CSCI-1200 Data Structures — Spring 2023
Test 3 — Thursday, April 6th 6-7:50pm

Kirsten Szeto	szetok2@rpi.edu lab section: 12
room: Darrin 318 zone: MAGENTA row: 3 seat: 11	6-7:50pm

Write the names of the undergraduate mentors for your lab sorted by height, from shortest to tallest. If two mentors are very close in height, you can put them in either order.

DAVID, SAMIR, XENIA

- This exam has 4 problems worth a total of 100 points (including the cover sheet).
- This packet contains 10 pages of problems numbered 1-10. Please count the pages of your exam and raise your hand if you are missing a page.
- The packet contains 1 blank pages. If you use a blank page to solve a problem, make a note in the original box and clearly label which problem you are solving on the blank page.
- This test is closed-book and closed-notes except for the .pdf notes you (optionally) uploaded to Submitty by last night. These notes are the last 2 pages of your exam packet.
- **DO NOT REMOVE THE STAPLE OR SEPARATE THE PAGES OF YOUR EXAM. DOING SO WILL RESULT IN A -10 POINT PENALTY!**
- You may have pencils, eraser, pen, tissues, water, and your RPI ID card on your desk. Place everything else on the floor under your chair. Electronic equipment, including computers, cell phones, calculators, music players, smart watches, cameras, etc. is not permitted and must be turned "off" (not just vibrate).
- Raise your hand if you need to ask a proctor something that is not related to one of the questions on the test. We will **not** be able to answer if it's about one of the test problems.
- Please state clearly any assumptions that you made in interpreting a question. Unless otherwise stated you may use any technique that we have discussed in lecture, lab, or on the homework.
- Please write neatly. If we can't read your solution, we can't give you full credit for your work.
- You do not need to write `#include` statements for STL libraries. Writing `std::` is optional. Do not use `auto`, do not use `goto`.

Kirsten Szeto szetok2@rpi.edu

1 Short Operators Questions [/ 20]

This problem consists of four short questions regarding C++ operators.

1.1 operator++ [/ 5]

For a class Baz, consider the two prototypes `Baz& Baz::operator++()` and `Baz Baz::operator++(int)`. How do we call them, and what's the "intuitive meaning" of these operators?

operator ++ () → returns a reference to the object after incrementing so that we're actually changing the value of the object

operator ++(int) & → returns a copy of the object with the value inside incremented by a specific int.

1.2 operator<< [/ 5]

If we have class `Foo`, and we want to do:

```
Foo bar;
cout << bar << endl;
```

is `operator<<` a member of `Foo`, friend of `Foo`, or `non-member` function? Why? What's the prototype for the `operator<<` function in this case?

The operator<< is a non-member function because the operator << has a default version in the stl class. It should normally defined as a non-member function.

1.3 string Race [/ 5]

Consider two STL strings, a, and b, where a.length() is much larger than b.length(). Even assuming that a has enough internal room pre-allocated for 8*(a.length() + b.length()), the first statement below is much faster than the second. Write the prototypes for all operators called, assuming that they are all members of the string class. Why is there such a speed difference?

```
a += b;      //faster
a = a + b;   //slower
```

operator=(a, (operator+(a,b))); → statement 1

operator=(a, string(operator+(a,b))) → statement 2

Statement 1 is faster because it is appending the b string string to the a string as opposed to statement 2 which is creating a new string that is the combination of a and b and then reassigning it to the variable name a.

1.4 Copy and Assignment [/ 5]

Given a class called Widget that has a default constructor, write code to call the default constructor, copy constructor, and the assignment operator. If there's a way to call the the copy constructor using an =, write that as well. Put a comment next to each line of code explaining what it calls.

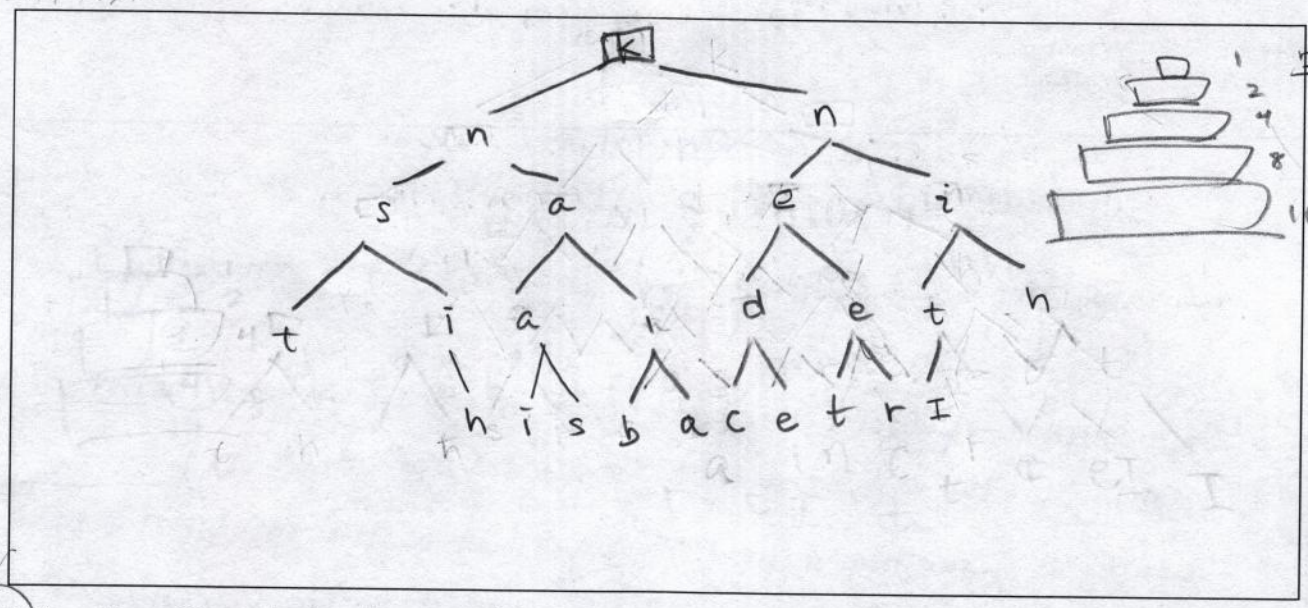
```
Widget a();
Widget b(a);
Widget c();

c = a;
Widget d = Widget();
```

Kirsten Szeto szetok2@rpi.edu

2 Short Answers - Trees [/22]

2.1 Post-Order Traversal [/5]

Draw a balanced tree of strings (words) with the following post-order traversal: ~~this is a balanced tree I think~~



2.2 Tree Complexity [/7]

For a binary tree with n nodes, what is the time complexity of starting from the root and finding the smallest value in the tree? Write the big O notations for both the balanced and the very unbalanced cases, assuming efficient algorithms:

balanced: $O(\log n)$

unbalanced: $O(n)$

For a binary *search* tree with n nodes, what is the time complexity of starting from the root and finding a leaf node in the tree? Write the big O notations for depth-first search and breadth-first search, both times considering a balanced tree and a very unbalanced tree, assuming efficient algorithms. (This gives you a total of four cases: balanced DFS, unbalanced DFS, balanced BFS, unbalanced BFS).

	balanced	unbalanced
DFS	$O(\log n)$	$O(n)$
BFS	$O(n)$	$O(n)$

2.3 Pre-Order Traversal [/5]

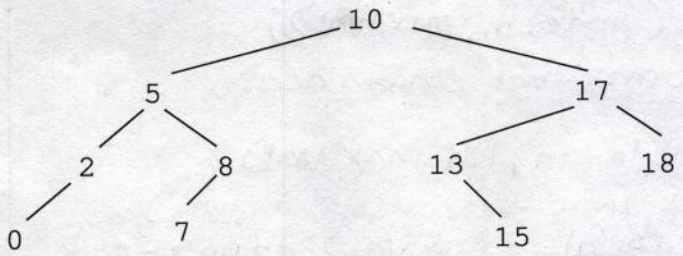
Write a small function called `PreOrderPrint` that takes a `TreeNode*` (recall these have a `left`, `right`, and `value` member) that prints the values out using an pre-order traversal.

```
void PreOrderPrint (TreeNode* p){
  if (p){
    cout << p->value << endl;
    PreOrderPrint (p->left);
    PreOrderPrint (p->right);
  }
}
```

sample solution: 7 line(s) of code

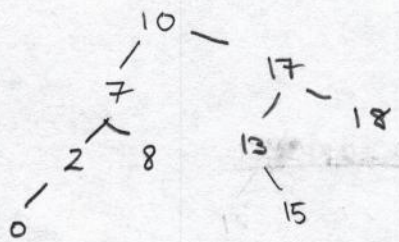
2.4 Erasing From Trees [/5]

Consider the following binary search tree:

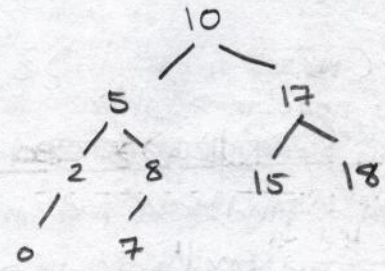


Draw the resulting trees from the following `erase` calls. Assume this is two independent calls to the tree shown above, and not `erase(5); erase(13);`. If you need to choose between the left and right subtree, choose the *right* subtree.

erase(5)

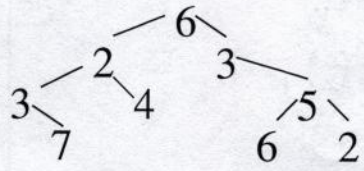


erase(13)



3 Tree Maximum Value Frequency [/30]

In this problem, you will write three versions of a function called `MaxFreq`. Each version takes in a pointer to the root of a binary tree that consists of positive integers represented by `Node` objects. You can assume the tree has at least one value in it. Your function should return a `std::pair` where the first part of the pair is the maximum (largest) element in the tree, and the second part of the pair is the number of times the maximum element appeared in the tree. For the example tree below, `MaxFreq(root)` will return `<7,1>` since 7 is the largest value in the tree and it only appears once. For runtimes we will use n to represent the number of nodes in the tree.




```
class Node{
public:
  int val;
  Node* left;
  Node* right;
};
```

3.1 `MaxFreq` With Helper Functions [/10]

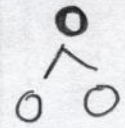
First, write a version of `MaxFreq` that uses two **recursive** helper functions, `FindMax` which takes a `Node*` pointing to the root of a tree and returns the largest value in the tree, and `CountFreq` which takes a `Node*` pointing to the root of a tree and `int`, and returns how many times the integer is found in the tree. You must implement these helper functions as well. The total runtime should be $O(n)$.

```
pair<int,int> MaxFreq(Node* node_in){
    int max_val = FindMax(node_in);
    int max_occ = CountFreq(node_in, max_val);
    return std::make_pair(max_val, max_occ);
}
int CountFreq(Node* node_in, int max_val){
  int times = 0;
  if(node_in != NULL && node_in->val == max_val){ times = 1; }
  return times + countFreq(node_in->left, max_val) + countFreq(node_in->right,
                                                               max_val);
}

int FindMax(Node* node_in){
  int curr = node_in->val;
  if(node_in != NULL){
    int right = FindMax(node_in->right);
    int left = FindMax(node_in->left);
    if(right > curr){ curr = right; }
    if(left > curr){ curr = left; }
  }
  return curr;
}
```

sample solution: 14 line(s) of code

3.2 Recursive `MaxFreq` With No Helpers [/10]

Next, write a **recursive** version of `MaxFreq` that takes only a `Node*` and returns a `pair`. Do not write any other functions. You can neither use `FindMax` nor `CountFreq`. The total runtime should be $O(n)$.



```
pair<int, int> MaxFreq ( Node* node_in ){
    int max_val = node_in -> val;
    int max_occ = 1;

    if(node_in -> right == NULL && node_in -> left == NULL){
        return std::make_pair ( max_val, max_occ);
    }
    else {
        if (node_in -> right != NULL){
            pair<int, int> right = MaxFreq ( node_in -> right);
            if (right.first > max_val ){
                max_val = right.first;
                max_occ = right.second;
            }
            else if ( right.first == max_val){
                max_occ += right.second;
            }
        }
        if (node_in -> left != NULL){
            pair <int, int> left = MaxFreq(node_in -> left);
            if (left.first > max_val){
                max_val = left.first;
                max_occ = left.second;
            }
            else if ( left.first == max_val){
                max_occ += left.second;
            }
        }
    return std::make_pair (max_val, max_occ);
}
```

sample solution: 18 line(s) of code

3.3 Iterative `MaxFreq` With No Helpers [/10]

Finally, write an **iterative** version of `MaxFreq` that takes only a `Node*` and returns a `pair`. Do not write any other functions. You can neither use `FindMax` nor `CountFreq`. The total runtime should be $O(n \log n)$.

```
pair<int, int> MaxFreq (Node* node_in) {

    go to each node;
        if ever at a node that has greater
          val than last recorded value
          reset occ and max_value
        if at node with equal value
          increment occ (occurence counter)

}
```

pseudo code.

sample solution: 18 line(s) of code

4 Sets and Strings [/25]

4.1 Counting Unique Characters [/10]

Write a function called `uniqueCharacters` that takes a `string` and returns `true` if every character in the string is unique, and `false` if there are any duplicate characters. The only STL containers you can use are `sets`, `strings`, and `pairs`. **Don't** use any arrays. If n is the length of the string, your function must have a time complexity no slower than $O(n \log n)$. You can assume $n \geq 1$. Some examples:

```
Input: "abcdefg" Output: true (because all characters are distinct)
Input: "abca" Output: false (because letter a appears twice)
Input: "abc1234" Output: true (because all characters are distinct)
```

```
bool uniqueCharacters(string input){
    std::set<char> letters;
    for(int i=0 ; i<input.length() ; i++){
        pair<iterator,bool> inserted;
        inserted = letters.insert(input[i]);
        if (inserted.second == false){
            return false;
        }
    }
}
```

sample solution: 11 line(s) of code

4.2 Duplicate Characters Within k Distance [/15]

Write a function `findNearbyDuplicates` that takes an input `string str`, a positive integer k, and returns a `string` consisting of all characters in `str` which satisfy the two conditions listed below. The only STL containers you can use are `sets`, `strings`, and `pairs`. **Don't** use any arrays.

1. the character appears more than once in `str`.
2. there is at least one pair of the duplicate character, which are positioned no more than `k` characters apart in `str`.

All characters in the `string` your function returns should be distinct. The order of these characters in your returned string do not matter. The time complexity of your solution must be faster than $O(n^2)$, where n is the length of the string. *Hint:* Keep a set of up to k characters at a time. Example input/return values:

```
Input: str="abcdefg!&*", k=3, Return: ""   //No duplicates found
Input: str="abca", k=3, Return: "a"   //'a' appears twice, the 'a's are within k of each other
Input: str="abca1234", k=2, Return: "" //'a' appears twice, but they are NOT within k of each other
Input: str="aaaaaaaaaabcabc123412", k=3, Return: "abc"   // "bca", "acb", etc. would also be correct
```

```
string findNearbyDuplicates(string str, int k){
    string toRet;
    set<char> chars_in;
    for(int i=0; i<str.length(); i++){
        pair<iterator, bool> inserted;
        inserted = chars_in.insert(str[i]);
        if(inserted.second == false){
            for(int j=0; j<=k; j++){
                int front = i+j;
                int back = i-j;
                if(back!<0 && str[back] == str[i]){
                    toRet += str.substring(i, 1);
                    break;
                }
                if(front !> str.length && str[front] == str[i]){
                    toRet += str.substring(i, 1);
                    break;
                }
            }
        }
    }
    return toRet;
}
```

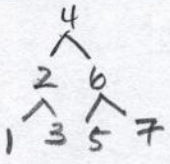
sample solution: 22 line(s) of code

(blank page)

3.2

```
pair <int, int> MaxFreq (Node* node_in) {
    int max_val = node_in -> val;
    int max_occ = 1;
    if (node_in -> right == NULL & node_in -> left == NULL){
        std::make_pair (max_val, max_occ);
    }
    else {
        pair <int, int> left = MaxFreq (node_in -> left);
        pair <int, int> right = MaxFreq (node_in -> right);
        int comp_max, comp_occ;
        if ( left.first == right.first){
                comp_max = left.first;
                comp_occ += ( left.second + right.second );
        }
        else if (left.first < right.first ){
                comp_max = right.first;
                comp_occ = right.second;
        }
        else {
                comp_max
```

Kirsten Szeto 662044026 Mentors: Nia, Samir, David, Xenia, Abby
Typedefs - typedef type t_name → EX: typedef map<string, vector<int> > map_vect;
Lists: sequentially linked structure
Push_front, pop_front, push_back, pop_back, list.sort(opt_func) (vect → std::sort(beg, end, opt))
Cannot add integers to list iterators ex: itr + 4 → can for vector + string itrs
Operators: as member functions → obj.operator=(s_obj)
→ Private variables can be accessed directly, s_obj can be accessed by rhs. || one parameter and const
Operators: as non-member functions → operator=(first, second)
→ Private variables cannot be accessed directly + two parameters
Friend Class: allows for access all the private member variables and functions of class as if public
Friend Function: allows for Friend function to access and work just like member function
Note: Friend class must be declared as public || We cannot create new operators or change # of args
Maps: #include <map> – search, insert, erase = O(log n) || keys are ordered (need op<) no duplicates
std::map<key_type, value_type> var_name - declaration || std::pair <**const** key_type, value_type> - entry
std::map<std::string, int>::const_iterator it - iterator
map.find(key) - find func → returns map itr to position of key else map.end() || .size()
map.insert(std::make_pair(val, val)); → inserts pair, returns pointer to insertion/place, bool of inserted
map.erase(iterator p) → deletes pair referred to by p || map.erase(itr first, itr, last) → deletes from first to last- not last || size_type erase(const key_type& k) → earases pair containing k return bool of deleted
[] operator - Ex: ++count[s] looks for key of s, if none create and increment, if there, just increment
Map Iterators: std::map<type, type>::iterator it, can be incremented with ++, - -. it can access first and second with it->first, it->second
Pairs: #include <utility>
std::pair<int, double> p1(8, 8.8) || std::make_pair(8, 9.0) - declarations || .first or .second - accessing
Sets: ordered containers storing unique "keys" || sorts with < to keys || Insert, erase, find = O(log n)
std::set<const val_type> set_1() - declaration NOTE: no []operator and have .size
pair<iterator, bool> set<key>::insert(const Key& entry) – returns pos of insertion or exist, bool if inserted
Iterator set<Key>::insert(iterator pos, const Key& entry) – return pos of insertion, pos is a "hint"
size_type set<Key>::erase(const Key& x); – returns 1 removed, 0 for not removed
void set<Key>::erase(itr p) – erases at iterator || void set<Key>::erase(iterator first, iterator last) – erase
Const_iterator set<key>::find(const Key& x) const; - find – returns pos or set.end() if not found
Binary Trees: empty or node that has pointers to two binary sub-trees;
top most node = root, left +right node = children + subtrees, node with null left + right = leafs
Binary Search Trees (BST): at each node of the tree, the value stored at the node is >= all the values stored in the left subtree and <= all the values stored in the right subtree
Balanced Tree: the number of nodes on each subtree is approximately the same
given : balanced tree from 1 - 7, 4 root ABCD - bring in print in order code



In-order Traversal - print from smallest to largest (print left, me, print right) - 1234567
Post-order Traversal - print children then parent (print left, print right, me) - 1325764
Pre-order Traversal - parents first then children - (print me, print left, print right) - 4213657
B+ Trees: trees that can have up to B children and B - 1 keys || Good for everything bc flat + wide
4 2 1 3 6 5 7
Depth-first - follow all the way to leaf then back up to decision point (in, pre, post order trav)
- Quick but takes a while if wrong decision is made early on

Breadth-first - nodes closer to roots visited first || solution closer to root found first || memory intensive

Kirsten Szeto szetok2@rpi.edu

Ds_set - root_, size_ → priv representation

```
void destroy_tree(TreeNode<T>* p) {
  if (!p) return;
  destroy_tree(p->right);
  destroy_tree(p->left);
  delete p;
}

iterator find(const T& key_value, TreeNode<T>* p) {
  if (!p) return end();
  if (p->value > key_value)
    return find(key_value, p->left);
  else if (p->value < key_value)
    return find(key_value, p->right);
  else
    return iterator(p,this);
}
```

```
template <class T> class TreeNode {
public:
  TreeNode() : left(NULL), right(NULL) {}
  TreeNode(const T& init) : value(init), left(NULL), right(NULL) {}
  T value;
  TreeNode* left;
  TreeNode* right;
};
```

```
std::pair<iterator,bool> insert(const T& key_value, TreeNode<T>*& p, TreeNode<T>* the_parent) {
  if (!p) {
    p = new TreeNode<T>(key_value);
    p->parent = the_parent;
    this->size_++;
    return std::pair<iterator,bool>(iterator(p,this), true);
  }
  else if (key_value < p->value)
    return insert(key_value, p->left, p);
  else if (key_value > p->value)
    return insert(key_value, p->right, p);
  else
    return std::pair<iterator,bool>(iterator(p,this), false);
}
```

```
int erase(T const& key_value, TreeNode<T>* &p) {
  if (!p) return 0;

  // look left & right
  if (p->value < key_value) {
    return erase(key_value, p->right);
  }
  else if (p->value > key_value) {
    return erase(key_value, p->left);
  }

  // Found the node.  Let's delete it
    assert (p->value == key_value);
    if (!p->left && !p->right) { // leaf
      delete p;
      p=NULL;
      this->size_--;
    } else if (!p->left) { // no left child
      TreeNode<T>* q = p;
      p=p->right;
      assert (p->parent == q);
      p->parent = q->parent;
      delete q;
      this->size_--;
    } else if (!p->right) { // no right child
      TreeNode<T>* q = p;
      p=p->left;
      assert (p->parent == q);
      p->parent = q->parent;
      delete q;
      this->size_--;
  } else { // Find rightmost node in left subtree
    TreeNode<T>* q = p->left;
    while (q->right) q = q->right;
    p->value = q->value;
    // recursively remove the value from the left subtree
    int check = erase(q->value, p->left);
    assert (check == 1);
  }
  return 1;
}
```

```
void print_in_order(std::ostream& ostr, const TreeNode<T>* p) const {
  if (p) {
    print_in_order(ostr, p->left);
    ostr << p->value << "\n";
    print_in_order(ostr, p->right);
  }
}
```

```
std::string s;
std::map<std::string, int> counters; // store each word and a

// read the input, keeping track of each word and how often w
while (std::cin >> s)
  ++counters[s];

// write the words and associated counts
std::map<std::string, int>::const_iterator it;
for (it = counters.begin(); it != counters.end(); ++it) {
  std::cout << it->first << "\t" << it->second << std::endl;
}
```

Height of tree

```
unsigned int find_height(Node* root)
{
  if (root == nullptr) {
    return 0;
  }
  else {
    unsigned int left_height =
find_height(root->left);
    unsigned int right_height =
find_height(root->right);
    return 1 + std::max(left_height,
right_height);
  }
```

Sum of odd map values

```
map<string, vector<int>> m;
unsigned int count = 0;
std::map<string,
vector<int>>::iterator it = m.begin();
while (it != m.end()) {
  for (unsigned int i=0;
i<it->second.size(); i++) {
    if (it->second[i]%2 == 1) {
      count += it->second[i];
    }
  }
  it++;
}
```

Smallest value in tree

```
template <class T>
TreeNode<T>*
FindSmallestInRange(const T& a,
const T& b, TreeNode<T>* root, T&
best_value){
 if(!root){ return NULL; }
  TreeNode<T>* left_subtree =
FindSmallestInRange(a,b,root->left,b
est_value);
  TreeNode<T>* right_subtree =
FindSmallestInRange(a,b,root->right,
best_value);
 if(root->value >= a && root->value <
best_value){
   best_value = root->value;
   return root;
 }
 else if(left_subtree &&
left_subtree->value == best_value){
   return left_subtree;
 }
 else if (right_subtree){
   return right_subtree;
 }
 return NULL;
}
```