2–3–4 Bäume

Problem:

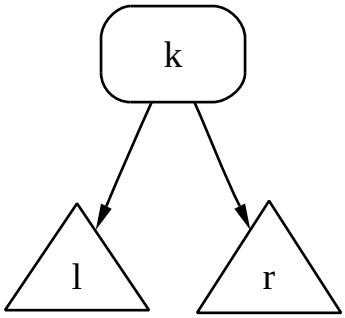
Binäre Bäume können entarten, wenn Schlüssel nicht zufällig sind.

Worst–Case Performance: $\mathcal{O}(n)$

Lösung: Aufweichung des Konzeptes binärer Bäume

Knoten können 0, 2, 3, oder 4 Schlüssel enthalten

1. 0–Knoten: leerer Baum, Höhe 0
2. 2–Knoten mit Söhnen der Höhe h gibt Baum der Höhe $h + 1$:



   (a) l ist 2–3–4 Baum der Höhe h

   (b) r ist 2–3–4 Baum der Höhe h

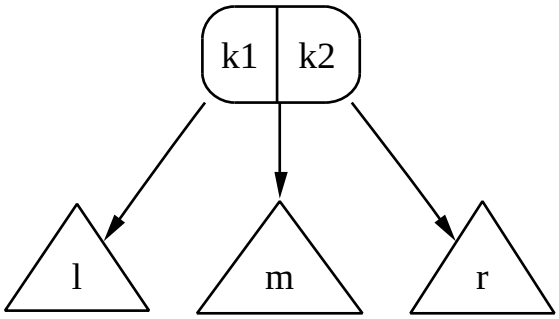
   (c) alle Schlüssel im Teilbaum l kleiner als k

   (d) k kleiner als alle Schlüssel im Teilbaum r

$$l < k < r$$

3–Knoten

1. 3–Knoten mit Söhnen der Höhe h gibt Baum der Höhe $h + 1$:



   (a) $k1 < k2$

   (b) alle Schlüssel im Teilbaum l kleiner als $k1$

   (c) $k1$ kleiner als alle Schlüssel im Teilbaum m

   (d) alle Schlüssel im Teilbaum m kleiner als $k2$

   (e) $k2$ kleiner als alle Schlüssel im Teilbaum r

   (f) l ist 2–3–4 Baum der Höhe h

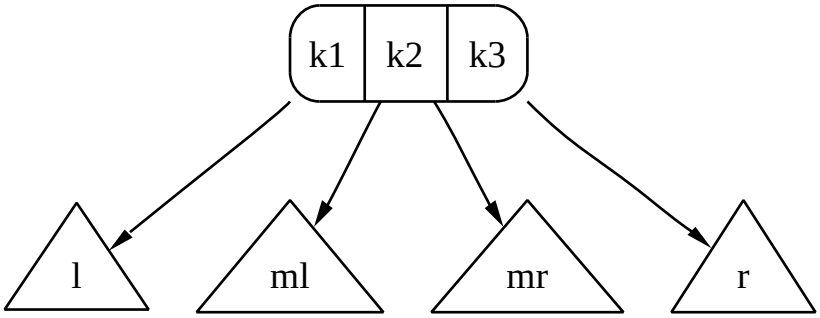
   (g) m ist 2–3–4 Baum der Höhe h

   (h) r ist 2–3–4 Baum der Höhe h

$$l < k1 < m < k2 < r$$

4–Knoten

4. 4–Knoten der Höhe $h + 1$



(a) $k1 < k2$

(b) $k2 < k3$

(c) alle Schlüssel im Teilbaum l kleiner als $k1$

(d) $k1$ kleiner als alle Schlüssel im Teilbaum ml

(e) alle Schlüssel im Teilbaum ml kleiner als $k2$

(f) $k2$ kleiner als alle Schlüssel im Teilbaum mr

(g) alle Schlüssel im Teilbaum mr kleiner als $k3$

(h) $k3$ kleiner als alle Schlüssel im Teilbaum r

(i) l ist 2–3–4 Baum der Höhe h

(j) ml ist 2–3–4 Baum der Höhe h

(k) mr ist 2–3–4 Baum der Höhe h

(l) r ist 2–3–4 Baum der Höhe h

$$l < k1 < ml < k2 < mr < k3 < r$$

Mögliche Repräsentation der Knoten in C

```
typedef enum { TWO, THREE, FOUR } NodeType;
typedef struct Node* NodePtr;

struct Node {
    NodeType type;
    Key      key1;
    Value    val1;
    Key      key2;
    Value    val2;
    Key      key3;
    Value    val3;
    NodePtr  ptr1;
    NodePtr  ptr2;
    NodePtr  ptr3;
    NodePtr  ptr4;
    unsigned label;   // used only for printing
};
typedef NodePtr Tree;
```

Interpretation von `type`:

1. `type == TWO`: 2–Knoten
2. `type == THREE`: 3–Knoten
3. `type == FOUR`: 4–Knoten

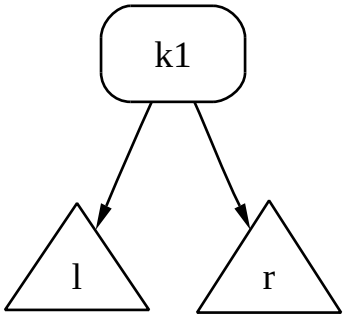
Vorteil dieser Repräsentation: konzeptuell einfach

Nachteil: verbraucht zuviel Platz, denn 2–Knoten verbraucht genauso viel Platz wie 4–Knoten.

Suche in 2–3–4–Bäumen

Suche konzeptuell wie in binären Bäumen, aber mehr Fälle:

1. 2–Knoten mit Schlüssel k

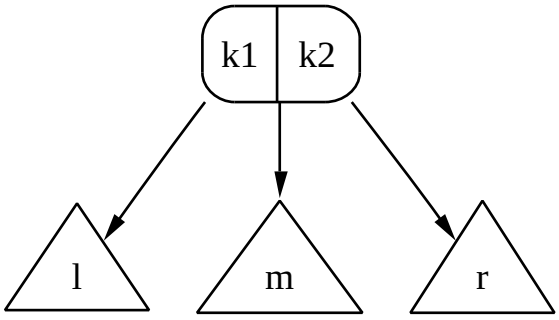


   (a) $k < k1$: Suche in l

   (b) $k = k1$: Fertig, Wert gefunden!

   (c) $k > k1$: Suche in r

2. 3–Knoten mit Schlüssel $k1$ und $k2$



   (a) $k < k1$: Suche in l

   (b) $k = k1$: Fertig, Wert gefunden!

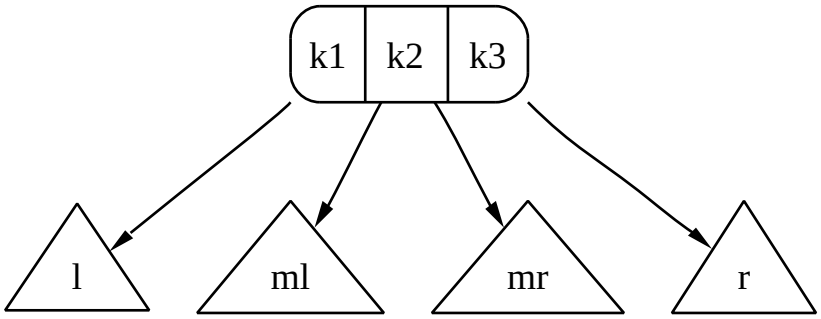
   (c) $k > k1 \wedge k < k2$: Suche in m

   (d) $k = k2$: Fertig, Wert gefunden!

   (e) $k > k2$: Suche in r

Suche in 2–3–4–Bäumen (Fortsetzung)

4. 4–Knoten mit Schlüssel $k1$ und $k2$



   (a) $k < k1$: Suche in l

   (b) $k = k1$: Fertig, Wert gefunden!

   (c) $k > k1 \wedge k < k2$: Suche in ml

   (d) $k = k2$: Fertig, Wert gefunden!

   (e) $k > k2 \wedge k < k3$: Suche in mr

   (f) $k = k3$: Fertig, Wert gefunden!

   (g) $k > k3$: Suche in r

Bei Suche sind pro Knoten bis zu 3 Vergleiche notwendig.

Aufwand der Suche: $\mathcal{O}(h)$, falls h Höhe des 2–3–4 Baums

Ein 2–3–4–Baum der Höhe h hat mindestens $n = 2^h - 1$ Knoten!

Also gilt $h = \log_2(n + 1)$, und damit haben wir:

Suche in 2–3–4–Baum immer logarithmisch!

Suche in 2–3–4–Bäumen (Implementierung)

```
// Search for a given key in the ordered 2-3-4 tree t.
Value* search(Table t, Key key)
{
    if (t == 0) return 0;
    switch (t->type) {
    case TWO: {
        int cmp = compareKey(key, t->key1);
        if (cmp == -1) {
            return search(t->ptr1, key);
        } else if (cmp == 0) {
            return &(t->val1);
        }
        assert(cmp == 1);
        return search(t->ptr2, key);
    }
    case THREE: {
        int cmp1 = compareKey(key, t->key1);
        if (cmp1 == -1) {
            return search(t->ptr1, key);
        } else if (cmp1 == 0) {
            return &(t->val1);
        }
        assert(cmp1 == 1);
        int cmp2 = compareKey(key, t->key2);
        if (cmp2 == -1) {
            return search(t->ptr2, key);
        } else if (cmp2 == 0) {
            return &(t->val2);
        }
        assert(cmp2 == 1);
        return search(t->ptr3, key);
    }
    ...
```

Implementierung der Suche (Fortsetzung)

```
    case FOUR: {
        int cmp1 = compareKey(key, t->key1);
        if (cmp1 == -1) {
            return search(t->ptr1, key);
        } else if (cmp1 == 0) {
            return &(t->val1);
        }
        assert(cmp1 == 1);
        int cmp2 = compareKey(key, t->key2);
        if (cmp2 == -1) {
            return search(t->ptr2, key);
        } else if (cmp2 == 0) {
            return &(t->val2);
        }
        assert(cmp2 == 1);
        int cmp3 = compareKey(key, t->key3);
        if (cmp3 == -1) {
            return search(t->ptr3, key);
        } else if (cmp3 == 0) {
            return &(t->val3);
        }
        assert(cmp3 == 1);
        return search(t->ptr4, key);
    }
    }
    //  added to avoid a compiler warning.
    assert(0);
    return 0;
}
```

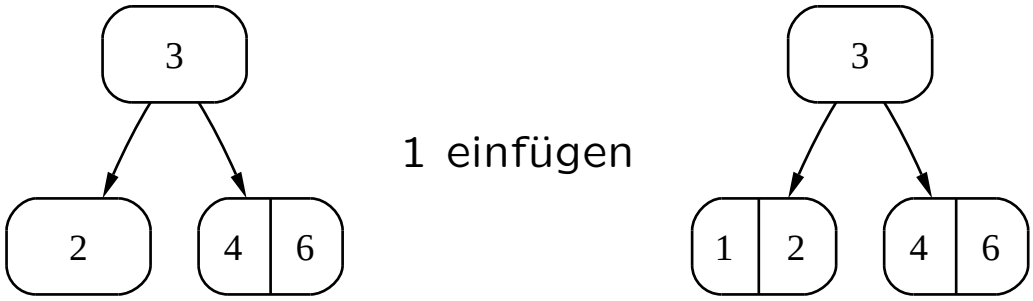
Einfügen in 2–3–4 Bäumen

Algorithmus zum Einfügen von gegebenen $\langle key, val \rangle$ Paar

1. Suche nach Schlüssel *key* bis Blatt erreicht ist
   (Ein Blatt ist ein 2–3–4–Baum der Höhe 1)

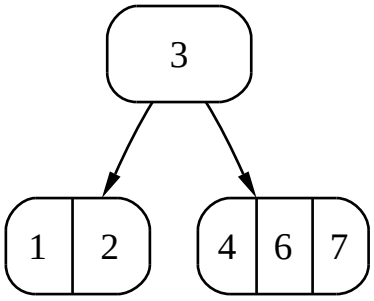
2. Fall–Unterscheidung nach Art des Blattes:

   (a) Blatt ist 2–Knoten: füge $\langle key, val \rangle$ ein
   Neues Blatt ist 3–Knoten



   (b) Blatt ist 3–Knoten: füge $\langle key, val \rangle$ ein
   Neues Blatt ist 4–Knoten

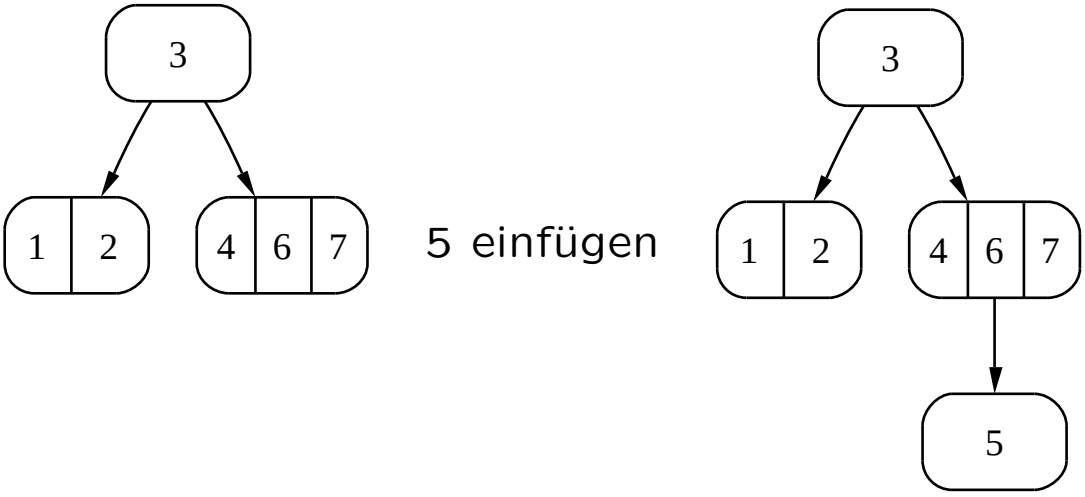
   Beispiel: oben 7 einfügen



3. Blatt ist 4–Knoten: Pech gehabt!

Einfügen in 2–3–4 Bäume (Fortsetzung)

Einfügen in 4–Knoten: Erster Versuch



Geht nicht: Teilbäume haben unterschiedliche Höhe!

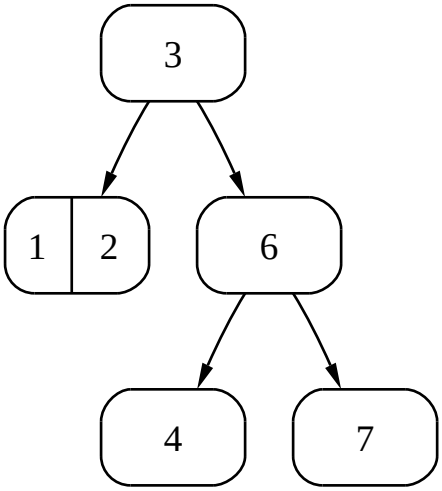
2–3–4 Bäume können nicht nach unten wachsen!

Zweiter Versuch: 4–Knoten aufspalten:

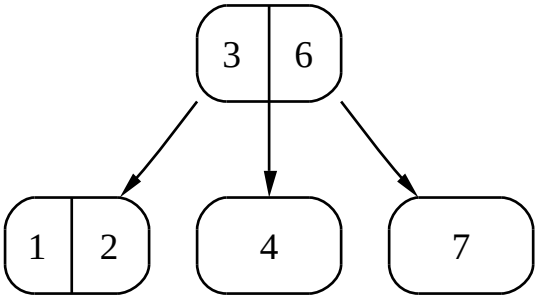
Dann Schlüssel 6 nach oben schieben!

Einfügen in 2–3–4 Bäume (Fortsetzung)

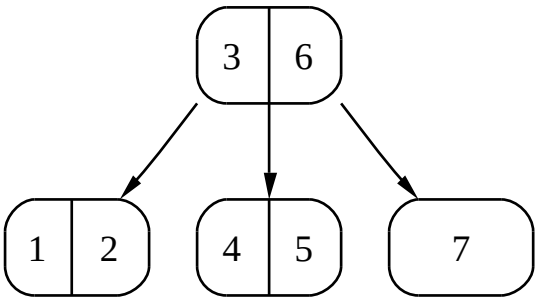
Schlüssel 6 nach oben schieben: Ausgangspunkt



Das ist kein 2–3–4 Baum, kann aber repariert werden, wenn Knoten mit 6 nach oben geschoben werden kann!



Jetzt einfügen von 5 möglich!



Einfügen in 2–3–4 Bäume (Fortsetzung)

Neuer Algorithmus

1. Suche nach Schlüssel *key* bis Blatt erreicht ist

   Falls bei Suche 4–Knoten erreicht wird:

   (a) Spalte 4–Knoten auf

   (b) Schiebe mittleren Schlüssel nach oben
   Frage: Warum geht das immer?
   (Problem wenn Knoten darüber 4–Knoten ist)

   Antwort: Algorithmus stellt sicher, dass Knoten über dem aktuellen Knoten nie 4–Knoten ist!

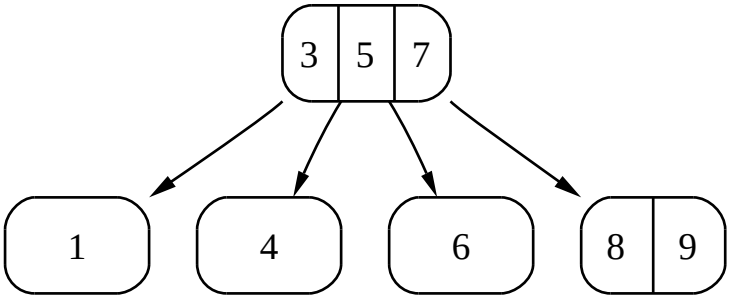
2. Falls Blatt erreicht ist, kann dies kein 4–Knoten sein:

   (a) Blatt ist 2–Knoten: füge $\langle key, val \rangle$ ein
   Neues Blatt ist 3–Knoten

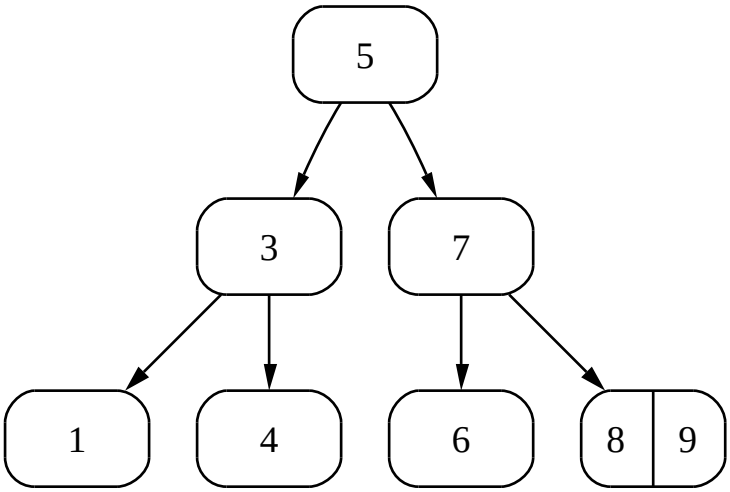
   (b) Blatt ist 3–Knoten: füge $\langle key, val \rangle$ ein
   Neues Blatt ist 4–Knoten

Baum wächst nur, wenn die Wurzel 4–Knoten ist und aufgespalten wird!

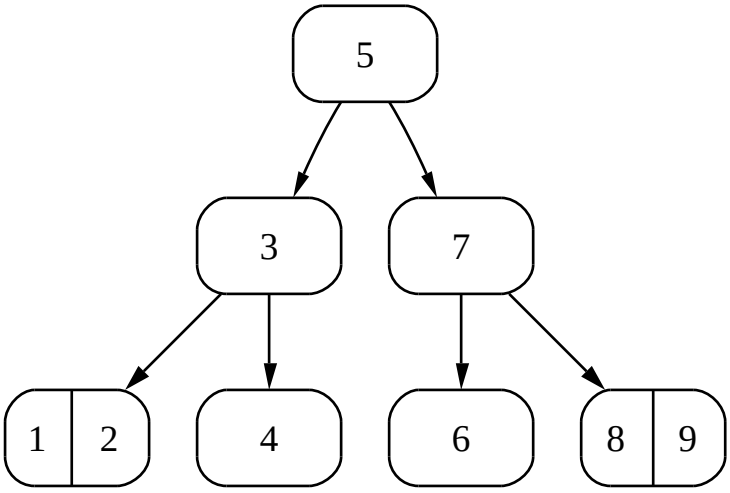
Wachstum von 2–3–4 Bäumen



1. Schritt: aufspalten aller 4–Knoten bei Suche:

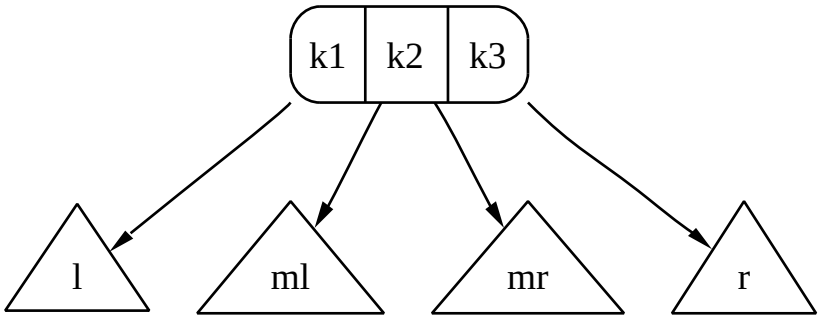


2. Schritt einfügen von 2

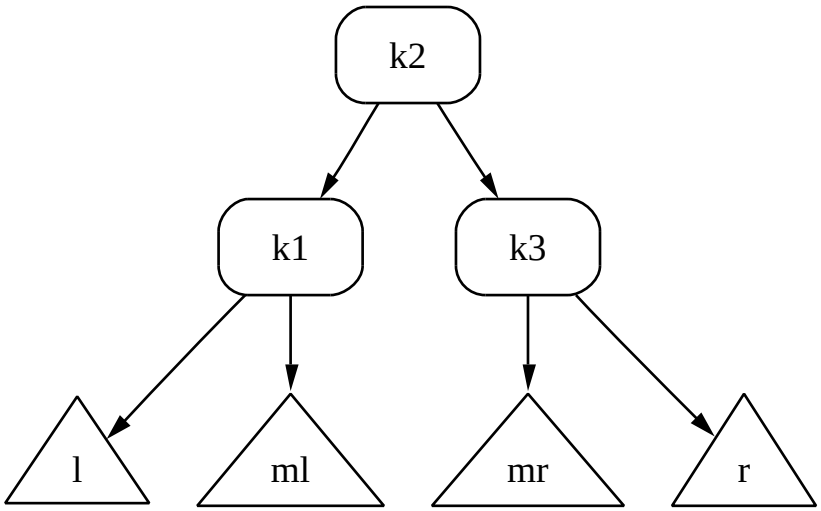


Aufspaltens von 4–Knoten

Der 4–Knoten



wird durch Aufspalten zu

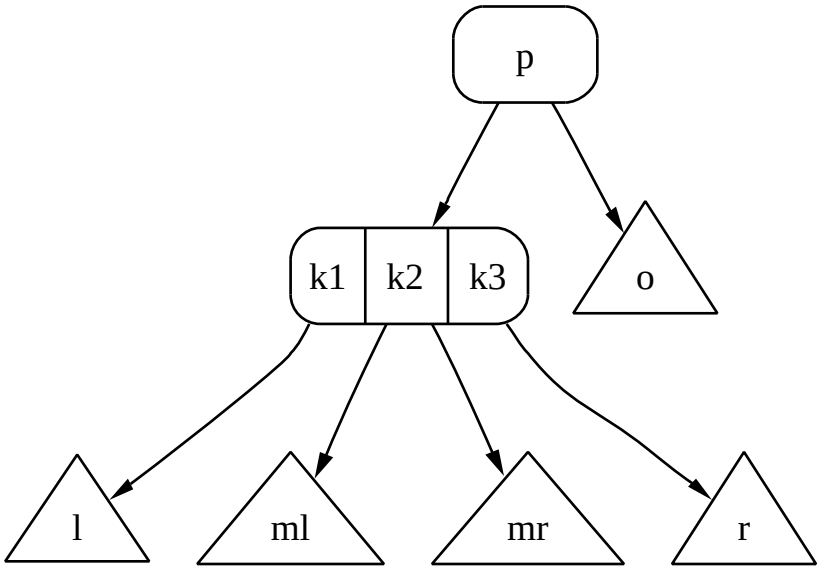


Implementierung des Aufspaltens

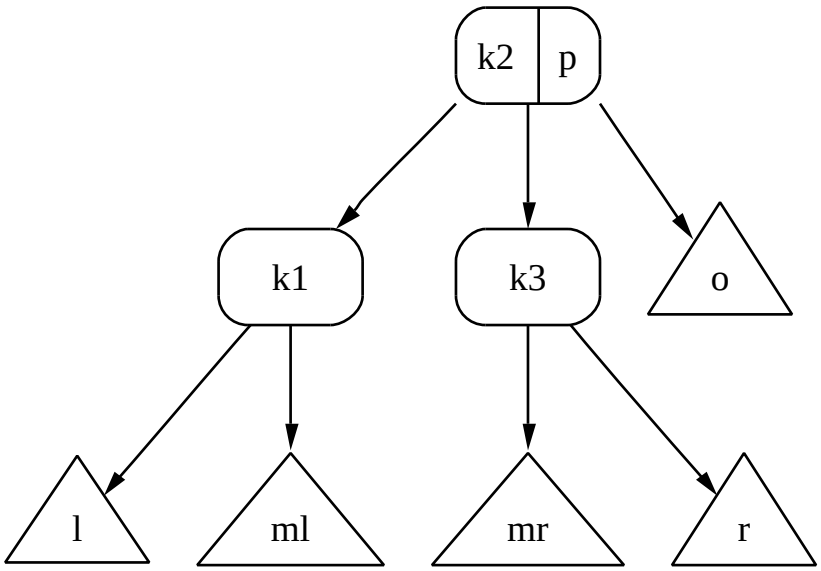
```
NodePtr split4Node(NodePtr t)
{
    assert(t->type == FOUR);
    // create the left node
    NodePtr left = malloc( sizeof(struct Node) );
    left->type = TWO;
    left->key1 = t->key1;
    left->val1 = t->val1;
    left->ptr1 = t->ptr1;
    left->ptr2 = t->ptr2;
    left->ptr3 = 0;
    left->ptr4 = 0;
    // create the right node
    NodePtr right = malloc( sizeof(struct Node) );
    right->type = TWO;
    right->key1 = t->key3;
    right->val1 = t->val3;
    right->ptr1 = t->ptr3;
    right->ptr2 = t->ptr4;
    right->ptr3 = 0;
    right->ptr4 = 0;
    // create the top node
    NodePtr top = malloc( sizeof(struct Node) );
    top->type = TWO;
    top->key1 = t->key2;
    top->val1 = t->val2;
    top->ptr1 = left;
    top->ptr2 = right;
    top->ptr3 = 0;
    top->ptr4 = 0;
    free(t);          // the old 4-node is disposed
    return top;
}
```

Knoten nach oben schieben

1. Fall: Vater–Knoten ist 2–Knoten, 4–Knoten ist links:

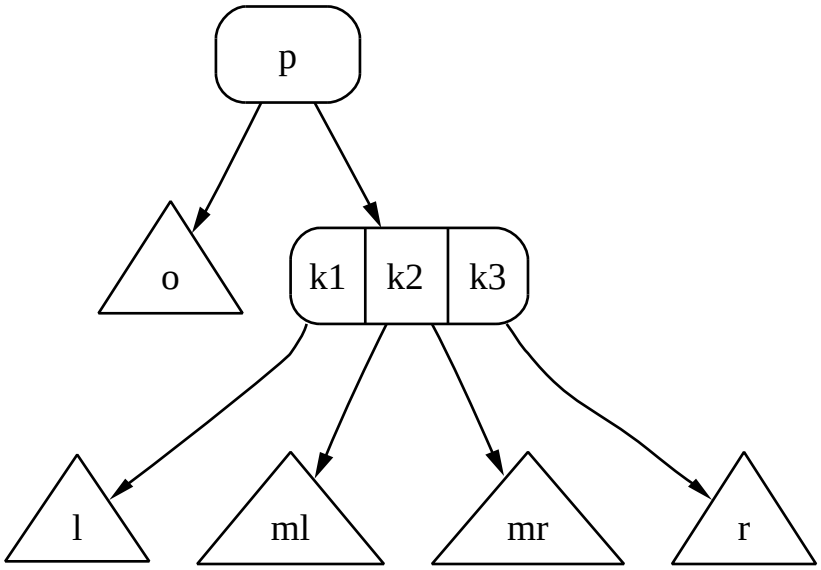


Aufspalten und Knoten nach oben schieben:

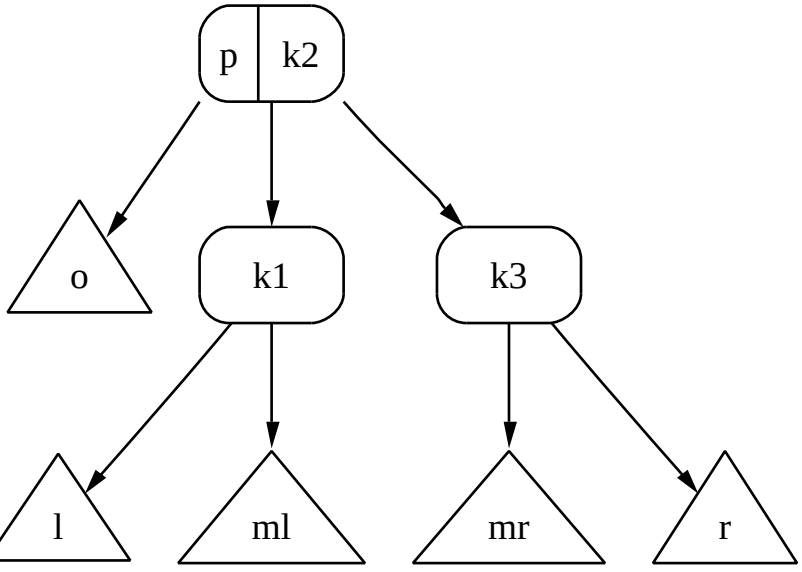


Knoten nach oben schieben

2. Fall: Vater–Knoten ist 2–Knoten, 4–Knoten ist rechts:

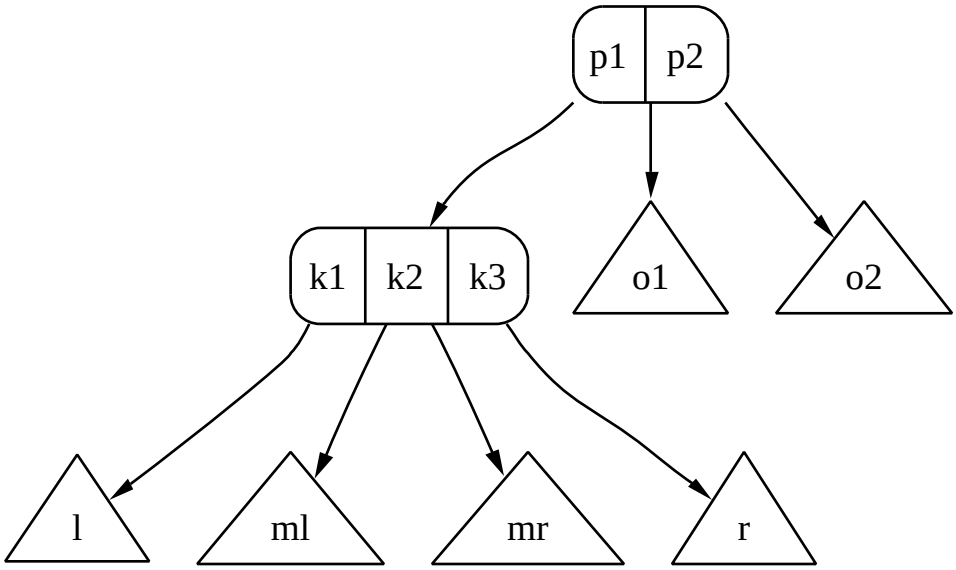


Aufspalten und Knoten nach oben schieben:

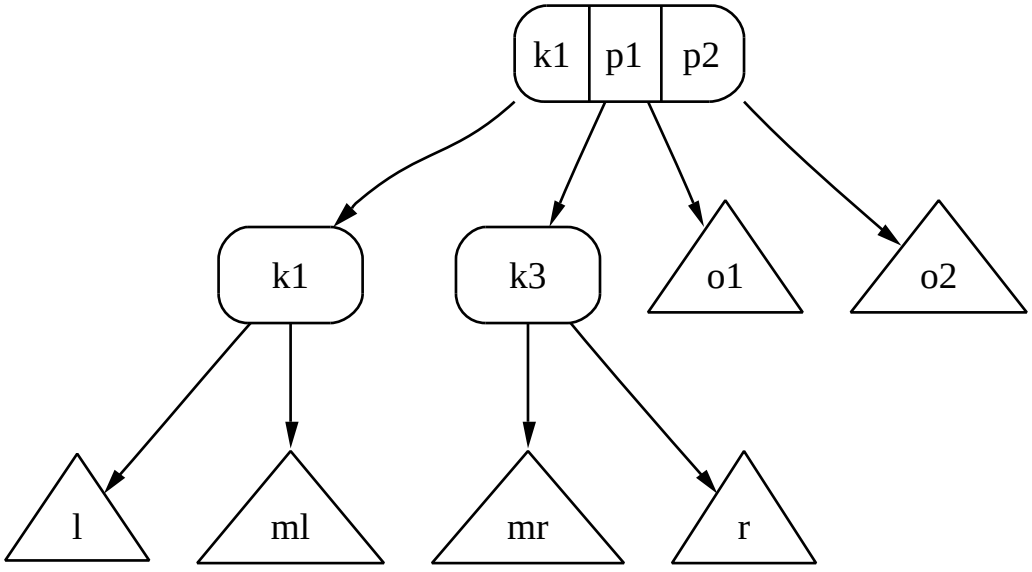


Knoten nach oben schieben

3. Fall: Vater–Knoten ist 3–Knoten, 4–Knoten ist 1. Sohn:

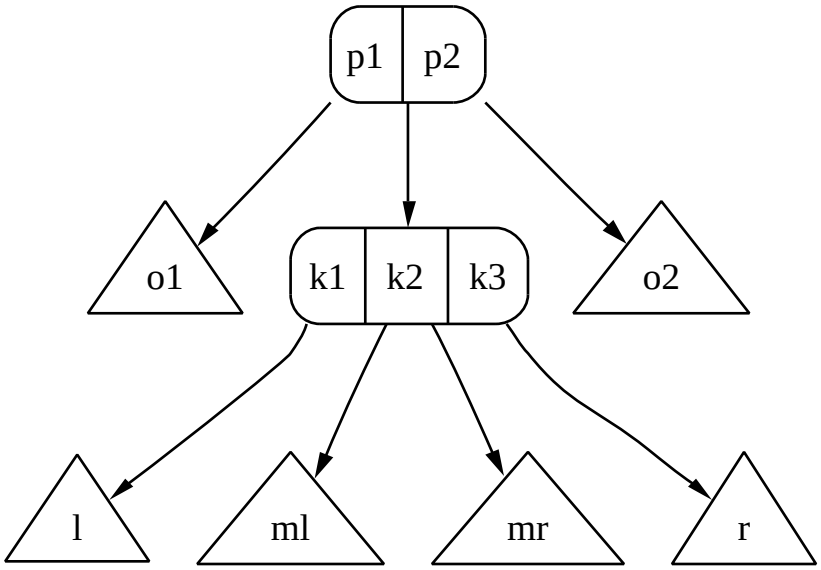


Aufspalten und Knoten nach oben schieben:

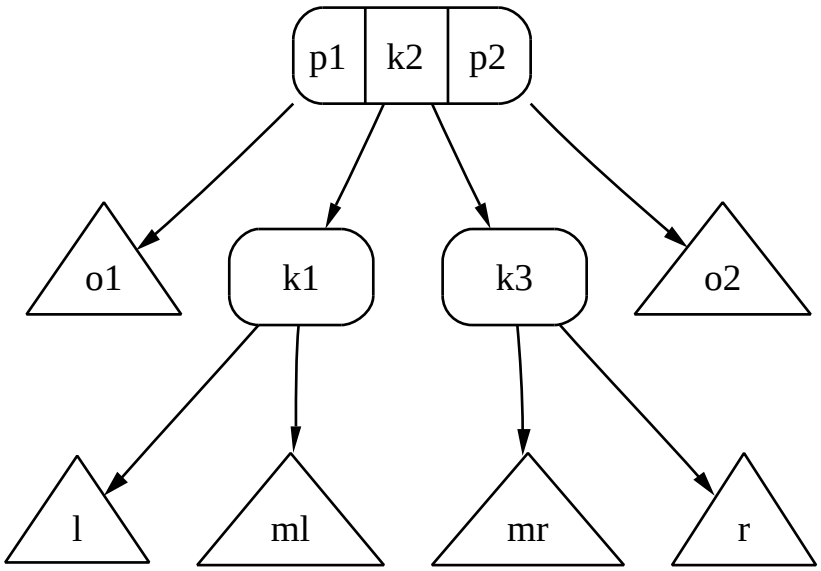


Knoten nach oben schieben

4. Fall: Vater–Knoten ist 3–Knoten, 4–Knoten ist 2. Sohn:

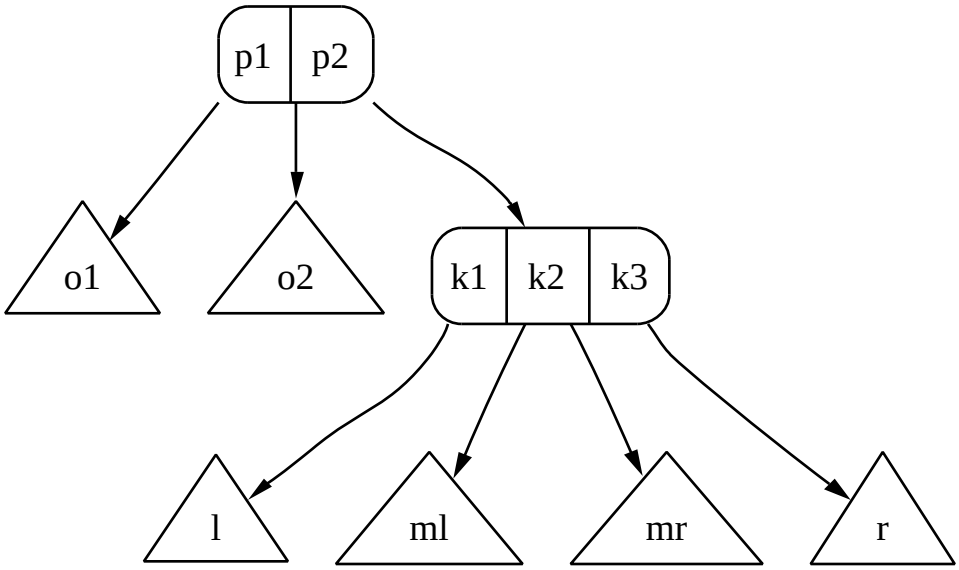


Aufspalten und Knoten nach oben schieben:

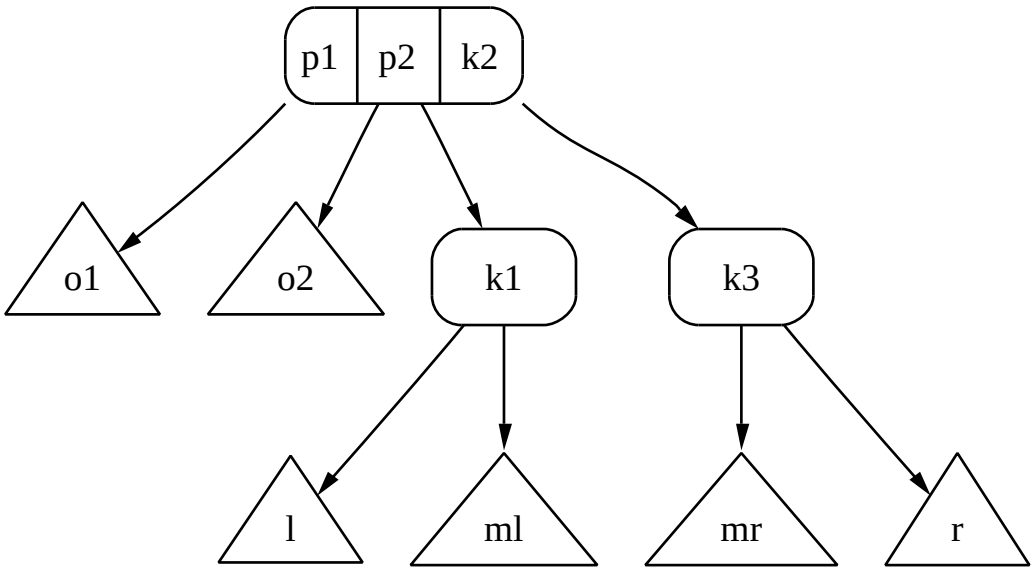


Knoten nach oben schieben

5. Fall: Vater–Knoten ist 3–Knoten, 4–Knoten ist 3. Sohn:



Aufspalten und Knoten nach oben schieben:



Knoten nach oben schieben (Implementierung)

Knoten ist 1. Sohn seines Vaters

```
void moveUp1(NodePtr t, NodePtr n)
{
    assert(t != 0);
    assert(t->type != FOUR);
    assert(n->type == TWO);
    if (t->type == TWO) {
        t->key2 = t->key1;
        t->val2 = t->val1;
        t->key1 = n->key1;
        t->val1 = n->val1;
        t->ptr3 = t->ptr2;
        t->ptr1 = n->ptr1;
        t->ptr2 = n->ptr2;
        t->type = THREE;
        free(n);
        return;
    }
    assert(t->type == THREE);
    t->key3 = t->key2;
    t->val3 = t->val2;
    t->key2 = t->key1;
    t->val2 = t->val1;
    t->key1 = n->key1;
    t->val1 = n->val1;
    t->ptr4 = t->ptr3;
    t->ptr3 = t->ptr2;
    t->ptr2 = n->ptr2;
    t->ptr1 = n->ptr1;
    t->type = FOUR;
    free(n);
    return;
}
```

Knoten nach oben schieben (Implementierung)

Knoten ist 2. Sohn seines Vaters

```
void moveUp2(NodePtr t, NodePtr n)
{
    assert(t != 0);
    assert(t->type != FOUR);
    assert(n->type == TWO);
    if (t->type == TWO) {
        t->key2 = n->key1;
        t->val2 = n->val1;
        t->ptr2 = n->ptr1;
        t->ptr3 = n->ptr2;
        t->type = THREE;
        free(n);
        return;
    }
    assert(t->type == THREE);
    t->key3 = t->key2;
    t->val3 = t->val2;
    t->key2 = n->key1;
    t->val2 = n->val1;
    t->ptr4 = t->ptr3;
    t->ptr3 = n->ptr2;
    t->ptr2 = n->ptr1;
    t->type = FOUR;
    free(n);
    return;
}
```

Knoten nach oben schieben (Implementierung)

Knoten ist 3. Sohn seines Vaters

```
void moveUp3(NodePtr t, NodePtr n)
{
    assert(t != 0);
    assert(t->type == THREE);
    assert(n->type == TWO);
    t->key3 = n->key1;
    t->val3 = n->val1;
    t->ptr4 = n->ptr2;
    t->ptr3 = n->ptr1;
    t->type = FOUR;
    free(n);
    return;
}
```

Auspalten von Knoten ist selten: Wird ein Knoten eingefügt, so muß im Mittel weniger als 1 Knoten aufgespaltet werden!

Einfügen von Knoten (Implementierung)

Einfügen im Blatt, betrachte zunächst 2–Knoten

```
void addKey(Tree p, Key key, Value val) {
    if (p->type == TWO) {
        int cmp = compareKey(key, p->key1);
        if (cmp == -1) {
            // move the old key to the right
            p->key2 = p->key1;
            p->val2 = p->val1;
            // insert the new key as first key
            p->key1 = key;
            p->val1 = val;
        } else {
            assert(cmp == 1);
            // insert the new key as second key
            p->key2 = key;
            p->val2 = val;
        }
        p->type = THREE;
        return;
    }
    ...
```

Einfügen im Blatt (3–Knoten)

```
    ...
    assert (p->type == THREE);
    int cmp1 = compareKey(key, p->key1);
    if (cmp1 == -1) {
        // move the old keys to the right
        p->key3 = p->key2;
        p->val3 = p->val2;
        p->key2 = p->key1;
        p->val2 = p->val1;
        // insert the new key as first key
        p->key1 = key;
        p->val1 = val;
    } else {
        assert(cmp1 == 1);
        int cmp2 = compareKey(key, p->key2);
        if (cmp2 == -1) {
            // move the second key right
            p->key3 = p->key2;
            p->val3 = p->val2;
            // insert the new key as second key
            p->key2 = key;
            p->val2 = val;
        } else {
            assert(cmp2 == 1);
            // insert the new key at the end
            p->key3 = key;
            p->val3 = val;
        }
    }
    p->type = FOUR;
    return;
}
```

Einfügen: Algorithmus

Gegeben:

1. `t`: Zeiger auf Wurzel–Knoten eines 2–3–4 Baums
2. `key`: Schlüssel, der eingefügt werden soll
3. `val`: Wert, der mit Schlüssel assoziiert ist

Gesucht: 2–3–4 Baum, der aus `*t` durch Einfügen von $\langle$`key`, `val`$\rangle$ hervorgeht.

Algorithmus:

1. Fall: Gegebener Baum ist leer
   Erzeuge neuen 2–Knoten mit Schlüssel `key`
2. Fall: Wurzel des gegebenen Baums ist 2–Knoten
   (a) Lokalisiere Teil–Baum, in den eingefügt wird
   (b) Falls Wurzel des Teil–Baums 4–Knoten: spalten
   (c) Füge Schlüssel passend ein:
   - Keine Aufspaltung: in Wurzel Teilbaum
   - Aufspaltung: in gegebenem Baum `*t`
3. Fall: Wurzel des gegebenen Baums ist 3–Knoten
   analog zum 2. Fall
4. Fall: Wurzel des gegebenen Baums ist 4–Knoten
   (a) Spalte Knoten auf
   (b) Füge Schlüssel in neuer Wurzel ein

Einfügen: Implementierung

```
Table insert(Tree t, Key key, Value val) {
    // If the tree *t is empty, create a new node.
    if (t == 0) {
        NodePtr node = malloc( sizeof(struct Node) );
        node->type = TWO;
        node->key1 = key;
        node->val1 = val;
        node->ptr1 = 0;
        node->ptr2 = 0;
        node->ptr3 = 0;
        node->ptr4 = 0;
        return node;
    }
    switch (t->type) {
    // find the appropriate subtree where the new key
    // needs to be inserted and split the root node
    // of this subtree if necessary
    case TWO: {
        int cmp = compareKey(key, t->key1);
        if (cmp == -1) {
            t = splitAndInsert(t, t->ptr1, key, val);
        } else if (cmp == 0) {
            t->val1 = val;
        } else {
            assert(cmp == 1);
            t = splitAndInsert(t, t->ptr2, key, val);
        }
        return t;
    }
    ...
```

Einfügen: Implementierung

Einfügen in 3–Knoten und 4–Knoten

```
    ...
    case THREE: {
        int cmp1 = compareKey(key, t->key1);
        if (cmp1 == -1) {
            t = splitAndInsert(t, t->ptr1, key, val);
        } else if (cmp1 == 0) {
            t->val1 = val;
        } else {
            assert(cmp1 == 1);
            int cmp2 = compareKey(key, t->key2);
            if (cmp2 == -1) {
                t = splitAndInsert(t, t->ptr2, key, val);
            } else if (cmp2 == 0) {
                t->val2 = val;
            } else {
                assert(cmp2 == 1);
                t = splitAndInsert(t, t->ptr3, key, val);
            }
        }
        return t;
    }
    case FOUR: {
        NodePtr n = split4Node(t);
        return insert(n, key, val);
    }
    }
    // avoid compiler warning about missing return
    assert(0);
    return 0;
}
```

Aufspalten und Einfügen: Algorithmus

Gegeben:

1. `p`, `t`: Knoten in 2–3–4 Baum
2. `p` is Vater von `t`
3. `key`: Schlüssel, der in `*t` eingefügt werden soll
4. `val`: zugehöriger Wert

Algorithmus:

1. Fall: `*t` ist leerer Baum
   $\langle$`key`, `val`$\rangle$ wird Blatt `*p` eingefügt
2. Fall: `*t` ist 4–Knoten
   (a) Spalte Knoten auf
   ```
   NodePtr n = split4Node(t)
   ```
   (b) Füge $\langle$`key`, `val`$\rangle$ in `n` ein:
   ```
   insert(n, key, val)
   ```
   (c) Schiebe Knoten n nach oben zu Vater–Knoten
   `moveUp`i`(p, n)` mit $i \in \{1, 2, 3\}$
3. Fall: Sonst füge $\langle$`key`, `val`$\rangle$ in `t` ein:
   ```
   insert(n, key, val)
   ```

Aufspalten und Einfügen: Implementierung

```
Tree splitAndInsert(NodePtr p, NodePtr t, Key key, Value val
{
    // If the parent is a leaf, the key is inserted in
    // the parent.
    if (t == 0) {
        addKey(p, key, val);
        return p;
    }
    // If the root of the subtree where key is to be
    // inserted is a 4-node, split the 4-node first,
    // insert the key into the resulting tree, and
    // finally move the middle key of the resulting
    // tree into the parent node p.
    if (t->type == FOUR) {
        NodePtr n = split4Node(t);
        insert(n, key, val);
        if (t == p->ptr1) {
            moveUp1(p, n);
        } else if (t == p->ptr2) {
            moveUp2(p, n);
        } else {
            assert(t == p->ptr3);
            moveUp3(p, n);
        }
        return p;
    }
    insert(t, key, val);
    return p;
}
```

Darstellung der Knoten als Union

```
typedef enum { TWO, THREE, FOUR } NodeType;
typedef struct Node* NodePtr;

typedef struct TwoNode*   TwoNodePtr;
typedef struct ThreeNode* ThreeNodePtr;
typedef struct FourNode*  FourNodePtr;

struct TwoNode {
    Key      key;
    Value    val;
    NodePtr  left;
    NodePtr  right;
    unsigned label;  // used for visualization
};

struct ThreeNode {
    Key      key1;
    Value    val1;
    Key      key2;
    Value    val2;
    NodePtr  left;
    NodePtr  middle;
    NodePtr  right;
    unsigned label;  // used for visualization
};
```

Darstellung der Knoten als Union (Fortsetzung)

```
struct FourNode {
    Key      key1;
    Value    val1;
    Key      key2;
    Value    val2;
    Key      key3;
    Value    val3;
    NodePtr  left;
    NodePtr  middleLeft;
    NodePtr  middleRight;
    NodePtr  right;
    unsigned label;  // used for visualization
};

struct Node {
    NodeType type;
    union {
        TwoNodePtr   twoNodePtr;
        ThreeNodePtr threeNodePtr;
        FourNodePtr  fourNodePtr;
    } node;
};

typedef NodePtr Tree;
typedef Tree    Table;
```

Löschen in 2–3–4 Bäumen

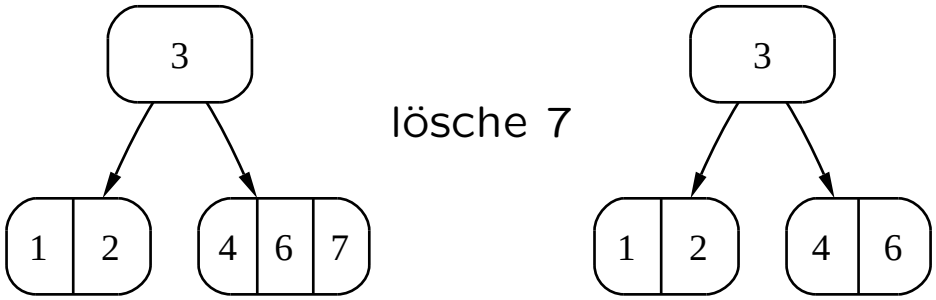
Algorithmus zum Löschen von gegebenen Schlüssel k

Betrachte zunächst nur den Fall, dass k in Blatt vorkommt

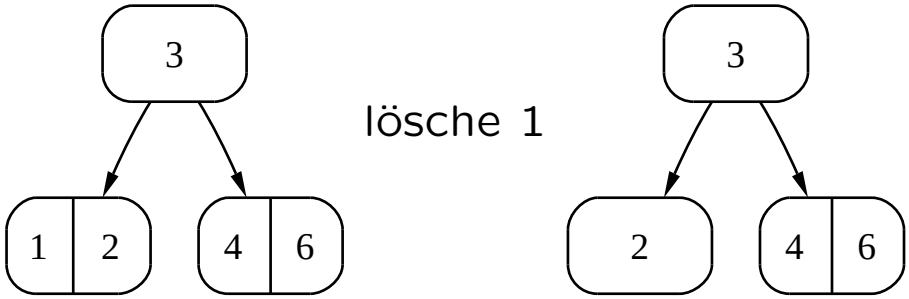
1. Suche nach Schlüssel k bis Blatt erreicht ist

2. Fall–Unterscheidung nach Art des Blattes:

   (a) Blatt ist 4–Knoten:
   Entferne Schlüssel, neues Blatt 3–Knoten



   (b) Blatt ist 3–Knoten:
   Entferne Schlüssel, neues Blatt 2–Knoten



3. Blatt ist 2–Knoten: Pech gehabt!

Behandlung von 2–Knoten beim Löschen

1. Beim Einfügen waren 4–Knoten das Problem

2. Lösung beim Einfügen:

   (a) Alle 4–Knoten auf dem Suchpfad werden aufgespalten.

   (b) Der dabei entstehende neue Knoten wird im Baum nach oben geschoben und mit seinem Vater verschmolzen.

3. Lösen beim Löschen analog:

   (a) Alle 2–Knoten auf dem Suchpfad werden in 3–Knoten oder 4–Knoten transformiert.

   (b) Wenn dann Blatt erreicht wird, kann dies kein 2–Knoten sein.

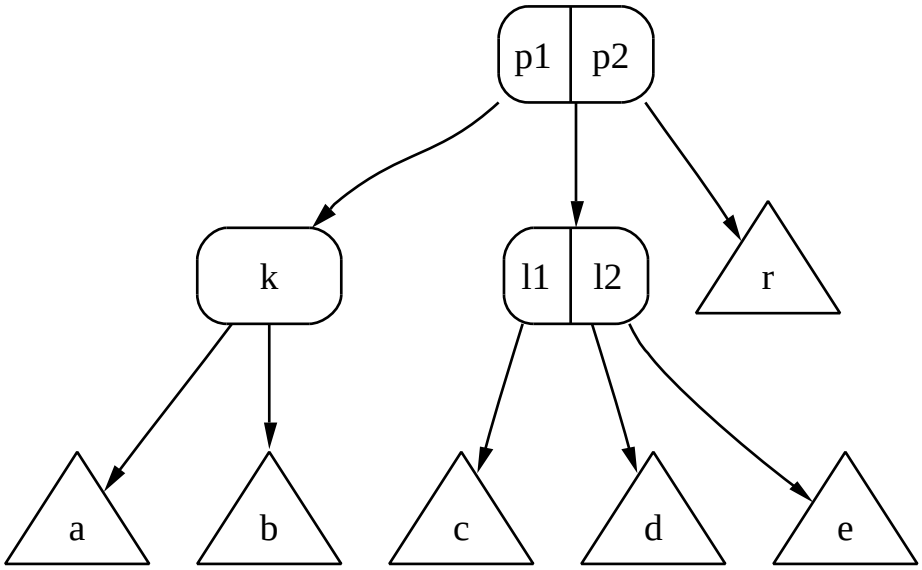
Um 2–Knoten in 3–Knoten oder 4–Knoten zu transformieren, gibt es prinzipiell zwei Möglichkeiten:

1. Stehlen eines Schlüssels von unmittelbar benachbartem Schwester–Knoten

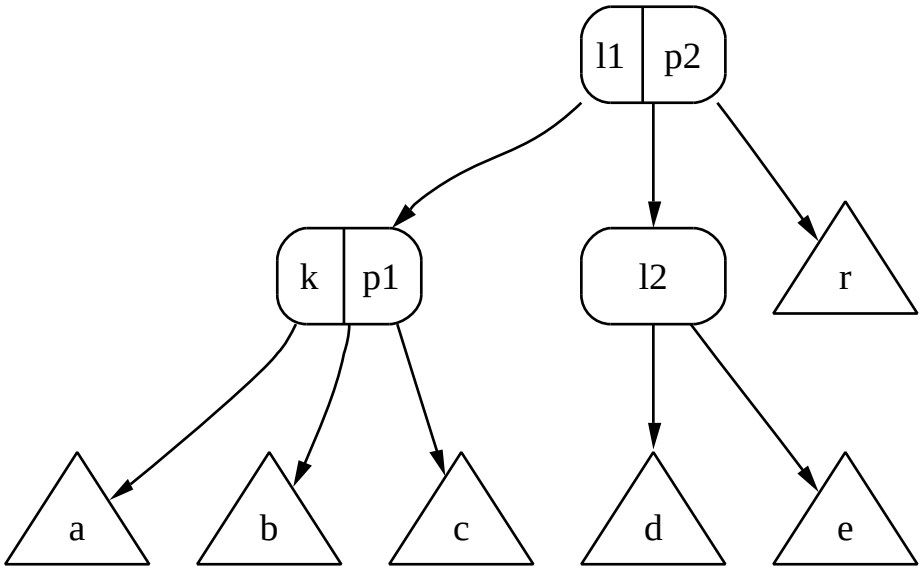
2. Fusionieren mit unmittelbar benachbartem Schwester–Knoten

Stehlen eines Schlüssels

Knoten mit Schüssel k in 3–Knoten transformieren:

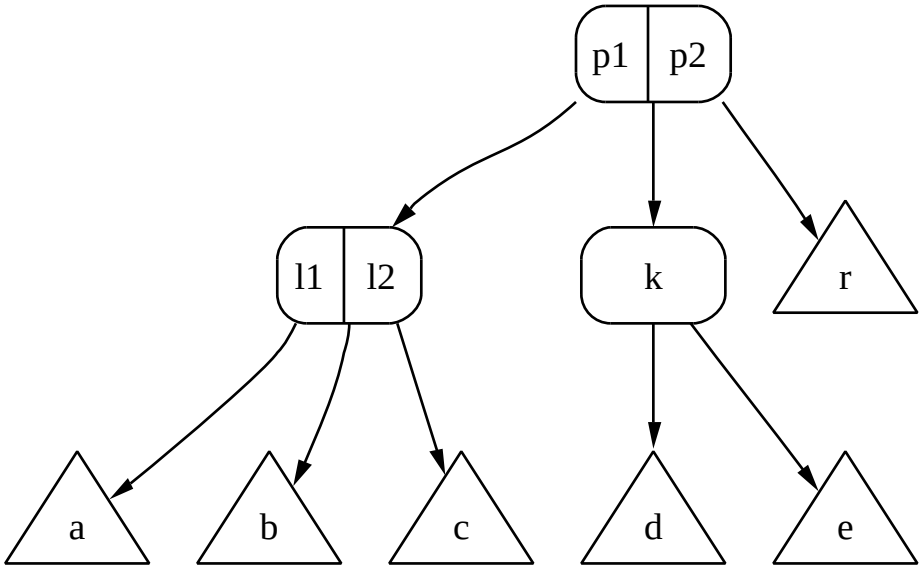


Stehlen von $l1$ aus Nachbar–Knoten (Rotation)

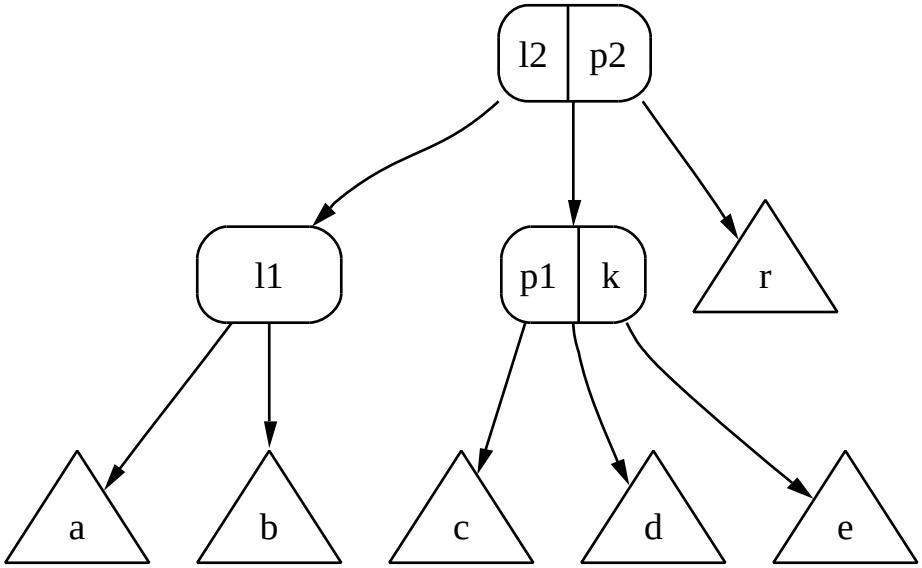


Stehlen eines Schlüssels (von links)

Knoten mit Schüssel k in 3–Knoten transformieren:

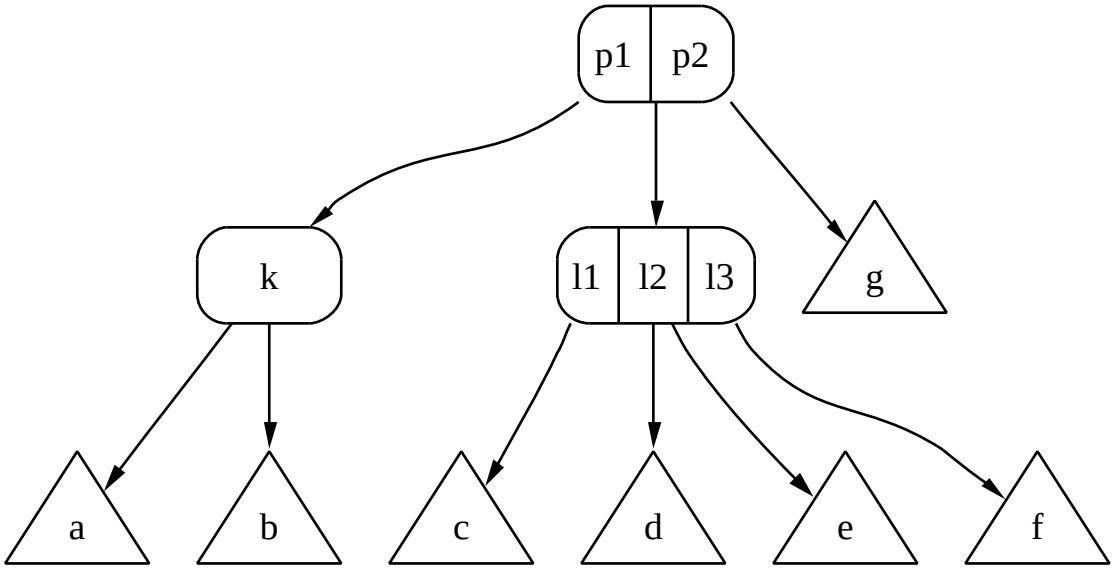


Stehlen von $l1$ aus Nachbar–Knoten (Rotation)

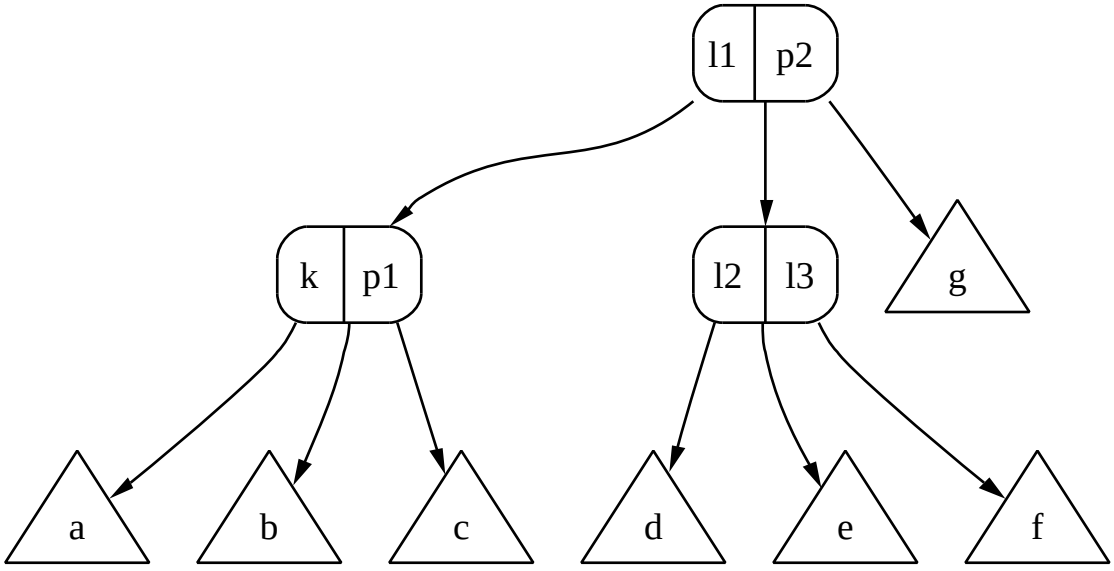


Stehlen eines Schlüssels (von 4–Knoten)

Knoten mit Schüssel k in 3–Knoten transformieren:



Stehlen von $l1$ aus Nachbar–Knoten (Rotation)



Stehlen eines Schlüssels

Beim Stehlen eines Schlüssels sind eine Reihe von Fallunterscheidungen durchzuführen

1. Der Vater–Knoten ist ein 3–Knoten
   (a) Der unmittelbare Bruder ist erstes Kind
   (b) Der unmittelbare Bruder ist zweites Kind
   (c) Der unmittelbare Bruder ist drittes Kind

   Jeder dieser Fälle spaltet sich in 2 Unterfälle auf:
   - Der unmittelbare Bruder ist 3–Knoten
   - Der unmittelbare Bruder ist 4–Knoten

2. Der Vater–Knoten ist ein 4–Knoten
   (a) Der unmittelbare Bruder ist erstes Kind
   (b) Der unmittelbare Bruder ist zweites Kind
   (c) Der unmittelbare Bruder ist drittes Kind
   (d) Der unmittelbare Bruder ist viertes Kind

   Auch hier haben wir die selbe zusätzliche Aufspaltung wie oben

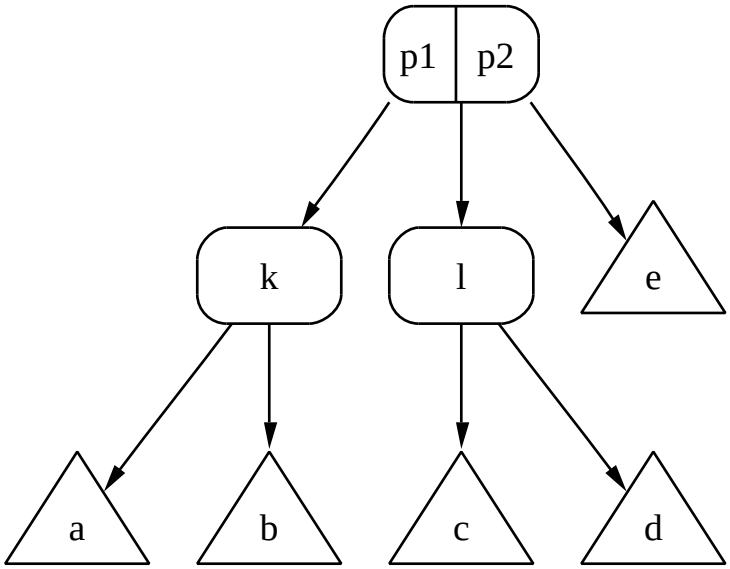
Zusätzlich muß noch unterschieden werden, ob der Bruder rechts oder links von dem Knoten liegt, der den Schlüssel stehlen will.

Insgesamt: 20 Fälle!

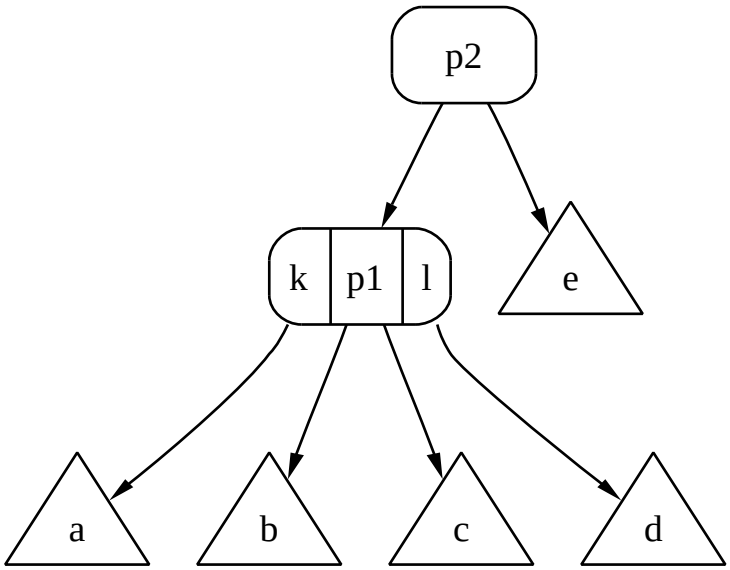
Fusionieren

Knoten mit Schüssel k in 3–Knoten transformieren:

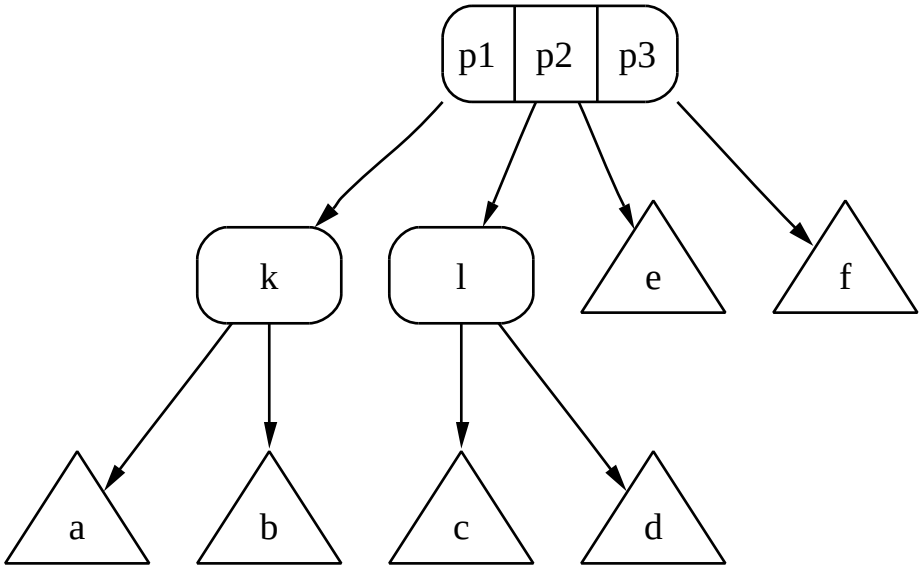
Stehlen nicht möglich, weil unmittelbarer Bruder 2–Knoten



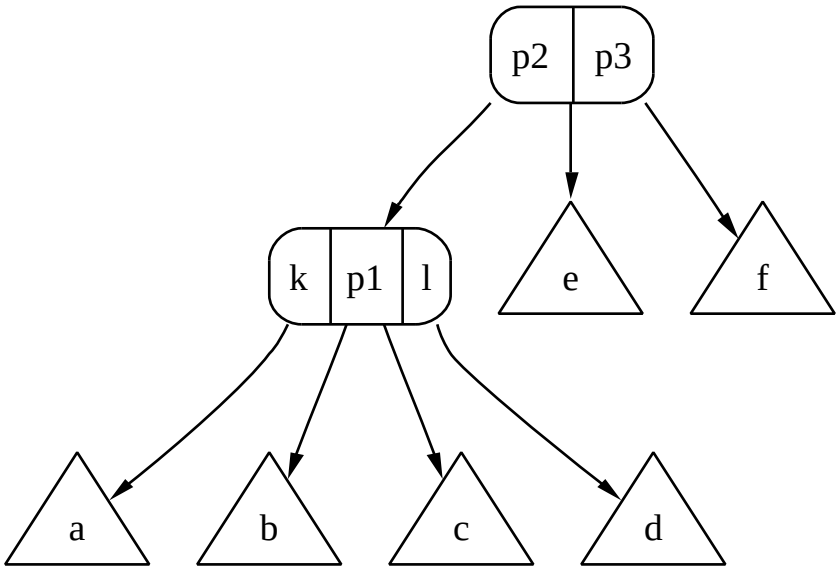
Knoten k mit Bruder verschmelzen



Fusionieren (Vater ist 4–Knoten)



Knoten k mit Bruder verschmelzen



Fusionieren

Beim Fusionieren sind eine Reihe von Fallunterscheidungen durchzuführen

1. Der Vater–Knoten ist ein 3–Knoten
   (a) Der unmittelbare Bruder ist erstes Kind
   (b) Der unmittelbare Bruder ist zweites Kind
   (c) Der unmittelbare Bruder ist drittes Kind
2. Der Vater–Knoten ist ein 4–Knoten
   (a) Der unmittelbare Bruder ist erstes Kind
   (b) Der unmittelbare Bruder ist zweites Kind
   (c) Der unmittelbare Bruder ist drittes Kind
   (d) Der unmittelbare Bruder ist viertes Kind

Zusätzlich muß noch unterschieden werden, ob der Bruder rechts oder links von dem Knoten liegt, der den Schlüssel stehlen will.

Insgesamt: 10 Fälle!

Löschen Allgemein

Bisher beschrieben: Löschen von Blatt–Knoten

Jetzt: Löschen von beliebigen Knoten

1. Ist Knoten m Blatt?

2. Ja: lösche Knoten m, fertig

   Nein:

3. Bestimme Nachfolger–Knoten n von m

   Sei k Schlüssel von n. Nachfolger–Knoten ist der Knoten n mit Schlüssel l für den gilt:

   (a) $k < l$

   (b) l ist der kleinste Knoten im Baum mit $k < l$.

4. Lösche n (möglich, da o Blatt)

5. Schreibe l und v in Knoten m

Komplexität: Ist t eine 2–3–4–Baum mit n Knoten, so beträgt der Aufwand für das Löschen eines Knotens sowohl im schlechtesten als auch im besten Fall

$$\mathcal{O}\big(\log(n)\big)$$

Löschen: Alternative Implementierung

1. Führe zusätzliches Felder vom Type `bool` in der Struktur ein, die Knoten beschreibt.

```
struct Node {
    NodeType type;
    bool     alive1;
    Key      key1;
    Value    val1;
    bool     alive2;
    Key      key2;
    Value    val2;
    bool     alive3;
    Key      key3;
    Value    val3;
    ...
};
```

2. `alive == true`: Schlüssel ist gültig.

3. `alive == false`: Schlüssel wurde gelöscht.

4. Verwalte außerdem zwei globale Zähler:
   (a) Anzahl aller gültigen Schlüssel
   (b) Anzahl aller gelöschten Schlüssel

5. Falls mehr gelöschte Schlüssel als gültige Schlüssel existieren, baue Baum neu auf.

   Komplexität für Baum mit n Knoten:

   (a) Im schlimmsten Fall: $\mathcal{O}(n)$
   (b) Im statistischen Durchschnitt: $\mathcal{O}\big(\log(n)\big)$

Löschen Alternative (Fortsetzung)

1. Zahl gelöschter Schlüssel größer Zahl gültiger Schlüssel?
2. Nein: Schlüssel suchen und als gelöscht markieren
3. Ja:
   (a) Sammle alle gültigen Schlüssel (nebst zugehörige Werte) aus dem Baum in 2 Feldern auf.
   (b) Lösche alten Baum.
   (c) Baue neuen 2–3–4 Baum aus diesen Feldern auf.

Aufsammeln:

```
void collectKeys(Tree t, unsigned* idx,
                 Key keyArray[], Value valArray[])
{
    if (t == 0)
        return;
    switch (t->type) {
    case TWO: {
        collectKeys(t->ptr1, idx, keyArray, valArray);
        if (t->alive1) {
            keyArray[*idx] = t->key1;
            valArray[*idx] = t->val1;
            *idx = *idx + 1;
        }
        collectKeys(t->ptr2, idx, keyArray, valArray);
        return;
    }
    ...
```

Baum neu Aufbauen

Gegeben:

1. `keyArray`: **geordnetes** Feld von Schlüsseln
2. `valArray`: Feld der zugehörigen Schlüssel
3. `length`: Länge der Felder

Gesucht: 2–3–4 Baum mit den Schlüsseln aus `keyArray`

Algorithmus

1. `length == 1`: erzeuge 2–Knoten
2. `length == 2`: erzeuge 3–Knoten
3. `length % 2 == 1, length > 2`:
   (a) Setze `m = length/2`
   (b) Rekursiv: Bilde Baum l aus

$$\underbrace{\texttt{keyArray[0]}, \cdots, \texttt{keyArray[m-1]}}_{\texttt{m}}$$

   (c) Rekursiv: Bilde Baum r aus

$$\underbrace{\texttt{keyArray[m+1]}, \cdots, \texttt{keyArray[m + m]}}_{\texttt{m}}$$

   (d) Erzeuge 2–Knoten n aus

$$l,\ r,\ \texttt{keyArray[m]},\ \texttt{valArray[m]}$$

   (e) Gebe n zurück

Baum neu Aufbauen (Fortsetzung)

4. `length % 2 == 0, length > 2`:
   (a) Setze `m = length / 2 - 1`, also gilt

   `length = m + 1 + 1 + m`

   (b) Rekursiv: Bilde Baum l aus

   $$\underbrace{\texttt{keyArray[0]}, \cdots, \texttt{keyArray[m-1]}}_{\texttt{m}}$$

   (c) Rekursiv: Bilde Baum r aus

   $$\underbrace{\texttt{keyArray[m+2]}, \cdots, \texttt{keyArray[m + m + 1]}}_{\texttt{m}}$$

   (d) Erzeuge 2–Knoten n aus

   l, r, `keyArray`$[m]$, `valArray`$[m]$

   (e) Füge `keyArray`$[m+1]$, `valArray`$[m+1]$ in r ein

   `res = insert(n, keyArray[m+1], valArray[m+1])`

   (f) Gebe `res` zurück

Wichtig: Bei den beiden rekursiven Aufrufen muß die Zahl der übergebenen Schlüssel die selbe sein!

gdb Command Line Interface

Aufruf

1. `gcc -std=c99 -Wall -lm -g -o test-2-3-4 test-2-3-4.c`
   Ohne **-g** wird beim Kompilieren keine Debug–Information erzeugt!

2. `gdb test-2-3-4`

gdb–Kommandos

1. `break` *fctName*
   Anhalten bei Funktion *fctName*

2. `break` *lineNumber*
   Anhalten in Zeile *lineNumber*

3. `info breakpoints`
   zeigt Halte–Punkte und Beobachtungs–Punkte an

4. `delete` *number*
   löscht Halte–Punkt / Beobachtungs–Punkt

5. `display` *expr*
   wertet jedesmal *expr* aus, druckt Ergebnis
   Beispiel: `display printTableDot(t, 42);`

6. `help`:
   Liste der Topics, zu denen Hilfe existiert

`gdb` Kommandos

1. `run`: startet das Programm
2. `continue`:
   rechnet bis zum nächsten Haltepunkt
3. `step`: führt eine Programm–Zeile aus
4. `next`: wie oben, springt über Prozedur–Aufruf
5. `until` *line*
   Programm läuft bis zur spezifizierten Zeile
6. `finish`: Programm läuft bis Ende aktueller Funktion
7. `print` *expr*
   gibt Ausdruck aus
8. `print` *fnct(arg1,...,argn)*
   wertet Benutzer–Funktion aus
   Beispiel: `p printTableDot(t, 42);`
9. `display` *expr*
   zeigt jedesmal `expr` an
   Beispiel: `display i`
10. `info display`
    listet Ausdrücke, die automatisch angezeigt werden
11. `delete display`: löschen

Benutzung von `gdb` unter *XEmacs*

1. Aufruf Compiler: `Meta-x compile`
2. Aufruf Debugger: `Meta-x gdb`
3. Aufruf Shell: `Meta-x shell`

Tastatur–Belegung

command	key sequence
`step`	Meta-s
`next`	Ctrl-c Ctrl-n
`continue`	Ctrl-c Meta-n
`finish`	Ctrl-c Ctrl-f
`break` *line*	Ctrl-x space
nächstes Kommando	Meta-p
letztes Kommando	Meta-n
`up`	Ctrl-c <
`down`	Ctrl-c >

Befehle sind ebenfalls über GUI ausführbar