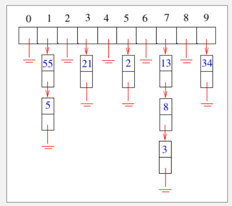
Hashtables Recap

- a good hash function h(k) –
  - is efficient to compute
    - ideally time is not dependent on |*k*| or *n*, but dependent on |*k*| is often effectively Θ(1)
  - spreads out the keys as much as possible
    - equal probability of h(k) = each value 0..N-1
- even with a good hash function, *collisions* (different keys hashing to the same array index) are inevitable
- two solutions
  - separate chaining
  - open addressing



0	1	2	3	4	5	6	7	8	9
34	5	55	21		2		3	8	13

Open Addressing

- requires n ≤ N

If h(k) is full, follow a *probe sequence* to locate element / find first empty slot for insertion.

- linear probing – h(k) + c·i [c is often 1]
  - c should be relatively prime to N (not a problem if N is prime)
  - *sequential probing* when c=1
- quadratic probing – h(k) + i^2
- double hashing – h(k) + i h'(k)

Open Addressing

Deletion requires special handling.

- can re-insert all elements following the deleted element
  - if the load factor is low enough, this should only be a small number of elements
- can mark empty slot as "deleted" – find continues on, insert can fill
  - drawback: probe sequence lengths are based on the largest the collection has been, not the current size
  - solution: can periodically re-hash everything to clean up

Perform the following operations on a hashtable of size 7 under the scenario listed, showing the contents of the hashtable after each step: insert 35, insert 10, insert 18, insert 24, insert 5, insert 11, delete 10, delete 24, delete 11, insert 74

- sequential probing, using hash function v%7

– linear probing – h(k) + c·i [c is often 1]
c should be relatively prime to N (not a problem if N is prime)
sequential probing when c=1

– quadratic probing – h(k) + i^2

– double hashing – h(k) + i h'(k)

Open Addressing

- linear probing – h(k) + c·i [c is often 1]
  - exhibits better memory locality than other options
  - suffers from clustering
    - keys that hash to the same index or adjacent indexes interfere with each other
    - performance degrades quickly as n approaches N
  - sensitive to key distribution
    - uneven key distribution exacerbates the clustering problem
- quadratic probing – $h(k) + i^2$
  - suffers from secondary clustering
    - keys that hash to adjacent slots have adjacent probe sequences
  - may not find an empty slot even if one exists
- double hashing – h(k) + i h'(k)
  - expected length of unsuccessful probe sequence is 1/(1-α) → O(1) if table is not too full
    - α = n/N (load factor)

Hashtables

If done properly, hashtables provide O(1) expected time for find, insert, remove – once h(k) has been computed.

- "done properly" means load factor isn't too high and is kept bounded, and there is good distribution of hash values

Computing h(k) can take time.

- e.g. for strings, computing h(k) = O(|k|) … which reduces to O(1) if |k| is bounded, but must be considered as O(|k|) otherwise

Worst-case behavior is O(n) for find and remove, unless separate chaining + a fancier bucket implementation is used (which has memory overhead).

- worst case occurs when key distribution is poor and load factor is high

Hashtables

What about other operations?

- initialization
  - O(N) – size of the array used for the hashtable
- traversal
  - in most cases O(n+N) for separate chaining – must examine each index of table as well as all elements
    - can be worse e.g. worst case dynamic perfect hashing
  - O(N) for open addressing
- find next larger/smaller key, find min/max key
  - full traversal is required because h(k) does not preserve original ordering of keys

Questions

How does the type of thing (`double`, `int`, `String`, object, etc) affect the running time?

- it doesn't, as long as only simple steps are involved
  - e.g. assignment is a simple step regardless of type – primitive types hold the value, object types hold the reference
  - e.g. copy is not necessarily a simple step – time to copy a `String` or array depends on the length

---

- typically the running time is expressed in terms of *n*, the number of elements in the collection
- there may be other factors which don't depend on *n* but which also aren't exactly constants
  - e.g. hashing a `String` depends on the length of the string, not the number of elements in the hashtable
  - keep those other quantities in the big-Oh unless you know they are bounded