DATA STRUCTURES AND ALGORITHMS

LECTURE 8

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2024 - 2025

- Doubly linked list on array
- Stacks, Queues and Priority Queues
- Binary heap

Today

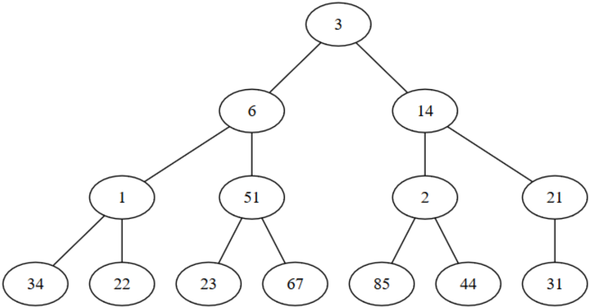
- Binary heap
- Binomial heap

- 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	$\Theta(1)$	$\Theta(n)$

Recap: Binary Heap

1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	6	14	1	51	2	21	34	22	23	67	85	44	31



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).

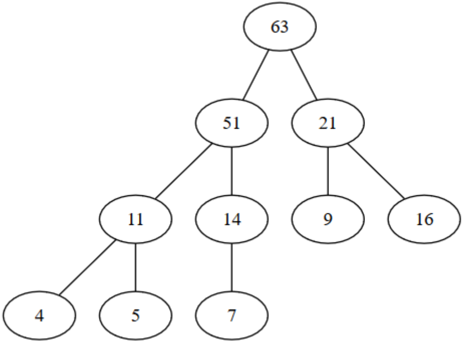
Binary Heap - representation

<u>Heap:</u>
cap: Integer
len: Integer
elems: TElem[]

- Depending on the problem, you might need to have a *relation* as well, as part of the heap.
- For the implementation we will assume that we have a MAX-HEAP.

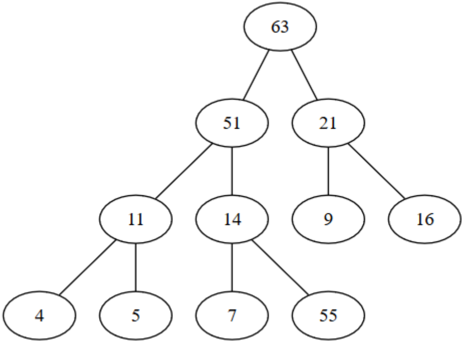
Binary Heap - Add - example

- Consider the following (MAX) heap:



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

- 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).

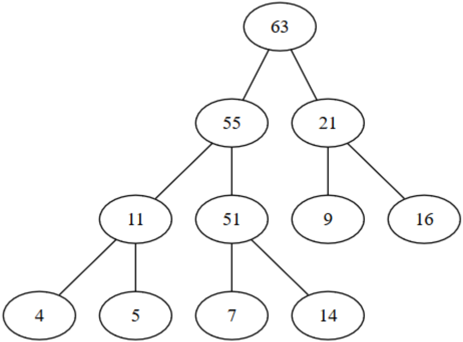


Binary Heap - Add - example

- *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:

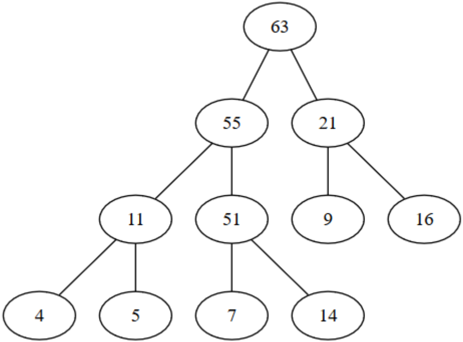
Binary Heap - add

```
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)?

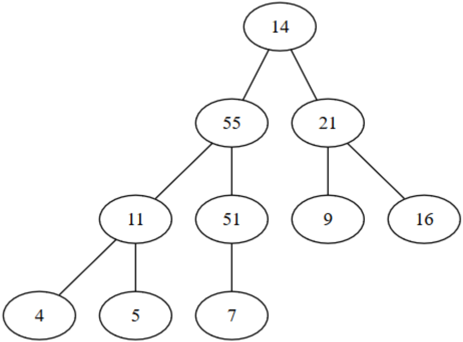
Binary Heap - Remove - example

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



Binary Heap - Remove - example

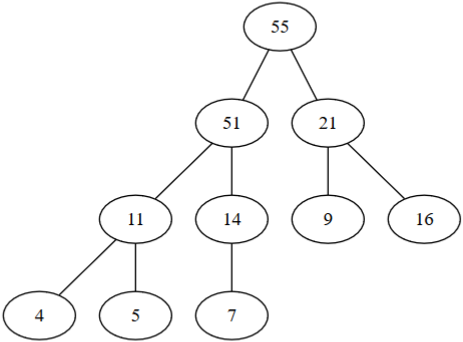
- In order to keep the *heap structure*, when we remove the root, we are going to move the last element from the array to be the root.



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.

- 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...
```

```
    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:

```
        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.)

- 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

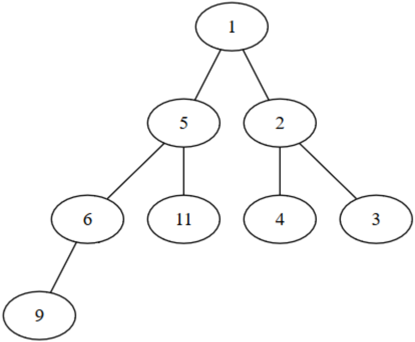
- Based on what we know so far, we can guess how heap-sort works:

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

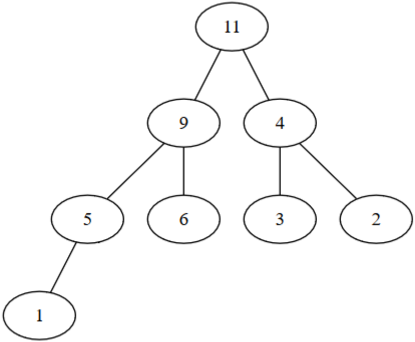
- What is the time complexity of the heap-sort algorithm described above?
- The time complexity of the algorithm is $O(nlog_2n)$
- What is the extra space complexity of the heap-sort algorithm described above (do we need an extra array)?

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.
  - Time complexity of this approach: $O(n)$ (but removing the elements from the heap is still $O(nlog_2 n)$)

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.

- 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	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

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 the priority of an existing element
  - delete an arbitrary element
  - merge two priority queues

Priority Queue - Extension

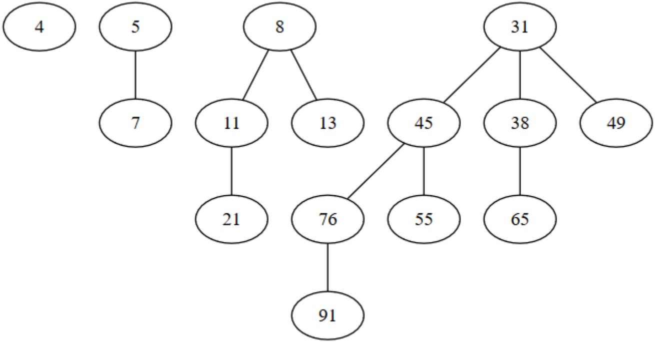
- What is the complexity of these three extra operations if we use as representation a binary heap?
  - Increasing the priority of an existing element is $O(log_2 n)$ if we know the position where the element is.
  - Deleting an arbitrary element is $O(log_2 n)$ if we know the position where the element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

- 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

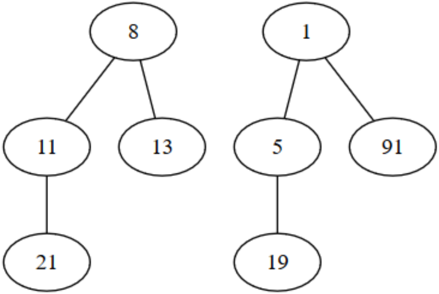


Binomial trees of order 0, 1, 2 and 3

Binomial tree

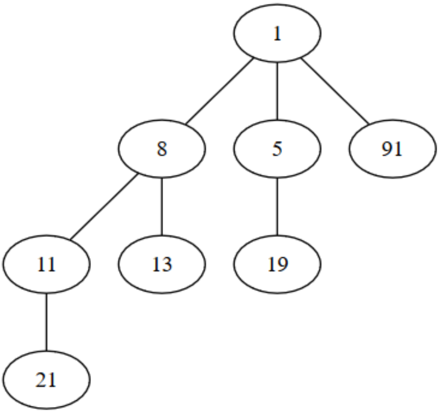
- A binomial tree of order k has exactly 2^k nodes.
- The height of a binomial tree of order k is k.
- If we delete the root of a binomial tree of order k, we will get k binomial trees, of orders $k-1$, $k-2$, ..., 2, 1, 0.
- Two binomial trees of the same order k can be merged into a binomial tree of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

Binomial tree - Merge I



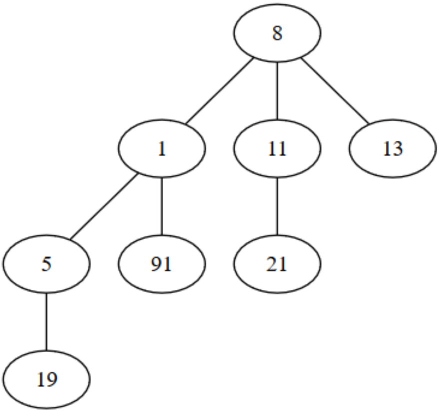
Before merge we have two binomial trees of order 2

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

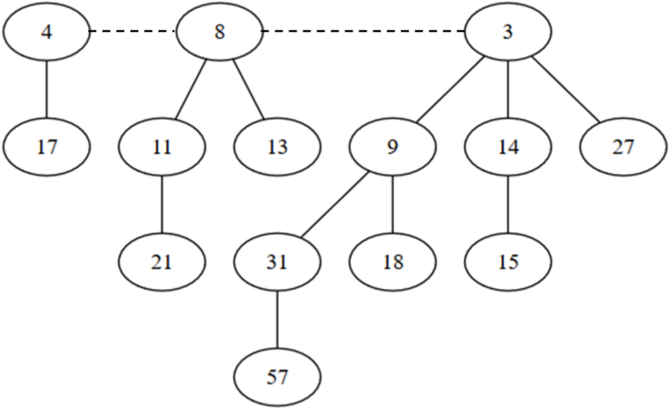
Binomial tree - representation

- 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 (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 $log_2 n$.

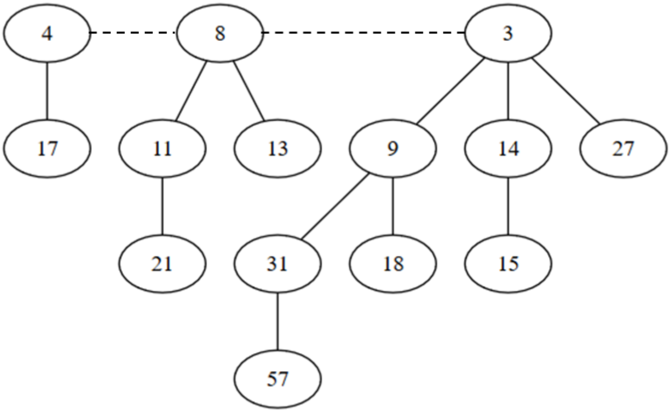
Binomial heap - merge

- 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.

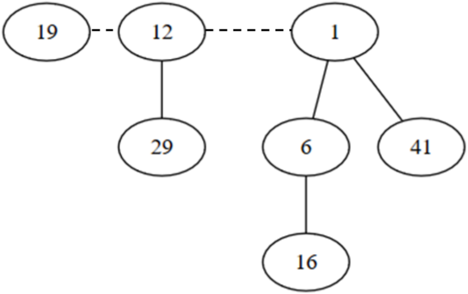
- Since both binomial heaps are sorted linked lists, the first step is to *merge* the two linked lists (standard merge algorithm for two sorted linked lists).

- 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.

- Let's merge the following two binomial heaps:

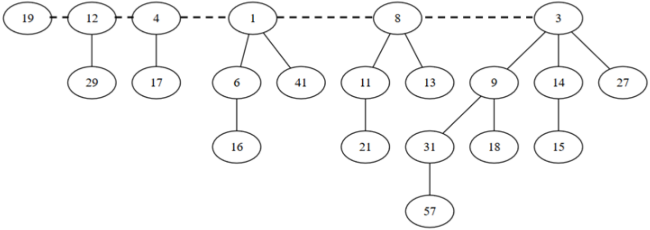


Binomial heap - merge - example II



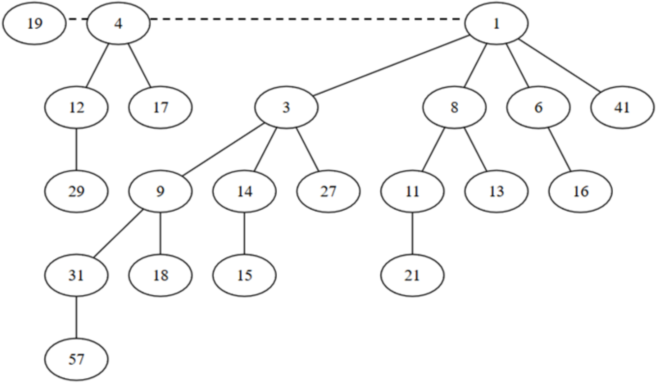
Binomial heap - merge - example III

- 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 $log_2 n$).

Summary

- Today we have talked about:
  - Binary heap
  - Binomial heap
- Extra reading - a nice problem statement, for which the solution will be given in next week's extra reading.