Fibonacci Heap

A priority queue is an abstract data type, which supports the following operations:

- INSERT(H, x): insert the key x to the heap
- EXTRACT-MIN(H): remove the minimum-key element from the heap and return it
- DECREASE-KEY(H, x, k): update the key of the element x to k, which is assumed to be no greater than the original key value.

Such a data structure is useful in many algorithms, most notably in Dijkstra's algorithm for finding Single Source Shortest Paths on a directed graph, as well as in Prim's algorithm for finding the Minimum Spanning Tree of a graph. In both of these algorithms, assuming we have a graph with n nodes and m edges, we make a total of n EXTRACT-MIN operations, and m DECREASE-KEY operations.

Usually, we implement a priority queue using a Binary Heap, which is a simple data structure, which can be implemented by an array. It supports all three operations in worst case $O(\log n)$ time.

A Fibonacci Heap is a data structure which implements the priority queue operations with better amortized costs. In particular, it supports $O(1)$ INSERT, $O(1)$ DECREASE-KEY and $O(\log n)$ EXTRACT-MIN operations (amortized). Theoretically, this provides an asymptotic improvement to the running times of Dijkstra's and Prim's algorithms (as compared to using a binary heap) from $O(n \log n + m \log n)$ to $O(n \log n + m)$. (This is a strict improvement when $m \notin O(n)$, i.e. the graph is dense.) Let us try to build up this data structure from the ground up.

Binomial Heaps

Let us consider an intermediate data structure called a binomial heap, and try to implement a priority queue using it.

A **Binomial Tree** of order k, B_k, is defined recursively:

- B_0 is a tree with a single node.
- B_k is defined as two B_{k-1} trees joined such that the root of one is the leftmost child of the root of the other.

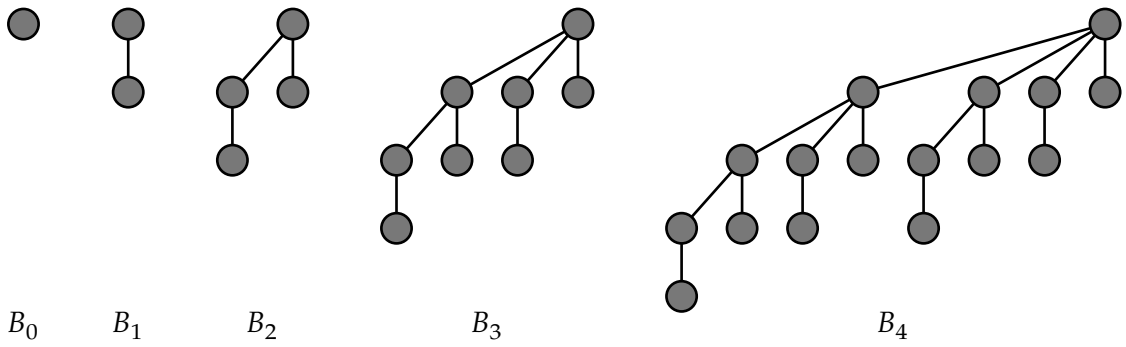

Figure 1: Binomial trees of orders 0 through 4

These trees have some special properties which we can prove easily by induction:

1. In the tree B_k, there are exactly 2^k nodes.
2. The height of tree B_k is k.

3. There are exactly $\binom{k}{i}$ nodes at depth i, $i = 0, 1, 2, \ldots$.
4. The order of the tree k is the degree of the root element, i.e. the root node of B_k has k children.
5. Each child of the root is itself a binomial tree. In particular, the i th child (from the right; as drawn in the figures, $i = 0, 1, 2, \ldots, k-1$) is B_i.
6. By extension, subtree of any node in a binomial tree is itself a binomial tree.

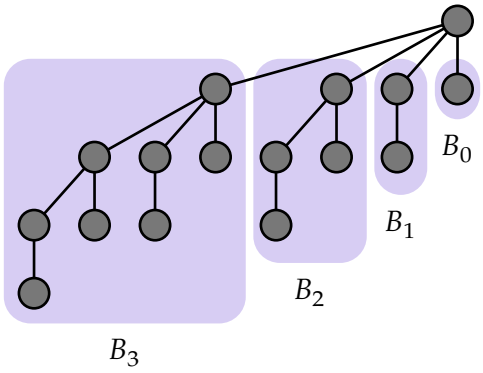

Figure 2: The children of B_4 are themselves binomial trees, B_0, B_1, B_2 and B_3 from right to left.

A **Binomial Heap** H is a set of binomial trees with the following properties:
1. Each node has a key.
2. Each binomial tree follows the min-heap property.
3. There may be at most one binomial tree of order k, for all $k = 0, 1, 2, \ldots$.

Observe, that this definition leads to a unique structural representation for a binary heap containing n elements. Since each B_i contains 2^i nodes, and each B_i can occur at most once, there is a unique set of B_is that need to be chosen to get exactly n nodes. In particular, this corresponds to the unique binary representation of n:

$$n = \sum_{i=0}^{\infty} b_i 2^i$$

The set of binomial trees that we use exactly correspond to is such that $b_i = 1$, i.e. the positions of the one bits in the binary representation of n.

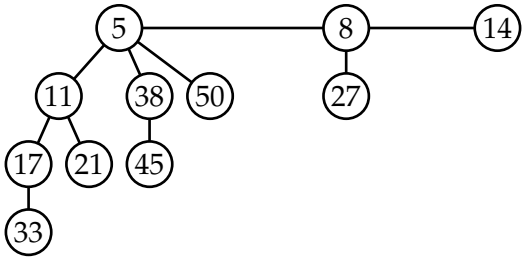

Figure 3: A binomial heap with $n = 11$ nodes, represented by a set of binomial trees B_3, B_1 and B_0

We can thus claim, that a binomial heap having n nodes can have at most $\lfloor \lg n \rfloor + 1 = O(\log n)$ binomial trees.

Let us try to implement the three operations on our binomial heap data structure.

INSERT

We need to add a new element to the binomial heap. Let us consider, we add it as a single node (B_0) to the root list. Now, if there was no other order 0 tree in the heap before this insertion, then we are done, the resulting structure is still a binomial heap.

Otherwise, we are violating the unique order constraint on the trees. Since we have 2 trees of the same order, we can *link* them; which is a constant time operation:

LINK(T_1, T_2)

```
1  check that order(T1) = order(T2)
2  if T1.root < T2.root
3      add T2 as the leftmost child of T1
4      remove T2 from the root list
5  else
6      add T1 as the leftmost child of T2
7      remove T1 from the root list
```

We need to continue merging trees of same degree until all our trees have unique degrees.

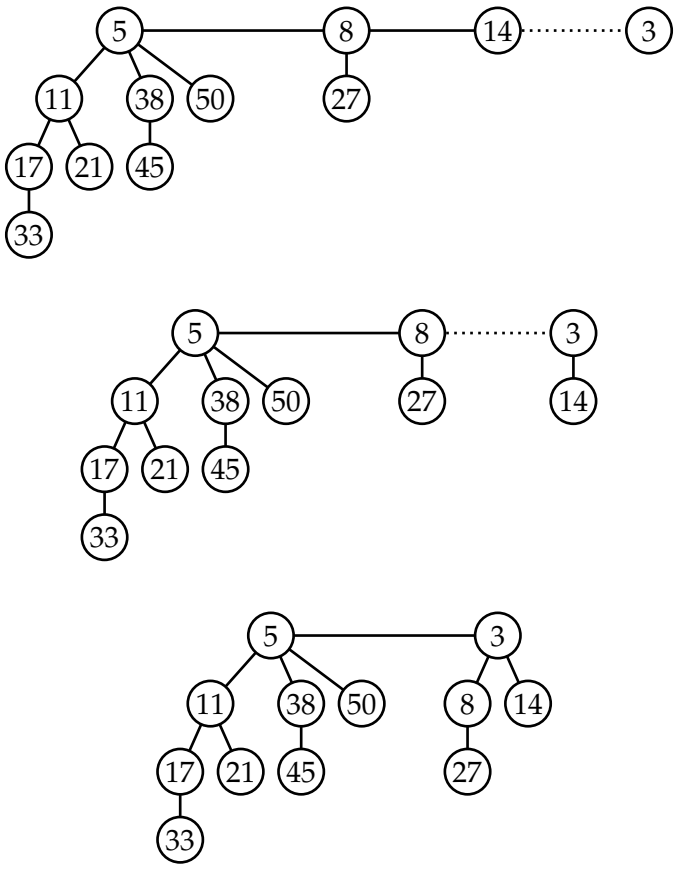

Figure 4: Insertion of a new node 3 to the binomial heap. The resulting heap is of size $n = 12$.

Observe that in the worst case, inserting into a n-size binomial heap will take $\log n$ LINK operations, so the worst case time complexity of INSERT is $O(\log n)$.

Notice, that this operation is actually analogous to incrementing a binary counter. Since a binomial heap of n elements is represented by trees corresponding to the set bits in the binary representation of n, and after the insertion there are $n+1$ nodes, the LINK operations being carried out correspond one-to-one to the resetting of trailing ones in a binary counter to zeroes while incrementing. This is another justification of the worst case cost of this operation being $O(\log n)$.

We can actually extend the analogy for computing the amortized cost for this operation. Recall, that we can show that the increment operation on a binary counter is amortized $O(1)$ by considering a potential function to be the number of set bits in the counter. This works, because during an expensive increment operation, the drop in potential is the number of trailing ones become zeroes, which exactly pays for the operation itself.

Similarly, in this case, define

$$\Phi(H) = t(H) = \text{number of trees in } H.$$

Denote by b_n the number of set bits in the binary representation of n, and t_n the number of trailing ones in the binary representation of n. We have, $b_{i-1} - b_i = t_{i-1} - 1$.

Then,

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= (t_{i-1} + 1) + b_i - b_{i-1} \\ &= (t_{i-1} + 1) - (t_{i-1} - 1) \\ &= 2\end{aligned}$$

Thus, INSERT is amortized constant time.

EXTRACT-MIN

Since each tree follows the min-heap property, we know that the overall minimum element in the heap must be one of the root elements. We can find this element in $O(\log n)$. (Alternatively, we could maintain a min pointer to always point to the minimum element in the heap, in which case this step can be done in $O(1)$).

Now, if we remove this node from the heap, all its children become free and form binomial trees themselves. In particular, if the deleted node is a root of a B_i tree, after deletion, we have new $B_0, B_1, ..., B_{i-1}$ trees.

At this stage, we have two (valid) binomial heaps, one consisting of all the trees originally other than the one whose root got deleted, and the other one consisting of the children of the deleted root tree.

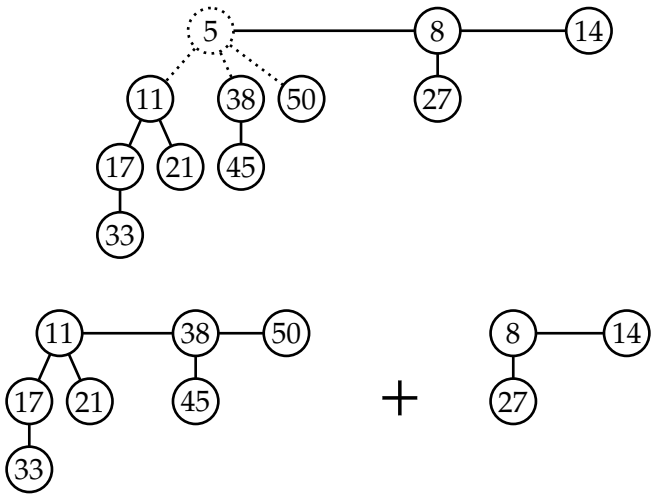

Figure 5: Deletion of the min key node (5) from the binomial heap

We need to *meld* these two binomial heaps into a single valid binomial heap.

Observe that just like inserting one element is analogous to incrementing a binary number, similarly, *melding* two binomial heaps is analogous to addition of two binary numbers (the sizes of the two heaps). We start from the smallest order (the lowest bit), if there are two trees of that order (both set bits), we link them (push a carry to the next bit). Then we go to the next order (next bit).

For any order k, there can be at most 3 trees of that order, if there are at least 2 trees, we link them and continue.

Thus, we can formulate the meld operation (simplified; assuming we keep track of nodes with next pointers as the root list):

MELD(H_1, H_2)

```
1  H = merge root lists of H1 and H2 by non-decreasing orders
2  if H.head == nil
3      return H
4  x = H.head
5  next = x.next
6  while next ≠ nil
7      if x.order ≠ next.order or (next.next ≠ nil and next.next.order == x.order)
8          x = next
9      else
10         x = LINK(x, next)
11     next = x.next
12 return H
```

By analogy to binary addition, this operation also works in worst case $O(\log n)$ time.

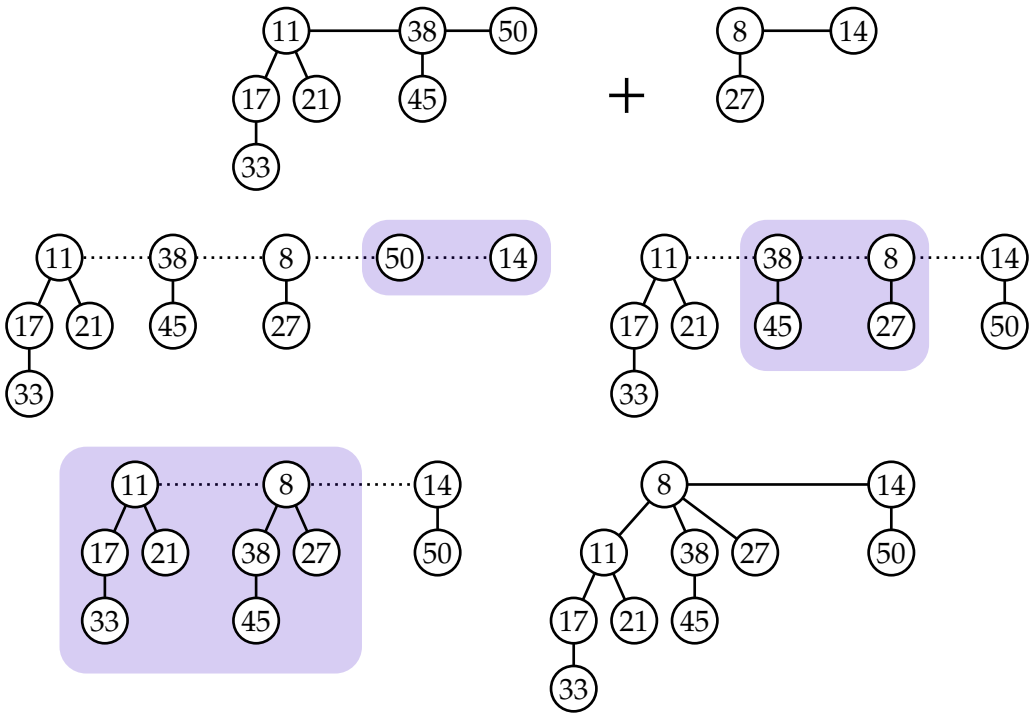

Figure 6: Union of two binomial heaps

DECREASE-KEY

This is simple to implement in this case. Assuming we have the pointer to the node, we can decrease the key of that node. Since we don't change any number of nodes anywhere, the only property to ensure is that trees must follow the min-heap property. So, after decreasing the key of the node, we can bubble it up to its appropriate position, to restore the min-heap property of that tree (exactly as is done in a binary heap). This operation runs in $O(\log n)$ worst case.

To summarize, here are the time complexities of the operations implemented:

	Amortized	**Worst case**
INSERT	$O(1)$	$O(\log n)$
DECREASE-KEY	$O(\log n)$	$O(\log n)$
EXTRACT-MIN	$O(\log n)$	$O(\log n)$

Table 1: Complexities of operations on a binomial heap

Towards our goal

As stated in the introduction, we want to reach a data structure which can support amortized constant time DECREASE-KEY operations, so we can obtain an improvement in the asymptotic complexity of some graph algorithms.

First, let's affirm that we can never get a data structure which supports all three operations in amortized constant time. This is because, such a data structure could be used to give a $O(n)$ generic sorting algorithm, which we know is not possible.

So for our target data structure, the amortized costs will be $O(1)$ INSERT, $O(1)$ DECREASE-KEY and $O(\log n)$ EXTRACT-MIN operations.

Let us think of how we can improve the cost of DECREASE-KEY from what we have in the binomial heap. The worst case $\log n$ time comes from the bubbling up which is required after decreasing the key to maintain the min-heap property. Also, we don't have any hope of making it constant amortized, since the user chooses which node to decrease-key from, so they may always choose a deep node which triggers the worst case. Nor can we bound the height of the tree any better than $O(\log n)$. So, we need to change the implementation itself.

Let's be lazy

To achieve $O(1)$ DECREASE-KEY, let us relax some of the restrictions on the structure of our heap.

For decreasing the key of an element, instead of bubbling it up the tree, let's chop it off the tree it is a part of, update the key, and add it as a new tree in the root list.

Clearly this is $O(1)$ (worst case)! But we have lost a lot of structure by doing this:
- The tree from which a subtree was severed is no longer a binomial tree.
- There may be more than one tree of the same degree after this operation.

Here, we refer to the degree of a root node of the tree as the degree of that tree. (We omit the usage of 'order' from here since the trees are no longer binomial trees).

While we're at it, since we are allowing multiple same-degree trees, we can make our INSERT operation worst case $O(1)$ as well; we simply add a new node to the root list, and don't bother about the merging steps.

The following figure shows the newly designed operations DECREASE-KEY(17, 10) and INSERT(2) performed sequentially on our original heap.

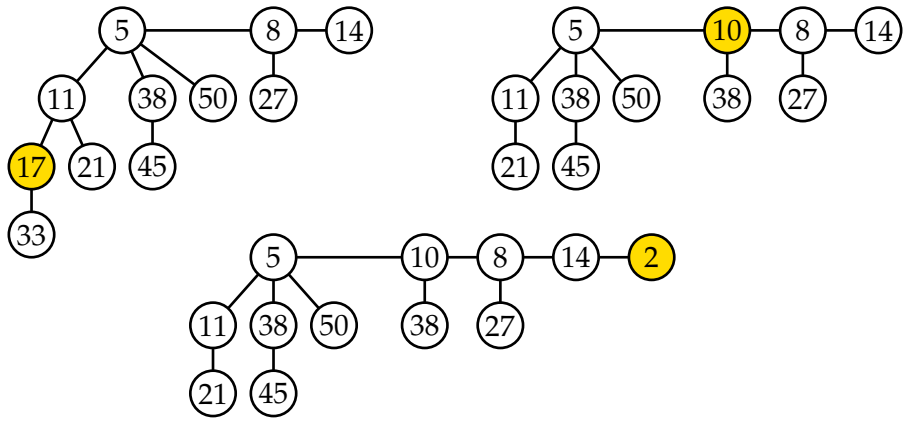

Figure 7: 1) DECREASE-KEY(17, 10) operation performed.
2) INSERT(2) operation performed.
The resulting trees are not binomial trees, also there are multiple trees of the same degree.

Alright, we seem to have both DECREASE-KEY and INSERT in $O(1)$ worst case time. Of course not for free, in return, we have completely destroyed any semblance of structure in the heap. We have pushed all the hard work to the EXTRACT-MIN, it needs to 'fix' our heap along with removing the minimum element.

Can we fit all the work that EXTRACT-MIN needs to do to restore structure in the heap, in $O(\log n)$ amortized time?

Restoring Structure

We want to restore the property that there is at most one tree of degree k for all k. How do we do this? We can do the following:

- While there are more than one tree of some degree k, link them.

What is the maximum degree a tree can be in our heap of n elements? In case of a binomial heap, this was clearly $\log n$, but that is not the case now. Let us call this maximum degree $D(n)$.

Formally, this is the high level algorithm we use to restore the one-tree property of our heap.

CONSOLIDATE(H)

```
1  let A[0..D(n)] be a new array
2  for each node w in the root list
3      x = w, d = x.d
4      while A[d] ≠ nil
5          y = A[d]
6          x = LINK(x, y)
7          A[d] = nil
8          d = d + 1
9      A[d] = x
10 recreate the root list of H with the trees in A, update the H.min pointer
```

We allocate an array of size $D(n)$ which is the maximum possible degree of a tree. At the end of the consolidation, there can be at most $D(n) + 1$ trees, each with a unique degree. We go through the root list, if we find a tree with degree d and also some other tree with degree d stored in the array, we link them (continuously till the array d slot is empty). If not, we just store that tree at $A[d]$.

This is how the EXTRACT-MIN procedure will work:

EXTRACT-MIN(H)

```
1  remove the minimum node from the rootlist
2  add all its children to the root list
3  CONSOLIDATE(H)
```

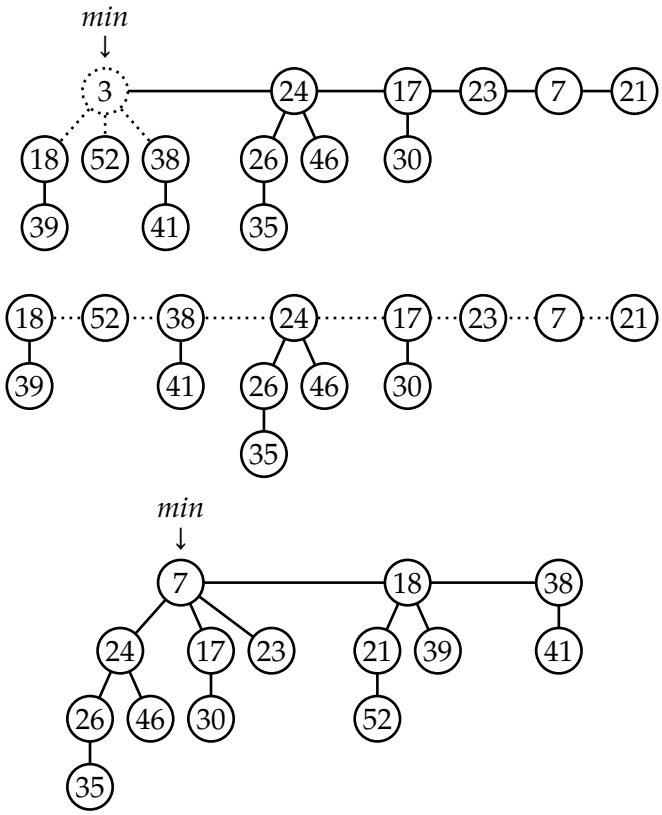

Figure 8: EXTRACT-MIN operation performed on a heap. Top: Initial state. Middle: State just after removing the minimum node. Bottom: Final heap after CONSOLIDATE. (For detailed steps of consolidation, see Figure 19.4, CLRS)

We need to find the time complexity of CONSOLIDATE. We will show that CONSOLIDATE runs in $O(D(n) + t(H))$ time. Firstly, lines 1 and 10 clearly run in $O(D(n))$ time. Note that when we call CONSOLIDATE, the number of nodes in the root list is bounded by $t(H) + D(n) - 1$. (originally $t(H)$ nodes in the root list, the min node removed, and all its children added to the root list).

Every time through the **while** loop, one of the root list nodes is linked with another. So, the total number of times the **while** loop can run is bounded by $t(H) + D(n)$. Therefore the actual running time of CONSOLIDATE is $O(t(H) + D(n))$.

Before talking about $D(n)$ let us consider $t(H)$. In case of a binomial heap, the maximum number of trees in a heap of n elements was $\log n$. In this case it is not true, in fact in the worst case we can have $O(n)$ trees, e.g. after n INSERT operations. Then the worst case cost of this operation becomes $O(n)$ which is undesirable.

Can we get a better amortized bound?

Consider the potential function $\Phi(H) = t(H)$. Here the expensive operation is EXTRACT-MIN which decreases the number of trees. And the cost of this operation is compensated by the drop in the potential (i.e. decrease in the number of trees!).

Formally,

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= t(H) + D(n) + D(n) - t(H) \\ &= O(D(n))\end{aligned}$$

So, our EXTRACT-MIN function is $O(D(n))$ amortized!

Almost there...

If we can somehow show $D(n) = O(\log n)$, we are done. But, as per our current structure, we can't claim this.

One can give a sequence of operations which leaves us with a 'wide' and 'short' tree, in which the max degree of the tree can become $O(n)$.

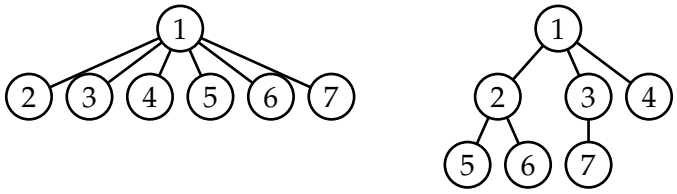

Figure 9: Left: a 'wide' and 'short' tree. Right: a 'deep' and 'bushy' tree

How can we add some structural property to our trees to prevent them from becoming wide and short due to excessive node removals? Since we want to bound the degree of the tree to be logarithmic on the number of nodes, equivalently, we need to ensure that the **number of nodes in the tree is exponential in its degree**.

Recall, in a binomial tree we had the property that the ith child had a degree of $i-1$, (1-indexed). Inductively, this property ensures that the number of nodes in B_k is exponential in k (specifically, 2^k). Clearly, we don't have this property now as we allow severing nodes from anywhere in the tree.

Let's instead require that in our new version of the heap (which we will see to be the Fibonacci Heap), the ith child must have degree **at least** $i-2$, i.e. one less that that in a binomial heap.

This means we need to enforce the following:
- from any node, at most one child can be deleted.

To ensure this, whenever we have already removed a child from a node, we **'mark'** it. When we try to remove another child from a marked node, we also sever the marked node itself from its parent and add it to the root list. **This ensures our $i-2$ property! (Why?)**

Now, due to the $i-2$ property, the number of nodes in a tree is exponential to its degree, which means the maximum degree of the heap is logarithmic in n, i.e. $D(n) = O(\log n)$. (We show this formally later.)

Now we have gotten our $O(\log n)$ amortized bound for EXTRACT-MIN operations! Are we done? Not just yet... We need to reanalyse our DECREASE-KEY function since we may need to do additional cuts to ensure the $i-2$ property. Let's bring everything together and complete our analysis.

Fibonacci Heaps!

A Fibonacci Heap is a collection of rooted trees that follow the **min-heap property**, and a *min* pointer which points to the minimum node in the root list. Each node's *degree* is defined as the number of children of that node. Also, each node has a boolean property *marked*, which represents whether any (direct) child of that node has been removed or not.

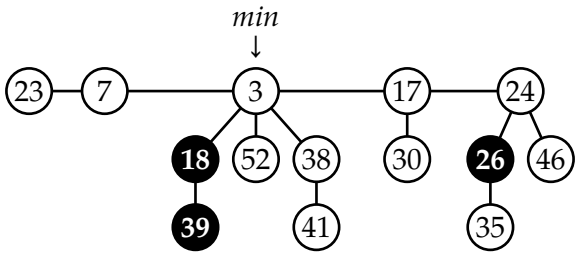

Figure 10: A fibonacci heap. Marked nodes are indicated with a dark background.

Let us look at each operation one by one.

INSERT(x)

Pretty simple – a new node is added to the root list of the heap, update the min pointer if required. Worst case $O(1)$ cost.

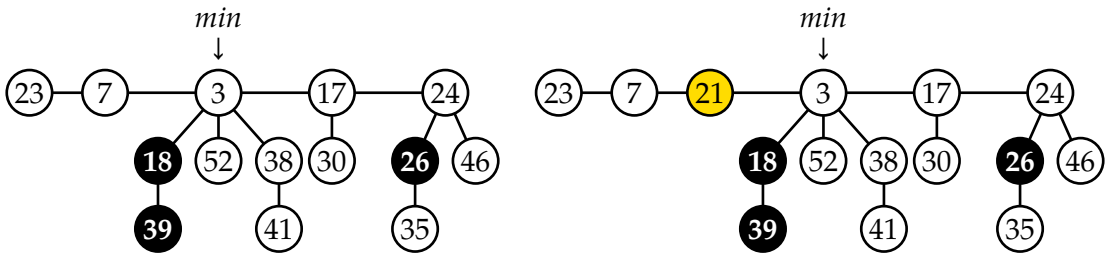

Figure 11: Running INSERT(21) on the Fibonacci heap.

DECREASE-KEY(x, k)

1. If we can decrease the key without violating the min-heap property, we do so.
2. Otherwise, we decrease the key, and cut the node from the tree, and add it to the root list. (Set *marked* for this node to be false.)
3. If the (non-root) parent was unmarked, then mark it. Else, cut the parent from the tree, add it to the root list, unmark it. Repeat Step 3.

DECREASE-KEY(H, x, k)

```
1  ensure k > x.key
2  x.key = k
3  y = x.parent
4  if y ≠ nil and x.key < y.key
5      CUT(H, x, y)
6      CASCADING-CUT(H, y)
7  if x.key < H.min.key
8      H.min = x
```

CUT(H, x, y)

```
1  remove x from child list of y
2  add x to the root list of H
3  set x.marked to false
```

CASCADING-CUT(H, y)

```
1  if y is a root node return
2  z = y.parent
3  if not y.marked
4      y.marked = true
5  else
6      CUT(H, y, z)
7      CASCADING-CUT(H, z)
```

The following figures show the operations DECREASE-KEY$(46, 15)$ (a-b) and DECREASE-KEY$(35, 5)$ (b-c-d-e).

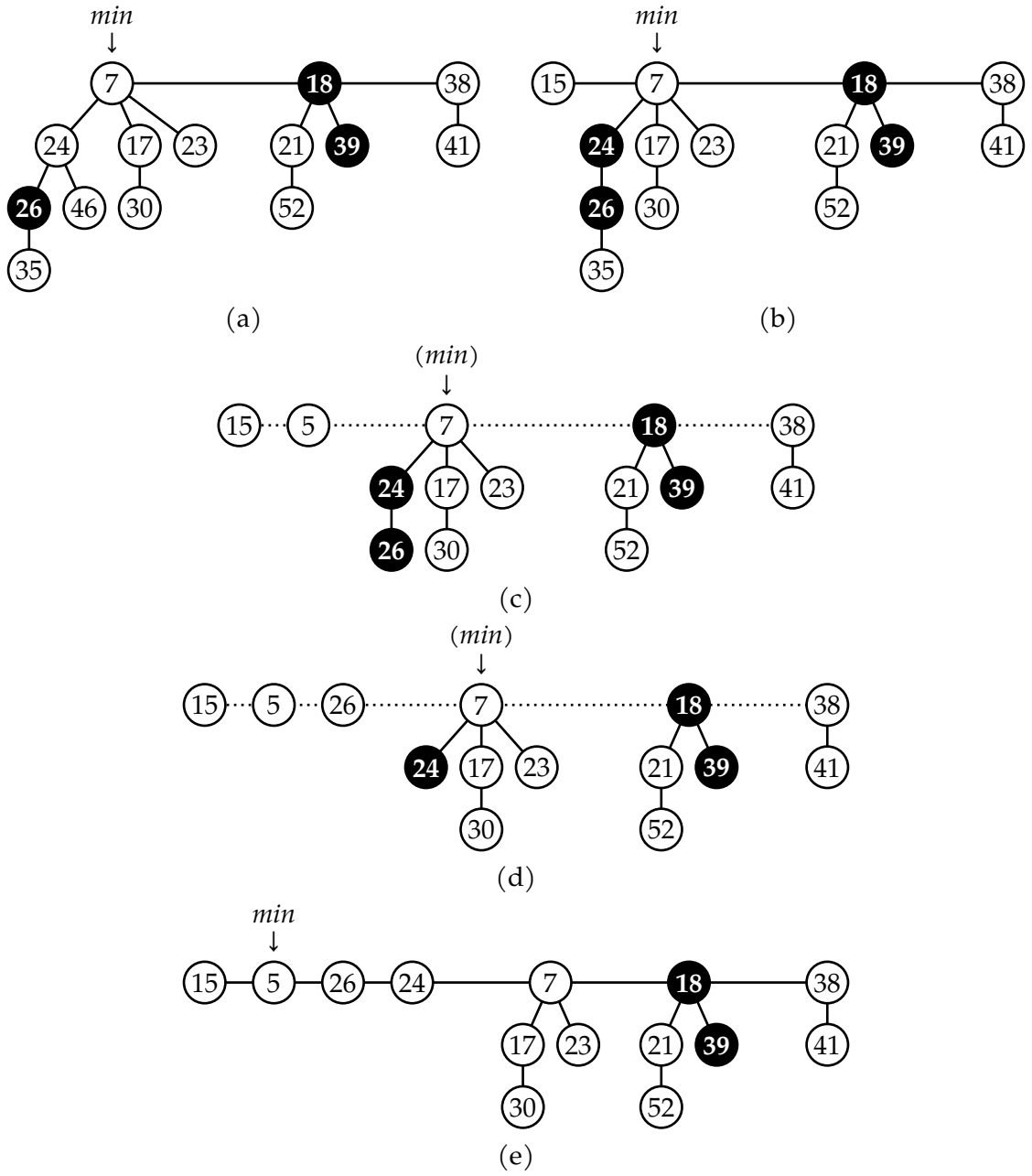

Figure 12: Two DECREASE-KEY operations performed on a Fibonacci Heap.
$b \to c \to d \to e$ shows how the cuts are cascaded when we have a chain of marked nodes.

What is the time complexity of this operation? It is proportional to the number of cascading cuts, which is bounded by the height of the tree. But, the height of the tree can be $O(n)$ in the worst case (See Problem 4). Therefore the worst case time complexity for DECREASE-KEY is $O(n)$.

How do we amortize the cost down to $O(1)$?

Earlier, for analysing the EXTRACT-MIN operation we had used the potential function $\Phi(H) = t(H)$. However this doesn't work here. Instead of the potential decreasing, it *increases* (as there are new trees being added to the root list). In fact, it increases by the number of cuts that were done.

If our potential function was $m(H)$ = number of marked nodes in H, then this will work, since the decrease in the number of marked nodes is exactly the number of cascading cuts that took place, which is the actual cost of the operation.

So, to support all operations, let us choose this potential function:

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

During an expensive DECREASE-KEY operation, the number of trees increases, but the number of marked nodes decreases (by the same amount), and this decrease in potential pays for the actual cost of the operation.

Consider a DECREASE-KEY operation where c cuts take place. Then

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &\leq c + (t(H) + c) + 2(m(H) - c + 2) - t(H) - 2m(H) \\ &= 4\end{aligned}$$

So, DECREASE-KEY is $O(1)$ amortized.

EXTRACT-MIN

The algorithm is already discussed above, which involves removing the min element, and running CONSOLIDATE. After running this operation, our heap will have at most one tree per degree.

See Figure 19.4, CLRS for a demonstration of the working of EXTRACT-MIN and CONSOLIDATE.

Let $D(n)$ denote the maximum possible degree of a tree in a Fibonacci heap containing n elements.

We know that the actual running time of EXTRACT-MIN is $O(D(n) + t(H))$. By the potential defined above ($\Phi(H) = t(H) + 2m(H)$), we can find the amortized cost of EXTRACT-MIN:

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &\leq (D(n) + t(H)) + (D(n) - 2m(H)) - (t(H) - 2m(H)) \\ &= O(D(n))\end{aligned}$$

Bounding $D(n)$

Lemma 1: Let x be a node in a Fibonacci heap with degree k and children $y_1, y_2, ..., y_k$ (in order in which they were linked to x). Then, the degree of y_i is at least $i - 2$.

Proof: When y_i was linked to x, then degree(x) was $i - 1$. We know that two trees can be linked only if they have the same degree, so at that time y_i also must have had $i - 1$ degree. By the structural property of the Fibonacci heap, we know that y_i cannot have lost more than one of its direct children, otherwise it would have been cut off from x. Therefore, degree$(y_i) \geq i - 2$.

Lemma 2: Let x be a node in a Fibonacci heap with degree k. Then, size$(x) \geq F_{k+2}$, where F_i is the ith Fibonacci number.

Proof: Let s_k denote the minimum possible size of any node with degree k in a Fibonacci heap. ($s_0 = 1, s_1 = 2$). Clearly, s_k is monotonically increasing.

If we are forming a minimal tree of degree k, its children must follow Lemma 1, so the ith child itself must be a minimal tree of degree $i - 2$. Based on this observation, we can write the recurrence:

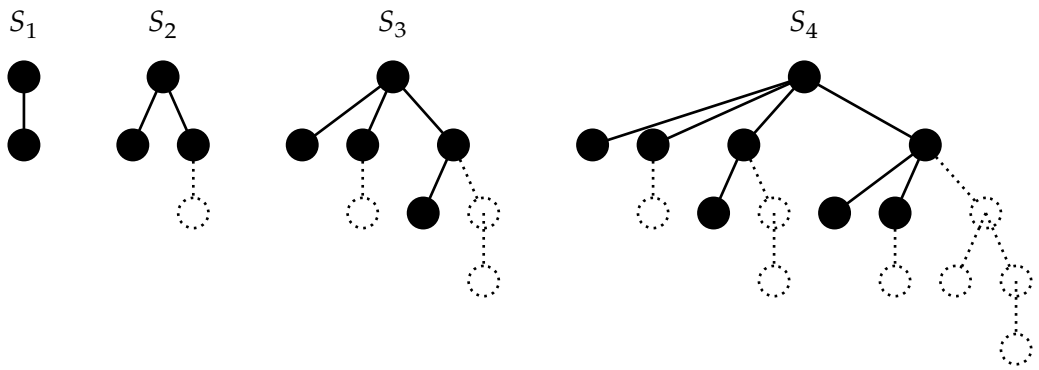

Figure 13: Minimal node trees of different degrees in a Fibonacci heap

$$s_k = 1 + \sum_{i=1}^{k} s_{i-2}$$

which can be rewritten as

$$s_k = s_{k-1} + s_{k-2}$$

Along with $s_0 = 1$ and $s_1 = 2$, we have $s_k = F_{k+2}$, i.e. the $(k+2)$th Fibonacci number.

Thus, size$(x) \geq s_k = F_{k+2}$, which completes the proof.

Lemma 3: $F_{k+2} \geq \phi^k$. (Can be proven by induction.)

So together, we have the property that for any node x with degree k, size$(x) \geq \phi^k$.

Corollary: The maximum degree $D(n)$ of any node in a n element Fibonacci heap is $O(\log n)$.

Proof: Let x be a node in the Fibonacci heap of degree k. Clearly, $n \geq \text{size}(x) \geq \phi^k \Rightarrow k \leq \log_\phi n$. Therefore the maximum degree $D(n)$ is $O(\log n)$.

Now that we have established that $D(n)$ is $O(\log n)$, it follows that the amortized cost of EXTRACT-MIN is $O(\log n)$.

To summarize, these are the costs of the different operations that we defined for the Fibonacci Heap:

	Amortized	**Worst case**
INSERT	$O(1)$	$O(1)$
DECREASE-KEY	$O(1)$	$O(n)$
EXTRACT-MIN	$O(\log n)$	$O(n)$

Table 2: Complexities of operations on a Fibonacci heap

Problems

1. Analyse the running time of the Dijkstra's algorithm for single source shortest path and Prim's algorithm for computing a minimum spanning tree in a graph when the priority queue being used in a Fibonacci Heap instead of a Binary Heap. Assume that the number of vertices in the graph is n and the number of edges m.

Dijkstra's and Prim's algorithms require n EXTRACT-MIN operations and m DECREASE-KEY operations. Since their amortized costs are $O(\log n)$ and $O(1)$ respectively, the total running time of the algorithm is bounded by $O(n \log n + m)$.

2. Suppose in the Fibonacci Heap, we allowed an operation INCREASE-KEY, which takes as input the pointer to a node in a Fibonacci Heap H and a new key value k, and changes x.key to k. Note that k could be larger than, equal to, or less than the current key value. Design an efficient algorithm to perform this operation so that the amortised time complexity of this operation is $O(\log n)$ without changing the amortised complexity of other heap operations.

If the new key is greater than the current key, we can perform the following sequence of operations:

- DECREASE-KEY$(H, x, -\infty)$ ($O(1)$ amortized)
- EXTRACT-MIN(H) ($O(\log n)$ amortized)
- INSERT(H, x, k) ($O(1)$ amortized)

which runs in amortized $O(\log n)$ time.

3. Suppose that we generalise the cascading-cut rule to cut a node x from its parent as soon as it loses its kth child, for some integer constant k. For what value of k, we have $D(n) = O(\log n)$? What are the running times of the standard operations on Fibonacci heaps with this new definition? Give an argument with the accounting method for the amortized costs of each operation.

Let s_m denote the minimum possible size of a node in a heap with degree m (assuming the generalised cascading-cut rule). Clearly, s_m is monotonically nondecreasing. ($s_0 = 1, s_1 = 2$)

Lemma: For a node x with degree m, its ith child (in order of linking) must have at least $i - k$ children.
Proof: Let its children be $y_1, y_2, \ldots, y_m$. When y_i was linked to x, it must have had $i - 1$ children. By the generalised cascading rule, after linking, y_i can have lost at most $k - 1$ of its direct children. So, it has at least $i - k$ children. (The rank of y_i among the children of x (in order of linking) can change, it can decrease, but it can never exceed i, so the bound still holds.)

Then, a minimal size node with degree m's ith child must itself be a minimal size node of degree $i - k$. Thus we form the following recurrence:

$$s_m = 1 + \sum_{i=1}^{m} s_{i-k}$$

(Take $s_i = 1$ for $i \leq 0$, since size$(z) \geq 1$ for any node.)

So,

$$s_m - s_{m-1} = s_{m-k}$$

The characteristic equation is

$$x^k - x^{k-1} - 1 = 0$$

There exists a real root $\omega, 1 < \omega \leq 2$. So, $s_m = \Omega(\omega^m)$. Clearly, $n \geq s_m = \Omega(\omega^m)$, thus $m = O(\log n)$.

So, for any $k > 1$, we have $D(n) = O(\log n)$.

($k = 1$ does not work as DECREASE-KEY won't be amortized constant time anymore, since every DECREASE-KEY operation will trigger long cascading cuts.)

Accounting for amortized costs: For each non-root node v, let $\ell(v) \in \{0, 1, ..., k-1\}$ be the number of children v has lost since becoming a child of its current parent; v is cut from its parent (and rejoins the root list with $\ell(v)$ reset to 0) as soon as $\ell(v)$ would reach k. Let $L(H) = \sum_v \ell(v)$ over non-root nodes, and $t(H)$ the number of root-list trees. Maintain credits so that the total stored credit always equals

$$\Phi(H) = t(H) + \frac{2}{k-1} L(H).$$

(Credits stored are 1 credit on each root node, and $2\ell/(k-1)$ credits on each non root node.)

- INSERT: charge 2 units. 1 pays for the cost of insertion, plus 1 credit stored on the new root.
- DECREASE-KEY: charge at most $2 + \frac{2}{k-1}$ units.
  Suppose the operation triggers c cascading cuts beyond the mandatory cut of x. Actual cost is $1 + c$. Each cascaded node had $\ell = k - 1$ just before being cut and resets to 0 (so they stored 2 credits each, one credit pays for the cut, the other stays as the credit for the new root node).
  Out of the $2 + \frac{2}{k-1}$ units charged, 1 unit pays for the cut of x (x may not hold any credits), 1 unit for the credit of the new root node (x), and $\frac{2}{k-1}$ to be stored as credit on the node whose ℓ was increased by 1 (if any).
- EXTRACT-MIN: charge $2D(n) + 1 = O(\log n)$ units.
  CONSOLIDATE requires $D(n) + t(H)$ actual cost, which is paid for by $D(n)$ from the charge, and $t(H)$ from credits stored on the root nodes. After this operation, the total number of trees is at most $D(n) + 1$, the credits on these root nodes are restored by the remaining units charged $(D(n) + 1)$.

Since we store 1 unit of credit on each root node, and fractional credits $(2\ell/(k-1))$ on each non root node, the total credits is always nonnegative. Hence, these charges suffice to pay for a sequence of n operations.

So for any constant $k \geq 2$, the generalized Fibonacci heap has exactly the same amortized complexities as the standard heap: $O(1)$ INSERT, $O(1)$ DECREASE-KEY, $O(\log n)$ EXTRACT-MIN.

4. Show that the height of a tree in an n-node Fibonacci heap could be $n - 1$.

We show, by induction on m, that there is a sequence of Fibonacci-heap operations producing a heap consisting of exactly one tree, a chain with m nodes (height $m - 1$).

Base case: C_1 requires a single INSERT. For C_2, INSERT two nodes p, q. Insert a dummy node with key $-\infty$, and call EXTRACT-MIN. CONSOLIDATE links p and q into a chain of two nodes.

Inductive step $(C_m \to C_{m+1})$**:** Suppose C_m is a chain with root equal to c.

1. Insert two new nodes p, q with keys smaller than c $(p < q < c)$, then insert a dummy node with key $-\infty$ and call EXTRACT-MIN: CONSOLIDATE links p, q (both degree 0) into a 2-node chain P (root p, child q), and then links P (degree 1) with C_m (degree 1). Since p has the smallest key overall, p becomes the new root, with two children: its old child q, and the root of C_m.
2. Call DECREASE-KEY$(q, -\infty)$ and EXTRACT-MIN. This removes node q from the heap.

The heap is now exactly C_{m+1}: a linear chain of $m + 1$ nodes.

Thus for any $n \geq 0$ we can form a Fibonacci heap with n nodes with height $n - 1$.

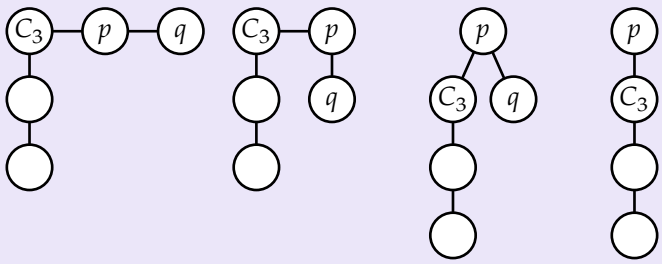

Figure 14: Inductive construction of C_4 from C_3

5. What is the maximum possible degree of an n-node Fibonacci heap? Given any natural number n, provide an example of a sequence of operations so that the Fibonacci heap has n nodes after this sequence of operations and there is a node of the maximum possible degree of an n-node Fibonacci heap. Prove your claim.

The maximum possible degree of an n-node Fibonacci heap is $D(n) = \lfloor \log_\phi n \rfloor$; this upper bound was shown in the Corollary above. We now show a construction to achieve it.

For each $k \geq 0$ we construct a tree S_k with **exactly** F_{k+2} nodes whose root has degree k:

1. Build a binomial tree (B_k) of degree k by a sequence of INSERT and EXTRACT-MIN operations. $(\star)$
2. From every non root node, select its highest degree child. On all these selected nodes, run DECREASE-KEY$(x, -\infty)$ and EXTRACT-MIN.

We **claim** that the resulting heap S_k has exactly F_{k+2} nodes with degree k. We can prove it as follows:

Lemma: Let B_m be a non root node with degree m before Step 2. After Step 2, B_m has exactly F_{m+1} nodes.

Proof by strong induction on m:

Base:
- $k = 0$: B_0 is a single node, $F_1 = 0$.
- $k = 1$: B_1 has two nodes, after removing its largest child, it is left with one. $F_2 = 1$.

Step: $m \geq 2$
Originally, the children of B_m were $B_{m-1}, B_{m-2}, \ldots, B_0$. In step 2, we remove its largest degree child, B_{m-1}. All the other children B_j get transformed by Step 2, and are finally left with F_{j+1} nodes. (Inductive Hypothesis)

Then, B_m is left with $1 + \sum_{j=0}^{m-2} F_{j+1} = 1 + \sum_{i=0}^{m-1} F_i = F_{m+1}$ nodes. ■

Proof of claim: Originally the root node was B_k with children $B_{k-1}, \ldots, B_0$. After Step 2, they have nodes $F_k, \ldots, F_1$. So total nodes in S_k is

$$1 + \sum_{i=1}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$

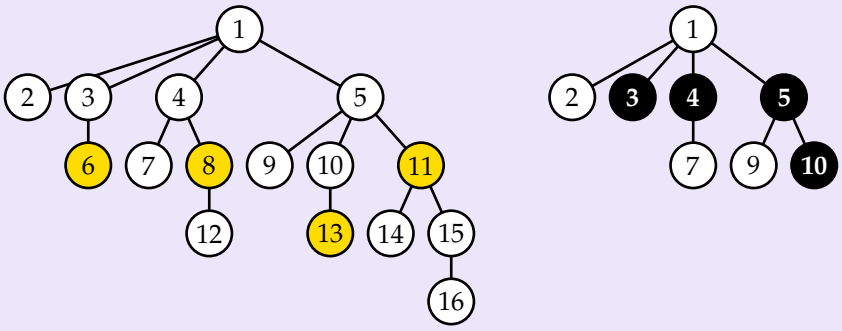

Figure 15: Transformation of B_4 into S_4 with $F_6 = 8$ nodes by Step 2

Given n, let $k = \lfloor \log_\phi n \rfloor$, the largest k with $F_{k+2} \leq n$. Build S_k as above, then perform $n - F_{k+2}$ INSERT operations. We get an n-node Fibonacci heap containing a node of degree $k = D(n)$.

Construction for $\star$:

- $k = 0$: INSERT
- $k = 1$: INSERT, INSERT, INSERT$(-\infty)$, EXTRACT-MIN
- $k \geq 2$: Build(B_{k-1}), Build(B_{k-1}), INSERT$(-\infty)$, EXTRACT-MIN. (CONSOLIDATE will link the two B_{k-1}s to form a B_k).

6. Prove that every node of degree k has at least F_{k+2} nodes in the subtree rooted at that node. Here, F_n denotes the n-th Fibonacci number $(F_0 = 0, F_1 = 1)$.

Let x be a node of degree k with children $y_1, \ldots, y_k$ in the order they were linked to x, and let s_k denote the minimum possible size (number of nodes in the subtree, including the root) of any degree-k node in a Fibonacci heap, so $s_0 = 1$, $s_1 = 2$, and s_k is monotonically nondecreasing in k.

By Lemma 1, degree$(y_i) \geq i - 2$. Since s is monotonic, size$(y_i) \geq s_{i-2}$ (taking $s_j = 1$ for $j \leq 0$). Counting x itself,

$$s_k = 1 + \sum_{i=1}^{k} s_{i-2}.$$

Subtracting the same relation for s_{k-1} gives $s_k - s_{k-1} = s_{k-2}$, i.e. $s_k = s_{k-1} + s_{k-2}$, the Fibonacci recurrence. Together with $s_0 = 1 = F_2$ and $s_1 = 2 = F_3$, this gives $s_k = F_{k+2}$ for all $k \geq 0$.

Since any degree-k node x has size$(x) \geq s_k$ by definition of s_k as the minimum, we conclude size$(x) \geq F_{k+2}$.

7. What is the maximum number of cascading cuts possible in a single decrease key operation on any n-node Fibonacci heap? Given any natural number n, provide an example of a sequence of operations with the last operation of the sequence being a decrease key operation and that decrease key operation performs the maximum number of cascading cuts possible on any n-node Fibonacci heap. Prove your claim.

Assuming that we don't count the base cut as a cascading cut.

Claim: The maximum number of cuts in a single DECREASE-KEY call on an n-node heap is $n - 2$.

Let d be the depth of x, so $d \leq n-1$. CASCADING-CUT walks up the ancestor chain $p_1 = y, p_2, ..., p_d =$ root: it cuts p_i only if p_i is marked (and not a root), and stops once it reaches the root. So at most $p_1, ..., p_{d-1}$ can be cut, i.e. $d - 1 \leq n - 2$ cascading cuts.

To achieve this bound we need to construct a linear chain of n nodes where all nodes except the root and the leaf node are marked.

Let us inductively construct a tree C_m which has a degree of 2, one of its children is a linear chain of $m-1$ marked nodes and one leaf node. The other child is a extra node e_m with key ∞.

Base:
C_1: INSERT(p), INSERT(∞), INSERT$(-\infty)$, EXTRACT-MIN. We have a chain of $p \to \infty$. Now INSERT(q), INSERT(q'), $p < q < q'$, INSERT$(-\infty)$, EXTRACT-MIN. Our root node is p and it has two children, q and ∞. Remove q' by DECREASE-KEY$(q', -\infty)$, EXTRACT-MIN.

Step: Let the key of the root of C_{m-1} be c.
- INSERT(p), $p < c$, INSERT$(e_m = \infty)$. Add a dummy node with key $-\infty$, then EXTRACT-MIN to link p and e.
- INSERT(q), $q > p$, INSERT$(e' = \infty)$. Add a dummy node with key $-\infty$, then EXTRACT-MIN to link q and e', and then p and q.
- Both trees rooted at p and C_{m-1} have degree 2. So trigger a linking by adding a $-\infty$ node and calling EXTRACT-MIN. C_{m-1} will be linked to p.
- DECREASE-KEY$(e_{m-1}, -\infty)$, EXTRACT-MIN. This marks the original root of C_{m-1}.
- DECREASE-KEY$(q, -\infty)$, EXTRACT-MIN to cleanup and remove the additional q node.

Thus we have formed C_m.

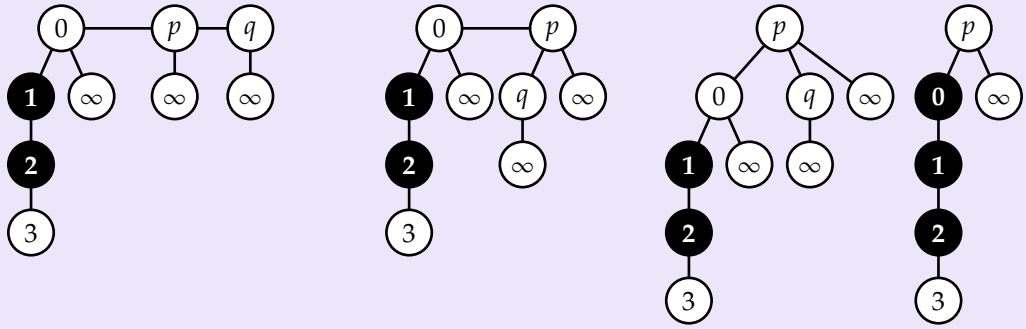

Figure 16: Inductive construction of a chain of marked nodes

To obtain the final tree from C_n, call DECREASE-KEY$(e_n, -\infty)$ and then EXTRACT-MIN. Then, we will be left with a single linear chain of n nodes where all but two nodes are marked. In this tree, calling DECREASE-KEY on the leaf node will trigger $n-2$ cascading cuts, completely breaking down the tree into individual nodes.

8. What is the maximum number of root nodes possible in an n-node Fibonacci heap at any point in time? What is the maximum number of root nodes possible in an n-node Fibonacci heap immediately after the EXTRACT-MIN operation? Justify your answers.

At any point in time, the maximum number of root nodes possible is n, which can be achieved by simply performing a sequence of n INSERT operations.

Immediately after an EXTRACT-MIN operation, the maximum number of root nodes possible is $D(n) + 1 = \lfloor \log_\phi n \rfloor + 1$. This is because,
1. After the CONSOLIDATE operation, there is at most one root node with degree k for any k.

2. $D(n)$ is the maximum possible degree, which can be at most $\lfloor \log_\phi n \rfloor$.
