DATA STRUCTURES AND ALGORITHMS
LECTURE 8

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 7...

- Circular List
- XOR List
- Skip List
- Linked Lists on Arrays

Today

- Iterator
- Stack, Queue, Deque
- Priority Queue
- Binary Heap

Iterator - why do we need it? I

- Most containers have iterators and for every data structure we will discuss how we can implement an iterator for a container defined on that data structure.

- Why are iterators so important?

- They offer a uniform way of iterating through the elements of any container

```
subalgorithm printContainer(c) is:
//pre: c is a container
//post: the elements of c were printed
//we create an iterator using the iterator method of the container
   iterator(c, it)
   while valid(it) execute
      //get the current element from the iterator
      elem ← getCurrent(it)
      print elem
      //go to the next element
      next(it)
   end-while
end-subalgorithm
```

Iterator - why do we need it? III

- For most containers the iterator is the only thing we have that lets us *see* the content of the container.
  - ADT List is the only container that has positions, for other containers we can use only the iterator.

Iterator - why do we need it? IV

- Giving up positions, we can gain performance.
  - Containers that do not have positions can be represented on data structures where some operations have good complexities, but where the notion of a position does not naturally exist and where enforcing positions is really complicated.

Iterator - why do we need it? V

- Even if we have positions, using an iterator might be faster.
  - Going through the elements of a linked list with an iterator is faster than going through every position one-by-one.

ADT Stack - Recap

- The ADT *Stack* represents a container in which access to the elements is restricted to one end of the container, called the *top* of the stack.
  - When a new element is added, it will automatically be added to the top.
  - When an element is removed, the one from the top is automatically removed.
  - Only the element from the top can be accessed.

- Because of this restricted access, the stack is said to have a **LIFO** policy: **L**ast **I**n, **F**irst **O**ut (the last element that was added will be the first element that will be removed).

Representation for Stack

- Data structures that can be used to implement a stack:
  - Arrays
    - Static Array - if we want a fixed-capacity stack
    - Dynamic Array
  - Linked Lists
    - Singly-Linked List
    - Doubly-Linked List

Array-based representation

- Where should we place the top of the stack for optimal performance?

Array-based representation

- Where should we place the top of the stack for optimal performance?
- We have two options:
  - Place top at the beginning of the array - every push and pop operation needs to shift every element to the right or left.
  - Place top at the end of the array - push and pop elements without moving the other ones.
- Conclusion: put it at the end of the array

Stack - Representation on SLL

- Where should we place the top of the stack for optimal performance?

Stack - Representation on SLL

- Where should we place the top of the stack for optimal performance?
- We have two options:
  - Place it at the end of the list (like we did when we used an array) - for every push, pop and top operation we have to iterate through every element to get to the end of the list.
  - Place it at the beginning of the list - we can push and pop elements without iterating through the list.
- Conclusion: put it at the beginning of the SLL

- Where should we place the top of the stack for optimal performance?

Stack - Representation on DLL

- Where should we place the top of the stack for optimal performance?
- We have two options:
  - Place it at the end of the list (like we did when we used an array) - we can push and pop elements without iterating through the list.
  - Place it at the beginning of the list - we can push and pop elements without iterating through the list.
- Conclusion: you can put it at either end of the DLL

Fixed capacity stack with linked list

- How could we implement a stack with a fixed maximum capacity using a linked list?

Fixed capacity stack with linked list

- How could we implement a stack with a fixed maximum capacity using a linked list?
- Similar to the implementation with a static array, we can keep in the *Stack* structure two integer values (besides the top node): maximum capacity and current size.

GetMinimum in constant time

- How can we design a *special stack* that has a *getMinimum* operation with $\Theta(1)$ time complexity (and the other operations have $\Theta(1)$ time complexity as well)?

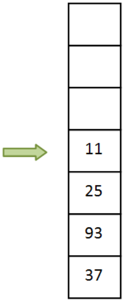
GetMinimum in constant time

- How can we design a *special stack* that has a *getMinimum* operation with $\Theta(1)$ time complexity (and the other operations have $\Theta(1)$ time complexity as well)?

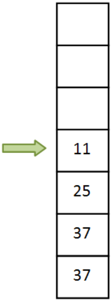
- We can keep an auxiliary stack, containing as many elements as the original stack, but containing the minimum value up to each element. Let's call this auxiliary stack a *min stack* and the original stack the *element stack*.

GetMinimum in constant time - Example

- If this is the *element stack*:



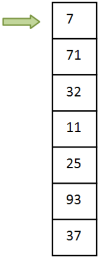
- This is the corresponding *min stack*:



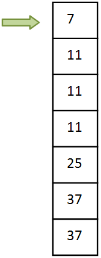
GetMinimum in constant time - Example

- When a new element is pushed to the *element stack*, we push a new element to the *min stack* as well. This element is the minimum between the top of the *min stack* and the newly added element.

- The *element stack*:



- The corresponding *min stack*:



GetMinimum in constant time

- When an element si popped from the *element stack*, we will pop an element from the *min stack* as well.

- The *getMinimum* operation will simply return the *top* of the *min stack*.

- The other stack operations remain unchanged (except *init*, where you have to create two stacks).

- Let's implement the *push* operation for this *SpecialStack*, represented in the following way:

<u>SpecialStack:</u>
  elementStack: Stack
  minStack: Stack

- We will use an existing implementation for the stack and work only with the operations from the interface.

Push for SpecialStack

```
subalgorithm push(ss, e) is:
   if isFull(ss.elementStack) then
      @throw overflow (full stack) exception
   end-if
   if isEmpty(ss.elementStack) then//the stacks are empty, just push the elem
      push(ss.elementStack, e)
      push(ss.minStack, e)
   else
      push(ss.elementStack, e)
      currentMin ← top(ss.minStack)
      if currentMin < e then //find the minim to push to minStack
         push(ss.minStack, currentMin)
      else
         push(ss.minStack, e)
      end-if
   end-if
end-subalgorithm //Complexity: Θ(1)
```

- We designed the special stack in such a way that all the operations have a $\Theta(1)$ time complexity.
- The disadvantage is that we occupy twice as much space as with the regular stack.

- Think about how can we reduce the space occupied by the *min stack* to O(n) (especially if the minimum element of the stack rarely changes). *Hint: If the minimum does not change, we don't have to push a new element to the min stack.* How can we implement the *push* and *pop* operations in this case? What happens if the minimum element appears more than once in the *element stack*?

ADT Queue - Recap

- The ADT Queue represents a container in which access to the elements is restricted to the two ends of the container, called *front* and *rear*.
  - When a new element is added (pushed), it has to be added to the *rear* of the queue.
  - When an element is removed (popped), it will be the one at the *front* of the queue.
- Because of this restricted access, the queue is said to have a **FIFO** policy: First In First Out.

Queue - Representation

- What data structures can be used to implement a Queue?
    - Dynamic Array - circular array (already discussed)
    - Singly Linked List
    - Doubly Linked List

- If we want to implement a Queue using a singly linked list, where should we place the *front* and the *rear* of the queue?

Queue - representation on a SLL

- If we want to implement a Queue using a singly linked list, where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the list and *rear* at the end
  - Put *front* at the end of the list and *rear* at the beginning
- In either case we will have one operation (push or pop) that will have $\Theta(n)$ complexity.

Queue - representation on a SLL

- We can improve the complexity of the operations if, even though the list is singly linked, we keep both the head and the tail of the list.
- What should the tail of the list be: the *front* or the *rear* of the queue?

- If we want to implement a Queue using a doubly linked list, where should we place the *front* and the *rear* of the queue?

Queue - representation on a DLL

- If we want to implement a Queue using a doubly linked list, where should we place the *front* and the *rear* of the queue?

- In theory, we have two options:
  - Put *front* at the beginning of the list and *rear* at the end
  - Put *front* at the end of the list and *rear* at the beginning

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.

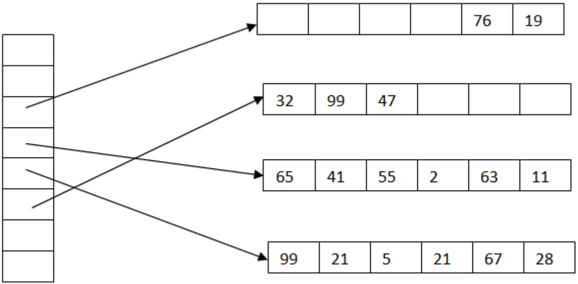
ADT Deque

- Possible (good) 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 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 - Recap

- 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.

Priority Queue - Representation

- What data structures can be used to implement a priority queue?
  - Dynamic Array
  - Linked List
  - (Binary) Heap - will be discussed later

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
    - Where would you put the element with the highest priority?
  - we can keep the elements in the order in which they were inserted

- 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?

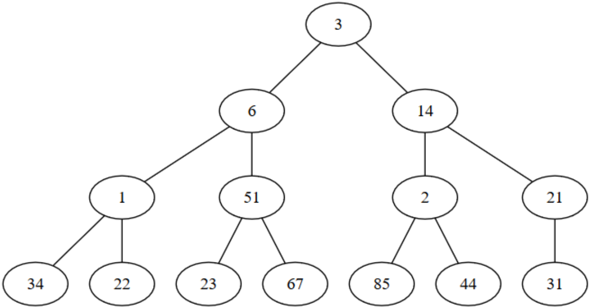
Binary Heap

- A binary heap is a data structure that can be used as an efficient representation for Priority Queues.
- A binary heap is a kind of hybrid between a dynamic array and a binary tree.
- The elements of the heap are actually stored in the dynamic array, but the array is visualized as a binary tree.

- Assume that we have the following array (upper row contains positions, lower row contains elements):

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

- We can visualize this array as a binary tree, where the root is the first element of the array, its children are the next two elements, and so on. Each node has exactly 2 children, except for the last two rows, but there the children of the nodes are completed from left to right.



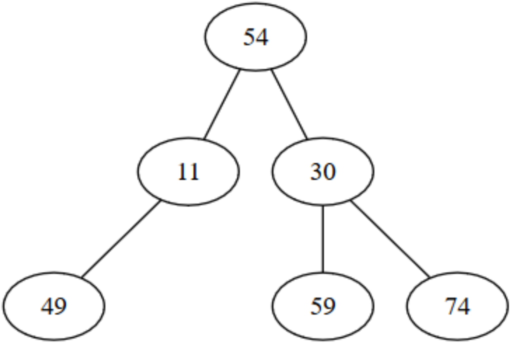
Binary Heap

- If the elements of the array are: $a_1, a_2, a_3, ..., a_n$, we know that:
  - a_1 is the root of the heap
  - for an element from position i, its children are on positions $2 * i$ and $2 * i + 1$ (if $2 * i$ and $2 * i + 1$ is less than or equal to n)
  - for an element from position i (i > 1), the parent of the element is on position $[i/2]$ (integer part of i/2)

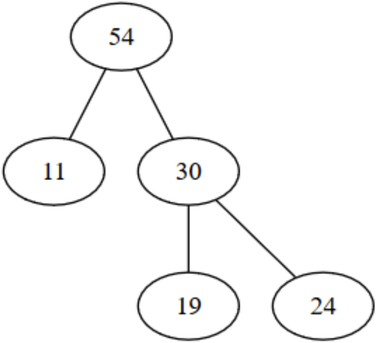
Binary Heap

- A *binary heap* is an array that can be visualized as a binary tree having a *heap structure* and a *heap property*.
  - *Heap structure:* in the binary tree every node has exactly 2 children, except for the last two levels, where children are completed from left to right.
  - *Heap property:* $a_i \geq a_{2*i}$ (if $2*i \leq n$) and $a_i \geq a_{2*i+1}$ (if $2*i+1 \leq n$)
  - The $\geq$ relation between a node and both its descendants can be generalized (other relations can be used as well).

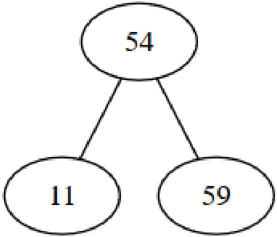
- Are the following binary trees heaps? If yes, specify the relation between a node and its children. If not, specify if the problem is with the structure, the property, or both.



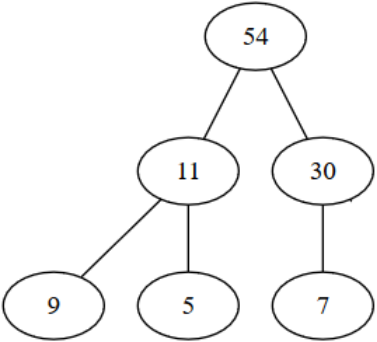
Binary Heap - Examples II



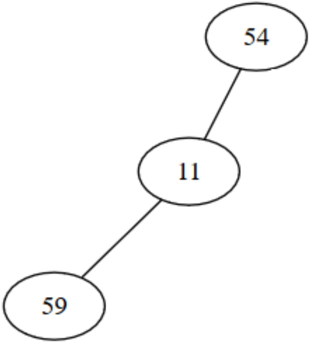
Binary Heap - Examples III



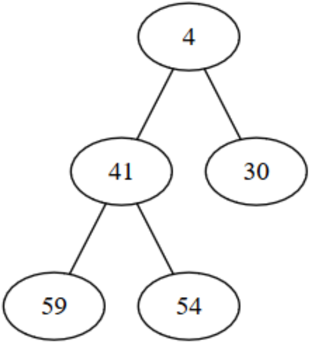
Binary Heap - Examples IV



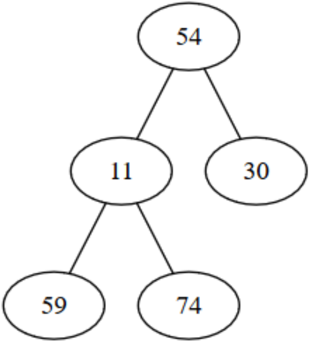
Binary Heap - Examples V



Binary Heap - Examples VI



Binary Heap - Examples VII



Binary Heap - Examples VIII

- Are the following arrays valid heaps? If not, transform them into a valid heap by swapping two elements.
    1. [70, 10, 50, 7, 1, 33, 3, 8]
    2. [1, 2, 4, 8, 16, 32, 64, 65, 10]
    3. [10, 12, 100, 60, 13, 102, 101, 80, 90, 14, 15, 16]

- If we use the $\geq$ relation, we will have a *MAX-HEAP*. Do you know why?

- If we use the $\geq$ relation, we will have a *MAX-HEAP*. Do you know why?
- If we use the $\leq$ relation, we will have a *MIN-HEAP*. Do you know why?

Binary Heap - Notes

- If we use the $\geq$ relation, we will have a *MAX-HEAP*. Do you know why?
- If we use the $\leq$ relation, we will have a *MIN-HEAP*. Do you know why?
- The height of a heap with n elements is $log_2 n$.

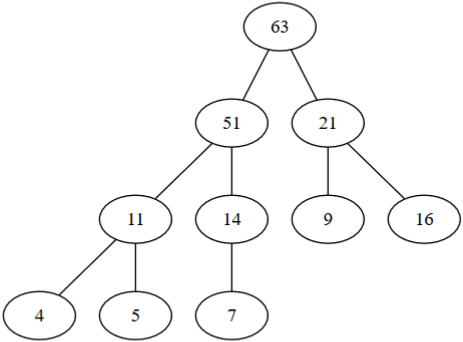
- A heap can be used as representation for a Priority Queue and it has two specific operations:
  - add a new element in the heap (in such a way that we keep both the heap structure and the heap property).
  - remove (we always remove the root of the heap - no other element can be removed).

Binary Heap - representation

Heap:
  cap: Integer
  len: Integer
  elems: TElem[]

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

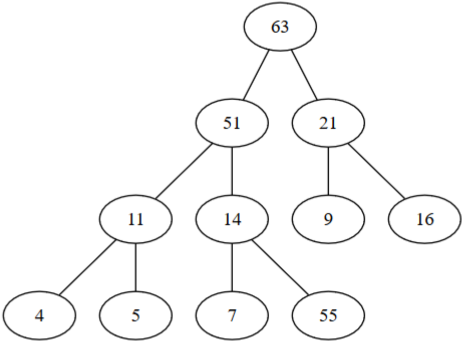
- Consider the following (MAX) heap:



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

Binary Heap - Add - example

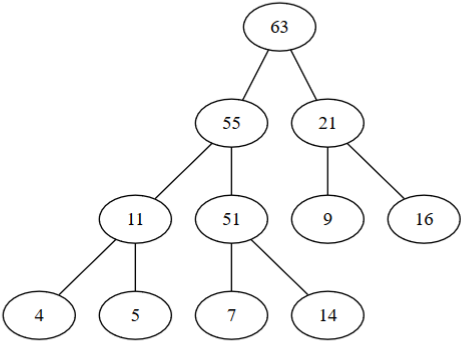
- In order to keep the *heap structure*, we will add the new node as the right child of the node 14 (and as the last element of the array in which the elements are kept).



- *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
```

```
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:

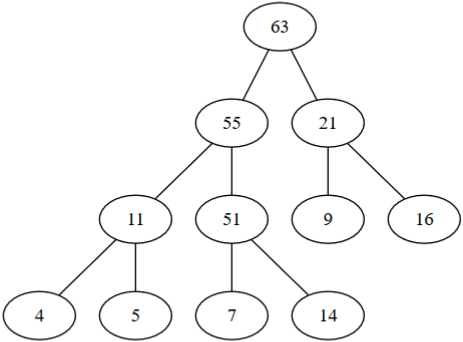
Binary Heap - add

```
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
```

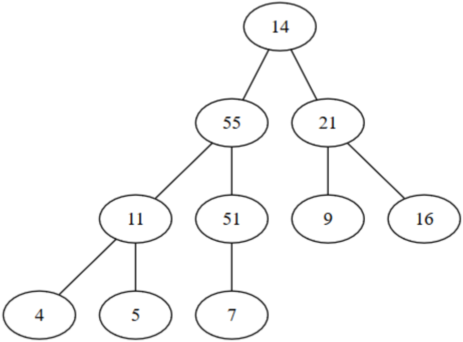
- Complexity: $O(log_2 n)$
- Can you give an example when the complexity of the algorithm is less than $log_2 n$ (best case scenario)?

Binary Heap - Remove - example

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



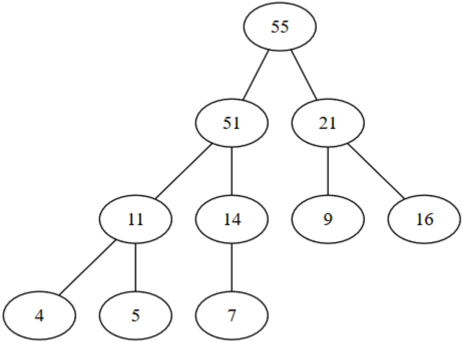
- In order to keep the *heap structure*, when we remove the root, we are going to move the last element from the array to be the root.



- *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...
```

if maxChild $\neq$ -1 **and** heap.elems[maxChild] $>$ elem **then**
    tmp $\leftarrow$ heap.elems[poz]
    heap.elems[poz] $\leftarrow$ heap.elems[maxChild]
    heap.elems[maxChild] $\leftarrow$ tmp
    poz $\leftarrow$ maxChild
else
    poz $\leftarrow$ 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)?

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.)