

Tree-based sorting

AP

**Learning
Programming in
University**



**Learning
Programming in
YouTube**



- Mock lecture on Tree-based sorting
- I will explain my pedagogy as I go through

Objectives

We continue work on the sorting problem

We start learning a new, powerful data structure: trees

Trees in CS were developed rather early to manage *scarce* dynamic memory

The grow *downward*

Concept Checking

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 about $\log_2 n$ comparisons
 - *stability*: multiple copies of the same number should keep their original ordering

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

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

Concept check: sorting in 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

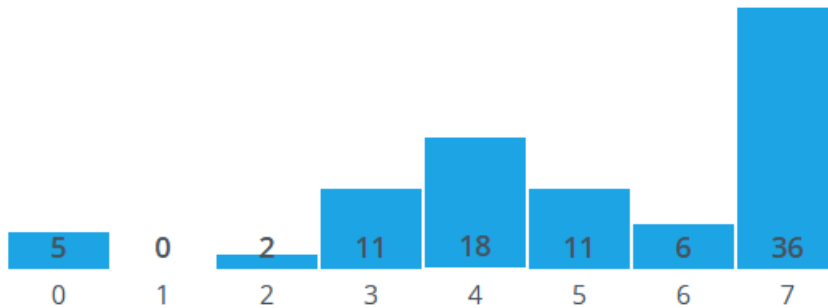
    // Constructor to initialize the internal array
    // capacity is the maximum allowed number of elements
    public MyArray(int capacity) {
        arrayData = new int[capacity];
        size = 0;
    }
}
```

See the [class file](#) from last week

Sorting

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;
            }
        }
    }
}
```

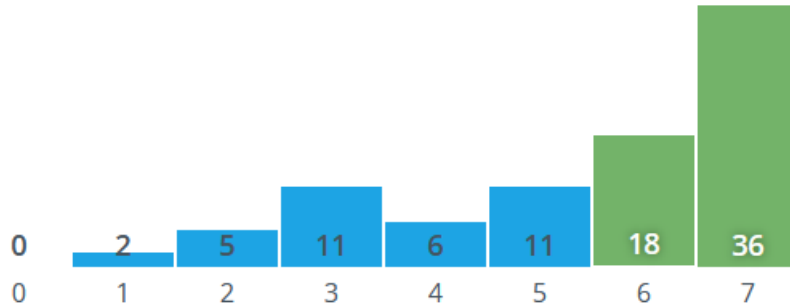


```
// Method to sort the array
public void sort() {
    // Simple implementation of the Bubble Sort algorithm
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arrayData[j] > arrayData[j + 1]) {
                // Swap arrayData[j] and arrayData[j+1]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

```

        arrayData[j] = arrayData[j + 1];
        arrayData[j + 1] = temp;
    ...

```

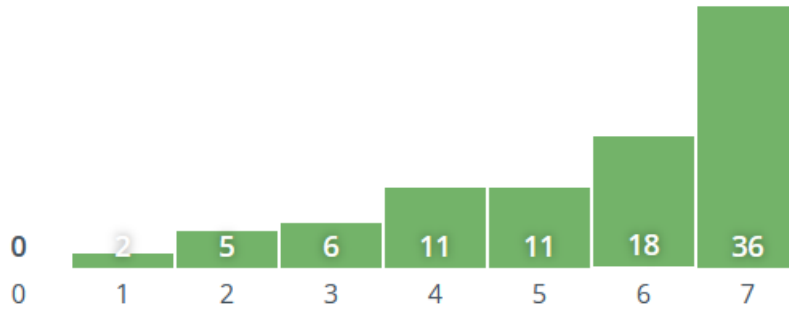


- values in green are in their final position
- *all blue elements have been seen already and we have ideas about where they will likely end up...*

```

// 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;
            }
        }
    }
}

```



- green elem. have indices corresponding to their ranking: the no. of \leq elements.

-
- only contiguous 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 of the array. So, sorting is called to re-establish the property.

Example:

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

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 seek algorithms that *look at data* and carry out only as many comparisons/swaps as needed.

The tree abstraction

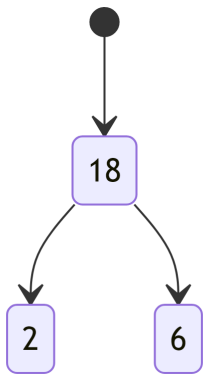
Idea

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.

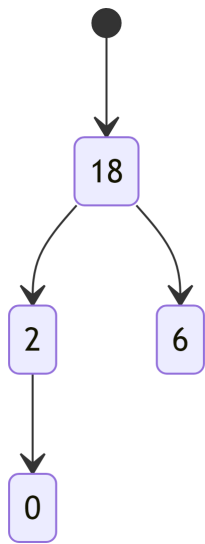
The new structure has visual properties that simplify algorithm design and analysis: it's everywhere in CS!

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

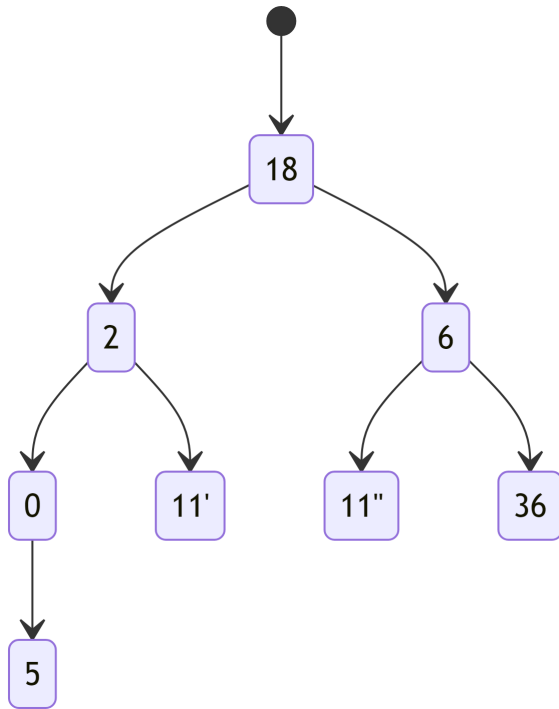
A tree, cont'd



- childless elements are called leaves
- the *height* of the tree is defined as the *longest* root-to-leaf path.

A Binary tree: $k=2$.

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

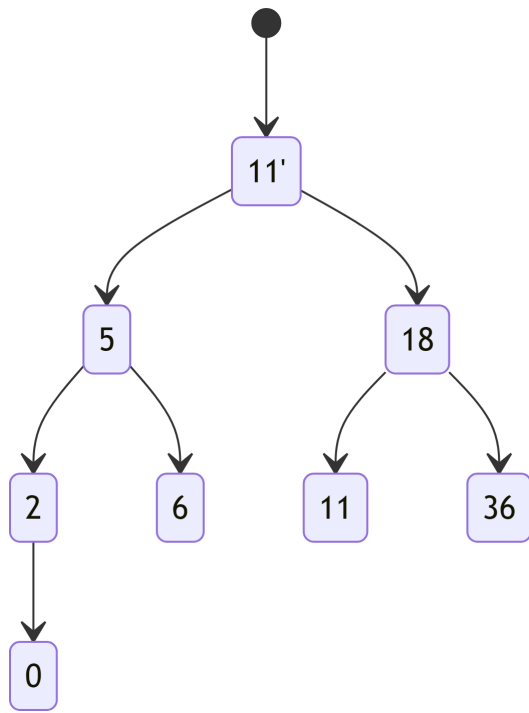


- Complete left-to-right: a BT with no ‘holes’
- never a right leaf node without its left sibling.

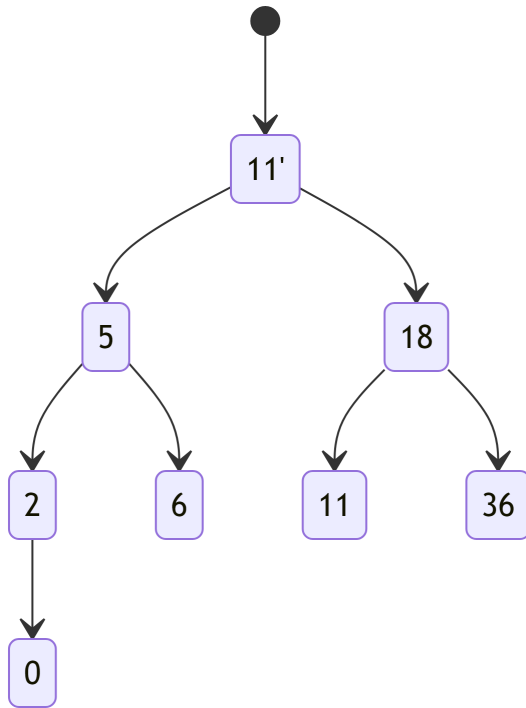
A Binary Search Tree (BST)

At each level

- the left child never exceed (\leq) its parent
- the right child always does



Position on the BST relates to ranking



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

Rethinking arrays

- each element, say $a[i]$ is *next* to two (at most): $a[i-1]$ and $a[i+1]$
- real physical memory or the simplest of abstractions?

Bin. trees as a data structure

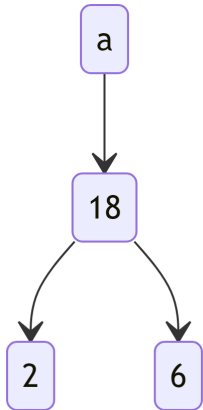
each `tree[i]` element is *next* to its parent, `tree[parent(i)]` and one or two children: `tree[left(i)]` and `tree[right(i)]`.

- Fact: complete BTs 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 BTs/BSTs: sorting accelerates!

BST as abstraction over RAM

The BST is a *view* of, essentially, an array:

```
int[] a = {18, 2, 6};
```



Assume indexing from 1 and try these functions:

```
int left(int i) {return 2*i;}  
int left(int i) {return 2*i+1;}  
int parent(int i) {return (int) i/2;}
```

In depth: tree navigation

Concept checking: tree navigation

$\text{left}(1) \rightarrow 2$

$\text{right}(1) \rightarrow 3$

$\text{left}(2) \rightarrow 4$

$\text{right}(2) \rightarrow 5$

$\text{left}(3) \rightarrow 6$

- $\text{left}(52) \rightarrow ?$
- $\text{right}(52) \rightarrow ?$

- `parent(52) → ?`

efficiency

- Thanks to binary representation, division/multiplication by 2 can be done in one CPU cycle

```
// implements left()
byte originalByte = 0b0011_0100; // 52 in binary

int shiftedByte = originalByte << 1; // Shift left by 1 positions
```

- visiting a complete BST is very efficient!

```
// implents parent()
byte originalByte = 0b0011_0100; // 52 in binary

int shiftedByte = originalByte >> 1; // Shift right by 1 positions
```

We now leave the idealised vision of complete BST to look at *general* BST whose shape could be *irregular*.

In Depth: The BST class

BST as a class, I

```
// Define a class for the nodes of the tree
class Node {
    int value;
    Node left, right;

    public Node(int item) {
        value = item;
        // at 'birth,' nodes are childless
        left = right = null;
    }
}
```

BST as a class, II

```
// Define the Binary Search Tree (BST) class
class BinarySearchTree {
    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {
        root = null;
    }
}
```

BST as a class, III

```
// Method to insert a new key
void insert(int value) {
    root = insertRec(root, value);
}

// Recursive insert function
Node insertRec(Node root, int value) {
    // If the tree is empty, return a new node
    if (root == null) {
        root = new Node(value);
        return root;
    }

    // Otherwise, recur down the tree
    if (value < root.value)
        root.left = insertRec(root.left, value);
    else if (value > root.value)
        root.right = insertRec(root.right, value);

    // Return the (unchanged) node pointer
    return root;
}
```

My first BST

```
// Main method to test the BinarySearchTree class
public static void main(String[] args) {
    BinarySearchTree bst = new BinarySearchTree();

    // Insert values into BST
    bst.insert(11);
    bst.insert(5);
    bst.insert(18);
    bst.insert(2);
    bst.insert(6);
    bst.insert(11);
    bst.insert(36);
    bst.insert(0);

    // Print the inorder traversal of the BST
    System.out.println("Inorder traversal of the BST:");
    bst.inorder();
}
}
```

Live Coding

Tree transversal

Given a reference to the root node, print its content in ascending order.

Exploit the recursive BST property: all sub-trees are BST themselves.

```
// Method to conduct inorder traversal of the tree
void inorder() {
    inorderRec(root);
}

// Visit the BT and print out the values
// in ascending order
```

```

// Method to conduct inorder traversal of the tree
void inorder() {
    inorderRec(root);
}

// Recursive function for inorder traversal
void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.print(root.value + " ");
        inorderRec(root.right);
    }
}

```

Analysis

Good properties

- in-order transversal of the BST corresponds to sorting.
- if the BST is *balanced*: no. of left successors and no. of right successor is roughly equal:
- height, i.e., the longest root-to-leaf possible visit, is going to be about $\log_2 n$
- finding max and min will require only $\log_2 n$ accesses.
- in general, we can find the element of a given rank with only $\log_2 n$ accesses.

Bad properties

- if the BST is *unbalanced*, it could end up, e.g., with all left successors and no right successor
- finding max or max would then take n accesses: no better than with an unsorted array.



Conclusions

Sorting arrays with BubbleSort: easy to visualise+implement but costly

Many comparisons are repeated: waste of time, costs grow with n^2

Arrays: a straightforward abstraction of RAM

- Tree: a new abstraction, relatively easy to code and lightweight
- Binary Search Trees: a type of tree that makes values easy to sort/search
- However BST may degenerate into (costly) *straight lists*
- We need techniques to turn arbitrary sequences into balanced BST