Two Constant-Factor-Optimal Realizations of Adaptive Heapsort

Stefan Edelkamp[1)] Amr Elmasry[2)]

Jyrki Katajainen[2)]

[1)] Universität Bremen
[2)] Københavns Universitet

These slides and all our programs are available via my home page
http://www.diku.dk/~jyrki

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.

1. Which of the two has more order?

$\langle 1, 3, 2, 7, 5, 4, 6 \rangle$ $\langle 7, 6, 1, 5, 2, 4, 3 \rangle$

Some measures of disorder

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

measure	definition
Osc	$\sum_{j=1}^{n-1} \left\lvert \left\{ i \mid 1 \leq i \leq n \text{ and } \min\{x_j, x_{j+1}\} < x_i < \max\{x_j, x_{j+1}\} \right\} \right\rvert$
Inv	$\left\lvert \left\{ (i,j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j \right\} \right\rvert$
Max	the maximum distance an element is from its correct position
$Runs$	$\left\lvert \left\{ i \mid 1 \leq i \leq n \text{ and } x_i > x_{i+1} \right\} \right\rvert + 1$

What is the amount of disorder in a sequence of length n that is in reversed sorted order?

Optimality

measure	asymptotically optimal (running time)	constant-factor-optimal (# element comparisons)
Osc	$O(n \lg (Osc/n))$	$\leq n \lg (Osc/n) + O(n)$
Inv	$O(n \lg (Inv/n))$	$\leq n \lg (Inv/n) + O(n)$
Max	$O(n \lg (Max))$	$\leq n \lg (Max) + O(n)$
$Runs$	$O(n \lg (Runs))$	$\leq n \lg (Runs) + O(n)$

[Levcopoulos & Petersson 1993]
[Guibas, McCreight, Plass & Roberts 1977]
[Estivill-Castro & Wood 1989]
[Mannila 1985]

Natural mergesort is an example of an adaptive sorting algorithm that is constant-factor-optimal; this is with respect to $Runs$.

[Knuth 1973, Section 5.2.4]

Local insertionsort

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

1 Construct an empty finger tree $\mathcal{F}$
2 $hint \leftarrow 0$
3 **for** $i \in \{1, 2, \ldots, n\}$
4 $\quad hint \leftarrow \mathcal{F}.insert(x_i,\ hint)$
5 **for** $j \in \{1, 2, \ldots, n\}$
6 $\quad x_j \leftarrow \mathcal{F}.extract\text{-}min()$

Idea: Jump over only a few elements in $insert$; the cost of $insert$ is $O(\lg \Delta)$, where Δ is the jump distance.

[Guibas, McCreight, Plass & Roberts 1977]

2. Is local insertionsort easy to implement?

Yes **No**

☐ ☐

3. Is local insertionsort optimal?

Yes **No**

☐ ☐

4. Is local insertionsort fast in practice?

Yes **No**

☐ ☐

Problem

Theory: Local insertionsort is asymptotically optimal with respect to Osc, Inv, Max, and $Runs$.

[Guibas, McCreight, Plass & Roberts 1977]
[Mannila 1985]
[Katajainen, Levcopoulos & Petersson 1989]

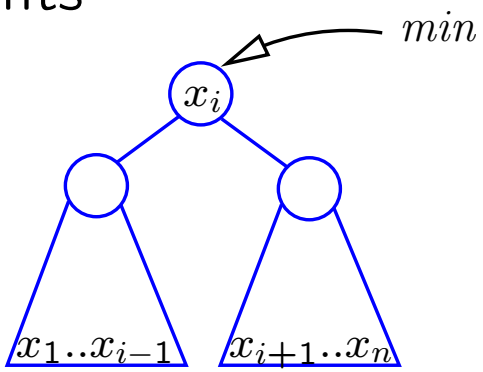
Practice: Only a few publicly-available implementations of finger trees exist; an implementation in the Haskell core libraries and an implementation in OCaml exists, and a C# implementation was published in 2008.

[http://en.wikipedia.org/wiki/Finger_tree]

Adaptive heapsort

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

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



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

[Levcopoulos & Petersson 1993]

Theoretical race

For priority queue $\mathcal{Q}$, the number of element comparisons performed is bounded by $\beta n \lg(Osc/n) + O(n)$.

$\mathcal{Q}$	β	reference
binary heap	3	
combined *extract-min insert*	2.5	[Levcopoulos & Petersson 1993]
binomial queue	2	[folklore]
weak heap	2	
combined *extract-min insert*	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!

Our contributions

Weak heap: $insert$: $O(1)$ amortized time; $extract\text{-}min$: $O(\lg n)$ worst-case time including at most $\lg n + O(1)$ element comparisons

Weak queue: $insert$: $O(1)$ amortized time; $extract\text{-}min$: $O(\lg n)$ worst-case time including at most $\lg n + O(1)$ element comparisons

Adaptive heapsort: constant-factor-optimal with respect to Osc, Inv, Max, and $Runs$

Idea: Temporarily store the inserted elements in a buffer and, once it is full, move its elements to the main structure using an efficient bulk-insertion procedure.

Array-based solution: Weak heap

n: # elements

k: # elements in the *buffer*

$[a_i | i \in [0..n-1]]$, a_i element

$[r_i | i \in [0..n-k-1]]$, $r_i \in \{0, 1\}$

$[b_i | i \in [0..k-1]] \equiv [a_i | i \in [n-k..n-1]]$

$min_{heap} \equiv a_0$, if $n > 0$

$min_{buffer} \equiv b_0$, if $k > 0$

Root a_0 has no left child

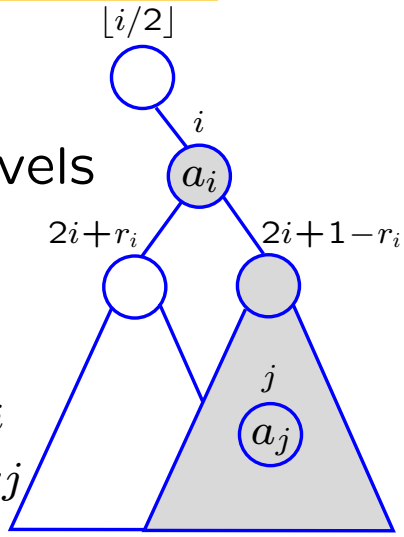
Leaves at the last two levels

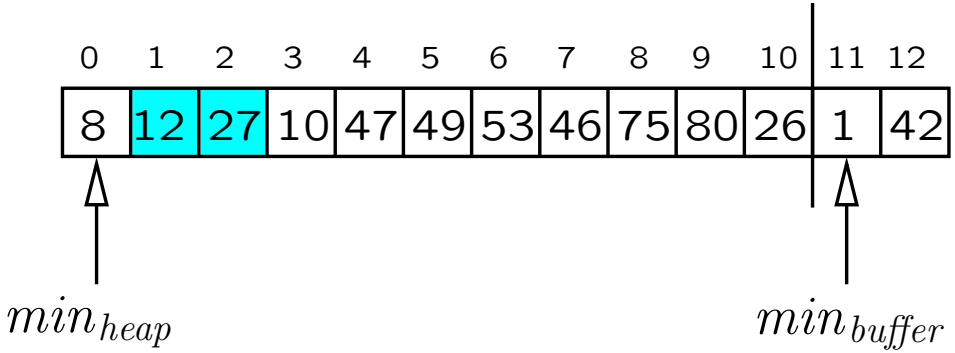
Parent of a_i: $a_{\lfloor i/2 \rfloor}$

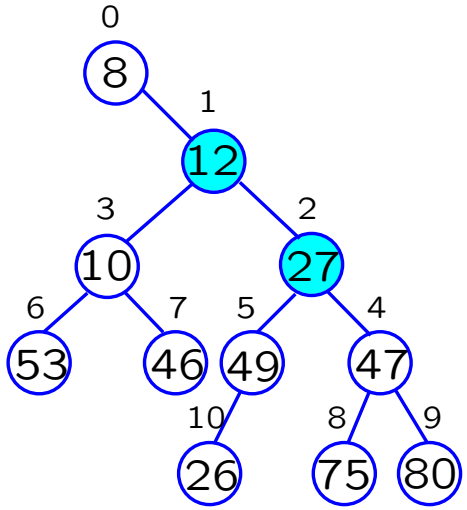
Left child of a_i: a_{2i+r_i}

Right child of a_i: a_{2i+1-r_i}

Weak-heap order: $a_i \not> a_j$







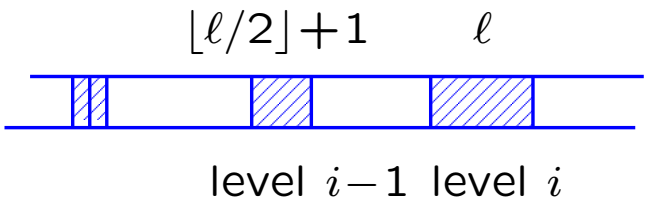
Bulk insertion in a weak heap

Algorithmic ideas

- Make the $buffer$ part of the $heap$ when $k = \lceil \lg n \rceil$.
- Fix the $heap$ bottom up level by level.

I) Find the distinguished ancestors for the levels with more than two nodes.

II) Traverse the two remaining paths to the root.



Analysis

- The number of nodes involved at most $2k + 2\lceil \lg n \rceil$
- One element comparison per node

∴ at most 4 **comparisons** per element

I) At most $(2k + o(k))/2^j$ of the nodes need j ancestor checks, where $j \geq 1$

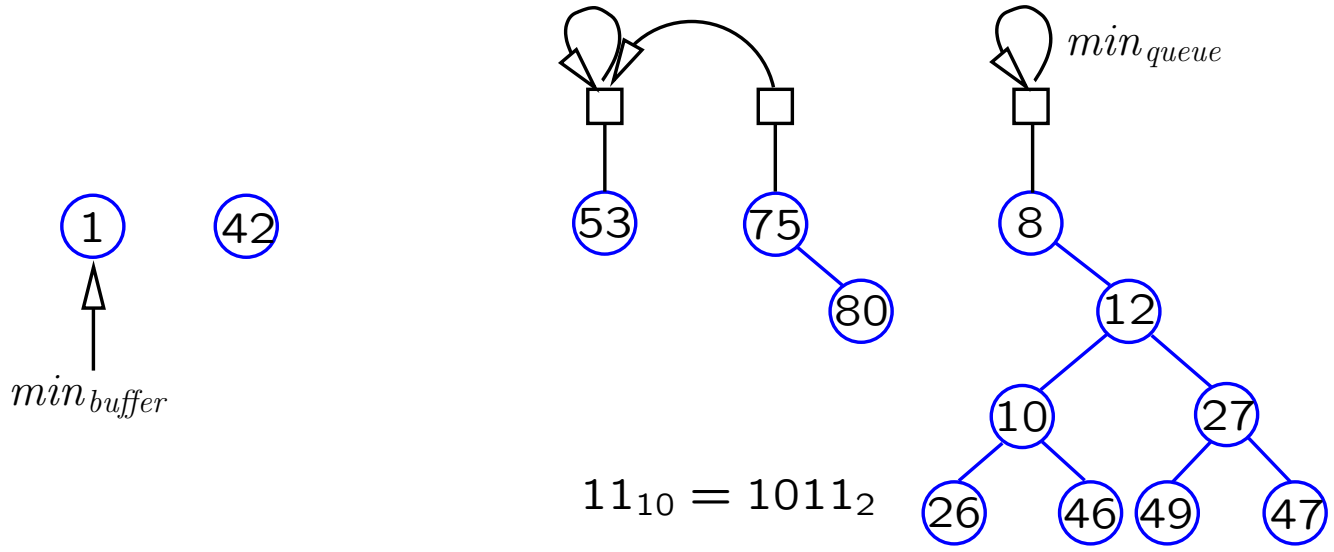
II) On the two paths at most $2\lceil \lg n \rceil$ ancestor checks

∴ $O(1)$ the **amortized cost** per element

Pointer-based solution: Weak queue

n: # elements
k: # elements in the $buffer$
Buffer: a singly-linked list
Prefix-minimum pointers: roots
Perfect weak heaps: left-child and right-child pointers for each node

$insert$: mimics an increment for a binary counter
$extract\text{-}min$: $borrow$-based



Bulk insertion in a weak queue

Algorithmic ideas

- Flush the $buffer$ out of its elements when $k = \lceil \lg n \rceil$.
- Perform normal $insert$'s without updating the prefix-minimum pointers.
- Update the prefix-minimum pointers once.

Analysis

- Give 1 € for each root
- Give 1 € for each $insert$
- Money at the roots pays the linkings of two heaps of the same size; money that is not used for linkings pays the pointer updates.

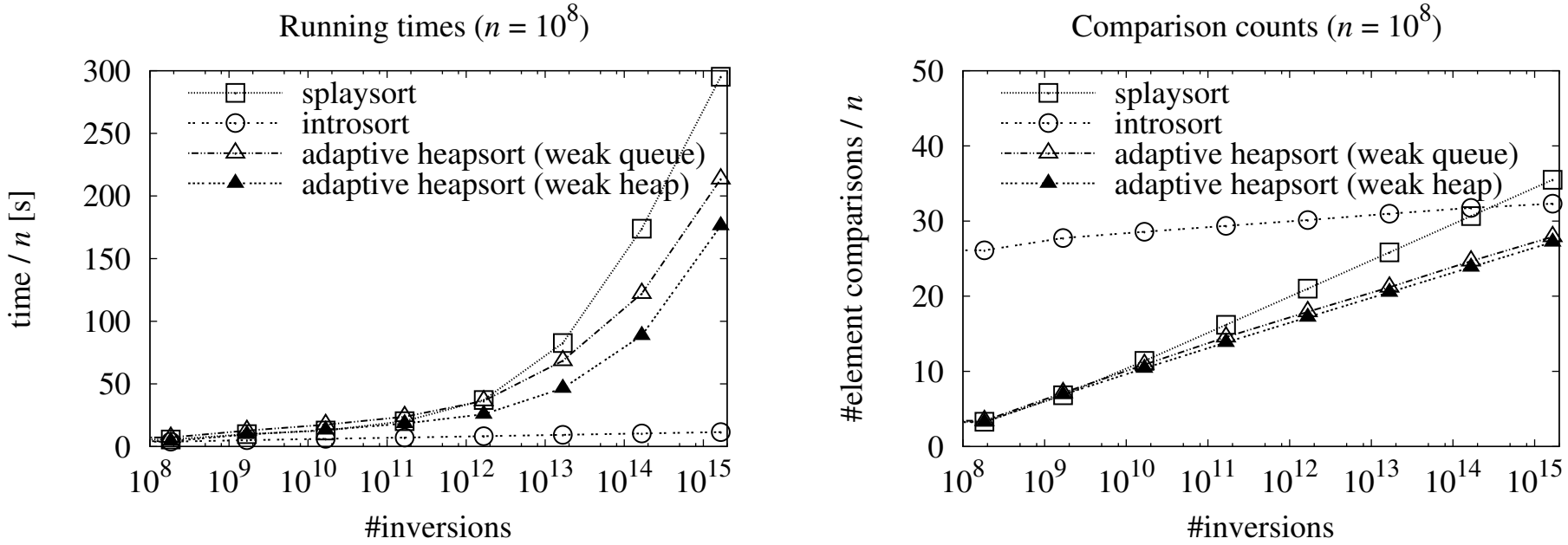
$\therefore$ at most 2 **comparisons** per element

5. Is adaptive heapsort practical?

Yes **No**

☐ ☐

Experiments



CPU time used and the number of element comparisons performed by different sorting algorithms for $n = 10^8$.

Further work

Cache behaviour: Heapsort has a bad cache performance. Can you improve this?

Space requirements: Adaptive heapsort has high space requirements. Can you make the algorithm more space economical?

Low-order constants: For our best implementation the number of element comparisons performed is bounded by $n \lg (1 + Osc/n) + 5.5n$. Can you improve this?

Practical performance: Our programs are publicly available. Can you beat them?