CSCI-1200 Data Structures
Test 3 — Practice Problem Solutions

1 Bitdiddle Post-Breadth Tree Traversal [/ 31]

1.1 Balanced Tree Example [/ 3]

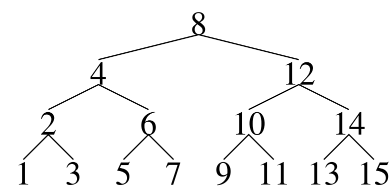
Ben Bitdiddle really wants to get his name on a traversal ordering. Even without a real world application for its use, he has invented what he calls the *post-breadth ordering.* His primary demonstration example is an exactly balanced, binary search tree with the numbers 1-15.

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:

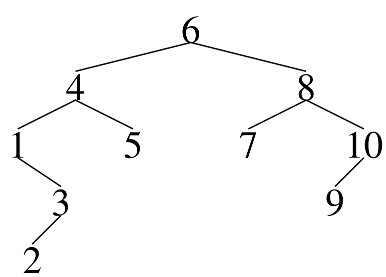
```
LEVEL 0:  1 3 5 7 9 11 13 15
LEVEL 1:  2 6 10 14
LEVEL 2:  4 12
LEVEL 3:  8
```

Solution:



1.2 Un-Balanced Tree Example [/ 3]

Solution:



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:

```
LEVEL 0:  2 5 7 9
LEVEL 1:  3 4 8 10
LEVEL 2:  1 6
```

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.

1.3 `CollectLeaves` Implementation [/ 11]

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.

```
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;
};
```

Solution:

```
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);
  }
}
```

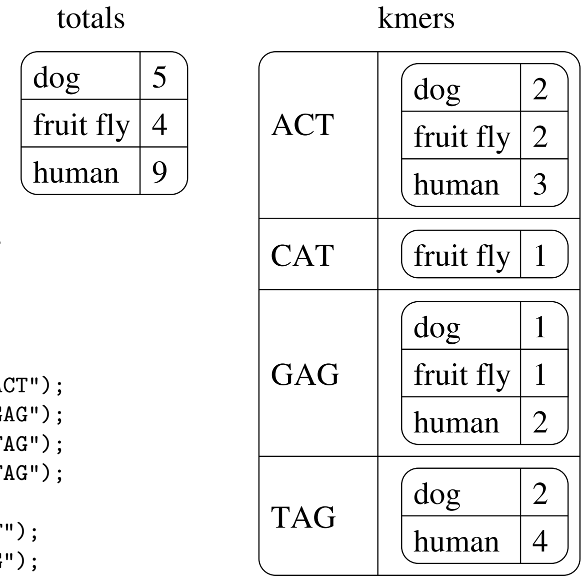
1.4 `PrintPostBreadth` Implementation [/ 14]

Now finish the implementation of the `PrintPostBreadth` function:

Solution:

```
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;
  }
}
```

2 Genome Difference Maps [/ 36]



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:

```
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","GAG");     Add(totals,kmers,"human","TAG");
Add(totals,kmers,"human","TAG");     Add(totals,kmers,"human","TAG");
Add(totals,kmers,"human","TAG");
Add(totals,kmers,"dog","ACT");       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","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:

```
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`:

ACT:	abs(2/5 - 3/9)	=	0.067
CAT:		=	0.000
GAG:	abs(1/5 - 2/9)	=	0.022
TAG:	abs(2/5 - 4/9)	=	0.044
overall:		=	0.133

Here is code to call the `Difference` helper function:

```
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
```

2.1 The `typedefs` [/ 4]

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

Solution:

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

2.2 `Add` Implementation [/ 7]

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

Solution:

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

2.3 `Query` Implementation [/ 6]

Solution:

```
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;
}
```

2.4 `MostCommon` Implementation [/ 7]

Solution:

```
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;
}
```

2.5 Difference Implementation [/ 12]

Solution:

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

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

3 Prescribed Pre-Ordering [/ 21]

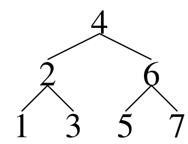
In this problem we will create an algorithm to construct a binary search tree from the desired pre-order traversal order. The driver function (below) takes in this sequence as a STL `vector`. If the contents of the vector is not a valid pre-order traversal order of a binary search tree, the function should return NULL.

```
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);
}
```

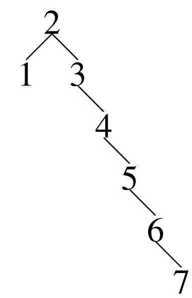
3.1 Test Cases [/ 7]

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.

valid: 4 2 1 3 6 5 7



valid: 2 1 3 4 5 6 7



Solution: (many correct answers!)

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

3.2 Finish the `MakePreOrderTree` Implementation [/ 14]

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:

```
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;
}
```

4 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.

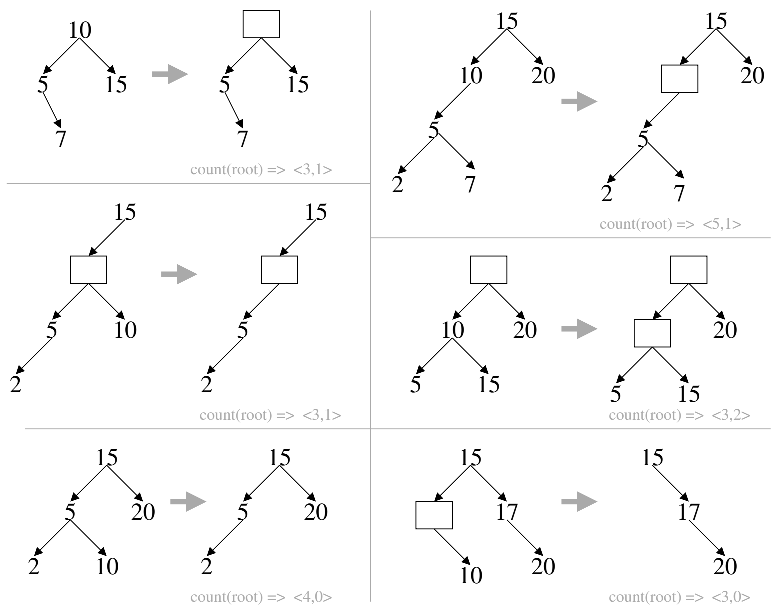
```
template <class T>
class Node {
public:
  Node(const T& v) :
    occupied(true), value(v),
    left(NULL), right(NULL) {}
  bool occupied;
  T value;
  Node* left;
  Node* right;
};
```

4.1 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.

Solution:



4.2 Counting Occupied & Unoccupied Nodes [/ 8]

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:

```
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);
}
```

4.3 Implement `erase` for Trees with Unoccupied Nodes [/ 13]

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:

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

4.4 Implement `insert` for Trees with Unoccupied Nodes [/ 12]

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:

```
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;
  }
}
```

5 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 `map`s would be a great fit for this application. Here's a portion of the main function with an example of how his program works:

```
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:

```
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:

```
success = make_reservation(rr, "DCC", 308, "Monday",   19, 3, "American Sniper") ||
          make_reservation(rr, "DCC", 307, "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
```

5.1 The `typedefs` [/ 5]

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

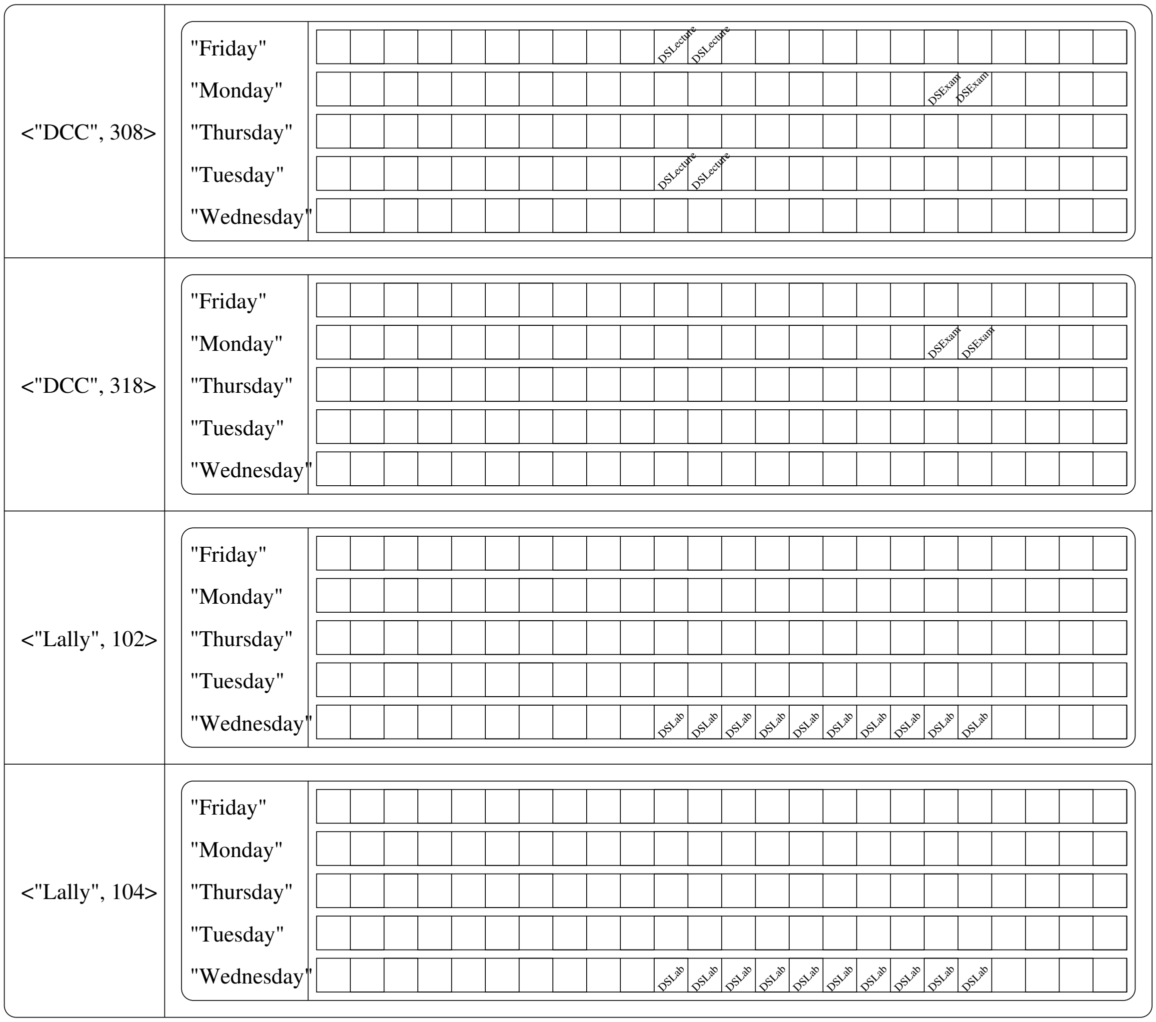
Solution:

```
typedef std::map < std::string, std::vector<std::string> > week_schedule;
typedef std::map < std::pair<std::string,int>, week_schedule > room_reservations;
```

5.2 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 *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.

Solution:



5.3 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:

```
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;
}
```

5.4 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 $b * c$ entries. To locate the specific room is $O(\log\ (b * c))$. 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 $O(\log\ (b * c) + \log\ d + s)$. We will also accept $O(\log(b * c) + s)$.

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 $b * c$ 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*c* (32 + 4 + d * (32 + t * 32)) = 36*b*c (memory to store each building & room pair) + 32*d*b*c (memory to store the days of the week strings) + 32*d*t*b*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 $b * c$ 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 $O(b * c * (\log\ d + s))$. We will also accept $O(b * c * s)$.

6 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 `string`s representing different types of clothing. `colors` is an STL `list` of STL `set`s of STL `string`s representing the different colors of each item of clothing. Your function should return an STL `vector` of STL `string`s describing each unique outfit (in any order) that can be created from these items of clothing.

Here is a small example:

```
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
```

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;
}
```

7 Spicy Chronological Sets using Maps [/ 33]

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:

```
  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");
```

chrono_set cs:

"basil"	<"cinnamon", "sage">
"cinnamon"	<"nutmeg", "basil">
"dill"	<"sage", "">
"garlic"	<"","oregano">
"nutmeg"	<"oregano","cinnamon">
"oregano"	<"garlic", "nutmeg">
"sage"	<"basil", "dill">

std::string oldest: "garlic"

std::string 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:

```
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
```

7.1 The `typedef` [/ 3]

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

Solution:

```
typedef std::map<std::string,std::pair<std::string,std::string> > chrono_set;
```

7.2 Printing out the spice collection [/ 8]

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

Solution:

```
  std::cout << "ALPHA ORDER:  ";
  chrono_set::const_iterator itr;
  for (itr = cs.begin(); itr != cs.end(); itr++) {
    std::cout << " " << itr->first;
  }
  std::cout << std::endl;
  std::cout << "OLDEST FIRST: ";
  std::string current = oldest;
  while (current != "") {
    std::cout << " " << current;
    current = cs.find(current)->second.second;
  }
  std::cout << std::endl;
```

7.3 Performance Analysis [/ 5]

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.*

Solution:
printing in alphabetical order:
Iterating through a map is linear in the number of elements in the map, $O(n)$.

printing in chronological order:
Finding each "next" element requires a *log* n find operation, overall = $O(n \textbf{ log } n)$.

`insert`**-ing a spice to the collection:**
Each map operation is *log* n, discard constant multiplier = $O(\textbf{log } n)$.

7.4 Implementing `insert` for the `chrono_set` [/ 17]

Finally, implement the `insert` function for Ben's spice collection. Make sure to handle all corner cases.

Solution:

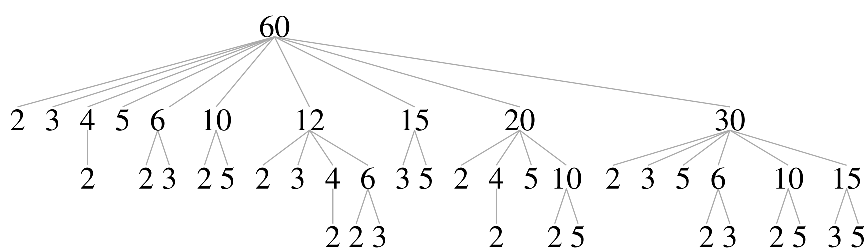
```
void insert(chrono_set &cs, std::string &oldest, std::string &newest, const std::string &spice) {
  // assume the spice isn't here, and try to add it to the end of the chronological order
  std::pair<chrono_set::iterator,bool> tmp =
    cs.insert(std::make_pair(spice,std::make_pair(newest,std::string(""))));
  // if the insert failed (spice was already there)
  if (tmp.second == false) {
    // need to edit the spices before & after the old copy of the spice
    std::string prev = tmp.first->second.first;
```

```
    std::string next = tmp.first->second.second;
    if (prev != "") {
      cs[prev].second = next;
    } else {
      // if the spice was the oldest
      oldest = next;
    }
    if (next != "") {
      cs[next].first = prev;
    } else {
      // if the spice was the newest
      newest = prev;
    }
    // reset the fields of this spice correctly
    tmp.first->second.second = "";
    tmp.first->second.first = newest;
  }
  if (cs.size() == 1) {
    // the very first spice
    oldest = newest = spice;
  } else {
    // point the previous newest spice at this spice
    cs[newest].second = spice;
  }
  newest = spice;
}
```

8 Factor Tree [/ 13]

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)`.

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



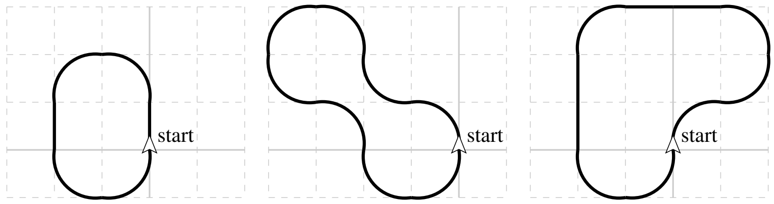
Solution:

```
Node* factor_tree(int num) {
  Node* answer = new Node;
  answer->value = num;
  for (int i = 2; i <= num/2; i++) {
    if (num % i == 0) {
      answer->factors.push_back(factor_tree(i));
    }
  }
  return answer;
}
```

9 Driving in Circles [/ 18]

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 is finishes where it started, pointing in the same direction. For example, here are three sample closed loop paths (also illustrated below):

```
  closed loop:  straight left left straight left left
  closed loop:  left right left left left right left left
  closed loop:  right left left straight straight left straight straight left left
```



We provide the Car class and several helper functions:

```
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:

```
std::vector<std::string> path;
Car car(0,0,"north");
int max_steps = 10;
driving (path,car,max_steps);
```

Now implement the recursive driving function.

Solution:

```
void driving(std::vector<std::string> &path, const Car &car, int max_steps,
     std::vector<Car> previous = std::vector<Car>()) {
  // base case, solution!
  if (path.size() > 0 && car == Car(0,0,"north")) {
    std::cout << "closed loop: ";
    for (int i = 0; i < path.size(); i++) {
      std::cout << " " << path[i];
    }
    std::cout << std::endl;
    return;
  }
  // base case, maximum recursion depth
  if (path.size() == max_steps) { return; }
  // make sure we aren't overlapping previous car positions
  // note: we are allowing the path to cross though!
```

```
  for (int i = 0; i < previous.size(); i++) {
    if (car == previous[i]) return;
  }
  previous.push_back(car);
  // try to go straight
  path.push_back("straight");
  driving(path,go_straight(car),max_steps,previous);
  path.pop_back();
  // try to go left
  path.push_back("left");
  driving(path,turn_left(car),max_steps,previous);
  path.pop_back();
  // try to go right
  path.push_back("right");
  driving(path,turn_right(car),max_steps,previous);
  path.pop_back();
  previous.pop_back();
}
```