

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 1

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

- Course organization
- Abstract data types and Data structures
- Pseudocode
- Algorithm analysis

- Guiding teachers

- Lect. PhD. Oneț-Marian Zsuzsanna
- Lect. PhD. Lupșa Dana
- PhD Student Mihai Andrei
- PhD Student Albu Alexandra

- Activities

- **Lecture:** 2 hours / week
- **Seminar:** 1 hour / week
- **Lab:** 1 hour / week
- **Email:** [marianzs@cs.ubbcluj.ro](mailto:marianzs@cs.ubbcluj.ro)
  - Please use your *ubbcluj* email address for communication (and make sure the email contains your name!). Alternatively, you can use MS Teams private chat.

## • Grading

- Written exam (**W**) - will be in the exam session
- Lab grade (**LG**) - average weighted grade of the presented lab assignments (more details in the LabRules.pdf document)
- Seminar bonus (**B**) - maximum 0.5 points bonus, received for the seminar activity.
- The final grade is computed as:

$$G = 0.7 * W + 0.3 * LG + B$$

- **To pass the exam  $W$  and  $LG$  have to be  $\geq 5$  (no rounding)!**

# Platforms used during the semester

- MS Teams - will be used for Lectures, Seminars and Labs. All the materials (lecture slides, lab requirements, etc.) will be uploaded in the General channel at the Files section.
- Moodle - used for optional quizzes and for the written exam at the end of the semester.
  - After every lecture you will have a quiz on Moodle, where you can test how well you understood the lecture. The quiz is optional, and will be available for one week.

# Rules I

- Attendance is compulsory for the laboratory and the seminar activity. You need at least
  - 6 attendances from the 7 labs
  - 5 attendances from the 7 seminars.
- **Unless you have the required number of attendances, you cannot participate in the written exam, neither in the regular nor in the retake session!**
- The General channel in Files → Class material folder contains a document (AttendanceDocLink.pdf) with a link to a Google Sheets document where you can check your attendance situation using your code from AcademicInfo. In this document you will also see your lab assignments, lab grades and seminar activity.

- You have to come to the seminar with your group and to the lab with your semi-group (we will consider the official student lists for the groups from the faculty's web page).
- If you want to *permanently* switch from one (semi-)group to another, you have to find a person in the other (semi-)group who is willing to switch with you and announce your lab/seminar teacher about the switch in the first two weeks of the semester.
- One seminar and one laboratory attendance can be recovered with another group, within the two weeks allocated for the seminar/laboratory, with the explicit agreement of the seminar/laboratory teacher.

- In case of illness, absences will be motivated by the lab/seminar teacher, based on a medical certificate. Medical certificates have to be presented in the first seminar/lab after the absence, certificates presented later than that will not be accepted.

- In the retake session only the written exam can be repeated, and grade G will be computed in the same way as in the regular session.
- If the lab grade, LG, is not at least 5, you cannot participate at the written exam in the regular exam session. In the retake session you can participate at the written exam and you will have to turn in all the laboratory assignments, for a maximum grade of 5.

# Bibliography I

- T. CORMEN, C. LEISERSON, R. RIVEST, C. STEIN,  
*Introduction to algorithms*, Third Edition, The MIT Press,  
2009
- S. SKIENA, *The algorithm design manual*, Second Edition,  
Springer, 2008
- N. KARUMANCHI, *Data structures and algorithms made  
easy*, CareerMonk Publications, 2016
- M. A WEISS, *Data structures and algorithm analysis in Java*,  
Third Edition, Pearson, 2012

# Bibliography II

- R. SEDGEWICK, *Algorithms*, Fourth Editions, Addison-Wesley Publishing Company, 2011
- R. LAFORE, *Data Structures & Algorithms in Java*, Second Edition, Sams Publishing, 2003
- M. A. WEISS *Data structures and problem solving using C++*, Second Edition, Pearson, 2003

# Course Objectives

- The study of the concept of **abstract data types** and the most frequently used container abstract data types.
- The study of different **data structures** that can be used to implement these abstract data types and the complexity of their operations.
- What you should learn from this course:
  - to design and implement different applications starting from the use of abstract data types.
  - to process data stored in different data structures.
  - to choose the abstract data type and data structure best suited for a given application.

# Abstract Data Types

- What is a data type? Could you give me examples of data types that you know?

# Abstract Data Types

- What is a data type? Could you give me examples of data types that you know?
- A *data type* is a set of values (domain) and a set of operations on those values. For example:
  - int
    - set of values are integer numbers from a given interval
    - possible operations: add, subtract, multiply, etc.
  - boolean
    - set of values: true, false
    - possible operations: negation, and, or, xor, etc.
  - String
    - "abc", "text", etc.
    - possible operations: get the length, get a character from a position, get a substring, concatenate two strings, etc.
  - etc.
- We can have built-in data types and user defined data types.

# Abstract Data Types

- An Abstract Data Type (ADT) is a *data type* having the following two properties:
  - the objects from the domain of the ADT are specified independently of their representation
  - the operations of the ADT are specified independently of their implementation

- The domain of an ADT describes what elements belong to this ADT.
- If the domain is finite, we can simply enumerate them.
- If the domain is not finite, we will use a rule that describes the elements belonging to the ADT.

# Abstract Data Types - Interface

- After specifying the domain of an ADT, we need to specify its operations.
- The set of all operations for an ADT is called its *interface*.
- The interface of an ADT contains the *signature* of the operations, together with their input data, results, preconditions and postconditions (but no detail regarding the implementation of the method).
- When talking about ADT we focus on the **WHAT** (it is, it does), not on the **HOW** (it is represented, it is implemented).

# ADT - Example I

- Consider the Date ADT (made of three numbers: year, month and day)
  - Elements belonging to this ADT are all the valid dates (year is positive, maybe less than 3000, month is between 1 and 12, day is between 1 and maximum number of possible days for the month)
  - One possible operation for the Date ADT could be:  
 $\text{difference}(\text{Date } d1, \text{Date } d2)$ 
    - Descr:** computed the difference in number of days between two dates
    - Pre:**  $d1$  and  $d2$  have to be valid Dates,  $d1 \leq d2$
    - Post:**  $\text{difference} \leftarrow d$ ,  $d$  is a natural number and represents the number of days between  $d1$  and  $d2$ .
    - Throws:** an exception if  $d1$  is greater than  $d2$

- This information specifies everything we need to use the Date ADT, even if we know nothing about how it is represented or how the operation *difference* is implemented.

- A *container* is a collection of data, in which we can add new elements and from which we can remove elements.
- Different containers are defined based on different properties:
  - do the elements need to be unique?
  - do the elements have positions assigned?
  - can any element be accessed or just some specific ones?
  - do we store simple elements or key - value pairs?

- A container should provide at least the following operations (the interface of the container should contain operations for):
  - *creating* an empty container
  - *adding* a new element to the container
  - *removing* an element from the container
  - returning the *number of elements* in the container
  - provide *access to the elements* from the container (usually using an *iterator*)

# Container vs. Collection

- Python - Collections
- C++ - Containers from STL
- Java - Collections framework and the Apache Collections library
- .Net - System.Collections framework
- In the following, in this course we will use the term **container**.

- During the semester we are going to talk about many different containers, but there are two containers that you are already familiar with from Python:

- During the semester we are going to talk about many different containers, but there are two containers that you are already familiar with from Python: *list* and *dict*.
- Moreover, you know them and you have used them as ADT!

- During the semester we are going to talk about many different containers, but there are two containers that you are already familiar with from Python: *list* and *dict*.
- Moreover, you know them and you have used them as ADT!
  - Do you know how a *list* or a *dict* is represented?
  - Do you know how their operations are implemented?
  - Did not knowing these, stop you from using them?

# Why container abstract data types?

- There are several different container Abstract Data Types, so choosing the most suitable one is an important step during application design.
- When choosing the suitable ADT we are not interested in the implementation details of the ADT (yet).
- Most high-level programming languages usually provide implementations for different Abstract Data Types.
  - In order to be able to use the right ADT for a specific problem, we need to know their domain and interface.

## Example

- Assume that you have to write an application to handle Roman Numerals (maybe transform them into Arabic numerals, or transform Arabic numerals into Roman ones, for example CXXI into 121 or 529 into DXXIX).
- Maybe you want to define some new "numerals" as well, for example, W to represent the number 200.
- In order to implement this application you will often need to find the number corresponding to a given letter and vice-versa.
- What container would you use?

# Advantages of working with ADTs I

- *Abstraction* is defined as the separation between the specification of an object (its domain and interface) and its implementation.
- *Encapsulation* - abstraction provides a promise that any implementation of an ADT will belong to its domain and will respect its interface. And this is all that is needed to use an ADT.

# Advantages of working with ADTs II

- *Localization of change* - any code that uses an ADT is still valid if the ADT changes (because no matter how it changes, it still has to respect the domain and interface).
- *Flexibility* - an ADT can be implemented in different ways, but all these implementation have the same interface. Switching from one implementation to another can be done with minimal changes in the code.

# Advantages of working with ADTs III

- Consider again our example with the Date ADT:
  - Assume that you have decided to represent the Date by using 3 integer numbers: one for the year, one for the month and one for the day.
  - Assume that you have defined a lot of operations for Date: compute the difference between two dates, check if a date is in the weekend, add a given number of days to a date, etc.
  - Assume that you have also defined an ADT to represent a *TimeInterval*, which contains two Dates: the start and the end of the interval.
  - Assume that you have implemented a list of operations for *TimeInterval*: check if two intervals overlap, check if an interval includes a specific day, create an interval starting from a date and a number of days, etc.

# Advantages of working with ADTs IV

- Assume that you have implemented a complex application for scheduling holidays (*TimeIntervals*) and setting up deadlines for tasks (*Date*) and all sort of similar functionalities.
- What happens if now you want to modify the representation of the Date? Instead of storing 3 numbers, you want to store a date as a number with 8 digits, first 4 digits are the year, next two are the month and the last two are the days. How much of your code do you need to change?

# Advantages of working with ADTs V

- Well, obviously ADT Date needs to be rewritten, together with all the operations defined for it.
- But, if you worked with Date as an ADT, even if the representation changes and even if the implementation of the operations changes, they still have to respect the interface (the signature and behaviour of the difference operation is still the same), the rest of the application is not affected by the changes.

- The domain of data structures studies how we can store and access data.
- A data structure can be:
  - Static: the size of the data structure is fixed. Such data structures are suitable if it is known that a fixed number of elements need to be stored.
  - Dynamic: the size of the data structure can grow or shrink as needed by the number of elements.

- For every container ADT we will discuss several possible data structures that can be used for the implementation. For every possibility we will discuss the advantages and disadvantages of using the given data structure. We will see that, in general, we cannot say that there is one single *best* data structure for a given container.

# Why?

- Why do we need to implement our own Abstract Data Types if they are already implemented in most programming languages?

# Why?

- Why do we need to implement our own Abstract Data Types if they are already implemented in most programming languages?
  - Implementing these ADT will help us understand better how they work (we cannot use them, if we do not know what they are doing)
    - Did you know that in Python the following instruction has a different complexity depending on whether *cont* is a list or a dict?

```
if elem in cont:  
    | print("Found")
```

- To learn to create, implement and use ADT for situations when:
  - we work in a programming language where they are not readily implemented.
  - we need an ADT which is not part of the standard ones, but might be similar to them.

- The aim of this course is to give a general description of data structures, one that does not depend on any programming language - so we will use the *pseudocode* language to describe the algorithms.
- Our algorithms written in pseudocode will consist of two type of instructions:
  - standard instructions (assignment, conditional, repetitive, etc.)
  - non-standard instructions (written in plain English to describe parts of the algorithm that are not developed yet). These non-standard instructions will start with @.

- One line comments in the code will be denoted by //
- For reading data we will use the standard instruction **read**
- For printing data we will use the standard instruction **print**
- For assignment we will use ←
- For testing the equality of two variables we will use =

- Conditional instruction will be written in the following way  
(the *else* part can be missing):

```
if condition then
    @instructions
else
    @instructions
end-if
```

- The *for* loop (loop with a known number of steps) will be written in the following way:

```
for i ← init, final, step execute  
    @instructions  
end-for
```

- init* - represents the initial value for variable i
- final* - represents the final value for variable i
- step* - is the value added to i at the end of each iteration. *step* can be missing, in this case it is considered to be 1.

- The *while* loop (loop with an unknown number of steps) will be written in the following way:

```
while condition execute
    @instructions
end WHILE
```

- Subalgorithms (subprograms that do not return a value) will be written in the following way:

```
subalgorithm name(formal parameter list) is:  
    @instructions - subalgorithm body  
end-subalgorithm
```

- The subalgorithm can be called as:

```
name (actual parameter list)
```

# Pseudocode VII

- Functions (subprograms that return a value) will be written in the following way:

```
function name (formal parameter list) is:  
    @instructions - function body  
    name  $\leftarrow$  v //syntax used to return the value v  
end-function
```

- The function can be called as:

```
result  $\leftarrow$  name (actual parameter list)
```

## Pseudocode VIII

- If we want to define a variable  $i$  of type Integer, we will write:  
 $i : \text{Integer}$
- If we want to define an array  $a$ , having elements of type  $T$ , we will write:  $a : T[]$ 
  - If we know the size of the array, we will use:  $a : T[Nr]$  - indexing is done from 1 to Nr
  - If we do not know the size of the array, we will use:  $a : T[]$  - indexing is done from 1

- A struct (record) will be defined as:

Array:

n: Integer  
elems: T[]

- The above struct consists of 2 fields: *n* of type Integer and an array of elements of type T called *elems*
- Having a variable *var* of type Array, we can access the fields using . (dot):
  - var.n
  - var.elems
  - var.elems[i] - the i-th element from the array

- For denoting pointers (variables whose value is a memory address) we will use  $\uparrow$ :
  - $p: \uparrow \text{Integer}$  -  $p$  is a variable whose value is the address of a memory location where an Integer value is stored.
  - The value from the address denoted by  $p$  is accessed using  $[p]$
- Allocation and de-allocation operations will be denoted by:
  - $\text{allocate}(p)$
  - $\text{free}(p)$
- We will use the special value NIL to denote an invalid address

# Specifications I

- An operation will be specified in the following way:
  - **pre:** - the preconditions of the operation
  - **post:** - the postconditions of the operation
  - **throws:** - exceptions thrown (optional - not every operation can throw an exception)
- When using the name of a parameter in the specification we actually mean its value.
- Having a parameter  $i$  of type  $T$ , we will denote by  $i \in T$  the condition that the value of variable  $i$  belongs to the domain of type  $T$ .

- The value of a parameter can be changed during the execution of a function/subalgorithm. To denote the difference between the value before and after execution, we will use the ' (apostrophe).
- For example, the specification of an operation *decrement*, that decrements the value of a parameter  $x$  ( $x : \text{Integer}$ ) will be:
  - **pre:**  $x \in \text{Integer}$
  - **post:**  $x' = x - 1$

# Generic Data Types I

- We will consider that the elements of a container ADT are of a generic type:  $TElem$
- The interface of the  $TElem$  contains the following operations:
  - assignment ( $e_1 \leftarrow e_2$ )
    - pre:**  $e_1, e_2 \in TElem$
    - post:**  $e'_1 = e_2$
  - equality test ( $e_1 = e_2$ )
    - pre:**  $e_1, e_2 \in TElem$
    - post:**

$$equal \leftarrow \begin{cases} True, & \text{if } e_1 \text{ equals } e_2 \\ False, & \text{otherwise} \end{cases}$$

# Generic Data Types II

- When the values of a data type can be compared or ordered based on a relation, we will use the generic type:  $TComp$ .
- Besides the operations from  $TElem$ ,  $TComp$  has an extra operation that compares two elements:
  - $\text{compare}(e_1, e_2)$ 
    - pre:**  $e_1, e_2 \in TComp$
    - post:**
- For simplicity, sometimes instead of calling the  $compare$  function, we will use the notations  $e_1 \leq e_2$ ,  $e_1 = e_2$ ,  $e_1 \geq e_2$

$$\text{compare} \leftarrow \begin{cases} \text{true} & \text{if } e_1 \leq e_2 \\ \text{false} & \text{if } e_1 > e_2 \end{cases}$$

# The RAM model I

- Analyzing an algorithm usually means predicting the resources (time, memory) the algorithm requires. In order to do so, we need a hypothetical computer model, called *RAM* (random-access machine) model.
- In the RAM model:
  - Each simple operation (+, -, \*, /, =, if, function call) takes one time step/unit.
  - We have fixed-size integers and floating point data types.
  - Loops and subprograms are *not* simple operations and we do not have special operations (ex. sorting in one instruction).
  - Every memory access takes one time step and we have an infinite amount of memory.

# The RAM model II

- The RAM is a very simplified model of how computers work, but in practice it is a good model to understand how an algorithm will perform on a real computer.
- Under the RAM model we measure the run time of an algorithm by counting the number of steps the algorithm takes on a given input instance. The number of steps is usually a function that depends on the size of the input data.

**subalgorithm** something( $n$ ) **is:**

// $n$  is an Integer number

**rez**  $\leftarrow$  0

**for**  $i \leftarrow 1, n$  **execute**

**sum**  $\leftarrow$  0

**for**  $j \leftarrow 1, n$  **execute**

**sum**  $\leftarrow$  **sum** +  $j$

**end-for**

**rez**  $\leftarrow$  **rez** + **sum**

**end-for**

**print** **rez**

**end-subalgorithm**

- How many steps does the above subalgorithm take?

**subalgorithm** something( $n$ ) **is:**

// $n$  is an Integer number

**rez**  $\leftarrow$  0

**for**  $i \leftarrow 1, n$  **execute**

**sum**  $\leftarrow$  0

**for**  $j \leftarrow 1, n$  **execute**

**sum**  $\leftarrow$  **sum** +  $j$

**end-for**

**rez**  $\leftarrow$  **rez** + **sum**

**end-for**

**print** **rez**

**end-subalgorithm**

- How many steps does the above subalgorithm take?
- $T(n) = 1 + n * (1 + n + 1) + 1 = n^2 + 2n + 2$

# Order of growth

- We are not interested in the exact number of steps for a given algorithm, we are interested in its *order of growth* (i.e., how does the number of steps change if the value of  $n$  increases)
- We will consider only the leading term of the formula (for example  $n^2$ ), because the other terms are relatively insignificant for large values of  $n$ .

## O-notation

For a given function  $g(n)$  we denote by  $O(g(n))$  the set of functions:

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s. t. } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\}$$

- The O-notation provides an *asymptotic upper bound* for a function: for all values of  $n$  (to the right of  $n_0$ ) the value of the function  $f(n)$  is on or below  $c \cdot g(n)$ .
- We will use the notation  $f(n) = O(g(n))$  or  $f(n) \in O(g(n))$ .

# O-notation II

- Graphical representation:

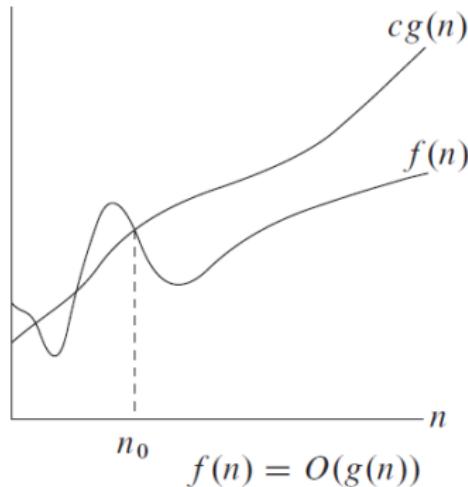


Figure taken from Cormen et. al: Introduction to algorithms, MIT Press, 2009

## Alternative definition

$$f(n) \in O(g(n)) \text{ if } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

is either 0 or a constant (but not  $\infty$ ).

- Consider, for example,  $T(n) = n^2 + 2n + 2$ :
  - $T(n) = O(n^2)$  because  $T(n) \leq c * n^2$  for  $c = 2$  and  $n \geq 3$
  - $T(n) = O(n^3)$  because

$$\lim_{n \rightarrow \infty} \frac{T(n)}{n^3} = 0$$

## $\Omega$ -notation

For a given function  $g(n)$  we denote by  $\Omega(g(n))$  the set of functions:

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s. t. } 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0\}$$

- The  $\Omega$ -notation provides an *asymptotic lower bound* for a function: for all values of  $n$  (to the right of  $n_0$ ) the value of the function  $f(n)$  is on or above  $c \cdot g(n)$ .
- We will use the notation  $f(n) = \Omega(g(n))$  or  $f(n) \in \Omega(g(n))$ .

# $\Omega$ -notation II

- Graphical representation:

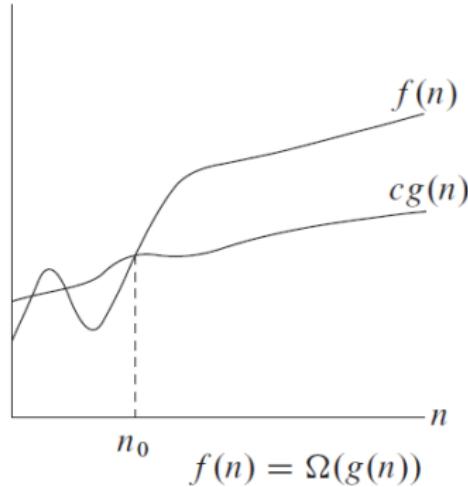


Figure taken from Cormen et. al: Introduction to algorithms, MIT Press, 2009

## Alternative definition

$$f(n) \in \Omega(g(n)) \text{ if } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

is  $\infty$  or a nonzero constant.

- Consider, for example,  $T(n) = n^2 + 2n + 2$ :
  - $T(n) = \Omega(n^2)$  because  $T(n) \geq c * n^2$  for  $c = 0.5$  and  $n \geq 1$
  - $T(n) = \Omega(n)$  because

$$\lim_{n \rightarrow \infty} \frac{T(n)}{n} = \infty$$

## $\Theta$ -notation

For a given function  $g(n)$  we denote by  $\Theta(g(n))$  the set of functions:

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ s. t. } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0\}$$

- The  $\Theta$ -notation provides an *asymptotically tight bound* for a function: for all values of  $n$  (to the right of  $n_0$ ) the value of the function  $f(n)$  is between  $c_1 \cdot g(n)$  and  $c_2 \cdot g(n)$ .
- We will use the notation  $f(n) = \Theta(g(n))$  or  $f(n) \in \Theta(g(n))$ .

# $\Theta$ -notation II

- Graphical representation:

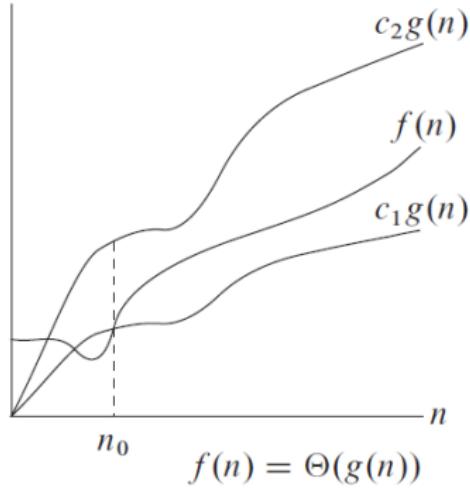


Figure taken from Cormen et. al: Introduction to algorithms, MIT Press, 2009

## Alternative definition

$$f(n) \in \Theta(g(n)) \text{ if } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

is a nonzero constant (and not  $\infty$ ).

- Consider, for example,  $T(n) = n^2 + 2n + 2$ :
  - $T(n) = \Theta(n^2)$  because  $c_1 * n^2 \leq T(n) \leq c_2 * n^2$  for  $c_1 = 0.5$ ,  $c_2 = 2$  and  $n \geq 3$ .
  - $T(n) = \Theta(n^2)$  because

$$\lim_{n \rightarrow \infty} \frac{T(n)}{n^2} = 1$$

# Best Case, Worst Case, Average Case I

- Think about an algorithm that finds the sum of all even numbers in an array. How many steps does the algorithm take for an array of length  $n$ ?
- Think about an algorithm that finds the first occurrence of a number in an array. How many steps does the algorithm take for an array of length  $n$ ?

## Best Case, Worst Case, Average Case II

- For the second problem the number of steps taken by the algorithm does not depend just on the length of the array, it depends on the exact values from the array as well.
- For an array of fixed length  $n$ , execution of the algorithm can stop after:
  - verifying the first number - if it is the one we are looking for
  - verifying the first two numbers - if the first is not the one we are looking for, but the second is
  - ...
  - verifying all  $n$  numbers - first  $n - 1$  are not the one we are looking for, but the last is, or none of the numbers is the one we are looking for

# Best Case, Worst Case, Average Case III

- For such algorithms we will consider three cases:
  - Best - Case - the best possible case, where the number of steps taken by the algorithm is the minimum that is possible
  - Worst - Case - the worst possible case, where the number of steps taken by the algorithm is the maximum that is possible
  - Average - Case - the average of all possible cases.
- Best and Worst case complexity is usually computed by inspecting the code. For our example we have:
  - Best case:  $\Theta(1)$  - just the first number is checked, no matter how large the array is.
  - Worst case:  $\Theta(n)$  - we have to check all the numbers

- For computing the average case complexity we have a formula:

$$\sum_{I \in D} P(I) \cdot E(I)$$

- where:
  - $D$  is the domain of the problem, the set of every possible input that can be given to the algorithm.
  - $I$  is one input data
  - $P(I)$  is the probability that we will have  $I$  as an input
  - $E(I)$  is the number of operations performed by the algorithm for input  $I$

## Best Case, Worst Case, Average Case V

- For our example  $D$  would be the set of all possible arrays with length  $n$
- Every  $I$  would represent a subset of  $D$ :
  - One  $I$  represents all the arrays where the first number is the one we are looking for
  - One  $I$  represents all the arrays where the first number is not the one we are looking for, but the second is
  - ...
  - One  $I$  represents all the arrays where the first  $n - 1$  elements are not the one we are looking for but the last one is
  - One  $I$  represents all the arrays which do not contain the element we are looking for
- $P(I)$  is usually considered equal for every  $I$ , in our case  $\frac{1}{n+1}$

$$T(n) = \frac{1}{n+1} \sum_{i=1}^n i + \frac{n}{n+1} = \frac{n \cdot (n+1)}{2 \cdot (n+1)} + \frac{n}{n+1} \in \Theta(n)$$

- When we have best case, worst case and average case complexity, we will report the maximum one (which is the worst case), but if the three values are different, the total complexity is reported with the O-notation.
- For our example we have:
  - Best case:  $\Theta(1)$
  - Worst case:  $\Theta(n)$
  - Average case:  $\Theta(n)$
  - Total (overall) complexity:  $O(n)$
- Obs:** For best, worst and average case we will always use the  $\Theta$  notation.

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 2

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

# In Lecture 1...

- Course Organization
- Abstract Data Types and Data Structures
- Pseudocode
- Algorithm Analysis
  - $O$  - notation
  - $\Omega$  - notation
  - $\Theta$  - notation
  - Best Case, Worst Case, Average Case
  - Extra reading - Empirical algorithm analysis

- Algorithm analysis
- Dynamic Array
- Iterators

# Algorithm Analysis for Recursive Functions

- How can we compute the time complexity of a recursive algorithm?

# Recursive Binary Search

```
function BinarySearchR (array, elem, start, end) is:  
    //array - an ordered array of integer numbers  
    //elem - the element we are searching for  
    //start - the beginning of the interval in which we search (inclusive)  
    //end - the end of the interval in which we search (inclusive)  
    if start > end then  
        BinarySearchR ← False  
    end-if  
    middle ← (start + end) / 2  
    if array[middle] = elem then  
        BinarySearchR ← True  
    else if elem < array[middle] then  
        BinarySearchR ← BinarySearchR(array, elem, start, middle-1)  
    else  
        BinarySearchR ← BinarySearchR(array, elem, middle+1, end)  
    end-if  
end-function
```

# Recursive Binary Search

- The first call to the *BinarySearchR* algorithm for an ordered array of  $nr$  elements is:

```
BinarySearchR(array, elem, 1, nr)
```

- How do we compute the complexity of the *BinarySearchR* algorithm?

# Recursive Binary Search

- We will denote the length of the sequence that we are checking at every iteration by  $n$  (so  $n = end - start$ )
- We need to write the recursive formula of the solution

# Recursive Binary Search

- We will denote the length of the sequence that we are checking at every iteration by  $n$  (so  $n = end - start$ )
- We need to write the recursive formula of the solution

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ T\left(\frac{n}{2}\right) + 1, & \text{otherwise} \end{cases}$$

# Master method

- The *master method* can be used to compute the time complexity of algorithms having the following general recursive formula:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

- where  $a \geq 1$ ,  $b > 1$  are constants and  $f(n)$  is an asymptotically positive function.

- Advantage of the master method: we can determine the time complexity of a recursive algorithm without further computations.
- Disadvantage of the master method: we need to memorize the three cases of the method and there are some situations when none of these cases can be applied.

# Computing the time complexity without the master method

- If we do not want to memorize the cases for the master method we can compute the time complexity in the following way:
- Recall, the recursive formula for BinarySearchR was:

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ T\left(\frac{n}{2}\right) + 1, & \text{otherwise} \end{cases}$$

# Time complexity for BinarySearchR

- We suppose that  $n = 2^k$  and rewrite the second branch of the recursive formula:

$$T(2^k) = T(2^{k-1}) + 1$$

- Now, we write what the value of  $T(2^{k-1})$  is (based on the recursive formula)

$$T(2^{k-1}) = T(2^{k-2}) + 1$$

- Next, we add what the value of  $T(2^{k-2})$  is (based on the recursive formula)

$$T(2^{k-2}) = T(2^{k-3}) + 1$$

# Time complexity for BinarySearchR

- The last value that can be written is the value of  $T(2^1)$

$$T(2^1) = T(2^0) + 1$$

# Time complexity for BinarySearchR

- Now, we write all these equations together and add them (and we will see that many terms can be simplified, because they appear on the left hand side of an equation and the right hand side of another equation):

$$T(2^k) = T(2^{k-1}) + 1$$

$$T(2^{k-1}) = T(2^{k-2}) + 1$$

$$T(2^{k-2}) = T(2^{k-3}) + 1$$

...

$$T(2^1) = T(2^0) + 1$$

---

$$+$$

$$T(2^k) = T(2^0) + 1 + 1 + 1 + \dots + 1 = 1 + k$$

- Obs:** For non-recursive functions adding a +1 or not, does not influence the result. In case of recursive functions it is important to have another term besides the recursive one.

# Time complexity for BinarySearchR

- We started from the notation  $n = 2^k$ .
- We want to go back to the notation that uses  $n$ . If  $n = 2^k \Rightarrow k = \log_2 n$

$$T(2^k) = 1 + k$$

$$T(n) = 1 + \log_2 n \in \Theta(\log_2 n)$$

# Time complexity for BinarySearchR

- We started from the notation  $n = 2^k$ .
- We want to go back to the notation that uses  $n$ . If  $n = 2^k \Rightarrow k = \log_2 n$

$$T(2^k) = 1 + k$$

$$T(n) = 1 + \log_2 n \in \Theta(\log_2 n)$$

- Actually, if we look at the code from *BinarySearchR*, we can observe that it has a best case (element can be found at the first iteration), so final complexity is  $O(\log_2 n)$

## Another example

- Let's consider the following pseudocode and compute the time complexity of the algorithm:

```
subalgorithm operation(n, i) is:  
//n and i are integer numbers, n is positive  
if n > 1 then  
    i ← 2 * i  
    m ← n/2  
    operation(m, i-2)  
    operation(m, i-1)  
    operation(m, i+2)  
    operation(m, i+1)  
else  
    write i  
end-if  
end-subalgorithm
```

- The first step is to write the recursive formula:

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ 4 \cdot T\left(\frac{n}{2}\right) + 1, & \text{otherwise} \end{cases}$$

- We suppose that  $n = 2^k$ .

$$T(2^k) = 4 \cdot T(2^{k-1}) + 1$$

- This time we need the value of  $4 \cdot T(2^{k-1})$

$$\begin{aligned} T(2^{k-1}) &= 4 \cdot T(2^{k-2}) + 1 \Rightarrow \\ 4 \cdot T(2^{k-1}) &= 4^2 \cdot T(2^{k-2}) + 4 \end{aligned}$$

- And the value of  $4^2 \cdot T(2^{k-2})$

$$4^2 \cdot T(2^{k-2}) = 4^3 \cdot T(2^{k-3}) + 4^2$$

- The last value we can compute is  $4^{k-1} \cdot T(2^1)$

$$4^{k-1} \cdot T(2^1) = 4^k \cdot T(2^0) + 4^{k-1}$$

- We write all the equations and add them:

$$T(2^k) = 4 \cdot T(2^{k-1}) + 1$$

$$4 \cdot T(2^{k-1}) = 4^2 \cdot T(2^{k-2}) + 4$$

$$4^2 \cdot T(2^{k-2}) = 4^3 \cdot T(2^{k-3}) + 4^2$$

...

$$4^{k-1} \cdot T(2^1) = 4^k \cdot T(2^0) + 4^{k-1}$$

---

+

$$T(2^k) = 4^k \cdot T(1) + 4^0 + 4^1 + 4^2 + \dots + 4^{k-1}$$

- $T(1)$  is 1 (first case from recursive formula)

$$T(2^k) = 4^0 + 4^1 + 4^2 + \dots + 4^{k-1} + 4^k$$

$$\sum_{i=0}^n p^i = \frac{p^{n+1} - 1}{p - 1}$$

$$T(2^k) = \frac{4^{k+1} - 1}{4 - 1} = \frac{4^k \cdot 4 - 1}{3} = \frac{(2^k)^2 \cdot 4 - 1}{3}$$

- We started from  $n = 2^k$ . Let's change back to  $n$

$$T(n) = \frac{4n^2 - 1}{3} \in \Theta(n^2)$$

- A *record* (or *struct*) is a static data structure.
- It represents the reunion of a fixed number of components (which can have different types) that form a logical unit together.
- We call the components of a record *fields*.
- For example, we can have a record to denote a *Person* formed of fields for *name*, *date of birth*, *address*, etc.

## Person:

name: String

dob: String

address: String

etc.

- An array is one of the simplest and most basic data structures.
- An array can hold a fixed number of elements of the same type and these elements occupy a contiguous memory block.
- Arrays are often used as a basis for other (more complex) data structures.

- When a new array is created we have to specify two things:
  - The type of the elements in the array
  - The maximum number of elements that can be stored in the array (*capacity* of the array)
- The memory occupied by the array will be the capacity times the size of one element.
- The array itself is memorized by the address of the first element.

# Arrays - C++ Example 1

- An array of *boolean* values (addresses of the elements are displayed in base 16 and base 10)

# Arrays - C++ Example 1

- An array of *boolean* values (addresses of the elements are displayed in base 16 and base 10)

```
Size of boolean: 1
Address of array: 00EFF760
Address of element from position 0: 00EFF760 15726432
Address of element from position 1: 00EFF761 15726433
Address of element from position 2: 00EFF762 15726434
Address of element from position 3: 00EFF763 15726435
Address of element from position 4: 00EFF764 15726436
Address of element from position 5: 00EFF765 15726437
Address of element from position 6: 00EFF766 15726438
Address of element from position 7: 00EFF767 15726439
```

- Can you guess the address of the element from position 8?

## Arrays - C++ Example 2

- An array of *integer* values (integer values occupy 4 bytes)

```
Size of int: 4
Address of array: 00D9FE6C
Address of element from position 0: 00D9FE6C 14286444
Address of element from position 1: 00D9FE70 14286448
Address of element from position 2: 00D9FE74 14286452
Address of element from position 3: 00D9FE78 14286456
Address of element from position 4: 00D9FE7C 14286460
Address of element from position 5: 00D9FE80 14286464
Address of element from position 6: 00D9FE84 14286468
Address of element from position 7: 00D9FE88 14286472
```

- Can you guess the address of the element from position 8?

## Arrays - C++ Example 3

- An array of *fraction* record values (the fraction record is composed of two integers)

Size of fraction: 8

Address of array: 007BF97C

Address of element from position 0: 007BF97C 8124796

Address of element from position 1: 007BF984 8124804

Address of element from position 2: 007BF98C 8124812

Address of element from position 3: 007BF994 8124820

Address of element from position 4: 007BF99C 8124828

Address of element from position 5: 007BF9A4 8124836

Address of element from position 6: 007BF9AC 8124844

Address of element from position 7: 007BF9B4 8124852

- Can you guess the address of the element from position 8?

- The main **advantage** of arrays is that any element of the array can be accessed in constant time ( $\Theta(1)$ ), because the address of the element can simply be computed (considering that the first element is at position 0):

Address of  $i^{th}$  element = address of array +  $i * \text{size of an element}$

- The above formula works even if we consider that the first element is at position 1, but then we need to use  $i - 1$  instead of  $i$ .

- An array is a static structure: once the *capacity* of the array is specified, you cannot add or delete slots from it (you can add and delete elements from the slots, but the number of slots, the capacity, remains the same)
- This leads to an important **disadvantage**: we need to know/estimate from the beginning the number of elements:
  - if the capacity is too small: we cannot store every element we want to
  - if the capacity is too big: we waste memory

# Dynamic Array

- There are arrays whose size can grow or shrink, depending on the number of elements that need to be stored in the array: they are called *dynamic arrays* (or *dynamic vectors*).
- Dynamic arrays are still arrays, the elements are still kept at contiguous memory locations and we still have the advantage of being able to compute the address of every element in  $\Theta(1)$  time.

# Dynamic Array - Representation

- In general, for a Dynamic Array we need the following fields:
  - *cap* - denotes the number of slots allocated for the array (its capacity)
  - *nrElem* - denotes the actual number of elements stored in the array
  - *elems* - denotes the actual array with *capacity* slots for TElems allocated

DynamicArray:

cap: Integer

nrElem: Integer

elems: TElem[]

# Dynamic Array - Resize

- When the value of *nrElem* equals the value of *capacity*, we say that the array is full. If more elements need to be added, the *capacity* of the array is increased (usually doubled) and the array is *resized*.
- During the *resize* operation a new, bigger array is allocated and the existing elements are copied from the old array to the new one.
- Optionally, *resize* can be performed after delete operations as well: if the dynamic array becomes "too empty", a resize operation can be performed to shrink its size (to avoid occupying unused memory).

# Dynamic Array - DS vs. ADT

- Dynamic Array is a data structure:
  - It describes how data is actually stored in the computer (in a single contiguous memory block) and how it can be accessed and processed
  - It can be used as representation to implement different abstract data types
- However, Dynamic Array is so frequently used that in most programming languages it exists as a separate container as well.
  - The Dynamic Array is not really an ADT, since it has one single possible implementation, but we still can treat it as an ADT, and discuss its interface.

- **Domain** of ADT DynamicArray

$$\mathcal{DA} = \{\mathbf{da} | da = (cap, nrElem, e_1 e_2 e_3 \dots e_{nrElem}), cap, nrElem \in N, nrElem \leq cap, e_i \text{ is of type TElem}\}$$

# Dynamic Array - Interface II

- What operations should we have for a *DynamicArray*?

# Dynamic Array - Interface III

- `init(da, cp)`
  - **description:** creates a new, empty DynamicArray with initial capacity  $cp$  (constructor)
  - **pre:**  $cp \in N^*$
  - **post:**  $da \in \mathcal{DA}$ ,  $da.cap = cp$ ,  $da.nrElem = 0$
  - **throws:** an exception if  $cp$  is zero or negative

- `destroy(da)`
  - **description:** destroys a DynamicArray (destructor)
  - **pre:**  $da \in \mathcal{DA}$
  - **post:**  $da$  was destroyed (the memory occupied by the dynamic array was freed)

- **size(da)**
  - **description:** returns the size (number of elements) of the DynamicArray
  - **pre:**  $da \in \mathcal{DA}$
  - **post:**  $\text{size} \leftarrow \text{the size of } da \text{ (the number of elements)}$

- `getElement(da, i)`

- **description:** returns the element from a position from the DynamicArray
- **pre:**  $da \in DA, 1 \leq i \leq da.nrElem$
- **post:**  $\text{getElement} \leftarrow e, e \in TElm, e = da.e_i$  (the element from position  $i$ )
- **throws:** an exception if  $i$  is not a valid position

- `setElement(da, i, e)`
  - **description:** changes the element from a position to another value
  - **pre:**  $da \in \mathcal{DA}$ ,  $1 \leq i \leq da.nrElem$ ,  $e \in TElm$
  - **post:**  $da' \in \mathcal{DA}$ ,  $da'.e_i = e$  (the  $i^{th}$  element from  $da'$  becomes  $e$ ),  $\text{setElement} \leftarrow e_{old}$ ,  $e_{old} \in TElm$ ,  $e_{old} \leftarrow da.e_i$  (returns the old value from position  $i$ )
  - **throws:** an exception if  $i$  is not a valid position

# Dynamic Array - Interface VIII

- **addToEnd(da, e)**
  - **description:** adds an element to the end of a DynamicArray.  
If the array is full, its capacity will be increased
  - **pre:**  $da \in \mathcal{DA}$ ,  $e \in TElm$
  - **post:**  $da' \in \mathcal{DA}$ ,  $da'.nrElem = da.nrElem + 1$ ;  $da'.e_{da'.nrElem} = e$  ( $da.cap = da.nrElem \Rightarrow da'.cap \leftarrow da.cap * 2$ )

- `addToPosition(da, i, e)`
  - **description:** adds an element to a given position in the `DynamicArray`. If the array is full, its capacity will be increased
  - **pre:**  $da \in \mathcal{DA}$ ,  $1 \leq i \leq da.nrElem + 1$ ,  $e \in TElm$
  - **post:**  $da' \in \mathcal{DA}$ ,  $da'.nrElem = da.nrElem + 1$ ,  $da'.e_j = da.e_{j-1} \forall j = da'.nrElem, da'.nrElem - 1, \dots, i + 1$ ,  $da'.e_i = e$ ,  $da'.e_j = da.e_j \forall j = i - 1, \dots, 1$  ( $da.cap = da.nrElem \Rightarrow da'.cap \leftarrow da.cap * 2$ )
  - **throws:** an exception if  $i$  is not a valid position ( $da.nrElem+1$  is a valid position when adding a new element)

- `deleteFromPosition(da, i)`
  - **description:** deletes an element from a given position from the DynamicArray. Returns the deleted element
  - **pre:**  $da \in \mathcal{DA}, 1 \leq i \leq da.nrElem$
  - **post:**  $\text{deleteFromPosition} \leftarrow e,$   
 $e \in TElem, e = da.e_i, da' \in \mathcal{DA}, da'.nrElem =$   
 $da.nrElem - 1, da'.e_j = da.e_{j+1} \forall 1 \leq j \leq da'.nrElem,$   
 $da'.e_j = da.e_j \forall 1 \leq j < i$
  - **throws:** an exception if  $i$  is not a valid position

- `iterator(da, it)`
  - **description:** returns an iterator for the DynamicArray
  - **pre:**  $da \in DA$
  - **post:**  $it \in \mathcal{I}$ ,  $it$  is an iterator over  $da$ , the current element from  $it$  refers to the first element from  $da$ , or, if  $da$  is empty,  $it$  is invalid

- Other possible operations:
  - Delete all elements from the Dynamic Array (make it empty)
  - Verify if the Dynamic Array is empty
  - Delete an element (given as element, not as position)
  - Check if an element appears in the Dynamic Array
  - Remove the element from the end of the Dynamic Array
  - etc.

- Most operations from the interface of the Dynamic Array are very simple to implement.
- In the following we will discuss the implementation of two operations: *addToEnd*, *addToPosition*.
- For the implementation we are going to use the representation discussed earlier:

DynamicArray:

cap: Integer

nrElem: Integer

elems: TElem[]

# Dynamic Array - addToEnd - Case 1

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

- capacity (cap): 10
- nrElem: 6

- Add the element 49 to the end of the dynamic array

# Dynamic Array - addToEnd - Case 1

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

- capacity (cap): 10
  - nrElem: 6
- Add the element 49 to the end of the dynamic array

51	32	19	31	47	95	49			
1	2	3	4	5	6	7	8	9	10

- capacity (cap): 10
- nrElem: 7

# Dynamic Array - addToEnd - Case 2

51	32	19	31	47	95
1	2	3	4	5	6

- capacity (cap): 6
- nrElem: 6

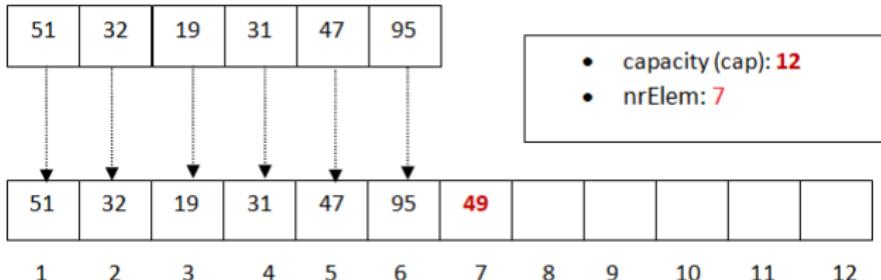
- Add the element 49 to the end of the dynamic array

# Dynamic Array - addToEnd - Case 2

51	32	19	31	47	95
1	2	3	4	5	6

- capacity (cap): 6
- nrElem: 6

- Add the element 49 to the end of the dynamic array



- capacity (cap): 12
- nrElem: 7

# Dynamic Array - addToEnd

```
subalgorithm addToEnd (da, e) is:  
    if da.nrElem = da.cap then  
        //the dynamic array is full. We need to resize it  
        da.cap ← da.cap * 2  
        newElems ← @ an array with da.cap empty slots  
        //we need to copy existing elements into newElems  
        for index ← 1, da.nrElem execute  
            newElems[index] ← da.elems[index]  
        end-for  
        //we need to replace the old element array with the new one  
        //depending on the prog. lang., we may need to free the old elems array  
        da.elems ← newElems  
    end-if  
    //now we certainly have space for the element e  
    da.nrElem ← da.nrElem + 1  
    da.elems[da.nrElem] ← e  
end-subalgorithm
```

- What is the complexity of *addToEnd*?

# Dynamic Array - addToPosition

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

- capacity (cap): 10
- nrElem: 6
- Add the element 49 to position 3

# Dynamic Array - addToPosition

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

- capacity (cap): 10
- nrElem: 6

• Add the element 49 to position 3

The diagram illustrates the insertion of element 49 at index 3. The array has indices 1 through 10 below it. Elements at indices 4, 3, 2, and 1 have dashed arrows pointing down to their respective positions in the array, indicating they will be shifted to the right to make space for the new element.

51	32	<b>49</b>	<b>19</b>	<b>31</b>	<b>47</b>	<b>95</b>			
1	2	3	4	5	6	7	8	9	10

- capacity (cap): 10
- nrElem: 7

• Add the element 49 to position 3

**subalgorithm** addToPosition (da, i, e) **is:**

**if**  $i > 0$  **and**  $i \leq da.nrElem + 1$  **then**

**if**  $da.nrElem = da.cap$  **then** //the dynamic array is full. We need to  
resize it

$da.cap \leftarrow da.cap * 2$

newElems  $\leftarrow @$  an array with  $da.cap$  empty slots

**for** index  $\leftarrow 1$ ,  $da.nrElem$  **execute**

newElems[index]  $\leftarrow da.elems[index]$

**end-for**

$da.elems \leftarrow newElems$

**end-if** //now we certainly have space for the element e

$da.nrElem \leftarrow da.nrElem + 1$

**for** index  $\leftarrow da.nrElem, i+1, -1$  **execute** //move the elements to the  
right

da.elems[index]  $\leftarrow da.elems[index-1]$

**end-for**

da.elems[i]  $\leftarrow e$

**else**

@throw exception

**end-if**

**end-subalgorithm**

- What is the complexity of *addToPosition*?

- Observations:

- While it is not mandatory to double the capacity, it is important to define the new capacity as a product of the old one with a constant number greater than 1 (just adding one new slot, or a constant number of new slots is not OK - you will see later why).

# Dynamic Array

- Observations:
  - While it is not mandatory to double the capacity, it is important to define the new capacity as a product of the old one with a constant number greater than 1 (just adding one new slot, or a constant number of new slots is not OK - you will see later why).
- How do dynamic arrays in other programming languages grow at resize?
  - Microsoft Visual C++ - multiply by 1.5 (initially 1, then 2, 3, 4, 6, 9, 13, etc.)
  - Java - multiply by 1.5 (initially 10, then 15, 22, 33, etc.)
  - Python - multiply by  $\approx 1.125$  (0, 4, 8, 16, 25, 35, 46, 58, etc.)
  - C# - multiply by 2 (initially 0, 4, 8, 16, 32, etc.)

# Dynamic Array

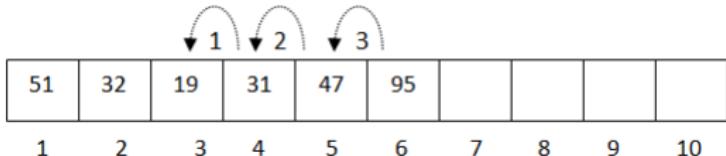
- After a resize operation the elements of the Dynamic Array will still occupy a contiguous memory zone, but it will be a different one than before.

# Dynamic Array

```
Address of the Dynamic Array structure: 00D3FE00 13893120
Length is: 3 si capacitate: 3
Address of array from DA: 0039E568 3794280
    Address of element from position 0 0039E568 3794280
    Address of element from position 1 0039E56C 3794284
    Address of element from position 2 0039E570 3794288
-----
Address of the Dynamic Array structure: 00D3FE00 13893120
Length is: 6 si capacitate: 6
Address of array from DA: 003A0100 3801344
    Address of element from position 0 003A0100 3801344
    Address of element from position 1 003A0104 3801348
    Address of element from position 2 003A0108 3801352
    Address of element from position 3 003A010C 3801356
    Address of element from position 4 003A0110 3801360
    Address of element from position 5 003A0114 3801364
-----
Address of the Dynamic Array structure: 00D3FE00 13893120
Length is: 8 si capacitate: 12
Address of array from DA: 0039E210 3760656
    Address of element from position 0 0039E210 3760656
    Address of element from position 1 0039E214 3760660
    Address of element from position 2 0039E218 3760664
    Address of element from position 3 0039E21C 3760668
    Address of element from position 4 0039E220 3760672
    Address of element from position 5 0039E224 3760676
    Address of element from position 6 0039E228 3760680
    Address of element from position 7 0039E22C 3760684
```

# Dynamic Array - delete operation

- To delete an element from a given position  $i$ , the elements after position  $i$  need to be moved one position to the left (element from position  $j$  is moved to position  $j-1$ ).



- capacity (cap): 10
- nrElem: 5
- Delete the element from position 3

# Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size -

# Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size -  $\Theta(1)$
  - getElement -

# Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size -  $\Theta(1)$
  - getElement -  $\Theta(1)$
  - setElement -

# Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size -  $\Theta(1)$
  - getElement -  $\Theta(1)$
  - setElement -  $\Theta(1)$
  - iterator -

# Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size -  $\Theta(1)$
  - getElement -  $\Theta(1)$
  - setElement -  $\Theta(1)$
  - iterator -  $\Theta(1)$
  - addToPosition -

# Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size -  $\Theta(1)$
  - getElement -  $\Theta(1)$
  - setElement -  $\Theta(1)$
  - iterator -  $\Theta(1)$
  - addToPosition -  $O(n)$
  - deleteFromEnd -

# Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size -  $\Theta(1)$
  - getElement -  $\Theta(1)$
  - setElement -  $\Theta(1)$
  - iterator -  $\Theta(1)$
  - addToPosition -  $O(n)$
  - deleteFromEnd -  $\Theta(1)$
  - deleteFromPosition -

# Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size -  $\Theta(1)$
  - getElement -  $\Theta(1)$
  - setElement -  $\Theta(1)$
  - iterator -  $\Theta(1)$
  - addToPosition -  $O(n)$
  - deleteFromEnd -  $\Theta(1)$
  - deleteFromPosition -  $O(n)$
  - addToEnd -

# Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size -  $\Theta(1)$
  - getElement -  $\Theta(1)$
  - setElement -  $\Theta(1)$
  - iterator -  $\Theta(1)$
  - addToPosition -  $O(n)$
  - deleteFromEnd -  $\Theta(1)$
  - deleteFromPosition -  $O(n)$
  - addToEnd -  $\Theta(1)$  amortized

# Amortized analysis

- In *asymptotic* time complexity analysis we consider one single run of an algorithm.
  - *addToEnd* should have complexity  $O(n)$  - when we have to resize the array, we need to move every existing element, so the number of instructions is proportional to the length of the array.
  - Consequently, a sequence of  $n$  calls to the *addToEnd* operation would have complexity  $O(n^2)$ .

# Amortized analysis

- In *asymptotic* time complexity analysis we consider one single run of an algorithm.
  - *addToEnd* should have complexity  $O(n)$  - when we have to resize the array, we need to move every existing element, so the number of instructions is proportional to the length of the array.
  - Consequently, a sequence of  $n$  calls to the *addToEnd* operation would have complexity  $O(n^2)$ .
- In *amortized* time complexity analysis we consider a sequence of operations and compute the average time for these operations.
  - In amortized time complexity analysis we will consider the total complexity of  $n$  calls to the *addToEnd* operation and divide this by  $n$ , to get the *amortized* complexity of the algorithm.

# Amortized analysis

- We can observe that if we consider a sequence of  $n$  operations, we rarely have to resize the array
- Consider  $c_i$  the cost ( $\approx$  number of instructions) for the  $i^{th}$  call to `addToEnd`
- Considering that we double the capacity at each resize operation, at the  $i^{th}$  operation we perform a resize if  $i-1$  is a power of 2. So, the cost of operation  $i$ ,  $c_i$ , is:

$$c_i = \begin{cases} i, & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

# Amortized analysis

- Cost of  $n$  operations is:

$$\sum_{i=1}^n c_i \leq n + \sum_{j=0}^{[\log_2 n]} 2^j < n + 2n = 3n$$

- The sum contains at most  $n$  values of 1 (this is where the  $n$  term comes from) and at most (integer part of)  $\log_2 n$  terms of the form  $2^j$ .
- Since the total cost of  $n$  operations is  $3n$ , we can say that the cost of one operation is 3, which is constant.

# Amortized analysis

- While the worst case time complexity of *addToEnd* is still  $O(n)$ , the amortized complexity is  $\Theta(1)$ .
- The amortized complexity is no longer valid, if the resize operation just adds a constant number of new slots.
- In case of the *addToPosition* operation, both the worst case and the amortized complexity of the operation is  $O(n)$  - even if resize is performed rarely, we need to move elements to empty the position where we put the new element.

# When do we have amortized complexity?

- The reason why in case of *addToEnd* we can talk about amortized complexity is that the worst case situation (the resize) happens rarely.
- Whenever you have an algorithm and you want to determine whether amortized complexity computation is applicable, ask the following questions:
  - Can I have worst case complexity for two calls in a row (one after the another)?
  - If the answer is YES, than you do not have a situation of amortized complexity computation. (If the answer is NO, it is still not sure that you do have amortized complexity, but if it is YES, you definitely do not have amortized complexity.)

- In order to avoid having a Dynamic Array with too many empty slots, we can resize the array after deletion as well, if the array becomes "too empty".
- How empty should the array become before resize? Which of the following two strategies do you think is better? Why?
  - Wait until the table is only half full ( $da.nrElem \approx da.cap/2$ ) and resize it to the half of its capacity
  - Wait until the table is only a quarter full ( $da.nrElem \approx da.cap/4$ ) and resize it to the half of its capacity

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 3

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

# In Lecture 2...

- Algorithm analysis - recursive algorithms
- Dynamic array
- Amortized algorithm analysis

- Iterators
- Containers

- In order to avoid having a Dynamic Array with too many empty slots, we can resize the array after deletion as well, if the array becomes "too empty".
- How empty should the array become before resize? Which of the following two strategies do you think is better? Why?
  - Wait until the table is only half full ( $da.nrElem \approx da.cap/2$ ) and resize it to the half of its capacity
  - Wait until the table is only a quarter full ( $da.nrElem \approx da.cap/4$ ) and resize it to the half of its capacity

- An *iterator* is an abstract data type that is used to iterate through the elements of a container.
- Containers can be represented in different ways, using different data structures. Iterators are used to offer a common and generic way of moving through all the elements of a container, independently of the representation of the container.
- Every container that can be iterated, has to contain in the interface an operation called *iterator* that will create and return an iterator over the container.

- An iterator usually contains:
  - a reference to the container it iterates over
  - a reference to a *current element* from the container
- Iterating through the elements of the container means actually moving this *current element* from one element to another until the iterator becomes *invalid*
- The exact way of representing the *current element* from the iterator depends on the data structure used for the implementation of the container. If the representation/implementation of the container changes, we need to change the representation/ implementation of the iterator as well.

- **Domain of an Iterator**

$\mathcal{I} = \{\text{it} | \text{it is an iterator over a container with elements of type TElem}\}$

- **Interface** of an Iterator:

- **init(it, c)**
  - **description:** creates a new iterator for a container
  - **pre:**  $c$  is a container
  - **post:**  $it \in \mathcal{I}$  and  $it$  points to the first element in  $c$  if  $c$  is not empty or  $it$  is not valid

- **getCurrent(it)**

- **description:** returns the current element from the iterator
- **pre:**  $it \in \mathcal{I}$ ,  $it$  is valid
- **post:**  $\text{getCurrent} \leftarrow e$ ,  $e \in TElm$ ,  $e$  is the current element from  $it$
- **throws:** an exception if the iterator is not valid

- `next(it)`
  - **description:** moves the current element from the container to the next element or makes the iterator invalid if no elements are left
  - **pre:**  $it \in \mathcal{I}$ ,  $it$  is valid
  - **post:**  $it' \in \mathcal{I}$ , the current element from  $it'$  points to the next element from the container or  $it'$  is invalid if no more elements are left
  - **throws:** an exception if the iterator is not valid

- **valid(it)**

- **description:** verifies if the iterator is valid
- **pre:**  $it \in \mathcal{I}$
- **post:**

$$valid \leftarrow \begin{cases} True, & \text{if it points to a valid element from the container} \\ False & \text{otherwise} \end{cases}$$

- **first(it)**

- **description:** sets the current element from the iterator to the first element of the container
- **pre:**  $it \in \mathcal{I}$
- **post:**  $it' \in \mathcal{I}$ , the current element from  $it'$  points to the first element of the container if it is not empty, or  $it'$  is invalid

# Types of iterators I

- The interface presented above describes the simplest iterator: *unidirectional* and *read-only*
- A *unidirectional* iterator can be used to iterate through a container in one direction only (usually *forward*, but we can define a *reverse* iterator as well).
- A *bidirectional* iterator can be used to iterate in both directions. Besides the *next* operation it has an operation called *previous* and it could also have a *last* operation (the pair of *first*).

## Types of iterators II

- A *random access* iterator can be used to move multiple steps (not just one step forward or one step backward).
- A *read-only* iterator can be used to iterate through the container, but cannot be used to change it.
- A *read-write* iterator can be used to add/delete elements to/from the container.

# Using the iterator

- Since the interface of an iterator is the same, independently of the exact container or its representation, the following subalgorithm can be used to print 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

getCurrent(*it*, *elem*)

**print** *elem*

//go to the next element

next(*it*)

**end-while**

**end-subalgorithm**

# Iterator for a Dynamic Array

- How can we define an iterator for a Dynamic Array?
- How can we represent that *current element* from the iterator?

# Iterator for a Dynamic Array

- How can we define an iterator for a Dynamic Array?
- How can we represent that *current element* from the iterator?
- In case of a Dynamic Array, the simplest way to represent a *current element* is to retain the position of the *current element*.

IteratorDA:

da: DynamicArray

current: Integer

- Let's see how the operations of the iterator can be implemented.

# Iterator for a Dynamic Array - init

- What do we need to do in the *init* operation?

- What do we need to do in the *init* operation?

**subalgorithm** init(it, da) *is:*

//*it is an IteratorDA, da is a Dynamic Array*

it.da  $\leftarrow$  da

it.current  $\leftarrow$  1

**end-subalgorithm**

- Complexity:

- What do we need to do in the *init* operation?

**subalgorithm** init(it, da) *is:*

//*it is an IteratorDA, da is a Dynamic Array*

it.da  $\leftarrow$  da

it.current  $\leftarrow$  1

**end-subalgorithm**

- Complexity:  $\Theta(1)$

# Iterator for a Dynamic Array - `getCurrent`

- What do we need to do in the `getCurrent` operation?

# Iterator for a Dynamic Array - `getCurrent`

- What do we need to do in the `getCurrent` operation?

```
function getCurrent(it) is:  
    if it.current > it.da.nrElem then  
        @throw exception  
    end-if  
    getCurrent ← it.da.elems[it.current]  
end-function
```

- Complexity:

# Iterator for a Dynamic Array - `getCurrent`

- What do we need to do in the `getCurrent` operation?

```
function getCurrent(it) is:  
    if it.current > it.da.nrElem then  
        @throw exception  
    end-if  
    getCurrent ← it.da.elems[it.current]  
end-function
```

- Complexity:  $\Theta(1)$

# Iterator for a Dynamic Array - next

- What do we need to do in the *next* operation?

- What do we need to do in the *next* operation?

**subalgorithm** next(it) *is:*

**if** it.current > it.da.nrElem **then**

    @throw exception

**end-if**

    it.current  $\leftarrow$  it.current + 1

**end-subalgorithm**

- Complexity:

- What do we need to do in the *next* operation?

**subalgorithm** next(it) *is:*

**if** it.current > it.da.nrElem **then**

    @throw exception

**end-if**

    it.current  $\leftarrow$  it.current + 1

**end-subalgorithm**

- Complexity:  $\Theta(1)$

# Iterator for a Dynamic Array - valid

- What do we need to do in the *valid* operation?

- What do we need to do in the *valid* operation?

```
function valid(it) is:
  if it.current <= it.da.nrElem then
    valid ← True
  else
    valid ← False
  end-if
end-function
```

- Complexity:

- What do we need to do in the *valid* operation?

```
function valid(it) is:
    if it.current <= it.da.nrElem then
        valid ← True
    else
        valid ← False
    end-if
end-function
```

- Complexity:  $\Theta(1)$

# Iterator for a Dynamic Array - first

- What do we need to do in the *first* operation?

# Iterator for a Dynamic Array - first

- What do we need to do in the *first* operation?

**subalgorithm** first(it) *is*:

    it.current  $\leftarrow 1$

**end-subalgorithm**

- Complexity:  $\Theta(1)$

# Iterator for a Dynamic Array

- We can print the content of a Dynamic Array in two ways:
  - Using an iterator (as present above for a container)
  - Using the positions (indexes) of elements

# Print with Iterator

```
subalgorithm printDAWithIterator(da) is:
    //pre: da is a DynamicArray
    //we create an iterator using the iterator method of DA
    iterator(da, 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
```

- What is the complexity of *printDAWithIterator*?

# Print with indexes

**subalgorithm** printDAWithIndexes(da) **is:**

//pre: da is a Dynamic Array  
**for** i  $\leftarrow$  1, size(da) **execute**  
    elem  $\leftarrow$  getElement(da, i)  
    **print** elem  
**end-for**  
**end-subalgorithm**

- What is the complexity of *printDAWithIndexes*?

# Iterator for a Dynamic Array

- In case of a Dynamic Array both printing algorithms have  $\Theta(n)$  complexity
- For other data structures/containers we need iterator because
  - there are no positions in the data structure/container
  - the time complexity of iterating through all the elements is smaller

- There are many different containers, based on different properties:
  - do the elements have to be unique?
  - do the elements have positions assigned?
  - can we access any element or just some specific ones?
  - do we have simple elements, or key-value pairs?

- The ADT Bag is a container in which the elements are not unique and they do not have positions.
- Interface of the Bag was discussed at Seminar 1.

- A Bag can be represented using several data structures, one of them being a dynamic array (others will be discussed later)
- Independently of the chosen data structure, there are two options for storing the elements:
  - Store separately every element that was added (R1)
  - Store each element only once and keep a frequency count for it. (R2)

# ADT Bag - R1 example

- Assume a dynamic array as data structure for the representation (but the idea is applicable for other representations as well)
- Assume that we have a Bag with the following numbers: 4, 1, 6, 4, 7, 2, 1, 1, 9
- In R1 the Bag looks in the following way:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	6	4	7	2	1	1	1	9						

- Add element -5

# ADT Bag - R1 example

- Assume a dynamic array as data structure for the representation (but the idea is applicable for other representations as well)
- Assume that we have a Bag with the following numbers: 4, 1, 6, 4, 7, 2, 1, 1, 9
- In R1 the Bag looks in the following way:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	6	4	7	2	1	1	1	9						

- Add element -5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	6	4	7	2	1	1	1	9	-5					

- Remove element 6

# ADT Bag - R1 example

- Assume a dynamic array as data structure for the representation (but the idea is applicable for other representations as well)
- Assume that we have a Bag with the following numbers: 4, 1, 6, 4, 7, 2, 1, 1, 9
- In R1 the Bag looks in the following way:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	6	4	7	2	1	1	1	9						

- Add element -5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	6	4	7	2	1	1	1	9	-5					

- Remove element 6

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	-5	4	7	2	1	1	1	9						

# ADT Bag - R2 example

- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R2 the Bag looks in the following way:

	1	2	3	4	5	6	7	8	9
elems	4	1	6	7	2	9			
freq	2	4	1	1	1	1			

- Add element -5

# ADT Bag - R2 example

- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R2 the Bag looks in the following way:

	1	2	3	4	5	6	7	8	9
elems	4	1	6	7	2	9			
freq	2	4	1	1	1	1			

- Add element -5

	1	2	3	4	5	6	7	8	9
elems	4	1	6	7	2	9	-5		
freq	2	4	1	1	1	1	1		

- Add element 7

# ADT Bag - R2 example

- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R2 the Bag looks in the following way:

	1	2	3	4	5	6	7	8	9
elems	4	1	6	7	2	9			
freq	2	4	1	1	1	1			

- Add element -5

	1	2	3	4	5	6	7	8	9
elems	4	1	6	7	2	9	-5		
freq	2	4	1	1	1	1	1		

- Add element 7

	1	2	3	4	5	6	7	8	9
elems	4	1	6	7	2	9	-5		
freq	2	4	1	2	1	1	1		

# ADT Bag - R2 example

- Remove element 6

# ADT Bag - R2 example

- Remove element 6

	1	2	3	4	5	6	7	8	9
elems	4	1	-5	7	2	9			
freq	2	4	1	2	1	1			

- Remove element 1

# ADT Bag - R2 example

- Remove element 6

	1	2	3	4	5	6	7	8	9
elems	4	1	-5	7	2	9			
freq	2	4	1	2	1	1			

- Remove element 1

	1	2	3	4	5	6	7	8	9
elems	4	1	-5	7	2	9			
freq	2	3	1	2	1	1			

- Besides the two representations presented above which can be used for other data structures as well, there are two other possible representations which are specific for dynamic arrays.

- Another representation would be to store the unique elements in a dynamic array and store separately the positions from this array for every element that appears in the Bag (R3).
- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R3 the Bag looks in the following way (assume 1-based indexing):

1	2	3	4	5	6	7	8	9
4	1	6	7	2	9			

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	1	4	5	2	2	2	6				

# ADT Bag - R3 example

- Add element -5

# ADT Bag - R3 example

- Add element -5

1	2	3	4	5	6	7	8	9
4	1	6	7	2	9	-5		

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	1	4	5	2	2	2	6	7			

- Add element 7

# ADT Bag - R3 example

- Add element -5

1	2	3	4	5	6	7	8	9
4	1	6	7	2	9	-5		

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	1	4	5	2	2	2	6	7			

- Add element 7

1	2	3	4	5	6	7	8	9
4	1	6	7	2	9	-5		

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	1	4	5	2	2	2	6	7	4		

# ADT Bag - R3 example

- Remove element 6

# ADT Bag - R3 example

- Remove element 6

1	2	3	4	5	6	7	8	9
4	1	-5	7	2	9			

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	4	1	4	5	2	2	2	6	3			

- Remove element 1

# ADT Bag - R3 example

- Remove element 6

1	2	3	4	5	6	7	8	9
4	1	-5	7	2	9			

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	4	1	4	5	2	2	2	6	3			

- Remove element 1

1	2	3	4	5	6	7	8	9
4	1	-5	7	2	9			

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	4	1	4	5	2	2	2	6				

- If the elements of the Bag are integer numbers (and a dynamic array is used for storing them), another representation is possible, where the positions of the array represent the elements and the value from the position is the frequency of the element. Thus, the frequency of the minimum element is at position 1 (assume 1-based indexing).
- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R4 the Bag looks in the following way:

1	2	3	4	5	6	7	8	9	10	11
4	1	0	2	0	1	1	0	1		

Minimum element: 1

# ADT Bag - R4 example

- Add element -5

## ADT Bag - R4 example

- Add element -5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	1	0	1		

Minimum element: -5

- When indexing starts from 1, the element in the dynamic array that is on position  $i$ , represents the actual value:  
 $minimum + i - 1 \Rightarrow$  position of an element  $e$  is  
 $e - minimum + 1$
- Add element 7

# ADT Bag - R4 example

- Add element -5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	1	0	1		

Minimum element: -5

- When indexing starts from 1, the element in the dynamic array that is on position  $i$ , represents the actual value:  
 $minimum + i - 1 \Rightarrow$  position of an element  $e$  is  
 $e - minimum + 1$
- Add element 7

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	2	0	1		

Minimum element: -5

# ADT Bag - R4 example

- Remove element 6

# ADT Bag - R4 example

- Remove element 6

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	0	2	0	1		

Minimum element: -5

- Remove element 1

# ADT Bag - R4 example

- Remove element 6

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	0	2	0	1		

Minimum element: -5

- Remove element 1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	3	1	0	2	0	0	2	0	1		

Minimum element: -5

- There are no positions in a Bag, but sometimes we need the elements to be sorted  $\Rightarrow$  ADT SortedBag.
- These were the operations in the interface of the ADT Bag:
  - $\text{init}(b)$
  - $\text{add}(b, e)$
  - $\text{remove}(b, e)$
  - $\text{search}(b, e)$
  - $\text{nrOfOccurrences}(b, e)$
  - $\text{size}(b)$
  - $\text{iterator}(b, it)$
  - $\text{destroy}$
- What should be different (new operations, removed operations, modified operations) in case of a *SortedBag*?

- The only modification in the interface is that the init operation receives a *relation* as parameter
- Domain of Sorted Bag:
  - $S\mathcal{B} = \{\mathbf{sb} \mid sb \text{ is a sorted bag that uses a relation to order the elements}\}$
- **init (sb, rel)**
  - **descr:** creates a new, empty sorted bag, where the elements will be ordered based on a relation
  - **pre:**  $rel \in Relation$
  - **post:**  $sb \in S\mathcal{B}$ ,  $sb$  is an empty sorted bag which uses the relation  $rel$

# The relation

- Usually there are two approaches, when we want to order elements:
  - Assume that they have a *natural ordering*, and use this ordering (for ex: alphabetical ordering for strings, ascending ordering for numbers, etc.).
  - Sometimes, we want to order the elements in a different way than the natural ordering (or there is no natural ordering)  $\Rightarrow$  we use a relation
  - A relation will be considered as a function with two parameters (the two elements that are compared) which returns *true* if they are in the correct order, or *false* if they should be reversed.

- While the other operations from the interface are the same for a Bag and an SortedBag, there is another difference between them:

- While the other operations from the interface are the same for a Bag and an SortedBag, there is another difference between them:
  - the iterator for a SortedBag has to return the elements in the order given by the relation.

- While the other operations from the interface are the same for a Bag and an SortedBag, there is another difference between them:
  - the iterator for a SortedBag has to return the elements in the order given by the relation.
  - Since the iterator operations should have a  $\Theta(1)$  complexity, this means that internally the elements have to be stored based on the relation.

- A SortedBag can be represented using several data structure, one of them being the dynamic array (others will be discussed later):
- Independently of the chosen data structure, there are two options for storing the elements:
  - Store separately every element that was added (in the order given by the relation)
  - Store each element only once (in the order given by the relation) and keep a frequency count for it

- Consider the following problem: *in order to avoid electoral fraud (especially the situation when someone votes multiple times in different locations) we want to build a software system which stores the personal numerical code (CNP) of everyone who votes.* What would be the characteristics of the container used to store these personal numerical codes?

- Consider the following problem: *in order to avoid electoral fraud (especially the situation when someone votes multiple times in different locations) we want to build a software system which stores the personal numerical code (CNP) of everyone who votes.* What would be the characteristics of the container used to store these personal numerical codes?
  - The elements have to be unique
  - The order of the elements is not important
- The container in which the elements have to be unique and the order of the elements is not important (there are no positions) is the **ADT Set**.

- Domain of the ADT Set:

$$\mathcal{S} = \{s | s \text{ is a set with elements of the type } \text{TElem}\}$$

- **init (s)**
  - **descr:** creates a new empty set
  - **pre:** true
  - **post:**  $s \in \mathcal{S}$ ,  $s$  is an empty set.

- **add( $s, e$ )**

- **descr:** adds a new element into the set if it is not already in the set
- **pre:**  $s \in \mathcal{S}$ ,  $e \in TElm$
- **post:**  $s' \in \mathcal{S}$ ,  $s' = s \cup \{e\}$  ( $e$  is added only if it is not in  $s$  yet. If  $s$  contains the element  $e$  already, no change is made).  
 $add \leftarrow \text{true}$  if  $e$  was added to the set,  $\text{false}$  otherwise.

- **remove(s, e)**
  - **descr:** removes an element from the set.
  - **pre:**  $s \in \mathcal{S}$ ,  $e \in TElm$
  - **post:**  $s \in \mathcal{S}$ ,  $s' = s \setminus \{e\}$  (if  $e$  is not in  $s$ ,  $s$  is not changed).  
 $remove \leftarrow \text{true}$ , if  $e$  was removed,  $false$  otherwise

- **search(s, e)**

- **descr:** verifies if an element is in the set.
- **pre:**  $s \in \mathcal{S}$ ,  $e \in TElm$
- **post:**

$$search \leftarrow \begin{cases} True, & \text{if } e \in s \\ False, & \text{otherwise} \end{cases}$$

- **size( $s$ )**

- **descr:** returns the number of elements from a set
- **pre:**  $s \in \mathcal{S}$
- **post:** size  $\leftarrow$  the number of elements from  $s$

- **isEmpty(s)**

- **descr:** verifies if the set is empty
- **pre:**  $s \in \mathcal{S}$
- **post:**

$$\text{isEmpty} \leftarrow \begin{cases} \text{True}, & \text{if } s \text{ has no elements} \\ \text{False}, & \text{otherwise} \end{cases}$$

- **iterator(s, it)**
  - **descr:** returns an iterator for a set
  - **pre:**  $s \in \mathcal{S}$
  - **post:**  $it \in \mathcal{I}$ ,  $it$  is an iterator over the set  $s$

- **destroy (s)**
  - **descr:** destroys a set
  - **pre:**  $s \in S$
  - **post:** the set  $s$  was destroyed.

- Other possible operations (characteristic for sets from mathematics):
  - reunion of two sets
  - intersection of two sets
  - difference of two sets (elements that are present in the first set, but not in the second one)

# ADT Set - representation

- If a Dynamic Array is used as data structure and the elements of the set are numbers, we can choose a representation in which the elements are represented by the positions in the dynamic array and a boolean value from that position shows if the element is in the set or not.
- Assume a Set with the following numbers: 4, 2, 10, 7, 6.
- This Set would be represented in the following way (the formulae discussed at Bag can be applied here as well):

1	2	3	4	5	6	7	8	9
T	F	T	F	T	T	F	F	T

Minimum element: 2

# ADT Set - representation

- Add element -3

- Add element -3

1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	F	F	F	F	T	F	T	F	T	T	F	F	T

Minimum element: -3

- Remove element 10

# ADT Set - representation

- Add element -3

1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	F	F	F	F	T	F	T	F	T	T	F	F	T

Minimum element: -3

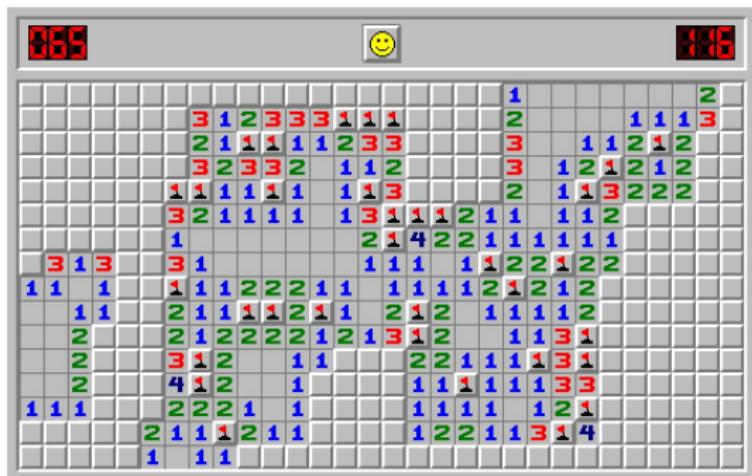
- Remove element 10

1	2	3	4	5	6	7	8	9	10	11
T	F	F	F	F	T	F	T	F	T	T

Minimum element: -3

- We can have a Set where the elements are ordered based on a *relation*  $\Rightarrow$  *SortedSet*.
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted set, the iterator has to iterate through the elements in the order given by the *relation*, so we need to keep them ordered in the representation.

- Imagine that you wanted to implement this game:



Source: <http://minesweeperonline.com/#>

- What would be the specifics of the container needed to store the game board (location of the mines)?

- The **ADT Matrix** is a container that represents a two-dimensional array.
- Each element has a unique position, determined by two indexes: its line and column.
- The domain of the ADT Matrix:  $\mathcal{MAT} = \{mat | mat \text{ is a matrix with elements of the type } T\text{Elem}\}$
- What operations should we have for a Matrix?

- **init(mat, nrL, nrC)**
  - **descr:** creates a new matrix with a given number of lines and columns
  - **pre:**  $nrL \in N^*$  and  $nrC \in N^*$
  - **post:**  $mat \in \mathcal{MAT}$ ,  $mat$  is a matrix with  $nrL$  lines and  $nrC$  columns
  - **throws:** an exception if  $nrL$  or  $nrC$  is negative or zero

- **nrLines(mat)**
  - **descr:** returns the number of lines of the matrix
  - **pre:**  $mat \in MAT$
  - **post:**  $nrLines \leftarrow$  returns the number of lines from *mat*

- **nrCols(mat)**
  - **descr:** returns the number of columns of the matrix
  - **pre:**  $mat \in MAT$
  - **post:**  $nrCols \leftarrow$  returns the number of columns from  $mat$

- **element(mat, i, j)**
  - **descr:** returns the element from a given position from the matrix (assume 1-based indexing)
  - **pre:**  $mat \in MAT$ ,  $1 \leq i \leq nrLines$ ,  $1 \leq j \leq nrColumns$
  - **post:**  $element \leftarrow$  the element from line  $i$  and column  $j$
  - **throws:** an exception if the position  $(i,j)$  is not valid (less than 1 or greater than  $nrLines/nrColumns$ )

- `modify(mat, i, j, val)`
  - **descr:** sets the element from a given position to a given value (assume 1-based indexing)
  - **pre:**  $mat \in \mathcal{MAT}$ ,  $1 \leq i \leq nrLines$ ,  $1 \leq j \leq nrColumns$ ,  $val \in TElm$
  - **post:** the value from position  $(i, j)$  is set to  $val$ .  $modify \leftarrow$  the old value from position  $(i, j)$
  - **throws:** an exception if position  $(i, j)$  is not valid (less than 1 or greater than  $nrLine/nrColumns$ )

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 4

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

# In Lecture 3...

- Iterators
- Containers
  - ADT Bag
  - ADT Sorted Bag
  - ADT Set and ADT SortedSet

- Containers

- The **ADT Matrix** is a container that represents a two-dimensional array.
- Each element has a unique position, determined by two indexes: its line and column.
- The domain of the ADT Matrix:  $\mathcal{MAT} = \{mat | mat \text{ is a matrix with elements of the type } TElem\}$
- What operations should we have for a Matrix?

- **init(mat, nrL, nrC)**
  - **descr:** creates a new matrix with a given number of lines and columns
  - **pre:**  $nrL \in N^*$  and  $nrC \in N^*$
  - **post:**  $mat \in \mathcal{MAT}$ ,  $mat$  is a matrix with  $nrL$  lines and  $nrC$  columns
  - **throws:** an exception if  $nrL$  or  $nrC$  is negative or zero

- **nrLines(mat)**
  - **descr:** returns the number of lines of the matrix
  - **pre:**  $mat \in MAT$
  - **post:**  $nrLines \leftarrow$  returns the number of lines from *mat*

- $\text{nrCols}(\text{mat})$ 
  - **descr:** returns the number of columns of the matrix
  - **pre:**  $\text{mat} \in \text{MAT}$
  - **post:**  $\text{nrCols} \leftarrow$  returns the number of columns from  $\text{mat}$

- **element(mat, i, j)**
  - **descr:** returns the element from a given position from the matrix (assume 1-based indexing)
  - **pre:**  $mat \in MAT$ ,  $1 \leq i \leq nrLines$ ,  $1 \leq j \leq nrColumns$
  - **post:**  $element \leftarrow$  the element from line  $i$  and column  $j$
  - **throws:** an exception if the position  $(i,j)$  is not valid (less than 1 or greater than  $nrLines/nrColumns$ )

- `modify(mat, i, j, val)`
  - **descr:** sets the element from a given position to a given value (assume 1-based indexing)
  - **pre:**  $mat \in \mathcal{MAT}$ ,  $1 \leq i \leq nrLines$ ,  $1 \leq j \leq nrColumns$ ,  $val \in TElm$
  - **post:** the value from position  $(i, j)$  is set to  $val$ .  $modify \leftarrow$  the old value from position  $(i, j)$
  - **throws:** an exception if position  $(i, j)$  is not valid (less than 1 or greater than  $nrLine/nrColumns$ )

- Other possible operations:
  - get the (first) position of a given element
  - create an iterator that goes through the elements by columns
  - create an iterator that goes through the elements by lines
  - etc.

- Usually a sequential representation is used for a Matrix (we memorize all the lines one after the other in a consecutive memory block).
- If this sequential representation is used, for a matrix with  $N$  lines and  $M$  columns, the element from position  $(i, j)$  can be found at the memory address:  
address of element from position  $(i, j) = \text{address of the matrix} + (i * M + j) * \text{size of an element}$
- The above formula works for 0-based indexing, but can be adapted to 1-based indexing as well.

```
Size of int: 4
Address of matrix (5 rows, 8 cols): 6224024
Address of element 0, 0: 6224024
Address of element 2, 4: 6224104
Address of element 2, 5: 6224108
Address of element 2, 6: 6224112
Address of element 2, 7: 6224116
Address of element 3, 0: 6224120
Address of element 3, 4: 6224136
Address of element 4, 7: 6224180
```

- In the Minesweeper game example above we have a matrix with 480 elements ( $16 * 30$ ) but only 99 bombs.
- If the Matrix contains many values of 0 (or  $0_{TElem}$ ), we have a *sparse matrix*, where it is more (space) efficient to memorize only the elements that are different from 0.

# Sparse Matrix Example

0	33	0	100	1	0	0	9
2	0	2	0	2	0	7	0
0	4	0	0	3	0	0	0
17	0	0	10	0	16	0	7
0	0	0	0	0	0	0	0
0	1	0	13	0	8	0	29

- Number of lines: 6
- Number of columns: 8

# Sparse Matrix - R1

- We can memorize (line, column, value) triples, where value is different from 0 (or  $0_{TElem}$ ). For efficiency, we memorize the elements sorted by the (line, column) pairs (if the lines are different we order by line, if they are equal we order by column) - R1.
- Triples can be stored in a dynamic array or other data structures (will be discussed later):

# Sparse Matrix - R1 example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line	1	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- In an ADT Matrix, there is no operation to add an element or to remove an element. In the interface we only have the *modify* operation which changes a value from a position. If we represent the matrix as a sparse matrix, the *modify* operation might add or remove an element to/from the underlying data structure. But the operation from the interface is still called *modify*.

- When we have a Sparse Matrix (i.e., we keep only the values different from 0), for the modify operation we have four different cases, based on the value of the element currently at the given position (let's call it *current\_value*) and the new value that we want to put on that position (let's call it *new\_value*).
  - current\_value* = 0 and *new\_value* = 0  $\Rightarrow$  do nothing
  - current\_value* = 0 and *new\_value*  $\neq$  0  $\Rightarrow$  insert in the data structure
  - current\_value*  $\neq$  0 and *new\_value* = 0  $\Rightarrow$  remove from the data structure
  - current\_value*  $\neq$  0 and *new\_value*  $\neq$  0  $\Rightarrow$  just change the value in the data structure

# Sparse Matrix - R1 example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line	1	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (1, 5) to 0

# Sparse Matrix - R1 example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line	1	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (1, 5) to 0

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Line	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (3, 3) to 19

# Sparse Matrix - R1 example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line	1	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (1, 5) to 0

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Line	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (3, 3) to 19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line	1	1	1	2	2	2	2	3	3	3	4	4	4	4	6	6	6	6
Col	2	4	8	1	3	5	7	2	3	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	19	3	17	10	16	7	1	13	8	29

- We can see that in the previous representation there are many consecutive elements which have the same value in the line array. The array containing this information could be compressed, in the following way:
  - Keep the *Col* and *Value* arrays as in the previous representation.
  - For the lines, have an array of number of lines + 1 element, in which at position  $i$  we have the position from the *Col* array where the sequence of elements from line  $i$  begins.
  - Thus, elements from line  $i$  are in the *Col* and *Value* arrays between the positions  $[Line[i], Line[i+1])$ .
- This is called **compressed sparse line representation**.
- Obs:** In order for this representation to work, in the *Col* and *Value* arrays the elements have to be stored by rows (first elements of the first row, then elements of second row, etc.)

# Sparse Matrix - R2 example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Lines	1	5	9	11	15	15	19											
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

# Sparse Matrix - R2 example

- Modify the value from position (1, 5) to 0

# Sparse Matrix - R2 example

- Modify the value from position (1, 5) to 0
  - First we look for element on position (1,5).
  - Elements from line 1 are between positions 1 and 4 (inclusive)
  - Since we have there an item with column 5, we found our element
  - Setting to 0, means removing from *Col* and *Value* array.
  - In *Lines* array just the values change, not the size of the array.

	1	2	3	4	5	6	7
Lines	1	4	8	10	14	14	18
Col	2	4	8	1	3	5	7
Value	33	100	9	2	2	7	4

Diagram illustrating the modification of the matrix. Red arrows point from the 'Lines' array to the 'Col' and 'Value' arrays at index 4, indicating the element (1,5) is being modified.

# Sparse Matrix - R2 example

- Modify the value from position (3, 3) to 19

# Sparse Matrix - R2 example

- Modify the value from position (3, 3) to 19
  - First we look for element on position (3,3)
  - Elements from line 3 are between positions 8 and 9 (inclusive)
  - Since we have no column 3 there, at this position currently the value is 0. To set it to 19 we need to insert a new element in the *Col* and *Value* array.
  - In *Lines* array just the values change, not the size of the array

	1	2	3	4	5	6	7
Lines	1	4	8	11	15	15	19
Col	2	4	8	1	3	5	7
Value	33	100	9	2	2	7	4

Diagram illustrating the modification of the matrix. Red arrows point from the 'Lines' array to the 'Value' array at index 8, indicating the insertion of the value 19 at position (3,3).

- In a similar manner, we can define **compressed sparse column representation**:
  - We need two arrays *Lines* and *Values* for the non-zero elements, in which first the elements of the first column are stored, than elements from the second column, etc.
  - We need an array with  $\text{nrColumns} + 1$  elements, in which at position  $i$  we have the position from the *Lines* array where the sequence of elements from column  $i$  begins.
  - Thus, elements from column  $i$  are in the *Lines* and *Value* arrays between the positions  $[Col[i], Col[i+1]]$ .

# Sparse Matrix - R3 example

Cols	1	2	3	4	5	6	7	8	9
Lines	2	4	1	3	6	2	1	4	6
Value	2	17	33	4	1	2	100	10	13
Lines	1	2	3	4	5	6	7	8	9
Value	2	17	33	4	1	2	100	10	13

Diagram illustrating the mapping between the column index array and the matrix rows. Red arrows point from the column index values (1, 3, 6, 7, 10, 13, 15, 16, 19) to their corresponding positions in the matrix rows.



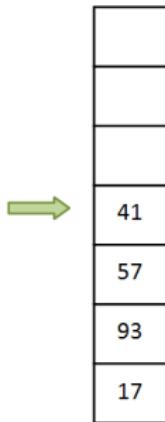
Source: <https://clipart.wpblink.com/wallpaper-1911442>

- Consider the above figure: if you had to add a new plate to the pile, where would you put it?
- If you had to remove a plate, which one would you take?

- 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: **Last In, First Out** (the last element that was added will be the first element that will be removed).

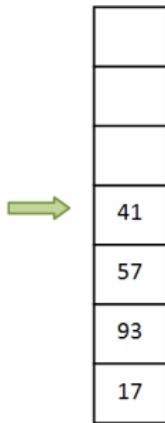
# ADT Stack Example

- Suppose that we have the following stack (green arrow shows the top of the stack):
- We *push* the number 33:

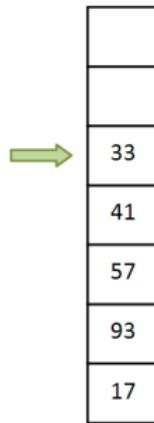


# ADT Stack Example

- Suppose that we have the following stack (green arrow shows the top of the stack):



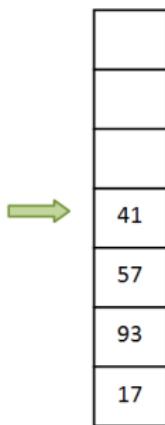
- We *push* the number 33:



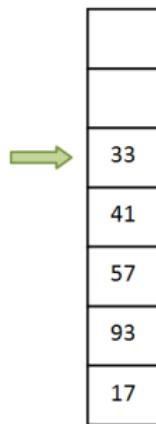
- We *pop* an element:

# ADT Stack Example

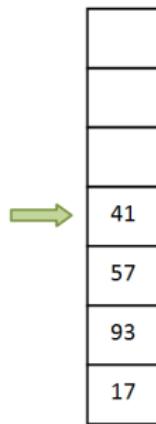
- Suppose that we have the following stack (green arrow shows the top of the stack):



- We *push* the number 33:

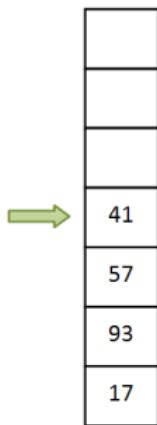


- We *pop* an element:



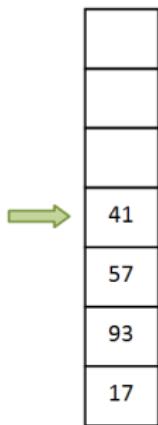
# ADT Stack Example

- This is our stack:
- We *pop* another element:

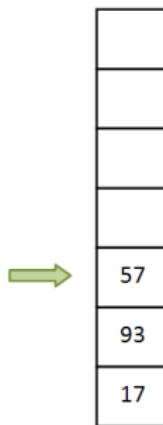


# ADT Stack Example

- This is our stack:



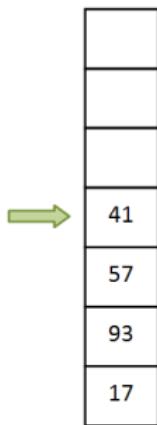
- We *pop* another element:



- We *push* the number 72:

# ADT Stack Example

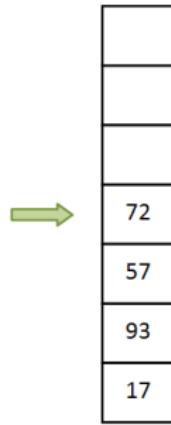
- This is our stack:



- We *pop* another element:



- We *push* the number 72:



- The domain of the ADT Stack:  
 $\mathcal{S} = \{s | s \text{ is a stack with elements of type } T\text{Elem}\}$
- The interface of the ADT Stack contains the following operations:

- **init(s)**
  - **descr:** creates a new empty stack
  - **pre:** True
  - **post:**  $s \in \mathcal{S}$ ,  $s$  is an empty stack

- **destroy( $s$ )**
  - **descr:** destroys a stack
  - **pre:**  $s \in \mathcal{S}$
  - **post:**  $s$  was destroyed

- **push(s, e)**
  - **descr:** pushes (adds) a new element onto the stack
  - **pre:**  $s \in \mathcal{S}$ ,  $e$  is a *TElem*
  - **post:**  $s' \in \mathcal{S}$ ,  $s' = s \oplus e$ ,  $e$  is the most recent element added to the stack

- **pop( $s$ )**

- **descr:** pops (removes) the most recent element from the stack
- **pre:**  $s \in \mathcal{S}$ ,  $s$  is not empty
- **post:**  $pop \leftarrow e$ ,  $e$  is a *TElem*,  $e$  is the most recent element from  $s$ ,  $s' \in \mathcal{S}$ ,  $s' = s \ominus e$
- **throws:** an *underflow* exception if the stack is empty

- **top(s)**

- **descr:** returns the most recent element from the stack (but it does not change the stack)
- **pre:**  $s \in \mathcal{S}$ ,  $s$  is not empty
- **post:**  $\text{top} \leftarrow e$ ,  $e$  is a  $TElem$ ,  $e$  is the most recent element from  $s$
- **throws:** an *underflow* exception if the stack is empty

- **isEmpty(s)**

- **descr:** checks if the stack is empty (has no elements)
- **pre:**  $s \in \mathcal{S}$
- **post:**

$$isEmpty \leftarrow \begin{cases} \text{true, if } s \text{ has no elements} \\ \text{false, otherwise} \end{cases}$$

- **Note:** stacks cannot be iterated, so they don't have an *iterator* operation!



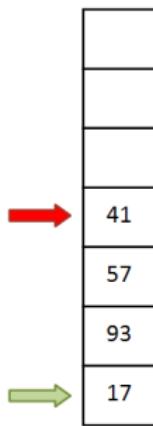
<http://www.rgbstock.com/photomeZ8AhAQueue+Line>

- Look at the queue above.
- If a new person arrives, where should he/she stand?
- When the blue person finishes, who is going to be the next at the desk?

- 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.

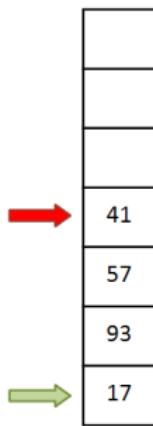
# ADT Queue - Example

- Assume that we have the following queue (green arrow is the front, red arrow is the rear)
- Push number 33:

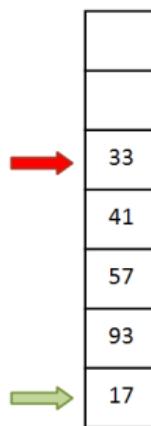


# ADT Queue - Example

- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



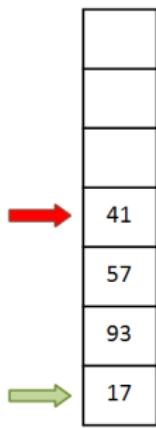
- Push number 33:



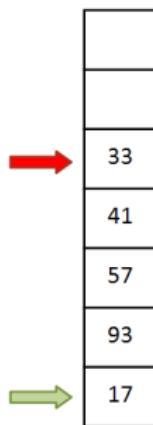
- Pop an element:

# ADT Queue - Example

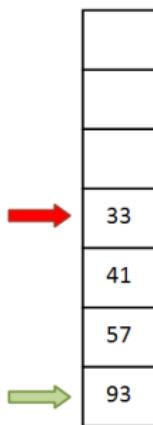
- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



- Push number 33:

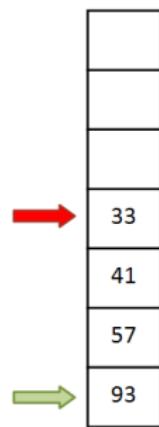


- Pop an element:



# ADT Queue - Example

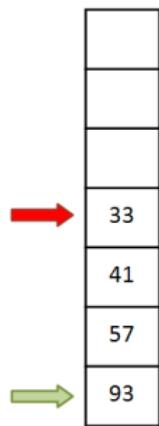
- This is our queue:



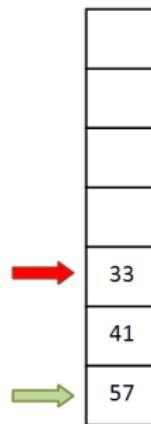
- Pop an element:

# ADT Queue - Example

- This is our queue:



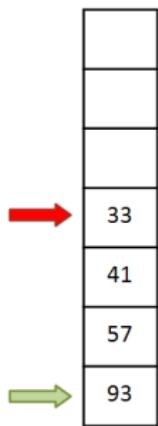
- Pop an element:



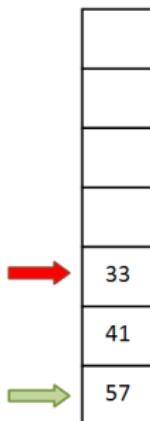
- Push number 72:

# ADT Queue - Example

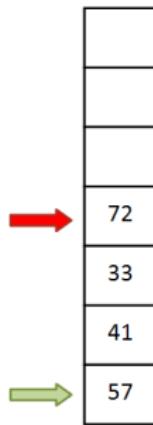
- This is our queue:



- Pop an element:



- Push number 72:



- The domain of the ADT Queue:  
 $\mathcal{Q} = \{q | q \text{ is a queue with elements of type } TElm\}$
- The interface of the ADT Queue contains the following operations:

- **init(q)**

- **descr:** creates a new empty queue
- **pre:** True
- **post:**  $q \in Q$ ,  $q$  is an empty queue

- **destroy(q)**
  - **descr:** destroys a queue
  - **pre:**  $q \in \mathcal{Q}$
  - **post:**  $q$  was destroyed

- **push(q, e)**
  - **descr:** pushes (adds) a new element to the rear of the queue
  - **pre:**  $q \in Q$ , e is a *TElem*
  - **post:**  $q' \in Q$ ,  $q' = q \oplus e$ , e is the element at the rear of the queue

- **pop( $q$ )**

- **descr:** pops (removes) the element from the front of the queue
- **pre:**  $q \in \mathcal{Q}$ ,  $q$  is not empty
- **post:**  $pop \leftarrow e$ ,  $e$  is a *TElem*,  $e$  is the element at the front of  $q$ ,  $q' \in \mathcal{Q}$ ,  $q' = q \ominus e$
- **throws:** an *underflow* exception if the queue is empty

- **top(q)**

- **descr:** returns the element from the front of the queue (but it does not change the queue)
- **pre:**  $q \in \mathcal{Q}$ ,  $q$  is not empty
- **post:**  $\text{top} \leftarrow e$ ,  $e$  is a *TElem*,  $e$  is the element from the front of  $q$
- **throws:** an *underflow* exception if the queue is empty

- `isEmpty(s)`

- **descr:** checks if the queue is empty (has no elements)
- **pre:**  $q \in \mathcal{Q}$
- **post:**

$$isEmpty \leftarrow \begin{cases} \text{true, if } q \text{ has no elements} \\ \text{false, otherwise} \end{cases}$$

- **Note:** queues cannot be iterated, so they do not have an *iterator* operation!

- What data structures can be used to implement a Queue?
  - Static Array - for a fixed capacity Queue
    - In this case an *isFull* operation can be added, and *push* can also throw an exception if the Queue is full.
  - Dynamic Array
  - other data structures (will be discussed later)

# ADT Queue - Array-based representation

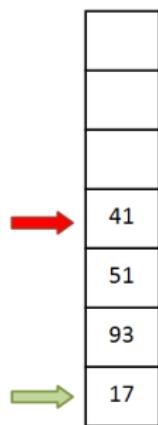
- If we want to implement a Queue using an array (static or dynamic), where should we place the *front* and the *rear* of the queue?

- If we want to implement a Queue using an array (static or dynamic), 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 array and *rear* at the end
  - Put *front* at the end of the array and *rear* at the beginning
- In either case we will have one operation (push or pop) that will have  $\Theta(n)$  complexity.

- We can improve the complexity of the operations, if we do not insist on having either *front* or *rear* at the beginning of the array (at position 1).

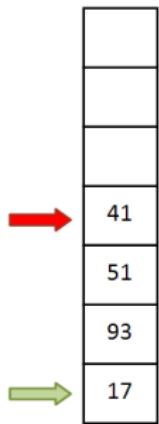
# ADT Queue - Array-based representation

- This is our queue  
(green arrow is the front, red arrow is the rear)
- Push number 33:

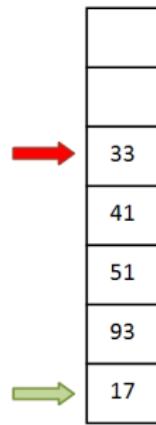


# ADT Queue - Array-based representation

- This is our queue  
(green arrow is the front, red arrow is the rear)



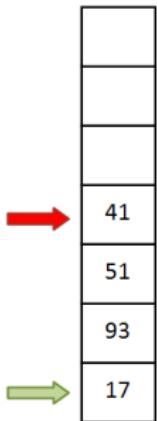
- Push number 33:



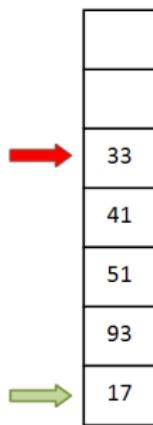
- Pop an element  
(and do not move the other elements):

# ADT Queue - Array-based representation

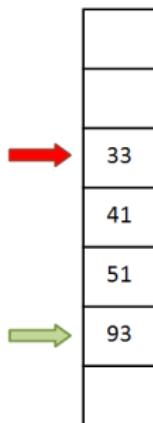
- This is our queue  
(green arrow is the front, red arrow is the rear)



- Push number 33:



- Pop an element  
(and do not move the other elements):



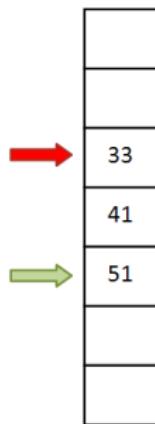
# ADT Queue - Array-based representation

- Pop another element:

# ADT Queue - Array-based representation

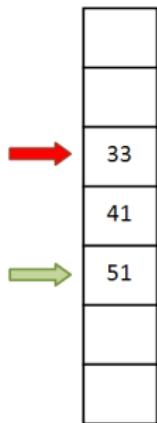
- Pop another element:

- Push number 11:

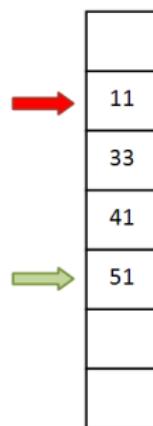


# ADT Queue - Array-based representation

- Pop another element:



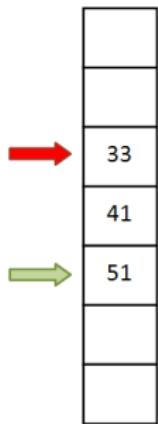
- Push number 11:



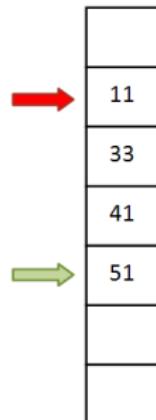
- Pop an element:

# ADT Queue - Array-based representation

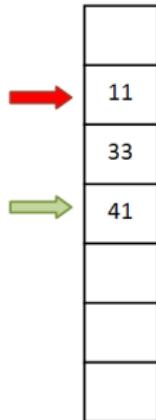
- Pop another element:



- Push number 11:



- Pop an element:

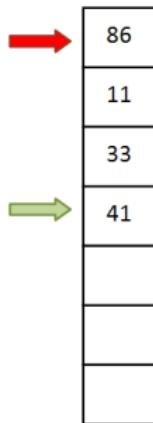


# ADT Queue - Array-based representation

- Push number 86:

# ADT Queue - Array-based representation

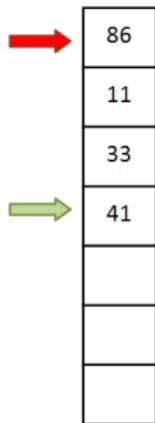
- Push number 86:



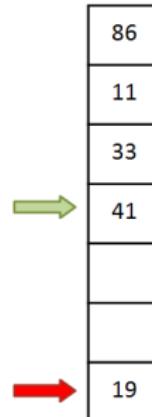
- Push number 19:

# ADT Queue - Array-based representation

- Push number 86:



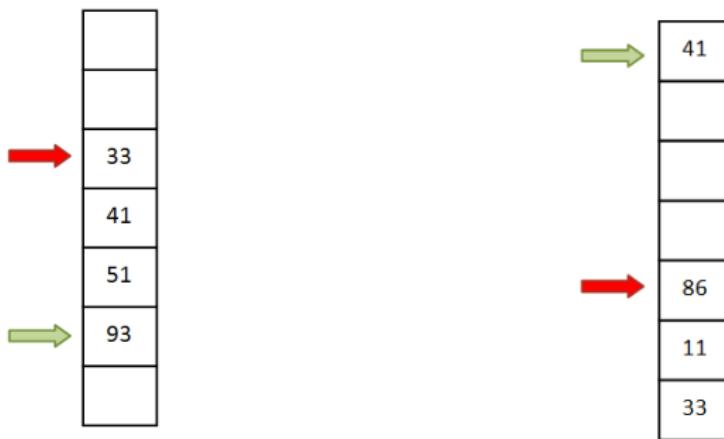
- Push number 19:



- This is called a **circular array**

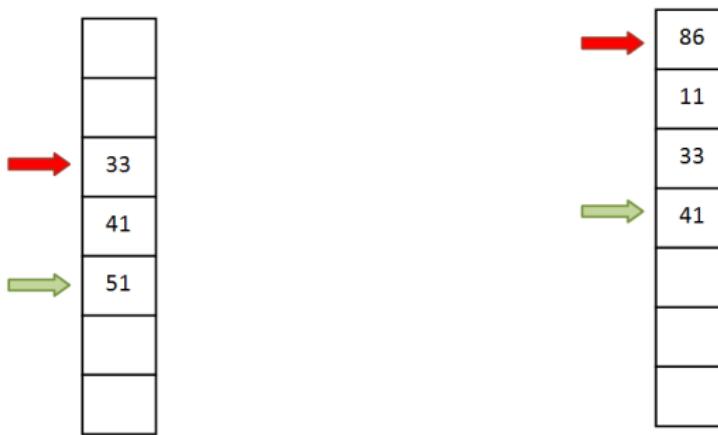
# ADT Queue - representation on a circular array - pop

- There are two situations for our queue (green arrow is the front where we pop from):



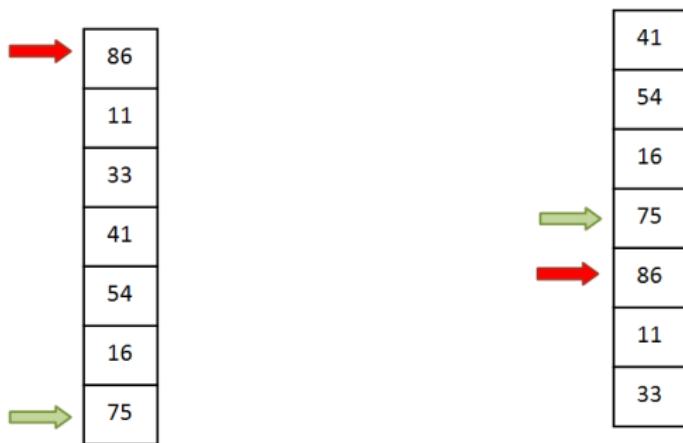
# ADT Queue - representation on a circular array - push

- There are two situations for our queue (red arrow is the end where we push):



## Queue - representation on a circular array - push

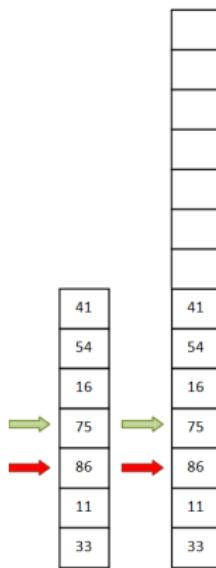
- When pushing a new element we have to check whether the queue is full



- For both example, the elements were added in the order: 75, 16, 54, 41, 33, 11, 86

# ADT Queue - representation on a circular array - push

- If we have a dynamic array-based representation and the array is full, we have to allocate a larger array and copy the existing elements (as we always do with dynamic arrays)
- But we have to be careful how we copy the elements in order to avoid having something like:





Source: <https://www.vectorstock.com/royalty-free-vector/patients-in-doctors-waiting-room-at-the-hospital-vector-12041494>

- Consider the following queue in front of the Emergency Room. Who should be the next person checked by the doctor?

# ADT Priority Queue

- 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.

- In order to work in a more general manner, we can define a relation  $\mathcal{R}$  on the set of priorities:  $\mathcal{R} : TPriority \times TPriority$
- When we say *the element with the highest priority* we will mean that the highest priority is determined using this relation  $\mathcal{R}$ .
- If the relation  $\mathcal{R} = " \geq "$ , the element with the *highest priority* is the one for which the value of the priority is the largest (maximum).
- Similarly, if the relation  $\mathcal{R} = " \leq "$ , the element with the *highest priority* is the one for which the value of the priority is the lowest (minimum).

# Priority Queue - Interface I

- The domain of the ADT Priority Queue:  
 $\mathcal{PQ} = \{pq | pq \text{ is a priority queue with elements } (e, p), e \in TElement, p \in TPriority\}$
- The interface of the ADT Priority Queue contains the following operations:

# Priority Queue - Interface II

- **init** ( $pq$ ,  $R$ )
  - **descr:** creates a new empty priority queue
  - **pre:**  $R$  is a relation over the priorities,  
 $R : TPriority \times TPriority$
  - **post:**  $pq \in \mathcal{PQ}$ ,  $pq$  is an empty priority queue

# Priority Queue - Interface III

- **destroy(pq)**
  - **descr:** destroys a priority queue
  - **pre:**  $pq \in \mathcal{PQ}$
  - **post:**  $pq$  was destroyed

- **push( $pq$ ,  $e$ ,  $p$ )**
  - **descr:** pushes (adds) a new element to the priority queue
  - **pre:**  $pq \in \mathcal{PQ}$ ,  $e \in TElem$ ,  $p \in TPriority$
  - **post:**  $pq' \in \mathcal{PQ}$ ,  $pq' = pq \oplus (e, p)$

# Priority Queue - Interface V

- **pop (pq)**
  - **descr:** pops (removes) from the priority queue the element with the highest priority. It returns both the element and its priority
  - **pre:**  $pq \in \mathcal{PQ}$ ,  $pq$  is not empty
  - **post:**  $pop \leftarrow (e, p)$ ,  $e \in TElem$ ,  $p \in TPriority$ ,  $e$  is the element with the highest priority from  $pq$ ,  $p$  is its priority.  
 $pq' \in \mathcal{PQ}$ ,  $pq' = pq \ominus (e, p)$
  - **throws:** an exception if the priority queue is empty.

- **top (pq)**

- **descr:** returns from the priority queue the element with the highest priority and its priority. It does not modify the priority queue.
- **pre:**  $pq \in \mathcal{PQ}$ ,  $pq$  is not empty
- **post:**  $top \leftarrow (e, p)$ ,  $e \in TElm$ ,  $p \in TPriority$ ,  $e$  is the element with the highest priority from  $pq$ ,  $p$  is its priority.
- **throws:** an exception if the priority queue is empty.

- **isEmpty(pq)**

- **Description:** checks if the priority queue is empty (it has no elements)
- **Pre:**  $pq \in \mathcal{PQ}$
- **Post:**

$$isEmpty \leftarrow \begin{cases} \text{true, if } pq \text{ has no elements} \\ \text{false, otherwise} \end{cases}$$

# Priority Queue - Interface VIII

- **Note:** priority queues cannot be iterated, so they don't have an *iterator* operation!

- Consider the following problem: *we have a text and want to find the word that appears most frequently in this text.* What would be the characteristics of the container used for this problem?

- Consider the following problem: *we have a text and want to find the word that appears most frequently in this text.* What would be the characteristics of the container used for this problem?
  - We need key (word) - value (number of occurrence) pairs
  - Keys should be unique
  - Order of the keys is not important
- The container in which we store key - value pairs, and where the keys are unique and they are in no particular order is the **ADT Map** (or Dictionary)

- Domain of the ADT Map:

$\mathcal{M} = \{m | m \text{ is a map with elements } e = < k, v >, \text{ where } k \in TKey \text{ and } v \in TValue\}$

- **init( $m$ )**
  - **descr:** creates a new empty map
  - **pre:** true
  - **post:**  $m \in \mathcal{M}$ ,  $m$  is an empty map.

- **destroy( $m$ )**
  - **descr:** destroys a map
  - **pre:**  $m \in \mathcal{M}$
  - **post:**  $m$  was destroyed

- $\text{add}(m, k, v)$ 
  - **descr:** add a new key-value pair to the map (the operation can be called *put* as well). If the key is already in the map, the corresponding value will be replaced with the new one. The operation returns the old value, or  $0_{TValue}$  if the key was not in the map yet.
  - **pre:**  $m \in \mathcal{M}, k \in TKey, v \in TValue$
  - **post:**  $m' \in \mathcal{M}, m' = m \cup \langle k, v \rangle, add \leftarrow v', v' \in TValue$  where

$$v' \leftarrow \begin{cases} v'', & \text{if } \exists \langle k, v'' \rangle \in m \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

- remove( $m, k$ )

- descr:** removes a pair with a given key from the map. Returns the value associated with the key, or  $0_{TValue}$  if the key is not in the map.
- pre:**  $m \in \mathcal{M}, k \in TKey$
- post:**  $remove \leftarrow v, v \in TValue$ , where

$$v \leftarrow \begin{cases} v', & \text{if } \exists < k, v' > \in m \text{ and } m' \in \mathcal{M}, \\ & m' = m \setminus < k, v' > \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

- $\text{search}(m, k)$

- **descr:** searches for the value associated with a given key in the map
- **pre:**  $m \in \mathcal{M}, k \in TKey$
- **post:**  $\text{search} \leftarrow v, v \in TValue$ , where

$$v \leftarrow \begin{cases} v', & \text{if } \exists < k, v' > \in m \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

- **iterator( $m$ ,  $it$ )**
  - **descr:** returns an iterator for a map
  - **pre:**  $m \in \mathcal{M}$
  - **post:**  $it \in \mathcal{I}$ ,  $it$  is an iterator over  $m$ .
- **Obs:** The iterator for the map is similar to the iterator for other ADTs, but the *getCurrent* operation returns a  $\langle key, value \rangle$  pair.

- **size( $m$ )**
  - **descr:** returns the number of pairs from the map
  - **pre:**  $m \in \mathcal{M}$
  - **post:** size  $\leftarrow$  the number of pairs from  $m$

- **isEmpty(m)**
  - **descr:** verifies if the map is empty
  - **pre:**  $m \in \mathcal{M}$
  - **post:**  $isEmpty \leftarrow \begin{cases} true, & \text{if } m \text{ contains no pairs} \\ false, & \text{otherwise} \end{cases}$

# Other possible operations I

- Other possible operations
- $\text{keys}(m, s)$ 
  - **descr:** returns the set of keys from the map
  - **pre:**  $m \in \mathcal{M}$
  - **post:**  $s \in \mathcal{S}$ ,  $s$  is the set of all keys from  $m$

# Other possible operations II

- $\text{values}(m, b)$ 
  - **descr:** returns a bag with all the values from the map
  - **pre:**  $m \in \mathcal{M}$
  - **post:**  $b \in \mathcal{B}$ ,  $b$  is the bag of all values from  $m$

# Other possible operations III

- $\text{pairs}(m, s)$ 
  - **descr:** returns the set of pairs from the map
  - **pre:**  $m \in \mathcal{M}$
  - **post:**  $s \in \mathcal{S}$ ,  $s$  is the set of all pairs from  $m$

- We can have a Map where we can define an order (a relation) on the set of possible keys
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted map, the iterator has to iterate through the pairs in the order given by the *relation*, and the operations *keys* and *pairs* return SortedSets.

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 5

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

- Containers

- ADT Matrix
- ADT Stack
- ADT Queue
- ADT PriorityQueue
- ADT Map
- ADT SortedMap

- Containers
- Linked Lists

- Morse Code, is a code which assigns to every letter a sequence of dots and dashes.

A	• -	J	• - - -	S	• • •
B	- • • •	K	- • -	T	-
C	- • - •	L	• - - •	U	• • -
D	- • •	M	--	V	• • • -
E	•	N	- •	W	• - -
F	• • - •	O	- - -	X	- • • -
G	- - - •	P	• - - •	Y	- • - -
H	• • • •	Q	- - - •	Z	- - - • •
I	• •	R	• - -		

<https://medium.com/@timboucher/learning-morse-code-35e1f4d285f6>

- Given a list of words, find the largest subset of the words, for which the Morse representation is the same.

- For example, if the words are *cat*, *ca*, *nna*, *abc* and *nnet*, their Morse code representation is:
  - cat* -.-..-
  - ca* -.-..-
  - nna* -.-..-
  - abc* .-...-.-.
  - nnet* -.-..-
- What would be the characteristics of the container used for this problem?

- For example, if the words are *cat*, *ca*, *nna*, *abc* and *nnet*, their Morse code representation is:
  - cat* -.-..-
  - ca* -.-..-
  - nna* -.-..-
  - abc* .-...-.-.
  - nnet* -.-..-
- What would be the characteristics of the container used for this problem?
  - We could solve the problem if we used the Morse representation of a word as a key and the corresponding word as a value
  - One key can have multiple values
  - Order of the elements is not important
- The container in which we store key - value pairs, and where a key can have multiple associated values, is called a **ADT MultiMap**.

- Domain of ADT MultiMap:

$$\mathcal{MM} = \{mm | mm \text{ is a Multimap with TKey, TValue pairs}\}$$

- **init (mm)**
  - **descr:** creates a new empty multimap
  - **pre:** true
  - **post:**  $mm \in MM$ ,  $mm$  is an empty multimap

- **destroy(mm)**
  - **descr:** destroys a multimap
  - **pre:**  $mm \in \mathcal{M}\mathcal{M}$
  - **post:** the multimap was destroyed

# ADT MultiMap - Interface III

- **add(mm, k, v)**
  - **descr:** add a new pair to the multimap
  - **pre:**  $mm \in \mathcal{MM}$ ,  $k - TKey$ ,  $v - TValue$
  - **post:**  $mm' \in \mathcal{MM}$ ,  $mm' = mm \cup \langle k, v \rangle$

- remove(mm, k, v)

- descr:** removes a key value pair from the multimap
- pre:**  $mm \in \mathcal{MM}$ ,  $k - TKey$ ,  $v - TValue$
- post:**  $remove \leftarrow$   
$$\begin{cases} true, & \text{if } \langle k, v \rangle \in mm, mm' \in \mathcal{MM}, mm' = mm - \langle k, v \rangle \\ false, & \text{otherwise} \end{cases}$$

- **search(mm, k, l)**
  - **descr:** returns a list with all the values associated to a key
  - **pre:**  $mm \in \mathcal{MM}$ ,  $k - TKey$
  - **post:**  $l \in \mathcal{L}$ ,  $l$  is the list of values associated to the key  $k$ . If  $k$  is not in the multimap,  $l$  is the empty list.

- **iterator(mm, it)**
  - **descr:** returns an iterator over the multimap
  - **pre:**  $mm \in \mathcal{MM}$
  - **post:**  $it \in \mathcal{I}$ ,  $it$  is an iterator over  $mm$ , the current element from  $it$  is the first pair from  $mm$ , or,  $it$  is invalid if  $mm$  is empty
- **Obs:** the iterator for a MultiMap is similar to the iterator for other containers, but the *getCurrent* operation returns a <key, value> pair.

- **size(mm)**
  - **descr:** returns the number of pairs from the multimap
  - **pre:**  $mm \in \mathcal{MM}$
  - **post:**  $\text{size} \leftarrow \text{the number of pairs from mm}$

- Other possible operations:
- **keys( $mm$ ,  $s$ )**
  - **descr:** returns the set of all keys from the multimap
  - **pre:**  $mm \in MM$
  - **post:**  $s \in S$ ,  $s$  is the set of all keys from  $mm$

- **values(mm, b)**
  - **descr:** returns the bag of all values from the multimap
  - **pre:**  $mm \in \mathcal{MM}$
  - **post:**  $b \in \mathcal{Bm}$   $b$  is a bag with all the values from  $mm$

- **pairs( $mm$ ,  $b$ )**
  - **descr:** returns the bag of all pairs from the multimap
  - **pre:**  $mm \in \mathcal{MM}$
  - **post:**  $b \in \mathcal{B}$ ,  $b$  is a bag with all the pairs from  $mm$

- We can have a MultiMap where we can define an order (a relation) on the set of possible keys. However, if a key has multiple values, they can be in any order (we order the keys only, not the values)  $\Rightarrow$  **ADT SortedMultiMap**
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted MultiMap, the iterator has to iterate through the pairs in the order given by the *relation*, and the operations *keys* and *pairs* return SortedSet and SortedBag.

# ADT MultiMap - representations

- There are several data structures that can be used to implement an ADT MultiMap (or ADT SortedMultiMap), the dynamic array being one of them (others will be discussed later):
- Regardless of the data structure used, there are two options to represent a MultiMap (sorted or not):
  - Store individual  $\langle \text{key}, \text{value} \rangle$  pairs. If a key has multiple values, there will be multiple pairs containing this key. (R1)
  - Store unique keys and for each key store a *list* of associated values. (R2)

- For the example with the Morse code, we would have:

-...- cat	-.- ca	-.-.. nna	.-.-.-. abc	-.-.. nnet
-----------	--------	-----------	-------------	------------

- Key is written with red and the value with black.
- Every element is one key - value pair.

- For the example with the Morse code, we would have:

	[cat]
	[ca, nna, nnet]
	[abc]

- Key is written with red and the value with black.
- Every element is one key together with all the values belonging to it. The *list of values* can be another dynamic array, or a linked list, or any other data structure.

- A *list* can be seen as a sequence of elements of the same type,  $\langle l_1, l_2, \dots, l_n \rangle$ , where there is an order of the elements, and each element has a *position* inside the list.
- In a list, the order of the elements is important (positions are important).
- The number of elements from a list is called the length of the list. A list without elements is called *empty*.

- A List is a container which is either *empty* or
  - it has a unique *first* element
  - it has a unique *last* element
  - for every element (except for the last) there is a unique *successor* element
  - for every element (except for the first) there is a unique *predecessor* element
- In a list, we can insert elements (using positions), remove elements (using positions), we can access the successor and predecessor of an element from a given position, we can access an element from a position.

- Every element from a list has a unique position in the list:
  - positions are relative to the list (but important for the list)
  - the position of an element:
    - identifies the element from the list
    - determines the position of the successor and predecessor element (if they exist).

- Position of an element can be seen in different ways:
  - as the *rank* of the element in the list (first, second, third, etc.)
  - similarly to an array, the position of an element is actually its index
  - as a *reference* to the memory location where the element is stored.
    - for example a pointer to the memory location
- For a general treatment, we will consider in the following the *position* of an element in an abstract manner, and we will consider that positions are of type *TPosition*

- A position  $p$  will be considered *valid* if it denotes the position of an actual element from the list:
  - if  $p$  is a pointer to a memory location,  $p$  is valid if it is the address of an element from a list (not NIL or some other address that is not the address of any element)
  - if  $p$  is the rank of the element from the list,  $p$  is valid if it is between 1 and the number of elements.
- For an invalid position we will use the following notation:  $\perp$

- Domain of the ADT List:

$\mathcal{L} = \{l | l \text{ is a list with elements of type } T\text{Elem, each having a unique position in } l \text{ of type } T\text{Position}\}$

- **init( $l$ )**
  - **descr:** creates a new, empty list
  - **pre:** true
  - **post:**  $l \in \mathcal{L}$ ,  $l$  is an empty list

- **first( $l$ )**

- **descr:** returns the  $TPosition$  of the first element
- **pre:**  $l \in \mathcal{L}$
- **post:**  $first \leftarrow p \in TPosition$

$$p = \begin{cases} \text{the position of the first element from } l & \text{if } l \neq \emptyset \\ \perp & \text{otherwise} \end{cases}$$

- **last(l)**

- **descr:** returns the TPosition of the last element

- **pre:**  $l \in \mathcal{L}$

- **post:**  $last \leftarrow p \in TPosition$

$$p = \begin{cases} \text{the position of the last element from } l & \text{if } l \neq \emptyset \\ \perp & \text{otherwise} \end{cases}$$

- **valid(l, p)**
  - **descr:** checks whether a TPosition is valid in a list
  - **pre:**  $l \in \mathcal{L}, p \in TPosition$
  - **post:**  $valid \leftarrow \begin{cases} true & \text{if } p \text{ is a valid position in } l \\ false & \text{otherwise} \end{cases}$

- $\text{next}(l, p)$

- **descr:** goes to the next *TPosition* from a list
- **pre:**  $l \in \mathcal{L}, p \in TPosition, valid(l, p)$
- **post:**

$$next \leftarrow q \in TPosition$$

$$q = \begin{cases} \text{the position of the next element after } p & \text{if } p \text{ is not the last position} \\ \perp & \text{otherwise} \end{cases}$$

- **throws:** exception if  $p$  is not valid

- $\text{previous}(l, p)$ 
  - **descr:** goes to the previous  $TPosition$  from a list
  - **pre:**  $l \in \mathcal{L}, p \in TPosition, valid(l, p)$
  - **post:**

$$previous \leftarrow q \in TPosition$$

$$q = \begin{cases} \text{the position of the element before } p & \text{if } p \text{ is not the first position} \\ \perp & \text{otherwise} \end{cases}$$

- **throws:** exception if  $p$  is not valid

- **getElement(l, p)**
  - **descr:** returns the element from a given TPosition
  - **pre:**  $l \in \mathcal{L}, p \in TPosition, valid(l, p)$
  - **post:**  $getElement \leftarrow e, e \in TElem, e = \text{the element from position } p \text{ from } l$
  - **throws:** exception if  $p$  is not valid

- position( $l, e$ )

- **descr:** returns the TPosition of an element
- **pre:**  $l \in \mathcal{L}, e \in TElm$
- **post:**

$$position \leftarrow p \in TPosition$$

$$p = \begin{cases} \text{the first position of element } e \text{ from } l & \text{if } e \in l \\ \perp & \text{otherwise} \end{cases}$$

- **setElement(l, p, e)**

- **descr:** replaces an element from a TPosition with another
- **pre:**  $l \in \mathcal{L}, p \in TPosition, e \in TElem, valid(l, p)$
- **post:**  $l' \in \mathcal{L}$ , the element from position  $p$  from  $l'$  is  $e$ ,  
 $setElement \leftarrow el, el \in TElem, el$  is the element from position  
 $p$  from  $l$  (returns the previous value from the position)
- **throws:** exception if  $p$  is not valid

- **addToBeginning(l, e)**
  - **descr:** adds a new element to the beginning of a list
  - **pre:**  $l \in \mathcal{L}$ ,  $e \in TElm$
  - **post:**  $l' \in \mathcal{L}$ ,  $l'$  is the result after the element  $e$  was added at the beginning of  $l$

- **addToEnd(l, e)**
  - **descr:** adds a new element to the end of a list
  - **pre:**  $l \in \mathcal{L}, e \in TElm$
  - **post:**  $l' \in \mathcal{L}$ ,  $l'$  is the result after the element  $e$  was added at the end of  $l$

- **addBeforePosition(l, p, e)**

- **descr:** inserts a new element before a given position
- **pre:**  $l \in \mathcal{L}, p \in TPosition, e \in TElem, valid(l, p)$
- **post:**  $l' \in \mathcal{L}$ ,  $l'$  is the result after the element  $e$  was added in  $l$  before the position  $p$
- **throws:** exception if  $p$  is not valid

- **addAfterPosition(l, p, e)**
  - **descr:** inserts a new element after a given position
  - **pre:**  $l \in \mathcal{L}, p \in TPosition, e \in TElem, valid(l, p)$
  - **post:**  $l' \in \mathcal{L}$ ,  $l'$  is the result after the element  $e$  was added in  $l$  after the position  $p$
  - **throws:** exception if  $p$  is not valid

- **remove(l, p)**
  - **descr:** removes an element from a given position from a list
  - **pre:**  $l \in \mathcal{L}, p \in TPosition, valid(l, p)$
  - **post:**  $remove \leftarrow e, e \in TElem, e$  is the element from position  $p$  from  $l, l' \in \mathcal{L}, l' = l - e$ .
  - **throws:** exception if  $p$  is not valid

- remove( $l, e$ )

- descr:** removes the first occurrence of a given element from a list
- pre:**  $l \in \mathcal{L}, e \in TElm$
- post:**

$$remove \leftarrow \begin{cases} true & \text{if } e \in l \text{ and it was removed} \\ false & \text{otherwise} \end{cases}$$

- **search(*l*, *e*)**

- **descr:** searches for an element in the list
- **pre:**  $l \in \mathcal{L}, e \in TElm$
- **post:**

$$search \leftarrow \begin{cases} \text{true} & \text{if } e \in l \\ \text{false} & \text{otherwise} \end{cases}$$

- **isEmpty( $l$ )**

- **descr:** checks if a list is empty
- **pre:**  $l \in \mathcal{L}$
- **post:**

$$isEmpty \leftarrow \begin{cases} \text{true} & \text{if } l = \emptyset \\ \text{false} & \text{otherwise} \end{cases}$$

- **size( $l$ )**

- **descr:** returns the number of elements from a list
- **pre:**  $l \in \mathcal{L}$
- **post:**  $\text{size} \leftarrow$  the number of elements from  $l$

- **destroy(l)**
  - **descr:** destroys a list
  - **pre:**  $l \in \mathcal{L}$
  - **post:** l was destroyed

- **iterator( $l$ ,  $it$ )**
  - **descr:** returns an iterator for a list
  - **pre:**  $l \in \mathcal{L}$
  - **post:**  $it \in \mathcal{I}$ ,  $it$  is an iterator over  $l$ , the current element from  $it$  is the first element from  $l$ , or, if  $l$  is empty,  $it$  is invalid

# TPosition - Integer

- In Python and Java, TPosition is represented by an index.
- We can add and remove using index and we can access elements using their index (but we have iterator as well for the List).

- For example (Python):

```
insert (int index, E object)  
index (E object)
```

- Returns an integer value, position of the element (or exception if *object* is not in the list)

- For example (Java):

```
void add(int index, E element)  
E get(int index)  
E remove(int index)
```

- Returns the removed element

- If we consider that TPosition is an Integer value (similar to Python and Java), we can have an *IndexedList*
- In case of an *IndexedList* the operations that work with a position take as parameter integer numbers representing these positions
- There are less operations in the interface of the *IndexedList*
  - Operations *first*, *last*, *next*, *previous*, *valid* do not exist

- **init( $I$ )**
  - **descr:** creates a new, empty list
  - **pre:** true
  - **post:**  $I \in \mathcal{L}$ ,  $I$  is an empty list

- `getElement(l, i)`
  - **descr:** returns the element from a given position
  - **pre:**  $l \in \mathcal{L}, i \in \mathcal{N}, i$  is a valid position
  - **post:**  $\text{getElement} \leftarrow e, e \in TElm, e = \text{the element from position } i \text{ from } l$
  - **throws:** exception if  $i$  is not valid

- **position(l, e)**

- **descr:** returns the position of an element
- **pre:**  $l \in \mathcal{L}, e \in T\text{Elem}$
- **post:**

$$position \leftarrow i \in \mathbb{N}$$

$$i = \begin{cases} \text{the first position of element } e \text{ from } l & \text{if } e \in l \\ -1 & \text{otherwise} \end{cases}$$

- **setElement( $l$ ,  $i$ ,  $e$ )**

- **descr:** replaces an element from a position with another
- **pre:**  $l \in \mathcal{L}, i \in \mathcal{N}, e \in TElm, i$  is a valid position
- **post:**  $l' \in \mathcal{L}$ , the element from position  $i$  from  $l'$  is  $e$ ,  
 $setElement \leftarrow el, el \in TElm, el$  is the element from position  $i$  from  $l$  (returns the previous value from the position)
- **throws:** exception if  $i$  is not valid

- **addToBeginning(l, e)**
  - **descr:** adds a new element to the beginning of a list
  - **pre:**  $l \in \mathcal{L}$ ,  $e \in TElm$
  - **post:**  $l' \in \mathcal{L}$ ,  $l'$  is the result after the element  $e$  was added at the beginning of  $l$

- **addToEnd(l, e)**
  - **descr:** adds a new element to the end of a list
  - **pre:**  $l \in \mathcal{L}, e \in TElm$
  - **post:**  $l' \in \mathcal{L}$ ,  $l'$  is the result after the element  $e$  was added at the end of  $l$

- `addToPosition(l, i, e)`

- **descr:** inserts a new element at a given position (it is the same as *addBeforePosition*)
- **pre:**  $l \in \mathcal{L}, i \in \mathcal{N}, e \in TElm, i$  is a valid position ( $\text{size} + 1$  is valid for adding an element)
- **post:**  $l' \in \mathcal{L}$ ,  $l'$  is the result after the element  $e$  was added in  $l$  at the position  $i$
- **throws:** exception if  $i$  is not valid

- `remove(l, i)`
  - **descr:** removes an element from a given position from a list
  - **pre:**  $l \in \mathcal{L}, i \in \mathcal{N}, i$  is a valid position
  - **post:**  $remove \leftarrow e, e \in TElem, e$  is the element from position  $i$  from  $l, l' \in \mathcal{L}, l' = l - e.$
  - **throws:** exception if  $i$  is not valid

- remove( $l, e$ )

- descr:** removes the first occurrence of a given element from a list
- pre:**  $l \in \mathcal{L}, e \in TElm$
- post:**

$$remove \leftarrow \begin{cases} \text{true} & \text{if } e \in l \text{ and it was removed} \\ \text{false} & \text{otherwise} \end{cases}$$

# ADT IndexedList X

- **search(l, e)**

- **descr:** searches for an element in the list
- **pre:**  $l \in \mathcal{L}, e \in TElm$
- **post:**

$$search \leftarrow \begin{cases} \text{true} & \text{if } e \in l \\ \text{false} & \text{otherwise} \end{cases}$$

- **isEmpty(l)**

- **descr:** checks if a list is empty
- **pre:**  $l \in \mathcal{L}$
- **post:**

$$isEmpty \leftarrow \begin{cases} \text{true} & \text{if } l = \emptyset \\ \text{false} & \text{otherwise} \end{cases}$$

- **size( $l$ )**

- **descr:** returns the number of elements from a list
- **pre:**  $l \in \mathcal{L}$
- **post:**  $\text{size} \leftarrow$  the number of elements from  $l$

- **destroy(l)**
  - **descr:** destroys a list
  - **pre:**  $l \in \mathcal{L}$
  - **post:** l was destroyed

- **iterator( $l$ ,  $it$ )**
  - **descr:** returns an iterator for a list
  - **pre:**  $l \in \mathcal{L}$
  - **post:**  $it \in \mathcal{I}$ ,  $it$  is an iterator over  $l$ , the current element from  $it$  is the first element from  $l$ , or, if  $l$  is empty,  $it$  is invalid

- In STL (C++), TPosition is represented by an iterator.
- For example - vector:

iterator insert(iterator position, const value\_type& val)

- Returns an iterator which points to the newly inserted element

iterator erase (iterator position);

- Returns an iterator which points to the element after the removed one

- For example - list:

iterator insert(iterator position, const value\_type& val)

iterator erase (iterator position);

- If we consider that TPosition is an Iterator (similar to C++) we can have an *IteratedList*.
- In case of an *IteratedList* the operations that take as parameter a position use an Iterator (and the position is the current element from the Iterator)
- Operations *valid*, *next*, *previous* no longer exist in the interface of the List (they are operations for the Iterator).

- **init( $l$ )**
  - **descr:** creates a new, empty list
  - **pre:** true
  - **post:**  $l \in \mathcal{L}$ ,  $l$  is an empty list

- **first( $I$ )**

- **descr:** returns an Iterator set to the first element
- **pre:**  $I \in \mathcal{L}$
- **post:**  $first \leftarrow it \in \text{Iterator}$

$$it = \begin{cases} \text{an iterator set to the first element} & \text{if } I \neq \emptyset \\ \text{an invalid iterator} & \text{otherwise} \end{cases}$$

- **last( $I$ )**

- **descr:** returns an Iterator set to the last element

- **pre:**  $I \in \mathcal{L}$

- **post:**  $last \leftarrow it \in \text{Iterator}$

$$it = \begin{cases} \text{an iterator set to the last element} & \text{if } I \neq \emptyset \\ \text{an invalid iterator} & \text{otherwise} \end{cases}$$

- **getElement(l, it)**

- **descr:** returns the element from the position denoted by an Iterator
- **pre:**  $l \in \mathcal{L}, it \in \text{Iterator}, valid(it)$
- **post:**  $\text{getElement} \leftarrow e, e \in TElm, e = \text{the element from } l \text{ from the current position}$
- **throws:** exception if  $it$  is not valid

- $\text{position}(l, e)$ 
  - **descr:** returns an iterator set to the first position of an element
  - **pre:**  $l \in \mathcal{L}, e \in T\text{Elem}$
  - **post:**

$$\textit{position} \leftarrow \textit{it} \in \textit{Iterator}$$

$$\textit{it} = \begin{cases} \text{an iterator set to the first position of element } e \text{ from } l & \text{if } e \in l \\ \text{an invalid iterator} & \text{otherwise} \end{cases}$$

- `setElement(l, it, e)`
  - **descr:** replaces the element from the position denoted by an Iterator with another element
  - **pre:**  $l \in \mathcal{L}$ ,  $it \in \text{Iterator}$ ,  $e \in TElm$ ,  $\text{valid}(it)$
  - **post:**  $l' \in \mathcal{L}$ , the element from the position denoted by  $it$  from  $l'$  is  $e$ ,  $\text{setElement} \leftarrow el$ ,  $el \in TElm$ ,  $el$  is the element from the current position from  $it$  from  $l$  (returns the previous value from the position)
  - **throws:** exception if  $it$  is not valid

- **addToBeginning(l, e)**
  - **descr:** adds a new element to the beginning of a list
  - **pre:**  $l \in \mathcal{L}$ ,  $e \in TElm$
  - **post:**  $l' \in \mathcal{L}$ ,  $l'$  is the result after the element  $e$  was added at the beginning of  $l$

- **addToEnd(l, e)**
  - **descr:** inserts a new element at the end of a list
  - **pre:**  $l \in \mathcal{L}, e \in TElm$
  - **post:**  $l' \in \mathcal{L}$ ,  $l'$  is the result after the element  $e$  was added at the end of  $l$

- **addToPosition(l, it, e)**

- **descr:** inserts a new element at a given position specified by the iterator (it is the same as *addAfterPosition*)
- **pre:**  $l \in \mathcal{L}, it \in \text{Iterator}, e \in T\text{Elem}, \text{valid}(it)$
- **post:**  $l' \in \mathcal{L}, l'$  is the result after the element *e* was added in *l* at the position specified by *it*
- **throws:** exception if *it* is not valid

- **remove(l, it)**
  - **descr:** removes an element from a given position specified by the iterator from a list
  - **pre:**  $l \in \mathcal{L}, it \in \text{Iterator}, valid(it)$
  - **post:**  $remove \leftarrow e, e \in TElem, e$  is the element from the position from  $l$  denoted by  $it, l' \in \mathcal{L}, l' = l - e.$
  - **throws:** exception if  $it$  is not valid

- remove( $l, e$ )

- descr:** removes the first occurrence of a given element from a list
- pre:**  $l \in \mathcal{L}, e \in TElm$
- post:**

$$remove \leftarrow \begin{cases} true & \text{if } e \in l \text{ and it was removed} \\ false & \text{otherwise} \end{cases}$$

- **search(*l*, *e*)**

- **descr:** searches for an element in the list
- **pre:**  $l \in \mathcal{L}, e \in TElm$
- **post:**

$$search \leftarrow \begin{cases} \text{true} & \text{if } e \in l \\ \text{false} & \text{otherwise} \end{cases}$$

- **isEmpty( $I$ )**

- **descr:** checks if a list is empty
- **pre:**  $I \in \mathcal{L}$
- **post:**

$$isEmpty \leftarrow \begin{cases} \text{true} & \text{if } I = \emptyset \\ \text{false} & \text{otherwise} \end{cases}$$

- **size( $l$ )**

- **descr:** returns the number of elements from a list
- **pre:**  $l \in \mathcal{L}$
- **post:**  $\text{size} \leftarrow$  the number of elements from  $l$

- **destroy(l)**
  - **descr:** destroys a list
  - **pre:**  $l \in \mathcal{L}$
  - **post:** l was destroyed

# ADT SortedList

- We can define the ADT *SortedList*, in which the elements are memorized in an order given by a relation.
- You have below the list of operations for ADT *List*

- init(l)
- first(l)
- last(l)
- valid(l, p)
- next(l, p)
- previous(l, p)
- getElement(l, p)
- position(l, e)

- setElement(l, p, e)
- addToBeginning(l, e)
- addToEnd(l, e)
- addToPosition(l, p, e)
- remove(l, p)
- remove(l, e)
- search(l, e)
- isEmpty(l)
- size(l)
- destroy(l)
- iterator(l, it)

- Which operations do no longer exist for a *SortedList*? What operations should be added? Should we change the parameters of some operations?

# ADT SortedList

- The interface of the ADT *SortedList* is very similar to that of the ADT *List* with some exceptions:
  - The *init* function takes as parameter a relation that is going to be used to order the elements
  - We no longer have several *add* operations (*addToBeginning*, *addToEnd*, *addToPosition*), we have one single *add* operation, which takes as parameter only the element to be added (and adds it to the position where it should go based on the relation)
  - We no longer have a *setElement* operation (might violate ordering)
- We can consider *TPosition* in two different ways for a *SortedList* as well  $\Rightarrow$  *SortedIndexedList* and *SortedIteratedList*

# Dynamic Array - review

- The main idea of the (dynamic) array is that all the elements from the array are in one single consecutive memory location.

# Dynamic Array - review

- The main idea of the (dynamic) array is that all the elements from the array are in one single consecutive memory location.
- This gives us the main advantage of the array:
  - constant time access to any element from any position
  - constant time for operations (add, remove) at the end of the array

# Dynamic Array - review

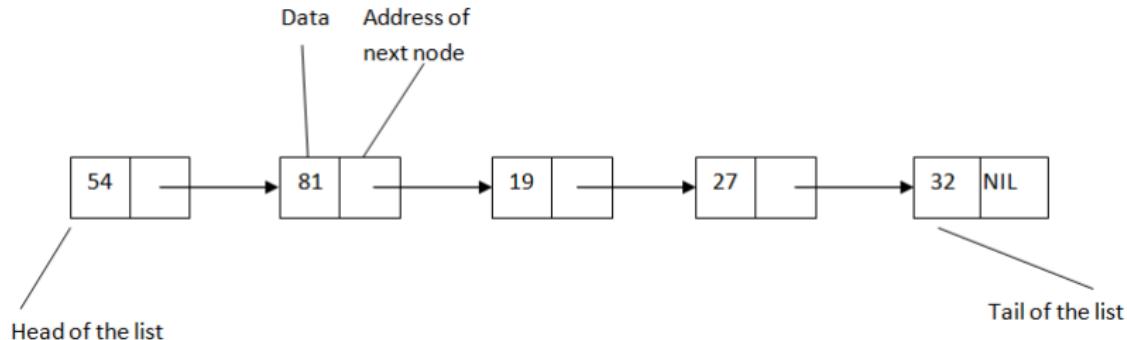
- The main idea of the (dynamic) array is that all the elements from the array are in one single consecutive memory location.
- This gives us the main advantage of the array:
  - constant time access to any element from any position
  - constant time for operations (add, remove) at the end of the array
- This gives us the main disadvantage of the array as well:
  - $\Theta(n)$  complexity for operations (add, remove) at the beginning of the array

- 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* (sometimes called *links*) 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.

- 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.

# Linked Lists

- Example of a linked list with 5 nodes:



# Singly Linked Lists - SLL

- 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: TElement //*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: TElement //*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).

- Possible operations for a singly linked list:
  - 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

```
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:

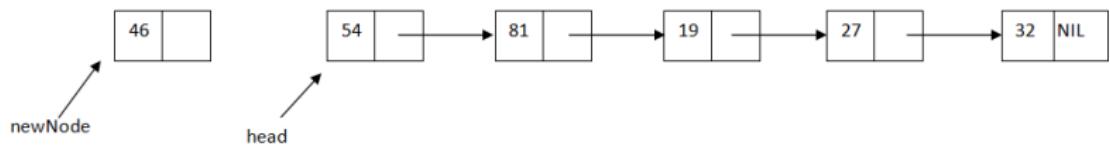
```
function search (sll, elem) is:  
    //pre: sll is a SLL - singly linked list; elem is a TElem  
    //post: returns the node which contains elem as info, or NIL  
    current ← sll.head  
    while current ≠ NIL and [current].info ≠ elem execute  
        current ← [current].next  
    end-while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

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

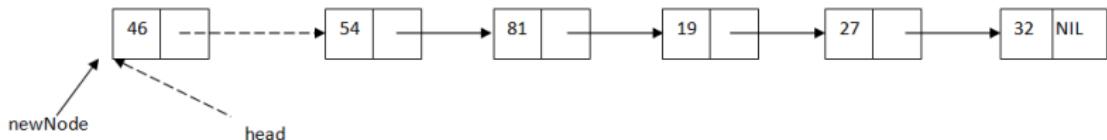
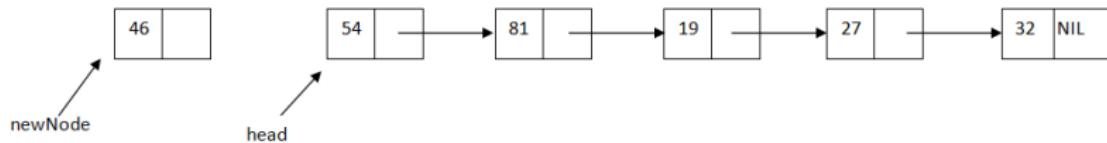
# SLL - Walking through a linked list

- 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  $\text{current} \leftarrow [\text{current}].\text{next}$  instruction)
  - we stop when the current node becomes *Nil*

# SLL - Insert at the beginning



# SLL - Insert at the beginning



# SLL - Insert at the beginning

**subalgorithm** insertFirst (sll, elem) **is:**

//pre: sll is a SLL; elem is a TElem

//post: the element elem will be inserted at the beginning of sll

newNode ← allocate() //allocate a new SLLNode

[newNode].info ← elem

[newNode].next ← sll.head

sll.head ← newNode

**end-subalgorithm**

- Complexity:

# SLL - Insert at the beginning

**subalgorithm** insertFirst (sll, elem) **is:**

//pre: sll is a SLL; elem is a TElem

//post: the element elem will be inserted at the beginning of sll

newNode ← allocate() //allocate a new SLLNode

[newNode].info ← elem

[newNode].next ← sll.head

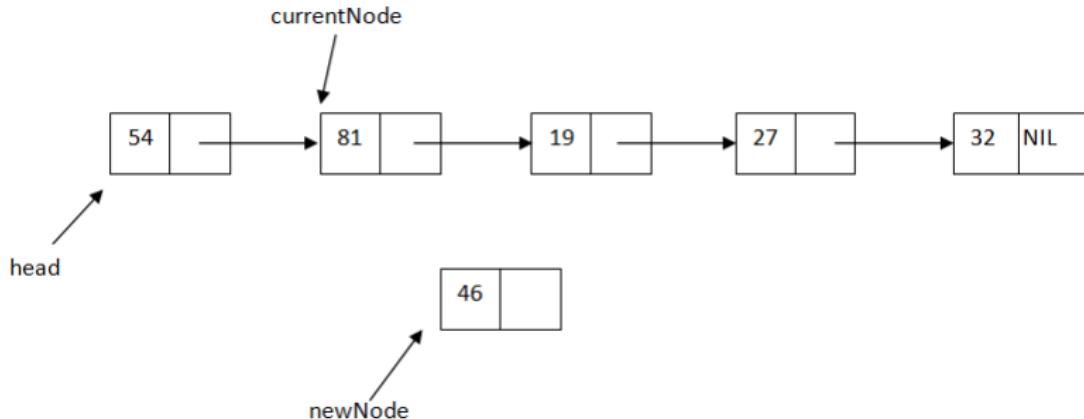
sll.head ← newNode

**end-subalgorithm**

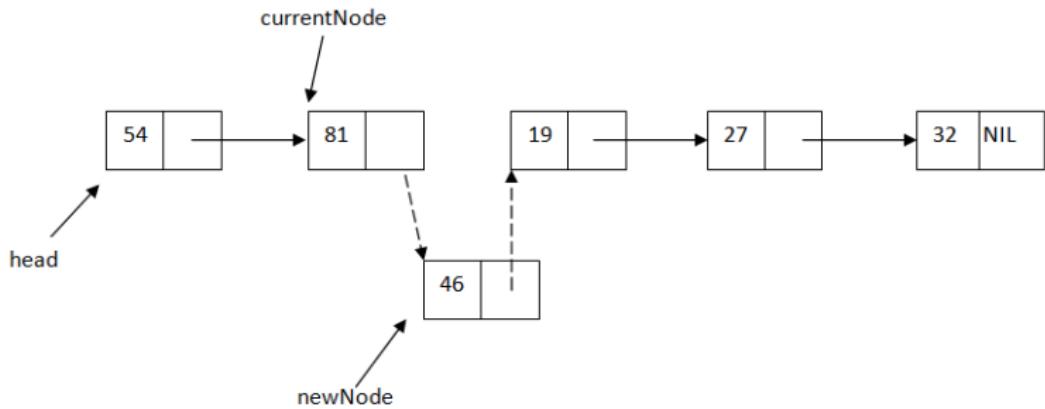
- Complexity:  $\Theta(1)$

# SLL - Insert after a node

- Suppose that we have the address of a node from the SLL and we want to insert a new element after that node.



# SLL - Insert after a node



**subalgorithm** insertAfter(sll, currentNode, elem) **is:**

//pre: *sll* is a SLL; *currentNode* is an SLLNode from *sll*;

//*elem* is a TElem

//post: a node with *elem* will be inserted after node *currentNode*

newNode ← allocate() //allocate a new SLLNode

[newNode].info ← elem

[newNode].next ← [currentNode].next

[currentNode].next ← newNode

**end-subalgorithm**

- Complexity:

# SLL - Insert after a node

**subalgorithm** insertAfter(sll, currentNode, elem) **is:**

//pre: *sll* is a SLL; *currentNode* is an SLLNode from *sll*;

//*elem* is a TElem

//post: a node with *elem* will be inserted after node *currentNode*

newNode ← allocate() //allocate a new SLLNode

[newNode].info ← elem

[newNode].next ← [currentNode].next

[currentNode].next ← newNode

**end-subalgorithm**

- Complexity:  $\Theta(1)$

# Insert before a node

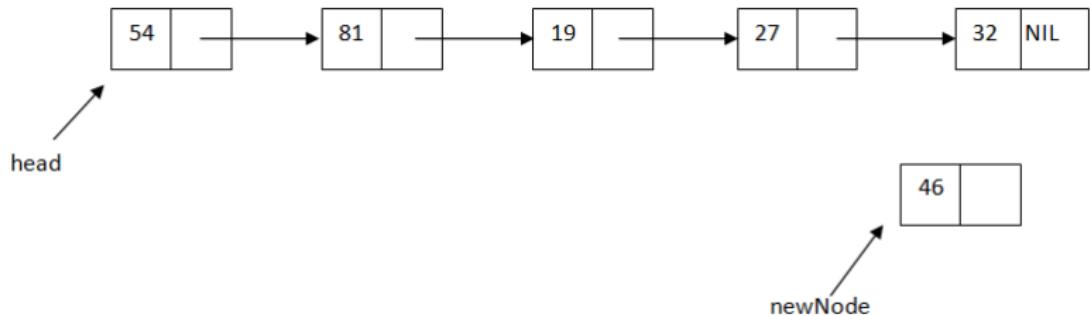
- Think about the following case: if you have a node, how can you insert an element in front of the node?

# SLL - Insert at a position

- We usually do not have the node after which we want to insert an element: we either know the position to which we want to insert, or know the element (not the node) after which we want to insert an element.
- Suppose we want to insert a new element at integer position  $p$  (after insertion the new element will be at position  $p$ ). Since we only have access to the *head* of the list we first need to find the position *after* which we insert the element.

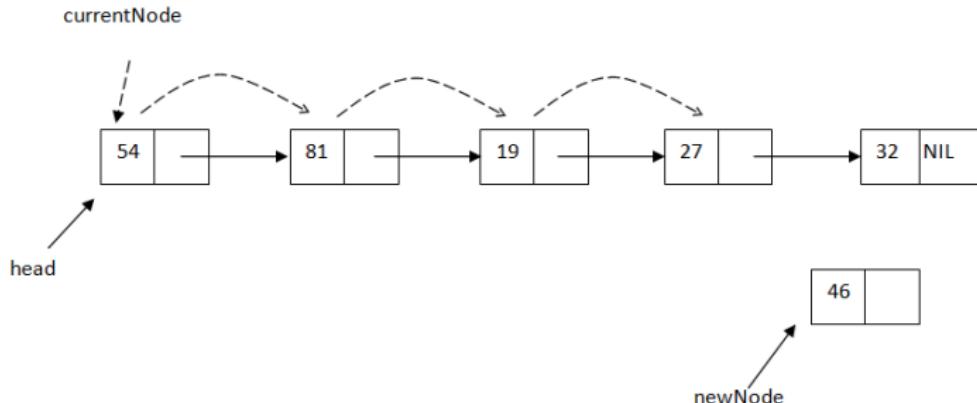
# SLL - Insert at a position

- We want to insert element 46 at position 5.



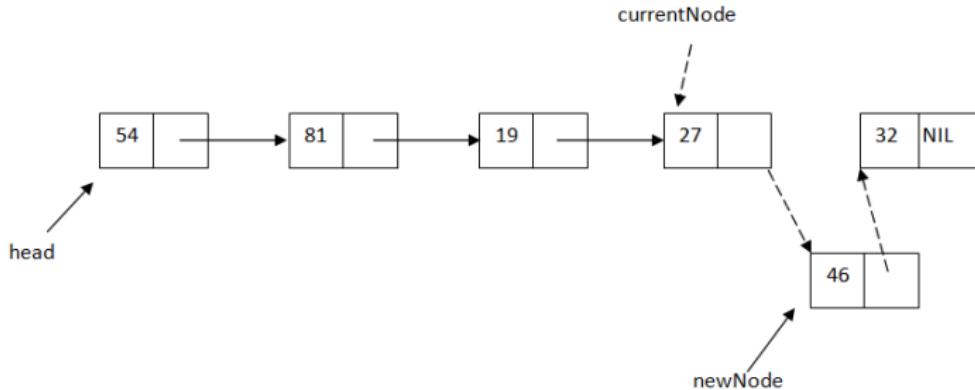
# SLL - Insert at a position

- We need the 4<sup>th</sup> node (to insert element 46 after it), but we have direct access only to the first one, so we have to take an auxiliary node (*currentNode*) to get to the position.



# SLL - Insert at a position

- Now we insert after node *currentNode*



# SLL - Insert at a position

```
subalgorithm insertPosition(sll, pos, elem) is:
    //pre: sll is a SLL; pos is an integer number; elem is a TElem
    //post: a node with TElem will be inserted at position pos
    if pos < 1 then
        @error, invalid position
    else if pos = 1 then //we want to insert at the beginning
        newNode ← allocate() //allocate a new SLLNode
        [newNode].info ← elem
        [newNode].next ← sll.head
        sll.head ← newNode
    else
        currentNode ← sll.head
        currentPos ← 1
        while currentPos < pos - 1 and currentNode ≠ NIL execute
            currentNode ← [currentNode].next
            currentPos ← currentPos + 1
        end-while
    //continued on the next slide...
```

```
if currentNode ≠ NIL then
    newNode ← allocate() //allocate a new SLLNode
    [newNode].info ← elem
    [newNode].next ← [currentNode].next
    [currentNode].next ← newNode
else
    @error, invalid position
end-if
end-if
end-subalgorithm
```

- Complexity:

```
if currentNode ≠ NIL then
    newNode ← allocate() //allocate a new SLLNode
    [newNode].info ← elem
    [newNode].next ← [currentNode].next
    [currentNode].next ← newNode
else
    @error, invalid position
end-if
end-if
end-subalgorithm
```

- Complexity:  $O(n)$

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 6

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

# In Lecture 5...

- Containers
  - ADT List
  - ADT SortedList
- Linked Lists

- Linked List
  - Singly Linked List
  - Doubly Linked List
  - Sorted List
  - Circular Lists
  - XOR Lists

- 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:  $\uparrow$  SLLNode //*address of the next node*

## SLL:

head:  $\uparrow$  SLLNode //*address of the first node*

# Get element from a given position

- Since we only have access to the head of the list, if we want to get an element from a position  $p$  we have to go through the list, node-by-node until we get to the  $p^{th}$  node.
- The process is similar to the first part of the *insertPosition* subalgorithm

# SLL - Delete a given element

- How do we delete a given element from a SLL?

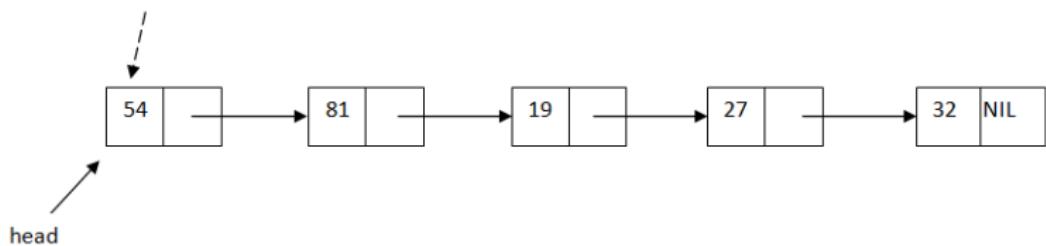
# SLL - Delete a given element

- How do we delete a given element from a SLL?
- When we want to delete a node from the middle of the list (either a node with a given element, or a node from a position), we need to find the node *before* the one we want to delete.
- The simplest way to do this, is to walk through the list using two pointers: *currentNode* and *prevNode* (the node before *currentNode*). We will stop when *currentNode* points to the node we want to delete.

# SLL - Delete a given element

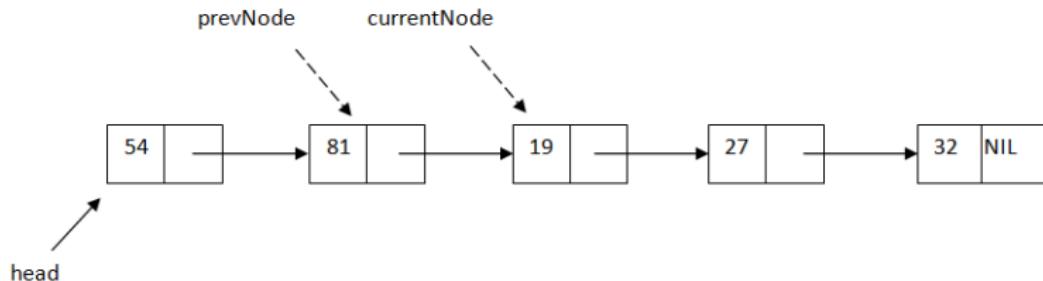
- Suppose we want to delete the node with information 19.

prevNode = NIL      currentNode



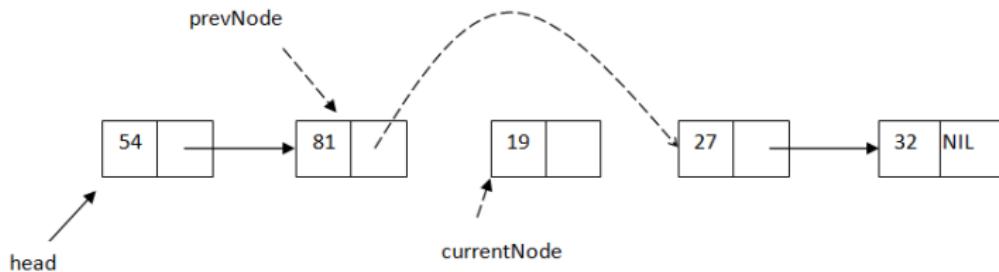
# SLL - Delete a given element

- Move with the two pointers until *currentNode* is the node we want to delete.



# SLL - Delete a given element

- Delete *currentNode* by *jumping over it*



# SLL - Delete a given element

```
function deleteElement(sll, elem) is:  
    //pre: sll is a SLL, elem is a TElm  
    //post: the node with elem is removed from sll and returned  
    currentNode ← sll.head  
    prevNode ← NIL  
    while currentNode ≠ NIL and [currentNode].info ≠ elem execute  
        prevNode ← currentNode  
        currentNode ← [currentNode].next  
    end-while  
    if currentNode ≠ NIL AND prevNode = NIL then //we delete the head  
        sll.head ← [sll.head].next  
    else if currentNode ≠ NIL then  
        [prevNode].next ← [currentNode].next  
        [currentNode].next ← NIL  
    end-if  
    deleteElement ← currentNode  
end-function
```

# SLL - Delete a given element

- Complexity of *deleteElement* function:

# SLL - Delete a given element

- Complexity of *deleteElement* function:  $O(n)$

- How can we define an iterator for a SLL?
- Remember, an iterator needs a reference to a *current element* from the data structure it iterates over. How can we denote a *current element* for a SLL?

- How can we define an iterator for a SLL?
- Remember, an iterator needs a reference to a *current element* from the data structure it iterates over. How can we denote a *current element* for a SLL?
- For the dynamic array the current element was the index of the element. Can we do the same here?

- In case of a SLL, the current element from the iterator is actually a node of the list.

SLLIterator:

list: SLL

currentElement:  $\uparrow$  SLLNode

- What should the *init* operation do?

- What should the *init* operation do?

**subalgorithm** init(it, sll) **is:**

```
//pre: sll is a SLL
//post: it is a SLLIterator over sll
it.sll ← sll
it.currentElement ← sll.head
end-subalgorithm
```

- Complexity:

- What should the *init* operation do?

**subalgorithm** init(it, sll) **is:**

```
//pre: sll is a SLL
//post: it is a SLLIterator over sll
it.sll ← sll
it.currentElement ← sll.head
end-subalgorithm
```

- Complexity:  $\Theta(1)$

# SLL - Iterator - *getCurrent* operation

- What should the *getCurrent* operation do?

# SLL - Iterator - getCurrent operation

- What should the *getCurrent* operation do?

```
function getCurrent(it) is:  
    //pre: it is a SLLIterator, it is valid  
    //post: getCurrent ← e, e is TElem, the current element from it  
    //throws: exception if it is not valid  
    if it.currentElement = NIL then  
        @throw an exception  
    end-if  
    e ← [it.currentElement].info  
    getCurrent ← e  
end-function
```

- Complexity:

# SLL - Iterator - getCurrent operation

- What should the *getCurrent* operation do?

```
function getCurrent(it) is:  
    //pre: it is a SLLIterator, it is valid  
    //post: getCurrent ← e, e is TElem, the current element from it  
    //throws: exception if it is not valid  
    if it.currentElement = NIL then  
        @throw an exception  
    end-if  
    e ← [it.currentElement].info  
    getCurrent ← e  
end-function
```

- Complexity:  $\Theta(1)$

# SLL - Iterator - next operation

- What should the *next* operation do?

## SLL - Iterator - next operation

- What should the *next* operation do?

**subalgorithm** next(it) **is:**

//pre: it is a SLLIterator, it is valid

//post: it' is a SLLIterator, the current element from it' refers to  
the next element

//throws: exception if it is not valid

**if** it.currentElement = NIL **then**

    @throw an exception

**end-if**

    it.currentElement  $\leftarrow$  [it.currentElement].next

**end-subalgorithm**

- Complexity:

- What should the *next* operation do?

**subalgorithm** next(it) **is:**

//pre: it is a SLLIterator, it is valid

//post: it' is a SLLIterator, the current element from it' refers to  
the next element

//throws: exception if it is not valid

**if** it.currentElement = NIL **then**

    @throw an exception

**end-if**

    it.currentElement  $\leftarrow$  [it.currentElement].next

**end-subalgorithm**

- Complexity:  $\Theta(1)$

# SLL - Iterator - valid operation

- What should the *valid* operation do?

# SLL - Iterator - valid operation

- What should the *valid* operation do?

```
function valid(it) is:  
//pre: it is a SLLIterator  
//post: true if it is valid, false otherwise  
if it.currentElement ≠ NIL then  
    valid ← True  
else  
    valid ← False  
end-if  
end-subalgorithm
```

- Complexity:

# SLL - Iterator - valid operation

- What should the *valid* operation do?

```
function valid(it) is:  
    //pre: it is a SLLIterator  
    //post: true if it is valid, false otherwise  
    if it.currentElement ≠ NIL then  
        valid ← True  
    else  
        valid ← False  
    end-if  
end-subalgorithm
```

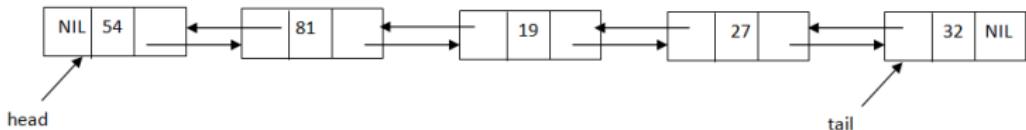
- Complexity:  $\Theta(1)$

- How could we define a bi-directional iterator for a SLL? What would be the complexity of the *previous* operation?
- How could we define a bi-directional iterator for a SLL if we know that the *previous* operation will never be called twice consecutively (two consecutive calls for the *previous* operation will always be divided by at least one call to the *next* operation)? What would be the complexity of the operations?

# Doubly Linked Lists - DLL

- A doubly linked list is similar to a singly linked list, but the nodes have references to the address of the previous node as well (besides the *next* link, we have a *prev* link as well).
- If we have a node from a DLL, we can go the next node or to the previous one: we can walk through the elements of the list in both directions.
- The *prev* link of the first element is set to *NIL* (just like the *next* link of the last element).

# Example of a Doubly Linked List



- Example of a doubly linked list with 5 nodes.

# Doubly Linked List - Representation

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

## DLLNode:

info: TElem

next:  $\uparrow$  DLLNode

prev:  $\uparrow$  DLLNode

# Doubly Linked List - Representation

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

## DLLNode:

info: TElem

next:  $\uparrow$  DLLNode

prev:  $\uparrow$  DLLNode

## DLL:

head:  $\uparrow$  DLLNode

tail:  $\uparrow$  DLLNode

- An empty list is one which has no nodes  $\Rightarrow$  the address of the first node (and the address of the last node) is NIL

**subalgorithm** init(dll) **is:**

```
//pre: true  
//post: dll is a DLL  
    dll.head  $\leftarrow$  NIL  
    dll.tail  $\leftarrow$  NIL
```

**end-subalgorithm**

- Complexity:

- An empty list is one which has no nodes  $\Rightarrow$  the address of the first node (and the address of the last node) is NIL

**subalgorithm** init(dll) **is:**

```
//pre: true  
//post: dll is a DLL  
dll.head  $\leftarrow$  NIL  
dll.tail  $\leftarrow$  NIL
```

**end-subalgorithm**

- Complexity:  $\Theta(1)$
- When we add or remove or search, we know that the list is empty if its head is NIL.

# DLL - Operations

- We can have the same operations on a DLL that we had on a SLL:
  - search for an element with a given value
  - add an element (to the beginning, to the end, to a given position, etc.)
  - delete an element (from the beginning, from the end, from a given positions, etc.)
  - get an element from a position
- Some of the operations have the exact same implementation as for SLL (e.g. search, get element), others have similar implementations. In general, we need to modify more links and have to pay attention to the *tail* node.

- Inserting a new element at the end of a DLL is simple, because we have the *tail* of the list, we do not have to walk through all the elements (like we have to do in case of a SLL).

**subalgorithm** insertLast(dll, elem) **is:**

//pre: dll is a DLL, elem is TElem

//post: elem is added to the end of dll

newNode ← allocate() //allocate a new DLLNode

[newNode].info ← elem

[newNode].next ← NIL

[newNode].prev ← dll.tail

**if** dll.head = NIL **then** //the list is empty

    dll.head ← newNode

    dll.tail ← newNode

**else**

    [dll.tail].next ← newNode

    dll.tail ← newNode

**end-if**

**end-subalgorithm**

- Complexity:

**subalgorithm** insertLast(dll, elem) **is:**

//pre: *dll* is a DLL, *elem* is TElem

//post: *elem* is added to the end of *dll*

newNode ← allocate() //allocate a new DLLNode

[newNode].info ← elem

[newNode].next ← NIL

[newNode].prev ← dll.tail

**if** dll.head = NIL **then** //the list is empty

    dll.head ← newNode

    dll.tail ← newNode

**else**

    [dll.tail].next ← newNode

    dll.tail ← newNode

**end-if**

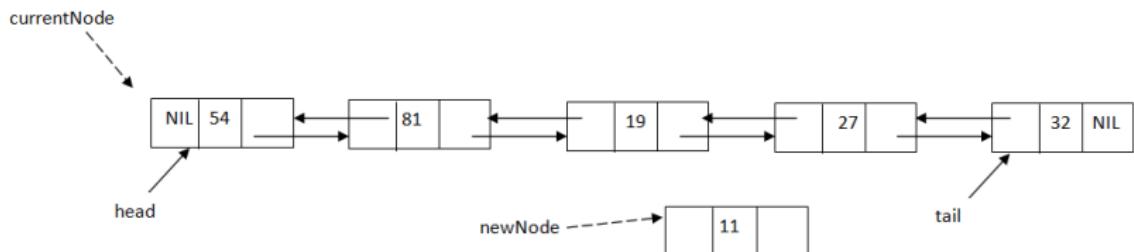
**end-subalgorithm**

- Complexity:  $\Theta(1)$

- The basic principle of inserting a new element at a given position is the same as in case of a SLL.
- The main difference is that we need to set more links (we have the *prev* links as well) and we have to check whether we modify the tail of the list.
- In case of a SLL we *had to* stop at the node after which we wanted to insert an element, in case of a DLL we can stop before or after the node (but we have to decide in advance, because this decision influences the special cases we need to test).

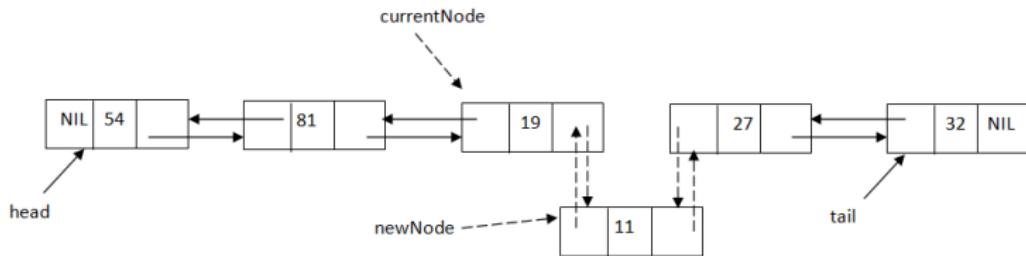
# DLL - Insert on position

- Let's insert value 46 at the 4<sup>th</sup> position in the following list:



# DLL - Insert on position

- We move with the *currentNode* to position 3, and set the 4 links.



# DLL - Insert at a position

```
subalgorithm insertPosition(dll, pos, elem) is:  
    //pre: dll is a DLL; pos is an integer number; elem is a TElem  
    //post: elem will be inserted on position pos in dll  
    if pos < 1 then  
        @ error, invalid position  
    else if pos = 1 then  
        insertFirst(dll, elem)  
    else  
        currentNode ← dll.head  
        currentPos ← 1  
        while currentNode ≠ NIL and currentPos < pos - 1 execute  
            currentNode ← [currentNode].next  
            currentPos ← currentPos + 1  
    end-while  
    //continued on the next slide...
```

# DLL - Insert at position

```
if currentNode = NIL then
    @error, invalid position
else if currentNode = dll.tail then
    insertLast(dll, elem)
else
    newNode ← alocate()
    [newNode].info ← elem
    [newNode].next ← [currentNode].next
    [newNode].prev ← currentNode
    [[currentNode].next].prev ← newNode
    [currentNode].next ← newNode
end-if
end-if
end-subalgorithm
```

- Complexitate:  $O(n)$

# DLL - Insert at a position

- Observations regarding the *insertPosition* subalgorithm:
  - We did not implement the *insertFirst* subalgorithm, but we suppose it exists.
  - The order in which we set the links is important: reversing the setting of the last two links will lead to a problem with the list.
  - It is possible to use two *currentNodes*: after we found the node after which we insert a new element, we can do the following:

```
nodeAfter ← currentNode
nodeBefore ← [currentNode].next
//now we insert between nodeAfter and nodeBefore
[newNode].next ← nodeBefore
[newNode].prev ← nodeAfter
[nodeBefore].prev ← newNode
[nodeAfter].next ← newNode
```

# DLL - Delete a given element

- If we want to delete a node with a given element, we first have to find the node:
  - we need to walk through the elements of the list until we find the node with the element
  - if we find the node, we delete it by modifying some links
  - special cases:

# DLL - Delete a given element

- If we want to delete a node with a given element, we first have to find the node:
  - we need to walk through the elements of the list until we find the node with the element
  - if we find the node, we delete it by modifying some links
  - special cases:
    - element not in list (includes the case with empty list)
    - remove head
    - remove head which is tail as well (one single element)
    - remove tail

```
function deleteElement(dll, elem) is:
    //pre: dll is a DLL, elem is a TElem
    //post: the node with element elem will be removed and returned
    currentNode ← dll.head
    while currentNode ≠ NIL and [currentNode].info ≠ elem execute
        currentNode ← [currentNode].next
    end-while
    deletedNode ← currentNode
    if currentNode ≠ NIL then
        if currentNode = dll.head then //remove the first node
            if currentNode = dll.tail then //which is the last one as well
                dll.head ← NIL
                dll.tail ← NIL
            else //list has more than 1 element, remove first
                dll.head ← [dll.head].next
                [dll.head].prev ← NIL
            end-if
        else if currentNode = dll.tail then
    //continued on the next slide...
```

# DLL - Delete a given element

```
dll.tail ← [dll.tail].prev
[dll.tail].next ← NIL
else
    [[currentNode].next].prev ← [currentNode].prev
    [[currentNode].prev].next ← [currentNode].next
    @set links of deletedNode to NIL to separate it from the
nodes of the list
end-if
end-if
deleteElement ← deletedNode
end-function
```

- Complexity:

# DLL - Delete a given element

```
dll.tail ← [dll.tail].prev
[dll.tail].next ← NIL
else
    [[currentNode].next].prev ← [currentNode].prev
    [[currentNode].prev].next ← [currentNode].next
    @set links of deletedNode to NIL to separate it from the
nodes of the list
end-if
end-if
deleteElement ← deletedNode
end-function
```

- Complexity:  $O(n)$

# Iterating through all the elements of a linked list

- Similar to the DynamicArray, if we want to go through all the elements of a (singly or doubly) linked list, we have two options:
  - Use an iterator
  - Use a for loop and the *getElement* subalgorithm
- What is the complexity of the two approaches?

# Dynamic Array vs. Linked Lists

- Advantages of Linked Lists

- No memory used for non-existing elements.
- Constant time operations at the beginning of the list.
- Elements are never *moved* (important if copying an element takes a lot of time).

- Disadvantages of Linked Lists

- We have no direct access to an element from a given position (however, iterating through all elements of the list using an iterator has  $\Theta(n)$  time complexity).
- Extra space is used up by the addresses stored in the nodes.
- Nodes are not stored at consecutive memory locations (no benefit from modern CPU caching methods).

# Algorithmic problems using Linked Lists

- Find the  $n^{th}$  node from the end of a SLL.

# Algorithmic problems using Linked Lists

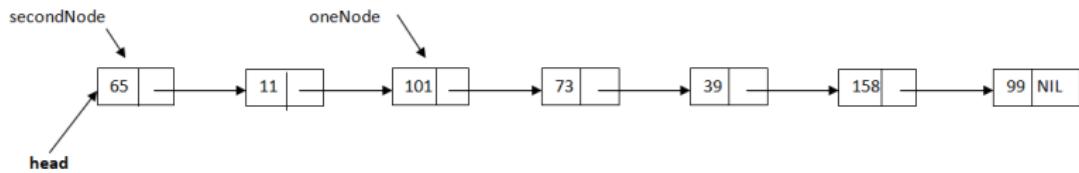
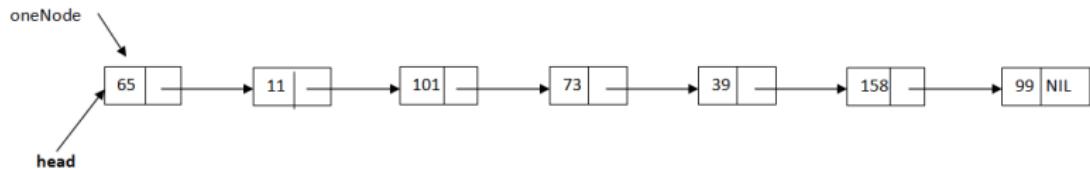
- Find the  $n^{th}$  node from the end of a SLL.
- Simple approach: go through all elements to count the length of the list. When we know the length, we know at which position the  $n^{th}$  node from the end is. Start again from the beginning and go to that position.
- Can we do it in one single pass over the list?

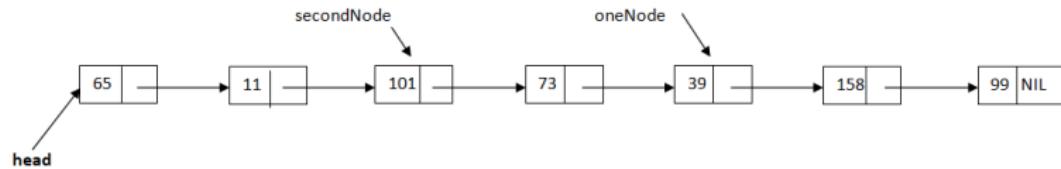
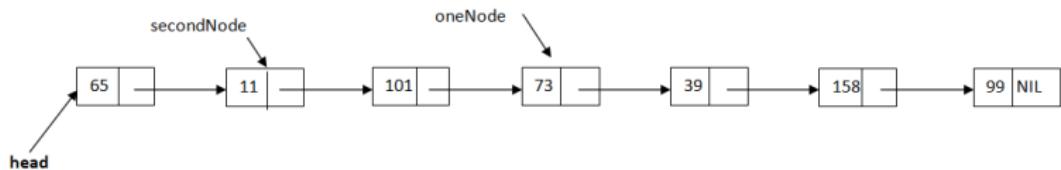
# Algorithmic problems using Linked Lists

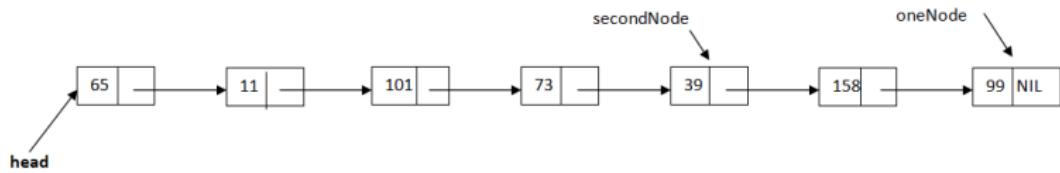
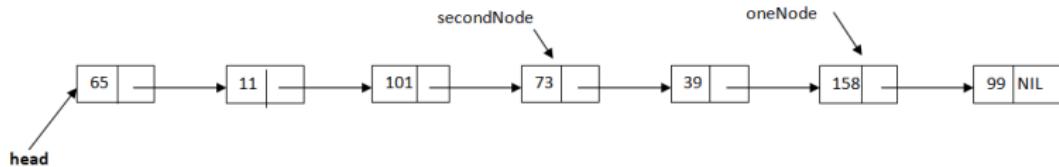
- Find the  $n^{th}$  node from the end of a SLL.
- Simple approach: go through all elements to count the length of the list. When we know the length, we know at which position the  $n^{th}$  node from the end is. Start again from the beginning and go to that position.
- Can we do it in one single pass over the list?
- We need to use two auxiliary variables, two nodes, both set to the first node of the list. At the beginning of the algorithm we will go forward  $n - 1$  times with one of the nodes. Once the first node is at the  $n^{th}$  position, we move with both nodes in parallel. When the first node gets to the end of the list, the second one is at the  $n^{th}$  element from the end of the list.

- We want to find the 3<sup>rd</sup> node from the end (the one with information 39)









# N-th node from the end of the list

```
function findNthFromEnd (sll, n) is:  
    //pre: sll is a SLL, n is an integer number  
    //post: the n-th node from the end of the list or NIL  
    oneNode ← sll.head  
    secondNode ← sll.head  
    position ← 1  
    while position < n and oneNode ≠ NIL execute  
        oneNode ← [oneNode].next  
        position ← position + 1  
    end-while  
    if oneNode = NIL then  
        findNthFromEnd ← NIL  
    else  
        //continued on the next slide...
```

# N-th node from the end of the list

```
while [oneNode].next ≠ NIL execute
```

```
    oneNode ← [oneNode].next
```

```
    secondNode ← [secondNode].next
```

```
end-while
```

```
findNthFromEnd ← secondNode
```

```
end-if
```

```
end-function
```

- Is this approach really better than the simple one (does it make fewer steps)?

- Write a subalgorithm which rotates a singly linked list (moves the first element to become the last one).

- Write a subalgorithm which rotates a singly linked list (moves the first element to become the last one).
  - We have to do two things: remove the first node and then attach it after the last one.
  - Special cases:

- Write a subalgorithm which rotates a singly linked list (moves the first element to become the last one).
  - We have to do two things: remove the first node and then attach it after the last one.
  - Special cases:
    - an empty list
    - list with a single node

**subalgorithm** rotate(sll) **is:**

```
if NOT (sll.head = NIL OR [sll.head].next = NIL) then
    first ← sll.head //save the first node
    sll.head ← [sll.head].next remove the first node
    current ← sll.head
    while [current].next ≠ NIL execute
        current ← [current].next
    end-while
    [current].next ← first
    [first].next ← NIL
    //make sure it does not point back to the new head node
end-if
end-subalgorithm
```

- Complexity:

**subalgorithm** rotate(sll) **is:**

```
if NOT (sll.head = NIL OR [sll.head].next = NIL) then
    first ← sll.head //save the first node
    sll.head ← [sll.head].next remove the first node
    current ← sll.head
    while [current].next ≠ NIL execute
        current ← [current].next
    end-while
    [current].next ← first
    [first].next ← NIL
    //make sure it does not point back to the new head node
end-if
end-subalgorithm
```

- Complexity:  $\Theta(n)$

- Given the first node of a SLL, determine whether the list ends with a node that has NIL as *next* or whether it ends with a cycle (the *last* node contains the address of a previous node as *next*).
- If the list from the previous problems contains a cycle, find the length of the cycle.
- Find if a SLL has an even or an odd number of elements, without counting the number of nodes in any way.
- Reverse a SLL non-recursively in linear time using  $\Theta(1)$  extra storage.

# Sorted Lists

- A *sorted list* (or ordered list) is a list in which the elements from the nodes are in a specific order, given by a *relation*.
- This *relation* can be  $<$ ,  $\leq$ ,  $>$  or  $\geq$ , but we can also work with an abstract relation.
- Using an abstract relation will give us more flexibility: we can easily change the relation (without changing the code written for the sorted list) and we can have, in the same application, lists with elements ordered by different relations.

# The relation

- You can imagine the *relation* as a function with two parameters (two *TComp* elems):

$$\text{relation}(c_1, c_2) = \begin{cases} \text{true}, & "c_1 \leq c_2" \\ \text{false}, & \text{otherwise} \end{cases}$$

- " $c_1 \leq c_2$ " means that  $c_1$  should be in front of  $c_2$  when ordering the elements.

# Sorted List - representation

- When we have a sorted list (or any sorted structure or container) we will keep the relation used for ordering the elements as part of the structure. We will have a field that represents this relation.
- In the following we will talk about a *sorted singly linked list* (representation and code for a *sorted doubly linked list* is really similar).

# Sorted List - representation

- We need two structures: *Node* - *SSLLNode* and *Sorted Singly Linked List* - *SSLL*

## SSLLNode:

info: TComp  
next:  $\uparrow$  SSLLNode

## SSLL:

head:  $\uparrow$  SSLLNode  
rel:  $\uparrow$  Relation

- The relation is passed as a parameter to the *init* function, the function which initializes a new SSLL.
- In this way, we can create multiple SSLLs with different relations.

**subalgorithm** init (ssll, rel) **is:**

```
//pre: rel is a relation  
//post: ssll is an empty SSLL  
ssll.head ← NIL  
ssll.rel ← rel  
end-subalgorithm
```

- Complexity:  $\Theta(1)$

- Since we have a singly-linked list we need to find the node *after* which we insert the new element (otherwise we cannot set the links correctly).
- The node we want to insert after is the first node whose successor is *greater than* the element we want to insert (where *greater than* is represented by the value *false* returned by the relation).
- We have two special cases:
  - an empty SSLL list
  - when we insert before the first node

# SSLL - insert

**subalgorithm** insert (ssll, elem) **is:**

//pre: ssll is a SSLL; elem is a TComp

//post: the element elem was inserted into ssll to where it belongs

newNode ← allocate()

[newNode].info ← elem

[newNode].next ← NIL

**if** ssll.head = NIL **then**

//the list is empty

ssll.head ← newNode

**else if** ssll.rel(elem, [ssll.head].info) **then**

//elem is "less than" the info from the head

[newNode].next ← ssll.head

ssll.head ← newNode

**else**

//continued on the next slide...

# SSLL - insert

```
cn ← ssll.head //cn - current node
while [cn].next ≠ NIL and ssll.rel(elem, [[cn].next].info) = false execute
    cn ← [cn].next
end-while
//now insert after cn
[newNode].next ← [cn].next
[cn].next ← newNode
end-if
end-subalgorithm
```

- Complexity:

# SSLL - insert

```
cn ← ssll.head //cn - current node
while [cn].next ≠ NIL and ssll.rel(elem, [[cn].next].info) = false execute
    cn ← [cn].next
end-while
//now insert after cn
[newNode].next ← [cn].next
[cn].next ← newNode
end-if
end-subalgorithm
```

- Complexity:  $O(n)$

## SSLL - Other operations

- The search operation is identical to the search operation for a SLL (except that we can stop looking for the element when we get to the first element that is "greater than" the one we are looking for).
- The delete operations are identical to the same operations for a SLL.
- The return an element from a position operation is identical to the same operation for a SLL.
- The iterator for a SSLL is identical to the iterator to a SLL.

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 7

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

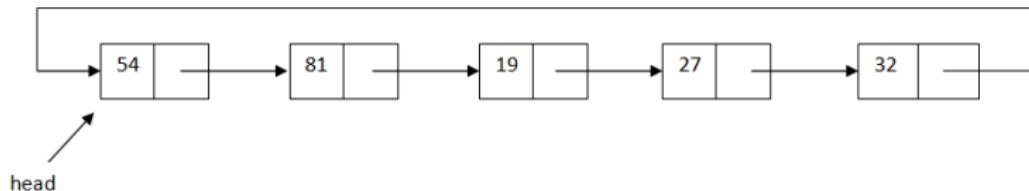
# In Lecture 6...

- Linked Lists
- Sorted Linked List

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

# Circular Lists

- For a SLL or a DLL the last node has as *next* the value *NIL*.  
In a *circular list* no node has *NIL* as next, since the last node contains the address of the first node in its next field.



- We can have singly linked and doubly linked circular lists, in the following we will discuss the singly linked version.
- In a circular list each node has a successor, and we can say that the list does not have an end.
- We have to be careful when we iterate through a circular list, because we might end up with an infinite loop (if we set as stopping criterion the case when *currentNode* or *[currentNode].next* is *NIL*).
- There are problems where using a circular list makes the solution simpler (for example: Josephus circle problem, rotation of a list)

- Operations for a circular list have to consider the following two important aspects:
  - The *last* node of the list is the one whose *next* field is the *head* of the list.
  - Inserting before the head, or removing the head of the list, is no longer a simple  $\Theta(1)$  complexity operation, because we have to change the *next* field of the last node as well (and for this we have to find the last node).

# Circular Lists - Representation

- The representation of a circular list is exactly the same as the representation of a simple SLL. We have a structure for a *Node* and a structure for the *Circular Singly Linked Lists - CSLL*.

## CSLLNode:

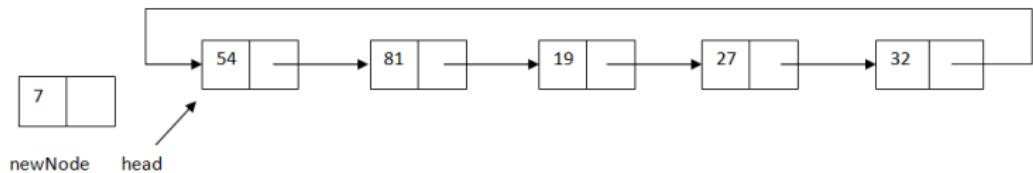
info: TElem

next:  $\uparrow$  CSLLNode

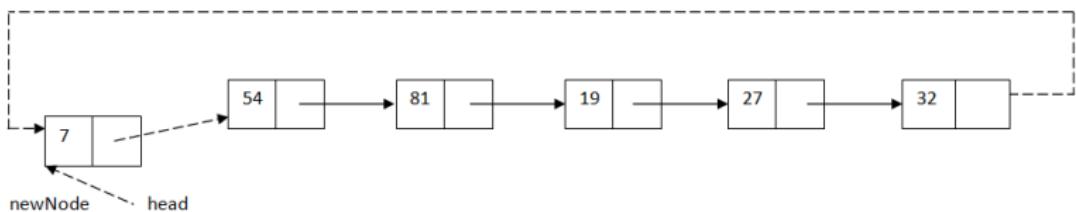
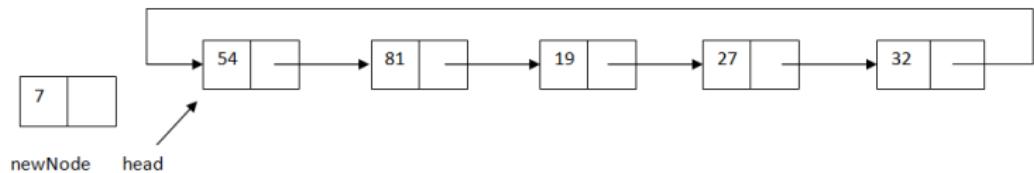
## CSLL:

head:  $\uparrow$  CSLLNode

# CSLL - InsertFirst



# CSLL - InsertFirst



# CSLL - InsertFirst

**subalgorithm** insertFirst (csll, elem) **is:**

//pre: csll is a CSLL, elem is a TElem

//post: the element elem is inserted at the beginning of csll

newNode ← allocate()

[newNode].info ← elem

[newNode].next ← newNode

**if** csll.head = NIL **then**

    csll.head ← newNode

**else**

    lastNode ← csll.head

**while** [lastNode].next ≠ csll.head **execute**

        lastNode ← [lastNode].next

**end-while**

//continued on the next slide...

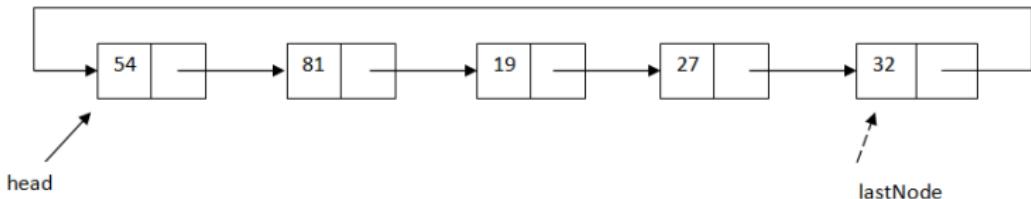
```
[newNode].next ← csll.head  
[lastNode].next ← newNode  
csll.head ← newNode
```

**end-if**

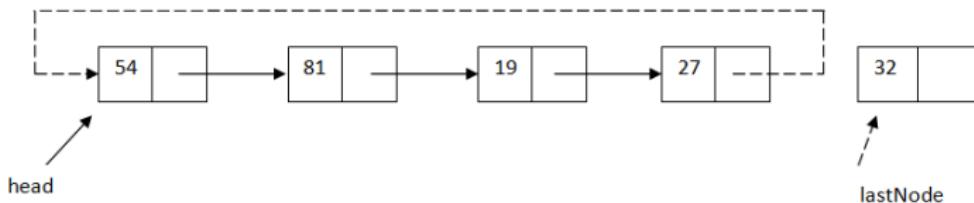
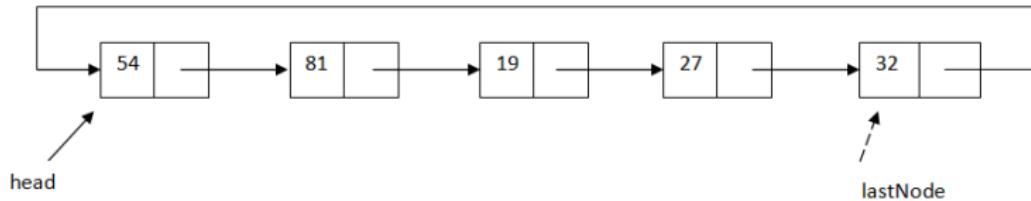
**end-subalgorithm**

- Complexity:  $\Theta(n)$
- Note: inserting a new element at the end of a circular list looks exactly the same, but we do not modify the value of *csll.head* (so the last instruction is not needed).

# CSLL - DeleteLast



# CSLL - DeleteLast



# CSLL - DeleteLast

```
function deleteLast(csll) is:  
    //pre: csll is a CSLL  
    //post: the last element from csll is removed and the node  
    //containing it is returned  
    deletedNode ← NIL  
    if csll.head ≠ NIL then  
        if [csll.head].next = csll.head then  
            deletedNode ← csll.head  
            csll.head ← NIL  
        else  
            prevNode ← csll.head  
            while [[prevNode].next].next ≠ csll.head execute  
                prevNode ← [prevNode].next  
            end-while  
    //continued on the next slide...
```

```
    deletedNode ← [prev].next
    [prev].next ← csll.head
end-if
end-if
[deletedNode].next ← NIL
deleteLast ← deletedNode
end-function
```

- Complexity:  $\Theta(n)$

- How can we define an iterator for a CSLL? What do you think is the most challenging part of implementing the iterator?

- How can we define an iterator for a CSLL? What do you think is the most challenging part of implementing the iterator?
- The main problem with the *standard* SLL iterator is its *valid* method. For a SLL *valid* returns false, when the value of the *currentElement* becomes *NIL*. But in case of a circular list, *currentElement* will never be *NIL*.
- We have finished iterating through all elements when the value of *currentElement* becomes equal to the *head* of the list.
- However, writing that the iterator is invalid when *currentElement* equals the *head*, will produce an iterator which is invalid the moment it was created.

- We can say that the iterator is invalid, when the *next* of the *currentElement* is equal to the *head* of the list.
- This will stop the iterator when it is set to the last element of the list, so if we want to print all the elements from a list, we have to call the *element* operation one more time when the iterator becomes invalid (or use a do-while loop instead of a while loop - but this causes problems when we iterate through an empty list).
- As a second problem, this violates the precondition that element should only be called when the iterator is valid.

- We can add a boolean flag to the iterator besides the *currentElement*, something that shows whether we are at the *head* for the first time (when the iterator was created), or whether we got back to the *head* after going through all the elements.
- For this version, standard iteration code remains the same.

- Similarly, if the CSLL contains a field for the size of the list, we can add a counter in the iterator (besides the current node), which counts how many times we called next. If it is equal to the size + 1, the iterator is invalid. It is a combination of how we represent current element for a dynamic array and a linked list.
- For this version, standard iteration code remains the same.

- Depending on the problem we want to solve, we might need a read/write iterator: one that can be used to change the content of the CSLL.
- We can have *insertAfter* - insert a new element after the current node - and *delete* - delete the current node
- We can say that the iterator is invalid when there are no elements in the circular list (especially if we delete from it).

# The Josephus circle problem

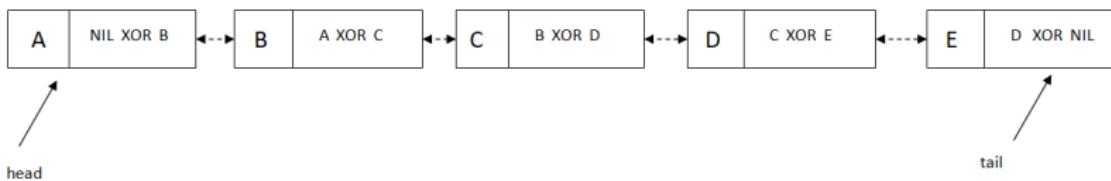
- There are  $n$  men standing a circle waiting to be executed. Starting from one person we start counting into clockwise direction and execute the  $m^{th}$  person. After the execution we restart counting with the person after the executed one and execute again the  $m^{th}$  person. The process is continued until only one person remains: this person is freed.
- Given the number of men,  $n$ , and the number  $m$ , determine which person will be freed.
- For example, if we have 5 men and  $m = 3$ , the  $4^{th}$  man will be freed.

# Circular Lists - Variations

- There are different possible variations for a circular list that can be useful, depending on what we use the circular list for.
  - Instead of retaining the *head* of the list, retain its *tail*. In this way, we have access both to the *head* and the *tail*, and can easily insert before the head or after the tail. Deleting the head is simple as well, but deleting the tail still needs  $\Theta(n)$  time.
  - Use a *header* or *sentinel* node - a special node that is considered the *head* of the list, but which cannot be deleted or changed - it is simply a separation between the head and the tail. For this version, knowing when to stop with the iterator is easier.

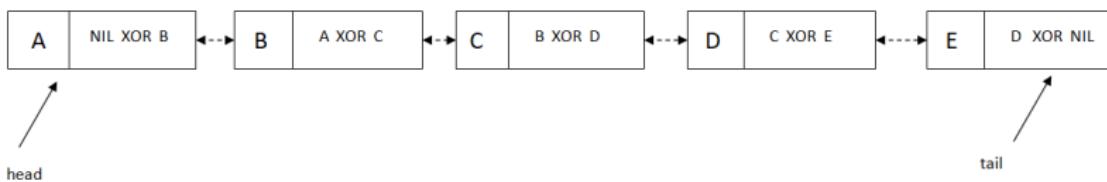
- Doubly linked lists are better than singly linked lists because they offer better complexity for some operations
- Their disadvantage is that they occupy more memory, because you have two links to memorize, instead of just one.
- A memory-efficient solution is to have a *XOR Linked List*, which is a doubly linked list (we can traverse it in both directions), where every node retains one single link, which is the XOR of the previous and the next node.

# XOR Linked List - Example



- How do you traverse such a list?

# XOR Linked List - Example



- How do you traverse such a list?

- We start from the head (but we can have a backward traversal starting from the tail in a similar manner), the node with A
- The address from node A is directly the address of node B ( $\text{NIL XOR } B = B$ )
- When we have the address of node B, its link is  $A \text{ XOR } C$ . To get the address of node C, we have to XOR B's link with the address of A (it's the previous node we come from):  $A \text{ XOR } C \text{ XOR } A = A \text{ XOR } A \text{ XOR } C = \text{NIL XOR } C = C$

# XOR Linked List - Representation

- We need two structures to represent a XOR Linked List: one for a node and one for the list

## XORNode:

info: TELem

link:  $\uparrow$  XORNode

## XORList:

head:  $\uparrow$  XORNode

tail:  $\uparrow$  XORNode

# XOR Linked List - Traversal

**subalgorithm** printListForward(*xorl*) **is:**

//pre: *xorl* is a XORList

//post: true (the content of the list was printed)

prevNode  $\leftarrow$  NIL

currentNode  $\leftarrow$  xorl.head

**while** currentNode  $\neq$  NIL **execute**

**write** [currentNode].info

    nextNode  $\leftarrow$  prevNode XOR [currentNode].link

    prevNode  $\leftarrow$  currentNode

    currentNode  $\leftarrow$  nextNode

**end-while**

**end-subalgorithm**

- Complexity:  $\Theta(n)$

# XOR Linked List - addToBeginning

- How can we add an element to the beginning of the list?

# XOR Linked List - addToBeginning

- How can we add an element to the beginning of the list?

**subalgorithm** addToBeginning(xorl, elem) **is:**

//pre: xorl is a XORList

//post: a node with info elem was added to the beginning of the list

newNode ← allocate()

[newNode].info ← elem

[newNode].link ← xorl.head

**if** xorl.head = NIL **then**

    xorl.head ← newNode

    xorl.tail ← newNode

**else**

    [xorl.head].link ← [xorl.head].link XOR newNode

    xorl.head ← newNode

**end-if**

**end-subalgorithm**

- Complexity:

# XOR Linked List - addToBeginning

- How can we add an element to the beginning of the list?

**subalgorithm** addToBeginning(xorl, elem) **is:**

//pre: xorl is a XORList

//post: a node with info elem was added to the beginning of the list

newNode ← allocate()

[newNode].info ← elem

[newNode].link ← xorl.head

**if** xorl.head = NIL **then**

    xorl.head ← newNode

    xorl.tail ← newNode

**else**

    [xorl.head].link ← [xorl.head].link XOR newNode

    xorl.head ← newNode

**end-if**

**end-subalgorithm**

- Complexity:  $\Theta(1)$

- Assume that we want to memorize a sequence of sorted elements. The elements can be stored in:
  - dynamic array
  - linked list (let's say doubly linked list)
- We know that the most frequently used operation will be the insertion of a new element, so we want to choose a data structure for which insertion has the best complexity. Which one should we choose?

- We can divide the insertion operation into two steps: *finding where to insert* and *inserting the elements*

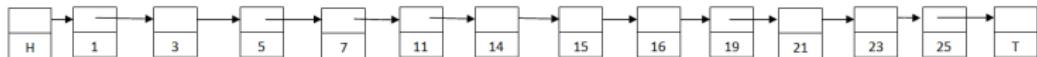
- We can divide the insertion operation into two steps: *finding where to insert* and *inserting the elements*
  - For a dynamic array finding the position can be optimized (binary search  $O(\log_2 n)$ ), but the insertion is  $O(n)$
  - For a linked list the insertion is optimal ( $\Theta(1)$ ), but finding where to insert is  $O(n)$

# Skip List

- A skip list is a data structure that allows *fast search* in an ordered sequence.
- How can we do that?

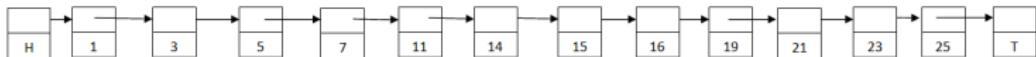
# Skip List

- A skip list is a data structure that allows *fast search* in an ordered sequence.
- How can we do that?



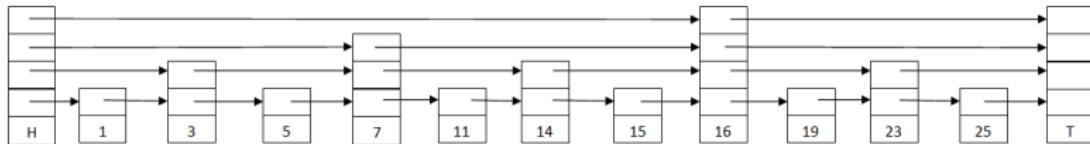
# Skip List

- A skip list is a data structure that allows *fast search* in an ordered sequence.
- How can we do that?



- Starting from an ordered linked list, we add to every second node another pointer that skips over one element.
- We add to every fourth node another pointer that skips over 3 elements.
- etc.

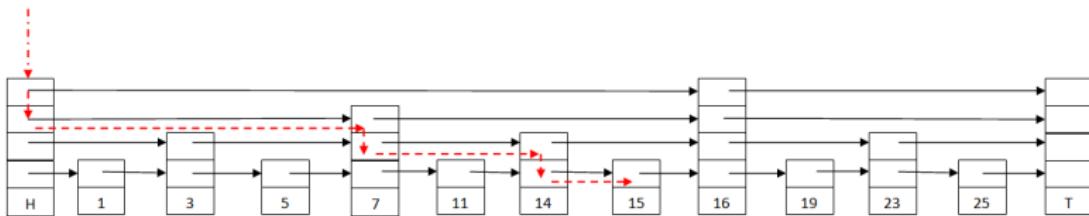
# Skip List



- H and T are two special nodes, representing *head* and *tail*. They cannot be deleted, they exist even in an empty list.

# Skip List - Search

- Search for element 15.

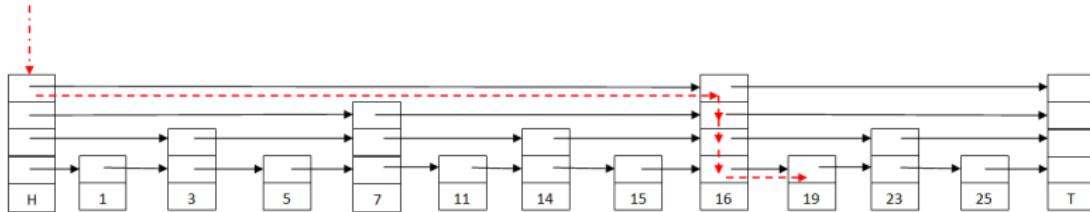


- Start from head and from highest level.
- If possible, go right.
- If cannot go right (next element is greater), go down a level.

- Lowest level has all  $n$  elements.
- Next level has  $\frac{n}{2}$  elements.
- Next level has  $\frac{n}{4}$  elements.
- etc.
- $\Rightarrow$  there are approx  $\log_2 n$  levels.
- From each level, we check at most 2 nodes.
- Complexity of search:  $O(\log_2 n)$

## Skip List - Insert

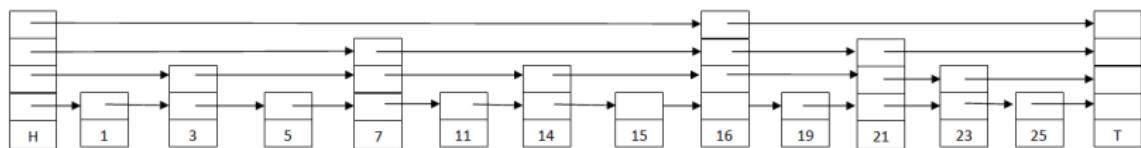
- Insert element 21.



- How *high* should the new node be?

# Skip List - Insert

- Height of a new node is determined *randomly*, but in such a way that approximately half of the nodes will be on level 2, a quarter of them on level 3, etc.



- Assume we randomly generate the height 3 for the node with 21.

- Skip Lists are *probabilistic* data structures, since we decide randomly the height of a newly inserted node.
- There might be a worst case, where every node has height 1 (so it is just a linked list).
- In practice, they function well.

# Linked Lists on Arrays

- What if we need a linked list, but we are working in a programming language that does not offer pointers (or references)?
- We can still implement linked data structures, without the explicit use of pointers or memory addresses, simulating them using arrays and array indexes.

# Linked Lists on Arrays

- Usually, when we work with arrays, we store the elements in the array starting from the leftmost position and place them one after the other (no empty spaces in the middle of the list are allowed).
- The order of the elements is given by the order in which they are placed in the array.

elems	46	78	11	6	59	19				
-------	----	----	----	---	----	----	--	--	--	--

- Order of the elements: 46, 78, 11, 6, 59, 19

# Linked Lists on Arrays

- We can define a linked data structure on an array, if we consider that the order of the elements is not given by their relative positions in the array, but by an integer number associated with each element, which shows the index of the next element in the array (thus we have a singly linked list).

elems	46	78	11	6	59	19			
next	5	6	1	-1	2	4			

head = 3

- Order of the elements: 11, 46, 59, 78, 19, 6

# Linked Lists on Arrays

- Now, if we want to delete the number 46 (which is actually the second element of the list), we do not have to move every other element to the left of the array, we just need to modify the links:

elems		78	11	6	59	19			
next		6	5	-1	2	4			

head = 3

- Order of the elements: 11, 59, 78, 19, 6

# Linked Lists on Arrays

- If we want to insert a new element, for example 44, at the 3<sup>rd</sup> position in the list, we can put the element anywhere in the array, the important part is setting the links correctly:

elems		78	11	6	59	19		44		
next		6	5	-1	8	4		2		

head = 3

- Order of the elements: 11, 59, 44, 78, 19, 6

# Linked Lists on Arrays

- When a new element needs to be inserted, it can be put to any empty position in the array. However, finding an empty position has  $O(n)$  complexity, which will make the complexity of any insert operation (anywhere in the list)  $O(n)$ . In order to avoid this, we will keep a linked list of the empty positions as well.

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

head = 3

firstEmpty = 1

# Linked Lists on Arrays

- In a more formal way, we can simulate a singly linked list on an array with the following:
  - an array in which we will store the elements.
  - an array in which we will store the links (indexes to the next elements).
  - the capacity of the arrays (the two arrays have the same capacity, so we need only one value).
  - an index to tell where the *head* of the list is.
  - an index to tell where the first empty position in the array is.

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

## SLLA:

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

- We can implement for a SLLA any operation that we can implement for a SLL:
  - insert at the beginning, end, at a position, before/after a given value
  - delete from the beginning, end, from a position, a given element
  - search for an element
  - get an element from a position

```
subalgorithm init(slla) is:  
//pre: true; post: slla is an empty SLLA  
slla.cap ← INIT_CAPACITY
```

```
subalgorithm init(slla) is:
    //pre: true; post: slla is an empty SLLA
    slla.cap ← INIT_CAPACITY
    slla.elems ← @an array with slla.cap positions
    slla.next ← @an array with slla.cap positions
    slla.head ← -1
    for i ← 1, slla.cap-1 execute
        slla.next[i] ← i + 1
    end-for
    slla.next[slla.cap] ← -1
    slla.firstEmpty ← 1
end-subalgorithm
```

- Complexity:

```
subalgorithm init(slla) is:
    //pre: true; post: slla is an empty SLLA
    slla.cap ← INIT_CAPACITY
    slla.elems ← @an array with slla.cap positions
    slla.next ← @an array with slla.cap positions
    slla.head ← -1
    for i ← 1, slla.cap-1 execute
        slla.next[i] ← i + 1
    end-for
    slla.next[slla.cap] ← -1
    slla.firstEmpty ← 1
end-subalgorithm
```

- Complexity:  $\Theta(n)$  -where n is the initial capacity

```
function search (slla, elem) is:  
    //pre: slla is a SLLA, elem is a TElem  
    //post: return True if elem is in slla, False otherwise  
    current ← slla.head  
    while current ≠ -1 and slla.elems[current] ≠ elem execute  
        current ← slla.next[current]  
    end-while  
    if current ≠ -1 then  
        search ← True  
    else  
        search ← False  
    end-if  
end-function
```

- Complexity:

```
function search (slla, elem) is:  
    //pre: slla is a SLLA, elem is a TElem  
    //post: return True if elem is in slla, False otherwise  
    current ← slla.head  
    while current ≠ -1 and slla.elems[current] ≠ elem execute  
        current ← slla.next[current]  
    end-while  
    if current ≠ -1 then  
        search ← True  
    else  
        search ← False  
    end-if  
end-function
```

- Complexity:  $O(n)$

- From the *search* function we can see how we can go through the elements of a SLLA (and how similar this traversal is to the one done for a SLL):
  - We need a *current* element used for traversal, which is initialized to the index of the *head* of the list.
  - We stop the traversal when the value of *current* becomes -1
  - We go to the next element with the instruction: *current*  $\leftarrow$  *slla.next[current]*.

```
subalgorithm insertFirst(slla, elem) is:
    //pre: slla is an SLLA, elem is a TElem
    //post: the element elem is added at the beginning of slla
    if slla.firstEmpty = -1 then
        newElems ← @an array with slla.cap * 2 positions
        newNext ← @an array with slla.cap * 2 positions
        for i ← 1, slla.cap execute
            newElems[i] ← slla.elems[i]
            newNext[i] ← slla.next[i]
        end-for
        for i ← slla.cap + 1, slla.cap*2 - 1 execute
            newNext[i] ← i + 1
        end-for
        newNext[slla.cap*2] ← -1
    //continued on the next slide...
```

```
//free slla.elems and slla.next if necessary
slla.elems ← newElems
slla.next ← newNext
slla.firstEmpty ← slla.cap+1
slla.cap ← slla.cap * 2
end-if
newPosition ← slla.firstEmpty
slla.elems[newPosition] ← elem
slla.firstEmpty ← slla.next[slla.firstEmpty]
slla.next[newPosition] ← slla.head
slla.head ← newPosition
end-subalgorithm
```

- Complexity:

```
//free slla.elems and slla.next if necessary
slla.elems ← newElems
slla.next ← newNext
slla.firstEmpty ← slla.cap+1
slla.cap ← slla.cap * 2
end-if
newPosition ← slla.firstEmpty
slla.elems[newPosition] ← elem
slla.firstEmpty ← slla.next[slla.firstEmpty]
slla.next[newPosition] ← slla.head
slla.head ← newPosition
end-subalgorithm
```

- Complexity:  $\Theta(1)$  amortized

# SLLA - DeleteElement

```
subalgorithm deleteElement(slla, elem) is:
    //pre: slla is a SLLA; elem is a TElem
    //post: the element elem is deleted from SLLA
    nodC ← slla.head
    prevNode ← -1
    while nodC ≠ -1 and slla.elems[nodC] ≠ elem execute
        prevNode ← nodC
        nodC ← slla.next[nodC]
    end-while
    if nodC ≠ -1 then
        if nodC = slla.head then
            slla.head ← slla.next[slla.head]
        else
            slla.next[prevNode] ← slla.next[nodC]
        end-if
    //continued on the next slide...
```

```
//add the nodC position to the list of empty spaces
    slla.next[nodC] ← slla.firstEmpty
    slla.firstEmpty ← nodC
else
    @the element does not exist
end-if
end-subalgorithm
```

- Complexity:  $O(n)$

- Iterator for a SSLA is a combination of an iterator for an array and of an iterator for a singly linked list:
- Since the elements are stored in an array, the *currentElement* will be an index from the array.
- But since we have a linked list, going to the next element will not be done by incrementing the *currentElement* by one; we have to follow the *next* links.
- Also, initialization will be done with the position of the head, not position 1.

- 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

- 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

- 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
        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
```

**subalgorithm** free (dll, poz) **is:**

//pre: *dll* is a DLLA, *poz* is an integer number

//post: the position *poz* was freed

*dll*.nodes[*poz*].next ← *dll*.firstEmpty

*dll*.nodes[*poz*].prev ← -1

**if** *dll*.firstEmpty ≠ -1 **then**

*dll*.nodes[*dll*.firstEmpty].prev ← *poz*

**end-if**

*dll*.firstEmpty ← *poz*

**end-subalgorithm**

# DLLA - InsertPosition

```
subalgorithm insertPosition(dlla, elem, poz) is:
    //pre: dlla is a DLLA, elem is a TElm, 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 - InsertPosition

```
    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:  $O(n)$

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

DLLAIterator:

list: DLLA

currentElement: Integer

**subalgorithm** init(it, dlla) **is:**

//pre: *dlla* is a DLLA

//post: it is a DLLAIterator for *dlla*

it.list  $\leftarrow$  *dlla*

it.currentElement  $\leftarrow$  *dlla.head*

**end-subalgorithm**

- For a (dynamic) array, currentElement is set to 0 when an iterator is created. For a DLLA we need to set it to the head of the list (which might be position 0, but it might be a different position as well).
- Complexity:

**subalgorithm** init(it, dlla) **is:**

//pre: *dlla* is a DLLA

//post: it is a DLLAIterator for *dlla*

it.list  $\leftarrow$  *dlla*

it.currentElement  $\leftarrow$  *dlla.head*

**end-subalgorithm**

- For a (dynamic) array, currentElement is set to 0 when an iterator is created. For a DLLA we need to set it to the head of the list (which might be position 0, but it might be a different position as well).
- Complexity:  $\Theta(1)$

## DLLAliterator - getCurrent

**subalgorithm** getCurrent(it) **is:**

//pre: it is a DLLAliterator, 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:

**subalgorithm** getCurrent(it) **is:**

//pre: it is a DLLAliterator, 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:  $\Theta(1)$

**subalgorithm** next (it) **is:**

//pre: it is a DLLAliterator, 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 a (dynamic) array, going to the next element means incrementing the *currentElement* by one. For a DLLA we need to follow the links.
- Complexity:

**subalgorithm** next (it) **is:**

//pre: it is a DLLAliterator, 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  $\leftarrow$  it.list.nodes[it.currentElement].next

**end-subalgorithm**

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

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

- Complexity:

## DLLAliterator - valid

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

- Complexity:  $\Theta(1)$

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 8

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

# In Lecture 7...

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

- 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?

# 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*  $\leftarrow$  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.

- 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.

- 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: **Last In, First Out** (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

# Stack - Representation on DLL

- 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.

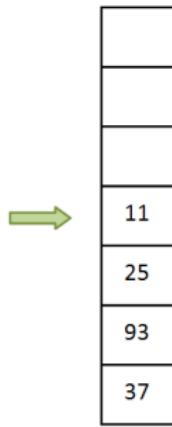
# GetMinimum in constant time

- 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)?

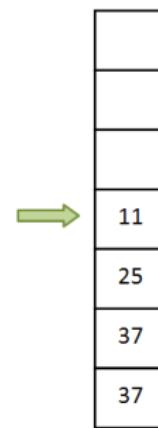
- How can we design a *special stack* that has a *getMinimum* operation with  $\Theta(1)$  time complexity (and the other operations have  $\Theta(1)$  time complexity as well)?
- We can keep an auxiliary stack, containing as many elements as the original stack, but containing the minimum value up to 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*:

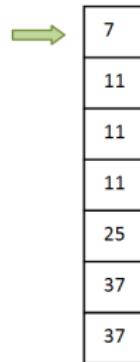
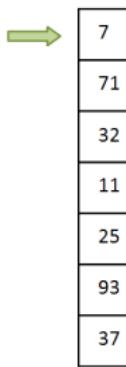


- This is the corresponding *min stack*:



# 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 is 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).

# GetMinimum in constant time

- 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)
```

- 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*?

- 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?

# 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?

# 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?

# Queue - representation on a DLL

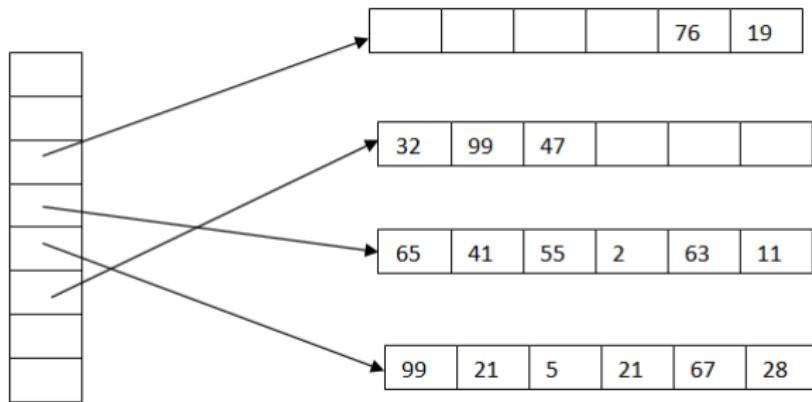
- If we want to implement a Queue using a doubly linked list, where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the list and *rear* at the end
  - Put *front* at the end of the list and *rear* at the beginning

- 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.

- Possible (good) representations for a Deque:
  - Circular Array
  - Doubly Linked List
  - A dynamic array of constant size arrays

- 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

# 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.

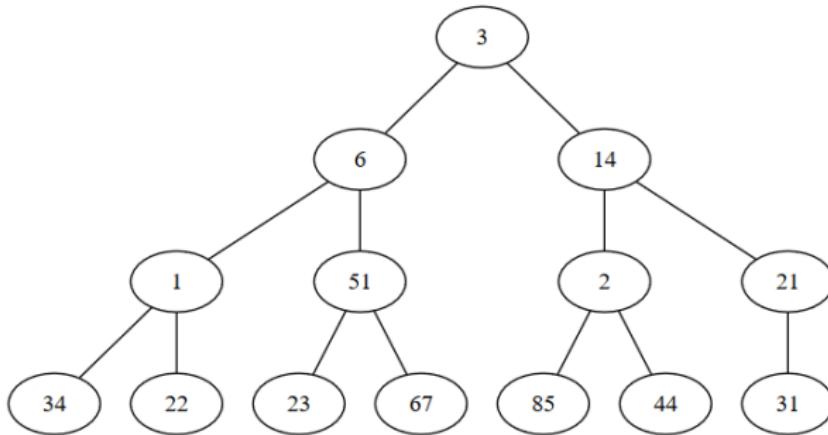
# Binary Heap

- 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

# Binary Heap

- 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

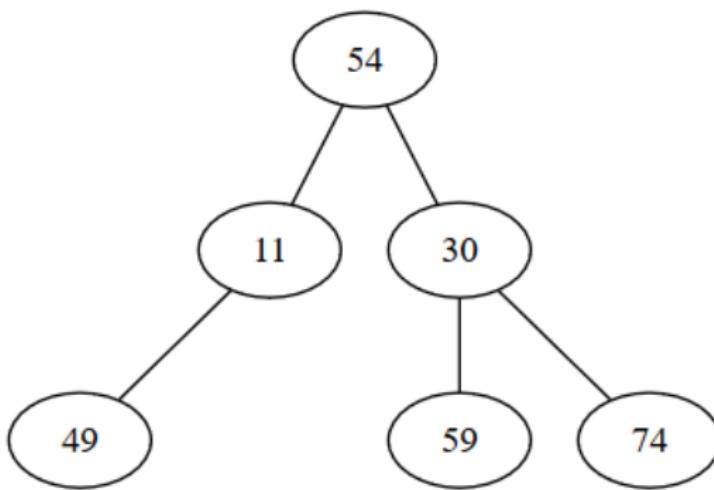
- If the elements of the array are:  $a_1, a_2, a_3, \dots, a_n$ , we know that:
  - $a_1$  is the root of the heap
  - for an element from position  $i$ , its children are on positions  $2 * i$  and  $2 * i + 1$  (if  $2 * i$  and  $2 * i + 1$  is less than or equal to  $n$ )
  - for an element from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

# Binary Heap

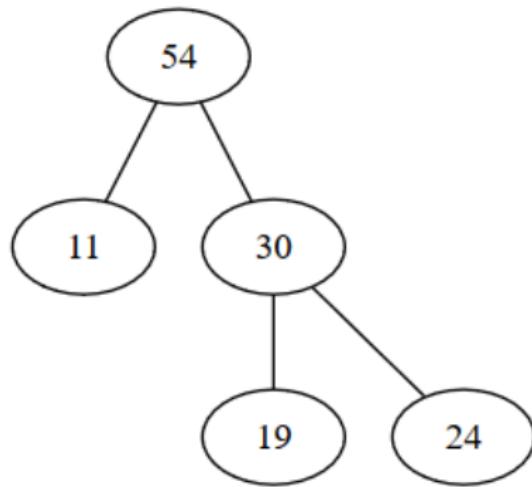
- A *binary heap* is an array that can be visualized as a binary tree having a *heap structure* and a *heap property*.
- *Heap structure*: in the binary tree every node has exactly 2 children, except for the last two levels, where children are completed from left to right.
- *Heap property*:  $a_i \geq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

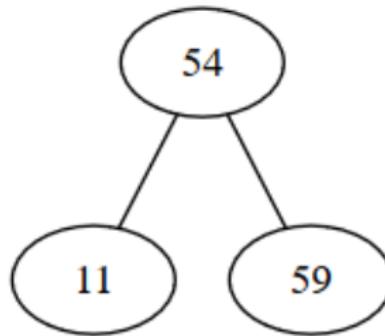
- Are the following binary trees heaps? If yes, specify the relation between a node and its children. If not, specify if the problem is with the structure, the property, or both.



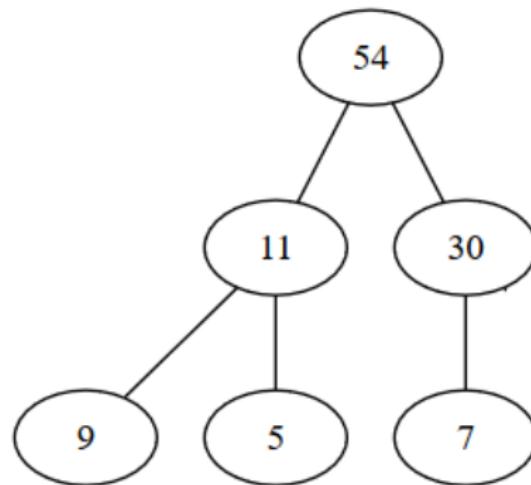
# Binary Heap - Examples II



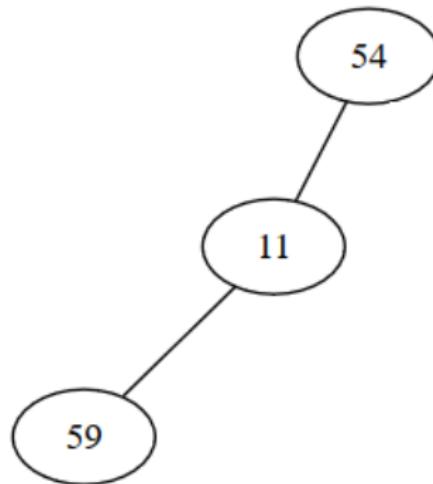
# Binary Heap - Examples III



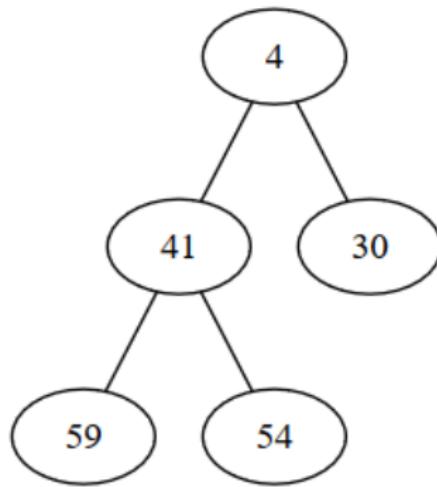
# Binary Heap - Examples IV



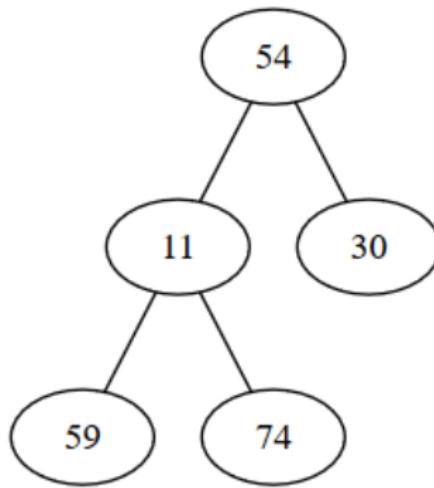
## Binary Heap - Examples V



## 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.
  - [70, 10, 50, 7, 1, 33, 3, 8]
  - [1, 2, 4, 8, 16, 32, 64, 65, 10]
  - [10, 12, 100, 60, 13, 102, 101, 80, 90, 14, 15, 16]

# Binary Heap - Notes

- 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  $\log_2 n$ .

# 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

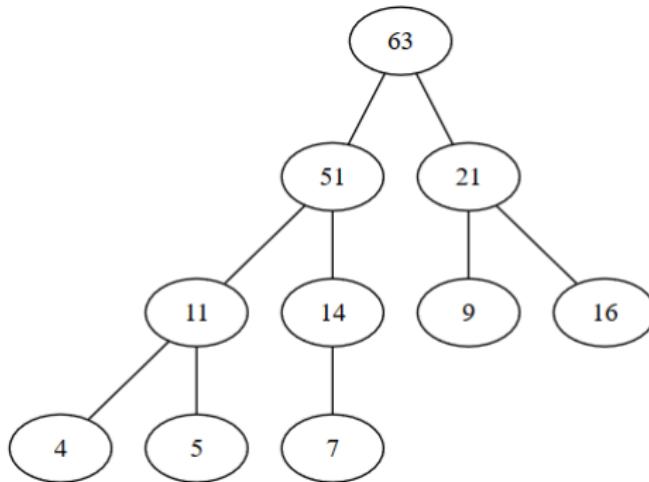
len: Integer

elems: TElem[]

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

## Binary Heap - Add - example

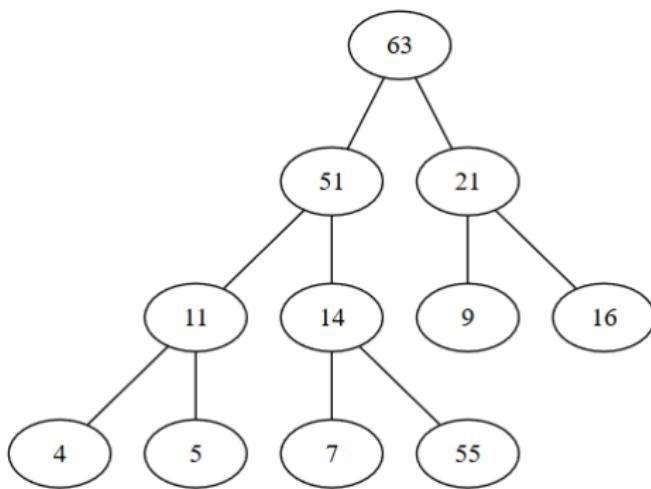
- Consider the following (MAX) heap:



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

## Binary Heap - Add - example

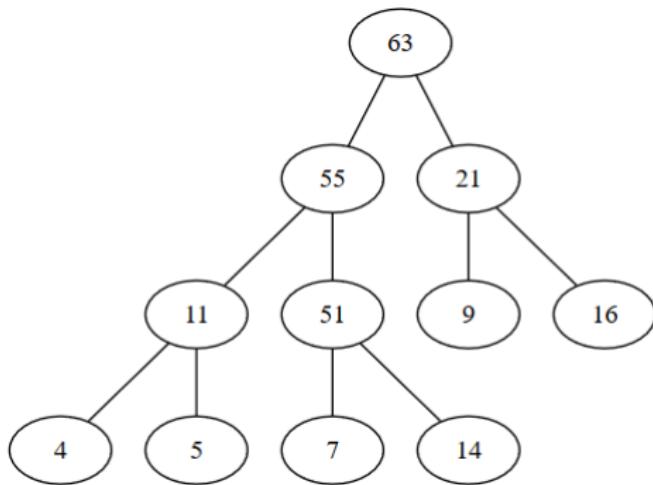
- In order to keep the *heap structure*, we will add the new node as the right child of the node 14 (and as the last element of the array in which the elements are kept).



- *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**

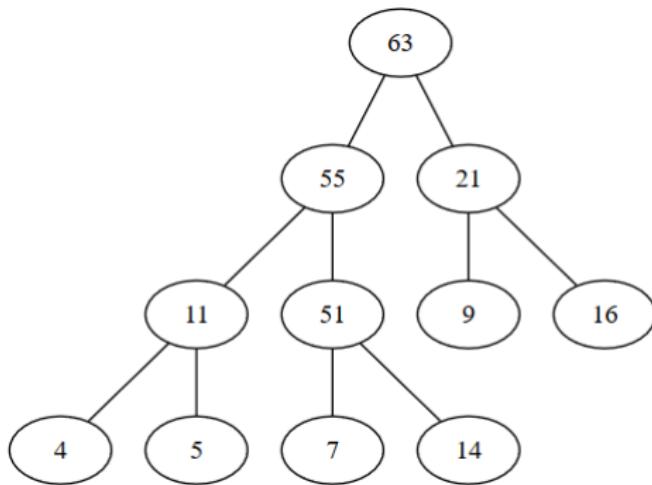
heap.elems[poz] ← elem

**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)?

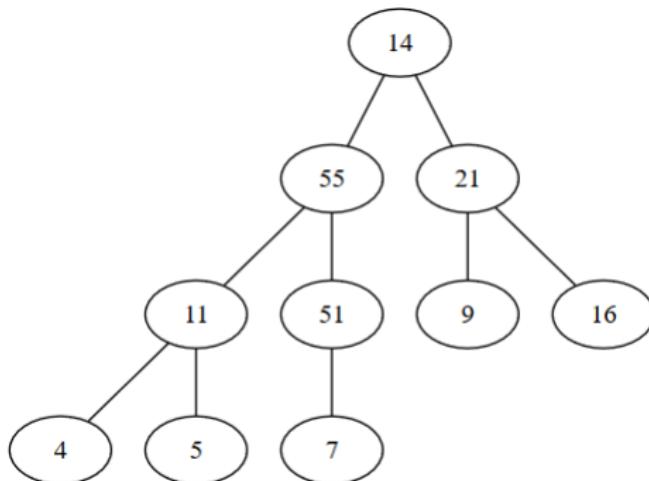
# Binary Heap - Remove - example

- 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.

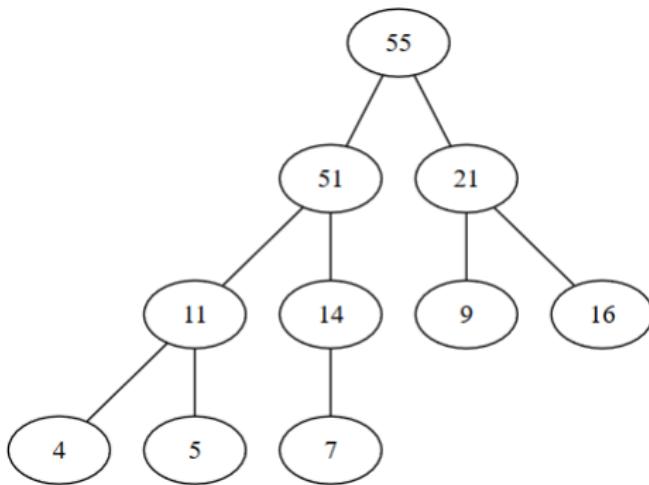


## Binary Heap - Remove - example

- *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.

# 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...

# 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:

## 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)?

- 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.)

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 9

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

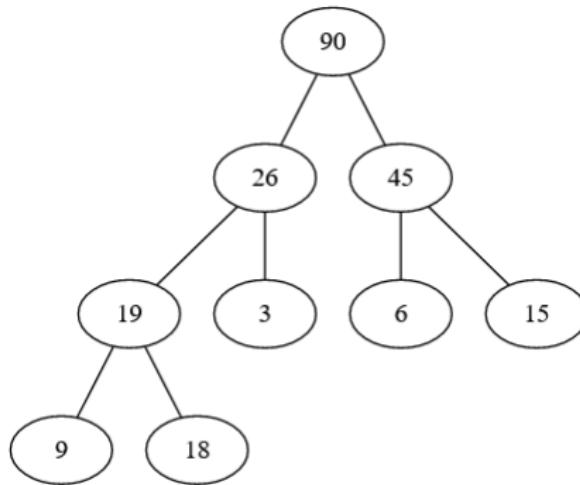
# In Lecture 8...

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

- Binary Heap
- Binomial Heap
- Hash Tables

# Binary heap - recap

- It is a data structure that can be used to represent Priority Queues (and no other containers)
- It is memorized as an array, which is interpreted/visualized in the form of a binary tree.
- In order to have a valid binary heap we need to have a *heap-structure* and a *heap property*.
- If the array is: [90, 26, 45, 19, 3, 6, 15, 9, 18]

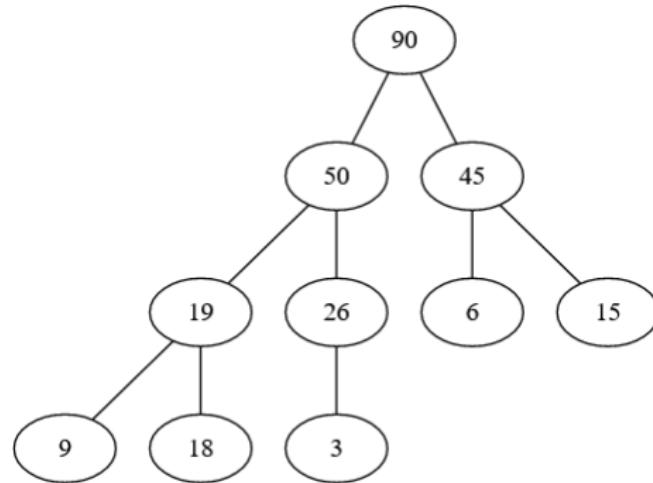


# Binary Heap - recap

- Add 50 to the previous heap

# Binary Heap - recap

- Add 50 to the previous heap

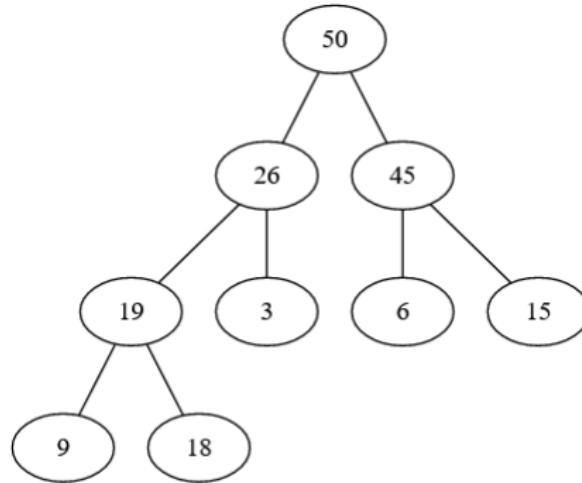


# Binary Heap - recap

- Remove any element from the previous heap

# Binary Heap - recap

- Remove any element from the previous heap



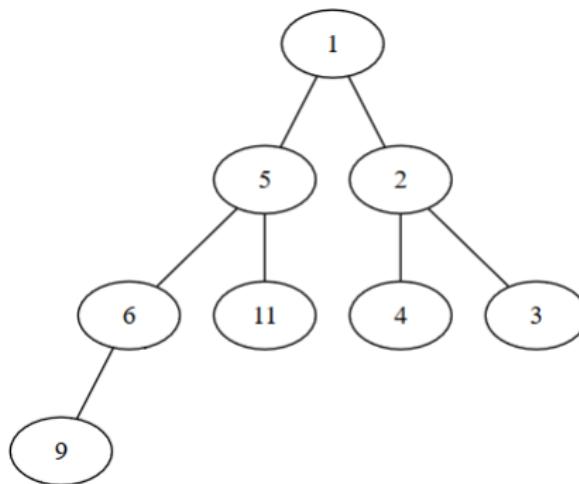
- There is a sorting algorithm, called *Heap-sort*, that is based on the use of a heap.
- In the following we are going to assume that we want to sort a sequence in ascending order.
- Let's sort the following sequence: [6, 1, 3, 9, 11, 4, 2, 5]

# 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?

## Heap-sort - Naive approach

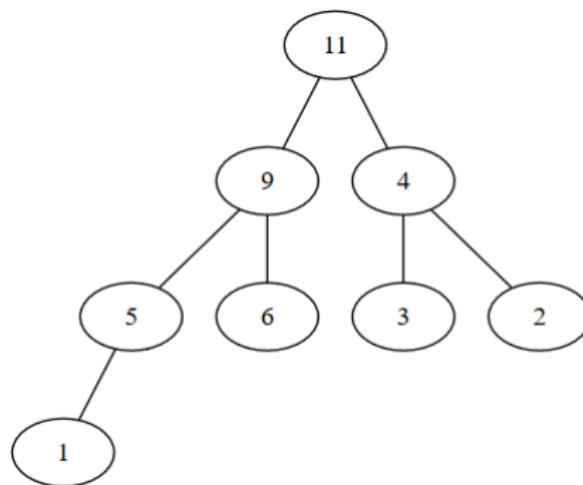
- What is the time complexity of the heap-sort algorithm described above?
- The time complexity of the algorithm is  $O(n \log_2 n)$
- What is the extra space complexity of the heap-sort algorithm described above (do we need an extra array)?

## 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  $O(n \log_2 n)$ )

# 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.

# 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).

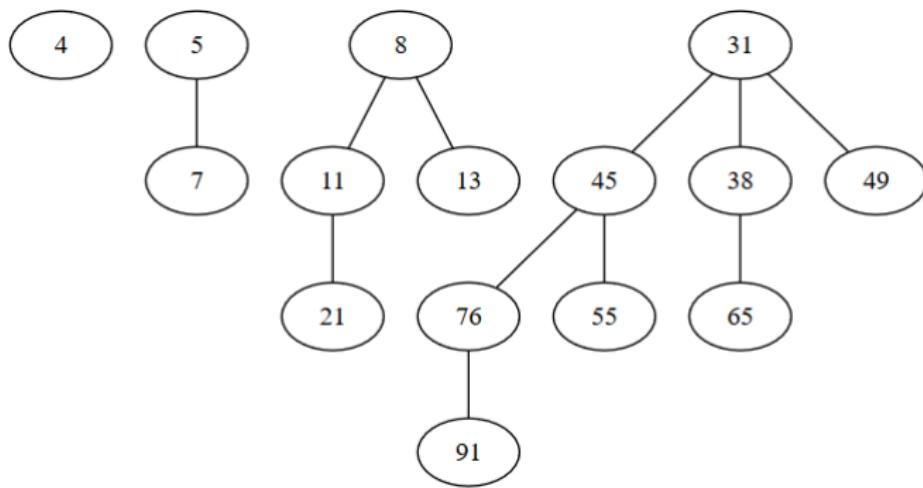
# Priority Queue - Other representations

- 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).

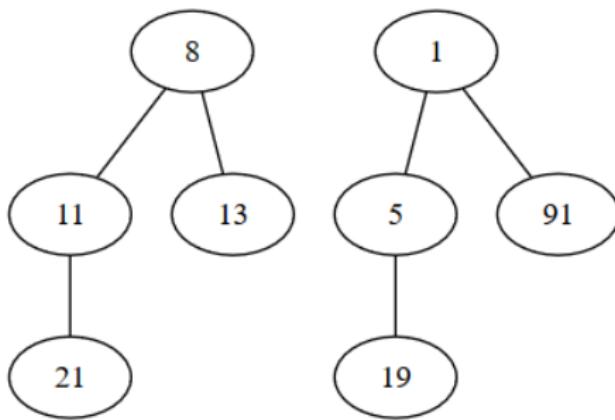
# Binomial tree - Example



Binomial trees of order 0, 1, 2 and 3

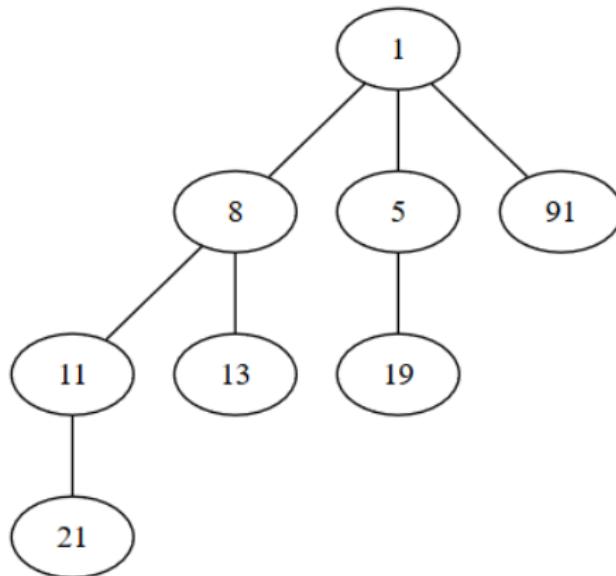
- A binomial tree of order  $k$  has exactly  $2^k$  nodes.
- The height of a binomial tree of order  $k$  is  $k$ .
- If we delete the root of a binomial tree of order  $k$ , we will get  $k$  binomial trees, of orders  $k - 1, k - 2, \dots, 2, 1, 0$ .
- Two binomial trees of the same order  $k$  can 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



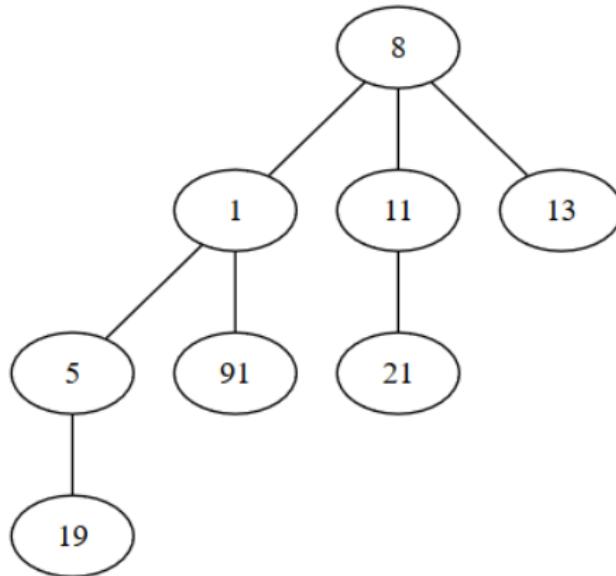
Before merge we have two binomial trees of order 2

# Binomial tree - Merge II



One way of merging the two binomial trees into one of order 3

# Binomial tree - Merge III



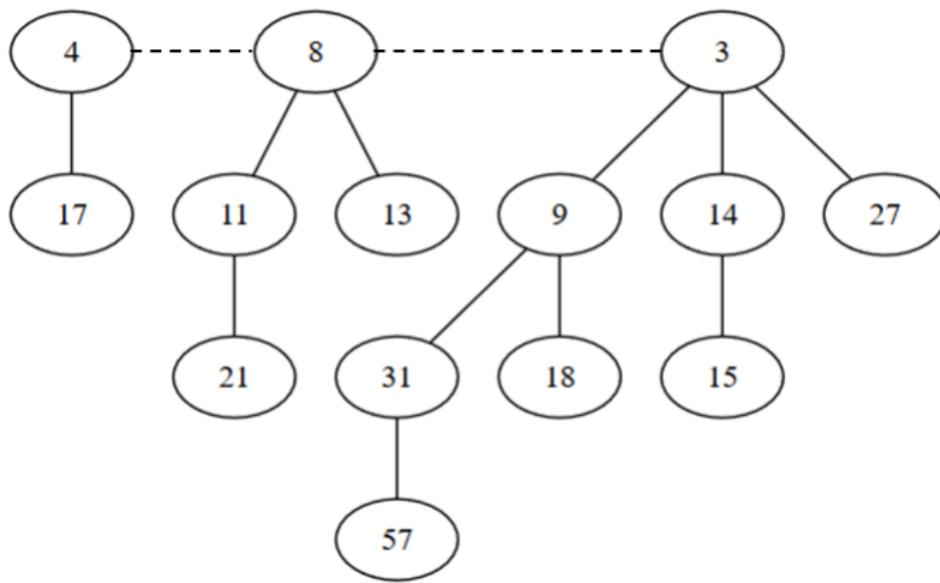
Another way of merging the two binomial trees into one of order 3

- 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

- 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.

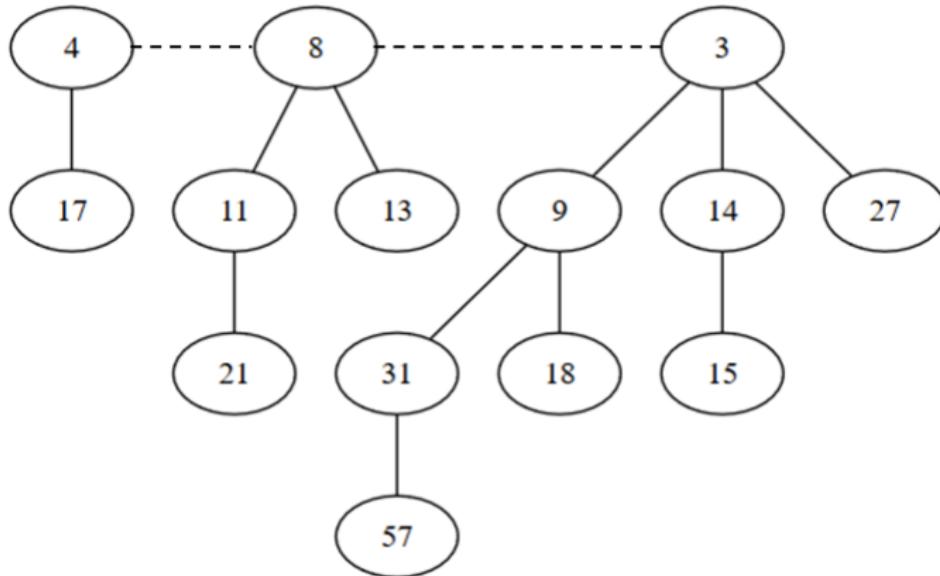
- A binomial heap with  $n$  elements contains at most  $\log_2 n$  binomial trees.
- The height of the binomial heap is at most  $\log_2 n$ .

## Binomial heap - merge

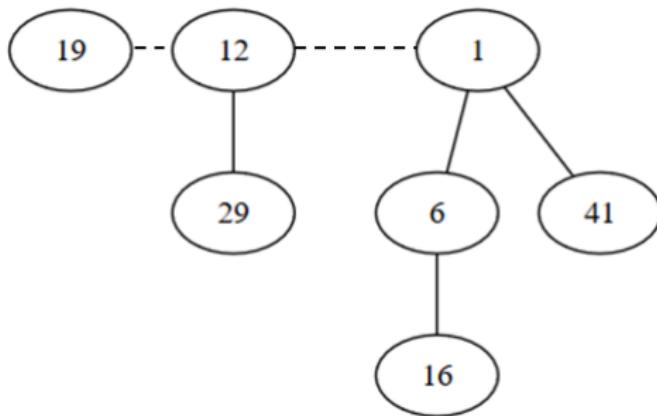
- The most interesting operation for two binomial heaps is the merge operation, which is used by other operations as well. After the merge operation the two previous binomial heaps will no longer exist, we will only have the result.
- Since both binomial heaps are sorted linked lists, the first step 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.

# Binomial heap - merge - example I

- Let's merge the following two binomial heaps:

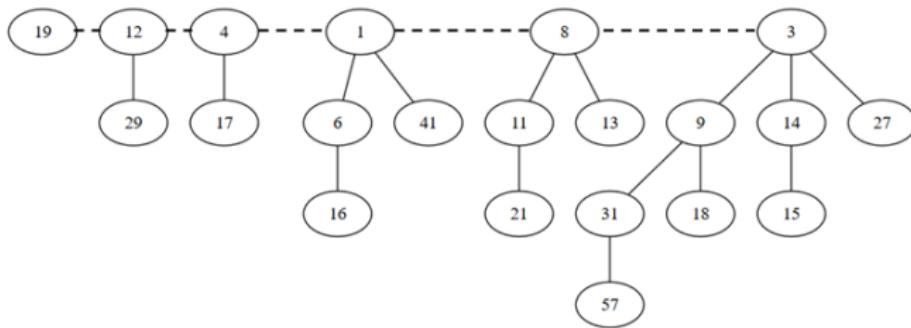


## Binomial heap - merge - example II



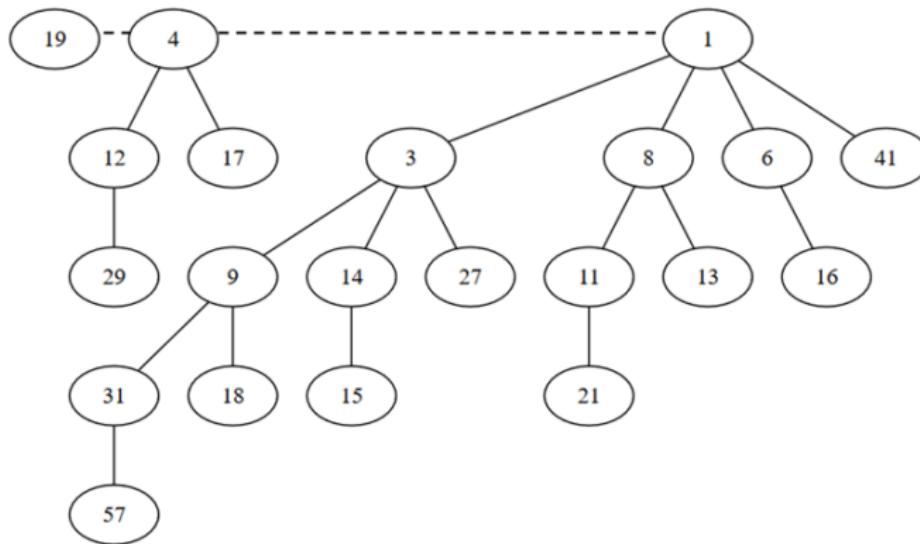
# Binomial heap - merge - example III

- 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  $\log_2 n$ ).

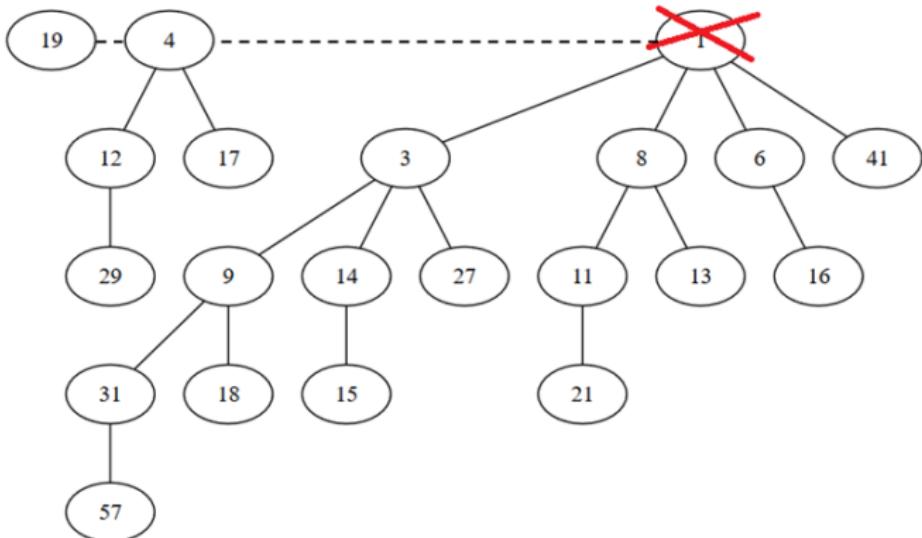
## Binomial heap - other operations I

- Most of the other operations that we have for the binomial heap (because we need them for the priority queue) will use the merge operation presented above.
- Push operation:* Inserting a new element means creating a binomial heap with just that element and merging it with the existing one. Complexity of insert is  $O(\log_2 n)$  in worst case ( $\Theta(1)$  amortized).
- Top operation:* The minimum element of a binomial heap (the element with the highest priority) is the root of one of the binomial trees. Returning the minimum means checking every root, so it has complexity  $O(\log_2 n)$ .

- *Pop operation:* Removing the minimum element means removing the root of one of the binomial trees. If we delete the root of a binomial tree, we will get a sequence of binomial trees. These trees are transformed into a binomial heap (just reverse their order), and a merge is performed between this new binomial heap and the one formed by the remaining elements of the original binomial heap.

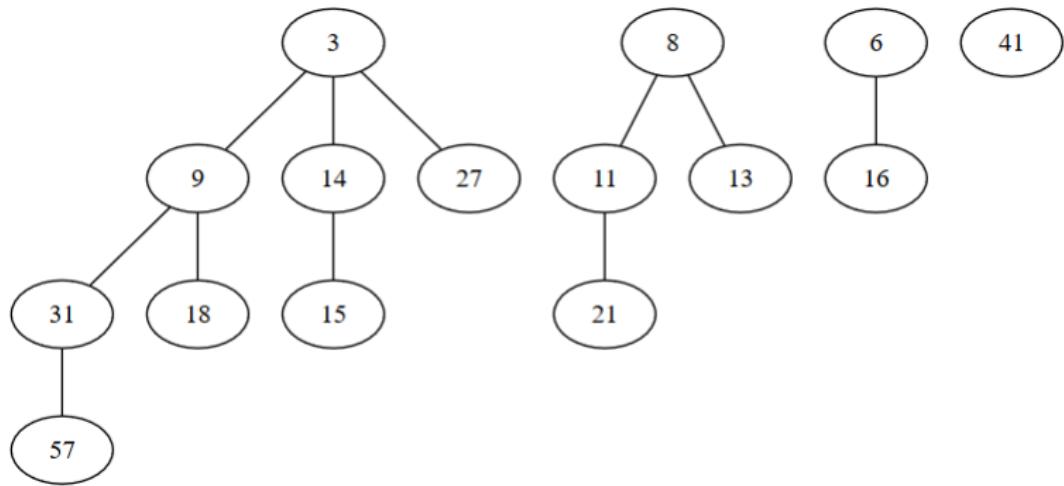
# Binomial heap - other operations III

- The minimum is one of the roots.



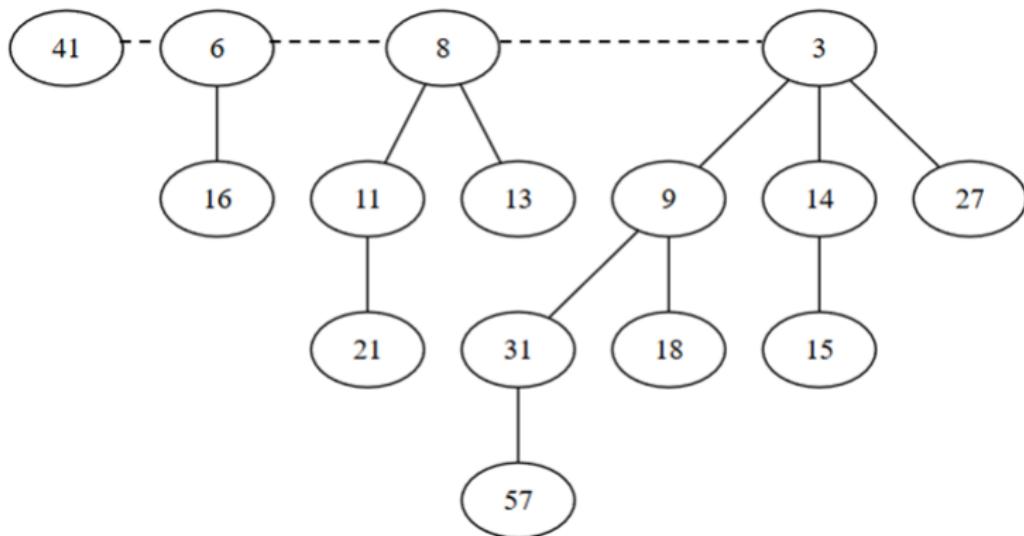
# Binomial heap - other operations IV

- Break the corresponding tree into  $k$  binomial trees



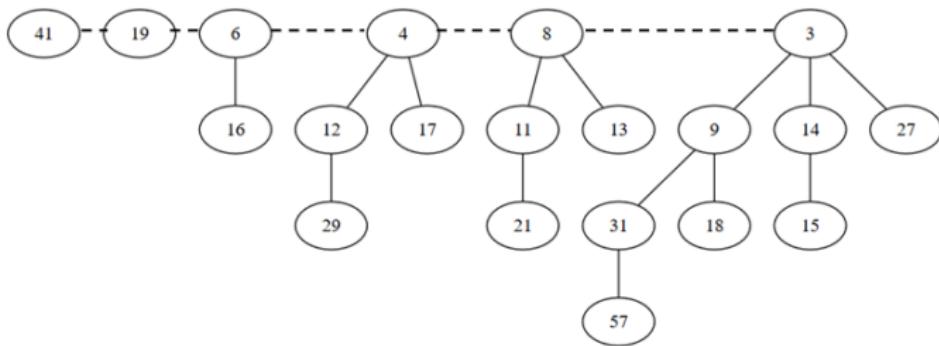
## Binomial heap - other operations V

- Create a binomial heap of these trees



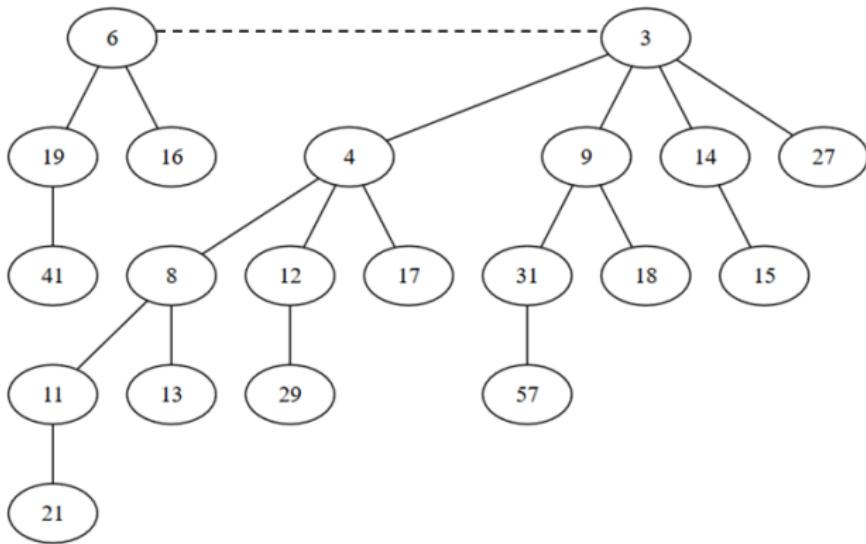
# Binomial heap - other operations VI

- Merge it with the existing one (after the merge algorithm)



# Binomial heap - other operations VII

- After the transformation



- The complexity of the remove-minimum operation is  $O(\log_2 n)$

- Assuming that we have a pointer to the element whose priority has to be increased (in our figures lower number means higher priority), we can just change the priority and bubble-up the node if its priority is greater than the priority of its parent. Complexity of the operation is:  $O(\log_2 n)$
- Assuming that we have a pointer to the element that we want to delete, we can first decrease its priority to  $-\infty$  (this will move it to the root of the corresponding binomial tree) and remove it. Complexity of the operation is:  $O(\log_2 n)$

- 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)

# Problems with stacks, queues and priority queues II

- 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 ← {initial position}
```

```
S ← {initial position}
```

```
while S ≠ ∅ execute
```

```
    Let p be one element of S
```

```
    S ← S \{p}
```

```
    for each valid position q where we can get from p and which is not in T do
```

```
        T ← T ∪ {q}
```

```
        S ← S ∪ {q}
```

```
    end-for
```

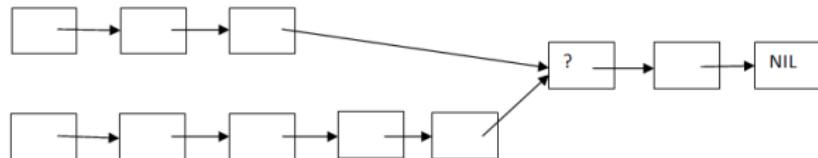
```
end-while
```

# 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?

# Direct-address tables I

- If we want to formalize the problem:
  - We have data where every element has a key (a natural number).
  - The universe of keys (the possible values for the keys) is relatively small,  $U = \{0, 1, 2, \dots, m - 1\}$
  - No two elements have the same key
  - We have to support the basic dictionary operations: INSERT, DELETE and SEARCH

- Solution:
  - Use an array  $T$  with  $m$  positions (remember, the keys belong to the  $[0, m - 1]$  interval)
  - Data about element with key  $k$ , will be stored in the  $T[k]$  slot
  - Slots not corresponding to existing elements will contain the value NIL (or some other special value to show that they are empty)

# Operations for a direct-address table

**function** search( $T, k$ ) **is:**

//pre:  $T$  is an array (the direct-address table),  $k$  is a key

    search  $\leftarrow T[k]$

**end-function**

# Operations for a direct-address table

**function** search( $T, k$ ) **is:**

//pre:  $T$  is an array (the direct-address table),  $k$  is a key

    search  $\leftarrow T[k]$

**end-function**

**subalgorithm** insert( $T, x$ ) **is:**

//pre:  $T$  is an array (the direct-address table),  $x$  is an element

$T[\text{key}(x)] \leftarrow x$  //key( $x$ ) returns the key of an element

**end-subalgorithm**

# Operations for a direct-address table

**function** search( $T, k$ ) **is:**

//pre:  $T$  is an array (the direct-address table),  $k$  is a key

    search  $\leftarrow T[k]$

**end-function**

**subalgorithm** insert( $T, x$ ) **is:**

//pre:  $T$  is an array (the direct-address table),  $x$  is an element

$T[\text{key}(x)] \leftarrow x$  //key( $x$ ) returns the key of an element

**end-subalgorithm**

**subalgorithm** delete( $T, x$ ) **is:**

//pre:  $T$  is an array (the direct-address table),  $x$  is an element

$T[\text{key}(x)] \leftarrow \text{NIL}$

**end-subalgorithm**

# Direct-address table - Advantages and disadvantages

- Advantages of direct address-tables:
  - They are simple
  - They are efficient - all operations run in  $\Theta(1)$  time.
- Disadvantages of direct address-tables - restrictions:
  - The keys have to be natural numbers
  - The keys have to come from a small universe (interval)
  - The number of actual keys can be a lot less than the cardinal of the universe (storage space is wasted)

- Assume that we have a direct address  $T$  of length  $m$ . How can we find the maximum element of the direct-address table? What is the complexity of the operation?
- How does the operation for finding the maximum change if we have a hash table, instead of a direct-address table (consider collision resolution by separate chaining, coalesced chaining and open addressing)?

- Hash tables are generalizations of direct-address tables and they represent a *time-space trade-off*.
- Searching for an element still takes  $\Theta(1)$  time, but as *average case complexity* (worst case complexity is higher)

# Hash tables - main idea I

- We will still have a table  $T$  of size  $m$  (but now  $m$  is not the number of possible keys,  $|U|$ ) - *hash table*
- Use a function  $h$  that will map a key  $k$  to a slot in the table  $T$  - *hash function*

$$h : U \rightarrow \{0, 1, \dots, m - 1\}$$

- Remarks:
  - In case of direct-address tables, an element with key  $k$  is stored in  $T[k]$ .
  - In case of hash tables, an element with key  $k$  is stored in  $T[h(k)]$ .

## Hash tables - main idea II

- The point of the hash function is to reduce the range of array indexes that need to be handled => instead of  $|U|$  values, we only need to handle  $m$  values.
- Consequence:
  - two keys may hash to the same slot => **a collision**
  - we need techniques for resolving the conflict created by collisions
- The two main points of discussion for hash tables are:
  - How to define the hash function
  - How to resolve collisions

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 10

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

Binary Heap

Binomial Heap

Hash Tables

- Hash Tables

# Hash tables - recap I

- We have a table  $T$  of size  $m$  - *hash table*
- Use a function  $h$  that will map a key  $k$  to a slot in the table  $T$  - *hash function*

$$h : U \rightarrow \{0, 1, \dots, m - 1\}$$

- Since  $m$  is less than the total number of possible keys:
  - two keys may hash to the same slot => **a collision**
  - we need techniques for resolving the conflict created by collisions
- The two main points of discussion for hash tables are:
  - How to define the hash function
  - How to resolve collisions

# A good hash function I

- A good hash function:

- can minimize the number of collisions (but cannot eliminate all collisions)
- is deterministic
- can be computed in  $\Theta(1)$  time

# A good hash function II

- satisfies (approximately) the assumption of simple uniform hashing: **each key is equally likely to hash to any of the  $m$  slots, independently of where any other key has hashed to**

$$P(h(k) = j) = \frac{1}{m} \quad \forall j = 0, \dots, m-1 \quad \forall k \in U$$

# Examples of bad hash functions

- $h(k) = \text{constant number}$

# Examples of bad hash functions

- $h(k) = \text{constant number}$
- $h(k) = \text{random number}$

# Examples of bad hash functions

- $h(k) = \text{constant number}$
- $h(k) = \text{random number}$
- assuming that the keys are CNP numbers:
  - a hash function considering just parts of it (first digit, birth year/date, county code, etc.)
  - assume  $m = 100$  and you use the birth day from the CNP (as a number):  $h(\text{CNP}) = \text{birthday \% } 100$

## Examples of bad hash functions

- $h(k) = \text{constant number}$
- $h(k) = \text{random number}$
- assuming that the keys are CNP numbers:
  - a hash function considering just parts of it (first digit, birth year/date, county code, etc.)
  - assume  $m = 100$  and you use the birth day from the CNP (as a number):  $h(\text{CNP}) = \text{birthday \% } 100$
- $m = 16$  and  $h(k) \% m$  can also be problematic

# Examples of bad hash functions

- $h(k) = \text{constant number}$
- $h(k) = \text{random number}$
- assuming that the keys are CNP numbers:
  - a hash function considering just parts of it (first digit, birth year/date, county code, etc.)
  - assume  $m = 100$  and you use the birth day from the CNP (as a number):  $h(\text{CNP}) = \text{birthday \% } 100$
- $m = 16$  and  $h(k) \% m$  can also be problematic
- etc.

- The simple uniform hashing theorem is hard to satisfy, especially when we do not know the distribution of data. Data does not always have a uniform distribution
  - dates
  - group numbers at our faculty
  - postal codes
  - first letter of an English word
- In practice we use heuristic techniques to create hash functions that perform well.
- Most hash functions assume that the keys are natural numbers. If this is not true, they have to be interpreted as natural number. In what follows, we assume that the keys are natural numbers.

# The division method

## The division method

$$h(k) = k \bmod m$$

For example:

$$m = 13$$

$$k = 63 \Rightarrow h(k) = 11$$

$$k = 52 \Rightarrow h(k) = 0$$

$$k = 131 \Rightarrow h(k) = 1$$

- Requires only a division so it is quite fast
- Experiments show that good values for  $m$  are primes not too close to exact powers of 2

# The division method

- Interestingly, Java uses the division method with a table size which is power of 2 (initially 16).
- They avoid a problem by using a second function for hashing, before applying the mod:

```
/*
 * Applies a supplemental hash function to a given hashCode, which
 * defends against poor quality hash functions. This is critical
 * because HashMap uses power-of-two length hash tables, that
 * otherwise encounter collisions for hashCodes that do not differ
 * in lower bits. Note: Null keys always map to hash 0, thus index 0.
 */
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

## Mid-square method

- Assume that the table size is  $10^r$ , for example  $m = 100$  ( $r = 2$ )
- For getting the hash of a number, multiply it by itself and take the middle  $r$  digits.
- For example,  $h(4567) = \text{middle 2 digits of } 4567 * 4567 = \text{middle 2 digits of } 20857489 = 57$
- Same thing works for  $m = 2^r$  and the binary representation of the numbers
- $m = 2^4$ ,  $h(1011) = \text{middle 4 digits of } 01111001 = 1110$

# The multiplication method I

## The multiplication method

$h(k) = \text{floor}(m * \text{frac}(k * A))$  where

$m$  - the hash table size

$A$  - constant in the range  $0 < A < 1$

$\text{frac}(k * A)$  - fractional part of  $k * A$

## For example

$m = 13$   $A = 0.6180339887$

$k=63 \Rightarrow h(k) = \text{floor}(13 * \text{frac}(63 * A)) = \text{floor}(12.16984) = 12$

$k=52 \Rightarrow h(k) = \text{floor}(13 * \text{frac}(52 * A)) = \text{floor}(1.790976) = 1$

$k=129 \Rightarrow h(k) = \text{floor}(13 * \text{frac}(129 * A)) = \text{floor}(9.442999) = 9$

## The multiplication method II

- Advantage: the value of  $m$  is not critical, typically  $m = 2^p$  for some integer  $p$
- Some values for  $A$  work better than others. Knuth suggests
$$\frac{\sqrt{5}-1}{2} = 0.6180339887$$

# Universal hashing I

- If we know the exact hash function used by a hash table, we can always generate a set of keys that will hash to the same position (collision). This reduces the performance of the table.
- For example:

$$m = 13$$

$$h(k) = k \bmod m$$

$k = 11, 24, 37, 50, 63, 76, \text{etc.}$

# Universal hashing II

- Instead of having one hash function, we have a collection  $\mathcal{H}$  of hash functions that map a given universe  $U$  of keys into the range  $\{0, 1, \dots, m - 1\}$
- Such a collection is said to be **universal** if for each pair of distinct keys  $x, y \in U$  the number of hash functions from  $\mathcal{H}$  for which  $h(x) = h(y)$  is precisely  $\frac{|\mathcal{H}|}{m}$
- In other words, with a hash function randomly chosen from  $\mathcal{H}$  the chance of collision between  $x$  and  $y$ , where  $x \neq y$ , is exactly  $\frac{1}{m}$

## Example 1

Fix a prime number  $p >$  the maximum possible value for a key from  $U$ .

For every  $a \in \{1, \dots, p - 1\}$  and  $b \in \{0, \dots, p - 1\}$  we can define a hash function  $h_{a,b}(k) = ((a * k + b) \bmod p) \bmod m$ .

- For example:
  - $h_{3,7}(k) = ((3 * k + 7) \bmod p) \bmod m$
  - $h_{4,1}(k) = ((4 * k + 1) \bmod p) \bmod m$
  - $h_{8,0}(k) = ((8 * k) \bmod p) \bmod m$
- There are  $p * (p - 1)$  possible hash functions that can be chosen.

## Example 2

If the key  $k$  is an array  $\langle k_1, k_2, \dots, k_r \rangle$  such that  $k_i < m$  (or it can be transformed into such an array, by writing the  $k$  as a number in base  $m$ ).

Let  $\langle x_1, x_2, \dots, x_r \rangle$  be a fixed sequence of random numbers, such that  $x_i \in \{0, \dots, m - 1\}$  (another number in base  $m$  with the same length).

$$h(k) = \sum_{i=1}^r k_i * x_i \bmod m$$

## Example 3

Suppose the keys are  $u - bits$  long and  $m = 2^b$ .

Pick a random  $b - by - u$  matrix (called  $h$ ) with 0 and 1 values only.

Pick  $h(k) = h * k$  where in the multiplication we do addition mod 2.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

# Using keys that are not natural numbers I

- The previously presented hash functions assume that keys are natural numbers.
- If this is not true there are two options:
  - Define special hash functions that work with your keys (for example, for real number from the  $[0,1)$  interval  $h(k) = [k * m]$  can be used)
  - Use a function that transforms the key to a natural number (and use any of the above-mentioned hash functions) - `hashCode` in Java, `hash` in Python

# Using keys that are not natural numbers II

- If the key is a string  $s$ :
  - we can consider the ASCII codes for every letter
  - we can use 1 for  $a$ , 2 for  $b$ , etc.
- Possible implementations for  $hashCode$ 
  - $s[0] + s[1] + \dots + s[n - 1]$ 
    - Anagrams have the same sum *SAUCE* and *CAUSE*
    - *DATES* has the same sum ( $D = C + 1$ ,  $T = U - 1$ )
    - Assuming maximum length of 10 for a word (and the second letter representation),  $hashCode$  values range from 1 (the word *a*) to 260 (*zzzzzzzzzz*). Considering a dictionary of about 50,000 words, we would have on average 192 word for a  $hashCode$  value.

# Using keys that are not natural numbers III

- $s[0] * 26^{n-1} + s[1] * 26^{n-2} + \dots + s[n - 1]$  where
  - n - the length of the string
  - Generates a much larger interval of *hashCode* values.
  - Instead of 26 (which was chosen since we have 26 letters) we can use a prime number as well (Java uses 31, for example).

# Cryptographic hashing

# Cryptographic hashing

- Another use of hash functions besides as part of a hash table
- It is a hash function, which can be used to generate a code (the hash value) for any variable size data
- Used for checksums, storing passwords, etc.

- When two keys,  $x$  and  $y$ , have the same value for the hash function  $h(x) = h(y)$  we have a *collision*.
- A good hash function can reduce the number of collisions, but it cannot eliminate them at all:
  - Try fitting  $m + 1$  keys into a table of size  $m$
- There are different collision resolution methods:
  - Separate chaining
  - Coalesced chaining
  - Open addressing

# The birthday paradox

- *How many randomly chosen people are needed in a room, to have a good probability - about 50% - of having two people with the same birthday?*
- It is obvious that if we have 367 people, there will be at least two with the same birthday (there are only 366 possibilities).

# The birthday paradox

- *How many randomly chosen people are needed in a room, to have a good probability - about 50% - of having two people with the same birthday?*
- It is obvious that if we have 367 people, there will be at least two with the same birthday (there are only 366 possibilities).
- What might not be obvious, is that approximately 70 people are needed for a 99.9% probability
- 23 people are enough for a 50% probability

# Separate chaining

- Collision resolution by separate chaining: each slot from the hash table  $T$  contains a linked list, with the elements that hash to that slot
- Dictionary operations become operations on the corresponding linked list:
  - $\text{insert}(T, x)$  - insert a new node to the beginning of the list  $T[h(\text{key}[x])]$
  - $\text{search}(T, k)$  - search for an element with key  $k$  in the list  $T[h(k)]$
  - $\text{delete}(T, x)$  - delete  $x$  from the list  $T[h(\text{key}[x])]$

# Hash table with separate chaining - representation

- A hash table with separate chaining would be represented in the following way (for simplicity, we will keep only the keys in the nodes).

## Node:

key: TKey

next:  $\uparrow$  Node

## HashTable:

T:  $\uparrow$ Node[] *//an array of pointers to nodes*

m: Integer

h: TFunction *//the hash function*

# Hash table with separate chaining - search

```
function search(ht, k) is:
    //pre: ht is a HashTable, k is a TKey
    //post: function returns True if k is in ht, False otherwise
    position ← ht.h(k)
    currentNode ← ht.T[position]
    while currentNode ≠ NIL and [currentNode].key ≠ k execute
        currentNode ← [currentNode].next
    end-while
    if currentNode ≠ NIL then
        search ← True
    else
        search ← False
    end-if
end-function
```

- Usually search returns the info associated with the key  $k$

# Analysis of hashing with chaining

- The average performance depends on how well the hash function  $h$  can distribute the keys to be stored among the  $m$  slots.
- **Simple Uniform Hashing** assumption: each element is equally likely to hash into any of the  $m$  slots, independently of where any other elements have hashed to.
- **load factor**  $\alpha$  of the table  $T$  with  $m$  slots containing  $n$  elements
  - is  $n/m$
  - represents the average number of elements stored in a chain
  - in case of separate chaining can be less than, equal to, or greater than 1.

# Analysis of hashing with chaining - Insert

- The slot where the element is to be added can be:
  - empty - create a new node and add it to the slot
  - occupied - create a new node and add it to the beginning of the list
- In either case worst-case time complexity is:  $\Theta(1)$
- If we have to check whether the element already exists in the table, the complexity of searching is added as well.

# Analysis of hashing with chaining - Search I

- There are two cases
  - unsuccessful search
  - successful search
- We assume that
  - the hash value can be computed in constant time ( $\Theta(1)$ )
  - the time required to search an element with key  $k$  depends linearly on the length of the list  $T[h(k)]$

- **Theorem:** In a hash table in which collisions are resolved by separate chaining, an unsuccessful search takes time  $\Theta(1 + \alpha)$ , on the average, under the assumption of simple uniform hashing.
- **Theorem:** In a hash table in which collisions are resolved by chaining, a successful search takes time  $\Theta(1 + \alpha)$ , on the average, under the assumption of simple uniform hashing.
- Proof idea:  $\Theta(1)$  is needed to compute the value of the hash function and  $\alpha$  is the average time needed to search one of the  $m$  lists

# Analysis of hashing with chaining - Search III

- If  $n = O(m)$  (the number of hash table slots is proportional to the number of elements in the table, if the number of elements grows, the size of the table will grow as well)
  - $\alpha = n/m = O(m)/m = \Theta(1)$
  - searching takes constant time on average
- Worst-case time complexity is  $\Theta(n)$ 
  - When all the nodes are in a single linked-list and we are searching this list
  - In practice hash tables are pretty fast

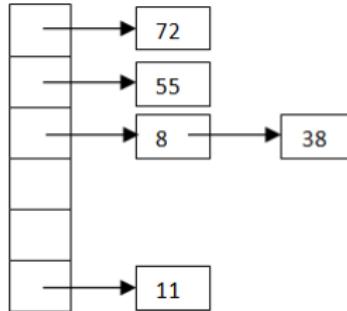
- If the lists are doubly-linked and we know the address of the node:  $\Theta(1)$
- If the lists are singly-linked: proportional to the length of the list
- **All dictionary operations can be supported in  $\Theta(1)$  time on average.**
- In theory we can keep any number of elements in a hash table with separate chaining, but the complexity is proportional to  $\alpha$ . If  $\alpha$  is too large  $\Rightarrow$  resize and rehash.

## Example

- Assume we have a hash table with  $m = 6$  that uses separate chaining for collision resolution, with the following policy: if the load factor of the table after an insertion is greater than or equal to 0.7, we double the size of the table
- Using the division method, insert the following elements, in the given order, in the hash table: 38, 11, 8, 72, 57, 29, 2.

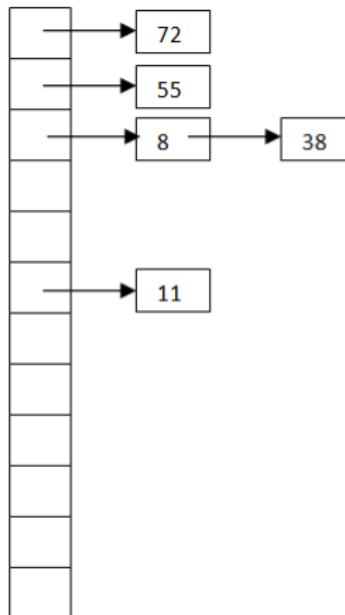
# Example

- $h(38) = 2$  (load factor will be  $1/6$ )
- $h(11) = 5$  (load factor will be  $2/6$ )
- $h(8) = 2$  (load factor will be  $3/6$ )
- $h(72) = 0$  (load factor will be  $4/6$ )
- $h(55) = 1$  (load factor will be  $5/6$  - greater than  $0.7$ )
- The table after the first five elements were added:



# Example

- Is it OK if after the resize this is our hash table?

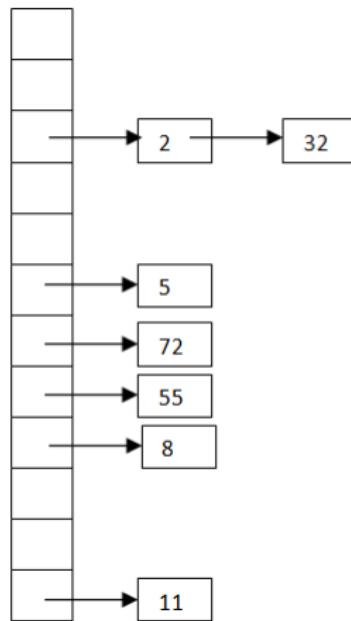


## Example

- The result of the hash function (i.e. the position where an element is added) depends on the size of the hash table. If the size of the hash table changes, the value of the hash function changes as well, which means that search and remove operations might not find the element.
- After a resize operation, we have to add all elements again in the hash table, to make sure that they are at the correct position → rehash

# Example

- After rehash and adding the other two elements:



- What do you think, which containers cannot be represented on a hash table?

- What do you think, which containers cannot be represented on a hash table?
- How can we define an iterator for a hash table with separate chaining?

- What do you think, which containers cannot be represented on a hash table?
- How can we define an iterator for a hash table with separate chaining?
- Since hash tables are used to implement containers where the order of the elements is not important, our iterator can iterate through them in any order.
- For the hash table from the previous example, the easiest order in which the elements can be iterated is: 2, 32, 5, 72, 55, 8, 11

- Iterator for a hash table with separate chaining is a combination of an iterator on an array (table) and on a linked list.
- We need a current position to know the position from the table that we are at, but we also need a current node to know the exact node from the linked list from that position.

## IteratorHT:

ht: HashTable

currentPos: Integer

currentNode:  $\uparrow$  Node

- How can we implement the *init* operation?

# Iterator - init

- How can we implement the *init* operation?

```
subalgorithm init(ith, ht) is:
//pre: ith is an IteratorHT, ht is a HashTable
    ith.ht ← ht
    ith.currentPos ← 0
    while ith.currentPos < ht.m and ht.T[ith.currentPos] = NIL execute
        ith.currentPos ← ith.currentPos + 1
    end-while
    if ith.currentPos < ht.m then
        ith.currentNode ← ht.T[ith.currentPos]
    else
        ith.currentNode ← NIL
    end-if
end-subalgorithm
```

- Complexity of the algorithm:

# Iterator - init

- How can we implement the *init* operation?

```
subalgorithm init(ith, ht) is:  
    //pre: ith is an IteratorHT, ht is a HashTable  
    ith.ht ← ht  
    ith.currentPos ← 0  
    while ith.currentPos < ht.m and ht.T[ith.currentPos] = NIL execute  
        ith.currentPos ← ith.currentPos + 1  
    end-while  
    if ith.currentPos < ht.m then  
        ith.currentNode ← ht.T[ith.currentPos]  
    else  
        ith.currentNode ← NIL  
    end-if  
end-subalgorithm
```

- Complexity of the algorithm:  $O(m)$

- How can we implement the *getCurrent* operation?

# Iterator - other operations

- How can we implement the *getCurrent* operation?
- How can we implement the *next* operation?

- How can we implement the *getCurrent* operation?
- How can we implement the *next* operation?
- How can we implement the *valid* operation?

# Sorted containers

- How can we define a sorted container on a hash table with separate chaining?

# Sorted containers

- How can we define a sorted container on a hash table with separate chaining?
  - Hash tables are in general not very suitable for sorted containers.
  - However, if we have to implement a sorted container on a hash table with separate chaining, we can store the individual lists in a sorted order and for the iterator we can return them in a sorted order.

# Coalesced chaining

- Collision resolution by coalesced chaining: each element from the hash table is stored inside the table (no linked lists), but each element has a *next* field, similar to a linked list on array.
- When a new element has to be inserted and the position where it should be placed is occupied, we will put it to any empty position, and set the *next* link, so that the element can be found in a search.
- Since elements are in the table,  $\alpha$  can be at most 1.

# Coalesced chaining - example

- Consider a hash table of size  $m = 16$  that uses coalesced chaining for collision resolution and a hash function with the division method
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.
- Let's compute the value of the hash function for every key:

Key	76	12	109	43	22	18	55	81	91	27	13	16	39
Hash	12	12	13	11	6	2	7	1	11	11	13	0	7

## Example

- Initially the hash table is empty. All next values are -1 and the first empty position is position 0.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

firstEmpty = 0

- 76 will be added to position 12. But 12 should also be added there. Since that position is already occupied, we add 12 to position firstEmpty and set the next of 76 to point to position 0. Then we reset firstEmpty to the next empty position

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12												76			

firstEmpty = 1

## Example

- And we continue in the same manner. We have no collisions up to 81, but we need to reset firstEmpty when we *accidentally* occupy it.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	81	18				22	55				43	76	109		
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1

firstEmpty = 3

- When adding 91, we put it to position firstEmpty and set the next link of position 11 to position 3.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	81	18	91			22	55				43	76	109		
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	3	0	-1	-1	-1

firstEmpty = 4

# Example

- The final table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	81	18	91	27	13	22	55	16	39		43	76	109		
8	-1	-1	4	-1	-1	-1	9	-1	-1	-1	3	0	5	-1	-1

firstEmpty = 10

# Coalesced chaining - representation

- What fields do we need to represent a hash table where collision resolution is done with coalesced chaining?

# Coalesced chaining - representation

- What fields do we need to represent a hash table where collision resolution is done with coalesced chaining?

HashTable:

T: TKey[]

next: Integer[]

m: Integer

firstEmpty: Integer

h: TFunction

- For simplicity, in the following, we will consider only the keys.

## Coalesced chaining - insert

**subalgorithm** insert (ht, k) **is:**

//pre: ht is a HashTable, k is a TKey

//post: k was added into ht

pos  $\leftarrow$  ht.h(k)

**if** ht.T[pos] = -1 **then** // -1 means empty position

    ht.T[pos]  $\leftarrow$  k

    ht.next[pos]  $\leftarrow$  -1

**else**

**if** ht.firstEmpty = ht.m **then**

        @resize and rehash

**end-if**

    current  $\leftarrow$  pos

**while** ht.next[current]  $\neq$  -1 **execute**

        current  $\leftarrow$  ht.next[current]

**end-while**

//continued on the next slide...

# Coalesced chaining - insert

```
ht.T[ht.firstEmpty] ← k  
ht.next[ht.firstEmpty] ← - 1  
ht.next[current] ← ht.firstEmpty  
changeFirstEmpty(ht)
```

**end-if**

**end-subalgorithm**

- Complexity:  $\Theta(1)$  on average,  $\Theta(n)$  - worst case

**subalgorithm** changeFirstEmpty(ht) **is:**

//pre: ht is a HashTable

//post: the value of ht.firstEmpty is set to the next free position

ht.firstEmpty  $\leftarrow$  ht.firstEmpty + 1

**while** ht.firstEmpty < ht.m **and** ht.T[ht.firstEmpty]  $\neq$  -1

**execute**

ht.firstEmpty  $\leftarrow$  ht.firstEmpty + 1

**end-while**

**end-subalgorithm**

- Complexity:  $O(m)$
- *Think about it:* Should we keep the free spaces linked in a list as in case of a linked lists on array?

# Coalesced chaining

- Remove and search operations for coalesced chaining will be discussed in Seminar 6.
- How can we define an iterator for a hash table with coalesced chaining? What should the following operations do?
  - init
  - getCurrent
  - next
  - valid
- How can we implement a sorted container on a hash table with coalesced chaining? How can we implement its iterator?

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 11

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

# In Lecture 10...

- Hash tables
  - Hash function
  - Separate chaining
  - Coalesced chaining

- Open addressing
- Trees
- Binary Trees

# Open addressing

- In case of open addressing every element of the hash table is inside the table, we have no pointers, no next links.
- When we want to insert a new element, we will successively generate positions for the element, check (*probe*) the generated position, and place the element in the first available one.

# Open addressing

- In order to generate multiple positions, we will extend the hash function and add to it another parameter,  $i$ , which is the *probe number* and starts from 0.

$$h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$$

- For an element  $k$ , we will successively examine the positions  $\langle h(k, 0), h(k, 1), h(k, 2), \dots, h(k, m - 1) \rangle$  - called the *probe sequence*
- The *probe sequence* should be a permutation of a hash table positions  $\{0, \dots, m - 1\}$ , so that eventually every slot is considered.
- We would also like to have a hash function which can generate all the  $m!$  permutations possible (spoiler alert: we cannot)

# Open addressing - Linear probing

- One version of defining the hash function is to use linear probing:

$$h(k, i) = (h'(k) + i) \bmod m \quad \forall i = 0, \dots, m - 1$$

- where  $h'(k)$  is a *simple* hash function (for example:  
 $h'(k) = k \bmod m$ )
- the *probe sequence* for linear probing is:  
 $< h'(k), h'(k) + 1, h'(k) + 2, \dots, m - 1, 0, 1, \dots, h'(k) - 1 >$

# Open addressing - Linear probing - example

- Consider a hash table of size  $m = 16$  that uses open addressing and linear probing for collision resolution
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.
- Let's compute the value of the hash function for every key for  $i = 0$ :

Key	76	12	109	43	22	18	55	81	91	27	13	16	39
Hash	12	12	13	11	6	2	7	1	11	11	13	0	7

# Open addressing - Linear probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	81	18	13	16		22	55	39			43	76	12	109	91

# Open addressing - Linear probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	81	18	13	16		22	55	39			43	76	12	109	91

- Disadvantages of linear probing:
  - There are only  $m$  distinct probe sequences (once you have the starting position everything is fixed)
  - *Primary clustering* - long runs of occupied slots
- Advantages of linear probing:
  - Probe sequence is always a permutation
  - Can benefit from caching

# Open addressing - Linear probing - primary clustering

- Why is primary clustering a problem?
- Assume  $m$  positions,  $n$  elements and  $\alpha = 0.5$  (so  $n = m/2$ )
- Best case arrangement: every second position is empty (for example: even positions are occupied and odd ones are free)
- What is the average number probes (positions verified) that need to be checked to insert a new element?

# Open addressing - Linear probing - primary clustering

- Why is primary clustering a problem?
- Assume  $m$  positions,  $n$  elements and  $\alpha = 0.5$  (so  $n = m/2$ )
- Best case arrangement: every second position is empty (for example: even positions are occupied and odd ones are free)
- What is the average number probes (positions verified) that need to be checked to insert a new element?
  
- Worst case arrangement: all  $n$  elements are one after the other (assume in the second half of the array)
- What is the average number of probes (positions verified) that need to be checked to insert a new element?

# Open addressing - Quadratic probing

- In case of quadratic probing the hash function becomes:

$$h(k, i) = (h'(k) + c_1 * i + c_2 * i^2) \bmod m \quad \forall i = 0, \dots, m - 1$$

- where  $h'(k)$  is a *simple* hash function (for example:  $h'(k) = k \bmod m$ ) and  $c_1$  and  $c_2$  are constants initialized when the hash function is initialized.  $c_2$  should not be 0.
- Considering a simplified version of  $h(k, i)$  with  $c_1 = 0$  and  $c_2 = 1$  the probe sequence would be:  
 $< k, k + 1, k + 4, k + 9, k + 16, \dots >$

# Open addressing - Quadratic probing

- One important issue with quadratic probing is how we can choose the values of  $m$ ,  $c_1$  and  $c_2$  so that the probe sequence is a permutation.
- If  $m$  is a prime number only the first half of the probe sequence is unique, so, once the hash table is half full, there is no guarantee that an empty space will be found.
  - For example, for  $m = 17$ ,  $c_1 = 3$ ,  $c_2 = 1$  and  $k = 13$ , the probe sequence is  
 $< 13, 0, 6, 14, 7, 2, 16, 15, 16, 2, 7, 14, 6, 0, 13, 11, 11 >$
  - For example, for  $m = 11$ ,  $c_1 = 1$ ,  $c_2 = 1$  and  $k = 27$ , the probe sequence is  $< 5, 7, 0, 6, 3, 2, 3, 6, 0, 7, 5 >$

# Open addressing - Quadratic probing

- If  $m$  is a power of 2 and  $c_1 = c_2 = 0.5$ , the probe sequence will always be a permutation. For example for  $m = 8$  and  $k = 3$ :

- $h(3, 0) = (3 \% 8 + 0.5 * 0 + 0.5 * 0^2) \% 8 = 3$
- $h(3, 1) = (3 \% 8 + 0.5 * 1 + 0.5 * 1^2) \% 8 = 4$
- $h(3, 2) = (3 \% 8 + 0.5 * 2 + 0.5 * 2^2) \% 8 = 6$
- $h(3, 3) = (3 \% 8 + 0.5 * 3 + 0.5 * 3^2) \% 8 = 1$
- $h(3, 4) = (3 \% 8 + 0.5 * 4 + 0.5 * 4^2) \% 8 = 5$
- $h(3, 5) = (3 \% 8 + 0.5 * 5 + 0.5 * 5^2) \% 8 = 2$
- $h(3, 6) = (3 \% 8 + 0.5 * 6 + 0.5 * 6^2) \% 8 = 0$
- $h(3, 7) = (3 \% 8 + 0.5 * 7 + 0.5 * 7^2) \% 8 = 7$

# Open addressing - Quadratic probing

- If  $m$  is a prime number of the form  $4 * k + 3$ ,  $c_1 = 0$  and  $c_2 = (-1)^i$  (so the probe sequence is  $+0, -1, +4, -9, \text{ etc.}$ ) the probe sequence is a permutation. For example for  $m = 7$  and  $k = 3$ :
  - $h(3, 0) = (3 \% 7 + 0^2) \% 7 = 3$
  - $h(3, 1) = (3 \% 7 - 1^2) \% 7 = 2$
  - $h(3, 2) = (3 \% 7 + 2^2) \% 7 = 0$
  - $h(3, 3) = (3 \% 7 - 3^2) \% 7 = 1$
  - $h(3, 4) = (3 \% 7 + 4^2) \% 7 = 5$
  - $h(3, 5) = (3 \% 7 - 5^2) \% 7 = 6$
  - $h(3, 6) = (3 \% 7 + 6^2) \% 7 = 4$

# Open addressing - Quadratic probing - example

- Consider a hash table of size  $m = 16$  that uses open addressing with quadratic probing for collision resolution ( $h'(k)$  is a hash function defined with the division method),  $c_1 = c_2 = 0.5$ .
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

# Open addressing - Quadratic probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	81	18	16		91	22	55	39		27	43	76	12	109	

# Open addressing - Quadratic probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	81	18	16		91	22	55	39		27	43	76	12	109	

- Disadvantages of quadratic probing:
  - The performance is sensitive to the values of  $m$ ,  $c_1$  and  $c_2$ .
  - Secondary clustering* - if two elements have the same initial probe positions, their whole probe sequence will be identical:
$$h(k_1, 0) = h(k_2, 0) \Rightarrow h(k_1, i) = h(k_2, i).$$
  - There are only  $m$  distinct probe sequences (once you have the starting position the whole sequence is fixed).

# Open addressing - Double hashing

- In case of double hashing the hash function becomes:

$$h(k, i) = (h'(k) + i * h''(k)) \% m \quad \forall i = 0, \dots, m - 1$$

- where  $h'(k)$  and  $h''(k)$  are *simple* hash functions, where  $h''(k)$  should never return the value 0.
- For a key,  $k$ , the first position examined will be  $h'(k)$  and the other probed positions will be computed based on the second hash function,  $h''(k)$ .

# Open addressing - Double hashing

- Similar to quadratic probing, not every combination of  $m$  and  $h''(k)$  will return a complete permutation as a probe sequence.
- In order to produce a permutation  $m$  and all the values of  $h''(k)$  have to be relatively primes. This can be achieved in two ways:
  - Choose  $m$  as a power of 2 and design  $h''$  in such a way that it always returns an odd number.
  - Choose  $m$  as a prime number and design  $h''$  in such a way that it always returns a value from the  $\{0, m-1\}$  set (actually  $\{1, m-1\}$  set, because  $h''(k)$  should never return 0).

# Open addressing - Double hashing

- Choose  $m$  as a prime number and design  $h''$  in such a way that it always return a value from the  $\{0, m-1\}$  set.
- For example:  
$$h'(k) = k \% m$$
$$h''(k) = 1 + (k \% (m - 1)).$$
- For  $m = 11$  and  $k = 36$  we have:  
$$h'(36) = 3$$
$$h''(36) = 7$$
- The probe sequence is:  $< 3, 10, 6, 2, 9, 5, 1, 8, 4, 0, 7 >$

# Open addressing - Double hashing - example

- Consider a hash table of size  $m = 17$  that uses open addressing with double hashing for collision resolution, with  $h'(k) = k \% m$  and  $h''(k) = (1 + (k \% 16))$ .
- Insert into the table the following elements: 75, 12, 109, 43, 22, 18, 55, 81, 92, 27, 13, 16, 39.
- Values of the two hash functions for each element:

key	75	12	109	43	22	18	55	81	92	27	13	16	39
$h'(\text{key})$	7	12	7	9	5	1	4	13	7	10	13	16	5
$h''(\text{key})$	12	13	14	12	7	3	8	2	13	12	14	1	8

# Open addressing - Double hashing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	18		55	109	22		75		43	27	39	12	81		13	92

## Open addressing - Double hashing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	18		55	109	22		75		43	27	39	12	81		13	92

- Main advantage of double hashing is that even if  $h(k_1, 0) = h(k_2, 0)$  the probe sequences will be different if  $k_1 \neq k_2$ .
- For example:
  - 75:  $< 7, 2, 14, 9, 4, 16, 11, 6, 1, 13, 8, 3, 15, 10, 5, 0, 12 >$
  - 109:  $< 7, 4, 1, 15, 12, 9, 6, 3, 0, 14, 11, 8, 5, 2, 16, 13, 10 >$
- Since for every  $(h'(k), h''(k))$  pair we have a separate probe sequence, double hashing generates  $\approx m^2$  different permutations.

# Open addressing - operations

- In the following we will discuss the implementation of some of the basic dictionary operations for collision resolution with open addressing.
- We will use the notation  $h(k, i)$  for a hash function, without mentioning whether we have linear probing, quadratic probing or double hashing (code is the same for each of them, implementation of  $h$  is different only).

# Open addressing - representation

- What fields do we need to represent a hash table with collision resolution with open addressing?

# Open addressing - representation

- What fields do we need to represent a hash table with collision resolution with open addressing?

HashTable:

T: TKey[]

m: Integer

h: TFunction

- For simplicity we will consider that we only have keys.

# Open addressing - insert

- What should the *insert* operation do?

# Open addressing - insert

- What should the *insert* operation do?

```
subalgorithm insert (ht, e) is:
    //pre: ht is a HashTable, e is a TKey
    //post: e was added in ht
    i ← 0
    pos ← ht.h(e, i)
    while i < ht.m and ht.T[pos] ≠ -1 execute
        // -1 means empty space
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    if i = ht.m then
        @resize and rehash and compute the position for e again
    else
        ht.T[pos] ← e
    end-if
end-subalgorithm
```

# Open addressing - other operations

- What should the *search* operation do?

# Open addressing - other operations

- What should the *search* operation do?
- How can we *remove* an element from the hash table?

# Open addressing - other operations

- What should the *search* operation do?
- How can we *remove* an element from the hash table?
- Removing an element from a hash table with open addressing is not simple:
  - we cannot just mark the position empty - *search* might not find other elements
  - you cannot move elements - *search* might not find other elements

# Open addressing - other operations

- What should the *search* operation do?
- How can we *remove* an element from the hash table?
- Removing an element from a hash table with open addressing is not simple:
  - we cannot just mark the position empty - *search* might not find other elements
  - you cannot move elements - *search* might not find other elements
- Remove is usually implemented to mark the deleted position with a special value, *DELETED*.
- How does this special value change the implementation of the *insert* and *search* operation?

# Open addressing - Performance

- In a hash table with open addressing with load factor  $\alpha = n/m$  ( $\alpha < 1$ ), the average number of probes is at most
  - for *insert* and *unsuccessful search*

$$\frac{1}{1 - \alpha}$$

- for *successful search*

$$\frac{1}{\alpha} * \ln \frac{1}{1 - \alpha}$$

- If  $\alpha$  is constant, the complexity is  $\Theta(1)$
- Worst case complexity is  $\Theta(n)$

- Trees are one of the most commonly used data structures because they offer an efficient way of storing data and working with the data.
- In graph theory a *tree* is a connected, acyclic graph (usually undirected).
- When talking about trees as a data structure, we actually mean *rooted trees*, trees in which one node is designated to be the *root* of the tree.

# Tree - Definition

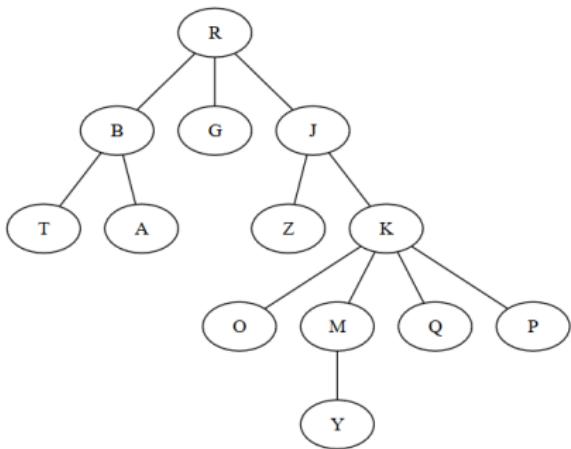
- A tree is a finite set  $\mathcal{T}$  of 0 or more elements, called *nodes*, with the following properties:
  - If  $\mathcal{T}$  is empty, then the tree is empty
  - If  $\mathcal{T}$  is not empty then:
    - There is a special node,  $R$ , called the *root* of the tree
    - The rest of the nodes are divided into  $k$  ( $k \geq 0$ ) disjunct trees,  $T_1, T_2, \dots, T_k$ , the root node  $R$  being linked by an edge to the root of each of these trees. The trees  $T_1, T_2, \dots, T_k$  are called the *subtrees (children)* of  $R$ , and  $R$  is called the *parent* of the subtrees.

# Tree - Terminology I

- An *ordered tree* is a tree in which the order of the children is well defined and relevant (instead of having a set of children, each node has a list of children).
- The *degree* of a node is defined as the number of children of the node.
- The nodes with the degree 0 (nodes without children) are called *leaf nodes*.
- The nodes that are not leaf nodes are called *internal nodes*.

- The *depth* or *level* of a node is the length of the path (measured as the number of edges traversed) from the root to the node. This path is unique. The root of the tree is at level 0 (and has depth 0).
- The *height* of a node is the length of the longest path from the node to a leaf node.
- The *height of the tree* is defined as the height of the root node, i.e., the length of the longest path from the root to a leaf.

# Tree - Terminology Example



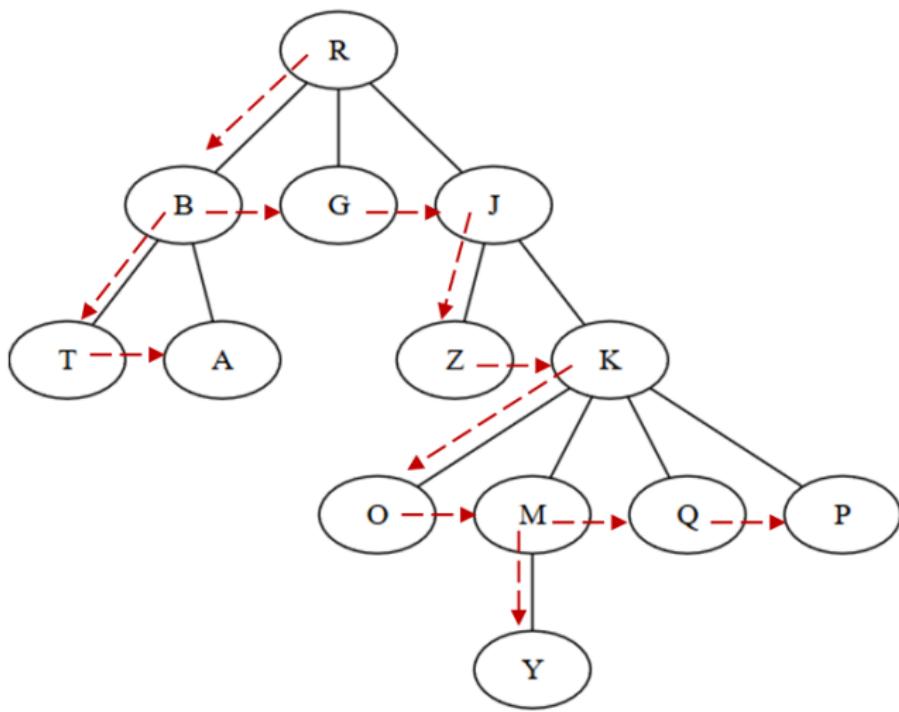
- Root of the tree:  $R$
- Children of  $R$ :  $B, G, J$
- Parent of  $M$ :  $K$
- Leaf nodes:  $T, A, G, Z, O, Y, Q, P$
- Internal nodes:  $R, B, J, K, M$
- Depth of node  $K$ : 2 (path  $R-J-K$ )
- Height of node  $K$ : 2 (path  $K-M-Y$ )
- Height of the tree (height of node  $R$ ): 4
- Nodes on level 2:  $T, A, Z, K$

- How can we represent a tree in which every node has at most  $k$  children?
- One option is to have a structure for a *node* that contains the following:
  - information from the node
  - address of the parent node (not mandatory)
  - $k$  fields, one for each child
- Obs: this is doable if  $k$  is not too large

- Another option is to have a structure for a *node* that contains the following:
  - information from the node
  - address of the parent node (not mandatory)
  - an array of dimension  $k$ , in which each element is the address of a child
  - number of children (number of occupied positions from the above array)
- Disadvantage of these approaches is that we occupy space for  $k$  children even if most nodes have less children.

- A third option is the so-called *left-child right-sibling* representation in which we have a structure for a node which contains the following:
  - information from the node
  - address of the parent node (not mandatory)
  - address of the leftmost child of the node
  - address of the right sibling of the node (next node on the same level from the same parent).

# Left-child right sibling representation example



- A node of a tree is said to be *visited* when the program control arrives at the node, usually with the purpose of performing some operation on the node (printing it, checking the value from the node, etc.).
- *Traversing* a tree means visiting all of its nodes.
- For a k-ary tree there are 2 possible traversals:
  - Depth-first traversal
  - Level order (breadth first) traversal

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 14

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

# In Lecture 13...

- AVL Trees
- Huffman encoding

- Parenthesis matching
- Perfect hashing
- Cuckoo hashing
- Linked Hash Table
- Conclusions
- Exam info

# Delimiter matching

- Given a sequence of round brackets (parentheses), (square) brackets and curly brackets, verify if the brackets are opened and closed correctly.
- For example:
  - The sequence  $()(())[((())])$  - is correct
  - The sequence  $[()()()()$  - is correct
  - The sequence  $[(())$  - is not correct (one extra closed round bracket at the end)
  - The sequence  $[(])$  - is not correct (brackets closed in wrong order)
  - The sequence  $\{[]\} ()$  - is not correct (curly bracket is not closed)

# Bracket matching - Solution Idea

- Stacks are suitable for this problem, because the bracket that was opened last should be the first to be closed. This matches the LIFO property of the stack.
- The main idea of the solution:
  - Start parsing the sequence, element-by-element
  - If we encounter an open bracket, we push it to a stack
  - If we encounter a closed bracket, we pop the last open bracket from the stack and check if they match
  - If they don't match, the sequence is not correct
  - If they match, we continue
  - If the stack is empty when we finished parsing the sequence, it was correct

# Bracket matching - Extension

- How can we extend the previous idea so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
  - Open brackets that are never closed
  - Closed brackets that were not opened
  - Mismatch

# Bracket matching - Extension

- How can we extend the previous idea so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
  - Open brackets that are never closed
  - Closed brackets that were not opened
  - Mismatch
- Keep count of the current position in the sequence, and push to the stack  $\langle \text{delimiter}, \text{position} \rangle$  pairs.

# Perfect hashing

# Perfect hashing

- Assume that we know all the keys in advance and we use *separate chaining* for collision resolution  $\Rightarrow$  the more lists we make, the shorter the lists will be (reduced number of collisions)  $\Rightarrow$  if we could make a large number of list, each would have one element only (no collision).
- How large should we make the hash table to make sure that there are no collisions?
- If  $M = N^2$ , it can be shown that the table is collision free with probability at least  $1/2$ .
- Start building the hash table. If you detect a collision, just choose a new hash function and start over (expected number of trials is at most 2).

# Perfect hashing

- Having a table of size  $N^2$  is impractical.
- Solution instead:
  - Use a hash table of size  $N$  (*primary hash table*).
  - Instead of using linked list for collision resolution (as in separate chaining) each element of the hash table is another hash table (*secondary hash table*)
  - Make the secondary hash table of size  $n_j^2$ , where  $n_j$  is the number of elements from this hash table.
  - Each secondary hash table will be constructed with a different hash function, and will be reconstructed until it is collision free.
- This is called **perfect hashing**.
- It can be shown that the total space needed for the secondary hash tables is expected to be at most  $2N$  (if it is larger, just pick a different hash function).

# Perfect hashing

- Perfect hashing requires multiple hash functions, this is why we use *universal hashing*.

- Perfect hashing requires multiple hash functions, this is why we use *universal hashing*.
- Let  $p$  be a prime number, larger than the largest possible key.
- The universal hash function family  $\mathcal{H}$  can be defined as:

$$\mathcal{H} = \{H_{a,b}(x) = ((a * x + b) \% p) \% m\}$$

where  $1 \leq a \leq p - 1, 0 \leq b \leq p - 1$

- $a$  and  $b$  are chosen randomly when the hash function is initialized.

# Perfect hashing - example

- Insert into a hash table with perfect hashing the values 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39
- Since we are inserting  $N = 13$  elements, we will take  $m = 13$ .

# Perfect hashing - example

- $p$  has to be a prime number larger than the maximum key  $\Rightarrow 151$
- The hash function will be:

$$H_{a,b}(x) = ((a * x + b) \% p) \% m$$

- where  $a$  will be 3 and  $b$  will be 2 (chosen randomly).

Value	76	12	109	43	22	18	55	81	91	27	13	16	39
H(Value)	1	12	1	1	3	4	3	3	7	5	2	11	2

# Perfect hashing - example

- Collisions:
  - position 1 - 76, 109, 43
  - position 2 - 13, 39
  - position 3 - 22, 55, 81
  - position 4 - 18
  - position 5 - 27
  - position 7 - 91
  - position 11 - 16
  - position 12 - 12
- Sum of the sizes of the secondary hash tables:  $9 + 4 + 9 + 1 + 1 + 1 + 1 = 27$

# Perfect hashing - example

- For the positions where we have no collision (only one element hashed to it) we will have a secondary hash table with only one element and hash function  $h(x) = 0$
- For the positions where we have two elements, we will have a secondary hash table with 4 positions and different hash functions, taken from the same universe, with different random values for  $a$  and  $b$ .
- For example for position 2, we can define  $a = 4$  and  $b = 11$  and we will have:

$$h(13) = 3$$

$$h(39) = 0$$

# Perfect hashing - example

- Assume that for the secondary hash table from position 1 we will choose  $a = 14$  and  $b = 1$ .
- Positions for the elements will be:  
$$h(76) = ((14 * 76 + 1)\%151)\%9 = 8$$
$$h(109) = ((14 * 109 + 1)\%151)\%9 = 8$$
$$h(43) = ((14 * 43 + 1)\%151)\%9 = 6$$
- In perfect hashing we should not have collisions, so we will simply chose another hash function: another random values for  $a$  and  $b$ . Choosing for example  $a = 2$  and  $b = 13$ , we will have  $h(76) = 5$ ,  $h(109) = 8$ ,  $h(43) = 0$ .

- When perfect hashing is used and we search for an element we will have to check at most 2 positions (position in the primary and in the secondary table).
- This means that the worst case performance of the table is  $\Theta(1)$ .
- But in order to use perfect hashing, we need to have static keys: once the table is built, no new elements can be added.

# Dynamic Perfect Hashing

- Traditionally, perfect hashing is said to work in a static environment (you need to know all the keys in advance).
- It is easy to see why: you can build a table to be collision free, by picking new hash functions. But if you allow new additions you might get a collision after you have built the table.
- However, dynamic perfect hashing was also introduced in 1994.
- It obviously implies a lot of rebuilding when a new element is added (the *small* hash table is rebuilt more often, but the primary hash table is also rebuilt after any M operations)

# Cuckoo hashing

# Cuckoo hashing

- In cuckoo hashing we have two hash tables of the same size, each of them more than half empty and each hash table has its hash function (so we have two different hash functions).
- For each element to be added we can compute two positions: one from the first hash table and one from the second. In case of cuckoo hashing, it is guaranteed that an element will be on one of these positions.
- Search is simple, because we only have to look at these two positions.
- Delete is simple, because we only have to look at these two positions and set to empty the one where we find the element.

- When we want to insert a new element we will compute its position in the first hash table. If the position is empty, we will place the element there.
- If the position in the first hash table is not empty, we will kick out the element that is currently there, and place the new element into the first hash table.
- The element that was kicked off, will be placed at its position in the second hash table. If that position is occupied, we will kick off the element from there and place it into its position in the first hash table.
- We repeat the above process until we will get an empty position for an element.
- If we get back to the same location with the same key we have a cycle and we cannot add this element  $\Rightarrow$  resize, rehash

# Cuckoo hashing - example

- Assume that we have two hash tables, with  $m = 11$  positions and the following hash functions:
  - $h1(k) = k \% 11$
  - $h2(k) = (k \text{ div } 11) \% 11$

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Position	0	1	2	3	4	5	6	7	8	9	10
T											

# Cuckoo hashing - example

- Insert key 20

# Cuckoo hashing - example

- Insert key 20
  - $h_1(20) = 9$  - empty position, element added in the first table
- Insert key 50

# Cuckoo hashing - example

- Insert key 20
  - $h_1(20) = 9$  - empty position, element added in the first table
- Insert key 50
  - $h_1(50) = 6$  - empty position, element added in the first table

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			20	

Position	0	1	2	3	4	5	6	7	8	9	10
T											

# Cuckoo hashing - example

- Insert key 53

# Cuckoo hashing - example

- Insert key 53
  - $h_1(53) = 9$  - occupied
  - 53 goes in the first hash table, and it sends 20 in the second to position  $h_2(20) = 1$

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20									

# Cuckoo hashing - example

- Insert key 75

# Cuckoo hashing - example

- Insert key 75
  - $h_1(75) = 9$  - occupied
  - 75 goes in the first hash table, and it sends 53 in the second to position  $h_2(53) = 4$

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53						

# Cuckoo hashing example

- Insert key 100

# Cuckoo hashing example

- Insert key 100
  - $h1(100) = 1$  - empty position
- Insert key 67

# Cuckoo hashing example

- Insert key 100
  - $h_1(100) = 1$  - empty position
- Insert key 67
  - $h_1(67) = 1$  - occupied
  - 67 goes in the first hash table, and it sends 100 in the second to position  $h_2(100) = 9$

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53					100	

# Cuckoo hashing example

- Insert key 105

# Cuckoo hashing example

- Insert key 105
  - $h_1(105) = 6$  - occupied
  - 105 goes in the first hash table, and it sends 50 in the second to position  $h_2(50) = 4$
  - 50 goes in the second hash table, and it sends 53 to the first one, to position  $h_1(53) = 9$
  - 53 goes in the first hash table, and it sends 75 to the second one, to position  $h_2(75) = 6$

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			50		75			100	

# Cuckoo hashing example

- Insert key 3

# Cuckoo hashing example

- Insert key 3
  - $h1(3) = 3$  - empty position
- Insert key 36

# Cuckoo hashing example

- Insert key 3
  - $h_1(3) = 3$  - empty position
- Insert key 36
  - $h_1(36) = 3$  - occupied
  - 36 goes in the first hash table, and it sends 3 in the second to position  $h_2(3) = 0$

Position	0	1	2	3	4	5	6	7	8	9	10
T		67		36			105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20			50		75			100	

# Cuckoo hashing example

- Insert key 39

# Cuckoo hashing example

- Insert key 39
  - $h_1(39) = 6$  - occupied
  - 39 goes in the first hash table and it sends 105 in the second to position  $h_2(105) = 9$
  - 105 goes to the second hash table and it sends 100 in the first to position  $h_1(100) = 1$
  - 100 goes in the first hash table and it sends 67 in the second to position  $h_2(67) = 6$
  - 67 goes in the second hash table and it sends 75 in the first to position  $h_1(75) = 9$
  - 75 goes in the first hash table and it sends 53 in the second to position  $h_2(53) = 4$
  - 53 goes in the second hash table and it sends 50 in the first to position  $h_1(50) = 6$
  - 50 goes in the first hash table and it sends 39 in the second to position  $h_2(39) = 3$

# Cuckoo hashing example

Position	0	1	2	3	4	5	6	7	8	9	10
T		100		36			50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20		39	53		67			105	

- It can happen that we cannot insert a key because we get in a cycle. In these situations we have to increase the size of the tables and rehash the elements.
- While in some situations insert moves a lot of elements, it can be shown that if the load factor of the tables is below 0.5, the probability of a cycles is low and it is very unlikely that more than  $O(\log_2 n)$  elements will be moved.

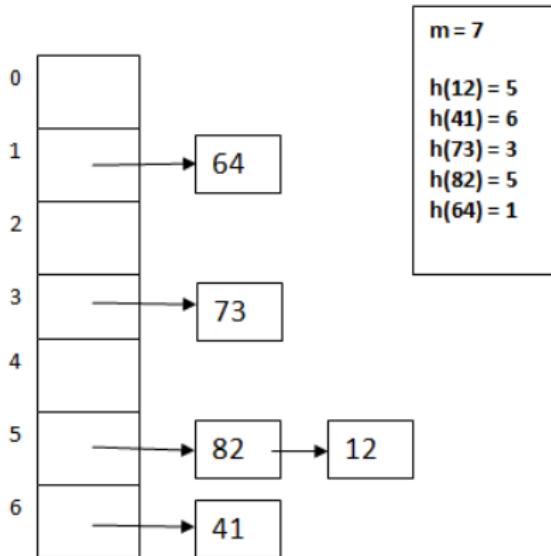
- If we use two tables and each position from a table holds one element at most, the tables have to have load factor below 0.5 to work well.
- If we use three tables, the tables can have load factor of 0.91 and for 4 tables we have 0.97

# Linked Hash Table

# Linked Hash Table

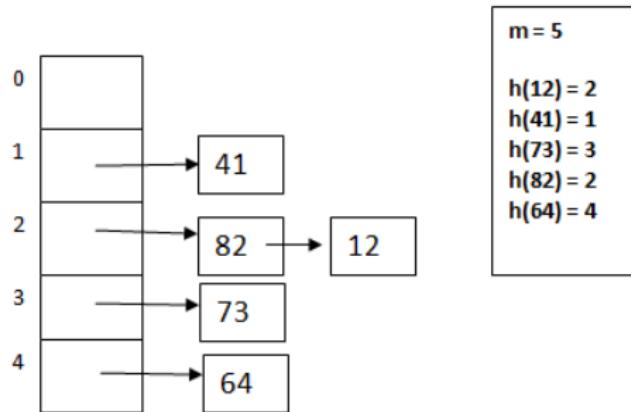
- Assume we build a hash table using separate chaining as a collision resolution method.
- We have discussed how an iterator can be defined for such a hash table.
- When iterating through the elements of a hash table, the order in which the elements are visited is *undefined*
- For example:
  - Assume an initially empty hash table (we do not know its implementation)
  - Insert one-by-one the following elements: 12, 41, 73, 82, 64
  - Use an iterator to display the content of the hash table
  - In what order will the elements be displayed?

# Linked Hash Table



- Iteration order: 64, 73, 82, 12, 41

# Linked Hash Table



- Iteration order: 41, 82, 12, 73, 64

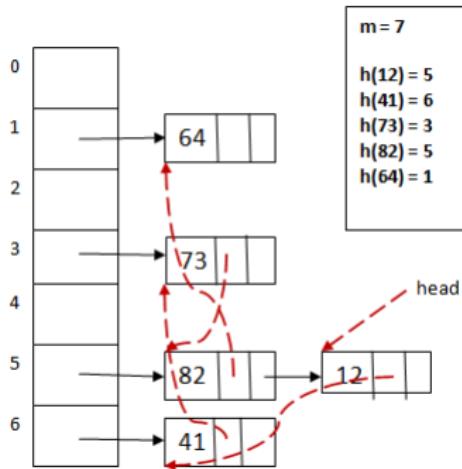
# Linked Hash Table

- A *linked hash table* is a data structure which has a *predictable* iteration order. This order is the order in which elements were inserted.
- So if we insert the elements 12, 41, 73, 82, 64 (in this order) in a linked hash table and iterate over the hash table, the iteration order is guaranteed to be: 12, 41, 73, 82, 64.
- How could we implement a linked hash table which provides this iteration order?

# Linked Hash Table

- A linked hash table is a combination of a hash table and a linked list. Besides being stored in the hash table, each element is part of a linked list, in which the elements are added in the order in which they are inserted in the table.
- Since it is still a hash table, we want to have, on average,  $\Theta(1)$  for insert, remove and search, these are done in the same way as before, the extra linked list is used only for iteration.

# Linked Hash Table



- Red arrows show how the elements are linked in insertion order, starting from a *head* - the first element that was inserted, 12.

# Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

# Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).
- The only operation that cannot be efficiently implemented if we have a singly linked list is the *remove* operation. When we remove an element from a singly linked list we need the element before it, but finding this in our linked hash table takes  $O(n)$  time.

# Linked Hash Table - Implementation

- What structures do we need to implement a Linked Hash Table?

## Node:

info: TKey

nextH:  $\uparrow$  Node *//pointer to next node from the collision*

nextL:  $\uparrow$  Node *//pointer to next node from the insertion-order list*

prevL:  $\uparrow$  Node *//pointer to prev node from the insertion-order list*

## LinkedHT:

m:Integer

T:( $\uparrow$  Node)[]

h:TFunction

head:  $\uparrow$  Node

tail:  $\uparrow$  Node

# Linked Hash Table - Insert

- How can we implement the *insert* operation?

```
subalgorithm insert(lht, k) is:
    //pre: lht is a LinkedHT, k is a key
    //post: k is added into lht
    allocate(newNode)
    [newNode].info ← k
    @set all pointers of newNode to NIL
    pos ← lht.h(k)
    //first insert newNode into the hash table
    if lht.T[pos] = NIL then
        lht.T[pos] ← newNode
    else
        [newNode].nextH ← lht.T[pos]
        lht.T[pos] ← newNode
    end-if
    //continued on the next slide...
```

# Linked Hash Table - Insert

```
//now insert newNode to the end of the insertion-order list
if lht.head = NIL then
    lht.head ← newNode
    lht.tail ← newNode
else
    [newNode].prevL ← lht.tail
    [lht.tail].nextL ← newNode
    lht.tail ← newNode
end-if
end-subalgorithm
```

# Linked Hash Table - Remove

- How can we implement the *remove* operation?

**subalgorithm** remove(lht, k) **is:**

//*pre: lht is a LinkedHT, k is a key*

//*post: k was removed from lht*

pos  $\leftarrow$  lht.h(k)

current  $\leftarrow$  lht.T[pos]

nodeToBeRemoved  $\leftarrow$  NIL

//*first search for k in the collision list and remove it if found*

**if** current  $\neq$  NIL **and** [current].info = k **then**

    nodeToBeRemoved  $\leftarrow$  current

    lht.T[pos]  $\leftarrow$  [current].nextH

**else**

    prevNode  $\leftarrow$  NIL

**while** current  $\neq$  NIL **and** [current].info  $\neq$  k **execute**

        prevNode  $\leftarrow$  current

        current  $\leftarrow$  [current].nextH

**end-while**

//*continued on the next slide...*

```
if current ≠ NIL then
    nodeToBeRemoved ← current
    [prevNode].nextH ← [current].nextH
else
    @k is not in lht
end-if
end-if

//if k was in lht then nodeToBeRemoved is the address of the node containing
//it and the node was already removed from the collision list - we need to
//remove it from the insertion-order list as well

if nodeToBeRemoved ≠ NIL then
    if nodeToBeRemoved = lht.head then
        if nodeToBeRemoved = lht.tail then
            lht.head ← NIL
            lht.tail ← NIL
        else
            lht.head ← [lht.head].nextL
            [lht.head].prev ← NIL
        end-if
    end-if
//continued on the next slide...
```

```
else if nodeToBeRemoved = lht.tail then
    lht.tail ← [lht.tail].prev
    [lht.tail].next ← NIL
else
    [[nodeToBeRemoved].next].prev ← [nodeToBeRemoved].prev
    [[nodeToBeRemoved].prev].next ← [nodeToBeRemoved].next
end-if
deallocate(nodeToBeRemoved)
end-if
end-subalgorithm
```

# Conclusions

- During the semester we have talked about the most important containers (ADT) and their main properties and operations
  - Bag, Set, Map, Multimap, List, Stack, Queue and their sorted versions
- We have also talked about the most important data structures that can be used to implement these containers
  - Dynamic array, Linked lists, Binary heap, Hash table, Binary Search Tree

# Conclusions

- You should be able to identify the most suitable container for solving a given problem:

# Conclusions

- You should be able to identify the most suitable container for solving a given problem:
- Example: *You have a type Student which has a name and a city. Write a function which takes as input a list of students and prints for each city all the students that are from that city. Each city should be printed only once and in any order.*
- How would you solve the problem? What container would you use?

# Conclusions

- When you use containers existing in different programming languages, you should have an idea of how they are implemented and what is the complexity of their operations:

# Conclusions

- When you use containers existing in different programming languages, you should have an idea of how they are implemented and what is the complexity of their operations:
- Consider the following algorithm (written in Python):

```
def testContainer(container, l):  
    """  
    container is a container with integer numbers  
    l is a list with integer numbers  
    """  
    count = 0  
    for elem in l:  
        if elem in container:  
            count += 1  
    return count
```

- The above function counts how many elements from the list / can be found in the container. What is the complexity of *testContainer*?

- Consider the following problem: *We want to model the content of a wallet, by using a list of integer numbers, in which every value denotes a bill. For example, a list with values [5, 1, 50, 1, 5] means that we have 62 RON in our wallet.*

*Obviously, we are not allowed to have any numbers in our list, only numbers corresponding to actual bills (we cannot have a value of 8 in the list, because there is no 8 RON bill).*

*We need to implement a functionality to pay a given amount of sum and to receive rest of necessary.*

*There are many optimal algorithms for this, but we go for a very simple (and non-optimal): keep removing bills of the wallet until the sum of removed bills is greater than or equal to the sum you want to pay.*

*If we need to receive a rest, we will receive it in 1 RON bills.*

# Conclusions

- For example, if the wallet contains the values [5, 1, 50, 1, 5] and we need to pay 43 RON, we might remove the first 3 bills (a total of 56) and receive the 13 RON rest in 13 bills of 1.

# Conclusions

- For example, if the wallet contains the values [5, 1, 50, 1, 5] and we need to pay 43 RON, we might remove the first 3 bills (a total of 56) and receive the 13 RON rest in 13 bills of 1.
- This is an implementation provided by a student. What is wrong with it?

```
public void spendMoney(ArrayList<Integer> wallet, Integer amount) {  
    Integer spent = 0;  
    while (spent < amount) {  
        Integer bill = wallet.remove(0); //removes element from position 0  
        spent += bill;  
    }  
    Integer rest = spent - amount;  
    while (rest > 0) {  
        wallet.add(0, 1);  
        rest--;  
    }  
}
```

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 13

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2020 - 2021

# In Lecture 12...

- Binary Trees
- Binary Search Trees

- AVL Trees
- Misc

# Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation  $\leq$ , insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.

# Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation  $\leq$ , insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.
- How would you count how many times the value 5 is in the tree?

# Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation  $\leq$ , insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.
- How would you count how many times the value 5 is in the tree?
- Remove 3 (show both options)

# Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation  $\leq$ , insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.
- How would you count how many times the value 5 is in the tree?
- Remove 3 (show both options)
- How would you count now how many times the value 5 is in the tree now?

# Balanced Binary Search Trees

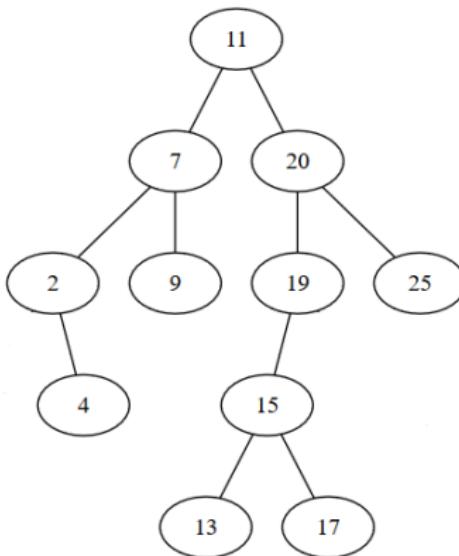
- Specific operations for binary trees run in  $O(h)$  time, which can be  $\theta(n)$  in worst case
- Best case is a balanced tree, where height of the tree is  $\Theta(\log_2 n)$

# Balanced Binary Search Trees

- Specific operations for binary trees run in  $O(h)$  time, which can be  $\theta(n)$  in worst case
- Best case is a balanced tree, where height of the tree is  $\Theta(\log_2 n)$
- To reduce the complexity of algorithms, we want to keep the tree balanced. In order to do this, we want every node to be balanced.
- When a node loses its balance, we will perform some operations (called rotations) to make it balanced again.

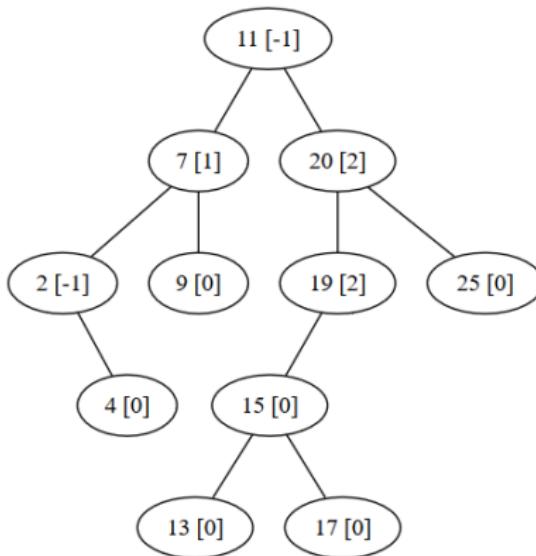
- Definition: An AVL (Adelson-Velskii Landis) tree is a binary tree which satisfies the following property (AVL tree property):
  - If  $x$  is a node of the AVL tree:
    - the difference between the height of the left and right subtree of  $x$  is 0, 1 or -1 (balancing information)
- Observations:
- Height of an empty tree is -1
- Height of a single node is 0

# AVL Trees



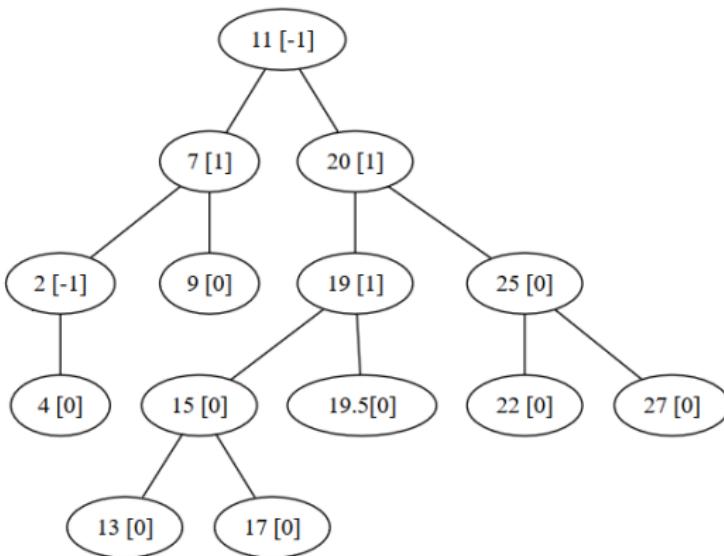
- Is this an AVL tree?

# AVL Trees



- Values in square brackets show the balancing information of a node. The tree is not an AVL tree, because the balancing information for nodes 19 and 20 is 2.

# AVL Trees

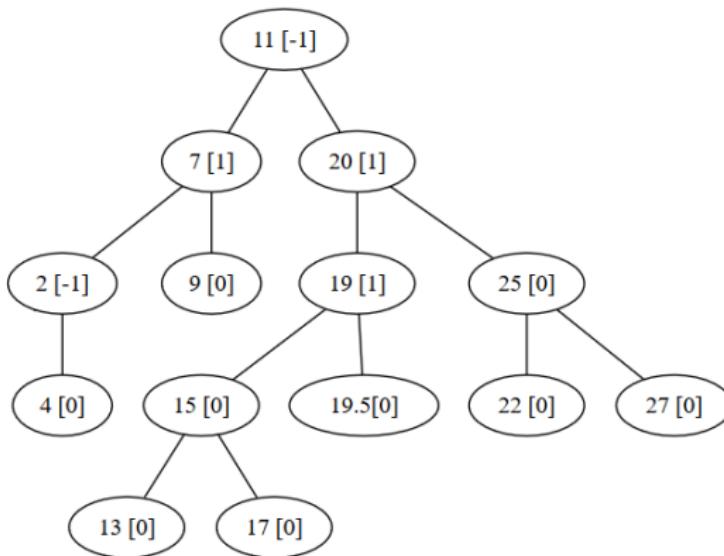


- This is an AVL tree.

- Adding or removing a node might result in a binary tree that violates the AVL tree property.
- In such cases, the property has to be restored and only after the property holds again is the operation (add or remove) considered finished.
- The AVL tree property can be restored with operations called **rotations**.

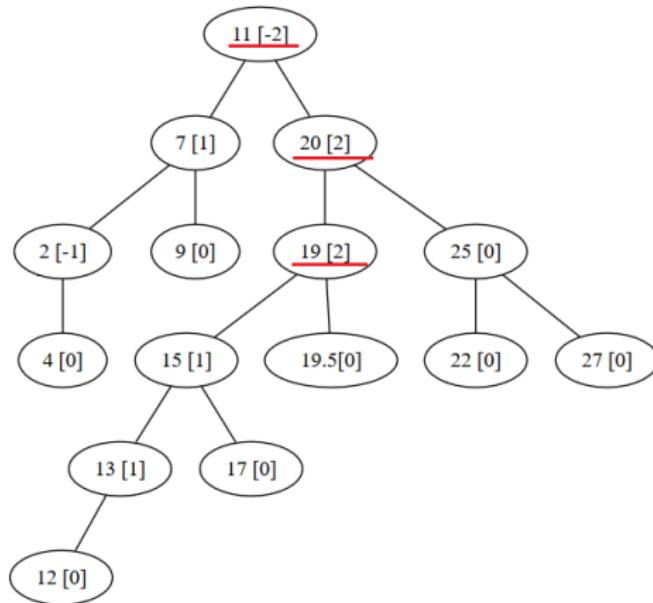
- After an insertion, only the nodes on the path to the modified node can change their height.
- We check the balancing information on the path from the modified node to the root. When we find a node that does not respect the AVL tree property, we perform a suitable *rotation* to rebalance the (sub)tree.

# AVL Trees - rotations



- What if we insert element 12?

# AVL Trees - rotations

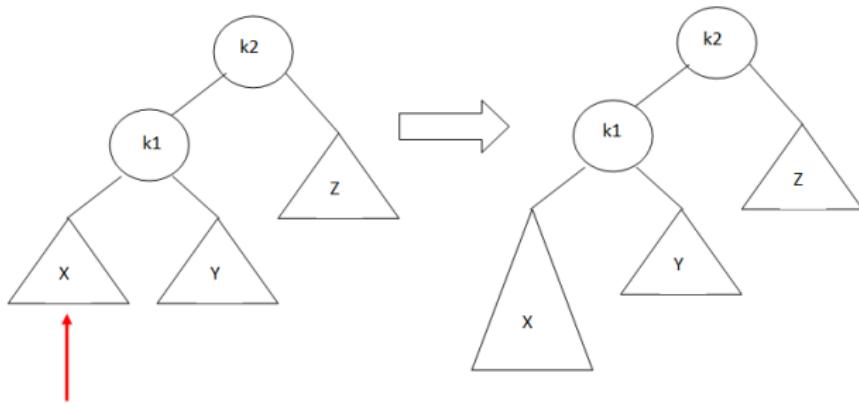


- Red lines show the unbalanced nodes. We will rebalance node 19.

# AVL Trees - rotations

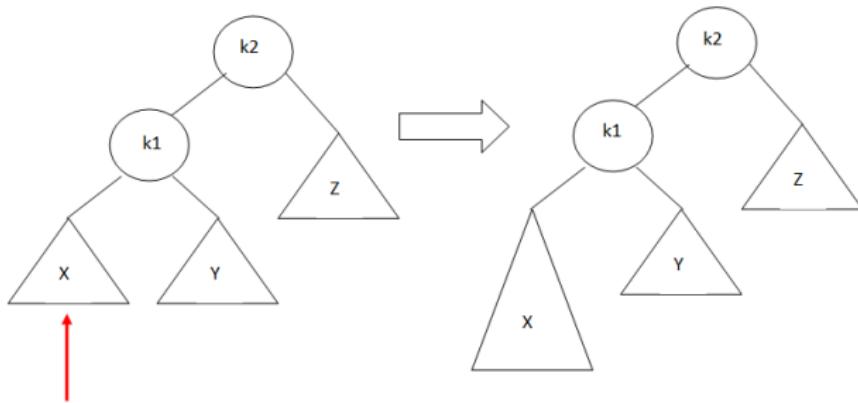
- Assume that at a given point  $\alpha$  is the node that needs to be rebalanced.
- Since  $\alpha$  was balanced before the insertion, and is not after the insertion, we can identify four cases in which a violation might occur:
  - Insertion into the left subtree of the left child of  $\alpha$
  - Insertion into the right subtree of the left child of  $\alpha$
  - Insertion into the left subtree of the right child of  $\alpha$
  - Insertion into the right subtree of the right child of  $\alpha$

# AVL Trees - rotations - case 1



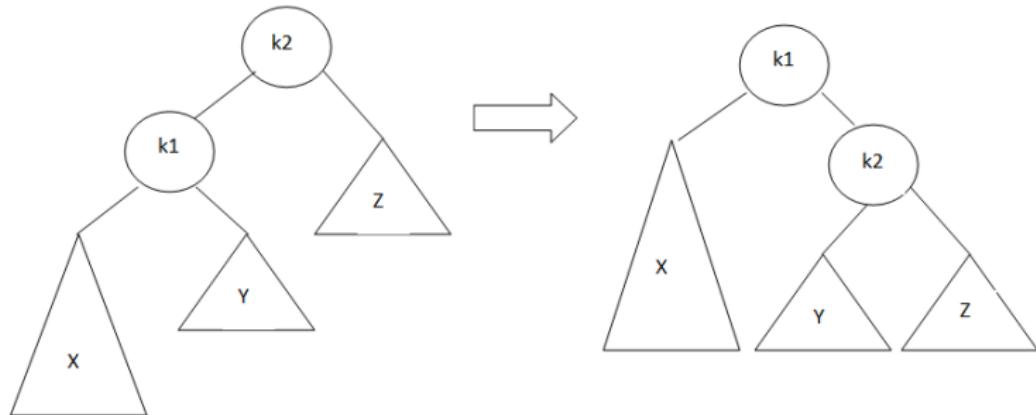
- Obs: X, Y and Z represent subtrees with the same height.

# AVL Trees - rotations - case 1

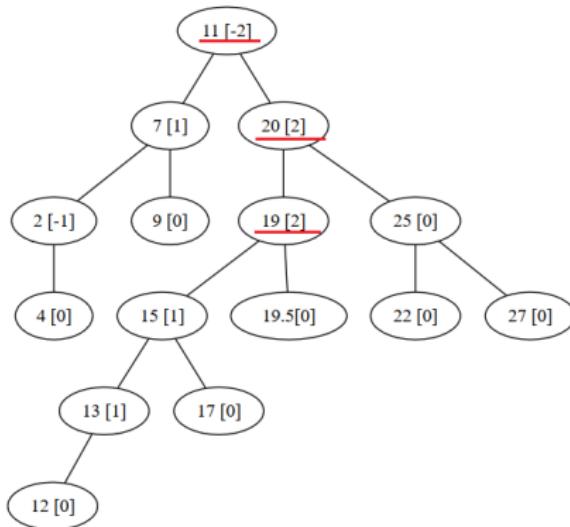


- Obs: X, Y and Z represent subtrees with the same height.
- Solution: single rotation to right

# AVL Trees - rotation - Single Rotation to Right

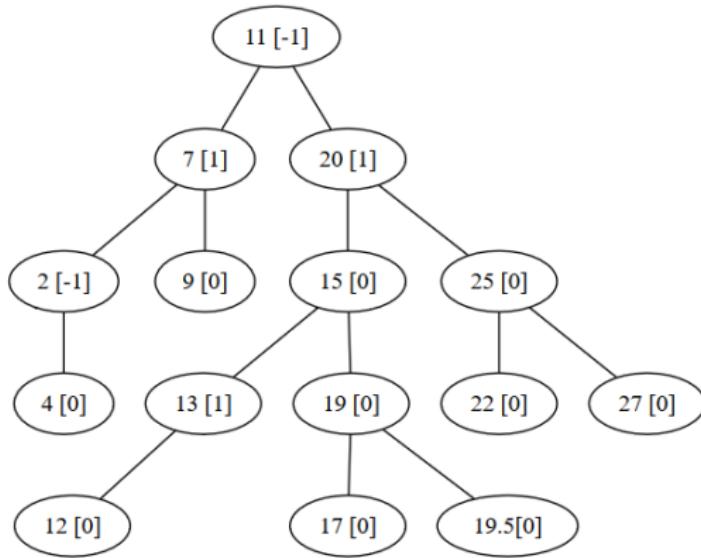


# AVL Trees - rotations - case 1 example

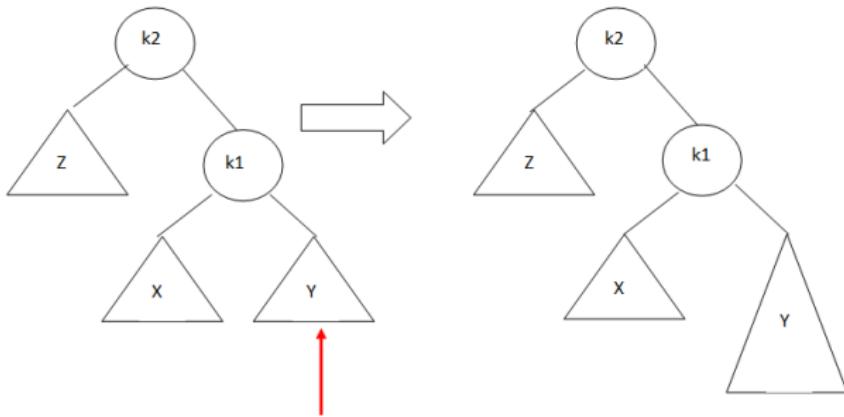


- Node 19 is imbalanced, because we inserted a new node (12) in the left subtree of the left child.
- Solution: **single rotation to right**

# AVL Trees - rotation - case 1 example

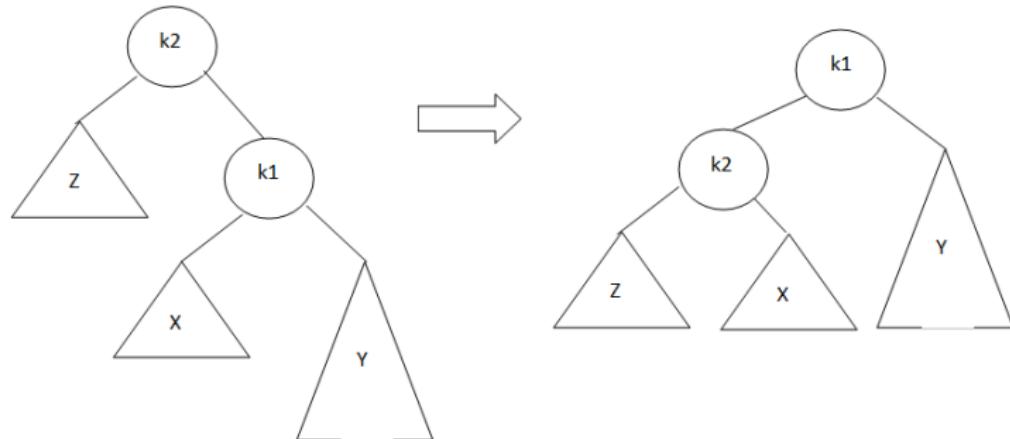


# AVL Trees - rotations - case 4

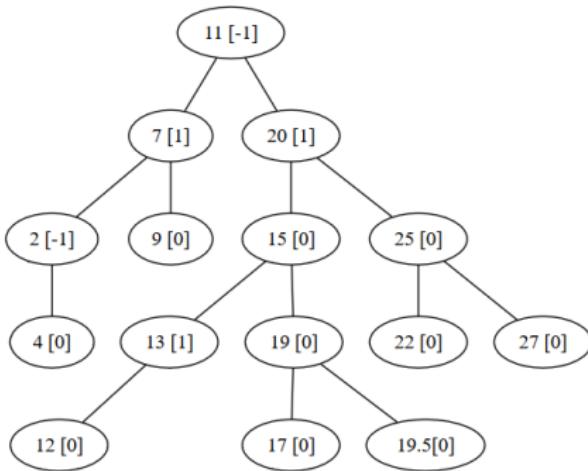


- Solution: **single rotation to left**

# AVL Trees - rotation - Single Rotation to Left

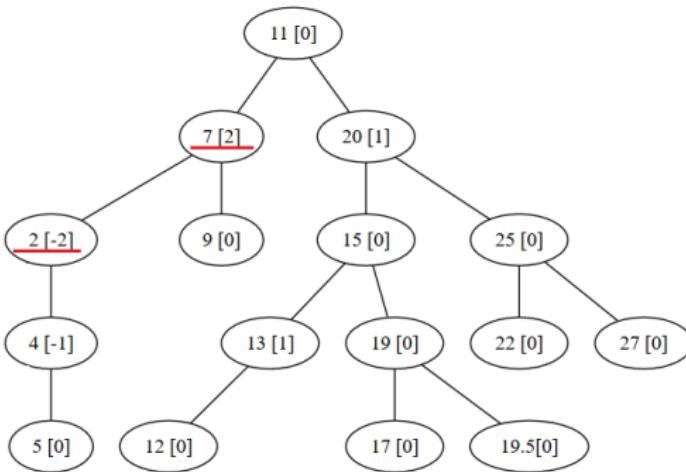


# AVL Trees - rotations - case 4 example



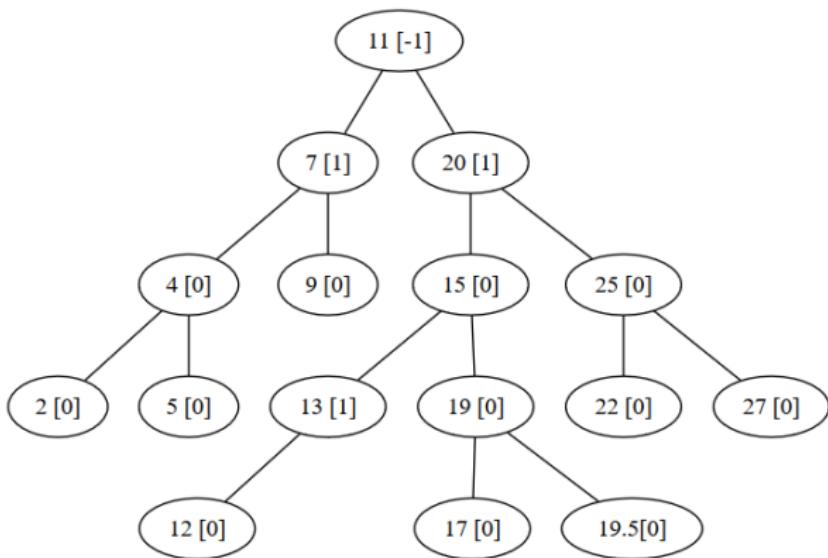
- Insert value 5

# AVL Trees - rotations - case 4 example



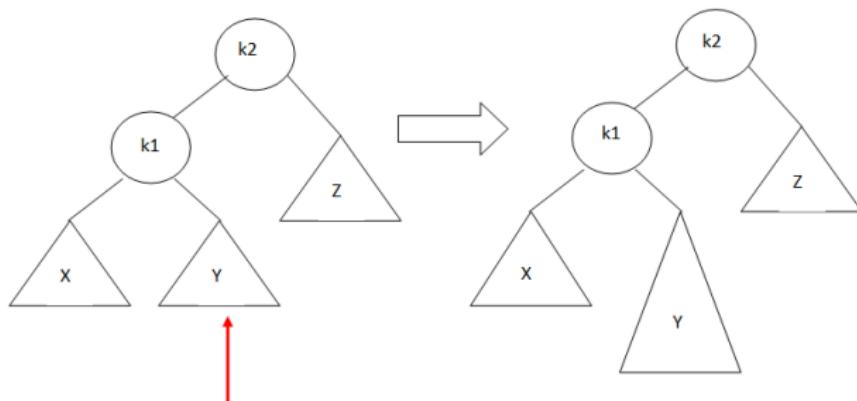
- Node 2 is imbalanced, because we inserted a new node (5) to the right subtree of the right child
- Solution: **single rotation to left**

# AVL Trees - rotation - case 4 example



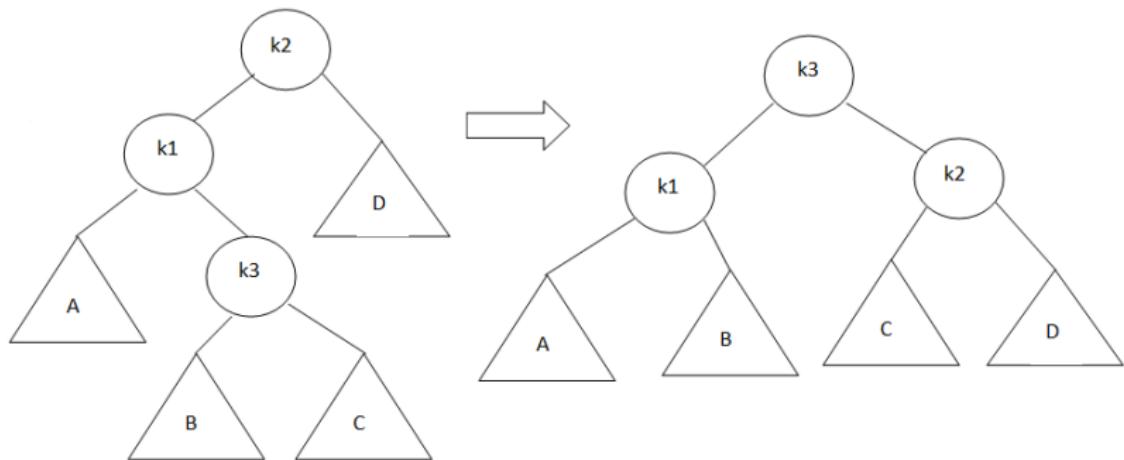
- After the rotation

# AVL Trees - rotations - case 2

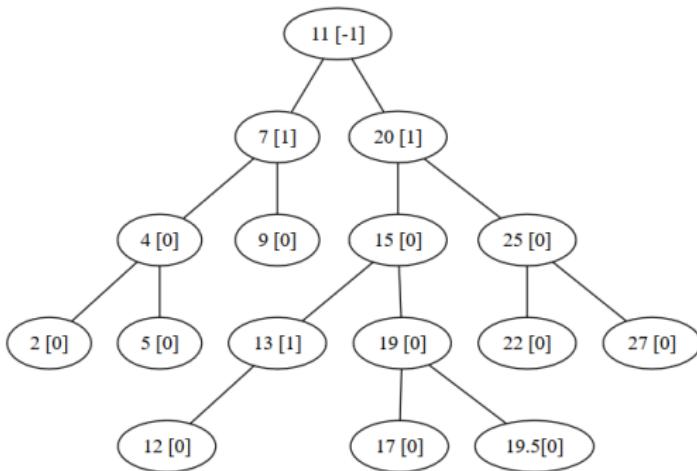


- Solution: **Double rotation to right**

# AVL Trees - rotation - Double Rotation to Right

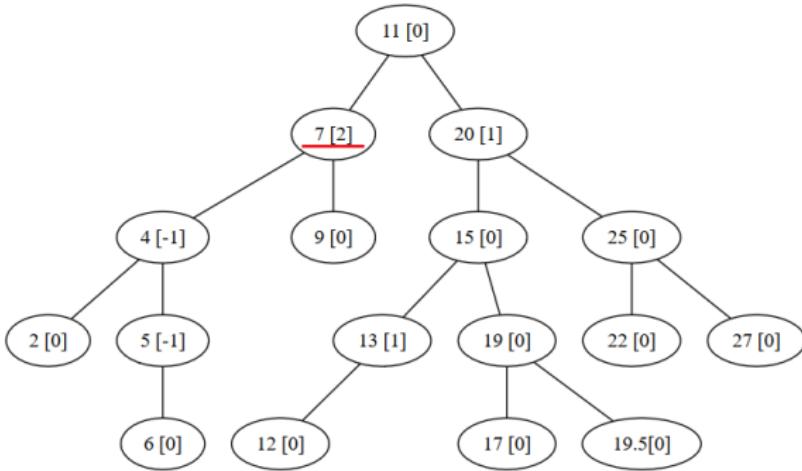


# AVL Trees - rotations - case 2 example



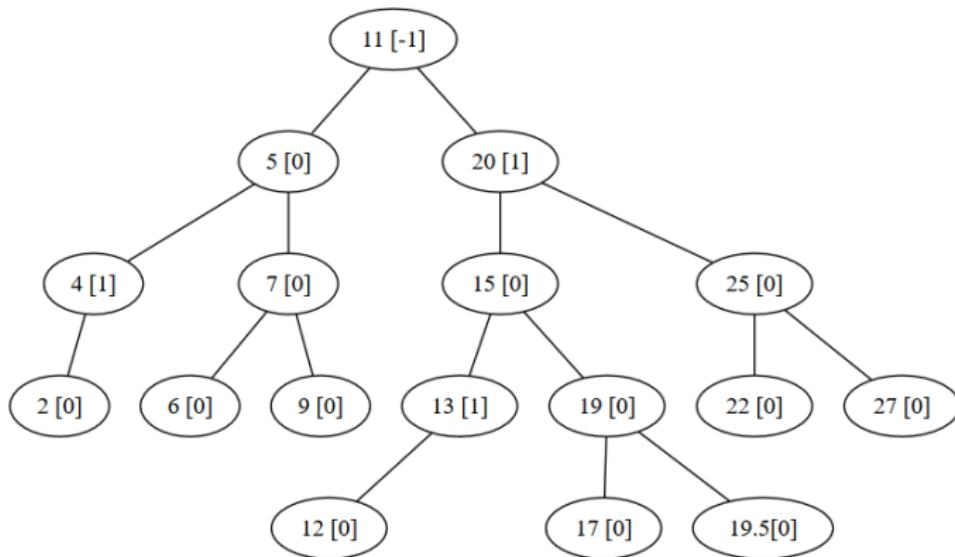
- Insert value 6

# AVL Trees - rotations - case 2 example



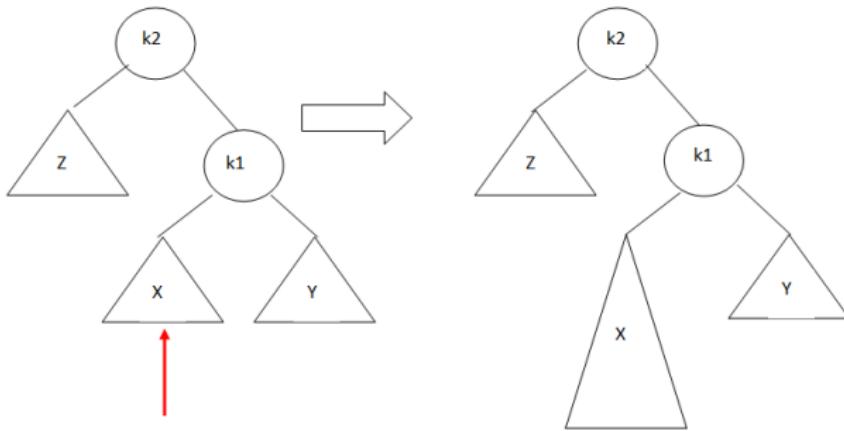
- Node 7 is imbalanced, because we inserted a new node (6) to the right subtree of the left child
- Solution: **double rotation to right**

# AVL Trees - rotation - case 2 example



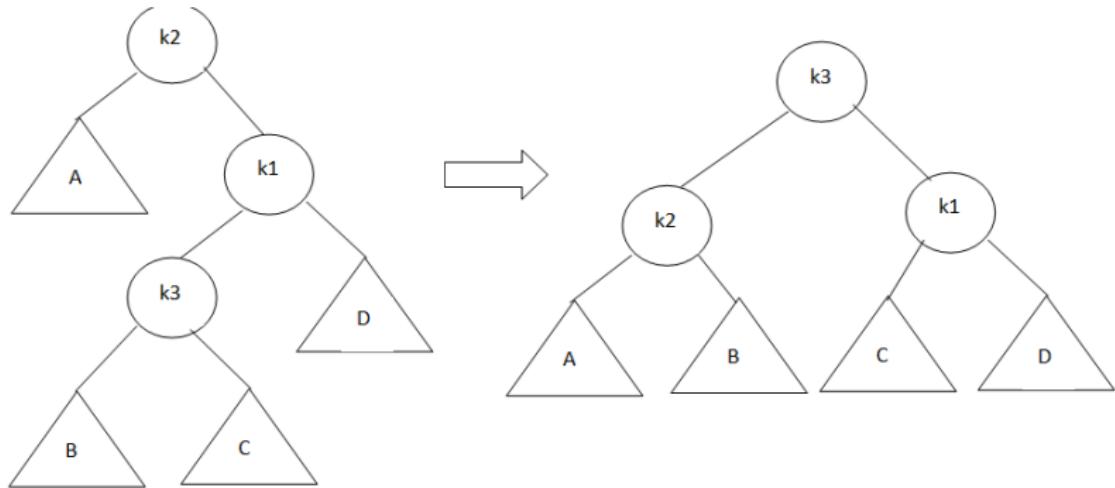
- After the rotation

# AVL Trees - rotations - case 3

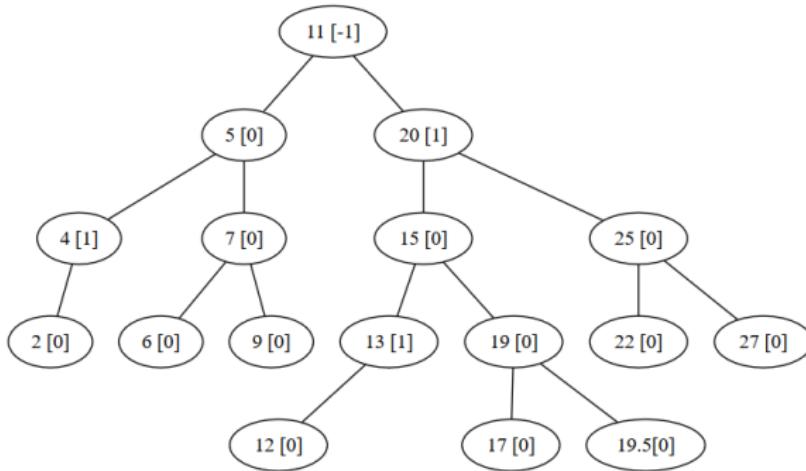


- Solution: **Double rotation to left**

# AVL Trees - rotation - Double Rotation to Left

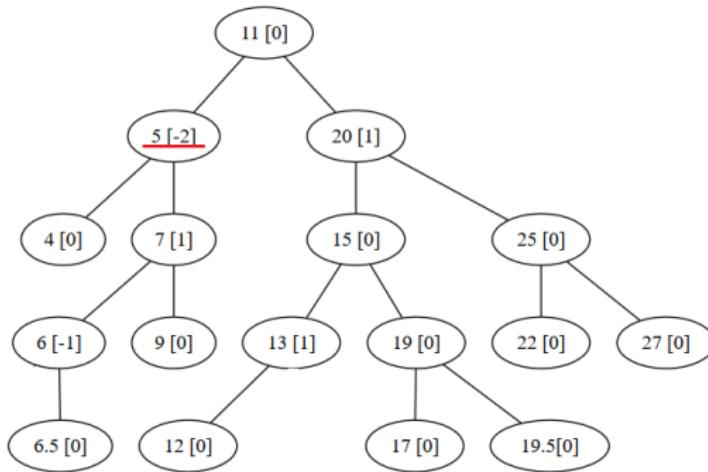


# AVL Trees - rotations - case 3 example



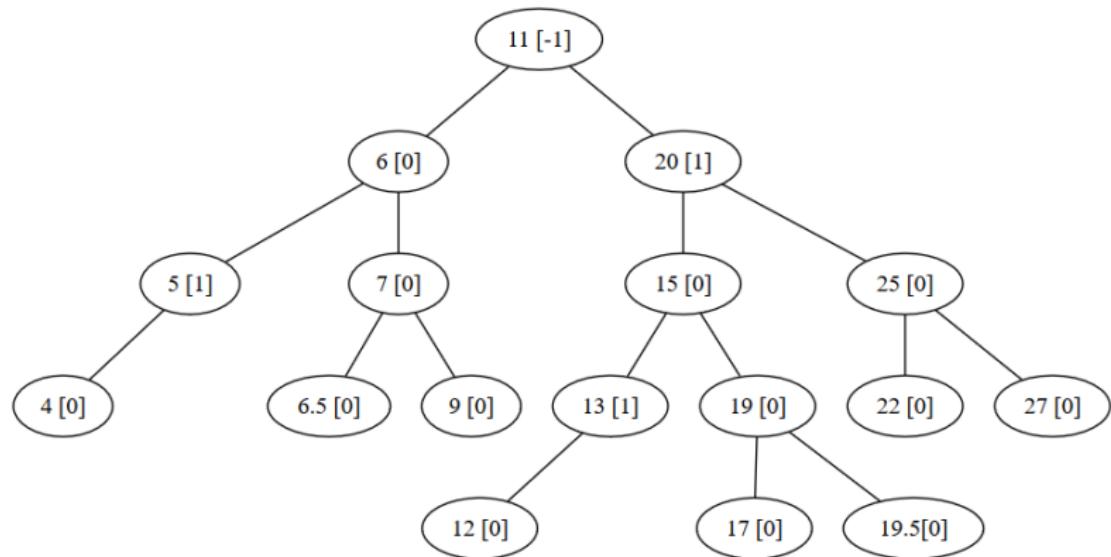
- Remove node with value 2 and insert value 6.5

# AVL Trees - rotations - case 3 example



- Node 5 is imbalanced, because we inserted a new node (6.5) to the left subtree of the right child
- Solution: **double rotation to left**

# AVL Trees - rotation - case 3 example



- After the rotation

# AVL rotations example I

- Start with an empty AVL tree
- Insert 2

## AVL rotations example II

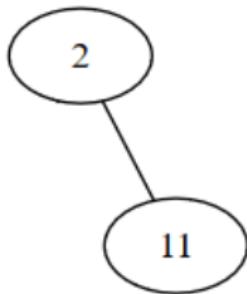


- Do we need a rotation?
- If yes, on which node and what type of rotation?

# AVL rotations example III

- No rotation is needed
- Insert 11

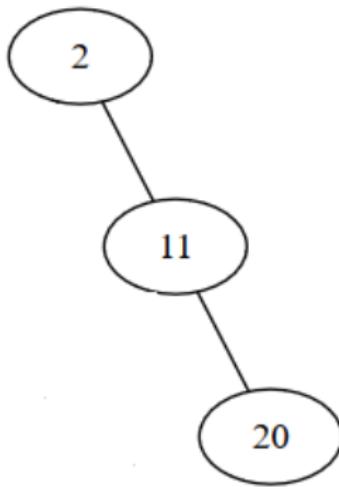
# AVL rotations example IV



- Do we need a rotation?
- If yes, on which node and what type of rotation?

- No rotation is needed
- Insert 20

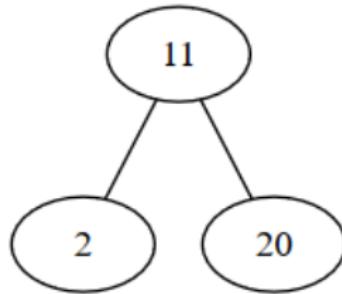
## AVL rotations example VI



- Do we need a rotation?
- If yes, on which node and what type of rotation?

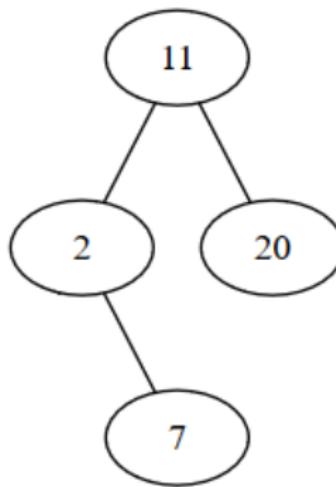
## AVL rotations example VII

- Yes, we need a single left rotation on node 2
- After the rotation:



- Insert 7

## AVL rotations example VIII

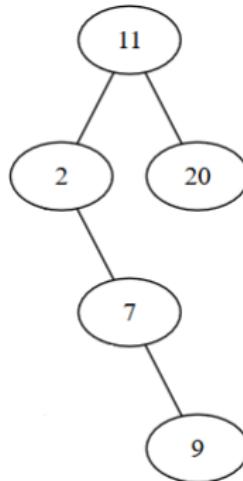


- Do we need a rotation?
- If yes, on which node and what type of rotation?

# AVL rotations example IX

- No rotation is needed
- Insert 9

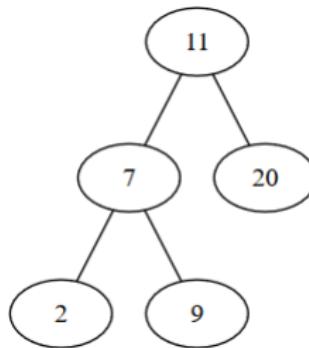
# AVL rotations example X



- Do we need a rotation?
- If yes, on which node and what type of rotation?

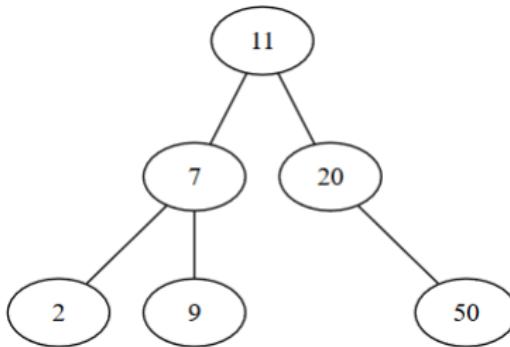
# AVL rotations example XI

- Yes, we need a single left rotation on node 2
- After the rotation:



- Insert 50

## AVL rotations example XII

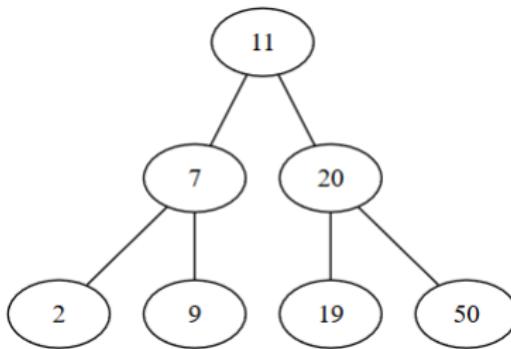


- Do we need a rotation?
- If yes, on which node and what type of rotation?

# AVL rotations example XIII

- No rotation is needed
- Insert 19

## AVL rotations example XIV

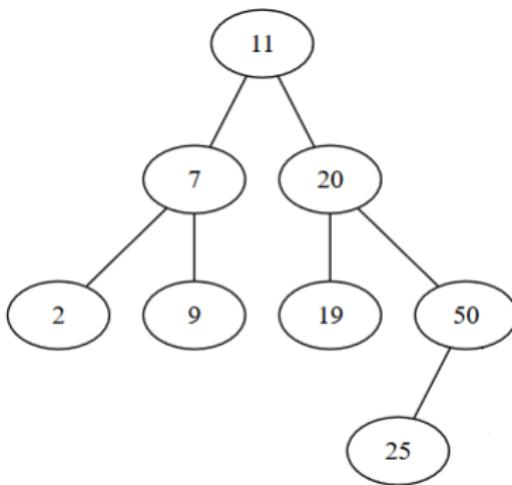


- Do we need a rotation?
- If yes, on which node and what type of rotation?

# AVL rotations example XV

- No rotation is needed
- Insert 25

# AVL rotations example XVI

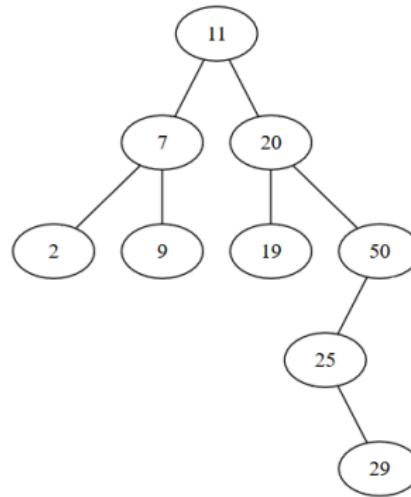


- Do we need a rotation?
- If yes, on which node and what type of rotation?

# AVL rotations example XVII

- No rotation is needed
- Insert 29

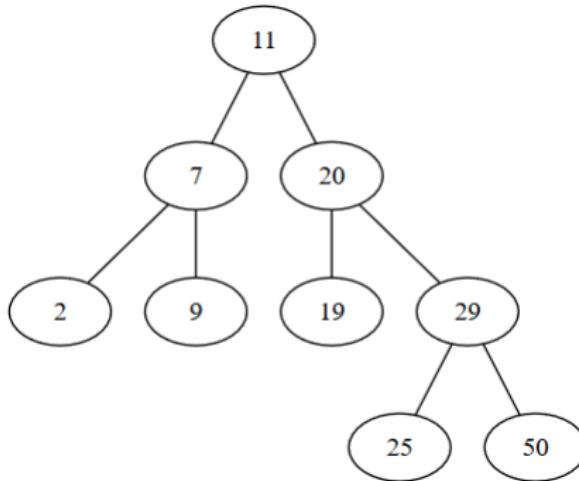
# AVL rotations example XVIII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

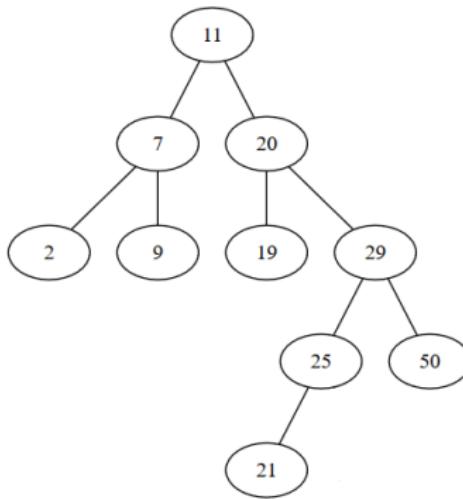
# AVL rotations example XIX

- Yes, we need a double right rotation on node 50
- After the rotation



- Insert 21

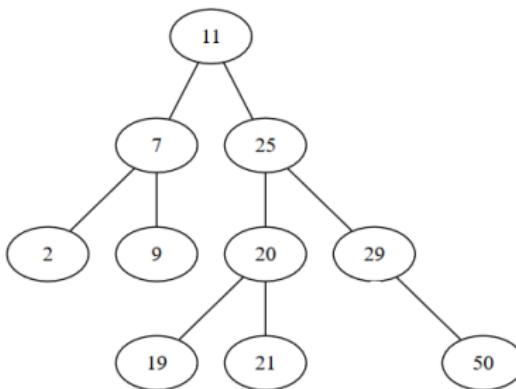
# AVL rotations example XX



- Do we need a rotation?
- If yes, on which node and what type of rotation?

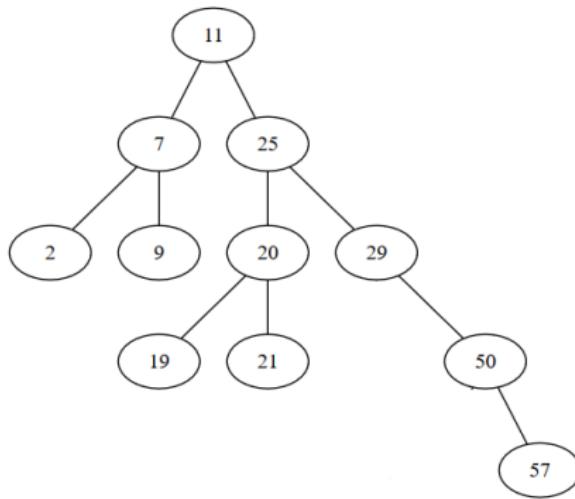
# AVL rotations example XXI

- Yes, we need a double left rotation on node 20
- After the rotation



- Insert 57

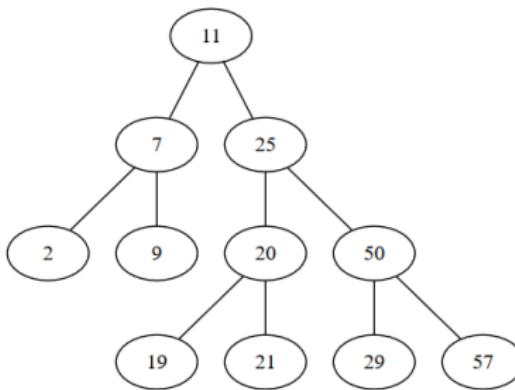
# AVL rotations example XXII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

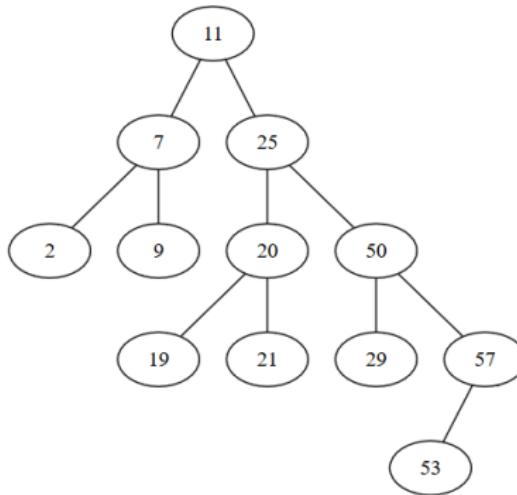
## AVL rotations example XXIII

- Yes, we need a single left rotation on node 50
- After the rotation



- Insert 53

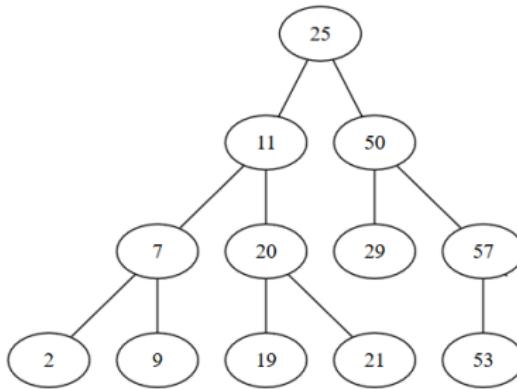
# AVL rotations example XXIV



- Do we need a rotation?
- If yes, on which node and what type of rotation?

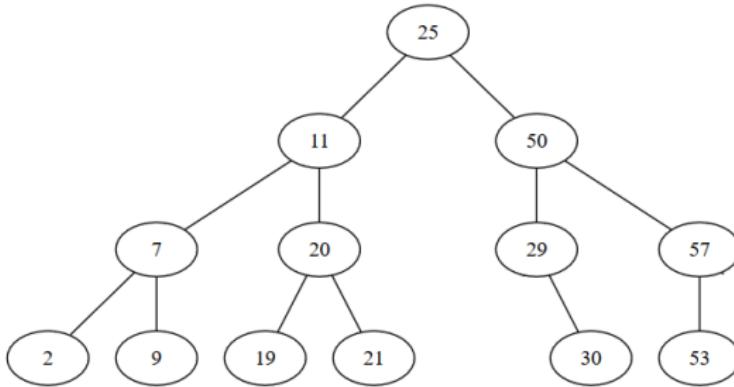
# AVL rotations example XXV

- Yes, we need a single left rotation on node 11
- After the rotation



- Insert 30

# AVL rotations example XXVI

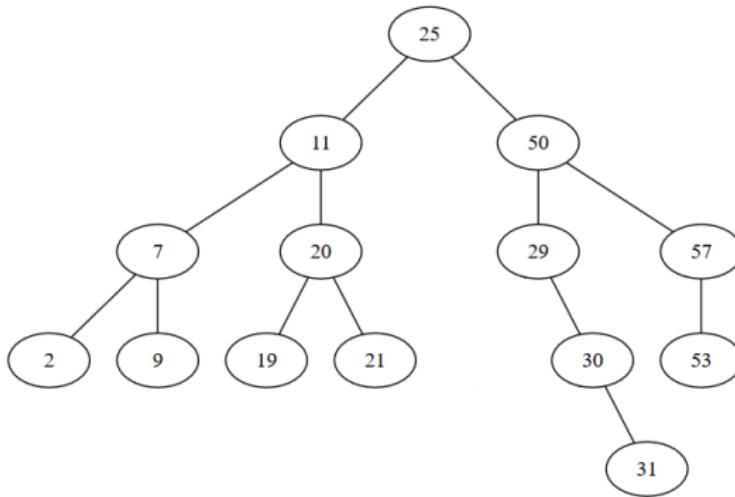


- Do we need a rotation?
- If yes, on which node and what type of rotation?

# AVL rotations example XXVII

- No rotation is needed
- Insert 31

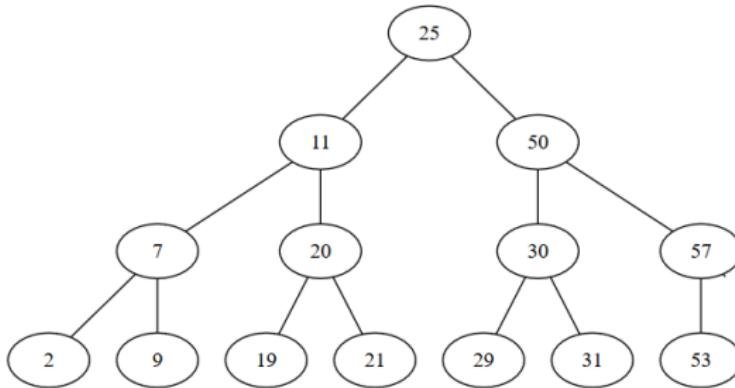
## AVL rotations example XXVIII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

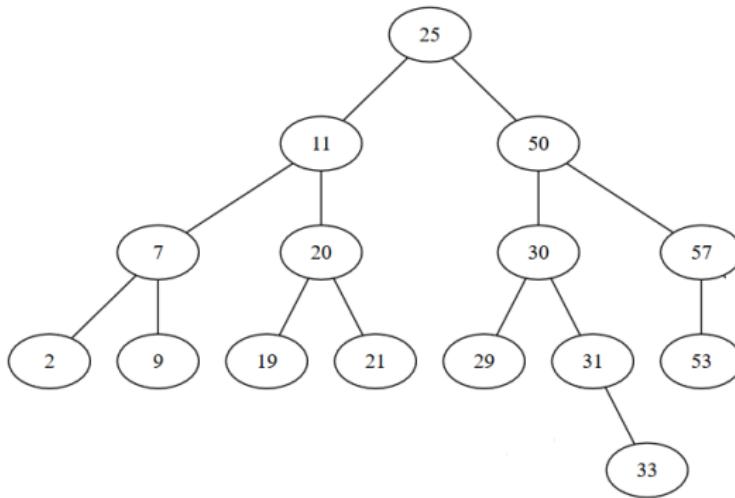
# AVL rotations example XXIX

- Yes, we need a single left rotation on node 29
- After the rotation



- Insert 33

# AVL rotations example XXX

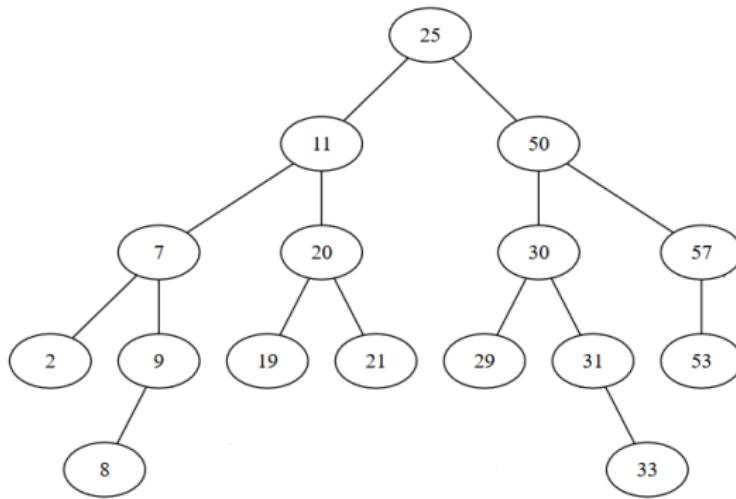


- Do we need a rotation?
- If yes, on which node and what type of rotation?

# AVL rotations example XXXI

- No rotation is needed
- Insert 8

## AVL rotations example XXXII

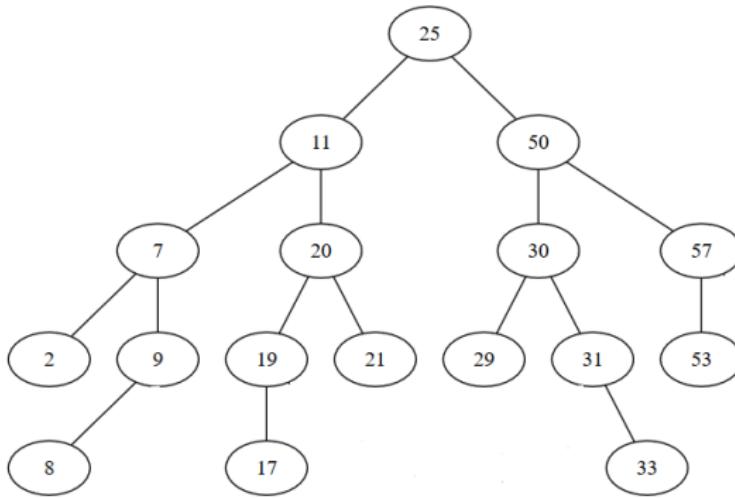


- Do we need a rotation?
- If yes, on which node and what type of rotation?

## AVL rotations example XXXIII

- No rotation is needed
- Insert 17

## AVL rotations example XXXIV

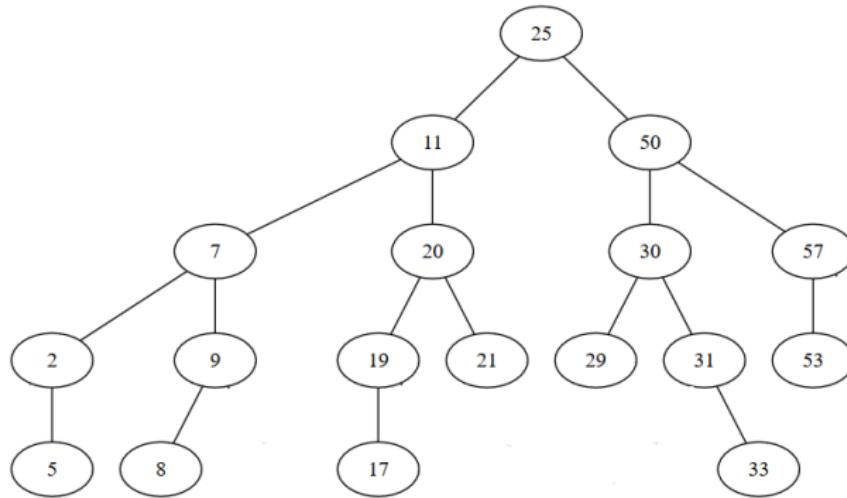


- Do we need a rotation?
- If yes, on which node and what type of rotation?

# AVL rotations example XXXV

- No rotation is needed
- Insert 5

# AVL rotations example XXXVI

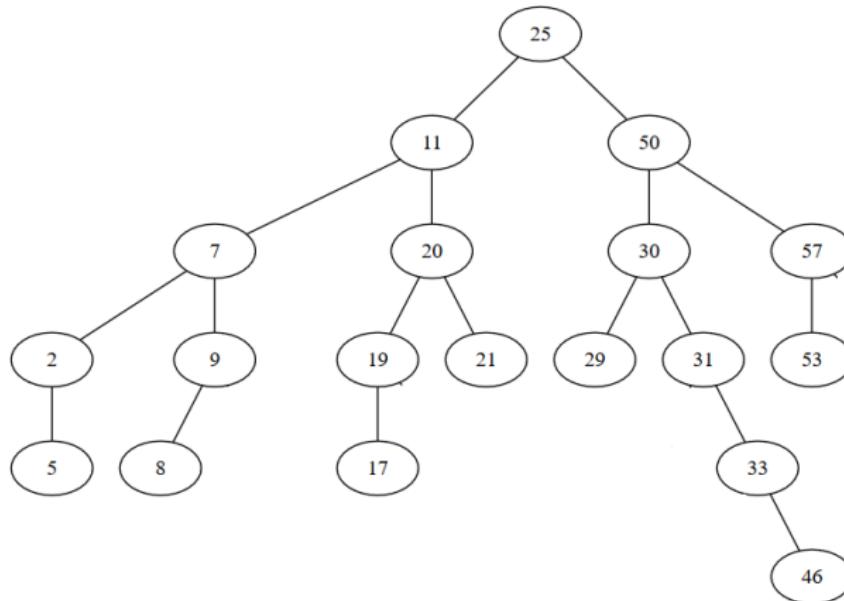


- Do we need a rotation?
- If yes, on which node and what type of rotation?

# AVL rotations example XXXVII

- No rotation is needed
- Insert 46

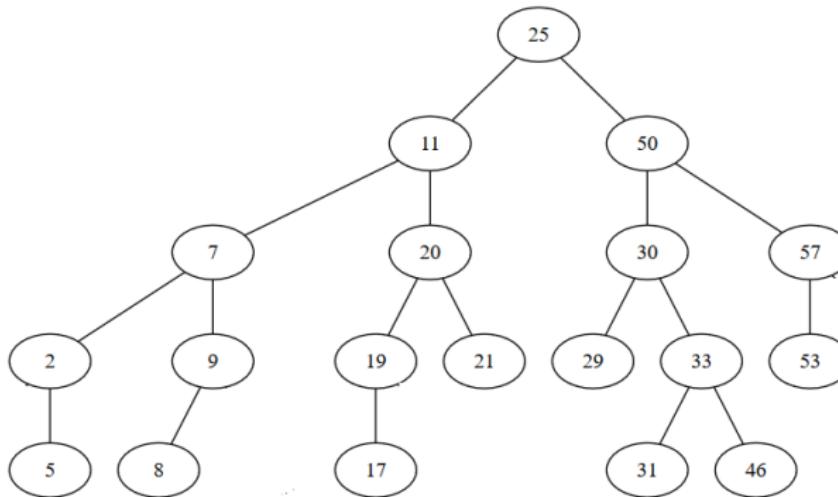
# AVL rotations example XXXVIII



- Do we need a rotation?
- If yes, on which node and what type of rotation?

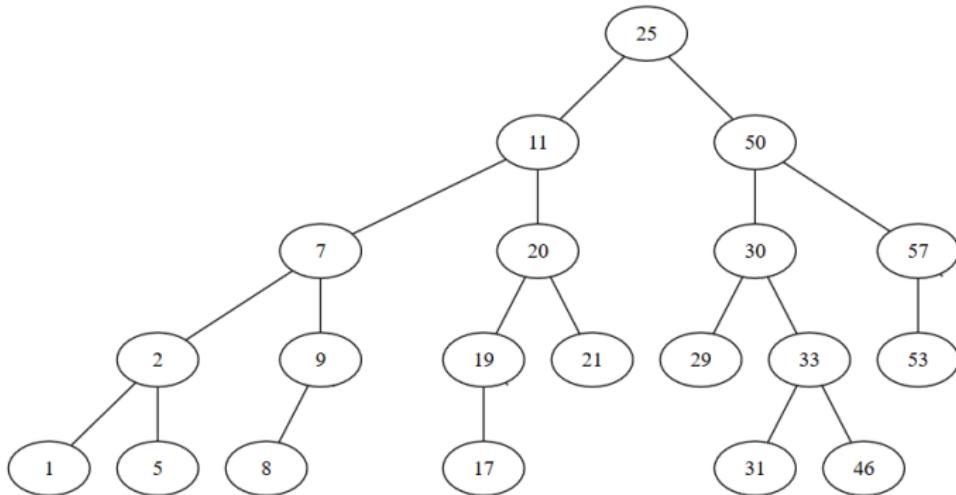
## AVL rotations example XXXIX

- Yes, we need a single left rotation on node 31
  - After the rotation



- ## ● Insert 1

# AVL rotations example XL



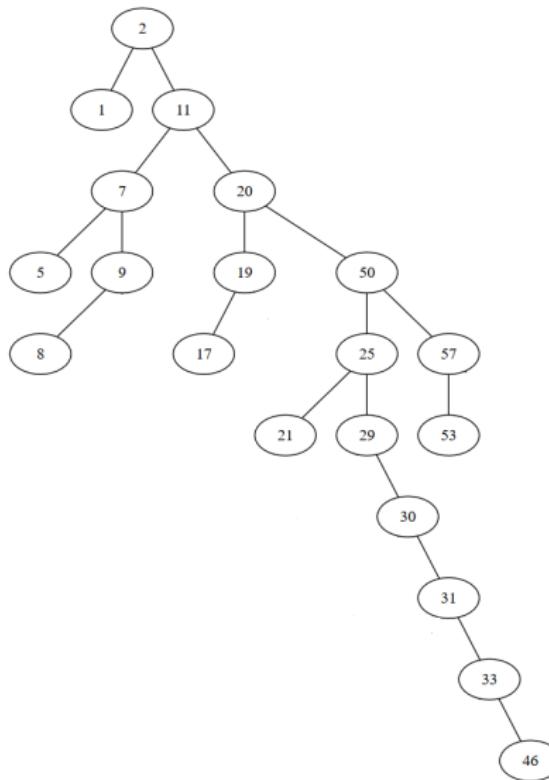
- Do we need a rotation?
- If yes, on which node and what type of rotation?

# AVL rotations example XLI

- No rotation is needed

# Comparison to BST

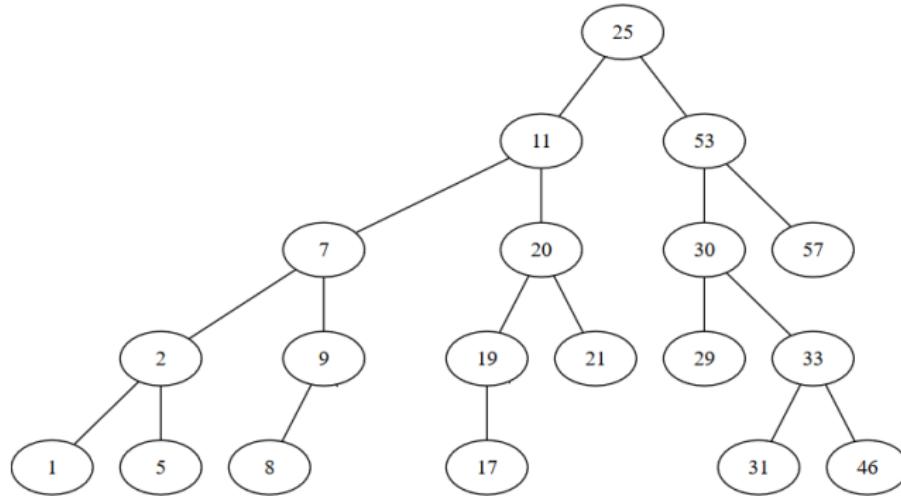
- If, instead of using an AVL tree, we used a binary search tree, after the insertions the tree would have been:



# Example of remove I

- Remove 50

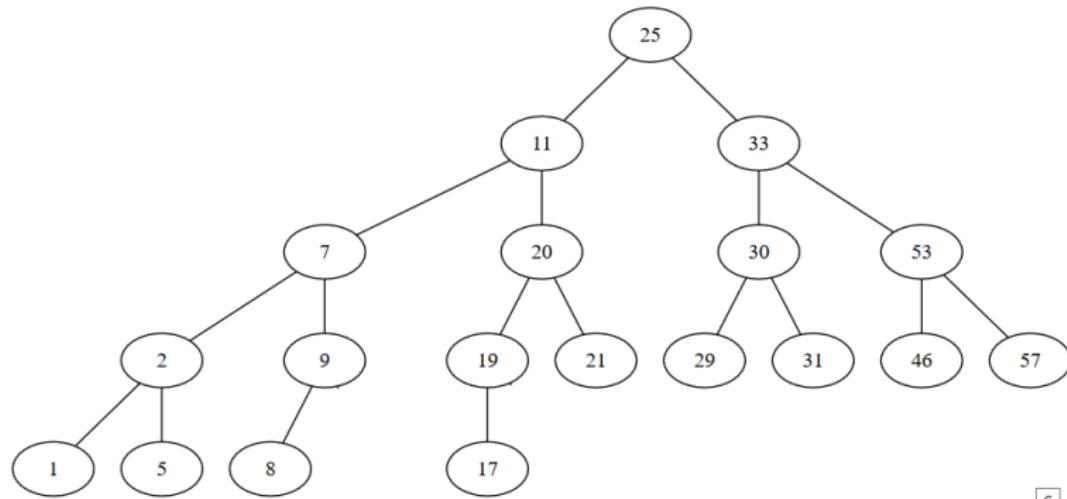
# Example of remove II



- Do we need a rotation?
- If yes, on which node and what type of rotation?

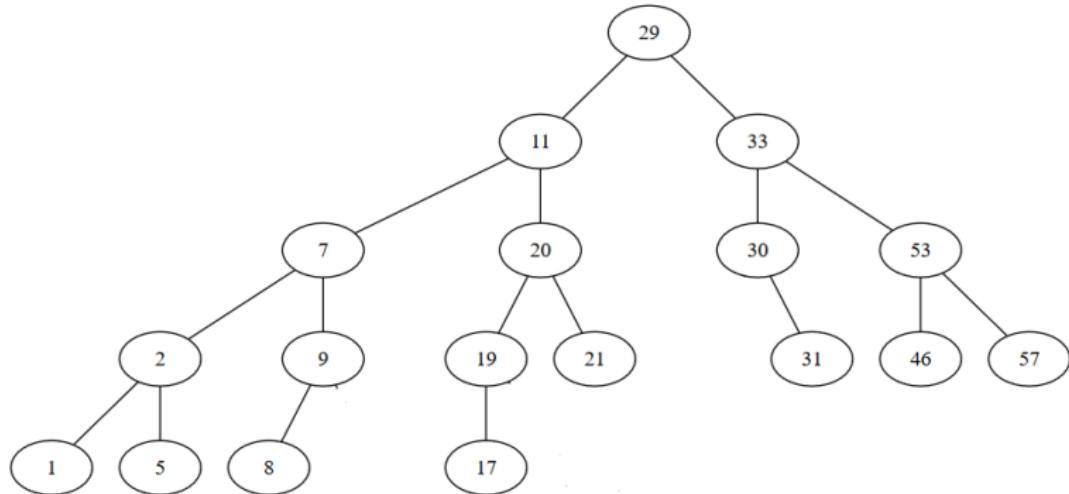
## Example of remove III

- Yes we need double right rotation on node 53
- After the rotation



- Remove 25

# Example of remove IV

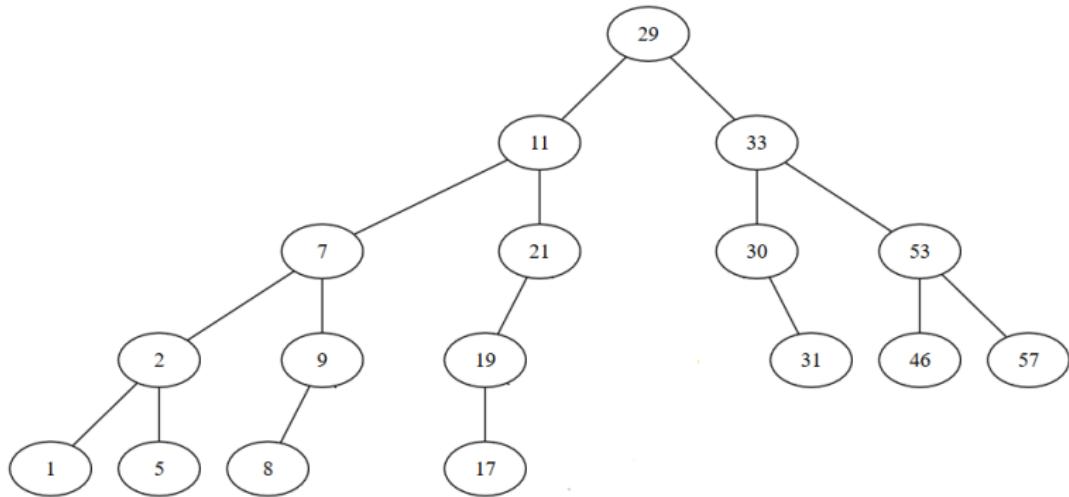


- Do we need a rotation?
- If yes, on which node and what type of rotation?

## Example of remove V

- No rotation is needed
- Remove 20

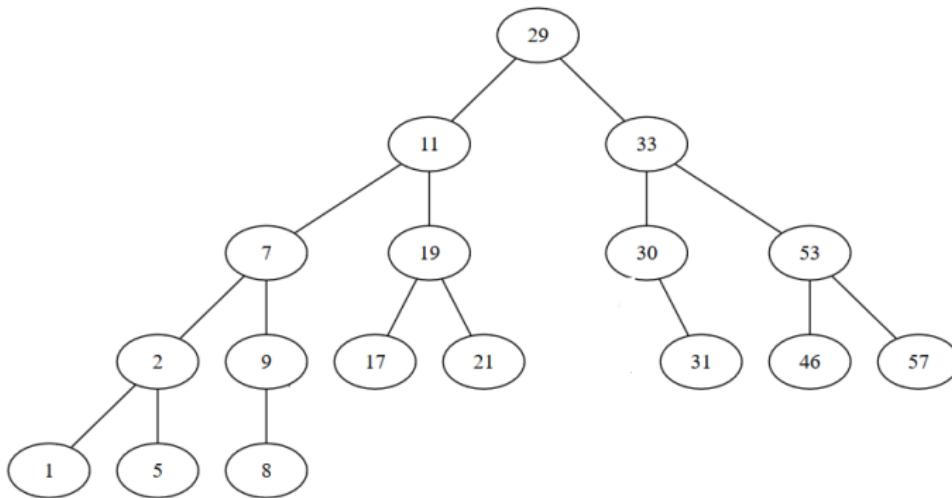
# Example of remove VI



- Do we need a rotation?
- If yes, on which node and what type of rotation?

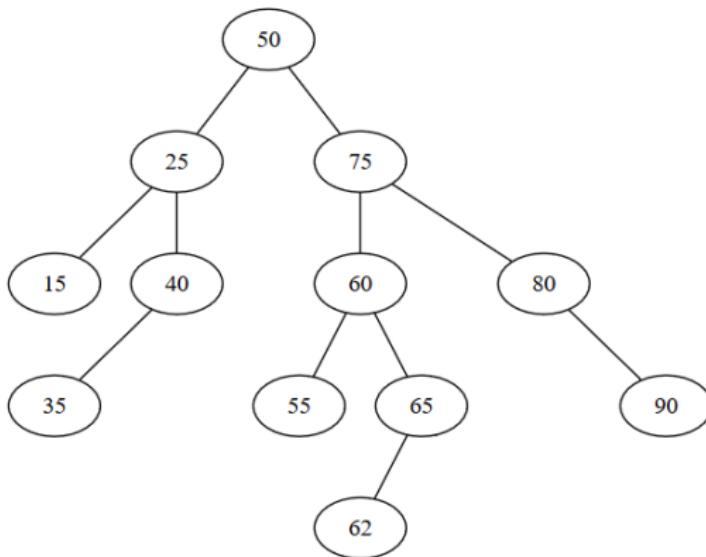
## Example of remove VII

- Yes, we need a single right rotation on node 21
- After the rotation



## Rotations for remove

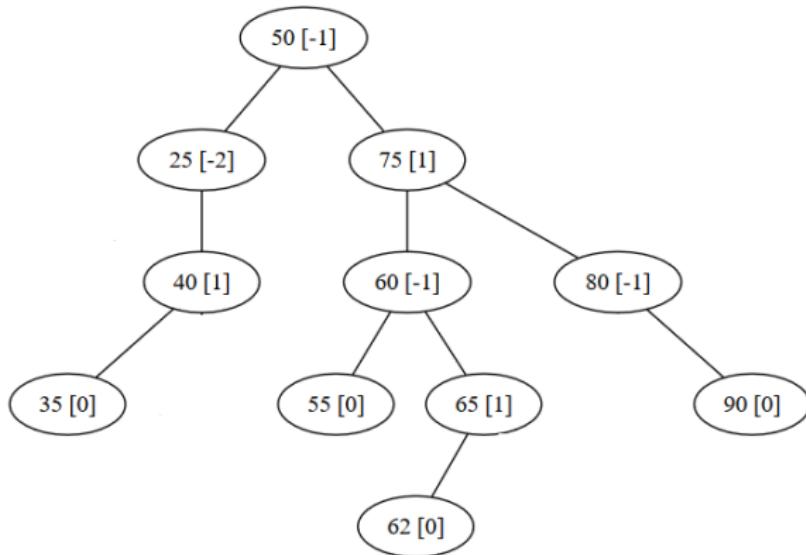
- When we remove a node, we might need more than 1 rotation:



- Remove value 15

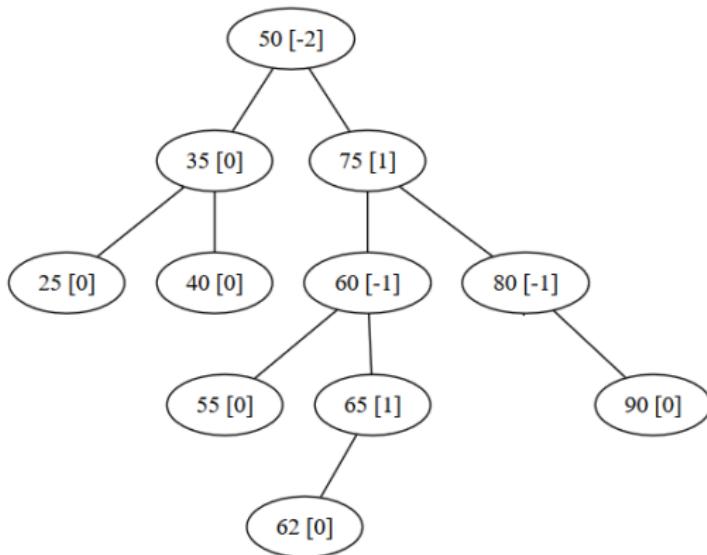
# Rotations for remove

- After remove:



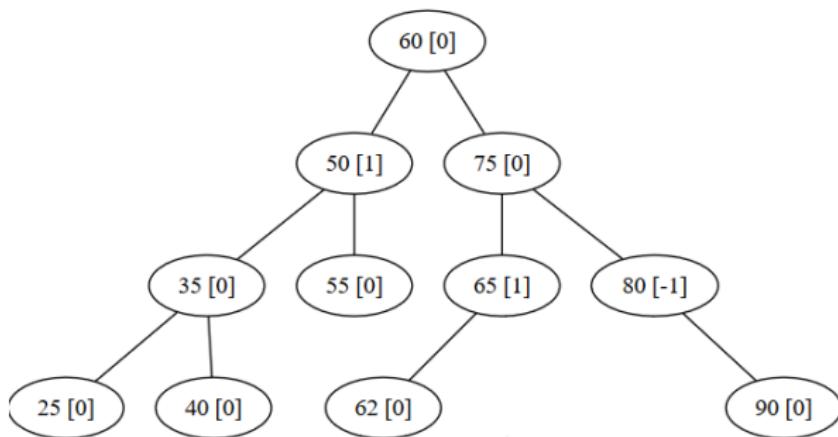
# Rotations for remove

- After the rotation



# Rotations for remove

- After the second rotation



# AVL Trees - representation

- What structures do we need for an AVL Tree?

# AVL Trees - representation

- What structures do we need for an AVL Tree?

## AVLNode:

info: TComp //*information from the node*  
left: ↑ AVLNode //*address of left child*  
right: ↑ AVLNode //*address of right child*  
h: Integer //*height of the node*

## AVLTree:

root: ↑ AVLNode //*root of the tree*

# AVL Tree - implementation

- We will implement the *insert* operation for the AVL Tree.
- We need to implement some operations to make the implementation of *insert* simpler:
  - A subalgorithm that (re)computes the height of a node
  - A subalgorithm that computes the balance factor of a node
  - Four subalgorithms for the four rotation types (we will implement only one)
- And we will assume that we have a function, *createNode* that creates and returns a node containing a given information (left and right are NIL, height is 0).

# AVL Tree - height of a node

**subalgorithm** recomputeHeight(node) **is:**

//*pre: node is an ↑ AVLNode. All descendants of node have their height (h) set  
//to the correct value  
//post: if node ≠ NIL, h of node is set*

# AVL Tree - height of a node

**subalgorithm** recomputeHeight(node) **is:**

//pre: node is an ↑ AVLNode. All descendants of node have their height (h) set

//to the correct value

//post: if node ≠ NIL, h of node is set

**if** node ≠ NIL **then**

**if** [node].left = NIL **and** [node].right = NIL **then**

        [node].h ← 0

**else if** [node].left = NIL **then**

        [node].h ← [[node].right].h + 1

**else if** [node].right = NIL **then**

        [node].h ← [[node].left].h + 1

**else**

        [node].h ← max ([[node].left].h, [[node].right].h) + 1

**end-if**

**end-if**

**end-subalgorithm**

- Complexity:  $\Theta(1)$

# AVL Tree - balance factor of a node

```
function balanceFactor(node) is:  
    //pre: node is an ↑ AVLNode. All descendants of node have their height (h) set  
    //to the correct value  
    //post: returns the balance factor of the node
```

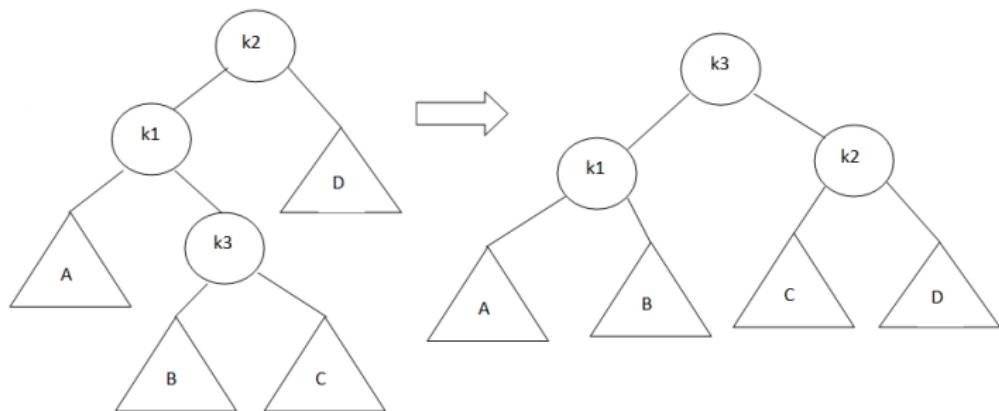
# AVL Tree - balance factor of a node

```
function balanceFactor(node) is:  
    //pre: node is an ↑ AVLNode. All descendants of node have their height (h) set  
    //to the correct value  
    //post: returns the balance factor of the node  
    if [node].left = NIL and [node].right = NIL then  
        balanceFactor ← 0  
    else if [node].left = NIL then  
        balanceFactor ← -1 - [[node].right].h //height of empty tree is -1  
    else if [node].right = NIL then  
        balanceFactor ← [[node].left].h + 1  
    else  
        balanceFactor ← [[node].left].h - [[node].right].h  
    end-if  
end-subalgorithm
```

- Complexity:  $\Theta(1)$

# AVL Tree - rotations

- Out of the four rotations, we will only implement one, double right rotation (DRR).
- The other three rotations can be implemented similarly (RLR, SRR, SLR).



# AVL Tree - DRR

```
function DRR(node) is: //pre: node is an ↑ AVLNode on which we perform  
the double right rotation  
//post: DRR returns the new root after the rotation  
k2 ← node  
k1 ← [node].left  
k3 ← [k1].right  
k3left ← [k3].left  
k3right ← [k3].right
```

# AVL Tree - DRR

```
function DRR(node) is: //pre: node is an ↑ AVLNode on which we perform  
the double right rotation  
//post: DRR returns the new root after the rotation  
k2 ← node  
k1 ← [node].left  
k3 ← [k1].right  
k3left ← [k3].left  
k3right ← [k3].right  
//reset the links  
newRoot ← k3  
[newRoot].left ← k1  
[newRoot].right ← k2  
[k1].right ← k3left  
[k2].left ← k3right  
//continued on the next slide
```

```
//recompute the heights of the modified nodes
    recomputeHeight(k1)
    recomputeHeight(k2)
    recomputeHeight(newRoot)
    DRR ← newRoot
end-function
```

- Complexity:  $\Theta(1)$

# AVL Tree - insert

```
function insertRec(node, elem) is
    //pre: node is a ↑ AVLNode, elem is the value we insert in the
    //      (sub)tree that
    //has node as root
    //post: insertRec returns the new root of the (sub)tree after the
    //      insertion
    if node = NIL then
        insertRec ← createNode(elem)
    else if elem ≤ [node].info then
        [node].left ← insertRec([node].left, elem)
    else
        [node].right ← insertRec([node].right, elem)
    end-if
    //continued on the next slide...
```

# AVL Tree - insert

```
recomputeHeight(node)
balance ← getBalanceFactor(node)
if balance = -2 then
```

# AVL Tree - insert

```
recomputeHeight(node)
balance ← getBalanceFactor(node)
if balance = -2 then
    //right subtree has larger height, we will need a rotation to the LEFT
    rightBalance ← getBalanceFactor([node].right)
    if rightBalance < 0 then
```

# AVL Tree - insert

```
recomputeHeight(node)
balance ← getBalanceFactor(node)
if balance = -2 then
    //right subtree has larger height, we will need a rotation to the LEFT
    rightBalance ← getBalanceFactor([node].right)
    if rightBalance < 0 then
        //the right subtree of the right subtree has larger height, SRL
        node ← SRL(node)
    else
        node ← DRL(node)
    end-if
//continued on the next slide...
```

# AVL Tree - insert

```
else if balance = 2 then
    //left subtree has larger height, we will need a RIGHT rotation
    leftBalance ← getBalanceFactor([node].left)
    if leftBalance > 0 then
```

# AVL Tree - insert

```
else if balance = 2 then
    //left subtree has larger height, we will need a RIGHT rotation
    leftBalance ← getBalanceFactor([node].left)
    if leftBalance > 0 then
        //the left subtree of the left subtree has larger height, SRR
        node ← SRR(node)
    else
        node ← DRR(node)
    end-if
end-if
insertRec ← node
end-function
```

- Complexity of the *insertRec* algorithm:  $O(\log_2 n)$
- Since *insertRec* receives as parameter a pointer to a node, we need a wrapper function to do the first call on the root

```
subalgorithm insert(tree, elem) is
    //pre: tree is an AVL Tree, elem is the element to be inserted
    //post: elem was inserted to tree
    tree.root ← insertRec(tree.root, elem)
end-subalgorithm
```

- remove subalgorithm can be implemented similarly (start from the remove from BST and add the rotation part).

# Huffman coding

- The *Huffman coding* can be used to encode characters (from an alphabet) using variable length codes.
- In order to reduce the total number of bits needed to encode a message, characters that appear more frequently have shorter codes.
- Since we use variable length code for each character, *no code can be the prefix of any other code* (if we encode letter E with 01 and letter X with 010011, during decoding, when we find a 01, we will not know whether it is E or the beginning of X).

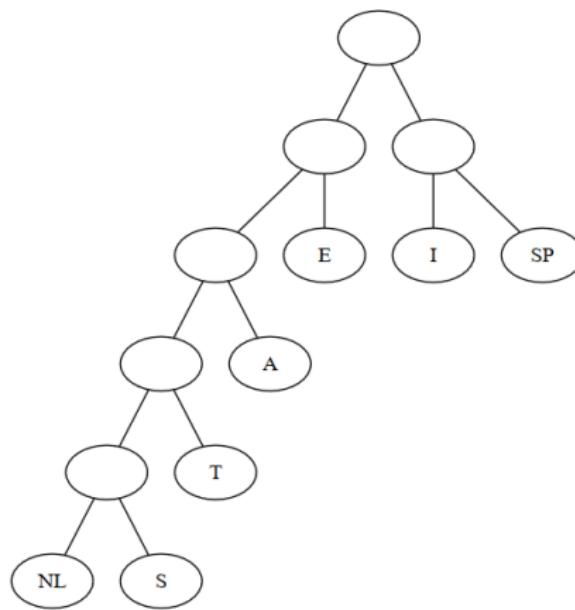
# Huffman coding

- When building the Huffman encoding for a message, we first have to compute the frequency of every character from the message, because we are going to define the codes based on the frequencies.
- Assume that we have a message with the following letters and frequencies

Character	a	e	i	s	t	space	newline
Frequency	10	15	12	3	4	13	1

- For defining the Huffman code a binary tree is build in the following way:
  - Start with trees containing only a root node, one for every character. Each tree has a weight, which is frequency of the character.
  - Get the two trees with the least weight (if there is a tie, choose randomly), combine them into one tree which has as weight the sum of the two weights.
  - Repeat until we have only one tree.

# Huffman coding



# Huffman coding

- Code for each character can be read from the tree in the following way: start from the root and go towards the corresponding leaf node. Every time we go left add the bit 0 to encoding and when we go right add bit 1.
- Code for the characters:
  - NL - 00000
  - S - 00001
  - T - 0001
  - A - 001
  - E - 01
  - I - 10
  - SP - 11
- In order to encode a message, just replace each character with the corresponding code

# Huffman coding

- Assume we have the following code and we want to decode it:  
011011000100010011100100000
- We do not know where the code of each character ends, but we can use the previously built tree to decode it.
- Start parsing the code and iterate through the tree in the following way:
  - Start from the root
  - If the current bit from the code is 0 go to the left child, otherwise go to the right child
  - If we are at a leaf node we have decoded a character and have to start over from the root
- The decoded message: E I SP T T A SP I E NL

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 12

Lect. PhD. Onet-Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

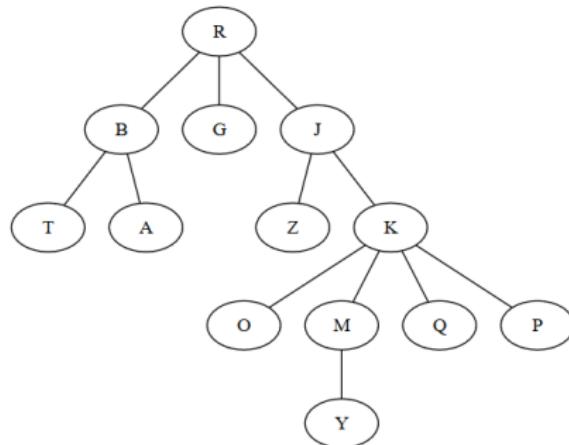
2020 - 2021

# In Lecture 11...

- Hash tables - open addressing
- Trees

- Trees
- Binary Trees
- Binary Search Trees

# Tree - Example

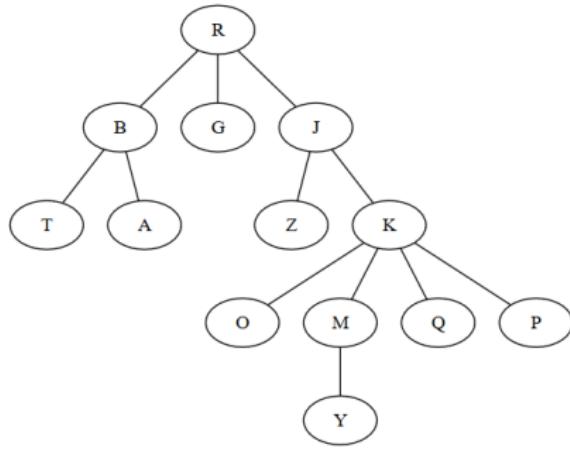


- A node of a tree is said to be *visited* when the program control arrives at the node, usually with the purpose of performing some operation on the node (printing it, checking the value from the node, etc.).
- *Traversing* a tree means visiting all of its nodes.
- For a k-ary tree there are 2 possible traversals:
  - Depth-first traversal
  - Level order (breadth first) traversal

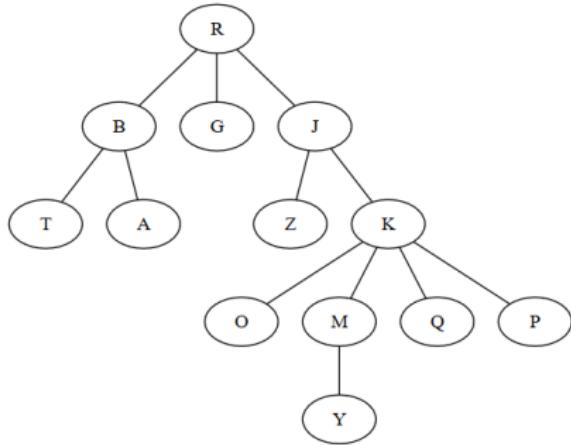
# Depth first traversal

- Traversal starts from root
- From root we visit one of the children, than one child of that child, and so on. We go down (in depth) as much as possible, and continue with other children of a node only after all descendants of the "first" child were visited.
- For depth first traversal we use a stack to remember the nodes that have to be visited.

# Depth first traversal example



# Depth first traversal example

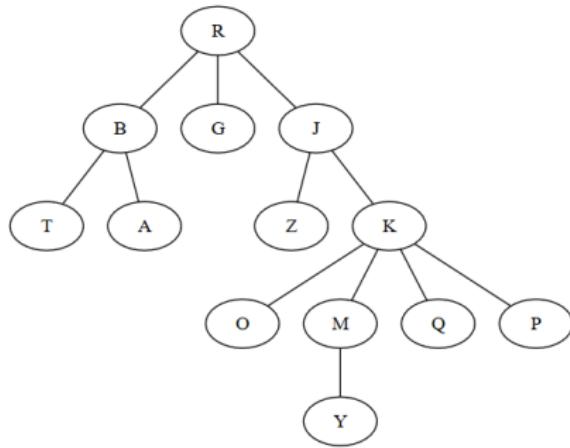


- Stack  $s$  with the root:  $R$
- Visit  $R$  (pop from stack) and push its children:  $s = [B \ G \ J]$
- Visit  $B$  and push its children:  $s = [T \ A \ G \ J]$
- Visit  $T$  and push nothing:  $s = [A \ G \ J]$
- Visit  $A$  and push nothing:  $s = [G \ J]$
- Visit  $G$  and push nothing:  $s = [J]$
- Visit  $J$  and push its children:  $s = [Z \ K]$
- etc...

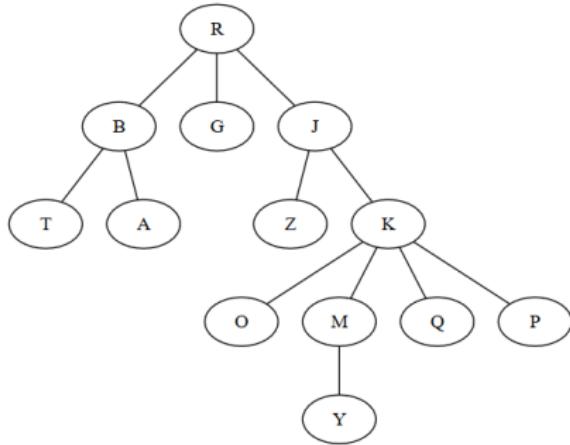
# Level order traversal

- Traversal starts from root
- We visit all children of the root (one by one) and once all of them were visited we go to their children and so on. We go down one level, only when all nodes from a level were visited.
- For level order traversal we use a queue to remember the nodes that have to be visited.

# Level order traversal example



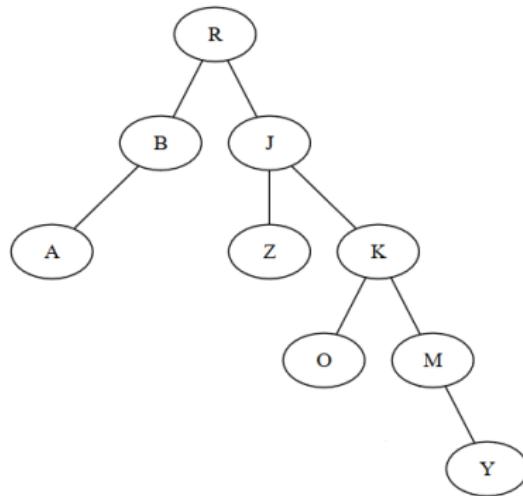
# Level order traversal example



- Queue  $q$  with the root:  $R$
- Visit  $R$  (pop from queue) and push its children:  $q = [B \ G \ J]$
- Visit  $B$  and push its children:  $q = [G \ J \ T \ A]$
- Visit  $G$  and push nothing:  $q = [J \ T \ A]$
- Visit  $J$  and push its children:  $q = [T \ A \ Z \ K]$
- Visit  $T$  and push nothing:  $q = [A \ Z \ K]$
- Visit  $A$  and push nothing:  $q = [Z \ K]$
- etc...

- An ordered tree in which each node has at most two children is called *binary tree*.
- In a binary tree we call the children of a node the *left child* and *right child*.
- Even if a node has only one child, we still have to know whether that is the left or the right one.

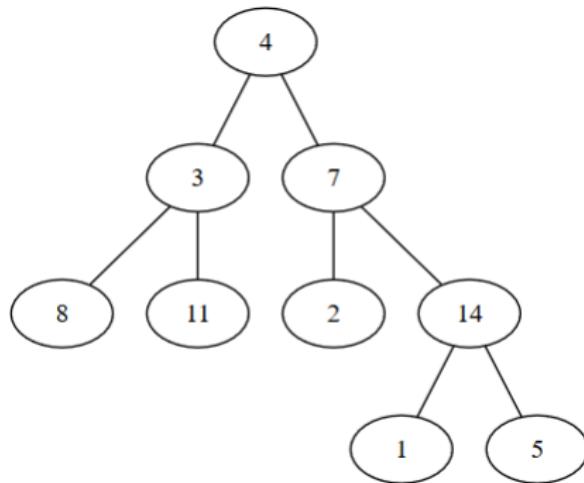
# Binary tree - example



- $A$  is the left child of  $B$
- $Y$  is the right child of  $M$

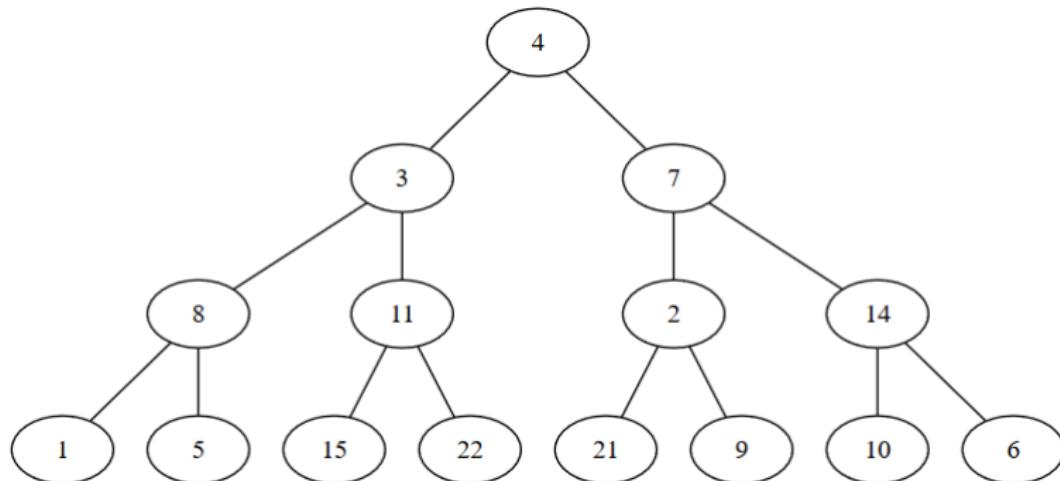
# Binary tree - Terminology I

- A binary tree is called *full* if every internal node has exactly two children.



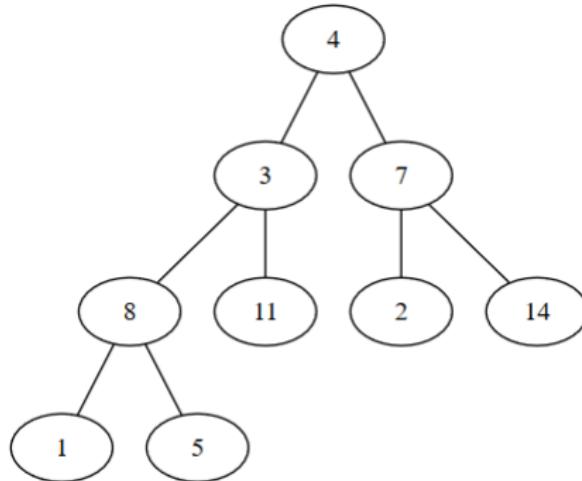
## Binary tree - Terminology II

- A binary tree is called *complete* if all leaves are one the same level and all internal nodes have exactly 2 children.



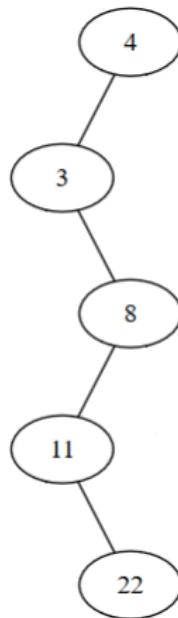
# Binary tree - Terminology III

- A binary tree is called *almost complete* if it is a *complete* binary tree except for the last level, where nodes are completed from left to right (binary heap - structure).



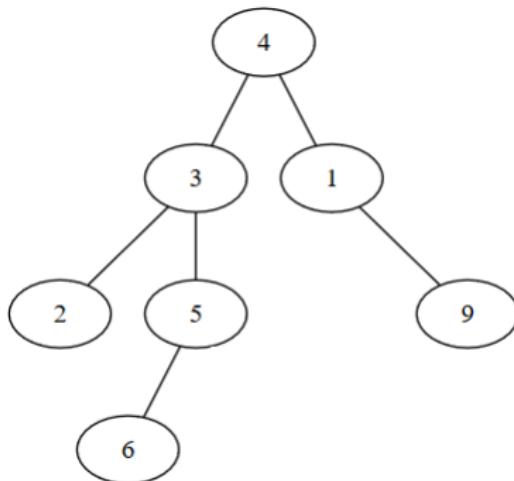
## Binary tree - Terminology IV

- A binary tree is called *degenerate* if every internal node has exactly one child (it is actually a chain of nodes).



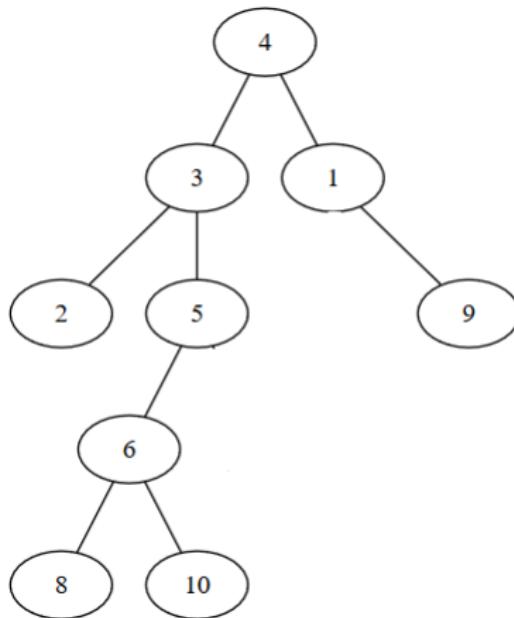
# Binary tree - Terminology V

- A binary tree is called *balanced* if the difference between the height of the left and right subtrees is at most 1 (for every node from the tree).



## Binary tree - Terminology VI

- Obviously, there are many binary trees that are none of the above categories, for example:



# Binary tree - properties

- A binary tree with  $n$  nodes has exactly  $n - 1$  edges (this is true for every tree, not just binary trees)
- The number of nodes in a complete binary tree of height  $N$  is  $2^{N+1} - 1$  (it is  $1 + 2 + 4 + 8 + \dots + 2^N$  )
- The maximum number of nodes in a binary tree of height  $N$  is  $2^{N+1} - 1$  - if the tree is complete.
- The minimum number of nodes in a binary tree of height  $N$  is  $N + 1$  - if the tree is degenerate.
- A binary tree with  $N$  nodes has a height between  $\log_2 N$  and  $N - 1$ .

# ADT Binary Tree I

- Domain of ADT Binary Tree:

$$\mathcal{BT} = \{bt \mid bt \text{ binary tree with nodes containing information of type } \text{TElem}\}$$

- **init( $bt$ )**

- **descr:** creates a new, empty binary tree
- **pre:** true
- **post:**  $bt \in BT$ ,  $bt$  is an empty binary tree

- $\text{initLeaf}(bt, e)$ 
  - **descr:** creates a new binary tree, having only the root with a given value
  - **pre:**  $e \in TElem$
  - **post:**  $bt \in BT$ ,  $bt$  is a binary tree with only one node (its root) which contains the value  $e$

- $\text{initTree}(\text{bt}, \text{left}, \text{e}, \text{right})$ 
  - **descr:** creates a new binary tree, having a given information in the root and two given binary trees as children
  - **pre:**  $\text{left}, \text{right} \in \mathcal{BT}$ ,  $\text{e} \in T\text{Elem}$
  - **post:**  $\text{bt} \in \mathcal{BT}$ ,  $\text{bt}$  is a binary tree with left child equal to  $\text{left}$ , right child equal to  $\text{right}$  and the information from the root is  $\text{e}$

- `insertLeftSubtree(bt, left)`
  - **descr:** sets the left subtree of a binary tree to a given value (if the tree had a left subtree, it will be changed)
  - **pre:**  $bt, left \in \mathcal{BT}$
  - **post:**  $bt' \in \mathcal{BT}$ , the left subtree of  $bt'$  is equal to  $left$

- `insertRightSubtree(bt, right)`
  - **descr:** sets the right subtree of a binary tree to a given value  
(if the tree had a right subtree, it will be changed)
  - **pre:**  $bt, right \in \mathcal{BT}$
  - **post:**  $bt' \in \mathcal{BT}$ , the right subtree of  $bt'$  is equal to  $right$

- **root(bt)**

- **descr:** returns the information from the root of a binary tree
- **pre:**  $bt \in BT$ ,  $bt \neq \Phi$
- **post:**  $root \leftarrow e$ ,  $e \in TElm$ ,  $e$  is the information from the root of  $bt$
- **throws:** an exception if  $bt$  is empty

- **left(*bt*)**
  - **descr:** returns the left subtree of a binary tree
  - **pre:**  $bt \in \mathcal{BT}$ ,  $bt \neq \Phi$
  - **post:**  $left \leftarrow I$ ,  $I \in \mathcal{BT}$ ,  $I$  is the left subtree of  $bt$
  - **throws:** an exception if  $bt$  is empty

- **right( $bt$ )**
  - **descr:** returns the right subtree of a binary tree
  - **pre:**  $bt \in \mathcal{BT}$ ,  $bt \neq \Phi$
  - **post:**  $right \leftarrow r$ ,  $r \in \mathcal{BT}$ ,  $r$  is the right subtree of  $bt$
  - **throws:** an exception if  $bt$  is empty

- **isEmpty( $bt$ )**

- **descr:** checks if a binary tree is empty
- **pre:**  $bt \in BT$
- **post:**

$$empty \leftarrow \begin{cases} True, & \text{if } bt = \Phi \\ False, & \text{otherwise} \end{cases}$$

- iterator ( $bt$ , traversal,  $i$ )
  - **descr:** returns an iterator for a binary tree
  - **pre:**  $bt \in BT$ ,  $traversal$  represents the order in which the tree has to be traversed
  - **post:**  $i \in \mathcal{I}$ ,  $i$  is an iterator over  $bt$  that iterates in the order given by  $traversal$

- **destroy(bt)**
  - **descr:** destroys a binary tree
  - **pre:**  $bt \in BT$
  - **post:**  $bt$  was destroyed

- Other possible operations:
  - change the information from the root of a binary tree
  - remove a subtree (left or right) of a binary tree
  - search for an element in a binary tree
  - return the number of elements from a binary tree

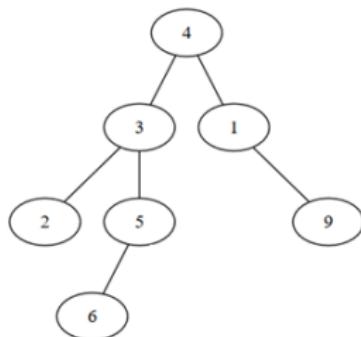
# Possible representations

- If we want to implement a binary tree, what representation can we use?
- We have several options:
  - Representation using an array (similar to a binary heap)
  - Linked representation
    - with dynamic allocation
    - on an array

# Possible representations I

- Representation using an array
  - Store the elements in an array
  - First position from the array is the root of the tree
  - Left child of node from position  $i$  is at position  $2 * i$ , right child is at position  $2 * i + 1$ .
  - Some special value is needed to denote the place where no element is.

## Possible representations II



Pos	Elem
1	4
2	3
3	1
4	2
5	5
6	-1
7	9
8	-1
9	-1
10	6
11	-1
12	-1
13	-1
...	...

- Disadvantage: depending on the form of the tree, we might waste a lot of space.

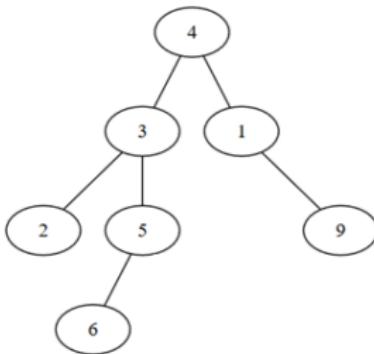
# Possible representations I

- Linked representation with dynamic allocation
  - We have a structure to represent a node, containing the information, the address of the left child and the address of the right child (possibly the address of the parent as well).
  - An empty tree is denoted by the value NIL for the root.
  - We have one node for every element of the tree.

## Possible representations II

- Linked representation on an array
  - Information from the nodes is placed in an array. The *address* of the left and right child is the *index* where the corresponding elements can be found in the array.
  - We can have a separate array for the parent as well.

# Possible representations III



Pos	1	2	3	4	5	6	7	8
Info	4	3	2	5	6	1	9	
Left	2	3	-1	5	-1	-1	-1	
Right	6	4	-1	-1	-1	7	-1	
Parent	-1	1	2	2	4	1	6	

- We need to know that the root is at position 1 (could be any position).
- If the array is full, we can allocate a larger one.
- We have to keep a linked list of empty positions to make adding a new node easier.

## Possible representations IV

- The linked list of empty positions has to be created when the empty binary tree is created. While a tree is a non-linear data structure, we can still use the left (and/or right) array to create a singly (or doubly) linked list of empty positions.
- Obviously, when we do a resize, the newly created empty positions have to be linked again.

info								
left	2	3	4	5	6	7	8	-1
right								
firstEmpty = 1								
root = -1								
cap = 8								

# Binary Tree Traversal

- A node of a (binary) tree is visited when the program control arrives at the node, usually with the purpose of performing some operation on the node (printing it, checking the value from the node, etc.).
- *Traversing* a (binary) tree means visiting all of its nodes.
- For a binary tree there are 4 possible traversals:
  - Preorder
  - Inorder
  - Postorder
  - Level order (breadth first) - the same as in case of a (non-binary) tree

# Binary tree representation

- In the following, for the implementation of the traversal algorithms, we are going to use the following representation for a binary tree:

BTNode:

info: TElem

left:  $\uparrow$  BTNode

right:  $\uparrow$  BTNode

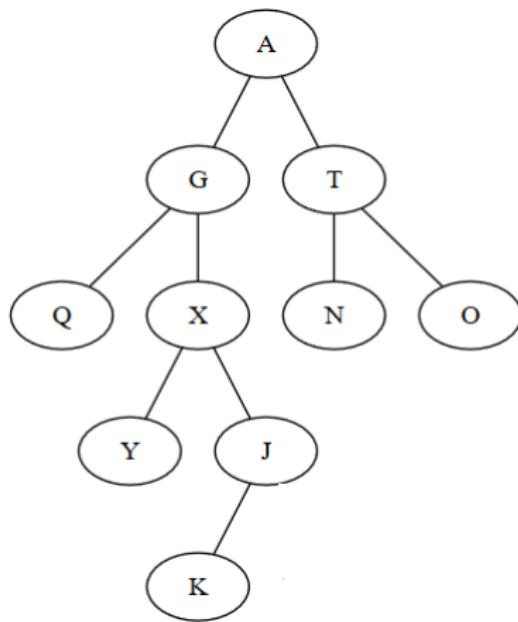
BinaryTree:

root:  $\uparrow$  BTNode

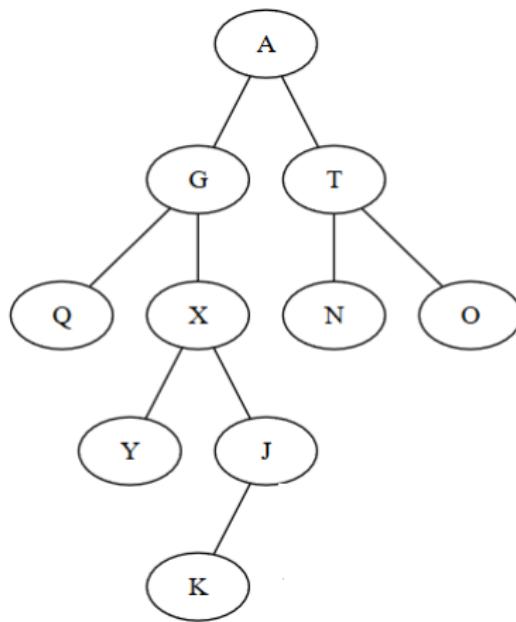
# Preorder traversal

- In case of a preorder traversal:
  - Visit the *root* of the tree
  - Traverse the left subtree - if exists
  - Traverse the right subtree - if exists
- When traversing the subtrees (left or right) the same preorder traversal is applied (so, from the left subtree we visit the root first and then traverse the left subtree and then the right subtree).

# Preorder traversal example



# Preorder traversal example



- Preorder traversal: A, G, Q, X, Y, J, K, T, N, O

# Preorder traversal - recursive implementation

- The simplest implementation for preorder traversal is with a recursive algorithm.

**subalgorithm** preorder\_recursive(*node*) **is:**

```
//pre: node is a ↑ BTNode
if node ≠ NIL then
    @visit [node].info
    preorder_recursive([node].left)
    preorder_recursive([node].right)
end-if
end-subalgorithm
```

# Preorder traversal - recursive implementation

- The *preorder\_recursive* subalgorithm receives as parameter a pointer to a node, so we need a wrapper subalgorithm, one that receives a *BinaryTree* and calls the function for the root of the tree.

**subalgorithm** preorderRec(tree) **is:**

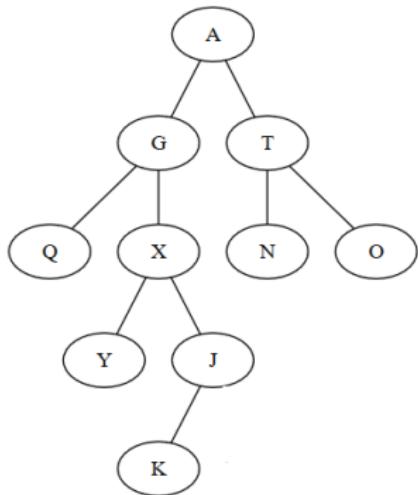
```
//pre: tree is a BinaryTree  
    preorder_recursive(tree.root)  
end-subalgorithm
```

- Assuming that visiting a node takes constant time (print the info from the node, for example), the whole traversal takes  $\Theta(n)$  time for a tree with  $n$  nodes.

# Preorder traversal - non-recursive implementation

- We can implement the preorder traversal algorithm without recursion, using an auxiliary *stack* to store the nodes.
  - We start with an empty stack
  - Push the root of the tree to the stack
  - While the stack is not empty:
    - Pop a node and visit it
    - Push the node's right child to the stack
    - Push the node's left child to the stack

# Preorder traversal - non-recursive implementation example



- Stack: A
- Visit A, push children (Stack: T G)
- Visit G, push children (Stack: T X Q)
- Visit Q, push nothing (Stack: T X)
- Visit X, push children (Stack: T J Y)
- Visit Y, push nothing (Stack: T J)
- Visit J, push child (Stack: T K)
- Visit K, push nothing (Stack: T)
- Visit T, push children (Stack: O N)
- Visit N, push nothing (Stack: O)
- Visit O, push nothing (Stack: )
- Stack is empty, traversal is complete

# Preorder traversal - non-recursive implementation

```
subalgorithm preorder(tree) is:
//pre: tree is a binary tree
s: Stack //s is an auxiliary stack
if tree.root ≠ NIL then
    push(s, tree.root)
end-if
while not isEmpty(s) execute
    currentNode ← pop(s)
    @visit currentNode
    if [currentNode].right ≠ NIL then
        push(s, [currentNode].right)
    end-if
    if [currentNode].left ≠ NIL then
        push(s, [currentNode].left)
    end-if
end-while
end-subalgorithm
```

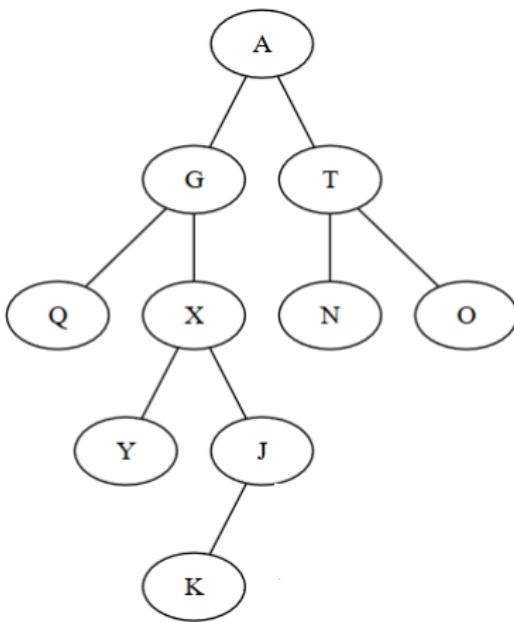
# Preorder traversal - non - recursive implementation

- Time complexity of the non-recursive traversal is  $\Theta(n)$ , and we also need  $O(n)$  extra space (the stack)
- Obs: Preorder traversal is exactly the same as *depth first traversal* (you can see it especially in the implementation), with the observation that here we need to be careful to first push the right child to the stack and then the left one (in case of *depth-first traversal* the order in which we pushed the children was not that important).

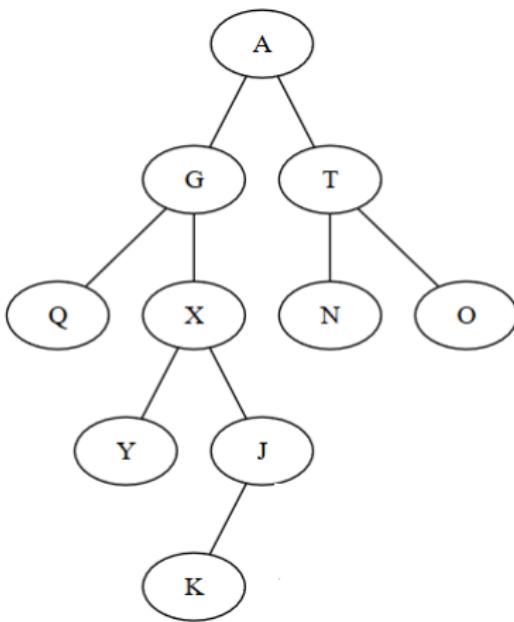
# Inorder traversal

- In case of *inorder* traversal:
  - Traverse the left subtree - if exists
  - Visit the *root* of the tree
  - Traverse the right subtree - if exists
- When traversing the subtrees (left or right) the same inorder traversal is applied (so, from the left subtree we traverse the left subtree, then we visit the root and then traverse the right subtree).

# Inorder traversal example



# Inorder traversal example



- Inorder traversal: Q, G, Y, X, K, J, A, N, T, O

# Inorder traversal - recursive implementation

- The simplest implementation for inorder traversal is with a recursive algorithm.

**subalgorithm** inorder\_recursive(*node*) **is:**

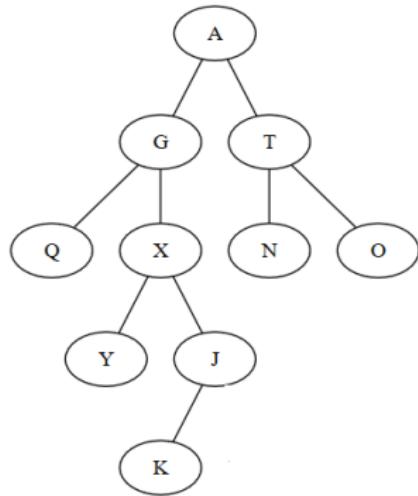
```
//pre: node is a ↑ BTNode
if node ≠ NIL then
    inorder_recursive([node].left)
    @visit [node].info
    inorder_recursive([node].right)
end-if
end-subalgorithm
```

- We need again a wrapper subalgorithm to perform the first call to *inorder\_recursive* with the root of the tree as parameter.
- The traversal takes  $\Theta(n)$  time for a tree with  $n$  nodes.

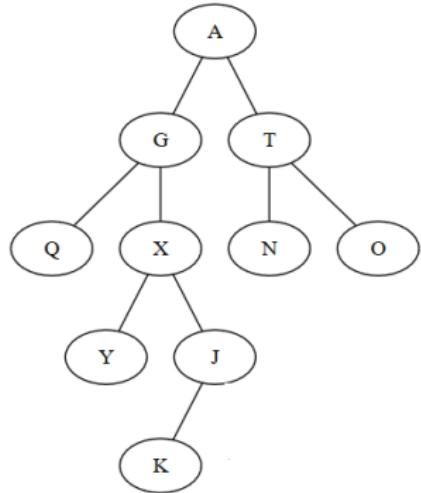
# Inorder traversal - non-recursive implementation

- We can implement the inorder traversal algorithm without recursion, using an auxiliary stack to store the nodes.
  - We start with an empty stack and a current node set to the root
  - While current node is not NIL, push it to the stack and set it to its left child
  - While stack not empty
    - Pop a node and visit it
    - Set current node to the right child of the popped node
    - While current node is not NIL, push it to the stack and set it to its left child

# Inorder traversal - non-recursive implementation example



# Inorder traversal - non-recursive implementation example



- CurrentNode: A (Stack: )
- CurrentNode: NIL (Stack: A G Q)
- Visit Q, currentNode NIL (Stack: A G)
- Visit G, currentNode X (Stack: A)
- CurrentNode: NIL (Stack: A X Y)
- Visit Y, currentNode NIL (Stack: A X)
- Visit X, currentNode J (Stack: A)
- CurrentNode: NIL (Stack: A J K)
- Visit K, currentNode NIL (Stack: A J)
- Visit J, currentNode NIL (Stack: A)
- Visit A, currentNode T (Stack: )
- CurrentNode: NIL (Stack: T N)
- ...

# Inorder traversal - non-recursive implementation

```
subalgorithm inorder(tree) is:
//pre: tree is a Binary Tree
    s: Stack //s is an auxiliary stack
    currentNode ← tree.root
    while currentNode ≠ NIL execute
        push(s, currentNode)
        currentNode ← [currentNode].left
    end-while
    while not isEmpty(s) execute
        currentNode ← pop(s)
        @visit currentNode
        currentNode ← [currentNode].right
        while currentNode ≠ NIL execute
            push(s, currentNode)
            currentNode ← [currentNode].left
        end-while
    end-while
end-subalgorithm
```

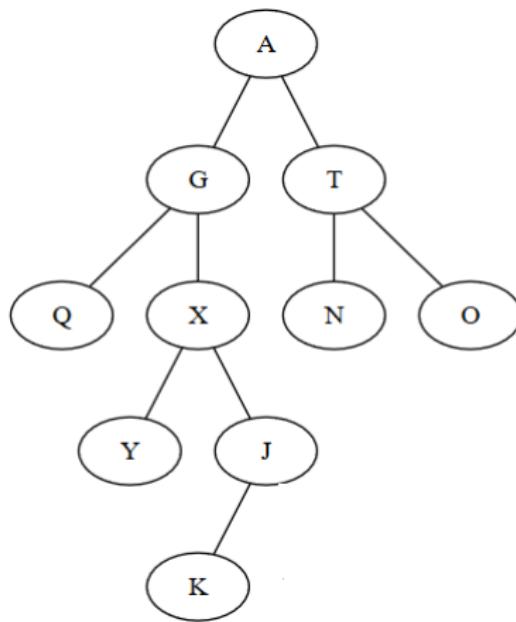
# Inorder traversal - non-recursive implementation

- Time complexity  $\Theta(n)$ , extra space complexity  $O(n)$

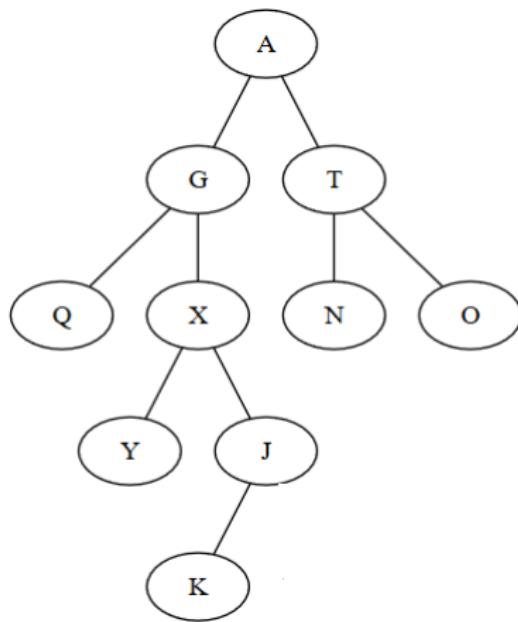
# Postorder traversal

- In case of *postorder* traversal:
  - Traverse the left subtree - if exists
  - Traverse the right subtree - if exists
  - Visit the *root* of the tree
- When traversing the subtrees (left or right) the same postorder traversal is applied (so, from the left subtree we traverse the left subtree, then traverse the right subtree and then visit the root).

# Postorder traversal example



# Postorder traversal example



- Postorder traversal: Q, Y, K, J, X, G, N, O, T, A

# Postorder traversal - recursive implementation

- The simplest implementation for postorder traversal is with a recursive algorithm.

**subalgorithm** postorder\_recursive(*node*) **is:**

```
//pre: node is a ↑ BTNode
if node ≠ NIL then
    postorder_recursive([node].left)
    postorder_recursive([node].right)
    @visit [node].info
end-if
end-subalgorithm
```

- We need again a wrapper subalgorithm to perform the first call to *postorder\_recursive* with the root of the tree as parameter.
- The traversal takes  $\Theta(n)$  time for a tree with  $n$  nodes.

## Postorder traversal - non-recursive implementation

- We can implement the postorder traversal without recursion, but it is slightly more complicated than preorder and inorder traversals.
- We can have an implementation that uses two stacks and there is also an implementation that uses one stack.

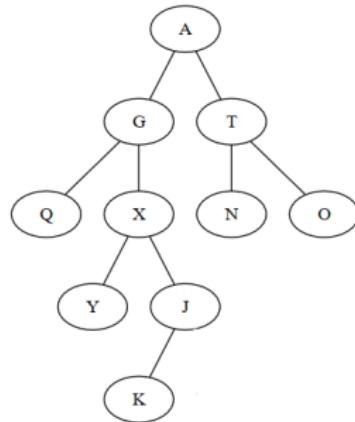
## Postorder traversal with two stacks

- The main idea of postorder traversal with two stacks is to build the reverse of postorder traversal in one stack. If we have this, popping the elements from the stack until it becomes empty will give us postorder traversal.
- Building the reverse of postorder traversal is similar to building preorder traversal, except that we need to traverse the right subtree first (not the left one). The other stack will be used for this.
- The algorithm is similar to *preorder* traversal, with two modifications:
  - When a node is removed from the stack, it is added to the second stack (instead of being visited)
  - For a node taken from the stack we first push the left child and then the right child to the stack.

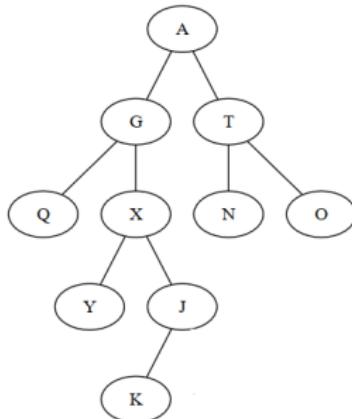
## Postorder traversal with one stack

- We start with an empty stack and a current node set to the root of the tree
- While the current node is not NIL, push to the stack the right child of the current node (if exists) and the current node and then set the current node to its left child.
- While the stack is not empty
  - Pop a node from the stack (call it current node)
  - If the current node has a right child, the stack is not empty and contains the right child on top of it, pop the right child, push the current node, and set current node to the right child.
  - Otherwise, visit the current node and set it to NIL
  - While the current node is not NIL, push to the stack the right child of the current node (if exists) and the current node and then set the current node to its left child.

# Postorder traversal - non-recursive implementation example



# Postorder traversal - non-recursive implementation example



- Node: A (Stack: )
- Node: NIL (Stack: T A X G Q)
- Visit Q, Node NIL (Stack: T A X G)
- Node: X (Stack: T A G)
- Node: NIL (Stack: T A G J X Y)
- Visit Y, Node: NIL (Stack: T A G J X)
- Node: J (Stack: T A G X)
- Node: NIL (Stack: T A G X J K)
- Visit K, Node: NIL (Stack: T A G X J)
- Visit J, Node: NIL (Stack: T A G X)
- Visit X, Node: NIL (Stack: T A G)
- Visit G, Node: NIL (Stack: T A)
- Node: T (Stack: A)
- Node: NIL (Stack: A O T N)
- ...

# Postorder traversal - non-recursive implementation

```
subalgorithm postorder(tree) is:
//pre: tree is a Binary Tree
s: Stack //s is an auxiliary stack
node  $\leftarrow$  tree.root
while node  $\neq$  NIL execute
    if [node].right  $\neq$  NIL then
        push(s, [node].right)
    end-if
    push(s, node)
    node  $\leftarrow$  [node].left
end-while
while not isEmpty(s) execute
    node  $\leftarrow$  pop(s)
    if [node].right  $\neq$  NIL and (not isEmpty(s)) and [node].right = top(s) th
        pop(s)
    push(s, node)
    node  $\leftarrow$  [node].right
//continued on the next slide
```

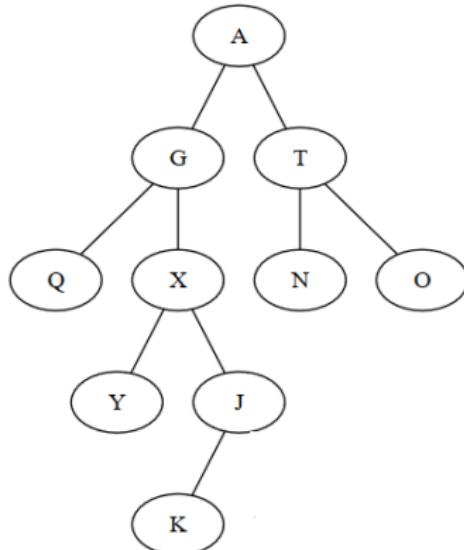
# Postorder traversal - non-recursive implementation

```
else
    @visit node
    node ← NIL
end-if
while node ≠ NIL execute
    if [node].right ≠ NIL then
        push(s, [node].right)
    end-if
    push(s, node)
    node ← [node].left
end-while
end-while
end-subalgorithm
```

- Time complexity  $\Theta(n)$ , extra space complexity  $O(n)$

# Level order traversal

- In case of level order traversal we first visit the root, then the children of the root, then the children of the children, etc.



- Level order traversal: A, G, T, Q, X, N, O, Y, J, K

# Binary tree iterator

- The interface of the binary tree contains the *iterator* operation, which should return an iterator.
- This operation receives a parameter that specifies what kind of traversal we want to do with the iterator (preorder, inorder, postorder, level order)
- The traversal algorithms discussed so far, traverse all the elements of the binary tree at once, but an iterator has to do element-by-element traversal.
- For defining an iterator, we have to divide the code into the functions of an iterator: *init*, *getCurrent*, *next*, *valid*

# Inorder binary tree iterator

- Assume an implementation without a parent node.
- What fields do we need to keep in the iterator structure?

InorderIterator:

bt: BinaryTree

s: Stack

currentNode:  $\uparrow$  BTNode

# Inorder binary tree iterator - init

- What should the *init* operation do?

**subalgorithm** init (it, bt) **is:**

//pre: it - is an InorderIterator, bt is a BinaryTree

  it.bt  $\leftarrow$  bt

  init(it.s)

  node  $\leftarrow$  bt.root

**while** node  $\neq$  NIL **execute**

    push(it.s, node)

    node  $\leftarrow$  [node].left

**end-while**

**if** not isEmpty(it.s) **then**

    it.currentNode  $\leftarrow$  top(it.s)

**else**

    it.currentNode  $\leftarrow$  NIL

**end-if**

**end-subalgorithm**

# Inorder binary tree iterator - getCurrent

- What should the *getCurrent* operation do?

```
function getCurrent(it) is:
    getCurrent ← [it.currentNode].info
end-function
```

# Inorder binary tree iterator - valid

- What should the *valid* operation do?

```
function valid(it) is:
  if it.currentNode = NIL then
    valid ← false
  else
    valid ← true
  end-if
end-function
```

# Inorder binary tree iterator - next

- What should the *next* operation do?

**subalgorithm** next(it) **is:**

```
node ← pop(it.s)
if [node].right ≠ NIL then
    node ← [node].right
    while node ≠ NIL execute
        push(it.s, node)
        node ← [node].left
    end-while
end-if
if not isEmpty(it.s) then
    it.currentNode ← top(it.s)
else
    it.currentNode ← NIL
end-if
end-subalgorithm
```

# Preorder, Inorder, Postorder

- How to remember the difference between traversals?
  - Left subtree is always traversed before the right subtree.
  - The visiting of the root is what changes:
    - PREOrder - visit the root before the left and right
    - INOrder - visit the root between the left and right
    - POSTOrder - visit the root after the left and right

- Assume you have a binary tree, but you do not know how it looks like, but you have the *preorder* and *inorder* traversal of the tree. Give an algorithm for building the tree based on these two traversals.
- For example:
  - Preorder: A B F G H E L M
  - Inorder: B G F H A L E M

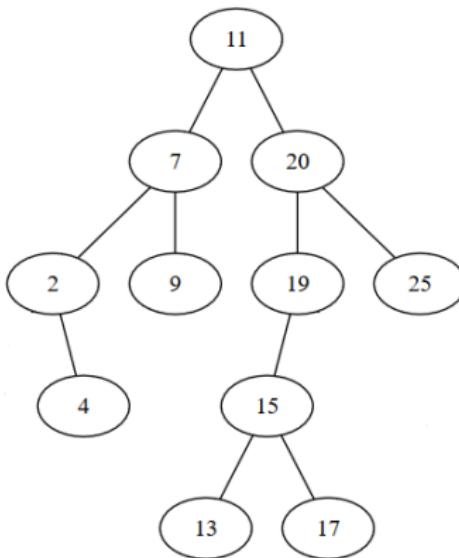
# Think about it

- Can you rebuild the tree if you have the *postorder* and the *inorder* traversal?
- Can you rebuild the tree if you have the *preorder* and the *postorder* traversal?

# Binary search trees

- A *Binary Search Tree* is a binary tree that satisfies the following property:
  - if  $x$  is a node of the binary search tree then:
    - For every node  $y$  from the left subtree of  $x$ , the information from  $y$  is less than or equal to the information from  $x$
    - For every node  $y$  from the right subtree of  $x$ , the information from  $y$  is greater than or equal to the information from  $x$
- Obviously, the relation used to order the nodes can be considered in an abstract way (instead of having " $\leq$ " as in the definition).

# Binary Search Tree Example



- If we do an inorder traversal of a binary search tree, we will get the elements in increasing order (according to the relation used).

- The terminology and many properties discussed for binary tree is valid for binary search trees as well:
  - We can have a binary search tree that is full, complete, almost complete, degenerate or balanced
  - The maximum number of nodes in a binary search tree of height  $N$  is  $2^{N+1} - 1$  - if the tree is complete.
  - The minimum number of nodes in a binary search tree of height  $N$  is  $N$  - if the tree is degenerate.
  - A binary search tree with  $N$  nodes has a height between  $\log_2 N$  and  $N$  (we will denote the height of the tree by  $h$ ).

- Binary search trees can be used as representation for sorted containers: sorted maps, sorted multimaps, priority queues, sorted sets, etc.
- In order to implement these containers on a binary search tree, we need to define the following basic operations:
  - search for an element
  - insert an element
  - remove an element
- Other operations that can be implemented for binary search trees (and can be used by the containers): get minimum/maximum element, find the successor/predecessor of an element.

# Binary Search Tree - Representation

- We will use a linked representation with dynamic allocation  
(similar to what we used for binary trees)

BSTNode:

info: TElem

left:  $\uparrow$  BSTNode

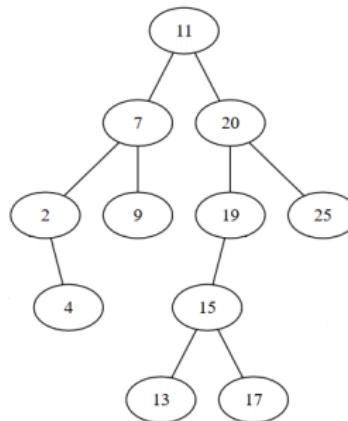
right:  $\uparrow$  BSTNode

BinarySearchTree:

root:  $\uparrow$  BSTNode

# Binary Search Tree - search operation

- How can we search for an element in a binary search tree?



- How can we search for element 15? And for element 14?

- How can we implement the *search algorithm* recursively?

# BST - search operation - recursive implementation

- How can we implement the *search algorithm* recursively?

```
function search_rec (node, elem) is:  
//pre: node is a BSTNode and elem is the TElm we are searching for  
if node = NIL then  
    search_rec ← false  
else  
    if [node].info = elem then  
        search_rec ← true  
    else if [node].info < elem then  
        search_rec ← search_rec([node].right, elem)  
    else  
        search_rec ← search_rec([node].left, elem)  
    end-if  
end-function
```

- Complexity of the search algorithm:

- Complexity of the search algorithm:  $O(h)$  (which is  $O(n)$ )
- Since the *search* algorithm takes as parameter a node, we need a wrapper function to call it with the root of the tree.

```
function search (tree, e) is:
```

```
//pre: tree is a BinarySearchTree, e is the elem we are looking for  
    search  $\leftarrow$  search_rec(tree.root, e)
```

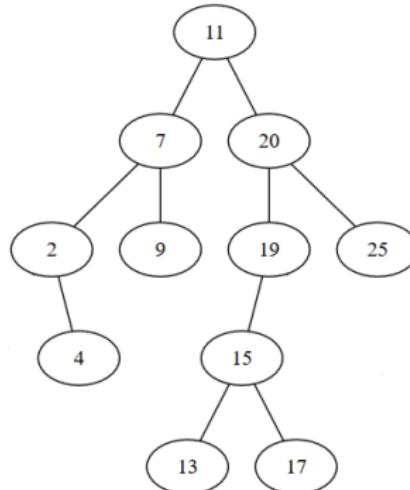
```
end-function
```

- How can we define the search operation non-recursively?

- How can we define the search operation non-recursively?

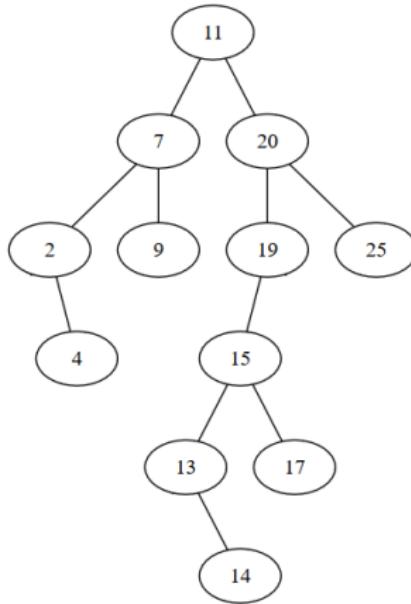
```
function search (tree, elem) is:  
    //pre: tree is a BinarySearchTree and elem is the TElem we are searching for  
    currentNode ← tree.root  
    found ← false  
    while currentNode ≠ NIL and not found execute  
        if [currentNode].info = elem then  
            found ← true  
        else if [currentNode].info < elem then  
            currentNode ← [currentNode].right  
        else  
            currentNode ← [currentNode].left  
        end-if  
    end-while  
    search ← found  
end-function
```

# BST - insert operation



- How/Where can we insert element 14?

# BST - insert operation



- How can we implement the *insert* operation?

- How can we implement the *insert* operation?
- We will start with a function that creates a new node given the information to be stored in it.

```
function initNode(e) is:
    //pre: e is a TComp
    //post: initNode ← a node with e as information
    allocate(node)
    [node].info ← e
    [node].left ← NIL
    [node].right ← NIL
    initNode ← node
end-function
```

# BST - insert operation - recursive implementation

```
function insert_rec(node, e)  is:  
  //pre: node is a BSTNode, e is TComp  
  //post: a node containing e was added in the tree starting from node  
  if node = NIL then  
    node ← initNode(e)  
  else if [node].info ≥ e then  
    [node].left ← insert_rec([node].left, e)  
  else  
    [node].right ← insert_rec([node].right, e)  
  end-if  
  insert_rec ← node  
end-function
```

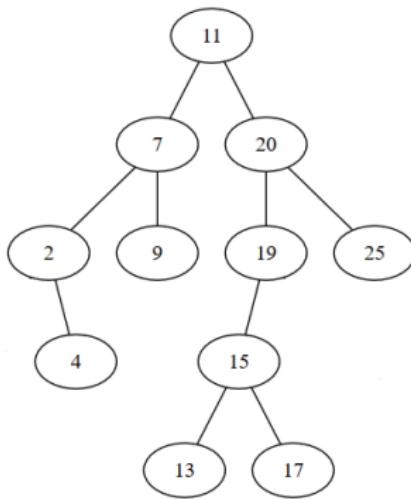
- Complexity:

# BST - insert operation - recursive implementation

```
function insert_rec(node, e) is:  
    //pre: node is a BSTNode, e is TComp  
    //post: a node containing e was added in the tree starting from node  
    if node = NIL then  
        node ← initNode(e)  
    else if [node].info ≥ e then  
        [node].left ← insert_rec([node].left, e)  
    else  
        [node].right ← insert_rec([node].right, e)  
    end-if  
    insert_rec ← node  
end-function
```

- Complexity:  $O(n)$
- Like in case of the *search* operation, we need a wrapper function to call *insert\_rec* with the root of the tree.

# BST - Finding the minimum element



- How can we find the minimum element of the binary search tree?

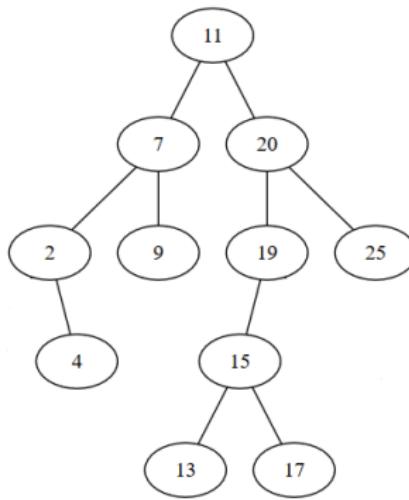
# BST - Finding the minimum element

```
function minimum(tree) is:  
    //pre: tree is a BinarySearchTree  
    //post: minimum ← the minimum value from the tree  
    currentNode ← tree.root  
    if currentNode = NIL then  
        @empty tree, no minimum  
    else  
        while [currentNode].left ≠ NIL execute  
            currentNode ← [currentNode].left  
        end-while  
        minimum ← [currentNode].info  
    end-if  
end-function
```

- Complexity of the minimum operation:

- Complexity of the minimum operation:  $O(n)$
- We can have an implementation for the minimum, when we want to find the minimum element of a subtree, in this case the parameter to the function would be a node, not a tree.
- We can have an implementation where we return the node containing the minimum element, instead of just the value (depends on what we want to do with the operation)
- Maximum element of the tree can be found similarly.

# Finding the parent of a node



- Given a node, how can we find the parent of the node?  
(assume a representation where the node has no parent field).

# Finding the parent of a node

```
function parent(tree, node) is:  
    //pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node ≠ NIL  
    //post: returns the parent of node, or NIL if node is the root  
    c ← tree.root  
    if c = node then //node is the root  
        parent ← NIL  
    else  
        while c ≠ NIL and [c].left ≠ node and [c].right ≠ node execute  
            if [c].info ≥ [node].info then  
                c ← [c].left  
            else  
                c ← [c].right  
            end-if  
        end-while  
        parent ← c  
    end-if  
end-function
```

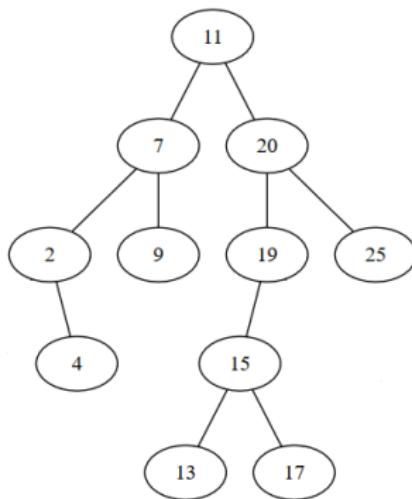
- Complexity:

# Finding the parent of a node

```
function parent(tree, node) is:  
    //pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node ≠ NIL  
    //post: returns the parent of node, or NIL if node is the root  
    c ← tree.root  
    if c = node then //node is the root  
        parent ← NIL  
    else  
        while c ≠ NIL and [c].left ≠ node and [c].right ≠ node execute  
            if [c].info ≥ [node].info then  
                c ← [c].left  
            else  
                c ← [c].right  
            end-if  
        end-while  
        parent ← c  
    end-if  
end-function
```

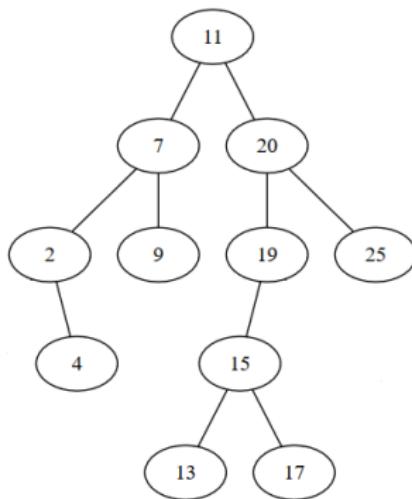
- Complexity:  $O(n)$

# BST - Finding the successor of a node



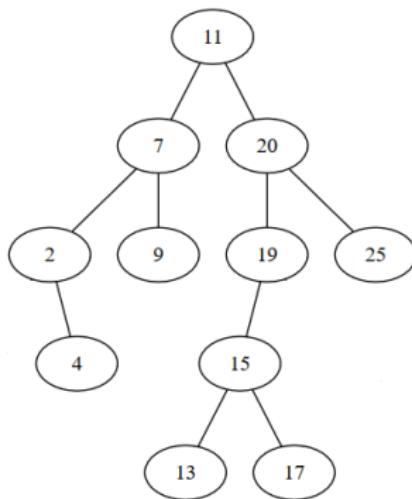
- Given a node, how can we find the node containing the next value (considering the relation used for ordering the elements)?
- How can we find the next after 11?

# BST - Finding the successor of a node



- Given a node, how can we find the node containing the next value (considering the relation used for ordering the elements)?
- How can we find the next after 11? After 13?

# BST - Finding the successor of a node



- Given a node, how can we find the node containing the next value (considering the relation used for ordering the elements)?
- How can we find the next after 11? After 13? After 17?

# BST - Finding the successor of a node

```
function successor(tree, node) is:  
    //pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node ≠ NIL  
    //post: returns the node with the next value after the value from node  
    //or NIL if node is the maximum  
    if [node].right ≠ NIL then  
        c ← [node].right  
        while [c].left ≠ NIL execute  
            c ← [c].left  
        end-while  
        successor ← c  
    else  
        p ← parent(tree, c)  
        while p ≠ NIL and [p].left ≠ c execute  
            c ← p  
            p ← parent(tree, p)  
        end-while  
        successor ← p  
    end-if  
end-function
```

# BST - Finding the successor of a node

- Complexity of successor:

## BST - Finding the successor of a node

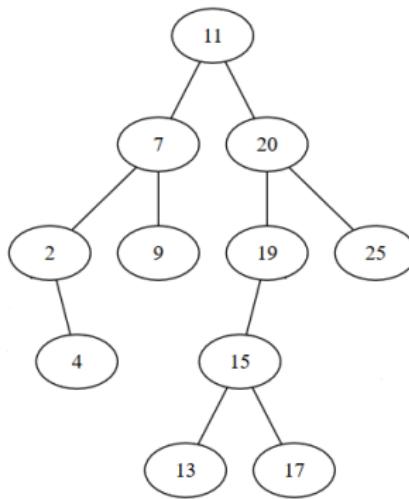
- Complexity of successor: depends on parent function:
  - If *parent* is  $\Theta(1)$ , complexity of successor is  $O(n)$
  - If *parent* is  $O(n)$ , complexity of successor is  $O(n^2)$
- What if, instead of receiving a node, successor algorithm receives as parameter a value from a node (assume unique values in the nodes)? How can we find the successor then?
- Similar to successor, we can define a predecessor function as well.

# BST - Finding successor of a node

- If we do not have direct access to the parent, finding the successor of a node is  $O(n^2)$ .
- Can we reduce this complexity?

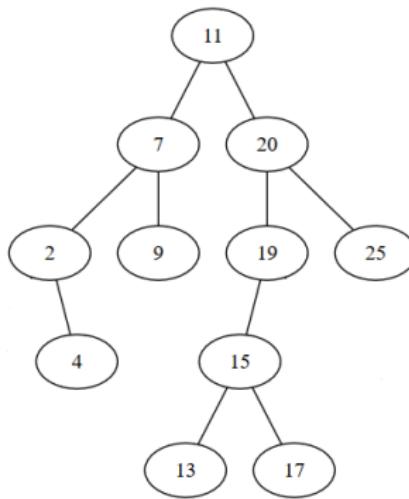
- If we do not have direct access to the parent, finding the successor of a node is  $O(n^2)$ .
- Can we reduce this complexity?
- $O(n^2)$  is given by the potentially repeated calls for the *parent* function. But we only need the *last* parent, where we went left. We can do one single traversal from the root to our node, and every time we continue left (i.e. current node is greater than the one we are looking for) we memorize that node in a variable (and change the variable when we find a new such node). When the current node is at the one we are looking for, this variable contains its successor.

# BST - Remove a node



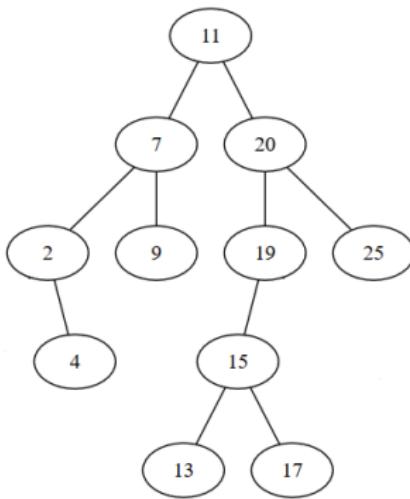
- How can we remove the value 25?

# BST - Remove a node



- How can we remove the value 25? And value 2?

# BST - Remove a node



- How can we remove the value 25? And value 2? And value 11?

# BST - Remove a node

- When we want to remove a value (a node containing the value) from a binary search tree we have three cases:
  - The node to be removed has no descendant
    - Set the corresponding child of the parent to NIL
  - The node to be removed has one descendant
    - Set the corresponding child of the parent to the descendant
  - The node to be removed has two descendants
    - Find the maximum of the left subtree, move it to the node to be deleted, and delete the maximum  
**OR**
    - Find the minimum of the right subtree, move it to the node to be deleted, and delete the minimum

# Binary Search Tree

- Think about it:
  - Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?

- Think about it:

- Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?
- Lists have positions and operations based on positions. In case of a SortedList we have an operation to remove an element from a position and to return an element from a position (but no insert to position). How can we keep track of positions in a binary search tree?

- Think about it:

- Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?
- Lists have positions and operations based on positions. In case of a SortedList we have an operation to remove an element from a position and to return an element from a position (but no insert to position). How can we keep track of positions in a binary search tree?
- We can keep in every node, besides the information, the number of nodes in the left subtree. This gives us automatically the "position" of the root in the SortedList. When we have operations that are based on positions, we use these values to decide if we go left or right.

# Binary Search Tree

