Hash Tables

November 25, 2019 8:27 AM

Dictionary

- Worst case comlexities for Insert, Remove, Find

	Insert	Remove	Find
Unordered array	$O(1)$	$O(n)$	$O(n)$
Ordered array	$O(n)$	$O(n)$	$O(\log n)$
Unordered list	$O(1)$	$O(n)$	$O(n)$
Ordered list	$O(n)$	$O(n)$	$O(n)$
BST	$O(n)$	$O(n)$	$O(n)$

- hash table consists of an array to store data
  - consists of complex types or pointers
  - one attribute of object designated as table's key
- hash function maps a key to an array index in 2 steps
  - key should be converted to integer
  - integer mapped to an array index using some function
- **Collisions**
  - a hash function may map two different keys to same index = collision
  - a good hash funciton can reduce number of collisions
  - have a policy to deal with any collisions
- **Hash functions**
  - map key values to array indexes
  - Map key value to an integer, then map integer to legal array index
  - Properties:
    - Fast (close to O(1)
    - Deterministic (no randomness)
    - Uniformity (look like a uniform random distribution)
- Deterministic hash functions
  - for given input, should always return same value, otherwise will not be found in hash table
  - Hash functions should not be determined by system time, memory location, pseudo-random numbers
- Scattering data
  - typical hash function usually results in some collisions
  - data should be distributed evenly over the table
  - reduce number of collisions, each search key value maps to different index
- Uniformity
  - generate each value in output range wiht same probability
  - each legal hash table index has same chance of being generated

Collision Handling

- 2 different keys mapped to same index, inevitable due to pigeonhole principle
- **Open Addressing**
  - when insertion results in collision, look for empty array element
    - start at index to which hasch func mapped inserted item
    - look for free space in array following particular search pattern, probing
  - Three major open addressing schemes:
  - **Linear probing**
    - hash table searched sequentially, starting with original hash location
    - each time table probed for free location, add 1 to index
    - search h(search key) + 1, h(search key) + 2

        - if sequence of probes reaches last element, wrap around to arr[0]
        - Primary clustering
            - table contains groups fo consecutively occupied locations, larger as time goes on, reduce effieciency of hash table
        - ex. Hash size 23, h(x) = x*mod23, x = search key val
            - insert 81, h= 81*mod 23 = 12, collides with 58, so use linear probing to find free space, 12+1=13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81								21	

        - **Searching:** ex. Find 59, h=59*mod23 = 13, index 13 does not contain 59, use linear probing to find 59 or an empty space, 13+1=14. Empty, therefore 59 not in table
        - **Efficiency:** load factor y = number of items/table size
        - as table fills, y increases, chance of collision increases, performance dec, as y inc
    - **Quadratic probing**
        - refinement of linear probing, prevents primary clustering
        - each p, add p^2 to original location index
        - 1st: $h(x) + 1^2$, 2nd: $h(x) + 2^2$, 3rd: $h(x) + 3^2$
        - secondary clustering: same seq of probes used whent 2 diff values hash to same location
        - no problem if data not skewed, hash table large enough hash func scatters data cross the table
        - may fail at y > 1/2
    - **Double hashing**
        - linear and quadratic probing the probe seq is independent of key
        - double hashing procedures key dependent probe sequences
        - a second hash func h_2 determines probe sequence
        - h_2(key) != 0, h_2 != h_1 h_2 is p - (key mod p), where p is a prime number
        - ex. hast table size 23, h = x mod 23, h_2 = 5 - (key mod 5)
        - insert 81, h = 81 mod 32 = 12, collides with 58, use h_2 to find probe sequence value: h_2 = 5-(81 mod 5) = 4, so insert at 12 + 4 = 16

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58				81					21	

- **Separate Chaining**
    - each entry in the hash table is a pointer to a linked list
    - if collision occurs, new item added to end of list at appropriate location
    - performance degrades less rapidly using separate chaining

- $h(x) = x \bmod 23$
- Insert 81, $h(x) = 12$, add to back (or front) of list
- Insert 35, $h(x) = 12$
- Insert 60, $h(x) = 14$