CS29003 ALGORITHMS LABORATORY
Assignment No: 4
Last Date of Submission: 19–August–2015

This assignment deals with storing strings in a one-dimensional hash table T using an algorithm that gives constant-time performance for search (worst-case), insert (expected—well, amortized), and delete (worst-case). From time to time, you may have to rehash. When the effort of rehashing is averaged over insert operations, this gives constant-time overhead per insert operation.

Part 1: Define a data type to store a hash table T. The data type contains the following fields: the allocation size $s = 2^t$, the logarithm $t = \log_2 s$ of s, the number n of elements stored in T, two odd integer parameters a and b, and a dynamic array *data* of strings (character pointers). We call $\lambda = n / s$ the load factor of T (this need not be stored explicitly). The roles of a and b will be clear in Part 2. The array *data* must be allocated memory to store exactly s character pointers. When the allocation size grows (see Part 5), this array should be reallocated memory to store the increased number of pointers. Write a function *init*() in order to initialize your hash table T. Take $s = 8$, $t = 3$, $n = 0$, $a = 1$, and $b = 3$, initially. Allocate memory to *data* so as to store exactly eight character pointers. Make all these pointers NULL (empty location).

Part 2: Write a function *hash*(str, c, t) in order to hash a string *str* to an index in the range 0 … $s - 1$. Here, t is as defined in Part 1, and c is either a or b. First, the string *str* is converted to a 32-bit unsigned integer m. Let l be the length of *str*. Take $m_{-1} = 0$, and for $i = 0, 1, \ldots, l - 1$, compute $m_i = (Am_{i-1} + str[i]) \bmod 2^{32}$, where $A = 2^{16} + 2^8 - 1 = 65791$, and $str[i]$ stands for the ASCII value of this character. Finally, take $m = m_{l-1}$. The 32-bit value m is still large enough to be an index in the hash table. Multiply m by c (mod 2^{32}), and return the most significant t bits of this product as the hash of the string *str*.

Note: If you multiply two 32-bit unsigned integers x and y, and store the result in another 32-bit unsigned integer z, you actually get the least-significant 32 bits of the product xy, that is, $xy \bmod 2^{32}$. The product xy may be 64 bit long. Just assign it to z. Code: `z = x * y`. The most significant t bits from z are extracted as `z >> (32 - t)`.

Part 3: Write a function *search*(T, *str*) to find whether the string *str* exists in T. As will be evident from Part 4, *str* can reside in one of the two indices *hash*(*str*, a, t) and *hash*(*str*, b, t). Go to these locations in the *data* array, and find out whether *str* resides there.

Part 4: Write a function to insert a string *str* in T. If *str* is already present in T, make no changes. Otherwise, start by running the algorithm outlined below. In order to make insert efficient, the loop must run for only a constant number of iterations. You may take MAX_ITER_CNT as ten.

> Repeat MAX_ITER_CNT times:
> Compute the two indices $i = hash(str, a, t)$ and $j = hash(str, b, t)$.
> If either the i-th or the j-th location in *data* is empty (NULL), insert *str* there, rehash (if necessary), and return.
> Insert *str* at the i-th position dislocating $data[i]$ which is renamed as *str* for insertion in the next iteration.

If insertion is successful in one of the above iterations, the size n increases by 1. If the load factor $\lambda = n / s$ exceeds half, rehash with double allocation size s (before returning)—the parameters a, b do not change. If the rehashing attempt fails, report insertion failure. If MAX_ITER_CNT iterations still leave a string *str* to be inserted, first make a rehash with changed parameters a and b (the size s does not change). If the rehashing attempt fails, make a second rehashing attempt with double allocation size s—the parameters a, b will continue to retain their last changed values. This second attempt is a desperate measure to accommodate a new string, and will be carried out even if the load factor is $\leq ½$. If the second attempt too fails, report failure to insert, and return. Otherwise, insert *str* in the rehashed table. The rehashing algorithm is explained in Part 5.

There remains a possibility that despite rehashing, insertion always fails. That is, there is an infinite loop of nested calls insert-rehash-insert-rehash-insert-rehash-… In practice, such a situation is extremely improbable, in particular, because the size of the hash table grows by a factor of two in each failed rehashing attempt. Anyway, you abort the insertion effort after MAX_REC_LEVEL (like four) insert-rehash attempts turn out to be unsuccessful.

Part 5: Write a function *rehash*() in order to rehash (reorganize) the stored strings in the *data* array. A rehash attempt may be of two types: CHANGE_PARAMS and DOUBLE_SIZE. For a CHANGE_PARAMS type of rehashing, increment both a and b by two (so the new a becomes the old b—this will save some relocations, see below), whereas the size s does not change. For a DOUBLE_SIZE type of rehashing, the parameters a and b do not change, but the allocation size s increases to $2s$ (so t increases by one).

In both types of rehashing, create a new hash table, make a pass through the entire *data* array of the old table, insert all elements residing in the old table to the new table (using the *insert*() function of Part 4). Each individual insertion

attempt in the new table is carried out non-recursively, that is, if MAX_ITER_CNT iterations fail to insert, return with failure status. This way you avoid *rehash*() inside *rehash*(). If all insertions are successful, rehashing is successful too, so you free the old table and return the new table. If at least one insertion fails, rehashing is unsuccessful, and you free the new table and return the old table. This guarantees that the table returned is not inconsistent.

Note: It is possible to do rehashing *in place*. Make a single pass through the entire *data* array, and relocate each string that is not in one of the two correct indices for that string. Relocating a string means deleting the string from the current position, and inserting it back to the table. A failed relocation attempt means that a potentially inconsistent table is returned. In this assignment, it suffices that you implement the copying strategy using two tables.

Part 6: Write a function *delete*(*T*, *str*) to delete a string *str* from *T*. If *T* is present in one of the two allowed positions in the *data* array, delete the string from that location. Otherwise, deletion fails, that is, *T* remains unchanged.

Note: The function *search*() should return only a Yes/No answer. The functions *init*(), *insert*(), *delete*(), and *rehash*() should return the updated table *T*. A status (success/failure) may be reported at an integer variable, a pointer to which is passed to the functions. The function *init*() does not require a status reporting. The functions *insert*() and *rehash*() report a status to inform the caller whether everything went fine. For *delete*(), a status may be reported to indicate whether any change was made in *T*. The restriction imposed by MAX_REC_LEVEL can be handled by passing a recursion level as a parameter to *insert*(). Whenever a recursive call of *insert*() is made, the level should increase by one. No recursive call should be made if the level of recursion has reached MAX_REC_LEVEL. The detailed printing as demonstrated in the sample output can be done within the called function or in the calling function (like *main*()), as per your convenience.

Part 7: Write a *main*() function to do the following.

- Read a file name from the user, and open the file. (The format of the file is as in the sample output.)
- Initialize *T* by calling *init*().
- Read *ninit* from the file. Then, read *ninit* strings one by one from the file (each string is stored in one line of the file, assume that the strings do not contain spaces), and insert them in *T*. Show at which index, each input string is inserted, and when rehashing is made.
- Read *nsearch* from the file. Then, read *nsearch* strings from the file. Report the results of searching these strings in *T*.
- Read *ndel* from the file. Then, read *ndel* strings from the file. Delete these strings one by one from *T*. For each deletion, report from which index the string is deleted. If the string does not exist in *T*, report failure.

Submit a single C/C++ file solving all the parts.

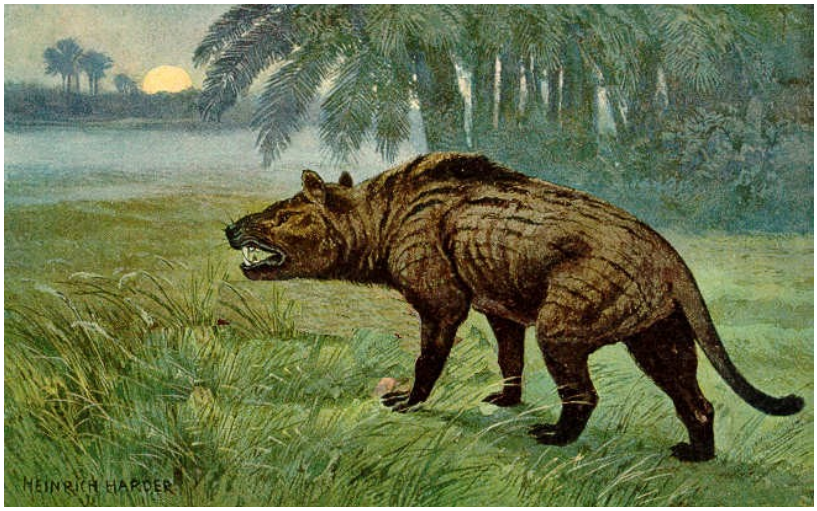
Sample Output

Let us maintain a hash table of prehistoric animals. The following output is a bit verbose, but clearly depicts the working of the algorithm. Against each string to be inserted, deleted, or searched, the two hash values are shown, like (4,6) against Hyaenodon. All insertion and deletion indices are shown. Rehashing requires reinserting, for which the indices are specified too.

INPUT FILE

```
25
Hyaenodon
Ambulocetus
Deinotherium
Mastodon
Hyaenodon
Scutosaurus
Megalodon
Entelodon
Gastornis
Hallucigenia
Nyctosaurus
Archaeopteryx
Propalaeotherium
Opabinia
Helicoprion
Basilosaurus
Dimorphodon
Leptictidium
Dinofelis
Megatherium
Homotherium
Ancylotherium
Indricothere
Microraptor
Smilodon
3
Megatherium
Abhijit
Indricothere
3
Ambulocetus
Scutosaurus
Arobinda
```

```
--- Insert(Hyaenodon): (4,6): insert at index 4: success
--- Insert(Ambulocetus): (1,4): insert at index 1: success
--- Insert(Deinotherium): (6,4): insert at index 6: success
--- Insert(Mastodon): (3,2): insert at index 3: success
--- Insert(Hyaenodon): (4,6): already present
--- Insert(Scutosaurus): (1,5): insert at index 5: success

+++ High load factor (n = 5, s = 8). Rehashing necessary...

+++ Rehashing with doubled size 8
    Entry   1: Relocating (Ambulocetus): --- Insert(Ambulocetus): (3,9): insert at index 3: success
    Entry   3: Relocating (Mastodon): --- Insert(Mastodon): (6,4): insert at index 6: success
    Entry   4: Relocating (Hyaenodon): --- Insert(Hyaenodon): (9,12): insert at index 9: success
    Entry   5: Relocating (Scutosaurus): --- Insert(Scutosaurus): (3,11): insert at index 11: success
    Entry   6: Relocating (Deinotherium): --- Insert(Deinotherium): (13,9): insert at index 13: success
    Rehashing successful...

--- Insert(Megalodon): (7,6): insert at index 7: success
--- Insert(Entelodon): (9,11): insert at index 12: success
--- Insert(Gastornis): (6,4): insert at index 4: success
--- Insert(Hallucigenia): (12,6): temporary failure

+++ Rehashing with changed parameters (3,5)
    Entry   3: Relocating (Ambulocetus): --- Insert(Ambulocetus): (9,0): insert at index 9: success
    Entry   4: Relocating (Gastornis): --- Insert(Gastornis): (4,1): insert at index 4: success
    Entry   6: Relocating (Mastodon): --- Insert(Mastodon): (4,1): insert at index 1: success
    Entry   7: Relocating (Megalodon): --- Insert(Megalodon): (6,5): insert at index 6: success
    Entry   9: Relocating (Hyaenodon): --- Insert(Hyaenodon): (12,15): insert at index 12: success
    Entry  11: Relocating (Scutosaurus): --- Insert(Scutosaurus): (11,3): insert at index 11: success
    Entry  12: Relocating (Hallucigenia): --- Insert(Hallucigenia): (6,0): insert at index 0: success
    Entry  13: Relocating (Deinotherium): --- Insert(Deinotherium): (9,5): insert at index 5: success
    Rehashing successful...

--- Insert(Entelodon): (11,14): insert at index 14: success

+++ High load factor (n = 9, s = 16). Rehashing necessary...

+++ Rehashing with doubled size 16
    Entry   0: Relocating (Hallucigenia): --- Insert(Hallucigenia): (13,1): insert at index 13: success

. . .


--- Insert(Dinofelis): (30,50): insert at index 30: success
--- Insert(Megatherium): (57,31): insert at index 57: success
--- Insert(Homotherium): (56,52): insert at index 56: success
--- Insert(Ancylotherium): (40,46): insert at index 40: success
--- Insert(Indricothere): (34,36): insert at index 36: success
--- Insert(Microraptor): (18,8): insert at index 18: success
--- Insert(Smilodon): (38,21): insert at index 21: success

+++ Search(Megatherium): (57,31): SUCCESS
+++ Search(Abhijit): (14,45): FAILURE
+++ Search(Indricothere): (34,36): SUCCESS

+++ Delete(Ambulocetus): (39,2): deletion at index 39: SUCCESS
+++ Delete(Scutosaurus): (47,14): deletion at index 47: SUCCESS
+++ Delete(Arobinda): (39,2): FAILURE
```



A Hyaenodon, illustration by Heinrich Harder
Source: http://www.copyrightexpired.com/Heinrich_Harder/