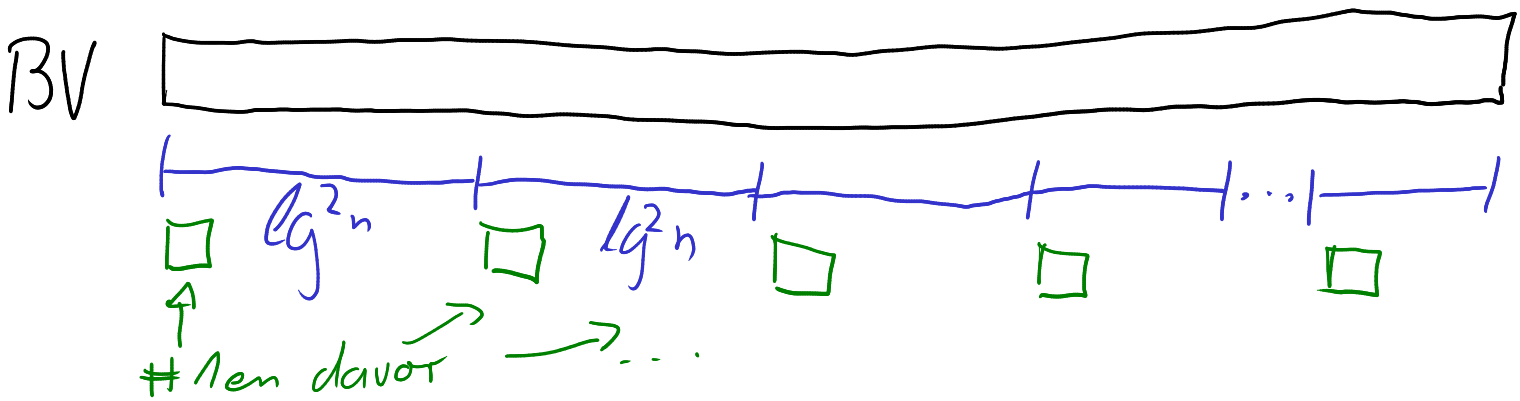
Select - Anfrage

a) $O(\lg n)$ Anfragezeit und $o(n)$ Bits Platz



1. Unterteile BV in Intervalle
2. Wir merken uns, wie viele 1en in Intervallen davor gesetzt sind
   - Warum machen wir das?
     Weil wir dann mit einer binären Suche das Intervall finden können
   - Wie viel Platz braucht das?
     #Samples · $\lg n$ Bits
   - Wir nehmen Intervalle der Größe $\lg^2 n$ Bits betrachten, denn dann haben wir $\frac{n}{\lg^2 n}$ viele Samples und brauchen $o(n)$ Bits Platz. ($\frac{n}{\lg n}$ Bits)
   - Das gesuchte Intervall finden wir in $O(\lg n)$ Zeit

3. Jetzt müssen wir noch $\lg^2 n$ viele Bits durchsuchen

↳ Können wir einfach scannen?
Nein, denn das benötigt $\Theta(\lg^2 n)$ Zeit

↳ Idee: Berechne #1en für Wörter der Länge $(\lg n)/2$ vor.

z.B.:
1101 → 3
1110 → 3
1111 → 4

↳ Wir müssen $2^{(\lg n)/2}$ Werte vorberechnen und jeder Wert braucht $\lg \lg n$ Bits Platz

↳ Wir können die Werte in $O(\sqrt{n} \cdot \lg \lg n)$ Bits Platz abspeichern. Das ist $o(n)$ Bits Platz.

↳ Jetzt können wir $(\lg n)/2$ Bits auf einmal scannen, in konstanter Zeit

↳ Wir scannen so lange bis

wir die $(\lg n)/2$ Bits gefunden haben, in denen das gesuchte Bit liegt.

↳ Danach müssen wir dann nur noch $(\lg n)/2$ Bits „manuell" scannen

↳ Das funktioniert in $O(\lg n)$ Zeit

b) $O(1)$ Anfragezeit, $o(n)$ Bits Platz und der BV ist dünn besetzt, d.h. es sind $o(\frac{n}{\lg n})$ Bits gesetzt.

↳ Wir überlegen uns, was es bedeutet, dass $o(\frac{n}{\lg n})$ Bits gesetzt sind.

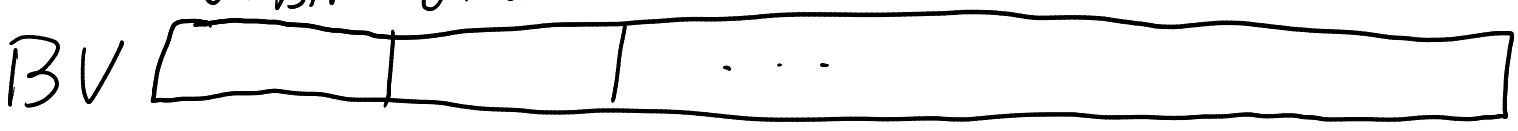
→ z.B. $\frac{n}{\lg^2 n}$, $\frac{n}{\lg n \cdot \sqrt{\lg n}}$, ...

$$\frac{n}{\lg n \cdot \lg\lg \dots \lg n}$$

→ Wir können die Positionen, an denen die 1en vorkommen direkt in einem Array abspeichern
→ $O(\#1\text{en} \cdot \lg n)$ Bits Platz $= o(n)$ Bits Platz $\#1\text{en} \in o(\frac{n}{\lg n})$

Implementierung Bit-Vektor

→ Wir befüllen die 64-Bit Wörter in unserem Bitvektor falsch rum



Index 0 1 2 3 ... 64 65 ...

Falsch herum 63 62 61 ... 0 127 126 ... 64

↳ Warum machen wir das ?
→ Es ist schneller

Entwicklung: Geht das besser?

`(block >> (63-(i%64))) & 1ULL;`

- Fülle Bit-Vector von links nach rechts

0	1	2	3	4	5	...	62	63
1	1	1	0	1	0	...	1	0
0	0	0	0	0	0	...	1	0

- Assembler-Code:

```
mov ecx, edi
not ecx
shr rsi, cl
mov eax, esi
and eax, 1
```

`(block >> (i%64)) & 1ULL;`

- Fülle Bit-Vector von rechts nach links

63	62	...	5	4	3	2	1	0
0	1	...	0	1	0	1	1	1
0	0	...	1	1	0	0	1	0

- Assembler-Code:

```
mov ecx, edi
shr rsi, cl
mov eax, esi
and eax, 1
```

← Wir sparen uns diese Negation

https://graphics.stanford.edu/~seander/bithacks.html ← Bit-Tricks wie hier genutzt

Operator-Überladung in C++

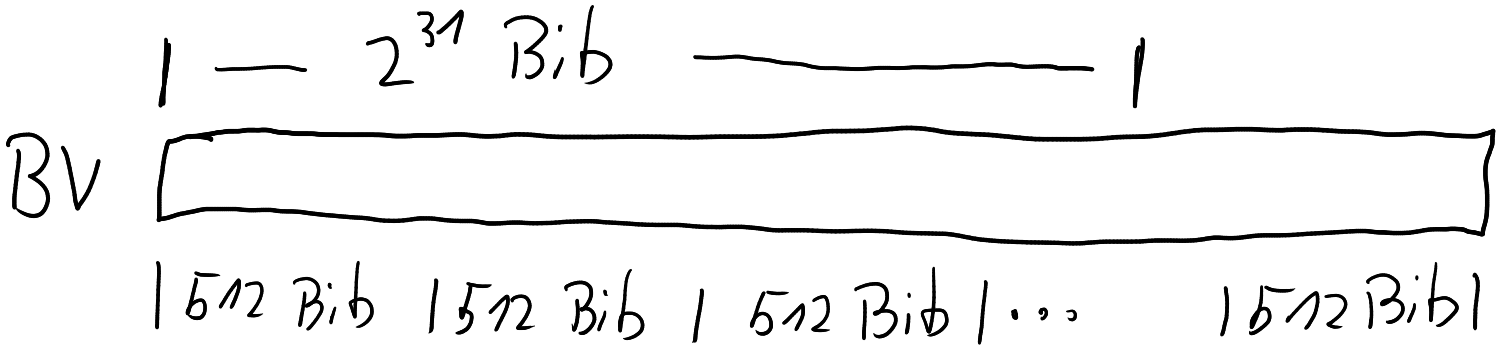
↳ T operator {[], (), ++, <<, >>, ...} (...)

↳ Wir können kein „Bit" zurückgeben

Rank für unseren Bitvektor

↳ Die Datenstruktur besteht aus drei Levels L0, L1, L2

grob ⟶ fein



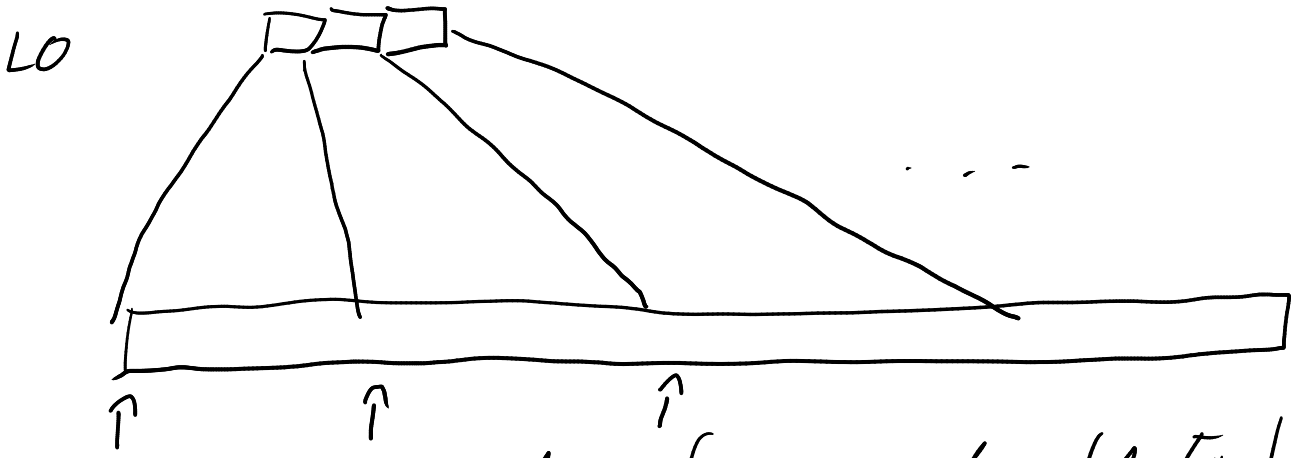
⟶ Um 512 darzustellen brauchen wir 10 Bit

⟶ L1 512 Bit-Blöcke erhalten einen L1 Eintrag ⟶ Wie viele Bit sind vor mir gesetzt

⟶ Nutze nur 32 Bit-Wort

→ L0 Eintrag für jeweils 2^{31} Bits

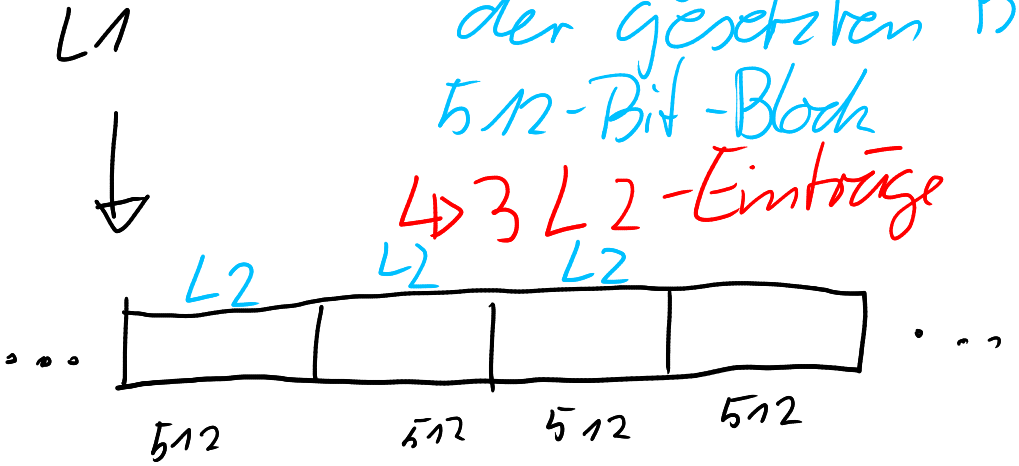
L1 Eintrag relativ zum aktuellen L0 Eintrag



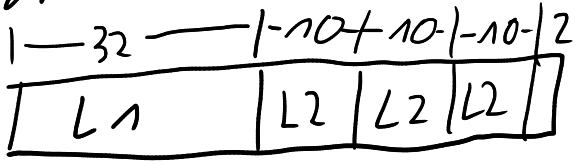
An diesen Stellen fangen die L1-Einträge bei 0 an

L2-Eintrag enthält die Anzahl der gesetzten Bits für einen 512-Bit-Block

→ 3 L2-Einträge pro L1 Block



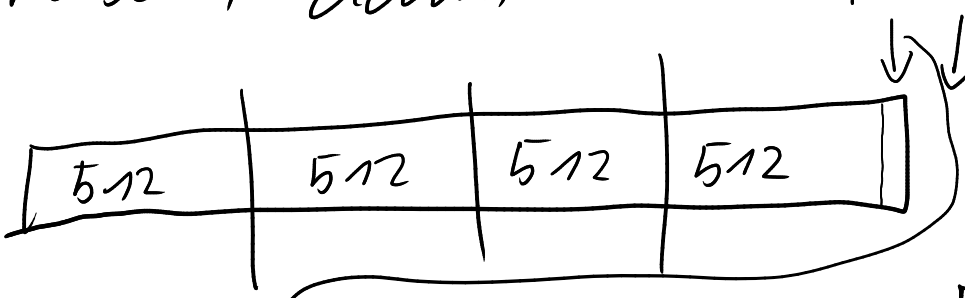
In der Praxis speichern wir die L1 und L2-Einträge in einem 64-Bit Wort



Bits bleiben übrig

Die Rank-Anfrage benötigt nur 3 L2-Einträge

Wenn wir unseren L1-Block identifiziert haben, dann



rank-Position

wenn die rank-Position hier ist, suchen wir im nächsten L1-Eintrag

Hier könnten wir den letzten L2-Eintrag gar nicht verwenden

Basiert auf:

Space-Efficient, High-Performance Rank & Select Structures on Uncompressed Bit Sequences

Dong Zhou, David G. Andersen, Michael Kaminsky†
Carnegie Mellon University, †Intel Labs

Abstract. Rank & select data structures are one of the fundamental building blocks for many modern *succinct data structures*. With the continued growth of massive-scale information services, the space efficiency of succinct data structures is becoming increasingly attractive in practice. In this paper, we re-examine the design of rank & select data structures from the bottom up, applying an architectural perspective to optimize their operation. We present our results in the form of a recipe for constructing space and time efficient rank & select data structures for a given hardware architecture. By adopting a *cache-centric* design approach, our rank & select structures impose space overhead as low as the most space-efficient, but slower, prior designs—only 3.2% and 0.39% extra space respectively—while offering performance competitive with the highest-performance prior designs.