A catalogue of algorithms for building weak heaps

Stefan Edelkamp

University of Bremen

Jyrki Katajainen

Amr Elmasry

University of Copenhagen

These slides are available at http://www.cphstl.dk

What this paper is about:

- An array-based weak heap is an efficient data structure for realizing an elementary priority queue.
- In this paper we focus on the construction of a weak heap.
- As the optimization criteria, we consider how to reduce the number of instructions, branch mispredictions, cache misses, and element moves.
- For most of the algorithms considered, we also study their effectiveness in practice.

The development of weak heaps

- The idea was introduced by [Peterson '87].
- Implementation details to support the operations *find-min* in $O(1)$ time, *insert* and *extract-min* in $O(\lg n)$ time involving $\lceil \lg n \rceil$ element comparisons [Dutton '93]
- Use for sorting, analysis, and support of *construct* in $O(n)$ time [Edelkamp and Wegner '00]

The development of weak heaps (cont.)

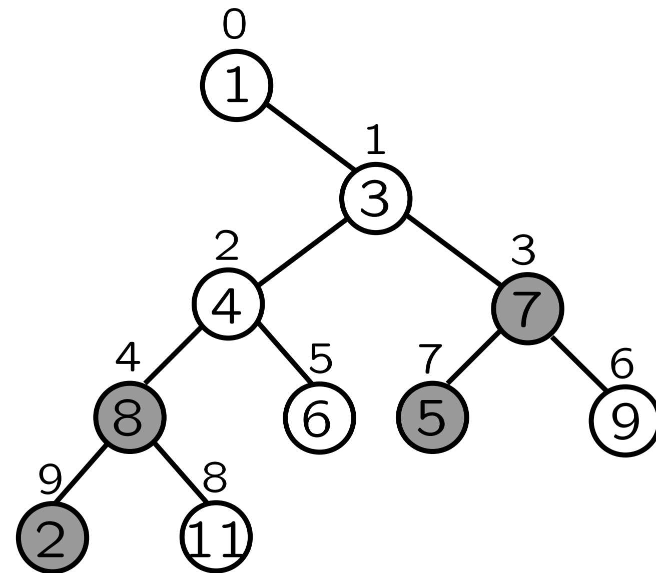
data structure	*construct*	*find-min*	*insert*	*extract-min*
binary heap	$2n$	0	$\lceil \lg n \rceil$	$2\lceil \lg n \rceil$
weak heap	$n - 1$	0	$\lceil \lg n \rceil$	$\lceil \lg n \rceil$

The number of element comparisons per operation

- Buffered variant supporting amortized constant *insert* [Edelkamp, Elmasry and Katajainen '11]
- Pointer-based variants supporting worst-case constant *decrease* and *insert* [Edelkamp, Elmasry and Katajainen '12]

The data structure

- The root of the entire tree has no left child.
- Except for the root, the nodes that have at most one child are at the last two levels only. Leaves at the last level can be scattered, i.e. the last level is not necessarily filled from left to right.
- Each node stores an element that is smaller than or equal to every element stored in the right subtree of this node.



A weak heap with reverse bits (set for grey nodes).

The data structure (cont.)

- An element array a and an array r of *reverse bits*.
- The index of the left child of a_i is $2i + r_i$, the index of the right child is $2i + 1 - r_i$, and the index of the parent is $\lfloor i/2 \rfloor$.
- Subtrees can be reversed in constant time by setting $r_i \leftarrow 1 - r_i$.
- The *distinguished ancestor* $d\text{-}ancestor(j)$ of a_j, $j \neq 0$, is the parent of a_j if a_j is a right child, and the distinguished ancestor of the parent of a_j if a_j is a left child.

Standard building procedure

Algorithm $d\text{-}ancestor(j\text{: index})$

```
while (j & 1) = r_{⌊j/2⌋}
    j ← ⌊j/2⌋
return ⌊j/2⌋
```

Algorithm $join(i, j\text{: indices})$

```
if a_j < a_i
    swap(a_i, a_j)
    r_j ← 1 − r_j
    return false
return true
```

Algorithm $construct(a\text{: array of } n \text{ elements}, r\text{: array of } n \text{ bits})$

```
for i ∈ {0, 1, ..., n−1}
    r_i ← 0
for j = n−1 to 1 step −1
    i ← d-ancestor(j)
    join(i, j)
```

Instruction optimization

Algorithm $d\text{-}ancestor(j\text{: index})$

$z \leftarrow trailing_zero_count(j)$
return j >> $(z + 1)$

program n	standard	instruction optimized
2^{10}	10.49	7.86
2^{15}	10.26	7.49
2^{20}	10.61	7.83
2^{25}	10.96	8.16

The running time in nanoseconds divided by n

Branch optimization

Algorithm $join(i, j\colon \text{indices})$

```
if a_j < a_i
    swap(a_i, a_j)
    r_j ← 1 − r_j
    return false
return true
```

if $a_j < a_i$
    swap(a_i, a_j)
    $r_j \leftarrow 1 - r_j$
    return $false$
return $true$

Algorithm $join(i, j\colon \text{indices})$

$smaller \leftarrow (a_j < a_i)$
$delta \leftarrow smaller * (j - i)$
$k \leftarrow i + delta$
$\ell \leftarrow j - delta$
$t \leftarrow a_\ell$
$a_i \leftarrow a_k$
$a_j \leftarrow t$
$r_j \leftarrow smaller$

Branch optimization (cont.)

program n	instruction optimized	branch optimized
2^{10}	7.86	6.28
2^{15}	7.49	6.39
2^{20}	7.83	6.72
2^{25}	8.16	7.08

The running time in nanoseconds divided by n

program n	standard	instruction optimized	branch optimized
2^{10}	1 061	512	1
2^{15}	34 171	16 385	1
2^{20}	1 093 963	524 110	2
2^{25}	35 005 433	16 776 271	34

The total number of branch mispredictions

Alternative construction: Don't look upwards

Algorithm *sift-down*(j: index)

$k \leftarrow 2j + 1 - r_j$
while $2k + r_k < n$
    $k \leftarrow 2k + r_k$
while $k \neq j$
    $join(j, k)$
    $k \leftarrow \lfloor k/2 \rfloor$

Algorithm *construct*(a: array of n elements, r: array of n bits)

for $i \in \{0, 1, \ldots, n-1\}$
    $r_i \leftarrow 0$
for $j = \lfloor n/2 \rfloor - 1$ **to** 0 **step** -1
    sift-down(j)

Practical behaviour is not promising due to cache effects.

Cache Optimization: Depth-First Construction

Algorithm $\mathit{df\text{-}construct}(i, j\text{: indices})$

```
if j < ⌊n/2⌋
    df-construct(j, 2j + 1)
    df-construct(i, 2j)
join(i, j)
```

Algorithm $\mathit{construct}(a\text{: array of } n \text{ elements}, r\text{: array of } n \text{ bits})$

```
for i ∈ {0, 1, ..., n−1}
    r_i ← 0
if n > 1
    df-construct(0, 1)
```

Cache Optimization: Depth-First Construction

program n	standard	instruction optimized	alternative	cache optimized
2^{10}	1.25	1.25	1.25	1.25
2^{15}	1.25	1.25	1.25	1.25
2^{20}	1.72	1.52	6.38	1.25
2^{25}	2.47	2.23	7.36	1.25

The number of cache misses divided by n/B

(where B is the length of the cache lines in words)

Move Optimization: Trading Swaps for Delayed Moves

Algorithm *df-construct*(i: index)

```
j ← 2i + 1
while j < ⌊n/2⌋
    df-construct(j)
    j ← 2j
sift-down(i)
```

Algorithm *construct*(a: array of n elements, r: array of n bits)

```
for i ∈ {0, 1, ..., n−1}
    r_i ← 0
if n > 1
    df-construct(0)
```

Move Optimization: Trading Swaps for Delayed Moves

program	standard		move optimized	
n	random	decreasing	random	decreasing
2^{10}	1.49	2.99	1.16	1.99
2^{15}	1.49	2.99	1.16	1.99
2^{20}	1.49	3.00	1.16	2.00
2^{25}	1.50	3.00	1.16	2.00

The number of element moves divided by n

Repeated insertions

```
Algorithm sift-up(j: index)

r_j ← 0
while j ≠ 0
    i ← d-ancestor(j)
    if join(i, j)
        break
    j ← i
```

```
Algorithm construct(a: array of n elements, r: array of n bits)
for k = 1 to n − 1
    sift-up(k)
```

Theorem. The number of element comparisons performed while constructing a weak heap using n in-a-row insertions is at most $3n$.

New Memory Layout: Less Work, Different Outcome

Algorithm $\mathit{dual\text{-}construct}(a$: array of n elements, r: array of n bits)

$\ell \leftarrow n$
while $\ell > 1$
  for $i \in \{0, \ldots, \lfloor \ell/2 \rfloor\}$
    $\mathit{join}(i, \lfloor \ell/2 \rfloor + i)$
  $\ell \leftarrow \lfloor \ell/2 \rfloor$

To access the parent of a node a_i whose depth is d, we need an extra check. If $i \geq 2^{d-1} + 2^{d-2}$, the parent is at location $i - 2^{d-1}$. Otherwise, the parent is at location $i - 2^{d-2}$.

Conclusions

- The weak heap is an amazingly simple and powerful structure.
- If perfectly balanced, weak heaps resemble heap-ordered binomial trees.
- We introduced several construction methods for weak heaps.
- Contrary to binary heaps, repeated insertions lead to a constant number of element comparisons per inserted element.
- A weak heap can be converted to a binary heap using 0.625 element comparisons.