Von van Emde Boas Bäumen I

Dictionary Datenstruktur

Bitfelder $\to$ Definition vEB-Tree

Abstrakter Datentyp Dictionary(++)

Gesucht: Datenstruktur **D** zum Speichern von Schlüsseln (ggf. Schlüssel/Daten-Paare) mit folgenden Operationen.

insert(x)	Füge x in D ein.	O(log n)
delete(x)	Lösche x aus D.	O(log n)
find(x)	Ist $x \in D$?	O(log n)
min()	Liefere kleinstes Element in D.	O(log n), **O(1)***
max()	Liefere größtes Element in D.	O(log n), **O(1)***
closeBelow(x)	Liefere das größtes Element y in D mit $y < x$.	O(log n)
closeAbove(x)	Liefere das kleinste Element y in D mit $y > x$.	O(log n)

Mögliche Datenstrukturen

Balancierte Suchbäume (AVL-Baum, Rot-Schwarz-Baum,…)

* Wenn min/max zusätzlich gespeichert und bei insert/delete aktualisiert wird

Optimal wenn Schlüssel nur durch Vergleiche unterschieden werden können!

Aber häufiger Spezialfall...

Was, wenn die Schlüssel **ganzzahlig** und nur aus dem Intervall **[0,...,u-1]** sind?

Bitfeld mit **u** Einträgen

Größe: O(u) statt **O(n)**

0	1	2	3	4	5		u-2	u-1
0	**1**	**1**	**0**	**0**	**1**	**...**	**1**	**0**

min (+ min/max Verweise) **max**

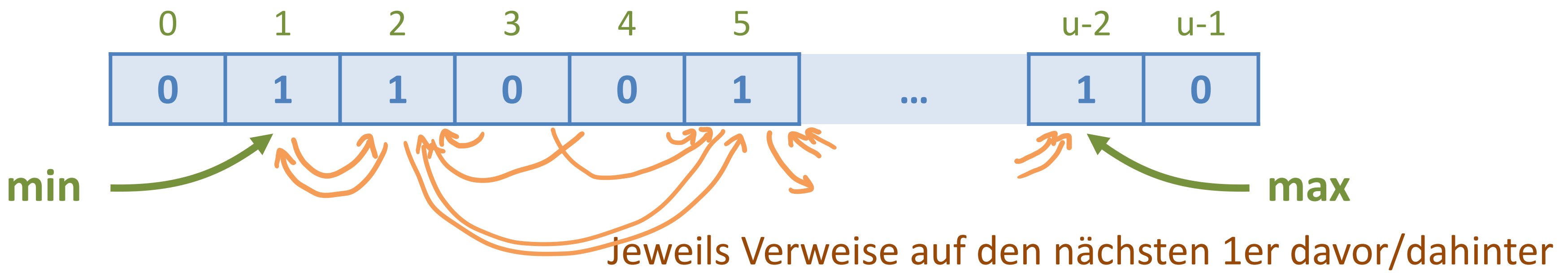
Operationen	**Bal. Suchbaum**	**Bitfeld**	**BF+mn/mx**
insert(x)	**O(log n)**	**O(1)**	**O(1)**
delete(x)	**O(log n)**	**O(1)**	**O(u)**
find(x)	**O(log n)**	**O(1)**	**O(1)**
min()/max()	O(log n), **O(1)**	**O(u)**	**O(1)**
closeAbove(x), cB(x)	**O(log n)**	**O(u)**	**O(u)**

n < u, also O(u) schlechter als O(n)

Bitfeld mit Links

Was, wenn die Schlüssel **ganzzahlig** und nur aus dem Intervall **[0,…,u-1]** sind?

Bitfeld mit **u** Einträgen (+ min/max Verweise) **Größe: O(u)** statt **O(n)**



Operationen	Bal. Suchbaum	Bitfeld	BF+mn/mx	Bitfeld mit Links
insert(x)	O(log n)	O(1)	O(1)	O(u)
delete(x)	O(log n)	O(1)	O(u)	O(u)
find(x)	O(log n)	O(1)	O(1)	O(1)
min()/max()	O(log n), **O(1)**	O(u)	O(1)	O(1)
closeAbove(x), cB(x)	O(log n)	O(u)	O(u)	O(1)

n < u, also O(u) schlechter als O(n)

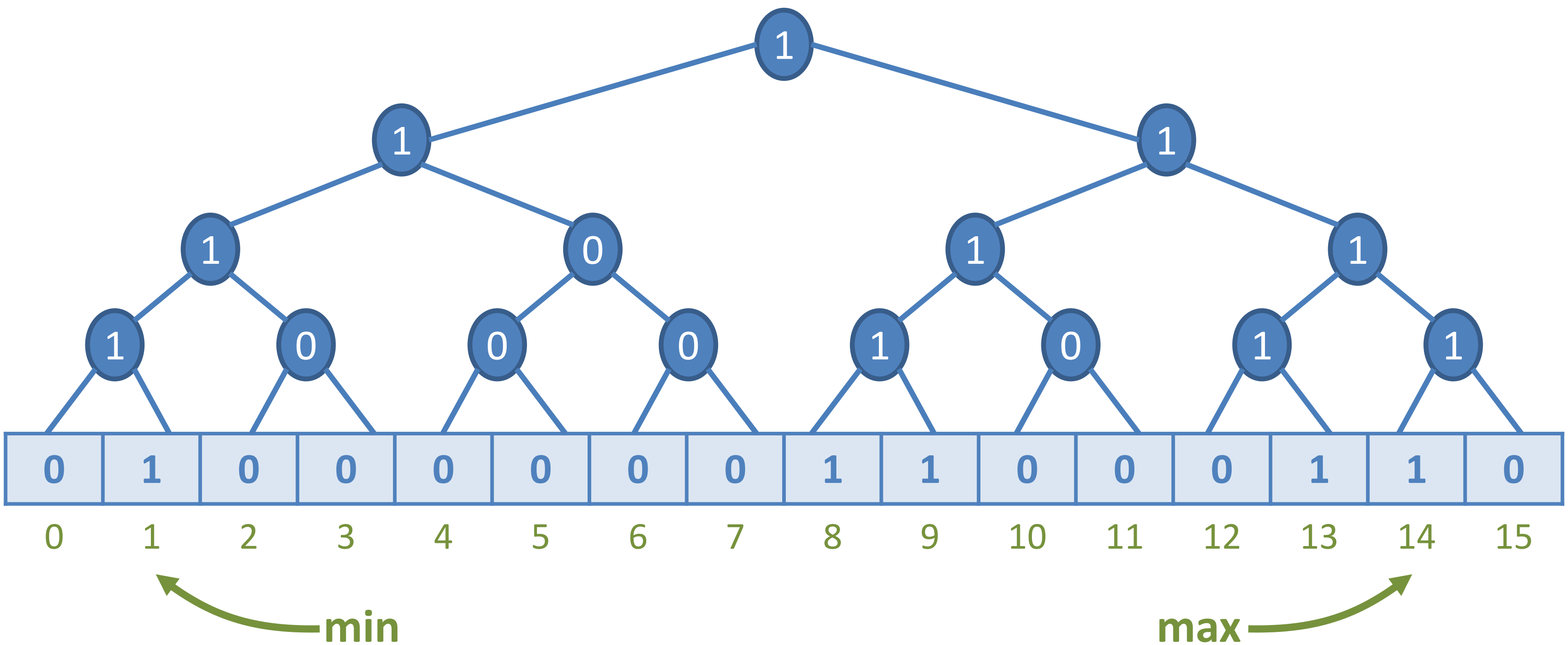
Bitfelder

Was, wenn die Schlüssel **ganzzahlig** und nur aus dem Intervall **[0,…,u-1]** sind?

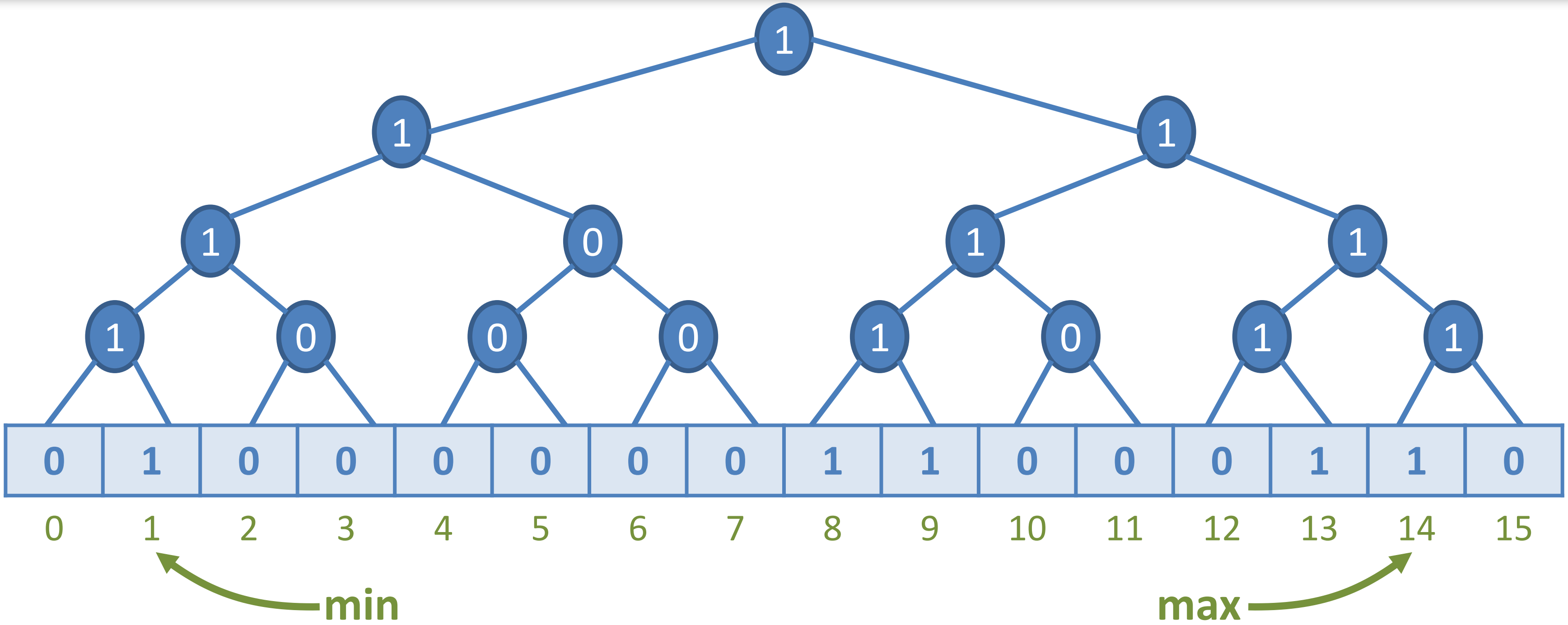
Bitfeld mit **u** Einträgen (+ min/max Verweise)

+ vollst. Binärbaum: Innerer Knoten „1“ falls mind. ein 1er im Teilbaum

(Einfachheitshalber: $u = 2^w$)

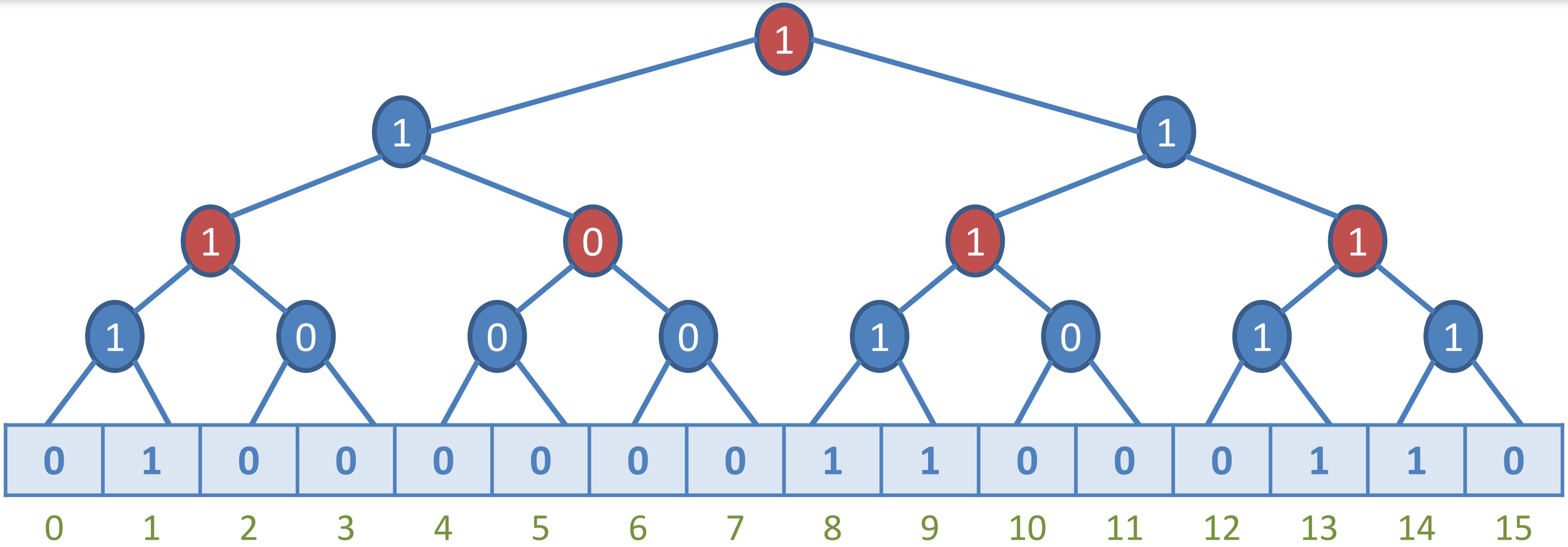


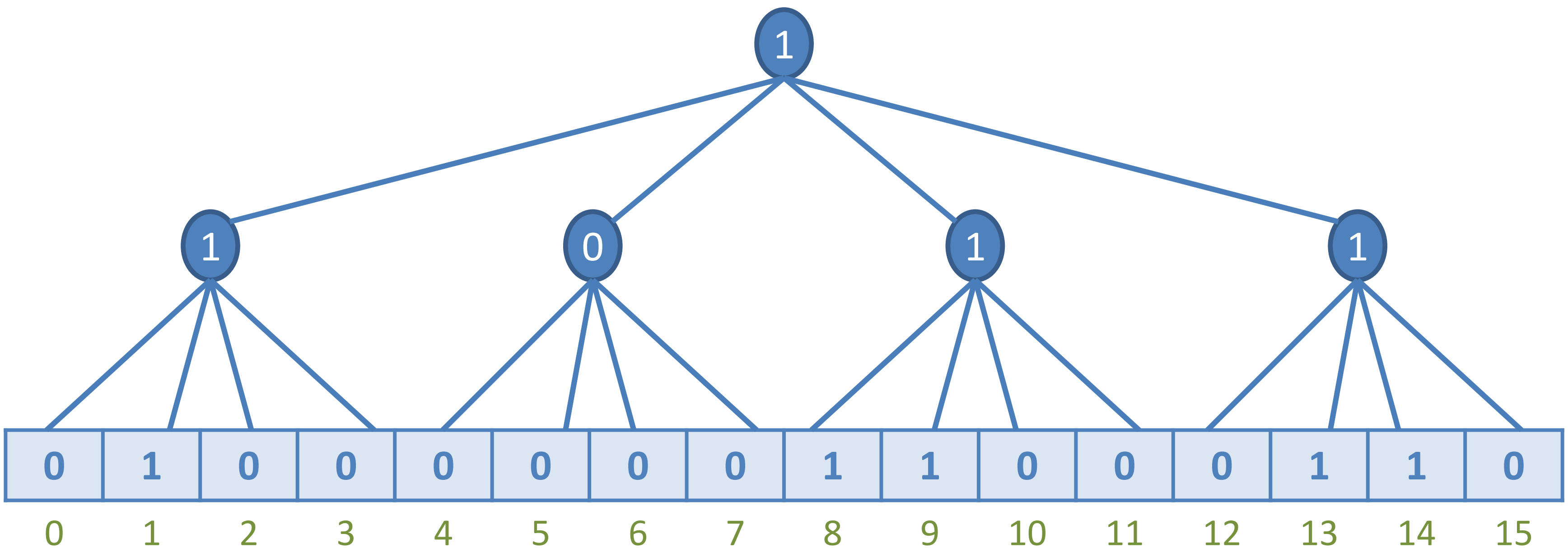
Bitfelder



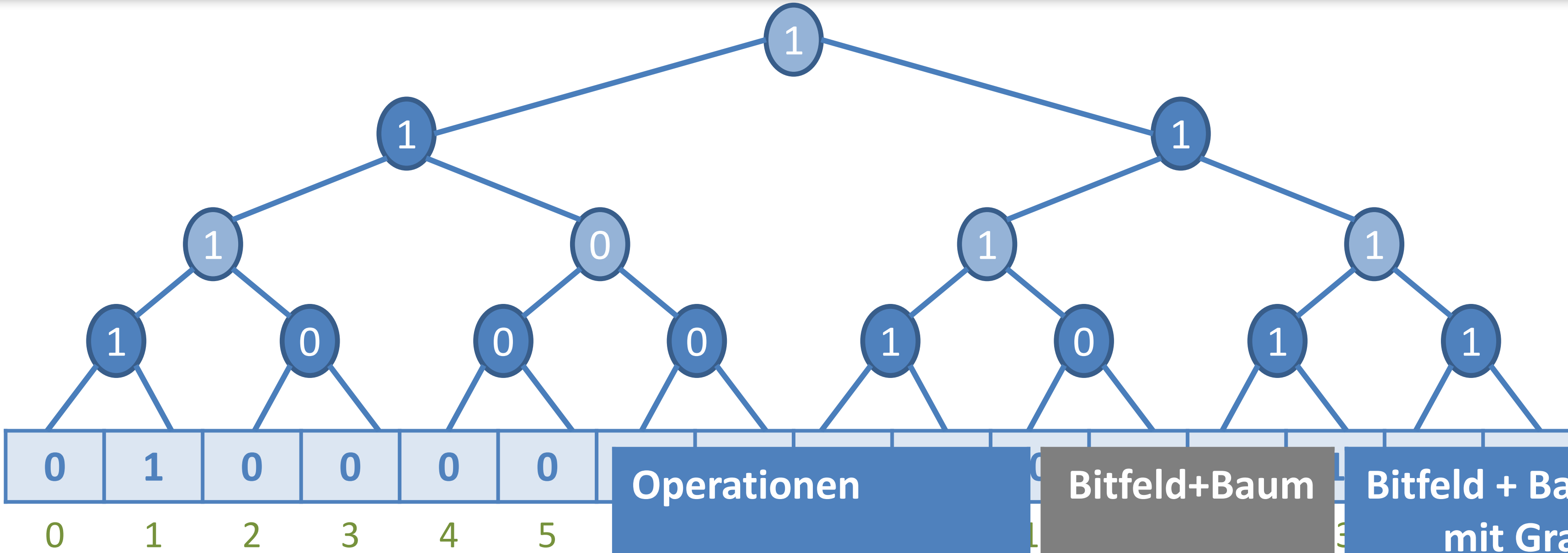
Operationen	Bal. Baum	Bitfeld	BF+mn/mx	BF+Links	Bitfeld+Baum
insert(x)	O(log n)	O(1)	O(1)	O(u)	O(log u)
delete(x)	O(log n)	O(1)	O(u)	O(u)	O(log u)
find(x)	O(log n)	O(1)	O(1)	O(1)	O(1)
min()/max()	O(1)	O(u)	O(1)	O(1)	O(1)
closeAbove(x), cB(x)	O(log n)	O(u)	O(u)	O(1)	O(log u)

Bitfeld + Baum mit Verzweigungsgrad S

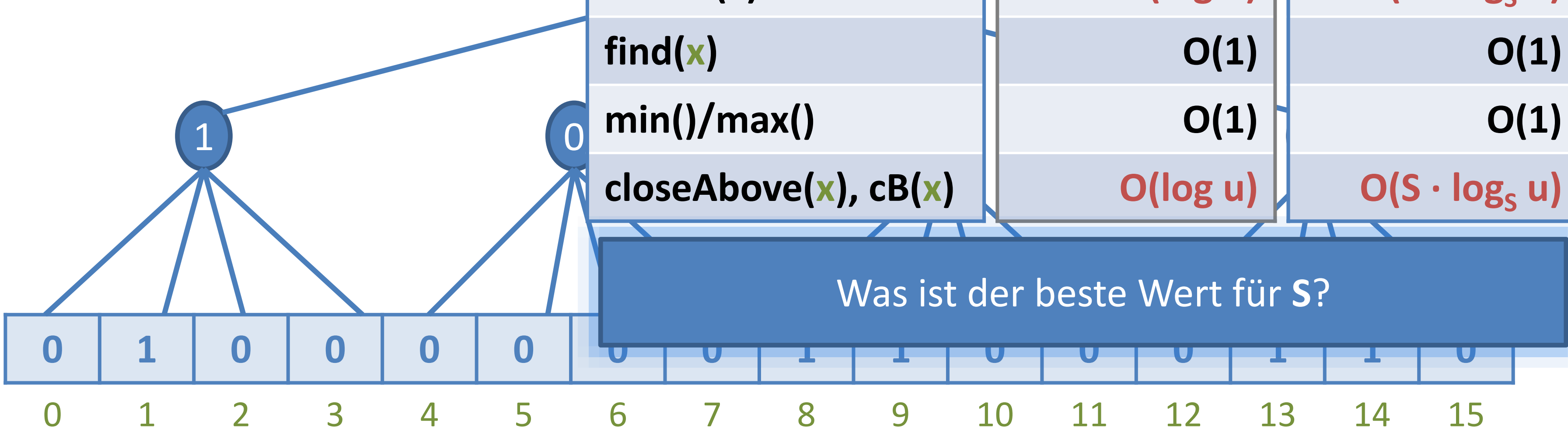




Bitfeld + Baum mit Verzweigungsgrad S



Operationen	Bitfeld+Baum	Bitfeld + Baum mit Grad S
insert(x)	O(log u)	O(log$_S$ u)
delete(x)	O(log u)	O(S · log$_S$ u)
find(x)	O(1)	O(1)
min()/max()	O(1)	O(1)
closeAbove(x), cB(x)	O(log u)	O(S · log$_S$ u)



Was ist der beste Wert für **S**?

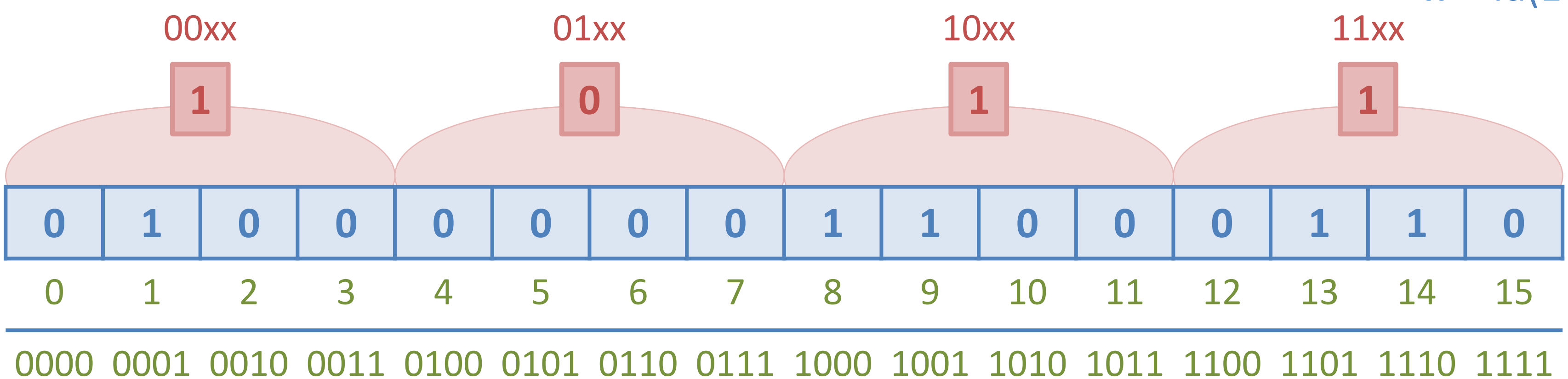
Idee, van Emde Boas…

Was, wenn die Schlüssel **ganzzahlig** und nur aus dem Intervall **[0,…,u-1]** sind?

$\Rightarrow$ Klaro: Ganze Zahlen werden am Computer als **Bitfeld** repräsentiert.

$\Rightarrow$ $\lceil \log_2 u \rceil = w$ Bits. Einfachheithalber wieder: $u = 2^w$

u = 16
w = ld(16) = 4



Wieviele Zahlen kann man mit **w/2** vielen Bits kodieren?

$\Rightarrow$ $2^{w/2} = \sqrt{2^w} = \sqrt{u}$ [Anders ausgedrückt: Verdopplung der Bits $\Rightarrow$ Quadrieren der Anzahl der darstellbaren Zahlen]

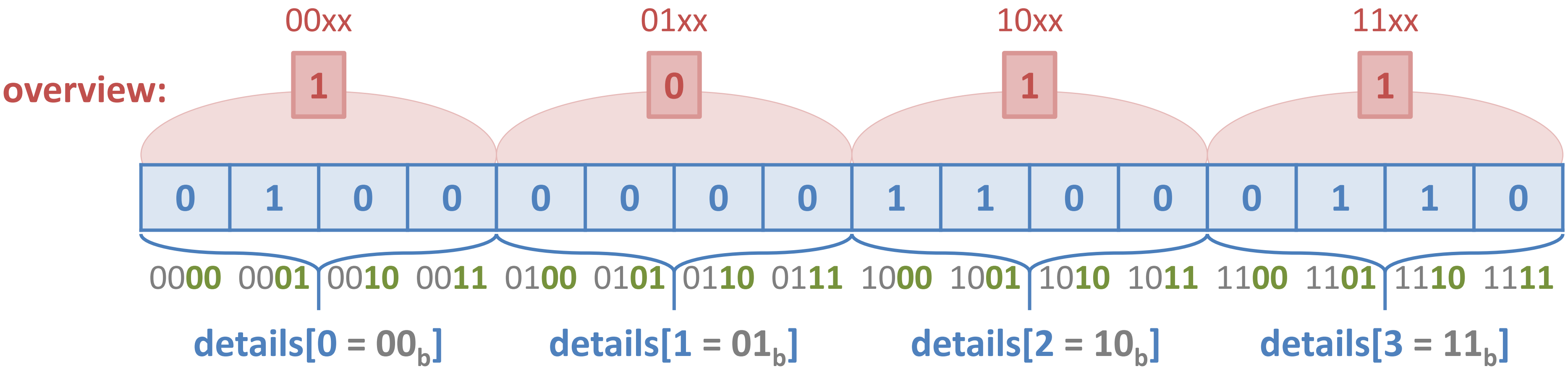
Definition. [Voraussetzung: **w** ist gerade]

$\mathrm{high}(x) := \lfloor x/\sqrt{u} \rfloor$ …Zahl, die sich aus den oberen **w/2** Bits von **x** ergibt

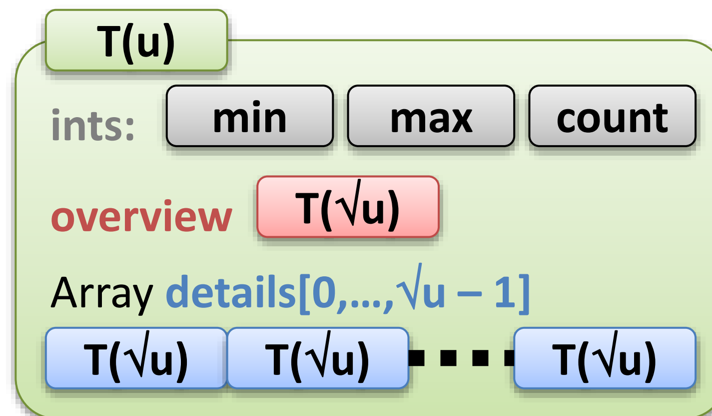
$\mathrm{low}(x) := x \bmod \sqrt{u}$ …Zahl, die sich aus den unteren **w/2** Bits von **x** ergibt

van Emde Boas Tree

Ganzzahlige Schlüssel aus dem Intervall **[0,…,u-1]** benötigen $\lceil \log_2 u \rceil = w$ Bits.
Annahme nun: $w = 2^k$



Rekursive Definition: van Emde Boas Tree.
Ein vEB-Tree **T(u)** kodiert, welche Schlüssel aus einem Intervall **[0,…,u-1]** enthalten sind.

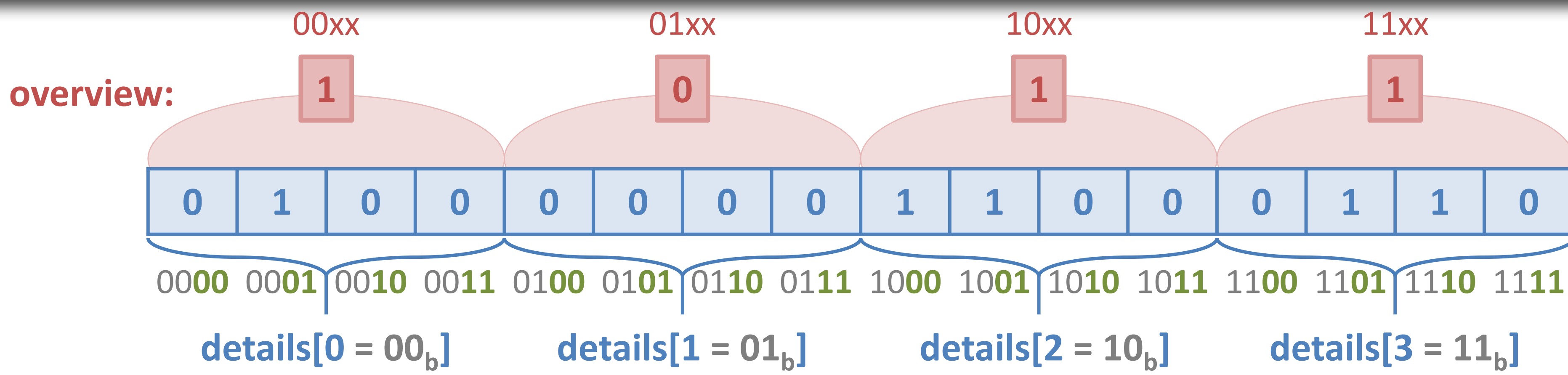


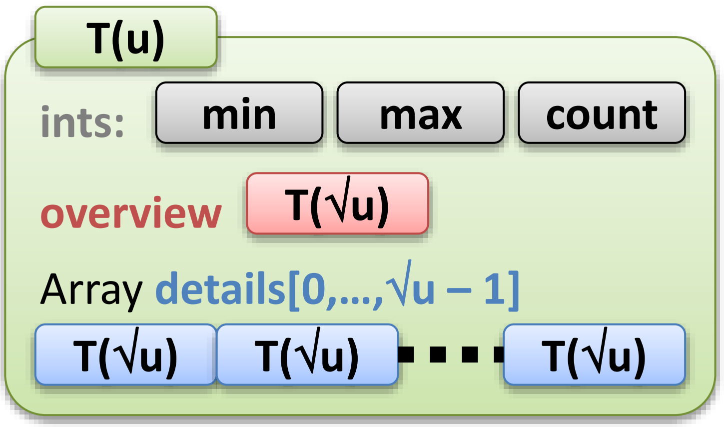

```
T(u) {
  int min, max, count;
  T(√u) overview;
  T(√u) details[√n];
}
```

T(u): Schlüssel hat **w** Bits
T(√u): Schlüssel hat **w/2** Bits

Wähle Konstante (zB. 64 bit): T(2^6=64) = einfaches Bitfeld mit O(1)-Ops

van Emde Boas Tree





Bedeutung.

min/max: Speichert kleinsten/größten Schlüssel in **T(u)**

count: #Schlüssel in **T(u)**, inkl. **min/max**

Schlüssel **x** (außer **min/max** !) wird codiert gespeichert: **x = high(x)·√n + low(x)**
Stelle sicher, dass **high(x)** in **overview** gespeichert ist, und speichere **low(x)** in **details[high(x)]**

Beobachtung: Alle Schlüssel **x** deren **low(x)** in **details[high(x)]** gespeichert werden haben die gleichen **w/2** high-order Bits (=**high(x)**).

Invariante: **h** ∊ **overview** ⇔ **details[h]** ≠ leer

van Emde Boas Tree: insert


```
insert(vEB-Tree T, Bits w, Schlüssel x):
if w <= 5:  //normalerweise: 2^w = Maschinenwortlänge
  insert-in-Bitfeld
else if T.count < 2:
  Trage x in min und/oder max ein
  Erhöhe T.count um 1, falls x neu
else if x∉{T.min,T.max}:
  if x < T.min:
    Vertausche T.min und x
  if x > T.max:
    Vertausche T.max und x
  if T.details[high(x)].count == 0:
    insert(T.overview, w/2, high(x))
  insert(T.details[high(x)], w/2, low(x))
  Aktualisiere T.count
```

van Emde Boas Tree: insert


```
insert(vEB-Tree T, Bits w, Schlüssel x):
if w <= 5:  //normalerweise: 2^w = Maschinenwortlänge
  insert-in-Bitfeld
else if T.count < 2:
  Trage x in min und/oder max ein
  Erhöhe T.count um 1, falls x neu
else if x∉{T.min,T.max}:
  if x < T.min:
    Vertausche T.min und x
  if x > T.max:
    Vertausche T.max und x
  if T.details[high(x)].count == 0:
    insert(T.overview, w/2, high(x))
  insert(T.details[high(x)], w/2, low(x))
  Aktualisiere T.count
```

Fall 1 — O(1)

Fall 2 — O(1)

Fall 3 — O(?)
- O(1)
- O(1) oder O(?)
- O(?) oder O(1) *
- O(1)

* O(1) falls `details[high(x)]` leer war
(`insert` ist „einfach“, d.h. Fall 1 oder Fall 2)

van Emde Boas Tree: insert


```
insert(vEB-Tree T, Bits w, Schlüssel x):
if w <= 5:  //normalerweise: 2^w = Maschinenwortlänge
  insert-in-Bitfeld
else if T.count < 2:
  Trage x in min und/oder max ein
  Erhöhe T.count um 1, falls x neu
else if x∉{T.min,T.max}:
  if x < T.min:
    Vertausche T.min und x
  if x > T.max:
    Vertausche T.max und x
  if T.details[high(x)].count == 0:
    insert(T.overview, w/2, high(x))
  insert(T.details[high(x)], w/2, low(x))
  Aktualisiere T.count
```

Fall 1: O(1)

Fall 2: O(1)

O(1)

T(w/2) + O(1)

O(1)

T(w)

$T(w) = T(w/2) + O(1)$ … wie löst man diese Rekursionsformel auf?

van Emde Boas Tree: find


```
find(vEB-Tree T, Bits w, Schlüssel x):
if w <= 5:
  find-in-Bitfeld
else if T.count == 0:
  return false
else if x∈{T.min,T.max}:
  return true
else
  if T.details[high(x)].count == 0:
    return false
  return find(T.details[high(x)], w/2, low(x))
```

Laufzeit. Wieder $T(w) = T(w/2) + O(1)$?

Nein! Find geht in $O(1)$!
Alle 64bit-Bitfelder liegen in einem konsekutiven Speicherbereich als ein großes Bitfeld mit **u** Bits. ⇒ Direkt nachschauen

van Emde Boas Tree: closeAbove


```
closeAbove(vEB-Tree T, Bits w, Schlüssel x):
Löse Spezialfälle für w <= 5                 //*
Löse Spezialfälle für T.count < 3
if x < T.min:
   return T.min
TT = T.details[high(x)]
if TT.count == 0 OR TT.max <= x:
   H = closeAbove(T.overview, w/2, high(x)) //*
   L = T.details[H].min
else
   H = high(x)
   L = closeAbove(TT, w/2, low(x))
return H·2^(w/2) + L
```

* **Achtung:** Falls kein Nachfolger existiert, müsste hier ein Fehler geworfen werden

Laufzeit. Wieder $T(w) = T(w/2) + O(1)$…

van Emde Boas Tree: Laufzeit

$T(w) = T(w/2) + O(1)$

$= T(w/4) + 2 \cdot O(1)$

$= T(w/8) + 3 \cdot O(1)$

$= T(w/16) + 4 \cdot O(1)$

$w = 2^k$

angenommen, wir teilen bis zum Schluss:

$= T(w/2^k) + k \cdot O(1)$

$T(w/2^k) = O(1)$ … Bitfeld funktioniert in O(1)

$= O(1) + k \cdot O(1)$

$= O(k)$

$= O(\log w)$

$= O(\log \log u)$

Theorem. **Insert**, **delete** (analog zu Insert) und **closeAbove/closeBelow** benötigen in einem vEB-Tree $O(\log \log u)$ Zeit.

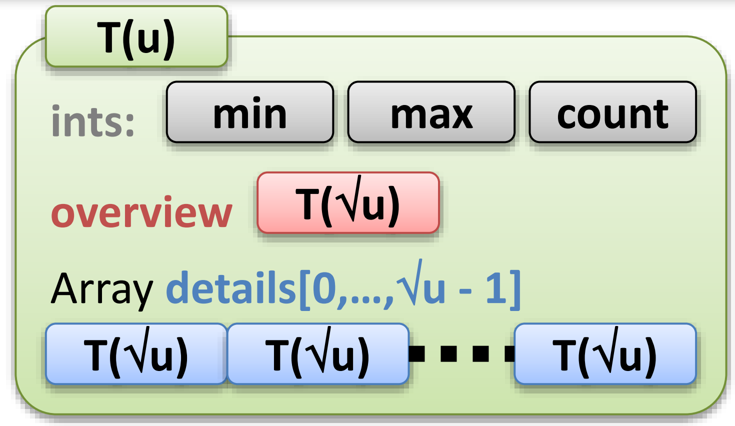
van Emde Boas Tree: Laufzeit

Operationen	Bal. Baum	BF	BF+min	BF+link	BF+Baum	vEB
insert(x)	O(log n)	O(1)	O(1)	O(u)	O(log u)	O(log log u)
delete(x)	O(log n)	O(1)	O(u)	O(u)	O(log u)	O(log log u)
find(x)	O(log n)	O(1)	O(1)	O(1)	O(1)	O(1)
min()/max()	O(lg n),**O(1)**	O(u)	O(1)	O(1)	O(1)	O(1)
closeAbove(x), cB(x)	O(log n)	O(u)	O(u)	O(1)	O(log u)	O(log log u)

$\log_2 \log_2$ **4,294,967,296 = 5**

$\log_2 \log_2$ **18,446,744,073,709,551,616 = 6**

van Emde Boas: Platzverbrauch



Theorem. vEB-Trees benötigen $O(u)$ Platz.

Beweis.

$S(u)$ … Platzverbrauch des vEB-Trees $T(u)$

Rekursionsformel:

$$
\begin{aligned}
S(u) = O(1) & \quad \text{// min, max, count} \\
+ S(\sqrt{u}) & \quad \text{// overview} \\
+ \sqrt{u} \cdot S(\sqrt{u}) & \quad \text{// details[]} \\
= O(\sqrt{u}) \cdot S(\sqrt{u}) + O(1) &
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow S(u) &= O(u^{1/2}) \cdot S(u^{1/2}) + O(1) \\
&= O(u^{1/2}) \cdot O(u^{1/4}) \cdot S(u^{1/4}) + 2 \cdot O(1) \\
&= O(u^{1/2}) \cdot O(u^{1/4}) \cdot O(u^{1/8}) \cdot S(u^{1/8}) + 3 \cdot O(1) \\
&= O(\textstyle\prod_{i=1\ldots k} u^{1/(2^i)}) \cdot S(u^{1/(2^k)}) + k \cdot O(1) \\
&= O(\textstyle\prod_{i=1\ldots \log\log u} u^{1/(2^i)}) \cdot O(1) + O(\log\log u)
\end{aligned}
$$

… ist der erste Term jetzt linear?

van Emde Boas: Platzverbrauch

$$\prod_{i=1\ldots\log\log u} u^{1/(2^i)} \overset{?}{=} O(u)$$

Setze ein: $u = 2^{2^k}$, $\log\log u = k$

$$\prod_{i=1\ldots k} (2^{2^k})^{1/(2^i)}$$
$$= \prod_{i=1\ldots k} 2^{(2^k)/(2^i)}$$
$$= \prod_{i=1\ldots k} 2^{2^{k-i}}$$
$$= \prod_{i=0\ldots k-1} 2^{2^i} \overset{?}{=} O(2^{2^k})$$
$$\overset{?}{=} O(2^{2\cdot 2^{k-1}})$$
$$\overset{?}{=} O(2^{2^{k-1}} \cdot 2^{2^{k-1}})$$

$$\Leftrightarrow$$

$$\prod_{i=0\ldots k-2} 2^{2^i} \overset{?}{=} O(2^{2^{k-1}})$$
$$\prod_{i=0\ldots k-3} 2^{2^i} \overset{?}{=} O(2^{2^{k-2}})$$

...

$$\prod_{i=0\ldots k-k} 2^{2^i} \overset{?}{=} O(2^{2^{k-(k-1)}})$$
$$2^{2^0} \overset{?}{=} O(2^{2^1})$$
$$2 = O(4) \quad ✓$$

[Ende Beweis „linearer Speicherbedarf"]