CS345, Spring' 19 22 March 2019

Lab Assignment: Extendible Hashing with Delayed Directory Expansion

We assume that you have working code for extendible hashing with you.

Modify this code to accommodate following specifications.

Suppose that to accommodate a new entry a bucket Y does not have available space. The traditional extendible hashing splits this bucket.

If the LocalDepth(Y) < GlobalDepth(Hash Table) then we can split the bucket Y without doubling the directory structure. Otherwise, we have to double the directory structure. As a result, the directory structure can grow quickly. We want to delay the directory doubling as follows.

Each bucket can now potentially have multiple overflow buckets attached to it. Initially, no bucket has the attached overflow bucket. When a bucket Y is full and we need to insert an additional entry, following two scenarios are possible.

Scenario 1: LocalDepth(Y) < GlobalDepth(Hash Table)

In this case, simply split the bucket. We will follow traditional extendible hashing.

Scenario 2: LocalDepth(Y) = GlobalDepth(Hash Table)

Within scenario 2, following are the possible case.

Case 2.1: Y already has at least one attached overflow bucket

In this case, insert the new key into the overflow bucket. If the overflow bucket is also full then attach one more overflow bucket.

Case 2.2: Y has no attached overflow bucket

Within the case 2.2, following subcases are possible.

Case 2.2.1: If all other buckets already have at least one overflow bucket attached, then double the directory structure and split all the buckets. Unlike traditional extendible hashing, if the crowding in bucket Y is still not solved, we will not allow multiple times doubling of the directory. In other words, while inserting a key, if we will never quadruple the directory structure. To accommodate the insertion of a key, the directory structure remains the same or at the most doubles.

Case 2.2.2: We will consider case 2.2.2 only if case 2.2.1 does not arise. In this subcase, simply attach an overflow bucket to bucket Y. The overflow bucket has the same capacity as a regular bucket. Insert, new entry in the bucket.

A Hint: Do you really have to write code for scenario 1?

Note on code submission:

Make sure that all your code fits into a single file <roll number>.cpp

How your code should compile?

g++ <roll number>.cpp

How your code should run?

a.out < input_file_name

Your code should write the output on the stdout

A TA will collect your code on a USB drive. Make sure that you compute md5 hash of your code file before you submit. Use command md5sum <filename>.

Submit code to TA only when you are sure that it is your final code. We will not entertain requests to accept newer version of the code.

Consider the example input sequence of keys given in the file new_input.txt:

We will first create an empty hash table with two buckets and each bucket having capacity of three keys. (Lines 1 to 3)

Then we insert keys 128, 256, 1024, 127, 255, and 1023 into the hash table. (Lines 4 to 15)

To insert key 512, we have to attach an overflow bucket to the bucket 0. (Lines 16 to 17)

This is an example of subcase 2.2.2

To insert key 2048, we accommodate it in the already existing overflow bucket. (Lines 18 to 19)

This is an example of subcase 2.1

Line 20 asks to display the hash table structure. It should show a hash table with two buckets.

Bucket 0: 128, 256, 1024, 512, 2048

Bucket 1: 127, 255, 1023

Line 21 and 22 ask for the insertion of key 2047. The key maps to bucket 1. However, bucket 1 is already full. The bucket 1 does not have any attached overflow bucket yet. Rest all other buckets have at least one other overflow bucket attached. This is an example of subcase 2.2.1

For this insertion, we did not consider case 2.2.2 because first we have to check if case 2.2.1 is true.

Now we double the directory structure and split all the buckets.

Line 23 asks to display the hash table structure. It should show hash table with four buckets.

Bucket 0: 128, 256, 1024, 512, 208

Bucket 1: Empty

Bucket 2: Empty

Bucket 3: 127, 255, 1023, 247

Note that, we did not quadruple the directory structure.

Lines 24 through 35 insert keys 41, 42, 45, 46, 49, and 50. These insertions are accommodated in buckets 1 and 2.

Line 36 and 37 ask for insertion of key 53. This is an example of subcase 2.2.2

Lines 38 through 45 ask for insertion of keys 57, 61, 65, and 69. All these are examples of subcase 2.2.2

Line 46 asks to display the hash table structure. It should show hash table with four buckets.

Bucket 0: 128, 256, 1024, 512, 208

Bucket 1: 41, 45, 49, 53, 57, 61, 65, 69

Bucket 2: 42, 46, 50

Bucket 3: 127, 255, 1023, 247

Line 47 and 48 ask for insertion of key 54. The target bucket is bucket 2. The bucket is already full. It does not have any attached overflow bucket yet. Rest all other buckets have at least one other overflow bucket attached. This is an example of subcase 2.2.1

For this insertion, we did not consider case 2.2.2 because first we have to check if case 2.2.1 is true.

Now we double the directory structure and split all the buckets.

Line 49 asks to display the hash table structure. It should show hash table with eight buckets.

Bucket 0: 128, 256, 1024, 512, 208

Bucket 1: 41, 49, 57, 65

Bucket 2: 42, 50

Bucket 3: Empty

Bucket 4: Empty

Bucket 5: 45, 53, 61, 69

Bucket 6: 46, 54

Bucket 7: 127, 255, 1023, 247

Line 50 asks to terminate the program.

Note that order of elements in each bucket is not important. No need to display the local depth of each bucket in the output.