# DATA STRUCTURES AND ALGORITHMS

## LECTURE 8

Lect. PhD. Onet-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2019 - 2020

# In Lecture 7

- Stack, Queue, Priority Queue
- Binary Heap

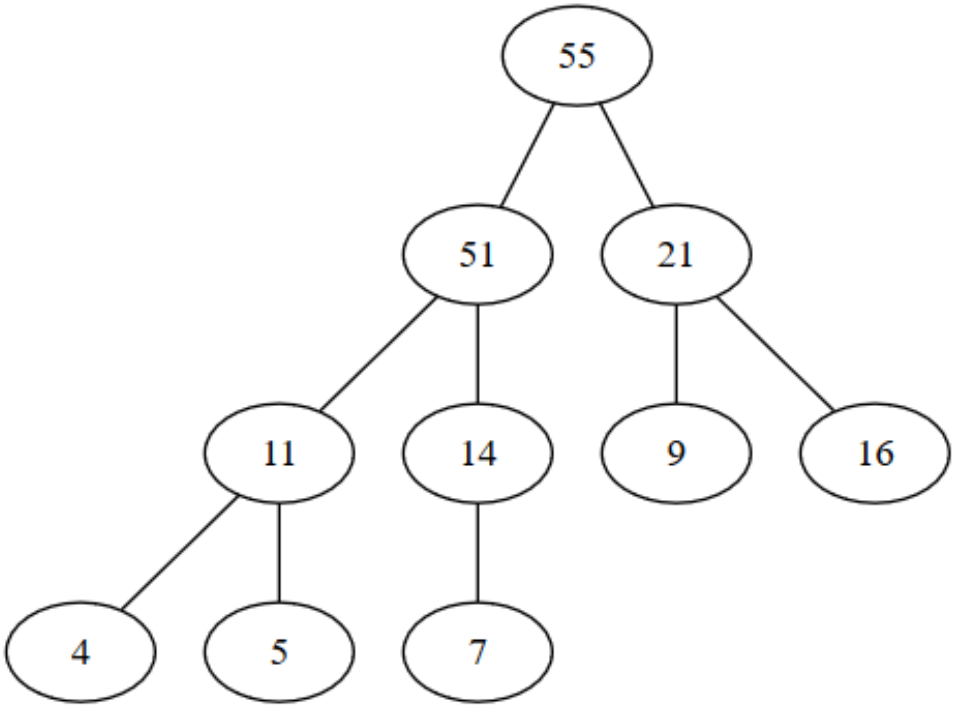
# Today

- Binary heap
- Binomial heap
- Hash tables

# Binary heap - recap

- It is a hybrid between a dynamic array and a binary tree: the elements are stored in a dynamic array, but we visualize the array in the form of a binary tree.
- For an element from position $i$ from the array its children are the elements from positions $2 * i$ and $2 * i + 1$ (if these positions exist in the array) and its parent is at position $i/2$.
- In order to have a valid binary heap we need a *heap structure* and a *heap property*

# Binary heap - example




- Example of a Max-heap

# Binary Heap - operations

- A heap can be used as representation for a Priority Queue and it has two specific operations:
  - add a new element in the heap (in such a way that we keep both the heap structure and the heap property).
  - remove (we always remove the root of the heap - no other element can be removed).

# Binary Heap - representation

Heap:

cap: Integer

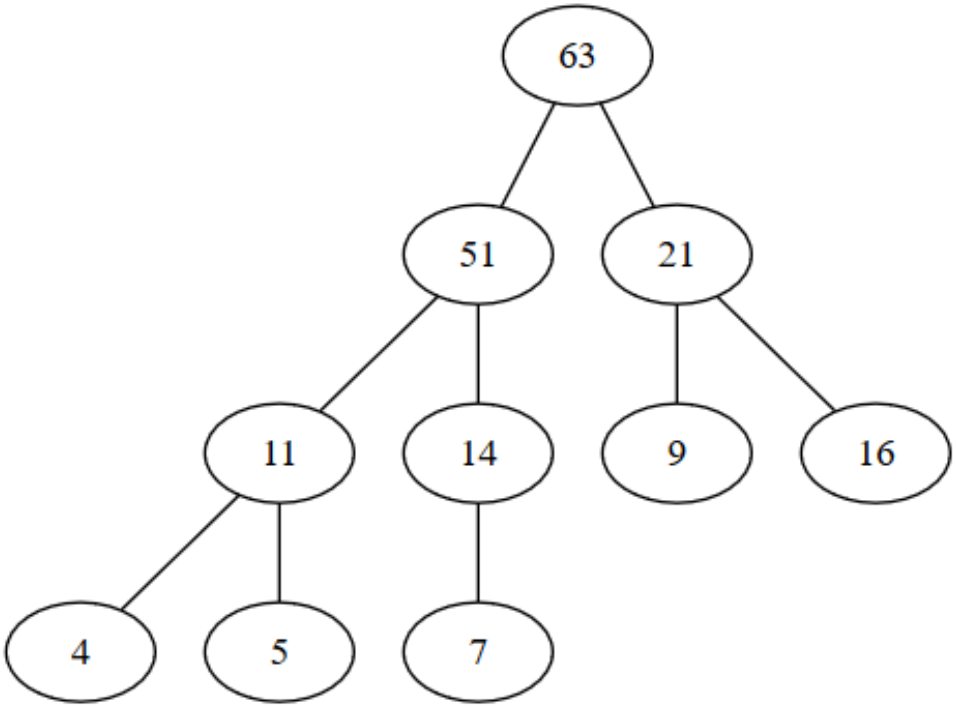
len: Integer

elems: TElem[]

- For the implementation we will assume that we have a MAX-HEAP.

# Binary Heap - Add - example

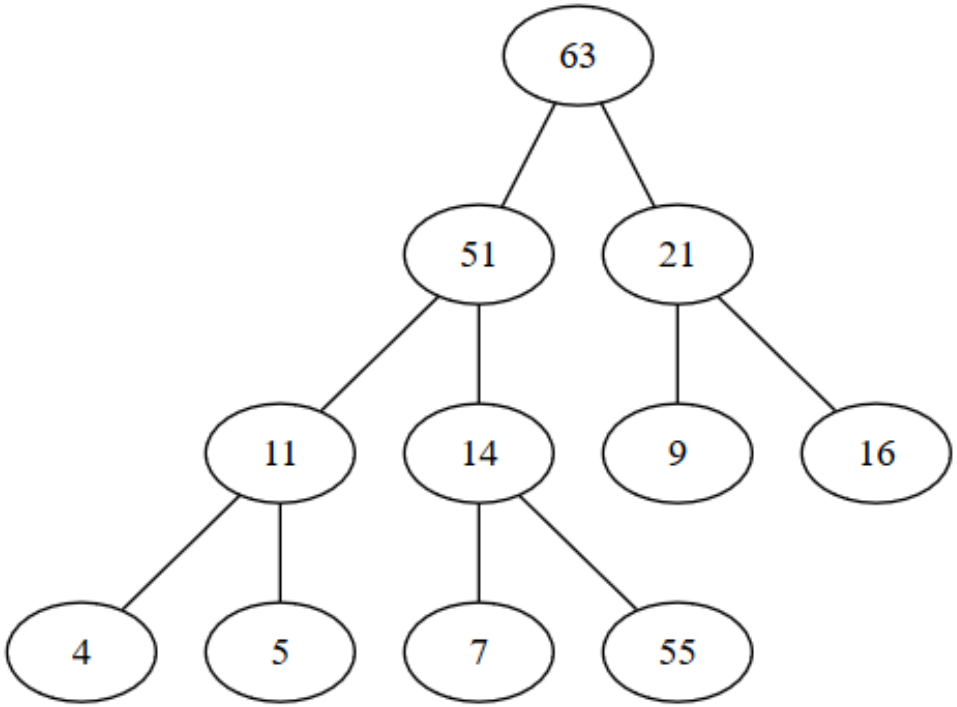
- Consider the following (MAX) heap:




- Let's add the number 55 to the heap.

# Binary Heap - Add - example

- In order to keep the *heap structure*, we will add the new node as the right child of the node 14 (and as the last element of the array in which the elements are kept).

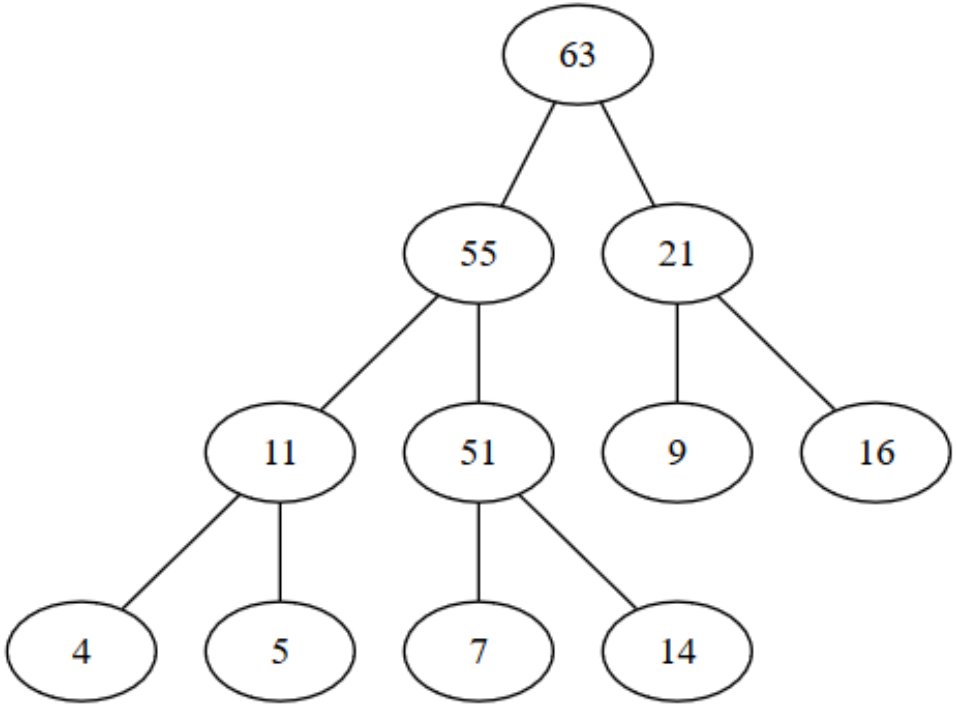


# Binary Heap - Add - example

- *Heap property* is not kept: 14 has as child node 55 (since it is a MAX-heap, each node has to be greater than or equal to its descendants).
- In order to restore the heap property, we will start a *bubble-up* process: we will keep swapping the value of the new node with the value of its parent node, until it gets to its final place. No other node from the heap is changed.

# Binary Heap - Add - example

- When *bubble-up* ends:



# Binary Heap - add

```
subalgorithm add(heap, e) is:
//heap - a heap
//e - the element to be added
    if heap.len = heap.cap then
        @ resize
    end-if
    heap.elems[heap.len+1] ← e
    heap.len ← heap.len + 1
    bubble-up(heap, heap.len)
end-subalgorithm
```

# Binary Heap - add

```
subalgorithm bubble-up (heap, p) is:
//heap - a heap
//p - position from which we bubble the new node up
    poz ← p
    elem ← heap.elems[p]
    parent ← p / 2
    while poz > 1 and elem > heap.elems[parent] execute
        //move parent down
        heap.elems[poz] ← heap.elems[parent]
        poz ← parent
        parent ← poz / 2
    end-while
    heap.elems[poz] ← elem
end-subalgorithm
```

- Complexity:

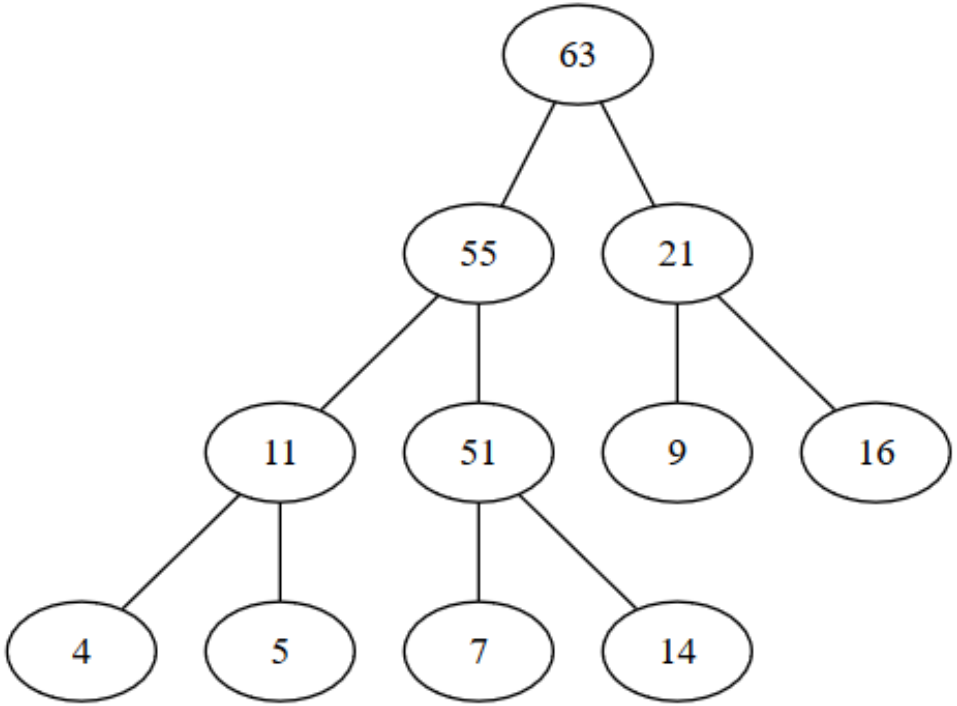
# Binary Heap - add

```
subalgorithm bubble-up (heap, p) is:
//heap - a heap
//p - position from which we bubble the new node up
    poz ← p
    elem ← heap.elems[p]
    parent ← p / 2
    while poz > 1 and elem > heap.elems[parent] execute
        //move parent down
        heap.elems[poz] ← heap.elems[parent]
        poz ← parent
        parent ← poz / 2
    end-while
    heap.elems[poz] ← elem
end-subalgorithm
```

- Complexity: $O(log_2 n)$
- Can you give an example when the complexity of the algorithm is less than $log_2 n$ (best case scenario)?

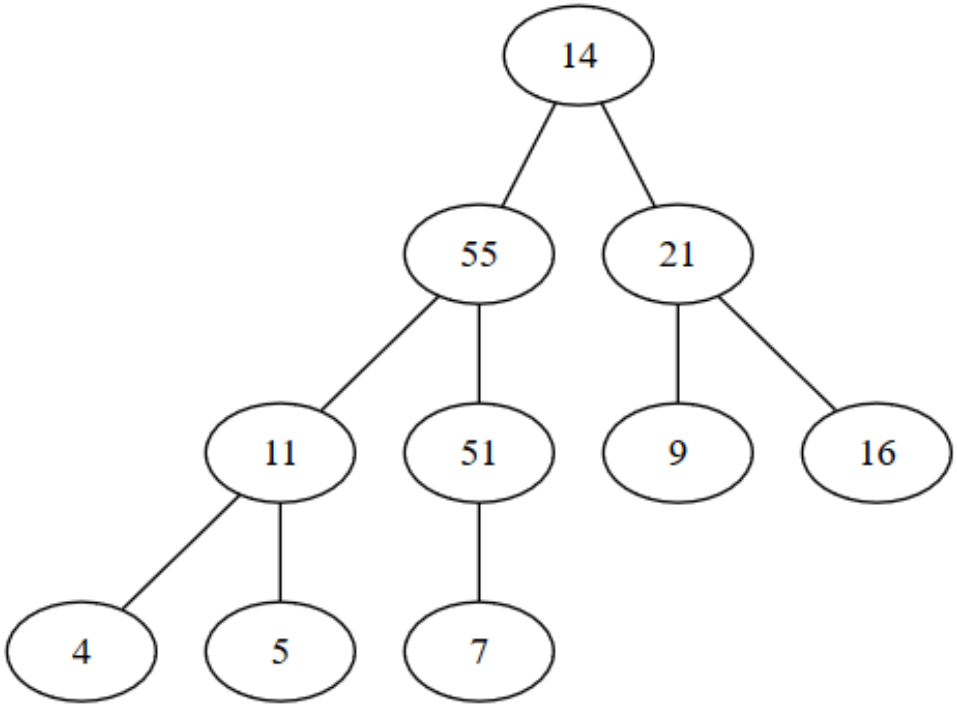
# Binary Heap - Remove - example

- From a heap we can only remove the root element.



# Binary Heap - Remove - example

- In order to keep the *heap structure*, when we remove the root, we are going to move the last element from the array to be the root.

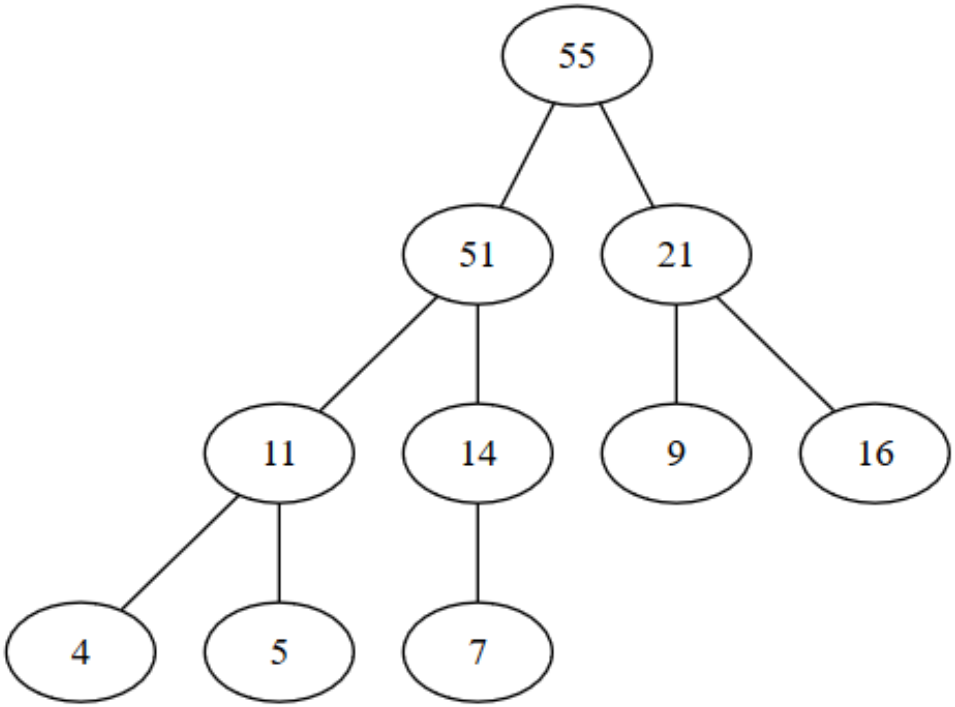


# Binary Heap - Remove - example

- *Heap property* is not kept: the root is no longer the maximum element.
- In order to restore the heap property, we will start a *bubble-down* process, where the new node will be swapped with its maximum child, until it becomes a leaf, or until it will be greater than both children.

# Binary Heap - Remove - example

- When the bubble-down process ends:



# Binary Heap - remove

```
function remove(heap) is:
//heap - is a heap
   if heap.len = 0 then
      @ error - empty heap
    end-if
   deletedElem ← heap.elems[1]
   heap.elems[1] ← heap.elems[heap.len]
   heap.len ← heap.len - 1
   bubble-down(heap, 1)
   remove ← deletedElem
end-function
```

# Binary Heap - remove

```
subalgorithm bubble-down(heap, p) is:
//heap - is a heap
//p - position from which we move down the element
    poz ← p
    elem ← heap.elems[p]
    while poz < heap.len execute
        maxChild ← -1
        if poz * 2 ≤ heap.len then
        //it has a left child, assume it is the maximum
            maxChild ← poz*2
        end-if
        if poz*2+1 ≤ heap.len and heap.elems[2*poz+1]>heap.elems[2*poz] th
        //it has two children and the right is greater
            maxChild ← poz*2 + 1
        end-if
//continued on the next slide...
```

# Binary Heap - remove

```
    if maxChild ≠ -1 and heap.elems[maxChild] > elem then
        tmp ← heap.elems[poz]
        heap.elems[poz] ← heap.elems[maxChild]
        heap.elems[maxChild] ← tmp
        poz ← maxChild
    else
        poz ← heap.len + 1
        //to stop the while loop
    end-if
  end-while
end-subalgorithm
```

- Complexity:

# Binary Heap - remove

```
    if maxChild ≠ -1 and heap.elems[maxChild] > elem then
        tmp ← heap.elems[poz]
        heap.elems[poz] ← heap.elems[maxChild]
        heap.elems[maxChild] ← tmp
        poz ← maxChild
    else
        poz ← heap.len + 1
        //to stop the while loop
    end-if
  end-while
end-subalgorithm
```

- Complexity: $O(log_2 n)$
- Can you give an example when the complexity of the algorithm is less than $log_2 n$ (best case scenario)?

# Questions

- In a max-heap where can we find the:
  - maximum element of the array?

# Questions

- In a max-heap where can we find the:
  - maximum element of the array?
  - minimum element of the array?

# Questions

- In a max-heap where can we find the:
  - maximum element of the array?
  - minimum element of the array?
- Assume you have a MAX-HEAP and you need to add an operation that returns the minimum element of the heap. How would you implements this operation, using constant time and space? (Note: we only want to return the minimum, we do not want to be able to remove it).

# Exercises

- Consider an initially empty Binary MAX-HEAP and insert the elements 8, 27, 13, 15*, 32, 20, 12, 50*, 29, 11* in it. Draw the heap in the tree form after the insertion of the elements marked with a * (3 drawings). Remove 3 elements from the heap and draw the tree form after every removal (3 drawings).

- Insert the following elements, in this order, into an initially empty MIN-HEAP: 15, 17, 9, 11, 5, 19, 7. Remove all the elements, one by one, in order from the resulting MIN-HEAP. Draw the heap after every second operation (after adding 17, 11, 19, etc.)

# Heap-sort

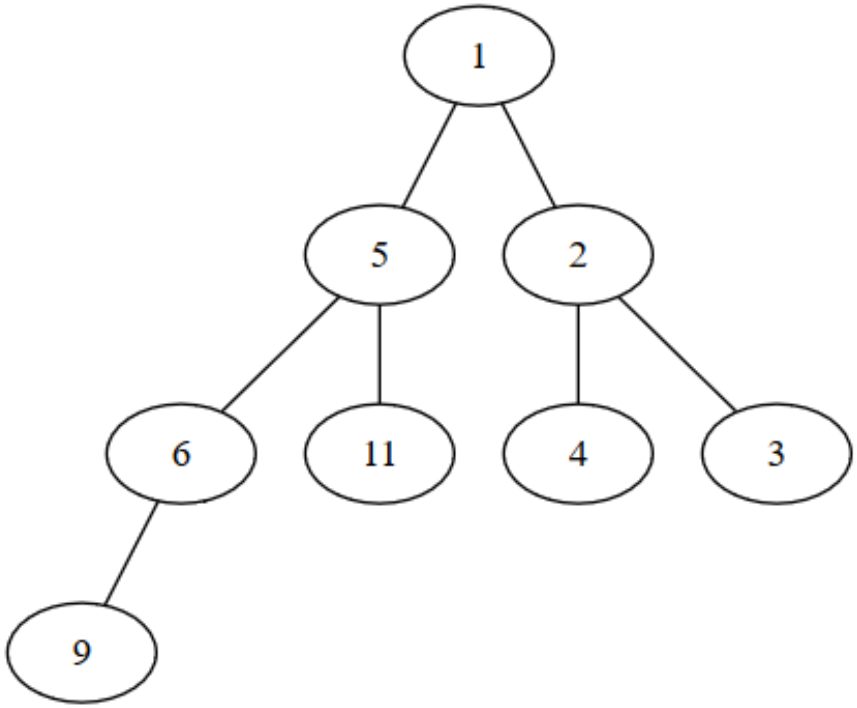
- There is a sorting algorithm, called *Heap-sort*, that is based on the use of a heap.
- In the following we are going to assume that we want to sort a sequence in ascending order.
- Let's sort the following sequence: [6, 1, 3, 9, 11, 4, 2, 5]

# Heap-sort - Naive approach

- Based on what we know so far, we can guess how heap-sort works:
  - Build a min-heap adding elements one-by-one to it.
  - Start removing elements from the min-heap: they will be removed in the sorted order.

# Heap-sort - Naive approach

- The heap when all the elements were added:




- When we remove the elements one-by-one we will have: 1, 2, 3, 4, 5, 6, 9, 11.

# Heap-sort - Naive approach

- What is the time complexity of the heap-sort algorithm described above?

# Heap-sort - Naive approach

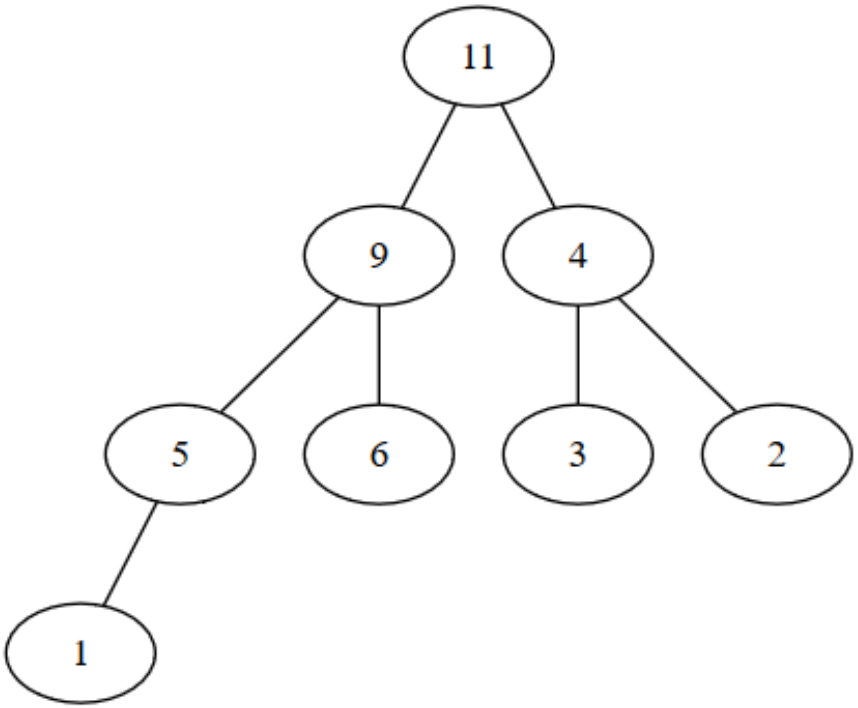
- What is the time complexity of the heap-sort algorithm described above?
- The time complexity of the algorithm is $O(n * log_2 n)$
- What is the extra space complexity of the heap-sort algorithm described above (do we need an extra array)?

# Heap-sort - Naive approach

- What is the time complexity of the heap-sort algorithm described above?
- The time complexity of the algorithm is $O(n * log_2 n)$
- What is the extra space complexity of the heap-sort algorithm described above (do we need an extra array)?
- The extra space complexity of the algorithm is $\Theta(n)$ - we need an extra array.

# Heap-sort - Better approach

- If instead of building a min-heap, we build a max-heap (even if we want to do ascending sorting), we do not need the extra array.



# Heap-sort - Better approach

- We can improve the time complexity of building the heap as well.

# Heap-sort - Better approach

- We can improve the time complexity of building the heap as well.
  - If we have an unsorted array, we can transform it easier into a heap: the second half of the array will contain leaves, they can be left where they are.
  - Starting from the first non-leaf element (and going towards the beginning of the array), we will just call *bubble-down* for every element (bubble-down has as parameter the position where we bubble-down from, even if at remove we always start from position 1).
  - Time complexity of this approach: $O(n)$ (but removing the elements from the heap is still $O(n * log_2 n)$, so the complexity of heap sort is still $O(n * log_2 n)$

# 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 - Extension

- We have discussed the *standard* interface of a Priority Queue, the 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/decrease 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/decreasing the priority of an existing element is $O(log_2 n)$ if we know the position where the element is.
    - Change the priority and bubble-down (for decrease in a max-heap) or bubble-up (for increase in a max-heap) the element
  - Deleting an arbitrary element is $O(log_2 n)$ if we know the position where the element is.
    - Move the last element to the position where you remove from and bubble-it down.
  - Merging two priority queues has complexity $\Theta(n)$ (assume both priority queues have $n$ elements).
    - Concatenate the arrays and heapify the result.

# Priority Queue - Other representations

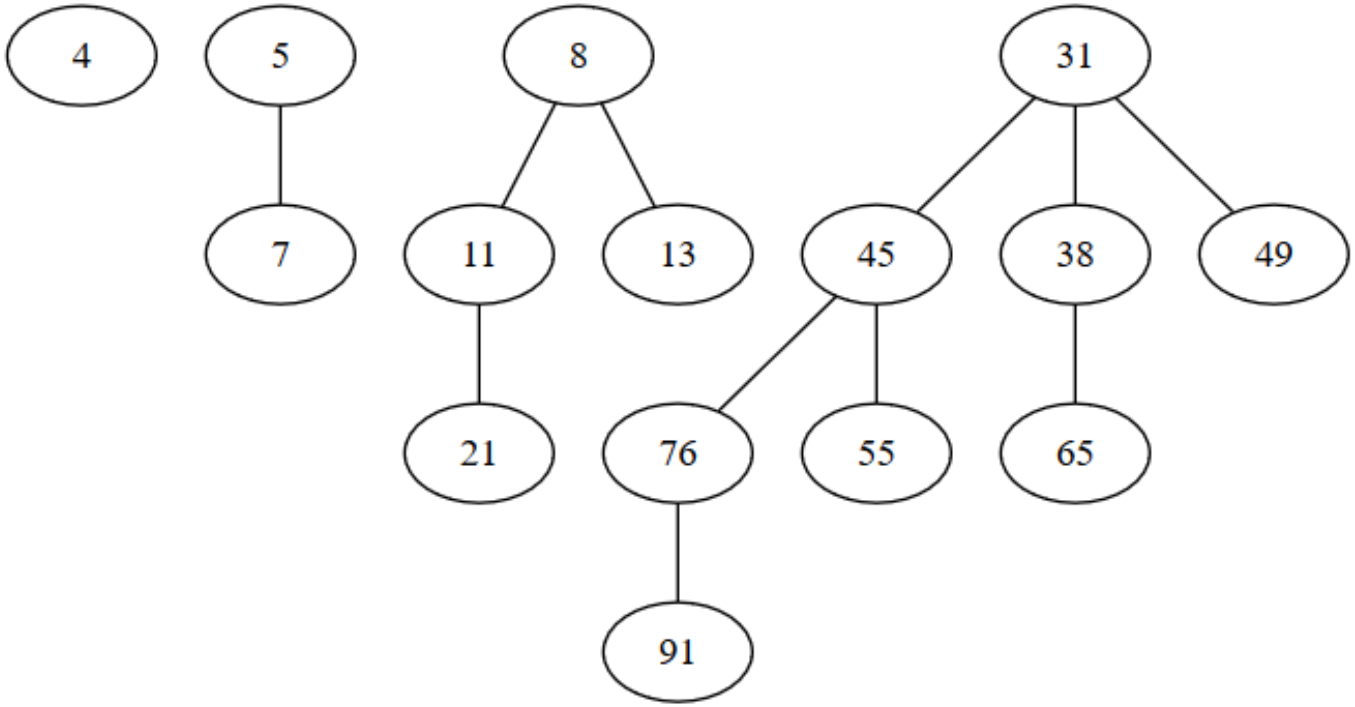
- If we do not want to merge priority queues, a binary heap is a good representation. If we need the merge operation, there are other heap data structures that can be used, which offer a better complexity.

- 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

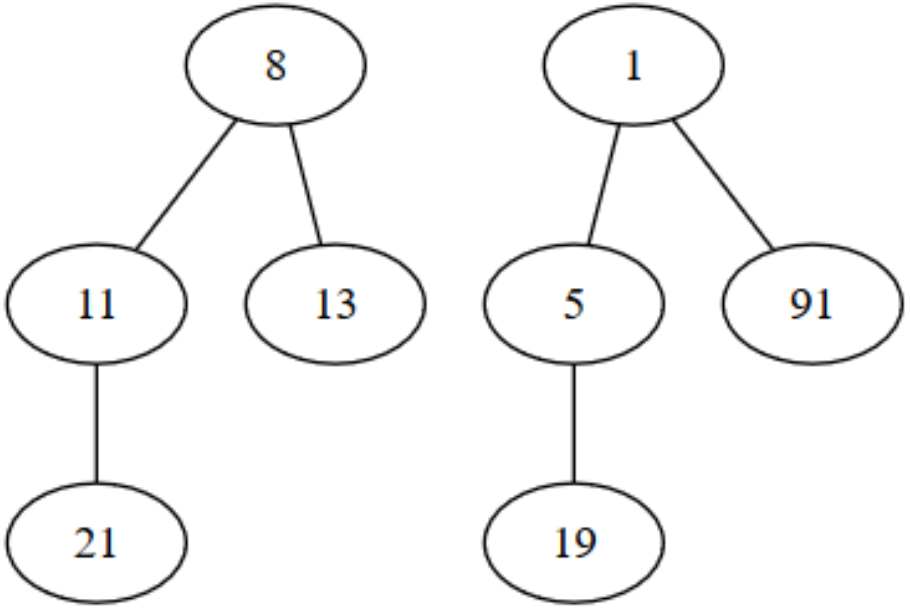



Binomial trees of order 0, 1, 2 and 3

# Binomial tree

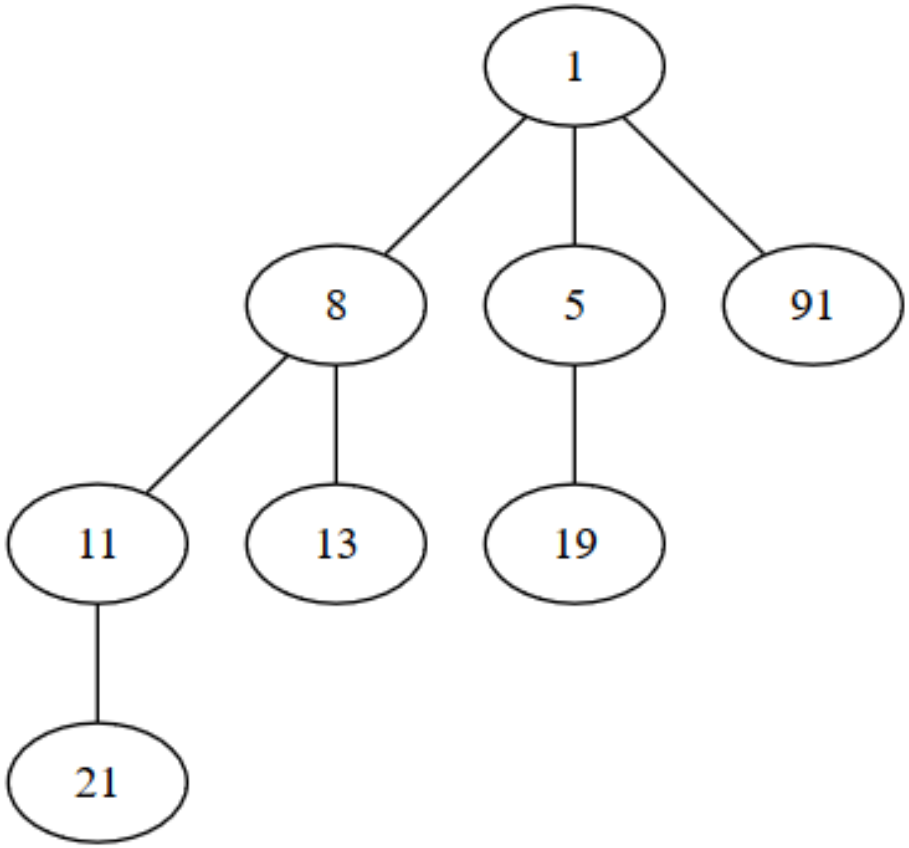
- A binomial tree of order k has exactly $2^k$ nodes.
- The height of a binomial tree of order k is k.
- If we delete the root of a binomial tree of order k, we will get $k$ binomial trees, of orders $k-1$, $k-2$, ..., 2, 1, 0.
- Two binomial trees of the same order $k$ can be merged into a binomial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

# Binomial tree - Merge I



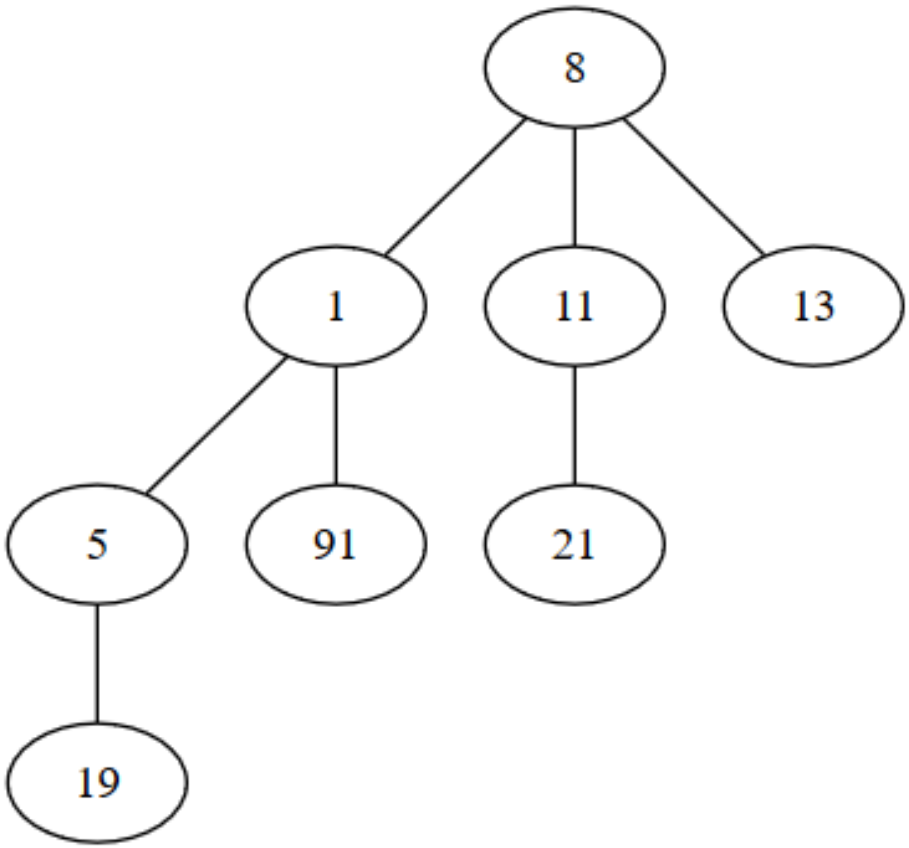

Before merge we have two binomial trees of order 2

# 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

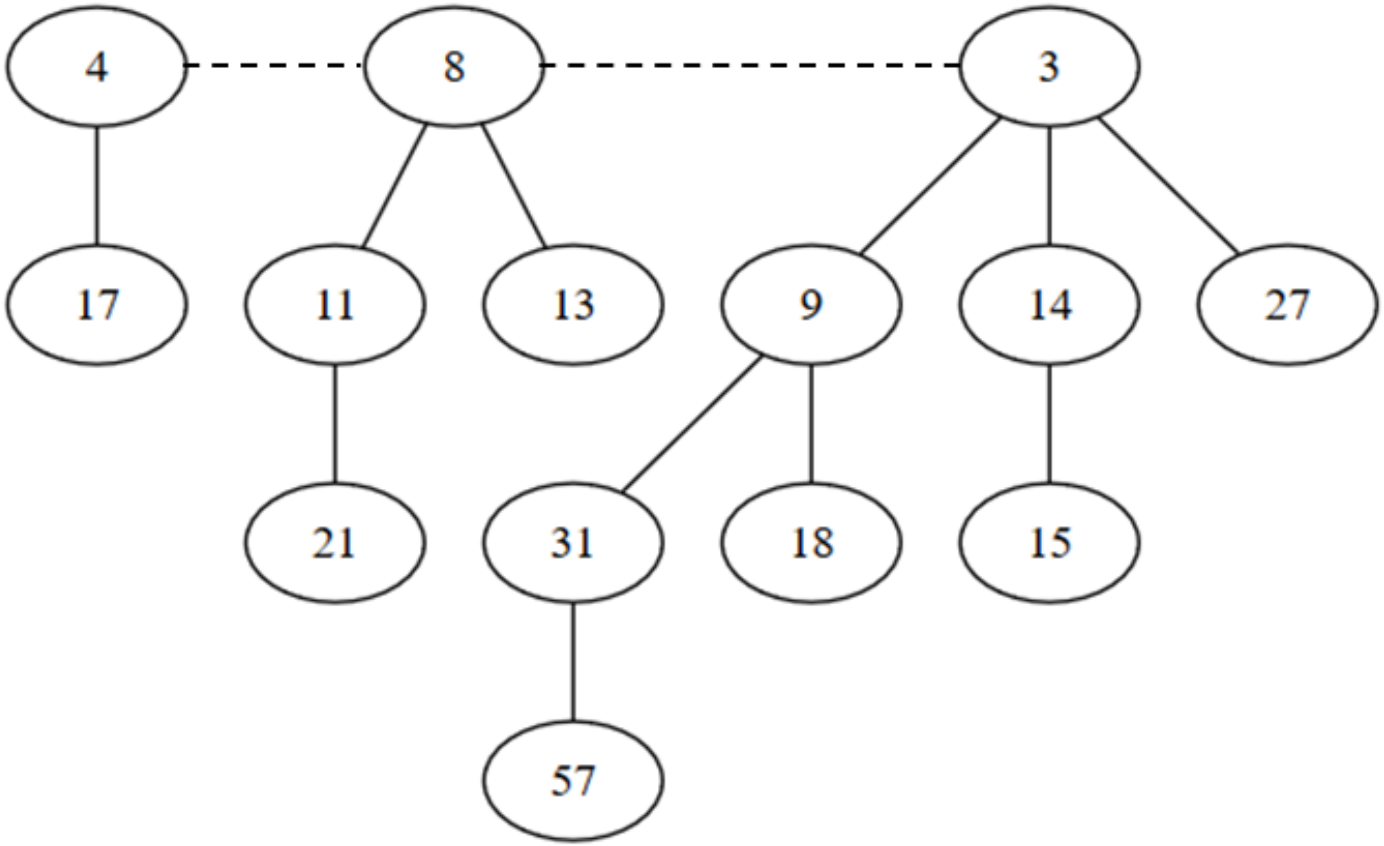
# Binomial tree - representation

- 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 or equal to 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

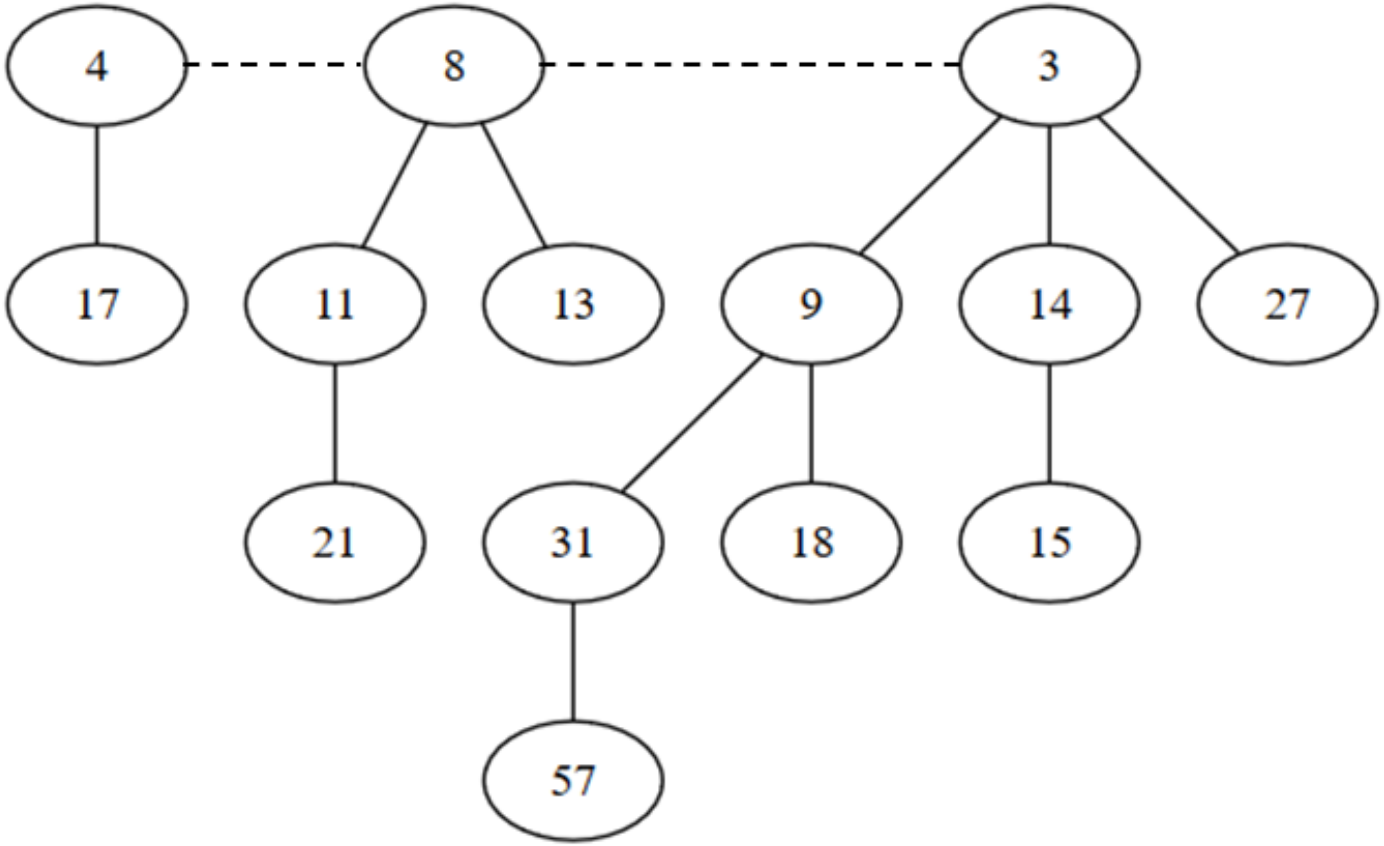
- The most interesting operation for two binomial heaps is the merge operation, which is used by other operations as well. After the merge operation the two previous binomial heaps will no longer exist, we will only have the result (destructive merge, we do not copy the existing trees).

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

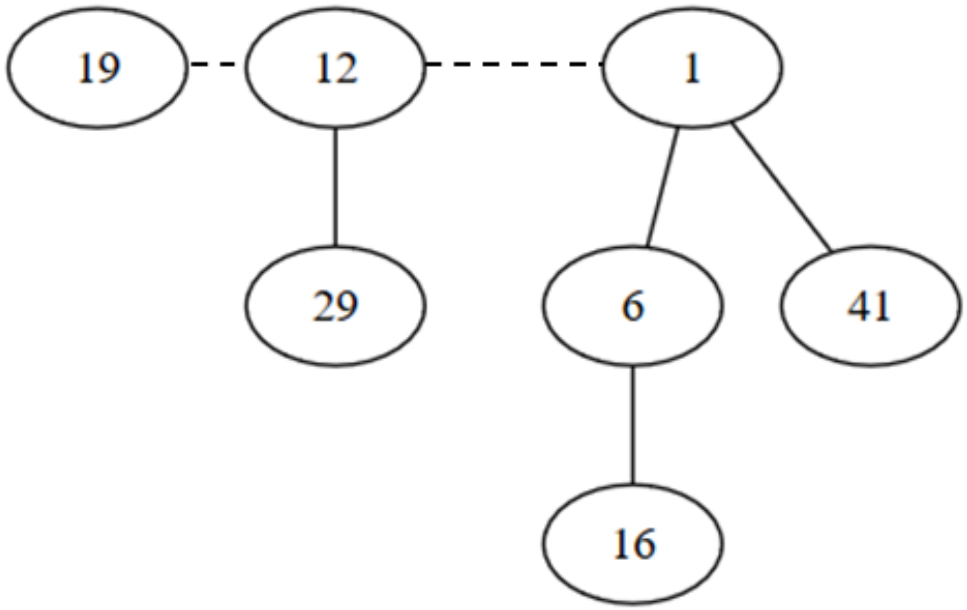
- The result of the merge can contain two binomial trees of the same order, so we have to iterate over the resulting list and transform binomial trees of the same order $k$ into a binomial tree of order $k+1$. When we merge the two binomial trees we must keep the heap property (this is how we decide which one is going to become the child of the other).

# Binomial heap - merge - example I

- Let's merge the following two binomial heaps:

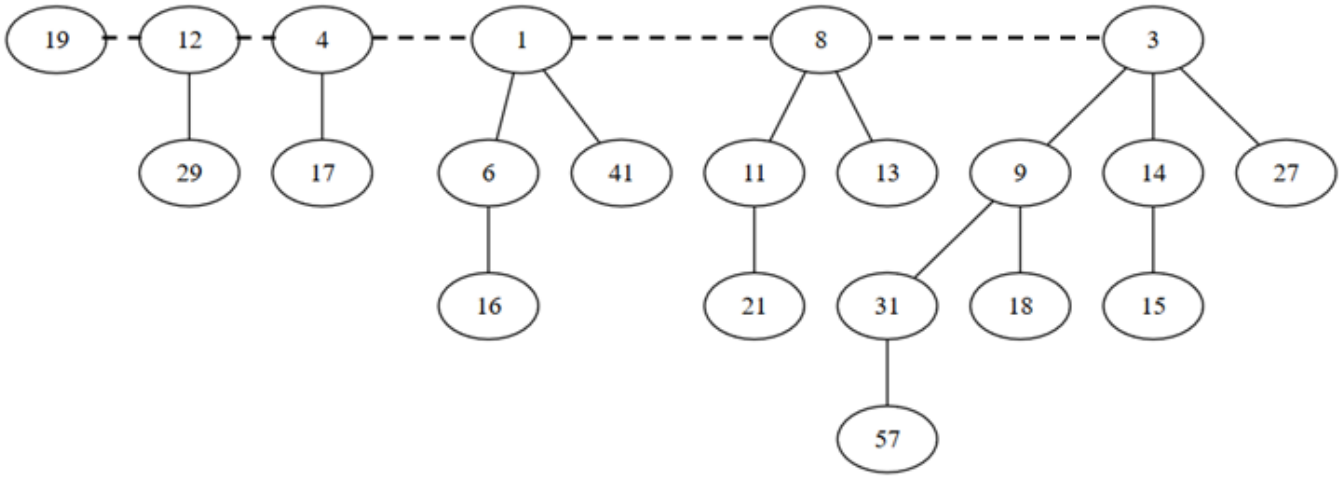


# Binomial heap - merge - example II



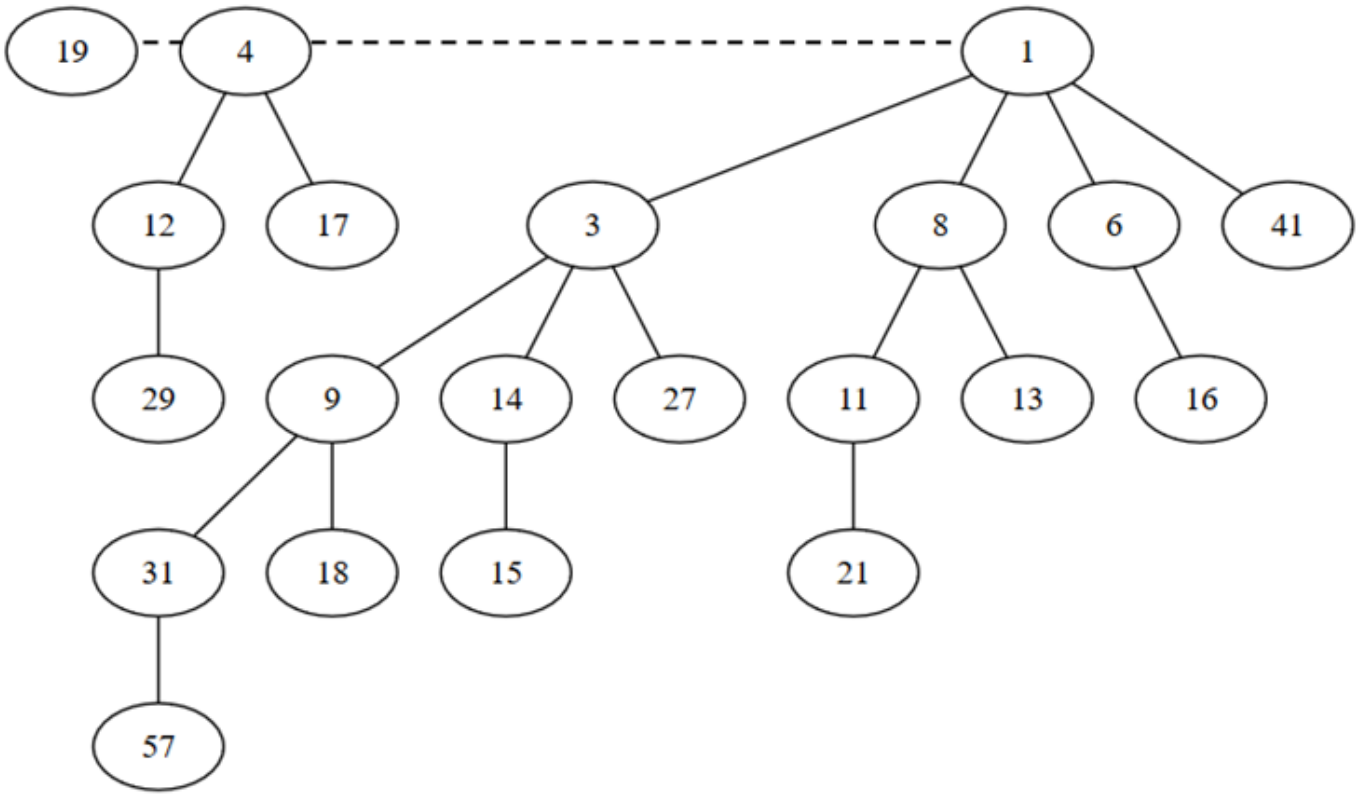
# Binomial heap - merge - example III

- 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

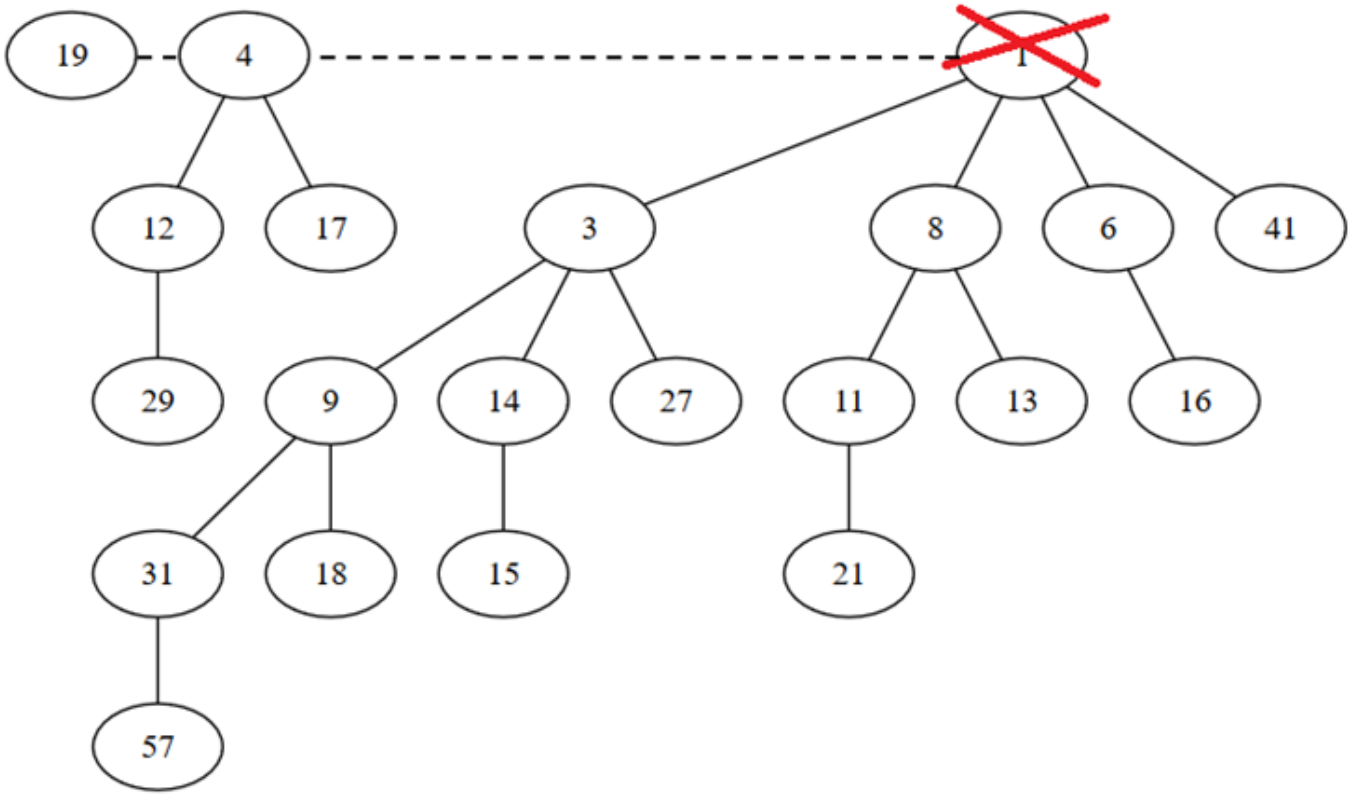
- Most of the other operations that we have for the binomial heap (because we need them for the priority queue) will use the merge operation presented above.
- *Push operation:* Inserting a new element means creating a binomial heap with just that element and merging it with the existing one. Complexity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

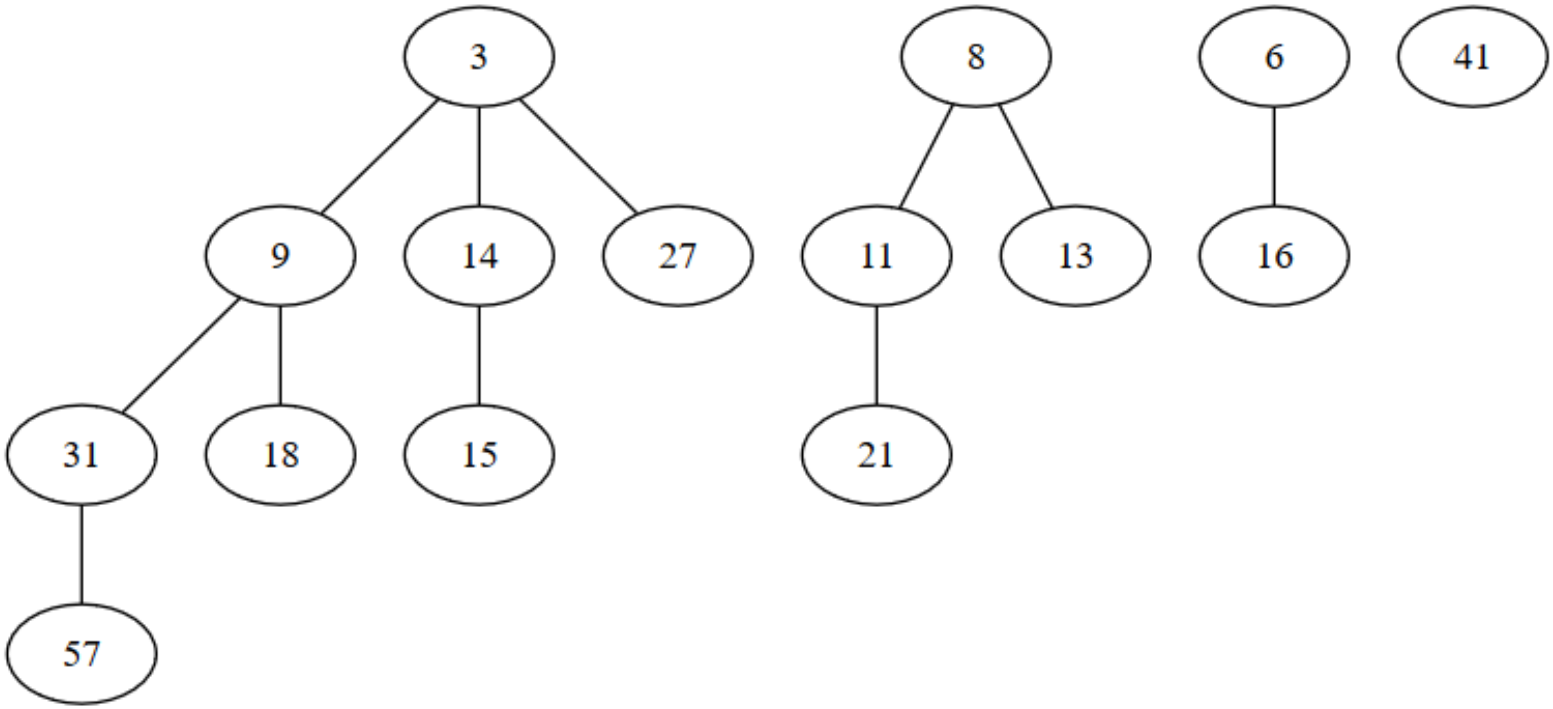
# Binomial heap - other operations III

- The minimum is one of the roots.



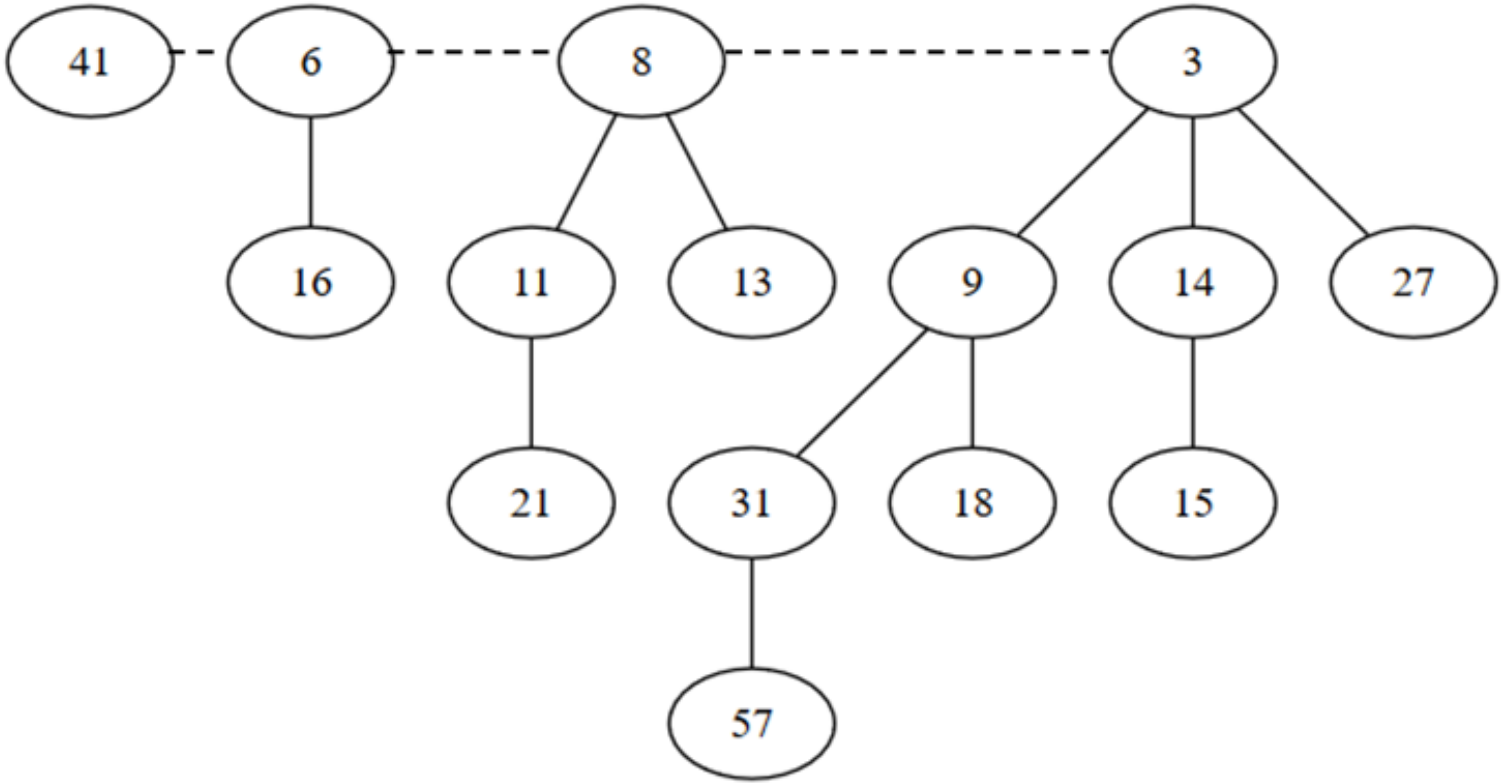
# Binomial heap - other operations IV

- Break the corresponding tree into $k$ binomial trees



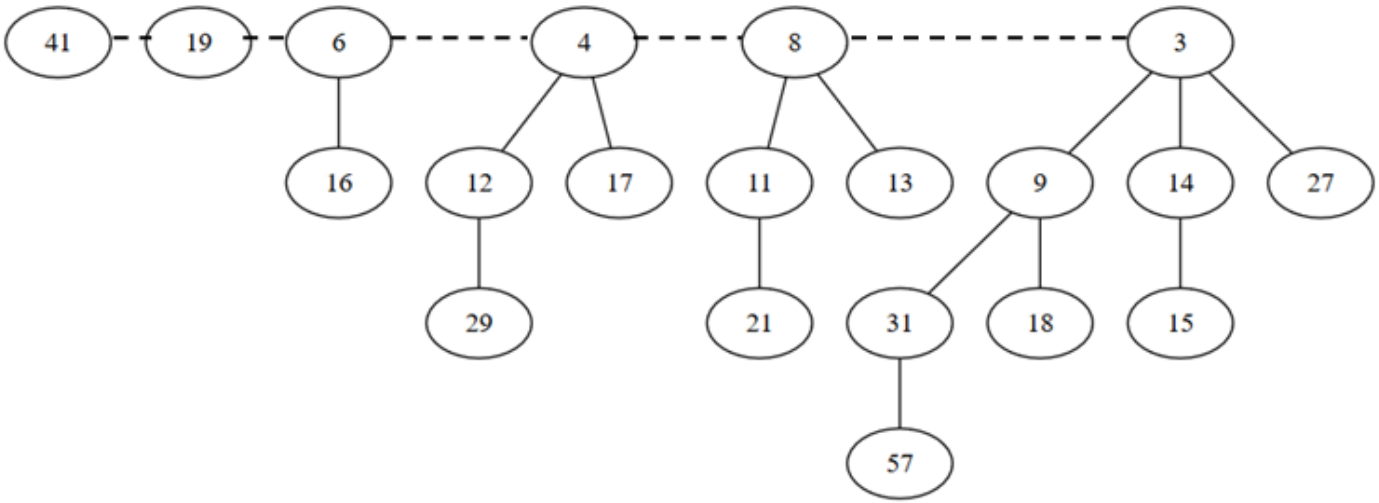
# Binomial heap - other operations V

- Create a binomial heap of these trees



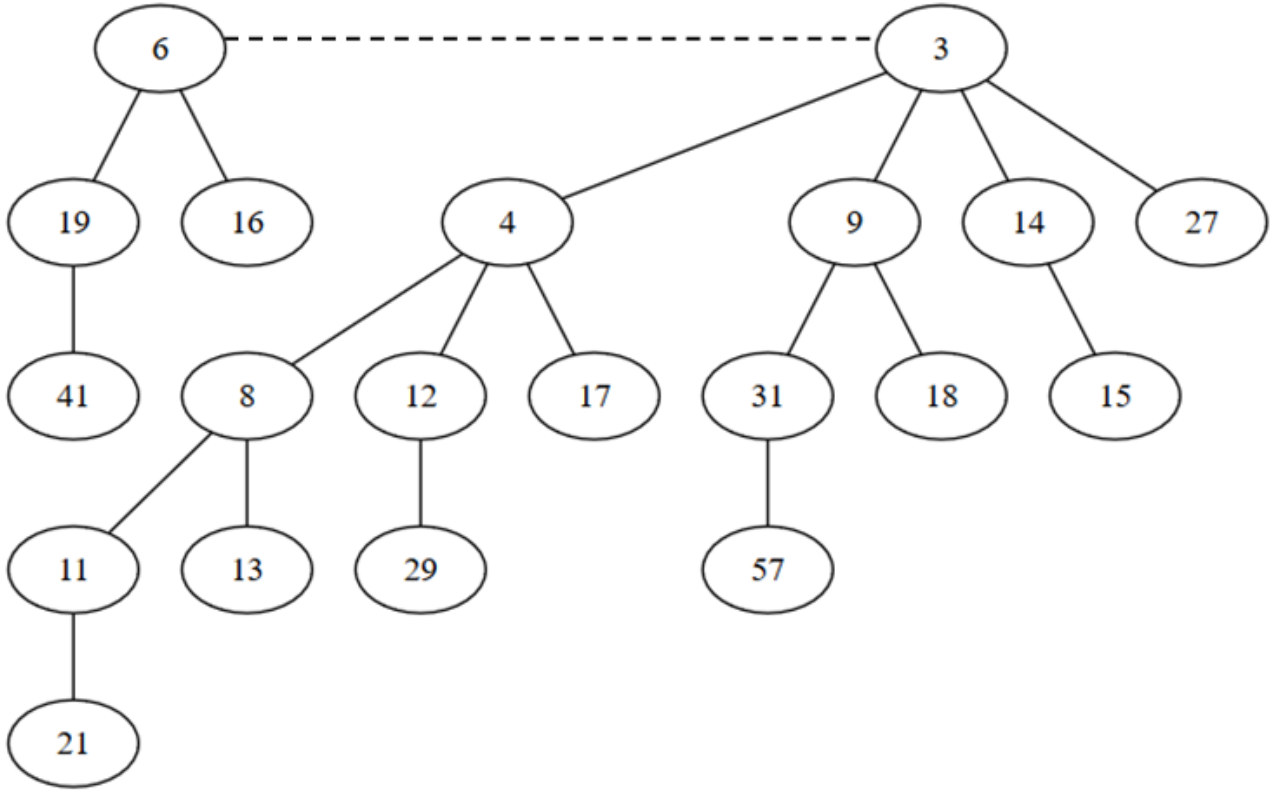
# Binomial heap - other operations VI

- 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

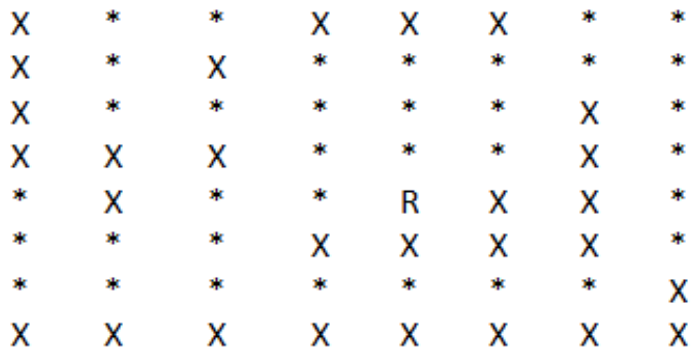
- Assuming that we have a pointer to the element whose priority has to be increased (in our figures lower number means higher priority), we can just change the priority and bubble-up the node if its priority is greater than the priority of its parent. Complexity of the operation is: $O(log_2 n)$
- Assuming 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: $O(log_2 n)$

# Think about it - Problems with stacks, queues and priority queues I

- Red-Black Card Game:
  - Statement: Two players each receive $\frac{n}{2}$ cards, where each card can be red or black. The two players take turns; at every turn the current player puts the card from the upper part of his/her deck on the table. If a player puts a red card on the table, the other player has to take all cards from the table and place them at the bottom of his/her deck. The winner is the player that has all the cards.
  - Requirement: Given the number $n$ of cards, simulate the game and determine the winner.
  - Hint: use stack(s) and queue(s)

# Think about it - Problems with stacks, queues and priority queues II

- Robot in a maze:
  - Statement: There is a rectangular maze, composed of occupied cells (X) and free cells (*). There is a robot (R) in this maze and it can move in 4 directions: N, S, E, V.
  - Requirements:
    - Check whether the robot can get out of the maze (get to the first or last line or the first or last column).
    - Find a path that will take the robot out of the maze (if exists).



# Think about it - Problems with stacks, queues and priority queues III

- Hint:
  - Let $T$ be the set of positions where the robot can get from the starting position.
  - Let $s$ be the set of positions to which the robot can get at a given moment and from which it could continue going to other positions.
  - A possible way of determining the sets T and S could be the following:

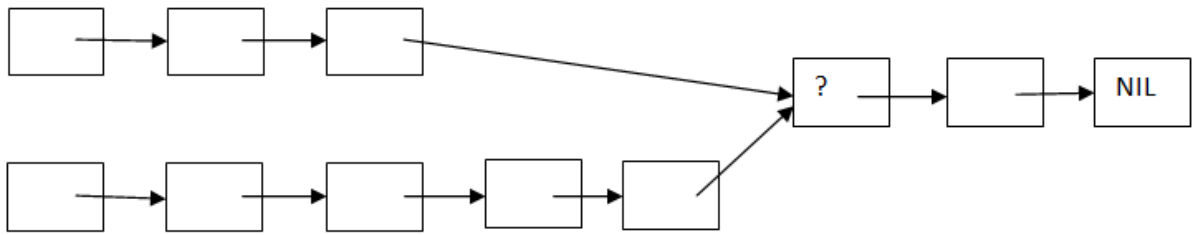
# Think about it - Problems with stacks, queues and priority queues IV

T $\leftarrow$ {initial position}
S $\leftarrow$ {initial position}
**while** S $\neq \emptyset$ **execute**
    **Let** $p$ be one element of S
    S $\leftarrow$ S $\setminus\{p\}$
    **for** each valid position $q$ where we can get from p and which is not in $T$ **do**
        T $\leftarrow$ T $\cup$ {q}
        S $\leftarrow$ S $\cup$ {q}
    **end-for**
**end-while**

- T can be a list, a vector or a matrix associated to the maze
- S can be a stack or a queue (or even a priority queue, depending on what we want)

# 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?

# Example

- Assume that you were asked to write an application for Cluj-Napoca's public transportation service.
- In your application the user can select a bus line and the application should display the timetable for that bus-line (and maybe later the application can be extended with other functionalities).
- Your application should be able to return the info for a bus line and we also want to be able to add and remove bus lines (this is going to be done only by the administrators, obviously).
- And since your application is going to be used by several hundred thousand people, we need it to be very very fast.
  - The public transportation service is willing to maybe rename a few bus lines, if this helps you design a fast application.
- How/Where would you store the data?

# Direct-address tables I

- If we want to formalize the problem:
  - We have data where every element has a key (a natural number).
  - The universe of keys (the possible values for the keys) is relatively small, $U = \{0, 1, 2, \ldots, m-1\}$
  - No two elements have the same key
  - We have to support the basic dictionary operations: INSERT, DELETE and SEARCH

# Direct-address tables II

- Solution:
  - Use an array $T$ with $m$ positions (remember, the keys belong to the $[0, m-1]$ interval)
  - Data about element with key $k$, will be stored in the $T[k]$ slot
  - Slots not corresponding to existing elements will contain the value NIL (or some other special value to show that they are empty)

# Operations for a direct-address table

**function** search(T, k) **is**:
*//pre: T is an array (the direct-address table), k is a key*
  search $\leftarrow$ T[k]
**end-function**

# Operations for a direct-address table

**function** search(T, k) **is**:
*//pre: T is an array (the direct-address table), k is a key*
  search $\leftarrow$ T[k]
**end-function**

**subalgorithm** insert(T, x) **is**:
*//pre: T is an array (the direct-address table), x is an element*
  T[key(x)] $\leftarrow$ x *//key(x) returns the key of an element*
**end-subalgorithm**

# Operations for a direct-address table

**function** search(T, k) **is**:
*//pre: T is an array (the direct-address table), k is a key*
search $\leftarrow$ T[k]
**end-function**

**subalgorithm** insert(T, x) **is**:
*//pre: T is an array (the direct-address table), x is an element*
T[key(x)] $\leftarrow$ x *//key(x) returns the key of an element*
**end-subalgorithm**

**subalgorithm** delete(T, x) **is**:
*//pre: T is an array (the direct-address table), x is an element*
T[key(x)] $\leftarrow$ NIL
**end-subalgorithm**

# Direct-address table - Advantages and disadvantages

- Advantages of direct address-tables:
  - They are simple
  - They are efficient - all operations run in $\Theta(1)$ time.
- Disadvantages of direct address-tables - restrictions:
  - The keys have to be natural numbers
  - The keys have to come from a small universe (interval)
  - The number of actual keys can be a lot less than the cardinal of the universe (storage space is wasted)

# Think about it

- Assume that we have a direct address $T$ of length $m$. How can we find the maximum element of the direct-address table? What is the complexity of the operation?

- How does the operation for finding the maximum change if we have a hash table, instead of a direct-address table (consider collision resolution by separate chaining, coalesced chaining and open addressing)?

# Hash tables

- Hash tables are generalizations of direct-address tables and they represent a *time-space trade-off*.
- Searching for an element still takes $\Theta(1)$ time, but as *average case complexity* (worst case complexity is higher)

# Hash tables - main idea I

- We will still have a table $T$ of size $m$ (but now $m$ is not the number of possible keys, $|U|$) - *hash table*
- Use a function $h$ that will map a key $k$ to a slot in the table $T$ - *hash function*

$$h : U \to \{0, 1, ..., m-1\}$$

- Remarks:
  - In case of direct-address tables, an element with key $k$ is stored in $T[k]$.
  - In case of hash tables, an element with key $k$ is stored in $T[h(k)]$.

# Hash tables - main idea II

- The point of the hash function is to reduce the range of array indexes that need to be handled $=>$ instead of $|U|$ values, we only need to handle $m$ values.
- Consequence:
  - two keys may hash to the same slot $=>$ **a collision**
  - we need techniques for resolving the conflict created by collisions
- We will talk about different ways of defining a hash function in the next lecture, currently we will use one of the simplest versions: *the division method*

# The division method

## The division method

$h(k) = k \mod m$

## For example:

$$m = 13$$
$$k = 63 => h(k) = 11$$
$$k = 52 => h(k) = 0$$
$$k = 131 => h(k) = 1$$

- Requires only a division so it is quite fast
- Experiments show that good values for $m$ are primes not too close to exact powers of 2

# Collisions

- When two keys, $x$ and $y$, have the same value for the hash function $h(x) = h(y)$ we have a *collision*.
- A good hash function can reduce the number of collisions, but it cannot eliminate them at all:
  - Try fitting $m + 1$ keys into a table of size $m$
- There are different collision resolution methods:
  - Separate chaining
  - Coalesced chaining
  - Open addressing

# The birthday paradox

- *How many randomly chosen people are needed in a room, to have a good probability - about 50% - of having two people with the same birthday?*
- It is obvious that if we have 367 people, there will be at least two with the same birthday (there are only 366 possibilities).

# The birthday paradox

- *How many randomly chosen people are needed in a room, to have a good probability - about 50% - of having two people with the same birthday?*
- It is obvious that if we have 367 people, there will be at least two with the same birthday (there are only 366 possibilities).
- What might not be obvious, is that approximately 70 people are needed for a 99.9% probability
- 23 people are enough for a 50% probability

# Separate chaining

- Collision resolution by separate chaining: each slot from the hash table $T$ contains a linked list, with the elements that hash to that slot
- Dictionary operations become operations on the corresponding linked list:
  - $insert(T, x)$ - insert a new node to the beginning of the list $T[h(key[x])]$
  - $search(T, k)$ - search for an element with key $k$ in the list $T[h(k)]$
  - $delete(T, x)$ - delete $x$ from the list $T[h(key[x])]$

# Hash table with separate chaining - representation

- A hash table with separate chaining would be represented in the following way (for simplicity, we will keep only the keys in the nodes).

Node:
key: TKey
next: ↑ Node

HashTable:
T: ↑Node[] *//an array of pointers to nodes*
m: Integer
h: TFunction *//the hash function*

# Hash table with separate chaining - search

```
function search(ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: function returns True if k is in ht, False otherwise
    position ← ht.h(k)
    currentN ← ht.T[position]
    while currentN ≠ NIL and [currentN].key ≠ k execute
        currentN ← [currentN].next
    end-while
    if currentN ≠ NIL then
        search ← True
    else
        search ← False
    end-if
end-function
```

- Usually search returns the info associated with the key $k$

# Analysis of hashing with chaining

- The average performance depends on how well the hash function $h$ can distribute the keys to be stored among the $m$ slots.
- **Simple Uniform Hashing** assumption: each element is equally likely to hash into any of the $m$ slots, independently of where any other elements have hashed to.
- **load factor** $\alpha$ of the table $T$ with $m$ slots containing $n$ elements
  - is $n/m$
  - represents the average number of elements stored in a chain
  - in case of separate chaining can be less than, equal to, or greater than 1.

# Analysis of hashing with chaining - Insert

- The slot where the element is to be added can be:
  - empty - create a new node and add it to the slot
  - occupied - create a new node and add it to the beginning of the list
- In either case worst-case time complexity is: $\Theta(1)$
- If we have to check whether the element already exists in the table, the complexity of searching is added as well.

# Analysis of hashing with chaining - Search I

- There are two cases
  - unsuccessful search
  - successful search
- We assume that
  - the hash value can be computed in constant time ($\Theta(1)$)
  - the time required to search an element with key $k$ depends linearly on the length of the list $T[h(k)]$

# Analysis of hashing with chaining - Search II

- **Theorem:** In a hash table in which collisions are resolved by separate chaining, an unsuccessful search takes time $\Theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.
- **Theorem:** In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.
- Proof idea: $\Theta(1)$ is needed to compute the value of the hash function and $\alpha$ is the average time needed to search one of the $m$ lists

# Analysis of hashing with chaining - Search III

- If $n = O(m)$ (the number of hash table slots is proportional to the number of elements in the table, if the number of elements grows, the size of the table will grow as well)
  - $\alpha = n/m = O(m)/m = \Theta(1)$
  - searching takes constant time on average
- Worst-case time complexity is $\Theta(n)$
  - When all the nodes are in a single linked-list and we are searching this list
  - In practice hash tables are pretty fast

# Analysis of hashing with chaining - Delete

- If the lists are doubly-linked and we know the address of the node: $\Theta(1)$
- If the lists are singly-linked: proportional to the length of the list

- **All dictionary operations can be supported in $\Theta(1)$ time on average.**
- In theory we can keep any number of elements in a hash table with separate chaining, but the complexity is proportional to $\alpha$. If $\alpha$ is too large $\Rightarrow$ resize and rehash.

# Example 2

- Assume we have a hash table with m = 6 that uses separate chaining for collision resolution, with the following policy: if the load factor of the table after an insertion is greater than or equal to 0.7, we double the size of the table

- Using the division method, insert the following elements, in the given order, in the hash table: 38, 11, 8, 72, 57, 29, 2.