(A Survey on)
Priority Queues

Gerth Stølting Brodal
Aarhus Universty

MADALGO
CENTER FOR MASSIVE DATA ALGORITHMICS

IanFest, University of Waterloo, Waterloo, Ontario, Canada, August 14-15, 2013

About Dagstuhl | Program | Publications | Library

You are here: Program » Calendar » Seminar Homepage

- Dagstuhl Seminars
- Dagstuhl Perspectives
- GI-Dagstuhl Seminars
- Events
- Research Guests
- **Calendar**
  - Seminars
  - Events

http://www.dagstuhl.de/9609

February 26 to March 1, 1996, Dagstuhl Seminar 9609

Data Structures

Organizers

S. Näher, H. Noltemeier, I. Munro

For support, please contact

Dagstuhl Service Team

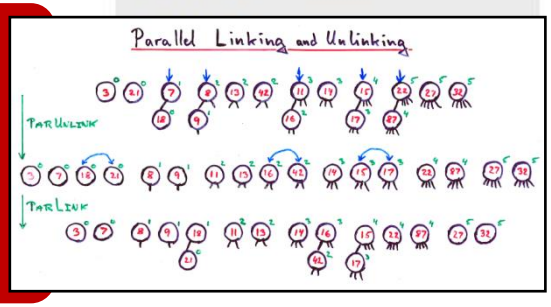
Documents

List of Participants

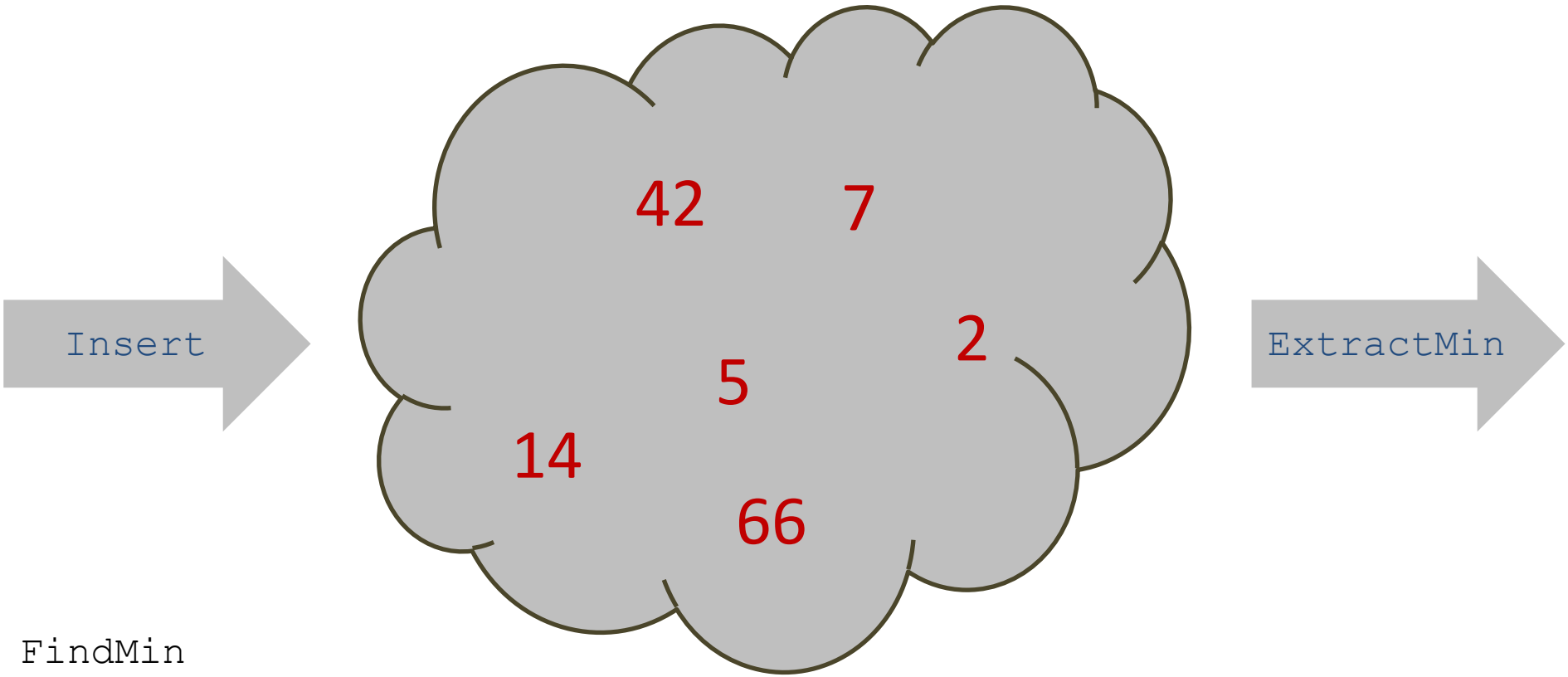
Program

Ian Munro : A Constant Time Priority Queue

Gerth Brodal : A Constant Time Priority Queue



Priority Queue

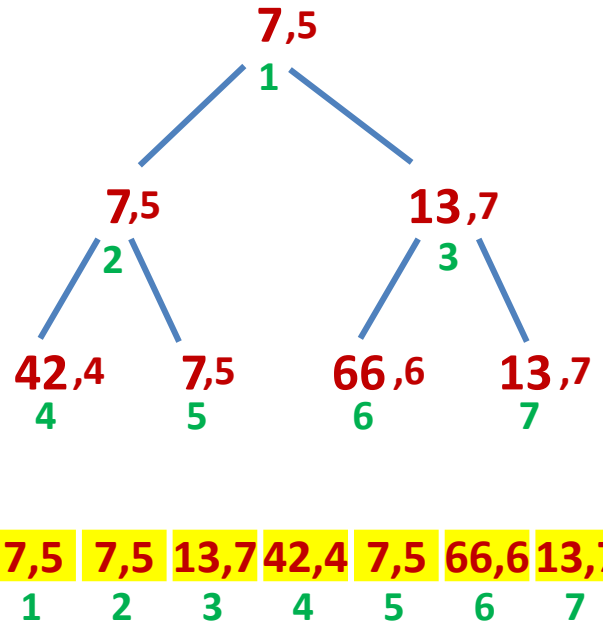


`FindMin`
`Delete`
`DecreaseKey`
`Meld`
`ExtractMax` (double-ended priority queue)

ALGORITHM 113
TREESORT
ROBERT W. FLOYD
Computer Associates, Inc., Woburn, Mass.

```
procedure TREESORT (UNSORTED, n, SORTED, k); value
  n, k;
integer n, k; array UNSORTED, SORTED;
comment TREESORT sorts the smallest k elements of the n-
  component array UNSORTED into the k-component array
  SORTED (the two arrays may be the same). The number of
  operations is on the order of 2 × n + k × log2(n). The number
  of auxiliary storage cells required is on the order of 2 × n. It is
  assumed that procedures are available for finding the minimum
  of two quantities, for packing one real number and one integer
  into a word, and for obtaining the left and right half of a packed
  word. The value of infinity is assumed to be larger than that of
  any element of UNSORTED;
begin integer i, j; array m[1:2 × n − 1];
for i := 1 step 1 until n do m[n + i − 1] := pack (UNSORTED
  [i], n + i − 1);
for i := n − 1 step − 1 until 1 do m[i] := minimum (m[2 × i],
  m[2 × i + 1]);
for j := 1 step 1 until k do
  begin SORTED [j] := left half (m[1]); i := right half (m[1]);
    m[i] := infinity;
  for i := i ÷ 2 while i > 0 do m[i] := minimum (m[2 × i], m[2 ×
    i + 1])
  end
end TREESORT
```

Communications of the ACM (1962)



7,5	7,5	13,7	42,4	7,5	66,6	13,7
1	2	3	4	5	6	7

ALGORITHM 232
HEAPSORT

J. W. J. Williams (Recd 1 Oct. 1963 and, revised, 15 Feb. 1964)
Elliott Bros. (London) Ltd., Borehamwood, Herts, England

comment The following procedures are related to *TREESORT* [R. W. Floyd, Alg. 113, *Comm. ACM 5* (Aug. 1962), 434, and A. F. Kaupe, Jr., Alg. 143 and 144, *Comm. ACM 5* (Dec. 1962), 604] but avoid the use of pointers and so preserve storage space. All the procedures operate on single word items, stored as elements 1 to n of the array A. The elements are normally so arranged that $A[i] \leqq A[j]$ for $2 \leqq j \leqq n$, $i = j \div 2$. Such an arrangement will be called a heap. $A[1]$ is always the least element of the heap.

The procedure *SETHEAP* arranges n elements as a heap, *INHEAP* adds a new element to an existing heap, *OUTHEAP* extracts the least element from a heap, and *SWOPHEAP* is effectively the result of *INHEAP* followed by *OUTHEAP*. In all cases the array A contains elements arranged as a heap on exit.

SWOPHEAP is essentially the same as the tournament sort described by K. E. Iverson—*A Programming Language*, 1962, pp. 223–226—which is a top to bottom method, but it uses an improved storage allocation and initialisation. *INHEAP* resembles *TREESORT* in being a bottom to top method. *HEAPSORT* can thus be considered as a marriage of these two methods.

The procedures may be used for replacement-selection sorting, for sorting the elements of an array, or for choosing the current minimum of any set of items to which new items are added from time to time. The procedures are the more useful because the active elements of the array are maintained densely packed, as elements $A[1]$ to $A[n]$;

```
procedure SWOPHEAP (A,n,in,out);
  value in,n; integer n; real in,out; real array A;
  comment SWOPHEAP is given an array A, elements A[1]
    to A[n] forming a heap, n≧0. SWOPHEAP effectively adds
    the element in to the heap, extracts and assigns to out
    the value of the least member of the resulting set, and leaves
```

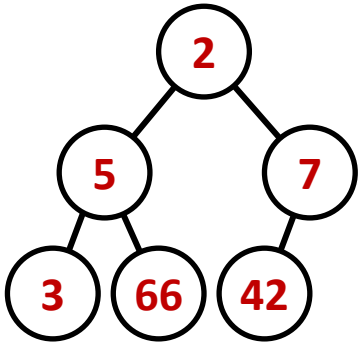
ALGORITHM 245
TREESORT 3 [M1]

Robert W. Floyd (Recd. 22 June 1964 and 17 Aug. 1964)
Computer Associates, Inc., Wakefield, Mass.

```
procedure TREESORT 3 (M, n);
  value n; array M; integer n;
comment TREESORT 3 is a major revision of TREESORT
  [R. W. Floyd, Alg. 113, Comm. ACM 5 (Aug. 1962), 434] suggested by HEAPSORT [J. W. J. Williams, Alg. 232, Comm.
  ACM 7 (June 1964), 347] from which it differs in being an in-place
  sort. It is shorter and probably faster, requiring fewer comparisons and only one division. It sorts the array M[1:n], requiring
  no more than 2 × (2↑p−2) × (p−1), or approximately 2 ×
  n × (log₂(n)−1) comparisons and half as many exchanges in
  the worst case to sort n = 2↑p − 1 items. The algorithm is
  most easily followed if M is thought of as a tree, with M[j÷2]
  the father of M[j] for 1 < j ≦ n;
begin
  procedure exchange (x,y); real x,y;
    begin real t; t := x; x := y; y := t
    end exchange;
  procedure siftup (i,n); value i, n; integer i, n;
  comment M[i] is moved upward in the subtree of M[1:n] of
    which it is the root;
  begin real copy; integer j;
    copy := M[i];
  loop: j := 2 × i;
    if j ≦ n then
    begin if j < n then
        begin if M[j+1] > M[j] then j := j + 1 end;
      if M[j] > copy then
        begin M[i] := M[j]; i := j; go to loop end
    end;
    M[i] := copy
  end siftup;
  integer i;
  for i := n÷2 step −1 until 2 do siftup (i,n);
  for i := n step −1 until 2 do
  begin siftup (1,i);
    comment M[j÷2] ≧ M[j] for 1 < j ≦ i;
    exchange (M[1], M[i]);
    comment M[i:n] is fully sorted;
  end
end TREESORT 3
```

Communications of the ACM (1964)

Heap



2	5	7	3	66	42
1	2	3	4	5	6

- *Simple*
- Implicit – single array of size n
- `Insert` and `ExtractMin` O(log n) — Willliams (1964)
- Construction O(n) — Floyd(1964)
- Insert Θ(loglog n) comparisons — Gonnet, Munro (1986)
- Select k smallest in O(k) time — Frederickson (1993)

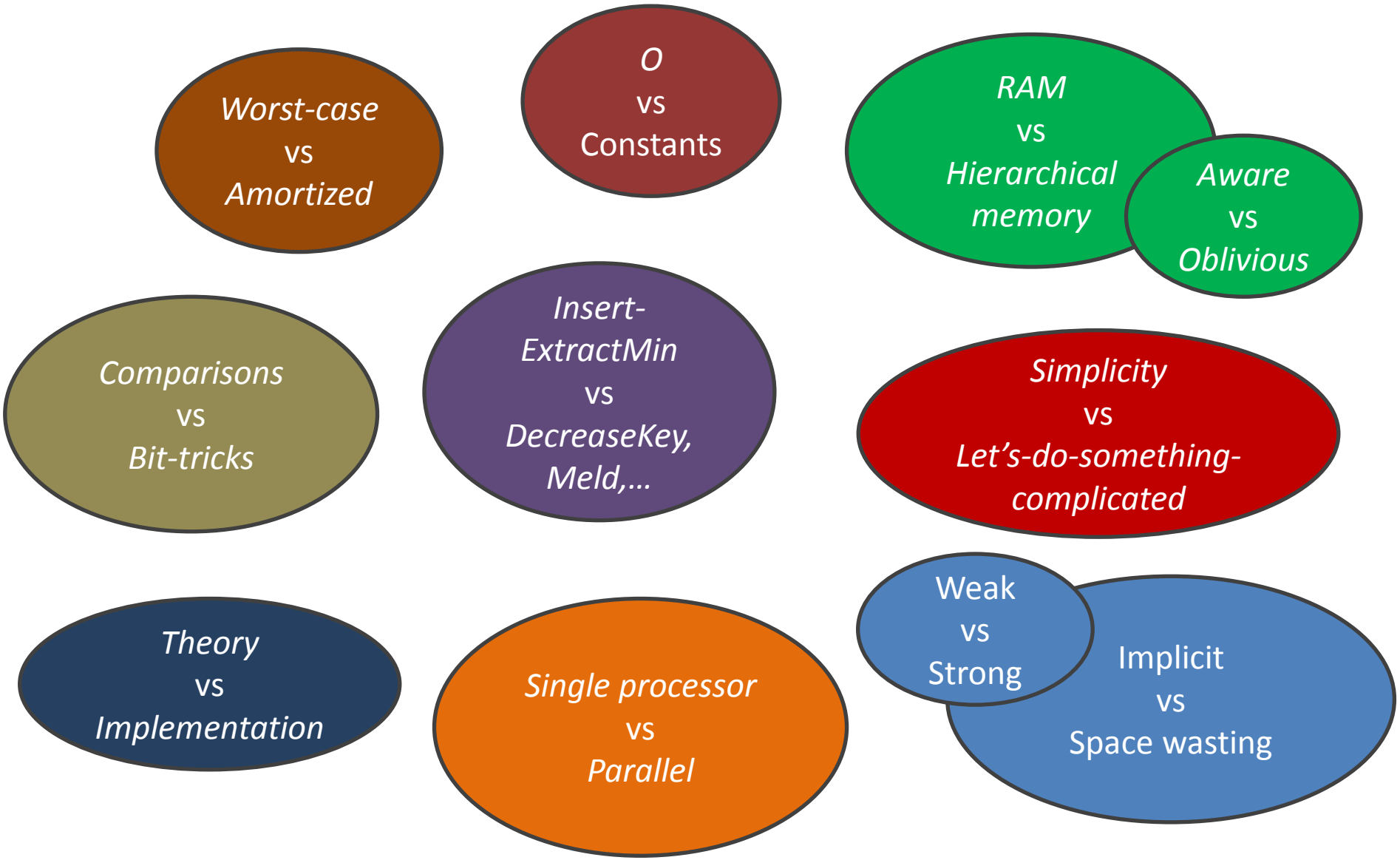
Comparison Lower Bound

n x `Insert` + n x `ExtractMin` $\Rightarrow$ Sorting

$\Rightarrow$ `Insert` or `ExtractMin` $\Omega(\log n)$ comparisons

	Comparisons	
	`Insert`	`ExtractMin`
Heap	O(log n)	O(log n)
Binary search	O(log n)	0
Carlsson, Munro, Poblette (1988)	O(1)	O(log n)

Priority Queues – Directions of Research



Open Problem

Can only
store n + array
between operations

Strongly implicit
priority queue
worst-case
O(1) `Insert` and O(log n) `ExtractMin`
supporting identical elements
O(1) swaps per operation
and cache oblivious optimal ?

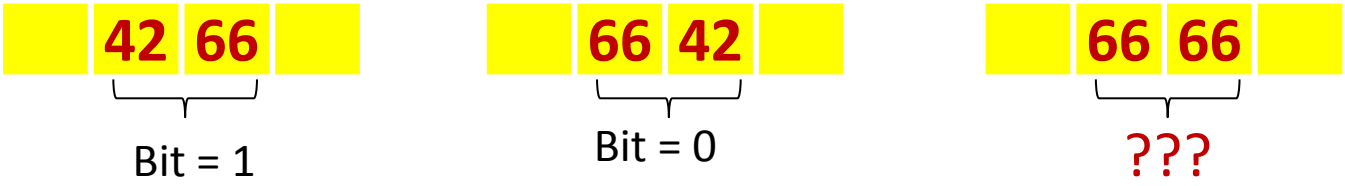
(Some Random) Results

♣ amortized bounds	`Insert`	`ExtractMin`	Implicit	Swaps	Identical elements	Cache oblivious
Heaps	log n	log n	Strong	log n	Yes	
Carlsson, Munro, Poblette (1988)	1	log *n*	Weak	log *n*	Yes	
Arge, Bender, Demaine, Holland-Minkley, Munro (2002)	♣log *n*	♣log *n*		♣log *n*	Yes	Yes
Munro, Franceschini (2006)	♣log *n*	♣log *n*	Strong	♣1		
Harvey, Zatloukal (200)	♣1	♣log *n*	Strong	log *n*	Yes	
Brodal, Nielsen, Tru n (2013)	1	log *n*	Strong	log *n*		
Open	1	log *n*	Strong	1	Yes	Yes
	x	x				x
	x	x		x		

Implicit dictionary
O(√n) Munro, Suwanda (1980)
O(log² *n*) Munro (1986)
O(log² *n*/loglog *n*) Franceschini, Grossi, Munro, Pagli (2004)

Some Observations

Pair encoding: Munro (1986)



$\Rightarrow$ 2log n elements can encode a pointer

$\Rightarrow$ O(log n) time allows pointer manipulations

Strict implicit + O(1) insertions

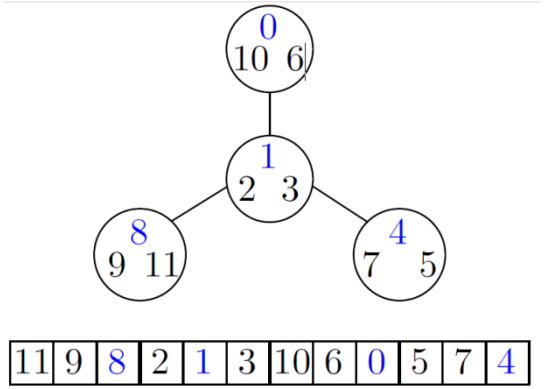
$\Rightarrow$ Insertions ”close to oblivious”

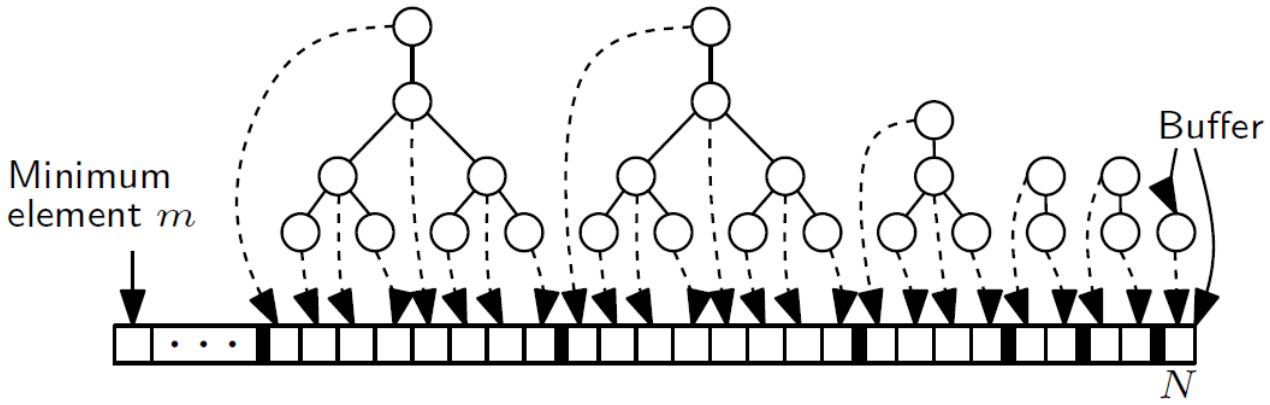
Strict Implicit Priority Queue

O(1) `Insert` and O(log *n*) `ExtractMin`

…work in progress Brodal, Nielsen, Truelsen (2013)

- Forest, trees powers-of-2
- Looser trees
- Pair-encode bits at nodes
- Insertions balance using LSB(n)





(Some Random) Results

♣ amortized bounds	`Insert`	`ExtractMin`	Implicit	Swaps	Identical elements	Cache oblivious
Heaps	log n	log n	Strong	log n	Yes	
Carlsson, Munro, Poblette (1988)	1	log *n*	Weak	log *n*	Yes	
Arge, Bender, Demaine, Holland-Minkley, Munro (2002)	♣log *n*	♣log *n*		♣log *n*	Yes	Yes
Munro, Franceschini (2006)	♣log *n*	♣log *n*	Strong	♣1		
Harvey, Zatloukal (200)	♣1	♣log *n*	Strong	log *n*	Yes	
Brodal, Nielsen, Tr n (2013)	1	log *n*	Strong	log *n*		
Open	1	log *n*	Strong	1	Yes	Yes
	x	x				x
	x	x		x		

Implicit dictionary
O(√n) Munro, Suwanda (1980)
O(log² *n*) Munro (1986)
O(log² *n*/loglog *n*) Franceschini, Grossi, Munro, Pagli(2004)

Thank You !