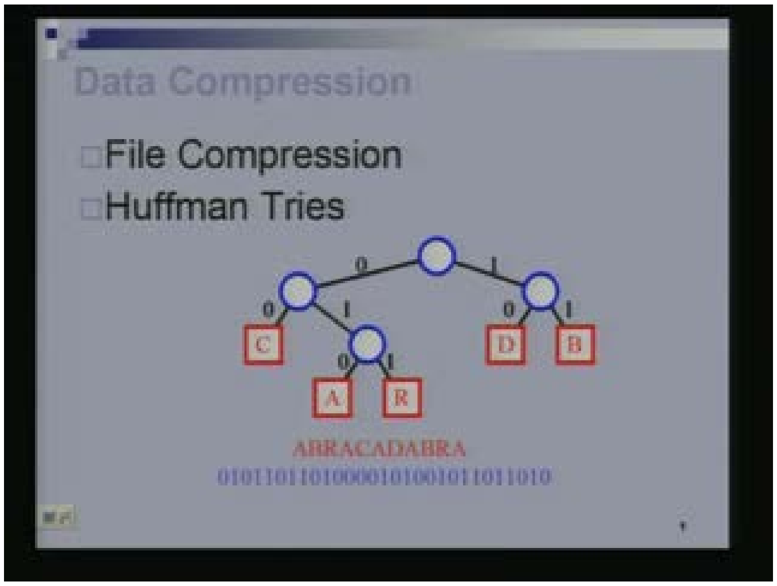
Data Structures and Algorithms

Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 19
Data Compression

Today we are going to be talking about Data compression.

(Refer Slide Time 01:17)



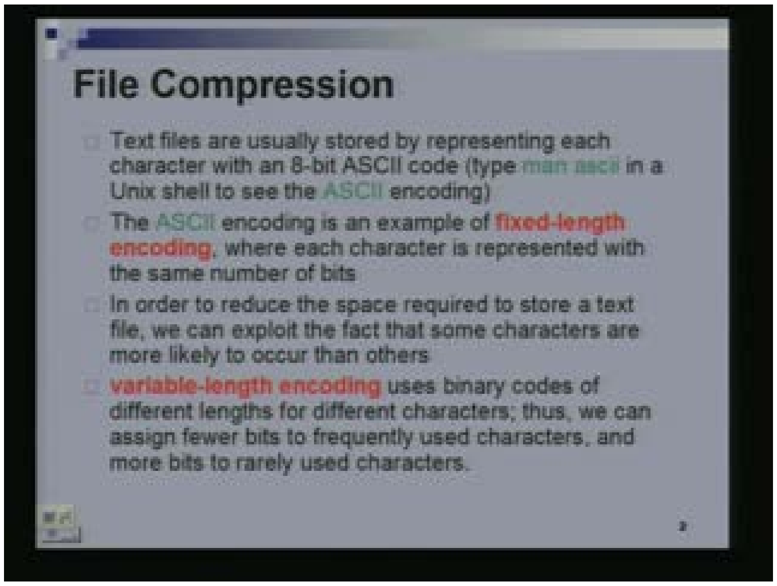
Data Compression

- File Compression
- Huffman Tries

ABRACADABRA
01011011010000101001011011010

We will begin with what the idea behind file compression is and then we are going to be talking about Huffman Tries which is the way of doing data compression. We are going to see how "ABRACADABRA" translate into these sequence of 0's and 1's. So what is file compression? As you know, if you have a piece of text, it's stored as bits in your computer and what is typically done is that, for each character, you have what's called an 'ASCII code'. So if you were to go into a unique shell and type <man ASCII>, then that will give you the ASCII code for all the various characters.

(Refer Slide Time 03:31)



File Compression

- Text files are usually stored by representing each character with an 8-bit ASCII code (type man ascii in a Unix shell to see the ASCII encoding)
- The ASCII encoding is an example of **fixed-length encoding**, where each character is represented with the same number of bits
- In order to reduce the space required to store a text file, we can exploit the fact that some characters are more likely to occur than others
- **variable-length encoding** uses binary codes of different lengths for different characters; thus, we can assign fewer bits to frequently used characters, and more bits to rarely used characters.

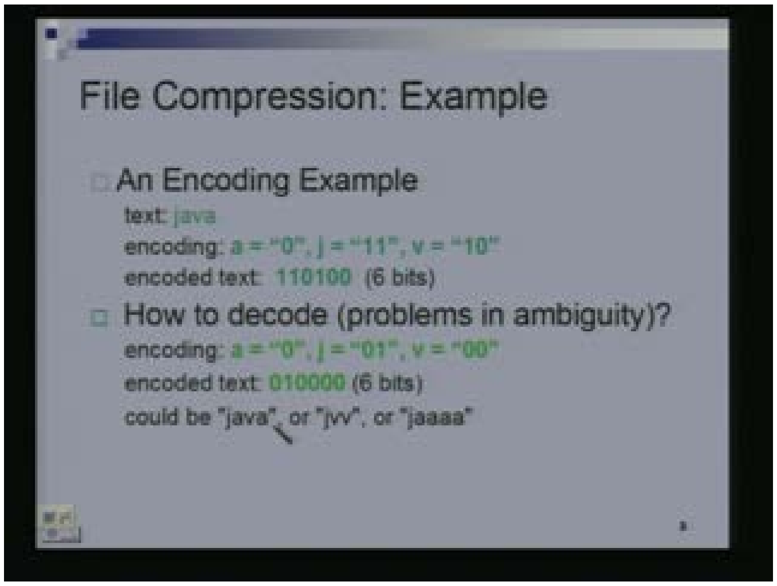
2

The ASCII code is an 8 bit code which means that every character is stored as 8 bits. So this is what is called ‘fixed-length encoding’. why fixed length because for each character, I have the same number of bits but our idea today is to try and reduce the amount of space required to encode a piece of text. If each character I am going to use 8 bits then the total number of bits required will be 8 times the number of characters in the piece of text. But suppose I don’t have to do fixed-length coding.

You know some character might have two bits associated with them. Some character will be encoded using three bits, some using four and so on, can we exploit this and the fact that some characters occur more frequently than others to design a coding screen which will represent the same piece of text using lesser number of bits using lesser number of bits. You understand the need for doing this kind of compression? Clearly the lesser memory you require, you know if you are transmitting the file, you have to send less number of bits.

If you are storing, you will have to store less number of bits and so on. So it’s very useful to be able to compress the information that you have. That will bring us to what we call variable-length coding. So the number of bits used to represent each character would be different. In particular, character which occur more frequently, we are going to represent them using less number of bits and use that. Characters which appear very infrequently, let’s say x or z in the English alphabet, we can have longer sequences. May be more than 8 bits. Let’s see how this is done. So let’s say my piece of text is just 4 characters java and I decide to encode ‘a’. How many characters are there?

(Refer Slide Time 07:00)



File Compression: Example

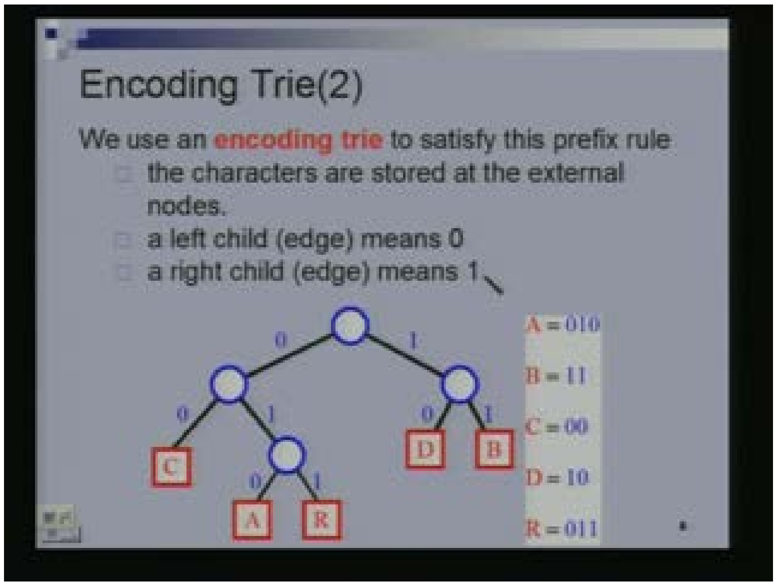
- An Encoding Example
  - text: java
  - encoding: a = "0", j = "11", v = "10"
  - encoded text: 110100 (6 bits)
- How to decode (problems in ambiguity)?
  - encoding: a = "0", j = "01", v = "00"
  - encoded text: 010000 (6 bits)
  - could be "java", or "jvv", or "jaaaa"

Suppose my alphabets were just a, j and v. The only things that ever encoded were strings on this alphabet a, j and b. So java is1 example of such a string and suppose I were doing fixed length encoding, how many bits should I associate with each of these? I will need at least two. I can't do with just1 because there are three different characters and there are only two possible values if I choose1 bit. So I need at least 2 bits and if I take 2 bits then how many bits do I need to represents java? 2 times 4 is 8. As straight forward as that. But suppose I decide to use 0, just single bit for 'a' and11 for j and 10 for v and I will tell you why I am doing this. Then java can be encoded as110100. That would be the encoding and it will take only 6 bits. It will take the lesser number of bits. Then you can ask me, "well, why did I do j as11 and v as10?" so the problem with variable length decoding - variable length encoding is that of decoding. How do you decode? Given a sequence of bits you want to decode it uniquely. So suppose I gave you a sequence of bits, then you should be able to retrieve java from this. Of course I've told what the codes were. You should be able to get back to java. Suppose for instance, I had used this as my encoding, for 'a' I use 0, 'j' is 0 1 and 'v' is00. So still it should take 6 bits only and the encoding would be 01000 and 0. From here can I get back to java given this code? Well 01 could be either 'a' or it could be a 'j'. It has to be a 'j' because we are using this '1'. So what would you do with these 4 0's? It could be 'java', it could be 'j v v' or it could be 'ja'. It's ambiguous. You see the problem? If you have to use variable length encoding, then you could have this problem of ambiguity while decoding.

So you have to be careful when you are using variable length codes. This problem would not arise when you have fixed length codes. You understand why because you will take those many bits and then you now determine what exactly the character was. So to prevent ambiguities in decoding, we will ensure that our encoding satisfies what's called 'the prefix rule' which is very simple. It says that no code is a prefix of another code. By code, I mean the bits I use for a particular character. When this was our code, you can see 0 was not a prefix of either j or v. j and v were also not prefixes of each other. The encoding arising out of this would be unambiguous and we will see an example to show that to illustrate that. But the encoding arising out of this

(Refer Slide Time: 08:15) will be ambiguous because ‘a’ is the prefix of these (Refer Slide Time: 08:21) and I will show you an example. You must understand what the prefix rule is. So if your codes satisfy the prefix rule, then decoding will be unambiguous. But if it does not, then you will have ambiguity in decoding. So I will come back to the prefix rule in the next slide. Code is the collection of code words.

(Refer Slide Time 08:48)



Encoding Trie(2)

We use an encoding trie to satisfy this prefix rule

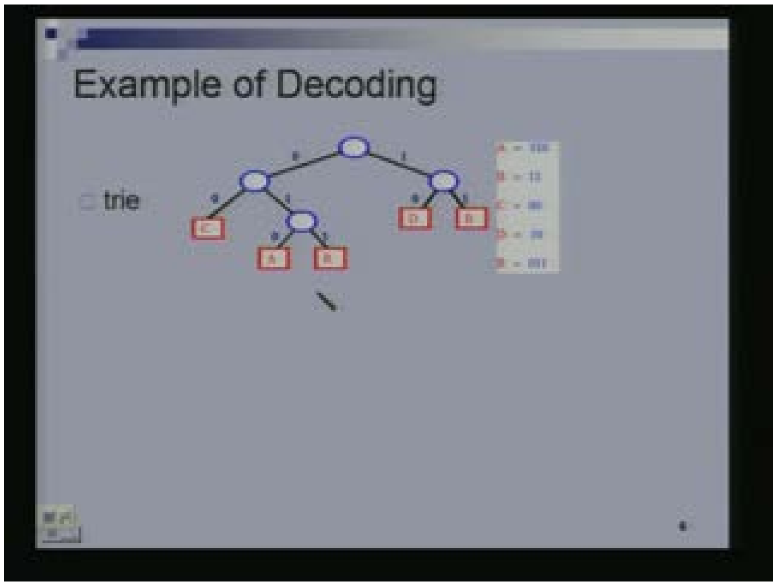
- the characters are stored at the external nodes.
- a left child (edge) means 0
- a right child (edge) means 1

A	010
B	11
C	00
D	10
R	011

What I have written out here is a code. Each one of these is called a code word. For each character the sequence of bits what’s called the code word and the entire thing is called a code. So code which satisfies the prefix rule can be represented as a tree. In particular as a trie. So recall the ‘trie’ that we have discussed in the last class, the branching was a 26 way branching. But here branching will only be a 2 way branching. Each node will have only a 2 children. Here my alphabets are a b c d r. five characters only.

The characters will be stored at the leaves or the external nodes and every for every node the left edge will label with 0 and the right edge will be labeled with 1. Now a = 010 because you know if you look at this, from root two A, you will encounter 010. When you are coming to R, the code for R will be 011. Can you see that if I drew such a picture for you, then the code word for any character will not be a prefix of the codeword for some other character. Otherwise it could have ended midway. But it’s not because everything is a leaf. Each character corresponds to a leaf. That’s the very first statement here. So the code word for one character would not be a prefix of code word for another character. We will represent our codes using such a we trace a from the root to the particular leaf to determine the code word for each character. I need not have shown this picture at all. I could have just drawn this trie and from this you can figure out what is the code word corresponding to this. Now how do we do the decoding? Suppose this is my trie and these are the code words (Refer Slide Time: 11:33).

(Refer Slide Time 11:28)

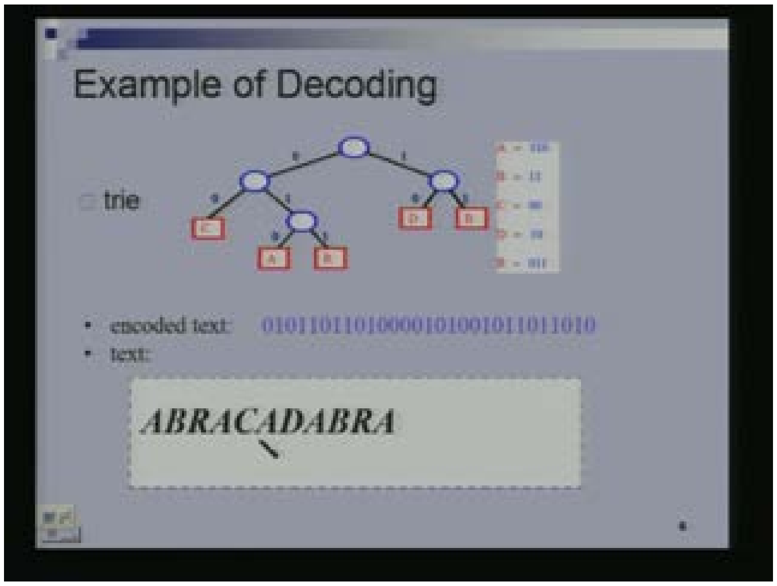


Example of Decoding

- trie

Now I give you a sequence of bits that’s this is my encoded text and I have to decode this text (Refer Slide Time: 12:01). As you can see, the code satisfies the prefix rule. So how do I do the decoding? So I start from the beginning. You always start from the beginning. You start from the beginning 010. So you will trace out 010. You will get a leaf. You stop and these 3 characters go away and I get an ‘a’. So I have struck off these 3 and I have written down an ‘a’. Now I will take 1 and the next 1 is also1. So I get a ‘b’ and I will strike these two off. I have taken care of these two. Now I get a 011. Its 01.1 so 0 remains. As you can imagine it will come to the same ‘abracadabra’. So that’s what this will turn out to be and we can decode it and see.

(Refer Slide Time 13:01)



Example of Decoding

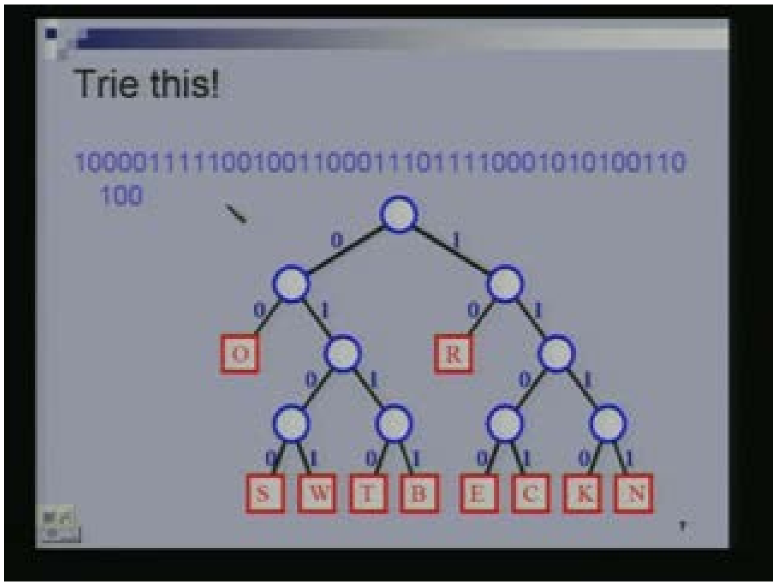
- trie

- encoded text: 01011011010000101001011011010
- text:

ABRACADABRA

Suppose I give you this trie and I give you this encoded text, how much time does it take to decode? Clearly I am just looking at a bit and I am going to spend 1 unit of time with every bit. I just look at that bit and go down one level in that trie.so it's basically length which you have to spend clearly. So this is another trie (Refer Slide Time 13:55) and you know this is a long piece of encoded text and you can figure out what this is.

(Refer Slide Time 13:55)

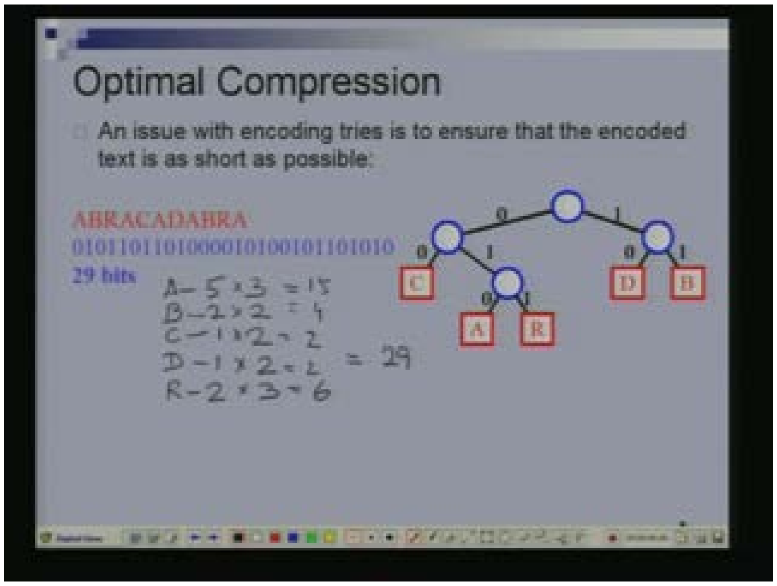


Trie this!

10000111110010011000111011110001010100110
100

You can take this is an exercise. So what is our aim in doing this?

(Refer Slide Time 15:52)



Optimal Compression

- An issue with encoding tries is to ensure that the encoded text is as short as possible:

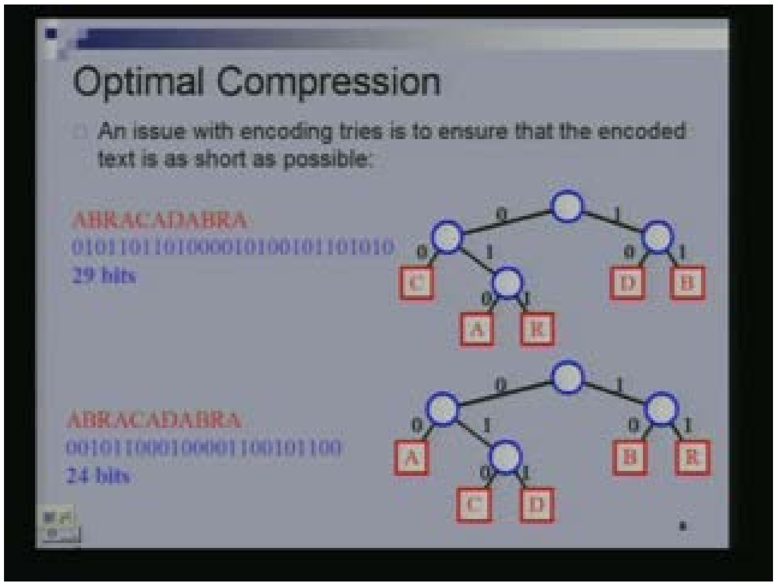
ABRACADABRA
01011011010000101001011011010
29 bits

A – 5 × 3 = 15
B – 2 × 2 = 4
C – 1 × 2 = 2
D – 1 × 2 = 2
R – 2 × 3 = 6

= 29

Recall we wanted to build up a code in such a manner such that the total length of the encoding is as small as possible. Suppose this trie was specifying my code and once again I was encoding abracadabra, this has 29 bits. How will you compute such a thing? Well let's quickly do that to make sure that you understand. What is the frequency of each character? A – 5,B - 2, C – 1,D – 1 and R – 2. Im using 3 bits for A, 2 for B, 2 for C, 2 for D and 3 for R. I am just counting the number of bits. So this becomes 15, 4,2 and 6.29 totally. Now suppose I had another trie say this one (Refer Slide Time 16:06), you think this will have less or more?

(Refer Slide Time 16:34)



Optimal Compression

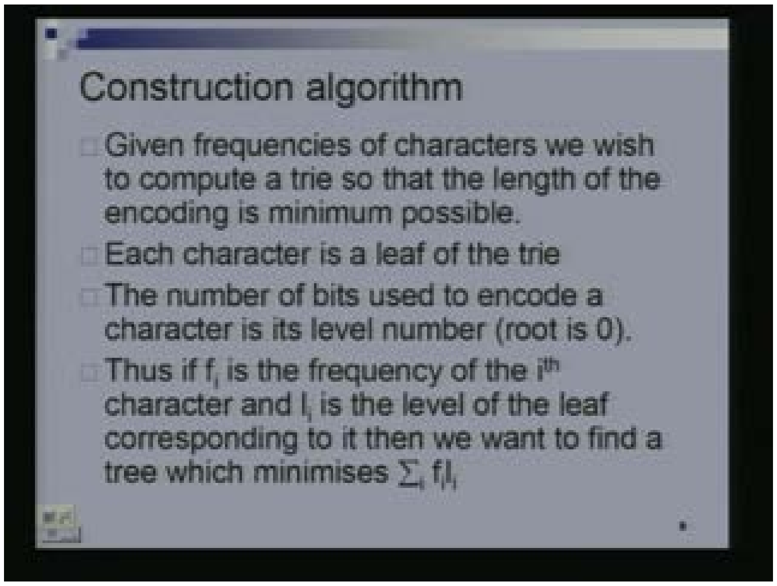
- An issue with encoding tries is to ensure that the encoded text is as short as possible:

ABRACADABRA
01011011010000101001011011010
29 bits

ABRACADABRA
001011000100001100101100
24 bits

This will have less. 'A' was occurring 5 times. Here I'm using 3 bits and I am using 2 bits here. I'm saving 5 bits for 'a'. Of course, I will have to compensate elsewhere. c and d, there are 2 here and 3 there. But c and d occur very infrequently. So it's good. This will have less than 29. We are also saving on R. this you can check. It will require 24 bits. We have to design this in such a manner so that the number of bits required it as little as possible. So let's try and understand. What is it that we are given? We are given the frequency of that character. Suppose I give you a piece of text. You will count the frequency of characters and we are trying to compute the trie so that the length of the encoding we get is as small as possible.

(Refer Slide Time 17:36)

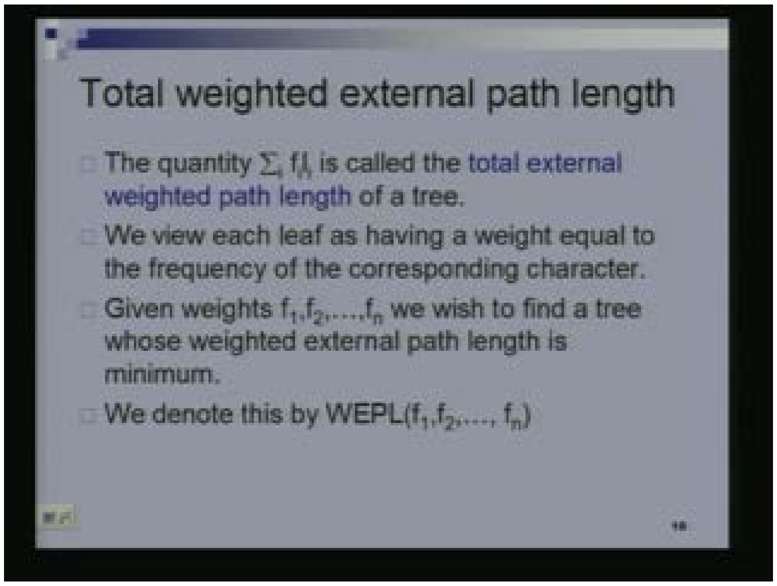


Construction algorithm

- Given frequencies of characters we wish to compute a trie so that the length of the encoding is minimum possible.
- Each character is a leaf of the trie
- The number of bits used to encode a character is its level number (root is 0).
- Thus if f_i is the frequency of the i^{th} character and l_i is the level of the leaf corresponding to it then we want to find a tree which minimises $\sum_i f_i l_i$

Recall that each character is a leaf in our tree. Number of bits used to encode the character is its level number where I'm assuming that the root is number 0. So if the i^{th} character has frequency of f_i and has level number of l_i, then what is it that we are trying to minimize? It is summation f_i l_i. so we have to choose a tree so that this quantity is minimized. Our tree will determine the l_i's. f_i is given to us. We cannot change the f_i's. We can pick a tree so that we can get appropriate l_i's. So summation f_i l_i is called the 'total external weighted path length' of a tree. It should be very easy to see why.

(Refer Slide Time: 19:51)



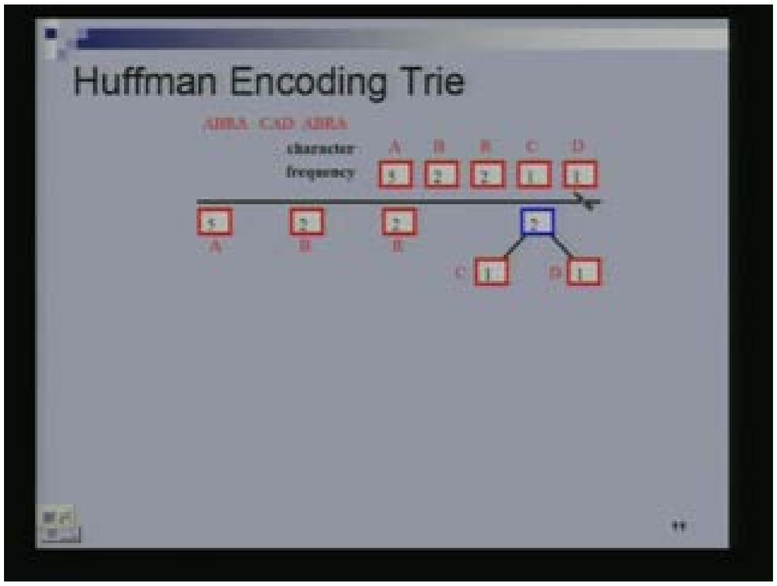
Total weighted external path length

- The quantity $\sum_i f_i l_i$ is called the total external weighted path length of a tree.
- We view each leaf as having a weight equal to the frequency of the corresponding character.
- Given weights $f_1, f_2, \ldots, f_n$ we wish to find a tree whose weighted external path length is minimum.
- We denote this by WEPL($f_1, f_2, \ldots, f_n$)

External because we are talking about external nodes which are our leaves. What is its path length? The length of the path from the root to the leaf which is basically the number of levels. 'Weighted' because we are multiplying it by the frequency and 'total' because we are summing it up. So we are viewing each leaf as having a weight which is equal to the frequency of the corresponding character. The weights in the weighted are referring to this frequency. From now on, I might call the same thing is weight or frequency. This means the same thing. Here for instance given these weights or frequencies, f1 through fn, we wish to find the tree whose total weighted external path length is minimum.

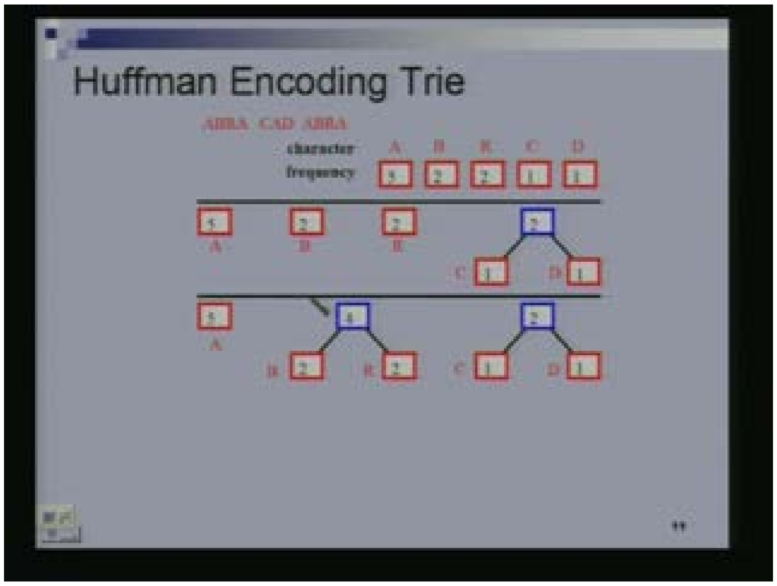
That's what we want to do. We are given the weight on the leaf. We want to build a tree whose leaves will be these and who's weighted external path length will be minimum and will denote this minimum weighted external path length by this quantity. so given 'n' leaves with weights f1, f 2, f 3,… fn; your problem is to build a binary tree whose leaves will be these 'n' leaves and which will have a minimum total weighted external path length. So we have managed to translate our question of finding the minimum length in encoding such that the length of the encoded message minimum to that of designing an appropriate tree. One thing I am going to assume is that when I write it in this way, f1 is smaller than f2 and these are in increasing order. Now let me show you what the algorithm is. Once again, my text is the same "abracadabra". There are 5 characters and I have put down the frequency of these 5 characters.

(Refer Slide Time 20:58)



Huffman Encoding Trie

(Refer Slide Time 22:13)

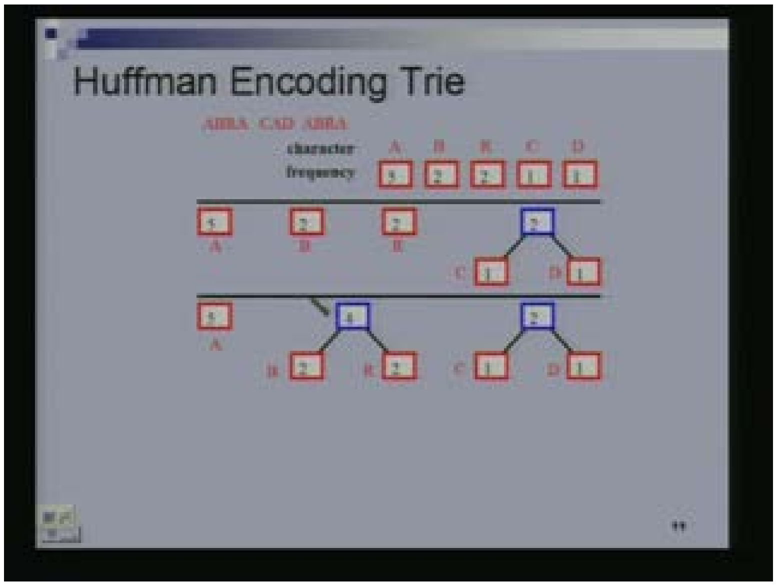


I have put red boxes around them. These will have to be the leaves of my tree. Now what I am going to do at the very step is I am going to take the 2 with the smallest value which are the ones in the ends. These are the two smallest ones. I am going to combine them. How am doing that? I am going to create another node. I am building up a tree now. These are my leaves. I make another node as the parent of these two and this node gets the value of weight of sum of these two. Now these two will disappear from the picture and I will just be left with 4 nodes and I will repeat the process. So what is the process? Take the two smallest and combine them together into one. Take the two smallest and combine them together into a node and when you combine, you basically sum up their weights. So at the next, we have an option .we can either combine b and r or we can combine r with this one (Refer Slide Time: 22:13) that we have created. Let's see which one I take. I decide b and r to be combined into one. When you have an option we can pick whichever you feel like.

So they have combined into one and what we are left is only three node. Now many times I will have to do this process? If I started off with 'n' leaves, how many times will I have to do this process? Every time you are reducing a node by 1. so it is (n -1) times not n by2. Now which will we combine? Two and four clearly because they are the two smallest ones.

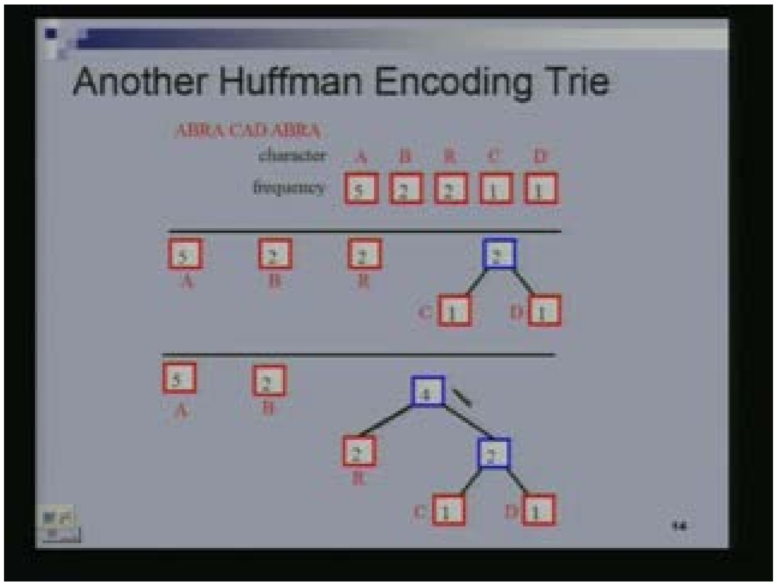
So you combine them into one and we get a six. Finally we have only two nodes left – 5 & 6. They will get combined. This is the picture. We will combine them into one and these are 11.now how do I label? I can just label whichever way I like to. It really does not make a difference. The length of the encoding was determined by the depth of these things. This becomes encoding now and the claim is this is the best. This will give you the same minimum. Whatever was the minimum for abracadabra will be achieved by this. As you can see 'a' which was occurring five times is getting only1 bit. Looks like it is the right thing to do. So this is our final trie.

(Refer Slide Time 24:07)


Huffman Encoding Trie

This is my text. This is the corresponding code. As you can see for ‘a’ you have 0. For b you have 100, the next three characters. For r you have 101, the next 3. For c you have 3 and so on and on. There should be a gap between this 101 and 0 because ‘a’ corresponds to the last one and these are only 23 bits. These is even better than the previous code which was 24 and this is the best possible which is what we will argue in this class. Can you do better than this? Let me take the same example and then build another trie. How can I build another trie? We recall that there was an option at each point. Let me take the other side of the option. Let me see what trie I get now. Here there are 2 minimums. Here there are 3 2’s (Refer Slide Time: 25:44). First I combined these two. Let me do that. I decide to combine r with this one (Refer Slide Time: 25:49). I get a four.

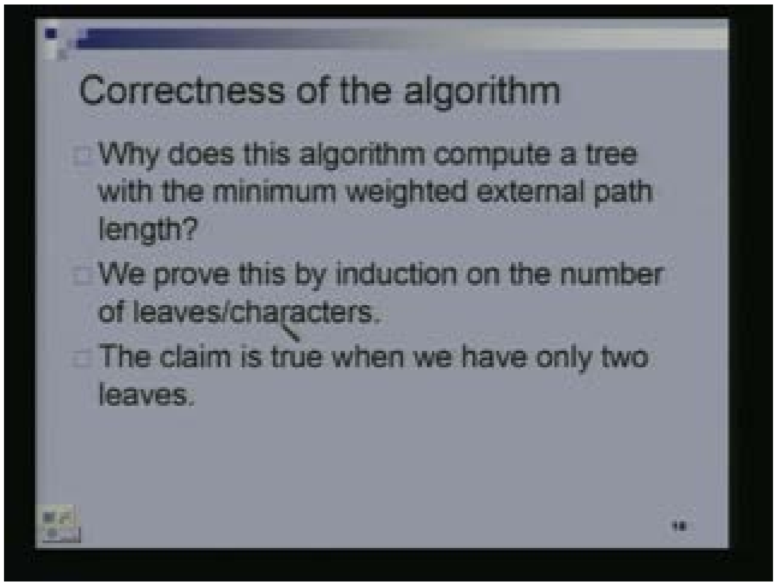
(Refer Slide Time 25:50)



Another Huffman Encoding Trie

Now which do I have to combine? This four with this two (Refer Slide Time: 25:55). I combine the four and the two and then I get six. Finally I am going to combine this and this. So this is the final trie I get. It's the same piece of text. Once again 'a' gets only 1 bit. b gets 2. r gets 3. 'a' gets 1. 'c' gets 4 now and d also gets 4. You think it will be different from 23? It should not be. Otherwise the theorem I am claiming is false. It should be the same. You can count. It will be 23 because this algorithm is computing the tree with the minimum weighted external path length. Since it's the minimum, it cannot be smaller or larger than the other one because they are both the minimum. So we now need to argue correctness. Why is this computing the minimum?

(Refer Slide Time 27:40)

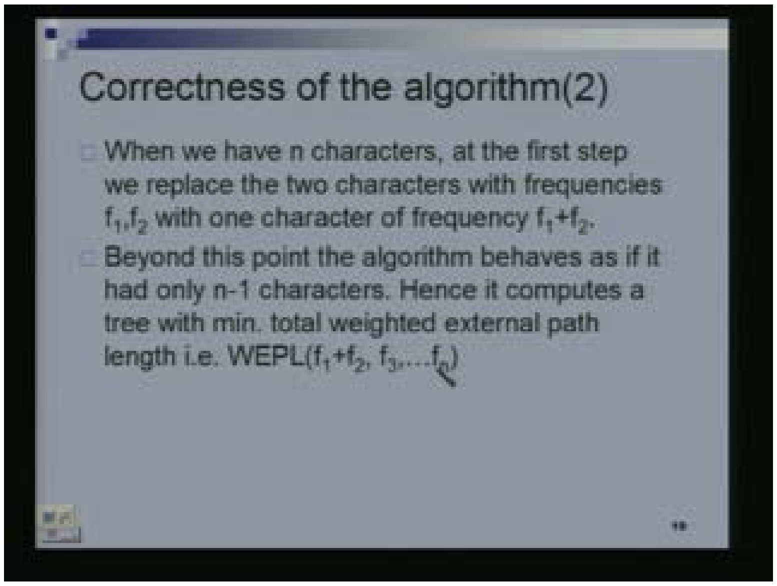


Correctness of the algorithm

- Why does this algorithm compute a tree with the minimum weighted external path length?
- We prove this by induction on the number of leaves/characters.
- The claim is true when we have only two leaves.

Why does this algorithm compute a tree with the minimum weighted external path length? So there's no reason to believe its too simple to do anything useful. Let us see what the argument for this one is. We will be proving it by using induction on the number of leaves which is same as the number of characters. Suppose I gave you only two leaves, then what is the algorithm going to do? It will just combine them into one and so it will basically give you a tree with the three nodes, one root. this will be 0 and this will be 1 (Refer Slide Time: 27:54) and this will be something and this will be something and clearly it's using 1 bit for each of the character, you cannot do better. You cannot take 0 bits for a character. so it's true when you have only two leaves. So we are going to assume the claim is true when you have (n -1) characters or leaves and we are going to show that it's going to be true when you have 'n' characters or leaves. So when we have 'n' characters, what are we doing at the very first step? We are taking the two characters with the smallest frequencies and combining them into one.

(Refer Slide Time 29:22)

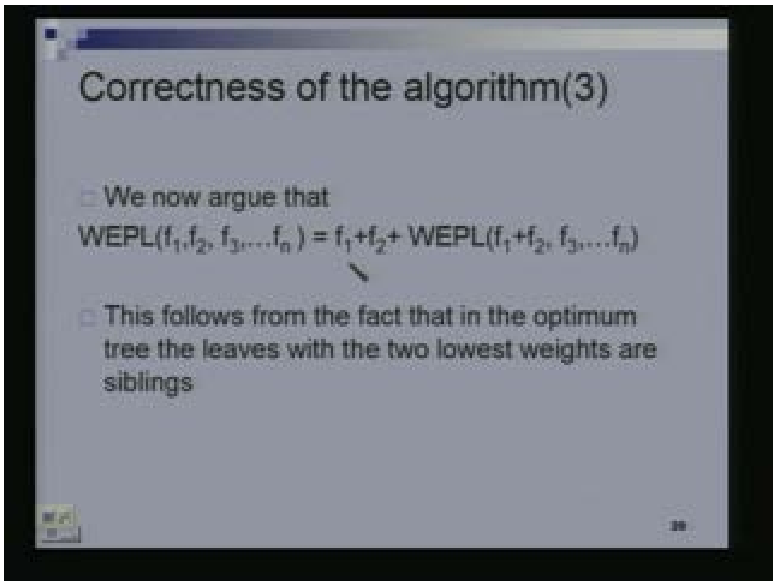


Correctness of the algorithm(2)

- When we have n characters, at the first step we replace the two characters with frequencies f_1, f_2 with one character of frequency f_1+f_2.
- Beyond this point the algorithm behaves as if it had only n-1 characters. Hence it computes a tree with min. total weighted external path length i.e. WEPL(f_1+f_2, $f_3, \ldots f_n$)

We are taking the two characters frequencies f1 and f 2 and we are replacing it with one node of weight f1 + f 2. It's as if you have one character now of frequency (f1+f2). so once it does that, beyond this point the behavior is as if it was given (n-1) characters with frequencies as f1 + f 2, f3, f 4, f5 upto f_n. Beyond that the point it's the same behavior. so beyond the point the algorithm behaves as if it had only (n-1) characters with frequencies f1+f2, f3 all the way up to fn. using our induction hypothesis it would have computed the best possible tree because these are only (n-1) characters now and it would have computed a tree on these (n-1) characters with total weighted external path length as this quantity. This was the minimum quantity. This was the notation we used for the minimum. The tree computed by this algorithm has weighted external path length f1 + f2+ this quantity (Refer Slide Time: 32:02). This is what our algorithm has computed. This is the weighted external path length computed by this Huffmen's Algorithm. Now we have to argue that this is the minimum possible. It will not go lesser than this. What we will argue is that the best solutions for f1 through f n equals f1 + f 2 + exactly the quantity that the algorithm had computed. We will argue this now.

(Refer Slide Time 32:56)



Correctness of the algorithm(3)

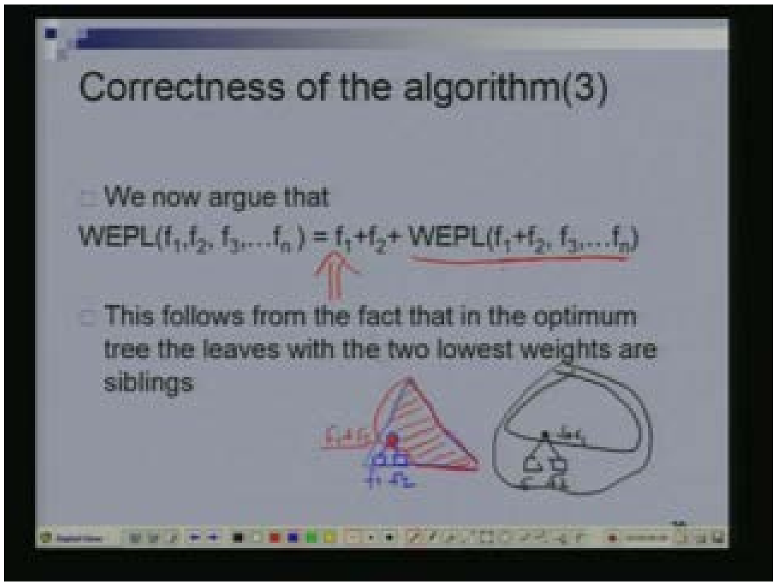
- We now argue that
$WEPL(f_1,f_2, f_3,\ldots f_n) = f_1+f_2+ WEPL(f_1+f_2, f_3,\ldots f_n)$
- This follows from the fact that in the optimum tree the leaves with the two lowest weights are siblings

This quantity that algorithm has computed is actually the best. It is the minimum weighted external path length when your weights are from f1 through f_n. How do we argue this? This will follow from the fact that in the optimum tree, by optimum tree I mean that tree whose weighted external path length is minimum. So let's prove this fact. Suppose this factor is true, how does this implies this? Why does this imply? Let's do one step at a time.

Suppose had proved this factor that in the best possible tree suppose we had prove that in the best possible tree that two lowest weights are siblings. We will do that in a slightly more formal way. Let's assume that the two minimum are always siblings. Let's argue that if this is true, then it will imply this (Refer Slide Time: 34:18). We have our best possible tree and the minimum are siblings. Why does it imply that we found the best> That's because it implies the best over f1 through f n equals this. This is the best tree. we are going from here to here (Refer Slide Time: 35:28) which means that we have assumed this statement and we are proving that the best tree over f1 through f n has weighted external path length equal to f1 + f 2 + the weighted external path length of the best tree over f1 + f2 through fn. so now we are kind of trying to mimic what we have already done. let me now look at this tree. Now I am just looking at the remaining tree. This is a tree (Refer Slide Time: 36:08) and let me give this node weight equal to f1 + f 2. Now this is the tree over leaves f1 + f2, f 3 up to f_n. What will its minimum weighted external path length become? If this is the best tree for the entire thing, for these values or leaves, this should be the best tree. We are looking at the best tree for f1 through f_n. I am saying the following.

Let's cut off the two leaves and just keep that. Let's give this a name of f1 + f 2. Then this tree is the best tree for these choice of weights also. Suppose there was something smaller possible, then this (Refer Slide Time: 37:21) blue would not have been and best tree for these guys. The best would be a green tree. So suppose the black tree was the best, better than this red one. (Hindi) This tree is the best possible tree when you have leaves with weights f1+ f 2, f3, f4, f5, f6. This is the best possible tree. So this red weighted external path length is this quantity then and so blue total external path length is red external path length + f1 + f 2.

(Refer Slide Time 40:51)



Correctness of the algorithm(3)

- We now argue that $\mathrm{WEPL}(f_1,f_2,f_3,\ldots f_n) = f_1+f_2+ \mathrm{WEPL}(f_1+f_2, f_3,\ldots f_n)$
- This follows from the fact that in the optimum tree the leaves with the two lowest weights are siblings

So which means blue weighted external path length which is this leaf is equal to this leaf which is equal to the best possible (Refer Slide Time: 41:28). What did we get in our previous algorithm? We got the right hand side exactly. So the algorithm is completing the best possible. If this black is better than the red, then what have I done? I have only increased the black by a quantity equal f1 + f 2 while this blue also differs from the red by this quantity f1 + f 2. If this black is better than this red, then this bigger black is also better than the bigger blue which violates our optimal case. But why is this statement true?

Why should it be the case that in the optimum tree, the leaves with those two lowest frequencies are siblings? Let’s take the leaf with the lowest weight. It will have the maximum level number. Suppose this leaf with the lowest weight comes in between and there is another leaf with a higher weight, this is not optimum. So we have to swap these two. So the total external weighted path length becomes minimum. So the leaf with the smallest weight has to be at the last level. Let’s look at its parent and let’s look at its siblings which is this (Refer Slide Time: 43:53) leaf. By the same argument this leaf is the second smallest weight because if it for anything else then once again you can swap and reduce. So the leaves with the two smallest weights are actually at the very last level. In all our examples you must have seen that happening. They are all at the last level and they are siblings. You just use this fact, make them siblings and then the problem reduces by 1 because now you have only (n -1) nodes. How do we take care of (n -1) nodes? It is the same way we took care of them. Once again you take the two smallest ones, make them siblings and continue. So just this one property being exploited in this algorithm. We got the very simple algorithm to compute the best possible trap. So with that I am going to stop today’s class. After we have done priority queues, we are we are going to analyze these particular algorithm to compute its running time. So I’m leaving the bit about computing its running time today and I will take it up after we have developed the notion of paradigms.