CSCI-1200 Data Structures — Fall 2018
Test 3 — Solutions

1 Intensely Overloading on TA Office Hours [ 41 ]

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>ta_assignments</th>
<th>office_hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abby</td>
<td>Monday 3pm, AE</td>
</tr>
<tr>
<td>Eric</td>
<td>Thursday 2pm, Folsom</td>
</tr>
<tr>
<td>Milo</td>
<td>Monday 4pm, J-Rowl</td>
</tr>
<tr>
<td>Ryan</td>
<td>Wednesday 6pm, Folsom</td>
</tr>
<tr>
<td>Sinclair</td>
<td>Thursday 4pm, Folsom</td>
</tr>
<tr>
<td>Tim</td>
<td>Thursday 4pm, Folsom</td>
</tr>
<tr>
<td>Wendy</td>
<td>Thursday 4pm, Folsom</td>
</tr>
<tr>
<td>Ben</td>
<td>bio calc1 ds psych</td>
</tr>
<tr>
<td>Louis</td>
<td>calc2 cs1 econ physics</td>
</tr>
<tr>
<td>students</td>
<td></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 [ 3 ]

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! [ 13 ]

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:

```cpp
// 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 of the TAs hours
            type_o::const_iterator o_itr = office_hours.find(t_itr->first);
            if (o_itr != office_hours.end()) {
                for (int j = 0; j < o_itr->second.size(); j++) {
                    // and add them to the set
                    // NOTE: we don't need to worry about duplicates with a set!
                    answer.insert(o_itr->second[j]);
                }
            }
        } else {
            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:

```cpp
GetShared(ta_assignments,office_hours,"Milo");
```

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

```cpp
bio
cs1
```

std::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);
                } else {
                    o_itr++;
                }
            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 $O(\log(s) + k \cdot t \cdot (\log(t) + h) + x \log(x))$. 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 $O(\log(t) + h \cdot t \cdot h \cdot \log(t) + x \log(x))$. 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: $O(h^2 \cdot t \cdot \log(t))$.

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 $O(\log(s) + k \cdot (\log(t) + h))$.

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 Diagrams [ / 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.

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!”

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 \textbf{is\_binary} Implementation [ \hspace{1em} / 6 \hspace{1em} ]

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

2.4 \textbf{is\_bst} Implementation [ \hspace{1em} / 6 \hspace{1em} ]

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

2.5 \textbf{is\_red\_black} Implementation [ \hspace{1em} / 7 \hspace{1em} ]

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

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

\begin{verbatim}
template <class T> bool is_red_black(Node<T> *n) {

Solution:
if (double_red(n)) return false;
if (black_count(n) == -1) return false;
return true;
}
\end{verbatim}
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.

```cpp
class Node {
public:
    Node(const std::string &v) { value=v; left=right=NULL; }
    std::string value;
    Node* left;
    Node* right;
};

Solution:
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)

```
sage
  
empac union
  
commons ricketts troy westhall
  
armory darrin lally
```