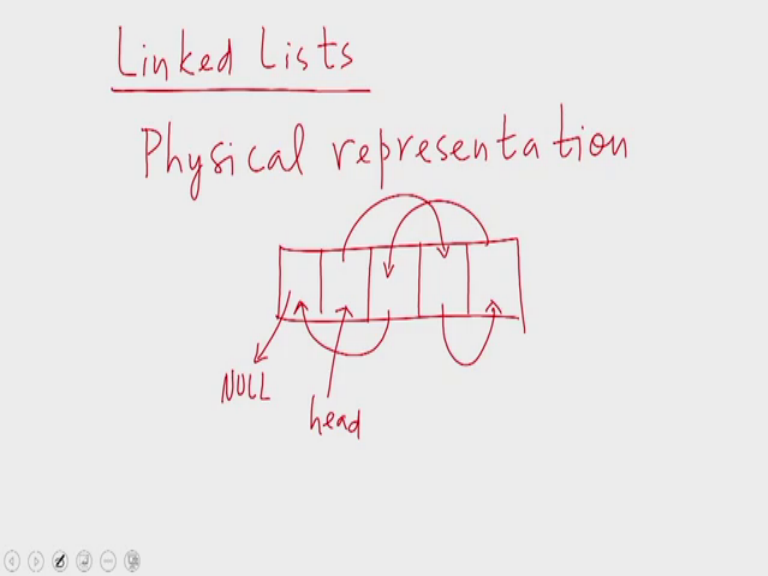
Parallel Algorithms
Prof. Sajith Gopalan
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 09
Basic Techniques 5

Welcome to the 9th lecture of the MOOC on Parallel Algorithms. In the previous lectures we have seen several algorithm design techniques; we will continue with the discussion on algorithm design techniques. Today we shall see an algorithm design technique called symmetry breaking. For example, when we have several entities conflicting with each other for some resource, you can construct a resource graph conflict graph for those entities.

And then when we allocate the resource, we might want to allocate the resources to entities that do not conflict with each other. So, we have to break the symmetry between the entities in some way. So, this is the essential idea of a symmetry breaking, we shall study symmetry breaking in the context of a linked list today.

(Refer Slide Time: 01:19)

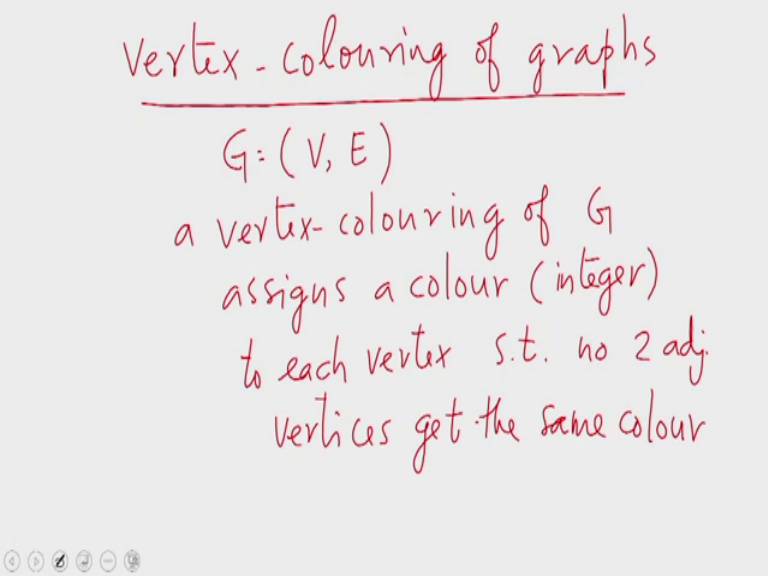


So, we have seen; we have seen linked lists before in this course and we have done an algorithm for ranking the linked lists. So, let us say we are given a linked list in an array, you are familiar with the physical representation of an linked list. In the physical

representation the notes are jumbled together and are given in an arbitrary order that is the order in which it is given in the array, has no relation to its logical order.

For example if the is the first node, you indicate the first node using a head pointer. This note points to another using the next pointer, which will point to another using the next pointer and so on. So, we have seen how the list can be ranked when it is given in this form, but the problem that we are going to address is different, we are given a linked list in the physical form and what is needed is to vertex colouring of the linked list.

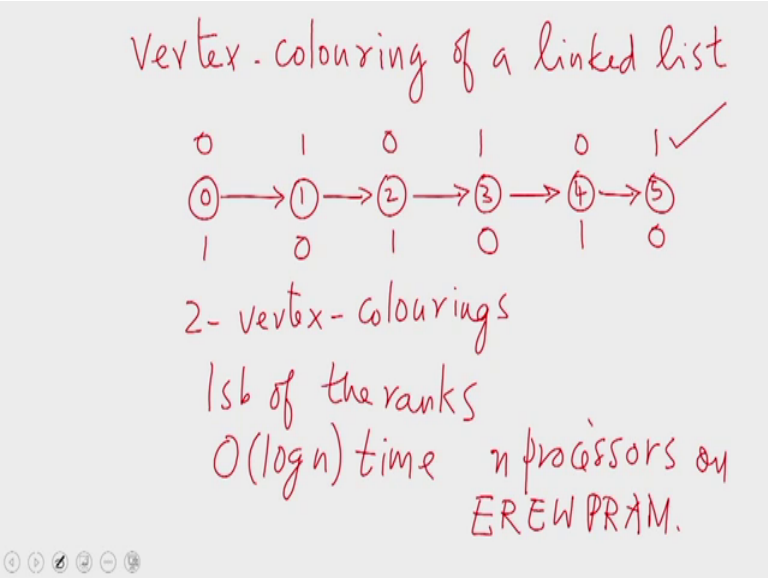
(Refer Slide Time: 02:43)



This is a special case of the vertex colouring of graphs. When we are given a graph G equal to V E; where V is the vertex set and E is the edge set. We said that a colouring of G vertex colouring of G assigns a colour could be just an integer to each vertex so, that no 2 adjacent vertices get the same colour.

So, the problem is just this, every vertex of the graph must be assigned a colour and no 2 adjacent vertices should get the same colour, its possible to repeat the colours the number of colours use could be much less than the number of vertices in the graph. So, this is the graph vertex colouring problem. We say vertex colouring in particular because we could also talk about the edge colouring problem, where we assign colours to the edges of the graph. But in this lecture we are going to consider the problem of vertex colouring and that too for a linked list.

(Refer Slide Time: 04:29)



So, let us consider a logical diagram of a linked list. So, as you can see a linked list as a by parted graph you can classify the vertices into 2 sets so, that every edges from 1 side to the other. So, that is the definition of a by parted graph.

So, a linked list is a by parted graph and therefore, it can be two coloured that is you can use utmost two colours to colour the linked list. So, there are 2 distinct colours to this linked list; it one is this you can give a colour of 0 to the first vertex 1 to the second vertex, 0 to the third vertex, 1 to the fourth vertex and so on.

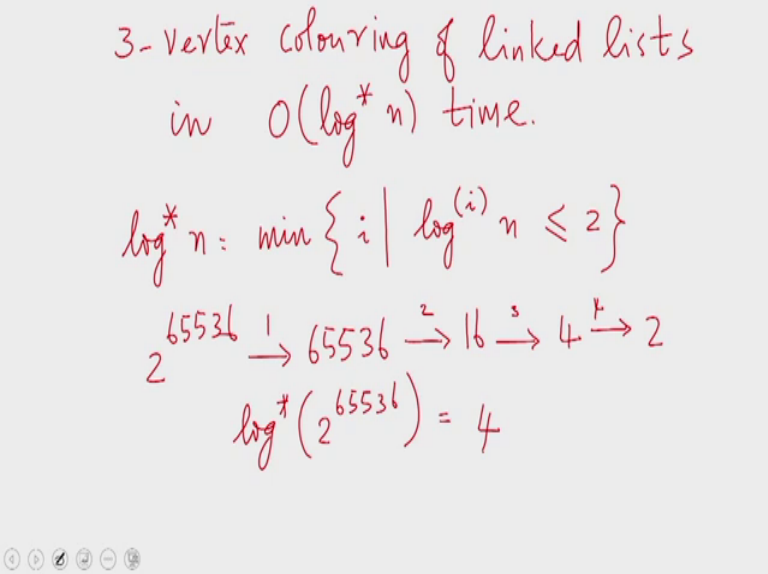
So, every odd vertex gets the colour of 0 and every even vertex gets the colour of 1 this is one possible colouring. The other possible colouring is to give a colour of 1 to the first vertex, 0 to the second vertex, 1 to the third vertex, 0 to the fourth vertex, 1 to the fifth vertex and 0 to the sixth vertex.

So, in those cases every odd vertex gets the colour of 1 and every even vertex gets the colour of 0. So, there are 2 valid 2 colourings for the linked list. So, these are what are called the vertex colourings, we call it a 2 vertex colouring because we are using 2 colours. Now a 2 vertex exploring of a linked list can be done using the list ranking algorithm. We have already seen the list ranking algorithm which runs in order of log n time using n processors on EREW PRAM. We can use the same algorithm here and after finding the ranks of the vertices, I can take the least significant bit of the ranks. So, for this list, if you rank the list from the left end you will be given ranks 0 1 2 3 4 and 5. So,

if you take the least significant bit, you will get 0 1 0 1 0 1 which is the first colouring, a bit flip of this will give us the second colouring.

So, a 2 vertex colouring of a linked list can be achieved using a list ranking algorithm and this can be done in order of log n time using n processors on EREW PRAM. But today we are going to see a much faster algorithm, I shall show that a vertex the vertex colouring of linked list can be achieved in order of log star n times where log start is a extremely slow growing function provided if we allow one more colour that is instead of 2 vertex colouring we will be 3 vertex colouring.

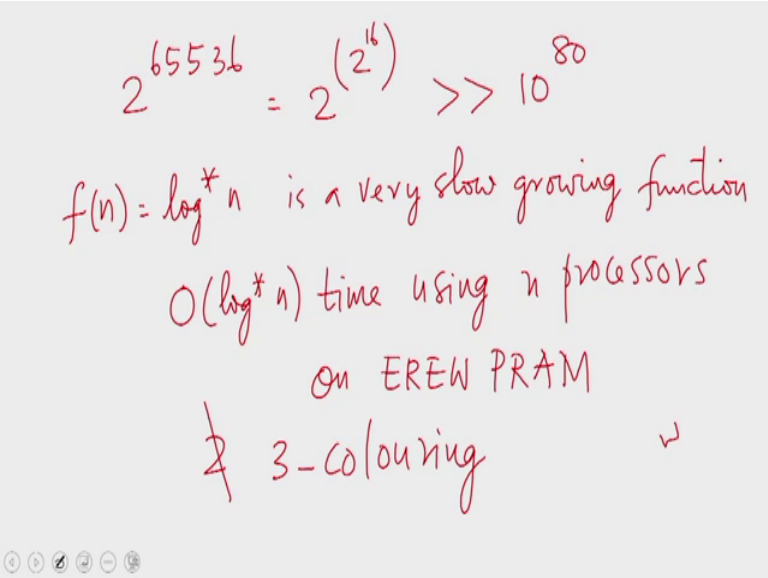
(Refer Slide Time: 08:07)



First let me define this function that is called log star of n. For an integer n log star of n is defined as the smallest i so, that when logarithm is taken i times over n we get a value less than or equal to 2. Log star of n were n is an integer use the smallest i so, that when n is taken when log is taken n time i times on n, we get a value less than or equal to 2. For example, let us consider the number 2 power 65536. If you take logarithm of this number ones you get 65536 you take logarithm once again you get 16, you take one more logarithm you get 4, you take another logarithm you get 2.

So, when you take logarithm 1 2 3 and 4 times over 2 power 65536 we get a value of 2 therefore, log star of 2 power 6 5 5 3 6 is 4. Even for a number as large as 2 power 65536 log star is only 4 and how big is 2 power 65536?

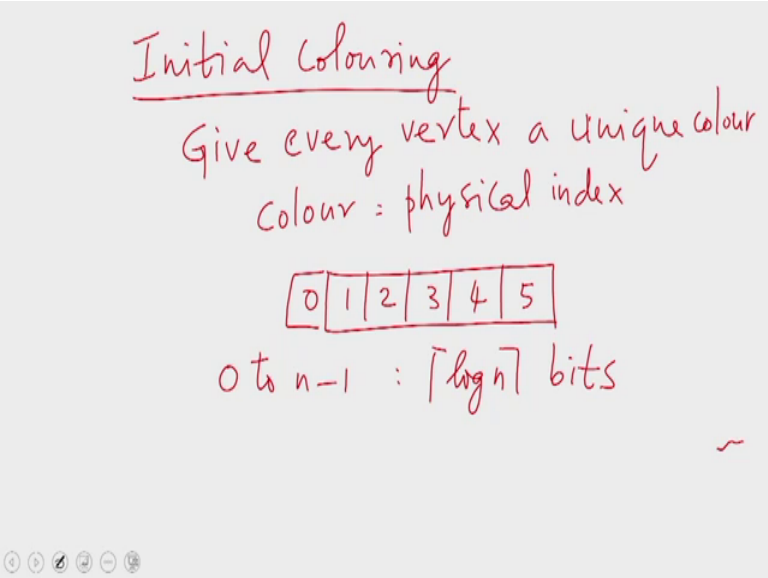
(Refer Slide Time: 10:15)



This is 2 power 2 power 16, which is far greater than 10 power 80 and 10 power 80 is approximately the number of atoms in the universe. So, the number that we are talking about 2 power 6 5 5 3 6 is extremely large indeed.

So, even for numbers which are extremely large, so, large that we are unlikely to encounter them in real life, even for such numbers the log star value is utmost 4. Therefore, for all numbers that we encounter on reality log star is less than or equal to 4 which shows that log star n is a very slow growing function indeed. So, slow growing that it is particularly a constant. So, we are going to show an algorithm that runs in order of log star n time, using n processors on the EREW PRAM. The only thing is that for this we have to pay a price in colours instead of 2 colouring, now I will be using a 3 colouring. So, now let us see how the algorithm proceeds.

(Refer Slide Time: 12:17)

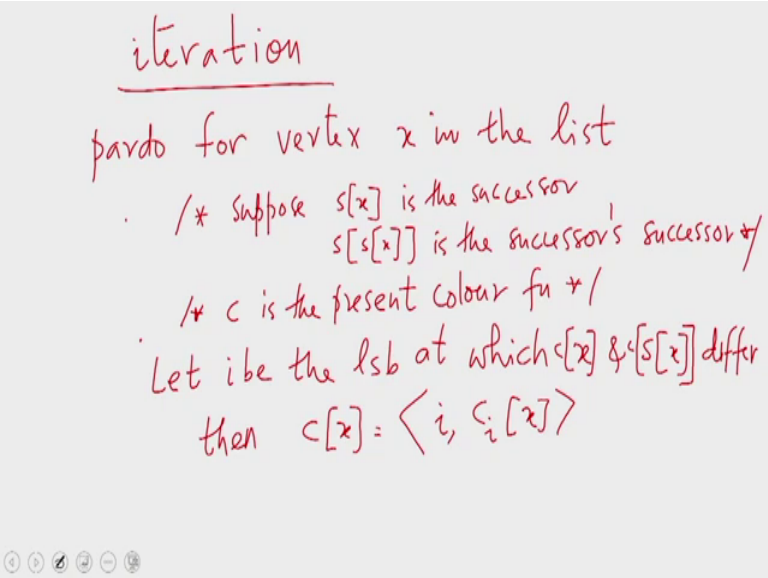


We begin with an initial colouring. In the initial colouring we give every vertex a unique colour; the purpose was to give every vertex is unique colour. We can give every vertex a unique colour if every vertex is coloured with its index. So, the colour is the same as the physical index. That is in the physical representation we have all the notes given an array these notes are numbered from the left end in this fashion they are numbered consecutively.

So, they are all going to get a unique number of course, we will start the colouring from 0 so, if we have 6 notes we will be colouring in this fashion. So, in general if we have n notes, the colours will range from 0 to n minus 1. So, the number of bits required to represent these colours would be ceiling of log n. The initial colours can be represented using ceiling of log n bits.

So, these are the initial colours every vertex has got a unique colouring therefore, we can say that the colouring is valid that is 2 adjacent vertices will not have the same colour. Now the algorithm proceeds in a number of iterations.

(Refer Slide Time: 14:13)



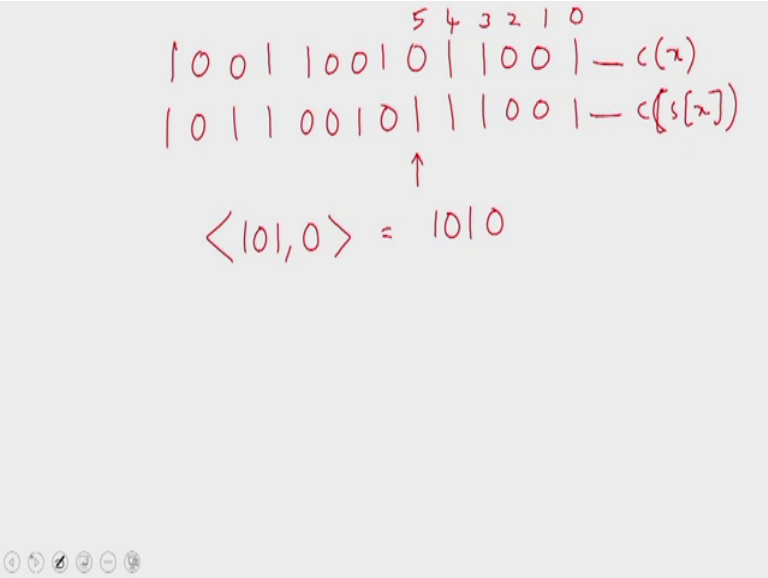
So, let me specify an iteration now, so, in an iteration what we do is this. For each vertex in the list we do in parallel the following. For vertex S suppose S of x is the successor of x and S of S of x is the successors successor. And let us assume that C is the present colour function that is C of x is the present colour of x and C of S of x is the present colour of the successor of x.

So, with these assumptions what we do is this. Let i be the least significant bit at which x and S of x differ; then the new colour C of x is defined as i concatenated with that is a binary representation of i concatenated with C i of x which is the ith bit of the present colour of x.

So, we take the binary representation of i, where i is the least significant bit at which C of x and C of x of x differ we look at the colour of x and the colour of the successor of x. We consider the least significant bit at which the colour of x and the colour of the successor of x differ and then form the new colour in this fashion take the binary representation of i and then take the ith bit of C of x and concatenate that with the binary representation of i. This is how we formed the new colour, this we do for every vertex in parallel and then we keep doing this again and again. This is what one iteration is we continue with the iteration until the colouring converges.

So, let us see how many iterations we will have to execute. First let us take an example.

(Refer Slide Time: 17:39)



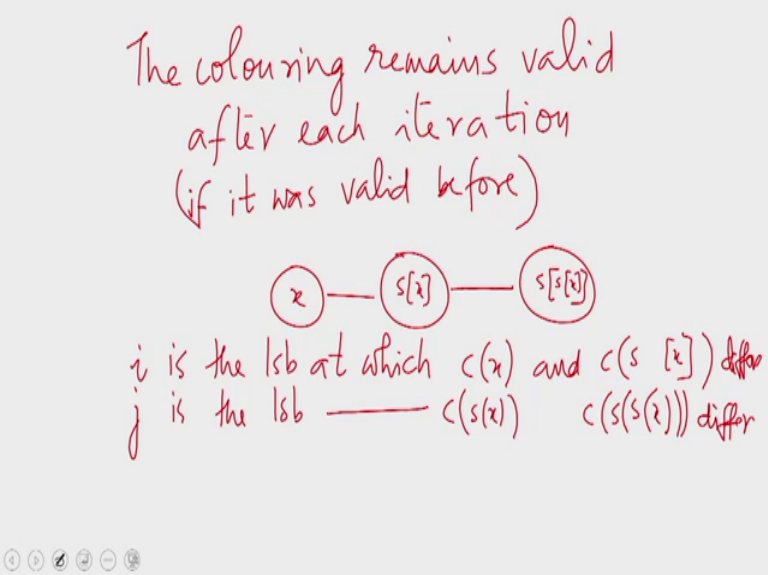
So, let us say these are the 2 colours this is the colour of x and this is the colour of the successor of x. So, in this case we find that the least significant bit at which they differ is this. So, if we number the bit positions starting from 0 in this fashion, we find that there are differing at the fifth position. The binary representation for 5 is 101; when that is concatenated with the bit value of C of x at position 5 which in this case happens to be 0, this is what is to be the new colour for x.

So, without the commas and the angular bracket it would read like this. So, 1 0 1 0 is going to be the new colour of vertex x. So, as you can see this method shrinks the size of the colour substantially. So, this is what we are going to do in the algorithm; in an iteration every vertex participates we do this in parallel for every single vertex, what each vertex does is it takes its own colour and its successors colour. So, every vertex should know its own colour and its successors colour, then it finds the least significant bit at which these 2 colours differ.

Suppose this happens at position i, we take the binary representation of i and then the bit of C of x at that position. When these two are concatenated we get the new colour for vertex x. At the same time the successor of x is also redefining its colour it would be looking at S of x and itself it takes those 2 colours that a C of S of x and C of f of S of x and then it finds the least significant bit at which those two differ that concatenated by the bit at that position of C of S of x will form the new colour of S of x.

So, in this fashion every vertex redefines its colour. When this iteration is repeated several times, we would be reducing the number of colours used in the list substantially. Remember we had begun with an initial colouring of ceiling log n bits that is we had n minus n colours used every vertex has a unique colouring to begin with. So, in the initial colouring we had n colours as this process proceeds the number of colours will substantially reduce. So, we shall see by how far we will achieve this reduction and let us also see how this reduction could be achieved.

(Refer Slide Time: 21:03)

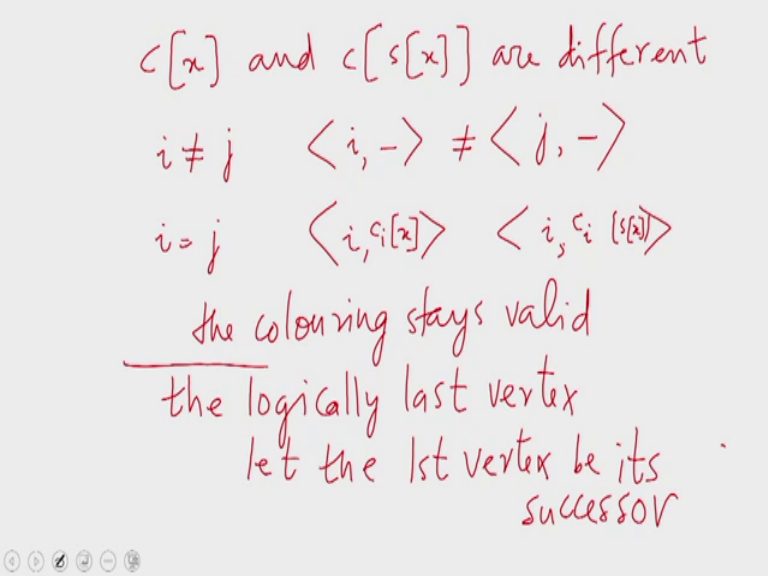


So, first let us show that after every iteration. The colouring remains valid after each iteration if it was valid before the iteration, we want to claim that the colouring remains valid after each iteration if it was valid before the iteration. So, to claim this let us consider 3 vertices x S of x and S of S of x and C is the present colour I mean present colour function which means C of x is the present colour of x. So, let us say x and S of x that differ in the ith bit that is i is the least significant bit at which the colours of x and S of x differ. That is C of x and C of S of x differ at the ith least ith least significant position and they agree on or less a significant positions and let us say j is the lsb at which C of S of x and C of S of S of x differ sorry in the previous line I used S twice I should have used it only once.

So, here let me change like this. So, vertex x is colouring itself with the binary representation of i, where i is the least significant bit at which C of x and C of S of x

differ and then it also uses the bit of C of x at that position. Whereas, S of x is colouring itself with the least significant j at which C of S of x and C of S of S of x differ and it concatenates that j with its own bit at that position now what I want to claim is that? X and S of x are going to get different colours.

(Refer Slide Time: 24:11)



That is C of x and C of S of x are different. So, how do I claim this? Suppose is not equal to j that is C of x and C of S of x are differ only at the ith position or lesser significant positions are the same for them and C of S of x and C of S of S of x differ in the j th position and all less significant positions are the same for them and let us say i and j are not the same therefore, any ordered pair of this form is not the same as any ordered pair of this form because they differ in the first component.

Therefore, the new colours of x and S of x would certainly be different, but as i is equal to j then the 2 ordered pairs of starting with the same first component, but I claim that they would still be different; that is because x is going to use its own bit at the ith position. So, here it would be using C i of x to form its new colour.

So, the new colour of x would be i C i of x whereas, the new colour of S of x would be I concatenated with C i of S of S of x sorry C i of S of x. So, what it means is that, the new colours of C of x the new colours of accent S of x are still different even after the iteration, which means the iteration would keep the colouring valid, provided that the previous colouring was valid. So, the first colouring that we started with in which we

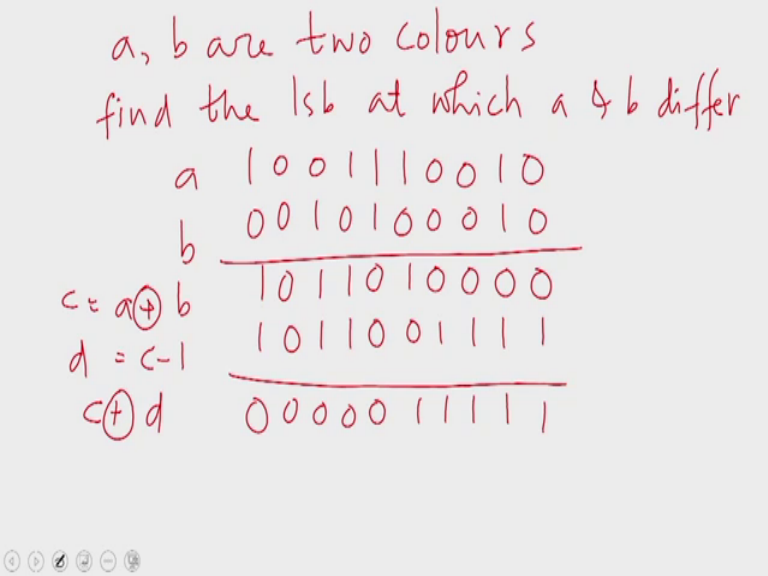
gave a unique colour to every vertex was a valid one therefore, the colouring that we obtain after every iteration will be valid therefore, we are finding valid colourings through the algorithm.

Now there are a couple of loose ends to be tied one is the last vertex, the logically last vertex in the list this vertex, does not have a successor, but then in the algorithm every vertex is supposed to look at the successor and take the least significant bit at which its colour and the successors colour differ. Since this vertex does not have a successor where would it look? Therefore, we should provide it with a successor. So, for the purpose of colouring we would assume that the first vertex is its successor.

So, we would make the last vertex point to the first vertex thereby making the list is circular one, but this is only for the purpose of finding the colouring. After finding the colouring for the other vertices of the list we can again remove this link. So, for the purpose of colouring we would make the last vertex point to the first vertex. Now the question is how do we find all those? That is in each iteration we are supposed to find for each vertex the least significant bit at which 2 colours differ.

So, how do we find the least significant bit at which 2 colours differ? So, let us say a and b are 2 colours, let us say we want to find the least significant bit at which a and b differ.

(Refer Slide Time: 28:21)



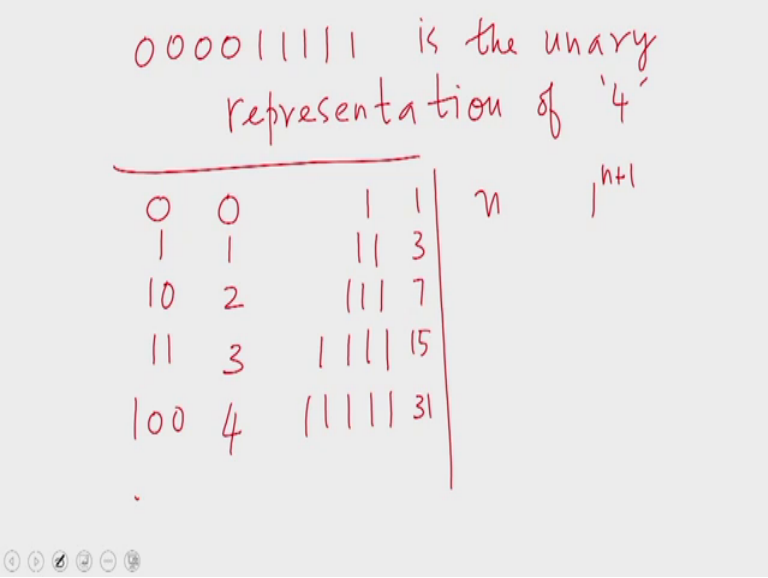
We want to find this in order of one time so, that each iteration of our algorithm will run in order of one time. So, what we do is this? Let me explain this through an example let me take 2 integers a and b. So, let us say these are the 2 integers a and b. Let me take the bitwise exclusive are of a and b. If I take the least bitwise exclusive of a and b I will get a 0 here, I will get a 0 here I will get a 0 here, get a 0 here and here I will get a 1.

So, the first one from the right end is at the position that we want. So, if you count from the right end 0 1 2 3 4 at the fourth position the 2 integers differ and the 1 that appears the first one from the right end appears it at position 4. So, we have to extract 4 out of this number. So, let us see how to do this let this integer be called C, now let me subtract 1 from C which I call d.

So, C minus 1 is what I call d. So, if I take C minus 1 what I get is this. Position 4 has a 1 in C and every lesser significant bit is 0 therefore, when I subtract 1 from C, these lesser significant positions will all flip to 1 and position willful will flip to 0 and the remaining positions will remain what they were. Now let me take the exclusive or of C and d, I will get a 0 at all these positions, but I will get 1 at all these positions.

So, we find that there are 5 ones in the result in the binary representation of C exclusive of d, we have 5 once at the least significant bit positions and every other position is 0.

(Refer Slide Time: 31:31)

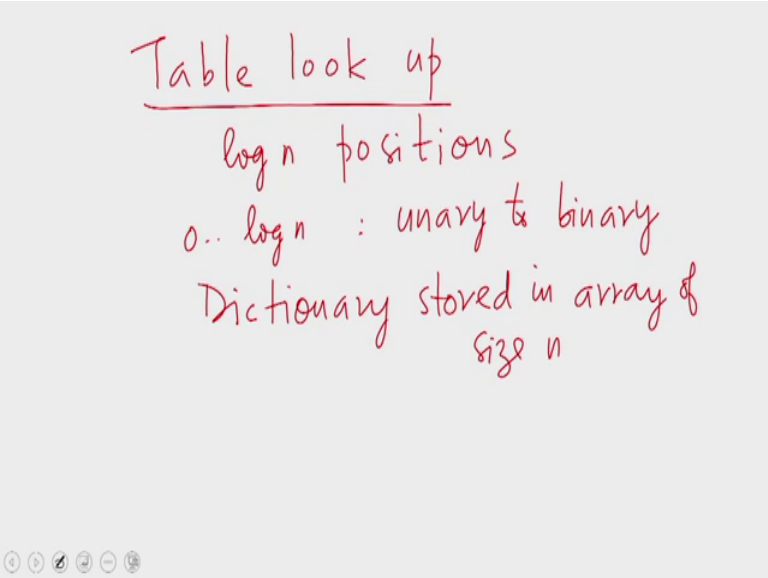


This is nothing, but the unary representation of number 4. In unary representation we represent 0 using 1, 1 using 1 1, 2 using triple 1, 3 using double 1 double 1, 4 using 1 double 1 double 1 or in general we will represent n using n plus 1 consecutive ones.

So, you find that the unary representation of a number is much larger than its binary representation. Now what we have done here is to find the unary representation of the number that we want. That is to find the least significant bit at which 2 colours differ; we should take the bitwise exclusive or of them and then subtract one from the bitwise exclusive or and then take the exclusive or of the numbers C and d and what we get would be the unary representation of the number that we want.

Now we should convert the unary representation into the binary representation; that is if you consider the binary representations here the binary representation of 0 is 1 of 1 is 1 and of 2 is 1 0, here it is 1 1, here it is 1 double 0 and so on. So, what we need is a conversion from the unary representation to the binary representation, we will do this using what is called a table look up.

(Refer Slide Time: 33:27)



Since we are dealing with colours in the range 0 to n minus 1, we require log n bit positions, each bit position can be represented using log n bits so, there are log n positions at the most.

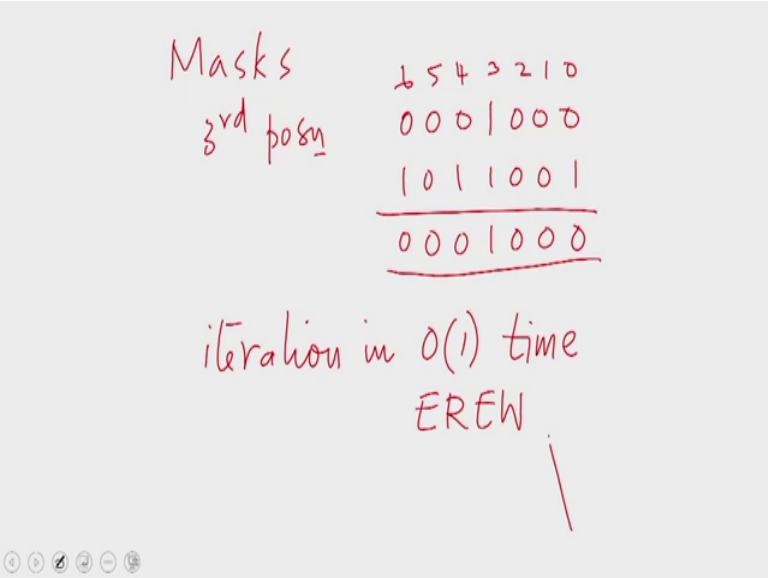
So, we might want to convert numbers in the range 0 to log n from unary to binary. So, let us see for this purpose we construct a dictionary. So, in the dictionary what we do is this, we will form the unary representations and the corresponding binary representations and store them at the appropriate positions. So, at what positions will be stored them this is the question. Now one is the binary representation of 1, 1 1 is the binary representation of 3, 1 1 1 is the binary representation of 7 and so on.

So, we can take an array in which we store these positions store these this dictionary. In the first position I will store the mapping from 1 to 0, in the third position I will store the mapping from 1 1 to 1, in the 7th position I will store the mapping from triple 1 to 1 0 and so, on. Now once I have stored the values in this fashion I can treat this array as an associative array. So, the dictionary is stored in an array of size n, but even though we have taken an array of size n we would be using only some locations of the array, we would be using in particular log n locations of the array. And once the array has been formed we can keep reusing this; however, many times that we want.

So, once a dictionary is formed we can keep reusing it as many times as we want and using the dictionary once requires only order one time as we have seen here. For example, if you want to find the binary representation of 1 1 1 1 all we need to do is to look at position 15 of the array where the value 1 1 will be stored, 1 1 is the binary representation of the unary number double 1 double 1 which stands for 3. So, the dictionary can be used in order of 1 time once it is formed.

Therefore, we do not consider the cost of forming the dictionary here, because that is a one-time operation and once it is formed we can keep reusing it as many times as we want for every execution of our algorithm. So, we rely on the fact that this dictionary is available and the dictionary can be looked up in order one time. Therefore, we now find the iteration that we specified earlier can be executed in order one time for every vertex. So, for every vertex in parallel we can compare the colours C of x and C of S of x and extract the least significant bit at which these 2 differ. Now the next step is to find the bit position of C of x at this position. So, we know that I can be found in order one time what is remaining is to find the bit of C of x at position i; for this purpose we can form what are called masks.

(Refer Slide Time: 37:07)

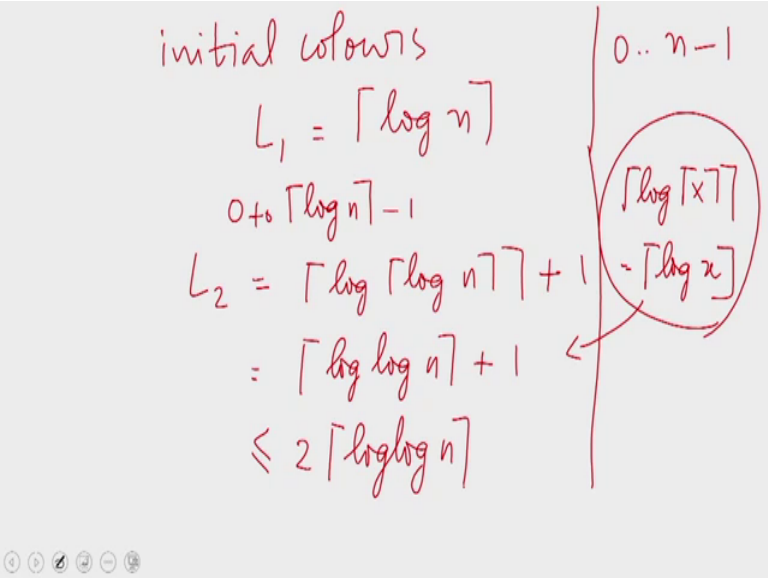


For each bit position let us say we have a mask, in particular for the 3rd position I have a mask like this. So, this is corresponding to the 3rd position. So, for each position I will keep a masks, I will prepare a mask in advance. So, that is done offline again like a dictionary and these masks again could be kept in a dictionary. So, to find extract the third number the third bit of a binary number, for example, is what we do is to take the bitwise and of this with the mask, which will give us.

And then if the numerical value of the result is 0, then the bit at that position is 0, if the numerical values not 0 then the bit at that position is 1. So, that is how we check how what the bit position at a particular position of an integer what the bit at a particular position of an integer is. So, combining all this we can execute an iteration in order of one time. And we have used only an EREW PRAM because there is never any concurrent read anywhere. Vertices need to look at their successors, but then every vertex will first access its own colour then they will access their successors colour.

So, when x is accessing the colour of S of x, S of x is accessing the colour of S of S of x so, again there is no read conflict. So, the iterations can be executed on EREW PRAM now let us see how many times the iterations have to be executed.

(Refer Slide Time: 39:19)

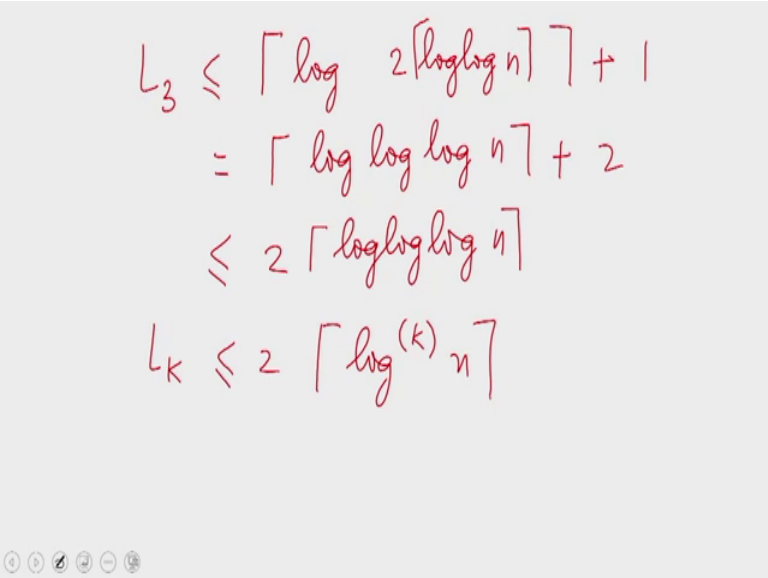


The initial colours had a length of L 1 equals ceiling of log n that is the colours range from 0 to n minus 1, each colour could be represented using ceiling log n bits. Therefore, the initial colours had a length of ceiling log n and then the second set of colours were formed by taking these bit positions the binary representations of these bit positions and concatenating them with 1 bit. Therefore, we needed to represent the positions 0 to ceiling log n minus 1, this range we have to represent in binary that would require ceiling of log of log n bit positions.

So, the new colours the second set of colours required these many bits plus 1 because we are concatenating them with one extra bit. Therefore, the exact length of the second set of colours each colour would be ceiling of log of ceiling of log n plus 1 bits, but then this is nothing, but ceiling of log of log n plus 1 bit positions that is because for every real number ceiling of log of ceiling of x is the same as ceiling of log x.

So, you can prove this as an exercise am not going to prove it here, but I will be using it here using this equality we write L 2 as ceiling of log of log n plus 1. And if n is sufficiently large, I can say this is less than or equal to 2 of 2 times ceiling of double log n. So, we find that L 2 is utmost twice double log n, these are the colours that we have after the first iteration or at the beginning of the second iteration.

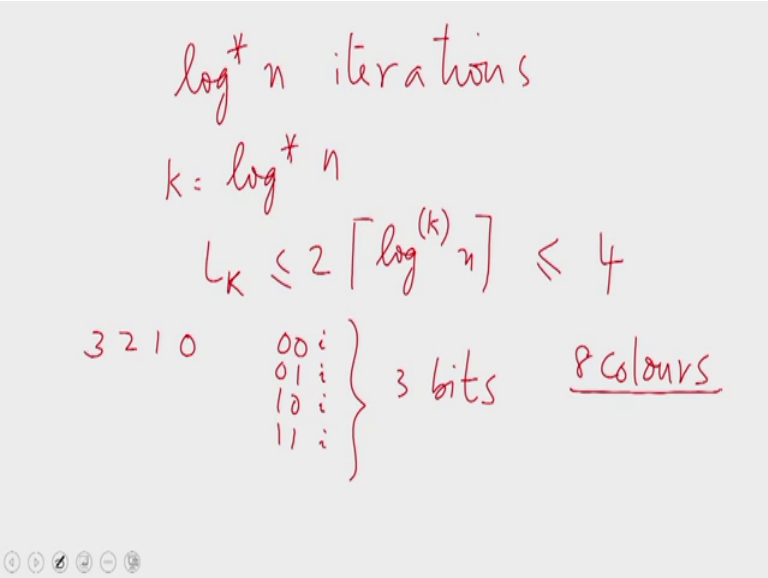
(Refer Slide Time: 41:53)



Now, in the second iteration continuing in the same fashion we find that L 3 is atmost ceiling of log of ceiling of 2 log of log n this ceiling is not needed, which is equal to ceiling of log of log of log n which is triple log n plus 2, which you can write as utmost twice triple log n if n is sufficiently large. So, for a large enough n you can write L 3 in this fashion.

So, continuing like this, we find that L K is less than or equal to twice log k of n that is logarithm applied k times on n, provided that n is sufficiently large, if n is not large enough for the algorithm to proceed for k iteration. So, much the better the algorithm would terminate earlier. So, in the worst case n is very large and the algorithm proceeds for k steps in that case the colours the colours that we use will have a length of utmost this much, now let us say we continue the algorithm for log star n iterations.

(Refer Slide Time: 43:27)

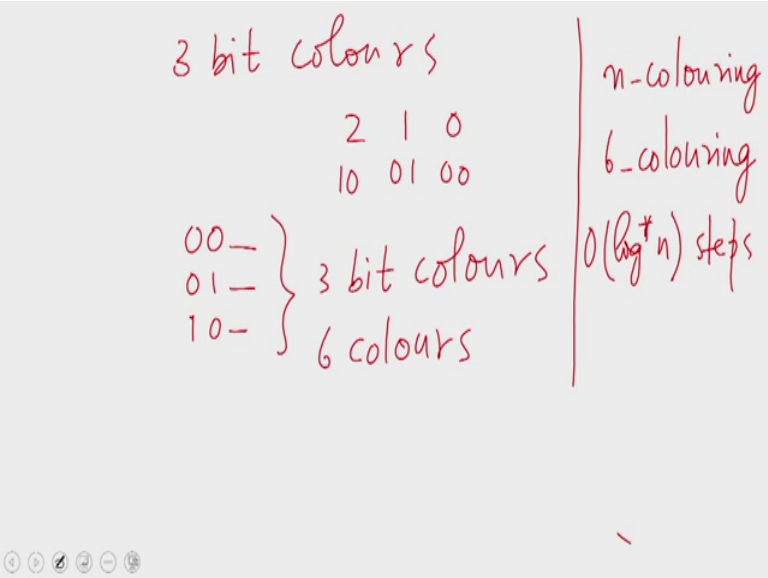


So, if k equals log star n we find that LK is less than or equal to twice log k n, but log k n by the definition of log star n is less than or equal to 2 therefore, this is less than or equal to 4.

So, if you continue the algorithm for log star n iterations, the length of the colours would reduce to 4. If you repeat the algorithm for one more step, we would be forming 3 bit colours that is because now we have 4 colours after log star n steps we are left with 4 colours the 4 bit colours. So, the big positions can be represented using 0 1 2 and 3. So, there are 4 bit positions to represent these can be represented using binary representation 0 0 0 1 1 0 and 1 1. So, you require 2 bits to represent the position numbers and then you concatenate 1 bit in the end.

So, you require a total of 3 bits to represent the new colours. So, if you repeat the iteration, we will be left with 3 colours 3 bit colours; 3 bit colours mean 8 colours. So, now, we have reduce the initial n colouring of the linked list to an 8 colouring of the list. So, to recap we run the algorithm for log star n iterations, when we put k equal to log star n we find that the length of the colours that are left would be utmost 4. So, now, we are left with 4 bit colours; 4 bit colours entail 16 colouring. If you repeat the algorithm once the number of bits would reduce by 1 again. So, we will be left with 3 bit colours; 3 bit colours entail 8 colours let us see what happens if we repeat the colouring once again.

(Refer Slide Time: 45:41)

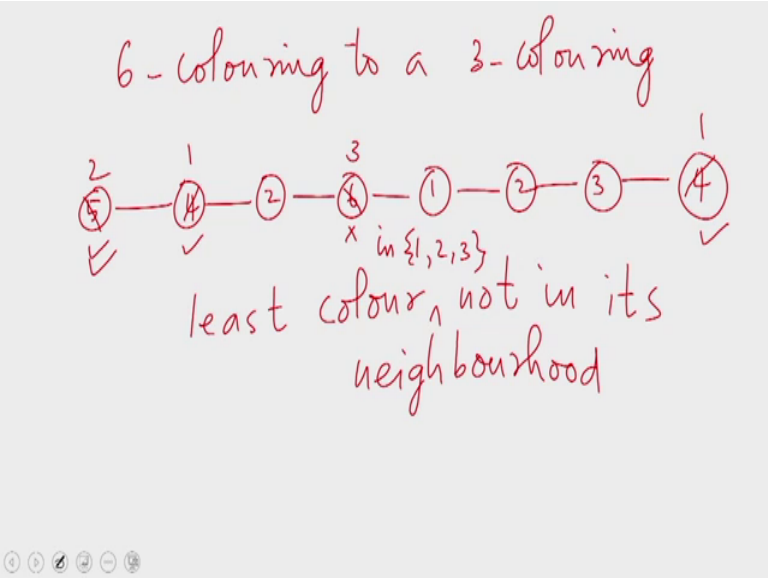


So, right now we have 3 bit colours so, the big positions are 0 1 and 2 there are only 3 positions the binary representation would be 0 0 0 1 and 1 0 therefore, if we repeat once again the log star n plus second iteration, will cause us to have colours of this form 0 0 with 1 bit concatenated 0 means 1 with bit concatenated 1 0 with 1 bit concatenated.

So, we are again left with 3 bit colours, but then instead of 8 colouring now we have a 6 colouring. Because these 3 bits would represent only these combinations 0 0 0 1 and 1 0 therefore, the 2 initial bits would be representing only these 3 combinations. So, in effect we have 6 colours. So, we have reduced from and n colouring to a 6 colouring, we have reduced the list from an n colouring to a 6 colouring in order of log star n steps. But then as I mentioned before log star n utmost 4 for all real values of n therefore, log star n plus 2 is utmost 6. So, for all real values of n this algorithm would execute utmost 6 steps, and we would have come down to a 6 colouring.

So, what we began with was n colouring and what we have now is a 6 colouring, but what we want is a 3 colouring. So, on an EREW PRAM, we have managed to reduce n colouring 2 or 6 colouring now we need to go and convert this 6 colouring into a 3 colouring then we would be done.

(Refer Slide Time: 47:45)



What we do is this? Let us consider the list let us say we have some colouring. So, we have a linked list that is 6 coloured we want to reduce this to a 3 coloured what we do is, this initially we make every vertex sleep we will wake up only those vertices that have a colour of 4. So, this vertex is woken up and this vertex is woken up. Every vertex that is of colour 4 is now awake this vertices will look to both sides.

So, this vertex looks to either side it finds that has a 2 1 one side and 5 on the other side it does not have a 1. So, what this vertex does is to adopt the least colour which is not in its neighborhood, every vertex that is woken up adopts the least colour not in its neighborhood, but remember we have woken up vertices of colour 4 no 2 of them are adjacent.

So, when they look around in the neighborhood they are not conflicting with each other because there are 2 there are not too vert colour 4 vertices that are adjacent to each other. So, in this example there are 2 colour 4 vertices when they look around, they find that colour 1 is absent for both of them so, they will adopt colour 1 each. And next we wake up every vertex of colour 5. So, in this examples there is only one vertex of colour 5 that is woken up when that vertex looks around it finds that it has only 1 colour in the neighborhood which is 1 and it will adopt the least colour not in its neighborhood.

What I mean is it the least valid colour. We are considering only colours in the range 1 2 3 these are the colours that we finally want. So, the vertices are looking for the least

colour least valid colour not in the neighborhood. So, vertex 5 finds that it has 1 in the neighborhood and the least colour not in the neighborhood is 2 therefore, this will be recoloured as 2.

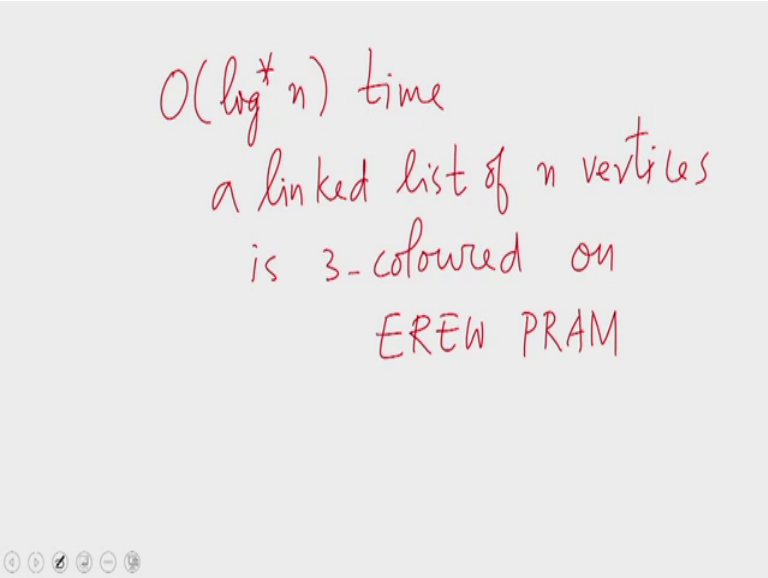
Now we come to vertices of colour 6. So, when I wake up vertex of colour 6 and it looks around it finds that it has both the colours 1 and 2 in its neighborhood therefore, the least colour which is not in its neighborhood is 3. It ends up adopting the third colour they will always be a third colour because what we are considering as a linked list therefore, every vertex has a neighborhood of size 2. So, there can be utmost 2 colours in the neighborhood therefore, the will always be a third valid colour not in the neighborhood so, every vertex has a colour to adopt.

So, we are waking up the vertices so, that no 2 awake vertices are adjacent to each other. So, in a sense the awake vertices will form an independent set and then they can all adopt a new colour without conflicting with each other. Because no 2 of them are adjacent and they will always be one free colour for them to adopt because every neighborhood has a size of utmost 2.

The size could be 1 also when we considered this vertex of colour 5 it had only 1 neighbor. So, the size of the neighborhood is utmost 2 therefore, suddenly there is one free colour as in the case of this vertex of colour 6 it has 2 neighbors of colours 1 and 2 each so, colour 3 is free. So, that is what this vertex adopts. So, we find that after doing all this the list is now 3 colour. So, we have taken 3 additional steps we assume that every vertex has a processor, with this assumption we can wake up any vertex that we want.

So, we wake up every vertex of colour 4 simultaneously all of them will act simultaneously they will scan the neighborhood, the neighborhood has size of utmost 2 and then it will find the least column not in the neighborhood and adopt that colour. So, this will take only order one time for each colour. So, we consider the 3 colours 4 5 and 6 so, a total of order one time is necessary to reduce the 6 colouring 2 or 3 colouring.

(Refer Slide Time: 52:49)



To summarize what we have got is that, in order of log star n time a linked list of n vertices is 3 coloured on EREW PRAM that is it from this lecture hope to see you in the next lecture.

Thank you.