Digitales Suchen

Nachteile des Hashings:

- der gesamte Schlüssel wird immer mit den eingetragenen Schlüsseln verglichen
- die Berechnung eines Indexes aus einem Schlüssel kann u.U. relativ aufwendig sein (siehe Hashing für Strings)

Idee:

- baue Binärbaum auf, der jedoch für jedes Bit des Schlüssels eine Links-/Rechts-Verzweigung vornimmt
- nach Abarbeitung jeden Bits eines Schlüssels hat man den gesuchten Schlüssel gefunden oder er ist nicht vorhanden

Digitales Suchen: Motivation

Vorteile des digitalen Suchens:

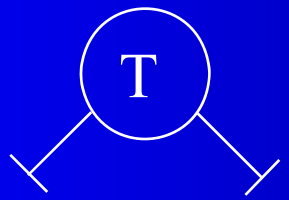
- nicht so kompliziert wie ausgeglichene Bäume (Rot-Schwarz-Bäume)
- trotzdem annehmbare Tiefen (damit Laufzeit) für ungünstige Anwendungen

Digitales Suchen: 1. Beispiel

Schlüssel sind Buchstaben:

- von jedem Buchstaben seine Binärcodierung betrachten
- hier: betrachte nur die Bits, in denen ein Unterschied besteht: Bit 0 bis 5
- Bit 6 und Bit 7 sind konstant 1 bzw. 0

T	84	01010100
e	101	01100101
s	115	01110011
t	116	01110100
i	105	01101001
n	110	01101110
g	103	01100111
		76543210

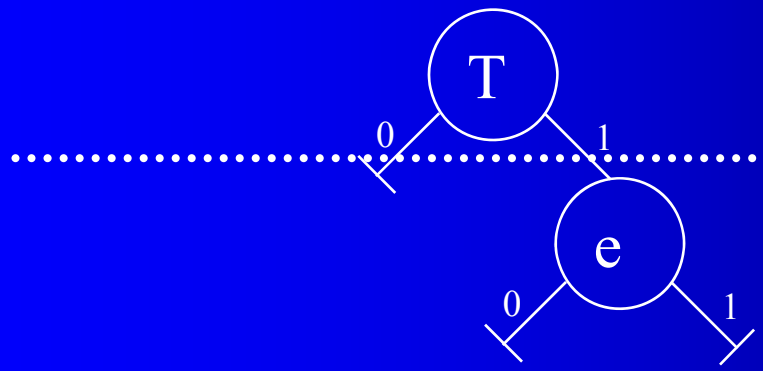


initiale Suchbaum, nur der Buchstabe 'T' ist eingetragen

Digitales Suchen: 1. Beispiel (Forts.)

- Buchstabe 'e' soll in den Baum eingetragen werden
- dazu werden solange die Bits 0 bis 5 entlanggegangen, bis ein Blatt erreicht ist
- bei 0 wird nach links gegangen
- bei 1 wird nach rechts gegangen

T	84	01010100
e	101	01100101
s	115	01110011
t	116	01110100
i	105	01101001
n	110	01101110
g	103	01100111
		76543210



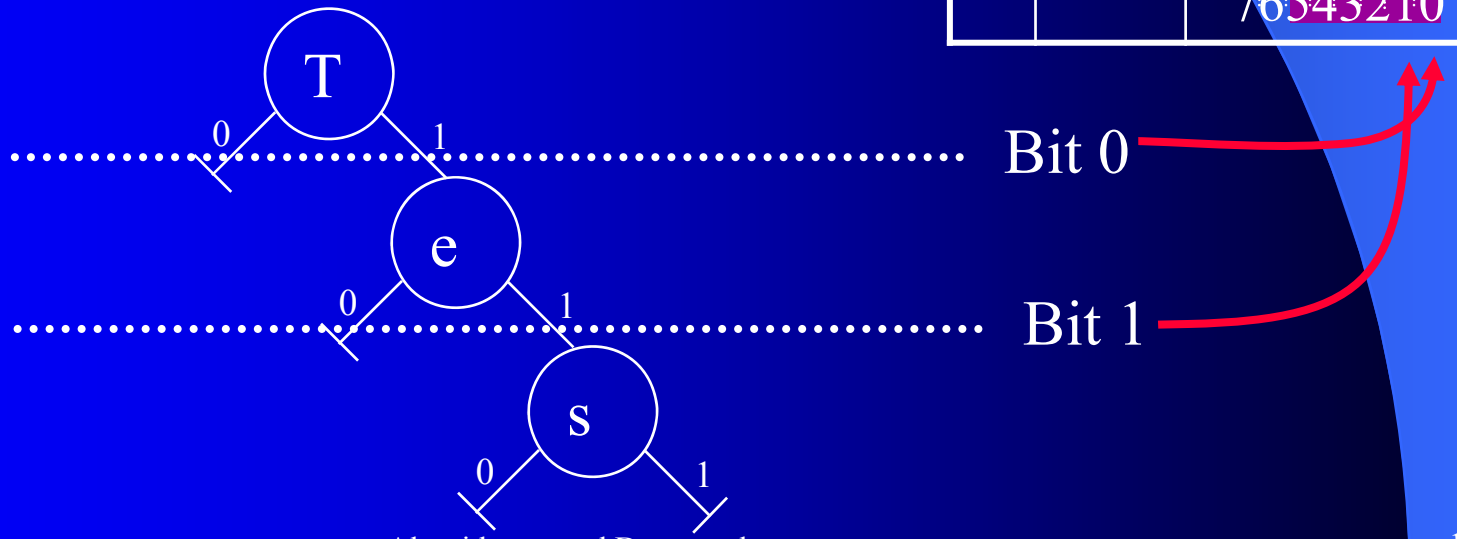
Bit 0

Suchbaum, mit 'T' und 'e'

Digitales Suchen: 1. Beispiel (Forts.)

- Buchstabe 's' soll in den Baum eingetragen werden
- dazu werden solange die Bits 0 bis 5 entlanggegangen, bis ein Blatt erreicht ist
- bei 0 wird nach links gegangen
- bei 1 wird nach rechts gegangen

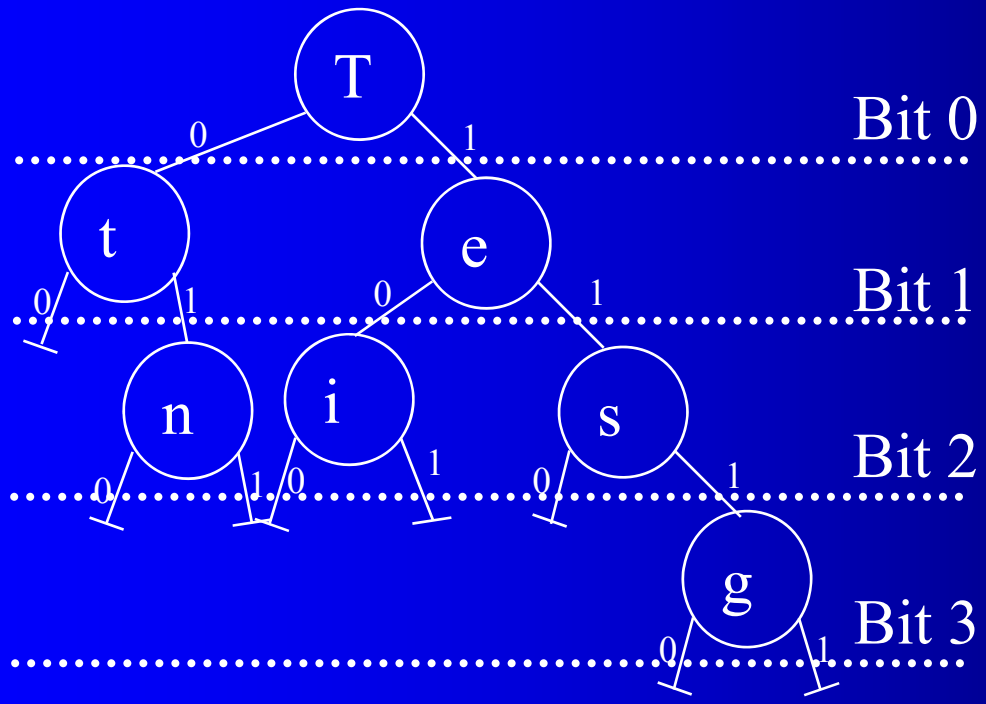
T	84	01010100
e	101	01100101
s	115	01110011
t	116	01110100
i	105	01101001
n	110	01101110
g	103	01100111
		76543210



Suchbaum, mit 'T', 'e' und 's'

Digitales Suchen: 1. Beispiel (Forts.)

- nachdem alle Buchstaben eingefügt sind, sieht der digitale Suchbaum wie folgt aus:

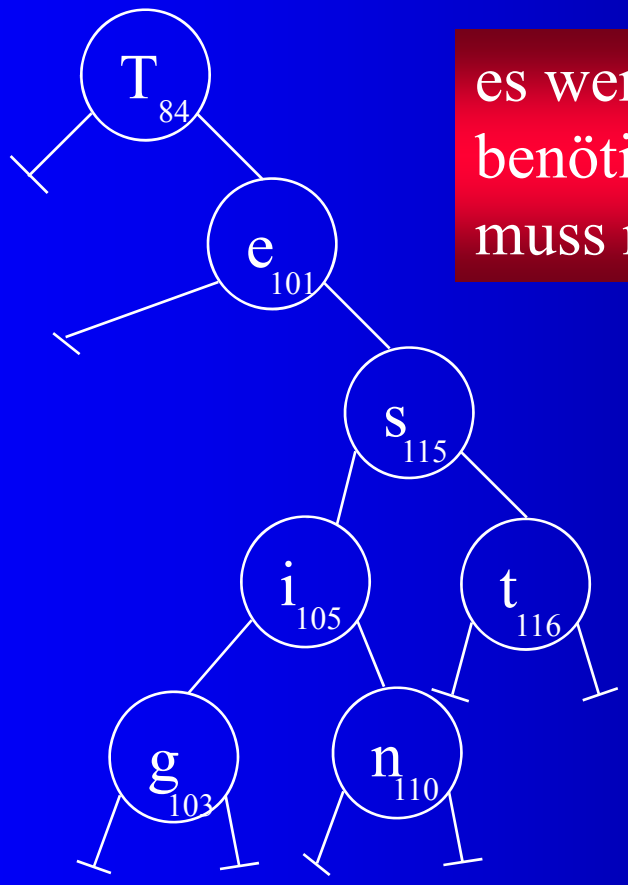


T	84	01010100
e	101	01100101
s	115	01110011
t	116	01110100
i	105	01101001
n	110	01101110
g	103	01100111
		76543210

es werden nur 4 von den maximalen 6 Bits angeschaut

Digitales Suchen: 1. Beispiel (Forts.)

- der zugehörige Binärbaum hätte folgende Form:



es werden 5 Ebenen benötigt, aber dass muss nicht sein

T	84
e	101
s	115
t	116
i	105
n	110
g	103

ohne ‚g‘ wären hier auch 5 Ebenen notwendig, beim digitalen Suchbaum nur 3

Digitales Suchen: Implementierung

```
class DigiTree {
    class Node {
        public Node(char key) {
            m_Key = key;
        }
        public char m_Key;
        public Node m_Left = null;
        public Node m_Right = null;
    }

    public boolean search(char c) {...}

    public void insert(char c) {...}

    private Node m_Root = null;
}
```

normalerweise sollte in einem Knoten neben dem Schlüssel auch das assoziierte Datum gespeichert werden

wie gehabt in BinTree oder RedBlackTree

Digitales Suchen: Implementierung (Forts.)

```
class DigiTree {
    ...
    public boolean search(char c) {
        Node tmp = m_Root;
        for(int i = 0; tmp != null; ++i) {
            if (tmp.m_Key == c)
                return true;
            tmp = (c & (1 << i)) != 0 ? tmp.m_Right : tmp.m_Left;
        }
        return false;
    }
    ...
}
```

solange noch Knoten vorhanden sind ...

... teste Bit 0, Bit 1, Bit2 usw. durch

Ist das i-te Bit im Schlüssel gesetzt ...

... dann nimm den rechten Nachfolger ...

... sonst den Linken !

Digitales Suchen: Implementierung (Forts.)

```
class DigiTree {
    ...
    public void insert(char c) {
        NodeHandler h = new NodeHandler(m_Root);
        int i = 0;
        for(i = 0;!h.isNull();++i)
            h.down((c & (1 << i)) == 0);
        h.set(new Node(c),(c & (1 << (i-1))) == 0);
    }
    ...
}
```

solange noch Knoten vorhanden sind ...

... teste Bit 0, Bit 1, Bit2 usw. durch

Abstieg nach Links und Rechts

Der NodeHandler muss wissen, ob der neue Knoten links oder rechts unter den Vater eingefügt werden soll

Digitales Suchen: Diskussion

- Vorteile gegenüber dem Hashing: nachdem ***maximal*** alle Bits ***angeschaut*** worden sind, kann entschieden werden, ob der gesuchte Schlüssel vorhanden ist
- Für ***jede Bitposition*** ist ein ***Vergleich*** mit dem ***aktuellen Schlüssel*** und dem ***gesuchten Schlüssel*** notwendig
- dies kann ein erheblicher Aufwand bei langen Schlüsseln sein (Beispiel: Strings mit ca. 20 Zeichen, 6 Bits pro Zeichen: maximal 120 Stringvergleiche)
- bei langen Schlüsseln (Schlüssel mit vielen Bits) dominiert der Schlüsselvergleich den Baumdurchlauf

Digitale Such-Tries

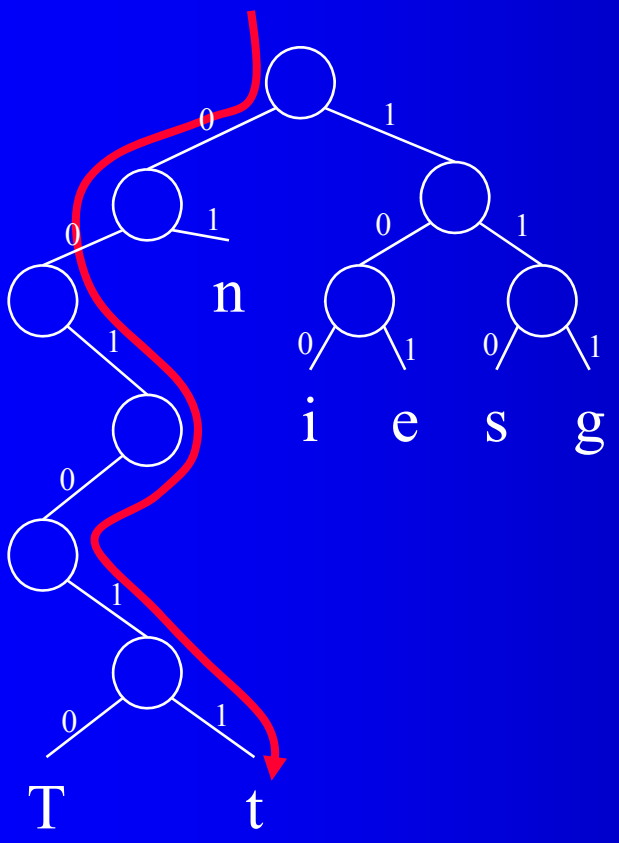
Ähnlich wie digitale Suchbäume, jedoch

- werden in den ***Knoten keine Schlüssel*** gespeichert
- ***nur*** in den ***Blättern*** werden die ***Schlüssel gespeichert***

Suchen und Einfügen erfolgen durch

- Abstieg analog zu den digitalen Suchbäumen, jedoch
- ***kein Schlüsselvergleich*** in den ***Knoten,*** sondern
- ***Schlüsselvergleich*** am Blatt, dadurch
- in dem Fall nur ***exakt ein Schlüsselvergleich***

Digitale Such-Tries: Beispiel



T	84	01010100
e	101	01100101
s	115	01110011
t	116	01110100
i	105	01101001
n	110	01101110
g	103	01100111
		76543210

Suchen von t

Digitale Such-Tries: Diskussion

Vorteil:

- nur am Ende muss maximal ein Schlüssel verglichen werden
- der Aufbau des Baums ist ***unabhängig*** von der Reihenfolge, in der die Schlüssel eingetragen werden

Nachteil:

- sehr viele innere Knoten, die nur zur Verzweigung dienen
- es gibt zwei unterschiedliche Knotentypen: aufwendig zu implementieren

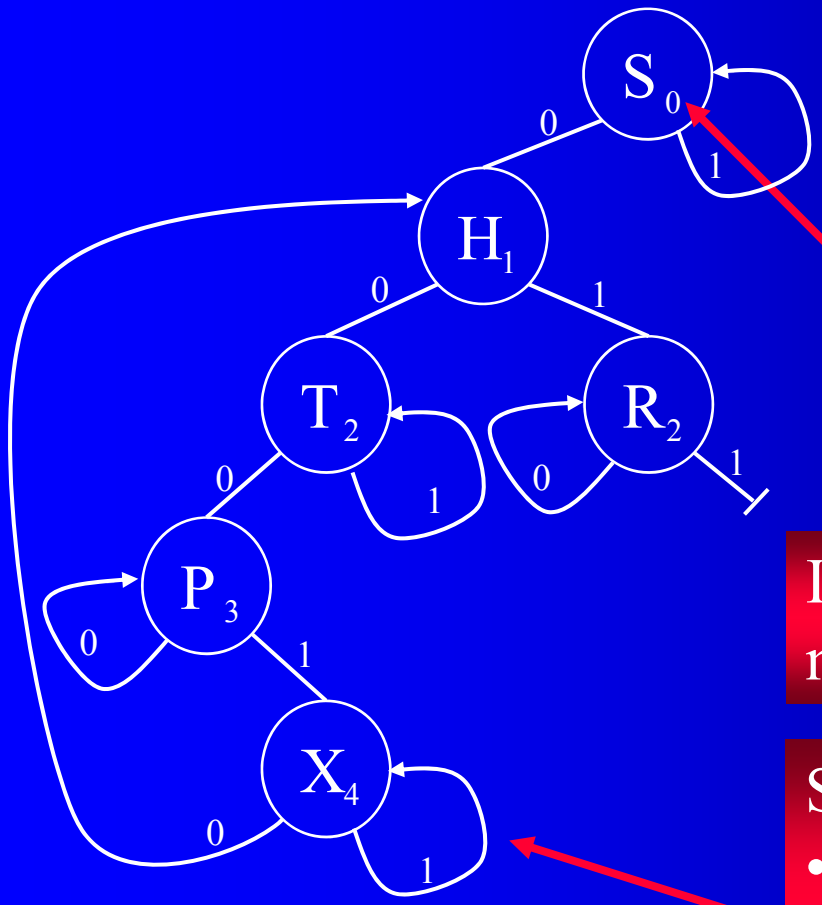
Patricia-Trees

Patricia: ***P***ractical ***A***lgorithm ***T***o ***R***etrieve ***I***nformation ***C***oded ***I***n ***A***lphanumeric

Idee:

- verwende normalen Digitalbaum, bei dem auch in den inneren Knoten Schlüssel abgespeichert sind
- vergleiche die Schlüssel dennoch erst am Ende
- um an den Blättern auf Schlüssel weiter oben im Baum zu verweisen zu können, führe Rückwärtskanten ein

Patricia-Trees: Beispiel



S	83	01010011
H	71	01001000
X	88	01011000
P	80	01010000
R	82	01010010
T	84	01010100
		76543210

Index gibt die Bitposition an, nach der entschieden wird

Suche:
- steige solange ab, bis wieder aufgestiegen wird
- vergleiche dann den Schlüssel

Patricia-Trees: Implementierung

```
class PatriciaTree {
    static boolean left(char key,int bitPos) {
        return (key & (1 << bitPos)) == 0;
    }
    class Node {
        public Node(char key,int bitPos,Node succ) {
            m_Key = key;
            m_BitPos = bitPos;
            boolean bIsLeft = left(key,bitPos);
            m_Left = bIsleft ? this : succ;
            m_Right = bIsLeft ? succ : this;
        }
        public Node(char key,int bitPos) {this(key,bitPos,null);}
        public char m_Key;
        public int m_BitPos;
        public Node m_Left;
        public Node m_Right;
    }
    ...
    private Node m_Root;
}
```

setzt Rück-verkettung

Standardkonstruktor ohne Nachfolger

Knoten merkt sich zusätzlich die Bitposition

Patricia-Trees: Implementierung (Fort.)

- das Suchen erfolgt im wesentlichen im NodeHandler

```
public boolean search(char c) {
    NodeHandler h = new NodeHandler(m_Root);
    h.search(c);
    return !h.isNull() && h.node(h.NODE).m_Key == c;
}
```

NodeHandler steigt ab, bis Rückwärts- oder Nullverweis gefunden wurde

ist der gefundene Knoten der gesuchte Knoten?

Patricia-Trees: Der NodeHandler

```
class NodeHandler {
    public final int NODE = 0;
    public final int DAD = 1;

    private Object[] m_Nodes = new Object[3];

    NodeHandler(Node n) {
        m_Nodes[NODE] = n;
    }

    void down(boolean left) {
        for(int i = m_Nodes.length-1;i >0;--i)
            m_Nodes[i] = m_Nodes[i-1];
        m_Nodes[NODE] = left ? node(DAD).m_Left : node(DAD).m_Right;
    }

    boolean isNull() {
        return m_Nodes[NODE] == null;
    }

    Node node(int kind) {
        return (Node)m_Nodes[kind];
    }
```

Analog zu RotSchwarz Bäumen: Knoten, Vater und Großvater (siehe später beim Löschen) müssen gemerkt werden

Abstieg

Zugriff auf die Knoten mittels der Konstanten NODE und DAD

Patricia-Trees: Der NodeHandler (Fort.)

```
void set(Node n,int kind) {
    if (node(kind+1) == null)
        m_Root = n;
    else if (  node(kind) != null ?
               node(kind+1).m_Left == node(kind) :
               left(n.m_Key,node(kind+1).m_BitPos))
        node(kind+1).m_Left = n;
    else
        node(kind+1).m_Right = n;
    m_Nodes[kind] = n;
}
void search(char c,int maxPos) {
    int lastBitPos = -1;
    while (   !isNull() &&
              lastBitPos < node(NODE).m_BitPos &&
              maxPos > node(NODE).m_BitPos) {
        lastBitPos = node(NODE).m_BitPos;
        down(left(c,lastBitPos));
    }
}
void search(char c) {
    search(c,Integer.MAX_VALUE);
}
```

Analog zu RotSchwarz Bäumen: setzen der Wurzel, wenn es keinen Vater gibt ...

... oder linke bzw. rechts unterhalb des Vaters

Abstieg bis zur maximalen Position (siehe Einfügen) maxPos

Abstieg bis zum Ende

Patricia-Trees: Einfügen

Idee:

- analog zu binären Bäumen: Absteigen und am Ende einfügen
- steige in dem Patricia Tree analog zu der Search Methode ab
- füge den neuen Knoten am Ende ein

3 Fälle sind zu unterscheiden:

- einzufügender Schlüssel existiert schon: fertig
- Suche endet in einem null-Verweis: neuen Knoten erzeugen
- Suche Ende in einem Knoten mit einem Verweis nach oben in den Baum

Fall 1

Fall 2

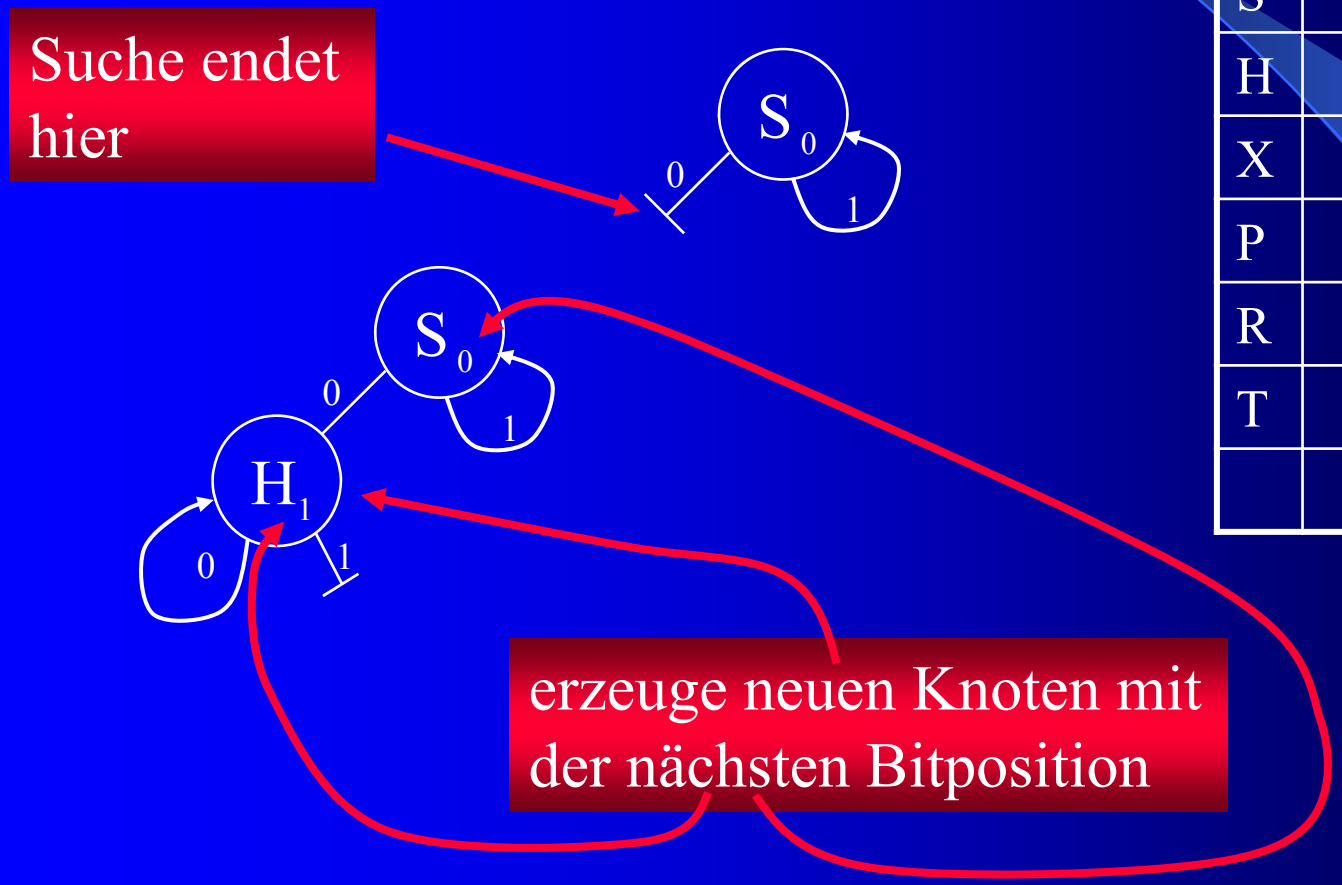
Fall 3

Fall 2

Patricia-Trees: Einfügen (Fort.)

- Suche endet in einem null-Verweis: neuen Knoten erzeugen
- H soll eingefügt werden

S	83	01010011
H	71	01001000
X	88	01011000
P	80	01010000
R	82	01010010
T	84	01010100
		76543210



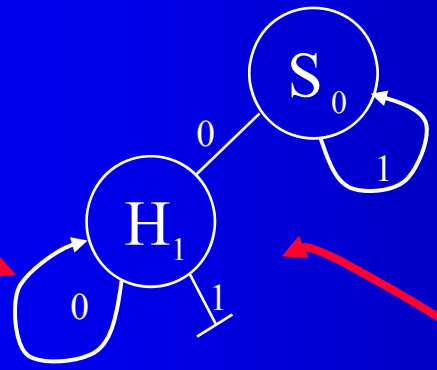
Fall 3 (leicht)

Patricia-Trees: Einfügen (Fort.)

- Suche endet in einem Knoten mit einem Verweis nach oben
- X soll eingefügt werden

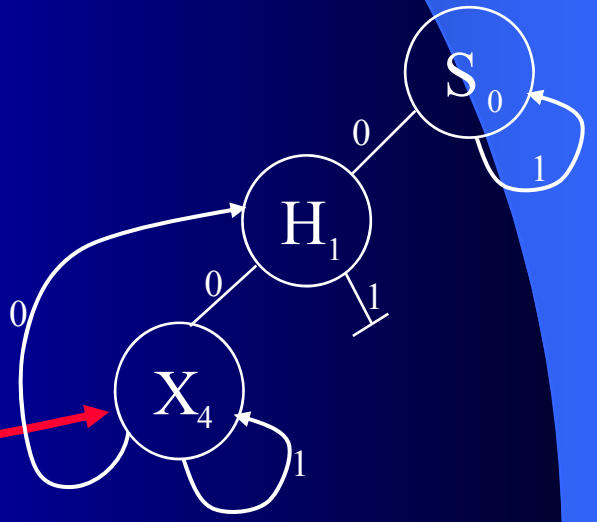
S	83	01010011
H	71	01001000
X	88	01011000
P	80	01010000
R	82	01010010
T	84	01010100
		76543210

Suche endet hier



Suche kleinste Bitposition, in der sich H und X unterscheidet: 4

hänge neuen Knoten unterhalb von H mit Bitposition 4 auf

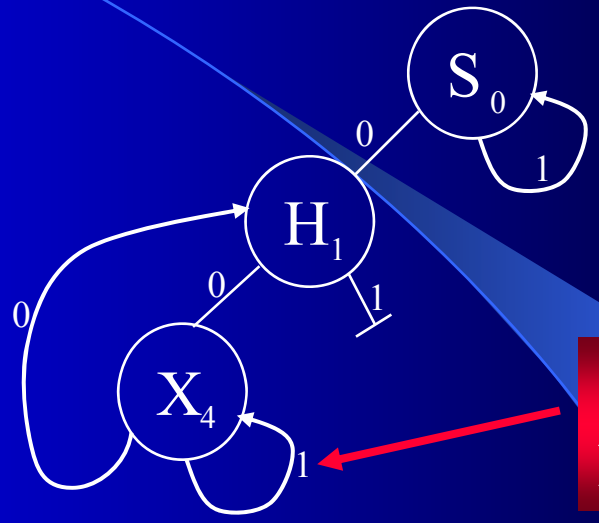


Patricia-Trees: Einfügen (Fort.)

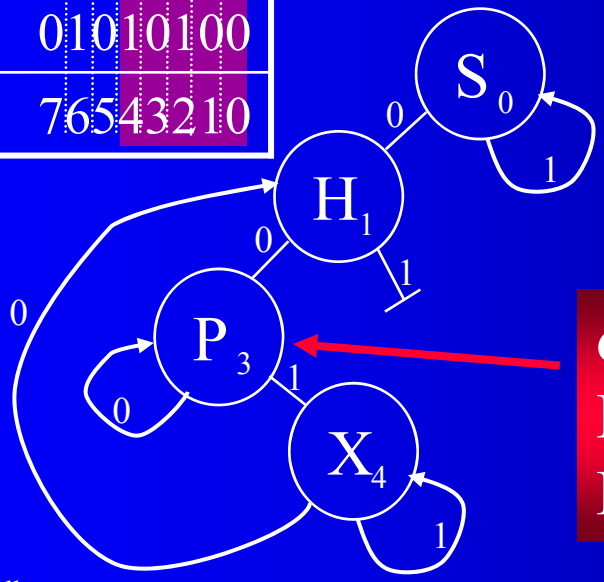
Fall 3 (schwierig)

- Suche endet in einem Knoten mit einem Verweis nach oben
- P soll eingefügt werden

S	83	01010011
H	71	01001000
X	88	01011000
P	80	01010000
R	82	01010010
T	84	01010100
		76543210



Suche endet hier

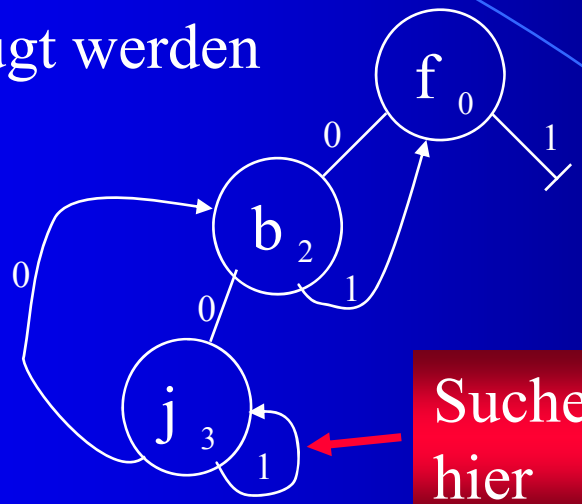


Suche kleinste Bitposition, in der sich X und P unterscheidet: 3

da 3<4 (von X), hänge neuen Knoten oberhalb von X mit Bitposition 3 auf

Patricia-Trees: Einfügen (Fort.)

- Suche endet in einem Knoten mit einem Verweis nach oben
- h soll eingefügt werden

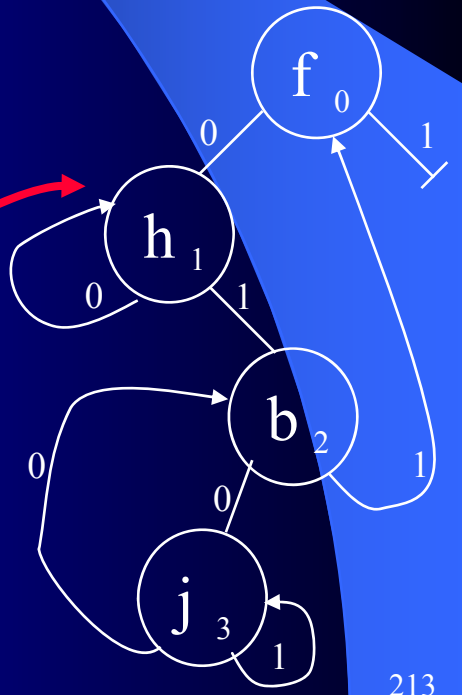


f	102	01100110
h	104	01101000
b	98	01100010
j	106	01101010

Suche endet hier

Suche kleinste Bitposition, in der sich j und h unterscheidet: 1

daher muss h zwischen f und b eingefügt werden. Dazu muss nochmals von oben der Baum durchlaufen werden



Patricia-Trees: Implementierung (Fort.)

```
public boolean insert(char c) {
    NodeHandler h = new NodeHandler(m_Root);
    h.search(c);
    int index = 0;
    if (h.isNull()) {
        if (h.node(h.DAD) != null) {
            index = h.node(h.DAD).m_BitPos + 1;
        }
    } else if (h.node(h.NODE).m_Key != c) {
        while (left(c,index) == left(h.node(h.NODE).m_Key,index))
            ++index;
    } else {
        // already inserted
        return false;
    }
    h = new NodeHandler(m_Root);
    h.search(c,index);
    h.set(new Node(c,index,h.node(h.NODE)),h.NODE);
    return true;
}
```

1. Abstieg: Suche nach Schlüssel

Fall 2

Fall 3

Kleinste unterschiedliche Bitposition

Fall 1

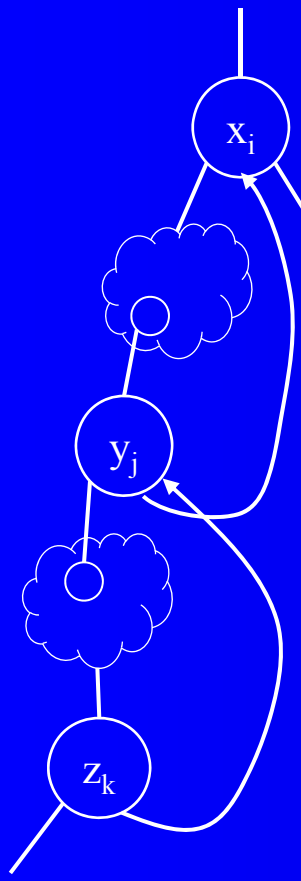
2. Abstieg: Suche nach Einfügeposition ...

... und einfügen

Patricia-Trees: Löschen

- das Löschen erfolgt analog zu Binärbaumen
- das zu löschende Element wird durch das Element ersetzt, das auf das zu löschende Element zeigt

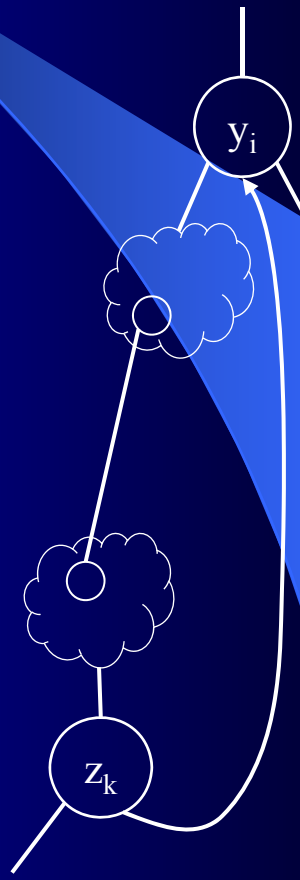
x soll gelöscht werden



y wandert nach oben,

$\Rightarrow$

Index j wird nicht mehr getestet



Patricia-Trees: Implementierung (Fort.)

```
boolean remove(char c) {
    NodeHandler h = new NodeHandler(m_Root);
    h.search(c);
    if (h.isNull() || h.node(h.NODE).m_Key != c) {
        return false;
    } else {
        NodeHandler h2 = new NodeHandler(h.node(h.DAD));
        h2.search(h.node(h.DAD).m_Key);
        h.node(h.NODE).m_Key = h.node(h.DAD).m_Key;
        h2.set(h.node(h.NODE),h2.NODE);
        h.set(h.brother(h.NODE),h.DAD);
    }
    return true;
}

class NodeHandler
    ...
    Node brother(int kind) {
        Node dad = node(kind+1);
        Node node = node(kind);
        return dad.m_Left == node ? dad.m_Right : dad.m_Left;
    }
...
```

1. Abstieg: Suche nach Schlüssel

existiert nicht: fertig

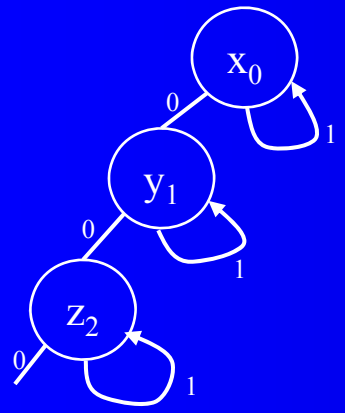
2. Abstieg: Suche nach dem Vater

kopieren des Schlüssels

Löschen des mittleren Knotens

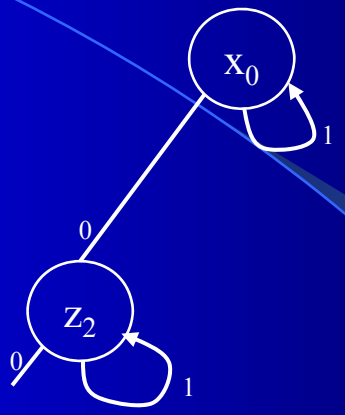
Umhängen des unteren Verweises

Patricia-Trees: Problem nach dem Löschen



löschen von y

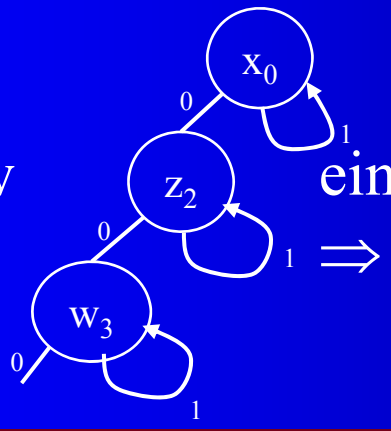
$\Rightarrow$



	3	2	1	0
x	_	_	_	1
y	_	_	1	0
z	_	1	0	0
w	1	0	1	0
u	1	1	1	0

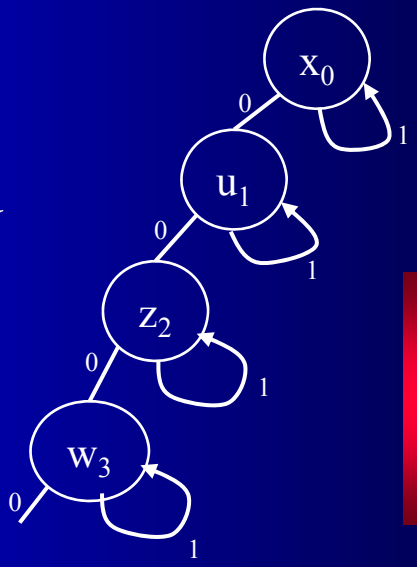
einfügen von w

$\Rightarrow$



einfügen von u

$\Rightarrow$



Fehler: w hätte mit Index 1 eingetragen werden müssen

w ist nicht mehr auffindbar

Patricia-Trees: Lösung für das Löschenproblem

- statt einfach nächsten Index beim "Null" Einfügen ...

```
public boolean insert(char c) {
    NodeHandler h = new NodeHandler(m_Root);
    h.search(c);
    int index = 0;
    if (h.isNull()) {
        if (h.node(h.DAD) != null)
            index = h.node(h.DAD).m_BitPos + 1;
    } else ...
```

- ... mit Vater vergleichen

```
public boolean insert(char c) {
...
    if (h.isNull()) {
        if (h.node(h.DAD) != null) {
            while (   left(c,index) == left(h.node(h.DAD).m_Key,index) &&
                      index < h.node(h.DAD).m_BitPos)
                ++index;
            if (index == h.node(h.DAD).m_BitPos)
                ++index;
        }
    } else ...
```

kleinster Index, in dem sich Vaterschlüssel und c unterscheiden

sonst nächster