for hashing, the order mentioned is on avg., & not the worst case. for worst case these, orders can blown upto n.

Date: / /
Page No.:

# Dynamic Data set (DDS)

We want to perform the following operations:-

| | Linked list (LL) | Array (A) | Sorted array (SA) | Hashing ↓ O(1) is avg not the worst case. | Balanced Binary Search Tree (BBST) |
|---|---|---|---|---|---|
| → Add(x) | O(1) | O(1) | O(n) | O(1) | O(log n) |
| → Delete(x) | O(1) - O(n) | O(1) - O(n) | O(n) | O(1) | O(log n) |
| → Search(x) | O(n) | O(n) | O(log n) | O(1) | O(log n) |

↳ ⇒ doesn't help much

∴ to delete X, you have to 1st search for X & moreover in array you have shift everything by one position, after deleting.

↓

~~needed only for sorted array~~

for general array delete X & put the last element at the position of X, since the order doesn't matter

Ex:- | 6 | 8 | 9 | 2 | 1 | & we have to delete '8'

the final array will be | 6 | 1 | 9 | 2 |

Order n is really bad, if we do n such operations, it suddenly blows upto to order $n^2$.

Date: / /
Page No.:

(*) In BBST, all the 3 operations can be done in order $\log n$, but if you just want to do add, delete & search, it's still worth taking risk & dealing with hashing, probably it's NOT worth going for BBST.

But if there are more operations that we want to do, we will see more examples later on. However one of the operations worth mentioning is Rank(X)

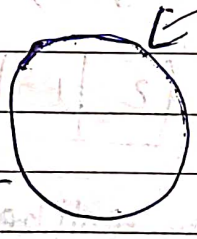
Rank(X) will return the no. of numbers $\geq X$ for a dynamic data set

↓

we keep adding, deleting & there is also a search.

| | Hashing | BBST |
|---|---|---|
| Rank(X) | $\Theta(n)$ | $\Theta(\log n)$ |

To start with hashing lets look at a concrete example. We're given a set of telephone numbers



10 digit no.

Lets say that no. of telephone numbers $= 10^6$

This is not too large a no. b/c the no. of airtel users in Bang. is much larger than this.

Date: / /
Page No.:

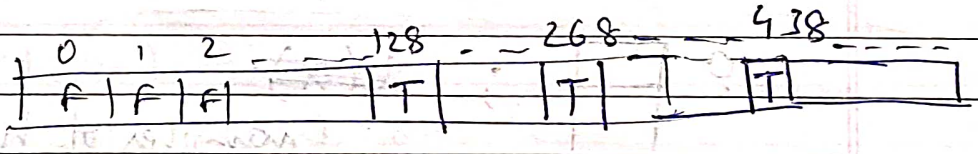
So we want to maintain a data str. which is ~~dynamic~~ dynamic, you should be able to add new no., delete any existing no. if needed & return true or false if asked that a certain no. exists in tel. directory or not.

First, now look at a similar en. we have a set of internal (college) telephone no. Here the no. of users $\leq 10^3$. Hence, a three digit no. will do the work. ~~However, let's say that we're 4 digit no.~~
Let no. of user $\approx 200$

Now, here we want to :-
add, delete & search

To do this we can use ~~an~~ array of size 1000 (say). If a no. corresponding to an index exists then set it to a boolean true, otherwise false.

ex:- 268, 128, 438 are the no.




∴ the amount of space needed = 1000

Add, Delete, Search } each of them happens in order 1.

It's just a table lookup.

→

However, if we compare this method for the prev. en., the no. of telep. no. ($10^6$) is not a problem. We can declare a array of ~~10~~ size $10^6$. But if we want to apply the same method as mentioned above we'll need an array of size $10^{10}$. (NOT possible)

Date: / /
Page No.:

Which is where hashing comes into picture.
Formal definition of hashing:-
hashing is a fn from a set S (a dynamic set), {for all practical purposes we'll assume S as an integer set. Ideally this is an infinite set.} to a set A.

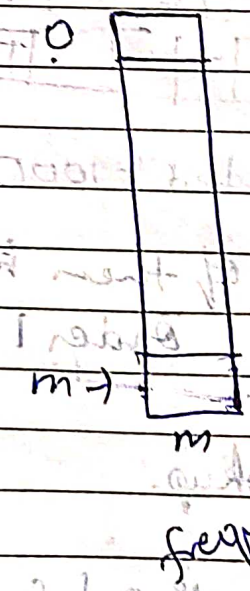
Q. Why can we assume that any data is an integer?

Sol$^n$ Even if we're given a book, we want to interpret that lets say as an integer. Even if you give me a 2000 pg book the way computer stores it is in the binary representation.

$A = \{0, 1, \ldots, m-1\}$

i.e. $h: S \to \{0, 1, \ldots, m-1\}$

where m is the size of the table.




Examples of hash fn$^n$:-

i) $h(x) = (ax+b) \bmod m$
$1 \le a, b < m$

In particular you can take
$b=0$ & $a=1$
$\Rightarrow h(x) = x \bmod m$
↓
This is a frequently used hash fn$^n$.

Date: / /
Page No.:

ii) $h(x) = \lfloor m\,(ax \bmod 1) \rfloor$ ; $0 < a < 1$

$(ax \bmod 1)$ → a fractional no. bet 0 & 1

$0 < a < 1$ → i.e. a is a fractional no. It may be .2, .3, .4 ---

when we multiply this by m, we'll get a fractional value b/w 0 & m-1. Just take the integer part of it

but this is NOT frequently used

$(x \bmod 1)$ will give us the fractional part of $x$, leaving the integer part of it.

for ex:- $|2.87\%| = .87$

→ —

Now, in the 10 digit teleph. no. ex n i.e. the no. of tel. no. is $10^6$.

i.e. $n = 10^6$

Q. The natural ques. arises is what should be the size of m?

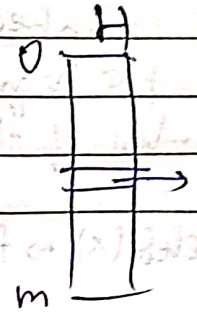
Soln Typically, we take $m = 2n$

Q. Why is it so?

Soln We want to store n no. So, we create a space of size 2n, so we have enough space to store these no. If we just take m smaller than n, the we can't obviously store everything in the table. So, just ensure that you make relatively a bigger table, so that we're able to store all those no.

→ —

Now, how to add → Add(x), we look at h(x) & store x at H[h(x)] i.e. at the index h(x)



27:43

Date: / /
Page No.:

Delete (x) → H[h(x)] = -1

i.e go to the index h(x) & make it as -1, -1 indicates that there is nothing. (∵ tel. no. are always +ve, so -ve no. are not possible)

Search (x) → check if H[h(x)] = x
If this is true then return true
otherwise, return false

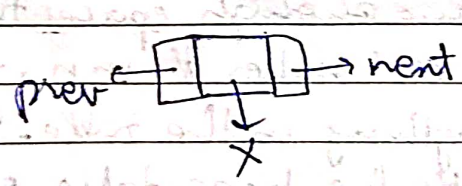
However, there exists a problem of collision when we add like this.

collision is when $x \neq y$ but $h(x) = h(y)$

To handle the problem of collision, we've techniques called collision resolution techniques:-

open addressing

~~One of which being is the~~ declaring array ~~as~~ an array of linked list (instead of simple array). where every index is pointing to NULL, when empty structure of each node will look like




Now, when we compute h(x), it will give a pointer to a linked list. Adding x will then be basically to add at the beginning of the linked list where index corresponding index of the array will be the head of the linked list

Delete(x) is find h(x), traverse through LL & delete that node when you find it.

Date: / /
Page No.:

Search(x) is find h(x), traverse through the LL & return true if you find the node x

Here, we have a problem that if a ~~malicious~~ naughty person knows ~~the~~ our hash fn i.e. a, b & m then he can provide us with a data where each one has the same hash value.

Real life eni:- About in 1995, Oracle had a database based on a hash fn & everybody knew what the hash fn is. So, on one fine day attackers hashed everything which mapped to the same location, the entire database crashed

So, declare what your hash fn is, is probably not the right way to do. So, if we have a large set of hash fn the I can ~~uniform~~ randomly pick a hash fn & work with that

So your program only says that I'm going to pick a random hash fn from a huge collection of hash fn, but what that hash fn is, nobody knows. That is decided on the runtime by the machine

this collection is called a universal collection of hash fn.

What is expected is that the no. of collision should be constant. Only then add, search & delete become order 1. Even if one of the location in hash table becomes $\frac{n}{2}$ or $\frac{n}{5}$ then the performing operations will become a linear time.

Date: / /
Page No.:

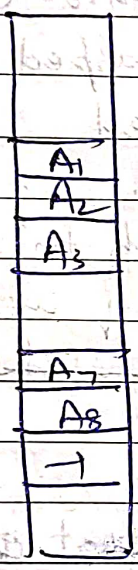
i.e. everytime we run the program a new hash table is created from scratch, so there should be a way to store it.

IF YOU WANT TO AVOID OPEN-ADDRESSING THEN YOU CAN USE EITHER OF THE FOLLOWING 3:—

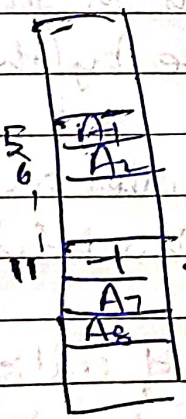
A.) Linear probing → Simplest of all

Let ~~A, B, C ...~~ $A_1, A_2, A_3 \ldots A_8$ have the same hash value 5

then we will store them like this




Now, suppose we want to search ~~delete~~ $A_7$ then we will go to index 5 & search for $A_7$ until we find it or we find (-1). But this can arise a problem. Suppose $A_6$ is deleted then the table will look like




& now we want to search for $A_7$. At 11th index the search will stop since we get -1.

So, to handle this each index should also have a flag value of True or False.

If the flag is true then, we will know that sth is deleted from that place and hence we continue our search.

Date: / /
Page No.:

Note :- if you want to search for a no. x then go to h(x) then go on searching for x until you find x or a (-1) with a false flag.

when you delete an x replace x with (-1) & set the flag to true.

when you add x, go to ~~is~~ inten h(x), if it's occupied just keep going forward until you find (-1) (doesn't matter whether flag is true or false) & insert x. If, when moving forward, you reach the end of the array, ~~come to zero th~~ ~~O th inten~~ ~~This is not of very efficient way to~~ ~~& continue the procedure.~~ then do (mod m).

probably
This is not a very efficient way when you consider the search & other thing but as far as the space is considered, it will only take n space, you don't have to create LL & other things. ~~Th~~

—x—

B) Quadratic Probing.

Everything is same like linear probing, except that instead of moving forward by one position we move forward by $i+i^2$

Ex:-

| | |
|---|---|
| 5 | $A_1$ |
| 7 | $A_2$ |
| : | |
| 11 | $A_3$ |

Initially i=0 then i=1

∴ ~~A~~ $A_2$ ~~will~~ will go to $h(A_2)+1+1^2 = h(A_2)+2$

$A_3$ will go to $h(A_3)+2+2^2 = h(A_3)+6$

Here also we've to take care of flag.

Date: / /
Page No.:

If $h(x)+i+i^2$ goes out of range then do (mod m).

C.) Double hashing

Everything stays same as quadratic probing. except that instead of moving forward by $i+i^2$, we move forward by $ih_1(x)$, where $h_1(x)$ is another hash fn.

If $h(x)+ih_1(x)$ goes out of range then do (mod m).

depending on $h_1$, mind that $h_1(x)$ can be -ve for any x, so instead of moving forward, actually we will backtrack. It's completely normal. Don't get psyched.

→ –

Even after knowing all these things, someone can give us a collection of no. which will map to the same location. In the open-addressing it all becomes a LL. In the cases above, it all becomes an array. However, operations in both ways will take order n time.

So, that's why wanted to choose the hash fn uniformly at random. In order to do this, we should have a large collection of hash fn with some nice ~~for~~ property. This collection is called Universal hash fn.

Date: / /
Page No.:

# UNIVERSAL HASHING

$\mathcal{H}$ → ~~set~~ symbol for collection of hash fn.

↳ ~~collection set~~.

↓

an infinite set of hash fn which all map to $\{0, \ldots, m-1\}$

This set is a universal hash fn if it satisfies the following property(s):—

$\forall\, x \neq y \quad (h(x) = h(y)); \quad h \in \mathcal{H}$

↓

The no. of such hash fn in $\mathcal{H}$ for any pair $x, y$ $\leq \frac{|H|}{m}$ ★

$m$ → size of the table

i.e. if the size of the hash table is smaller, then the no. of hash fn where the collision actually happens is more.

So, the no. of hash fn, where there will be a collision is directly proportional to the ~~no.~~ total no. of hash fn that you have & inversely proportional to the size of the hash table

Date: / /
Page No.:

of hash fn$^n$

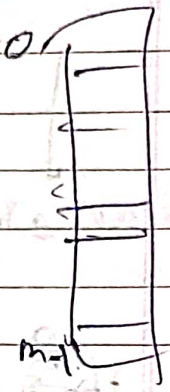
Q If we are given a set, then what is the probability ~~the~~ that there will be a collision at a position in the hash table?

Ans

$$\text{Probability} = \left(\frac{1}{m}\right) \times \binom{n}{2} = \frac{n(n-1)}{2m}$$

collision can occur at any one of the m places

any 2 no. that we choose can collide there




So if we take the size of the table in order of $n^2$ then

$$\text{Probability of collision} \leq \frac{1}{2}$$

but size of order $n^2$ is too much. B/c in all the examples, we saw that the assumption for the amount of space that we can use is of order n.

—x—

A universal hash fn$^n$ can be given in the following manner (no need to known us to why it is a universal hash fn$^n$)

$$H(x) = ((ax+b) \bmod p) \bmod m$$

a, b, p

$m \simeq 2n$ (of order $10^6$)

$p >> m$ → of order $10^9$ or greater

Date: / /
Page No.:

& ~~a, b~~ $0 < a, b < p$

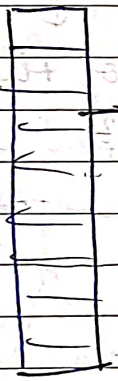
We'll use all these things to create, what is known as the perfect hashing.

## PERFECT HASHING

A hash fn is said to be a perfect hash fn if there is no collisions at all.

Well in advance, if we know the no. which we want to insert into the table then we can create nicely hash fn which will help us to create the perfect hash fn.

We have a simple algo. which includes 2 level hashing.



$n_i$ = no. of elements which will hash to the ith location.

$n$ = no. of elements to be hashed

$m = 2n$

$\Rightarrow \boxed{\sum n_i = n}$ *

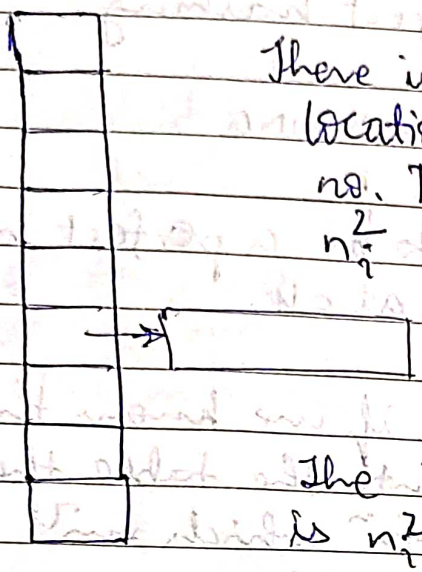
(II) The probability for choosing hash fn which will satisfy this propert $> \frac{1}{2}$

∴ on avg. we may have to try with 2 hash fn $\because \frac{1}{2} + \frac{1}{2} = 1$

We first compute $\sum n_i^2$, & if

summation of space used in the secondary level $\boxed{\sum n_i^2 \leq 4n}$ ~~then~~ for the chosen hash fn

(known as the primary hash fn.)
then the chosen hash fn is correct, otherwise choose another hash fn by choosing another p, a & b



There is another hash table at each location which will deal with $n_i$ no. The size of this hash table is $n_i^2$

$n_i^2$

The moment you tell me what is $n_i^2$, I can choose a hash fn uniformly at random b/c you just use the following

$h(x) = ((ax+b) \bmod p) \bmod m$
$a, b, p$

↓

just instead of this use $\bmod n_i^2$

So, we already saw that if $n_i^2$
⇒ Probability of collision in 2nd level hash table $\leq \frac{1}{2}$

Here also on avg. we may have to try 2 hash fn. $\frac{1}{2} + \frac{1}{2} = 1$

T

Howeves, if there is still collision in the 2nd level, we choose another hash fn.

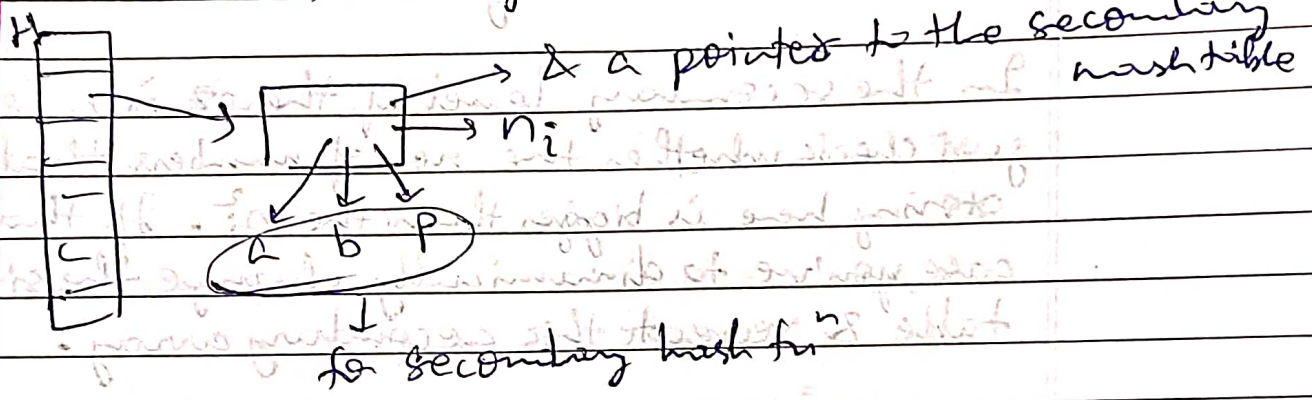
1st level hash table is primary hash table
2nd " " " " secondary hash table

Date: / /
Page No.:

i.e. the primary hash table will store the pointers to the secondary hash table. The secondary hash table will have different hash fn.

In the secondary level, whatever we're going to write is not a linked list. Each of them is a hash table. Hash table will have a fixed length i.e. $n_i^2$

So each index of primary hash table will have the following:-




For different secondary hash table at each index there may be different secondary hash fn

So, here the total space used will be

$$\underbrace{2n}_{m} + \underbrace{\leq 4n}_{\Sigma n_i^2} = \underbrace{\leq 6n}_{\text{size of order } n.}$$

∴ the complexity for finding hash fn for both level from (I) & (II) is $O(n)$

This is all what we do, when we're given the set of no. is already known to us. First we build this tables keep it. That is there're no entries but we

know what are the hash fn the we're gonna use. Now, if the entries are given, we can guarantee that there'll be no collision.

Q. But what to do if S is not given?

Sol^n We start doing the same thing. There're 2 invariants needed to maintain. One is that the $\sum n_i^2 \leq 4n$ & the 2nd thing worth mentioning is that at secondary level there is no collision.

In the secondary level if there is a collision, just check whether the no. of numbers that you're storing here is bigger than the $n_i^2$. If that is the case, you've to dinamically change the size of the table & recreate this secondary array.

The other thing that can go wrong is that $\sum n_i^2 > 4n$. In that case you've to change your entire hash fn from scratch. So, what you need to change is, you've to change your primary hash fn. The moment primary hash fn changes, things will change in the secondary hash fn also. Whatever you've stored, you've to delete everything & then add it again. So if we don't know anything about S, this is the best that we can do.

So, in most of the time insert will take only order 1 but occassionally, when you've to recreate the hash table from scratch, it will take order n. So you have to resize the hash

Date: / /
Page No.:

table & you also have to pick different hash table for different level. (that will take linear time) But that will not happen very frequently.

Maybe once in a while it happens. So if you do n operation, only some of them will be order n. So overall it's expected that the complexity will be $O(n)$.