Algorithm Engineering the Theory

Gerth Stølting Brodal
Aarhus University

ADYN Summer School on Algorithm Engineering for Network Problems, Hasso Plattner Institute, Potsdam, Germany, September 19-22, 2023

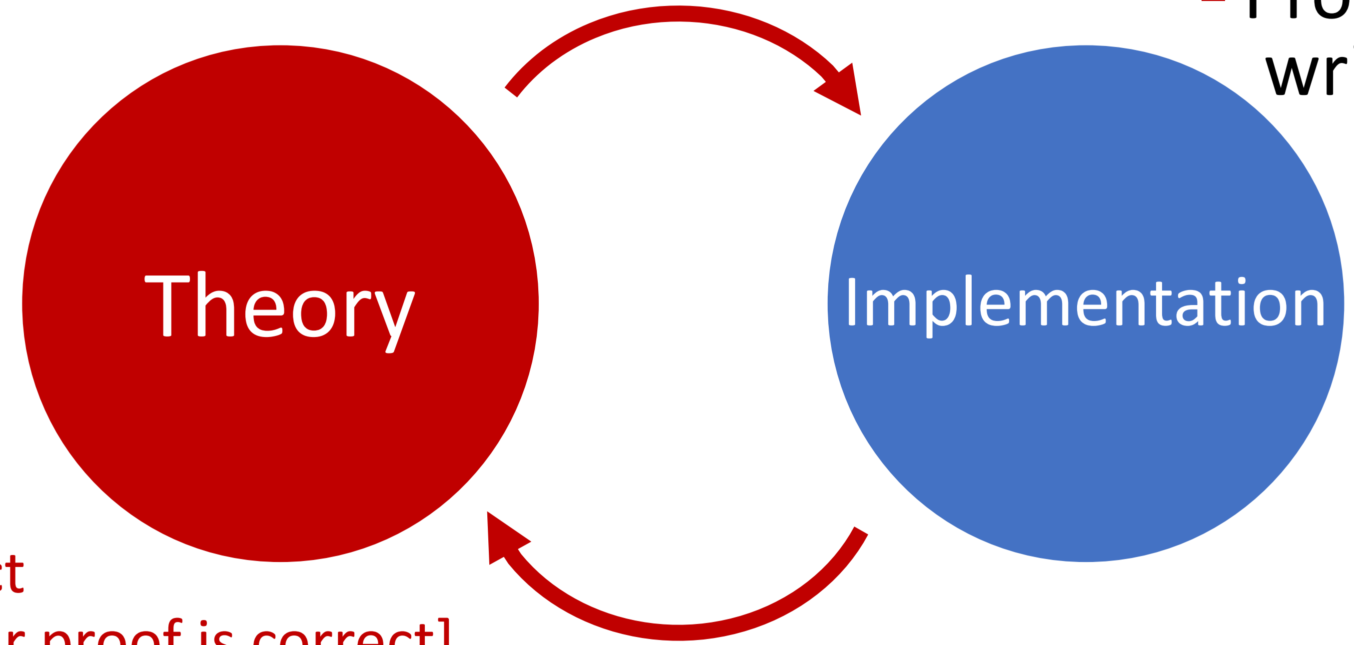
Gerth Stølting Brodal

Research
Data structures 1993 –

Teaching
Algorithms and Data Structures 2002 –
Introduction to Programming (Python) 2018 –
Bachelor project advising

Algorithms

- Creating new theory is cool
- Filling in proof details less exciting
- Uncertainty – have all cases been addressed ? [you prove the algorithm is correct but rarely that your proof is correct]
- Frustrating when errors make their way into published papers



- Coding is healthy
- Coding is fun
- Debugging less so…
- Procrastinating from writing the theory ?
- Document relevance of theory
- Study theory vs real world
- Identify shortcomings of theory
- Inspire new theory

Goal

- Have more celebrations
- Make progress on bugs more frequently
- Not necessarily fewer bugs !

https://www.commitstrip.com/en/2018/05/09/progress/

Certifying algorithms

R.M. McConnell[a], K. Mehlhorn[b,*], S. Näher[c], P. Schweitzer[d]

[a] Computer Science Department, Colorado State University Fort Collins, USA
[b] Max Planck Institute for Informatics and Saarland University, Saarbrücken, Germany
[c] Fachbereich Informatik, Universität Trier, Trier, Germany
[d] College of Engineering and Computer Science, Australian National University, Canberra, Australia

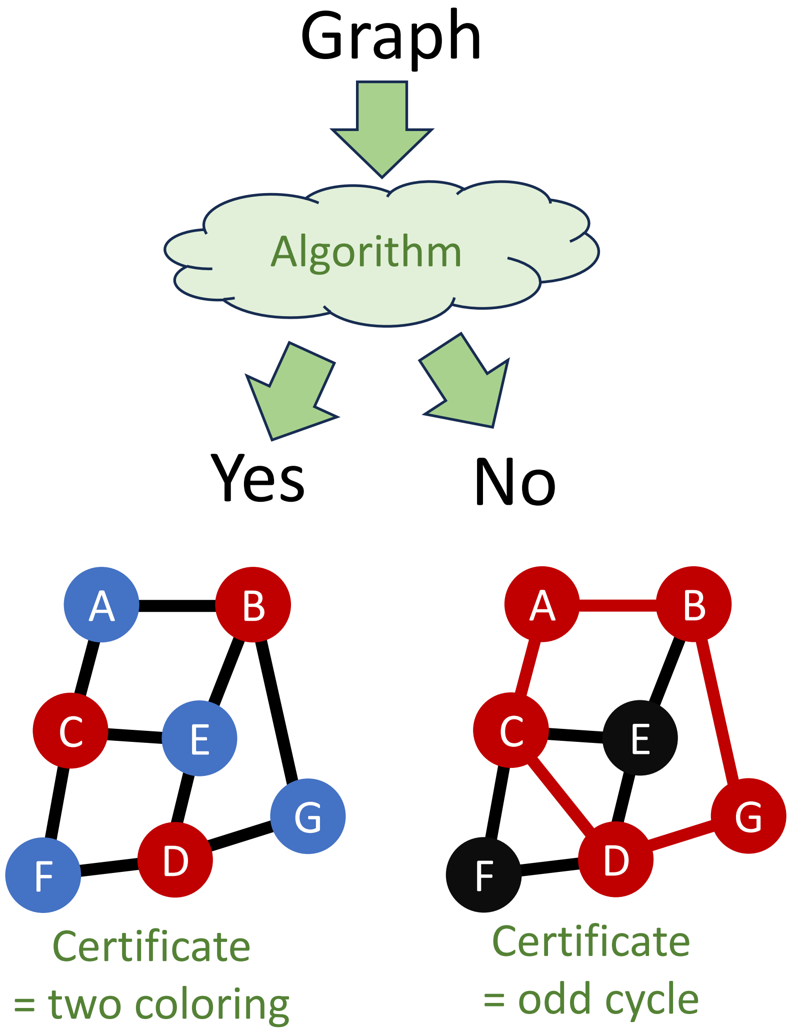
A B S T R A C T

A certifying algorithm is an algorithm that produces, with each output, a certificate or witness (easy-to-verify proof) that the particular output has not been compromised by a bug. A user of a certifying algorithm inputs x, receives the output y and the certificate w, and then checks, either manually or by use of a program, that w proves that y is a correct output for input x. In this way, he/she can be sure of the correctness of the output without having to trust the algorithm.

We put forward the thesis that certifying algorithms are much superior to non-certifying algorithms, and that for complex algorithmic tasks, only certifying algorithms are satisfactory. Acceptance of this thesis would lead to a change of how algorithms are taught and how algorithms are researched. The widespread use of certifying algorithms would greatly enhance the reliability of algorithmic software.

We survey the state of the art in certifying algorithms and add to it. In particular, we start a theory of certifying algorithms and prove that the concept is universal.

Example : Bipartite graph ?



Computer Science Review 5(2), 119-161, 2011
DOI: doi.org/10.1016/j.cosrev.2010.09.009

Automatic testing of algorithm implementation

```
while True:
    x = generate_random_input()
    answer, certificate = algorithm(x)
    assert verify(x, answer, certificate)
    print('.')
```

- Happy when sequence of dots grow ....
- Program crashes – debugger can perhaps help you find the bug
- Verification fails – a bug somewhere in the program/algorithm

Simplifying failed input (Greedy DFS)

```
def simplify_bug(failed_input):
    for x in simplifications(failed_input):
        try:
            algorithm(x)
        except Bug:
            print('YES! - failed on', x)
            return simplify_bug(x)
    return failed_input
```

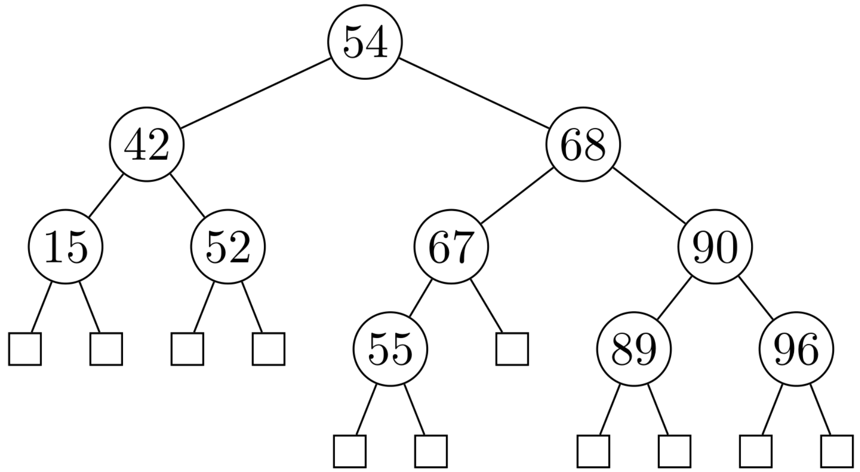
- `simplifications` could e.g. report all ways of removing a single vertex or edge from a graph
- Complex input triggering the bug can sometimes be simplified to an input of manageable size

Invariants

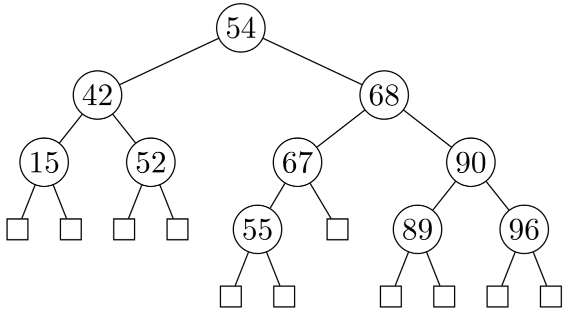
- **Invariants** are a fundamental tool when *designing* and *analyzing* algorithms and data structures

- Capture state of algorithm

- Example: AVL tree invariant
  1) Search tree
  2) $\forall\, v : |v.\text{left.height} - v.\text{right.height}\,| \leq 1$



- Invariants can be made **assertions** in code $\Rightarrow$ ensure code integrity




```
def validate(tree, min_value=None, max_value=None):
    '''Validate AVL-tree invariants.'''

    if not is_empty(tree):
        assert min_value == None or min_value <= tree.root
        assert max_value == None or tree.root <= max_value
        assert tree.height == 1 + max(tree.left.height, tree.right.height)
        assert abs(tree.left.height - tree.right.height) <= 1
        validate(tree.left, min_value, tree.root)
        validate(tree.right, tree.root, max_value)

def inorder(tree):
    '''Generator that yields values in tree in sorted order.'''

    if not is_empty(tree):
        yield from inorder(tree.left)
        yield tree.root
        yield from inorder(tree.right)

def test_insertions(n):
    data = random.choices(range(10 * n), k=n)
    tree = empty_tree
    for i, x in enumerate(data):
        tree = insert(tree, x)
        validate(tree)
        assert sorted(data[:i + 1]) == list(inorder(tree))
```

- Write validate and test methods *before* implementing insert

“Test driven algorithm design”

- Formulate **invariants** as the driving tool for algorithm design
- Implement invariants in code as **assertions** and **verifier methods**
- Automate **stress tests**
- Develop algorithm through **failed tests**

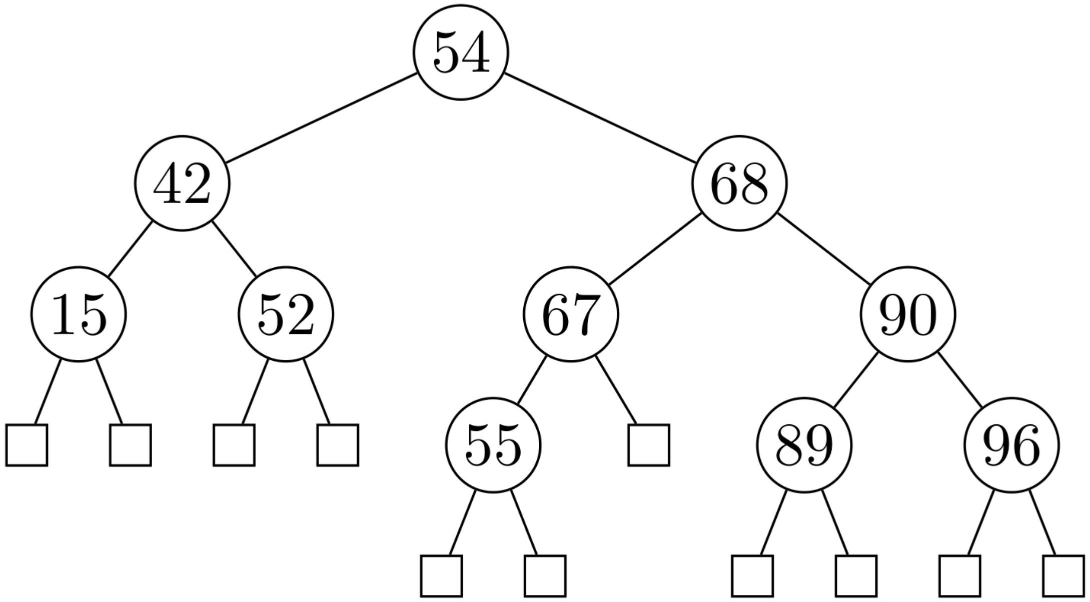
  $\Rightarrow$ likely good coverage of special cases

  $\Rightarrow$ little redundant code

- Note: *verification methods might slow down code significantly (asymptotic slower!), but the focus is on developing correct theory*

Visual test/debugging of autogenerated figure

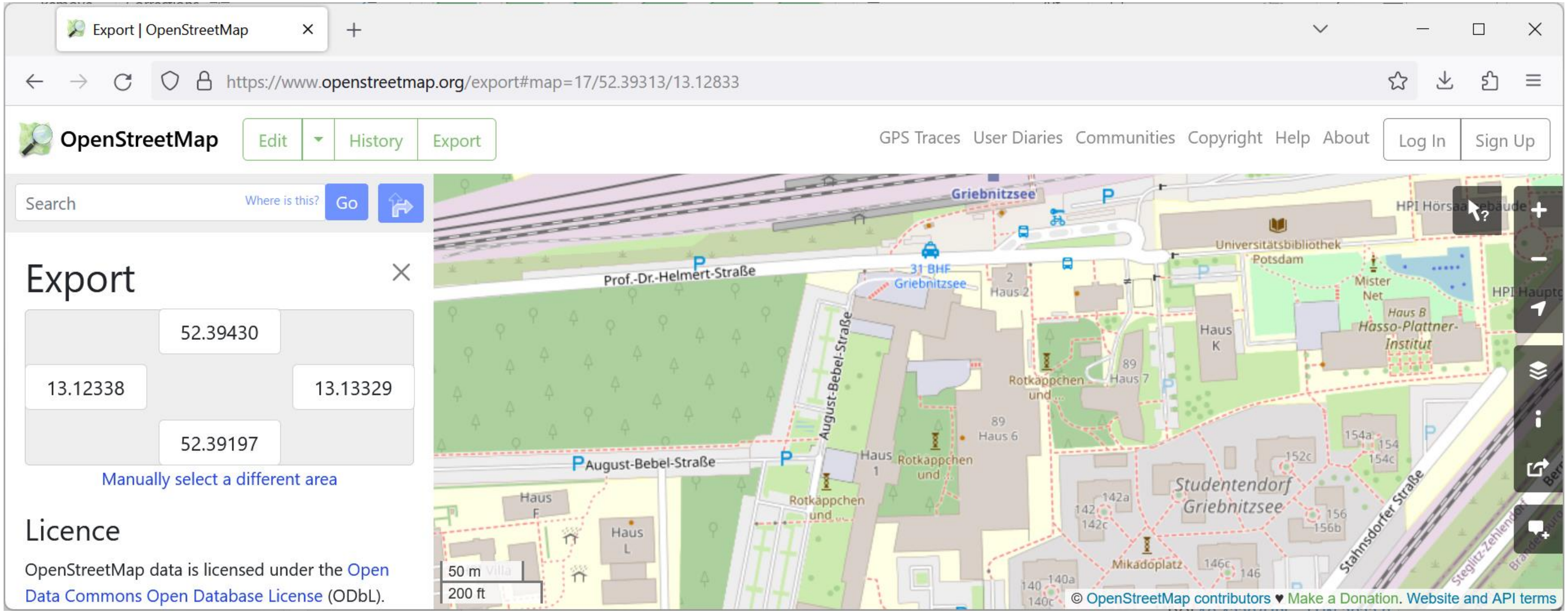
```python
def tikz(tree):
    def recurse(tree):
        if tree is empty_tree:
            return '{}'
        else:
            return f'[.{tree.root} {recurse(tree.left)} {recurse(tree.right)} ]'
    return r'\Tree ' + recurse(tree)
```




```
\Tree [.54 [.42 [.15 {} {} ] [.52 {} {} ] ] [.68 [.67 [.55 {} {} ] {} ] [.90 [.89 {} {} ] [.96 {} {} ] ] ] ]
```

An unexpected journey

- Bachelor project = shortest paths on Open Street Map graphs
- Students have trouble implementing Dijkstra's algorithm in Java™

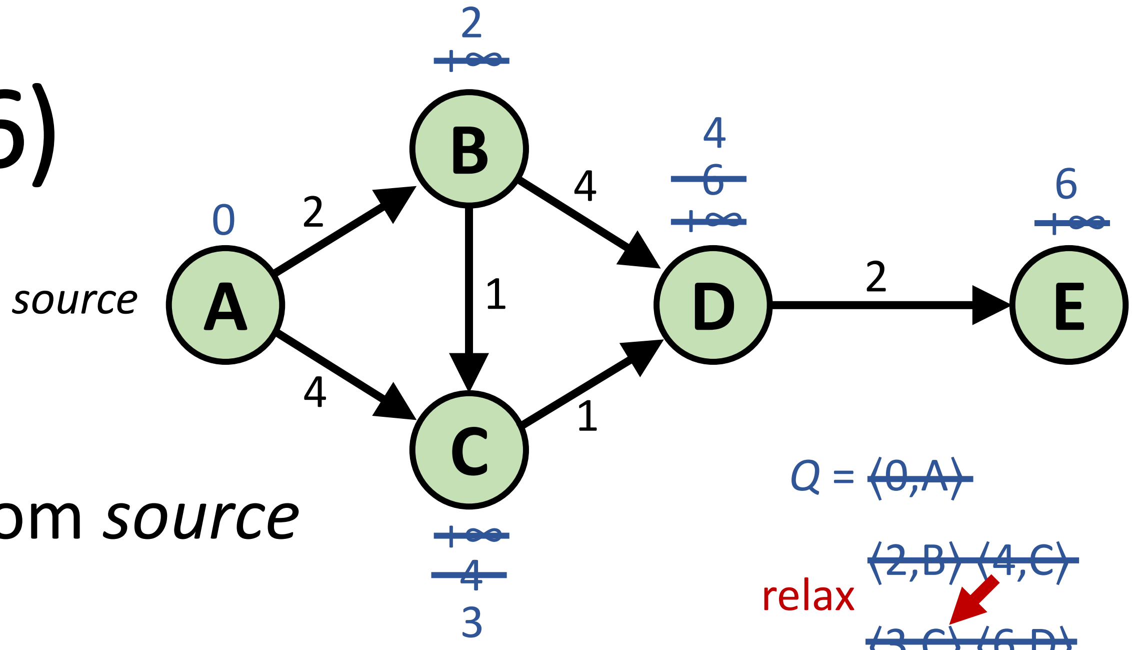


DOI 10.4230/LIPIcs.FUN.2022.8

```
 <way id="79388407" visible="true" version="17" changeset="107546769" timestamp="2021-07-07T08:48:29Z" user="KartoffelOS"
uid="10758523">
  <nd ref="296937646"/>
  <nd ref="926885043"/>
  <nd ref="926884234"/>
  <nd ref="4868434116"/>
  <nd ref="528571257"/>
  <tag k="access" v="private"/>
  <tag k="bicycle" v="yes"/>
  <tag k="delivery" v="yes"/>
  <tag k="emergency" v="yes"/>
  <tag k="foot" v="yes"/>
  <tag k="highway" v="service"/>
  <tag k="lit" v="yes"/>
  <tag k="maxspeed" v="20"/>
  <tag k="name" v="August-Bebel-Straße"/>
  <tag k="postal_code" v="14482"/>
  <tag k="service" v="parking_aisle"/>
  <tag k="surface" v="paving_stones"/>
 </way>
 <way id="970133467" visible="true" version="7" changeset="135350751" timestamp="2023-04-25T16:12:30Z" user="tecmap15"
uid="4798255">
  <nd ref="8977535608"/>
  <nd ref="8977535605"/>
  <nd ref="8977535606"/>
  <nd ref="8977535607"/>
  <nd ref="8977535601"/>
  <nd ref="8977535602"/>
  <nd ref="8977535608"/>
  <tag k="addr:city" v="Potsdam"/>
  <tag k="addr:country" v="DE"/>
  <tag k="addr:housenumber" v="88"/>
  <tag k="addr:postcode" v="14482"/>
  <tag k="addr:street" v="August-Bebel-Straße"/>
  <tag k="addr:suburb" v="Babelsberg"/>
  <tag k="building" v="university"/>
  <tag k="building:levels" v="3"/>
  <tag k="name" v="Haus L"/>
  <tag k="roof:levels" v="0"/>
  <tag k="roof:shape" v="flat"/>
  <tag k="wheelchair" v="yes"/>
 </way>
```

Dijkstra's algorithm (1956)

- Non-negative edge weights
- Visits nodes in increasing distance from *source*



Q = ~~⟨0,A⟩~~

~~⟨2,B⟩~~ ~~⟨4,C⟩~~

relax

~~⟨3,C⟩~~ ~~⟨6,D⟩~~

~~⟨4,D⟩~~

~~⟨6,E⟩~~

proc $\texttt{Dijkstra}_1(V, E, \delta, s)$
  $dist[v] = +\infty$ for all $v \in V \setminus \{s\}$
  $dist[s] = 0$
  $\texttt{Insert}(Q, \langle dist[s], s\rangle)$
  while $Q \neq \emptyset$ **do**
    $\langle d, u\rangle = \texttt{ExtractMin}(Q)$
    for $(u, v) \in E \cap (\{u\} \times V)$ **do**
      if $dist[u] + \delta(u, v) < dist[v]$ **then**
        $dist[v] = dist[u] + \delta(u, v)$
        if $v \in Q$ **then**
          $\texttt{DecreaseKey}(Q, v, dist[v])$
        else
          $\texttt{Insert}(Q, \langle v, dist[v]\rangle)$
  return $dist$

relax

Fibonacci heaps
(Fredman, Tarjan 1984)
$\Rightarrow O(m + n \cdot \log n)$

proc $\texttt{Dijkstra}_2(V, E, \delta, s)$
  $dist[v] = +\infty$ for all $v \in V \setminus \{s\}$
  $dist[s] = 0$
  $\texttt{Insert}(Q, \langle dist[s], s\rangle)$
  while $Q \neq \emptyset$ **do**
    $\langle d, u\rangle = \texttt{ExtractMin}(Q)$
    for $(u, v) \in E \cap (\{u\} \times V)$ **do**
      if $dist[u] + \delta(u, v) < dist[v]$ **then**
        $dist[v] = dist[u] + \delta(u, v)$
        if $v \in Q$ **then**
          $\texttt{Remove}(Q, v)$
        $\texttt{Insert}(Q, \langle dist[v], v\rangle)$
  return $dist$

$O(\log n)$ Remove
$\Rightarrow O(m \cdot \log n)$

The challenge - Java's builtin binary heap

- no `decreasekey`
- `remove` O(n) time
  - $\Rightarrow$ Dijkstra O($m \cdot n$)
- comparator function

Java SE 18 & JDK 18

SEARCH: Search

Implementation note: this implementation provides O(log(n)) time for the enqueuing and dequeuing methods (`offer`, `poll`, `remove()` and `add`); linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek`, `element`, and `size`).

This class is a member of the Java Collections Framework.

Since:

1.5

Java SE 18 & JDK 18

SEARCH: Search

`PriorityQueue(int initialCapacity)`	Creates a `PriorityQueue` with the specified initial capacity that orders its elements according to their natural ordering.
`PriorityQueue(int initialCapacity, Comparator<? super E> comparator)`	Creates a `PriorityQueue` with the specified initial capacity that orders its elements according to the specified comparator.

Repeated insertions

- Relax inserts new copies of item
- Skip outdated items

$\textbf{proc}\ \texttt{Dijkstra}_3(V, E, \delta, s)$

$\quad \mathit{dist}[v] = +\infty$ for all $v \in V \setminus \{s\}$

$\quad \mathit{dist}[s] = 0$

$\quad \texttt{Insert}(Q, \langle \mathit{dist}[s], s\rangle)$

$\quad \textbf{while}\ Q \neq \emptyset\ \textbf{do}$

$\quad\quad \langle d, u\rangle = \texttt{ExtractMin}(Q)$

outdated ? → $\quad\quad \textbf{if}\ d = \mathit{dist}[u]\ \textbf{then}$

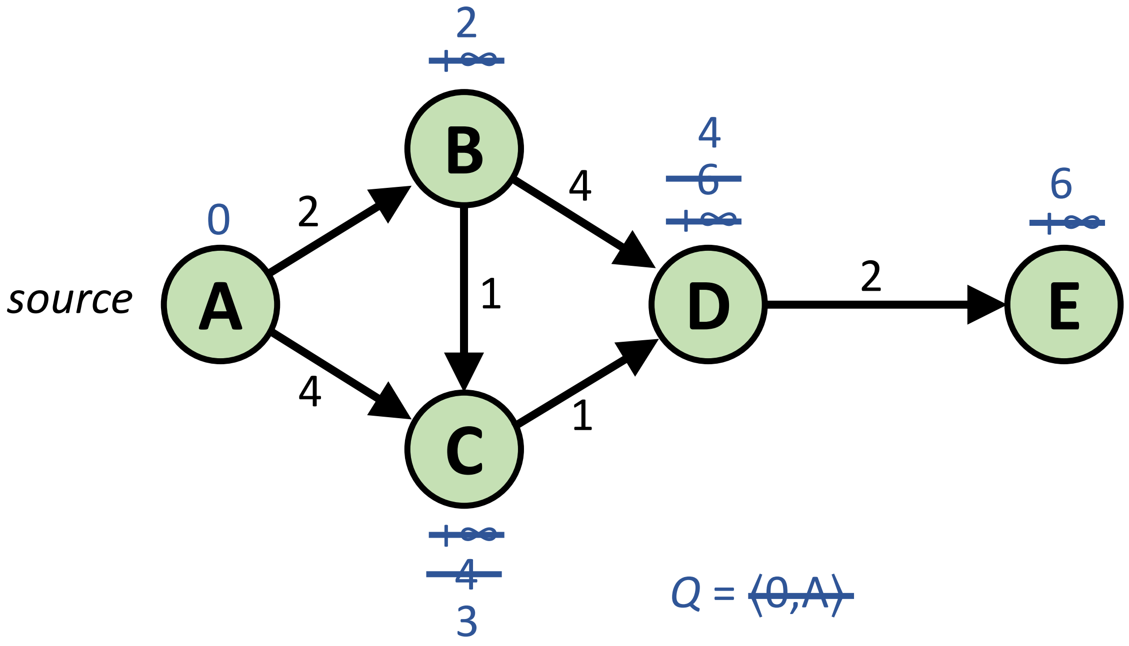
$\quad\quad\quad \textbf{for}\ (u, v) \in E \cap (\{u\} \times V)\ \textbf{do}$

$\quad\quad\quad\quad \textbf{if}\ \mathit{dist}[u] + \delta(u, v) < \mathit{dist}[v]\ \textbf{then}$

$\quad\quad\quad\quad\quad \mathit{dist}[v] = \mathit{dist}[u] + \delta(u, v)$

relax = reinsert → $\quad\quad\quad\quad\quad \texttt{Insert}(Q, \langle \mathit{dist}[v], v\rangle)$

$\quad \textbf{return}\ \mathit{dist}$



Q = ~~⟨0,A⟩~~

~~⟨2,B⟩ ⟨4,C⟩~~

~~⟨3,C⟩ ⟨4,C⟩ ⟨6,D⟩~~

~~⟨4,C⟩ ⟨4,D⟩ ⟨6,D⟩~~

~~⟨4,D⟩ ⟨6,D⟩~~

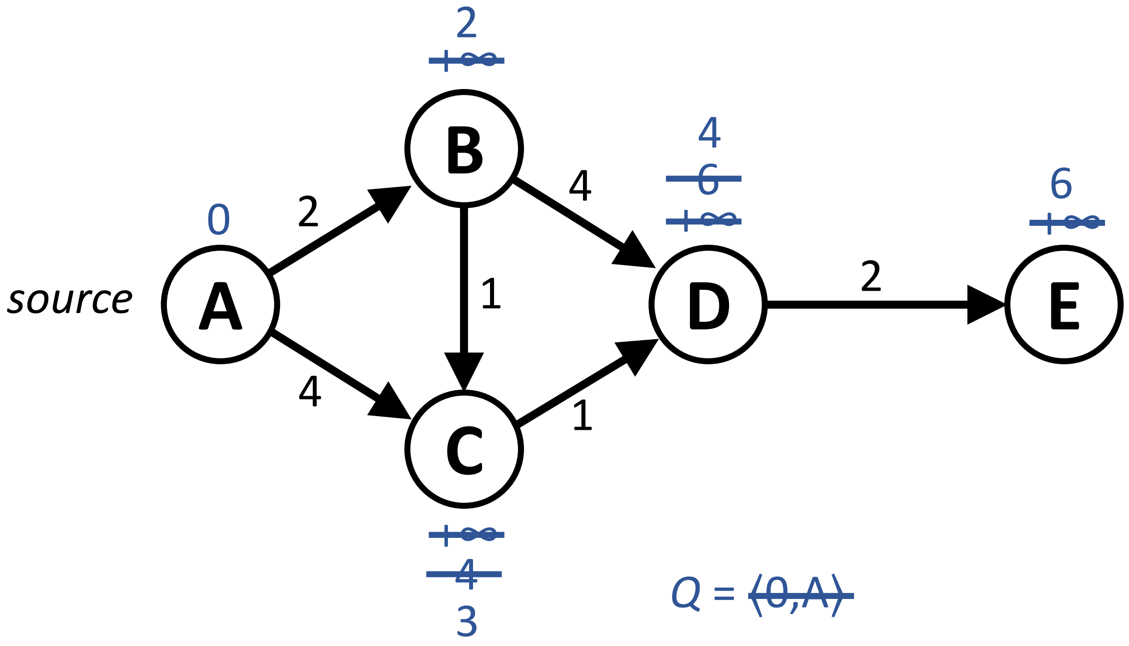
~~⟨6,D⟩ ⟨6,E⟩~~

~~⟨6,E⟩~~

Using a visited set

$$
\begin{array}{l}
\textbf{proc}\ \texttt{Dijkstra}_4(V, E, \delta, s) \\
\quad \mathit{dist}[v] = +\infty \text{ for all } v \in V \setminus \{s\} \\
\quad \mathit{dist}[s] = 0 \\
\quad \mathit{visited} = \emptyset \\
\quad \texttt{Insert}(Q, \langle \mathit{dist}[s], s \rangle) \\
\quad \textbf{while}\ Q \neq \emptyset\ \textbf{do} \\
\quad\quad \langle d, u \rangle = \texttt{ExtractMin}(Q) \\
\quad\quad \textbf{if}\ u \notin \mathit{visited}\ \textbf{then} \\
\quad\quad\quad \mathit{visited} = \mathit{visited} \cup \{u\} \\
\quad\quad\quad \textbf{for}\ (u, v) \in E \cap (\{u\} \times V)\ \textbf{do} \\
\quad\quad\quad\quad \textbf{if}\ \mathit{dist}[u] + \delta(u, v) < \mathit{dist}[v]\ \textbf{then} \\
\quad\quad\quad\quad\quad \mathit{dist}[v] = \mathit{dist}[u] + \delta(u, v) \\
\quad\quad\quad\quad\quad \texttt{Insert}(Q, \langle \mathit{dist}[v], v \rangle) \\
\quad \textbf{return}\ \mathit{dist}
\end{array}
$$

bitvector →



Q = ~~⟨0,A⟩~~

~~⟨2,B⟩ ⟨4,C⟩~~

~~⟨3,C⟩ ⟨4,C⟩ ⟨6,D⟩~~

~~⟨4,C⟩ ⟨4,D⟩ ⟨6,D⟩~~

~~⟨4,D⟩ ⟨6,D⟩~~

~~⟨6,D⟩ ⟨6,E⟩~~

~~⟨6,E⟩~~

A shaky idea...

proc $\texttt{Dijkstra}_4(V, E, \delta, s)$

$\quad dist[v] = +\infty$ for all $v \in V \setminus \{s\}$

$\quad dist[s] = 0$

$\quad visited = \emptyset$

$\quad \texttt{Insert}(Q, \langle dist[s], s\rangle)$

$\quad$ **while** $Q \neq \emptyset$ **do**

d never used ⟶ $\quad\quad \langle \cancel{d}, u\rangle = \texttt{ExtractMin}(Q)$

$\quad\quad$ **if** $u \notin visited$ **then**

$\quad\quad\quad visited = visited \cup \{u\}$

$\quad\quad\quad$ **for** $(u, v) \in E \cap (\{u\} \times V)$ **do**

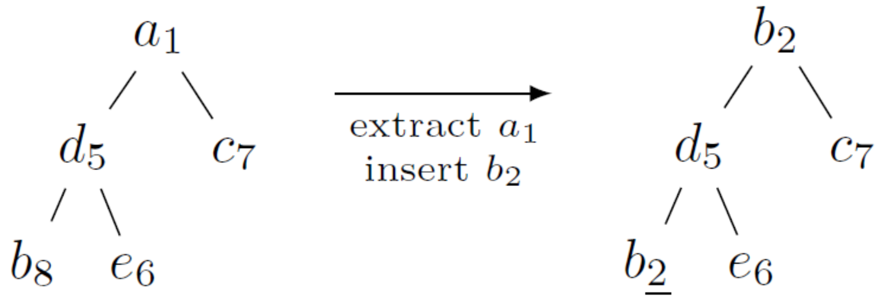
$\quad\quad\quad\quad$ **if** $dist[u] + \delta(u, v) < dist[v]$ **then**

$\quad\quad\quad\quad\quad dist[v] = dist[u] + \delta(u, v)$

$\quad\quad\quad\quad\quad \texttt{Insert}(Q, \langle \cancel{dist}[v], v\rangle)$

$\quad$ **return** $dist$

- Q only store nodes (save space)
- Comparator
- Key = current distance *dist*



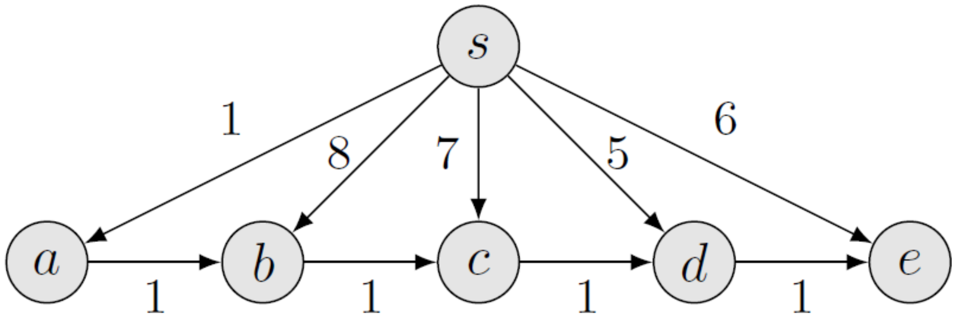
Heap invariants break

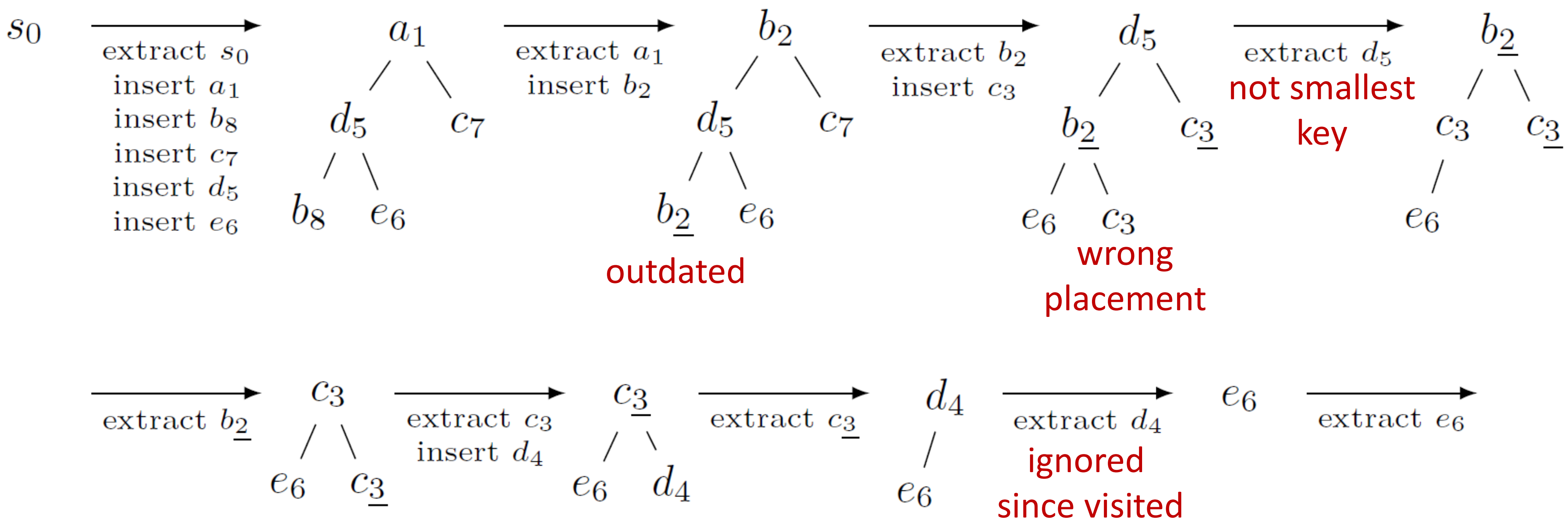
Experimental study

```
visited = set()
Q = Queue()
Q.insert(Item(0, source))
while not Q.empty():
    u = Q.extract_min().value
    if u not in visited:
        visited.add(u)
        for v in G.out[u]:
            dist_v = dist[u] + G.weights[(u, v)]
            if dist_v < dist[v]:
                dist[v] = dist_v
                parent[v] = u
                Q.insert(Item(dist[v], v))
```

- Implemented Dijkstra$_4$ in Python
- Stress test on random cliques
- Binary heaps failed (default priority queue in Java and Python)

Binary heaps using *dist* in a comparator fails





Experimental study

```
visited = set()
Q = Queue()
Q.insert(Item(0, source))
while not Q.empty():
    u = Q.extract_min().value
    if u not in visited:
        visited.add(u)
        for v in G.out[u]:
            dist_v = dist[u] + G.weights[(u, v)]
            if dist_v < dist[v]:
                dist[v] = dist_v
                parent[v] = u
                Q.insert(Item(dist[v], v))
```

- Implemented Dijkstra$_4$ in Python
- Stress test on random cliques
- Binary heaps failed (default priority queue in Java and Python)

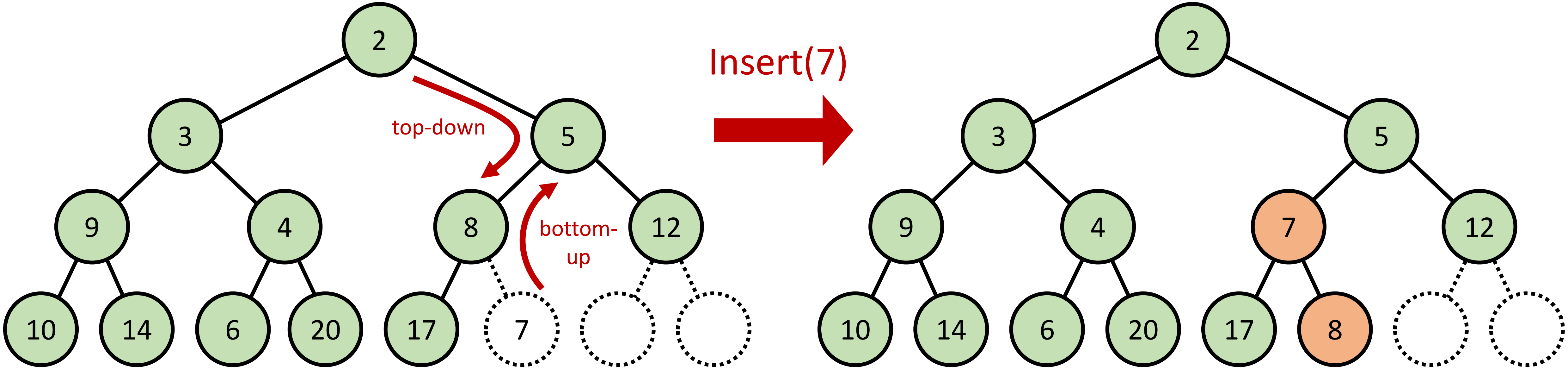
unexpected

- Skew heaps worked
- Leftist heaps worked
- Pairing heaps worked
- Binomial queues worked
- Post-order heaps worked
- Binary heaps with top-down insertions worked

Pointer based

Implicit (space efficient)

Binary heap insertions – bottom-up vs top-down



Definition: Priority queues with decreasing keys

- Items = ⟨key, value⟩
- Over time keys can decrease – *priority queue is not informed*
- Items are compared w.r.t. their current keys
- The original key of an item is the key when it was inserted

`Insert(`item`)`

`ExtractMin()` returns an item with current key less than or equal to all original keys in the priority queue

Theorem 1

`Dijkstra`$_4$ correctly computes shortest paths when using *dist* as current key and a priority queue supporting *decreasing keys*

Theorem 2

The following priority queues support *decreasing keys* (out of the box)

- binary heaps with top-down insertions
- skew heaps
- leftist heaps
- pairing heaps
- binomial queues
- post-order heaps

Proof of Theorem 2 - Basic idea

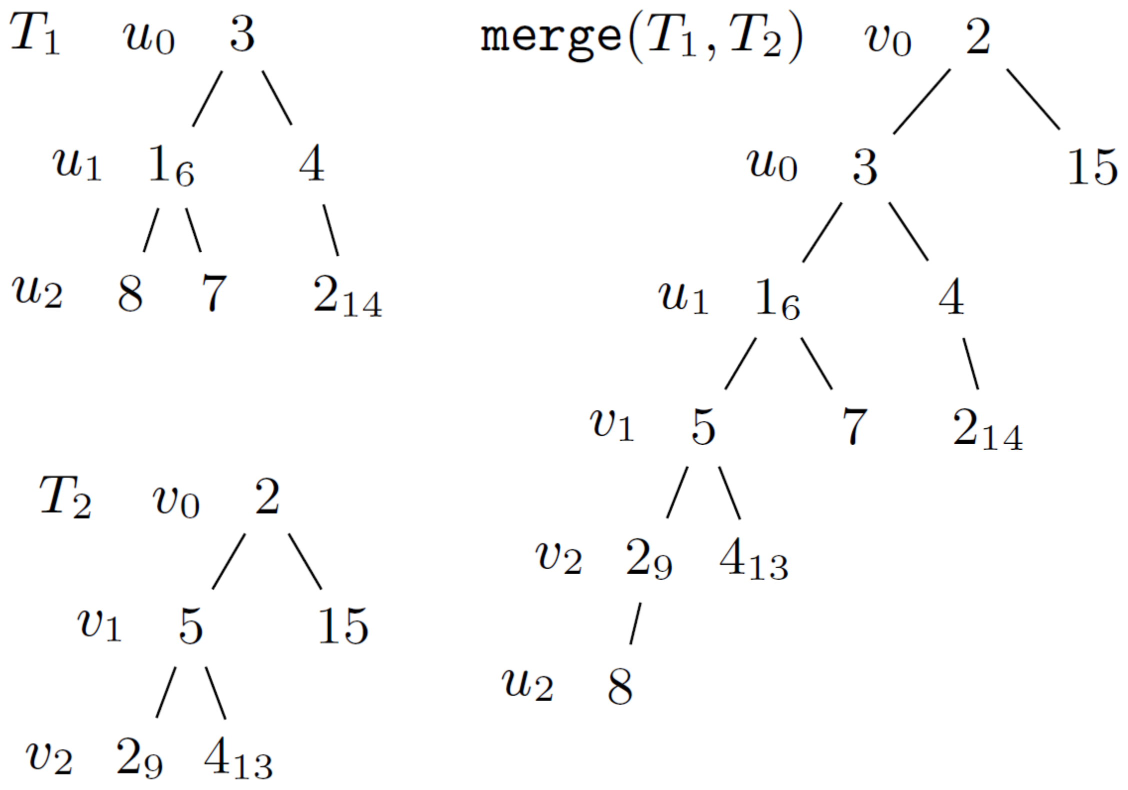
- Decreased heap order

  u ancestor of *v* ⇒
  current key *u* ≤ original key *v*

- Root valid item to extract

- Top-down merging two paths preserves decreased heap order

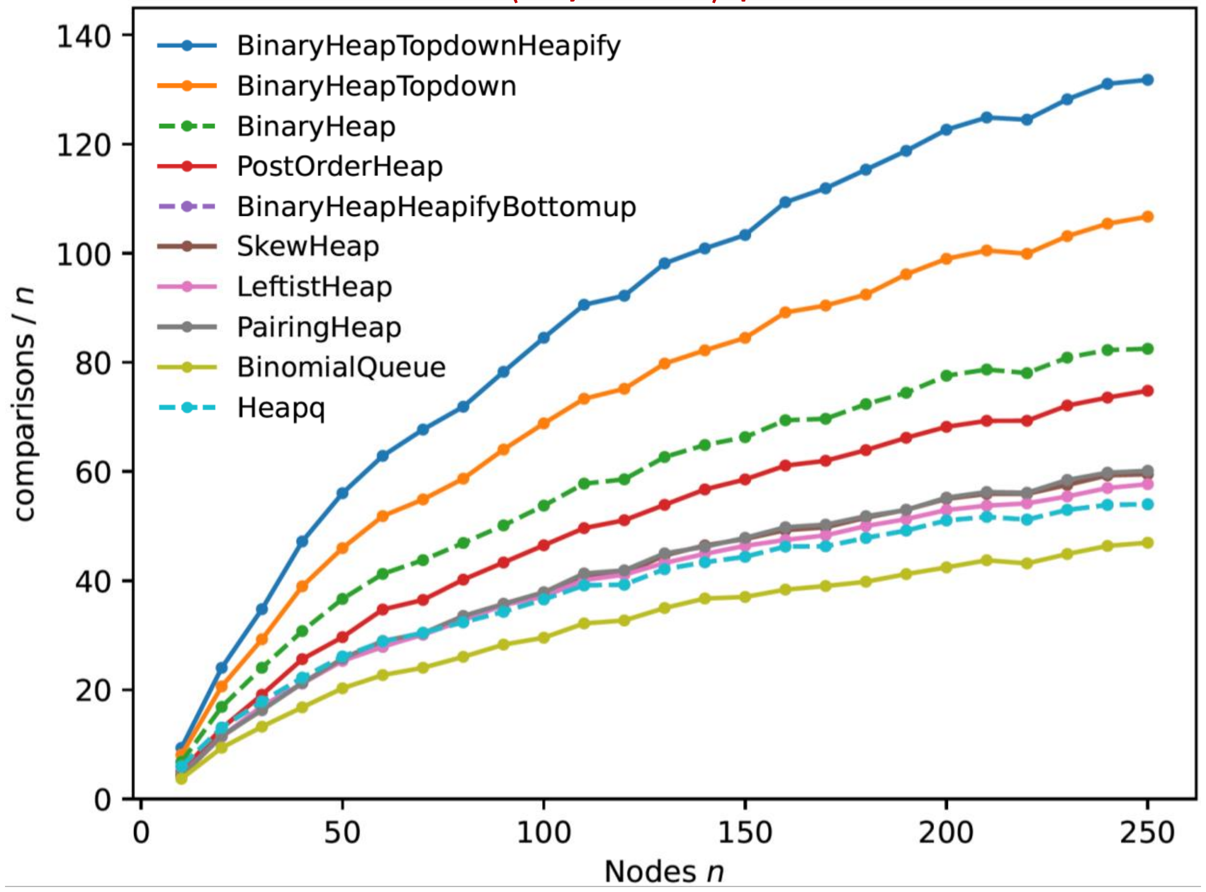
  ⇒ skew heaps and leftist heaps support decreasing keys



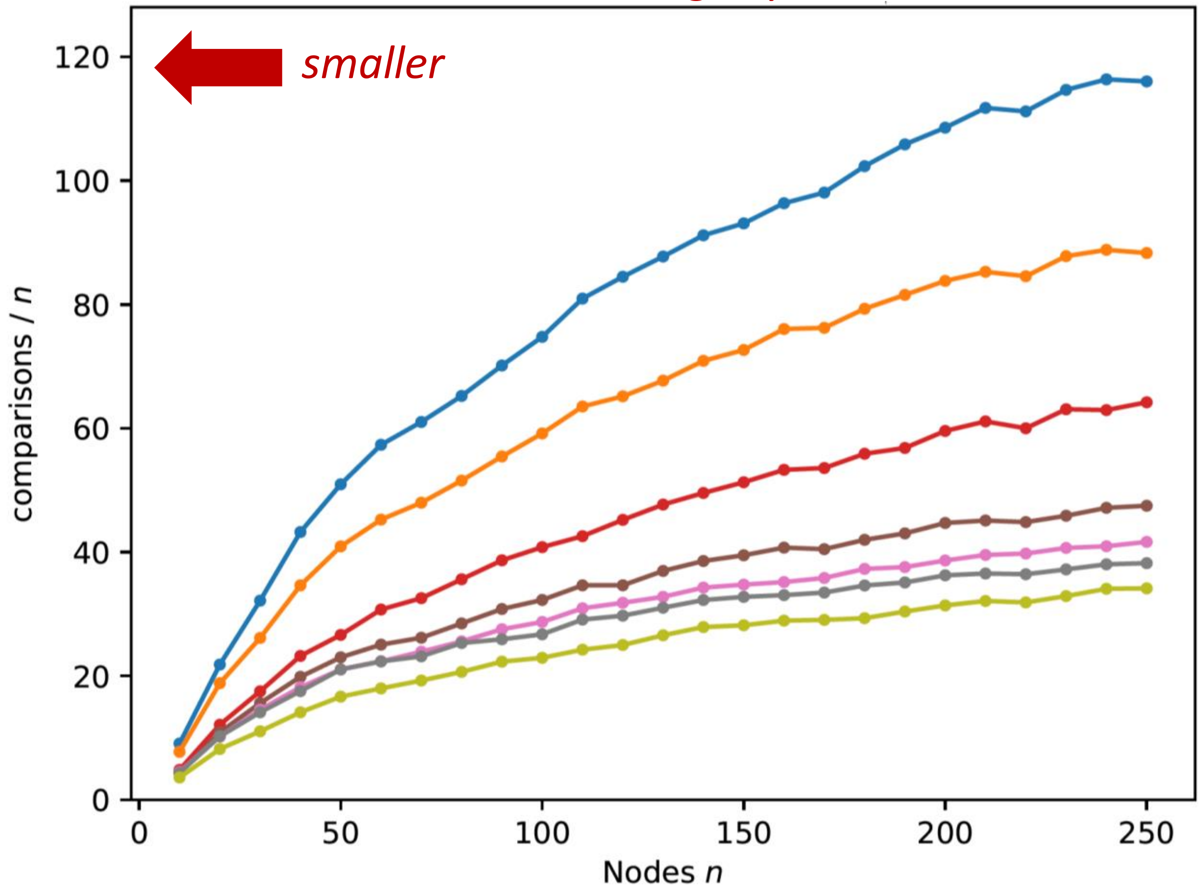
Experimental evaluation of various heaps

- Cliques with uniform random weights
- With decreasing keys less comparisons (outdated items removed earlier)

⟨key, value⟩ pairs

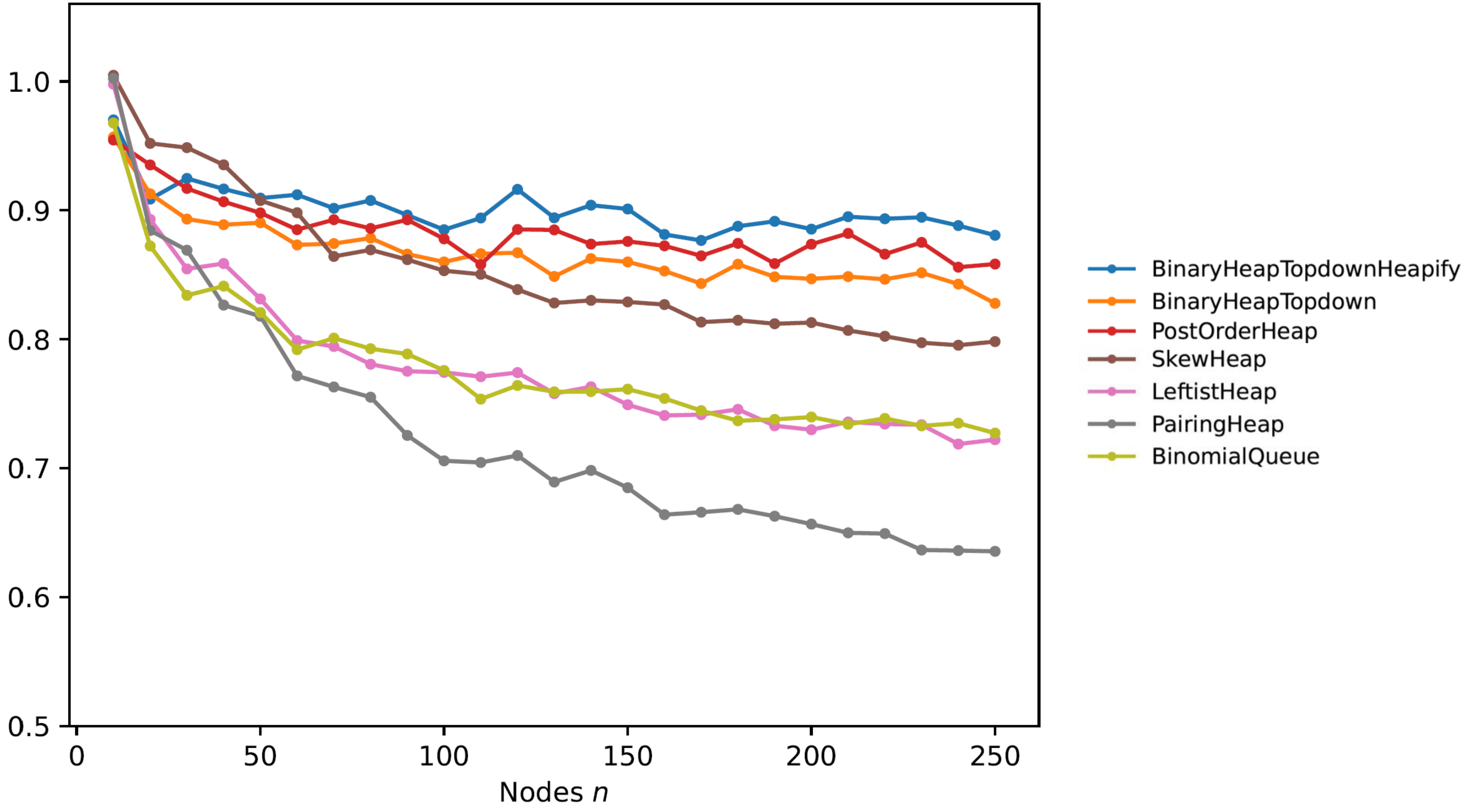


decreasing keys

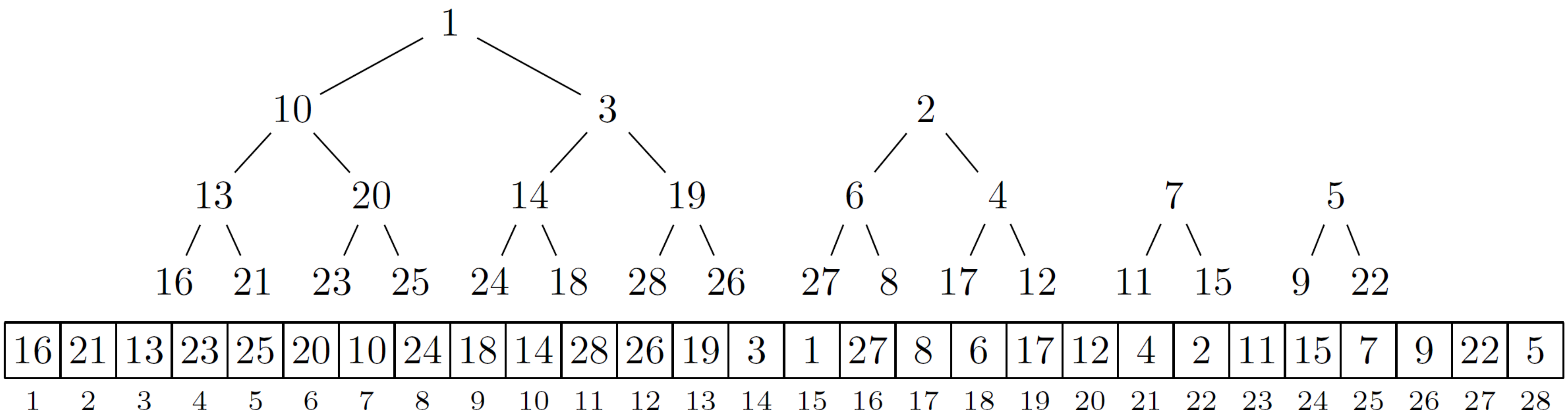


Reduction in comparisons

comparisons decreasing keys / comparisons ⟨key, value⟩ pairs



Postorder heap [Harvey and Zatloukal, FUN 2004]

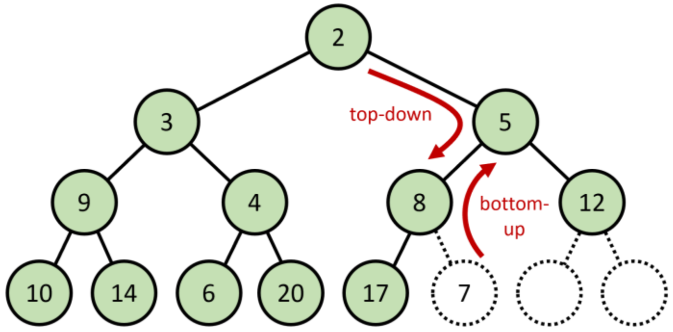


- `Insert` amortized O(1), `ExtractMin` amortized O(log *n*)
- Implicit (space efficient)
- Best implicit comparison performance (and good time performance)

Summary of the unexpected journey

- Introduced notion of priority queues with decreasing keys
  ... as an approach to deal with outdated items in Dijkstra's algorithm
- Experiments identified priority queues supporting decreasing keys
  ... just had to prove it
- Builtin priority queues in Java and Python are binary heaps
  ... do not support decreasing keys
- Binary heaps with top-down insertions do support decreasing keys
  ... and also

skew heaps, leftist heaps, pairing heaps,
binomial queues, post-order heaps



The reviewer is always right

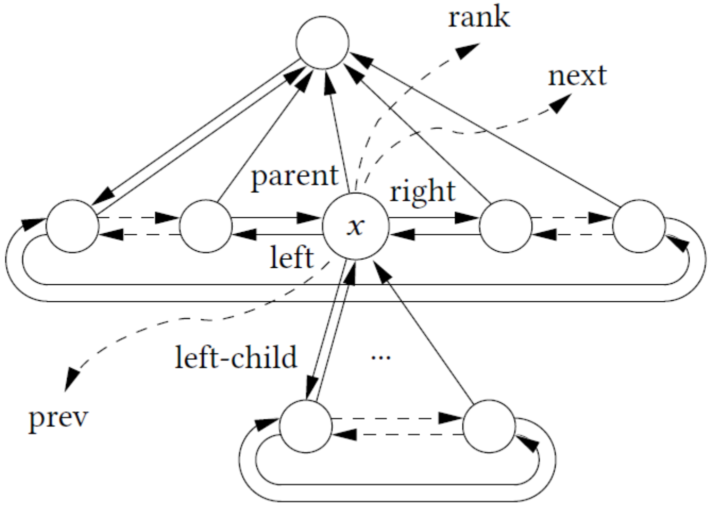
”If there was a implementation where the authors verified that everything did what it was supposed to, I would be more confident that things were correct (I am not talking about a practical implementation, I am talking about one to make sure all invariants hold).”

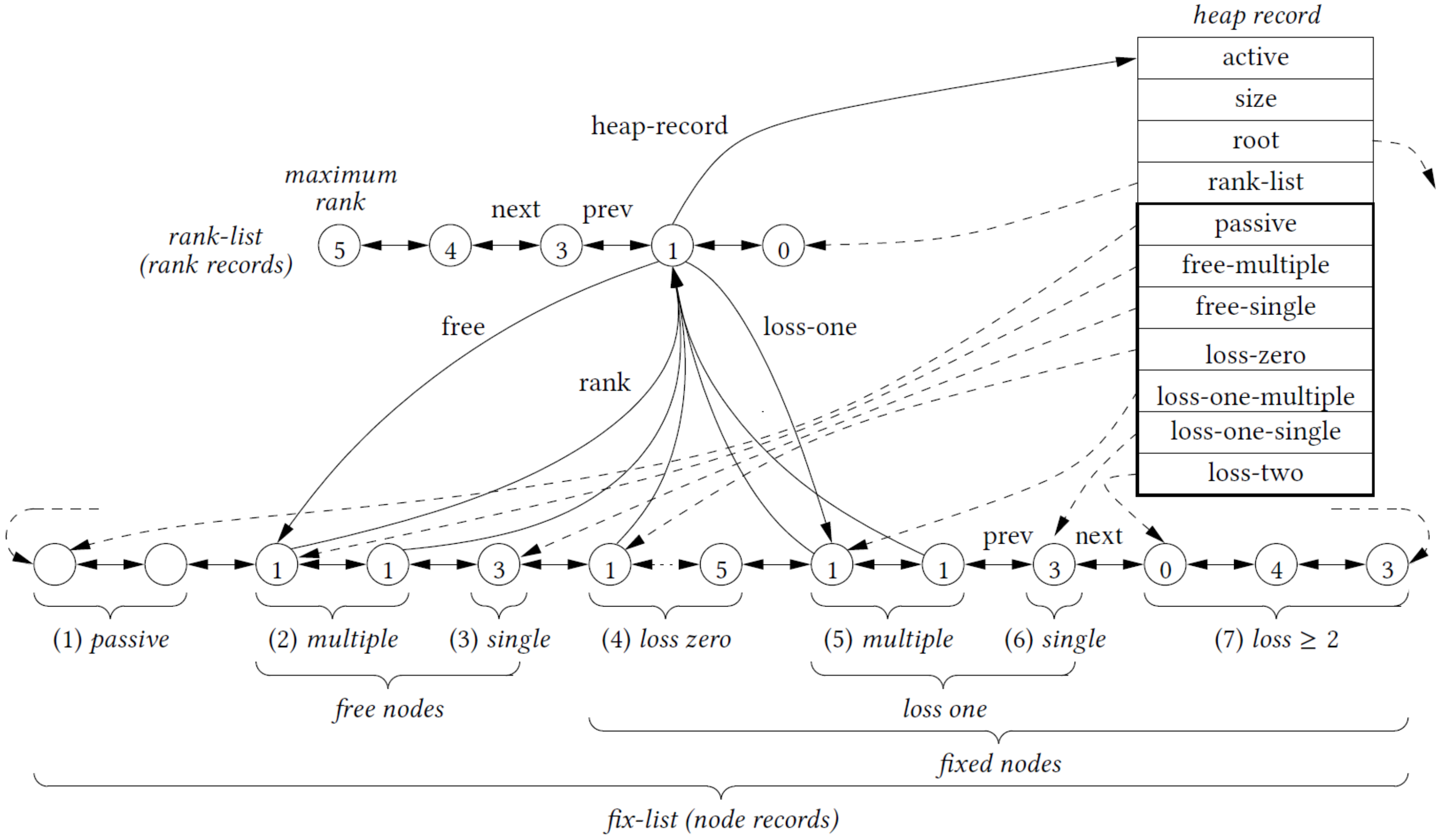
Anonymous reviewer

Strict Fibonacci heaps

	Binary heap **[Williams 1964]** **worst-case**	**Fibonacci heap** **[Fredman, Tarjan 1984]** **amortized**	**Strict Fibonacci heap** **[B., Lagogiannis, Tarjan 2012]** **worst-case**
Insert	O(log n)	O(1)	O(1)
ExtractMin	O(log n)	O(log n)	O(log n)
DecreaseKey	O(log n)	O(1)	O(1)
Meld	-	O(1)	O(1)

Strict Fibonnacci heaps

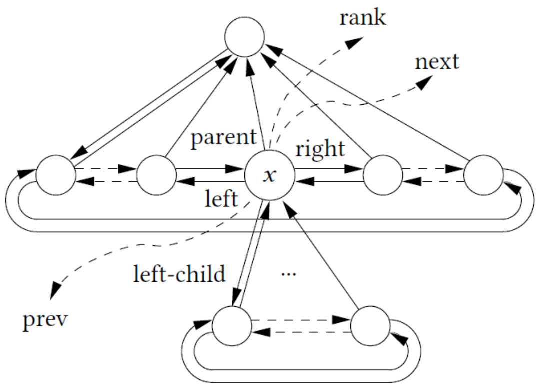


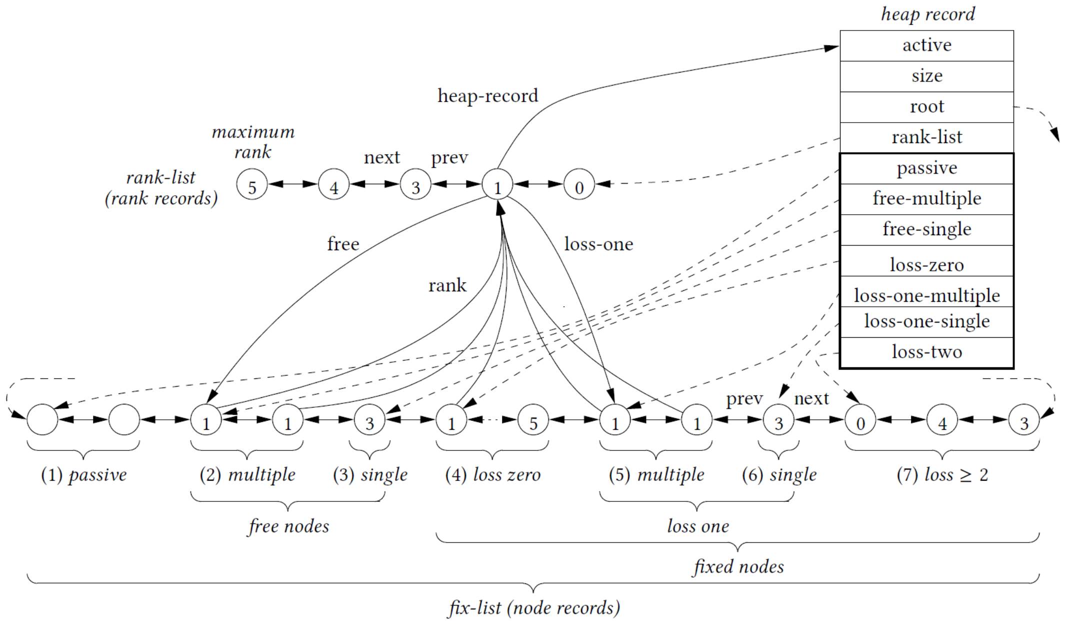


+ many structural invariants

Python implementation

- 1589 lines
- 215 assert statements
- All claimed invariants turned into assert statements
- Validation methods to traverse full structure to verify all claimed invariants
- Stress test using random inputs
- Supported the theory





www.cs.au.dk/~gerth/strict_fibonacci_heaps.py

Code coverage

- Used the Python module coverage
- Some code rarely executed
- Repeat random test 1.000.000 times
- Most code executed at least once

- Realized there was code for cases which provably never can occur
- Implementation → new invariants discovered

coverage.readthedocs.io

Code coverage

odd_even.py

```python
 1 def f(x):
 2     if x % 2 == 0:
 3         return 'even'
 4     elif x % 4 == 0:
 5         return 'even more even'
 6     elif x % 2 == 1:
 7         return 'odd'
 8 import random
 9 for i in range(10):
10     x = random.randint(0, 10)
11     print(x, f(x))
```

Shell

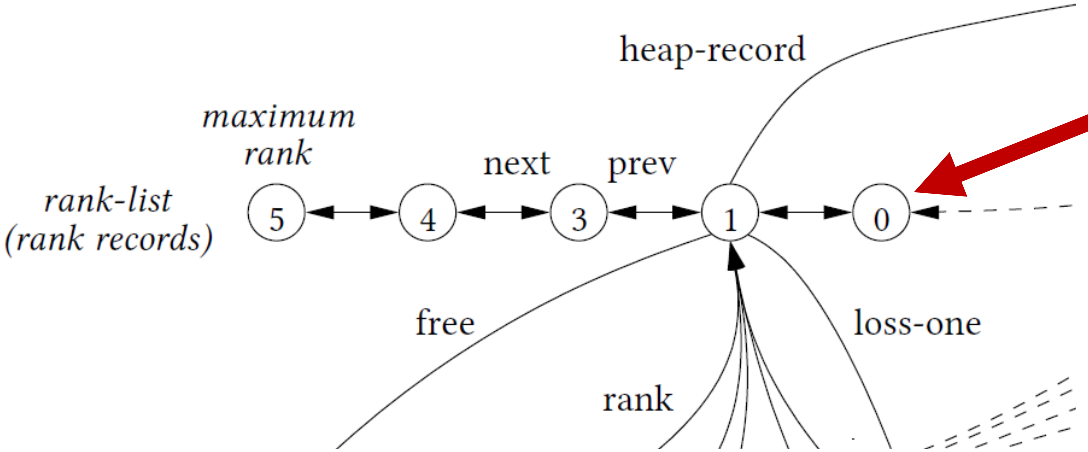
```
> coverage run odd_even.py
| 4 even
| 5 odd
| ...
| 1 odd
> coverage report -m odd_even.py
| Name            Stmts   Miss  Cover   Missing
| ---------------------------------------------
| odd_even.py        11      1    91%   5
| ---------------------------------------------
| TOTAL              11      1    91%
```

- Usually, code coverage is a measure of the quality of test cases
- ...but, can also help to identify missing logical insights

pypi.org/project/coverage

Branch coverage

- Thought code coverage would find all ”logical errors”
- Found several **if** statements with no **else** part, where condition provably would always be true
- Implementation → new invariants discovered (and assertions added)



always exists

coverage.readthedocs.io

Branch coverage

odd_even.py

```python
 1 def f(x):
 2     if x % 2 == 0:
 3         return 'even'
 4     elif x % 4 == 0:
 5         return 'even more even'
 6     elif x % 2 == 1:
 7         return 'odd'
 8 import random
 9 for i in range(10):
10     x = random.randint(0, 10)
11     print(x, f(x))
```

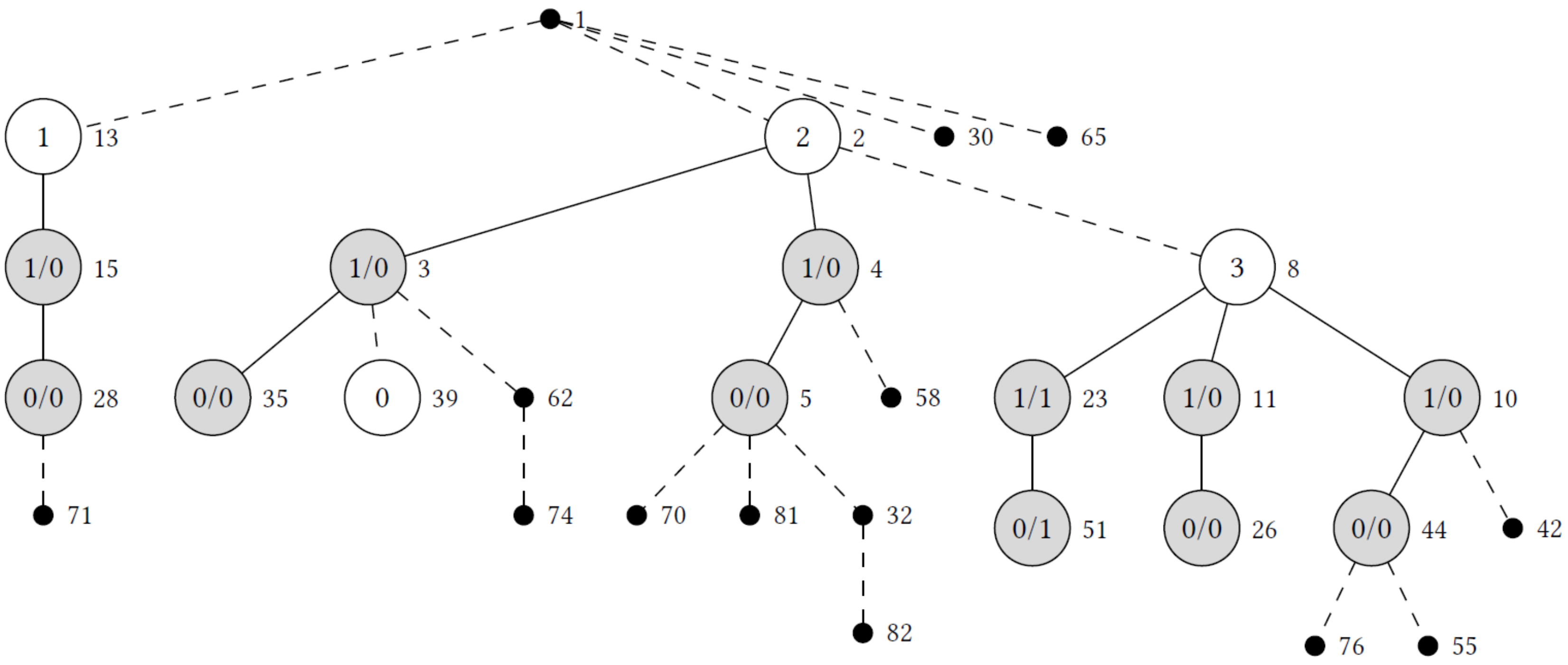
Shell

```
> coverage run --branch odd_even.py
| 5 odd
| 4 even
| ...
| 8 eve
> coverage report -m odd_even.py
| Name           Stmts   Miss Branch BrPart  Cover   Missing
| ----------------------------------------------------------
| odd_even.py       11      1      8      2    84%   5, 6->exit
| ----------------------------------------------------------
| TOTAL             11      1      8      2    84%
```

pypi.org/project/coverage

”The first main suggestion is to have at least one figure with a logical diagram of a non-trivial example structure, […]. This would go a long way in giving some idea of what the structure is.”

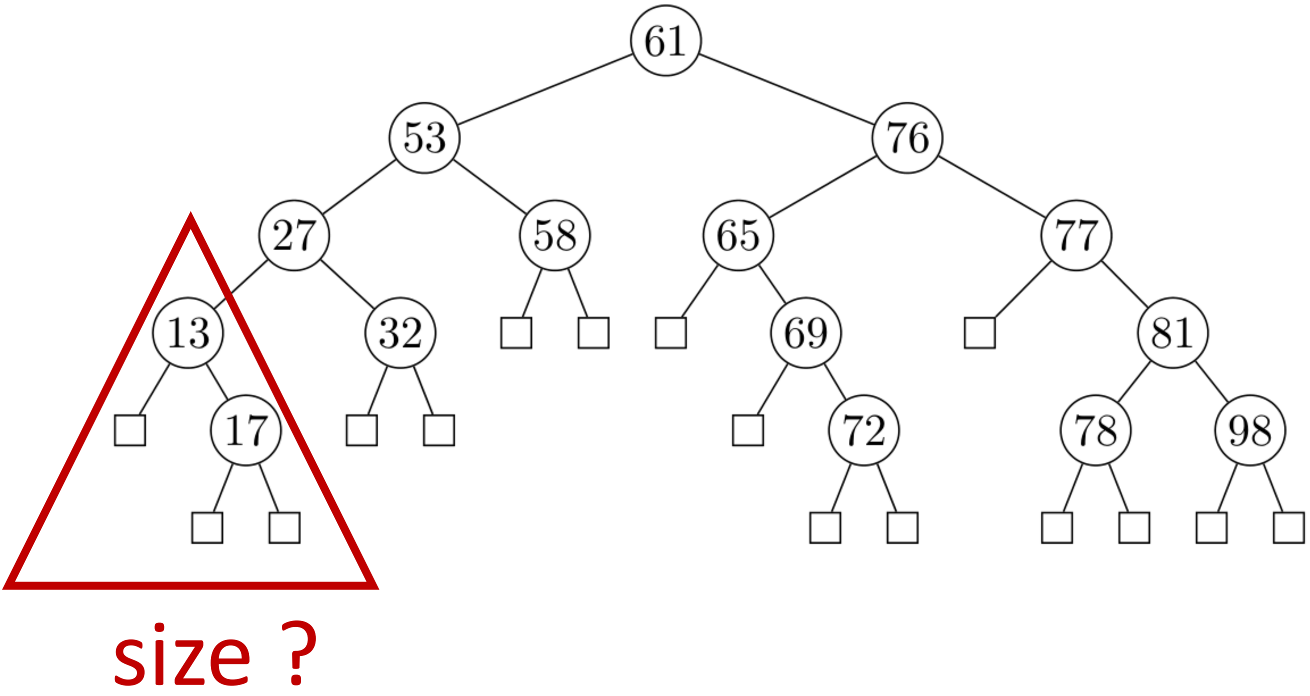
Anonymous reviewer

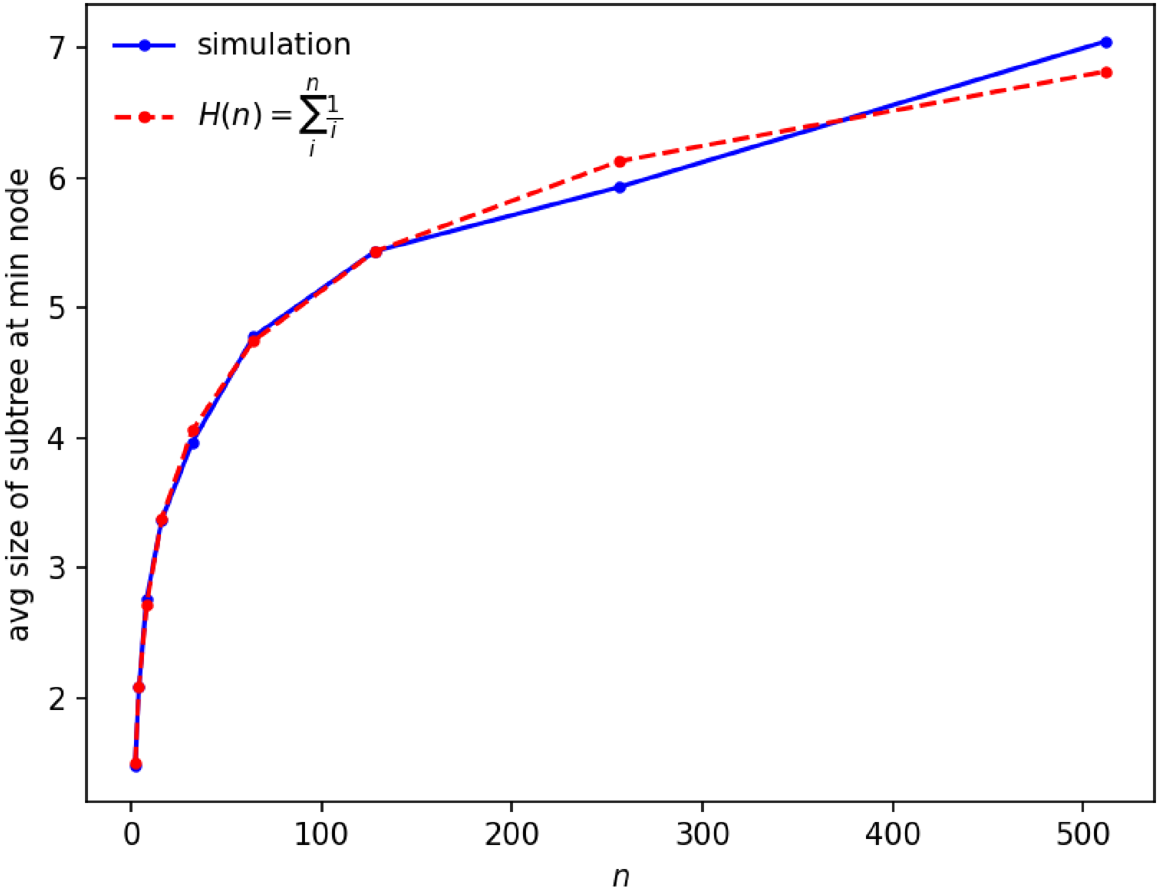


- Hard to manually create a figure that was guaranteed to be a real example
- Could use implementation to automatically generate (LaTeX tikz) figures
- Generated random inputs
- Formalized requirements to figure as a loop condition
- Repeat until happy

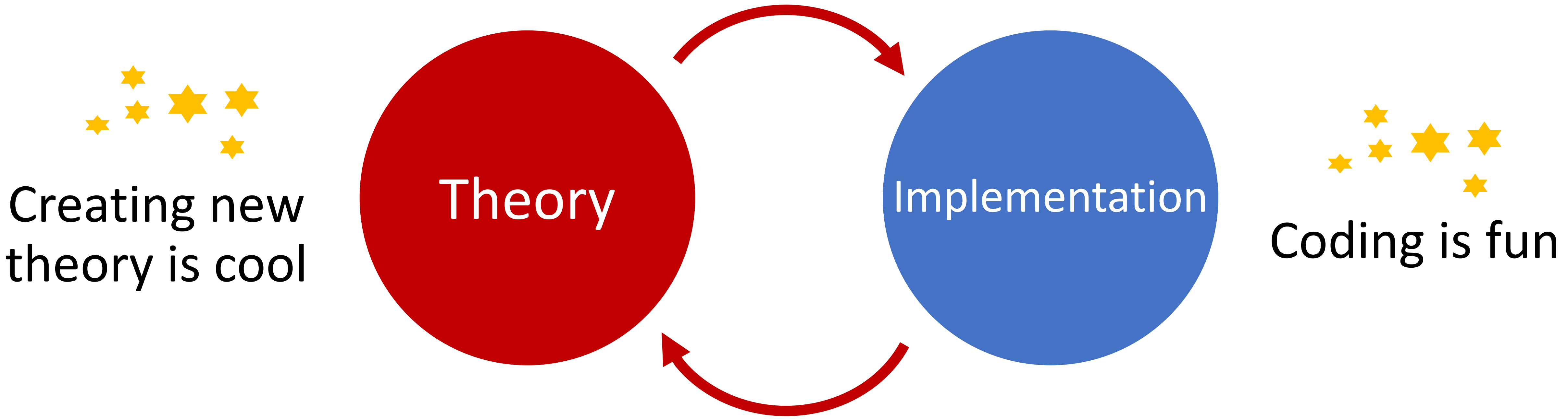
A question by John Iacono at Dagstuhl

- *After inserting n random elements into an unbalanced binary search tree, what is the expected size of the subtree rooted at the minimum?*





Summary



- Implementations support stronger theory
- Experimentation can identify what to prove
- Invariants can be verified and identified using assertions in code
- Stress tests and code coverage ensures integrity of code and theory