HASH TABLES

HASHING

- Key indexed arrays had perfect search performance O(1)
  - But required a dense range of index values
    - Otherwise memory is wasted
- Hashing allows us to approximate this performance but
  - Allows arbitrary types of keys
  - Map(hash) keys into compact range of index values
    - Items are stored in an array accessed by this index value
  - Allows us to approach the ideal of title[hashfunction(“COMP1927”)] = “Computing 2”;

HASHING

- A hash table implementation consists of two main parts:
  - (1) A hash function to map each key to an index in the hash table (array of size N).
    - Key->[0..N-1]
  - (2) A collision resolution so that
    - if hash table at the calculated index is already occupied with an item with a different key, an alternative slot can be found
    - Collisions are inevitable when dom(Key) > N

HASH FUNCTIONS

- Requirements:
  - if the table has TableSize entries, we need to hash keys to [0..TableSize-1]
  - the hash function should be cheap to compute
  - the hash function should ideally map the keys evenly to the index values - that is, every index should be generated with approximately the same probability
    - this is easy if the keys have a random distribution, but requires some thought otherwise

- Simple method to hash keys: modular hash function
  - compute i%TableSize
  - choose TableSize to be prime

HASHING STRING KEYS

- Consider this potential hash function:
  - we can turn a string into an Integer value:

```
int hash (char *v, int TableSize) {
   int h = 0, i = 0;
   while (v[i] != '\0') {
     h = h + v[i];
     i++;
   }
   return h % TableSize;
}
```

- What is wrong with this function?
  - How can it be improved?

HASHING STRING KEYS

- A better hash function:

```
 int hash (char *v, int TableSize) {
   int h = 0, i = 0;
   int a = 127; //prime number
   while (v[i] != '\0') {
     h = (a*h + v[i]) % TableSize;
     i++;
   }
   return h;
}
```

HASHING STRING KEYS

- Universal hash function for string keys:
  - Uses all of value in hash, with suitable randomization

```
 int hashU (char *v, int TableSize) {
   int h = 0, i = 0;
   int a = 31415, b = 27183;
   while (v[i] != '\0') {
     h = (a*h + v[i]) % TableSize;
     a = a*b% (TableSize-1);
     i++;
   }
   return h;
}
```

Real Hash Function

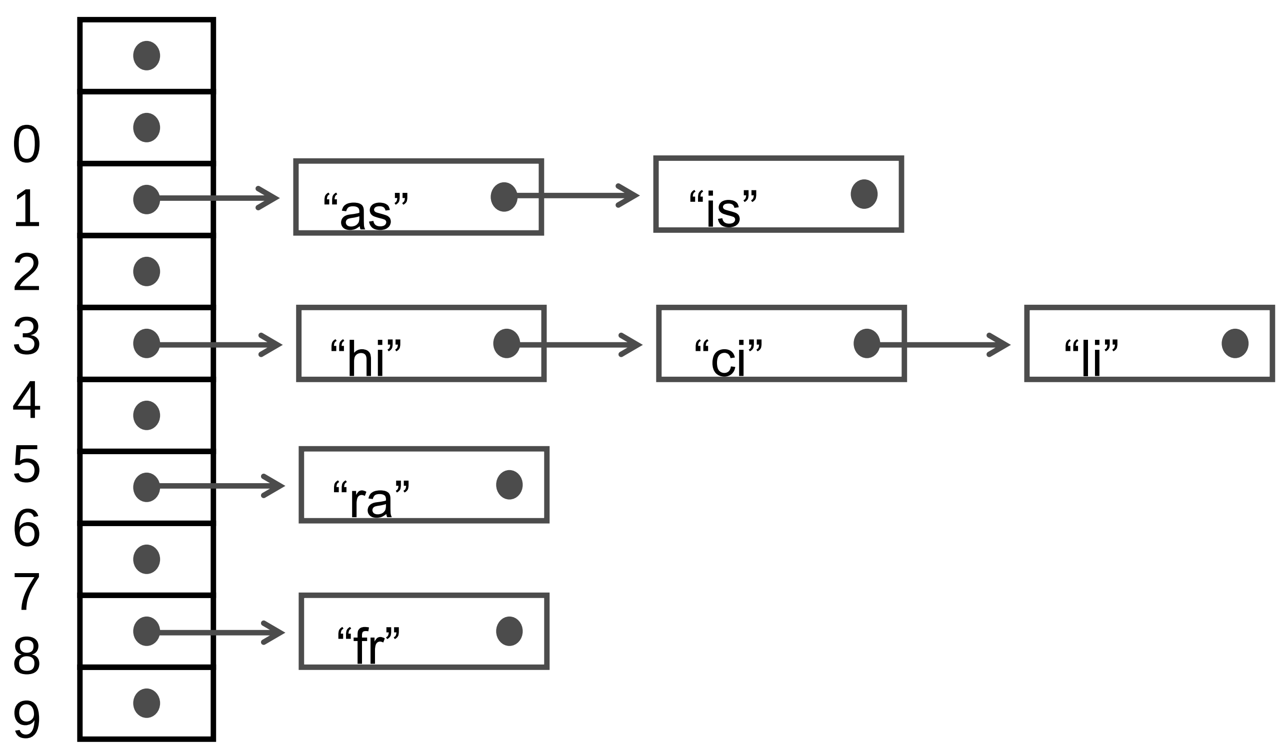
```
//from PostgreSQL DBMS
hash_any(unsigned char *k, register int keylen, int N) {
  register uint32 a, b, c, len;
  // set up internal state
  len = keylen;
  a = b = 0x9e3779b9; c = 3923095;
  // handle most of the key, in 12-char chunks
  while (len >= 12) {
    a += (k[0] + (k[1] << 8) + (k[2] << 16) + (k[3] << 24));
    b += (k[4] + (k[5] << 8) + (k[6] << 16) + (k[7] << 24));
    c += (k[8] + (k[9] << 8) + (k[10] << 16) + (k[11] << 24));
    mix(a, b, c);
    k += 12;
    len -= 12;
  }
  // collect any data from remaining bytes into a,b,c
  mix(a, b, c); return c % N; }
```

Collision Resolution: Separate Chaining

- What do we do if two entries have the same array index?
  - maintain a list of entries per array index (separate chaining)
  - use the next entry in the hash table (linear probing)
  - use a key dependent increment for probing (double hashing)

SEPARATE CHAINING

- Can be viewed as a generalisation of sequential search
- Reduces number of comparisons by a factor of TableSize
- See lecture code for implementation



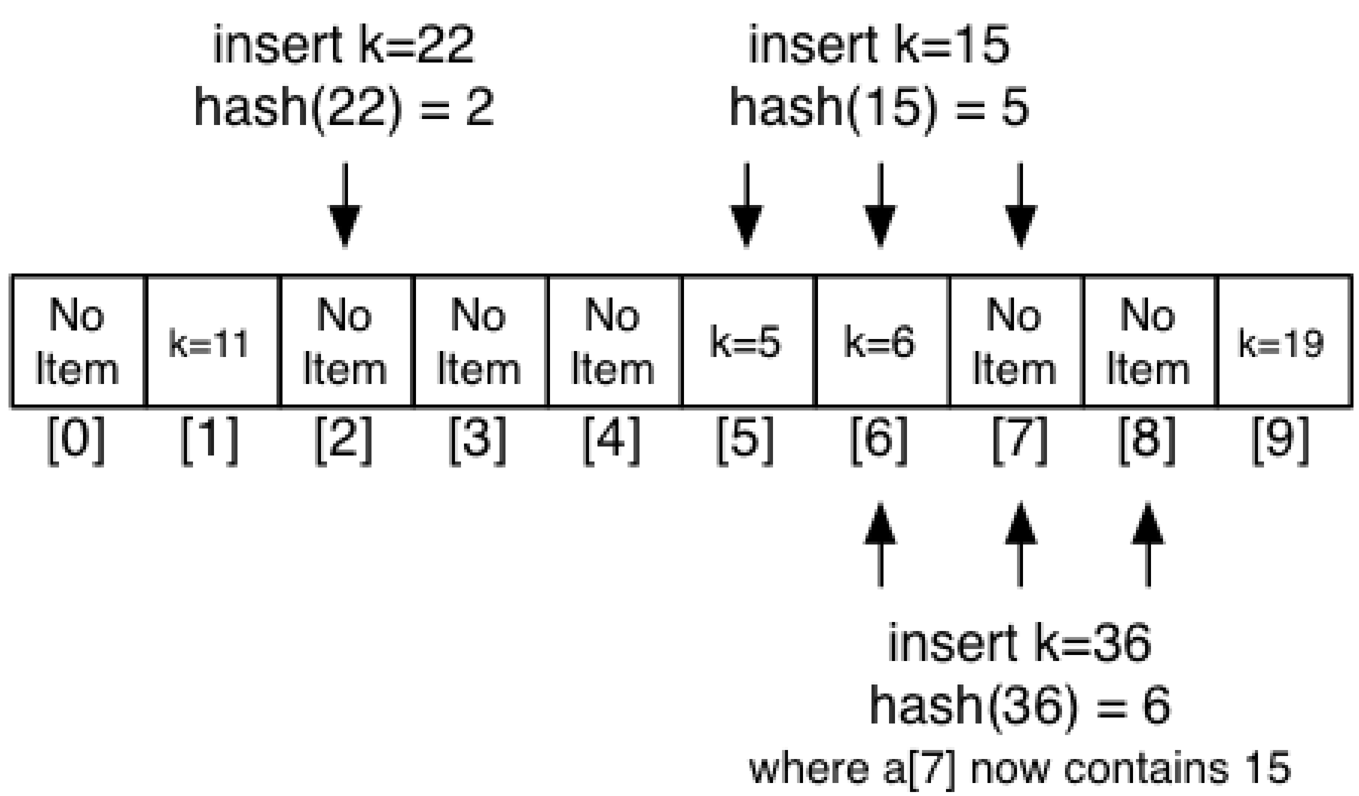
Separate Chaining

- Cost Analysis:
  - N array entries(slots), M stored items
  - Best case: all lists are the same length
    - M/N
  - Worst case: one list of size M all the rest are size 0
  - If good hash and M<= N, cost is 1
  - If good hash and M> N, cost is M/N

  - Ratio of items/slots is called **load** $\alpha = M/N$

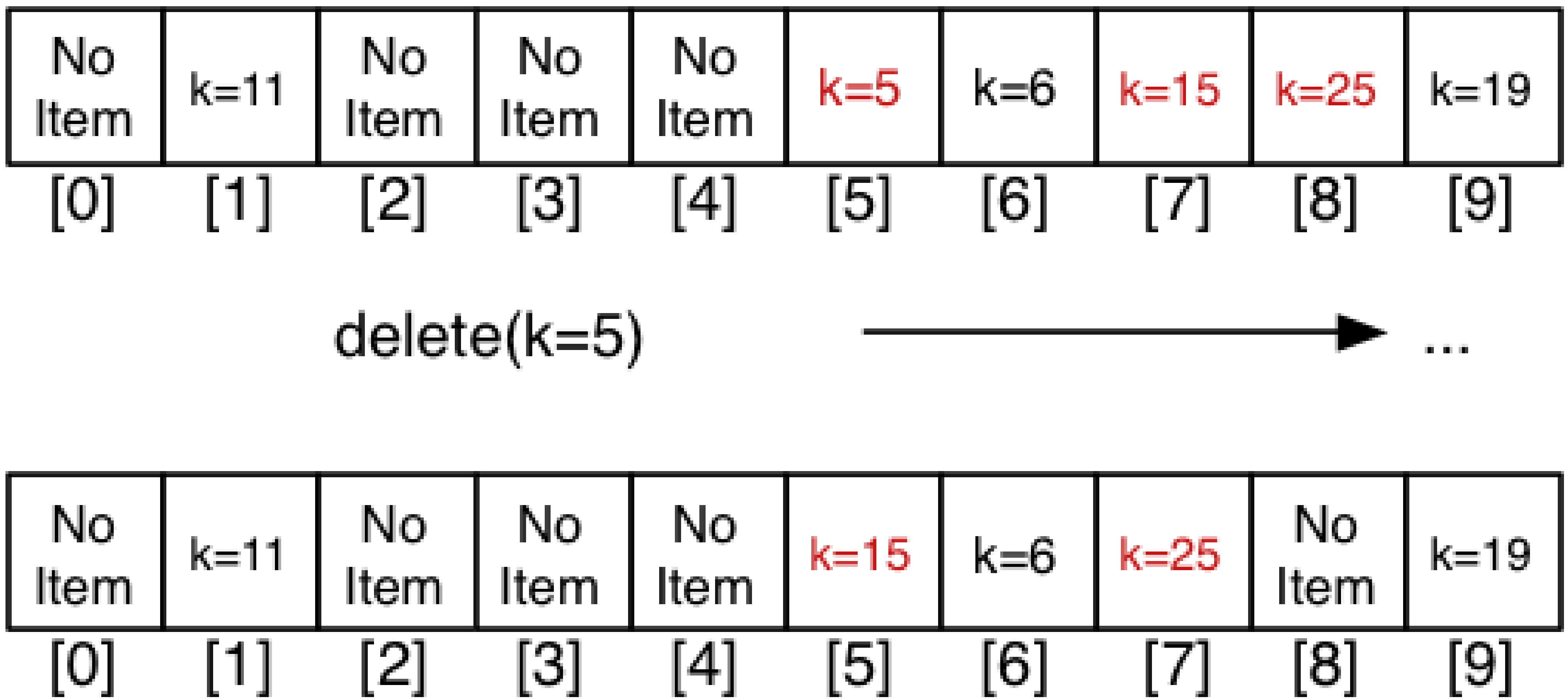
LINEAR PROBING

- Resolve collision in the primary table:
  - if the table is not close to be full, there are many empty slots, even if we have a collision
  - in case of a collision, simply use the next available slot
  - this is an instance of open-addressing hashing



Linear Probing: Deletion

Need to delete and reinsert all values after the index we delete at, till we reach a slot with no value



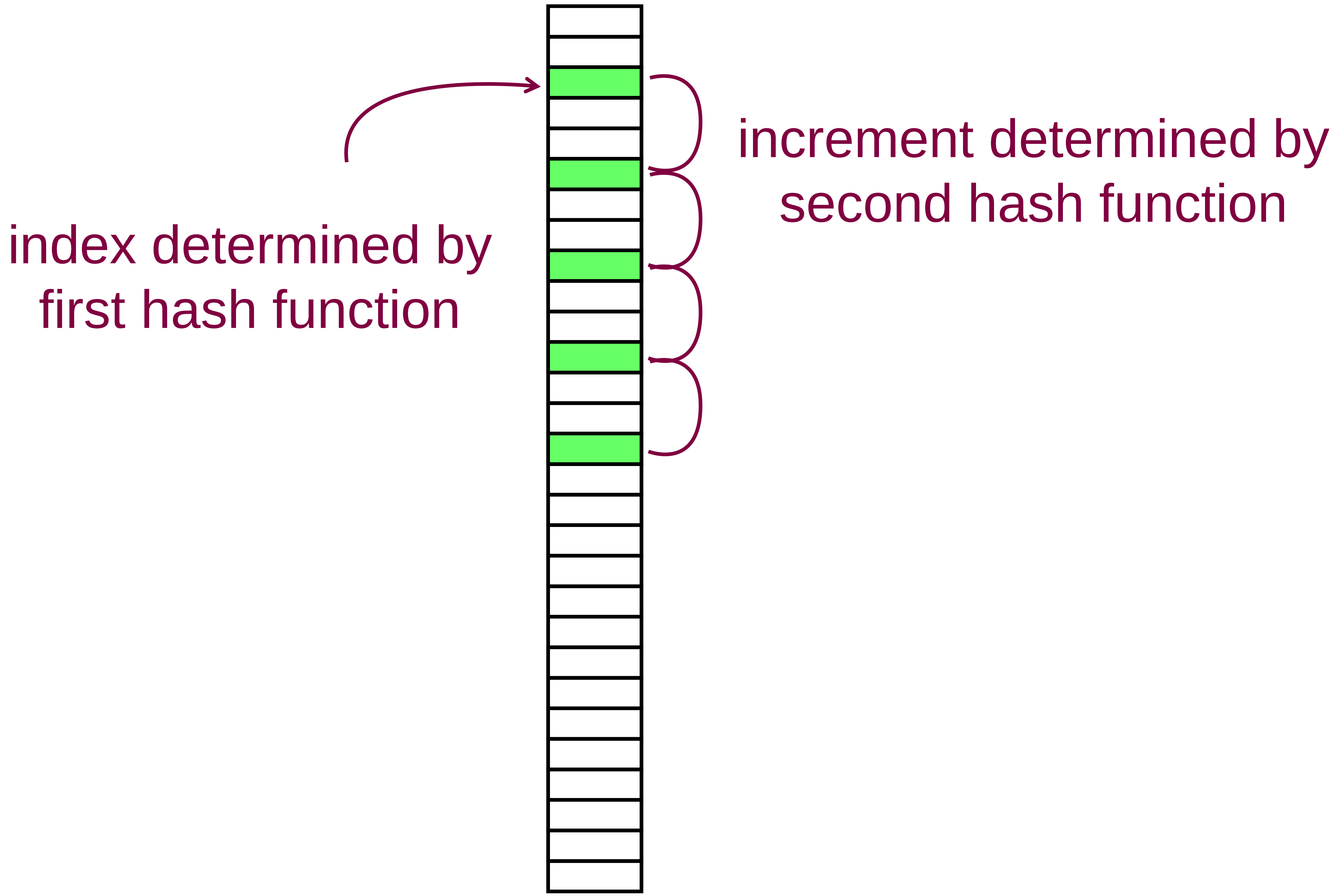
LINEAR PROBING

- Cost Analysis:
  - Cost to reach location where item is mapped is O(1), but then we may have to scan along to find it in the worst case this could be O(M)
  - affected by the load factor M/N

- Problems
  - When the table is starting to fill up, we can get clusters
  - Inserting an item with one hash value can increase access time for items with other hash values
  - Linear probing can become slow for near full hash tables

Double Hashing

- To avoid clustering, we use a second hash function to determine a fixed increment to check for empty slots in the table:



Double Hashing

- Requirements for second hashing function:
  - must never evaluate to zero
  - increment should be relatively prime to the hash table size
    - This ensures all elements are visited

- To generate relatively prime set table size to prime e.g. N=127
- hash2() in range [1..N1] where N1 < 127 and prime

- Can be significantly faster than linear probing especially if the table is heavily loaded.

DYNAMIC HASH TABLES

- All the hash table methods we looked at so far have the same problem
  - once the hash table gets full, the search and insertion times increases due to collisions
- Solution:
  - grow table dynamically
  - this involves copying of table content, amortised over time by reduction of collisions

EVALUATION

- Choice of the hash function can significantly effect the performance of the implementation, in particular when the hash table starts to fill up
- Choice of collision methods influences performance as well
  - linear probing (fastest, given table is sufficiently big)
  - double hashing (makes most efficient use of memory, req. 2nd hash function, fastest if table load is higher)
  - separate chaining (easiest to implement. table load can be more than 1 but performance degrades)