**COMP232 – Data Structures & Problem Solving**

**Homework #8 - Solutions**

**Hashing**

1. Explain the importance of having a hash function that approximately satisfies the simple uniform hashing assumption?

A hash function that approximately satisfies the simple uniform hashing function will distribute keys uniformly among the buckets in the hash table. In the case of linear probing, such a hash function will reduce clustering (both primary and secondary). In the case of chaining it ensures that all of the chains are of approximately equal length. The less clustering or the more even the length of the chains the more likely the average cost of a get operation will be O(1) in practice.

2. Consider the following sequence of Map operations to be applied to a hash map with capacity 10 and the simple hash function h(k)=k:

add(57, “A”)

add(107, “X”)

add(27, “Q”)

add(37, “B”)

add(88, “M”)

add(89, “R”)

add(100, “Z”)

add(66, “N”)

a. Show the contents of the hash map after this sequence of operations if it were using open addressing with linear probing. Assume that the hash table does not resize.



b. What elements are in a primary cluster?

The definition of primary clustering says that it occurs when different keys hash into the same probe sequence. This would occur for any key hashing into indices 6-9 or 0-3. Thus, all of the values in this table form a large primary cluster.

c. Show the contents of the hash map after this sequence of operations if it were using closed addressing with chaining. Assume that the hash table does not resize.



d. What is the load factor of the hash tables in parts a and c?

The load factor is given by the number of elements in the hash table (n) divided by the capacity the table (m). In this case, the load factor for part’s a and b are the same:

λ = 8 / 10 = 0.8

Note: The capacity of the table is distinct from the number of occupied cells in the table. In both cases above the capacity is 10.

3. In class we saw that for a hash map using closed addressing with chaining the average case for the get operation had an asymptotic bound of O(1).

a. What is the worst-case asymptotic bound for get in this implementation of hashing? Explain.

The worst case asymptotic bound for get with closed addressing with chaining would be O(n). In the worst case every key would have the same hash code and thus would hash to the same bucket. So in a table with n elements the chain in that bucket would have length n. To get a value, the bucket to be searched can be found in O(1) time, however, to find the desired value it might be necessary to traverse the entire chain of elements within the bucket, requiring O(n) time.

b. Explain how could you obtain a worst case bound of O(lg n) for the get operation with closed addressing?

The worst case performance of closed addressing with hashing can be improved to O(lg n) by using an AVL (or other balanced BST) instead of a linked list for chaining values that hash to the same bucket. If an AVL tree is used, the bucket to be searched is still be found in O(1) time however, the search within the bucket can be completed in O(lg n) time because of the BST structure.

c. Explain why the added overhead of your approach in part b be worth it, or not?

It is unlikely that the added overhead of using an AVL tree would be worth it. A good hash function will ensure that the keys are spread approximately uniformly across the buckets. Thus, the number of values in each bucket should be small. For small numbers of elements the O(n) scan of the linked list is likely to be faster than the O(lg n) search of the AVL tree (remember the constants hidden by the asymptotic notation). Further, the bound for a put operation would become O(lg n) if an AVL tree were used as opposed to O(1) when a simple linked list is used.

4. Discuss the major tradeoff that exists between the use of open and closed addressing.

The major tradeoff between open and closed addressing is one of time versus space.

With linear probing the direct random access to array elements will make access very fast provided that clusters are small. However, to ensure that clusters are small it is necessary to keep the load factor small, which implies that many buckets will be empty (i.e. wasted space). Typically a load factor of 50% works well, so 50% of the buckets are empty. So while we may save time, it costs space.

With closed addressing the load factor can be higher (possibly > 1), reducing the number of empty buckets and thus reducing wasted space. This space savings may however, be mitigated by the need to store the links in the linked list. As the load factor increases so will the length of the chains and thus the time required traverse them during put and get operations will become longer. So while we may save space it costs time.

There is also a complexity/clarity trade off. The code for closed addressing is simpler to understand and does not rely on a good probe function for good performance and thus may be simpler to implement.

5. As we saw in class the hash map operations with open hashing run in O(n/m) time. If we can assume Simple Uniform Hashing and ensure that n<=m then this reduces to O(1) expected time per operation. The current implementation in CS232OpenHashMap does not guarantee that n<=m because the hash table is never resized. It thus does not guarantee O(1) average time per operation. Add resizing to the put operation such that the hash table is doubled in size whenever the load factor equals or exceeds the MAX\_LOAD\_FACTOR. The No5Tests class contains tests that you can use to check your implementation of this functionality. All but 2 of the tests pass using the provided code because they do not rely in the resize functionality. These tests are included, along with those that test the resizing, to ensure that the resize operation does not break the existing functionality.

The following lines are added to the put method after the if that throws the exception and before the call to getKeyvaluePair:

/\*

\* If the current load factor >= MAX\_LOAD\_FACTOR then resize the hash

\* table to make room for new elements.

\*/

**if** ((currentSize / capacity()) >= *MAX\_LOAD\_FACTOR*) {

resizeHashTable();

}

The resizeHashTable method is as shown on the following page:

/\*

\* Resize the hash table. Make a new hash table that is twice as big and

\* rehash all of the values from the old one into the new one.

\*/

**private** **void** resizeHashTable() {

/\*

\* Keep a link to the current hashTable and then make the new one. It is

\* important to do it this way because the getIndex() method relies on

\* the length of the hashTable.

\*/

CS232IterableDoublyLinkedList<KeyValuePair<K, V>>[] oldHashTable =

hashTable;

hashTable = (CS232IterableDoublyLinkedList<KeyValuePair<K, V>>[]) **new**

CS232IterableDoublyLinkedList<?>[capacity() \* 2];

**for** (**int** i = 0; i < hashTable.length; i++) {

hashTable[i] = **new** CS232IterableDoublyLinkedList<KeyValuePair<K, V>>();

}

/\*

\* Just created a new empty hash table, so no contents currently. But we

\* will be using put to rehash the values from the current hash table to

\* the new one. Each value that is rehashed via put will add one to the

\* current size in the put method.

\*/

currentSize = 0;

/\*

\* For each entry in the old hash table, iterate over the key,value

\* pairs in the linked list that is stored there. For each key,value

\* pair put it into the hashTable, which will cause it to be rehashed

\* into the correct location in the new table.

\*/

**for** (CS232IterableDoublyLinkedList<KeyValuePair<K, V>> kvList :

oldHashTable) {

CS232Iterator<KeyValuePair<K, V>> it = kvList.getIterator();

**while** (it.hasNext()) {

KeyValuePair<K, V> cur = it.next();

**this**.put(cur.key, cur.value);

}

}

}

6. Complete the remove method in the CS232OpenHashMap so that it executes in O(1) expected time. In practice, the remove method will decrease the size of the hash table to reduce wasted space if the load factor becomes too small. It is not necessary to implement this reduction in the size of the hash table for this problem. The No6Tests class contains tests that you can use to check your implementation of this functionality. All of the tests from No5Tests are also run to ensure that the remove operation does not break the existing functionality.

**public** V remove(K key) {

/\*

\* NOTE: Should resize to be smaller if there are lots of unused

\* entries. But to keep it simple we aren't going to worry about that

\* here.

\*/

// Get the index to which the key hashes.

**int** index = getIndex(key);

// Use and iterator to go through the linked list and find the key.

CS232Iterator<KeyValuePair<K, V>> it = hashTable[index].getIterator();

**while** (it.hasNext()) {

KeyValuePair<K, V> kvp = it.next();

**if** (kvp.key.equals(key)) {

/\*

\* found it... adjust size, remove it and return the associated

\* value.

\*/

currentSize--;

**return** it.remove().value;

}

}

// key was not in the list, so key,value pair is not in the map.

**return** **null**;

}

7. With open hashing we perform a linear search of a doubly linked list during put, get and remove operations. These are O(n) operations, where n is the number of items in the list. We know that an AVL tree can perform the add, get and remove operations in O(lg n) time. Would it be beneficial to implement the lists for open hashing using AVL trees instead of linked lists?

Ooops… this is the same as question #3b. Sorry.

8. Implement the following methods in the CS232ClosedHashMap using linear probing:

a. put and get. You do not have to implement the resizing of the hash table in the put method. The No8aTests class contains tests that you can use to check your implementation of this functionality.

A final solution for put and get along with a helper method (probeForIndexOfKey) are shown on the following pages.

b. remove with appropriate modifications to put and get to deal with deleted elements. The No8bTests class contains tests that you can use to check your implementation of this functionality.

A final solution for remove, which also uses the helper method (probeForIndexOfKey) is shown in the solution on the following pages.

/\*

\* Helper method that gets the index of the KeyValuePair matching the key if

\* it exists. If the key is not in the table this method returns either the

\* index of the first null or (possibly) the first DEL encountered. If

\* skipDEL is true then DEL values will be skipped over as if they contain a

\* key,value pair. This allows put, get and remove to use this single helper

\* function.

\*/

**private** **int** probeForIndexOfKey(K key, **int** homeIndex, **boolean** skipDEL) {

// Compute the cur index in case the probe function includes a constant.

**int** curIndex = (homeIndex + probeFunction(key, 0)) % hashTable.length;

/\*

\* Starting at the homeIndex, probe using the probeFunction while the

\* current index is not empty (null or del) and is not the key. I.e.

\* stop when we find an empty spot (key is not in table) or we find the

\* key.

\*

\* NOTE: Okay to use == with DEL instead of .equals. The same DEL object

\* is used over and over, so comparing the references with != has the

\* desired effect.

\*/

**int** probeNum = 0;

// Some local vars to help simplify the logic...

**boolean** isNull = hashTable[curIndex] == **null**;

**boolean** isDEL = hashTable[curIndex] == DEL;

**boolean** isKey = !isNull && !isDEL

&& hashTable[curIndex].key.equals(key);

/\*

\* Done probing when we find a null or when we find a DEL (and we are

\* not skipping them) or we find the key.

\*/

**boolean** doneProbing = isNull || (isDEL && !skipDEL) || isKey;

**while** (!doneProbing) {

// get the next index based on the probe function...

probeNum++;

curIndex = (homeIndex + probeFunction(key, probeNum))

% hashTable.length;

isNull = hashTable[curIndex] == **null**;

isDEL = hashTable[curIndex] == DEL;

isKey = !isNull && !isDEL && hashTable[curIndex].key.equals(key);

doneProbing = isNull || (isDEL && !skipDEL) || isKey;

}

**return** curIndex;

}

**public** **void** put(K key, V value) {

// NOTE: Does not currently resize.

**if** (key == **null**) {

**throw** **new** IllegalArgumentException("key cannot be null.");

}

// get the home index of key.

**int** homeIndex = getIndex(key);

/\*

\* Get the actual index for the key. This will either be the index of

\* the key if it already exists in the table, or the index of the first

\* null or DEL encountered in the probe sequence.

\*/

**int** index = probeForIndexOfKey(key, homeIndex, **false**);

**if** (hashTable[index] == **null** || hashTable[index] == DEL) {

/\*

\* The key does not exist in the hash table, so put a new key,value

\* pair into the index.

\*/

hashTable[index] = **new** KeyValuePair<K, V>(key, value);

currentSize++;

} **else** {

/\*

\* The key already exists in the hash table, so replace the value

\* associated with the key with the new value.

\*/

hashTable[index].value = value;

}

}

**public** V get(K key) {

// get the home index of key.

**int** homeIndex = getIndex(key);

/\*

\* Get the actual index for the key. This will be either the index of

\* the key or the index of the first null value encountered on the prob

\* sequence.

\*/

**int** index = probeForIndexOfKey(key, homeIndex, **true**);

**if** (hashTable[index] == **null**) {

// key was not found in the hash table.

**return** **null**;

} **else** {

// key was found in hash table, so return value.

**return** hashTable[index].value;

}

}

**public** V remove(K key) {

// get the home index of key.

**int** homeIndex = getIndex(key);

/\*

\* Get the actual index for the key. This will be either the index of

\* the key or the index of the first null value encountered on the probe

\* sequence.

\*/

**int** index = probeForIndexOfKey(key, homeIndex, **false**);

**if** (hashTable[index] == **null**) {

// key was not found in the hash table.

**return** **null**;

} **else** {

// found it, so remove it by replacing it with a DEL element.

V val = hashTable[index].value;

hashTable[index] = DEL;

currentSize--;

**return** val;

}

}

9. In class we claimed that the put operation for open hashing with chaining runs in Θ(1) average time under the Simple Uniform Hashing assumption. Give an analysis that shows that this is true. In your analysis, assume that the hash table is resized when the load factor > 1.0.

For open hashing with chaining our claim was that put operations would take Θ(1) average time under the Simple Uniform Hashing assumption. Without considering resize operations the time required for a put will be:

T(n) = O(1) + O(λ) with λ=(n/m)

Where n = number of key,value pairs in the map and m = the capacity of the hash table. If we can ensure λ ≤ 1 then T(n) will be O(1). To ensure that λ ≤ 1 requires resizing the hash table any time that λ > 1.

Resizing the hash table when λ > 1 (i.e. n > m) requires O(n) time. Thus, put will also require O(n) time in the worst case. However, as we saw when analyzing the add operation for ArrayLists, which resize in a similar way, the average case is much better because the resizing happens rarely.

Following the same technique as the analysis of add for ArrayList, this analysis will find the time required for 2k put operations and divide the result by 2k to find the average time for a put.

We assume an initially empty hash table with capacity m and take k = m. Also, note that under the uniform hashing assumption, the length of each of the chains will be n/m.

For the first k put operations n ≤ m, so λ = n/m ≤ 1. Thus, thus each of these k put operations requires O(1) expected time, or k\*O(1) total time. When the k+1st put operation occurs n = k + 1 > m and the hash table will be resized and all of the elements will be rehashed. This requires O(k+1) = O(k) time plus O(1) to put the k+1st element after the resize. For the remaining k-1 put operations n ≤ m and thus each requires O(1) expected time, or (k-1)\*O(1) total time.

So, the total time for the 2k put operations is:

T(n) in k\*O(1) + O(k) + O(1) + (k-1)\*O(1)

in 2k \*O(1) + O(k)

in O(k)

Finally, dividing by the 2k operations, gives the result that the average time for each put operation is in O(1).

10. Consider each of the following possible hash functions where m is the capacity of the underlying hash table:

a. h(k) = k/m

b. h(k) = 1

c. h(k) = (k + rnd.nextInt(m))

rnd.nextInt(m) gives a uniformly distributed random integer in the range [0…m).

For each of these hash functions, discuss why it would not be a good hash function to use.

a. h(k) = k/m

This can be a reasonable hash function if the values of k are uniformly distributed and the range of k is sufficiently large with respect to m. However, if the range of k is small, then many of the keys will hash to a small number of the indices. For example, if k=[0…2m-1] then all of the keys will hash to index 0 or 1. Similarly, even if the range of k is large with respect to m, if the values are not evenly distributed then they will not hash uniformly across the table.

b. h(k) = 1

This hash function hashes every key to the same index. This does a very poor job of approximating the simple uniform hashing assumption and will lead to very long chains or probe functions. In fact, put, get and remove operations will degrade to O(n) operations under this hash function.

c. h(k) = (k + rnd.nextInt(m))

rnd.nextInt(m) gives a uniformly distributed random integer in the range [0…m).

This hash function hashes a key to a random index. This closely approximates the simple uniform hashing assumption. However, because we cannot be certain that the hash index generated for a key will be the same each time all of the operations may require a full search of the table and thus degrade to O(n) operations.