1. (*20 Points*) Complexity Analysis with the Master Theorem:

   ***Theorem 4.1 (Master theorem)***
   Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by

   $$T(n) = aT(n/b) + f(n),$$

   where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:

   1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
   2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
   3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$. ■

   ---

   (a) Use the Master Theorem to find the complexity of a function that takes an array of size $n$ and does two recursive calls. The first recursive call uses the first quarter of the array and the second call uses the third quarter of the array. The other work done in the function is completed in constant time and does not depend on the size of the array. Simplify all logarithms.

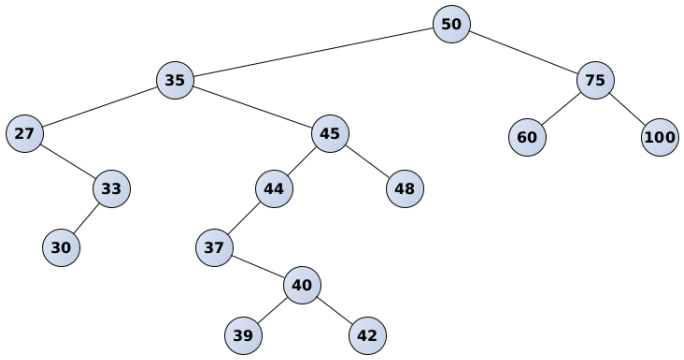
   **Solution:** In this case $a = 2$, $b = 4$, and $f(n) = c$. Looking at the first scenario. Is there an $\epsilon > 0$ with $c = O(n^{\log_4(2) - \epsilon}) = O(n^{1/2 - \epsilon})$? Yes, let $\epsilon = 1/4$. So the complexity of this function is $\Theta(n^{1/2})$.

   (b) Use the Master Theorem to find the complexity of a function that takes an array of size $n$ and does two recursive calls. The first recursive call uses the first quarter of the array and the second call uses the third quarter of the array. The other work done in the function is a single for loop that runs through the input array one time and does one operation to each array element. Simplify all logarithms.

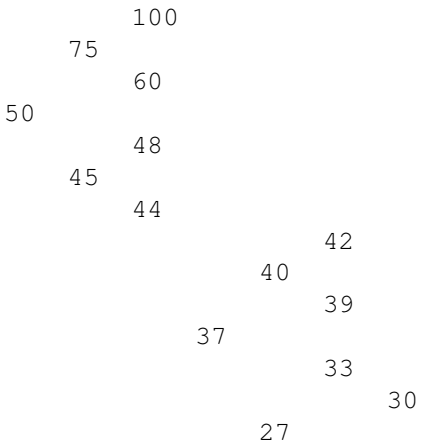
   **Solution:** In this case $a = 2$, $b = 4$, and $f(n) = 3n$.

   - Is there an $\epsilon > 0$ with $3n = O(n^{\log_4(2) - \epsilon}) = O(n^{1/2 - \epsilon})$? No.
   - Is there a $k \geq 0$ with $3n = \Theta(n^{\log_4(2)} \lg^k(n)) = \Theta(n^{1/2} \lg^k(n))$? No, any positive power of $n$ will overtake any power of a logarithm.
   - Is there an $\epsilon > 0$ with $3n = \Omega(n^{\log_4(2) + \epsilon}) = \Omega(n^{1/2 + \epsilon})$? Yes, let $\epsilon = 1/4$. Also, $af(n/b) = 2(3(n/4)) = 3n/2 = 1.5n$, so if we let $c = 0.9 < 1$, we have $cf(n) = 0.9 \cdot 3n = 2.7n > 1.5n$. So the regularity condition is satisfied and the complexity of this function is $\Theta(n)$.

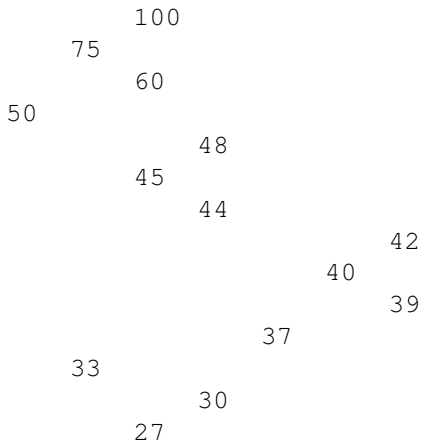
2. (*15 Points*) Given the following binary search tree.



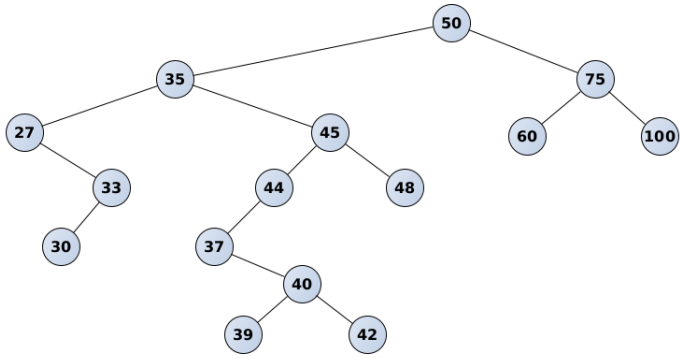

(a) Draw the tree after a delete by merge of 35.
**Solution:**




(b) Draw the tree after a delete by copy of 35.
**Solution:**




3. (*15 Points*) Given the following binary search tree.




(a) Write the output of a preorder traversal print on this tree.
**Solution:** 50 35 27 33 30 45 44 37 40 39 42 48 75 60 100

(b) Write the output of a postorder traversal print on this tree.
**Solution:** 30 33 27 39 42 40 37 44 48 45 35 60 100 75 50

(c) Write the output of a breadth-first traversal print on this tree.
**Solution:** 50 35 75 27 45 60 100 33 44 48 30 37 40 39 42

4. (*15 Points*) Write a search function for a binary search tree that is templated and returns a pointer to the value stored in the node if the value is found and returns `nullptr` if the value is not found. The header to the function is below.

```
template <class T> T *BinaryTree<T>::search(T &item)
```

**Solution:**

```
template <class T> T *BinaryTree<T>::search(T &item) {
  TreeNode *nodePtr = root;

  while (nodePtr) {
    if (nodePtr->value == item)
      return &nodePtr->value;
    else if (item < nodePtr->value)
      nodePtr = nodePtr->left;
    else
      nodePtr = nodePtr->right;
  }
  return nullptr;
}
```

5. (*20 Points*) Write the `makeDeletion` function for the templated binary search tree class that does a deletion by merging. The input parameter is a node pointer sent by reference and is pointing to the node to be deleted. The header to the function is below.

```
template <class T> void BinaryTree<T>::makeDeletion(TreeNode *&nodePtr)
```

**Solution:**

```
template <class T> void BinaryTree<T>::makeDeletion(TreeNode *&nodePtr) {
  TreeNode *tempNodePtr = nullptr;

  if (nodePtr == nullptr)
    cout << "Cannot delete empty node.\n";
  else if (nodePtr->right == nullptr) {
    tempNodePtr = nodePtr;
    nodePtr = nodePtr->left;
    delete tempNodePtr;
  } else if (nodePtr->left == nullptr) {
    tempNodePtr = nodePtr;
    nodePtr = nodePtr->right;
    delete tempNodePtr;
  }
  else {
    tempNodePtr = nodePtr->right;
    while (tempNodePtr->left)
      tempNodePtr = tempNodePtr->left;
    tempNodePtr->left = nodePtr->left;
    tempNodePtr = nodePtr;
    nodePtr = nodePtr->right;
    delete tempNodePtr;
  }
}
```

6. (*15 Points*) Write a function traverseInOrder that takes in a single parameter that is a pointer to a function that can update an integer value, so void return and single int parameter by reference. This can be written as an addition to the IntBinaryTree class. The traversal function will traverse the binary tree in an inorder fashion and apply the function parameter to each of the node values. You will, as usual, want a public non-recursive version taking a single variable and a recursive counterpart that also includes a pointer to a treenode.

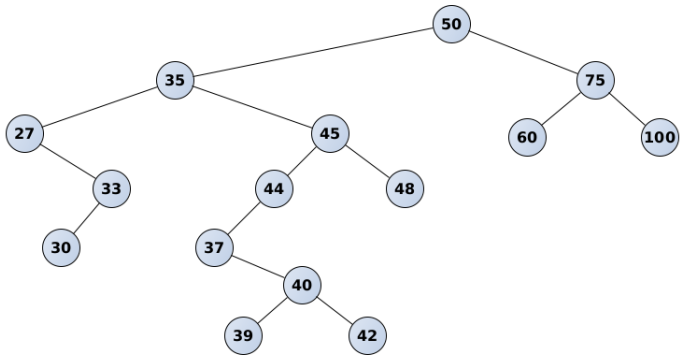
   **Solution:**

```
using updater = void (*)(int &);

void traverseInOrder(TreeNode *, updater) const;
void traverseInOrder(updater fct) const { traverseInOrder(root, fct); }

void IntBinaryTree::traverseInOrder(TreeNode *nodePtr, updater fct) const {
  if (nodePtr) {
    traverseInOrder(nodePtr->left, fct);
    fct(nodePtr->value);
    traverseInOrder(nodePtr->right, fct);
  }
}
```

7. (*10 Points*) **Extra Credit:** Given the following binary search tree.




   (a) Draw the tree after the first stage of the DSW algorithm.

   **Solution:**

```
                                                  100
                                              75
                                          60
                                      50
                                  48
                              45
                          44
                      42
                  40
              39
          37
      35
  33
30
27
```

(b) Show each rotation pass of the second stage of the DSW algorithm and the final prefect tree. Recall that $m = 2^{\lfloor \lg(n+1) \rfloor} - 1$, where $n$ is the number of nodes.

**Solution:** $n = 15$ and $m = 2^{\lfloor \lg(16) \rfloor} - 1 = 2^4 - 1 = 15$. So the first (initial) pass is not done.

```
                        100
                    75
                        60
                50
                    48
            45
                44
        42
            40
    39
        37
35
        33
30
    27
----------------------------
            100
        75
            60
    50
            48
        45
            44
42
            40
    39
            37
35
            33
    30
            27
----------------------------
        100
    75
        60
50
        48
    45
        44
42
        40
    39
        37
35
        33
    30
        27
```