

In-Place Data Structures: Which Complexity Measures Do Matter?

Jyrki Katajainen^{1,2}

Jingsen Chen³, Stefan Edelkamp⁴, Amr Elmasry⁵,
Max Stenmark²

¹ Københavns Universitet

² Jyrki Katajainen and Company

³ Luleå Tekniska Universitet

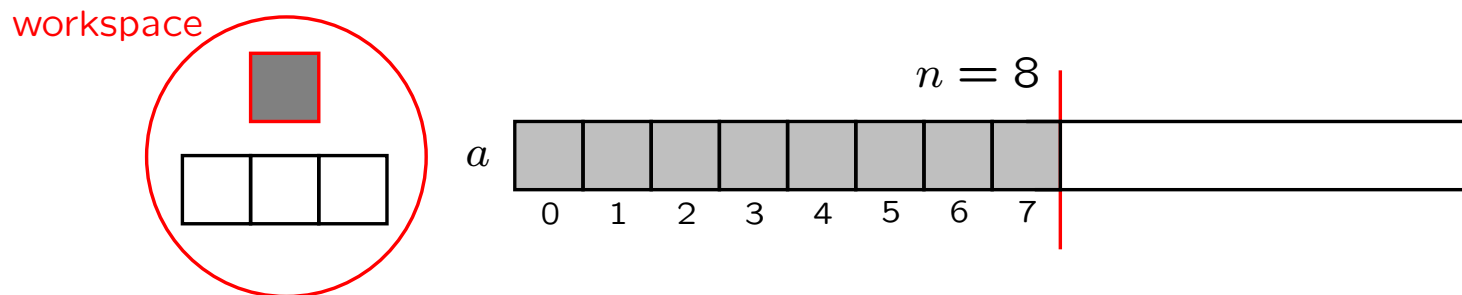
⁴ Universität Bremen

⁵ Alexandria University

Model of computation

Available

- An **infinite** array a suitable for storing elements
- $O(1)$ number of other memory locations for storing elements
- $O(1)$ number of other variables (counters, indices, bit strings of length $\lceil \lg(1 + n) \rceil$)



Requirement

- If the data structure stores n elements, these elements must be kept in the first n locations of a .

Coverage

In-place data structures

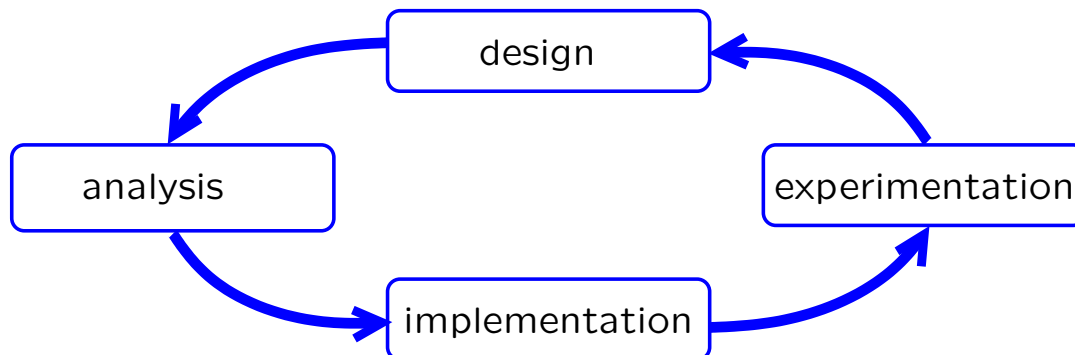
- Binary heaps
- Static search trees

Complexity measures

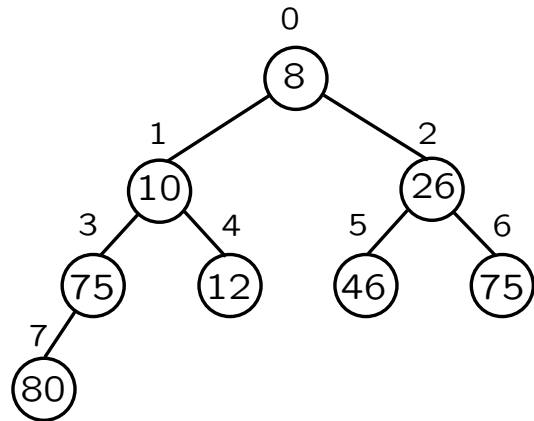
- Space utilization
- # Element comparisons
- # Element moves
- # Cache misses
- # Branch mispredictions
- Running time

Aha! The whole cycle

What is important?



Binary heaps



$n = 8$

a	8	10	26	75	12	46	75	80
	0	1	2	3	4	5	6	7

$left-child(i)$
return $2i + 1$

$right-child(i)$
return $2i + 2$

$parent(i)$
return $\lfloor (i - 1) / 2 \rfloor$

$construct()$
for ($i = parent(n - 1); i \geq 0; --i$)
 $sift-down(i)$

$minimum()$
return $a[0]$

$insert(x)$
 $a[n] = x$
 $sift-up(n)$
 $n += 1$

$extract-min()$
 $min = a[0]$
 $n -= 1$
 $a[0] = a[n]$
 $sift-down(0)$
return min

Experimental setup

Standard benchmark

- construct a heap of size n

Input data

All elements are of type `int`

Repetitions

Repeat each experiment r times, $r = 2^{26}/n$

Reported value

Measurement result divided by $r \times n$

Processor

Intel[®] Core[™] i5-2520M
CPU @ 2.50GHz × 4

Memory system

12-way-associative L3 cache:
3 MB
cache lines: 64 B
main memory: 3.8 GB

Operating system

Ubuntu 12.04 (Linux kernel
3.2.0-29-generic)

Compiler

g++ compiler (gcc version
4.6.3) with optimization `-O3`

Reduce # element comparisons

Inventor	<i>construct</i>	<i>insert</i>	<i>extract-min</i>	Extra Space
Williams/Floyd	$2n$	$\sim \lg n$	$\sim 2 \lg n$	$O(1)$ words
Gonnet & Munro	$1.625n$			$\Theta(n)$ words
Gonnet & Munro		$\sim \lg \lg n$	$\sim \lg n + \log^* n$	$O(1)$ words
Lower bounds	$\sim 1.37n$	$\Omega(1)$	$\sim \lg n$	$\Omega(1)$ words

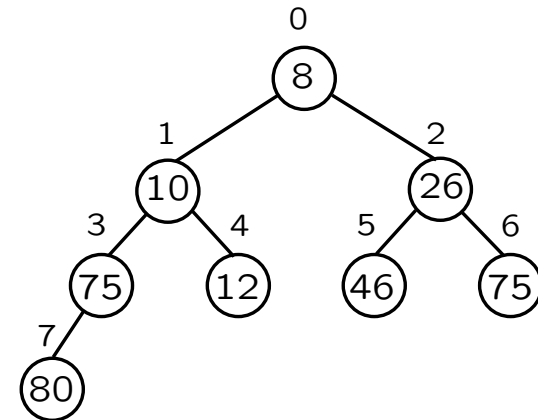
construct: Use a binomial tree in the construction

insert: Binary search on the *siftup* path

extract-min: $\lg n - \lg \lg n$ levels down along the *siftdown* path, *siftup* or recur further down

Floyd's heap-construction program

```
1 template <typename position, typename index, typename comparator>
2 void siftdown(position a, index i, index n, comparator less) {
3     typedef typename std::iterator_traits<position>::value_type element;
4     element copy = a[i];
5 loop:
6     index j = 2 * i;
7     if (j <= n) {
8         if (j < n)
9             if (less(a[j], a[j + 1]))
10                j = j + 1;
11         if (less(copy, a[j])) {
12             a[i] = a[j];
13             i = j;
14             goto loop;
15         }
16     }
17     a[i] = copy;
18 }
19
20 template <typename position, typename comparator>
21 void make_heap(position first, position beyond, comparator less) {
22     typedef typename std::iterator_traits<position>::difference_type index;
23     position const a = first - 1;
24     index const n = beyond - first;
25     for (index i = n / 2; i > 0; --i)
26         siftdown(a, i, n, less);
27 }
```



$n = 8$

a	8	10	26	75	12	46	75	80
	0	1	2	3	4	5	6	7

[Floyd 1964]

Remove an easy-to-predict if

opt₁: Make sure that `siftdown` is always called with an odd n

```
if (j < n)
  ...
for (index i = n / 2; i > 0; --i)
  siftdown(a, i, n, less);
```

→

```
template <typename position, typename index, typename comparator>
void siftup(position a, index j, comparator less) {
  ...
}
```

```
index const m = (n & 1) ? n : n - 1;
for (index i = m / 2; i > 0; --i)
  siftdown(a, i, m, less);
siftup(a, n, less);
```

Construction time [ns]

n	F	F ₁
2 ¹⁰	7.5	7.1
2 ¹⁵	7.4	7.0
2 ²⁰	8.2	7.9
2 ²⁵	8.9	8.4

Remove a hard-to-predict if

opt₂: Interpret the result of a comparison as an integer and use this value in normal index arithmetic

```
if (condition) {  
    j = j + 1;  
}
```

→

```
j = j + condition;
```

Construction time [ns]

n	F_1	F_{12}
2^{10}	7.1	4.8
2^{15}	7.0	4.9
2^{20}	7.9	6.3
2^{25}	8.4	7.2

Lean programs

- A program has a **constant** number of unnested loops.
- Each loop is **branch-free**, except the final conditional branch at the end.
- A branch predictor is **static**: forward branches are not taken and backward branches are taken.
- Each such program induces $O(1)$ branch mispredictions in this model.

Theorem. Let \mathcal{P} be a program of length κ , measured in the number of assembly-language instructions. Assume that the running time of \mathcal{P} is $t(n)$ for an input of size n . There exists a program \mathcal{Q} of length $O(\kappa)$ that is equivalent to \mathcal{P} , runs in $O(\kappa t(n))$ time for the same input as \mathcal{P} , and induces $O(1)$ branch mispredictions.

[Elmasry, Katajainen 2012]

Reduce \neq element moves

opt₃: Do not make any element moves when the element at the root stays in its original location

```
element copy = a[i];
```

→

```
element copy;  
index k = 2 * i;  
k = k + less(a[k], a[k + 1]);  
if (less(a[i], a[k])) {  
    copy = a[i];  
    a[i] = a[k];  
}  
else {  
    return;  
}  
i = k;
```

Aha! Loop unrolling

Construction time [ns]

n	F_{12}	F_{123}
2^{10}	4.8	4.3
2^{15}	4.9	4.6
2^{20}	6.3	5.9
2^{25}	7.2	6.9

Element moves

n	F	F_{123}
2^{10}	1.73	1.52
2^{15}	1.74	1.53
2^{20}	1.74	1.53
2^{25}	1.74	1.52

Reduce # cache misses

opt₄: Visit the nodes in reverse depth-first order instead of reverse breadth-first order [Bojesen et al. 2000]

```
for (index i = n / 2; i > 0; --i)
    siftdown(a, i, n, less);
```

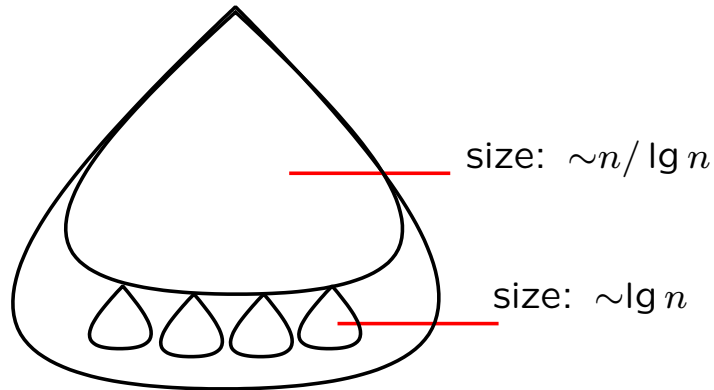
→

```
index j = n / 2;
index const i = j / 2;
while (j > i) {
    siftdown(a, j, n, less);
    index z = j;
    while ((z & 1) == 0) {
        z /= 2;
        siftdown(a, z, n, less);
    }
    --j;
}
```

Construction time [ns]

n	F	F₁₂₃	F₁₋₄
2^{10}	7.4	4.3	5.2
2^{15}	7.4	4.6	5.1
2^{20}	8.2	5.9	5.2
2^{25}	8.7	6.9	5.1

Making the GM algorithm in-place



1. Improve **GM**:
 $O(n)$ words $\rightarrow O(n)$ bits
2. Apply the improved algorithm for all bottom trees; keep the bits needed compactly in a word
3. Use **F**'s *sift*down approach for the top tree.

Element comparisons

$$\sim 2n \rightarrow \sim 1.625n$$

Element moves

$$\sim 2n \rightarrow \sim 2.125n$$

Cache misses

$$\sim \frac{n \lg B}{B} \rightarrow \sim \frac{n}{B}, \text{ assuming } B \lg n \ll M \text{ (} B \text{ block size; } M \text{ memory size)}$$

Construction time [ns]

n	F	GM
2^{10}	7.4	8.0
2^{15}	7.4	7.7
2^{20}	8.2	7.7
2^{25}	8.7	7.7

Construction time [ns]

n	std	F	F ₁₂₃	F ₁₋₄	GM
2 ¹⁰	10.7	7.4	4.3	5.2	8.0
2 ¹⁵	10.4	7.4	4.6	5.1	7.7
2 ²⁰	11.0	8.2	5.9	5.2	7.7
2 ²⁵	11.5	8.7	6.9	5.1	7.7

Instructions

n	std	F	F ₁₂₃	F ₁₋₄	GM
2 ¹⁵					
2 ²⁰	35.5	20.8	13.4	16.2	42.9
2 ²⁵					

Element comparisons Branches | mispredictions

n	std/F	GM
2 ¹⁰	1.98	1.80
2 ¹⁵	1.99	1.66
2 ²⁰	1.99	1.63
2 ²⁵	2	1.63

n	std	F	F ₁₂₃	F ₁₋₄
2 ¹⁰	5.39 0.96	4.53 0.81	2.17 0.27	2.42 0.47
2 ¹⁵	5.40 0.89	2.43 0.78	2.18 0.24	2.43 0.47
2 ²⁰	5.41 0.89	4.57 0.78	2.18 0.24	2.43 0.47
2 ²⁵	5.41 0.89	4.56 0.78	2.18 0.24	2.43 0.47

Element moves

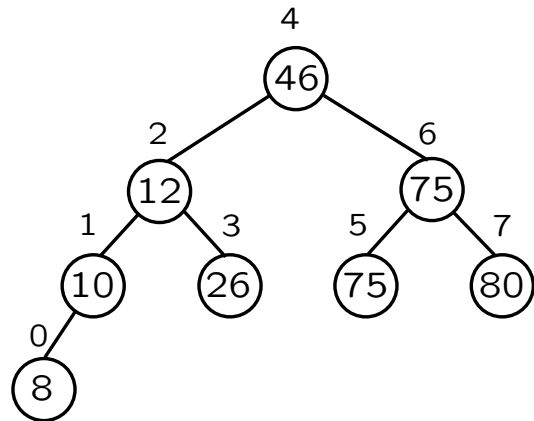
n	std	F	GM
2 ¹⁰	3.99	1.99	2.15
2 ¹⁵	3.99	1.99	2.39
2 ²⁰	4	1.99	2.38
2 ²⁵	4	2	2.38

I/Os | misses (per n/B)

n	std/F	F ₁₋₄	GM
2 ¹⁰	1.00 1.00	1.00 1.00	0.95 0.95
2 ¹⁵	5.66 1.00	1.03 1.00	1.03 1.00
2 ²⁰	5.87 4.94	1.04 1.00	— —
2 ²⁵	5.87 5.84	1.04 0.99	— —

GM	
3.60	0.66
2.39	0.38
—	—
—	—

Static search trees



$n = 8$

a	8	10	12	26	46	75	75	80
	0	1	2	3	4	5	6	7

$left-child(i)$
return ...

$right-child(i)$
return ...

```
construct()  
    sort( $a, a + n$ )
```

```
is-member( $x$ )
```

```
     $i = 0$ 
```

```
     $k = n$ 
```

```
    while  $i \neq k$ 
```

```
        if  $x < a[i]$ 
```

```
             $k = i$ 
```

```
             $i = left-child(i)$ 
```

```
        else if  $a[i] < x$ 
```

```
             $i = right-child(i)$ 
```

```
        else
```

```
            return yes
```

```
    return no
```

Static search trees vs. red-black trees

Standard benchmark

r random *is-member* queries, $r = 10^6$

Input data

All elements are of type `int`

Reported value

Measurement result divided by $r \times \lg n$

Search time [ns]

n	Sorted array	Red-black tree
2^{10}	6.8	5.5
2^{15}	6.9	10.9
2^{20}	14.7	36.6
2^{25}	32.1	64.3

Idea: Rely on two-way element comparisons

Three-way comparisons

```
is-member(x)
  i = 0
  k = n
  while i ≠ k
    if x < a[i]
      k = i
      i = left-child(i)
    else if a[i] < x
      i = right-child(i)
    else
      return yes
  return no
```

Two-way comparisons

```
is-member(x)
  i = 0
  j = -1
  k = n
  while i ≠ k
    if x < a[i]
      k = i
      i = left-child(i)
    else
      j = i
      i = right-child(i)
  if j == -1 or a[j] < x
    return no
  return yes
```

[Andersson 1991]

Performance of the optimized program

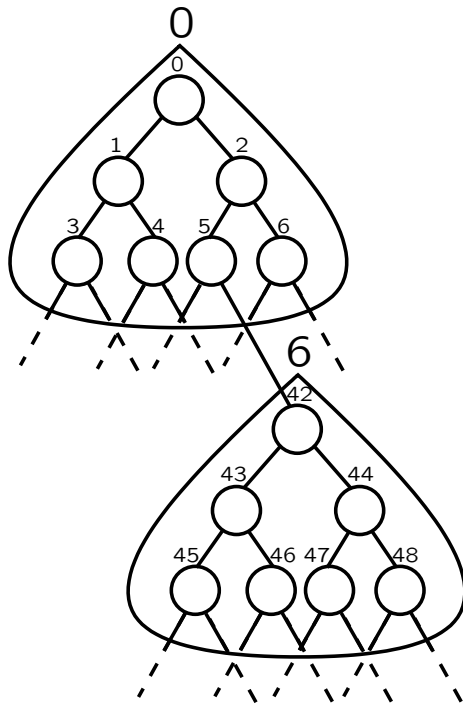
Element comparisons

$\sim 2 \lg n \rightarrow \sim \lg n$

Search time [ns]

n	Three-way	Two-way
2^{10}	6.6	5.5
2^{15}	7.7	6.9
2^{20}	15.6	14.7
2^{25}	33.3	32.1

Idea: Use another memory layout



$$F = 7$$

```
left-child(i)  
  j = i / F  
  if i <  $\lfloor F/2 \rfloor + j * F$   
    return  $2 * i - j * F + 1$   
  else  
    return  $F * (2 * i - (1 - F) * j - F + 2)$ 
```

```
right-child(i)  
  // Left as an exercise
```

```
construct()  
  // More complicated than sorting  
  ...
```

```
is-member(x)  
  // As before
```

Performance of implicit local search trees

Fat-node visits

$\sim \lg n - \lg F \longrightarrow \sim \lg n / \lg F$,
where F is the size of the fat
nodes measured in elements

Cache behaviour

All values are divided by $r \times \log_B n$, where B is the number of elements that fit in a cache line (16 in our test)

Search time [ns]; $F = 15$

n	Sorted array	Implicit local
2^{10}	5.5	15.0
2^{15}	6.9	15.0
2^{20}	14.7	16.3
2^{25}	32.1	20.1

n	Sorted array			Implicit local		
	Refs.	I/Os	Misses	Refs.	I/Os	Misses
2^{10}	7.40	0.00	0.00	10.56	0.00	0.00
2^{15}	6.93	2.00	0.00	9.46	0.39	0.00
2^{20}	6.70	3.20	0.73	8.80	0.81	0.12
2^{25}	6.56	3.37	3.01	9.00	1.01	0.51

Idea: Avoid conditional branches

Two-way comparisons

```
is-member(x)
  j = -1
  i = 0
  while i < n
    if x < a[i]
      i = left-child(i)
    else
      j = i
      i = right-child(i)
  if j == -1 or a[j] < x
    return no
  return yes
```

Hard-to-predict if removed

```
choose(condition, i, j)
  return j + condition * (i - j)

is-member(x)
  j = -1
  i = 0
  while i < n
    smaller = x < a[i]
    j = choose(smaller, j, i)
    i = choose(smaller, left-child(i), right-child(i))
  if j == -1 or a[j] < x
    return no
  return yes
```

Performance of the *if*-free programs

Conditional branches

$$\sim 2 \lg n \longrightarrow \sim \lg n$$

Branch mispredictions

$$\sim 0.5 \lg n \longrightarrow O(1)$$

Search time [ns]; $F = 15$

n	Sorted array		Implicit local	
	Two-way <i>if</i> -free		Two-way <i>if</i> -free	
2^{10}	5.5	6.6	15.0	15.0
2^{15}	6.9	7.2	15.0	14.2
2^{20}	14.7	20.7	16.3	15.6
2^{25}	32.1	45.8	20.1	22.8

Branch behaviour

n	Sorted array		Sorted array		Implicit local		Implicit local	
	Two-way		<i>if</i> -free		Two-way		<i>if</i> -free	
	\otimes	Mispred.	\otimes	Mispred.	\otimes	Mispred.	\otimes	Mispred.
2^{10}	2.20	0.62	1.20	0.10	3.56	0.89	1.32	0.11
2^{15}	2.07	0.57	1.13	0.07	3.21	0.86	1.12	0.07
2^{20}	2.05	0.55	1.10	0.05	3.05	0.84	1.05	0.05
2^{25}	2.04	0.54	1.08	0.04	3.18	0.82	1.09	0.04

Conclusions

- Branch optimization is only effective for small problem instances
- There is no reason to remove easy-to-predict conditional branches
- Cache optimization is effective for large problem instances, but it may make the solution slower for small problem instances
- It would be cool if branch optimization was done automatically by the compiler
- Element comparisons and element moves are still relevant in the cases where they are expensive

What else?

and open questions

- Devise an in-place priority queue for which *insert* requires $O(1)$ worst-case time and *extract-min* $O(\lg n)$ worst-case time including at most $\lg n + O(1)$ element comparisons.
- Can you improve the bounds for heap construction?

Some relevant papers

Binary heaps

- Jesper, Jyrki, Maz: Performance engineering case study: Heap construction, WAE 1999
- Claus, Jyrki, Fabio: Experimental evaluation of local heaps, CPH STL Report 2006-1
- Jingsen, Stefan, Amr, Jyrki: In-place heap construction with optimized comparisons, moves, and cache misses, MFCS 2012
- Amr, Jyrki: Bypassing the lower bounds for binary heaps

Branch prediction

- Amr, Jyrki: Lean programs, branch mispredictions, and sorting, FUN 2012
- Amr, Jyrki, Max: Branch mispredictions don't affect merge-sort, SEA 2012
- Amr, Jyrki: Microbenchmarking the search procedure for balanced search trees