Hash table

What is hashing?

- To reduce the key space to a reasonable size
- Each key is mapped to a unique index (**hash value/code/address**)
- Search time remains O(1) on the average

hash function: {all possible keys} → {0, 1, 2, …, h-1}

- The array is called a **hash table**
- Each entry in the hash table is called a **hash slot**
- When multiple keys are mapped to the same hash value, a **collision** occurs
- If there are n records stored in a hash table with h slots, its **load factor** is $\boldsymbol{\alpha} = \frac{n}{h}$

PROPERTIES OF A *GOOD* HASH FUNCTION

- A good hash function should:
  1. Return indexes that fit within the size of the array
     - *i.e.,* [0 .. arrayLength-1]
  2. Be fast to compute
     - The hash function is a critical factor in access time
  3. Be repeatable (*i.e.,* always return same index) for a given key
  4. Distribute keys evenly over the full range of the array
     - This is to minimise collisions, a major issue in hash tables
- Properties 1–3 are easy to ensure, Property 4 is not
  - You don't know what keys you'll be getting in advance, so it's not possible to ensure they will be evenly distributed

COLLISION HANDLING METHODS

- OpenAddressing: Upon a collision, jump forward ('probe') a set amount to a new index and try again
  - If the new index is also used, repeat the probe until an empty index is found

- 1. Linear Probing – probe by step size of one every time
- 2. Quadratic Probing – probe forward by $(\text{probeNum})^2$
  - *i.e.,* probe first by 1, then 4, then 9, 16, 25, 36, 49, …
- 3. Double Hashing – use a *secondary* (different) hash function on the key to generate the probe step length

- Separate Chaining: Key-value pairs are added to a linked list anchored at the colliding hash index

DOUBLE HASHING

- Quadratic probing's increasing-step-size is an issue
  - Cannot guarantee that all slots will be visited
    - So if only one free slot, the quadratic stepping might miss it
      - Although hash tables aren't usually *that* full
- The secondary clustering is also an issue: inefficient

- Double hashing seeks to solve these problems
  - Calculate a step size based on the *key*
    - Each key will have its own step-size increment
      - Greatly reduces secondary clustering
    - Step-size for a given key won't change, thus with a prime-sized table it is *guaranteed* to be able to visit all slots
      - Because no common divisor between step-size and table-size

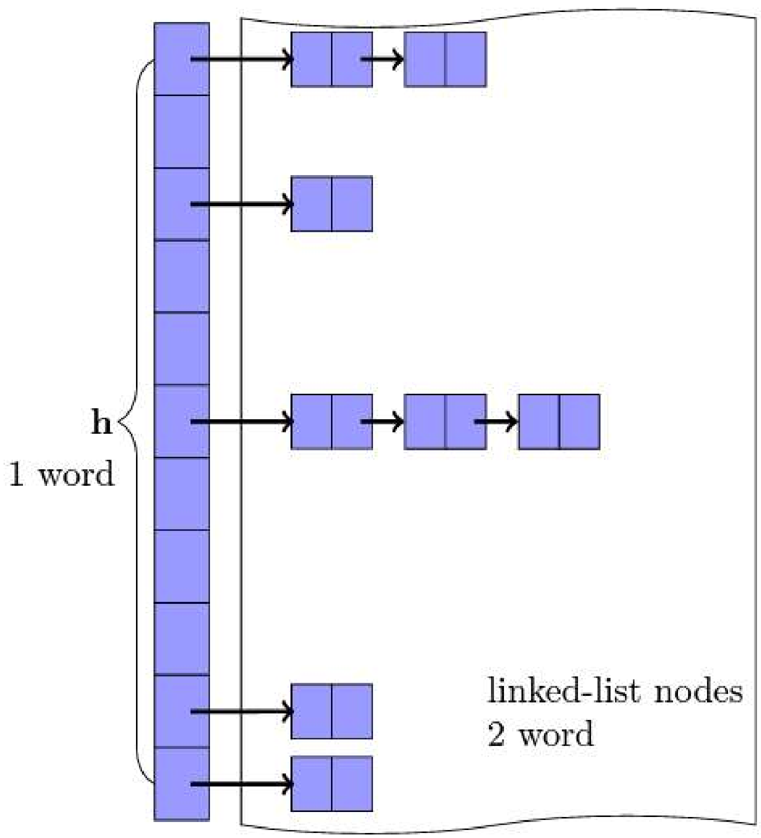
DOUBLE HASHING

- Step-size calc is done by a *second* hash function
  - Simple hash functions are good enough: we aren't overly concerned with even distribution since it's just step-size
  - Define a maximum step-size and make that the modulo
  - Must not produce 0 step sizes though
  - A good secondary hash function is:

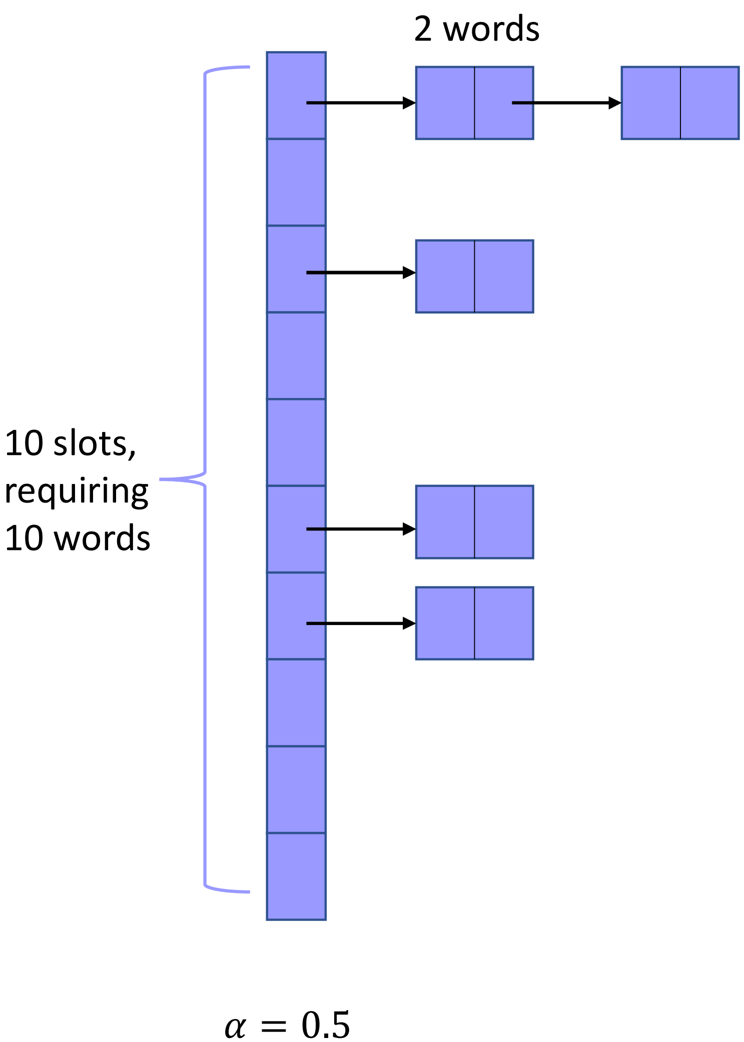
```
FUNCTION stepHash IMPORT key (integer)
                  EXPORT hashStep (integer)
 hashStep ← MAX_STEP – ( key % MAX_STEP ) // Step size will be between 1 and maxStep
```

  - MAX_STEP should be small-ish; certainly << table size!
  - Use a prime number for MAX_STEP

Q1

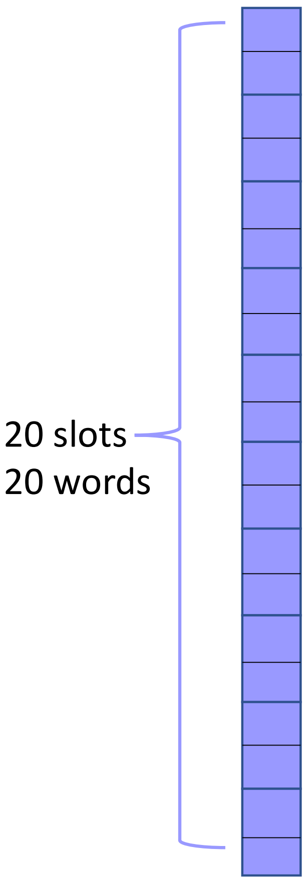


- The type of a hash table *H* under closed addressing is an array of list references, and under open addressing is an array of keys. Assume a key requires one "word" of memory and a linked list node requires two words, one for the key and one for a list reference.
- Consider each of these load factors for closed addressing: 0.5, 1.0, 2.0. Estimate the total space requirement, including space for lists, under closed addressing
- Assuming that the same amount of space is used for an open addressing hash table, what are the corresponding load factors under open addressing?



- Let *h* be hash table size. There are *h* slots.
- Load factor $\alpha = \frac{n}{h}$

1. When $\alpha = 0.5$, under closed addressing
   - *n*=0.5*h*, meaning there are 0.5*h* keys, which are 0.5*h* nodes.
   - Each node require 2 words.
   - Total space is $2\text{n} + h = 2 \times 0.5h + h = 2h$.
2. When *α*=1
   - There are *h* nodes
   - Total space: $h \times 2 + h = 3h$.
3. When *α=2*
   - There are 2*h* nodes
   - Total space: $2h \times 2 + h = 5h$.



- Assuming that the same amount of space is used for an open addressing hash table, what are the corresponding load factors under open addressing?

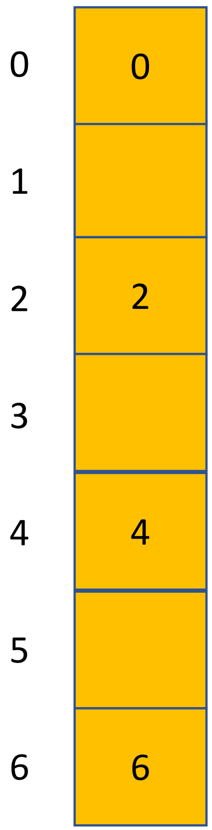
1. When there are 0.5*h* keys, and given 2*h* space, the corresponding load factor under open addressing is $\alpha = \frac{0.5h}{2h}$=0.25
2. When there are 1*h* keys, and given 3*h* space, $$\alpha = \frac{h}{3h} = 0.33$$
3. When there are 2*h* keys, and given 5*h* space, $$\alpha = \frac{2h}{5h} = 0.4$$

Q2

- Consider a hash table of size *n* using open address hashing and linear probing. Suppose that the hash table has a load factor of 0.5, describe with a diagram of the hash table, the best-case and the worst-case scenarios for the key distribution in the table.
- For each of the two scenarios, compute the average-case time complexity in terms of the number of key comparisons when inserting a new key. You may assume equal probability for the new key to be hashed into each of the *n* slots. [Note: Checking if a slot is empty is not a key comparison.]

n=7

$H(k)$ = k mod n



- Linear Probing: probe the next slot when there is a collision
  - $H(k,i)$ = $(k+i)$ mod n, where $i \in [0, n-1]$
- There are n slots, α=0.5, there are $n/2$ keys.
- Best case scenario:
  - The n/2 keys are hashed and distributed evenly into the n slots
- Assume that equal probability for a key to be hashed into each of the n slots, the average-case time complexity

$$= \frac{1}{n}\left(\sum_{i=1}^{\frac{n}{2}} 0\right) + \frac{1}{n}\left(\sum_{i=1}^{\frac{n}{2}} 1\right) = \frac{1}{n} \times \frac{n}{2} = 0.5 = \Theta(1)$$

n=7

$H(k)$ = k mod n

0	
1	
2	
3	3
4	10
5	17
6	24

- Linear Probing: probe the next slot when there is a collision
  - $H(k,i)$ = $(k+i)$ mod n, where $i \in [0, n-1]$
- There are *n* slots, α=0.5, there are *n*/2 keys.
- Worse case scenario:
  - The n/2 keys are hashed in consecutive slots. Each key always has to rehash and visit every key in the table. The ith key is hashed and rehash i times to get the slot.
- Average-time-complexity

$$= \frac{1}{n}\left(\sum_{i=1}^{\frac{n}{2}} 0\right) + \frac{1}{n}\left(\sum_{i=1}^{\frac{n}{2}} i\right) = \frac{1}{n} \times \frac{\frac{n}{2}\times(1+\frac{n}{2})}{2}$$

$$= \frac{n}{8} + \frac{1}{4} = \Theta(n)$$

Q3

- Each character in the string is assigned a numeric value, and the entire string is then interpreted as a number in radix 2^p
- For a string $S=C_1C_2C_3 \ldots C_n$, where each C_i is a character with a numerical value a_i, its numerical representation in radix 2^p is:
- $k = a_1 \times (2^p)^{n-1} + a_2 \times (2^p)^{n-2} + \cdots + a_n \times (2^p)^0$

Q3

- Let x and y be two strings that are permutations of each other. This means they contain the same characters, just arranged differently.

The numerical representations of x and y are:

- $k_x = a_1 \times (2^p)^{n-1} + a_2 \times (2^p)^{n-2} + \cdots + a_n \times (2^p)^0$
- $k_y = b_1 \times (2^p)^{n-1} + b_2 \times (2^p)^{n-2} + \cdots + b_n \times (2^p)^0$
- $Fact$: $(2^p)^i \equiv 1 \; mod \; 2^p - 1$, for any integer i. This means that every power of 2^p is congruent to 1 modulo $2^p - 1$.

Q3

- Applying Modulo $2^p - 1$
- $k_x \equiv a_1 + a_2 + \cdots a_n \, mod \, (2^p - 1)$
- $k_y \equiv b_1 + b_2 + \cdots b_n \, mod \, (2^p - 1)$
- As $a_1 + a_2 + \cdots a_n = b_1 + b_2 + \cdots b_n$
- $k_x \equiv k_y \, mod \, (2^p - 1)$
- $h(k_x) = h(k_y)$

Q3

- This property can lead to hash collisions in applications where order matters.
- Password Hashing"1234" and "4321" should produce different hashes, but under this scheme, they do not.
- Database Indexing "ABCD" and "BCDA" mapping to the same hash could cause incorrect lookups.
- Cryptographic Signatures: Digital signatures should change if even a small character swap happens.