CS261L Data Structures and Algorithms (Pr)

Lab Manual (Week 7)

Instructor:

- Mr. Samyan Qayyum Wahla

Registration No.__________________

Name: _________________________

Guide Lines/Instructions:

You may talk with your fellow CS261-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

Today's Task:

- Implementation of Hash Tables
- Linear Time Sorting

Part 1: Hash Tables Implementation

Hash tables are among the most common and most useful general-purpose data structures. A hash table is a generalization of an array. An array can be thought of as a data structure that maps integers to items of a given type. A hash table can map arbitrary (constant) items of one type (known as keys) to items (not necessarily constant) of another type (known as values). Hash tables are most often used to map strings to data, and that's what we'll be doing here. You can think of this as if you had an array which was indexed by a string value instead of by an integer. Note that hash tables are not the only way to build such a mapping, but they are one of the most efficient.

Hash functions

The basic idea behind a hash table is this. You take your key and process it with a special function called a hash function. This function will transform any key into a non-negative integer with a bound which is proportional to the size of the table. In our case it will transform a string into a value between 0 and 127. Ideally, a hash function will map different keys to different hash values so that a random population of keys will give a uniform distribution of hash values.

The data structure

The kind of hash table we are implementing in this lab manual is essentially an array of objects. In fact, we define it as a class which contains an array of objects as its only field.

The reason we use a class containing an array instead of just using a plain array is that in a real hash table implementation, there would be more fields. For instance, a real hash table would have (at a minimum):

- a field representing the total number of entries in the hash table
- a field representing the size of the array

Every time a key/value pair was added to the hash table, the field representing the total number of entries would be incremented. Once the number of entries was sufficiently larger than the size of the array (say, more than twice as large), then a larger array would be allocated and all the entries in the original array would be copied to the new array (this is called rehashing). Doing this ensures that hash table lookups are fast.

For the purposes of this assignment, though, we will just use an array inside a class. The main reason for this is that it gives you experience working with more complex data structures.

The array indices are the same as the hash values (so in our case the array length is exactly 128), and the lists associated with each index are initially empty (set to NIL).

Adding items to a hash table
We add items to a hash table as a "key/value pair". Let’s say our keys represent movie names and our values represent the rating of the movie in stars (0-5). A typical key/value pair might be ("Movie Title", 5). First, we compute the hash value for "Movie Title" (you’ll see how below) and get (say) 47. The 47th location in the node pointer array will initially be empty (i.e. it will contain a NIL value). We then create a new list at this index. The object being inserted in the list here will have two fields: a field called key(string), a field called value which holds the int value. We put the (address of the) key string into the key field (here, "Movie Title"), and the value into the value field (here, 5). We then link it to location 47 in the hash table array by putting a pointer to the node in the node pointer array at location 47. Later, we might want to add another entry, say ("Movie Title 2", 0). First, we compute the hash value of " Movie Title 2". Now one of two things could happen:

If the hash value is not 47, we create another list with the appropriate key and value fields and add it to the place in the array corresponding to the hash value, just as we did for the first item.

If the hash value is 47 again (which ideally it won’t be), we create a new object. Then we have to add it to the old list at array position 47. We do this by inserting the new object into the old list at any point (the order doesn’t matter). Usually we either put it at the front or the back of the list.

Now you can see why hash values should be distributed randomly over the set of all possible keys: if all keys hashed to 47, we would end up with a list at one location in the array and nothing anywhere else. In that case, looking up elements would take time proportional to the number of entries. If most of the lists are either empty or have only one or two values, lookup will require a constant (small) amount of time. Of course, as the number of elements in the hash table increases in size, eventually all the slots will be occupied even with a perfect hash function. At this point (as mentioned above) you should probably create a bigger hash table.

Retrieving items from a hash table
To retrieve an item from a hash table means to retrieve the value given the key. The way this is done is as follows:

- Compute the hash value for the key.
- Find the location in the list corresponding to the hash value.
- Search the list at that location for the key. In general, there will be a very small number of items in the list (usually one or none), so the search will not take long.
- If the key is found, return the value. If not, return a "not found" value (this will become clearer later). If there was no list at that location in the array, also return the "not found" value.

As we said above, this means that our list objects have to have two elements:

- the key,
- the value associated with the key

Program to write
Your program will use a hash table to count the number of occurrences of words in a text file. The basic algorithm goes as follows. For each word in the file:

Look up the word in the hash table.

- If it isn't there, add it to the hash table with a value of 1. In this case you'll have to create a new object and link it to the hash table as described above.
- If it is there, add 1 to its value. In this case you don't create a new object.
- At the end of the program, you should output all the words in the file, one per line, with the count next to the word e.g.

```
cat 2
hat 4
green 12
eggs 3
ham 5
algorithmic 14
deoxyribonucleic 3
dodecahedron 400
```

The hash function
The hash function you will use is extremely simple: it will go through the string a character at a time, convert each character to an integer, add up the integer values, and return the sum modulo 128.

(Don't hard-code the number 128 into your code, use a variable)
This will give an integer in the range of [0, 127]. This is a poor hash function; for one thing, anagrams will always hash to the same value. However, it's very simple to implement. Note that a value of type char in a python can also be treated as if it were an int, because internally it's a one-byte integer represented by the ASCII character encoding. If you like, you can convert the char to an int explicitly. However, a string of characters is not an int, and you can't simply convert it into one (mainly because most of the keys will be words, not string representations of numbers). You also shouldn't try to type cast the string to an int. A string is an array of chars, which is not the same as a single char; keep that in mind. So you'll need a loop to go through the string character-by-character.

Note: Explore the anagrams yourself.

Other details
Since a value associated with a key will always be positive, if you search for a key and don't find it in a hash table, just return 0. That's the special "not found" value for this hash table.

For simplicity, the program assumes that the input file has one word per line (we've written this code for you). The order of the words you output isn't important.

What to Submit:

1. Create a class KeyNode containing following attributes
    a. Key(String)
    b. Value(int)
2. Create a class MyHashTable, with the following attributes.
    - Array to list of KetNode objects
    - Size of HashTable
    - KeysOccupied(initially zero)
3. MyHashTable Contain the following functions
    a. Constructor(hsize) – create the array of size hsize
    b. GetHashTableSize()
    c. GetNumberOfKeys()
    d. HashFunction() – logic to calculate hash value
    e. UpdateKey(key, value)
    f. SearchKey(key)—returns value
    g. Rehash()—in case of hash table is full
4. Now use the MyHashTable to write the above program to count the words occurrence.
5. Write the perfect hash function for your task, justify your answer.

Part 2: Linear Time Sorting

Counting Sort Algorithm

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Pseudo Code

```
function CountingSort(input)
k =  range of elements of array
count ← array of $k$ + 1 zeros
    output ← array of same length as input

    for $i$ = 0 to length(input) - 1 do
        $j$ = key(input[$i$])
        count[$j$] += 1

    for $i$ = 1 to $k$ do
        count[$i$] += count[$i$ - 1]

    for $i$ = length(input) - 1 down to 0 do
        $j$ = key(input[$i$])
count[$j$] -= 1
        output[count[$j$]] = input[$i$]

    return output
```

Radix Sort Implementation

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

Algorithm

```
Do following for each digit i where i varies from least significant digit to the most significant digit.
    Sort input array using counting sort (or any stable sort) according to the i’th digit.
```

Bucket Sort

Bucket sort is mainly useful when input is uniformly distributed over a range.

Algorithm

```
bucketSort(arr[], n)
1) Create n empty buckets (Or lists).
2) Do following for every array element arr[i].
.......a) Insert arr[i] into bucket[n*array[i]]
3) Sort individual buckets using insertion sort.
4) Concatenate all sorted buckets.
```

Exercise

Sort the array using counting sort algorithm.	`Input: arr[] = [-5, -10, 0, -3, 8, 5, -1, 10]`
	`Output:[-10, -5, -3, -1, 0, 5, 8, 10]`

Sort the array using radix sort.	`Input: arr[] =[110, 45, 65,50, 90, 602, 24, 2, 66]`
	`Output: [2,24,45,50,65,66,90,110,602]`

Sort the array using Bucket Sort	`Input: arr[]=[0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434]`
	`Output: [0.1234, 0.3434, 0.565, 0.656, 0.665, 0.897]`