(Email mmgeorg@umich.edu if you have questions or if there are typos)

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)
{
      // Base case-- no nodes in this tree
      if(root == nullptr) { return 0; }

      // Recursive case-- one node here + num in left
      // subtree + num in right subtree
      return num_nodes(root->right) + num_nodes(root->left) + 1;
}

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

// Note -- this one is pretty tricky! It's a good exercise, but don't panic
// if you struggled with it. I probably should have said in the directions
// that you could use a helper function and the num_nodes function you wrote.

int median(Tree_node * root)
{
      return median_helper(0, 0, root)->datum;
}

// Helper function for median
```

```
Tree_node * median_helper(int nodes_left, int nodes_right, Tree_node * root)
{
      // Base case
      if(root == nullptr) { return nullptr; }

      // Calculate "balance" of this node-- how many nodes are to its
      // left vs. how many nodes are to its right?
      int balance = nodes_left + num_nodes(root->left)
                  - nodes_right - num_nodes(root->right);

      // If this is the middle node, this is the median
      if(balance == 0) { return root; }

      // If the balance is positive, there are more nodes to the left
      // than to the right, so we have to look left for the median
      else if(balance > 0) {
            return median_helper(nodes_left, nodes_right
            + num_nodes(root->right) + 1, root->left);
      }

      // If the balance is negative, there are more nodes to the right
      // than to the left, so we have to look right for the median
      else {
            return median_helper(nodes_left + num_nodes(root->left) + 1,
            nodes_right, root->right);
      }
}
```

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
```

VERIFIED

```
Node * reverse_list(Node * head){
    Node * last_visited = nullptr;
    Node * to_fix = head;
    Node * save;

    while(to_fix) {
        save = to_fix->next;
        to_fix->next = last_visited;
```

```
        last_visited = to_fix;
        to_fix = save;
    }

    return last_visited;
}
```

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
VERIFIED
bool is_circular(Node * head){
        if(!head) { return false; }

        Node * save = head;
        head = head->next;

        while(head) {
            if(save == head) { return true; }
            head = head->next;
        }
        return false;
}
```

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:
template <typename IterType, typename T>
bool Internal_Vec::am_I_before(IterType it, IterType end)
{
   if(it == end) { return false; }
```

```
    T val = *it;
    --it;
    if(it == end) { return false; }
    T prev = *it;
    v1.push_back(val);
    return prev == val;
}
```

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

VERIFIED

```
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:

```
in: 42
```

Caught at HahaExcept

1

in: 7

Caught at LolExcept 1

in: 7

Caught at LolExcept 2

(Note: after the catch at LolExcept 2, the try_catch(0) does not execute!)

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{
      string word;
public:
      FunFunc(const string & word_in){
         word = word_in;
      }
      bool operator() (const string & other){
         return other < word;
      }
};
```

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 and both contain the same data type
// 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>
int grab_on_false(IterType begin, IterType end, Itertype2 dest, Pred pred){
   int new_size_dest = 0;
   while(begin != end) {
      if(pred(*begin)) {
         *dest = *begin;
         ++new_size_dest;
         ++dest;
      }
     ++begin;
   }
   return new_size_dest;
}
```

3.3

Do you need all the templates above in grab_on_false?

Yes, because we’re not sure if IterType is the same as IterType2.

3.4

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

Iterators:

Encapsulation, allows us to move through ADTs

Functors:

They help reduce code duplication, help us compare custom types (comparators), etc. Functors aren’t fun. -> sorry Melissa but they are

(Dynamic memory 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)]?
The memory pointed to by arr (the same memory pointed to by arr2) is leaked because it is never deleted with delete[].

1.2 What double deletes happen or bad access?
When we call start_me the first time, am gets deleted. When we call it again, we try to access that zombie object (bad access) and then we delete it again (double free).

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
Lm (in main) first Second (in lm object within main) arr Arr2 (local pointers to heap memory) first_in, i (local variables in LeakMem constructor)	new int(5) new int(first_in) new int[where]	where am