Hash Tables

-maps keys to values (keys: unique integer that is used for indexing the values, value: data that is associated with a key)

-combine the random access ability of an array with the dynamism of a linked list

-Insertion, search, deletion take O(1) time

-hash tables are pretty bad for ordering and sorting data (time becomes O(n)) -> ordering is not supported

-difficult to specify the size of the hash table (if i make the Hash table too small then i get more collisions and performance gets bad, make them large enough (2n or 5n or 10n)

-hash tables are not good for very dynamic environments (use trees instead if we have lots of insertions and deletions)

-hash tables cannot be used for efficient range queries since they do not guarantee that the keys are in sorted order (unlike binary search trees). Hash tables are very efficient but don't allow to search values in an interval or other range queries (for that we can rather use trees (BSTs or RB trees)).

-a Hash table is a combination of:

-a function: Hash function, which returns an nonnegative integer value called hash code

-an array capable of storing the data type we with to place into the data structure

-simple uniform hashing: any given item is equally likely to hash into any of the m slots of the hash table (evenly distributed)

-h(k) can be computed in constant time (independent from number of elements): best case when n = 1 and n = 1Mio so alpha is 1/1Mio and it is theta of 1, worst case when n = 1Mio and m = 1 so alpha is 1Mio which is n and it is theta of n->in worst case a hash function becomes a list

-A hash table should not contain duplicates (if we have same number two times we keep the position of the already inserted equal number)

-A good hash function:

can't prevent the collisions completely, however it can reduce the number of collisions.

should be fast to calculate (computation in O(1) time)

the keys should be distributed uniformly (not all keys should be mapped to the same slot but each slot is mapped to the same number of keys)

the distribution is random: patterns in the distribution of the keys (e.g. if all the keys are even) should not affect uniform distribution of keys into slots.

Is deterministic: when we pass the same input to the hash function, it always generates the same index for that value.

1) Division Method:

-If k is a key and m is the size of the hash table, the hash function h() is calculated as: h(k) = (k mod m) + 1

For example, If the size of a hash table is 10 and k = 112 then h(k) = 112 mod 10 = 2. The value of m must not be the powers of 2.

2)Multiplication Method:

$h(k) = \lfloor m(kA \bmod 1)\rfloor + 1$ (kA mod 1 gives the fractional part kA, A is any constant and its value lies between 0 and 1)

An optimal choice for A would be $\approx (\sqrt{5}-1)/2 = 0.618$ (suggested by Knuth)

When the hash function generates the same index for multiple keys, a collision occurs:

-presumably we want to store both of data in the hash tale, so we shouldn't simply overwrite the data that was inserted there first.

2 Techniques: -collision resolution by chaining (closed addressing), -open addressing(linear/quadratic probing and double hashing)

closed addressing: stores multiple elements into the same slot (no overflow chains), open addressing: doesn't store multiple elements in the same slot (each table slot is either empty or contains an element).

In open addressing, we do not want a step size of i = 0 since all elements would go all in the same slot and in case of double hashing the second hash function would be useless.

M should not be a power of 2.

When i becomes equal m, we know that the hash table is full!

<u>Chaining (combinations of array and linked lists)</u>

-idea of each element of the array holding multiple pieces of data (for example if data type is a linked list)

-efficient: we know that insertion with linked lists is a O(1) operation

-for lookup, we only need to search through a small list, since we're distributing what would otherwise be one huge list across n lists

-Each slot contains either NIL or a pointer to the head of the list in that slot.

-**Insertion is always done at the head of the linked list** (and not at the tail!, this is because in this way insertion takes O(1) time and not O(n)$\rightarrow$efficiency and time complexity advantage)

<u>Linear Probing</u>

-if we have a collision, we try to place the data in the next consecutive element in the array (wrapping around to the beginning if necessary) until we find a spot.

In linear probing, collision is resolved by checking the next slot.

-->$h(k, i) = (h'(k) + c*i) \bmod m$ (c is often = 1: guarantees that all the locations are probed)

Linear probing is subject to a problem called clustering. Once there is a miss, two adjacent cells will contain data, making it more likely in the future that the cluster will grow.

<u>Quadratic Probing</u>

-It works similar to linear probing but the spacing between the slots is increased (greater than one) by using the following relation:

$h(k, i) = (h'(k) + c1i + c2i2) \bmod m$

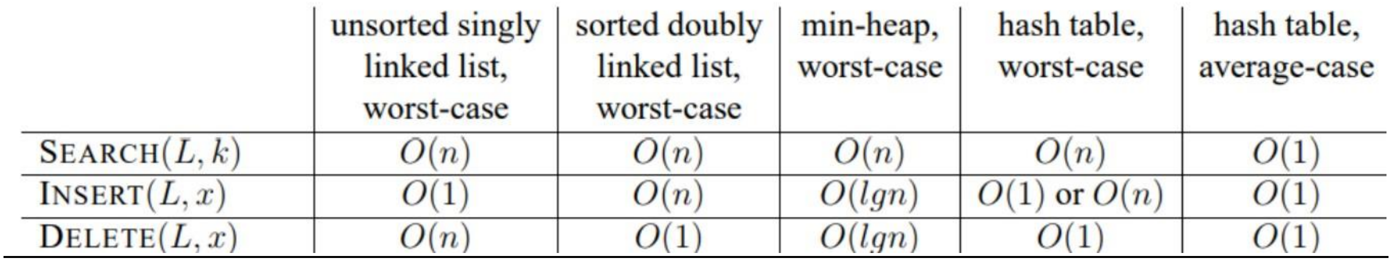
Double Hashing

-If a collision occurs after applying a hash function h(k), then another hash function is calculated for finding the next slot.

-->h(k, i) = (h1(k) + i*h2(k)) mod m

H2(k) must be relatively prime (no common factors in the factorization) to hash table size m.

Hash Tables: Efficiency



	unsorted singly linked list, worst-case	sorted doubly linked list, worst-case	min-heap, worst-case	hash table, worst-case	hash table, average-case
SEARCH(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$
INSERT(L, x)	$O(1)$	$O(n)$	$O(lgn)$	$O(1)$ or $O(n)$	$O(1)$
DELETE(L, x)	$O(n)$	$O(1)$	$O(lgn)$	$O(1)$	$O(1)$