4.4 Binary Trees

Arbitrary hierarchical orders allow an arbitrary number of children per node; however, many trees are restricted, by definition, to having at most only two children per node. Examples include:

1. Expression trees using binary operators
2. An ancestral tree including an individual, his or her parents, their parents, and so on,
3. Phylogenetic trees,
4. Binary space partitioning, and
5. Huffman coding (lossless data compression).

4.4.1 Description

A binary tree is a tree where each node is restricted to having at most two children.

By restricting the number of children to two, this allows us to order the children not only with respect to the other, but also with respect to the parent. One node will be denoted the *left* child and the other the *right* child, as shown in Figure 1. Similarly, we can discuss the left and right sub-trees of a given node, as shown in Figure 2.

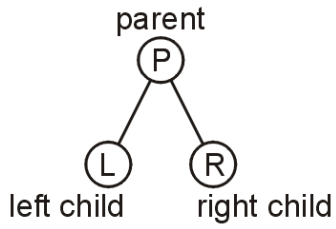


Figure 1. The left and right children of a given node.

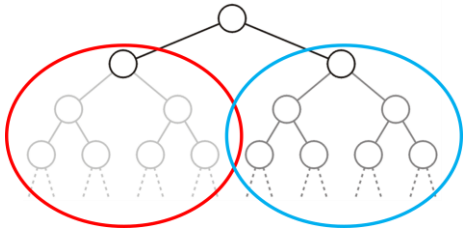


Figure 2. The left and right sub-trees of a given node.

A node that has two children is called a *full node*, as is shown in Figure 3.

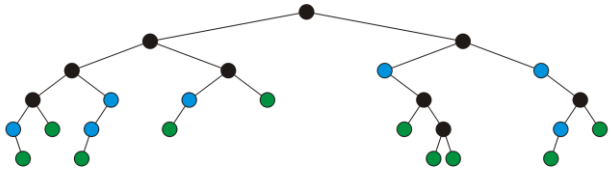


Figure 3. A sample binary tree with full nodes (black), leaf nodes (green), and one-child internal nodes (blue).

Because a binary tree has explicitly named sub-trees, each non-assigned child could be thought of as either an *empty node* or a *null sub-tree*. Figure 4 shows the locations of empty nodes/null sub-trees of the tree shown in Figure 3.

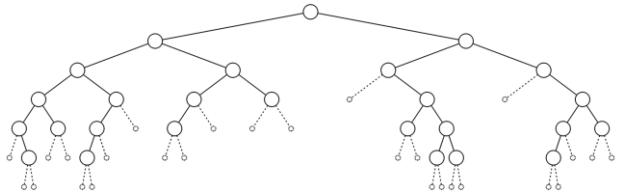


Figure 4. A binary tree with explicitly marked *empty nodes* or *null sub-trees*.

In certain applications, every node must be either a full node or a leaf node. Such a tree, known as a *full binary tree*, is shown in Figure 5.

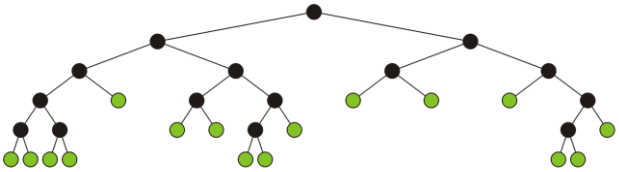


Figure 5. A full binary tree where all nodes are either full or leaf nodes.

These have applications in:

1. Expression trees,
2. Huffman encoding, and
3. Phylogenetic trees.

4.4.2 Implementation

We will look at an implementation of a binary tree that is very close to our `Single_node` class implementation of a singly linked list. A full implementation of this class is found on the ECE 250 web site under `Algorithms/Trees/Binary_trees/`. The basic definition of the class is:

```
#include <algorithm>

template <typename Type>
class Binary_node {
    protected:
        Type element;
        Binary_node<Type> *left_tree;
        Binary_node<Type> *right_tree;

    public:
        Binary_node( Type const & );

        Type retrieve() const;
        Binary_node<Type> *left() const;
        Binary_node<Type> *right() const;

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

Many of these have a functionality similar to that of the `Single_node` or `Simple_tree` classes.

```
template <typename Type>
Binary_node<Type>::Binary_node( Type const &obj ):
element( obj ),
left_tree( 0 ),
right_tree( 0 ) {
    // Empty constructor
}

template <typename Type>
Type Binary_node<Type>::retrieve() const {
    return element;
}

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

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

The first executable instruction of any program is in memory location 0 and instructions cannot be accessed through memory. Thus, 0 is used to represent the *null pointer*. It is possible to call a member function on a null pointer; however, if there is any attempt to access memory at a null pointer, the operating system will halt the program. This checks if the node is either an empty node/null sub-tree:

```
template <typename Type>
bool Binary_search_node<Type>::empty() const {
    return ( this == 0 );
}

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

The number of nodes in a tree is the number of nodes in the left sub-tree plus the number of nodes in the right sub-tree plus one, as is shown in Figure 5.

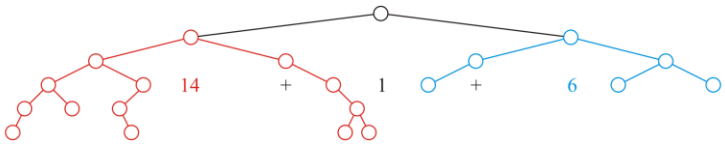


Figure 5. The size (number of nodes) in a tree may be computed recursively: 21 = 14 +1 + 6.

The run time of such a routine, however, will be $\Theta(n)$.

```
template <typename Type>
int Binary_search_node<Type>::size() const {
    return ( this == 0 ) ? 0 : 1 + left()->size() + right()->size();
}
```

Similarly, if we wanted, we could recursively compute the height where we must define the heigh of an empty nod to be -1, as is shown in Figure 6.

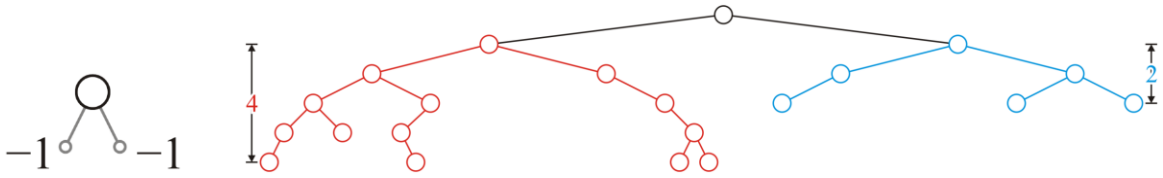


Figure 6. The height of an empty node is -1 and the height of any other tree is one plus the maximum height of either sub-tree—in this case 4 + 1 = 5.

```
template <typename Type>
int Binary_search_node<Type>::height() const {
    return ( this == 0 ) ? -1 : 1 + std::max( left()->height(), right()->height() );
}
```

In order to remove (clear) all the entries in a tree, it is necessary to:

1. Tell the left and right sub-trees to clear all their entries, and
2. Delete the current node.

```
template <typename Type>
void Binary_node<Type>::clear() {
    if ( empty() ) {
        return;
    }
    left()->clear();
    right()->clear();

    delete this;
}
```

4.4.3 Run Times

In general, many operations on a binary tree will require an algorithm to begin at the root and to continue until the algorithm reaches a leaf node. Thus, the run time of many algorithms will depend directly on the height of the tree, *h*. In the worst case, if each node other than a single leaf node has only one child, a tree containing *n* nodes would be of height *n*. We will see in the next class that the best (most shallow) height of a binary tree is approximately $h = \lg(n)$. There is another result that performing random insertions and removals from a binary tree of size *n* will ultimately tend toward a tree with $h = \Theta\left(\sqrt{n}\right)$.

Consider the following table:

n	lg(*n*)	Size
1000	≈ 10	kB
1 000 000	≈ 20	MB
1 000 000 000	≈ 30	GB
1 000 000 000 000	≈ 40	TB
1 000 000 000 000 000	≈ 50	EB

A trillion objects can be stored in a binary tree of height 40.

4.4.4 Applications

We will look at two applications: ropes and binary expression trees. Both require full binary trees.

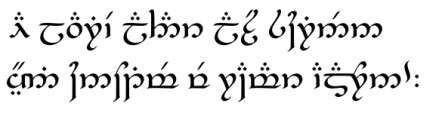
4.4.4.1 Ropes

A string is a linear ordering of characters from a given alphabet. A `char` is a built-in data type representing an 8-bit number from 0 to 255 where many numbers represent letters of the alphabet, numbers, or symbols.

Character	Value	
	Base 10	Binary
`'A'`	65	`01000001`
`'B'`	66	`01000010`
`'a'`	97	`01100001`
`'b'`	98	`01100001`
`' '`	32	`00100000`
`'\0'`	0	`00000000`

Switching case requires only the flip of a single bit, that is, add or subtract 32.

Unicode allows multiple bytes to encode letters from various alphabets; however, we're still waiting for Tengwar alphabet:



J.R.R. Tolkien

The C-style representation of a string is to have an array of characters followed by the *null* character, `'\0'`, that has a value of 0. For example, if we assign

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

this would allocate 32 + 1 = 33 bytes where `story` is assigned the address of the first byte and a character appearing after the period would be `'\0'`. One problem with C-style strings is that concatenation is an expensive operation:

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

Concatenating these two strings would require the allocation of new memory after which all the characters in both strings must be copied to the new memory.

Defintition

Alternatively, we could take the two strings above and assign them to be the leaf nodes of a single node representing concatenation, as is shown in Figure 6.

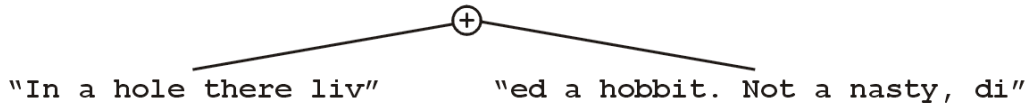


Figure 6. The concatenation of two strings using a binary concatenation operator.

A rope is a general tree structure where all internal nodes represent concatenation and all leaf nodes are strings. A rope is, essentially, a *heavy-duty* string. Figure 7 shows the first paragraph of "The Hobbit" by J.R.R. Tolkien.

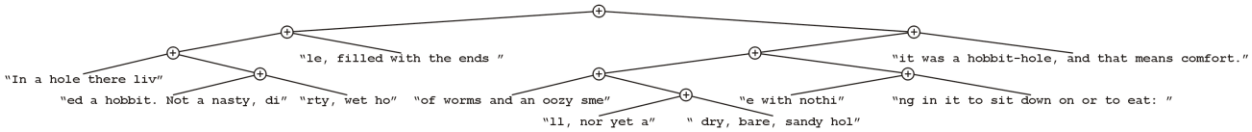


Figure 7. The first paragraph of "The Hobbit" representing the features of a rope.

It is possible to keep information at each of the nodes, for example, the number of characters in the left and right sub-trees. This would help in various operations such as finding the k^{th} character in the string.

4.4.4.2 Binary Expression Trees

Given an expression using binary operators (operators taking two operands), we can represent such an expression using binary trees where the left and right sub-trees are the operands of a given operator. Leaf nodes For example, the expression $3(4a + (b + c)) + d/5 + (6 - e)$ could be represented by the binary tree in Figure 8.

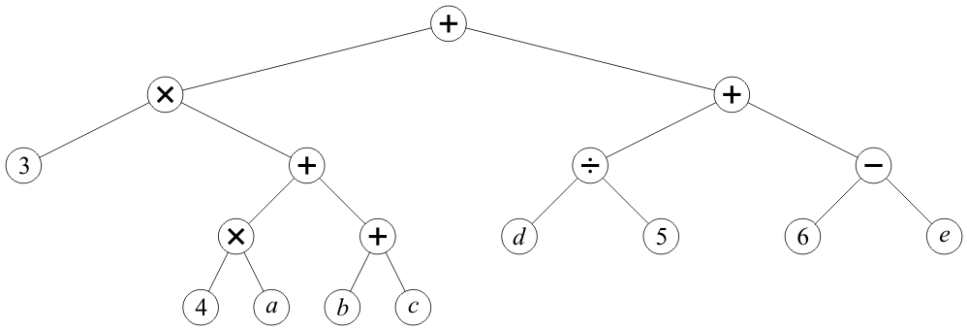


Figure 8. A binary expression tree.

In this example, you will note that:

1. All internal nodes store operators,
2. Leaf nodes store literals or variables,
3. Each node representing an operator has two children
4. The relevance of ordering depends on whether or not the operator is commutative:
    a. Addition and multiplication are commutative while
    b. Subtraction and division are not.

It is possible to interpret non-commutative operations as commutative operations:

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

Additionally, it is possible to convert a binary expression tree into the equivalent reverse-Polish expression by performing a post-order depth-first traversal. Performing such an operation on the binary expression tree in Figure 8 produces the expression

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

Parsers will take a language like C++ and convert it into an expression tree. This allows the parser to convert the expression using in-order operations

```
f = 3*(4*a + (b + c)) + d/5 + (6 - e);
```

into the corresponding sequence of assembly instructions.