HASH TABLES

Chapter 11 but not section 11.5

DISCLAIMER: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2019; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Introduction

Some applications require the implementation of a **dictionary data structure** with operations **Insert, Search and Delete**. If one implements a dictionary using a linked list or an unordered array, Search would require $\Theta(n)$ time. For an **ordered (i.e. sorted)** table, **binary search** can speedup Search to logarithmic time; yet the other two operations can still take linear time. The introduction of **hash tables** allows faster dictionary operations ON THE AVERAGE. Though the worst case time for such operations are as bad as that of linked lists and arrays, under some reasonable assumptions hash tables implement operations such as Search in **expected constant time** $O(1)$. Hash tables, because of their simplicity have many practical applications. For example, a hash table is used in compilers to implement a keyword look-up table and constitutes an integral part of languages that deal with manipulation of string structures (e.g. Perl).

Case 1. In **direct addressing of an array** we allocate an array of size equal **to the number of possible key values** and store a key in the corresponding table slot in $O(1)$ time. x below is a record/key and $key(x)$ contains/retrieves the numeric-key value of x. Let $U = \{0, \ldots u-1\}$ be a universe of key values. We allocate a table $T[0..u-1]$ and allocate key k to the k-th entry of the table.

```
DAddress-Search (T, k)  DAddress-Insert(T,x)  DAdress-Delete(T,x)
 return(T(k))            T[key(x)]=x           T[key(x)]=NIL;
```

The drawback of this scheme is that if x is not a number but a URL of length 50 (string), the value of u can grow too large to $2^{6 \cdot 50} = 2^{300}$. (This assumes a URL can only contain 26 upper case English character, 26 lower case English characters, 10 digits, and may be one slash and a colon for a total of 64 characters which is 2^6.) Even if we can afford that much space of 1 Googol (10^{100}), the number of keys actually stored might be less than u. Think for example how many 10-letter words or character combinations we have and how many of them are used in the average program.

Case 2. If allocation of a table of size equal to $m = u$ is not possible, direct addressing is not an option as it wastes space. An alternative to direct addressing is **hashing** using a **hash table**.

What is a hash table? A hash table is a generalization of an array.

A hash table allocates space **proportional to the (maximum) number of keys that we plan to use/store in the hash table** NOT TO THE TOTAL NUMBER OF POSSIBLE KEY VALUES. So there are $\geq 2^{5 \cdot 10}$ 10-letter strings (52 upper and lower case letters). yet few programs processed by a compiler use more than a million variable names.

(c) Copyright A. Gerbessiotis. All rights reserved.

How many bits do we need to represent 4? 3 bits (100)
How many bits do we need to represent 7? 3 bits (111)
How many bits do we need to represent n? Answer: floor(lgn)+1 =ceiling(lg(n+1)).
How do we map binary number 1011 into a decimal number?

```
   3  2  1  0          n bits represent              n
  2  2  2  2         non negative numbers     0 .. 2 -1.
  x  x  x  x
+ 1  0  1  1         lgn bits represent             lgn
  ==========         non negative numbers     0 .. 2  -1 = 0 .. n-1
  8+ 0 +2 +1 = 11
```

How do we map string "ABD" into a number/key for direct addressing?

```
Devise an "alphabet" correspondence  for capital-case letters only
 empty/space -> 0,  A -> 1, B->2,   ...   Z-> 26

    2    1    0           2
  27   27   27          27  = 27*27 = 729
   x    x    x
+  A    B    D
   1    2    4
 -------------
  27*27*1+27*2+4 = 729+27*2 + 4 = 787
```

In direct addressing, key x is 1-1 associated with k and k is stored in slot k of the table, i.e. in $T[k]$. In hashing instead we first try to map x into a number value $k = key(x)$ (and thus ABD into 787). In general however such a function need not be 1-1. Such a `key` function can be an errorcheck such as CRC-32 or a "hash or fingerprint" function such as SHA-1 or MD5. An arbitrary string of 50 or so characters is then down converted into a smaller footprint of 4 or 20 bytes: `a fingerprint`.

(c) Copyright A. Gerbessiotis. All rights reserved.

Collisions then become unavoidable: that is there could be two keys $x_1 \neq x_2$ whose CRC-32 or SHA-1 or MD5 is such that $key(x_1) = key(x_2)$ even if $x_1 \neq x_2$. The use of `k=key(x)` is still uneconomical to effectively index a hash table, of much smaller size. We need to make sure that k is no more than the number of slots of the hash table. Thus to somehow utilize k to index a hash table we apply a `hash function` **h** that maps $k = key(x)$ into the range of indexes of a hash table, as determined by its number of slots.

In hashing, starting with key x we first turn the arbitrary x into a numeric value $k = key(x)$, and then compute the hash index of k using a hash function h. The hash index of k is then $h(k)$. Then we access slot $h(k)$ of the (hash) table T to store k (or x) The range of $h(k)$ is the range of the indexes of the slots of T.

Let $U = \{0, \ldots u-1\}$ be a universe of key values. Instead of allocating a hash table $T[0..u-1]$ of size u we allocate a hash table of size m, where m is an estimate of the (maximum) number of keys that will be stored in the hash table. Since in general $m \ll u$, there may be two keys k_1 and k_2 that hash into the same slot, i.e. it may be possible (and likely) that $h(k_1) = h(k_2)$. This occurrence is called a **key collision**. As $u > m$, it is unrealistic not to expect collisions.

A minimization of collisions is a reasonable objective in designing a hash table and choosing a hash function. Key collision resolution is, however, essential in dealing with hashing.

```
In hashing starting from  string "ABD" we first get its ordinal number 787, and then use the hash function to hash 787 into a slot of the hash table
  x    mapping   k=key(x)       h           h(k)
"ABD" ----------> 787   ----------------->   5   assuming h(787)=5
```

For the remainder we shall assume that all keys are not in the x form but in the k form (i.e. they are represented as decimal numbers).

1. **The number of keys to be hashed is denoted by n.**
2. **The size (number of slots) of the hash table is denoted by m, and the slots are then $0, \ldots, m-1$.**
3. **Load factor is the ratio of n/m.**

(c) Copyright A. Gerbessiotis. All rights reserved.

Collision resolution through chaining

One scheme in resolving collisions is by **chaining**. In **chaining**, slot j stores a pointer to a linked list containing all keys k with hashed value $h(k) = j$.

The operations Insert, Search, Delete looks as follows. **Note that many time we ignore the distinction between a record x whose key is $k = key(x)$ and thus use as argument k the key directly, instead of x.**

```
ChainHashInsert(T,x)
   Insert record x at
   head of T(h(key(x)))
```

```
ChainHashSearch(T,k)
   Searchfor k in T(h(k))
   if x found with key(x)=k
     return(record x)
   else
     return(FAILED)
```

```
ChainHashDelete(T,x)
   Delete record x from
          T(h(key(x)))
```

A **Hash-Insert** operation inserts an element in the head of the list $T[h(k)]$ provided that the key is not already in the list. A search operation can be successful or unsuccessful and is reduced into a linked-list search. Whether the linked list is a singly or a doubly linked list affects performance of ChainHashDelete. We shall assume that the linked list is a DOUBLY LINKED LIST. The worst-case running time bound for an insertion operation is $O(1)$ as keys are inserted at the head of the list. Deletion takes $O(1)$ worst case time for a doubly linked list.

As far as the search operation is concerned we shall prove that if n elements are stored in a hash table of size m and the collision resolution protocol is chaining, then a successful/unsuccessful search takes on the average time $\Theta(1 + a)$, where $a = n/m$ is the load factor under the **SIMPLE UNIFORM HASHING ASSUMPTION** which states that any key k is equally likely to hash into any of the m slots.

Let us assume that the hash table T contains m entries and it currently stores n keys. This means that the average number of keys per linked list (also called CHAIN) is $a = n/m$. a can be less than, equal to or greater than 1. The cost of computing $h(k)$ from k will be assumed to be $O(1)$. The worst case behavior of chaining is $\Theta(n)$ as all keys may get hashed on the same value. We are however interested in examining the average performance of hashing. The best we can hope for is $\Omega(1 + a)$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Theorem. With collision resolution by chaining under simple uniform hashing, an **unsuccessful** search takes time $\Theta(1+a)$ on the average.

Proof. Any key k NOT IN THE HASH TABLE is equally likely to hash into any one of m slots. There are n elements and m slots i.e on the average n/m elements per slot chain. The result follows.

In a successful search the probability a chain is searched is proportional to its size. On the average half the chain will be searched before the key is found.

Theorem. With collision resolution by chaining under simple uniform hashing, a **successful** search takes time $\Theta(1+a)$ on the average.

Proof. When key k is inserted, its record x is inserted at the head of the list. Subsequent (in time) insertions of keys in the same list occur before x. Prior insertions of keys occurred after x, i.e. traversing the chain from head to just before x, we access keys that were inserted into the chain after key k (whose inserted record was x). Suppose k was the ith key and thus x was the i-th record inserted in the hash table. After k, $n-i$ keys were inserted in the table. As each one of these keys was equally likely to go into any of the m slots, on the average $(n-i)/m$ records will be inserted before x and all these records will be searched before x is located during a successful search, for a total number of records search of $((n-i)/m)+1$. Therefore for the i-th record x with key k of the hash table $((n-i)/m)+1$ records will be searched before it is found. We sum the number of records searched for every $i=1,\ldots,n$ and dividing the sum by n we get the average successful search time.

$$\frac{1}{n}\sum_{i=1}^{n}(1+(n-i)/m) = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) = 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) = 1+\frac{(n-1)}{2m} = \Theta(1+a).$$

Question Why $\Theta(1+a)$ and not $\Theta(a)$?

What makes a good hash function

A good hash function is one that satisfies the assumption of simple uniform hashing: each key is equally likely to hash into any of m slots. Let $P(k)$ be the probability that k is drawn from U. Then we would like

$$\sum_{k:h(k)=j} P(k) = 1/m, \ \ \forall\, j$$

Unfortunately it is quite difficult to verify this condition as we don't know P.

Example Let k be a random real number drawn independently and uniformly from the real number interval $[0, 1]$. Then, hash function $h(k) = \lfloor km \rfloor$ satisfies above condition.

In practice a heuristic approach is followed. We guess a hash function and try to measure its performance on expected data. For example in a C++ compiler it makes sense that no collision results when i, j, k, l are mapped under a chosen hash function or when words contain the string **class** or **pt** (pointer) or similar commonly used patterns in variable/function definition/declaration. This way we may also like keys that are **close** to each other such as i, j or **ipt, jpt** to be hashed far away.

Finally the hashed keys are more often strings than integers. Different mappings of strings to characters may be devised for this (eg ASCII based translations).

(c) Copyright A. Gerbessiotis. All rights reserved.

The Division Method

$$h(k) = k \bmod m$$

This hash function requires a single division. It can be even faster if m is a power of 2 ie $m = 2^l$.

What m's NOT to choose

Never choose m to be a power of 2! If $m = 2^l$ then $h(k)$ is the lower l bits of k which may not be random at all. Unless we know that this is so (ie that they are random) THEN AND ONLY THEN is such a choice of m acceptable.

If k is a decimal, avoid m to be a power of 10 as well.

What m's To choose

Choose m to be a prime but not very close to a power of 2.

The Multiplication Method

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

Let A be a constant $0 < A < 1$. We first multiply k and A and take the fractional part of the product (on the right of the .) which is a number between 0 and 1 and then make this number to be an integer between 0 and $m - 1$ as earlier.

In such a case m can be a power of 2, typical choice is $2^p + 2$ for some prime p. It seems that an optimum choice of A is $A = (\sqrt{5} - 1)/2 = 0.618033$.

(c) Copyright A. Gerbessiotis. All rights reserved.

In **open addressing** we address the case that $n \leq m$ and we store the keys in the hash table itself. A slot contains either a key or a flag that indicates that the slot is available. The Load factor a is thus at most 1 for open addressing.

```
  0  1  2  3  4  5  6  7  8  9 10
---------------------------------   h'(k)=k%11
|  |  |  |  |  |  |  |  |  |  |  |   h(k,i)= (h'(k) + i) % 11
|  |  |  |  |  | 5| 6|16|  |  |  |
---------------------------------   Figure 1.
```

In order to insert k we **repeatedly probe** the hash table until we can find an empty slot in which to put the key. Instead of probing slots in a predetermined order $0, 1, \ldots, m-1$, the **probing sequence** depends on k. We thus extend hash function $h(k)$, to include a probe index/number starting from 0

$$h : U \times \{0, \ldots m-1\} \rightarrow \{0, \ldots m-1\}$$

thus defining for every key k a probe sequence,

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

that is required to be a permutation of $0, \ldots, m-1$, or there may be some slots that may be missed during the probing.

Thus for key k, we first probe $h(k,0)$, then if necessary $h(k,1)$, and so on. If $h(k,m-1)$ is also full, then we determine that the hash table is full and this causes an overflow.

Example. Let us assume that we have the very simple case depicted in Figure 1. The first key that needs to be hashed has value $k = 5$. Since $h'(5) = 5$, for $i = 0$ we have $h(k,0) = 5$ and we probe slot 5. It is empty (no key) present. Thus we insert 5 in slot 5. Next comes key 6. Similarly it is inserted in slot 6. Next come key 16. As $h'(16) = 5$, we first try $h(k,0) = 5$. Slot 5 is occupied by 5 and thus a **collision** occurs; the collision is due to the fact that 5,6 are such that $h'(5) = h'(16)$. We then try $h(k,1) = 6$ which is also occupied and a collision also occurs. In this case the cause of the collision is not that $h'(16)$ is equal to $h'(6)$: they are not the same. The reason is that $h(6,0) = h(16,1)$. The next possible slot is $h(16,2) = 7$, which is available and 16 is inserted there.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

```
  Insert(T,k)
1. i=0;
2. do  j=h(k,i);
3.   if EMPTY(T[j]){   //  EMPTY(T(j)) :  (T(j)==NULL OR T(j)==DELETED)
4.       T[j]=k
5.       return j
     }
6.   else i=i+1;
7. while (i<m);
8. error("Hash overflow");

  Search(T,k)
1. i=0;
2. do j=h(k,i);
3.    if T[j]==k
4.        return j
5.    i=i+1;
6. while ((i<m) && (T[j] != NULL));
7. return (-1) // Not found!
```

In a deletion a deleted slot for key x cannot be marked NULL. If it is marked NULL then, a subsequent search for key y that was inserted after x and with $key(x) = key(y)$ (i.e. the slot for x was searched and found full during the Insert operation for y) will not be found, because the NULL in x's slot will disrupt the search. To avoid such a problem a freed slot is marked DELETED.

In the analysis of open addressing we shall make the assumption of **uniform hashing that is, each key is equally likely to have any of the $m!$ permutations as its probing sequence**.

Linear Probing

Given an ordinary hash function $h'(.)$, linear probing uses h such that $h(k,i) = (h'(k) + i) \bmod m$. The first position is $h'(k)$. The problem with this method is **primary clustering**. Long runs of occupied slots are being built.

Say we have $m/2$ keys in T. In case 1, these keys occupy the first $m/2$ position of T (clustering). In case 2, they occupy the odd positions of the array. In case 1 average probing is $m/8$, in case 2 it is 1.5.

Quadratic Probing Given an ordinary hash function $h'(.)$, quadratic probing uses $h(k,i) = (h'(k)+c_0+c_1 i+c_2 i^2) \bmod m$. The first position is $h(k,0) = h'(k)$. If no c_0, c_1 and c_3 are specified (e.g. in homeworks or exams), then $c_1 = c_2 = 0$ and $c_3 = 1$. A milder form of clustering appears under quadratic probing which is called **secondary clustering**.

Double Hashing Given two hash function h_1, h_2, double hashing uses h such that $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$. The probe sequence depends on two hash functions i.e. depends twice on k. In addition $h_2(k)$ must be relatively prime to m for the entire hash table to be searched. If their gcd is d instead only a $1/d$ fraction of the hash table is searched. This can be enforced by making m a power of 2 and force $h_2(k)$ to produce an odd number.

What do we NEVER store inside the Hash table? In all the previous examples it made sense to store in the hash table the "key" itself. Since k was numeric, a fixed number of bits (most likely a 32-bit or 64-bit integer) makes sense to be stored directly into the table. If x however needs to be stored as well, and x is a variable length string, we have a problem: x is not of fixed length. If one stores x into the table one must accommodate the longest (i.e. widest) string in the table i.e. each slot must be wide enought to accommodate the k the the longest x. A better approach is NOT to store x in the table; we store there k and a pointer or index or reference p that points to some other memory area that stores x itself. However storing strings in memory needs to be done economically. One way is to store all strings x of the table one after the other (concatenation of strings). To determine the end of a string then we need either a terminating pointer (eg a NULL value) or a length value stored at the first address pointed by the pointer and prior to x, or just a special character separating strings (equivalent to a NULL value or `'\0'`).

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Analysis of unsuccessful Search: Theorem Given an open addressing hash table with load factor a the expected number of probes in an unsuccessful search is $1/(1-a)$ assuming uniform hashing.

Proof. The time for an unsuccessful search in a hash table of size m with n keys in it is the same as the time for inserting an $n+1$-st key in the hash table. Insert(n,m) gives the AVERAGE number of probes required for inserting a key in a table of m slots ALREADY containing n keys. Therefore

$$UnSearch(n,m) = Insert(n,m)$$

It is obvious that $Insert(0,m) = 1$, for any m. $Insert(n,m) = 1$ if the $n+1$-st key hashes into an empty slot. The probability of this happening is $(m-n)/m$. Otherwise, with probability n/m the key hashes to an occupied slot, so the number of probes is 1 plus the number of probes required to be inserted in the rest of the table. Therefore

$$Insert(n,m) = 1 \cdot \frac{m-n}{m} + \frac{n}{m}\left(1 + Insert(n-1,m-1)\right) = 1 + \frac{n}{m} Insert(n-1,m-1).$$

By induction on n and for $n < m$ we can prove that

$$Insert(n,m) = \frac{m+1}{m+1-n} \approx \frac{1}{1-a}$$

Analysis of Insert: Theorem Given an open addressing hash table with load factor a the expected number of probes for an insert is $1/(1-a)$ assuming uniform hashing.

Proof An element is inserted if there is space and it is preceded by an unsuccessful search and placement of key in the first empty slot found, and the time of an unsuccessful search is thus $1/(1-a)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Performance Analysis continued

Analysis of successful Search: Theorem Given an open addressing hash table with load factor a, the expected number of probes in a successful search is $1/a \ln(1/(1-a))$, assuming uniform hashing.

Proof A search for k has the same probing sequence as when k was inserted. When k was inserted, let it have been the $i+1$-st inserted key. Then insertion required on the average $1/(1-i/m) = m/(m-i)$ probes. Averaging over all keys we get (note $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n \approx \ln(n)$).

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{m}{n}(H_m - H_{m-n}) = \frac{1}{a}(H_m - H_{m-n}) = \frac{1}{a}(\ln(m) - \ln(m-n)) = \frac{1}{a}\ln(1/(1-a))$$

and the result follows. If $n/m = 0.5$ expected number is 1.387. If $n/m = 0.9$ then 2.55.

(c) Copyright A. Gerbessiotis.

In real-life the open-addressing scheme is used more often than hashing by chaining. Instead of using the simple hash functions related to linear probing or quadratic probing, more elaborate hash functions are used. The hashing scheme (e.g. table organization) is also more involved. In any such scheme rehashing and reorganization might be necessary if there is an overflow condition. In such a case the size of the hash table gets increased (say by a factor of two or more), and the keys are deleted one by one from the previous (overflowed) hash table to be inserted into the new. Such a reorganization is slow and time consuming and should thus be avoided or done as infrequently as possible. Sometimes this reorganization can be avoided by the following method: when an overflow condition occurs in hashing by open-addressing in which the size m of the table was carefully chosen, any new keys are inserted into a balanced binary search tree data structure. This way one avoids reorganization by paying a penalty of using more space per key (pointers) and more time for search/insertion/deletion requests for the newly inserted keys.

In addition, keys are not plain numbers of fixed size (e.g. 32-bit integers) but arbitrarily long strings. Thus it makes then sense to store in the table key values that are pointers to memory in which the string itself resides. However pointers might be a very crude representation of a key value itself (and not very arbitrarily chosen). In such cases associated with a string s we might have not only a pointer p in memory that points to s itself, but also a key value k that can be some kind of a hash of s itself. (Thus p and k are different from each other.) A checksum function (such as a 32-bit CRC, or 128-bit MD5, or variable length SHA functions can serve as such) can then be used to generate k from s. A `docID` identifies a document better than a variable and usually long sting that a URL is. A `docID` can be obtained through a hash function such as MD5 but usually is user assigned (eg popular documents have short `docID`s.)

Google's. In Google (original 1997 realization) one of the tables used stores URLs and docIDs. One table used docIDs as the key and stores for each document with a given docID a variety of data associated with it such as the status of the document, a pointer to the Google repository holding the document (the "cached" copy of the Web document), the document checksum established by Google's internal checksum function (which can be one of the previously listed functions), statistics about the document, the URL itself and its title (if any).

In order to access this table, one needs to previously access a more compact table that contains for each document only a URL checksum and the docID for the document. This latter table is sorted based on checksum. Thus binary search on the checksum can retrieve the corresponding docID. Thus depending on the problem requirements, hashing can be avoided by using a binary-search method. This of course is possible if the table is already sorted; the application in question maintains that secondary table sorted by merging a (sorted) new instance into the previous (also sorted) instance of the table.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Case study: Three organizations for `word` or `wordIDs`

The average length of a word in a dictionary is roughly 7 characters (English). Yet the average length of a word in a text is roughly 4 characters. Why? Because texts contain repetitions of words and the words that are usually repeated a lot are short ones such as articles, prepositions, pronouns, conjunctions.

Case T1. Suppose we would like to maintain a table of n words using a hash table. Collision resolution by open addressing will be used. For efficiency reasons we use $m = 2n$, i.e. the table should be at least twice as large as the number of words that will be maintained in it. We choose to store x itself i.e. `word` inside the hash table. Let the maximum length of a word to be maintained be 32 characters, yet the average length of a word be 7. If ASCII encoding gets used the number of charactes is also the number of bytes used. Total space of the hash table is $m \cdot 32B = 2n \cdot 32B = 64nB$. (If one has a PC with 512MiB of space, a large capacity for 1997 configurations, then one could barely accommodate $n = 512/64 = 8$M words with such a scheme.)

Case T2a. Scheme T2a is a minor variation of **T1**; it maintains inside the hash table not the string itself, but only some-form-of-a-pointer to strings that are (represent) `word`s The bitlength of a `pointer` can be 32-bits(4B). However if an `index` to a table is used instead, we may only use 24bits, or 16bits of a "short" pointer. The words themselves are stored in an array A of characters where words are separated by NULL characters i.e. $\backslash 0$. For hash table T this scheme used $4m = 4 \cdot 2n = 8nB$ if a pointer is 4B. The length of A is $n \cdot 8 = 8n$. This is because A maintains n words of average length 7 (see Case T1 assumptions). Adding the NULL at the end, this becomes an 8. The total space used for T and A is thus $8n + 8n = 16nB$, a fraction (one quarter) of the space used previously. Four times as many words can be accommodated. **Google's Lexicon** in 1998 bears similarity to this organization. A number of $n \approx 14,000,000$ words were maintained in the Google lexicon of 1997 that utilized a PC with 256-384MiB of RAM.

Case T2b. In this scheme we maintain a `wordID` in addition to the pointer. If we have n words we need roughly $\lg n + 1$ bits to represent a `wordID` in the range 0 to $n - 1$. Thus for $m = 2n$ we need $2n(\lg n + 1)$ bits for `wordID`s. Definitely for $n < 2^{32}$ we would need no more than 4 Bytes per wordID. Thus `wordID`s can consume $8n$ more bytes. If the previous scheme used $16n$ bytes of space this one uses $24n$, still better than $64n$!

(c) Copyright A. Gerbessiotis.

Size of a lexicon

One application of a hash table is to maintain a vocabulary (or lexicon). The lexicon (vocabulary) may be implemented by a more elaborate version of an open-addressing scheme. The hash table itself (similar to the table used for open-addressing) might be a hash table of pointers. The strings themselves are stored into a separate contiguous memory separated by NULL (`'\0'`) characters. For efficiency purposes vocabularies used in say Web-searching are memory resident.

Problem. Say we have a PC with 256 MiB (megabytes binary) of RAM. We want to build a vocabulary of English words. How many words can it accommodate? How will it be organized?

Answer. The amount of main memory available is a hint that a memory resident lexicon shall be used. We could use a hash table of only pointers and store the strings separately NULL separated. Say the number of words allowed by this organization is n. The average length of an English word (before stopwords and stemming are applied) is around 6 to 7. Including the extra NULL character that will be used as a string/word terminator, we get a rough size of $8n$ bytes for simply storing into consecutive memory locations the strings. This way an array of size $8n$ will be used to store all the strings; we do not need to use in array of variable length strings! (This reminds us of scheme **T2a**.)

A pointer is 4 bytes, thus a hash table of pointers of length n would need $4n$ bytes. But then the table will be full and hash-table performance will be suboptimal. If the table is made $m = 2n$ long, hash table performance will improve. Such a table of pointers uses $4m = 4 \cdot 2n = 8n$ bytes. In total the amount of memory used is roughly $8n + 8n = 16n$. Thus $16n = 256MiB$ and solving for n we get that this scheme can accommodate approximately $16,000,000$ words. Of course in that memory one needs to store the operating system processes, and also the hash table related code for hash table maintenance and thus vocabulary size would be even lower.

One could store along with each pointer, a `wordID` i.e. a hash (or checksum) of the individual word (similarly to a docID). (This become **T2b** then.) Given that the previous estimation can support up to 16,000,000 words, we conclude that we do not need more than 24-bits for a wordID since $\lg 16,000,000 \approx 24$. This is roughly 3 bytes per word. Thus for a hash table of length $m = 2n$ we need 4 bytes for a pointer and 3 bytes for a wordID or a total of $2n \cdot (3 + 4) = 14n$ bytes an increase by $6n$ bytes over the previous $8n$. Adding to that $8n$ for the strings, we get $22n$. Equating this to 256MiB we conclude that this scheme can accommodate a vocabulary of slightly smaller size (10-12 million).

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.