Binary trees

Outline

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

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

Definition

The arbitrary number of children in general trees is often unnecessary—many real-life trees are restricted to two branches

- Expression trees using binary operators
- Phylogenetic trees
- encoding algorithms

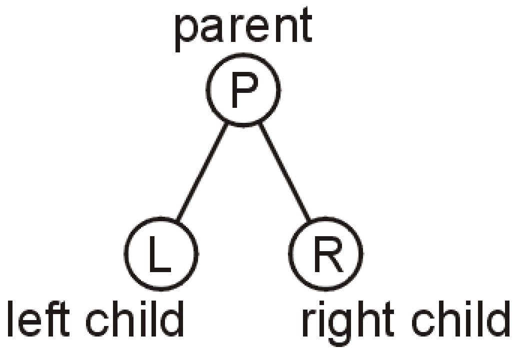
There are also issues with general trees:

- There is no natural order between a node and its children

Definition

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

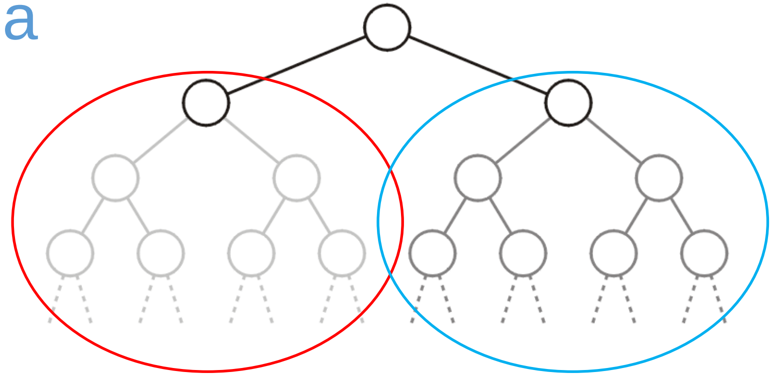
- Each child is either empty or another binary tree
- This restriction allows us to label the children as *left* and *right* subtrees



Definition

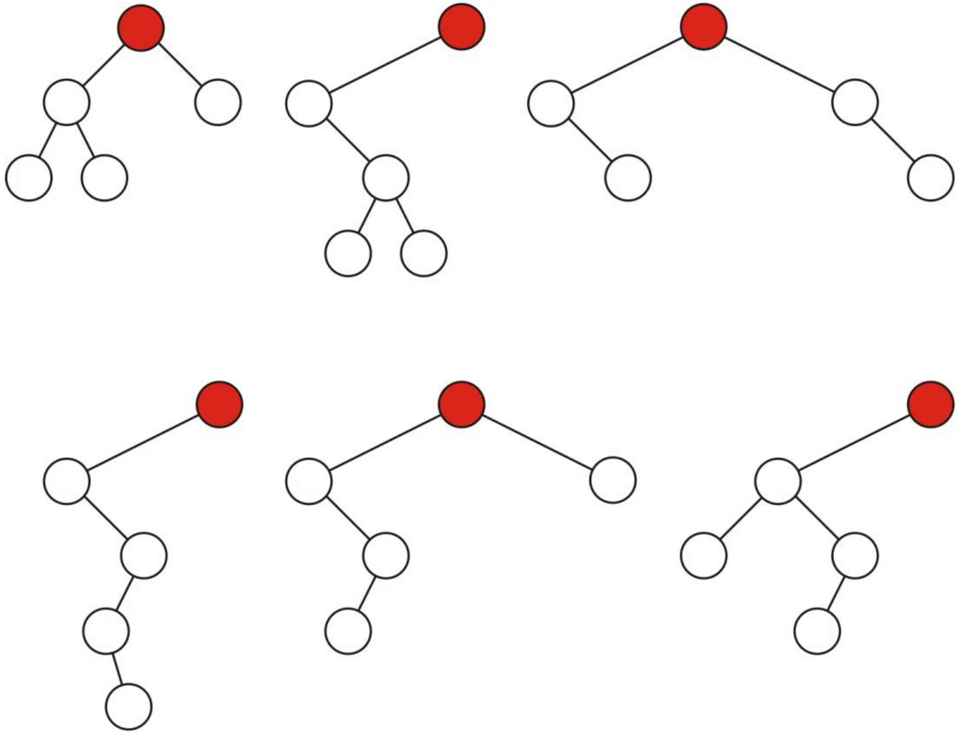
We will also refer to the two sub-trees as

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



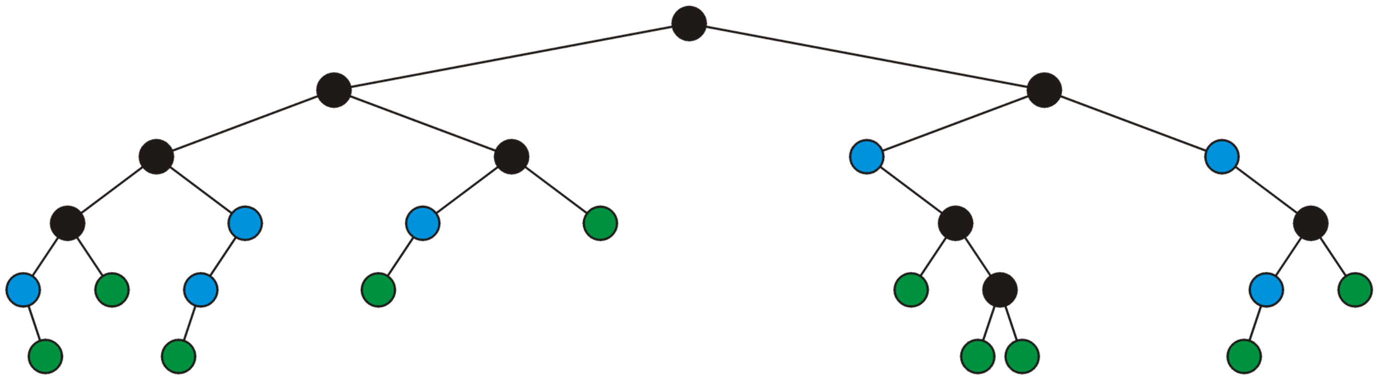
Definition

Sample variations on binary trees with five nodes:



Definition

A *full* node is a node where both the left and right sub-trees are non-empty trees

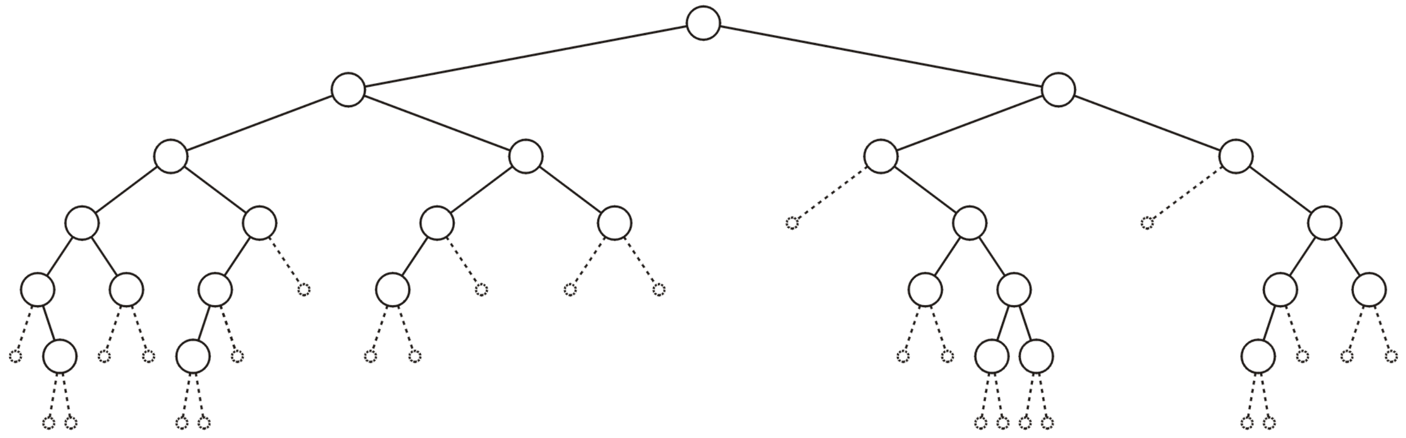


Legend:

full nodes neither leaf nodes

Definition

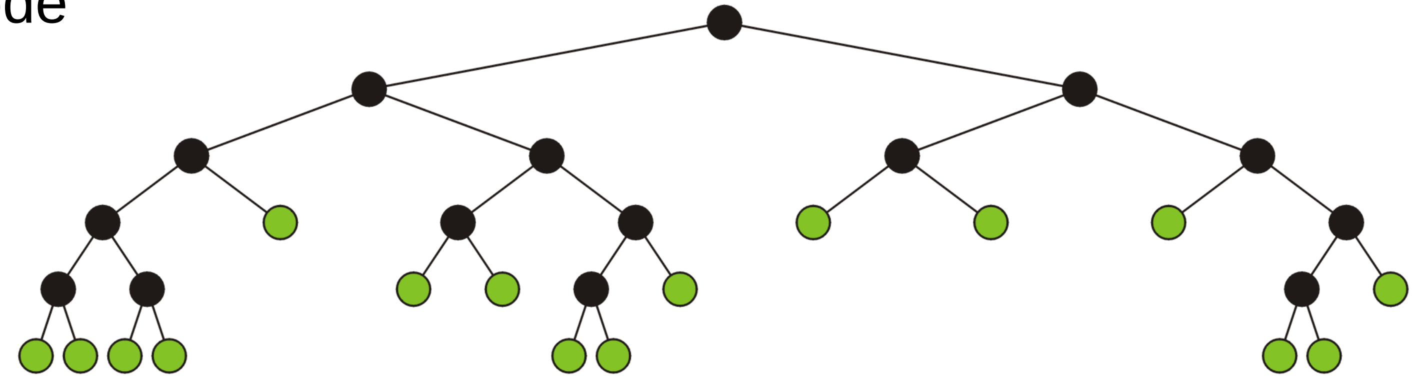
An *empty node* or a *null sub-tree* is any location where a new leaf node could be appended



Definition

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

- A full node, or
- A leaf node



These have applications in

- Expression trees
- Huffman encoding

Binary Node Class

The binary node class is similar to the single node class:

```
template <typename Type>
class Binary_node {
    protected:
        Type node_value;
        Binary_node *p_left_tree;
        Binary_node *p_right_tree;

    public:
        Binary_node( Type const & );

        Type value() const;
        Binary_node *left() const;
        Binary_node *right() const;

        bool is_leaf() const;
        int size() const;
        int height() const;
        void clear();
}
```

Binary Node Class

We will usually only construct new leaf nodes

```
template <typename Type>
Binary_node<Type>::Binary_node( Type const &obj ):
node_value( obj ),
p_left_tree( nullptr ),
p_right_tree( nullptr ) {
    // Empty constructor
}
```

Binary Node Class

The accessors are similar to that of `Single_list`

```
template <typename Type>
Type Binary_node<Type>::value() const {
    return node_value;
}

template <typename Type>
Binary_node<Type> *Binary_node<Type>::left() const {
    return p_left_tree;
}

template <typename Type>
Binary_node<Type> *Binary_node<Type>::right() const {
    return p_right_tree;
}
```

Binary Node Class

Much of the basic functionality is very similar to `Simple_tree`

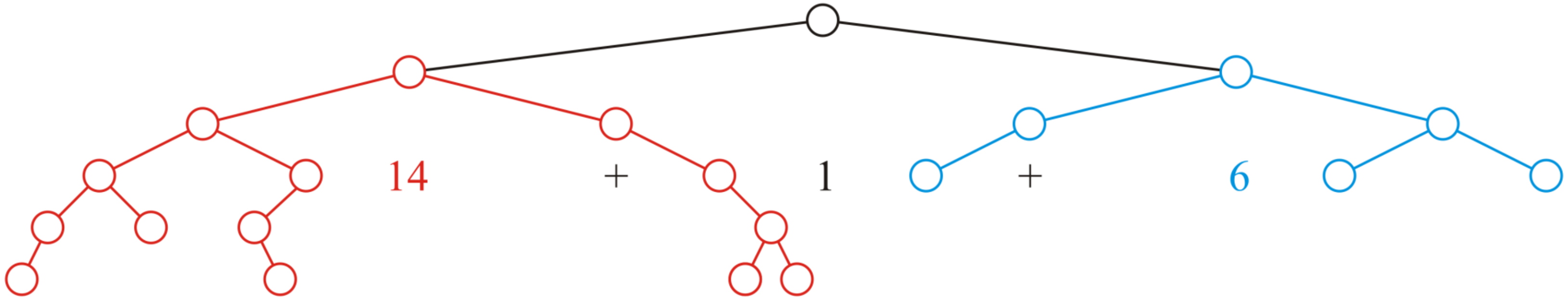
```
template <typename Type>
bool Binary_node<Type>::is_leaf() const {
    return (left() == nullptr) && (right() == nullptr);
}
```

Size

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

- These can be implemented to run in $\Theta(1)$

```
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();
}
```

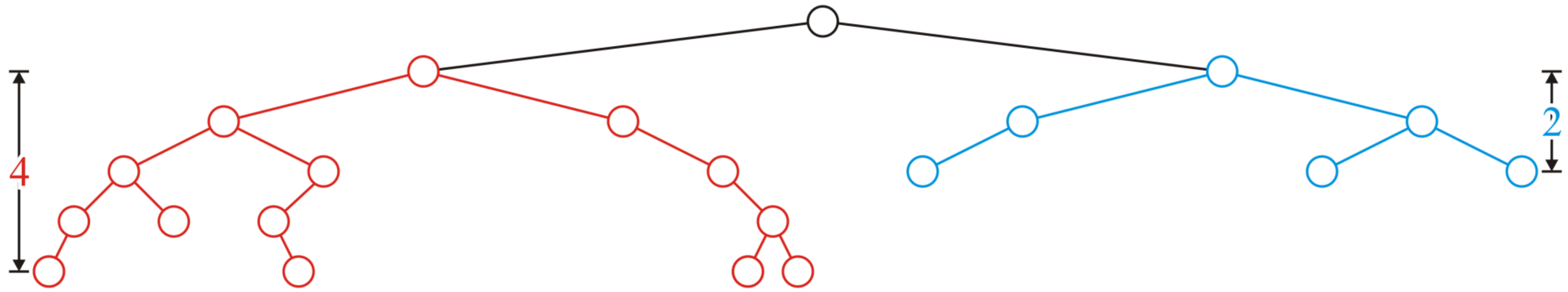


Height

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

- Later we will implement this in $\Theta(1)$ time

```
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();
}
```



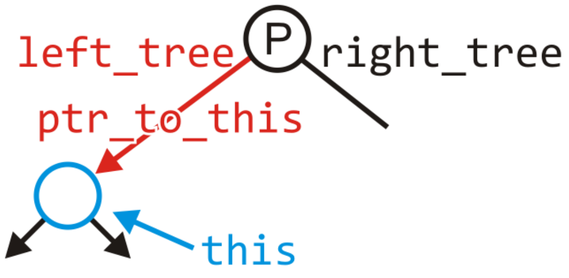
Clear

Removing all the nodes in a tree is similarly recursive:

```
template <typename Type>
void Binary_node<Type>::clear( Binary_node *&p_to_this ) {
    if ( left() != nullptr ) {
        left()->clear( p_left_tree );
    }

    if ( right() != nullptr ) {
        right()->clear( p_right_tree );
    }

    delete this;
    p_to_this = nullptr;
}
```



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

For example,

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

Unicode extends character encoding beyond the Latin alphabet

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.";
```

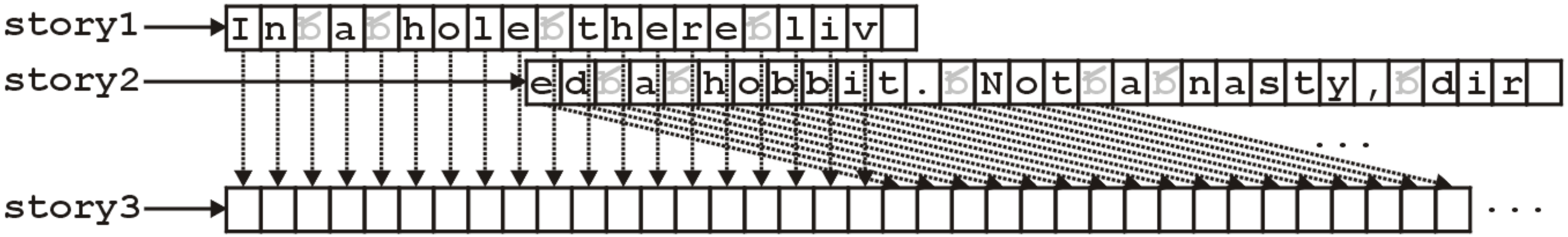
On problem with using arrays is the runtime required to concatenate two strings

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";
```

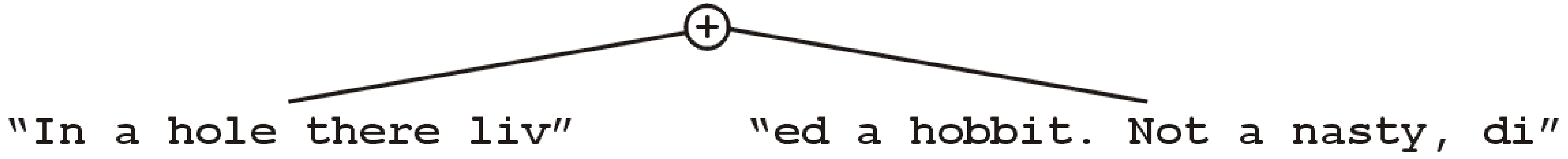


Application: Ropes

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

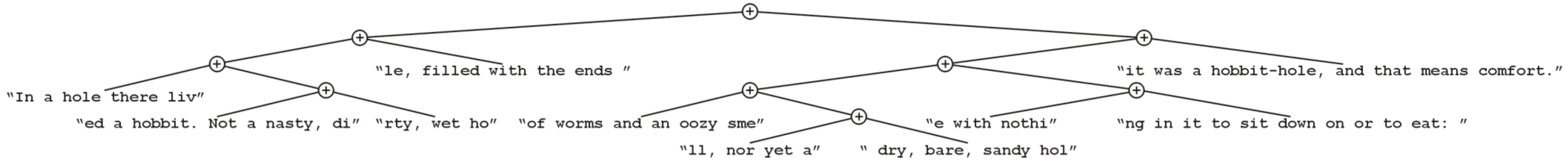


Application: Ropes

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



Application: Ropes

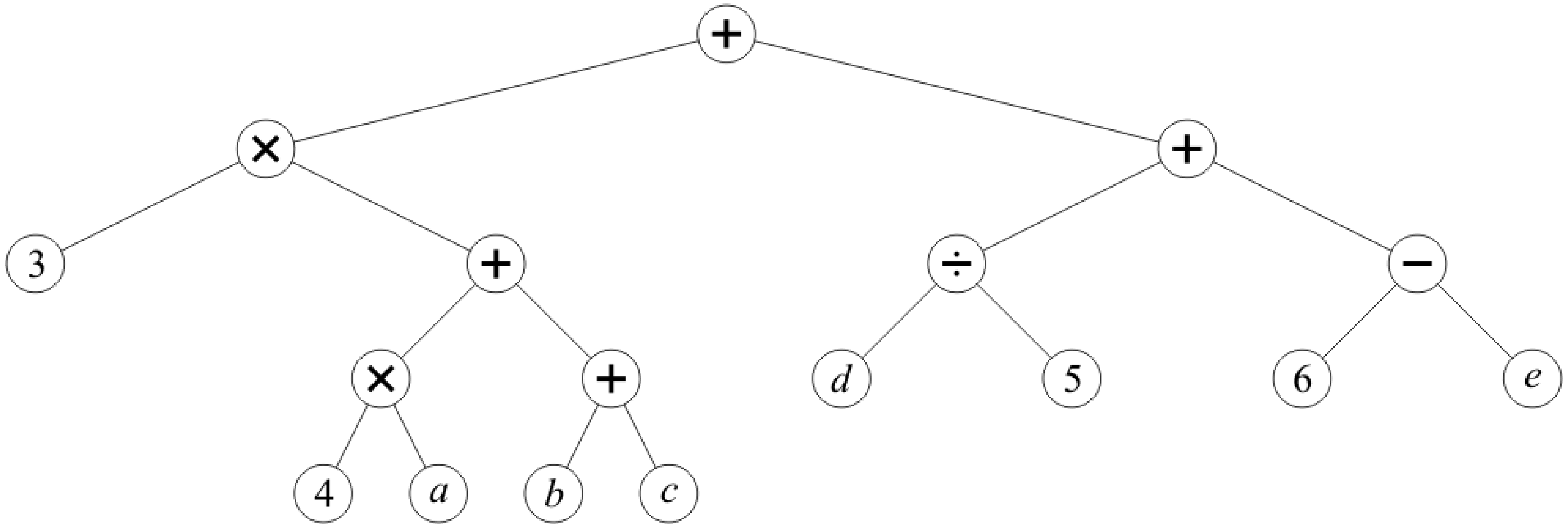
Additional information may be useful:

- Recording the number of characters in both the left and right sub-trees

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

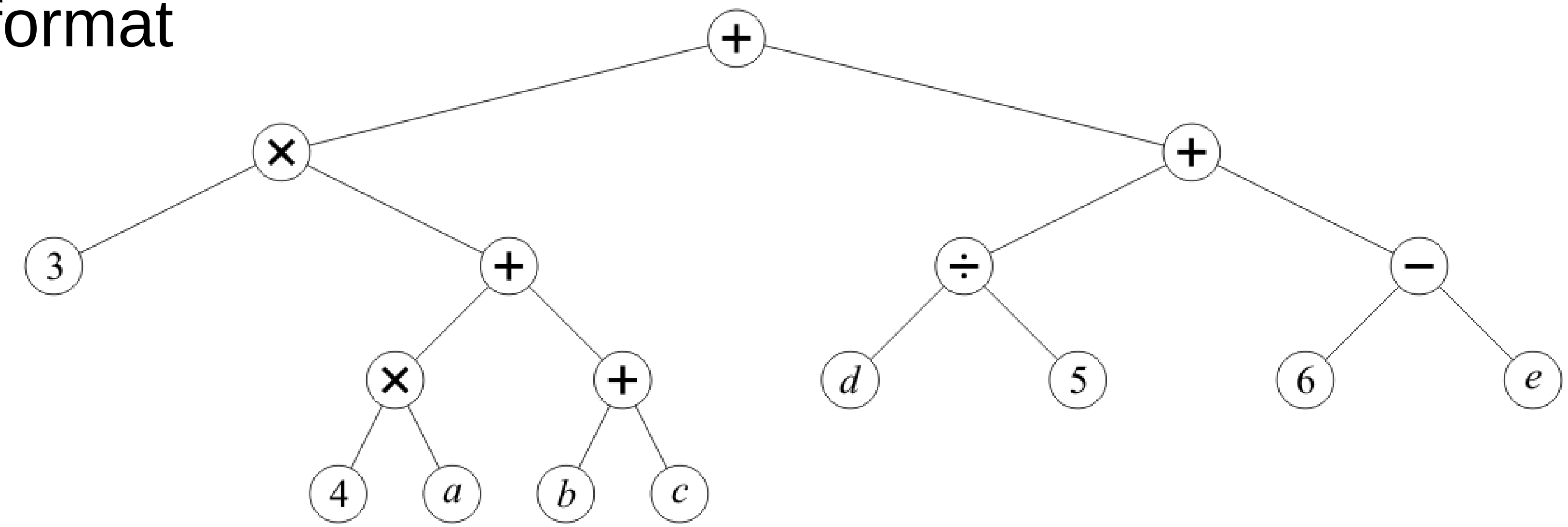
Observations:

- Internal nodes store operators
- Leaf nodes store literals or variables
- No nodes have just one sub tree
- The order is not relevant for
  - Addition and multiplication (commutative)
- Order is relevant for
  - Subtraction and division (non-commutative)
- It is possible to replace non-commutative operators using the unary negation and inversion:

$$(a/b) = a\,b^{-1} \qquad (a - b) = a + (-b)$$

Application: Expression Trees

A post-order depth-first 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 use in-order notation (C, C++, 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