ROBDD's

Reduced **O**rdered **B**inary **D**ecision **D**iagrams

- represents a logic function by a **graph**. (*many logic functions can be represented compactly - usually better than SOP's*)

- **canonical** form (**important**) (*only canonical if an ordering of the variables is given*)

- Many logic operations can be preformed **efficiently** on BDD's (*usually linear in size of result - tautology and complement are constant time*)

- **size** of BDD critically dependent on **variable ordering**.

ROBDD's

- directed acyclic graph (DAG)
- one root node, two terminals 0,1
- each node, two children, and a variable
- Shannon co-factoring tree, except **reduced** and **ordered**

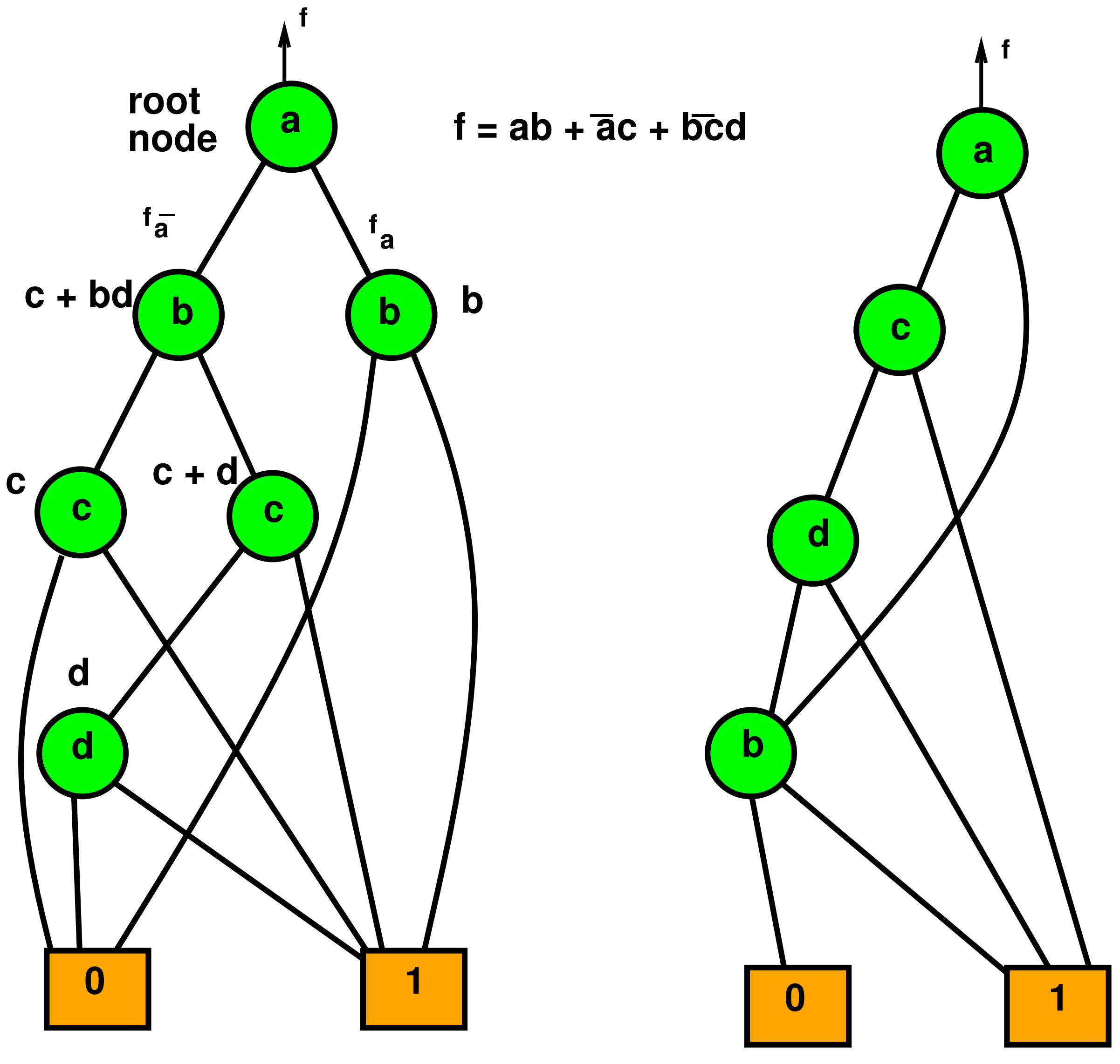
Reduced:

1. any node with identical children is removed
2. two nodes with isomorphic BDD's are merged

Ordered: Co-factoring variables (splitting variables) always follow the same order

$$x_{i_1} < x_{i_2} < x_{i_3} < \ldots < x_{i_n}$$

Example

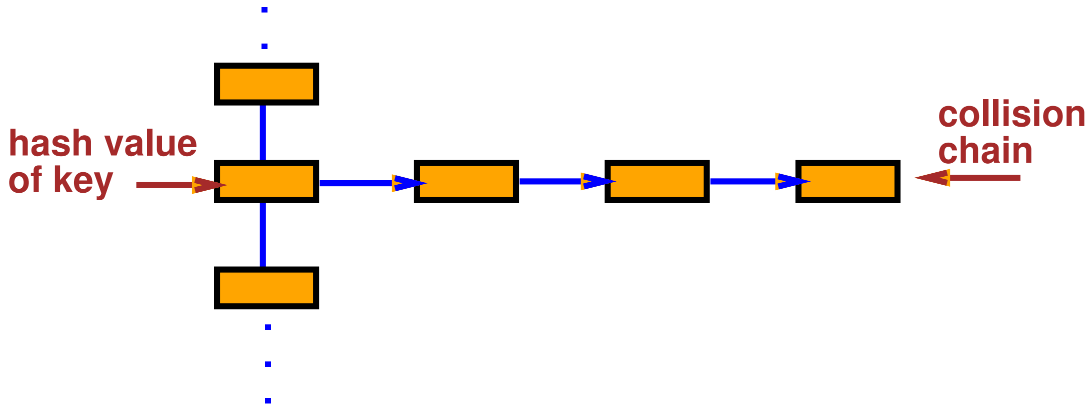


Two different orderings, same function.

Efficient Implementation of BDD's

(*Reference: Brace, Rudell, Bryant - DAC 1990*)

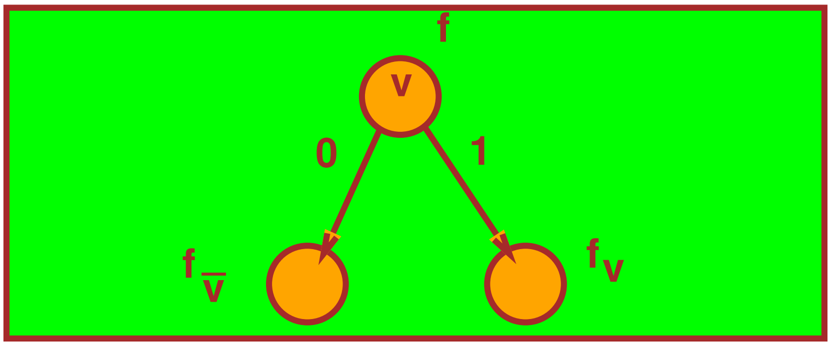
Hash-Table: hash-fcn(key) = value



Strong canonical form: A "unique-id" is associated (through a hash table) uniquely with each element in set.

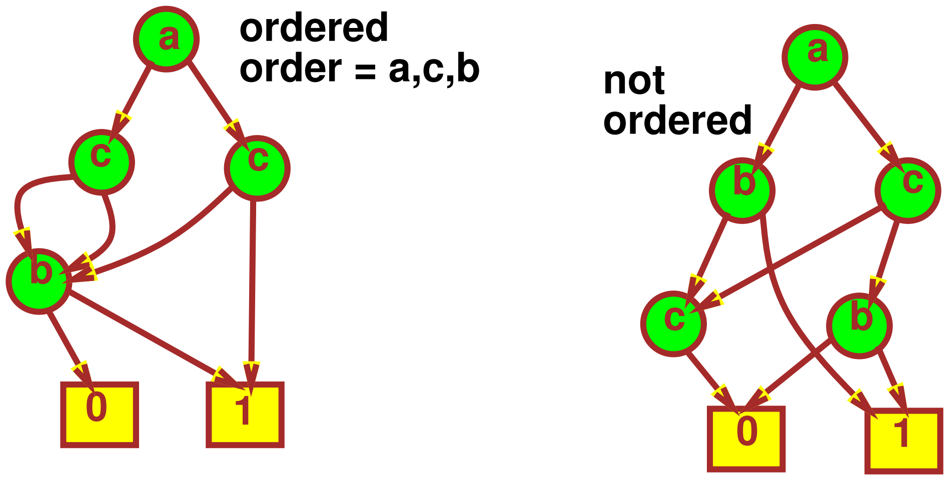
With BDD's the set is the set of all logic functions. A BDD node is a function. Thus each function has a unique-id in memory.

BDD is **compressed** Shannon co-factoring tree.



ROBDD

Ordered BDD (OBDD) Input variables are ordered - each path from root to sink visits nodes with labels (variables) in ascending order.



Reduced Ordered BDD - reduction rules:

1. if the two children of a node are the same, the node is eliminated - $f = cf + \bar{c}f$.

2. if two nodes have isomorphic graphs, they are replaced by one of them.

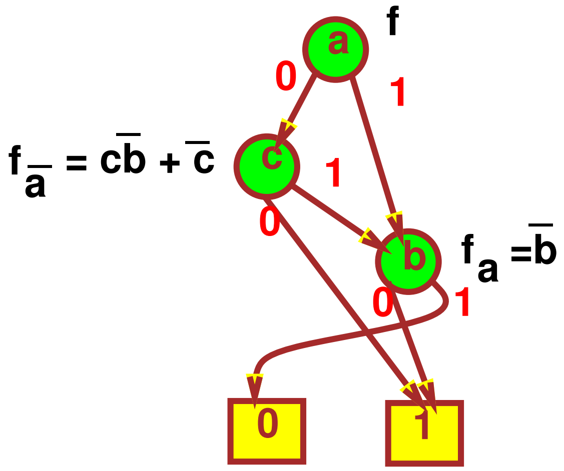
These two rules make it so that each node represents a distinct logic function.

Theorem 1 *(Bryant - 1986) ROBDD's are canonical*

Thus two functions are the same iff their ROBDD's are equivalent graphs (isomorphic). Of course must use **same order** for variables.

Function is Given by Tracing All Paths to 1

$f = \overline{b} + \overline{ac} = a\overline{b} + \overline{a}c\overline{b} + \overline{ac}$ all paths to the 1 node



Notes:

- By tracing paths to the 1 node, we get a cover of pairwise disjoint cubes.
- The power of the BDD representation is that it does not explicitly enumerate all paths; rather it represents paths by a graph whose size is measured by its nodes and not paths.
- A DAG can represent an exponential number of paths with a linear number of nodes.
- Each node is given by its Shannon representation: $f = af_a + \overline{a}f_{\overline{a}}$.

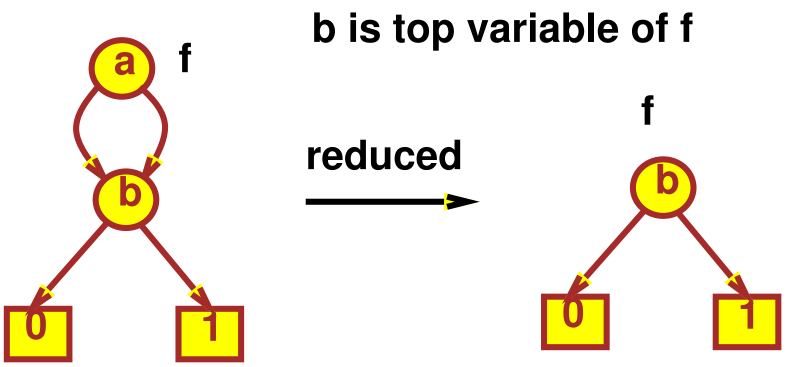
Implementation

Variables are **totally ordered**: If $v < w$ then v occurs "higher" up in the ROBDD (call it BDD from now on).

Definition 1 **Top variable** *of a function f is a variable associated with its* **root node**.

Example: $f = ab + \overline{a}bc + \overline{a}b\overline{c}$. Order is (a, b, c).

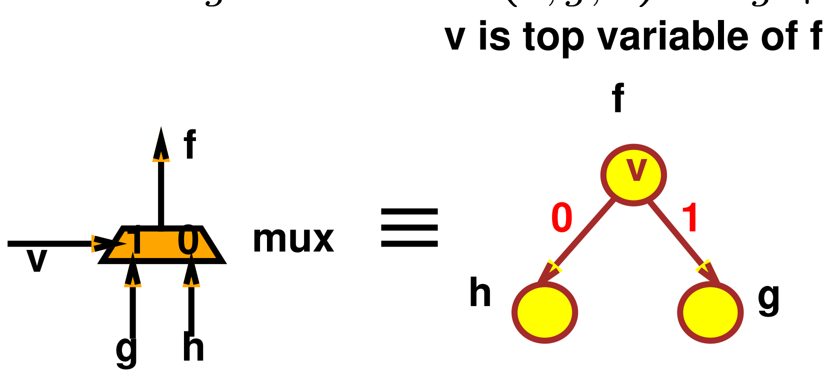
$$f_a = b \quad f_{\overline{a}} = b$$



f does not depend on a, since $f_a = f_{\overline{a}}$.

Each node is written as a triple: $f = (v, g, h)$ where $g = f_v$ and $h = f_{\overline{v}}$. We read this triple as

$$f = \text{ if } v \text{ then } g \text{ else } h = ite(v, g, h) = vg + \overline{v}h$$



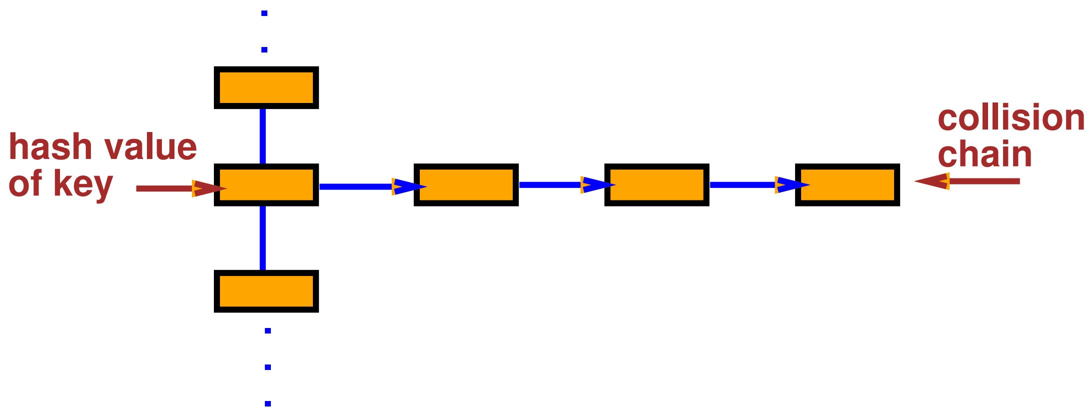
ITE Operator

$$\text{ite}(f, g, h) = fg + \overline{f}h$$

ite operator can implement any two variable logic function. There are 16 such functions corresponding to all subsets of vertices of B^2: $\overline{f}\overline{g}, \overline{f}g, f\overline{g}, fg$

Table	Subset	Expression	Equivalent Form
0000	0	0	0
0001	$AND(f, g)$	fg	$ite(f, g, 0)$
0010	$f > g$	$f\overline{g}$	$ite(f, \overline{g}, 0)$
0011	f	f	f
0100	$f < g$	$\overline{f}g$	$ite(f, 0, g)$
0101	g	g	g
0110	$XOR(f, g)$	$f \oplus g$	$ite(f, \overline{g}, g)$
0111	$OR(f, g)$	$f + g$	$ite(f, 1, g)$
1000	$NOR(f, g)$	$\overline{f + g}$	$ite(f, 0, \overline{g})$
1001	$XNOR(f, g)$	$f \overline{\oplus} g$	$ite(f, g, \overline{g})$
1010	$NOT(g$	$\overline{g}$	$ite(g, 0, 1)$
1011	$f \geq g$	$f + \overline{g}$	$ite(f, 1, \overline{g})$
1100	$NOT(f)$	$\overline{f}$	$ite(f, 0, 1)$
1101	$f \leq g$	$\overline{f} + g$	$ite(f, g, 1)$
1110	$NAND(f, g)$	$\overline{fg}$	$ite(f, \overline{g}, 1)$
1111	1	1	1

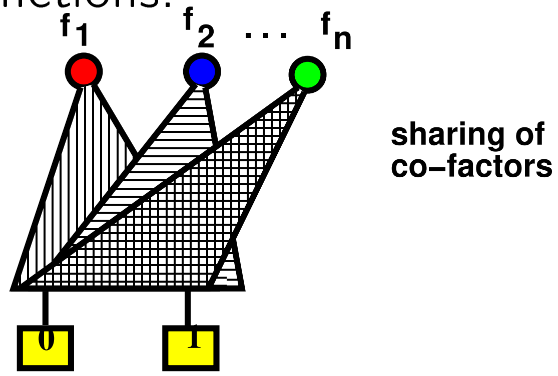
Unique Table - Hash Table



Before a node (v, g, h) is added to BDD data base, it is looked up in the "unique-table". If it is there, then existing pointer to node is used to represent the logic function. Otherwise, a new node is added to the unique-table and the new pointer returned.

Thus a **strong canonical form** is maintained. The node for $f = (v, g, h)$ exists **iff** (v, g, h) is in the unique-table. There is only one pointer for (v, g, h) and that is the address to the unique-table entry.

Unique-table allows single multi-rooted DAG to represent all users' functions:



Recursive Formulation of ITE

v = top-most variable among the three BDD's f, g, h

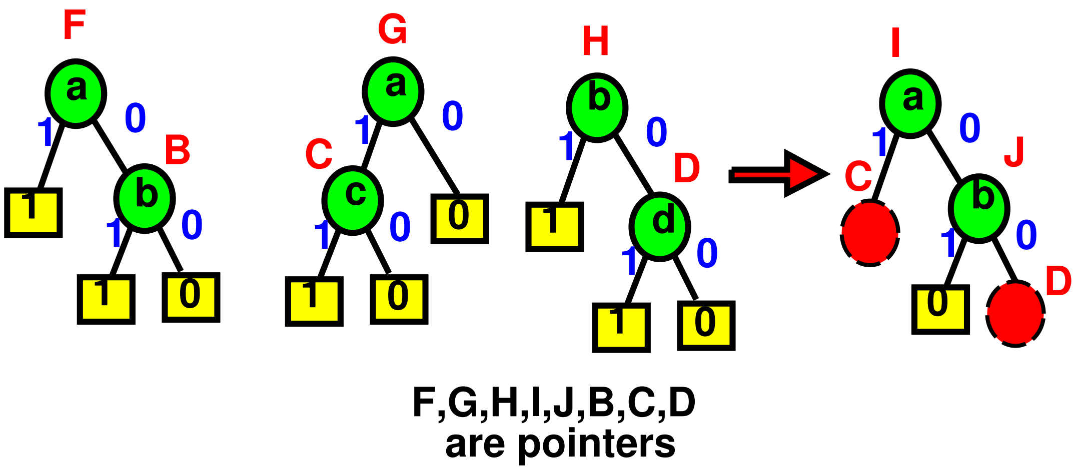
$$
\begin{aligned}
ite(f,g,h) &= fg + \overline{f}h \\
&= v(fg+\overline{f}h)_v + \overline{v}(fg+\overline{f}h)_{\overline{v}} \\
&= v(f_v g_v + \overline{f}_v h_v) + \overline{v}(f_{\overline{v}} g_{\overline{v}} + \overline{f}_{\overline{v}} h_{\overline{v}}) \\
&= ite(v, ite(f_v, g_v, h_v), ite(f_{\overline{v}}, g_{\overline{v}}, h_{\overline{v}})) \\
&= (v, ite(f_v, g_v, h_v), ite(f_{\overline{v}}, g_{\overline{v}}, h_{\overline{v}})) \\
&= (v, \widetilde{f}, \widetilde{g}) = R
\end{aligned}
$$

Terminal cases:
$$
\begin{aligned}
(0, g, f) &= (1, f, g) = f \\
ite(f, g, g) &= g
\end{aligned}
$$

```
ite(f, g, h)
    if(terminal case){
        return result;
    } else if (computed-table has entry (f, g, h){
        return result;
    }else{
        let v be the top variable of (f, g, h);
        f~ ← ite(f_v, g_v, h_v);
        g~ ← ite(f_v̄, g_v̄, h_v̄);
        if (f~ equals g~) return g~;
        R ← find_or_add_unique_table(v, f~, g~);
        insert_computed_table({f, g, h}, R);
        return R; } }
```

The "insert_computed_table" is a cache table where ite results are cached.

Example



$$
\begin{aligned}
I &= ite(F,G,H) \\
&= (a, ite(F_a, G_a, H_a), ite(F_{\overline{a}}, G_{\overline{a}}, H_{\overline{a}})) \\
&= (a, ite(1, C, H), ite(B, 0, H)) \\
&= (a, C, (b, ite(B_b, 0_b, H_b), ite(B_{\overline{b}}, 0_{\overline{b}}, H_{\overline{b}}))) \\
&= (a, C, (b, ite(1, 0, 1), ite(0, 0, D))) \\
&= (a, C, (b, 0, D)) \\
&= (a, C, J)
\end{aligned}
$$

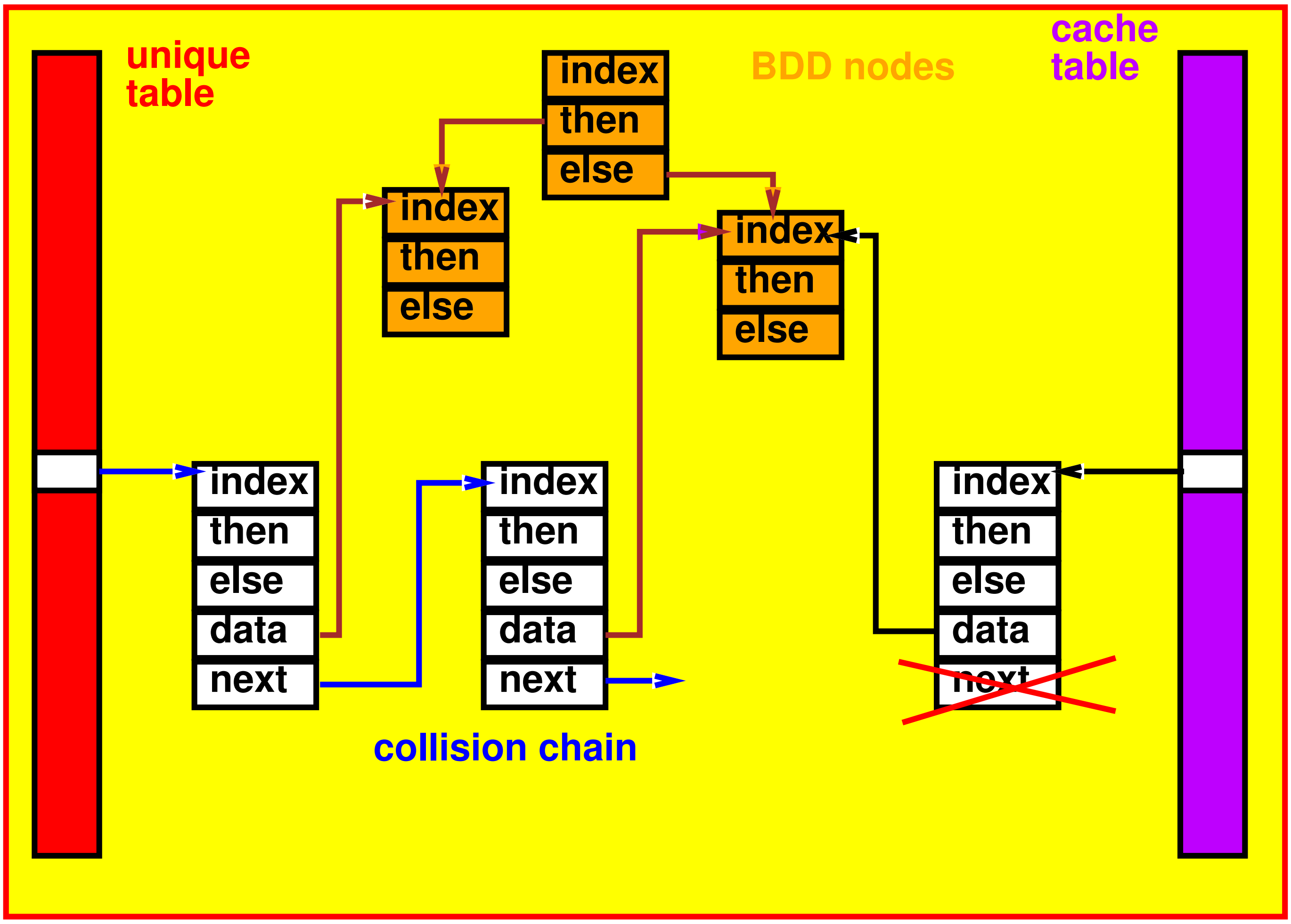
Check:

$$
\begin{aligned}
F &= a + b \\
G &= ac \\
H &= b + d
\end{aligned}
$$

$$
\begin{aligned}
ite(F,G,H) &= (a+b)(ac) + \overline{a}\overline{b}(b+d) \\
&= ac + \overline{a}\overline{b}d
\end{aligned}
$$

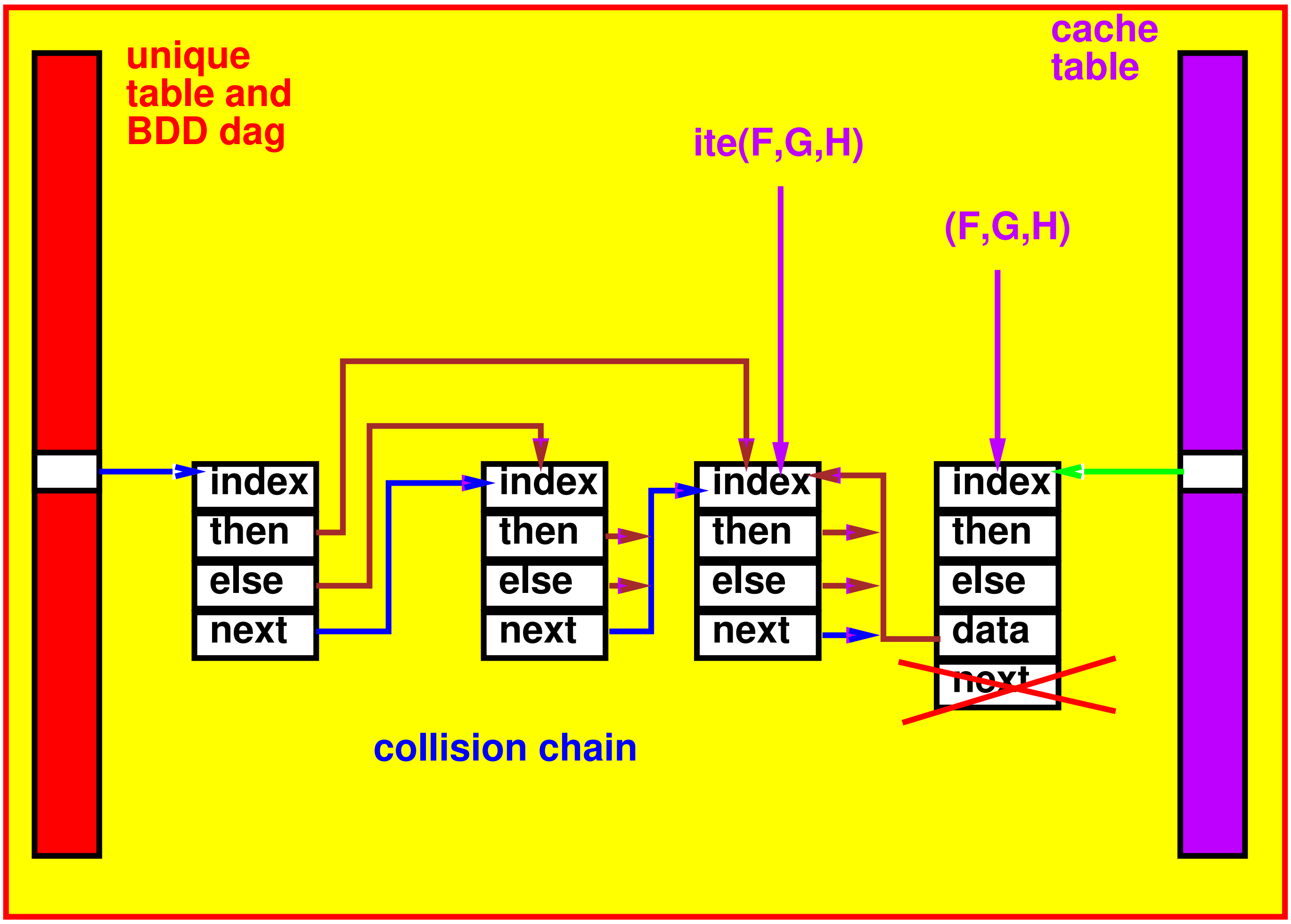
Computed Table

Keep a record of (F, G, H) triplets already computed by the ite operator in a **hash-based cache** (*"cache" table*). This means that the collision chain is not used (*if collision, old entry thrown away*).



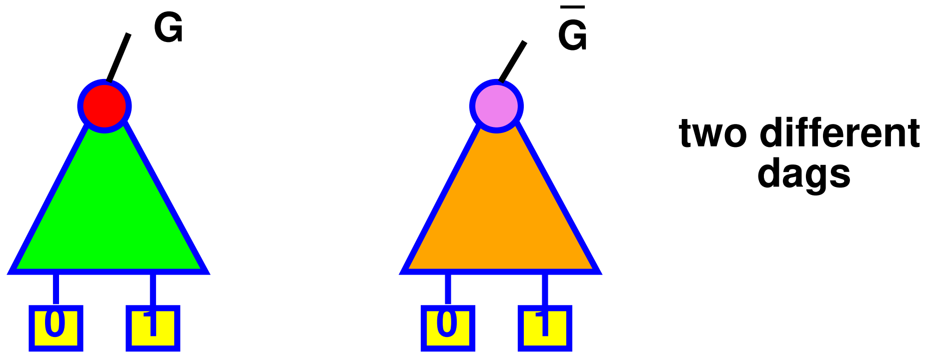
However, this is wasteful since the BDD nodes and collision chain can be merged.

Better Implementation

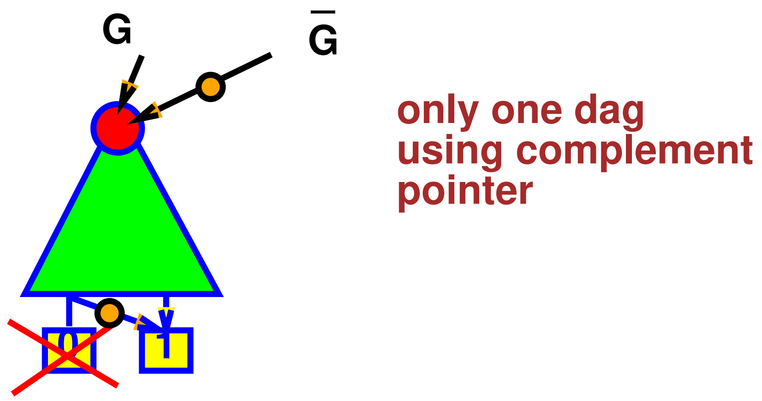


Here the BDD nodes and the collision chain are merged.
On average, only 4 pointers per BDD node.

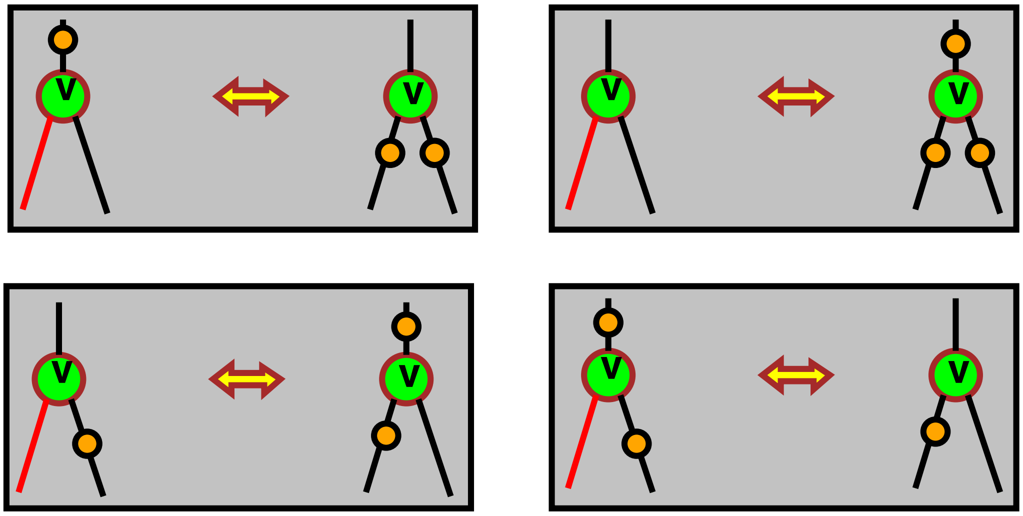
Extension - Complement Edges



Can combine by making complement edges:



To maintain strong canonical form, need to resolve 4 equivalences:



Solution: Always choose one on **left**, i.e. the "then" leg must have **no** complement edge.

Ambiguities in Cache Table

Standard Triples:

$$\begin{array}{rcl} ite(F,F,G) & \Longrightarrow & ite(F,1,G) \\ ite(F,G,F) & \Longrightarrow & ite(F,G,0) \\ ite(F,G,\overline{F}) & \Longrightarrow & ite(F,G,1) \\ ite(F,\overline{F},G) & \Longrightarrow & ite(F,0,G) \end{array}$$

To resolve equivalences:

$$\begin{array}{rcl} ite(F,1,G) & \equiv & ite(G,1,F) \\ ite(F,0,G) & \equiv & ite(\overline{G},0,\overline{F}) \\ ite(F,G,0) & \equiv & ite(G,F,0) \\ ite(F,G,1) & \equiv & ite(\overline{G},\overline{F},1) \\ ite(F,G,\overline{G}) & \equiv & ite(G,F,\overline{F}) \end{array}$$

1. first argument is chosen with smallest top variable.

2. break ties with smallest address pointer.

Triples:

$$ite(F,G,H) \equiv ite(\overline{F},H,G) \equiv \overline{ite(F,\overline{G},\overline{H})}, \equiv \overline{ite(\overline{F},\overline{H},G)}$$

Choose the one such that the first and second argument of ite should not be complement edges (*i.e. the first one above*).

Tautology Checking

Tautology returns 0,1, or NC (not constant).

```
ITE_constant(F,G,H){
    if (trivial case) {
        return result (0,1, or NC);
    } else if (cache table has entry for (F,G,H)) {
        return result;
    } else {
        let v be the top variable of F,G,H;
        R ← ITE_constant(F_v, G_v, H_v);
        if (R = NC) {
            insert_cache_table({F,G,H},NC);
            return NC;
        }
        E ← ITE_constant(F_v̄, G_v̄, H_v̄);
        if (E = NC or R ≠ E){
            insert_cache_table({F,G,H},NC);
            return NC;
        insert_cache_table({F,G,H}, E);
        return E;
    }
}
```

Note, that in computing ITE_constant, we set up a temporary cache-table for storing results of the ITE_constant operator. When done, we can throw away this table if we like.

Compose

Compose is an important operation for building the BDD of a circuit.

$$\text{compose}(F, v, G): \quad F(v, x) \to F(G(x), x)$$

Means substitute $v = G(x)$.

compose (F, v, G) $\{$ (*in F replace v with G*)
    if(top_var(F) $> v$) return F;
    (*because F does not depend on v*)
    if(top_var(F) $= v$) return ITE(G, F_1, F_0);
    $R \leftarrow$ compose(F_1, v, G);
    $E \leftarrow$ compose(F_0, v, G);
    return ITE(top_var(F),R, E);
    (*note that we call ITE on this rather than*)
    (*returning* (top_var(F),R, E)) $\}$

Notes:

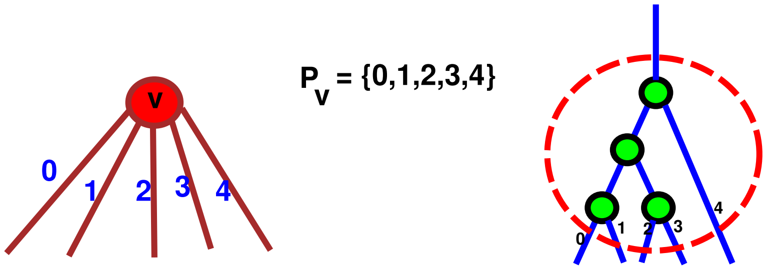
1. F_1 is the 1-child of F, F_0 the 0-child

2. G, R, E are not functions of v

3. if top_var of F is v, then $ite(G, R, E)$ does the replacement of v by G.

Multivalued Decision Diagrams (MDD's) - "BDD's" for MV-functions

There is an equivalent theory (canonical etc.) for MDD's:



Typically, we encode the multi-valued variable with $log_2(|P_v|)$ binary variables and use unused codes as "don't cares" in a particular way

Sets and Graphs:
Thus we can represent and manipulate general **sets and graphs**.

$$\begin{array}{rcl} \text{Set} & \Longrightarrow & \text{characteristic function of set} \\ & \equiv & ((f(v) = 1) \Longleftrightarrow (v \in S \subseteq P_v)) \end{array}$$

$$\text{Graph: } ((f(x, y) = 1) \Longleftrightarrow (x, y) \text{ is an edge in graph}$$

where x and y are multi-valued variables representing nodes in the graph.

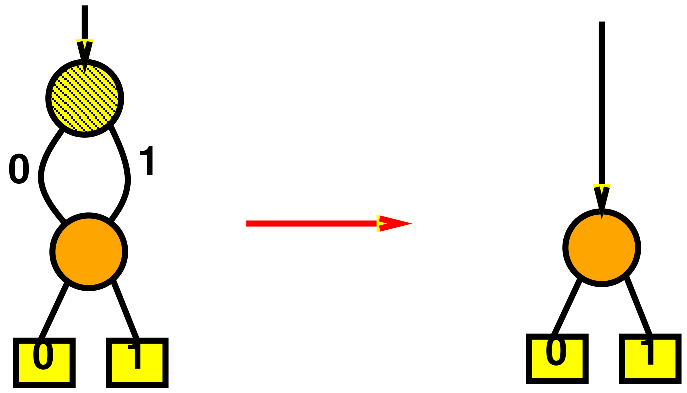
ZBDD's were invented by Minato to efficiently represent **sparse** sets. They have turned out to be extremely useful in implicit methods for representing primes (which usually are a sparse subset of all cubes).

Zero Suppressed BDD's - ZBDD's

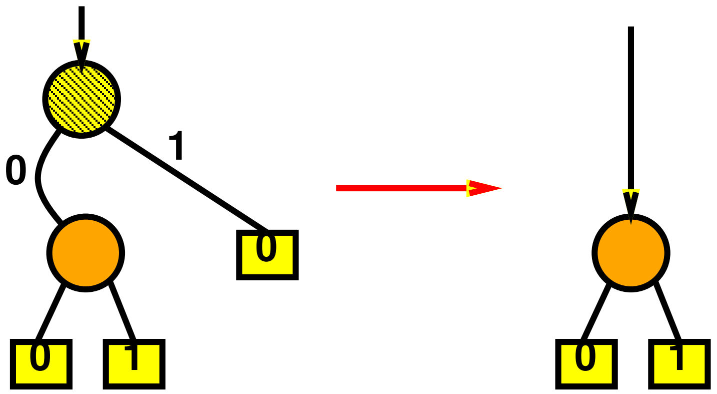
Different reduction rules:

- BDD: eliminate all nodes where **then** edge and the **else** edge point to the same node.
- ZBDD: eliminate all nodes where the **then** node points to 0. Connect incoming edges to **else** node.
- For Both: share equivalent nodes.

BDD:



ZBDD:



Canonicity

Theorem 2 *(Minato) ZBDD's are canonical given a variable ordering and* **the support set**.

Example: