Set insert<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
There are two different versions of the insert member function. The first version inserts the entry into the set and returns a pair. The first component of the returned pair refers to the location in the set containing the entry. The second component is true if the entry wasn't already in the set and therefore was inserted. It is false otherwise. The second version also inserts the key if it is not already there. The iterator pos is a "hint" as to where to put it. This makes the insert faster if the hint is good.

```
pair<iterator,bool> set<Key>::insert(const Key& entry);
iterator set<Key>::insert(iterator pos, const Key& entry);
```

Set erase<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
There are three versions of erase. The first erase returns the number of entries removed (either 0 or 1). The second and third erase functions are just like the corresponding erase functions for maps. Note that the erase functions do not return iterators. This is different from the vector and list erase functions.

```
size_type set<Key>::erase(const Key& x);
void set<Key>::erase(iterator p);
void set<Key>::erase(iterator first, iterator last);
```

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
// increment & decrement operators
tree_iterator<T> & operator++() {
  if (ptr_->right != NULL) { // find the leftmost child of the right node
    ptr_ = ptr_->right;
    while (ptr_->left != NULL) ptr_ = ptr_->left;
  }
  else { // go upwards along right branches.stop after the first left
    while (ptr_->parent != NULL && ptr_->parent->right == ptr_)
      ptr_ = ptr_->parent;
    ptr_ = ptr_->parent;
  }
  return *this;
}
tree_iterator<T> operator++(int) {
  tree_iterator<T> temp(*this);
  ++(*this);
  return temp;
}
tree_iterator<T> & operator--() {
  if (ptr_ == NULL) {
    assert(set_ != NULL);
    ptr_ = set_->root_;
    while (ptr_->right != NULL) ptr_ = ptr_->right;
  }
  else if (ptr_->left != NULL) {
    ptr_ = ptr_->left;
    while (ptr_->right != NULL) ptr_ = ptr_->right;
  }
  else {
    while (ptr_->parent != NULL && ptr_->parent->left == ptr_)
      ptr_ = ptr_->parent;
    ptr_ = ptr_->parent;
  }
  return *this;
}
tree_iterator<T> operator--(int) {
  tree_iterator<T> temp(*this);
  --(*this);
  return temp;
}
TreeNode<T>*  copy_tree(TreeNode<T>* old_root,
TreeNode<T>* the_parent) {
  if (old_root == NULL) return NULL;
  TreeNode<T> *answer = new TreeNode<T>();
  answer->value = old_root->value;
  answer->left = copy_tree(old_root->left,answer);
  answer->right = copy_tree(old_root->right,answer);
  answer->parent = the_parent;
  return answer;
}
void destroy_tree(TreeNode<T>* p) {
  if (!p) return;
  destroy_tree(p->right);
  destroy_tree(p->left);
  delete p;
}
iterator find(const T& key_value, TreeNode<T>* p) {
  if (!p) return end();
  if (p->value > key_value) return find(key_value, p->left);
  else if (p->value < key_value) return find(key_value, p->right);
  else return iterator(p,this);
}
std::pair<iterator,bool> insert(const T& key_value,
TreeNode<T>*& p,
TreeNode<T>* the_parent) {
  if (!p) {
    p = new TreeNode<T>(key_value);
    p->parent = the_parent;
    this->size_++;
    return std::pair<iterator,bool>(iterator(p,this), true);
  }
  else if (key_value < p->value) return insert(key_value, p->left, p);
  else if (key_value > p->value) return insert(key_value, p->right,
p);
  else return std::pair<iterator,bool>(iterator(p,this), false);
}
int erase(T const& key_value, TreeNode<T>* &p) {
  if (!p) return 0;
  // look left & right
  if (p->value < key_value) return erase(key_value, p->right);
  else if (p->value > key_value) return erase(key_value, p->left);
  // Found the node.  Let's delete it
  assert (p->value == key_value);
  if (!p->left && !p->right) { // leaf
    delete p;
    p=NULL;
    this->size_--;
  }
  else if (!p->left) { // no left child
    TreeNode<T>* q = p;
    p=p->right;
    assert (p->parent == q);
    p->parent = q->parent;
    delete q;
    this->size_--;
  }
  else if (!p->right) { // no right child
    TreeNode<T>* q = p;
    p=p->left;
    assert (p->parent == q);
    p->parent = q->parent;
    delete q;
    this->size_--;
  }
  else { // Find rightmost node in left subtree
    TreeNode<T>* q = p->left;
    while (q->right) q = q->right;
    p->value = q->value;
    // recursively remove the value from the left subtree
    int check = erase(q->value, p->left);
    assert (check == 1);
  }
  return 1;
}
```

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Stacks allow access, insertion and deletion from only one end called the top
There is no access to values in the middle of a stack.
Stacks may be implemented efficiently in terms of vectors and lists, although vectors are preferable.
All stack operations are O(1)
Queues allow insertion at one end, called the back and removal from the other end, called the front
There is no access to values in the middle of a queue.
Queues may be implemented efficiently in terms of a list. Using vectors for queues is also possible, but requires more work to get right.
All queue operations are O(1)
Priority Queue<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
// add a new element to the heap
template <class T> void PriorityQueue<T>::push(T element) {
  // first, verify that the element isn't already in the heap
  typename std::map<T,int>::iterator itr = locations.find(element);
  if (itr != locations.end()) {
    std::cout << "ERROR!  priority queue aleady contains " <<
element << std::endl;
    assert (element == itr->first);
  }
  assert (itr == locations.end());
  // add the element at the edge of heap vector and percolate
up
  m_heap.push_back(element);
  locations[element] = m_heap.size()-1;
  this->percolate_up(int(m_heap.size()-1));
}
// remove the top element (minimum value) from the heap
template <class T> void PriorityQueue<T>::pop() {
  assert(!m_heap.empty());
  int success = locations.erase(m_heap[0]);
  assert (success == 1);
  // place the last element temporarily at the top of the heap,
but
  // push it down to a proper position using percolate down
  m_heap[0] = m_heap.back();
  m_heap.pop_back();
  this->percolate_down(0);
}
// remove a specific element that could be anywhere in the
heap
template <class T> void PriorityQueue<T>::remove(T element) {
  // Put the last element to the location where the element is to
be RemoveEdge Pop back the last element and percolate
down the heap
  if (exist(element)) {
    int loc = locations[element];
    locations.erase(element);
    m_heap[loc] = m_heap.back();
    locations[m_heap.back()] = loc;
    m_heap.pop_back();
    update_position(m_heap[loc]);
  }
}
// this element may have had its value change, so we should
adjust the position of the element within the heap (it might
need to move up or down)
template <class T> void PriorityQueue<T>::update_position(T
element) {
  typename std::map<T,int>::iterator itr = locations.find(element);
  assert (itr != locations.end());
  this->percolate_up(itr->second);
  this->percolate_down(itr->second);
}
// for debugging, print all of the data in the heap
template <class T>
void PriorityQueue<T>::print_heap(std::ostream & ostr) const {
  for (int i=0; i<(int)m_heap.size(); ++i)
    ostr << "[" << std::setw(4) << i << "] : "
         << std::setw(6) << m_heap[i]->getPriorityValue()
         << " " << *m_heap[i] << std::endl;
}
// allow the element at this location to move up
template <class T> void PriorityQueue<T>::percolate_up(int i) {
  while(i > 0) {
    if (m_heap[i]->getPriorityValue() < m_heap[get_parent(i)]-
>getPriorityValue()) {
      locations[m_heap[i]] = get_parent(i);
      locations[m_heap[get_parent(i)]] = i;
      std::swap(m_heap[i], m_heap[get_parent(i)]);
      i = get_parent(i);
    }
    else break;
  }
}
// allow the element at this location to move down
template <class T> void PriorityQueue<T>::percolate_down(int i) {
  while (has_left_child(i)) {
    int child = 0;
    // Choose the child to compare against
    if (has_right_child(i) && m_heap[get_right_child(i)]-
>getPriorityValue() <
      m_heap[get_left_child(i)]->getPriorityValue())
      child = get_right_child(i);
    else child = get_left_child(i);
    if (m_heap[child]->getPriorityValue() < m_heap[i]-
>getPriorityValue()) {
      locations[m_heap[child]] = i;
      locations[m_heap[i]] = child;
      std::swap(m_heap[child], m_heap[i]);
      i = child;
    }
    else break;
  }
}
```

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
percolate_down(TreeNode<T> * p) {
  while (p->left) {
    TreeNode<T>* child;
    if (p->right && p->right->value < p->left->value) child = p->right;
    else child = p->left;
    if (child->value < p->value) {
      swap(child, p);
      p = child;
    }
    else break;
  }
}
percolate_up(TreeNode<T> * p) {
  while (p->parent) {
    if (p->value < p->parent->value) {
      swap(p, parent);
      p = p->parent;
    }
    else break;
  }
}
```

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
int my_func(int a, int b) throw(double,bool) {
  if (a > b) throw 20.3;
  else throw false;
}
int main() {
  try my_func(1,2);
  catch (double x) std::cout << " caught a double " << x <<
std::endl;
  catch (…) std::cout << " caught some other type " << std::endl;
}
```

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
for (std::list<Polygon*>::iterator i = polygons.begin(); i!
=polygons.end(); ++i) {
  Quadrilateral *q = dynamic_cast<Quadrilateral*> (*i);
  if (q) std::cout << "diagonal: " <<  q->LongerDiagonal() <<
std::endl;
}
```

Map<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
1. Map search, insert and erase are O(log n).
2. Maps are ordered by increasing value of the key. Therefore, there must be an operator< defined for the key.
3. The function std::make pair **creates a pair object from the given values..**
4. The result of using [] is that the key is always in the map afterwards.
5. m.find(key) **where m is the map object and key is the search key. It returns a map iterator: If the key is in one of the pairs stored in the map, find returns an iterator referring to this pair. If the key is not in one of the pairs stored in the map, find returns m.end().**
6. Insert: m.insert(std::make_pair(key, value)); **returns a pair of a map iterator and a bool:** std::pair<map<key_type, value_type>::iterator, bool> **The insert function checks to see if the key being inserted is already in the map. If so, it does not change the value, and returns a (new) pair containing an iterator referring to the existing pair in the map and the bool value false. If not, it enters the pair in the map, and returns a (new) pair containing an iterator referring to the newly added pair in the map and the bool value true.**
7. void erase(iterator p) **erase the pair referred to by iterator p.**
void erase(iterator first, iterator last) **erase all pairs from the map starting at first and going up to, but not including, last.**
size_type erase(const key_type& k) **erase the pair containing key k, returning either 0 or 1, depending on whether or not the key was in a pair in the map**
Erase and Insert<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
1. The erase member function (for STL vector and STL list) takes in a single argument, an iterator pointing at an element in the container. It removes that item, and the function returns an iterator pointing at the element after the removed item.
2. Similarly, there is an insert function for STL vector and STL list that takes in 2 arguments, an iterator and a new element, and adds that element immediately before the item pointed to by the iterator. The function returns an iterator pointing at the newly added element.
3. Even though the erase and insert functions have the same syntax for vector and for list, the vector versions are O(n), whereas the list versions are O(1).
4. Iterators positioned on an STL vector, at or after the point of an erase operation, are invalidated. Iterators positioned anywhere on an STL vector may be invalid after an insert (or

push back or resize) operation.
5. Iterators attached to an STL list are not invalidated after an insert or erase (except iterators attached to the erased element!) or push back/push front.

Un-Occupied Erase<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>Counting Occupied & Unoccupied Nodes

```
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);
}
```

>>>>>>>>>Implement erase for Trees with Unoccupied Nodes

```
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;
      } // otherwise mark this node as unoccupied
      else 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;
}
```

>>>>>>>>>Implement insert for Trees with Unoccupied Nodes

```
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;
    else if (p->value > v) return insert(p->left,v);
    else return insert(p->right,v);
  }
  else {
// duplicate element: recurse left & recurse right
// this node is unoccupied, but the value doesn't necessarily
fit here
// if there are elements to the left, and at least one is larger,
recurse left
    if (p->left != NULL && v <= largest_value(p->left))
      return insert(p->left,v);
// if there are elements to the right, and at least one is smaller,
recurse right
    else if (p->right != NULL && v >= smallest_value(p->right))
      return insert(p->right,v);
    // otherwise this value does fit here!
    p->occupied = true;
    p->value = v;
    return true;
  }
}
```

Classroom Scheduler Maps<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
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;
}
```

Fashionable Sets<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
// intentionally copying the items & colors lists (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);
      // special case for first item of clothing
      else answer.push_back(*itr+" "+item);
    }
  }
  return answer;
}
```

Factor Tree<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
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;
}
```

Driving in Circles<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
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();
}
```

Maps of Sets of Factors<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>Implementing add factors

```
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?
O(x * log j + log n) or O(x + j * log j + log n)

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>Implementing reverse

```
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;
}
```

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

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>Implementing remove

```
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);
}
```

O(log n + j * (log f + log k))
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

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.

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.

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.

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.

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.

When the same member function is implemented in more than one class within the inher- itance 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.

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.

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.

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.

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

TA: Srinivasan Iyer M 6-8 F 4-6
mentors: Michael DiBuduo,Cameron Root,Isabella Siu,Renjie Xie,Wilson Gregory