1. [8 points] For each of the following trees, fill in “true” or “false” in the appropriate cells of the table below to indicate if the tree is full, complete, a binary search tree and an AVL tree.

<table>
<thead>
<tr>
<th></th>
<th>Full</th>
<th>Complete</th>
<th>Binary Search</th>
<th>AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>(b)</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>(c)</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>(d)</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
2. [14 points] Consider the following partial declaration of a doubly-linked sorted list class template. (The questions are on the following page.)

```
template <class T>
class SortedList
{
  public:
    // A list of type T.
    // Modeled by:
    //  L sequence of T
    //
    // INV: L is sorted in non-decreasing order.

    // Local types
    enum Status { Ok, NewFail, NoSuchElement };

    // Constructors
    SortedList();
    // Post: L = _
    SortedList(const SortedList<T>& r);
    // Post: L' = r.L

    // Destructor
    ~SortedList();

    // Accessors
    // ...
    Status getStatus() const { return err; }
    // Post: Result = status of last access
    int find(const T& e) const;
    // Pre: L is sorted in non-decreasing order.
    // Post: if e is in L then result = the index of the first occurrence of e
    //       and err' = Ok
    //       if e is not in L then err' = NoSuchElement

    // Mutators
    // ...

  private:
    // Representation:
    // head != 0 -> L = { head->data, head->next->data, ... } /
    // head = 0 -> L = _

    class Node {
      public:
        T data;
        Node* next;
        Node* prev;
        Node(T d = 0, Node* n = 0, Node* p = 0)
        : data(d), next(n), prev(p) { }
        // Note this assumes that T has a copy constructor.
      };

      Node* head;
      Node* tail;

      mutable Status err; // Status of the last call.
  };
```
a) [10 points] Give an implementation of the `find` method such that it satisfies the given specification. Your function should stop searching as soon as a match is found or there is no possibility of finding one.

```cpp
template <class T>
int find(const T& e) const
// Pre: L is sorted in non-decreasing order.
// Post: if e is in L then result = the index (zero based) of the first
// occurrence of e
// and err' = Ok
// if e is not in L then err' = NoSuchElement
{
  int index = 0;
  Node* cur = head;
  while (cur != 0 && cur->data < e) {
    cur = cur->next;
    index++;
  }
  if (cur != 0 && cur->data == e) {
    err = Ok;
  } else {
    err = NoSuchElement;
  }
  return index;
}
```

b) [2 points] What is the time complexity (in big-Oh notation) of your `find` method? (Be sure to specify what \(n\) represents.)

\[O(n)\] where \(n\) is the length of the list.

c) [2 points] The `insert` method must ensure that the class invariant is kept true (i.e., that the list is sorted in non-decreasing order) by inserting elements in the correct position of the list. What is the time complexity (in big-Oh notation) of this algorithm? (Again be sure to specify what \(n\) represents.)

\[O(n)\] where \(n\) is the length of the list.

3. [10 points] The number of 1’s in a binary representation of a natural number, \(N\), is equal to the number of 1’s in the binary representation of \(N/2\) plus 0 if \(N\) is even, or 1 if \(N\) is odd. For example, the binary representation of 6 is 110, which contains two 1’s, as does the binary representations of 3 (11), which is \(1+\) the number of 1’s in the binary representation of 1. Consider a recursive function `int numOnes(int n)` that computes this number.

a) [1 points] What is the base case for the recursion?

\(n = 0\)

b) [1 points] What is the variant expression for the recursion?

\(n\)

c) [8 points] Give an implementation of the function.

```cpp
int numOnes(int n)
{
  int result = 0;
  if (n > 0) {
    result = numOnes(n/2) + n%2;
  }
  return result;
}
```
4. [18 points] Consider the following partial implementation of a binary tree class. (Note: Not a binary search tree.)

```cpp
template <class T>
class BinaryNode // The node of the binary tree
{
    public:
        T key;
        BinaryNode* left;
        BinaryNode* right;
        BinaryNode(BinaryNode *l = 0, BinaryNode* r = 0)
        : left(l), right(r) { }
        BinaryNode(const T& k, BinaryNode *l = 0, BinaryNode* r = 0)
        : key(k), left(l), right(r) { }
};

template <class T>
class BinaryTree
{
    public:
        enum Status { Ok, NewFail, NoSuchElement, DuplicateElement }; // Constructors
        BinaryTree(); // Post: empty tree constructed
        BinaryTree(const BinaryTree<T>& r); // Post: tree is a copy of r
    // Destructor
        virtual ~BinaryTree();
    // Mutators
        Status insert(const T& x); // Post: x is inserted in the tree keeping it optimally filled
        // ...
    protected:
        BinaryNode<T>* root;
        mutable Status err; // Status of the last call.
    private:
        Status rInsert(BinaryNode<T>*& r, const T& x);
        int rMinSpace(BinaryNode<T>* r) const;
        // ...
};

/***************************************************************************/
// insert -- insert an element in the tree, keeping it filled
// Post: //**************************************************************/
template <class T>
BinaryTree<T>::Status
BinaryTree<T>::insert(const T& x)
{
    return rInsert(root, x);
}
```
a) [8 points] An alternative technique for deciding where to insert new nodes is to look for the shortest path from the root to an empty sub-tree (i.e., a 'hole' where a node can be inserted) and insert there. Give a recursive implementation for rMinSpace such that it returns the number of nodes in the shortest path to an empty sub-tree of r. (You may use the STL template function `T min(T l, T r)`, which returns the minimum of its arguments.)

```cpp
template <class T>
int BinaryTree<T>::rMinSpace(BinaryNode<T>* r) const
{
    int result = 0;
    if (r != 0) {
        result = 1 + std::min(rMinSpace(r->left), rMinSpace(r->right));
    }
    return result;
}
```

b) [2 points] What is the time complexity (in big-Oh notation) of your `rMinSpace` function, above? Be sure to state what `n` represents.

**O(n)** where `n` is the number of nodes in the tree.

c) [8 points] Give a recursive implementation of `rInsert` that inserts in the sub-tree containing the closest 'hole', or the left sub-tree if the distance to the closest hole is the same in both sub-trees.

```cpp
template <class T>
BinaryTree<T>::Status BinaryTree<T>::rInsert(BinaryNode<T>*& r, const T& x)
{
    if (r == 0) {
        r = new(std::nothrow) BinaryNode<T>(x);
        if (r != 0) {
            err = Ok;
        } else {
            err = NewFail;
        }
    } else if (rMinSpace(r->left) <= rMinSpace(r->right)) {
        err = rInsert(r->left, x);
    } else {
        err = rInsert(r->right, x);
    }
    return err;
}
```