Louis B. Reasoner has started a new job in the RPI housing office and is tasked with re-writing their dorm assignment software. An example of the input text file is shown on the right. He has already written code to parse the input file and store the data:

```cpp
Crockett 320 Beth Collins
Barton 201 Sally Morris
Barton 201 Alice Williams
Barton 201 Jessica Smith
Nugent 316 George Davis
Nugent 316 Fred Harrison
Crockett 106 Chris Thompson
Nugent 112 Erin Lee
Nugent 112 Kathy Newton
```

He wrote a helper function to email students with their room assignment and roommate information:

```cpp
void Info(std::string dorms, std::string people, std::string first, std::string last) {
    std::string dorm;
    int room;
    LookupRoomAssignment(people, first, last, dorm, room);
    if (dorm == "") {
        std::cout << first << " " << last << " does not have a room assignment." << std::endl;
    } else {
        std::cout << first << " " << last << " is assigned to " << dorm << " " << room << "." << std::endl;
        PrintRoommates(dorms, first, last, dorm, room);
    }
}
```

And here is sample usage of his code:

```cpp
Info(dorms, people, "Fred", "Harrison");
Info(dorms, people, "Sally", "Harrison");
Info(dorms, people, "Sally", "Morris");
Info(dorms, people, "Chris", "Thompson");
std::cout << "Barton has " << NumRoomsWithOccupancy(dorms, "Barton", 3) << " triple(s)." << std::endl;
std::cout << "Crockett has " << NumRoomsWithOccupancy(dorms, "Crockett", 1) << " singles(s)." << std::endl;
```

Resulting in this output:

Fred Harrison is assigned to Nugent 316.
Fred's roommate(s) are: George
Sally Harrison does not have a room assignment.
Sally Morris is assigned to Barton 201.
Sally's roommate(s) are: Alice Jessica
Chris Thompson is assigned to Crockett 106.
Chris has no roommates.
Barton has 1 triple(s).
Crockett has 2 singles(s).

You will write a few functions to help Louis finish the implementation and then analyze the running time.
1.1 The typedefs

First, let's fill in these typedef declarations. Note: type_a is optional, but it may help simplify your code.

Solution:

typedef std::map<int, std::vector<std::pair<std::string, std::string> > > type_a;
typedef std::map<std::string, type_a> type_b;
typedef std::map<std::pair<std::string,std::string>, std::pair<std::string,int> > type_c;

1.2 Data Structure Sketch

Sketch the contents of the dorms variable for the sample input text file. Follow the conventions from lecture for this diagram.

Solution:

```
<table>
<thead>
<tr>
<th>Dorm</th>
<th>Room</th>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barton</td>
<td>201</td>
<td>&lt;/Sally&quot;, &quot;Morris&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/Alice&quot;, &quot;Williams&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/Jessica&quot;, &quot;Smith&quot;&gt;</td>
</tr>
<tr>
<td>Crockett</td>
<td>106</td>
<td>&lt;/Chris&quot;, &quot;Thompson&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>320</td>
<td>&lt;/Beth&quot;, &quot;Collins&quot;&gt;</td>
</tr>
<tr>
<td>Nugent</td>
<td>112</td>
<td>&lt;/Erin&quot;, &quot;Lee&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>316</td>
<td>&lt;/Kathy&quot;, &quot;Newton&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/George&quot;, &quot;Davis&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/Fred&quot;, &quot;Harrison&quot;&gt;</td>
</tr>
</tbody>
</table>
```

1.3 Implementation of NumRoomsWithOccupancy

Solution:

```cpp
int NumRoomsWithOccupancy(const type_b &dorms, const std::string &dorm, int count) {
    type_b::const_iterator dorm_itr = dorms.find(dorm);
    int answer = 0;
    if (dorm_itr != dorms.end()) {
        for (type_a::const_iterator room_itr = dorm_itr->second.begin();
             room_itr != dorm_itr->second.end(); room_itr++) {
            if (room_itr->second.size() == count)
                answer++;
        }
    }
    return answer;
}
```

1.4 Implementation of LookupRoomAssignment

Solution:

```cpp
void LookupRoomAssignment(const type_c &people, const std::string &first, const std::string &last, std::string &dorm, int &room) {
    type_c::const_iterator person_itr = people.find(std::make_pair(last,first));
    if (person_itr != people.end()) {
        dorm = person_itr->second.first;
        room = person_itr->second.second;
    } else {
        dorm = "";
    }
}
```
1.5 Implementation of PrintRoommates [ / 9 ]

Solution:

```cpp
void PrintRoommates(const type_b &dorms, const std::string &first,
                     const std::string &last, const std::string &dorm, int room) {
    type_b::const_iterator dorm_itr = dorms.find(dorm);
    assert (dorm_itr != dorms.end());
    type_a::const_iterator room_itr = dorm_itr->second.find(room);
    assert (room_itr != dorm_itr->second.end());
    if (room_itr->second.size() == 1) {
        std::cout << first << " has no roommates." << std::endl;
    } else {
        std::cout << first << ", roommate(s) are:
        for (int i = 0; i < room_itr->second.size(); i++) {
            if (room_itr->second[i] == std::make_pair(first,last)) continue;
            std::cout << " " << room_itr->second[i].first;
        }
        std::cout << std::endl;
    }
}
```

1.6 Order Notation [ / 6 ]

If the housing system contains $s$ students and $d$ dorms, with $r$ rooms per dorm, and an average/max of $k$ students per room, what is the running time for each of the functions above?

Solution:

- `NumRoomsWithOccupancy` quickly finds the specific dorm, but then must iterate over all of the rooms within the dorm. Accessing the size of a vector is constant. $O(\log(d) + r)$.
- `LookupRoomAssignment` quickly finds the person. $O(\log(s))$.
- `PrintRoommates` quickly finds the dorm, and the room within that dorm. Then we must iterate over the roommates. $O(\log(d) + \log(r) + k)$.

2 Updating Binary Search Tree Height [ / 17 ]

Alyssa P. Hacker has augmented our binary search tree with an additional member variable, the `height` of each node. She argues that this small change will allow `erase` to help minimize the overall tree height. She has defined a leaf node to have height = 1. Fill in the missing pieces to finish the implementation of `erase`.

```cpp
class Node {
    public:
    int value;
    int height;
    Node* left;
    Node* right;
};

bool erase(Node* &n, int v) {
    bool answer = true;
    if (n==NULL) { return false; }
    if (v < n->value) { answer = erase(n->left,v); }
    else if (v > n->value) { answer = erase(n->right,v); }
    else {
        if (n->right==NULL && n->left==NULL) { delete n; n=NULL; }
        else if (n->right == NULL) { Node* tmp = n->left; delete n; n=tmp; }
        else if (n->left == NULL) { Node* tmp = n->right; delete n; n=tmp; }
        else { // handle the case of 2 children
            if (n->left->height > n->right->height) {
                Node* tmp = n->left;
                while (tmp->right != NULL) { tmp = tmp->right; }
                n->value = tmp->value;
                answer = erase(n->left,tmp->value);
            } else {
                Node* tmp = n->right;
                while (tmp->left != NULL) { tmp = tmp->left; }
                n->value = tmp->value;
                answer = erase(n->right,tmp->value);
            }
        }
    }
    return answer;
```

Solution:

```cpp
if (n->left->height > n->right->height) {
    Node* tmp = n->left;
    while (tmp->right != NULL) { tmp = tmp->right; }
    n->value = tmp->value;
    answer = erase(n->left,tmp->value);
} else {
    Node* tmp = n->right;
    while (tmp->left != NULL) { tmp = tmp->left; }
```

```cpp```
3 Exactly Balanced Binary Search Trees Insertion Order [ / 20 ]

Draw an exactly balanced binary search tree that contains the characters: b, e, r, n, a, d, and i.

Solution:

\[
\text{In order for a binary tree to be exactly balanced, what must be true about } n, \text{ the number of elements?}
\]

\[
\text{Solution: } n \text{ must be equal to } 2^k - 1 \text{ for an integer } k
\]

Using the simple binary search tree insertion algorithm from lecture, give two different orderings for insertion of these elements that will result in an exactly balanced binary search tree.

Solution:

The nodes can be inserted in a breadth-first traversal ordering: e b n a d i r

Or in a pre-order depth-first traversal ordering: e b a d n i r

And other variations work too (as long as each node is inserted after its parent is inserted).

Suppose we have an STL set named data, whose size allows construction of an exactly balanced tree (as you specified earlier). Let’s write a recursive function named print_balanced_order that will print to std::cout an ordering of these elements for insertion to make an exactly balanced binary search tree.

print_balanced_order(data.begin(), data.end());

Solution:

```cpp
template <class T>
void print_balanced_order(typename std::set<T>::const_iterator a,
                          typename std::set<T>::const_iterator b) {
    if (a == b) return;
    typename std::set<T>::const_iterator tmp_a = a;
    typename std::set<T>::const_iterator tmp_b = b;
    tmp_b--;
    while (tmp_a != tmp_b) {
        tmp_a++;
        tmp_b--;
    }
    std::cout << " " << *tmp_a;
    tmp_b++;
    print_balanced_order(a, tmp_a);
    print_balanced_order(tmp_b, b);
}
```
4 Upside-Down Binary Search Tree [ / 36 ]

4.1 Binary Search Tree Diagram Warmup [ / 7 ]

Draw the tree that results when this sequence of 12 numbers is inserted (in this order) into a binary search tree using the algorithm covered in lecture and lab.

[Tree Diagram]

Which numbers are the leaves of this tree? *Hint: There are five.*

**Solution:** 2 6 10 16 23

Upside-Down Binary Search Tree

Ben Bitdiddle has come up with another wacky tree scheme (with questionable usefulness). He proposes to represent a binary search tree not with a single pointer to the tree root, but instead with an STL list of the leaf nodes. And then it follows that each *Node* will store only a pointer to its parent.

```cpp
class Node {
public:
  Node(int v) : value(v), parent(NULL) {}  
  int value;
  Node* parent;
};
```

Ben is sure that because his new representation only has one pointer per *Node* this structure will be much more memory efficient than the typical binary tree. Here’s how he proposes to construct the tree you drew above:

```cpp
std::list<Node*> leaves;
Insert(leaves,13); Insert(leaves,8); Insert(leaves,20); Insert(leaves,4);
Insert(leaves,18); Insert(leaves,25); Insert(leaves,2); Insert(leaves,10);
Insert(leaves,16); Insert(leaves,21); Insert(leaves,6); Insert(leaves,23);
assert (leaves.size() == 5);
```

4.2 BelongsInSubtree [ / 14 ]

Rather than jumping straight into the implementation of the *Insert* function, Alyssa P. Hacker suggests that Ben start by implementing the *BelongsInSubtree* function. This recursive function takes in two arguments: *node*, a pointer to a *Node* already in the upside down tree, and *value*, an element we would like to add. The function returns false if placing *value* within a subtree of *node* violates the binary search tree property of the whole tree and true otherwise. Note: Ben’s tree does not allow duplicate elements.

**Solution:**

```cpp
bool BelongsInSubtree(Node* node, int value) {
  if (node == NULL) return false;
  // check for duplicate
  if (node->value == value) return false;
  // made it to the root! this value fits on this branch
  if (node->parent == NULL) return true;
  // doesn't belong to the left of the grandparent
  if (node->value < node->parent->value && value > node->parent->value) return false;
  // doesn't belong in the right subtree of the grandparent
  if (node->value > node->parent->value && value < node->parent->value) return false;
  return BelongsInSubtree(node->parent,value);
}
```

The implementation of *Insert* will call the *BelongsInSubtree* function on each *Node* in the tree. Note: This function will return true for at least one, but possibly many nodes in the tree! Of these possible choices, *Insert* will select the
node that is furthest (in number of parent pointer links) from the root of the tree. For example, if we’d like to insert
the value 15 into our example tree, there are four nodes that will return true for the \texttt{BelongsInSubtree} function
above. What values are stored in those nodes? Which of these nodes will be selected by \texttt{insert} as the immediate
parent for ‘15’?

**Solution:** The Nodes storing 13, 20, 18, and 16 all return true.
15 will be placed as the left child of 16.

### 4.3 Destroy Tree [ / 15 ]

Now, let’s write the \texttt{DestroyTree} function, which cleans up all of the dynamically allocated memory associated with
Ben’s upside-down tree leaving a valid empty tree.

**Solution:**

```cpp
void DestroyTree(std::list<Node*> &leaves) {
    // use an STL set to collect all tree nodes (removes duplicates)
    std::set<Node*> nodes;
    for (std::list<Node*>::iterator itr = leaves.begin(); itr != leaves.end(); itr++) {
        Node* tmp = *itr;
        while (tmp != NULL) {
            if (!nodes.insert(tmp).second)
                break;
            tmp = tmp->parent;
        }
    }
    // now delete everything
    for (std::set<Node*>::iterator itr = nodes.begin(); itr != nodes.end(); itr++) {
        delete *itr;
    }
    // set the tree to the empty tree
    leaves.clear();
}
```

If the tree contains \( n \) elements, and is approximately balanced, what is the order notation of your implementation
of destroy tree? Write 2-3 sentences justifying your answer.

**Solution:** Walking from each leaf to root: \( \# \) of leaves * tree height = \( n/2 \) * \( \log(n) \) = \( O(n \log(n)) \). But
by checking the return value of set insertion, it’s only \( O(n) \) total set insertions. Each set insertion
costs \( \log(n) \), so it’s \( O(n \log(n)) \) to collect the nodes without duplicates. It only costs \( O(n) \) to iterate
over the nodes and delete them. Final answer: \( O(n \log(n)) \).

### 5 Halloween History Maps [ / 34 ]

The costume shop owner from Homework 7 has asked for help predicting what
costumes their indecisive customers might choose in the future. Looking at the
history of costume rentals they suspect there might be a pattern when a customer
changes their mind about their Halloween costume.

For the example data on the left, we can see two instances where a customer
switched from a pirate costume to a doctor costume, and only once did a customer
switch from a pirate costume to a zombie costume. Here is the output we expect
from the sample ‘r’ = \texttt{rental} and ‘h’ = \texttt{print costume history} commands:

```
history for pirate
next rental was doctor 1 time(s)
next rental was zombie 2 time(s)
no next rental history for elf
history for doctor
next rental was pirate 1 time(s)
```

The shop owner emphasizes the need for fast performance in this implementation, since the system will be handling
the records for thousands of customers and costumes in many different cities.

#### 5.1 Data Structure Sketch [ / 6 ]

Let’s store this data in two variables, one with the current customer information, and the second with the costume
history. (You’ll specify the typedefs in the next part). Sketch the contents of these variables after the rental commands
above. Follow the conventions from lecture for your diagrams.

<table>
<thead>
<tr>
<th>PEOPLE_TYPE</th>
<th>HISTORY_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice Jones</td>
<td>zombie</td>
</tr>
<tr>
<td>Bob Williams</td>
<td>doctor</td>
</tr>
<tr>
<td>Chris Smith</td>
<td>elf</td>
</tr>
</tbody>
</table>

Solution:

5.2 The typedefs [ / 4 ]

Next, fill in these typedef declarations.

Solution:

typedef std::map<std::string, std::string> PEOPLE_TYPE;
typedef std::map<std::string, std::map<std::string, int>> HISTORY_TYPE;

5.3 Implementation of the Rental Command [ / 9 ]

Now, complete the implementation:

```cpp
int main() {
    PEOPLE_TYPE people;
    HISTORY_TYPE history;
    std::string first, last, costume;
    char c;
    while (std::cin >> c) {
        if (c == 'r') {
            std::cin >> first >> last >> costume;
            Solution:
            std::string name = first + " " + last;
            PEOPLE_TYPE::iterator itr = people.find(name);
            if (itr != people.end() && itr->second != costume) {
                history[itr->second][costume]++;
            }
            people[name] = costume;
        }
    }
    NOTE: main function code continued on next page...
}
```

5.4 Implementation of the History Command [ / 10 ]

else {
    assert (c == 'h');
    std::cin >> costume;

Solution:

```cpp
HISTORY_TYPE::const_iterator itr = history.find(costume);
if (itr != history.end()) {
    std::cout << "history for " << costume << std::endl;
    std::map<std::string, int>::const_iterator itr2;
    for (itr2 = itr->second.begin(); itr2 != itr->second.end(); itr2++) {
        std::cout << " next rental was " << itr2->first << " " << itr2->second << " time(s)" << std::endl;
    }
} else {
    std::cout << "no next rental history for " << costume << std::endl;
}
```

}
5.5 Order Notation

If the shop has \( p \) customers, \( c \) costumes, and \( r \) total rental events, what is the order notation for performing a single rental (the 'r' command)? Write 1-2 sentences justification.

Solution: \( O(\log p) \) to find this customer in the people map. \( O(\log c) \) to find the old costume in the history map. \( O(\log c) \) to find the new costume in the interior history map. Overall: \( O(\log p + \log c) \).

What is the order notation for performing a history query (the 'h' command)? (justify your answer)

Solution: \( O(\log c) \) to the costume in the history map. \( O(c) \) to loop over all of the “next” costumes in the interior history map. Overall: \( O(c) \).

6 Allergic to for and while

Complete the functions below without using any additional for or while expressions. Given an STL vector of words, find all pairs of those words that share at least one common letter. For example, if \( \text{words} \) contains: apple boat cat dog egg fig then \( \text{common(words)} \) should return:

(\text{apple,boat}) (\text{apple,cat}) (\text{apple,egg}) (\text{boat,cat}) (\text{boat,dog}) (\text{dog,egg}) (\text{dog,fig}) (\text{egg,fig})

typedef std::set<std::pair<std::string,std::string> > set_of_word_pairs;

bool common(const std::string &a, const std::string &b) {
    for (int i = 0; i < a.size(); i++)
        for (int j = 0; j < b.size(); j++)
            if (a[i] == b[j]) return true;
    return false;
}

void common(set_of_word_pairs &answer, const std::vector<std::string>& words, int a, int b) {
    if (common(words[a],words[b])) {
        answer.insert(std::make_pair(words[a],words[b]));
    }
    if (b < words.size()-1)
        common(answer,words,a,b+1);
}

void common(set_of_word_pairs &answer, const std::vector<std::string>& words, int a) {
    if (a < words.size()-2) {
        common(answer,words,a+1);
    }
    common(answer,words,a,a+1);
}

set_of_word_pairs common(const std::vector<std::string>& words) {
    set_of_word_pairs answer;
    if (words.size() >= 2) {
        common(answer,words,0);
    }
    return answer;
}
Ben Bitdiddle Post-Breadth Tree Traversal

7.1 Balanced Tree Example

Ben Bitdiddle really wants to get his name on a traversal ordering. Even without a real world application for its use, he has invented what he calls the *post-breadth ordering*. His primary demonstration example is an exactly balanced, binary search tree with the numbers 1-15.

Your first task is to make a neat diagram of this tree in the box on the right.

For this example, Ben decrees that the `PrintPostBreadth` function should output:

```
LEVEL 0: 1 3 5 7 9 11 13 15
LEVEL 1: 2 6 10 14
LEVEL 2: 4 12
LEVEL 3: 8
```

Solution:

```
8
2 6
7 5 3 1
12
10 14
15 13 11 9
4
```

7.2 Un-Balanced Tree Example

Alyssa P. Hacker rolls her eyes at Ben but agrees to help him with the implementation. However, before tackling the implementation she wants to make sure that Ben’s idea is sound. She sketches the unbalanced tree shape on the left.

Your second task is to place the numbers 1-10 in this diagram so it is a proper binary search tree.

This unbalanced tree initially confuses Ben. But he thinks for a while and decides that for his new traversal ordering, level 0 is defined to be all of the leaves of the tree, level 1 is the parents of the leaves, level 2 is the grandparents, etc. So he decrees that for this second example, the output of the `PrintPostBreadth` function is:

```
LEVEL 0: 2 5 7 9
LEVEL 1: 3 4 8 10
LEVEL 2:  1 6
```

Solution:

```
2
4 6
1 3 5 7 8 10
```

Alyssa studies Ben’s sample output carefully and then asks Ben if the traversal ordering will ever contain repeated elements. Ben says no, each element in the structure should be output exactly once. Alyssa suggests that they add a boolean `mark` member variable to the `Node` class since it will be helpful for an efficient implementation. This flag will help ensure the traversal ordering does not contain duplicates.
7.3 CollectLeaves Implementation [ / 11 ]

Alyssa’s Node class is on the right.

She further suggests starting with the implementation of a helper function named CollectLeaves. This is a void recursive function that takes in two arguments: ptr is a pointer to a Node (initially the root of the tree), and leaves is an STL list of pointers to Nodes (the list is initially empty) that will collect all of the leaves of the tree.

She also indicates that this function should initialize all of the mark variables. Only the leaf nodes should be marked true.

Complete the implementation below.

Solution:

```cpp
class Node {
public:
    // CONSTRUCTOR
    Node(int v) : value(v), mark(false),
        left(NULL), right(NULL), parent(NULL) {}
    // REPRESENTATION
    int value;
    bool mark;
    Node* left;
    Node* right;
    Node* parent;
};

void CollectLeaves(Node *ptr, std::list<Node*>& leaves) {
    if (ptr == NULL) return;
    if (ptr->left == NULL && ptr->right == NULL) {
        ptr->mark = true;
        leaves.push_back(ptr);
    } else {
        ptr->mark = false;
        CollectLeaves(ptr->left, leaves);
        CollectLeaves(ptr->right, leaves);
    }
}
```

7.4 PrintPostBreadth Implementation [ / 14 ]

Now finish the implementation of the PrintPostBreadth function:

Solution:

```cpp
void PrintPostBreadth(Node *root) {
    // call the helper function
    std::list<Node*> current;
    CollectLeaves(root, current);
    int count = 0;
    while (current.size() > 0) {
        std::cout << "LEVEL " << count << " : ";
        // prepare a list of the parents of the current level
        std::list<Node*> next;
        for (std::list<Node*>::const_iterator itr = current.begin();
            itr != current.end(); itr++) {
            std::cout << " " << (*itr)->value;
            if ((*itr)->parent != NULL && !(*itr)->parent->mark) {
                next.push_back((*itr)->parent);
                (*itr)->parent->mark = true;
            }
        }
        // increment & switch to the new level
        count++;
        current = next;
    }
    std::cout << std::endl;
}
```
Louis B. Reasoner has taken a job at a genome sequencing startup working on algorithms to detect differences between the genomes of different species. He came up with the sketch of the data structure on the right and showed it to his manager and got approval to start implementation.

He's defined two typedefs named `count_t` and `kmer_t` to improve the readability of his code. Here's an example of how this data structure is constructed using the `Add` function:

```cpp
kmer_t kmers;
count_t totals;
Add(totals,kmers,"human","ACT"); Add(totals,kmers,"human","ACT");
Add(totals,kmers,"human","ACT"); Add(totals,kmers,"human","GAG");
Add(totals,kmers,"human","TAG"); Add(totals,kmers,"human","TAG");
Add(totals,kmers,"human","TAG"); Add(totals,kmers,"dog","ACT");
Add(totals,kmers,"dog","ACT"); Add(totals,kmers,"dog","TAG");
Add(totals,kmers,"dog","TAG"); Add(totals,kmers,"fruit fly","ACT");
Add(totals,kmers,"fruit fly","ACT"); Add(totals,kmers,"fruit fly","GAG");
```

Two of the key operations for this data structure are to query the number of matches of a given k-mer for a particular species and to find the most frequently occurring k-mer for a species. Here are several example usages of the `Query` and `MostCommon` functions:

```cpp
assert (Query(kmers,"human","ACT") == 3); assert (MostCommon(kmers,"human") == "TAG");
assert (Query(kmers,"human","CAT") == 0); assert (MostCommon(kmers,"fruit fly") == "ACT");
assert (Query(kmers,"human","TAG") == 4); assert (MostCommon(kmers,"cat") == "");
assert (Query(kmers,"cat","ACT") == 0);
assert (Query(kmers,"dog","GAG") == 1);
```

Finally, we can compute the difference between two species. The k-mer fraction is the percent of a species total k-mers that match the particular k-mer. The k-mer difference is the absolute value of the difference between the k-mer fractions for each of the species. And the overall difference between two species is the sum over all k-mers of the k-mer difference. Here is the math to calculate the difference between a human and a dog:

```
ACT: abs(2/5 - 3/9) = 0.067
CAT: = 0.000
GAG: abs(1/5 - 2/9) = 0.022
TAG: abs(2/5 - 4/9) = 0.044
overall: = 0.133
```

Here is code to call the `Difference` helper function:

```cpp
std::cout << "Difference between human & dog 
 " << Difference(totals,kmers,"human","dog") << std::endl;
std::cout << "Difference between human & fruit fly 
 " << Difference(totals,kmers,"human","fruit fly") << std::endl;
std::cout << "Difference between dog & fruit fly 
 " << Difference(totals,kmers,"dog","fruit fly") << std::endl;
```

And the resulting output:

```
Difference between human & dog 0.133
Difference between human & fruit fly 0.889
Difference between dog & fruit fly 0.800
```
8.1 The typedefs [4]

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

Solution:

typedef std::map<std::string,int> count_t;
typedef std::map<std::string,count_t> kmer_t;

8.2 Add Implementation [7]

Next, finish the implementation of the Add function.

Solution:

void Add(count_t& totals, kmer_t& kmers, const std::string& species, const std::string& kmer) {
    totals[species]++;
    kmers[kmer][species]++;
}

If the data structure contains \( s \) different species, and \( k \) unique k-mers, and each animal contains \( p \) total k-mers, what is the order notation for the running time of a single call to Add? Write 2-3 concise and well-written sentences justifying your answer.

Solution: The first operator[] costs \( O(\log s) \) because there are \( s \) species in the map. The second operator[] costs \( O(\log k) \) because there are \( k \) unique k-mers in the outer k-mers map. The third operator[] costs \( O(\log s) \) because there are \( s \) species in the inner k-mers map. These quantities are simply added together (we don’t search every inner map, just one!). Overall: \( O(\log s + \log k) \).

8.3 Query Implementation [6]

Solution:

int Query(const kmer_t& kmers, const std::string &species, const std::string& kmer) {
    kmer_t::const_iterator itr = kmers.find(kmer);
    if (itr == kmers.end())
        return 0;
    count_t::const_iterator itr2 = itr->second.find(species);
    if (itr2 == itr->second.end())
        return 0;
    return itr2->second;
}

8.4 MostCommon Implementation [7]

Solution:

std::string MostCommon(const kmer_t& kmers, const std::string &species) {
    std::string answer = "";
    int count = -1;
    for (kmer_t::const_iterator itr = kmers.begin(); itr != kmers.end(); itr++) {
        count_t::const_iterator itr2 = itr->second.find(species);
        if (itr2 != itr->second.end() &&
            (answer == "" || count < itr2->second)) {
            answer = itr->first;
            count = itr2->second;
        }
    }
    return answer;
}
8.5 Difference Implementation [12]

Solution:

```cpp
float Difference(const count_t& totals, const kmer_t& kmers, const std::string& speciesA, const std::string& speciesB) {
    float diff = 0;
    count_t::const_iterator itrA = totals.find(speciesA);
    count_t::const_iterator itrB = totals.find(speciesB);
    if (itrA == totals.end() || itrB == totals.end()) {
        std::cerr << "ERROR! One or both species are unknown" << std::endl;
        return -1;
    }
    float totalA = itrA->second;
    float totalB = itrB->second;
    for (kmer_t::const_iterator itr = kmers.begin(); itr != kmers.end(); itr++) {
        int countA = Query(kmers, speciesA, itr->first);
        int countB = Query(kmers, speciesB, itr->first);
        diff += fabs(countA/float(totalA)-countB/float(totalB));
    }
    return diff;
}
```

If the data structure contains \(s\) different species, and \(k\) unique k-mers, and each animal contains \(p\) total k-mers, what is the order notation for the running time of a single call to `Difference`? Write 2-3 concise and well-written sentences justifying your answer.

Solution: Finding the two species in the totals map is \(O(\log s)\). We loop over all \(k\) unique k-mers, and for each of them (multiplication) we look up the species in the inner map. If we use `Query` (we did above), this will be \(O(\log s + \log k)\). If instead we inline a portion of this function (more code, but faster) it will be \(O(\log s)\).

Overall (w/ `Query`): \(O(\log s + k \times (\log s + \log k))\), which simplifies to \(O(k \times (\log s + \log k))\).
Overall (w/o `Query`): \(O(\log s + k \times \log s)\), which simplifies to \(O(k \times \log s)\).

9 Prescribed Pre-Ordering [21]

In this problem we will create an algorithm to construct a binary search tree from the desired pre-order traversal order. The driver function (below) takes in this sequence as a STL vector. If the contents of the vector is not a valid pre-order traversal order of a binary search tree, the function should return NULL.

```cpp
template <class T> class Node {
public:
    Node(T v) : value(v), left(NULL), right(NULL) {}    
    T value;
    Node* left;
    Node* right;
};

template <class T> void destroy(Node<T>* root) {
    if (root == NULL) return;
    destroy(root->left);
    destroy(root->right);
    delete root;
}

// "driver" function (starts the recursive function that does the actual work)
template <class T> Node<T>* MakePreOrderTree(const std::vector<T>& values) {
    if (values.size() == 0) return NULL;
    return MakePreOrderTree(values, 0, values.size()-1);
}
```
9.1 Test Cases

First, create 4 different test cases of input for this problem. Each input vector should contain the numbers 1-7. The first two should be valid pre-orderings for a binary search tree containing these 7 numbers. **Draw the corresponding tree for these cases.** The other two test case inputs should be invalid pre-orderings.

valid: 4 2 1 3 6 5 7  
valid: 2 1 3 4 5 6 7

Solution: (many correct answers!)

invalid: 5 4 3 6 2 1 7  invalid: 4 6 5 7 2 1 3

9.2 Finish the MakePreOrderTree Implementation

Note: If you discover the input sequence is an invalid pre-ordering for a binary search tree, make sure you do not leak any memory!

Solution:

```cpp
template <class T>
Node<T>* MakePreOrderTree(const std::vector<T>& values, int start, int end) {
    assert (start <= end);
    // find the split between the left & right branches
    int split = start+1;
    // the split is the first element that is greater than the "root"
    while (split <= end && values[split] < values[start]) {
        split++;
    }
    // check that all elements after the split are also greater than the "root"
    for (int i = split; i <= end; i++) {
        if (values[i] < values[start]) {
            // failure
            return NULL;
        }
    }
    // make the new node
    Node<T>* answer = new Node<T>(values[start]);
    // if there is at least one node to the left, recurse left
    if (start+1 <= split-1) {
        answer->left = MakePreOrderTree(values,start+1,split-1);
        // if the left tree is NULL (failure), cleanup
        if (answer->left == NULL) {
            destroy(answer);
            return NULL;
        }
    }
    // if there is at least one node to the right, recurse right
    if (split <= end) {
        answer->right = MakePreOrderTree(values,split,end);
        // if the right tree is NULL (failure), cleanup
        if (answer->right == NULL) {
            destroy(answer);
            return NULL;
        }
    }
    return answer;
}
```
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.

```cpp
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;
};
```

### 10.1 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:**

![Tree Diagrams]

- **count(root) => <3,1>**
- **count(root) => <5,1>**
- **count(root) => <4,0>**
- **count(root) => <3,2>**
- **count(root) => <3,1>**
- **count(root) => <3,1>**

### 10.2 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:**

```cpp
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:

```cpp
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);
}
```

```cpp
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);
}
```

### 10.3 Implement erase for Trees with Unoccupied Nodes

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:**

```cpp
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;
    }
}
```

### 10.4 Implement insert for Trees with Unoccupied Nodes

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:

```cpp
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;
    }
}
```

11 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 maps would be a great fit for this application. Here's a portion of the main function with an example of how his program works:

```cpp
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:

```cpp
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 typedefs 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:

```cpp
success = make_reservation(rr, "DCC", 308, "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
```

### 11.1 The typedefs [ / 5 ]

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

**Solution:**

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

### 11.2 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.
11.3 Implementing `make_reservation`

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:**

```cpp
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++) {
    // check if the time is available
    if (day_itr->second[start_time + i] || day_itr->second[start_time + i + 1]) {
      std::cerr << "ERROR! time conflict at " << start_time + i << "-" << start_time + i + 1 << std::endl;
      return false;
    }
  }
  return true;
}
```
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;

11.4 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 \( b \times c \) entries. To locate the specific room is \( O(\log (b \times c)) \). 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 \( O(\log (b \times c) + \log d + s) \). We will also accept \( O(\log(b \times c) + s) \).

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 \( b \times c \) 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: \( b \times c \times (32 + 4 + d \times (32 + t \times 32)) = 36 \times b \times c \) (memory to store each building & room pair) + \( 32 \times d \times b \times c \) (memory to store the days of the week strings) + \( 32 \times d \times t \times b \times c \) (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 \( b \times c \) 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 \( O(b \times c \times (\log d + s)) \). We will also accept \( O(b \times c \times s) \).

12 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 strings representing different types of clothing. colors is an STL list of STL sets of STL strings representing the different colors of each item of clothing. Your function should return an STL vector of STL strings describing each unique outfit (in any order) that can be created from these items of clothing.

Here is a small example:

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

```c++
red hat & red shirt & blue pants
red hat & green shirt & blue pants
red hat & white shirt & blue pants
red hat & white shirt & black pants
red hat & green shirt & black pants
red hat & green 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;
}