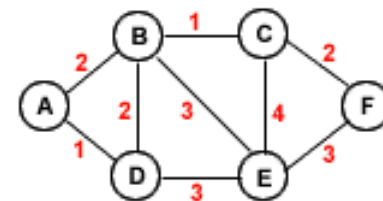




Effiziente Algorithmen

Hartmut Klauck
Universität Frankfurt
SS 06
29.6.





Amortisierte Laufzeit

- Die amortisierte Laufzeit einer Menge von Operationen einer Datenstruktur sei $O(c(n))$, wenn jede beliebige Folge von n solcher Operationen Laufzeit $O(n c(n))$ hat.
- Genauer: wenn es Operationen O^1, \dots, O^k gibt, und es gilt, daß jede Folge von Operationen, die n_1 mal O^1 und n_2 mal O^2 ... und n_k mal O^k enthält, eine Laufzeit von $O(\sum_i n_i c_i)$ hat, so nennen wir $O(c_i)$ die amortisierten Kosten von O^i
- Beispiel: n beliebige Operationen DecreaseKey, Insert haben Laufzeit $O(n)$, dann ist amortierte Zeit $O(1)$
- Fibonacci Heaps haben amortisierte Zeit $O(1)$ für Insert und DecreaseKey, $O(\log n)$ für ExtractMin



Ein Beispiel

- Betrachte einen binären Zähler:
 - Wir beginnen mit 0^n
 - Enden mit 1^n
 - In jedem Schritt wird von i nach $i+1$ erhöht
 - Naive Analyse:
 - Es gibt Schritte die Kosten n haben:
gehe von 0111111111 nach 1000000000
 - Daher Gesamtkosten $O(2^n n)$



Ein Beispiel

o Besser:

- Zahlen mit k konsekutiven Einsen am Ende kosten $k+1$ Schritte; t_x sei Anzahl der kons. Einsen am Ende von x
- Diese bilden einen $\frac{1}{2}^k$ Anteil aller Zahlen
- Das letzte Bit flippt jedes zweite mal beim Zählen
- Das vorletzte jedes vierte Mal
- etc.
- Gesamtkosten:

$$\sum_{i=0}^n \left\lfloor \frac{2^n}{2^i} \right\rfloor < 2^n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2 \cdot 2^n$$



Analyse mit Potentialfunktion

- Wir analysieren das Zählerproblem mit Hilfe einer Potentialfunktion
- Diese ist zu Beginn 0, und ihre Änderung wird zu jeder Operation addiert
- Assoziiert ein „Potential“ mit jeder Datenstruktur
- Definition:
 - Eine Potentialfunktion Φ bildet die Belegungen einer Datenstruktur auf reelle Zahlen ab.
 - Wenn D_{i-1} und D_i konsekutive Belegungen einer Datenstruktur sind, welche durch eine Operation O_i entstehen, so sind die Kosten von O_i definiert durch $t_i + \Phi(D_i) - \Phi(D_{i-1})$ für die tatsächlichen Kosten t_i
- Die Potentialfunktion erlaubt es während billiger Operationen Zeit für spätere teure Operationen zu kaufen



Analyse mit Potentialfunktion

- Die amortisierten Kosten für n Operationen sind dann

$$\sum_{i=1}^n (t_i + \phi(D_i) - \phi(D_{i-1})) = \left(\sum_{i=1}^n t_i \right) + \phi(D_n) - \phi(D_0)$$

- Φ sollte daher $\Phi(D_n) \geq \Phi(D_0)$ erfüllen, dann erhalten wir eine obere Schranke für die amortisierte Laufzeit mittels einer worst case Abschätzung der Kosten $t_i + \Phi(D_i) - \Phi(D_{i-1})$



Beispiel

- Für den Zähler setze $\Phi(x)$ =Anzahl Einsen in der Binärdarstellung von x
- $\Phi(0)=0$
- $\Phi(2^n - 1)=n$
- Inkrement:
 - Beim Zählen von x nach $x+1$ werden t_x Einsen geflippt, Kosten sind damit t_x+1
 - $x+1$ hat t_x-1 weniger Einsen als x
 - Damit sind die amortisierten Kosten
$$t_x+1+\Phi(x+1)-\Phi(x)$$
$$=2$$
- Wir erhalten also wieder eine Schranke von $2 \cdot 2^n$, bzw amortisierte Kosten von 2



Fibonacci Heaps

- Die Datenstruktur unterstützt folgende Operationen mit amortisierten Kosten
 - `MakeHeap()` : $O(1)$
 - `Insert(H,x)` : $O(1)$
 - `Minimum(H)` : $O(1)$
 - `ExtractMin(H)` : $O(\log n)$
 - `Union(H1,H2)` : $O(1)$
 - `DecreaseKey(H,x,k)` : $O(1)$
 - `Delete(H,x)` : $O(\log n)$



Geht es noch schneller?

- Füge n Zahlen x_1, \dots, x_n ein
- Dann n mal Extract_Min
- Wir erhalten somit einen Sortieralgorithmus, der mindestens Laufzeit $\Omega(n \log n)$ braucht, wenn wir mit Vergleichen arbeiten
- Daher entweder ExtractMin oder Insert mit amortisierten Kosten $\log n$
- Man kann auch erreichen, daß ExtractMin amortisierte Zeit $O(1)$ und Insert $O(\log n)$ hat!



Fibonacci Heaps

- Die Struktur hat Ähnlichkeit mit binomischen Heaps
- Wenn niemals `Decrease_Key` oder `Delete`, sind sie sogar bin. Heaps
- Schwierigkeit, bei bin. Heaps konstante Zeit zu erreichen:
 - Beibehaltung der Eigenschaft, nur bin. Bäume zu verwenden



Struktur

- Ein Fibonacci Heap ist eine Menge von (ungeordneten) Bäumen, an deren Knoten Schlüssel gespeichert sind, so daß jeder Baum heapgeordnet ist.
- Die Wurzeln sind in einer doppelt verketteten Liste
- Es gibt einen Pointer auf das Minimum (eine der Wurzeln)
- Die Kinder jedes Knotens sind in einer doppelt verketteten Liste verlinkt
- Jeder Knoten ist entweder markiert oder nicht markiert

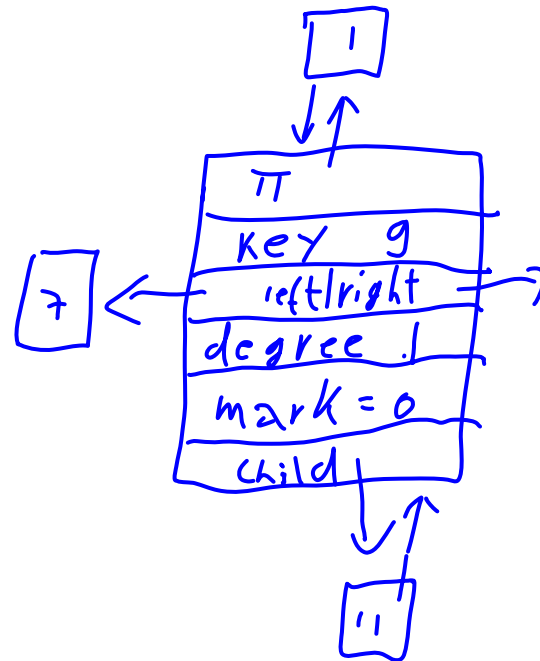
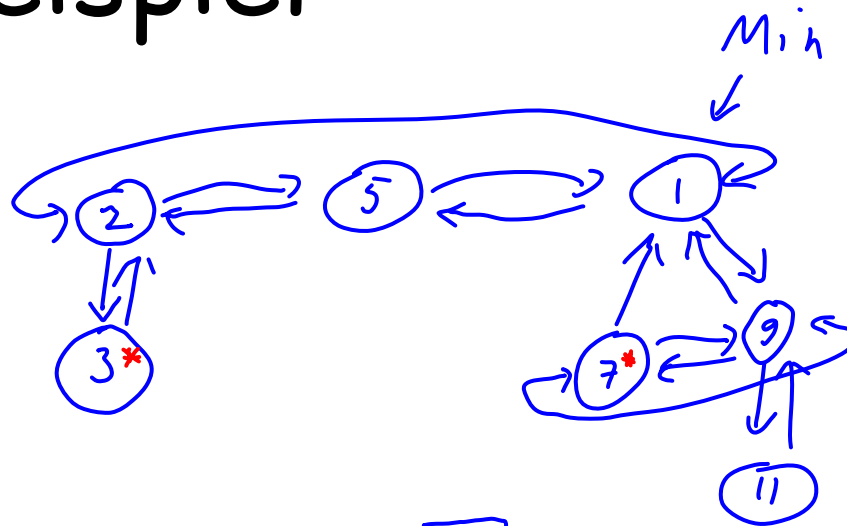


Bemerkungen

- Es gibt keine Einschränkung an die Struktur der Bäume
- Es wird keine Suchfunktion unterstützt, d.h. wie bei bin. Heaps muss für Decrease-Key ein Pointer zum Knoten vorliegen (Array von Pointers auf Knoten z.B. für Dijkstra)
- Daten per Knoten:
 - π : Pointer zum Elternknoten
 - degree: Anzahl Kinder
 - Key: Schlüssel
 - left, right: Pointer zu den Geschwistern/den anderen Wurzeln)
 - child: Pointer zu einem der Kinder
 - mark: Bit, markiert oder nicht
- Zusätzlich: Pointer zum Minimum, Feld $n(H)$: Größe



Beispiel





Die Potentialfunktion

- Für einen Fibonacci Heap H sei die Potentialfunktion wie folgt:
 - $t(H)$ sei die Anzahl Bäume
 - $m(H)$ sei die Anzahl der markierten Knoten
 - $\Phi(H) = t(H) + 2m(H)$
- Klar: der leere Fheap hat Potential 0, jeder Fheap hat nichtnegatives Potential, daher Potentialanalyse anwendbar
- Wir werden ausserdem zeigen, daß der maximale Knotengrad in einem Fheap \hat{H} mit n Elementen nur $O(\log n)$ ist



Ungeordnete binomische Bäume

- U_0 : einzelner Knoten
- U_{k+1} :
 - zwei U_k werden verbunden, indem einer zu einem der Kinder der Wurzel des anderen wird
- Eigenschaft: Wurzel von U_k hat Grad k und ihre Kinder sind alle U_0, \dots, U_{k-1}



Das Grundprinzip

- Verschiebe Arbeit so lange wie möglich!
- Wenn wenig Bäume vorhanden, so ist Extract_Min einfacher
- In bin. Heaps wird dies erreicht durch langsames Insert
- Verschiebe „Aufräumarbeit“ für Insert, bis es sich lohnt, d.h. nur wenn Extract_Min ausgeführt wird



Die Operationen

- Make-Heap:
Der leere Heap: $n(H)=0$, $\text{min}=\text{NIL}$
- Min:
 - Lese den Zeiger min



Insert

- Es wird ein neuer Knoten erzeugt, dieser ist ein eigener Baum
- Der Baum wird in die Wurzelliste eingefügt
- Der min Pointer und das $n(H)$ Feld werden eventuell neu gesetzt
- Der Knoten wird nicht markiert
- amortisierte Kosten:
 $O(1) + t(H) + 1 + 2m(H) - t(H) - 2m(H) = O(1)$



Unite

- Gegeben zwei Fheaps, bilde gemeinsamen Fheap
- Potentialfunktion einer Menge von Fheaps ist die Summe der Einzelpotentialfunktionen
- Verkette die Wurzellisten, bestimme das neue Minimum, setze $n(H)$
- Amortisierte Kosten: $O(1)$
 - Zeit $O(1)$
 - Potential ändert sich nicht



Extract_Min

- Sei $\min(H)$ die Wurzel eines Baums T
- Entferne die Wurzel von T und füge alle Kinder in H ein
- Danach Konsolidierung:
 - verschmelze Bäume mit gleichem Wurzelgrad, bis nur noch höchstens ein Baum mit Wurzelgrad k für jedes k vorhanden ist
 - Verschmelze Bäume indem eine Wurzel zum Kind der anderen wird, bewahre dabei Heapeigenschaft, d.h. die Wurzel mit kleinerem Schlüssel wird zum Kind
- D sei der maximale Grad eines Knotens in H



Konsolidierung

- Verwende Array $A[0], \dots, A[D]$:
Interpretation: $A[i]=y$: dann hat y Grad i
- 1. Initialisiere A mit NIL
- 2. Durchlaufe Wurzelliste
 - a) x sei die aktuelle Wurzel, setze d auf Grad von x
 - b) Teste ob $A[d]=\text{NIL}$
 - c) Ja: setze $A[d]=x$, gehe zu nächster Wurzel und 2.
 - d) Nein
 1. Wenn $A[d]=y$, dann verschmelze x und y , dabei wird die Wurzel mit größerem Schlüssel Kind und verliert ihre Markierung, x bezeichne nun die Wurzel mit kleinerem Schlüssel
 2. Setze $A[d]=\text{NIL}$ und danach $d:=d+1$, gehe zu 2b)
- 3. Durchlaufe A , bestimme das Minimum neu und erzeuge neue Wurzelliste aus A



Bemerkungen

- Die Wurzelliste wird genau einmal durchlaufen
- Nach der Schleife gibt es nur noch höchstens einen Baum für jeden Wurzellgrad
- Wenn der maximale Grad D ist, gibt es nachher also noch höchstens D viele Wurzeln
- Wir werden zeigen, daß D nur $O(\log n)$ ist
- Wenn alle Bäume ungeordnete binomische Bäume sind, so bleibt diese Eigenschaft erhalten
 - Wir fügen als Kinder alle B_0, \dots, B_k ein
 - wir verschmelzen jeweils zwei B_i



Laufzeit

- Sei D der maximale Knotengrad, der auftreten kann
- Behauptung: Laufzeit $O(D)$
 - $\leq D$ Kinder werden eingefügt
 - Nach der Konsolidierung sind noch $\leq D$ Bäume vorhanden, daraus wird in $O(D)$ die neue Wurzelliste erzeugt
 - Sonstige Kosten:
 - Es gebe $t(H)$ Bäume zu Beginn
 - $\leq t(H)+D -1$ Bäume nach Einfügung der Kinder
 - Durchlauf der neuen Wurzelliste: konstante Kosten pro Verschmelzung, und somit pro Baum, da jeweils zwei Bäume zu einem verschmelzen
 - Gesamtzeit also $O(t(H)+D)$, dies sei $C(t(H)+D)$ für eine Konstante C



Amortisierte Laufzeit

- Potentialänderung:
 - Vorher $t(H)$ Bäume, nachher $\leq D$
 - Änderung also $\leq D - t(H)$
- Diese Potential verwenden wir um den $t(H)$ Summanden in der Laufzeit zu amortisieren
- Dazu muss die Potentialfunktion insgesamt mit einem konstanten Faktor skaliert werden
[Kosten einer Verschmelzung, Faktor C]
- Damit amortisierte Kosten
 $C(D + t(H)) + C D + 2C m(H) - C t(H) - 2C m(H)$
 $= O(D)$
- Werden sehen: $D = O(\log n)$