size is arraySize, and the number of items currently stored there is maintained in count. If the table becomes more than 50% filled, the program throws an exception indicating there is insufficient room to continue. This is not optimal but works for this example.

```
// calculate an index from the value
hash(value)
        return value % arraySize
```

```
// hashes an item and adds to table
// uses open addressing with linear probing
// if table is approaching full, throws an exception
addItem(value)

        // arbitrarily quit when ½ full
        // alternatively, you could create new table and fill with any valid data
        if count >= arraySize / 2
                throw error("Table Full")

        // table has room, so add item
        // get the starting index
        index = hash(value)

        // loop until find an available slot
        // either empty or an item has been deleted
        while theTable[index] != EMPTY and theTable[index] != DELETED
                index += 1

                // wrap when you hit the top end of array
                if index >= arraySize
                        index = 0

        // increment count
        count += 1

        // now we have an available slot, so use it
        theTable[index] = value

// returns true if value is present in the table
```

```
// uses open addressing with linear probing
findItem(value)

        // get the starting index
        index = hash(value)

        // flag variable for loop
        done = false

        // loop looking at each stop
        // quit when find an empty spot without finding item
        while theTable[index] != EMPTY and !done

                // found it, go back and celebrate
                if theTable[index] == value
                        done = true

                // not it, keep looking
                // wrapping at the end of the array
                else
                        index += 1
                        if (index >= arraySize)
                                index = 0

        // empty spot or found it, return result
        return done

// remove item by marking it as deleted
// uses open addressing with linear probing
removeItem(value)
```

```
    // get the starting index
    index = hash(value)

    // flag variable for loop
    done = false

    // loop looking at each stop
    // quit when find an empty spot without finding item
    while theTable[index] != EMPTY and !done

        // found it, mark as deleted and decrement count
        if theTable[index] == value
            theTable[index] = DELETED
            count -= 1
            done = true

        // not it, keep looking
        // wrap at the end
        else
            index += 1
            if index >= arraySize
                    index = 0

// got to empty spot, so not there
return
```

Quadratic Probing and Double Hashing

As stated above, linear probing is when the index is increment by one each time there is a collision in the hash table. For many data sets, this causes clumping of values where the hash indexes are equal. Two other methods of stepping the index to find an empty location are quadratic probing and secondary hashing.

In ***quadratic probing***, instead of stepping forward 1 space at a time, the step varies, 1 the first time, 4 the second time, 9 the third time, and so on. This spreads out the values in the array when the collision does not occur due to hashing to the same value, but instead is due to the stepping forward hitting an occupied location.

In ***double hashing***, instead of a predefined amount that is stepped forward, the step is calculated using a secondary hash function. If the original table is of a prime size, then using a second prime for the secondary remainder function allows reaching all locations without repeating. This works well when the first hash method returns the same index for multiple sets of items.

Rehashing when full

Rather than throwing an exception, which is not very user friendly, if the hash table gets too full, it is possible to create a new table and copy the items over. If this is not done, as stated before the performance degrades from O(1) to O(N) to finally infinite time when the array fills completely up.

For the exercises in this course, double the size whenever the array gets more than 50% filled.

Since the items were placed based on the remainder after dividing by table size, it is necessary to calculate a new hash location for each value. So, it is not possible to simply create a new table twice as large and move the values over.

Instead, it is necessary to define a new hash function, and using it, walk through the array and store any values into the new array. Obviously, locations containing EMPTY or DELETED are ignored.

A good design plan is to have the array sizes be prime numbers to allow better spreading out of data. Rather than calculating a new prime twice as large, it is convenient to store a few sizes ahead of time (19, 37, 67, 131, …).

Hash Functions for Strings

Hash tables are commonly used to store strings or other data types rather than integers. Thus, it is necessary to create a hash function that is more complex than a simple % operator. This explanation will show how a hash function could be created for string values.

The first step would be to simply use the letters as numbers, add them, and use the % function. This is shown here for the two strings “cat” and “dog” where the a = 1, b = 2, and so forth. Also, consider an arraySize of 19.

The string cat is 3 + 1 + 20 → 24 → 5.

The string dog is 4 + 15 + 7 → 26 → 7.

This appears to work, but what happens with different words with the same letters, but in varying orders, such as cat, act, and tac. These all would hash to the same index 5. This is similar to considering numbers the same simply because they have the same digits, but normal usage is that 365 is not the same as 536. Instead, in writing numbers, the ordering defines what the value is.

If this was applied to letters, using the same a = 1, b = 2, and assuming there are a total of 26 letters, the results would change as shown by comparing cat and act.

cat == $c * 26^2 + a * 26 + t$ == $3 * 676 + 1 * 26 + 20$ == 2074
act == $a * 26^2 + c * 26 + t$ == $1 * 676 + 3 * 26 + 20$ == 774

Then apply % arraySize to reduce this value to an index that fits in the hash table.

Obviously, for long words, the calculations could cause integer overflow since each additional letter in length adds another power of 26. Instead, there is a simple approach where the % is applied after each step.

Also, to make calculations simpler, instead of multiplying by 26, it would be possible to shift the value 5 places, which is equivalent to multiplying by 32. To allow any ascii character to be used, simply multiply by 128 (shift 7). This code shows the multiplication for simplicity.

Hash Function Pseudo Code

This has the same variable arraySize for the size of the hash table.

```
hashFunc(key)

    // initialize index
    hashValue = 0

    // walk through string one char at a time
    for i = 0; i < key.length(); i++

        // multiply current sum
        hashValue *= 128

        // add current character's ascii value
        hashValue += key[i]

        // shrink to fit
        hashValue %= arraySize

    // return the result
    return hashValue
```