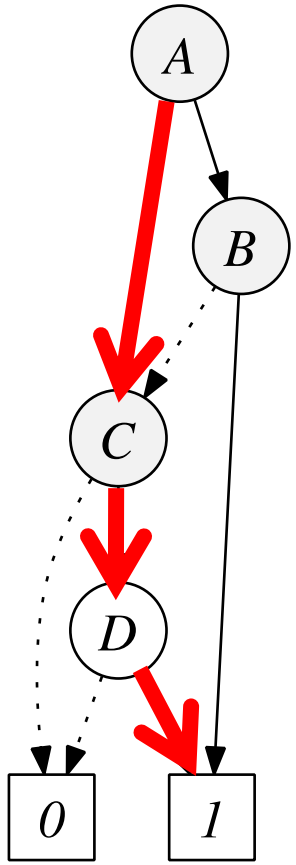
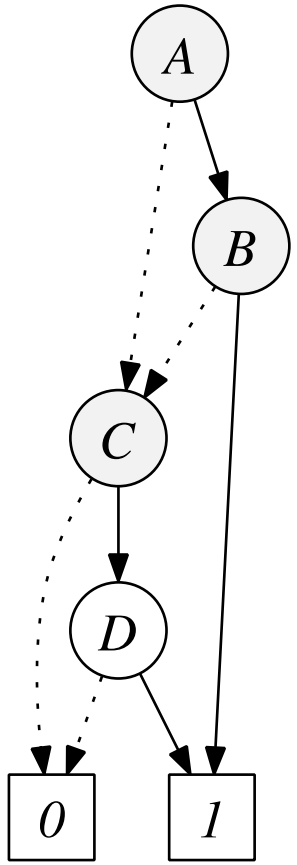
Sentential Decision Diagrams

SDD Circuits

Ordered Binary Decision Diagrams (OBDDs)



Ordered Binary Decision Diagrams (OBDDs)



- Variable order needed

- Unique for a given variable order, if reduced (canonical)

- Variable order matters!

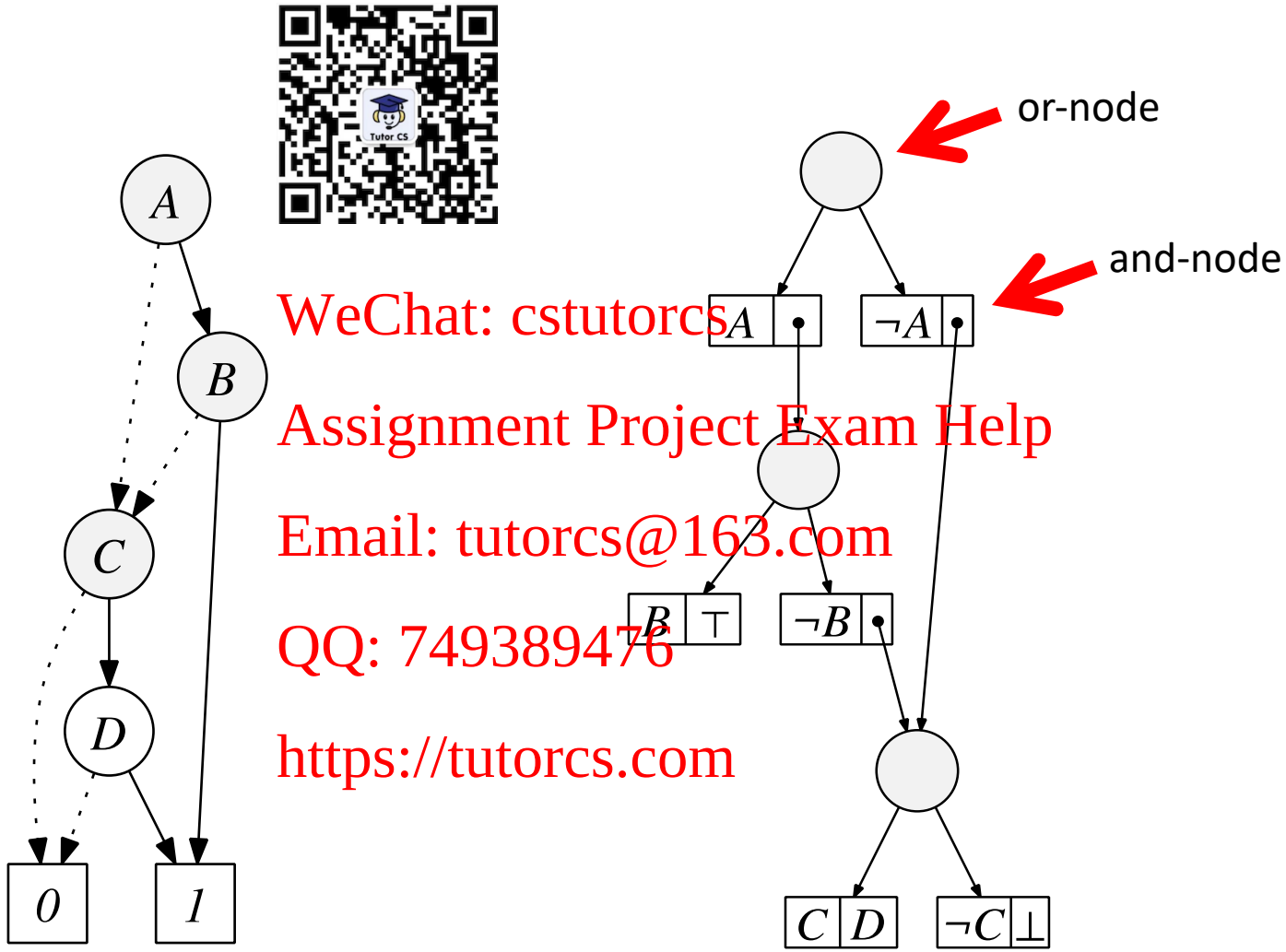
OBDDs are Influential

- Theory: One of the most studied tractable representations of Boolean functions
- Practice:
  - Canonicity
  - Polytime Apply:
    - e.g. Compile CNF/DNF bottom up
    - e.g. Represent/manipulate states and transitions
  - Publicly available implementations (CUDD package)
  - Algorithms for finding good variable orders (heavily researched)

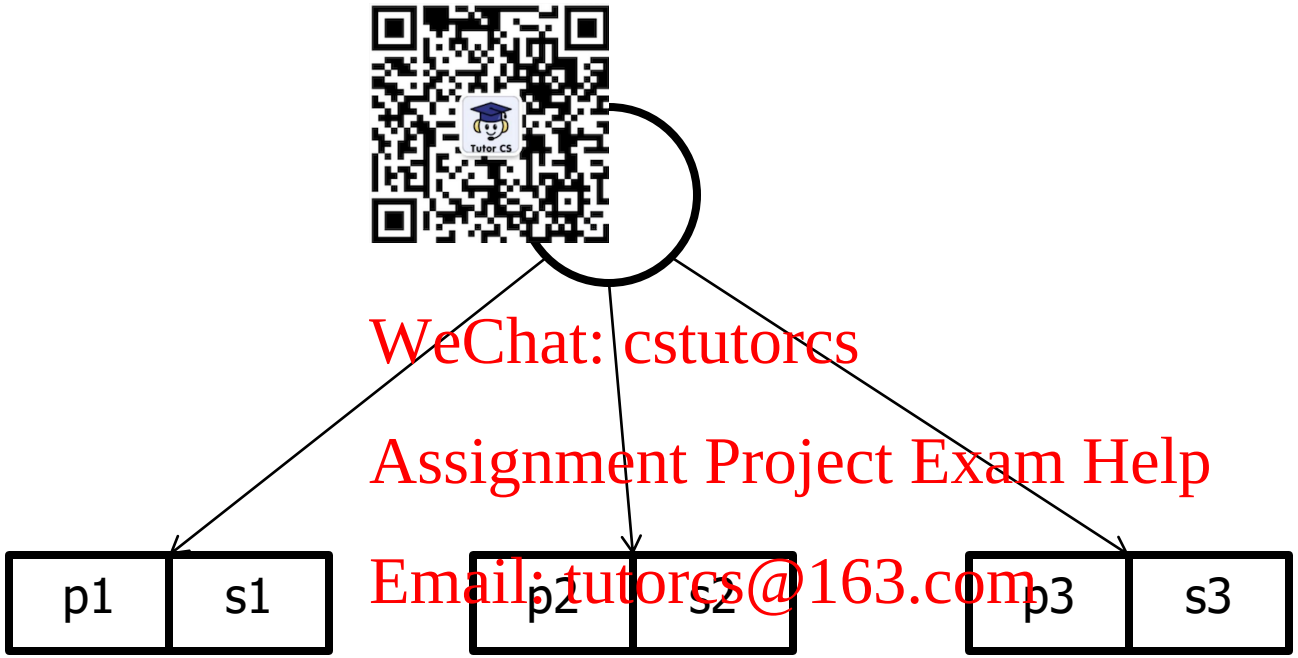
SDD: Sentential Decision Diagram

BDD: Binary Decision Diagram

OBDD: Branch on Literals



SDD: Branch on Sentences

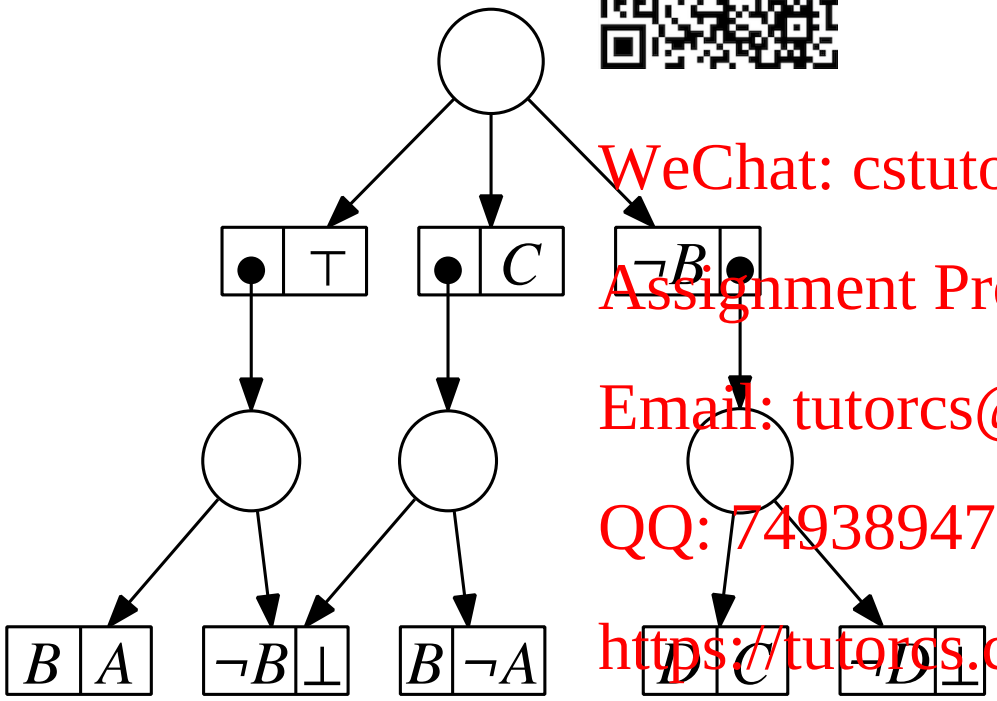


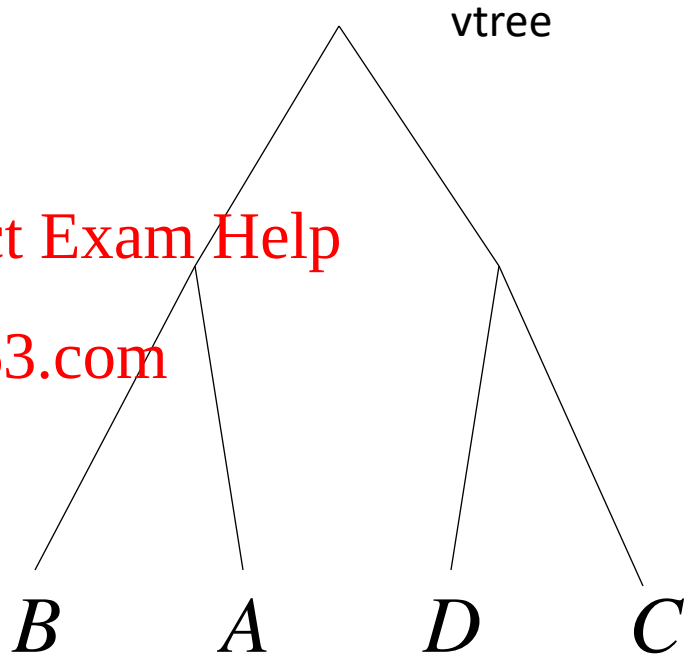
Branch on sentences p1, p2, and p3:

- p1, p2, p3 are mutually exclusive and exhaustive, pi cannot be false

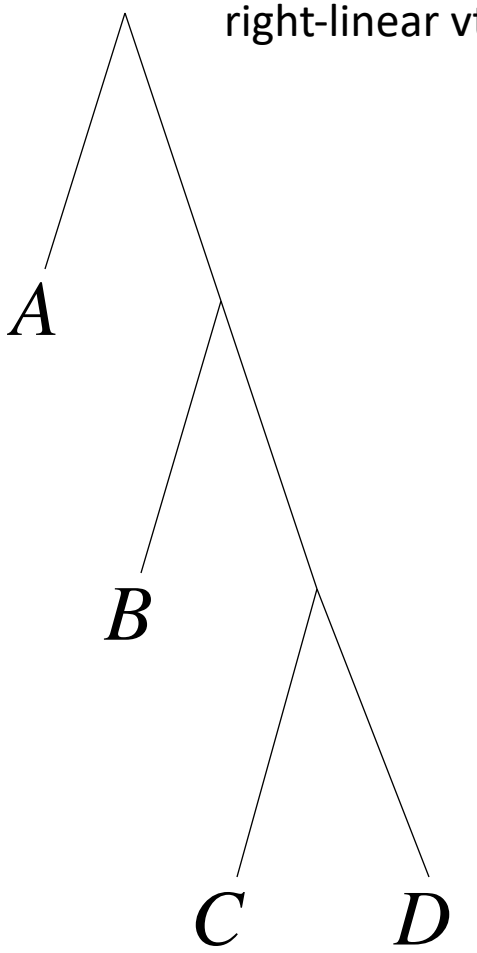
- p1, p2, p3 are called **primes** and represented by SDDs
- s1, s2, s3 are called **subs** and represented by SDDs

Sentential Decision Diagram (SDD)

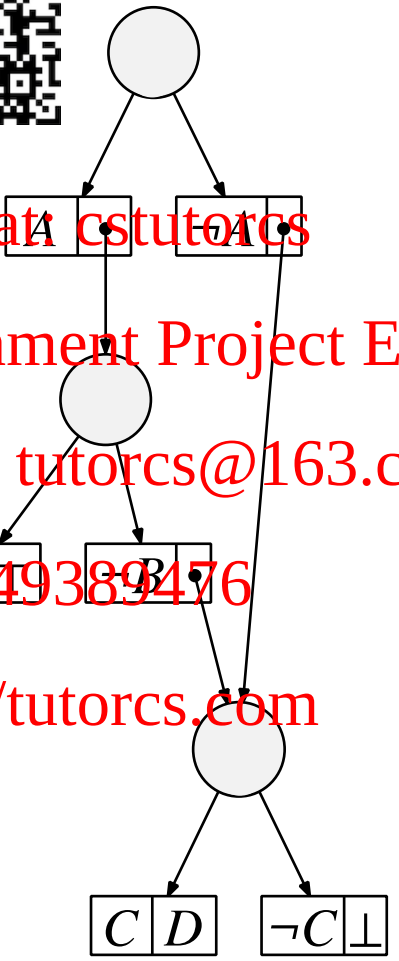


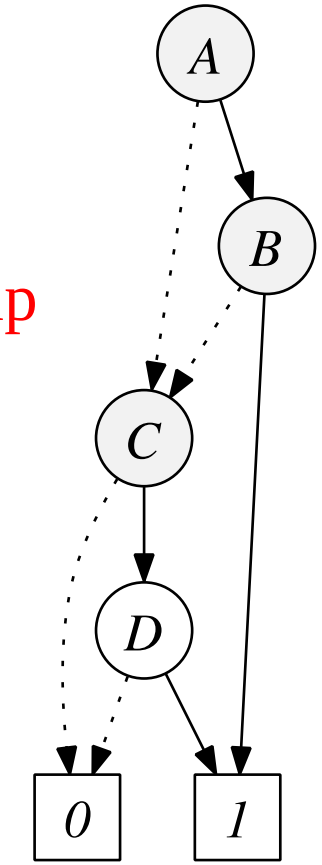


OBDDs are SDDs



right-linear vtree





Punch Line(s)

- SDD a strict superset of
  - Characterized by trees, include orders
  - Branch over sentences, instead of variables
- SDDs maintain key properties of OBDDs:
  - Canonical, Polytime Apply operation*
- SDDs: treewidth, OBDD: pathwidth
- SDDs can be exponentially smaller than OBDDs
- SDD package (open source)

Boolean Function Decompositions

Decomposability

$$f(ABCD) = [g_1(AB) \wedge h_1(CD)] \vee [g_2(A) \wedge h_2(BCD)] \vee [g_3(ACD) \wedge h_3(B)]$$

$$f(ABCD) = [g_1(AC) \wedge h_1(BD)] \vee [g_2(AC) \wedge h_2(BD)] \vee [g_3(AC) \wedge h_3(BD)]$$

decomposability

structured decomposability

DNNF

structured DNNF

Determinism

$$
\begin{array}{c|l}
 & f(ABCD) = \\
f_1 & [g_1(AB) \wedge h_1(CD)] \vee \\
f_2 & [g_2(A) \wedge h_2(BCD)] \vee \\
f_3 & [g_3(ACD) \wedge h_3(B)]
\end{array}
$$

$$f_1 \wedge f_2 = false$$
$$f_1 \wedge f_3 = false$$
$$f_2 \wedge f_3 = false$$

d-DNNF
structured d-DNNF

Partitioned Determinism

(**X**,**Y**)-partition: subset of $(\mathbf{X} \times \mathbf{Y})$ composition

$$f(\mathbf{X}, \mathbf{Y}) = g_1(\mathbf{X})h_1(\mathbf{Y}) + \ldots + g_n(\mathbf{X})h_n(\mathbf{Y})$$

$$g_i \wedge g_j = false$$

$$g_1 \vee \ldots \vee g_n = true$$

$$f = \underbrace{g_1(\mathbf{X})}_{prime}\underbrace{h_1(\mathbf{Y})}_{sub} + \ldots + \underbrace{g_n(\mathbf{X})}_{prime}\underbrace{h_n(\mathbf{Y})}_{sub}$$

$$f = (g_1, h_1) \ldots (g_n, h_n)$$

(X,Y)-Partition

$$f = (A \wedge B \wedge C) \vee (C \wedge D)$$

$$\mathbf{X} = \{A, B\}, \quad \mathbf{Y} = \{C, D\}$$

$$f = \underbrace{(A \wedge B)}_{prime}\underbrace{(true)}_{sub} + (\overline{A} \wedge B)(C) + (\overline{B})(C \wedge D)$$

Compression

- (**X,Y**)-partition is compre[illegible]no equal subs:

$$f = (g_1, h_1) \ldots (g_n, h_n)$$

$$h_i \neq h_j \text{ for all } i \neq j$$

- A function f(**X**,**Y**) has a unique compressed (**X,Y**)-partition

Compression

$$f = (A \wedge (B \wedge C) \vee (C \wedge D)$$

$$\mathbf{X} = \{A, B\}, \quad \mathbf{Y} = \{C, D\}$$

prime	sub
$A \wedge B$	
$A \wedge \overline{B}$	
$\overline{A} \wedge B$	
$\overline{A} \wedge \overline{B}$	

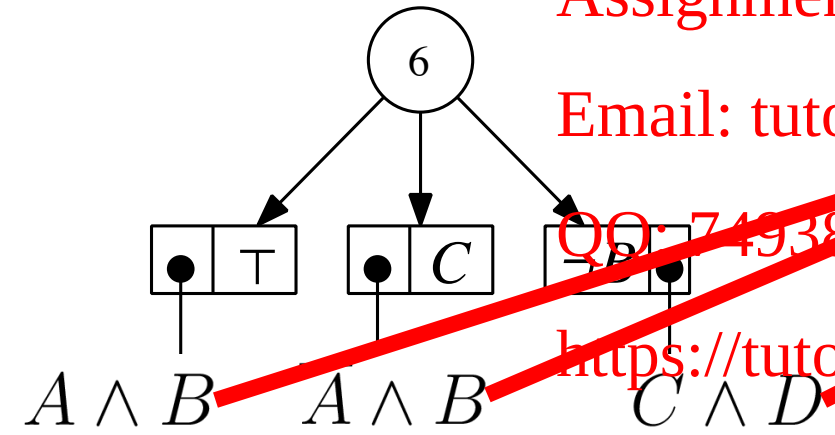
prime	sub
$A \wedge B$	true
$A \wedge B$	C
$\overline{B}$	$C \wedge D$

Canonicity

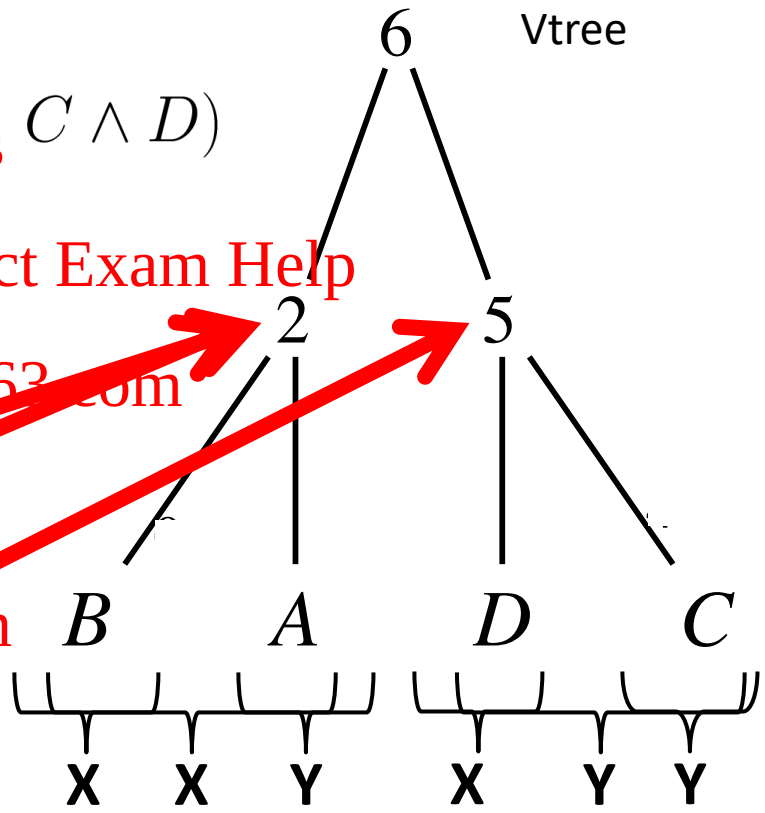
$$f = (A \wedge B \; C) \vee (C \wedge D)$$

(X,Y)-Partition:

$$f = (A \wedge B \;,\; true) \;\; (\overline{A} \wedge B,\; C) \;\; (\overline{B},\; C \wedge D)$$

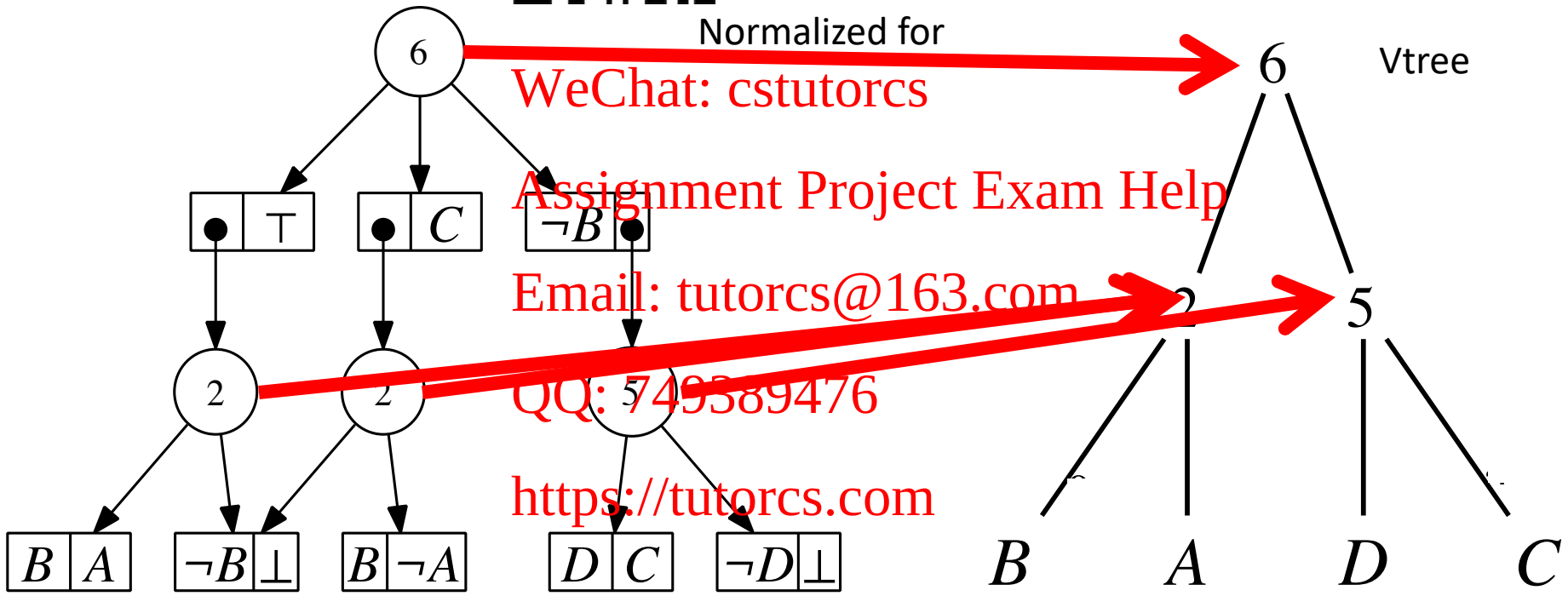


(B, A) $(\overline{B}, false)$



Canonicity

$$f = (A \wedge B) \vee (C \wedge D)$$

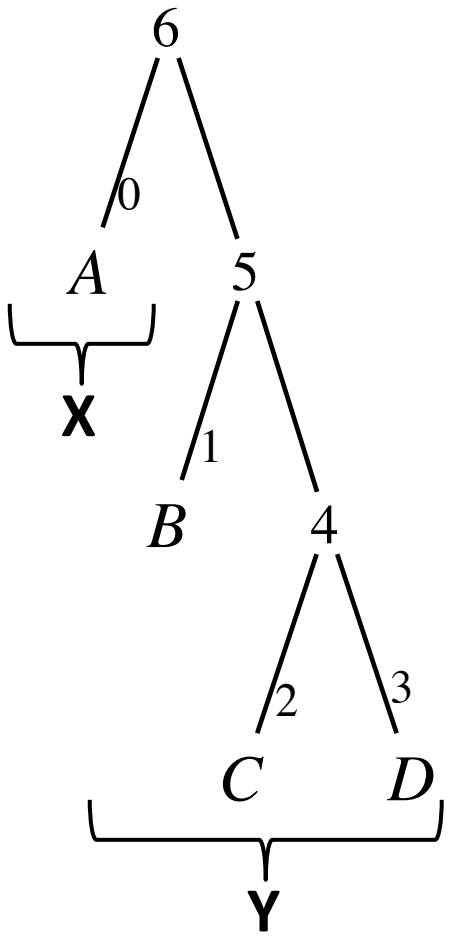


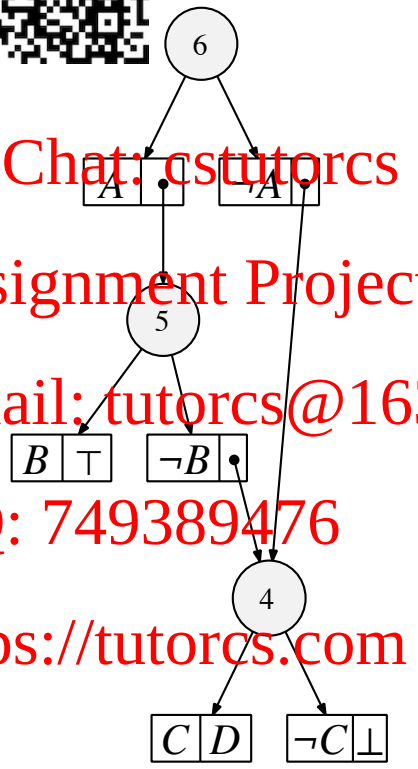
OBDDs are SDDs

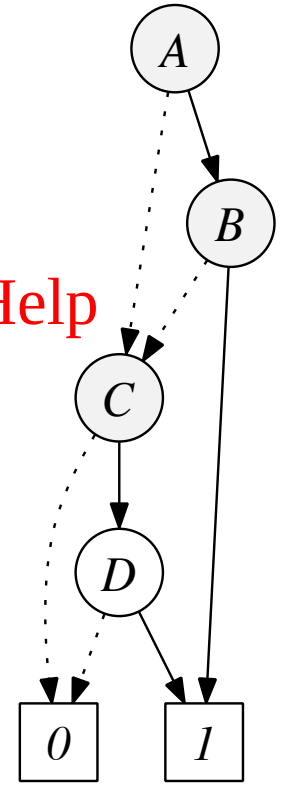
({X},**Y**)-partiti  hannon decomposition

$$f = \underbrace{(X)}_{prime}\underbrace{(f|X)}_{sub} + (\overline{X})(f|\overline{X})$$

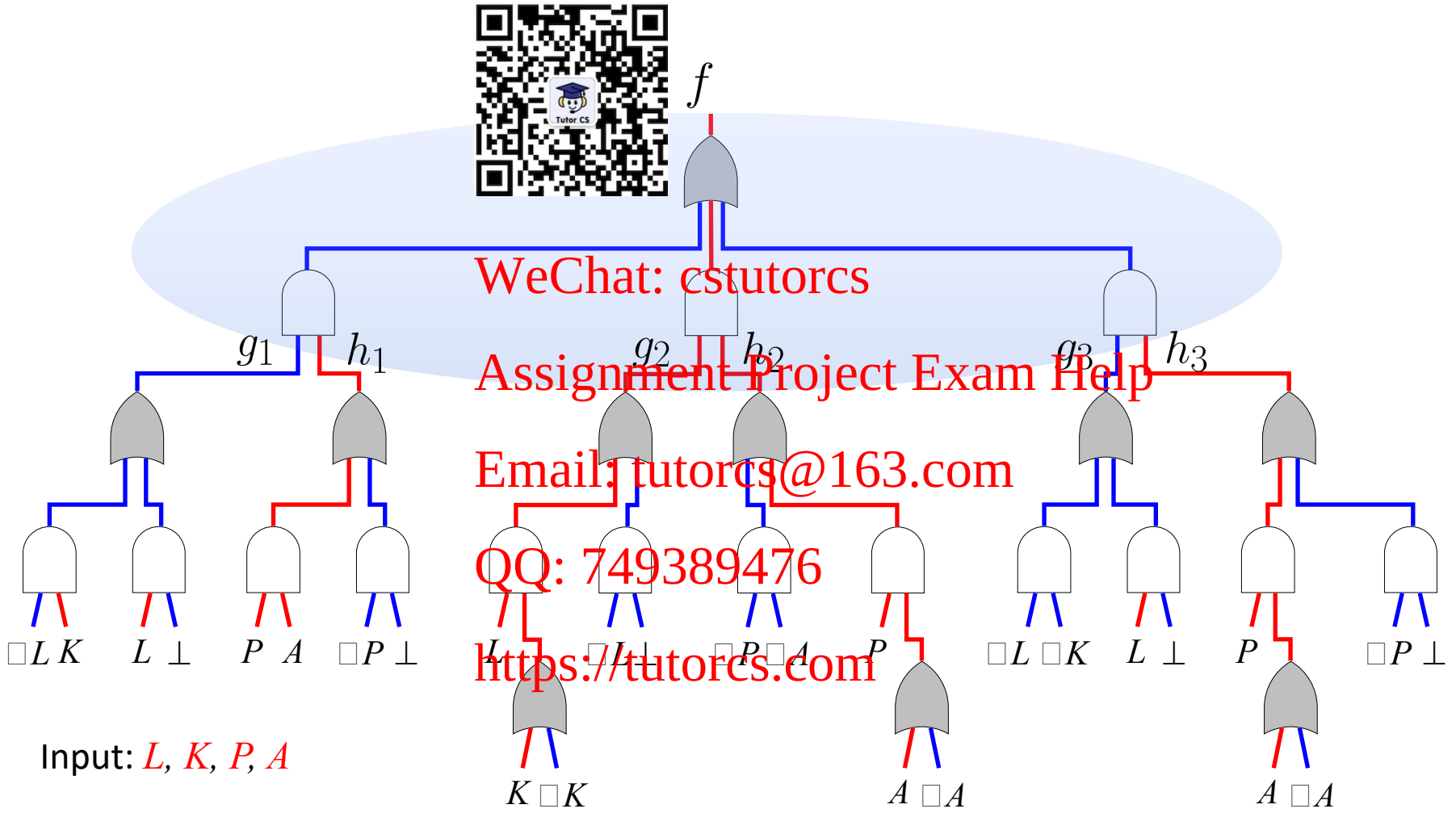
OBDDs are SDDs



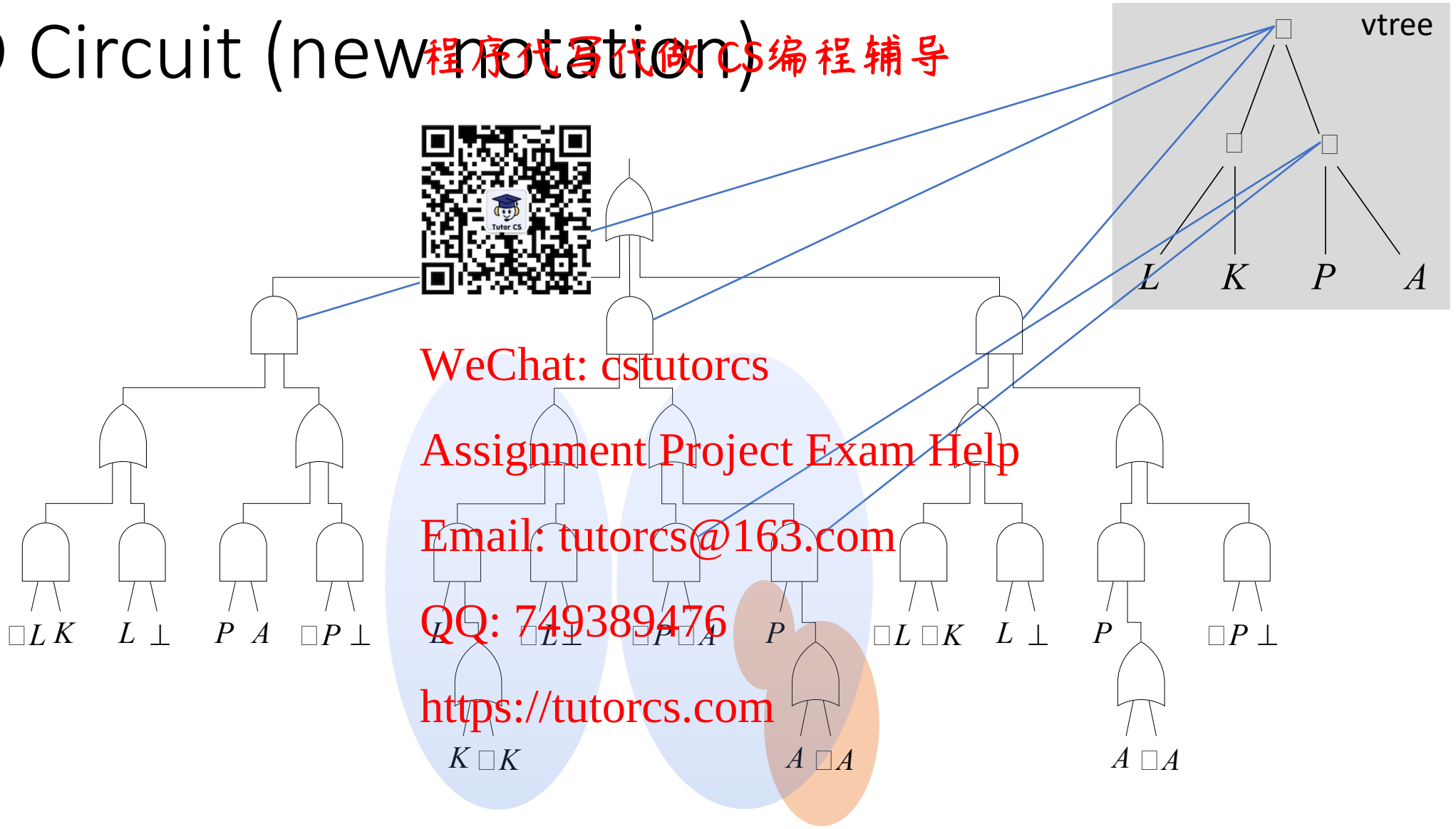




SDD Circuit (new notation)



SDD Circuit (new notation)



Polytime Apply Operation

$(\mathbf{X}, \mathbf{Y})$–partiti

$(p_1, q_1) \ldots (p_n, q_n)$

$(\mathbf{X}, \mathbf{Y})$–partition of g:

$(r_1, s_1) \ldots (r_m, s_m)$

$(\mathbf{X}, \mathbf{Y})$–partition of $f \circ g$:

$$(p_i \wedge r_j \ , \ q_i \circ s_j) \mid p_i \wedge r_j \neq false$$

compression not guaranteed

Bottom-Up Compilation

(OBDD/SDD)

Bottom-Up Compilation

- To compile a CNF:
  - SDD for literals
  - Disjoin literals to compile clause
  - Conjoin clauses to compile CNF
- Similar procedure to compile DNF
- Works for any Boolean formula (not just CNF/DNF)

Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs
using Boolean operators

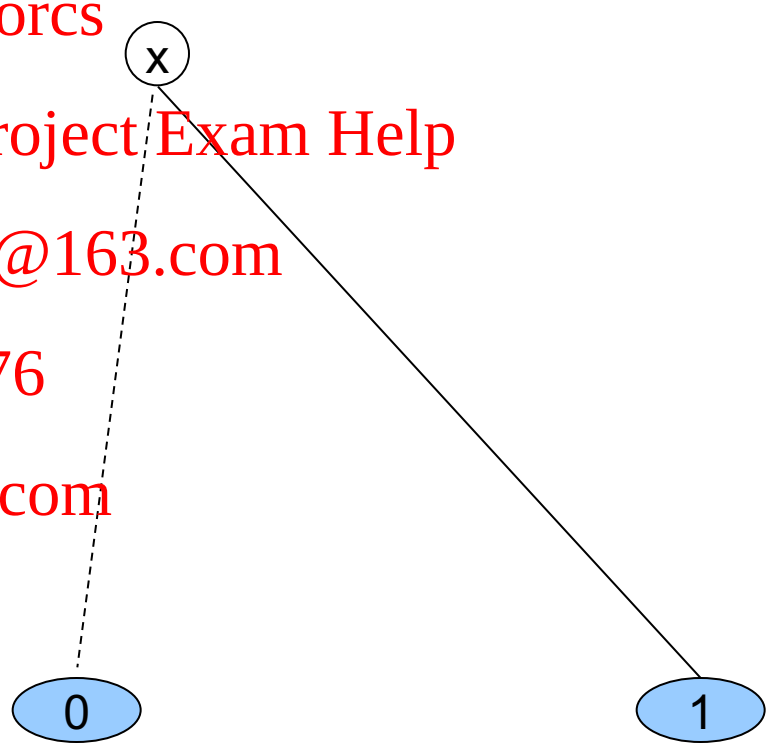
Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs
using Boolean operators



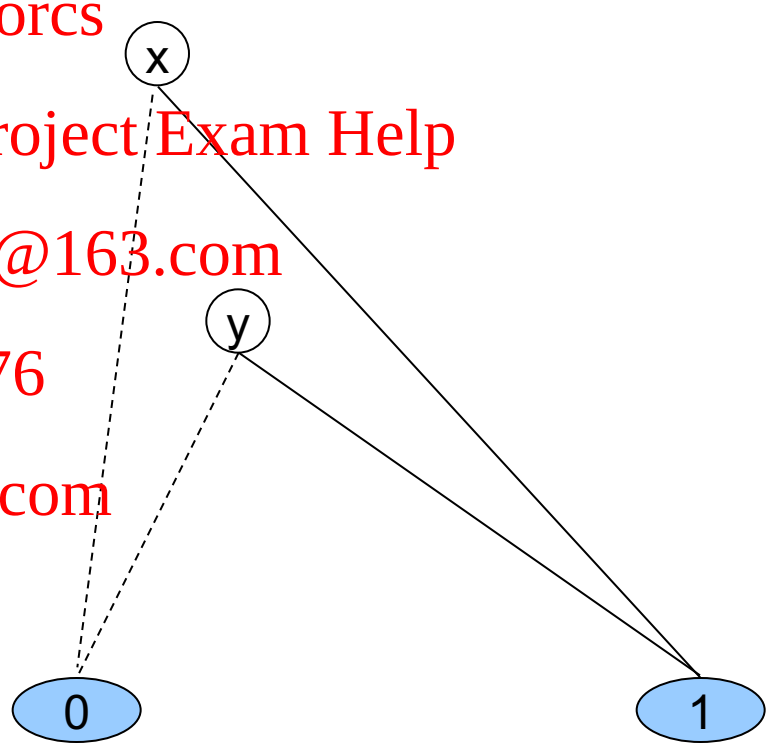
Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs
using Boolean operators



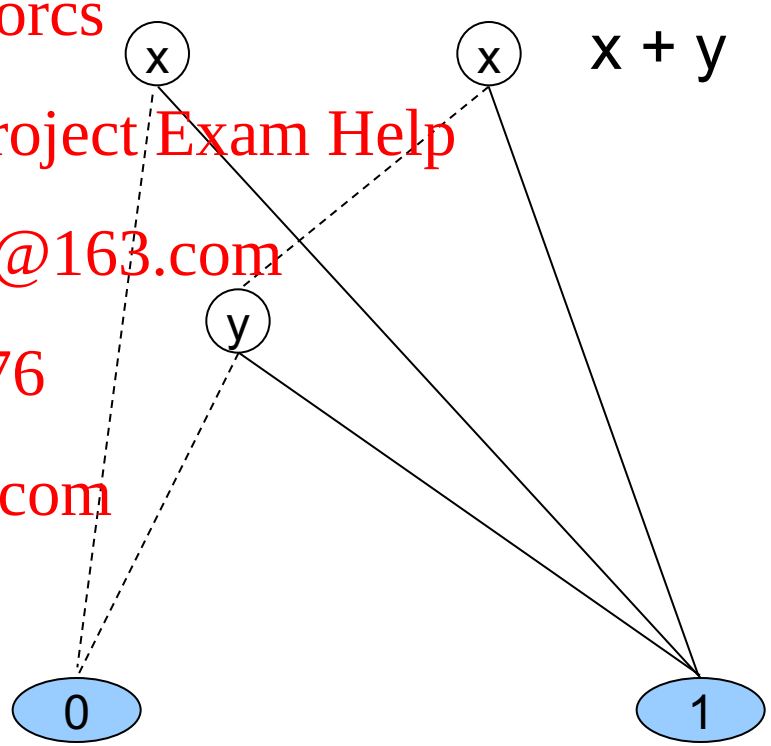
Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs
using Boolean operators



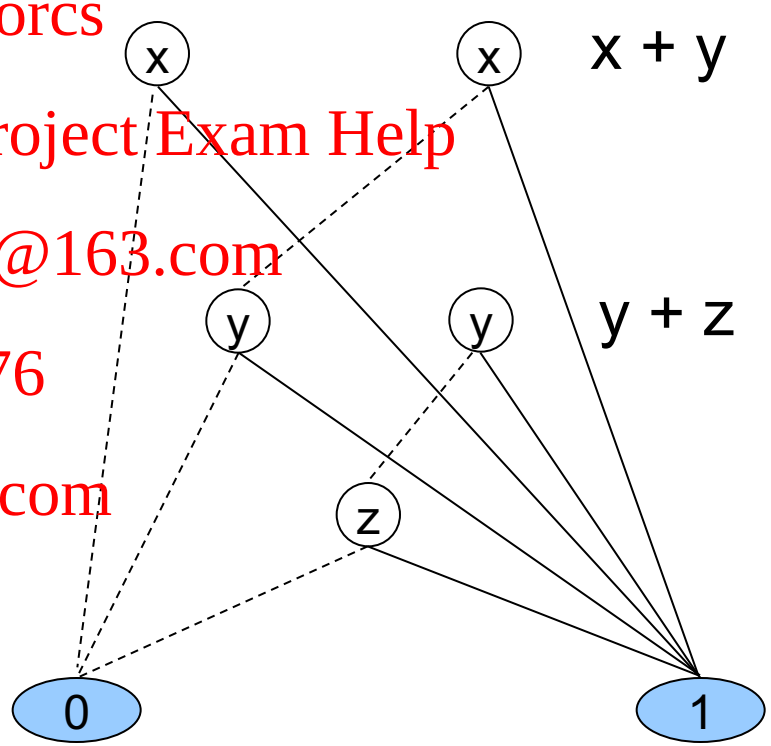
Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs
using Boolean operators



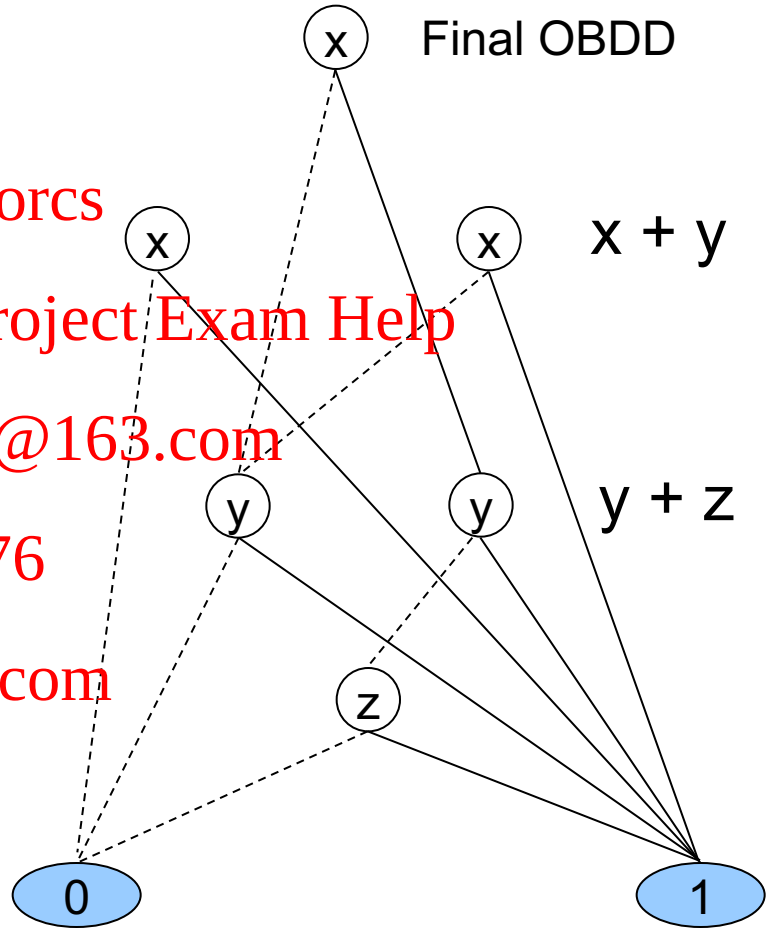
Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs
using Boolean operators



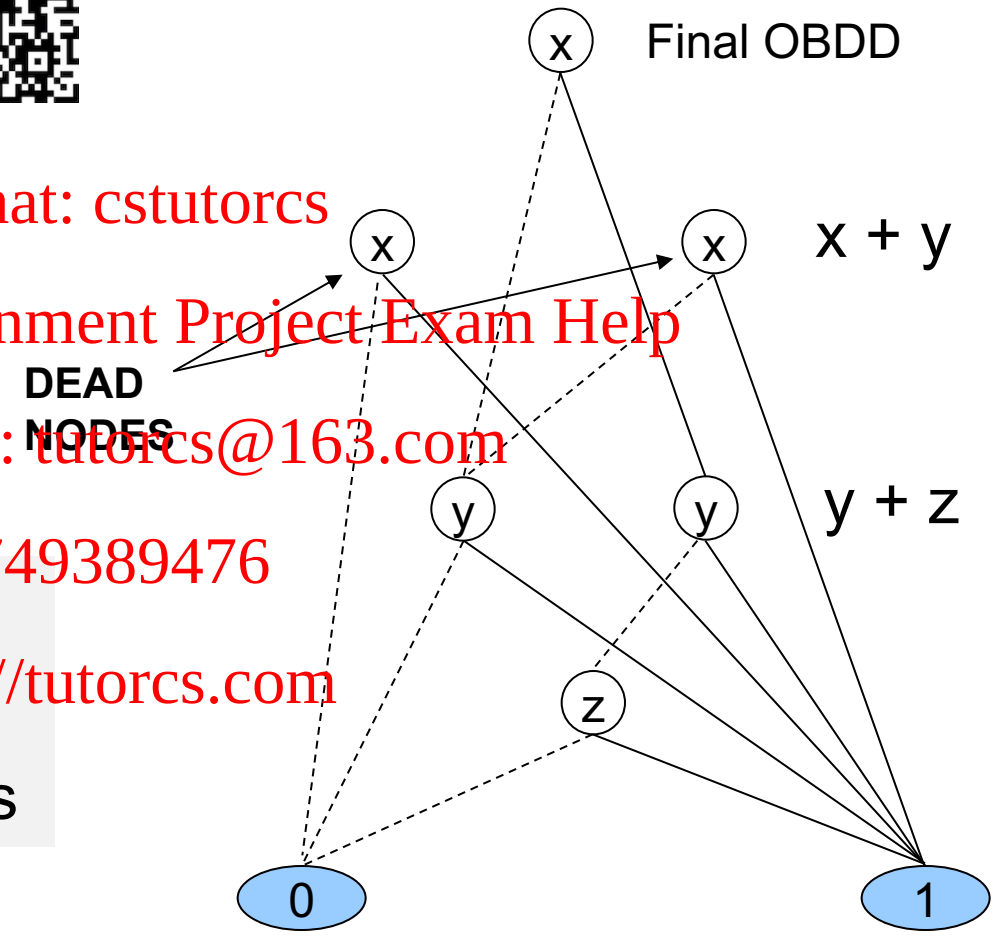
Bottom-up Compilation

CNF: (x + y) (y + z)

Variable order: x, y, z

Apply:

combines two OBDDs using Boolean operators



Bottom-up Compilation

- Requires:
  - *Apply* (conjoin, disjoin, etc)
  - Garbage collection of dead nodes
- Challenges:
  - Good variable order
  - Good schedule of Apply operations
- uf100-08 (32 models):
  - 176 nodes in final OBDD under MINCE variable order
  - 30,640,582 intermediate nodes using CUDD package
- Contrast with top-down compilation (space, form of input, incremental compilation)

Canonicity in Compilation

- OBDDs are canonical
  variable order → unique OBDD

  (reduced OBDDs)

- SDDs are canonical
  vtree → unique SDD

  (trimmed and compressed SDDs)

  trimmed: $(\top, \alpha) \rightarrow \alpha, \quad (\alpha, \top)(\neg\alpha, \bot) \rightarrow \alpha$

- Minimizing size amounts to finding best variable order or vtree

Vtrees (Variable Orders) Matter

- A vtree can have a significant impact on the size of an SDD
- Good vtrees can be obtained either
  - **Statically**: by analyzing the Boolean function structure before compilation
  - **Dynamically**: by searching for an appropriate vtree during compilation

Minimizing OBDD Size

- OBDDs are characteri[illegible] total variable orders (*n!* orders for *n* variables)

ABCD, CADB, ...

- 24 total orderings of 4 variables

- 24 OBDDs for every function over 4 variables

- Searching for an optimal OBDD is a search over the space of total variable orders

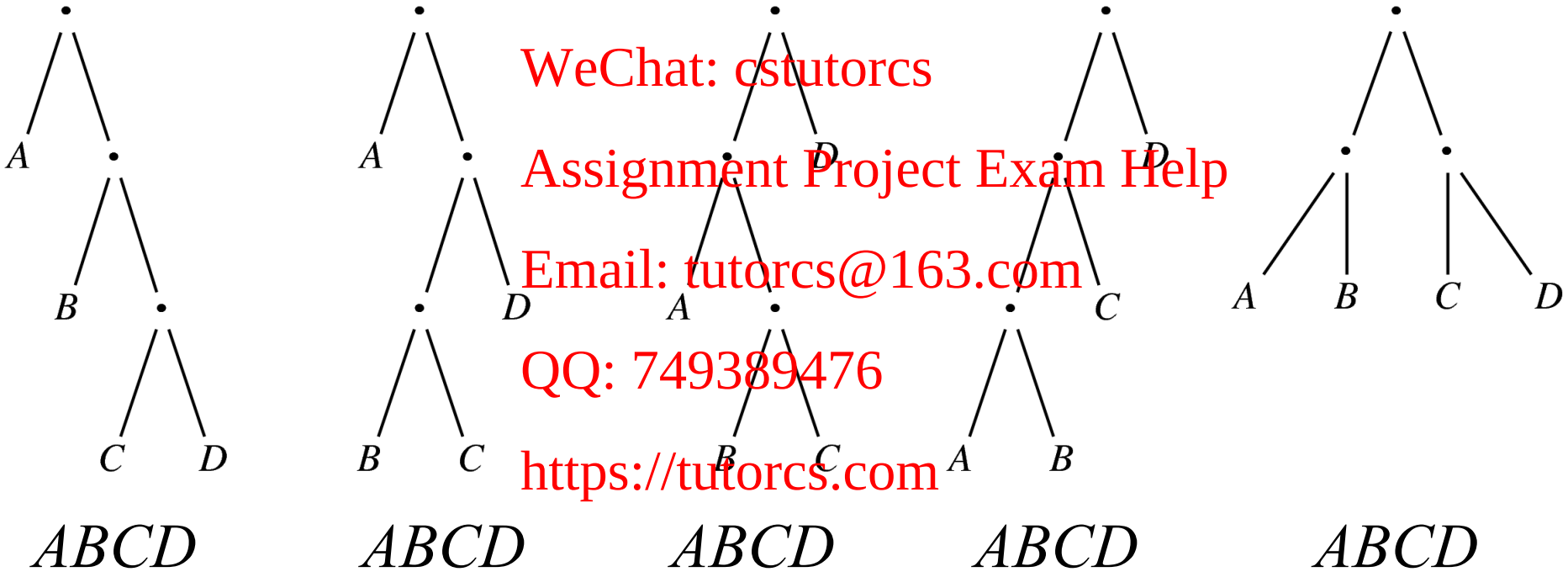
Minimizing OBDD Size

$$ABCD \Rightarrow ABDC \Rightarrow ADBC \Rightarrow DABC \Rightarrow DACB \Rightarrow ADCB \Rightarrow$$
$$ACDB \Rightarrow ACBD \Rightarrow CABD \Rightarrow CADB \Rightarrow CDAB \Rightarrow DCAB \Rightarrow$$
$$DCBA \Rightarrow CDBA \Rightarrow CBDA \Rightarrow CBAD \Rightarrow BCAD \Rightarrow BCDA \Rightarrow$$
$$BDCA \Rightarrow DBCA \Rightarrow DBAC \Rightarrow BDAC \Rightarrow BADC \Rightarrow BACD \Rightarrow$$

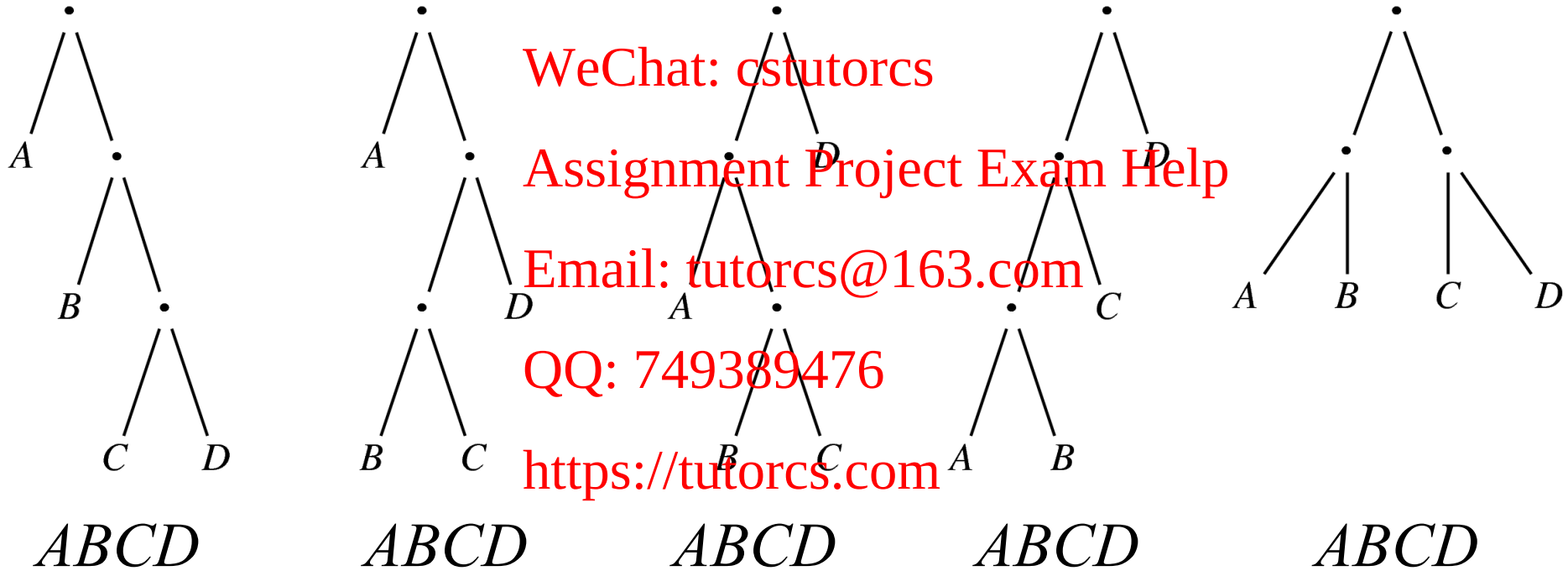
Searching for Vtrees (dynamically)

A Vtree Embeds a Variable Order

Obtained from a le[illegible]ersal of the vtree



A Vtree Dissects a Variable Order



How Many Vtrees?

# of variable orderings over n variables

$$n!$$

# of dissections over n variables:

$$C_{n-1} = \frac{(2(n-1))!}{n!(n-1)!}$$

C_{n-1} is the # of full binary trees with n leaves (Catalan number)

# of vtrees over n variables:

$$n! \times C_{n-1} \quad = \quad n! \times \frac{(2(n-1))!}{n!(n-1)!} \quad = \quad \frac{(2(n-1))!}{(n-1)!}$$

How Many Vtrees?

n	1	2	3	4	5	6
# of orderings	1	2	6	24	120	720
# of dissections	1	1	2	5	14	42
# of vtrees	1	2	12	120	1680	30240

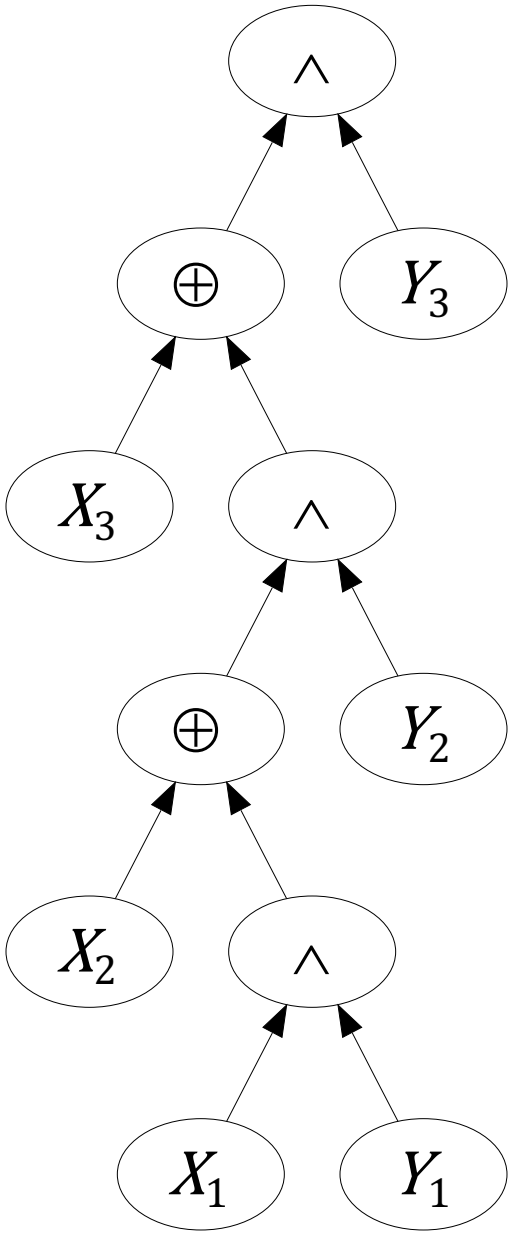
Vtrees Matter

- The choice of a vtree can lead to exponential differences in the size of an SDD (implied by what we know about OBDDs)
- The choice of a dissection can also lead to exponential differences in the size of an SDD

The Power of Dissection

There is a class of Boolean functions and a corresponding variable order π such that

- The right-linear dissection of order π leads to an SDD (OBDD) of exponential **exponential** size
- Some other dissection of order π leads to an SDD that has **linear** size



variables: $\mathbf{X} = \{ X_1\ X_2\ X_3 \}$

$\mathbf{Y} = \{ Y_1\ Y_2\ Y_3 \}$

if $\mathbf{x} \neq \mathbf{x}'$, then $f|_{\mathbf{x}} \neq f|_{\mathbf{x}'}$

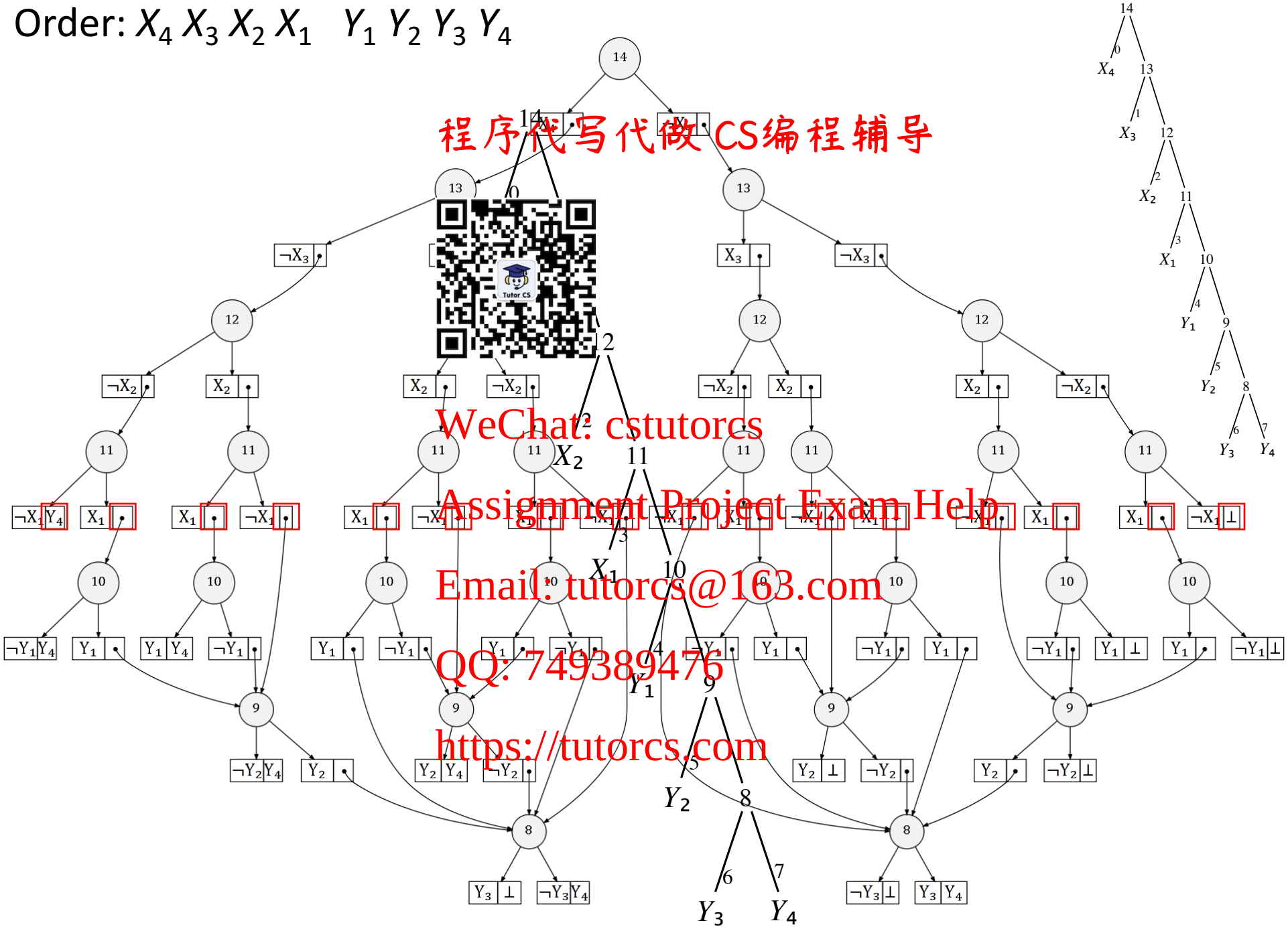
Sieling & Wegener bound

implies exponential OBDD

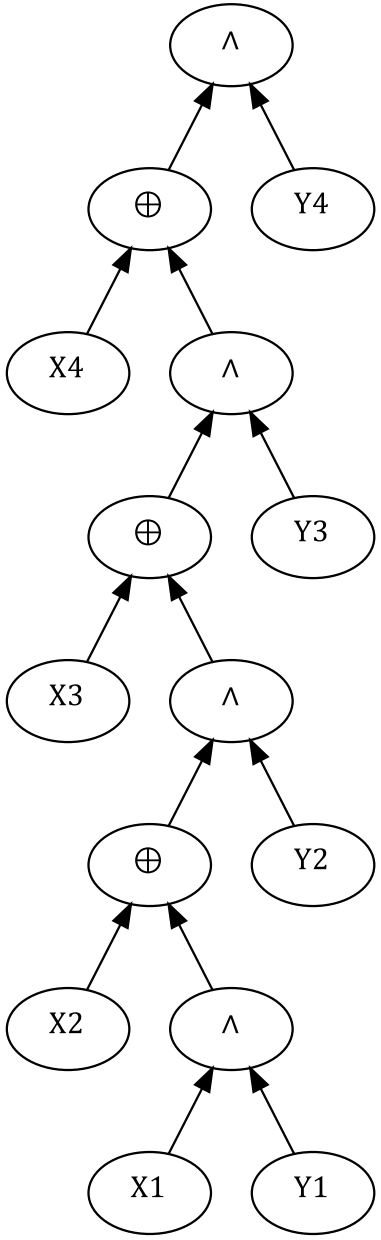
for any ordering that places

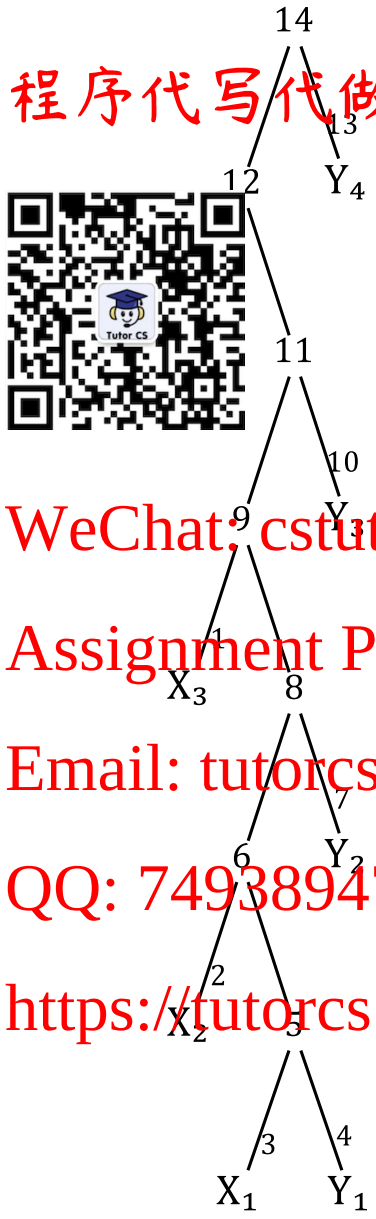
variables **X** before **Y**

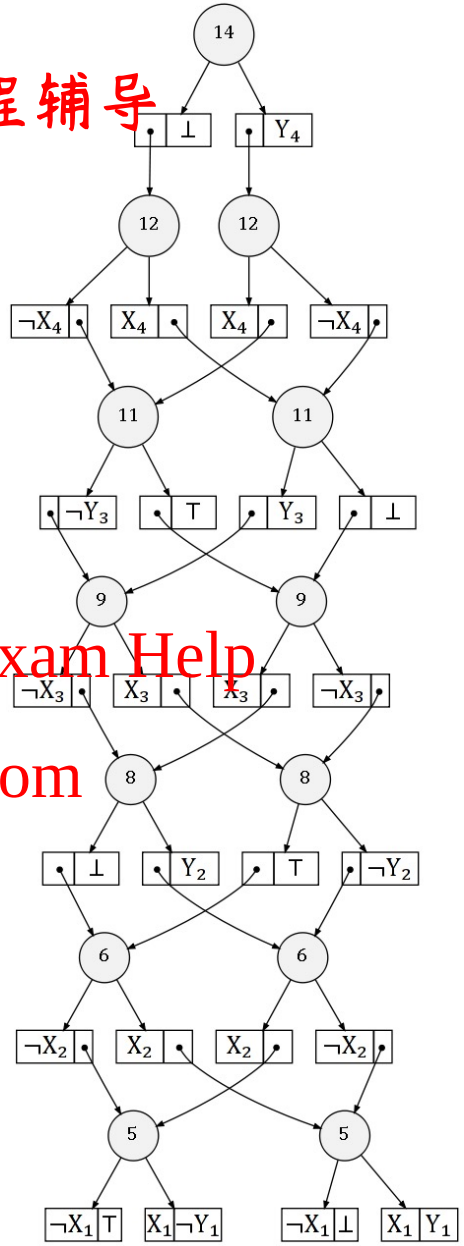
Order: $X_4\ X_3\ X_2\ X_1 \quad Y_1\ Y_2\ Y_3\ Y_4$



Order: $X_4\ X_3\ X_2\ X_1\quad Y_1\ Y_2\ Y_3\ Y_4$



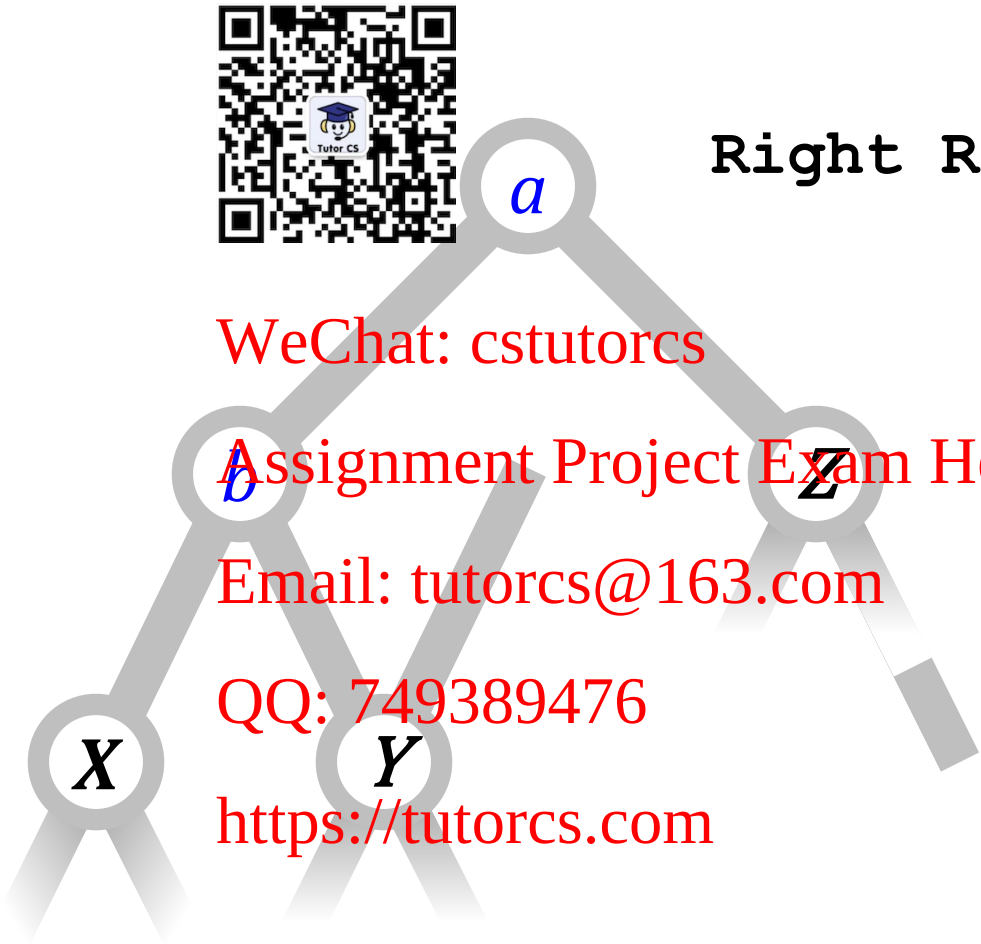




Searching Over Vtrees

- Double search problem
  - Find variable order
  - Find dissection
- Tree operations:
  - Rotation
  - Swapping
- Can enumerate all vtrees

Tree Rotations



Right Rotate(a)

Tree Rotations

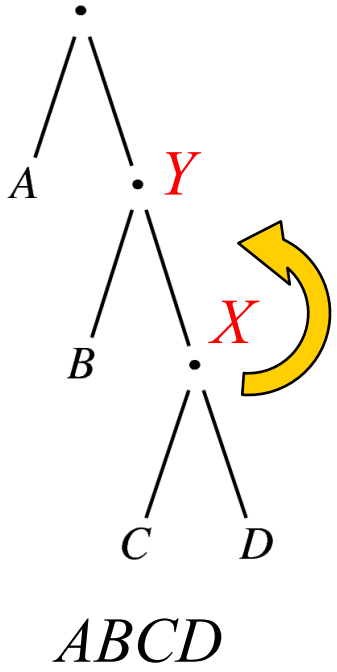
Left Rotate(a)

Rotation Preserves Variable Order

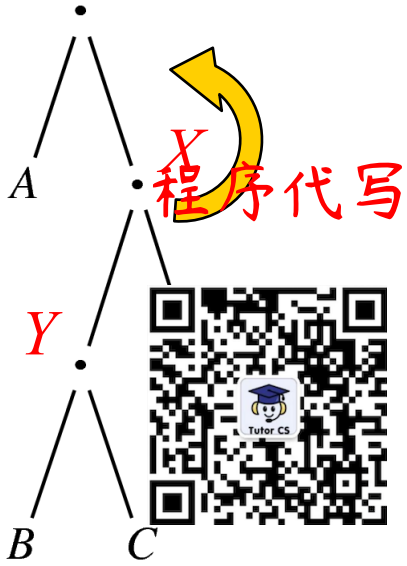
X, Y, Z X, Y, Z

Rotation Enumerates all Dissections

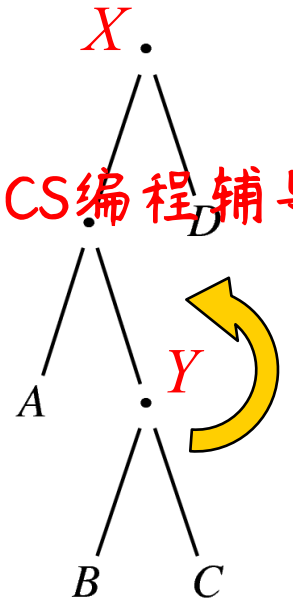
- Rotations can enumerate all dissection of a given variable order
- Systematic methods exist for this purpose
- See, e.g., Knuth's
  Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees



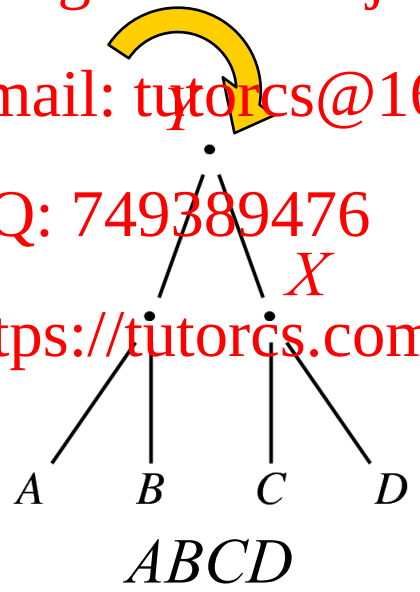
ABCD



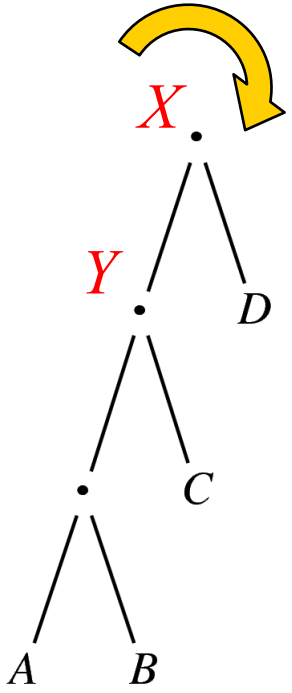
ABCD



ABCD

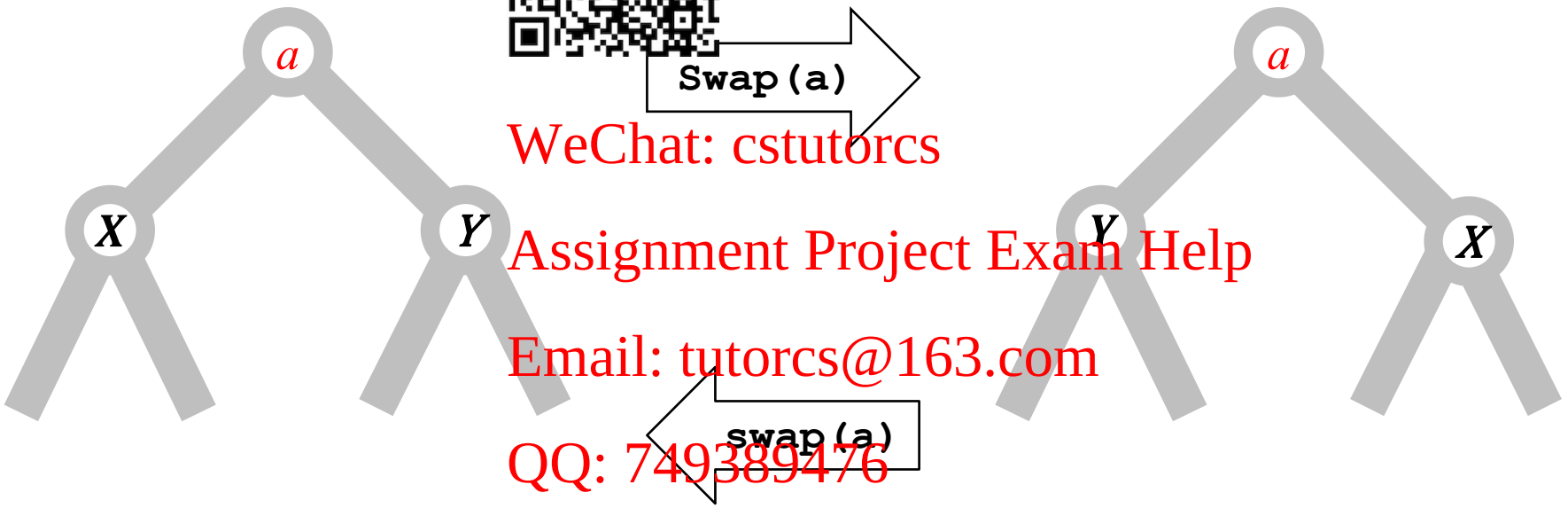


ABCD



ABCD

Tree Swapping



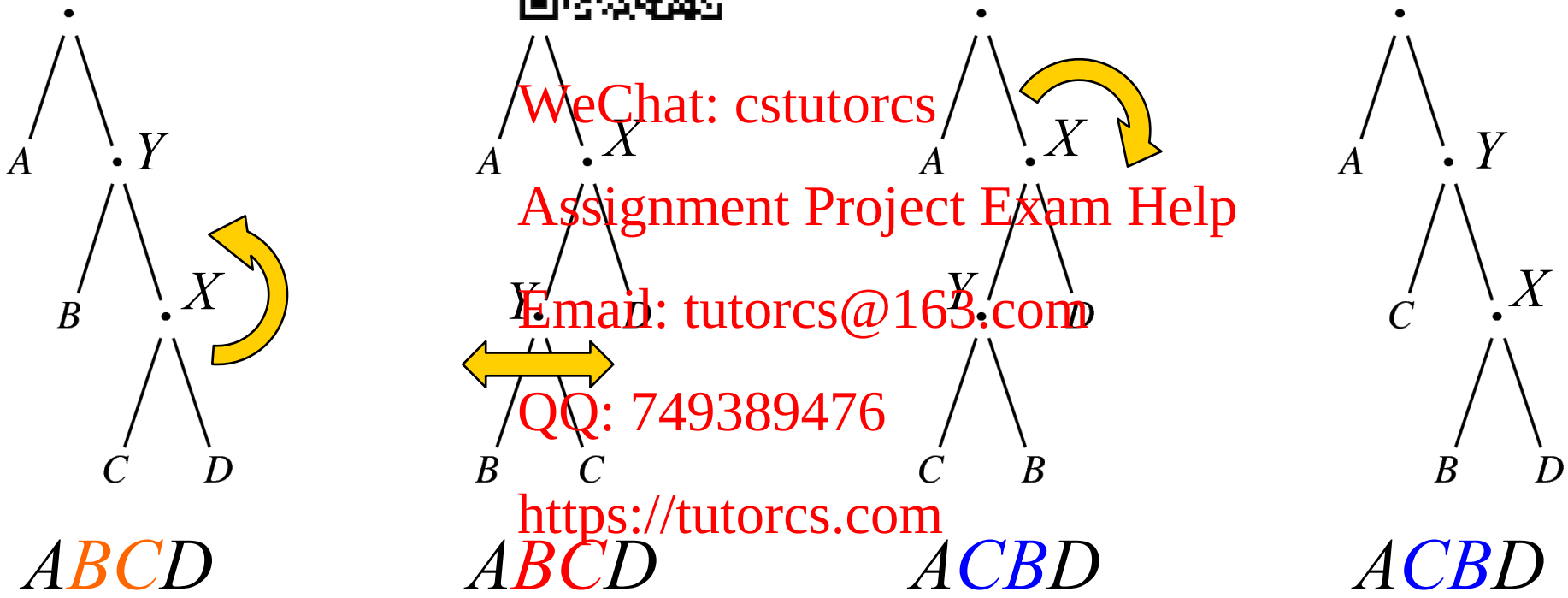
X, Y

Y, X

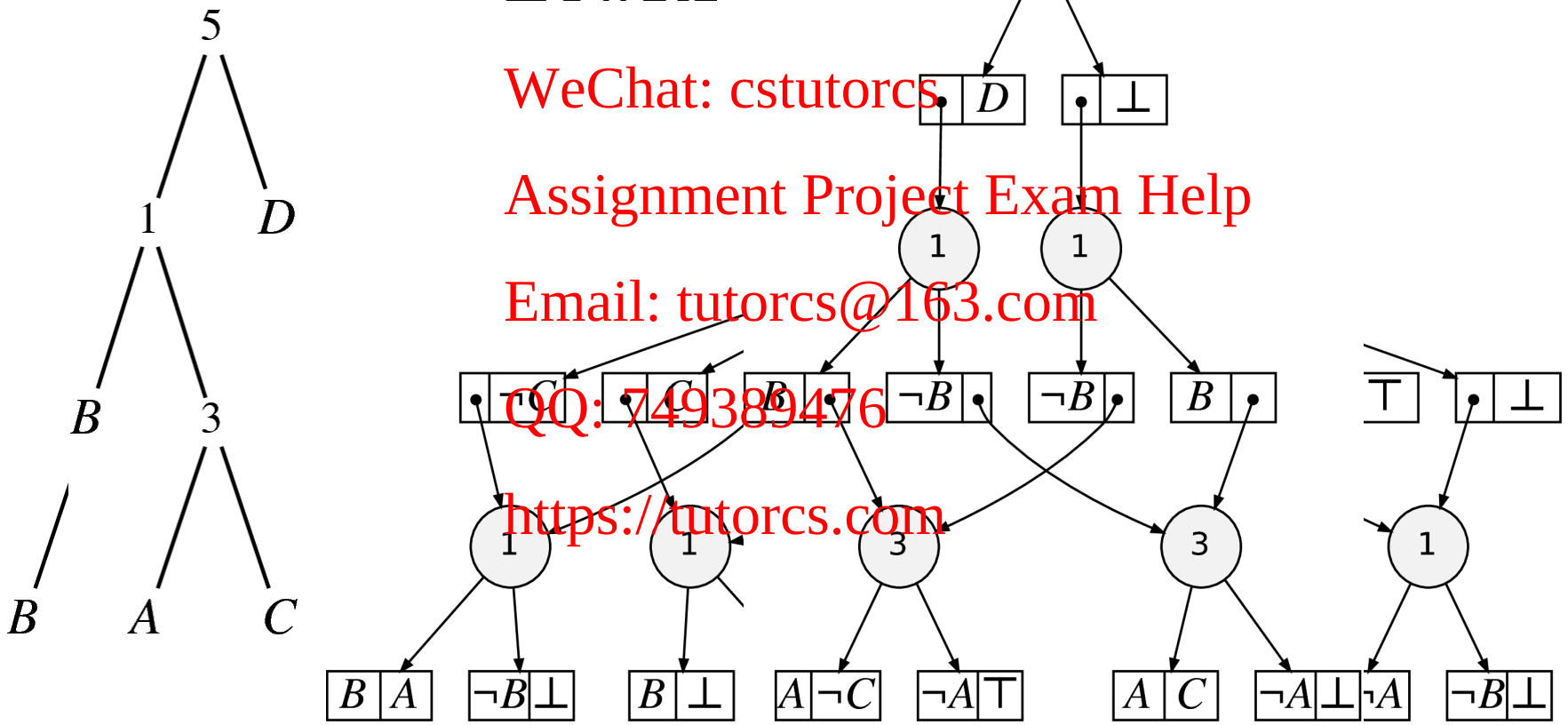
$$f(\mathbf{X}, \mathbf{Y}) = \sum \underbrace{p(\mathbf{X})}_{prime} q(\mathbf{Y})$$

$$f(\mathbf{X}, \mathbf{Y}) = \sum \underbrace{q(\mathbf{Y})}_{!prime} p(\mathbf{X})$$

Rotation + Swapping



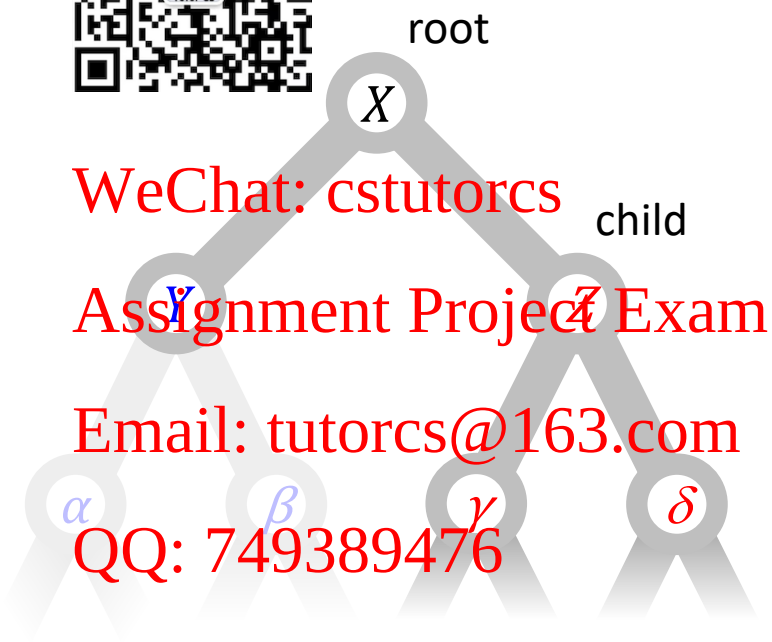
The SDD Package



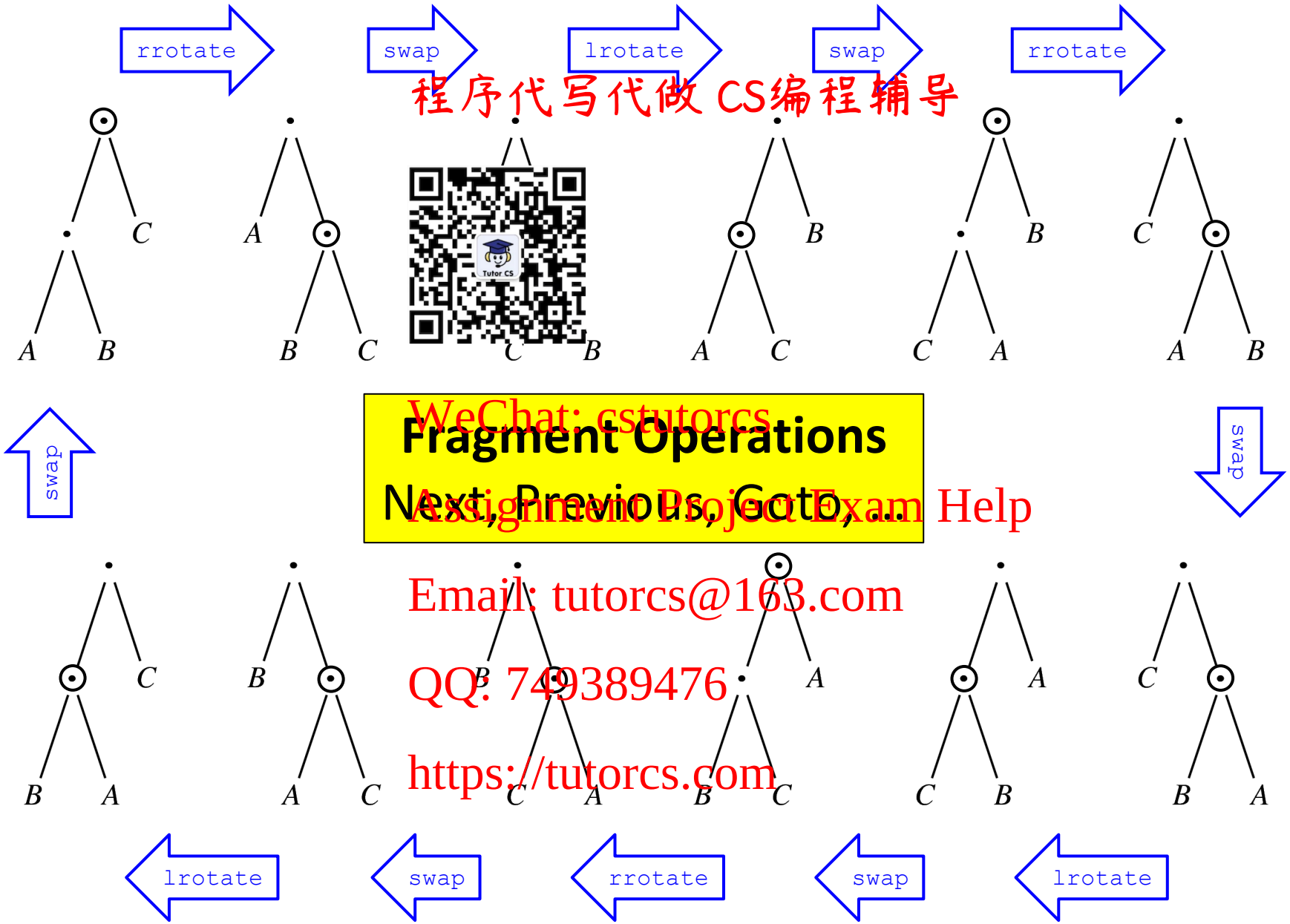
Vtree Fragments

left-linear fragment

Vtree Fragments



right-linear fragment

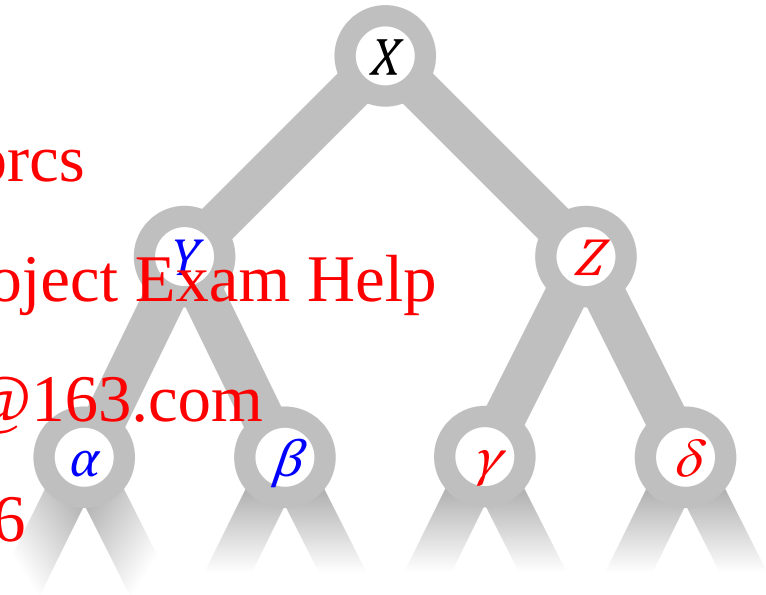


Fragment Operations

Next, Previous, Cctor, ...

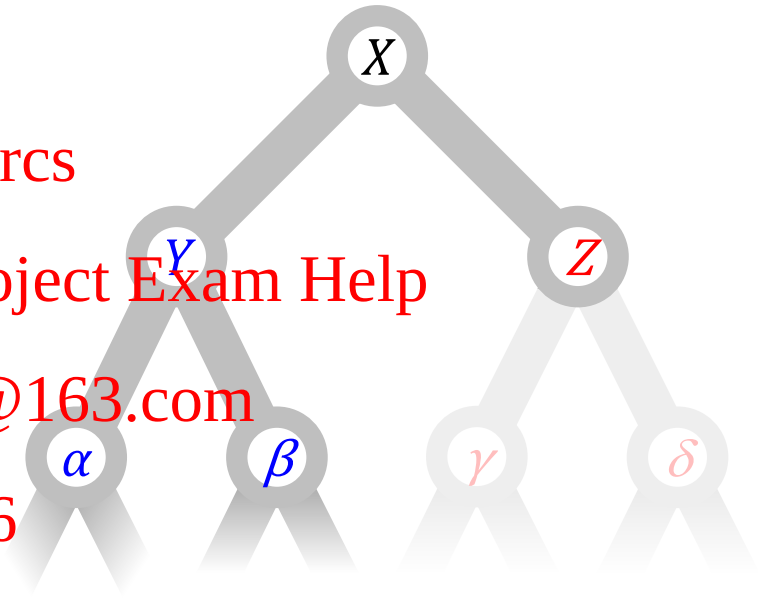
Greedy Search

- Enumerate all
  over a “window



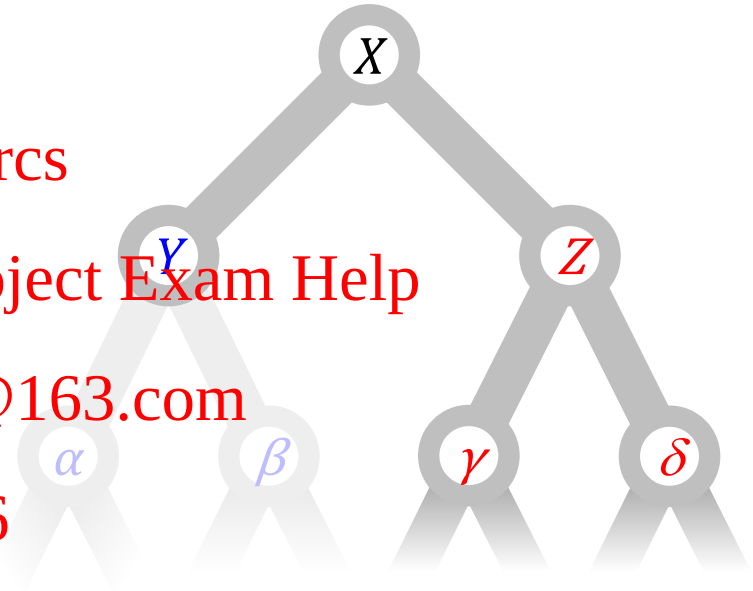
Greedy Search

- Enumerate all trees over a “window”
- Enumerate 12 trees



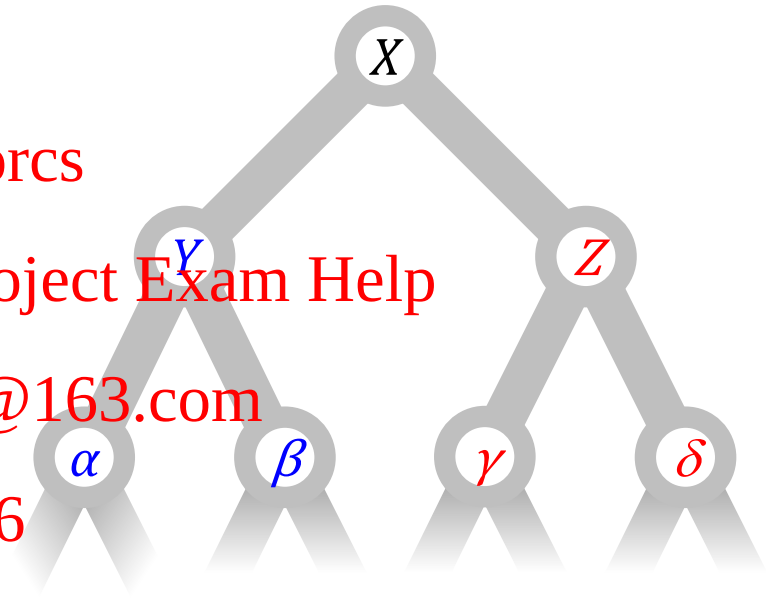
Greedy Search

- Enumerate all trees over a “window”
- Enumerate 12 vtrees
- Enumerate 12 vtrees



Greedy Search

- Enumerate all vtrees over a "window"
- Enumerate 12 vtrees
- Enumerate 12 vtrees
- 24 vtrees in total
- Greedily accept best vtree, move window



The SDD Package

- reasoning.cs.ucla.edu/s
- Written in C (open source)
- Available as a library + code for compiling CNF/DNF into SDDs
- Key primitives:
  - apply_sdd
  - left_rotate, right_rotate, swap
  - garbage_collect, ref, deref
  - dynamic_vtree
- Two manuals (beginner and advanced)

Concluding Remarks

- SDD a strict superset of
  - Characterized by trees, v ude orders
  - Branch over sentences, which include literals
- SDDs maintain key properties of OBDDs:
  - Canonical, Polytime Apply Operation*
- SDDs: treewidth, OBDD: pathwidth
- Dissecting an order can lead to exponential reduction
- SDDs can be exponentially smaller than OBDDs