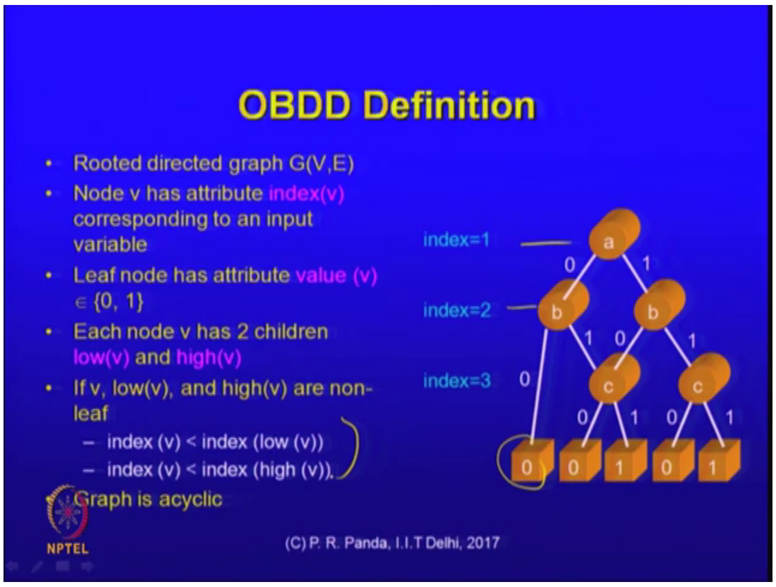
Synthesis of Digital Systems
Dr. Preeti Ranjan Panda
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Lecture – 20
Binary Decision Diagrams

So, reviewing the OBDD definition.

(Refer Slide Time: 00:21)



OBDD Definition

- Rooted directed graph G(V,E)
- Node v has attribute index(v) corresponding to an input variable
- Leaf node has attribute value (v) ∈ {0, 1}
- Each node v has 2 children low(v) and high(v)
- If v, low(v), and high(v) are non-leaf
  - index (v) < index (low (v))
  - index (v) < index (high (v)).
- Graph is acyclic

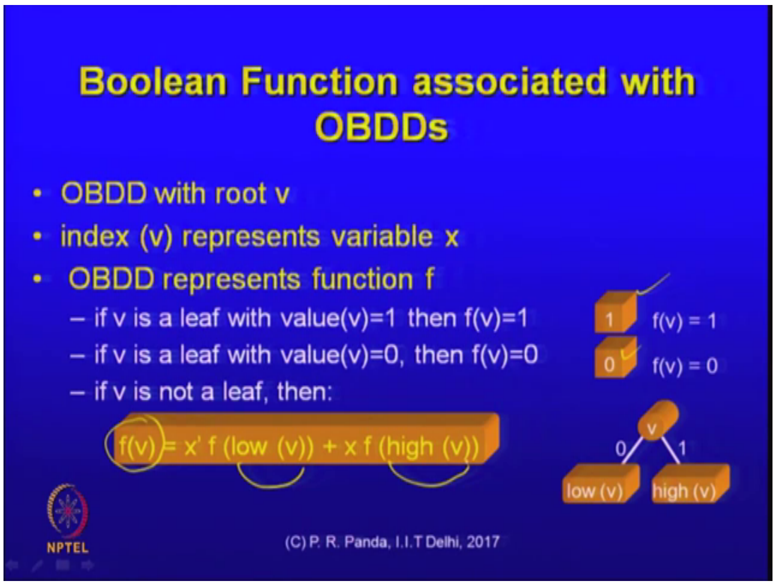
(C) P. R. Panda, I.I.T Delhi, 2017

index=1
index=2
index=3
a
b
b
c
c
0
1
0
1
0
1
0
1
0
1
0
0
1
0
1

This is a binary decision diagram in which every node is associated with a variable of course, but we are doing an ordering of those variables. So, there is an index number that is associated with all the variables and whichever is associated with the node that index is now associated.

So, the height is in the order of the indices or we start with the root being at index 1 and the next level being at index 2 and so on ok. So, there is an index value associated with every node, the leaf node has the function values being 0 and 1, where the true and false this is a binary diagram which means that each node has 2 children, one is low and the other is high. This relationship holds which is the index of a parent is less than the index of its children, so because of that property the graph is acyclic.

(Refer Slide Time: 01:32)



Boolean Function associated with OBDDs

- OBDD with root v
- index (v) represents variable x
- OBDD represents function f
  - if v is a leaf with value(v)=1 then f(v)=1
  - if v is a leaf with value(v)=0, then f(v)=0
  - if v is not a leaf, then:

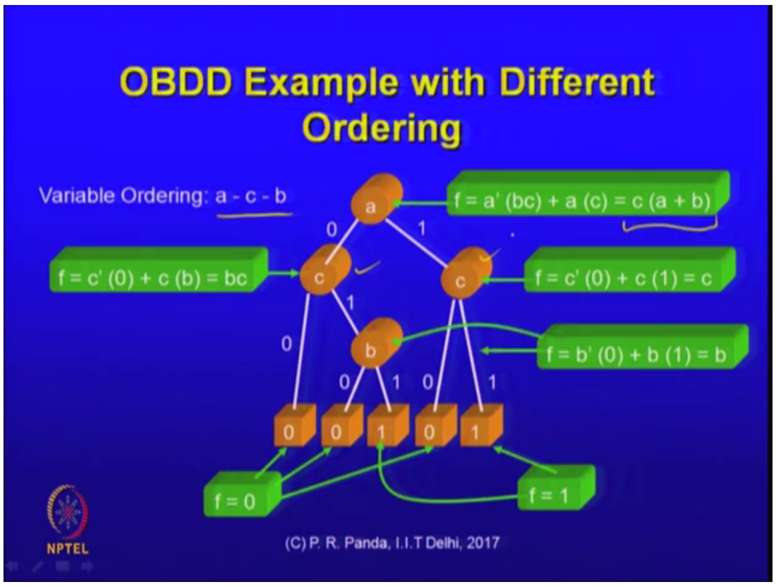
$f(v) = x' f(\text{low}(v)) + x f(\text{high}(v))$

1
f(v) = 1
0
f(v) = 0
v
0
1
low (v)
high (v)
NPTEL

(C) P. R. Panda, I.I.T Delhi, 2017

And there is a Boolean function that is associated with these OBDDs; it is inductively defined as follows if it is a leaf node then that function is true or false depending on whether value is 1 or 0. If it is a non-leaf node then that Boolean function corresponding to the node v is composed in terms of the function associated with the low child and the high child, so that is all that is there as far as the definition is concerned.

(Refer Slide Time: 02:05)



OBDD Example with Different Ordering

Variable Ordering: a - c - b

(C) P. R. Panda, I.I.T Delhi, 2017

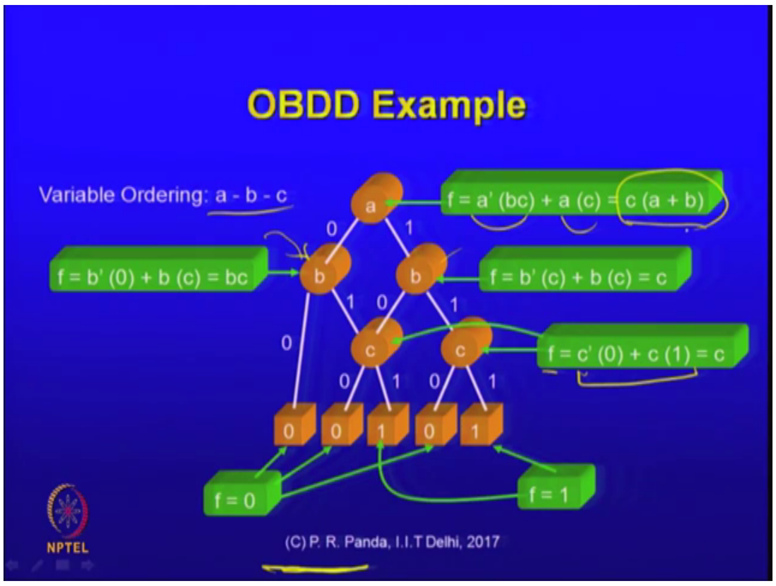
We started off with one example, where you have some function like that c equals a plus b, but my ordering is a b c. So, I start off with a. Remember the function ultimately is that I should be able to express this as a prime time something plus a times something, but we already showed that any function can be decomposed in that way; Shannon's expansion can be used to do that decomposition.

So, once I do that this is my function, but I have a prime times b c plus a times c. So, somehow that node has to be associated with the function b c and that node has to be associated with the function c. So, we had gone through a traversal of this B D D structure, at the end there is a Boolean function that is associated with every node in the diagram ok.

So, and that function is composed using that simple rule. So, the function is 0 for all the leaf nodes and as you go up the levels, you would compose more complex functions based on the values of the children. Just to illustrate that this is not unique first of all there is a dependence on the variable ordering, if you change that ordering to a c b instead of a b c, for that same function this is the same function that we started off with, you would have a very different diagram that should be obvious everything changes the diagram itself completely changes if the second level is c instead of b, because those lower level functions would decompose differently. Now that I have picked up and expressed things in terms of c first rather than b first, in general you expect a different structure.

So, the way this evaluates to c times a plus b is these are still zeros and once at the leaf level, you have a node with a b here 0 leading to 0 and 1 leading to 1 meaning that the function is b prime time's 0 plus b times 1, so that is b. This with 0 branch leading to 0 1 branch leading to 1 is essentially c this 1 with the c label here means that if c is 0 then the value is 0, if c is 1 then the value is b and my function then is b c this is b c and this is c the function at the root is a prime time's b c plus a times c which gives me c times a plus b it is the same function, but the structure is different here.

(Refer Slide Time: 05:13)

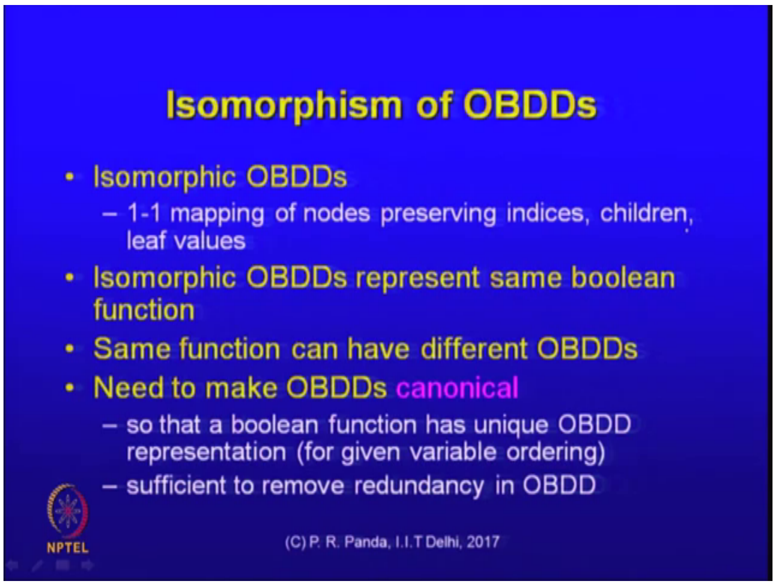

OBDD Example
Variable Ordering: a - b - c
f = a' (bc) + a (c) = c (a + b)
f = b' (0) + b (c) = bc
f = b' (c) + b (c) = c
f = c' (0) + c (1) = c
f = 0
f = 1
(C) P. R. Panda, I.I.T Delhi, 2017

Student: Each different Shannon's canonical from itself this is not yet canonical, we will have to do something to make it canonical we are moving in the direction of a canonical representation. But this is not yet canonical you can see why it is not canonical it is because that same function, could be written in some other way.

If you wrote it like this, then this is the way to decompose it and this is your OBDD you get. But you could write their same Boolean function in some other way, we did not do any optimization here there is no minimization right this is I am just taking the algebraic expression that was given to me and just doing a straightforward decomposition. So, there is no optimization here, but equivalently you could have written some other function for example, involving a prime and so on.

Which is the same function, but the expression is different you would end up with a different tree you can easily see that by working out two different Boolean representations of the same function. For example of course, one default representation is you just list out all the min terms, that word in general lead to a different structure yeah ok.

(Refer Slide Time: 06:35)



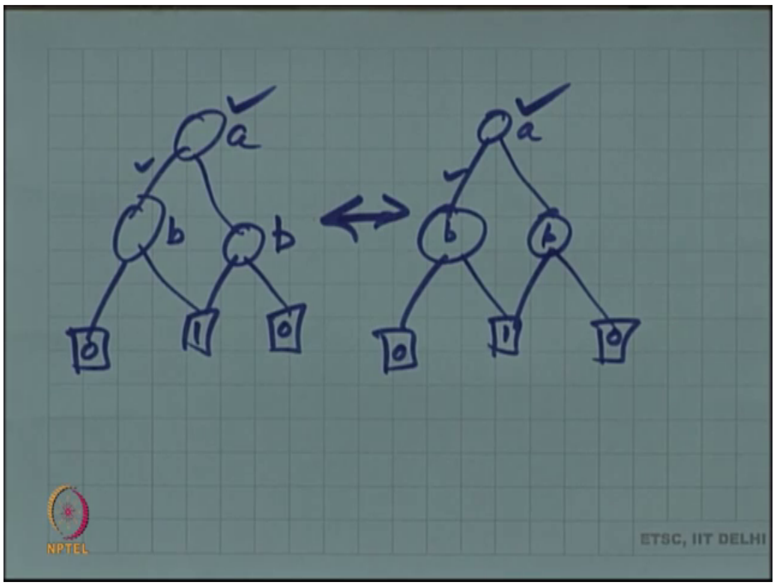
Isomorphism of OBDDs

- Isomorphic OBDDs
  - 1-1 mapping of nodes preserving indices, children, leaf values
- Isomorphic OBDDs represent same boolean function
- Same function can have different OBDDs
- Need to make OBDDs canonical
  - so that a boolean function has unique OBDD representation (for given variable ordering)
  - sufficient to remove redundancy in OBDD

(C) P. R. Panda, I.I.T Delhi, 2017

So, first of all if the ordering changes then the structure changes, now let us look at some properties of these OBDDs. 2 OBDDs are said to be isomorphic, if you can find a 1 to 1 mapping between their nodes that preserves the indices the children and leaf values everything. So, if I have something like that this is a b b and this is 0 1 0 or some such.

(Refer Slide Time: 07:03)

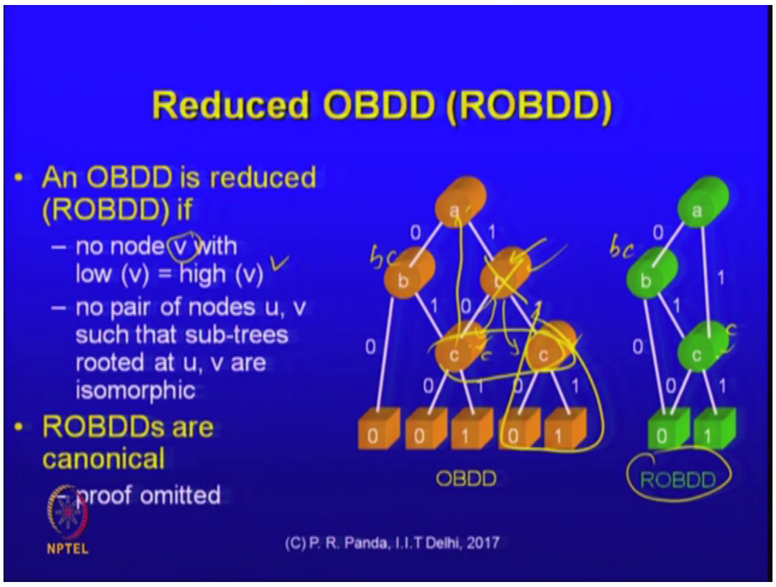


So, to such OBDDs would be called isomorphic, if I can find an association these are graphs remember in the general case. If I can find an association between these structures so, that for every node I find a corresponding node for every edge I find a corresponding edge with the same label of course, leaf node similarly right if they are identical not just structurally, but also in terms of all the annotations that we have made they are with respect to the labels I should have an a here if this was an a here, I should have b if b there and the zeroes and one's everything should match, then we say that these 2 are isomorphic.

Isomorphic OBDDs of course, would represent the same function, because there is a function that is associated with every node. So, if you have established that the nodes are identical, they have the same value and everything that is rooted at that node those are also identical then the functions are also identical. However, the same function can still have different OBDDs there is a need for us to make these OBDDs canonical it is not yet canonical because the same function can have different OBDDs we will see with an example

There is some advantage of this canonical representation, it is that if it has this property then a Boolean function has a unique OBDD representation for that given variable ordering if variable ordering changes then canonicity is still not guaranteed in these data structures right. How do you make them canonical this is an interesting exercise? If you remove redundancy in the OBDD then we argue that you have a canonical representation.

(Refer Slide Time: 09:25)



Reduced OBDD (ROBDD)

- An OBDD is reduced (ROBDD) if
  - no node v with low (v) = high (v)
  - no pair of nodes u, v such that sub-trees rooted at u, v are isomorphic
- ROBDDs are canonical
  - proof omitted

a
0
1
bc
b
c
0
1
c
c
0
1
0
1
0
0
1
0
1
OBDD
a
bc
0
b
1
c
0
1
0
1
0
1
ROBDD

(C) P. R. Panda, I.I.T Delhi, 2017

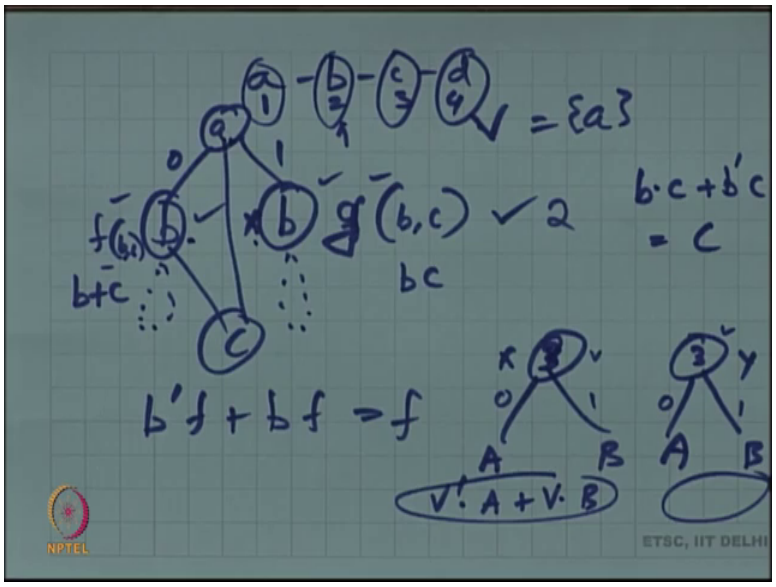
So, let us see what kind of redundancy could be there in an OBDD, an OBDD is called a reduced OBDD that is a reduced ordered binary decision diagram. So, mouthful, but there are other extensions also to this is still intermediate in complexity as far as pronunciation is concerned.

It does not help ROBDD takes the same amount of time to pronounce as a reduced ordered binary decision diagram, but writing it is easier. So, let us define what is a reduced OBDD, we start off with an OBDD and we say that it is reduced if a couple of properties are held essentially we are saying that we will remove redundancy, what kind of redundancy could be there in a typical OBDD this structure here can you identify any redundancy.

Student: Yes both the child of a or b.

Both the children of a being labelled b does not constitute their redundancy. In fact, that is the way you expect it to be.

(Refer Slide Time: 10:43)



Right you have a and 2 b children are there, but this is some general function right this function is different, let us say this is a function of b and c this function is some other function of b and c, they would both look like this at that level they would be different at the lower levels, but if you have let us say b plus c here and you have b c here both of them will have b at the root they are distinguished by what appears below them, but this by itself is not a redundancy.

If there are other things that come below it because these functions could be different in fact, this is how you expect that OBDD structure to be this is not by itself, but there is some redundancy in this diagram.

Student: Last c.

Yes what is the function associated with this node with with that node.

Student: C is same.

The function is c right there is only 1 rule for evaluating it, you look at the 2 children functions and you compose the parent function these 2 are isomorphic the sub graphs that are rooted at the cs are isomorphic and therefore, the functions are also the same this is c prime 0 plus c times 1 which is c.

Student: C

This is also c.

Student: Same.

So, there is some redundancy there what other kind of redundancy.

Student: (Refer Time: 12:22) 1 of that b on the graph

Yes there are 2 kinds of a if these 2 are the same what does it tell us about b.

Student: It is not there.

It means that irrespective of whether the value of b is 0 or 1 the function here is the same.

Student: Same.

It means that it is independent of b, in fact b should not even appear yeah the reason it appeared there is that that is the redundancy, but if you insist on finding the cofactors with respect to b what will happen the same function will appear as both the cofactor with respect to band also with respect to b prime and. So, you will end up with a situation like this it is not invalid the structure is fine, but this is not the minimal structure.

Student: What would happen if we have only a to c direct connection.

Yeah right. So, I essentially need to get rid of this first of all I need to get rid of one of these sub trees because it is not necessarily they both represent the same function anyway right, but after doing that after noticing that both the branches from b are leading to the same function, we can get rid of b and we just replace it by that structure that is all that is there. So, an OBDD we call it reduced if you don't have a node with this low of v is equal to high of, which is which example out of the 2 redundancies we identified this property is which of.

Student: This is the bone.

Of the b 1.

Student: B 1.

Right because the low function is the same as the high function. So, such a node should not be there in a reduced OBDD that is one other property is there shouldn't be a pair of nodes u and v such that these sub trees that are rooted at u and v are isomorphic that is which example.

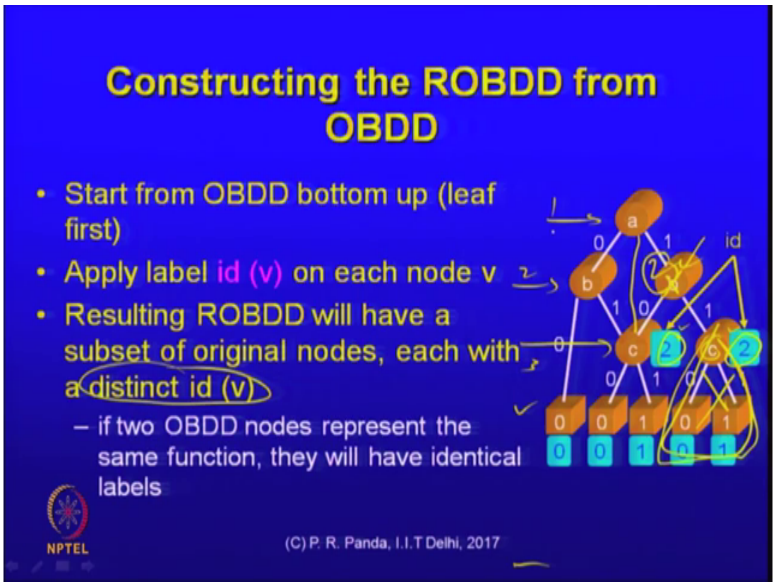
Student: C.

That is the c example that we said there is duplication there and there is duplication I would like to get rid of one of them, these are the two redundancies if I remove them then you have essentially one copy of this of this function b itself over there is redundant. So, I remove that the other b still remains, but this b is removed and I have this structure that is called a reduced OBDD.

Student: A b plus a c or a prime b prime a c plus a prime b c.

This was b c right this function was b c and that function is c, so that is b c and that is c. So, I have a prime b c plus a c this r OBDD structure is a canonical representation. Why that is the case we are omitting the proof for this discussion, but all that we need to do is get rid of those 2 kinds of redundancies from an OBDD and you would have ROBDD.

(Refer Slide Time: 15:20)



Constructing the ROBDD from OBDD

- Start from OBDD bottom up (leaf first)
- Apply label id (v) on each node v
- Resulting ROBDD will have a subset of original nodes, each with a distinct id (v)
  - if two OBDD nodes represent the same function, they will have identical labels

(C) P. R. Panda, I.I.T Delhi, 2017

a
b
c
c
id
0
1
2
2

Since it is an important structure let us try constructing a reduced OBDD from an OBDD. We will later on move to a generic algorithm for doing so, but let us look at what the strategy would be you start from the OBDD bottom up right from the leaf levels right. You apply a label called an i d to each of these nodes for every distinct function that we come across we will identify a new label. So, initially would have just the zeros and ones here, but for these two nodes I would give the same level because they are essentially the same.

Student: Same.

Function right, so I would apply a label, but that my traversal is from the bottom up. So, these labels are also getting applied in a bottom up way right in the process at the end that result in ROBDD structure will have a subset of the original nodes. Why because two node with the same label means that one of those entire sub trees rooted at that node can disappear.

Student: Sir when we are applying the same label we are only seeing the parents to a work, only the children right.

Strident: Not the parent.

Right remember that is how the Boolean function for that node is defined, it is defined only in terms of its children its parents have nothing to do with that Boolean function itself. So, two labels essentially correspond to the same Boolean function and I would keep one of them. Any way what we are driving at is that the ROBDD is a subset, in terms of nodes at least it is a subset of the original OBDD, because we are getting rid of some of the nodes and edges what we are retaining are the nodes with distinct labels distinct ids.

If there is a repeated id then we would get rid of that we would keep only one of them and of course, if we keep one of those nodes if we remove one of those nodes it means that we remove everything that is under it unless it is overlapping with some other parts, essentially what it would translate to is really remove the node also remove its associated edges its children edges at least that much you can remove. So, if that child does not have any other edge leading to it, then that child also gets remove from that.

So, a point is you retain a subset of the original node some stitching up of the structure is needed because in the process of getting rid of nodes, you would end up with sometimes a node being redundant ok. So, if this label is 2 and that label is also 2 then what would be the label for the parent? We want to invent a new label in this labelling process when we come across a new function right that has not been encountered yet, so far.

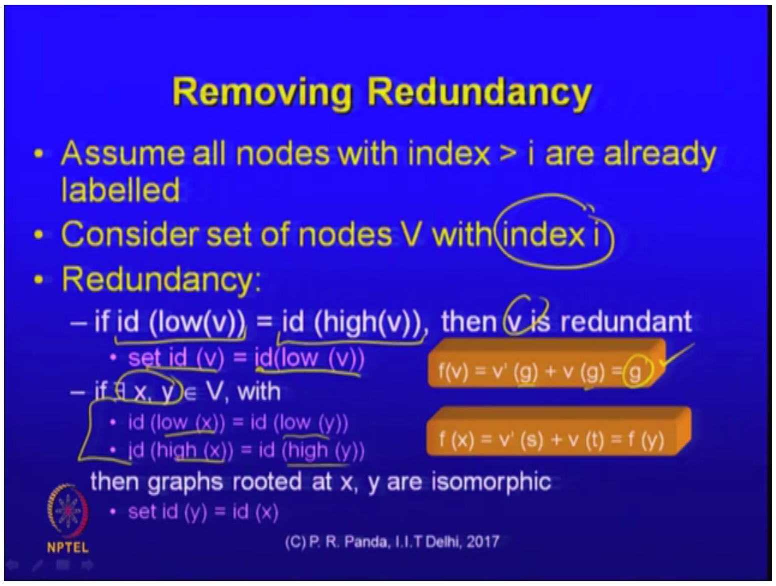
Student: (Refer Time: 18:37).

So, I should apply 2 here because the value of that function is this Boolean function associated with label 2 irrespective of whether b is 0 or 1 right. So, it is the same Boolean function that should be propagated to the parent because what you have is b prime times whatever that function is.

Student: Plus b.

Plus b times that same function right, so this is nothing, but f. So, I should propagate that same label up, at the end I would have a structure that has a subset of all of these nodes I would drop some nodes, but in the process I would get rid of some of the edges and some stitching up has to be needed because now I have to introduce a new edge if I got rid of this, and I retained this, I got rid of this node then what happens to this edge right I would have to reroute it to that node. So, that adjustment also I will have to make.

(Refer Slide Time: 19:35)



Removing Redundancy

- Assume all nodes with index > i are already labelled
- Consider set of nodes V with index i
- Redundancy:
  - if id (low(v)) = id (high(v)), then v is redundant
    - set id (v) = id(low (v))
  - if ∃ x, y ∈ V, with
    - id (low (x)) = id (low (y))
    - id (high (x)) = id (high (y))

    then graphs rooted at x, y are isomorphic
    - set id (y) = id (x)

f(v) = v' (g) + v (g) = g

f (x) = v' (s) + v (t) = f (y)

(C) P. R. Panda, I.I.T Delhi, 2017

So this process of removing redundancy assume that nodes with index greater than I are already labelled, we are moving in a bottom up fashion anyway. So, the higher indices would be processed first.

Student: First.

Right if you consider a set of nodes v with index I what are those at this level here there are 2 nodes. So, here v equals a or at least that node here v would be the set of these 2 nodes. Actually the structure can grow exponentially large, as you go to the lower levels you may have more and more nodes in the general case right. So, you collect all of those nodes, so redundancy removal corresponds to essentially this kind of a check, I say if the label for the low is same as the label for the high child then v is redundant we are left with this kind of a function that is v prime time's g plus.

So, g is that function and we essentially want to make g the value of the parent node also. So, how do you do that essentially in that traversal you just set the node of v to set the label for v to be the same as the label for 1 of its children because the same function right. So, that is 1 redundancy, that is when that node itself is redundant other is if you find 2 nodes belonging to, we are checking those nodes that all have the same index we are talking about nodes with the same index it these are all nodes at the same level of the.

Student: OBDD.

Structure yeah, so if you find a pair of them in which the low of x is the same as low of y and the high of x is the same as high of y, the same as means we are just comparing the label numbers these are in integer labels that we are assigning and remember a new number is assigned if a new function is arrived at, so if I can find this what does this mean.

Student: Isomorphic no.

It means that the children are identical children functions are identical. So, I found two nodes x and y actually the nodes are called x and y the indices are actually the same if the index was 3 here the index is also ah.

Student: 3 here.

Three here meaning that it is the same variable right and this has labels a if this left labels match, and the right labels also match what does it mean the expansion would be the same right for both of these, whatever that variable is that is common and it is that variable. So, it is a v is that variable then you have this situation of v prime times a plus v times b in both the cases.

Student: Sir just the thing we said that for the same label we have for the same level of the graph we have the same variable.

Right.

Student: This is because we are considering the OBDD as of now.

Yes yes.

Student: If the ROBDD this is not the case the same level we can have different variables.

We still have the same ordering if I get rid of a node then this is still the third level this is still the second this is still the first.

Student: But sir when we have removed that node then that c will come at the second level no we will not.

We are constructing the ROBDD, we aren't yet the ROBDD yet we are starting with the OBDD only right.

Student: That is what I am saying.

Right.

Student: That this condition is true for the OBDD.

Yeah yeah.

Student: When we have reduced it to the ROBDD this condition is no longer true then that at the same level we have the same variables.

Same variables could still be there at the same because the functions could be different let us say. So, far I didn't say its reduced or not, but in a way it is the reduced form even in that form if that my ordering was a b c d then I would have the route like this and this would be some function of b c d that would be some other function of b c d b would still appear in both of them.

Student: Ordering levels (Refer Time: 24:12).

Ordering is you start with a and then you have a b, so just because these 2 nodes have the same label does not mean that the function is the same or that there is redundancy right, because this is some function of b c this is some other function of b c.

Student: Sir what is in the reduction process we get rid of one of the b s in the branch.

That is fine the ordering is here right 1 2 3 4 that is my index right b is still associated with index 2, which means that it appears in the second level.

If that other node does not appear in the reduced form that is all right this goes to the third level. The level is still the same the meaning of index an level these are still the same it is just that the number of nodes at that level may go down as a result of any redundancies that we remove.

Student: Is it possible that a complete level can go wrong.

It is possible.

Student: Possible.

Right you have a function of a b c d, but it turns out that it is actually independent of b if you have b times c somewhere plus b prime time's c, then this becomes c and the b level completely goes away that is all right the index according to our original function and its representation remains the same, but 1 level goes off and that is fine.

Student: The all the node level was combine level up and still that.

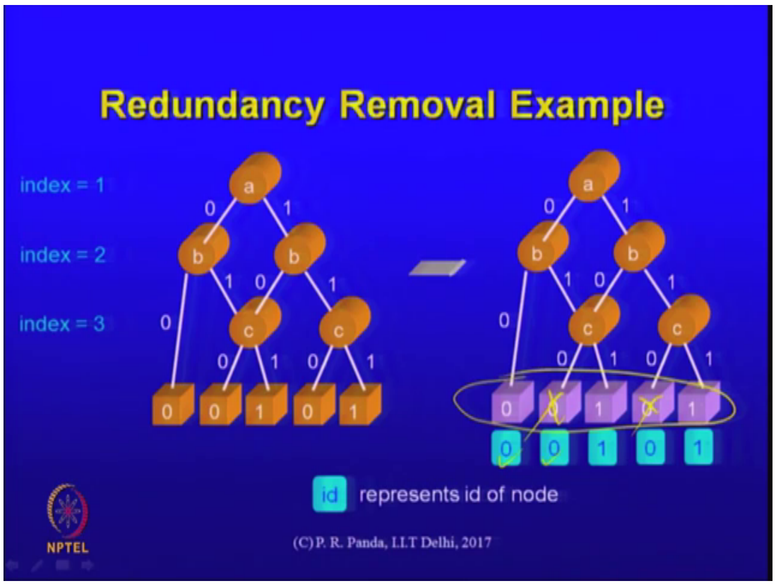
This is a graph structure the level is only an association of a node with an index you have a data structure in which you started off with 1 2 3 4, but 2 is not there anymore that is all right.

Student: It is all right.

Right this level anyway is only a visual thing, but what binds these is the assignment of that integer called index, you still have the property that the index of the parent is less than the index of its children. If that property holds then the graphs that are rooted at x and y are isomorphic you set the id of one of the nodes to be the id of the other right of course, you are processing them in some order.

So, one of them would have been identified first you created a new i d for x perhaps and now when you are trying to create one for y you will see that the one for x is already there. So, you repeat it instead of creating a new one going through the process.

(Refer Slide Time: 26:49)



Redundancy Removal Example

For this example, I start off with the leaf level right at this level I can assign the ids there are two ids let us just initialize our data structure with two labels those are 0 and 1 ok. So, for the 0 nodes we have a label 0 for the 1 nodes we have a label these 2 both having the same label means that there is a redundancy, redundancy means that ultimately I can get rid of that node I can get rid of this node also and instead route these edges to the distinct 0 you need only 1 0 and 1 1.

Student: 1.

You do not need multiple zeroes because they are the same Boolean function so, but as of now this is just a labelling exercise we will do the pruning of the structure later on right now we are just assigning the labels to the OBDD.

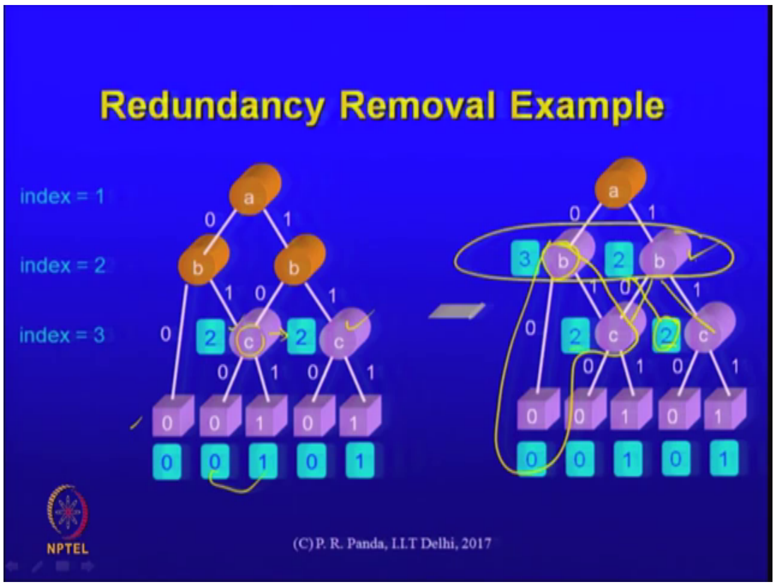
Student: What is pruning?

Pruning means getting rid discarding the redundant nodes. So, already in this process we have found redundant nodes out of the 5, 3 are redundant right so, but I do not remove them as of now I am just proceeding with the traversal anyway right for the purpose of this illustration, we will look at the formal algorithm for doing. So, later on that is more efficient than this you can combine some of this you can argue why first generate duplicate structures and then.

Student: Yes yes.

So, but we look at the two more formal algorithm for doing. So, this is still only a visual representation of the labelling process.

(Refer Slide Time: 28:31)


Redundancy Removal Example

(C) P. R. Panda, I.I.T Delhi, 2017

So, I start off with the leaf structures then I go one level up this is a new function, that is when I come to the index 3 the function computed here is c prime time's 0 plus c times 1 that is the new function it has not been seen as of now because our initialization was with just that these two Boolean function 0 and 1 ok.

So, this is a creation of a new function and new label. So, the 2 is created when I come here I find that the redundancy criteria applies and therefore, I will reuse that same label to the other node when I come to that node that is a new label because this is a new function. So, with b as their index and as 0, as the left function and two as the right function that is this right.

Student: Yes sir.

That function has not been seen yet, so I create a new function I come to this guy what is the redundancy removal process. I look at all of these nodes in that level at that level and see whether I can find another node with the same Boolean function, how do I find that any way b is common for all of these nodes b is common. So, I try to find whether I already computed a function in which the index is b which is 2 here left child is 2 and right child is also.

Student: 2.

2 right righty such a node is not there, so it is not redundant according to that criteria the second criteria, but it is redundant according to the first criteria because.

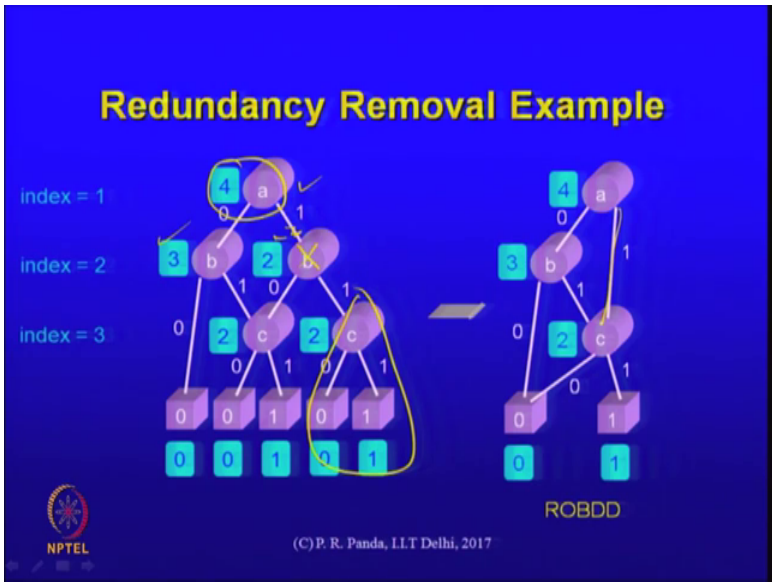
Student: The low is.

Yeah both its children have the same id that must mean that b itself is redundant. So, I just propagate that id to the parent it is the same function. So, these 2 redundancy criteria I evaluate this one is easier because this is just a local just look at both of the children both the children have already been assigned labels.

Student: Yes.

So, it is a trivial check, check whether they are equal or not if they are equal then is fine then you already know that that node is redundant you just propagate the label upwards to the parent, otherwise if they are different then you would look at this level to find out whether there is another node that you have identified with the same children. So, these are the checks that we perform, so two are propagated up and the fine finally, when we move up.

(Refer Slide Time: 31:22)


Redundancy Removal Example
index = 1
index = 2
index = 3
4
a
3
b
2
c
0
1
ROBDD
NPTEL
(C) P. R. Panda, I.I.T Delhi, 2017

This is the creation of a new node because this is the first instance of my looking for a function where a is the index. So, index is one left child is 3 right child is 2 that is the first time it is occurring. So, this is a new function that is my labelling process. So, if my nodes have been labelled from 0 to 4 it means that in that ROBDD structure that I am going to condense from this OBDD structure I would have five nodes each label needs to be retained only once right. So, that is my structure a is fine this b is dropped this entire sub tree is dropped that is what I have here and that edge is instead routed like this to c.

Student: Sir we have just cosine distinct ids to the.

To the nodes in the OBDD.

Student: This function.

Right.

Student: How do we determine out of lines say there are 3 nodes having that two id how do we.

Yes.

Student: Remind this one of them to keep and which 2 of them to.

As it happens one of them would have been created first.

Student: Ok. So, the first one will be kept always.

First one is kept the second one you would get rid off, but of course, this is assuming that we first constructed the OBDD and are extracting the ROBDD from there.

Student: Yeah.

Yeah, but we need not take that approach we can actually build the structure by keeping these rules in mind only one. So, that we construct only the OBDD we don't necessarily first have the r OBDD although the principle is this only.

Student: My concern is that in that index with a c variable.

Right.

Student: We remove the right hand side one.

Yeah.

Student: And it was all right.

Student: And we could have as also computed for right hand c 1 first and then the left hand sides.

Yeah.

Student: C.

Yeah.

Student: And in that case we would have kept the.

We would have retained the right hand side it is ok.

Student: And then we would have to at that edge because it is not a.

Right? That that is all right if we, we see this as a tree restructuring algorithm it could be that I get rid of this it could also be that I get rid of this and I keep this in which case that edge would now.

Student: Have to be rerouted.

Have to be rerouted to that its already conceptually it is a c you keep one sub tree and for the edges that point to the other sub trees now you have to make them point 2 itself.

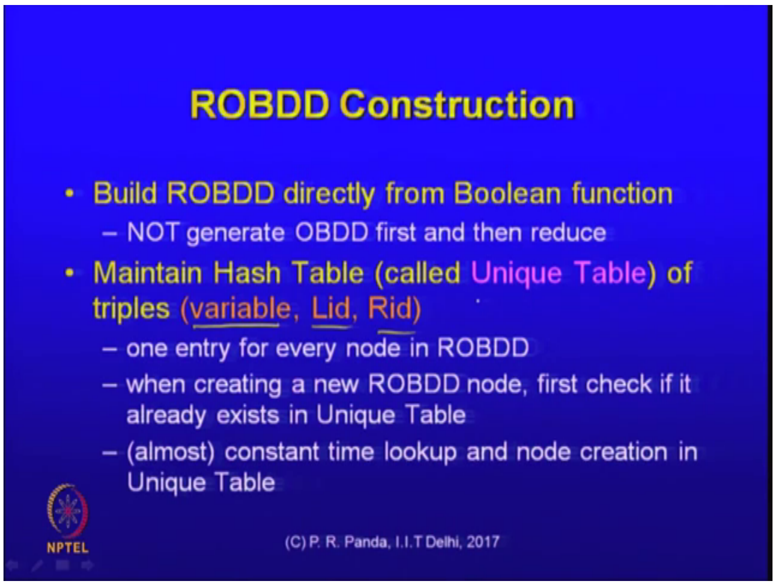
Student: I mean in the optimization we just take it as this guy if there is no edge then we will add that edge or we keep.

But why do not we just get to the form we would not even want to construct the second instance.

Student: Ok.

We do not duplicate, so the duplicate structure is there here for illustration, but the algorithm need not construct the duplicate structure at all, but how it does that why do not we just take a look.

(Refer Slide Time: 34:29)



ROBDD Construction

- Build ROBDD directly from Boolean function
  - NOT generate OBDD first and then reduce
- Maintain Hash Table (called Unique Table) of triples (variable, Lid, Rid)
  - one entry for every node in ROBDD
  - when creating a new ROBDD node, first check if it already exists in Unique Table
  - (almost) constant time lookup and node creation in Unique Table

(C) P. R. Panda, I.I.T Delhi, 2017

Student: Sir the other thing is you said all the edges we have to rerouted.

Yeah.

Student: There is no new edge creation actually this is one difference in the way one more.

Yeah there is no new edge.

Student: We do not create a new edge.

Right the destination.

Student: You just know the.

This is the directed edge right the source is still the same.

Student: Same

But the destination would change ok. So, let us look at the construction of that ROBDD itself we would like to build it directly from the Boolean function given a Boolean function we would like to construct the ROBDD directly not necessarily first generate an OBDD and then do the reduction process, but the principle that we looked at for reduction is the same as what we will use in the construction.

This relies on our maintenance of the search structure which we implicitly used in the redundancy removal, which is now I find a new what is essentially a triple that; consists of a variable; that is the index id of the left child id of the right child right. So, when I come across a new node in that OBDD I want to check whether this thing of that variable the left id and the right id has this already been computed or not has this been encountered before or not, if it has been encountered I should just reuse that label I should not create a new label that is the essential computation that we are doing how do you do that efficiently.

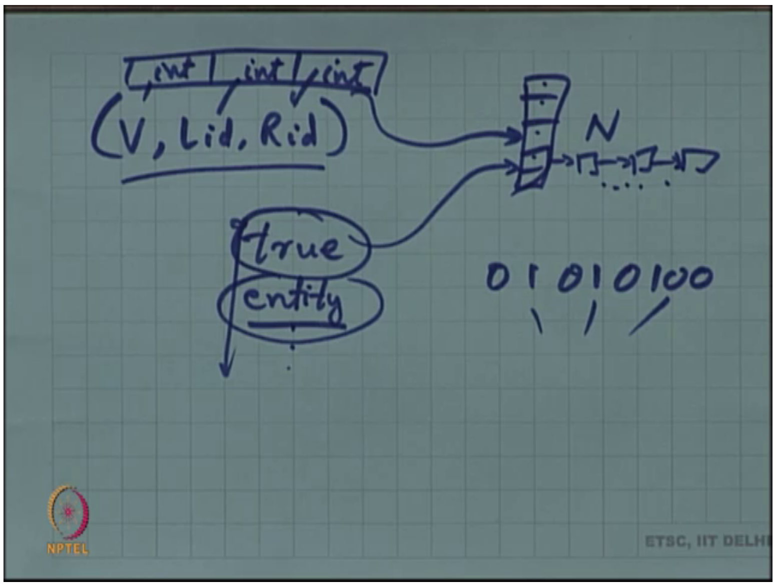
This is essentially an application of the hash table data structure, where it is the search that you want to perform right that search yeah there are 3 elements of that, but that is these all of these are integers.

Student: That is why, but I am I mean I used hash table and I have assigned and it.

Yeah.

Student: It mapped one to the other, so there was an layout there we can have 3 variables also now hash table.

(Refer Slide Time: 36:49)



So, what are these variables anyway, so I have variable left id right id all of these are integers this is this is an index right. So, that is an integer this is also an label is also an integer that is how we have been using it this is also an integer, you could think of this whole thing as constituting a new integer why not let us let us say there were ten bits for this ten bits for this ten bits for this concatenate them you still have 1 integer

Student: Have one integer.

Conceptually this is just a concatenation of bits the hash table works on such a principle you have some string it that is a hash function that takes whatever you have and maps it right. So, this is like a string or whatever is essentially a sequence of bits, so now, if that hash table has n entries, it will take this whatever this number is maybe it is a 65 bit number, but if that hash function would map it to one of the indices of the table.

So, it does not matter too much what the data type is that there are 3 entries there is only a conceptual thing in terms of its implementation you could as well treat them as 1 bit string, and anyway that is what we do it hash table is used when you have a bunch of strings, what is an example suppose a compiler is trying to take an identifier and is giving the result is this a keyword or not right, so it would take keywords like this true entity and so on.

A bunch of these keywords and it will try to find out quickly whether this string that you have given is a keyword or not because the way it proceeds in the grammar check is different, how does it do that a hash table is a good structure because otherwise you it is expensive to do a string compare for every string this there may be 100s of such keywords.

Student: Keywords.

So, they use a hash table structure which will essentially first evaluate each of these guys and assign them some hash, that that hash function will take each of these strings and will map it to one of those indices integer indices of the hash table. So, the way to do it is you first do the mapping it is possible that more than 1 of those strings maps to the same index and therefore, they have a linked list kind of a structure starting from each of those indices.

Student: Ok.

So, when there is a conflict when two elements mapped to that same index then they will have a list structure, but it is still efficient because this list is typically not very long if those hash function is distributing the indices well then this may not be too long. So, essentially this translates to almost a constant time search if the hash function was good then it becomes like a constant time search. So, in that sense having three different things is just a detail you can see that what are these anyway string of characters. So, we are talking about a string of zeros and ones on which that function would be applied. So, conceptually there is nothing different here yeah.

Student: Basically I have never learnt about how hashing happens inside, but from your discussion it feels that it tries to come up with an encoding mechanism.

Yeah.

Student: Where no matter how long the string you give.

Right.

Student: It is quickly able to encode it into a integer map or some map.

That is right.

Student: And then.

That is right.

Student: That

Yes.

Student: That conversion is fairly fast that encoding.

The conversion has to be fast for us to argue that this is a constant time operation you cannot take too much time for that.

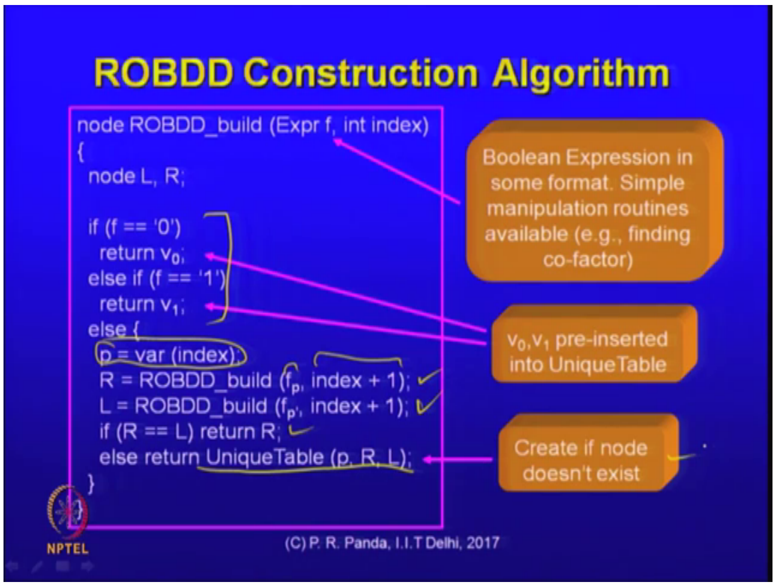
Student: Time for encoding.

You evaluate those hash functions in terms of the properties of how nicely they distribute given whatever the application is essentially it should not be of the kind that there is space in other locations other locations are empty, but this linked list is unnecessarily built up on one of them. It means that large number of conflicts are there that is an example of a bad hash function because it translates to essentially how will you search in the link list it becomes linear time search again. So, the property that a good hash function will try to incorporate is that it is as unpredictable as possible with respect to the mapping it randomizes the mapping to the extent possible.

So, anyway I maintain the hash table here where I give a variable id. So, these are all integers and I have one entry here in the hash table for every distinct node in the OBDD, which becomes every node in the reduced OBDD. When creating a new ROBDD node I will first check if it already exists in the unique table, how do I check I run this search that I just do this search in the hash table and if the search is true it means that this node is already there in the unique table there is no need to create a new one.

So, this is a constant lookup time almost that is usually the case for these hash functions, and a new node is created in the unique table if it was not there if it was there I should just reuse the node itself.

(Refer Slide Time: 42:19)



ROBDD Construction Algorithm

```
node ROBDD_build (Expr f, int index)
{
  node L, R;

  if (f == '0')
    return v0;
  else if (f == '1')
    return v1;
  else {
    p = var (index);
    R = ROBDD_build (fp, index + 1);
    L = ROBDD_build (fp', index + 1);
    if (R == L) return R;
    else return UniqueTable (p, R, L);
  }
}
```

Boolean Expression in some format. Simple manipulation routines available (e.g., finding co-factor)

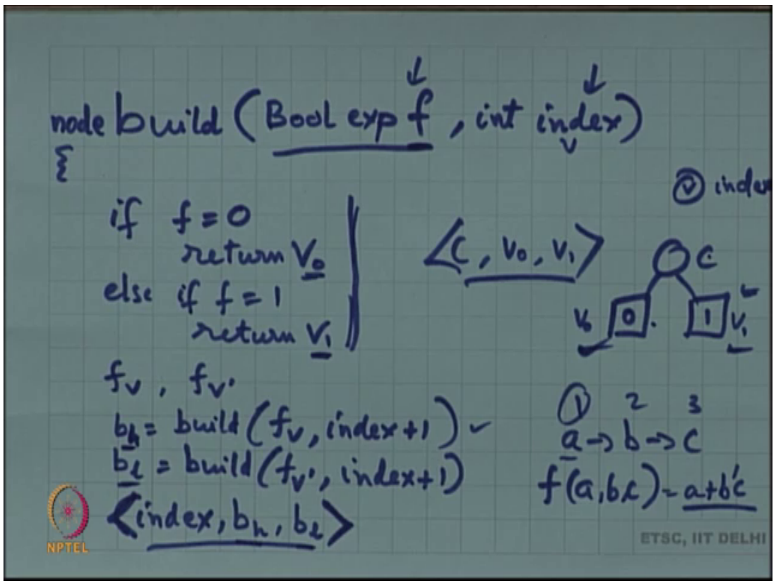
v_0, v_1 pre-inserted into UniqueTable

Create if node doesn't exist

(C) P. R. Panda, I.I.T Delhi, 2017

So, let us just quickly develop the algorithm itself the strategy is what we just said. So, our input is a Boolean expression that is there in some format that format let us ignore for now, but in algebraic expression is able to take a sum of implicates or sum of min terms or whatever it is. So, that Boolean expression is an input, so if I write an algorithm build.

(Refer Slide Time: 42:49)



```
node build (Bool exp f , int index)
{
    if f=0
        return V0
    else if f=1
        return V1
    fv , fv'
    bh = build(fv, index+1)
    bl = build(fv', index+1)
    <index, bh, bl>
```

$\langle c, v_0, v_1 \rangle$

index
c
v_0
0
1
v_1

$\overset{1}{a} \to \overset{2}{b} \to \overset{3}{c}$

$f(a,b,c) - a+b'c$

ETSC, IIT DELHI

And that function is an input for me what other input would be there to the BDD construction algorithm I have two inputs one should be that function itself other should be.

Student: (Refer Time: 43:15).

Yes other is the ordering the ROBDD is dependent on the order if it is a b c, I get some ROBDD if it is a c b I get a different ROBDD. So, that index which is an integer that also has to be an input to the function, what it should return is a node of the ROBDD. If you have a function f of a b c something like that and that order is a b c.

Then I can call that build function with a, or one as the index first this is a recursive definition we can build it inductively using recursion in the following way. So, that function whatever if it is a plus b prime c and so on, that function is provided here as an input and maybe that first one is provided as an as the second input, so how do I go about building this.

Student: Take the cofactors (Refer Time: 44:34).

Yeah the idea is I should use their Shannon expansion and derive the cofactors, cofactors, so the building can be bottom up even though the traversal is top down because we start from the top level. In fact, it is easier to build that structure in a bottom up way the algorithm can do the building up can actually do the traversing top down, but start doing the construction bottom up.

Student: Bottom up.

In the following way let us handle the leaf cases first, so I create these 2 elements first at 0 and 1. So, I insert them into my unique table that unique table is the hash table into which all the nodes are being inserted. So, 0 and 1 are already in such a pre inserted before the function begins, so I can say if that function itself is 0, then what should I return. So, this is v 0 these two nodes are already there, so if the function that is passed to me is 0 then.

Student: Because I am using.

Right I should just return v 0 that node is already there else I say if f equals 1.

Student: Then it is.

Then I should return v 1 if its neither 0 nor 1 what do we do.

Student: Then we.

Then it means that it is an intermediate node somewhere right.

Student: We need to create with that later on that we are.

So, the index is what is already given to us that is the variable. So, let us say that corresponds to variable v then what should I do the variable is given to me function is given to me what do I need to do.

Student: Could be adjust on doing.

Right now nothing is there other than these 2 leaf nodes I have to construct them of course, so how do I construct them.

Student: We cannot put the edge until we.

Until the children nodes are there we cannot put the edges right, but as of now only these two children nodes are there. So, I somehow have to use a recursive process until I end up with the leaf nodes.

Student: The leaf nodes.

Right, so let us say I reach the leaf nodes then at the next level what do I do.

Student: I the add the index to the hash node and the edges according.

So, these are the two base cases of the induction right that is that is already taken care of if it is not one of them then it is an intermediate node. And what I could do is this is an expression right Boolean expression, I can determine f v this is also given the index is also given which variable I know I can return f v and f v prime I can compute them in terms of whatever tool I have given this expression is given to me in some way using that I should be able to drop all the a primes or drop all the as and get f v and f v prime from there.

Student: 2 nodes

In this format in that expression format whatever that expression format is, so this is a local computation I can do of the 2 cofactors I get some other 2 expressions then what.

Student: Then I recall this.

Then I call this build routine again on those expressions. So, I should have build now I give f v the cofactor of f with respect to v with what index.

Student: V index.

Cofactor of f with respect to v.

Student: Index.

Does not have the variable v anymore.

Student: Right.

I should give index plus 1.

Student: Index is.

Index plus 1.

Student: Index plus 1.

Index corresponds to v index plus 1, corresponds to the next node and both of its child functions would be in terms of index plus 1 index plus 2 and so on, the rest of the sub tree right. So, these two I should, so let us say I say low is equal to this the other sub tree I think. So, this is high this is with respect to v prime and the index also being index plus 1. So, this would return the nodes here right, what do I do after this? So, this is a recursive call you understand how the recursive process works if this it turns out is another expression then it will call the same function again and the termination of the recursion happens when.

Student: Right (Refer Time: 49:52).

Right here right when ah, so you can see that in this forward pass there is no construction that is happening we are just invoking the recursion all the way down until we.

Student: Reach.

Terminate we reach the leaf, but the moment we reach the leaf it means that we have enough information to build the lowest level of nodes right. So, then these two are already. So, in that case that b h and b l would be the my v 0 and v 1, this would return v 0 right at some point at and some other point for this leaf I would return a v 1 instead, so that is what would be captured here.

Student: And then we.

As the left sub tree and the right sub tree these are the node type that is being returned so I got these two nodes.

Student: Edges.

In the special case at the leaf node this b h and b l essentially are v 1 and v 0, but in some intermediate case it would be some two nodes that correspond to the left sub tree and the right sub tree, whatever the Boolean functions are. So, then what do I do essentially I have my two children now then what.

Student: Then I add the edges and that mapping thing has to be done now because now.

So, now that I have these two functions, I can now search in my hash table these two are already there in the hash table remember we initialized them. So, now, I want index because that is the variable in terms of which I am going to compose my new function and these to b h and b, I think of this as the tuple that we are searching for in that unique table.

Student: Right if the node.

This is what we are searching for in the hash table. So, we are looking for is there an entry with index as the variable and b h and b l whatever they are if it turns out that they are v 0 and v 1 its fine via if this is c and this is v 0 and v 1 then we are essentially looking for c v 0 v 1 right, otherwise it is whatever it is right. So, this means that the first time you are looking for c v 0 v 1 you would not find it you would insert it, but the second time you are looking for the same thing you would find it and therefore, there is no duplicate structure that is created you would just return the same label.

Student: Same label.

It is the (Refer Time: 52:29) that is all that is there to this algorithm there is nothing else the algorithm is clear there are those two redundancies that we had if you find it then you do not build a new one corresponds to which rule the two rules for redundancy removal one of them is that the 2 children are the same other.

Student: Are the same.

Thing is that the 2 nodes are isomorphic.

Student: Low and high.

Yeah. So, this thing of looking up in the hash table to find whether such an entry is already there is implicitly doing one of those redundancy checks.

Student: Sir it is checking at the same level.

Right it is checking that isomorphic.

Student: Isomorphic.

Sub trees right.

Student: This is also.

This is the isomorphic sub trees, so this is happening just by virtue of us using that hash table data structure and making sure that if something is there then you don't create it that is basically the duplication avoidance what about the other rule, so we will say that.

Student: B h and b l

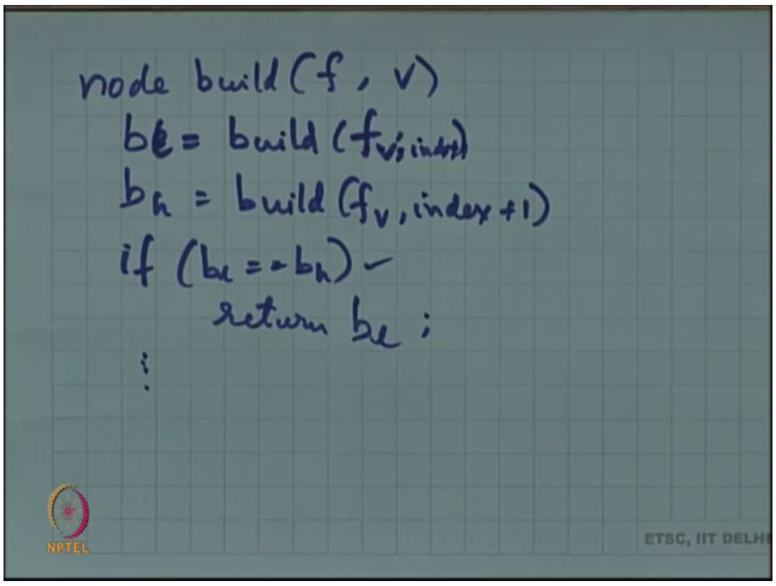
Continuing this if this was v then you return v you looked up in the hash table you got v either because this is the first time it was created a new v was returned or if it was already there then an old v was returned, you can see that this node is actually nothing, but that label you think of this as nothing just like label just so.

Student: Sir before this should not we check for b h is equals to b.

Yeah, so that is the other one this is just we were going with the flow of the search that is why it was introduced first. I returned the v here, but that other one has to be checked which is are the children identical, how do I do that. In fact, there is enough information these two I just compare the two values if they are equal then. In fact, I should at here itself even before doing the lookup there is no need to look up the hash table in that case, because this is returning the value right the call to build is returning the right values if both those values are equal then I should return.

Student: The same index should be propagated for the same.

(Refer Slide Time: 54:55)



```
node build(f, v)
  bl = build(fv', index+1)
  bh = build(fv, index+1)
  if (bl == bh)
     return bl;
  :
```

I got b b l equals remember this build call is going to propagate things further that is what it was doing b h equals another build right. So, essentially this was f of v with then index plus 1 f of v.

Student: Sir it is also written in the node name if they are equal then if we just you can get that with an another hash table.

Right now our check was if b l equals.

Student: B h.

B h then it means that I did the traversal of the two cofactors it turns out that the two cofactors are identical. In fact, the structure is corresponding to those cofactors have already been built and designed and inserted appropriately into that hash table as of now, so what else remains to be done for the parent node.

Student: We need to send the one of the information of b l b h either of them.

Yes.

Student: That return either b l or return either b h.

Right.

Student: This is what we will write

This is not with v right, so I should just return either b 1 that is all because there is nothing new to be done, in fact, I am going to drop this node.

Student: Right yeah.

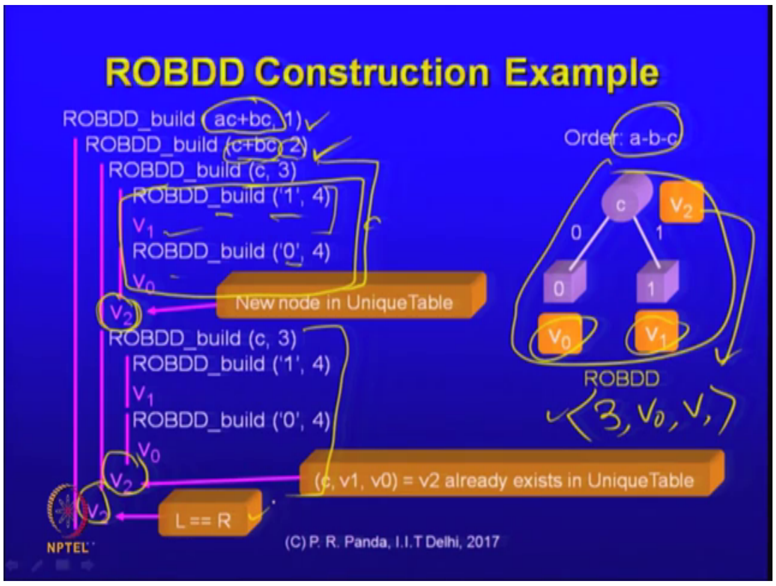
Right I am I have to drop this node. So, I should just propagate the label of the child to.

Student: To the parent.

It is parent right, so that is all that we do otherwise I should do the unique table search that I was that is it, these are the two redundancies that are identified one is this simple check to find whether are the 2 children are equal other is the lookup in the hash table to make sure there is no duplication. So, we do not create the duplicate entries, that is what my algorithm is that is the expression 2 nodes that is what we already saw this is the termination condition p is the variable corresponding to that index you call the same two routine this these are the recursive calls on the cofactors with index plus 1 ok.

Now you are doing this check to find whether the 2 children are equal if yes then you return one of them otherwise you return whatever that unique table lookup is giving you create the node. If it does not exist if it does exist then you just return the note there is no creation.

(Refer Slide Time: 57:35)



ROBDD Construction Example

```
ROBDD_build (ac+bc, 1)
  ROBDD_build (c+bc, 2)
    ROBDD_build (c, 3)
      ROBDD_build ('1', 4)
      V1
      ROBDD_build ('0', 4)
      V0
    V2          <- New node in UniqueTable
    ROBDD_build (c, 3)
      ROBDD_build ('1', 4)
      V1
      ROBDD_build ('0', 4)
      V0
    V2          <- (c, v1, v0) = v2 already exists in UniqueTable
  V2            <- L == R
```

Order: a-b-c

ROBDD

$\langle 3, V_0, V_1 \rangle$

(C) P. R. Panda, I.I.T Delhi, 2017

We can go through a construction example, but hopefully the logic is clear. Suppose I start off with this is the same function that we used in our examples there is 1 2 3 4 these are the index values that are sent. So, the ordering is a b c means that a is 1 b is 2 m c is 3. So, this call at the highest level just results in a recursive call there is nothing further that is done, you cannot do anything more until you return from those recursive calls.

So, you first compute the cofactor of that function with respect to a you get c plus b c, that itself is also a complex expression you can't do anything you have to call the recursion further. So, c plus b c again you take the cofactor with respect to b now 2 is b right. So, you take the cofactor with respect to b you get c and now you call with the recursion this time with 3 which is c, fine with c. Now again you do the same things you take the cofactor of c with respect to c you get 1 the cofactor of c with respect to c would be you replace c by 1.

Student: Yeah.

So, you get one if the cofactor with respect to c prime is 0 you replace c by 0, but this is the termination of our recursion because you are sending a 1 there true or false there is enough information. In fact, these are the nodes that would be returned from that level of the recursion the innermost level right v 1 is returned from here v 0 is returned so; that means, that that much is done I am back at this level of recursion where c was my expression, I would now look for a node in which c is the variable.

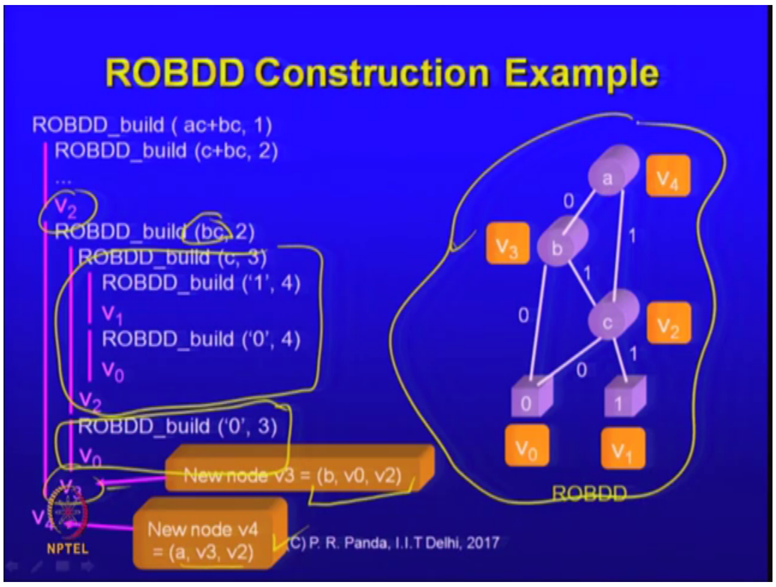
So, the index is 3 left child is v 0 right child is v 1 such a node is not yet there because my table. So, far has only v 0 and v 1. So, a new node v 2 is created with that function, so that is my ROBDD. So, far I returned from here with respect to b you get see that is what this was, but when you take the cofactor of c plus b c with respect to b prime what do you get.

Student: Still get c.

You also get c, so essentially this is another call we can do something about it this is the same thing that is called twice, that is it is the same logic all over again because it that function is again c because v 2 was already created corresponding to that function c now at the end of this recursion. In fact, what do I have I have c s index is 3 I have v 0 and v 1 that is my tuple right. So, this was inserted the first time that was my v 2 second time when I look up I will find this index in the hash table.

So, I also return v 2, which means that as I comeback to this level here I find that both of these cofactors are actually same both are c, then what do I do the v 2 is actually what is propagated at that level itself because both of the cofactors are the same so, but therefore, both the children are the same node both the children are v 2. So, the parent also would return v 2 that is one ok. So, this is the recognition that l equals r, so l equals r I would just return v 2.

(Refer Slide Time: 61:39)



ROBDD Construction Example

```
ROBDD_build ( ac+bc, 1)
  ROBDD_build (c+bc, 2)
  ...
  V2
  ROBDD_build (bc, 2)
    ROBDD_build (c, 3)
      ROBDD_build ('1', 4)
      V1
      ROBDD_build ('0', 4)
      V0
    V2
    ROBDD_build ('0', 3)
    V0
    New node v3 = (b, v0, v2)
  V3
V4
New node v4 = (a, v3, v2)
```

(C) P. R. Panda, I.I.T Delhi, 2017

That is part of the recursion done this is one of that a plus b c then I can go down the a plus b c with respect to a prime I would get what function.

Student: B.

No b c because the first one would go away I would get b c that is what this is similarly I would take the cofactor of b c with respect to b you get c ah, which is what has already been done cofactor of b with respect to b prime is what cofactor of b c with respect to b prime is.

Student: 0.

Is 0, so I would call these returns 0 so nothing new here, but when I come here now I am looking for?

Student: Node.

A node in the hash table with b as the index, so 2 as the index and v 0 on the left and v 2 on the right such a node is not yet there. So, I create a new node v 3 that is essentially part of my new ROBDD that god created, once I am done with that I return from all the recursions at the highest level I am looking for a node with a as the variable which is index is 1 and v 3 and v 2. So, from 1 I got v 3 from the other I got v 2, so the 2 children are v 3 and v 2 that is what I am looking for that node is not yet there.

So, I would construct this then this becomes my ROBDD, any questions the basic algorithm is fine and the way we traversed through the example illustration that is also clear right. In fact, that hash table has within it everything that I need I do not create a separate data structure graph or anything, because the children are there you just need a pointer to the top from there you know what the children are and therefore, you actually have the entire structure implicitly constructed in the table itself.

Student: Yes.

There is no need to maintain any separate graph node edge or anything we have been calling it a node, but. In fact, you do not really need a separate graph structure the result of this traversal. In fact, everything is encoded in that hash table itself and that is the BDD the representation of the BDD even though we are showing all these nice diagrams and colours and nodes and edges you actually have a more boring table, but it is embedding everything that we have been talking about it is a very powerful structure it BDD it is the ROBDD.

Student: ROBDD.

That we are talking about what to do with it there are lots of interesting applications and operations that we can perform we look up about it next, but the basic representation is just this.