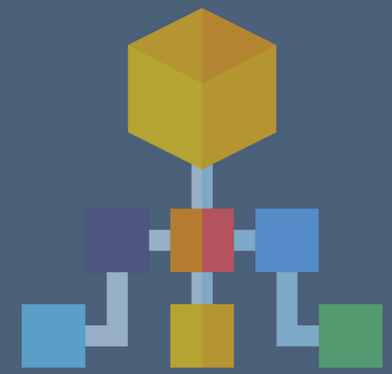
Data Structures Design

Data Structures - Recall

Dr. Lilia SFAXI



Introduction to Data Structures

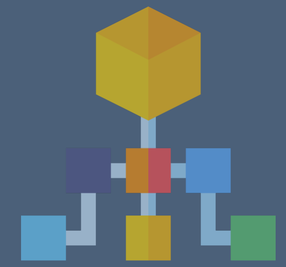
Data Structures Design

Dr. Lilia SFAXI

eFrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

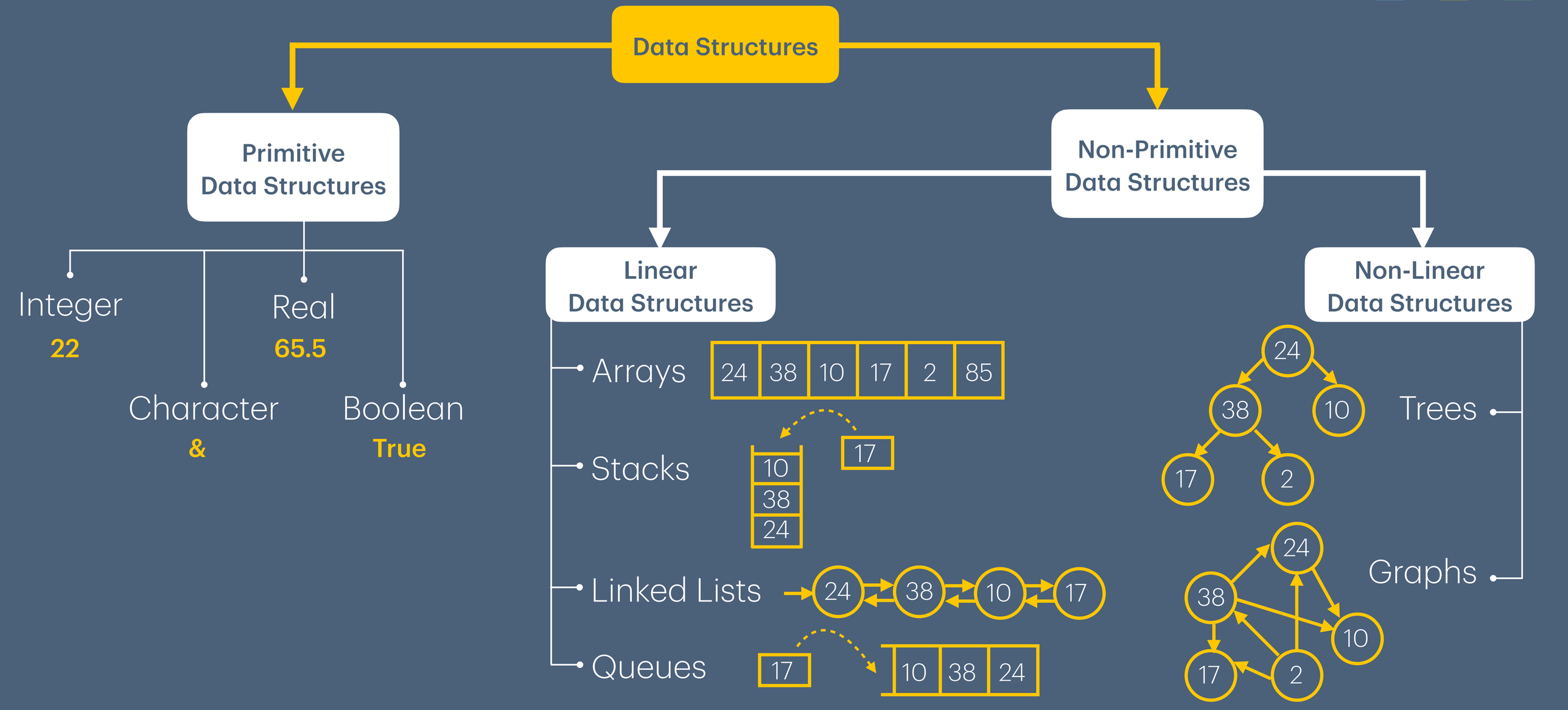
What is a Data Structure?

Data Structures - Recall



- A data structure is a **logical and mathematical model** of a particular organisation of data
- It provides means to manage large amounts of data efficiently
- The choice of a particular data structure depends on :
  - Its capacity to **represent** the inherent relationships of the data
  - Its simplicity in **storing and processing** data when necessary

Classification of Data Structures



Lists

Linear Data Structures

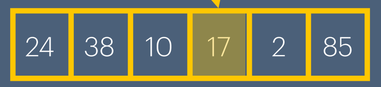
Dr. Lilia SFAXI

eFrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

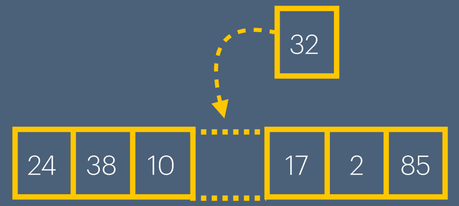
Abstract List

Linear Data Structures

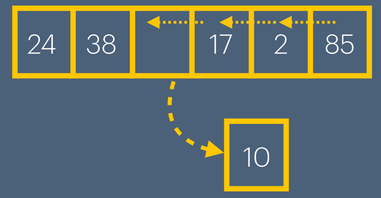
- An Abstract List (or List ADT) is linearly ordered data where the programmer explicitly defines the ordering
- Common Operations



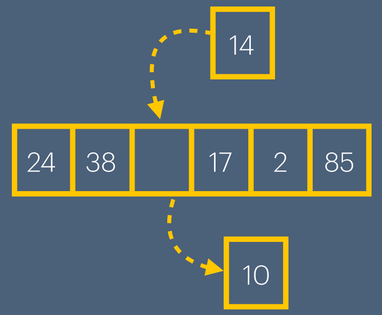
Access an element



Insert an element



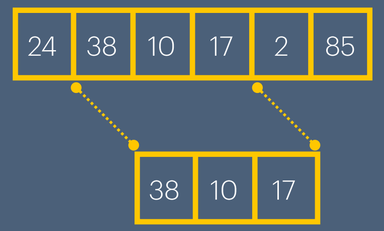
Remove an element



Replace an element

- Advanced Operations

Concatenate two lists

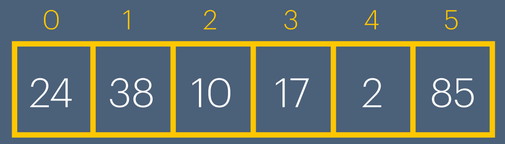


Determine if one is a sub-list of the other

Abstract List

Linear Data Structures

- Abstract lists can be implemented mainly with two structures
  - Arrays



  - Linked Lists

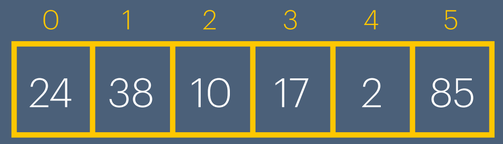
Singly Linked Lists

Doubly Linked Lists

Arrays

Linear Data Structures

- A collection of elements identified by index or key.



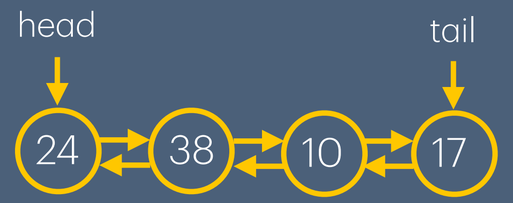
- Characteristics:
  - Fixed Size
  - Elements stored in contiguous memory locations
  - Random access through indices
- Common operations
  - Access by index: **O(1)**
  - Insertion / Deletion : **O(n)** if not at the end, else **O(1)**

Linked Lists

Linear Data Structures

- A sequence of nodes where each node contains data and a reference (link) to the next node.
- Types:
  - **Singly Linked List**: Each node points to the next node.

  - **Doubly Linked List**: Each node points to both the next and previous nodes.



- Characteristics:
  - Dynamic size.
  - Insertion: **O(1)** (append and prepend), **O(n)** (*k-th* entry).
  - Deletion: **O(1)** (pop first, and pop for DLL), **O(n)** (*k-th* entry, and pop for SLL),
  - Sequential access: **O(n)**

Stacks and Queues

Linear Data Structures

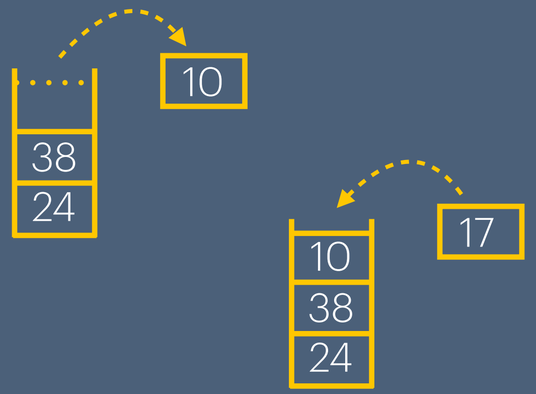
Dr. Lilia SFAXI

eFrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

Stacks

Linear Data Structures

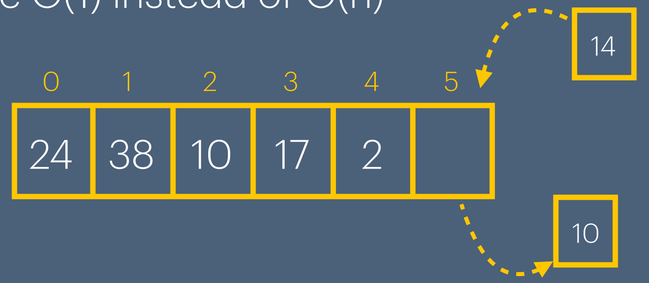
- A Stack is an abstract data type which emphasises specific operations:
  - Uses an explicit linear ordering
  - Insertions and removals are performed individually
  - Inserted objects are pushed onto the stack
  - The top of the stack is the most recently object pushed onto the stack
  - When an object is popped from the stack, the current top is erased
- A stack is also called a **LIFO** (Last In First Out)
- Only two operations are possible on a stack:
  - **Pop** : removes the last inserted item
  - **Push**: inserts a new item



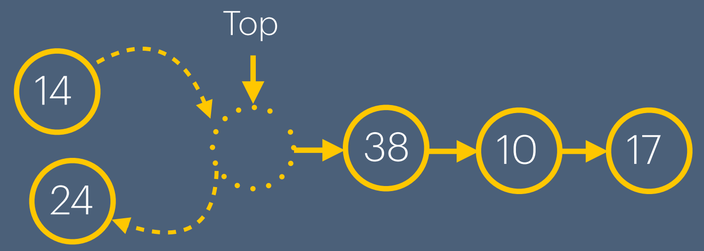
Stacks : *Implementation*

Linear Data Structures

- Common ways to implement a stack:
  - As an **Array**, where the last item is considered to be the top of the stack
    - This way, the pop and push operations are O(1) instead of O(n)



  - As a **Linked List**, where the head is used as the top of the stack
    - So that pop and push are O(1) instead of O(n)



Queues

Linear Data Structures

- A Queue is an abstract data type which emphasises specific operations:
  - Uses an explicit linear ordering
  - Insertions and removals are performed individually
  - There are no restrictions on objects inserted into (pushed onto) the queue—that object is designated the back of the queue
  - The object designated as the front of the queue is the object which was in the queue the longest
  - The remove operation (*popping* from the queue) removes the current front of the queue
- A queue is also called a **FIFO** (First In First Out)
- Only two operations are possible on a queue:
  - **Enqueue** : inserts a new item at the end of the queue

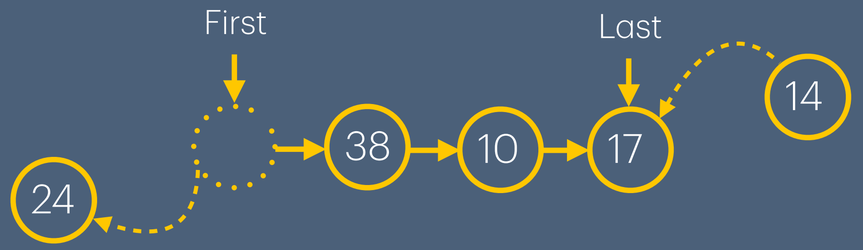
  - **Dequeue**: removes the first inserted item

Queues : *Implementation*

Linear Data Structures

- Common ways to implement a queue:
  - As an **Array**
    - However, whether the first item is the head or the last item, one operation (enqueue or dequeue) will always be O(n)

  - As a **Linked List**, where the head is used as the first item of the queue and the tail the last one
    - So that enqueue and dequeue are both O(1)



Trees

Non-Linear Data Structures

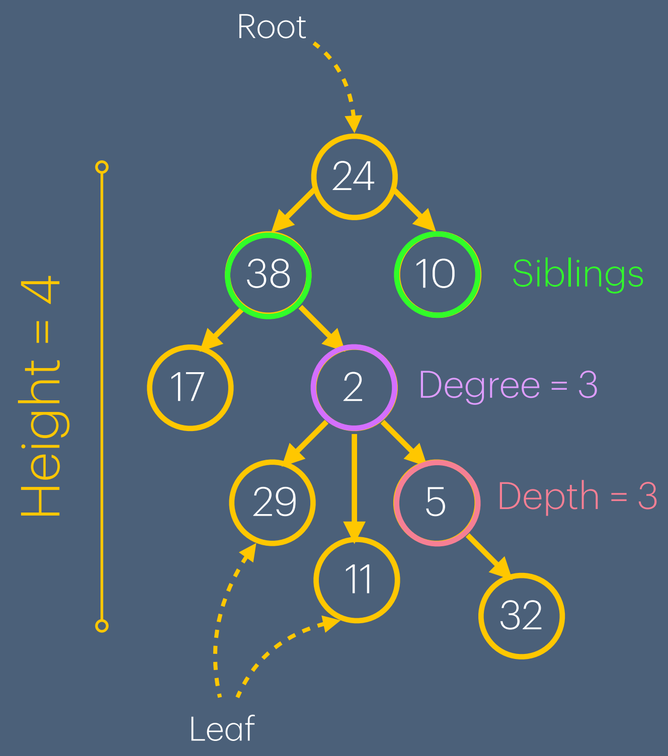
Dr. Lilia SFAXI

eFrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

Trees

Non-Linear Data Structures

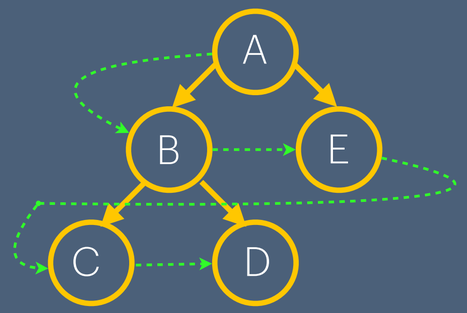
- A rooted tree data structure stores information in nodes
  - There is a first node, or **root**
  - Each node has a variable number of references to successors
  - Each node, other than the root, has exactly one node pointing to it
  - All nodes will have zero or more child nodes or children
  - For all nodes other than the root node, there is one parent node
- The **degree** of a node is defined as the number of its children
- Nodes with the same parent are **siblings**
- Nodes with degree zero are also called **leaf nodes**
- For each node in a tree, there exists a unique **path** from the root node to that node
  - The length of this path is the **depth** of the node
- The **height** of a tree is defined as the maximum depth of any node within the tree



Trees : *Traversals*

Non-Linear Data Structures

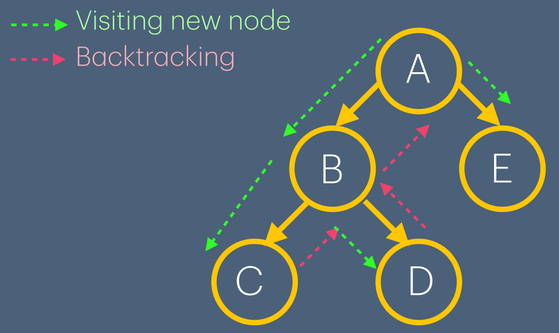
Breadth-First Traversal (BFS)



Visits all nodes at depth k before proceeding onto depth $k+1$

A - B - E - C - D

Depth-First Traversal (DFS)



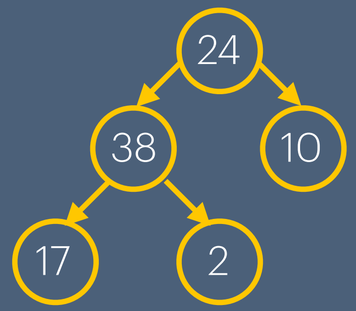
Explores as far down a branch as possible before backtracking.

A - B - C - D - E

Trees : *Types*

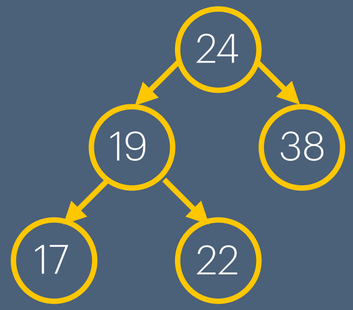
Non-Linear Data Structures

Binary Trees



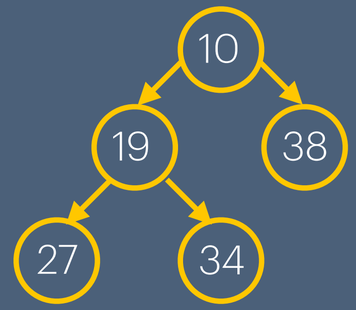
A tree in which each node has at most 2 children.

Binary Search Trees



A binary tree where each node's left child contains values less than the node, and the right child contains values greater than the node.

Heap (Min-Heap)



The value of each node is smaller (higher) than those of all its descendants

Min-Heap: Parent nodes are <= child nodes.

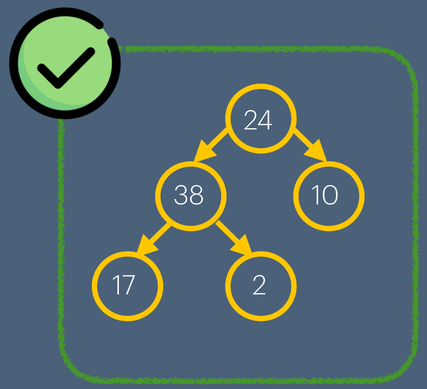
Max-Heap: Parent nodes are >= to child nodes.

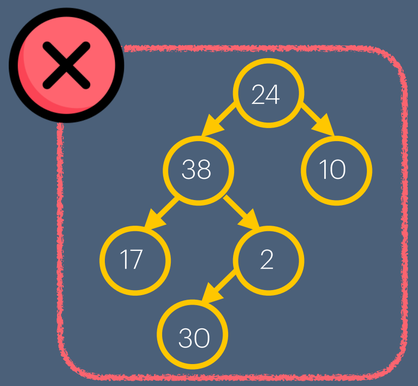
Binary Trees : *Characteristics*

Non-Linear Data Structures

FULL

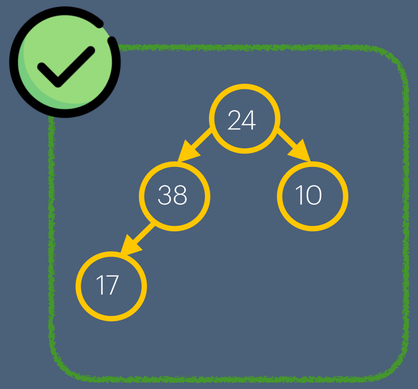
Every node either points to zero nodes or two nodes

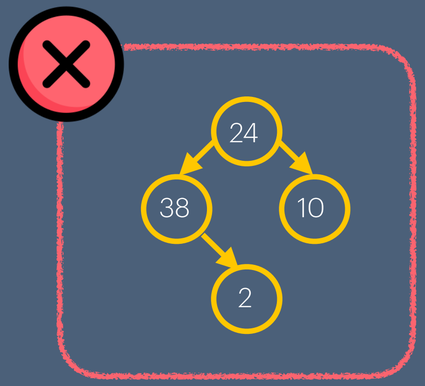




COMPLETE

The tree is filled from left to right with no gaps

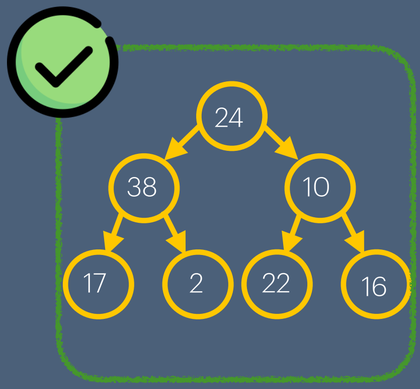


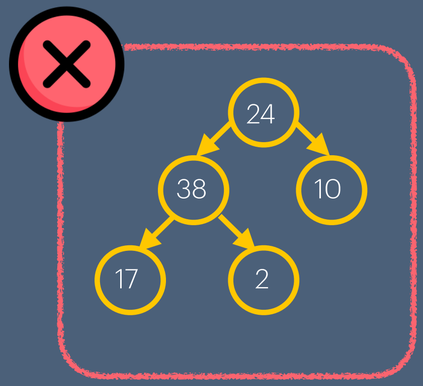


PERFECT

All internal nodes have exactly two children.

All leaf nodes are at the same level.

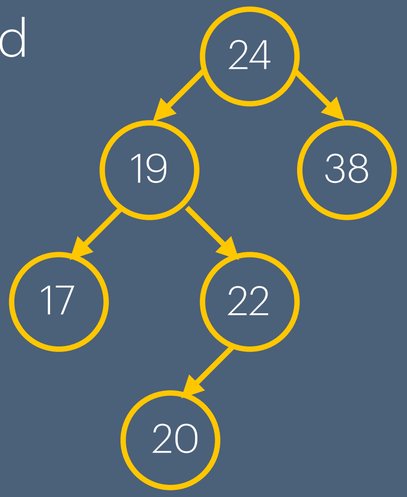


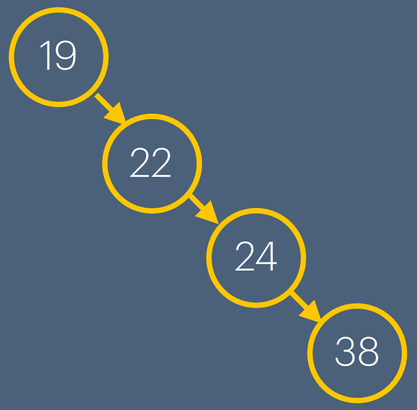


Binary Search Tree (BST)

Non-Linear Data Structures

- A binary tree where each node's left child contains values less than the node, and the right child contains values greater than the node.
- If you take any node in a BST:
  - All nodes below it to the right have higher values
  - All nodes below it to the left have lower values
- Possible operations:
  - Lookup, insertion and remove: **$O(\log_n)$**
    - We are doing a *divide and conquer* by discarding half of the tree for every operation
  - However the absolute worst case would be to have a tree with only right (or left) node
    - Then, time complexity will become **O(n)**
    - However, we never consider it as such, as this case is very rare





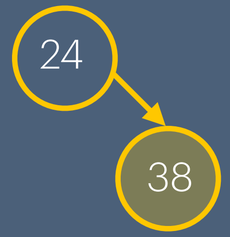
Binary Search Trees (BST): *Insertions*

Non-Linear Data Structures

STEP 1 : Insert 24

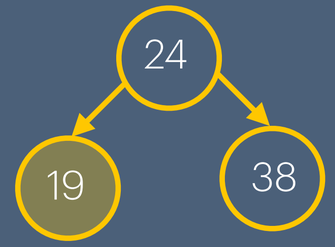
STEP 2 : Insert 38

38> 24 -> we place it on the left



STEP 3 : Insert 19

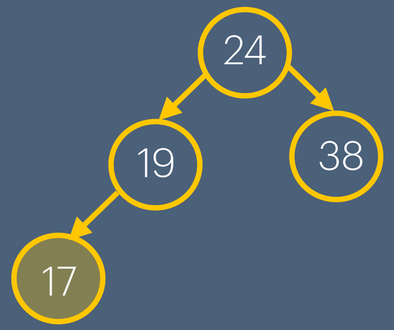
19 < 24 -> we place it on the right



STEP 4 : Insert 17

17 < 24 -> we go to the left

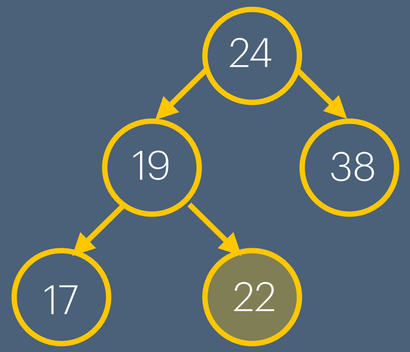
17 < 19 -> we place it on the left



STEP 5 : Insert 22

22 < 24 -> we go to the left

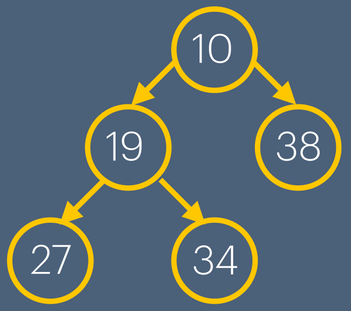
22 > 19 -> we place it on the right



Heaps

Non-Linear Data Structures

- The value of each node is smaller (higher) than those of all its descendants
  - **Min-Heap**: Parent nodes are <= child nodes.
  - **Max-Heap**: Parent nodes are >= to child nodes.

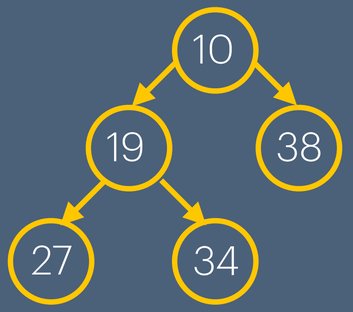


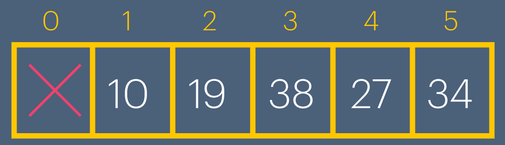
- Other than that, there is no relation between the elements of the same level
- Heaps are not good for searching
  - Finding a value may sometimes require traversing the whole tree, as there is no indication on where to find it
- They are used to keep track of largest (smallest) item, to pop it quickly
  - Used as **Priority Queues**

Heaps: *Implementation*

Non-Linear Data Structures

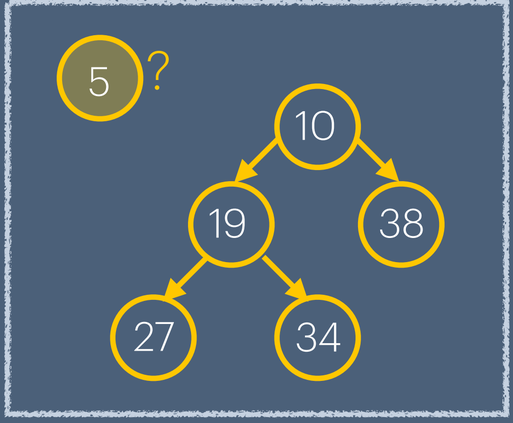
- A heap is always **COMPLETE**
  - There are no gaps from left to right
- A heap is stored in a list
  - Only the values are stored
  - The elements are stored in the order of their appearance in the tree, from left to right
  - It is recommended to store the list starting from index 1 instead of 0, to facilitate the lookup
    - left_child_index = 2 * parent_index
    - right_child_index = 2 * parent_index + 1
- Possible operations on heaps:
  - Insert: **O(log$_n$)**
  - Pop the top node: **O(log$_n$)**



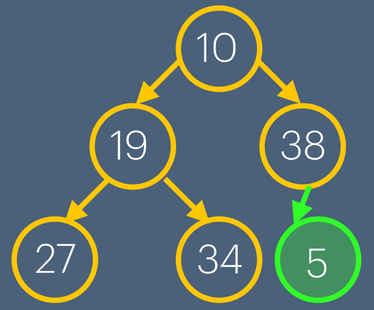


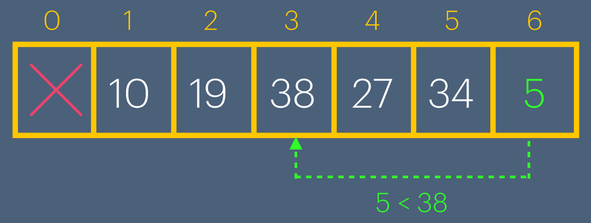
Heaps: *Insertion*

Non-Linear Data Structures

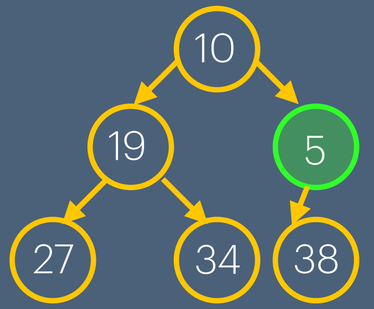


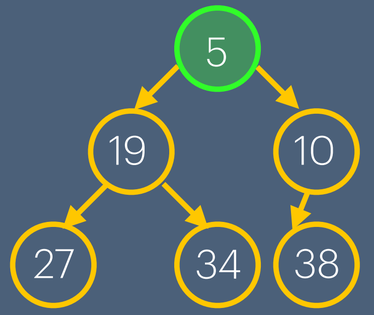
- Inserting a new value in the heap goes like this:
  1. We first start by inserting the value at the end of the array
     - This way, the heap is always *COMPLETE*

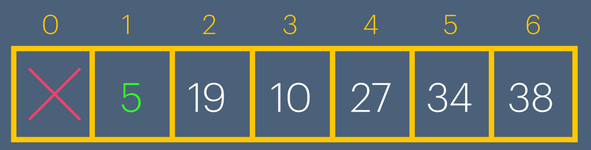




  2. We bubble the node up to its corresponding spot
     - We compare it recursively with the parent
     - If it is lower, we swap them
       - We compare again...
     - If not, it stays there, and the insertion is finished

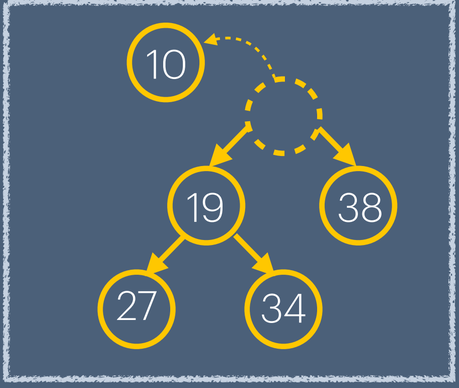




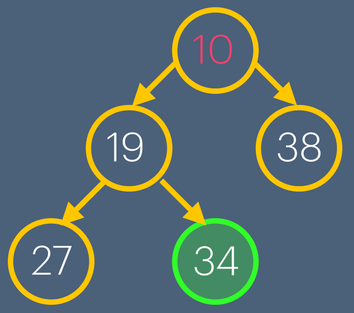


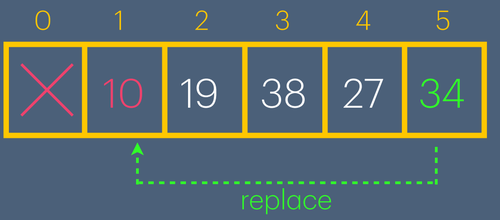
Heaps: *Remove*

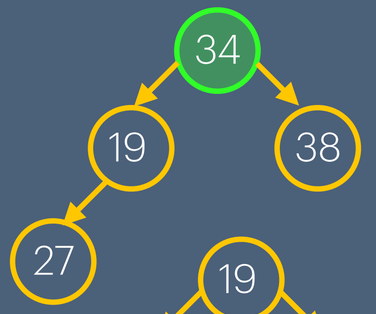
Non-Linear Data Structures

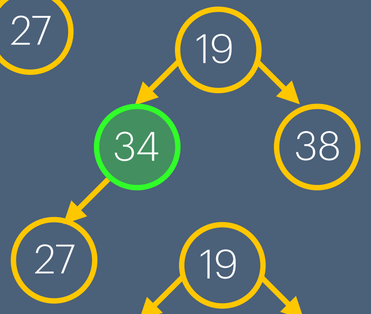


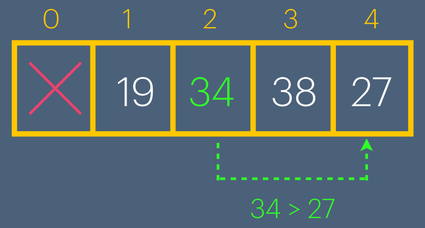
- Removing the top element from the heap goes as follows:
  1. We first start by deleting the top element, and replacing it by the last element
     - This way, the heap is always *COMPLETE*

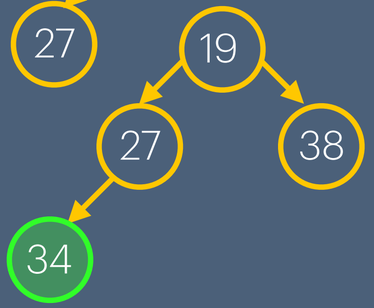












  2. We sink the node to its corresponding spot
     - We compare it recursively with its children
     - If it is higher, we swap it with the minimum value
       - We compare again...
     - If not, it stays there, and the deletion is finished

Graphs

Non-Linear Data Structures

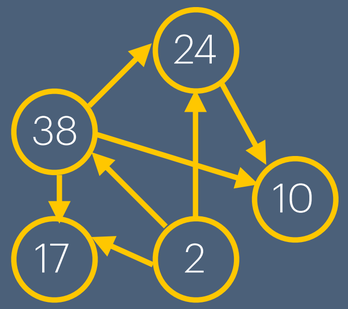
Dr. Lilia SFAXI

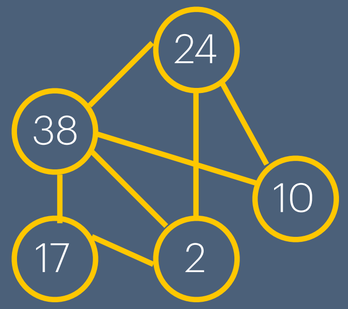
eFrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

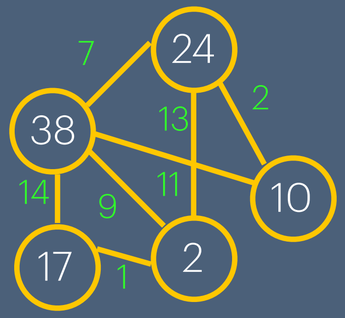
Graphs

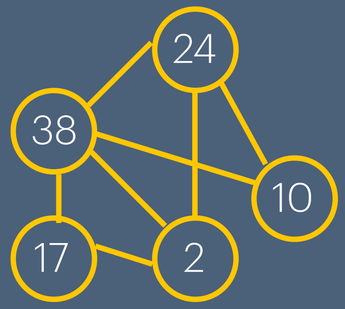
Non-Linear Data Structures

- A graph is composed of:
  - Nodes 

  - Edges 
- A graph can be:
  - Directed 

  or undirected 

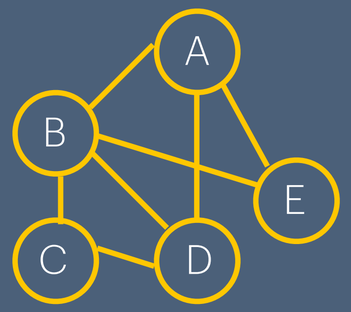
  - Weighted 

  or unweighted 

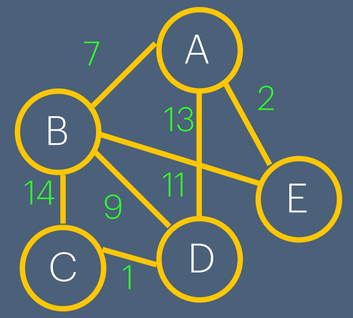
Graphs: *Representation*

Non-Linear Data Structures

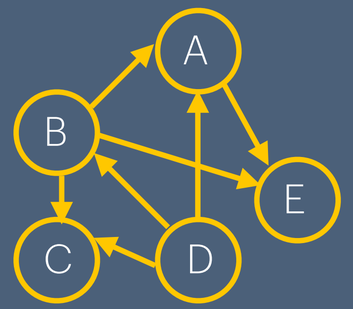
- There are two main ways to represent graphs:
  - As an adjacency matrix



	A	B	C	D	E
A		1	0	1	1
B	1		1	1	1
C	0	1		1	0
D	1	1	1		0
E	1	1	0	0	

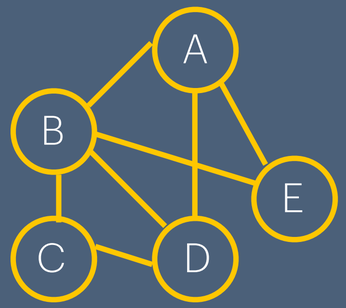


	A	B	C	D	E
A		7	0	13	2
B	7		14	9	11
C	0	14		1	0
D	13	9	1		0
E	2	11	0	0	

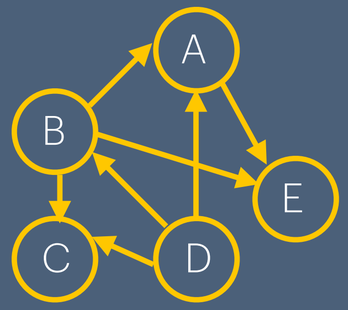


	A	B	C	D	E
A		0	0	0	1
B	1		1	0	1
C	0	0		0	0
D	1	1	1		0
E	0	0	0	0	

  - As an adjacency list



A	[B, E, D]
B	[A, E, D, C]
C	[B, D]
D	[A, B, C]
E	[A, B]

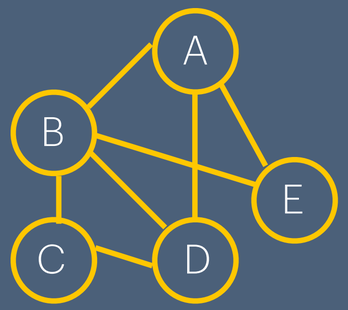


A	[E]
B	[A, E, C]
C	[]
D	[A, B, C]
E	[]

Graphs: *Operations*

Non-Linear Data Structures

Graph



Adjacency Matrix

	A	B	C	D	E
A		1	0	1	1
B	1		1	1	1
C	0	1		1	0
D	1	1	1		0
E	1	1	0	0	

Adjacency List

A	[B, E, D]
B	[A, E, D, C]
C	[B, D]
D	[A, B, C]
E	[A, B]

Space Complexity	$O(\|V\|^2)$	$O(\|V\|+\|E\|)$
Adding a Vertex	$O(\|V\|^2)$	$O(1)$
Adding an Edge	$O(1)$	$O(1)$
Removing an Edge	$O(1)$	$O(\|E\|)$
Removing a Vertex	$O(\|V\|^2)$	$O(\|V\|+\|E\|)$