Hashing

→ It is mainly used to implement dictionary. Where you have key value pair, as well it also used when you set-

08

→ Search
09 Insert — O(1) on average
delete

10

→ Not useful for
11 (a) finding closest value } AVL trees (or) Red black tree.
(b) sorted data
12 (c) Prefix searching. } trie

01 # Application of Hashing.
(1) Dictionaries
02 (2) Database Indexing
(3) cryptography
03 (4) Caches
(5) Symbol tables in compilers/ Interpreters
04 (6) Routers
(7) caches
05 (8) getting data from databases
and so on.

06

2nd most used DS in Computer Science.

07

Direct Address Table

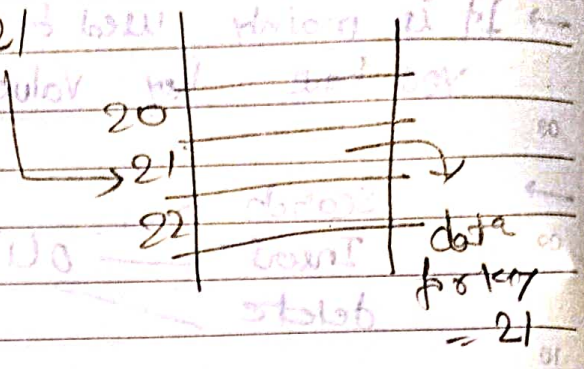
DAT is a DS that has the capability of mapping record to their corresponding keys using arrays. In direct address tables, records are placed using their key values directly as indexes. These facilitate fast searching, insertion and deletion operation.

MAY
05
'24
19th Week
126-240
SUNDAY

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

Searching
Insertion } = O(1) time
deletion



Limitions :-
1. Prior knowledge of maximum key value.
2. Practically useful only if the maximum value is very less.
3. It causes wastage of memory spaces if there is a significant difference b/w total records and maximum value.

Hashing can overcome these limition of direct address table.

Hash Functions

Hashing is a techniques (or) process of mapping keys and values into the hash table by using a hash function. It is done for faster access to elements.
The efficiency depend on the hash function.

The task of hash function to convert those large keys, string, floating point number into small values that can be used as an index in the hash table.

(a) Should always map a large key to same small keys.
(b) Should generate values from 0 to m-1.
(c) Should be fast, O(1) for integers and O(len) for string of length 'len'.
(d) Should uniformly distribute large keys into hash tables slo

Example Hash function.

① h(large_key) = large_key % m

② for string (weight sum) str = 'abcd'

eg (str[0] * n^0 + str[1] * n^1 + str[2] * n^2 + str[3] * n^4) % m

③ universal hashing.

↳ We choose Prime No. which are not close to Power.

→ Set of hash function, we have to pick randomly and your hash function

Collision handling

Birthday Paradox 23 men

(*) Collision is the conditions in which two keys map to the same entry in the hash table.

(*) If we know keys in advance, then we can go with perfect hashing

(*) If we don't know keys, then we use one of the following.

Following are the ways to handle collision

- chaining
- open addressing
  - Linear Probing.
  - Quadratic Probing.
  - double hashing.

Chaining (Separate chaining):

→ The idea behind Separate chaining is to implement the array as a linked list called a chain.

→ The linked list data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these element are inserted into a singly linked list which is known as chain.

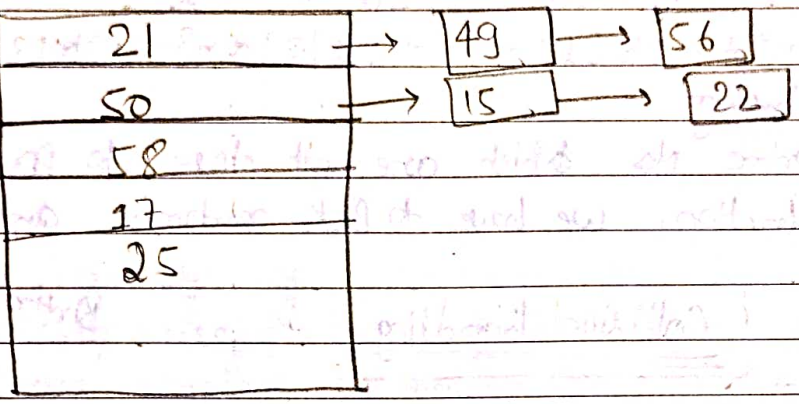
MAY
07
'24
19th Week
128-238
TUESDAY

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

hash(key) = key % 7

keys = {50, 21, 58, 17, 15, 49, 56, 22, 23, 8, 5}



Performance

m = No. of slots in hash table.
n = No. of keys to be inserted.
Load factor $\alpha = n/m$

Expected chain length = α
Expected time to search = $O(1+\alpha)$
" " " insert/Delete = $O(1+\alpha)$

Data Structure to storing chains

(a) Linked list — search $O(l)$, Delete $O(l)$, insert $O(1)$ (not cache friendly)
(b) Dynamic Size arrays (cache friendly) (vector in c++, Array list in Java, list in Python) — TC = $O(l)$
(c) Self balancing of BST (AVL tree, Red black tree) (cache friendly)
$O(\log l)$ — search, insert, delete

Implementation of chaining in Python

BUCKET = 7

0	[ ]
1	[ ]
2	[ ]
3	[ ]
4	[ ]
5	[ ]
6	[ ]

Insert → 70, 71, 9, 56, 72

70	56
71	
9	72
[ ]	
[ ]	
[ ]	
[ ]	

[[ ], [ ], [ ], [ ], [ ], [ ], [ ]]

[hash = n % BUCKET]

[[70, 56], [71], [9, 72], [ ], [ ], [ ], [ ]]

```
h = MyHash(7)
h.insert(70)
h.insert(71)
h.insert(9)
h.insert(56)
h.insert(72)
Print(h.search(56))
h.remove(56)
print(h.search(56))
o/p - True
      False
```

```
class MyHash:
    def __init__(self, b):
        self.BUCKET = b
        self.table = [[] for _ in range(b)]
    def insert(self, n):
        i = n % self.BUCKET
        self.table[i].append(n)
    def remove(self, n):
        i = n % self.BUCKET
        self.table[i].remove(n)
    def search(self, n):
        i = n % self.BUCKET
        return n in self.table[i]
```

MAY
09
'24
19th Week
130-236
THURSDAY

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

Frequencies of Array Elements

i/p — arr[] = {10, 20, 12, 10, 15, 10, 12, 12}

o/p — 10 3
12 3
15 1
20 1

i/p — arr[] = {10, 10, 10, 10}

o/p — 10 4

i/p — arr[] = {10, 20}

o/p — 10 1
20 1

(a)

```
def countFreq(arr, n):
    mp = dict()
    for i in range(n):
        if arr[i] in mp.keys():
            mp[arr[i]] += 1
        else:
            mp[arr[i]] = 1
    for x in mp:
        print(x, '', mp[x])
```

TC <= O(n^2)

Wise-men never sit and wait their loss

Set (s)

08 → A set is an unordered collection data types that is iterable, mutable and has no duplicate elements.

09 (a) Distinct element
(b) unordered
10 (c) No indexing
(d) union, intersection, set difference etc are fast.
11 (*)(e) uses hashing Internally.

12 S.add()
S.update()
01 S.discard()
S.remove()
02 S.clear()
len(S)

$S_1 = \{2,4,6,8\}$, $S_2 = \{3,6,9\}$
Print($S_1 | S_2$) — {2,3,4,6,8,9}
Print($S_1 \& S_2$) — {6}
Print($S_1 - S_2$) — {8,2,4}
Print($S_1 \wedge S_2$) — {2,3,4,8,9}

03 Intersection difference Symmetric difference

04 union
→ S_1.union(S_2) S_1.intersection(S_2) S_1.difference(S_2) S_1.symmetric_difference(S_2)

05 Dictionary Key: Value pair

06 → Dictionary is a collection of keys values, used to store data value like a map, which, unlike other data types which holds only single value
07 as an elements.
(a) unordered.
(b) All keys must be distinct.
(c) Value may be repeated.
(*)(d) uses hashing internally.

d = {110: 'xyz', 101: 'abc', 105: 'bcd', 104: 'abc'}
→ Similar to hashmap in Java, unordered map in c++.

MAY
11
'24
19th Week
132-234
SATURDAY

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

d.get(101) → 'xyz'
d.get(125) → None
len(d) → 4
d.pop(105) → {110: 'abc', 101: 'wxy', 106: 'bcd'}
del d[106]
d.popitem(1)

Open addressing

Like separate chaining, open addressing is a method for handling collision. In open addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys. (Note that we can increase table size by copying old data if needed) This approach is also known as closed hashing.
The entire procedure is based upon probing. We will understand the types of Probing ahead. (Cache friendly)

(a) Linear Probing.
Linearly search for next empty slots.
rehash(key) = (h+1) % table-size;

0	49
1	50
2	51
3	16
4	56
5	15
6	19

Search in circular manner for next empty slot:
50, 51, 49, 16, 56, 15, 19
hash(Key) = Key % 7

insert(50), insert(51), insert(15), search(15), search(64), delete(15) search(15)

Search — We compute hash function we go to that index and compare. If we find, we return true. Otherwise we linearly hash.

→ We stop when one of the three cases arise

(i) Empty slot — False
(ii) Key found — True
(iii) Traversed through the whole table — False

Delete — Problem with simply making slot empty when we delete. So we don't marked the slot empty we marked it as deleted.

Q→ Primary clusting is problem with linear probing.

To handle primary clusting problem with linear probing is

$h(key) = key \% 7$

$hash(key, i) = (h(key) + i) \% 7$

(1) Quadratic Probing

$hash(key, i) = (h(key) + i^2) \% m$

(2) Double hashing

$hash(key, i) = (h_1(key) + i \times h_2(key)) \% m$.

(a) Secondary clusting suffer by Quadratic Probing.
(b) It might happend that it doesn't find empty slot even if there are empty slot

(1) $\alpha < 0.5$ if load factor is less than 0.5 then only you will able to find empty slot.
(2) m is prime in (n/m).

if two these things happen together then quadratic probing works and they will finds empty slots.

MAY
13
'24
MONDAY
20th Week
134-232

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

(b) **Quadratic Probing:**

Quadratic Probing is a method with the help of which we can solve the problem of clustering that was discussed above.
This method is also known as the mid square method.
In this method, we look for the i^2th slot in the ith iteration.
We always start from the original hash location. If only the location is occupied then we check the other slots.

(c) **Double hashing:**

Double hashing is a technique that reduces clustering in an optimized way. $hash(key, i) = (h_1(key) + i\,h_2(key)) \% m$
In this technique, the increments for the probing sequence are computed by using another hash function.
We use another hash function $hash2(x)$ and look for the $i * hash2(x)$ slot in the ith rotation.

(a) If $h_2(key)$ is relatively prime to m, then it always find a free slot if there is one.
(b) Distributes keys more uniformly than linear probing and quadratic hashing.
(c) No clustering.

$hash(key, i) = (h_1(key) + i\,h_2(key)) \% m$

49, 63, 56, 52, 54, 48

0	49
1	
2	54
3	63
4	56
5	52
6	48

$m = 7$

$h_1(key) = (key \% 7)$

$h_2(key) = 6 - (key \% 6)$

$56 \rightarrow (0 + (1 * 4)) \% 7 = 4$

$52 \rightarrow (3 + (6 - (52 \% 6))$

$(5 - 2)$

1 ✓

$54 \rightarrow 5 + (6 - 54 \% 6)$

$5 + 6 = 11 \% 7 = 4$

Will a wild cat observe the fast of Sivaratri?

Performance Analysis of search

(a) Cache Performance of chaining is not good because when we traverse a linked list, we are basically jumping from one node to another, all across the computer's memory.

Like chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table.

m = No. of slots in hash table

n = No. of keys to be inserted in hash table.

Load factor $\alpha = \frac{n}{m}$ (< 1)

Expected time to search / insert / delete $< \frac{1}{(1-\alpha)}$

$(1-\alpha)$ fraction of the table is empty.

Unsuccessful search means when you have either traverse through the whole table or you ended up with a empty slot.

Chaining	Open Addressing.
(a) It is simple to implement	(a) It requires more computation.
(b) Hash table never fills up, we can always add more elements.	(b) It may becomes full.
(c) Less sensitive to hash function	(c) Extra care required for clustering.
(d) Cache performance of chaining is not good as keys are stored using linked list.	(d) It provide better cache performance as everything is stored in same table
(e) extra spaces are required for links $(1+\alpha)$	(e) might be needed to achieve same performance $= [1/(1-\alpha)]$
→ better technique	→ old technique
→ It is used when frequency and No. of keys are not known may dynamically increase.	→ It is used when frequency and No of keys are known.

When poverty come in the door, love goes out the window

MAY
15 '24
20th Week
136-230
WEDNESDAY

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

Implementation of open addressing in Python

```
Class myhash:
    def __init__(self, c):
        self.cap = c
        self.table = [-1]*c
        self.size = 0

    def hash(self, n):
        return n%self.cap

    def search(self, n):
        h = self.hash(n)
        t = self.table
        i = h
        while t[i] != -1:
            if t[i] == n:
                return True
            i = (i+1)%self.cap
            if i == h:
                return False
        return False

    def insert(self, n):
        if self.size == self.cap: return False
        if self.search(n) == True: return False
        i = self.hash(n)
        t = self.table
        while t[i] not in (-1, -2):
            i = (i+1)%self.cap
        t[i] = n
        self.size = self.size + 1
        return True

    def remove(self, n):
        h = self.hash(n)
        t = self.table
        i = h
        while t[i] != -1:
            if t[i] == n:
                t[i] = -2
                return -2
            i = (i+1)%self.cap
            if i == h:
                return False
        return False
```

o/p - True
False

```
h = Myhash(7)
h.insert(70)
h.insert(71)
h.insert(9)
h.insert(56)
h.insert(42)
print(h.search(56))
h.remove(56)
print(h.search(56))
h.remove(56)
```