Direct File Organization

Computed Chaining

- The methods that use a link field provide better performance but require more storage.
- Those that don't use a link field require less space but provide poorer performance.
- Consider a third class in between. Instead of storing an actual address in the link, they store a pseudolink (which requires additional processing before it yields an actual address)
- If storage is limited, information needed can be computed rather than stored.

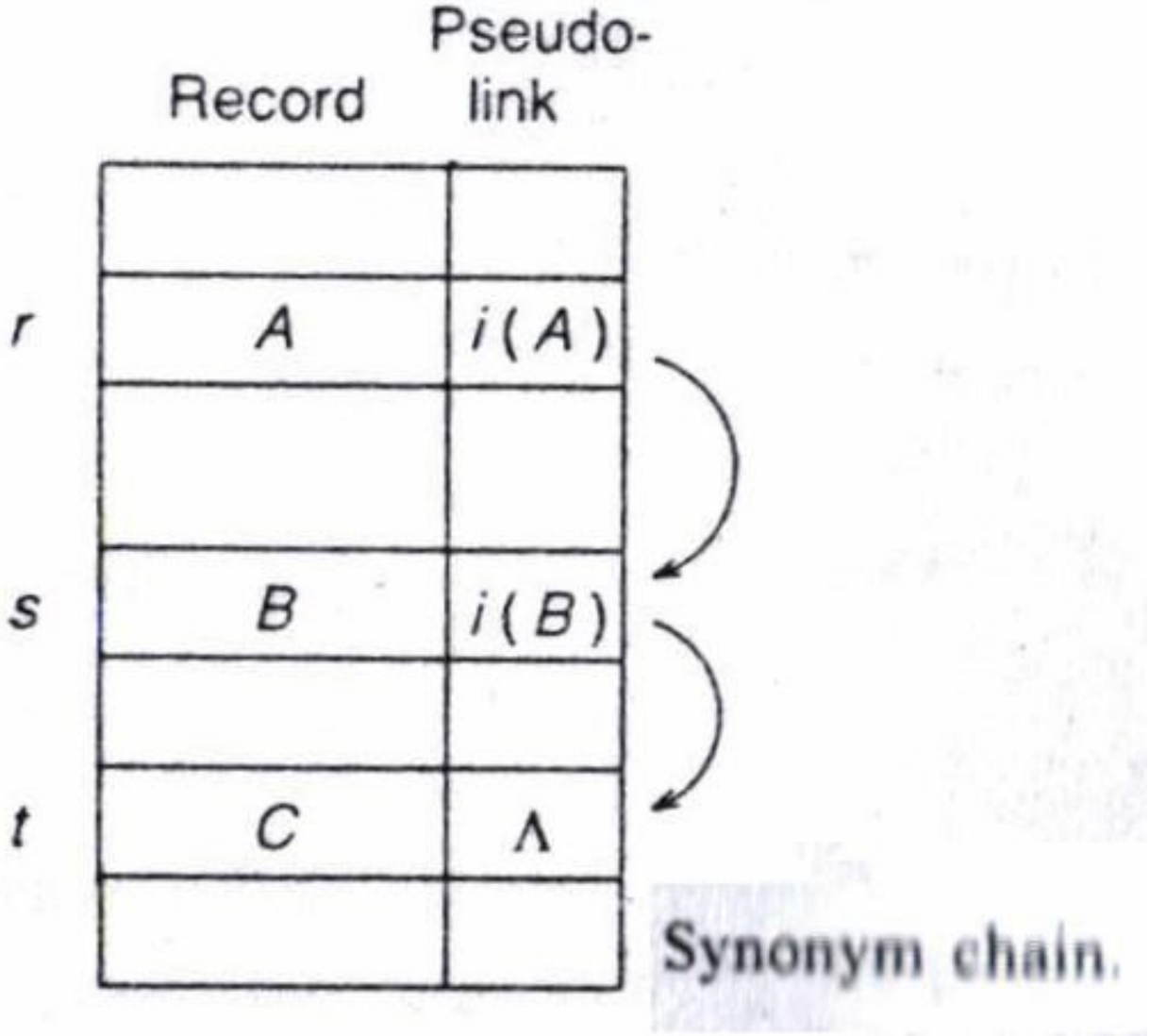
Computed Chaining

- The performance of the pseudolink methods are better than the nonlink methods. The number of offsets or locations which would need to be searched to locate the successor item are stored in the pseudolink so that we can compute the successor's actual address.
- The intermediate locations do not have to be searched so performance is improved.
- It takes fewer bits (less storage) to store the number of offsets rather than a full address.
- On the negative side, do not perform as well as the coalesced hashing methods and they require more storage than the nonlink methods.

Computed Chaining

- Follows the process of coalesced hashing in that the first item on a chain points to its immediate successor which points to its immediate successor.
- To eliminate the coalescing of chains, computed chaining also *moves* a record stored at another record's home address.
- There is no coalescing (improves performance).
- Unlike the nonlink methods, computed chaining uses the key of the record *stored* at a probe address to locate the next probe address and not the key of the record being *inserted* or *retrieved*.

Computed Chaining



Synonym chain.

- Using a function of the item stored at a location ensures that only one actual probe will be needed to locate the successor record.

Computed Chaining

Algorithm 3.5
COMPUTED CHAINING INSERTION

I. Hash the key of the record to be inserted to obtain the home address for storing the record.

II. If the home address is empty, insert the record at that location.

III. If the record is a duplicate, terminate with a "duplicate record" message.

IV. If the item stored at the hashed location is not at its home address, move it to the next empty location found by stepping through the table using the increment associated with its predecessor element, and then insert the incoming record into the hashed location, else

   A. Locate the end of the probe chain and in the process, check for a duplicate record.
   B. Use the increment associated with the last item in the probe chain to find an empty location for the incoming record. In the process, check for a full table.
   C. Set the pseudolink at the position of the predecessor record to connect to the empty location.
   D. Insert the record into the empty location.

Computed Chaining

Algorithm 3.6
PSEUDOCODE COMPUTED CHAINING INSERTION

```
proc   computed_chaining_insert

       /*  Inserts a record with a key x according to the computed chaining hashing
           method. Table[r] refers to the contents at location r in the file and pseudolink[r]
           contains the offset value associated with that location.   */

1      h ← Hash(x)   /* Locate the home address. */
2      if Table[h] = Λ then [Table[h] ← x; return] /* The home address is empty, so insert
                                the item.*/
3      if Table[h] = x then return /* The item is a duplicate.*/
4      if Hash(Table[h]) ≠ h then move the item /* The item stored at h is not stored at
                                its home address, so move it and store x at h. */
5      while pseudolink[h] ≠ Λ
6            do
7            h ← probe(Table[h],pseudolink[h],h) /* probe is a function to locate the
                          address of the next item in the probe chain */
8            if Table[h] = x then return /* The item is a duplicate*/
9            end
```

Computed Chaining

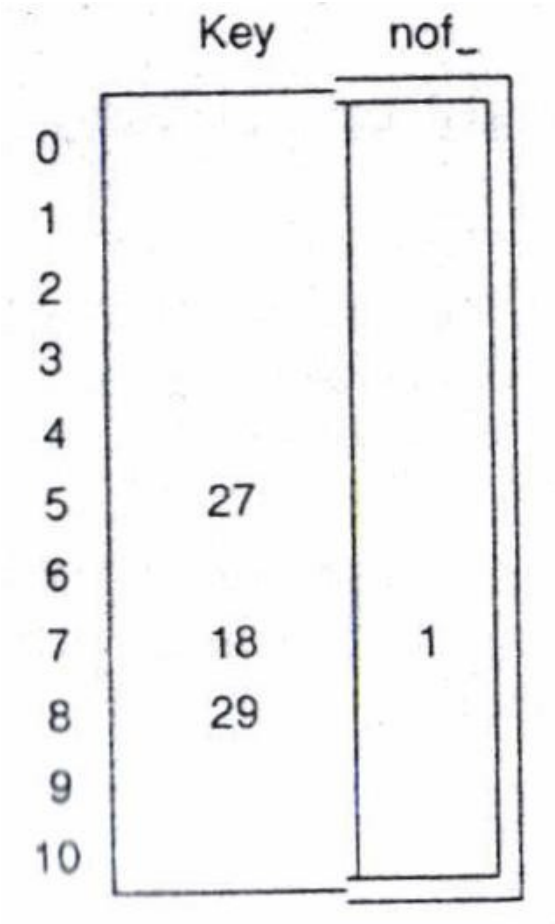
```
10      i ← 1    /* Initialize a loop variable to locate the first empty cell for storing x. */
11      j ← probe(Table[h],i, h) /* Locate the next probe address.*/
12      while Table[j] ≠ Λ
13            do
14            i ← i + 1
15            if i > table_size then [print "table full"; stop]
16            j ← probe(Table[h],i,h) /*Locate the next probe address. */
17            end
18      pseudolink[h] ← i /* Insert the probe number */
19      Table[j] ← x /* Store the item */

end     computed_chaining_insert
```

The **probe** function *computes* the address of the successor element given the key of the record stored at the current location and the pseudolink of the current location. The incrementing scheme of linear quotient is used to calculate the successor position.

Computed Chaining - Example

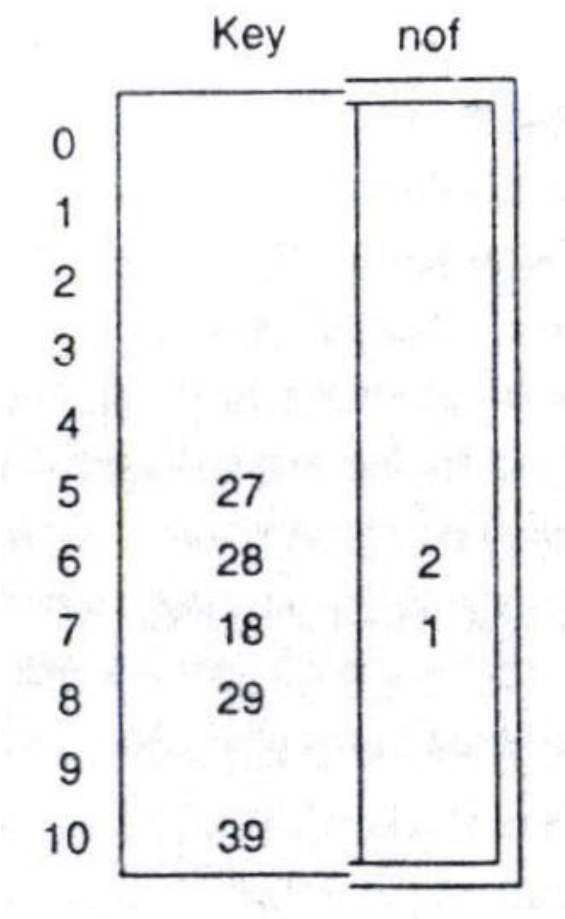
- Keys: 27, 18, 29, 28, 39, 13, 16, 38, 53
- Hash(key) = key mod 11
- i(key) = Quotient(Key / 11) mod 11

Computed Chaining - Example



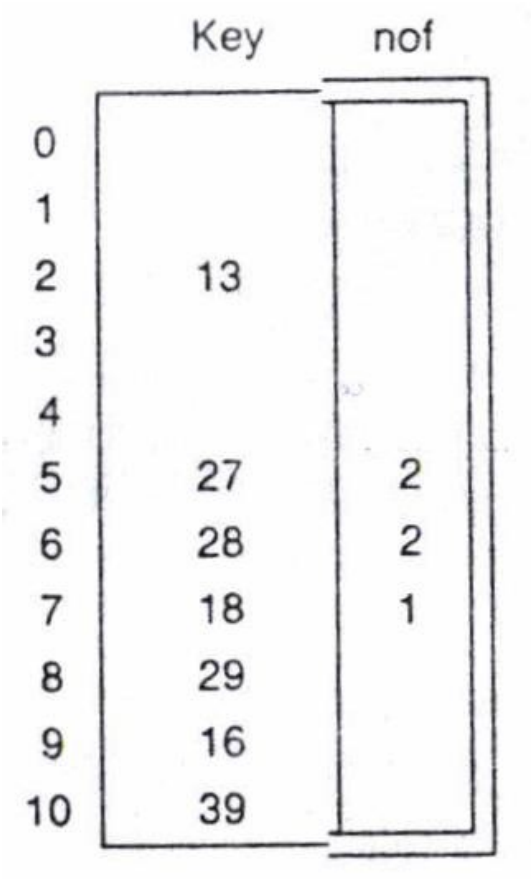
"nof" represents the number of offsets or the pseudolink value.

Computed Chaining - Example



- The increment is changed only when we reach the successor element on a probe chain.
- The current increment is always the one associated with the most recently visited record *on the current probe chain*.

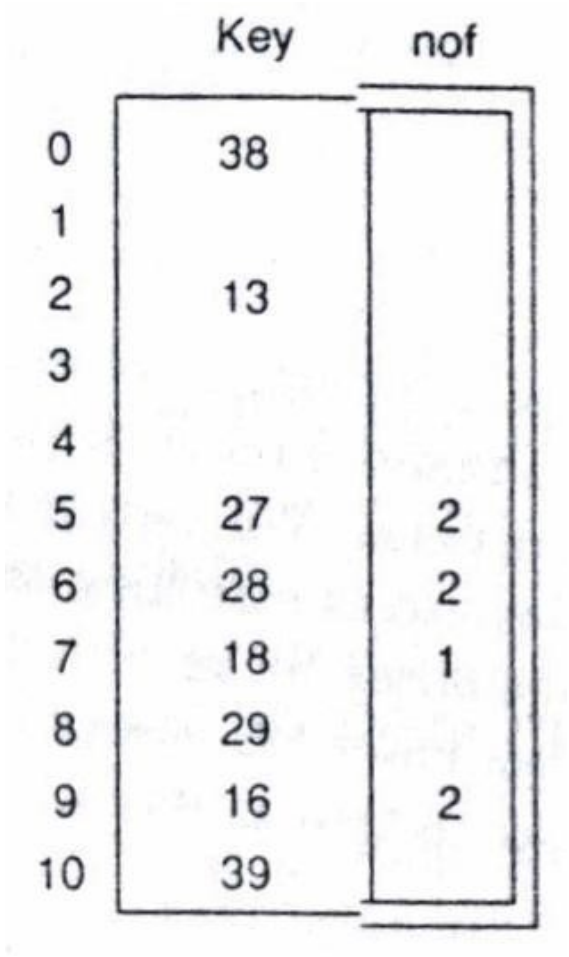
Computed Chaining - Example



	Key	nof
0		
1		
2	13	
3		
4		
5	27	2
6	28	2
7	18	1
8	29	
9	16	
10	39	

- Note: We don't change increments when we encounter an occupied location; we continue with an increment of two until we find an empty location or until we find that the table is full.

Computed Chaining - Example



	Key	nof
0	38	
1		
2	13	
3		
4		
5	27	2
6	28	2
7	18	1
8	29	
9	16	2
10	39	

- Which pseudolink field do we need to set?
- The one associated with 16.
- Since we had to look at two additional locations, the pseudolink value is two.

Computed Chaining - Example

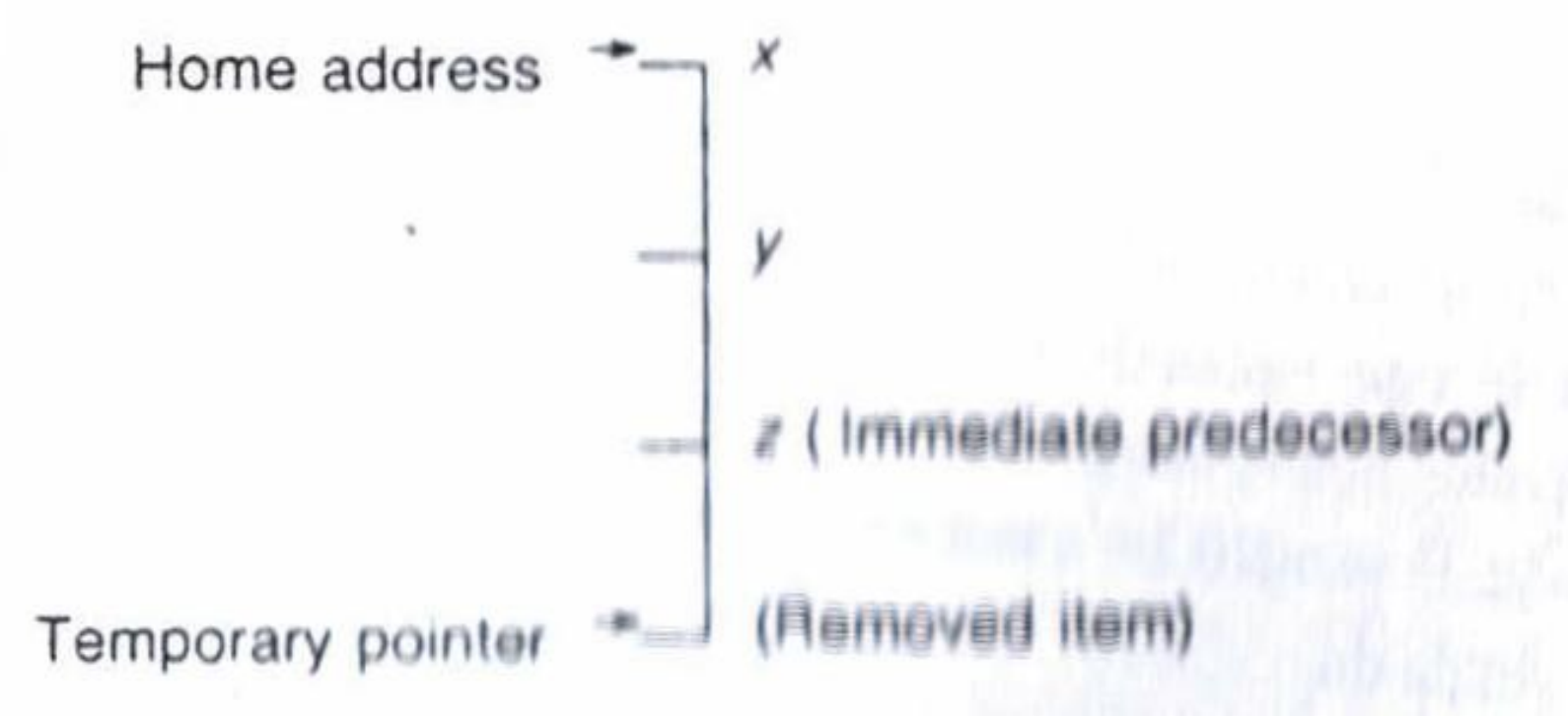
- We finally add the record with key 53.
- We observe a *move* process. We move 16 *plus* all its successors. Otherwise we would no longer be able to retrieve them.
- The record with key 53 may be inserted directly at location 9.
- We now reinsert the two records that were displaced by the insertion.

Computed Chaining - Example

- We need to keep a pointer or remember location 9 so that we can find the end of the chain that those two records were on previously.
- We need to identify the immediate predecessor to the item that was removed from location 9 so that we can relink it to the new location for the removed record.
- We know where to begin the search by hashing 16 qhich yields location 5. We search from that location until we find a pseudolink to location 9. We find that immediately at location 5.

Computed Chaining - Example

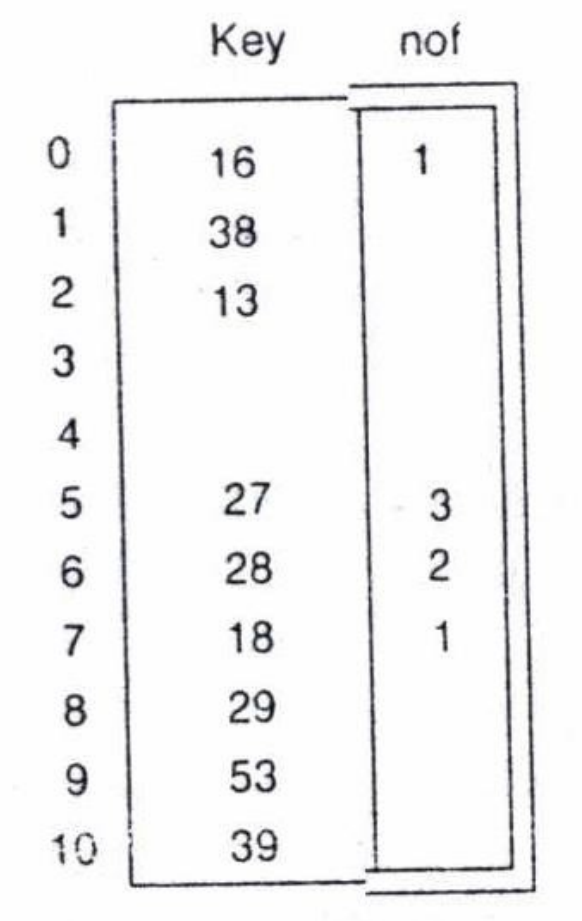
In general, the search for the immediate predecessor of a removed item may be represented as



Computed Chaining - Example

- We begin the search at the home address which contains a record and continue the search until we find a record that points to the location of the removed item which is indicated with the temporary pointer.
- We now know the location of the pseudolink field that needs to be modified in the reinsertion process.
- In the example, we then need to use that increment associated with 27 until we can find another empty space to insert the record with key 16.

Computed Chaining - Example



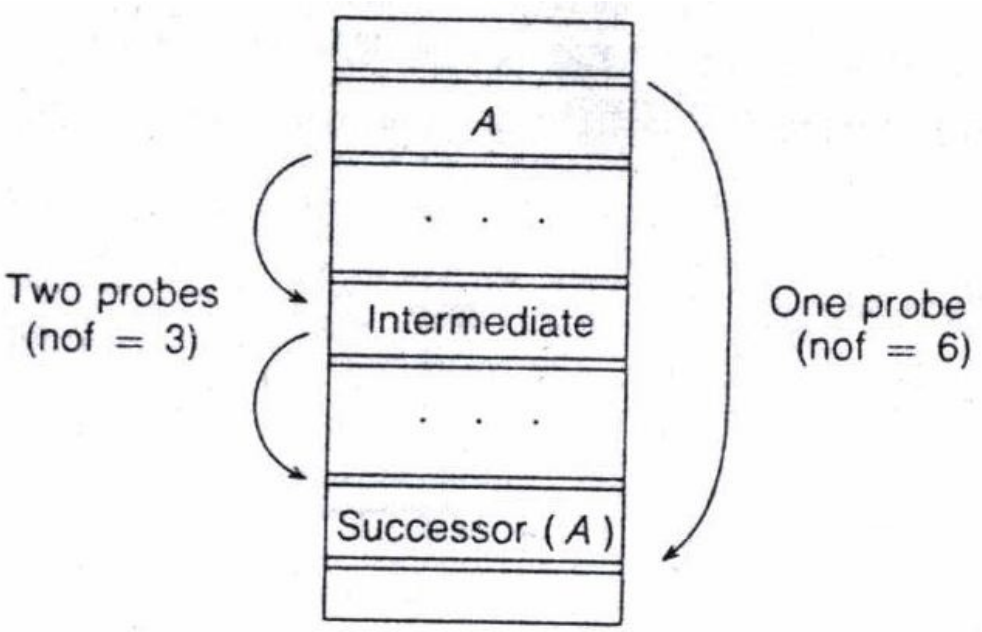
	Key	nof
0	16	1
1	38	
2	13	
3		
4		
5	27	3
6	28	2
7	18	1
8	29	
9	53	
10	39	

- Did we degrade the performance for retrieving 16 by moving it?
- No, we essentially have a linked list such that the actual physical location of a record is not important.

Computed Chaining - Discussion

• With computed chaining, the pseudolink field is normally only of a size sufficient to store the largest possible pseudolink value. But what if we do not know what the largest value is? Or what if only a limited, fixed number of bits is available for the pseudolink field?

If insufficient bits exist to store the actual pseudolink value, its greatest factor that can fit into the given number of bits is stored instead. But, this will increase the number of retrieval probes.



Computed Chaining - Discussion

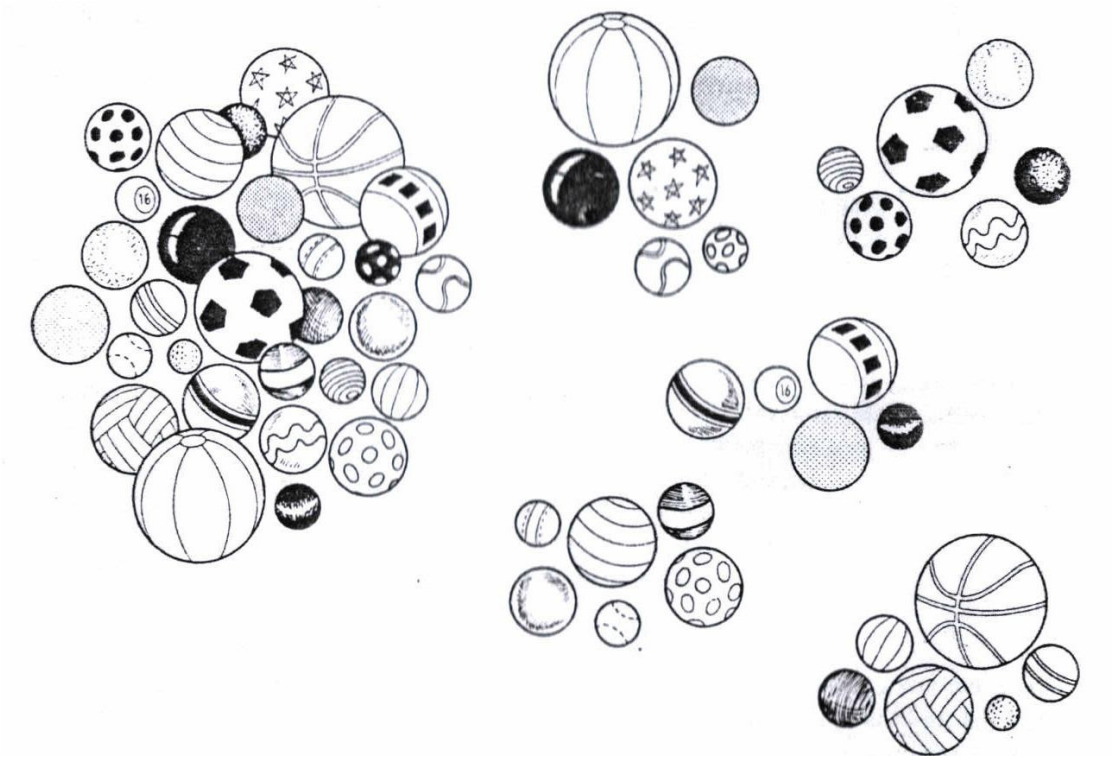
- The number of probes required to *retrieve* a record is equal to the position of that record in its probe chain.
- The worst case retrieval performance is *bounded* by the number of records that collide at any one home address. The other collision resolution methods have worst case performances based on the placement of the records inserted previously into the table.
- With computed chaining, the worst case retrieval performance can be improved by choosing a hashing function that evenly distributes the probable addresses for the given data.

Computed Chaining - Deletion

- To *delete* a record from a table formed using computed chaining requires the reinsertion of *all* the records that follow the deleted record on that probe chain.
- Computed chaining requires more time for inserting and deleting so that less time may be needed for retrieval.

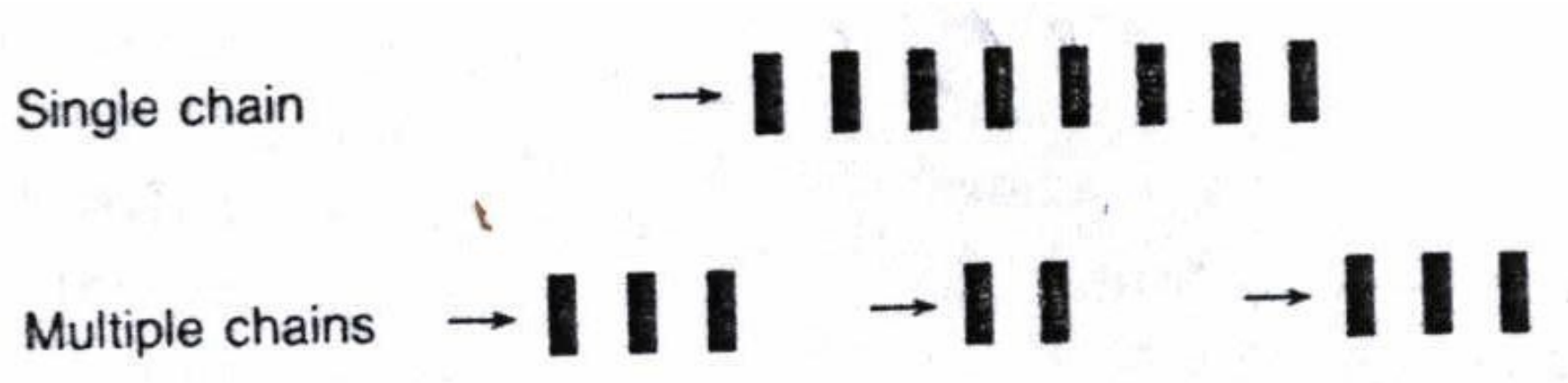
Computed Chaining - Variant

Instead of searching an entire file of records or an entire probe chain of records, if we could *divide* the records into smaller groups, we could narrow the search considerably if we knew the *one* group that the record must be in if it were in the file. Then we *conquer*.



Computed Chaining - Variant

- The concept of having multiple probe chains instead of a single probe chain for organizing records.
- Each chain will be smaller and the searching will be faster.



Computed Chaining - Variant

- In addition to the hashing function, Hash(key), to obtain the home address for a record and the incrementing function, *i*(key), to obtain an increment, we introduce a *third* hashing function, *g*(key), to tell us which probe chain to insert into or to search. This is an example of *triple hashing*.

$$g(\text{key}) \rightarrow 0, 1, ..., R - 1$$

where R is the number of subgroupings. A simple function for *g*(key) is

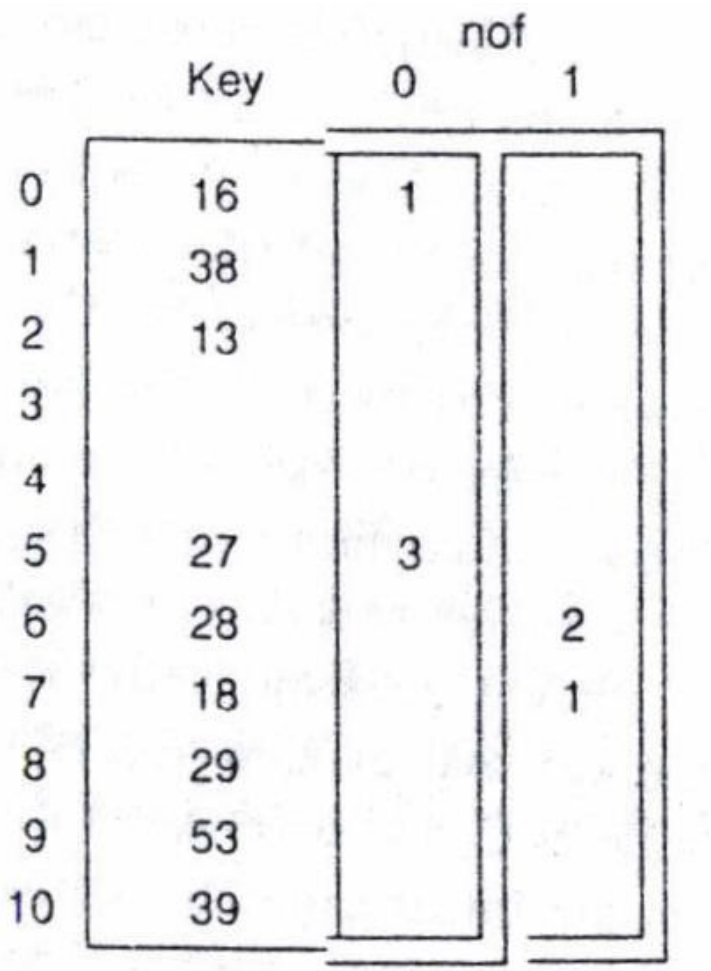
$$g(\text{key}) = \text{key mod } R$$

An Example with Multiple Chains

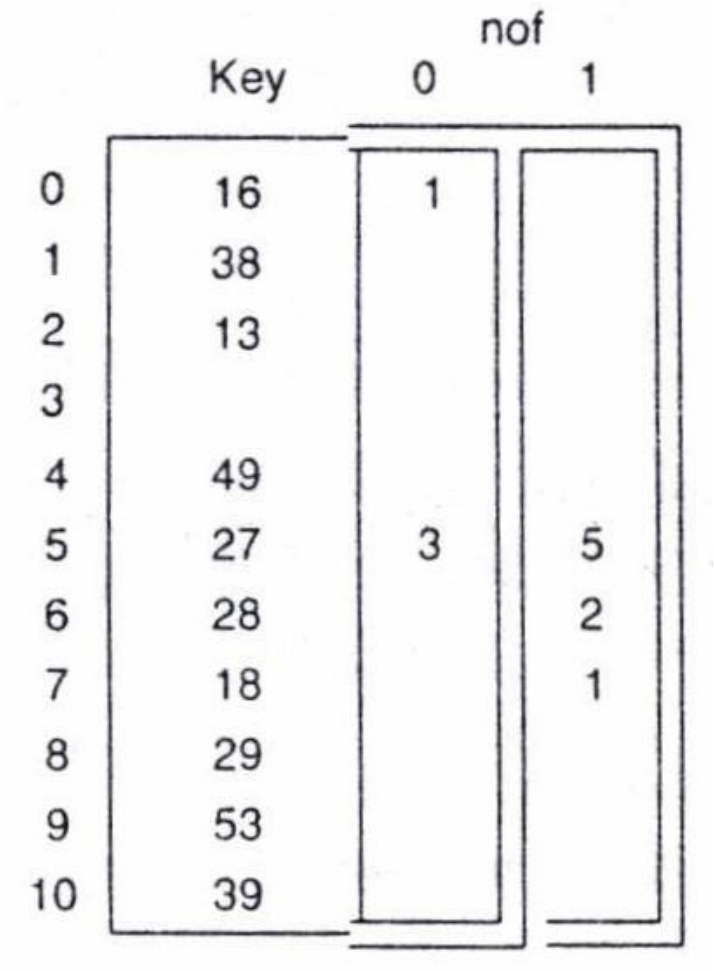
- Keys: 27, 18, 29, 28, 39, 13, 16, 38, 53
- Hash(key) = key mod 11
- *i*(key) = Quotient(Key / 11) mod 11
- *g*(key) = key mod 2

- If a key is even, it will go on the zeroth chain, if it is odd, it will go on the first chain. The records are inserted as before with the only difference being the placement of the pseudolink values: in chain zero for even key values and in chain one for odd key values.

An Example with Multiple Chains



	Key	nof 0	nof 1
0	16	1	
1	38		
2	13		
3			
4			
5	27	3	
6	28		2
7	18		1
8	29		
9	53		
10	39		



	Key	nof 0	nof 1
0	16	1	
1	38		
2	13		
3			
4	49		
5	27	3	5
6	28		2
7	18		1
8	29		
9	53		
10	39		

Comparison of Collision Resolution Methods

TABLE 3.3 COMPARISON OF MEAN NUMBER OF PROBES FOR SUCCESSFUL LOOKUP ($n = 997$; α = packing factor)

α (percent)	DCWC (10-bit link)*	LISCH (10-bit link)	Progressive overflow†	Linear quotient	Brent's method	Binary tree	Computed chaining (6-bit link)	Computed chaining (2-bit link)
20	1.100	1.106	1.125	1.15	1.102	1.102	1.070	1.070
40	1.200	1.232	1.333	1.277	1.217	1.217	1.168	1.214
60	1.300	1.379	1.750	1.527	1.367	1.364	1.264	1.381
70	1.350	—	2.167	1.720	1.444	—	1.323	1.528
80	1.400	1.566	3.000	2.011	1.599	1.579	1.356	1.715
90	1.450	1.674	5.500	2.558	1.802	1.751	1.408	2.062
95	1.475	1.734	10.500	3.153	1.972	1.880	1.433	2.414
99	1.495	1.783	50.500	4.651	2.242	2.049	1.601	3.330
100	1.500	—	—	6.522	2.494	2.134	—	—

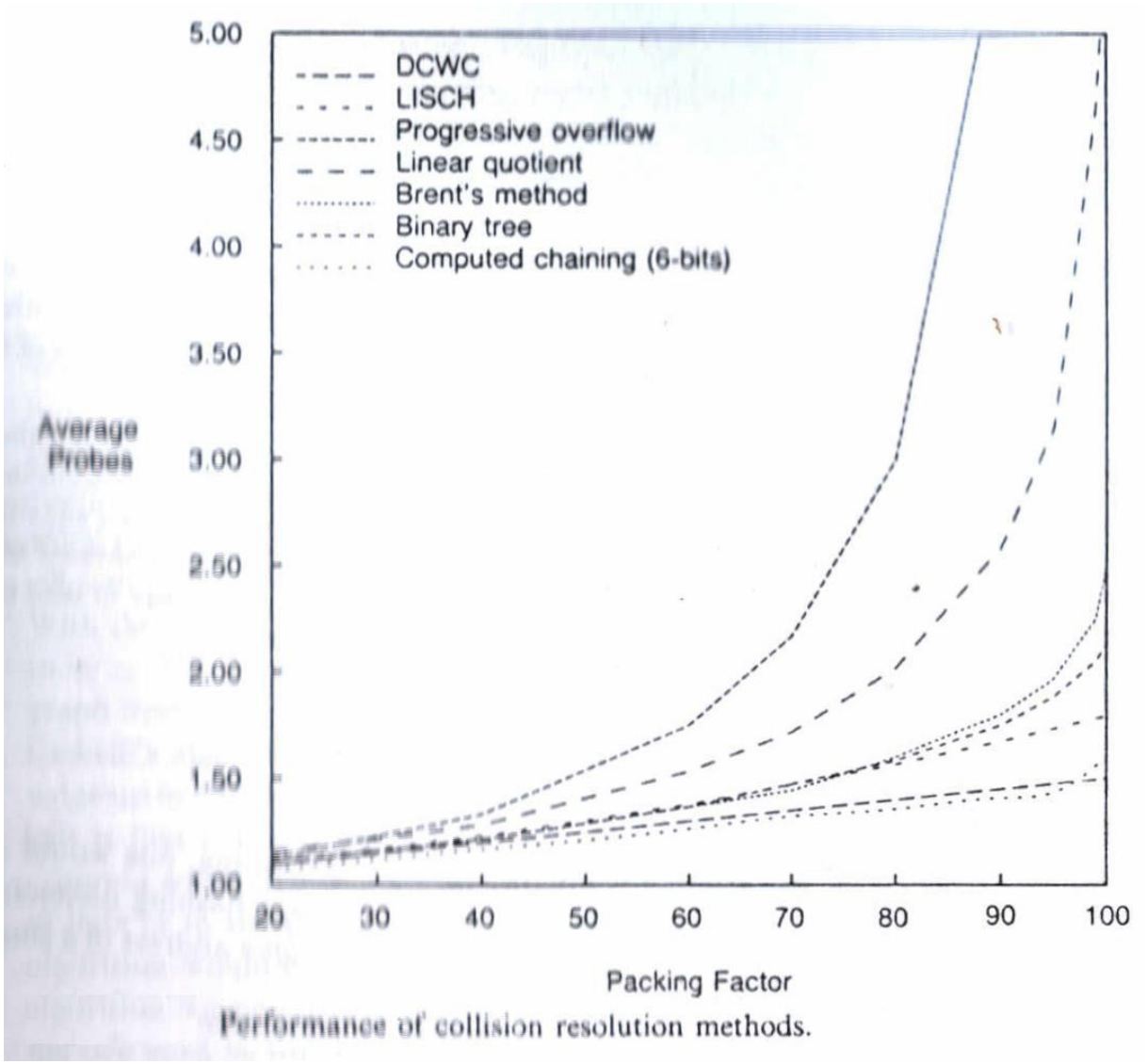
*Theoretical results; mean probes = $1 + \alpha/2$.

†Theoretical results; mean probes = $(1 - \alpha/2) / (1 - \alpha)$

For successful searches

Although the differences may not seem that great for a single retrieval, multiply those numbers by 10000 or 10000000.

Comparison of Collision Resolution Methods



Performance of collision resolution methods.

Comparison of Collision Resolution Methods

TABLE 3.4 SEARCH, RELOCATION, AND STORAGE COMPARISONS

Criteria	LISCH	Progessive overflow	Linear quotient	Brent's method	Binary tree	Computed chaining
Successful search						
Best case	1	1	1	1	1	1
Worst case	n	n	n	$\sqrt{n}$	$\ln n$	n
Move an item	No	No	No	Yes	Yes	Yes
Extra storage	Yes	No	No	No	No	Yes

Although the worst case performance for locating a record with both LISCH and computed chaining is n, their typical performances would be better, because only records on one chain need to be searched. With computed chaining, all the records on a chain must have the same home address, which further limits the size of the chain.

Comparison of Collision Resolution Methods

- The method that provides the lowest average number of probes, in general is DCWC. The second is computed chaining.
- Computed chaining gives better results over LISCH. It eliminated coalescing. Without coalescing LISCH is DCWC and does perform better than computed chaining.
- In computed chaining, for a record that has a large number of locations visited in locating its successor, it may be necessary to store a factor of the number of offsets. That requires more probes on retrieval.

Comparison of Collision Resolution Methods

- If storage is somewhat scarce, computed chaining will then have an advantage over DCWC.
- If space is limited, the method that gives the best performance without requiring additional storage for the records is the binary tree method. But, it requires more processing time and temporary storage for insertions than those methods that do not use a link or pseudolink.

Comparison of Collision Resolution Methods

TABLE 3.5 ADVANTAGES, DISADVANTAGES, AND WHEN TO USE VARIOUS COLLISION RESOLUTION METHODS

Method	Advantages	Disadvantages	When to use
Coalesced hashing	Excellent performance	Requires space for link field	When ample space is available
[DCWC]	Superior performance	Requires space for link field	When ample space is available and insertion time is not critical
Progressive overflow	No additional space; simple algorithm	Very poor performance	Almost NEVER
Linear quotient	No additional space; reasonable performance	Performance below that of other methods	When space is at a premium; retrieval performance is not critical
Brent's method	No additional space; good performance	Performance below that of linked methods	When space is at a premium, even space for a binary tree
Binary tree	No additional space; very good performance	Performance below that of linked methods	When file space is limited but temporary space is available for tree; performance is important
Computed chaining	Small amount of additional space; excellent performance	Requires some space for pseudolink field; performance below that of DCWC	When some extra file space is available and performance is important; insertion time is not critical

Perfect Hashing

hash(key) → **unique** address

- Both primary and secondary clustering are eliminated.
- With perfect hashing, we need only a single probe to retrieve a record.
- Perfect hashing is applicable to files with only a relatively small number of records because the computing time is big.

Perfect Hashing

- A **perfect hashing function** maps a key into a unique address. If the range of potential addresses is the same as the number of keys, the function is a minimal (in space) perfect hashing function.
- A separate hashing needs to be devised for *each* set of keys. If one or more of the keys change, a new hashing function must be constructed.

Perfect Hashing

- In *minicycle algorithm*, for a table of size N, a perfect hashing function may be characterized as:

  p.hash(key) = (h_0(key)+g[h_1(key)]+g[h_2(key)]) mod N

- What needs to be decided are the functions h_0, h_1, h_2 and g.
- These functions should be efficient.

Perfect Hashing

- The worst case of minicycle algorithm is $O(r^6)$ where r is the numbr of records in the set. An upper bound for r seems to be about 512.
- A special case of minicycle alg. is the Cichelli's algorithm which is not as efficient as minicycle alg., appropriate for r upto 60, plus more disadvantages, but it is straightforward to understand.
- For larger amounts, the minicycle algorithm is appropriate.

Cichelli's Algorithm

$$h_0 = \text{length(key)}$$

$$h_1 = \text{first\_character(key)}$$

$$h_2 = \text{last\_character(key)}$$

$$g = T(x)$$

- where T is a table of values associated with individual characters x which may appear in a key. The time consuming part is determining T.
- In the table, a value may be assigned to more than one character.

Cichelli's Algorithm

TABLE 3.6 VALUES ASSOCIATED WITH THE CHARACTERS OF THE PASCAL RESERVED WORDS

A=11	B=15	C=1	D=0	E=0	F=15
G=3	H=15	I=13	J=0	K=0	L=15
M=15	N=13	O=0	P=15	Q=0	R=14
S=6	T=6	U=14	V=10	W=6	X=0
Y=13	Z=0				

Let's apply the alg. to the keyword **begin**.

$$\text{p.hash}(\textbf{begin}) = 5 + T(h_1(\text{key})) + T(h_2(\text{key}))$$

$$= 5 + T(\text{b}) + T(\text{n})$$

$$= 5 + 15 + 13$$

$$= 33$$

Algorithm 3.7
CICHELLI'S ALGORITHM FOR DETERMINING A PERFECT HASH FUNCTION

I. Order the set of keys based upon the sum of the frequencies of occurrence of the first and last characters of the keys in the entire set. Place the keys with the most frequently occurring first and last characters at the top of the ordering.

II. Modify the ordering. Place a key whose first and last characters both appear in prior keys as high in the ordering such that this condition holds.

III. Assign the value 0 to the first and last characters of the first key in the set.

IV. For each remaining key in the set, in order,

   A. If the first and last characters have already been assigned values, hash the key to determine if a conflict arises. If yes, discontinue processing on this key and recursively apply this step with the previous key (backtrack).

   B. If either the first *or* last character has already been assigned a value, assign a value to the other character by trying all possibilities from 0 to the maximum allowable value. If a conflict still exists after trying all possible values for the other character, then discontinue processing on this key and recursively apply this step with the previous key (backtrack).

   C. If both the first and last characters are as yet unassigned, vary the first and then the last character, trying each combination. If all combinations cause conflicts, then discontinue processing on this key and recursively apply this step with the previous key (backtrack).

V. If all keys have been processed, terminate successfully, else terminate unsuccessfully.

Cichelli's Algorithm

- The algorithm involves ordering the keys based on the frequencies of occurrence of the first and last characters in the keys.
- Assignment of values are made to the characters in the first and last positions of the keys from the top of the ordering to the bottom.
- The algorithm uses an exhaustive search with backtracking.

Cichelli's Algorithm - Example

Keys: cat, ant, dog, gnat, chimp, rat, toad

- We assume that the maximum value that may be assigned to a character is 4. If we cannot find a solution using 4, we would try a larger value. If this maximum value is too small, we will either have no solution or a great amount of backtracking. If the value is too large we won't obtain a minimal solution.

Cichelli's Algorithm - Example

- The frequencies of occurrence of the first and last characters are

a=1 *c*=2 *d*=2 *g*=2 *p*=1 *r*=1 *t*=5

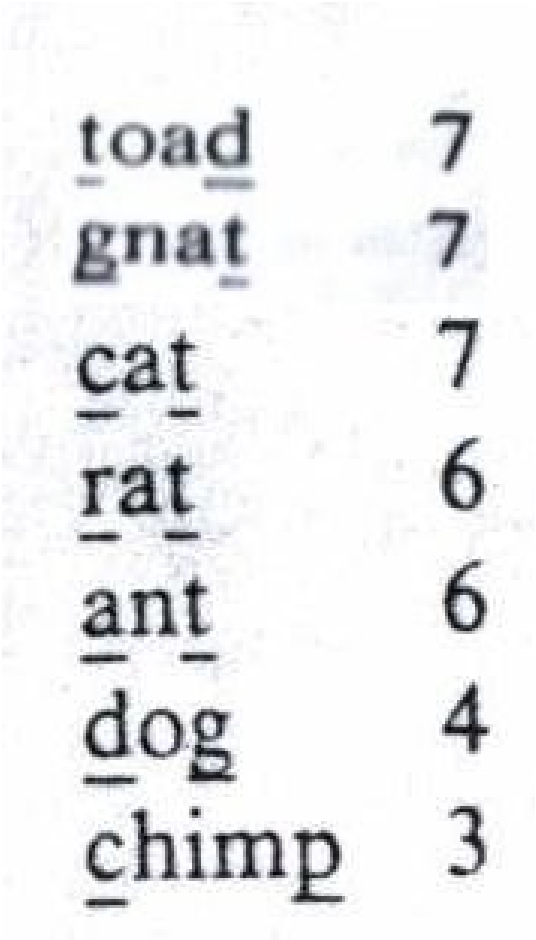
Cichelli's Algorithm - Example

We can compute the sums of the frequencies of occurrence of the first and last characters of each key. We obtain

cat	7
ant	6
dog	4
gnat	7
chimp	3
rat	6
toad	7

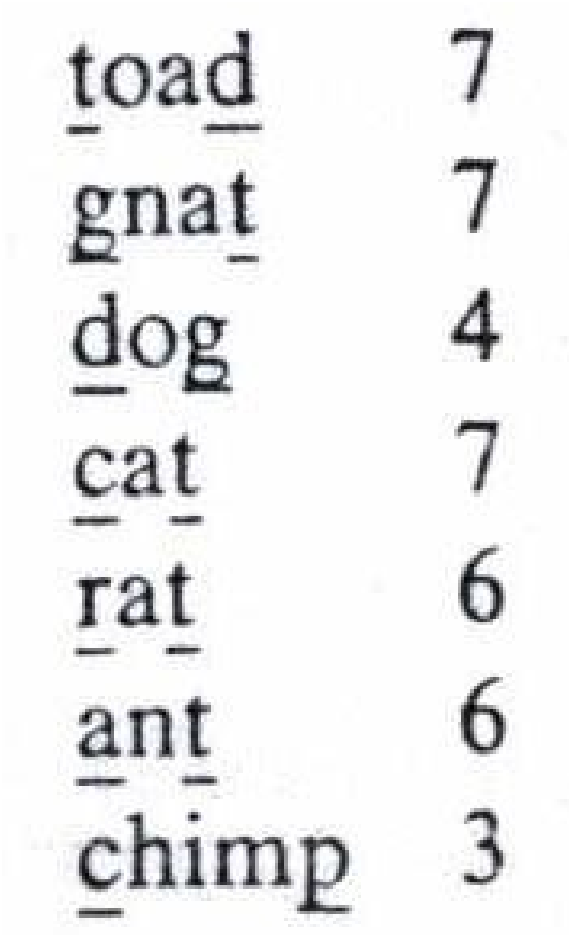
Cichelli's Algorithm - Example

- We next order the keys in descending order based on the sum of the frequencies to obtain what is on the right.
- We arbitrarily chose the descending order. A random ordering would have been equally acceptable.



Cichelli's Algorithm - Example

- Then we check the ordering to see if any keys exist that have both their first and last characters appearing in previous keys.
- Notice that the *d* and *g* of "dog" have appeared previously so that "dog" would then be moved to the position following "gnat".
- The ordering is shown on the right.



Cichelli's Algorithm - Example

- We begin assigning values to the characters.

$t = 0 \quad d = 0$

p.hash(toad) = 4

$t = 0 \quad d = 0 \quad g = 1$

p.hash(toad) = 4

p.hash(gnat) = 5

Cichelli's Algorithm - Example

- In "dog" we have a conflict. So, backtrack

$t = 0 \quad d = 0 \quad g = 2$

p.hash(toad) = 4

p.hash(gnat) = 6

$t = 0 \quad d = 0 \quad g = 2$

p.hash(toad) = 4

p.hash(gnat) = 6

p.hash(dog) = 5

Cichelli's Algorithm - Example

$t = 0 \quad d = 0 \quad g = 2 \quad c = 0$

p.hash(toad) = 4

p.hash(gnat) = 6

p.hash(dog) = 5

p.hash(cat) = 3

Cichelli's Algorithm - Example

$t = 0 \quad d = 0 \quad g = 2 \quad c = 0 \quad r = 4$

p.hash(toad) = 4

p.hash(gnat) = 6

p.hash(dog) = 5

p.hash(cat) = 3

p.hash(rat) = 7

Cichelli's Algorithm - Example

$t = 0 \quad d = 0 \quad g = 3 \quad c = 0 \quad r = 2$

p.hash(toad) = 4

p.hash(gnat) = 7

p.hash(dog) = 6

p.hash(cat) = 3

p.hash(rat) = 5

- We are not able to assign *a* value to a without a collision. So, we backtracked.

Cichelli's Algorithm - Example

$t = 0 \quad d = 0 \quad g = 4 \quad c = 0 \quad r = 2 \quad a = 3$

p.hash(toad) = 4

p.hash(gnat) = 8

p.hash(dog) = 7

p.hash(cat) = 3

p.hash(rat) = 5

p.hash(ant) = 6

- Problem still existed and we backtracked.

Cichelli's Algorithm - Example

$t = 0 \quad d = 0 \quad g = 4 \quad c = 0 \quad r = 2 \quad a = 3 \quad p = 4$

p.hash(toad) = 4

p.hash(gnat) = 8

p.hash(dog) = 7

p.hash(cat) = 3

p.hash(rat) = 5

p.hash(ant) = 6

p.hash(chimp) = 9

Since this solution maps the seven keys into seven consecutive locations, it is a minimal perfect hashing function.

Cichelli's Algorithm - Discussion

- The amount of backtracking can be big. and computational time is substantial. On the positive side, needs to be computed only once.
- The disadvantages:
  - It requires that no two keys of the same length share the first and last characters.
  - For a list with more than 45 elements, it may be necessary to segment it into sublists.
  - For certain lists, it may be impossible to tell in advance if the method will yield a minimal perfect hashing function.

Cichelli's Algorithm - Discussion

- But even as the applicable file size increases recently, the other hashing techniques that we examined will still have their place, for minimal perfect hashing requires a *static* set of keys, and in many circumstances that requirement cannot be met.