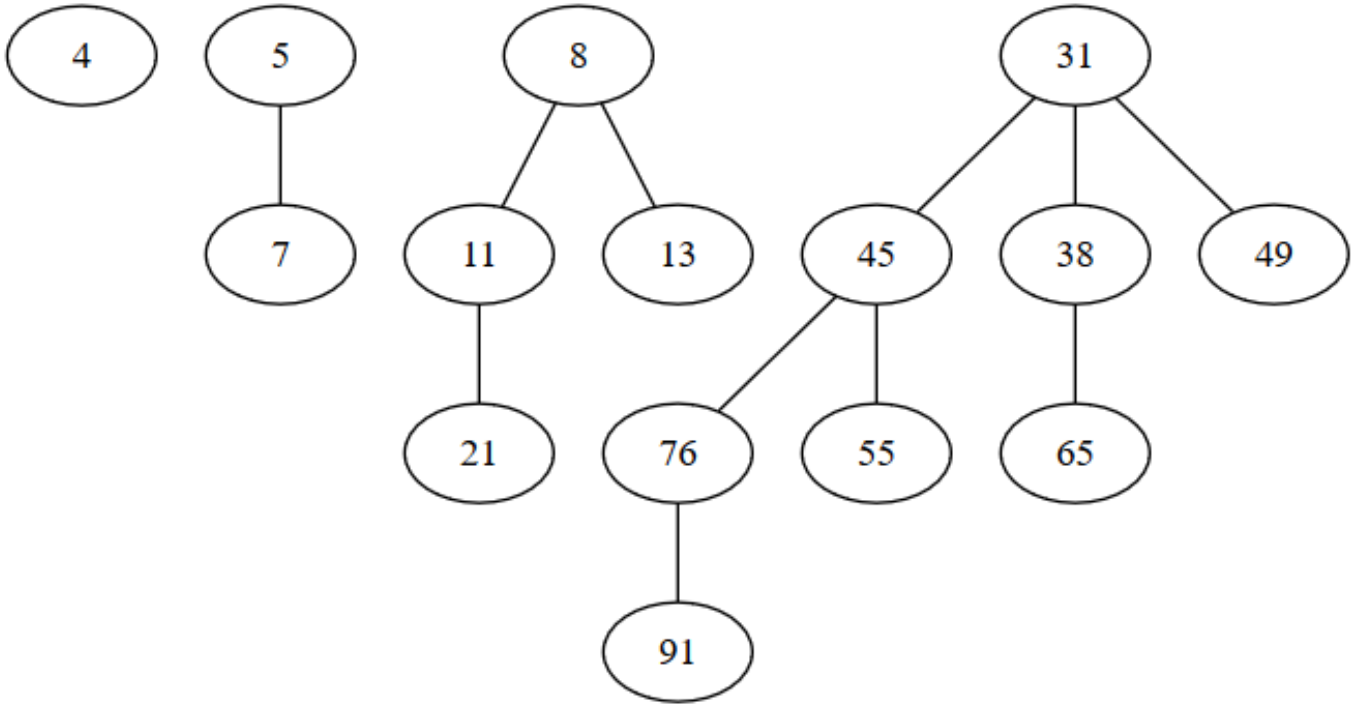
Binomial heap

- A *binomial heap* is a collection of *binomial trees*.
- A *binomial tree* can be defined in a recursive manner:
  - A *binomial tree of order 0* is a single node.
  - A *binomial tree of order k* is a tree which has a root and k children, each being the root of a binomial tree of order $k-1$, $k-2$, ..., 2, 1, 0 (in this order).

Binomial tree - Example

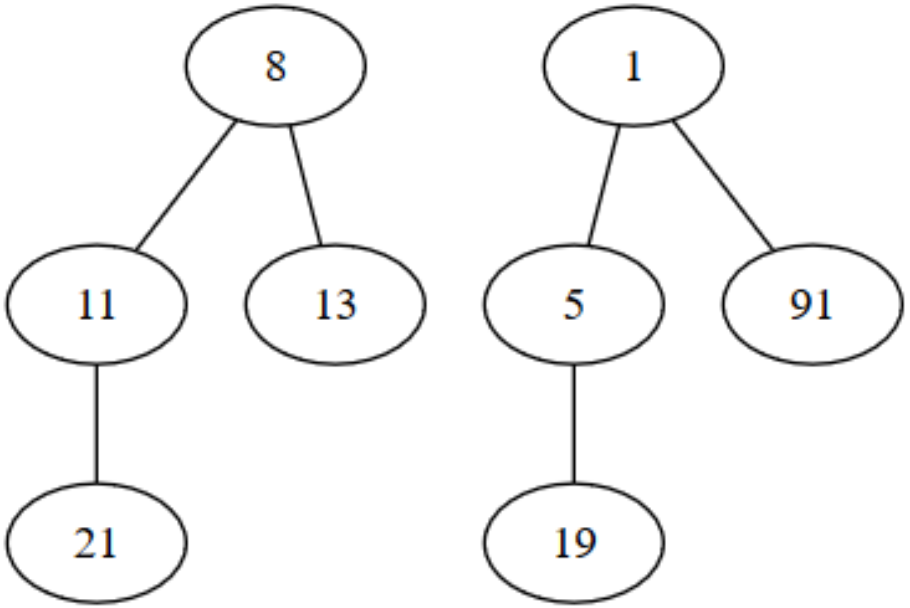


Binomial trees of order 0, 1, 2 and 3

Binomial tree

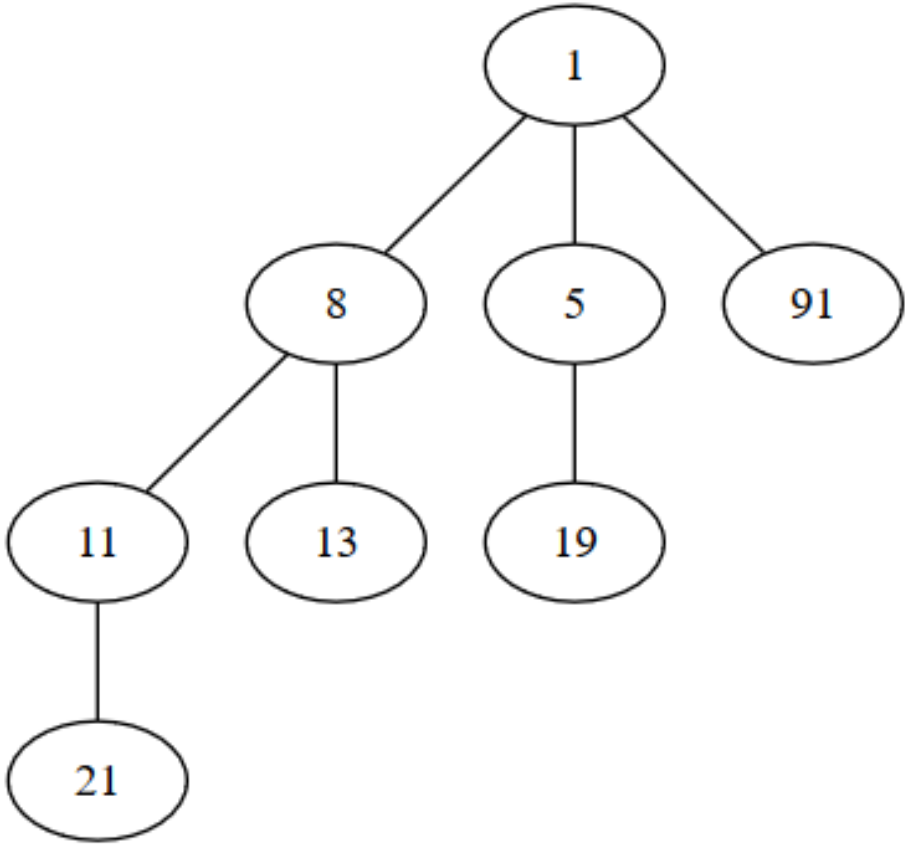
- A binomial tree of order k has exactly 2^k nodes.
- The height of a binomial tree of order k is k.
- If we delete the root of a binomial tree of order k, we will get k binomial trees, of orders $k-1$, $k-2$, ..., 2, 1, 0.
- Two binomial trees of the same order k can be merged into a binomial tree of order $k+1$ by setting one of them to be the leftmost child of the other.

Binomial tree - Merge I



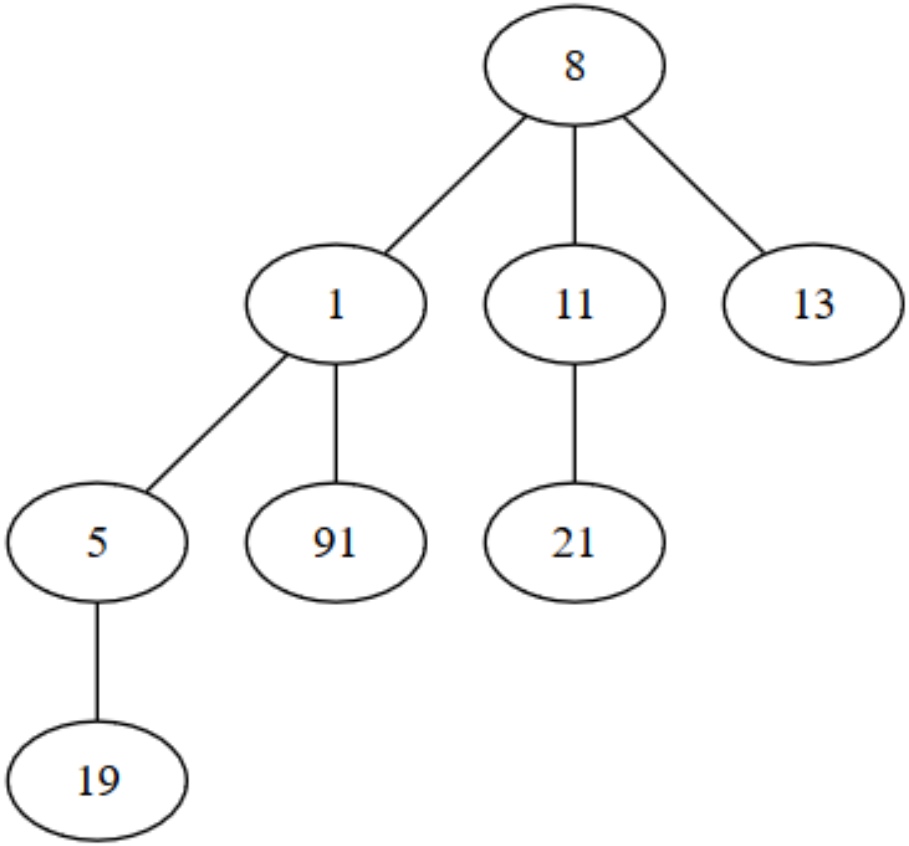
Before merge we have two binomial trees of order 2

Binomial tree - Merge II



One way of merging the two binomial trees into one of order 3

Binomial tree - Merge III



Another way of merging the two binomial trees into one of order 3

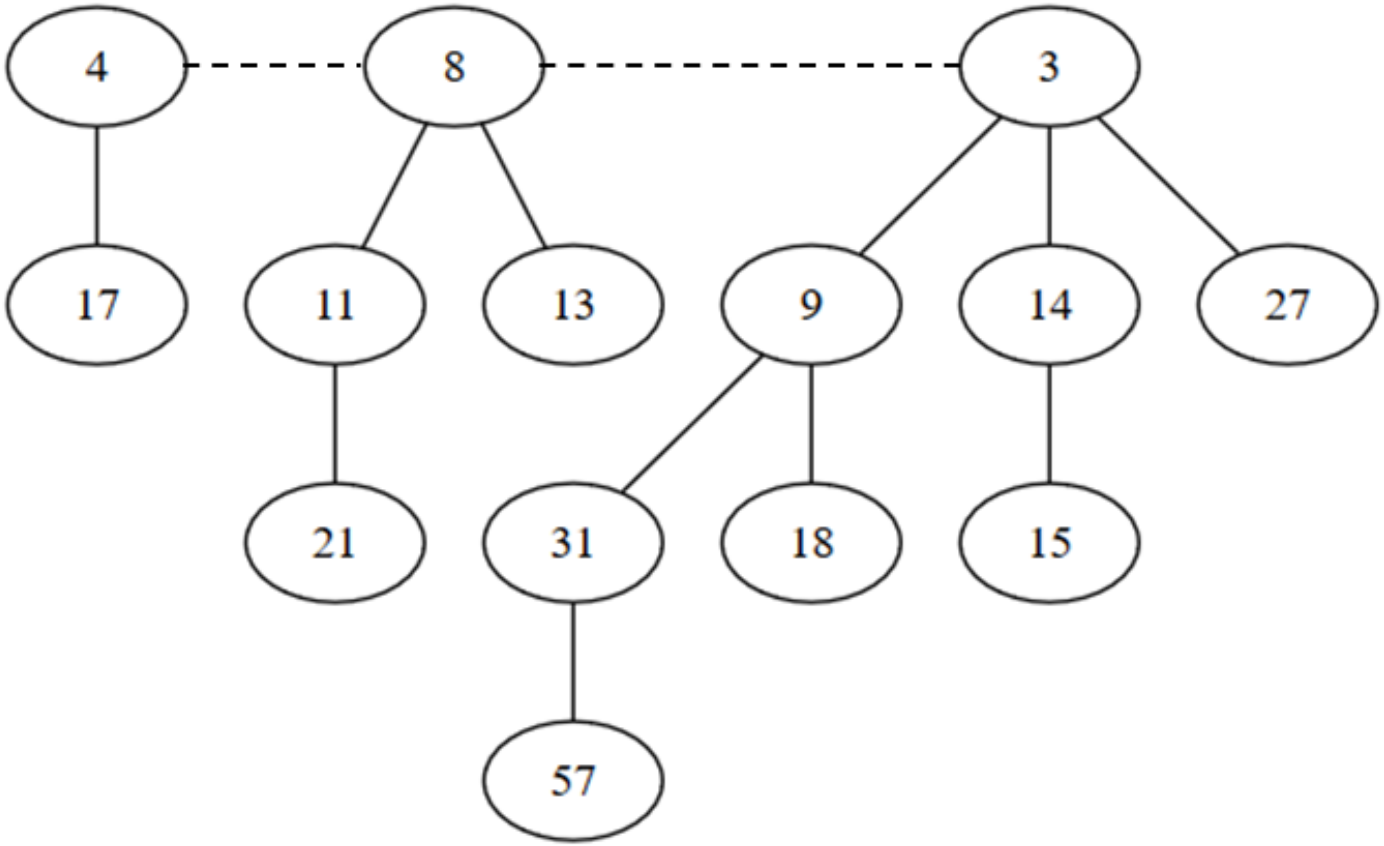
Binomial tree - representation

- If we want to implement a binomial tree, we can use the following representation:
  - We need a structure for nodes, and for each node we keep the following:
    - The information from the node
    - The address of the parent node
    - The address of the first child node
    - The address of the next sibling node
  - For the tree we will keep the address of the root node (and probably the order of the tree)

Binomial heap

- A binomial heap is made of a collection/sequence of binomial trees with the following property:
  - Each binomial tree respects the heap-property: for every node, the value from the node is less than the value of its children (assume MIN_HEAPS).
  - There can be at most one binomial tree of a given order k.
  - As representation, a binomial heap is usually a sorted linked list, where each node contains a binomial tree, and the list is sorted by the order of the trees.

Binomial tree - Example



Binomial heap with 14 nodes, made of 3 binomial trees of orders 1, 2 and 3

Binomial tree

- For a given number of elements, n, the structure of a binomial heap (i.e. the number of binomial trees and their orders) is unique.
- The structure of the binomial heap is determined by the binary representation of the number n.
- For example $14 = 1110$ (in binary) $= 2^3 + 2^2 + 2^1$, so a binomial heap with 14 nodes contains binomial trees of orders 3, 2, 1 (but they are stored in the reverse order: 1, 2, 3).
- For example $21 = 10101 = 2^4 + 2^2 + 2^0$, so a binomial heap with 21 nodes contains binomial trees of orders 4, 2, 0.

Binomial heap

- A binomial heap with n elements contains at most $log_2 n$ binomial trees.
- The height of the binomial heap is at most $log_2 n$.

Binomial heap - merge

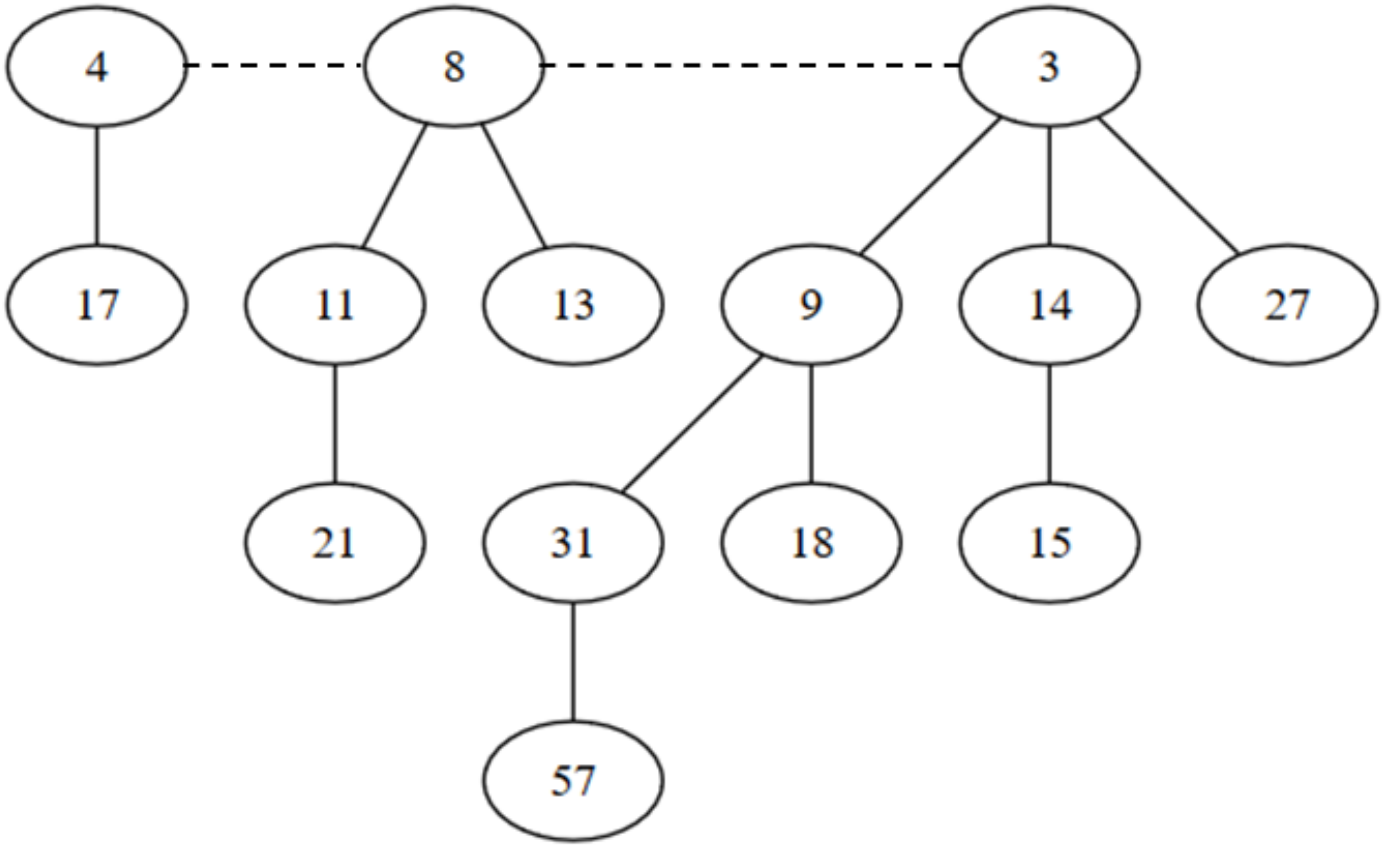
- The most interesting operation for two binomial heaps is the merge operation, which is used by other operations as well. After the merge operation the two previous binomial heaps will no longer exist, we will only have the result.

- Since both binomial heaps are sorted linked lists, the first step is to *merge* the two linked lists (standard merge algorithm for two sorted linked lists).

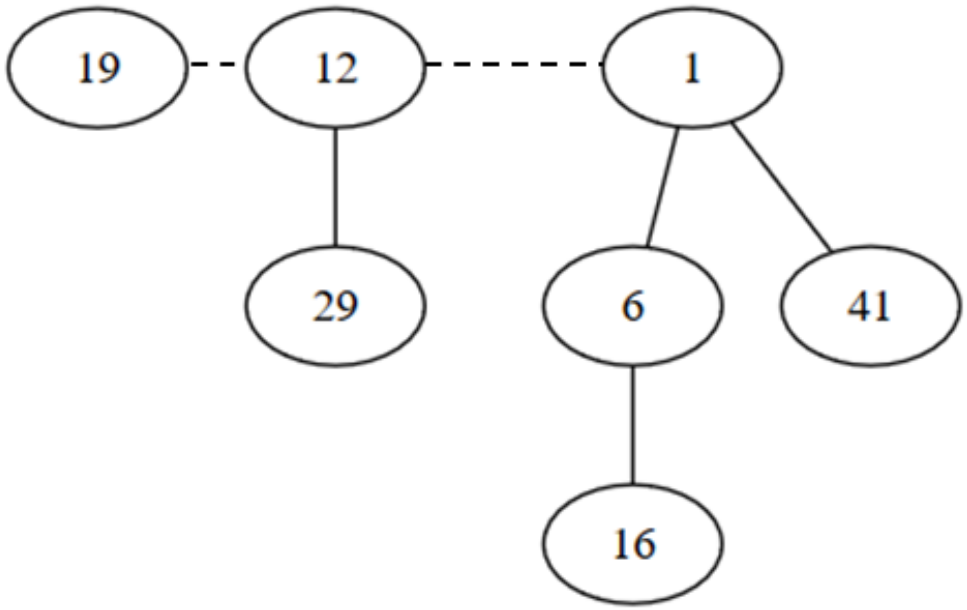
- The result of the merge can contain two binomial trees of the same order, so we have to iterate over the resulting list and transform binomial trees of the same order k into a binomial tree of order $k+1$. When we merge the two binomial trees we must keep the heap property.

Binomial heap - merge - example I

- Let's merge the following two binomial heaps:

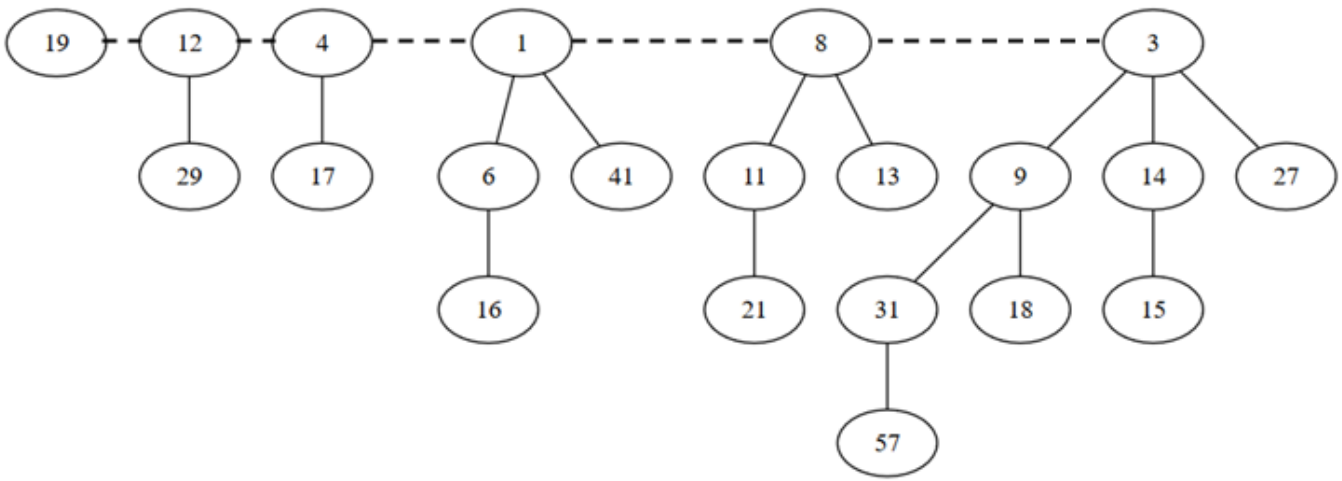


Binomial heap - merge - example II



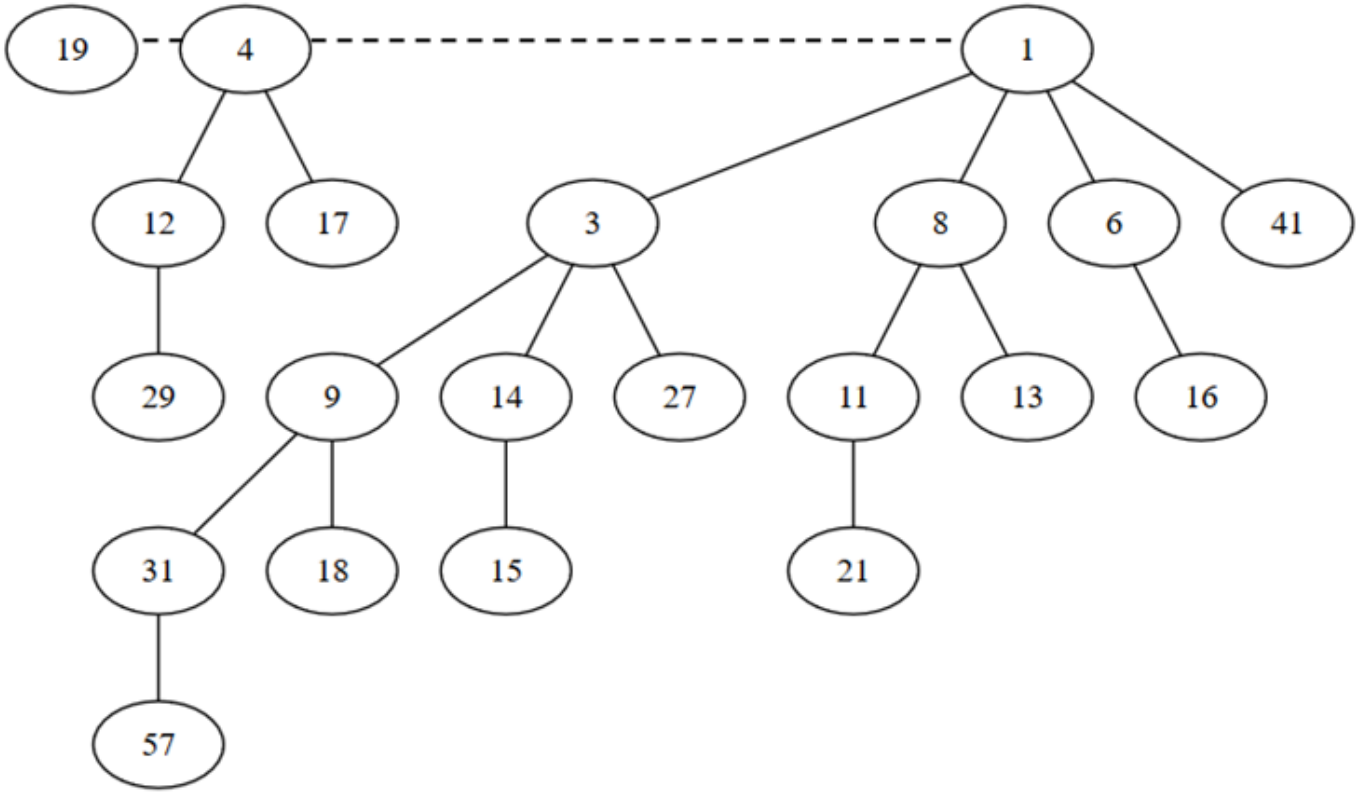
Binomial heap - merge - example III

- After merging the two linked lists of binomial trees:



Binomial heap - merge - example IV

- After transforming the trees of the same order (final result of the merge operation).



Binomial heap - Merge operation

- If both binomial heaps have n elements, merging them will have $O(log_2 n)$ complexity (the maximum number of binomial trees for a binomial heap with n elements is $log_2 n$).

Binomial heap - other operations I

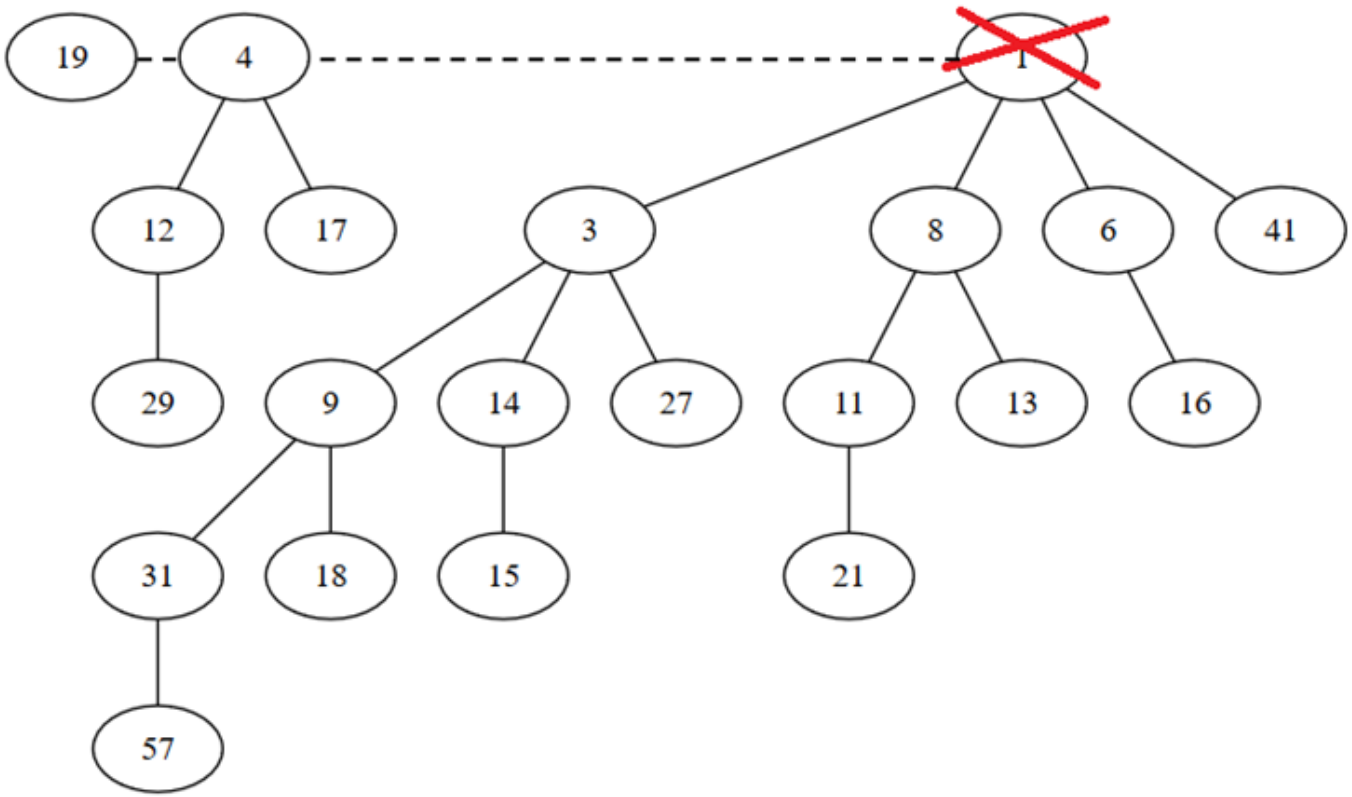
- Most of the other operations that we have for the binomial heap (because we need them for the priority queue) will use the merge operation presented above.
- *Push operation:* Inserting a new element means creating a binomial heap with just that element and merging it with the existing one. Complexity of insert is $O(log_2 n)$ in worst case ($\Theta(1)$ amortized).
- *Top operation:* The minimum element of a binomial heap (the element with the highest priority) is the root of one of the binomial trees. Returning the minimum means checking every root, so it has complexity $O(log_2 n)$.

Binomial heap - other operations II

- *Pop operation:* Removing the minimum element means removing the root of one of the binomial trees. If we delete the root of a binomial tree, we will get a sequence of binomial trees. These trees are transformed into a binomial heap (just reverse their order), and a merge is performed between this new binomial heap and the one formed by the remaining elements of the original binomial heap.

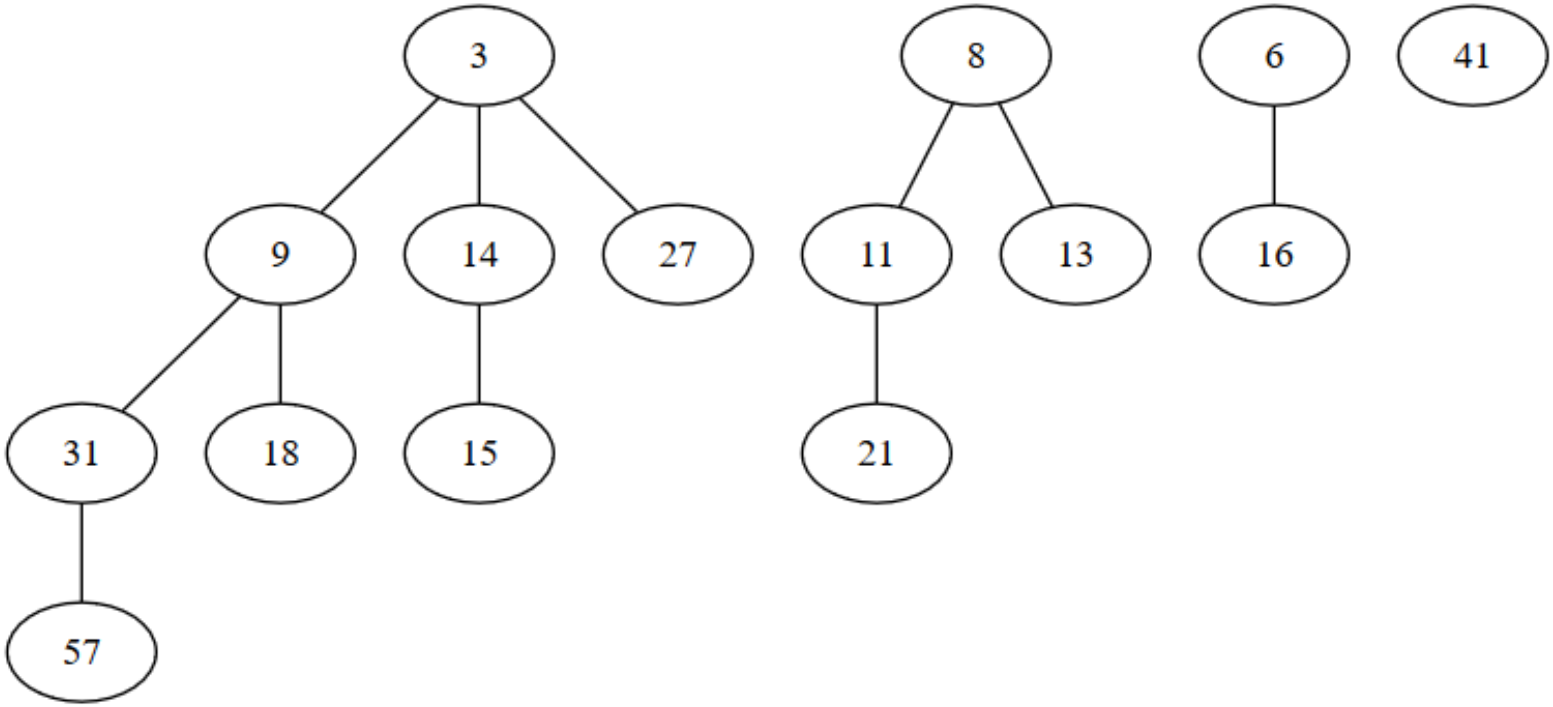
Binomial heap - other operations III

- The minimum is one of the roots.



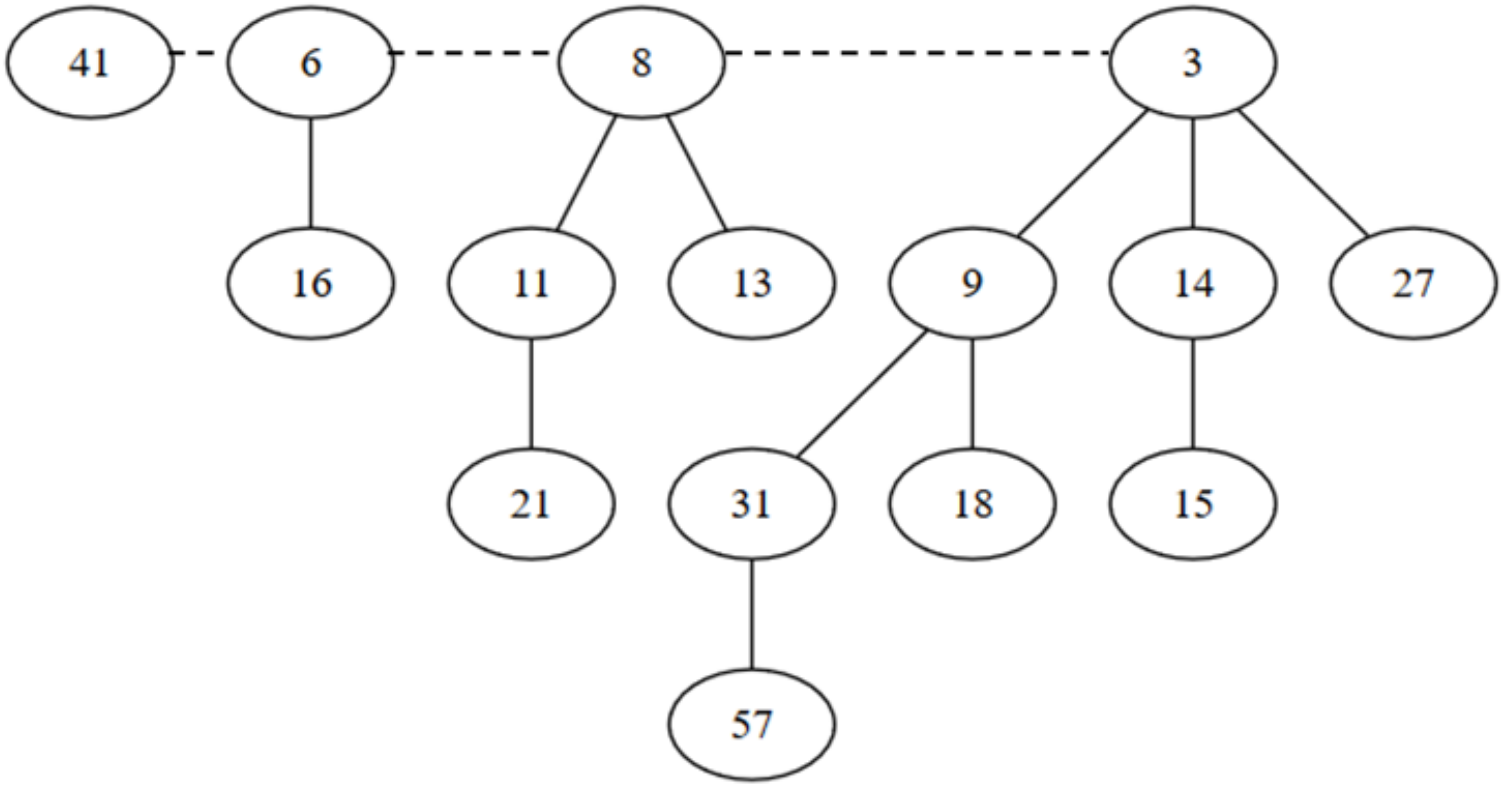
Binomial heap - other operations IV

- Break the corresponding tree into k binomial trees



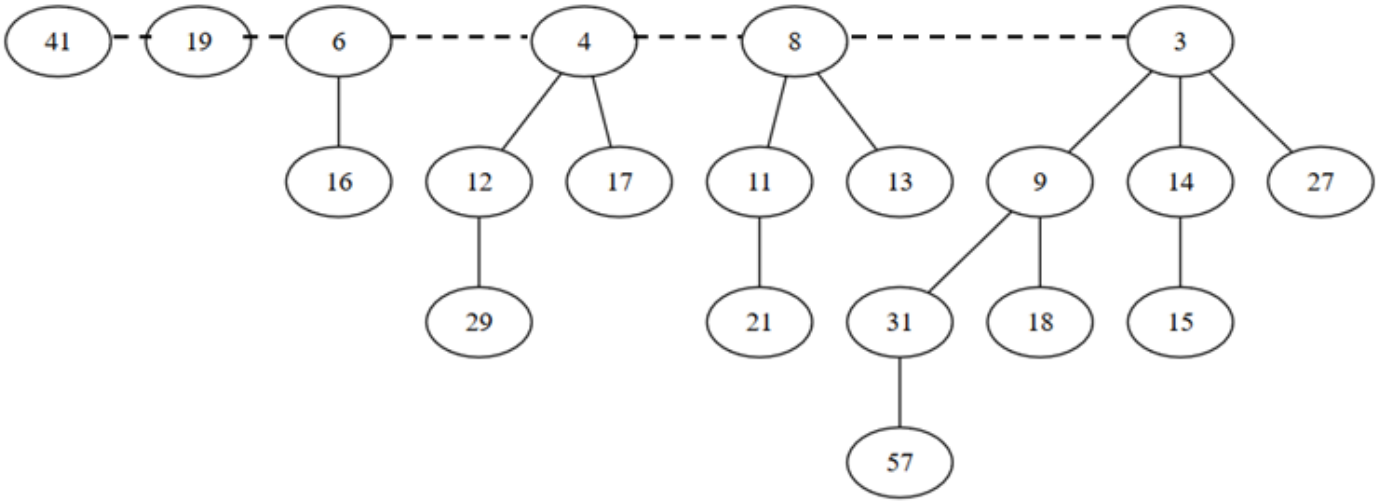
Binomial heap - other operations V

- Create a binomial heap of these trees



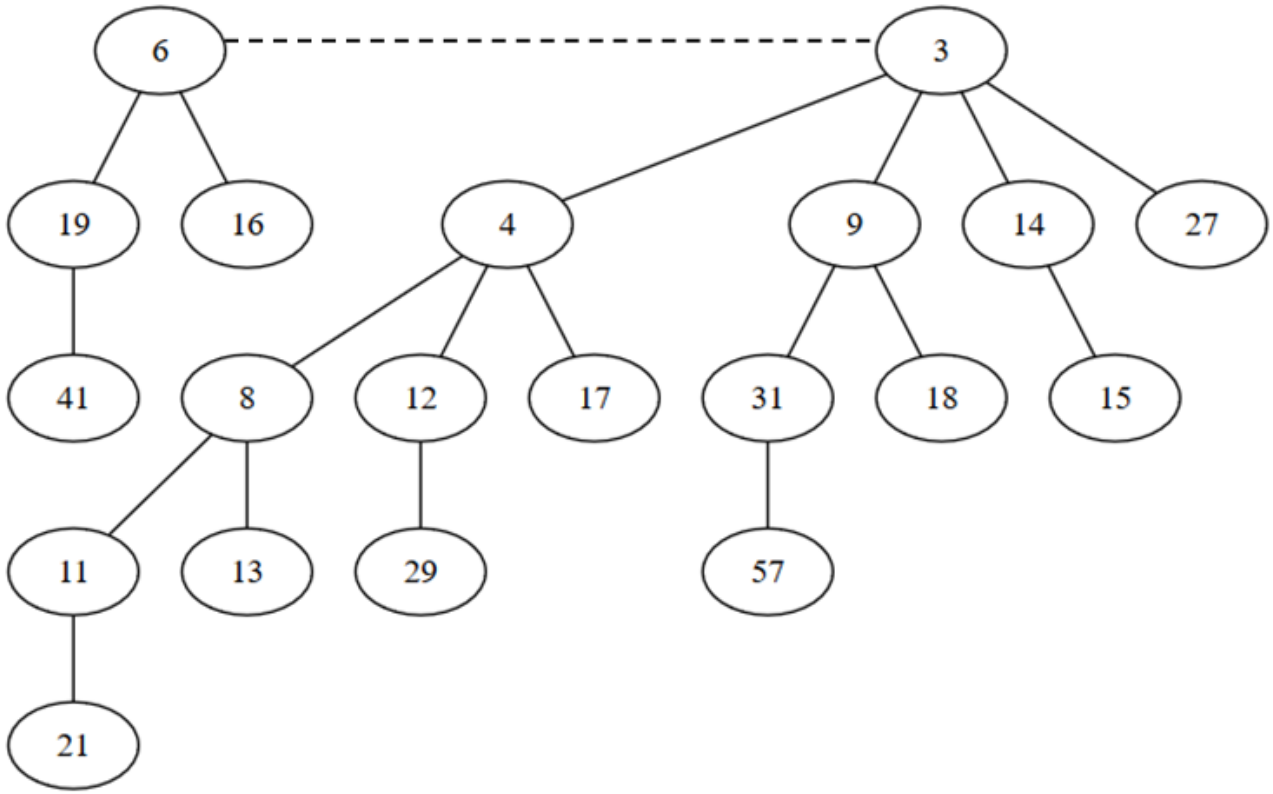
Binomial heap - other operations VI

- Merge it with the existing one (after the merge algorithm)



Binomial heap - other operations VII

- After the transformation



- The complexity of the remove-minimum operation is $O(log_2 n)$

Binomial heap - other operations VIII

- Assuming that we have a pointer to the element whose priority has to be increased (in our figures lower number means higher priority), we can just change the priority and bubble-up the node if its priority is greater than the priority of its parent. Complexity of the operation is: $O(log_2 n)$
- Assuming that we have a pointer to the element that we want to delete, we can first decrease its priority to $-\infty$ (this will move it to the root of the corresponding binomial tree) and remove it. Complexity of the operation is: $O(log_2 n)$