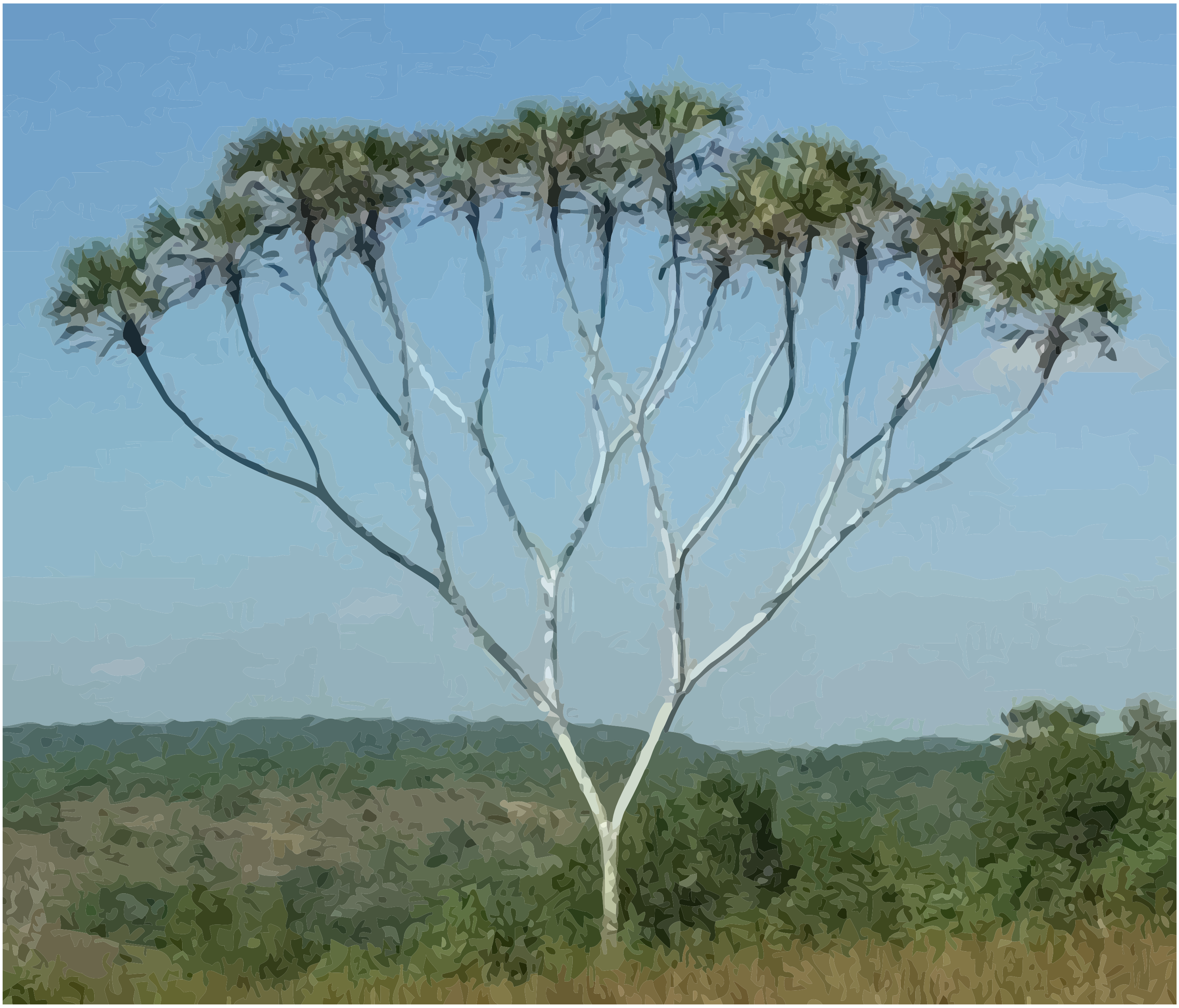
12.0 - Week 12 - Workshop (MA)

Learning Objectives

- Implementing Binary Trees.
- Implementing Binary Search Trees.
- Understanding Max Heaps

Week 12 Padlet Discussion Board link: https://monashmalaysia.padlet.org/fermi/2022week12



Binary Trees

Question 1 *Submitted Oct 17th 2022 at 11:28:00 am*

In Computer Science, the concept of **binary tree** is quite popular. So what is it?

● It is a tree data structure, in which a node has at most two children, normally called *left child* and *right child*.

○ It is a tree data structure, in which a node has children marked solely by sequences of 0's and 1's.

○ That's a joke as *proper gum trees* aren't binary!

Question 2 *Submitted Oct 17th 2022 at 11:28:09 am*

Given a binary tree containing N nodes, what is the ***maximum height*** such a tree can get?

□ 1

☑ $\log N$ if the tree is properly *balanced*

☑ N if the tree is *unbalanced*

□ N^2

Question 3 *Submitted Oct 17th 2022 at 11:28:16 am*

How many children can a ***non-leaf node*** have in a binary tree?

○ Easy, 1!

○ Of course 2!

● Maybe at most 2?

Question 4 *Submitted Oct 17th 2022 at 11:28:28 am*

How many parents can a ***non-root node*** have in a binary tree?

● Easy, 1!

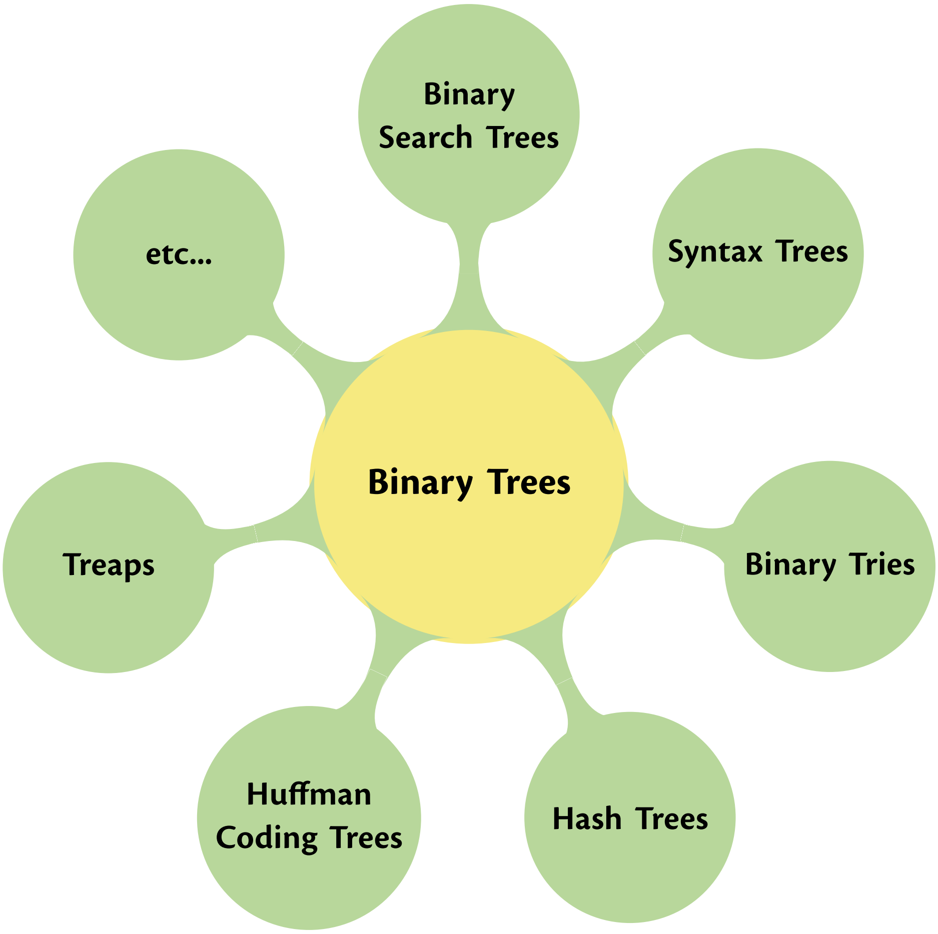
○ Of course 2!

○ Maybe at most 2?..

○ Wait, what?!

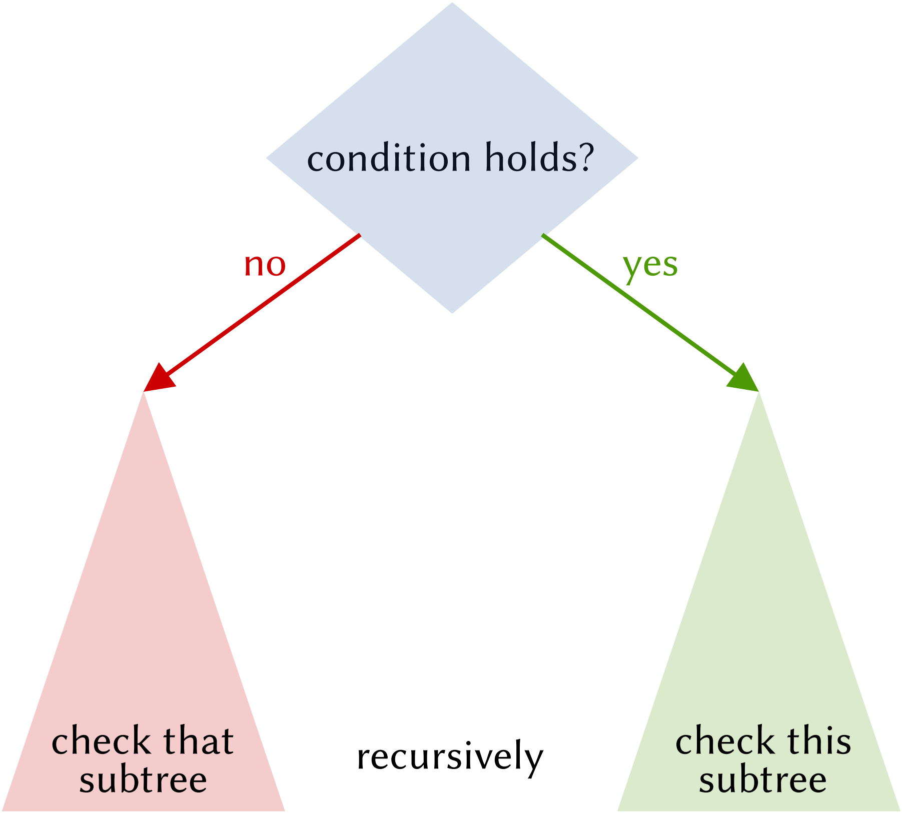
Why Binary Trees?

Binary trees is a foundation for a large number of *more complex* data structures!

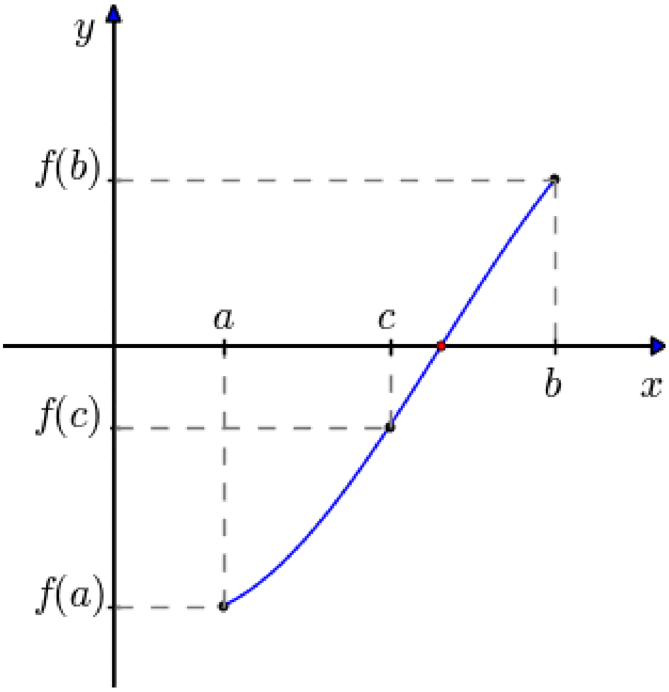


Behind Binary Search

Binary trees *implicitly* underlie any binary search algorithm!



In fact, binary trees can be related to the **dichotomy principle**. As an example, recall the **bisection method** in maths!

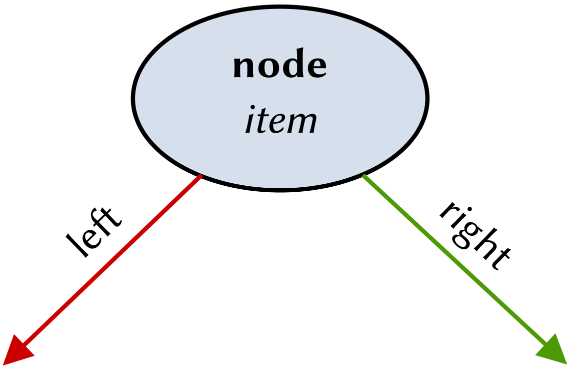


y
$f(b)$
a
c
b
x
$f(c)$
$f(a)$

Implementing Binary Trees

Typical representation of a node:

```
class BinaryTreeNode:
    """ A typical implementation of a binary tree node. """
    self.item = None
    self.left = None
    self.right = None
```

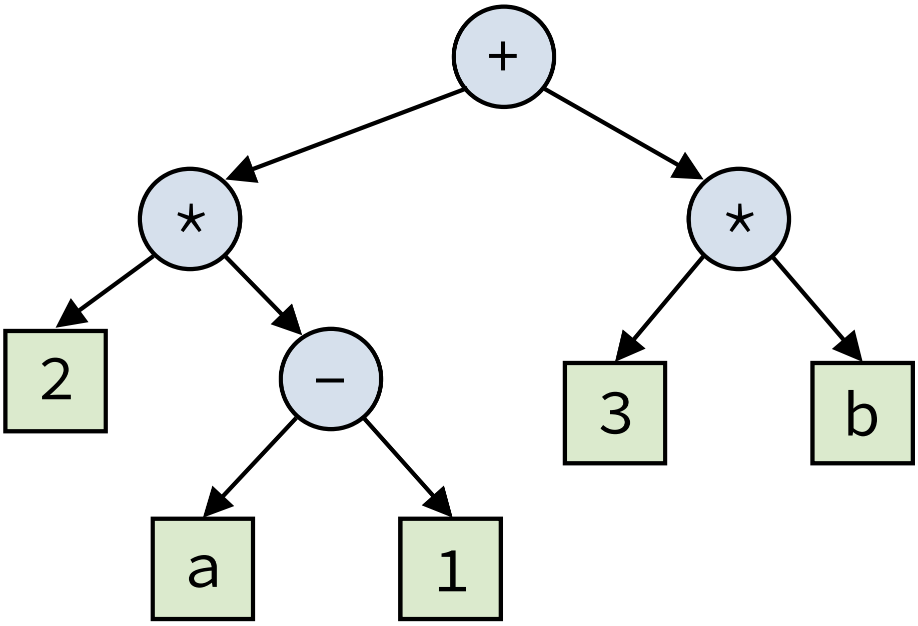


The binary tree itself is just a reference to the ***root node*** plus a bunch of methods!

A ***natural way*** to use binary trees is with **recursion**! Let's implement a few methods in class `BinaryTree`:

- `__len__()` and `len_aux()` - to obtain the number of nodes in the tree
- `preorder()` and `preorder_aux()` - to traverse the tree using ***pre-order***
- `postorder()` and `postorder_aux()` - to traverse the tree using ***post-order***

Example: consider this binary ***expression tree***:

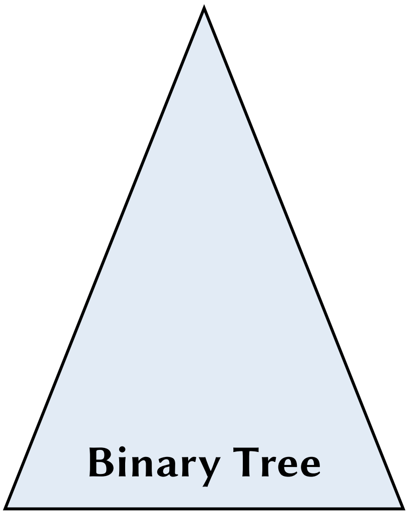


SETU time! :)

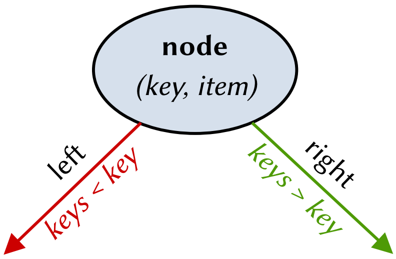
Let's spend a couple of minutes to complete the SETU survey.

Binary Search Trees

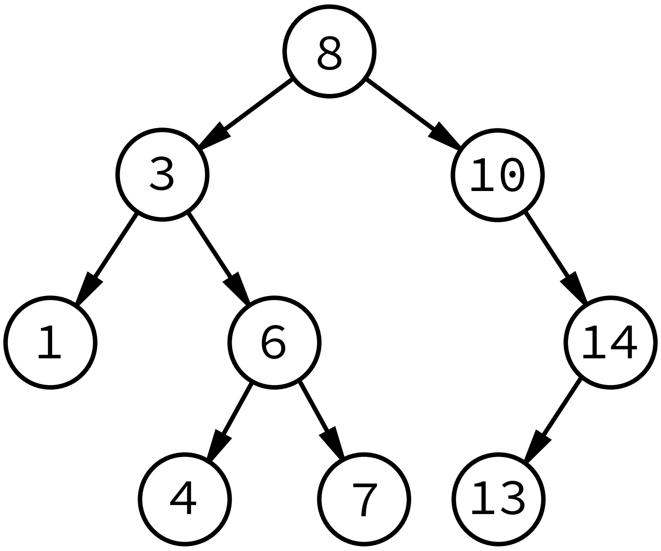
Binary Search Tree =



+ invariant:



✓ **Example** (for simplicity, let's assume that the *keys and the items are the same*)**:**



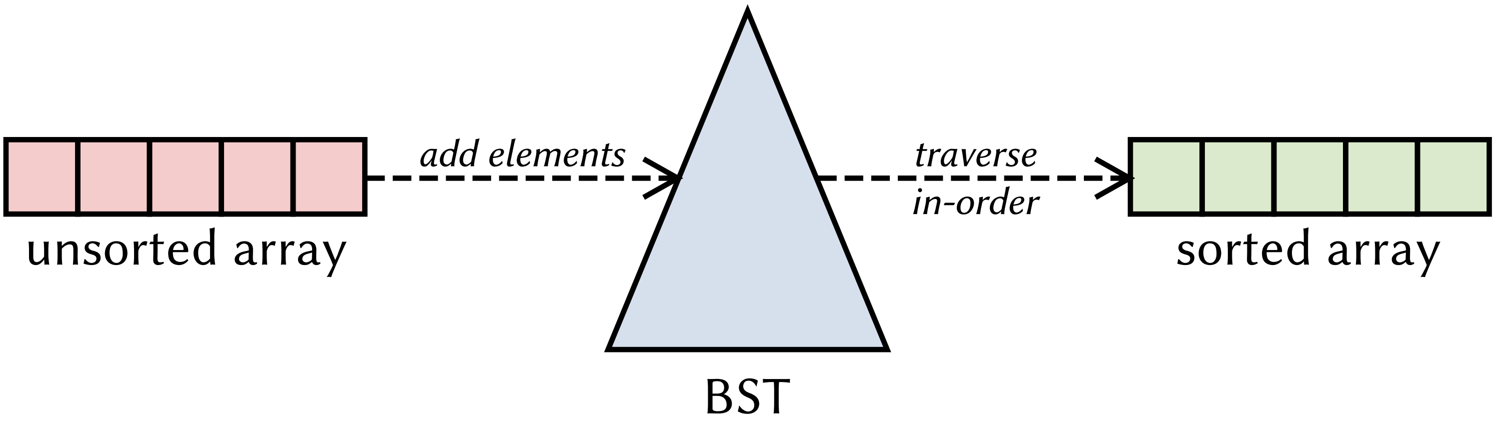
BSTs and TreeSort

Given a class `BinarySearchTree`, let's implement methods:

- `insert_aux()` - for inserting pairs of `(key, item)` into the tree
- `inorder()` and `inorder_aux()` - for traversing the tree ***in-order***

TreeSort algorithm

- let's implement it too!
- assume for now that duplicate array elements are *not allowed*.



Binary Search Trees

Question 1 *Submitted Oct 17th 2022 at 11:28:38 am*

Assuming that the tree has n nodes already, what is the *worst-case* complexity of `insert()`?

○ $\mathcal{O}(1)$

○ $\mathcal{O}(\log n)$

● $\mathcal{O}(n)$

○ $\mathcal{O}(n^2)$

Question 2

What do you think is the worst-case complexity of TreeSort?

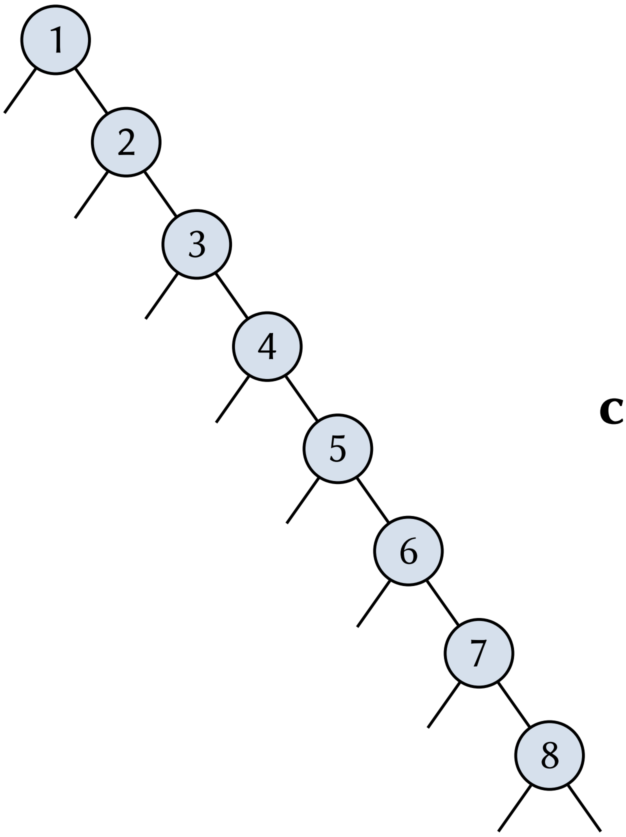
○ Why don't you just tell us?!

○ $\mathcal{O}(\log n)$

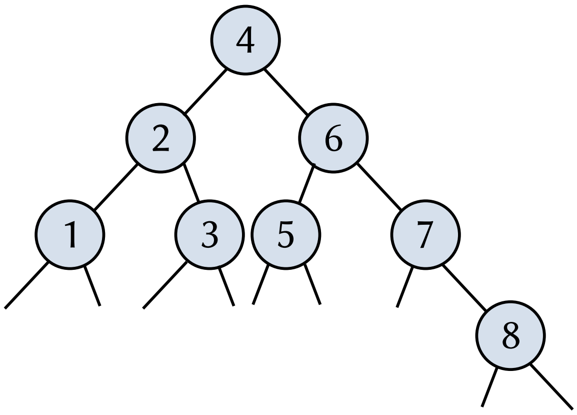
○ $\mathcal{O}(n)$

○ $\mathcal{O}(n^2)$

It's all about balancing!



compare:



If the tree is **balanced**, the worst-case complexity of TreeSort is $\mathcal{O}(n \times \log n)$.

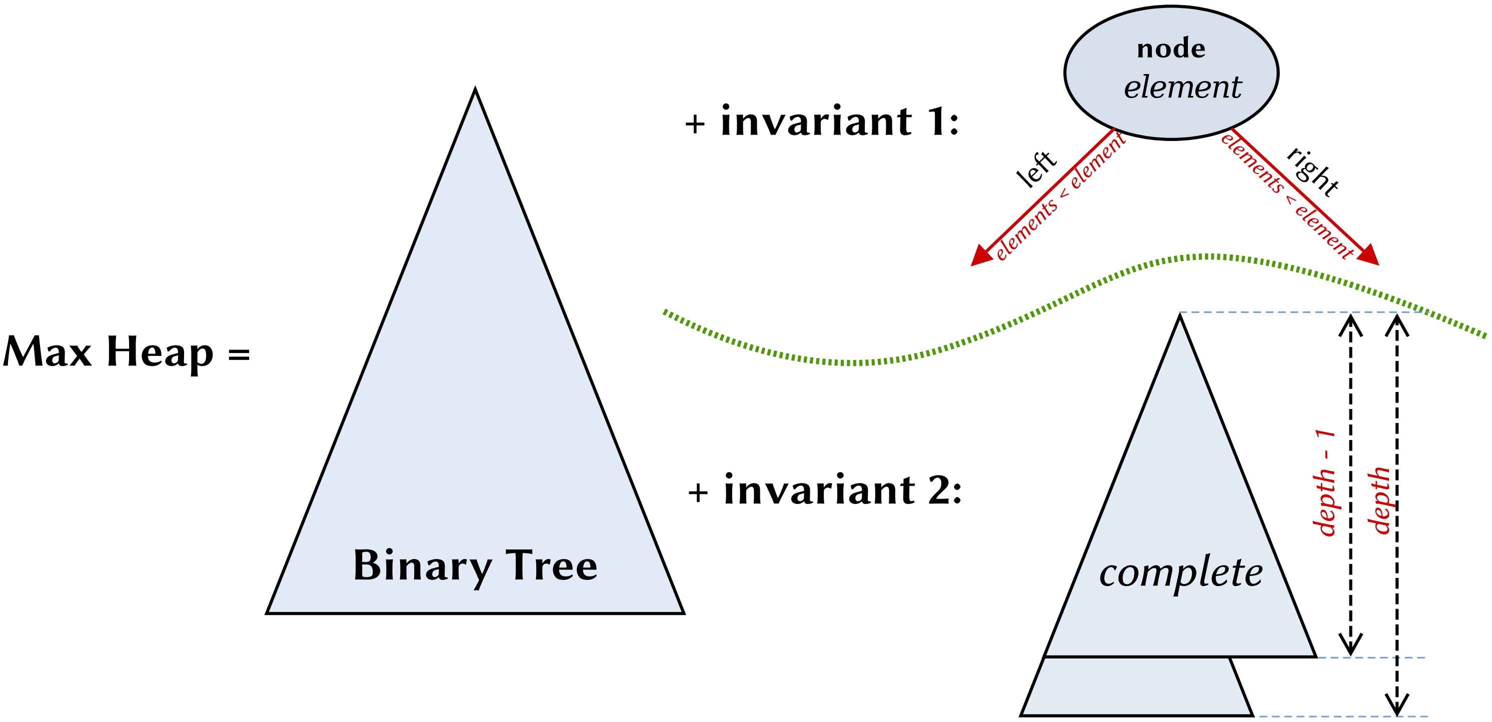
A final remark on balancing…

In practice, to benefit from Binary Search Trees, one has to keep it **balanced**.

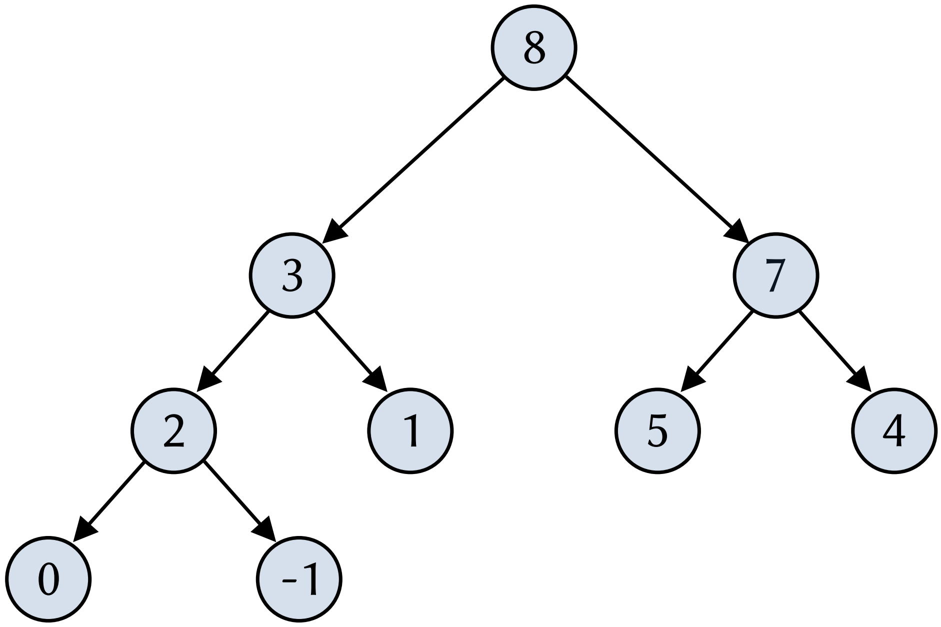
But:

✓ **Good luck with the final assignment! :)**

Max Heaps

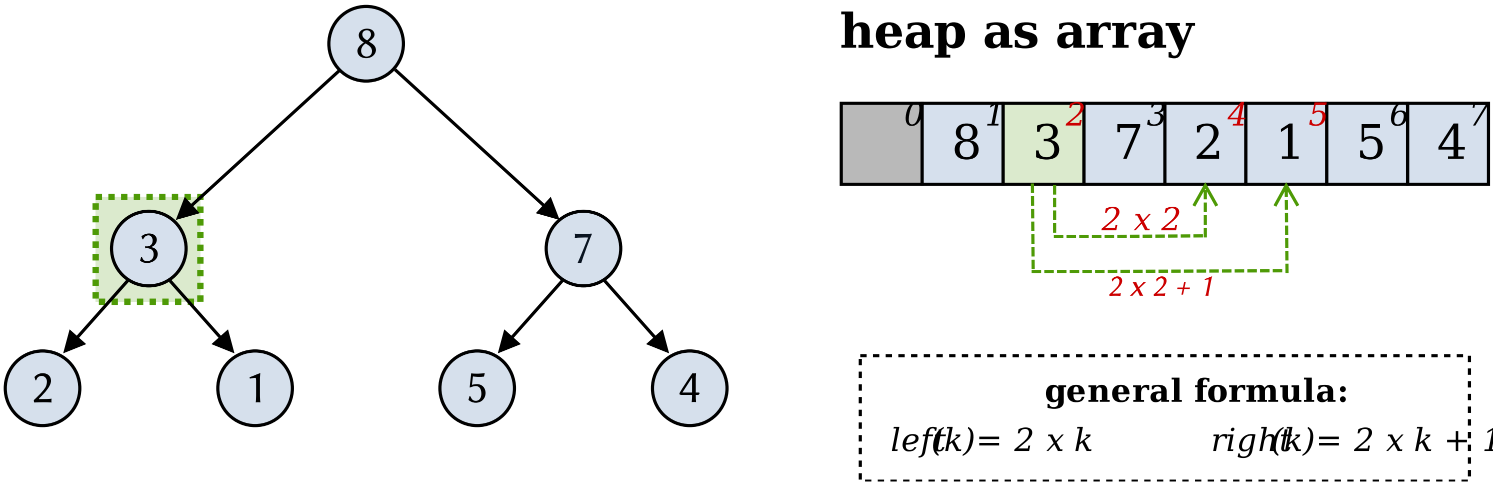


✓ **Example** of a heap *(observe that both invariants hold)***:**



Representing Heaps with Arrays

Here is an example of a heap and its *array representation*:



0	1	2	3	4	5	6	7
	8	3	7	2	1	5	4

Observe (a) that *nothing* is stored at *position 0* and (b) how indices of a node's *children* are computed.

Implementing Heaps

Consider the array-based representation of a heap. When removing the ***top*** of the heap, we want to remove the *leftmost element* of the array. But to maintain completeness of the heap, we actually need to remove an element from the *right*.

So we swap the *rightmost* element to the root, then move the new root element down to its correct location.

Given a partial `Heap` class, let's implement the methods:

- `largest_child(k)` - which returns the index of the maximum-value child of `k`
- `sink(k)` - which moves the element at index `k` downward to its correct position in the heap; this method should use the method `largest_child()`

In this week's lesson, you'll see we can also use `sink` to turn an array into a heap in $\mathcal{O}(Comp \times n)$ operations.

Feedback Form

Weekly Workshop Feedback Form

Question 1

I am enrolled in:

○ FIT1008

○ FIT2085

○ FIT1054

Question 2

What needs improvement?

No response

Question 3

What worked best?

No response

Question 4

How engaged were you by the workshop?

○ ▯▯▯ Very engaged

○ ▯▯▯ Engaged

○ ▯😐▯ Not impressed

○ 😕😴▯ Lost