Hashtables

Balanced search trees provide O(log n) find, insert, remove.
But can we do better?

O(1) would be the logical goal to strive for.
But how?

Observations.

- find is presumably the most commonly-used operation for Map, so it should be most efficient
- arrays have O(1) lookup by index

So – can we find a way to convert a key to an integer array index in O(1) time?

Hashtables

Let N be the size of the array.

- key → index is easy if the key is already an integer 0..N-1

Otherwise use a *hash function* h(k) to convert key k to an index.

- e.g. h(k) = k mod N if k is an integer
- e.g. $h(k) = \sum a^{|k|-(i+1)} \text{char}(k_i) \bmod N$ if k is a string
  - a = size of the alphabet
  - char(c) maps c to an integer 0..a-1

Hash Functions

Challenges.

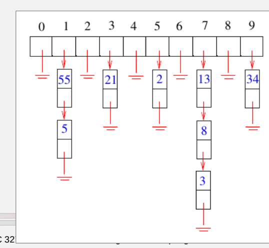
- h(k) must be efficient to compute, since it must be computed for every find, insert, remove operation
  - h(k) = k mod N → O(1)
  - $h(k) = \sum a^{|k|-(i+1)} \text{char}(k_i) \bmod N$ → O(|k|)

  Must factor in this time if not O(1) – though it often depends on something which is in practice a constant with respect to n.
- h(k) typically maps a large range of key values into the much smaller range 0..N-1 so collisions may occur
  - should spread keys over indexes as evenly as possible
    - choosing N to be a reasonably large prime helps with this
      - (but there is a tradeoff – larger N means more space for hashtable)
  - sensitive to particular distribution of keys in a given application

Collision Resolution

What to do with two elements whose keys hash to the same value?

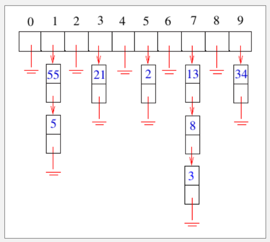
- separate chaining – store a list of elements at each slot in the array
- open addressing – find an alternate slot if the desired one is full



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

Separate Chaining

- operations
  - find – compute h(k), then search that list for desired key
  - insert – compute h(k), then add to that list
  - remove – find + remove from list



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

- chaining, using hash function v%7

Separate Chaining

- expected size of each list is n/N
  - assuming hash function distributes keys well
  - reduces to O(1) if n ≤ N or is never more than a fixed multiple of N i.e. hashtable is not too full
- typical implementations use unsorted linked lists
  - insert – O(1)
  - find, remove
    - expected O(n/N) if keys are well distributed
      - reduces to O(1) if n/N is bounded (e.g. n < N)
    - worst case O(n) if all keys hash to same index
  - can add move-to-front heuristic if some keys are searched for more frequently than others
  - overhead for storing pointers

Separate Chaining

- what about sorted linked lists?
  - can't exploit binary search with linked lists, but approximately halves the cost of an unsuccessful search for find, remove
  - insert O(n/N)
- what about arrays?
  - find is faster if sorted (binary search) but then have cost of shifting on insert/remove
  - still have space overhead (empty slots to avoid frequent shrinking/growing) + time overhead (shrinking/growing)

Separate Chaining

- more sophisticated implementations – array-based
  - eliminate space overhead – use an array of size k for a list of k elements (*dynamic array*)
    - no linked list pointers or empty slots
    - can exploit hardware features that provide greater efficiency for dealing with sequential memory positions
    - adds cost of array resizing on insert, remove
  - eliminate search through chain – use a hashtable of size k^2 for a list of k elements with a perfect hash function (no collisions), rebuilding when a collision occurs (*dynamic perfect hashing*)
    - guaranteed O(1) worst-case find
    - low amortized insert time – rebuilding is infrequent because load factor of secondary tables is 1/k
    - with N = O(n), expected total space is O(n), worst case $O(n^2)$

Separate Chaining

- more sophisticated implementations – other data structures
  - O(log n) operations – balanced search tree
    - O(log n) worst case for find, insert, remove
    - additional overhead not generally worth it except in special cases
      - e.g. high load factor (n/N ≥ 10)
      - e.g. likely non-uniform hash distribution (some long chains)
      - e.g. need to guarantee good performance in worst case
    - using a larger hash table or finding a better hash function may be better alternatives

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