Paper 8-26

Table Look-Up by Direct Addressing: Key-Indexing -- Bitmapping -- Hashing

Paul M. Dorfman,
CitiCorp AT&T Universal Card, Jacksonville, FL

ABSTRACT

Table look-up is the most time-consuming part of many SAS programs. Base SAS offers a rich collection of built-in searching techniques. Merging, SQL joins, formats, SAS indexes - all serve the purpose of locating relevant data. For custom programming, SAS offers arrays, whose direct addressability lends itself to implementing just about any searching algorithm. Array-based lookup is not a ready-to-go food; it has to be cooked at home. However, it may result in dishes digested by the computer more easily, more programmatically nutritious, and with fewer computer resources ending up in the garbage disposer.

This paper shows how arrays can be used to organize the fastest class of in-memory table look-up -- direct-address searching. Three such techniques -- key-indexing, bitmapping, hashing -- are considered in a logical sequence using a real-life example of matching two data files by a common key. The results of benchmarking presented in the paper show that home-cooked direct addressing methods beat even the quickest ready-to-go tools like the "large formats" by a wide margin. As such, they can be indispensable in any massive data processing setting, where speed and efficiency considerations are paramount.

INTRODUCTION

Table lookup being one of the most frequent data processing operations, SAS provides a rich collection of built-in searching techniques. Merging, SQL joins, formats, indexes, to name a few, all serve the purpose of looking up relevant data. In addition, SAS Language incorporates arrays – the data structures ideal for implementing just about any searching algorithm "by hand". SAS arrays are not ready-to-go tools: Array-based lookups have to be custom-coded and tuned. However, this approach is more flexible and often results in programs that search faster and use fewer resources than the "heavy artillery".

This paper concentrates on a group of in-memory lookup methods based on direct or almost direct addressing into a temporary SAS array. First, we shall consider *key-indexed search.* Then we will try to expand its domain by viewing an *array as a bitmap*. Finally, we will see how to generalize the core idea of key-indexing to arrive at a hybrid search method called *hashing*.

To make the discussion less abstract, we will consider a common task of matching two data files by a common variable. This will help us how different lookup techniques compare to each other and to some of the ready-to-go methods such as "large formats" and SQL.

Consider an unsorted file SMALL containing N_SMALL records with a key variable KEY and satellite variable S_SAT. Another unsorted file, LARGE, with N_LARGE records, also contains KEY and a satellite field L_SAT. *Let us assume, for the time being, that the keys are integers.* Imagine that LARGE is so big that sorting is not an option; however, also assume that we have enough memory to hold all keys from SMALL at once. Under these conditions, *What is the most efficient way to subset LARGE based on the values of KEY in SMALL* to produce a file MATCH? SAS offers a number of ready-to-go tools based on in-memory table lookup. For example:

1. Compile unduplicated keys from SMALL into a format using CNTLIN= option, and search it for each KEY read from LARGE.
2. Join the files using BUFFERSIZE large enough to prompt the SQL optimizer to use SQXJHSH access method.
3. Load the keys from SMALL into a sorted array, then use a hand-coded binary or interpolation search.

With plenty of methods available, why try something else? Because there are faster and more efficient ways to do the trick!

I. KEY-INDEXING

Most of the ready-to-go and hand-coded searching methods are based on *comparing a search key to all or some keys in a memory table.* It makes them *principally limited* since generally, no comparison-based method can search in fewer iterations than binary search. We could therefore try to remove the limitation by *doing away with key comparisons* altogether. But is it possible to search for a key without at least one comparison? The answer is "yes" and given by a radically different searching philosophy called *direct addressing,* that finds its pure expression in *key-indexed search.* Its idea is simple. Imagine that all keys are 1-digit numbers from 0 to 9, and that SMALL has just 9 records:

OBS	1	2	3	4	5	6	7	8	9
KEY	2	3	5	2	7	9	5	7	3
S_SAT	1	2	3	0	4	5	6	9	7

Let us create a temporary array HKEY with *one node (location, address) allocated for each possible key value,* and initialize the contents of all buckets to a missing value. (In SAS, such initialization will be done be default. In the case the satellites might have legitimate missing values, the table can be primed using special missing values.) The array HKEY can be thought of as the following table in memory:

H	0	1	2	3	4	5	6	7	8	9
HKEY	.	.	.	.	.	.	.	.	.	.

Now, for each key from SMALL, let us look at the array location H *whose index is equal to the value of the KEY,* that is, simply at HKEY(KEY). Since we have created a separate bucket for each possible key value, we are always guaranteed to find the address with H=KEY. Let us check first if the node is empty, i.e. if HKEY(KEY) is missing. If it is empty, let us move the satellite S_SAT to H=KEY. After repeating this procedure for all nine test keys, HKEY acquires the following shape:

H	0	1	2	3	4	5	6	7	8	9
HKEY	.	.	1	2	.	3	.	4	.	5

What we have just created is termed a *key-indexed table*. It comprises two types of *entries*: *empty and occupied*. Inserting the satellite only when the node is empty retains its first instance corresponding to a repeating key value; otherwise, the last instance would be used. Either way, *duplicate keys are deleted automatically* as the table is loaded. If SMALL has no satellites or they are of no interest, the entries of the key-indexed table could be marked as occupied by moving 1 into the node, the unduplication effect remaining intact.

Given a search key to look for, all we have to do is examine the table location *whose index is equal to the key*. If the corresponding location is empty (missing), the key is not in the table. If it is occupied, the search has been successful, and the node contains the satellite value related to the search key. For example, if KEY=1, the search fails since the address 01 of the table is empty. If KEY=7, we have to look at the node 07. It is occupied; therefore, the key is found, and the node returns the satellite value HKEY(7)=4.

Note that *searching is implicitly incorporated in the process of loading the table*: To determine if the node is empty, we in effect search the table to find out if the key has already been marked in the table as present. If it has, it is a duplicate, and need not be inserted, so we can merrily proceed to the next record. The nature of the process makes it unnecessary to sort SMALL or insert the keys themselves, because effectively, the keys are "inserted" by making their corresponding nodes occupied by satellites or, in lieu of the satellites, by a non-missing value, e.g. 1.

The utter simplicity of key-indexing lends itself to a very simple DATA step implementation. Suppose, for example, that we are dealing with integer keys ranging from –4E6 to +4E6. The range thus naturally defines the bounds of the array HKEY representing the key-indexed table:

```
** Key-Indexed Load and Search **;

data match;
   array hkey (-4000000:4000000) _temporary_;
   ** load key-indexed table from small;
   do until (eof1);
      set small end=eof1;
      if hkey(key) = . then hkey(key) = s_sat;
   end;
   ** for each obs in large, search table and output matches;
   do until (eof2);
      set large end=eof2;
      s_sat = hkey(key);
      if s_sat > . then output;
   end;
run;
```

From the nature of the algorithm, it is clear that *no lookup method is simpler and/or can run faster than key-indexing*: It completes any search, hit or miss, without comparing any keys, via a single array reference. It also possesses the fundamental property: Its speed does not depend on the number of keys "inserted" into the table, i.e. any single act of key-indexed search takes precisely the same time.

To see how well key-indexing performs, it was compared in load and search phases to formatting, SQXJHSH, and other methods presented below, for N_LARGE=2E6 and a number of N_SMALL values using SMALL and LARGE. The results shown in the Section "TESTING" testify that - at a high memory expense - key-indexing completely dominates the competition. For instance, it out-performs MERGE running against (pre-sorted!) input as 5:1.

Well, if key-indexed search is all that good, why not use it at all times and instead of everything else? Unfortunately, there is a fly on the ointment. As we have seen, key-indexing, due to its very nature, is practically applicable only when the lookup keys are integers falling in a limited range. For our test keys taking on only as many as 8,000,001 distinct values, sufficient array space can be allocated using about 64 MB of memory. Having 80 MB of memory, one can get away with 7-digit keys. However, to deal with 9-digit SSN, an array with 1 billion elements would be needed, which is almost impossible even with the modern memories, while 16-digit credit card numbers would make key-indexing a technical utopia.

On the other hand, there is a plenitude of real-world applications where key values do indeed fall in a limited range. In all such cases, key-indexing, with its blazing speed and simplicity, is beyond competition. Here are some examples:

1. SAS date is simply the number of days between a given date and 01JAN1960. Any date value, from the lowest possible in the SAS System up to year 4000 can be accommodated by a [-138061:380217] table, and it will occupy mere 4 MB of real storage (RAM).
2. SAS times. An array sized as [0:86400] will key-index any SAS time value.
3. ICD9/CPT4 codes. If some character is a letter, it can be converted into a number to 1-26 range, and then the entire code can be represented as a limited-range integer, probably not exceeding 1 million.
4. PIB2. informat maps any 1- and 2-byte character key onto the [0:65535] range.
5. Any fractional key if limited in range when multiplied by a suitable scaling constant.

So, key-indexing can be really useful and extremely fast in a variety of data processing situations. And yet, it remains inherently limited to the domain of restricted-range, integer keys. But the idea on which the method is based is so beautiful that it would be a shame to let it go underused just because of its greed for memory. It is worth trying to expand it. However, to do so, we must find a practical way to keep memory usage at bay.

The question is: How? First, let us observe that both the speed of key-indexing and its limitations rest upon several simple facts:

- The lookup table is directly addressed by the value of a key itself.
- The entire set of possible key values is addressable. It means that a separate node must be allocated for each possible value a search key can assume.
- No comparisons between the search key and any key in the table are made.

Based on these facts, we can devise two principal approaches that could loosen the restriction self-imposed by key-indexed search.

1. Keep all possible key values addressed, but *expand the addressable range of keys* by making much smarter use of the available memory resources. For, instance, we can try to key-index bits instead of bytes.
2. Eliminate *any restrictions* imposed on the nature of lookup keys. This can be done by dropping the requirements that (a) no two distinct keys shall reside in one node, and (b) no comparisons between the search key and keys in the table shall be allowed.

The first approach results in a technique called *bitmapping*. The second path leads to a more versatile hybrid searching method known as *hashing*.

II. BITMAPPING

Suppose we have a situation where the satellite information in SMALL is of no interest for us, and therefore we need not drag S_SAT through the memory to the output. In such a case, the key-indexed table only serves one purpose: To indicate whether a memory node, whose index corresponds to the key value, is empty or occupied. The occupied nodes can be populated with 1, and the empty ones can be either left missing or else initialized to zero. Now if a node number KEY contains 1, the key whose value is KEY is present. Otherwise, if the node number KEY contains 0, the key is not in the table. Hence, all a table node must be able to tell is whether it contains 0 or 1. Such functionality can be amply served by nothing more than a single bit. Yet looking back at our key-indexing implementation, we see that it uses full 8 *bytes*, the memory length of a numeric array item, to store a binary value. This is 64 times the difference! But if we to realize such a potential, how do we make efficient use of bits in such a setting?

If it were possible to have a temporary array with 1 bit per item, the question would not even arise – we would simply have the bits addressed directly via an index. Unfortunately, the shortest *memory* length reserved by a temporary SAS array element is not 1 bit but always full 64 bits regardless of the declared *expression* length. If, for example, a temporary array is allocated as $1, its item's expression length is 1, but its memory length is still full 8 bytes. Hence, to properly index the bits that compose an array element, some additional computations are needed.

At first glance, it seems natural to try bitmapping a character array with the shortest allowable memory length, $8, and the number of elements equal to the number of all possible key values divided by 64. This would obviously allow cutting memory usage 64 times, just as projected above. Say, for the time being, that we are dealing with 8-digit natural keys. In order to be able to address all of them, we will need, accordingly, 100 million (1E+8) bits. The equivalent amount of real storage, about 12 MB, is nowadays insignificant, especially compared to what key-indexing would require (760 MB). The entire universe of all possible keys can be thus covered by the bits of an array consisting of 1,562,500 8-byte character items. To mark a key as present, we might proceed as follows:

1. Declare ARRAY BITMAP (0:1562500) $8 _TEMPORARY_.
2. Compute X= INT(KEY/64). Now X points to the array item containing the bit having to be turned on.
3. Compute R=1+MOD(KEY, 64). Now R points to the correct bit.

To turn the bit on, we would compute X and R. If BITMAP(X) is blank (none of bits is on yet), we would set it to the binary zero "0000000000000000"X, then convert it to a 64-byte character variable BITSTR mirroring the bit content of BITMAP(X) , set R-th byte to 1, reconstruct BITMAP(X), and reinsert it into the proper array location:

```
BITSTR = PUT(BITMAP(X),$BINARY64.);
SUBSTR(BITSTR,R,1) = '1';
BITMAP(X) = INPUT(BITSTR,$BINARY64.);
```

This way, marking the keys from SMALL as present one by one, we would eventually "compile" the entire bitmap. Now, given a key from LARGE, how would we search for it? First of all, we would find X and test BITMAP(X). If it is blank, then we have a miss since obviously none of this item's bits has been turned on. Otherwise, if the entire item is not blank, one or more of its bits are on, and we have to compute R and check the R-th bit using either of the two expressions:

- `SUBSTR(PUT(BITMAP(X),$BINARY64.),R,1) EQ '1'`
- `INPUTN(BITMAP(X),'BITS1.'||PUT(R,Z2.))`

Both of them evaluate to 1 if the R-th bit is 1, and to 0 if the R-th bit is 0.

This looks fairly simple and actually does work! Unfortunately, when this scheme was implemented and tested, it returned lackluster performance. And, after some reflection, it should be no surprise: Just to examine a bit or turn it on, we in effect have to either memory-write full 64 bytes or use a modified informat which, unfortunately, executes quite slowly. (Note that in version 8.1, the situation might have changed because supposedly, 8.1 will allow to allocate character temporary arrays with $1 memory length.)

To achieve a decent searching speed (the name of the whole game), we must find a way to spot individual bits much more rapidly. In turn, this can only be achieved by performing some kind of fast computation on the entire array item rather than by breaking it apart – which immediately suggests using a numeric array instead of the character one. That brings about two interrelated questions:

1. If a numeric array is chosen to represent the bits of a bitmap, what sort of rapid operation can be performed on a numeric item in order to turn its R-th bit on?
2. Given a numeric item, what kind of rapid direct calculation can we use to find out whether R-th bit is on?

The first issue is resolved easily. If R-th bit of a numeric item is off, adding 2**(R-1) to the entire item is equivalent to turning its R-th bit on. Moreover, we do not even have to compute the binary power on the fly, since a series of consecutive binary powers can be pre-computed and stored in an auxiliary array. To resolve the second issue, it is first necessary to delve into a number of subtle caveats.

Caveat 1. It must be never attempted to turn a bit on if it is already turned on. Otherwise the added unity will become a carry turning one or more of the higher bits to 1, thus wreaking havoc in the entire bitmap. So, the answer to question 1 above cannot be found without answering question 2 first. (Of course, bit checking is superfluous if BITMAP(X) is missing, as none of the bits has been set to 1 yet.)

Caveat 2. Before the very first bit can be turned on, a missing item must be set to 0. However, initializing the entire array is not necessary, because at the searching stage, the missing items can be simply skipped.

Caveat 3. How many bits per array element can we use in such a manner? Since they must be limited to the mantissa, it leaves us with 56 usable bits per element under OS/390 and 53 bits under NT, so the divisor 64 in the formulae above must be changed accordingly. Thus, by switching to a numeric array, we sacrifice about 15% memory utilization for the sake of speed.

Now we can answer question 2 posed above, but to do so, let us first figure out how to determine whether, say, the 4th least significant *decimal* digit of a numeric variable N is not zero. Naturally, we would divide N by 1E+4 and find the remainder. If the latter is not less than 1E+3, the digit being tested is not zero; otherwise, it is zero. By induction, it can be concluded that a boolean expression

```
MOD(N,10**R) => 10**(R-1)
```

indicates whether R-th decimal place is "on" or "off". By the same token, the expression

```
MOD(N,2**R) => 2**(R-1)
```

returns 0 or 1 depending on whether R-th bit of N is turned off or on. So, in effect, the result to which the expression above evaluates is set directly to the value of R-th bit of N's mantissa. That gives us all we need to key a numeric-array bitmap:

Step 1. Allocate a numeric temporary array BITMAP bound from 0 to 10**[number of key digits]/M, where M=56 or 53, depending on OS. Create an auxiliary array B and fill it with the serial powers of 2.

Step 2. Read a record with KEY from SMALL.

Step 3. Locate the array element: X=INT(KEY/M). If BITMAP(X) is missing, then set it to 0.
Step 4. Locate the right bit within the array element: R=MOD(KEY, M). If BITMAP(X) is missing, go straight to Step 5. Otherwise check the bit. If it is already 1, then the key is a duplicate: return to Step 2.
Step 5. Turn R-th bit on: BITMAP(X) ++ B(R), and go back to Step 2.

Given a bitmap "compiled" by the procedure just detailed, searching for a key is equally simple:

Step 1. Read a record from LARGE.
Step 2. Locate the array element: X=INT(KEY/M). If BITMAP(X) is missing, the search is unsuccessful, so go back to Step 1.
Step 3. Locate the bit: R = MOD(KEY, M).
Step 4. Check the bit. If it is zero, the key is not found, hence go to Step 1. Otherwise, write the record out and go back to Step 1.

Translating the algorithm into the SAS Language is now straightforward, but first the possibility of negative keys should be accounted for. With key-indexing, it is natural and easy to do by giving the lower bound of the table the lowest negative key value. In the case of a bitmap, we have to rummage around with the keys a little bit first: Shift them up by the absolute value of the lowest key (say, MINKEY), rescale the upper BITMAP bound accordingly, and leave the lower bound at 0.

```
*** Bitmap Table Load And Search ***;

%let m      =   56; *** if OS/390, else 53;
%let minkey = -4e6;
%let maxkey = +4e6;
%let lo = %sysfunc(floor(&minkey/&m));
%let hi = %sysfunc(floor(&maxkey/&m));
%let hb = %eval(&hi - &lo);

data match (keep=key l_sat);
   array bitmap (0:&hb) _temporary_;
   array b      (0: &m) _temporary_;

   ** precompute powers of 2;
   do x=0 to &m; b(x) = 2**x; end;

   ** load the bitmap from SMALL;
   do until (eof1);
      set small end=eof1;
      x = int((key - &minkey) / &m);
      if bitmap(x) eq . then bitmap(x) = 0;
      r = key - &minkey - x*&m;
      if mod(bitmap(x),b(r+1)) < b(r) then bitmap(x) ++ b(r);
   end;

  ** search bitmap for keys from LARGE, output matches;
   do until (eof2);
      set large end=eof2;
      x = int((key - &minkey) / &m);
      if bitmap(x) eq . then continue;
      r = key - &minkey - x*&m;
      if mod(bitmap(x),b(r+1)) => b(r) then output;
   end;
   stop;
run;
```

The macro assignments at the top take care of the necessary shifting and rescaling. M=53 will work under any OS; however, under a system with 56-bit mantissa, say OS/390, it is possible to choose M=56 instead and thus improve memory utilization. The first DO loop populates the auxiliary array B with the powers of 2. The process in the <EOF1> DO loop can be thought of as "bitmap compilation". Finally, the <EOF2> DO loop performs the actual bitmap search and outputs the records from LARGE whose key imprints are found in the bitmap. Because of the extra computations, bitmapping runs about 50 per cent slower than key-indexing, yet it uses 53 to 56 times less memory and is still twice as fast as MERGE after sorting!

In the approach implemented above, the bitmap compilation loop resides in the same step where it is searched. However, a bitmap can also be compiled in a separate step, stored in a file, and reused thereafter. To do so, we would only have to change the DATA statement to, for instance,

```
data lib.bitmap (keep=byte8);
```

and replace the entire <EOF2> loop with

```
do x=0 to &hb;
   byte8 = bitmap(x);
   output;
end;
```

The name of the variable BYTE8 is, of course, absolutely arbitrary. At this point, the BYTE8 in each observation of the SAS data set LIB.BITMAP corresponds to exactly one array item of the would-be bitmap. A bitmap saved in this manner can be then utilized any time a file similar to LARGE needs to be subset, validated, scrubbed, etc., based on the bit pattern stored in the bitmap, only in the beginning of the searching step, the array BITMAP must be loaded in memory from LIB.BITMAP one item at a time. This is accomplished by replacing the <EOF1> DO loop with

```
do x=0 to bitobs-1;
   set lib.bitmap nobs=bitobs;
   bitmap(x) = byte8;
end;
```

Principally, key-indexing and bitmapping are nearly twins:

- The table size is dictated solely by the overall key range and not by the number of lookup keys.
- Neither the "driver" file nor "master" file has to be sorted.
- Duplicate lookup keys are eliminated automatically as the table is "loaded".
- The speed of searching does not depend on the number of keys.

However, there are a few differences worth noting as well:

- Given the same memory resource, bitmapping has the addressable key range 53 to 56 times that of key-indexing.
- A bitmap cannot be used for dragging lookup satellites through memory into the output.
- With key-indexing, locating and rejecting a key takes exactly the same time. With bitmapping, a successful search requires, on the average, slightly more computing, and therefore, more time.

Because of its relatively wide key range and purely direct-addressing nature, bitmapping operates with incredible speed in a niche no other searching method can touch. As an example, imagine SMALL file with 50 million 8-digit keys (hardly "small" but let us stick with the name), and LARGE with mere 100 million records – figures not unusual in data warehouses nowadays. Sorting either one for MERGE is not exactly painless operation, especially if L_SAT tail is long, or say LARGE is actually is a view into a RDBMS table. Storing the lookup keys in a format is practically hopeless, as memory usage by a format usually tops 600 MB already at mere 10 million keys. Key-indexing would consume about 800 MB. A hash table (described later) would need at least 400 MB. However, a bitmap can be safely compiled with 120 million bits of temporary array storage in the worst case scenario (M=53). It means that it can be easily accommodated within only 15 MB of memory! If we decide to store the bitmap on disk, it will take about 1.9 million 1-variable observations; loading such a file into an array is a matter of several seconds. What is more, the lookup speed and memory usage will remain exactly the same, no matter whether we have 100, 100,000, or 100,000,000 keys to search.

Thus, the bitmapping niche can be defined as "no-matter-how-many-short-keys". Bitmap is a champion when we only need to rapidly find out if the record with a given key should be selected, and if memory resources are sufficient for key-indexing the entire key range into memory bits.

But how can we capitalize on direct addressing if the keys have a huge, say 16-digit, range, or are long character strings (256-radix integers), and therefore neither key-indexing nor bitmapping can do the job? Welcome to *hashing* !

III. HASHING

As noted above, compared to key-indexing, bitmapping changes nothing principally – it simply expands the workable universe of keys about 53+ times by using memory more efficiently. Hashing methods approach the problem quite differently: They eliminate the requirement of a separate slot for each possible key *and* allow some amount of comparisons between the search key and keys in the table. A simple example might be the easiest way of making the idea transparent. Let us suppose that SMALL contains just 10 3-digit keys:

185 971 400 260 922 970 543 532 050 067

To use key-indexing, we would have to allocate a table sized as [0:999] and map each key to the node corresponding to its value. Out of 1000 table nodes, only 10 will end up occupied, while the rest will play the role of placeholders, that is, will be simply wasted! The crucial question is, therefore, *Can we get away with a smaller table* of a reasonable size, only somewhat larger than the number of keys at hand, and still be able to take advantage of direct addressing?

Let us choose some number HSIZE greater than the number of keys N_SMALL in SMALL, for instance, 17, and allocate an array sized as HKEY(0:17). Let us agree to call the array HKEY the *hash table*, HSIZE - the *hash table size*, and the ratio N_SMALL/HSIZE - the *load factor*. Thus, the load factor shows the number of lookup keys relative to the total number of nodes in the hash table, in other words, how *sparse* the hash table is. In our example, the load factor equals 0.588, that is, the hash table is about 41 percent sparse.

Imagine some rapidly-computing function H(KEY) taking a key as an argument and returning an address into HKEY, unique to each key supplied, so that H(KEY) would map each key to its own location in a one-to-one manner. Were such *perfect hash function* available, we would only have to plug it in the code for key-indexed search and be done. Such functions *are* possible; however, they are quite difficult to discover, and once one is found, it can only be used for the same set of keys: Adding just an extra key will ruin everything.

A lot less rigid method can be obtained if we give up the one-to-one requirement of the relationship between the keys and table addresses and let H(KEY) *map two or more distinct keys to the same location* in HKEY. Of course, if more than one key is sent to the same node, a phenomenon termed a *collision* occurs, and we must invoke some *collision resolution policy* in order to tell the keys apart in the process of insertion or searching. Thus, we arrive at the *core concept behind hashing*. If the hash function H(KEY) is good enough to map only a few keys to any particular hash address H, in other words, spread the keys evenly throughout the table, we can adopt the following strategy:

1. Given a search KEY, use H(KEY) to *hash* KEY to some address H in the table.
2. If the address H is empty, the search is unsuccessful, since no key has ever hashed to H.
3. If the address is occupied, search all the keys that have hashed to H *sequentially*.

Thus, hashing is a typical *hybrid algorithm*: It combines direct addressing with *sequential search*, a method based on comparisons between keys. The *average* number of keys mapping to any hash node equals N_SMALL/HSIZE, i.e. the load factor. If the hash table is not full and the keys are spread uniformly, the average number of key comparisons required to find or reject a key is less than 1. Also, searching for a key should be the faster, the sparser the table is. So, to make a good practical use of a hash table, we ought to:

1. Choose a proper hash function H(KEY).
2. Find an efficient way of resolving collisions.

Before we could formulate the requirements for a *good hash function* let us consider how a *bad hash function* would behave. On one extreme, if a function is lightning fast but maps all keys or their majority to the same hash address, it defies the very purpose of distributing keys among different addresses: For then we would have to search all these keys sequentially! Hence, for a good hash function, it is paramount that it should map the keys evenly across all hash table nodes, without burdening some addresses with huge clusters of keys and leaving the rest of the slots empty. On the other extreme, if a function maps the keys extremely uniformly but takes an inordinate time to compute, it is of no good use, either - direct addressing itself would become a bottleneck. Now we can formulate some rational requirements a good hash function should satisfy:

1. Its computation should be as fast as possible.
2. It should distribute the keys uniformly to minimize collisions.
3. It must return addresses in the range from 0 to HSIZE-1.

There is a number of mapping methods conforming to these requirements [1]. We will discuss and use the simplest technique called the *division method*, which utilizes the remainder modulo:

```
H = MOD (KEY, HSIZE);
```

It certainly fits requirement 3, since for any value of KEY, this function always returns an integer in the range from 0 to HSIZE-1. It also satisfies requirement 1, for although it incorporates a division, its computation is still reasonably fast. However, to satisfy requirement 2, the value for HSIZE must be chosen rather carefully. The number theory tells us (see, e.g., [1]) that if HSIZE is a *prime number* and not too close to the power of 2, the MOD function tends to spread the keys uniformly across the nodes, with the majority of the occupied nodes receiving 1 to 3 keys. Let us see how this would work for our sample set of 10 keys. If we choose the "target" load factor as 0.625 and divide it into the number of keys, we obtain 16. The first prime number greater or equal to 16 is 17, so let us select HSIZE=17. (The actual load factor is now 10/17 = 0.588.) We may want, therefore, to allocate the table as

```
ARRAY HKEY (0:17) _TEMPORARY_;
```

To obtain a hash address, KEY is divided by HSIZE=17, and the remainder H is computed. H points to the H-th slot in the table where KEY must be inserted. Repeating this operation for every test key, we end up with the following pattern (the numbers atop the table indicate the corresponding array buckets, and the colliding keys are shown in boldface):

```
00    .   .   .
01  970   .   .
02  971   .   .
03    .   .   .
04  922   .   .
05  260 532   .
06    .   .   .
07    .   .   .
08    .   .   .
09  400   .   .
10    .   .   .
11    .   .   .
12    .   .   .
13    .   .   .
14    .   .   .
15  185   .   .
16  543 050 067
17    .   .   .
```

The keys 970, 971, 922, 400, and 185 all map to their slots in HKEY one-to-one. The keys 260 and 532 produce a single collision at the address 05, and the keys 543, 050, and 067 result in a double collision in the node 16. If this table is to be stored in memory and searched, the collisions at the locations 05 and 16 have to be *resolved*.

Before we move on, let us solve the small technical problem of finding the correct prime HSIZE, given the file SMALL and load factor LOAD. Instead of computing it by hand or from a table of primes, it can be calculated and stored in a macro variable LOAD dynamically using a short (and extremely fast) SAS program:

```
%let load = 0.8;
data _null_;
   do p=ceil(p/&load) by 1 until (j = up + 1);
      up = ceil(sqrt(p));
      do j=2 to up until (not mod(p,j)); end;
   end;
   call symput('hsize',left(put(p,best.)));
   stop;
   set small nobs=p;
run;
```

As we already know, selecting a decent hash function is just one part of the deal: No matter how good the function is, it is practically guaranteed that some keys will hash to the same addresses in the table, so we have to devise a method of resolving collisions. This is another point at which hashing radically deviates from key-indexing and bitmapping where we needed not store the keys in the table itself. With hashing, the keys themselves have to reside in the table, because they will have to be compared to a search key unless the search key hashes to an empty node. Various *collision resolution policies* differ in the ways by means of which colliding keys are stored, linked as "belonging" to the same hash address, and traversed. Let us consider them one at a time.

1. Separate Chaining

One way of resolving collisions suggests itself naturally once we cast a rapid glance at the distribution of our 10 keys among the addresses of the hash table shown in the previous section. Keys "attached" to each occupied address form visible "chains" - consisting of a single key in the absence of collisions. Making use of such chains to resolve collisions is logically called *separate chaining*.

There are two ways the chains of keys can be utilized in terms of the SAS DATA step. First, the keys comprising the chains could be stored *outside the table* by placing them in the occurrences of a two-dimensional array. A significant drawback of this method, however, is poor memory utilization. If we have 100,000 keys in SMALL and a single "bad" address colliding 10 keys, we will be forced to create a 2-dimensional array sized as (0:10, 0:100000) to resolve the collisions. Even with the load factor 1, it requires 10 times the memory the keys would occupy by themselves. On the positive side, the 2-dimensional chaining is quite fast, and it can work with load factors greater than 1, if necessary. So, if good memory utilization is not a paramount consideration, the method could be recommended. (Feel free to contact the author for the details of implementation.) What is more, because 2-dimensional separate chaining provides a natural way of working with long chains, it turns out to be extremely valuable when hashing is used for *external* searching, i.e. searching on disk rather than in high-speed memory. It will be mentioned once again in the section "Applications".

Returning to the main course of the paper, memory-resident hashing, the idea of chaining can be exploited in a much neater fashion than by using a huge, and mostly wasted, 2-dimensional array! Once the philosophy of allocating the main storage for colliding keys is changed from sequential to *linked* , we arrive at an extremely elegant collision resolution policy, both very fast and reasonably memory-efficient.

2. Coalesced Chaining

The *core idea* of this method is to place the chains of colliding keys into the hash table itself and combine the keys mapping to the same node in a linked list, with the head residing at the colliding address. Setting the last link of each chain to null designates the end of the chain, thus helping us tell where to stop when the list is traversed serially. Since the linked lists are thus allowed to overlap in the hash table sharing the same storage locations, this approach is termed *coalesced list chaining*, or, shorter*, coalesced chaining*. To make it possible, all we need is a numeric array of link items LINK, parallel to the "main" hash table where the keys are inserted. It is extremely important that in order for this method to work, *at least one entry in the table must be empty.* Otherwise if the table is full, there would be no empty node where a null link should point in order to terminate the loop traversing the list. Since a table allocated as (0:HSIZE) has HSIZE+1 entries, but the modulo-based hash function only addresses HSIZE nodes from 0 to HSIZE-1, this requirement will always be satisfied. Let us agree to *always leave the address 0 empty* by hashing keys as

```
MOD (KEY, HSIZE) + 1.
```

That is, if KEY modulo HSIZE is 05, it will map to the address 06. Adding a unity to the modulo has the additional advantage of allowing to use 0 as a null value for the end-of-chain. As stated above, we have to allocate two parallel arrays, one for the hash table itself and one to hold the links:

```
ARRAY HKEY (0:&HSIZE) _TEMPORARY_;
ARRAY LINK (0:&HSIZE) _TEMPORARY_;
```

Now we are ready to spell a detailed plan of loading a coalesced list hash table:

STEP 1. Set a counter variable R to the top address: R=HSIZE.
STEP 2. Hash: H=MOD(KEY,HSIZE) + 1.
STEP 3. If LINK(H)= ., the node is empty, no list is attached to it. Go to step 8 to insert the key.
STEP 4. Otherwise *traverse the chain* to find if the key is already in the table:
A. If KEY=HKEY(H) the key is duplicate. Get the next key and return to step 2.
B. Else If HKEY(H) is not 0, it is not the end of the list yet. Set H= HLINK(H) and repeat step 4.
STEP 5. Find an empty address closest to the top: Decrement R until LINK(R)= .
STEP 6. Store the key at this address: HKEY(R)=KEY.
STEP 7. Memorize where KEY actually belongs: LINK(H)=R; H=R.

STEP 8. Insert KEY into the address H and set its link to null: HKEY(H)=KEY; LINK(H)=0. Now the node has been marked as occupied.

Let us see, by inserting one key at a time, what kind of linked list hash table is actually created by this process for our 10 sample keys and HSIZE=17. Please refer to the resultant table, Table A1, in the Appendix. The key being inserted, as well as the colliding keys, are shown in boldface. The two bottom rows represent the final state of the loaded table. A peek at it quickly reveals how the collisions are being handled:

All the way up to the attempt to insert KEY=532, each key finds its unique slot without any contention. But at KEY=532, we have the first collision, because it hashes to the address 06, already occupied by the key 260. In accordance with the algorithm, we look at the link at address 06 and find it to be zero. Therefore, it is the end-of-chain - and the only key in the chain so far. The first available empty address counting from the top of the table (right to left on the diagram above) is 15. The new key 532 goes there, and the node is marked as occupied with 00 in its link field. To tell the key 260 where its successor in the chain, 532, resides, we store the address of 532, i.e. 15 in the link field of node 06 that holds 260.

The keys 543, 050, and 067, all hashing to the address 17, are placed in the table in the same manner. The first key in this chain, 543, must be stored at this address, and there it is. The link of the address 17 is not 0, hence, it is not the end of the list. Instead, LINK(17)=14. This is the node where the next key in the chain, 050, must reside, and it is there, indeed. But once again, the list must continue because the address 14 contains a non-zero link, LINK(14)=13. Finally, we find the key 067 in the node 13, and it is the last key colliding at the hash address 17, for LINK(04)=0.

As opposed to the colliding keys, the keys hashing to their addresses uniquely, bump in a zero link at once. For example, the key 922 hashes to the address 05, with LINK(05)=0. Now the reason of leaving the address 00 always empty should be transparent. We are using 0 to indicate the end of chain (null link), but actually a chain traversal terminates when a null, i.e. missing value, link field has been encountered. A zero in a link field will always lead to address 00, and since it is always missing, the traversal will inevitably stop.

At this point, it should be crystal clear how this linked table organization facilitates searching. Suppose that we need to look for KEY=051. It hashes to the address 01 where the link field LINK(01) is missing. That is, none of the keys in the table has ever hashed to this address, hence the key is not in the table. However, searching for KEY=047 that hashes to the address 14, we are in a different situation, because LINK(14)=13 is not null. Hence, some other keys in the table may have also hashed to this address, and so the entire chain must be examined for the presence of 047. Since the key 050 in the node 14 does not match the search key, we have to look at the next key in the chain located at the address 13 to which LINK(14) is pointing. The key 067 in the node 13 does not match the search key 047, either, and it is the end of the list since LINK(13)=0. This, finally, points to address 00, whose link is (always) null. Hence, 047 is not present in the table.

As an example of a successful search, let us try to find KEY=050. It hashes to the address 17 with the key 543, different from 050. But it is not the end of story: LINK(17) = 14 is not null telling us that the next comparison should be made with HKEY(14) = 050. At this point, the search key is found, the list need not be traversed any further, and the process of searching terminates successfully.

After this walk-through, it should not take a Certified SAS Programmer to schedule *hash searching:*

STEP 1. Hash: H = MOD(KEY,HSIZE) + 1.
STEP 2. If LINK(H) = . then search terminates unsuccessfully.
STEP 3. Traverse the list. If KEY = HKEY(H) then search terminates successfully.
STEP 4. Else examine LINK(H). If LINK(H) = 0 then KEY is not found. Stop.
STEP 5. Next link. Set H = LINK(H) and go to step 3.

Now we can finally give a solution to the matching problem using coalesced chain hashing. An additional array parallel to the hash table and links, HSAT, is used to pull the satellites from the lookup file SMALL. If we do not need S_SAT it may be omitted along with the corresponding instructions.

```
** Coalesced Linked List Chaining **;

data match (keep=key s_sat l_sat);
   array hkey (0:&hsize) _temporary_;
   array link (0:&hsize) _temporary_;
   array hsat (0:&hsize) _temporary_;
   ** load and link hash table using keys from SMALL;
   do until (eof1);
      set small end=eof1;
      h = mod(key,&hsize) + 1;
      found = 0;
      if link(h) > . then do;
         link traverse;
         if found then continue;
         do r=&hsize by -1 until (link(r) = .); end;
         link(h) = r;
         h       = r;
      end;
      link(h) = 0    ;
      hkey(h) = key  ;
      hsat(h) = s_sat;
   end;
   ** search table for key from LARGE, output matches;
   do until (eof2);
      set large end=eof2;
      found = 0;
      h = mod(key,&hsize) + 1;
      if link(h) > . then link traverse;
      if found then do;
         s_sat = hsat(h);
         output;
      end;
   end;
   stop;
   traverse: if key = hkey(h) then found = 1;
             else if link(h) ne 0 then do;
                h = link(h);
                go to traverse;
             end;
run;
```

Since the code intentionally parallels the algorithm above, you should not be surprised to find the GO TO instruction. Those believing that "GO TO" and "structured programming" cannot peacefully coexist, may prefer to rewrite the TRAVERSE block at the expense of an extra comparison at the bottom of the loop as

```
do until (found);
   if hkey(h) = key then found = 1;
   else if link(h) = 0 then leave;
   else h = link(h);
end;
```

Now that the coalesced chaining routine is ready, it can be tested using the same sample files as have been used for key-indexing. The program was tested for the load factors 0.5 and 0.8 (50 and 20 per cent sparse table). The results shown in Table 1 generally corroborate the conjectures made earlier:

1. The sparser the table, the faster the search, but it consumes proportionally more memory.
2. If the hash table is relatively sparse, its lookup time does not depend on the number of keys in the table.
3. It runs somewhat slower than key-indexing, but still 2 to 3 times faster than even SQLXJHSH and is much easier on memory than the rest of the methods tested.
4. Just like with key-indexing and bitmapping, hashing needs neither sorting nor removing duplicates.

Judging from the test results (see the "Benchmarking" section below), chaining performs very well, with the added benefit of not being too sensitive to the sparsity of the table (load factor). However, it requires an extra array to hold the links. The link array is as large as the hash table itself, so if SMALL is actually not quite small, the additional memory burden can be significant. In a different class of collision resolution policies collectively called *open addressing,* memory utilization is improved by doing away with the links altogether. We shall discuss two such methods: Linear probing and double hashing.

3. Open Addressing with Linear Probing

The main idea behind open addressing can be described as follows. Just like in the case of coalesced chaining, keys are stored in the hash table itself. Suppose we have a key KEY to be loaded in the table. First, let us hash it using the division method, but straight, i.e. without adding a unity:

```
H = MOD(KEY, &HSIZE);
```

If H points to an empty slot, we simply store the key at this location. If the slot H is occupied, we have a collision. Let us compare the key with the one already sitting at H. If the keys are equal, the current key should be discarded because it is a duplicate, and the next key obtained from the input. Otherwise we have to find a different slot for the current KEY. Let us step down the table one or more times one node at a time. If H becomes less than 0, i.e. we have stepped off the bottom of the table, let us return to its top, and continue to do so in this wrap-around cycle until having encountered either a duplicate key - in which case we just stop and get the next key, or an empty node - in which case we insert the colliding key into it. The method of resolving collisions just described is called linear probing – for the table is being "probed" using a fixed *probe decrement*, C=1, regardless of the key.

Since we have HSIZE+1 nodes in the table, but can only address HSIZE nodes from 0 to HSIZE-1, at least 1 location in the table, the top one, will always remain empty, thus preventing the loop from iterating infinitely. Let us observe how this process works step by step while our 10 test keys are being inserted in the table sized as [00:17] (See the dynamic table Table A2 in the Appemdix):

All the keys up to and including 543, hash uniquely to their very own nodes. However the next key, 532, hashes to the same address 05 as the key 260, already sitting there. According to the plan outlined above, we step down the table until an empty slot is found. This happens at H=03, and so 532 is inserted at this address. The next key, 050, is not too friendly, either, since it claims the same seat, H=16, that is already assigned to 543. The nearest free slot down the table is H=14, and so that is where 050 goes. But the next key, 067, happens to hash to the same H=16 again! Now, to find where to place this one, we have to travel all the way to H=13, at which point the hash table is loaded and ready to be searched.

As in the case with chaining, the process of loading the table readily suggests the way of looking it up. As an example of an unsuccessful search, let us look for KEY=51. MOD(51,17) yields 0, and address 00 is empty. Hence, 51 is not in the table, period. Searching for KEY=66 is more complex, since it hashes to H=15 occupied by the key

185. Since there is no match, we look at the next key, 050, one node down the table, find a mismatch again, and proceed in this manner all the way to H=12, which is empty. It means that 66 is not in the table, either, for if it had been inserted in the table, it would have been found before an empty node is hit.

As an example of a successful search, let us look for KEY=922. It hashes to H=04, and we have an immediate match. Another successful search for KEY=067 is a bit more laborious, for it hashes to H=15, and we have to step down the table twice until the key is identified in H=13.

From these simple examples, it should be clear how linear probing reduces the number of probes sequential search would require. In the worst case scenario in the example above, linear probing examines 5 keys until it either finds or rejects a search key; but on the average, with the load factor 10/17, the number of comparisons will be close to 2 per search, hit or miss. Sequential search, on the other hand, would require, on the average, 8 probes for a hit and 17 for a miss. Now it is time to translate all these verbal speculations into SAS:

```
** Hashing by Open Addressing with Linear Probing **;

data match (keep=key s_sat l_sat);
   array hkey (0:&hsize) _temporary_;
   array hsat (0:&hsize) _temporary_;
   ** load table with keys from SMALL;
   do until (eof1);
      set small end=eof1;
      do h=mod(key,&hsize) by -1 until (hkey(h)=. or hkey(h)=key);
         if h < 0 then h = &hsize-1;
      end;
      hkey(h) = key  ;
      hsat(h) = s_sat;
   end;

   ** search table for each key from LARGE and output matches;
   do until (eof2);
      set large end=eof2;
      do h=mod(key,&hsize) by -1 until (hkey(h) = .);
         if h < 0 then h = &hsize-1;
         if hkey(h) = key then do;
            s_sat = hsat(h);
            output;
            leave;
         end;
      end;
   end;
   stop;
run;
```

The main advantage of this scheme, as it is evident from the code above, is its profound simplicity. In fact, none of existing hashing methods is simpler or more straightforward than the linear probing. And, if the table is sparse enough, it performs quite well, too! As a rule of thumb, the linear probing will do the hashing job just right if about half of all nodes in the table are left empty, i.e. with the load factor of about 0.5. However, as the table gets fuller, its performance deteriorates the quicker, the fuller the table is. With load factors above 0.9, the only good things we can say about the linear probing is that it is simple and it works, albeit slowly but surely.

The reason why linear probing exhibits such a behavior in a crowded table lies in the phenomenon called *primary clustering*. When looking for an unoccupied node for a colliding key, we fill out the very first empty location we come across. Therefore the groups of adjacent occupied addresses tend to aggregate, forming clusters of keys. Worse still, the clusters can bridge together forming bigger clusters. (For instance, consider what happened to the clusters 970, 971 and 922, 260 in our test table above.) Hence, if the table is not quite sparse, we will eventually have to travel through almost the entire table before finding an empty location to either insert a key or stop the loop in the case of an unsuccessful search.

One apparent way to alleviate the problem of primary clustering is to try stepping through the table using more than one node at a time. It turns out to be a very good and sound idea. Complemented with another good and sound idea, it leads to the open addressing method called *double hashing* that eliminates primary clustering entirely. Therefore, it would allow achieving the same speed of search with a less sparse table resulting in a superior memory utilization.

3. Open Addressing with Double Hashing

So, as suggested above, let us try stepping down the table using some probe decrement C > 1. However, the value of C must be chosen rather carefully. With linear probing, it is guaranteed by virtue of C=1 that in the wraparound process of probing the table, each node can be examined, and examined exactly once. What kind of value should C > 1 have to retain the same fundamental property? It follows from the number theory that if the probe decrement C and the hash table size HSIZE are *relatively prime*, this property holds. Now remember, we have chosen the table size prime in order to minimize the collisions. Therefore, selecting C as any integer between 1 and HSIZE-1 inclusively will make C and HSIZE relatively prime.

However, there is one more important consideration helping choose C even wiser. Namely, if we could make C depend on the key in a random yet deterministic manner, it would help spread diversify the probing sequences belonging to different keys, and hence distribute the keys even more evenly in the table. MOD function, as we know, possesses quite good randomizing capabilities (which is why it is used as a hash function in the first place). Therefore, if we compute C as

```
C = 1 + MOD(KEY, HSIZE-2),
```

it will both distribute the values of C among the keys pseudo-randomly and guarantee that any C value obtained this way and HSIZE are relatively prime. Indeed, C can result in nothing else but some integer between 1 and HSIZE-2, and since HSIZE is prime, C and HSIZE will always by relatively prime. In practice, such a choice for C has been proven to work satisfactorily in most cases.

In essence, what we are doing is hashing the key the second time to obtain the probe decrement, which is why this method of resolving collisions is called double hashing. Of course, the second hashing is an extra computation, but it is not too expensive, and it is situated outside the inner loop of the routine. Therefore, we should not expect a lot if computational overhead, all the more that eliminating primary clustering turns out to be much more important from the standpoint of performance.

With the exception of C > 1, the basic linear probing algorithm remains intact. Like before, if in the process of decrementing H, it is found that H < 0, that is, we have fallen off the bottom of the table, we wrap around it; only in this case, instead of returning right to &HSIZE-1, we shall return to H+&HSIZE. For example, if HSIZE=17, C=5, and we have found that H=-2, we shall wrap around the table to $-2+17=15^{th}$ array item. Let us see, using the set of our experimental keys, what kind of hash table this process will compile, starting with an empty table and inserting one key at a time. The dynamic table A3 created by this process is shown in the Appendix.

Comparing the final state of the table with that compiled by linear probing, we clearly see that it is much more uniform, with the clusters of keys well separated from each other, and with no cluster containing more than 3 keys. It means that no matter what key we are looking for, no search will require more than 3 comparisons between keys in the worst case scenario.

While theoretically, double hashing is significantly more involved that linear probing, amending the program for linear probing in order to accommodate double hashing boils down to a single line of code preceding the main hash, and a subtle change in the way to wrap around (below, all the changes to the linear probing routine are shown in upper case):

```
** Open Addressing with Double Hashing **;

data match (keep=key s_sat l_sat);
   array hkey (0:&hsize) _temporary_;
   array hsat (0:&hsize) _temporary_;
   ** load table with keys from SMALL;
   do until (eof1);
      set small end=eof1;
      C = 1 + MOD(KEY,&HSIZE-2);
      do h=mod(key,&hsize) by -C until (hkey(h)=. or hkey(h)=key);
         if h < 0 then H ++ &HSIZE;
      end;
      hkey(h) = key  ;
      hsat(h) = s_sat;
   end;
   ** search table for each key from LARGE and output matches;
   do until (eof2);
      set large end=eof2;
      C = 1 + MOD(KEY,&HSIZE-2);
      do h=mod(key,&hsize) by -C until (hkey(h) = .);
         if h < 0 then H ++ &HSIZE;
         if hkey(h) = key then do;
            s_sat = hsat(h);

            output;
            leave;
          end;
      end;
   end;
   stop;
run;
```

Let us take a look at the performance Table 1. With a 50 per cent sparse table, double hashing runs just a tad slower than coalesced chaining with 20 percent sparsity, but on the positive side, it uses less memory. So, double hashing is quite fast; it even loads an equally sparse table somewhat faster than the chaining because it does not have to worry about the links. The fact that a searching method based on stepping through the table before an empty node is found works so well, may seem surprising. However, this is a direct result of the double hashing probing methodology. In fact, independent experiments (corroborating theoretical conclusions) show that if the table is no more than half full, double hashing makes on the average no more than 2 comparisons per miss, and no more than 1.3 comparisons per hit.

IV. HASHING WITH NON-NATURAL KEYS

As the test results show, hashing performs admirably by any account regardless of the collision resolution policy being used. However, even though hashing schemes we have discussed impose no limitations on the range of keys, they have been developed under the assumption that the keys are integers. Now it is time to remove this restriction as well.

The reason it is possible to do is rooted in the fact that in its final stage, hashing is strictly comparison-based, which effectively renders the nature of keys non-critical. Both hashes and traversals are used merely to minimize the number of comparisons necessary to carry out a search, yet the final hit-or-miss decision - if a hash address is not empty - is made by comparing some keys in the table to the search key. Therefore, in order to be able to operate on keys of any type, we only have to figure out how to

hash a key if it is not a non-negative integer. For the hash function to remain uniform and fast, it is critical to adhere to a few simple rules:

- Hashing process should involve as many key characters as possible.
- String operations and conversions must be minimized.

Let us consider a number of distinct practical situations.

1. Fractional Signed Keys

In this case, we can simply rescale each key before hashing by multiplying it by a suitable integer constant and adding another constant to the result if necessary. For instance, if our keys are in the decimal form X.Y, multiplying each key by Y would suffice. If, in addition, they can be negative, we would simply add an integer Z known to exceed the largest absolute value a negative key can assume. So, the entire change to the programs above needed to accommodate fractional signed keys would be using

```
MOD ( KEY*Y + Z, HSIZE)
```

in the hashing formulae instead of the straight modulo. It will not cause any noticeable deterioration in performance, since in SAS this kind of computation is quite fast.

2. Digital Strings

First of all, since digital strings are character variables (consisting of digits only), the hash table itself will have to be declared as a character array of appropriate expression length, for example:

```
ARRAY HKEY(0:&HSIZE) $12 _TEMPORARY_;
```

Hashing a digital string is a simple matter of using the INPUT function and an appropriate numeric informat. For example, if the keys were 16-digit account numbers stored in a character variable, we could simply choose

```
MOD (INPUT(KEY,16.), HSIZE)
```

as our hash function. Another way to hash a digital string is to apply the same methods that are used for hashing character variables in general (see below).

3. Generic Character Keys

Numerous techniques have been developed to hash arbitrary character keys well [2, 3, 4]. Almost all of them are based on breaking a character key apart and then involving the individual bytes into a sort of computation resulting in an integer in the range [0:HSIZE-1]. Some of these methods, for instance, universal hashing, actually guarantee to hash *any* input evenly. However, they are based on the assumption that the process of extracting individual bytes from a string is very fast. Unfortunately, this is exactly what is slow in SAS. We would be much better off converting a character string to an integer in a single shot, and PIBw. informat is just the tool:

```
MOD (INPUT(LEFT(KEY),PIBw.).
```

Generally, the wider is the informat width, the better, because the wider it is, the more key information is involved in the hashing process. However, selecting the informat too wide may result in a large integer rendering the result produced by MOD function incorrect. Experimentally, it has been found that under NT, the maximum allowable width, 8, works fine. Under OS/390, it should not exceed 7, and under HP-UNIX, 6 is the limit. The method has an extra advantage of avoiding the slow SUBSTR function, for it automatically chops the number of characters from the beginning of KEY equal to the informat width. Note that we use PIBw. instead of S370FPIBw.. First, it is faster. Secondly, with hashing, the order of bytes does not matter: We only want to use as many key bytes as possible to minimize collisions. The LEFT function may help by squeezing leading blanks to the right. If a key is longer than the practical informat width, the trick still works, provided that the input characters distinguish the keys well. However, if they have a good chance of being identical, they can be selected from a different portion of the key.

4. Composite Keys

This situation arises quite often. A natural inclination is to concatenate the components and hash the result. Principally, there is nothing wrong about it; however, there are two pitfalls. First, in the context of hashing, where computing a hash function fast is paramount, concatenation is slow. Second, the components may concatenate into an integer lying beyond SAS integer precision. Third, too large a value can cause the MOD function to return a no-sense result, for instance, a remainder greater than the divisor.

Consider a (real-life) situation when records are uniquely identified by two numeric variables, a 16-digit ID and 9-digit MEM, while particular ID can point to multiple accounts. Concatenating the keys as ID || MEM and hashing the result would have the effect of scrambling the entire MEM. All keys with the same ID would then hash to the same address regardless of MEM and lead to multiple collisions and horrible performance. Luckily, it can be avoided since we are not interested in the value of the key itself, but only in its remainder modulo HSIZE. Hence, Horner's algorithm can be used to hash the components separately and then combine the results in the final address. The outcome is the same as if we had enough integer precision to store the combined key accurately. For the ID and MEM, it means that the hash function can be computed in the form:

```
MOD(MOD(ID,HSIZE)*1E9 + MEM, HSIZE) .
```

If the partial keys are longer or the range is wider, they can be split further, and Horner's rule can be applied to the components once again. Of course, in order for this method to work, the parts of the key must be kept in parallel hash arrays, and loaded and tested separately. If, for instance, ID and MEM were hashed by chaining, the HKEY declaration would have to be replaced with

```
ARRAY HID  (0:&HSIZE) _TEMPORARY_;
ARRAY HMEM (0:&HSIZE) _TEMPORARY_;
```

The instruction

```
HKEY(H)=KEY
```

would become

```
HID  (H) = ID    ;
HMEM (H) = MEMNO ;
```

Also, in the TRAVERSE subroutine, the instruction

```
IF KEY=HKEY(H)
```

would transform into the following:

```
IF ID = HID(H) AND MEMNO = HMEM(H);
```

Similar modifications could be done if the open addressing methods were used.

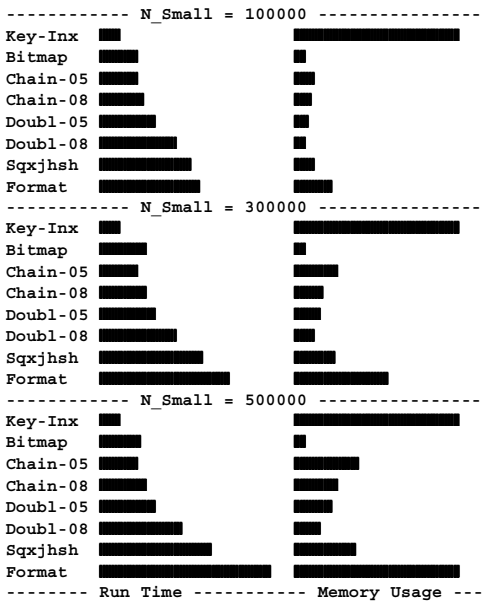
V. BENCHMARKING

Each technique presented above operates best in its own "area of expertise" defined by the number of lookup keys and key range. To compare them to each other and two SAS-supplied methods, SMALL and LARGE were created with random integer keys in [0:8E6] range where all methods could work within the system imposed memory limit of 70 MB. To include bitmapping into the comparison group, the satellite S_SAT was omitted from SMALL. The input was prepared in such a way that hits and misses were equally likely to occur. LARGE with fixed N_LARGE=2E6 was then matched against SMALL with varying number of records in batch on S/390 G5 R36 Enterprise Server running SAS Version 6.09E. For key-indexing, bitmapping, and hashing LOAD represents the time needed to load a table from SMALL (<EOF1> loop). In the case of formatting, LOAD is the time required to unduplicate SMALL and compile the format. For MERGE, it is the time needed to sort the files. The value LF= is the load factor used for the run. LOAD, SEARCH, and TOTAL are given in CPU seconds, MEMORY – in kilobytes.

Table 1. Benchmarking.

N_Small	Method	Load	Search	Run	Memory
100,000	Key-Inx	0.42	12.18	12.60	65261
	Bitmap	0.36	22.64	23.00	3925
	Chain-05	0.31	21.78	22.09	5997
	Chain-08	0.34	26.28	26.62	4829
	Doubl-05	0.28	32.78	33.06	4445
	Doubl-08	0.33	47.91	48.24	3961
	Sqxjhsh	0.00	52.66	52.66	5881
	Format	6.27	51.92	58.19	10866
	Merge	19.74	46.16	65.90	3276
300,000	Key-Inx	0.67	12.07	12.74	65261
	Bitmap	0.99	26.17	27.16	3925
	Chain-05	0.87	21.47	22.34	12093
	Chain-08	0.95	26.03	26.98	8637
	Doubl-05	0.80	31.24	32.04	7493
	Doubl-08	0.97	47.03	48.00	5769
	Sqxjhsh	0.00	58.38	58.38	11401
	Format	18.71	55.19	73.90	26199
	Merge	20.91	46.63	67.54	3267
500,000	Key-Inx	0.92	12.09	13.01	65261
	Bitmap	1.59	26.56	28.15	3925
	Chain-05	1.37	21.60	22.97	18033
	Chain-08	1.53	25.30	26.83	12357
	Doubl-05	1.29	31.90	33.19	10465
	Doubl-08	1.57	42.17	43.74	7625
	Sqxjhsh	0.00	64.49	64.49	16921
	Format	29.77	67.26	97.03	57289
	Merge	21.59	47.27	68.85	3267

The same benchmarking information might be digested better if presented in a more visual form. On the chart below, the left half of bars represents relative run-times, and the right half shows relative memory utilization.



Note that the run times exhibited by key-indexing, bitmapping, and hashing, in agreement with their direct-addressing nature, are virtually independent from the number of lookup keys. For key-indexing and bitmapping, memory usage is always fixed since the number of keys has no effect on the universe of keys they embrace. Hashing uses memory strictly proportional to the number of keys in the table and sparsity of the table.

VI. APPLICATIONS

1. Subsetting

Subsetting used above as a sample problem is an important but only one of many tasks to which direct-addressing based methods can be applied successfully. However, before discussing other applications, we have to make a few final observations about subsetting, all the more that it has been used as our proving grounds. From the test results, it follows that when it comes to one-time subsetting, direct-addressing methods result in lookup speeds unmatched even by methods written in the underlying software and specifically designed for searching. As an icing on the cake, hashing is significantly more memory-efficient than formatting and SQL. The latter is extremely important when the number of keys in SMALL grows beyond a couple of million. Hash memory is strictly proportional to the number of lookup keys and can be accurately estimated beforehand. Contrary to that, the amount of memory used by formats or SQL seems to grow uncontrollably after a certain threshold has been reached.

On a different note, we should exercise caution dragging satellites from SMALL through the memory. If there is more than one satellite, one may be tempted to create a separate parallel satellite array for each, but this is not always the right thing to do. Remember, character temporary SAS arrays are allocated in 8-byte multiples per item (unless you are running V8.1). If we have four 8-byte character satellites, a separate array can be declared as $8 for each with 100 per cent memory utilization. However, if we have four 2-byte satellites and create 4 parallel arrays $2 each, it will waste gobs of memory, for SAS will allocate the arrays with 8 bytes per item, anyway. So, in this case, we will be much better off memory-wise allocating one array as $8, stringing the satellites together in the load phase, and unstringing them into separate variables just prior to outputting a record.

2. Dynamic DATA Step Data Dictionaries

Let us take a look at key-indexing and hashing from a different, more philosophical, standpoint. The key-indexed and hash tables we have used to facilitate direct address and hybrid searching can be viewed as *some abstract data type* (*ADT*) in memory, that allows to efficiently perform certain operations on its *entries*. The ADT used in key-indexing and hashing is simply called *a table*. The entries contain keys and maybe some satellite information. There are two operations we have learned how to perform in the process of solving our sample problem: *Insert and search*. Many kinds of ADTs other than a hash and key-indexed table can facilitate these operations. A simple sequentially searched array, binary searched sorted array, AVL tree are just a few ADT examples.

The difference between various ADTs lies in the time necessary to insert an entry or search the entries given a key. For example, a plain array requires O(1), i.e. constant time, independent from the number of entries N, to insert a new entry - we simply append it to the right. However, searching such a structure occurs in O(N) time, i.e. proportional to N. If the ADT is a sorted array, we need O(N) time to insert an entry because it is necessary to shift a number of items proportional to N to free a node for the new key keeping the table sorted. In exchange, searching an ordered array, as we know from Part 1, occurs only in O(log(N)), or even O(loglog(N)) time. Yet another ADT, an AVL tree, facilitates both operations in O(log(N)) time as its worst case.

From these examples, it is clear what kind of advantage key-indexing and hashing offer: If a hash table is sparse enough, they support both insert and search operations in constant time O(1), because, as we have seen before, it takes practically the same time to search the table or to insert a new key, no matter how many keys the table may contain.

As a side note, from this standpoint, the difference between key-indexing and hashing is merely superficial. A key-indexed table is, in effect, nothing else but an infinitely sparse hash table, and the hash function used to access it is simply constant.

The fact that hashing supports searching (and thus retrieval and update) in constant time makes it ideal for implementing DATA step dynamic data dictionaries. Imagine that in the course of DATA step processing, we need to memorize certain key elements and their attributes as we go, and at different points in the program, ask and answer questions like the following:

1. Has the current key already been used before?
2. If it is new, how to insert it in the table, along with its attribute, in such a way that the question 1 could be answered as fast as possible in the future?
3. How to access a key element in the most speedy fashion and update its satellite datum?
4. If the key is no longer needed, how to delete it?

If the "key element" satisfies the conditions making key-indexing applicable (for instance, it is a SAS date), there is no better tool for the job. All the actions are performed in O(1) time and do not get any simpler:

1. See if the node whose value equals KEY contains a missing value.
2. Fill the node with the attribute.
3. Overwrite the attribute already in the node.
4. Move a missing value to the node.

If the keys are not limited-range integers, we will have to organize a hash table using either of the collision resolution policies given in the text. In both programs, the body of the first DO UNTIL(EOF) loop constitutes nothing else but a ready-to-go combined hash search-and-insertion. That answers questions 1 and 2, or 1 and 3. The second DO UNTIL(EOF) loop is a pure hash search, and answers question 1 itself.

A practical application of these principles immediately coming to mind is obtaining frequency counts in the case of a huge number of distinct levels of a categorical variable, when FREQ or SUMMARY either run out of memory or take too long to run. To compute frequencies without sorting, we must be able to maintain a table in memory allowing to immediately locate the value coming with the next record and add a unity to its count.

The following question was asked in SAS-L: "I have an unsorted SAS data set with almost 100 million records. It has a numeric variable FLDR_ID that can be any integer number from -500,000 to +500,000. How to create a file with frequencies, cumulative frequencies, percents and cumulative percents for all values of FLDR_ID having only 50 MB of RAM?" The problem with the "standard" approaches (FREQ or SUMMARY) is that there are too many discrete values of the categorical variable, and both procedures, if applied "head-on", either run out of memory or seem to run endlessly. From the standpoint of direct addressing, the key FLDR_ID is a restricted-range integer, and therefore for the purpose of the data dictionary, key-indexing should be here right at home. This was realized by Ian Whitlock and the author:

```
data freq (keep=fldr_id freq cfreq pcnt cpcnt);
   array f (-500001:500000) _temporary_;
   do until(end);
      set ids end=end nobs=nobs;
      if fldr_id = . then fldr_id = lbound(f);
      f(fldr_id) ++ 1;
   end;
   ptot = 1/nobs * 100;
   do i=lbound(f) to hbound(f);
      if f(i) = . then continue;
      freq =  f(i);
      cfreq ++ freq;
      pcnt  = freq * ptot;
      cpcnt ++ pcnt;
      if i > lbound(f) then fldr_id = i;
      else fldr_id = .;
      output;
   end;
run;
```

The program uses 12 MB of memory and runs an order of magnitude faster than either FREQ or SUMMARY (provided that they do not run out of memory in the process).

3. Stable Sortless Unduplication

While discussing hashing, we saw that as an attempt is made to load the next key into a hash table, the search-and-insert subroutine first determines whether the key has already been inserted, and if it has, goes to the next record. As this occurs very fast, the search-and-insert subroutine can be successfully used to remove duplicates from a file without sorting.

Speaking of the latter, for a SAS programmer, "duplicate removal" almost instantly rings "PROC SORT NODUPKEY" or "SELECT DISTINCT", depending on the prior exposure and taste preferences. It is an interesting phenomenon. We have, in effect, accustomed to using the side effects of two very time-consuming procedures just to kick out records

with repeating keys. Of course, in the situation when a file has to be both sorted and unduplicated, PROC SORT is just the tool for the job. However, if sorting is not needed, a lot of extra work is done for no reason. What is more, consider a situation when not only we need to delete the duplicates from a file, but also retain the original order of its records, in other words, unduplicate the file in a *stable* manner. Should we decide to sort with NODUPKEY, we would be looking at at least 3 steps:

1. Add a sequence variable, say SEQ, to the file.
2. Sort the file with NODUPKEY EQUALS options by the key.
3. Re-sort the file by SEQ, and drop SEQ from the output.

Not only it does not look efficient, it does not make a whole lot of sense. Imagine that we have to remove duplicate cards from a deck; would we sort the deck first? Probably not! We would most likely take the cards off the deck one by one and memorize which cards have been taken out so far. If a card is "new", it goes face up to the output deck; if it is "old", it goes to the waste basket. At the end, the output will contain no duplicates and have the same relative order as input. All along in this process, we are using our human memory to keep track of the "keys" having been already used. Getting back to real files, a direct-address-based dictionary table can play the same role, providing both the quickest way to memorize "used" keys and establish whether the current key has already been used. Of course, the table must have a sufficient memory capacity, so we have to exercise a good judgement choosing between key-indexing, bitmapping, or hashing.

As an example, let us consider unduplicating a file similar to SMALL (how "small", depends on the range of keys and number of records) having 1,000,000 records, say. Assume that KEY has 16 digits, so neither key-indexing nor bitmapping can be used. However, a 50% sparse open-addressed hash table can be deployed at the expense of about 30 MB of memory. (It is not a small change, but with "usual" PC memories steadily creeping towards 1 GB, such memory usage can be considered tolerable.) Moreover, with 50% of nodes guaranteed to be empty, we can use linear probing, the simplest collision resolution method, with great deal of confidence. The plan (paralleling the playing card analogy above) is plain:

Step 1. Read a record from SMALL.
Step 2. Search for the key associated with the record in the hash table.
Step 3. If the key is found, it is a duplicate. Go to step 1.
Step 4. Otherwise insert the key in the table, output the record and go to step 1.

In the language of the SAS DATA step, it does not get any simpler, either:

```
** Sortless Stable Unduplication with Linear Probing;

data nodup (keep=key l_sat);
   array hkey (0:2000003) _temporary_;
   set large;
   do h=mod(key,2000003) by -1 until (hkey(h) = .);
      if h < 0 then h = 2000003;
      if hkey(h) = key then delete;
   end;
   hkey(h) = key;
run;
```

That is all it takes. Of course, the number 2000003 is not just arbitrary – it is the first prime number greater than 2000000, the "target" hash table size. But what about performance? On the same real computer the rest of the tests for this paper has been done, this step finishes the task in 3.1 CPU seconds. This compares quite favorably with PROC SORT EQUALS NODUPKEY (4.3 CPU seconds, and of course more for two extra steps if the stable output is required), and SQL with DISTINCT (11.2 CPU seconds).

4. Other Applications

It is impossible to embrace all conceivable applications of direct addressing methodology in one paper, so let us superficially mention just two more directions.

The author has participated in a "fuzzy matching" project, where the records from multi-million files with insufficient and redundant key information had to be linked using probabilistic matching. The linkage was essentially done in two stages. The first stage, using multiple composite redundant keys, identified probable matches, which were then scored pair-wise in the second stage. In both stages, key-indexing and hashing techniques were used to boost performance. They successfully supplanted "large" formats and SAS indexes, and as a result, the matching process was able to finish in about 1/5 of the original run-time on the same UNIX server where the original programs were run.

In this paper, only memory-resident direct-addressing methods have been considered. But what if we have so many distinct keys that none of the methods above will work just because of sheer memory limitations? Is it possible to apply the direct-addressing techniques, working so well in the high-speed memory, to some form of disk searching? The answer to this question is "yes". In fact, using a hybrid disk/memory hashing methodology, a plain SAS data set can be organized in such a way that the speed of accessing it randomly will exceed that of SAS index several times. Moreover, because of the intrinsic properties of hashing, the performance of such a lookup table does not depend on the distribution of the search keys. However, it is a topic for another paper.

CONCLUSION

Key-indexing is an in-memory lookup technique based strictly on direct addressing into an array with no comparisons between keys made. Its area of applicability is limited to integer keys falling in a limited range defined by available memory resources. However, when applicable, key-indexed search exhibits unmatched performance, and is the most straightforward way of implementing an ADT where all operations, such as search, insert, retrieve, update, delete, and enumerate are done in constant, O(1), time.

Bitmapping does not deviate a bit from the key-indexing philosophy, but uses available memory resources smarter by indexing keys directly into the bits, rather than 8-byte elements, of a numeric array. This way, bitmapping can address a much larger universe of integer keys than pure key-indexing. Both techniques have the advantage of working very fast with unlimited number of keys falling into their workable range. For instance, for keys restricted to 8 digits, up to 100 million integer keys can be in effect "stored" and subsequently extremely rapidly searched in a bitmap occupying only about 12 MB of real storage (RAM).

Hashing helps direct addressing work on keys of any type and range by bringing serial search and collision resolution policies into the equation. A bit slower that pure direct addressing, hashing searches times faster than SAS formats and SQL, and uses significantly less memory. Massive data processing applications like a data warehouse or production list management system are examples of the fields where the unmatched speed and efficiency of direct-addressing methods can be utilized. Compared to "traditional" techniques, they can successfully supplant formats and SQL in eliminating costly table joins, and tremendously accelerate the processes of data extraction, scrubbing, and validation, based on a large predetermined set of keys. The larger the data, the bigger advantage direct addressing can offer. Finally, direct-addressing searching methods are just additional, free programming tools, and can be used by any SAS programmer interested in efficiency and performance.

Key-indexing, bitmapping, and hashing are cool. They allow operating in the niches where "standard" approaches may run out of memory or take a frustrating time to run. The author encourages other SAS users to use these tools, modify them, tweak them, improve the code, and discover new areas of application. Karsten M. Self wrote once after having tried hashing in a real-world application: "Hash rocks, Dude!" Needless to say, the author eagerly agrees.

REFERENCES

1. D. E.Knuth, *The Art of Computer Programming*, **2**.
2. D. E.Knuth, *The Art of Computer Programming*, **3**.
3. R. Sedgewick, Algorithms in C, Parts 1-4.
4. T. A. Standish. Data Structures, Algorithms and Software Principles in C.

ACKNOWLEDGEMENTS

Thanks to Karsten M. Self, Ian Whitlock, F. Joseph Kelley, Sigurd Hermansen, and Base SAS R&D team for their enthusiastic support of direct-addressing methods in SAS, valuable discussions full of ideas, wit, and vigor, and giving the author an opportunity to apply the techniques to solve practical problems. The author gratefully acknowledges the contribution of the individuals who have, directly or indirectly, encouraged the author and supported his efforts of making direct addressing an accepted and practically used DATA step philosophy:

Michael V. Dorfman	Paul Gorell	David Pider
Eugenia P. Kravchenko	Steven Kleiman	Robert Workman
Doris H. Bogar	Alex V. Martchenko	David Cassell
Victor P. Dorfman	Thomas Mendicino	Jim Groeneveld
Vera Voloshin	Alex L. Voloshin	Diana Noble
Koen Vyverman	Vladimir A. Kirillov	Gerard Pauline
Benjamin Guralnik	Yuri Katsnelson	Ray Pass
Gennady Taratut	Michael A. Raithel	Art Carpenter
Jane King	Michael Rhoads	Shiling Zhang
Jay Melesky	Dianne Rhodes	Ronald J. Fehd
Bob Abelson	Don Stanley	Thomas Zicafoose
Ashiru Babatunde	Mark Terjeson	Christoph Edel
Peter Crawford	William W. Viergever	John Whittington
Colin Earle	Gregg Snell	Paul Kent
Peter Lund	Rick Aster	Kathy Y. Knorozova
Viacheslav V. Tsiolko	Igor A. Soloshenko	Michael M. Begun

AUTHOR CONTACT INFORMATION

Paul M. Dorfman
10023 Belle Rive Blvd. 817, Jacksonville, FL 32256
(904) 564-1931 (h) / (904) 954-8533 (o)
sashole@bellsouth.net
paul.dorfman@citicorp.com
paul_dorfman@hotmail.com

APPENDIX

Table A1. Inserting the sample keys into a hash table with collision resolution using coalesced linked list chaining.

H -->	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	
HKEY	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	**185**	.	KEY=185
LINK	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	**00**	.	
HKEY	.	.	.	**971**	.	.	.	.	.	.	.	.	.	.	.	.	185	.	KEY=971
LINK	.	.	.	**00**	.	.	.	.	.	.	.	.	.	.	.	.	00	.	
HKEY	.	.	.	971	.	.	.	.	.	**400**	.	.	.	.	.	.	185	.	KEY=400
LINK	.	.	.	00	.	.	.	.	.	**00**	.	.	.	.	.	.	00	.	
HKEY	.	.	.	971	.	.	**260**	.	.	.	400	.	.	.	.	.	185	.	KEY=260
LINK	.	.	.	00	.	.	**00**	.	.	.	00	.	.	.	.	.	00	.	
HKEY	.	.	.	971	.	**922**	260	.	.	.	400	.	.	.	.	.	185	.	KEY=922
LINK	.	.	.	00	.	**00**	00	.	.	.	00	.	.	.	.	.	00	.	
HKEY	.	.	**970**	971	.	922	260	.	.	.	400	.	.	.	.	.	185	.	KEY=970
LINK	.	.	**00**	00	.	00	00	.	.	.	00	.	.	.	.	.	00	.	
HKEY	.	.	970	971	.	922	260	.	.	.	400	.	.	.	.	.	185	**543**	KEY=543
LINK	.	.	00	00	.	00	00	.	.	.	00	.	.	.	.	.	00	**00**	
HKEY	.	.	970	971	.	922	**260**	.	.	.	400	.	.	.	.	**532**	185	543	KEY=532
LINK	.	.	00	00	.	00	**15**	.	.	.	00	.	.	.	.	**00**	00	00	Collision @ 06
HKEY	.	.	970	971	.	922	260	.	.	.	400	.	.	.	**050**	532	185	**543**	KEY=050
LINK	.	.	00	00	.	00	15	.	.	.	00	.	.	.	**00**	00	00	**14**	Collision @ 17
HKEY	.	.	970	971	.	922	260	.	.	.	400	.	.	**067**	**050**	532	185	**543**	KEY=067
LINK	.	.	00	00	.	00	15	.	.	.	00	.	.	**00**	**13**	00	00	**14**	Collision @ 17

Table A2. Inserting the sample keys into a hash table with collision resolution by open addressing with linear probing.

H --→	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	
KEY=185	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	**185**	.	.	
971	.	.	**971**	.	.	.	.	.	.	.	.	.	.	.	.	185	.	.	
400	.	.	971	.	.	.	.	.	.	**400**	.	.	.	.	.	185	.	.	
260	.	.	971	.	.	**260**	.	.	.	400	.	.	.	.	.	185	.	.	
922	.	.	971	.	**922**	260	.	.	.	400	.	.	.	.	.	185	.	.	
970	.	**970**	971	.	922	260	.	.	.	400	.	.	.	.	.	185	.	.	
543	.	970	971	.	922	260	.	.	.	400	.	.	.	.	.	185	**543**	.	
532	.	970	971	**532**	922	260	.	.	.	400	.	.	.	.	.	185	543	.	Collision @ 05
050	.	970	971	532	922	260	.	.	.	400	.	.	.	.	**050**	185	543	.	Collision @ 16
067	.	970	971	532	922	260	.	.	.	400	.	.	.	**067**	050	185	543	.	Collision @ 16

Table A3. Inserting the sample keys into a hash table with collision resolution by open addressing with double-hashing.

H --→	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	
KEY=185	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	**185**	.	.	
971	.	.	**971**	.	.	.	.	.	.	.	.	.	.	.	.	185	.	.	
400	.	.	971	.	.	.	.	.	.	**400**	.	.	.	.	.	185	.	.	
260	.	.	971	.	.	**260**	.	.	.	400	.	.	.	.	.	185	.	.	
922	.	.	971	.	**922**	260	.	.	.	400	.	.	.	.	.	185	.	.	
970	.	**970**	971	.	922	260	.	.	.	400	.	.	.	.	.	185	.	.	
543	.	970	971	.	922	260	.	.	.	400	.	.	.	.	.	185	**543**	.	
532	.	970	971	.	922	260	.	.	.	400	.	.	.	.	**532**	185	543	.	Collision @ 05; C=8
050	.	970	971	.	922	260	.	.	.	400	**050**	.	.	.	532	185	543	.	Collision @ 16; C=6
067	.	970	971	.	922	260	.	.	**067**	400	050	.	.	.	532	185	543	.	Collision @ 16; C=8