template <class T>

**class TreeNode** {

public:

TreeNode() : left(NULL), right(NULL), parent(NULL) {}

TreeNode(const T& init) : value(init), left(NULL), right(NULL), parent(NULL) {}

T value;

TreeNode\* left;

TreeNode\* right;

TreeNode\* parent; // to allow implementation of

};

**template <class T> class ds\_set**;

template <class T>

class tree\_iterator {

public:

tree\_iterator() : ptr\_(NULL), set\_(NULL) {}

tree\_iterator(TreeNode<T>\* p, const ds\_set<T> \* s) : ptr\_(p), set\_(s) {}

tree\_iterator(const tree\_iterator& old) : ptr\_(old.ptr\_), set\_(old.set\_) {}

~tree\_iterator() {}

tree\_iterator& operator=(const tree\_iterator& old) { ptr\_ = old.ptr\_; set\_ = old.set\_; return \*this; }

const T& operator\*() const { return ptr\_->value; }

friend bool operator== (const tree\_iterator& lft, const tree\_iterator& rgt)

{ return (lft.set\_ == rgt.set\_ && lft.ptr\_ == rgt.ptr\_); }

friend bool operator!= (const tree\_iterator& lft, const tree\_iterator& rgt)

{ return (lft.set\_ != rgt.set\_ || lft.ptr\_ != rgt.ptr\_); }

tree\_iterator<T> & operator++() {

if (ptr\_->right != NULL) { // find the leftmost child of the right node

ptr\_ = ptr\_->right;

while (ptr\_->left != NULL) { ptr\_ = ptr\_->left; }

} else {

while (ptr\_->parent != NULL && ptr\_->parent->right == ptr\_) { ptr\_ = ptr\_->parent; }

ptr\_ = ptr\_->parent; }

return \*this; }

tree\_iterator<T> operator++(int) {

tree\_iterator<T> temp(\*this);

++(\*this);

return temp; }

tree\_iterator<T> & operator--() {

qif (ptr\_ == NULL){

assert( set\_ != NULL);

ptr\_ = set\_ -> root\_;

while ( ptr\_ -> right != NULL) { ptr\_ = ptr\_ -> right; }}

else if ( ptr\_ -> left != NULL ){

ptr\_ = ptr\_ -> left;

while ( ptr\_ -> right != NULL) { ptr\_ = ptr\_ -> right; }

}else {

while ( ptr\_ -> parent != NULL && ptr\_ -> parent -> left == ptr\_ ){

ptr\_ = ptr\_ -> parent;xs }

ptr\_ = ptr\_ -> parent; }

return \*this;}

tree\_iterator<T> operator--(int) {

tree\_iterator<T> temp(\*this);

--(\*this);

return temp; }

private:

TreeNode<T>\* ptr\_;

const ds\_set<T>\* set\_;};

**template <class T>**

**class ds\_set {**

public:

ds\_set() : root\_(NULL), size\_(0) {}

ds\_set(const ds\_set<T>& old) : size\_(old.size\_) {

root\_ = this->copy\_tree(old.root\_,NULL); }

~ds\_set() {

this->destroy\_tree(root\_);

root\_ = NULL;}

ds\_set& operator=(const ds\_set<T>& old) {

if (&old != this) {

this->destroy\_tree(root\_);

root\_ = this->copy\_tree(old.root\_,NULL);

size\_ = old.size\_;}

return \*this; }

typedef tree\_iterator<T> iterator;

friend class tree\_iterator<T>;

int size() const { return size\_; }

bool operator==(const ds\_set<T>& old) const { return (old.root\_ == this->root\_); }

void accumulate ( T & string ) const {

ds\_set<T>::iterator itr;

for ( itr = this -> begin(); itr != this -> end(); itr ++ ){

string += \*itr; }

}

iterator find(const T& key\_value) { return find(key\_value, root\_); }

std::pair< iterator, bool > insert(T const& key\_value) { return insert(key\_value, root\_, NULL); }

int erase(T const& key\_value) { return erase(key\_value, root\_); }

// OUTPUT & PRINTING

friend std::ostream& operator<< (std::ostream& ostr, const ds\_set<T>& s) {

s.print\_in\_order(ostr, s.root\_);

return ostr;}

void print\_as\_sideways\_tree(std::ostream& ostr) const {

print\_as\_sideways\_tree(ostr, root\_, 0);}

iterator begin() const {

if (!root\_) return iterator(NULL,this);

TreeNode<T>\* p = root\_;

while (p->left) p = p->left;

return iterator(p,this);}

iterator end() const { return iterator(NULL,this);}

bool sanity\_check() const {

if (root\_ == NULL) return true;

if (root\_->parent != NULL) {

return false;}

return sanity\_check(root\_);}

private:

TreeNode<T>\* root\_;

int size\_;

// PRIVATE HELPER FUNCTIONS

TreeNode<T>\*  **copy\_tree**(TreeNode<T>\* old\_root, TreeNode<T>\* the\_parent) {

if (old\_root == NULL)

return NULL;

TreeNode<T> \*answer = new TreeNode<T>();

answer->value = old\_root->value;

answer->left = copy\_tree(old\_root->left,answer);

answer->right = copy\_tree(old\_root->right,answer);

answer->parent = the\_parent;

return answer; }

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);}

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);

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;

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); } }

void print\_as\_sideways\_tree(std::ostream& ostr, const TreeNode<T>\* p, int depth) const {

if (p) {

print\_as\_sideways\_tree(ostr, p->right, depth+1);

for (int i=0; i<depth; ++i) ostr << " ";

ostr << p->value << "\n";

print\_as\_sideways\_tree(ostr, p->left, depth+1);}}

bool sanity\_check(TreeNode<T>\* p) const {

if (p == NULL) return true;

if (p->left != NULL && p->left->parent != p) {

return false;}

if (p->right != NULL && p->right->parent != p) {

return false; }

return sanity\_check(p->left) && sanity\_check(p->right);

} };

#endif

16.1Which of the following statements is least true? **If you can code something with fewer keystrokes it is always better software.**

16.2 Which of the following is false for the STL pair class? **The first item in an STL pair is always const and cannot be changed**

16.3 Which of the following is true for the STL map iterators? **None of the above (hint, data is accessed in the….)**

17.1 Which of the following statements is false? **Any valid C++ type can be used as the key(first) part of an STL map.**

17.2 what is the order natation to look up student X’s grade in course?

**O(logs + k)**

17.3 what is the order notation to make a list of all students who have taken course Y? **O(s + k)**

17.4 Which of the following statement is false about a binary search tree holding the integer 1-10? **If 7 is the parent of 5, then 5 is the right child of 7**

18.1 How many exactly balanced binary search trees exist with the numbers 4.5 9.8 3.5 … **1, 7!**

18.2 Which of the following statements about STL container types is true? **A program that uses an STL set can easily be changed to use an STL map instead, with no performance impact.**

18.3 order notation of find\_smailest function we just wrote,assuming the tree has n nodes in it and a height h. **O(h)**

18.4 assuming the tree has n nodes in it and a height h?

**O(n)**

19.1 What is the post-order traversal of this tree? **1 3 2 5 7 6 4**

19.2 what is the height of the binary search tree that has pre-order traversal 1 2 3 4 5 6 7? **7**

19.3 what is the traversal order of the destroy\_tree function we wrote earlier?

**Poster order**

19.4 What is the sum of the last 4 elements in a breadth first traversal of an exactly balanced binary search tree with the elements 1-7? **16**

19.5 **running time: best: O(n) Average: O(n) Worst:O(n) Memory usuage: Best O(1) Average: O(n) Worst: O(n)**

19.6 Which of the following statements about tree iterator is false? **If the tree iterator is pointing at the node containing the last element in sorted order, that node must be a leaf node.**

20.2 with the tree with n nodes, fot the recursive tree height algorithm, ….? **Running time:O(n) O(n) O(n) memory:O(log n) O(log n) O(n)**

20.3 for the tree with n nodes, for a breadth-first shortest path to leaf node algorithm…? **RT: O(1) O(n) O(n) MU: O(1) O(n) O(n)**

20.4 For either version of the tree iterator operator++ function, for a balanced tree with n elements what is the order notation for the worst case call to operator++….? **S\_wor: O(logn) S\_avg: O(1) total: O(n)**

21.1: **2,4&6**

21.2 Which of the following statement about operator overloading is true? **If a member function takes in as an argument a second object of the class type, it has access to the private member variable of both the “this” object and the argument object.**

21.3 is false? **You can overload operators for every symbol on your** keyboardMAP:

int main(int argc, const char \* argv[]) {

std::ifstream istr(argv[1]);\

if (!istr) {

std::cerr << "Could not open " << argv[1] << std::endl;

return 1;}

std::map<int, int> numbers;

int x;

while ( istr >> x ){

std::map<int, int>::iterator itr = numbers.find(x);

if ( itr == numbers.end()){

numbers.insert(std::make\_pair(x,1));}

else{

itr->second ++;}}

int mode = numbers.begin()->first;

for ( std::map<int, int>::const\_iterator itr = numbers.begin(); itr != numbers.end(); ++ itr ){

if ( itr-> second > numbers[mode] ){

mode = itr -> first;}}

RECURSION:

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

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();

……….

------------------------------------------------------------

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;}}

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;}

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 ofevents, 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).

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)

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;}