# Hash Tables

15-123

Systems Skills in C and Unix

# Questions?

- Why do we need hashing?

  Fast (O(1)) find & insert

- Can there be entries in the hash table with same key?

  No

- Can there be entries in the hash table with same value?

  Yes

- Can there be two entries in the hash table with same key and same value?

  NO

# Questions

- What would be a good table size to select given n keys to insert

  Closest prime $(n/5)$

- What is load factor? $= \frac{\#\text{ of entries}}{\#\text{ of occupied places in table}}$

- What would be a good load factor?

  $\approx 5$

- What would you do if the load factor is too high?

  ↓ bad hash function

  Rehash

# questions

- how would you select a hash function?

hashcode % tablesize

- How do you know if your hash function is a good one?

load factor ≈ 5

- Is it possible to pick a function that is 1-1? How difficult is it to find one?

yes

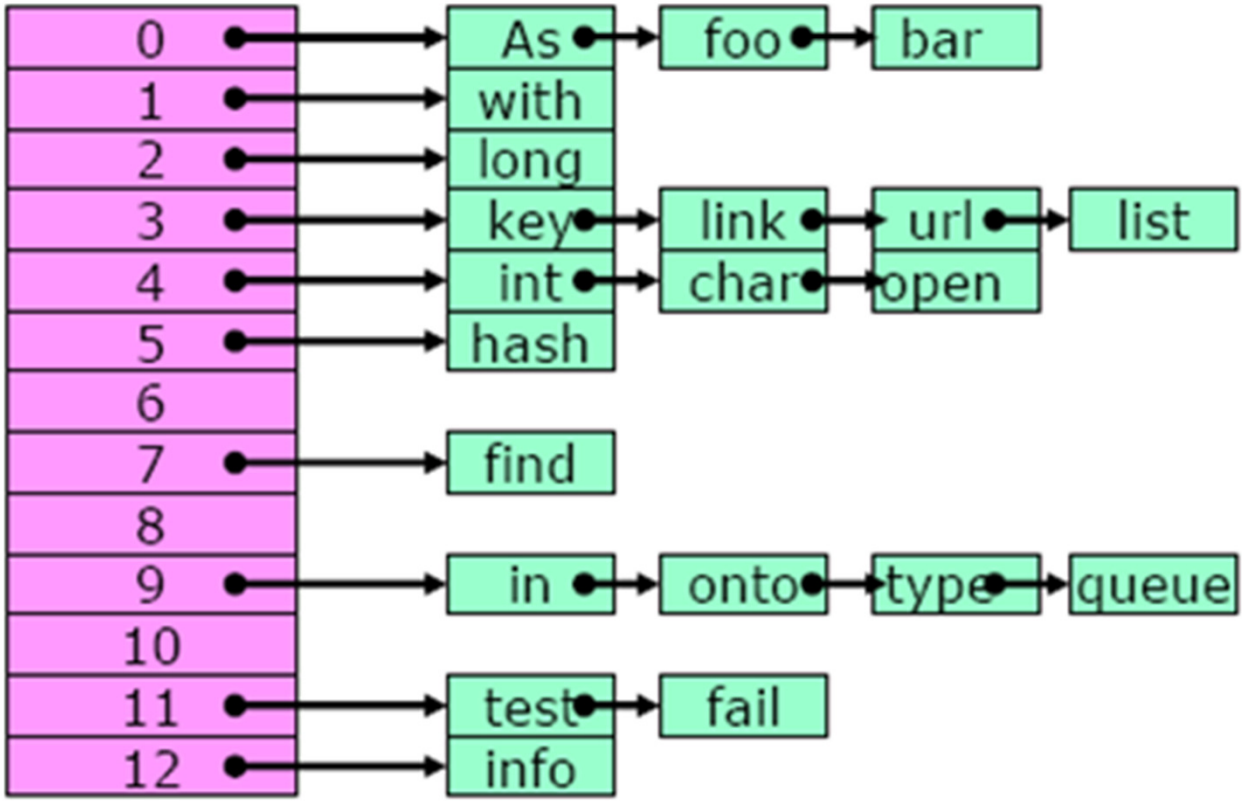
O(n!)

# What is a collision

- A collision occurs when two keys map to the same location
- Why do collisions occur?
  - Mainly due to bad hash functions
  - Eg: imagine hashing 1000 keys, where each key is on average 6 characters long, using a simple function like H(s) = ∑ characters, and a table size of at least 1001, how many collisions can be expected per cell (collisions occur only when the cell is taken and another key wants to map into the same place)

# How to resolve collisions

# Separate Chaining



# Separate Chaining

- Pros
  - No probing necessary
    - Each node has a place in the same hashcode
  - List gets never full
    - Performance can go down though
- Cons
  - Complicated implementation of array of linked lists
  - Still lots of collisions can create a “bad” hash table

# Load factor

- Need to keep the load factor reasonably under control
- If load factor becomes too large, rehash

# Rehash

- The process of creating a larger table to distribute the keys better

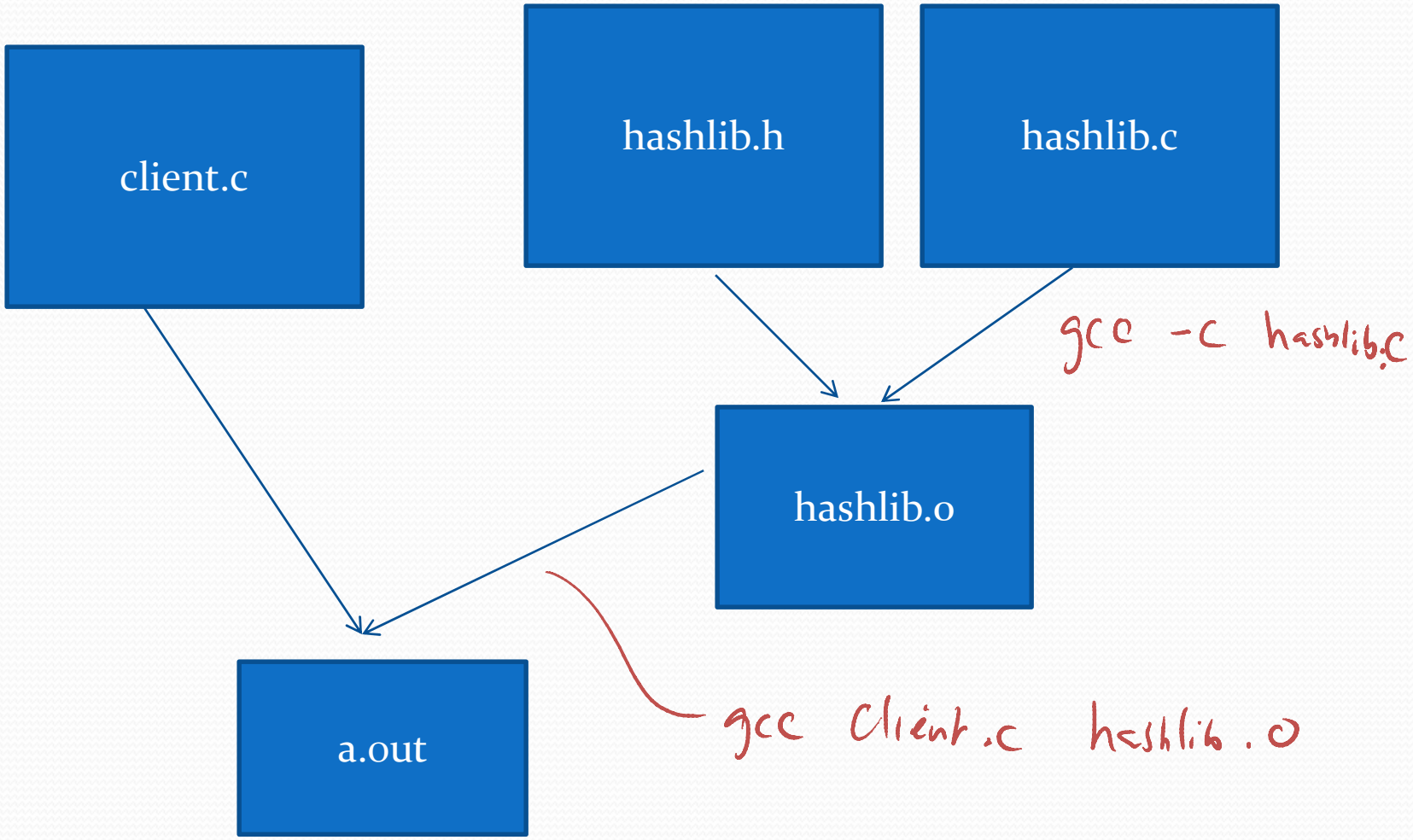
# Implementing a generic hash table

- Library design considerations
  - hash_node – a node that contains (key, value, next)
  - A struct that contains
    - Array of hashnode*'s
    - Size of the table
    - Function pointers
      - equal – compare two elems and return success (equal) or failure(not equal)
      - free_key, free_value

# Client considerations

- Must provide a hash function
  - It is also possible to provide a generic hash function like java API
- Must allocate memory for key and value (if necessary)

# Implementation



# Data Structures

```
typedef struct HASH_NODE {
  void *key;
  void *value;
  struct HASH_NODE *next;
} hash_node;


typedef struct hashtable {
        hash_node **table;
        int size;
        int (*equal)(const void*, const void*);
        void (*free_key)(void*);
        void (*free_value)(void*);
} hashtable;
```

# Library Interface

- ht_init (hashtable *ht, int size);
- ht_insert (hashtable *ht, void *key, void *value, int code)
- ht_retrieve (hashtable *ht, void *key, void **value)
- ht_rehash (hashtable *ht, int newsize);
- ht_set functions
  - equal, free_key, free_value

# Client implementation

```
int hashcode(void* s, int m) {
    /* this takes a pointer to a key and
        computes the hash code. m is string size
   */
}
```

# Code Examples