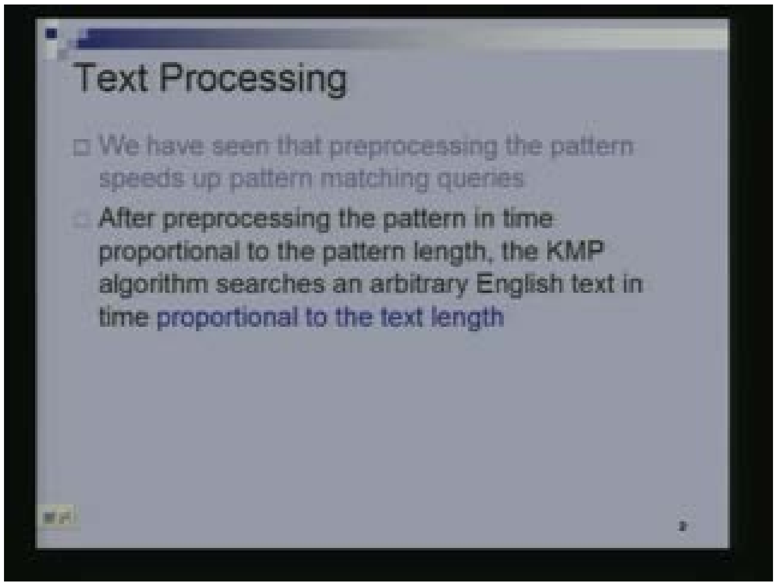
Data Structures and Algorithms

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

Lecture – 18
Tries

Today we are going to be talking about another data structure called "tries" and we are going to see it's used in pattern matching. so I am going to be starting off with what are called "Standard Tries" which is the plain version of tries and we are going to move on to "Compressed Tries". This is a space sufficient way of keeping tries and the last topic we are going to look at today is what are called "Suffix Trees". So first 2 terms have tries in them and the 3rd has trees in them. Recall in the last class we were looking at pattern matching. Given a piece of text, we were interested in matching patterns. Finding out at what all places certain pattern appears in the text and what we had done there if you recall was that we had preprocessed the pattern. That is, we took the pattern. We computed this failure function h on the pattern and then we used that information to search for the pattern in the text and the Time we took was proportional to the size of the text.

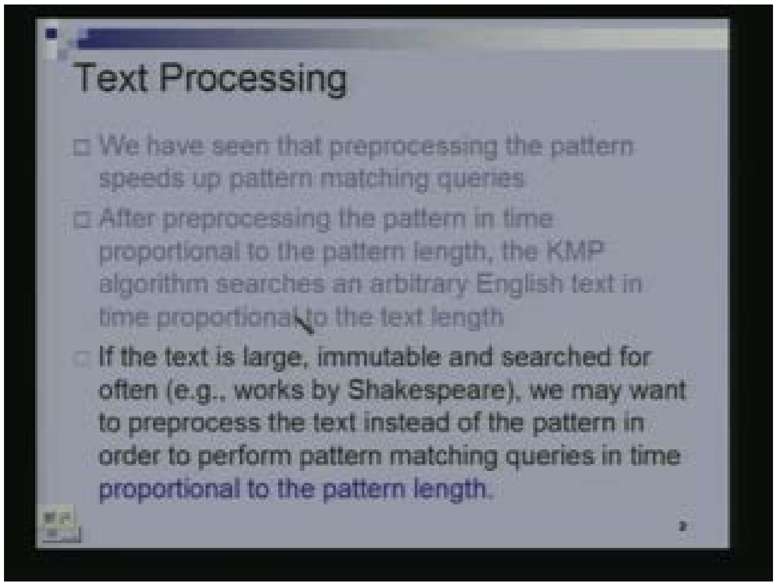
(Refer Slide Time: 02:45)


Text Processing

- We have seen that preprocessing the pattern speeds up pattern matching queries
- After preprocessing the pattern in time proportional to the pattern length, the KMP algorithm searches an arbitrary English text in time proportional to the text length

So this preprocessing the pattern speeded up the Time it took to match the pattern. If we did not compute the failure function h, then we just had this brute force method of matching the pattern which took order m n time. 'm' size of text and 'n' size of the pattern. So after processing the pattern in Time proportional to the length of the pattern, the Knuth-Morris-Pratt Algorithm searches an arbitrary text in time proportional to the length of the text. Now if the text is very large, this is not a very good situation to have.

(Refer Slide Time 02:54)



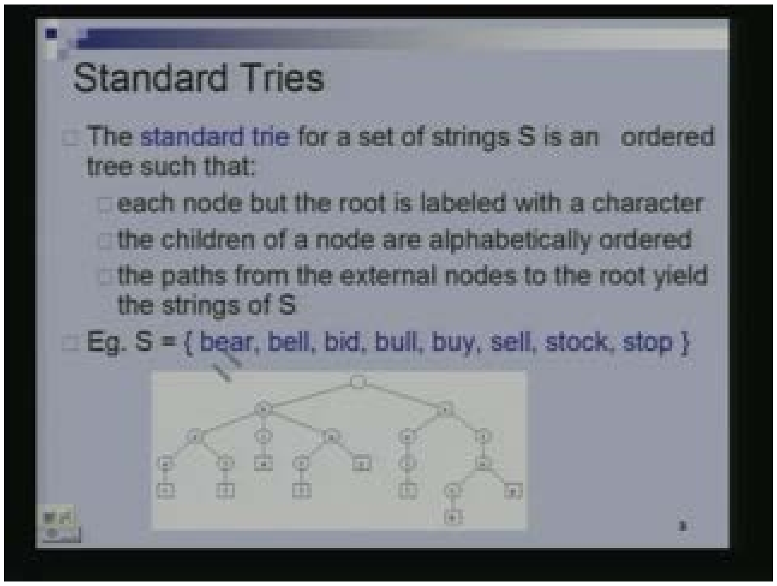
Text Processing

- We have seen that preprocessing the pattern speeds up pattern matching queries
- After preprocessing the pattern in time proportional to the pattern length, the KMP algorithm searches an arbitrary English text in time proportional to the text length
- If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern in order to perform pattern matching queries in time proportional to the pattern length.

So I have a very large piece of text which doesn't change and I am searching for patterns in the text. Every Time I search for a small pattern, if I am going to spend Time proportional to the size of the text, that's a lot of Time. so you have let's say, collected works of Shakespeare in which you want to search for 'veronica' and you are going to spend Time proportional to the size of the text which is very huge. Now what we want to do today is to process the text so that I can search for the pattern in Time proportional to the length of the pattern. see this becomes great because now patterns are typically very small; 7 characters, 15 characters, something like that while your text could be a million characters large. You don't want to spend that much time every time you are searching for a text. But here today we are going to require that we will preprocess that text. We will work on the text initially and we will have created some data structure on that so that when a pattern comes, we can search for that pattern. All occurrences of that pattern we can spot in a very little Time proportional to the length of the pattern, no matter what pattern comes. In the previous KMP algorithm, it was the other way around. You processed the pattern. You did a preprocessing on the pattern so that no matter what text came, you could search on that text. But there you were taking Time proportional to the length of the text which is quite expensive. We will come to the notion of tries today.

What is a trie? I will perhaps show you a picture and I will explain it through this picture. So trie is a data structure to maintain a set of strings. Let's say I have a set 'S' of strings. S = (bear, bell, bid, bull, buy, sell, stock, stop). Now I am going to create a tree here. Now this is not a binary tree. In fact the number of number of children that particular node can have. It can be as large as the size of the alphabet. We are working with the English alphabet. Let's all our strings are lower case characters. So the size of the alphabet is 26. Each node can have up to 26 children. Now the children of the node are ordered alphabetically. What does it mean? Each node is going to have a particular character in it and if I look at all the children of this particular node, then those are going to be ordered alphabetically. So b will proceed c if c were there and c would come after b and so on. I have just 2 b and s. so b comes to the left of s. this had 3 e, I and u. so they come one after the other in this order. So it's an ordered tree.

(Refer Slide Time 06:44)



Standard Tries

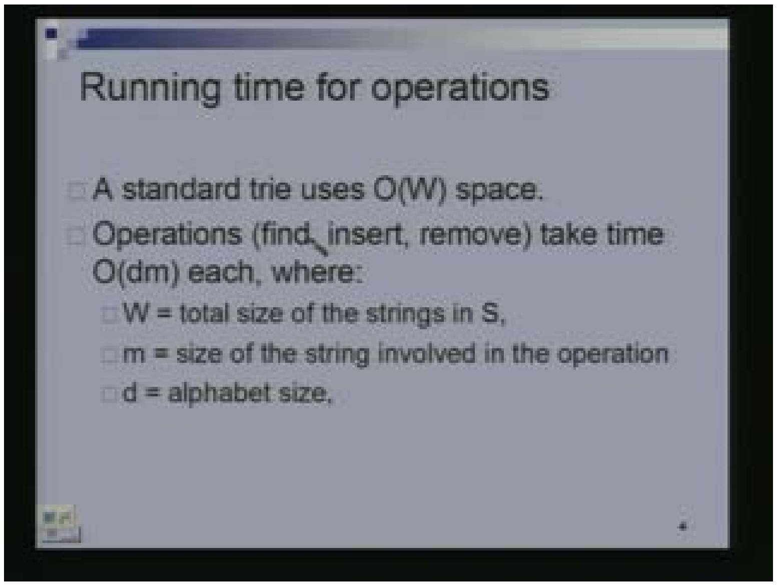
- The standard trie for a set of strings S is an ordered tree such that:
  - each node but the root is labeled with a character
  - the children of a node are alphabetically ordered
  - the paths from the external nodes to the root yield the strings of S
- Eg. S = { bear, bell, bid, bull, buy, sell, stock, stop }

This is read left to. Now how is this organized? Suppose I have to start from here and I have to follow a path in this tree. Suppose I came this way "b e a r". (Refer Slide Time: 07:05). Bear is one of the words here. Suppose I were to take some other path "s e l l". Sell is another one. "s t o c k" – stock. This is another word here. So now you can build this thing. If I give you set of words can you build this trie. It's straight forward. What am I doing? At the very first level I am looking at the first characters of all my words and see what are the various occurrences. If you look at the first character I have just b's and s's. So there will be one node corresponding to b and one corresponding to s and within this b. then this b has only 5 words associated with it. What are the second character of these words. e i u. so that's why e i u is the children of b and so on.

The square here just reflects it's a leaf node corresponds to a word. Suppose I had built such a trie, how much Time does it take to search for a word here? Suppose I give you a word. How much Time does it take? Suppose I said 'bed', how much Time does it take to search for bed here? First I come here. So this has two children. How is this organized? How do you think what kind of data structure would I have to organize this? It's a multi way search tree. It is in some sense but each node can have up to 26 children as a set. So one way of organizing it is each of the nodes has an array of size 26 sitting inside it. The first location of the array points to the node corresponding to a. second to the node corresponding to b. third to the node corresponding to c and so on. If you organize it in an array, you waste space. Each of the nodes has 2 to 3 children here. In that case, instead of an array you could keep a linked list.

Ordered according in the alphabetical order in which case you know you will have two nodes here. the first nodes will be pointing to b and the second node will be pointing to s. you will say that this is b and this is s. now given this, how much Time does it take to search? Suppose I had a link list at each nodes, why would this change the search Time? It is 26 Times the length of the word I am searching for. so I have a link list sitting here (Refer Slide Time: 10:38) and in each of the nodes of the link list, there is a particular character which says that if you are searching for this word and if this is its first character, then follow this pointer. If you are looking for a word beginning with ‘S’, you will have to run through the link list first to get to s and then follow the pointer. If s is not there you can stop away but if s is there in which case you will have to follow the pointer and repeat these things. So how much Time does it take in the worst case? You might have to traverse 26 nodes of the linked list into the length of the word.

(Refers Slide Time 12:32)



Running time for operations

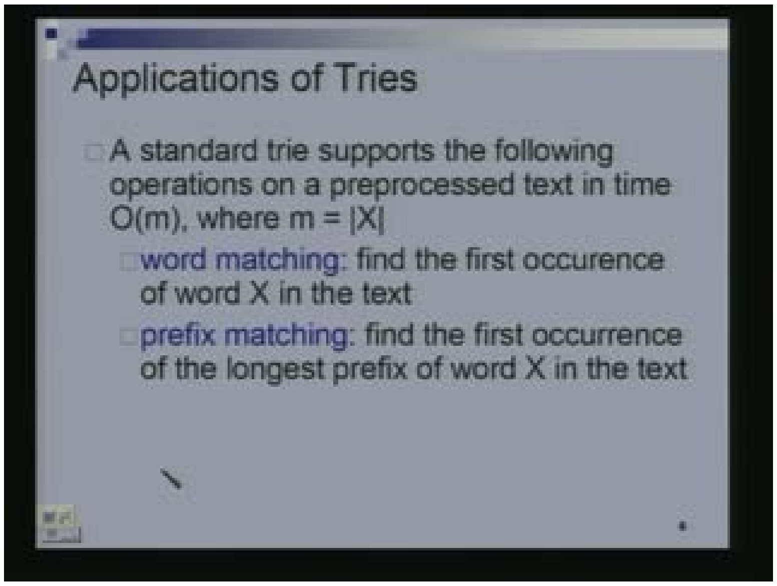
- A standard trie uses O(W) space.
- Operations (find, insert, remove) take time O(dm) each, where:
  - W = total size of the strings in S,
  - m = size of the string involved in the operation
  - d = alphabet size,

Let’s look at the operation of find. How much Time does find take? I am using let’s say linked list of presentation on each other. So 26 is the alphabet size. I am using d to denote it. May be the alphabet was not 26 large. You may have a smaller alphabet or a larger alphabet and m is the size of the string of the word that I am searching for. So that’s the Time for find. can you see that this is also the Time for insert and for delete? When you know you are searching, you keep coming down and then you don’t find it any more. If you don’t find it any more, what you do is insert. You will create that letter, put it in the linked list, make a pointer down and may be you will have to create new nodes. You want me to show how you do this?

Suppose we were trying to insert let me quickly do it. What do what do? I want to insert bed. What will I do? I will search for bed. I will come down here. ‘d’ found. Here I would have a node. I would have seen a ‘b’. So I come down here then I would search for a ‘e’. I come down here. Here I am searching for a ‘d’ in the linked list here. There is no ‘d’. So I create a node. Now it will be a square node because it’s the end of the word and this would have ‘d’ written in it. But if I had “b e d s”, then I would create one circular node and then a square node below. I might have to create such a longer change here. In any case total Time taken would be proportional to the length of the word. We will see later when one of the words is the prefix of the other. That’s what you worried about. (Refer Slide Time 12:32)

So find, insert and delete all take the same Time order dm but one thing is bad with this data structure and that’s the space requirement of the data structure. How much space does it take? 26 Times the number of nodes. How many nodes are in this tree that we have created? Total number of characters in the entire text which is the size of the text. That's the worst case and it can be close to the worst case. I have let's say 10 words. Let's say I have 10 words, each beginning with a different character. The first word begins with the 'a' second with the b third with the c the fourth with the d and so and on and you can make a long chain below this depending upon what the size of the word is. There can be as many as many as total. Not total number of words which is exactly total size of the strings in it. by size I mean put all the characters together and their total size. Let's call that double. So that's the space required which is too large. So we got to do something about this one. Before that, let's see applications. So this actually does our task what we started off with.

(Refer Slide Time 17:07)

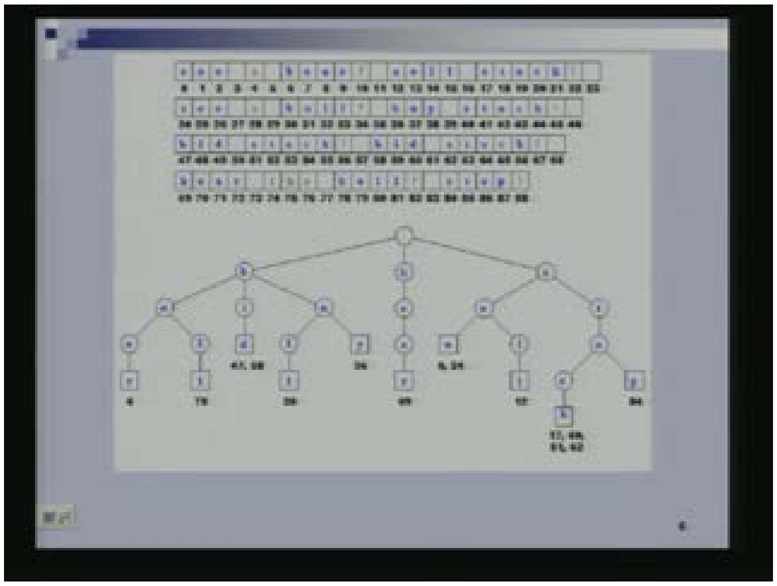


Applications of Tries

- A standard trie supports the following operations on a preprocessed text in time O(m), where m = |X|
  - word matching: find the first occurence of word X in the text
  - prefix matching: find the first occurrence of the longest prefix of word X in the text

So suppose I give you a peace of text and we take all the words in the text and throw them into a trie. I make a trie out of all the words in the text. Now if I have to search for a particular word, I can search for the word in Time proportional to the length of the word. This is what we started off today. So I can do word matching. Find the first occurrence of word 'x' in the text. Why have I said first occurrence? It will be inserted only once. So one occurrence I can detect by doing that. We can also do all occurrences. We come to all of this in a second. Let me show you an example and you will see that we can actually even do all occurrences. So each of these operations of matching is done by tracing the path corresponding to that word in this trie. So let's look at an example. This is a piece of the text.

(Refer Slide Time 18:27)

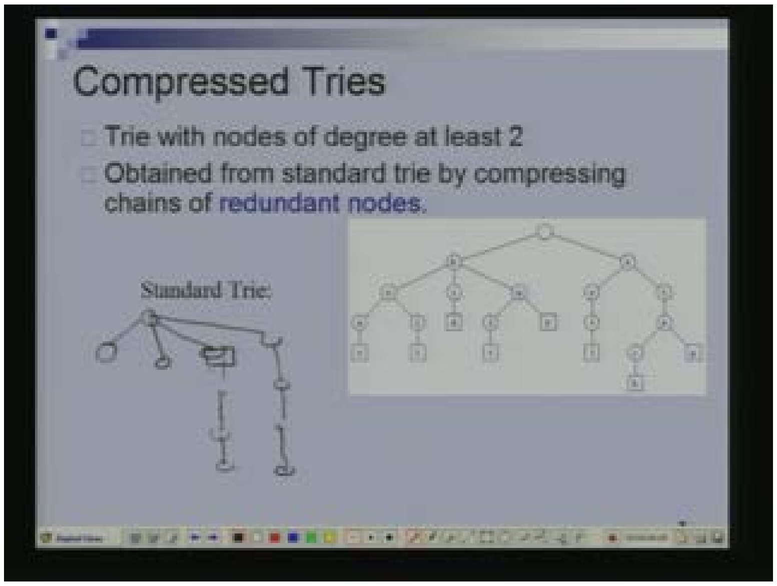


So you have a bunch of words. “See” for instance appears twice. so I am looking at all the distinct words that are there in this piece of text which is “see”, “bear”, “sell stock”, “buy stock”, “bull”, “bid” and “bell”. So these are all the words I threw them into a trie. This is the trie I get and as you can see, this leaf corresponds to “b e a r” (Refer Slide Time: 19:15) and bear occurs at position six that text that is so with this leaf I store 6.

Let’s look at the bid. Bid is occurring at two places. Perhaps starting at 47 and starting at 58. So I will store both 47 and 58 here (Refer Slide Time: 19:42). This is what we call preprocessing the text. I took my initial text and did something, built this trie on it, stored this information in each of these leaves, so that now if you come with queries like where does this particular word appear, I can quickly tell you. How much Time do I need? It’s very little. It’s just proportional to the length of the word and I will be able to tell you all the places where this word is, by looking at this number down here. This doesn’t really solve the problem that we were talking of in the last class which was that I give you piece of text and I give you pattern and find where all the pattern appears in the text. Because my text need not be a collection of words. As I said you know my text could be let’s say, sequence of basis in a gene database. So I have just a long sequence of “A C T G” that kind of thing and I am searching for a particular sequence in there. so here we have a separate notion of words and if we are searching for words, that’s okay. Suppose I was searching for “a r blank s e”, then I cannot search for patterns here. So if I have a pattern like that. So if I don’t know if I think of these blanks also as some special character of my alphabet, then I cannot really search for anything.

So the reason I can search here is because there are well defined boundaries. My pattern has to begin with the boundary and end with the boundary. That’s why I can search. So now first we will address the issue of the large size that this trie has. Let’s try and reduce the size of the trie first. We will do the following.

(Refer Slide Time 24:19)

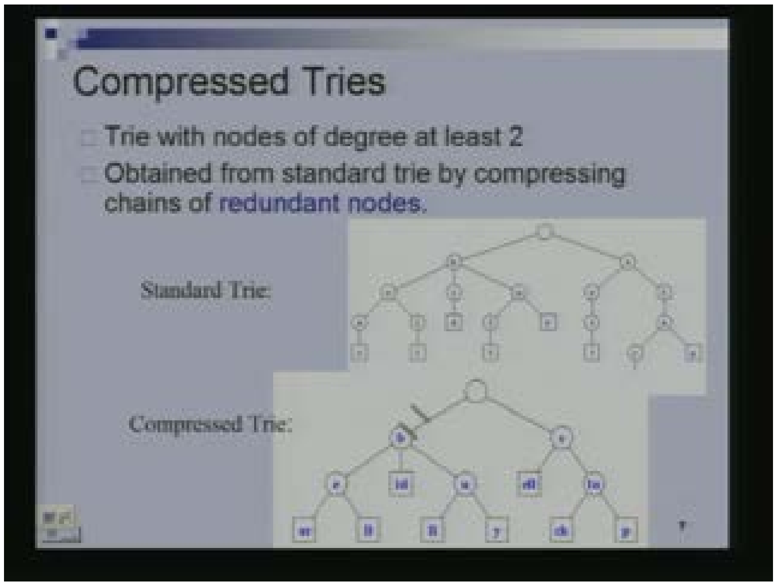


Compressed Tries

- Trie with nodes of degree at least 2
- Obtained from standard trie by compressing chains of redundant nodes.

We are going to look at all nodes of the trie which have degree only one and remove those nodes. By degree one I mean who have only one child and I am going to compress those nodes. So if this is my standard trie, see that there are whole lot of nodes here which have only one child. (Refer Slide Time: 23:30 to 23:40) this node for instance this node this node this node in fact this node as well as this node. This node also has only one child this has only one child. So I am going to compress that. By compress I mean I am going to take the child and collapse it in to the parent and if the resulting node also has only one child, I am again going to take the child and collapse it in to the parent and keep doing this. In my previous example, I had said I have trie in which there were a bunch of words, each of which begins with a different character. So I had created a long chain like this. Now if I compress them in to a single thing, then this entire thing becomes just one node. So I will just show that to you in a second. This is what my compressed trie would look like (Refer Slide Time: 24:39).

(Refer Slide Time 24:36)



Compressed Tries

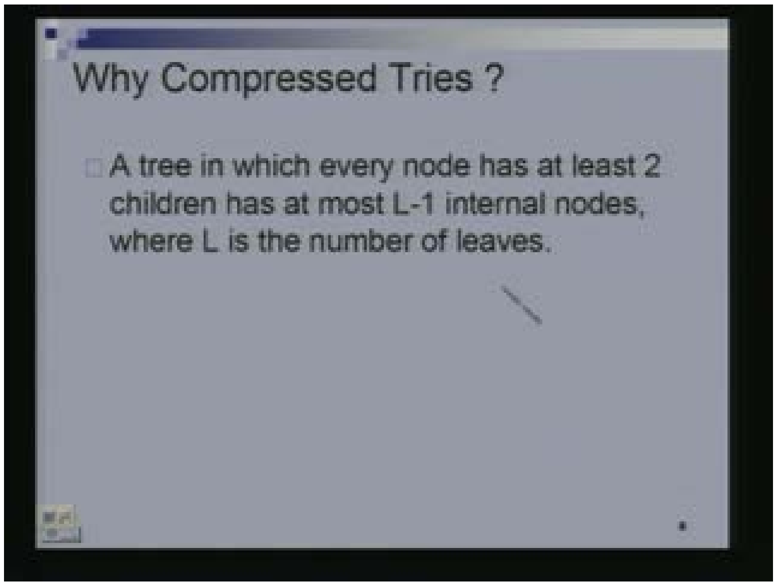
- Trie with nodes of degree at least 2
- Obtained from standard trie by compressing chains of redundant nodes.

Standard Trie:

Compressed Trie:

b and s are the same. ‘i d’ collapse in to one. Now a node doesn’t have one single character but a string as its labeled. This e a r collapses here. l collapses there and so on. As you can see, the compressed trie is smaller than the standard trie. Why would this take less space now? What is the number of nodes in this thing now?

(Refer Slide Time 25:29)



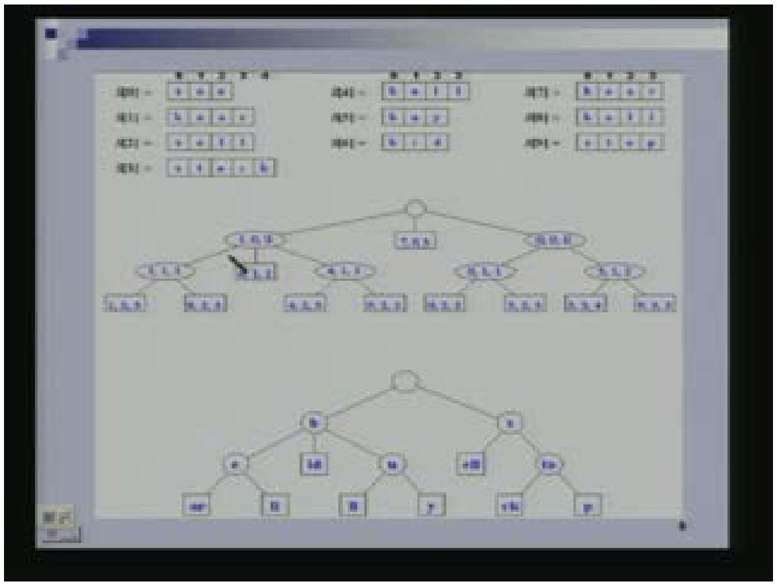
Why Compressed Tries ?

- A tree in which every node has at least 2 children has at most L-1 internal nodes, where L is the number of leaves.

So this is a fact which will prove for yourself. It’s very simple. Suppose I have a tree which has “l” leafs in it and every node of the tree has at least two children. Every internal node of the tree has at least two children. Then the number of internal nodes cannot be more then l -1. a tree in which every node has at least two children, by that I mean except leaf node clearly. Leaf nodes don’t have any children. Every node has at least two children. It has at most l -1 internal nodes where ‘l’ is the number of leaves. If every node has at least two children, then the number of internal nodes is not too much. This is a very simple thing. You can prove it by induction. How are we going to use this? How many leaves are there in my trie? It’s the number of words. This says that the number of internal nodes is going to be (number of words – 1) at most. So the number of nodes in a compressed trie is order of s where s is the number of words. S was the set of string. So s is the number of words. This is the number of nodes in a compressed trie. This doesn’t solve our problem completely. Why because each node now has a longer label inside it. We will also have to store that label. We need space to store that label. From where do we get that space now?

We are going to store labels not as labels but as numbers. Let’s see what I mean by that. Let’s look at this label “i d”. This was the last two characters of the word “b I d”.

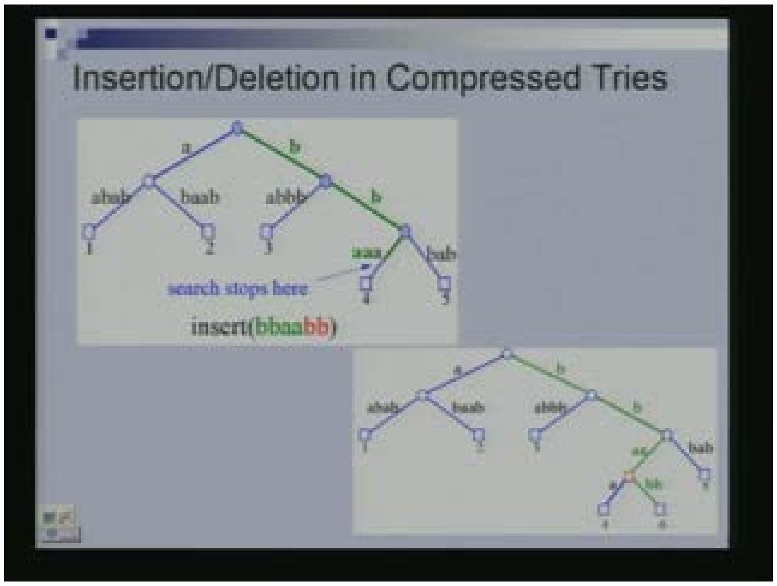
(Refer Slide Time 27:27)



So “bid” is the 6th word in my collection. I have kept all the words in some array in some arrays and id is the last two characters. So in the 6th word, ‘i d’ begins at position one and ends at position 2. So each of these labels no matter how long they are, can be stored as three numbers.

This is because each of these labels will be a substring of one of these words. Do you follow what I mean by sub-string of one of the words? Not necessarily a suffix or the prefix although in this example it looks like a suffix while it is not necessarily. For instance “to” is not a suffix. I am not saying prefix or the suffix. I am saying it’s a sub string. It is a contiguous part of the string. Now what is the space used by the trie? You will have to store these words somewhere. This is your input. This is stored somewhere. So we are just trying to figure out how much additional space we are taking for the data structure there. Then how much additional space are we taking? Now this space I am taking by this data structure is number of nodes. Number of nodes is two times number of words at most into three for three integers each because then there are some pointers. How does searching happen? Let’s look at that. Now how does insertion and deletion happen in a compressed type?

(Refer Slide Time 32:52)

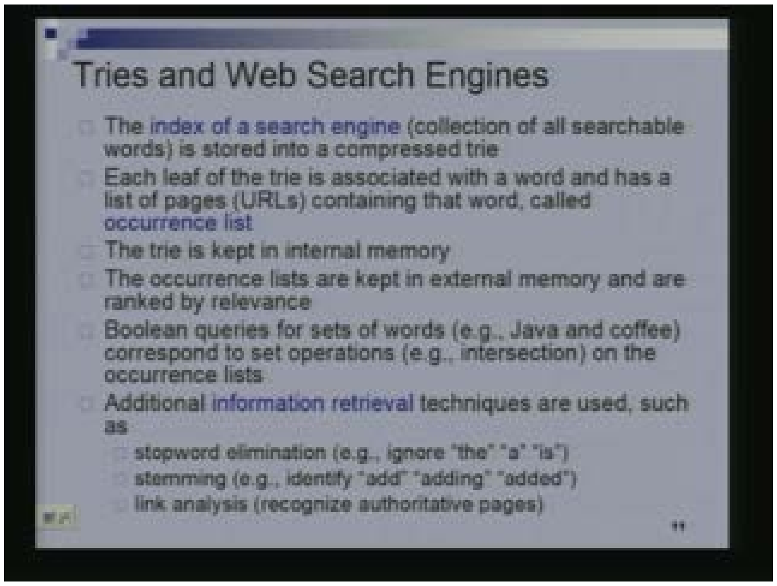


Insertion/Deletion in Compressed Tries

Suppose this is the trie I have. So the labels that I have put at a node, we can also think of that they are the labels on the parent edge of that node. It’s one and the same thing you understand what I mean? There is also a subtle reason why I am doing it this way and we will see why. I am searching for this string “b b a a b b”. I start searching. So conceptually it is the same as saying I see a ‘b’ here and come down. Then the first here is the a. this is a b. so I should go this way and I come down here (Refer Slide Time: 32:04 to 32:29). Now the third character I have is an ‘a’. So I am looking for an ‘a’. So the first character here is an ‘a’. The first character here is a ‘b’.

So I should come down to ‘b’ and then I start matching this with this the label here. We are not doing anything sophisticated. We should now get familiar with this. We are searching for this pattern. We are moving down the tree. (Hindi) degree is the number of children. So we are inserting b b a a bb which means we first search for it. We search for it. We reach the middle of this edge. Till the middle of this edge, we have matched b b a a. (hindi) next character is b (hindi). Red node is the node I’ve inserted (hindi). This is how you will insert. Now how will you delete? We proceed, we find the node and we delete. Now something else has to be done. Suppose I have to delete b b a a b b. I will come here and I will delete this guy. Now I look at the parent. If the parent has only one child left now, collapse the child with the parent and you might have to do this multiple times. So it’s a very simple data structure. I am leaving out the implementation details. You will have to figure a few things out. Tries are very useful they are used in web search because you can imagine why.

(Refer Slide Time 36:59)



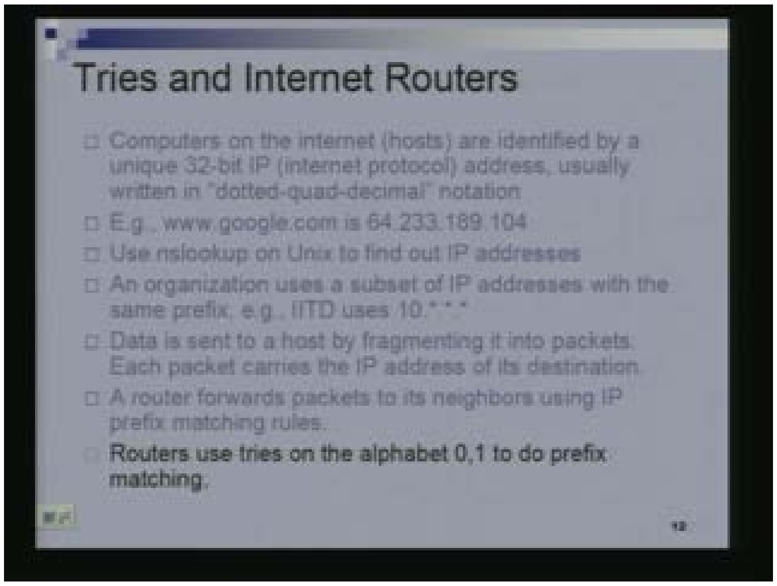
Tries and Web Search Engines

- The index of a search engine (collection of all searchable words) is stored into a compressed trie
- Each leaf of the trie is associated with a word and has a list of pages (URLs) containing that word, called occurrence list
- The trie is kept in internal memory
- The occurrence lists are kept in external memory and are ranked by relevance
- Boolean queries for sets of words (e.g., Java and coffee) correspond to set operations (e.g., intersection) on the occurrence lists
- Additional information retrieval techniques are used, such as
  - stopword elimination (e.g., ignore "the" "a" "is")
  - stemming (e.g., identify "add" "adding" "added")
  - link analysis (recognize authoritative pages)

Imagine you are going to Google. You type a word and it retrieves to you all the web pages that have that word. That part which helps you retrieve all the things is called the index of the search engine. So it is stored as the compressed trie typically. I am not saying that Google doesn't this way. This is what generic search engine do and each leaf of the trie is associated with the word. (hindi). That's called the 'occurrence list'. The trie is kept in an internal memory. The list can be very long. If you type a word like 'computer', you imagine the number of URLs pages that could contain that word. So this occurrence list will be huge. So that's why it's not kept in the main memory. It's kept on disk.

Now suppose you wrote 'computer and music', so now it will search for computer and it will search for music. It will get two occurrence lists. Now it has to take their intersection. So Boolean queries corresponds to set operation of this occurrence list. If it is 'and' it is union and if it is 'or', it is the intersection. Of course there are lots and lots of techniques that go into speed up the thing. You eliminate stop words and many other things. We will not go into that. They are also used in internet routers.

(Refer Slide Time 39:02)



Tries and Internet Routers

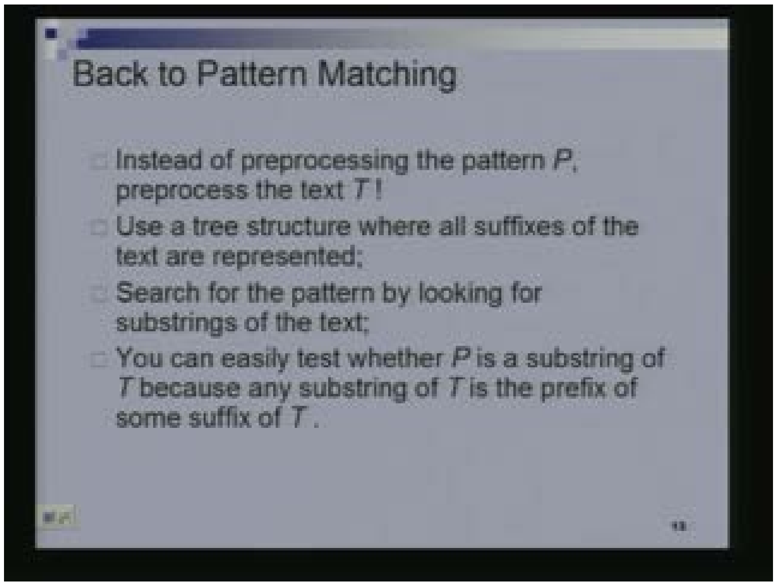
- Computers on the internet (hosts) are identified by a unique 32-bit IP (internet protocol) address, usually written in "dotted-quad-decimal" notation
- E.g., www.google.com is 64.233.189.104
- Use nslookup on Unix to find out IP addresses
- An organization uses a subset of IP addresses with the same prefix, e.g., IITD uses 10.*.*.*
- Data is sent to a host by fragmenting it into packets. Each packet carries the IP address of its destination.
- A router forwards packets to its neighbors using IP prefix matching rules.
- **Routers use tries on the alphabet 0,1 to do prefix matching.**

12

Now you are all perhaps familiar that each computer on internet has an internet or an IP address which is a 32-bit number. So type google.com. You can use nslookup to find out the IP address. So a particular organization just uses the subset which are all related in a certain manner. For instance, all IIT Delhi address will look something like 10. Now how is routing done? In a router when packet comes in, it has an ip address written to it. It doesn't say if this is the IP address, send it here. Router is a bunch of links coming in links and going out. So packets comes on one of the links and the router has to figure out which links to send it out to. There are 2^{32} IP addresses.

It says take the IP address of the packet and find out the longest match. Your table would have the following. Anything that begins with a 10, send it here. Anything that begins with the 10.27, send it here, anything that begins with the 10.27.36, send it here. So now what is the router going to do? It's going to find out the best possible match of these three. It will try to find out the longest match. So if the packet had 10.27.36, then it will go on the 36 route. But if it was 10.27.34, it will take the 10.27 route. If it was 10.28, it will take the 10 route. This is the way routing tables are organized. So they are also tries I used to do this. Tries could be one way of doing it. So we will come back to pattern matching now.

(Refer Slide Time 40:01)



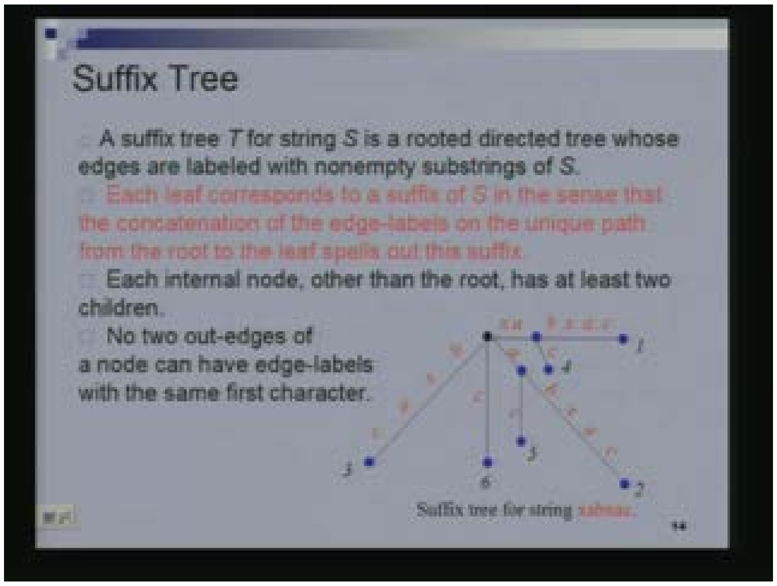
Back to Pattern Matching

- Instead of preprocessing the pattern *P*, preprocess the text *T* !
- Use a tree structure where all suffixes of the text are represented;
- Search for the pattern by looking for substrings of the text;
- You can easily test whether *P* is a substring of *T* because any substring of *T* is the prefix of some suffix of *T* .

13

We saw compressed tries are doing the job reasonably well provided there was the notion of words or delimiter and our pattern started and ended at the delimiter. But suppose you are as I gave you an example if you are searching in a biological data base there is no notion for the delimiter there. What do you do then? So this is something we said before. Instead of preprocessing the pattern, we are going to be preprocessing the text. Now what we are going to do is the notion of what’s called the suffix tree. We will take all suffixes of the text and organize them in to a tree and you will see what I am trying to say in a second. Let’s see piece of text x a b x a c (Refer Slide Time: 41:09).

(Refer Slide Time: 41:09)



Suffix Tree

- A suffix tree *T* for string *S* is a rooted directed tree whose edges are labeled with nonempty substrings of *S*.
- Each leaf corresponds to a suffix of *S* in the sense that the concatenation of the edge-labels on the unique path from the root to the leaf spells out this suffix.
- Each internal node, other than the root, has at least two children.
- No two out-edges of a node can have edge-labels with the same first character.

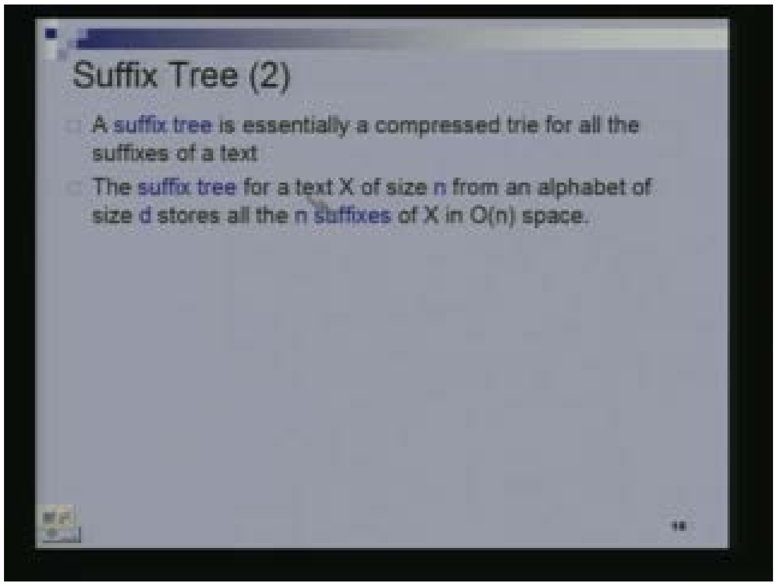
Suffix tree for string xabxac.

14

How many suffixes does it have? There are 6 suffixes. I am not saying proper suffix. I am going to take them as my words. x a b x a c is a one word. a b x c is another. b x c is another. x a c is the 4th. a c is the 5th and c is the 6th. There are 6 words and I am going to create a trie of these words, in particular a compressed trie and this is what the structure is.

So let's see why it's a trie. Here if it were a 'b', I would go this way. If it were a 'c', I would go this way. If it were an 'a', I would go this way. If it were an 'x', I would go this way (Refer Slide Time: 42:06). (hindi) numbers are the starting position of that suffix. So this corresponds to x a b x a c. what is the starting position? It's one. This corresponds to x a c. Its starting position is 4. This corresponds to a b x a c. The starting position is 2. The starting position for a c is 5 and so on and so 4th. So put all our suffixes in a trie. So it's essentially a compressed trie for all the suffixes of the text.

(Refer Slide Time 44:09)

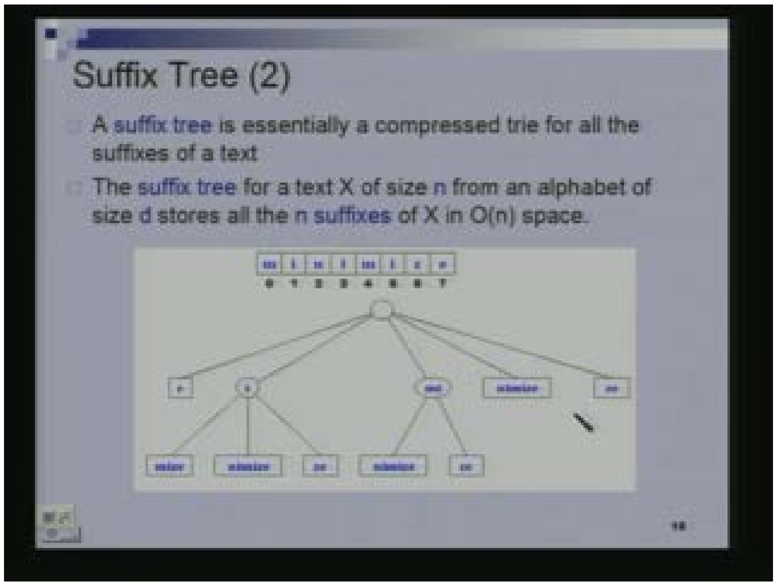


Suffix Tree (2)

- A suffix tree is essentially a compressed trie for all the suffixes of a text
- The suffix tree for a text X of size n from an alphabet of size d stores all the n suffixes of X in O(n) space.

So it seems it would be huge but why should it be huge? How many suffixes are there? There are as many as length of text. So we will have that many words. Recall that the size of the trie is just the number of words order number of words. So its order length of the text. (hindi) so this size of the trie is not too much. So suffix tree for us text x of size n from an alphabet of size d stores all the n suffixes of x in order n. d is typically small. So it doesn't require too much space. (hindi)We will come to why we are doing suffixes. Can someone think of why suffixes? So I was searching for a b. what will happen if I am searching for a b? Suppose I start searching for a b. I will come at 'a' here and then I will come at 'b' here and I will stop in the middle but can I say something now? I did not find. That's what you will be tempted to say. If the pattern appears in the text then there is some suffix whose prefix is that pattern. That means that there is some word in the collection of words that I have thrown in whose prefix is that pattern which means that when I am searching for the pattern, that initial part of the word will match up and I will be able to do something with that. Many of you can see what I will be able to do. I will just look at the leaves of that sub tree and identify. We will come to all of that. That's the remaining of this lecture. So let's say I had this word "minimized".

(Refer Slide Time 46:49)

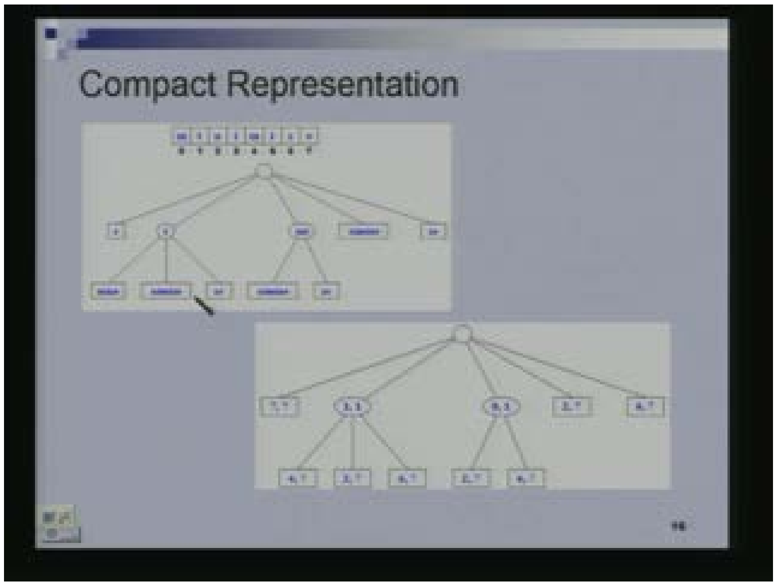


Suffix Tree (2)

- A suffix tree is essentially a compressed trie for all the suffixes of a text
- The suffix tree for a text X of size n from an alphabet of size d stores all the n suffixes of X in O(n) space.

I don’t make the suffix tree for each word in the text. I make a suffix tree for the entire text. So if this is my entire text I make a suffix tree for it. There would be 8 suffixes. This would be the corresponding suffix tree I would get. Once again I have collapsed my nodes. You can all make this suffix tree and now we want to do a compact representation once again.

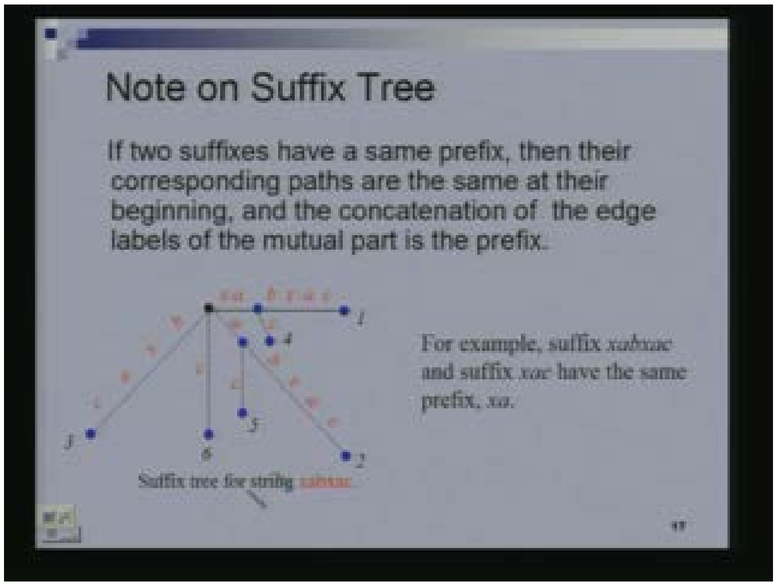
(Refer Slide Time 47:58)



Compact Representation

How much space do I require? So instead of storing labels, there are big labels here. We don’t want to store labels. So once again we can store by numbers. Once again each one of them is a sub string. So I just need to know what the start and the end position of the sub string is. I don’t even need three integers now. I just need two because they are all part of one single text. So this is for instance what would happen. m I n I m i z e - m i n i m i z e starts at position 2 and ends at position 7. So I can store it very efficiently. Now this is the key thing which we are using in pattern matching.

(Refer Slide Time 48:22)



Note on Suffix Tree

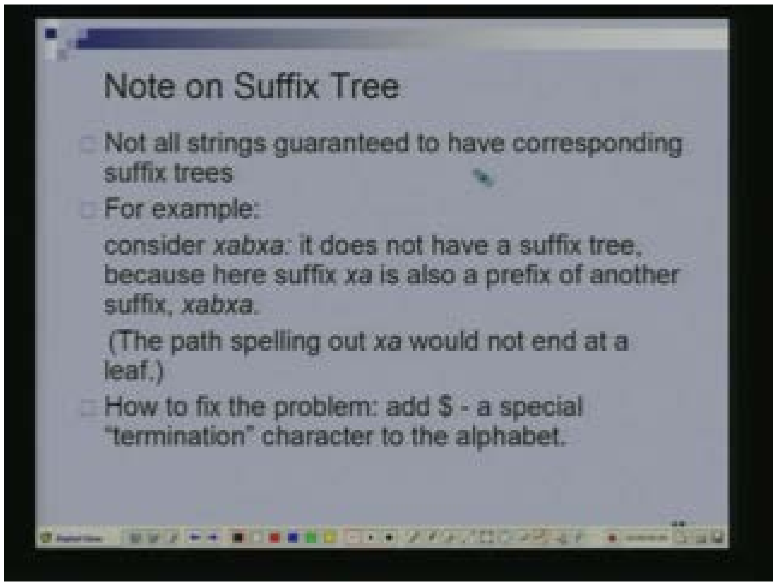
If two suffixes have a same prefix, then their corresponding paths are the same at their beginning, and the concatenation of the edge labels of the mutual part is the prefix.

For example, suffix *xabxac* and suffix *xac* have the same prefix, *xa*.

Suffix tree for string xabxac

So if I have two suffixes “x a b” “ x a c” and they have the same prefix “x a”, their corresponding paths are the same at the beginning and it’s just the concatenation of the edge labels of the mutual parts. So x a b x a c x a b x a c, its common part is “x a” and it comes here (48:53). This is going to be crucial in a short while because now if I was searching for x a, I would end up here (Refer Slide Time: 49:12). So I have to actually report all occurrences and this will help me do that. So what do I have to do report all occurrences basically I have to look at its children. Look at the leaves in the sub tree and that will give me the position. We will come to all of that. This was the problem that some one had pointed out very briefly in the beginning. If one word in my trie is contained in another word, what happens? Suppose my text is x a b x a, now what is going to happen? I have one suffix which is x a and another suffix which is x a b x a. this suffix x a is a prefix of the other suffix. So in my trie, what is going to happen? You know one of the words is going to end up at some internal node. You don’t see a big problem with this? Let me quickly show you what I am trying to say.

(Refer Slide Time 50:54)

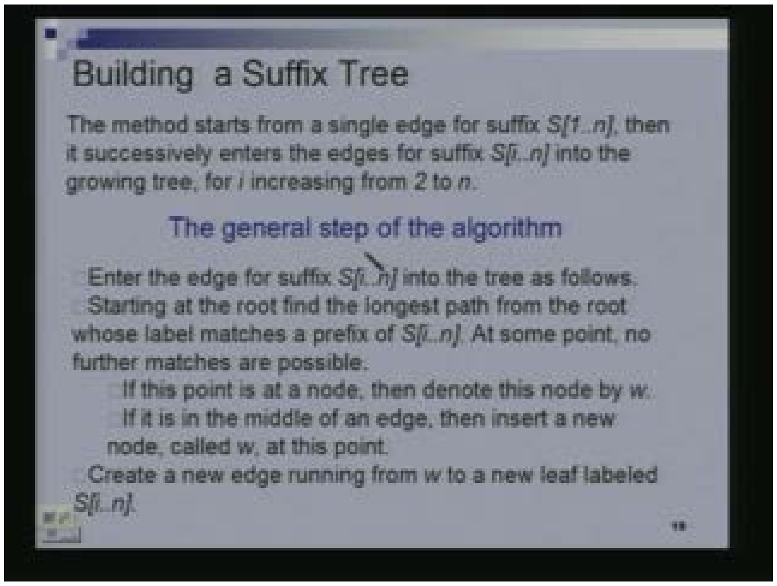


Note on Suffix Tree

- Not all strings guaranteed to have corresponding suffix trees
- For example:
  consider *xabxa*: it does not have a suffix tree, because here suffix *xa* is also a prefix of another suffix, *xabxa*.
  (The path spelling out *xa* would not end at a leaf.)
- How to fix the problem: add $ - a special "termination" character to the alphabet.

Let's call it "x a" and "b x a". There are 2 suffixes "x a" and " x a b x a". We are ignoring the other suffixes. (hindi) what is special about dollar? There is nothing special about dollar. It just is a character which is not part of an original alphabet. (hindi) so now how does one build a suffix tree?

(Refer Slide Time 53:10)



Building a Suffix Tree

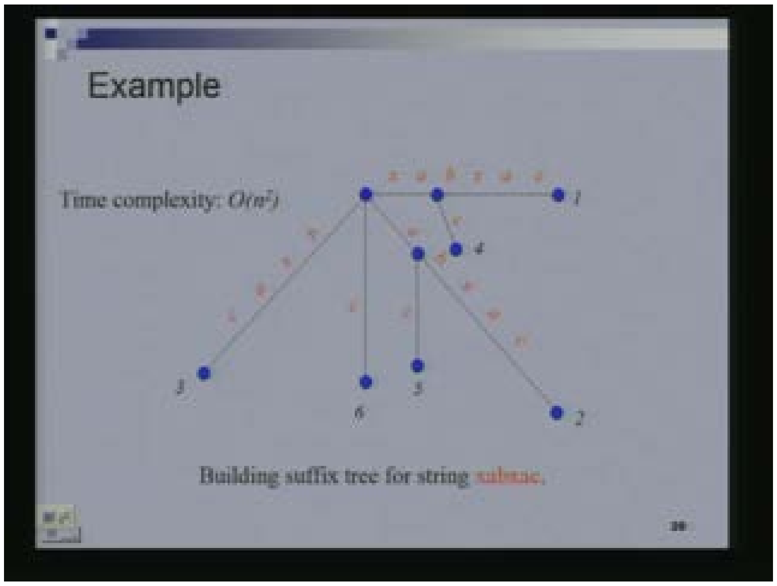
The method starts from a single edge for suffix *S[1..n]*, then it successively enters the edges for suffix *S[i..n]* into the growing tree, for *i* increasing from 2 to *n*.

The general step of the algorithm

- Enter the edge for suffix *S[i..n]* into the tree as follows.
- Starting at the root find the longest path from the root whose label matches a prefix of *S[i..n]*. At some point, no further matches are possible.
  - If this point is at a node, then denote this node by *w*.
  - If it is in the middle of an edge, then insert a new node, called *w*, at this point.
- Create a new edge running from *w* to a new leaf labeled *S[i..n]*.

We start with one initial - one suffix. Let's say this is the entire text. So that basically is one edge and then we keep breaking this edge. So we will search for the next suffix. (hindi). starting at the root, find the longest path from the root whose label matches a prefix of si through n. at some point if no matches are possible and if this point is at the node, then we denote this by a 'w'. If it is in the middle of an edge, we insert a new node and then call this node ‘w’ and we create an edge running from the root to the suffix that we create. So we can take an example quickly. One suffix is ‘ x a b x a c’. (hindi)

(Refer Slide Time 55:16)



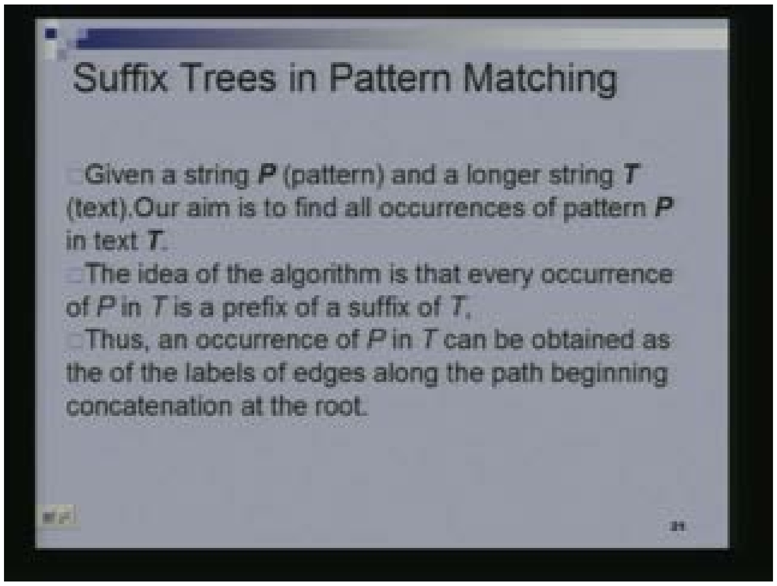
Example

Time complexity: $O(n^2)$

Building suffix tree for string xabxac.

So this will take time proportional to the length of the text. If the length of my text was n then it takes time order summation n^2. It is a bit more but we will see what we can do about this one.

(Refer Slide Time 55:29)

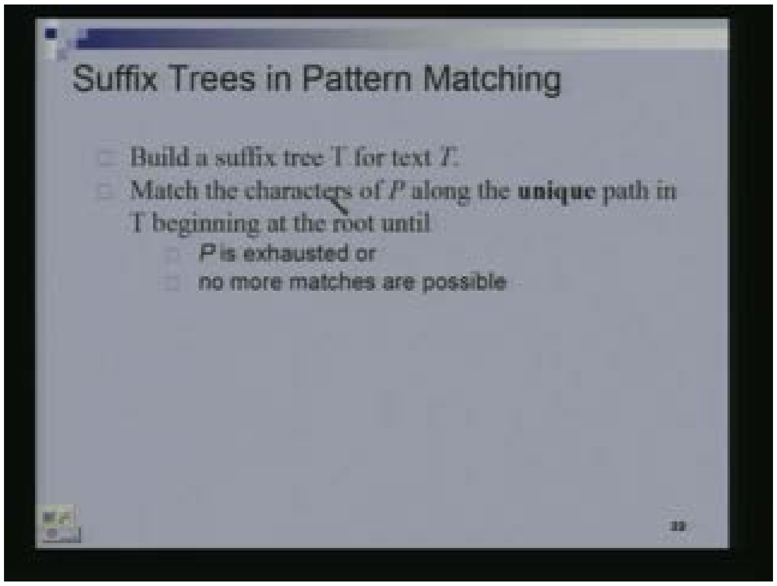


Suffix Trees in Pattern Matching

- Given a string ***P*** (pattern) and a longer string ***T*** (text).Our aim is to find all occurrences of pattern ***P*** in text ***T***.
- The idea of the algorithm is that every occurrence of *P* in *T* is a prefix of a suffix of *T*.
- Thus, an occurrence of *P* in *T* can be obtained as the of the labels of edges along the path beginning concatenation at the root.

This is the key idea that we are exploiting in pattern matching. So given a pattern ‘p’ and has text t, our aim is to find all occurrences of pattern p in the text. So the idea of algorithm is that every occurrences of p in t is a prefix of a suffix of t. (hindi) thus an occurrence of p can be obtained as concatenation of the labels of edges of the path beginning at the root. So how do we do pattern matching? We build a suffix tree for the text, match the characters of the pattern along the path beginning at the root until the pattern is exhausted.

(Refer Slide Time 56:25)



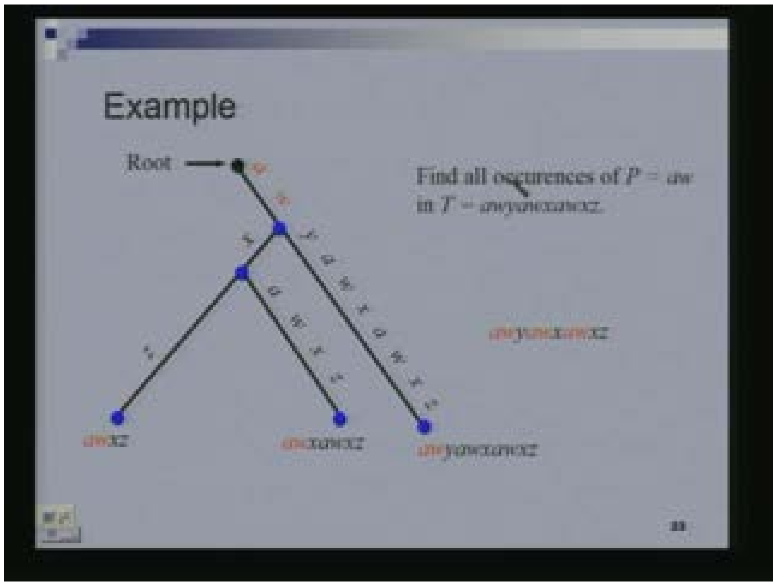
Suffix Trees in Pattern Matching

- Build a suffix tree T for text *T*.
- Match the characters of *P* along the **unique** path in T beginning at the root until
  - *P* is exhausted or
  - no more matches are possible

33

If the pattern is exhausted completely, that means that we have found a match. If no more matches are possible, then that means that pattern does not exist. So 2 in this case, p does not appear anywhere in the text. In case 1, p is the prefix of a suffix of a certain suffix which is obtained by extending the path and till we reach a leaf. Each extension gives a suffix. All the leaves we can reach from there will tell us the occurrences of the pattern. Each extension provides af occurrence of the p in t. what are the extensions? They are basically all the leaves below that. (hindi) let’s quickly see an example.

(Refer Slide Time 57:44)

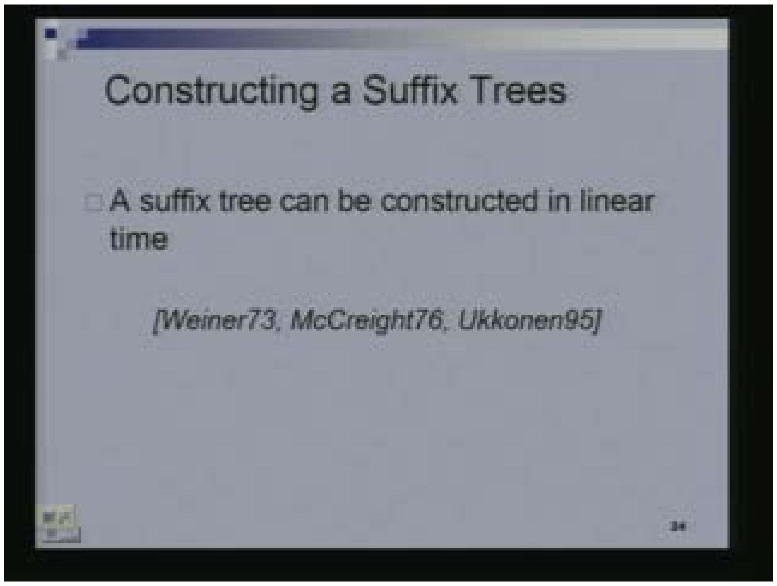


Example

Find all occurences of P = aw in T = awyawxawxz.

This corresponds to this suffix. This corresponds to this suffix this corresponds to this suffix (Refer Slide Time: 58:08). The number that you write here is the starting position of this pattern of this suffix. So we write a 7 here. We saw only an order n^2 algorithm for constructing the suffix tree.

(Refer Slide Time 58:41)



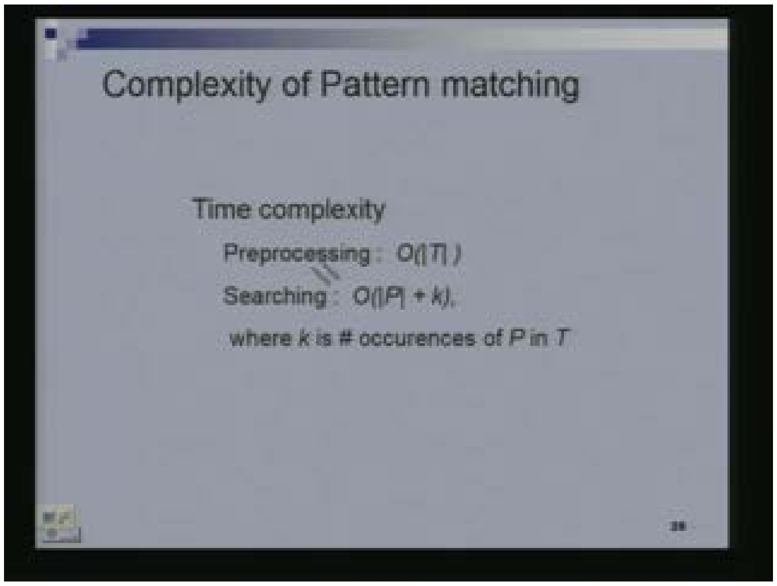
Constructing a Suffix Trees

- A suffix tree can be constructed in linear time

[Weiner73, McCreight76, Ukkonen95]

It can actually be done in order n time but that's a fairly complicated algorithm. We will not be doing it in this class. So that gives us the total complexity of pattern matching.

(Refer Slide Time 58:56)



Complexity of Pattern matching

Time complexity

Preprocessing : $O(|T|)$

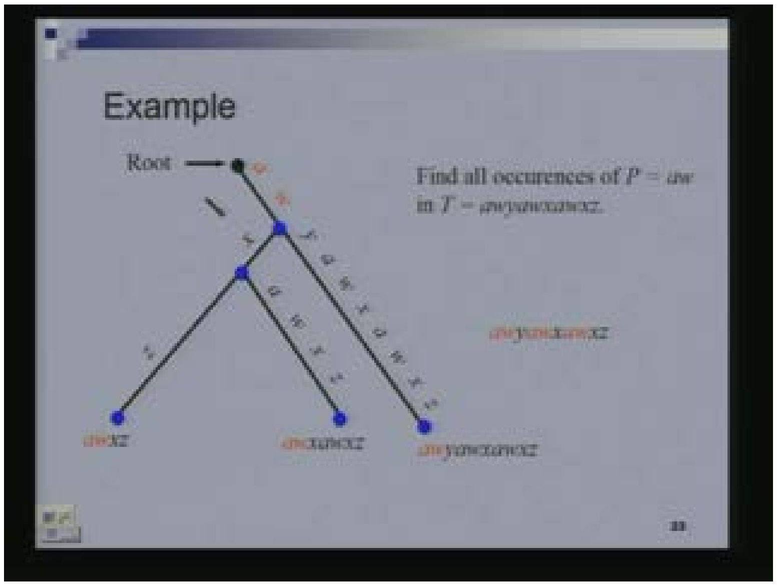
Searching : $O(|P| + k)$,

where k is # occurences of P in T

So preprocessing which means building the suffix tree. We said it can be done in order n times proportional to the size of the text although we saw only an n^2 algorithm today. And for searching I've said size of the pattern plus 'k'- number of occurrences of the pattern in this.

This is completely essential. If a pattern is only 3 characters long but occurs one thousand times in the text and you have to say all the times it appear then clearly you are going to take time proportional to 1000. So this is clearly a requirement and why is this coming up? This is because we have to report all the leaves of this node.

(Refer Slide Time 59:41)

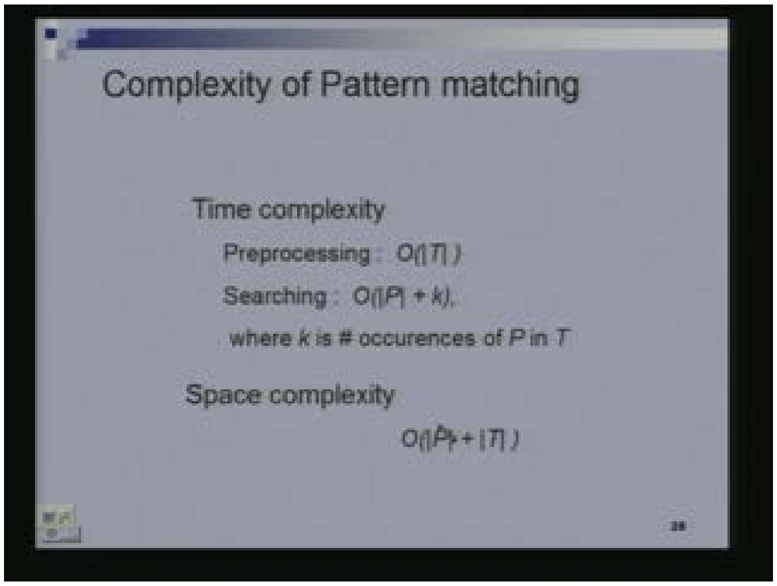


Example

Find all occurences of $P = aw$ in $T = awyawxawxz$.

If there are k leaves we have to go and report all the k leaves. Have many internal nodes are there in this sub-tree? There are (k – 1) nodes. What is the size of the entire sub-tree? It’s of order k.

(Refer Slide Time 1:00:42)



Complexity of Pattern matching

Time complexity

Preprocessing : $O(|T|)$

Searching : $O(|P| + k)$,

where k is # occurences of P in T

Space complexity

$O(|P| + |T|)$

28

That gives us the total complexity of pattern matching. Let me go the last side. The total space we require is the proportional to the size of pattern to store the pattern. So with this I end today’s lectures. So we looked at a faster, faster in the sense now we decide to preprocess the text and to search for the pattern we just need time proportional to the length of the pattern.