Fibonacci Heap

Advanced Data Structures and Algorithms

D. Costa[1] M. Couto[1] R. Tuna [1]

[1]Faculty of Engineering
University of Porto

6, March 2023

Table of Contents

Table of Contents

Heaps

Heap - tree based data structure that respects the **heap rule**.

Heap Rule

Given 2 nodes **p** and **c**, if p is c's parent then p's value is smaller than or equal to c's value (in the case of a min-heap)

The Heap Rule ensures that the root of the tree is the minimum element, making heaps ideal for the removal or access of the minimum element.

The First Heap

The **Binary Heap** was the first heap. It was invented to be used as a priority queue: a queue where the element's position is related to its value rather than the order of insertion.

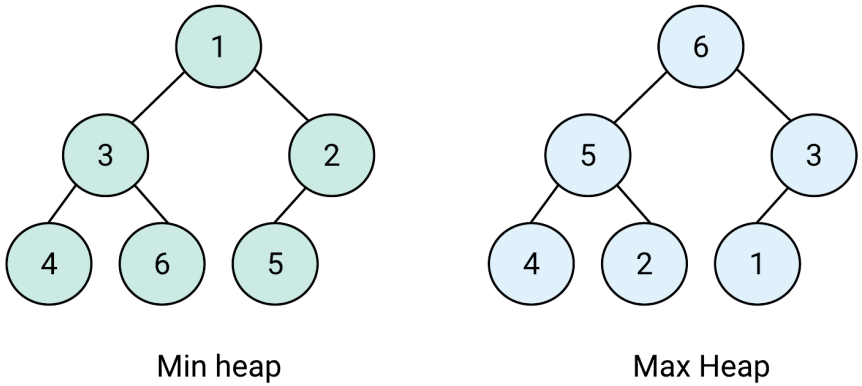


Figure 1: Binary Heap[1]

The Binary Heap is a Binary Tree that follows 3 invariants:

- Every level of the tree is full (last level from left to right)
- The Heap Rule
- Every node has a maximum of 2 children (it is a Binary Tree)

[1] https://guides.codepath.com/compsci/Heaps

Fibonacci Heap Motivation I

The Binary Heap has logarithmic complexity on **Insert** and **Decrease-Key** operations because they are required to enforce a certain structure on the tree.

Operation	**Complexity**
Get-Min	$\Theta(1)$
Insert	$O(log(N))$
Extract-Min	$\Theta(log(N))$
Decrease-Key	$O(log(N))$
Delete	$O(log(N))$

Table 1: Binary Heap Time Complexities

- What if there was not such a rigid structure?
- Could we achieve constant time on Insert and Decrease-key without compromising the other operations' complexity?

Fibonacci Heap Motivation II

The **Fibonacci Heap** (or F-Heap for short) was invented in 1987 [1] and it aimed to **reduce the time complexities** of the operations mentioned before for the Binary Heap.

Operation	**Complexity**
Get-Min	$\Theta(1)$
Insert	$\Theta(1)$
Extract-Min	$O(log(N))$
Decrease-Key	$\Theta(1)$
Delete	$O(log(N))$

Table 2: Fibonacci Heap Goal Time Complexities

Table of Contents

Key Concepts

Rank/Degree of a tree (d) number of children of the root node

Amortized Analysis a style of analysis where the time considered is the one measured from the average of a sequence of operations rather than the worse case (allowing for some alternative techniques)

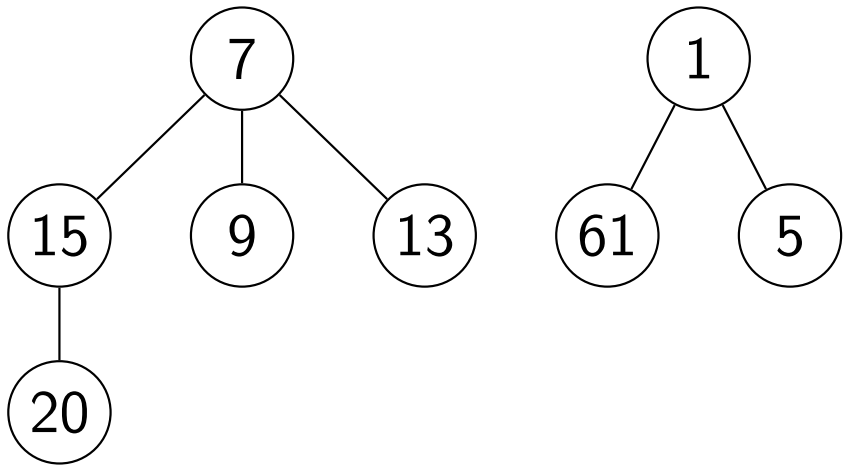
Binomial Trees a single node tree or a tree formed by merging two other trees with the same rank

F-Heap Structure and Get-Min

Get-Min(H)

- Return the item referenced to as the min root.
- **Time Complexity:** $\Theta(1)$

Fibonacci Heaps are structured as a **linked list of trees**. These trees, not only are not binary, trees but also only respect one of the other **invariants** imposed to Binary Heaps: the **heap rule**.



This structure is meant to facilitate the insertion of new nodes.
The minimum is one of the roots, and a reference/pointer to it is kept.

Merge

Merge(T1, T2)

- Merge operation of two trees performed by making the root of the tree with greatest root a child of the other tree's root
- **Time Complexity:** $\Theta(1)$

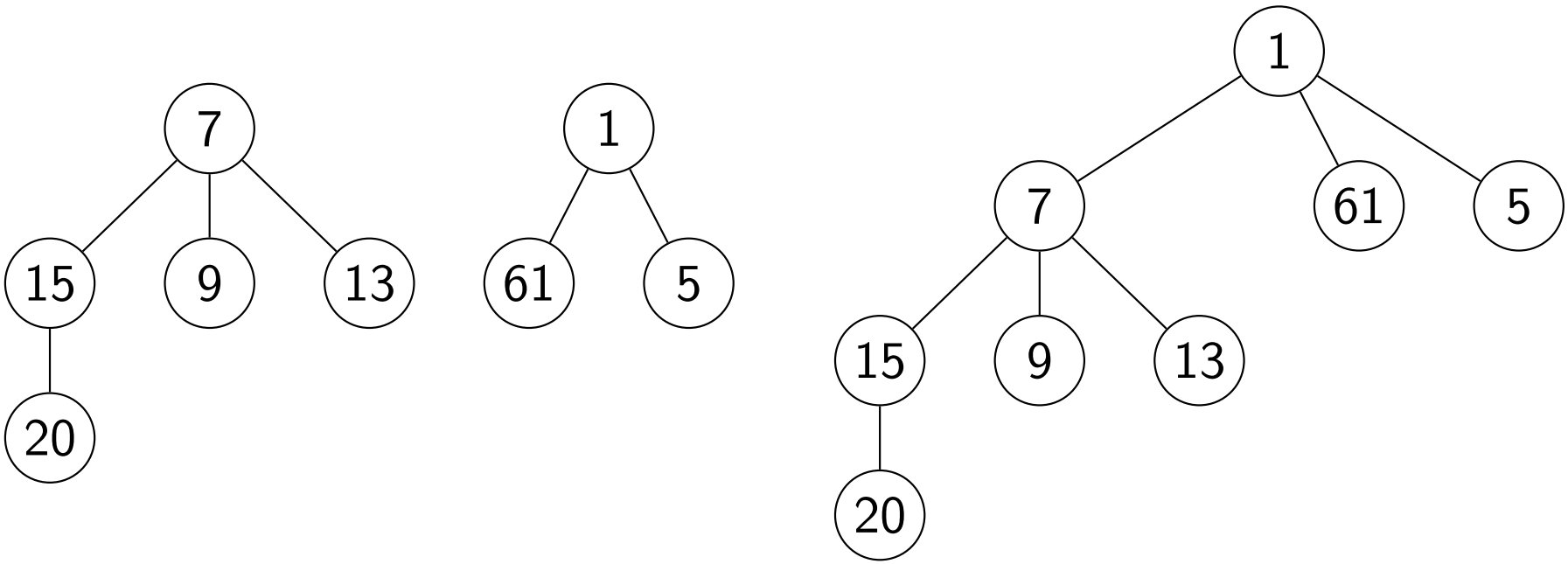


Table of Contents

Insert

Insert(H, X)

- Insert node at the root list and update *H.min* if its key is lower than the current one.
- **Time Complexity:** $\Theta(1)$

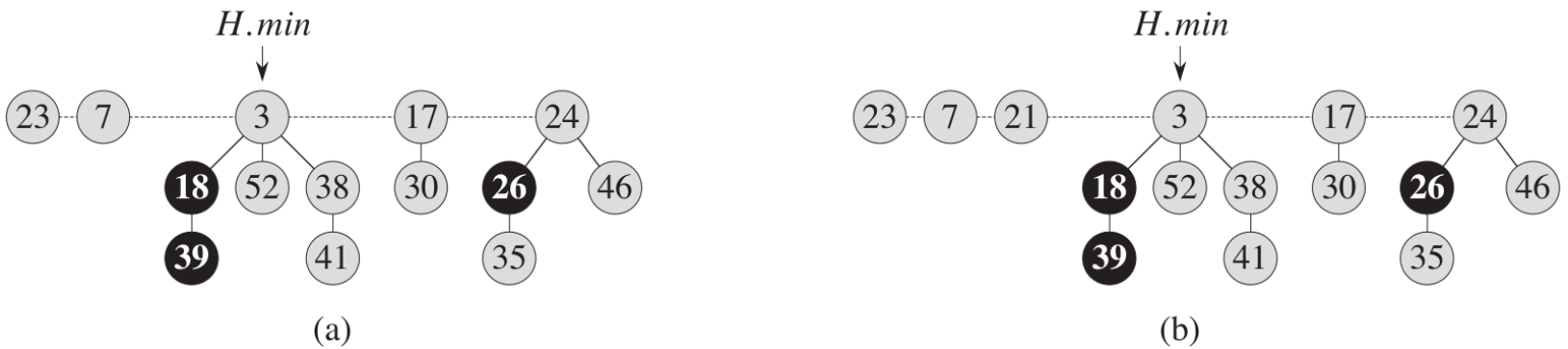


Figure 2: Inserting node with *key* = 21 into the F-Heap H. [2] **(a)** Before **(b)** After

Remark: If after F-Heap creation, only insert operations are performed, the F-Heap will essentially be a Doubly linked list with a pointer to its minimum, *H.min*.

Extract-Min I

- Idea: Promote the minimum's children to root nodes and remove it.
- Problem: Increasing number of trees in the F-Heap.
- Solution: **Consolidate** the F-Heap.

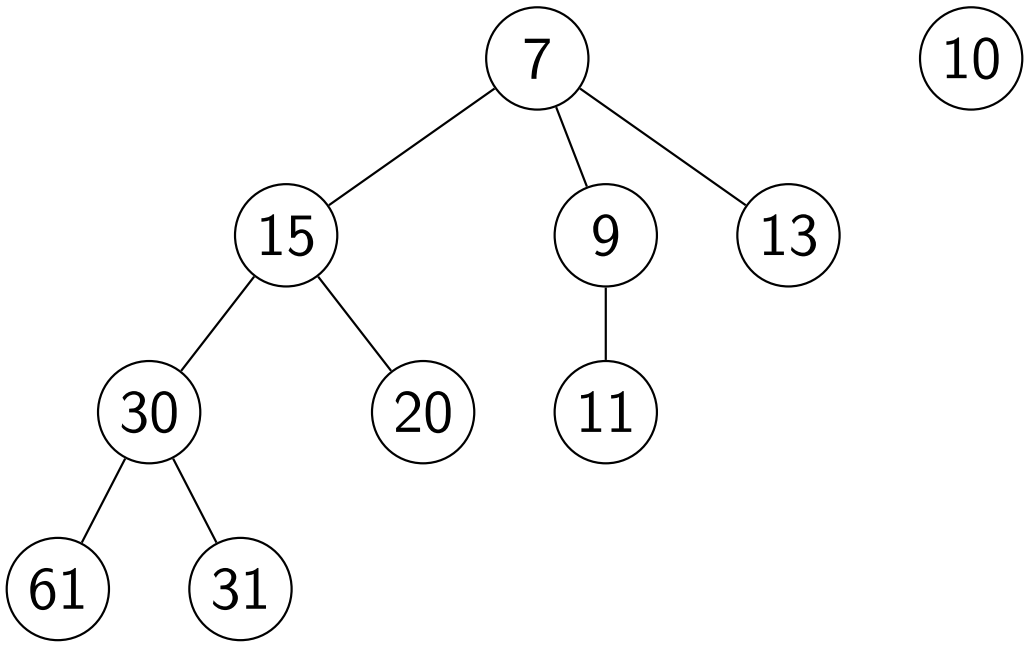
Extract-Min II

Extract-Min(H)

- Make a root out of every children of the minimum node and **remove the minimum** from the F-Heap, keeping a reference to it.
- **Consolidate**:
  - Initialize array for all ranks of trees up to the D(n) (largest rank of a tree)
  - Iteratively merge root trees with same rank until all root trees have different ranks.
- **Recreate heap** by adding the new trees to the list and defining the new minimum
- Return the minimum node.
- **Time Complexity:** $O(log(n))$

Extract-Min Example I

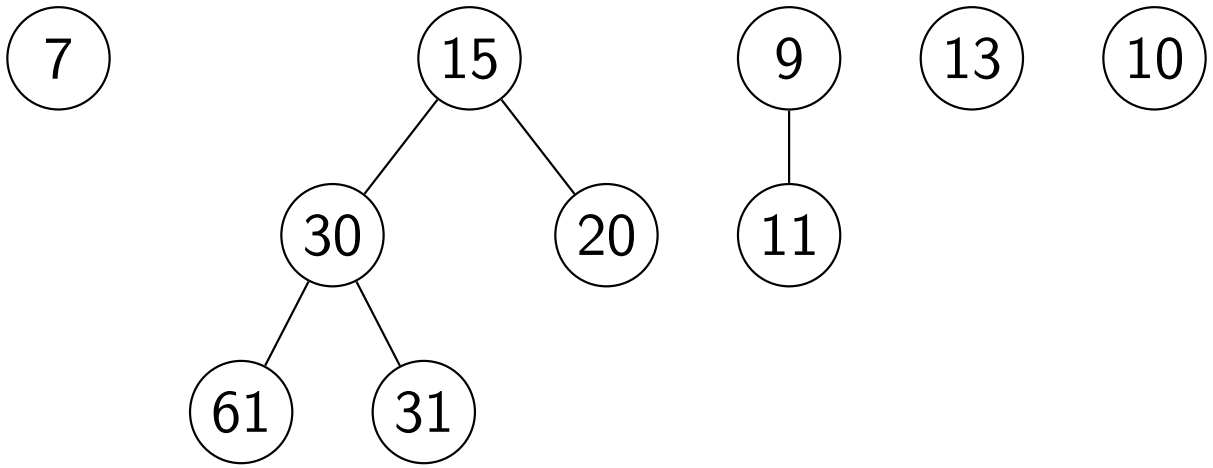
- We start with the following F-Heap:



- We want to extract the minimum node.

Extract-Min Example II

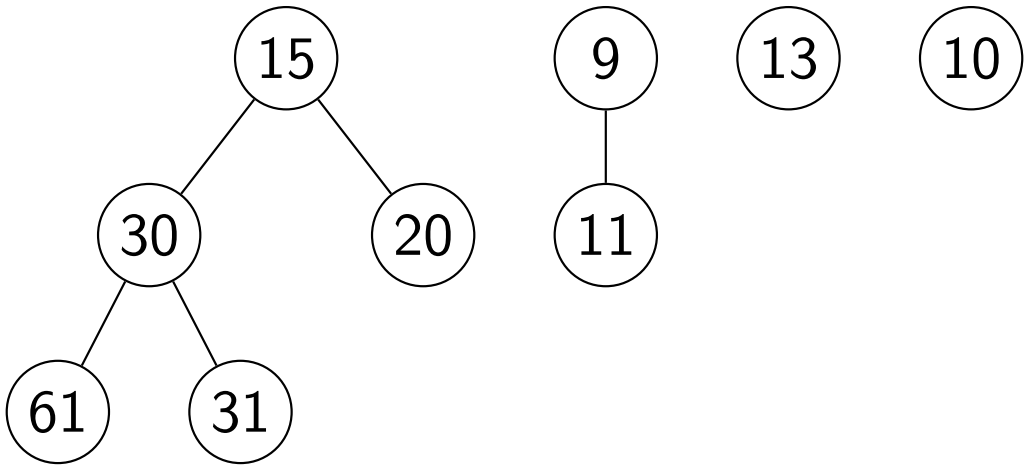
- After the minimum node's children are turned into root trees:



- Now we can easily extract the minimum node from the F-Heap.

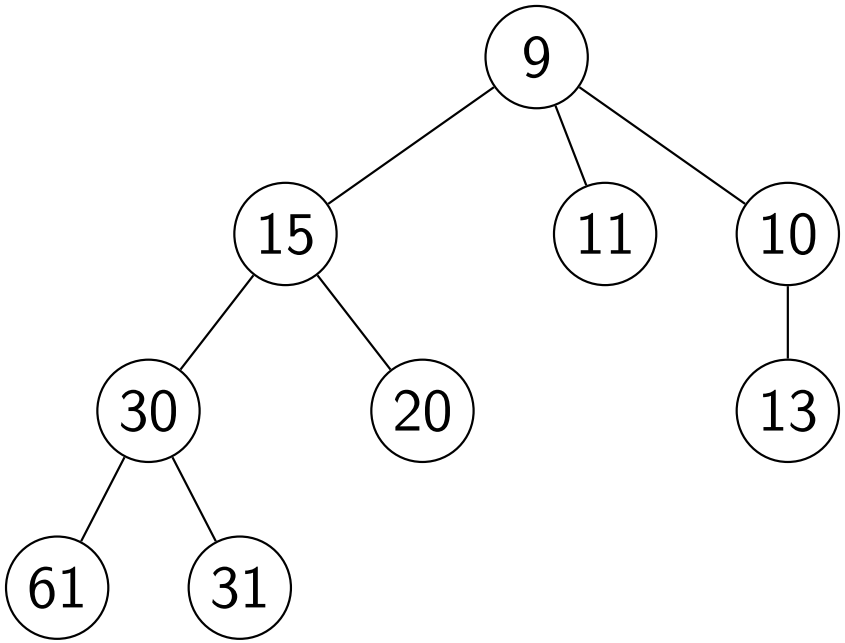
Extract-Min Example III

- After removing the minimum node from the F-Heap:



Extract-Min Example IV

- After consolidating the F-Heap:



Extract-Min Time Complexity Proof I

Lemma 1: Being **D(n)** the **maximum rank** of a tree in the fibonacci heap, Extract-Min's amortized time complexity is $O(D(n))$.

Proof

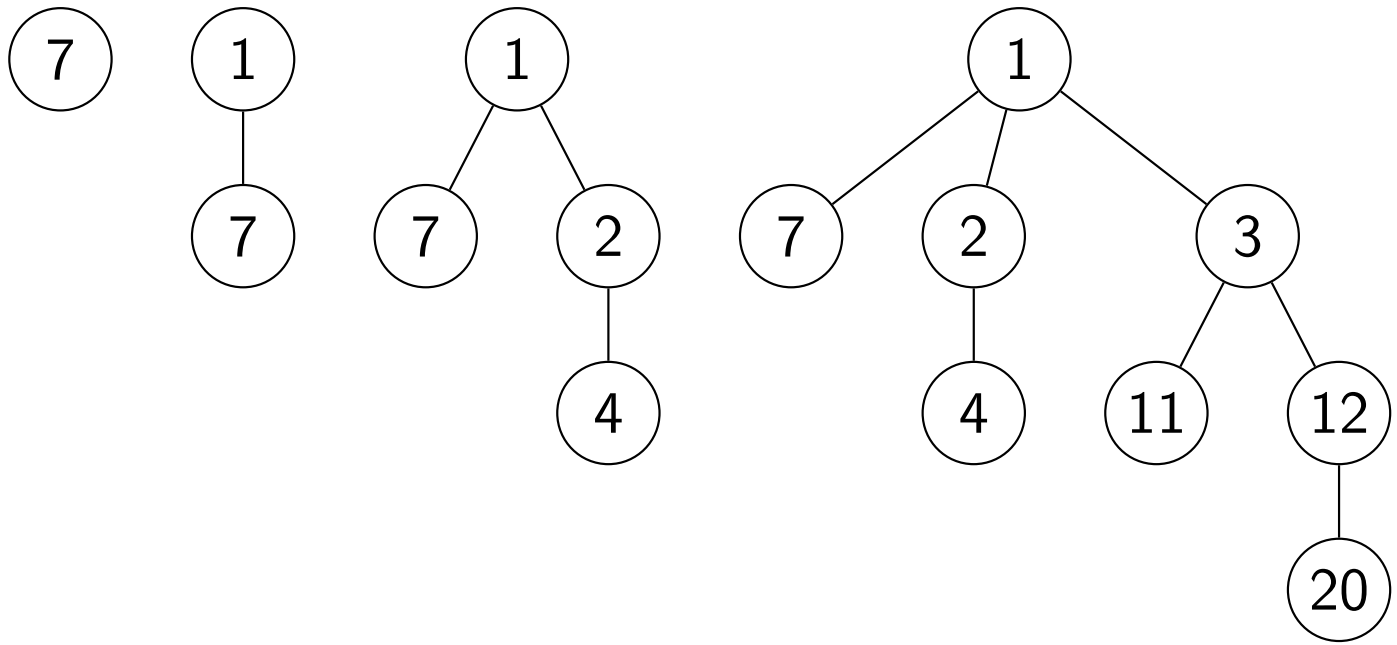
Being **t(H)** the initial **number of trees** in the heap H:

- **Removing the minimum** takes $O(D(n))$ time for the 'upgrade' of D(n) children to root
- **Consolidate takes** $O(t(H) + D(n))$ time for the initialization of the array of ranks $(D(n))$ and the worse case merging scenario $t(H)$ trees
- **Recreating the heap** takes $O(D(n))$ for traversing the array of ranks (the trees) of size $D(n)$ to find the new minimum

The worse time it takes for the operation is $O(D(n) + t(H))$.
The **amortized time** it takes for the operation is $O(D(n))$ because $t(H)$ time can be attributed to the $t(H)$ previous Insert operations

Extract-Min Time Complexity Proof II

Corollary 1: All trees in a F-Heap (where no delete and extract-min operations have occurred) will be **binomial trees** as they are all generated from the merging of two binomial trees of same rank.



Corollary 2: Binomial Trees grow exponentially: $n = 2^d$, **n** being the number of nodes and **d** the rank.

Extract-Min Time Complexity Proof III

Theorem 1: Being n the number of nodes of a tree, Extract-Min's amortized time complexity is $O(log(n))$.

Proof

As per Corollaries 1 and 2, a fibonacci heap's trees' size grows exponentially ($n = 2^d$).
As such, $n = 2^{D(n)}$ which implies $log(n) = D(n)$; which means $O(D(n)) = O(log(n))$.

Table of Contents

Decrease-Key

- Idea: Cut node that has been decreased and add it to the root list.

Decrease-Key(H,x,k)

- Change the key of x to k.
- If the heap rule is violated, cut the edge linking x to its parent p(x), by removing x from the list of children of p(x) and making the parent pointer of x null.
- The subtree rooted at x into a new tree of H.
- Requires decreasing the rank of p(x) and adding x to the list of roots of H.
- **Time Complexity:** $\Theta(1)$

Delete

- Idea: Use previously defined functions to implement Delete.

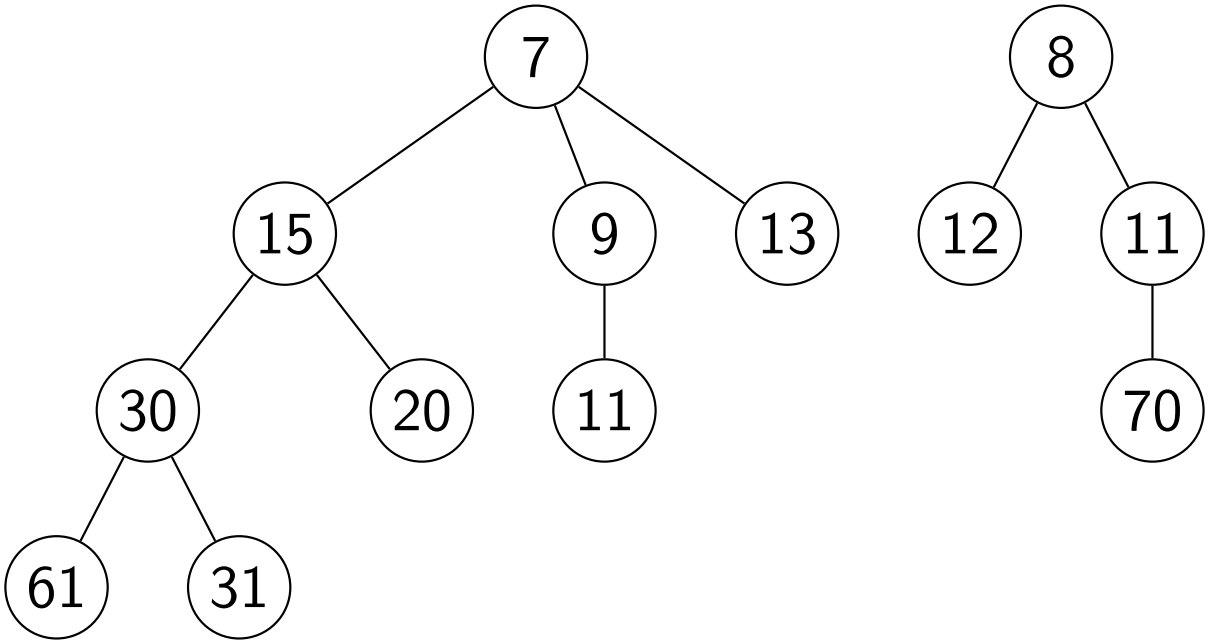
Delete(H,x)

- Decrease the key of X to $-\infty$.
- Call Extract-Min, which will remove x.
- **Time Complexity:** $O(log(n))$

- Can be also implemented like Decrease-Key, but all children of x are added to the root list [1]. Actual time of this implementation is $O(1)$.
- Amortized time is still $O(log(n))$ has we increase the number of trees in the F-heap.

Examples I

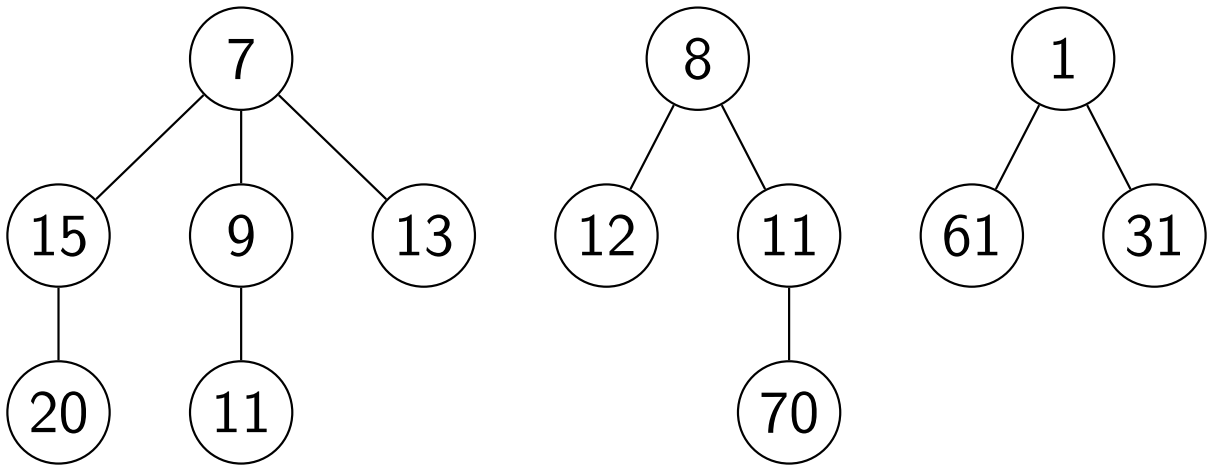
- We start with the following F-Heap.



- We want to decrease the node with key 30 to 1.

Examples II

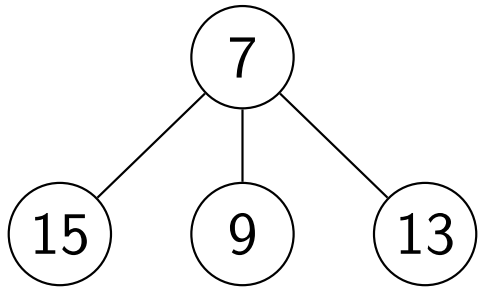
- After the node has been decreased.



- What if we keep decreasing keys and removing the nodes? What problems may arise?

Examples III

- What is wrong with the following F-Heap?



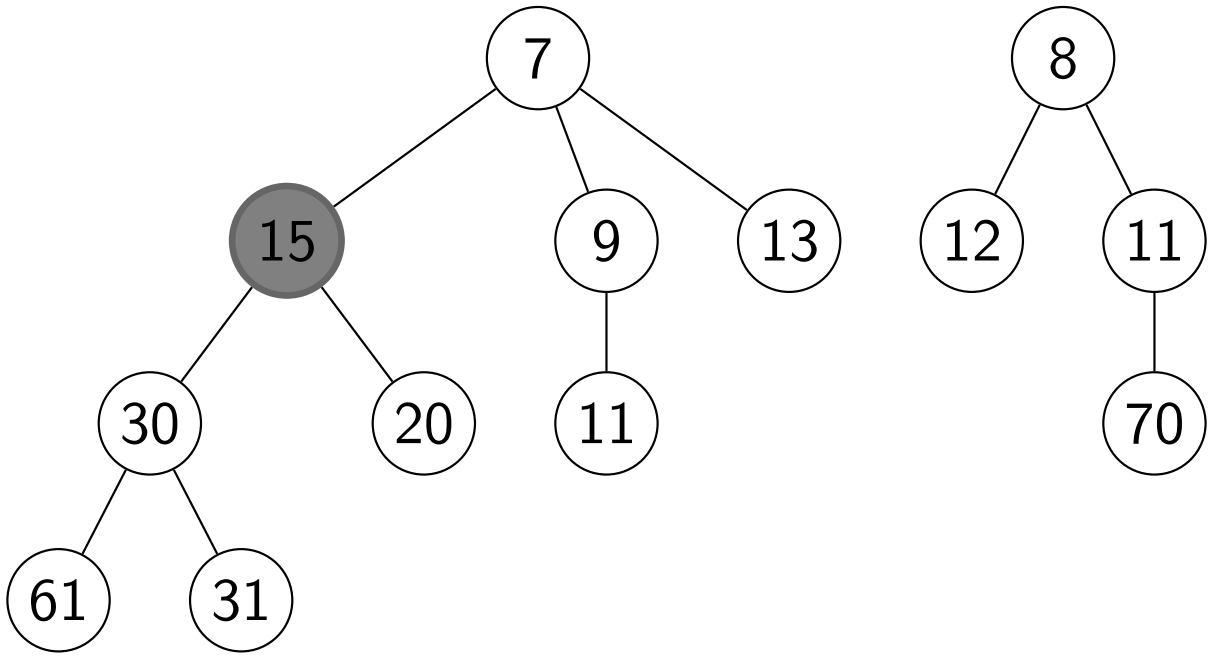
- Extract-Min is only $O(log(n))$ because we could bound $D(n)$.

Marked nodes and cuts

- Bounds proved earlier for $D(n)$ are no longer valid.
- We introduce the idea of **marked nodes**.
- A node is said to be marked if it has **lost one child**.
- After a marked node loses another child, it **becomes a root - cut**.
- Nodes can lose at most 2 children before being removed.
- Cuts can cascade through nodes, but **Decrease-Key remains** $\Theta(1)$ in amortized time (on average, for 10 Decrease-Keys we cut 20 nodes)
- Cutting increases the number of trees. However, **Extract-Min is still** $O(log(n))$ in amortized time

Examples I

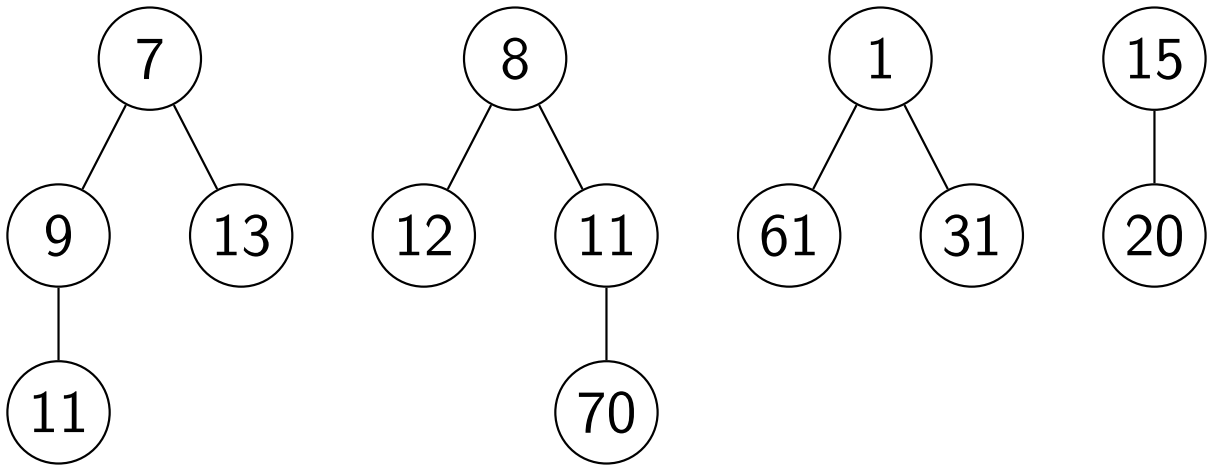
- We start with the following F-Heap, node with 15 is marked.



- We want to decrease the node with key 30 to 1, like we did before.

Examples II

- After the node has been decreased.



- In this way, $D(n)$ is decreased.
- Yet, with marked nodes and cuts in action, the trees are no longer guaranteed to be Binomial. For Extract-Min to be $O(log(n))$, the trees need to grow exponentially.

Proof of Logarithmic Time I

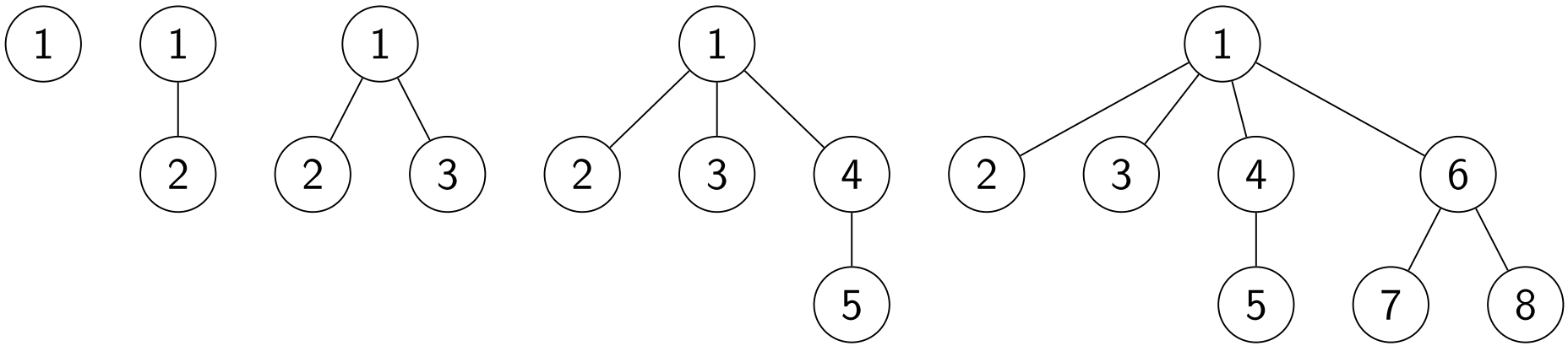
Lemma 2: Let x be any node in an F-heap. Arrange the children of x in the order they were linked to x, from earliest to latest. Then the i^{th} **child of x's rank** is greater than or equal to **i - 2**.

Proof

Let y be the ith child of x, and consider the time when y was linked to x. **Just before the linking**, **x had at least i - 1** children (some of which it may have lost after the linking). Since **x and y had the same rank** just before the linking, they both had a rank of at least i - 1 at this time. After the linking, the **rank of y could have decreased by at most one** without causing y to be cut as a child of x.

Proof of Logarithmic Time II

Taking into account the minimum rank of a node, the smallest trees ordered by rank:



- Each tree can be created by the merge of the 2 previous ones
- The number of nodes of a tree follows the Fibonacci Sequence!

Proof of Logarithmic Time III

Corollary 3: A node of rank k in an F-heap has at least F_{k+2} descendants, including itself; where F_k is the k^{th} Fibonacci number ($F_0 = 0, F_1 = 1, F_k = F_{k-2} + F_{k-1}$, for $k > 2$)

Proof

Let S_k be the minimum possible number of descendants of a node of rank k, including itself. Obviously, $S_0 = 1$, and $S_1 = 2$. Lemma 1 implies that $S_k \geq \sum_{i=0}^{k-2} S_i + 2$ for $k \geq 2$. The Fibonacci numbers satisfy $F_{k+2} = \sum_{i=2}^{k} Fi + 2$ for $k \geq 2$, from which $S_k \geq F_{k+2}$ for $k \geq 0$ follows by induction on k.

Because $F_{k+2} \geq \phi^k$ where $\phi = (1 + \sqrt{5})/2$ - the golden ratio. We can bound the number of descendants as $n \geq \phi^{D(n)}$ and thus $D(n) \leq log(n)$. Bounding the time for Extract-Min.

Table of Contents

Visualization

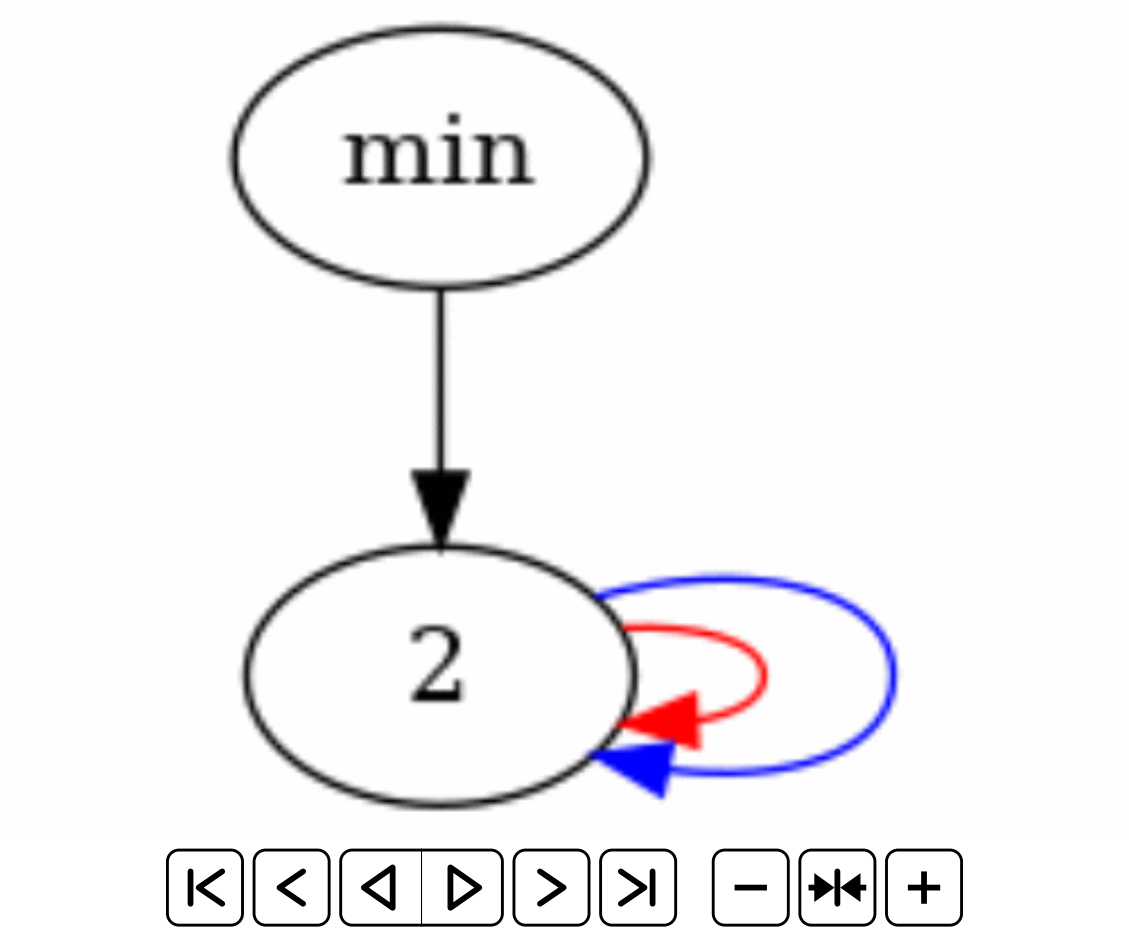


Figure 3: Fibonacci Heap implementation visualized.[2]

[2] /fibonacci-heap GitHub repository with source code

Table of Contents

Pitfalls

Amortized time the time complexity analysis in most of the operations on the F-Heap are obtained through *Amortized Analysis*, rather than denoting the worst case scenario

High constant times due to the complexity of the operations, although some of them have constant time complexity, they are already very time consuming operations

Poor locality the data structure has very poor locality i.e. information is very spread out in memory, making use of a great quantity of pointers, making its implementation very cache inefficient

Complex implementation compared to the Binary Heap

Due to these problems, F-Heaps' worth lied more in the statement they attempted to prove rather than its applications in practice.

Evolution

Following Fibonacci Heaps came other attempts at attaining worst case constant time Insert and Decrease-Key and logarithmic time Extract-Min

- **Brodal Queues** appeared in a 1996 article by Gerth Stølting Brodal [3]
- **Strict Fibonacci Heaps** came up later (2012) in a paper authored by many of the previous authors of the other two data structures [4]

Both implementations **managed to attain** the temporal complexities of the first Fibonacci Heap implementation in **worse case analysis**. However, they were **even more complex** to implement and did not improve on the other flaws of the original data structure, making them even more **unfit** for usage in the **real world**.

Applications

Despite the imperfections, the data structure still has some applications:

- Algorithms where the Decrease-Key operation is often executed
- Projects/applications that deal with very large amounts of data

In practice:

- At the time of its invention it was used to accelerate implementations of the **Dijkstra's Algorithm** [1]
- One example of a **recent application** (2017) of Fibonacci Heaps is its usage in a new algorithm used idealized for the calculation of *Augmented Merge Trees* [5], in the field of Topological Analysis.

References I


M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, 1987, ISSN: 0004-5411. DOI: 10.1145/28869.28874. [Online]. Available: https://doi.org/10.1145/28869.28874.


T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd. The MIT Press, 2009, ISBN: 0262033844.


G. S. Brodal, "Worst-case efficient priority queues", 1996. [Online]. Available: https://dl.acm.org/doi/10.5555/313852.313883.


G. L. Brodal Gerth Stølting and R. E. Tarjan, "Strict fibonacci heaps", 2012. DOI: 10.1145/2213977.2214082. [Online]. Available: https://doi.org/10.1145/2213977.2214082.

References II

 e. a. Gueunet Charles, "Task-based augmented merge trees with fibonacci heaps,", 2017. DOI: 10.1109/LDAV.2017.8231846. [Online]. Available: https://doi.org/10.1109/LDAV.2017.8231846.