„Run-Relaxed Weak-Queues“

Stefan Edelkamp

Übersicht

- Marktanalyse State-of-the-Art
- Perfekte Weak-Heaps
- Weak Queues
- Heap-Speicherstruktur
- Knoten-Speicherstruktur
- Markierung und Lambda-Relaxierung
- Engineering

Datenstruktur Priority Queue

Abstrakter Datentyp mit den Operationen

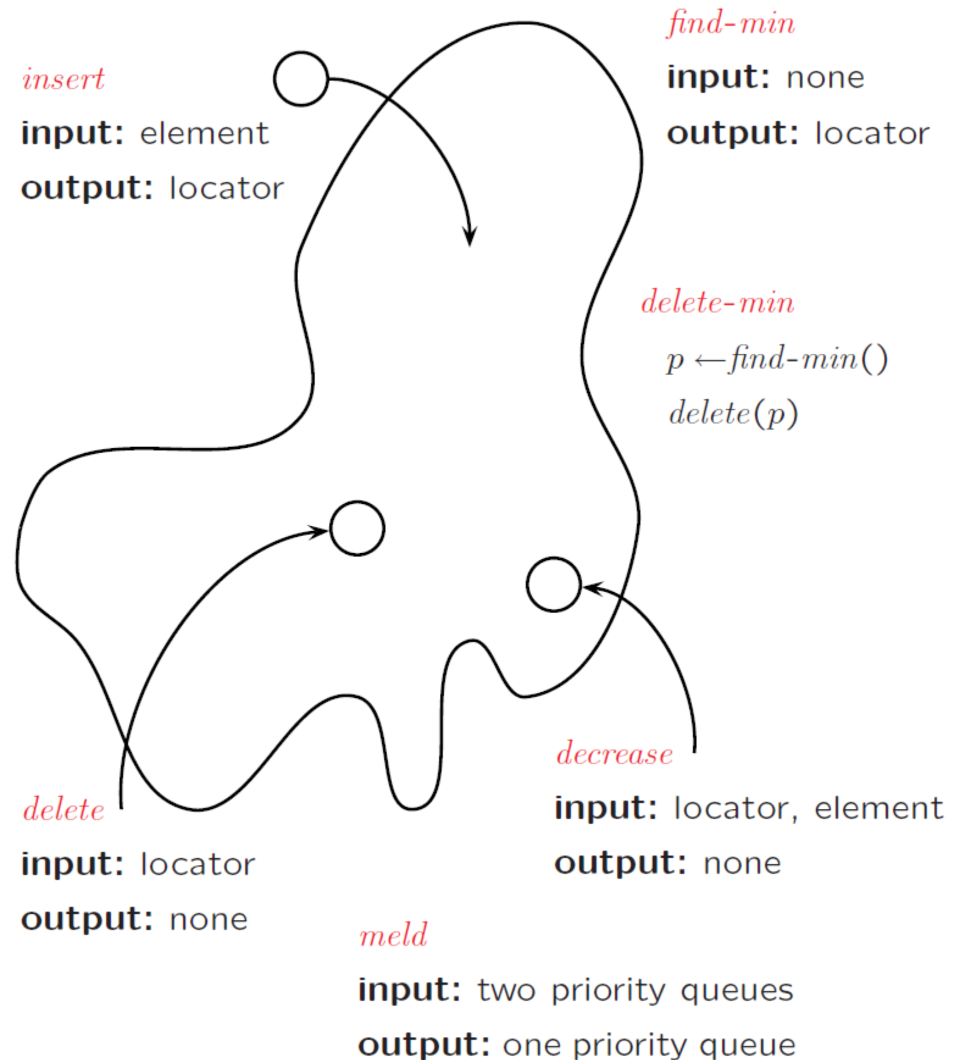
- *Insert,*
- *DeleteMin, and*
- *DecreaseKey.*

Wir unterscheiden Ganzzahl und allgemeine Gewichte

- Für Ganzzahlen nehmen wir an dass der Unterschied zwischen dem größten und kleinstem Schlüssel kleiner-gleich C ist

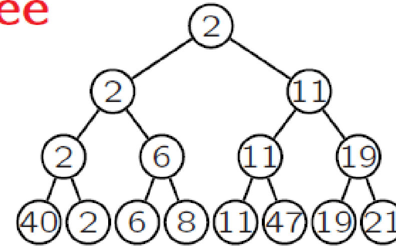
Für Dijkstra entspricht das w(e) = {1,…,C}

Alle Priority Queue Operationen im Bild



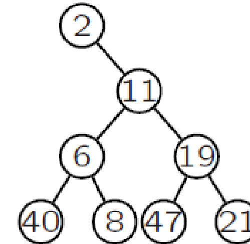
Verschiedene Ansätze

winner tree



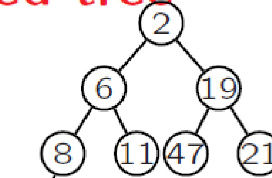
selection tree
navigation pile
binomial tree

loser tree



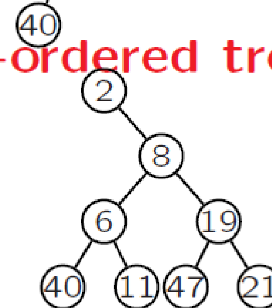
Vheap

heap-ordered tree



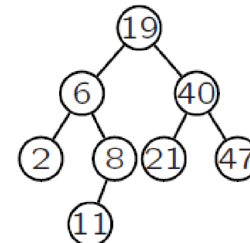
binary heap
leftist heap

weak-heap-ordered tree



weak heap

search tree



AVL tree
⋮

Marktanalyse

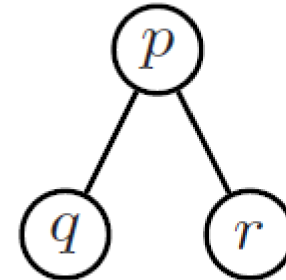
efficiency / method	binary heap worst case	binomial queue worst case	Fibonacci heap amortized	run-relaxed heap worst case
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
decrease	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(1)$
delete	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$
meld	$\Theta(\lg m \times \lg n)$	$\Theta(\min\{\lg m, \lg n\})$	$\Theta(1)$	$\Theta(\min\{\lg m, \lg n\})$

Here m and n denote the number of elements in the priority queues just prior to the operation.

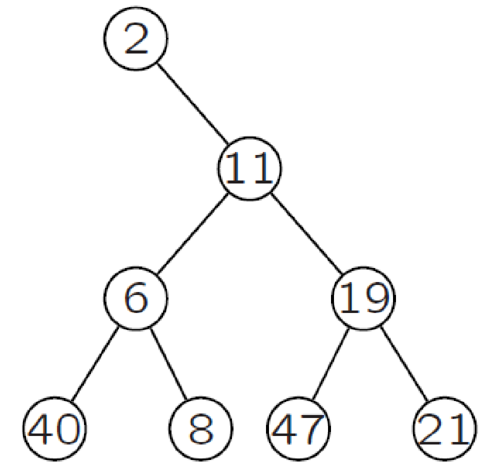
Run Relaxed Weak Queues

- Alternative zu Run-Relaxed Heaps:
- Einfacher zu programmieren
- Asymptotisch äquivalent zu Run-Relaxed Heaps
- Niedrige konstante Faktoren
  - delete benötigt 3 log n+O(1) Vergleiche
  - kann zu log n+O(log log n) verbessert werden
- Wenig Platz
  - Weniger als 3n+O(log n) extra words;
  - 4n+O(log n) with meld].

Notation



- p ist der Stiefvater/elter (surrogate parent) von q.
- p ist der wirkliche Vater/Elter (real parent) von r.
- Sei s ein Knoten in einem Binärbaum
- Jeder Vorgänger von s der ein wirklicher Vater/Elter
- Eines anderen Vorgänger einen wirklichen Vorgänger (grandparent, real ancestor) von s.



Perfekte Weak-Heaps

Ein perfekter Weak-Heap ist ein Binärer Baum mit den folgenden 3 Eigenschaften:

- Die Wurzel hat keinen linken Teilbaum
- Der rechte Teilbaum der Wurzel ist ein kompletter Binärbaum
- Für jeden Knoten s, ist das Element an s nicht kleiner als das Element an dem ersten wirklichen Vorgänger von s (grandparent)

Beobachtungen

1. Ein perfekter Weak-Heap speichert 2^h viele Elemente mit $h \geq 0$.
2. Die Wurzel eines Perfekten Weak-Heaps muss das minimale Element speichern.
3. Es gibt eine 1-zu-1 Abbildung von Heap-geordneten Binomialbäumen zu perfekten Weak-Heaps

Weak-Queues

A **weak queue** Q storing n elements is a collection of disjoint perfect weak heaps. Consider the binary representation of n

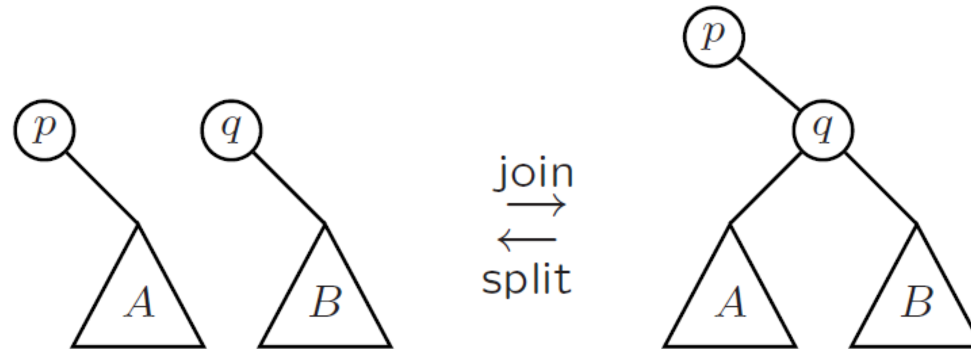
$$n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i ,$$

where $b_i \in \{0,1\}$ for all $i \in \{0,\ldots,\lfloor \lg n \rfloor\}$. In its basic form, Q contains a perfect weak heap H_i of size 2^i if and only if $b_i = 1$, i.e.

$$Q = \{H_i \mid n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i \text{ and } b_i = 1\}.$$

Joining/Merging and Splitting

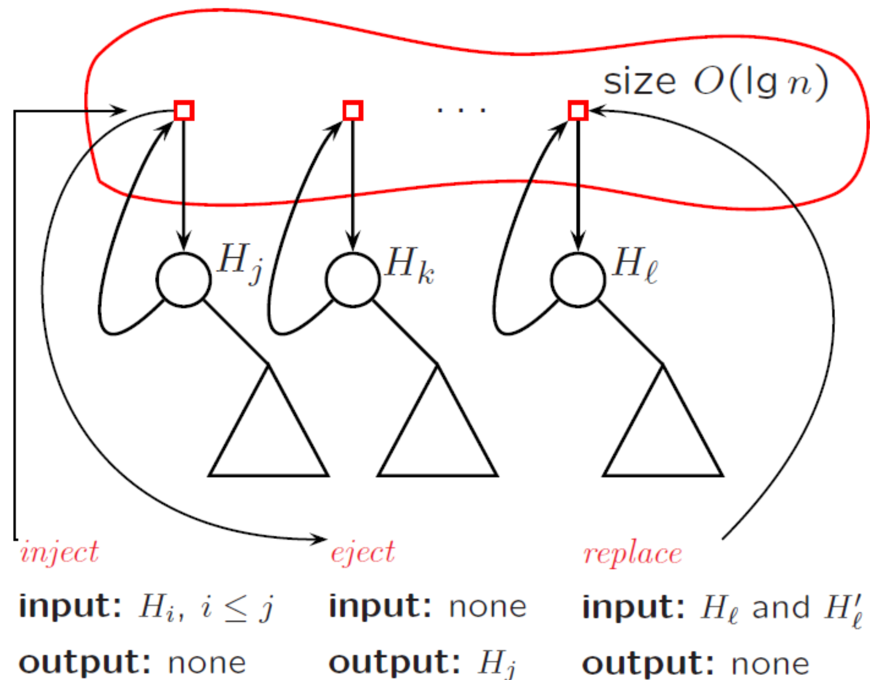
Joining and splitting two perfect weak heaps of the same size:



Note that for a binary heap a join may take logarithmic time.

Heap-Speicherstruktur (heap store)

Ein “heap store” ist eine Sequenz von perfekten Weak-Heaps
In aufsteigender Anordnung ihrer Größe



Lazy Injection

Heap-Einfügungen (injections) werden relaxiert (lazy) durchgeführt, wobei nicht jede Join-Operation direkt ausgeführt wird.

Dabei erlauben wir zwischen 0 und 2 perfekte Weak-Heaps jeder Größe.

Ziel: Alle Heap-Speicher Operationen “inject”, “eject”, und “replace” brauchen O(1) worst-case Zeit.

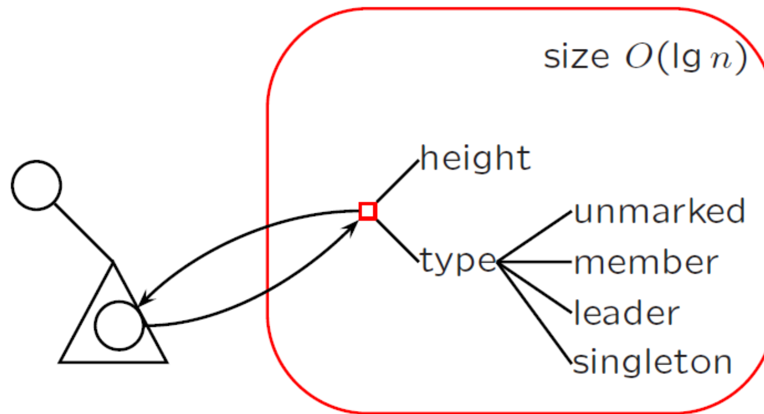
Reguläre rank-Sequenzen

- The rank-Sequenz (r0,..,rk) 2 isr regulär, falls jeder Ziffer 2 von eine 0 vorangeht, mit möglichen 1en in dazwischen
- Jede Teilsequenz der Form(0 1^l 2) wird “Block” genannt.
- Damit muss jede 2 Teil eines Blocks sein, nicht aber 1en und 0en
- Bsp. Die Sequenz (1011202012) beinhaltet 3 Blöcke
- Bei Einfügung werden die ersten zwei 1en gleicher Größe verbunden (join). Für konstanten Zugriff verwaltet ein “join schedule” alle noch ausstehenden joins als Paare mit Zeigern zu den perfekten Weak Heaps

Knotenspeicher

- Die wichtigste Aufgabe des Knotenspeichers (node store) ist es, die Weak-Heap Bedingung potentiell verletzenden Knoten (potential violation nodes) zu verwalten
- Diese werden je nach Operation mit Typinformation (unmarked, member, leader, singleton) markiert und unmarkiert
- Die zweit-wichtigste Aufgabe des Knotenspeichers ist es, die Höhen und Typen zu speichern (mitunter kann diese Information auch direkt an den Knoten gespeichert werden)
- Dazu gibt (in der Originalimplementation) einen Zeiger von dem Knoten in der Weak-Queue zu dem Knotenelement in dem Speicher und vice versa

Knotenspeicher-Operationen



mark
input: a node
output: none

unmark
input: a node
output: none

reduce
input: none
output: none
effect: Unmark at least one arbitrary marked node.

Ziel: Die Knotenspeicher Operationen

mark, unmark, und reduce

benötigen O(1) Zeit im worst-case.

Terminologie Knotenspeicher

A **weak-heap-order violation** occurs if the element stored at a node is smaller than the element stored at the **first** real ancestor of that node. In a **marked node** a weak-heap-order violation may occur.

A marked node is **tough** if it is the left child of its parent and also the parent is marked.

A chain of consecutive tough nodes followed by a single nontough marked node is called a **run**.

All tough nodes of a run are called its **members**.

The single nontough marked node of a run is called its **leader**.

A marked node that is neither a member nor a leader of a run is called a **singleton**.

Primitive und komplexe Operation für „Reduce“

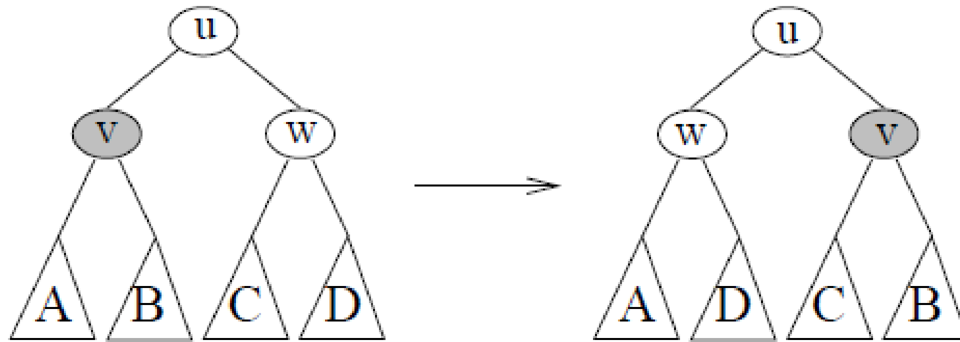
- Cleaning Transformation
- Parent Transformation
- Sibling Transformation
- Pair Transformation

Werden als Primitive zu einer

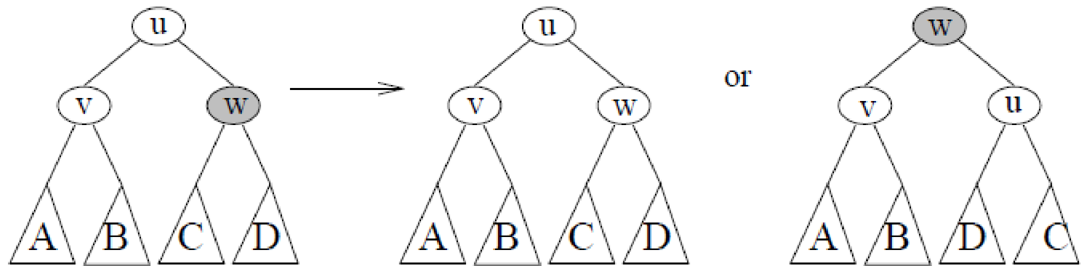
- Leader-Transformation
- Singleton-Transformation

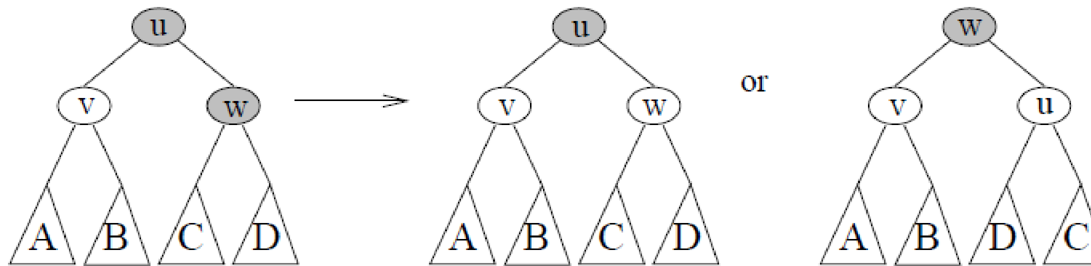
Zusammengestellt, die in den PQ-Operationen benutzt wird

Cleaning Transformation a)

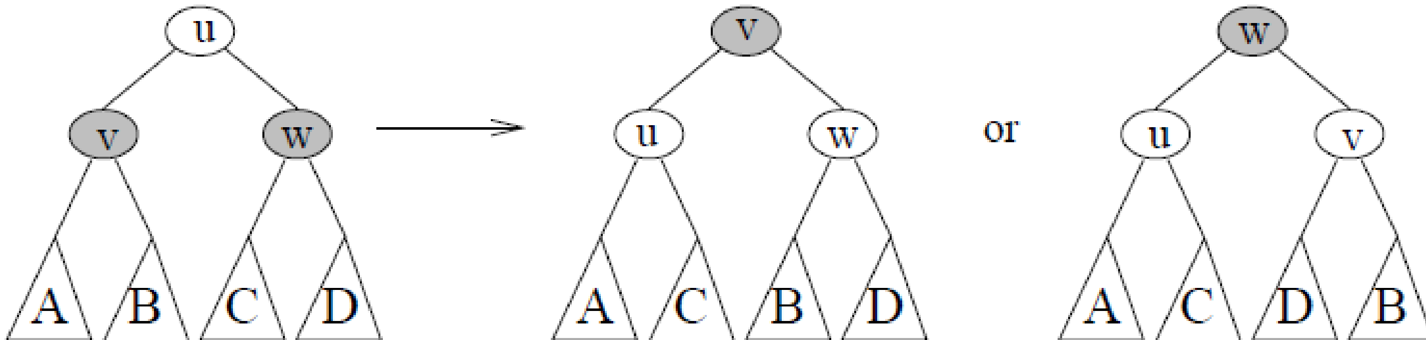


Parent Transformation b)

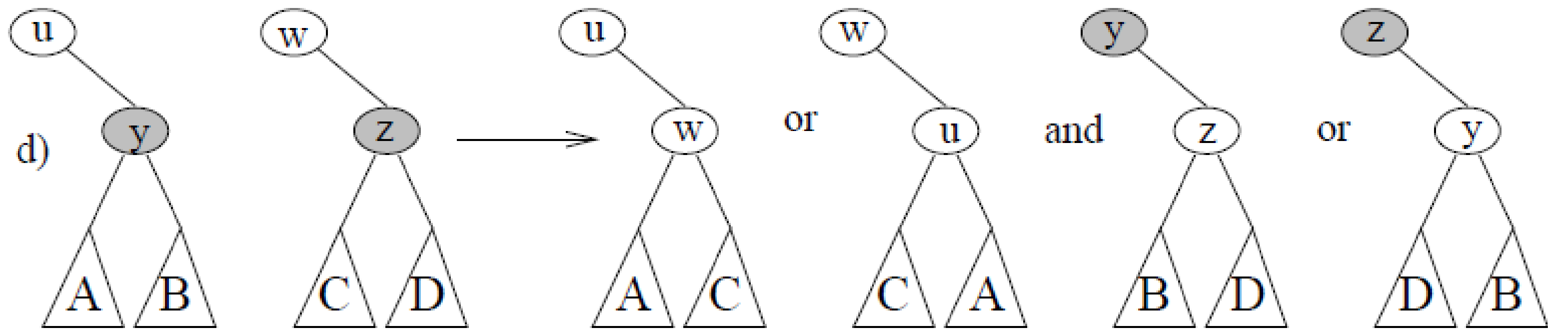




Sibling Transformation c)



Pair Transformation d)



Singleton Transformation

- two marked nodes q and s do not have the same parent and that they are of the same height
- q and s are the right children of their respective parents p and r, which both are unmarked.

1. subheaps rooted at p and r are split.
2. the produced subheaps rooted at p and r are joined and the resulting subheap is put in the place of the subheap originally rooted at p or r, depending on which becomes the root of the resulting subheap.
3. two remaining subheaps rooted at q and s are joined and the resulting subheap is put in the place of the subheap originally rooted at p or r, depending on which is still unoccupied after the second step. If after the third step q or s becomes a root, the node is unmarked.

➔ By this transformation at least one marked node is eliminated.

Run Transformation

- Idea: purpose of a run transformation is to move the two top-most marked nodes of a run upwards and at the same time remove at least one marking.
- Assume now that q is the leader of a run taken from the leader-object list and that r is the first member of that run. There are two cases depending on the position of q.
- Case 1. q is a right child.
- Case 2. q is a left child.

Case 1

- Apply the parent transformation to q.
- If the number of marked nodes decreased, stop. Now the parent of r is unmarked.
- If the sibling of r is marked, apply the sibling transformation to r and its sibling, and stop.
- Thereafter, apply the parent transformation once or twice to r to reduce the number of marked nodes.

Case 2

- If the sibling of q is marked, apply the sibling transformation to q and its sibling, and stop.
- Otherwise, apply the cleaning transformation to q, thereby making it a right child.
- Now the parent of r is unmarked.
- If the sibling of r is marked, apply the sibling transformation to r and its sibling, and stop.
- Otherwise, apply the cleaning transformation followed by the parent transformation to r.
- Now q and r are marked siblings with an unmarked parent; apply the sibling transformation to them to reduce the number of marked nodes.

Procedur reduce (leader transform)

```
if (leaders ≠ ∅)                                              ;; Leader exists on some level
  leader ← leaders.first ; leaderparent ← parent(leader)      ;; Select leader and parent
  if (leader = leaderparent.right)                            ;; Leader is right child
    parenttrans(leaderparent)                                 ;; Transform into left child
    if (¬marked(leaderparent) ∧ marked(leader))               ;; Parent also marked
      if (marked(leaderparent.left) siblingtrans(leaderparent); return ;; Case c) suffices
      parenttrans(leaderparent)                               ;; Case b) applies first time
    if (marked(leaderparent,right)) parenttrans(leader)       ;; Case b) applies second time
  else                                                        ;; Leader is left child
    sibling ← leaderparent.right                              ;; Temporary variable
    if (marked(sibling)) siblingtrans(leaderparent); return   ;; Case c) suffices
    cleaningtrans(leaderparent)                               ;; Toggle marking of leader's children
    if (marked(sibling.right)) siblingtrans(sibling); return  ;; Case c) suffices
    cleaningtrans(sibling)                                    ;; Toggle marking of sibling's children
    parenttrans(sibling)                                      ;; Case b) applies
    if (marked(leaderparent.left)) siblingtrans(leaderparent) ;; Case c) suffices
```

Prozedur reduce (singleton transform)

```
else if (chairmen ≠ ∅)                                          ;; Fellow pair on some level
   first ← chairmen.first; firstparent ← parent(first)          ;; 1st item and its parent
   if (firstparent.left = first and marked(firstparent.right) or       ;; Two children ...
      firstparent.left ≠ first and marked(firstparent.left))           ;; ...marked already
         siblingtrans(firstparent); return                              ;; Case c) suffices
   second ← chairmen.second; secondparent ← parent(second) ;; 2nd item and its parent
   if (secondparent.left = second and marked(secondparent.right) or  ;; Two children ...
      secondparent.left ≠ second and marked(secondparent.left))        ;; ...marked already
         siblingtrans(secondparent); return                             ;; Case c) suffices
   if (firstparent.left = first) cleaningtrans(firstparent)     ;; Toggle children marking
   if (secondparent.left = second) cleaningtrans(secondparent)        ;; Case a) applies
   if (marked(firstparent) or root(firstparent))                   ;; Parent also marked
      parenttrans(firstparent); return                                 ;; Case b) applies
   if (marked(secondparent) or root(secondparent))                 ;; Parent also marked
      parenttrans(secondparent); return                                ;; Case b) applies
      pairtrans(firstparent, secondparent)                             ;; Case d) applies
```

Insert(e)

1. Allocate a new node and put e there.
2. Place the new node, which is also a perfect weak heap of height 0, into the heap store by invoking *inject*.
3. Correct the minimum pointer to point to the new node if e is smaller than the current minimum.

Worst-case time: $\Theta(1)$ with at most 2 element comparisons

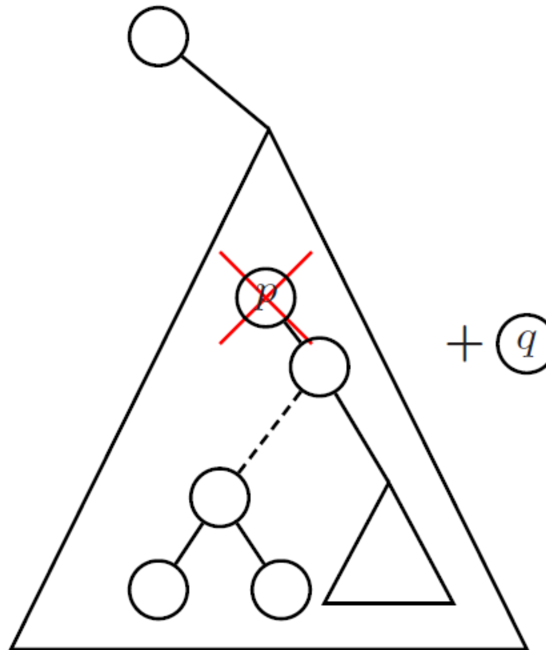
Decrease(p,e)

1. Make the element replacement at p.
2. Make p a potential violation node by invoking $mark$.
3. Reduce the number of potential violation nodes, if possible, by invoking $reduce$.
4. Correct the minimum pointer if necessary.

Worst-case time: $\Theta(1)$ with at most 4 element comparisons

Delete(p)

The idea is to extract the subheap rooted at p from the perfect weak heap, in which it resides, borrow another node q from the smallest perfect weak heap to fill in the hole created by p, and put the new subheap in the place of the extracted subheap.



Worst-case time: $\Theta(\lg n)$ with at most $3 \lg n + O(1)$ element comparisons

Meld(Q,R)

Assume that the sizes of Q and R are m and n, respectively, and that $m \leq n$.

1. Move all singleton and run-leader objects from the node store of Q to the node store of R.
2. Eject all perfect weak heaps of Q and store them to a temporary stack S_Q.
3. Eject all perfect weak heaps of R, whose height is no greater than $\lfloor \lg m \rfloor$, and store them to a temporary stack S_R.
4. Process all perfect weak heaps in S_Q and S_R in height order and inject them into the heap store of R.
5. Reduce the number of potential violation nodes in R by at most $\lfloor \lg m \rfloor$.
6. Destroy Q and return R.

Worst-case time: $\Theta(\lg m)$ with at most $5 \lg m$ element comparisons

Implementierung / Engineering

Rasmussen (2008) bietet im Technischen Report eine Implementierung an, die

- eine worst-case Schranke nicht genau trifft (heap store)
- Halb so schnell ist wie die von Fibonacci-Heaps in LEDA
- Fehlerhaft ist, eine beliebige Folge von insert, delete und deleteMins führt

Bruun, Edelkamp at al. (2010)

- beheben die Fehler von Rasmussen
- Doppelt so schnell wie Fibonacci-Heaps in LEDA
- relaxieren reduce von O(1) worst-case zu O(1) amortisiert
- Gewinnen Speicher (ein Pointer)

Neues reduce

```
while (leaders ∪ chairmen ≠ ∅)                                   ;; New loop
  if (chairmen ≠ ∅)                ;; New ordering: first fellows, then members
    first ← chairmen.first; firstparent ← parent(first)
    if (firstparent.left = first and marked(firstparent.right) or
      firstparent.left ≠ first and marked(firstparent.left))
        siblingtrans(firstparent); continue
    second ← chairmen.second; secondparent ← parent(second)
    if (secondparent.left = second and marked(secondparent.right) or
      secondparent.left ≠ second and marked(secondparent.left))
        siblingtrans(secondparent); continue
    if (firstparent.left = first) cleaningtrans(firstparent)
    if (secondparent.left = second) cleaningtrans(secondparent)
    if (marked(firstparent) or root(firstparent))
      parenttrans(firstparent); continue
    if (marked(secondparent) or root(secondparent))
      parenttrans(secondparent); continue
      pairtrans(firstparent, secondparent)
```

Neues Reduce (ctd.)

```
else if (leaders ≠ ∅)
  leader ← leaders.first ; leaderparent ← parent(leader)
  if (leader = leaderparent.right)
    parenttrans(leaderparent)
    if (¬marked(leaderparent) ∧  marked(leader))
      if (marked(leaderparent.left) siblingtrans(leaderparent); continue
      parenttrans(leaderparent)
    if (marked(leaderparent,right)) parenttrans(leader)
  else
    sibling ← leaderparent.right
    if (marked(sibling)) siblingtrans(leaderparent); continue
    cleaningtrans(leaderparent)
    if (chairmen)continue                                        ;; New case
    if (marked(sibling.right)) siblingtrans(sibling); continue
    cleaningtrans(sibling)
    parenttrans(sibling)
    if (marked(leaderparent.left)) siblingtrans(leaderparent)
```

Beobachtung

Durch while-Schleife, neuem Cut und Umordnung des Codes als Invariante:

- Weniger als 2 „leader“ und „chairmen“

➔ Bitvektorimplementierung anstatt STL-Listen
Implementierung mit Splice-Kommando

Problem: Finden des höchst-signifikanten Bits

Berechnung des Höchst-Signifikant Bits

- According to graphics.stanford.edu~/seander/bithacks.html there are some alternative options to quickly compute the most signicant bit in an unsigned int x,mostly based on considering x & -x.
- Options to identify the position of the bit in the result include converting it to a oat, a modulo computation, or a multiplication.
- We experimented with the latter and got slightly better results than with the 64K table with 65,536 entries denoting the most signicant bit of all 16-bit numbers
- The best option, however, is to use new processor technology, e.g. in the Barcelona AMD chips processors, which features the LZCNT command of SSE4a. In GCC (version 4.3 and later) there are aggressive optimization ags like -mtune=amdfam10 that allow to exploit its usage

Resultate

	$n = 25'000'000$			$n = 50'000'000$		
	Insert	*Dec.Key*	*Del.Min*	*Insert*	*Dec.Key*	*Del.Min*
Relaxed Weak Queues	0.048	0.223	4.38	0.049	0.223	5.09
Weak Heaps	0.047	0.047	1.30	0.047	0.047	1.85
Pairing Heaps	0.010	0.020	6.71	0.009	0.020	8.01
Fibonacci Heaps	0.062	0.116	6.98	-	-	-
Heaps	0.090	0.064	5.22	0.082	0.065	6.37

Table 4.1: Performance of priority queue data structures on n integers.

	$n = 5'000'000$			$n = 20'000'000$		
	Insert	*Dec.Key*	*Del.Min*	*Insert*	*Dec.Key*	*Del.Min*
Relaxed Weak Queues	0.334	1.910	7.50	0.390	1.986	9.92
Weak Heaps	0.692	1.288	6.70	0.779	1.372	8.49
Pairing Heap	0.262	1.002	8.99	0.302	1.043	12.51
Fibonacci Heap	0.388	1.042	12.12	0.439	1.097	16.24
Heaps	0.698	1.388	10.81	0.809	1.435	14.21

Table 4.2: Performance of priority queue data structures on n strings.