Map/Dictionary Implementation Recap (So Far)

Dictionary operation	Unsorted array	Sorted array	Singly linked unsorted	Singly linked sorted	balanced BST
Search(A, k)	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	O(log n)
Insert(A, x)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	O(log n)
Delete(A, x) or Delete(A,k) (given location of *x*)	$O(1)^*$	$O(n)$	O(1) *	O(1) *	O(log n)
Remove(A,x) or Remove(A,k) (not given location of *x*)	O(n) requires search + delete	O(n)	O(n)	O(n)	O(log n)

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

Hash Functions

Challenges.

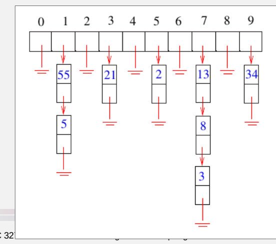
- h(k) must be efficient to compute, since it must be computed for every find, insert, remove operation
  - h(k) = k mod N → O(1)
  - $h(k) = \sum a^{|k|-(i+1)}\, char(k_i) \bmod N$ → O(|k|)

  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?

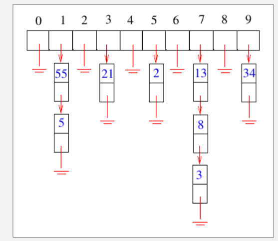
- separate chaining – store a list of elements at each slot in the array
- open addressing – find an alternate slot if the desired one is full



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



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)
    - insert at head
  - 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, 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 ($n/N \geq 10$)
      - 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