1. (10 pts) In the boxes below, show the **red-black** trees that result from the successive addition of the given values. Use doubled-lines for red links. Clearly indicate recoloring and rotations, if any, with intermediate trees and “left” or “right” for direction of rotation.

<table>
<thead>
<tr>
<th>1. After adding 62 to a tree with 58.</th>
<th>2. After adding 48 to the previous tree.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. After adding 96 to the previous tree.</td>
<td>4. After adding 34 to the previous tree.</td>
</tr>
<tr>
<td>5. After adding 104 to the previous tree.</td>
<td>6. After adding 85 to the previous tree.</td>
</tr>
</tbody>
</table>
2. (10 pts) In the boxes below, show the **binary heaps** in tree form that result from the successive additions of the given values, where larger values beat lower values. Clearly indicate what swaps occur to maintain the heap.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. After adding 62 to a tree with 58.</td>
<td>2. After adding 48 to the previous tree.</td>
</tr>
<tr>
<td>3. After adding 96 to the previous tree.</td>
<td>4. After adding 34 to the previous tree.</td>
</tr>
<tr>
<td>5. After adding 104 to the previous tree.</td>
<td>6. After adding 101 to the previous tree.</td>
</tr>
</tbody>
</table>
3. (5 pts) Using the heap generated in question 2 as a priority queue, show the swaps that would occur after the first item in the queue is removed.

4. (20 pts) The function getWinner() is supposed to take a vector of names representing votes for candidates and return the name that appears strictly more than half the time, if any, or the empty string. Examples:


Three correct definitions are below. For each, give the computational complexity with a reasoned justification.

a) 
```cpp
string getWinner1( const vector<string> &ballots ) {
    int len = ballots.size();
    for ( int i = 0; i < len; ++i )
        if ( count( ballots.begin(), ballots.end(), ballots[i] ) > len / 2 )
            return ballots[i];
    return "";
}
```
b)  
```cpp
string getWinner2( const vector<string> &ballots ) {  
    int len = ballots.size();  
    map<string, int> votes;  
    for ( int i = 0; i < len; ++i ) ++votes[ ballots[i] ];  
    for ( map<string, int>::iterator iter = votes.begin();  
         iter != votes.end();  
         ++iter )  
        if ( iter->second > len / 2 ) return iter->first;  
    return "";  
}
```

c)  
```cpp
string getWinner3( const vector<string> &ballots ) {  
    int len = ballots.size();  
    string winner = "";  
    int tally = 0;  
    for ( int i = 0; i < len; ++i ) {  
        if ( tally == 0 ) winner = ballots[i];  
        if ( winner == ballots[i] ) ++tally; else --tally;  
    }  
    if ( count( ballots.begin(), ballots.end(), winner )  
         > len / 2 )  
        return winner; else  
        return "";  
}
```

d) Give an argument for the correctness of getWinner3(). Hint: a vote for one candidate cancels a vote for another candidate.
5. (10 pts) Using the C++ tree class below, implement `zigzigRight( Node * &node )` so that `zigzigRight(node->left)` or `zigzigRight(node->right)` inside a Tree member function would do the rotation shown to the specified subtree:

```
template <typename T> class Tree {
    private:
        struct Node {
            Node * left, *right;
            T data;
            ...
        };
    public:
        Node * root;
        void zigzigRight(Node * &node) {
            // Implementation...
        }
    };
```