NPTEL
NPTEL ONLINE CERTIFICATION COURSE

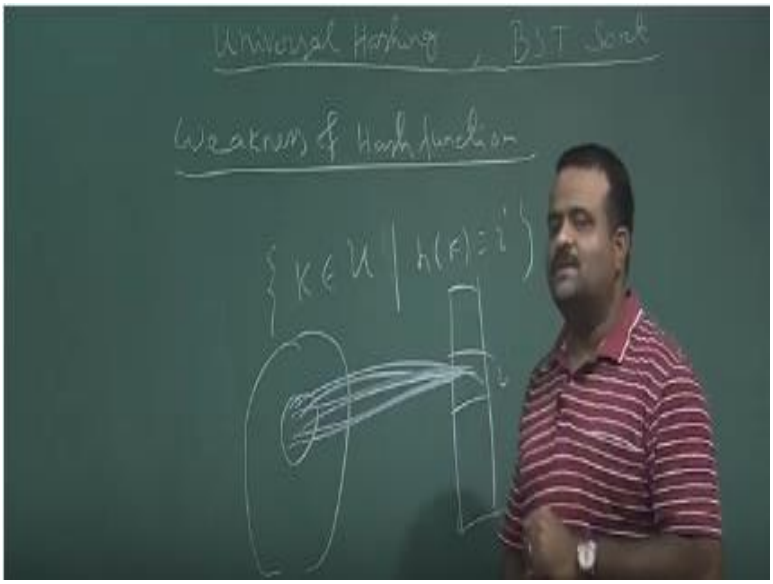
Course Name
Fundamental Algorithms:
Design and Analysis

by
Prof. Sourav Mukhopadhyay
Department of Mathematics
IIT Kharagpur

Lecture 10: Universal Hashing, BST Sort

Okay, so we will talk about universal hashing and then we will talk about BST sort by then there are two sort in this module.

(Refer Slide Time: 00:33)



So let us start with the weakness of hash function. Okay, so suppose we have design a hash function and you want to sell it to the Microsoft and your frames has design the hash function and he or she wants to sell it to Microsoft.

So the Microsoft open the competition and so what Microsoft will do, Microsoft will take your hash function and give it to your friend and your friend hash function give it to you and as it starts for find out the input, find out the set where it is performing badly. So find out the set of K such that it is giving a, it is colliding actually for the same slot for some slot i.

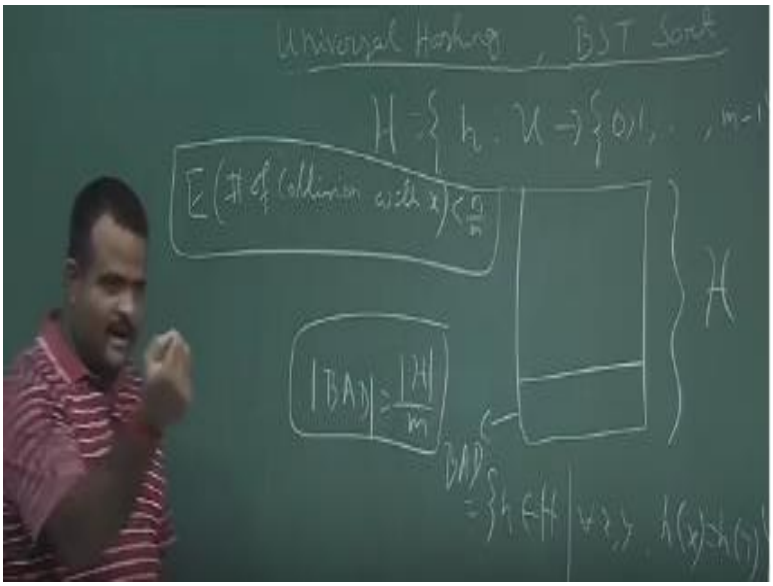
So you can, if you have time you can try for all possible key values and you can find out that where your frames has code is colliding for this input, so this is the set. So for this input it is all are colliding to the i^{th} slot okay. So if the hash function is given to you, you can easily come with such a input set.

And your friend is also doing that, so how to avoid this, to avoid this we can choose the hash function randomly, so that nobody who can come with some input where it will be performing, there will be definitely collision, because this is.

(Refer Slide Time: 02:28)

This table size is small and this set of key size is very huge. So there has to be collision, but we cannot find out the set of these where it is colliding that is difficult if we choose the hash function randomly. So want to do that is universal hashing.

(Refer Slide Time: 02:51)



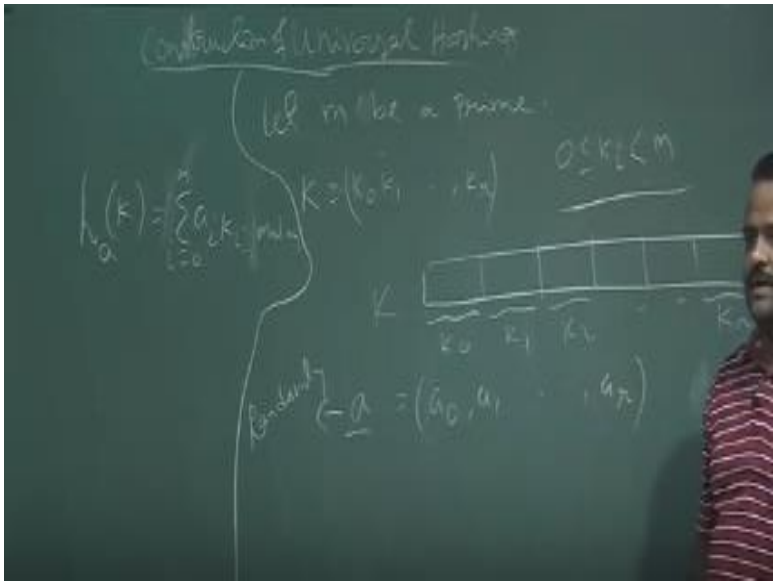
Suppose this is our set of all hash function from u to okay, and we call this hash function is universal hash function or universal hashing if among this set we have considered the portion which is denoted by bad portion that means in this portion the functions. So bad is defined as such that, if you take any key, if you take any key is y it will colloid.

So if the cardinality of this bad portion is cardinality of this by m, then we call this and then if we choose the hash function randomly from this set, then this is along H then we call that is a universal hash function, because that will, that collision will be less in that case we can prove that, but this is clear. So this is the bad portion, bad portion means, so this is a set of all hash function among this bad portion means if you take any two keys for those hash function which is coming from this bad portion they will be colliding.

So if the bad portion size is 1/m fraction of the total size, then we will have a, then we will call this is the universal hash function. And it can be shown that if you choose a hash function randomly from this head, from this universe then the expected number of collision with X with a key is less than n/m the load factor, this can be proved.

So that means if we choose a key randomly and the expected number of collision with that key is less than load factor. So it is a good choice if we choose this hash function uniformly and random from this universal of the hash function. Now we will see an example of a universal hash function, how we can construct a universal hash function.

(Refer Slide Time: 05:47)



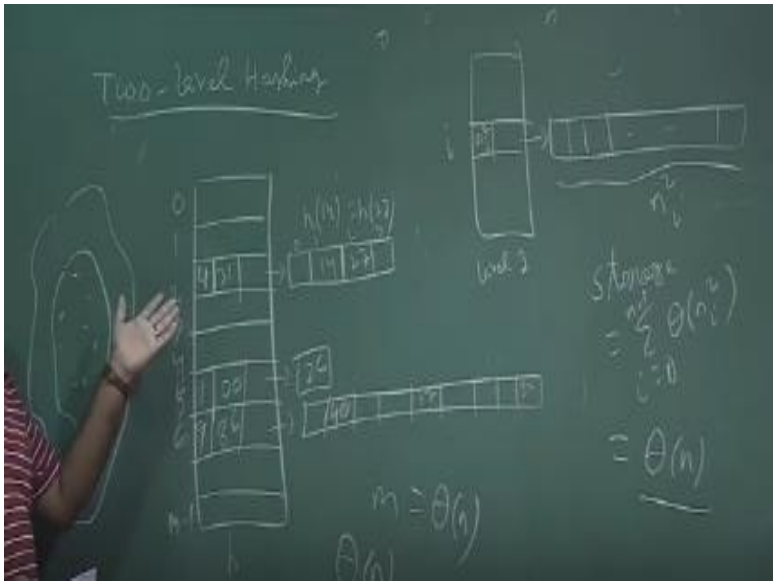
So construction of one construction, there may be other construction, construction of, so this is one example of universal hash function. So what we do, we choose that number of slot as prime m, let m be a prime okay and we decompose the key so key is of this from k_0, k_1 up to k_r and each of this k_i are coming from so we have big key and we decompose this into this is the key k_0,k_1, k_2 like this so k_r okay so r blocks and each of this part of the keys are k_i are in the range of this now we choose constant vector A randomly so these we choose randomly a_0, a_1 …a_r so this all the a_i are also less than n and this is chosen randomly.

Okay so then we defined a hash function using so this we denoted by $h_a(k)$ this a is the effecter and this is the key is basically $\Sigma(a_i k_i)$, i=0 to r and this mod m okay so this is a this is a construction or you can take mode and then take the sum a that sum so Σ this is linear product basically $\Sigma(a_i k_{i)}$ mod m so this is, this is we have a universal of the hash function and we chose

a randomly and we denote the hash of that k is like this $\Sigma (a_i k_i)$, mod n and it can be shown that this is basically this set is a universal hash function.

So we if we choose a randomly then it will give you a universal hashing okay so now we will take about perfect hashing okay.

(Refer Slide Time: 09:01)



So perfect hashing is what so suppose we have given a number of piece give this a static arrangement and we have give an we have number of slot which is basically O(n) and we have to do a static arrangement for this keys so that when we start it should be dominate constant time so that searching will be searching in constant time guaranteed and this is static arrangement.

So we are not allowing any insertion or any new keys to be any new record to be joined or nay keys to be deleted so this is a static arrangement so how we can do that this is basically a two level hashing so this in the worst case guaranteed so we should able to search in s constant time the worst case guaranteed so this idea is to do the two level hashing okay so here we have the set of keys which is fixed we are not allowing any change over here and so what we do we first apply okay so this our 0,1,2 up to m-1.

This is our number of slot so we first have apply so these are the keys k_1, k_2 k_n so we first apply the our hash function on it and we choose the hash function from the universal hash function so it will distribute the keys in a uniform way over the slots okay now some of the slots will get collusion so to handle that collusion suppose this slot suppose this slot having two keys colliding so what we do in the second level we will use another hash function another hash table where if two keys are collating so here we will have 4 slot so if 2 key are collating so that means we will have 4 slot over here this is in a second level.

So this is basically and we will have a key we i will have A, A is basically coming from the random choice universal hash function and this 31 is the A that 31 will decide the second level hash function and this is basically suppose we have some keys say 14, 27, okay. So basically under this hash function suppose this is the first level in h.

So that means h (14) = h(27) = 2 this is in the first level so we know 2 % that two keys are colliding here so we have a second level hash function with 2 square 4 slots and there we have to choose the hash function, so that hash function we are choosing this way this is our A so this is 14 this is say 0, 1, 2, 3 so this I basically one h [31, 27]= 2 like this, now if there is only one say suppose for this slots.

Say suppose this is 4 this is 3, 4, 5 suppose for this slot we have only one so there is nothing in at this level so we have only one element and suppose for this slot we have 3 elements 3 keys which are colliding here at 6 so that means we should have a 9 slotting for the second level hash level 1, 2, 3, 4, 5, 6, 7, 8, 9, 9 slot and we need to have a, A that random for the universal hashing we need to have a random A which will determine the next level hash function.

And suppose these are the number which are colliding here 40, 378, 22 say so all the three elements but 2 I have heard the second level collision so we do not know all, any collision in the level two so that is why you are choosing number of slot is more if there are there are 3 elements colliding here so we are at the second level we are choosing 3 squared 9 slot, guarantee that there will be no collision at the second level.

So all this h value, h(40), h(37), h(22) all these are basically 6 they are all colliding in this here and at the original hash function then we have to choose the hash function for the second level that is basically h(86) and for this 41 we have one like this, so this is the second level hashing so now when we have to so this is this way we are ending we are stating arrangement so we are not allowing anybody to join the so join the at least or join this set or anybody to delete.

Now how the search will be for linear time? I not linear time constant time because if we have to sort some key what we do? We will just go apply the h of this we will go to the first table and then we will check whether if it is there we will return otherwise we will apply once more the second level hash function and for that we know the hash function because A is given so once A is given then we can construct we can have h(A) the hash function.

So we will compute that h(A) on this and we go to that table and we will check whether this is there are not so this way the time is the search time is linear not linear constant it is the constant time so just two times hash function we are applying, okay. So now if so what guarantee as that there will be no collision in the second level is that, if at the if this is i slot if n_i is the number of element which is colliding in this level.

This is the level one hashing then what we are taking we are taking n_i square now so ni is the number of keys colliding in the first level at the i^{th} slot then we are taking n_i square so here it is it was 3 keys colliding so we are taking 9 slot and 9 slot we are putting the 3 keys so the collision chances is even it can be shown mathematically that the collision is even less than half, okay. So now the question is, it intuitively it may looks like that we are using so much storage.

Because for if n_i is the number of number of keys colliding in the i^{th} slot so n_i square, so what is the storage, what is the storage for the second level? Storage is basically $n_i^{\ 2}$ I = 0 to n-1, okay. So intuitively it looks like you but if you remember the bucket sort where we have the n keys n keys are distributing over some buckets and there we have this n_i, n_i at the number of keys in the i^{th} bucket and this we have seen this is basically order of n.

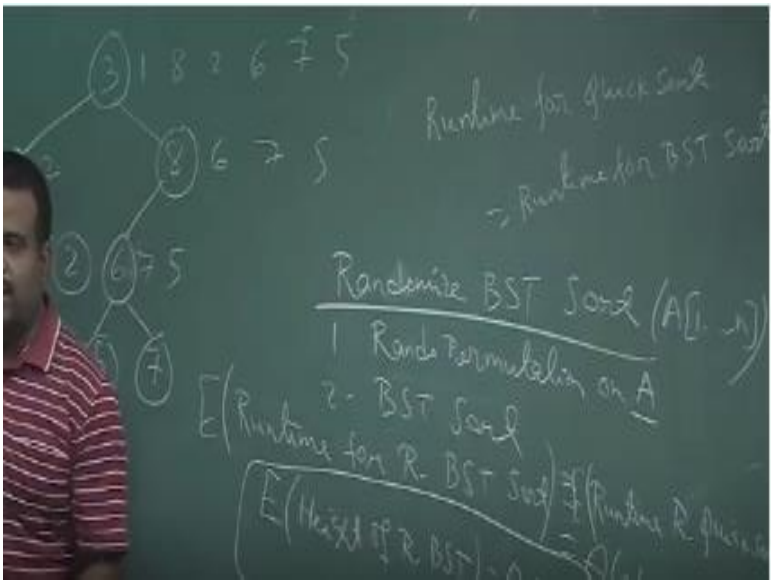
So in the second level we are using this analysis we have done while we talk about bucket sort, so they are so this is the, so in the second level hashing we are also spending the linear storage so overall or if m is also linear m is, so overall our storage is order of n, so we not using the huge storage inventively it is looks like that but it is not.

Okay, so this is call perfect hashing which is basically two level hashing and it is showing that the guaranteed worst case starts time and it is basically and the storage is also is really good. Okay, so next we will talk about binary start tree sort, so this is all about hashing up to this is our syllabus.

(Refer Slide Time: 19:36)

So next we will go to the, move to the next topic which is binary search tree sort, in sort it is BST, so what is a binary search tree it is a binary tree where binary search property means if you take any element X the key value, so all the key value is in the left sub tree is less than X all the key values in the right sub tree is get up the next, so that property is called binary search property. So now how we can have by sorting algorithm based on the this binary search tree.

(Refer Slide Time: 20:23)



So let us have this, so suppose this is BST sort yeah, BST sort so we have an array of n element and I meaning to sort it, so what we do initially our tree is empty and then we slowly insert the element into the tree we will form the BST. So, 1 to n we just do the tree insert BST insert we will see how this code is and then after that we will perform a in order tree walk, then it will give as a sorted.

Perform an in order tree walk, okay so for example suppose we have this tree suppose we have given this 3,1,8,2,6,7,5 so suppose this is our array, this is our element we need to sort so first tree is empty we need to form the trees so this is the tree insert, so 3 then 1 is less than 3 we will put it here this is the tree insert 8 and then 2 is less than 3 which is greater than 1, 6, 6 is greater than this but less than 8, so 6 is here, 7,7 is greater than this, less than this, greater this and then 5, 5 is greater than this, less than this 5. So this is the, this is after this BST insert.

Now we, we need to perform this in order traversal we need to traverse the tree so for that in order traversal means we first visit the left then root then the right, this are recursive call left sub tree root at right. So if you do that, so left then again recursively so it is basically give as 1, 2, 3 and again for this part 5, 6, 7, 8 sorted. Okay, so this is the, this is what is call BST sort. Now we

need to analysis this code hoe much time we have, what is the run time for this code, so what is the run time for in order traversal this, this will take linear time order of in because this is just we are visiting each of this node.

Now the question is what is the run time for this, so this will be, this will be time for this sorting because this is the measure time, so well this will depend on the, on the tree on the numbers. Now suppose our tree is like this, suppose our input is, input is already sorted so that means our tree will be like this and then what is the runtime what is the time to insert this, this is basically depth of the note and this is basically 1+ 2+ up to end so this is ending to order of n2 is the washed case now when is the based case if the tree is like this so this the washed case then the runtime is already n2 now if the tree is like this very good tree.

I will have to draw this tree anyway so this is what is called balance tree so now if the tree is like this then what is the runtime then what is the this tree insert tree insert is basically hearing to so order of l logn okay because there are n/2 notes over here so for this notes each time is logn so for that it is order be so this the based this is the lucky case so can you remember some sorting algorithm we know which will give us this runtime washed cases is L2 this is yeah that is the quick sort.

So you want to see how this quick sort is related with the BST sort so for that let us just take one example suppose we have 3 1 8 2 suppose we want to sort this using quick sort and we choose these are the three word element so if we do that if we apply the partition a little partition into two part 1 2 over here and then remaining over here 8 6 7 5 and here there assuming that we are ordering at not changing so this is some sort of table.

Partition we can have so ordering and not changing the same ordering there achieving here so all the elements are get at the links so again we choose these as a p word so these will be 2 and again we choose these as a p word so it is partition like this so this is all the elements 6 7 5 and again we choose these as a p word so 5 7 like this so this is just the week sort partition this is the quick short and then if we just draw these if you just draw this okay so this is if we remember the form of that we already done the.

BST sort for this example and you have the same tree so that mean same number of comparison we need for week sort and the BST sort I mean to get the BST tree in maybe some different order because same number of comparison 3 everybody as to compare with three so there also when insert when we make the tree everybody has to come so same number of comparison is required for both the cases that means.

That mean run time for week sort is same as runtime for BST sort this is the good observation so these observation we will use when we talk about randomize part of BST sort and randomize version randomize BST sort so what we do we have given the array we just random the permit these numbers we apply some random permutation on and random permutation on A array and then we perform the BST sort.

Okay we have some numbers we are randomly permuting these number and then in perform the BST sort and in that case so this is similar to the randomize version of quick sort so the runtime for this is runtime for randomize BST sort is same as runtime for randomize quick sort so the expect runtime for randomize for BST sort is same as exceptive runtime for this so expected runtime we know l logn so from here we can so that expected height this is the things we want achieve expected.

Height of this tree height of randomize BST tree BST is basically logn so expected height is logn so that mean so if we have some numbers and if we just do the random permutation on this number then it will give us the balance tree expected do it will be logn height tree so very nice tree balance tree but this is expected there is no guarantee but we can say the expected runtime expected height of this tree is logn thank you.