# Lab 6 – Trees & Heaps

**Instructors:** Lorenzo De Carli & Maan Khedr

Slides by Lorenzo De Carli

# What we are going to focus on today

- Understanding implementation and performance of trees and heaps

# Exercise #1: BST performance [1 pt]

- BST performance depends on whether the tree is balanced or not
- In this exercise we are going to look at randomization as a way to improve balancing

# Exercise #1 /2

1. Implement a binary search tree with insertion and search operations as seen in class [0.2 pts]
2. Measure search performance using timeit as follows: [0.3 pts]
    1. Generate a 10000-element sorted vector and use it to build a tree by inserting each element
    2. Search each element. Time the search (averaged across 10 tries for each element), and return average and total time.
3. Measure search performance using timeit as follows: [0.3 pts]
    1. Shuffle the vector used for question 2 (using random.shuffle)
    2. Search each element. Time the search (averaged across 10 tries for each element), and return average and total time
4. Discuss the results. Which approach is faster? Why? [0.2 pts]

# Exercise #1 - What to deliver

- Submit a file named ex1.py with your answer to questions 1, 2 and 3
- The file should also contain the answer to question 4 as code comment

# Exercise #2 – BST vs array [1 pt]

- A reasonable question is whether BSTs are faster than arrays for value search
- In this exercise, we will explore this question

# Exercise #2 /2

1. Implement a binary search tree with insertion and search operations as seen in class, and binary search in arrays as seen in class [0.2 pts]
2. Measure BST performance using timeit as follows: [0.3 pts]
    1. Generate a 10000-element sorted vector, shuffle, and use it to build a tree by inserting each element
    2. Search each element. Time the search (averaged across 10 tries for each element), and return average and total time
3. Using the same shuffled vector from question 2: [0.3 pts]
    1. Sort the vector
    2. Search each element using binary search. Time the search (averaged across 10 tries for each element), and return average and total time
4. Discuss: which approach is faster? Why do you think is that? [0.2 pts]

# Exercise #2 - What to deliver

- Submit a file named ex2.py with your answer to questions 1, 2 and 3
- The file should also contain the answer to question 4 as code comment

# Exercise #3: execute programs [3 pts]

- In this exercise, you will write an interpreter for arithmetic expressions of the form: `( ( 1 + 2 ) + ( 3 * ( 4 * 5 ) ) )`
- Your interpreter will build binary trees for each expression, and use post-order traversal for computing its result
- Your submission should be a script that receives an expression as a command line parameter and returns its value by printing it on the terminal

# Exercise #3: specifications and assumptions

- Each expression E can be recursively defined as:
  - `E -> (E+E)`
  - `E -> (E*E)`
  - `E -> (E-E)`
  - `E -> (E/E)`
  - `E -> <integer number>`
- In practice, this means that your expression will consist of arithmetic operations between integers, with parentheses to express priorities
- Note, negative numbers/expressions are not supported
  - In other words, no expressions of the form “-5” or “-(6+42)”

# Exercise #3: more specifications/assumptions

- Examples of valid expressions:
  - `5`
  - `( 1 + 1 )`
  - `( 6 – 3 ) * 4`
  - `(  ( 5 – 28 ) * 345 ) – ( 6 / 2 ) ) * 0`
- Also note, an input expressions may only include integers, but division is allowed so the value of an expression may not be an integer
- You can assume that operations will appear in parentheses, but not constants – e.g., you will have `( 1 + 3 )` but not `( 5 )`
- Finally, you can assume that, in the input, each token is separated by a space - e.g., `( ( 1 + 5 ) / (4 – 3 ) ) * 6 )`
  - This implies that you can use `split()` to divide each expression into tokens

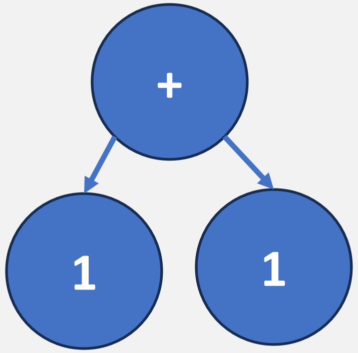
# Exercise #3: what to do

- Parse an input expression into a tree [2 pts]
  - Each instance of an operator “+”, “-”, “*”, “/” defines a node
  - The left operand of that operator (be it an expression or a constant) must be the left child
  - The right operand of that operator must be the right child
  - Note, parentheses “(” and “)” are used to define priorities but do not need to be represented by nodes in the tree)

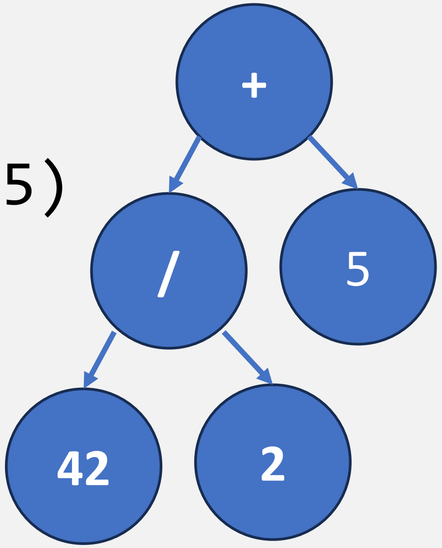
# Exercise #3 – examples of expressions and trees

5

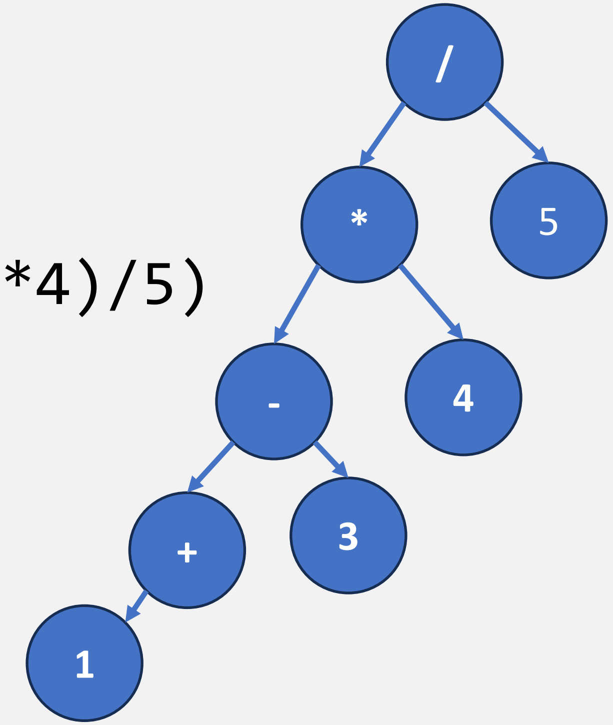

(1+1)




((42/2)+5)




((((1+2)-3)*4)/5)



# Exercise #3: what to do /2

- Using post-order traversal, compute the value of an expression and print it to terminal [1 pt]
- Example of execution:

```
> python ex3.py "5 + ( 3 / 1 )"
> 8
>
> python ex3.py 4
> 4
```

# Exercise #3 - What to deliver

- Submit a file named ex3.py implementing the requested functionality

# Exercise #4 - Build a heap [1 pt]

- In this exercise, you will build a simple heap implementation in Python and come up with a few unit tests

# Exercise #4 /2

- Write a class named which uses a Python array as a storage backend for heap nodes and:
  - Has a `heapify` method which receives as input an array of integers, stores into the internal array, and turn it into a heap [0.3 pts]
  - Has an `enqueue` method which adds an element to the heap (while correctly maintaining the heap's properties) [0.3 pts]
  - Has a `dequeue` method which removes an element from the heap (while correctly maintaining the heap's properties) [0.3 pts]
- Write 3 tests for the following cases: [0.1 pts]
  - Input array is already a correctly sorted heap
  - Input array is empty
  - Input array is a long, randomly shuffled list of integers
- Each test must consist of running the code on an appropriate input, and comparing the output (heapified array) with the expected value

# Exercise #4 - What to deliver

- Submit a file named ex4.py implementing the requested functionality

# Exercise #5: are heaps useful?

- In this exercise, you will compare a priority queue based on a heap with a priority queue based on a linked list

# Exercise #5 /2

1. Implement a class ListPriorityQueue which implements a priority queue using a linked list: [0.2 pts]
   - enqueue must insert an element in order
   - dequeue must retrieve the first (smallest) element on a list
2. Implement a class HeapPriorityQueue which implements a priority queue using a heap: [0.2 pts]
   1. Can reuse implementation from Exercise 4
3. Measure execution time of both implementations [0.4 pts]
   1. Generate a random list of 1000 tasks, where a task is enqueue of a random integer with probability 0.7, and dequeue with probability 0.3
   2. Use timeit to measure how long it takes for each implementation to process the list. Return overall time and average time per task
4. Discuss the results: which implementation is faster? Why do you think is that? [0.2 pts]

# Exercise #5 - What to deliver

- Submit a file named ex1.py with your answer to questions 1, 2 and 3
- The file should also contain the answer to question 4 as code comment

# How to submit

- Upload a zip file to the “Lab 5” dropbox on D2L, containing the required content for every exercise

# Grading rubric

- You get **3 pts** for uploading a **partial solution** by **end of lab**
  - **Must not be an empty file or irrelevant material**
- Then, you'll have until **11:59PM of the day before the next lab** to upload the **complete solution**. That will be graded as follows:
  - Exercise 1, 2, 4, 5: 1 pt each
  - Exercise 3: 3 pts
  - **Can upload the complete solution to the same dropbox**

# That’s all folks!