DATA STRUCTURES AND ALGORITHMS
LECTURE 7

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2023 - 2024

In Lecture 6...

- XOR list
- Skip list
- Singly linked list on array

Today

- Doubly linked list on array
- Iterator
- Stack and Queue
- Priority queue
- Binary heap

SLL on Array - recap

- It is a linked list, but the elements are stored in an array. Each element has a *link*, denoting the next element, but this *link* is not a pointer, it is the position of the next element in the array.

	1	2	3	4	5	6	7	8	9	10
elems		78	11	6	59	19		44		
next	7	6	5	-1	8	4	9	2	10	-1

head = 3

first empty

firstEmpty = 1

SLL on Array - Representation

- The representation of a singly linked list on an array is the following:

SLLA:
elems: TElem[]
next: Integer[]
cap: Integer
head: Integer
firstEmpty: Integer

- Obviously, we can define a doubly linked list as well without pointers, using arrays.
- For the DLLA we will see another way of representing a linked list on arrays:
  - The main idea is the same, we will use array indexes as links between elements
  - We are using the same information, but we are going to structure it differently
  - However, we can make it look more similar to linked lists with dynamic allocation

DLLA - Node

- Linked Lists with dynamic allocation are made of nodes. We can define a structure to represent a node, even if we are working with arrays.
- A node (for a doubly linked list) contains the information and links towards the previous and the next nodes:

DLLANode:
    info: TElem
    next: Integer
    prev: Integer

DLLA

- Having defined the *DLLANode* structure, we only need one array, which will contain *DLLANodes*.

- Since it is a doubly linked list, we keep both the head and the tail of the list.

DLLA:
  nodes: DLLANode[]
  cap: Integer
  head: Integer
  tail: Integer
  firstEmpty: Integer
  size: Integer *//it is not mandatory, but useful*

DLLA - Allocate and free

- To make the representation and implementation even more similar to a dynamically allocated linked list, we can define the *allocate* and *free* functions as well.

function allocate(dlla) **is:**
//pre: dlla is a DLLA
//post: a new element will be allocated and its position returned
    newElem ← dlla.firstEmpty
    if newElem ≠ -1 **then**
        dlla.firstEmpty ← dlla.nodes[dlla.firstEmpty].next // reset the first empty
        if dlla.firstEmpty ≠ -1 **then**
            dlla.nodes[dlla.firstEmpty].prev ← -1
        end-if
        dlla.nodes[newElem].next ← -1
        dlla.nodes[newElem].prev ← -1
    end-if
    allocate ← newElem
end-function

DLLA - Allocate and free

subalgorithm free (dlla, poz) **is:**
//pre: dlla is a DLLA, poz is an integer number
//post: the position poz was freed
  dlla.nodes[poz].next ← dlla.firstEmpty
  dlla.nodes[poz].prev ← -1
  if dlla.firstEmpty ≠ -1 **then**
    dlla.nodes[dlla.firstEmpty].prev ← poz
  end-if
  dlla.firstEmpty ← poz
end-subalgorithm

DLLA - InsertPosition

```
subalgorithm insertPosition(dlla, elem, poz) is:
//pre: dlla is a DLLA, elem is a TElem, poz is an integer number
//post: the element elem is inserted in dlla at position poz
   if poz < 1 OR poz > dlla.size + 1 execute
      @throw exception
   end-if
   newElem ← alocate(dlla)
   if newElem = -1 then
      @resize
      newElem ← alocate(dlla)
   end-if
   dlla.nodes[newElem].info ← elem
   if poz = 1 then
      if dlla.head = -1 then
         dlla.head ← newElem
         dlla.tail ← newElem
      else
//continued on the next slide...
```

```
            dlla.nodes[newElem].next ← dlla.head
            dlla.nodes[dlla.head].prev ← newElem
            dlla.head ← newElem
        end-if
    else
        nodC ← dlla.head
        pozC ← 1
        while nodC ≠ -1 and pozC < poz - 1 execute
            nodC ← dlla.nodes[nodC].next
            pozC ← pozC + 1
        end-while
        if nodC ≠ -1 then //it should never be -1, the position is correct
            nodNext ← dlla.nodes[nodC].next
            dlla.nodes[newElem].next ← nodNext
            dlla.nodes[newElem].prev ← nodC
            dlla.nodes[nodC].next ← newElem
//continued on the next slide...
```

DLLA - InsertPosition

```
            if nodNext = -1 then
                dlla.tail ← newElem
            else
                dlla.nodes[nodNext].prev ← newElem
            end-if
        end-if
    end-if
end-subalgorithm
```

- Complexity: $O(n)$

DLLA - Iterator

- The iterator for a DLLA contains as *current element* the index of the current node from the array.

DLLAIterator:
  list: DLLA
  currentElement: Integer

DLLAIterator - init

subalgorithm init(it, dlla) **is:**
//pre: dlla is a DLLA
//post: it is a DLLAIterator for dlla
  it.list ← dlla
  it.currentElement ← dlla.head
end-subalgorithm

- For a (dynamic) array, currentElement is set to 1 when an iterator is created. For a DLLA we need to set it to the head of the list (which might be position 1, but it might be a different position as well).
- Complexity: $\Theta(1)$

DLLAIterator - getCurrent

subalgorithm getCurrent(it) **is:**
//pre: it is a DLLAIterator, it is valid
//post: e is a TElem, e is the current element from it
//throws exception if the iterator is not valid
  if it.currentElement = -1 **then**
    @throw exception
  end-if
  getCurrent ← it.list.nodes[it.currentElement].info
end-subalgorithm

- Complexity: $\Theta(1)$

DLLAIterator - next

subalgoritm next (it) **is:**
//pre: it is a DLLAIterator, it is valid
//post: the current elements from it is moved to the next element
//throws exception if the iterator is not valid
  if it.currentElement = -1 **then**
    @throw exception
  end-if
  it.currentElement ← it.list.nodes[it.currentElement].next
end-subalgorithm

- In case of a (dynamic) array, going to the next element means incrementing the *currentElement* by one. For a DLLA we need to follow the links.
- Complexity: $\Theta(1)$

DLLAIterator - valid

```
function valid (it) is:
//pre: it is a DLLAIterator
//post: valid return true is the current element is valid, false otherwise
   if it.currentElement = -1 then
      valid ← False
   else
      valid ← True
   end-if
end-function
```

- Complexity: $\Theta(1)$

Iterator - why do we need it? I

- Most containers have iterators and for (almost) 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?

Iterator - why do we need it? II

- 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?
- 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.

    not good $O(n)$
  - 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?
- 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

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?

- 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.

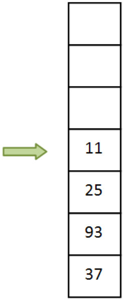
GetMinimum in constant time

- 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)?

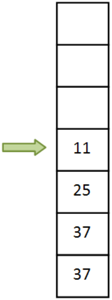
- We can keep an auxiliary stack, containing as many elements as the original stack, but containing the minimum value up to each element. Let's call this auxiliary stack a *min stack* and the original stack the *element stack*.

GetMinimum in constant time - Example

- If this is the *element stack*:



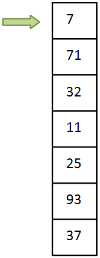
- This is the corresponding *min stack*:



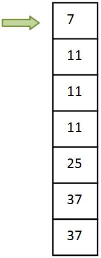
GetMinimum in constant time - Example

- 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:

```
SpecialStack:
    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)
```

SpecialStack - Notes / Think about it

- 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

Queue - representation on a SLL

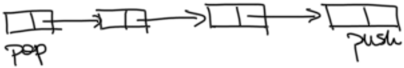
- If we want to implement a Queue using a singly linked list, where should we place the *front* and the *rear* of the queue?

- 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?



- We can easily insert after the tail in a SLL, but we cannot remove it in $\Theta(1)$ time (you need the previous node for removal).

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

both work the same?

- 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.

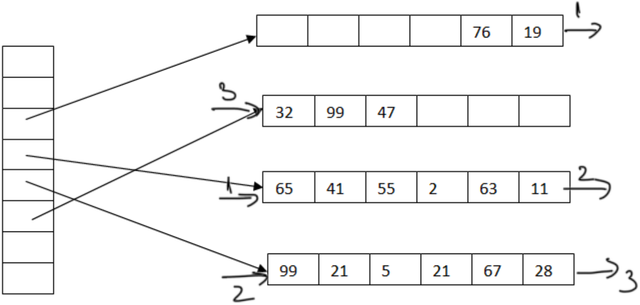
ADT Deque

- 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.

- The above representation is used by C++, because in C++ deques have another important operation besides the already mentioned ones: access to element based on position.
- What is the complexity of this operation for this representation?
- And on the alternative representations?

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

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

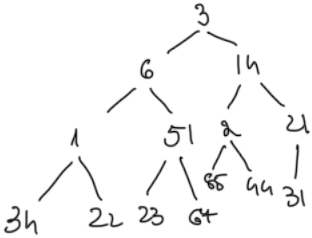
- What happens if we keep in a separate field the element with the highest priority?

Binary Heap

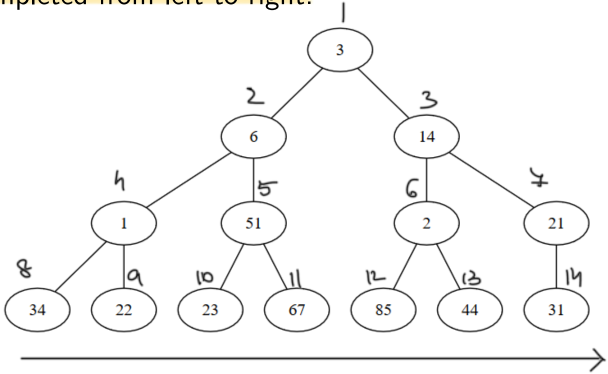
- A binary heap is a data structure that can be used as an efficient representation for Priority Queues.
- A binary heap is a kind of hybrid between a dynamic array and a binary tree.
- The elements of the heap are actually stored in the dynamic array, but the array is visualized as a binary tree.

- 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.



Binary Heap

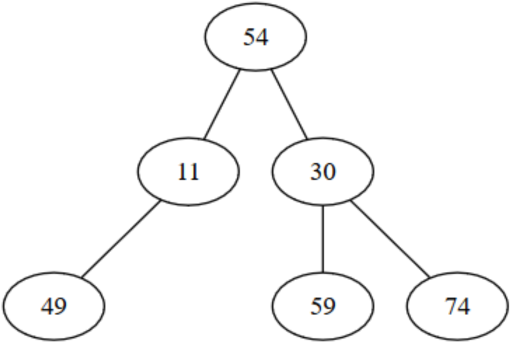
- If the elements of the array are: $a_1, a_2, a_3, ..., a_n$, we know that:
  - a_1 is the root of the heap
  - for an element from position i, its children are on positions $2 * i$ and $2 * i + 1$ (if $2 * i$ and $2 * i + 1$ is less than or equal to n)
  - for an element from position i (i $>$ 1),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

Binary Heap

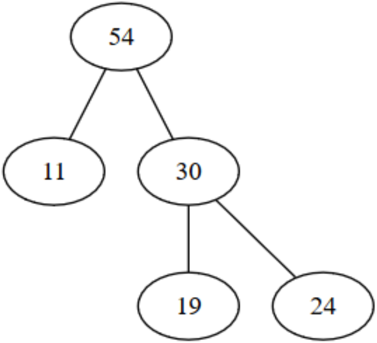
- A *binary heap* is an array that can be visualized as a binary tree having a *heap structure* and a *heap property*.
  - *Heap structure:* in the binary tree every node has exactly 2 children, except for the last two levels, where children are completed from left to right.
  - *Heap property:* $a_i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

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.

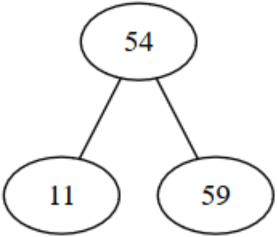


both

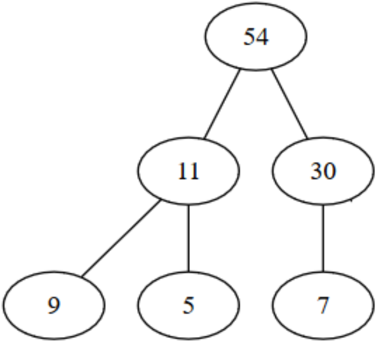


wrong structure

Binary Heap - Examples III

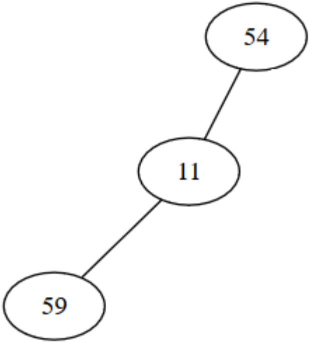


Good structure, but not the property

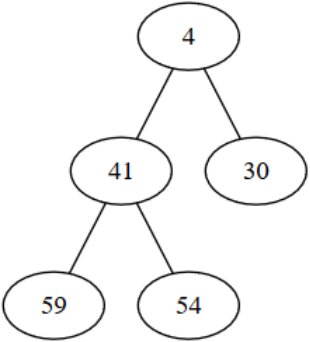


correct max heap

Binary Heap - Examples V

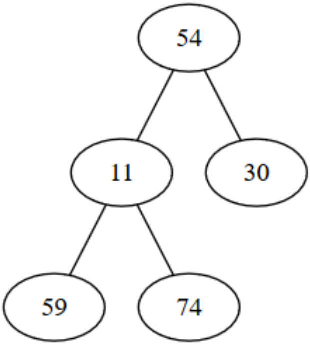


both



correct min heap

Binary Heap - Examples VII



NO - property

- 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~~] 8, 3 — max heap
    2. [1, 2, 4, 8, 16, 32, 64, 65, 10] — good   min heap
    3. [10, 12, ~~100~~, 60, 13, 102, 101, 80, 90, 14, 15, ~~16~~] — min heap

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).

Summary

- Today we have talked about:
  - Doubly linked list on array
  - Iterators
  - Stack, Queue, Deque implementations
  - Priority queue implementation
  - Binary heap