DATA STRUCTURES

LECTURE 9

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 8...

- Linked Lists on Arrays
- Stack, Queue, Priority Queue

Today

- Binary Heap

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?

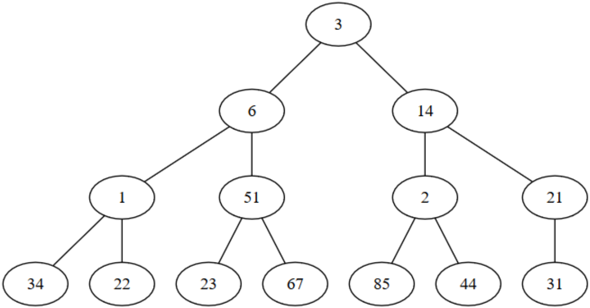
Binary Heap

- 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.

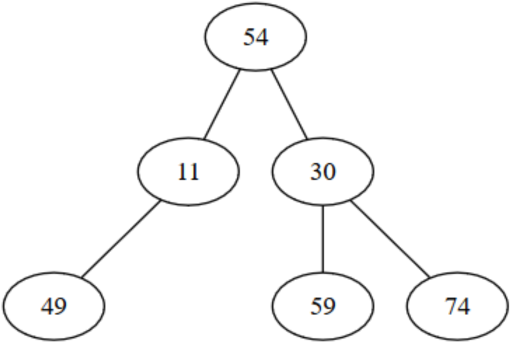


- If the elements of the array are: $a_1, a_2, a_3, ..., a_n$, we know that:
  - a_1 is the root of the heap
  - for an element from position i, its children are on positions $2 * i$ and $2 * i + 1$ (if $2 * i$ and $2 * i + 1$ is less than or equal to n)
  - for an element from position i (i $>$ 1),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

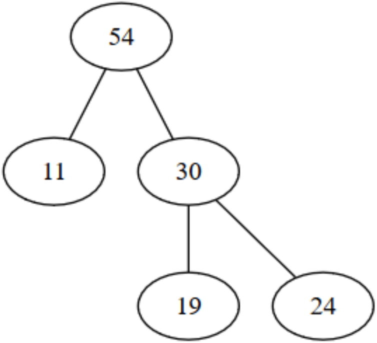
- A *binary heap* is an array that can be visualized as a binary tree having a *heap structure* and a *heap property*.
  - *Heap structure:* in the binary tree every node has exactly 2 children, except for the last two levels, where children are completed from left to right.
  - *Heap property:* $a_i \geq a_{2*i}$ (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).

Binary Heap - Examples I

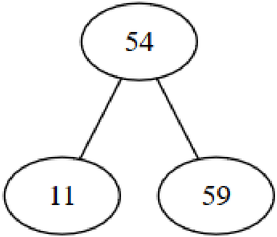
- Are the following binary trees heaps? If yes, specify the relation between a node and its children. If not, specify if the problem is with the structure, the property, or both.



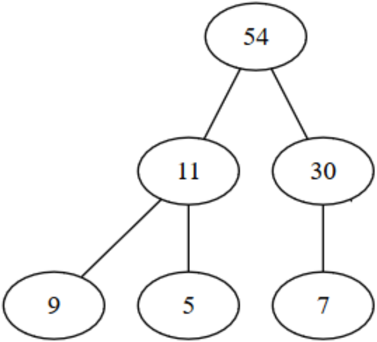
Binary Heap - Examples II



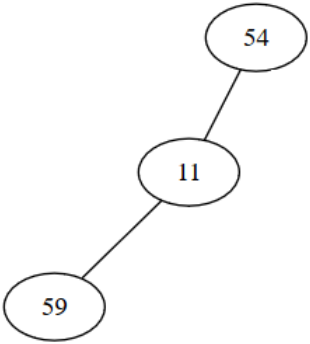
Binary Heap - Examples III



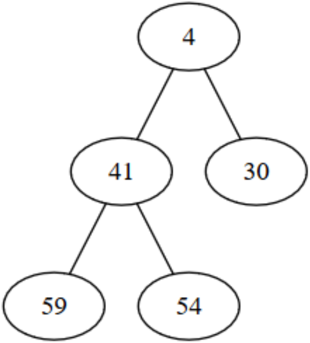
Binary Heap - Examples IV

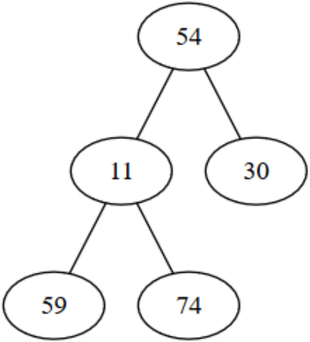


Binary Heap - Examples V



Binary Heap - Examples VI




54
11
30
59
74

- 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?

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?

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 $log_2 n$.

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).

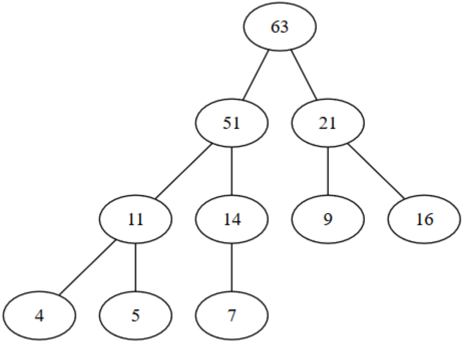
Binary Heap - representation

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

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

Binary Heap - Add - example

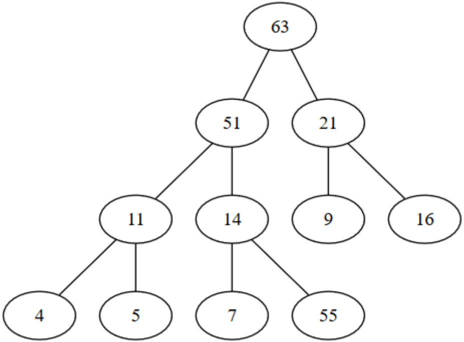
- Consider the following (MAX) heap:



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

Binary Heap - Add - example

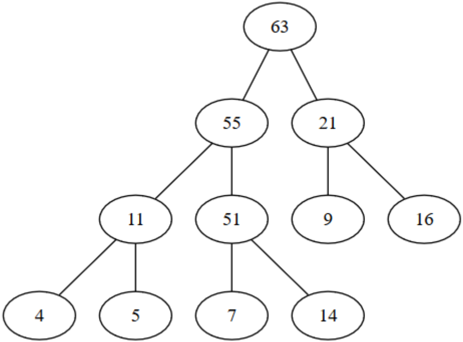
- In order to keep the *heap structure*, we will add the new node as the right child of the node 14 (and as the last element of the array in which the elements are kept).



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.

- 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:

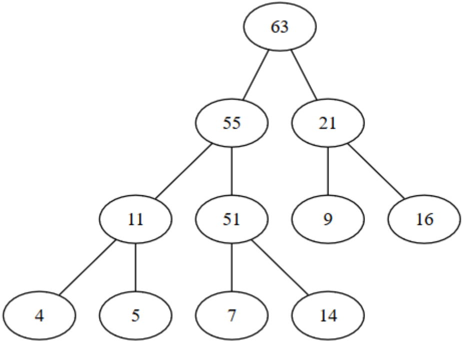
Binary Heap - add

```
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
```

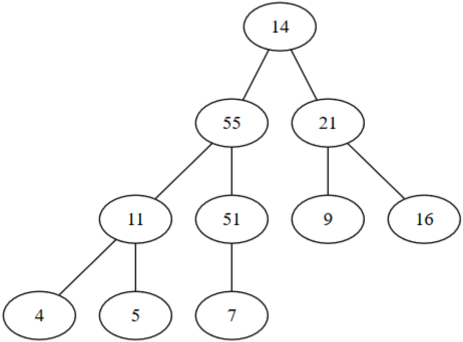
- Complexity: $O(log_2 n)$
- Can you give an example when the complexity of the algorithm is less than $log_2 n$ (best case scenario)?

Binary Heap - Remove - example

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



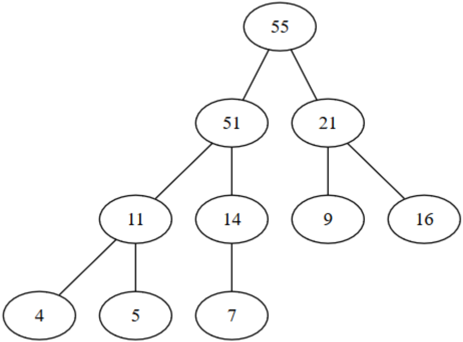
- In order to keep the *heap structure*, when we remove the root, we are going to move the last element from the array to be the root.



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:



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 $\neq$ -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: $O(log_2 n)$
- 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.)

- https://open.kattis.com/problems/chewbacca

Heap-sort

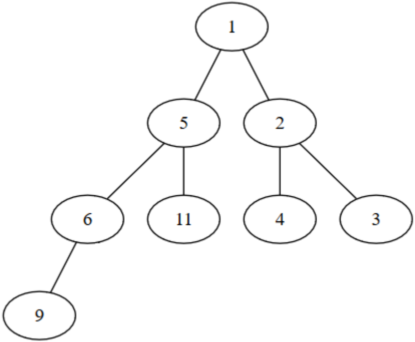
- There is a sorting algorithm, called *Heap-sort*, that is based on the use of a heap.
- In the following we are going to assume that we want to sort a sequence in ascending order.
- Let's sort the following sequence: [6, 1, 3, 9, 11, 4, 2, 5]

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.

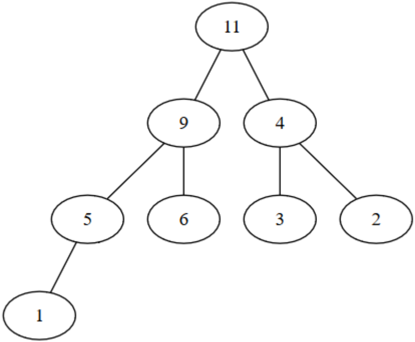
Heap-sort - Naive approach

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

- 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)?

- What is the time complexity of the heap-sort algorithm described above?
- The time complexity of the algorithm is $O(nlog_2 n)$
- 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.

- 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.



- 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 $O(nlog_2n)$)

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

- 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)

- 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).

X	*	*	X	X	X	*	*
X	*	X	*	*	*	*	*
X	*	*	*	*	*	X	*
X	X	X	*	*	*	X	*
*	X	*	*	R	X	X	*
*	*	*	X	X	X	X	*
*	*	*	*	*	*	*	X
X	X	X	X	X	X	X	X

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:

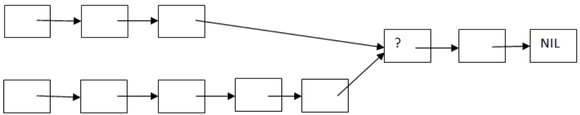
T $\leftarrow$ {initial position}
S $\leftarrow$ {initial position}
while S $\neq \emptyset$ **execute**
    Let p be one element of S
    S $\leftarrow$ S $\setminus\{p\}$
    for each valid position q where we can get from p and which is not in T **do**
        T $\leftarrow$ T $\cup$ {q}
        S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

Problems with stacks, queues and priority queues IV

- 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?