

HASH TABLES

Chapter 6-6.3 (but not 6.1 but yes 6.1.2) of GT

DISCLAIMER: *Report an error / typo to the author so that he resolves it.*
These notes are NOT to be distributed outside of NJIT without the expressly written permission of A. Gerbessiotis

Definition 1 (Dictionary ADT). A dictionary ADT contains words (data model) and supports operations **Insert**, **Search** and **Delete**.

Implementing a dictionary means instantiating a data structure for the data model. The data structure used defined the operations and their runtime behavior and performance.

Corollary 1 (Dictionary using a Linked List). If one implements a dictionary ('s data model) using a *linked list*, operations **Search** and **Delete** would require $O(n)$ time and **Insert** might require $O(n)$ time. Thus there would be an operation that requires in the worst case $\Theta(n)$ time.

Corollary 2 (Dictionary using an Array). If one implements a dictionary using a *array*, operations **Search** and **Delete** would require $O(n)$ time and **Insert** might require $O(n)$ time. Thus there would be an operation that requires in the worst case $\Theta(n)$ time.

For an ordered (i.e. sorted) array, binary search can speedup **Search** to logarithmic time. Yet the other two operations can still take linear time. And the discussion here is time rather than just comparisons. If ones uses balanced tree structures such as a red-black tree, AVL tree, 2-3-4 trees, or B-trees, one can achieve logarithmic time for all three operations. The extra space for pointers however becomes a concern.

Definition 2 (Hash Table Informal definition). A Hash Table is an implementation of the dictionary ADT using hashing.

Definition 3 (Hash Table's worst-case performance). A Hash table's worst-case performance is that of a linked list or an unordered array.

The introduction of hash tables allows for faster dictionary operations ON THE AVERAGE. It is a bit tedious however to express what this means. Though the worst case time for dictionary operations are as bad as those of linked lists and arrays, under some reasonable assumptions hash tables implement operations like 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).

Definition 4 (Hash table's arrays T and A , slots, keys, and load factor a). A Hash table T with N slots has length N . It can accommodate up to n keys where depending on the structure of the hash table it can be $n \leq N$ or it is also possible for some hash table organizations that $n > N$. Thus it can be inferred that the key values are not stored in T but in an auxiliary array A and T and A are linked to each other. The load factor a of the hash table is the ratio $a = n/N$.

Corollary 3. In table T we NEVER store variable length keys or key values. We might store pointers (an operating system or programming language defined data type), indexes (user defined data type) or other references.

Definition 5 (x and k and $h(k)$). Let $U = \{0, \dots, u-1\}$ be a universe of key values. An $x \in U$ would denote a variable length key value. Using a function name **key** we map variable length x into a fixed length key k such that $k = \text{key}(x)$. We store x or k into a table T or A as needed. The index of the slot into T that will be used would be determined and be based on the value $h(k)$, where h is a function known generically as a “hash function” h . Therefore $0 \leq h(k) < N$.

$$\begin{array}{ccccc} \text{variable length} & & \text{fixed length} & & \text{index in } T \\ x & \xrightarrow{\text{key}} & k = \text{key}(x) & \xrightarrow{h} & h(k) \end{array}$$

Example 1. Let $U = \{0, \dots, u-1\}$ be a universe of key values. Let $x \in U$. An $x \in U$ can be a URL or a word, making U a universe of string (key) values. Function **key** can be a cryptographic fingerprint function such as CRC-32, or MD5, or SHA2. The k generated by such a function is of fixed width usually 4B or 8B or as much as 64B. When x is a URL or a word we might refer to its k as docID or wordID respectively. The h of the function that maps k into T must generate indexes (slot references) in a much smaller range $0 \dots N-1$. In $h(k)$ or at a slot reference (index) based on $h(k)$ we might store a key k or reference to an auxiliary array A that would contain x or even k itself.

A URL in Internet Explorer 8/9 can be up to 2083 characters and in Google Search we can have up to 1855 characters. According to a google search the longest english word is “pneumonoultramicroscopicsilicovolcanosomiosis”. This provides a guidance on how long a string x can be and why we need to use something like k that is more manageable (4B or 8B or at most 64B) in length.

Hash Tables

T is for fixed width keys

Definition 6. A Hash table organization or a reference to hash table T would refer not just to T but also to A . Thus it becomes a reference to (T, A) .

If we only use the first element of the pair as a reference, that is T , this is done so in the context of examples or a theoretical discussion or for economy of space. In that context the keys are of fixed length and are numeric.

Method 1 (Storing variable length key values in T is inefficient). Let T be a table of N slots intended to accommodate $n \leq N$ keys and the keys are English language words. We allocate space so that each slot can store the longest word of English consisting of 45 characters and given earlier. Let us assume that the representation for a word is in ASCII thus one character is stored in one byte. Then the size of T in byte would be.

$$|T|_{size} = n \times 45 = 45nB.$$

We know certain thing about the English language the average length w_d of a word in an English language dictionary is $w_d = 7$ even if the array length of a word in a text is $w_t = 5$. (In a text we have repetitions of short words.) Thus even if T is filled with n words, the amount of useful space utilized by those words would be on the average

$$|T|_{useful} = n \times w_d = 7nB.$$

The utilization of T is a lowly

$$\frac{|T|_{useful}}{|T|_{size}} = 7/45 \approx 0.16 = 16\%.$$

Search Tables

Direct addressing

Definition 7 (Bucket array or Direct addressing). In a bucket array or direct addressing of an array, the number of possible key values x is usually small and (T, A) can become just T ; we then store a key directly into a slot of T and be able to retrieve it in $O(1)$ time.

Example 2. Let $U = \{0, \dots, u-1\}$ be a universe of key values. Set U is of small enough length and size that can be accommodate by the amount of memory available. Thus $k = \text{key}(x) = x$. Moreover, we allocate table $T[0..u-1]$ in main memory and store key k into the slot with index k . Thus $h(k) = k$ as well.

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

Example 3. An IP-v4 address is summarily four byte values separate by three periods. Thus 128.235.32.140 is an IP-v4 address. Internally it can be represented by a 32-bit unsigned integer. Furthermore $|U|$ is 2^{32} . For $x=a.b.c.d$ where $0 \leq a, b, c, d < 256$, we define

$$k = \text{key}(x) = a \times 2^{24} + b \times 2^{16} + c \times 2^8 + d$$

and furthermore $h(k) = k$. This a direct addressing scheme for IP-v4 addresses. The length of table T to accommodate direct addressing is no more than 2^{32}B .

The drawback of this scheme is that if x is not an IP-v4 but an IP-v6 address we need 16B instead of 4B. Thus table T now must be of size 2^{128}B which is not realizable nowadays. If x is a word (string) of length 50, the cardinality of U can grow too large to say $2^{6 \cdot 50}$. (1 Googol is 10^{100} .) Even if we can afford that space of 1 Googol, the number of keys actually stored might be less $|U|$. Think for example how many 10-letter words or character combinations we have and how many of them map to legible (English language) words.

Hash Tables

A formal introduction

Thus when we deal with a hash table organization we realize the following

- The universe U of possible values of x is very large,
- The universe of $k = \text{key}(x)$ values is still very large,
- and we need to map an x or its k into an index of a table T that is not only small but also of a limited given range $0 \dots N - 1$. Moreover T resides in main memory.

When we organize keys in a hash table, collision of keys becomes unavoidable.

Definition 8 (*Collision*). A collision is the event where two different keys $x_1 \neq x_2$ get mapped to the same slot and thus $h(k_1) = h(k_2)$. For two keys $x_1 \neq x_2$ a collision occurs if $k_1 = k_2$ or $h(k_1) = h(k_2)$ even if $k_1 \neq k_2$, where $k_1 = \text{key}(x_1)$ and $k_2 = \text{key}(x_2)$. In the remainder we are going to define as collision the following.

$$k_1 \neq k_2 \Rightarrow h(k_1) = h(k_2).$$

Collisions become unavoidable no matter how hard we try to avoid them. There can be two keys $x_1 \neq x_2$ whose CRC-32 or SHA-1 or MD5 is such that $\text{key}(x_1) = \text{key}(x_2)$ even if $x_1 \neq x_2$. (This even might be rare to encounter accidentally though for MD5.) When the universe of k values is still too large, we apply a **hash function** h that maps $k = \text{key}(x)$ into the range of indexes of a hash table, as determined by its number of slots that is small and easy to accommodate in main memory.

In hashing, starting with key x we first turn the arbitrary x into a numeric value $k = \text{key}(x)$, 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 $h(k)$ or some other slot derived from it of the (hash) table T to store k (rather than x) or also a reference to x (index or pointer). The range of $h(k)$ is the range of the indexes of the slots of T . In real-life $|U| > N$ and thus collisions are unavoidable.

Question 1 (Elimination of Collisions: Collision resolution). Can we eliminate collisions?

Answer. No unless direct addressing is possible. The best we can hope for is that **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. ■

Definition 9 (Remainder). In the remainder we shall use the notation of n being the number of keys that will be maintained in a hash table organization, N would be the number of slots T utilized by such organization and a is the load-factor $a = n/N$. Moreover in the remainder we will be dealing with keys k rather than x , i.e. there exists a key function that has already been used to convert an x into a $k = key(x)$.

Definition 10 (Chaining). All keys hashing to a given slot are maintained into a linked list structure associated with that slot. Thus, if two keys $k_1 \neq k_2$ get hashed into the same slot $h(k_1) = h(k_2)$, then the keys are inserted into a (doubly) linked list associated with slot $h(k_1) = h(k_2)$.

One scheme in resolving collisions is by **separate chaining** or just **chaining** or simply **by chains**. In **separate chaining**, slot j stores a pointer (reference) to a linked list and thus Table T is an array of N doubly linked lists and $T(j)$, contains all keys k with hashed value $h(k) = j$.

The operations Insert, Search, Delete looks as follows. Note that many times we ignore the distinction between an x whose key is $k = \text{key}(x)$ and thus use as argument the key k directly, instead of x . Moreover in the linked list we might decide to store both a k and its x .

ChainHashInsert (T, x) Insert record x at head of $T(h(\text{key}(x)))$	ChainHashSearch (T, x) Searchfor x in $T(h(\text{key}(x)))$ if there is x in $T(h(\text{key}(x)))$ return(reference to x) else return(NOTFOUND)	ChainHashDelete (T, x) Delete record x from $T(h(\text{key}(x)))$
--	--	--

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.

Definition 11 (Simple Uniform Hashing Assumption). The simple uniform hashing assumption (SUHA) states that any key k is equally likely to hash into any of the N slots.

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 N and the collision resolution protocol is chaining, then a successful/unsuccessful search takes on the average time, under SUHA, $\Theta(1 + a)$, where $a = n/N$ is the load factor. Let us assume that the hash table T contains N entries and it currently stores n keys. This means that the average number of keys per linked list (also called CHAIN) is $a = n/N$. 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 $\Omega(1 + a)$.

Theorem 1. *With collision resolution by chaining under SUHA 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 N slots. There are n elements and N slots i.e on the average n/N elements per slot chain. The result follows. ■

Collision resolution by chains

Performance: SUHA

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 2. *With collision resolution by chaining under SUHA a **successful** search takes time $O(1 + a)$ 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 i th 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 N slots, on the average $(n - i)/N$ 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 $((n - i)/N) + 1$. Therefore for the i -th record x with key k of the hash table $((n - i)/N) + 1$ records will be searched before it is found. We sum the number of records searched for every $i = 1, \dots, n$ and dividing the sum by n we get the average successful search time.

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

■

Theorem 3. *With collision resolution by chaining under SUHA an Insert takes time $\Theta(1)$, and a delete $O(1 + a)$ on the average.*

Question 2. Why $\Theta(1 + a)$ and not $\Theta(a)$?

Collision resolution by chains

Worst case Performance

Theorem 4. *With collision resolution by chaining the worst-case running time for `ChainHashInsert` is $\Theta(1)$, for `ChainHashSearch` it is $O(n)$ for a successful search and $\Theta(n)$ for an unsuccessful search, and for `ChainHashDelete` it is $O(n)$.*

Consider the situation where all the keys get hashed into the same slot and let that slot be slot with index 0. Hashing operations degenerate into linked list operations!

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

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

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

Choice of a hash function Challenges

Example 4. Let k be a random real number drawn independently and uniformly from the real number interval $[0, 1]$. Then, hash function $h(k) = \lfloor kN \rfloor$ 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).

The Division Method.

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

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

What N's NOT to choose

Never choose N to be a power of 2! If $N = 2^l$ 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 N acceptable.

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

What N's To choose Choose N to be a prime but not very close to a power of 2.

The Multiplication Method

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

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 $N - 1$ as earlier. In such a case N 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 $A = (\sqrt{5} - 1)/2 = 0.618033$.

Definition 12 (Open-addressing). In open-addressing we do not use a linked list structure for (T, A) . In a slot of T we store a reference to A that stores x (or k or both) or a flag that indicates that the slot is available for insertion or not. Thus references to x are stored directly into a slot of T rather than to an adjoined linked list. Because of this in open-addressing n is at most N and thus the load factor a is $a \leq 1$.

In real applications an x (or k or both) are stored in auxiliary array A , and in T we store a reference to the location of A that stores x (or k or both) in that case. In our examples we are going to disregard x key values. Only the k values are to be used and given that they are of fixed width they will be stored in T directly and thus an A will not be used. This simplifies the exposition and the examples presented.

Just because a can be close to if not equal to one, this does not mean that optimal hash table performance can be made for a full table T .

In practice we consider that a hash table has optimal performance (in open addressing) when the load-factor is equal to 0.5, i.e. T is half-full or half-empty.

Definition 13 (Desirable load factor). For optimal (expected time) performance we would rather have a load factor (n/N) approximately equal to 0.5.

For collision resolution by open-addressing probing just slot $h(k)$ for some key value k (or x) is not enough. The hash function $h(\cdot)$ needs to be enhanced.

Collision resolution by open addressing

Probing sequence

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.

If we follow the chaining approach, after obtaining a k from x ($k = \text{key}(x)$), we attempt to insert k into the hash table T (or (T, A) in the general case). With chaining one probe operation checking $h(k)$ would have been enough. If another key k_1 got hashed into $h(k)$ the linked list attached to $h(k)$ would have accommodated both k and k_1 . This is not the case for open-addressing. If k_1 was inserted first a reference to k_1 has already been inserted into slot $h(k)$ and thus slot $h(k)$ is unavailable for k . We need to look elsewhere in T to find a place (slot) for k .

Instead of probing slots in a predetermined order $0, 1, \dots, N-1$, the **probing sequence** would depend on key k . We thus extend hash function $h(k)$ used in chaining, to include a probe index starting from 0

$$h : U \times \{0, \dots, N-1\} \rightarrow \{0, \dots, N-1\}$$

thus defining for every key k a probing sequence,

$$\langle h(k, 0), h(k, 1), \dots, h(k, N-1) \rangle$$

that is desired to be a permutation of $0, \dots, N-1$. If it is not a permutation of $0, \dots, N-1$ there might be some slots that may be missed during the probing involving k and those slots might be the only ones available to accommodate an insertion.

Having enhanced function $h(k)$ into an $h(k, i)$, whereas the former indicates only one slot $h(k)$ for the insertion of key k (and/or x), the latter one indicates a sequence of possible slots, a probing sequence $h(k, 0), h(k, 1), \dots$.

Thus for key k , we first probe $h(k, 0)$, then if necessary $h(k, 1)$, and so on. If $h(k, N-1)$ is also occupied, then we determine that we have an overflow condition, whether the hash table is full or not!

Collision resolution by open addressing

Example

Example 5. 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.

Question 3. What if a key cannot be inserted into T ? We need to reorganize the hash table. This means potentially increase N (double it), allocate space for a new T compatible with the new N , and then delete all keys from the old version of T and insert them into the new T . (Note that cut and pasting does not work. New N implies new slot indexes to store the keys already stored in the older T .) Moreover in the new T the order of insertions might not be preserved as usually we have no insertion timing information stored in T (i.e. (T, A)).

Collision resolution by open addressing

Operations

```
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<N);
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<N) && (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 $N!$ permutations as its probing sequence.**

Collision resolution by open addressing Probing

Definition 14 (Linear Probing). Given an ordinary hash function $h(\cdot)$, linear probing uses h such that $h(k, i) = (h(k) + c_0 + c_1 i) \bmod N$.

The problem with this method is **primary clustering**. Long runs of (consecutively) occupied slots are being built. Say we have $N/2$ keys in T . In case 1, these keys occupy the first $N/2$ position of T (clustering). In case 2, they occupy the odd positions of the array. In case 1 average probing is $N/8$, in case 2 it is 1.5.

Definition 15 (Quadratic Probing). Given an ordinary hash function $h(\cdot)$, quadratic probing uses $h(k, i) = (h(k) + c_0 + c_1 i + c_2 i^2) \bmod N$.

If no c_0 , c_1 and c_2 are specified then $c_1 = c_2 = 0$ and $c_2 = 1$. A milder form of clustering appears under quadratic probing which is called **secondary clustering**.

Definition 16 (Double Hashing). Given two hash function h_1, h_2 , double hashing uses h such that $h(k, i) = (h_1(k) + i h_2(k)) \bmod N$. The probe sequence depends on two hash functions i.e. depends twice on k .

In addition $h_2(k)$ must be relatively prime to N 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 N a power of 2 and force $h_2(k)$ to produce an odd number.

Collision resolution by open addressing (Worst case) performance

What do we store in the Hash table?

It is quite clear that in T we store either a reference (index or pointer) to A or a fixed-width key value if we don't need A . An example for the latter case is the fixed width k value associated with x (though we lose the x affiliation then).

If the x values are strings, we can store the strings the same way we stored them in Subject 1, for a doubly linked list implementation. We then store in T a reference (pointer or just index into A) and in the indexed position of A we store the string x NUL value delimited (i.e. the string value is ended with a byte of zeroes, the NUL character). An alternative would be to store at the indexed position the length of the string as a one byte, two byte or four byte quantity followed by the string itself (and with no NUL). In such a scheme it is also possible to store k in A as well.

Theorem 5. *In the worst-case the performance of Insert, Delete and Search is the performance of the corresponding operation in unordered array. Thus $O(N)$ is a derived bound in general; note this bound does not depend on n .*

In this bound we do not distinguish between a successful and unsuccessful search.

Definition 17 (Uniform Hashing). 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 $N!$ permutations as its probing sequence.

Theorem 6 (Unsuccessful Search). Under open-addressing with a hash table with load factor a , the expected number of probes for an unsuccessful search is at most $1/(1 - a)$ assuming uniform hashing.

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

$$UnSearch(n, N) = I(n, N)$$

It is obvious that $I(0, N) = 1$, for any N . $I(n, N) = 1$ if the $n + 1$ -st key hashes into an empty slot. The probability of this happening is $(N - n)/N$. Otherwise, with probability n/N 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

$$I(n, N) = 1 \cdot \frac{N - n}{N} + \frac{n}{N} (1 + I(n - 1, N - 1)) = 1 + \frac{n}{N} I(n - 1, N - 1).$$

Thus $I(n, N) \leq N/(N - n)$. Why?

Open-addressing Performance of Insert under UH

We may use induction and assume by the inductive hypothesis that $I(n-1, N-1) \leq (N-1)/(N-1-(n-1)) = (N-1)/(N-n)$. Then

$$I(n, N) = 1 + \frac{n}{N} I(n-1, N-1) \leq 1 + \frac{n}{N} \frac{N-1}{N-n} \leq 1 + \frac{n}{N} \frac{N}{N-n} \leq \frac{N}{N-n} = \frac{1}{1-a}$$

or rely on the iteration method instead by observing the following.

Since

$$I(n, N) = 1 + \frac{n}{N} I(n-1, N-1).$$

we substitute $0, 1, \dots$ for n in $I(n, N)$.

$$\begin{aligned} I(0, N) &= 1 \quad \forall N = \frac{N+1}{N+1-0} \\ I(1, N) &= 1 + \frac{1}{N} I(0, N-1) = 1 + 1/N = \frac{N+1}{N+1-1} \\ I(2, N) &= 1 + \frac{2}{N} I(1, N-1) = 1 + \frac{2 \cdot (N-1) + 1}{N \cdot (N-1)} = \frac{N+1}{N-1} = \frac{N+1}{N+1-2} \\ I(n, N) &= \frac{N+1}{N+1-n} \approx \frac{N}{N-n} = \frac{1}{1-a}. \end{aligned}$$

■

Open-addressing Performance of Insert under UH

Proof. (Exact Answer). We find q_i i.e. the probability that at least i probes access occupied slots.

$q_1 = n/N$ as n slots out of N are occupied.

For q_2 we have an unsuccessful probe (i.e found one slot occupied). Therefore the remaining $N - 1$ slots would contain $n - 1$ occupied slots and thus $q_2 = (n/N)((n - 1)/(N - 1)) \leq (n/N)^2$. In general $q_i \leq (n/N)^i = a^i$. Then the average number of probes is, where p_i is the probability that EXACTLY i probes access occupied slots (and the last one is empty).

$$\begin{aligned}
 1 + \sum_i i p_i &= \\
 1 + (p_1 + 2p_2 + 3p_3 + 4p_4 + \dots + \dots) &= \\
 1 + p_1 + p_2 + p_3 + p_4 + \dots + \dots & \quad \text{This is } q_1 \\
 + p_2 + p_3 + p_4 + \dots + \dots & \quad \text{This is } q_2 \\
 p_3 + p_4 + \dots + \dots & \quad \text{This is } q_3 \\
 p_4 + \dots + \dots & \quad \text{This is } q_4 \\
 &= 1 + q_1 + q_2 + q_3 + q_4 + \dots \\
 &= \sum_i q_i \leq 1 + a + a^2 + a^3 + a^4 + \dots + a^i + \dots \Leftrightarrow \\
 1 + \sum_i i p_i &\leq 1/(1 - a).
 \end{aligned}$$

■

Theorem 7 (Insert). *Under open-addressing with a hash table with load factor a , the expected number of probes for an Insert operation is at most $1/(1 - a)$ 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 $1/(1 - a)$. ■

Open-addressing Performance of Delete under UH

Theorem 8 (Delete). *Under open-addressing with a hash table with load factor a , the expected number of probes for an successful search operation is $1/a \ln(1/(1-a))$, assuming uniform hashing. This is also the time (expected number of probes) for a Delete.*

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 $1/(1-i/N) = N/(N-i)$ probes. Averaging over all keys we get (note $H_n = 1 + 1/2 + 1/3 + \dots + 1/n \approx \ln(n)$).

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

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

After a successful search, the found key can be deleted in constant additional time that does not affect further the number of probes already accounted. ■

Collision resolution by open addressing

Theory vs Practice

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 N 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 string 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 `docIDs`.)

Google's. In Google (original 1997 realization) one of the tables used stores URLs and `docIDs`. One table related uses `docID` 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. The average length of a word in an English dictionary is about 7 characters. Yet the average length of a word in a text is roughly 4 characters. This is because texts contain repetitions of words and in particular short ones such as articles, prepositions, pronouns, conjunctions.

Theory vs Practice

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 $N = 2n$, 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 characters is also the number of bytes used. Total space of the hash table is $N \cdot 32B = 2n \cdot 32B = 64nB$. (If one has a PC with 512MiB of space, a large capacity for 1997 configurations, then one could barely accommodate $n = 512/64 = 8\text{M}$ words with such a scheme.)

DRAFT

Theory vs Practice

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) **words**. The bitlength of a **pointer** can be 32-bits i.e. 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 uses $4N = 4 \cdot 2n = 8nB$ 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.

DRAFT

Theory vs Practice

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 $N = 2n$ we need $2n(\lg n + 1)$ bits for **wordIDs**. Definitely for $n < 2^{32}$ we would need no more than 4 Bytes per **wordID**. Thus **wordIDs** can consume $8n$ more bytes. If the previous scheme used $16n$ bytes of space this one uses $24n$, still better than $64n$!

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 1. 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?

Solution. 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 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 $N = 2n$ long, hash table performance will be optimal. Such a table of pointers uses $4N = 4 \cdot 2n = 8n$ bytes. In total the amount of memory used is roughly $8n + 8n = 16n$. Thus $16n = 256 \text{ MiB}$ 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.

Theory vs Practice

Case T2b

Solution continued. 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 $N = 2n$ 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). ■

DRAFT

Theory vs Practice

Case T3 Naive

T storing x that are English words: English Dictionary. Let us violate the rule that says no x appears in T . And let us violate the rule that says $n/N = 0.5$. We would like to maintain n keys (English words) into a dictionary structure T . Let us use a T with $N = n$ slots where a slot is wide enough to accommodate the longest English word with $w_{max} = 45$. Let the number of slots m be equal to number of keys n maintained. Let us assume that characters are ASCII and thus 1B/char. The size of T in B is ($N = n$)

$$|T| = m \times w_{max} = 45n$$

The average length of a word in a dictionary is $w_d = 7$. Thus the number of B of T storing characters is

$$|T'| = m \times w_d = 7n$$

Space utilization is then

$$\frac{7n}{45n} \approx 15\%$$

The utilization of available space is too low!

-----n=N=4	T = 4 x 45 =180
pneumonoultramicroscopicsilicovolcanoconiosis	T' = 45+4+3+5=57
alex	T' / T =57/180 ~31%
tom	
jerry	

Case T4

```

n      : # of keys
N      : # of slots of hash table T
a      : load factor  $a = n / N$  (Aim at 50%)
x      : a 'word' ;
k=key(x) : a numeric representation of a word
h(k)    : index of T storing info about word (k or x or none)  $0 \leq h(k) < N$ 
T[h(k)] : usually stores k and/or a pointer to some other space
          (eg an array A); that other space can contain x.
More over T[h(k)] can contain other info related to x or k
          or a pointer to memory that contains this info.
h(k,i)  : In open-addressing h(k) becomes a more complicated h(k,i)
          i : probing index  $0 \leq i < N$ 

```

RAM 256MiB, 56MiB (Linux + software).

English ASCII words:w_t = 5,w_d = 7,w_max =45

APPROACH 1 (NAIVE ala T3) * APPROACH 2 (BETTER)

Open addressing * Open addressing

a = 50% Desired and used * a=50% Desired and used

$$a = n/m = 50\% \Rightarrow N = 2 * n \quad * \quad N = 2 * n$$

Store words in T per * Store pointers/indexes in T to A

```

CASE 3 (~page 4)      *      Store words in A but NUL delimited
                       *      (NUL= 8 bit all zero)

```

```
T      *      T      A <----- n strings ----->
```

$\wedge$ ----- $*$ $\wedge$ ----- -----

```
| |          | * | | | | tom0alex0data0jerry0
```

					*					,
--	--	--	--	--	---	--	--	--	--	---

N=2n	jerry		*N=2n	p	
------	-------	--	-------	---------	--

		I			*			
--	--	---	--	--	---	--	--	--

| | * |

					*					
--	--	--	--	--	---	--	--	--	--	--

$$\sqrt{\quad} \quad * \quad \sqrt{\quad}$$

45B

```

|1| = m x w_max= 21x45=9011*      |1| = m x p = 21p (p pointer:#byte 1B)

```

$$|A| = \prod x_i (w_i + NOL) = \prod x_i (i+1) = \delta \prod$$

20. $\approx (253.53) \text{ M.B.}$

3011 (200 00)H1D 1011 (200 00)H1D
~ 3M ~ 12M

.....

APPROACH 2 = 6 = APPROACH 1 (BTU 9-20-10-01007 : = =

Notice

Problem Set with Solutions

These Problem Sets are not for credit. They are provided as are for your practice. Several problems are provided with solutions. These can be discussed with the instructor after a student has studied the relevant material. Some other problems are provided with either no solutions at all or a hint of a solution. They are there for your reference and will not be discussed further. For some solutions provided you might have to fill in details related to calculations or intermediate steps.

Before you try these problems: (a) Study the designated textbook, (b) use the Subject notes (typewritten notes available through the course web-page from the password-protected section C) to review your study of the designated textbook material, (c) resolve the examples done in class, (d) then practice with some of these problems PS (Problem Set), and then if you have time, (e) solve some of the problems of the designated textbook, and (f) solve more problems from the PS especially the ones with minimal solution information.

Several PS problems appeared in past exams or homeworks.

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 1.

How many words are there in the textbook? Justify your answer. (BTW GT has 708 pages.) Make sure your answer has lots of trailing zeroes. Thus 1,000,000 has trailing zeroes, but 123,456 does not have. Even if the correct answer is 100,000 the former will get you 0 points but the latter will get you a 0!

Proof. 708pages and roughly 45 lines per page and roughly 80 characters per page gives 2.4M characters. The average length of an English word in a text is 5 thus roughly 500,000 words. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 2.

- (a) Consider inserting keys 25,19,13,7,1 into a hash table of length $N = 6$ using open addressing with the auxiliary hash function $h(k) = k \bmod N$. Illustrate the result of inserting these keys using (i) linear probing with hash function $h(k, i) = (h(k) + i) \bmod N$, (ii) quadratic probing with hash function $h(k, i) = (h(k) + i^2) \bmod N$.
- (b) Consider inserting keys 7,13,1,37,25. (Same set up as before.)
- (c) Consider inserting keys 11, 22, 20, 6, 26, 16, 7 into a hash table of length $N = 9$; same setup otherwise as before.
- (d) Consider inserting keys 1, 3, 5, 6, 12, 23, 10, 17, 34 into a hash table of length $N = 10$ using open addressing with the primary hash function $h(k) = k \bmod N$. Illustrate the result of inserting these keys using (i) linear probing with hash function $h(k, i) = (h(k) + i) \bmod N$, (ii) quadratic probing with hash function $h(k, i) = (h(k) + i^2) \bmod N$.

Proof. (a) and (b).

n=5 m=6

@a,b,c, etc means collisions at slot a,b,c, etc in that sequence
for i=0, i=1, i=2, etc

Part (a)	Part (b)
25,19,13,7,1	7,13,1,37,25
0:	0:
1: 25	1: 7
2: 19 @1	2: 13 @ 1
3:	3:
4: 7 @1,2,5	4: 37 @ 1,2,5
5: 13 @1,2	5: 1 @ 1,2
1 @ 1,2,5,4,5,2 overflow!	25 @ 1,2,5,4,5,2 overflow!
#Collisions: 12	#Collisions : 12

- (c). Skipped.
- (d). Shows an alternate description of collisions

Problem Set Continued

	Linear Probing	Quadratic Probing
Slot		
0 :	10	10
1 :	1	1
2 :	12	12
3 :	3	3
4 :	23@3	23@3
5 :	5	5
6 :	6	6
7 :	17	17
8 :	34@4,5,6,7	34@4,5
9 :		

k@a,b,c means key k had collision at a,b,c before inserted here

In the Linear probing case you can observe the long cluster that causes problems to 34. Observe that for quadratic probing the collisions are fewer (by two) for the same key.



Problem Set
Problem 3.

Answer the following questions by justifying your answer. If necessary provide asymptotically tight bounds. Expected times are under the SIMPLE UNIFORM HASHING (function) ASSUMPTION.

- (a) What's the running time of Hash Table Search for collision resolution by chaining?
- (b) What's the running time of Hash Table Unsuccessful Search for collision resolution by chaining?
- (c) What's the running time of Hash Table Delete(z) for collision resolution by chaining? (You know the element containing the key).
- (d) What's the running time of Hash Table Delete(k) for collision resolution by chaining? (You know the key value only.)
- (e) What's the expected running time of Hash Table Insert for collision resolution by chaining? (You know the key value only.)
- (f) What's the expected running time of Hash Table Successful Search for collision resolution by chaining? (You know the key value only.)
- (g) What's the expected running time of Hash Table Unsuccessful Search for collision resolution by chaining? (You know the key value only.)

Proof. (a) $O(n)$. (b) $\Theta(n)$, (c) $\Theta(1)$. (d)-(f) Skipped. (g) $\Theta(1 + \alpha)$.

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 4.

Answer the following questions by justifying your answer. If necessary provide asymptotically tight bounds. For expected running time it is under Uniform Hashing assumption.

- (a) What's the running time of Hash Table Successful Search for collision resolution by open-addressing?
- (b) What's the running time of Hash Table Unsuccessful Search for collision resolution by open-addressing?
- (c) What's the running time of Hash Table Insert for collision resolution by open-addressing assuming the key is not already in the table?
- (d) What's the expected running time of Hash Table Successful Search for collision resolution by open-addressing?
- (e) What's the expected running time of Hash Table Unsuccessful Search for collision resolution by open-addressing?
- (f) What's the expected running time of Hash Table Insert for collision resolution by open-addressing assuming the key is not already in the table?
- (g) What's the expected running time of Hash Table Insert for collision resolution by open-addressing assuming the key is not already in the table? The table contains 50 keys before the insertion and it has 1000 slots.
- (h) What's the expected running time of Hash Table Delete for collision resolution by open-addressing assuming the key is not already in the table? The load factor prior to deletion is $1 - 1/e$ where e is the base of the Neperian algorithms. ($1 - 1/e$ is roughly 60%). Which of the following is close to the average number of probes (i) 1.6, (ii) 3.5, (iii) a , (iv) 5.5, (v) e^3 ?

Proof. (a) $O(N)$. (b) $\Theta(N)$, (c) $O(N)$. (d)-(g) Skipped. (h) (i)

■

Problem Set Problem 5.

Suppose that we insert n keys into a hash table of size m using open addressing and uniform hashing. Let $p(n, m)$ be the probability that no collisions occur. Show that $p(n, m) \leq \exp(-n(n-1)/(2m))$. Argue that when n exceed $\sqrt{m}$, the probability of avoiding collisions goes rapidly to zero. **Hint:** Use induction... Also use $\exp(x) \geq 1 + x$.

Proof. Consider $p(n+1, m)$ and argue similarly to the derivation for Insertion done in class. For the $n+1$ -st key not to cause any collision in a hash table with n keys inserted with no collisions, it must hit an empty slot an event that occurs with probability $1 - n/m$. Therefore $p(n+1, m) = (1 - n/m)p(n, m)$, since the probability that the insertion of say $n+1$ cause not collision is the probability that the insertion of the $n+1$ -st key causes no collisions and the prior insertion of the remaining n keys caused no collisions; the latter is $p(n, m)$. Therefore

$$\begin{aligned} p(n+1, m) &= (1 - n/m)p(n, m) \\ &= (1 - n/m)(1 - (n-1)/m)p(n-1, m) \\ &= (1 - n/m)(1 - (n-1)/m) \dots (1 - 1/m)(1 - 0/m)p(0, m) \\ &= (1 - n/m)(1 - (n-1)/m) \dots (1 - 1/m) \end{aligned}$$

Since $e^x \geq 1 + x$, setting $x = -i/m$ we obtain that $1 - i/m \leq e^{-i/m}$. Therefore

$$\begin{aligned} p(n+1, m) &= (1 - n/m)(1 - (n-1)/m) \dots (1 - 1/m) \\ &\leq \exp(-n/m) \exp(-(n-1)/m) \dots \exp(-1/m) \\ &\leq \exp(-(1 + 2 + \dots + n)/m) \\ &= \exp(-n(n+1)/2m) \end{aligned}$$

Therefore $p(n, m) \leq \exp(-(n-1)n/(2m))$ as required. Consider $n \gg \sqrt{m}$, the $n(n-1)/2m \gg 1$, and thus $p(n, m)$ is upper bounded by $\exp(-A)$, where A is large and positive. However $\exp(-A)$ goes to zero and thus $p(n, m)$ goes to zero as well.

This problem is also known as the Birthday problem. Let $m = 365$ (forget about leap years). Suppose you have n people in a room. What is the probability of having no two with the same birthday? For what value of n is the probability of having two people with the same birthday close to 1? One can see that for $n = \sqrt{m}$ collisions start to show up with a significant (non negligible) probability, i.e. people exist with the same birthday. ■

Problem Set

Problem 6.

Suppose that we are given a key k to search for in a hash table with positions $0, \dots, m-1$, and suppose that we have a hash function h mapping the key space into the set $\{0, \dots, m-1\}$. The search scheme is as follows.

1. Compute the value $i = h(k)$ and set $j = 0$.
2. Probe in position i for the desired k . If you find it, or if this position is empty, terminate the search.
3. Set $j = (j+1) \bmod m$ and $i = (i+j) \bmod m$ and return to step 2.

Assume that m is a power of 2.

- a. Show that this scheme is an instance of the general quadratic probing scheme by exhibiting the appropriate constant c_1, c_2 for $h(k, i) = (h(k) + c_1 i + c_2 i^2) \bmod m$.
- b. Prove that this algorithm examines every table position in the worst case.

Proof. a. The hashing scheme searches entries $h(k), h(k) + 1, h(k) + 1 + 2, h(k) + 1 + 2 + 3, \dots, h(k) + 1 + 2 + \dots + j \bmod m$, where m is a power of two.

The value of i after j iterations (i.e. after j probes) is $i = h(k, j) = h(k) + 1 + 2 + 3 + \dots + j = h(k) + j(j+1)/2 = h(k) + 1/2 j + 1/2 j^2 \bmod m$, i.e. we indeed have a quadratic probing scheme with $c_1 = c_2 = 1/2$ and hash function $h(k, j)$.

b. (Proof by contradiction). Suppose that the algorithm does not examine every table position in the m probes $h(k, j)$, $j = 0, \dots, m-1$. This means that at least one table entry is visited at least twice (and at least one is not visited at all) i.e. there exist $0 \leq l, r \leq m-1$ such that $h(k, l) = h(k, r)$, $l \neq r$ and therefore $h(k, l) - h(k, r) \equiv 0 \pmod{m}$. Without loss of generality, we assume $l < r$. We shall then show that for l, r as above we have that $h(k, l) \neq h(k, r)$, and therefore $h(k, l) - h(k, r) \not\equiv 0 \pmod{m}$, i.e. a contradiction. It is true that $h(k, r) - h(k, l) = r(r+1)/2 - l(l+1)/2 = (r-l)(r+l+1)/2 \equiv 0 \pmod{m}$.

Consider $(r-l)(r+l+1)/2 \equiv 0 \pmod{m}$ or in the other words. $(r-l)(r+l+1)/2 = Am$ for some integer A . For this to be true, m or a power of two must divide $r-l$ or $r+l+1$ or both. We distinguish two major cases.

Problem Set Continued

1. Let $r - l$ be even (i.e. r, l are both odd or both even). Then, $r + l + 1$ is odd and it is not divisible by m or 2 as both m (a power of 2) and 2 are even and $r + l + 1$ was shown to be odd. On the other hand, $r - l \leq m - 1$ as $r, l \leq m - 1 < m$. Then $(r - l)/2 < m$ and therefore m cannot divide $(r - l)/2$ either. This means that $(r - l)(r + l + 1)/2 \not\equiv 0 \pmod{m}$, since m can divide neither $r + l + 1$ nor $r - l$.
2. Let $r - l$ be odd. Then, if one of r, l is odd, the other one is even, and the sum $r + l + 1$ is even. As $r - l$ is odd it is not divisible by m or 2. We shall show that $(r + l + 1)/2$ is not divisible by m as well. As $r, l \leq m - 1$, $r + l + 1 \leq 2(m - 1) + 1 = 2m - 1$. $r + l + 1$ is even, $2m - 1$ is odd, thus $r + l + 1$ can not be equal to $2m - 1$ and therefore $r + l + 1 \leq 2m - 2 = 2(m - 1)$. Then $(r + l + 1)/2 \leq m - 1 < m$ and therefore it is not divisible by m (as this would imply that $(r + l + 1)/2 \geq m$). This means that $(r - l)(r + l + 1)/2 \not\equiv 0 \pmod{m}$ as well.

For both cases we derived a contradiction $h(k, l) - h(k, r) = (r - l)(r + l + 1)/2 \not\equiv 0 \pmod{m}$ to the assumption that $h(k, l) - h(k, r) \equiv 0 \pmod{m}$. Since no two values among the 0 through $m - 1$ collide with each other, they cover all the values/indices/slots of the hash table as required. ■