VERIFIZIERTE IMPLEMENTIERUNG VON PATRICIA-BÄUMEN

TIM HABERMAAS

Studienarbeit

Fakultät für Informatik
Karlsruher Institut für Technologie

Januar 2013

GUTACHTER:
Prof. Dr.-Ing. Gregor Snelting

BETREUENDE MITARBEITER:
Dr. Andreas Lochbihler
Dipl.-Inform. Denis Lohner

Tim Habermaas: *Verifizierte Implementierung von Patricia-Bäumen,* Studienarbeit, © Januar 2013

ABSTRACT

Die Aufgabe dieser Studienarbeit bestand darin eine Map mit ganzzahligen Schlüsseln beliebiger Länge zu implementieren. Dabei sollte die Map mit Hilfe von Patricia-Bäumen implementiert und ihre Korrektheit mit dem Theorembeweiser Isabelle/HOL bewiesen werden.

INHALTSVERZEICHNIS

1 EINFÜHRUNG 1
  1.1 Verwandte Arbeiten 1
2 PATRICIA-BÄUME 3
  2.1 Definition 3
  2.2 Patricia-Bäume für Maps mit ganzzahligen Schlüsseln 4
  2.3 Aufbau des Baumes 4
  2.4 Operationen 5
    2.4.1 Nachschlagen 5
    2.4.2 Einfügen 6
    2.4.3 Vereinigen 8
  2.5 Besonderheiten bei der Verwendung von Ganzzahlen beliebiger Bitlänge 10
  2.6 Big-Endian vs. Little-Endian 11
3 VERIFIKATION 13
  3.1 Einfügen 13
  3.2 Vereinigen 13
  3.3 Invarianten 14
    3.3.1 Valide Präfixe 14
    3.3.2 Negative und positive Schlüssel müssen getrennt werden 15
    3.3.3 Jeder innere Knoten besitzt genau zwei Kinder 16
  3.4 Weitere Invarianten 16
    3.4.1 Schlüssel passen zur Maske 17
    3.4.2 Maske ist Zweierpotenz 17
4 EVALUIERUNG 19
  4.1 Performance 19
    4.1.1 Setup 19
    4.1.2 Nachschlagen 19
    4.1.3 Einfügen 21
    4.1.4 Vereinigen 22
    4.1.5 Weitere Tests und Auffälligkeiten 22
    4.1.6 Fazit 23
5 FAZIT UND AUSBLICK 25
  5.1 Fazit 25
  5.2 Ausblick 25
  5.3 Statistiken 26

LITERATURVERZEICHNIS 27

ABBILDUNGSVERZEICHNIS

Abbildung 1 Präfixbaum und Patricia-Baum 3
Abbildung 2 Beispiel eines Patricia-Baumes 4
Abbildung 3 Entarteter Patricia-Baum 6
Abbildung 4 Einfügen in einen Patricia-Baum 8
Abbildung 5 Little-Endian Patricia-Baum 11
Abbildung 6 Baum t vor dem Einfügen von $(1, z)$ 15
Abbildung 7 Baum t' nach dem Einfügen von $(1, y)$ 15
Abbildung 8 Unterschiedliche Vorzeichen im selben Baum 16
Abbildung 9 Gegenbeispiel branchingBitDecides 17
Abbildung 10 Performance Nachschlagen sequentiell 20
Abbildung 11 Performance Nachschlagen randomisiert 20
Abbildung 12 Performance sequentielles Einfügen 21
Abbildung 13 Performance zufälliges Einfügen 21
Abbildung 14 Benchmark-Ergebnisse Vereinigen sequentiell 22
Abbildung 15 Performance Vereinigen zufällig 22
Abbildung 16 Performance Einfügen in umgekehrte Reihenfolge 24

TABELLENVERZEICHNIS

Tabelle 1 Benchmark-Ergebnisse 23

LISTINGS

Listing 1 Datentyp Patricia-Baum 5
Listing 2 Typ-Synonyme 5
Listing 3 Nachschlagen 5
Listing 4 zeroBit 6
Listing 5 mask 7
Listing 6 highestBit 7
Listing 7 join 8
Listing 8 Einfügen 8
Listing 9 Vereinigen 9
Listing 10 Paar aus Patricia-Bäumen 11

Listing 11 Nachschlagen in dem Tupel 11
Listing 12 lowestBit 11

1 EINFÜHRUNG

Im Compilerbau tauchen Maps als Datenstrukturen sehr häufig auf. Sie werden zum Beispiel für Symboltabellen wie auch bei der Datenflussanalyse benötigt. Zusätzlich muss man während des Kompilierens immer wieder rekursiv diese Maps über Teilbäume berechnen und sie dann vereinigen. Deswegen ist ein effizientes Vereinigen ebenso wichtig wie ein effizientes Einfügen.

Nutzt man eine der bekannten Implementierungen für Maps – balancierte Bäume oder Hash-Tabellen – stellt man schnell fest, dass das Vereinigen sehr teuer ist (bis zu $\mathcal{O}(n \log(n))$).

Patricia-Bäume sind eine Datenstruktur aus dem Jahre 1968 [6] und ermöglichen ein deutlich effizienteres Vereinigen von Maps. *Chris Okasaki* und *Andy Gill* haben diese alte Datenstruktur wieder entdeckt und damit eine endliche Map mit ganzzahligen Schlüsseln implementiert. [4] Okasaki und Gill beschränkten sich in ihrer Arbeit auf Schlüssel fester Bitbreite. In dieser Arbeit habe ich die Patricia-Bäume auf Schlüssel beliebiger Bitbreite erweitert, damit sie mit dem $\mathtt{int}$-Datentyp von *Isabelle/HOL* [9] zusammenarbeiten können. Dabei diente *Isabelle/HOL* sowohl als Implementierungssprache als auch als Theorembeweiser um die Korrektheit dieser Map zu zeigen.

In Kapitel 2 stelle ich die so erweiterten Patricia-Bäume vor und gehe auf die Implementierungsdetails ein. Insbesondere sind in Abschnitt 2.5 die notwendingen Erweiterungen gegenüber dem Patricia-Baum von *Okasaki* und *Gill* zu finden. Anschließend beschäftige ich mich in Kapitel 3 mit der Korrektheit der Implementierung und stelle die notwendigen Invarianten (Abschnitt 3.3) vor, die für die *Korrektheitslemmas* notwendig sind. Danach prüfe ich in Kapitel 4 noch inwiefern meine Implementierung des Patricia-Baumes den Behauptungen von einem schnellen Vereinigen standhält. Schlussendlich nenne ich in Kapitel 5 noch verschiedene Punkte wie die Implementierung oder die Beweise verbessert werden können.

1.1 VERWANDTE ARBEITEN

In der Haskell-Standard-Bibliothek [1] findet man bereits eine Implementierung der Patricia-Bäume, wie sie von *Okasaki* und *Gill* vorgestellt wurden. Diese nutzen als Schlüssel aber den Ganzzahl-Datentyp $\mathtt{Int}$ aus Haskell mit fester Bitbreite. Für Java findet man eine ähnliche Implementierung [3]. Im Gegensatz zu dieser Arbeit können beide Implementierungen aber nur mit Schlüssel fester Länge umgehen.

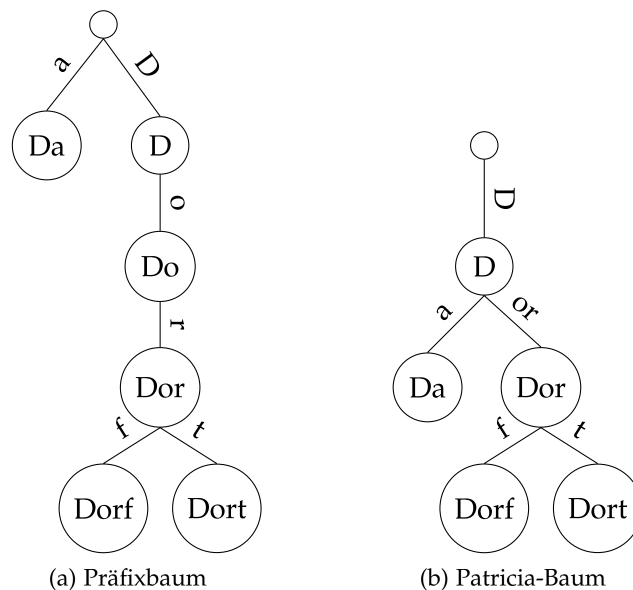
Sucht man dagegen nach Maps mit beliebig langen Schlüsseln, die nicht auf Patricia-Bäumen aufbauen, so wird man in fast jeder modernen Programmiersprache, die Datentypen für Zahlen beliebiger Länge bereitstellt, fündig. Unter anderem existiert in *Isabelle/HOL* selbst eine Map basierend auf einem Rot-Schwarz-Baum. [5]

2 PATRICIA-BÄUME

2.1 DEFINITION

Patricia-Bäume sind komprimierte Präfixbäume bei denen innere Knoten, die nur ein Kind besitzen, vereinigt werden.

PRÄFIXBAUM Mit Hilfe eines Präfixbaumes können Schlüssel bestehend aus Zeichenketten gespeichert werden. Dabei stehen die Blätter des Baumes für die zu speichernden Zeichenketten. In den Blättern können zusätzlich noch weitere Informationen gespeichert werden – in etwa Werte zu den Schlüsseln. Die Kanten an jedem inneren Knoten repräsentieren die einzelnen Buchstaben, die zu den gespeicherten Zeichenketten führen. Abbildung 1a zeigt einen Präfixbaum, der die Zeichenketten „Da“, „Dorf“ und „Dort“ enthält. Dabei teilen sich „Dorf“ und „Dort“ für die ersten drei Zeichen den rechten Unterbaum.



(a) Präfixbaum (b) Patricia-Baum

Abbildung 1: Präfixbaum und Patricia-Baum

UNTERSCHIED ZWISCHEN PRÄFIX- UND PATRICIA-BAUM Wie in Abbildung 1a entstehen beim Präfixbaum schnell lange Ketten von Knoten bei denen jeder maximal ein Kind besitzt. Diese Ketten kann man zu einem Knoten zusammenfassen, indem man die Buchstaben konkateniert und die resultierende Zeichenkette in dem neuen Kno-

ten speichert – wie in Abbildung 1b geschehen. Die so komprimierten Präfixbäume heißen Patricia-Bäume.

NAME Dieser Name stammt von Morrison, der die Bäume in „Pratical Algorithm to Retrieve Information Coded in Alphanumeric“[6] – abgekürzt mit **PATRICIA** – im Jahre 1968 vorgestellt hat.

2.2 PATRICIA-BÄUME FÜR MAPS MIT GANZZAHLIGEN SCHLÜSSELN

Beschränkt man sich bei dem Alphabet auf die Menge $\{0, 1\}$, so lässt sich mit Hilfe von Patricia-Bäumen eine Map mit ganzzahligen Schlüsseln realisieren. Durch diesen Spezialfall fallen beim Suchen Vergleiche mit Präfixen weg und man kann vieles auf Bit-Operatoren runterbrechen. „Ganzzahlige Schlüssel“ ist allerdings erst einmal nur die halbe Wahrheit. Im Weiteren beschränke ich mich zunächst auf Schlüssel größer oder gleich 0, da negative Zahlen nicht ohne Ausnahme zu behandeln sind. In Abschnitt 2.5 folgen dann auch negative Schlüssel.

2.3 AUFBAU DES BAUMES

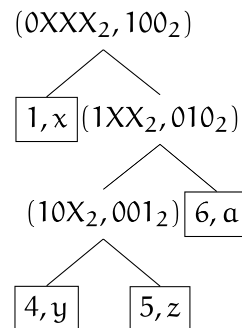


Abbildung 2: Patricia-Baum mit den Schlüsseln 1, 4, 5 und 6

Die Schlüssel-Wert-Paare der Map werden direkt in den *Blättern* gespeichert. In Abbildung 2 ist zum Beispiel dem Schlüssel 1 der Wert x zugeordnet.

Die *inneren Knoten* speichern wie in Abbildung 1b ein *Präfix* p. Um Patricia-Bäume effizient implementieren zu können, wird das Präfix als ganze Zahl abgespeichert – und nicht etwa als Array über dem zweielementigen Alphabet $0, 1$. Dies hat zur Folge, dass man nicht mehr zwischen den Präfixen $0\ 1$ und 1 unterscheiden kann, da beide die Binärdarstellung 01_2 besitzen. Um diesen Missstand zu beseitigen, speichert man zusätzlich das Bit ab, ab dem man das Präfix abschneidet. Dieses wird als Zweierpotenz gespeichert und heißt *Maske*

m. Besitzt ein innerer Knoten t zum Beispiel das *Präfix* $0\dots010X_2$, wobei X für ein beliebiges Bit steht, und die *Maske* $0\dots01_2$, so beginnen alle Blätter der beiden Unterbäume von t mit $0\dots010_2$. (siehe niedrigster innerer Knoten in Abbildung 2). Ob sich ein Schlüssel im rechten oder linken Unterbaum von t befindet, entscheidet dabei die Maske. Ist das Bit eines Schlüssels an der $\ddot{m}$-ten Stelle gesetzt, befindet er sich im rechten, ansonsten im linken Unterbaum.

$\ddot{m} = \log_2(m) + 1$

Um die Map darzustellen, die keinen Schlüssel enthält, nimmt man noch den leeren Baum *Empty* hinzu. Somit ergibt sich der Datentyp in Isabelle/HOL wie folgt:

Listing 1: Datentyp Patricia-Baum

```
datatype 'a PatriciaTree =
  Empty |
  Lf key 'a |
  Br prefix branchingBit "'a patriciaTree" "'a patriciaTree"
```

dabei sind key, prefix und branchingBit Typ-Aliase für int aus Isabelle/HOL:

Listing 2: Typ-Synonyme

```
type_synonym key          = "int"
type_synonym prefix       = "int"
type_synonym branchingBit = "int"
```

2.4 OPERATIONEN

Im folgenden stelle ich die Operationen *Nachschlagen*, *Einfügen* und *Vereinigen* auf diesem Datentyp vor.

2.4.1 *Nachschlagen*

Um einen passenden Wert zu einem Schlüssel k zu finden, fängt man an der Wurzel an zu suchen. Dabei betrachtet man die Maske m des inneren Knotens und den Schlüssel k und sucht im linken Unterbaum weiter, falls das $\ddot{m}$-te Bit von k nicht gesetzt ist. Ansonsten sucht man rechts weiter.

Ist man bei einem Blatt angekommen, muss man noch den Schlüssel des Blatts mit k auf Gleichheit testen.

In Isabelle/HOL sieht das folgendermaßen aus:

Listing 3: Nachschlagen

```
fun lookup' :: "key ⇒ 'a patriciaTree ⇒ 'a option" where
  "lookup' k Empty = None" |
  "lookup' k (Lf key value) = (if k=key then Some value else None
      )" |
```

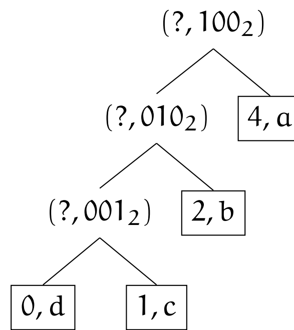


Abbildung 3: Entarteter Patricia-Baum mit den Schlüsseln 0, 1, 2 und 4

```
  "lookup' k (Br p m left right) = (if zeroBit k m then lookup' k
      left else lookup' k right)"
```

Dabei überprüft *zeroBit* mittels bitweiser Und-Verknüpfung, ob k an der Stelle $\dot{m}$ nicht gesetzt ist.

Listing 4: zeroBit

```
abbreviation zeroBit :: "int ⇒ int ⇒ bool" where
  "zeroBit k m ≡ ((k AND m) = 0)"
```

LAUFZEIT Die Worst-Case-Laufzeit von *Nachschlagen* beträgt $\mathcal{O}(min(n, W))$ [4], wobei n für die Anzahl der Blätter steht und W die maximale Anzahl an Bits bezeichnet, die im Baum verwendet werden. Für gewöhnlich ist W deutlich kleiner als n, aber im schlechtesten Fall – siehe Abbildung 3 – entsteht ein Baum, bei dem $n = W + 1$ gilt.

2.4.2 *Einfügen*

Einfügen eines Schlüssel-Wert-Paares (k, v) in die Map M lässt sich wie folgt beschreiben:

$$M' = \{(j, w) : j \in keys(M) \land k \neq j\} \cup \{(k, f(v, w)) : (k, w) \in M\}$$

wobei

$$keys(M) = \{k : (k, v) \in M\}$$

Dabei dient die Funktion f zum Auflösen der Konflikte, falls k bereits in M zu finden war.

Um (k, v) in den zur Map M gehörigen Patricia-Baum einzufügen, sucht man rekursiv von der Wurzel ab nach

- einem Blattknoten, dessen Schlüssel j ungleich k ist, oder
- einem inneren Knoten, dessen Präfix p nicht mit k übereinstimmt.

Dabei stimmt ein Präfix nicht mit einem Schlüssel k überein, falls gilt:

$$\mathsf{mask}\ k\ m \neq p$$

wobei mask alle Bits rechts vom $\breve{m}$-ten Bit zu 1 macht und das $\breve{m}$-te Bit selbst zu 0. Es wäre auch jede andere Bit-Kombination ab dem $\breve{m}$-ten Bit vorstellbar, solange man mask konsistent verwendet. Durch Präfixe der Form $X \dots X01 \dots 1$ lässt sich aber der $\mathsf{zeroBit}$-Vergleich bei lookup durch einen Kleiner-als-Test mit p ersetzen, was je nach Prozessor-Architektur schnellere Ergebnisse liefern kann. [4]

Listing 5: mask

```
abbreviation mask :: "key ⇒ branchingBit ⇒ int" where
  "mask k m = (k OR m - 1) AND (NOT m)"
```

Unabhängig davon, ob *Einfügen* an einem Blatt- oder einem inneren Knoten angekommen ist, werden der gefundene Baum und das einzufügende Schlüssel-Wert-Paar (k, v) mit join zu einem neuen Knoten vereinigt.

JOIN erstellt aus zwei Bäumen t_1 und t_2, deren Präfixe p_1 und p_2 sich unterscheiden, einen neuen inneren Knoten, der t_1 und t_2 als Unterbäume besitzt. Dabei legt das *höchste Bit*, in dem sich p_1 und p_2 unterscheiden, die Maske m fest. Das neue Präfix p ist dann das gemeinsame Präfix von p_1 und p_2.

Um dieses höchste Bit, in dem sich p_1 und p_2 unterscheiden zu berechnen, berechnet man zunächst die bitweise XOR-Verknüpfung p_x. Somit erhält man alle Bits, in denen sich p_1 und p_2 unterscheiden. Das höchste Bit von p_x ist dann das gesuchte Bit. Zum Beispiel gilt mit $p_0 = 0100_2$ und $p_1 = 0111_2$

$$\mathsf{branchingBit}\ p_0\ p_1 = \mathsf{highestBit}\ p_0\ \text{XOR}\ p_1 = 0010_2$$

wobei *highestBit* durch rekursives Rechtsschieben ausgerechnet werden kann:

Listing 6: highestBit

```
fun highestBit :: "int ⇒ int" where
  "highestBit x = (if x ≤ 0 then
                     0
                   else (
                     if x = 1 then
                       1
                     else
                       (highestBit (x div 2)) * 2))"
```

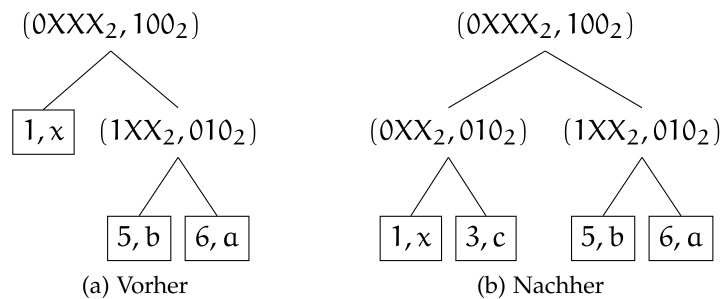
Listing 7: join

```
definition join :: "int ⇒ 'a patriciaTree ⇒ int ⇒ 'a
    patriciaTree ⇒ 'a patriciaTree" where
  "join p₀ t₀ p₁ t₁ = (let m = branchingBit p₀ p₁ in
                    (if zeroBit p₀ m then
                       Br (mask p₀ m) m t₀ t₁
                     else
                       Br (mask p₀ m) m t₁ t₀))"
```

Listing 8: Einfügen

```
fun insert' :: "('a × 'a ⇒ 'a) ⇒ key ⇒ 'a ⇒ 'a patriciaTree
    ⇒ 'a patriciaTree" where
  "insert' f k v Empty = Lf k v" |
  "insert' f k v (Lf j w) = (if j=k then
                              Lf k (f (w, v))
                            else
                              join k (Lf k v) j (Lf j w))" |
  "insert' f k v (Br p m t₀ t₁) = (
                 if mask k m = p then
                   if zeroBit k m then
                     Br p m (insert' f k v t₀) t₁
                   else
                     Br p m t₀ (insert' f k v t₁)
                 else
                   join k (Lf k v) p (Br p m t₀ t₁))"
```

LAUFZEIT Da *Einfügen* nur eine Suche mit anschließendem Erstellen eines neuen Knotens ist, beträgt wie schon beim *Nachschlagen* die Worst-Case-Laufzeit $\mathcal{O}(n, W)$.



(a) Vorher (b) Nachher

Abbildung 4: Einfügen von $(3, c)$

2.4.3 *Vereinigen*

Um zwei Maps M_1 und M_2 zu vereinigen, vereinigt man die zwei Schlüssel-Wert-Mengen der beiden Maps. Für jeden Schlüssel, der

sowohl in M_1 als auch in M_2 vorkommt, nutzt man wie schon beim *Einfügen* eine Funktion f, die den Konflikt auflöst.

Somit ergibt sich die neue Map M' zu:

$$\begin{aligned} M' = &\{(k,v) : (k,v) \in M_1 \wedge k \notin keys(M_2)\} \\ &\cup \{(k,v) : k \notin keys(M_1) \wedge (k,v) \in M_2\} \\ &\cup \{(k, f(v,w)) : (k,v) \in M_1 \wedge (k,w) \in M_2\} \end{aligned}$$

Um die Vereinigung bei Patricia-Bäumen durchzuführen, reicht es die Präfixe der beiden Bäume t_1 und t_2, die die Maps M_1 und M_2 repräsentieren, zu betrachten und folgende vier Fälle zu unterscheiden:

FALL I Sind die Präfixe p_1 und p_2 von t_1 und t_2 identisch, so lässt sich der linke Unterbaum von t_1 mit dem linken Unterbaum von t_2 vereinigen. Das gleiche gilt für den rechten Unterbaum der beiden Bäume.

FALL II Ist das Präfix p_1 in p_2 enthalten, so weiß man, dass t_1 in einen der beiden Unterbäume von t_2 gehört. Ob t_1 dabei in den rechten oder linken Unterbaum von t_2 gehört, entscheidet das Bit der Maske von t_2. Ein Beispiel, was es bedeutet, dass p_1 in p_2 enthalten ist:

$$\begin{aligned} p_1 &= 0101XXX_2 \\ p_2 &= 010XXXX_2 \end{aligned}$$

FALL III ist analog zu *Fall II* mit vertauschten Rollen von t_1 und t_2.

FALL IV Unterscheiden sich die Präfixe, so sind automatisch die Mengen der Schlüssel von t_1 und t_2 disjunkt und man kann mittels join einen neuen Knoten erzeugen, der t_1 und t_2 als Kinder enthält.

Listing 9: Vereinigen

```
fun merge' :: "('a × 'a ⇒ 'a) ⇒ 'a patriciaTree ⇒ 'a
    patriciaTree ⇒ 'a patriciaTree" where
  "merge' f Empty t1 = t1" |
  "merge' f t0 Empty = t0" |
  "merge' f (Lf k v) t1 = insert' f k v t1" |
  "merge' f t0 (Lf k v) = insert' (f ∘ swap) k v t0" |
  "merge' f (Br p m s0 s1) (Br q n t0 t1) = (
   if m=n ∧ p=q then
     Br p m (merge' f s0 t0) (merge' f s1 t1)
   else if m > n ∧ mask q m = p then
     if zeroBit q m then Br p m (merge' f s0 (Br q n t0 t1)) s1
```

```
                  else Br p m s0 (merge’ f s1 (Br q n t0 t1))
else if n > m ∧ mask p n = q then
  if zeroBit p n then Br q n (merge’ f (Br p m s0 s1) t0) t1
                 else Br q n t0 (merge’ f (Br p m s0 s1) t1)
else
  join p (Br p m s0 s1) q (Br q n t0 t1))"
```

LAUFZEIT Die Worst-Case-Laufzeit von *Vereinigen* beträgt $\mathcal{O}(n_0 + n_1)$ [4], wobei n_0 und n_1 jeweils für die Anzahl der Blätter der zu vereinigenden Bäume stehen. Da der Algorithmus den Baum strikt von oben nach unten abläuft, lässt er sich auch sehr gut parallelisieren.

2.5 BESONDERHEITEN BEI DER VERWENDUNG VON GANZZAHLEN BELIEBIGER BITLÄNGE

Falls man versucht einen Patricia-Baum aufzubauen, der sowohl einen negativen Schlüssel k_1, als auch einen positiven Schlüssel k_2 besitzt, fällt einem auf, dass sich das Präfix p nicht bestimmen lässt, da der negative Schlüssel beliebig viele führende Einsen und der positive Schlüssel beliebig viele führende Nullen besitzt. Somit existiert kein p, das für beide Schlüssel Präfix ist.

$$
\begin{aligned}
k_1 &= \dots 1 \dots 1X \dots X_2 \\
k_2 &= \dots 0 \dots 0X \dots X_2 \\
p &= ?
\end{aligned}
$$

Da p nicht zu bestimmen ist, ist die Maske m ebenfalls nicht zu bestimmen und es lässt sich kein Baum aufbauen, in dem sowohl positive als auch negative Zahlen vorkommen.

LÖSUNGEN Um dieses Problem zu beseitigen, boten sich drei Möglichkeiten. Die erste, die mir einfiel, war das Erweitern des Baum-Datentyps aus Listing 1 um einen weiteren Knoten BrN mit zwei Kindern, deren Aufgabe es sein sollte immer an der Wurzel des Baumes zu stehen und zwischen negativen und positiven Schlüsseln zu unterscheiden. Damit würde sich aber jede Funktionsdefinition auf dem Baum unnötig aufblähen.

Eine weitere Möglichkeit ist das explizite Speichern der maximalen Bitlänge aller positiven Zahlen im aktuellen Baum. Somit wäre ein Abbrechen bei der Suche nach dem höchstwertigsten Bit vorzeitig möglich. Allerdings wäre ein weiteres Attribut im Br-Knoten nötig gewesen, welches immer bis zur Wurzel propagiert werden müsste. Außerdem wäre das Neuberechnen dieses Maximums bei jedem Einfügen nötig gewesen. Deswegen habe ich die Idee auch schnell wieder verworfen.

Die dritte Möglichkeit und schlussendlich diejenige, die ich implementiert habe, bestand darin den Baum konsequent in einen negativen und einen positiven Teil aufzuteilen. Allerdings nicht mit Hilfe eines weiter Knotentyps, sondern indem man den Baum als Tupel von zwei Patricia-Bäumen auffasst. Das erste Element enthält den Baum für negative Schlüssel, das zweite Element den für positive. In Isabelle sieht das folgendermaßen aus:

Listing 10: Paar aus Patricia-Bäumen

```
type_synonym 'a pt = "('a patriciaTree × 'a patriciaTree)"
```

Für jede Operation aus Abschnitt 2.4 muss es somit auch eine Meta-Operation geben, die die Aufgaben auf die Elemente des Tupels verteilt. Exemplarisch für Nachschlagen sieht das so aus:

Listing 11: Nachschlagen in dem Tupel

```
definition lookup :: "key ⇒ 'a pt ⇒ 'a option" where
  "lookup k t = (if k < 0 then
                   lookup' k (fst t)
                 else
                   lookup' k (snd t))"
```

2.6 BIG-ENDIAN VS. LITTLE-ENDIAN

Bisher wurden alle Zahlen im Patricia-Baum als Big-Endian aufgefasst. Das heißt je höherwertig das Bit ist, desto weiter oben im Baum ist es zu finden. Es gibt aber auch die Möglichkeit die Schlüssel, Präfixe und Masken als Little-Endian-Zahlen zu interpretieren. In Abbildung 5 ist der Baum aus Abbildung 2 in Little-Endian-Darstellung zu sehen.

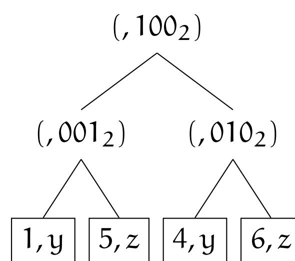


Abbildung 5: Patricia-Baum in Little-Endian-Darstellung

Der Vorteil, den man durch Little-Endian-Darstellung erhält, ist die vereinfachte Berechnung von *branchingBit*. Wie in [4] beschrieben, lässt sich *lowestBit* – die äquivalente Operation zu *highestBit* bei Big-Endian – durch einfache Bit-Operationen implementieren:

Listing 12: lowestBit

```
fun lowestBit :: "int ⇒ int" where
  "lowestBit x = x AND x̃"
```

wobei $\tilde{x}$ das Zweierkomplement von x ist.

Neben diesem einen Vorteil kommt die Little-Endian-Darstellung allerdings mit Nachteilen wie *schlechterem Cacheverhalten* und *langsamerer Vereinigung* bei Bäumen mit *aufeinanderfolgenden* Schlüsseln – in der Praxis häufig der Fall – daher. Außerdem ist es in der Little-Endian-Darstellung nicht möglich durch einfaches Ablaufen von links nach rechts die Schlüssel sortiert auszulesen. Insgesamt wirkt der Nachteil von Little-Endian also schwerer. [4]

3 VERIFIKATION

Um die Korrektheit der Implementierung aus Kapitel 2 zu beweisen, wurde der interaktive Theorembeweiser Isabelle [9] verwendet und folgende Korrektheitslemmas für die zwei Operationen *Einfügen* und *Vereinigen* formuliert.

3.1 EINFÜGEN

Damit eine Einfüge-Operation auf einer Map M korrekt ist, müssen grundsätzlich zwei Dinge gelten:

- Fügt man ein Schlüssel-Wert-Paar (k, v) in M ein, muss danach der Schlüssel k in der Map M gefunden werden.
- Existiert das Paar (k, v) bereits in M und fügt man (j, w) mit $k \neq j$ in M ein, so muss beim Suchen nach k der Wert v gefunden werden.

Existiert im ersten Fall bereits ein Paar (k, w) in M, so müssen die zwei Werte v und w mittels der Funktion f aus Kapitel 2 vereinigt werden.

Diese notwendigen Eigenschaften führen zu folgendem Lemma formuliert in *Isabelle/HOL*:

```
lookup k (insert f j y t) = (
  if j = k then
    case lookup k t of None → Some y |
                     Some x → Some (f (x, y))
  else
    lookup k t
)
```

3.2 VEREINIGEN

Wie in Unterabschnitt 2.4.3 beschrieben, ist die Vereinigung M zweier Maps M_1 und M_2 die Vereinigung der zwei Schlüssel-Wert-Paar-Mengen mit zusätzlicher Auflösung der Duplikate.

Um dies als Korrektheitslemma zu formulieren, unterscheidet man vier Fälle:

- Ist der Schlüssel k weder in M_1 noch in M_2 vorhanden, darf er auch nicht in M vorkommen.

- Ist der Schlüssel k in M_1, aber nicht in M_2 zu finden, so muss er in M zu finden sein.
- Ist der Schlüssel k in M_2, aber nicht in M_1 zu finden, so muss er in M zu finden sein.
- Ist der Schlüssel k sowohl in M_1 als auch in M_2 vorhanden, so muss er auch in M zu finden sein. Der korrekte Wert muss dabei wie bei Einfügen mittels f ermittelt worden sein.

Diese Eigenschaften lassen sich etwas kompakter als folgendes Lemma formulieren:

```
lookup k (merge f t1 t2) = (
  case lookup k t1 of
    None    ⇒ lookup k t2 |
    Some y1 ⇒ (case lookup k t2 of
                         None    ⇒ Some y1 |
                         Some y2 ⇒ Some (f (y2, y1)))
)
```

3.3 INVARIANTEN

Die Korrektheitslemmas für Einfügen und Vereinigen gelten im Allgemeinen nicht. Der Baum t muss erst gewisse Eigenschaften erfüllen, damit sie sich beweisen lassen. Diese Eigenschaften werden Invarianten genannt und im Folgenden stelle ich alle Invarianten vor, die notwendig sind, um die Korrektheit der beiden Lemmas zu beweisen. Außerdem gebe ich jeweils einen Baum an für die die Lemmas aus Abschnitt 3.1 bzw. Abschnitt 3.2 nicht gelten.

Dabei müssen die Invarianten sowohl nach dem *Einfügen* als auch nach dem *Vereinigen* weiterhin gelten, also folgende Lemmas erfüllen:

```
isPatriciaTree t ⟹ isPatriciaTree (insert f k v t)"

isPatriciaTree s ∧ isPatriciaTree t ⟹
  isPatriciaTree (merge f s t)"
```

isPatriciaTree t steht dabei für die $\wedge$-verknüpften Invarianten.

3.3.1 *Valide Präfixe*

Das Präfix p eines jeden inneren Knotens t gibt die Bits vor, mit denen alle Schlüssel k dieses Unterbaums beginnen. hasValidPrefixes stellt sicher, dass diese Eigenschaft auch tatsächlich für alle k erfüllt ist.

```
fun hasValidPrefixes' :: "'a patriciaTree ⇒ bool" where
  "hasValidPrefixes' Empty = True" |
  "hasValidPrefixes' (Lf _ _) = True" |
  "hasValidPrefixes' (Br p m t1 t2) = (
```

```
    (∀ k ∈ treeKeys t1 ∪ treeKeys t2. mask k m = p) ∧
    hasValidPrefixes' t1 ∧ hasValidPrefixes' t2)"
```

Existiert ein Schlüssel k, welcher nicht mit dem Präfix eines seiner Vorfahren übereinstimmt, lässt sich das Korrektheitslemma *Einfügen* nicht beweisen. Ein Baum für den die Invariante nicht gilt ist zum Beispiel folgender:

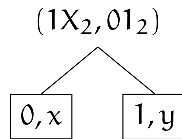


Abbildung 6: Baum t vor dem Einfügen von $(1, z)$

Das Präfix $p = 1X_2$ ist weder von 0 noch von 1 Präfix und mit $t = \text{Br } 2\ 1\ (\text{Lf } 0\ x)\ (\text{Lf } 1\ y)$ gilt

$$\text{lookup } 0\ t = \text{Some } x.$$

Fügt man nun $(1, z)$ in den Baum ein, erhält man

$$t' = \text{Br } 1\ 2\ (\text{Lf } 1\ z)\ (\text{Br } 2\ 1\ (\text{Lf } 0\ x)\ (\text{Lf } 1\ y)).$$

Damit gilt aber

$$\text{lookup } 0\ t' = \text{None} \neq \text{Some } x.$$

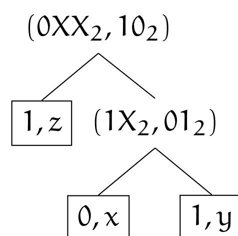


Abbildung 7: Baum t' nach dem Einfügen von $(1, y)$

3.3.2 *Negative und positive Schlüssel müssen getrennt werden*

Wie in Abschnitt 2.5 erwähnt, lässt sich keine Maske für eine positive und negative Zahl finden und deswegen sind negative und positive Schlüssel in unterschiedlichen Bäumen gespeichert. Fügt man allerdings sowohl -1 als auch 1 in einen leeren Patricia-Baum ein, so erhält man

$$t = \text{Br } 0\ 0\ (\text{Lf } -1\ b)\ (\text{Lf } 1\ a).$$

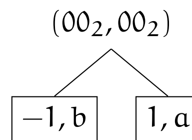


Abbildung 8: Unterschiedliche Vorzeichen im selben Baum

Die Maske m ist dabei 0, da *branchingBit* bei Eingaben mit unterschiedlichen Vorzeichen immer 0 liefert. Versucht man nun den Schlüssel $k = 1$ in t zu finden, wird das nicht gelingen, da gilt:

$$\text{zeroBit } k\ m = \text{zeroBit } 1\ 0 = 0$$

und somit immer im linken Unterbaum nach 1 gesucht wird:

$$\text{lookup } 1\ t = \text{lookup } 1\ (\text{Lf } -1\ b) = \text{None}$$

3.3.3 *Jeder innere Knoten besitzt genau zwei Kinder*

Diese Invariante ist mehr eine Meta-Invariante, die sicher stellt, dass keine inneren Knoten mit beliebigen Präfixen und Masken existieren. Für diese wären die oberen Invarianten automatisch erfüllt. Es werden also nur innere Knoten erlaubt, die genau zwei nicht-leere Kinder besitzen.

Verhindert man leere Kinder nicht, lässt sich zum Beispiel durch Einfügen die hasValidPrefixes-Invariante verletzen:

$$\text{Br } 0\ 1\ \text{Empty Empty}$$

wird nach Einfügen von $(3, x)$ zu

$$\text{Br } 0\ 1\ \text{Empty } (\text{Lf } 3\ x)$$

und 0 ist offensichtlich kein Präfix von 3.

```
fun noBrNodeHasEmptyChild' :: "'a patriciaTree ⇒ bool" where
  "noBrNodeHasEmptyChild' (Br _ _ t1 t2) = (
      t1 ≠ Empty ∧ t2 ≠ Empty ∧
      noBrNodeHasEmptyChild' t1 ∧ noBrNodeHasEmptyChild' t2)" |
  "noBrNodeHasEmptyChild' t = True"
```

3.4 WEITERE INVARIANTEN

Die Invarianten aus Abschnitt 3.3 sind zwar ausreichend um die Korrektheitslemmas für *Einfügen* und *Vereinigen* aus Abschnitt 3.1 und Abschnitt 3.2 beweisen zu können, sie decken allerdings nicht alles ab,

was man unter einem Patricia-Baum versteht. Im Folgenden sind deshalb weitere Invarianten gelistet, die der Intuition entsprechen und zusätzlich das Beweisen erleichtern.

3.4.1 *Schlüssel passen zur Maske*

Zum Beispiel lassen sich Bäume wie in Abbildung 9 konstruieren, die Schlüssel besitzen, welche aber nie durch *lookup* gefunden werden können. Sucht man nach 1 in diesem Beispielbaum, so wird man im rechten Unterbaum nachsehen, da das erste Bit von 1 gesetzt ist und findet nicht wie zu erwarten wäre $(1, b)$.

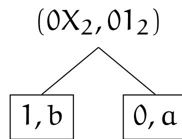


Abbildung 9: Hier sind $(1, b)$ und $(0, a)$ vertauscht

Es ist also wichtig, dass die Schlüssel, die sich im linken Unterbaum eines inneren Knotens t mit der Maske m befinden das $\tilde{m}$-te Bit nicht und die Schlüssel im rechten Unterbaum dieses Bit gesetzt haben. Das stellt die folgende Invariante sicher:

```
fun branchingBitDecides' :: "'a patriciaTree ⇒ bool" where
  "branchingBitDecides' (Br p m t1 t2) = (
     (∀ k ∈ treeKeys t1. zeroBit k m) ∧ (∀ k ∈ treeKeys t2. ¬
         zeroBit k m) ∧ branchingBitDecides' t1 ∧
         branchingBitDecides' t2)" |
  "branchingBitDecides' t = True"
```

Damit lässt sich dann auch zeigen, dass jeder Schlüssel, der im Baum exisitert auch mittels *lookup* gefunden werden kann:

```
k ∈ treeKeys t ⇒ lookup k t ≠ None
```

3.4.2 *Maske ist Zweierpotenz*

Jeder innere Knoten entscheidet immer an Hand von genau einem Bit, ob Schlüssel im rechten oder linken Unterbaum zu finden sind. Die Maske, die für diese Entscheidung zuständig ist, muss somit von der Form $m = 2^n$ mit $n \in \mathbb{N}$ sein. Diese Eigenschaft wird bei beiden Korrektheitslemmas benötigt. Dass die Maske in einem Patricia-Baum immer eine Zweier-Potenz ist, wird schon von der Invariante $hasValidPrefixes$ sichergestellt. Allerdings ist der Beweis

$$hasValidPrefixes\ t \Rightarrow maskIsPowerOfTwo\ t$$

nicht leicht. Einfacher war es die Invariante

```
fun maskIsPowerOfTwo' :: "'a patriciaTree ⇒ bool" where
  "maskIsPowerOfTwo' (Br p m t1 t2) = (
     isPowerOfTwo m ∧
     maskIsPowerOfTwo' t1 ∧ maskIsPowerOfTwo' t2)" |
  "maskIsPowerOfTwo' t = True"
```

zu formulieren und zu beweisen, dass nach jedem Einfügen und Vereinigen weiterhin $m = 2^n$ gilt. Es genügt dabei

$$\text{isPowerOfTwo}\ (\text{branchingBit}\ k\ j)$$

für $k \neq j$ zu zeigen.

4 EVALUIERUNG

In diesem Kapitel werde ich ein Auge darauf werfen wie sich die Patricia-Bäume gegen die Rot-Schwarz-Bäume aus *Isabelle/HOL* bezüglich der Geschwindigkeit schlagen.

4.1 PERFORMANCE

Den Patricia-Bäumen wird gegenüber anderen Maps ein deutlicher Geschwindigkeitsvorteil bei der Vereinigung nachgesagt. Um das zu überprüfen, habe ich sie mit der Rot-Schwarz-Baum-Implementierung in *Isabelle/HOL* [5] verglichen. Sowohl die Rot-Schwarz-Bäume als auch die Patricia-Bäume besitzen dabei Schlüssel mit beliebiger Bitlänge – intern benutzen beide den Datentyp *Int* aus der *Isabelle/HOL*-Bibliothek [7]. Aus diesem Grund erscheint ein Performance-Vergleich der beiden fair.

4.1.1 *Setup*

Isabelle/HOL selbst ist nur bedingt zum Ausführen von Code geeignet, deswegen dient die Code-Erzeugung von Isabelle/HOL [8] dazu die Isabelle-Theorien zunächst nach SML zu übersetzen und später mit Hilfe von *MLton* [2] ausführbaren Maschinencode zu erzeugen.

Die Benchmarks wurden mit folgender Hard- und Software durchgeführt:

- **CPU**: Intel Core i3 3,2 GHz
- **Speicher**: 2 GB
- **Betriebssystem**: Ubuntu 10.04
- **Theorembeweiser**: Isabelle/HOL 2012
- **Compiler**: MLton 20100608

4.1.2 *Nachschlagen*

Um die Operation *Nachschlagen* zu testen, wurde eine Map mit n Schlüsseln und *lookup* eine Millionen mal mit unterschiedlichen Schlüsseln aufgerufen.

Mit n entweder 10^4, 10^5, 10^6, $2 \cdot 10^6$ oder $5 \cdot 10^6$

Ob die Maps mit aufeinanderfolgenden Schlüsseln ($-\frac{n}{2}+1$ bis $\frac{n}{2}$) gefüllt war oder ob die Schlüssel zufällig gewählt waren, spielte dabei

nur eine kleine Rolle – Maps mit zufällig gewählten Schlüsseln waren in beiden Implementierungen geringfügig schneller.

Insgesamt schnitt die Rot-Schwarz-Baum-Implementierung besser ab – sie war bei wenigen Schlüsseln etwa doppelt so schnell wie die Patricia-Baum-Implementierung und ihre Laufzeit blieb mit zunehmender Anzahl an Schlüsseln beinahe konstant. (Abbildung 10)

Ein deutlicher Unterschied war allerdings festzustellen, falls man nicht nach aufeinander folgenden Schlüsseln sucht, sondern die Schlüssel zufällig wählt. Beide Implementierungen brauchten bis zu zehnmal länger bei diesem Szenario – was vermutlich auf das deutlich schlechtere Cache-Verhalten zurückzuführen ist – und die Patricia-Baum-Implementierung verlor deutlich an Boden. (Abbildung 11)

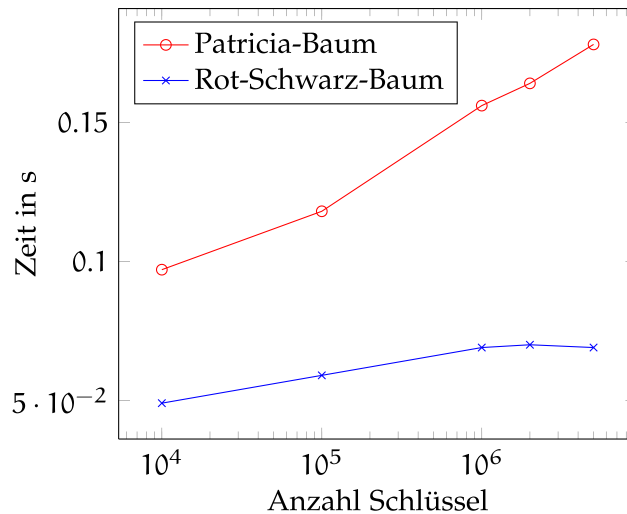


Abbildung 10: Performance von Nachschlagen mit aufeinanderfolgenden Schlüsseln

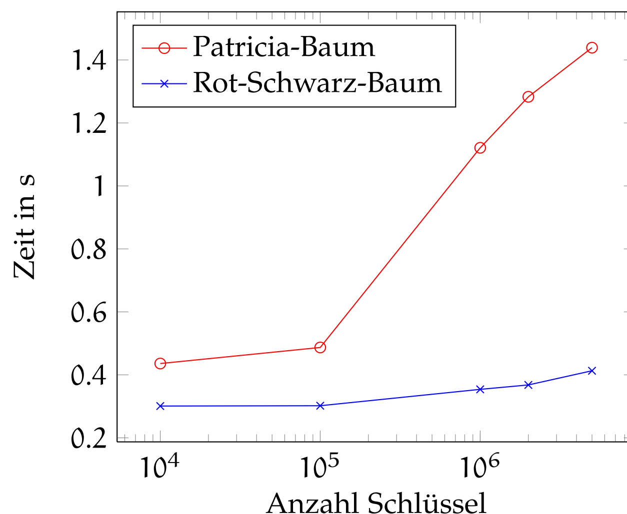


Abbildung 11: Performance von Nachschlagen in randomisierter Map

4.1.3 *Einfügen*

Den Graph aus Abbildung 12 erhält man, indem man n aufeinanderfolgende Schlüssel in eine leere Map einfügt. Dabei fällt auf, dass sich beide Implementierungen mit zunehmendem n in etwa gleich verhalten. Insgesamt ist der Patricia-Baum aber schneller.

Wählt man hingegen die einzufügenden Schlüssel zufällig, werden beide Implementierungen deutlich langsamer (Abbildung 13) – die Patricia-Implementierung bei fünf Millionen Schlüsseln um ein 20-faches, die Rot-Schwarz-Baum-Implementierung um ein 5-faches [Tabelle 1].

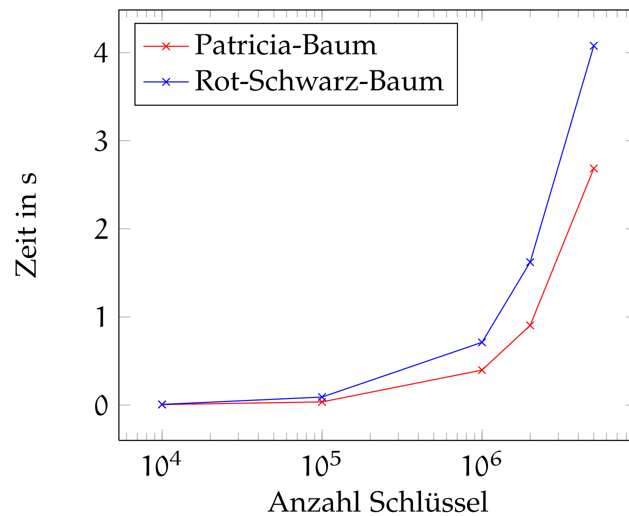


Abbildung 12: Performance von Einfügen mit aufeinanderfolgenden Schlüsseln

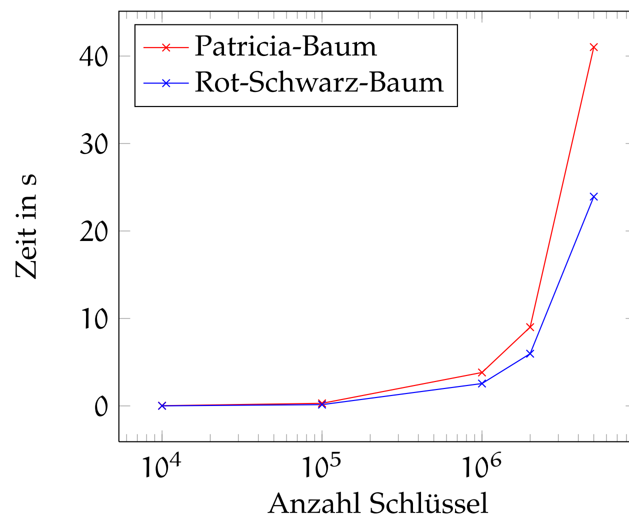


Abbildung 13: Performance von Einfügen mit zufällig gewählten Schlüsseln

4.1.4 *Vereinigen*

Bei wenigen Schlüssen sind beide Implementierungen in etwa gleich schnell. Erst ab etwa fünf Millionen Schlüsseln wird das performante Vereinigen der Patricia-Bäumen deutlich.

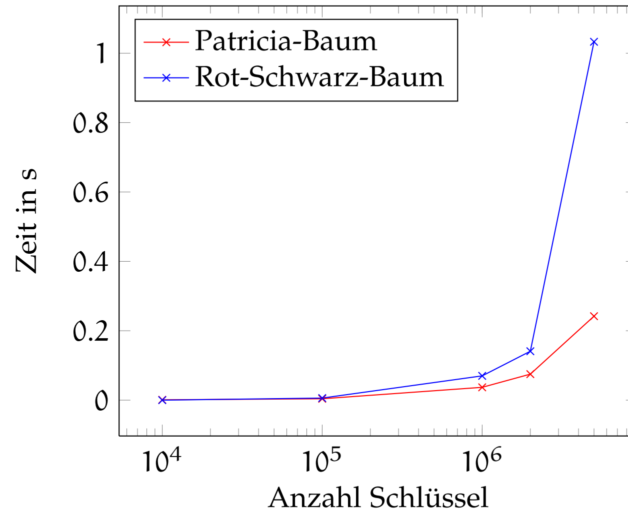


Abbildung 14: Benchmark-Ergebnisse von Vereinigen mit aufeinanderfolgenden Schlüsseln

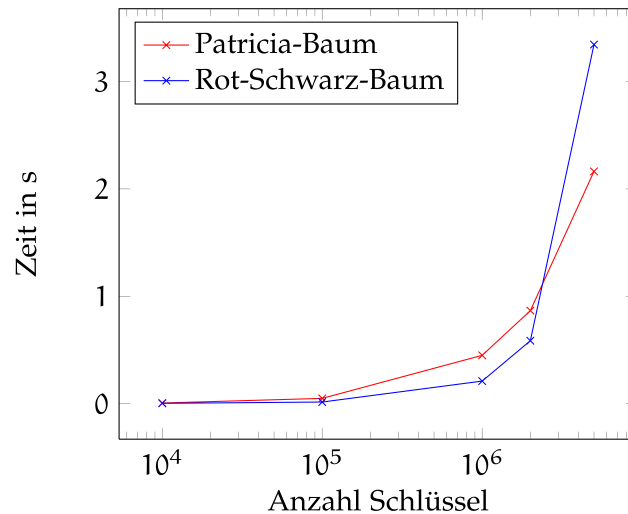


Abbildung 15: Benchmark-Ergebnisse von Vereinigen mit zufälligen Schlüsseln

4.1.5 *Weitere Tests und Auffälligkeiten*

In Tabelle 1 sind die Ergebnisse aller Benchmark-Tests aufgelistet. Dabei fallen noch folgende Punkte auf:

OPERATION	SETUP	PB	RSB	GEWINN
Nachschlagen	sequentiell	0,178	**0,069**	-158%
	zufällige Suche	1,439	**0,413**	-248%
	zufällig	0,154	**0,061**	-152%
Einfügen	sequentiell	**2,697**	4,198	35,8%
	sequentiell (Lücken)	**6,114**	7,275	16,0%
	sequentiell ($k > 2^{32}$)	56,740	**14,523**	-291%
	zufällig	41,188	**23,709**	-73,7%
Vereinigen	sequentiell	**0,242**	1,033	76,5%
	zufällig	**2,163**	3,344	35,5%
	sequentiell (alt)		3,232	92,5%
	zufällig (alt)		6,676	67,6%

Tabelle 1: Benchmark-Ergebnisse für fünf Millionen Schlüssel (Angaben in Sekunden)

ZAHLEN GRÖSSER 2^{32} Verwendet man als Schlüssel Zahlen, die mehr als vier Byte Speicher benötigen, stellt man sowohl bei den Rot-Schwarz-Bäumen als auch bei den Patricia-Bäumen eine deutliche Verlangsamung fest. Da die Schlüssel nicht mehr in den *Int32*-Datentyp von SML passen, können beide Implementierungen nicht mehr die nativen Implementierungen der CPU verwenden.

EINFÜGEN IN UMGEKEHRTER REIHENFOLGE Bei den Performance-Tests mit fünf Millionen und mehr Schlüsseln zeigt die Patricia-Implementierung deutliche Schwächen, wenn man aufeinanderfolgende Schlüssel nicht von negativen nach positiven Zahlen einfügt, sondern in umgekehrter Reihenfolge (siehe Abbildung 16). Die Rot-Schwarz-Bäume hatten dieses Problem nicht.

4.1.6 *Fazit*

Die Ergebnisse der Benchmarks aus [4] decken sich im Groben mit denen aus dieser Arbeit. Nur die Vereinigungs-Operation des Rot-Schwarz-Baumes aus Isabelle/HOL ist um ein Vielfaches schneller im Vergleich zu der Implementierung aus dem Paper. In [4] war das Vereinigen zweier sequentieller Maps mit Rot-Schwarz-Bäumen 15-mal so langsam wie mit Patricia-Bäumen. In dieser Arbeit bewegte sich der Faktor zwischen zwei und fünf.

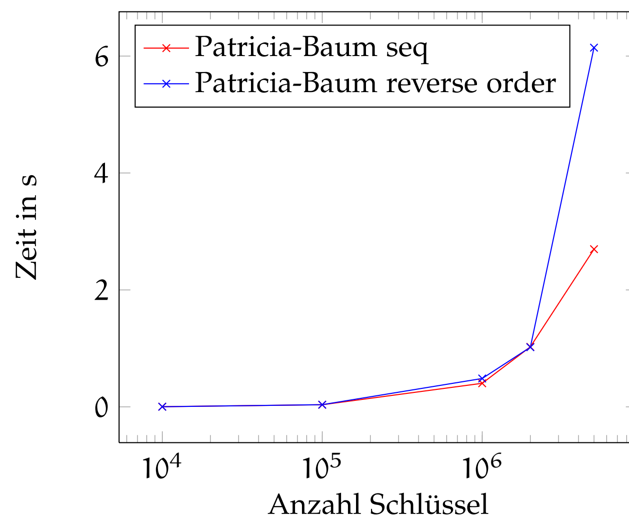


Abbildung 16: Benchmark-Ergebnisse von Einfügen in umgekehrter sequentieller Reihenfolge

5 FAZIT UND AUSBLICK

5.1 FAZIT

Patricia-Bäume stellten sich als eine elegante und einfach zu implementierende Datenstruktur heraus. Genau wie *Chris Okasaki* und *Andrew Gill* in [4] bin ich deshalb der Meinung, dass diese Datenstruktur mehr Aufmerksamkeit verdient hat. Schlüssel beliebiger Länge erfordern leider eine kleine Sonderbehandlung wie in Abschnitt 2.5 besprochen. Trennt man den Baum in zwei Teilbäume, ist man auch oft gezwungen Beweise für jeden Teilbaum extra zu führen. Insgesamt ist der Patricia-Baum eine gute Alternative zu den bisher bekannten Maps. Falls man, wie im Compiler üblich, mit vielen Vereinigungsoperationen rechnen muss, sind sie auf Grund der guten Performance sogar teilweise den üblichen Kandidaten wie *Rot-Schwarz-Bäumen* überlegen.

5.2 AUSBLICK

In Sachen *Performance* ist bei den Patricia-Bäumen noch nicht das letzte Wort gesprochen. Es gibt noch mindestens zwei Optimierungen, die möglich sind, die ich aber auf Grund der komplizierteren Beweise nicht implementiert habe.

AUSRECHNEN DES HÖCHSTEN BITS Die jetztige Implementierung von `highestBit x` geht jedes Bit von x durch, bis es bei dem höchstwertigen angekommen ist. Okasaki und Gill beschreiben in ihrer Arbeit einen Algorithmus, der nur die gesetzten Bits abläuft. Der Beweis der Terminierung dieser Funktion erwies sich aber als schwierig. Da die Funktion bei jeder Einfüge- und Vereinigungs-Operation höchstens einmal aufgerufen wird, ist der zu erwartende Gewinn auch nicht besonders groß.

NACHSCHLAGEN Durch die spezielle Form der Präfixe (siehe Unterabschnitt 2.4.2), lassen sich alle Tests nach einem Schlüssel an einem inneren Knoten durch Kleiner-Als-Vergleiche implementieren. Das könnte insbesondere für große Zahlen interessant sein, die nicht mehr nativ vom Prozessor behandelt werden können.

INVARIANTEN EINSPAREN Auch lässt sich überlegen, ob man nicht `Empty` aus dem Datentyp des Baumes herausnimmt und somit die Invariante *noBrNodeHasEmptyChild* aus Unterabschnitt 3.3.3 überflüs-

sig werden lässt. Die Beweise sollten dadurch teilweise einfacher werden. Den ursprünglichen Baum muss man dann allerdings über den `option`-Datentyp realisieren – `None` würde dem leeren Baum entsprechen, `Some t` dem nicht-leeren Baum.

NEGATIVE SCHLÜSSEL Der nächste Punkt, der verbessert werden könnte, ist die Sonderbehandlung von negativen Schlüsseln (Abschnitt 2.5). Ich bin mir sicher, dass sich eine elegantere Lösung finden lässt, die weder die Implementierung noch die Beweise unnötig aufbläht.

5.3 STATISTIKEN

Patricia-Bäume lassen sich in einer funktionalen Sprache sehr kompakt hinschreiben. Die eigentliche Implementierung umfasste nur **69** Zeilen. Weitere Zahlen über die Arbeit:

- Isabelle-Zeilen: **1556**
  - davon Codezeilen: **69**
  - und Beweiszeilen: **1487**
- Funktionen: **10**
- Definitionen und Abkürzungen: **16**
- Lemmas: **135**

LITERATURVERZEICHNIS

[1] An efficient implementation of maps from integer keys to values. http://www.haskell.org/ghc/docs/6.6/html/libraries/base/Data-IntMap.html. [Online; accessed 13-January-2013].

[2] MLton – an optimizing Standard ML compiler. http://www.mlton.org/. [Online; accessed 7-January-2013].

[3] Patricia trie in java. https://github.com/rkapsi/patricia-trie. [Online; accessed 13-January-2013].

[4] Andrew Gill Chris Okasaki. Fast mergeable integer maps. 1998.

[5] Florian Haftmann. Definition of Red-Black Trees in Isabelle/HOL. http://isabelle.in.tum.de/dist/library/HOL/HOL-Library/RBT.html. [Online; accessed 7-January-2013].

[6] Donald R. Morrison. Pratical algorithm to retrieve information coded in alphanumeric. 1968.

[7] Jeremy Dawson und Gerwin Klein. Definitions and basic theorems for bit-wise logical operations for integers. http://isabelle.in.tum.de/dist/library/HOL/HOL-Word/Bit_Int.html, 2012. [Online; accessed 7-January-2013].

[8] Florian Haftmann und Lukas Bulwahn. Code generation from Isabelle/HOL theories. http://isabelle.in.tum.de/documentation.html, 2012. [Online; accessed 7-January-2013].

[9] Lawrence Paulson (Universität Cambridge) und Tobias Nipkow (TU München) und Burkhart Wolff (Universität Paris-Süd). http://isabelle.in.tum.de/, 2012. [Online; accessed 10-January-2013].

DECLARATION

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Karlsruhe, Januar 2013

Tim Habermaas