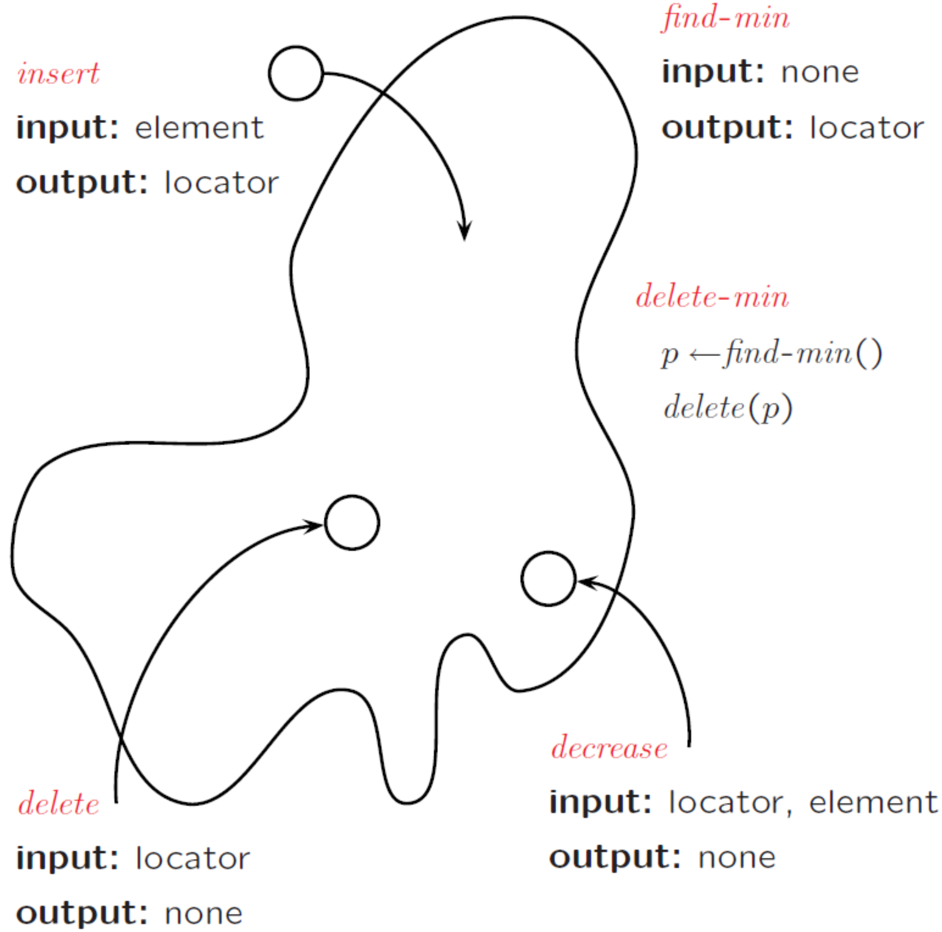
„Policy-Based Benchmarking of Weak Heaps and Their Relatives“

Asger Bruun*, **Stefan Edelkamp**, Jyrki Katajainen*, Jens Rasmussen*

*University of Copenhagen

Priority-Queue Operations



Market Analysis

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	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$

Looking at Constants…

Framework	Structure	*delete*	*insert*	*decrease*
single heap	weak heap	$\lceil \lg n \rceil$	$\lfloor \lg n \rfloor + 1$	$\lceil \lg n \rceil$
multiple heap	weak queue	$2 \lg n + O(1)$	2	$\lfloor \lg n \rfloor$
relaxed heap	run-relaxed weak queue	$3 \lg n + O(1)$	2	4

Further Engineering ➔ Policy-Based Benchmarking…

➔ Methodology

(Policy-Based Benchmarking)

Results Highlights

(LEDA vs. CPH-STL)

Lessons Learnt

Generic Component Frameworks in the CPH STL

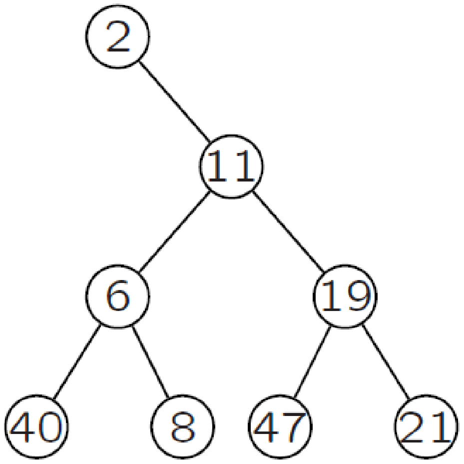
C++ template design: useful to carry out unbiased experiments & (micro-)benchmarking

➔ interfaces are decoupled from their implementations

Clear division of labour:

- Containers de/allocate nodes
- Realizators extract nodes and work on them

(Unidirectional) Iterators traverse through the elements and are used as handles to elements



(Perfect) Weak-Heaps

A (perfect) Weak Heap is a binary tree where:

- the root has no left subtree
- the right subtree of the root is a balanced (complete) binary tree
- each element is smaller than the element on its left „spine“

Observation: 1-to-1 mapping between nodes in heap-ordered binomial trees and perfect weak heaps

Single-Heap Framework

Resizable Array: For iterators validity, store elements indirectly & maintain pointers between array and elements

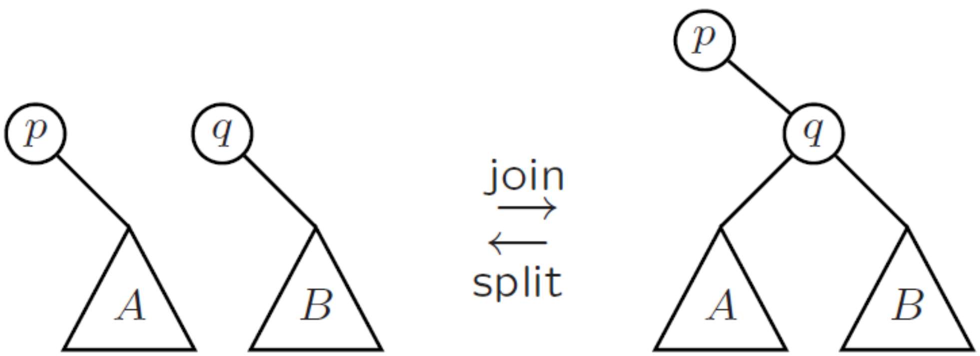
(does not destroy worst-case complexity!)

Heap Structure: different heapifier policies allow switching between weak heaps and different implementations of binary heaps

(e.g., alternative bottom-up sift-down strategy)

Joining/Merging and Splitting

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

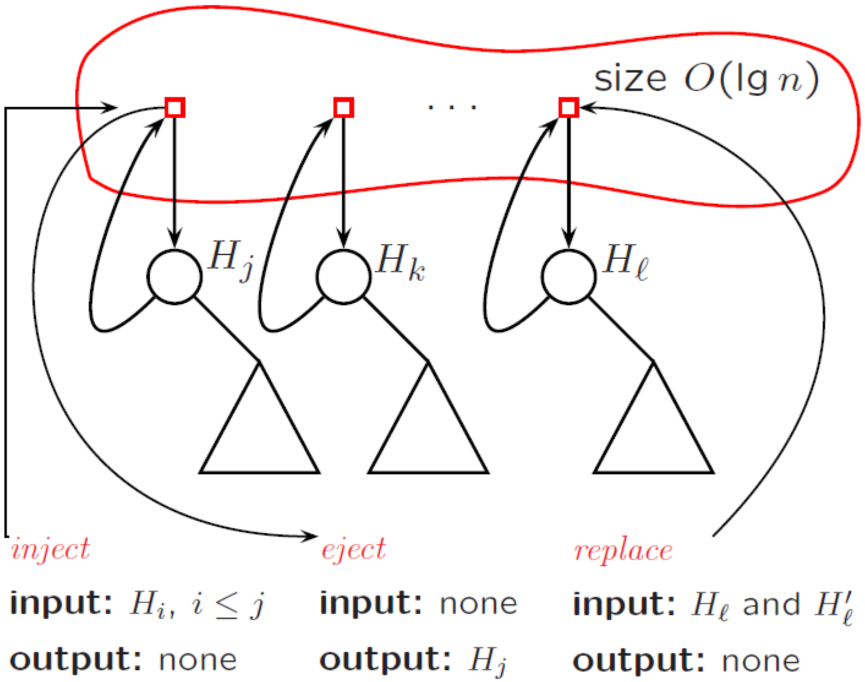


Note that for a binary heap a join may take logarithmic time.

Heap Store

A heap store is a sequence of perfect weak heaps

Joins are delayed using redundant number representation



Insert (node p)

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

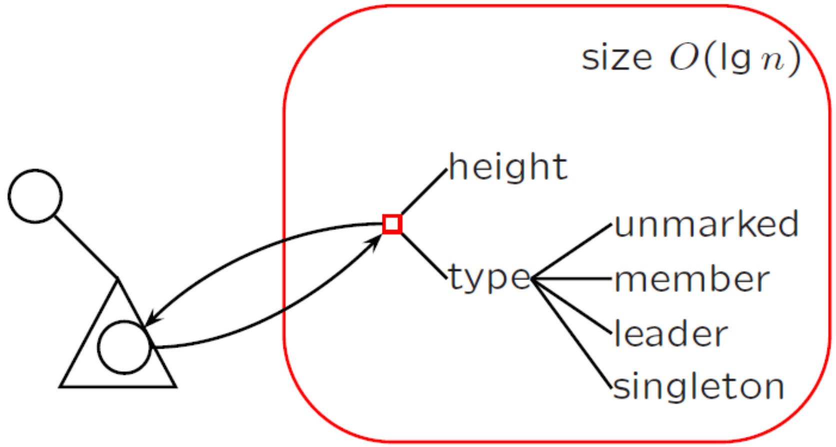
Worst-case time: $\Theta(1)$ with at most 2 element comparisons

Multiple-Heap Framework

Node: 2 pointers per node are sufficient to cover the parent-child relationships, but this space optimization costs execution time

Heap Store: list of proxies was slower than maintaining the roots in a linked list by reusing the pointers at the nodes, where the heights of the heaps are maintained in a bit vector

Node Store



mark

input: a node

output: none

unmark

input: a node

output: none

reduce

input: none

output: none

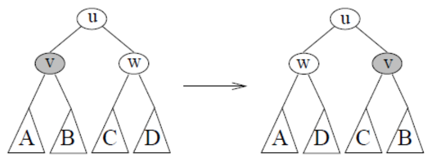
effect: Unmark at least one arbitrary marked node.

maintains heap-order violating marked nodes efficiently

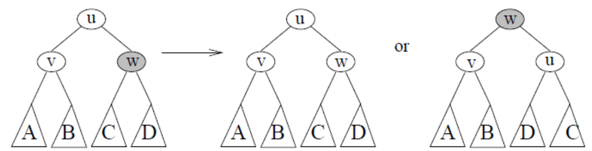
➔ worst case for mark, unmark, and reduce is O(1)

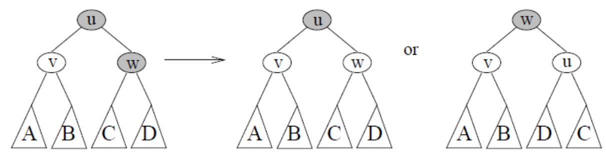
Transformations

- Cleaning:

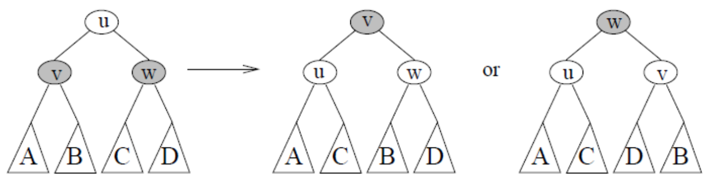


- Parent:

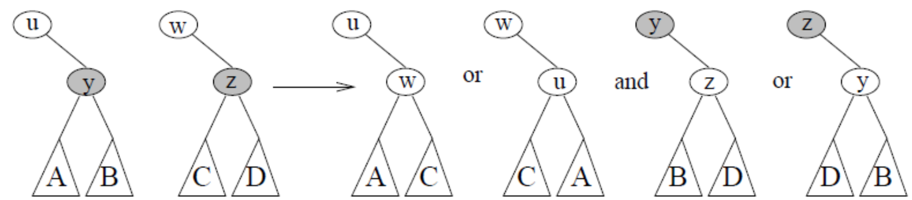




- Sibling:



Pair:



➔ Run (reduces 1 marking, max. 3 comps)

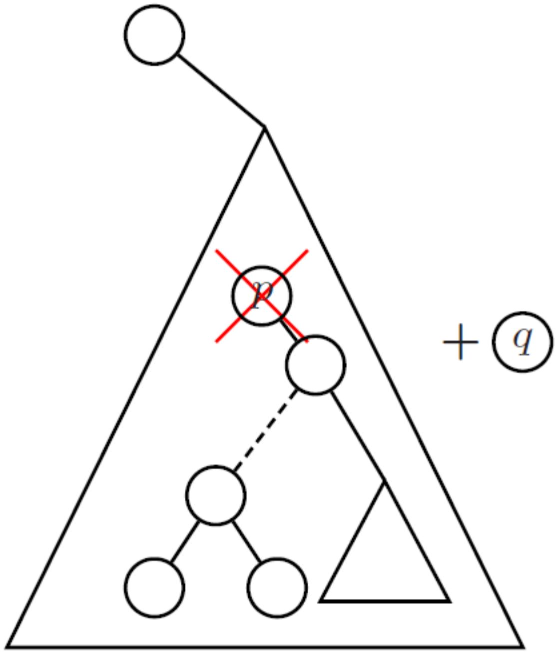
➔ Singleton (reduces 1 marking, max. 3 comps)

Decrease (at node p)

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 (node p)



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

Relaxed-Heap Framework

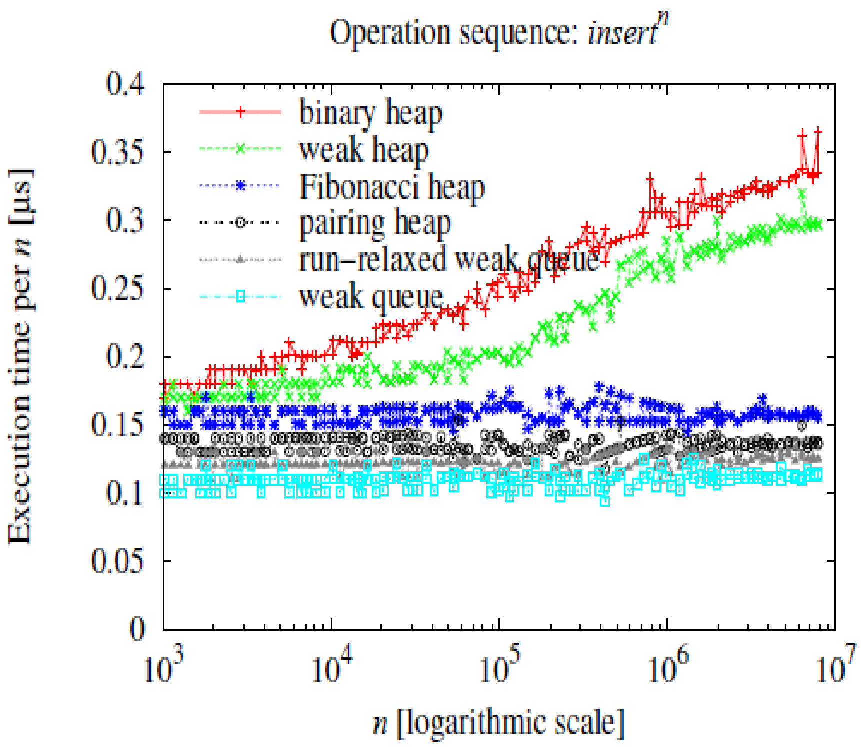
Node Store: doubly-linked lists of leaders, and singletons at each height too slow ➔

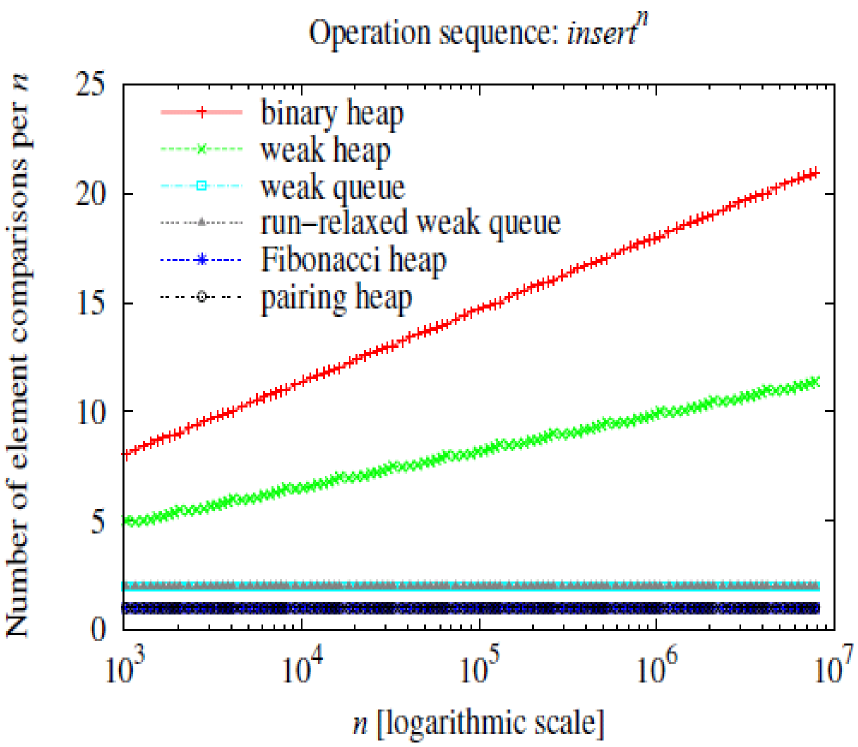
Engineering: array of bit vectors, each occupying a single word, indicating which of the marked nodes are singletons (access via most significant bit)

Rank Relaxation: apply transformations eagerly

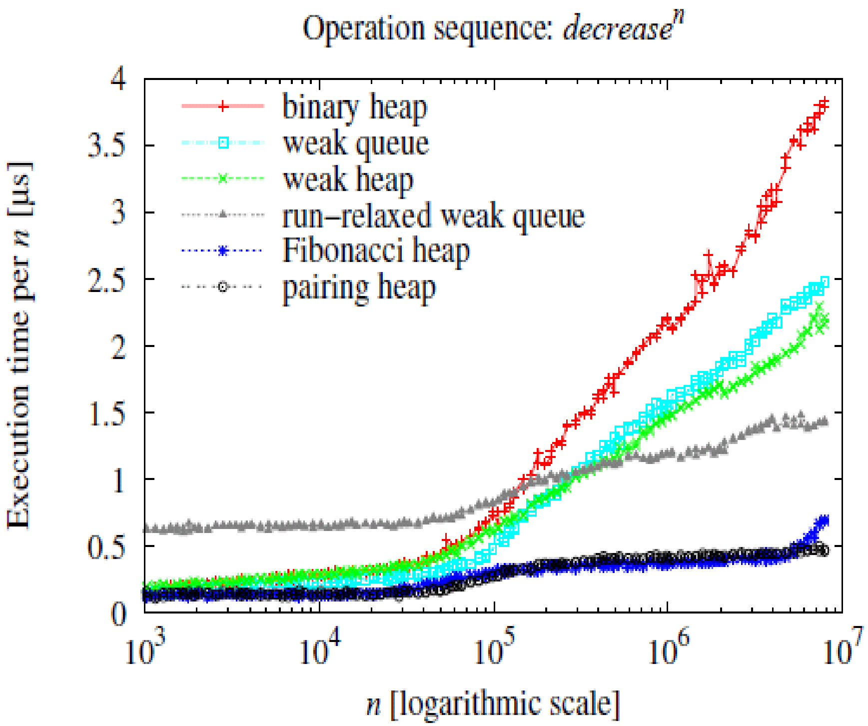
➔ 2 bitvectors

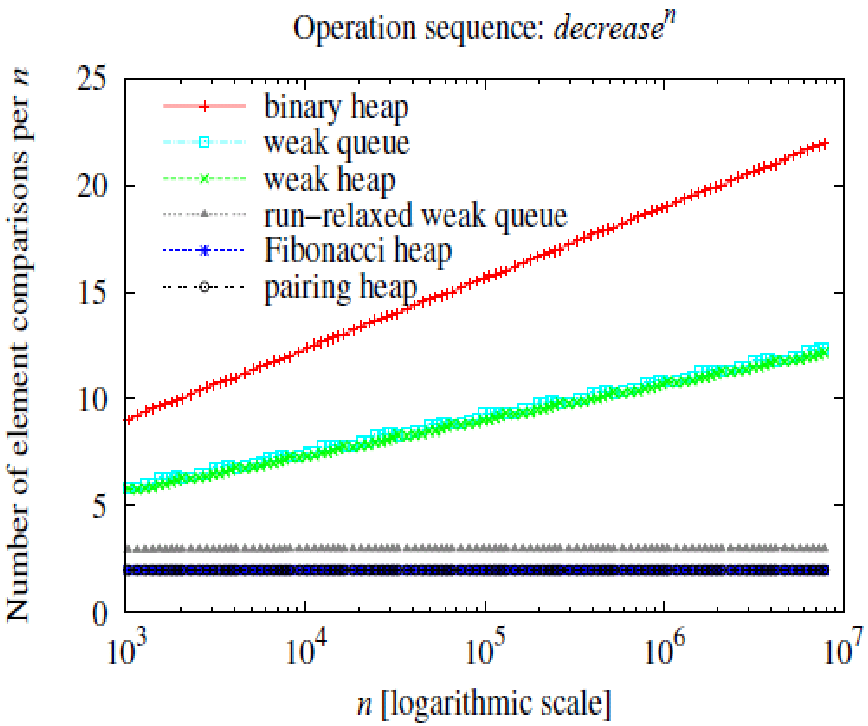
Insert (in sorted order)



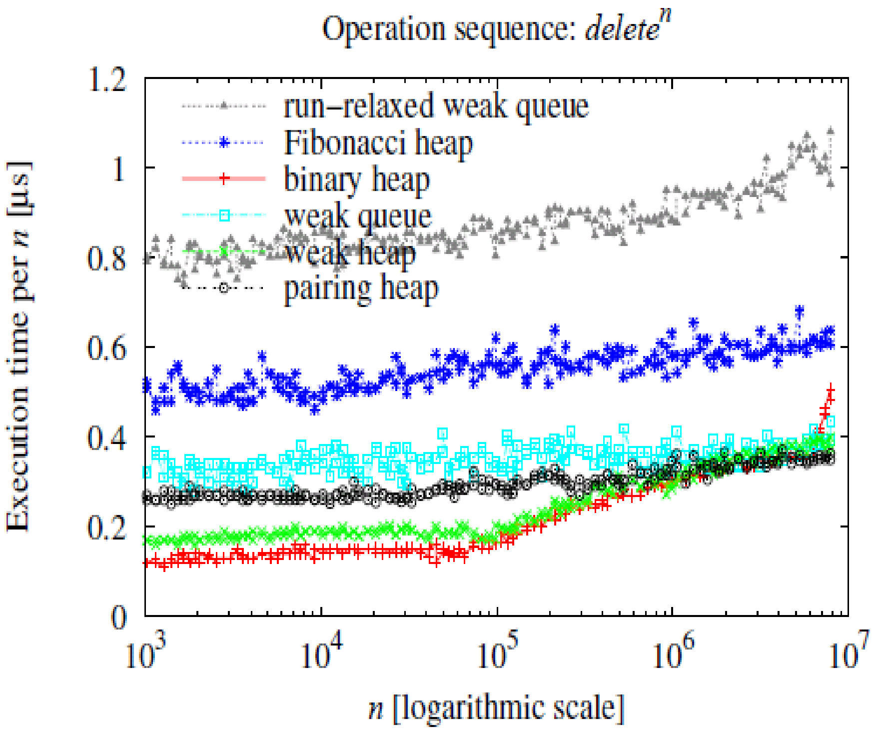


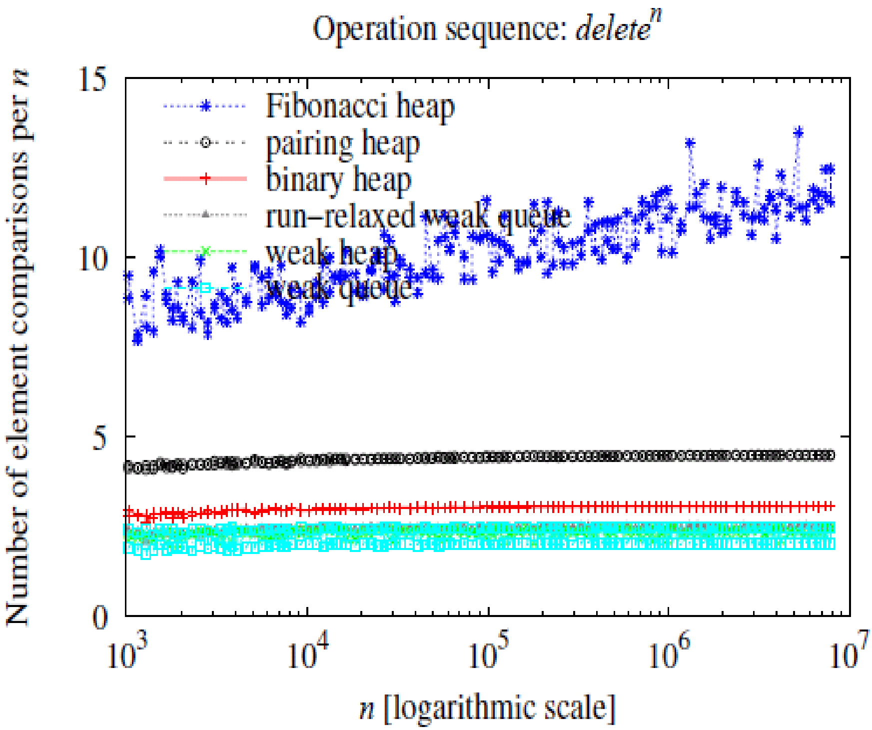
Results: Decrease (to top-1)



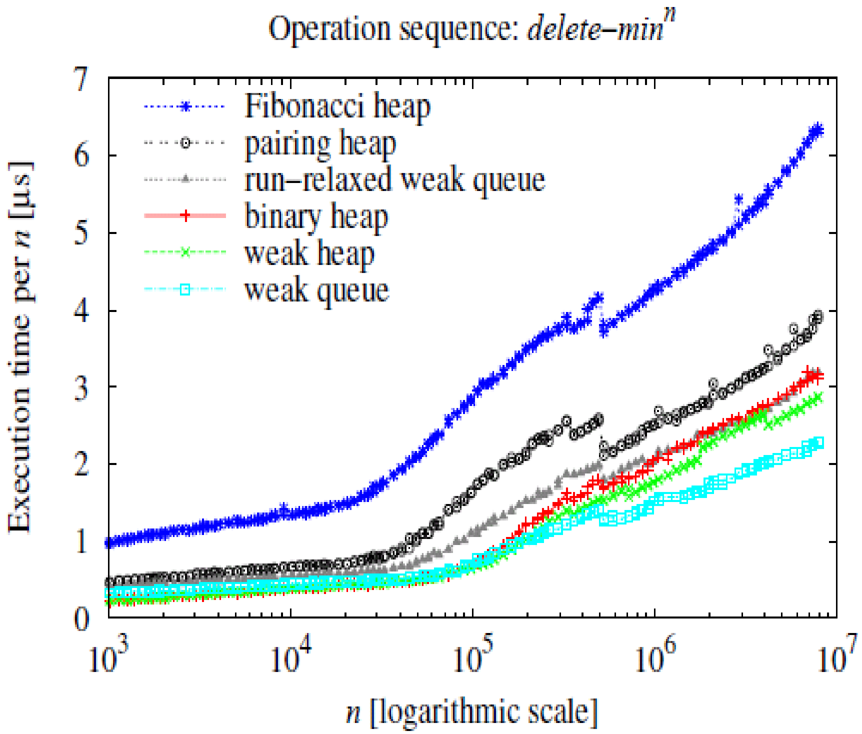


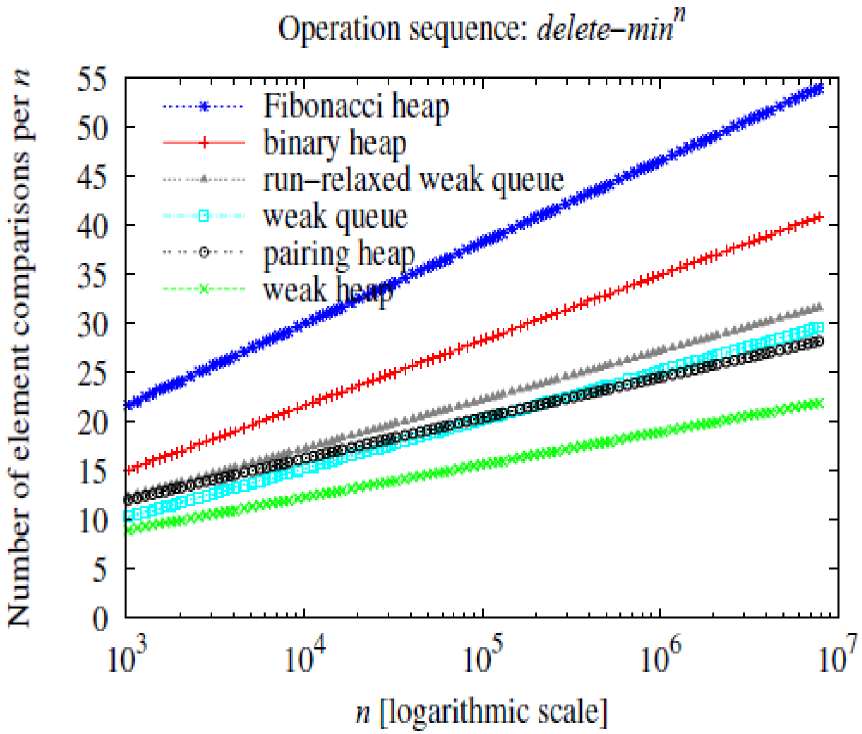
Results: Delete (random position)





Results: Delete-Min (in random heap)





LEDA vs. CPH STL (Dijkstra‘s SSSP)

- CPH STL bin heap 353.334.592 cmps 34.45s
- CPH STL weak heap **194.758.826** cmps 34.45s
- CPH STL weak queue 272.386.118 cmps **29.70s**
- CPH STL run relaxed 324.826.547 cmps 35.98s
- CPH STL rank relaxed 321.256.461 cmps 33.69s

- LEDA pairing heap 276.780.966 cmps 36.72s
- LEDA Fibonacci heap 566.343.539 cmps 47.03s

(Hot: CPH STL Fibonacci heaps 258.767.545 cmps 30.87s)

Lessons Learnt (1)

1) Read the masters: the original implementation of a binomial queue of Vuillemin, in essence a weak queue, turned out to be one of the best performers.
2) PQs that guarantee good performance in the worst-case setting have difficulties in competing against solutions that guarantee good performance in the amortized setting.
3) Memory management is expensive: in our early code many unnecessary memory allocations were performed.

Lessons Learnt (2)

4) For most practical values of n, the difference between lg n and O(1) is small; e.g. for heaps, the loop sifting up an element is extremely tight.
5) For random data, the typical running time of insert, decrease, and delete is O(1) for binary heaps, weak heaps, and weak queues.
6) Generic component frameworks help algorithm engineers to carry out unbiased experiments

Thanks