Manipulation von Mengen

Übersicht

Formulierung von Algorithmen unter Verwendung von abstrakten Operationen auf Mengen.

Spezielle Algorithmen und Datenstrukturen zur Repräsentation der Datenmengen und der benötigten Operationen.

Allgemeiner Fall: Wörterbuchproblem (Suchen, Einfügen, Entfernen).

Spezialfälle mit besonderen (eingeschränkten Operationen) sind oft einfacher/effizienter.

- Vorrangwarteschlangen (Priority queues)
- Union-Find-Strukturen

Vorrangwarteschlangen (Priority queues)

Gegeben: Menge von Elementen mit einer Prioritäts-ordnung.

Wir betrachten hier: Mengen von Schlüsseln aus $\mathbb{N}$ mit der normalen Ordnung $<$.

Gesucht: Eine Datenstruktur auf der folgende Operationen effizient ausführbar sind.

- Initialisieren (der leeren Struktur)
- Einfügen (eines Elements)
- Minimum suchen
- Minimum entfernen

Oft werden auch weitere Operationen verlangt.

- Entfernen beliebiger Elemente
- Decrease-Key (Herabsetzen eines Schlüssels um einen geg. Wert)

Dabei nimmt man an, dass die Position des zu entfernenden/ändernden Elements bekannt ist, d.h., die Kosten der Suche werden nicht berücksichtigt.

Ausserdem: Zusammenfügen (Merge/Meld) von zwei elementfremden Priority queues.

Bemerkung: Die Operation "Suchen eines Elements" wird nicht verlangt!

Beispiel für die Anwendung von Priority Queues

Dijkstras Algorithmus zur Berechnung kürzester Wege

Single-source-shortest-paths Problem.

Gegeben:

- Graph G mit Knotenmenge V und Menge gerichteter Kanten E.
- Nichtnegatives Kantengewicht (Länge) $l(e)$ für alle Kanten $e \in E$.
- Startknoten s.

Gesucht: Kürzester Weg von s nach v für alle Knoten $v \in V$.

Der Einfachheit halber hier nicht der kürzeste Weg, sondern nur dessen Länge $d(v)$.

Dijkstras Algorithmus

Jedem Knoten $v \in V$ ist eine Distanz $d(v)$ zugeordnet.

Betrachte Teilmenge $S \subseteq V$ für die $d(v)$ schon bekannt ist. S entählt anfangs nur s und wird schrittweise vergrössert.

Knoten v aus $V - S$ wird eine vorläufige Distanz zugeordnet. $d(v) = min_{w \in S}(d(w) + l(wv))$ bzw. ∞.

Wiederhole folgende Schritte bis $S = V$.

1. Wähle Knoten $v \in V - S$ mit minimaler Distanz $d(v)$.
2. Füge v zu S hinzu.
3. Für alle $w \in V - S$, falls eine Kante vw existiert, ersetze $d(w)$ durch $min\{d(w), d(v) + l(vw)\}$.

Implementiere $V - S$ als Priority Queue mit Prioritäten $d(v)$.

Dijkstras Algorithmus (2)

Sei $N(v) := \{w \in V \mid vw \in E\}$.
Sei $K := V - S$, d.h., $S = V - K$.

procedure shortestpath((V,E):Graph, s:Knoten);
begin
for all $v \in V - \{s\}$ **do** $d(v) := \infty$;
$d(s) := 0; S := \emptyset$;
$K := V$(als Priority Queue, geordnet nach $d(v)$)
while $K \neq \emptyset$ **do**
begin
  $v := min(K); deletemin(K);$
  $S := S \cup \{v\}$;
  for all $w \in N(v) \cap K$ **do**
    if $d(v) + l(vw) < d(w)$
      then $decreasekey(w, d(v) + l(vw))$
end
end

Ergebnis am Ende in S gespeichert, d.h., am Ende sind alle Knoten in S und ihre zugeordneten Distanzen sind die richtigen Distanzen.

Komplexität von Dijkstras Algorithmus

Für einen Graphen G mit n Knoten und m Kanten

$min(K)$ und $deletemin(K)$ höchstens n mal ausgeführt und

$decreasekey$ höchstens m mal ausgeführt.

Die Laufzeit ist also

$$O(t_{init} + n \cdot (t_m + t_{dm}) + m \cdot t_{dk})$$

t_{init} Zeit zur Initialisierung, vor allem zum Aufbau der Priority Queue mit n Elementen.

t_m Zeit für Access Min. (bei PQ mit max. n Elementen).

t_{dm} Zeit für Delete Min. (bei PQ mit max. n Elementen).

t_{dk} Zeit für Decrease Key (bei PQ mit max. n Elementen).

Sehr abhängig davon, wie Priority Queues implementiert werden.

Implementierungen von PQ

Heaps

Problem: Zusammenfügen (Merge/Meld) von zwei Heaps zu teuer.

1. Möglichkeit: Einfügen der Elemente des kleineren Heaps in den grösseren. $O(N_1 log(N_1 + N_2))$.

2. Möglichkeit: Neuaufbau eines gemeinsamen Heaps. $O(N_1 + N_2)$.

Implementierungen von PQ (2)

Unsortierte lineare Listen

Initialisieren $O(n)$

Access Min. $O(n)$

Delete Min. $O(n)$ (bzw. $O(1)$ falls Position schon bekannt).

Merge $O(1)$.

Sortierte lineare Listen

Initialisieren $O(n \log n)$

Access Min. $O(1)$

Delete Min. $O(1)$

Merge $O(n_1 + n_2)$.

PQs als Bruder-Bäume (analog auch für jede andere Klasse von balancierten Bäumen)

Innere Knoten enthalten das Minimum im Teilbaum unterhalb.

Einfügen neuer Schlüssel an beliebigem Blatt, z.B. ganz rechts. Dann evtl. umstrukturieren um wieder einen Bruder-Baum zu erhalten und Minima an den internen Knoten adjustieren. Komplexität $O(\log n)$.

Access Min an der Wurzel in $O(1)$.

Delete Min: Minimalem Pfad im Baum folgen; minimales Blatt finden und entfernen; evtl. Baum umstrukturieren um wieder einen Bruder-Baum zu erhalten; Minima an den internen Knoten adjustieren. Komplexität $O(\log n)$.

Entfernen beliebiger Knoten (mit schon bekannter Position) ebenso wie Delete Min in $O(\log N)$.

Decrease-Key durch Entfernen eines Knotens und wieder Einfügen des neuen Wertes.

PQs als Bruder-Bäume (2)

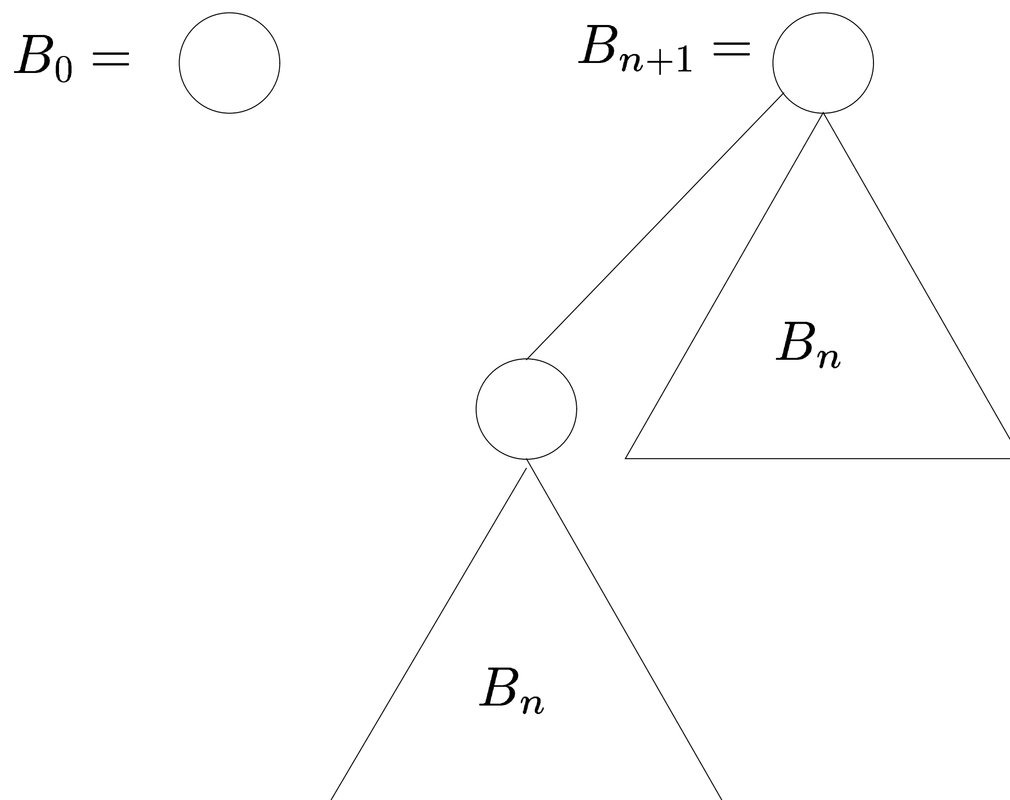
Zusammenfügen zweier PQs A und B die als Bruder-Bäume gespeichert sind. Sei (ohne Einschränkung) A höher als B. Nehme einen Teilbaum von A mit gleicher Höhe wie B. Falls dessen Vater unär: B als zweiten Sohn einfügen. Sonst: B als dritten Sohn einfügen und Baum umstrukturieren um wieder einen Bruder-Baum zu erhalten. Ausserdem: Minima an den internen Knoten adjustieren. Komplexität $O(\log n)$.

PQ als Binomial Queues

Binomial Trees B_n

B_0 besteht aus genau einem Knoten.

B_{n+1} wird aus zwei Exemplaren von B_n zusammengebaut. Die Wurzel des zweiten wird zu einem weiteren Sohn der Wurzel des ersten.



Eigenschaften von Binomial Trees

1. B_n hat 2^n Knoten.

2. B_n hat die Höhe n.

3. Die Wurzel von B_n hat die Ordnung n (n Söhne).

4. Die n Teilbäume der Wurzel von B_n sind $B_{n-1}, \ldots, B_1, B_0$.

5. B_n hat $\binom{n}{i}$ Knoten der Tiefe i.
   Bew. durch Induktion. $n = 0$ trivial.
   Schritt $i \to i + 1$:

$$\binom{n-1}{i} + \binom{n-1}{i-1} = \binom{n}{i}$$

Binomial Queues

Priority Queue mit N Elementen realisiert als Binomial Queue.

Stelle $N = (d_{n-1}, d_{n-2}, \ldots, d_0)_2$ als Dualzahl dar.

Die Binomial Queue F_N ist ein Wald von Binomial Trees. Dieser Wald enthält B_k genau dann wenn $d_k = 1$.

Da jeder Binomialbaum B_k genau 2^k Knoten enthält, enthält die Binomial Queue F_N genau N Knoten.

Die Binomial Queue F_N enthält also nur $O(\log N)$ Binomial Trees.

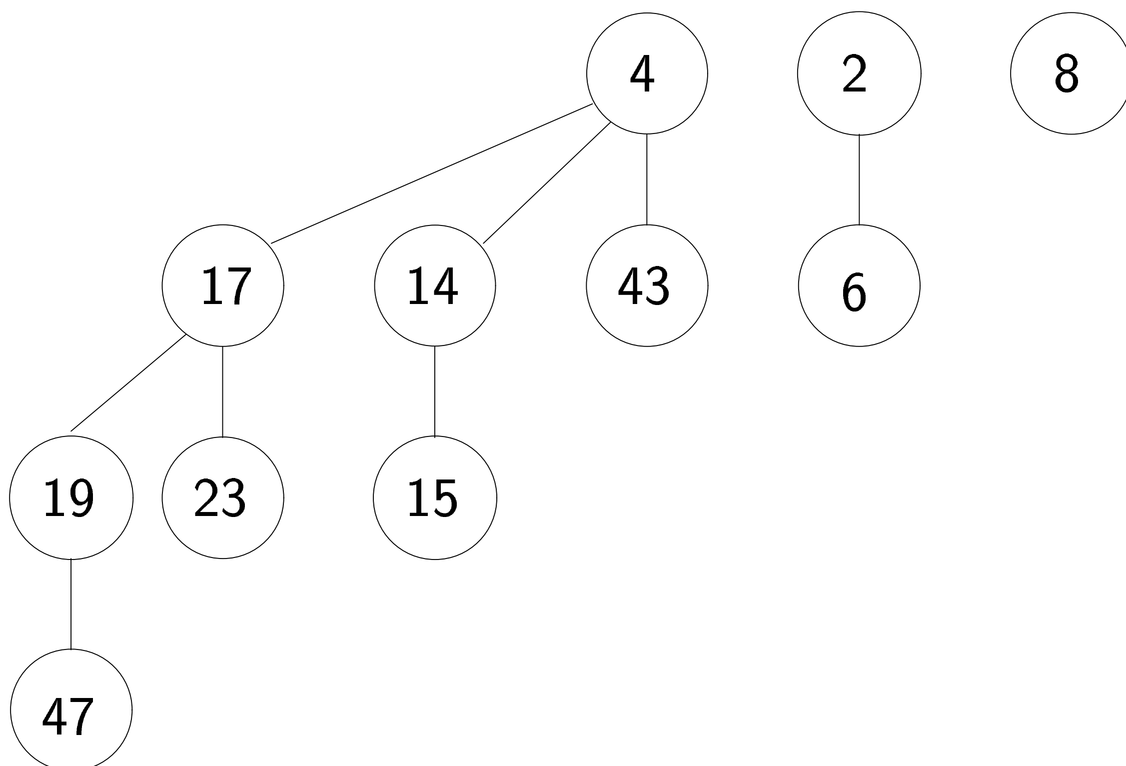
Jeder Binomial Tree für sich ist *heapgeordnet*, d.h., für jeden Knoten gilt: Der gespeicherte Schlüssel ist kleiner als die Schlüssel in den Söhnen des Knoten.

Binomial Queues: Beispiel

Gegeben sei eine Menge von 11 Schlüsseln: $\{2, 4, 6, 8, 14, 15, 17, 19, 23, 43, 47\}$.

Weil $11 = (1011)_2$ ist enthält F_{11} die Binomial Trees B_3, B_1 und B_0.

Eine mögliche Speicherung mit heapgeordneten Binomial Trees ist folgende:



Binomial Queues: Operationen

Access Min. Betrachte alle Wurzeln der Binomial Trees in der Binomial Queue F_n. Das geht in $O(\log N)$ Schritten, da es nur $O(\log N)$ Trees in der Queue gibt.

Verschmelzen:

Verschmelzen zweier gleich grosser heapgeordneter Binomial Trees B_n: Nehme den Baum mit dem kleineren Schlüssel an der Wurzel und füge den anderen als neuen Sohn ein. Man erhält einen heapgeordneten Binomial Tree B_{n+1}.

Ungleich grosse Binomial Trees müssen nie verschmolzen werden.

Verschmelzen zweier Binomial Queues F_{N_1} und F_{N_2}.

F_{N_1} und F_{N_2} bestehen jeweils aus $O(\log N_1)$ und $O(\log N_2)$ Binomial Trees. Verschmelze diese jeweils miteinander, analog zur Addition von Dualzahlen.

Komplexität: $O(\log N_1 + \log N_2)$.

Binomial Queues: Operationen (2)

Einfügen eines Elements lässt sich trivial auf Verschmelzen zurückführen.

Delete Min.: Das Minimum ist an der Wurzel eines der Binomialbäume B_k in der Binomial Queue. Entferne B_k aus der Binomial Queue. Entferne die Wurzel aus B_k, dann zerfällt dieser Baum in k Teilbäume $B_{k-1}, \ldots, B_0$. Man kann $B_{k-1}, \ldots, B_0$ als Binomial Queue auffassen. Verschmelze diese mit dem Rest der ursprünglichen Binomial Queue. Komplexität $O(\log N)$.

Löschen eines beliebigen Elements x: Nehme den Binomialbaum der x enthält aus der Binomial Queue heraus. Mache aus diesem Binomialbaum eine zweite Binomial Queue (durch Aufspalten des Baumes in Teilbäume bis x die Wurzel eines Teilbaums ist; dann wird x gelöscht wie bei Delete Min.). Verschmelze die beiden Binomial Queues.

Decrease-Key kann auf Löschen und Einfügen zurückgeführt werden. (Oder den Schlüssel verringern und so lange mit dem Vater vertauschen bis die Heapordnung wieder hergestellt ist.)

Binomial Queues: Implementierung

Binomialbäume sind keine Binärbäume. B_n hat Verzweigungsgrad n (an der Wurzel).

Implementierung von Bäumen mit unbeschränktem Verzweigungsgrad durch Binärbäume:

Jeder Knoten hat zwei Zeiger:

- Ein Zeiger auf den linkesten Sohn
- Ein Zeiger auf den rechten Nachbarn (Bruder).

Zusammenfügen von zwei (gleich grossen) Binomial Trees damit in $O(1)$ möglich.

Insgesamt: Alle Priority-Queue Operationen auf Binomial Queues in $O(\log N)$ möglich.

Union-Find Strukturen

Gegeben sei eine Menge die in disjunkte Untermengen aufgeteilt (partitioniert) wird. Die Aufteilung wird schrittweise durch Vereinigen von Untermengen vergröbert.

Make-set(e,i) schafft neue Menge i mit einzigem Element e.

Find(x) liefert den Namen der Menge die das Element x enthält.

Union(i,j,k) vereinigt Mengen i und j zu einer neuen Menge k. (Die Mengen i und j werden aus der Kollektion von Mengen gelöscht.)

Namen sind unwichtig. Man kann Mengen durch ein kanonisches Element identifizieren (z.B. das kleinste Element, falls eine totale Ordnung auf der Menge gegeben ist). Dann Operationen **Make-set(e)**, **Find(x)** und **Union(x,y)**.

Kruskals Verfahren zur Berechnung von MST: Problem

Gegeben:

- Graph G mit Knotenmenge V und Menge ungerichteter Kanten E.
- Kantengewicht (Cost) $c(e)$ für alle Kanten $e \in E$.

Gesucht: Ein minimaler spannender Baum (minimal spanning tree (MST)) für G. MST besteht aus allen Knoten V von G, aber nur einer Teilmenge der Kanten $E' \subseteq E$. Alle Knoten sind durch E' verbunden und die Summe der Kantengewichte in E' ist minimal.

Die Lösung muss ein Baum sein. Ansonsten gäbe es mindestens eine überflüssige Kante die entfernt werden kann.

Kruskals Verfahren zur Berechnung von MST: Idee

Lasse einen Wald von Teilbäumen zum MST zusammen wachsen.

Beginne mit N Teilbäumen die aus je genau einem Knoten bestehen.

Suche eine Kante mit minimalem Gewicht die nützlich ist, d.h., die zwei bisher getrennte Teilbäume verbindet, und nehme sie hinzu. Wiederhole das solange bis nur noch ein Baum übrig ist.

Die Mengen der Knoten in den Teilbäumen bilden eine Aufteilung von V. Durch Verbinden von Teilbäumen wird die Aufteilung vergröbert.

Kruskals Verfahren

procedure MST((V,E): Graph);
begin
$E' := \emptyset$;
$K := \emptyset$;
Bilde eine Priority Queue Q aller Kanten in E mit den Kantengewichten als Prioritäten.;
for all $v \in V$ **do** Make-set(v);
{jetzt ist K die Kollektion aller Mengen $\{v\}$, $v \in V$}
while K enthält mehr als eine Menge **do**
begin
  $(v, w) := min(Q); deletemin(Q)$;
  $v_0 := Find(v); w_0 := Find(w)$;
  if $v_0 \neq w_0$ **then**
  begin
    $Union(v_0, w_0)$;
    $E' := E' \cup \{(v, w)\}$;
  end
end
end

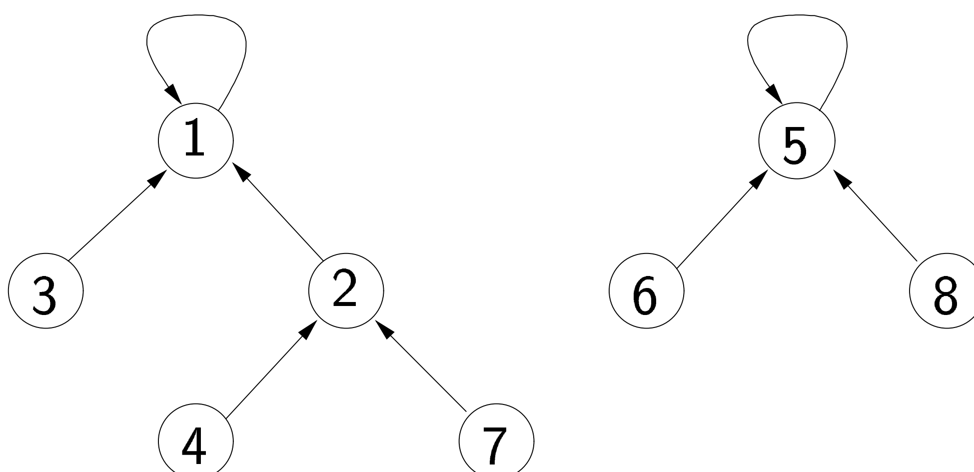
Union Find Strukturen: Datenstruktur

Teilmengen werden als unsortierte Bäume beliebigen Grades dargestellt. Jeder Knoten zeigt auf seinen Vater; die Wurzel zeigt auf sich selbst und enthält das kanonische Element.

Es gibt keine Operation “Einfügen”, d.h., die Gesamtmenge wächst nicht dynamisch. Stelle den Baum als Array dar.

type element = $\{1, \ldots, N\}$;
var p: **array** [element] **of** element;

Beispiel:



x	:	1	2	3	4	5	6	7	8
$p[x]$	:	1	1	1	2	5	5	2	5

Union Find Strukturen: Implementierung

procedure Make-set(x: element);
begin p[x]:=x **end**

procedure Union(x,y: element);
begin p[y] := x **end**

function Find(x: element):element;
var y: element;
begin
  $y := x$;
  while $p[y] \neq y$ **do** $y := p[y]$;
  Find := y
end

Komplexität:

Make-set: O(1)
Union: O(1)

Find: $\Omega(N)$, da im worst-case ein degenerierter Baum entstehen kann.

Verbesserung: Vereinigung nach Grösse (oder Höhe).

Vereinigung nach Grösse

Idee: Mache die Wurzel des kleineren Baumes zum weiteren Sohn des grösseren Baumes.

Extra Information über die Grösse der Teilbäume.

var Grösse: **array** [element] of Nat;

procedure Make-set(x: element);
begin
  p[x]:=x;
  Grösse[x]:=1
end

procedure Union(x,y: element);
begin
  if Grösse[x] < Grösse[y] **then** vertausche(x,y);
  {x ist jetzt kanonisches Element der grösseren Menge}
  $p[y] := x$;
  Grösse[x] := Grösse[x] + Grösse[y]
end

Die Komplexität von Make-set und Union ist immer noch O(1).

Vereinigung nach Grösse: Komplexität

Lemma Das Verfahren "Vereinigung nach Grösse" erhält die Eigenschaft "Ein Baum mit Höhe h hat $\geq 2^h$ Knoten."

Beweis

Vereinige zwei Bäume T_1 und T_2 mit Höhen h_1, h_2 und Grössen $g(T_1), g(T_2)$. Sei ohne Einschränkung $g(T_1) \geq g(T_2)$.

Nach Voraussetzung ist $g(T_i) \geq 2^{h_i}$, $i = 1, 2$.

1. Fall: $h :=$ Höhe$(T_1 \cup T_2) = max(\{h_1, h_2\})$.
Dann hat $T_1 \cup T_2$ trivialerweise mindestens 2^h Knoten.

2. Fall: $h :=$ Höhe$(T_1 \cup T_2) = max(\{h_1, h_2\}) + 1$.
Nach unseren Annahmen ist das nur möglich, falls $h = h_2 + 1$. Es gilt also

$$g(T_1) \geq g(T_2) \geq 2^{h_2}$$

und somit

$$g(T_1 \cup T_2) = g(T_1) + g(T_2) \geq 2 \cdot 2^{h_2} = 2^{h_2+1} = 2^h$$

Vereinigung nach Grösse: Komplexität

Folgerung aus dem Lemma:

Beginnt man mit N einelementigen Mengen und wendet das Verfahren “Vereinigung nach Grösse” iteriert an, so haben alle entstehenden Bäume eine Höhe $h \leq \log_2 N$.

Dann ist die Komplexität der Operation **Find** nur $O(\log N)$.

Vereinigung nach Höhe (statt Grösse) funktioniert genauso. (Ein analoges Lemma ist leicht zu beweisen.)

Kleiner Vorteil für Höhe: Die Höhe wächst mit $\log N$, also braucht man nur $\log\log N$ Bits um die Höheninformation zu speichern.

(Die Grösse wächst mit N, also braucht man dafür $\log N$ Bits.)

Pfadverkürzung

Bei Vereinigung nach Grösse/Höhe wächst die Höhe der Bäume nur mit $\log N$. Also braucht der Operation **Find** nur $O(\log N)$.

Wie geht es noch schneller (auf lange Sicht, d.h., falls **Find** oft ausgeführt wird) ?

Pfadverkürzung: Kompressionsmethode

Bei jeder **Find**-Operation werden alle durchlaufenen Knoten direkt an die Wurzel angehängt. (Man muss den Pfad zweimal durchlaufen, da man die Wurzel anfangs nicht kennt).

Also braucht die **Find**-Operation beim nächsten Mal für **diese** Knoten nur $O(1)$.

```
function Find(x: element):element;
var y,z,t: element;
begin
  y := x;
  while p[y] ≠ y do y := p[y];
  {jetzt ist y die Wurzel}
  z:=x;
  while p[z] ≠ y do
    begin t := z; z := p[z]; p[t] := y end;
  Find:=y
end
```

Kompressionsmethode: Komplexität

Betrachte n Elemente und m Operationen, mit $m \geq n$.

Verwende Vereinigung nach Grösse/Höhe und die Kompressionsmethode (bei **Find**-Operationen).

Dann benötigt die Ausführung einer beliebigen Folge von $m \geq n$ Operationen $\Theta(m \cdot \alpha(m,n))$ Schritte [Tarjan].

$\alpha(m,n)$ ist die Inverse der Ackermannfunktion und wächst sehr sehr langsam. $\alpha(m,n) \leq 4$ für alle praktischen Werte von n, m. Die Zeit für jeden einzelnen Schritt ist also fast konstant.

$$\alpha(m,n) = min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log n\}$$

$$\begin{aligned} A(1,j) &= 2^j, \text{ für } j \geq 1 \\ A(i,1) &= A(i-1,2), \text{ für } i \geq 2 \\ A(i,j) &= A(i-1, A(i,j-1)), \text{ für } i,j \geq 2 \end{aligned}$$

Pfadverkürzung: Weitere Methoden

Aufteilungsmethode (Splitting)

Änderung bei jeder **Find**-Operation: Lasse jeden Knoten (ausser dem letzten und vorletzten) statt auf seinen Vater auf seinen Grossvater zeigen.

function Find(x: element): element;
var x,t: element;
begin
  $y := x$;
  while $p[p[y]] \neq y$ **do**
  begin
    $t := y;\ y := p[y];\ p[t] := p[p[t]]$
  end
end

Halbierungsmethode (Halving)

Änderung bei jeder **Find**-Operation: Lasse jeden **zweiten** Knoten (ausser evtl. dem letzten und vorletzten) statt auf seinen Vater auf seinen Grossvater zeigen.