Algorithm for Protein Modelling and Engineering

Professor Pralay Mitra
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture-10

Hashing

Welcome back! In lecture 10, we are planning to cover the concept of hashing. For a few protein docking strategies, the hashing will be required to generate the different orientations. Therefore, in the last couple of lectures, we covered various ways to generate protein complexes.

Given two protein molecules, you are generating various orientations. Then score the orientations – either while generating it (integrated scoring) or after generating all (edge scoring). I prefer to score when the orientations are also being generated.

We noted that all the atoms are being considered while developing the algorithm. However, to know given two protein molecules where they will bind. Sometimes the user may provide binding information either on where they can bind or where they cannot or by combining these two as other information.

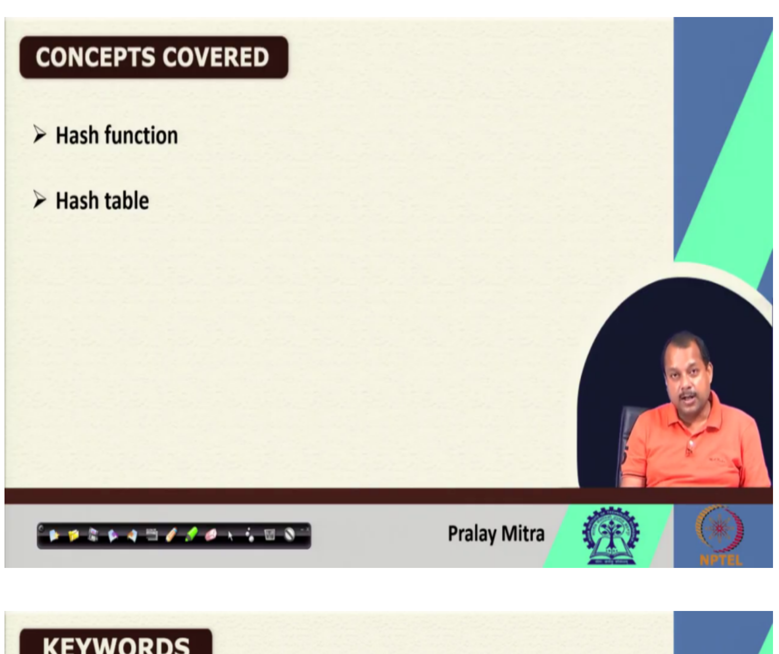
Sometimes while developing the algorithm or analysing the problem, the biologist or chemists may provide some essential inputs in terms of anchoring points. As an analogy, I can say that there is a river. When a boat or a ship is going through that river may dock at a port.

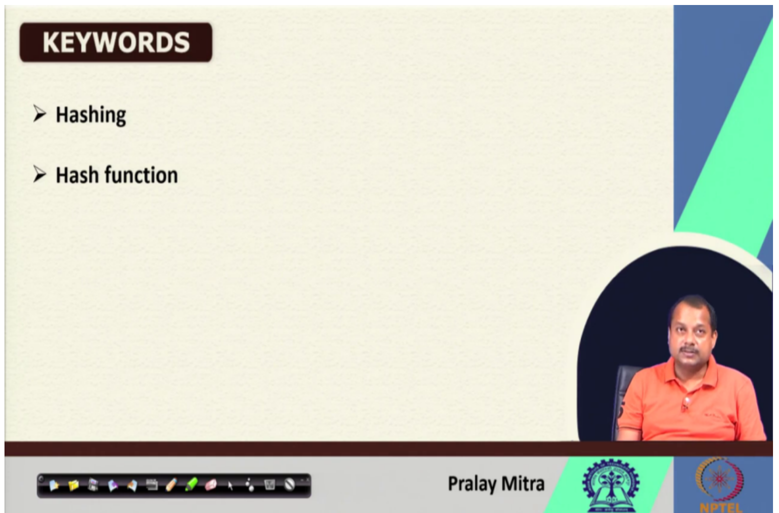
Theoretically, a ship can go and harbour or dock anywhere depending on various factors like whether the depth is sufficient, whether harbouring at that position is enough or not etc. At the port, there is some identification mark of the port so that a ship can understand where to dock.

Similar to that here also, when two protein molecules are there and if you have some prior knowledge regarding these two protein molecules, then perhaps you know where to bind. We have discussed it while developing a brute force or FFT based algorithm where we need not have to consider all the atoms, we have to consider only the surface atoms.

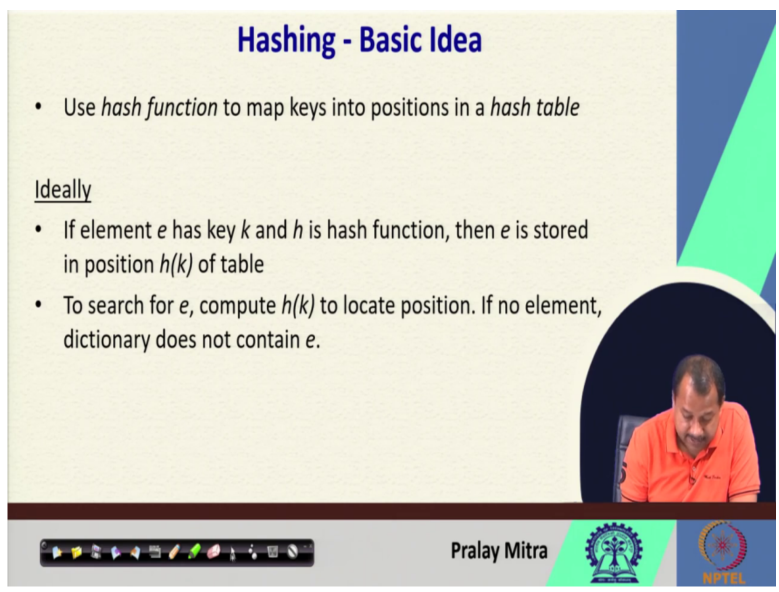
Considering the surface atoms are enough, the question is whether all the surface atoms are also required or only a subset of them are required. Another algorithm exists to address that issue for which we need hashing concept on which that algorithm has been designed.

(Refer Slide Time: 04:55)





(Refer Slide Time: 05:10)

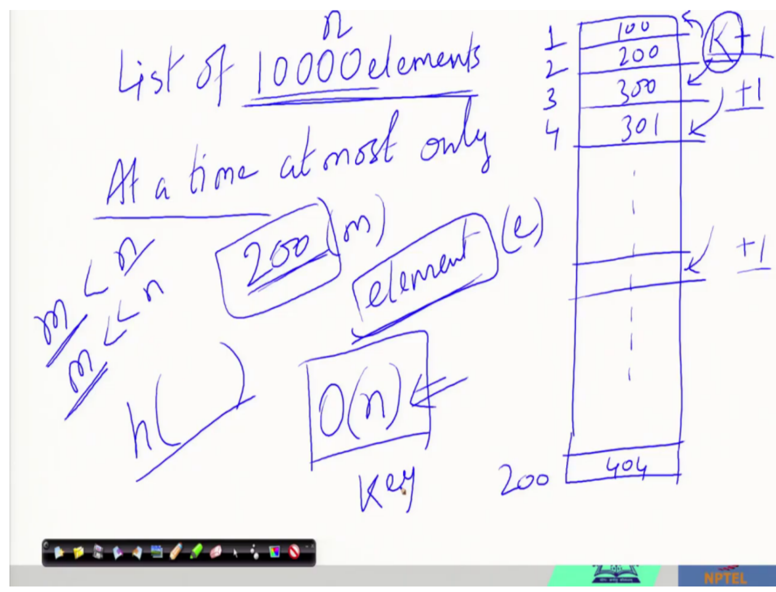


So, we plan to cover a few topics here Hashing function and the Hash table. We shall not go into deep details of hash, but the concept which will be relevant for us and which is required will be discussed here.

The basic idea of hashing is that whenever there is a list of elements which do not occur at a time then use hashing. For example, let us assume that there are 200 students registered in a class and you will find a few students are absent every day. Now, if I have an idea regarding the average number of students present then it is preferable to declare storage which can accommodate the average number of students rather than 200 students. That way I can save some storage space.

Use a hash function to map keys into positions in a hash table, ideally, if element e has key k and h is a hash function, then e is stored in position $h(k)$ of the hash table. To search for e compute $h(k)$ to locate.

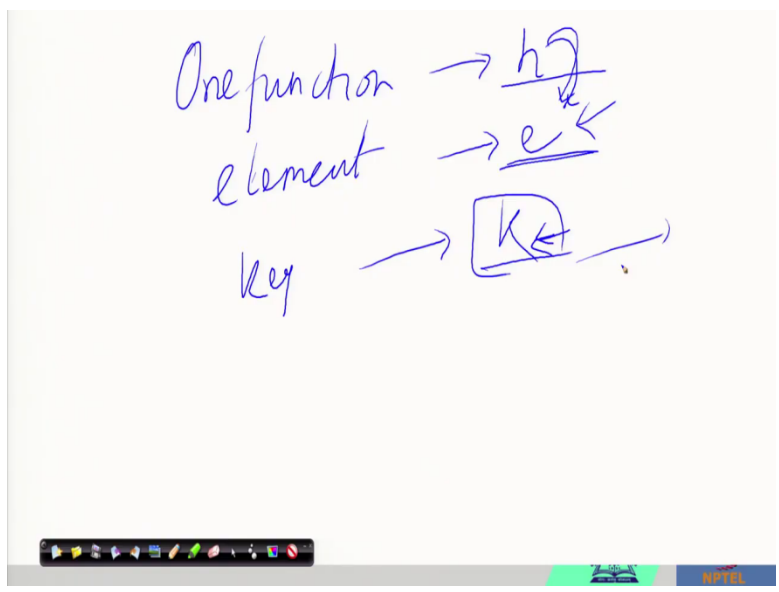
(Refer Slide Time: 07:20)



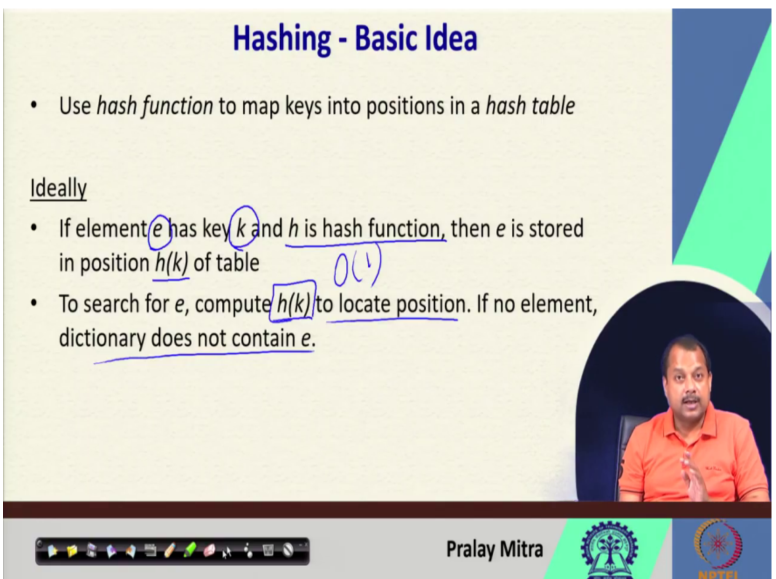
Let me give you one example. I have a list of 1000 elements and at a time at most 200 can occur. Then I suggest declaring storage for only 200 elements. Inside this what we store is 1000 elements, which can be 100, 200, 300, 301, 404, etc. And my suggestion is that if this is my n and this is my m where $m<n$ or $m<<n$ then it might be a good idea to work with m storage locations for n variables. Now, you will also agree with me if instead of 1000, it is 10,000 then also with storage space for 200 elements insertion, and searching will be faster. That is the motivation of hashing.

If I store that information serially, then insertion will be easy using an index variable, say k. Since the insertion is not in a sorted way so we cannot apply binary searching, and the only option is the linear searching. Hence, searching for an element in the list will be of time complexity $O(n)$. To improve this I will define one function viz., h, which will operate on element e to generate position/key k.

(Refer Slide Time: 11:41)

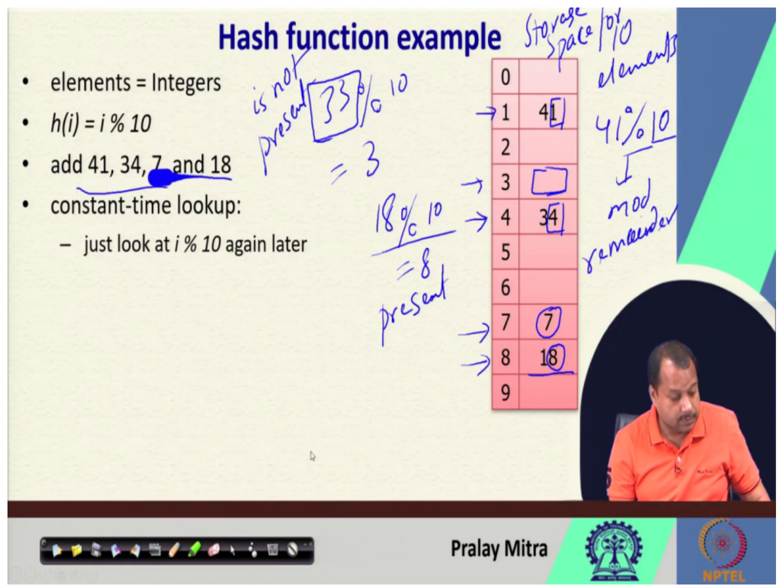


(Refer Slide Time: 12:51)



Function h will operate on element e to generate key k and then e will be stored at $h(k)$. Therefore, insertion and searching are simply calculating a mathematical function which can be done in constant time. In the case of searching e, if e is present at location $h(k)$ then the search is successful, unsuccessful otherwise. This is the core idea behind hashing. Here you can see that each element e has key k and h is a hash function then e is stored in position $h(k)$. This is a mathematical calculation which can be done in constant time to locate a position.

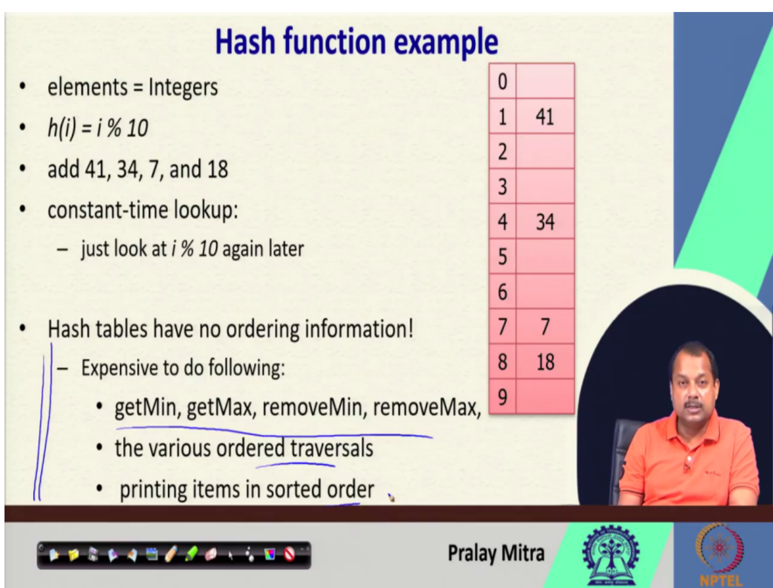
(Refer Slide Time: 13:30)



In this example, I have the storage for all the ten elements 0 through 9 and you see 41, 34, 18 all those elements are present which means the range of the elements which can be is more than 10 but I did not have to declare 41 sizes of an array to store 41. This is more advantageous specifically when I have 10,000 elements and I am working only with less than 100 or at most 100 elements. So, if that is the case then this is an advantage, yes.

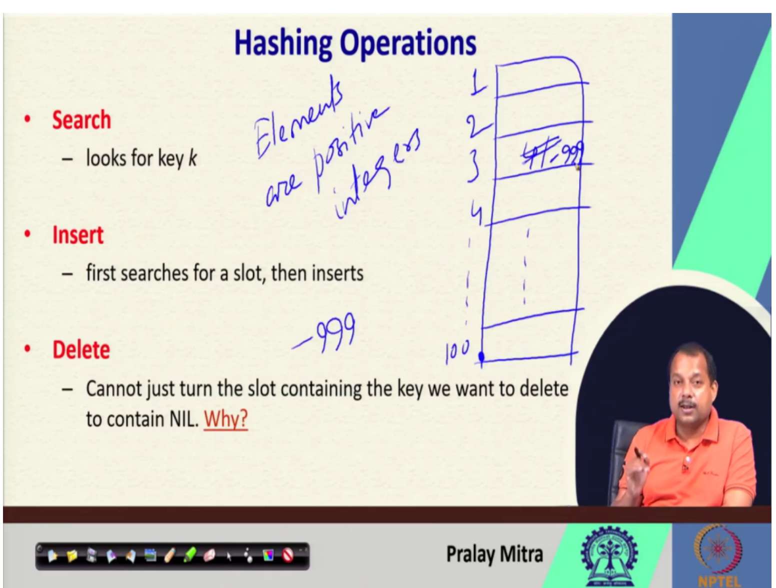
Now, in the storage space for 10 elements how can I insert 41, 34, 7, 8? I declare one mathematical function $h(k)$ that *mod 10* function. Thus, I shall divide 41 by 10 and will take the remainder which is 1 that's 41 is going to 1. For 34, it is stored at 4, for 7 it is at 7, for 8 it is at 8. Here, elements are integers and the hash function is *mod 10* and 41, 34, 7, 18 all will get some position in this list in constant time. Now, I am asked to check whether 33 is present or not. What shall I do? I take *mod 10* of 33 which will give me 3. Then I check position 3 which is blank, so 33 is not present. Similarly, given 18, 18 *mod* 10 = 8. Check at position 8 and it is present. How much time is it required - just one modular function and then you go directly to that location in search of the element. As I mentioned, if the possibility of total integers elements is several hundred or millions whereas, at a time only less than 100 elements occur then this kind of technique is advantageous.

(Refer Slide Time: 17:20)



Hash tables have no ordering information. It is expensive to perform *getMin()*, *getMax()*, *removeMin()*, *removeMax()*, and the various order traversal printing items in sorted order. These are the demerits of the hash function which says that in your application if there is a requirement of either of these then this hashing is not a good idea. But, if it is the case that you just wish to insert some element you search that element for some processing mostly when you are using that one for this intermediate storage purpose then it is very good.

(Refer Slide Time: 18:08)



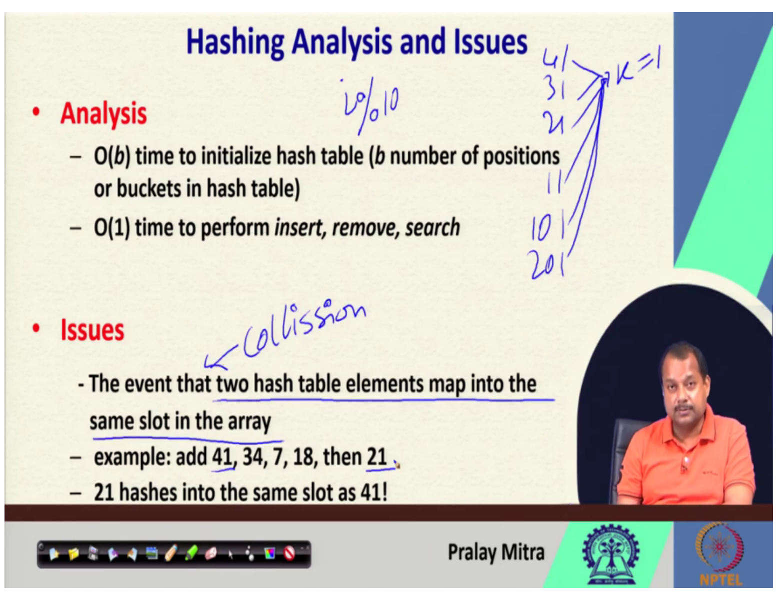
We have just seen the modulus as one hash function. Using that modulus operation, we have seen the search operation. When an element is given to you and you are getting key *k* using *mod 10*. If a new element has arrived you insert it. Again you identify or calculate the key

where it will be stored. We have discussed *mod*, but there can be other hash functions also. I shall show you some and you can also discover others.

Deletion means when I wish to delete some entry. But one thing you should remember, deletion in a computer system is not possible, because erasing option is not there. The only thing you can do is overwrite. So if you wish to remove that one, you assumed some data, which will not be a member of that list and you write that one in that position to delete.

Let us assume this is your hash table. Elements are there now. Here 41 was written and without any loss of generality, I am assuming this is 1, 2, 3, 4 dots dot to dot 100. Now, you wish to delete this 41. Deletion is not possible you can do that at this position if you have the knowledge and I am sure that you should have the knowledge of the range of the elements and if you know that your elements are positive integers then you can pick any negative number. For example, -999 and you write it here -999. -999 indicates that space has been removed, so you can have another entry in that location.

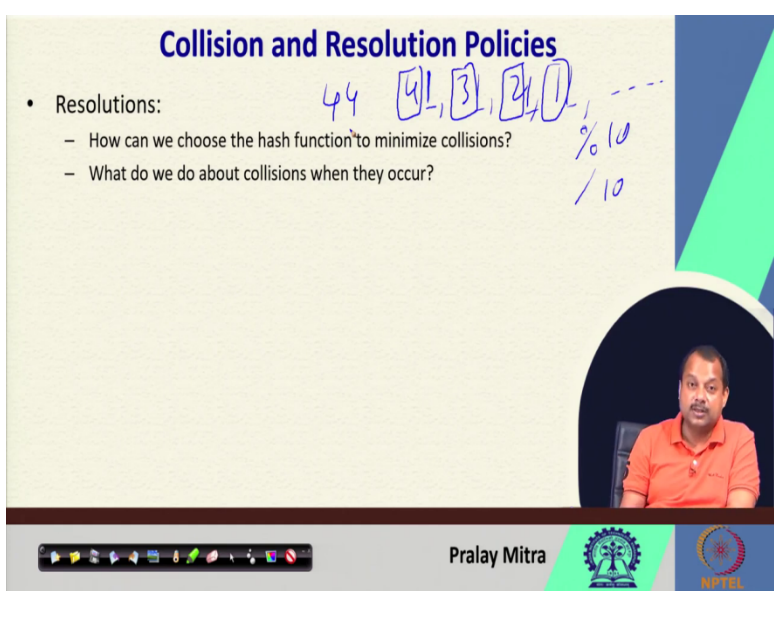
(Refer Slide Time: 21:20)

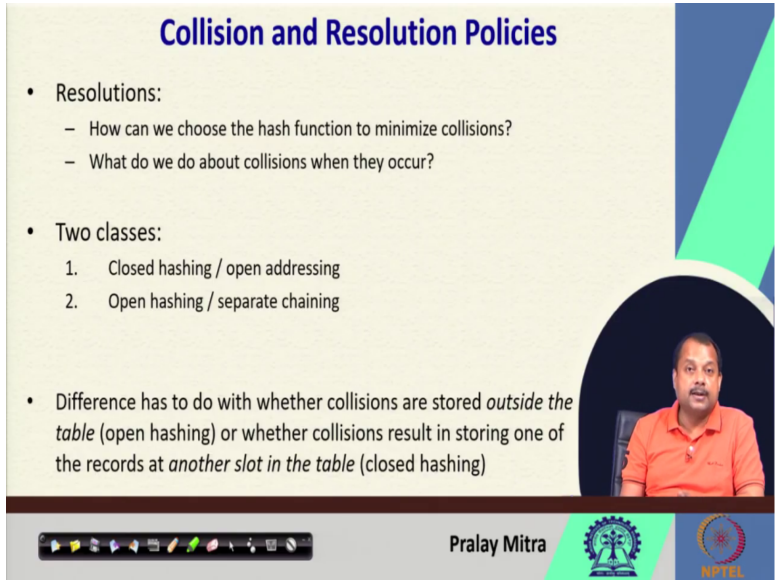


Now, there are several issues also with the implementation that I shall discuss now. The first one is when two elements map into the same slot in the hash table then how to handle that. I believe that this question is coming to your mind when I started to write *mod 10* as one hash function. If *mod 10* is my hash function then elements like 41, 31, 21, 11, 101, and 201 are supposed to go to one location because they are generating the same key 1. That leads to a situation called a collision. Collision is a standard thing and hardly can be avoided.

In this example after inserting 41 if I was to insert 21 then using that *mod 10* function, both are generating the key *k* in the same slot they are supposed to go so there is a collision. There are ways to avoid this collision without losing the advantage of the hash function.

(Refer Slide Time: 23:44)



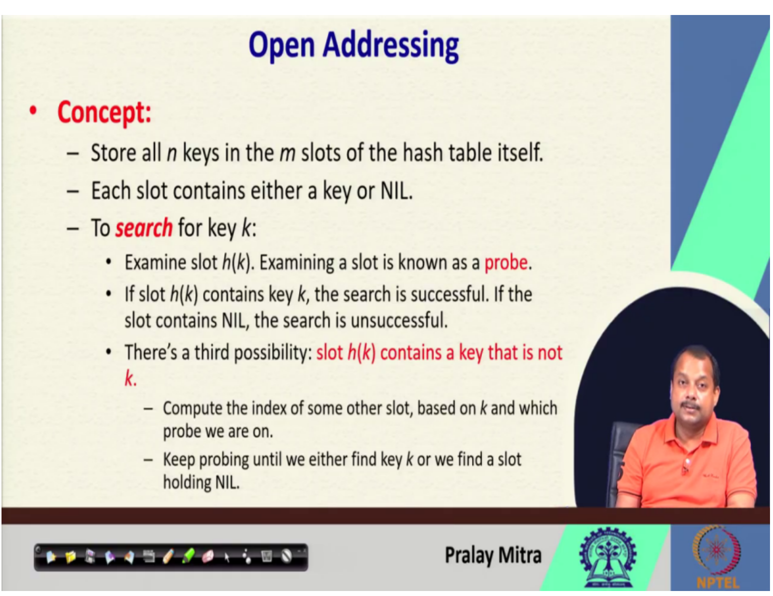


The first resolution is how can we choose the hash function to minimise a collision. We should look at the distribution of the list of the numbers based upon that one we will decide what could be the best hash function so that such kind of collision will not arise. Is it possible that instead of one hash function *mod 10*, I can do integer division by 10 then what I will get instead of all 1, I will get this 4, 3, 2, 1, and there is no collision.

Again, if I take this one then 41 will collide with 44 so, you have to look at the pattern or the distribution of the numbers accordingly you decide. And what do we do about collisions when they occur? It is true as of now what we have discussed based upon that one that collision cannot be avoided completely. I can minimise it by designing a good hash function, but I cannot avoid that one. If I cannot avoid that one there should be some technique through which the collisions can be handled. So, there are two classes closed hashing, open addressing open hashing separate chaining.

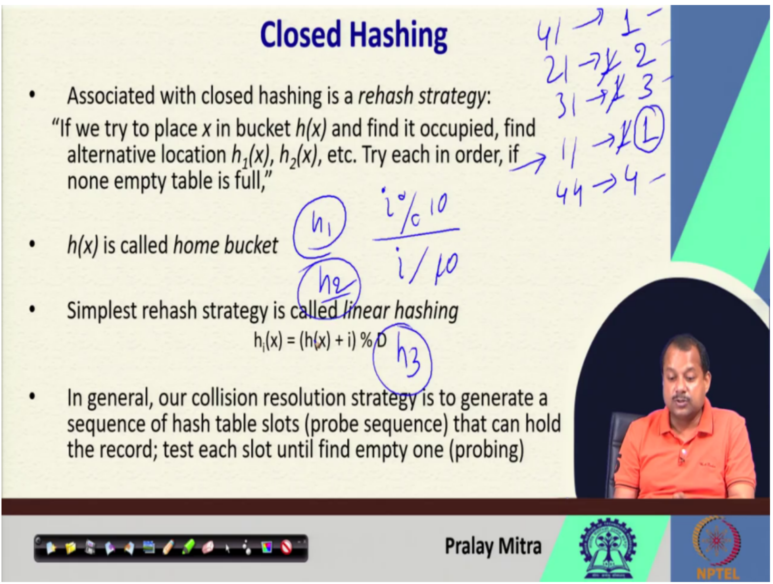
In brief, we shall discuss those and the difference has to do with whether collisions are stored outside the table open hashing or whether collisions result in storing one of the records to another slot in that table closed hashing. I mentioned that open hashing means that table will be intact, so whenever there is a collision, you store that information outside the table in some other location or the same location in the same table in the same table. You can store if you know that at a time at most the size of the table number of elements can occur. Then it might be a good idea to store it on the same table. But, if that is not guaranteed, then possibly you can store that one outside of that table.

(Refer Slide Time: 26:24)



Here is an open addressing concept store all n keys in the main slots of the hash table itself each slot contains either a key or NIL to search for key k examine slot $h(k)$. Examining a slot is known as a probe. Probe the hash table's each slot with $h(k)$ to search if it is a success or unsuccessful. There is a third possibility also that says that $h(k)$ contains a key that is not k, then what to do?

(Refer Slide Time: 27:05)

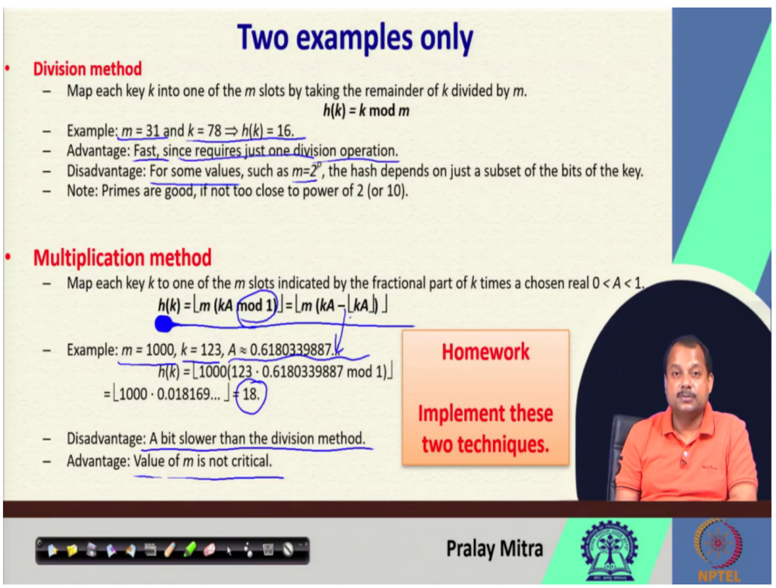


In the case of close hashing, it is associated with the rehash technique where after one hashing another hashing will take place to minimize the chance of collision. In the previous example, of 41, 21, 31, 11, and 44 I look at the distribution and instead of *mod 10* alone use *div 10* in conjunction with *mod 10*.

Therefore, this will go to key 1 then it will go to 1 but because of the collision, it will go to 2 following the second hash function. So this is 1 I am considering as h2. I am reconsidering this and it will go to 1 but because of the collision, it will go to 3 using this h2. This 1 and 1 this collision I cannot avoid.

Again the collision is there, so this triggers the situation of another hash function h3 but, before that, you see this 44. When I shall take this h1 fine. It will go to 4, so I can accommodate 1, 2, 3, 4 but still, there is one collision for 1 or how I am doing that one I am going for another hashing. So hx is called the home bucket this h1 and these are the auxiliary hash functions which may be required to deal with this situation when there is a collision.

(Refer Slide Time: 29:26)



Now, two examples I have presented to you. One is the division method. I just mentioned also mapping each key *k* into one of the *m* slots by taking the remainder of *k* divided by *m*. Here also *m* 31, *k* 78 and then *h(k)* equal 16. The advantage is that it is fast since it requires just one division operation. In constant time, it can be done. Disadvantages for more values such as *m* equals 2 to the power P depend on just a subset of the bits of the key. Note, Prime's are good, if not too close to the power of 2 or 10.

Another situation then is the multiplication method. In the multiplication method map, each key *k* to one of the *m* slots is indicated by the fractional part of *k* times a chosen real and that real number will be between 0 and 1 example is given here. Each *k* equals to this *h(k)* equals *mkA mod 1* and I am taking the flow function that will give me *mkA minus kA*. Since it is a *mod 1* and I am also taking the flow function an example is given here *m* equals 100, *k* equals 1 2 3 and that way, I am getting an approximately like this. The advantage is the value of *m* is not critical and the disadvantage is that it is a bit slower than that division method. So, what I am proposing is just two examples. Based upon that you try to implement these two techniques. This you can consider as your homework, implement these two techniques. Thank you very much.