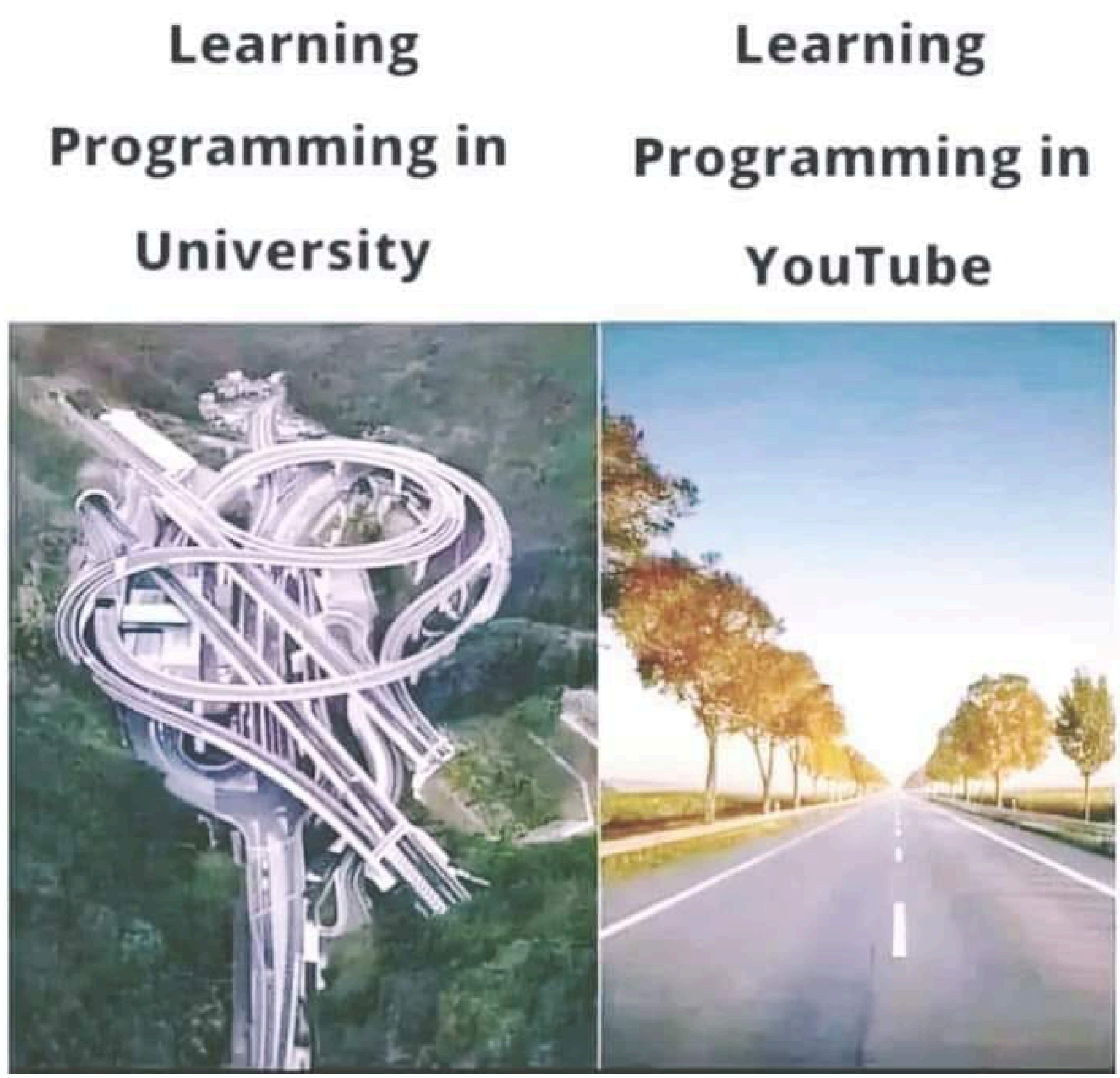
Tree-based sorting

AP



Concept check: Sorting

input: a sequence of integers

output: a reorganisation such that each element will be less than or equal the next

$a \;=\; [5, 0, 2, 11, 18, 11, 6, 36]$

$a.sorted() \;=\; [0, 2, 5, 6, 11, 11, 18, 36]$

"easy to check, not so easy to establish' '

Q: sorting might in fact destroy some information. What might it be?

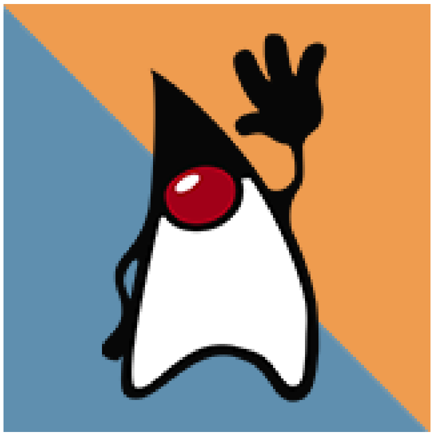
- min, max and median are available in constant time: $a[0]$, $a[n-1]$ and $a[\frac{n}{2}]$, respectively.
- membership can be checked with $\log_2 n$ comparisons at most
- *stability:* multiple copies of the same number should keep their original ordering

$a \;=\; [5, 0, 2, 11', 18, 11'', 6, 36]$

$a.sorted() \;=\; [0, 2, 5, 6, 11', 11'', 18, 36]$

Concept check: sorting in Java

```java
import java.util.Arrays;

int[] MyArray = { 5, 0, 2, 11, 18, 11, 6, 36 };

Arrays.sort(MyArray);

System.out.println(Arrays.toString(MyArray));
```



CC: build your arrays class

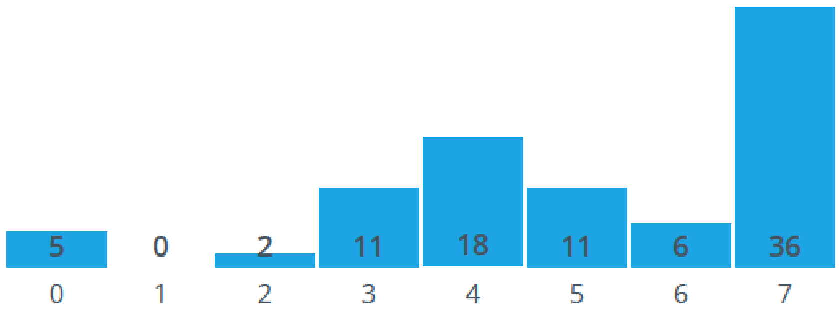
```
public class MyArray {
    private int[] arrayData; // Internal array to store elements
    private int size; // Number of actual elements in the array

    // Constructor to initialize the internal array
    public MyArray(int capacity) {
        arrayData = new int[capacity];
        size = 0;
    }
```

See the class file from last week

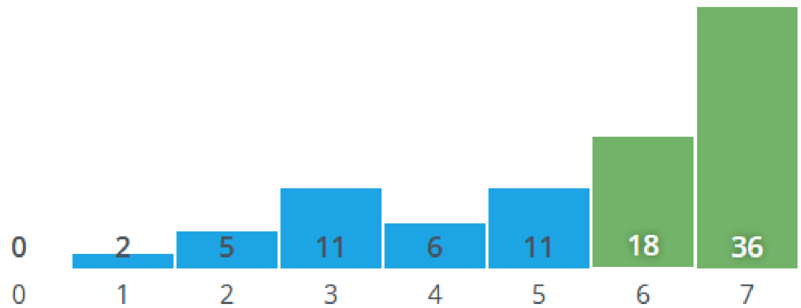
Sorting by pairwise comparison

```
    // Method to sort the array
    public void sort() {
        // Simple implementation of the Bubble Sort algorithm
        for (int i = 0; i < size - 1; i++) {
            for (int j = 0; j < size - i - 1; j++) {
                if (arrayData[j] > arrayData[j + 1]) {
                    // Swap arrayData[j] and arrayData[j+1]
```

```
                    int temp = arrayData[j];
                    arrayData[j] = arrayData[j + 1];
                    arrayData[j + 1] = temp;
                }
            }
        }
    }
```

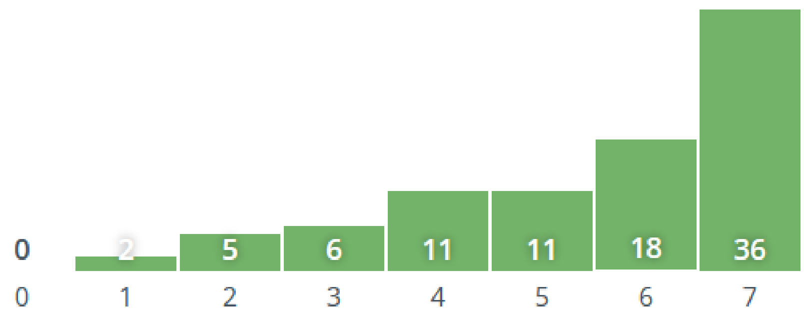


```
    // Method to sort the array
    public void sort() {
        // Simple implementation of the Bubble Sort algorithm
        for (int i = 0; i < size - 1; i++) {
            for (int j = 0; j < size - i - 1; j++) {
                if (arrayData[j] > arrayData[j + 1]) {
                    // Swap arrayData[j] and arrayData[j+1]
                    int temp = arrayData[j];
                    arrayData[j] = arrayData[j + 1];
                    arrayData[j + 1] = temp;
                }
            }
        }
    }
```



- values in green are in their final position
- their array index corresponds to their ranking: the number of less-than-or-equal elements.
- *all blue elements have been seen already and we have ideas about where they will likely end up...*

```
// Method to sort the array
public void sort() {
    // Simple implementation of the Bubble Sort algorithm
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arrayData[j] > arrayData[j + 1]) {
                // Swap arrayData[j] and arrayData[j+1]
                int temp = arrayData[j];
                arrayData[j] = arrayData[j + 1];
                arrayData[j + 1] = temp;
            }
        }
    }
}
```



- only contigous elements will ever be swapped
- all pairwise comparisons are attempted, often several times: is it really needed?
- what if the data is already half-sorted?

```
Sorting often takes place after an update to one or more values destroys the sorted property
So, sorting is called to re-establish the property.
```

$b = [0, 2, 6, 5, 11, 11, 18, 36]$

Cost analysis

- when `i=0` the inner cycle on `j` executes n-1 times,
- then `i=1` and the inner cycle on `j` executes n-2 times, and so on.
- all in all, the innermost code will execute about $\frac{n(n-1)}{2} \approx n^2$ times
- our BubbleSort algorithm won't scale up to web data, log analysis, machine learning etc.
- we need an algorithm that looks at data and only carries out the needed comparisons/swaps.

The tree metaphor

Idea: a data structure that stores values in a way that *represents* what is known about its *rank* in the final version of the sequence.

It will reduce unnecessary comparisons.

The new structure has visual properties that simplify algorithm design and analysis: it's everywhere in computer science.

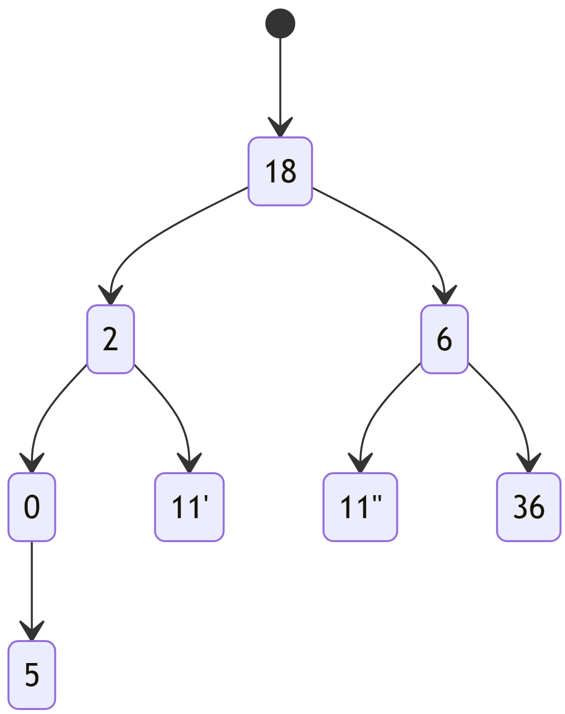
A tree

- a special *root* element which is directly accessible
- each element has access to 0..k elements, called *children*
- siblings are not connected to each other directly
- childless elements are called leaves
- the *height* of the tree is defined as the *longest* root-leaf connection.

A Binary tree

Assumption: let k=2.

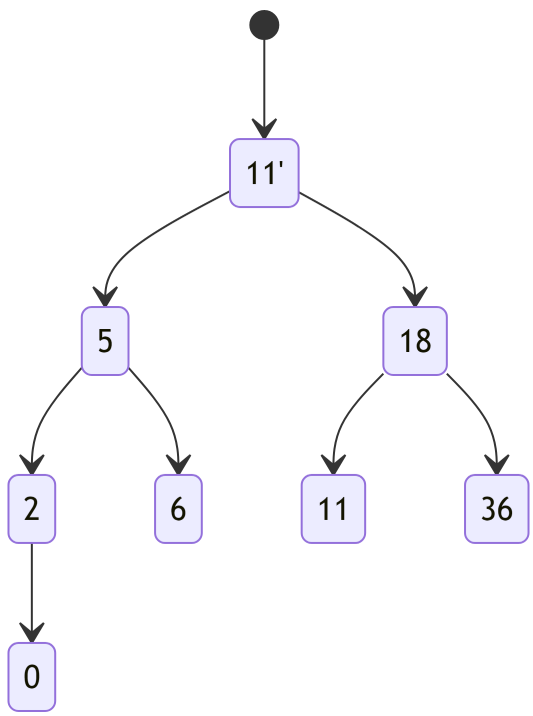
Children elements will be *left* and *right*, resp.



A Binary Search Tree (BST)

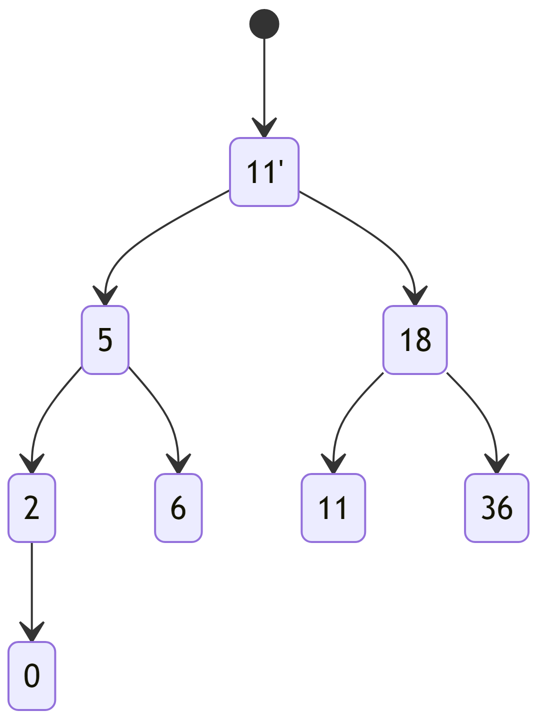
- left children are always less than or equal than their parent

- right children are always greater than their parent.



(here *11* stands for *11''*)

BST position is related to ranking



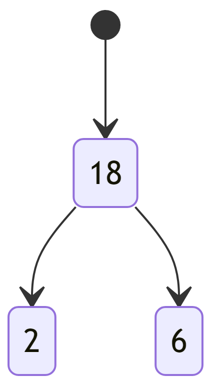
- Q: where are min, max and median elements?
- Q: Can you think of an algorithm that will print out the values in sorted fashion?

Tree as a data structure

- in arrays, each element, say *a[i]* is 'next' to two (at most): *a[i-1]* and *a[i+1]*
- in binary tree, *t[i]* is 'next' to its parent, *t[parent(i)]* and up to two children: *myTree[left(i)]* and *t[right(i)].*
- fact: the binary tree organization can be implemented in RAM with no extra space and minimal time overhead to compute the *parent(), left()* and *right()* functions.
- elegant functions will implement ordered trees and make sort and in general accessing the sequence quick.

BST serialization

The BST is a *view* over an array:




```
a = {18, 2, 6}
```

Assume indexing from 1 Try these functions:

```
left(i) = 2*i
```

```
right(i) = 2*i + 1
```

```
left(i) = int(i/2)
```

Build your class

```
public class MyArray {
    private int[] arrayData; // Internal array to store elements
    private int size; // Number of actual elements in the array
}
```