4.6a Consider a perfect binary tree with \( n \) nodes and of height \( h \) and then add one more leaf node onto the left-most sub-tree. What are the values of \( \left\lfloor \log(n) \right\rfloor \) and \( \left\lfloor \log(n+1) \right\rfloor \).

4.6b A complete binary tree of height \( h \) has either:

1. A complete binary tree of height \( h - 1 \) as a left sub-tree, and a perfect binary tree of height \( h - 2 \) as a right sub-tree, or
2. A perfect binary tree of height \( h - 1 \) as a left sub-tree, and a complete binary tree of height \( h - 1 \) as a right sub-tree.

Use this to prove by induction that a complete tree of height \( h \) has between \( 2^h \) and \( 2^{h+1} - 1 \) nodes.

4.6c What is the relationship between the number of nodes in a complete binary tree and the number of internal nodes that are not full nodes?

4.6d What is the number of leaf nodes in a complete binary tree with \( n \) nodes?

4.6e Use our array representation to store the complete binary tree in Figure 1 using an array as discussed in class.

![Figure 1. A complete binary tree.](image)

Which entry \( k \) is 42 located in?

Using \( k \), what is the entry of the parent of 42? What are the entries of the children of 42?

4.6f The following is an array representation of a complete binary tree. What is the actual tree?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>84</td>
<td>57</td>
<td>81</td>
<td>42</td>
<td>54</td>
<td>73</td>
<td>60</td>
<td>31</td>
<td>25</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Without referring to the binary tree, what are the parent and children of the entry containing 42? How would find the parent and children of the node containing 54?
Consider the following class:

template <typename Type, int N>
class Complete_binary_tree {
private:
    Type array[N + 1];
    int complete_size;
    int find( Type const & ) const;

public:
    Complete_binary_tree();
    Type parent( Type const & );
    Type left( Type const & );
    Type right( Type const & );
    void push_back( Type const & );
    Type pop_back();
};

Complete_binary_tree(): complete_size( 0 ) {
    // nothing else to initialize
}

where

1. find(...) searches through the array and returns the index of the entry containing it and returns 0 if the argument is not found in the array.
2. parent(...) returns the element that is stored in the parent node of the node containing the argument; it throws underflow() if this member function is called on the root of the tree and illegal_argument() if the argument is not in the tree.
3. left(...) returns the element that is stored in the left child of the node containing the argument; it throws overflow() if this member function is called on a node with no left child and illegal_argument() if the argument is not in the tree.
4. right(...) returns the element that is stored in the left child of the node containing the argument; it throws overflow() if this member function is called on a node with no left child and illegal_argument() if the argument is not in the tree.
5. push_back(...) does nothing if the argument is already in the tree and inserts a new unique argument into the next available location in the complete tree structure. It throws overflow() if the complete binary tree is full (it contains N entries) when attempting to add a new unique element.
6. pop_back(...) removes the last object in the complete tree structure. It throws underflow() if the complete binary tree is empty (it contains no entries).

Note that N is declared in the template: consequently, all memory is immediately allocated. For example, I could declare

    Complete_binary_array<int, 16> cba;
and the compiler would immediately memory for the `complete_size` member variable and an array of size 17 on the call stack (it is a local variable). This memory would be immediately cleaned up whenever the variable `cba` goes out of scope.

If one would call

```cpp
Complete_binary_array<int, 16> *pcba = new Complete_binary_array<int, 16>();
```

this would request memory for $4 + 17 \times 4 = 72$ bytes from the operating system. When delete is called on the returned memory location, all the memory will be immediately freed.

The member function `find(…)` is given here:

```cpp
template <typename Type, int N>
int Complete_binary_tree::find( Type const &obj ) {
    for ( int i = 1; i <= complete_size; ++i ) {
        if ( array[i] == obj ) {
            return i;
        }
    }
    return 0;
}
```

Implement the other member functions. Note that you can use `N` like any other member variable, only you cannot assign to it.