Priority Queues with Decreasing Keys

Gerth Stølting Brodal ✉ 🏠 ⬤
Department of Computer Science, Aarhus University, Denmark

Abstract

A priority queue stores a set of items with associated keys and supports the insertion of a new item and extraction of an item with minimum key. In applications like Dijkstra's single source shortest path algorithm and Prim-Jarník's minimum spanning tree algorithm, the key of an item can decrease over time. Usually this is handled by either using a priority queue supporting the deletion of an arbitrary item or a dedicated `DecreaseKey` operation, or by inserting the same item multiple times but with decreasing keys.

In this paper we study what happens if the keys associated with items in a priority queue can decrease over time *without* informing the priority queue, and how such a priority queue can be used in Dijkstra's algorithm. We show that binary heaps with bottom-up insertions fail to report items with unchanged keys in correct order, while binary heaps with top-down insertions report items with unchanged keys in correct order. Furthermore, we show that skew heaps, leftist heaps, and priority queues based on linking roots of heap-ordered trees, like pairing heaps, binomial queues and Fibonacci heaps, work correctly with decreasing keys without any modifications. Finally, we show that the post-order heap by Harvey and Zatloukal, a variant of a binary heap with amortized constant time insertions and amortized logarithmic time deletions, works correctly with decreasing keys and is a strong contender for an implicit priority queue supporting decreasing keys in practice.

1 Introduction

A priority queue is a data structure storing a set of items, where each item has an associated key. A basic priority queue supports the two operations `Insert` and `ExtractMin`, which insert a new item into the priority queue and extract an item with minimum key from the priority queue. A classic example of a data structure supporting these operations is the binary heap by Williams from 1964 [20]. Although many priority queues exist supporting a more comprehensive list of operations or having better asymptotic bounds, the binary heap is the standard priority queue implementation in many languages, like in Python (module `heapq`) and Java (class `java.util.PriorityQueue`). The popularity of binary heaps is due to its simplicity, it can be stored implicitly in an (extendable) array only storing the n items currently in the heap, and the number of comparisons is relatively low. Insertions require at most $\log_2 n$ comparisons, and minimum extractions at most $2\log_2 n$ comparisons, but the number of comparisons performed are often lower in practice.

Two graph algorithms fundamentally relying on efficient priority queue implementations are Dijkstra's algorithm [5] for finding shortest paths from a source node in directed graphs with non-negative edge weights, and Prim-Jarník's algorithm [11, 16] for finding the minimum spanning tree in a graph. Both maintain a priority queue over the nodes not included yet in the shortest path tree and minimum tree, respectively. A node in the priority queue has an

associated key equal to the shortest distance to the node discovered so far and the lightest edge connecting the node to the minimum spanning tree constructed so far, respectively. For both algorithms the key of a node in the priority queue can decrease over time, which challenges the interface of the basic priority queue. One solution is to apply a more specialized priority queue, like Fibonacci heaps [9], which support a dedicated `DecreaseKey` operation – but this structure is more complicated, pointer based, and often not part of a standard library. Although theoretically worst-case superior, the overhead of supporting `DecreaseKey` only pays off when a large fraction of the edges cause a `DecreaseKey` operation to be performed. A simpler solution is to stay with a basic priority queue and just insert a node multiple times, once whenever the key decreases. This leaves outdated copies of nodes in the priority queue, but these can be skipped whenever they are extracted from the priority, since only the first extraction of a node is not outdated. Two possible implementations of this idea for Dijkstra's algorithm are shown as algorithms $\texttt{Dijkstra}_3$ and $\texttt{Dijkstra}_4$ in Figure 1. In the following we do not discuss Prim-Jarník's algorithm any further.

In a typical implementation of Dijkstra's algorithm one maintains an array $dist$, where $dist[v]$ is the currently shortest known distance from the source node to node v, and the items inserted into the priority queue are pairs $\langle dist[v], v \rangle$, where $dist[v]$ is the key of the item. In this paper we consider adopting the idea of only storing v as an item in the priority queue *without* an explicitly associated key. The comparison between two items in the priority instead compares the current distances $dist[v]$. This will reduce the space usage for the priority queue, e.g., a binary heap only needs to store an array of node ids. The challenge is now that keys of inserted items can decrease over time, i.e., the ordering of the items in the priority queue changes over time and potentially invalidates the internal invariants maintained by a priority queue. In this paper we identify comparison based priority queues working correctly with decreasing keys. In particular we show that skew heaps [18], leftist heaps [4], binomial queues [19], pairing heaps [8], Fibonacci heaps [9], and post-order heaps [10] work correctly even with decreasing keys. For binary heaps [20] we show that the standard implementation with bottom-up insertions fails to support decreasing keys, whereas binary heaps work correctly if operations are performed top-down.

Model

We define a *priority queue with decreasing keys* as follows. It stores a set of items where each item has an associated key from some totally ordered universe. Over time the key of an item can decrease an arbitrary number of times. We let the *original key* of an item refer to the key when the item was inserted, whereas the *current key* refers to the key at the current time. If the current key equals the original key we say that the item has an *unchanged* key. The priority queue is not informed when keys decrease, and whenever two items are compared by the priority queue the comparison is performed with respect to their current keys. The priority queue has no access to the original keys of the items; it can only compare two items and get the relative order of their current keys. Note that the answer to the comparison between two items can vary over time depending on how keys decrease.

A priority queue with decreasing keys should support the following two operations:

- `Insert(x)` inserts an item into the priority queue.
- `ExtractMin()` returns an item from the priority queue with *current key* less than or equal to the *original keys* of all items in the priority queue.

It follows that if `ExtractMin` returns an item with unchanged key, the item has smallest key among all items with unchanged key. Furthermore, if several items in the priority queue have current key less than or equal to the smallest original key in the priority queue, then the

priority queue is allowed to return any of these items, i.e. its behavior is non-deterministic. As an example, consider a priority queue with four inserted items A, B, C and D with original keys 5, 2, 6 and 4, respectively. Assume C and D have had their keys decreased to have current keys 3 and 1, respectively. Then `ExtractMin` should return either B or D, with current keys 2 and 1, respectively, since B has smallest original key equal to 2, and A and C have current keys 5 and 3, respectively. In Section 2 we discuss how an implementation of Dijkstra's algorithm can benefit from priority queues with decreasing keys.

Contributions

This paper introduces no new data structure. Only existing data structures are analyzed in the context of decreasing keys. Our contributions are:

- Section 2: We show that Dijktra's algorithm [5] (`Dijkstra`$_4$ in Figure 1) works correctly when using a priority queue with decreasing keys, i.e., items in the priority queue only store a node v instead of the pair $\langle dist[v], v \rangle$.
- Section 3: Binary heaps [20] with bottom-up insertions do not support decreasing keys, in particular we show that sorting (with interleaved key decreases) and Dijkstra's algorithm fail on small examples.
- Section 4: Binary heaps with top-down insertions support decreasing keys, i.e., inserting a new item considers the ancestors of the new leaf top-down until the first ancestor is found with key greater than or equal to the new key. The central invariant used in the analysis, and also used in Sections 5 and 6, is *decreased heap order* requiring that that any ancestor of a node v in a tree must store an item with current key less than or equal to the original key of v.

  Without decreasing keys, bottom-up and top-down insertions cause the nodes of the resulting heaps to store identical keys, but for random insertions the number of comparisons increases from average $O(1)$ [15] to $\Theta(\log n)$. In Section 7 we do an experimental comparison of the two variants of a binary heap, and in particular the overhead introduced by performing insertions top-down.
- Section 5: Skew heaps [18], leftist heaps [4], pairing heaps [8], binomial queues [19], and Fibonacci heaps [9] work correctly with decreasing keys.
- Section 6: The post-order heap by Harvey and Zatloukal [10] supports decreasing keys. The post-order heap is a simple implicit heap based on binary heaps that supports insertions in amortized constant time and extractions in amortized logarithmic time (like, e.g., binomial queues). In Section 7 our experimental evaluation shows that the post-order heap is a strong contender for an efficient implicit priority queue supporting decreasing keys.
- Section 7: We supplement our theoretical results with an experimental evaluation of priority queues supporting decreasing keys and compare the number of key comparisons performed to sort and for running Dijkstra's algorithm on cliques.

Related work

The literature on priority queues is comprehensive, see, e.g., the survey by Brodal [1]. Fibonacci heaps [9] support `DecreaseKey` in amortized constant time and their discovery initiated the study of data structures supporting efficient `DecreaseKey` operations. Subsequently, e.g., relaxed heaps [6] were introduced, which support `DecreaseKey` in worst-case constant time. In this paper we focus on simpler data structures, not supporting `DecreaseKey` operations. Many priority queues can be extended to support an arbitrary `Remove` operation,

by having a separate index keeping track of where each item is stored in the data structure. This introduces a space overhead for the index and a time overhead to keep the index updated, e.g., swapping two items in a binary heap requires two entries in the index to be updated. If only `Insert` and `ExtractMin` need to be supported, many simple constructions exist – a few commonly used are mentioned and evaluated in this paper. A special interesting class of priority queues are those that can be stored in a single array containing the items, known as *implicit* priority queues. The classic example is the binary heap [20], but other examples are, e.g., the implicit binomial trees by Carlsson, Munro and Poblette [2], and the post-order heap by Harvey and Zatloukal [10] that is our focus in Section 6. Many priority queues maintain a forest of trees of sizes corresponding to digits in the binary or skew binary representation of the total number of items stored. Elmasry, Jensen and Katajainen [7] give an overview of this relationship for various constructions.

Background

The motivation for studying priority queues with decreasing keys arose from experience with undergraduate students having problems translating Dijkstra's shortest path algorithm into correct Java programs based on the description in the standard text book by Cormen *et al.* [3, Section 24.3]. Students are challenged by the fact that the priority queue implementation supported by the Java standard library[1] does not support `DecreaseKey`, the `Remove` operation requires linear time, and the ordering of values is provided using a comparator (or the natural ordering of the values). A priority queue with decreasing keys allows Dijkstra's algorithm to store node identifiers only in the priority and using a comparator to compare two nodes by comparing their currently best known distances – the Java solution many students implement, but fails since their priority queue does not support decreasing keys.

2 Dijkstra's algorithm with decreasing keys

Assume we are given a directed graph $G = (V, E)$ with non-negative edge weights δ and a source node $s \in V$, and we want to compute the shortest distance from s to all nodes in the graph. This problem can be solved using Dijkstra's algorithm [5]. The basic idea of Dijkstra's algorithm is to visit nodes in increasing distance from s. For each node v not visited yet its currently known distance $dist[v]$ is stored in an array $dist$, i.e., the distance to v along paths only containing v and already visited nodes. The next node to visit is a node u not visited so far and with smallest $dist[u]$ value. When visiting a node u we *relax* along its outgoing edges (u, v) by performing the update $dist[v] := \min(dist[v], dist[u] + \delta(u, v))$. To obtain an efficient solution, the set of nodes not visited yet are stored in a priority queue, with $dist[v]$ as the key of v. Fibonacci heaps [9] provide a dedicated `DecreaseKey` operation to update (decrease) the key of a node whenever the known distance to a node decreases. Using a Fibonacci heap Dijkstra's algorithm can be implemented as shown in Dijkstra_1 in Figure 1 obtaining running time $O(|E| + |V| \log |V|)$. If no `DecreaseKey` is available, but an arbitrary item can be removed by a `Remove` operation, we can simulate `DecreaseKey` by first removing the node using `Remove` and then reinserting the node with its smaller distance as key using `Insert` as shown in Dijkstra_2 in Figure 1. If `Remove` takes logarithmic time, the resulting running time is $O(|E| \log |V|)$. For sparse graphs, i.e., $|E| = O(|V|)$, the two running times are asymptotically identical.

---

[1] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/PriorityQueue.html`

```
proc Dijkstra₁(V, E, δ, s)
    dist[v] = +∞ for all v ∈ V \ {s}
    dist[s] = 0
    Insert(Q, ⟨dist[s], s⟩)
    while Q ≠ ∅ do
        ⟨d, u⟩ = ExtractMin(Q)
        for (u, v) ∈ E ∩ ({u} × V) do
            if dist[u] + δ(u, v) < dist[v] then
                dist[v] = dist[u] + δ(u, v)
                if v ∈ Q then
                    DecreaseKey(Q, v, dist[v])
                else
                    Insert(Q, ⟨v, dist[v]⟩)
    return dist


proc Dijkstra₃(V, E, δ, s)
    dist[v] = +∞ for all v ∈ V \ {s}
    dist[s] = 0
    Insert(Q, ⟨dist[s], s⟩)
    while Q ≠ ∅ do
        ⟨d, u⟩ = ExtractMin(Q)
        if d = dist[u] then
            for (u, v) ∈ E ∩ ({u} × V) do
                if dist[u] + δ(u, v) < dist[v] then
                    dist[v] = dist[u] + δ(u, v)
                    Insert(Q, ⟨dist[v], v⟩)
    return dist
```

```
proc Dijkstra₂(V, E, δ, s)
    dist[v] = +∞ for all v ∈ V \ {s}
    dist[s] = 0
    Insert(Q, ⟨dist[s], s⟩)
    while Q ≠ ∅ do
        ⟨d, u⟩ = ExtractMin(Q)
        for (u, v) ∈ E ∩ ({u} × V) do
            if dist[u] + δ(u, v) < dist[v] then
                dist[v] = dist[u] + δ(u, v)
                if v ∈ Q then
                    Remove(Q, v)
                Insert(Q, ⟨dist[v], v⟩)
    return dist


proc Dijkstra₄(V, E, δ, s)
    dist[v] = +∞ for all v ∈ V \ {s}
    dist[s] = 0
    visited = ∅
    Insert(Q, ⟨dist[s], s⟩)
    while Q ≠ ∅ do
        ⟨d, u⟩ = ExtractMin(Q)
        if u ∉ visited then
            visited = visited ∪ {u}
            for (u, v) ∈ E ∩ ({u} × V) do
                if dist[u] + δ(u, v) < dist[v] then
                    dist[v] = dist[u] + δ(u, v)
                    Insert(Q, ⟨dist[v], v⟩)
    return dist
```

Figure 1 Four variations of Dijkstra's algorithm for the single source shortest path problem on a digraph with nodes V, edges E, edge weights δ, and source node s. The main result of this paper is that Dijkstra₄ still works correctly if we adopt a priority with decreasing keys.

Here we consider a simpler implementation using a binary heap only supporting `Insert` and `ExtractMin`, but also achieving running time $O(|E| \log |V|)$. Whenever a shorter distance is found to a node v, we insert the item $\langle dist[v], v \rangle$ into the heap, i.e., the same node v can be inserted multiple times, but with decreasing keys. All instances of v in the heap, except for the one with key equal to the current $dist[v]$, are outdated and should be ignored/skipped when extracted from the heap. We can identify nodes to be skipped by either comparing the extracted distance with the currently best known distance or by keeping a set of all visited nodes, e.g., a bit-vector. Algorithms Dijkstra₃ and Dijkstra₄ in Figure 1 contain the pseudo code for these implementations of Dijkstra's algorithm. We will not argue further about the correctness of these variations of Dijkstra's algorithm (leaving outdated items in the priority queue is also a common approach to external memory algorithms for the single source shortest path problem, see, e.g., [12, Section 4.2]).[2]

---

[2] In the worst-case the priority queue stores $O(|E|)$ items, but this can be reduced to $O(|V|)$ items by rebuilding the heap whenever it contains $> (1+\varepsilon)|V|$ items, for some constant $\varepsilon > 0$, where all outdated items are removed. The time for rebuilding the heap can be charged to the removed items, i.e., the asymptotic running time remains unchanged.

Crucial to the above implementations of Dijkstra's algorithm is that each item we insert into the heap is a pair $\langle dist[v], v \rangle$, where the key $dist[v]$ is fixed when inserted. These keys require space in the heap, that could be tempting to save. The (potentially dangerous) idea is now: *Skip storing the keys explicitly in the items and instead use the current value $dist[v]$ as the current key for all items storing v in the heap.*

In the following we argue that $\mathtt{Dijkstra}_4$ still works correctly if we adopt a priority with decreasing keys, e.g., those in Sections 4–6. The only changes to $\mathtt{Dijkstra}_4$ is that $\mathtt{Insert}$ should only take the node to insert (without the distance), and $\mathtt{ExtractMin}$ does not return the key/distance d (that anyway was not used by $\mathtt{Dijkstra}_4$ after the item was extracted), and whenever the priority queue compares the keys of two nodes u and v it compares the currently known distances $dist[u]$ and $dist[v]$.

The invariant maintained by the algorithm is that only nodes v with $dist[v] < +\infty$ are stored in the priority queue, and for all nodes with $dist[v] < +\infty$ and $v \notin visited$, the priority queue contains an item containing v with unchanged key equal to the current $dist[v]$. This is true since we insert an item containing v with original key $dist[v]$ whenever $dist[v]$ decreases.

Whenever an item with a node v is extracted from the priority queue, we have three cases: *i*) $v \in visited$, *ii*) $v \notin visited$ and the current key of the item equals the original key, and *iii*) $v \notin visited$ and current key of the item is less than the original key. In case *i*) we extract a node that has already been visited, and therefore should be skipped. In case *ii*) we extract an item with current key equal to its original key. Since the priority queue guarantees that the current key of the extracted item is less than or equal to all original keys stored in the priority queue, the item has minimum original key among all items in the priority queue. From the invariant it follows that v has smallest $dist[v]$ value among all nodes not visited yet – as expected by Dijkstra's algorithm. Finally, in case *iii*) an item is extracted containing a node v not visited yet and with current key less than its original key. By the priority queue specification, its current key, i.e., $dist[v]$, is less than all original keys in the priority queue, in particular those stored with unchanged keys. Since $v \notin visited$ the invariant implies there must exist another item in the priority queue storing v with original and current key equal to $dist[v]$ – i.e., v is a node not visited yet with minimum $dist$ value as expected by Dijkstra's algorithm. It follows that a priority queue supporting decreasing keys extracts unvisited nodes in increasing order of distance as required by Dijkstra's algorithm.

3   Binary heaps fail with decreasing keys

In this section we show how binary heaps with bottom-up insertions [20] fail to support decreasing keys on two simple examples: Sorting and Dijkstra's single source shortest path algorithm.

We first briefly recall the structure of a binary heap. A binary heap stores n items in an array $H[1..n]$, that can be viewed as a binary tree where node i has children $2i$ and $2i + 1$.[3] The items are stored such that *heap order* is satisfied, i.e., the key of item $H[i]$ is greater than or equal to the key of its parent $H[\lfloor i/2 \rfloor]$. A bottom-up insertion places the new item as the last item in H and repeatedly swaps it with its parent (sift-up) as long as its key is less than the parent's *current* key. A minimum extraction returns the item at the root $H[1]$, and moves the last item x from $H[n]$ to $H[1]$, and sifts-down x by repeatedly swapping x with the item with smallest key among its children until no child stores an item with key less than x.

---

[3] In the paper we assume arrays start at index 1. In our implementation we adapt to Python lists, which start at index 0.
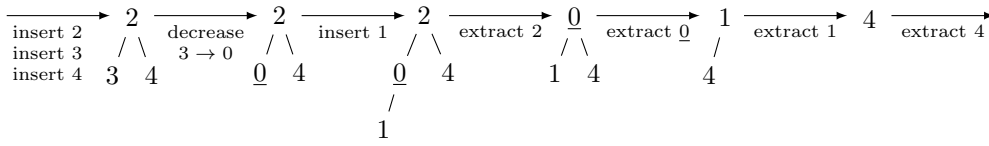
Figure 2 Binary heap failing to sort $\langle 2, 3, 4, 1 \rangle$ if 3 is decreased to 0 before inserting 1.
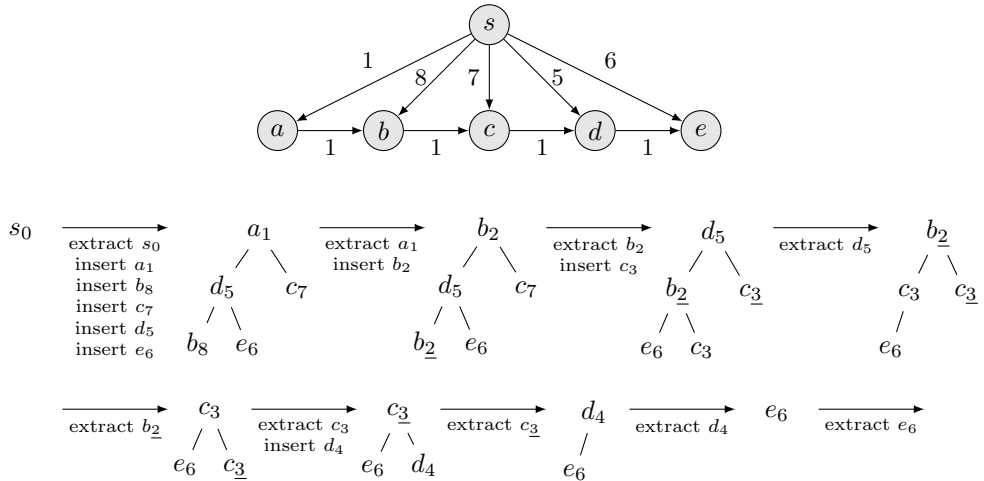
Figure 3 Execution of Dijkstra's single source shortest path algorithm ($\texttt{Dijkstra}_4$), using $dist[v]$ as keys and a binary heap with bottom-up insertions, incorrectly computing the distance to e as 6. Top is input graph and below the content of the binary heap. Subscripts are keys, and underlined keys are decreased keys.

Sorting

Consider sorting items by inserting them into a priority queue and then extracting them in increasing key order. If an item gets its key decreased during the sequence of operations, a priority queue with decreasing keys guarantees that the items with unchanged keys are still reported in increasing key order. A binary heap with bottom-up insertions fails to do so when inserting four items with keys 2, 3, 4 and 1, and where key 3 is decreased to 0 before inserting 1, as illustrated in Figure 2. Recall that the heap is not informed when keys decrease, causing the current keys to violate heap order. Instead of reporting the items with unchanged keys in order $\langle 1, 2, 4 \rangle$ they are reported in order $\langle 2, 1, 4 \rangle$. Note that when inserting 1 as the last leaf, it is compared to its parent with current key 0 (but original key 3), where the sift-up terminates, and incorrectly leaves 1 in the subtree of 2, causing 2 to be the first item extracted by $\texttt{ExtractMin}$.

Dijkstra's algorithm

Figure 3 shows a graph with 6 nodes and 9 edges, where Dijkstra's algorithm ($\texttt{Dijkstra}_4$) fails when using a binary heap with bottom-up insertions and using $dist[v]$ as the current key for node v. Whenever a smaller distance to a node is discovered, the node is inserted with the new distance as its original key. The previously inserted copies of the node (if any) get their current keys decreased to the new smaller distance, like b where $b_{\underline{2}}$ is the copy of b

in the heap with original key 8 and current key 2. When extracting b_2 in the example, e_6 is sifted down and leaving the leftmost path top-down with nodes d_5, $b_{\underline{2}}$ and e_6, causing the inserted node c_3 to stay at a leaf when compared with $b_{\underline{2}}$. The algorithm incorrectly visits node d before node c, causing the distance to node e to be computed incorrectly as 6. Note that node d is extracted twice from the heap, both with original keys, but only the first time we visit d and consider paths with d as the second to last node on the paths (the test $u \notin visited$ prevents us from visiting d a second time).

It should be noted that if we skipped the test $u \notin visited$, the algorithm would compute the correct distances by revisiting nodes whenever a shorter distance to a node has been discovered. This is against the principle idea of Dijkstra's algorithm to only visit nodes once in increasing order of distance from the source, so that each edge is considered at most once. See the discussion in Section 7.3 on Figure 9 for further details.

4 Binary heaps with top-down insertions

In a binary heap with top-down insertions $\texttt{Insert}(x)$ creates a new empty leaf at position n, and items on the path from the root to the new leaf are compared with x top-down until the first node u is found with current key greater than or equal to the new key. The items on the path from u to the new leaf are sifted one level down and item x is inserted in node u. The ancestor at depth $d = 0, \ldots, \lfloor \log_2 n \rfloor$ is node $\lfloor n/2^{\lfloor \log_2 n \rfloor - d} \rfloor$. If keys are distinct and do not decrease, then top-down insertions and bottom-up insertions yield identical structures.

In the following we let key_{org} denote an original key and key_{cur} a current key. For a tree structure, where each node stores an item, we say that the tree satisfies the *decreased heap order* if and only if $u.key_{\mathrm{cur}} \leq v.key_{\mathrm{org}}$ for all ancestors u of a node v. Note that if decreased heap order is satisfied, then the item at the root of a tree satisfies the conditions to be returned by $\texttt{ExtractMin}$, and decreased heap order remains satisfied when current keys decrease.

During $\texttt{Insert}(x)$ a node v can only get one new ancestor, namely x. This happens when x is compared with an ancestor u of v with the result $x.key_{\mathrm{cur}} \leq u.key_{\mathrm{cur}}$. Since before the insertion decreased heap order ensures $u.key_{\mathrm{cur}} \leq v.key_{\mathrm{org}}$, we have $x.key_{\mathrm{cur}} \leq v.key_{\mathrm{org}}$ and the tree satisfies decreased heap order after the insertion.

An $\texttt{ExtractMin}$ operation returns the root of the tree. By the decreased heap order the item returned has current key less than or equal to all original keys in the tree. Before returning the answer, the last item x is moved to the root and sifted down, where x is swapped with the item at the child with smallest current key until the items of both children have current key $\geq x.key_{\mathrm{cur}}$. Whenever two siblings v and w are compared, say $v.key_{\mathrm{cur}} \leq w.key_{\mathrm{cur}}$, we have two cases. If x becomes the parent of v and w, since $x.key_{\mathrm{cur}} \leq v.key_{\mathrm{cur}} \leq w.key_{\mathrm{cur}}$, then $x.key_{\mathrm{cur}}$ is less than or equal to the original keys in all nodes below x, since either $v.key_{\mathrm{cur}}$ or $w.key_{\mathrm{cur}}$ was so before the operation. Otherwise, v becomes the parent of w, and x and all nodes in w's subtree get v as a new ancestor. Since $v.key_{\mathrm{cur}} \leq w.key_{\mathrm{cur}}$, then $v.key_{\mathrm{cur}}$ is less than or equal to all original keys in w's subtree, and $v.key_{\mathrm{cur}} \leq x.key_{\mathrm{cur}} \leq x.key_{\mathrm{org}}$. It follows that after $\texttt{ExtractMin}$ the tree still satisfies decreased heap order.

## 5 Existing heaps supporting decreasing keys

In the following we argue that the internal workings of several existing priority queues ensure decreased heap order to be maintained in the presence of decreasing keys.

Skew heaps and Leftist heaps

Skew heaps [18] and leftist heaps [4] support `Insert` and `ExtractMin` on a heap storing n items in time $O(\log n)$, where the time for skew heaps is amortized and for leftist heaps worst-case. Both data structures represent a priority queue by a (decreased) heap ordered binary tree, and support `ExtractMin` by removing the root and returning its item. All other structural changes consist of merging two root-to-leaf paths in two (decreased) heap ordered binary trees, and potentially swapping the left and right subtrees at nodes of the resulting tree. Swapping the left and right subtrees of a node does not change ancestor relationships, i.e., does not affect decreased heap order. To argue that the two data structures maintain decreased heap order, we only need to argue that merging two paths ensures that the resulting tree satisfies decreased heap order.

Normally the keys along the two paths would appear in increasing key order, but this is not necessarily the case when keys can decrease (in fact the current keys can appear in any order). Assume the nodes along the two root-to-leaf paths to be merged are $\langle u_0, u_1, u_2, \ldots \rangle$ and $\langle v_0, v_1, v_2, \ldots \rangle$, and the merging is performed top-down recursively as in Figure 4. If u_i ends up before v_j, i.e., u_i is a new ancestor of v_j, then there exists $j' \leq j$ where u_i and $v_{j'}$ have been compared and $u_i.key_{\mathrm{cur}} \leq v_{j'}.key_{\mathrm{cur}}$. Since by assumption $v_{j'}.key_{\mathrm{cur}} \leq v_j.key_{\mathrm{org}}$, it follows that u_i satisfies decreased heap order with v_j and all its descendants. It follows that the resulting tree after merging two root-to-leaf paths in two decreased heap ordered trees also satisfies decreased heap order.
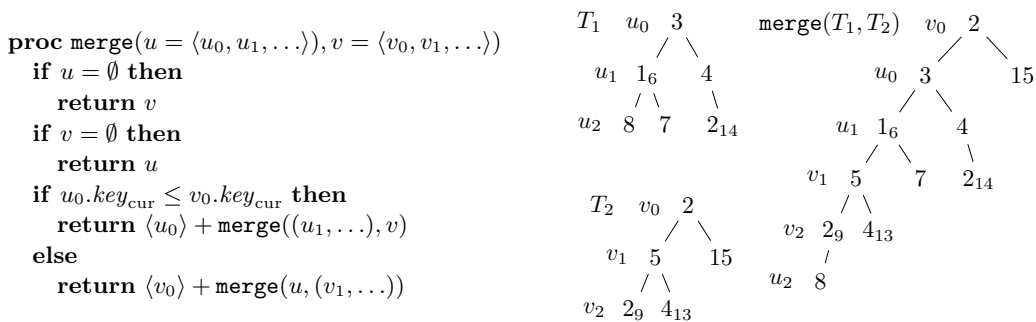
Figure 4 Top-down merging two paths u and v. In the example values are current keys and subscripts original keys (subscripts are omitted if current and original keys are equal).

Pairing heaps, Binomial queues, and Fibonacci heaps

Many priority queues represent a priority queue by one or more heap ordered trees of arbitrary degree. Pairing heaps [8], binomial queues [19], and Fibonacci heaps [9] are examples of such priority queues supporting `Insert` in amortized time $O(1)$ and `ExtractMin` in amortized time $O(\log n)$. Here we prove that these data structures maintain decreased heap order when used with decreasing keys. Pairing heaps only represent a priority queue by a single tree, whereas binomial queues and Fibonacci heaps maintain a forest, and the item removed by `ExtractMin` is the root with minimum current key, i.e., the returned item has current key less than or equal to all original keys in all trees. Nodes only get new ancestors when two roots are *linked*, that makes the root v with greatest current key a child of the root u with smallest current key. Since $u.key_{\mathrm{cur}} \leq v.key_{\mathrm{cur}}$ and any node w in the tree rooted at

v has $v.key_{\text{cur}} \leq w.key_{\text{org}}$, it follows that $u.key_{\text{cur}} \leq w.key_{\text{org}}$, i.e., the linked tree satisfies decreased heap order. It follows that the resulting heaps satisfy decreased heap order.

For Fibonacci heaps, the operations `DecreaseKey` replaces the key of an item by a smaller key. In our context this corresponds to lowering the original key. In the context of decreasing keys, it is important that `DecreaseKey` is implemented to always cut the edge from the node to its parent (without comparing with the current key of the parent, since that could have been decreased arbitrarily), and adds the node as a new root to the forest.

6   Post-order heap

Binary heaps with bottom-up insertions often benefit from the fact that insertions do not sift-up items far in the tree in practice. With top-down insertions this property cannot be exploited. In this section we consider the post-order heap by Harvey and Zatloukal [10]. In our experiments it appears to be a strong contender for an efficient implicit priority queue supporting decreasing keys, which is why we consider it in more detail in this section. Harvey and Zatloukal [10] did an experimental comparison of C# implementations of post-order heaps and binary-heaps with bottom-up insertions and found that post-order heaps had faster insertions but slower deletions.

A post-order heap consists of a forest of complete heap ordered binary trees, where all trees have distinct size, except for possibly the two trees of smallest size. Since a complete binary tree has size $2^i - 1$, for some i, the number of trees of each size corresponds to the digits in the skew binary number representation of n. Myers [14] proved that the set of tree sizes is unique for a given n. The trees are laid out consecutively in a single array H in decreasing size order, and each tree in post-order. For a subtree of size s with root $H[i]$, the subtree is stored in $H[i-s+1,i]$, the subtrees at the children have size $\lfloor s/2 \rfloor$, the right child is $H[i-1]$, and the left child is $H[i-1-\lfloor s/2 \rfloor]$. See Figure 5.
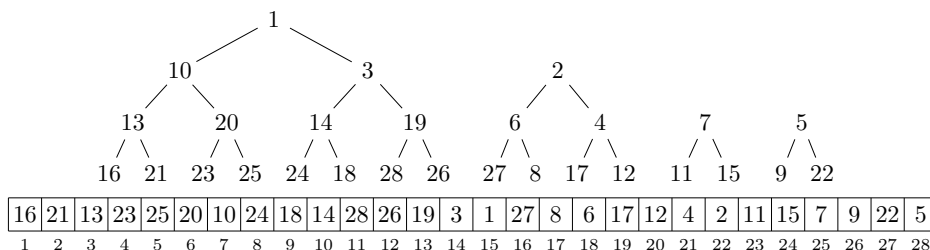

Figure 5 Top: A post-order heap storing 28 items in four trees of size 15, 7, 3 and 3. Bottom: The implicit post-order layout in a single array.

`Insert(x)` inserts the item x as the last item of H. If the last two trees had different size, x becomes a tree of size one. Otherwise, the last two trees of size s together with x are combined to a new tree of size $2s+1$ with x as the root, and we apply the sift-down operation `Heapify($|H|, 2s+1$)` (where the first argument is the node position, and the second argument is the size of the subtree). Except for node indexing, `Heapify` is implemented as for binary heaps. To support decreasing keys, it is important that `Heapify` is performed top-down. `ExtractMin()` identifies the root *min* with minimum (current) key to return, and removes the root x from the rightmost tree (causing the two subtrees to become new trees). If $x \neq min$, x replaces the root *min*, and x is sifted down using `Heapify`. Pseudo-code for the operations is given in Figure 6 (in [10] it is discussed how the list of tree sizes S of length $O(\log n)$ can be stored using only $O(\log n)$ bits). Since a post-order heap structurally is

```
proc Insert(x)                          proc ExtractMin()
  push(H, x)                              min = +∞
  if |S| ≥ 2 and S[|S|] = S[|S| − 1] then   i = |H|
    size = pop(S) + pop(S) + 1            for j = 1 to |S| do
    push(S, size)                           size = S[|S| − j + 1]
    Heapify(|H|, size)                      if H[i] < min then
  else                                        min = H[i]
    push(S, 1)                                i_min = i
                                              size_min = size
                                            i = i − size
proc Heapify(i, size)                     size = ⌊pop(S)/2⌋
  if size > 1 then                        if size > 0 then
    size = ⌊size/2⌋                          push(S, size)
    right = i − 1                           push(S, size)
    left = right − size                   x = pop(H)
    smallest = H[left] < H[right] ? left : right   if i_min < |H| then
    if H[smallest] < H[i] then              H[i_min] = x
      swap H[i] and H[smallest]             Heapify(i_min, size_min)
      Heapify(smallest, size)             return min
```

■ Figure 6 Post-order heap operations, where H stores the items and S is a list of tree sizes.

just a collection of binary heaps updated only using top-down `Heapify`, the discussion from Section 4 carries over to prove that post-order heaps support decreasing keys.

Harvey and Zatloukal [10] proved that post-order heaps support `Insert` in amortized time $O(1)$ and `ExtractMin` in amortized time $O(\log n)$. If we consider the worst-case number of comparisons for binary heaps with top-insertions, then `Insert` uses at most $\lfloor \log_2 n \rfloor$ whereas `ExtractMin` at most $2\lfloor \log_2 n \rfloor$, i.e., sorting using a binary heap requires at most $3\lfloor \log_2 n \rfloor$ comparisons. The worst-case number of comparisons for operations on post-order heaps is not competitive, since in the worst-case `Insert` performs `Heapify` on a tree containing all items, i.e., requiring at most $2\lfloor \log_2 n \rfloor$ comparisons, and `ExtractMin` first must find the root with minimum value, and then perform `Heapify` on this tree, requiring at most $3\lfloor \log_2 n \rfloor$ comparisons. Using these bounds for deriving a bound on sorting using a post-order heap gives us an upper bound of $5n \log_2 n$ comparisons. But for sorting we can derive a better bound. During the n insertions at most $n/2^h$ roots are created at height h each requiring $2h$ comparisons for a sift-down, causing a total of at most $\sum_{h=0}^{\infty} 2h \cdot n/2^h = O(n)$ comparisons for insertions. Furthermore, during the sequence of minimum extractions the average number of trees is $\frac{1}{2} \log_2 n + O(1)$, causing the total number of comparisons for finding the minimum roots to be at most $\frac{1}{2} n \log_2 n + O(n)$. Together with the upper bound of $2 \log_2 n$ comparisons for each `Heapify` caused by an `ExtractMin` gives a total bound of $2.5n \log_2 n + O(n)$ comparisons for sorting using a post-order heap. The advantage of post-order heaps over binary heaps with top-down insertions is studied in Section 7 (Figure 7).

7 Experimental evaluation

The previous sections state that many priority queue data structures work in the setting with decreasing keys. Any implementation of these data structures will also work if explicit keys can be removed and handled by implicit decreasing keys, e.g., implemented by a comparator

accessing the array *dist* in Dijkstra's algorithm. The worst-case analysis of these data structures carries over to the setting with decreasing keys and the worst-case running time analysis of Dijkstra's algorithm remains unchanged, though in practice the picture could be different. In particular items to be skipped (i.e., with decreased keys) in Dijkstra's algorithm can be extracted earlier when allowing decreasing keys.

We implemented various priority queues and Dijkstra's algorithm in Python 3.9 to have code as close as possible to pseudo code, and to focus on measuring counts that were hardware, language, and compiler independent.[4] A priority queue was implemented as a class with `extract_min` and `insert` methods to update the priority queue, and a method `empty` to test for emptiness. Items are compared using the `<` operator, i.e., using the `__lt__` method of the items. Finally, each priority queue has a method `validate` to check the structural integrity of its current content, e.g., a recursive traversal checking heap order, number of children, and balance conditions. The experimental evaluation was done on a Lenovo T460s laptop (Intel i7-6600U CPU, 12 GB RAM) running Python 3.9.4 under Windows 10.

The following priority queues were implemented: Skew heaps [18], leftist heaps [4], binomial queues [19], pairing heaps [8], post-order heaps [10], and binary heaps [20] with bottom-up and top-down insertions. Finally, we made a wrapper class around Python's builtin module `heapq` that is a `C` implementation[5] of binary heaps with bottom-up insertions. We did not implement Fibonacci heaps [9], since we do not consider dedicated `DecreaseKey` operations, and without `DecreaseKey` Fibonacci heaps are identical to binomial queues.

We considered four versions of binary heaps, where the first two do not support decreasing keys: BinaryHeap is a standard binary heap with bottom-up insertions and top-down heapify to sift-down the new root value during minimum extractions. BinaryHeapHeapifyBottomup improves typical performance by letting heapify first recursively pull up the child with smallest (current) key until an empty leaf is created, where the item from the last leaf is inserted and sifted up (our experiments confirm that module `heapq` implements this idea). The last two variants support decreasing keys. BinaryHeapTopdown supports insertions by performing comparisons top-down along the root-to-new-leaf path, until the first node is reached with greater or equal (current) key. The new item is inserted in this node, and all remaining nodes on the path to the last leaf are sifted one level down. BinaryHeapTopdownHeapify has a slightly naïve insertion implementation, where all items on the root-to-new-leaf path are sifted one level down, and the new item is inserted at the root and sifted down by `heapify`.

Experiments were parameterized by the priority queue class to be tested, to ensure identical testing overhead for the different classes. In our experiments we have measured the number of comparisons performed, various counts and running time. All priority queues were tested with exactly the same set of inputs. Data structures not supporting decreasing keys (Heapq, BinaryHeap and BinaryHeapHeapifyBottomup) are shown with dashed lines. In all plots Heapq and BinaryHeapHeapifyBottomup have identical curves, except for the time for sorting in Figure 7(*e*).

7.1   Correctness of implementation

To have some evidence for the correctness of our implementations, we performed two simple sorting tests: The first checks if `Insert` and `ExtractMin` work correctly if no keys decrease, and the second checks if decreasing keys are supported. The second stress test was in fact

---

[4] Python source code used for experiments and data visualized in figures is available at `https://www.cs.au.dk/~gerth/papers/fun22code.zip`

[5] `https://github.com/python/cpython/blob/master/Modules/_heapqmodule.c`

used to identify which priority queues supported decreasing keys, before knowing if they did so, directing the search for the arguments presented in Sections 3–6.

Sorting

Each priority queue implementation was used to sort various input sequences of n numbers, $1 \leq n \leq 1000$, with input being the increasing sequence $1, \ldots, n$, the decreasing sequence $n, \ldots, 1$, a uniform random permutation of $1, \ldots, n$, and n uniformly selected random integers from $1, \ldots, n$ (with possible repetitions). Only the random integer inputs can contain duplicates, where the expected number of distinct integers among n integers is $n(1 - (1 - 1/n)^n) \approx n(1 - 1/e) \approx 0.632n$. Each input was first inserted using n calls to `insert` followed by calls to `extract_min` until `empty` returned true. The output was checked against Python's built-in function `sorted`. After each update the priority queue's `validate` method was called to check for internal integrity.

Decrease key support

Each priority queue implementation was tested for the support of decreasing keys by performing the following experiment 1000 times with $n = 100$: Insert a random permutation of $1, \ldots, n$ using n calls to `Insert`, followed by n calls to `ExtractMin`. After each operation, with probability $1/2$ a random inserted item has its key decreased to zero. At the end it is checked if all items with unchanged key were reported in sorted order. As expected by the theory, all priority queues not supporting decreasing keys failed this test, whereas the others succeeded on all inputs

7.2 Sorting performance

We evaluated the performance of the different priority queues by using them to sort n integers for different n (powers of two, $n \leq 2^{20}$) and different input distributions: increasing sequences, decreasing sequences, uniformly permuted sequences, uniformly selected random integers from $1, \ldots, n$. Each input was run at least 3 times and until at least 0.2 seconds were passed. For random inputs 10 different inputs were generated and the average computed. The measured average number of comparisons for each input size and type is shown in Figure $7(a, b, c, d)$. In the plots we on the y-axis have number of comparisons performed divided by $n \log_2 n$, i.e., the theoretical asymptotic worst-case bound of sorting.

That Heapq is equivalent to the Python implementation BinaryHeapHeapifyBottomup follows from the plots, where the two priority queues achieve coinciding number of comparisons (it was checked that the number of comparisons performed were identical). Among the implicit constructions based on binary heaps we have a clear ordering (except for decreasing sequences) where BinaryHeapHeapifyBottomup (and Heapq) performs the fewest comparisons, followed by BinaryHeap, BinaryHeapTopdown and BinaryHeapTopdownHeapify, where only the last two support decreasing keys. In all cases the implicit PostOrderHeap achieves a better performance than the other implicit binary heaps supporting decreasing keys. In particular we see that PostOrderHeap performs about $\frac{1}{2} n \log_2 n$ fewer comparisons as Binary-HeapTopdown for random input, as expected by the discussion in Section 6. The only priority queues supporting decreasing keys that achieve significant better bounds on the number of comparisons are all pointer based (SkewHeap, LeftistHeap, PairingHeap and BinomialQueue). This leaves the post-order heap as a strong contender for an implicit priority queue supporting decreasing keys – at least with respect to comparisons.

With respect to running times the built-in `Heapq` consistently achieved the best time-wise performance, which is not surprising since it is implemented in `C`, whereas the other implementations are clearly penalized by the overhead of Python being interpreted. Running times for random input is shown in Figure 7(e). Interestingly, for large random inputs the running time of post-order heaps is only outmatched by the builtin `heapq`, although the overhead of using Python makes the results less conclusive. Since the motivation for studying priority queues with decreasing keys is to achieve space efficiency by avoiding storing keys with the items, the implicit post-order heap appears to be a good choice of data structure in this context.

7.3   Dijkstra's algorithm performance

We implemented a generic version of Dijkstra's algorithm for the single-source shortest path problem corresponding to `Dijkstra`$_4$ in Figure 1. This algorithm was selected since it allows to be executed both with a priority queue supporting decreasing keys (and no explicit keys in the items) and with a standard priority queue (with explicit keys in the items), and to study if allowing decreasing keys caused the number of comparisons performed to increase or decrease. We also tested what happens if we in addition to removing the explicit keys from the items also removed the visited array from the algorithm. This further reduces the space requirement for the algorithm, but breaks the $O(|E|\log|V|)$ running time guarantee.

As arguments the function takes the graph, the priority queue to be used, and two flags `use_visited` and `use_dist`. If `use_visited` is false, the algorithm will not check if extracted nodes from the priority queue have been visited before, i.e., we could save the space for having a bit-vector for the visited nodes, the cost being that we might revisit nodes (and relax their outgoing edges) multiple times (once for each shorter distance discovered to the node). If `use_dist` is true, $dist[v]$ is used as the key of v when comparing v, otherwise an item in the priority queue consists of a pair $\langle\text{distance}, \text{node}\rangle$.

As input we tested directed cliques (including self loops) with n nodes and n^2 edges, where $10 \leq n \leq 250$, with random integer weights from $1, \ldots, n$, and weights forcing the worst-case number of key decreases. In the latter case, the edge from node u to v has weight $\max(0, 2(v-u)-1)$, where $0 \leq u, v \leq n-1$, and node 0 is the source node, i.e., the path $0 \xrightarrow{1} 1 \xrightarrow{1} \cdots \xrightarrow{1} v-1 \xrightarrow{1} v$ is the shortest path to node v with distance v.

In the experiments we measured the number of comparisons performed by the priority queues, the number of nodes inserted into the priority queue, the number of nodes visited, and the number of edges relaxed. The results are summarized in Figures 8–10, where different combinations of `use_visited` and `use_dist` were tested. E.g., "+visited −dist" is when `use_visited` is true and `use_dist` is false, i.e., `Dijkstra`$_4$. The y-axis in Figures 8 and 9 is the measured cost divide by n, i.e., the average cost per node.

Since we only consider cliques, the number of edges relaxed is exactly n times the number of nodes visited. Furthermore, when a visited bit-vector is used, each node is visited and each edge relaxed exactly once, whereas if visited is not used, then each node inserted into a priority queue is also visited. For cliques with worst-case edge weights each edge (u, v), where $u < v$, will cause a new shorter distance of $2v - u - 1$ to v to be discovered, causing the number of insertions into the priority queue to be $\binom{n}{2}$, and the priority queue to grow to size $\Theta(n^2)$. When visited is not used, the total number of nodes visited is $\binom{n}{2} \approx n^2/2$ and the number of edges relaxed is $n\binom{n}{2} \approx n^3/2$. We therefore only show data for the number of nodes inserted into the priority queues for cliques with random weights in Figure 9 and omit data for the number of edges relaxed, nodes visited, and insertions for worst-case graphs.

For random edge weights, the probability is at most $1/k$ that the k'th edge considered

with node v as target decreases the distance to v (this follows by a simple backwards analysis argument), i.e., the expected number of times node v is inserted into the priority queue is at most $\sum_{k=1}^{n-1} \frac{1}{k} \approx \ln n$. This explains the nature of the curves in Figure 9(a), where the priority queue stores $\langle \text{distance}, \text{node} \rangle$ pairs. When using $dist[v]$ as key, it follows from Section 2 that the same applies to the priority queues supporting decreasing keys (non-dashed plots in Figure 9(b)).

Priority queues not supporting decreasing keys (Heapq, BinaryHeap and BinaryHeap-HeapifyBottomup) might return the wrong items when using the current $dist$ as keys. In combination with using a visited bit-vector, the algorithm might fail to compute the correct shortest distances. This was the case for random weights. Surprisingly, the algorithm worked correctly for worst-case weights. If the visited bit-vector is not used, then incorrect nodes returned by the priority queue will just be revisited later with a shorter distance, so the algorithm will still correctly find shortest paths – but there is no guarantee on the number of edge relaxes performed (except for an exponential upper bound that holds for any relax based approach). For our inputs the potential increase in number of revisits to nodes appears neglectable though, cf. Figure 9(b).

Finally, the number of comparisons performed in the priority queues is depicted in Figure 8. Interestingly, the number of comparisons performed appears to be a little bit lower if we use $dist$ as key, cf. (e) versus (g), and (f) versus (h). In Figure 10 we have plotted the number of comparisons performed when using $dist$ as the key relative to using $\langle \text{distance}, \text{node} \rangle$ pairs, for priority queues supporting decreasing keys and when using visited. The gain is largest for random cliques where the number of comparisons is reduced by typically at least 10 %. Interestingly, it appears that *using a priority queue supporting decreasing keys one can both save the space for storing explicit keys in the priority queue and reduce the number of comparisons performed.*

8 Conclusion

We have considered using priority queues supporting decreasing keys in Dijkstra's single source shortest path algorithm, motivated by the idea of saving space by omitting explicit keys for the items in the priority queue. Although standard binary heaps with bottom-up insertion fail to support decreasing keys, many other priority queues have been identified to do so, and in particular post-order heaps have been identified as a strong contender for a good alternative implicit priority queue in this context.

An open problem is to do a detailed experimental evaluation of the priority queues supporting decreasing keys in a low-level programming language like C. This is beyond the scope of this paper. Optimizing the running time of such implementations would require to carefully tune the code to take into account, e.g., caching, paging, branch mispredictions, and exploiting SIMD instructions [17]. Since the cache performance of binary heaps is known to be improvable by increasing the degree of the heap [13], one should also consider post-order heaps of higher degree.

───── **References** ─────

1   Gerth Stølting Brodal. A survey on priority queues. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 150–163. Springer, 2013. `doi: 10.1007/978-3-642-40273-9_11`.

2   Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In Rolf G. Karlsson and Andrzej Lingas, editors, *SWAT 88, 1st Scandinavian Workshop on Algorithm Theory, Halmstad, Sweden, July 5-8, 1988, Proceedings*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1988. `doi:10.1007/3-540-19487-8_1`.

3   Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: `http://mitpress.mit.edu/books/introduction-algorithms`.

4   Clark A. Crane. *Linear Lists and Priority Queues as Balanced Binary Trees*. PhD thesis, Department of Computer Science, Stanford University, 1972.

5   Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. `doi:10.1007/BF01386390`.

6   James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert Endre Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31(11):1343–1354, 1988. `doi:10.1145/50087.50096`.

7   Amr Elmasry, Claus Jensen, and Jyrki Katajainen. Two skew-binary numeral systems and one application. *Theory Comput. Syst.*, 50(1):185–211, 2012. `doi:10.1007/s00224-011-9357-0`.

8   Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986. `doi:10.1007/BF01840439`.

9   Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987. `doi:10.1145/28869.28874`.

10 Nicholas J. A. Harvey and Kevin C. Zatloukal. The post-order heap. In *Proceedings Third International Conference on Fun with Algorithms (FUN 2004)*, 2004. URL: `http://people.csail.mit.edu/nickh/Publications/PostOrderHeap/FUN04-PostOrderHeap.pdf`.

11 Vojtěch Jarník. O jistém problem minimálním. *Práce Moravské Pridovedecké Spolecnosti v Brně*, 4:57–63, 1930.

12 Irit Katriel and Ulrich Meyer. Elementary graph algorithms in external memory. In Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors, *Algorithms for Memory Hierarchies, Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*, pages 62–84. Springer, 2002. `doi:10.1007/3-540-36574-5_4`.

13 Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of heaps. *ACM J. Exp. Algorithmics*, 1:4, 1996. `doi:10.1145/235141.235145`.

14 Eugene W. Myers. An applicative random-access stack. *Inf. Process. Lett.*, 17(5):241–248, 1983. `doi:10.1016/0020-0190(83)90106-0`.

15 Thomas Porter and István Simon. Random insertion into a priority queue structure. *IEEE Trans. Software Eng.*, 1(3):292–298, 1975. `doi:10.1109/TSE.1975.6312854`.

16 R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957. `doi:10.1002/j.1538-7305.1957.tb01515.x`.

17 Peter Sanders. Fast priority queues for cached memory. *ACM J. Exp. Algorithmics*, 5:7, 2000. `doi:10.1145/351827.384249`.

18 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, 1986. `doi:10.1137/0215004`.

19 Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, 1978. `doi:10.1145/359460.359478`.

20 J. W. J. Williams. Algorithm 232 heapsort. *Commun. ACM*, 7(6):347–348, 1964. `doi:10.1145/512274.512284`.
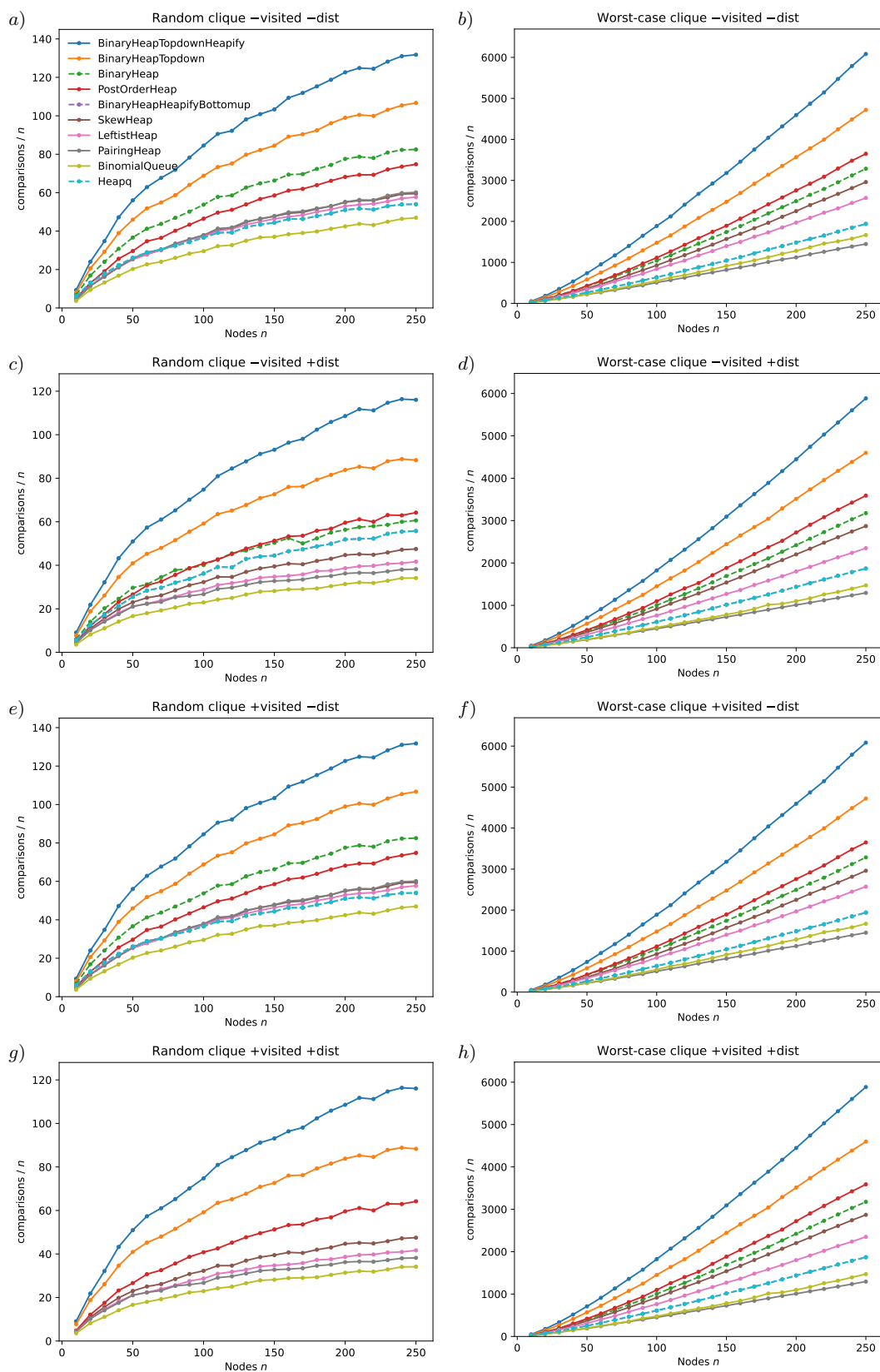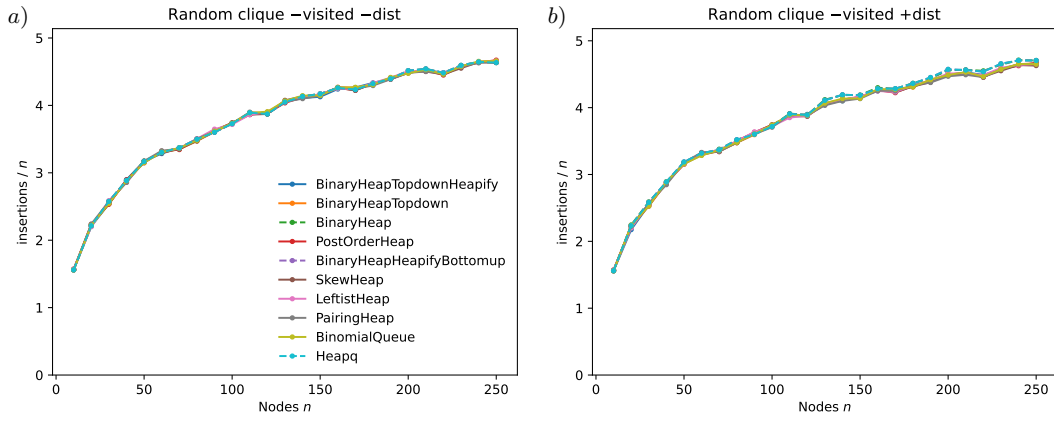
A   Plots
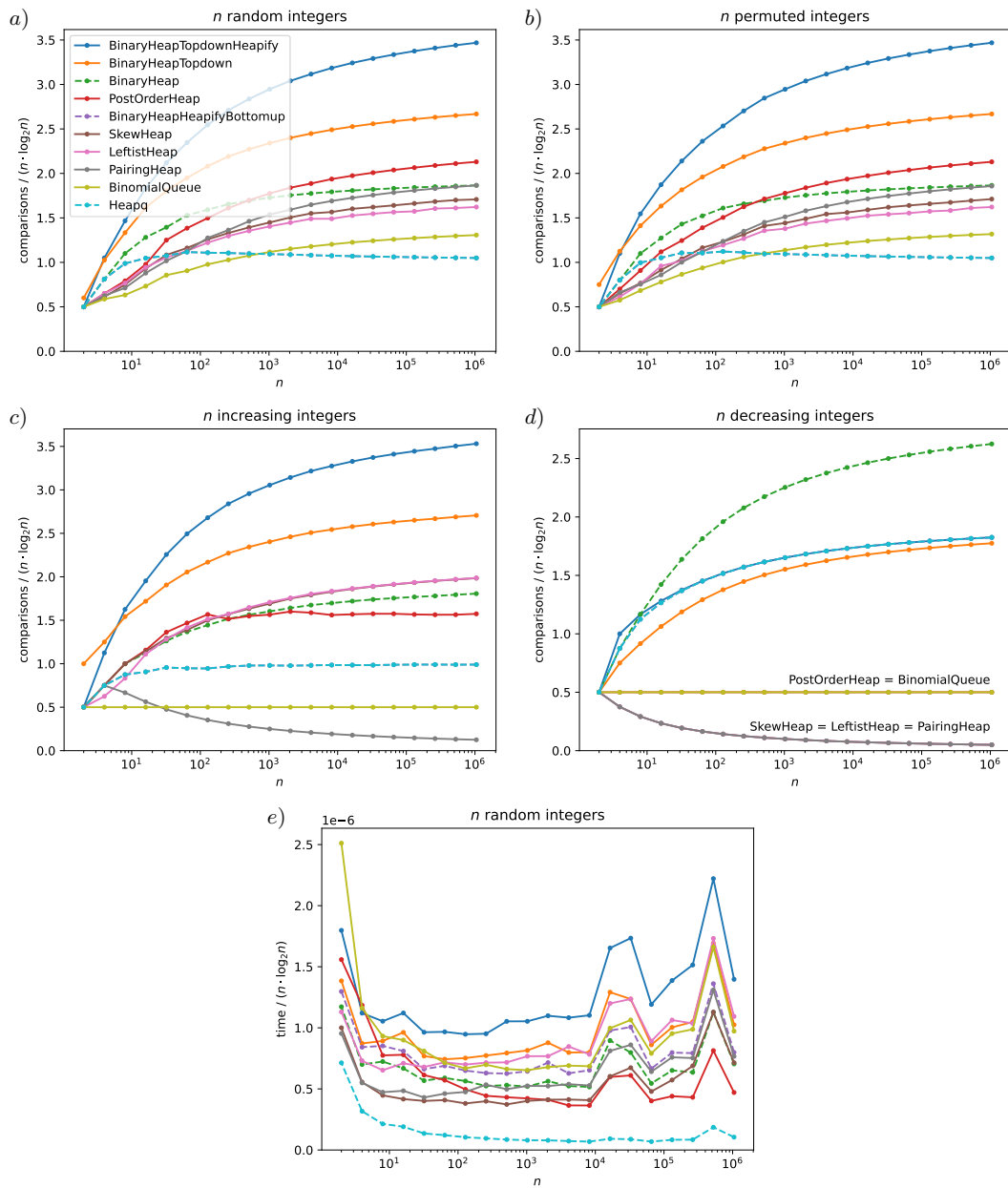
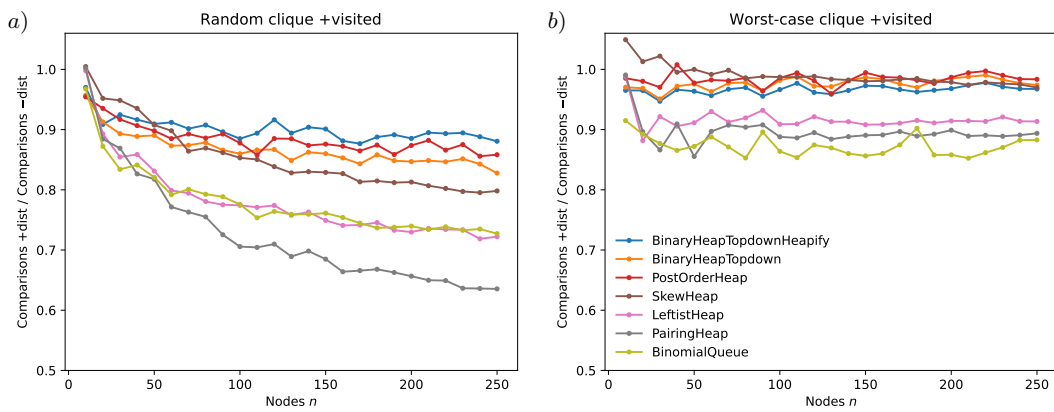**Figure 7** Sorting n integers using various priority queues.

Figure 8 Dijkstra, comparisons performed.

Figure 9 Dijkstra, priority queue insertions.

Figure 10 Dijkstra, number of comparisons performed when using *dist* as key divided by the number of comparisons performed when storing explicit keys in the priority queues.