Hashing Summary Notes (Supplementary) (Part I -IV)

Hashing (implementation of ADT *Symbol Table)*  
🡪 No successor/predecessor query  
🡪 Load α = n/m = E(items per bucket) //[n items, m buckets]  
🡪 Hash function: h(key): U -> {1…m}  
🡪 2 solutions to collision:   
1) Chaining  
2) Open Addressing  
  
**Good Java Hash Function**- Defined hashCode(), Override equals()  
- objects that are equals (including itself) must return same hash code  
1) Enum all possible buckets  
2) Uniform Hashing Assumption (all different permutations same probability)

1) Chaining (m buckets, L size of linked list)  
🡪 Given SUHA (equally likely allocation, independent)  
- Insert operation : O(1 + cost(h))  
- Search operation : O(L + cost(h)) = O(n/m + 1)  
- worst case search : O(n)

//growing and shrinking   
- Insert operation: amortized O(1)   
- Search operation: expected: O(1)

2) Open Addressing (i.e. *Linear Probing/Weird Probing*)  
🡪 Given UHA (perms equally likely, allocation independent)  
- Probe a sequence of buckets until you find empty bucket  
- h(key, i): U -> {1…m} //i is number of collisions  
- Operation Cost <=   
- Prob: Operation cost degrades badly as α approaches 1  
-> Cost of Growing / Shrinking Table : O(m1 + m2 + n)  
- E(insert) = O(1)

**Chaining**  
Assume) Simple Uniform Hashing Assumption (SUHA)  
Assume)Hashing with Chaining  
- E(search time) = O(1 + n/m), {O(1) IF m = Ω(n) buckets}   
- E(search time) = Worst case: O(n)   
- Worst case (insertion): O(1)  
- Inserting n items, Expected max cost: O(logn)  
-> if (m < 2n): too many collisions  
-> if (m > 10n): too much wasted space  
=> Prob: don’t know n in advance  
=> Idea: Grow and shrink table when needed

How to grow table:  
1. Choose new table size m  
2. Choose new hash function h  
- hash function depends on table size (unlike Java hashCode)  
- Remember h: U -> {1…m}  
3. For each item in old hash table  
- Compute new hash function  
- Copy item to new bucket

Time Complexity of growing table:  
- Assume: [m1 size of old hash table, m2 size of new hash table, n number of elem]  
- Costs: Scan old table [O(m1)], Create new table [O(m2)], Insert each elem in: [O(1) per elem]  
- Total: O(m1 + m2 + n)  
-> if (n==m): m = m + 1 🡪 Cost: O(n) per resize,   
 Total Cost of inserting n items + resizing (from table size of 7) [7+8+9+…n] 🡪 O(n^2)   
-> if (n==m): m = 2m 🡪 Cost: O(n) per resize,   
 Total Cost of … [ = 7 + 15 + 31 + … + n] 🡪 O(n)  
Average Cost of inserts: O(1) [most O(1), some O(n)]

Shrinking table  
- if (n == m), then m = 2m [Every time you double table of size m, at least m/2 new items were added]  
- if (n < m/4), then m = m/2 [Every time you shrink a table of size m, at least m/4 items were deleted]

Amortized Analysis  
- Operation as amortized cost T(n) if for every integer k, the cost of k operations is <= k \* T(n)  
- Not equal to average cost (cos has to be for EVERY integer) so if insert sequence is 13 5 5 5 5 7 cannot say amortized cost of 7, but if it is 5 5 5 13 7 can  
-> Inserting k elements into a hash table takes O(k) -> Amortized cost of insert is O(1) [given insertions from 0]

Accounting Method  
- Avg time per operation = money / num of operations  
- Total cost: Inserting k elements   
-> Deffered dollars: O(k) [to pay for resizing  
-> Immediate dollars: O(k) [to pay for inserting k elem]  
=> Total (Deferred + Immediate): O(k)  
🡺 Cost Per Operation: O(1) / operation [deferred dollars: O(1), immediate dollars: O(1)]

Conclusion: Hashing with Chaining  
- with SUHA (Simple Uniform Hashing Assumption)  
🡪 Insert operation: amortized O(1)  
🡪 Search operation: expected: O(1)  
\*Note difference between randomization analysis and amortized analysis, expected vs amortized

**Set** (ADT) {Set<Key>}  
- methods: insert(k), contains(k), delete(k), intersect(Set<K> s), union(Set<K> s)  
🡪 No defined ordering  
🡪 SPEED & SPACE is critical

Implementing a Set

1. Fingerprint Hash Function  
   - Doesn’t store the key, only store 0/1 vector  
   - Doesn’t prevent collisions  
   - No false negatives, potential false positives  
   -   
   - Trade off:

* Reduced space: only 1-bit per slot
* Increased space: bigger tables to avoid collisions.

1. Bloom Filter

* More than 1 hash function (use k hash functions)
* Reduces collission
* As k increases, P(false positives) decrease -> Less false positives
* Optimal k can be calculated []
* Trade off:  
  -> Each item takes more ‘space’ in table  
  -> Requires k collision for false positives