Fortgeschrittene Datenstrukturen — Vorlesung 4

Schriftführer: Richard Hertel

10.11.2011

1 Van Emde Boas Trees

1.1 Literaturempfehlungen

- P. van Emde Boas: Preserving Order in a Forest in less than Logarithmic Time. Proc. 16th Annual Symposium on Foundations of Computer Science (FOCS), p. 75-84, 1975.
- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: Introduction to Algorithms (3rd ed.), MIT Press, 2009, Chapter 20.
- André Schulz: Skriptum zur Vorlesung "Datenstrukturen für Fortgeschrittene" d. Universität Münster (WS10/11), VL 19, zu finden unter dieser Adresse: http://wwwmath.uni-muenster.de/logik/Personen/Schulz/WS10/VL19.pdf.

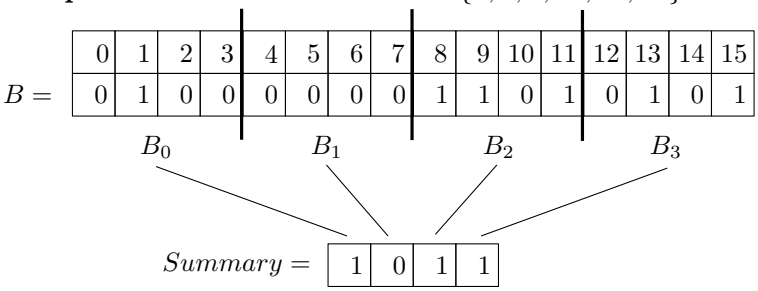
1.2 Laufzeiten

Bei Van Emde Boas Trees handelt es sich um eine dynamische Datenstruktur, auf der sich die Operationen zum Auffinden eines Nachfolgers bzw. Vorgängers in logarithmischer Zeit bezüglich der Schlüssellänge ausführen lassen. Da sich die Schlüssellänge wiederum logarithmisch zur Größe des Universums verhält, ergibt sich eine Laufzeit von $\mathcal{O}(\log w) = \mathcal{O}(\log \log u)$ im Worst Case.
Bezüglich der Laufzeit der Einfüge- und Löschoperationen sowie des Speicherbedarfs haben wir zwei Möglichkeiten: Entweder wir nehmen einen Speicherbedarf in Kauf, der sich linear zur Größe des Universums verhält und können dafür im Worst Case in $\mathcal{O}(\log w)$ Zeit löschen und einfügen, oder wir nehmen in Kauf beide Operationen nur erwartet und amortisiert in $\mathcal{O}(\log w)$ Zeit auszuführen und haben dafür einen Speicherbedarf linear zur Größe der Eingabemenge.

1.3 Konstruktion

Wir beginnen mit einem Bitvektor B der Größe des Universums, bei dem das Bit an der Stelle i genau dann gesetzt ist, wenn i in unserer Datenmenge enthalten ist. Anschließend unterteilen wir B in $\sqrt{u}$ Blöcke $B_0, \ldots, B_{\sqrt{u}-1}$ der Größe $\sqrt{u}$. Außerdem fügen wir noch einen weiteren Bitvektor der Größe $\sqrt{u}$ hinzu, der als Zusammenfassung dient und die Blöcke markiert, die nicht leer sind. Das $i$te Bit in diesem Vektor ist also genau dann gesetzt, wenn B_i mindestens eine Eins enthält.

Beispiel 1. *Sei* $u = 16$ *und* $S = \{1, 8, 9, 11, 13, 15\}$. *Dann sehen die Bitvektoren wie folgt aus:*



Ein Nachfolger einer Zahl kann jetzt gefunden werden, indem zunächst der Block, in dem sich die Zahl befindet, durchsucht wird. Wird dabei kein Nachfolger gefunden, wird über die Zusammenfassung der nächste nicht leere Block rechts davon bestimmt. Anschließend wird dort das kleinste Element bestimmt, welches dann der gesuchte Nachfolger ist. Alle diese Operationen können in $\mathcal{O}(\sqrt{u})$ ausgeführt werden. Äquivalent dazu kann auch die Vorgängersuche realisiert werden. Eingefügt werden kann in konstanter Zeit, da hierzu nur die entsprechenden Bits in den Blöcken und der Zusammenfassung gesetzt werden müssen. Löschen geschieht wiederum in $\mathcal{O}(\sqrt{u})$, da hierzu überprüft werden muss, ob der jeweilige Unterblock nach dem Löschen noch eine Zahl enthält. In diesem Fall wird darf das entsprechende Bit in der Zusammenfassung nicht gelöscht werden.
Man beobachtet, dass das Nehmen der Wurzel des Universums dem halbieren der Wortlänge entspricht:

$$\log \sqrt{u} = \log 2^{\frac{1}{2}w} = \frac{1}{2}w$$

Wir halbieren also die jeweiligen Wörter in einen Teil $high(x)$, der sich aus der ersten Hälfte der Bits zusammensetzt und die Blocknummer festlegt, sowie einen Teil $low(x)$, der aus den restlichen Bits besteht und die Position im jeweiligen Block angibt.

Beispiel 2. *Die Zahl 14 entspricht dem Wort 1110.* $high(14)$ *ist also 11 beziehungsweise 3,* $low(14)$ *ist 10, also 2. 14 wird also in Block 3 an Stelle 2 gespeichert. Die Blöcke sowie Stellen werden dabei von 0 beginnend nummeriert.*

Die Zahlen $high(x)$ und $low(x)$ können mit Hilfe des Shift-Operators und Bit-Maskierung in konstanter Zeit berechnet werden.
Man beobachtet, dass das Finden des ersten nicht leeren Blocks rechts von $high(x)$ einer Nachfolgersuche in der Zusammenfassung entspricht. Das Finden des Minimums in einem Block entspricht ebenfalls einer Nachfolgersuche, nämlich der Suche nach dem Nachfolger der Null. Äquivalentes gilt für das Bestimmen des ersten nicht leeren Blocks links von $high(x)$ und der Suche nach dem Maximum in einem Block, was beides einer Vorgängersuche entspricht. Es liegt also nahe die Datenstruktur rekursiv aufzubauen, also die Blöcke der Größe $\sqrt{u}$ weiter aufzuteilen in Blöcke der Größe $\sqrt{\sqrt{u}}$ mit jeweils einer weiteren Zusammenfassung und so weiter.
Um die Anzahl der rekursiven Aufrufe einzuschränken, speichern wir in jedem Block das Minimum, sodass wir in konstanter Zeit darauf zugreifen können. Der Algorithmus zur Nachfolgersuche sieht dann wie folgt aus:

```
function succ(B, x):
inblockSucc ← succ(B_{high(x)}, low(x))
if inblockSucc ≠ ⊥ then
    return inblockSucc + high(x) · √|B|
else
    succBlock ← succ(B.summary, high(x))
    if succBock = ⊥ then
        return ⊥
    else
        return min(B_{succBlock}) + succBlock · √|B|
    end
end
```

Wie im Algorithmus zu sehen, müssen wir maximal zwei rekursive Aufrufe auf Probleme der Größe $\sqrt{|B|}$ tätigen. Die Laufzeit wird also durch folgende Rekursionsgleichung beschrieben:

$$T(u) = 2T(\sqrt{u}) + \mathcal{O}(1).$$

Betrachten wir statt der eigentlichen Problemgröße die Wortlänge, so erhalten wir die Gleichung

$$T(w) = 2T\left(\frac{w}{2}\right) + \mathcal{O}(1).$$

Dies lässt sich mit dem Master-Theorem oder induktiv lösen zu: $T(w) \in \Theta(w)$. Die Laufzeit hängt also linear von der Wortlänge und somit logarithmisch von der Problemgröße ab: $T(u) \in \Theta(\log u)$. Diese Laufzeit ist uns zu langsam. Unser Ziel ist es, die Anzahl der rekursiven Aufrufe auf einen einzigen zu verringern. Dann erhalten wir die Gleichung

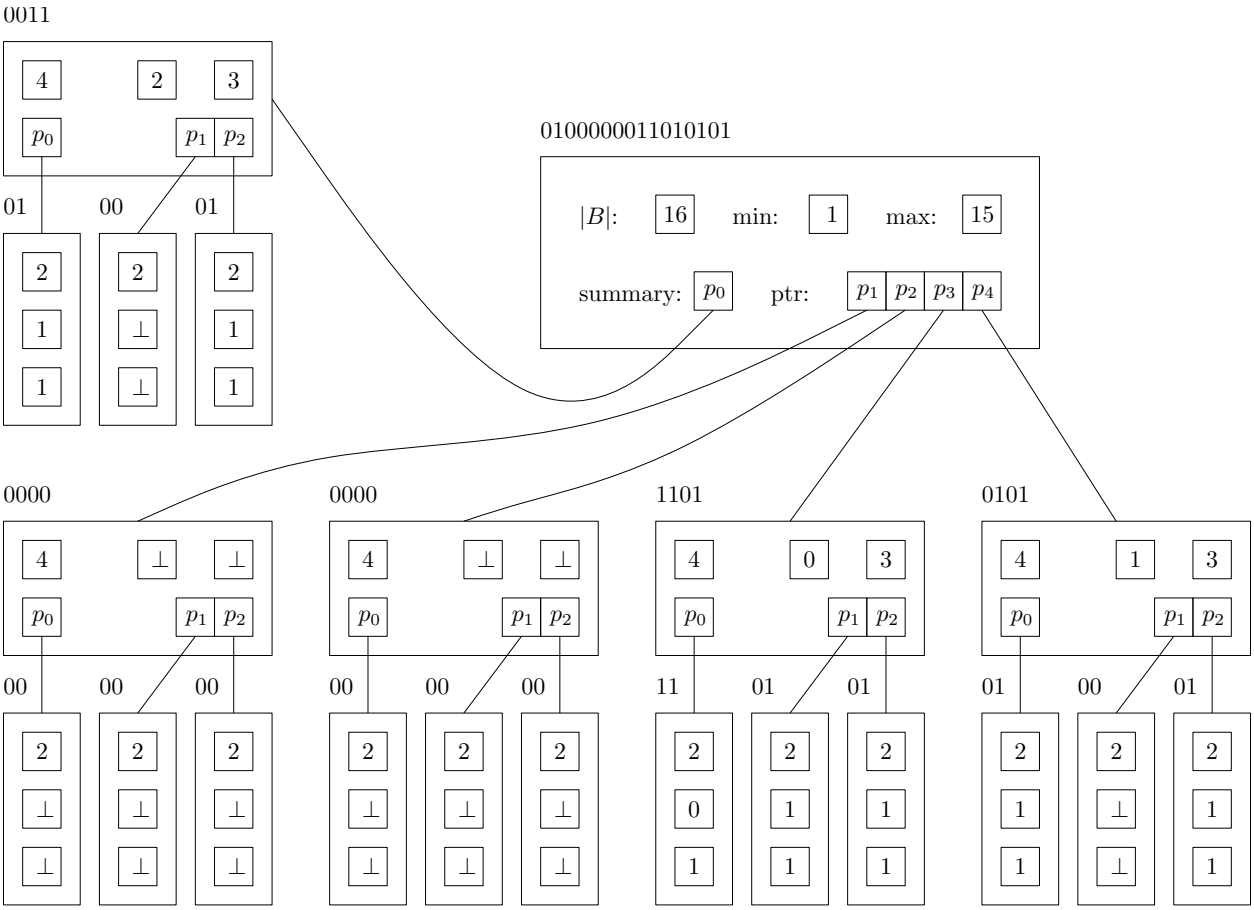
$$T'(w) = T'\left(\frac{w}{2}\right) + \mathcal{O}(1),$$

und somit eine Laufzeit von $T(w) \in \Theta(\log w)$ bzw. $T(u) \in \Theta(\log \log u)$.
Um dies zu erreichen speichern wir in jedem Block zusätzlich das Maximum, sodass wir jetzt in konstanter Zeit feststellen können, ob sich der Nachfolger in dem Block des Elements selbst, oder in einem Block rechts davon befindet. Danach führen wir entweder eine Rekursion in dem Block des Elements durch, oder in der Zusammenfassung um den Nachfolgerblock zu finden. Es kann aber nicht mehr vorkommen, dass beide Rekursionen durchgeführt werden.

Die Nachfolgersuche sieht dann wie folgt aus:

```
function succ(B, x):
if min(B) ≠ ⊥ and x < min(B) then
    return min(B)
else
    blockMax ← max(B_high(x))
    if blockMax ≠ ⊥ and low(x) < blockMax) then
        inblockSucc ← succ(B_high(x), low(x))
        return inblockSucc + high(x) · √|B|
    else
        succBlock ← succ(B.summary, high(x))
        if succBlock = ⊥ then
            return ⊥
        else
            return min(B_succBlock) + succBlock · √|B|
        end
    end
end
```

Als nächstes wollen wir die Einfügeoperation betrachten. Um dabei nicht zu viele Rekursive Aufrufe zu erhalten, bedienen wir uns eines Tricks: Das jeweilige Minimum eines Blocks wird ausschließlich in diesem Block selbst gespeichert und nicht in den entsprechenden Unterblock übernommen. Auch das entsprechende Bit in der Zusammenfassung wird nicht gesetzt, sofern sich im Unterblock auch keine weiteren Elemente befinden. Auf die Nachfolgersuche hat das keine weiteren Auswirkungen. Die Vorgängersuche erfolgt symmetrisch zur Nachfolgersuche, allerdings muss hier in der vorletzten Anweisung des Algorithmus zusätzlich der Fall betrachtet werden, dass der Vorgänger das Minimum des Blocks ist, und deswegen nicht in den Unterblöcken gespeichert wird.
Um das Nicht-Speichern des Minimums in Unterblöcken besser verständlich zu machen folgt als Beispiel der vollständige Baum für die Menge $S = \{1, 8, 9, 11, 13, 15\}$. Über den einzelnen Knoten stehen jeweils nochmal die Bitvektoren, die der entsprechende Unterbaum darstellt.



Nachdem wir dieses Beispiel eingehend betrachtet und verstanden haben ◡̈, können wir uns weiter der Einfügeoperation widmen:

```
procedure insert(B, x):
if min(B) = ⊥ then
    min(B) ← x
    max(B) ← x
else
    if x < min(B)) then
        swap(x, min(B))
    end
    if min(B_high(x)) = ⊥ then
        insert(B.summary, high(x))
    end
    insert(B_high(x), low(x))
    if max(B) < x) then
        max(B) ← x
    end
end
```

Dadurch, dass wir das Minimum nicht in den Unterblöcken speichern, müssen wir immer nur

eine volle Rekursion ausführen: ist der Block $B_{high(x)}$ leer, dann verursacht das Einfügen in die Zusammenfassung eine volle Rekursion, nicht aber das Einfügen von $low(x)$ in den Block $B_{high(x)}$. Das Einfügen in diesen leeren Block ist der Basisfall und geschieht in konstanter Zeit. Ist der Block $B_{high(x)}$ dagegen nicht leer, so ändert sich in der Zusammenfassung nichts und wir haben wieder nur einen rekursiven Aufruf.
Die Laufzeit ist also wieder durch die rekursive Gleichung

$$T(u) = T(\sqrt{u}) + \mathcal{O}(1)$$

beschrieben, sodass $T(u) \in \mathcal{O}(\log \log u)$ gilt. Das Löschen von Elementen verläuft äquivalent dazu.
Zu guter Letzt bleibt noch der Speicherbedarf zu analysieren. Die zugehörige rekursive Gleichung ist

$$S(u) = (\underbrace{\sqrt{u}}_{\text{blocks}} + \underbrace{1}_{\text{sum.}})S(\sqrt{u}) + \underbrace{\Theta(\sqrt{u})}_{\text{pointers}}.$$

Diese lässt sich, zum Beispiel induktiv auflösen zu $S(u) \in \Theta(u)$.
Um den benötigten Speicherplatz auf $\Theta(n)$ zu reduzieren können wir folgendes tun:

- Wir speichern nur nicht leere Blöcke rekursiv.
- Wir speichern die Zeiger in Hashtabellen, zum Beispiel mit cuckoo-hashing, statt in Arrays.
- Auch die Zusammenfassung wird nur gespeichert, wenn es mindestens einen nicht leeren Block gibt.

Da hierbei beim Einfügen und Löschen unter Umständen neu gehasht werden muss, ergeben sich die entsprechenden Laufzeiten nur noch erwartet und amortisiert.