

Entscheidungsverfahren für die Software-Verifikation

4 - BDDs

Datenstruktur BDD

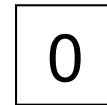
- ▶ 1986 von R. Bryant vorgeschlagen
- ▶ zur Darstellung von aussagenlogischen Formeln (genauer: Booleschen Funktionen)
 - ▶ Boolesche Fkt.: $f: \mathbf{B}^n \rightarrow \mathbf{B}$ (für $n \in \mathbf{N}$)
- ▶ f wird repräsentiert als gerichteter azyklischer Graph (DAG)
 - ▶ „Baum mit Sharing“
 - ▶ **Blätter**: Boolesche Konstanten 0/1;
innere Knoten: markiert mit Variablen, genau zwei Nachfolger (über 0-/1-Kante)

BDD: Semantik

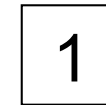
- ▶ Terminalknoten stellen konstante Funktion 0 bzw. 1 dar
- ▶ Innere Knoten interpretiert als:

$$\begin{aligned} f &= \text{if } x \text{ then } f_1 \text{ else } f_0 \\ &= (x ? f_1 : f_0) \\ &= (x \Rightarrow f_1) \wedge (\neg x \Rightarrow f_0) \end{aligned}$$

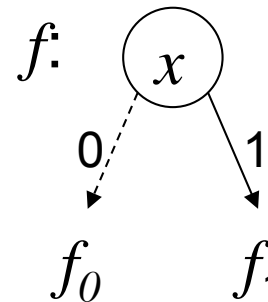
[f_0 und f_1 wiederum BDD-Knoten]



Konstante 0



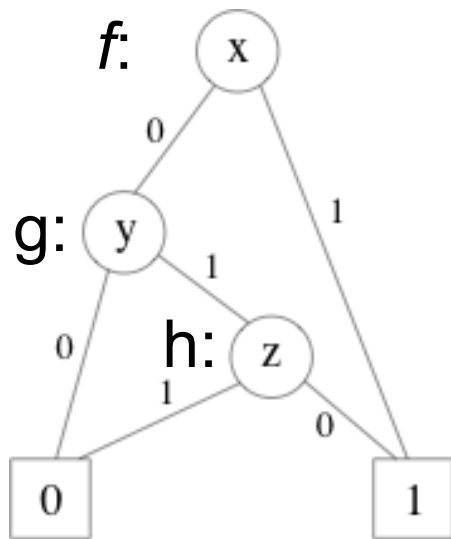
Konstante 1



Innerer Knoten mit

- Entscheidungsvariable x
- f_0 an 0-Kante und
- f_1 an 1-Kante

BDD: Beispiel



BDD-Darstellung
von $x \vee (y \wedge \neg z)$

$$\begin{aligned} f &= \text{if } x \text{ then } 1 \text{ else } g \\ &= \text{if } x \text{ then } 1 \text{ else (if } y \text{ then } h \text{ else } 0) \\ &= \text{if } x \text{ then } 1 \text{ else} \\ &\quad (\text{if } y \text{ then (if } z \text{ then } 0 \text{ else } 1) \text{ else } 0) \\ &= \text{if } x \text{ then } 1 \text{ else (if } y \text{ then } \neg z \text{ else } 0) \\ &= \text{if } x \text{ then } 1 \text{ else } (y \Rightarrow \neg z) \wedge (\neg y \Rightarrow 0) \\ &= \text{if } x \text{ then } 1 \text{ else } (y \Rightarrow \neg z) \wedge y \\ &= \text{if } x \text{ then } 1 \text{ else } \neg z \wedge y \\ &= (x \Rightarrow 1) \wedge (\neg x \Rightarrow \neg z \wedge y) \\ &= x \vee (\neg z \wedge y) \end{aligned}$$

Reduced Ordered BDDs (ROBDDs)

1. Totale **Ordnung** auf Variablen: $>$

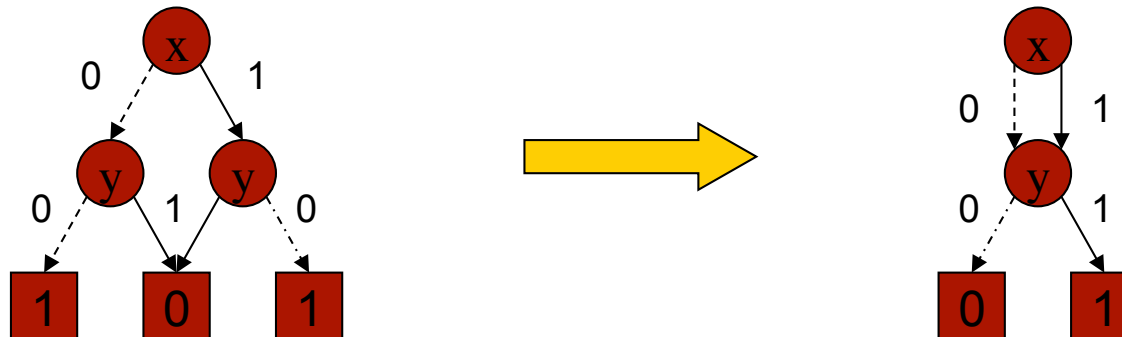
- ▶ **Bedingung:** Variable des Elternknotens muss kleiner sein als Variable beider Kindknoten
- ▶ Auf jedem Pfad kommt jede Variable höchstens einmal vor

2. **Reduktionen:**

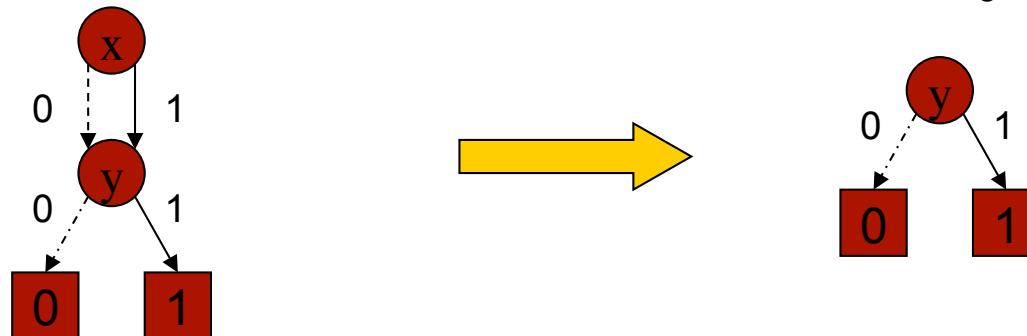
- a) Isomorphe Teilbäume dürfen nicht mehrfach vorkommen, mehrere Vorkommen werden auf eines **reduziert**.
- b) Knoten f mit $f_0 = f_1$ werden gelöscht und Kanten, die auf f zeigen auf f_0 (bzw. f_1) umgebogen.

ROBDDs: Beispiel Reduktionen

Reduktion a) [gleiche Teilbäume nur 1x]



Reduktion b) [gleiche Nachfolger, d.h. $f_0=f_1$]

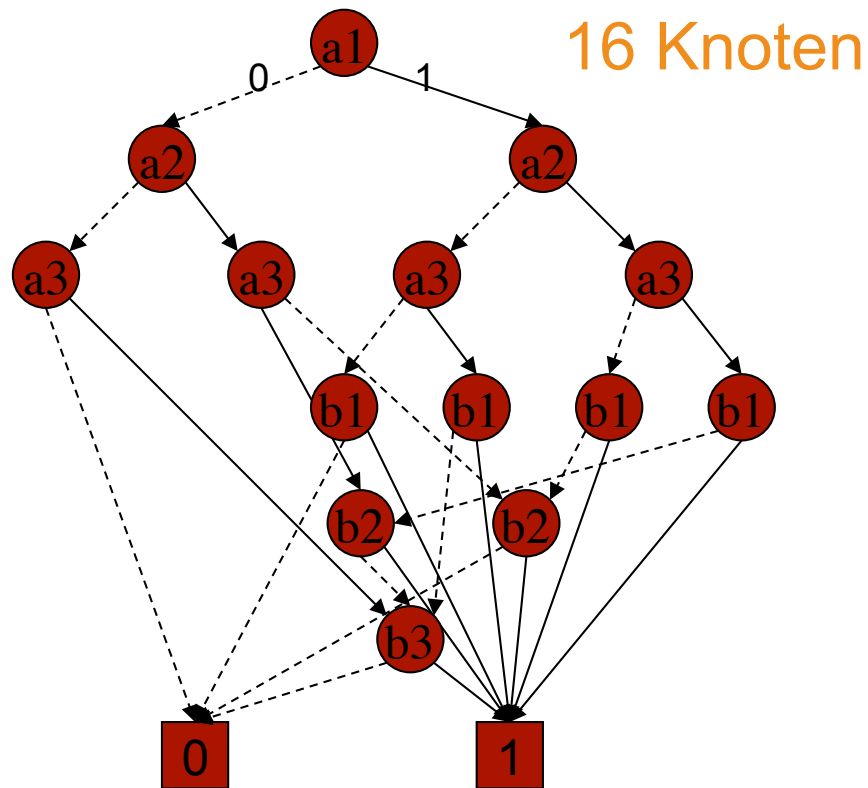


ROBDDs: Wichtige Eigenschaften

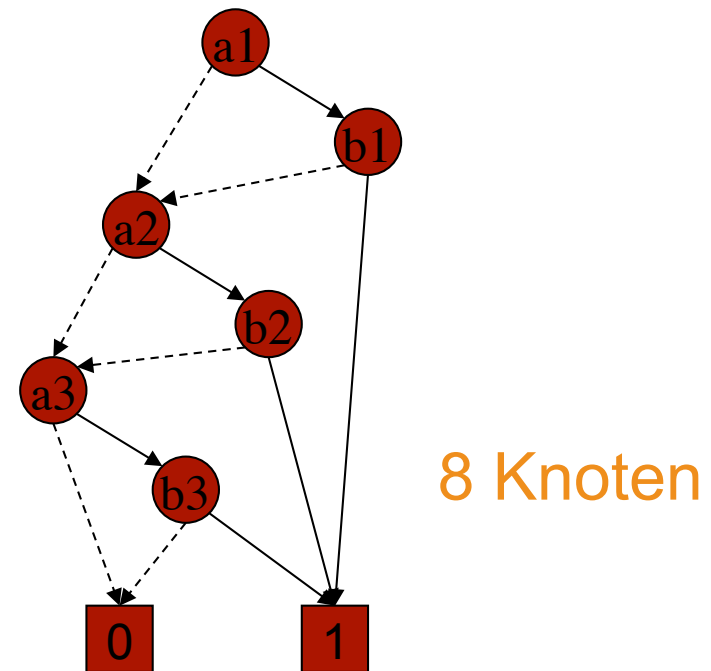
- ▶ ROBDD b zu gegebener Booleschen Funktion f **eindeutig**, d.h.:
 - ▶ b ist eindeutiger Repräsentant aller zu f äquivalenten Formeln.
 - ▶ **Spezialfall**: Unerfüllbare (allgemeingültige) Formeln werden durch 0-Knoten (1-Knoten) repräsentiert.
- ▶ Größe des ROBDDs kann stark von der gewählten Variablenordnung $>$ abhängen.

Abhängigkeit von Variablenordnung

Bsp. $f = a_1 * b_1 + a_2 * b_2 + a_3 * b_3$



$a_1 < a_2 < a_3 < b_1 < b_2 < b_3$



$a_1 < b_1 < a_2 < b_2 < a_3 < b_3$

Generierung von (RO)BDDs

1. „Top-Down“-Ansatz

- Zerlegung der Eingabeformel f unter gleichzeitiger Generierung des BDDs, d.h.
 - a) Selektion der kleinsten in f vorkommenden Variablen
 - b) Berechnung von f_0 und f_1
 - c) Rekursive BDD-Transformation von f_0 und f_1

2. „Bottom-Up“-Ansatz

- BDDs für atomare Formeln (Variablen/0/1)
- Operationen zur Generierung komplexer BDDs aus einfacheren (z.B. BDD-Disjunktion, BDD-Konjunktion)

Top-Down-Ansatz: Grundlagen

► Definition: Restriktion

Sei F eine aussagenlogische Formel, x eine Variable und $b \in \{0, 1\}$. Die Restriktion $F|_{x=b}$ ist dann rekursiv wie folgt definiert:

$$0|_{x=b} = 0$$

$$1|_{x=b} = 1$$

$$y|_{x=b} = \begin{cases} 1 & \text{falls } x = y \text{ und } b = 1 \\ 0 & \text{falls } x = y \text{ und } b = 0 \\ y & \text{sonst} \end{cases}$$

$$(\neg G)|_{x=b} = \neg(G|_{x=b})$$

$$(G \vee H)|_{x=b} = G|_{x=b} \vee H|_{x=b}$$

$$(G \wedge H)|_{x=b} = G|_{x=b} \wedge H|_{x=b}$$

Top-Down-Ansatz

► Shannon-Expansion:

$$\begin{aligned} f &= (x \wedge f|_{x=1}) \vee (\neg x \wedge f|_{x=0}) \\ &= (x \Rightarrow f|_{x=1}) \wedge (\neg x \Rightarrow f|_{x=0}) \end{aligned}$$

► f_0 und f_1 werden Kofaktoren von f (bezüglich x) genannt.

► Benannt nach Claude Shannon (1916-2001)

► Top-Down-Ansatz berechnet BDDs mittels rekursiver Shannon-Expansion

► d.h. $f_1 = f|_{x=1}$, $f_0 = f|_{x=0}$ für einen BDD-Knoten mit Variable x

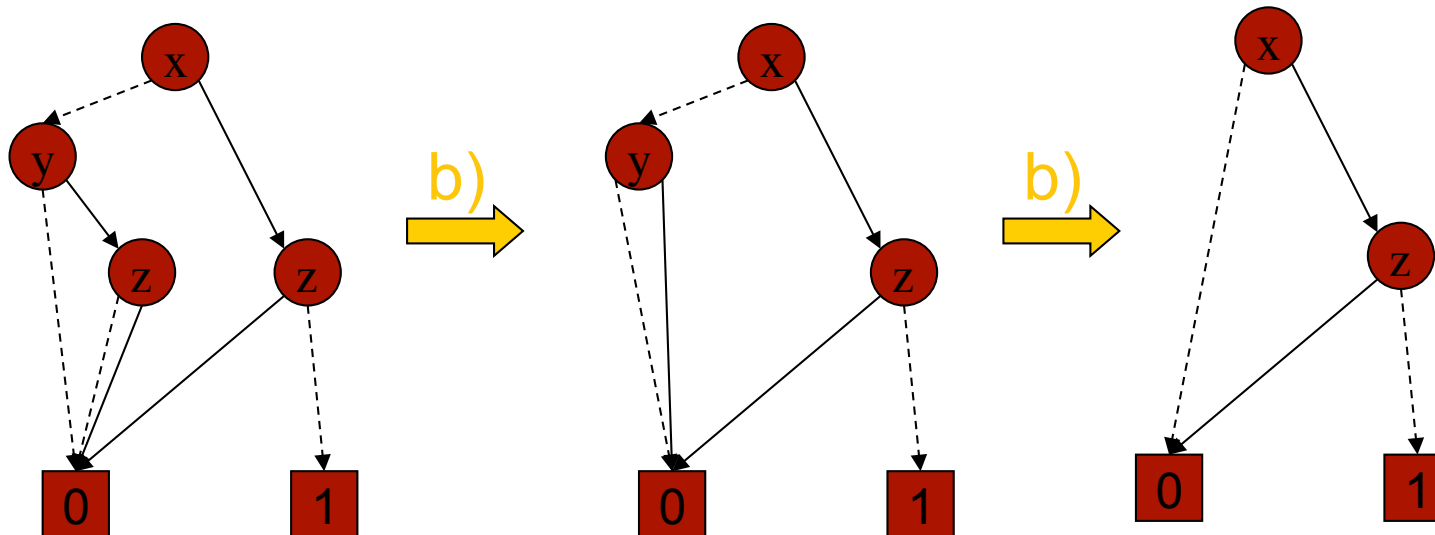
► Nachträgliche Reduktion erforderlich.

Top-Down-Ansatz: Nachträgliche Reduktion

Bsp: $f = (x \vee y) \wedge (x \vee \neg y \vee z) \wedge \neg z$, $z > y > x$

$$f|_{x=0} = (0 \vee y) \wedge (0 \vee \neg y \vee z) \wedge \neg z = y \wedge (\neg y \vee z) \wedge \neg z$$

$$f|_{x=1} = (1 \vee y) \wedge (1 \vee \neg y \vee z) \wedge \neg z = \neg z$$



Bottom-Up-Ansatz

- ▶ BDDs für atomare Formeln plus Konstruktoren für komplexe Formeln

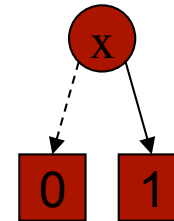
- ▶ Boolesche Konstanten:

- ▶ Variablen:



- ▶ Komplexe Formeln:

- ▶ Mittels BDD-Algorithmen
 - ▶ *apply*-Algorithmus zur Berechnung von $F \otimes G$ für Boolesche Verknüpfung $\otimes$.



BDD-Algorithmus: „BDD-and“

Algorithm 1: BDD-and(a, b)

- 1: if $a = 0$ or $b = 0$ then return 0
 - 2: if $a = 1$ then return b else if $b = 1$ then return a
 - 3: $(x, a_0, a_1) = \text{decompose}(a)$; $(y, b_0, b_1) = \text{decompose}(b)$
 - 4: if $x > y$ then return new-node(y , BDD-and(a, b_0), BDD-and(a, b_1))
 - 5: if $x = y$ then return new-node(x , BDD-and(a_0, b_0), BDD-and(a_1, b_1))
 - 6: if $x < y$ then return new-node(x , BDD-and(a_0, b), BDD-and(a_1, b))
-

- ▶ $(x, f_0, f_1) = \text{decompose}(f)$ zerlegt Knoten in Variable und Kofaktoren.
- ▶ **new-node**(x, f_0, f_1) legt neuen BDD-Knoten mit Variable x und Kofaktoren f_0 und f_1 an, sofern dieser noch nicht existiert. Ansonsten wird der bereits vorhandene Knoten verwendet.

Automatische Reduktion mittels *new-node*

- ▶ *new-node*(x, f_0, f_1) implementiert auch Reduktion:
 - ▶ Falls $f_0 = f_1$, so wird f_0 zurückgegeben.
 - ▶ Falls ein Knoten (x, f_0, f_1) bereits angelegt wurde, so wird dieser zurückgegeben (implementiert mittels Hash-Tabelle)

Abstraktion

- ▶ **Zwei Arten von Abstraktion**
 - ▶ $\forall xF$ (universelle Quantifikation)
 - ▶ $\exists xF$ (existentielle Quantifikation)
- ▶ **Definiert durch:**
 - ▶ $\forall xF \equiv F|_{x=0} \wedge F|_{x=1}$
 - ▶ $\exists xF \equiv F|_{x=0} \vee F|_{x=1}$
 - ▶ Berechnung mittels apply-Algorithmus
- ▶ **Verwendung im Model Checking zur Berechnung von Fixpunkten**

BDDs als Beweissystem für Formeln in CNF

- ▶ Gegeben: $F = C_1 \wedge \dots \wedge C_m$ unerfüllbar
- ▶ Gesucht: Beweis für die Unerfüllbarkeit von F
- ▶ Beweisregeln eines BDD-Kalküls:

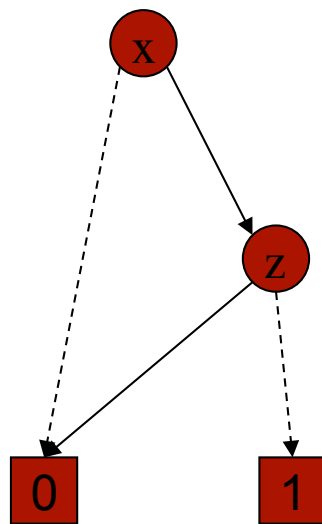
$$\frac{}{BDD(C)} \quad \text{Axiom-Regel (für alle Klauseln } C \in F)$$

$$\frac{B}{\exists x B} \quad \text{Projektions-Regel (für beliebige Variable } x)$$

$$\frac{B_1 \quad B_2}{B_1 \wedge B_2} \quad \text{Join-Regel (Konjunktion)}$$

Bottom-Up-Ansatz: Implementierung

- ▶ Hash-Tabelle zur Speicherung von nodes:



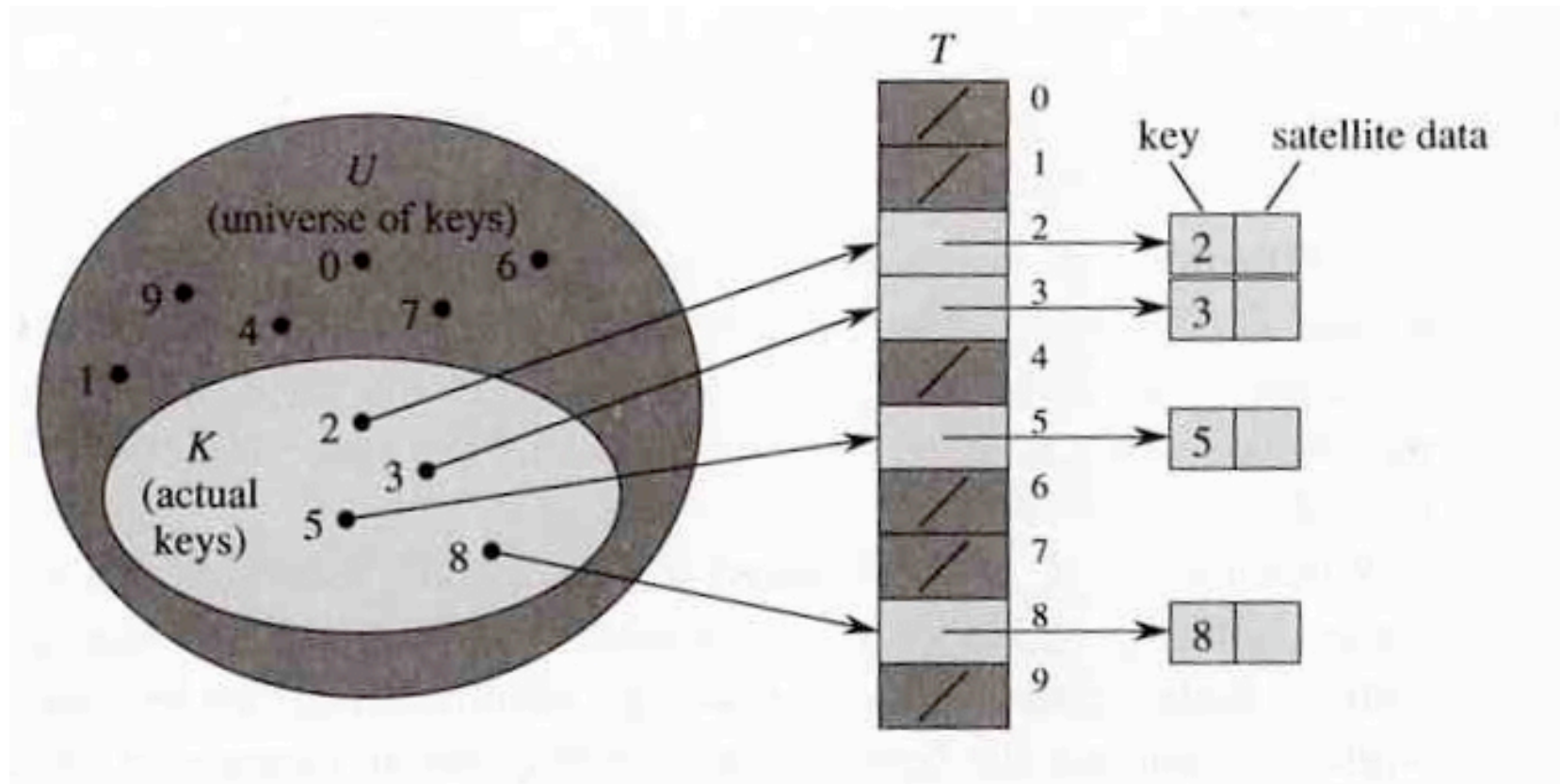
Idx	Var	Low	High
0	0	--	--
1	1	--	--
⋮			
6	z	1	0
⋮			
8	x	0	6
⋮			

- ▶ Weitere Hash-Tabelle zum Zwischenspeichern von Ergebnissen der BDD-Konstruktions-Algorithmen

Hashing (I)

- ▶ Verallgemeinerung der „array“-Datenstruktur
 - ▶ **Array**: direkte Adressierung der Elemente anhand von Index.
 - ▶ **Hash-Tabelle**: Berechnung des Indexes, an dem ein Element abgelegt wurde durch **Hash-Funktion**.
- ▶ Erlaubt Suchen, Einfügen und Löschen von Elementen in $O(1)$.

Daten-Speicherung in Array



[Quelle: Corman/Leiserson/Rivest: Introduction to Algorithms]

Daten-Speicherung in Hash-Tabelle

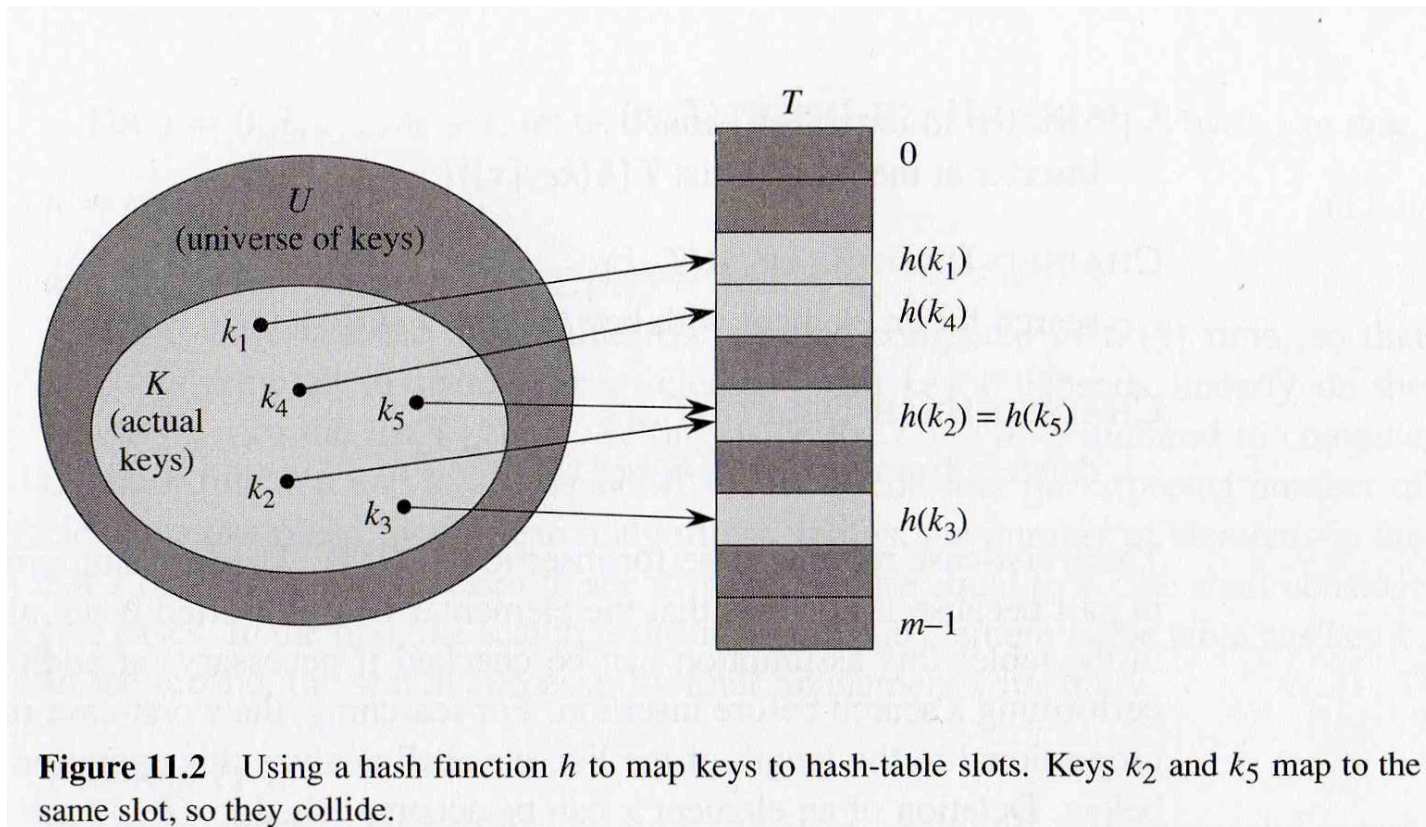
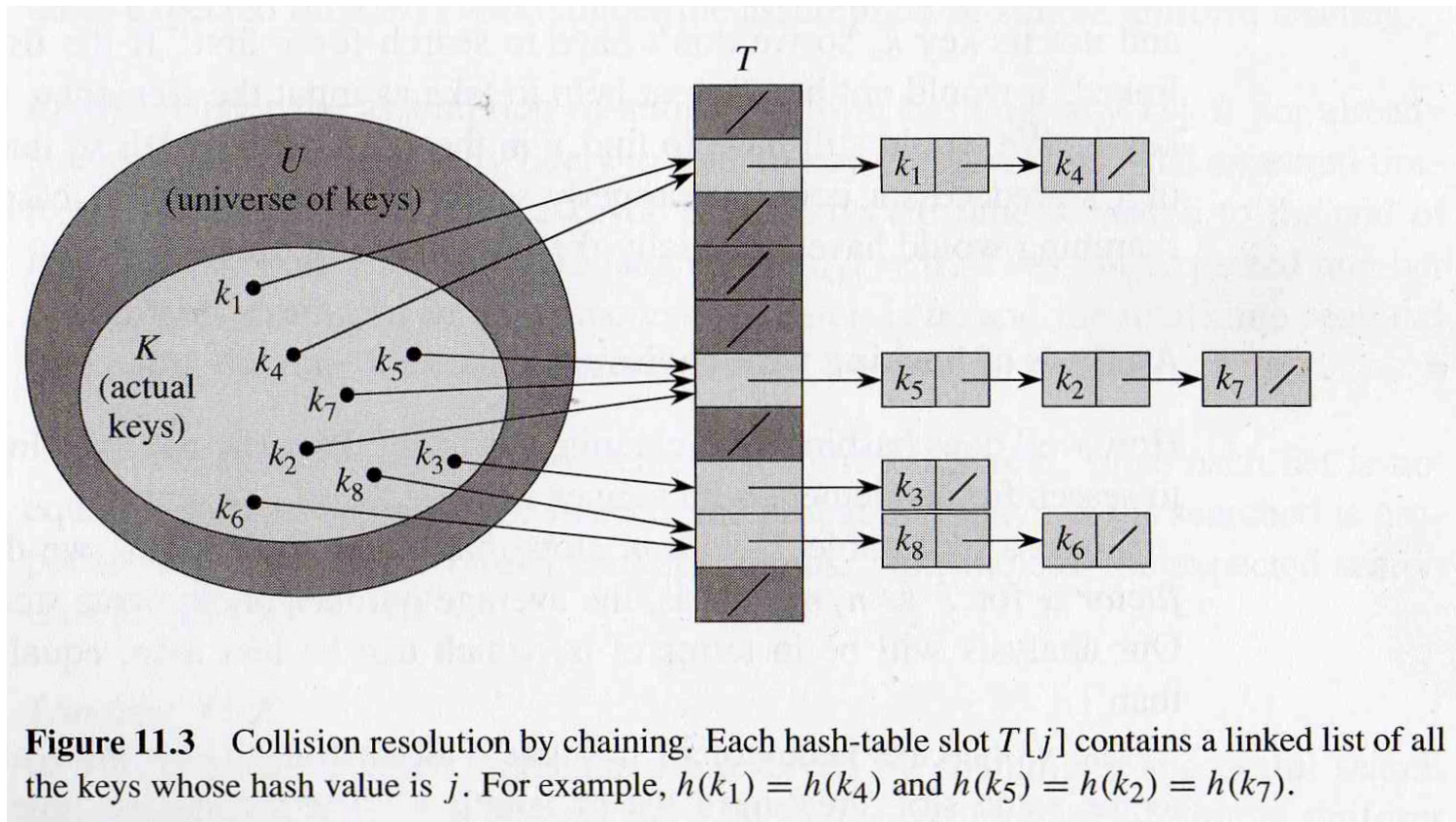


Figure 11.2 Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

[Quelle: Corman/Leiserson/Rivest: Introduction to Algorithms]

Hash-Tabelle

Kollisions-Auflösung mittels *Chaining*



[Quelle: Corman/Leiserson/Rivest: Introduction to Algorithms]

Hash-Funktionen

- ▶ Berechnen Indizes in Hash-Tabelle
- ▶ Gewünschte Eigenschaften:
 - ▶ möglichst gute Verteilung auf alle Indizes
- ▶ Problem:
 - ▶ zu speichernde Elemente im Voraus nicht bekannt
- ▶ Typische Hash-Funktionen:
 - ▶ $h(k) = k \bmod m$ (Divisions-Methode)
 - ▶ $h(k) = \lfloor m (kA \bmod 1) \rfloor$ mit $A \in [0,1]$
(Multiplikations-Methode)

Hash-Tabellen: Kollisions-Auflösung

1. Chaining

2. Open Addressing

□ **linear probing:** $h(k,i) = (h'(k) + i) \bmod m$

□ **quadratic probing:**
 $h(k,i) = (h'(k) + c i + d i^2) \bmod m$

□ **double hashing:**
 $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$

BDD-Varianten (I)

▶ FDDs: [Kebschull, Schubert, Rosenstiel; 1992]

- ▶ Functional Decision Diagrams

- ▶ (Positive) Davio-Expansion anstelle der Shannon-Expansion

$$f = f|_{x=0} \oplus x \wedge (f|_{x=0} \oplus f|_{x=1})$$

- ▶ Reduktion b) ersetzt durch Reduktion c): Falls $f_1=0$, so ersetze f durch f_0 .

▶ ZDDs: [Minato, 1992]

- ▶ Zero-Suppressed Decision Diagrams

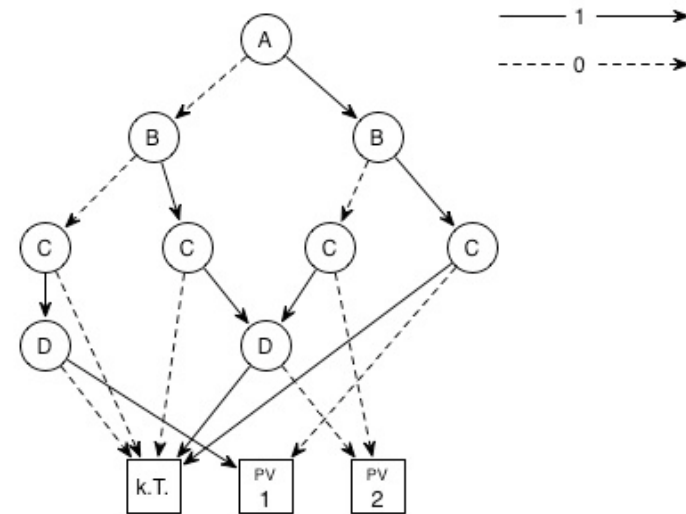
- ▶ Zur Darstellung von Teilmengen einer endlichen Menge M

- ▶ Unterschied BDD: fehlende Variablen auf Pfad werden als 0 interpretiert, nicht als don't care.

BDD-Varianten (II)

► MTBDDs:

- Multi-Terminal BDDs
- Einsatz z.B. in der Darstellung von Fahrzeugstücklisten



□ ADDs:

- Arithmetic Decision Diagrams
- Ganze Zahlen in Terminalknoten
- „Gewichtete Terme“

BDD-Pakete im Internet

- ▶ **CUDD** [<http://vlsi.colorado.edu/~fabio/CUDD>]
 - ▶ CU (University of Colorado) DD package
 - ▶ BDDs, ZDDs, ADDs
 - ▶ Dynamische Variablen-Umordnung
 - ▶ C / C++
- ▶ **Buddy** [<http://buddy.sourceforge.net/>]
 - ▶ Universität Kopenhagen
 - ▶ C++, Dynamische Variablen-Umordnung
 - ▶ Graphische Ausgabe von BDDs