Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Übersicht I

Empfehlungen:

oder: die Edward-Kmett-is-awesome-slide

„Functionally Oblivious and Succinct“ - Edward Kmett
`https://www.youtube.com/watch?v=WE2a90Bov0Q`

„Purely functiontional data structures“ - Chris Okasaki
`www.cs.cmu.edu/~rwh/theses/okasaki.pdf`

Vokabular:

Vokabular: Amortisation

Remember big O?

Ihr erinnert euch wahrscheinlich alle noch an Laufzeitanalysen mit der $\mathcal{O}$-Notation. Hierbei wird die Größe des Eingabeproblems (z.B. Länge eines Arrays oder Anzahl Knoten in einem Graph, i.d.R. n) zur Anzahl der Rechenschritte zur Lösung des Problems in Verbindung gesetzt.

Remember big O?

Ihr erinnert euch wahrscheinlich alle noch an Laufzeitanalysen mit der $\mathcal{O}$-Notation. Hierbei wird die Größe des Eingabeproblems (z.B. Länge eines Arrays oder Anzahl Knoten in einem Graph, i.d.R. n) zur Anzahl der Rechenschritte zur Lösung des Problems in Verbindung gesetzt.

Dabei gibt es ein paar Dinge zu beachten:

Remember big O?

Ihr erinnert euch wahrscheinlich alle noch an Laufzeitanalysen mit der $\mathcal{O}$-Notation. Hierbei wird die Größe des Eingabeproblems (z.B. Länge eines Arrays oder Anzahl Knoten in einem Graph, i.d.R. n) zur Anzahl der Rechenschritte zur Lösung des Problems in Verbindung gesetzt.

Dabei gibt es ein paar Dinge zu beachten:

- Es wird (i.d.R.) nur die *Laufzeit* betrachtet, und z.B. nicht, wie viel Speicher benötigt wird.

Remember big O?

Ihr erinnert euch wahrscheinlich alle noch an Laufzeitanalysen mit der $\mathcal{O}$-Notation. Hierbei wird die Größe des Eingabeproblems (z.B. Länge eines Arrays oder Anzahl Knoten in einem Graph, i.d.R. n) zur Anzahl der Rechenschritte zur Lösung des Problems in Verbindung gesetzt.

Dabei gibt es ein paar Dinge zu beachten:

- Es wird (i.d.R.) nur die *Laufzeit* betrachtet, und z.B. nicht, wie viel Speicher benötigt wird.
- Es wird nur die Laufzeit eines *optimalen* Algorithmus zur Lösung des Problems betrachtet.

Remember big O?

Ihr erinnert euch wahrscheinlich alle noch an Laufzeitanalysen mit der $\mathcal{O}$-Notation. Hierbei wird die Größe des Eingabeproblems (z.B. Länge eines Arrays oder Anzahl Knoten in einem Graph, i.d.R. n) zur Anzahl der Rechenschritte zur Lösung des Problems in Verbindung gesetzt.

Dabei gibt es ein paar Dinge zu beachten:

- Es wird (i.d.R.) nur die *Laufzeit* betrachtet, und z.B. nicht, wie viel Speicher benötigt wird.
- Es wird nur die Laufzeit eines *optimalen* Algorithmus zur Lösung des Problems betrachtet.
- Es wird nur die Verbindung zu einer *Klasse* von Komplexität hergestellt. Konstante Faktoren werden ignoriert.

Remember big O?

Ihr erinnert euch wahrscheinlich alle noch an Laufzeitanalysen mit der $\mathcal{O}$-Notation. Hierbei wird die Größe des Eingabeproblems (z.B. Länge eines Arrays oder Anzahl Knoten in einem Graph, i.d.R. n) zur Anzahl der Rechenschritte zur Lösung des Problems in Verbindung gesetzt.

Dabei gibt es ein paar Dinge zu beachten:

- Es wird (i.d.R.) nur die *Laufzeit* betrachtet, und z.B. nicht, wie viel Speicher benötigt wird.
- Es wird nur die Laufzeit eines *optimalen* Algorithmus zur Lösung des Problems betrachtet.
- Es wird nur die Verbindung zu einer *Klasse* von Komplexität hergestellt. Konstante Faktoren werden ignoriert.
- Es wird nur das *asymptotische* Wachstumsvehalten in der Zeit (also für „unendlich“ große Eingaben) betrachtet.

Das zweithäufigste Beispiel: Suchen

Angenommen, wir haben eine Liste von n Einträgen irgendeiner Art und wir suchen einen bestimmten davon.
Der Ansatz ist, dass wir von vorne jeden Eintrag einmal anschauen, überprüfen ob es unser Ziel ist, und im Zweifelsfall mit dem nächsten Element weiter machen.

Das zweithäufigste Beispiel: Suchen

Angenommen, wir haben eine Liste von n Einträgen irgendeiner Art und wir suchen einen bestimmten davon.
Der Ansatz ist, dass wir von vorne jeden Eintrag einmal anschauen, überprüfen ob es unser Ziel ist, und im Zweifelsfall mit dem nächsten Element weiter machen.

Jetzt können verschiedene Dinge passieren:

Das zweithäufigste Beispiel: Suchen

Angenommen, wir haben eine Liste von n Einträgen irgendeiner Art und wir suchen einen bestimmten davon.
Der Ansatz ist, dass wir von vorne jeden Eintrag einmal anschauen, überprüfen ob es unser Ziel ist, und im Zweifelsfall mit dem nächsten Element weiter machen.

Jetzt können verschiedene Dinge passieren:

- *best case:* Der Eintrag ist der erste in der Liste. Wir sind sofort fertig, ohne weitersuchen zu müssen.

Das zweithäufigste Beispiel: Suchen

Angenommen, wir haben eine Liste von n Einträgen irgendeiner Art und wir suchen einen bestimmten davon.
Der Ansatz ist, dass wir von vorne jeden Eintrag einmal anschauen, überprüfen ob es unser Ziel ist, und im Zweifelsfall mit dem nächsten Element weiter machen.

Jetzt können verschiedene Dinge passieren:

- *best case:* Der Eintrag ist der erste in der Liste. Wir sind sofort fertig, ohne weitersuchen zu müssen.
- *worst case:* Der Eintrag ist der letzte oder gar nicht in der Liste. Wir müssen die gesamte Liste durchgehen.

Das zweithäufigste Beispiel: Suchen

Angenommen, wir haben eine Liste von n Einträgen irgendeiner Art und wir suchen einen bestimmten davon.
Der Ansatz ist, dass wir von vorne jeden Eintrag einmal anschauen, überprüfen ob es unser Ziel ist, und im Zweifelsfall mit dem nächsten Element weiter machen.

Jetzt können verschiedene Dinge passieren:

- *best case:* Der Eintrag ist der erste in der Liste. Wir sind sofort fertig, ohne weitersuchen zu müssen.
- *worst case:* Der Eintrag ist der letzte oder gar nicht in der Liste. Wir müssen die gesamte Liste durchgehen.
- *average case:* Der Eintrag ist irgendwo sonst in der Liste. Im Schnitt müssen wir uns die Hälfte aller Elemente ansehen.

Ein paar typische Laufzeiten:

- $\mathcal{O}(1)$: Konstante Zeit
  Feststellen, ob eine ganze Zahl gerade oder ungerade ist.

Ein paar typische Laufzeiten:

- $\mathcal{O}(1)$: Konstante Zeit
  Feststellen, ob eine ganze Zahl gerade oder ungerade ist.
- $\mathcal{O}(\log n)$: Logarithmische Zeit
  Binäre Suche

Ein paar typische Laufzeiten:

- $\mathcal{O}(1)$: Konstante Zeit
  Feststellen, ob eine ganze Zahl gerade oder ungerade ist.
- $\mathcal{O}(\log n)$: Logarithmische Zeit
  Binäre Suche
- $\mathcal{O}(n)$: Lineare Zeit
  Lineare Suche

Ein paar typische Laufzeiten:

- $\mathcal{O}(1)$: Konstante Zeit
  Feststellen, ob eine ganze Zahl gerade oder ungerade ist.
- $\mathcal{O}(\log n)$: Logarithmische Zeit
  Binäre Suche
- $\mathcal{O}(n)$: Lineare Zeit
  Lineare Suche
- $\mathcal{O}(n \cdot \log n)$: Linear-logarithmische Zeit
  Mergesort, Heapsort ...
- $\mathcal{O}(n^2)$: Quadratische Zeit
  Bubblesort, Insertion sort

Ein paar typische Laufzeiten:

- $\mathcal{O}(1)$: Konstante Zeit
  Feststellen, ob eine ganze Zahl gerade oder ungerade ist.
- $\mathcal{O}(\log n)$: Logarithmische Zeit
  Binäre Suche
- $\mathcal{O}(n)$: Lineare Zeit
  Lineare Suche
- $\mathcal{O}(n \cdot \log n)$: Linear-logarithmische Zeit
  Mergesort, Heapsort ...
- $\mathcal{O}(n^2)$: Quadratische Zeit
  Bubblesort, Insertion sort
- $\mathcal{O}(n^3)$: Kubische Zeit
  Naive Matrizenmultiplikation und -inversion

Ein paar typische Laufzeiten:

- $\mathcal{O}(1)$: Konstante Zeit
  Feststellen, ob eine ganze Zahl gerade oder ungerade ist.
- $\mathcal{O}(\log n)$: Logarithmische Zeit
  Binäre Suche
- $\mathcal{O}(n)$: Lineare Zeit
  Lineare Suche
- $\mathcal{O}(n \cdot \log n)$: Linear-logarithmische Zeit
  Mergesort, Heapsort ...
- $\mathcal{O}(n^2)$: Quadratische Zeit
  Bubblesort, Insertion sort
- $\mathcal{O}(n^3)$: Kubische Zeit
  Naive Matrizenmultiplikation und -inversion
- $2^{\mathcal{O}(n)}$: Exponentielle Zeit
  TSP mit der Magie dynamischer Programmierung

Ein paar typische Laufzeiten:

- $\mathcal{O}(1)$: Konstante Zeit
  Feststellen, ob eine ganze Zahl gerade oder ungerade ist.
- $\mathcal{O}(\log n)$: Logarithmische Zeit
  Binäre Suche
- $\mathcal{O}(n)$: Lineare Zeit
  Lineare Suche
- $\mathcal{O}(n \cdot \log n)$: Linear-logarithmische Zeit
  Mergesort, Heapsort ...
- $\mathcal{O}(n^2)$: Quadratische Zeit
  Bubblesort, Insertion sort
- $\mathcal{O}(n^3)$: Kubische Zeit
  Naive Matrizenmultiplikation und -inversion
- $2^{\mathcal{O}(n)}$: Exponentielle Zeit
  TSP mit der Magie dynamischer Programmierung
- $\mathcal{O}(n!)$: Faktorielle Zeit
  TSP mit Brute-Force-Ansatz

Jetzt wollen wir unser Repertoire um ein neues Konzept erweitern:

Amortisierte Laufzeitanalyse betrachtet nicht die asymptotische Laufzeit einer Operation unter bestimmten Annahmen (z.B. was ist der „typische“ String?), sondern produziert eine Garantie, dass eine lange Sequenz von Operationen eine bestimmte Grenze nicht überschreitet.

Jetzt wollen wir unser Repertoire um ein neues Konzept erweitern:

Amortisierte Laufzeitanalyse betrachtet nicht die asymptotische Laufzeit einer Operation unter bestimmten Annahmen (z.B. was ist der „typische“ String?), sondern produziert eine Garantie, dass eine lange Sequenz von Operationen eine bestimmte Grenze nicht überschreitet.

Die amortisierten Kosten sind also *nicht* (!) das Gleiche wie der average case. Letzterer ist eine Aussage über das Verhältnis von Eingabe und Laufzeit. Amortisierte Kosten beschreiben eine obere Grenze, die auf lange Sicht nicht überschritten wird, auch wenn einzelne Operationen darüber liegen können.

Beispiel: Dynamisches Array

Man stelle sich ein Array vor, das wächst, wenn Elemente hinzugefügt werden. Lassen wir es leer mit Platz für vier Elemente starten.

Beispiel: Dynamisches Array

Man stelle sich ein Array vor, das wächst, wenn Elemente hinzugefügt werden. Lassen wir es leer mit Platz für vier Elemente starten.

Die ersten vier `push`-Operationen brauchen nur konstante Zeit ($\mathcal{O}(1)$). Die fünfte allerdings benötigt länger, weil jetzt erst ein neues Array (der Länge 8) angelegt werden muss und die Werte übertragen werden müssen ($\mathcal{O}(n)$), usw. . .

Beispiel: Dynamisches Array

Man stelle sich ein Array vor, das wächst, wenn Elemente hinzugefügt werden. Lassen wir es leer mit Platz für vier Elemente starten.

Die ersten vier `push`-Operationen brauchen nur konstante Zeit ($\mathcal{O}(1)$). Die fünfte allerdings benötigt länger, weil jetzt erst ein neues Array (der Länge 8) angelegt werden muss und die Werte übertragen werden müssen ($\mathcal{O}(n)$), usw. . .

Im Schnitt müssen wir also nur alle n Operationen eine $\mathcal{O}(n)$-teure Operation durchführen, sonst konstant. Das bringt uns zu einer amortisierten Kostenfunktion $\mathcal{O}\left(\frac{n}{n}\right) = \mathcal{O}(1)$ für `push` auf dieser Sorte Array.

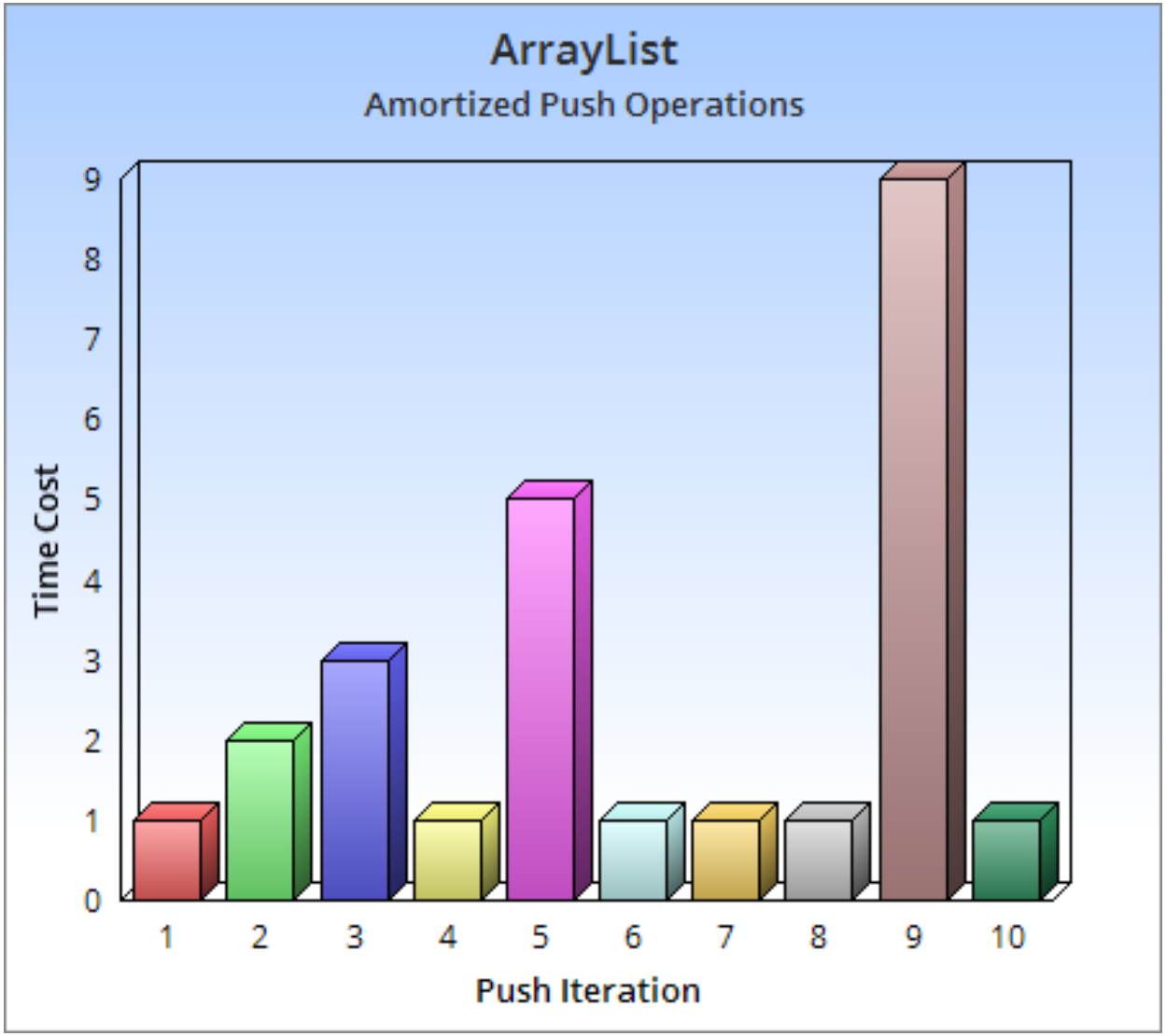


Bild: ScottDNelson, Wikipedia

Vokabular:
Succinct data structures

Eine *succinct data structure* (engl. *succinct*: knapp / kurz) ist eine Datenstruktur, die nur extrem wenig Speicherplatz benötigt, aber trotzdem noch schnelle (-ish) Abfragen zulässt.

Eine *succinct data structure* (engl. *succinct*: knapp / kurz) ist eine Datenstruktur, die nur extrem wenig Speicherplatz benötigt, aber trotzdem noch schnelle (-ish) Abfragen zulässt.

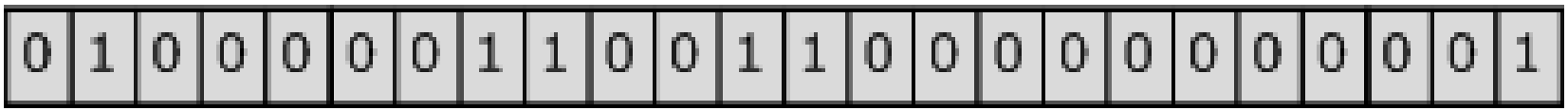
Wir bedienen uns hierfür einem Konzept aus der modernen Informationstheorie, der sogenannten (Shannon-) *Entropie*.

$$H_0(A) = \log_2 A$$

Die Einheit dieser Größe ist das Bit. Will eine Datenstruktur succinct genannt werden, sollte sie nicht mehr als einen Skalar mal ihrer Entropie im Speicher belegen.

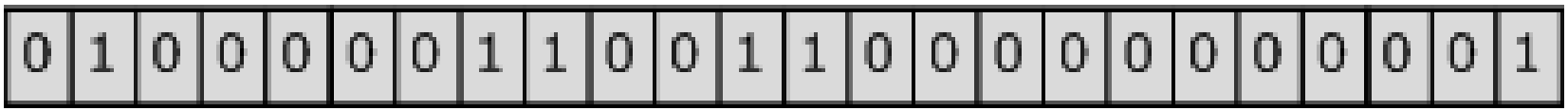
Beispiel: Succinct Dictionaries

Gegeben ein Bitvektor der Länge n mit k Einsen drin. z.B.:



Beispiel: Succinct Dictionaries

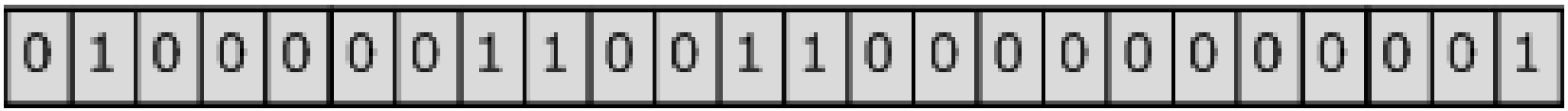
Gegeben ein Bitvektor der Länge n mit k Einsen drin. z.B.:



Von diesen Vektoren gibt es aber überhaupt nur $\binom{n}{k}$. Also kann ich mir auch einfach merken, welcher davon es ist.

Beispiel: Succinct Dictionaries

Gegeben ein Bitvektor der Länge n mit k Einsen drin. z.B.:



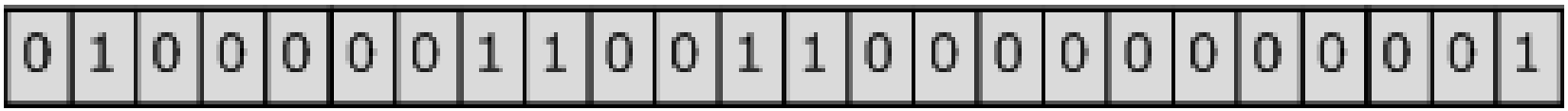
Von diesen Vektoren gibt es aber überhaupt nur $\binom{n}{k}$. Also kann ich mir auch einfach merken, welcher davon es ist.

$$H_0 = \left\lceil \log_2 \binom{n}{k} \right\rceil$$

Der Vektor lässt sich also in $\sim H_0$ Bits speichern.

Beispiel: Succinct Dictionaries

Gegeben ein Bitvektor der Länge n mit k Einsen drin. z.B.:



Von diesen Vektoren gibt es aber überhaupt nur $\binom{n}{k}$. Also kann ich mir auch einfach merken, welcher davon es ist.

$$H_0 = \left\lceil \log_2 \binom{n}{k} \right\rceil$$

Der Vektor lässt sich also in $\sim H_0$ Bits speichern.Allerdings mit $\mathcal{O}(1)$-Laufzeiten für `access` (was ist das i-te Bit?), `rank` (wie oft kommt 0/1 in $S[0..i]$ vor?) und `select` (welche Position hat die i-te 0/1?).

Wiederholung(?): Caches

Ein *Cache* (vom franz. *cacher*, verstecken) ist ein Zwischenspeicher, in den Ergebnisse (also Daten) gelegt werden können, um danach schnell (wiederholt) abgerufen zu werden statt aufwändig abgefragt oder neu berechnet zu werden.

Ein *Cache* (vom franz. *cacher*, verstecken) ist ein Zwischenspeicher, in den Ergebnisse (also Daten) gelegt werden können, um danach schnell (wiederholt) abgerufen zu werden statt aufwändig abgefragt oder neu berechnet zu werden.

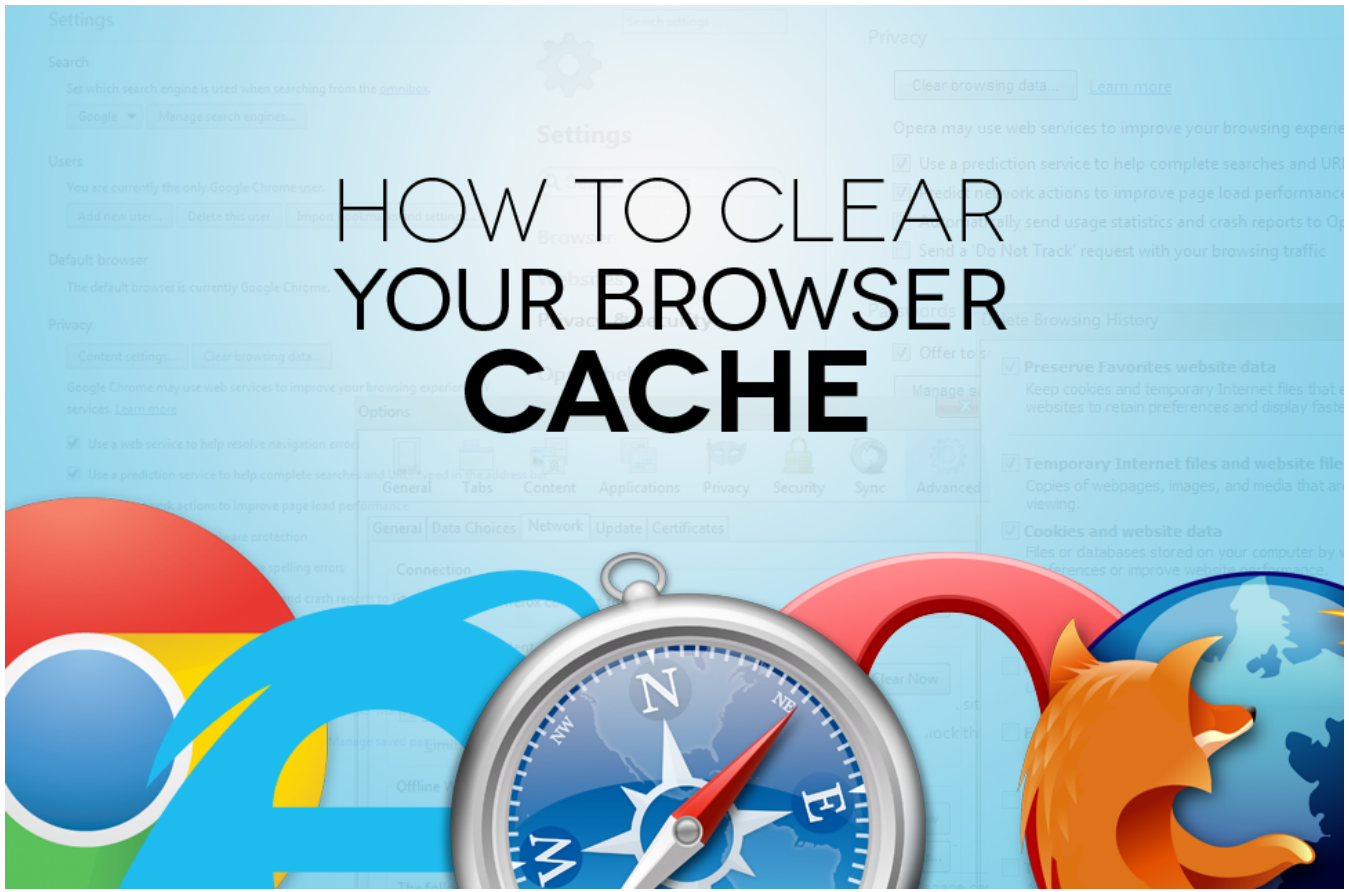
Caches (realisiert sowohl Hardware als auch Software) gibt es in eurer CPU, auf eurer Festplatte, dazwischen, im Browser, auf Webservern, und auf gewisse Art sogar in eurem Gehirn.

Ein *Cache* (vom franz. *cacher*, verstecken) ist ein Zwischenspeicher, in den Ergebnisse (also Daten) gelegt werden können, um danach schnell (wiederholt) abgerufen zu werden statt aufwändig abgefragt oder neu berechnet zu werden.

Caches (realisiert sowohl Hardware als auch Software) gibt es in eurer CPU, auf eurer Festplatte, dazwischen, im Browser, auf Webservern, und auf gewisse Art sogar in eurem Gehirn.

Kann ein Ergebnis aus dem Cache verwendet werden, so nennt man das einen *cache hit*, falls nicht, einen *cache miss*.

Tradeoff: Size vs. Speed



Tradeoff: Size vs. Speed

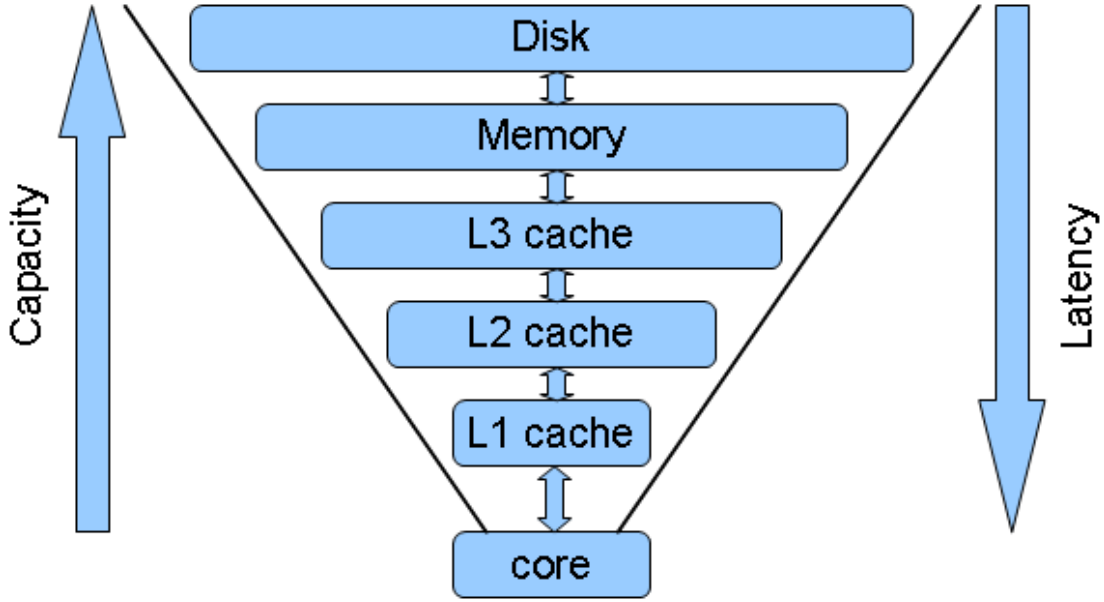
Jedes Mal wenn ein Datum angefragt wird, muss zunächst der Cache durchsucht werden, der dieses Datum enthalten könnte. Bei einem miss muss die nächste Ebene durchsucht werden, usw.

Tradeoff: Size vs. Speed

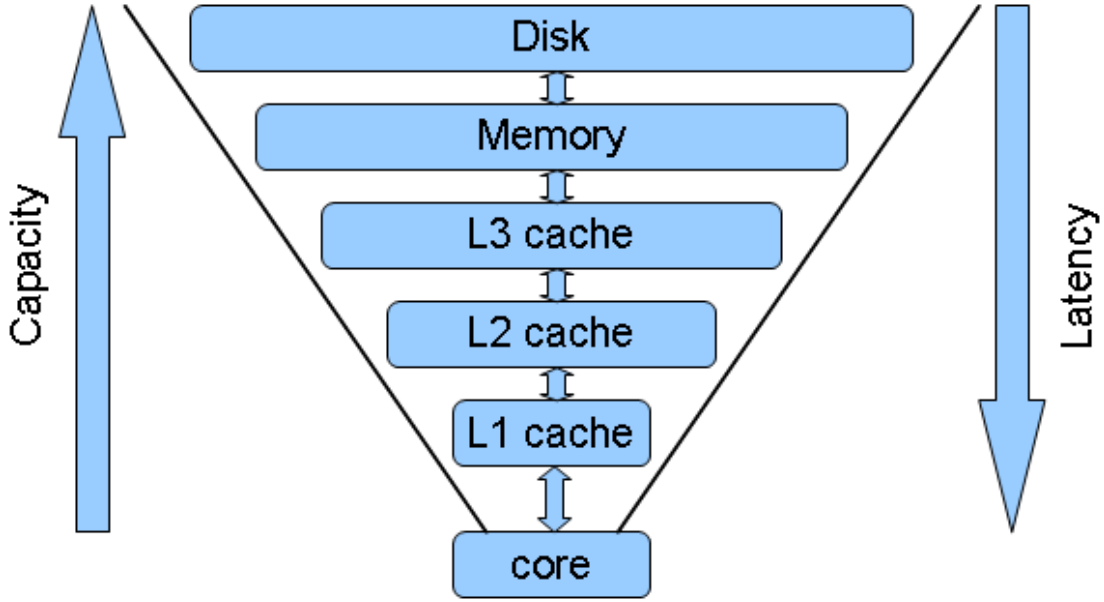
Jedes Mal wenn ein Datum angefragt wird, muss zunächst der Cache durchsucht werden, der dieses Datum enthalten könnte. Bei einem miss muss die nächste Ebene durchsucht werden, usw.

Caches in unseren Computern wachsen folglich exponentiell in sowohl ihrer Größe als auch in ihrer Langsamkeit (Latenz). Größere Caches sind also *öfter*, dafür aber *weniger* nützlich.

Zwischen eurem Code und der Festplatte liegen viele Caches. . .

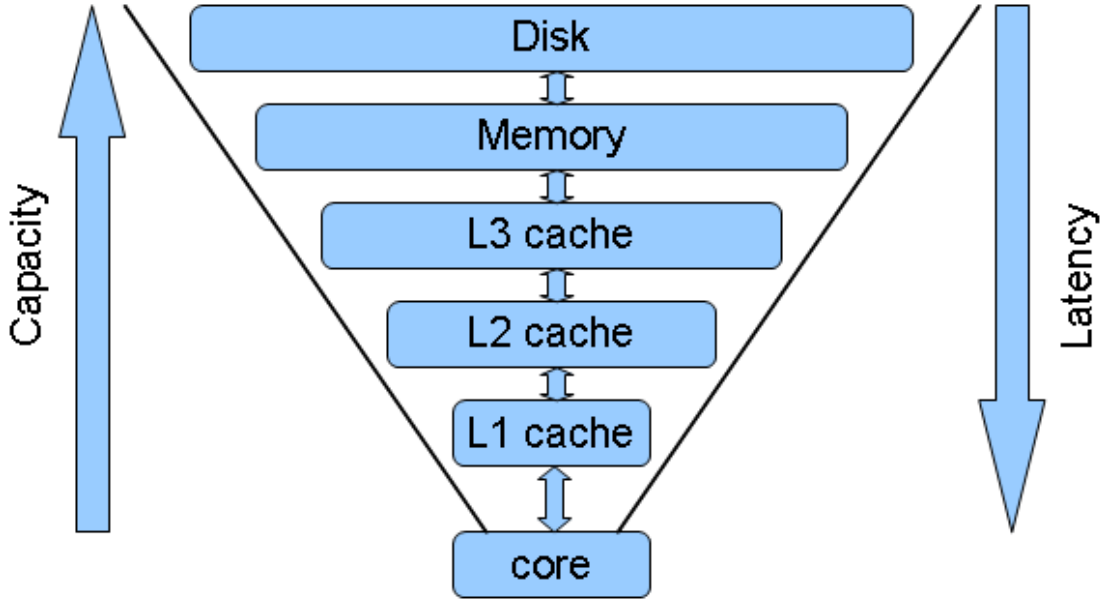


Zwischen eurem Code und der Festplatte liegen viele Caches. . .



. . . und ihr wisst nicht von allen, dass sie überhaupt existieren!

Zwischen eurem Code und der Festplatte liegen viele Caches. . .



. . . und ihr wisst nicht von allen, dass sie überhaupt existieren!
Zur Orientierung: Typische Werte wären 65 kB, 512kB und 8 MB für L1, L2 und L3.

cache-oblivious

succinct

`Data.Map`

Was haben wir?

Wir haben `Data.Map`, eine sehr gut gepflegte Bibliothek und der de-facto-Standard für Performance-Benchmarks in Haskell.

Was haben wir?

Wir haben `Data.Map`, eine sehr gut gepflegte Bibliothek und der de-facto-Standard für Performance-Benchmarks in Haskell.
In anderen Sprachen gibt es das gleiche Konzept unter anderen Namen. In Java heißt es `HashMap` und in Python `Dictionary`.

Was haben wir?

Wir haben `Data.Map`, eine sehr gut gepflegte Bibliothek und der de-facto-Standard für Performance-Benchmarks in Haskell.
In anderen Sprachen gibt es das gleiche Konzept unter anderen Namen. In Java heißt es `HashMap` und in Python `Dictionary`.
Intern basiert alles auf *trees of bounded balance*, welche wir hier allerdings nicht breit besprechen.

Was haben wir?

Wir haben `Data.Map`, eine sehr gut gepflegte Bibliothek und der de-facto-Standard für Performance-Benchmarks in Haskell.
In anderen Sprachen gibt es das gleiche Konzept unter anderen Namen. In Java heißt es `HashMap` und in Python `Dictionary`.
Intern basiert alles auf *trees of bounded balance*, welche wir hier allerdings nicht breit besprechen.

Exportiert eine reiche Auswahl an Funktionen:

```
empty  :: Map k a -- construction
size   :: Map k a -> Int
member :: Ord k => k -> Map k a -> Bool
insert :: Ord k => k -> a -> Map k a -> Map k a
lookup :: Ord k => k -> Map k a -> Maybe a
map    :: (a -> b) -> Map k a -> Map k b
...
union        :: Ord k => Map k a -> Map k a -> Map k a
difference   :: Ord k => Map k a -> Map k b -> Map k a
intersection :: Ord k => Map k a -> Map k b -> Map k a
...
```

Was hätten wir gerne?

Was hätten wir gerne?

- hocheffiziente Variante einer `Map`

Was hätten wir gerne?

- hocheffiziente Variante einer `Map`
- insbesondere *range queries* und *inserts*

Was hätten wir gerne?

- hocheffiziente Variante einer `Map`
- insbesondere *range queries* und *inserts*
- Unterstützung für *unboxed* data
  d.h. Datentypen, die einen direkten Wert darstellen und nicht einfach ein Pointer auf ein Objekt auf dem Heap sind.

Was hätten wir gerne?

- hocheffiziente Variante einer `Map`
- insbesondere *range queries* und *inserts*
- Unterstützung für *unboxed* data
  d.h. Datentypen, die einen direkten Wert darstellen und nicht einfach ein Pointer auf ein Objekt auf dem Heap sind.
- . . . während wir nicht die Nettigkeiten und Vorteile von `Haskell` aufgeben wollen.

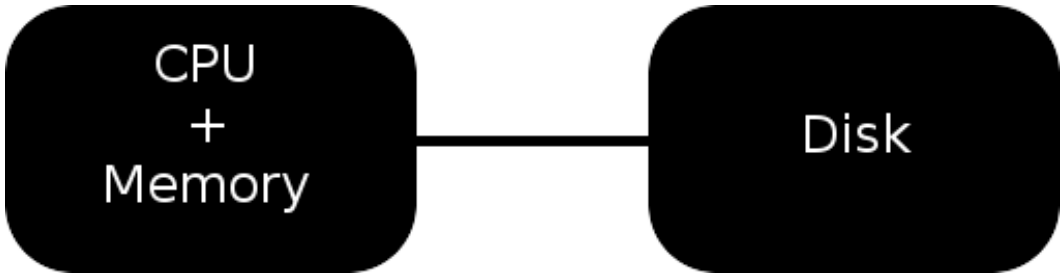
Was hätten wir gerne?

- hocheffiziente Variante einer `Map`
- insbesondere *range queries* und *inserts*
- Unterstützung für *unboxed* data
  d.h. Datentypen, die einen direkten Wert darstellen und nicht einfach ein Pointer auf ein Objekt auf dem Heap sind.
- . . . während wir nicht die Nettigkeiten und Vorteile von `Haskell` aufgeben wollen.

Was uns nicht so enorm wichtig ist, sind Anfragen nach genau einem Datenpunkt.

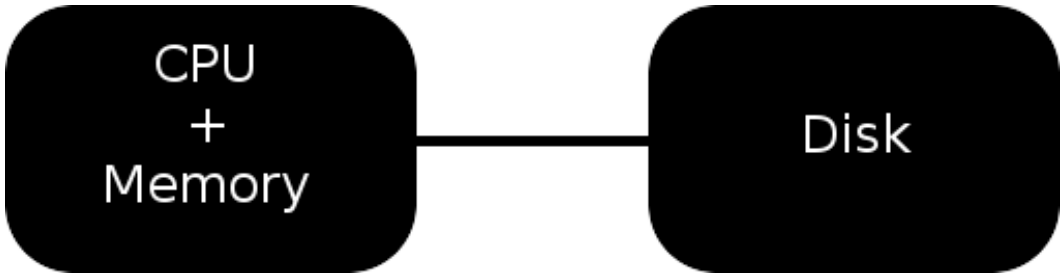
Alles in allem sieht das etwas mehr nach einer Datenbank aus. Es ist aber insbesondere ein gutes Beispiel für das Konzept, was wir in Aktion sehen wollen.

Das IO-Model (damals):



Sei hier M die Größe des RAMs.

Das IO-Model (damals):

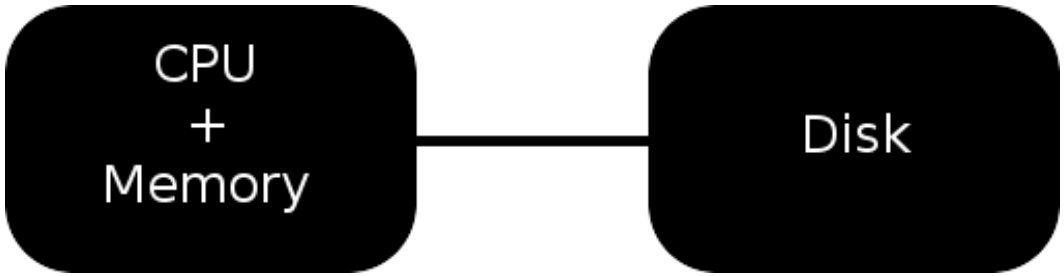


Sei hier M die Größe des RAMs.

Eigenschaften dieses Modells:

- Wir können Blöcke der Größe B lesen und schreiben

Das IO-Model (damals):

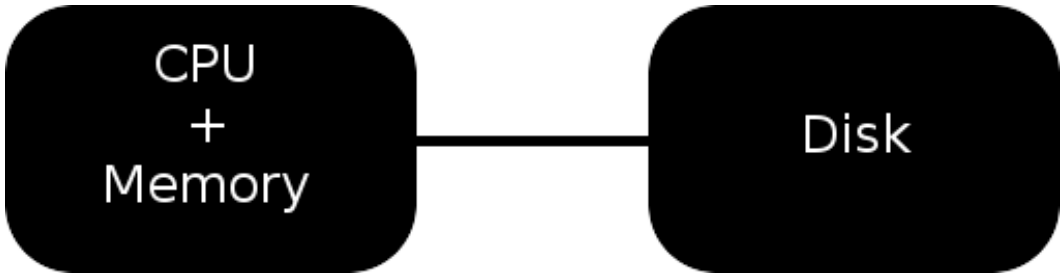


Sei hier M die Größe des RAMs.

Eigenschaften dieses Modells:

- Wir können Blöcke der Größe B lesen und schreiben
- Wir können max. M/B Blöcke vorhalten, absolute Kontrolle

Das IO-Model (damals):



Sei hier M die Größe des RAMs.

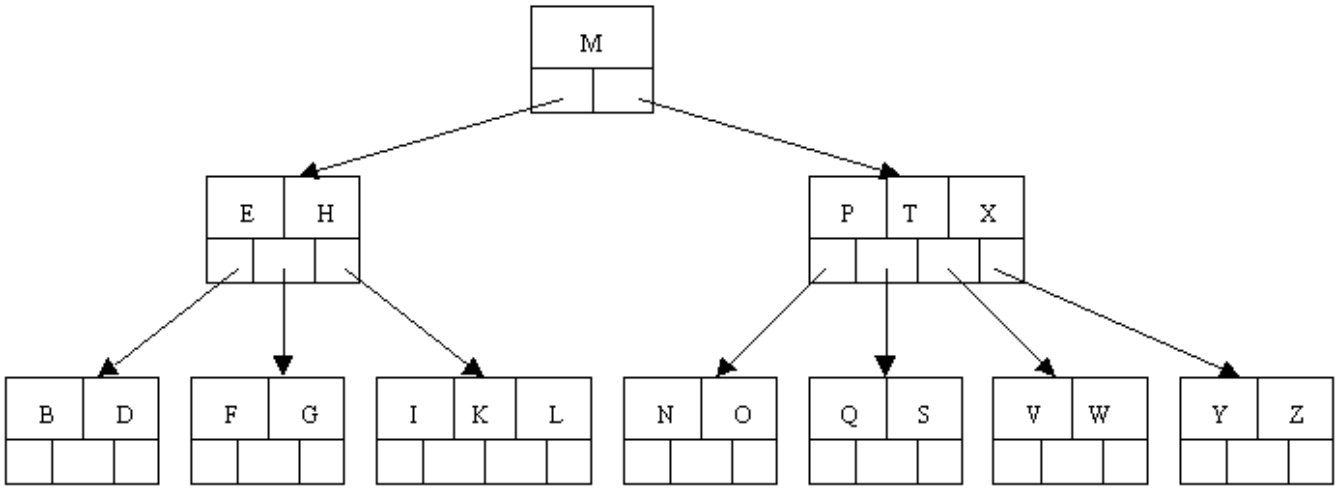
Eigenschaften dieses Modells:

- Wir können Blöcke der Größe B lesen und schreiben
- Wir können max. M/B Blöcke vorhalten, absolute Kontrolle
- Alle anderen Operationen sind „umsonst“

Dies erlaubt uns, *optimale* Datenstrukturen für bestimmte M zu finden, inklusive hübscher Asymptoten:

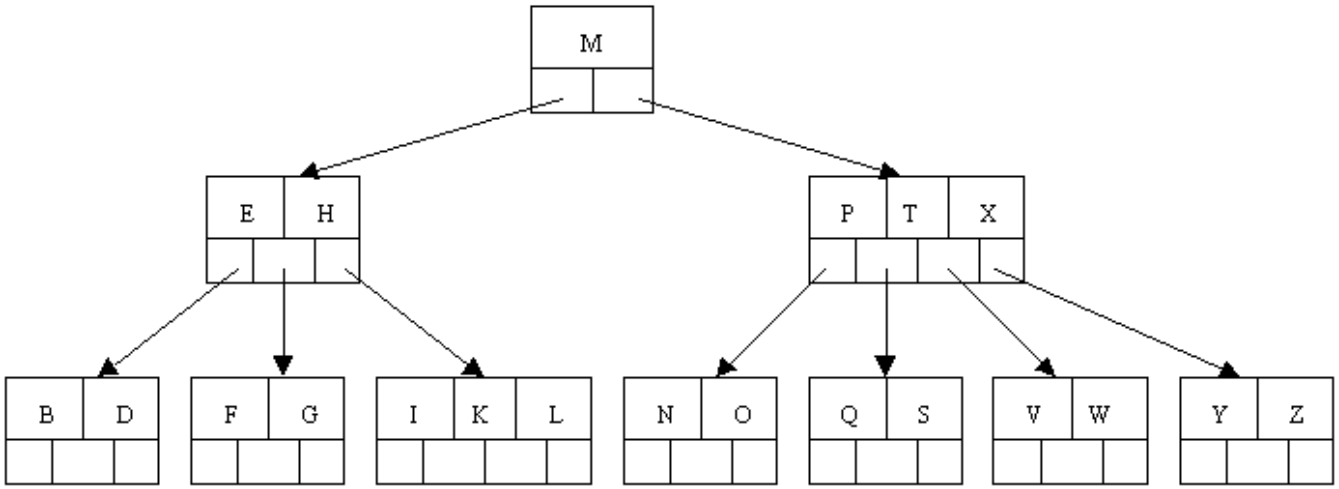
Dies erlaubt uns, *optimale* Datenstrukturen für bestimmte M zu finden, inklusive hübscher Asymptoten:

B-Trees (nicht zu verwechseln mit binary trees):



Dies erlaubt uns, *optimale* Datenstrukturen für bestimmte M zu finden, inklusive hübscher Asymptoten:

B-Trees (nicht zu verwechseln mit binary trees):



- Belegt $\mathcal{O}(\frac{N}{B})$ Blöcke Speicher
- Update möglich in $\mathcal{O}(\log \frac{N}{B})$
- Suche möglich in $\mathcal{O}(\log \left(\frac{N}{B}\right) + \frac{a}{B})$ wobei a Resultatgröße

Dieses Modell gibt uns zwar viel schönes, weil wir direkte Kontrolle über den einen Cache haben, es gibt aber zwei Probleme damit:

Dieses Modell gibt uns zwar viel schönes, weil wir direkte Kontrolle über den einen Cache haben, es gibt aber zwei Probleme damit:

- Wenn wir die Architektur wechseln müssen wir von vorne anfangen, unsere Konstanten abzustimmen.

Dieses Modell gibt uns zwar viel schönes, weil wir direkte Kontrolle über den einen Cache haben, es gibt aber zwei Probleme damit:

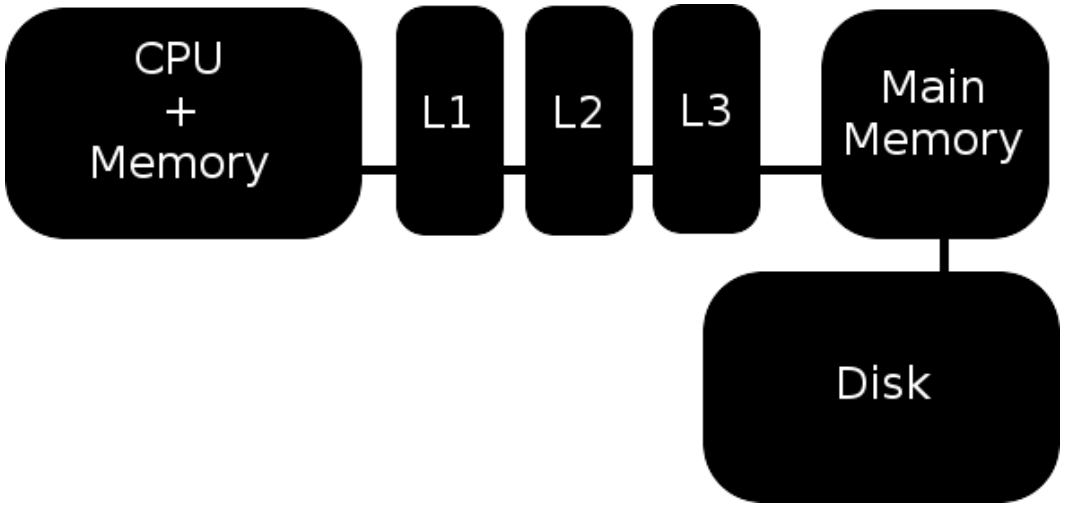
- Wenn wir die Architektur wechseln müssen wir von vorne anfangen, unsere Konstanten abzustimmen.
- Das ist nicht, wie heutige Rechner tatsächlich aussehen.

Dieses Modell gibt uns zwar viel schönes, weil wir direkte Kontrolle über den einen Cache haben, es gibt aber zwei Probleme damit:

- Wenn wir die Architektur wechseln müssen wir von vorne anfangen, unsere Konstanten abzustimmen.
- Das ist nicht, wie heutige Rechner tatsächlich aussehen.

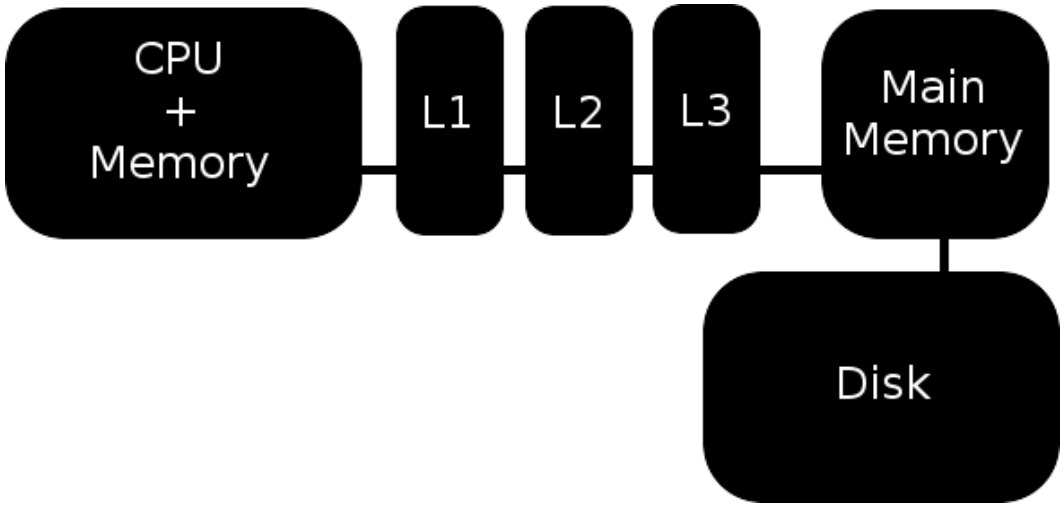
All models are wrong but some are useful. This one isn't.

Realistischer: Das IO-Model („gestern“):



Sei hier M wieder die Größe des RAMs. Oder besser M_1, M_2, ...

Realistischer: Das IO-Model („gestern“):

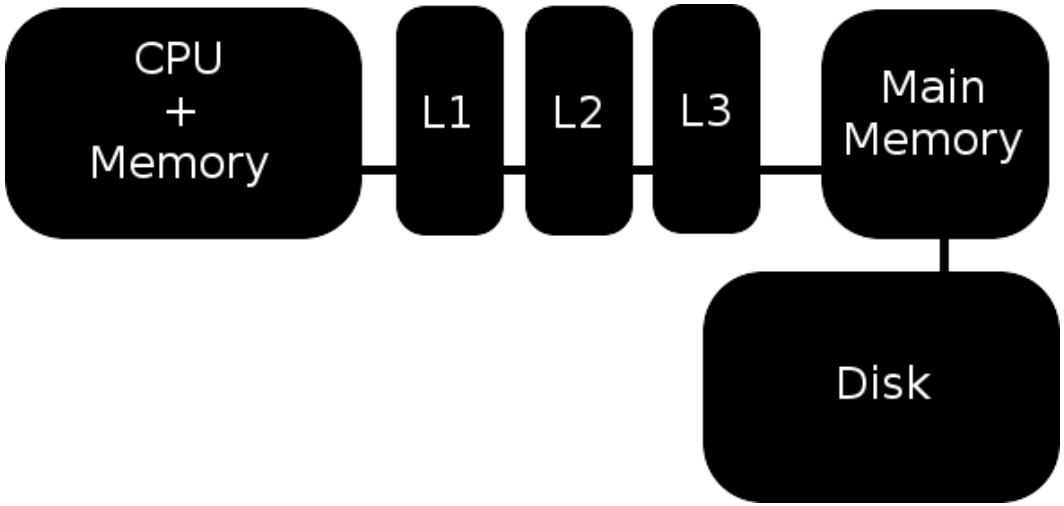


Sei hier M wieder die Größe des RAMs. Oder besser M_1, M_2, ...

Probleme:

- Jede Menge Konstanten abzustimmen, sehr viel Arbeit

Realistischer: Das IO-Model („gestern“):

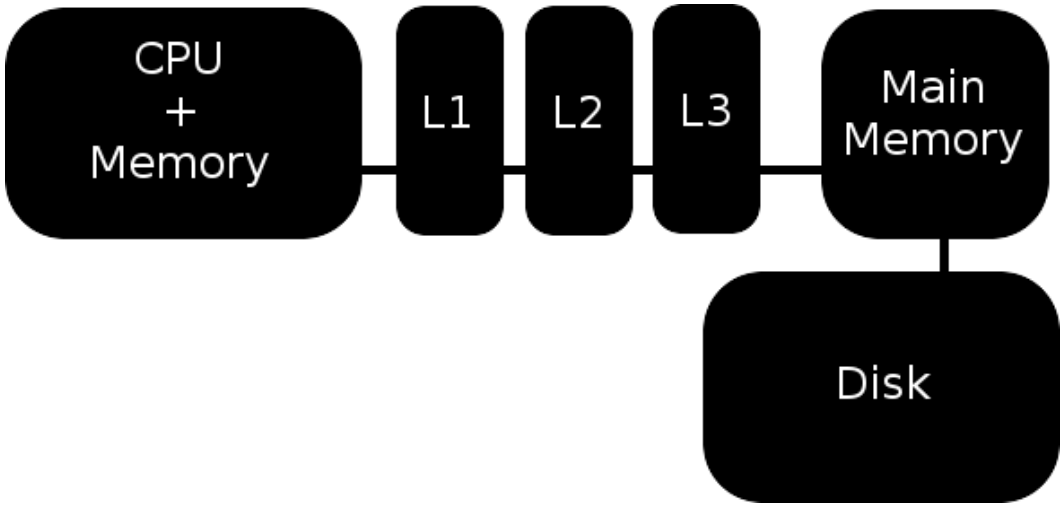


Sei hier M wieder die Größe des RAMs. Oder besser M_1, M_2, ...

Probleme:

- Jede Menge Konstanten abzustimmen, sehr viel Arbeit
- Optimierung für einen Cache suboptimiert für andere!

Realistischer: Das IO-Model („gestern“):

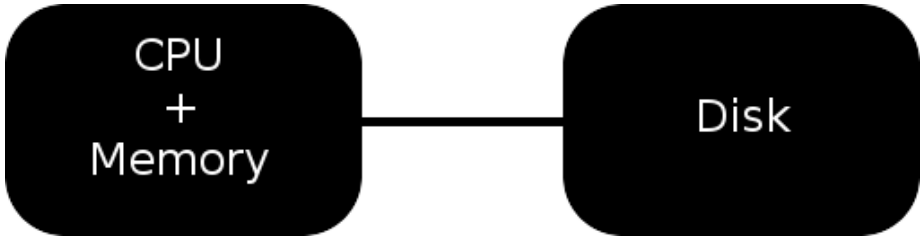


Sei hier M wieder die Größe des RAMs. Oder besser M_1, M_2, ...

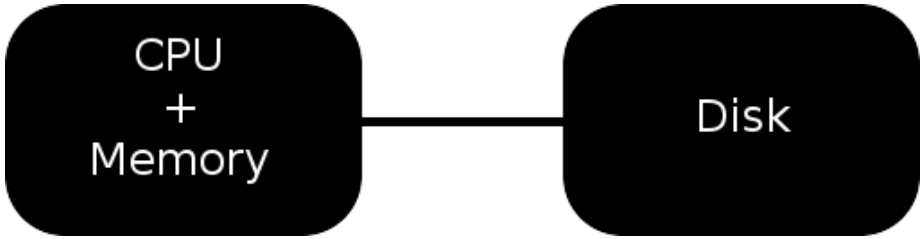
Probleme:

- Jede Menge Konstanten abzustimmen, sehr viel Arbeit
- Optimierung für einen Cache suboptimiert für andere!
- Fordert Unmengen an Gehirnpower und Fachwissen

Deswegen: Das Cache-Oblivious-Model (heute):



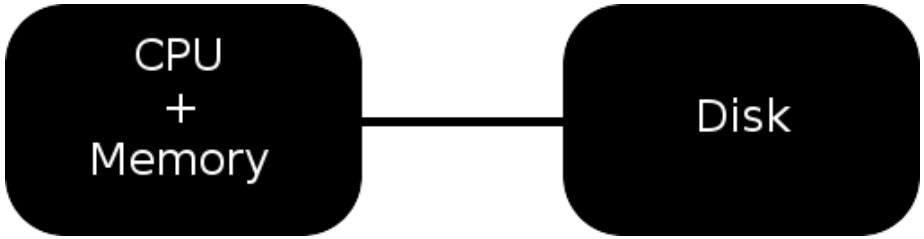
Deswegen: Das Cache-Oblivious-Model (heute):



Erstmal wie gehabt:

- Kann Blöcke der Größe B lesen und schreiben
- Kann $\frac{M}{B}$ Blöcke vorhalten
- Alle anderen Operationen sind „umsonst“

Deswegen: Das Cache-Oblivious-Model (heute):

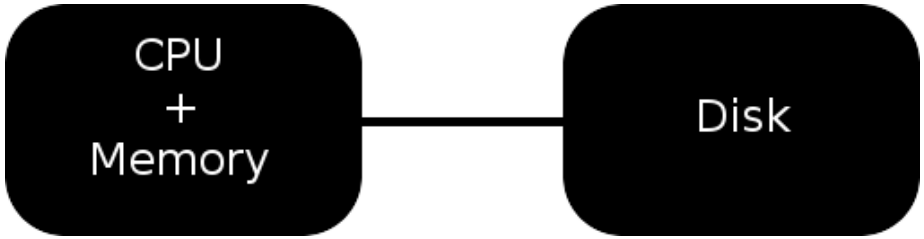


Erstmal wie gehabt:

- Kann Blöcke der Größe B lesen und schreiben
- Kann $\frac{M}{B}$ Blöcke vorhalten
- Alle anderen Operationen sind „umsonst“

ABER:
Wir kennen weder M, noch B!

Deswegen: Das Cache-Oblivious-Model (heute):



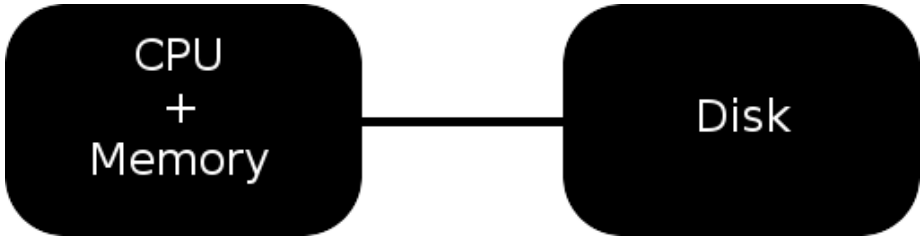
Erstmal wie gehabt:

- Kann Blöcke der Größe B lesen und schreiben
- Kann $\frac{M}{B}$ Blöcke vorhalten
- Alle anderen Operationen sind „umsonst“

ABER:
Wir kennen weder M, noch B!

- Asymp. optimale Alg. für unbekannte Größen sind asymp. optimal für *alle* Caches!

Deswegen: Das Cache-Oblivious-Model (heute):



Erstmal wie gehabt:

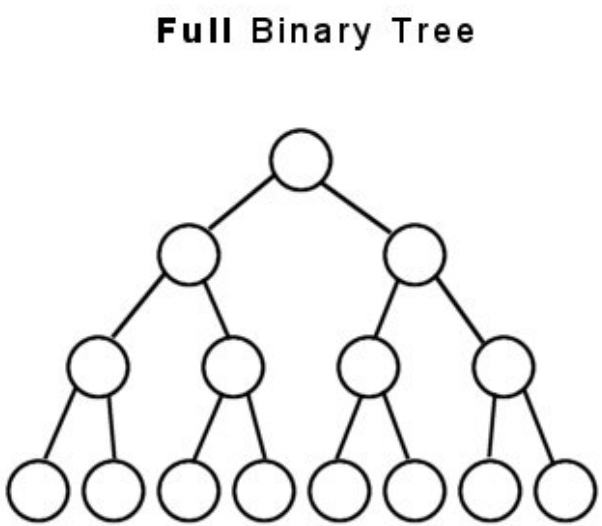
- Kann Blöcke der Größe B lesen und schreiben
- Kann $\frac{M}{B}$ Blöcke vorhalten
- Alle anderen Operationen sind „umsonst“

ABER:
Wir kennen weder M, noch B!

- Asymp. optimale Alg. für unbekannte Größen sind asymp. optimal für *alle* Caches!
- . . . gegeben ein Orakel mit perfekter *eviction policy*.

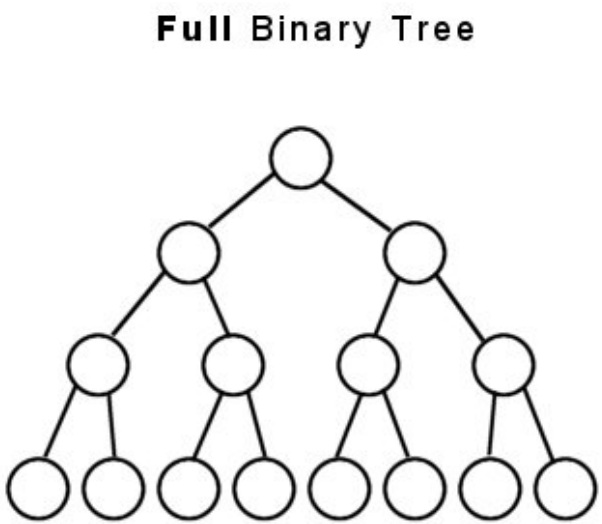
Cache-oblivious trees:

Ein gutes Beispiel dafür, dass optimale Datenstrukturen manchmal unintuitiv sein können, sind Bäume. Es wird gelehrt, dass Zugriffszeiten optimal sind, wenn Bäume perfekt gleich angeordnet werden:



Cache-oblivious trees:

Ein gutes Beispiel dafür, dass optimale Datenstrukturen manchmal unintuitiv sein können, sind Bäume. Es wird gelehrt, dass Zugriffszeiten optimal sind, wenn Bäume perfekt gleich angeordnet werden:



Aber auf echter Hardware kann es sinnvoller sein, z.B. drei Elemente in den linken und sieben in den rechten Teilbaum zu legen. So kann der linke Teilbaum im kleineren Cache gehalten werden, wenn ein gebalancter Baum zu groß wäre.

Vom Zahlensystem zur Datenstruktur

Bisher haben wir uns nur mit dem *Auslesen* unserer super-simplen Datenstruktur von vorhin beschäftigt. . . Aber was ist mit dem *Einfügen* von Daten?

Bisher haben wir uns nur mit dem *Auslesen* unserer super-simplen Datenstruktur von vorhin beschäftigt. . . Aber was ist mit dem *Einfügen* von Daten?

Netterweise haben uns Bentley und Saxe in 1980 bereits einen Weg gegeben (genannt das *Bentley-Saxe dynamisation scheme*):

Bisher haben wir uns nur mit dem *Auslesen* unserer super-simplen Datenstruktur von vorhin beschäftigt. . . Aber was ist mit dem *Einfügen* von Daten?

Netterweise haben uns Bentley und Saxe in 1980 bereits einen Weg gegeben (genannt das *Bentley-Saxe dynamisation scheme*):

- Man nehme: Verlinkte Liste einer beliebigen flachen Datenstruktur

Bisher haben wir uns nur mit dem *Auslesen* unserer super-simplen Datenstruktur von vorhin beschäftigt. . . Aber was ist mit dem *Einfügen* von Daten?

Netterweise haben uns Bentley und Saxe in 1980 bereits einen Weg gegeben (genannt das *Bentley-Saxe dynamisation scheme*):

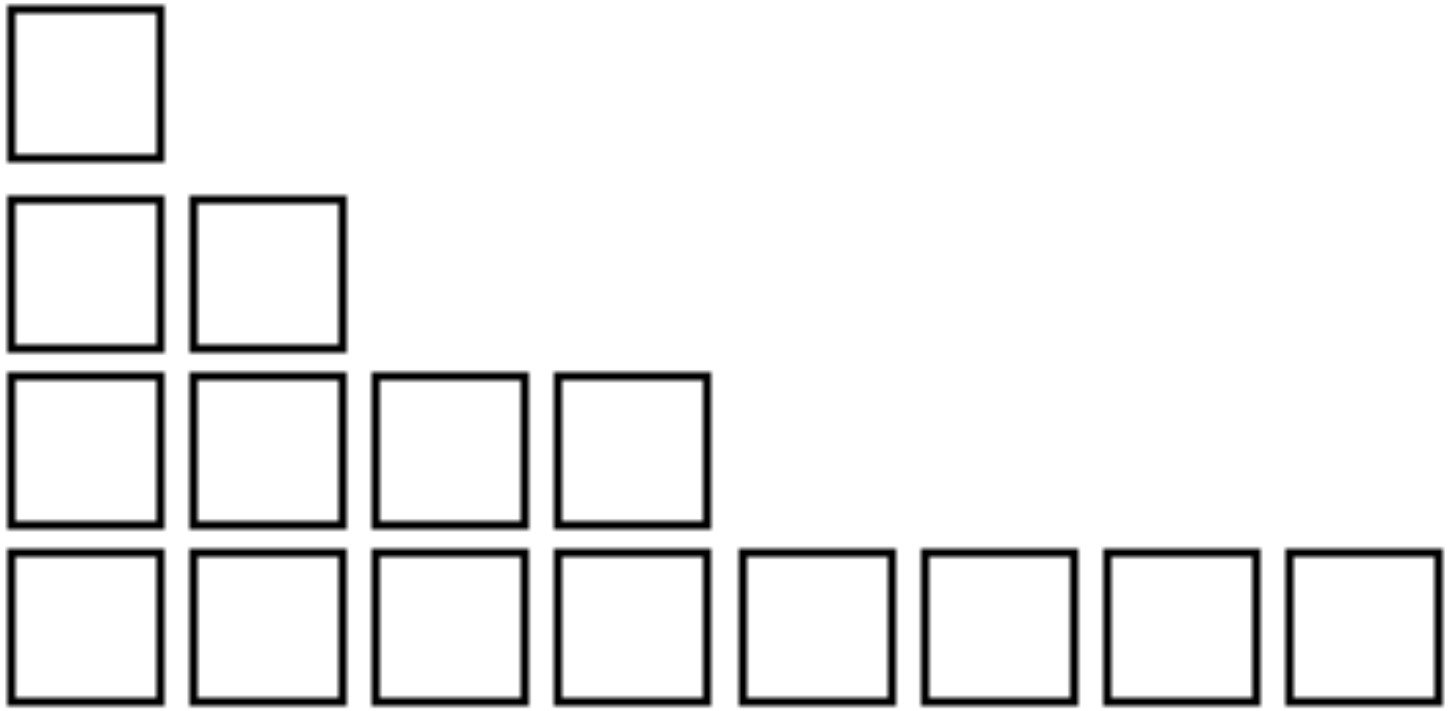
- Man nehme: Verlinkte Liste einer beliebigen flachen Datenstruktur
- Jedes Element hat Größe von aufsteigenden Zweierpotenzen

Bisher haben wir uns nur mit dem *Auslesen* unserer super-simplen Datenstruktur von vorhin beschäftigt. . . Aber was ist mit dem *Einfügen* von Daten?

Netterweise haben uns Bentley und Saxe in 1980 bereits einen Weg gegeben (genannt das *Bentley-Saxe dynamisation scheme*):

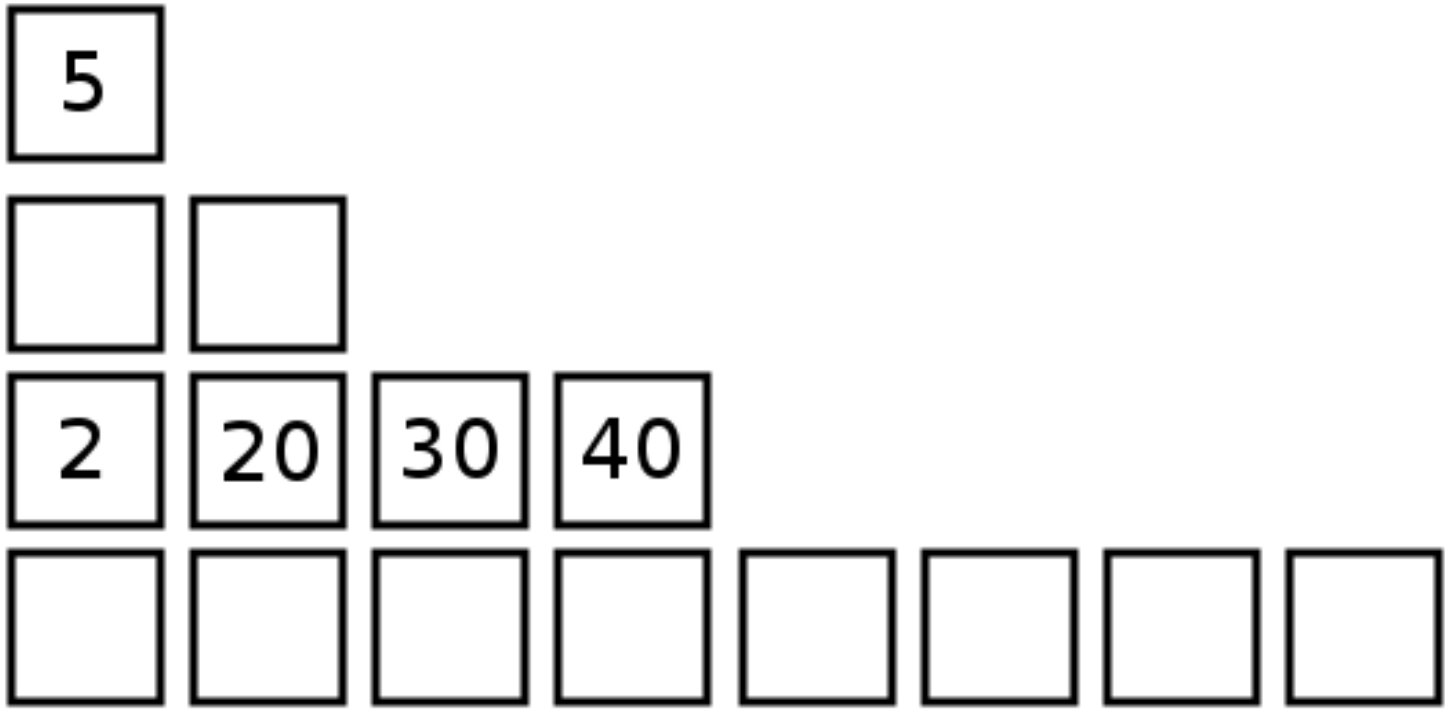
- Man nehme: Verlinkte Liste einer beliebigen flachen Datenstruktur
- Jedes Element hat Größe von aufsteigenden Zweierpotenzen
- Die Liste ist aufsteigend nach Größe sortiert

Bentley-Saxe:



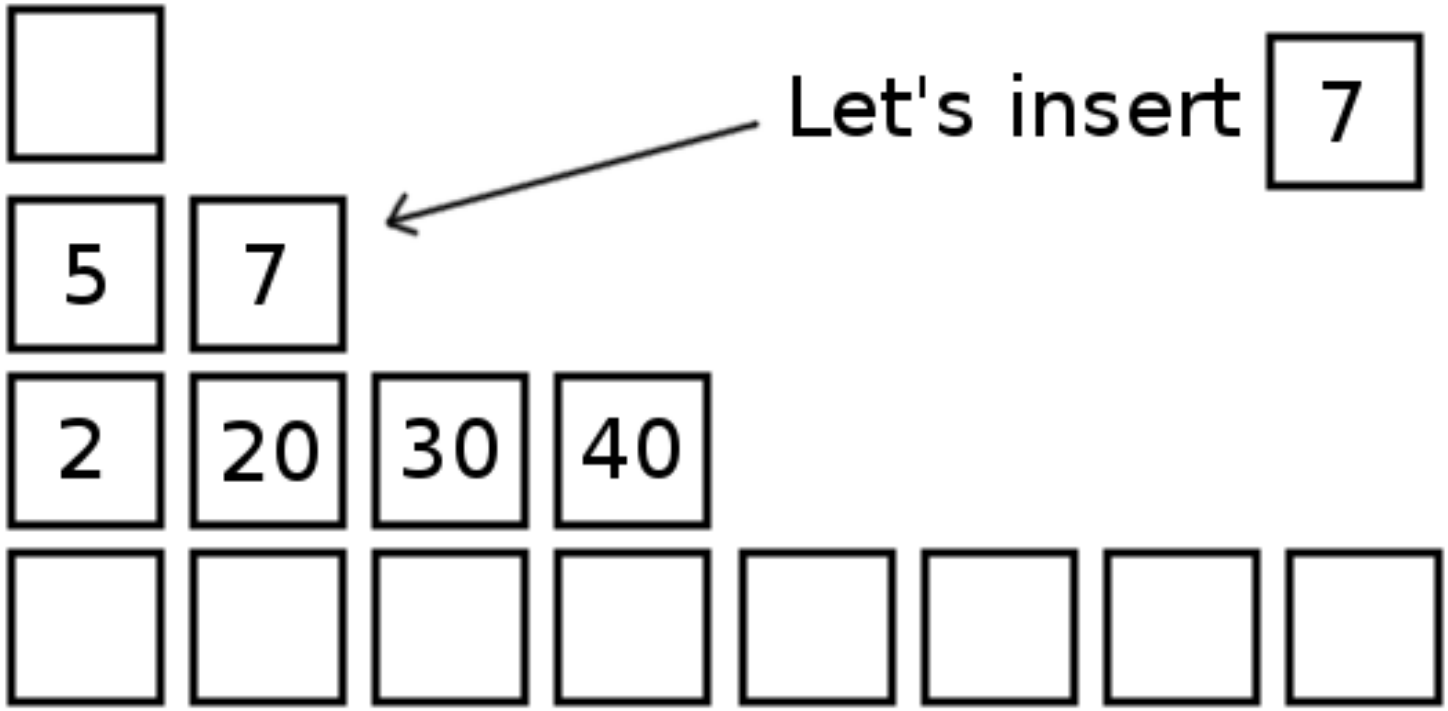
Wie oben besprochen...

Bentley-Saxe:



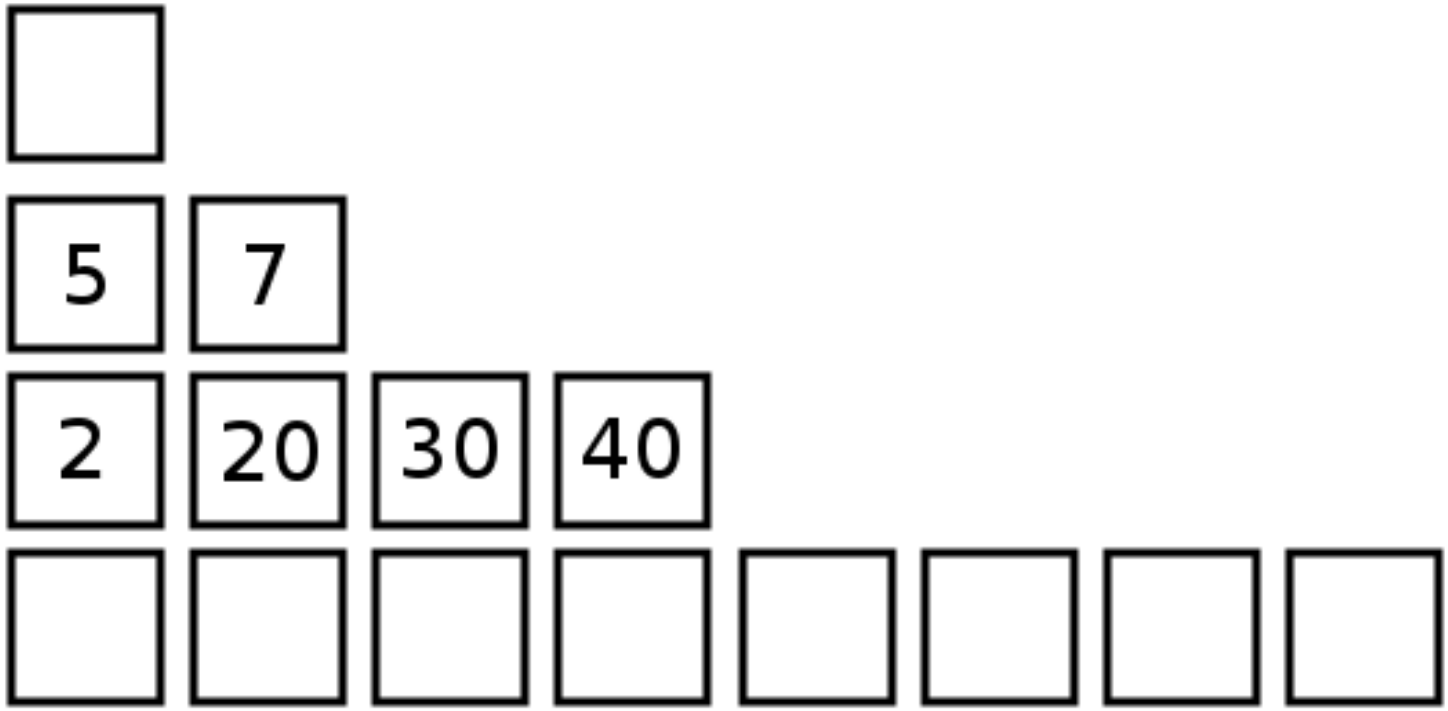
Jetzt gefüllt mit ein paar Daten.

Bentley-Saxe:



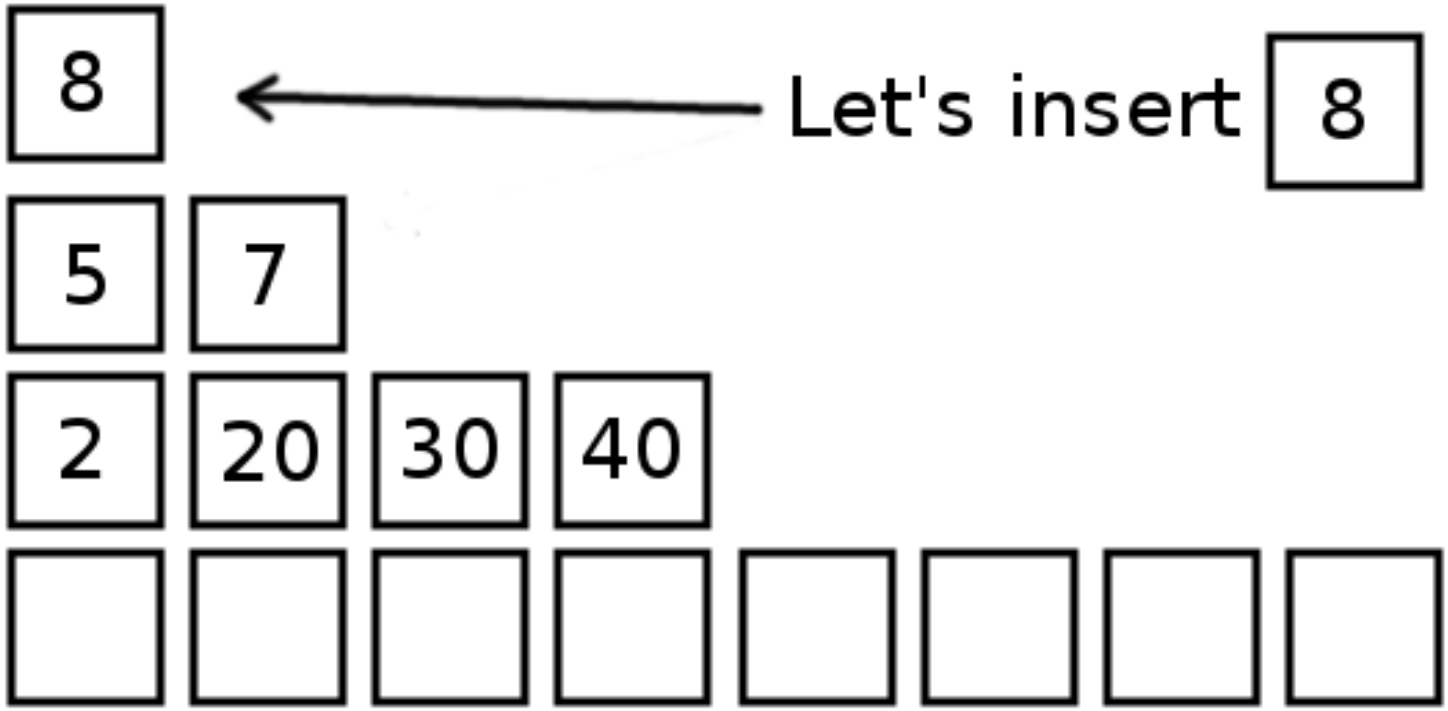
Wenn wir 7 einfügen, rutscht 5 in die größere Liste und merged.

Bentley-Saxe:



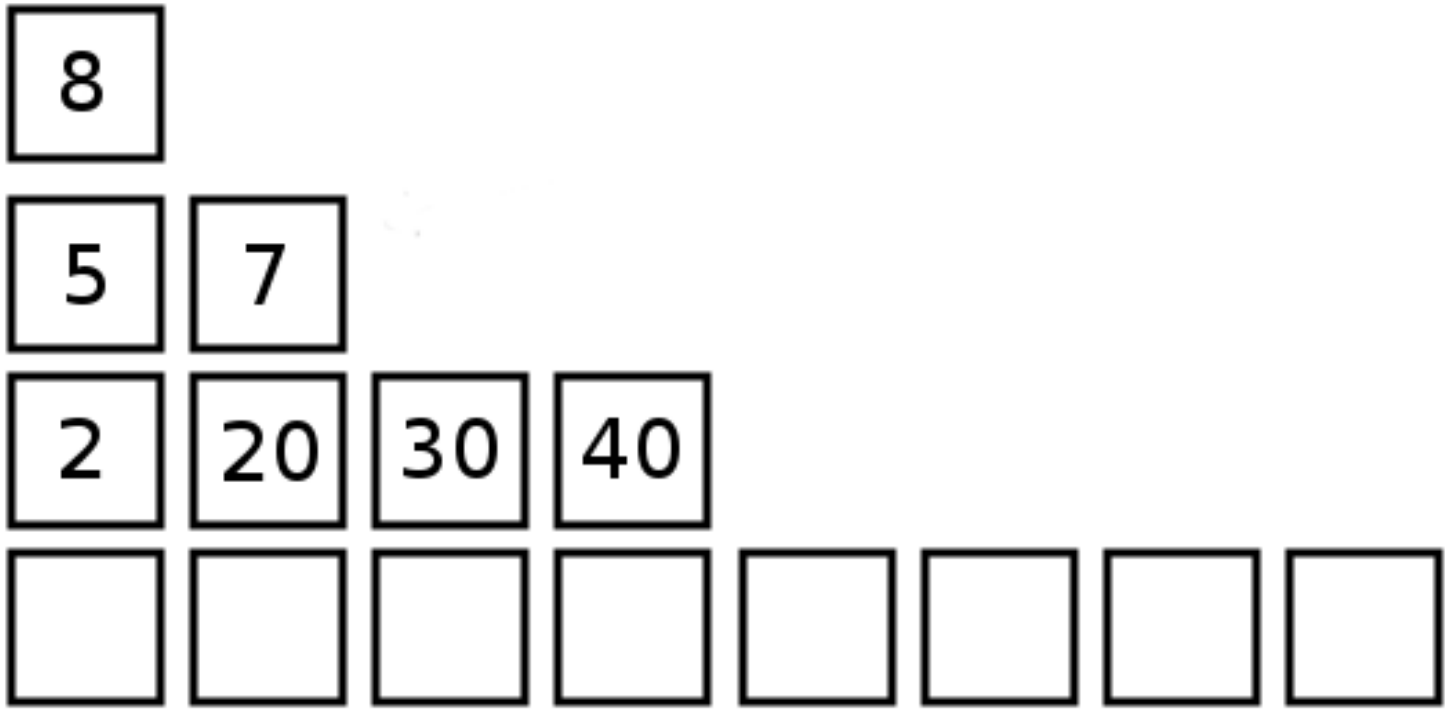
Wir haben außerdem „gezählt“. Von 101 zu 110

Bentley-Saxe:



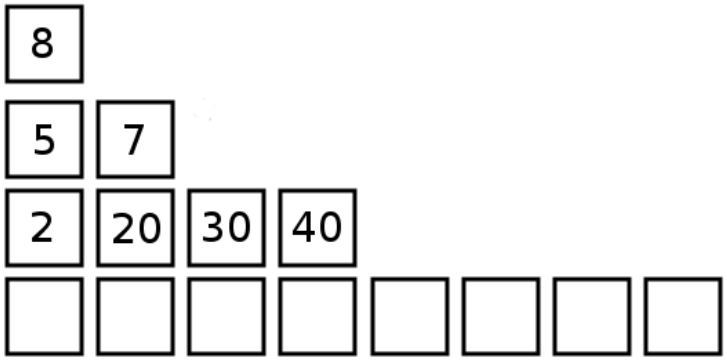
Dieser Insert benötigt keinen gesonderten Mergevorgang.

Bentley-Saxe:



Was gibt uns das für Asymptoten?

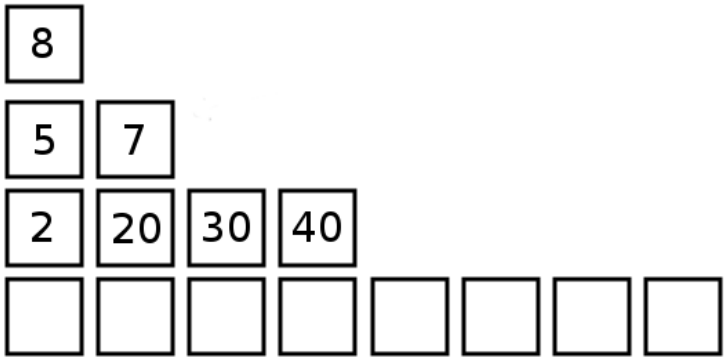
Bentley-Saxe:



Was gibt uns das für Asymptoten?

- worst-case insert liegt in $\mathcal{O}(\frac{N}{B})$

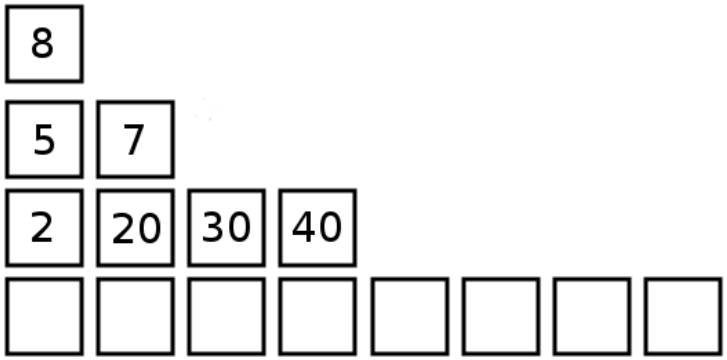
Bentley-Saxe:



Was gibt uns das für Asymptoten?

- worst-case insert liegt in $\mathcal{O}(\frac{N}{B})$
- amortisiertes insert liegt in $\mathcal{O}(\frac{\log N}{B})$

Bentley-Saxe:



Was gibt uns das für Asymptoten?

- worst-case insert liegt in $\mathcal{O}(\frac{N}{B})$
- amortisiertes insert liegt in $\mathcal{O}(\frac{\log N}{B})$

Das gleiche Ergebnis wie im „optimalen“ B-Tree, das allerdings *ohne B zu kennen!*

Sloppy and dysfunctional:

Chris Okasaki would not approve!

Sloppy and dysfunctional:

Chris Okasaki would not approve!

Unsere Datenstruktur basiert bisher auf dem herkömmlichen Binärsystem. Wenn uns ein langes Carry dazwischen kommt, müssen wir die ganze Datenstruktur neu aufbauen lassen, und wenn wir dann zu einer alten Version zurück kehren und was anderes inserten wollen, müssen wir das gleiche noch mal machen.

Sloppy and dysfunctional:

Chris Okasaki would not approve!

Unsere Datenstruktur basiert bisher auf dem herkömmlichen Binärsystem. Wenn uns ein langes Carry dazwischen kommt, müssen wir die ganze Datenstruktur neu aufbauen lassen, und wenn wir dann zu einer alten Version zurück kehren und was anderes inserten wollen, müssen wir das gleiche noch mal machen.

„We can't earn credit and spend it twice."

Sloppy and dysfunctional:

Chris Okasaki would not approve!

Unsere Datenstruktur basiert bisher auf dem herkömmlichen Binärsystem. Wenn uns ein langes Carry dazwischen kommt, müssen wir die ganze Datenstruktur neu aufbauen lassen, und wenn wir dann zu einer alten Version zurück kehren und was anderes inserten wollen, müssen wir das gleiche noch mal machen.

„We can't earn credit and spend it twice."

Können wir ein anderes Zahlensystem finden, dass besser auf unsere Aufgabe passt?

Binärsystem:

Dezimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010

Binärsystem:

Dezimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010

- Alle Ziffern sind 0 oder 1

Binärsystem:

Dezimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010

- Alle Ziffern sind 0 oder 1
- Stellenwerte sind zweierPotenzen $(2^0, 2^1, 2^2 \ldots)$

Binärsystem:

Dezimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010

- Alle Ziffern sind 0 oder 1
- Stellenwerte sind zweierPotenzen $(2^0, 2^1, 2^2 \ldots)$
- Bekannt und beliebt! :-)

Binärsystem:

Dezimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010

- Alle Ziffern sind 0 oder 1
- Stellenwerte sind zweierPotenzen $(2^0, 2^1, 2^2 \ldots)$
- Bekannt und beliebt! :-)
- . . . aber für unsere Zwecke nicht geeignet.

Zeroless Binary:

Dezimal	ZL Binary
0	000
1	001
2	002
3	011
4	012
5	021
6	022
7	111
8	112
9	121
10	122

Zeroless Binary:

Dezimal	ZL Binary
0	000
1	001
2	002
3	011
4	012
5	021
6	022
7	111
8	112
9	121
10	122

- Alle Ziffern sind 1 oder 2 (nur führende 0en erlaubt)

Zeroless Binary:

Dezimal	ZL Binary
0	000
1	001
2	002
3	011
4	012
5	021
6	022
7	111
8	112
9	121
10	122

- Alle Ziffern sind 1 oder 2 (nur führende 0en erlaubt)
- Stellenwerte aus dem Binärsystem ($2^0, 2^1, 2^2 \ldots$) werden beibehalten

Zeroless Binary:

Dezimal	ZL Binary
0	000
1	001
2	002
3	011
4	012
5	021
6	022
7	111
8	112
9	121
10	122

- Alle Ziffern sind 1 oder 2 (nur führende 0en erlaubt)
- Stellenwerte aus dem Binärsystem ($2^0, 2^1, 2^2 \ldots$) werden beibehalten
- Es existiert eine eindeutige Darstellung, Zahlen sind also unambiguitiv darstellbar

Zeroless Binary:

Dezimal	ZL Binary
0	000
1	001
2	002
3	011
4	012
5	021
6	022
7	111
8	112
9	121
10	122

- Alle Ziffern sind 1 oder 2 (nur führende 0en erlaubt)
- Stellenwerte aus dem Binärsystem ($2^0, 2^1, 2^2 \ldots$) werden beibehalten
- Es existiert eine eindeutige Darstellung, Zahlen sind also unambiguitiv darstellbar
- . . . und trotzdem noch immmer nicht das, was wir suchen.

Modified Zeroless Binary:

Dezimal	Modified ZLB
0	000
1	001
2	002
3	003
4	012
5	013
6	022
7	023
8	032
9	033
10	122

Modified Zeroless Binary:

Dezimal	Modified ZLB
0	000
1	001
2	002
3	003
4	012
5	013
6	022
7	023
8	032
9	033
10	122

- Alle Ziffern sind 1, 2 oder 3

Modified Zeroless Binary:

Dezimal	Modified ZLB
0	000
1	001
2	002
3	003
4	012
5	013
6	022
7	023
8	032
9	033
10	122

- Alle Ziffern sind 1, 2 oder 3
- Nur an vorderster Stelle darf eine 1 stehen

Modified Zeroless Binary:

Dezimal	Modified ZLB
0	000
1	001
2	002
3	003
4	012
5	013
6	022
7	023
8	032
9	033
10	122

- Alle Ziffern sind 1, 2 oder 3
- Nur an vorderster Stelle darf eine 1 stehen
- Stellenwerte aus dem Binärsystem ($2^0, 2^1, 2^2 \ldots$) werden beibehalten

Modified Zeroless Binary:

Dezimal	Modified ZLB
0	000
1	001
2	002
3	003
4	012
5	013
6	022
7	023
8	032
9	033
10	122

- Alle Ziffern sind 1, 2 oder 3
- Nur an vorderster Stelle darf eine 1 stehen
- Stellenwerte aus dem Binärsystem ($2^0, 2^1, 2^2 \ldots$) werden beibehalten
- Es existiert eine eindeutige Darstellung

Modified Zeroless Binary:

Dezimal	Modified ZLB
0	000
1	001
2	002
3	003
4	012
5	013
6	022
7	023
8	032
9	033
10	122

- Alle Ziffern sind 1, 2 oder 3
- Nur an vorderster Stelle darf eine 1 stehen
- Stellenwerte aus dem Binärsystem ($2^0, 2^1, 2^2 \ldots$) werden beibehalten
- Es existiert eine eindeutige Darstellung
- Hat genau die richtige Menge an Verzögerung!

Vergleich der Zahlensysteme:

Dezimal	Binary	ZL Binary	Modified ZLB
0	0000	000	000
1	0001	001	001
2	0010	002	002
3	0011	011	003
4	0100	012	012
5	0101	021	013
6	0110	022	022
7	0111	111	023
8	1000	112	032
9	1001	121	033
10	1010	122	122

Why the *bleep* do we care?!

Why the *bleep* do we care?!

Because now we have this:

```
data Map k a
  = M0
  | M1 !(Chunk k a)
  | M2 !(Chunk k a) !(Chunk k a) ~(Chunk k a) !(Map k a)
  | M3 !(Chunk k a) !(Chunk k a) !(Chunk k a) ~(Chunk k a) !(Map k a)

data Chunk k a = Chunk !(Array k) !(Array a)

-- O(log(N)/B) persistently amortised. insert an element.
insert :: (Ord k, Arrayed k, Arrayed v) => k -> v -> Map k v -> Map k v
insert k0 v0 = g0 $ Chunk (singleton k0) (singleton v0) where
  go as M0                    = M1 as
  go as (M1 bs)               = M2 as bs (merge as bs) M0
  go as (M2 bs cs bcs xs)     = M3 as bs cs bcs xs
  go as (M3 bs _ _ cds xs) = cds `seq` M2 as bs (merge as bs) (go cds xs)
{-# INLINE insert #-}
```

Why the *bleep* do we care?! (2)

Why the *bleep* do we care?! (2)

- `insert` ist jetzt 7-10x schneller als Äquivalent aus `Data.Map` und wird beim Skalieren nur schneller

Why the *bleep* do we care?! (2)

- `insert` ist jetzt 7-10x schneller als Äquivalent aus `Data.Map` und wird beim Skalieren nur schneller
- Wir können eine unboxed map bauen, wenn wir unboxed Datentypen reinstecken.

Why the *bleep* do we care?! (2)

- `insert` ist jetzt 7-10x schneller als Äquivalent aus `Data.Map` und wird beim Skalieren nur schneller
- Wir können eine unboxed map bauen, wenn wir unboxed Datentypen reinstecken.
- Asymp. *B*-Tree performance ohne *B* kennen zu müssen.

Why the *bleep* do we care?! (2)

- `insert` ist jetzt 7-10x schneller als Äquivalent aus `Data.Map` und wird beim Skalieren nur schneller
- Wir können eine unboxed map bauen, wenn wir unboxed Datentypen reinstecken.
- Asymp. *B*-Tree performance ohne *B* kennen zu müssen.
- Keine Konstanten, die wir feinabstimmen müssten.