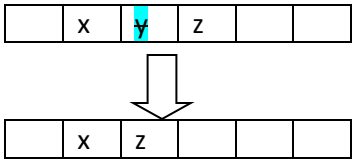
Birla Institute of Technology & Science, Pilani
2nd Semester 2016-17 - CS F211 – Data Structures and Algorithms

Lab 8 – 16th March
Topics – Hashtables with Open Addressing

Objective

1. Build an open-addressed hashtable (with operations *find*, *add*, and *delete*) and test it with different probing schemes: (i) linear probing (ii) quadratic probing, and (iii) double hashing. In particular, for each probing scheme:
   (a) count the number of probes for each *find* (and average it over a number of *find* operations) when the mix of operations includes only *find* and *add* operations;
   (b) Repeat step (a) for mixes with different proportions of find and *add* operations (i.e. number of *find* operations varying from, say, 10% to, say, 90%).
   (c) Repeat step (b) for different load factors (from, say, 0.25 to, say, 0.99).
   (d) For high load factors (from, say, 0.6 to, say, 0.99) measure the number of clusters, the average size of clusters, and the maximum size of clusters; [Definition: A cluster is sequence of non-empty cells or buckets. End of Definition.]
2. Repeat steps 1.(b) and 1.(d) for the same hashtable after including *delete* operations in the mix:
   a. Consider suitable proportions of the three operations while repeating (b);
   b. Assume that *delete* merely marks the cell *deleted* and does not recover space but add *may* reuse the deleted space to insert an item.
3. Implement a *sweep* operation on a hashtable which scans the table and "pulls up" items in a chain following a cell marked *delete.* A sweep operation is typically used to minimize the chains formed due to probing. Consider a chain of three elements x, y and z, all of them gave the same output from a given hash function. So they must have formed a chain in the hash table resultant of probing (say linear probing) as shown below. If y gets lazy deleted and marked as delete, the probing chain is not contiguous anymore. Sweep operation essentially pulls up z into the location of y making the chain of probing contiguous.



Implementation

Implement a hash table with open addressing with a "word (string)" as key. The choice of probing scheme (linear probing, quadratic probing, and double hashing) will be given as user input. Size (number of buckets) will be derived as 2^t for a user specified value of t. Each entry of the hash table should be a character pointer, which will either point to NULL or to base address of the string, stored in an array of strings, which have been assigned to this index. Each entry of hash table may have additional fields other than this character pointer.

Hashing Function: `int hash(str, t)`
Required to hash a string *str* to an index in the range $[0, 2^t - 1]$.

/**** pseudocode for the hash function ******

1. First, the string str is converted to a 32-bit unsigned integer m using the following steps
    a. Let l be the length of str.
    b. $m_{-1} = 0$
    c. for $i = 0, 1 \ldots . l - 1$, compute $m_i = (Am_{i-1} + str[i])\ mod\ 2^{32}$, where $A = 2^{16} + 2^8 - 1 = 65791$ and $str[i]$ stands for ASCII value of the character.
2. $m = m_{l-1}$
3. Return most significant t bits of $(m * 7) mod\ 2^{32}$

/****end of hash function pseudocode********

Note: In case of double hashing, for second hash function, modify step 3 of above hash function to return ***least*** significant t bits of $(m * c) mod\ 2^{32}$. **End of Note.**

Structure of data type to store hash table

The data type for hash table should contain the following fields.

- `t` : user given parameter
- `size` : size of the table (size = 2^t)
- `entries`: number of elements in the hash table
- `loadFactor`: a double variable storing the ratio = `entries`/`size`. Should be updated after each add and delete call.
- `findCount`: number of times find function have been called (do not include the calls made from inside add and delete functions)
- `findProbes`: number of times probing have been done (do not include the probing done during call to add and delete functions)
- `findRatio`: a double variable storing the ratio = `findProbes`/`findCount`. Initial value = 0.0

Cluster Calculation

A cluster is sequence of non-empty cells or buckets. Given a hash table, some cluster information can be calculated as follows:

- `clusterCount`: number of clusters in the hash table.
- `clusterAvgSize`: average size of clusters. It can be calculated by dividing the sum of length of each cluster by clusterCount.
- `clusterMaxSize`: maximum size of clusters

Input format

Each line will start with a one of the following key values (0, 1, 2, 3, 4, 5, 6, 7, or -1). Each key corresponds to a function and has a pattern. Implement following function according to given pattern and write a driver function which can call the functions according to given key value.

Key	Function to call	Input Format	Description
1	*readData*	1 N str_1 str_2 ….. str_N	Read next N lines having a single word of at most 20 characters and store it in an array of strings, called as **`data`**. Allocate memory dynamically for this array.
2	*createHashtable*	2 K t	2 shows creation of a hash table with open addressing. K represents probing scheme • 0 → linear probing • 1 → quadratic probing

			• 2 → double hashing "t" will be a parameter for hashing function and size of hash table will be 2^t . It should only create an empty hash table. Each entry of the hash table should be a character pointer, which will either point to NULL or to base address of the string in **`data`** array, which have been assigned to this index. Each entry of hash table may have additional fields other than this character pointer.
3	*find*	3 i	Search for i^{th} element of **`data`** array in the current hash table. Counting starts from 0. It should return a pair <number of probes, character pointer>. Returning character pointer should either contain NULL or address of cell containing the value.
4	*add*	4 i	Insert i^{th} element of **`data`** array in the current hash table. There will be a single hash table in a program. Counting starts from 0. It should call *find* function to know if the value already exists.
5	*delete*	5 i	Delete i^{th} element of **`data`** array from the current hash table by making the character pointer point to NULL. Counting starts from 0. It should call *find* function to know if the value already exists.
6	*lazyDelete*	6 i	To delete i^{th} element of **`data`** array from the current hash table, do not destroy the character pointer, instead, mark the cell as deleted. If a subsequent probe from *find* reaches here, it should be treated as if the value has NOT been deleted (but it should fail, if the value being searched is the deleted value i.e. if probing stops here). But, if a probe from *add* reaches here then the character pointer should point to base address of new string in data array i.e. new value should get added here.
7	*experiment*	7 K 4 a_1 4 a_2 3 a_3 … 3 a_K	Next K lines, will contain calls to mix of *add*, *find*, *delete*, and *lazyDelete* functions (one function call in each line). After processing each line, print the value of `loadFactor` and `findRatio` in a new line (tab separated). After processing all K lines, calculate clusters and print clusterCount, clusterAvgSize and clusterMaxSize in a new line (tab separated).
8	*clear*	8	Clear the hash table (like it was at the end of call to createHashtable) and restore member variables to their default values.
9	*sweep*	9	Apply sweep operation on the hash table.
-1			stop the program.

Test Case 1:

Input	**Output**
1 10 kuuxfit omqjn xqsdpag uekd zfnv hqnmzh nukxhjv byncerx	

balbg xwgojtn 2 0 4 7 12 4 1 4 5 4 9 4 0 3 2 4 5 4 3 4 7 5 1 6 8 4 6 3 6 -1	