DATA STRUCTURES AND ALGORITHMS

LECTURE 8

Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017

In Lecture 7...

- ADT Stack
- ADT Queue

Today

Written test - General Info

- Will be at the first half of seminar 5
- Will last 50 minutes
- **Everybody has to participate at the test with his/her own group!**

Written test - Subjects I

- Each subject will be of the form: Container + Representation (a data structure) + Operation
- Containers:
  - Bag
  - Set
  - Map
  - MultiMap
  - List
  - sorted version of the above containers
  - Sparse Matrix

Written test - Subjects II

- Data structures
  - Singly Linked List with Dynamic Allocation
  - Doubly Linked List with Dynamic Allocation
  - Singly Linked List on an Array
  - Doubly Linked List on an Array

Written test - Subjects III

- Operation
  - Add
  - Remove
  - Search
  - For Sparse Matrix: Modify value (from 0 to non-zero or from non-zero to 0), search for element from position (i,j)

- Example of a subject: **ADT Set - represented on a doubly linked list on an array - operation: add**.

Written test - Grading

- **1p - Start**
- **0.5p - Specification of the operation** - header, preconditions, postconditions
  - add(s, e)
  - **pre:** $s \in \mathcal{S}$, $e \in TElem$
  - **post:** $s' \in S$, $s' = s \cup e$ (if e is in s, it will not be added)
- **0.5p - Short description of the container**
  - A Set is a container in which no duplicates are allowed and the order of the elements is not important (there are no positions in a Set).

Written test - Grading II

- **1.25p - representation**
  - 1p - structure(s) needed for the container
  - 0.25p - structure for the iterator for the container

Set:
cap: Integer
elems: TElem[]
next: Integer[]
prev: Integer[]
head: Integer
tail: Integer
firstEmpty: Integer

Iterator:
set: Set
current: Integer

Written test - Grading III

- **4.5p - Implementation of the operation in pseudocode**
  - If you have a data structure with dynamic allocation, you can use the *allocate* and *deallocate/free* operations. If you call any other function(s) in the implementation, you have to implement them.
  - If you have a data structure on an array, you do not need to write code for *resize*-ing the data structure, but you need to show where the resize part would be:

```
...
if s.firstEmpty = -1 then
    @resize
end-if
```

Written test - Grading IV

- **1.25p - Complexity**
  - 0.25p - Best Case - with explanations
  - 0.5p - Worst Case - with computations
  - 0.5p - Average Case - with computations
- **1p - Style**
  - Is it general (uses TElem, TComp, a generic Relation)?
  - Is it efficient?
  - etc.

ADT Deque

- The ADT Deque (Double Ended Queue) is a container in which we can insert and delete from both ends:
  - We have *push_front* and *push_back*
  - We have *pop_front* and *pop_back*
  - We have *top_front* and *top_back*
- We can simulate both stacks and queues with a deque if we restrict ourselves to using only part of the operations.

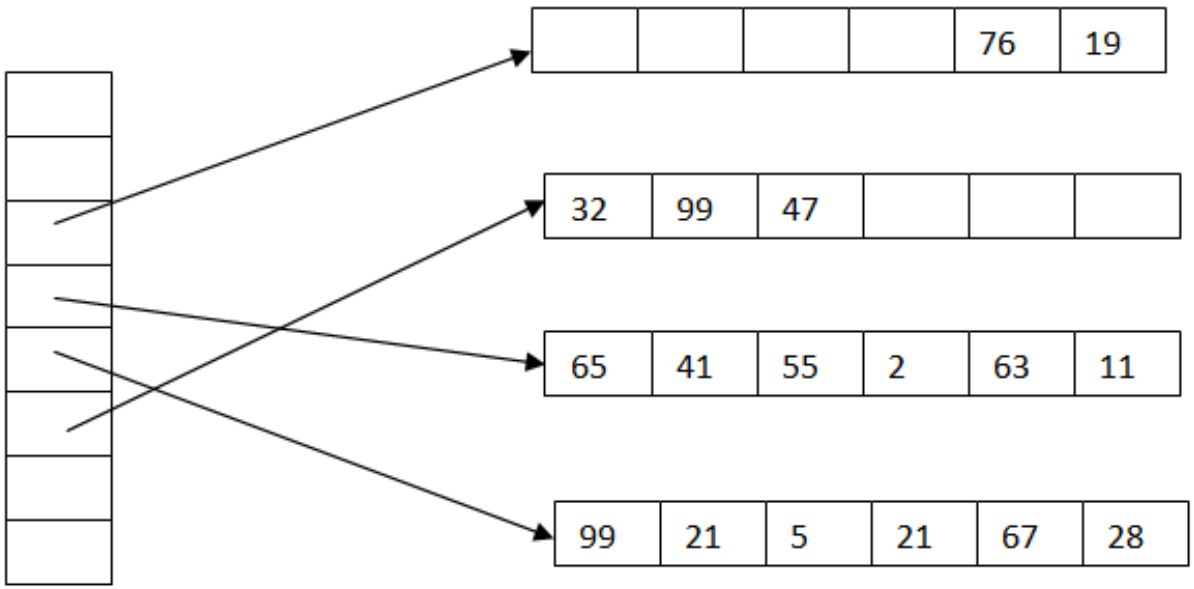
ADT Deque

- Possible representations for a Deque:
  - Circular Array
  - Doubly Linked List
  - A dynamic array of constant size arrays

ADT Deque - Representation

- An interesting representation for a deque is to use a dynamic array of fixed size arrays:
  - Place the elements in fixed size arrays (blocks).
  - Keep a dynamic array with the addresses of these blocks.
  - Every block is full, except for the first and last ones.
  - The first block is filled from right to left.
  - The last block is filled from left to right.
  - If the first or last block is full, a new one is created and its address is put in the dynamic array.
  - If the dynamic array is full, a larger one is allocated, and the addresses of the blocks are copied (but elements are not moved).

Deque - Example



- Elements of the deque: 76, 19, 65, ..., 11, 99, ..., 28, 32, 99, 47

Deque - Example

- Information (fields) we need to keep to represent a deque using a dynamic array of blocks:
  - Block size
  - The dynamic array with the addresses of the blocks
  - Capacity of the dynamic array
  - First occupied position in the dynamic array
  - First occupied position in the first block
  - Last occupied position in the dynamic array
  - Last occupied position in the last block
  - The last two fields are not mandatory if we keep count of the total number of elements in the deque.

ADT Priority Queue

- The ADT Priority Queue is a container in which each element has an associated *priority* (of type *TPriority*).
- In a Priority Queue access to the elements is restricted: we can access only the element with the highest priority.
- Because of this restricted access, we say that the Priority Queue works based on a **HPF - Highest Priority First** policy.

ADT Priority Queue

- In order to work in a more general manner, we can define a relation $\mathcal{R}$ on the set of priorities: $\mathcal{R} : \mathit{TPriority} \times \mathit{TPriority}$
- When we say *the element with the highest priority* we will mean that the highest priority is determined using this relation $\mathcal{R}$.
- If the relation $\mathcal{R} = "\geq"$, the element with the *highest priority* is the one for which the value of the priority is the largest (maximum).
- Similarly, if the relation $\mathcal{R} = "\leq"$, the element with the *highest priority* is the one for which the value of the priority is the lowest (minimum).

Priority Queue - Interface I

- The domain of the ADT Priority Queue:
  $\mathcal{PQ} = \{pq | pq$ is a priority queue with elements $(e, p), e \in TElem, p \in TPriority\}$
- The interface of the ADT Priority Queue contains the following operations:

Priority Queue - Interface II

- init (pq, R)
  - **Description:** creates a new empty priority queue
  - **Pre:** R is a relation over the priorities, $R : TPriority \times TPriority$
  - **Post:** $pq \in \mathcal{PQ}$, pq is an empty priority queue

Priority Queue - Interface III

- destroy(pq)
  - **Description:** destroys a priority queue
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:** pq was destroyed

Priority Queue - Interface IV

- push(pq, e, p)
  - **Description:** pushes (adds) a new element to the priority queue
  - **Pre:** $pq \in \mathcal{PQ}, e \in TElem, p \in TPriority$
  - **Post:** $pq' \in \mathcal{PQ}, pq' = pq \oplus (e, p)$

Priority Queue - Interface V

- pop (pq, e, p)
  - **Description:** pops (removes) from the priority queue the element with the highest priority. It returns both the element and its priority
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:** $e \in TElem, p \in TPriority$, e is the element with the highest priority from pq, p is its priority.
    $pq' \in \mathcal{PQ}, pq' = pq \ominus (e, p)$
  - **Throws:** an exception if the priority queue is empty.

Priority Queue - Interface VI

- top (pq, e, p)
  - **Description:** returns from the priority queue the element with the highest priority and its priority. It does not modify the priority queue.
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:** $e \in TElem, p \in TPriority$, e is the element with the highest priority from pq, p is its priority.
  - **Throws:** an exception if the priority queue is empty.

Priority Queue - Interface VII

- isEmpty(pq)
  - **Description:** checks if the priority queue is empty (it has no elements)
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:**

$$isEmpty \leftarrow \begin{cases} true, \ if \ pq \ has \ no \ elements \\ false, \ otherwise \end{cases}$$

Priority Queue - Interface VIII

- isFull (pq)
  - **Description:** checks if the priority queue is full (not every implementation has this operation)
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:**

$$isFull \leftarrow \begin{cases} true, \ if \ pq \ is \ full \\ false, \ otherwise \end{cases}$$

Priority Queue - Interface IX

- **Note:** priority queues cannot be iterated, so they don't have an *iterator* operation!

Priority Queue - Representation

- What data structures can be used to implement a priority queue?
  - Dynamic Array
  - Linked List
  - Heap

Priority Queue - Representation

- If the representation is a Dynamic Array or a Linked List we have to decide how we store the elements in the array/list:
  - we can keep the elements ordered by their priorities
  - we can keep the elements in the order in which they were inserted

Priority Queue - Representation

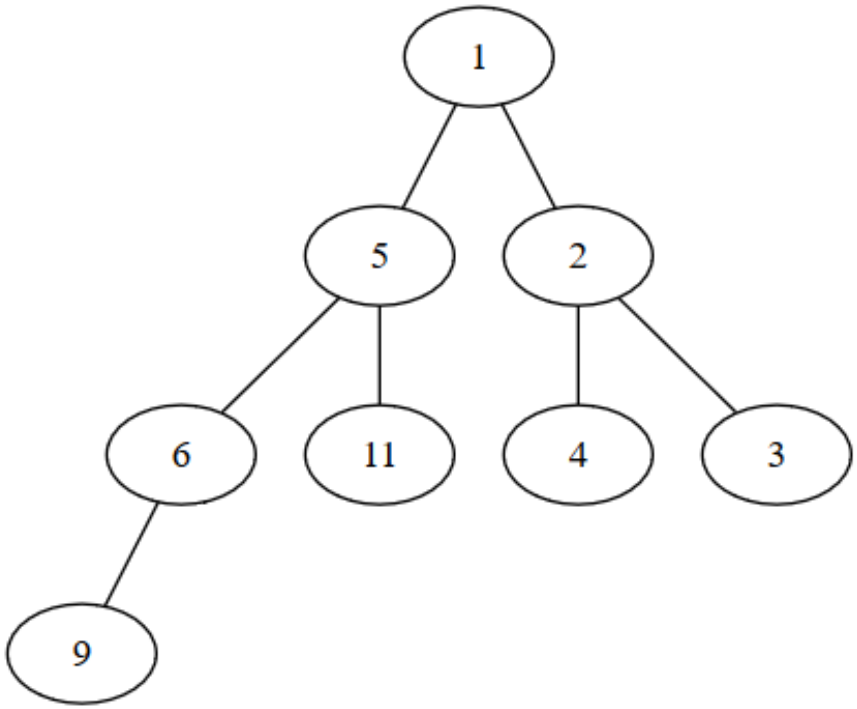
- Complexity of the main operations for the two representation options:

Operation	**Sorted**	**Non-sorted**
push	$O(n)$	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(n)$
top	$\Theta(1)$	$\Theta(n)$

- What happens if we keep in a separate field the element with the highest priority?

Priority Queue - Representation

- Another representation for a Priority Queue is to use a binary heap, where the root of the heap is the element with the highest priority.



Priority Queue - Representation on a binary heap

- When an element is pushed to the priority queue, it is simply added to the heap (and bubbled-up if needed)
- When an element is popped from the priority queue, the root is removed from the heap (and bubble-down is performed if needed)
- Top simply returns the root of the heap.

Priority Queue - Representation

- Let's complete our table with the complexity of the operations if we use a heap as representation:

Operation	**Sorted**	**Non-sorted**	**Heap**
push	O(n)	$\Theta(1)$	$O(log_2 n)$
pop	$\Theta(1)$	$\Theta(n)$	$O(log_2 n)$
top	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

- Consider the total complexity of the following sequence of operations:
  - start with an empty priority queue
  - push n random elements to the priority queue
  - perform pop n times

Priority Queue - Applications

- Problems where a priority queue can be used:
  - Triage procedure in the hospitals
  - Scholarship allocation - see Seminar 5
  - Give me a ticket on an airplane (war story from Steven S. Skiena: *The Algorithm Design Manual*, Second Edition, page 118)

Priority Queue - Extension

- We have discussed the *standard* interface of a Priority Queue, one that contains the following operations:
  - push
  - pop
  - top
  - isEmpty
  - init
- Sometimes, depending on the problem to be solved, it can be useful to have the following three operations as well:
  - increase the priority of an existing element
  - delete an arbitrary element
  - merge two priority queues

Priority Queue - Extension

- What is the complexity of these three extra operations if we use as representation a binary heap?
  - Increasing the priority of an existing element is $O(log_2 n)$ if we know the position where the element is.
  - Deleting an arbitrary element is $O(log_2 n)$ if we know the position where the element is.
  - Merging two priority queues has complexity $\Theta(n)$ (assume both priority queues have n elements).

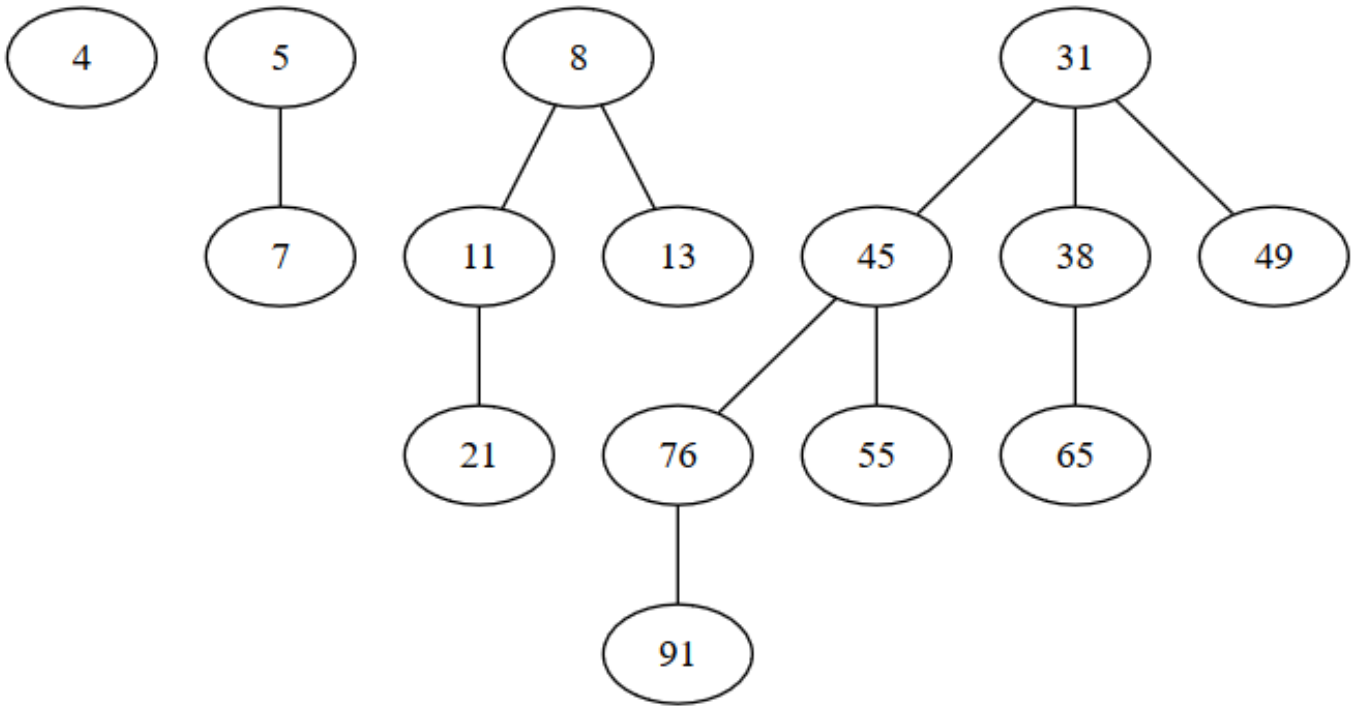
Priority Queue - Other representations

- If we do not want to merge priority queues, a binary heap is a good representation. If we need the merge operation, there are other heap data structures that can be used, which offer a better complexity.
- Out of these data structures we are going to discuss one: the *binomial heap*.

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

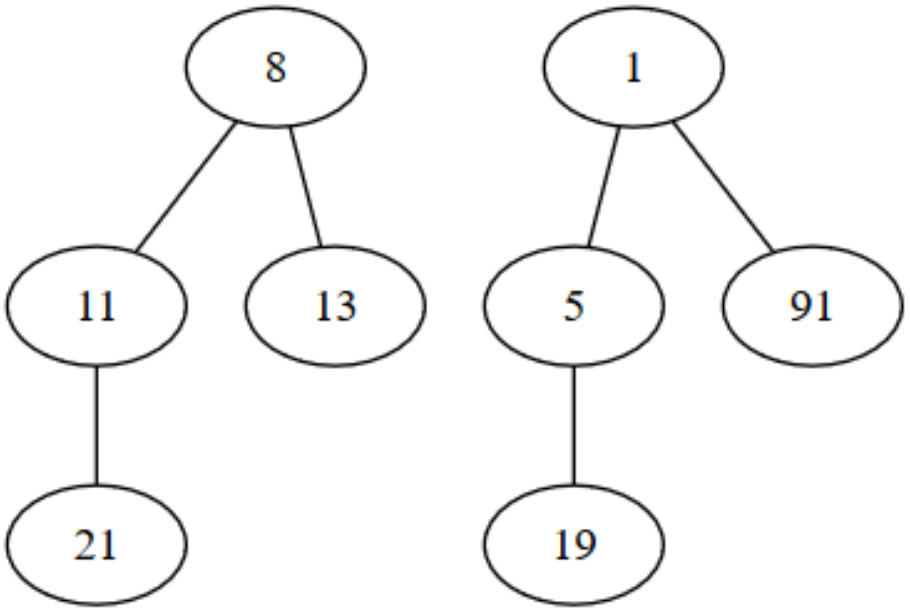


Binomial trees of order 0, 1, 2 and 3

Binomial tree

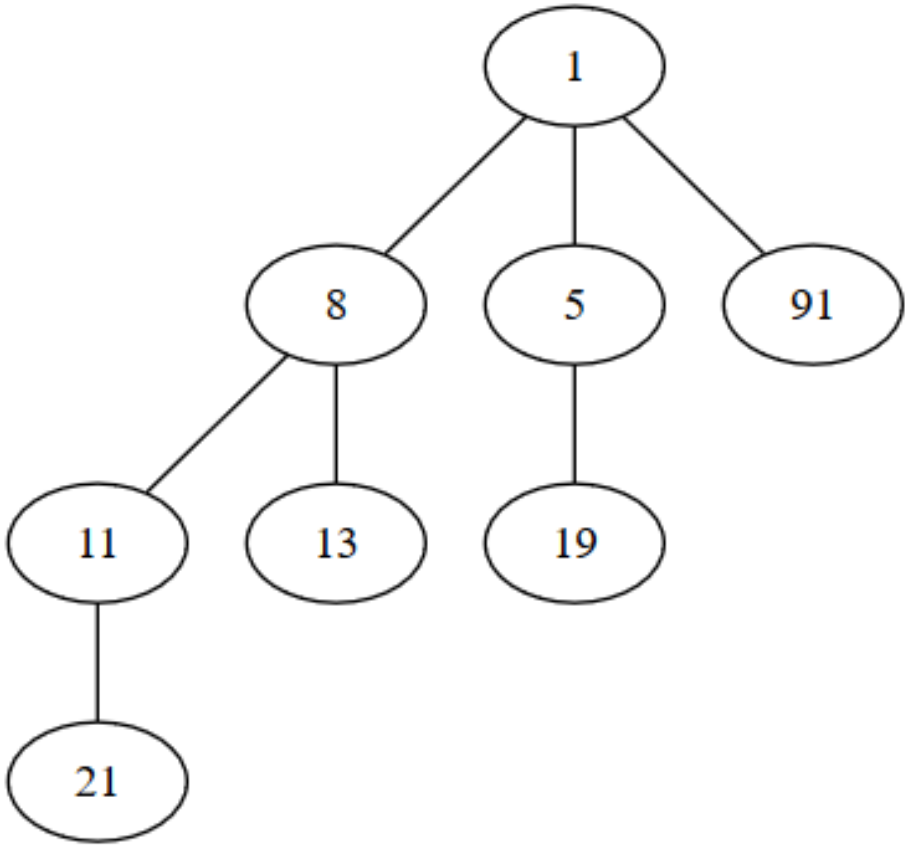
- A binomial tree of order k has exactly 2^k nodes.
- The height of a binomial tree of order k is k.
- If we delete the root of a binomial tree of order k, we will get k binomial trees, of orders $k-1$, $k-2$, ..., 2, 1, 0.
- Two binomial trees of the same order k can be merged into a binomial tree of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

Binomial tree - Merge I



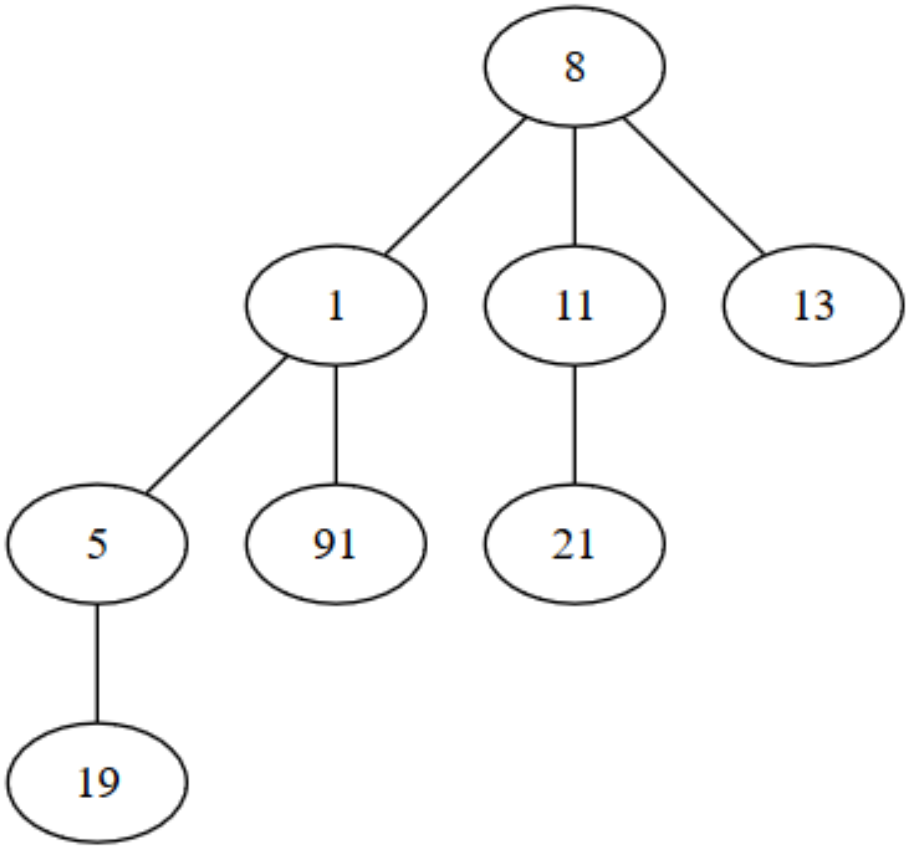
Before merge we have two binomial trees of order 2

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

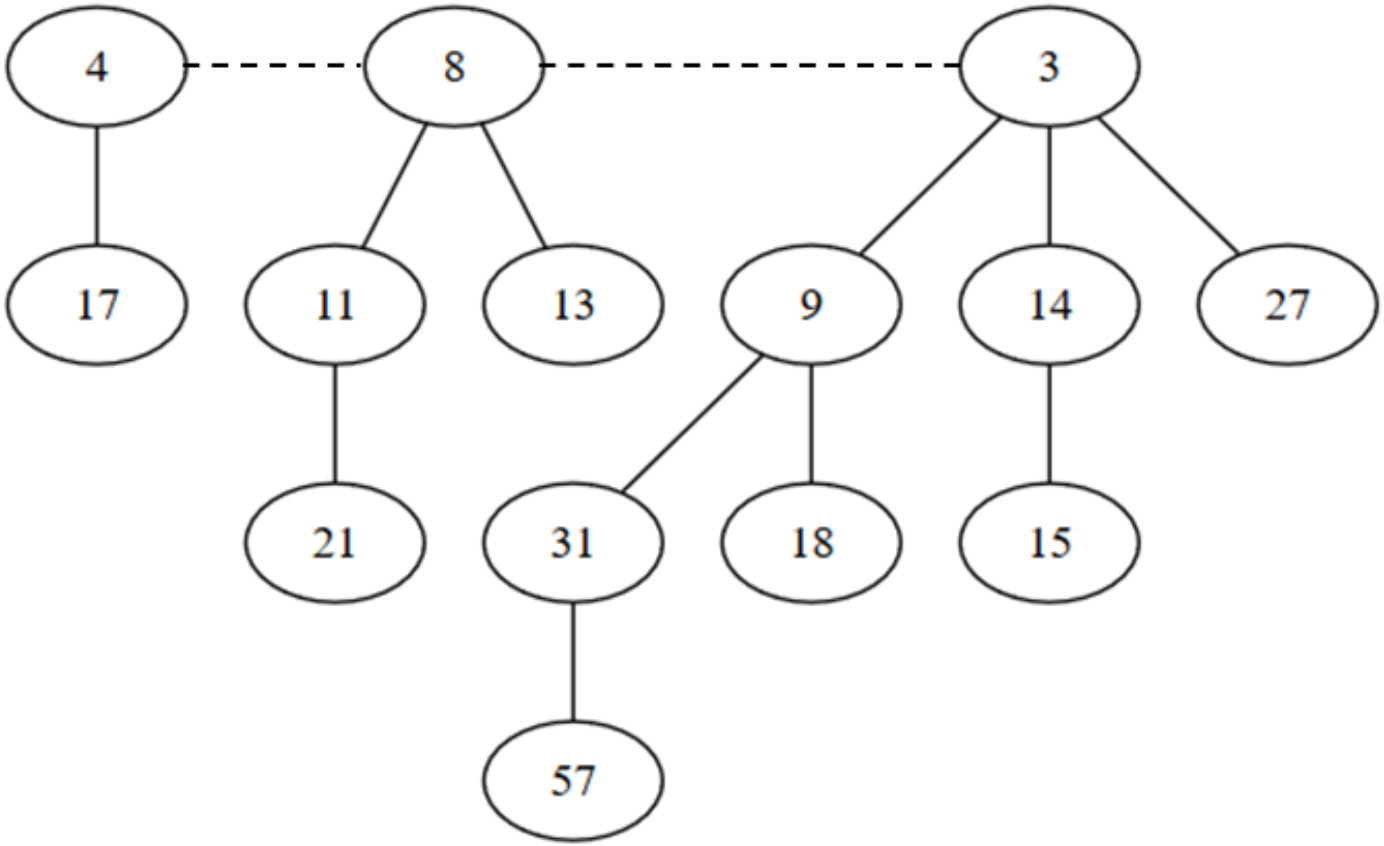
Binomial tree - representation

- 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.
  - 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.

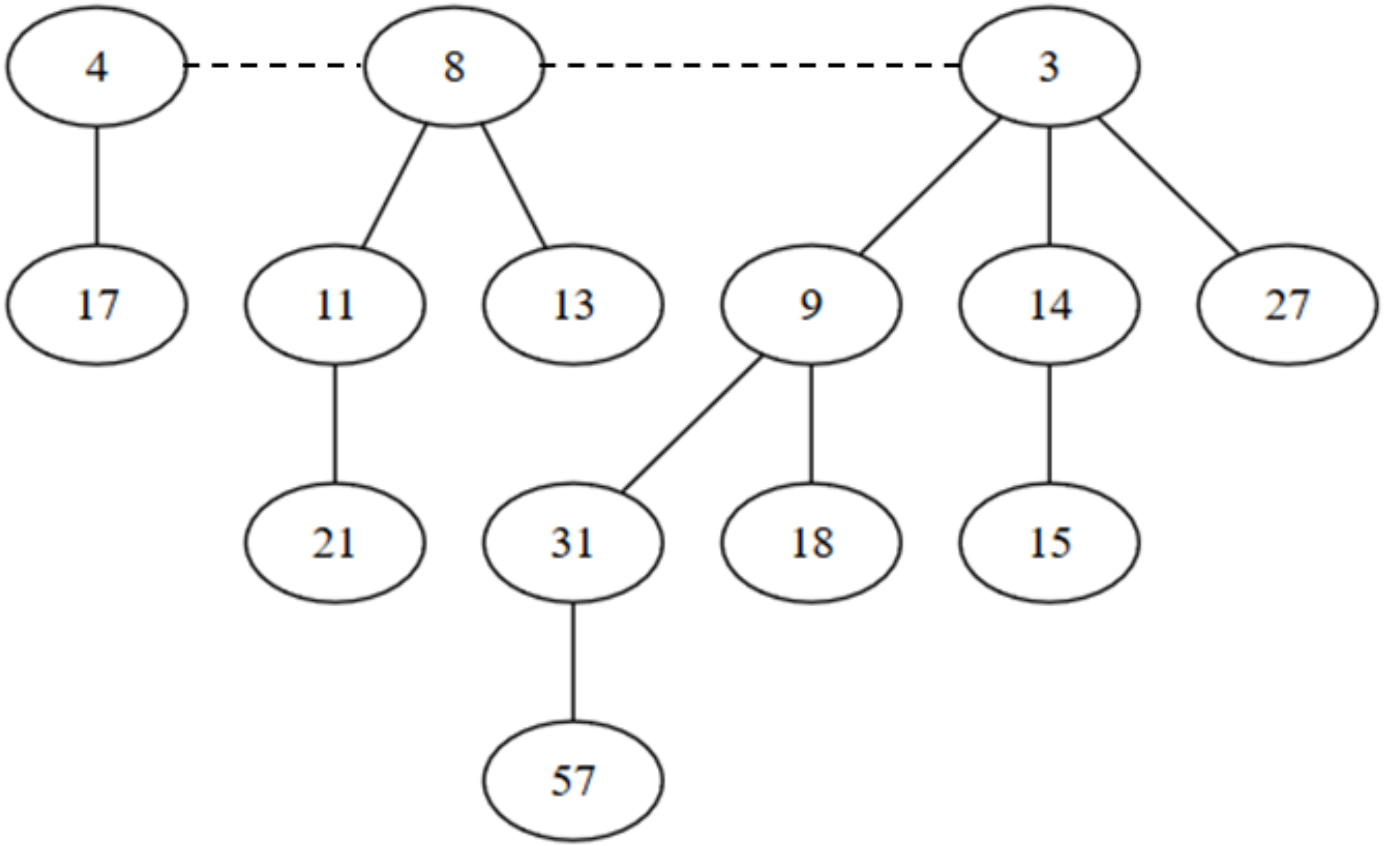
- Since both binomial heaps are sorted linked lists, the first step is to *merge* the two linked lists (standard merge algorithm for two sorted linked lists).

- 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.

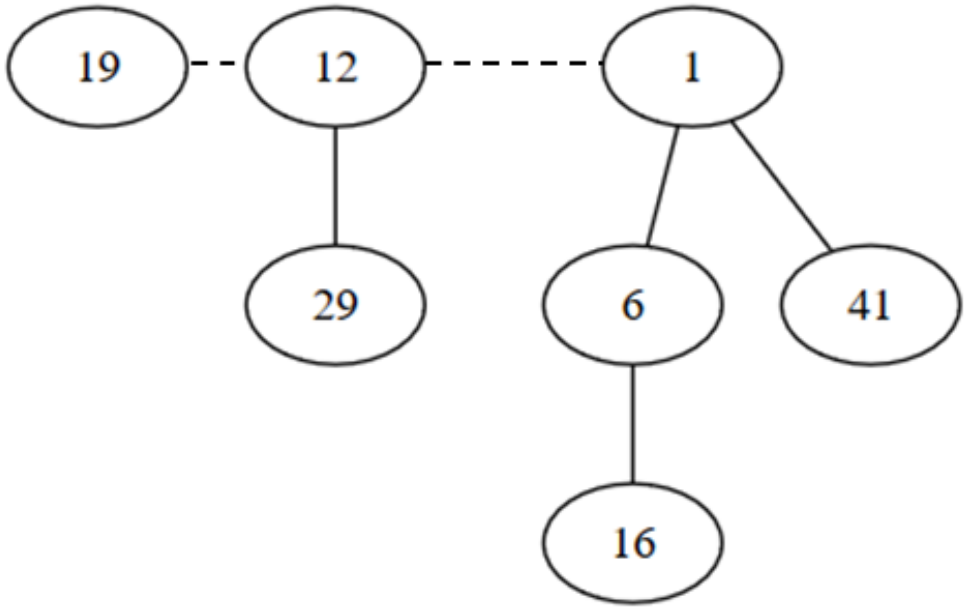
Binomial heap - merge - example I

- Let's merge the following two binomial heaps:

Binomial heap - merge - example II

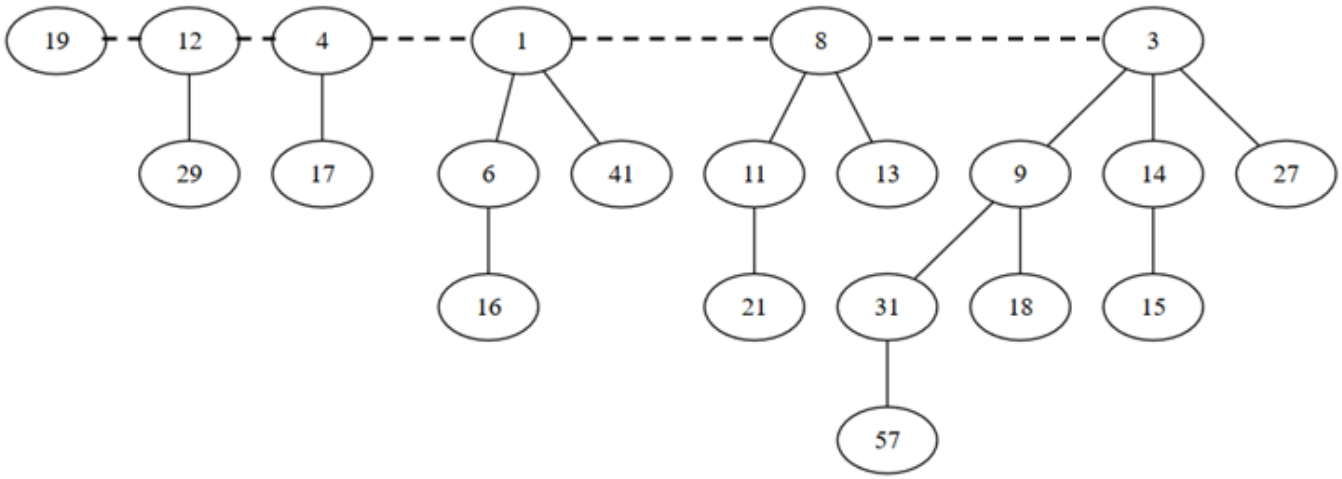


Binomial heap - merge - example III



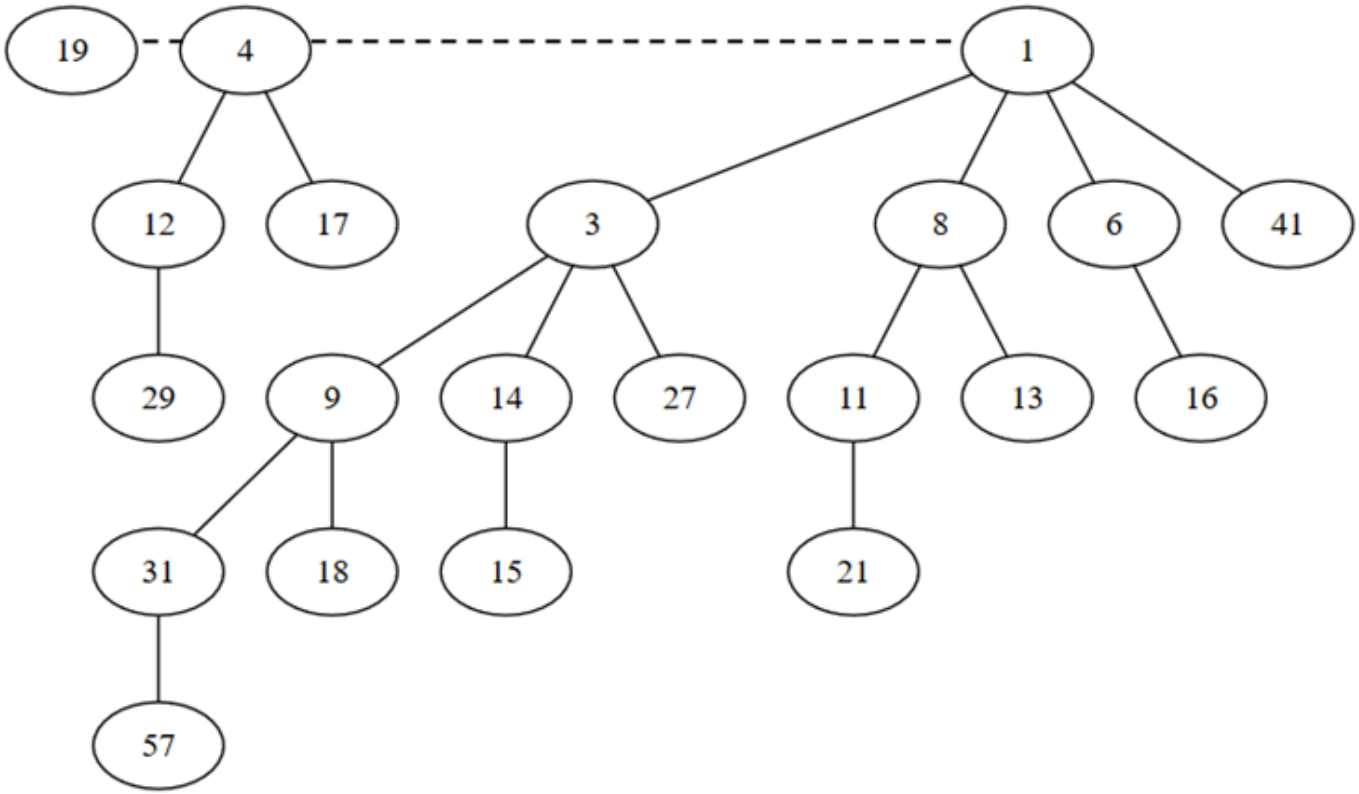
Binomial heap - merge - example IV

- After merging the two linked lists of binomial trees:



Binomial heap - merge - example V

- 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

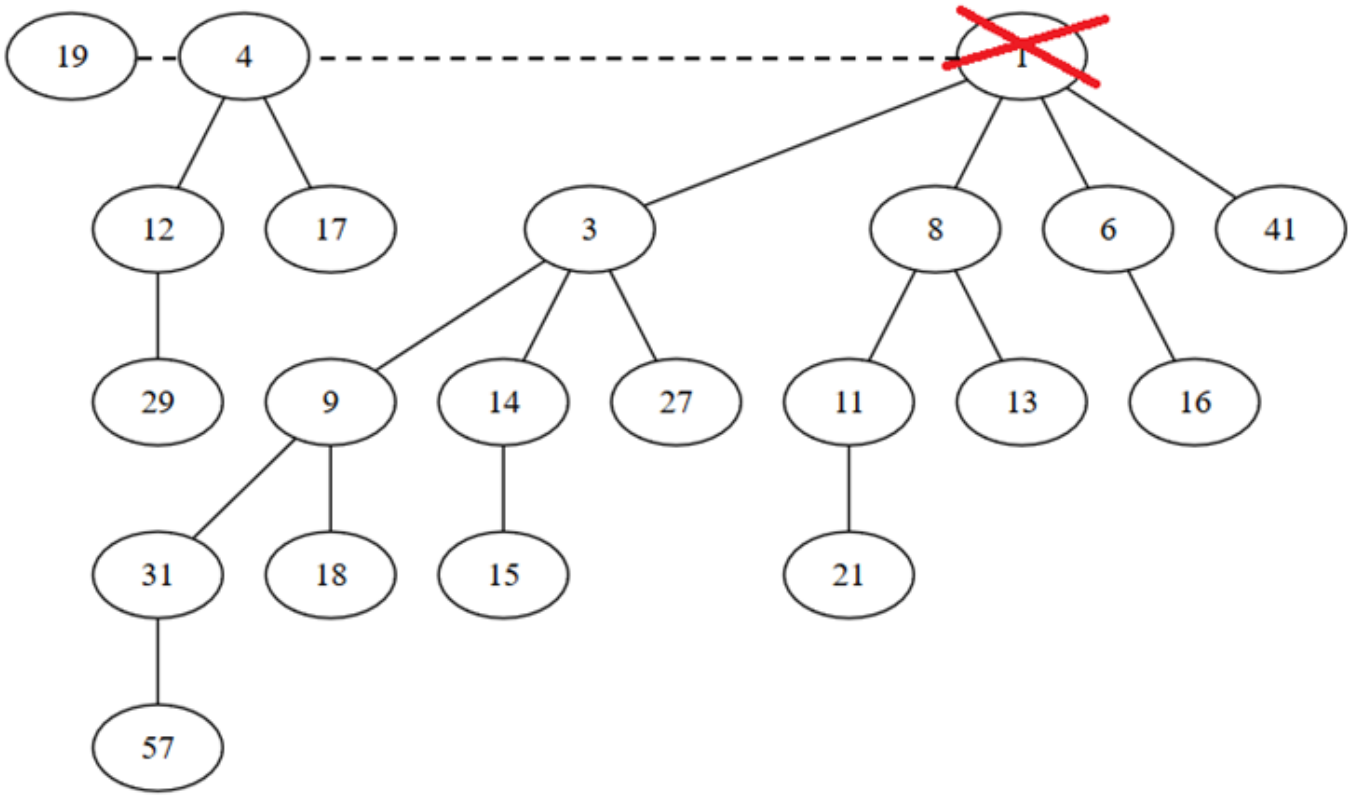
- Most of the other operations that we have for the binomial heap (because we need them for the priority queue) will use the merge operation presented above.
- *Push operation:* Inserting a new element means creating a binomial heap with just that element and merging it with the existing one. Complexity 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.

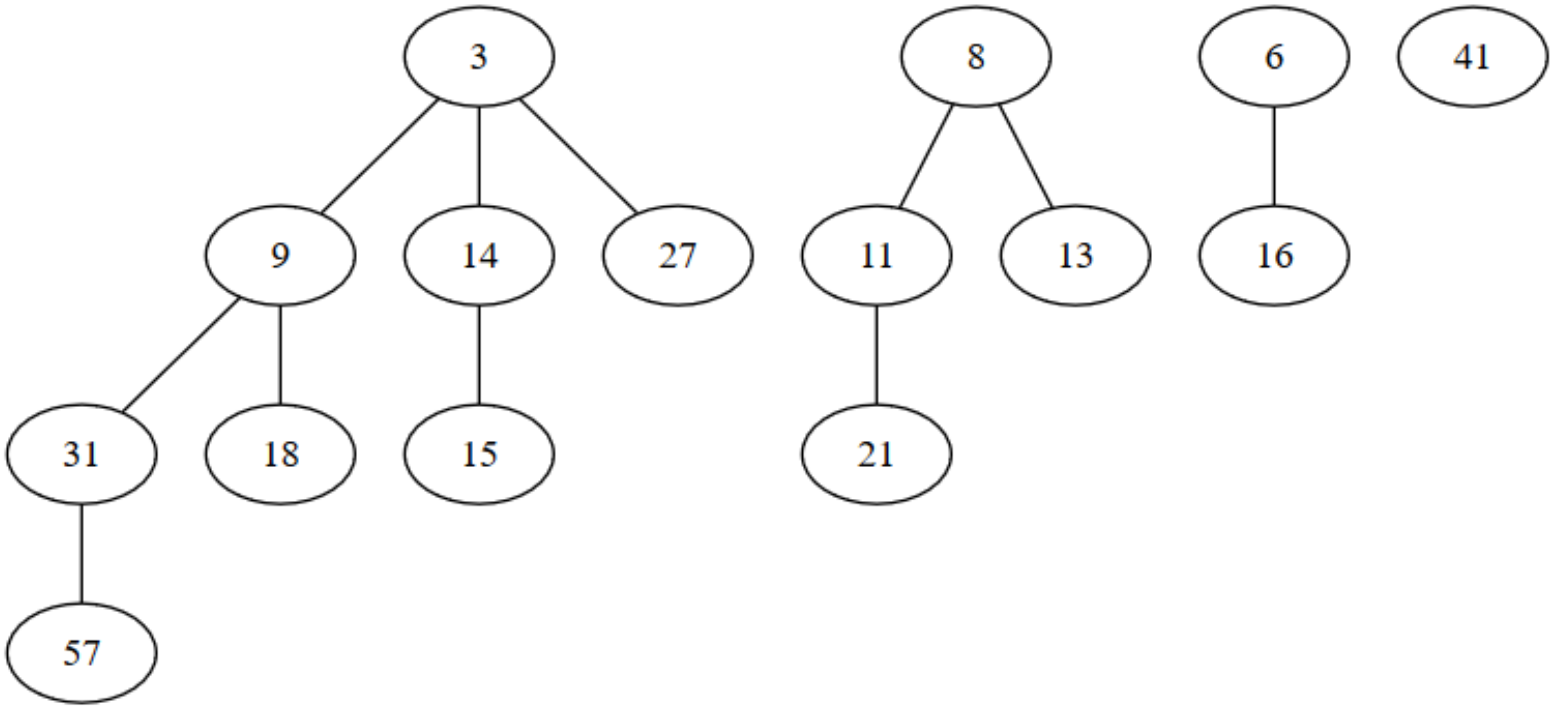
Binomial heap - other operations III

- The minimum is one of the roots.



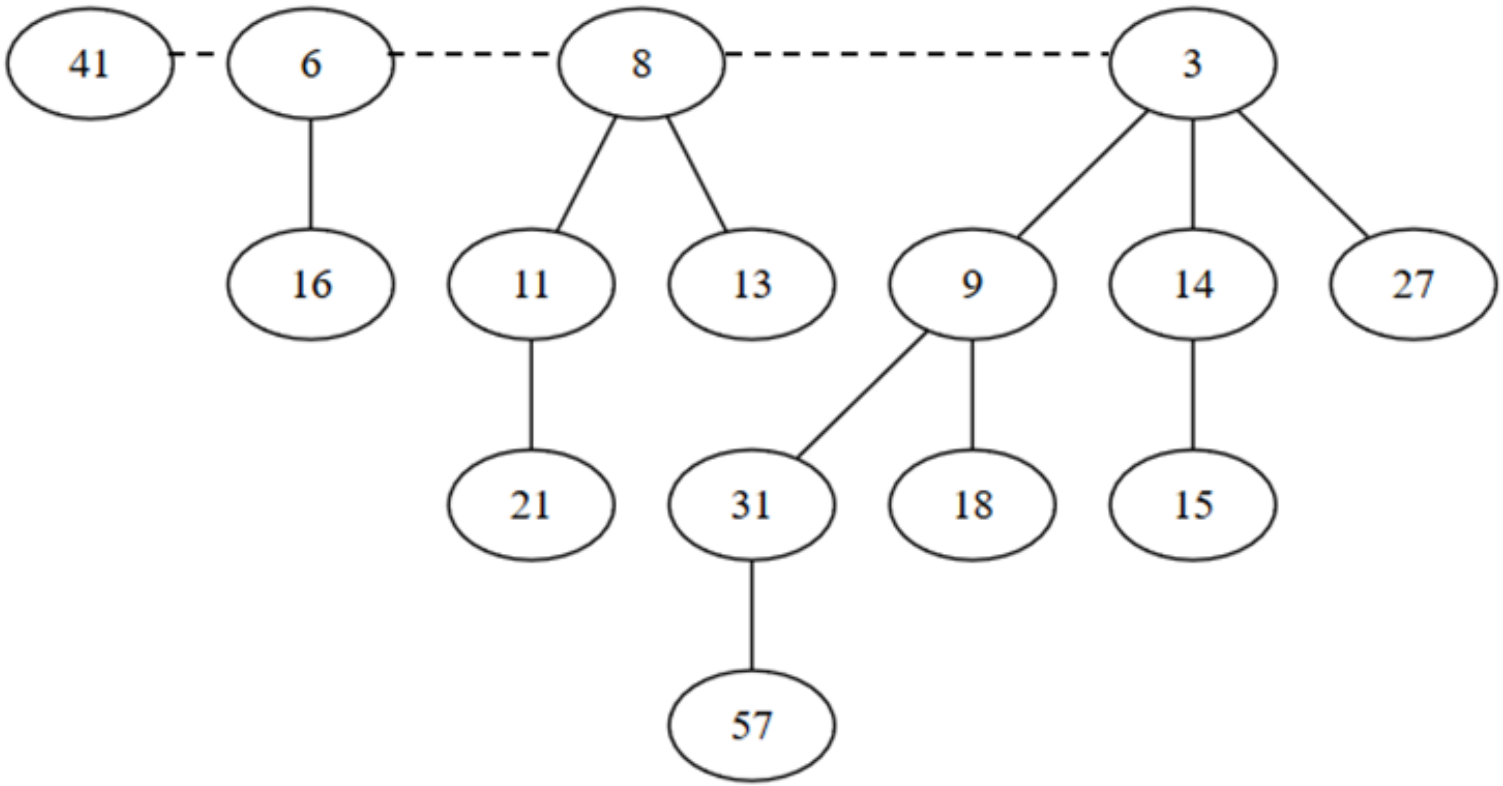
Binomial heap - other operations IV

- Break the corresponding tree into k binomial trees



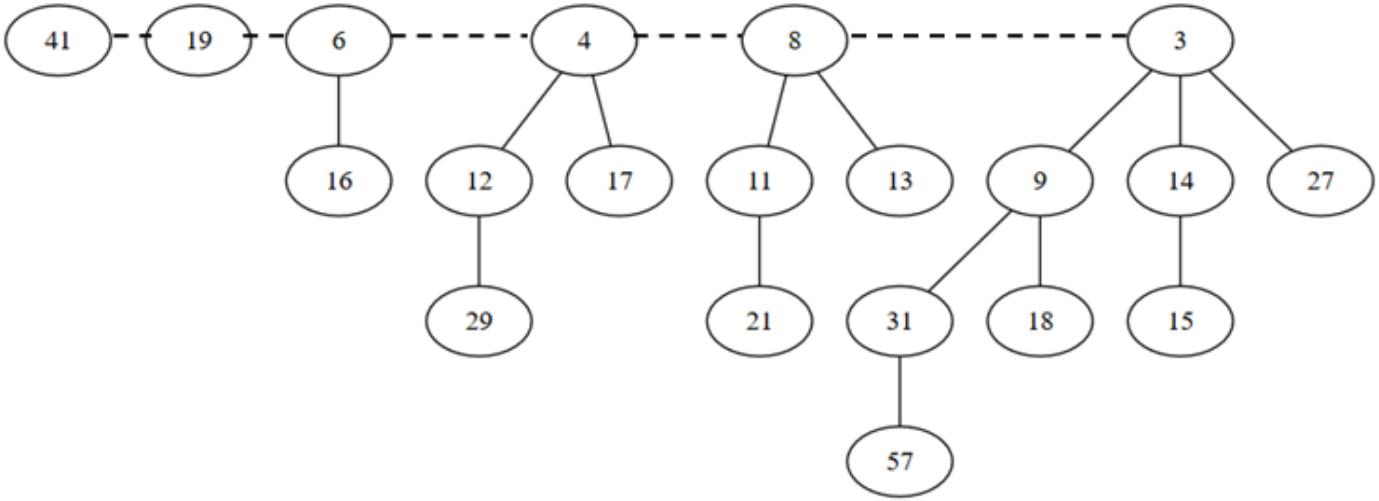
Binomial heap - other operations V

- Create a binomial heap of these trees



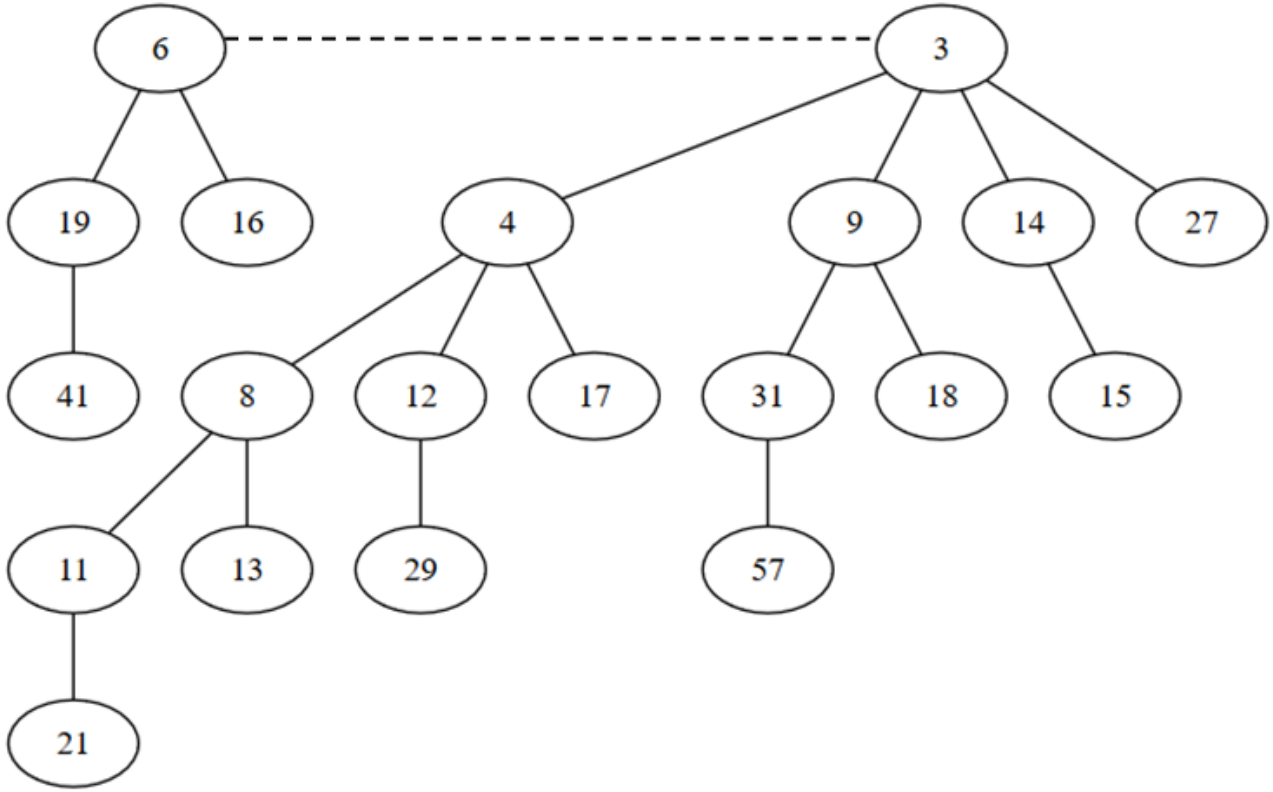
Binomial heap - other operations VI

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



Binomial heap - other operations VII

- After the transformation



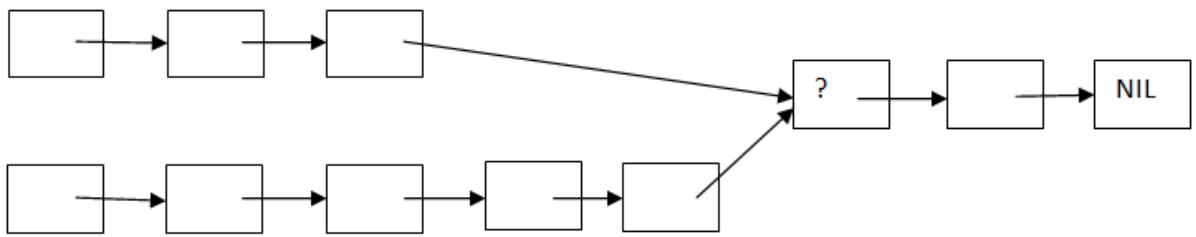
- The complexity of the remove-minimum operation is $O(log_2 n)$

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: $O(log_2 n)$

Think about it - Linked Lists

- Write a non-recursive algorithm to reverse a singly linked list with $\Theta(n)$ time complexity, using constant space/memory.
- Suppose there are two singly linked lists both of which intersect at some point and become a single linked list (see the image below). The number of nodes in the two list before the intersection is not known and may be different in each list. Give an algorithm for finding the merging point (hint - use a Stack)



Think about it - Stacks and Queues I

- How can we implement a Stack using two Queues? What will be the complexity of the operations?
- How can we implement a Queue using two Stacks? What will be the complexity of the operation?
- How can we implement two Stacks using only one array? The stack operations should throw an exception only if the total number of elements in the two Stacks is equal to the size of the array.

Think about it - Stacks and Queues II

- Given a string of lower-case characters, recursively remove adjacent duplicate characters from the string. For example, for the word "mississippi" the result should be "m".

- Given an integer k and a queue of integer numbers, how can we reverse the order of the first k elements from the queue? For example, if $k=4$ and the queue has the elements [10, 20, 30, 40, 50, 60, 70, 80, 90], the output should be [40, 30, 20, 10, 50, 60, 70, 80, 90].

Think about it - Priority Queues

- How can we implement a stack using a Priority Queue?
- How can we implement a queue using a Priority Queue?