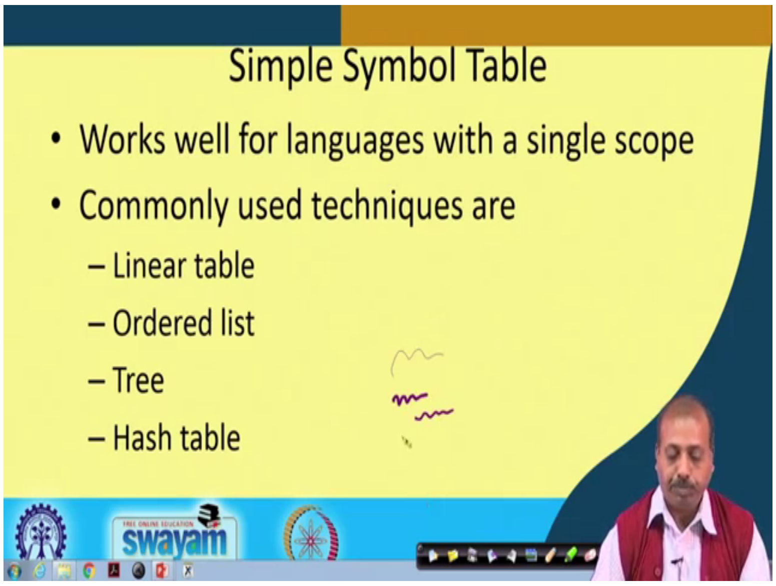
Compiler Design
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture - 44
Symbol Table (Contd.)

In our last class we have been discussing on simple Symbol Table and we have seen that this simple symbol tables they work well for languages with a single scope.

(Refer Slide Time: 00:25)



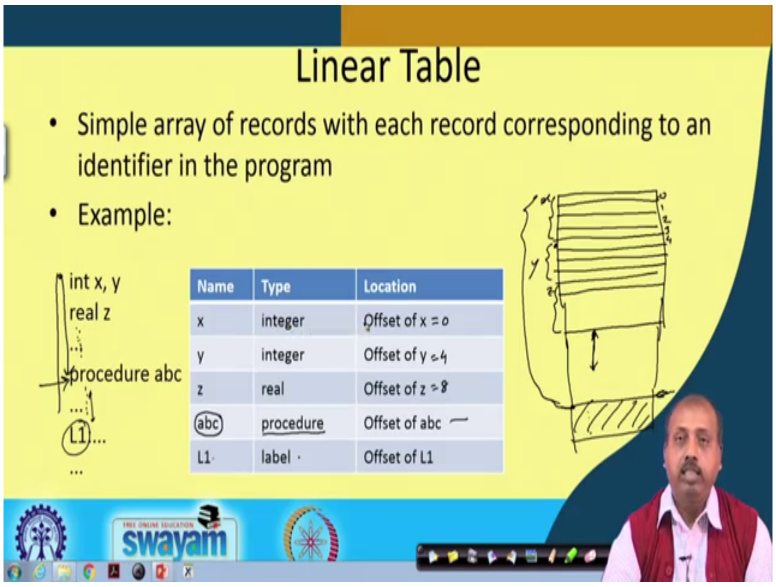
Simple Symbol Table

- Works well for languages with a single scope
- Commonly used techniques are
  - Linear table
  - Ordered list
  - Tree
  - Hash table

And single scope means all variables and identifiers that we have in the program they are visible to the entire program. So, there is nothing like there are like where they are, like if we have got local variables in the visible only within the function or procedure. So, that type of considerations are not there. So, all the variables and identifiers so, they are visible to the entire program. So, for this type of programming languages, it is we can have a very simple structure and these are the alternative; like we can have a linear table, we can have ordered list tree or hash table. These are the commonly found of symbol table organisations.

So, one thing I would like to tell you that it is not mandatory that you should follow only one of these 4 alternatives for symbol table organisation. So, depending on the programming language and the type of information, that you need to store so, you may prefer some other organisation as well. So, so or some complex data structures can be thought about and then the symbol table maybe organised like that. So, these are actually the commonly used options that we are exploring.

(Refer Slide Time: 01:37)



Linear Table

- Simple array of records with each record corresponding to an identifier in the program
- Example:

```
int x, y
real z
...
procedure abc
...
L1...
...
```

Name	Type	Location
x	integer	Offset of x = 0
y	integer	Offset of y = 4
z	real	Offset of z = 8
abc	procedure	Offset of abc
L1	label	Offset of L1

So, so next day if we look into this different table organisations the first one is the linear table. So, it is an simple array of records with each record corresponding to an identifier in the program. So, in the pro so, like here is an example, we have got a program where we have got this variables integer variables x and y, a real variable z, the procedure a b c and we have got a label L1. So, so for as per as the symbol table contents are concerned so, we are storing the name of the symbol or name of the identifier, its type and the location; location is the offset from the beginning of the program.

So, what do you mean offset? It is like this that suppose we have got a program say, this program it is now if you if you say that at the if you consider the corresponding object file the object code that is produced. So, in that object code, I should have space for this x and y. So, x is the first location that we have in the program. So, you can say that the offset of x is equal to 0 fine, then the after that we have got integer y.

So, assuming that these integers are 4 byte long so, this 4 bytes will be reserve for x. So, this 4 bytes are reserved for x and after that the next 4 bytes it will be storing for e it will have the y values. So, these 4 values so, they are corresponding to y. So, y is offset is this one so, that is equal to 4. So, if you look in terms of byte offset so, z is 0 1 2 3 and then y starts at 4. So, that way the offset of y is 4 and after that offset of z. So, offset of z is 4 plus size of y the size of y is 4. So, z will start from this point and it will be spanning over next 6 bytes.

So, if you assume that the real numbers are required in 6 byte so, it will have 6 bytes. So, this offset is equal to 4 this offset equal to 8 and then we have got some statements here. So, it is taken as if we do not have any more definitions here. So, they are some pro some statements are coming. So, for so, we can say that from this point onward the actual code for the program we will start. So, from this point onward the code we will start. So, accordingly we can calculate the offsets like, as we are car converting this programming language statements into machine language.

So, accordingly it will take few lines of the few bytes of the machine language program to store the corresponding source language statements. So, that way the offsets will be proceed processing. Now, when we come to this procedure a b c. So, with so, this a b c name is installed into the symbol table, its type is set as procedure and the offset is from the beginning of the program after how much time, after how much byte this a b c is coming. So, if I assume that this so, this translation goes up to this point and then from this point onwards I am storing the procedure the code for procedure a b c. So, this is the code for procedure a b c.

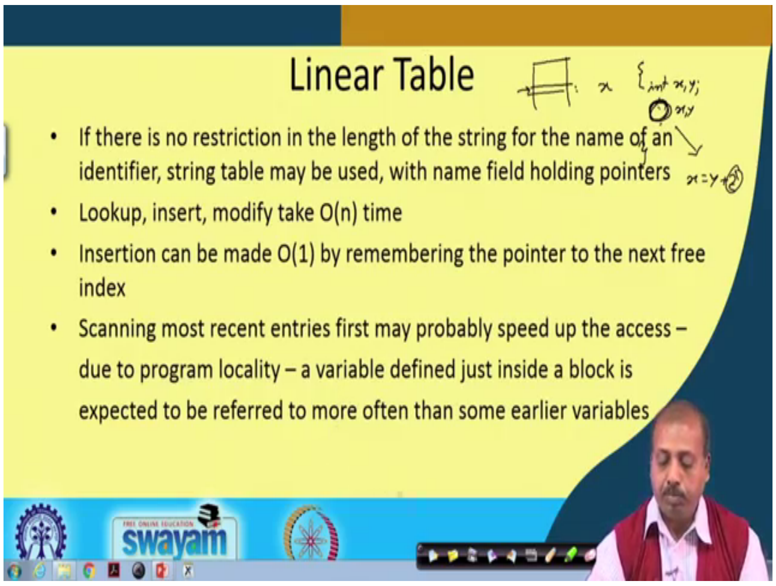
So, the, what is the offset that is starting from the beginning of the program; if I assume that this location is 0 so, what is the corresponding byte index. So, that is the offset of a b c. So, the concept comes like this because ultimately this program will be loaded into memory and when it is loaded so, a similar such image we will get created onto the memory. So, they will be copied byte by byte from the secondary storage to the memory main memory.

So, we they are the offsets will be maintaining their relative positions, variables that I diff defining the program the program statements that we are defining. So, they will be at relative offsets as it is there in this translated program. So, this procedure a b c so, we will have the offset of a b c is told a here, then L1.

So, L so, after that again some statements are coming so, they are coming here and at some point of time this level L1 occurs. Though in a languages like C we are not having this levels much, but in some programming languages a we have that. So, in C also this labels are allowed, but it is not encouraged, but anyway so, since it is there as an identifier in many programming languages. So, it is also kept in the symbol table that L1 is of type level and as similarly we can compute the offset of L1 from the beginning of the program.

So, that way all the offsets are calculated and those they those offset values are stored in the location part of the symbol table. So, this is the linear table organisation. So, this is simply an area of records and each record has got 3 fields in it: name of the symbol, the type of the symbol and the offset. So, so that way we can have this structure.

(Refer Slide Time: 06:43)

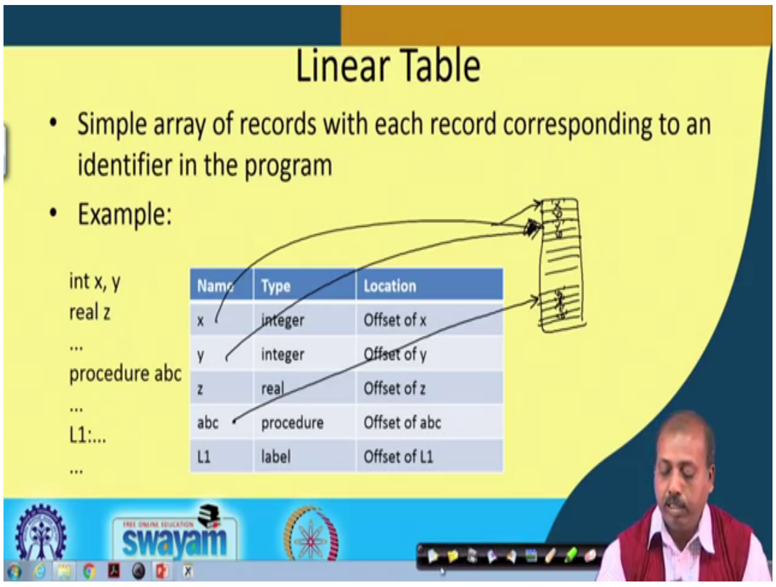


Linear Table

- If there is no restriction in the length of the string for the name of an identifier, string table may be used, with name field holding pointers
- Lookup, insert, modify take O(n) time
- Insertion can be made O(1) by remembering the pointer to the next free index
- Scanning most recent entries first may probably speed up the access – due to program locality – a variable defined just inside a block is expected to be referred to more often than some earlier variables

Now apart from this linear table so, other organisations are also possible, but linear table if there is no restriction on the length of the string for the name of an identifier string table may be used. So, what I want to so, mean is that this name part so, name part ultimately; so, we are storing the name of the symbol. Now, if there is if the programming language we will tell whether this identifiers should have a finite length or not. So, if it is not finite so, in that case it is better that this name field instead of being a character array so, it can be a pointer and it points to the string table.

(Refer Slide Time: 07:17)



Linear Table

- Simple array of records with each record corresponding to an identifier in the program
- Example:

```
int x, y
real z
...
procedure abc
...
L1:...
...
```

Name	Type	Location
x	integer	Offset of x
y	integer	Offset of y
z	real	Offset of z
abc	procedure	Offset of abc
L1	label	Offset of L1

So, as I as I was discussing in the last class so, this string table; so, it will have the all the string variables the all the strings that are occurring in the program. So, here this the so, suppose for this x. So, this it will be store it will be storing this x followed by the null character and this pointer will be pointing to the this particular entry.

Similarly after that we have got the symbol y y followed by null character. So, this will be a pointer to this y. So, we can have the situation like this; similarly at some point we have the name level procedure name a b c. So, it is stored like this a b c and then the null character and then this will be a pointer to this entry in the string table.

So, this way using string table so, we can have we can store the variable names which are which whose length is not restricted. So, if the programming language says the length of the symbols not restricted then we can do that. However, mean in most of the cases we have got this length the restricted. In that case so, we can have this simple character array as the name part. So, in that case when it is when string table is used in the name part will be holding the pointers. The operations that we do on symbol table as we say previously that lookup insert modify and delete.

So, if you want to search in the table then it will take order n time because, assuming that there are a number of entries in the table. So, it will use a simple linear search so, it will take order n time. Similarly, if you want to insert then you need to go to the end of the table. So, assuming that you go to the end so, it so, that can be taken like that. So, then modification; modification also it needs to search. So, because so, that will take order n time, so, insertion can be made order 1 by remembering the pointer to the next free index. So, if you remember what is the next free index then it is order 1 because, we do not have to search for a free slot in the table ok.

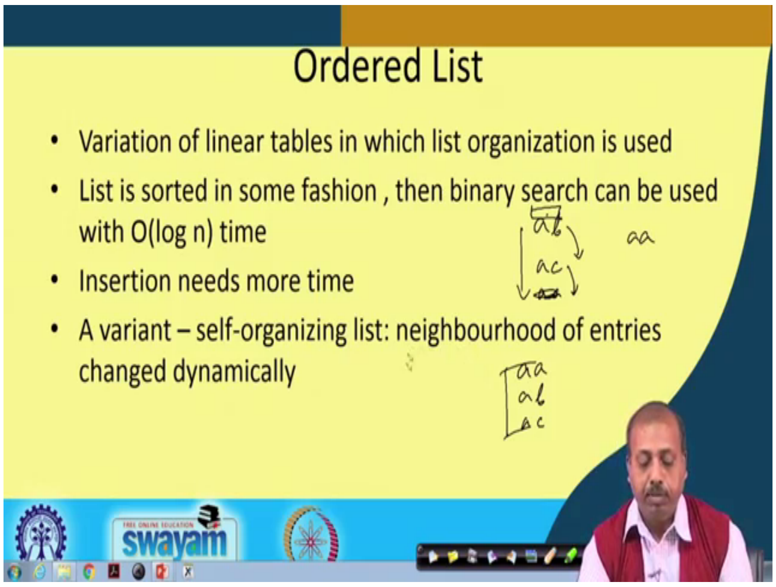
So, if you are remembering the say one point one index like how many entries in the table or fool in that case the next insertion is at the next free slot. So, that way it can be done in constant time. Scanning most recent entries first may pa probably speed up the access due to program locality. So, what we mean is that what we mean is that; so, the so, program locality means that at any point of time if I am if I having a program. So, suppose I am at this particular line in the program. So, in this line suppose it is accessing the variable x, then it is very much likely that in near future, it will be accessing the variables which are close to x.

Close to x in the sense that they may be defined within the same block, like if this is a C program and I have got a block like this where this x and y variables have been defined. Then in this part of the code it is very much likely that it will be referring to the variables x and y. So, if I can somehow make this x and y easily accessible then my compilation speed will be better because, I do not have to search the symbol to the entire symbol table need not be search. So, or searching the first few entries itself so, we can get the x and y variables. So, this is due to program locality because due to the locality so, it is very much likely that; so, the variables which are defined in the large in the most nested block.

So, they will be referred to more often compared to some other variable; like there may be a statement like say x equal to y plus z where z is a non-local variable. But, this is the possible the chances are less because it will be accessing the local variables more. So, scanning the most recent entries first it will be probably speed up the access process; as a result the compilation will be easier will be faster.

And, a variable defined just inside a block is expected to be referred to more often than some variable some then some earlier variable. So, this is what I was talking about that in that x equal to y plus z; so, in that access you see two local variables are accessed and one global variable is access one non-local variable is accessed. So, as a result most of the time it is accessing local variables. So, we should do something so, that this most recent entries they can be searched very easily.

(Refer Slide Time: 12:17)



Ordered List

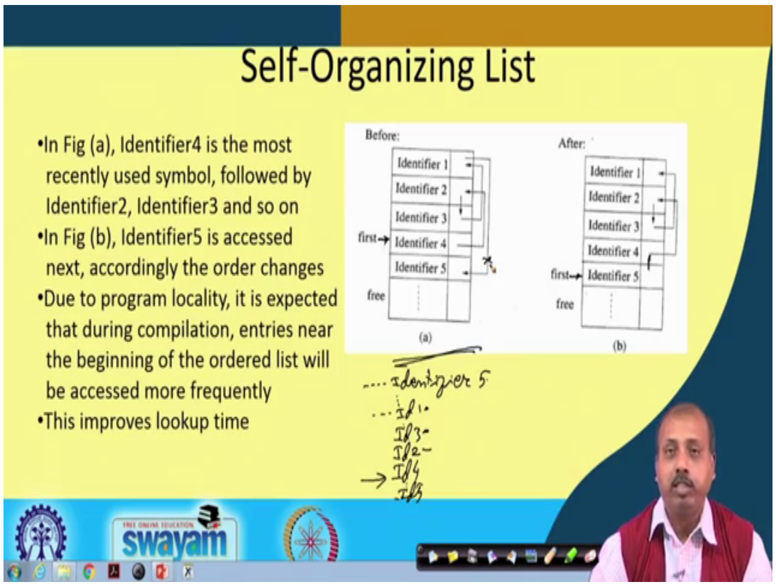
- Variation of linear tables in which list organization is used
- List is sorted in some fashion , then binary search can be used with O(log n) time
- Insertion needs more time
- A variant – self-organizing list: neighbourhood of entries changed dynamically

So, so this is the variation of linear list which is an ordered list. So, it is a variation of linear tables in which least organisation is used. So, this is the least organisation and least is sorted in some fashion and then a binary search is used for log n time. So, sorted so, if you are so, sorted in some fashion means it may be based on the name of the symbol you solve them on the ascending order. Then if it is so, then as you know that once the array is sorted so, we can use binary search and that binary search takes order log n time. So, that way it is easy; however, the problem comes when we are trying to do insertion.

Because, insertion it has to be made in ordered fashion; like if I have got say in the symbol table the mm symbols if we have in the symbol table the mm symbols like say a b and a c ok. And, then suppose I want I want to insert a new symbol a a then the difficulty is that you can so, you cannot insert a a at the end because, then this sorting order will be will be lost. So, what you have to do is that you have to have a a at the beginning followed by a b followed by a c. Now, for doing that this a c and a b they need to be copied down by one position making the first slot free and there you can insert this a a. So, that it becomes like this. So, as a result this insertion becomes order n time whereas, search remain search becomes log n order log n time.

So, this is the, this is a trade off that we have to that we have to do. So, it is a insertion is made more costly so, that this search becomes easier. A variant which is known as self organising list so, neighbourhood of entries are change dynamically. So, we will see how is it taking place. So, we change the so, the so, so that as and when these variables are accessed so, local variables so, they are neighbour to each other. So, that way we can have some better access method. So, we will see how this can be done.

(Refer Slide Time: 14:35)



Self-Organizing List

- In Fig (a), Identifier4 is the most recently used symbol, followed by Identifier2, Identifier3 and so on
- In Fig (b), Identifier5 is accessed next, accordingly the order changes
- Due to program locality, it is expected that during compilation, entries near the beginning of the ordered list will be accessed more frequently
- This improves lookup time

So, this is an organisation of self or this is, as this is an example of self organisationally self organizing list. So, you see suppose in so, first we consider this particular figure; this figure a in this figure a identifier 4 is the most recently used symbol. So, either so, we keep track of the sequence in which the symbols are being accessed. So, for so, the there is a pointer first or if that that points to the last referred a symbol ok. So, in this case identifier 4 is the most recently used symbol and before that it was identifier 2. And before that it was identifier 3, before that it was identifier 1 and before that it was identifier 5; that means, in my program there was a sequence of access like in the some in some lines I accessed identifier 5 identifier 5.

After that I have accessed before this identifier 5, identifier 1, then after some lines I have got this identifier 1. Then after some line we have got this identifier 3, identifier 3, after that I have accessed a identifier 2 and then identifier 4. So, that this identifier 4 this is the most recent one before that we have accessed identifier 2 in this statement.

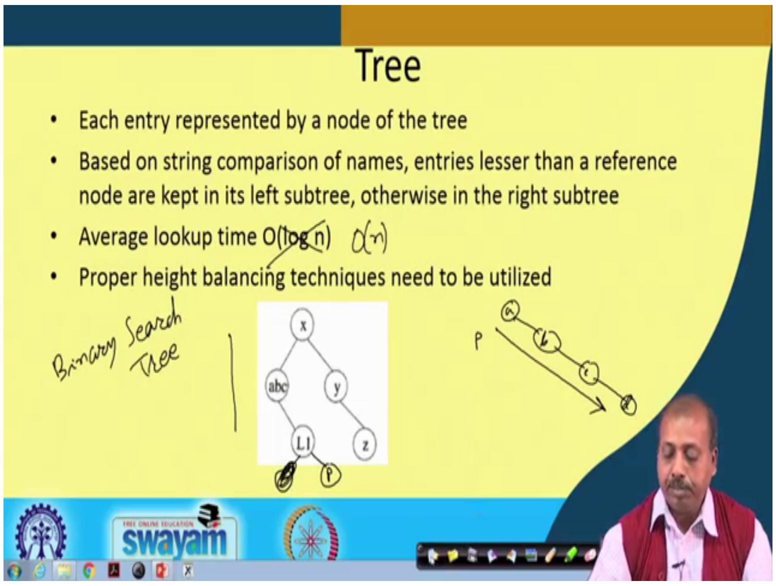
Before that we have accessed identifier 3 in this statement then identifier 1 in this statement and then identifier 5. So, so, this is the current sequence ok. Now, suppose after this so, the what is the advantage like it by program locality it is expected that in near future will be accessing in this order only. So, it is more likely that identifier 4 will be accessed compared to identifier 2.

And, identifier 3 will be just chances of accessing identifier 3 is even less, identifier 1 is even lesser and identifier 5 access probability is least. So, that is the expectation and that happens due to program locality. Now, after sometime so, after sometimes so, if identifier 5 is accessed. So, here suppose so, identifier 5 is accessed then this least will change the this first pointer comes to identifier 5 and this points to identifier 4 from 4; so, remaining part remains unaltered ok. So, that they are remaining unaltered and we are just remembering last 5 accesses. So, this identifier 5 access so, this particular link is lost because we are just remembering last 5 accesses.

So, that way we can have this mod this list modifiers. At any point of time when you are doing compilation so, this first pointer points to the least recently referred symbol. And then successive pointers so, they will be access they will be referring to the previously accessed symbols in that order. So, program locality so, it will be say that it will have the expectation that during compilation entries near the beginning of the order list will be accessed more frequently. So, this ID 4 ID 5 this ID 4 ID 3 ID 4 ID 2 ID 3; so, they are supposed to be accessed more compared to ID 5. But, here the example that we have taken so, it is slightly non-intuitive and all on a sudden instead of identifier 4 3 to identifier 5 has been accessed.

So, that can happen because it is not mandatory the program will always follow the locality of reference; sometimes it act does non local reference also. And here it is done in identifier 5 and as a result this list gates reorganized, but after that so, it will be now it is now it is having in the figure b. So, we are having the situation where the least recently least recently access symbol is pointed to by the first pointer. So, this way it can improve the lookup time because, I do not I can search this local identifiers much more faster compared to non-local non-local identifiers.

(Refer Slide Time: 19:19)



Tree

- Each entry represented by a node of the tree
- Based on string comparison of names, entries lesser than a reference node are kept in its left subtree, otherwise in the right subtree
- Average lookup time O(~~log~~ n) O(n)
- Proper height balancing techniques need to be utilized

So, next will be looking into another type of organisation, which is known as tree organisation. So, each entry represented by a node of the tree. So, here the entries are same like we have got the name of the symbol, then we have got its type, its offset etcetera. But instead of putting them on a linear table so, we are putting them in the form of a tree. And, based on string comparison of names entries lesser than a reference node are kept in the left sub tree otherwise it is on the right sub tree. So, if you look into the tree at this point you see that root is x and then the symbol a b c.

So, symbol a b c should come to the left side of x because, when you are doing a comparison the first characters are compare. So, x is compared with a and since x is if you look into the corresponding ASCII codes then x ASCII code is more than s ASCII code. So, as a result if x is taken to be greater than a b c or a b c is taken to be less than a. So, as a result this a b c is put on the left sub tree and similarly y y is greater than x.

So, y goes to the right sub tree of x and then when it comes to L1 so, you see that x L1 L so, L is less than x. So, it comes to the left sub tree. Now, this node has got the symbol a b c and this L is more than small a in terms of ASCII code. So, as a result L goes to the L1 goes to the right sub tree of a b c.

Similarly, here when z comes so, z is greater than x. So, it is in the right sub tree of x then the compared with y it is greater than y. So, z is ultimately put in the left sub right sub tree of y. So now, if we have got a new symbol coming for example, suppose there is a symbol say P; now where will it go? So, P so, it is less than x. So, it will come to the left sub tree of x then it is a b c. So, a is greater than so, P is greater than a so, it will come to the right sub tree; right sub tree has got this L and this P is less than L. So, P will come a P is sorry P is greater than L. So, P will come at this point K L M N O P so, P is greater. So, P will come to the right sub tree so, that way these new symbols will be put into the symbol tree into this tree type symbol table.

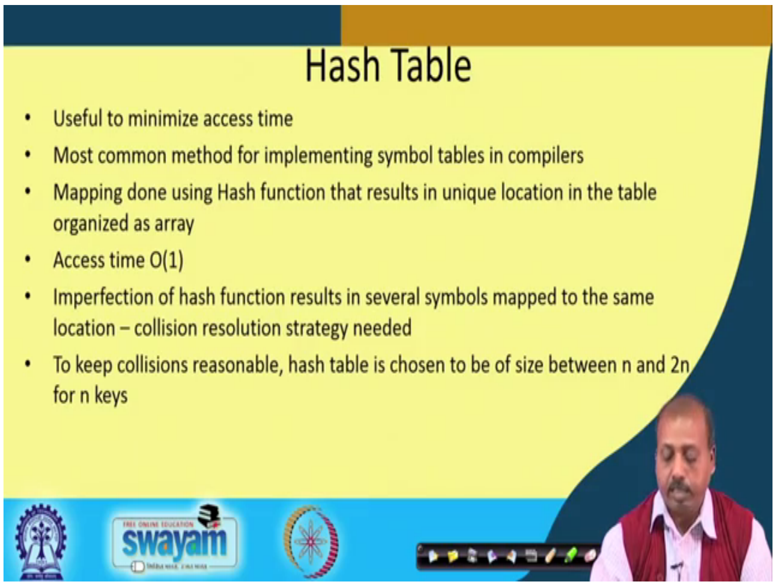
So, what you are doing is we are maintaining some structure which is known as binary search tree. So, your maintaining this structure of binary search tree such that the nodes which are less than the root node; so, they will go to the left sub tree and nodes which are larger than the root so, they will go to the right sub tree. And, this process we will go on recursively.

Now, look up top L average look up time will be order log n because, there are if there are n symbols and it is pa perfectly balanced. If this tree is perfectly balanced then there will be log n such levels of course, that may be a situation where this symbols are such that they give rise to a tree like this. Suppose I have got the symbols coming in this sequence a b c d like that such that successive symbols are all greater than the previous symbol.

So, this will create a tree like this and in that case complexity instead of being log n; so, this will become order n such complexity will become order n. But for all practical purposes so, this will not happen because we will be getting symbol such that we get the left and right branches both and it becomes more balanced. And, if it is not balanced then there are some techniques by which we can do this balancing which is known as height balancing. I will suggest that you look into some data structure book that will be talking about this height balanced tree and all.

So, proper height balancing techniques can be utilised; so, that this complexity of this lookup can be made to be equal to log n. So, this way we can take help of this tree type of structures.

(Refer Slide Time: 23:41)

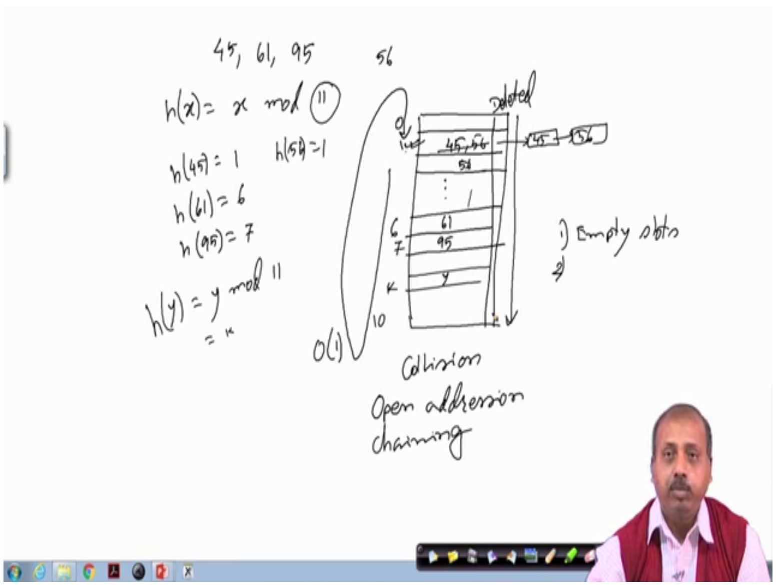


Hash Table

- Useful to minimize access time
- Most common method for implementing symbol tables in compilers
- Mapping done using Hash function that results in unique location in the table organized as array
- Access time O(1)
- Imperfection of hash function results in several symbols mapped to the same location – collision resolution strategy needed
- To keep collisions reasonable, hash table is chosen to be of size between n and 2n for n keys

Next we will be looking into another structure which is known as hash table. So, hash table this is useful to minimise access time. So, when it is the most common method for implementing symbol tables in compilers. So, what is a hash table?

(Refer Slide Time: 24:03)



So, hash table is something like this like suppose, I have got some symbols some numbers. Let us a work with same numbers say 45 61 and say 95. So, these are the 3 numbers and what we do we take so, so if this is equal. So, you take a function h of x which is x mod 11. So, 45 mod 11 so, h 45 will give me 1 ok; then h 61 will give me 6 and h 95 will give me 7 ok. Now, if it is like this then we can so, what we do in the table we have got entries running from 0 to 10. If it is mod 11 so, that entries will be running from 0 to 10. So, in the 0 slot there is nothing, in slot number 1 we have got 45, in slot number 6; so, these are all entry a empty at slot number 6 we have got 61 and at slot number 7 we put 95.

So advantages that now, suppose I give you a number and ask whether it is there in the table or not search for a particular number. So, for any particular number is given say y is a number given as a question. So, what we do we apply the h function on y to get y mod 11 and we need to come to the corresponding index. So, if this value is equal to say k then we look at the kth entry into this table. So, if this number is there if the number y is there as per our policy the number y will be in this slot only.

So, as a result for searching a number I do not have to go for a sequential search through this entire table, I can do it in constant time. So, this access can be made constant time. However, the difficulty with this type of organisation is that there will be many slots which are empty. So, in between there may be many empty slots; so, that is one problem.

So, first problem is empty slots because depending upon the, this mode value that we have got. So, I will get in day array indices in that range table Indies in that arranged. So, many of the entries may not have the corresponding values there and second thing is that there can be several numbers which fall on to the same index like say; so, if I have got say 56, suppose the next number given is 56. So, when I apply this function so, h of 56 is also equal to 1. So, far so, I have I need to store 2 numbers here now 45 and 56, but that is not possible because every slot has got a single number. So, this is known as the problem of collision.

So, if you look into the data structures and there the there are some techniques by which we can resolve this collisions. So, so, what you can do maybe we store 56 at the next free slot ok; the next free slot that is available. Or, what we can do is that we can we do not put the numbers here directly rather we make a chain where we put this numbers on a chain 45 56 like that. So, these are these are known as collision resolution techniques.

So, we resolve collision like that and then if we are depend you doing like this. So, when we are putting this number of the next available free slot. So, this is called open addressing, this collision resolution technique is called open addressing and the other one where we are putting them on a change. So, they are that is that technique is known as chaining.

So, both the methods though they have got their overheads like you have got we have got this thing; we need to somehow search through this chain or search through this entries ok. So, till for example, if the question is say 56 and we have use a open addressing so, 56 is here. So, applying the hash function will come to slot number 1 and then we have to go on searching from this point and round come back to that entry. So, if the number 56 is not there, then will be searching for the entire table and we will not be getting it so, that, that at that point we can declare that the number is not present in the table.

But it is a bit costly definitely whereas, so but it is a very much likely that if the number is there then you can get it the then you can get it within a very short amount of time and chaining is slightly better than that. The difficulty that we have is with the deletion because, if you want to delete an entry then how do you delete it; how do you say that this particular slot does not have the number. So, we need to have another flag here there which we which we call say deleted flag and that flag has to be turned on if a number is deleted. If a number is not called if this particular table entry is not containing a valid data in the deleted flag should be on.

Now so, for symbol tables of this is not a big problem because for symbol table we can just have this deleted we can have the deletions are very less. So, we have got only in mostly insertion and search and the search is the most important operation in symbol table. And if we have got a hash table type of organisation then this search can be done very fast.

So, that way that is why this hash table based say symbol table hash table based symbol table organisation is very popular and it is used in many compiler design or compiler designs. So, that when there are large number of symbols and a symbol table is very large; so, instead of searching through the table in linear or binary search fashion. So, it is better that we organise them as hash table and in constant time, we can access the individual entries from the table.