## CSC 226

# Algorithms and Data Structures: II Rich Little rlittle@uvic.ca

## Analysis of Hash table Access

- Given a hash table of size N containing n elements
  - > We saw that the running time for a put () with chaining is O(1)
  - > We also saw the runtime for get () is O(size of list)
  - This is O(n) in the worst-case.
- Some degree of clustering is inevitable with any hashing scheme.
  - ➤ If the hash function is chosen properly we can get away with this

#### Load Factor

- Assume our hash function, h, maps n keys to independent uniform random values in range [0, N-1].
- Let X be the number of items that map to index i in array A.
- Then, the expected value of *X*

$$E(X) = \frac{n}{N}$$

• The **load factor**,  $\alpha$ , of a hash table is the ratio of occupied slots to total slots (which is E(X)). <sup>3</sup>

## Expected Run Time

• **Theorem:** Under the assumption of uniformity, the *expected* size of the linked list in each index of a hash table of size *N* storing *n* keys is which is equal to the load factor.

$$\alpha = n/N$$

• **Corollary:** The expected run time of get() is O(expected size of list) which is

$$O(\alpha)$$

by above theorem.

#### Load Factor

- If the number of collisions is small, then searching, inserting, and deleting elements in a hash table take O(1) time
- To reduce the number of collisions, in addition to using a good hash function, we should make sure that the table does not get too full
- If it gets higher, we should extend the hash table and **rehash** all of its elements (i.e., remove all elements and re-insert the elements)
  - Look at this more when talking about open addressing

#### Advice

- Choosing a high quality hash function for a particular application usually requires some knowledge of the expected input data.
  - ➤ Simulations using sample data can be helpful in choosing a good function for particular inputs.
- *Random linear hash function:* For a hash table with *n* integer keys, the following rules of thumb often produce good results
  - $\triangleright$  Choose the table size to be a prime number p such that 1.5n < p.
  - $\triangleright$  Choose values 0 < a < p,  $0 \le b < p$  and randomly

$$h(k) = ak + b \bmod p$$

## Open Addressing

- **Open Addressing** collision resolution schemes store every key in a table index, using a *probing scheme* to find an available index when a collision occurs.
- **Linear probing** starts at the hash value h(k), then checks successive indices until an empty space is found. The probe sequence is

$$h(k)$$
,  $h(k) + 1$ ,  $h(k) + 2$ , ...

- $\triangleright$  where all values are taken mod the table size, N.
- Thus, at probe *i* the index checked is

$$(h(k) + i) \mod N$$

## Open Addressing

• Quadratic probing uses the sequence

$$h(k) + 0^2$$
,  $h(k) + 1^2$ ,  $h(k) + 2^2$ ,

• Thus, at probe *i* the index checked is

$$(h(k) + i^2) \mod N$$

• **Double hashing** uses two hash functions  $h_1(k)$  and  $h_2(k)$ , where  $h_2(k) \neq 0$  for any k. The probe sequence is

$$h_1(k)$$
,  $h_1(k) + h_2(k)$ ,  $h_1(k) + 2h_2(k)$ , ...

• Thus, at probe *i* the index checked is

$$(h_1(k) + ih_2(k)) \mod N$$

Index	Value
0	
1	
2	
3	
4	
5 6	
6	

$$h(k) = 2k + 4 \bmod 7$$

- **Exercise:** Insert the keys 1,3,4,10,18 into the table using the given hash function
- Resolve collisions with linear probing

Index	Value
0	18
1	
2	
2 3 4	3
4	10
5 6	4
6	1

$$h(k) = 2k + 4 \bmod 7$$

- Exercise: Search for key 10 in the resulting table. (i.e. perform get (10)).
- **Exercise:** get (17).

Index	Value
0	18
1	
2	
3	3
4	10
5 6	4
6	1

$$h(k) = 2k + 4 \bmod 7$$

- **Exercise:** get (17).
- To search for a key, probe successive indices starting at the initial hash value until the key is found or an empty space is reached.

Index	Value
0	18
1	
2	
3	3
4	10
2 3 4 5 6	4
6	1

$$h(k) = 2k + 4 \bmod 7$$

- Exercise: remove (1).
- **Exercise:** get (18).

Index	Value
0	18
1	
2	
3	3
4	10
2 3 4 5 6	4
6	

$$h(k) = 2k + 4 \bmod 7$$

- Exercise: remove (1).
- **Exercise:** get (18).

#### Remove function

- For separate chaining, delete element in the linked list.
- For probing, mark element as *deleted* in the hash table since there might be elements following the deleted element in the open addressing probing chain.
- The usual protocol is to replace the key with a sentinel 'invalid element' marker, which is treated as an empty space during put () operations but treated as an element during get () operations.

Index	Value
0	18
1	
2	
3	3
4	10
2 3 4 5 6	4
6	×

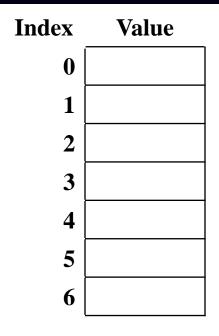
$$h(k) = 2k + 4 \bmod 7$$

- Exercise: remove (1).
- Exercise: get (18).

## Linear Probing

- Note that when we did the get (17) operation we scanned every key in the table.
- This was due to clustering.
- One disadvantage of linear probing is that if a region of the table becomes clustered, every index in that region will suffer from long probing sequences.
- One way to alleviate this problem is to make the 'step size' of a probing sequence vary in some way based on the starting point.

## Quadratic Probing Example



$$h(k) = 2k + 4 \mod 7$$

- **Exercise:** Insert the keys 1,3,4,10,18 into the table using the given hash function
- Resolve collisions with quadratic probing

## Quadratic Probing Example

Index	Value
0	
1	
2	18
2 3 4	3
4	10
5 6	4
6	1

$$h(k) = 2k + 4 \mod 7$$

- The table elements are all still clustered in a group of consecutive indices.
- In this case, the clustering is mainly caused by the table size being too small.
- Quadratic probing does result in shorter probe sequences for the 18 get () operation, though.

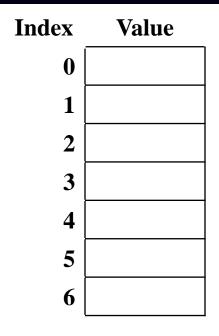
## Quadratic Probing Example

Index	Value
0	
1	
2	18
2 3 4	3
4	10
5 6	4
6	1

$$h(k) = 2k + 4 \bmod 7$$

• **Exercise:** get (17).

## Double Hashing Example



$$h_1(k) = 2k + 4 \mod 7$$
  
 $h_2(k) = 1 + (k \mod 6)$ 

- **Exercise:** Insert the keys 1,3,4,10,18 into the table using the given hash function
- Resolve collisions with double hashing

## Double Hashing Example

$$h_1(k) = 2k + 4 \mod 7$$
  
 $h_2(k) = 1 + (k \mod 6)$ 

• **Exercise:** get (17).

#### Load Factor

- In separate chaining,  $\alpha$  is the average length of a chain.
  - $\triangleright$  can be > 1 and still efficient
- for open addressing, α is the % of occupied cells
   has to be < 1</li>
- Proposition: Linear probing with *n* elements in a table of size *N*, the average number of probes is

$$\leq \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)$$
 for search hits

$$\leq \frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right)$$
 for search misses

#### Load Factor

• So, if  $\alpha = 1/2$ , then the expected number of probes for a search hit is

$$\leq \frac{1}{2} \left( 1 + \frac{1}{1 - \frac{1}{2}} \right) = \frac{1}{2} (1 + 2) = \frac{3}{2}$$

• The expected number of probes for a search miss is

$$\leq \frac{1}{2} \left( 1 + \frac{1}{\left(1 - \frac{1}{2}\right)^2} \right) = \frac{1}{2} (1 + 4) = \frac{5}{2}$$

• Theorem: The expected running time for doing a get (), put () or remove () in a hash table of size N, containing n items, with a load factor of  $\alpha \leq 1/2$ , is O(1).

## Rehashing

- As we have seen, the load factor,  $\alpha = n/N$ , has a big impact on the performance of a has table.
- For a table using chaining, as long as the load factor is a small constant (near 1), our methods run in O(1) time.
- For tables using probing, it needs to be sufficiently smaller than 1 (near  $\frac{1}{2}$ ).
- In both cases as the load factor exceeds these limits the performance drops off quickly.
- It turns out that it's worth it to **rehash** when this happens.

## Rehashing

- **Rehashing** Double the size of the table and apply a new hash function to every element.
- It may seem that by doing this we are now running in O(n) time for each of our methods but this is actually not the case.
- It turns out that the **amortized** running time for each of the hash table methods is still O(1).
- Why? The cost of each regular put () is O(1). After n regular puts there is a put that costs O(n) time.
- So, all together you have n puts that cost O(n) time in total. Thus, each costs O(1) time.

## Other Collision Strategies

- Separate chaining with binary trees
- Coalesced hashing separate chaining in place
- Cuckoo hashing new keys "push" old keys elsewhere
- Hopscotch hashing constant number of neighbor buckets, move empty buckets closer
- Robin Hood hashing displace key based on probe counts