Fibonacci Heap — Notes

Operations, Implementation Intuition, and Amortized Analysis

(Prepared for revision)

August 9, 2025

Overview

A *Fibonacci heap* is a data structure for a priority queue. It stores a collection (a forest) of heap-ordered rooted trees. The key ideas are:

- Make **insert** and **decrease-key** very cheap (constant actual time) by postponing work.
- Do heavier work in **extract-min** (via `consolidate`). Amortized analysis shows operations are efficient.

Node fields. Each node stores:

$$(\text{key},\ \text{degree},\ \text{mark},\ \text{parent},\ \text{child},\ \text{left},\ \text{right})$$

`left/right` form circular doubly-linked lists (used for root lists and child lists). `mark` is true iff the node has lost a child since it became a child of its current parent.

Heap fields. The heap stores:

$$\texttt{minimum} \quad \text{(pointer to a root with minimum key)}, \qquad n \quad \text{(total nodes)}.$$

---

1 High-level operations (what and why)

For each operation below, I give the *purpose*, a short *how/why* and the *expected amortized cost.*

Make-Heap

Purpose: create an empty heap.
How: set `minimum` = NULL, $n = 0$.
Amortized cost: $O(1)$.

Insert(H, x)

Purpose: add a new key/node.
How/Why: create a one-node tree and splice it into the root list. Update `minimum` if needed. Very cheap because we do no restructuring.
Amortized cost: $O(1)$.

Minimum(H)

Purpose: read the minimum key.
How: return `minimum`.
Cost: $O(1)$ exact.

Union / Meld(H1, H2)

Purpose: unify two heaps in $O(1)$.
How/Why: concatenate circular root lists and set `minimum` to the smaller of the two minima. No consolidation now.
Amortized cost: $O(1)$.

Extract-Min(H)

Purpose: remove and return the minimum key.
How/Why: remove root $z =$ `minimum`; move all of z's children into the root list (making them roots and clearing their parent pointers); then call `consolidate()` which links roots of equal degree until at most one root per degree remains. This reorganizes the forest so the number of roots is $O(\log n)$.
Amortized cost: $O(\log n)$.

Decrease-Key(H, x, newKey)

Purpose: reduce a node's key (maintain heap-order).
How/Why: if the decreased key still respects parent's key, nothing else needed. Otherwise, `cut` the node from its parent and add it to the root list. If the parent was previously unmarked, mark it; otherwise (if it was marked) cut it too and continue upward (a *cascading cut*). Marking defers repeated cuts, storing potential that pays for future cuts.
Amortized cost: $O(1)$.

Delete(H, x)

Purpose: remove arbitrary node.
How/Why: do `decrease-key`$(x, -\infty)$ (or smaller than current minimum) then `extract-min`.
Amortized cost: $O(\log n)$.

---

2 Key helper routines: pseudocode

Below is compact pseudocode for the core helpers. You can map this directly to your implementation.

Link

When two root-trees of the same degree need to be combined, make one root the child of the other.

```
Link(y, x):    // assumes x.key <= y.key
    remove y from root list
    add y to x.child list
    y.parent = x
    x.degree = x.degree + 1
    y.mark = false
```

Cut

Remove a child x from its parent y and add x to the root list.

```
Cut(H, x, y):
    remove x from y.child list
    y.degree = y.degree - 1
    if y.child was x: update y.child to some other child or NULL
    add x to root list
    x.parent = NULL
    x.mark = false
```

Cascading-Cut

```
Cascading-Cut(H, y):
    z = y.parent
    if z != NULL:
        if y.mark == false:
            y.mark = true
        else:
            Cut(H, y, z)
            Cascading-Cut(H, z)
```

Consolidate

```
Consolidate(H):
    A = array[0 .. Dmax] initially all NULL
    // snapshot root list into a list ‘roots‘ (because list is mutated)
    for w in roots:
        x = w
        d = x.degree
        while A[d] != NULL:
            y = A[d]
            if x.key > y.key: swap(x,y)
            Link(y, x)     // now x has degree d+1
            A[d] = NULL
            d = d + 1
        A[d] = x
    // rebuild root list from non-NULL entries in A and set H.minimum
```

Note on array size. Use an upper bound $D_{\max} = O(\log n)$. A safe simple choice is $D_{\max} = \lfloor \log_2 n \rfloor + 2$. The tighter bound uses the golden ratio φ, but $\log_2$ is simpler and safe.

---

3 Amortized analysis

We use the standard potential-method argument. Define:

$$\Phi(H) = t(H) + 2 \cdot m(H),$$

where

- $t(H)$ is the number of trees (roots) in the root list,
- $m(H)$ is the number of marked (non-root) nodes.

Potential is always nonnegative. For an operation, amortized cost = actual cost $+\Delta\Phi$.

Degree bound (why $D(n) = O(\log n)$)

Let size(x) be the number of nodes in the subtree rooted at x. If x has degree k, we can show by induction that

$$\text{size}(x) \geq F_{k+2},$$

where F_i is the i-th Fibonacci number ($F_0 = 0, F_1 = 1$). Sketch:

- size$(0) = 1$.
- When a node gets children, those children must have had distinct degrees at least $0, 1, \ldots, k-1$ (because of the way linking and cuts operate), giving

$$\text{size}(k) \geq 1 + \sum_{i=0}^{k-1} \text{size}(i).$$

- This recurrence yields size$(k) \geq F_{k+2}$.

Since $F_t \geq \varphi^{t-2}$ (where $\varphi = (1+\sqrt{5})/2$), we get

$$n \geq \text{size}(x) \geq F_{k+2} \geq \varphi^k \quad \Rightarrow \quad k \leq \log_\varphi n = O(\log n).$$

Thus every node degree is $O(\log n)$ and the number of possible degrees is $O(\log n)$.

Make-Heap

Actual cost $= O(1)$. $\Delta\Phi = 0$. $\Rightarrow$ amortized $O(1)$.

Insert

Actual cost: $O(1)$ (splice a new one-node tree into root list).

$$\Delta t = +1, \qquad \Delta m = 0 \quad \Rightarrow \quad \Delta\Phi = +1.$$

So amortized cost $= O(1) + 1 = O(1)$.

Union / Meld

Actual cost: $O(1)$ (concatenate two root lists). Potential change bounded by $O(1)$. Amortized $O(1)$.

Extract-Min

Let z be the removed minimum with degree $d = \deg(z)$.

Actual work:

- Move d children of z to the root list: $O(d)$ pointer updates.
- Remove z from the root list: $O(1)$.
- `consolidate`: we process each root and perform $\leq$ one `link` per degree-collision. Each `link` is $O(1)$. The number of links is at most the number of roots before consolidation, and final number of roots after consolidation is at most $D(n) + 1 = O(\log n)$.

So actual cost $= O(d + \#\text{links})$, and $\#\text{links} = O(t + d)$ in the naive worst accounting, but we will combine with potential change.

Potential change (upper bound). Let t and m be the numbers of roots and marked nodes before extraction. After moving children and removing z, the number of roots becomes at most $t - 1 + d$. After consolidation the number of roots t' satisfies $t' \leq D(n) + 1 = O(\log n)$. Also the number of marked nodes $m' \leq m$ (moving children to root clears their marks). Hence

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

Combine actual + potential. Although the above bound looks to depend on t, the actual work included many link operations that reduce the number of roots; the net amortized cost per extract-min simplifies to:

$$\text{amortized cost} = O(d + \#\text{links}) + \Delta\Phi = O(D(n)) = O(\log n).$$

Intuition: earlier cheap operations (insert / decrease-key) created many roots and possibly marks; the potential stored pays for the costly consolidation. Using the degree bound $D(n) = O(\log n)$ gives the final amortized $O(\log n)$.

Conclusion: `extract-min` is $O(\log n)$ amortized.

Decrease-Key

Let x be the node whose key is decreased to k.

- If x is a root or still respects heap-order (key $\geq$ parent's key), then actual cost is constant and potential change is small $\Rightarrow$ amortized $O(1)$.
- If x violates heap-order (i.e., $x.key < parent.key$), we `cut` x and add it to the root list. That is $O(1)$ actual cost. Then:
  - If the parent y was unmarked, we set $y.mark = \text{true}$ (no further cuts). This increases m by 1 and t by 1, so $\Delta\Phi = +1 + 2 \cdot 1 = +3$: a constant increase which pays for the small actual cost.
  - If y was already marked, we `cut` y as well and continue upwards (cascading cut). Each additional cut is $O(1)$ actual. However each cut (after the first) reduces the number of marked nodes by 1 (since a marked node gets cut and becomes a root and becomes unmarked), and increases number of roots by 1. Thus the potential decrease covers the actual cost of further cuts.

Overall, summing actual work plus potential change shows `decrease-key` has $O(1)$ amortized cost.

Informal accounting (sketch). If cascading cut does c cuts:

$$\text{actual} = O(c).$$

Potential change: roots $+c$, marked nodes $-(c - 1)$ (first parent changed from unmarked to marked or vice versa; bookkeeping depends on exact starting state). So

$$\Delta\Phi = c + 2 \cdot (-(c - 1)) = c - 2c + 2 = 2 - c.$$

Thus amortized cost $= O(c) + (2 - c) = O(1)$. (Precise bookkeeping in CLRS yields a small constant upper bound.)

Delete

$$\text{delete}(x) = \text{decrease-key}(x, -\infty) + \text{extract-min} \quad \Rightarrow \quad O(1) + O(\log n) = O(\log n).$$

Summary table (amortized)

Operation	Amortized cost
Make-Heap	$O(1)$
Insert	$O(1)$
Minimum	$O(1)$
Union (meld)	$O(1)$
Extract-Min	$O(\log n)$
Decrease-Key	$O(1)$
Delete	$O(\log n)$

Practical implementation notes (quick checklist)

- When iterating the root list while you will modify it, **snapshot** the roots into a vector first (consolidation needs that).
- When moving children to the root list during extract-min, avoid inserting into the soon-to-be-removed `minimum` node. Use a safe anchor (e.g., keep pointer to a different root or insert while z is still present).
- Always isolate nodes when storing them into the degree array (set left=right=node) to avoid stale links.
- Be careful about **handles**: if you store raw node pointers externally, extracting and then deleting nodes will create dangling pointers. Either return ownership to caller or use safe handles/IDs.
- Choose the degree-array size conservatively: $\lfloor \log_2 n \rfloor + 2$ is safe.

References / further reading

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, chapter on heaps (Fibonacci heap section).
- Original paper by Michael L. Fredman and Robert E. Tarjan (1987) introducing Fibonacci heaps.

4 Step-by-step consolidation example (TikZ)

Below we trace `extract-min` followed by `consolidate` for the inserted keys $\{7, 3, 17, 24, 10, 2, 8, 15\}$. After insertion the root list printed (as in your run) was:

$$2,\ 3,\ 7,\ 17,\ 24,\ 10,\ 8,\ 15$$

Removing the minimum 2 leaves the roots

$$3,\ 7,\ 17,\ 24,\ 10,\ 8,\ 15$$

We process the roots in that order during `consolidate`. The diagrams below show how collisions of equal degrees are resolved by repeated `link` operations.

Notes on the trace.

- We processed roots in the order shown: $3, 7, 17, 24, 10, 8, 15$. Every time two roots have the same degree we `link` them: the root with the smaller key becomes the parent.
- The consolidation algorithm uses an array A indexed by degree. The snapshot after processing all roots would be (degree $\mapsto$ root):

$$A[0] = 15, \quad A[1] = 8, \quad A[2] = 3,$$

  which yields the final distinct-degree root list $(15,\ 8,\ 3)$ (order may be rebuilt differently but the trees are the same). Minimum is the root with smallest key, here 3.
- This trace shows the typical *chain reaction* where linking two trees can cause a new degree-collision and another link, continuing until an empty $A[d]$ slot is found.

5 Consolidation trace with degree-array state

Figure 1: *
Step 0 (start): Root list after extracting minimum (2). All roots are degree 0. Process order: 3,7,17,24,10,8,15.

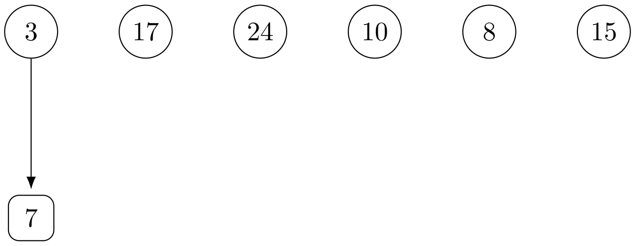


Figure 2: *
Step 1: Processed root `3` then `7`. Since both had degree 0, they collided: `link(7,3)` makes 7 a child of 3. Now deg(3) = 1.

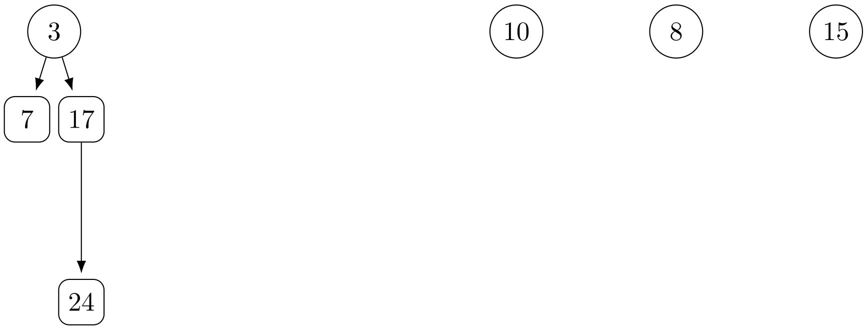


Figure 3: *
Step 2: Processed root `17` then `24`. `link(24,17)` makes 24 a child of 17. Then degrees collide with the tree rooted at 3 (degree 1), so `link(17,3)` attaches 17 (with its child 24) under 3. Now deg(3) = 2.

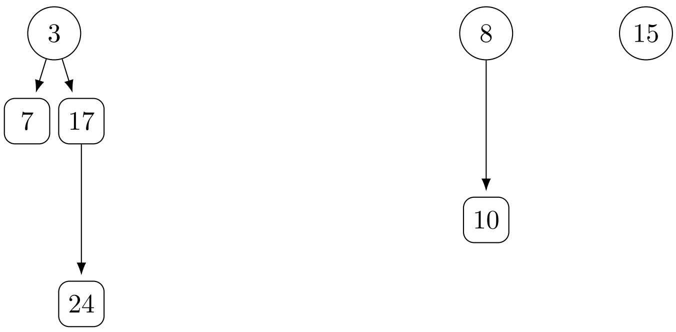
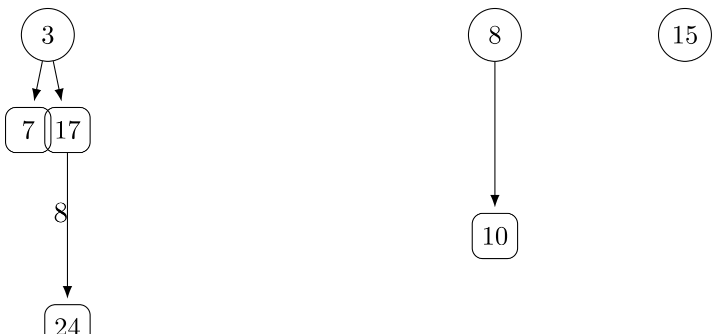


Figure 4: *
Step 3: Processed `10` and `8`. They collided at degree 0 so `link(10,8)` makes 10 a child of 8, giving deg(8) = 1.

minimum = 3 (root with smallest key)



A after 0 roots

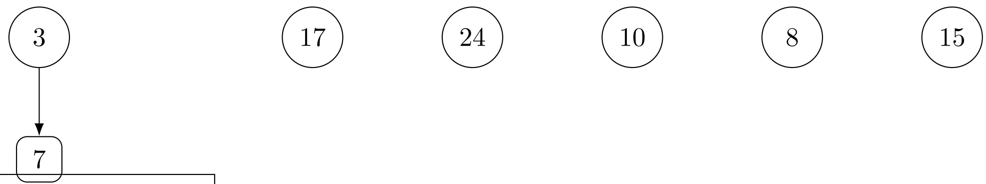
A[0]	A[1]	A[2]
NULL	NULL	NULL

Step 0: start (no root processed yet).

A after processing 3

A[0]	A[1]	A[2]
3	NULL	NULL

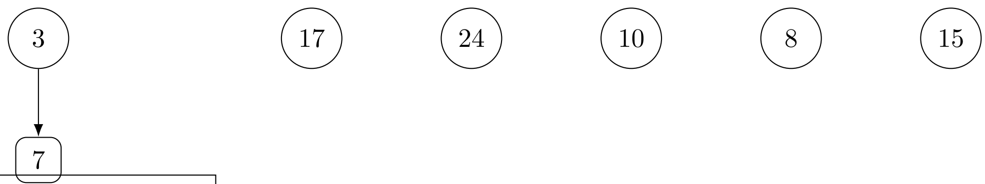
Processed: 3. $A[0] = 3$.



A after processing 7

A[0]	A[1]	A[2]
NULL	3	NULL

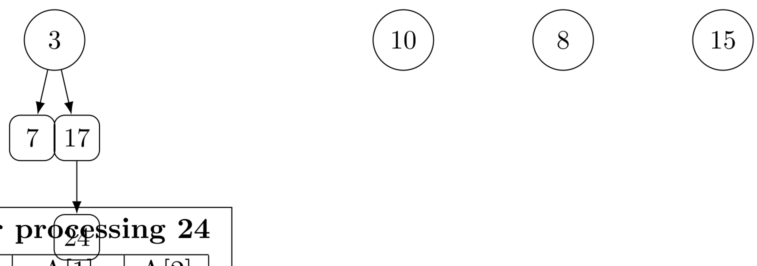
Processed: 7. Collision at degree 0 $\rightarrow$ link(7,3). Now deg(3) = 1, so $A[1] = 3$.



A after processing 17

A[0]	A[1]	A[2]
17	3	NULL

Processed: 17. No collision at degree 0, so $A[0] = 17$.



A after processing 24

A[0]	A[1]	A[2]
NULL	NULL	3

Processed: 24. link(24,17) $\rightarrow$ 17 has degree 1, collide with A[1]=3 $\rightarrow$ link(17,3). Now deg(3) = 2 so $A[2] = 3$.