14 November, 2014 ARCO Workshop, Copenhagen

What your teachers never told you about Fibonacci heaps

Jyrki Katajainen[1]

Stefan Edelkamp[2] **Jesper Larsson Träff**[3]

[1] University of Copenhagen
[2] University of Bremen
[3] Vienna University of Technology

These slides are available from my research information system (see http://www.diku.dk/~jyrki/ under Presentations).

Said about Fibonacci heaps

- Among all the data structures that guarantee constant `decrease-key` and logarithmic `delete-min` cost, Fibonacci heaps have remained the most popular to **teach** [Chan 2013]
- In spite of the many competitors ..., the original data structure remains one of the simplest to **describe** and **implement** [Kaplan, Tarjan, Zwick 2014]
- For the **analysis**, the potential function itself is not complicated, but one needs to first establish bounds on the maximum degree of the trees [Chan 2013]
- Fibonacci heaps do not perform well in **practice**, but pairing heaps do [Haeupler, Sen, Tarjan 2013]
- One reason Fibonacci heaps perform poorly is that they need an extra **pointer** per node [Haeupler, Sen, Tarjan 2013]
- The `decrease-key` operation uses a simple "cascading cut" strategy, which requires an extra **bit** per node [Chan 2013]

Inspirational source

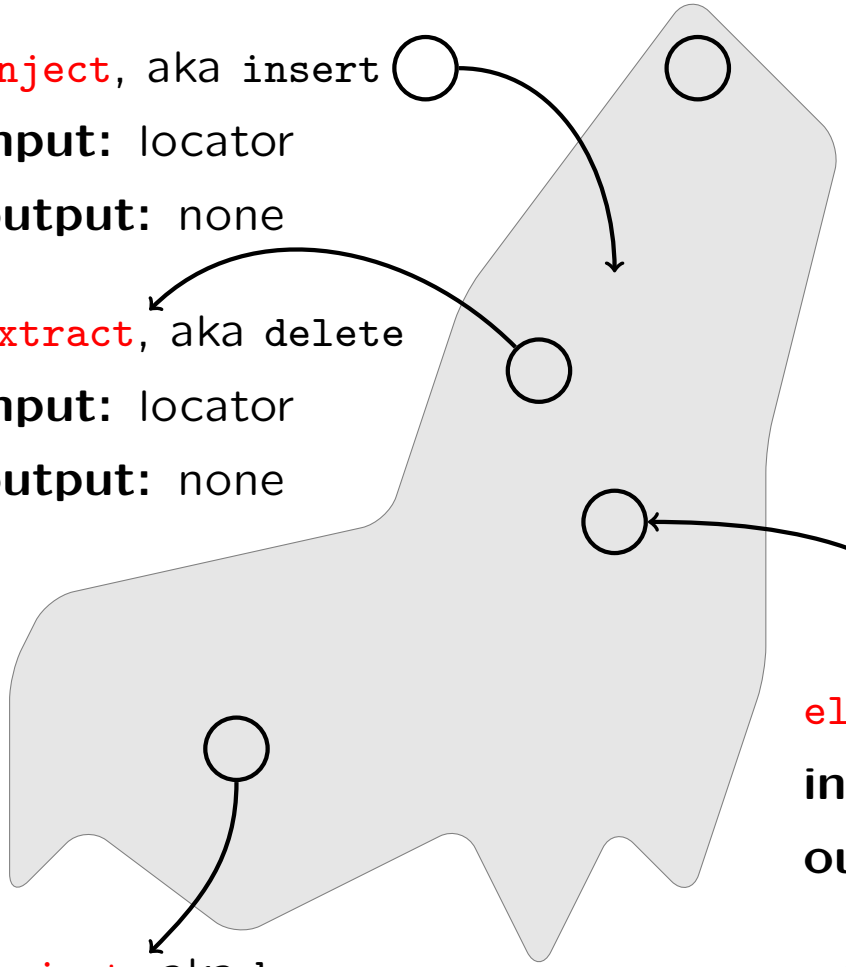
Kaplan, Tarjan, Zwick: Fibonacci heaps revisited, E-print arXiv:1407.5750, arXiv.org (2014)

- Simple Fibonacci heaps
- Pseudo-code on one page
- Beautiful analysis
- No experiments
- No validation of "simplicity"

What does word "simple" mean when we talk about data structures and algorithms?

How would you measure "simplicity"?

Classification of priority queues



`inject`, aka `insert`

input: locator

output: none

`top`, aka `minimum`

input: none

output: locator

`extract`, aka `delete`

input: locator

output: none

`extract-top`

$p \leftarrow$`top()`

`extract`(p)

`elevate`, aka `decrease-key`

input: locator, value

output: none

`eject`, aka `borrow`

input: none

output: locator

`merge`, aka `union`

input: two priority queues

output: one priority queue

Elementary

`top`

`inject`

`eject`

`extract-top`

Addressable

`top`

`inject`

`elevate`

`eject`

`extract`

Mergeable

As above +

`merge`

Software library vs. algorithm text

```
namespace cphstl {
  template <
    typename V, // value
    typename C, // comparator
    typename A, // allocator
    typename E, // encapsulator
    typename R, // realizator
    typename I, // iterator
    typename J  // iterator const
  >
  class mergeable_priority_queue {
    // 30+ convenience functions
  }
}

namespace cphleda {
  template <
    typename K, // key
    typename V, // information
    typename C, // comparator
    typename R, // realizator
    typename J  // iterator const
  >
  class p_queue {
    // 23 convenience functions
  }
}
```

```
namespace cphstl {
  template <
    typename E, // element
    typename C, // comparator
    typename N  // node
  >
  class fibonacci_heap {
    fibonacci_heap(C const &);
    ~fibonacci_heap();
    N* begin() const;
    N* end() const;

    N* top();
    void inject(N*);
    void elevate(N*, E const &);
    N* eject();
    void extract(N*);
    void merge(fibonacci_heap &);
    void swap(fibonacci_heap &);
  }
}
```

The interface of our realizators is similar to that used in a **modern** algorithm text.

Where is `eject` needed?

```
void clear() {
  while (n != 0) {
    I t = eject();
    destroy(t);
  }
}
```

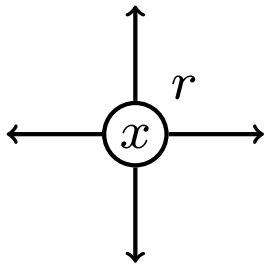
`clear` is used by

- `~mergeable_priority_queue()`
- `template <typename K> void insert(K, K)`
- `mergeable_priority_queue(mergeable_priority_queue const&)`
- `operator=(mergeable_priority_queue const&)`

In the last three, `clear` is needed to achieve exception safety.

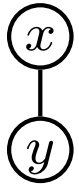
A collection of heap-ordered multi-ary trees

Node x



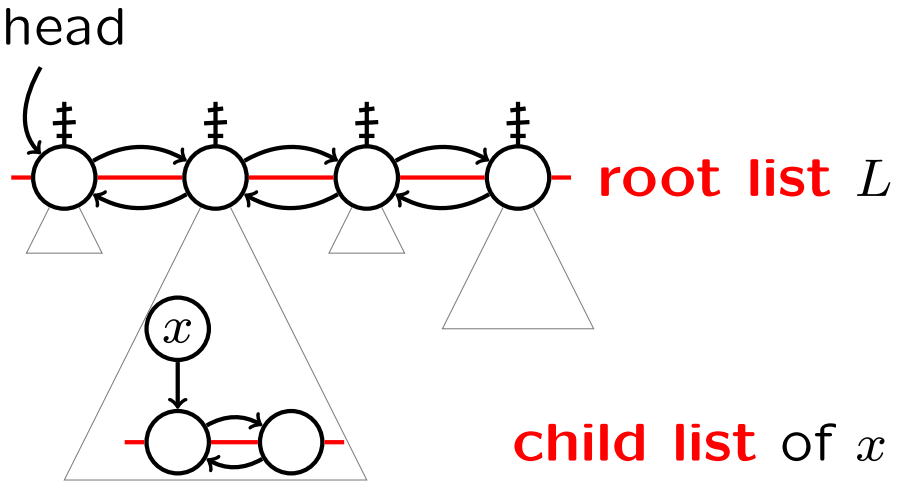
- element
- rank r
- state: unmarked or marked ✓
  (a marked node has lost a child)
- pointers: parent ↑, (first) child ↓,
  before ⟵, after ⟶

Heap order

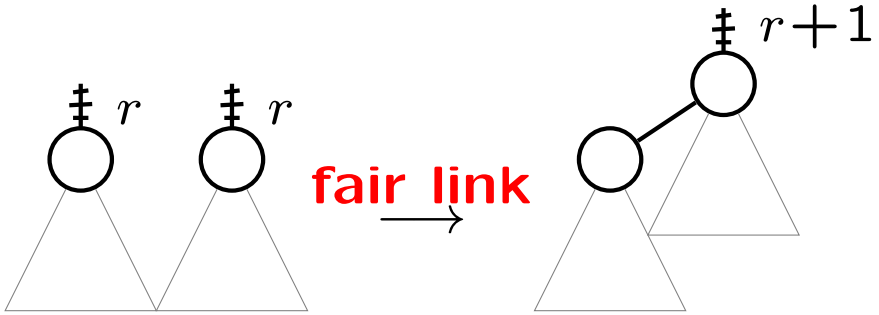


element at x
∨|
element at y

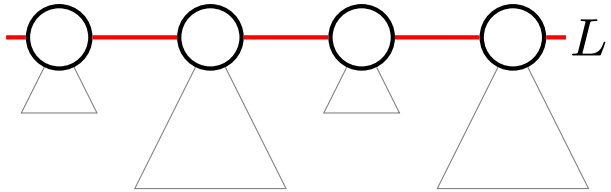
Structure



Basic primitive



Lazy Fibonacci heap



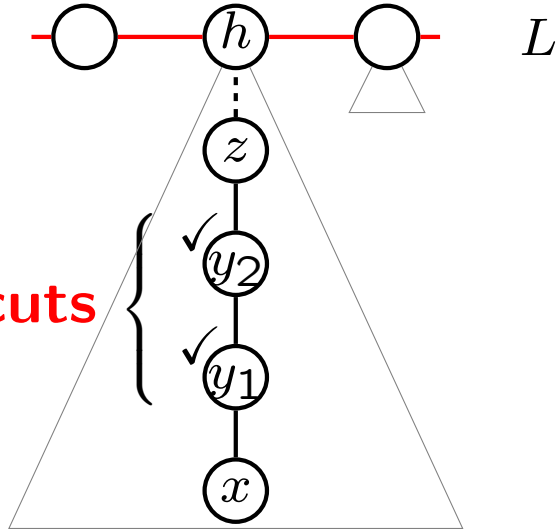
`top`

- do all fair links on L
- find the top in the remaining L

`inject`(x)

- append x to L

`elevate`$(x,\cdot)$



cascading cuts

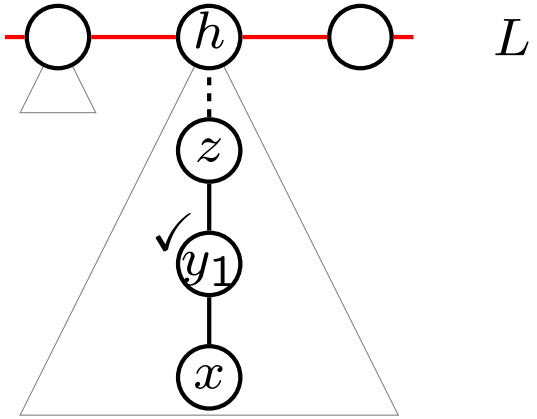
if $x \neq h$,

- cut x, move it to L
- cut y_i, unmark, move to L
- mark z (if $z \neq h$)

`eject`

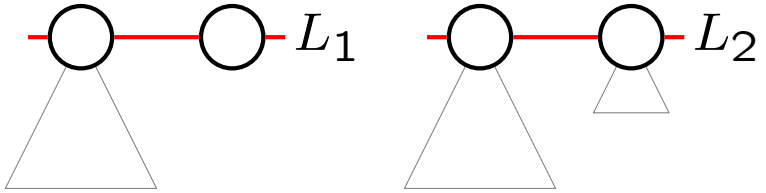
- take the first root x
- catenate its child list and L
- return x

`extract`(x)



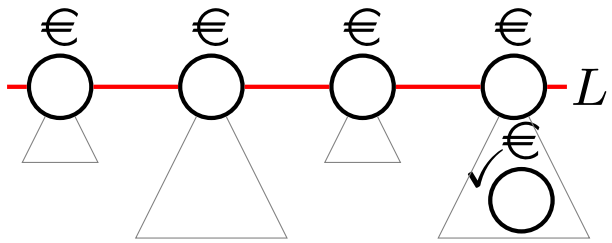
- cut x
- handle markings as above
- remove x, add its children to L

`merge`



- catenate L_1 and L_2

Analysis



Invariants: Deposit

– 1 comparison € at every root

– 1 comparison € and 1 work € at every marked node

– 1 work € for each operation

Fact: $max\text{-}rank \leq \log_\varphi n$ where φ is the golden ratio

---

`top`

- Use comparison € at the roots to pay fair links
- Give 1 comparison € for each node visited in top finding

`inject`

- Give 1 comparison € for the new root

`elevate`

- Give 1 comparison € for the node cut
- Use the work € at a marked node to move it to L
- Give 1 comparison € and 1 work € for the node marked

`eject`

- Give 1 comparison € for each created root

`extract`

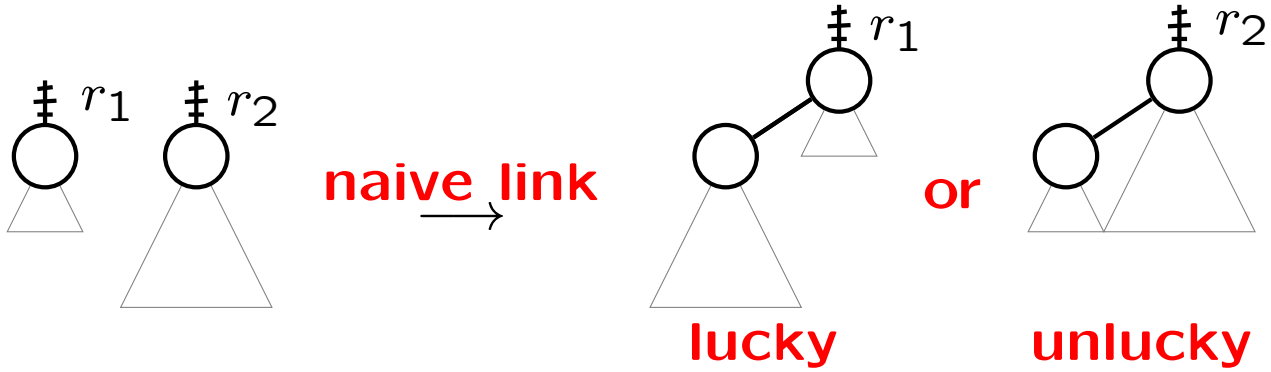
- Handle markings as above
- Give 1 comparison € for each created root

`merge`

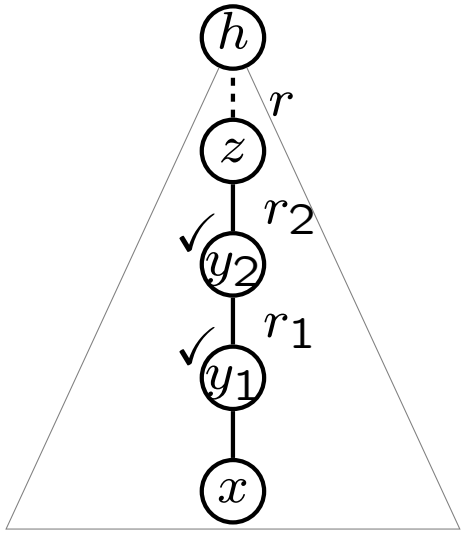
- No additional money needed

Simple Fibonacci heap: Two ingredients

1) A single tree instead of L; link the remaining roots naively



2) Cascading rank decreases instead of cascading cuts



if $x \neq h$,

- cut x
- unmark y_i, set its rank r_i to $\max\{r_i - 1, 0\}$
- mark z, set its rank r to $\max\{r - 1, 0\}$
- link x and h naively

Lines of code (LOC)

Lazy Fibonacci heap	realizator	160
	node	56
	Total	**216**
Simple Fibonacci heap	realizator	121
	node	56
	Total	**177**
Pairing heap	realizator	131
	node	36
	Total	**167**
Addressable multi-ary heap	realizator	110
	node	24
	Total	**134**

All programs were written in one-statement-per-line style. Comments, empty lines, lines only having a single parenthesis, debugging code, and assertions are not included in the counts.

Critical comparison

	Simple Fibonacci heap	Addressable 4-ary heap
Space	$5n$ words, n elements	$2n+O(\sqrt{n})$ words, n elements
`inject`	10 pointer assignments	$\lceil \lg n \rceil + 2$ pointer assignments
`elevate`	10 pointer assignments	$\lceil \lg n \rceil + 2$ pointer assignments
`eject`	$O(\lg n)$ amortized time	$O(1)$ worst-case time
`extract`	$2.88 \lg n$ element comparisons	$2 \lg n$ element comparisons
`merge`	$O(1)$ amortized time	$O(m \lg n)$ worst-case time

```
void elevate(N* x, E const & v) {
  (*x).element() = v;
  if (x == root) {
    return;
  }
  (*root).state(N::unmarked);
  decrease_ranks(x);
  cut(x);
  root = naive_link(x, root);
}
```

```
void elevate(locator_type pair, E const & v) {
  N* x = pair.first;
  (*x).element() = v;
  std::size_t c = (*x).index();
  while (c > 0) {
    std::size_t p = parent(c);
    N* u = sequence[p];
    if (! comparator((*u).element(), v)) {
      break;
    }
    set_slot(c, u);
    c = p;
  }
  set_slot(c, x);
}
```

siftup

Element-comparison game

What is the best solution when handling a request sequence consisting of n `inject`, m `elevate`, and n `extract-top` operations?

Simple Fibonacci heap	$4m + 2.88n \lg n$
Lazy Fibonacci heap	$2m + 2.88n \lg n$
Rank-relaxed weak heap	$2m + 1.5n \lg n$

Katajainen's 3rd conjecture: This request sequence can be handled with $2m + n \lg n + o(n \lg n)$ element comparisons in $O(m + n \lg n)$ worst-case time

Folklore: The bound $(1 + (1/k))\, m + (2k/\lg k)\, n \lg n$ is known to be achievable (for $k \geq 2$). Hint: Let a node loose at most k children.

Pointer-assignment game

pointer assignments per operation [`inject` increasing sequence; `elevate` random permutation, priority increase n]

	Lazy Fibonacci heap	Simple Fibonacci heap	Pairing heap	Addressable 4-ary heap
`inject`^{n}				
$n = 10^4$	10	10	8	15
$n = 10^5$	10	10	8	18
$n = 10^6$	10	10	8	22
`elevate`^{n}				
$n = 10^4$	12	10	10	16
$n = 10^5$	12	10	11	19
$n = 10^6$	12	10	11	22
`extract-top`^{n}				
$n = 10^4$	239	161	148	11
$n = 10^5$	304	203	186	13
$n = 10^6$	368	245	223	14

Running-time game

Average running time [ns] per operation [`inject` increasing sequence; `elevate` random permutation, priority increase n]

	Lazy Fibonacci heap	Simple Fibonacci heap	Pairing heap	Addressable 4-ary heap
`inject`^{n}				
$n = 10^4$	47	49	50	55
$n = 10^5$	45	50	51	60
$n = 10^6$	45	45	44	70
`elevate`^{n}				
$n = 10^4$	10	46	24	20
$n = 10^5$	104	116	98	113
$n = 10^6$	162	242	254	271
`extract-top`^{n}				
$n = 10^4$	294	242	153	119
$n = 10^5$	645	581	445	302
$n = 10^6$	1189	1093	1139	775

Special-sequence game

Request sequence `inject`^{n} (`elevate`^{k} `extract-top`)^n [`elevate` random alive]
Average running time [ns] divided by $kn + n \lg n$ **What is wrong?**

k	Lazy Fibonacci heap	Simple Fibonacci heap	Pairing heap	Addressable 4-ary heap
2 $n = 10^4$	25	19	14	21
$n = 10^5$	45	37	29	43
$n = 10^6$	58	51	40	83
4 $n = 10^4$	34	30	21	26
$n = 10^5$	62	57	44	56
$n = 10^6$	86	84	64	109
6 $n = 10^4$	39	37	25	29
$n = 10^5$	74	71	55	66
$n = 10^6$	106	112	64	129
8 $n = 10^4$	44	43	29	31
$n = 10^5$	85	83	65	72
$n = 10^6$	129	135	100	142

Fixing `top`

Problem: For lazy Fibonacci heap, `top` takes $O(|L| + max\text{-}rank)$ time

Fix 1: Keep a pointer to the node storing the top element

	Before	After
`top`	$A(n)$	$O(1)$
`inject`	$B(n)$	$B(n)+O(1)$
`elevate`	$C(n)$	$C(n)+O(1)$
`eject`	$D(n)$	$B(n)+2\cdot D(n)+O(1)$
`extract`	$E(n)$	$A(n)+E(n)+O(1)$
`merge`	$F(m,n)$	$F(m,n)+O(1)$

Cost of `top`: worst-case $O(1)$

Cost of `elevate`: 3 comparison €

Element-comparison game:

$3m + 2.88n \lg n$

Fix 2: Reimplement `top` [Kaplan, Tarjan, Zwick 2014]

- Do all fair links on L
- Do all naive links on the rest
- Return the single root

Exercise: Implement so that the actual running time is $O(|L|)$

Goody: All element comparisons explicitly represented in the data structure

Element-comparison game:

$2m + 2.88n \lg n$

Yet another special sequence

Request sequence $(\texttt{inject top})^n$ $(\texttt{extract top})^n$ [extract random]
Average running time [ns] divided by n

	Lazy	Fix 1	Fix 2	Simple	Pairing heap	4-ary
$n = 10^4$	203	62	90	80	68	97
$n = 10^5$	230	54	102	88	78	110
$n = 10^6$	258	68	110	91	101	131

Cascading cuts vs. cascading rank decreases

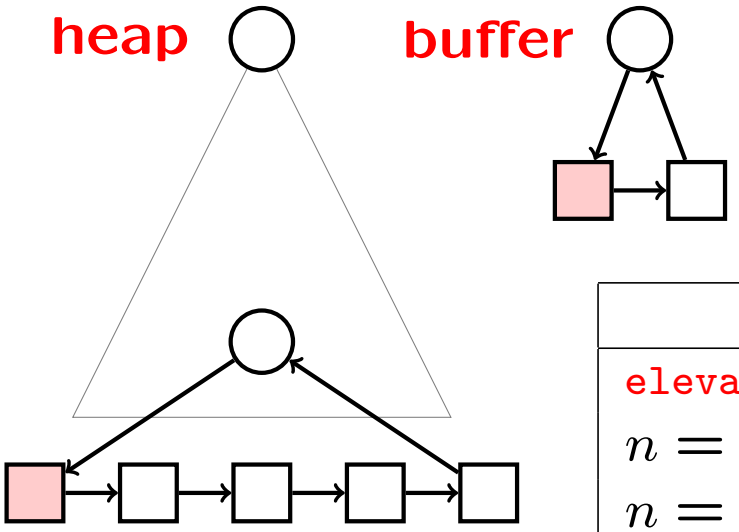
Average running time [ns] per operation [`elevate` random permutation, priority increase n]

	Lazy (cuts)	**Fix 1** (cuts)	**Fix 2** (cuts)	**Lazy** (ranks)	**Fix 2** (ranks)	**Simple** (ranks)	**Simple** (cuts)
`elevate`^{n}							
$n = 10^4$	10	11	10	17	11	46	56
$n = 10^5$	104	108	101	122	108	116	135
$n = 10^6$	162	172	163	210	171	242	287
`extract-top`^{n}							
$n = 10^4$	294	285	270	283	296	242	317
$n = 10^5$	645	624	640	643	621	581	732
$n = 10^6$	1189	1368	1234	1171	1361	1093	1420

What are your conclusions?

Space optimization

Idea: Let each node store a pointer to a chunk of k elements



	Before	After
Space	$S(n)$	$S(n/k)+n+n/k+O(1)$
`top`	$A(n)$	$A(n/k)+O(1)$
`inject`	$B(n)$	$B(n/k)+O(1)$
`elevate`	$C(n)$	$C(n/k)+O(k)$
`eject`	$D(n)$	$D(n/k)+O(1)$
`extract`	$E(n)$	$B(n/k)+E(n/k)+O(k)$
`merge`	$F(m,n)$	$B(n/k)+F(m/k,n/k)+O(k)$

	Lazy	$k=5$	$k=10$	Simple	$k=5$	$k=10$
`elevate`^{n}						
$n=10^4$	10	12	27	11	41	53
$n=10^5$	104	106	145	108	140	183
$n=10^6$	162	226	309	172	306	433
`extract-top`^{n}						
$n=10^4$	294	290	293	242	234	236
$n=10^5$	645	648	680	581	559	604
$n=10^6$	1189	1410	1585	1093	1240	1399

Why is iterator support non-trivial?

Addressable multi-ary heap

```
using node_type = N;
using sequence_type = S;
using locator_type = std::pair<N*, S*>;
```

In `merge`, nodes are moved from from one heap to another. Because the sequence, where a node is in, will change, locators to that node become invalid.

Simple Fibonacci heap

```
using node_type = N;
using locator_type = N*;
```

Here we can use the standard successor function for multi-ary trees to implement `operator++`.

Lazy Fibonacci heap

```
using node_type = N;
using locator_type = N*;
```

To support iterator `operator++`, every node should know how to get to its successor without consulting the involved heap. Because the root list and child lists are circular, some other information should be stored at the nodes.

Concluding remarks

- My biased opinion: Algorithm engineering should be driven by experiments!
- Can you fix elegantly any of the problems encountered?
  - `top` is not strongly exception safe.
  - If `top` is supported at $O(1)$ cost, `elevate` requires 3 comparison €.
  - For both lazy and simple versions, `eject` takes $O(\lg n)$ amortized time.
  - Structures that rely on circular linking do not support memory-less iterators.
- Can you break the $2m + 1.5n \lg n$ element-comparison bound for the `inject-elevate-extract-top` request sequence?
- Ultimate measure of simplicity: The code that is not there!