CSCI-1200 Data Structures
Test 3 — Practice Problem Solutions

1 Intensely Overloading on TA Office Hours

Cameron (who helped develop YACS) is thinking about an extension to assist students in maximizing use of TA office hours for their registered classes. He has access to the following tables of data from the registrar using STL maps and vectors (shown here with sample data):

<table>
<thead>
<tr>
<th>students</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben</td>
</tr>
<tr>
<td>Louis</td>
</tr>
<tr>
<td>Abby</td>
</tr>
<tr>
<td>Eric</td>
</tr>
<tr>
<td>Milo</td>
</tr>
<tr>
<td>Ryan</td>
</tr>
<tr>
<td>Sinclair</td>
</tr>
<tr>
<td>Tim</td>
</tr>
<tr>
<td>Wendy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ta_assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abby cs1</td>
</tr>
<tr>
<td>Eric calc1</td>
</tr>
<tr>
<td>Milo ds</td>
</tr>
<tr>
<td>Ryan bio</td>
</tr>
<tr>
<td>Sinclair cs1</td>
</tr>
<tr>
<td>Tim physics1</td>
</tr>
<tr>
<td>Wendy ds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>office_hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abby Monday 3pm, AE Thursday 2pm, Folsom</td>
</tr>
<tr>
<td>Cameron Friday 4pm, Sage</td>
</tr>
<tr>
<td>Eric Tuesday 5pm, J−Rowl</td>
</tr>
<tr>
<td>Milo Thursday 4pm, Folsom Wednesday 6pm, Folsom</td>
</tr>
<tr>
<td>Ryan Thursday 4pm, Folsom</td>
</tr>
<tr>
<td>Sinclair Thursday 4pm, Folsom</td>
</tr>
<tr>
<td>Wendy Thursday 4pm, Folsom</td>
</tr>
</tbody>
</table>

Cameron proposes we write a function GetHelp that takes in these 3 tables of data, and the name of a student, and returns an STL set of strings with the day, time, and location of all relevant office hours. For example, by printing the output of:

```
GetHelp(ta_assignments,students,office_hours,"Ben");
```

We will learn that Ben should attend the following office hours:

- Thursday 4pm, Folsom
- Tuesday 5pm, J−Rowl
- Wednesday 6pm, Folsom

1.1 Just the typedefs, Ma’am

After looking ahead through the rest of this problem, let’s define a few helpful typedefs:

**Solution:**

```cpp
typedef std::map<std::string,std::string> type_t;
typedef std::map<std::string,std::vector<std::string> > type_s;
typedef std::map<std::string,std::vector<std::string> > type_o;
```

1.2 Let’s get Ben some help in Office Hours!

Write the GetHelp function below. Perform some basic error checking. We suggest you use assert to verify any assumptions as you work through the implementation.

```cpp
std::set<std::string> GetHelp(const type_t &ta_assignments, const type_s &students, const type_o &office_hours, const std::string &student) {
```
Solution:

// efficiently find the correct student
type_s::const_iterator s_itr = students.find(student);
assert (s_itr != students.end());
std::set<std::string> answer;

// loop over all of the student's courses
for (int i = 0; i < s_itr->second.size(); i++) {
    // loop over all TAs (unfortunately, we can't be more efficient)
    type_t::const_iterator t_itr = ta_assignments.begin();
    while (t_itr != ta_assignments.end()) {
        // if the TA matches the course...
        if (t_itr->second == s_itr->second[i]) {
            // loop over all TAs hours
            type_o::const_iterator o_itr = office_hours.find(t_itr->first);
            if (o_itr != office_hours.end()) {
                // and add them to the set
                // NOTE: we don't need to worry about duplicates with a set!
                answer.insert(o_itr->second[i]);
            }
        }
        t_itr++;
    }
}

return answer;

1.3 What Would Milo Do? [ / 12 ]

Milo notices someone forgot their laptop charger in office hours. In addition to posting a message on the Submitty Discussion Forum, Milo would like to message students in the other courses that share his assigned office hour times and locations. Let’s write another function, GetShared, that returns an STL set of the names of other courses that share an office hour time and location with the specified TA. For example, if we print the return value of:

GetShared(ta_assignments, office_hours, "Milo");

We will learn that Milo’s Data Structures office hours overlap with these other courses:

bio

cs1

t::set<std::string> GetShared(const type_t &ta_assignments,
const type_o &office_hours,
const std::string &ta) {

Solution:

std::set<std::string> answer;
// make sure the TA is assigned to a course
type_t::const_iterator a_itr = ta_assignments.find(ta);
assert (a_itr != ta_assignments.end());
// make sure TA is assigned to office hours
type_o::const_iterator to_itr = office_hours.find(ta);
assert (to_itr != office_hours.end());
// loop over the specific TAs office hours
for (int i = 0; i < to_itr->second.size(); i++) {
    // and ALL assigned office hours
    for (type_o::const_iterator o_itr = office_hours.begin(); o_itr != office_hours.end(); o_itr++) {
        // skip this TA
        if (o_itr->first == ta) continue;
        // looking for matches with this timeslot
        if (o_itr->second[i] == to_itr->second[i]) {
            type_t::const_iterator b_itr = ta_assignments.find(o_itr->first);
        }
    }
}
assert (b_itr != ta_assignments.end());
// skip other TAs also assigned to the specific TAs course
if (b_itr->second != a_itr->second)
    answer.insert(b_itr->second);
}
}
return answer;

1.4 Who’s Afraid of the Big Bad ‘O’ Notation? [ / 7 ]

What is the running time for each of the functions you wrote? Let’s say that we have $s$ students, each student takes on average $k$ courses, the university has $t$ TAs, each TA is assigned on average to $h$ office hour slots per week, the university has $c$ total courses, and the output contains $x$ values. Justify your answers.

Solution:

GetHelp $O(\log(s) + k \times t \times (\log(t) + h) + x \log(x) )$. We are able to quickly find the student, and quickly find a specific TAs office hours, but because of a poor table design we have to linearly walk over all TAs to find those assigned to each course. We also linearly walk over the office hours, and building the answer set will take $O(x \log(x) )$ time. The final term is likely small and can probably be ignored.

GetShared $O(\log(t) + h \times t \times h \times \log(t) + x \log(x) )$. The expensive part is looping over all other TAs and all of their office hour assignments and comparing with the specified TA’s hours. At some point we need to check if they are TAing the same course. This is written as a big product but is likely not this expensive in practice due to the if statements – unless the overlap in office hours is very large and a large fraction of the university TAs are covering the same course. Building the answer set is will take $O(x \log(x) )$ time, but is small and can probably be ignored. Final answer: $O( h^2 \times t \times \log(t) )$.

1.5 Making some Mappy Improvements [ / 3 ]

Is this the best organization of data to accomplish these tasks? Describe a minor change to one of these tables (still sticking with STL maps and vectors) that will simplify the algorithm(s) you wrote above and/or improve the big ‘O’ notation. Write 2-3 sentences describing the change and why it’s an improvement.

Solution: If we flipped the key and value of the ta_assignments table to go from course to TA (or stored both versions of the table or used a bi-directional map), then the GetHelp function would be dramatically improved. We wouldn’t need to linearly search over the table for who TA’ed a specific class. We could do a log search instead. The running time would then be $O(\log(s) + k \times (\log(t) + h) )$.

1.6 Are Hash Tables always better than Binary Search Trees? [ / 3 ]

If we switched the maps and sets in this problem to be hash tables, what would be one advantage/improvement? What would be one disadvantage/loss? Write 2 concise, technical, and well-written sentences.

Solution: If we switched to hash tables, and had good hash functions and not the worst case performance, all of the log terms in the running time would be eliminated and replaced with $O(1)$. However, we lose the automatic sorting of the output, and would need to separately sort the output if desired. Note: Sorting the days of the week alphabetically is confusing and impractical!

2 I am the Lorax, I speak for the Trees [ / 36 ]

In this problem we will inspect the structure of a memory diagram composed of Node objects, as declared on the right. We will decide if the current arrangement, contents, and coloring of nodes is a valid, well-balanced, binary search tree. Note: “red” nodes are visualized white. On the next two pages you will complete the implementation of the recursive functions used in the fragment of code below:

```cpp
template <class T> class Node {
public:
    T value;
    bool is_black;
    std::vector<Node*> children;
};
```
if (!is_tree(root)) std::cout << "NOT A TREE\n";
else if (!is_binary(root)) std::cout << "NOT A BINARY TREE\n";
else if (!is_bst(root)) std::cout << "NOT A BINARY SEARCH TREE\n";
else if (!is_red_black(root)) std::cout << "NOT A RED BLACK TREE\n";
else std::cout << "you're a well-balanced binary search tree!\n";

## 2.1 Diagnostics [ / 8 ]

For each of the 4 diagrams below, write the statement that will print when the diagram is passed to the code fragment above. *Hint: Each of the four "NOT A ..." phrases will be used exactly once.*

![Diagram 1](image1.png)

Solutions:
- NOT A BINARY SEARCH TREE
- NOT A RED BLACK TREE
- NOT A TREE
- NOT A BINARY TREE

Now, draw a diagram that contains the integers 1-9 that will print the statement “you’re a well-balanced binary search tree!”

![Diagram 2](image2.png)

Solution: (and many other correct solutions)

## 2.2 is_tree Implementation [ / 9 ]

Solution:
```cpp
template <class T> bool is_tree(Node<T> *n, std::set<Node<T>*> &already_seen) {
    if (n == NULL) return true;
    if (already_seen.find(n) != already_seen.end()) return false;
    already_seen.insert(n);
    for (int i = 0; i < n->children.size(); i++) {
        if (!is_tree(n->children[i],already_seen))
            return false;
    }
    return true;
}

template <class T> bool is_tree(Node<T> *n) {
    std::set<Node<T>*> already_seen;
    return is_tree(n,already_seen);
}
```
2.3 is_binary Implementation

template <class T> bool is_binary(Node<T> *n) {

Solution:
if (n == NULL) return true;
if (n->children.size() != 2) return false;
for (int i = 0; i < n->children.size(); i++) {
    if (!is_binary(n->children[i]))
        return false;
}
return true;
}

2.4 is_bst Implementation

template <class T>
bool is_bst(Node<T> *n, Node<T> *lower=NULL, Node<T> *upper=NULL) {

Solution:
if (n == NULL) return true;
if (lower != NULL && n->value < lower->value) return false;
if (upper != NULL && n->value > upper->value) return false;
if (!is_bst(n->children[0],lower,n)) return false;
if (!is_bst(n->children[1],n,upper)) return false;
return true;
}

2.5 is_red_black Implementation

template <class T> bool double_red(Node<T> *n) {

Solution:
if (n == NULL) return false;
if (!n->is_black) {
    if (n->children[0] != NULL && !n->children[0]->is_black) return true;
    if (n->children[1] != NULL && !n->children[1]->is_black) return true;
}
return double_red(n->children[0]) || double_red(n->children[1]);
}

template <class T> int black_count(Node<T> *n) {

Solution:
if (n == NULL) return 1;
int left = black_count(n->children[0]);
int right = black_count(n->children[1]);
if (left == -1 || right == -1 || left != right) return -1;
if (n->is_black) return left+1;
return left;
}

template <class T> bool is_red_black(Node<T> *n) {
if (double_red(n)) return false;
if (black_count(n) == -1) return false;
return true;
}
Write a function named `construct_breadth` that takes in an STL list of STL strings and creates and returns a pointer to a well-balanced binary tree of Nodes with breadth-first traversal order that matches the input.

Solution:
```cpp
Node* construct_breadth(std::list<std::string> input) {
    // note: input passed by copy since the editing below
    // these changes should not be permanent
    if (input.size() == 0) return NULL;
    Node* answer = new Node(input.front());
    input.pop_front();
    // store the current leaves (nodes with no children)
    std::list<Node*> leaves;
    leaves.push_back(answer);
    while (input.size() > 0) {
        // add a left child to the 'next' leaf
        Node* tmp = leaves.front();
        leaves.pop_front();
        tmp->left = new Node(input.front());
        input.pop_front();
        leaves.push_back(tmp->left);
        if (input.size() > 0) {
            // also add a right child
            tmp->right = new Node(input.front());
            input.pop_front();
            leaves.push_back(tmp->right);
        }
    }
    return answer;
}
```

Give an input list of 10 RPI building names that when passed to `construct_breadth` will create a valid binary search tree. Also, draw the produced tree.

Solution:
sage, empac, union, commons, ricketts, troy, westhall, armory, darrin, lally
(and many other correct answers)

![Breadth-First Construction](image)

Give an input list of 10 RPI building names that when passed to `construct_breadth` will create a valid binary search tree. Also, draw the produced tree.

Solution:
sage, empac, union, commons, ricketts, troy, westhall, armory, darrin, lally
(and many other correct answers)

![Breadth-First Construction](image)

4 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
type_b dorms;
type_c people;
```

<table>
<thead>
<tr>
<th>Dorm</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crockett 320</td>
<td>Beth Collins</td>
</tr>
<tr>
<td>Barton 201</td>
<td>Sally Morris</td>
</tr>
<tr>
<td>Barton 201</td>
<td>Alice Williams</td>
</tr>
<tr>
<td>Barton 201</td>
<td>Jessica Smith</td>
</tr>
<tr>
<td>Nugent 316</td>
<td>George Davis</td>
</tr>
<tr>
<td>Nugent 316</td>
<td>Fred Harrison</td>
</tr>
<tr>
<td>Crockett 106</td>
<td>Chris Thompson</td>
</tr>
<tr>
<td>Nugent 112</td>
<td>Erin Lee</td>
</tr>
<tr>
<td>Nugent 112</td>
<td>Kathy Newton</td>
</tr>
</tbody>
</table>
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:
(the parameter types have been omitted)

```cpp
void Info( /*?*/ dorms, /*?*/ people, /*?*/ first, /*?*/ 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:

```
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.

4.1 The typedefs [ / 6 ]

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

Solution:

```cpp
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;
```

4.2 Data Structure Sketch [ / 7 ]

Sketch the contents of the dorms variable for the sample input text file. Follow the conventions from lecture for this diagram.
4.3 Implementation of NumRoomsWithOccupancy [ / 8 ]

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;
}
```

4.4 Implementation of LookupRoomAssignment [ / 9 ]

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 = "";
    }
}
```

4.5 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 << "'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;
}

4.6 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. $O(\log(d) + \log(r) + k)$.

5 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.

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;

6 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.

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.

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);
}
```

7 Upside-Down Binary Search Tree [ / 36 ]

7.1 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.

13 8 20 4 18 25 2 10 16 21 6 23
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);
```

7.2 BelongsInSubtree

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 `BelongsToInSubtree` function above. What values are stored in those nodes? Which of these nodes will be selected by `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.

7.3 Destroy Tree [ 15 ]

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

Solution:

```c++
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)).

8 Halloween History Maps [ 34 ]

| r Bob Williams pirate | r Chris Smith doctor |
| r Chris Smith doctor | r Chris Smith zombie |
| r Alice Jones zombie | r Chris Smith pirate |
| r Bob Williams pirate | r Chris Smith zombie |
| r Chris Smith elf | r Bob Williams doctor |

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’ = rental and ‘h’ = 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.

8.1 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>

<table>
<thead>
<tr>
<th>doctor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pirate</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>zombie</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>elf</td>
<td>1</td>
</tr>
<tr>
<td>pirate</td>
<td>1</td>
</tr>
</tbody>
</table>

Solution:

8.2 The typedefs [ 4 ]

Next, fill in these typedef declarations.

Solution:

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

8.3 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...
}
```

8.4 Implementation of the History Command [ 10 ]

```cpp
else {
    assert (c == 'h');
    std::cin >> costume;
    Solution:
    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;
    }
}
```
8.5 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: \( O(c) \).

9 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 words contains: apple boat cat dog egg fig then 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) {
    Solution:
    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) {
    Solution:
    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) {
    Solution:
    set_of_word_pairs answer;
    if (words.size() >= 2) {
        common(answer,words,0);
    }
    return answer;
}