Large-Scale Knowledge Processing

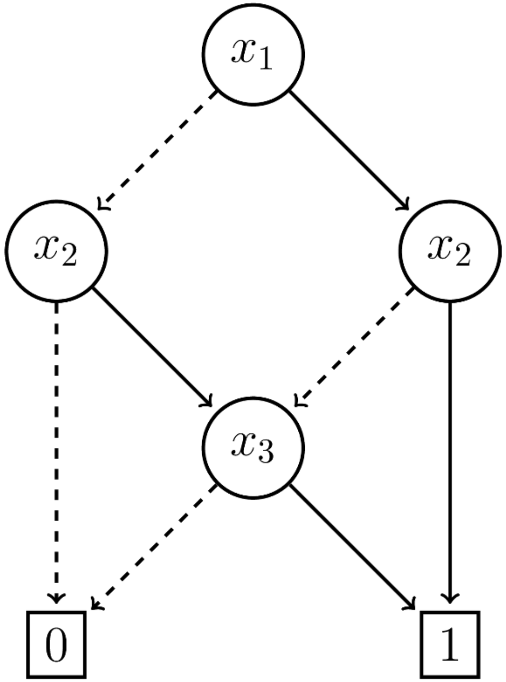
Using Binary Decision Diagrams (1)

Faculty of Information Science
and Technology, Hokkaido Univ.

Takashi Horiyama

Prev. class : Binary Decision Diagram (BDD)

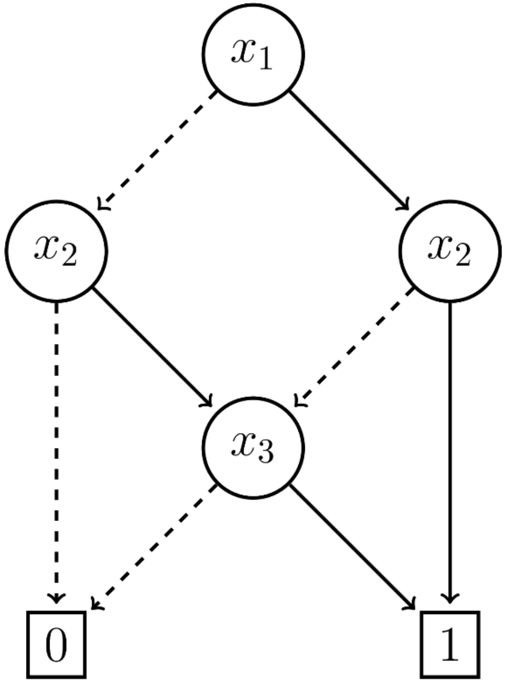
- Representation of Boolean functions by a **directed acyclic graph**
- **Variable order**
  - Variables appear according to a total order
- Two rules for simplifying BDDs
  - Remove **redundant nodes**
  - Share **equivalent nodes**
  - **Reduce** BDDs until we have no redundant and equivalent nodes



c.f. Lecture slides on "Representation of Boolean functions"

Prev. class : Binary Decision Diagram (BDD)

- Representation of Boolean functions by a **directed acyclic graph**
- The **representation** is **unique** if variable order is defined
- Many practical Boolean functions are **compact**ly represented
  (We can efficiently compress logical structures)
- **Efficient operations** for BDDs
  [Bryant 1986]
- Applications in various fields



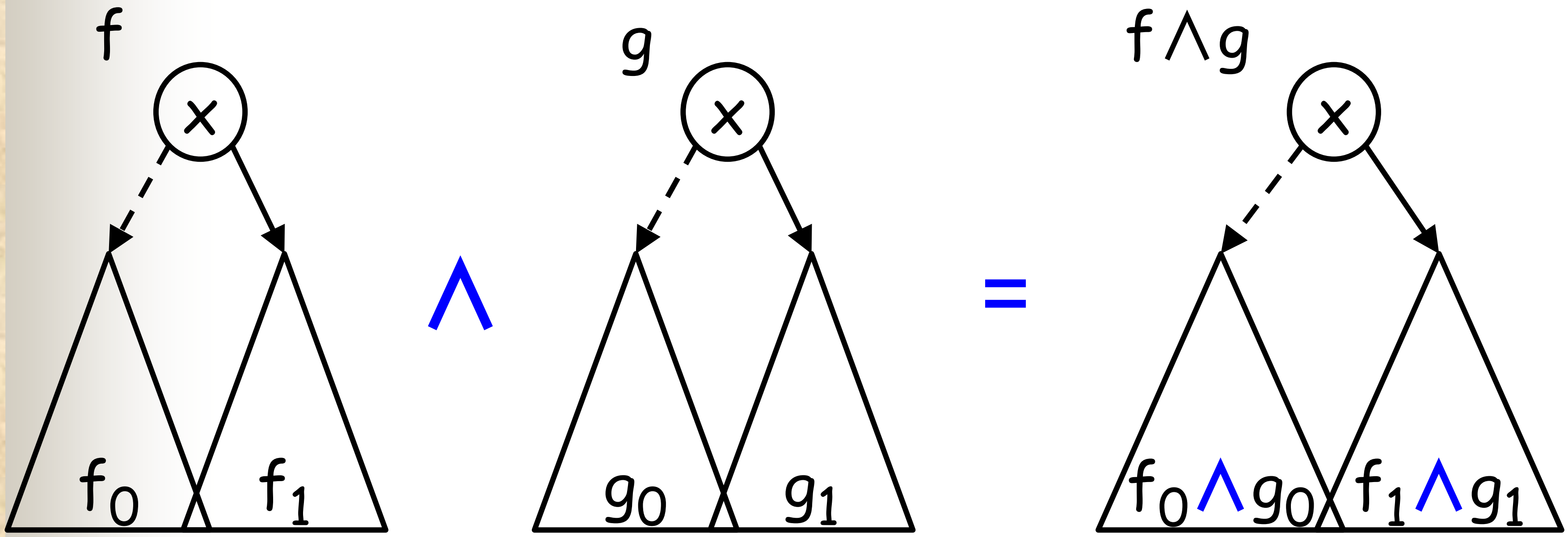
c.f. Lecture slides on "Representation of Boolean functions"

Logical operations (Boolean operations)

- Logical AND of $f = \overline{x} f_0 \vee x f_1$ and $g = \overline{x} g_0 \vee x g_1$

- $f \wedge g = \overline{x}(\underline{f_0 \wedge g_0}) \vee x(\underline{f_1 \wedge g_1})$

We can obtain these ANDs recursively



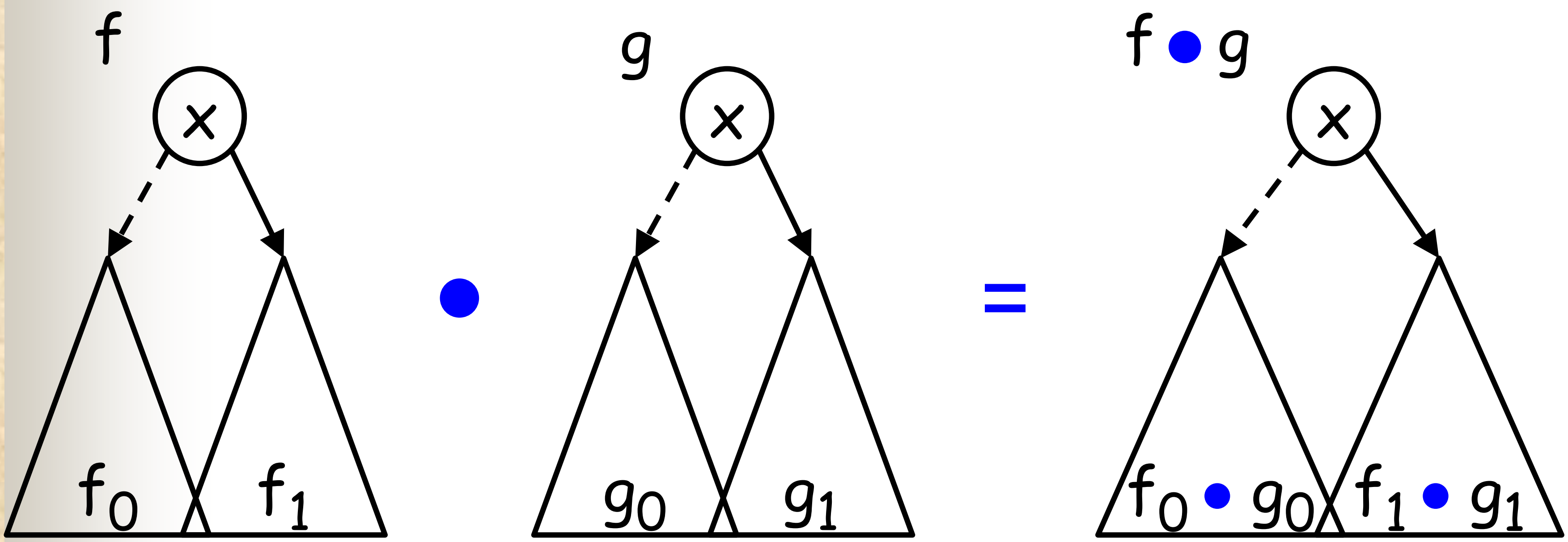
Logical operations (Boolean operations)

AND, OR, XOR, ...

- Binary operation of $f = \overline{x} f_0 \vee x f_1$ and $g = \overline{x} g_0 \vee x g_1$

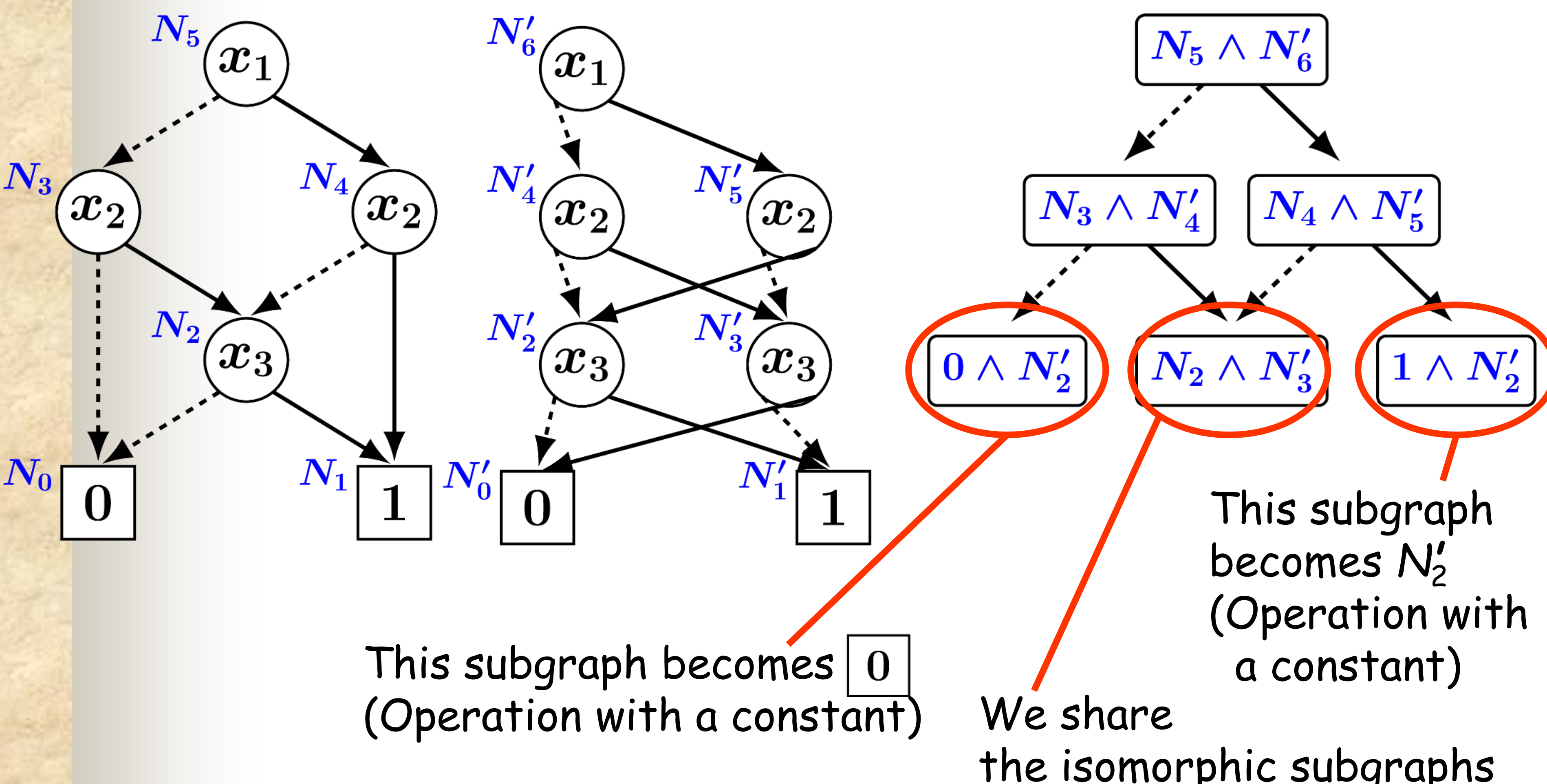
- $f \bullet g = \overline{x} (f_0 \bullet g_0) \vee x (f_1 \bullet g_1)$

We can obtain these recursively



practice : Apply logical operations

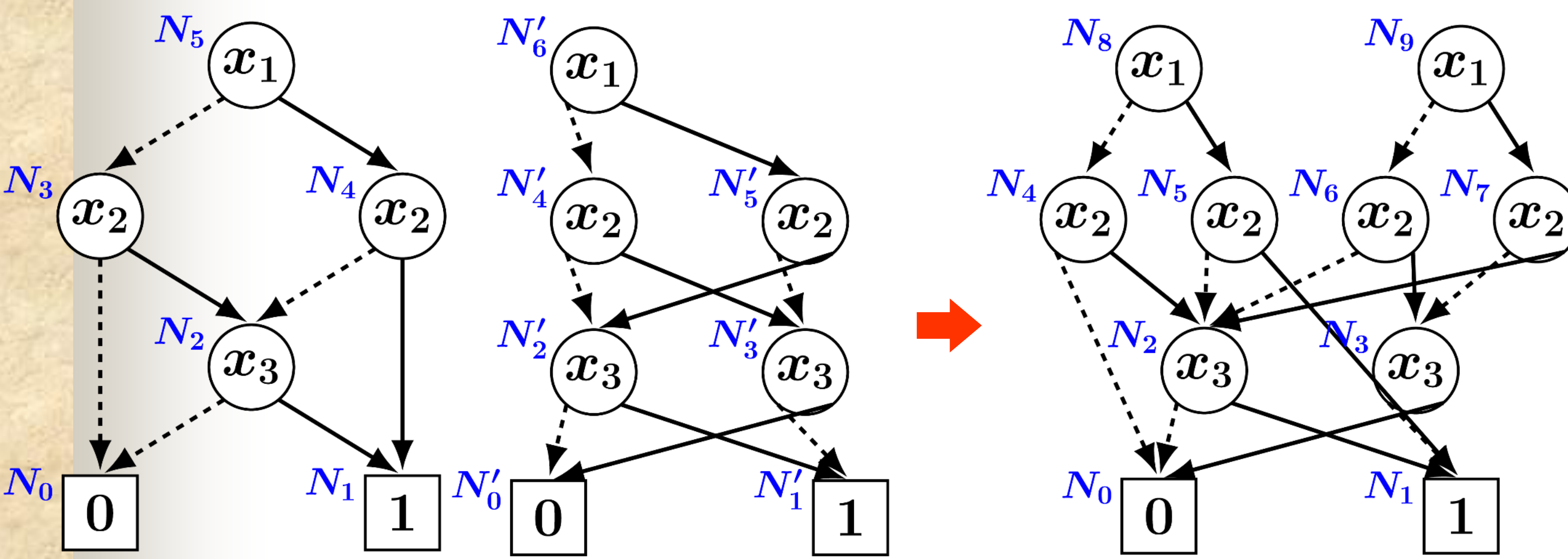
- Logical AND of the following two BDDs



Short break

- Take a deep breath, and relax yourself

Shared BDDs



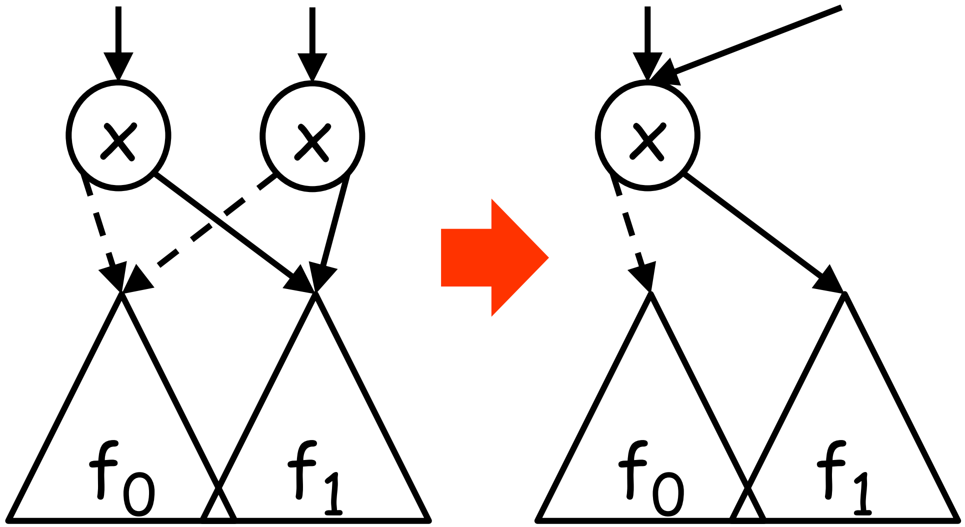
- By using the same variable order for representing BDDs, we can **share equivalent nodes** of two (or more) **BDDs**

→ **Uniqueness** of Boolean functions in a BDD management system (No two BDDs represent the same Boolean function)

Management system: Ensure uniqueness

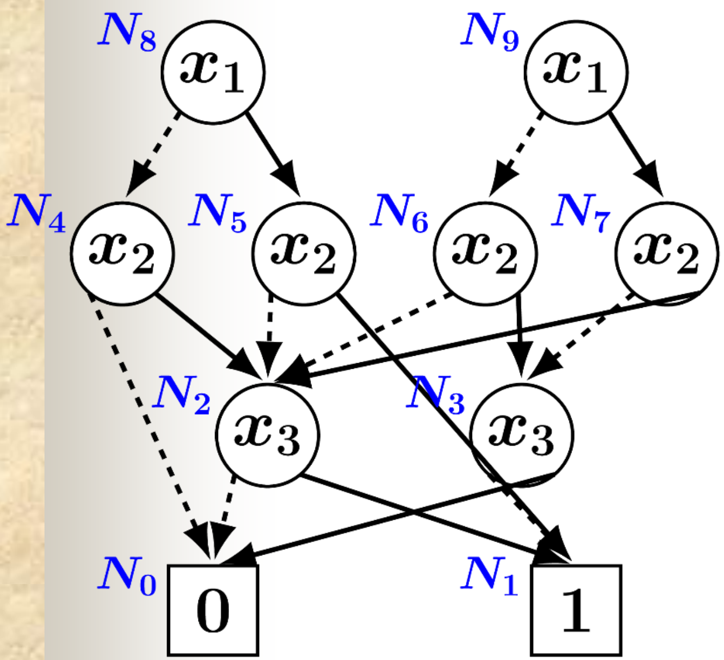
- **Equivalent nodes should be shared**
  (No equivalent nodes in a BDD management system)

Share **equivalent nodes**



- In a BDD management system, **node v** is **represented** as a **triple** of (variable name, the node pointed by the 0-edge of v, the node pointed by the 1-edge of v)

Management system: Ensure uniqueness



Node table

node ID	var.	0-edge	1-edge
N_0	-	-	-
N_1	-	-	-
N_2	x_3	N_0	N_1
N_3	x_3	N_1	N_0
N_4	x_2	N_0	N_2
N_5	x_2	N_2	N_1
N_6	x_2	N_2	N_3
N_7	x_2	N_3	N_2
N_8	x_1	N_4	N_5
N_9	x_1	N_6	N_7

- In a BDD management system, **node v** is **represented** as a **triple** of (variable name, the node pointed by the 0-edge of v, the node pointed by the 1-edge of v)

Management system: Ensure uniqueness

- Given a triple as a request for creating a node
  - Check: If the triple is already registered in the node table, return its node ID
  - Otherwise, create a new node
- Node table is implemented by a hash table (triples are used as hash keys)
  - Above check is done in O(1) time

Hash table is the key for managing BDDs efficiently

Node table

node ID	var.	0-edge	1-edge
N_0	-	-	-
N_1	-	-	-
N_2	x_3	N_0	N_1
N_3	x_3	N_1	N_0
N_4	x_2	N_0	N_2
N_5	x_2	N_2	N_1
N_6	x_2	N_2	N_3
N_7	x_2	N_3	N_2
N_8	x_1	N_4	N_5
N_9	x_1	N_6	N_7

- In a BDD management system, **node v** is **represented** as a **triple** of (variable name, the node pointed by the 0-edge of v, the node pointed by the 1-edge of v)

Management system: Ensure uniqueness

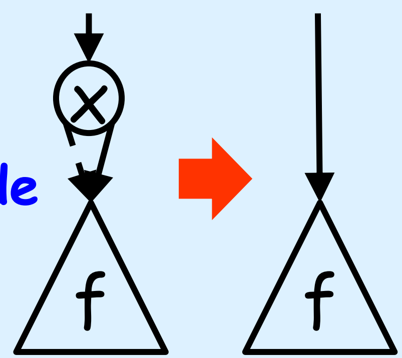
- Given a triple as a request for creating a node
  - Check: If the triple is already registered in the node table, return its node ID
  - Otherwise, create a new node

Node table

node ID	var.	0-edge	1-edge
N_0	-	-	-
N_1	-	-	-
N_2	x_3	N_0	N_1
N_3	x_3	N_1	N_0
N_4	x_2	N_0	N_2
[illegible]	[illegible]	N_2	N_1
[illegible]	[illegible]	N_2	N_3
[illegible]	[illegible]	N_3	N_2
[illegible]	[illegible]	N_4	N_5
[illegible]	[illegible]	N_6	N_7

In case the 0-edge and the 1-edge point to the same node (i.e., same Boolean function), return its node ID

Delete **a redundant node**



Due to the uniqueness of Boolean functions, the isomorphism of the subgraphs can be checked simply by comparing their node IDs

epresented

nted by the

dge of v)

Node request: GetNode(x, N_{f0}, N_{f1})

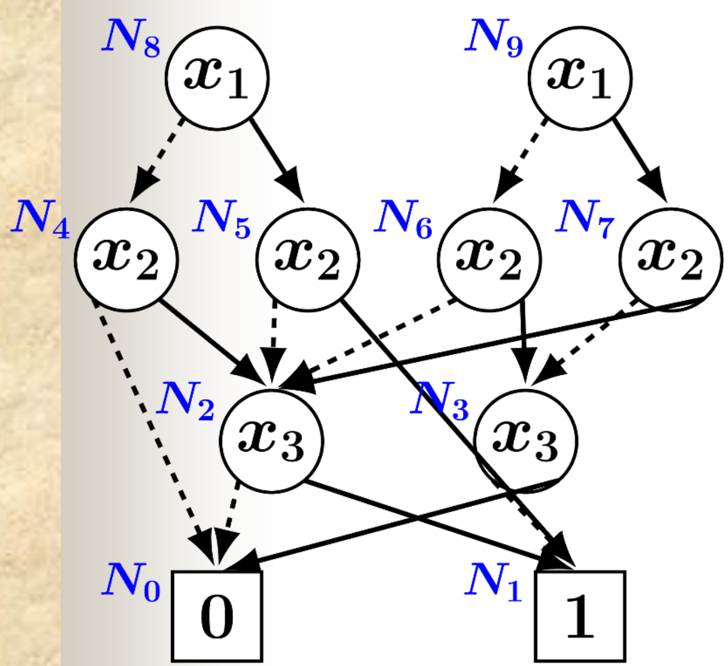
1. If $N_{f0} = N_{f1}$, return N_{f0}
2. If triple (x, N_{f0}, N_{f1}) is already registered in the node table, return its node ID
3. Otherwise, register (x, N_{f0}, N_{f1}) in the node table, and return its node ID

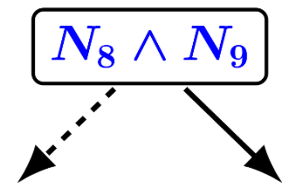
Short break

- Take a deep breath, and relax yourself

Apply logical operations

- Logical AND of the following two BDDs



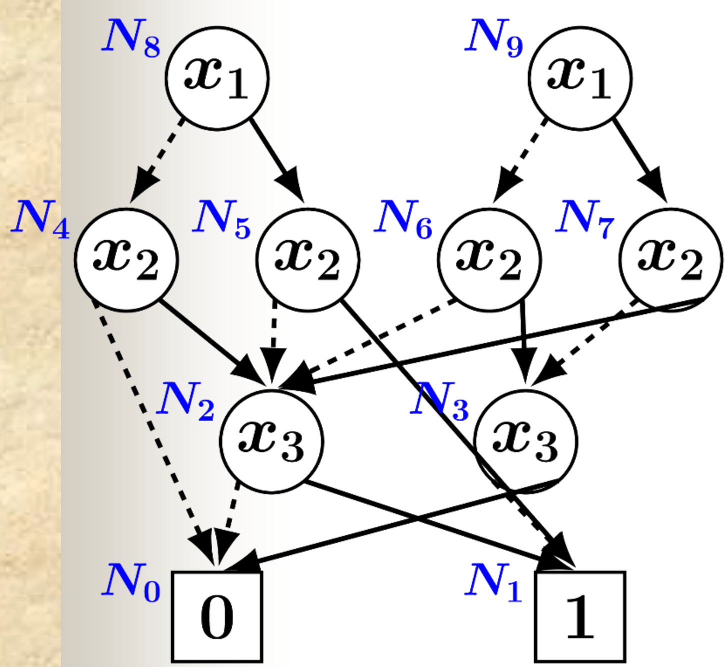


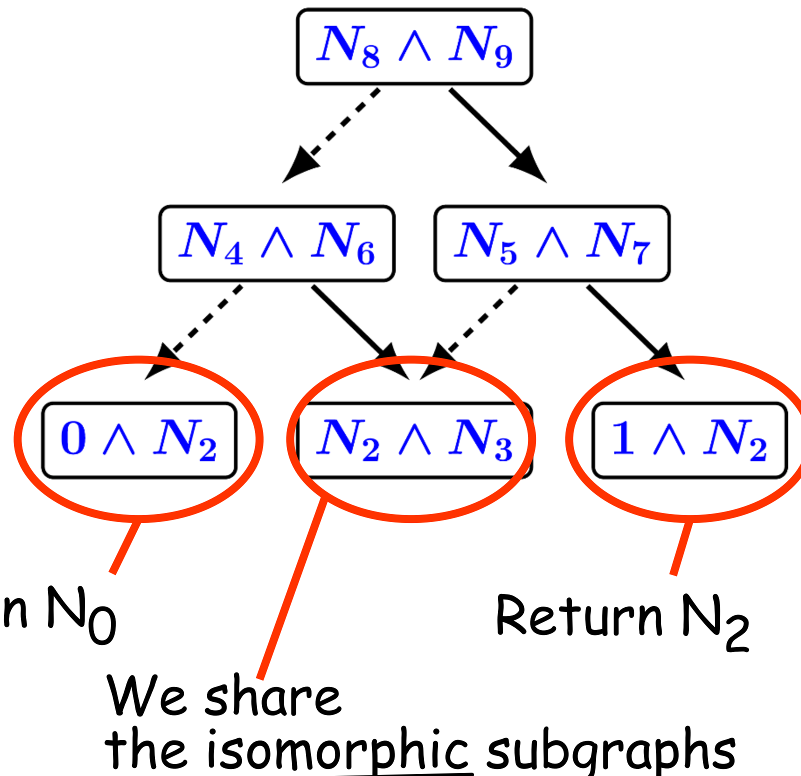
- At first, we create node N_i pointed by the 0-edge and node N_j pointed by the 1-edge
- Then, GetNode(x_1, N_i, N_j)

Recursion to the children pointed by the 0-edges and 1-edges

practice: Apply logical operations

- Logical AND of the following two BDDs

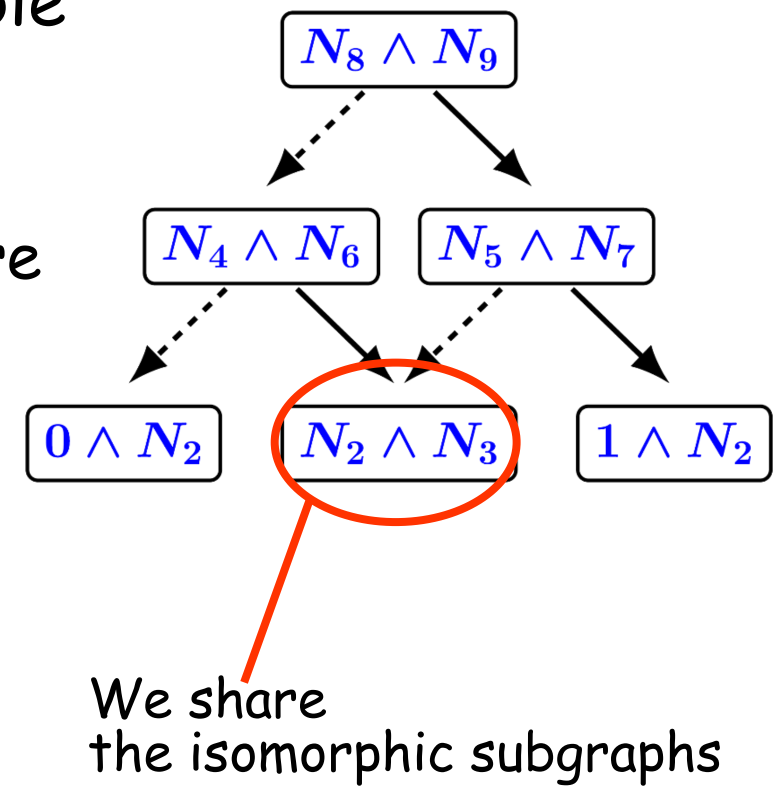




GetNode( ) for $N_2 \wedge N_3$ is called **after creating the subgraphs** pointed by the 0-edge and the 1-edge of $N_2 \wedge N_3$
→ The **isomorphic subgraphs** are **created twice**, **then** they are **shared** ...
(This approach is time consuming....)

Management system: Do not apply the same operation twice (or more)

- Register the results of logical operations in the operation result table (hash table)
  - Hash key: triple (op, N_f, N_g), where op is operation ID (representing AND, OR, ...), N_f is node ID of node f, N_g is node ID of node g
  - Operation result table returns the node ID of the root node of the resulting BDD

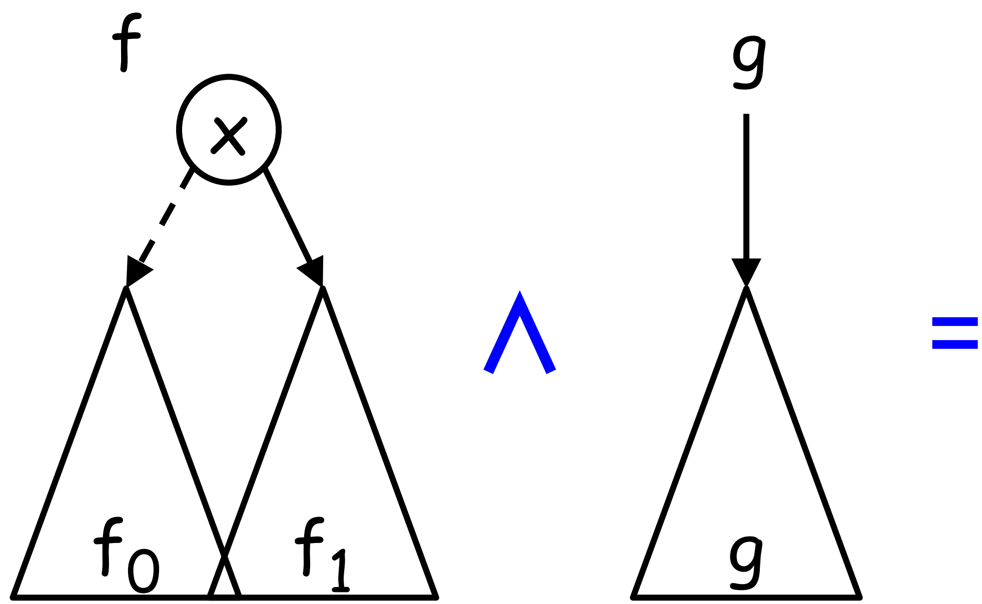


p. 4 is the case when the root nodes of f and g have the same variable

In case the root nodes of f and ghave different vars.

By definition, $f \wedge g = \overline{x}(f_0 \wedge g_0) \vee x(f_1 \wedge g_1)$

In case g does not depend on x ?



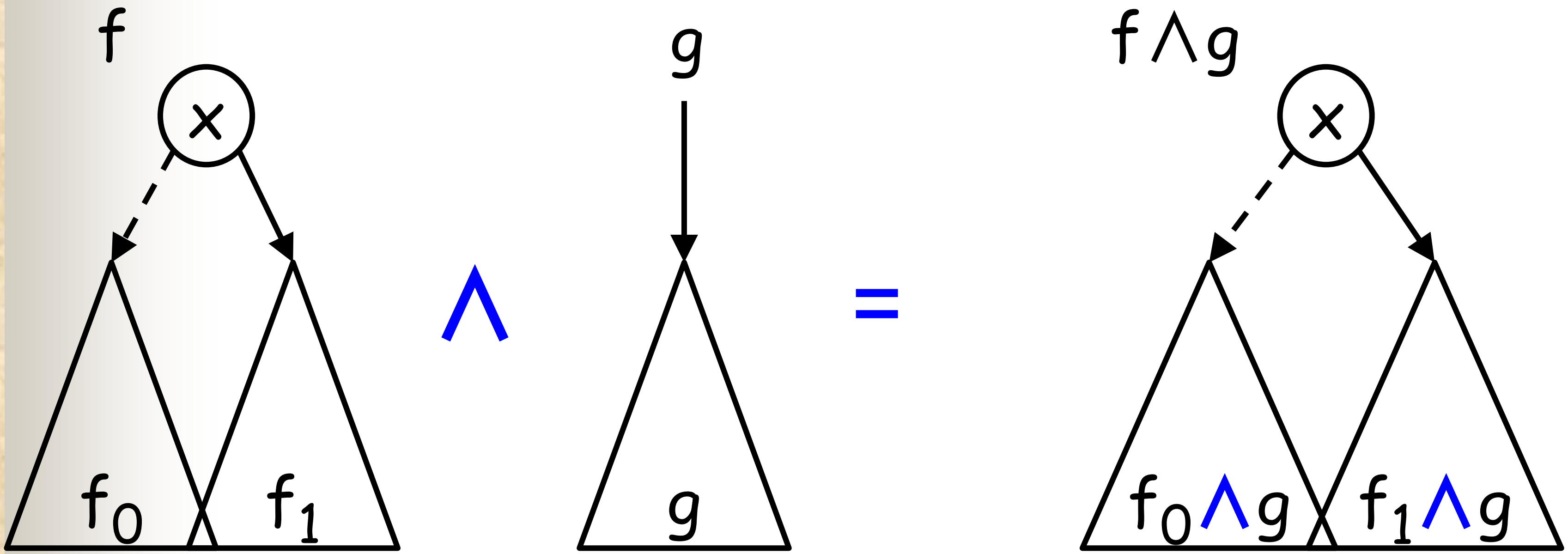
p. 4 is the case when the root nodes of f and g have the same variable

In case the root nodes of f and ghave different vars.

By definition, $f \wedge g = \overline{x}(f_0 \wedge g_0) \vee x(f_1 \wedge g_1)$

In case g does not depend on x ?

$$f \wedge g = \overline{x}(f_0 \wedge g) \vee x(f_1 \wedge g)$$



Apply operation $Apply(op, N_f, N_g)$

$N_f: (x_f, N_{f0}, N_{f1})$

$N_g: (x_g, N_{g0}, N_{g1})$

1. If at least one of N_f and N_g is a constant node, or if $N_f = N_g$ holds, return the node ID of the resulting BDD (according to op) (e.g., $0 \wedge N_f = 0$, $1 \wedge N_f = N_f$, $N_f \wedge N_f = N_f$)
2. If (op, N_f, N_g) is registered in the operation result table, return the node ID of the result
3. If variables x_f and x_g are the same
   - 3-1. $N_{h0} := Apply(op, N_{f0}, N_{g0})$, $N_{h1} := Apply(op, N_{f1}, N_{g1})$
   - 3-2. If $N_{h0} = N_{h1}$ holds, return N_{h0}

     Otherwise, return the resulting node ID of $GetNode(x_f, N_{h0}, N_{h1})$
4. If variable x_f appears in higher level than x_g
   - 4-1. $N_{h0} := Apply(op, N_{f0}, N_g)$, $N_{h1} := Apply(op, N_{f1}, N_g)$
   - 4-2. Same as 3-2
5. If variable x_f appears in lower level than x_g
   - Same as 4 (exchange the roles of N_f and N_g)

Time complexity of Apply operation

- Worst-case time complexity: O(|f| |g|)
  - This is because the size of the resulting BDD of the operation can be O(|f| |g|)
- For a long time, the time complexity is believed to be less than O(|f| |g|) in case the size of the resulting BDD is small
- Unfortunately, however, it is proved that "even if the sizes of the input and result BDDs are small, there exists an instance that requires O(|f| |g|) time" [Yoshinaka et al. 2012]
- Empirically, in many cases, we can apply operations within the time proportional to |f| + |g|

By utilizing hash tables

Extra: Reference counter

- Reference count of node v:
  - The number of reference from other nodes (i.e., in-degree of v; how many times node v is pointed from other nodes)
- In many BDD management systems, reference counter is used in the node table
- How to use reference counter?
  - Repetition of GetNode( ) (i.e., creating nodes) floods the node table
  - Garbage collection: Recycle nodes of reference count 0
- Things to consider
  - Recycled nodes still exist in the operation result table → We need to clear the operation result table (whole table)
  - Suppose that we recycle a node in each time when the reference count becomes 0 (which means clearing the op. table) → The efficiency of the operation result table is spoiled
  - Garbage collection is done when the node table is almost full

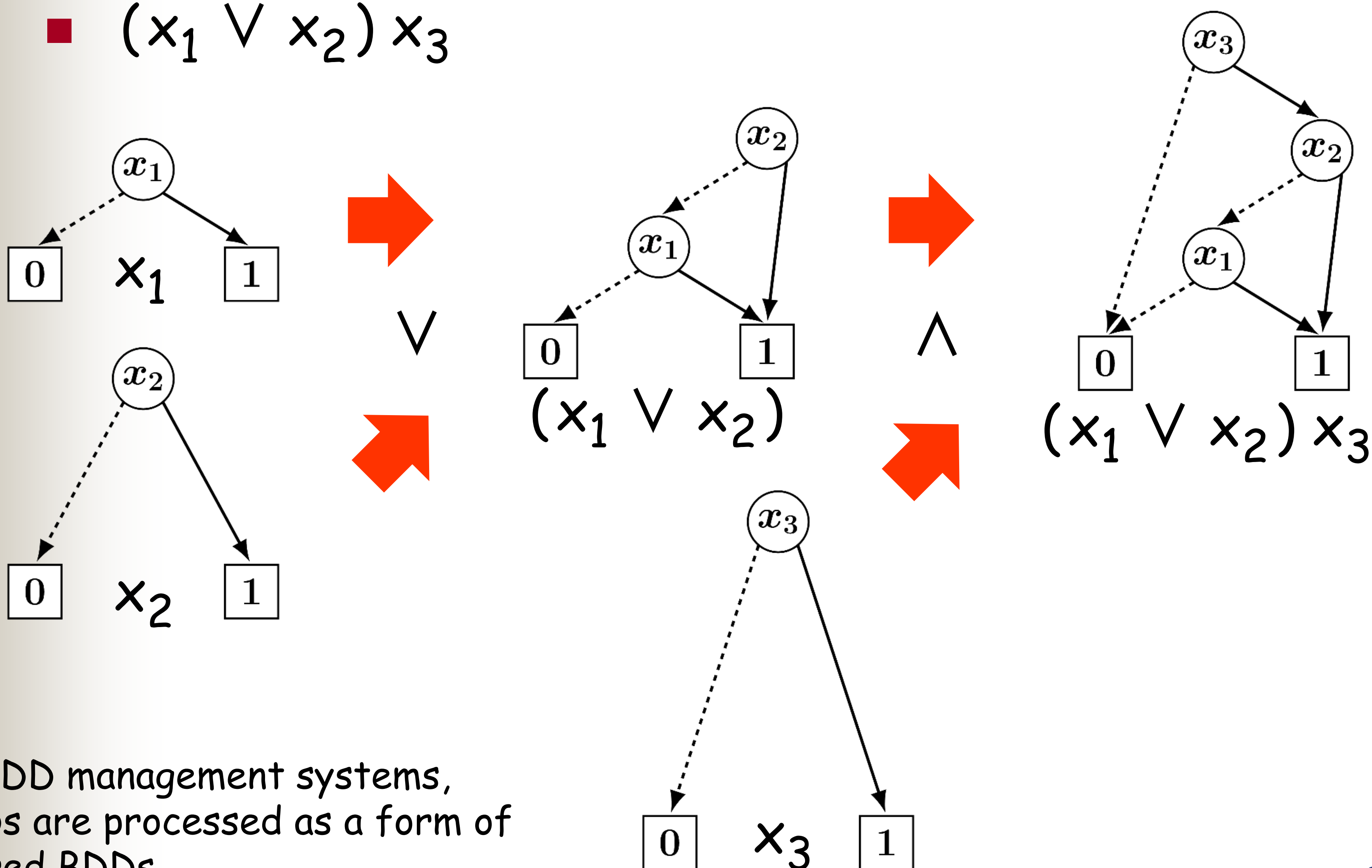
practice： Create BDDs

- Represent the following Boolean functions by BDDs
  1. AND： $x_1 x_2 x_3 x_4$
  2. OR： $x_1 \vee x_2 \vee x_3 \vee x_4$
  3. Combination of AND and OR： $(x_1 \vee x_2) x_3$
  4. Exclusive-OR (XOR)： $x_1 \oplus x_2 \oplus x_3 \oplus x_4$

- Three ways for creating BDDs
  - Create a truth table → decision tree → BDD
  - Create a BDD from top by considering the subfunctions (p. 6)
  - Create a BDD by Apply operations (see the following page)

Create a BDD by Apply operations

- $(x_1 \vee x_2)\, x_3$



In BDD management systems, BDDs are processed as a form of shared BDDs

Summary

- Binary Decision Diagram (BDD)
- Apply operations on two BDDs
  - Recursion

Techniques for efficient manipulation

- Shared BDDs: Uniqueness of Boolean functions
- Two hash tables for efficient operations
  - Node table:
    - Ensure the uniqueness
      i.e., do not create equivalent nodes twice (or more)
  - Operation result table:
    - Do not apply the same operation twice (or more)