Heaps and Priority Queues

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Why priority queues matter

Some algorithms need the next item in arrival order, but many need the item with the **best priority so far**. An event simulator processes the earliest event, a scheduler selects the most urgent job, and Dijkstra's and Prim's algorithms repeatedly select the smallest tentative key. A priority queue provides this access pattern without keeping every item fully sorted.

The central design question is not simply "Which heap is fastest?" Different representations make insertion, minimum extraction, key changes, and melding cheap in different combinations. The right choice follows from the workload and from the cost model of the machine.

2 Learning goals

After studying this note, you should be able to:

- specify the priority-queue operations and the invariants that implement them;
- derive array indices and operation costs for binary and d-ary heaps;
- explain and prove why bottom-up heap construction is linear;
- represent a binomial heap as a binary decomposition of its size and trace a meld;
- use a potential function to interpret amortised bounds;
- explain the structural ideas behind Fibonacci-heap bounds; and
- state the universe assumptions behind van Emde Boas structures.

3 Prerequisites

- Arrays and zero-based indexing.
- Complete rooted trees, height, and parent/child relationships.
- Asymptotic notation, logarithms, and geometric series.
- The distinction between worst-case, expected, and amortised bounds.

4 The priority-queue model

A min-priority queue stores **items** carrying comparable **keys**. The item and its key need not be the same object. Its usual interface is:

- `insert(Q, item, key)`;
- `minimum(Q)`, which observes but does not remove a minimum item;
- `extract_min(Q)`, which removes and returns one minimum item;
- `decrease_key(Q, handle, new_key)`; and
- sometimes `meld(Q1, Q2)`, which combines two queues.

Duplicate keys require a stated policy but do not invalidate a heap: any minimum-key item may be returned. A `decrease_key` operation needs a handle, index map, or node reference; finding an arbitrary item by value is a separate search problem.

A sorted linked list makes `minimum` and `extract_min` constant-time but needs linear-time insertion. An unsorted list reverses that trade-off. Heaps maintain only enough order to expose an extremum efficiently.

5 Binary heaps

5.1 Representation and invariants

A binary min-heap maintains two properties:

1. **Shape invariant:** it is a complete binary tree, with every level full except possibly the last, which is filled from left to right.
2. **Heap-order invariant:** for every non-root node v,

$$key(parent(v)) \leq key(v).$$

Completeness gives an implicit array representation with no pointers. For zero-based index i,

$$left(i) = 2i + 1, \qquad right(i) = 2i + 2, \qquad parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor.$$

The root is a minimum: following parent links from any node to the root never increases the key. The array is **not** globally sorted; the relation between nodes in different subtrees is generally unknown.

5.2 Restoring order locally

Insertion first preserves the shape invariant by appending a leaf. Only the new leaf-to-root path can violate heap order, so `sift_up` repairs exactly that path.

```
insert(A, x):
    append x to A
    i = len(A) - 1
    while i > 0:
        p = (i - 1) // 2
        if key(A[p]) <= key(A[i]): break
        swap A[p], A[i]
        i = p
```

For `extract_min`, save the root, move the last item to index 0, and remove the last cell. Only a root-to-leaf path can now violate heap order. At each step, swap with the smaller child; choosing the larger child could leave the smaller child below an invalid parent.

```
extract_min(A):
    require len(A) > 0
    answer = A[0]
    A[0] = A[-1]
    remove the last array cell
    i = 0
    while left(i) < len(A):
        c = left(i)
        if right(i) < len(A) and key(A[right(i)]) < key(A[c]):
            c = right(i)
        if key(A[i]) <= key(A[c]): break
        swap A[i], A[c]
        i = c
    return answer
```

Each repair crosses at most the tree height $\lfloor \log_2 n \rfloor$, so insertion, extraction, and decrease-key take $O(\log n)$; `minimum` takes $O(1)$.

6 d-ary heaps

A d-ary heap, for an integer $d \geq 2$, gives each node at most d children while retaining completeness and array storage. With zero-based indexing,

$$parent(i) = \left\lfloor \frac{i-1}{d} \right\rfloor,$$

and the possible children of i occupy indices $di + 1, di + 2, \ldots, di + d$, limited by the array length. Some course tasks call the same structure a **k-ary heap**.

The height is $\Theta(\log_d n)$. `sift_up` compares with one parent per level, so insertion and decrease-key use $O(\log_d n)$ comparisons. A downward step must inspect up to d children to identify the smallest, giving `extract_min` $O(d \log_d n)$ comparisons in the comparison model.

Increasing d therefore trades fewer levels for more work per downward level. A flatter layout can also improve locality, while a very large branching factor may waste comparisons. Report both comparisons

and elapsed time when evaluating a d-ary heap: hardware behaviour cannot be inferred from asymptotic height alone.

7 Building a heap bottom-up

Repeated insertion constructs a heap in $O(n \log n)$, but it does unnecessary upward work. Leaves already satisfy heap order. Starting at the last internal node and sifting each internal node downward makes every processed subtree a heap:

```
build_heap(A, d):
    if len(A) < 2: return
    for i = floor((len(A) - 2) / d) down to 0:
        sift_down(A, i, d)
```

Invariant. Just before index i is processed, all subtrees rooted at indices greater than i are heaps. The children of i have larger indices, so after `sift_down` the subtree at i is also a heap. At termination the root's subtree is the whole heap.

The operation is linear because most nodes are near the leaves. For a binary heap, at most about $n/2^{h+1}$ nodes have height h, so

$$\sum_{h \geq 0} O\Big(\frac{n}{2^{h+1}} h\Big) = O(n).$$

The same geometric argument gives $O(n)$ for every fixed $d \geq 2$, even after counting up to d child inspections per downward level. Multiplying n calls by the worst cost of one call would give a valid but unnecessarily loose $O(n \log n)$ bound.

8 Binomial heaps and melding

Binary and d-ary heaps keep all items in one complete tree, which is excellent for locality but makes a general meld expensive. Meldable heaps instead use a forest whose trees can be combined structurally. Binomial heaps make that structure explicit; Fibonacci heaps later postpone some of the same combining work.

A **binomial tree** is defined recursively. B_0 is a single node, and B_k is formed by linking two copies of B_{k-1}, making one root a child of the other. Consequently, B_k has 2^k nodes, height k, and root degree k. Its root's children are roots of $B_{k-1}, B_{k-2}, \ldots, B_0$.

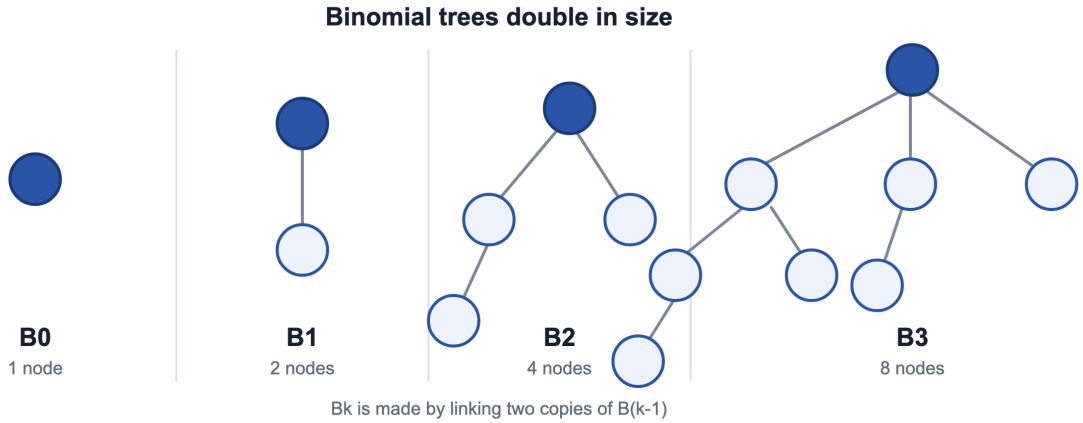

Figure 1: The binomial-tree family: each step links two trees of the preceding degree, producing trees with 1, 2, 4, and 8 nodes.

A **binomial heap** is a forest of heap-ordered binomial trees with at most one tree of each degree. Roots are stored in increasing degree order. Tree B_k is present exactly when bit k of the heap size is 1, so the forest is a structural binary representation of n.

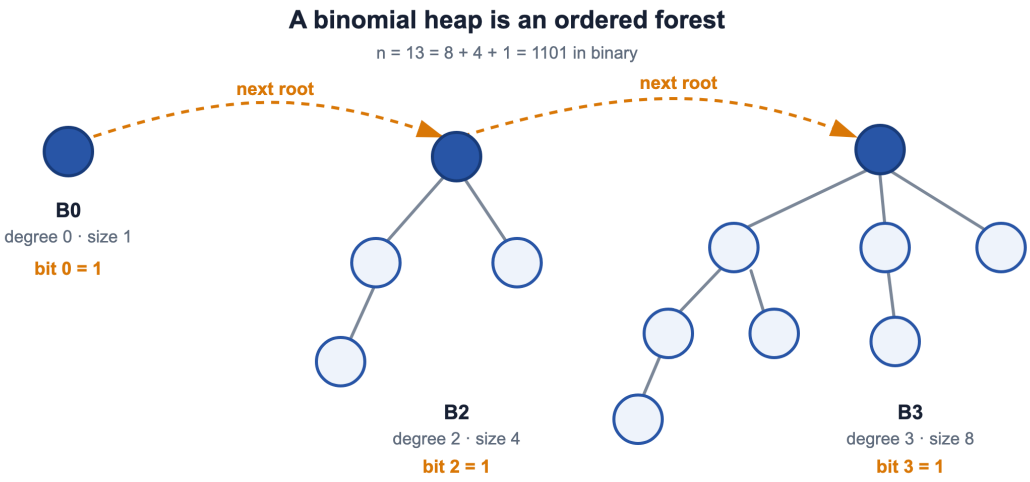

Figure 2: A 13-node binomial heap contains B_0, B_2, and B_3, corresponding to the set bits of $13 = 1101_2$.

To **link** two degree-k trees, compare their roots and make the larger-key root a child of the smaller-key root. Heap order is preserved, and the result is a B_{k+1}.

To **meld** two heaps, first merge their root lists by degree. Equal-degree trees are linked, possibly creating another collision at the next degree. This is carry propagation in binary addition.

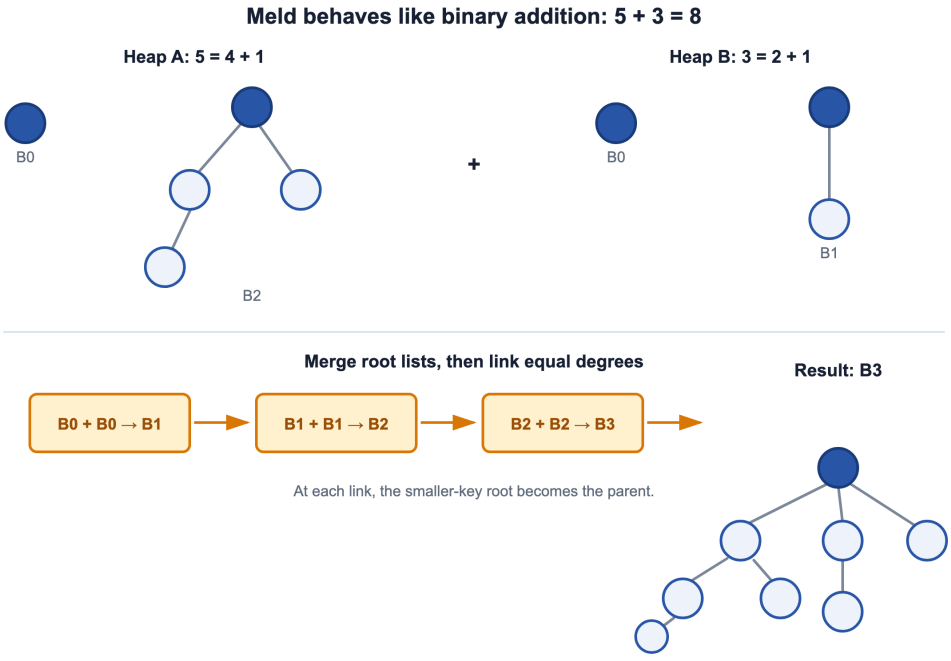

Figure 3: Melding heaps of sizes 5 and 3: equal-degree trees link in a carry chain, leaving one B_3 with 8 nodes.

There are $O(\log n)$ possible degrees, so meld takes $O(\log n)$. Insertion is a meld with a one-node B_0. To extract a minimum, scan the roots, remove the minimum-root tree, reverse its descending-degree child list into an increasing-degree heap, and meld it back. Decrease-key swaps a decreased item upward through parent links until heap order is restored. These operations take $O(\log n)$; `minimum` is $O(\log n)$ unless a maintained minimum pointer makes it $O(1)$.

9 Amortised analysis from first principles

Worst-case analysis prices one operation in isolation. **Amortised analysis** bounds the total cost of every sequence of operations; it is not an average over random inputs.

Let c_i be the actual cost of operation i, and let a nonnegative potential Φ_i measure stored work after it. Define

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}.$$

The potential changes telescope:

$$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} \hat{c}_i - \Phi_m + \Phi_0.$$

If $\Phi_0 = 0$ and potentials stay nonnegative, total actual cost is at most total amortised cost.

For a binary counter, let the cost of increment be the number of flipped bits and choose $\Phi =$ number of 1-bits. If an increment clears t trailing ones and sets one zero, its actual cost is $t + 1$ and its potential change is $1 - t$. Its amortised cost is therefore 2, even though an individual increment can flip $\Theta(\log n)$ bits.

The same carry idea explains lazy binomial heaps: insertion may append a B_0 in constant actual time, leaving extra roots as potential. A later consolidation pays for links by reducing the number of roots.

10 Fibonacci heaps

A Fibonacci heap pushes this laziness further. It stores a heap-ordered forest, a circular root list, and a pointer to a minimum root.

- `insert` adds a singleton root.
- `meld` concatenates two root lists and keeps the smaller minimum pointer.
- `extract_min` removes the minimum root, promotes its children to roots, and consolidates roots of equal degree.
- `decrease_key` cuts a node whose new key violates parent order and may perform **cascading cuts** up the ancestor chain.

A non-root node is marked after it loses its first child. If it loses another, it is cut; this may expose a marked parent and continue the cascade. With t roots and m marked nodes, the standard potential is

$$\Phi = t + 2m.$$

Suppose one `decrease_key` cuts k nodes. The first cut creates a root and clears that node's mark if it had one. Every later cut removes a marked node from its parent, creates another root, and clears the mark; the cascade may finish by marking one previously unmarked ancestor. Therefore $\Delta t = k$ and $\Delta m \le 2 - k$, so

$$\Delta\Phi = \Delta t + 2\Delta m \le 4 - k.$$

This potential change pays for all but constant amortised work in the $\Theta(k)$-cost cascade. Thus insertion, meld, minimum, and decrease-key have $O(1)$ amortised cost.

Extraction must consolidate degrees. Why are there only $O(\log n)$ degrees after arbitrary cuts? Order a degree-k node's children by when they were linked to it. Its i-th child had degree at least $i - 1$ when linked and can lose at most one child while remaining below its parent, so its current degree is at least $i - 2$.

If S_k is the minimum possible size of a degree-k subtree, then $S_0 = 1$, $S_1 = 2$, and

$$S_k \geq 2 + \sum_{j=0}^{k-2} S_j \geq F_{k+2} \qquad (k \geq 2),$$

where $F_0 = 0$, $F_1 = 1$, and $F_{i+2} = F_{i+1} + F_i$.

Thus a degree-k node has $\Omega(\varphi^k)$ descendants, where $\varphi = (1 + \sqrt{5})/2$, so the maximum degree is $D = O(\log n)$.

An `extract_min` may begin with many roots. Its actual work is $O(t + D)$: process the root list, promote at most D children, and consolidate. Afterwards at most one root of each degree remains, so $t' \leq D + 1$. Promoted children become unmarked roots, hence $\Delta m \leq 0$, and

$$\Delta\Phi = (t' - t) + 2\Delta m \leq D + 1 - t.$$

The $-t$ term pays for scanning a long root list, leaving $O(D) = O(\log n)$ amortised work.

These are theoretical improvements for workloads with many decrease-key operations. Pointer-rich layouts, memory allocation, and larger constants often make binary, d-ary, or pairing heaps faster in real programs.

11 Enrichment: van Emde Boas structures

A van Emde Boas (vEB) structure changes the model: keys must be integers in a known universe $\{0, \ldots, U - 1\}$. For the clean recursive description, round U up so repeated square roots divide the key into equally sized high and low parts.

The structure stores minimum and maximum keys, divides the universe into $\sqrt{U}$ clusters of size $\sqrt{U}$, and recursively records which clusters are nonempty in a summary structure. Successor, predecessor, membership, insertion, and deletion recurse into a universe of size $\sqrt{U}$:

$$T(U) = T(\sqrt{U}) + O(1) = O(\log \log U).$$

The classic eager representation uses $O(U)$ space, so it is attractive only when the integer universe and density justify it. Sparse and practical variants require additional ideas. The $O(\log \log U)$ bound is not a comparison-based priority-queue bound; it relies on word operations and bounded integer keys.

12 Worked trace

Insert keys `7, 3, 9, 1` into a binary min-heap:

1. Append 7: `[7]`.
2. Append 3 and swap it with 7: `[3, 7]`.
3. Append 9; its parent 3 is already smaller: `[3, 7, 9]`.
4. Append 1: `[3, 7, 9, 1]`. Swap with 7, then with 3: `[1, 3, 9, 7]`.

Now extract the minimum. Save 1, move the last item 7 to the root, and shrink the array to `[7, 3, 9]`. Of children 3 and 9, choose 3 and swap, producing `[3, 7, 9]`. Shape and heap order are restored. Notice that 7 and 9 need not be sorted relative to one another.

13 Complexity and trade-offs

Structure	Minimum	Insert	Extract min	Decrease key	Meld
Binary heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$ by rebuild
d-ary heap	$O(1)$	$O(\log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(n)$ by rebuild
Binomial heap	$O(\log n)$, or $O(1)$ tracked	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Fibonacci heap, amortised	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$
van Emde Boas	$O(1)$ stored	$O(\log \log U)$	$O(\log \log U)$	model-dependent	not primary operation

The table hides important assumptions: Fibonacci bounds are amortised, vEB bounds use bounded integers, and decrease-key presumes a direct handle to the item.

14 Connections

- Arrays and tree indexing make binary and d-ary heaps compact.
- Binary representation explains binomial trees and melding.
- Amortised analysis also explains dynamic arrays and union-find sequences.
- Dijkstra's and Prim's algorithms use `extract_min` and `decrease_key`; their bounds depend on the chosen heap.
- Integer-universe structures connect comparison-based data structures to word-RAM algorithms.

15 Common mistakes

- Treating the heap array as fully sorted.
- Restoring heap order with the larger rather than smaller child in a min-heap.
- Forgetting to update an external item-to-index map during swaps.
- Claiming bottom-up construction is n times $O(\log n)$ without summing actual node heights.
- Assuming a larger d is always faster without counting child comparisons.
- Calling an amortised bound a per-operation worst-case guarantee.
- Applying a vEB bound without stating the integer universe and space cost.

16 Self-check

1. Derive the zero-based parent and child formulas for a 4-ary heap.
2. Why must `sift_down` choose the minimum child?
3. Give the invariant that proves bottom-up construction correct.
4. Explain why heap sizes 13 and 14 have different binomial-tree forests.
5. Use a binary counter to explain the role of potential.
6. What prevents a Fibonacci-heap node from losing arbitrarily many children while remaining below its parent?
7. When can a theoretically stronger heap lose to a binary heap in practice?

17 Revision summary

- A heap maintains partial order sufficient to expose an extremum, not total order.
- Completeness enables pointer-free binary and d-ary heaps.
- Bottom-up construction is linear because most nodes can move only a short distance.
- Binomial heaps turn melding into binary carry propagation.
- Potential functions transfer saved work across an operation sequence.

- Fibonacci heaps defer structural work and control degree through cascading cuts.
- van Emde Boas bounds depend on a bounded integer universe and different machine assumptions.

18 Implementations and hands-on exploration

- Python's `heapq` exposes an array-based binary min-heap and documents patterns for stable priorities and lazy deletion. It has no handle-based `decrease_key` or efficient meld, so graph algorithms commonly insert a new pair and ignore stale entries when popped.
- Boost.Heap provides C++ d-ary, binomial, Fibonacci, pairing, and skew heaps, including mutable handles where supported. The theoretical distinctions in this note are visible in one API, but handle validity and the chosen mutability policy must be checked carefully.
- Rust's `BinaryHeap` is a standard-library binary max-heap; wrap keys in `Reverse` for minimum-first order. It can move every item from another heap with `append`, but the API promises no efficient structural meld and provides no handles for arbitrary decrease-key.
- Java's `PriorityQueue` is an unbounded priority queue with logarithmic insertion and removal at the head. Its iterator is not sorted, and changing fields used by the comparator while an item remains queued breaks the intended ordering.

18.1 Small experiments

1. Generate the same random arrays at increasing sizes. Time Python `heapq.heapify(data)` against inserting every item with `heappush`, then pop everything to verify that both outputs are sorted and plot time divided by n.
2. Implement `sift_down` for $d \in \{2, 4, 8, 16\}$. On identical heaps, count key comparisons and elapsed time for a full sequence of extractions; explain why the d with the fewest levels need not perform the fewest comparisons.

19 Sources and further study

- Heaps lecture deck, Jaak Vilo.
- (Cormen et al. 2022), Chapters 6, 19, and 20.

References

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. 4th ed. MIT Press.