The challenges of implementing Dijkstra's algorithm

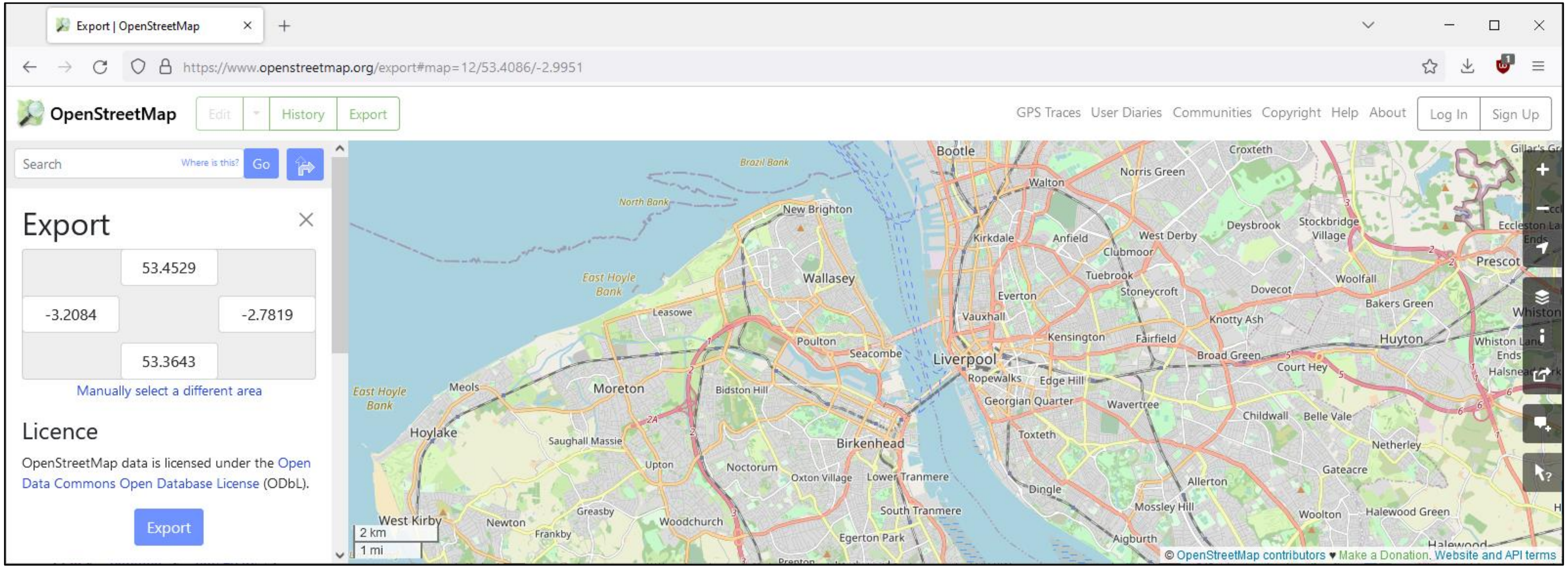
Gerth Stølting Brodal
Aarhus University

(Work presented at the 11th International Conference on Fun with Algorithms, FUN 2022)

Department of Computer Science, University of Liverpool, UK, January 9, 2023

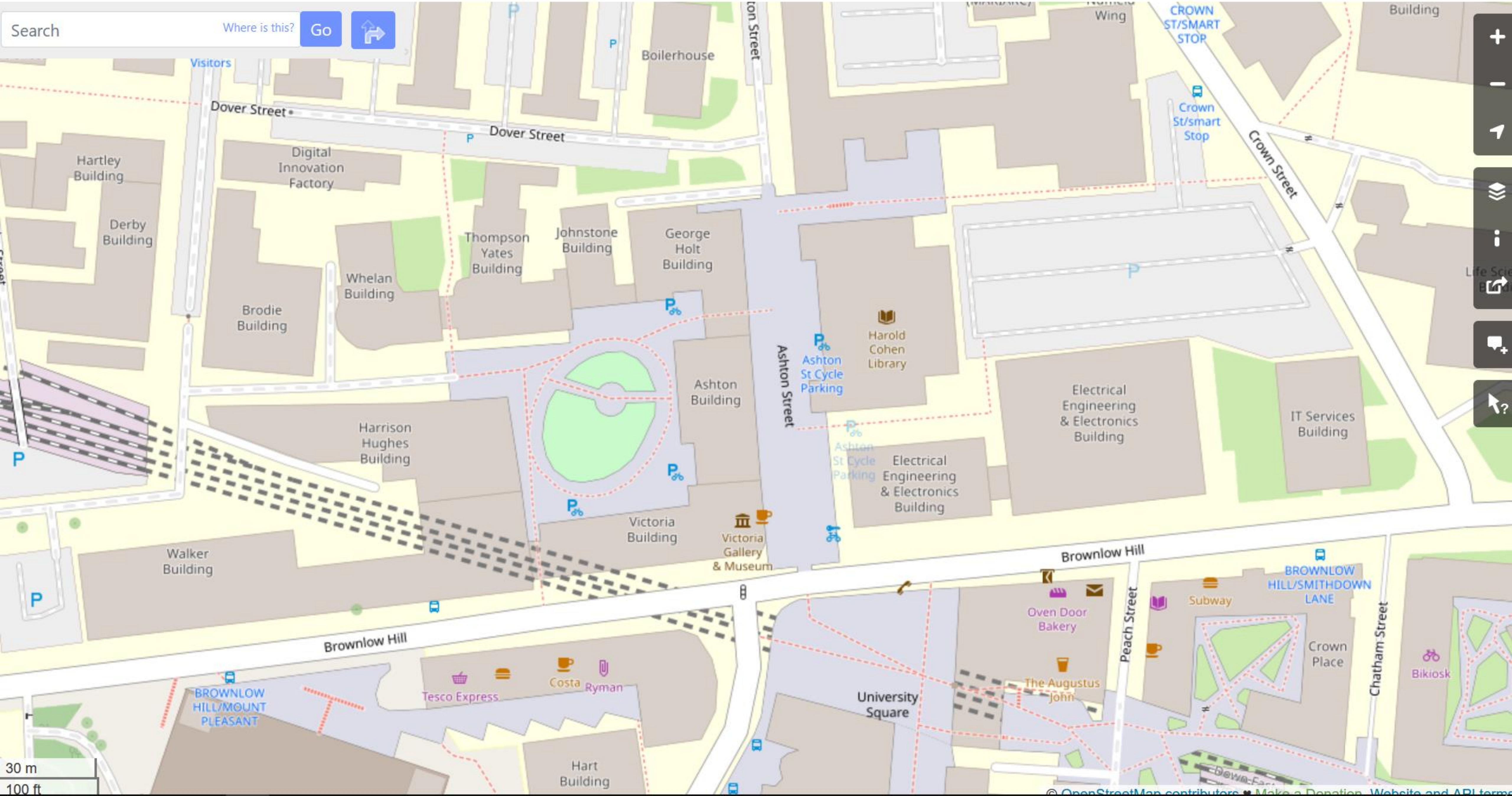
Background

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



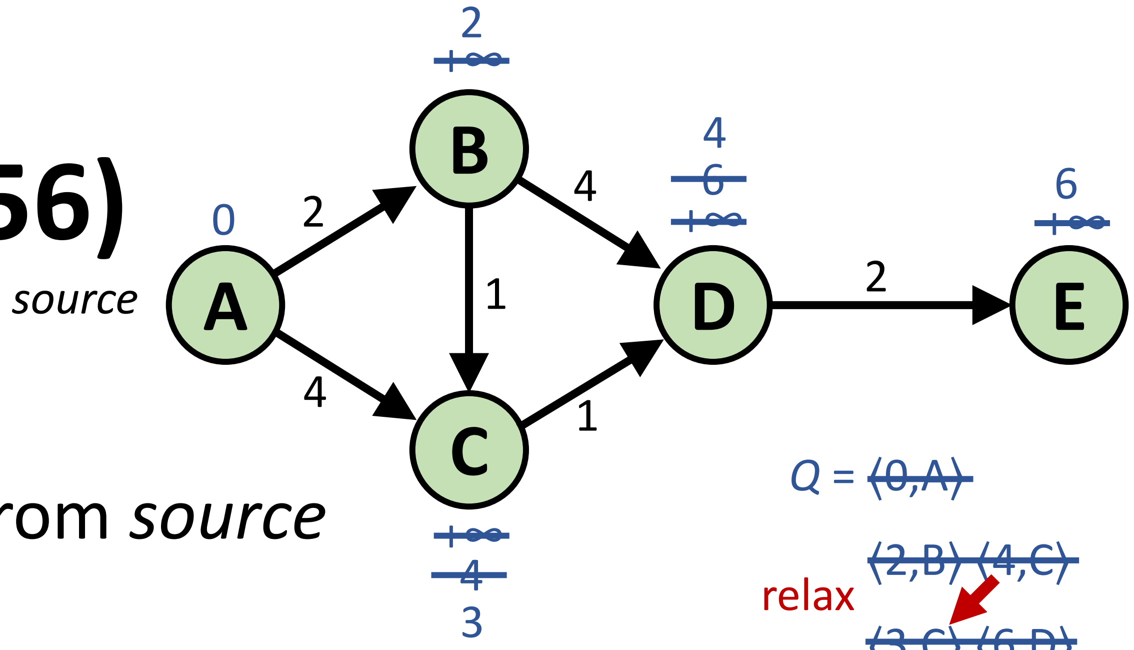
OpenStreetMap Edit History Export GPS Traces User Diaries Communities Copyright Help About Log In Sign Up

Search Where is this? Go



```
<node id="4606377719" visible="true" version="1" changeset="45143381" timestamp="2017-01-13T19:11:48Z" user="Dyserth" uid="1892838" lat="53.4064488" lon="-2.9660774"/>
<way id="4961017" visible="true" version="11" changeset="66454995" timestamp="2019-01-19T14:25:32Z" user="Redhochs" uid="1744940">
  <nd ref="348087292"/>
  <nd ref="4606377719"/>
  <nd ref="268780437"/>
  <nd ref="2700274352"/>
  <nd ref="268779145"/>
  <tag k="highway" v="pedestrian"/>
  <tag k="lit" v="yes"/>
  <tag k="name" v="Ashton Street"/>
  <tag k="surface" v="paving_stones"/>
</way>
<way id="826473129" visible="true" version="3" changeset="129789472" timestamp="2022-12-06T16:23:14Z" user="pmyteh" uid="81653">
  <nd ref="107366213"/>
  <nd ref="3468209232"/>
  <nd ref="3476406048"/>
  <nd ref="4112993338"/>
  <nd ref="2188828954"/>
  <nd ref="534390991"/>
  <nd ref="7839585527"/>
  <nd ref="348087295"/>
  <tag k="bicycle" v="yes"/>
  <tag k="foot" v="yes"/>
  <tag k="highway" v="tertiary"/>
  <tag k="horse" v="yes"/>
  <tag k="lanes" v="3"/>
  <tag k="lit" v="yes"/>
  <tag k="maxspeed" v="30 mph"/>
  <tag k="name" v="Brownlow Hill"/>
  <tag k="oneway" v="no"/>
  <tag k="postal_code" v="L3"/>
  <tag k="sidewalk" v="both"/>
  <tag k="surface" v="asphalt"/>
</way>
```

Dijkstra's algorithm (1956)



Q = ~~⟨0,A⟩~~

~~⟨2,B⟩~~ ~~⟨4,C⟩~~

relax

~~⟨3,C⟩~~ ~~⟨6,D⟩~~

~~⟨4,D⟩~~

~~⟨6,E⟩~~

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

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($m \cdot n$)

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

- comparator function

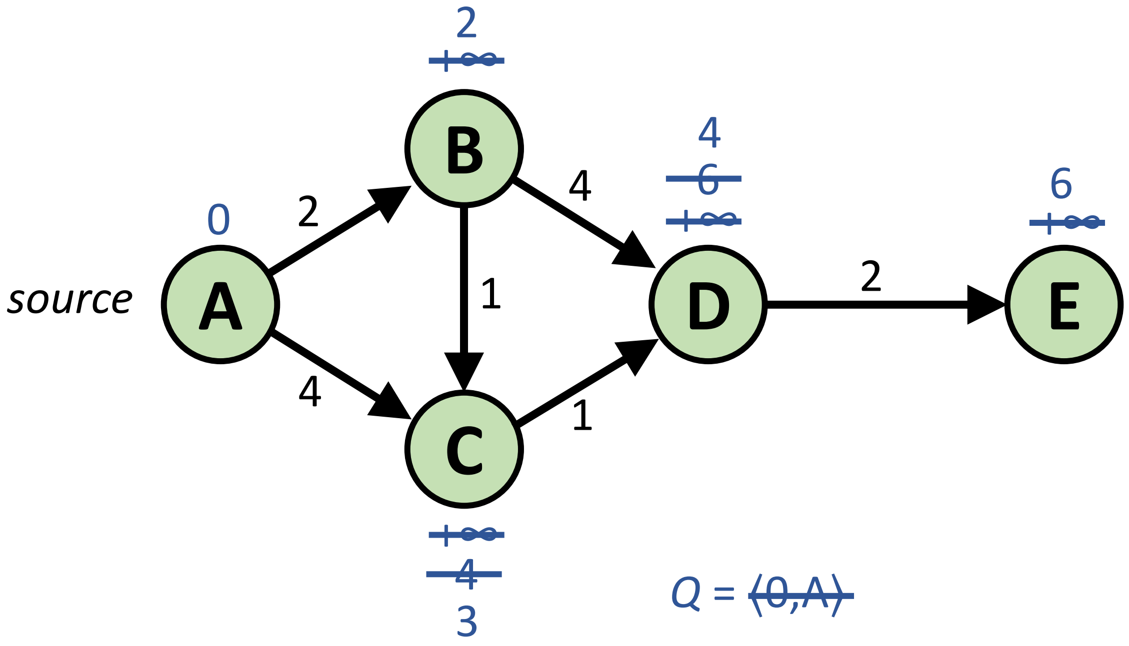
Java SE 18 & JDK 18

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



Q = ~~⟨0,A⟩~~

~~⟨2,B⟩ ⟨4,C⟩~~

~~⟨3,C⟩ ⟨4,C⟩ ⟨6,D⟩~~

~~⟨4,C⟩ ⟨4,D⟩ ⟨6,D⟩~~

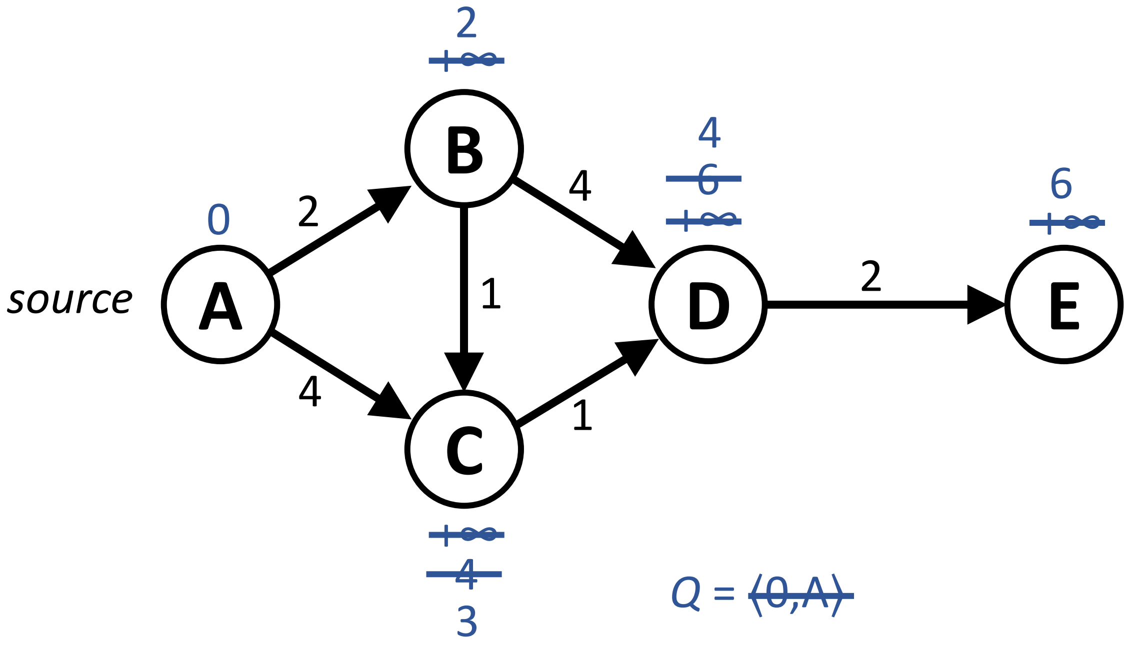
~~⟨4,D⟩ ⟨6,D⟩~~

~~⟨6,D⟩ ⟨6,E⟩~~

~~⟨6,E⟩~~

proc $\texttt{Dijkstra}_3(V, E, \delta, s)$
  $\mathit{dist}[v] = +\infty$ for all $v \in V \setminus \{s\}$
  $\mathit{dist}[s] = 0$
  $\texttt{Insert}(Q, \langle \mathit{dist}[s], s\rangle)$
  while $Q \neq \emptyset$ **do**
    $\langle d, u\rangle = \texttt{ExtractMin}(Q)$
outdated ? → **if** $d = \mathit{dist}[u]$ **then**
      for $(u, v) \in E \cap (\{u\} \times V)$ **do**
        if $\mathit{dist}[u] + \delta(u, v) < \mathit{dist}[v]$ **then**
          $\mathit{dist}[v] = \mathit{dist}[u] + \delta(u, v)$
relax = reinsert → $\texttt{Insert}(Q, \langle \mathit{dist}[v], v\rangle)$
  return dist

Using a visited set



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

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

$\quad dist[s] = 0$

$\quad visited = \emptyset$

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

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

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

use bitvector → $\qquad \textbf{if}\ u \notin visited\ \textbf{then}$

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

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

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

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

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

$\quad \textbf{return}\ dist$

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⟩~~

A shaky idea…

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

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

$\quad dist[s] = 0$

$\quad visited = \emptyset$

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

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

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

$\quad\quad \textbf{if}\ u \notin visited\ \textbf{then}$

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

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

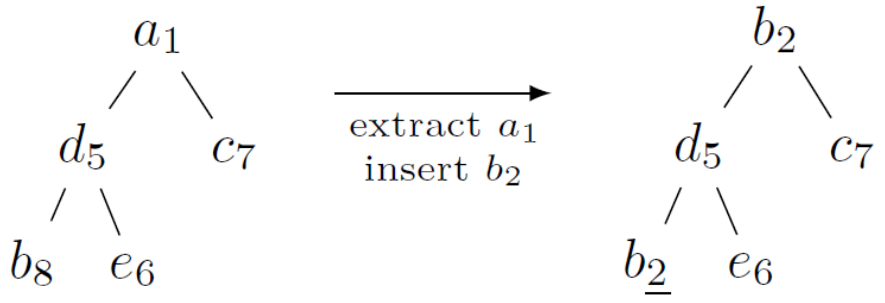
$\quad\quad\quad\quad \textbf{if}\ dist[u] + \delta(u, v) < dist[v]\ \textbf{then}$

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

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

$\quad \textbf{return}\ dist$

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



Heap invariants break

Experimental study

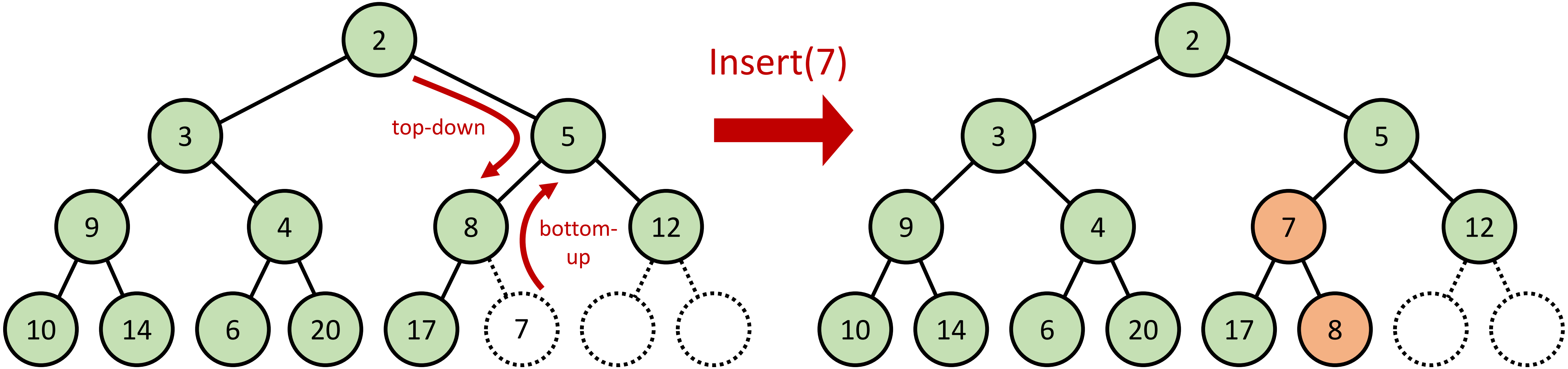
```
visited = set()
Q = Queue()
Q.insert(Item(0, source))
while not Q.empty():
    u = Q.extract_min().value
    if u not in visited:
        visited.add(u)
        for v in G.out[u]:
            dist_v = dist[u] + G.weights[(u, v)]
            if dist_v < dist[v]:
                dist[v] = dist_v
                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)
- Skew heaps worked
- Leftist heaps worked
- Pairing heaps worked
- Binomial queues worked
- Post-order heaps worked
- Binary heaps with top-down insertions worked

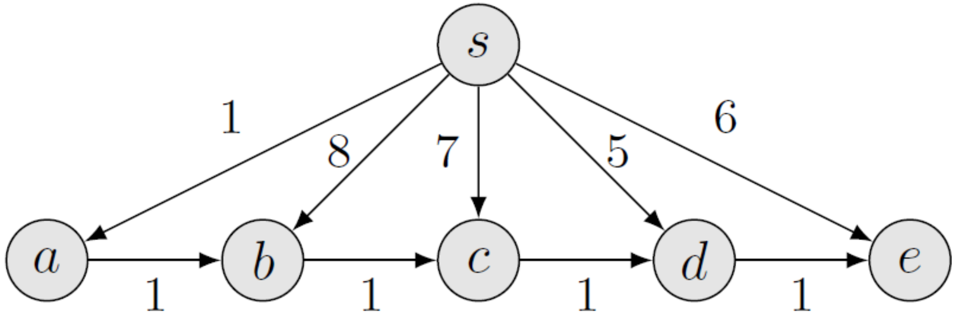
Pointer based

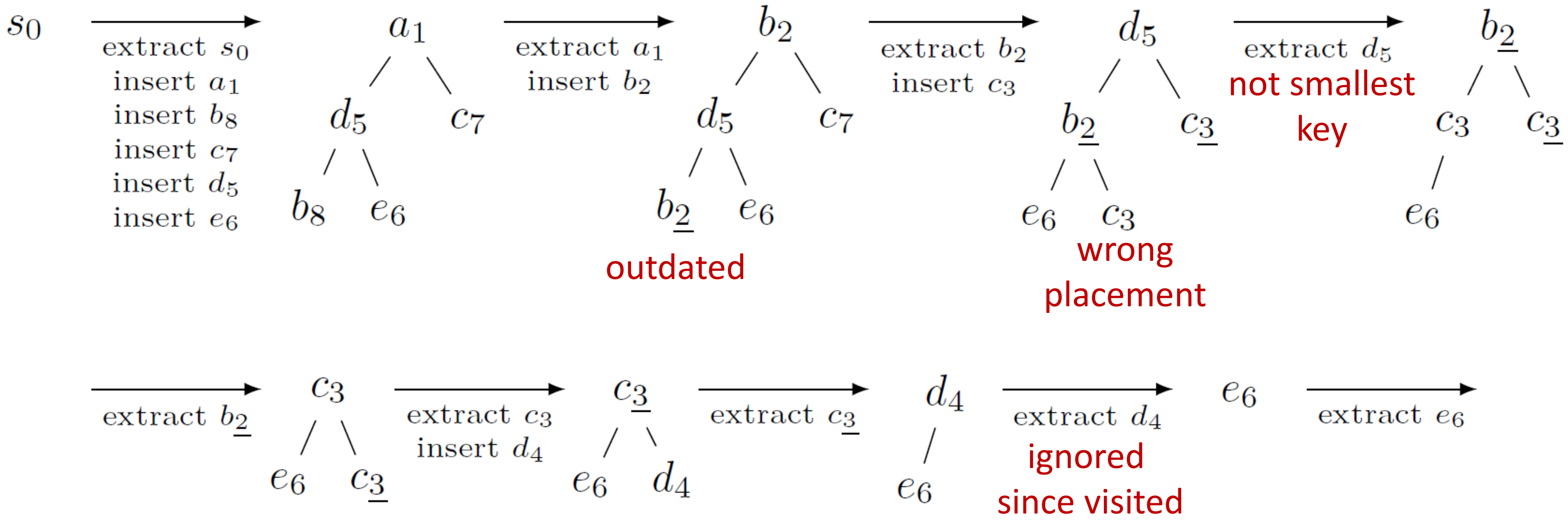
Implicit (space efficient)

Binary heap insertions
– bottom-up vs top-down



Binary heaps using *dist* in a comparator fails





Definition
Priority Queues supporting 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 ((Williams 1964))
- leftist heaps (Crane 1972)
- binomial queues (Vuillemin, 1978)
- skew heaps (Sleator, Tarjan 1986)
- pairing heaps (Fredman, Sedgewick, Sleator, Tarjan 1986)
- post-order heaps (Harvey, Zatloukal 2004)

Proof of Theorem 2 - Basic idea

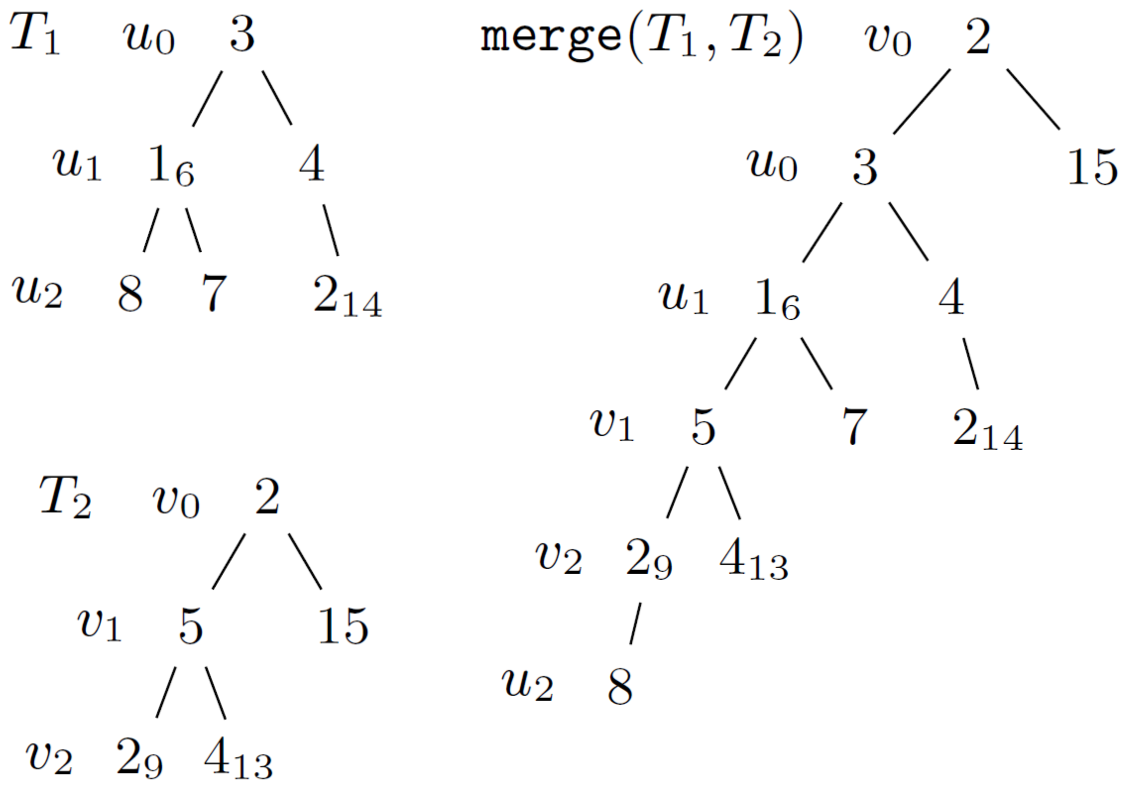
- Decreased heap order

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

- Root valid item to extract

- Top-down merging two paths preserves decreased heap order

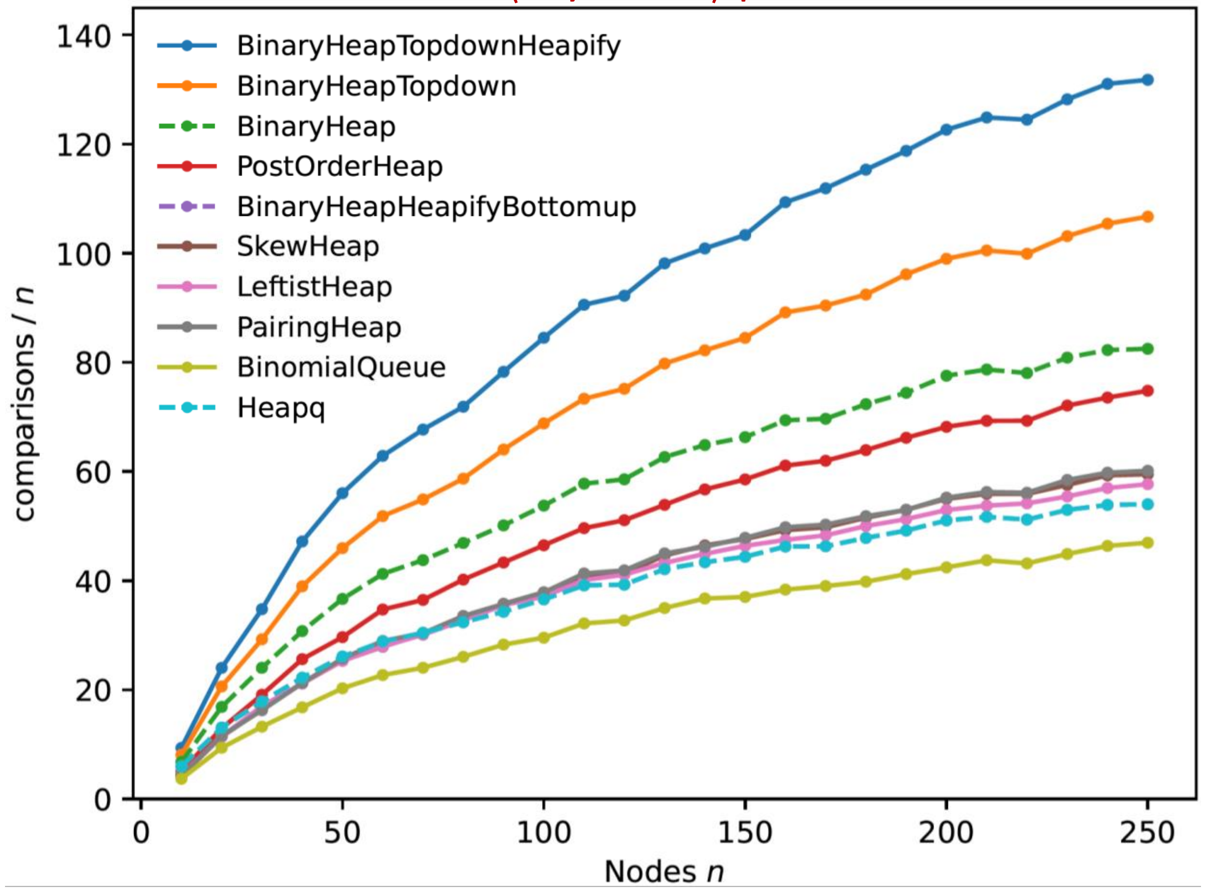
⇒ **skew heaps** and **leftist heaps** support decreasing keys



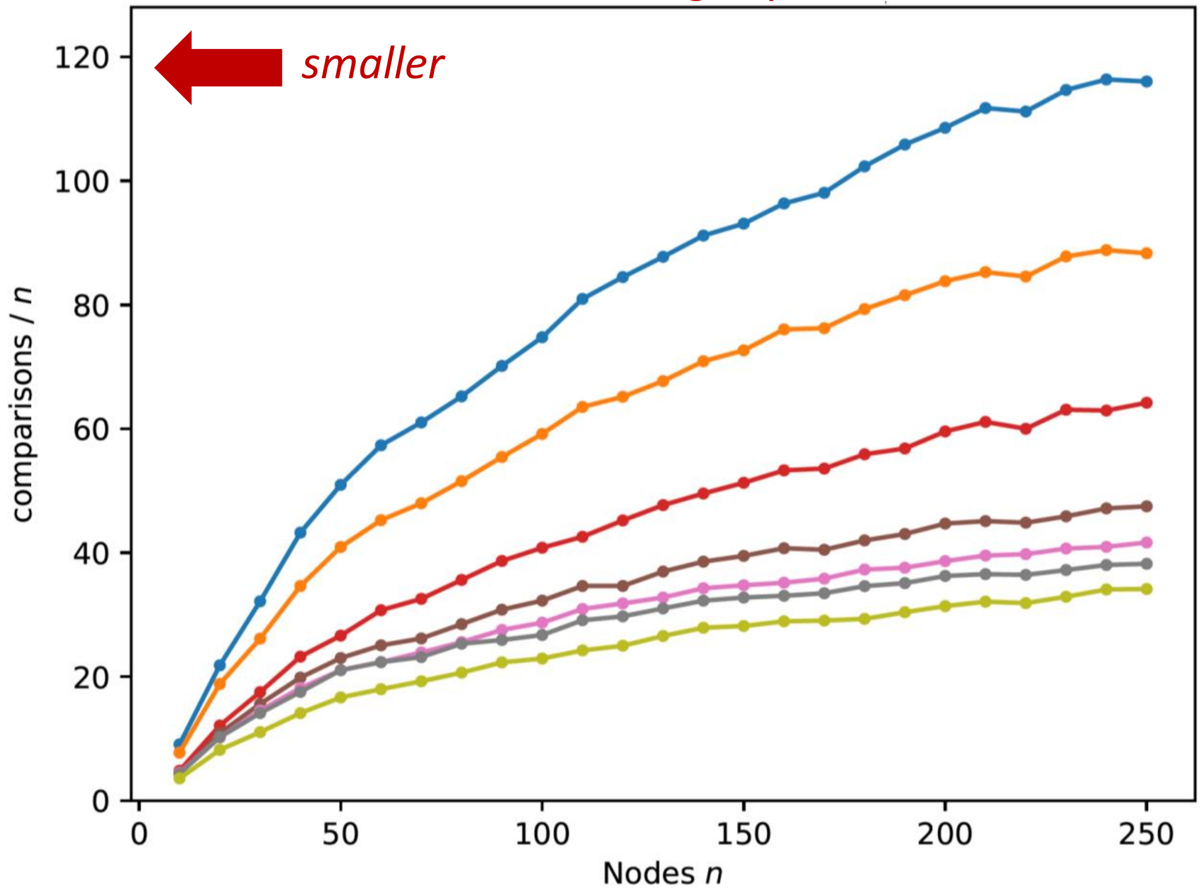
Experimental evaluation of various heaps

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

⟨key, value⟩ pairs

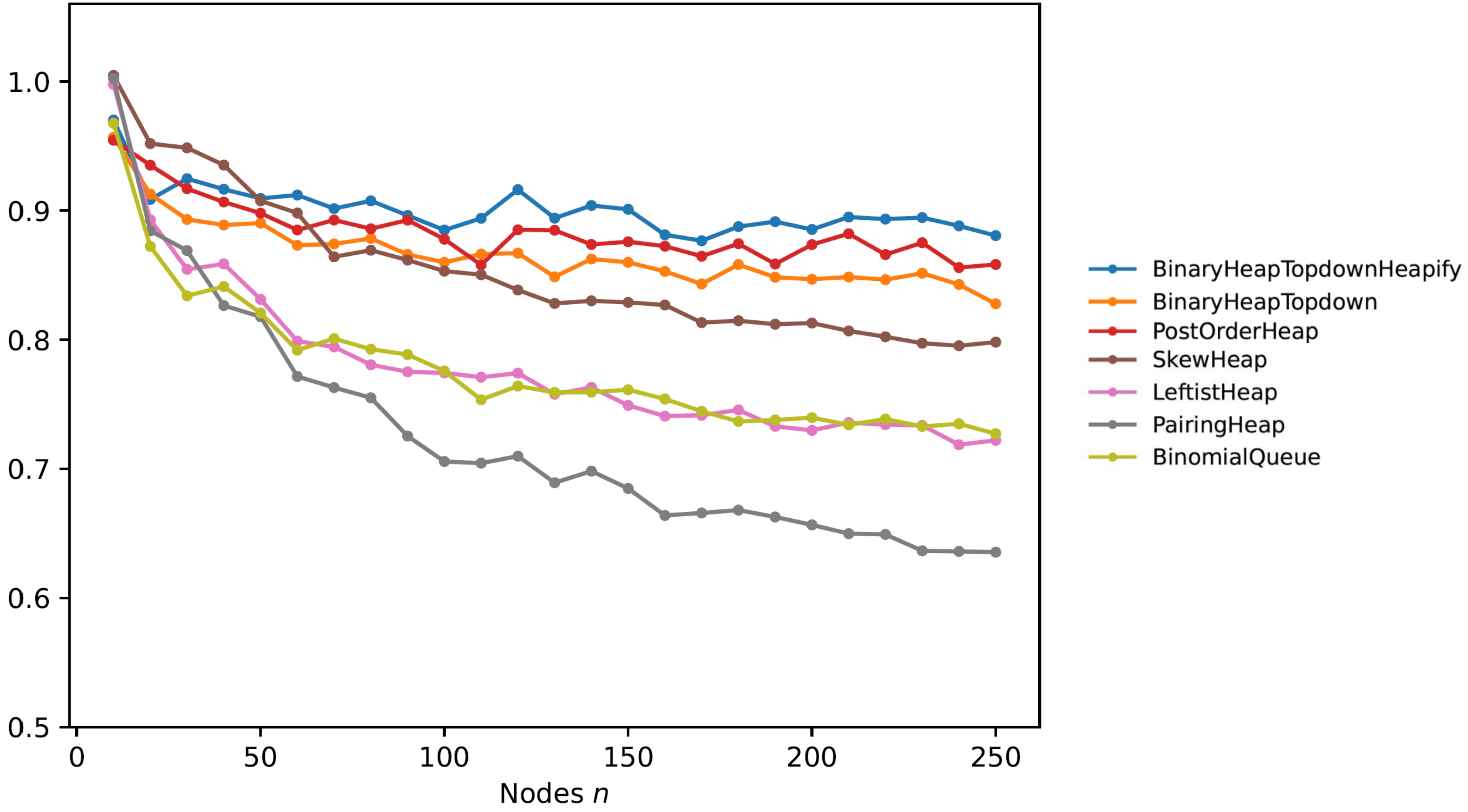


decreasing keys

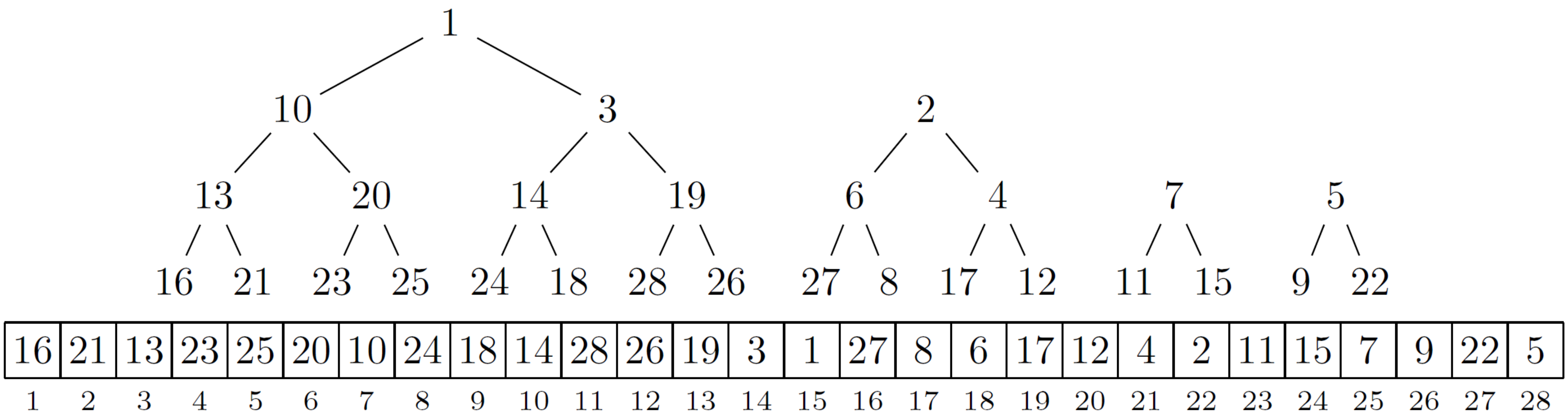


Reduction in comparisons

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)

Conclusion

- 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