CSCI-1200 Data Structures
Test 3 — Practice Problems

Note: This packet contains practice problems from three previous exams. Your exam will contain approximately one third as many problems.

1 Un-Occupied Erase [ / 39 ]

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.

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; 
};

1.1 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.
1.2 Counting Occupied & Unoccupied Nodes

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.

```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);
}

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);
}
```

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);
}
```
1.3 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.

*sample solution: 28 line(s) of code*
1.4 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.

*sample solution: 25 line(s) of code*
2 Classroom Scheduler Maps

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 `typedef`s 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
```
2.1 The typedefs

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

```cpp
typedef week_schedule;
```

```cpp
typedef room_reservations;
```

2.2 Diagram of the data stored in room_reservations rr

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.
2.3 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.
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.

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.

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.
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" } }

red hat & red shirt & blue pants
red hat & green shirt & blue pants
red hat & white shirt & blue pants
red hat & red shirt & black pants
red hat & green shirt & black pants
red hat & white shirt & black pants
```

sample solution: 22 line(s) of code
4 Spicy Chronological Sets using Maps

Ben Bitdiddle is organizing his spice collection using an STL set but runs into a problem. He needs the fast find, insert, and erase of an STL set, but in addition to organizing his spices alphabetically, he also needs to print them out in chronological order (so he can replace the oldest spices).

Ben is sure he’ll have to make a complicated custom data structure, until Alyssa P. Hacker shows up and says it can be done using an STL map. She quickly sketches the diagram below for Ben, but then has to dash off to an interview for a Google summer internship.

Alyssa’s diagram consists of 3 variables. The first variable, containing most of the data, is defined by a typedef. Even though he’s somewhat confused by Alyssa’s diagram, Ben has pushed ahead and decided on the following interface for building his spice collection:

```cpp
chrono_set cs;
std::string oldest = "";
std::string newest = "";
insert(cs,oldest,newest,"garlic");
insert(cs,oldest,newest,"oregano");
insert(cs,oldest,newest,"nutmeg");
insert(cs,oldest,newest,"cinnamon");
insert(cs,oldest,newest,"basil");
insert(cs,oldest,newest,"sage");
insert(cs,oldest,newest,"dill");
```

Ben would like to output the spices in 3 ways:

**ALPHA ORDER:** basil cinnamon dill garlic nutmeg oregano sage  
**OLDEST FIRST:** garlic oregano nutmeg cinnamon basil sage dill  
**NEWEST FIRST:** dill sage basil cinnamon nutmeg oregano garlic

If he buys more of a spice already in the collection, the old spice jar should be discarded and replaced. For example, after calling:

```cpp
insert(cs,oldest,newest,"cinnamon");
```

The spice collection output should now be:

**ALPHA ORDER:** basil cinnamon dill garlic nutmeg oregano sage  
**OLDEST FIRST:** garlic oregano nutmeg basil sage dill cinnamon  
**NEWEST FIRST:** cinnamon dill sage basil nutmeg oregano garlic

4.1 The typedef

First, help Ben by completing the definition of the typedef below:

```cpp
typedef chrono_set;
```
4.2 Printing out the spice collection

Next, write the code to output (to `std::cout`) Ben’s spices in alphabetical and chronological order:

```cpp
std::cout << "ALPHA ORDER: ";
std::cout << std::endl;
std::cout << "OLDEST FIRST: ";
```

---

4.3 Performance Analysis

Assuming Ben has \( n \) spices in his collection, what is the order notation for each operation? *Note: You may want to first complete the implementation of the `insert` operation on the next page.*

- printing in alphabetical order:

- printing in chronological order:

- `insert`-ing a spice to the collection:
Finally, implement the `insert` function for Ben’s spice collection. Make sure to handle all corner cases.

*sample solution: 26 line(s) of code*
5 Factor Tree

Write a recursive function named *factor_tree* that takes in a single argument of integer type and constructs the tree of the factors (and factors of each factor) of the input number. The function returns a pointer to the root of this tree. The example below illustrates the tree returned from the call *factor_tree*(60).

```cpp
class Node {
public:
    int value;
    std::vector<Node*> factors;
};
```

![Factor Tree Diagram]

*sample solution: 10 line(s) of code*
In this problem you will write a recursive function named `driving` that outputs to `std::cout` all *closed loop* paths of driving instructions on a rectangular grid less than or equal to a specified maximum path length. The car begins at (0,0) pointing north and at each step can go *straight*, *left*, or *right*. A path is said to “close the loop” if it finishes where it started, pointing in the same direction. For example, here are three sample closed loop paths (also illustrated below):

<table>
<thead>
<tr>
<th>Closed Loop Path</th>
<th>Illustration</th>
</tr>
</thead>
<tbody>
<tr>
<td>straight left left straight left left</td>
<td><img src="image1.png" alt="Image 1" /></td>
</tr>
<tr>
<td>left right left left right left left</td>
<td><img src="image2.png" alt="Image 2" /></td>
</tr>
<tr>
<td>right left left straight straight left straight straight left left</td>
<td><img src="image3.png" alt="Image 3" /></td>
</tr>
</tbody>
</table>

We provide the `Car` class and several helper functions:

```cpp
class Car {
public:
    Car(int x_, int y_, std::string dir_) : x(x_), y(y_), dir(dir_) {}
    int x;
    int y;
    std::string dir;
};

bool operator==(const Car &a, const Car &b) {
    return (a.x == b.x && a.y == b.y && a.dir == b.dir);
}

Car go_straight(const Car &c) {
    if (c.dir == "north") { return Car(c.x, c.y+1, c.dir); }
    else if (c.dir == "east") { return Car(c.x+1, c.y, c.dir); }
    else if (c.dir == "south") { return Car(c.x, c.y-1, c.dir); }
    else { return Car(c.x-1, c.y, c.dir); }
}

Car turn_left(const Car &c) {
    if (c.dir == "north") { return Car(c.x-1, c.y+1, "west"); }
    else if (c.dir == "east") { return Car(c.x+1, c.y+1, "north"); }
    else if (c.dir == "south") { return Car(c.x+1, c.y-1, "east"); }
    else { return Car(c.x-1, c.y-1, "south"); }
}

Car turn_right(const Car &c) {
    if (c.dir == "north") { return Car(c.x+1, c.y+1, "east"); }
    else if (c.dir == "east") { return Car(c.x+1, c.y-1, "south"); }
    else if (c.dir == "south") { return Car(c.x-1, c.y-1, "west"); }
    else { return Car(c.x-1, c.y+1, "north"); }
}
```

Your function should take in 3 arguments: the path constructed so far, the current car position & direction, and the maximum number of steps/instructions allowed. For example:

```cpp
std::vector<std::string> path;
Car car(0,0,"north");
int max_steps = 10;
driving (path, car, max_steps);
```
Now implement the recursive \texttt{driving} function.
In this problem we will use STL map and STL set to store and access a collection of integers and their factors. Below are the commands we use to initialize the two data structures diagrammed on the right.

```cpp
factor_type factors;
add_factors(factors,6);
add_factors(factors,14);
add_factors(factors,5);
add_factors(factors,8);
add_factors(factors,10);
add_factors(factors,9);
add_factors(factors,13);
add_factors(factors,15);
add_factors(factors,12);
add_factors(factors,21);
factor_type is_factor_of = reverse(factors);
```

### 7.1 The typedef

First, complete the definition of the typedef below:

```cpp
typedef factor_type;
```

### 7.2 Implementing add_factors

Now, implement the add_factors function. Note that this function only initializes the factors table.

```cpp
sample solution: 8 line(s) of code
```
If we are storing the factors of \( n \) different numbers in the *factors* structure, \( f \) different factors will eventually be stored in the *is_factor_of* structure, each number has on average (or at most) \( j \) factors, and each factor is a factor of on average (or at most) \( k \) numbers, what is the order notation for the running time of your *add_factors* function to add the number \( x \) and the factors of \( x \)?

### 7.3 Implementing reverse [ / 10 ]

Next, implement the *reverse* function to build the *is_factor_of* table from the completed *factors* table.

Using the variables \( n \), \( f \), \( j \), and \( k \) as defined above, what is the order notation for the running time of your *reverse* function?
7.4 Implementing remove

Finally, we would like to remove data from the tables. The remove function will remove a given number’s row from the factors table and remove the number from each of its factors in the is_factor_of table. For example, the call below results in the tables to the right.

```
remove(12,factors,is_factor_of);
```

Your task is to efficiently implement the remove function. Using the variables defined above, you should assume that \( n \geq f \geq k \geq j \).

```
<table>
<thead>
<tr>
<th>factors</th>
<th>is_factor_of</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2 6 8 10 14</td>
</tr>
<tr>
<td>6 2 3</td>
<td>3 6 9 15 21</td>
</tr>
<tr>
<td>8 2 4</td>
<td>4 8</td>
</tr>
<tr>
<td>9 3</td>
<td>5 10 15</td>
</tr>
<tr>
<td>10 2 5</td>
<td>7 14 21</td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14 2 7</td>
<td></td>
</tr>
<tr>
<td>15 3 5</td>
<td></td>
</tr>
<tr>
<td>21 3 7</td>
<td></td>
</tr>
</tbody>
</table>
```

Sample solution: 14 line(s) of code

Using the variables \( n, f, j, \) and \( k \) as defined above, what is the order notation for the running time of your remove function?
8 Double Tries [ / 30 ]

Ben Bitdiddle thinks he’s come up with a fantastic enhancement for binary search trees of integers he calls the Double Trie. Each Node will have up to 3 children. The \texttt{a} branch will store all elements less than the current node. The \texttt{b} branch will store all elements greater than the current node, but less than or equal to twice the current node. And the \texttt{c} branch will store all elements greater than twice the current node.

Ben suggests we start the implementation by writing a recursive \texttt{insert} function that takes in a pointer to node (initially the root of the Double Trie), the value to insert, and a pointer to the parent node (initially NULL). The function returns true if the value was successfully inserted and false if the value is already in the structure.

8.1 Implementing \texttt{insert} [ / 10 ]

\begin{figure}
\centering
\includegraphics[width=\textwidth]{double_trie_diagram.png}
\caption{Double Trie Diagram}
\end{figure}
Ben’s project partner Alyssa P. Hacker isn’t thrilled with the design. (She’s not sure it will significantly reduce the tree height because this structure is difficult to keep balanced.) However, they have a deadline, so this is a make-it-work moment and she tackles the challenge of reverse iteration over this structure. Specifically if the root variable points to the top of the diagram on the previous page, she would like this fragment of code:

```cpp
Node *tmp = find_largest(root);
while (tmp != NULL) {
    std::cout << tmp->value << " ";
    tmp = find_previous(tmp);
}
std::cout << std::endl;
```

to print all of the data in the tree in reverse order:

```
450 210 200 150 92 70 64 59 52 45 32 31 30 24 19 16 13 12 10 5 4 1
```

### 8.2 Implementing find_largest

Next, Alyssa implements the `find_largest` function:

```cpp
sample solution: 8 line(s) of code
```

Given a reasonably balanced Double Trie with \( n \) elements, what is the order notation of the running time of the `find_largest` function? Write one or two sentences explaining your answer.
Finally, Alyssa implements the `find_previous` function:
The statements below are false. Make a small change to correct each statement, ensuring that it remains interesting and informative.

**Binary Search Tree Iterators** [ /2] The average number of child or parent links that must be traversed when moving from one node to the next node in an in-order traversal is $O(\log n)$, where $n$ is the number of elements in the tree.

**Incomplete type** [ /2] In HW8 Friendly Recursion, many students encountered the compiler message “error: invalid use of incomplete type 'class Message'”, which should be solved by implementing all custom class member functions in the class declaration .h file.

**Breadth-First Search** [ /3] Executing a breadth-first search for the shortest path from root to leaf on a binary search tree will often be faster and require less additional memory than a depth-first search on the same tree.

**Been Here Before?** [ /2] To optimize the HW6 Ricochet Robots solver, the search tree for a board with three robots can be pruned (and the forward search from that point terminated) if any one of the robots reaches a board location that it has previously occupied.

**Hash Function Performance** [ /2] A hash function should run in $O(1)$ time, to ensure that the hash table will achieve $O(\log n)$ query time, where $n$ is the number of elements in the hash table.

**Red-Black Property** [ /3] Maintaining the Red-Black property for a hash table ensures that the data remains balanced and elements can be accessed in $O(\log n)$ time.
In this problem you will work with a simplified version of the Twitter or Google+ friendship/follower directed graph from HW8. Your task is to write a `merge_account` function that takes in a `Graph` object and the STL `string` names of two accounts and then modifies the graph to merge the two corresponding `Person` objects into a single object. Other people in the graph who were connected to either of the original accounts by a connection of either direction will be updated to link to the merged account. For example, let’s start with the graph connectivity on the left below. We want to merge the accounts for “Jennifer” and “Jenny”, preserving the name “Jennifer” on the merged account. We execute the following statement, resulting in the picture on the right.

```
merge_accounts(graph, "Jennifer", "Jenny");
```

**10.1 Corner Cases for `merge_accounts`**

Think carefully about a typical use case for merging accounts, and also about corner cases for the `merge_accounts` function. What different test cases will you need to write to ensure that your implementation is fully debugged and will work when attempting to join two arbitrary `Person` objects in a large graph? Write three or four concise and well written sentences describing sample input and the expected output.
10.2 Implementation of `merge_accounts`

Now, implement the `merge_accounts` function. Make sure that your function does not lead to memory errors or memory leaks.

`sample solution: 33 line(s) of code`