Algorithms 2

Section 3: Advanced Data Structures

Amortised Analysis

Sometimes, a worst-case analysis is too pessimistic.

For example, consider a **vector**: an array that grows when necessary by allocating an array of twice the size and copying existing elements into the new array. The worst case cost of INSERT would assume that resizing is necessary.

Three common methods can be used to give more representative cost estimates:

1. Aggregate Analysis
2. The Accounting Method
3. The Potential Method

Aggregate Cost of Vector Insert

Let's start with an array of 16 items. The first 16 inserts take O(1) time. The 17th insert allocates an array of size 32, copies 16 items in O(n) time since n=16 at that point, then inserts one more item in O(1) time. The next 15 inserts take O(1) time and the next uses O(n) time again.

If we perform $N=2^k$ inserts, the total cost is:

16 + (16+1) + 15 + (32+1) + 31 + (64+1) + 63 + …

…which has sum $\in$ O(N). Dividing by N inserts, we conclude that the typical cost per insert is O(N)/N = O(1) amortised, per item.

💡 This proves a result we assumed in Algorithms 1 (see Stacks)

Accounting Method

The accounting method is more sophisticated.

We declare the amortised cost for each operation as the amount we charge our customer. Amortised costs might exceed the actual costs, with the excess going into a ‘credit’ account. When an amortised cost is less than the actual cost, the ‘credit’ pays for the shortfall.

The accounting method yields a valid set of amortised costs provided for *any sequence of operations*, the total amortised cost is an upper bound for the actual cost, and the credit never goes negative.

Potential Method

The potential method is similar but does not attribute ‘credit’ to particular operations or items within the data structure.

Instead, we measure the potential of the whole data structure.

$\phi(d_i)$ is the potential of the data structure in each state, i, it can get into through sequences of the supported operations. We require that $\phi(\text{initial}) = 0$ and that $\phi(d_i) \geq 0$ for all states, i.

Each operation’s amortised cost is the sum of the actual cost and the change in potential caused by the operation.

Example of the Potential Method [1]

Suppose we have a binary counter stored as a list of bits. We can use the potential function to calculate the amortised cost of INCREMENT, which adds one to the current value represented in binary by the string of bits.

We can use the number of 1s in the list of bits, b_i, after the ith increment as the potential function mapping any state of the list of bits to potential ≥ 0.

The initial state (counter=0) has no 1s in its binary representation so $\phi = 0$: this meets the requirement that the potential of the initial state is zero.

Example of the Potential Method [2]

If the ith increment operation, resets r_i bits from 1 to 0, the total actual cost is at most $r_i + 1$: from the least significant bit, we walk the string of bits either setting a 0 to a 1 and terminating, or setting a 1 to a 0 and rippling to the next bit.

The difference in potential before and after the increment is:

$\phi(d_i) - \phi(d_{i-1}) \leq (b_{i-1} - r_i + 1) - b_{i-1} = 1 - r_i$

The amortised cost is (actual + ϕ change) = $(r_i + 1) + (1 - r_i) = 2 \in O(1)$.

The total amortised cost for any sequence is an upper bound for the actual costs. All checks pass so n INCREMENT operations have amortised O(n) cost: O(1) each.

Abstract Data Types (ADTs)

We used the acronym ADT (three times) in Algorithms 1 but have yet to properly define it.

An **abstract data type** is to data structures what a Java Interface is to an algorithm: a list of the operations that must be supported (names, inputs/outputs, semantics), but without a specific implementation.

We have seen some examples already: a stack is an ADT and our implementation using an array is a data structure that implements the interface.

One ADT can extend another, adding further operations.

Binomial Heaps

Binomial Heaps implement the *Mergeable* Priority Queue ADT:

- CREATE(): creates a new, empty Binomial Heap.
- INSERT(bh, (k,p)): insert key/payload into a Binomial Heap.
- PEEK-MIN(bh): returns without removing the min key and its payload.
- EXTRACT-MIN(bh): returns are removes the min key and its payload.
- DECREASE-KEY(bh, ptr_k, nk): decreases key k (in node ptr_k) to nk.
- DESTRUCTIVE-UNION(bh1, bh2): merges two Binomial Heaps.
- COUNT(bh): returns the number of keys present.

DELETE(bh, ptr_k) = {DECREASE-KEY(bh, ptr_k, -∞); EXTRACT-MIN(bh);}

Binomial Heaps vs ordinary Heaps

The heaps we saw in Algorithms 1 perform all these operations in O(lg n) time or better *except for* DESTRUCTIVE-UNION.

The best implementation of DESTRUCTIVE-UNION on ordinary heaps would be to copy the two arrays into a single, larger array and call FULL-HEAPIFY. This would cost $\Theta(n_1 + n_2)$ when two heaps with those sizes are merged.

Binomial Heaps can perform DESTRUCTIVE-UNION in $O(\lg (n_1+n_2))$ time.

Binomial Heaps (like Heaps) do not provide a SEARCH operation. DECREASE-KEY and DELETE require the caller to be able to provide a pointer to a node.

Binomial Trees [1]

A Binomial Heap is a collection of Binomial Trees.

In a Binomial Tree, each node keeps its children in a strictly ordered list: these are not *binary* trees.

A Binomial Tree, B_k, is formed by linking two B_{k-1} trees together such that the root of one is the leftmost child of the other. B_0 is a single node.

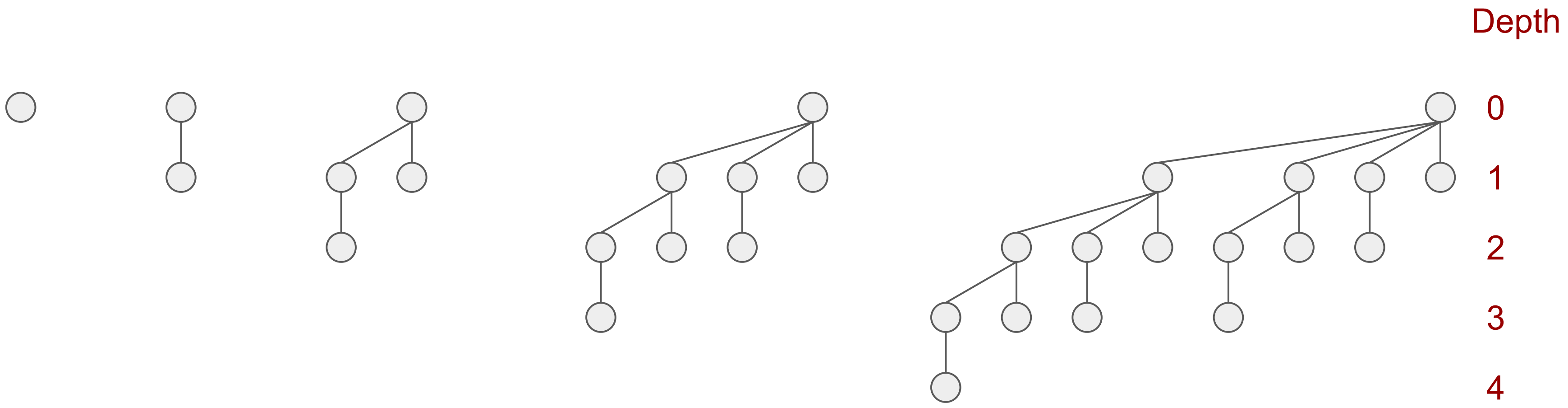
Binomial Trees [2]

These characteristics follow from the recursive definition of Binomial Tree, B_k:

1. There are 2^k nodes in the tree (note that these are not *binary* trees!)
2. The height of the tree is k
3. There are exactly ${}^kC_i$ nodes at depth i, for i = 0, 1, .. k
4. The root has degree k, which is greater than that of any other node
5. The children of the root are ordered: k-1, k-2, .. 0 and child i is the root of a subtree B_i obeying these defining characteristics.

All can trivially be proven by induction.
The maximum degree of any node is lg n (follows from 1 & 4).

Binomial Trees [3]



B_0 B_1 B_2 B_3 B_4 : a Binomial Tree with depth 4

Binomial Heaps

We build a Binomial Heap, H, out of Binomial Trees, as follows.

- Each Binomial Tree in H obeys the min-heap property: each node’s key is greater than or equal to that of its parent.
- For any non-negative integer k, there is at most one Binomial Tree in H with root node having degree k.

Notice this means that the overall minimum key must be one of the roots of the Binomial Trees.

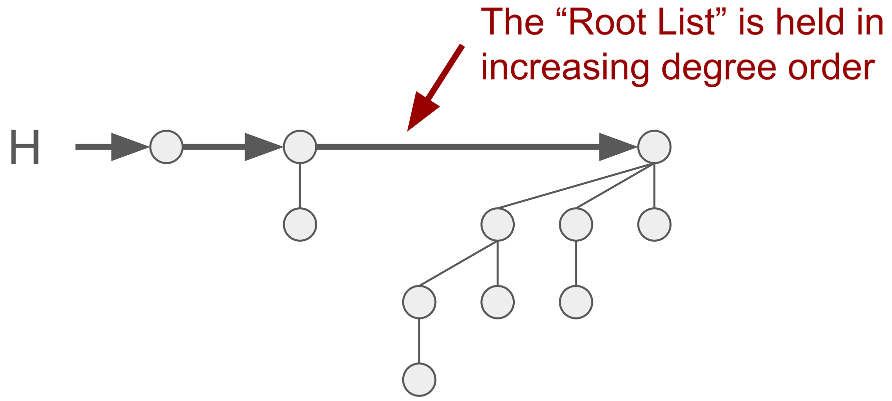
An n-node Binomial Heap contains at most $\lfloor \lg n \rfloor + 1$ Binomial Trees.

Binary Structure

Because the Binomial Tree B_i has 2^i nodes, it follows that a Binomial Heap with n nodes must contain trees B_i corresponding to the 1s in the binary representation of n.

$11 = 8 + 2 + 1 = 2^3 + 2^1 + 2^0$

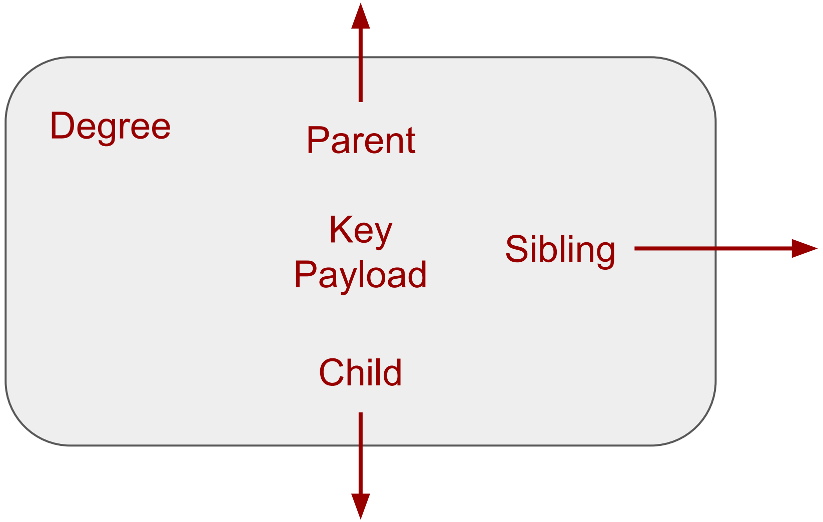
$11 = 1011_2$



Binomial Heap Data Structure [1]

To represent this structure, we need six attributes in each node:

1. Key
2. Payload
3. Next sibling pointer
4. Parent pointer
5. Child pointer (to ONE child)
6. Degree (number of children)



💡 Duplicate keys are permitted.
⚠️ Degree is the number of immediate children, not the number of descendants!

Binomial Heap Data Structure [2]

A reference to the root list of Binomial Heap is a pointer to the root of the first (lowest degree) Binomial Heap in the root list.

It is permitted (indeed, *required*) to keep pointers to nodes within the heap structure.

```
BH-CREATE()
    return NIL
```

Create is clearly O(1).

BH-PEEK-MIN(bh)

The minimum key has to be one of the roots.

We perform a sequential scan through the root list to find the minimum.

The root list contains at most $\lfloor \lg n \rfloor + 1$ Binomial Trees so this is $O(\lfloor \lg n \rfloor)$.

Note that is BH-PEEK-MIN: it does not BH-EXTRACT-MIN!

BH-DESTRUCTIVE-UNION(bh1, bh2) [1]

First consider the task of merging two Binomial Trees of the *same degree*.

BH-MERGE(bt1, bt2) makes bt2 become the first child of bt1 (increasing bt1.degree in the process). This is achieved by setting bt2.sibling = bt1.child and then bt1.child = bt2, and setting bt2.parent = bt1.

This is O(1) and maintains the order of the child list (characteristic #5 of Binomial Heaps): descending order of degree.

BH-DESTRUCTIVE-UNION(bh1, bh2) [2]

Now we can merge two Binomial Heaps.

bh1 and bh2 each has a root list that is sorted by increasing order of degree. We merge these in order, using BH-MERGE when we encounter two degrees of the same degree (smaller key remains in the root list). This ensures that the resulting Binomial Heap has at most one Binomial Tree of each degree and preserves the property that the root list contains at most $\lfloor \lg n \rfloor + 1$ Binomial Trees.

Because BT-MERGE is O(1), the running time of the operation to merge the two root lists is $O(\lfloor \lg n1 \rfloor + \lfloor \lg n2 \rfloor)$ and this is $O(\lfloor \lg n \rfloor)$, where n is the total number of nodes in the merged Binomial Heap.

BH-INSERT(bh, (key, payload))

The process to insert one new (key, payload) pair is to:

1. Create a new node, n, containing the (key, payload).
   n.child = NIL, n.parent = NIL, n.sibling = NIL, n.degree = 0.
2. A pointer to this node, p, is itself a Binomial Heap so we can return the result of a call to BH-DESTRUCTIVE-UNION(bh, p).

The running time is O(lg n), dominated by the BH-DESTRUCTIVE-UNION.

BH-EXTRACT-MIN(bh)

This is also straightforward!

1. Cut the Binomial Tree containing the old minimum out of the root list
   a. Use BH-PEEK-MIN to find the minimum if you don’t have a pointer to it already.
2. Reverse the list of the old minimum’s child list
3. BH-DESTRUCTIVE-UNION the (reversed) child list and the root list

All three steps can be achieved in O(lg n) time since that dominates both the length of the root list and the largest degree (child list length) of any node.

Note that we do not need to find the new minimum because the BH-PEEK-MIN operations searches for it each time.

BH-DECREASE-KEY(bh, ptr_k, nk)

ptr_k is a pointer to the node containing the key we wish to decrease.

Remember that this node is a node in a Binomial Tree, which is min-heap ordered!

We decrease the key using the same method as on a Min-Heap, in O(lg n) time:

1. Decrease the key to nk (it’s an error if nk > ptr_k.key)
2. If ptr_k.parent != NIL, access the parent and swap the keys (and payloads) if the new child key compares as smaller in the key sort order
3. Recurse up the tree until either the bubbling stops or we attempt to go to root’s parent (identified by parent = NIL). Max height is O(lg n), hence cost.

Fibonacci Heaps

Fibonacci Heaps implement the *Mergeable* Priority Queue ADT:

- CREATE(): creates a new, empty Fibonacci Heap. O(1)
- INSERT(fh, (k,p)): insert key/payload into a Fibonacci Heap. O(1) amortised
- PEEK-MIN(fh): returns without removing the min key and its payload. O(1)
- EXTRACT-MIN(fh): returns are removes the min key and its payload. O(lg n)
- DECREASE-KEY(fh, ptr_k, nk): decreases key k to nk. O(1) amortised
- DESTRUCTIVE-UNION(fh1, fh2): merges two Fibonacci Heaps. O(1) amortised
- COUNT(fh): returns the number of keys present. O(1)

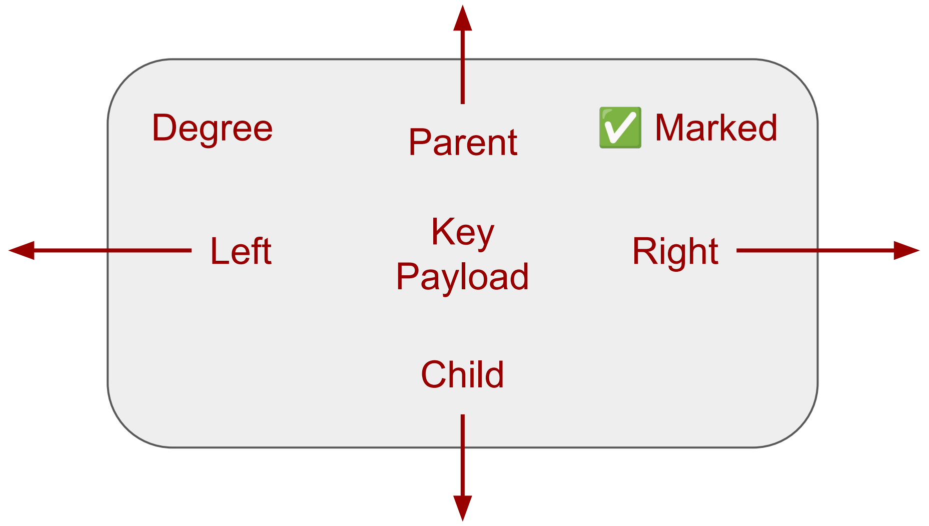
The low costs are what make Fibonacci Heaps special. Let's see how it's done!

💡 DELETE(fh, ptr_k) is DECREASE-KEY(fh, ptr_k, -∞); EXTRACT-MIN(fh). O(lg n) amortised
💡 INCREASE-KEY(fh, ptr_k, nk) is DELETE(fh, ptr_k); INSERT(fh,(ptr_k.key,ptr_k.payload)). O(lg n) amortised

Fibonacci Heap Data Structure [1]

Fibonacci Heap nodes have eight attributes:

1. Key
2. Payload
3. Left sibling pointer
4. Right sibling pointer
5. Parent pointer
6. Child pointer (to ONE child)
7. Degree (number of children)
8. Marked flag (a boolean)



💡 Duplicate keys are permitted.
⚠️ Degree is the number of immediate children, not the number of descendants!
💡 Marked: has this node lost a child since it became a child of its current parent?

Fibonacci Heap Data Structure [2]

A reference to the root of a Fibonacci Heap is a 2-tuple: fh = (r, n).

- r is a pointer to the node containing a current minimum key
- n is the number of keys currently present in the Fibonacci Heap

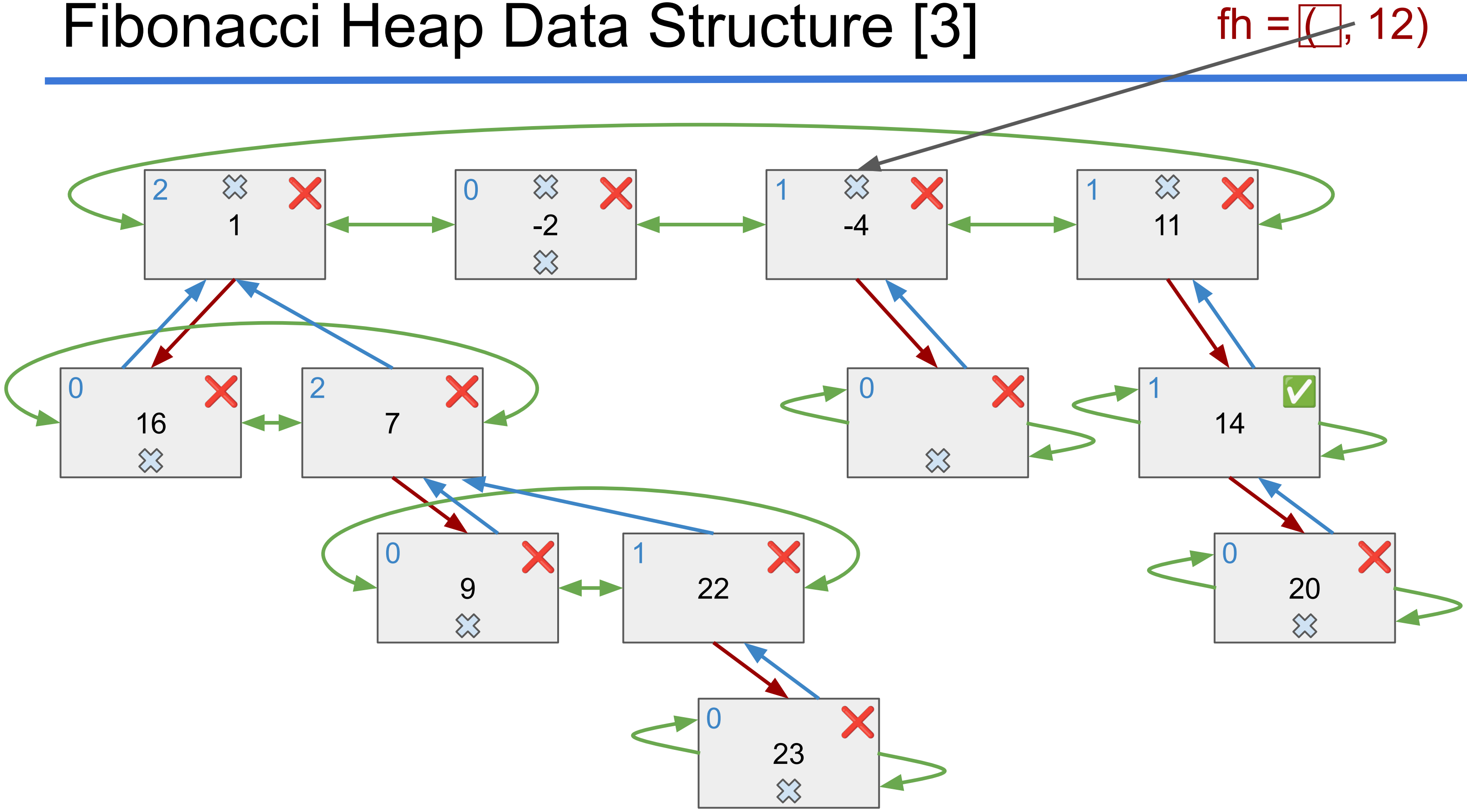
It is permitted (indeed, *required*) to keep pointers to nodes within the heap structure.

```
FH-CREATE()
    return (NIL, 0)
```

Create is clearly O(1).

Fibonacci Heap Data Structure [3]

fh = (□, 12)



Sibling
Child
Parent
Key

Fibonacci Heap Data Structure [4]

A collection of binomial min-heaps, held unordered in a doubly linked cyclic list.

The children of every node are held in unordered doubly linked cyclic lists.

Nodes in the root list are never marked.

If a node’s key is decreased and becomes smaller than the parent’s key then it violates the heap property and cannot remain in its current place in the heap:

- Move it into the root list
- Mark the parent (unless the parent is in the root list) but if the parent was already marked, move the parent to root list and recurse on its parent.

New FibHeapNode(k, p)

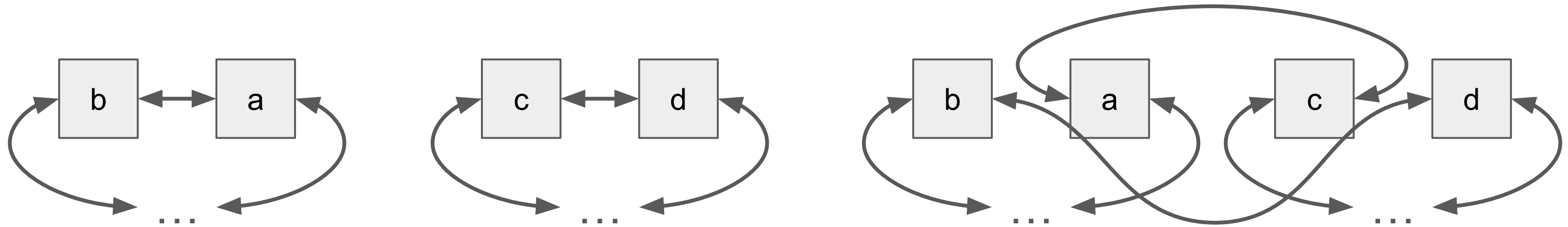
We initialise the 8 fields to create a valid 1-item Fibonacci Heap:

- Key = k
- Payload = p
- Left = <pointer to itself>
- Right = <pointer to itself>
- Parent = NIL
- Child = NIL
- Marked = false
- Degree = 0

FH-Destructive-Union(fh1, fh2) DLL-Splice(a, b, c, d)

```
let (p1, n1) = fh1, (p2, n2) = fh2
if (p1 == NIL) return fh2
if (p2 == NIL) return fh1
DLL-SPLICE(p1, p1.left, p2, p2.right)
if (p1.key ≤ p2.key) return (p1, n1+n2)
return (p2, n1+n2)
```

```
a.left = c
c.right = a
b.right = d
d.left = b
```



FH-INSERT(fh, (k, p))

```
let fh2 = new FibHeapNode(k, p)
return FH-DESTRUCTIVE-UNION(fh, fh2)
```

Notice that this *does* handle the case where fh is an empty Fibonacci Heap (see line 2 of FH-DESTRUCTIVE-UNION).

Notice that this does not put the new key into the correct place in a binomial heap structure. Instead, it “dumps” the new key into the root list.

FH-PEEK-MIN(fh)

```
let (p, _) = fh
if p == NIL
    return NIL
else
    return (p.key, p.payload)
```

FH-COUNT(fh)

```
let (_, n) = fh
return n
```

FH-DECREASE-KEY(fh, ptr_k, nk)

```
if (ptr_k.key < nk) error "New key is not smaller!"
ptr_k.key = nk
if ptr_k.parent != NIL && ptr_k.key < ptr_k.parent.key
    do if (!ptr_k.parent.marked)
            CHOP-OUT(fh, ptr_k); break
        else CHOP-OUT(fh, ptr_k); ptr_k = ptr_k.parent
    while ptr_k.parent != NIL
if (fh.p.key > nk) fh.p = nk
```

Private helper function CHOP-OUT(fh, ptr_k)

```
1   if (ptr_k.parent.degree == 1) ptr_k.parent.child = NIL
2   else if (ptr_k.parent.child == ptr_k)
3       ptr_k.parent.child = ptr_k.left
4   ptr_k.parent.degree = ptr_k.parent.degree - 1
5   if ptr_k.parent.parent != NIL ptr_k.parent.marked = true
6   ptr_k.left.right = ptr.right; ptr.right.left = ptr.left
7   ptr_k.parent = NIL
8   ptr_k.left = ptr_k.right = ptr_k
9   ptr_k.marked = false
10  DLL-SPLICE(fh.p, fh.p.left, ptr_k, ptr_k.right)
```

Only child of its parent

The child that the parent pointed to

Parent has one fewer children

Parent has lost a child

Cut out of parent's child list

Prepare this node to enter the root list

Splice into the root list

⚠️ Line 5 is careful to avoid marking nodes in the root list!

FH-EXTRACT-MIN(fh) [1]

This is where the magic happens.

Let (p, n) = fh.

- If n==1 then this is the last node in the Fibonacci Heap so return (NIL, 0).

The new minimum key has to be one of the children of the old minimum, or one of the other keys in the root list. We begin by dropping the current minimum's children into the root list:

- If p.child != NIL then set v.parent=NIL and v.marked=false for all nodes, v, in the p.child list; and then call DLL-SPLICE(p, p.left, p.child, p.child.right)

FH-Extract-Min(fh) [2]

Now we can cut the old minimum out of the root list:

- p.left.right = p.right
- p.right.left = p.left
- p = p.left
- n = n - 1

We need to walk around the root list looking for the new minimum key. Because each entry in the root list is a min-heap, we know the overall minimum cannot be deep into any of the heaps, but it could be any of the roots as there is no ordering between them.

FH-EXTRACT-MIN(fh) [3]

- let start = p, t = p.right
- while t != p
    - if (t.key < p.key) then p = t

p and n are now set correctly so we could say that we’re done and return (p, n). Although that would implement the operations correctly, it would not achieve the asymptotic costs we claimed.

It turns out that all we need to do is clean up the heap at this point.

FH-Extract-Min(fh) [4]

We are going to need an array with D(n) + 1 = $\lfloor \log_\varphi n \rfloor + 1$ elements, initialised to NIL (‘n’ is the node count *after* removing the old minimum). $\varphi = (1+\sqrt{5})/2$. D(n) is the maximum degree of any node in a Fibonacci Heap with n nodes.

It will be convenient to index this array from 0, not from 1.

```
- let A = new Array[ $\lfloor \log_\varphi n \rfloor + 1$ ]
- for i = 0 to A.length-1
    - A[i] = NIL
```

While we are walking around the root list, we combine heaps with the same degree, using the array to remember where we last saw a heap with each degree.

⚠️ The array length must be 1 + max_degree. We will prove this value later.

FH-EXTRACT-MIN(fh) [5]

When considering node ‘t’ in the root list we…

- if A[t.degree] == NIL
    - A[t.degree] = t
- else
    - old_start = start
    - old_start_right = start.right
    - merge(t, A[t.degree])
    - if (old_start.parent != NIL) start = old_start_right

What does merge do? Well…

FH-EXTRACT-MIN(fh) [6]

MERGE(a, b):

```
- if a.key >= b.key
    - A[b.degree] = NIL
    - b.degree = b.degree + 1
    - a.left.right = a.right; a.right.left = a.left
    - a.left = a.right = a
    - if (b.degree == 0)
        - b.child = a
    - else
        - DLL-SPLICE(b.child, b.child.left, a, a.right)
    - if (A[b.degree] != NIL) MERGE(b, A[b.degree])
    - else A[b.degree] = b
```

FH-EXTRACT-MIN(fh) [7]

```
- else
    - A[a.degree] = NIL
    - a.degree = a.degree + 1
    - b.left.right = b.right; b.right.left = b.left
    - b.left = b.right = b
    - if (a.degree == 0)
        - a.child = b
    - else
        - DLL-SPLICE(a.child, a.child.left, b, b.right)
    - if (A[a.degree] != NIL) MERGE(a, A[a.degree])
    - else A[a.degree] = a
```

Intuition behind Fibonacci Heaps

Before we prove the asymptotic running times for the key operations, let's get an intuition for why they're so cheap.

To get started, let's consider only the Priority Queue ADT operations operations: CREATE, INSERT, PEEK-MIN, EXTRACT-MIN.

It's obvious that our implementations of CREATE and PEEK-MIN are O(1): the code only performs a fixed number of fixed-time operations. No loops in either case.

Intuition behind Fibonacci Heap INSERT

It’s also obvious that INSERT does a constant amount of work when it is called but we might consider that it is only doing part of the job: it is putting the item into the data structure but not into the correct place in the data structure.

There are two problems with only doing half a job:

1. You have to come back and do the rest of the work later; and
2. There is a price to pay for putting “spurious” items into the root list: it costs more time to run extract-min.

When we consider EXTRACT-MIN, we see that neither is asymptotically important.

Intuition behind Fibonacci Heap EXTRACT-MIN [1]

EXTRACT-MIN:

- Drops the old minimum's children into the root list – O(1)
    - Because the lists are cyclic, we do not need to walk to the start/end of either to append lists
    - Because the lists are unordered, it does not matter where we join the two lists together
    - We do have to set the parent pointers to NIL and the marked flags to false: come back to this!
- Cuts the old minimum out of the root list – O(1)
    - We do not need to search for the node containing the minimum (we have a pointer to it)
    - The root list is doubly linked so we can delete in O(1) time
- Walks around the root list looking for the new minimum – O(r), r: len root list
    - It's O(1) work per item to compare it to the running minimum
    - It's O(1) per item to set parent=NIL and marked=false so we can absorb the earlier costs!

Intuition behind Fibonacci Heap EXTRACT-MIN [2]

EXTRACT-MIN is doing O(r) work but we only charge the customer O(lg n), so there is a shortfall to explain with an amortised analysis.

Suppose, for every O(1) Insert also put O(1) money into bank account, and that money can be used to pay for work that is done later. O(1) money can be used to pay for any constant amount of work.

We need to spend O(r - lg n) money from the bank account to balance the books for EXTRACT-MIN.

We can only spend the money once so what about the second EXTRACT-MIN?

Intuition behind Fibonacci Heap EXTRACT-MIN [3]

The reason that $r > \lg n$ is because there is “junk” in the root list that shouldn’t be there: all the keys we inserted cheekily in the wrong place!

It’s OK to use the bank account to pay for scanning through those keys once to find the new minimum but we have to make sure those keys do not need to be scanned the next time we run extract-min.

This is exactly achieved by combining of roots of the same degree: trees begin as single nodes (1 key), are combined into 2s, 4s, 8s, 16s, etc. so if we have n keys then we have at most lg n min-heaps in the root list, i.e. the root list shrunk from r to lg n, and $r - \lg n$ is exactly the correct amount of money to balance the books!

⚠️ This informal intuition does not account for DECREASE-KEY. (We use “lg n”, not “$\log_\varphi n$”.)

Intuition behind Fibonacci Heap DECREASE-KEY [1]

To account for decrease key, we note that it takes O(1) time to replace the key, compare it to the parent (thanks to the parent pointer) and, if necessary to cut it out of the parent's child list and splice it into the root list (thanks to both being doubly linked). Even if we cut the parent out of its sibling list as well, that's still only O(1) work.

The problem comes when the parent's parent is already marked, and its parent, and so forth – we do an amount of work that is proportional to the height of that min-heap and this is not O(1).

Same trick! Actual cost is O(h); customer pays O(1); bank funds O(h - 1). How?

Intuition behind Fibonacci Heap DECREASE-KEY [2]

Every time we remove a child and mark its parent node (the parent *not* already being marked), we put 2x O(1) amounts of money into our bank account. DECREASE-KEY costs O(1) so this does not change its asymptotic cost.

When that marked node loses another child, one of those O(1) amounts of money can pay for its removal and splicing into the root list. If its parent is also marked then it, too, has 2x O(1) amounts of money in the bank, one of which can be used to pay for it to fall into the root list. This continues and as each level in the min-heap pays for its own O(1) work, the total paid is the O(h - 1) shortfall.

BUT… don't all those items in the root list add to the cost of EXTRACT-MIN?!

Intuition behind Fibonacci Heap DECREASE-KEY [3]

Yes, all those decreased-keys in the root list do increase the cost of EXTRACT-MIN but we have that second O(1) amount of money that we haven't spent yet.

The second O(1) amount of money is what pays for the additional costs of scanning the root list during the next EXTRACT-MIN operation. Since that recombines trees, leaving O(lg n) items in the root list it only needs to be spent once.

The decreased keys and marked parents that fell into the root list have a corresponding O(1) amount of money in the bank, mirroring the O(1) money contributed by INSERT for new keys, and paying for their clean-up.

Formal analysis: amortised analysis

We need a potential function that is non-negative, zero for the empty data structure, and sufficient to “pay for” the expensive steps in the algorithms such that we can claim amortised costs:

- CREATE(), COUNT(fh), PEEK-MIN(fh) : O(1)
- INSERT(fh, (k,p)): O(1) amortised
- EXTRACT-MIN(fh): O(lg n) amortised
- DECREASE-KEY(fh, ptr_k, nk): O(1) amortised
- DESTRUCTIVE-UNION(fh1, fh2): O(1) amortised

Coming up with a potential function

- It often helps to compare the ideal state of your data structure with the actual state, which might be “damaged” by the cheeky operations that did something cheaply but imperfectly.
  - The potential needs to be (at least) what it would cost to fix the damage.
- Consider the cost actual cost of the operations you need to perform and what you want to charge for them, since the difference is what you need the potential to cover. Which other operations lead to these operations being more expensive than they might be, and could you get them to pay for the clean-up in advance?

Potential for a Fibonacci Heap

It turns out that a good potential function is $\phi = r + 2m$, where r is the number of items in the root list and m is the number of marked nodes.

Check: $\phi = 0$ for an empty Fibonacci heap?

- Yes, because CREATE returns (NIL, 0): the root list is empty and there are no marked nodes (no nodes at all).

We proceed to check the other operations…

Amortised analysis of FH-INSERT(fh, (k,p))

Start with any Fibonacci Heap, fh, (including the empty case), with potential $\phi_1 = r + 2m$.

FH-INSERT adds (via a call to the destructive union operation) one item to the root list. The new node is never marked. Existing marked nodes remain marked; existing unmarked nodes remain unmarked. **r increases by 1; m is unchanged.**

The potential after insert, $\phi_2 = (r+1) + 2m = \phi_1 + 1$.

The amortised cost of insert is {cost of immediate work} + {change in potential} $= k + (\phi_2 - \phi_1) = k + 1 \in O(1)$ ✅ this is as claimed.

Amortised analysis of FH-DESTRUCTIVE-UNION(fh1, fh2)

Start with Fibonacci Heaps, fh1 and fh2, with potentials
$\phi_1 = r_1 + 2m_1$ and $\phi_2 = r_2 + 2m_2$.

FH-DESTRUCTIVE-UNION splices the roots lists together, adds n1 and n2, and compares the min keys to return a value. No marked flags are changed. The potential of the combined Fibonacci Heap is $\phi_3 = (r_1 + r_2) + 2(m_1 + m_2)$.

The amortised cost of destructive-union is {cost of immediate work} + {change in potential}
$= k + (\phi_3 - (\phi_1 + \phi_2)) = k \in O(1)$ ✅ this is as claimed.

⚠️ The input heaps cannot be used after the call to FH-DESTRUCTIVE-UNION so they do not need their potential to "repair" them: that potential can be repurposed to repair the damage on the combined heap!

Amortised analysis of FH-DECREASE-KEY(fh, ptr_k, nk) [1]

Start with any Fibonacci Heap, fh, with potential $\phi_1 = r + 2m$.

After decreasing a key, we might be in a variety of states.

If we decreased the key but it remains greater than its parent, then no changes are made to the root list, nor to any marked flags.

The potential is unchanged so the amortised cost is the actual cost in this case, and is clearly $\in O(1)$.

Amortised analysis of FH-DECREASE-KEY(fh, ptr_k, nk) [2]

If the ptr_k.key becomes smaller than its parent, but the parent is not marked then the decreased key falls into the root list and the parent becomes marked.

If the node pointed to by ptr_k was marked before, it becomes unmarked when it falls into the root list.

If !ptr_k.marked beforehand, then the change in potential is (r+1)-r + 2(m+1-m) = 3

If ptr_k.marked beforehand, then the change in potential is (r+1)-r + 2(m-m) = 1

In both cases, the work done immediately is constant and the contribution to the potential is constant so the amortised cost is ∈ O(1).

Amortised analysis of FH-DECREASE-KEY(fh, ptr_k, nk) [3]

If the ptr_k.key becomes smaller than its parent, and the parent is already marked then the parent falls into the root list (and becomes unmarked), and its parent might do the same. Suppose, in total, ‘a’ ancestor nodes fall into the root list.

The total work done is $k_1 + a.k_2$.

The potential afterwards is $(r+1+a) + 2(m-a)$ so the change is $1+a-2a = 1-a$.

The sum is $k_1 + a.k_2 + (1-a) = k_3$ (choosing $k_2 = 1$) so the amortised cost is $\in O(1)$.

In all cases, FH-DECREASE-KEY has amortised cost O(1) ✅ this is as claimed.
⚠️ We should also consider the two cases of ptr_k being marked and not before the decrease-key.
⚠️ There is really another case split here! The final ancestor might be in the root list (so does not get marked) or not in the root list (does get marked). This slide shows the second (worst case).

Amortised analysis of FH-EXTRACT-MIN(fh) [1]

- We start with r nodes in the root list and m marked nodes.
- We add the old minimum's children into the root list: at most D(n) children where D(n) is the maximum degree of a node in a Fibonacci Heap with n nodes.
- We remove one item from the root list (the minimum).
- When we scan the root list, looking for the new minimum, there are at most r + D(n) - 1 nodes in the root list.
- The cost of finding the new minimum $\in$ O(r + D(n))

Amortised analysis of FH-EXTRACT-MIN(fh) [2]

- When we combine nodes of the same degree, we loop through each node in the root list:
  - If we do not remove a node from the root list (now or through later merges with it), we do constant work on it because we put it in the array and do not remove it.
  - If we do remove a node from the root list, we only compare it to one other before doing so: one is found in O(1) time using the array, and we can only remove a node once.
- The total work to merge nodes is O(r + D(n))
- This leaves at most D(n)+1 nodes in the root list because, if there were more, there would be two items in the same array position and we would have merged them. (Remember the array length was D(n)+1.)

Amortised analysis of FH-EXTRACT-MIN(fh) [3]

Now we can analyse the change in potential. Note that no nodes changed their marked flag during the merges.

Potential before = $r + 2m$

Potential afterwards (worst case) = $(D(n) + 1) + 2m$
Change in potential = $(D(n) + 1) + 2m - (r + 2m) = D(n) + 1 - r$
Amortised cost = $(r + D(n)) + (D(n) + 1 - r) \in O(D(n))$

⚠️ Worst case because fewer items in the root list would release more potential to pay for work. Also, if any of the old minimum's children were marked, unmarking them would release potential.

Amortised analysis of FH-EXTRACT-MIN(fh) [4]

To show that the cost of EXTRACT-MIN is O(lg n), we need to show that D(n) is bounded from above by k.lg n.

We will show that D(n) ≤ $\lfloor \log_{\varphi} n \rfloor$ where $\varphi = (1+\sqrt{5})/2$ is the golden ratio.

For any node x, define size(x) to be the total number of nodes in the heap rooted at node x, including node x itself. (Node x need not be in the root list.)

Amortised analysis of FH-EXTRACT-MIN(fh) [5]

Lemma 1: let x be a node in Fibonacci Heap; if x has degree k then let its children be c_1, c_2, .. c_k in the order they were added as children of x; we have that c_1.degree ≥ 0 and c_i.degree ≥ i-2 for i=2..k.

Proof

c_1's degree must be at least zero because any node's degree is non-negative.

x and c_i had the same degree when they were merged, and x had i-1 children at that point. Since then, c_i can have lost at most one child (since losing a second would have removed it from x's parentage) so c_i.degree ≥ i-2.

Amortised analysis of FH-EXTRACT-MIN(fh) [6]

Fibonacci numbers, indexed as the 0th, 1st, … are defined by

```
let rec fib(k) = if (k < 2) then k else fib(k-1) + fib(k-2)
```

0, 1, 1, 2, 3, 5, 8, 13, 21, … correspond to k=0, 1, 2, 3, 4, 5, …

Notice that

$\mathrm{fib}(k+2) = 1 + \Sigma^{k}_{i=0}\, \mathrm{fib}(i)$

(Trivially proved by induction.)

Amortised analysis of FH-EXTRACT-MIN(fh) [7]

Notice also that the (k+2)th Fibonacci number, $\mathrm{fib}(k+2) \geq \varphi^k$.

Proof: by induction.

Base case k=0: $\mathrm{fib}(0+2) = 1 = \varphi^0$.
Base case k=1: $\mathrm{fib}(1+2) = 2 > \varphi^1 = 1.619\ldots$

Inductive step uses strong induction: assume that $\mathrm{fib}(i+2) \geq \varphi^i$ for all $i = 2..k-1$ and prove $\mathrm{fib}(k+2) \geq \varphi^k$ for $k \geq 2$.

$$\mathrm{fib}(k+2) = \mathrm{fib}(k+1) + \mathrm{fib}(k) \geq \underbrace{\varphi^{k-1} + \varphi^{k-2}}_{\text{By inductive hypothesis}} = \underbrace{\varphi^{k-2}(\varphi + 1) = \varphi^{k-2}\,\varphi^2}_{\text{Because } \varphi \text{ is the positive root of } x^2 = x+1} = \varphi^k$$

Amortised analysis of FH-EXTRACT-MIN(fh) [8]

Lemma 2: let x be any node in a Fibonacci Heap and let k = x.degree; then size(x) ≥ fib(k+2) ≥ φ^k.

Proof

Denote the minimum possible size of any node of degree k as s_k.

s_0 = 1 and s_1 = 2, and considering a node x with degree k, s_k ≤ size(x).

Note that s_k increases monotonically with k (adding children cannot decrease the minimum size).

Amortised analysis of FH-EXTRACT-MIN(fh) [9]

Now consider some node z with degree k and size(z) = s_k (i.e. minimum size).

$s_k \leq$ size(x) so a lower bound on s_k is a lower bound on size(x).

Consider the children $c_1, c_2, .. c_k$ of z in the order they were added.
size(x) $\geq s_k \geq$
1 (for z itself)
+ 1 (for c_1, also a zero-degree node when merged with z, or
now a larger child if the original first child was removed)
+ $\Sigma^k_{i=2}\ s_{ci.degree}$
$\geq 2 + \Sigma^k_{i=2}\ s_{i-2}$ // using Lemma 1 and monotonicity

Amortised analysis of FH-EXTRACT-MIN(fh) [10]

Next, we show that $s_k \geq \mathrm{fib}(k+2)$ for $k \geq 0$, using induction.

Bases cases $k = 0$ and $k = 1$ follow immediately from the definitions of s_k and fib(k+2): $s_0 = 1 = \mathrm{fib}(0+2)$ and $s_1 = 2 = \mathrm{fib}(1+2)$.

Inductive step: $k \geq 2$. The induction hypothesis gives us that $s_i \geq \mathrm{fib}(i+2)$ for $i = 0..k-1$ and we seek to prove this property for $i = k$.

$$\begin{aligned} s_k &\geq 2 + \Sigma^k_{i=2}\ s_{i-2} \\ &\geq 2 + \Sigma^k_{i=2}\ \mathrm{fib}(i) \\ &= 1 + \Sigma^k_{i=0}\ \mathrm{fib}(i) \\ &= \mathrm{fib}(k+2) \geq \varphi^k \end{aligned}$$

using the properties of Fibonacci numbers

Amortised analysis of FH-EXTRACT-MIN(fh) [11]

If x is any node in a Fibonacci Heap and has k = x.degree, then we know that $n \geq \text{size}(x) \geq \varphi^k$.

Taking logs to base φ, we have that $k \leq \log_{\varphi} n$.

Since k must be an integer, we have $k \leq \lfloor \log_{\varphi} n \rfloor$.

Because this is true for any node, we have that the maximum degree of any node in a Fibonacci Heap with n nodes is $D(n) \leq \lfloor \log_{\varphi} n \rfloor \in O(\lg n)$.

Hence the amortised cost of EXTRACT-MIN $\in O(\lg n)$. ✅ this is as claimed.

Uses for Fibonacci Heaps

Fibonacci Heaps used to be used in the Linux Kernel as the priority queue of processes, waiting to be chosen to run by the Process Scheduler.

It was replaced with a Red-Black tree that, although it has larger asymptotic costs, runs faster on the typical size of problem instance, due to (much) lower constant factors.

Fibonacci Heaps in Dijkstra's Algorithm

We said that in the worst case, Dijkstra's algorithm will call CREATE once, INSERT O(|V|) times, EXTRACT-MIN O(|V|) times, and DECREASE-KEY O(|E|) times.

Priority Queue	**CREATE**	**INSERT**	**EXTRACT-MIN**	**DECREASE-KEY**	**Total**
Sorted Linked List	O(1)	O(|V|)	O(1)	O(|V|)	$O(\|V\|^2 + \|V\|\|E\|)$
Sorted Array	O(|V|)	O(|V|)	O(|V|)	O(|V|)	$O(\|V\|^2 + \|V\|\|E\|)$
Heap	O(1)	O(lg |V|)	O(lg |V|)	O(lg |V|)	O(|V| lg |V| + |E| lg |V|)
Fibonacci Heap	O(1)	O(1) amortised	O(lg |V|) amortised	O(1) amortised	O(|V| lg |V| + |E|) amortised

Are there any better mergeable priority queues?

Actually, there are two!

1. 1996: Gerth Stølting Brodal (Aarhus University, Denmark) invented **Brodal Heaps**, which achieve *actual* O(1) worst case running time for all operations except EXTRACT-MIN, which is O(lg n). Actual means “not amortised”.
2. 2012: Gerth S. Brodal, George Lagogiannis, and Robert E. Tarjan invented **Strict Fibonacci Heaps**, which achieve the same *actual* asymptotic bounds and a simpler set of algorithms.

These are asymptotically optimal (on conventional hardware). Proof: we could do comparison sorts in less than O(n lg n) time with better data structure/algorithms.

Disjoint Sets

The **Disjoint Set ADT** can be implemented in many ways, including by the use of a data structure that is based on an amortised cost analysis.

The Disjoint Set ADT is initialised with a collection of n distinct keys. Each key is placed into its own set. The data structure supports two operations:

1. UNION(s_1, s_2): combine two disjoint sets, s_1 and s_2, into a single set
2. IN-SAME-SET(k_1, k_2): report whether keys k_1 and k_2 are currently in the same set (return true) or different sets (return false)

Disjoint Sets using Doubly Linked Lists

CREATE: for n provided keys, create n linked lists, each of length 1, holding their corresponding key.

UNION(a, b): walk forwards along a’s list until you reach the end, and along b’s list until you reach the beginning. Change the list pointers to join the end of the ‘a’ list to the start of the ‘b’ list.

IN-SAME-SET(k_1, k_2): from the node holding k_1, walk in both directions until you reach a NIL pointer. If a node containing k_2 is found, return true; else return false.

CREATE: O(n) UNION: O(n) IN-SAME-SET: O(n)

Disjoint Sets using Cyclic Doubly Linked Lists

CREATE: for n provided keys, create n cyclic linked lists, each of length 1, holding their corresponding key.

UNION(a, b): splice a’s list and b’s list together. As the lists are unordered, we can splice at the positions pointed to by a and b (which do not require any searching to find).

IN-SAME-SET(k_1, k_2): from the node holding k_1, walk around until you find k_2 or get back to k_1. If a node containing k_2 is found, return true; else return false.

CREATE: O(n) UNION: O(1) IN-SAME-SET: O(n)

Disjoint Sets using Hash Tables

CREATE: create a hash table and insert the (key, payload) pairs (k_i, i). This represents the starting point that key k_i is in set i.

UNION(a, b): scan through every record in the hash table; if some key maps to payload b, change the payload to a.

IN-SAME-SET(k_1, k_2): return HT-SEARCH(k_1) == HT-SEARCH(k_2)

CREATE: O(n) UNION: O(n) IN-SAME-SET: O(1)

Disjoint Sets with Path Compression & Union by Rank [1]

CREATE: create a tree node for each key. This yields n separate trees. The data stored in each tree node is a pointer to a tree node, initialised to NIL. Each node also contains an integer estimating (upper-bounding) the depth of the tree, initialised to 0.

CHASE(k): starting from the node for key k, follow the pointers until you reach the root, r, of its tree (where pointer == NIL). Change the pointer of each node you went through to ‘r’. This ensures that the next time we CHASE(k), or we chase any descendant of k, we jump straight from k to r. The cost of walking the path from k to r is only paid once. This is called **path compression**.

CREATE: O(n) UNION: ~O(1) amortised IN-SAME-SET: ~O(1) amortised

Disjoint Sets with Path Compression & Union by Rank [2]

UNION(a, b): let r_a = CHASE(a), r_b = CHASE(b). If the estimated depth of r_a is strictly greater than that of r_b, then change r_b's pointer to r_a. If r_b is deeper than r_a then change r_a's pointer to r_b. If both depths are equal, make either point to the other and increment the estimated depth of the root (the one pointed *to*). This is called **union-by-rank**.

IN-SAME-SET(k_1, k_2): return CHASE(a) == CHASE(b). If the roots of the trees containing the two keys are the same then the keys are in the same set.

CREATE: O(n) UNION: ~O(1) amortised IN-SAME-SET: ~O(1) amortised

Kruskal’s Algorithm using Disjoint Sets

In the worst case, Kruskal’s Algorithm CREATEs a disjoint set representation exactly once, calls UNION |V|-1 times and calls IN-SAME-SET |E| times.

Kruskal also sorts the edges: O(|E| lg |E|) = O(|E| lg |V|) since |E| is at most $|V|^2$.

Disjoint Set	**Sort**	**CREATE**	**UNION**	**IN-SAME-SET**	**Total**
DLLs	O(|E| lg |V|)	O(|V|)	O(|V|)	O(|V|)	$O(\|V\|^2 + \|V\|\|E\|)$
Cyclic DLLs	O(|E| lg |V|)	O(|V|)	O(1)	O(|V|)	O(|V| + |V||E|)
Hash table	O(|E| lg |V|)	O(|V|)	O(|V|)	O(1)	$O(\|V\|^2 + \|E\| \lg \|V\|)$
Trees with PC & UbR	O(|E| lg |V|)	O(|V|)	~O(1) amortised	~O(1) amortised	~O(|V| + |E| lg |V|) amortised