1 Upside-Down Binary Search Tree [ / 36 ]

1.1 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

Solution:

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);
```

1.2 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 BelongsInSubtree 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.

1.3 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:

```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: $O(n \log(n))$.

2 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’ = rental and ‘h’ = print costume history commands:

```text
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.
2.1 Data Structure Sketch

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.

Solution:

2.2 The typedefs

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;

2.3 Implementation of the Rental Command

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...
    }
    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;
        }
    }
}
```
2.5 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: \( O(c) \).

3 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: (apple,boat) (apple,cat) (apple,egg) (boat,cat) (boat,dog) (dog,egg) (dog,fig) (egg,fig)

```cpp
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;
}
4 Bitdiddle Post-Breadth Tree Traversal [ / 31 ]

4.1 Balanced Tree Example [ / 3 ]

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:

Solution:

4.2 Un-Balanced Tree Example [ / 3 ]

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:

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.
4.3 CollectLeaves Implementation

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);
    }
}
```

4.4 PrintPostBreadth Implementation

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","GAG"); Add(totals,kmers,"dog","TAG");
Add(totals,kmers,"dog","TAG"); Add(totals,kmers,"fruit fly","ACT");
Add(totals,kmers,"fruit fly","CAT"); 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_:

\[
\begin{align*}
\text{ACT:} & \quad \frac{2}{5} - \frac{3}{9} = 0.067 \\
\text{CAT:} & \quad = 0.000 \\
\text{GAG:} & \quad \frac{1}{5} - \frac{2}{9} = 0.022 \\
\text{TAG:} & \quad \frac{2}{5} - \frac{4}{9} = 0.044 \\
\text{overall:} & \quad = 0.133
\end{align*}
\]

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
### 5.1 The typedefs

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

**Solution:**

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

### 5.2 Add Implementation

Next, finish the implementation of the `Add` function.

**Solution:**

```cpp
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: $O(\log s + \log k)$.

### 5.3 Query Implementation

**Solution:**

```cpp
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;
}
```

### 5.4 MostCommon Implementation

**Solution:**

```cpp
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;
}
```
5.5 Difference Implementation

Solution:

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

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

6 Prescribed Pre-Ordering

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);
    }
};
```
6.1 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.

Solution: (many correct answers!)

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

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

6.2 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;
}
```
7 Un-Occupied Erase  

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.

7.1 Diagramming the Expected Output of `erase`  

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:**

```
10
  
7 5
  
15

\[\text{count(root)} \Rightarrow <3,1>\]
```

```
15
  
7

\[\text{count(root)} \Rightarrow <5,1>\]
```

```
2
  
5

\[\text{count(root)} \Rightarrow <3,2>\]
```

```
15
  
7

\[\text{count(root)} \Rightarrow <3,1>\]
```

```
15
  
20

\[\text{count(root)} \Rightarrow <3,1>\]
```

```
15
  
17

\[\text{count(root)} \Rightarrow <3,1>\]
```

7.2 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.

**Solution:**

```
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);
}

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);
}
```

### 7.3 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) { // recurse left
            return erase(p->left, v);
        } else { // recurse right
            return erase(p->right, v);
        }
    } 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;
    }
}
```

### 7.4 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;
    }
}
```

8 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 `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`:

```cpp
ERROR! conflicts with prior event: DS Exam
ERROR! room DCC 307 does not exist
ERROR! invalid time range: 22-25
ERROR! invalid day: Saturday
```

8.1 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;
```

8.2 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 every thing, but be neat and draw enough detail to demonstrate that you understand how each component of the data structure is organized and fits together.
8.3 Implementing make_reservation [ / 16 ]

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:

```c++
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++) {
```
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;

8.4 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 \( O(b \times c \times s) \).

9 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 & red 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;
}