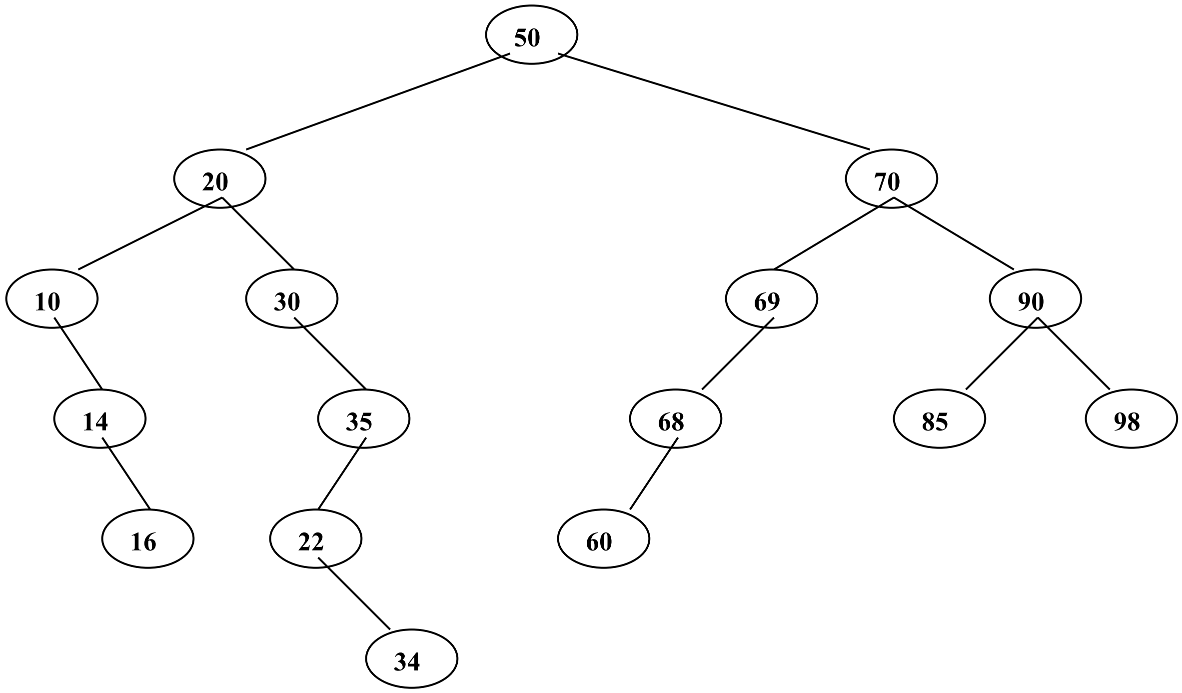


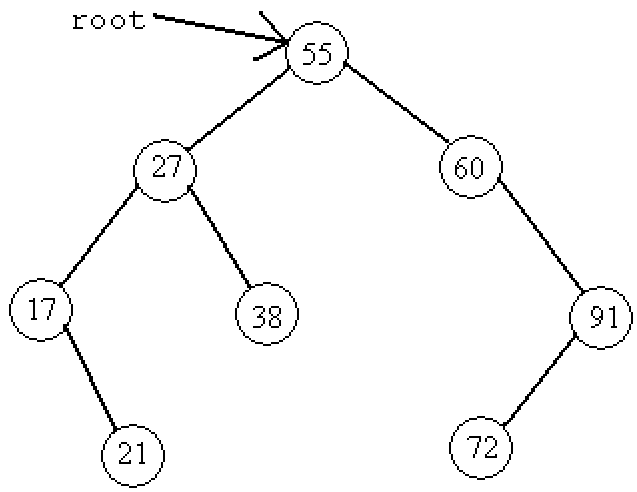
(c) Execute the algorithm shown below using the tree shown above. Show the exact output produced by the algorithm. Assume that the initial call is: **prob3(root)** and that the tree nodes and pointers are defined as shown.

```
struct treeNode{
     int data;
        struct treeNode *left, *right;
}
struct treeNode *tree_ptr;

void prob3(struct tree_ptr *node_ptr) {
   if (node_ptr != NULL){
        if (node_ptr->data % 3 == 0){
             printf("%d ", node_ptr->data);
             prob3(node_ptr->left);
        }
        else{
             if (node_ptr->data % 3 == 1){
                  printf("%d ", node_ptr->data+2);
                  prob3(node_ptr->right);
             }
             else{
                  if (node_ptr->data % 3 == 2){
                       prob3(node_ptr->left);
                       prob3(node_ptr->right);
                  }
             }
        }
   }
 }
```

Output:

12	18	30	72	90	87					

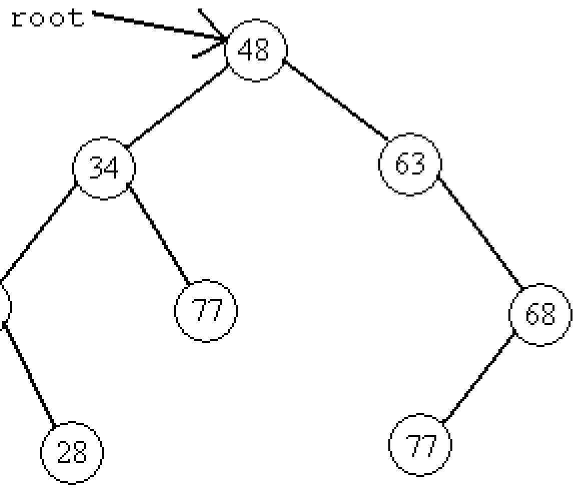
1. **[9 pts]** For the binary tree given below **`root`** is a pointer to the root of the tree.



Redraw (on the following page) the tree shown above when the following function is executed. Assume that the initial call is `modifyT(root, 7, 65)`.

```
struct treeNode
{
   int data;
   struct treeNode *left;
   struct treeNode *right;
};

void modifyT(struct treeNode* node_ptr, int key, int num)
{
   if (node_ptr != NULL)
   {
      if (node_ptr->data % 3 == 0)
      {
         node_ptr->data += key;
         modifyT(node_ptr->left, key + 2, num - key);
         modifyT(node_ptr)->right, key - 3, num + key);
      }
      else if (node_ptr->data % 5 == 0)
      {
         node_ptr->data -= key;
         modifyT(node_ptr->right, key - 4, num);
         modifyT(node_ptr->left, key, num + 5);
      }
      else
      {
         node_ptr->data = num;
         modifyT(node_ptr->right, key - 2, num + 10);
         modifyT(node_ptr->left, key + 5, num - 7);
      }
   }
}
```

Answer for Problem 3



1 point for each correct node modification
1 point extra if the entire tree is correct

5. **[8 pts]** Write a recursive function that compares two given binary trees. It returns 1 if two trees are different and it returns 0 otherwise. Use the node structure and the function prototype provided below:

```
struct treeNode {
      int data;
      struct treeNode * left;
      struct treeNode * right;
};

int check(struct treeNode *A, struct treeNode *B)
```

One possible solution:

```
int check(struct treeNode *A, struct treeNode *B)
{
      if(A == NULL && B == NULL)
            return 0;
      else if(A == NULL || B == NULL)
            return 1;
      else if(a->data != b->data)
            return 1;

      if(check(A->left, B->left) || check(A->right, B->right))
            return 1;
      else
            return 0;
}
```

Grading:
Base cases:
The trees are the same if both trees are NULL (1 point)
The trees are different if one is NULL, the other isn't (1 point)
The trees are different if neither is NULL, but the data differs (1 point)

Recursive cases:
The two trees are different if either the left or right is different (3 points)
The two trees are the same only if both the left and right are the same (2 points)

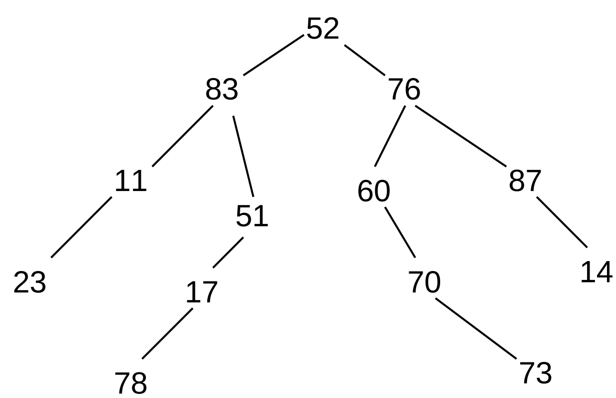
5. **a) [5 pts]** Indicate in few words, the purpose of the following function. The struct treenode is the same as the one defined in problem 4 on the previous page.

```
int f(struct treenode * p) {

    int val;
    if (p == NULL) return 0;
    val = p->data;
    if (val%2 == 0)
        return val + f(p->left) + f(p->right);
    else
        return f(p->left) + f(p->right);

}
```

`Returns the sum of the nodes storing even values in the tree.`

`Grading: 2 pts for sum, 1 pt for node values, 2 pts for even`

b) **[5 pts]** What does the function return, given the following tree?



Answer: **78+52+76+60+70+14 = 350**

3. (10 points) Write a recursive function to compute the height of a tree, defined as the length of the longest path from the root to a leaf node. For the purposes of this problem a tree with only one node has height 1 and an empty tree has height 0. Your function should make use of the following tree node structure:

```
struct treenode {
     int data;
     struct treenode* left;
     struct treenode* right;
};

int height(struct treenode* root)
{
     int leftheight, rightheight;

     if(root == NULL)
          return 0;

     leftheight = height(root->left);
     rightheight = height(root->right);

     if(leftheight > rightheight)
          return leftheight + 1;

     return rightheight + 1;















}
```

3. Write a recursive function **struct node * largest(struct node * B)** which returns a pointer to the node containing the largest element in a BST (binary search tree). The node structure is as follows:

```
struct node {
     int node_value;
     struct node * left, *right;
};
```

[8 pts]

```
struct node* largest(struct node *B){
     if ( B==NULL)
          return NULL;
     else if (B->right ==NULL)
          return B;
     else return largest(B->right);
}
```

Grading: 4 pts for an iterative solution

4. In a binary tree, each node may have a single child, two children, or no child. Write a recursive function **`int one (struct tree_node *p)`** for a binary tree which returns the number of nodes with a single child.
Use the node structure

```
struct tree_node {
     int data;
     struct tree_node * left, *right;
};
```

[10 pts]

```
int one  ( struct tree_node *p){
     if ( p != NULL)
      {
          if( p->left == NULL)
               if( p->right != NULL)
                    return 1+ one(p->right);
          else if( p->right == NULL)
                if( p->left != NULL)
                    return 1+ one(p->left);
     else
                    return one (p->left) + one(p->right) ;
     }
}
```

4) (10 pts) DSN (Binary Trees)

Consider the problem of finding the k^{th} smallest item in a binary search tree storing ***unique*** values. Write a ***recursive*** function with the prototype shown below to solve this problem. The first parameter to the function will be a pointer to the root node of the binary tree and the second value will be the 1-based rank of the item to be returned. (If k is 1, you should return the smallest item in the tree.) **Note that the number of nodes in the subtree at each node is stored inside the node.** You may assume that the value of k will be in between 1 and the value of numNodes in the struct pointed to by root.

```
typedef struct treenode {
    int data;
    int numNodes;
    struct treenode *left;
    struct treenode *right;
} treenode;

int kthSmallestValue(treenode* root, int k) {

    // 2 pts for this case.
    if (root->left == NULL && k == 1)
        return root->data;

    // 2 pts for this case.
    int leftCnt = root->left->numNodes;
    if (k - 1 == leftCnt)
        return root->data;

    // 3 pts for this case.
    if (k - 1 < leftCnt)
        return kthSmallestValue(root->left, k);

    // 3 pts for this case.
    return kthSmallestValue(root->right, k-leftCnt-1);
}
```

4) (10 pts) DSN (Binary Trees)

Write a function that is given a pointer to the root of a valid binary search tree with unique elements and prints out a list of all the odd numbers stored in the binary search tree, in descending order. Use the struct definition and function prototype provided.

```
typedef struct treenode {
    int data;
    struct treenode *left;
    struct treenode *right;
} treenode;

void printOddDescending(treenode* root) {

    if (root == NULL) return;                    // 1 pt

    printOddDescending(root->right);             // 2 pts

    if (root->data%2 == 1)                       // 2 pts
        printf("%d ", root->data);               // 1 pt

    printOddDescending(root->left);              // 2 pts

                                                 // 2 pts for correct
                                                 // order of calls.
}
```

4) (10 pts) DSN (Binary Trees)

Mark and his buddy Travis have devised a password scheme to secure files that they send among themselves. Their scheme hides the password in a string of English letters. The password is the alphabetically ordered sequence of the consonants in the string. So as not to have to compute the password each time, Mark has written a function called `printPassword`, which takes the letters of the original string stored in a binary search tree and prints out the password. For example, if the string in the message is **mental**, the password printed out would be **lmnt**. Or if the string was ***fragile***, then the password would be ***fglr.*** You may call the following function in your solution:

```
// Returns 1 if c is a consonant, 0 otherwise.
int isConsonant(char c)
```

Using the struct definition given below, complete the function in the space provided.

```
typedef struct treenode {
    char ch;
    struct treenode *left;
    struct treenode *right;
} treenode;

void printPassword(treenode* root) {

    if (root != NULL) {                         // 2 pts
        printPassword(root->left);              // 2 pts
        if (isConsonant(root->ch))              // 2 pts
            printf("%c", root->ch);             // 2 pts
        printPassword(root->right);             // 2 pts
    }
}
```