ECE430.217 Data Structures

Binary Trees

Weiss Book Chapter 4.2/4.3

Byoungyoung Lee

https://compsec.snu.ac.kr

byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

In this talk, we will look at the binary tree data structure:

- Definition
- Properties
- A few applications
  - Ropes (strings)
  - Expression trees

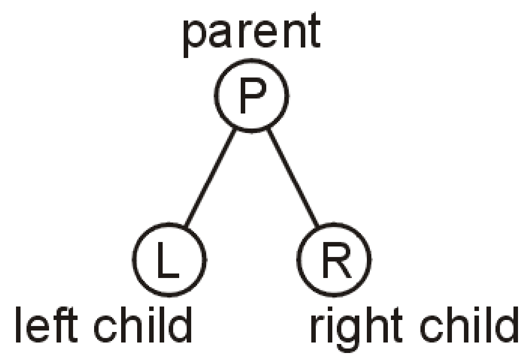
Definition

A binary tree is a restriction where each node has exactly two children:

- Each child is either

  i) empty or ii) another binary tree
- Each child is labeled as either

  i) *left* or ii) *right* subtrees

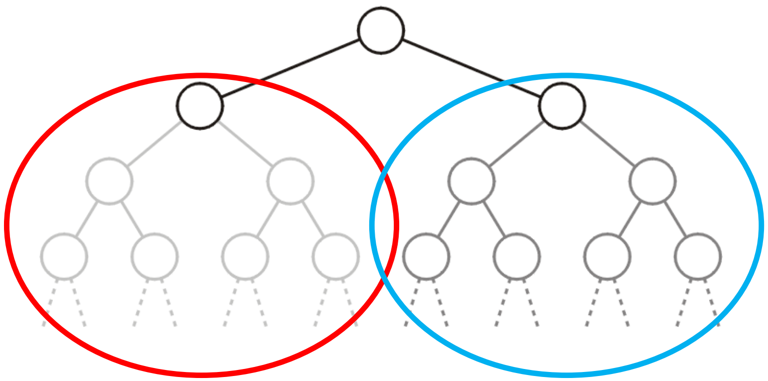


At this point, recall that $\lg(n) = \Theta(\log_b(n))$ for any b

Definition

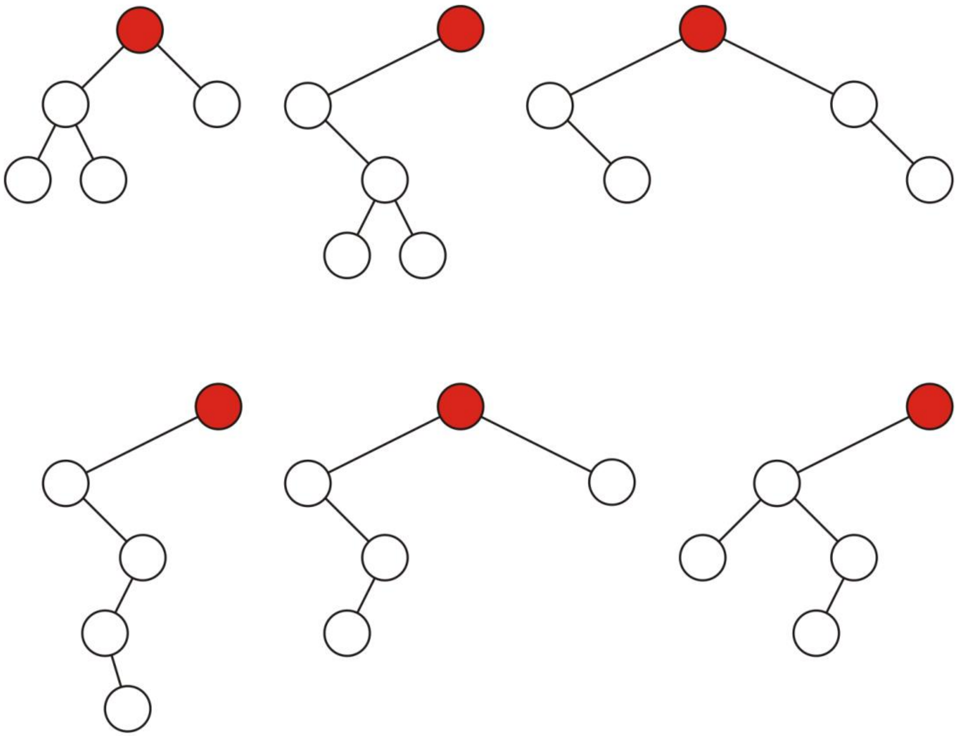
We will also refer to the two sub-trees as

- The left-hand sub-tree, and
- The right-hand sub-tree



Definition

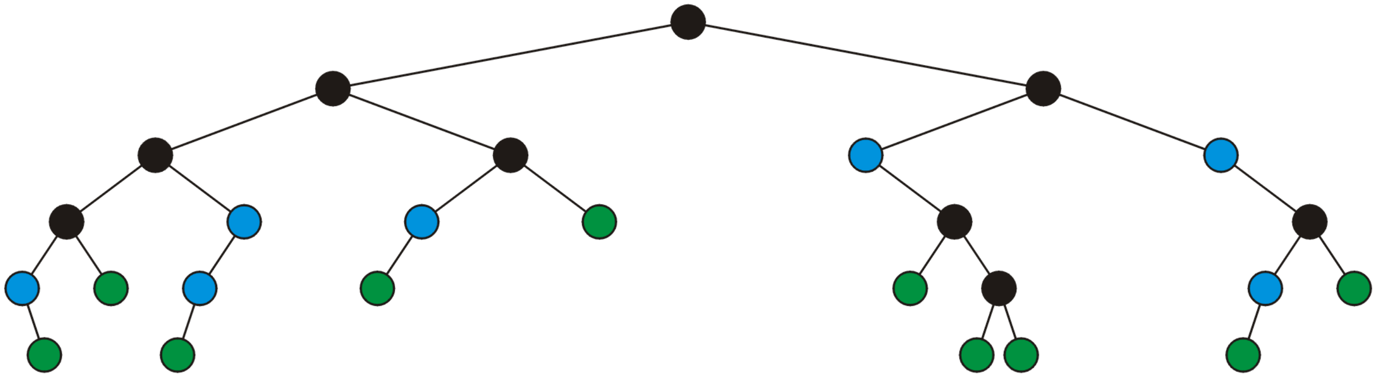
Sample variations on binary trees with five nodes:



Q. How to count the number of all possible variations?

Definition

A *full* node is a node where both sub-trees are non-empty



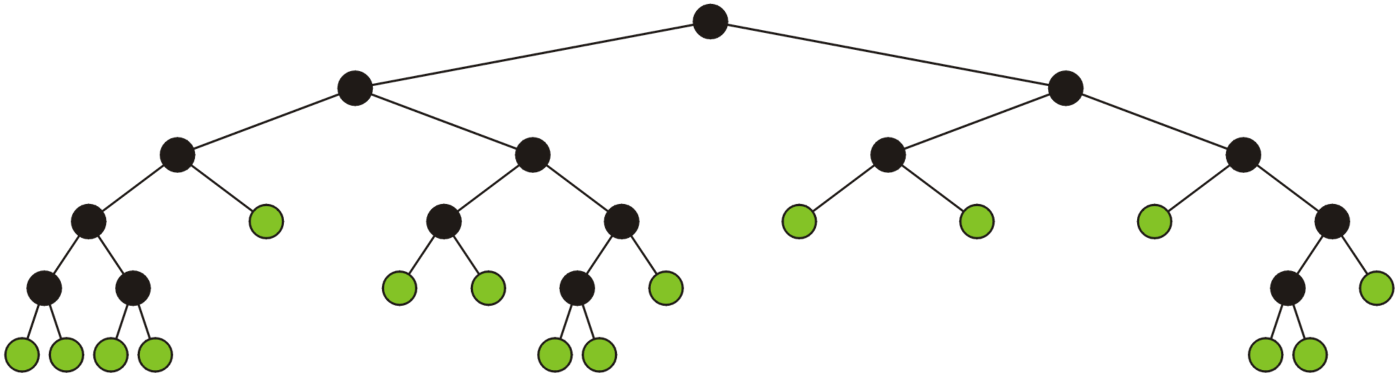
Legend:

full nodes neither leaf nodes

Definition

A *full binary tree* is where each node is either:

- A full node, or
- A leaf node



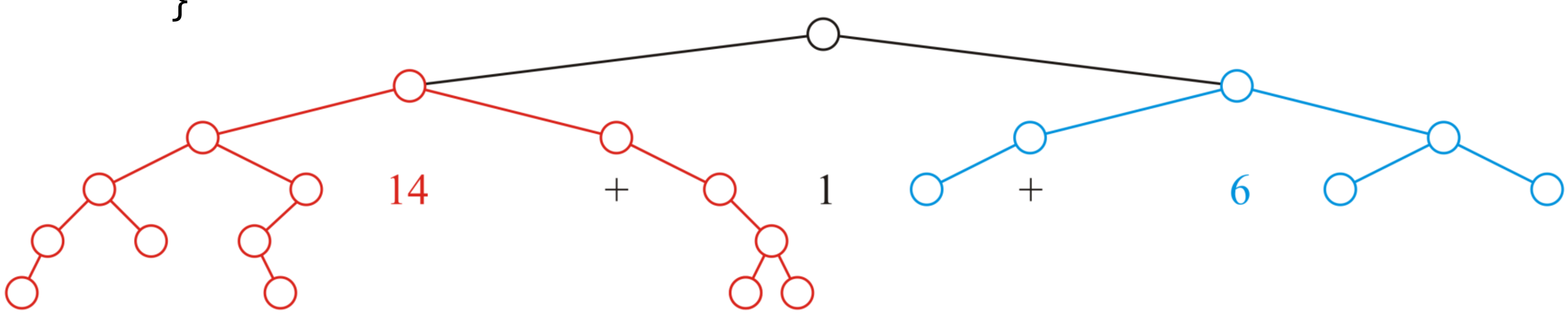
These have applications in

- Expression trees
- Huffman encoding

Size

The recursive size function runs in Q1 time and Q2 memory

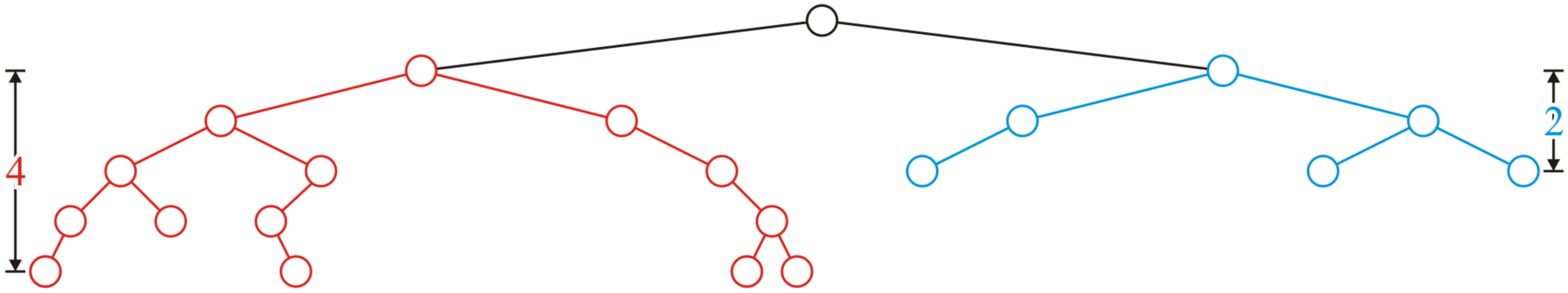
```
template <typename Type>
int Binary_node<Type>::size() const {
    if ( left == nullptr ) {
        return ( right == nullptr ) ? 1 : 1 + right->size();
    } else {
        return ( right == nullptr ) ?
            1 + left->size() :
            1 + left->size() + right()->size();
}
```



Height

The recursive height function also runs in $\Theta(n)$ time and $\Theta(h)$ memory

```
int Binary_node<Type>::height() const {
    if ( left == nullptr ) {
        return ( right == nullptr ) ? 0 : 1 + right->height();
    } else {
        return ( right == nullptr ) ?
              1 + left->height() :
              1 + left->height() + right->height();
}
```



Run Times

Recall that with linked lists and arrays, some operations would run in $\Theta(n)$ time

The run times of operations on binary trees, we will see, depends on **the height of the tree**

Q. What's the height of the tree for the number of nodes "n"?

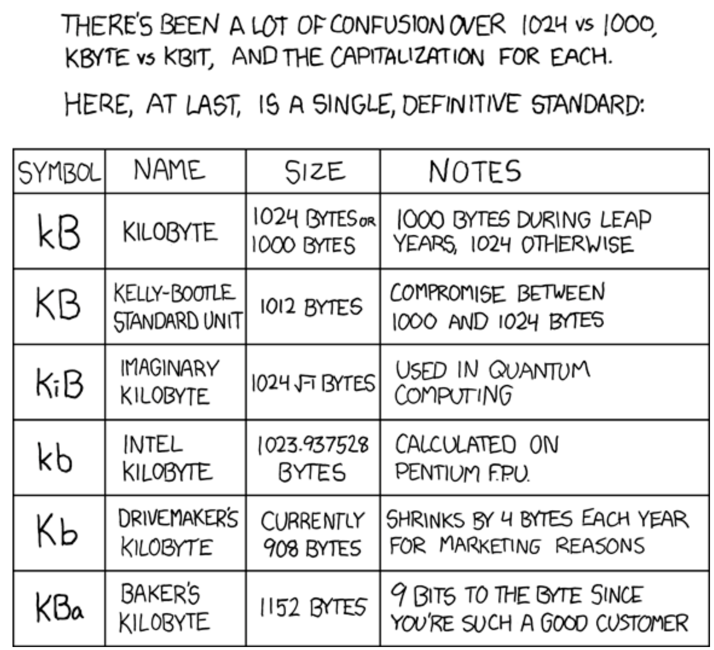
- Worst case: Q1
- Average case: Q2
- Best case: Q3

Run Times

If we can achieve and maintain a height $\Theta(\lg(n))$, many operations of the binary tree can run in $\Theta(\lg(n))$ we

Logarithmic time is fairly good compared to constant time:

$\lg(\ 1000\ ) \approx 10$	kB
$\lg(\ 1\ 000\ 000\ ) \approx 20$	MB
$\lg(\ 1\ 000\ 000\ 000\ ) \approx 30$	GB
$\lg(\ 1\ 000\ 000\ 000\ 000\ ) \approx 40$	TB
$\lg(\ 1000^n\ ) \approx 10\ n$	



http://xkcd.com/394/

Application: Ropes

In 1995, Boehm *et al*. introduced the idea of a rope, or a *heavyweight* string

Application: Ropes

Alpha-numeric data is stored using a *string* of characters

- A character (or `char`) is a numeric value from 0 to 255 where certain numbers represent certain letters
- ASCII code: http://www.asciitable.com/

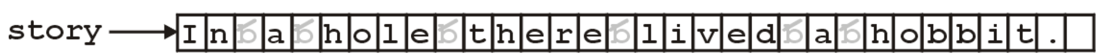
For example,

'A'	65	01000001_2
'B'	66	01000010_2
'a'	97	01100001_2
'b'	98	01100010_2
' '	32	00100000_2

Application: Ropes

A C-style string is an array of characters followed by the character with a numeric value of `0`

```
char * story = "In a hole there lived a hobbit.";
```



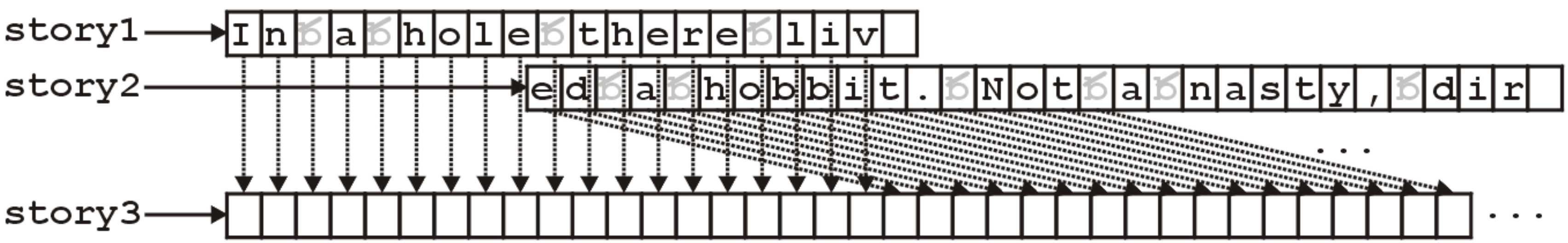
On problem with using arrays is the runtime required to concatenate two strings

Application: Ropes

Concatenating two strings requires the operations of:

- Allocating more memory, and
- Coping both strings $\Theta(n + m)$

```
char * story1 = "In a hole there liv";
char * story2 = "ed a hobbit. Not a nasty, di";
```

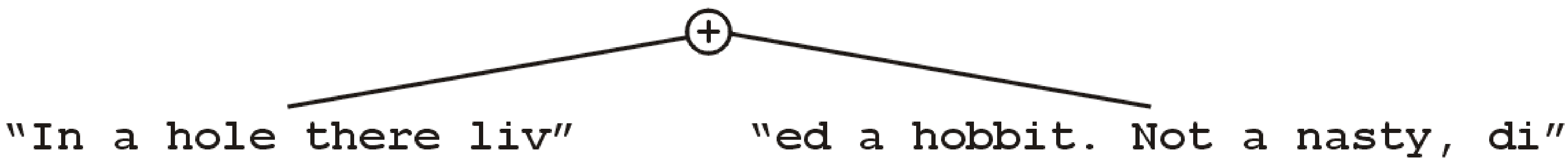


Application: Ropes

The rope data structure:

- Stores strings in the leaves,
- Internal nodes (full) represent the concatenation of the two strings, and
- Represents the string with the right sub-tree concatenated onto the end of the left

The previous concatenation may now occur in $\Theta(1)$ time

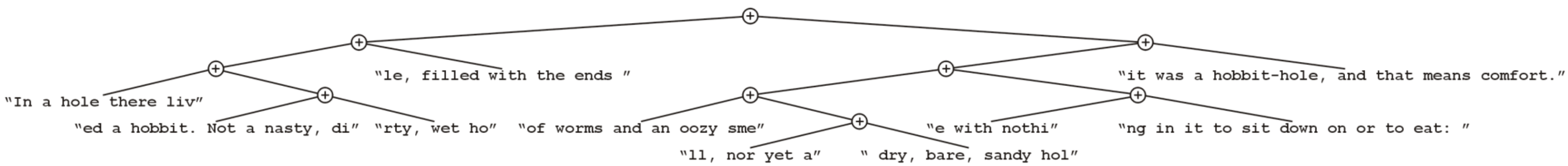


Application: Ropes

The string

"In a hole there lived a hobbit. Not a nasty, dirty, wet hole,
filled with the ends of worms and an oozy smell, nor yet a dry, bare,
sandy hole with nothing in it to sit down on or to eat: it was a
hobbit-hole, and that means comfort."

may be represented using the rope

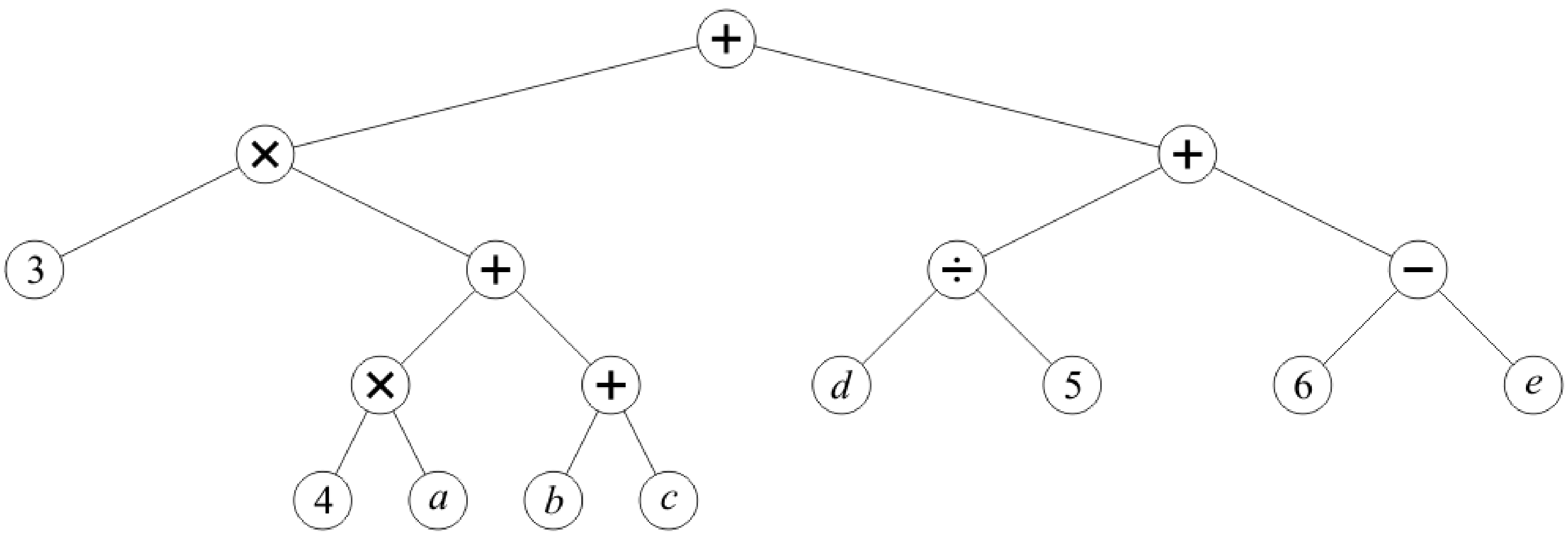


References: http://en.wikipedia.org/wiki/Rope_(computer_science)
J.R.R. Tolkien, *The Hobbit*

Application: Expression Trees

Any basic mathematical expression containing binary operators may be represented using a binary tree

For example, $3(4a + b + c) + d/5 + (6 - e)$



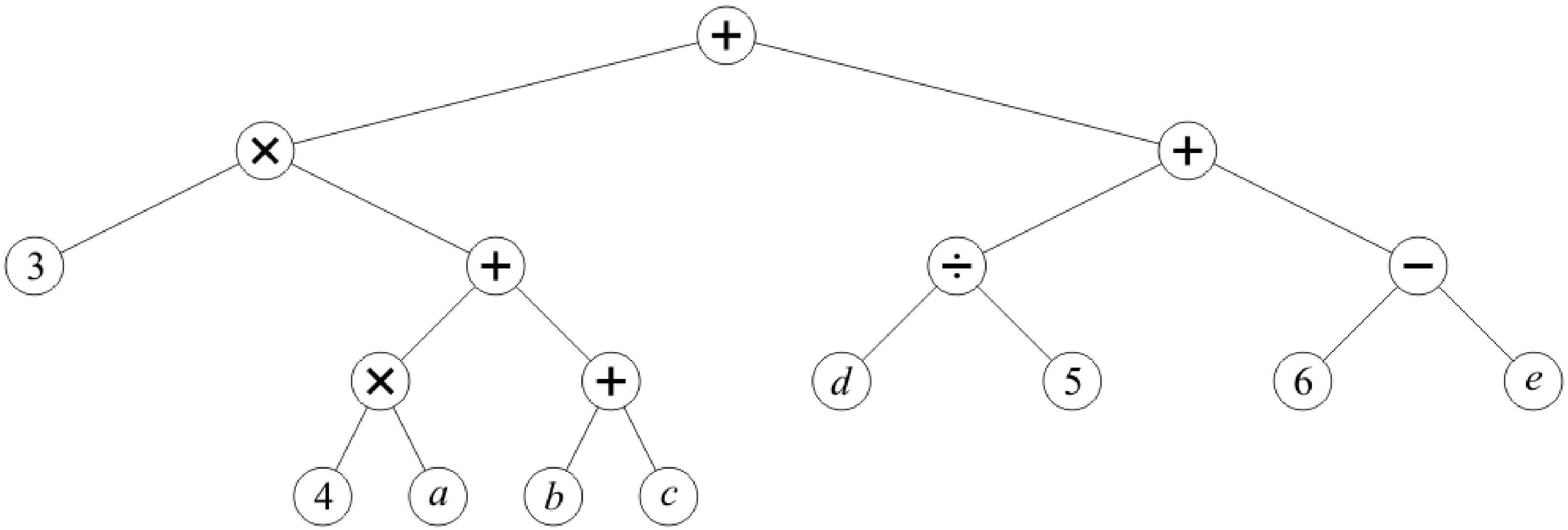
Application: Expression Trees

Observations:

- Internal nodes store operators
- Leaf nodes store literals or variables
- No nodes have just one subtree: either no subtree or two subtress
- The order is not relevant for
  - Addition and multiplication (commutative)
- Order is relevant for
  - Subtraction and division (non-commutative)

Application: Expression Trees

A Q1 **-order** Q2 **traversal** converts such a tree to the reverse-Polish format



$$3 \;\; 4 \;\; a \;\; \times \;\; b \;\; c \;\; + \;\; + \;\; \times \;\; d \;\; 5 \;\; \div \;\; 6 \;\; e \;\; - \;\; + \;\; +$$

Application: Expression Trees

Humans think in in-order

Computers think in post-order:

- Both operands must be loaded into registers
- The operation is then called on those registers

Most programming languages use in-order notation (C, C++, Python, Java, C#, etc.)

- Necessary to translate in-order into post-order

Summary

In this talk, we introduced binary trees
- Each node has two distinct and identifiable sub-trees
- Either sub-tree may optionally be empty
- The sub-trees are ordered relative to the other

We looked at:
- Properties
- Applications