

# Weak Heaps and Friends: Recent Developments

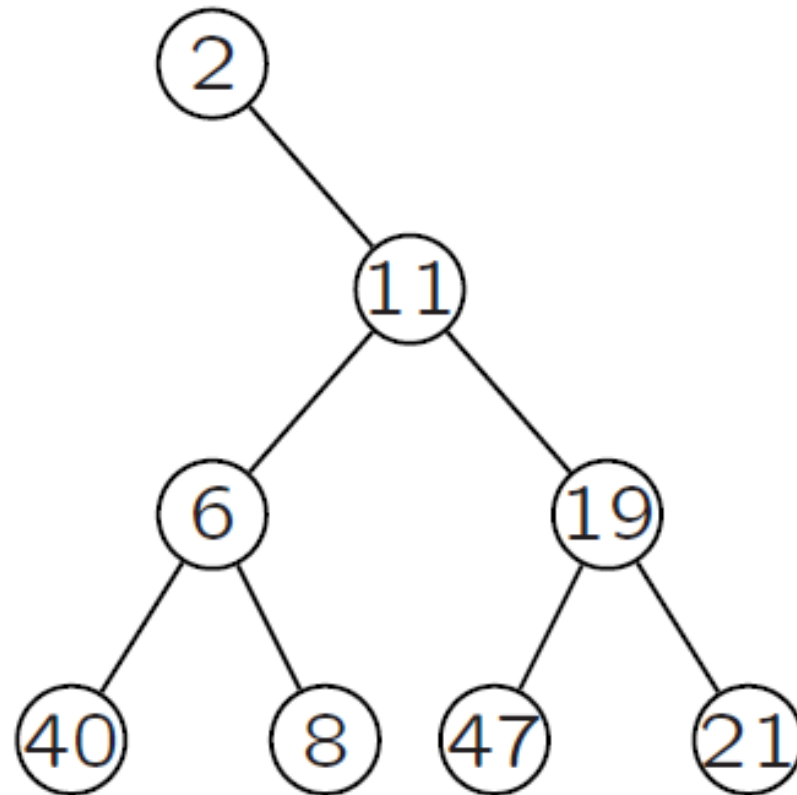
Stefan Edelkamp<sup>1</sup>, Amr Elmasry<sup>2</sup>, Jyrki Katajainen<sup>3,4</sup>,  
Armin Weiß<sup>5</sup>

- 1) University of Bremen
- 2) Alexandria University
- 3) University of Copenhagen
- 4) Jyrki Katajainen and Company
- 5) University of Stuttgart

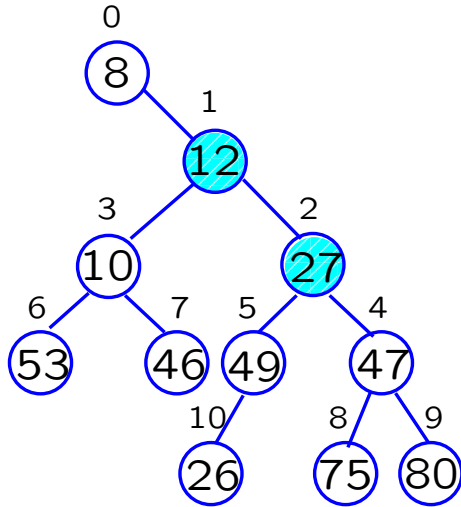
These are Stefan's slides for his invited talk at IWOCA 2013

# (Complete) Weak Heap

---

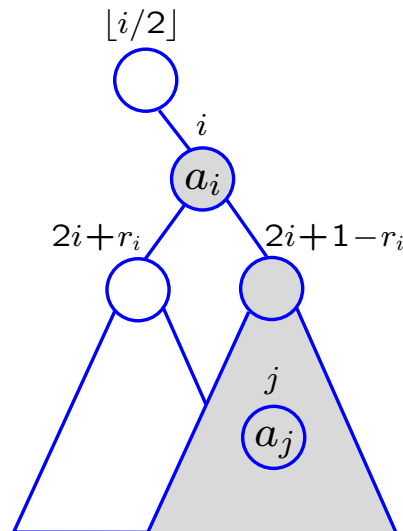


# Array-Based Representation



Array  $a$  of elements  
Array  $r$  of bits (1's in cyan)

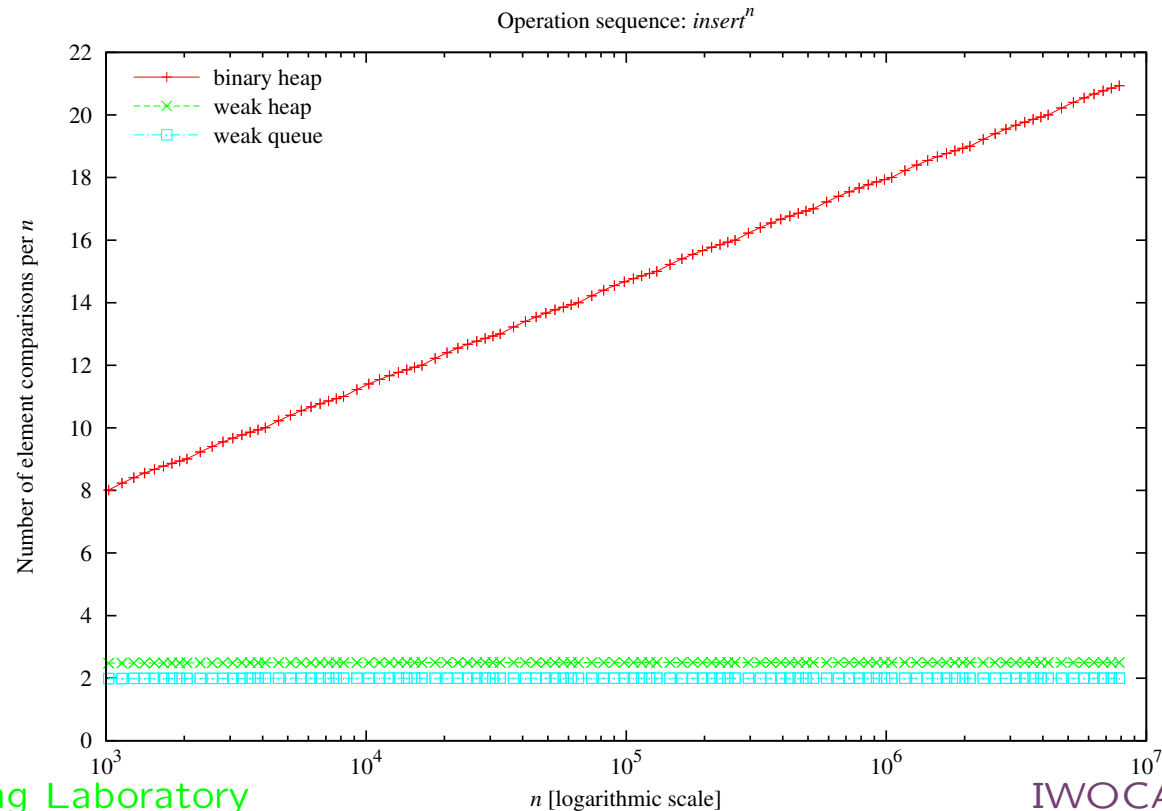
0	1	2	3	4	5	6	7	8	9	10
8	12	27	10	47	49	53	46	75	80	26



# Why Weak Heaps?

<b>Data structure</b>	<i>construct</i>	<i>minimum</i>	<i>insert</i>	<i>extract-min</i>
binary heap [Flo64,Wil64]	$2n$	0	$\lceil \lg n \rceil$	$2\lceil \lg n \rceil$
weak heap [Dut93]	$n - 1$	0	$\lceil \lg n \rceil$	$\lceil \lg n \rceil$

## Repeated insertions [IWOCA-12, JDA-13]



# Structure of the Talk: Research Questions

---

**What is the "best" heap-construction algorithm?**

**What is the "best" sorting algorithm?**

**What is the "best" priority queue?**

## What is the best in-place heap-construction algorithm?

**Best**  $\sim$  In terms of element comparisons and practical running time

**In-place**  $\sim \Theta(1)$  extra words

# Some Options

---

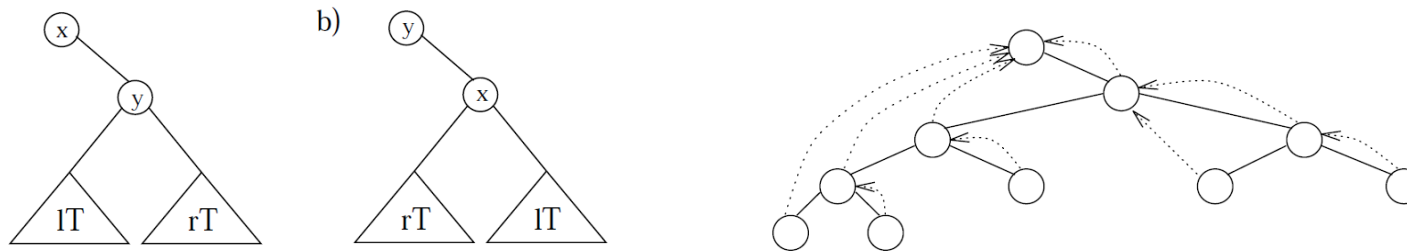
## Element comparisons

<b>Inventor</b>	<b>Abbreviation</b>	<b>Worst</b>	<b>Average</b>	<b>Extra space</b>
Floyd	<b>alg. F</b>	$2n$	$\sim 1.88n$	$\Theta(1)$ words
Gonnet & Munro	<b>alg. GM</b>	$\sim 1.625n$	$\sim 1.625n$	$\Theta(n)$ words
McDiarmid & Reed	<b>alg. MR</b>	$2n$	$\sim 1.52n$	$\Theta(n)$ bits
Li & Reed	<b>lower bound</b>	$\sim 1.37n$	$\sim 1.37n$	$\Omega(1)$ words

Average-case results assume that the input is a random permutation of  $n$  distinct elements

# Building Binary Heaps

---



**Weak heap:** Lower bound  $n - 1$  (element comparisons)

**Weak heap  $\rightarrow$  binary heap:**  $\sim 0.625n$  [IWOCA-12, MFCS-12]

$\rightsquigarrow 1.625n$  heap construction,  $n$  bits (worst case)

**Bottom trees:**  $\rightsquigarrow 1.625n$  in-place heap construction (worst case)

[  $\rightsquigarrow 1.52n$  in-place heap construction (average case) ]



# Weak Heap -> Binary Heap

**GM:** Build a binary heap in two phases: 1) Construct a heap-ordered binomial tree 2) Convert this tree into a binary heap

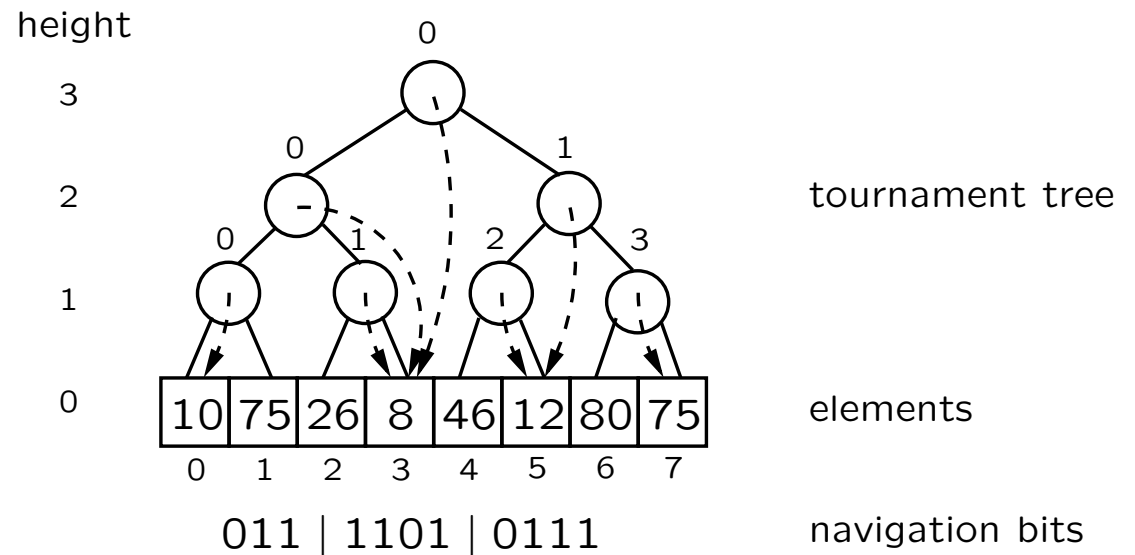
Alternative: a **complete weak heap** → # element comparisons

$$C(8) = 1,$$

$$C(2^k) = 2C(2^{k-1}) + k - 1$$

For  $n = 2^k \geq 8$ , the solution of this relation is  $C(n) = 5/8 \cdot n - \lg n - 1$

Alternative: a **navigation pile** → less element moves



# Bottom-Tree Conversion

---

**Bottom trees:** All complete binary trees of size  $m = 2^{\lfloor \lg \lg n \rfloor + 1} - 1$

1. Convert all bottom trees to **bottom heaps**
2. Ensure heap order at upper levels by using Floyd's sift-down procedure
3. Optimize element moves by handling binary **micro trees** of size 7 differently
  - Elements involved in all bottom-heap constructions  $\leq n$   
→  $1.625n$  element comparisons
  - At most  $\lceil n/2^{h+1} \rceil$  nodes of height  $h$   
→  $o(n)$  element comparisons at the levels above the bottom trees

# Experimental Setup and Summary

---

Random permutations of  $n$  distinct int(eger)s for different (small, medium, large, and very large) problem sizes

Programs tuned to construct binary heaps of size  $2^k - 1$

- **GM** showed acceptable practical performance
- number of element comparisons and element moves was larger for **in-situ GM** than for **in-situ MR**
- **in-situ GM** was faster than **in-situ MR**
- but beaten by **F** and its **BKS** variant

# Element Comparisons

---

$n$	<b>std</b>	<b>F</b>	<b>BKS</b>	<b>in-situ GM</b>	<b>in-situ MR</b>
$2^{10} - 1$	1.64	1.86	1.86	1.74	1.52
$2^{15} - 1$	1.64	1.88	1.88	1.65	1.54
$2^{20} - 1$	1.64	1.88	1.88	1.63	1.53
$2^{25} - 1$	1.65	1.88	1.88	1.63	1.53

**std:** Bottom-up heap construction (`make_heap`, Floyd, Wegener)

**BKS:** Improved version of Floyd's algorithm (Bojesen et al. [JEA-00])

# Execution Times

---

$n$	<b>std</b>	<b>F</b>	<b>BKS</b>	<b>in-situ GM</b>	<b>in-situ MR</b>
$2^{10} - 1$	22.3	14.6	17.1	21.3	26.2
$2^{15} - 1$	22.2	14.6	17.4	23.0	24.4
$2^{20} - 1$	29.3	21.9	17.8	22.9	23.6
$2^{25} - 1$	29.8	21.7	17.5	22.9	23.6

**std:** Bottom-up heap construction (`make_heap`, Floyd, Wegener)

**BKS:** Improved version of Floyd's algorithm (Bojesen et al. [JEA-00])

# What is the best constant-factor-optimal in-situ/adaptive sorting algorithm?

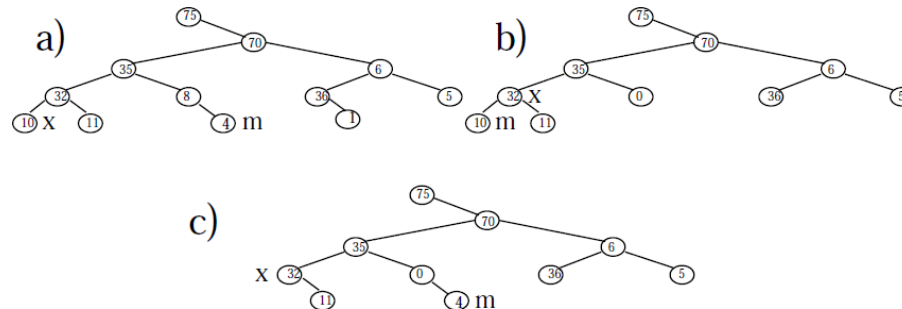
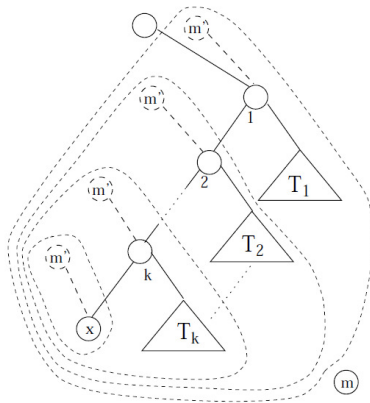
**Best**  $\sim$  In terms of element comparisons and practical running time

**In-situ**  $\sim \Theta(\lg n)$  extra words

**Adaptive**  $\sim$  with respect to inversions

# Sequential Sorting

**Lower bound:**  $\lg n! = n \lg n - n / \ln 2 + O(\lg n)$ , where  $1 / \ln 2 = 1.4426$



- **Worst case:**  $n \lg n + 0.1n$  [Dutton 1993, BIT]
- **Best case/index sorting:**  $n \lg n - 0.9n$  [STACS-00, JEA-02]
- **QuickWeakHeapsort:**  $n \lg n + 0.2n$  on average, in-place [JEA-02]
- **Optimal adaptive sorting:**  $n \lg(\text{Inv}(n)/n) + O(n)$  worst case, two options [IWOCA-11, JDA-12]

# Constant-Factor-Optimal Algorithms

	Space	Time	Worst	Average	Observed
Lower bound	$O(1)$	$\Omega(n \lg n)$	-1.44	-1.44	
BUHeapsort [Weg93]	$O(1)$	$O(n \lg n)$	$\omega(1)$	–	[0.35,0.39]
WeakHeapsort [Dut93]	$O(n/w)$	$O(n \lg n)$	0.09	–	[-0.46,-0.42]
RWeakHeapsort [ES02]	$O(n)$	$O(n \lg n)$	-0.91	-0.91	-0.91
Mergesort [Knu73]	$O(n)$	$O(n \lg n)$	-0.91	-1.26	–
EWeakHeapsort	$O(n)$	$O(n \lg n)$	-0.91	<b>-1.26</b>	–
Insertionsort [Knu73]	$O(1)$	$O(n^2)$	-0.91	<b>-1.38</b>	–
MergeInsertion [Knu73]	$O(n)$	$O(n^2)$	-1.32	<b>-1.3999</b>	<b>[-1.43,-1.41]</b>
InPlaceMergesort [R92]	$O(1)$	$O(n \lg n)$	-1.32	–	–
QuickHeapsort [DW13]	$O(1)$	$O(n \lg n)$	$\omega(1)$	-0.03	$\approx 0.20$
	$O(n/w)$	$O(n \lg n)$	$\omega(1)$	-0.99	$\approx -1.24$
QuickMergesort (IS)	$O(\lg n)$	$O(n \lg n)$	-0.32	<b>-1.38</b>	–
QuickMergesort	$O(1)$	$O(n \lg n)$	-0.32	<b>-1.26</b>	<b>[-1.29,-1.27]</b>
QuickMergesort (MI)	$O(\lg n)$	$O(n \lg n)$	-0.32	<b>-1.3999</b>	<b>[-1.41,-1.40]</b>



# Idea of QuickXsort

---

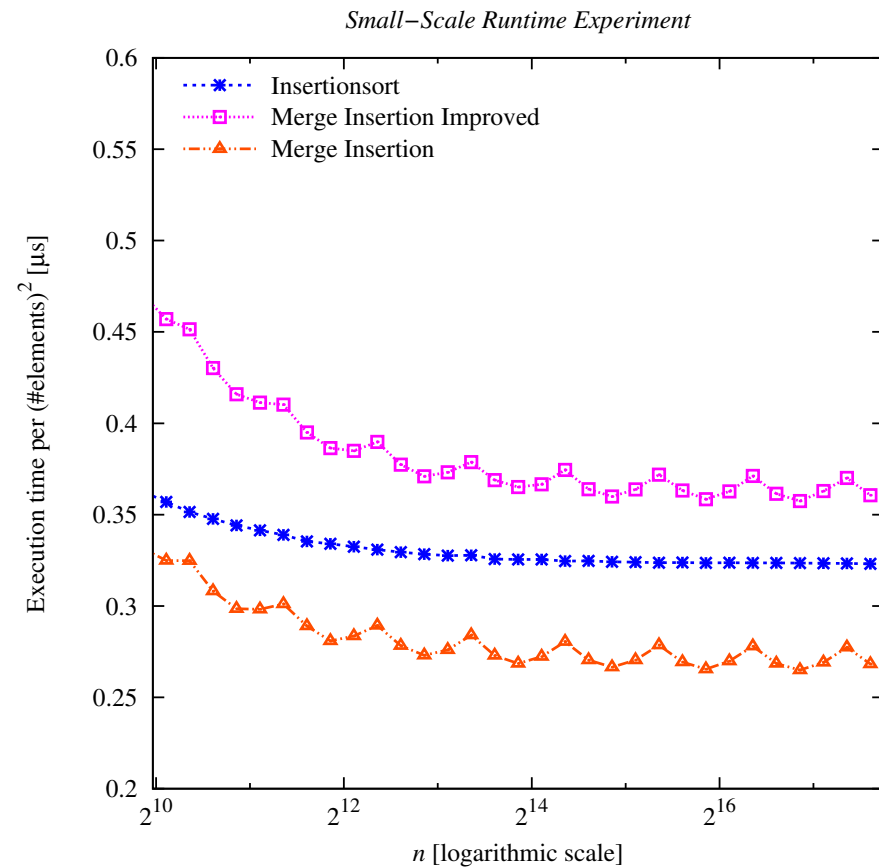
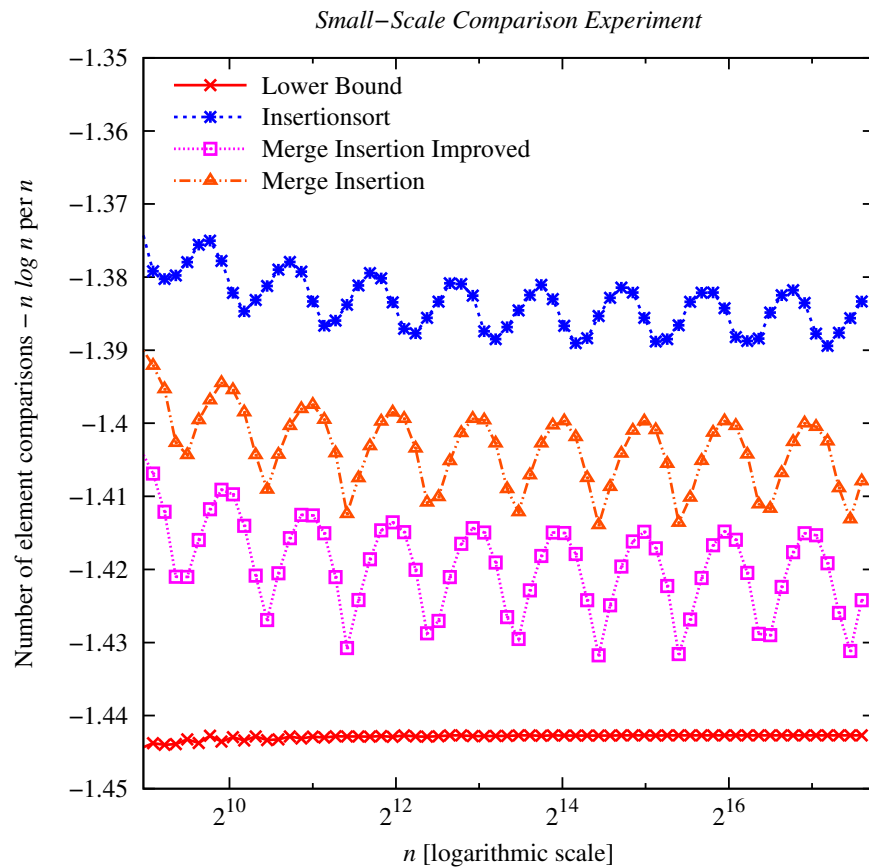
As in **Quicksort** the array is partitioned into the elements greater and less than some pivot element

Then one part of the array is sorted by some algorithm **X** and the other part is sorted recursively

The advantage of this procedure is that, if **X** is a black box, then in **QuickXsort** the part of the array which is not currently being sorted may be used as temporary space, what yields an **in-situ** variant of **X**

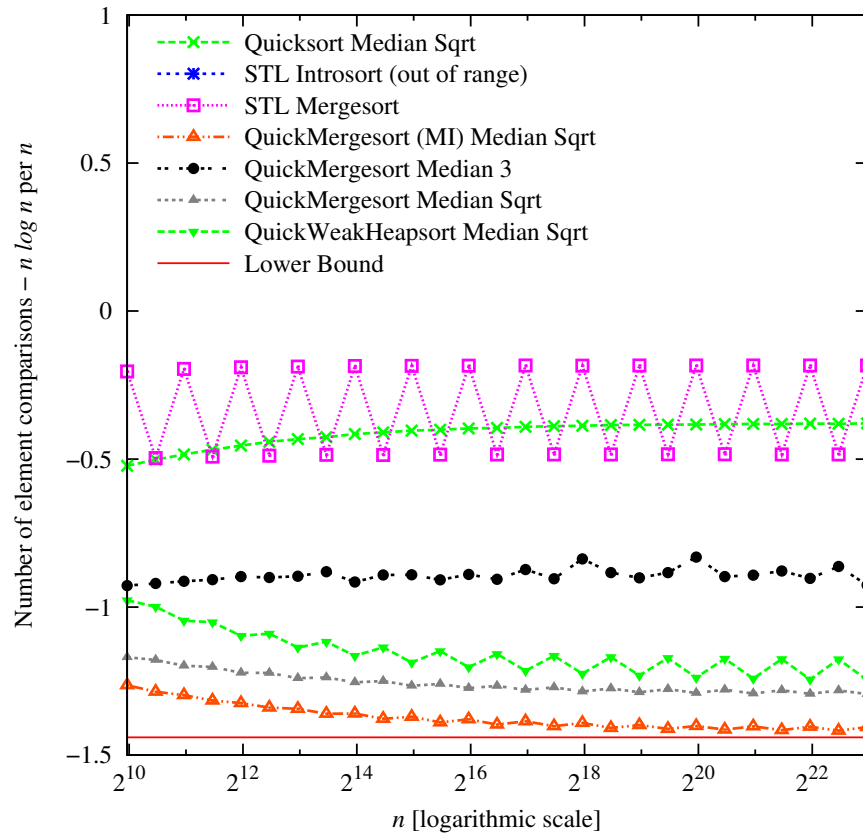
By taking a sample of  $\Theta(\sqrt{n})$  elements when selecting the pivot, **QuickXsort** performs, on an average, the same number of element comparisons as **X** up to an  $o(n)$  lower-order term

# Results for Small Datasets

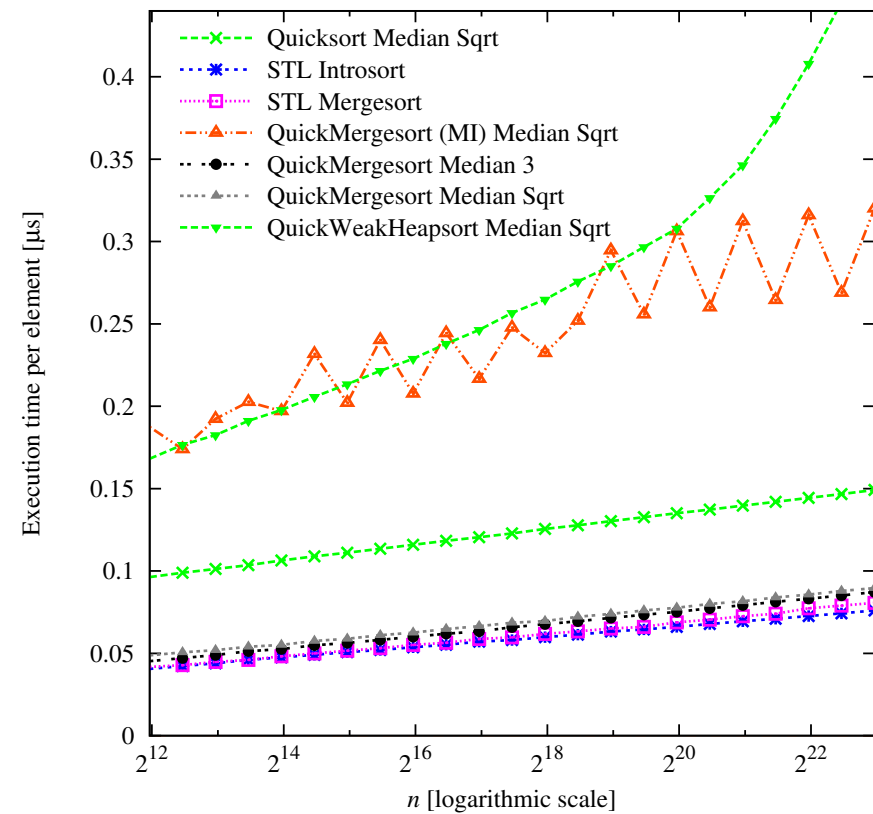


# Results for Large Datasets

Large-Scale Comparison Experiment



Large-Scale Runtime Experiment



# Adaptive Sorting

---

- A sorting algorithm is **adaptive** with respect to a measure of disorder, if it sorts all input sequences, but performs particularly well on those that have a low amount of disorder.
- The running time of such algorithm is measured as a function of the length of the input,  $n$ , and the amount of disorder. Hence, the running time varies between  $O(n)$  time and  $O(n \lg n)$  depending on the amount of disorder.
- The algorithm should be adaptive without knowing the amount of disorder beforehand.

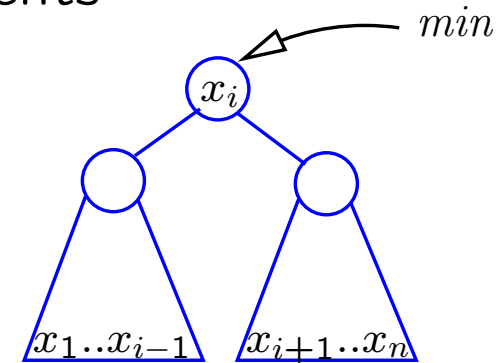
Let  $\langle x_1, x_2, \dots, x_n \rangle$  be a sequence of  $n$  elements. For simplicity, assume that all elements are distinct.

$Inv(n) := \left| \left\{ (i, j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j \right\} \right|$  is one measure of disorder

# Adaptive Heapsort

**input:** sequence  $\langle x_1, x_2, \dots, x_n \rangle$  of  $n$  elements

- 1 Construct an empty Cartesian tree  $\mathcal{C}$
- 2  $hint \leftarrow 0$
- 3 **for**  $i \in \{1, 2, \dots, n\}$
- 4 |  $hint \leftarrow \mathcal{C}.insert(x_i, hint)$
- 5 Construct an empty priority queue  $\mathcal{Q}$
- 6  $\mathcal{Q}.insert(\mathcal{C}.minimum())$
- 7 **for**  $j \in \{1, 2, \dots, n\}$
- 8 |  $x_j \leftarrow \mathcal{Q}.extract-min()$
- 9 | Let  $Y$  be the set of children  $x_j$  has in  $\mathcal{C}$
- 10 | **for** each  $y \in Y$
- 11 | |  $\mathcal{Q}.insert(y)$



**Idea:** Keep  $\mathcal{Q}$  small

[Levcopoulos & Petersson 1993]

# Theoretical Race

---

For priority queue  $\mathcal{Q}$ , element comparisons  $\leq \beta n \lg(\text{Inv}(n)/n) + O(n)$

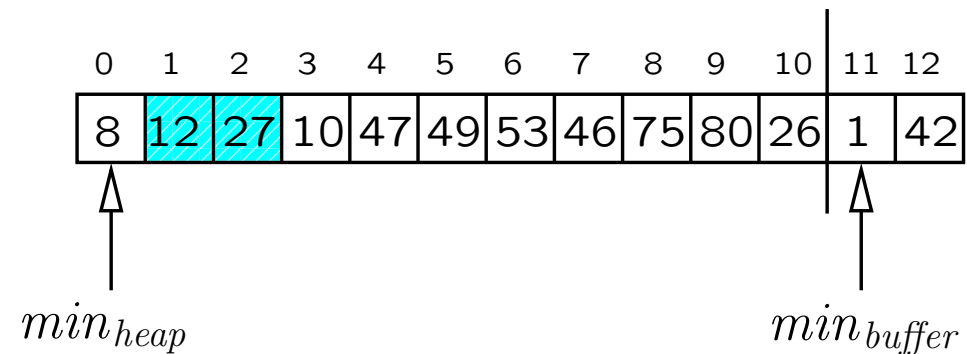
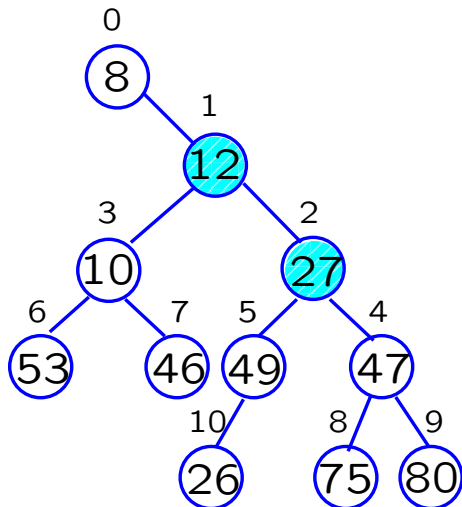
$\mathcal{Q}$	$\beta$	Reference
binary heap	3	
combined <i>extract-min insert</i>	2.5	[Levcopoulos & Petersson 1993]
binomial queue	2	[folklore]
weak heap	2	
combined <i>extract-min insert</i>	1.5	[folklore]
multipartite priority queue	1	[Elmasry, Jensen & Katajainen 2008]

**Goal:** Achieve the constant-factor optimality, i.e.  $\beta = 1$ , and in the meantime ensure practicality!

# One-Buffer Solution

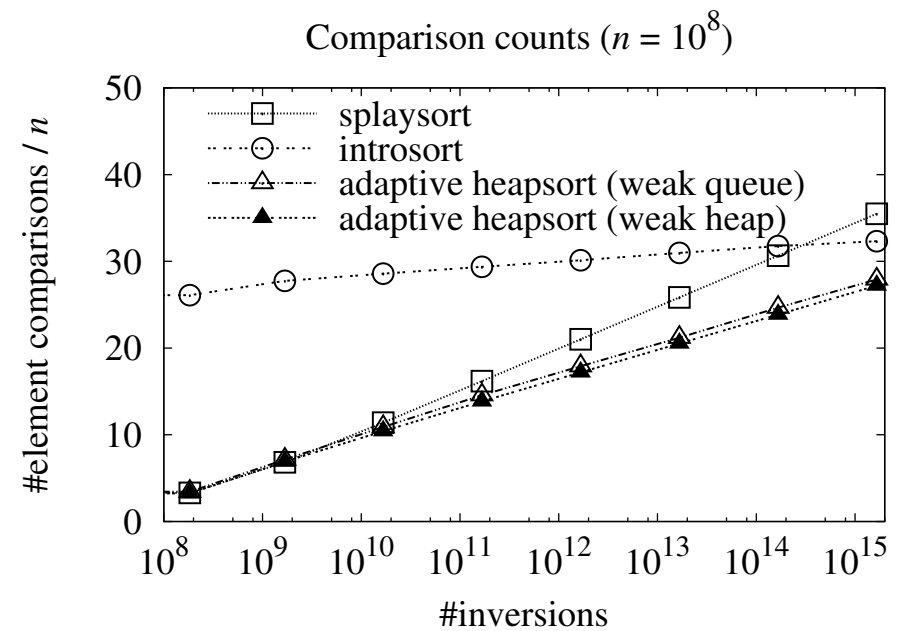
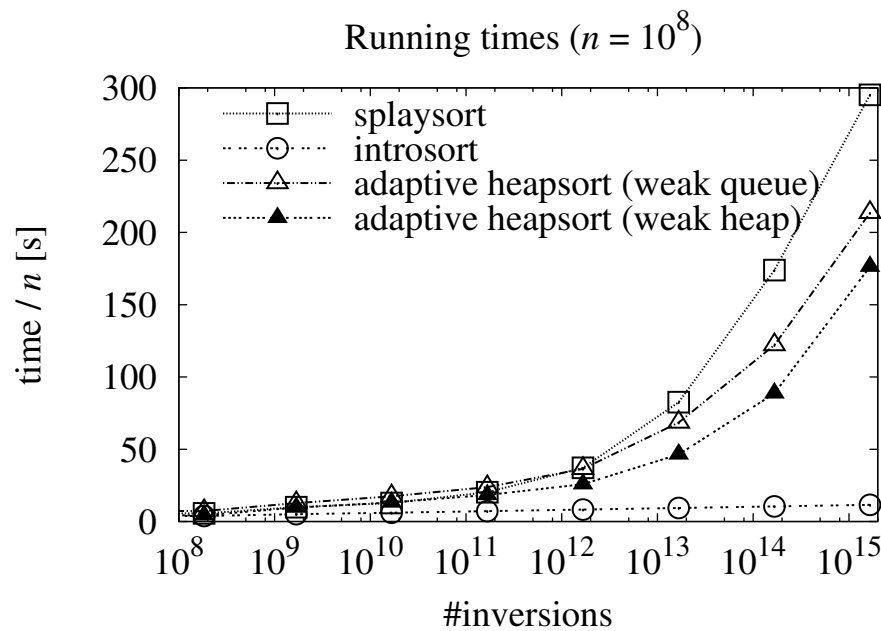
**Task:** *insert* in  $O(1)$  amortized time; *extract-min* in  $O(\lg n)$  worst-case time including at most  $\lg n + O(1)$  element comparisons

**Idea:** Temporarily store inserted elements in a buffer and, once it is full (size  $\lg n$ ), move elements to the main structure as a **bulk**



# Experiments

Running time and element comparisons when  $n = 10^8$





# What is the best elementary priority queue?

**Best** ~ In terms of element comparisons and practical running time

**Elementary** ~ no handles, no support of *decrease* and *delete*

# Elementary Priority Queues

---

## Lower bounds for binary heaps:

$\Omega(\lg \lg n)$  for *insert*,  $\lg n + \log^* n - O(1)$  for *extract-min* (under some assumptions)

## Bulk insertion in weak heaps:

$O(1)$  amortized for *insert*,  $\lg n$  for *extract-min*

## ↪ Engineered weak heaps:

$O(1)$  for *insert*,  $\lg n$  for *extract-min*, space  $n/w + O(1)$

## Bulk insertion in binary heaps:

$O(1)$  amortized for *insert*,  $\lg n$  amortized for *extract-min*

## ↪ Optimal in-place heaps:

$O(1)$  for *insert*,  $\lg n$  for *extract-min*, space  $O(1)$  [submitted]

# Performance of Some Priority Queues

---

<b>Data structure</b>	<b>Space</b>	<i>insert</i>	<i>extract-min</i>
binary heaps [Wil64]	$O(1)$	$\lg n + O(1)$	$2 \lg n + O(1)$
binom. queues [Bro78,Vui78]	$O(n)$	$O(1)$	$2 \lg n + O(1)$
heaps on heaps [GM86]	$O(1)$	$\lg \lg n + O(1)$	$\lg n + \log^* n + O(1)$
queue of pennants [CMP88]	$O(1)$	$O(1)$	$3 \lg n + \log^* n + O(1)$
multipartite PQs [EJK08]	$O(n)$	$O(1)$	$\lg n + O(1)$
engin. weak heaps [EEK13]	$n/w + O(1)$	$O(1)$	$\lg n + O(1)$
optimal in-place heaps	$O(1)$	$O(1)$	$\lg n + O(1)$

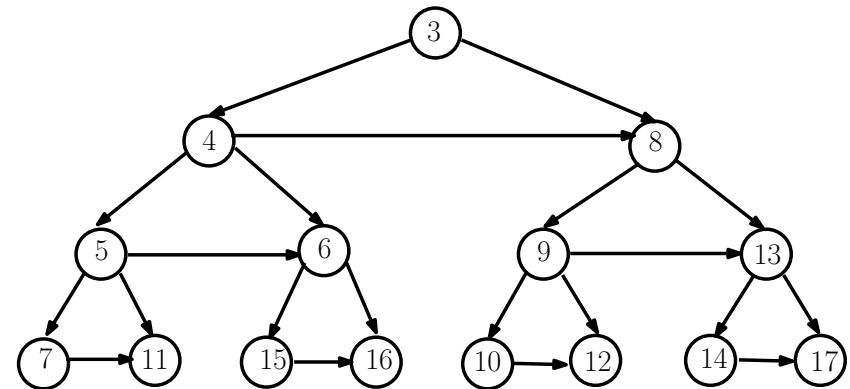
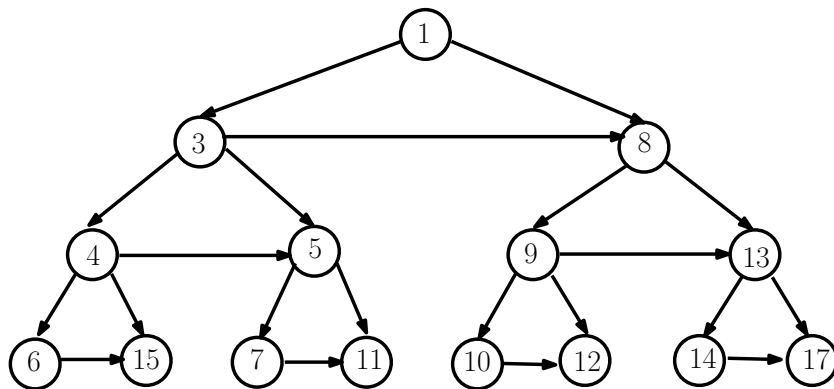
All data structures support *construct* in  $O(n)$  and *minimum* in  $O(1)$  worst-case time

# Strong Heaps

A **strong heap** is a binary heap where nodes dominate their right siblings

## Rotating sift-down

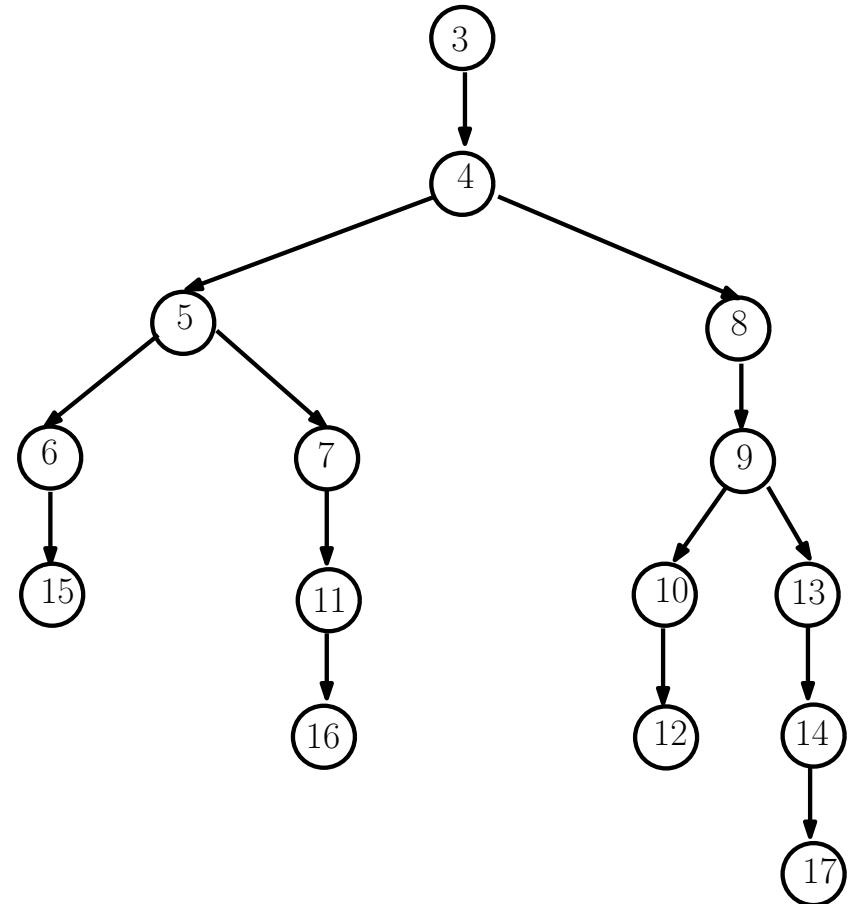
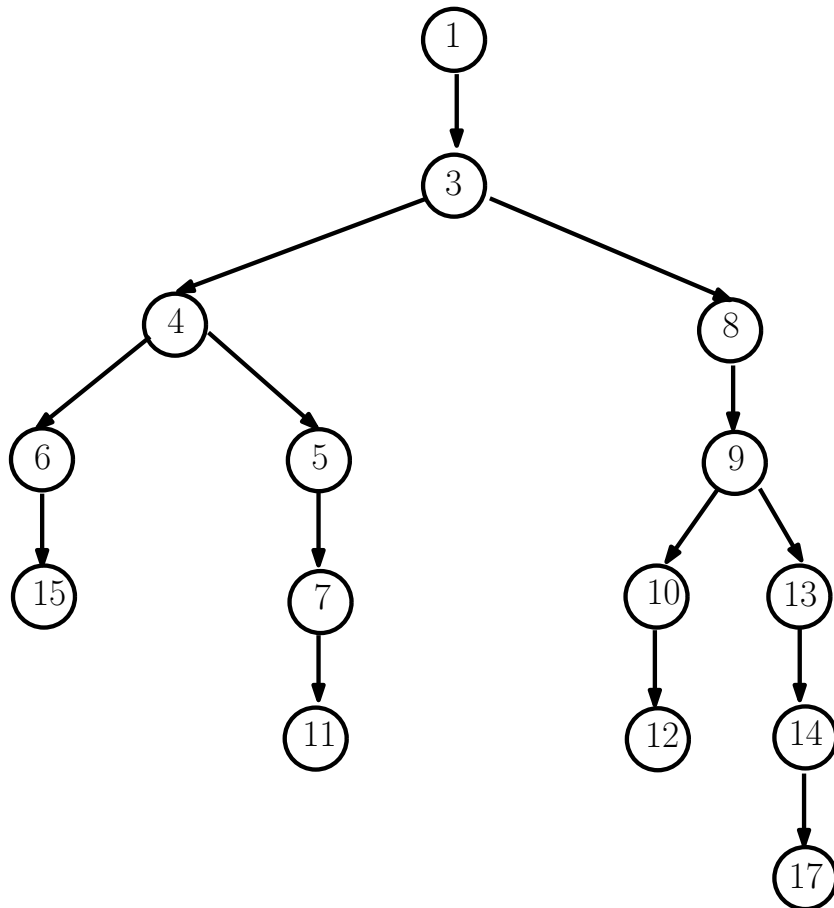
Replace the minimum with 16



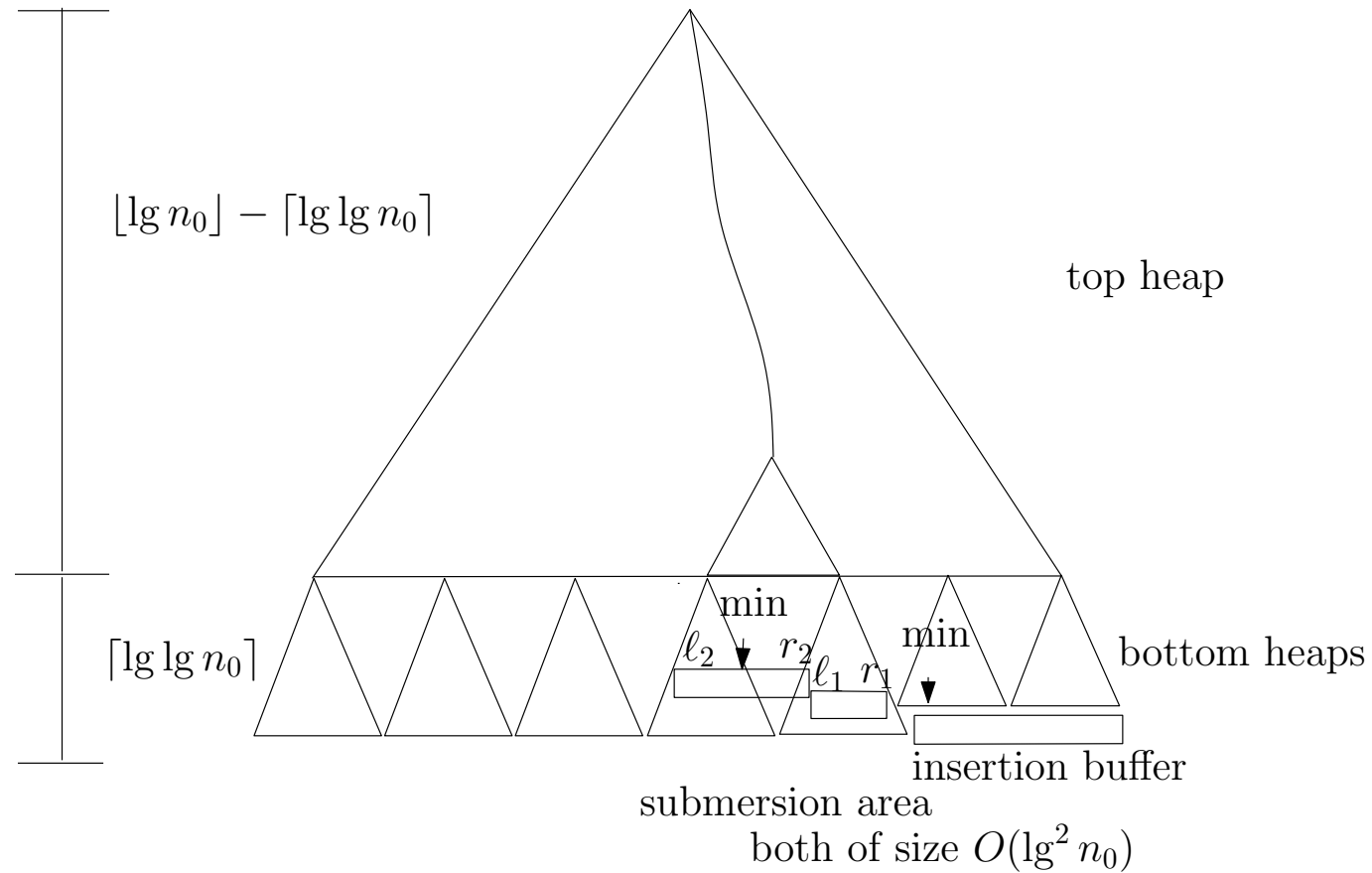
# Strong Heaps

## Strong sift-down

Replace the minimum with 16



# Optimal In-Place Heaps

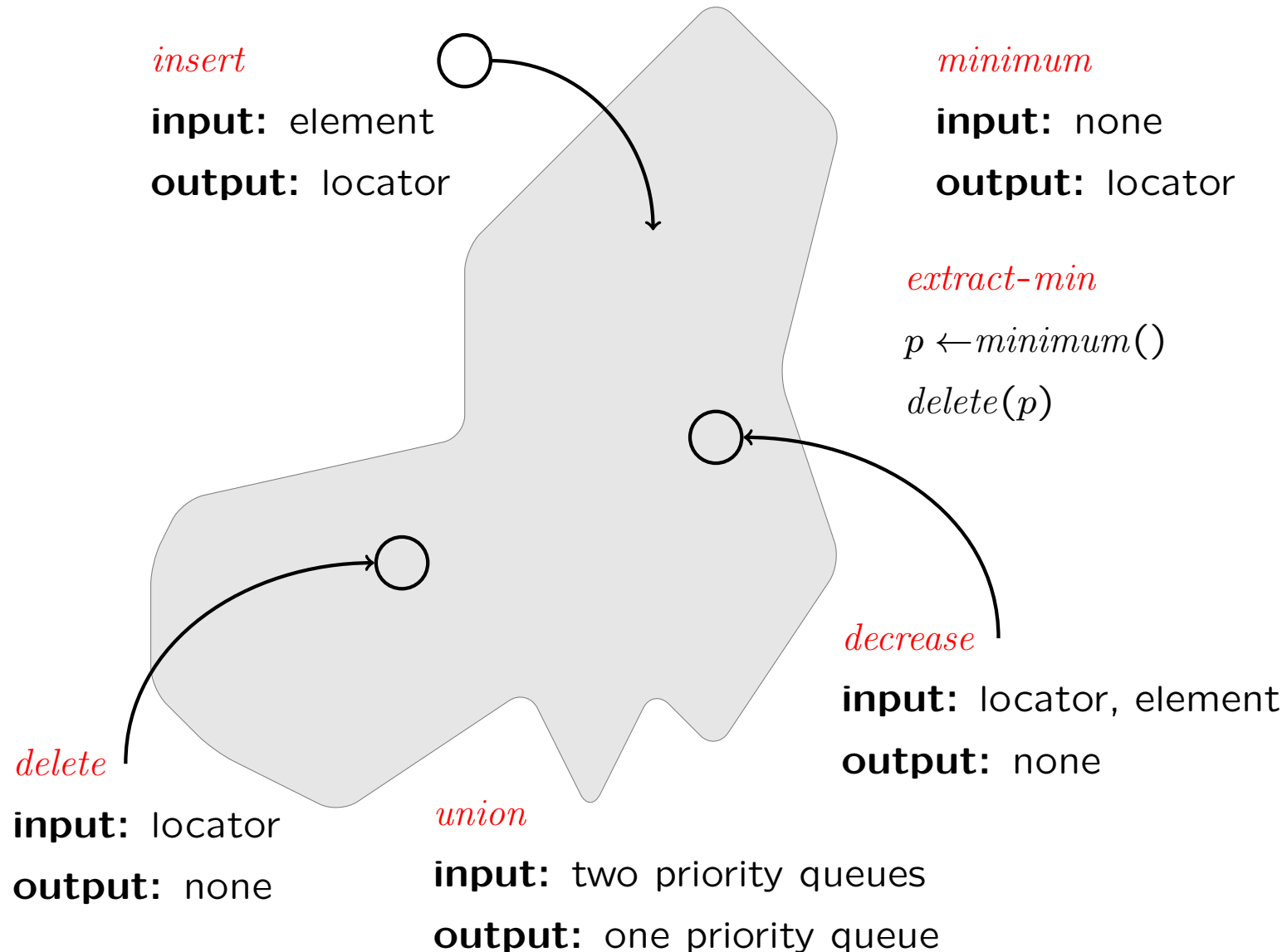


What is the best bound when handling a request sequence consisting of  $n$  *insert*,  $n$  *extract-min*, and  $m$  *decrease* operations?

**Best**  $\sim$  In terms of element comparisons and practical running time

# Addressable Priority Queues

---





# Market Analysis

Efficiency Operation	binary heap worst case	binomial queue worst case	Fibonacci heap amortized	run-relaxed heap worst case
<i>minimum</i>	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
<i>insert</i>	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
<i>decrease</i>	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(1)$
<i>delete</i>	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$
<i>union</i>	$\Theta(\lg m \times \lg n)$	$\Theta(\min\{\lg m, \lg n\})$	$\Theta(1)$	$\Theta(\min\{\lg m, \lg n\})$

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

# Result

---

Rank-relaxed weak heaps are **better** than Fibonacci heaps!

Data structure	Element comparisons
Fibonacci heap	$2m + 2.89n \lg n$
Rank-relaxed weak heap	$2m + 1.5n \lg n$

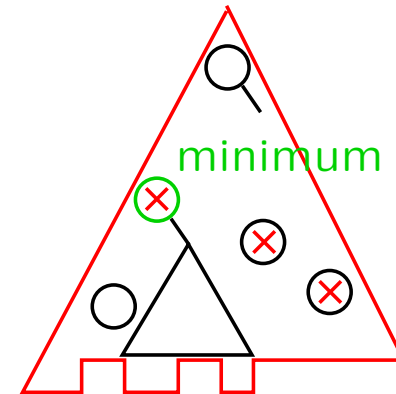
But they are not **simpler**!

Data structure	Lines of code
Binary heap	205
Fibonacci heap	296
Rank-relaxed weak heap	883

# Pointer-Based Representation

---

- Three pointers per node: left child, right child, parent
- One element per node; in the right subtree no element is larger
- Except the root, the nodes that have at most one child are at the last two levels only
- Recall where the minimum is

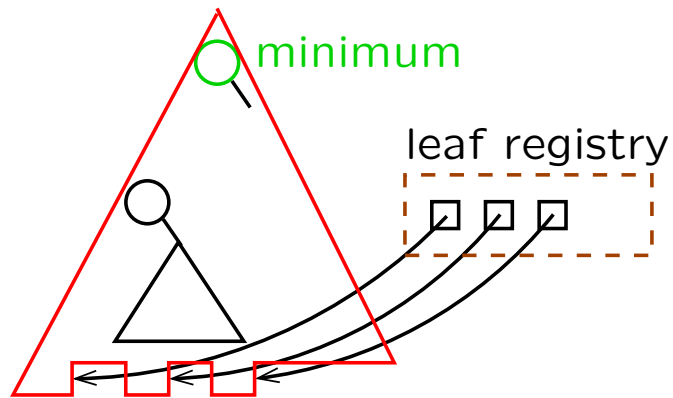


**Run relaxation:** Remove a potential violation if absolutely necessary (lazy)

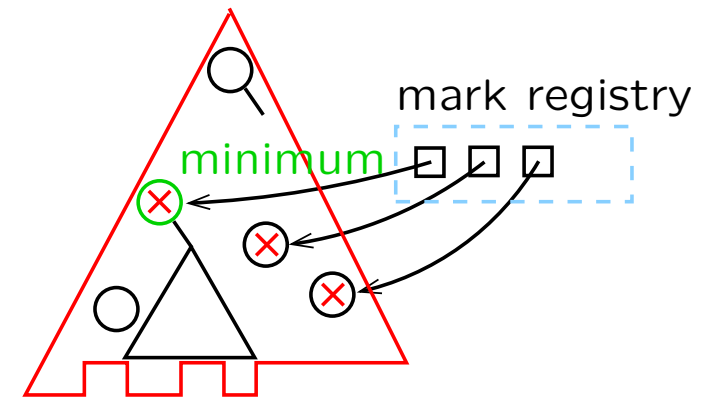
**Rank relaxation:** Remove a potential violation whenever possible (eager)

# Registries

**Leaf registry:** Keep track of the nodes that have one child or no children

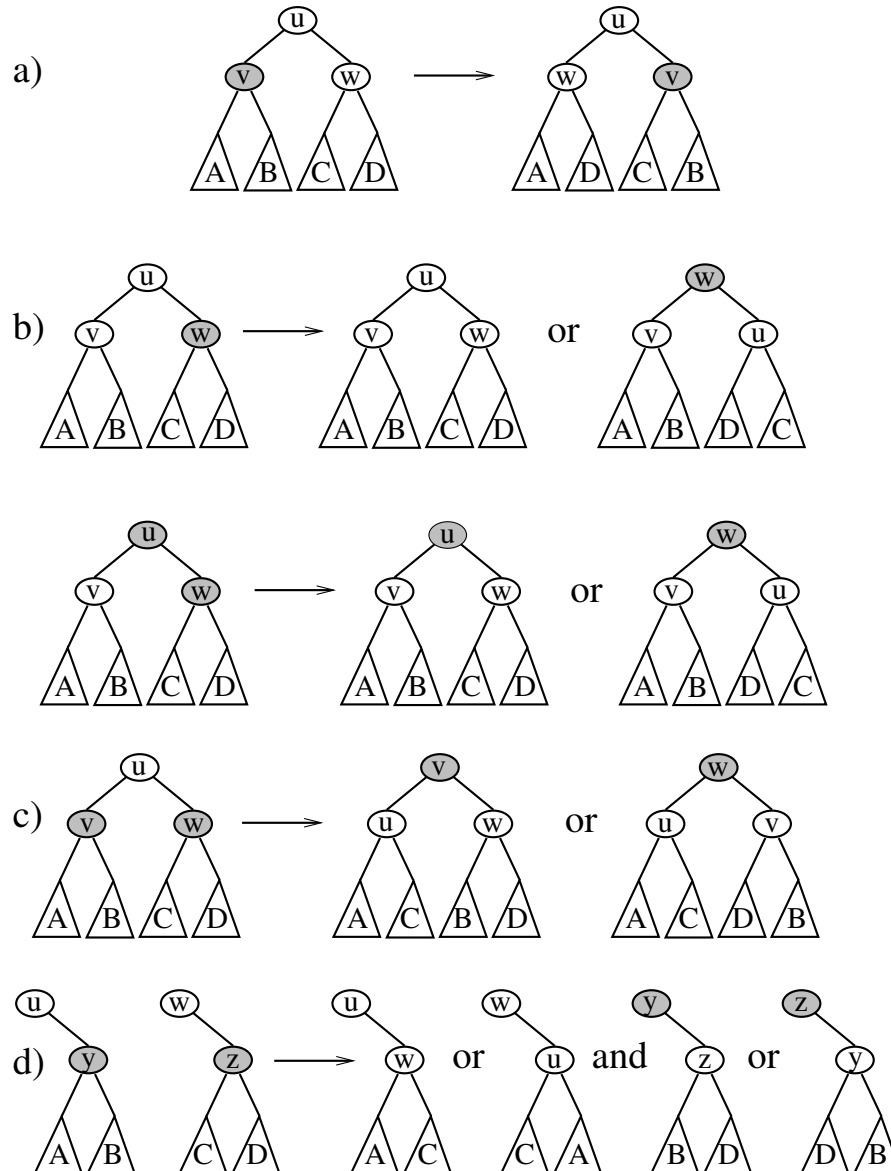


**Mark registry:** Keep track of the potential violations



**Basic operations in both:** Location-based *insert* and *delete*

# Transformations



a) cleaning transformation

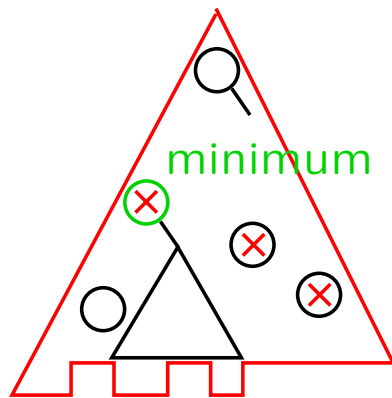
b) parent transformation

c) sibling transformation

d) pair transformation

gray nodes are marked

# Rank-Relaxed Weak Heap



- $\lambda \leq \lfloor \lg n \rfloor - 1$  nodes marked; they may violate the weak-heap ordering

*insert*: Insert a leaf, mark it, apply  $\lambda$ -reducing transformations as long as possible.

*decrease*: Decrease the value in the given node, mark it, apply  $\lambda$ -reducing transformations as long as possible.

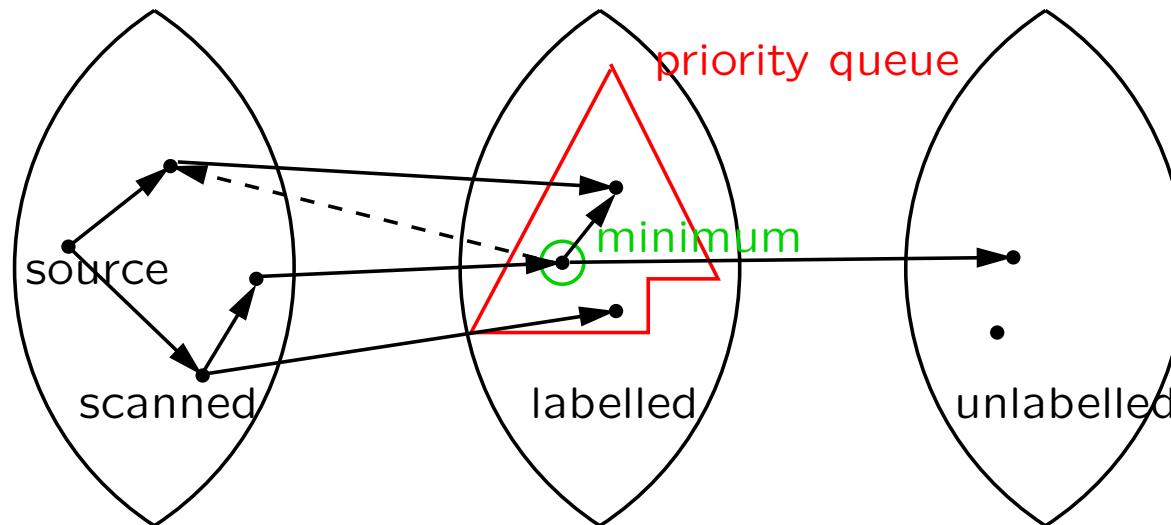
*extract-min*: Find the minimum (at the root or one of the marked nodes), borrow a leaf, fix the structure of the subtree that lost its root, mark the root of the fixed subtree, apply  $\lambda$ -reducing transformations as long as possible.

**Improvement in *extract-min***: If the mark registry is more than half full before the minimum finding, empty it.

# Our Play with Dijkstra's Algorithm

With your search engine, you will find many experimental studies on Dijkstra's algorithm. Be critical when you read the results.

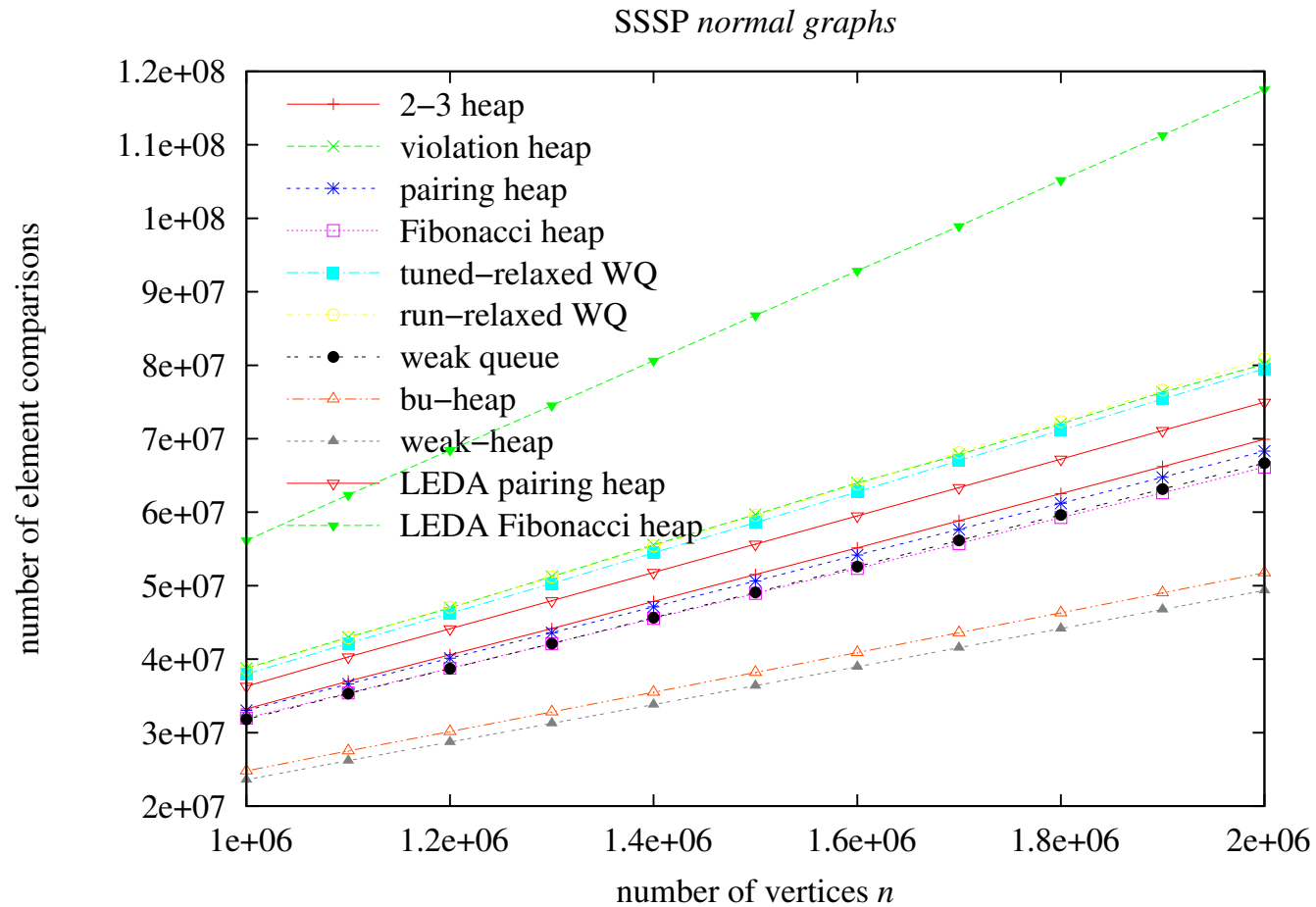
- Which algorithm
- Which graph representation
- Which priority queue
- Which tuning level



*a factor of two speed-up*

# Policy-Based Benchmarking

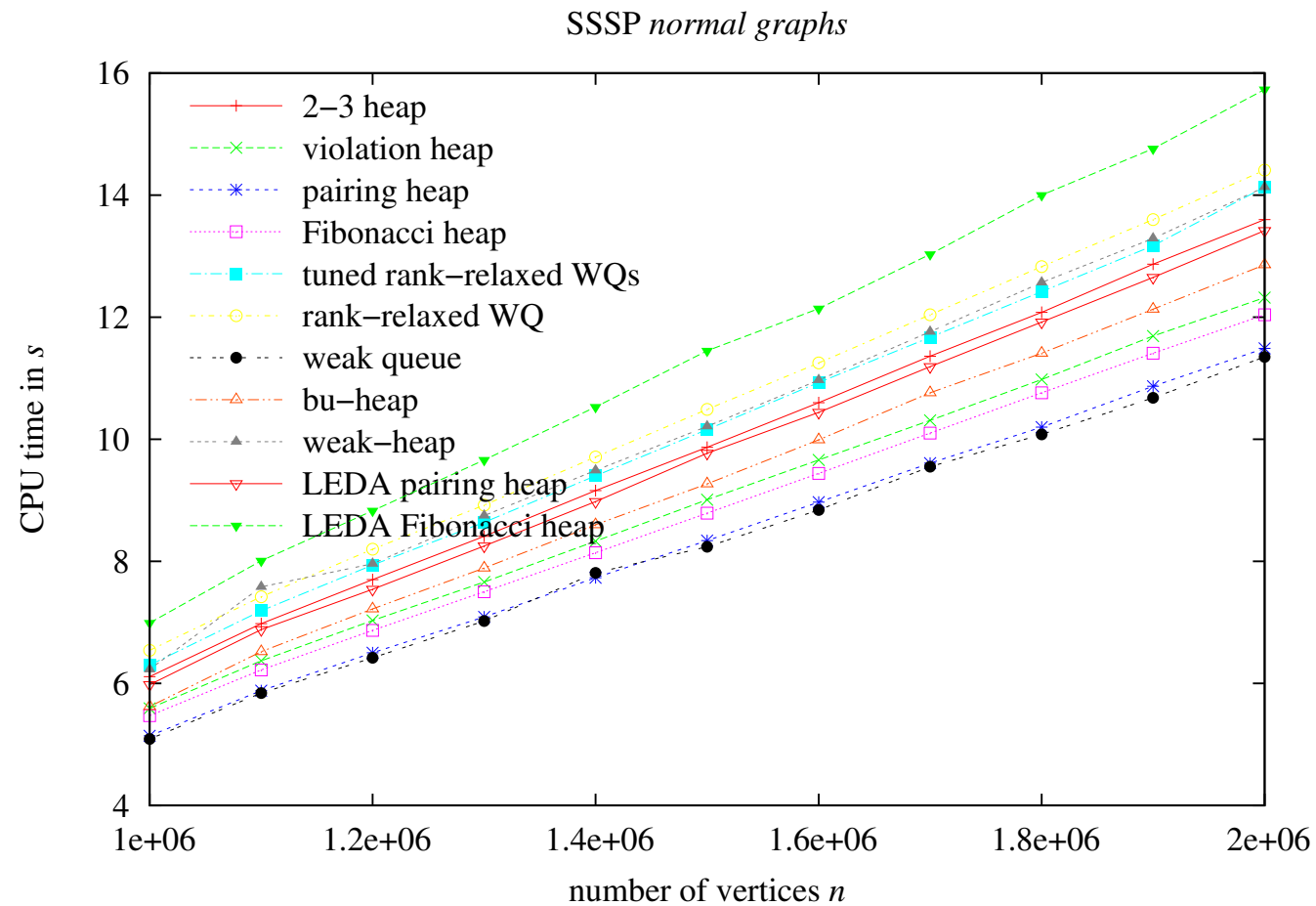
## Element comparisons





# Policy-Based Benchmarking

## Running time



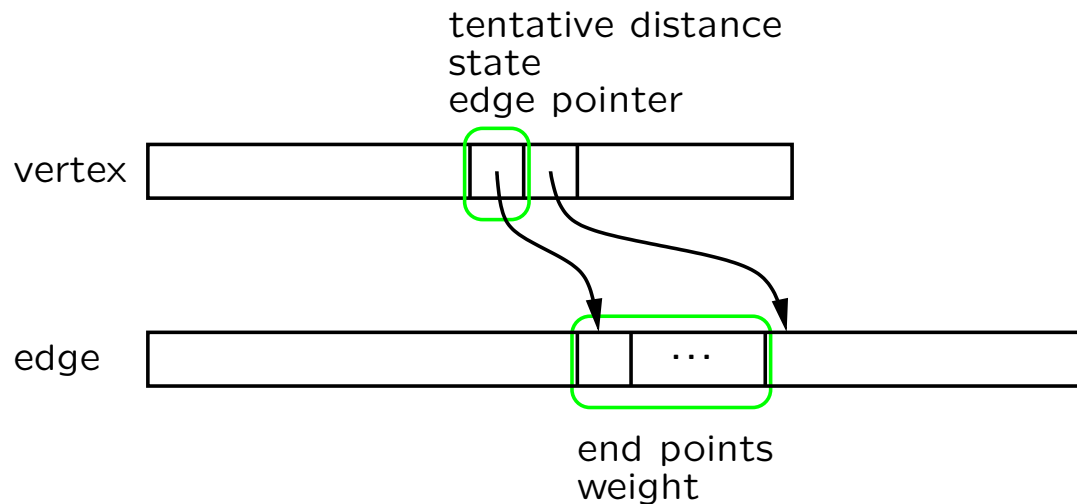
# Graph Representation

## CPH STL

- adjacency arrays
- simple
- static
- $16m + 16n + O(1)$  bytes for a graph with  $m$  edges and  $n$  vertices

## LEDA

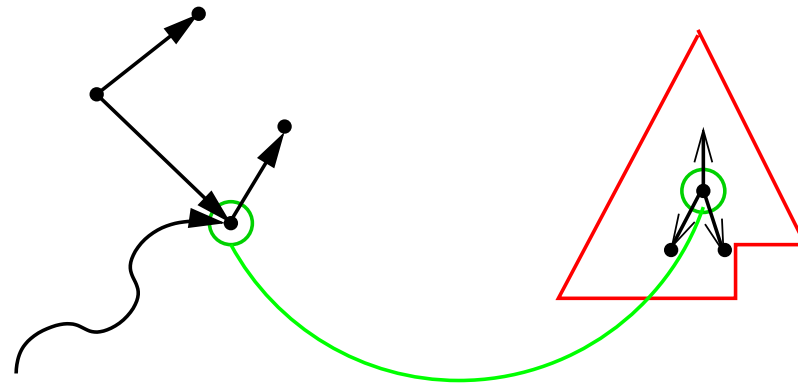
- adjacency lists
- nice interface
- fully dynamic
- parameterized
- $52m + 60n + O(1)$  bytes [LEDA Book, § 6.14]



a factor of two speed-up

# Avoiding Indirection

---



Combine the graph vertex and the priority-queue node [Knuth 1994]

→ improves cache behaviour

*a factor of two speed-up*

# Tuning

## Running time per $n$ [ $\mu$ s]

Structure Operation	CPH STL Fibonacci heap	LEDA 6.2 Fibonacci heap
<i>insert</i>		
$n$ : 10 000	0.10	0.18
$n$ : 100 000	0.09	0.15
$n$ : 1 000 000	0.09	0.15
<i>decrease</i>		
$n$ : 10 000	0.03	0.06
$n$ : 100 000	0.05	0.22
$n$ : 1 000 000	0.06	0.31
<i>extract-min</i>		
$n$ : 10 000	0.7	1.2
$n$ : 100 000	1.4	2.7
$n$ : 1 000 000	2.8	4.5

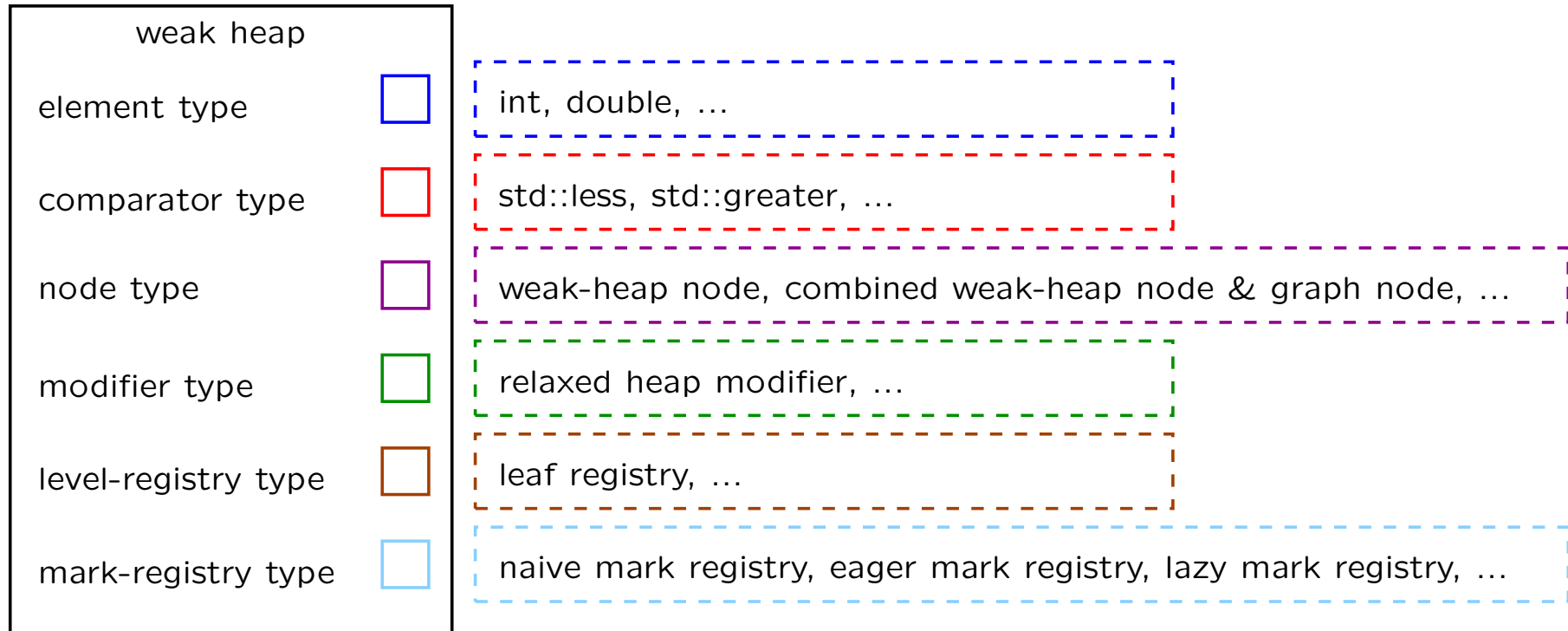
## Element comparisons per $n$

Structure Operation	CPH STL Fibonacci heap	LEDA 6.2 Fibonacci heap
<i>insert</i>		
$n$ : 10 000	0	1
$n$ : 100 000	0	1
$n$ : 1 000 000	0	1
<i>decrease</i>		
$n$ : 10 000	0	2
$n$ : 100 000	0	2
$n$ : 1 000 000	0	2
<i>extract-min</i>		
$n$ : 10 000	16.2	29.9
$n$ : 100 000	21.2	38.3
$n$ : 1 000 000	26.2	46.5

a factor of two speed-up

On my computer (Ubuntu, g++, with -O3)

# Parameterized Design



- comparators shared
- nodes shared
- transformations shared
- level registries shared
- mark registries shared

*a factor of two less code*

# What Is the Best?

---

## Our reference sequence

**Theory:** rank-relaxed weak heap

**Dijkstra—time:** binary heap

[Williams 1964]

**Dijkstra—comps:** weak heap

[Dutton 1993]

## Worst case per operation

*insert*—**time:** Fibonacci heap

[Fredman & Tarjan 1987]

*insert*—**comps:** Fibonacci heap

*decrease*—**time:** Fibonacci heap

*decrease*—**comps:** Fibonacci heap

*extract-min*—**time:** weak queue

[Vuillemin 1978]

*extract-min*—**comps:** weak heap

# Questions

---



- J. Bojesen, J. Katajainen, and M. Spork. Performance engineering case study: Heap construction. *ACM J. Exp. Algorithmics* **5**:Article 15, 2000
- A. Bruun, S. Edelkamp, J. Katajainen, and J. Rasmussen. Policy-based benchmarking of weak heaps and their relatives. *SEA 2010*, LNCS **6049**, pp. 459–435. Springer, 2010
- D. Cantone and G. Cinotti. QuickHeapsort, an efficient mix of classical sorting algorithms. *Theoret. Comput. Sci.* **285**(1):25–42, 2002
- J. Chen, S. Edelkamp, A. Elmasry, and J. Katajainen. In-place heap construction with optimized comparisons, moves, and cache misses. *MFCS 2012*, LNCS **7464**, pp. 259–270, Springer, 2012
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 3rd edition, The MIT Press, 2009
- V. Diekert and A. Weiß. Quickheapsort: Modifications and improved analysis. *CSR 2013*, LNCS **7913**, pp. 24–35, Springer, 2013
- R. D. Dutton. Weak-heap sort. *BIT* **33**(3):372–381, 1993
- S. Edelkamp, A. Elmasry, and J. Katajainen. Two constant-factor-optimal realizations of adaptive heapsort. *IWOCA 2011*, LNCS **7056**, pp. 195–208, Springer, 2011
- S. Edelkamp, A. Elmasry, and J. Katajainen. A catalogue of algorithms for building weak heaps. *IWOCA 2012*, LNCS **7643**, pp. 249–262, Springer, 2012
- S. Edelkamp, A. Elmasry, and J. Katajainen. The weak-heap data structure: Variants and applications. *J. Discrete Algorithms* **16**:187–205, 2012
- S. Edelkamp, A. Elmasry, and J. Katajainen. The weak-heap family of priority queues in theory and praxis. *CATS 2012*, Conferences in Research and Practice in Information Technology **128**, Australian Computer Society, pp. 103–112, 2012



- S. Edelkamp, A. Elmasry, and J. Katajainen. Optimal in-place heaps, 2013
- S. Edelkamp, A. Elmasry, and J. Katajainen. Weak heaps engineered. *J. Discrete Algorithms*, 2013
- S. Edelkamp and P. Stiegeler. Implementing Heapsort with  $n \log n - 0.9n$  and Quicksort with  $n \log n + 0.2n$  comparisons. *ACM J. Exp. Algorithmics* **7**:Article 5, 2002
- S. Edelkamp and I. Wegener. On the performance of weak-heapsort. *STACS 2000*, LNCS **1770**, pp. 254–266, Springer, 2000
- S. Edelkamp and A. Weiß. Quickmergesort: Efficient sorting with  $n \log n - 1.399n + o(n)$  comparisons on average, 2013
- A. Elmasry, C. Jensen, and J. Katajainen. Multipartite priority queues. *ACM Trans. Algorithms* **5**(1):Article 14, 2008
- J. Katajainen. The ultimate heapsort. *CATS 1998*, Australian Computer Science Communications **20**(3), pp. 87–96, Springer-Verlag Singapore, 1998
- D. E. Knuth. *Sorting and Searching*. The Art of Computer Programming **3**, 2nd edition, Addison Wesley Longman, 1998
- C. J. H. McDiarmid and B. A. Reed. Building heaps fast. *J. Algorithms* **10**(3):352–365, 1989
- J. Vuillemin. A data structure for manipulating priority queues. *Commun. ACM* **21**(4):309–315, 1978
- I. Wegener. The worst case complexity of McDiarmid and Reed’s variant of Bottom-Up Heapsort is less than  $n \log n + 1.1n$ . *Inform. and Comput.* **97**(1):86–96, 1992
- J. W. J. Williams. Algorithm 232: Heapsort. *Commun. ACM* **7**(6):347–348, 1964