In Lecture 4...

- Iterator
- Binary Heap
Today

1. Binary Heap

2. Linked List
   - Singly Linked Lists
A binary heap is a dynamic array visualized as a binary tree.
The array [56, 9, 20, 4, 8, 15, 19, 3, 1, 6, 5] can be visualized as

How can we add element 66 to the heap?
Heap - representation

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

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

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

Complexity:
Heap - remove

```plaintext
if maxChild ≠ -1 and heap.elems[maxChild] > elem then
  heap.elems[poz] ← heap.elems[maxChild]
  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 implement 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).
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]
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:

```
1
/   \
5     2
/ \   / \n6   11 4 3
 /   /  \
9   11
```

- When we remove the elements one-by-one we will have: 1, 2, 3, 4, 5, 6, 9, 11.
What is the time complexity of the heap-sort algorithm described above?

The time complexity of the algorithm is $O(n \log_2 n)$.

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

The time complexity of the algorithm is $O(n \log_2 n)$

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 \( O(n \log_2 n) \)

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.
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 \textit{bubble-down} for every element.

- Time complexity of this approach: \(O(n)\) (but removing the elements from the heap is still \(O(n \log_2 n)\))
A linked list is a linear data structure, where the order of the elements is determined not by indexes, but by a pointer which is placed in each element.

A linked list is a structure that consists of nodes and each node contains, besides the data (that we store in the linked list), a pointer to the address of the next node (and possibly a pointer to the address of the previous node).

The nodes of a linked list are not necessarily adjacent in the memory, this is why we need to keep the address of the successor in each node.
Linked Lists

- Elements from a linked list are accessed based on the pointers stored in the nodes.

- We can directly access only the first element (and maybe the last one) of the list.
Example of a linked list with 5 nodes:
The linked list from the previous slide is actually a *singly linked list - SLL*.

In a SLL each node from the list contains the data and the address of the next node.

The first node of the list is called *head* of the list and the last node is called *tail* of the list.

The tail of the list contains the special value *NIL* as the address of the next node (which does not exist).

If the head of the SLL is *NIL*, the list is considered empty.
Singly Linked Lists - Representation

- For the representation of a SLL we need two structures: one structure for the node and one for the list itself.

**SLLNode:**

- info: TElem //the actual information
- next: ↑ SLLNode //address of the next node
Singly Linked Lists - Representation

- For the representation of a SLL we need two structures: one structure for the node and one for the list itself.

**SLLNode:**
- info: TElem //the actual information
- next: ↑ SLLNode //address of the next node

**SLL:**
- head: ↑ SLLNode //address of the first node

- Usually, for a SLL, we only memorize the address of the head. However, there might be situations when we memorize the address of the tail as well (if the application requires it).

- If the SLL is empty (it has no elements) then the value of head is NULL.
Possible operations for a singly linked list (any operation that can be implemented on a Dynamic Array can be implemented on a linked list as well):

- search for an element with a given value
- add an element (to the beginning, to the end, to a given position, after a given value)
- delete an element (from the beginning, from the end, from a given position, with a given value)
- get an element from a position

These are possible operations; usually we need only part of them, depending on the container that we implement using a SLL.
function search (sll, elem) is:

//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL

current ← sll.head
while current ≠ NIL and [current].info ≠ elem
execute
    current ← [current].next
end-while

search ← current

end-function

Complexity: \( O(n) \) - we can find the element in the first node, or we may need to verify every node.

What happens if sll is empty?
function search (sll, elem) is:
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
    current ← sll.head
    while current ≠ NIL and [current].info ≠ elem execute
        current ← [current].next
    end-while
    search ← current
end-function

- Complexity:
**SLL - Search**

```plaintext
function search (sll, elem) is:
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
  current ← sll.head
  while current ≠ NIL and [current].info ≠ elem execute
    current ← [current].next
  end-while
  search ← current
end-function
```

- Complexity: $O(n)$ - we can find the element in the first node, or we may need to verify every node.

- What happens if $sll$ is empty?
In the search function we have seen how we can walk through the elements of a linked list:

- we need an auxiliary node (called current), which starts at the head of the list
- at each step, the value of the current node becomes the address of the successor node (through the current ← [current].next instruction)
- we stop when the current node becomes NIL