CSCI-1200 Data Structures
Test 3 — Practice Problem Solutions

1 Un-Occupied Erase [/ 39]

Ben Bitdiddle was overwhelmed during the Data Structures lecture that covered the implementation details of `erase` for binary search trees. Separately handling the cases where the node to be erased had zero, one, or two non-NULL child pointers and then moving data around within the tree and/or disconnecting and reconnecting pointers seemed pointlessly complex (pun intended). Ben's plan is to instead leave the overall tree structure unchanged, but mark a node as *unoccupied* when the node containing the value to be erased has one or more children.

Ben's modified `Node` class is provided on the right.

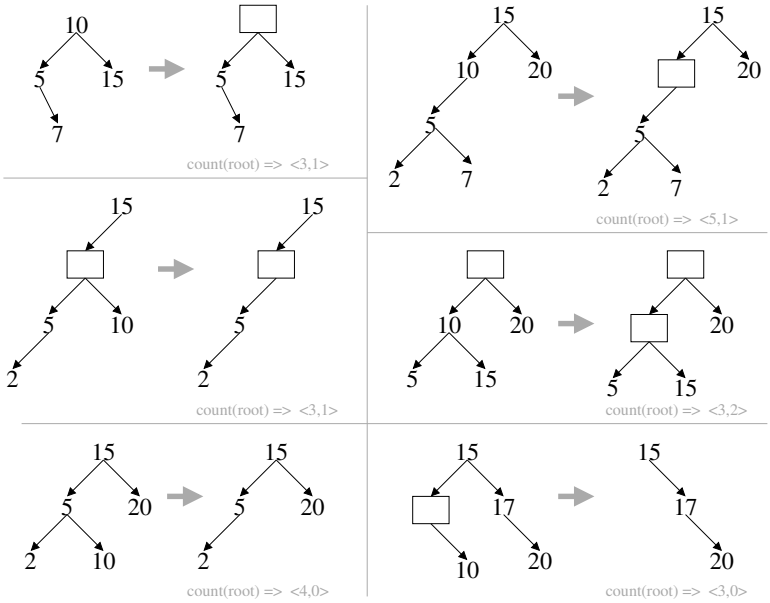
```
template <class T>
class Node {
public:
  Node(const T& v) :
    occupied(true), value(v),
    left(NULL), right(NULL) {}
  bool occupied;
  T value;
  Node* left;
  Node* right;
};
```

1.1 Diagramming the Expected Output of `erase` [/ 6]

First, help Ben work through different test cases for the `erase` function. For each of the sample trees below, draw the tree after the call `erase(root,10)`. The first one has been done for you.

If a node is unoccupied, we draw it as an empty box. Below each result diagram we note the counts of occupied nodes and the number of unoccupied nodes within the tree. (We'll write the `count` function on the next page!) Note that an unoccupied node should always have at least one non-NULL child.

Solution:



1.2 Counting Occupied & Unoccupied Nodes [/ 8]

Now let's write a recursive `count` function that takes a single argument, a pointer to the root of the tree, and returns an STL pair of integers. The first integer is the total number of *occupied* nodes in the tree and the second integer is the total number of *unoccupied* nodes in the tree. Refer to the diagrams on the previous page as examples.

Solution:

```
template <class T>
std::pair<int,int> count(Node<T>* p) {
  if (p == NULL)
    return std::make_pair(0,0);
  // recurse down both branches
  std::pair<int,int> l = count(p->left);
  std::pair<int,int> r = count(p->right);
  // calculate the two totals
```

```
  int occupied = int(p->occupied==true)  + l.first  + r.first;
  int unoccupied = int(p->occupied==false)  + l.second  + r.second;
  // prepare the return value
  return std::make_pair(occupied,unoccupied);
}
```

Alyssa P. Hacker stops by to see if Ben needs any help with his programming. She notes that when we insert a value into a tree, sometimes we will be able to re-use an unoccupied node, and other times we will have to create a new node and add it to the structure. She suggests a few helper functions that will be helpful in implementing the `insert` function for his binary search tree with unoccupied nodes:

```
template <class T>
const T& largest_value(Node<T>* p) {
  assert (p != NULL);
  if (p->right == NULL) {
    if (p->occupied)
      return p->value;
    else
      return largest_value(p->left);
  }
  return largest_value(p->right);
}
```

```
template <class T>
const T& smallest_value(Node<T>* p) {
  assert (p != NULL);
  if (p->left == NULL) {
    if (p->occupied)
      return p->value;
    else
      return smallest_value(p->right);
  }
  return smallest_value(p->left);
}
```

1.3 Implement `erase` for Trees with Unoccupied Nodes [/ 13]

Now implement the `erase` function for Ben's binary search tree with unoccupied nodes. This function takes in two arguments, a pointer to the root node and the value to erase, and returns true if the value was successfully erased or false if the value was not found in the tree.

Solution:

```
template <class T>
bool erase(Node<T>* &p, const T& v) {
  if (p == NULL) {
    return false;          // value not found
  }
  if (p->occupied) {
    if (p->value == v) {   // found the value!
      if (p->left == NULL && p->right == NULL) {
        // leaf node is simply deleted
        delete p;
        p = NULL;
      } else {
        // otherwise mark this node as unoccupied
        p->occupied = false;
      }
      return true;
    } else if (p->value > v) {
      return erase(p->left,v);   // recurse left
    } else {
      return erase(p->right,v);  // recurse right
    }
  } else {
    // this node is unoccupied, and the value to erase might be down
    // either path! recurse in both directions
    bool success = erase(p->left,v) || erase(p->right,v);
    // if after erasing, this node is now a leaf...  delete it!
    if (p->left == NULL && p->right == NULL) {
      assert (success);
      delete p;
      p = NULL;
    }
    return success;
  }
}
```

1.4 Implement `insert` for Trees with Unoccupied Nodes [/ 12]

Now implement the `insert` function for Ben's binary search tree with unoccupied nodes. This function takes in two arguments, a pointer to the root node and the value to insert, and returns true if the value was successfully inserted or false if the value was not inserted because it was a duplicate of a value already in the tree. Use the provided `smallest_value` and `largest_value` functions in your implementation.

Solution:

```
template <class T>
bool insert(Node<T>* &p, const T& v) {
  if (p == NULL) {
    // empty tree, must add a new node!
    p = new Node<T>(v);
    return true;
  }
  if (p->occupied) {
    if (p->value == v) {
      return false;                // duplicate element
    } else if (p->value > v) {
      return insert(p->left,v);    // recurse left
    } else {
      return insert(p->right,v);   // recurse right
    }
  } else {
    // this node is unoccupied, but the value doesn't necessarily fit here
    if (p->left != NULL && v <= largest_value(p->left)) {
      // if there are elements to the left, and at least one is larger, recurse left
      return insert(p->left,v);
    }
    else if (p->right != NULL && v >= smallest_value(p->right)) {
      // if there are elements to the right, and at least one is smaller, recurse right
      return insert(p->right,v);
    }
    // otherwise this value does fit here!
    p->occupied = true;
    p->value = v;
    return true;
  }
}
```

2 Classroom Scheduler Maps [/ 37]

Louis B. Reasoner has been hired to automate RPI's weekly classroom scheduling system. A big fan of the C++ STL `map` data structure, he decided that `map`s would be a great fit for this application. Here's a portion of the main function with an example of how his program works:

```
room_reservations rr;
add_room(rr,"DCC",308);
add_room(rr,"DCC",318);
add_room(rr,"Lally",102);
add_room(rr,"Lally",104);

bool success = make_reservation(rr, "DCC",   308, "Monday",    18, 2,  "DS Exam")    &&
               make_reservation(rr, "DCC",   318, "Monday",    18, 2,  "DS Exam")    &&
               make_reservation(rr, "DCC",   308, "Tuesday",   10, 2,  "DS Lecture") &&
               make_reservation(rr, "Lally", 102, "Wednesday", 10, 10, "DS Lab")     &&
               make_reservation(rr, "Lally", 104, "Wednesday", 10, 10, "DS Lab")     &&
               make_reservation(rr, "DCC",   308, "Friday",    10, 2,  "DS Lecture");
assert (success == true);
```

In the small example above, only 4 classrooms are schedulable. To make a reservation we specify the building and room number, the day of the week (the initial design only handles Monday-Friday), the start time (using military 24-hour time, where 18 = 6pm), the duration (in # of hours), and an STL `string` description of the event.

Here are a few key functions Louis wrote:

```
bool operator< (const std::pair<std::string,int> &a, const std::pair<std::string,int> &b) {
  return (a.first < b.first || (a.first == b.first && a.second < b.second));
}

void add_room(room_reservations &rr, const std::string &building, int room) {
  week_schedule ws;
  std::vector<std::string> empty_day(24,"");
  ws[std::string("Monday")]    = empty_day;
  ws[std::string("Tuesday")]   = empty_day;
  ws[std::string("Wednesday")] = empty_day;
  ws[std::string("Thursday")]  = empty_day;
  ws[std::string("Friday")]    = empty_day;
  rr[std::make_pair(building,room)] = ws;
}
```

Unfortunately, due to hard disk crash, Louis has lost the details of the two `typedef`s and his implementation of the `make_reservation` function. Your task is to help him recreate the implementation.

He does have a few more test cases for you to examine. Given the current state of the reservation system, these attempted reservations will all fail:

```
success = make_reservation(rr, "DCC", 308, "Monday",   19, 3, "American Sniper") ||
          make_reservation(rr, "DCC", 307, "Monday",   19, 3, "American Sniper") ||
          make_reservation(rr, "DCC", 308, "Monday",   22, 3, "American Sniper") ||
          make_reservation(rr, "DCC", 308, "Saturday", 19, 3, "American Sniper");
assert (success == false);
```

With these explanatory messages printed to `std::cerr`:

```
ERROR! conflicts with prior event: DS Exam
ERROR! room DCC 307 does not exist
ERROR! invalid time range: 22-25
ERROR! invalid day: Saturday
```

2.1 The `typedefs` [/ 5]

First, fill in the `typedef` declarations for the two shorthand types used on the previous page.

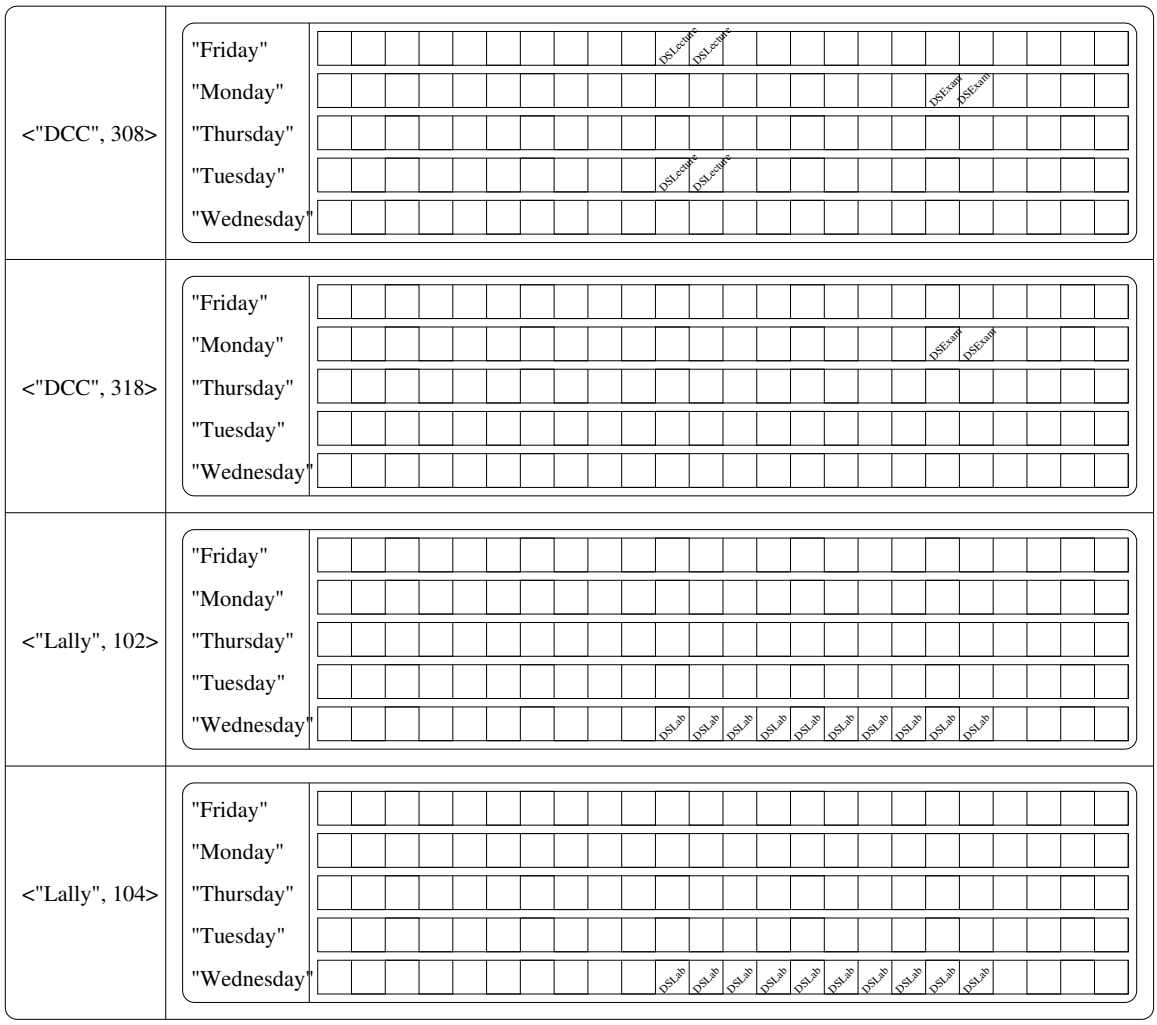
Solution:

```
typedef std::map < std::string, std::vector<std::string> > week_schedule;
typedef std::map < std::pair<std::string,int>, week_schedule > room_reservations;
```

2.2 Diagram of the data stored in `room_reservations rr` [/ 8]

Now, following the conventions from lecture for diagramming `map` data structures, draw the specific data stored in the `rr` variable after executing the instructions on the previous page. Yes, this is actually quite a big diagram, so don't attempt to draw *everything*, but be neat and draw enough detail to demonstrate that you understand how each component of the data structure is organized and fits together.

Solution:



2.3 Implementing `make_reservation` [/ 16]

Next, implement the `make_reservation` function. Closely follow the samples shown on the first page of this problem to match the arguments, return type, and error checking.

Solution:

```
bool make_reservation(room_reservations &rr, const std::string &building, int room,
                      const std::string &day, int start_time, int duration, const std::string &event) {
  // locate the room
  room_reservations::iterator room_itr = rr.find(std::make_pair(building,room));
  if (room_itr == rr.end()) {
    std::cerr << "ERROR! room " << building << " " << room << " does not exist" << std::endl;
    return false;
  }
  // grab the specific day
  week_schedule::iterator day_itr = room_itr->second.find(day);
  if (day_itr == room_itr->second.end()) {
    std::cerr << "ERROR! invalid day: " << day << std::endl;
    return false;
  }
  // check that the time range is valid
  if (start_time + duration > 24) {
    std::cerr << "ERROR! invalid time range: " << start_time << "-" << start_time+duration << std::endl;
    return false;
  }
  // loop over the requested hours looking for a conflict
  assert (day_itr->second.size() == 24);
  for (int i = 0; i < duration; i++) {
```

```
    std::string prior = day_itr->second[start_time+i];
    if (prior != "") {
      std::cerr << "ERROR! conflicts with prior event: " << prior << std::endl;
      return false;
    }
  }
  // if everything is ok, make the reservation
  for (int i = 0; i < duration; i++) {
    day_itr->second[start_time+i] = event;
  }
  return true;
}
```

2.4 Performance and Memory Analysis [/ 8]

Now let's analyze the running time of the `make_reservation` function you just wrote. If RPI has b buildings, and each building has on average c classrooms, and we are storing schedule information for d days (in the sample code d=5 days of the week), and the resolution of the schedule contains t time slots (in the sample code $t = 24$ 1-hour time blocks), with a total of e different events, each lasting an average of s timeslots (data structures lecture lasts 2 1-hour time blocks), what is the order notation for the running time of this function? Write 2-3 concise and complete sentences explaining your answer.

Solution: The outer map has $b * c$ entries. To locate the specific room is $O(\log (b * c))$. Then to locate the specific day is $O(\log d)$, however since the number of days of the week is a small constant, we could say this is $O(1)$. Now, we must loop over the vector and check for availability. We only need to check the specific range of time, s. The total number of slots per day, t, and the total number of events, e, do not impact the running time. Thus, the overall running time is $O(\log (b * c) + \log d + s)$. We will also accept $O(\log(b * c) + s)$.

Using the same variables, write a simple formula for the approximate upper bound on the memory required to store this data structure. Assume each int is 4 bytes and each string has at most 32 characters = 32 bytes per string. Omit the overhead for storing the underlying tree structure of nodes & pointers. Do not simplify the answer as we normally would for order notation analysis. Write 1-2 concise and complete sentences explaining your answer.

Solution: The outer map has $b * c$ entries. Each inner map has d rows. Each row has a vector with t timeslots. Each slot of the vector will store at most a 32 character string. The e and s variables don't matter if we assume the schedule is rather full. Overall answer: b*c* (32 + 4 + d * (32 + t * 32)) = 36*b*c (memory to store each building & room pair) + 32*d*b*c (memory to store the days of the week strings) + 32*d*t*b*c (memory to store an event name string in each timeslot)

Finally, using the same variables, what would be the order notation for the running time of a function (we didn't ask you to write this function!) to find all currently available rooms for a specific day and time range? Write 1-2 concise and complete sentences explaining your answer.

Solution: We would need to loop over all $b * c$ entries in the outer map. The query to see if each room is available is $O(\log d + s)$. Thus, the overall running time is $O(b * c * (\log d + s))$. We will also accept $O(b * c * s)$.

3 Fashionable Sets [/ 14]

In this problem you will write a recursive function named `outfits` that takes as input two arguments: `items` and `colors`. `items` is an STL `list` of STL `string`s representing different types of clothing. `colors` is an STL `list` of STL `set`s of STL `string`s representing the different colors of each item of clothing. Your function should return an STL `vector` of STL `string`s describing each unique outfit (in any order) that can be created from these items of clothing.

Here is a small example:

```
items   =  { "hat", "shirt", "pants" }
colors  =  { { "red" },
             { "red", "green", "white" },
             { "blue", "black" } }
```

```
red hat &   red shirt &  blue pants
red hat & green shirt &  blue pants
red hat & white shirt &  blue pants
red hat &   red shirt & black pants
red hat & green shirt & black pants
red hat & white shirt & black pants
```

Solution:

```
// intentionally copying the items & colors lists (we will edit them later)
std::vector<std::string> outfits(std::list<std::string> items, std::list<std::set<std::string> > colors) {
  assert (items.size() == colors.size());
  // base case, no items!
  std::vector<std::string> answer;
  if (items.size() == 0) {
    // one answer, the empty outfit
    answer.push_back("");
    return answer;
  }
  // pop off the last item & set of colors
  std::string item = items.back();
  items.pop_back();
  std::set<std::string> c = colors.back();
  colors.pop_back();
  // recurse with the shortened item list & colors list
  std::vector<std::string> recurse_answer = outfits(items,colors);
  // combine each color with the current item
  for (std::set<std::string>::iterator itr = c.begin(); itr != c.end(); itr++) {
    // add that to the front of the list
    for (int i = 0; i < recurse_answer.size(); i++) {
      if (recurse_answer[i].size() > 0) {
        answer.push_back(recurse_answer[i]+" & "+*itr+" "+item);
      } else {
        // special case for first item of clothing
        answer.push_back(*itr+" "+item);
      }
    }
  }
  return answer;
}
```

4 Spicy Chronological Sets using Maps [/ 33]

Ben Bitdiddle is organizing his spice collection using an STL `set` but runs into a problem. He needs the fast `find`, `insert`, and `erase` of an STL set, but in addition to organizing his spices alphabetically, he also needs to print them out in chronological order (so he can replace the oldest spices).

Ben is sure he'll have to make a complicated custom data structure, until Alyssa P. Hacker shows up and says it can be done using an STL `map`. She quickly sketches the diagram below for Ben, but then has to dash off to an interview for a Google summer internship.

Alyssa's diagram consists of 3 variables. The first variable, containing most of the data, is defined by a `typedef`. Even though he's somewhat confused by Alyssa's diagram, Ben has pushed ahead and decided on the following interface for building his spice collection:

```
chrono_set cs;
std::string oldest = "";
std::string newest = "";
insert(cs,oldest,newest,"garlic");
insert(cs,oldest,newest,"oregano");
insert(cs,oldest,newest,"nutmeg");
insert(cs,oldest,newest,"cinnamon");
insert(cs,oldest,newest,"basil");
insert(cs,oldest,newest,"sage");
insert(cs,oldest,newest,"dill");
```

chrono_set cs:

"basil"	<"cinnamon", "sage">
"cinnamon"	<"nutmeg", "basil">
"dill"	<"sage", "">
"garlic"	<"","oregano">
"nutmeg"	<"oregano","cinnamon">
"oregano"	<"garlic", "nutmeg">
"sage"	<"basil", "dill">

std::string oldest: "garlic"

std::string newest: "dill"

Ben would like to output the spices in 3 ways:

```
ALPHA ORDER:        basil   cinnamon       dill     garlic     nutmeg    oregano       sage
OLDEST FIRST:      garlic    oregano     nutmeg   cinnamon      basil       sage       dill
NEWEST FIRST:        dill       sage      basil   cinnamon     nutmeg    oregano     garlic
```

If he buys more of a spice already in the collection, the old spice jar should be discarded and replaced. For example, after calling:

```
insert(cs,oldest,newest,"cinnamon");
```

The spice collection output should now be:

```
ALPHA ORDER:        basil   cinnamon       dill     garlic     nutmeg    oregano       sage
OLDEST FIRST:      garlic    oregano     nutmeg      basil       sage       dill   cinnamon
NEWEST FIRST:    cinnamon       dill       sage      basil     nutmeg    oregano     garlic
```

4.1 The `typedef` [/ 3]

First, help Ben by completing the definition of the typedef below:

Solution:

```
typedef std::map<std::string,std::pair<std::string,std::string> > chrono_set;
```

4.2 Printing out the spice collection [/ 8]

Next, write the code to output (to `std::cout`) Ben's spices in alphabetical and chronological order:

Solution:

```
std::cout << "ALPHA ORDER:  ";
chrono_set::const_iterator itr;
for (itr = cs.begin(); itr != cs.end(); itr++) {
  std::cout << " " << itr->first;
}
std::cout << std::endl;
std::cout << "OLDEST FIRST: ";
std::string current = oldest;
while (current != "") {
  std::cout << " " << current;
  current = cs.find(current)->second.second;
}
std::cout << std::endl;
```

4.3 Performance Analysis [/ 5]

Assuming Ben has n spices in his collection, what is the order notation for each operation? *Note: You may want to first complete the implementation of the* `insert` *operation on the next page.*

Solution:
printing in alphabetical order:
Iterating through a map is linear in the number of elements in the map, $O(n)$.

printing in chronological order:
Finding each "next" element requires a *log* n find operation, overall = $O(n$ *log* $n)$.

`insert`-ing a spice to the collection:
Each map operation is *log* n, discard constant multiplier = $O($*log* $n)$.

4.4 Implementing `insert` for the `chrono_set` [/ 17]

Finally, implement the `insert` function for Ben's spice collection. Make sure to handle all corner cases.

Solution:

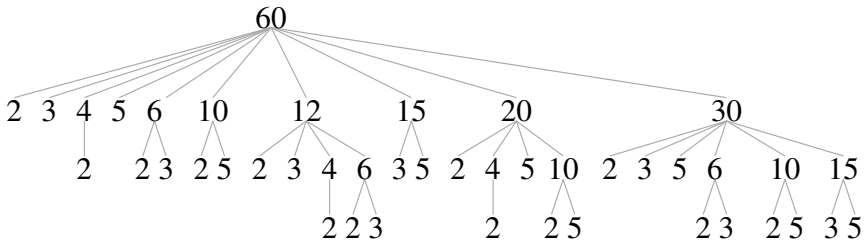
```
void insert(chrono_set &cs, std::string &oldest, std::string &newest, const std::string &spice) {
  // assume the spice isn't here, and try to add it to the end of the chronological order
  std::pair<chrono_set::iterator,bool> tmp =
    cs.insert(std::make_pair(spice,std::make_pair(newest,std::string(""))));
  // if the insert failed (spice was already there)
  if (tmp.second == false) {
    // need to edit the spices before & after the old copy of the spice
    std::string prev = tmp.first->second.first;
```

```
    std::string next = tmp.first->second.second;
    if (prev != "") {
      cs[prev].second = next;
    } else {
      // if the spice was the oldest
      oldest = next;
    }
    if (next != "") {
      cs[next].first = prev;
    } else {
      // if the spice was the newest
      newest = prev;
    }
    // reset the fields of this spice correctly
    tmp.first->second.second = "";
    tmp.first->second.first = newest;
  }
  if (cs.size() == 1) {
    // the very first spice
    oldest = newest = spice;
  } else {
    // point the previous newest spice at this spice
    cs[newest].second = spice;
  }
  newest = spice;
}
```

5 Factor Tree [/ 13]

Write a recursive function named `factor_tree` that takes in a single argument of integer type and constructs the tree of the factors (and factors of each factor) of the input number. The function returns a pointer to the root of this tree. The example below illustrates the tree returned from the call `factor_tree(60)`.

```
class Node {
public:
  int value;
  std::vector<Node*> factors;
};
```



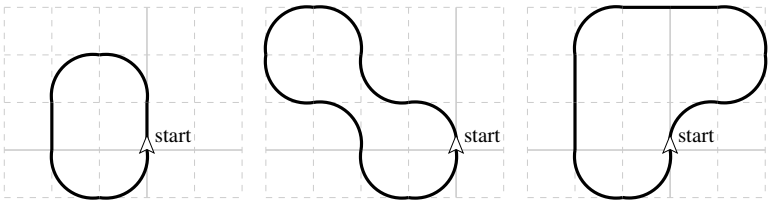
Solution:

```
Node* factor_tree(int num) {
  Node* answer = new Node;
  answer->value = num;
  for (int i = 2; i <= num/2; i++) {
    if (num % i == 0) {
      answer->factors.push_back(factor_tree(i));
    }
  }
  return answer;
}
```

6 Driving in Circles [/ 18]

In this problem you will write a recursive function named `driving` that outputs to `std::cout` all *closed loop* paths of driving instructions on a rectangular grid less than or equal to a specified maximum path length. The car begins at (0,0) pointing north and at each step can go *straight*, *left*, or *right*. A path is said to "close the loop" if it is finishes where it started, pointing in the same direction. For example, here are three sample closed loop paths (also illustrated below):

```
  closed loop:  straight left left straight left left
  closed loop:  left right left left left right left left
  closed loop:  right left left straight straight left straight straight left left
```



We provide the `Car` class and several helper functions:

```
class Car {
public:
  Car(int x_,int y_,std::string dir_) : x(x_),y(y_),dir(dir_) {}
  int x;
  int y;
  std::string dir;
};
bool operator==(const Car &a, const Car &b) {
  return (a.x == b.x && a.y == b.y && a.dir == b.dir);
}
Car go_straight(const Car &c) {
  if      (c.dir == "north") { return Car(c.x  ,c.y+1,c.dir); }
  else if (c.dir == "east")  { return Car(c.x+1,c.y  ,c.dir); }
  else if (c.dir == "south") { return Car(c.x  ,c.y-1,c.dir); }
  else                       { return Car(c.x-1,c.y  ,c.dir); }
}
Car turn_left(const Car &c) {
  if      (c.dir == "north") { return Car(c.x-1,c.y+1,"west");  }
  else if (c.dir == "east")  { return Car(c.x+1,c.y+1,"north"); }
  else if (c.dir == "south") { return Car(c.x+1,c.y-1,"east");  }
  else                       { return Car(c.x-1,c.y-1,"south"); }
}
Car turn_right(const Car &c) {
  if      (c.dir == "north") { return Car(c.x+1,c.y+1,"east");  }
  else if (c.dir == "east")  { return Car(c.x+1,c.y-1,"south"); }
  else if (c.dir == "south") { return Car(c.x-1,c.y-1,"west");  }
  else                       { return Car(c.x-1,c.y+1,"north"); }
}
```

Your function should take in 3 arguments: the path constructed so far, the current car position & direction, and the maximum number of steps/instructions allowed. For example:

```
std::vector<std::string> path;
Car car(0,0,"north");
int max_steps = 10;
driving (path,car,max_steps);
```

Now implement the recursive `driving` function.

Solution:

```
void driving(std::vector<std::string> &path, const Car &car, int max_steps,
     std::vector<Car> previous = std::vector<Car>()) {
  // base case, solution!
  if (path.size() > 0 && car == Car(0,0,"north")) {
    std::cout << "closed loop: ";
    for (int i = 0; i < path.size(); i++) {
      std::cout << " " << path[i];
    }
    std::cout << std::endl;
    return;
  }
  // base case, maximum recursion depth
  if (path.size() == max_steps) { return; }
  // make sure we aren't overlapping previous car positions
  // note: we are allowing the path to cross though!
```

```
  for (int i = 0; i < previous.size(); i++) {
    if (car == previous[i]) return;
  }
  previous.push_back(car);
  // try to go straight
  path.push_back("straight");
  driving(path,go_straight(car),max_steps,previous);
  path.pop_back();
  // try to go left
  path.push_back("left");
  driving(path,turn_left(car),max_steps,previous);
  path.pop_back();
  // try to go right
  path.push_back("right");
  driving(path,turn_right(car),max_steps,previous);
  path.pop_back();
  previous.pop_back();
}
```

7 Maps of Sets of Factors [/ 32]

In this problem we will use STL `map` and STL `set` to store and access a collection of integers and their factors. Below are the commands we use to initialize the two data structures diagrammed on the right.

```
factor_type factors;
add_factors(factors,6);
add_factors(factors,14);
add_factors(factors,5);
add_factors(factors,8);
add_factors(factors,10);
add_factors(factors,9);
add_factors(factors,13);
add_factors(factors,15);
add_factors(factors,12);
add_factors(factors,21);
factor_type is_factor_of = reverse(factors);
```

factors

5	
6	2 3
8	2 4
9	3
10	2 5
12	2 3 4 6
13	
14	2 7
15	3 5
21	3 7

is_factor_of

2	6 8 10 12 14
3	6 9 12 15 21
4	8 12
5	10 15
6	12
7	14 21

7.1 The `typedef` [/ 2]

First, complete the definition of the typedef below:

Solution:

```
typedef std::map<int,std::set<int> > factor_type;
```

7.2 Implementing `add_factors` [/ 8]

Now, implement the `add_factors` function. Note that this function only initializes the `factors` table.

Solution:

```
void add_factors(factor_type &factors, int x) {
  // prepare the set of factors
  std::set<int> tmp;
  // O(x) loop over the range of all possible factors
  for (int i = 2; i <= x/2; i++) {
    if (x % i == 0) {
      // insert the factors into the set O(log j)
      tmp.insert(i);
    }
  }
  // O (log n)
  factors.insert(make_pair(x,tmp));
```

```
}
```

If we are storing the factors of n different numbers in the `factors` structure, f different factors will eventually be stored in the `is_factor_of` structure, each number has on average (or at most) j factors, and each factor is a factor of on average (or at most) k numbers, what is the order notation for the running time of your `add_factors` function to add the number x and the factors of x?

Solution: O($x * \log j + \log n$) or O($x + j * \log j + \log n$)

7.3 Implementing `reverse` [/ 10]

Next, implement the `reverse` function to build the `is_factor_of` table from the completed `factors` table.

Solution:

```
factor_type reverse(const factor_type &factors) {
  factor_type answer;
  // O(n) loop over the numbers/rows in the map table
  for (factor_type::const_iterator itr = factors.begin(); itr != factors.end(); itr++) {
    // O(j) loop over the factors
    for (std::set<int>::iterator tmp = itr->second.begin(); tmp != itr->second.end(); tmp++) {
      // O(log f + log k) add an association to the output table
      answer[*tmp].insert(itr->first);
    }
  }
  return answer;
}
```

Using the variables n, f, j, and k as defined above, what is the order notation for the running time of your `reverse` function?

Solution: O($n * j * (\log f + \log k)$)

7.4 Implementing `remove` [/ 12]

Finally, we would like to remove data from the tables. The `remove` function will remove a given number's row from the `factors` table **and** remove the number from each of its factors in the `is_factor_of` table. For example, the call below results in the tables to the right.

```
remove(12,factors,is_factor_of);
```

Your task is to efficiently implement the `remove` function. Using the variables defined above, you should assume that $n \geq f \geq k \geq j$.

factors

5	
6	2 3
8	2 4
9	3
10	2 5
13	
14	2 7
15	3 5
21	3 7

is_factor_of

2	6 8 10 14
3	6 9 15 21
4	8
5	10 15
7	14 21

Solution:

```
void remove(int n, factor_type &factors, factor_type &is_factor_of) {
  // locate the item in the factors table, O(log n)
  factor_type::iterator itr = factors.find(n);
  if (itr == factors.end()) return;
  // loop over the j factors of that number, O(j)
  for (std::set<int>::iterator tmp = itr->second.begin(); tmp != itr->second.end(); tmp++) {
    // locate the factor in the is_factor_of table, O(log f)
    factor_type::iterator itr3 = is_factor_of.find(*tmp);
    assert (itr3 != is_factor_of.end());
    // remove from the set, O(log j)
    int success = itr3->second.erase(n);
    assert (success);
    if (itr3->second.size() == 0) {
      is_factor_of.erase(itr3);
    }
  }
  factors.erase(itr);
```

```
}
```

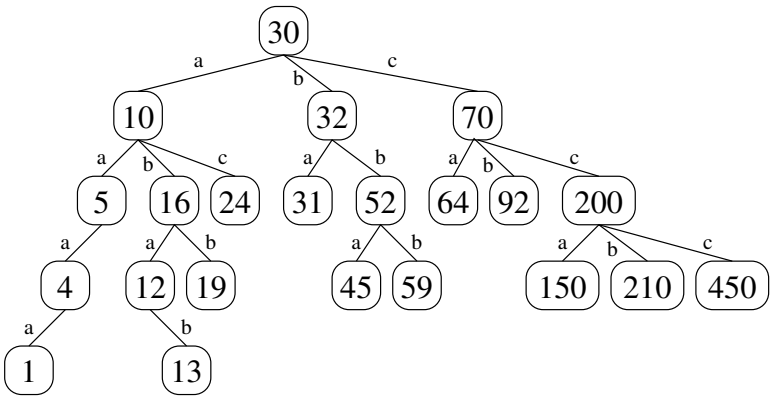
Using the variables n, f, j, and k as defined above, what is the order notation for the running time of your `remove` function?

Solution: O(log n + j * (log f + log k))

8 Double Tries [/ 30]

```
class Node {
public:
  int value;
  Node *a;
  Node *b;
  Node *c;
  Node *parent;
};
```

Ben Bitdiddle thinks he's come up with a fantastic enhancement for binary search trees of integers he calls the Double Trie. Each `Node` will have up to 3 children. The a branch will store all elements less than the current node. The b branch will store all elements greater than the current node, but less than or equal to twice the current node. And the c branch will store all elements greater than twice the current node.



Ben suggests we start the implementation by writing a recursive `insert` function that takes in a pointer to node (initially the *root* of the Double Trie), the value to insert, and a pointer to the parent node (initially NULL). The function returns true if the value was successfully inserted and false if the value is already in the structure.

8.1 Implementing `insert` [/ 10]

Solution:

```
bool insert(Node* &n, int value, Node* parent = NULL) {
  if (n == NULL) {
    n = new Node;
    n->value = value;
    n->a = n->b = n->c = NULL;
    n->parent = parent;
    return true;
  }
  if (n->value == value) {
    return false;
  }
  if (value < n->value) {
    return insert (n->a,value,n);
  } else if (value < n->value*2) {
    return insert (n->b,value,n);
  } else {
    return insert (n->c,value,n);
  }
}
```

Ben's project partner Alyssa P. Hacker isn't thrilled with the design. (She's not sure it will significantly reduce the tree height because this structure is difficult to keep balanced.) However, they have a deadline, so this is a make-it-work moment and she tackles the challenge of *reverse iteration* over this structure. Specifically if the `root` variable points to the top of the diagram on the previous page, she would like this fragment of code:

```
Node *tmp = find_largest(root);
while (tmp != NULL) {
  std::cout << tmp->value << " ";
  tmp = find_previous(tmp);
}
std::cout << std::endl;
```

to print all of the data in the tree *in reverse order*:

```
450 210 200 150 92 70 64 59 52 45 32 31 30 24 19 16 13 12 10 5 4 1
```

8.2 Implementing `find_largest` [/ 6]

Next, Alyssa implements the `find_largest` function:

Solution:

```
Node* find_largest(Node *n) {
  assert (n != NULL);
  if (n->c != NULL)
    return find_largest(n->c);
  if (n->b != NULL)
    return find_largest(n->b);
  return n;
}
```

Given a reasonably balanced Double Trie with n elements, what is the order notation of the running time of the `find_largest` function? Write one or two sentences explaining your answer.

Solution: Each additional level of the tree cuts the size of the data by a factor of 3. The bottom row of a perfect balanced tree has 3^h elements, where h is the height of the tree. Therefore, $h \sim O(\log_3 n)$. The `find_largest` function simply walks the height of the tree, which can be reduced – is equivalent to – O(log n).

8.3 Implementing `find_previous` [/ 14]

Finally, Alyssa implements the `find_previous` function:

Solution:

```
Node* find_previous(Node *n) {
  // assume we are given a legal node
  assert (n != NULL);
  // first choice: go down the 'a' branch (smaller values)
  if (n->a != NULL) {
    return find_largest(n->a);
  }
  // otherwise, walk up the tree until we find a parent/grandparent/etc.
  // that is smaller in value (we are a 'b' or 'c' child of the parent)
  while (1) {
    if (n->parent == NULL) {
      // if we hit a NULL parent, we are done
      return NULL;
    }
    if (n->parent->a == n) {
      n = n->parent;
    } else {
      break;
    }
  }
  // if we are a 'c' branch, visit the 'b' branch (if it exists)
  if (n == n->parent->c && n->parent->b != NULL) {
    return find_largest(n->parent->b);
  }
  // otherwise, return the parent (we'll visit the 'a' branch after that)
  return n->parent;
}
```

9 Re-Truthization [/ 14]

The statements below are false. Make a small change to correct each statement, ensuring that it remains interesting and informative.

Binary Search Tree Iterators [/2] The average number of child or parent links that must be traversed when moving from one node to the next node in an in-order traversal is $O(\log n)$, where n is the number of elements in the tree.

Solution: The average number of child or parent links that must be traversed when moving from one node to the next node in an in-order traversal is $O(1)$ = CONSTANT. The WORST CASE number of links that must be traversed when moving from one node to the next node in an in-order traversal is $O(\log n)$, where n is the number of elements in the tree AND THE TREE IS BALANCED.

Incomplete type [/2] In HW8 Friendly Recursion, many students encountered the compiler message "`error: invalid use of incomplete type 'class Message'`", which should be solved by implementing all custom class member functions in the class declaration `.h` file.

Solution: In HW8 Friendly Recursion, many students encountered the compiler message "`error: invalid use of incomplete type 'class Message'`", which should be solved by ADDING `#include "message.h"` AT THE TOP OF THE EACH `.cpp` FILE, AS NEEDED.

Breadth-First Search [/3] Executing a breadth-first search for the shortest path from root to leaf on a binary search tree will often be faster and require less additional memory than a depth-first search on the same tree.

Solution: Executing a breadth-first search for the shortest path from root to leaf on a balanced binary search tree is most likely going to be faster BUT ALSO REQUIRE MORE ADDITIONAL MEMORY than a depth-first search on the same tree.

Been Here Before? [/2] To optimize the HW6 Ricochet Robots solver, the search tree for a board with three robots can be pruned (and the forward search from that point terminated) if any one of the robots reaches a board location that it has previously occupied.

Solution: To optimize the HW6 Ricochet Robots solver, the search tree for a board with three robots can be pruned (and the forward search from that point terminated) if ALL OF THE ROBOTS SIMULTANEOUSLY REACH A BOARD STATE THEY (AS A GROUP) HAVE previously occupied (WITH FEWER OR EQUAL NUMBER OF MOVES).

Hash Function Performance [/2] A hash function should run in $O(1)$ time, to ensure that the hash table will achieve $O(\log n)$ query time, where n is the number of elements in the hash table.

Solution: A hash function should run in $O(1)$ time, to ensure that the hash table will be able to achieve $O(1)$ = CONSTANT EXPECTED query time, where n is the number of elements in the hash table.

Red-Black Property [/3] Maintaining the Red-Black property for a hash table ensures that the data remains balanced and elements can be accessed in $O(\log n)$ time.

Solution: Maintaining the Red-Black property for a BINARY SEARCH TREE, ensures that the data remains balanced and elements can be accessed in $O(\log n)$ time.

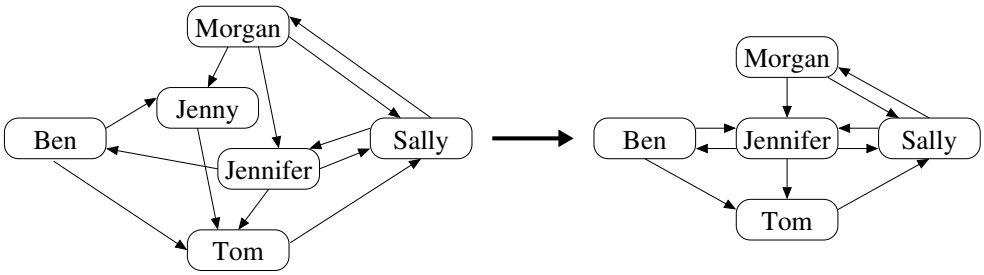
10 Friend Graphs [/ 21]

In this problem you will work with a simplified version of the Twitter or Google+ friendship/follower directed graph from HW8. Your task is to write a `merge_account` function that takes in a `Graph` object and the STL `string` names of two accounts and then modifies the graph to merge the two corresponding `Person` objects into a single object. Other people in the graph who were connected to either of the original accounts by a connection of either direction will be updated to link to the merged account. For example, let's start with the graph connectivity on the left below. We want to merge the accounts for "Jennifer" and "Jenny", preserving the name "Jennifer" on the merged account. We execute the following statement, resulting in the picture on the right.

```
merge_accounts(graph, "Jennifer", "Jenny");
```

```
class Person {
public:
  std::string name;
  std::set<Person*> friends;
};

class Graph {
public:
  std::vector<Person*> people;
};
```



10.1 Corner Cases for `merge_accounts` [/ 6]

Think carefully about a typical use case for merging accounts, and also about corner cases for the `merge_accounts` function. What different test cases will you need to write to ensure that your implementation is fully debugged and will work when attempting to join two arbitrary `Person` objects in a large graph? Write three or four concise and well written sentences describing sample input and the expected output.

Solution: We need to handle bad input, the input strings are equal to each other or one or more input strings does not match a person object in the graph. We need to handle the case where the two person objects are directly linked to each other with a pointer in one or both directions. We'll want to handle cases where the two accounts have no nodes in common, or where both accounts point to or are pointed at by the same account (and prevent duplicates in the merged structure). We should also test cases where one or both accounts is not connected to the other nodes in the graph or where one node has only outgoing links and the other node has only ingoing links.

10.2 Implementation of `merge_accounts` [/ 15]

Now, implement the `merge_accounts` function. Make sure that your function does not lead to memory errors or memory leaks.

Solution:

```
void merge_accounts(Graph &g, const std::string &a, const std::string &b) {
  // the accounts to merge must be different
  if (a == b) return;   /* then do nothing */
  // look up the corresponding person objects
  Person *person_a = NULL;
  Person *person_b = NULL;
  // remember the index of the 2nd object (so we can delete it later)
  int which = -1;
  for (int i = 0; i < g.people.size(); i++) {
    if (g.people[i]->name == a)
      person_a = g.people[i];
    if (g.people[i]->name == b) {
      person_b = g.people[i];
      which = i;
    }
  }
  // make sure both objects exist
  if (person_a == NULL || person_b == NULL) return;  /* then do nothing */
  // loop over all of the outgoing pointers from the 2nd object
  for (std::set<Person*>::iterator itr = person_b->friends.begin(); itr != person_b->friends.end(); itr++) {
    if (person_a->friends.find(*itr) != person_a->friends.end() && person_a != *itr) {
      // if the first object does not already have that outgoing link (and it
      // is not the first object) add that outgoing link to the first object
      person_a->friends.insert(*itr);
    }
  }
  // loop over all of the graph objects looking for pointers to the 2nd object
  for (int i = 0; i < g.people.size(); i++) {
    if (g.people[i]->friends.find(person_b) != g.people[i]->friends.end()) {
      // remove that incoming link to the second object
      g.people[i]->friends.erase(person_b);
      if (g.people[i]->friends.find(person_a) != g.people[i]->friends.end() && g.people[i] != person_a) {
        // and instead add an incoming link to the first object (if it
```

```
        // does not already exist and it's not a self link to itself
        g.people[i]->friends.insert(person_a);
      }
    }
  }
  // edit the graph vector to remove the 2nd object
  g.people[which] = g.people.back();
  g.people.pop_back();
  // delete the second object
  delete person_b;
}
```

11 Lamp Class Inheritance [/26]

In this problem you will complete the implementation for a group of three interrelated classes. The first class is used to represent a light bulb and simply stores the wattage of that bulb:

```
class Bulb {
public:
  int getWatts() const { return watts; }
  void setWatts(int w) { watts = w; }
private:
  int watts;
};
```

The second class represents an electric lamp with a fixed number of sockets to hold `Bulb` objects. The constructor takes the number of bulbs and initial wattage and dynamically allocates an array of bulbs of the specified wattage. The `switch_lights_on` member function turns on the bulbs in the lamp and returns the total number of watts consumed by the lights in the lamp (the sum of the individual bulb wattages).

```
class Lamp {
public:
  Lamp(int n, int watts);
  virtual ~Lamp();
  // MODIFIERS
  void replace_bulb(int i, int watts) {
    assert (i >= 0 && i < num_bulbs);
    sockets[i].setWatts(watts); }
  int switch_lights_on();
  virtual void switch_off() { lights_on = false; }
private:
  // REPRESENTATION
  int num_bulbs;
  Bulb *sockets; // a dynamically allocated array
  bool lights_on;
};
```

The third class is derived from the Lamp object to represent ceiling lamps that also have a fan. The lights and fan can be separately switched on for `FanLamp` objects, but the `switch_off` member function should turn switch off both components.

```
class FanLamp : public Lamp {
public:
  FanLamp(int n, int watts) : Lamp(n, watts) {}
  ~FanLamp() {}
  // MODIFIERS
  void switch_fan_on() { fan_on = true; }
  void switch_off();
private:
```

```
    // REPRESENTATION
    bool fan_on;
  };
```

Here's an example of how to construct a polymorphic vector of these objects, switch on the light bulbs in the third lamp, and replace one of the bulbs in the second lamp (a `FanLamp`).

```
  vector<Lamp*> lamps;
  lamps.push_back(new Lamp(1,100));
  lamps.push_back(new FanLamp(3,40));
  lamps.push_back(new Lamp(2,60));

  lamps[2]->switch_lights_on();
  lamps[1]->replace_bulb(2,60);
```

11.1 Constructors & Destructors [/9]

Implement the constructor and destructor for the `Lamp` class, as they would appear in the implementation (`.cpp`) file.

Solution:

```
Lamp::Lamp(int n, int watts) {
  num_bulbs = n;
  sockets = new Bulb[num_bulbs];
  for (int i = 0; i < num_bulbs; i++) {
    sockets[i].setWatts(watts);
  }
}

Lamp::~Lamp() {
  delete [] sockets;
}
```

11.2 The `virtual` keyword [/3]

What is the purpose of the `virtual` keyword? Write 1-2 concise and well-written sentences.

Solution: When the same member function is implemented in more than one class within the inheritance hierarchy, the virtual keyword on the parent class function indicates that the derived class function should be used if it is available. Without the virtual keyword, the search for a matching function begins at the pointer type and searches "up" the inheritance hierarchy as necessary.

11.3 Turning the Lamps On & Off [/9]

Implement the the `Lamp::switch_lights_on()` and `FanLamp::switch_off()` functions as they would appear in the implementation file.

Solution:

```
int Lamp::switch_lights_on() {
  lights_on = true;
  int answer = 0;
  for (int i = 0; i < num_bulbs; i++) {
    answer += sockets[i].getWatts();
  }
  return answer;
}
```

```
void FanLamp::switch_off() {
  fan_on = false;
  Lamp::switch_off();
}
```

11.4 Manipulating a Polymorphic Vector of Lamps [/5]

Now write a fragment of code that manipulates `lamps`, a polymorphic vector of pointers to `Lamp` objects, to turn the lights on for only the `FanLamp` objects in that vector.

Solution:

```
for (unsigned int i = 0; i < lamps.size(); i++) {
  FanLamp *f = dynamic_cast<FanLamp*>(lamps[i]);
  if (f)
    f->switch_lights_on();
}
```

12 Advanced Topics Potpourri [/9]

Most (but not all) of the statements below are false. Identify each statement as false or true, and correct each false statement so that it is true (but still informative).

12.1 Exceptions [/3]

True or False A class constructor may "fail" only in two manners: it may throw an exception or it may return NULL (useful when the system is out of memory).

Solution: False, the only way for a constructor to fail is to throw an exception. If the system is out of memory, the constructor should throw an out-of-memory exception.

12.2 Multiple Inheritance [/3]

True or False When the inheritance diagram includes a set of classes that form a diamond, two instances of the base class will be created unless the keyword "trapezoid" is used in the `.cpp` file to explicitly specify the construction of only one instance of the base class.

Solution: False. The keyword "virtual" should be used in the `.h` file to specify virtual inheritance from the base class.

12.3 Operator Overloading [/3]

True or False Good programming style for class design encourages the use of operator overloading even when the operator meaning is not intuitively clear because a shorter program will always be easier to understand and maintain.

Solution: False. Operator overloading should be used sparingly, and only when intuitively clear, because otherwise it will be easy to use incorrectly, and it may be confusing to track down the problem.