1 Dorm Assignment Maps [ / 45 ]

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
```

```cpp
type_b dorms;
type_c people;
std::ifstream istr("dorm_data.txt");
std::string dorm, first, last;
int room;
while (istr >> dorm >> room >> first >> last) {
    dorms[dorm][room].push_back(std::make_pair(first,last));
    people[std::make_pair(last,first)] = std::make_pair(dorm,room);
}
```

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

```cpp
void Info( /*?*/ dorms, /*?*/ people, /*?*/ first, /*?*/ last) {
    std::string dorm;
    int room;
    LookupRoomAssignment(people,first,last,dorm,room);
    if (dorm == ") {"n"
        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>People</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barton</td>
<td>201</td>
<td>&lt;'Sally', 'Morris'&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;'Alice', 'Williams'&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;'Jessica', 'Smith'&gt;</td>
</tr>
<tr>
<td>Crockett</td>
<td>106</td>
<td>&lt;'Chris', 'Thompson'&gt;</td>
</tr>
<tr>
<td></td>
<td>320</td>
<td>&lt;'Beth', 'Collins'&gt;</td>
</tr>
<tr>
<td>Nugent</td>
<td>112</td>
<td>&lt;'Erin', 'Lee'&gt;</td>
</tr>
<tr>
<td></td>
<td>316</td>
<td>&lt;'Kathy', 'Newton'&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;'George', 'Davis'&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;'Fred', 'Harrison'&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
def 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 << "'s 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 Recursive Subtree Matching [ / 15 ]

In this problem we will compare the shape (not values) of binary trees.

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

First, write a function named `shape_match` that takes in two `Node` pointers and returns true if those trees have the same shape. For example, `shape_match(a, b)` returns true and `shape_match(a, c)` returns false.

**Solution:**

```cpp
bool shape_match(Node* a, Node* b) {
    if (a == NULL && b == NULL) return true;
    if (a == NULL) return false;
    if (b == NULL) return false;
    if (!shape_match(a->left, b->left)) return false;
    if (!shape_match(a->right, b->right)) return false;
    return true;
}
```

Next, write a function named `find_subtree_match` that takes a pointer to a large tree and a pointer to a pattern and returns a pointer to a node anywhere within the tree that matches the pattern, or NULL if there is no match. For example `find_subtree_match(c, a)` will return a pointer to the node storing 14.
Solution:
Node* find_subtree_match(Node* large, Node* pattern) {
    if (shape_match(large, pattern)) return large;
    if (large == NULL) return NULL;
    if (find_subtree_match(large->left, pattern) != NULL)
        return large->left;
    return find_subtree_match(large->right, pattern);
}

3 Updating Binary Search Tree Height

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
            Solution:
            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);
            }
            assert (answer);
        }
    }
    if (n != NULL) { // update the height
        Solution:
        int l = 0;
        int r = 0;
        if (n->left != NULL) l = n->left->height;
        if (n->right != NULL) r = n->right->height;
        n->height = std::max(l + 1, r + 1);
    }
    return answer;
}
```
Draw an exactly balanced binary search tree that contains the characters: b, e, r, n, a, d, and i.

In order for a binary tree to be exactly balanced, what must be true about \( n \), the number of elements?

Solution:
\( n \) must be equal to \( 2^k - 1 \) 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.

\begin{verbatim}
print_balanced_order(data.begin(),data.end());
\end{verbatim}

Solution:

\begin{verbatim}
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);
}
\end{verbatim}