Algorithmentheorie

„Priority Queues“

Stefan Edelkamp

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}

Anwendungen „Vorrangwarteschlange“

- Sortieren (wie in Heapsort)
- Kürzeste Wege Suche (Single Source Shortest Path) mit Dijkstra‘s Algorithmus oder A*
  - DeleteMin entnimmt zu expandierenden Knoten
  - DecreaseKey aktualisiert gemäß Relaxierungsoperation
  - Insert fügt ein, falls Knoten neu
- Minimaler Spannbaum via Kruskal‘s Algorithmus. (Algorithmus von Prim nutzt Union/Find Struktur)
- …

Übersicht

- 1-Level Buckets
- 2-Level Buckets
- Radix Heaps
- Ende-Boas
- Balancierte Suchbäume (z.B. AVL)
- Heaps & Weak-Heaps
- Binomial Queues & Fibonacci-Heaps
- Run-Relaxed Weak-Queues

1-Level Buckets

- The i-th bucket contains all elements with a f-value equal to i.
- With the array we now associate three numbers *minVal, minPos and n:*
- - *minVal denotes the smallest f value in the queue,*
- - n the number of elements and
- - *minPos fixes the index of the bucket with the smallest key.*
- The i-th bucket b[i] contains all elements v with
- f(v) = *minVal+(i − minPos) mod C.*

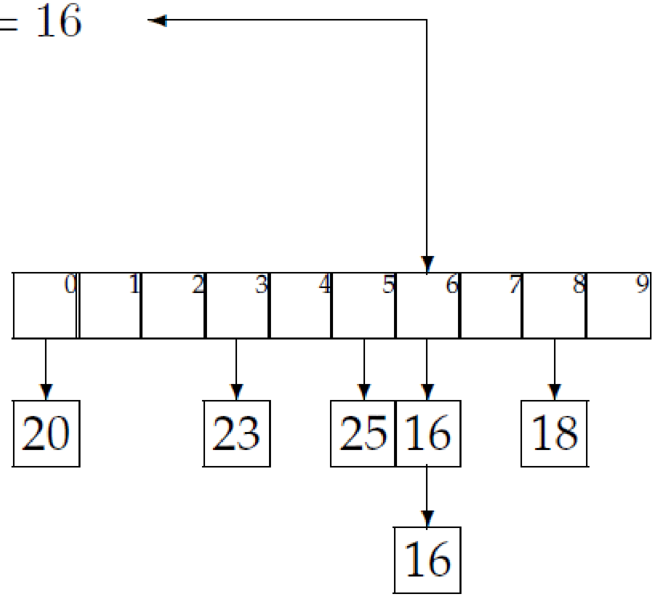
Beispiel

$C = 9$

$\text{minValue} = 16$

$minPos = 6$

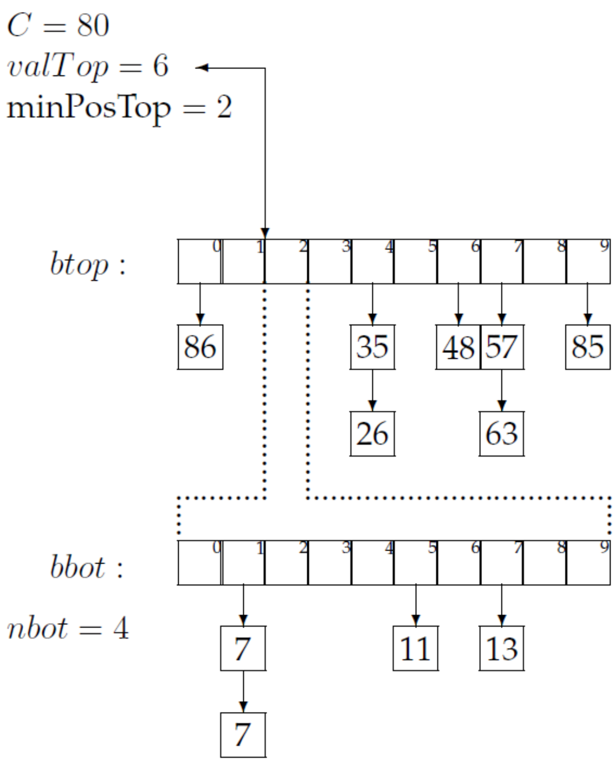
$n = 6$



2-Level Buckets

- Goal: Reduce worst case complexity O(C) for *DeleteMin to O(sqrt(*C))
- Top level and bottom level both of length ceil(sqrt(C +1)+1).
- The bottom level refines the smallest bucket of the *minPosTop in the top level.*
- Lower level buckets created only when the current bucket at *MinPosTop becomes* empty
- Refinements include an involved k-level bucket architecture.

Beispiel


$C = 80$
$valTop = 6$
$\text{minPosTop} = 2$
$btop$:
0 1 2 3 4 5 6 7 8 9
86
35
48
57
85
26
63
$bbot$:
$nbot = 4$
0 1 2 3 4 5 6 7 8 9
7
11
13
7

Pseudo Code

Procedure Initialize
Input: 1-Level Bucket array $b[0..C]$ (implicit constant C)
Side Effect: Updated 1-Level Bucket $b[0..C]$

$n \leftarrow 0$;; No element in so far
minValue $\leftarrow \infty$;; Default value for current minimum

Algorithm 4.1: Initializing an 1-Level Bucket.

Procedure Insert
Input: 1-Level Bucket $b[0..C]$, element x with key k
Side Effect: Updated 1-Level Bucket $b[0..C]$

$n \leftarrow n + 1$;; Increase number of elements
if ($k <$ *minValue*) ;; Element with smallest key
    minPos $\leftarrow k$ *mod* $(C + 1)$;; Update location of minimum
    minValue $\leftarrow k$;; Update current minimum
Insert x in $b[k$ *mod* $(C + 1)]$;; Insert into list

Algorithm 4.2: Inserting an element into an 1-Level Bucket.

Pseudo Code

Procedure DeleteMin
Input: 1-LEVEL BUCKET $b[0..C]$
Output: Element x with key *minPos*
Side Effect: Updated 1-LEVEL BUCKET $b[0..C]$

```
Remove x in b[minPos] from doubly-ended list            ;; Eliminate element
n ← n − 1                                               ;; Decrease number of elements
if (n > 0)                                              ;; Structure non-empty
  while (b[minPos] = ∅)                                 ;; Bridge possible gaps
    minPos ← (minPos + 1) mod (C + 1)                   ;; Update location of pointer
  minValue ← Key(x), x ∈ b[minPos]                      ;; Update current minimum
else minValue ← ∞                                       ;; Structure empty
return x                                                ;; Feedback result
```

Algorithm 4.3: Deleting the minimum element in an 1-LEVEL BUCKET.

Procedure DecreaseKey
Input: 1-LEVEL BUCKET $b[0..C]$, element x, key k
Side Effect: Updated 1-LEVEL BUCKET $b[0..C]$ with x moved

```
Remove x from doubly-ended list                         ;; Eliminate element
n ← n − 1                                               ;; Decrease number of elements
Insert x with key k in b                                ;; Re-insert element
```

Algorithm 4.4: Updating the key in an 1-LEVEL BUCKET.

Amortisierte Analyse

Amortized complexity analysis disinguishes between:

- t_l, the real cost for operation l,

- Φ_l, the potential after execution operation l, and

- a_l, the amortized costs for operation l

We have $a_l = t_l + \Phi_l - \Phi_{l-1}$, so that

$$\sum_{l=1}^{m} a_l \;=\; \sum_{l=1}^{m} t_l + \Phi_l - \Phi_{l-1} = \sum_{l=1}^{m} t_l - \Phi_0 + \Phi_m$$

and

$$\sum_{l=1}^{m} t_l \;=\; \sum_{l=1}^{m} a_l + \Phi_0 - \Phi_m \leq \sum_{l=1}^{m} a_l$$

Hier

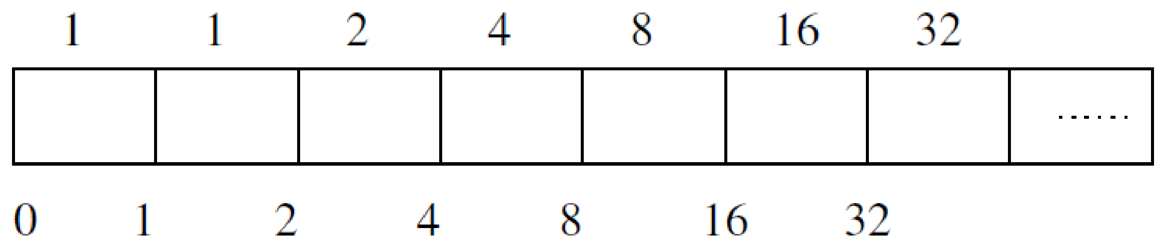
Let Φ_l be the number of elements in the top level bucket for the l-th operation, then

- *DeleteMin* uses $O(\sqrt{C} + m_l)$ time in the worst-case, where m_l is the number of elements that move from top to bottom

By amortization we have $O(\sqrt{C} + m_l + (\Phi_l - \Phi_{l-1})) = O(\sqrt{C})$ operations.

- Both operations *Insert* and *DecreaseKey* run in $O(1)$.

$\Rightarrow$ Dijkstra/A* results in $O(e + n\sqrt{C})$ worst-case run time



Radix Heaps

Radix-heaps maintain a list of $\lceil \log(C+1) \rceil + 1$ buckets of sizes 1, 1, 2, 4, 8, 16, etc.

We maintain buckets $b[0..B]$ and bounds $u[0..B+1]$ with $B = \lceil \log(C+1) \rceil + 1$ and $u[B+1] = \infty$

Bucket number $\phi(x)$ denotes the index of the actual bucket for x.

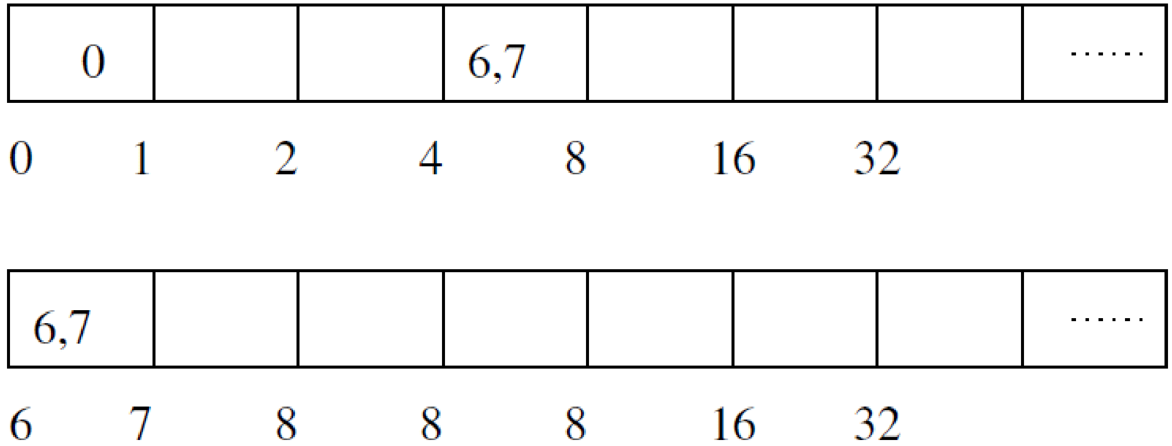
Invariants:

$i)$ all keys in $b[i]$ are in $[u[i], u[i+1]]$,

$ii)$ $u[1] = u[0] + 1$, and

$iii)$ for all $i \in \{1, \ldots, B-1\}$ we have $0 \leq u[i+1] - u[i] \leq 2^{i-1}$.

Beispiel



- Given radix heap (written as [u[i]] : b[i]):
- [0] : {0}, [1] : {} [2] : {} [4] : {6, 7}, [8] : {}, [16] : {}.
- Extracting key 0 from bucket 1 yields [6] : {6, 7}, [7] : {}, [8] : {}, [8] : {}, [8] : {},[16] : {}.
- Now, key 6 and 7 are distributed.
- - if b[i] <> {} then the interval size is at most 2^{i−1}.
- - for b[i] we have i − 1 buckets available.
- Since all keys in b[i] are in [k, min{k +2^{i−1} − 1, u[i+1] − 1}] all elements fit into b[0], . . . , b[i − 1].

Operationen

- - *Initialize generates empty buckets and bounds:*

  for i in {2, . . . ,B} set u[i] to u[i − 1]+2^{i−2}.
- - *Insert(x) performs linear scan for bucket i, starting from i = B. Then the new* element x with key k is inserted into b[i], with i = max{j | k <= u[j]]}
- - For *DecreaseKey, bucket i for element x is searched linearly from the actual* bucket i for x.
- - For *DeleteMin we first search for the first non-empty bucket i = min{j | b[j] <> {}}* and identify the element with minimum key k therein.

DeleteMin (cont.)

- If the smallest bucket contains more than an element, it is returned
- If the smallest bucket contains no element
- - u[0] is set to k, u[1] is set to k +1 and for j > 2 bound u[j] is set to min{u[j − 2]+2^{j−2}, u[i+1]}.
- - The elements of b[i] are distributed to buckets b[0], b[1], . . . , b[i − 1] and the minimum element is extracted from the non-empty smallest bucket.

Pseudo Code

Procedure Initialize
Input: Array $b[0..B]$ of lists and array $u[0..B]$ of bounds
Side Efect: Initialized RADIX HEAP with arrays b and u

for each i **in** $\{0, \ldots, B\}$ $b[i] \leftarrow \emptyset$	;; Initialize buckets
$u[0] \leftarrow 0; u[1] \leftarrow 1$	;; Initialize bounds
for each i **in** $\{2, \ldots, B\}$ $u[i] \leftarrow u[i-1] + 2^{i-2}$	;; Initialize bounds

Algorithm 4.5: Creating a RADIX HEAP.

Procedure Insert
Input: RADIX HEAP with array $b[0..B+1]$ of lists and array $u[0..B+1]$, key k
Side Effect: Updated RADIX HEAP

$i \leftarrow B$	;; Initialize index
while $(u[i] > k)$ $i \leftarrow i - 1$	;; Decrease index
Insert k in $b[i]$	;; Insert element in list

Algorithm 4.6: Inserting an element into a RADIX HEAP.

Pseudo Code

Procedure DecreaseKey
Input: RADIX HEAP with array $b[0..B+1]$ of lists and array $u[0..B+1]$
Index i in which old key k is stored, new key k'
Side Effect: Updated RADIX HEAP

```
while (u[i] > k') i ← i − 1          ;; Decrease index
Insert k' in b[i]                     ;; Insert element in list
```

Procedure DecreaseMin
Input: RADIX HEAP with array $b[0..B+1]$ of lists and array $u[0..B+1]$
Output: Minimum element
Side Effect: Updated RADIX HEAP

```
i ← 0                                          ;; Start with first bucket
r ← Select(b[i])                               ;; Select (any) minimum key
b[i] ← b[i] \ {r}                              ;; Eliminate minimum key
while (b[i] = ∅) i ← i + 1                     ;; Search for first non-empty bucked
if (i > 0)                                     ;; First bucket empty
  k ← min b[i]                                 ;; Select miniumum key
  u[0] ← k, u[1] ← k + 1                       ;; Update bounds
  for each j in {2, ..., i}                    ;; Loop on array indices
    u[j] ← min{u[j − 1] + 2^(j−2), u[i + 1]}   ;; Update bounds
  j ← 0                                        ;; Initialize index
  for each k in b[i]                           ;; Keys to distribute
    while (k > u[j + 1]) j ← j + 1             ;; Increase index
    b[j] ← b[j] ∪ {k}                          ;; Distribute
  return r                                     ;; Output minimum element
```

Algorithm 4.8: Delete the minimum from a RADIX HEAP.

Amortisierte Analyse

Potential $\Phi_l = \sum_{x \in \textit{Radix-Heap}} \phi_l(x)$ for operation l.

- *Initialize* and *Insert* run in $O(B)$.

- *DecreaseKey* has an amortized time complexity in
$O(\phi_l(x) - \phi_{l-1}(x)) + 1 + (\Phi_l - \Phi_{l-1}) =$
$O((\phi_l(x) - \phi_{l-1}(x)) - (\phi_l(x) - \phi_{l-1}(x)) + 1) = O(1)$, and

- *DeleteMin* runs in time
$O(B + (\sum_{x \in b[i]} \phi_l(x) - \sum_{x \in b[i]} \phi_{l-1}(x)) + (\Phi_l + \Phi_{l-1})) = O(1)$ amortized.

$\Rightarrow O(m \log C + l)$ for m Insert and l *DecreaseKey* and *ExtractMin* operations.

$\Rightarrow$ Dijkstra/A* runs in time $O(e + n \log C)$.

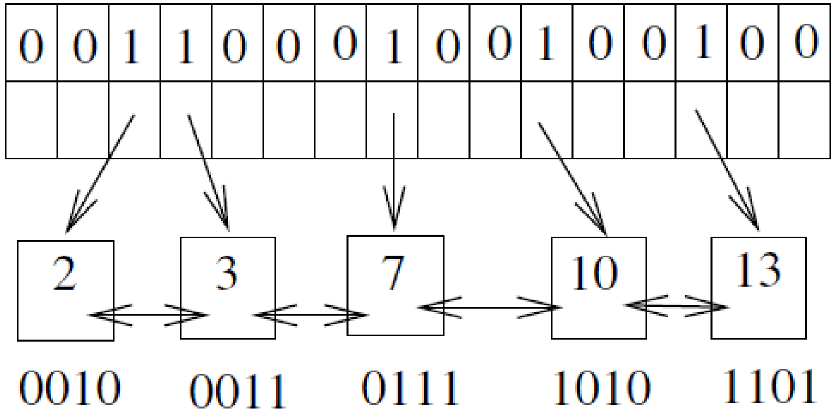
Van-Emde-Boas

- Assumes a universe $U = \{0, . . . , N − 1\}$ of keys for S
- All priority queue operations reduce to the successor calculation which runs in $O(\log \log N)$ time.
- The space requirements are $O(N \log \log N)$.

k-Struktur T besteht aus

1. a number m = |S|,
2. a doubly-connected list, which contains all elements of S in increasing order,
3. a bit vector b[0..2^k − 1], with b[i] = *true if and only if i in S,*
4. a pointer array p, with p[i] pointing to key i in the linked list if b[i] = *true,*
5. a k' = ceil(k/2)-structure *top and a field bottom[0..2^k'−1].*

- If m = 1, then *top and bottom are not needed;*
- for m > 1 *top is a k'-structure with the prefix bit elements ceil(x/2^k'')* for x in S and k'' = ceil(k/2), and each *bottom[x], is a k''-structure containing the* matching suffix bit elements x *mod 2^k'' for x in S.*



Beispiel

- For the example k = 4, S = {2, 3, 7, 10, 13} and m = 5
- - *top is a 2-structure on {0, 1, 2, 3} and*
- - *bottom is a vector of 2-structures with*
- *bottom[0] = {2, 3}, bottom[1] = {3},*
- *bottom[2] = {2}, and bottom[3] = {1},*
- since 2 = 00|10, 3 = 00|11, 7 = 01|11, 10 = 10|10, and 13 = 11|01.

Operation Succc

- *succ(x) finds min{y in S | y > x} in the k-structure T.*
- If the *top-bit at position x' = ceil(x/2^k'')* is set
- ➔ return (x‘ · 2^k“)+*bottom[x].*
- Otherwise let z' = *succ(x', top)*
- ➔ return z‘ · 2^k“ +min{*bottom[z‘]*}*.*
- By the recursion we have T(k) <= c+T(ceil(k/2)) = O(log k), so that we can determine the sucessor in O(log log N) time.

Operationen Insert und Delete

- *Insertion for x in T determines the successor succ(x) of x, computes* x‘ = ceil(x/2^k“) and x’’ = *mod 2^k’’*
  - It divides into the calls *insert(x’, top) and insert(x’’,bottom[x’’]).*
  - Integration the computation in a recursive scheme leads a running time of O(log log N).
- *Deletion used the doubly-linked structure and the successor relation and also runs* in O(log logN) time.

Platzbedarf einer k-Struktur

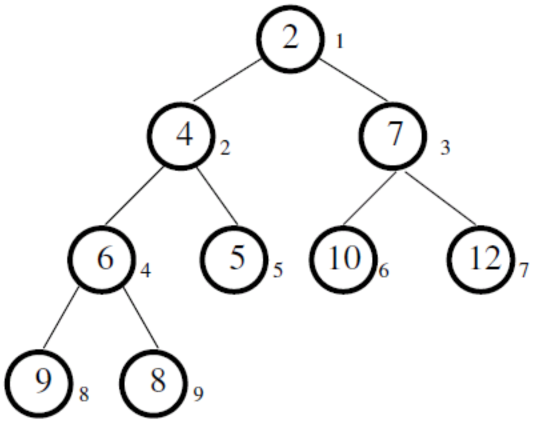
For $s(k)$ we have $s(1) = c$, and $s(k) \leq c2^k + s(k/2) + 2^{k/2}s(k/2)$.

We inductively assume $s(k) \leq c'2^k \log k$. For $k = 1$ there is nothing to show.

$$\begin{aligned}
s(k) &\leq c2^k + c'2^{k/2}(\log k - 1) + 2^{k/2}c'2^{k/2}(\log k - 1) \\
&= c2^k + c'2^{k/2}(1 + 2^{k/2})(\log k - 1) \\
&= c2^k + c'2^{k/2}(2^{k/2}\log k - 2^{k/2} + \log k - 1) \\
&\leq c2^k + c'2^{k/2}(2^{k/2}\log k - 2^{k/2} + \log k) \\
&\leq c'2^k \log k.
\end{aligned}$$

if $c2^k + c'2^{k/2}(2^{k/2}\log k - 2^{k/2}) + \log k) \leq c'2^k \log k$. This is equivalent with $c'2^{k/2}\log k \leq (c' - c)2^k$ and $(c' - c)/c \geq \log k/2^k$, which is true for large c'.

Bitvektor und Heap



- *Dijkstra's original implementation: reduces to a bitvector* indicating if elements are currently *open or not.*
- The minimum is found by a complete scan yielding O(n^2) time.
- *Heap implementation with in array implementation with A[i] > A[i/2] for all i > 1* leads to an O((e+n) log n) shortest path algorithm
- - *DeleteMin implemented as in Heapsort,*
- - *Insert at the end of the array, followed by a sift-up*
- *Dynamics: growing and shrinking heaps base on dynamic tables/arrays.*

Pairing Heaps

- A pairing heap is a heap-ordered (not necessarily binary) self-adjusting tree.
- The basic operation on a pairing heap is pairing, which combines two pairing heaps by attaching the root with the larger key to the other root as its leftmost child.
- More precisely, for two pairing heaps with respective root values k1 and k2, pairing inserts the first as the leftmost subtree of second if k1 > k2, and otherwise inserts the second into the first as its leftmost subtree. Pairing takes constant time and the minimum is found at the root.

„Multiple-Child“ Implementierung

In a heap-ordered multi-way tree representation realizing the priority queue operations is simple.

- Insertion pairs the new node with the root of heap.
- DecreaseKey splits the node and its subtree from the heap (if the node is not the root), decreases the key, and then pairs it with the root of the heap.
- Delete splits the node to be deleted and its subtree, performs a DeleteMin on the subtree, and pairs the resulting tree with the root of the heap.
- DeleteMin removes and returns the root, and then, in pairs, pairs the remaining trees. Then, the remaining trees from right to left are incrementally paired.

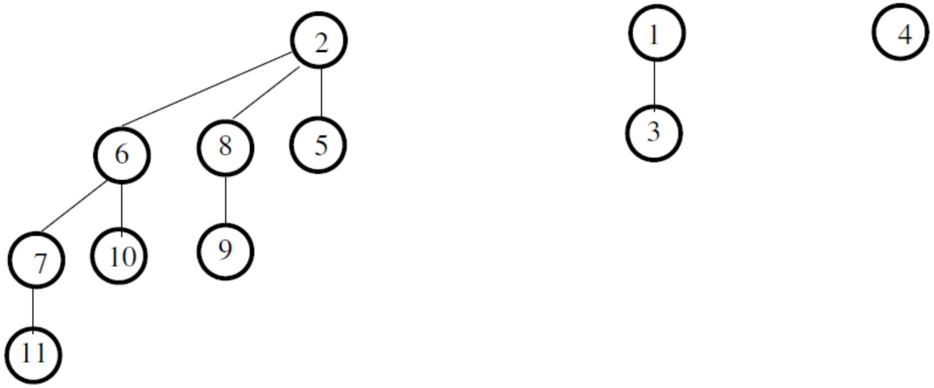
„Child-Sibling" Implementierung

- Since the multiple child representation is difficult to maintain, the child-sibling binary tree representation for pairing heaps is often used, in which siblings are connected as follows.
- The left link of a node accesses its first child, and the right link of a node accesses its next sibling, so that the value of a node is less than or equal to all the values of nodes in its left subtree.
- It has been shown that in this representation insert takes O(1) and delete-min takes O(log n) amortized, while decrease-key takes at least Omega(log log n) steps.

Fibonacci Heaps

- *Fibonacci-heaps are lazy-meld versions on of binomial queues that base on binomial trees.*
- A *binomial tree Bn is a tree of height n with 2^n nodes in total and (n choose i)* nodes in depth i.
- The structure of Bn is given by unifying two structure Bn−1, where one is added as an additional successor to
- In *Fibonacci-Heaps*
- \- *DecreaseKey runs in O(1) amortized*
- \- *DeleteMin runs in O(log n) amortized*

Binomial Queues



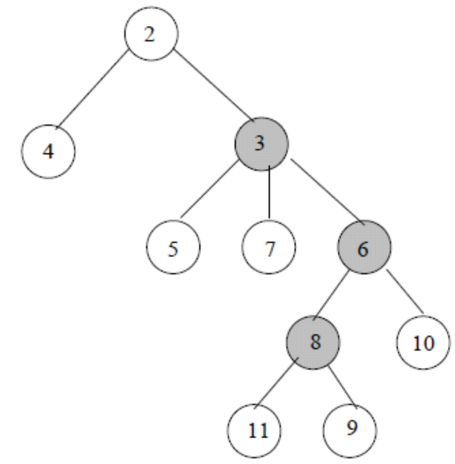
- *Binomial-queues are a union of heap-ordered binomial trees.*
- Tree Bi is represented in queue Q if the ith bit in the binary representation of n is set.
- The partition of structure Q into trees Bi is unique.
- \- *Min takes O(log n) time, since the minimum is always located at the root of one* Bi,
- \- Binomial queues Q1 and Q2 of sizes n1 and n2 are *meld by simulating binary* addition of n1 and n2 in their dual representation.
- This corresponds to a parallel scan of the root lists of Q1 and Q2. If n ~ n1 +n2 then the meld can be performed in time O(log n) time.

Andere Operationen

- \- Operations *Insert and DeleteMin both use procedure meld as a subroutine.*
- The former creates a tree B_0 with one element, while the latter extracts tree B_i containing the minimal element and splits it into its subtrees B_0, . . . ,B_{i−1}.
- In both cases the resulting trees are merged with the remaining queue to perform the update.
- \- *DecreseKey for element v updates the heap-ordered tree Bi in which v is located* by sifting the element.
- All operations run in O(log n) time.

Fibonacci-Heaps

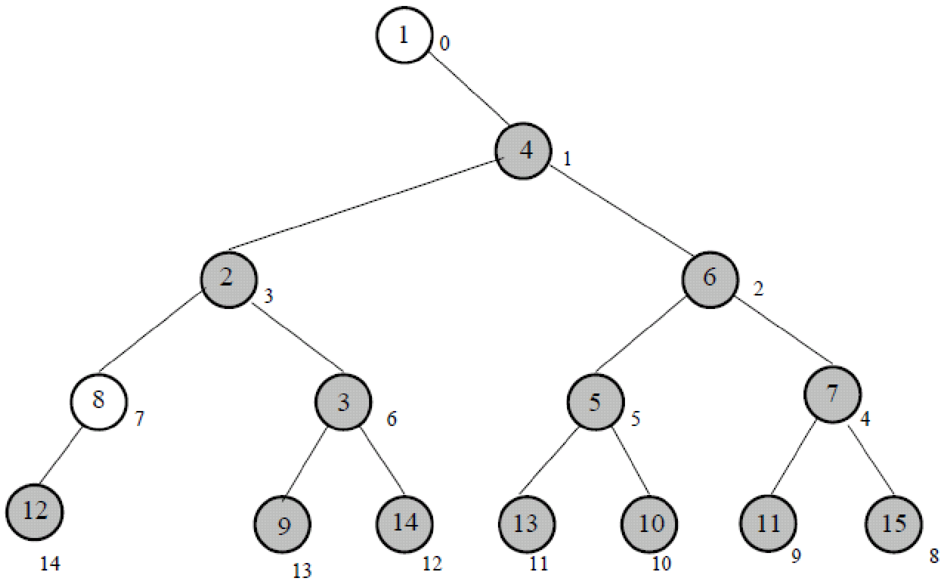
- Collection of heap-ordered binomial trees, maintained in form a circular doubly-connected unordered list of root nodes.
- In difference to binomial queues, more than one binomial tree of rang i may be represented.
- However, after performing a *consolidate operation that traverses the linear list and* merges trees of the same rang, each rang will become unique.
- For this purpose an additional array of size at most 2 log n is devised that supports finding the trees of same rang in the root list.



Operationen

- *- Min is accessible in O(1) time through a pointer in the root list.*
- *- Insert performs a meld operation with a singleton tree.*
- *- DeleteMin extracts the minimum and includes all subtrees into the root list. In this* case, *consolidation is mandatory.*
- *- DecreaseKey performs the update on the element in the heap-ordered tree. It* removes the updated node from the child list of its parent and inserts it into the root list, while updating the minimum.
- To assert amortized constant run time, selected nodes are marked to perform *cascading cuts, where a cascading cut is a cut operation propagated to the parent* node.

Weak-Heaps



- *- DeleteMin: Similar to Weak-Heapsort*
- *- Insert: Climb up the grandparents until the definition is fulfilled.*
- On the average the path length of grandparents from a leaf node to a root is approximately half the depth of the tree.
- *- DecreaseKey: start at the node x that has changed its value.*

Pseudo Code

Procedure DeleteMin
Input: WEAK HEAP of size n
Output: Minimum element
Side Effect: Updated WEAK HEAP of size $n-1$

```
Swap(A[0], A[n-1])        ;; Swap last element to root position
Merge-Forest(0)           ;; Restore WEAK HEAP property
n ← n - 1                 ;; Decrease size
return A[n]               ;; Return minimum element
```

Algorithm 4.18: Extracting the minimum element from a WEAK HEAP.

Pseudo Code

Procedure Insert
Input: Key k, WEAK HEAP of size n
Side Effect: Updated WEAK HEAP of size $n+1$

```
A[n] ← k; x ← n                                      ;; Place element at empty place at end of array
Reverse[x] ← 0                                       ;; Initialize bit
while (x ≠ 0) and (A[Grandparent(x)] > A[x])         ;; Unless finished or root node found
    Swap(Grandparent(x), x)                          ;; Exchange keys
    Reverse[x] ← ¬Reverse[x]                         ;; Rotate subtree rooted at x
    x ← Grandparent(x)                               ;; Climb up structure
n ← n + 1                                            ;; Increase size
```

Algorithm 4.19: Inserting an element into a WEAK HEAP.

Procedure DecreaseKey
Input: WEAK HEAP, index x of element that has improved to k
Side Effect: Updated WEAK HEAP

```
A[x] ← k                                             ;; Update key value
while (x ≠ 0) and (A[Grandparent(x)] > A[x])         ;; Unless finished or root node found
    Swap(Grandparent(x), x)                          ;; Exchange keys
    Reverse[x] ← ¬Reverse[x]                         ;; Rotate subtree rooted at x
    x ← Grandparent(x)                               ;; Climb up structure
```

Algorithm 4.20: Decreasing the key of an element in a WEAK HEAP.

Run-Relaxed Weak Queues

➔ Originalfolien
von Elmasry et al. (2008)

Procedure λ**-Reduce**
Side Effect: Relaxed Weak Queue structure modified

```
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
else 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
```

Algorithm 4.22: Reducing number of marked nodes in a Relaxed Weak Queue.

Engineering

	$n = 25'000'000$			$n = 50'000'000$		
	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.