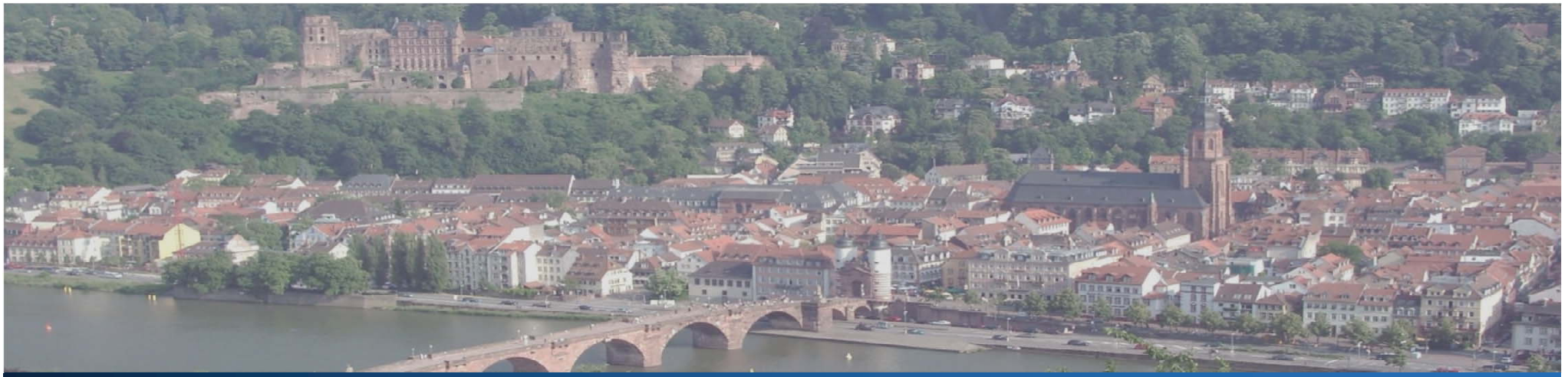
Einführung in die Praktische Informatik

Prof. Björn Ommer **HCI, IWR**

Computer Vision Group



Effiziente Algorithmen und Datenstrukturen

- Heap
- Sortierverfahren mit quadratischer Komplexität
- Sortierverfahren optimaler Ordnung
- Suchen

Effiziente Algorithmen und Datenstrukturen

Beobachtung: Einige der bisher vorgestellten Algorithmen hatten einen sehr hohen Aufwand (z. B. $O(n^2)$ bei Bubblesort, $O(n)$ bei Einfügen/Löschen aus der Priority-Queue). In vielen Fällen ist die STL-Implementation viel schneller.

Frage: Wie erreicht man diese Effizienz?

Ziel: In diesem Kapitel lernen wir Algorithmen und Datenstrukturen kennen, mit denen man hohe (in vielen Fällen sogar optimale) Effizienz erreichen kann.

Heap

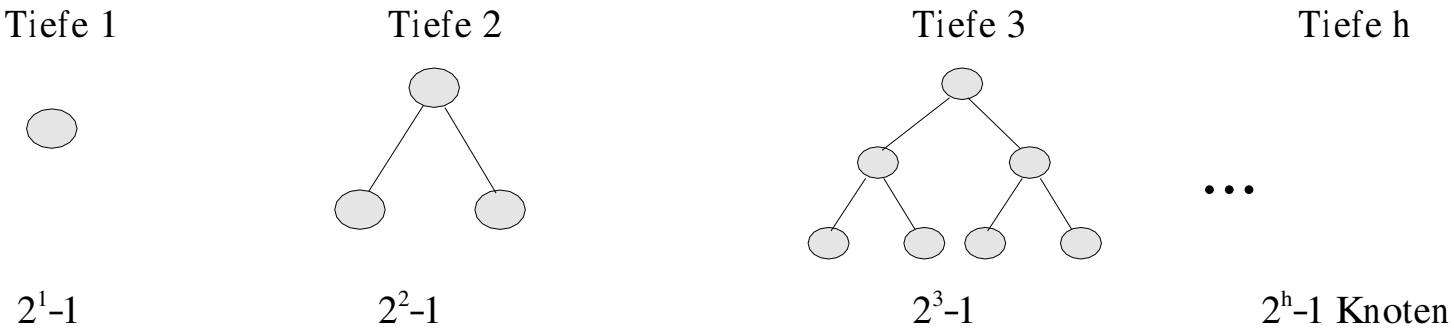
Die Datenstruktur Heap erlaubt es, Einfügen und Löschen in einer Prioritätswarteschlange mit $O(\log n)$ Operationen zu realisieren. Sie ist auch Grundlage eines schnellen Sortierverfahrens (Heapsort).

Definition: Ein Heap ist

- ein fast vollständiger binärer Baum
- Jedem Knoten ist ein Schlüssel zugeordnet. Auf der Menge der Schlüssel ist eine totale Ordnung (z. B. durch einen Operator $\leq$) definiert.
  Totale Ordnung: reflexiv ($a \leq a$), transitiv ($a \leq b, b \leq c \Rightarrow a \leq c$), total ($a \leq b \vee b \leq a$).
- Der Baum ist partiell geordnet, d. h. der Schlüssel jedes Knotens ist *nicht kleiner* als die Schlüssel in seinen Kindern (Heap-Eigenschaft).

Bezeichnung: Ein vollständiger binärer Baum ist ein

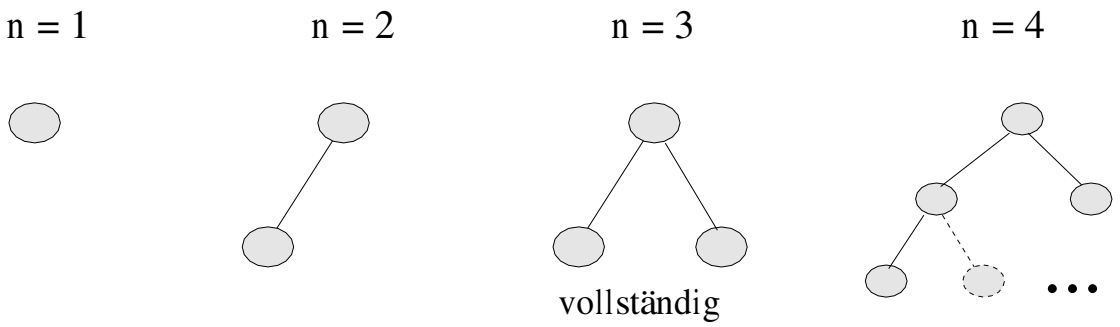
- binärer Baum der Tiefe h mit maximaler Knotenzahl,
- bei dem sich alle Blätter auf der gleichen Stufe befinden.



Bezeichnung: Ein fast vollständiger binärer Baum ist ein binärer Baum mit folgenden Eigenschaften:

- alle Blätter sind auf den beiden höchsten Stufen
- maximal ein innerer Knoten hat nur ein Kind
- Blätter werden von links nach rechts aufgefüllt.

Ein solcher Baum mit n Knoten hat eine eindeutige Struktur:

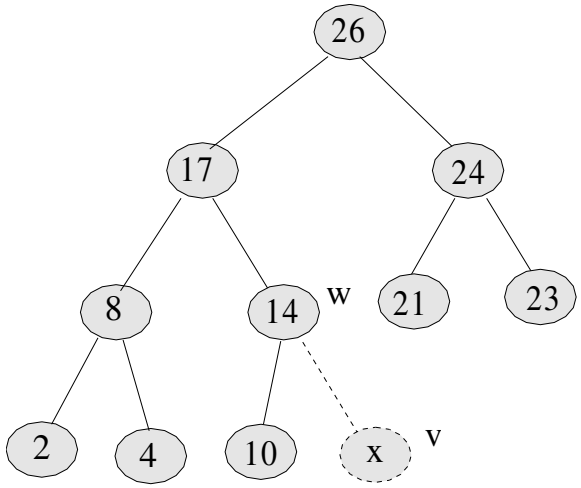


Einfügen

Problem: Gegeben ist ein Heap mit n Elementen und neues Element x. Konstruiere daraus einen um x erweiterten Heap mit $n+1$ Elementen.

Beobachtung: Die Struktur des Baumes mit $n+1$ Elementen liegt fest. Wenn man daher x an der neuen Position v einfügt, so kann die Heapeigenschaft nur im Knoten $w = \mathrm{Elter}(v)$ verletzt sein.

Beispiel:



Algorithmus: Wiederherstellen der Heapeigenschaft in maximal $\lceil \log_2 n \rceil$ Vertauschungen:

Falls $Inhalt(w) < Inhalt(v)$ dann
    tausche Inhalt
    Falls w nicht die Wurzel ist:
        setze $w = Elter(w); v = Elter(v);$
    sonst $\rightarrow$ fertig
sonst $\rightarrow$ fertig

Reheap

Die im folgenden beschriebene Reheap-Operation wird beim Entfernen der Wurzel gebraucht.

Problem: Gegeben ist ein fast vollständiger Baum mit Schlüsseln, so dass die Heapeigenschaft in allen Knoten exklusive der Wurzel gilt. Ziel ist die Transformation in einen echten Heap.

Algorithmus:

1. Tausche Schlüssel in der Wurzel mit dem größeren der beiden Kinder.

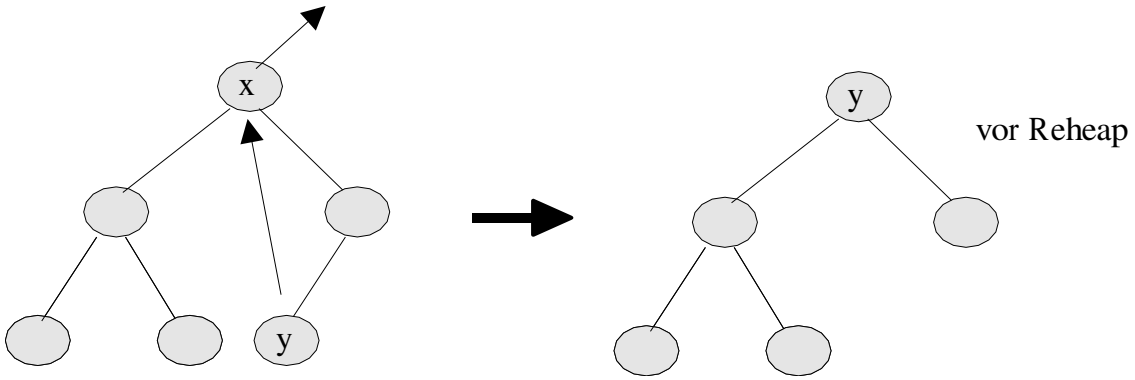
2. Wenn die Heap-Eigenschaft für dieses Kind nicht erfüllt ist, so wende den Algorithmus rekursiv an, bis ein Blatt erreicht wird.

Entfernen des Wurzelelements

Algorithmus:

- Ersetze den Wert in der Wurzel (Rückgabewert) durch das letzte Element des fast vollständigen binären Baumes.
- Verkleinere den Heap und rufe Reheap auf

Beispiel:



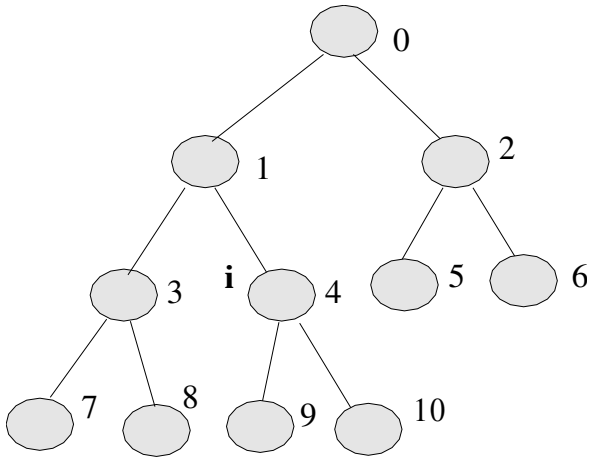
Komplexität

Ein fast vollständiger Baum mit n Knoten hat genau $\lceil \mathrm{ld}(n+1) \rceil$ Ebenen. Somit benötigt das Einfügen maximal $\lceil \mathrm{ld}(n+1) \rceil - 1$ Vergleiche und die Operation reheap maximal $2(\lceil \mathrm{ld}(n+1) \rceil - 1)$ Vergleiche.

Datenstruktur

Beobachtung: Die Knoten eines fast vollständigen binären Baumes können sehr effizient in einem Feld gespeichert werden:

Numeriert man die Knoten wie folgt:



Dann gilt für Knoten i:

Linkes Kind:	$2i + 1$
Rechtes Kind:	$2(i + 1)$
Elter:	$\lfloor \frac{i-1}{2} \rfloor$

Implementation

Programm: **Definition und Implementation (Heap.hh)**

```
template <class T>
class Heap
{
public:
  bool empty();
  void push( T x );
  void pop();
  T top();
private:
  std::vector<T> data;
  void reheap( int i );
};
```

```
template <class T>
void Heap<T>::push( T x )
{
  int i = data.size();
  data.push_back( x );
  while ( i > 0 && data[i] > data[(i-1)/2] )
  {
    std::swap( data[i], data[(i-1)/2] );
    i = (i - 1) / 2;
  }
}

template <class T>
void Heap<T>::reheap( int i )
{
  int n = data.size();
  while ( 2*i+1 < n )
```

```
    {
      int l = 2 * i + 1;
      int r = l + 1;
      int k = ( (r < n) && (data[r] > data[l]) ) ? r : l;
      if ( data[k] <= data[i] ) break;
      std::swap( data[k], data[i] );
      i = k;
    }
}

template <class T>
void Heap<T>::pop()
{
    std::swap( data.front(), data.back() );
    data.pop_back();
    reheap( 0 );
}
```

```
template <class T>
T Heap<T>::top()
{
  return data[0];
}

template <class T>
inline bool Heap<T>::empty()
{
  return data.size() == 0;
}
```

Programm: Anwendung (UseHeap.cc)

```
#include <vector>
#include <iostream>
```

```
#include "Heap.hh"
#include "Zufall.cc"

int main()
{
  Zufall z( 87123 );
  Heap<int> h;

  for ( int i=0; i<10; i=i+1 )
  {
    int k = z.ziehe_zahl();
    std::cout << k << std::endl;
    h.push( k );
  }
  std::cout << std::endl;
  while ( !h.empty() )
```

```
    {
      std::cout << h.top() << std::endl;
      h.pop();
    }
  }
```

Beobachtung: Mit Hilfe der Heap-Struktur lässt sich sehr einfach ein recht guter Sortieralgorithmus erzeugen. Dazu ordnet man Elemente einfach in einen Heap ein und extrahiert sie wieder. Dies wird später noch genauer beschrieben.

Sortierverfahren mit quadratischer Komplexität

Gegeben: Eine „Liste“ von Datensätzen $(D_0, \ldots, D_{n-1})$. Zu jedem Datensatz D_i gehört ein Schlüssel $k_i = k(D_i)$. Auf der Menge der Schlüssel sei eine $totale$ Ordnung durch einen Operator $\leq$ definiert.

Definition: Eine Permutation von $I = \{0, \ldots, n-1\}$ ist eine bijektive Abbildung $\pi : I \to I$.

Gesucht: Eine Permutation $\pi : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$, so dass gilt

$$k_{\pi(0)} \leq \ldots \leq k_{\pi(n-1)}$$

Bemerkung: In der Praxis hat man:

- Die Datensätze sind normalerweise in einer Liste oder einem Feld gespeichert. Wir betrachten im folgenden den Fall des Felds.
- Oft braucht man die Permutation π nicht weiter, und es reicht aus, als Ergebnis einer Sortierfunktion eine sortierte Liste/Feld zu erhalten.
- Die Relation $\leq$ wird durch einen Vergleichsoperator definiert.
- Für große Datensätze sortiert man lieber ein Feld von Zeigern.
- Internes Sortieren: Alle Datensätze sind im Hauptspeicher.
- Externes Sortieren: Sortieren von Datensätzen, die auf Platten, Bändern, etc. gespeichert sind.

Die folgenden Implementierungen können z. B. mit einem `std::vector` aufgerufen werden.

Selectionsort (Sortieren durch Auswahl)

Idee:

- Gegeben sei ein Feld $a = (a_0, \ldots, a_{n-1})$ der Länge n.
- Suche das Minimum im Feld und tausche mit dem erstem Element.
- Danach steht die kleinste der Zahlen ganz links, und es bleibt noch ein Feld der Länge $n - 1$ zu sortieren.

Programm: **Selectionsort (Selectionsort.cc)**

```
template <class C>
void selectionsort( C& a )
{
  for ( int i=0; i<a.size()-1; i=i+1 )
```

```
    {    // i Elemente sind sortiert
      int min = i;
      for ( int j=i+1; j<a.size(); j=j+1 )
        if ( a[j] < a[min] ) min = j;
      std::swap( a[i], a[min] );
    }
  }
```

Bemerkung:

- Komplexität: in führender Ordnung $\frac{n^2}{2}$ Vergleiche, n Vertauschungen $\rightarrow O(n^2)$.
- Die Anzahl von Datenbewegungen ist optimal, das Verfahren ist also zu empfehlen, wenn möglichst wenige Datensätze bewegt werden sollen.

Bubblesort

Idee: (Siehe den Abschnitt über Effizienz generischer Programmierung.)

- Gegeben sei ein Feld $a = (a_0, \ldots, a_{n-1})$ der Länge n.
- Durchlaufe die Indizes $i = 0, 1, \ldots, n-2$ und vergleiche jeweils a_i und a_{i+1}. Ist $a_i > a_{i+1}$ so vertausche die beiden.
- Nach einem solchen Durchlauf steht die größte der Zahlen ganz rechts, und es bleibt noch ein Feld der Länge $n-1$ zu sortieren.

Programm: **Bubblesort mit STL (Bubblesort.cc)**

```
template <class C>
void bubblesort( C& a )
{
  for ( int i=a.size()-1; i>=0; i-- )
    for ( int j=0; j<i; j=j+1 )
      if ( a[j+1] < a[j] )
        std::swap( a[j+1], a[j] );
}
```

Bemerkung:

- Komplexität: in führender Ordnung $\frac{n^2}{2}$ Vergleiche, $\frac{n^2}{2}$ Vertauschungen

Insertionsort (Sortieren durch Einfügen)

Beschreibung: Der bereits sortierte Bereich liegt links im Feld und das nächste Element wird jeweils soweit nach links bewegt, bis es an der richtigen Stelle sitzt.

Programm: **Insertionsort mit STL (Insertionsort.cc)**

```
template <class C>
void insertionsort( C& a )
{
  for ( int i=1; i<a.size(); i=i+1 )
  {
    // i Elemente sind sortiert
    int j = i;
    while ( j > 0 && a[j-1] > a[j] )
    {
      std::swap( a[j], a[j-1] );
      j = j - 1;
```

```
        }
    }
}
```

Bemerkung:

- Komplexität: $\frac{n^2}{2}$ Vergleiche, $\frac{n^2}{2}$ Vertauschungen $\rightarrow O(n^2)$.

- Ist das Feld bereits sortiert, endet der Algorithmus nach $O(n)$ Vergleichen. Sind in ein bereits sortiertes Feld mit n Elementen m weitere Elemente einzufügen, so gelingt dies mit Insertionsort in $O(nm)$ Operationen. Dies ist optimal für sehr kleines m ($m \ll \log n$).

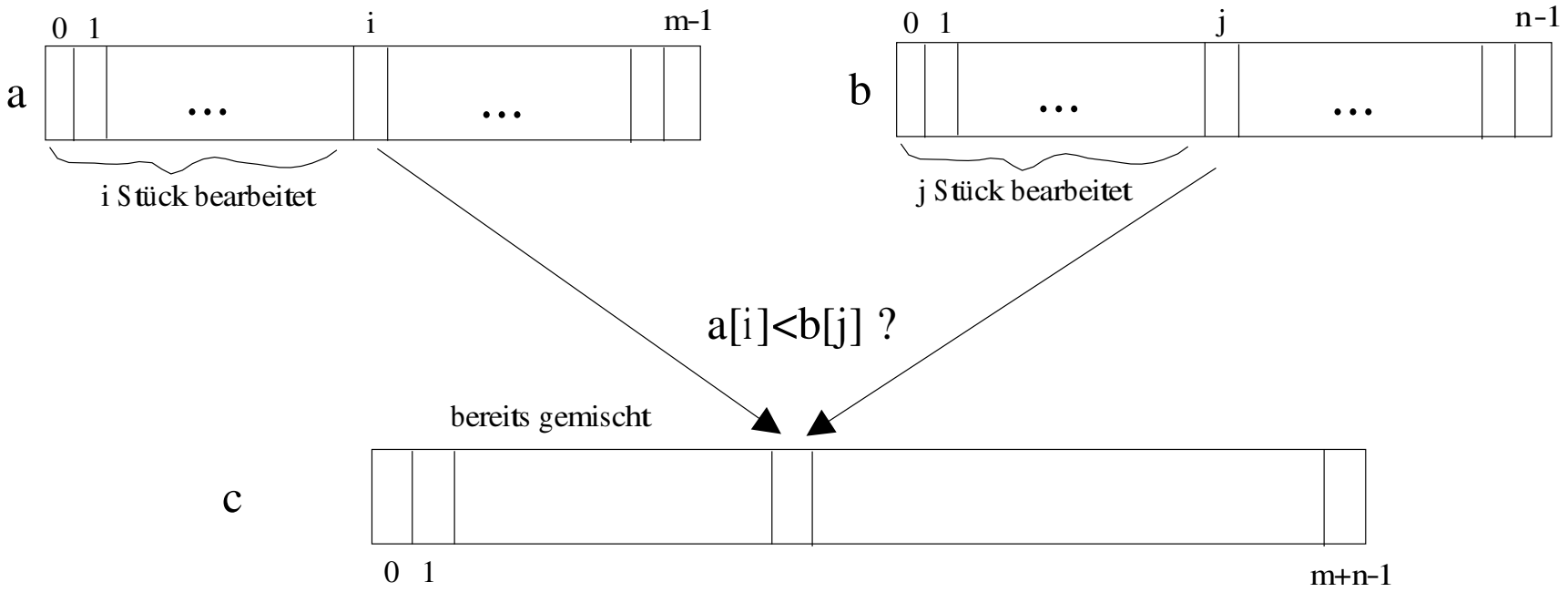
Sortierverfahren optimaler Ordnung

In diesem Abschnitt betrachten wir Sortierverfahren, die den Aufwand $O(n \log n)$ haben. Man kann zeigen, dass dieser Aufwand für allgemeine Felder von Datensätzen optimal ist.

Erinnerung: $\log x = \log_e x$, $\mathrm{ld}\, x = \log_2 x$. Wegen $\mathrm{ld}\, x = \frac{\log x}{\log 2} = 1.44\ldots \cdot \log x$ gilt $O(\log n) = O(\mathrm{ld}\, n)$.

Mergesort (Sortieren durch Mischen)

Beobachtung: Zwei bereits sortierte Felder der Länge m bzw. n können sehr leicht (mit Aufwand $O(m+n)$) zu einem sortierten Feld der Länge $m+n$ „vermischt“ werden:



Dies führt zu folgendem Algorithmus vom Typ „Divide and Conquer“:

Algorithmus:

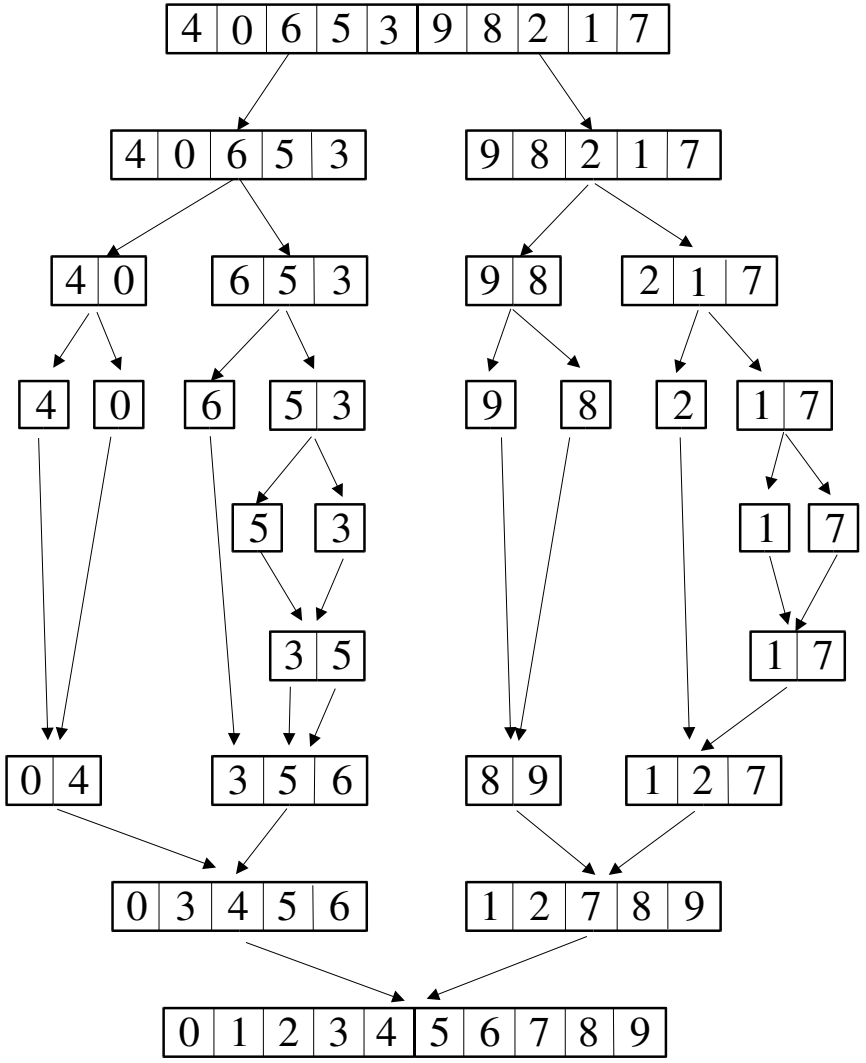
- Gegeben ein Feld a der Länge n.
- Ist $n = 1$, so ist nichts zu tun, sonst
- Zerlege a in zwei Felder $a1$ mit Länge $n1 = n/2$ (ganzzahlige Division) und $a2$ mit Länge $n2 = n - n1$,
- sortiere $a1$ und $a2$ (Rekursion) und
- mische $a1$ und $a2$.

Programm: Mergesort mit STL (Mergesort.cc)

```
template <class C>
void rec_merge_sort( C& a, int o, int n )
{   // sortiere Eintraege [ o, o+n-1 ]
  if ( n == 1 ) return;

  // teile und sortiere rekursiv
  int n1 = n / 2;
  int n2 = n - n1;
  rec_merge_sort( a, o, n1 );
  rec_merge_sort( a, o + n1, n2 );

  // zusammenfuegen
  C b( n ); // Hilfsfeld
  int i1 = o, i2 = o + n1;
  for ( int k=0; k<n; k=k+1 )
    if ( ( i2 >= o+n ) || ( i1 < o+n1 && a[i1] <= a[i2] ) )
      b[k] = a[i1++];
```

```
      else
        b[k] = a[i2++];

    // umkopieren
    for ( int k=0; k<n; k=k+1 ) a[o + k] = b[k];
}

template <class C>
void mergesort( C& a )
{
    rec_merge_sort( a, 0, a.size() );
}
```

Beispiel:


4 0 6 5 3 9 8 2 1 7
4 0 6 5 3
9 8 2 1 7
4 0
6 5 3
9 8
2 1 7
4
0
6
5 3
9
8
2
1 7
5
3
1
7
3 5
1 7
0 4
3 5 6
8 9
1 2 7
0 3 4 5 6
1 2 7 8 9
0 1 2 3 4 5 6 7 8 9

Bemerkung:

- Mergesort benötigt zusätzlichen Speicher von der Größe des zu sortierenden Felds.

- Die Zahl der Vergleiche ist aber (in führender Ordnung) $n \operatorname{ld} n$.
  Beweis für $n = 2^k$: Für die Zahl der Vergleiche $V(n)$ gilt (Induktion)

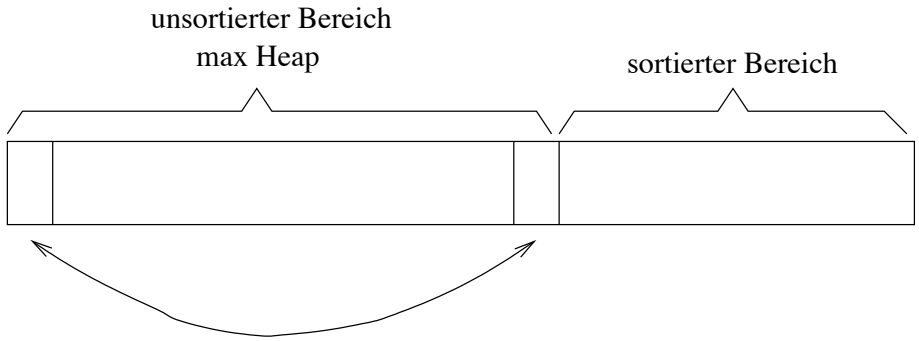
$$\begin{aligned} V(1) &= 0 = 1 \operatorname{ld} 1 \\ V(n) &= 2V(\frac{n}{2}) + n - 1 \leq 2(k-1)\frac{n}{2} + n \leq nk \end{aligned}$$

  ($2V(n/2)$: Sortieren beider Hälften; $n-1$: Merge). Man kann zeigen, dass dies optimal ist.

- Mergesort ist stabil, d. h. Datensätze mit gleichen Schlüsseln verbleiben in derselben Reihenfolge wie zuvor

Heapsort

Idee: Transformiere das Feld in einen Heap, und dann wieder in ein sortiertes Feld. Wegen der kompakten Speicherweise für den Heap kann dies ohne zusätzlichen Speicherbedarf geschehen, indem man das Feld wie folgt unterteilt:



Bemerkung:

- Die Transformation des Felds in einen Heap kann auf zwei Weisen geschehen:

  1. Der Heap wird von vorne durch `push` aufgebaut.
  2. Der Heap wird von hinten durch `reheap` aufgebaut.

  Da wir `reheap` sowieso für die pop-Operation brauchen, wählen wir die zweite Variante.

- Heapsort hat in führender Ordnung die Komplexität von $2n \operatorname{ld} n$ Vergleichen. Der zusätzliche Speicheraufwand ist unabhängig von n (in-situ-Verfahren)!

- Im Gegensatz zu Mergesort ist Heapsort nicht stabil.

Programm: **Heapsort mit STL (Heapsort.cc)**

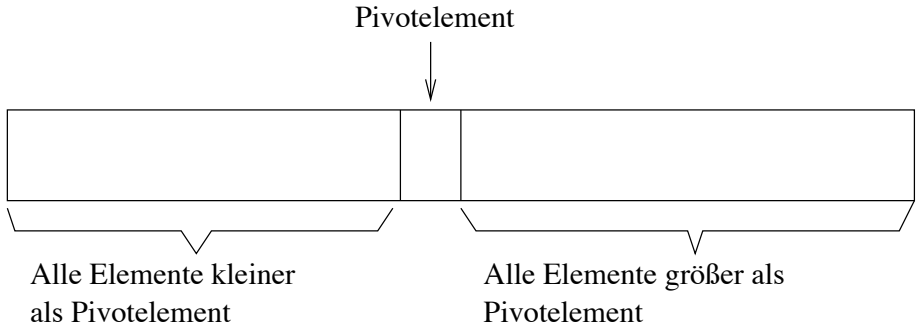
```
template <class C>
inline void reheap( C& a, int n, int i )
{
  while ( 2*i+1 < n )
  {
    int l = 2 * i + 1;
    int r = l + 1;
    int k = ( ( r < n ) && ( a[r] > a[l] ) ) ? r : l;
    if ( a[k] <= a[i] ) break;
    std::swap( a[k], a[i] );
    i = k;
  }
}

template <class C>
```

```
void heapsort( C& a )
{
  // build the heap by reheaping from the rear
  for ( int i=a.size()-1; i>=0; i-- )
    reheap( a, a.size(), i );
  // build the sorted list by popping the heap
  for ( int i=a.size()-1; i>=0; i-- )
  {
    std::swap( a[0], a[i] );
    reheap( a, i, 0 );
  }
}
```

Quicksort

Beobachtung: Das Hauptproblem bei Mergesort war das speicheraufwendige Mischen. Man könnte es vermeiden, wenn man das Feld so in zwei (möglichst gleichgroße) Teile unterteilen könnte, das alle Elemente des linken Teilfelds kleiner oder gleich allen Elementen des rechten Teilfeldes sind.

Idee: Wähle „zufällig“ ein beliebiges Element $q \in \{0, \ldots, n-1\}$ aus, setze Pivot$= a[q]$ und zerlege das Feld so, dass das Eingabefeld folgende Gestalt hat:



(Elemente gleich $a[q]$ dürfen in linkes oder rechtes Teilfeld eingefügt werden).

Programm: Quicksort mit STL (Quicksort.cc)

```
template <class C>
int qs_partition( C& a, int l, int r, int q )
{
  std::swap( a[q], a[r] );
  q = r;        // Pivot ist jetzt ganz rechts
  int i = l - 1, j = r;

  while ( i < j )
  {
    i = i + 1; while ( i < j && a[i] <= a[q] ) i = i + 1;
    j = j - 1; while ( i < j && a[j] >= a[q] ) j = j - 1;
    if ( i < j )
      std::swap( a[i], a[j] );
    else
      std::swap( a[i], a[q] );
  }
  return i; // endgueltige Position des Pivot
```

```
}

template <class C>
void qs_rec( C& a, int l, int r )
{
  if ( l < r )
  {
    int i = qs_partition( a, l, r, r );
    qs_rec( a, l, i-1 );
    qs_rec( a, i+1, r );
  }
}

template <class C>
void quicksort( C& a ) {
  qs_rec( a, 0, a.size()-1 );
}
```

Bemerkung:

- Man kann im allgemeinen $nicht$ garantieren, dass beide Hälften gleich groß sind.
- Im schlimmsten Fall wird das Pivotelement immer so gewählt, dass man ein einelementiges Teilfeld und den Rest als Zerlegung erhält. Dann hat Quicksort den Aufwand $O(n^2)$.
- Im besten Fall ist die Zahl der Vergleiche so gut wie Mergesort.
- Im „Mittel“ erhält man in führender Ordnung $1.386\, n \operatorname{ld} n$ Vergleiche.
- Auch Quicksort ist nicht stabil.
- Praktisch wählt man oft drei Elemente zufällig aus und wählt das mittlere. Damit wird Quicksort ein randomisierter Algorithmus.

- Die Wahrscheinlichkeit den $O(n^2)$ Fall zu erhalten ist bei zufälliger Pivotwahl $1/n!$.

Anwendung

Mit folgendem Programm kann man die verschiedenen Sortierverfahren ausprobieren:

Programm: **UseSort.cc**

```
#include <iostream>
#include <vector>
#include "Bubblesort.cc"
#include "Selectionsort.cc"
#include "Insertionsort.cc"
#include "Mergesort.cc"
#include "Heapsort.cc"
#include "Quicksort.cc"
#include "Zufall.cc"
#include "timestamp.cc"

void initialize( std::vector<int>& a )
{
```

```
  Zufall z( 8267 );
  for ( int i=0; i<a.size(); ++i )
    a[i] = z.ziehe_zahl();
}

int main()
{
  int n = 100000;
  std::vector<int> a( n );

  initialize( a );
  time_stamp();
  quicksort( a );
  std::cout << "n=" << n << " quicksort t="
            << time_stamp() << std::endl;

  initialize( a );
  time_stamp();
  mergesort( a );
```

```
std::cout << "n=" << n << " mergesort t="
          << time_stamp() << std::endl;

initialize( a );
time_stamp();
heapsort( a );
std::cout << "n=" << n << " heapsort t="
          << time_stamp() << std::endl;

initialize( a );
time_stamp();
bubblesort( a );
std::cout << "n=" << n << " bubblesort t="
          << time_stamp() << std::endl;

initialize( a );
time_stamp();
insertionsort( a );
std::cout << "n=" << n << " insertionsort t="
```

```
            << time_stamp() << std::endl;

  initialize( a );
  time_stamp();
  selectionsort( a );
  std::cout << "n=" << n << " selectionsort t="
            << time_stamp() << std::endl;
}
```

Suchen

Binäre Suche in einem Feld

Idee: In einem sortierten Feld kann man Elemente durch sukzessives Halbieren schnell finden.

Bemerkung:

- Aufwand: in jedem Schritt wird die Länge des Suchbereichs halbiert. Der Aufwand beträgt daher $\lceil \mathrm{ld}(n) \rceil$ Vergleiche, denn dann kann man nicht weiter halbieren. Anschließend braucht man noch einen Vergleich, um zu prüfen, ob das Element das Gesuchte ist. $\Rightarrow \lceil \mathrm{ld}(n) \rceil + 1$ Vergleiche.

- Die binäre Suche ermöglicht also auch die Aussage, dass ein Element nicht enthalten ist!

Programm: Nicht-rekursive Formulierung (Binsearch.cc)

```
template <class C>
int binsearch( C& a, typename C::value_type x )
{   // returns either index (if found) or -1
  int l = 0;
  int r = a.size();
  while ( 1 )
  {
    int m = ( l + r ) / 2;
    if ( m == l )
      return ( a[m] == x ) ? m : -1;
    if ( x < a[m] )
      r = m;
    else
      l = m;
  }
```

$\}$

Bemerkung:

- Die binäre Suche beschleunigt nur das Finden.
- Einfügen und Löschen haben weiterhin Aufwand $O(n)$, da Feldelemente verschoben werden müssen.
- Binäre Suche geht auch nicht mit einer Liste (kein wahlfreier Zugriff).

Binäre Suchbäume

Beobachtung: Die binäre Suche im Feld kann als Suche in einem binären Suchbaum interpretiert werden (der aber in einem Feld abgespeichert wurde).

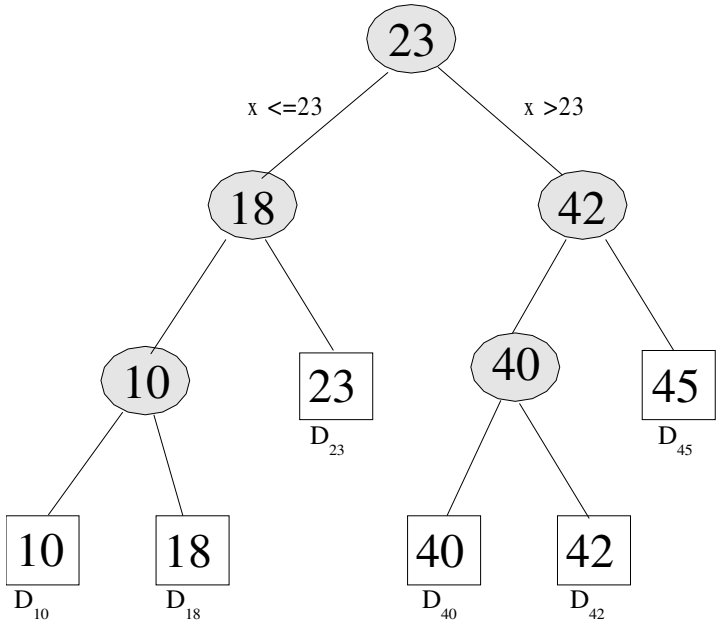
Idee: Die Verwendung einer echten Baum-Datenstruktur kann schnelles Einfügen und Entfernen von Knoten ermöglichen.

Definition: Ein binärer Suchbaum ist ein binärer Baum, in dessen Knoten Schlüssel abgespeichert sind und für den die Suchbaumeigenschaft gilt:

> Der Schlüssel in jedem Knoten ist größer gleich allen Schlüsseln im linken Teilbaum und kleiner als alle Schlüssel im rechten Teilbaum (Variante 1).

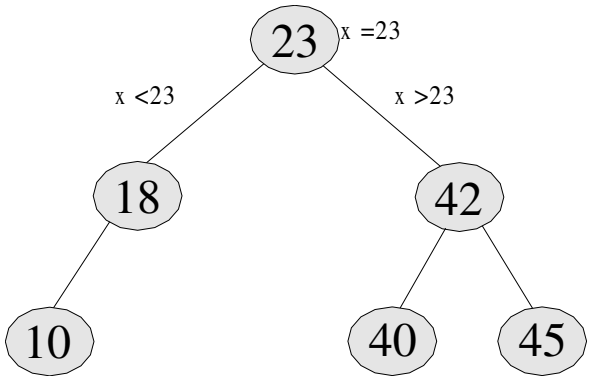
Bemerkung: Der Aufwand einer Suche entspricht der Höhe des Baumes.

Variante 1



- *Innere* Knoten enthalten nur Schlüssel
- (Verweise auf) Datensätze sind in den *Blättern* des Baumes gespeichert.

Variante 2 Man speichert Schlüssel und Datensatz genau einmal in einem Knoten:



Bemerkung: Innere Knoten unterscheiden sich von Blättern dann nur noch durch das Vorhandensein von Kindern.

Die Suchalgorithmen unterscheiden sich leicht:

Variante 1: Am Blatt prüfe Schlüssel; innerer Knoten: bei $\leq$ gehe links, sonst rechts. Innere Knoten haben immer zwei Kinder!

Variante 2: Prüfe Schlüssel; Falls Kind links existiert und Schlüssel $<$ gehe links; falls Kind rechts existiert und Schlüssel $>$ gehe rechts; sonst nicht gefunden.

Einfügen

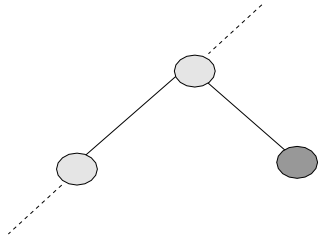
Das Einfügen (in Variante 2) geschieht, indem man durch den Baum bis zu einem Knoten läuft, wo man den Datensatz/Schlüssel einfügen kann.

Beispiel: Einfügen von 20 im Baum auf der letzten Folie. Diese wird rechts unter der 18 eingefügt. Wo wird die 41 eingefügt?

Das Löschen ist etwas komplizierter, weil verschiedene Situationen unterschiedlich behandelt werden müssen.

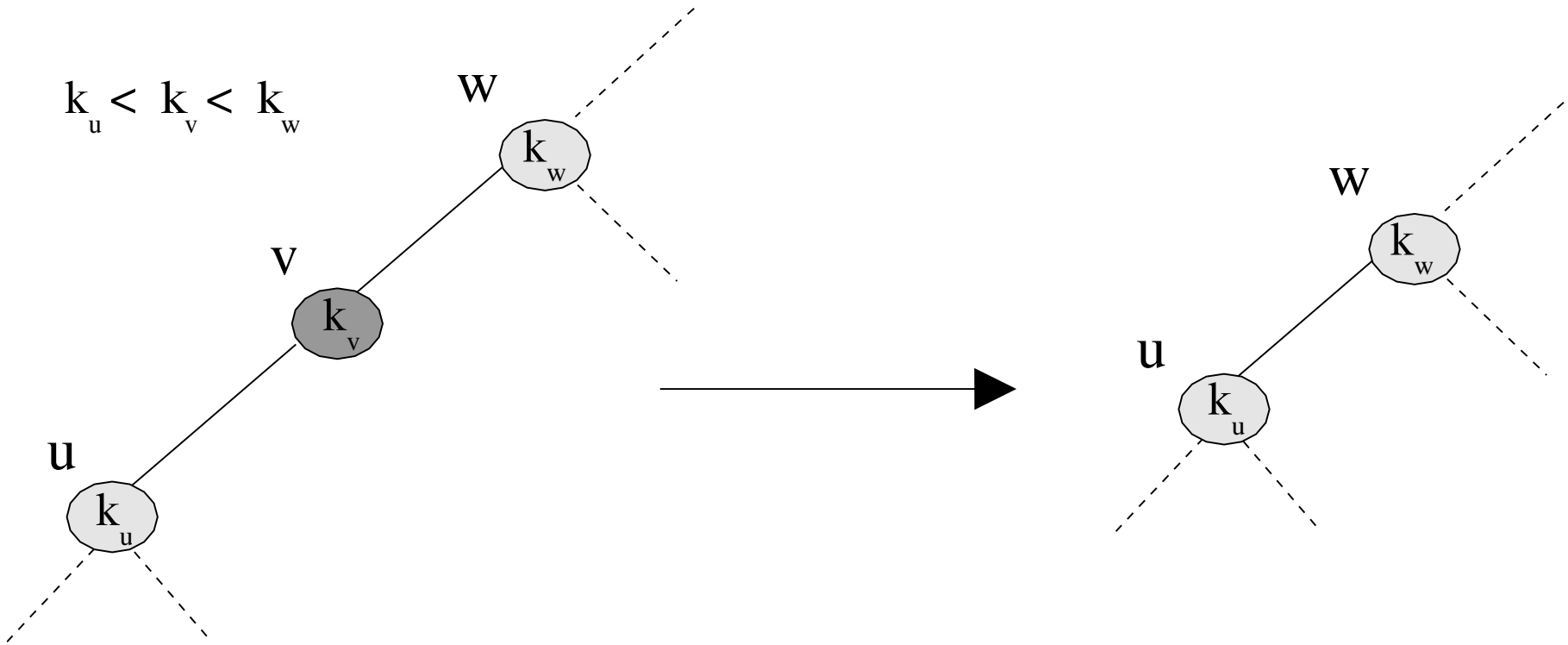
Löschen eines Blattes (Variante 2)

$\rightarrow$ einfach wegnehmen



Löschen eines Knotens mit einem Kind (Variante 2)

Der Knoten kann einfach herausgenommen werden.



Löschen eines Knotens mit zwei Kindern (Variante 2)

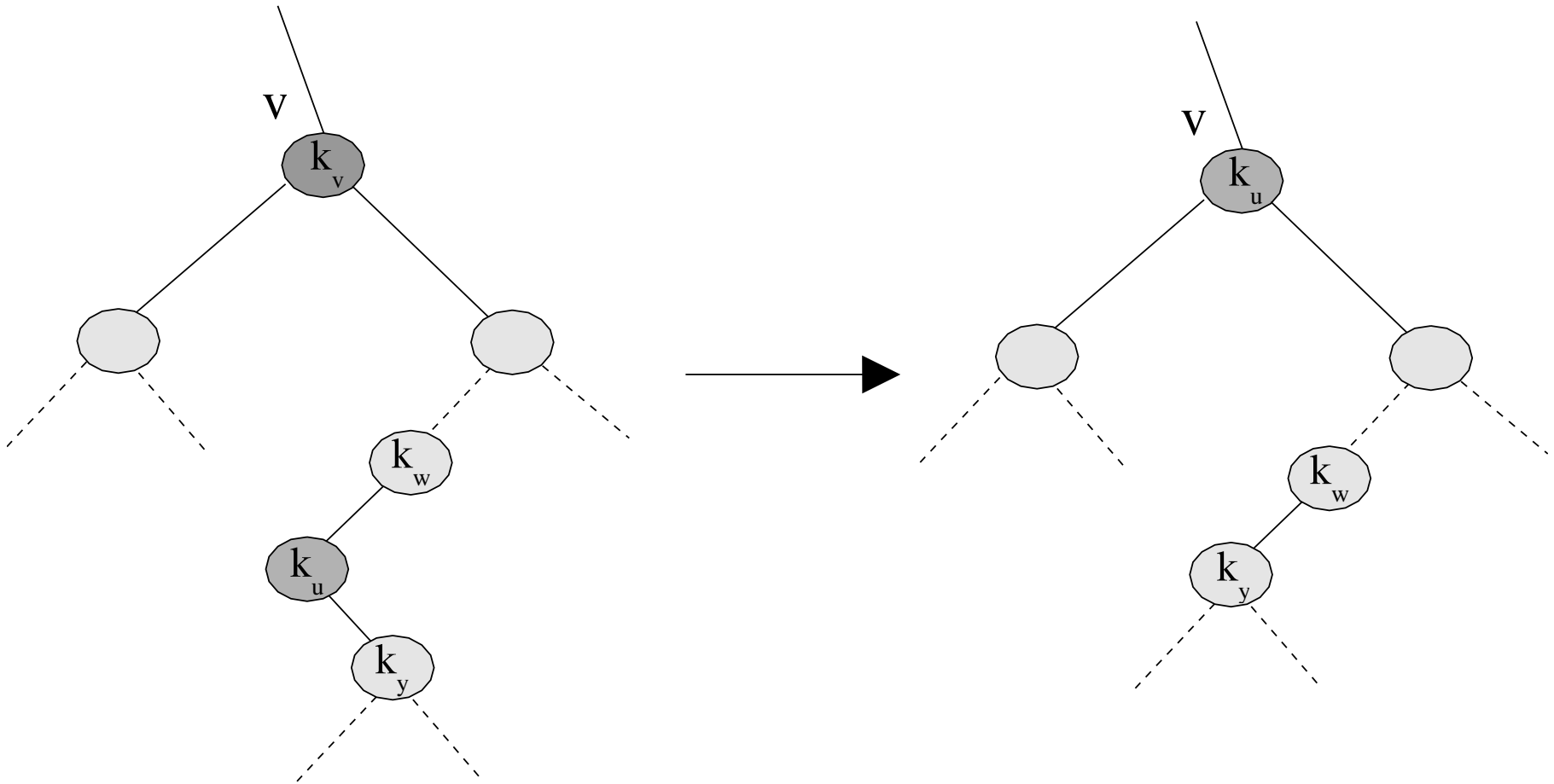
Schlüssel k_v soll gelöscht werden. Betrachte k_u: kleinster Schlüssel im rechten Teilbaum von v.

Behauptung: u hat höchstens einen rechten Teilbaum (also keine zwei Kinder)! Das ist klar, denn hätte u einen linken Teilbaum so wären die Schlüssel dort kleiner als k_u und somit wäre k_u nicht der minimale Schlüssel rechts von v.

Vorgehensweise:

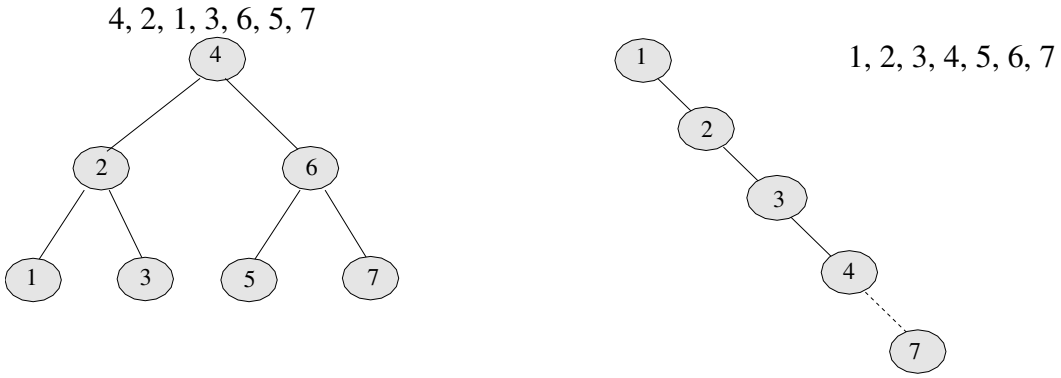
- ersetze k_v durch k_u und lösche Knoten u (dies ist einfach, da der keine zwei Kinder hat).

- Suchbaumeigenschaft in v bleibt erhalten: Alle rechts verbleibenden Knoten haben Schlüssel größer als k_u, da k_u minimal war. Alle links verbleibenden Schlüssel waren ohnehin kleiner.


v
k v
k w
k u
k y
v
k u
k w
k y

Problem: Die Gestalt des Suchbaumes und damit der Aufwand für seine Bearbeitung hängt entscheidend von der Reihenfolge des Einfügens (und eventuell Löschens) der Schlüssel ab!

Beispiel:



Der rechte Binärbaum entspricht im wesentlichen einer Listenstruktur! Der Aufwand zur Suche ist entsprechend $O(n)$.

Ausgeglichene Bäume

Beobachtung: Um optimale Effizienz zu gewährleisten, müssen sowohl Einfüge- als auch Löschoperation im Suchbaum sicherstellen, dass für die Höhe $H(n) = O(\log n)$ gilt. Dies kann auf verschiedene Weisen erreicht werden.

AVL-Bäume

Die AVL-Bäume wurden 1962 von Adelson-Velskii-Landis eingeführt. Es sind Binärbäume, die garantieren, dass sich die Höhen von rechtem und linken Teilbaum ihrer Knoten maximal um 1 unterscheiden (Höhenbalancierung).

(2,3)-Baum

Der (2,3)-Baum wurde 1970 von Hopcroft eingeführt. Er ist ein Spezialfall des später besprochenen (a,b)-Baums. Die Idee ist, mehr Schlüssel/Kinder pro Knoten zuzulassen.

B-Bäume

B-Bäume wurden 1970 von Bayer und McCreight eingeführt. Der B-Baum der

Ordnung m ist gleich dem (a, b)-Baum mit $a = \lceil \frac{m}{2} \rceil$, $b = m$. Hier haben Knoten bis zu m Kinder. Für großes m führt das zu sehr flachen Bäumen. Der B-Baum wird oft zur Suche in externem Speicher verwendet. Dabei ist m die Anzahl der Schlüssel, die in einen Sektor der Platte passen (z. B. Sektorgröße 512 Byte, Schlüssel 4 Byte $\Rightarrow$ m=128).

α-balancierter Baum

α-balancierte Bäume wurden um 1973 von Nievergelt und Reingold eingeführt. Idee: Die Größe $|T(v)|$ des Baums am Knoten v und des rechten (oder linken) Teilbaums $|T_l(v)|$ erfüllen

$$\alpha \leq \frac{|T_l(v)| + 1}{|T(v)| + 1} \leq 1 - \alpha$$

Dies garantiert wieder $H = O(\log n)$.

Rot-Schwarz-Bäume

Rot-Schwarz-Bäume wurden 1978 von Bayer, Guibas, Sedgewick eingeführt. Hier

haben Knoten verschiedene „Farben“. Die Einfüge- und Löschoperationen erhalten dann gewisse Anforderungen an die Farbreihenfolge, welche wieder $H = O(\log n)$ garantieren. Man kann auch eine Äquivalenz zum (2,4)-Baum zeigen.

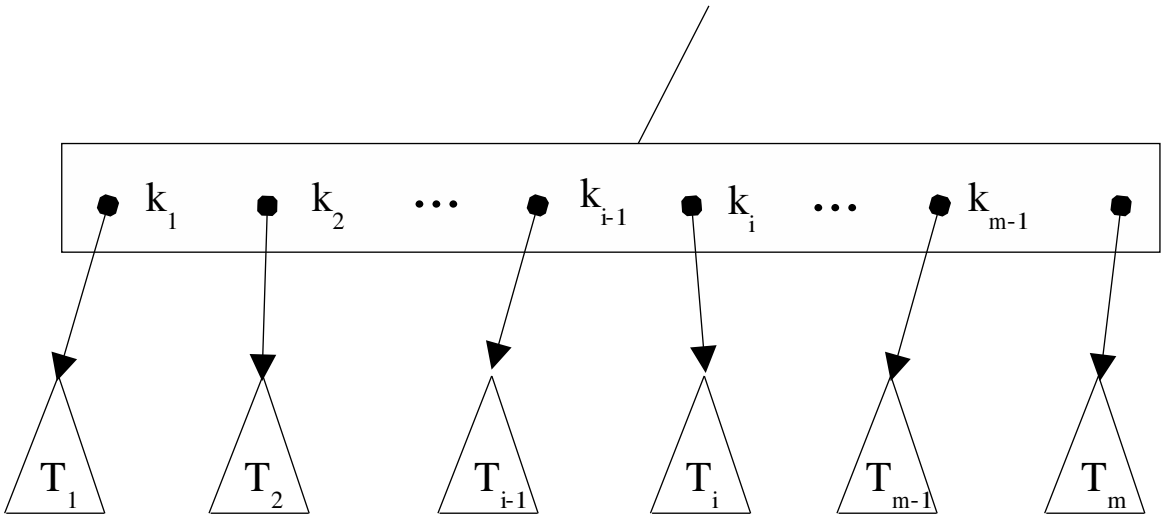
(a,b)-Bäume

(a,b)-Bäume wurden 1982 von Huddleston und Mehlhorn als Verallgemeinerung der B-Bäume und des (2,3)-Baums eingeführt. Alle inneren Knoten haben hier mindestens a und höchstens b Kinder für $a \geq 2$ und $b \geq 2a - 1$. Außerdem befinden sich alle Blätter auf der gleichen Stufe. Dies garantiert dann wieder $H = O(\log n)$.

(a, b)-Bäume (Mehlhorn 1982)

Der (a, b)-Baum stellt eine Erweiterung des binären Baumes auf viele Schlüssel pro Knoten dar:

- Jeder innere Knoten enthält Schlüssel
- Blätter enthalten Datensätze (Variante 1 oben)



Allgemeiner innerer Knoten

- höherer Verzweigungsgrad: Ein Knoten hat bis zu m Kinder und $m-1$ Schlüssel
- alle Schlüssel sind sortiert: $k_1 < k_2 < k_3 < \cdots < k_{m-1}$
- für alle Schlüssel k im Teilbaum T_i, $1 \leq i \leq m$, gilt: $k_{i-1} < k \leq k_i$ (setze $k_0 = -\infty, k_m = \infty$)

Ein Baum aus solchen Knoten heißt (a, b)-Baum, falls gilt:

- alle Blätter sind auf derselben Stufe
- jeder innere Knoten hat maximal b Kinder und für die minimale Zahl der Kinder innerer Knoten gilt
  - Wurzel hat mindestens 2 Kinder
  - andere haben $mindestens\ a \geq 2$ Kinder
- Es gilt: $b \geq 2a - 1$

z. B. $a = 2, b = 3$ minimaler (a, b)-Baum (Hopcroft 1970)

$a = 2, b = 4$ $(2, 4)$-Baum$\Longleftrightarrow$rot-schwarz-Baum

$a = \lceil \frac{m}{2} \rceil, b = m$ B-Baum, zur Suche in externem Speicher

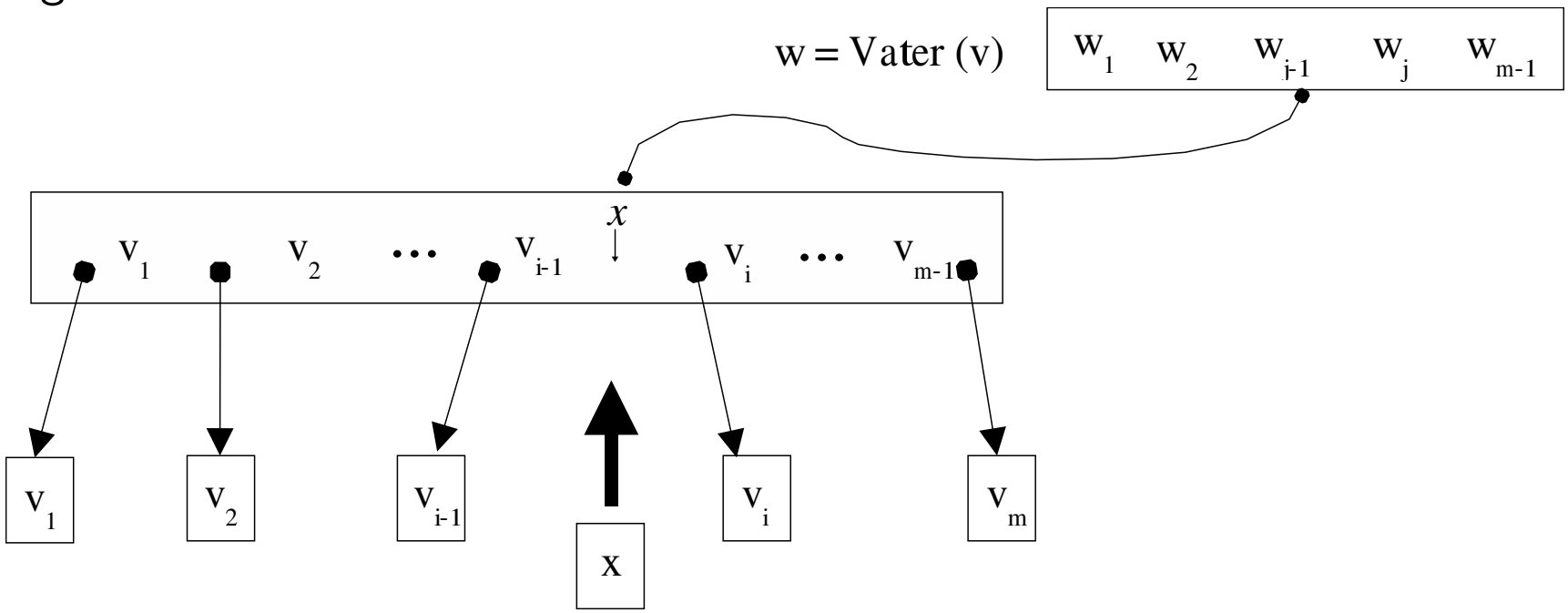
$\Rightarrow m$ Anzahl der Schlüssel, die in einen Sektor der Platte pass

z. B. Sektorgröße 512 Byte, Schlüssel 4 Byte $\Rightarrow$ b=128

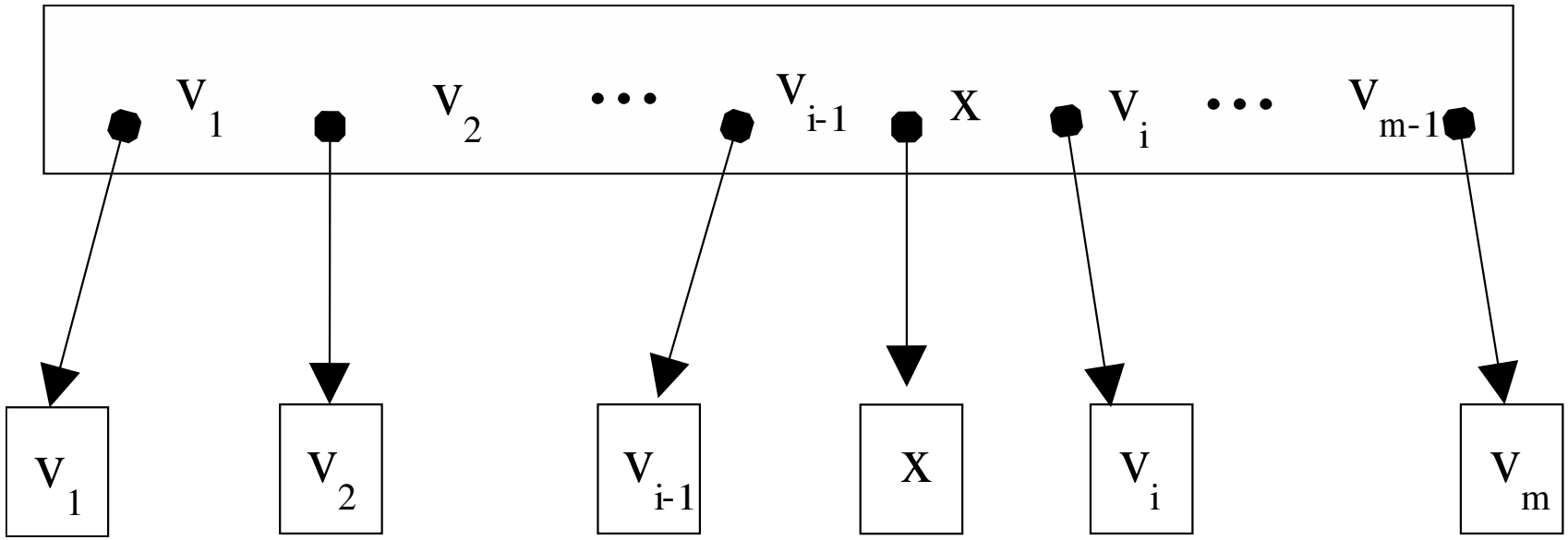
$\Rightarrow$sehr breite, flache Bäume

Einfügen in (a,b)-Baum

Schritt 1: Suche Schlüssel x. Falls x noch nicht drin ist, wird die Einfügeposition für x gefunden:



Schritt 2: Füge x in v ein. v ist ein Knoten, dessen Kinder Blätter sind.

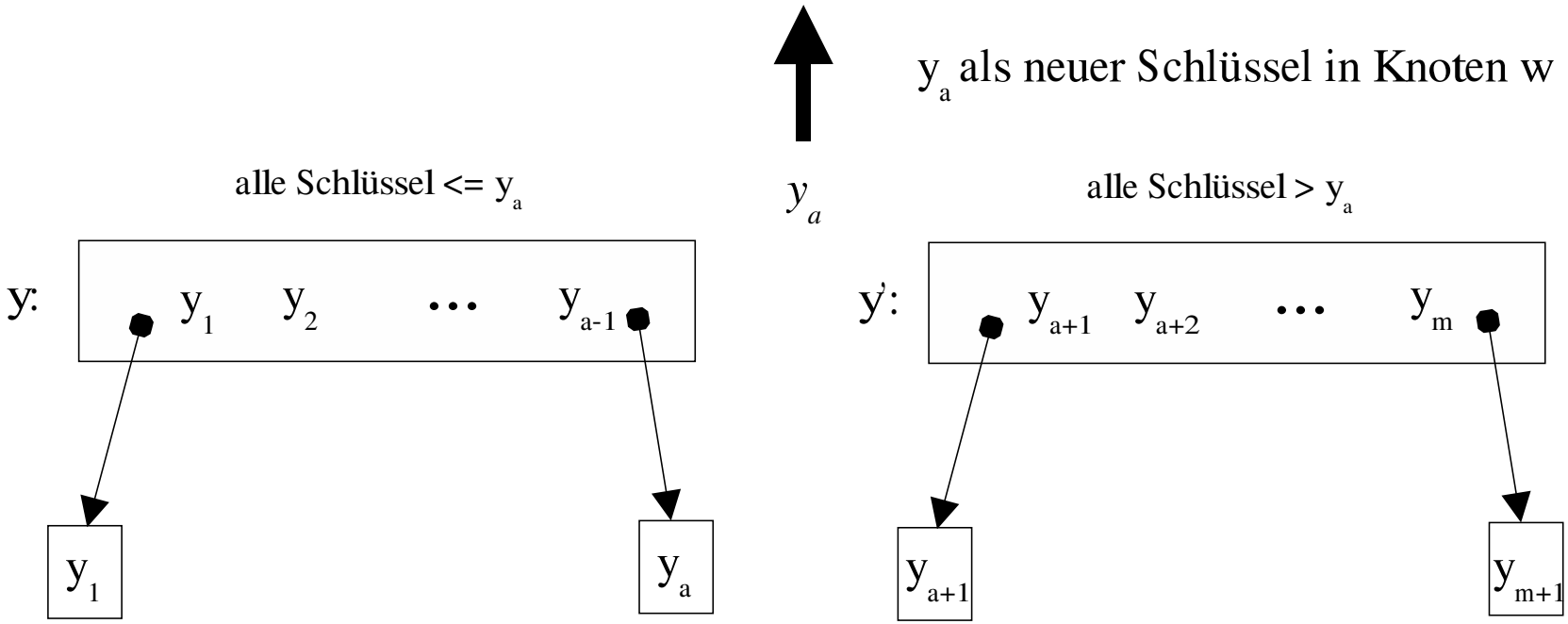


Es ergeben sich nun zwei Fälle:

- $m+1 \leq b \Rightarrow$ fertig!
- $m+1 = b+1$ (d. h. vorher war $m = b$) $\Rightarrow$ Schritt 3

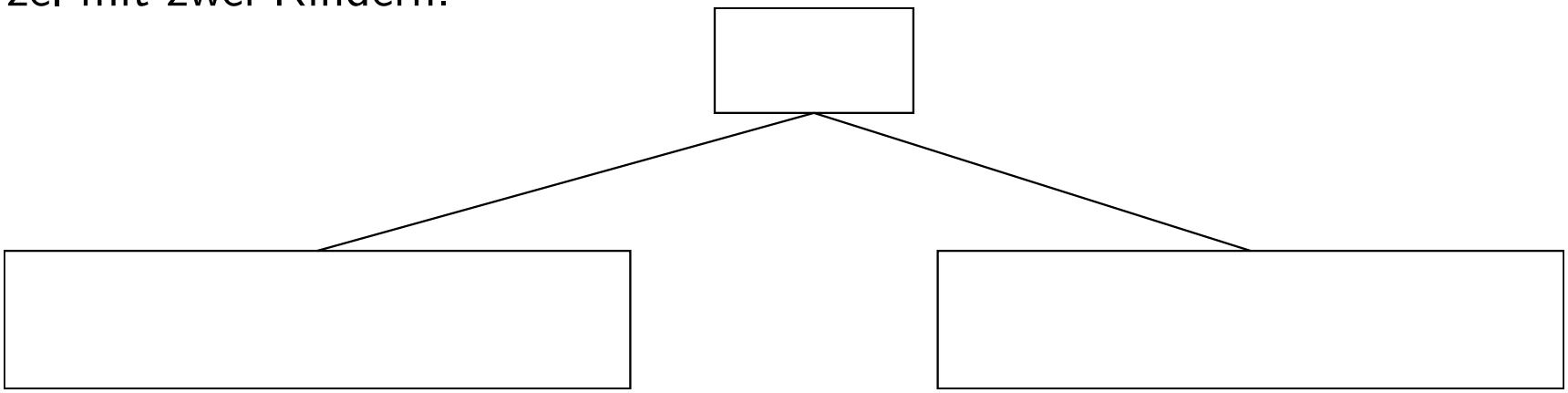
Schritt 3: Es war $m = b$ in v. Nenne den neu entstandenen Knoten y. Dieser hat entsprechend zunächst $b+1$ Kinder, das ist verboten.

- Spalte y in zwei Knoten: y, y'
- y behält a Kinder
- y' erhält $b + 1 - a \geq (2a - 1) + 1 - a = a \Rightarrow$ d. h. y' hat min. a Kinder



Schritt 4: Fahre rekursiv bis zur Wurzel fort (falls nötig)

Schritt 5: Nach (eventuell notwendigem) Splitten der Wurzel erhält man eine neue Wurzel mit zwei Kindern.



Bemerkung: Das Löschen eines Schlüssels verläuft analog in umgekehrter Reihenfolge.

Implementation von (a,b)-Bäumen

Als Beispiel geben wir mit dem folgenden Programm eine mögliche Implementation von (a,b)-Bäumen an.

Programm: **(ab-tree.cc)**

```
#include <set>
#include <iostream>
#include <vector>
#include <cassert>
using namespace std;

const int m = 2;        // B-tree of order m
const int a = m;        // minimal number of keys
const int b = 2 * m;    // maximal number of keys

template <class T>
struct Node
```

```
{
  // data
  vector<T> keys;
  vector<Node *> children;
  Node* parent;
  // interface
  Node( Node* p ) { parent = p; }
  bool is_leaf() { return children.size() == 0; }
  Node* root() { return (parent == 0) ? this : parent->root(); }
  Node* find_node( T item );
  int find_pos( T item );
  bool equals_item( int pos, T item );
};

// finds first position i such that keys[i] >= item
template <class T>
int Node<T>::find_pos( T item )
{
  int i = 0;
```

```
  while ( ( i < keys.size() ) && ( keys[i] < item ) ) i++;
  return i;
}

// checks if the key at position pos contains item
template <class T>
bool Node<T>::equals_item( int pos, T item )
{
  return ( pos < keys.size() ) && !( item < keys[pos] );
}

// finds the node in which the item should be stored
template <class T>
Node<T>* Node<T>::find_node( T item )
{
  if ( is_leaf() ) return this;
  int pos = find_pos( item );
  if ( equals_item( pos, item ) )
    return this;
```

```
  else
    return children[pos]->find_node( item );
}

template <class VEC>
VEC subseq( VEC vec, int start, int end )
{
  int size = ( vec.size() == 0 ) ? 0 : end - start;
  VEC result( size );
  for ( int i = 0; i<size; i++ )
    result[i] = vec[i + start];
  return result;
}

// if necessary, split the node.  Returns 0 or a new root
template <class T>
Node<T>* balance( Node<T>* node )
{
  int n = node->keys.size();
```

```
if ( n <= b ) return 0;
T median = node->keys[a];
// create a new node
Node<T>* node2 = new Node<T>( node->parent );
node2->keys = subseq( node->keys, a+1,
                      node->keys.size() );
node2->children = subseq( node->children, a+1,
                          node->children.size() );
for ( int i=0; i<node2->children.size(); i++ )
  node2->children[i]->parent = node2;
// handle node
node->keys = subseq( node->keys, 0, a );
node->children = subseq( node->children, 0, a+1 );

Node<T>* parent = node->parent;
if ( parent == 0 )  // split the root!
{
  Node<T>* root = new Node<T>( 0 );
  root->keys.push_back( median );
```

```
    root->children.push_back( node );
    root->children.push_back( node2 );
    node->parent = root;
    node2->parent = root;
    return root;
  }
  // otherwise: insert in parent
  int pos = 0;
  while ( parent->children[pos] != node ) pos++;
  parent->keys.insert( parent->keys.begin() + pos, median );
  parent->children.insert( parent->children.begin()+pos+1, node2 );
  // recursive call;
  return balance( parent );
}

template <class T>
void show( Node<T> *node )
{
  cout << node << ": ";
```

```
    if ( node->children.size() > 0 )
    {
      cout << node->children[0];
      for ( int i=0; i<node->keys.size(); i++ )
        cout << " |" << node->keys[i] << "| "
             << node->children[i + 1];
    }
    else
      for ( int i=0; i<node->keys.size(); i++ )
        cout << node->keys[i] << " ";
    cout << endl;
    for ( int i=0; i<node->children.size(); i++ )
      show( node->children[i] );
}

// we could work with a root pointer, but for later use it is
// better to wrap it into a class

template <class T>
```

```
class abTree
{
public:
  abTree() { root = new Node<T>( 0 ); }
  void insert( T item );
  bool find( T item );
  void show() { ::show( root ); }
private:
  Node<T>* root;
};

template <class T>
void abTree<T>::insert( T item )
{
  Node<T>* node = root->find_node( item );
  int i = node->find_pos( item );
  if ( node->equals_item( i, item ) )
    node->keys[i] = item;
  else
```

```
  {
    node->keys.insert( node->keys.begin()+i, item );
    Node<T>* new_root = balance( node );
    if ( new_root ) root = new_root;
  }
}

template <class T>
bool abTree<T>::find( T item )
{
  Node<T>* node = root->find_node( item );
  int i = node->find_pos( item );
  return node->equals_item( i, item );
}

int main()
{
  abTree<int> tree;
  // insertion demo
```

```
for ( int i=0; i<5; i++ )
{
  tree.insert( i );
  tree.show();
}
// testing insertion and retrieval
int n = 10;
for ( int i=0; i<n; i++ )
  tree.insert( i * i );
cout << endl;
tree.show();
for ( int i=0; i<2*n; i++ )
  cout << i << " " << tree.find( i ) << endl;
// performance test
//abTree<int> set;
set<int> set;  // should be faster
int nn = 1000000;
for ( int i=0; i<nn; i++ )
  set.insert( i * i );
```

```
    for ( int i=0; i<nn; i++ )
      set.find( i * i );
}
```

Bemerkung:

- Die Datensätze sind in allen Knoten gespeichert (Variante 2).
- Die Einfüge-Operation spaltet Knoten auf, wenn sie zu groß werden (mehr als b Schlüssel).
- Die Lösch-Operation ist nicht implementiert. Hier müssen Knoten vereinigt werden, wenn sie weniger als a Schlüssel enthalten.
- Die Effizienz ist zwar nicht schlecht, kommt aber nicht an die Effizienz der `set`-Implementation der STL heran. Ein wesentlicher Grund dafür ist die Verwendung variabel langer Vektoren zum Speichern von Schlüsseln und Kindern.

Literatur

Für eine weitergehende Darstellung von ausgeglichenen Bäumen sei auf das Buch „Grundlegende Algorithmen“ von Heun verwiesen. Noch mehr Informationen findet man im Buch „Introduction to Algorithms“ von Cormen, Leiserson, Rivest und Stein.

Bzw. besuchen Sie die Vorlesung ,,Algorithmen und Datenstrukturen”.