Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

Indian Institute of Science, Bangalore

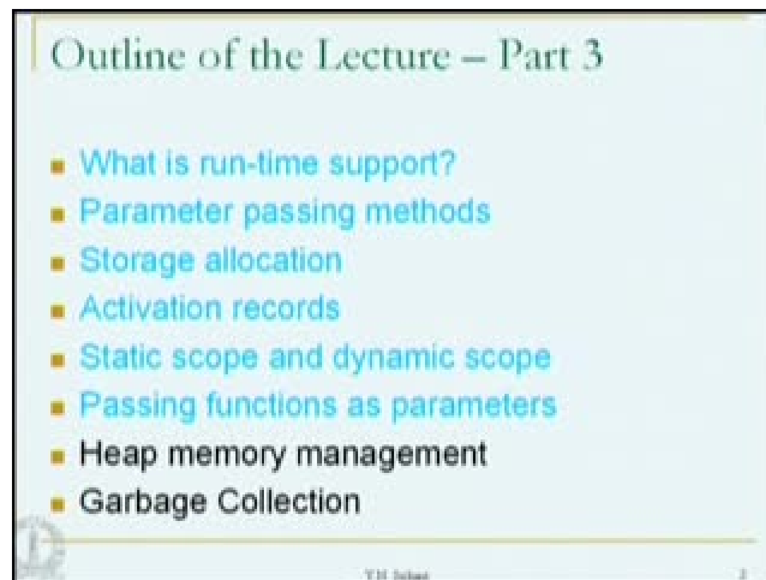
Module No. # 02

Lecture No. # 05

Run-time Environments-Part 3 and

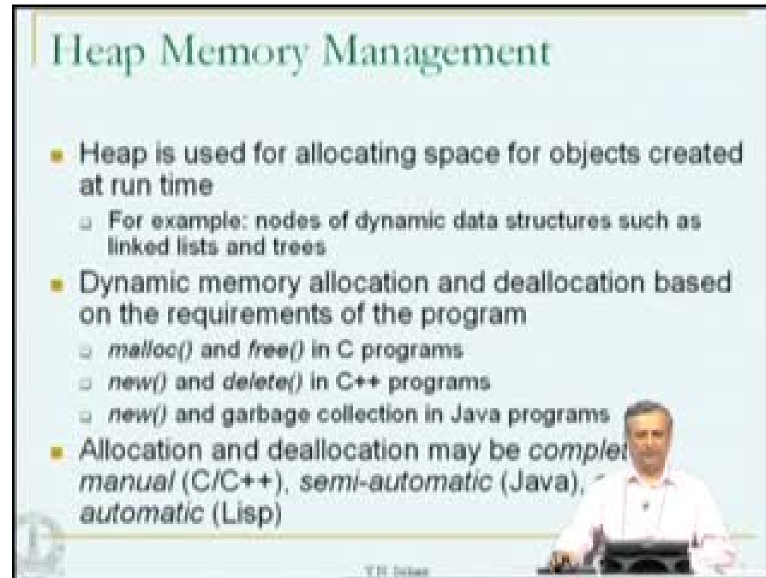
Local Optimizations

(Refer Slide Time: 00:23)



Welcome to part 3 of the lecture on run time environments. In the last part of this lecture, we learnt about various parameter passing methods, storage allocation, activation records, static scope, dynamic scope, passing functions as parameters and so on.

(Refer Slide Time: 00:50)

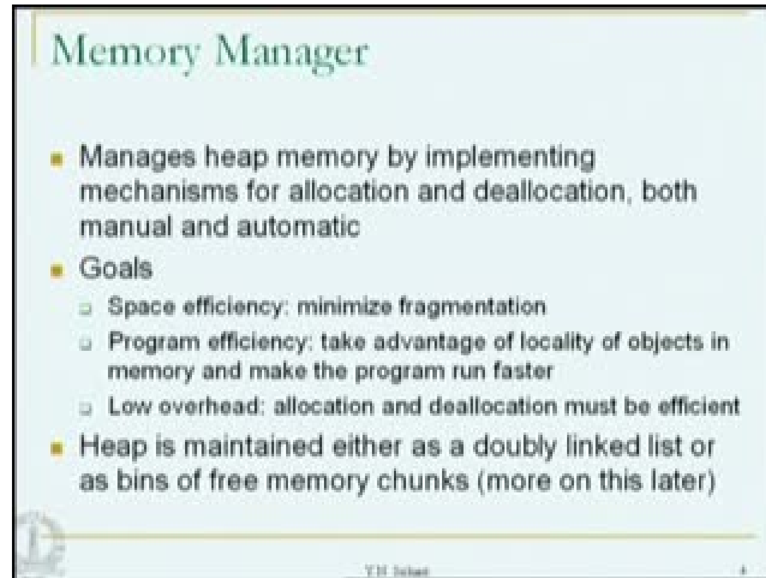


Today, we will discuss heap memory management and also garbage collection. What is heap memory management? What is a heap? We must describe what a heap is.

If you look at the very systematic allocation and deallocation of activation records, it is clear that we can use regular stack as far as activation record allocation and deallocation is concerned, but if you consider dynamic data structure such as linked lists and trees, the nodes of such lists and trees are created arbitrarily all over the program possibly and they are also deleted all over the program. The allocation, deallocation is not as systematic as in the case of activation records.

Therefore, we require a slightly different mechanism - The dynamic memory allocation, deallocation to perform dynamic memory allocation and deallocation, in the case of a heap. In C programs, we have malloc and free to perform allocation and deallocation; the equivalents in C plus plus programs are new and delete; and finally, in java programs, there is only allocation, which is going to happen using the new call; there is no deallocation by the programmer. It is actually done automatically by the java run time system. What happens is that when the program runs out of the heap memory, a call to the garbage collector is made. The garbage collector collects all the memory area which is not in use, and returns to the program and continues with the allocation. That is how allocation and deallocation happen in various programming languages.

(Refer Slide Time: 02:50)



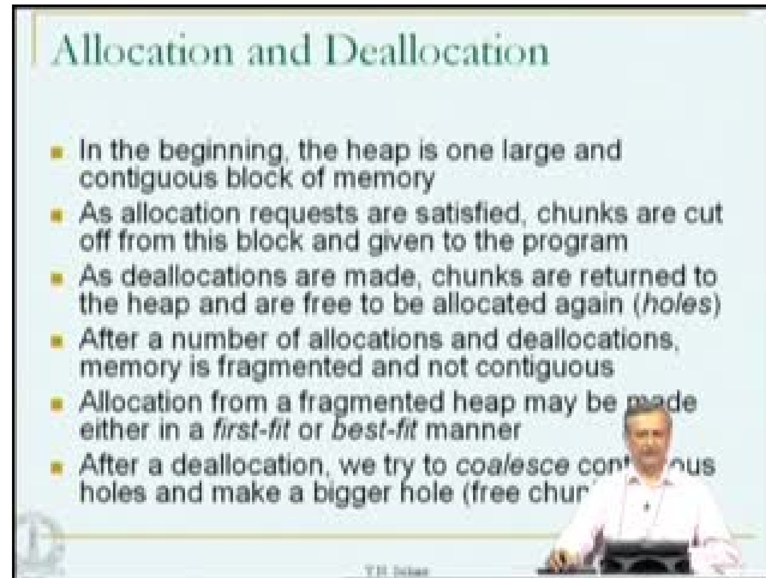
A memory manager is the one which manages heap memory; it implements the allocation, deallocation policies - possibly manual or possibly automatic.

Of course, the goals of a good memory manger are many; first of all, space efficiency - as we will see soon, whenever there is allocation followed by deallocation at random, the heap memory available gets fragmented. The problem with fragmentation is that you may have enough memory available in small chunks, but request for very large chunk of memory cannot be satisfied.

This is fragmentation; and we need to minimize fragmentation so that we can satisfy as many request for allocation as possible; then the efficiency of the program itself - we must actually take advantage of the locality of objects in the memory and make the program faster. How do we do this? If we are able to take advantage of the locality of objects, then possibly the cache will accommodate many objects which are going to be used very frequently, and when objects are located in cache, obviously, the programs run faster.

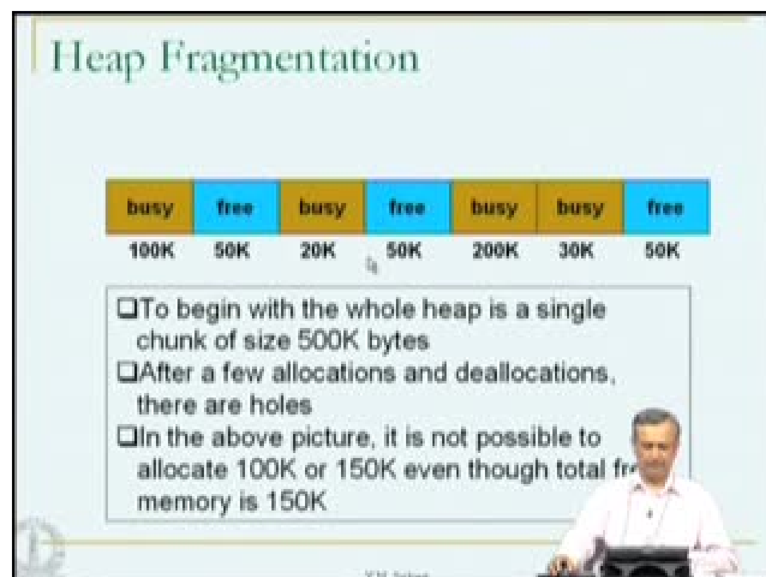
The memory manager must also have low overhead. Allocation and deallocation by themselves should not take too much time; they must take as little time as possible. The heap also needs a data structure for its maintenance. Usually, it is maintained as a doubly link list or as bins of free memory chunks; these are all going to be discussed, in some detail, in the coming slides.

(Refer Slide Time: 05:08)



What is allocation and what is deallocation? Let us look at a picture along with this slide in order to understand. In the beginning, the heap is one large and contiguous block of memory. Then, as allocation requests are satisfied, parts of this large piece of memory are cut off from this block and given to the program. These are based on the allocation requests of the program and then, possibly, some nodes are deleted in the data structure. So, deallocations are performed by the program. Then, these chunks, which are deallocated, are returned to the heap and now they can be allocated again. Such free chunks are called as holes.

(Refer Slide Time: 06:17)

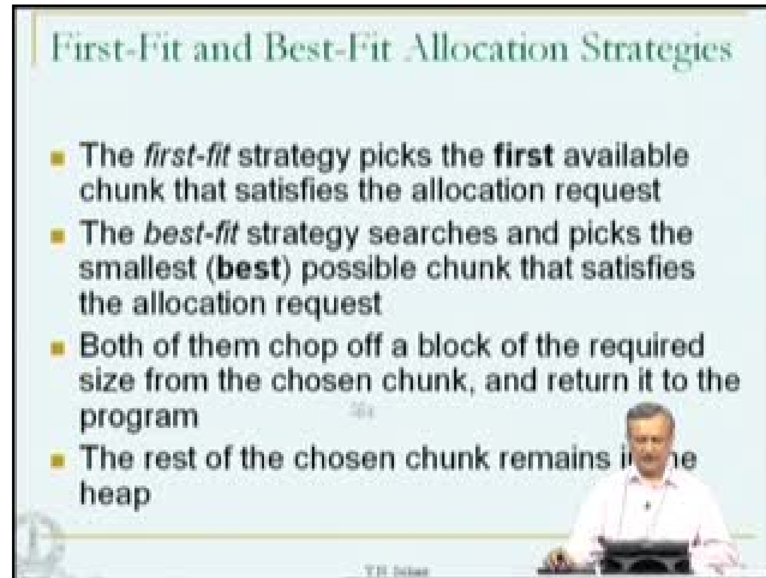


After a number of such allocations and deallocations, memory gets fragmented and it is not contiguous any more. Let us look at a picture to understand what is happening. Here is a large piece of memory; to begin with, the whole heap is a single chunk of size 500k bytes.

Let us say we allocated all these to begin with; we made a request for 100, then 50, then 20, then 50, then 200, then 30, then 50 and so on. Then, somewhere in the middle of these allocations, we also freed this particular chunk of memory 50k; and then we freed another 50k; finally, we freed one more 50k. It is just coincidence that these are all equal in size; they could have been even the 200k or the 20k nodes. Once a sequence of such allocations and deallocations happens in any particular order, the memory looks as shown here; these are the free holes. These are the only ones which can be allocated to the program when there is a request. For example, if there is a request for 50k block, this can be released or this can be released or this can be released, but if it is for less than 50k, then we can cut off a part of this free memory and return it. If the request is for more than 50k, say 70 or 100k or even more than 100k, say 110 and 120 etcetera, even though the total size of the free memory is 50 plus 50 plus 50 - 150, none of these requests can be satisfied. It is not possible to combine these two into a single piece of memory and then return that contiguous piece as what is required by the program.

Let us go back. Allocation from a fragmented heap may be made in either a first-fit manner or the best-fit manner; we will see what these are. After a deallocation, we try to combine or coalesce contiguous holes and make a bigger hole. This is needed to minimize the fragmentation; that is what I was explaining a few minutes ago. If possible, we could shift this free chunk to this place, then the busy chunk to this place and then may be combine the two free chunks to get a much bigger chunk. These are possibilities, not that every memory manager does all these.

(Refer Slide Time: 09:00)

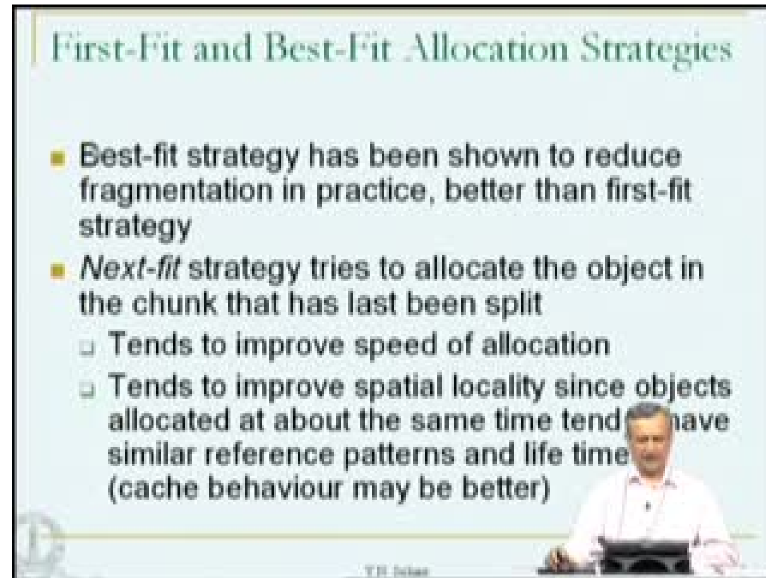


What is first-fit and what is best-fit? The first-fit strategy: it always picks the first available chunk that satisfies the allocation request. There may be many chunks, which satisfy the request; as I told you it is maintained as a doubly link list.

The free chunks are all maintained as a doubly link list. Any one of these, which actually satisfy the programmer's request, can be picked and then returned to the program. So, in the case of first-fit, the first one which satisfies the request is taken and returned to the program. Of course, the whole chunk, if it is much greater than what the program requires is not returned; part of it, which satisfies the programs request is cut off, chopped off from the free memory block and then returned; the rest remains in the doubly link list pool.

In the case of best fit strategy, the allocator searches through the doubly link list of the free blocks, picks the smallest which is the best possible chunk that satisfies the allocation request. In such a case, even if that smallest chunk is bigger than what we require only a small part will be left out in the heap because of this chopping process. Whereas in the first fit case, even if is the first one is much bigger than what we require, then large piece has to be cut off and the remaining large piece remains in the heap. This may actually increase the fragmentation. So, rest of the chunk remains in the heap itself.

(Refer Slide Time: 11:00)



As I explained the best fit strategy has been shown to reduce fragmentation in practice; that is the best. It is much better than the first fit strategy because first fit strategy cuts off chunks from free memory in some arbitrary manner and is not like the best fit strategy. These two, first fit and best fit are really very well-known strategies. A slightly less known strategy is the next fit strategy. What is it? The next fit strategy tries to allocate the object in the chunk that has last been split. In other words, suppose there is a free chunk x which was actually very large, so the allocator chopped off one part of it and returned it to the program at this instant. The next allocation is also made from the same chunk. So, whatever was chopped and returned to the pool, the remaining one that stays in the heap; that is used for allocating the next request as well.

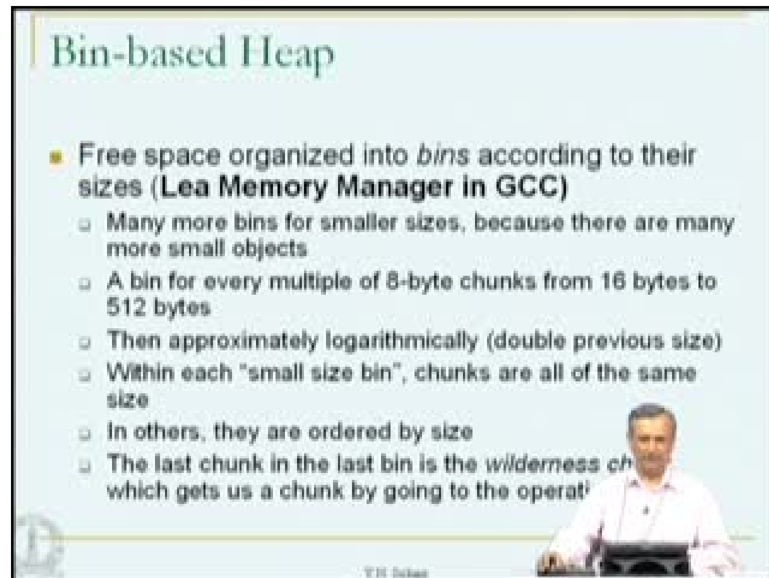
If this happens, it obviously improves speed of allocation. Why? The pointer in the doubly link list - one of the pointers will be pointing to the last allocated or chopped off chunk. If the next request is automatically done from that particular chunk, the speed of allocation is good; allocation is done immediately. There is no need to search anything; searching will be definitely required if that chunk does not satisfy our request; then you have to find the different chunk .if it satisfies the request then the same chunk is used.

It tends to improve the spatial locality also, since objects allocated at about the same time tend to have similar reference patterns and life time as well. The result of this is these

objects get into the cache one after another and the behavior of the cache may become much better.

These are all together; they all get into the cache at the same time. The program is going to refer to these objects very soon and therefore, the behavior becomes much better.

(Refer Slide Time: 13:37)

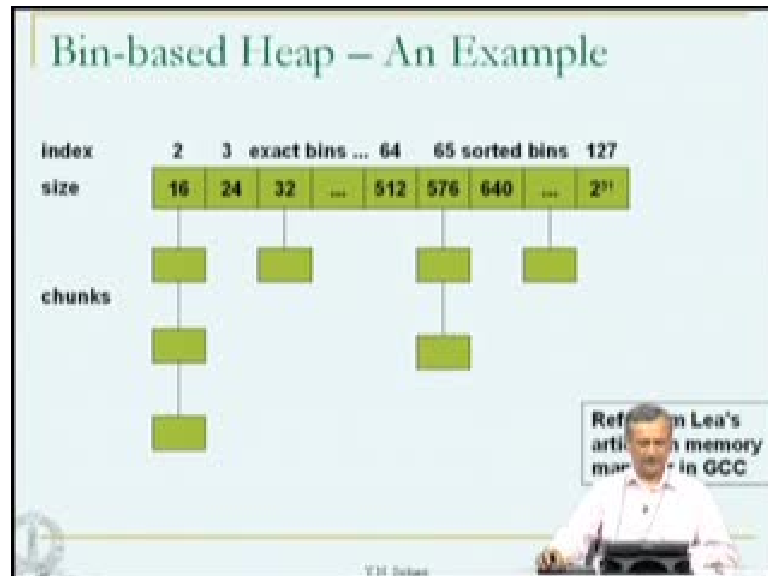


That was about first-fit and best-fit strategy and that used doubly link list based approach for allocation and deallocation. Here is a different strategy called the bin based heap. In this case, the free space is organized into bins according to their sizes. Such a memory manager was proposed by Lea; therefore, it is called as the Lea memory manager and that is what is used in the GNU C compiler, GCC.

Many more bins for small sizes exist in such a manager because there are many more small objects than large objects; that is the statistical information. Small sizes are used many times where as large sizes are not used so often.

A bin for every multiple of eight byte chunks from 16 bytes to 512 bytes is used. Let me show a picture.

(Refer Slide Time: 14:53)



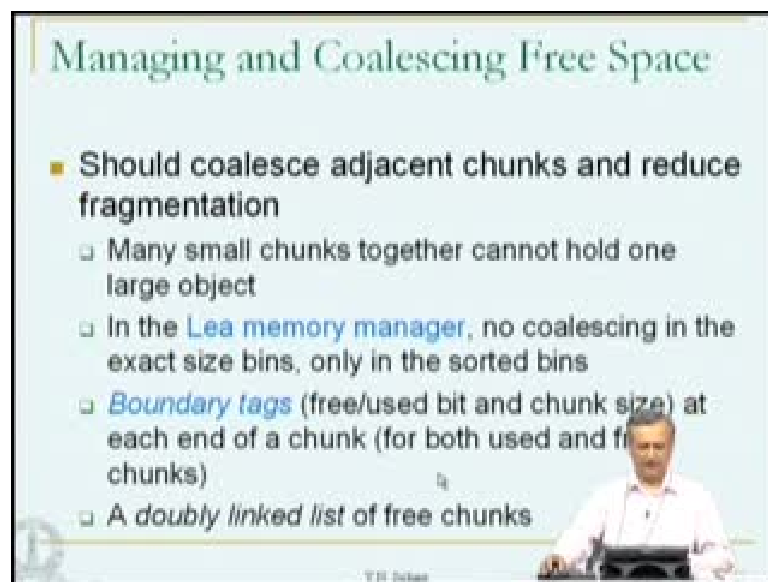
These are the various bins and these are the chunks which are actually used for allocation. Here from 2 to 64 we have exact size bins; in other words, the chunks here are all of 16 bytes size. The next one will have a linked list here, we have not shown any because of lack of space. The linked list here will have chunks of size 24 bytes; the third one will have chunks of 32 bytes and so on and so forth. Up to this sixty fourth bin which will have chunks of size 512, these are all exact bins, there is no variation in the size of these bins. They are all of the same size; these are all of size 16; these are all of size 32 etcetera.

Then we have many of these bins which actually have variable size chunks. Let us see. A bin for every multiple of 8 byte chunks from 16 bytes to 512 bytes that is what I just now explained. Then approximately logarithmically, in other words, every bin is of double the previous size - that is here.

So, 576, 640 and so on and so forth. These are the sizes of the various bins. There are chunks of this size possibly between 512 and 576, between 576 and 640 and so on and so forth. That is how it would be. Within each small size bin, chunks are all of the same size. I mentioned this just now. In others, they are ordered by size. Here these chunks are not of the same size they may be of different sizes, but they are stored in sorted order. So, between 512 and 576, there are many chunks; they are all stored here, between 576 and 640, they are all stored here and so on and so forth.

The last chunk in the last bin is called as the wilderness chunk which gets us a chunk by going to the operating system. What does that mean? (Refer Slide Time: 17:22) For example, here there may be many chunks. Suppose the memory manager wants something which is much bigger than what is available even in the last bin - this particular bin. The last chunk or the node of this particular linked list is called as the wilderness chunk; that is not really a chunk. It is actually going to be some kind of a sentinel and when we reach it, the memory manager makes a call to the operating system saying that we do not have any more memory. Please give us more memory. Then more memory is returned to the memory manager and the operation continues from that point onwards; that is why it is called as a wilderness chunk.

(Refer Slide Time: 18:18)



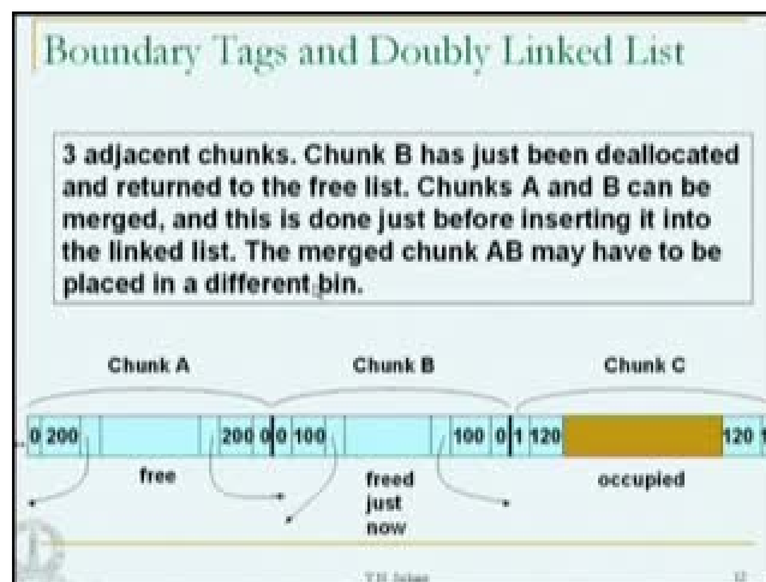
How does one manage and coalesce free space? So far, we saw first fit based strategy then, best fit based strategy and bin based strategy for allocating and deallocating space. How is free space managed? Let us see.

Whenever there is deallocation of chunk of memory, it is necessary that the adjacent free chunks be coalesced. If they are coalesced, then fragmentation reduces because the chunk size now increases. The problem with fragmentation is there are many small chunks and we cannot satisfy a big request. If it is possible to combine smaller chunks and make them bigger; that can be done when free chunks are adjacent to each other; then the fragmentation automatically reduces.

Many small chunks together cannot hold one large object. In the Lea memory manager, no coalescing in the exact size bins, only in the sorted bins. Exact size bins have chunks of the same size. So, there cannot be any coalescing performed on those bins, but in the other bins that we saw there are chunks of various sizes. So, it is possible to coalesce them. Only thing is when you coalesce and they become much bigger, they may be have to be shifted to a different bin altogether.

Then how is this coalescing done? We use what are known boundary tags. This is a bit which indicates whether the chunk is used or it is free and there is also information about the chunk size. This is maintained at the end of each chunk for both used and free chunks and there is a doubly linked list of free chunks. This is true for first fit, best fit strategy and also for the bin based strategy. Let us look at a picture.

(Refer Slide Time: 20:28)

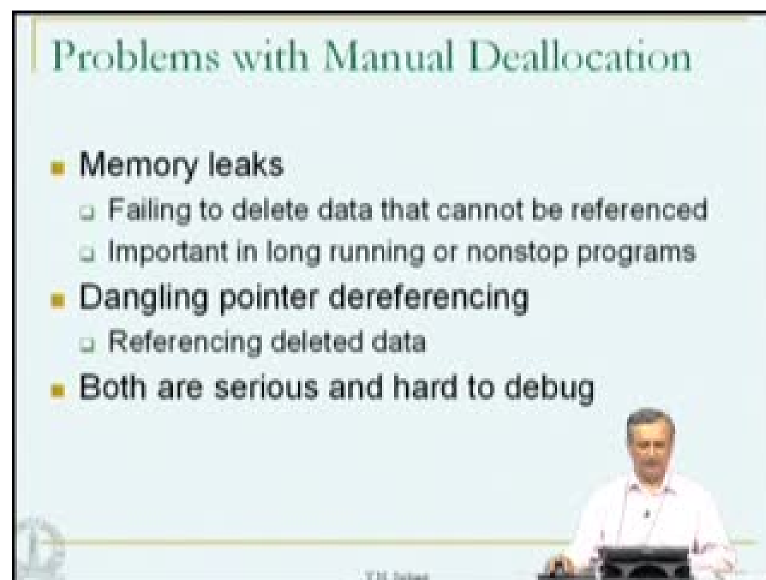


We have three chunks here. All the three chunks are adjacent. C is occupied, so, it is really being used by the program. Chunk B has just been deallocated; A was deallocated quit some time ago. This was returned to the free list or the heap. Now, because A and B are adjacent they can be merged and we can create a single node of double this size. How can this be done? Let us look at the boundary tags first. 0 here 0 here, at the two ends of chunk A indicate that chunk is free. Similarly, 200 here and 200 here at the end of chunk A indicate the size of this free chunk. The same is true for chunk B. Observe that this boundary tag and this boundary tag are zero indicating that both are free, but this chunk

is also free; whereas for chunk C, which is occupied, the boundary tag is one; the size information is maintained; and the boundary tag information is maintained for all free and occupied chunks. When this particular chunk B is freed, it is easy to see that the adjacent bit - the boundary tag bit, free used bit can be checked. It is one location beside this in memory. If that is a zero, then we know that the entire chunk is free. When it checks the chunk on the right side, it is one; therefore, entire chunk is not free. So, you cannot really coalesce B and C; you can only coalesce A and B. When we coalesce A and B, these two links have no meaning anymore; it is only this link and this link which matters to us.

(Refer Slide Time: 22:25) We manipulate only these two links and we forget about these two links. This boundary remains as zero; the size information becomes 200 plus 100, 300. Here it becomes 300 and here also it becomes 300. That is how chunk coalescing happens. The merged chunk AB may have to be placed in a different bin, if it is a bin based memory manager.

(Refer Slide Time: 22:55)

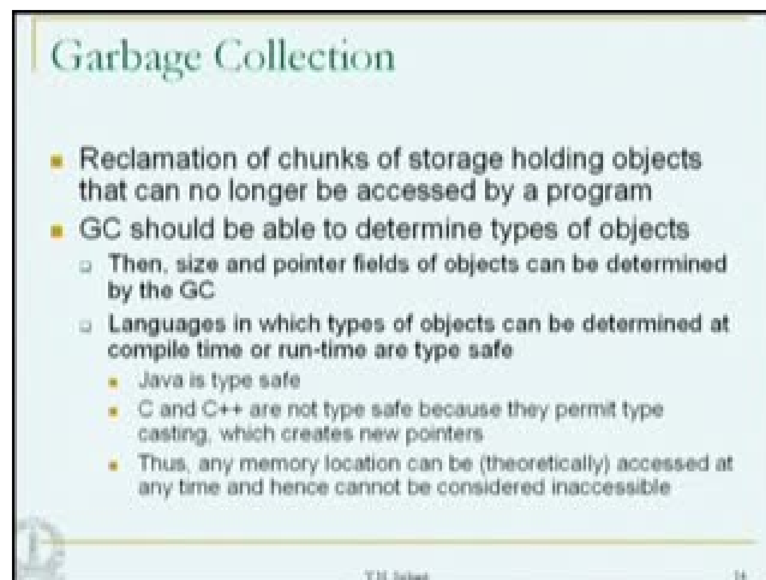


What are the problems with manual deallocation? For example, there may be memory leaks: failing to delete data that cannot be referenced. We have actually started manipulating links in a linked list. May be we made a small mistake in the program. A link which was pointing to the next node now points to some other node. So, the node which was being pointed to is not reachable any more. It cannot be referenced at all, but

the programmer forgot to delete that particular node. So, this is failing to delete data that cannot be referenced any more. The memory is being used; it has not been returned to the pool or the heap; that memory cannot be used for allocation and it is a waste. This is a memory leak and such memory leaks are very important in long running or nonstop programs such as monitors and operating system programs because it is not as if the program stops and memory gets flushed and all the objects go away and we start all over again. The memory which is wasted in this manner will actually be wasted forever; there is nothing we can do about it.

So, memory leaks are very important in such long running or nonstop programs. Then we have the famous dangling pointer dereferencing. Here we are referencing deleted data. This is very common. You think that the link is not null; you try to actually do some a dot or a pointer data and so on and so forth. Unfortunately A is null. There was a problem with the program; it was not written well. A is null and you are trying to access null dot something; not possible at all. This is a dangling pointer dereferencing problem; it happens because of programmer's bugs.

(Refer Slide Time: 25:15)

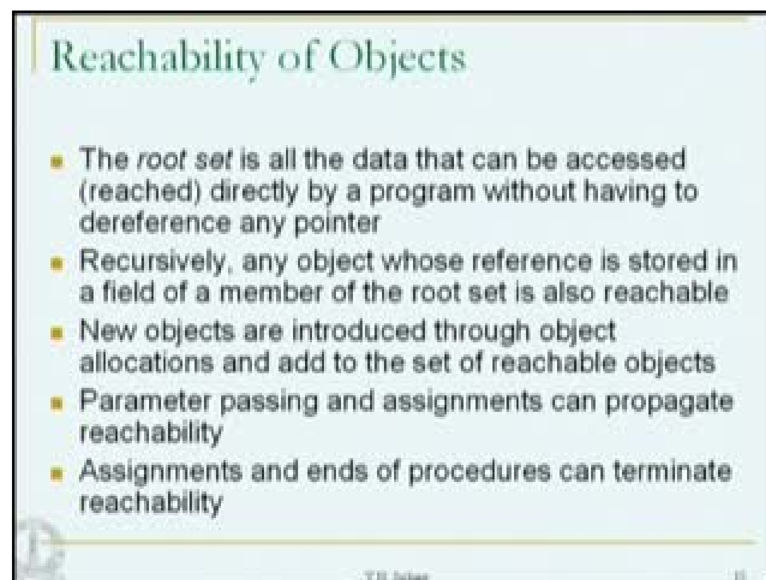


Both these are very serious and very hard to debug. Therefore, if we can actually do allocation automatically, then some of these problems may be eliminated. So reclamation of chunks of storage holding objects that can no longer be accessed by a program is called garbage collection.

GC should be able to determine types of the objects. We will know why, very soon we will know that. Then the size and pointer fields of objects can be determined by the garbage collector. Once I know the type of the object, I know the size of the object as well - this is the point. Languages in which types of objects can be determined at compile time are known as type safe. Even if they can be done at run time, even then they are called type safe. So, it is even stronger, if it can be done at compile time.

Java is type safe. You can determine the types of objects, most of the time, at compile time. C and C plus plus are not type safe. Why? It is because they permit type casting. You can convert any type to any other type in C and C plus plus which creates new pointers. Thus, any memory location can be accessed theoretically at any time and hence it cannot be considered as inaccessible. You have a problem here. You type cast one variable to another type and you try accessing it. In other words, you can make anything a pointer anytime, try accessing any part of the memory. Well, you may actually get a core dump and the program may stop, but that is beside the point.

(Refer Slide Time: 27:27)



How is garbage collection performed? The concept of reachability of object is very important here. See, there is something called a root set. The root set is all the data that can be accessed or reached directly by a program without having to dereference any pointer. In other words, most of the time these are the variables which are declared by the programmer.

Of course, there are slightly error possibilities, but most of the elements of the root set are programmer defined variables - pointer variables. Then, we determine this root set to begin with. They are all pointers. So, recursively, any object whose reference is stored in a field of a member of the root set is also reachable. So these are now first level are programmer defined variables, they are definitely accessed by the programs and so they are reachable. Now you look at the inside of this particular object which is pointed to by one of the field members of the root set. That is also reachable. Many pointers has fields; so, the objects that can be accessed from these pointers; they are all reachable.

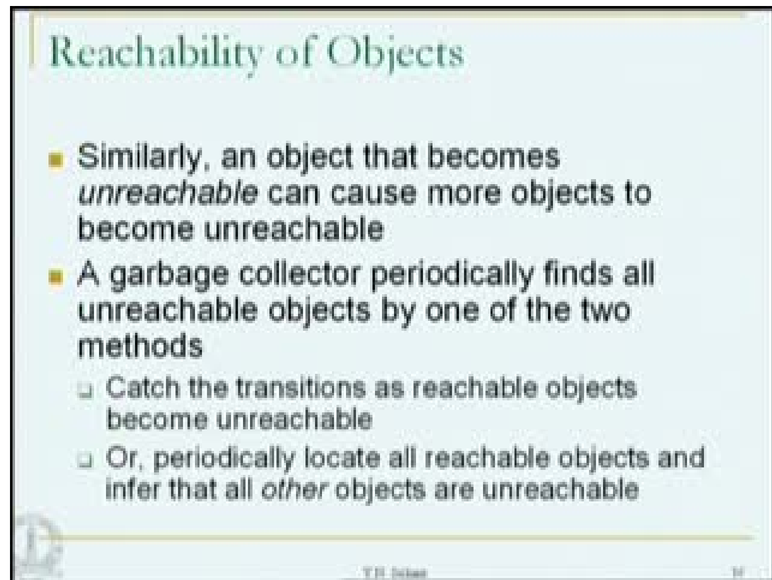
This goes on recursively. In other words it is like a graph. Imagine a tree. Start with the root, which is pointed to by a programmer defined variable then slowly, gradually the left link and right link can be traversed then they become reachable. Then the left link and right link of the children can be traversed and so on and so forth. So recursively the whole tree becomes reachable.

New objects are introduced through object allocations and they are added to the set of reachable objects. How we do this, we will see very soon.

Parameter passing and assignments can propagate the reachability of these objects. New objects are created through new or malloc calls and when we pass some pointer as a parameter to another function, then the objects pointed to by this parameter are visible within that function also, then you can manipulate a little more and the reachability propagates.

Assignments and ends of procedures can terminate reachability. If I have A equal to B, both are pointers, whatever was pointed to by A is now destroyed. B is copied to A now. From now on that variable A points to whatever B points to. So, this assignment has ended the reachability of objects which were reached by A. Similarly, ends of procedures: when I return from a procedure all the local variables are lost. So, reachability of objects from those local variables is also reduced or affected.

(Refer Slide Time: 31:06)



Similarly, an object that becomes unreachable can cause more objects to become unreachable. I will show you a picture to demonstrate this. What does a garbage collector do? A garbage collector periodically finds all unreachable objects by one of the two methods: first method is catch the transitions as reachable objects become unreachable; we will see the reference counting method which is one such method or periodically locate all reachable objects and infer that all other objects are unreachable. This method we are not going to describe. There are many such methods under the second category; they are far more complex than the garbage collector itself and become the topic of very serious discussion

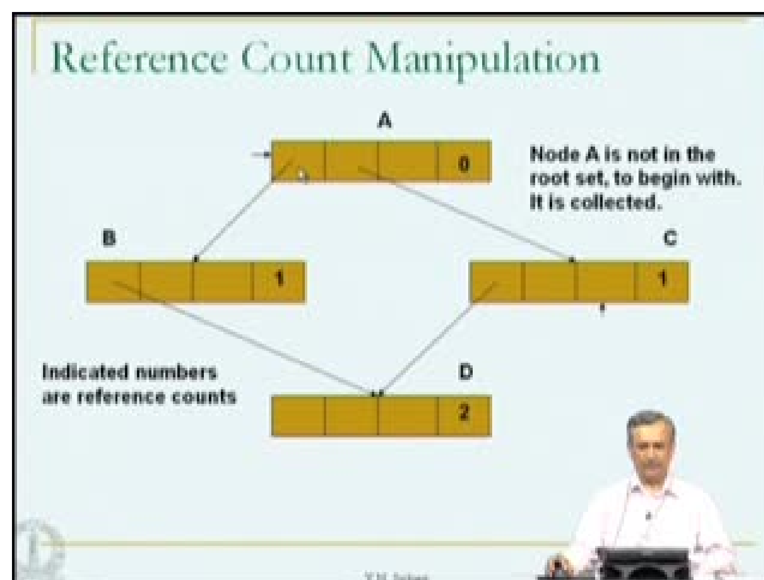
(Refer Slide Time: 32:04)

Reference Counting Garbage Collector

- This is an approximation to the first approach mentioned before
- We maintain a count of the references to an object, as the mutator (program) performs actions that may change the reachability set
- When the count becomes zero, the object becomes unreachable
- Reference count requires an extra field in the object and is maintained as below

Let us look at the reference counting garbage collector, which is an approximation to the first approach mentioned above. That is when an object changes its state from a reachable object to unreachable object we catch that object and do something. Let us see what we do. We maintain a count of the references to an object. As the program or the mutator performs actions that may change the reachability set. So, this count also is going to be changed. When the count becomes zero, the object becomes unreachable and reference count requires an extra field in the object and is maintained as below.

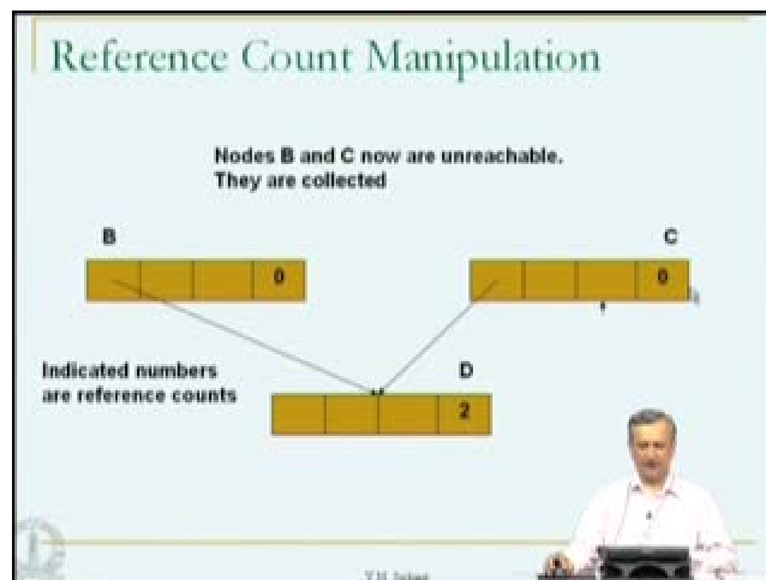
(Refer Slide Time: 33:00)



Let me show you a picture to understand reference count manipulation and then we go back to discussion. Here is a data structure; there are four nodes A, B C and D. A was pointed to by some programmer defined variable. The programmer did not want this data structure anymore. So, he made the pointer which points to A point elsewhere. This particular thing became garbage; it is not reachable anymore, but the reference counting fields do not say the same thing. For this, the reference counting field has now become zero because it is not reachable from any other variable or pointer.

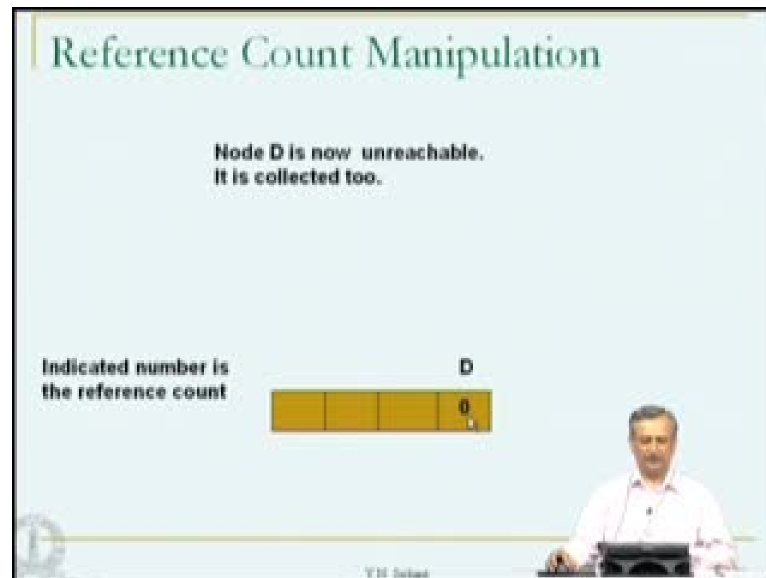
(Refer Slide Time: 33:45) This reference count field is one because this node is pointing to it. Similarly, the reference count field for this is one because this node is pointing to it. The reference count field for D is two because both these nodes are pointing to it, but the situation changes very soon. The reference count of this is zero. So, it is garbage collected; this disappears.

(Refer Slide Time: 34:13)



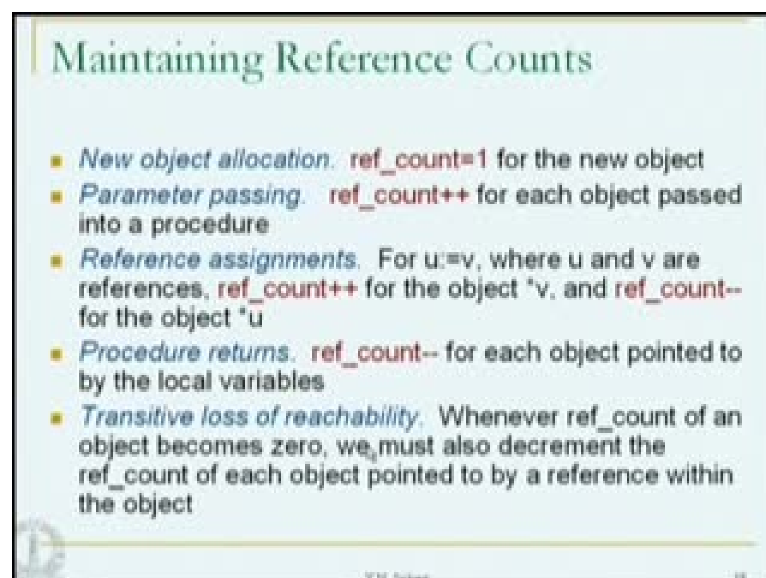
Now, because the node A disappeared, the reference counts of B and C have also been made zero; that happens during the collection process. So, these two will also be now removed by the garbage collector.

(Refer Slide Time: 34:31)



Then the reference count field of D also becomes zero and that is also collected by the garbage collector. Let us go back and see how the reference counts are maintained.

(Refer Slide Time: 34:40)



If there is a new object allocation, then reference count is set as one for the new object. The new object is created. There is a pointer pointing to it and that is what is returned by

the malloc or new call. It is reasonable to set the reference count one for the new object; there is only one pointer to pointing to it.

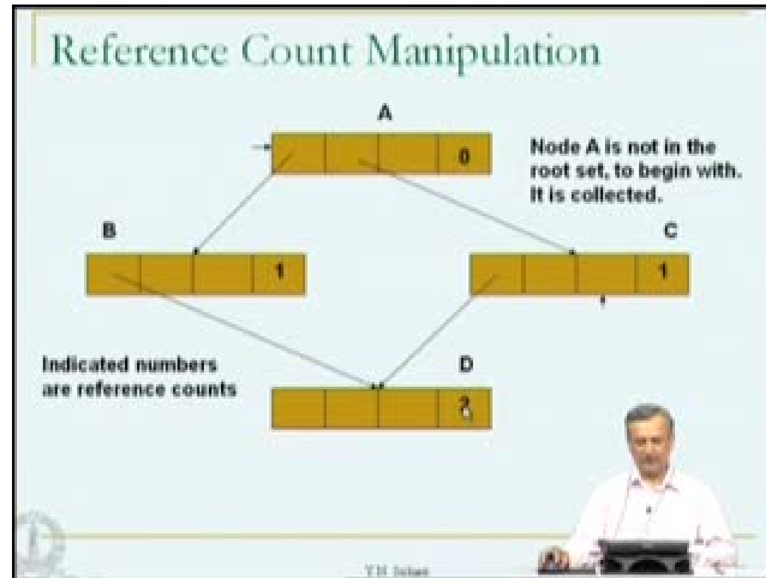
Parameter passing: Reference count plus plus for each object passed into a procedure. That is each parameter that comes into the procedure is pointing to some object. This becomes a variable in a different procedure. We can say that the reference count now becomes one more because there is a new variable which is pointing to it - that is parameter itself.

Reference assignments: u equal to v is the assignment statement where u and v are the references. Reference count plus plus for the object star v; that is this particular object now is star v; whatever is pointed to by v will also be pointed to by u now. v was pointing to something; now u also points to it. Now the object star v has one more reference. So, reference count plus plus. For u, it was pointing to something; from now on it points to what v points to. So, for the object which was pointed to by u, it is reference count minus minus object star u.

Procedure returns: For returns, reference count minus minus for each object pointed to by the local variables. This is what I was saying. Once the procedure returns, all the local variables which are pointing to some objects are not available any more. Reachability reduces; so, reference count minus minus for each object pointed to by the local variables.

Transitive loss of reachability: This is what I showed just now. Whenever reference count of an object becomes zero, we must also decrement the reference count of each object pointed to by a reference within the object.

(Refer Slide Time: 37:14)



This is what it is. These 2 fields are pointing to this node and this node. The reference count of A has become 0. We are going to actually reduce the reference count of this by one. So, this becomes 0; this also becomes 0. Finally, this is pointing to this and this is pointing to this. Both of them decrement this reference count by one; so, this also becomes 0; that is what I have showed just now; finally, D is also collected.

(Refer Slide Time: 37:43)

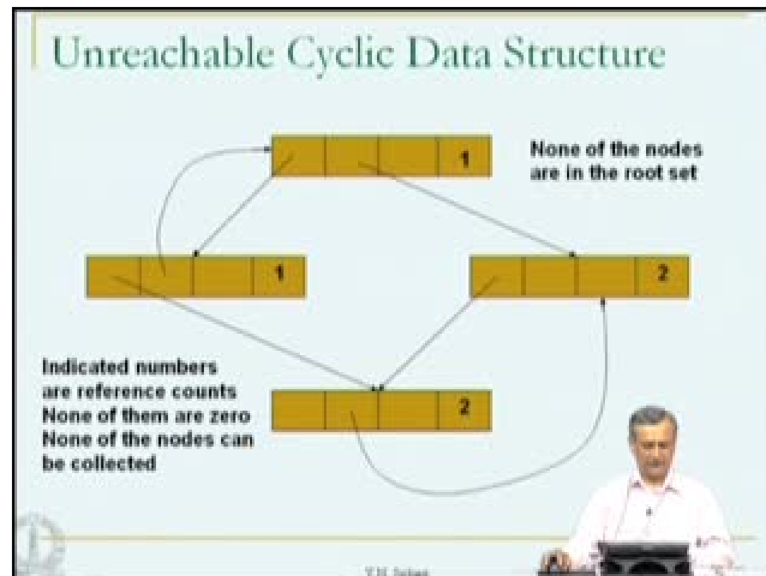
Reference Counting GC:
Advantages and Disadvantages

- High overhead due to reference maintenance
- Cannot collect unreachable cyclic data structures (ex: circularly linked lists), since the reference counts never become zero
- Garbage collection is incremental
  - overheads are distributed to the mutator's operations and are spread out throughout the life time of the mutator
- Garbage is collected immediately and hence space usage is low
- Useful for real-time and interactive applicati where long and sudden pauses are unac

What are the advantages and disadvantages of the reference counting type of garbage collector? The red ones are all disadvantages and the blue ones are all advantages. There

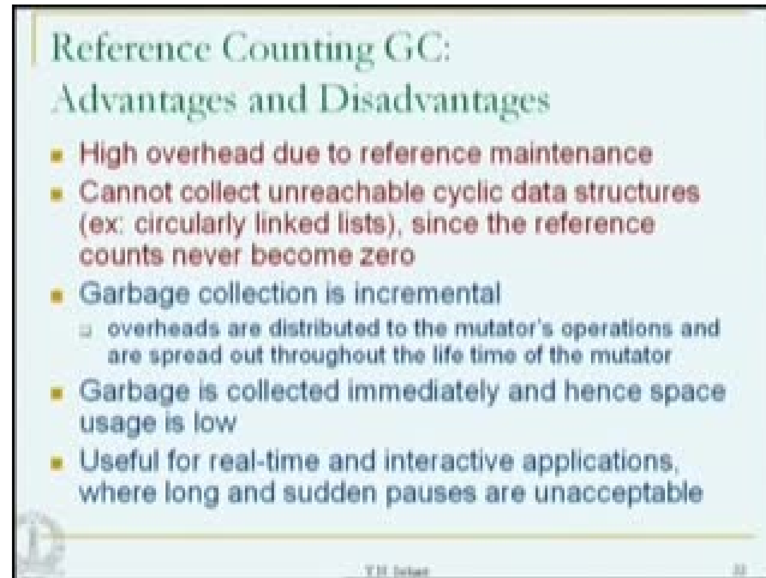
is very high overhead due to reference maintenance. For every assignment statement involving pointers, you do a plus plus or minus minus etcetera. So, there is a lot of overhead due to reference maintenance. You have to keep manipulating the reference counter and you cannot collect unreachable cyclic data structures, circularly linked lists I will show an example of this, since the reference counts never become zero.

(Refer Slide Time: 38:27)



The same old picture, this pointing to this, this pointing to this and this pointing to this, but I have also added cyclic part here. So, this points to this and this points to this. This counter is still one because this is the pointer pointing to this; this field is pointing to this particular node. This counter is one because this field is pointing to this node. This counter is two because this is pointing to it and this is also pointing to it. This is two because these two are pointing to it. None of the nodes are in the root set. No programmer defined variable is pointing to any of these, but because of the cyclic nature of this particular data structure none of these reference counts are zero. Therefore, none of the nodes can be garbage collected so this is the problem.

(Refer Slide Time: 39:27)



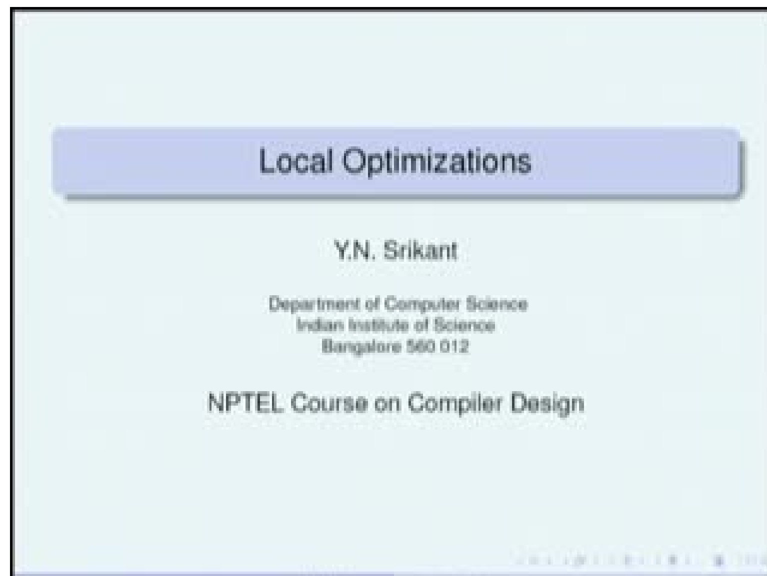
What are the advantages? Garbage collection is incremental; overheads are distributed to the mutator or program's operations and are spread out throughout the life time of the mutator. If you have run any java program and when it runs out of memory; the java program just pauses. You try to do anything, click a mouse or something like that, click a button, nothing works.

This is because the java program is now doing garbage collection. This is the pause that I was talking about. In the reference counting type of garbage collection, there is no such pause. Overheads of reference counting are distributed slowly gradually. The memory becomes unreachable and as soon as it becomes unreachable, it is returned; that is reference count becomes zero, the memory chunk is returned to the heap.

So, it is incremental and it is very fast. Garbage is collected immediately and hence space usage is low. Imagine you had a large chunk of memory, you went on allocating it and no collection was made. Only when the entire memory was consumed, the garbage collector ran and then it collected and went back to the program. So, here, you probably need much more memory even though some parts of it became free in the middle because you are not really collecting such free memory somewhere in the middle. Whereas with reference counting type of garbage collection garbage is collected immediately, once the reference count becomes zero and hence you can probably do with less amount of heap memory.

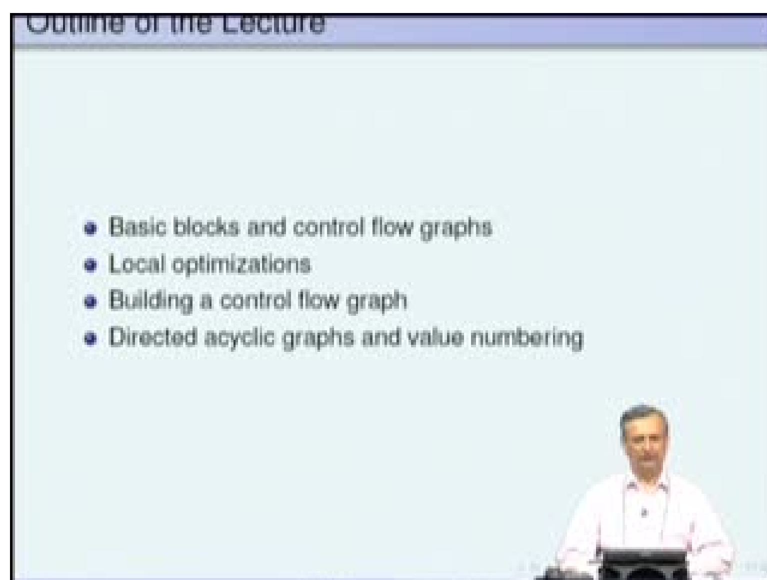
Reference counting method is very useful for real time and interactive applications where long and sudden pauses are unacceptable. This is what I just now told you java program introduce such pauses and these may be unacceptable to us. Therefore, this reference counting GC even with its limitations has its place in real time and interactive applications. That is the end of the lecture on run time environments.

(Refer Slide Time: 42:09)



Now, we begin the lecture on local optimizations.

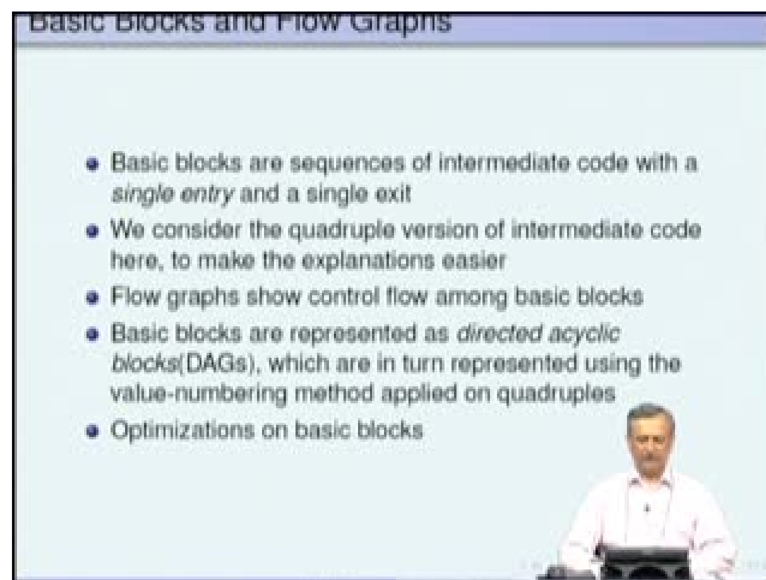
(Refer Slide Time: 42:14)



The outline of the lecture is we study and understand: what are the basic blocks, what are control flow graphs, some of the local optimizations, how to build a control flow graph, directed acyclic graphs, value numbering and things like that. Why are all these very important? We know that the intermediate code generator produces intermediate code in the form of quadruples or trees and so on.

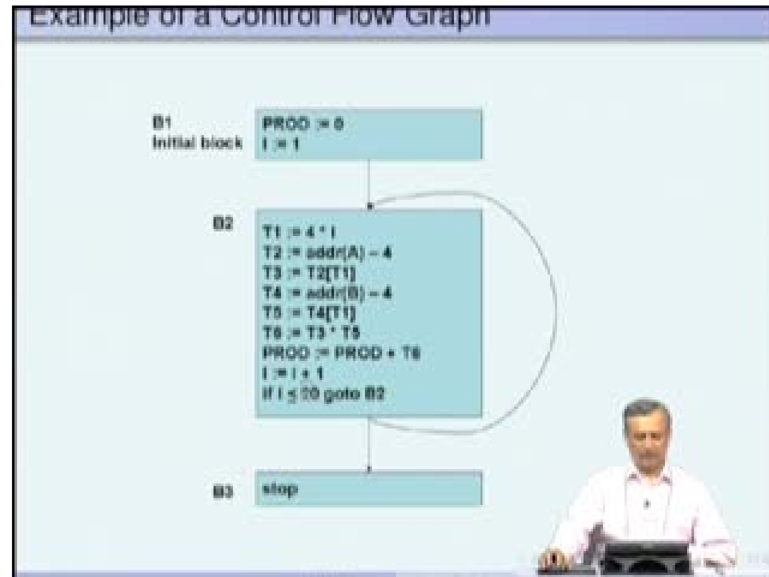
Now because of the control instructions, jump instructions in the program the sequence of quadruples can be actually partitioned into smaller blocks and a graph structure can be accorded to them. These blocks are called basic blocks and the graph is called as a control flow graph. This basic block plus control flow graph structure is very useful in optimizations. We will see how they are done and they are also useful for machine code generation that will be studied sometime later.

(Refer Slide Time: 43:35)



What are basic blocks? Basic blocks are sequences of code with a single entry and a single exit.

(Refer Slide Time: 43:57)



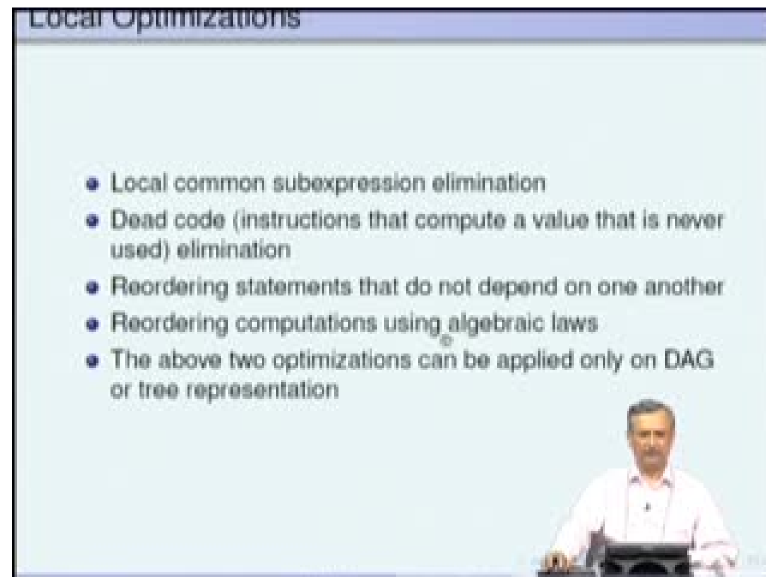
Let me give you an example. There is only a straight line sequence within such basic blocks. You cannot quit until you come to the end of the basic blocks. Take this big block; there is no jump instruction anywhere in the middle of this particular block. It is only at the end. This type of sequence of quadruple is known as a basic blocks.

So, they have a single entry and a single exit. We consider the quadruple version of intermediate code so that explanations are easier, but with some modification we can also process tree intermediate code. Flow graphs actually show the control of flow between these basic blocks and basic blocks are represented as directed acyclic graph. I will show you some examples of this which are in turn represented using the value numbering method which is applied on quadruples. We are going to all these processes along with optimizations on basic blocks.

This is an example of a control flow graph. If you look at the sequence of quadruples alone, you would have PROD equal to zero, I equal to one then T1 equal to 4 star I etcetera. Then we have if I less than equal to 20 go to B2, instead of B2 it would be quadruple number three. This is number one, this is number two, this is number three and so on and so forth. Finally, this will also become number three and following that would

be the stop. Each of these is a basic block and the control flow is between these basic blocks.

(Refer Slide Time: 45:54)



What are the local optimizations that we can perform? Optimization which are performed within a basic block are called local optimizations. We are not going to move any quadruples from the basic block to some other basic block, when we perform local optimizations. We move quadruples within the basic block or we delete quadruples within the basic block or change them within the basic block, but nothing leaves the basic block and goes to some other basic block. If it goes to some other basic block and there is code movement across basic blocks, that would be a global optimization and that it would be the subject, a little time later.

Local common subexpression elimination is one of the very important optimizations. As the name suggests if there are many expressions a equal to b plus c, x is equal to b plus c etcetera within the same basic block, b plus c is a common subexpression.

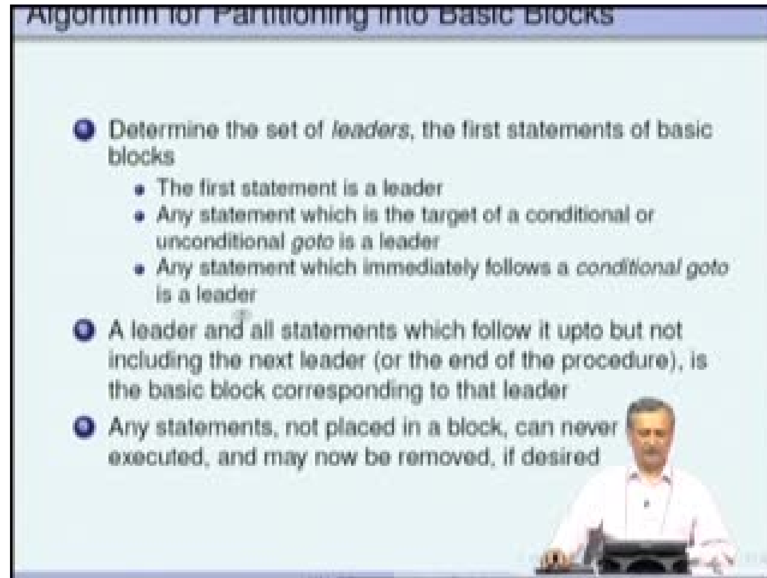
There is no need to compute this subexpression again and again under certain conditions that is, b and c are not modified. Then it is also possible to do dead code elimination. What is dead code? They are instructions that compute a value that is never used. It is possible that we just use another extra variable, assign it a value and the variable is never used in the program.

How does us such a thing happen? It may not be intentional and it may not be done by the programmer the first time he or she writes a program. When the program is being debugged, when the program is being changed it is possible that some variables are introduced and some are not used anymore and the programmer forgets to delete the declaration of such variables or even the assignments of such variables. It is also possible that some code is introduced into the program for debugging purposes, but it does not actually contribute to the program performance alone.

Reordering statements that do not depend on one another: Such reordering may actually increase the speed of the program. For example, if a load can be done early some other load can also be initiated at the same time. These loads may not depend on each other. So, we may want to move them appropriately so that some delays are taken care off.

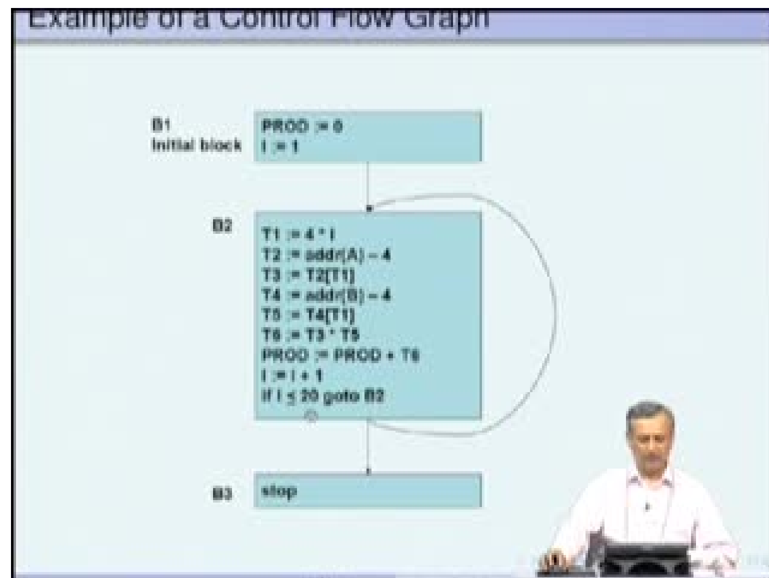
Reordering computations using algebraic laws: Algebraic laws are commutativity and associativity. We will see how this can be used later on. For example, to give you a simple example if v plus c is an expression, c plus b is another expression and if commutativity holds, these two are equivalent. If you can detect such computations using commutativity or any other algebraic law then the code can be improved. The above two optimizations can be applied only on direct acyclic graph or tree representation - that is the reordering of statements that do not depend on one another and reordering computations using algebraic laws. We will see how to do this using value numbering as well; one of these can be done is value numbering.

(Refer Slide Time: 49:32)



How do you partition the quadruples into basic blocks? That is the next topic. Algorithm is simple; essentially we determine the set of leaders. What are leaders? The leaders are the first statements of basic blocks. The first statement in the program that is a procedure is a leader; any statement which is the target of a conditional or unconditional goto is a leader. I will give you an example of this and any statement which immediately follows a conditional goto is also a leader.

(Refer Slide Time: 50:14)

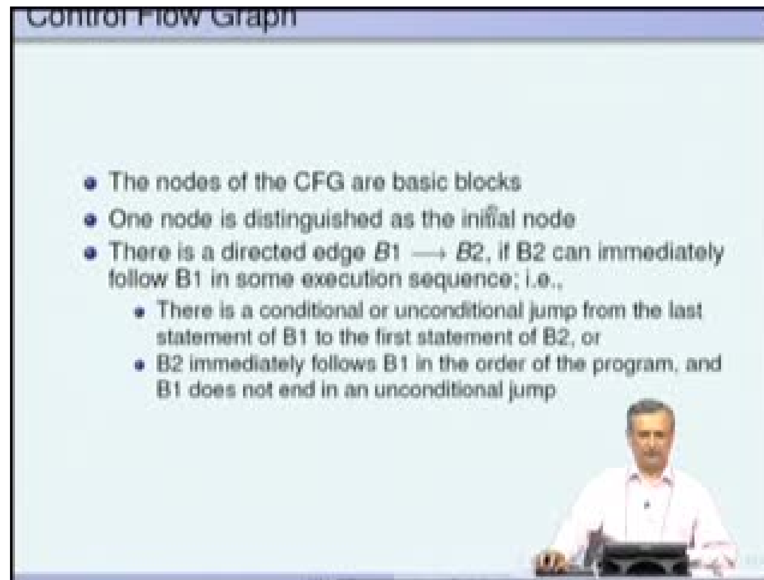


Look at this example. We have PROD equal to zero, then we have I equal to zero, then we had T1 equal to 4 star I etcetera. Finally, read this B2 as three - number three because it is going to three. At this point when we scan this we find that if I is less than 20, go to three. That means three becomes a leader because it is the target of a conditional goto statements. This statement, the stop statement immediately following this is also a leader simply because it is following this conditional goto and it is not after an unconditional goto.

So, any statement which immediately follows a conditional goto is also a leader. A leader and all statements following it up to, but not including the next leader is the basic block corresponding to that leader.

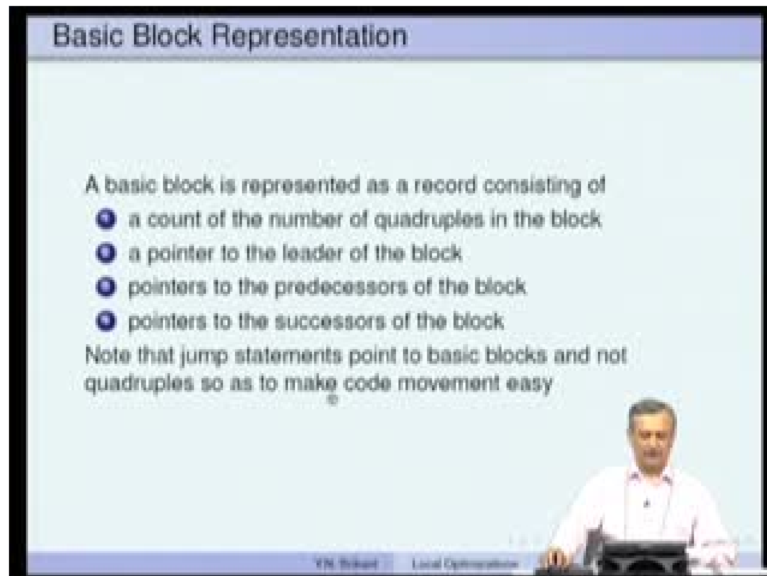
(Refer Slide Time: 51:08) For example here, we marked this as a leader, we marked this as a leader and we marked this as a leader. Start scanning from here up to this point that is, these two statements that is one block, from here which is a leader to this point just before that is another basic block and this stop statement is a basic block on its own. A leader and all statements following it up to but not including the next leader is a basic block corresponding to that leader. Any statements, not placed in a block, can never executed and may now be removed if you want to.

(Refer Slide Time: 51:53)



What is a control flow graph? The nodes of the control flow graph are basic blocks. We saw this already. One node is distinguished as the initial node; (Refer Slide Time: 52:03) here this is the initial node. There is a directed edge from B1 to B2, if B2 can immediately follow B1 in some execution sequence. How do you determine this? There is a conditional or unconditional jump from the last statement of B1 to the first statement of B2. Let us do this. There is conditional or unconditional jump from the last statement of B1 to the first statement of B2. For example here, last statement of this particular block to the first statement of the same block, it could have been a different block; this is the arc. Here is a control flow and this corresponds to an arc. B2 immediately follows B1 in the order of the program and B1 does not end in an unconditional jump statement. Here the stop statement follows this and this is a conditional jump and not an unconditional jump. This is another arc which is added to the control flow graph.

(Refer Slide Time: 53:15)

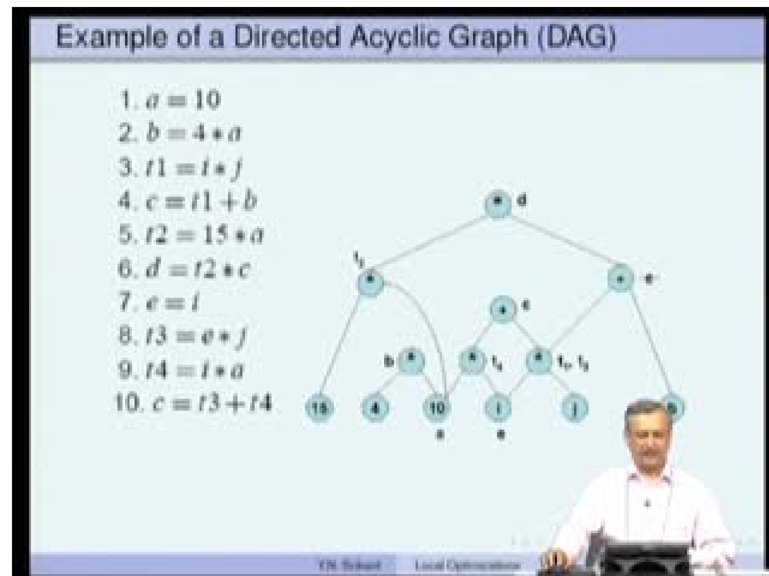


How do we represent basic blocks? A basic block is represented as a record. What does the record consist of? A count of the number of quadruples in the block, a pointer to the leader of the block; these are essential, pointers to the predecessors of the block; this is the representation for the graph structure and pointers to the successors of the block; again, this is a representation for the graph structure of the control flow graph. We need these things when we want to traverse the control flow graph in a particular order. We may want to do a first search on the control flow graph or we want to do a DFS on the reverse flow graph. So, all these become useful at that point.

Note that jump statements point to the basic blocks and not quadruples so as to make code movement easy. Let us see what happens here. As I said this if I less than equal to 20, goto, it says B2 and it does not say three. Suppose it had said three and because of some optimizations not necessarily local, it could have been global also, we moved some of the statements such as T2 is equal to address a minus 4 and T4 equal to address b minus 4 into block one; these two move here. So, then block one will have four quadruples and T1 equal to 4 star I will become the fifth quadruple. If we had written I less than 20 goto three, it would have been an error. We had to probably go and modify this as well. So, what we do is write this as B2 which implies a pointer to the beginning of the block there by any changes in the block, the movement of quadruples even to the outside will not affect it. We always know what the first quadruple in the block is. That would be T1 equal to 4 star I even if these are moved even if T1 equal to 4 star I is

removed from the block, there would be some other quadruple. It does not matter which quadruple is the first. We only want to make sure that this pointer is to the beginning of the quadruple. That is what this is saying. So, it makes code movement easy and then it is not necessary to patch code whenever we move some quadruple outside the basic block.

(Refer Slide Time: 55:42)



We will stop at this point and in the next part of the lecture, we will discuss the directed acyclic graph presentation of a basic block and how to perform various optimizations on the basic block. Thank you very much.