Hash tables offer exceptional performance when not overly full.

This is the traditional dilemma of all array-based data structures:

- Make the table too small, performance degrades and the table may overflow
- Make the table too big, and memory gets wasted.

Rehashing or *variable hashing* attempts to circumvent this dilemma by expanding the hash table size whenever it gets too full.

Table size : M > N

For small load factor the performance is much better,

than for N/M close to one.

Best choice: N/M = 0.5

When N/M > 0.75 – rehashing

Build a second table twice as large as the original and rehash there all the keys of the original table.

Expensive operation,

running time O(N)

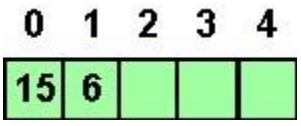
However, once done, the new hash table will have good performance.

1. Expanding the hash Table

For example, using open addressing (linear probing) on a table of integers with `hash(k)=k` (assume the table does an internal `% hSize`):

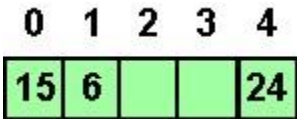
We know that performance degrades when $\lambda > 0.5$

Solution: rehash when more than half full



So if we have this table, everything is fine.

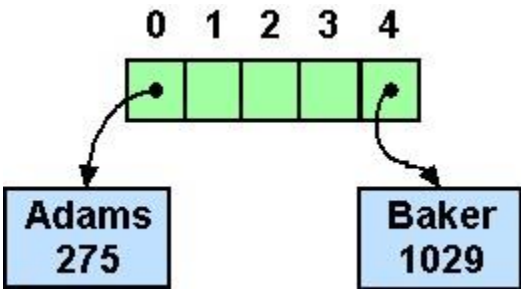
But if we try to add another element (24), then more than half the slots are occupied…



So we expand the table, and use the hash function to relocate the elements within the larger table…

In this case, I've shown the hash table size doubling, because that's easy to do, despite the fact that it doesn't lead to prime-number sized tables. If we were going to use quadratic probing, we would probably keep a table of prime numbers on hand for expansion sizes, and we would probably choose a set of primes such that each successive prime number was about twice the prior one.

2. Saving the Hash Values



The rehashing operation can be quite lengthy. Luckily, it doesn't need to be done very often.

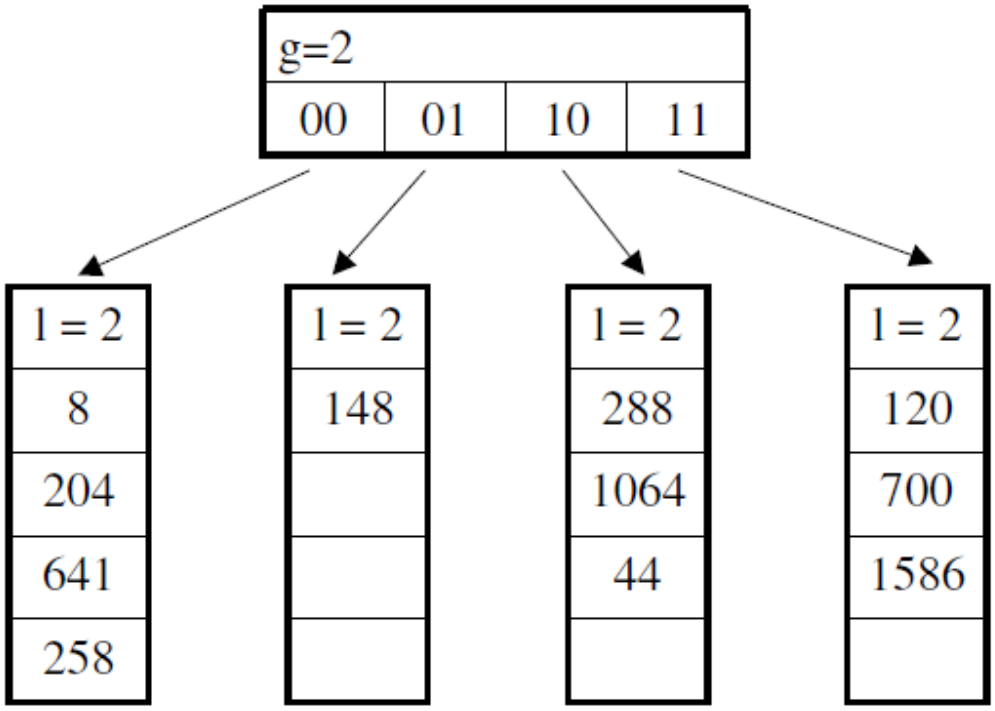
We can speed things up somewhat by storing the hash values in the table elements along with the data so that we don't need to recompute the hash values. Also, if we structure the table as a vector of *pointers to* the hash elements, then during the rehashing we will only be copying pointers, not the entire (potentially large) data elements.

Extendible Hashing Example

- Suppose that g=2 and bucket size = 4.
- Suppose that we have records with these keys and hash function h(key) = key mod 64:

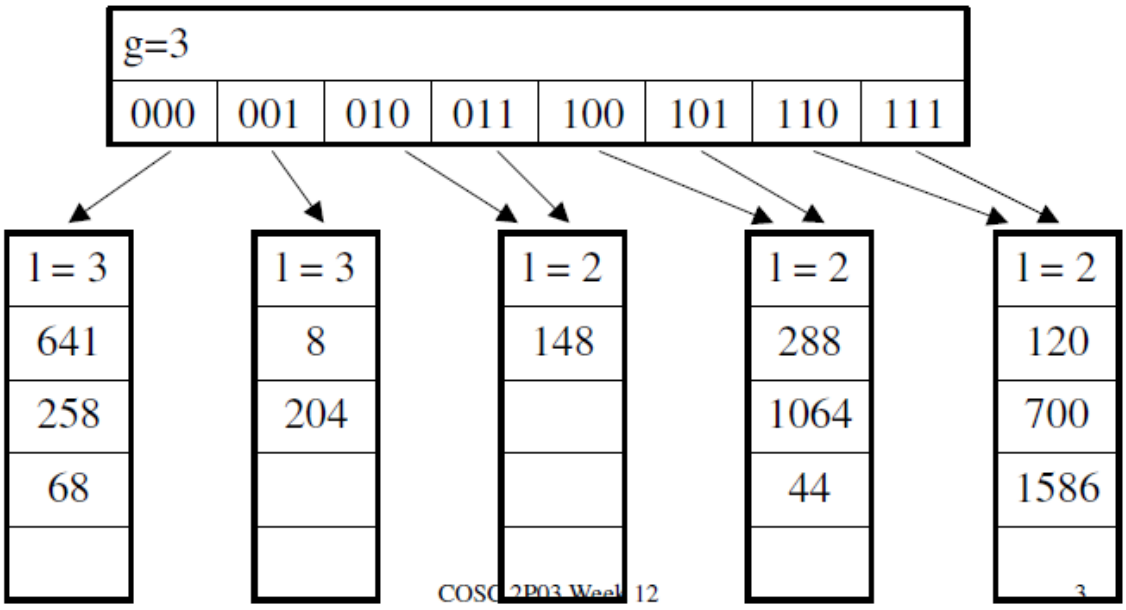
key	h(key) = key mod 64	bit pattern
288	32	100000
8	8	001000
1064	40	101000
120	56	111000
148	20	010100
204	12	001100
641	1	000001
700	60	111100
258	2	000010
1586	50	110010
44	44	101010

Extendible Hashing Example – directory and bucket structure



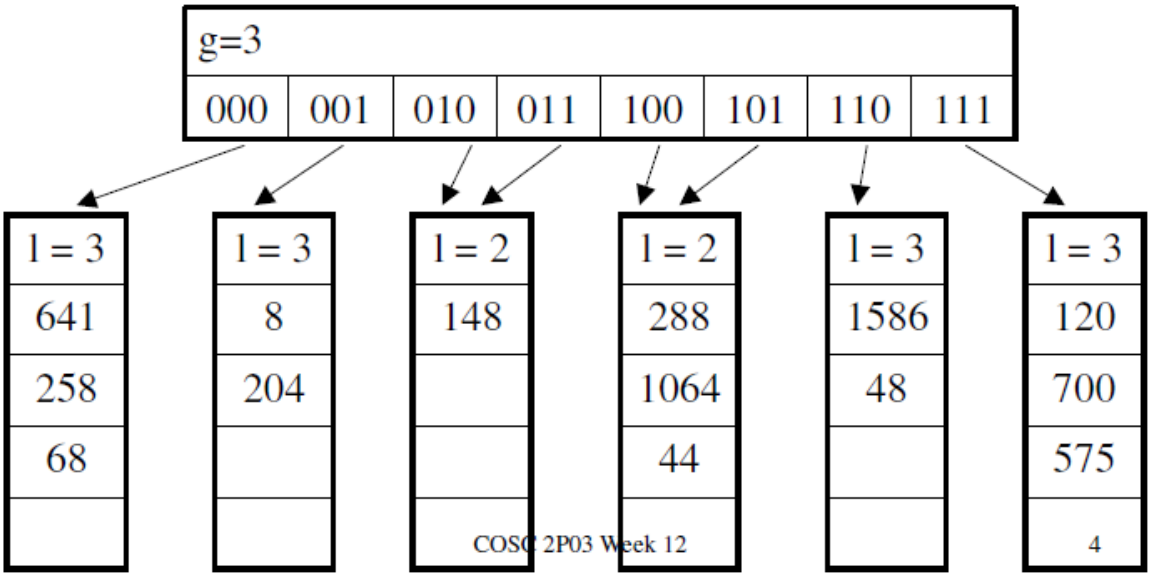
Bucket and directory split

- Insert 68
- 68 mod 64 = 4 = 000100



Bucket split – no directory split

- Insert 48 and 575
- 48 mod 64 = 48 = 110000
- 575 mod 64 = 63 = 111111

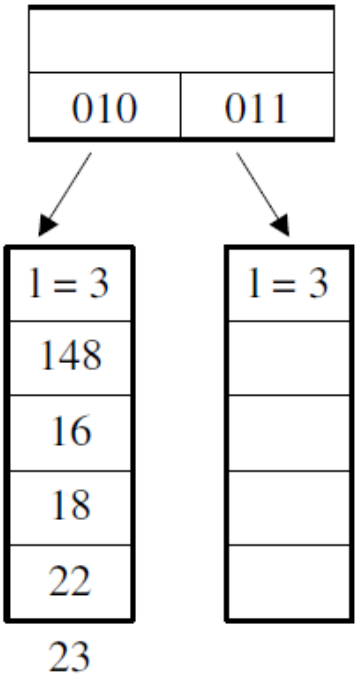


Multiple splits

- Insert 16, 18, 22, 23
- 16 mod 64 = 16 = 010000
- 18 mod 64 = 18 = 010010
- 22 mod 64 = 22 = 010110
- 23 mod 64 = 23 = 010111

Setting l=3 gives this intermediate (partial) picture…

Continue to next page…



Multiple splits, continued

- Setting l=4 (and thus g=4) gives this final result…