Q4) Answer the following question (focus tree recursion)

```
// A tree is defined as either being empty with a nullptr or having nodes of
// the following type. The tree is also sorted.

struct Tree_node{
      Tree_node * left;
      Tree_node * right;
      int datum;
};

// Requires: root points to valid tree described above
// Modifies: nothing
// Effects: returns the number of nodes in the tree
// Ex:               3
//   num_nodes(     / \    ) -> 3
//                 1   7

int num_nodes(Tree_node * root)
{


}

// Requires: root points to valid tree described above
//        root points to a tree with an odd number of nodes (for simplicity)
//        tree is non-empty
// Modifies: nothing
// Effects: returns the median value found in the tree
// Ex:               3
//      median(     / \    ) -> 3
//                 1   7

int median(Tree_node * root)
{



}
```

Q2) Answer the following questions, focus linked list and templates/iterators

```
// List is singly linked list
// Having a Node the following members: {Node * next; int datum}
```

2.1

```
// Requires: List is valid list (can be nullptr)
// Modifies: The list pointed to by
// Effects: Returns pointer to head of the list given in reverse
// Ex: HEAD[1] -> [2] -> NULL returns HEAD[2] -> [1] -> NULL

Node * reverse_list(Node * head){

}
```

2.2

```
// Requires: List is valid
// Modifies: nothing
// Effects: Returns if this list is circular, empty is not circular
// Ex: HEAD[1] -> [2] -> HEAD[1]... == true, HEAD[1] -> NULL == false

bool is_circular(Node * head){

}
```

2.3

```
template<typename IterType, typename T>
class Internal_Vec{
      vector<T> v1;
public:
      Internal_Vec(){}
      bool am_I_before(IterType it, IterType end); // IMPLEMENT ON NEXT PAGE
};


// am_I_before
// Requires: it is valid iterator and points to a container with type “T”
// Modifies : this
// Effects: Returns true if the element’s datum before it is the same as it’s
//        : then pushes this datum on v1 if true
// Ex: [1][2], it points to [2], returns false;
//     [2][2], it points to second [2], returns true;
// IMPORTANT, this iterator could be pointing at anything, not necessarily v1
// Do everything you must here to make this work, including func signatures.
// You may assume that IterType has --, *, ++ operators implemented, and that
// ==, !=, <, > are implemented for type T
```

Q5) Give output of code below code (focus on try catch)

```
class LolExcept{};
class HahaExcept : public LolExcept {};

void try_catch(int in)
{
      cout << "in: " << in << endl;
      try{
            if(in == 42) throw HahaExcept();
            if(in == 7) throw LolExcept();
      }
      catch(HahaExcept &){
            cout << "Caught at HahaExcept" << endl;
      }
      cout << (42/in) << endl;
}

int main(int argc, char * argv[])
{
      try{
            try_catch(42);
            try_catch(7);
      }
      catch(LolExcept &){
            cout << "Caught at LolExcept 1" << endl;
      }
      catch(...){
            cout << "Caught by everything 1" << endl;
      }
      try{
            try_catch(7);
            try_catch(0);
      }
      catch(LolExcept &){
            cout << "Caught at LolExcept 2" << endl;
      }
      catch(...){
            cout << "Caught by everything 2" << endl;
      }
      return 0;
}
```

5.1

What is output:

Q3) Answer the following questions (focus functors and iterators)

3.1

```
// Write a functor that returns true if earlier in alphabet (< operator)
// Ex. FunFunc f1(“dog”);
// f1(“cat”)); -> True
// f1(“whale”); -> False
class FunFunc{


public:
      FunFunc(                ){

      }
      bool operator() (             ){

      }
};
```

3.2

```
// Requires: begin/dest point to the beginning of a data structure, end to
//           end duh, data structure pointed to by dest is >= size of data
//           structure pointed to by begin
// Modifies: data structure pointed to by dest
// Effects: if pred is false, copy the value into the second data structure
//          pointed to by dest
// Ex: begin -> [“a”,”b”,”c”] and if pred = FunFunc(“b”)
//     then you should end up dest -> [“b”, “c”]

template <typename IterType, typename IterType2, typename Pred>
void grab_on_false(IterType begin, IterType end, Itertype2 dest, Pred pred){









}
```

3.3

Do you need all the templates above in grab_on_false?

3.4

What are the benefits to the following, why exist? Why are functors fun?

Iterators:

Functors:

```
// QUESTIONS ON NEXT PAGE
// CODE:
int where = 4;
int * am = new int(5);

class LeakMem
{
      int * first;
      int second;
      int * arr = new int[where];
      int * arr2 = arr;
public:
      LeakMem(int first_in) : first(new int(first_in))
      {
            cout << "LeakMem Norm Ctor Called" << endl;
            second = 5;
            for(int i = 0; i < 4; i++)
            {
                  arr[i] = i;
            }
      }

      void start_me()
      {
            cout << second << endl;
            cout << *am << endl;
            delete am;
      }

      void run_me()
      {
            cout << where << endl;
            delete first;
            cout << first << endl;
      }
};


int main(int argc, char * argv[])
{
      LeakMem lm(5);
      lm.start_me();
      lm.run_me();
      lm.start_me();
      return 0;
}
```

Q1) Answer the questions about the code on prev. page [wud rec. the diagram first] (focus Dynamic Memory)

1.1: What memory is leaked [from what variable(s)]?

1.2 What double deletes happen or bad access?

1.3: Draw a memory diagram of the process running using the table below [make sure to use the following variables: lm (including all members and what they create), where, am, and functions if you feel like having fun

Stack	Heap	Global