CSCI-1200 Data Structures
Test 3 — Practice Problem Solutions

1 Short Recursion Problems [/ 18]

1.1 Donut Boxes [/ 10]

In this problem you are given a vector of strings representing donut flavors. You can assume they are unique and that the vector is not empty. Fill in the call in the driver function, and then implement your function, which should be recursive and modifies a vector argument to contain all possible ways to arrange a box with one of each flavor of donut.

```
typedef std::vector<std::string> DonutBox;
DonutBox donuts;
std::vector<DonutBox> boxes;
donuts.push_back("strawberry"); donuts.push_back("chocolate"); donuts.push_back("maple");
findBoxes(donuts, boxes);
printDonutBoxes(boxes);
```

Gives the output:

```
Printing 6 boxes:
Box has: strawberry chocolate maple
Box has: strawberry maple chocolate
Box has: chocolate strawberry maple
Box has: chocolate maple strawberry
Box has: maple strawberry chocolate
Box has: maple chocolate strawberry
```

Solution:

```
void findBoxes(const DonutBox& box, DonutBox& current_box, std::vector<DonutBox>& boxes){
    if(box.empty()){
        boxes.push_back(current_box);
        return;
    }
    for(unsigned int i=0; i<box.size(); i++){
        DonutBox tmp_box = box;
        current_box.push_back(box[i]);
        tmp_box.erase(tmp_box.begin()+i);
        findBoxes(tmp_box, current_box, boxes);
        current_box.pop_back();
    }
}

void findBoxes(const DonutBox& box, std::vector<DonutBox>& boxes){
    DonutBox tmp;
    findBoxes(box, tmp, boxes);
}
```

1.2 String Reversal [/ 8]

In this problem you will write *reverse()*, which should do an in-place reversal of a string. You cannot use any for/while loops and you can only use `#include<iostream>` and `#include<string>`. You cannot declare any additional strings, i.e. the reverse operation must happen in-place, but you may declare an additional `char`. You may write helper functions as long as they follow the same rules.

```
std::string a = "ThisIsEven";
std::string b = "ThisIsOdd";
std::string c(a);
std::string d(b);
```

```
reverse(c);
reverse(d);

std::cout << a << " reversed is " << c << std::endl;
std::cout << b << " reversed is " << d << std::endl;
```

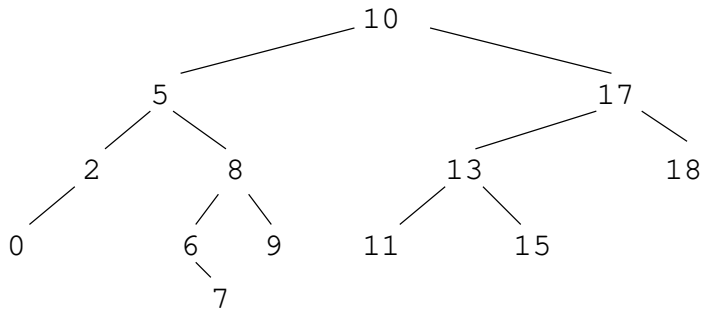
Outputs:

```
ThisIsEven reversed is nevEsIsihT
ThisIsOdd reversed is ddOsIsihT
```

Solution:

```
void reverse(std::string& str){
    if(str.length()>1){
        reverse(str,0,str.length()-1);
    }
}

void reverse(std::string& str, int i, int j){
    if(i>j){
        return;
    }
    char c = str[i];
    str[i] = str[j];
    str[j] = c;
    reverse(str,i+1,j-1);
}
```

2 Tree Drawings [/ 17]



2.1 Tree Construction [/5]

Write a list of calls to *insert()* in order that would create the binary search tree shown above. You can simply write `insert(300); insert(240);` etc. but the ordering should be clear. You do not need to declare any variables.

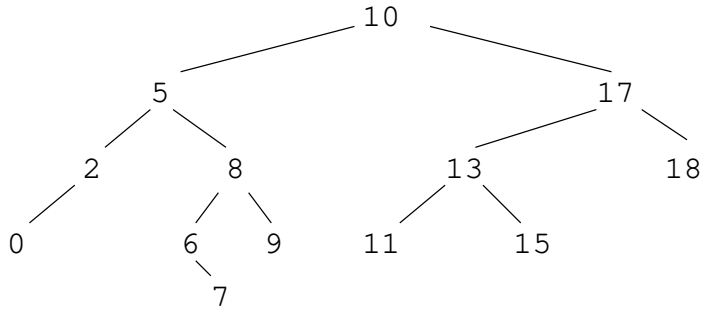
Solution: Many orderings, but an easy one is BFS traversal

```
insert(10);
insert(5);
insert(17);
insert(2);
insert(8);
insert(13);
insert(18);
insert(0);
insert(6);
insert(9);
insert(11);
insert(15);
insert(7);
```

2.2 Post-order Traversal [/6]

Next, write the post-order traversal of this tree:

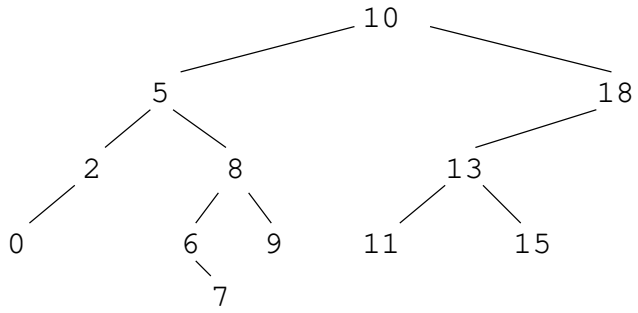
Solution: 0 2 7 6 9 8 5 11 15 13 18 17 10



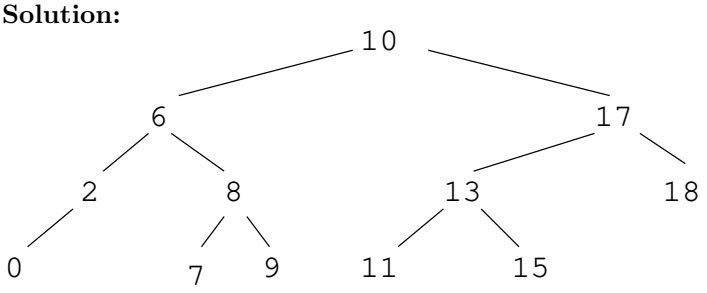
2.3 Erasing From Trees [/6]

Draw the resulting trees from the following `erase` calls. If you need to choose between the left and right subtree, choose the right subtree. Assume this is two independent calls to the binary search tree shown above, in other words each solution should have 12 nodes.

`erase(17)`
Solution:



`erase(5)`
Solution:



3 Linked Trees [/ 25]

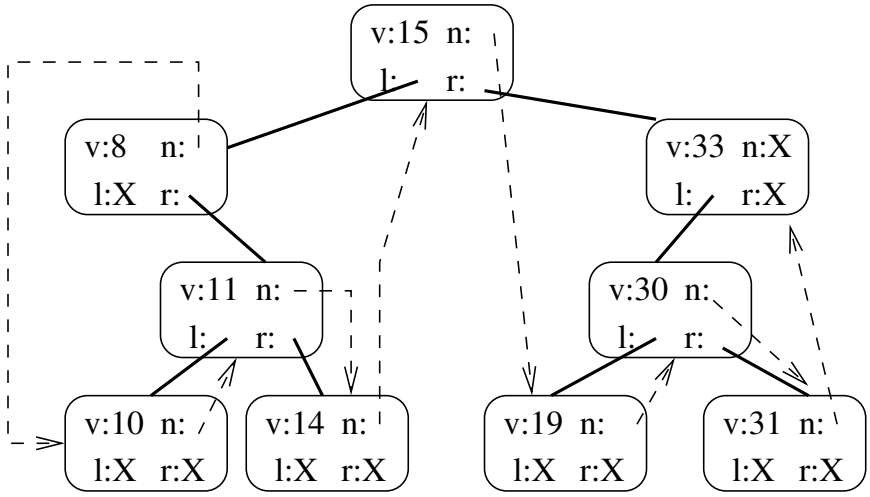
Ben Bitdiddle, Data Structures extraordinaire, is concerned about the running time of his binary search tree iterator's increment operation. Fearing that it might have to go through many nodes, he wants a data structure that makes his operator++() easy to write, an invention he calls "Linked Trees." The modified implementation of TreeNode is below. The new *next* member variable points to the next in-order node for the tree.

```
class TreeNode{
    TreeNode(int init): value(init), left(NULL), right(NULL),
    next(NULL) {}
    int value;
    TreeNode* left;
    TreeNode* right;
    TreeNode* next;
};
```

3.1 Visualizing the Tree [/ 5]

First, for the BST below, draw in all the *next* pointers to make it a "linked tree". Use a **X** to denote a NULL pointer, and try to avoid crossing lines:

Solution:



With Ben's addition, what will be *linked_tree_iterator::operator++()*'s worst case running time, assuming n nodes in the tree and a maximum height of h?

Solution: It will always be $O(1)$ since we can just follow the *next* pointer.

3.2 Writing *insert()* [/ 20]

Implement *insert()* for the linked tree. Similarly to the insert we've done in labs, your function should take in a value to insert and a root pointer that represents the root of a binary search tree. To simplify things, your function should simply return a boolean indicating if the element was added to the tree. You cannot use *linked_tree_iterator::operator--*. Make sure that *insert()* still runs in $O(h)$ time, where h is the height of the tree.

Solution:

```
//driver function
bool insert(int val, TreeNode*&p){
    TreeNode* start = p;
    //get the leftmost element
    while(start && start->left)
        start = start->left;
    return insert(val,p,start);
}

bool insert(int val, TreeNode*& p, TreeNode* first, TreeNode* prev = NULL){
    //we've reached a leaf node
    if(!p){
        //set p to a new node (passed by reference, so parent auto updates)
        p = new TreeNode(val);
        //this means this element is not the first, so we can just use prev's next
        if(prev){
            p->next = prev->next;
            prev->next = p;
        }
        else //otherwise, p's next is the first element
            p->next = first;
        return true;
    }
    else if (val < p->value) //if we go left, prev should not be changed
      return insert(val, p->left, first, prev);
    else if (val > p->value) //if we go right, prev should be p
      return insert(val, p->right, first, p);
      else //element already exists in the set
        return false;
}
```

4 Actor Voting [/ 22]

In this problem you will fill in the blanks in a function *bestActors()*, which takes in an STL `set` of strings which are actor names, and an STL `vector` of strings, where every entry in the vector is one vote for an actor. The three actors with the highest numbers of votes should be printed, ordered by the number of votes. The worst actor (the one with the smallest number of votes) should also be printed. Ties should be resolved by choosing the name that comes first alphabetically. Assume that there will be at least 4 actors. **See the next page for instructions on answering the question.**

As an example, if there were 2 votes for "KevinBacon", 3 votes for "OwenWilson", 2 votes for "LukeWilson", 1 vote for "JackBlack", and 5 votes for "MarilynMonroe", the output should be:

```
Actor #1: MarilynMonroe
Actor #2: OwenWilson
Actor #3: KevinBacon
Worst actor: JackBlack
```

```
typedef /***FILL IN #1***/ TallyType;
typedef /***FILL IN #2***/ DataType;

void bestActors(/***FILL IN #3***/ actors, /***FILL IN #4***/ votes) {
    std::set<std::string>::const_iterator a_itr;
    TallyType tally;
    for (a_itr = actors.begin(); a_itr != actors.end(); a_itr++) {
        /***FILL IN #5***/ ;
    }

    for (int i=0; i<votes.size(); i++) {
        /***FILL IN #6***/ ;
    }

    DataType data;
    TallyType::iterator t_itr;

    for (t_itr = tally.begin(); t_itr != tally.end(); t_itr++) {
        data[ /***FILL IN #7***/ ].insert( /***FILL IN #8***/ );
    }

    DataType::iterator d_itr = /***FILL IN #9***/;
    std::set<std::string>::const_iterator s_itr = /***FILL IN #10***/;
    for(int i=0; i<3; i++){
        std::cout << "Actor #" << i+1 << ": " << *s_itr << std::endl;
        s_itr++;
        if(s_itr == /***FILL IN #11***/){
            d_itr--;
            s_itr = /***FILL IN #12***/;
        }
    }

    d_itr = /***FILL IN #13***/; s_itr = /***FILL IN #14***/;
    std::cout << "Worst actor: " << *s_itr << std::endl;
}
```

4.1 Fill In the Blanks [/ 22]

Using the code bank and empty boxes below, write which letter corresponds to the correct code fragment for each "FILL IN" comment. You cannot use a letter more than once.

A) `map<int, set<string> >`

B) `map<string, int>`

C) `const set<string>&`

D) `const vector<string>&`

E) `tally[votes[i]]++`

F) `tally[*a_itr]++`

G) `t_itr->first`

H) `--data.end()`

I) `--data.end()`

J) `map<int, set<string> >`

K) `map<string, int>`

L) `set<string>`

M) `vector<string>`

N) `tally[votes[i]] = 0`

O) `tally[*a_itr] = 0`

P) `t_itr->second`

Q) `data.end()`

R) `data.end()--`

S) `d_itr->second.begin()`

T) `d_itr->second.begin()`

U) `d_itr->second.begin()`

V) `d_itr->second.end()`

W) `d_itr->second.end()`

X) `d_itr->second.end()`

Y) `data.end()--`

Z) `data.begin()`

FILL IN #1: **B/K**

FILL IN #2: **A/J**

FILL IN #3: **C**

FILL IN #4: **D**

FILL IN #5: **O**

FILL IN #6: **E**

FILL IN #7: **P**

FILL IN #8: **G**

FILL IN #9: **H/I**

FILL IN #10: **S/T/U**

FILL IN #11: **V/W/X**

FILL IN #12: **S/T/U**

FILL IN #13: **Z**

FILL IN #14: **S/T/U**

5 Set Difference [/ 27]

A set difference of two sets, A and B, (sometimes written A-B) returns everything in A that is not in B. The ordering should be preserved. For example:

```
A: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
B: 5 10 15
A-B: 1 2 3 4 6 7 8 9 11 12 13 14
```

Ben Bitdiddle is told to write a function to calculate the set difference. He is told he cannot use std::set_difference for this problem (found in `<algorithm>`), but that he is free to use any return type. Since the result is supposed to be sorted, he decides to implement a version returning a set, *SlowSetDiffSet()*. Alyssa P. Hacker looks over Ben's solution and tells him that it is inefficient:

```
typedef std::set<int> intset;
typedef intset::const_iterator int_it;

std::set<int> SlowSetDiffSet(const intset& s1, const intset& s2){
    std::set<int> ret;
    for(int_it it1 = s1.begin(); it1 != s1.end(); it1++){
```

```
        if(s2.find(*it1) == s2.end()){
            ret.insert(*it1);
        }
    }
    return ret;
}
```

Ben thinks about it, and realizes that he could do this faster if he returned a vector instead, and writes *SlowSetDiffVec()*. Alyssa takes another look at his code, and tells him that switching to a vector return type is a good idea, but he could do this faster if he used two iterators and did not call `std::set<T>::find()` - she says to call this function *FastSetDiff()* and that if `s1` has k elements and `s2` has n elements, that it can be done in $O(k)$ time since the inputs are already sorted.

```
std::vector<int> SlowSetDiffVec(const intset& s1, const intset& s2){
    std::vector<int> ret;
    for(int_it it1 = s1.begin(); it1 != s1.end(); it1++){
        if(s2.find(*it1) == s2.end()){
            ret.push_back(*it1);
        }
    }
    return ret;
}
```

5.1 SlowSetDiff Performance [/ 8]

If `s1` has k elements and `s2` has n elements, what are the time complexities of *SlowSetDiffSet()* and *SlowSetDiffVec()*?

SlowSetDiffSet: $O(k * log\ n + k * log\ k) = O(k * (log\ n + log\ k))$. For each of the k elements in s1, we call find in s2, which is $O(logn)$. All the searching takes $O(k\ log\ n)$. If none of s2 is in s1, then we will insert k things into ret, which is a set. This takes $O(k\ log\ k)$ time.

SlowSetDiffVec: $O(k * log\ n + k) = O(k * log\ n)$. Just like in SlowSetDiffSet, searching takes $O(k\ log\ n)$ time. However, insertion is now k operations each taking $O(1)$ on average, meaning the insertions take $O(k + n)$ in total.

5.2 FastSetDiff Implementation [/ 19]

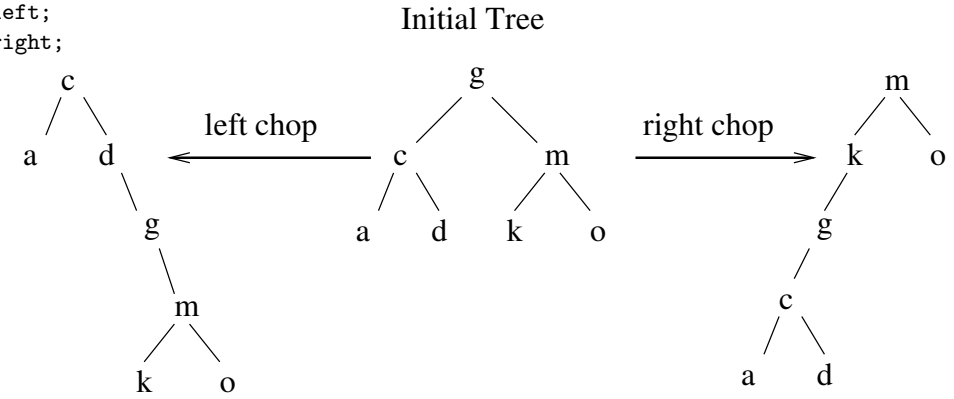
```
std::vector<int> FastSetDiff(const intset& s1, const intset& s2){
    std::vector<int> ret;
    int_it sit1 = s1.begin();
    int_it sit2 = s2.begin();
    while (sit1 != s1.end()) {
        // nothing more to "remove" from set A
        if (sit2 == s2.end()) {
            ret.push_back(*sit1);
            ++sit1;
        }
        // s2 iterator needs to be advanced until it's >= *sit1
        else if (*sit2 < *sit1) {
            ++sit2;
        }
        // s2 itr and s1 itr equal, advance both and don't add
        else if (*sit2 == *sit1) {
            ++sit2;
            ++sit1;
        }
        // s1 itr is behind s2, which means we have things in s1 that *aren't* in s2, to add to return vec
        else {
            ret.push_back(*sit1);
            ++sit1;
        }
    }
    return ret;
}
```

6 Tree Cut [/ 18]

In this problem you will implement a function called *TreeChop()* which takes in a TreeNode* pointing to the root of a binary search tree, and a `bool` which is true if we want to chop the left side, and false if we want to chop the right side. When a "chop" is done to the left side, the left child of the root becomes the new root, and the old root (along with its right subtree) become part of the new root's right subtree. If the chop happens on the right, the process is symmetric to a left chop. After the chop, the resulting tree should still be a binary search tree. *Hint: think about how erase works in a BST.*

Below is a TreeNode you can use, as well as examples of both a left and right chop on a tree:

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



You can assume that the root exists and that it has a left and right child. Do not use `new/delete`. Change no more than 2 pointers *inside TreeNode*s.

```
template <class T>
void TreeChop(TreeNode<T>*& root, bool left){
    if(left){
        //Find the rightmost thing in the left tree
        TreeNode<T>* search = root->left;
        while(search->right){
            search = search->right;
        }
        search->right = root; //Link root to the bottom right of left subtree
        root = root->left; //Update root
        search->right->left = NULL; //Make the old root's left pointer null
    }
    else{
        //Find the leftmost thing in the right tree
        TreeNode<T>* search = root->right;
        while(search->left){
            search = search->left;
        }
        search->left = root; //Link root to the bottom right of left subtree
        root = root->right; //Update root
        search->left->right = NULL; //Make the old root's right pointer null
    }
}
```

7 Restaurant Maps [/ 40]

A restaurant owner has started some code to help manage orders, but they aren't sure how to finish it. They would like for the kitchen to be able to view individual orders, so they have already done some typedefs and made an *Item* class (representing one food or drink item), an *Order* class (representing all the Items that would be on one paper order), and a *Diner* class representing one customer (a customer may have multiple Orders in one visit). The owner wants to use this both for the kitchen to manage orders, and for the server to be able to figure out the total cost of meals at the end of a visit. Since split checks are requested so often, the restaurant always figures out individual totals instead of handling groups.

The code that is already written:

```
class Diner{
public:
    Diner() {}
    Diner(const std::string& name) : name_(name) {}
    const std::string& getName() const { return name_; }
private:
    std::string name_;
};

class Item{
public:
    Item(const std::string& name, float price) : name_(name), price_(price) {}
    const std::string& getName() const { return name_; }
    float getPrice() const { return price_; }
private:
    std::string name_;
    float price_;
};

class Order{
public:
    Order() : order_id(-1) {}
    Order(int id) : order_id(id) {}
    void AddItem(const Item& item) { items.push_back(item); }
    const std::vector<Item>& getItems() const { return items; }
    int getID() const { return order_id; }
private:
    int order_id;
    std::vector<Item> items;
```

```
};

bool operator<(const Diner& d1, const Diner& d2){
    return d1.getName() < d2.getName();
}

bool operator<(const Order& o1, const Order& o2){
    return o1.getID() < o2.getID();
}

//////////////End classes/////////////

typedef std::map<Order, Diner> ORDERS_TYPE;
typedef std::map<Diner, float> BILL_TYPE;
```

An example of the main part of the program is:

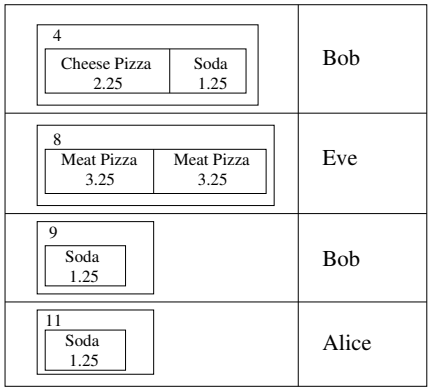
```
ORDERS_TYPE orders;
Item cheese_pizza("Cheese Pizza",2.25);
Item meat_pizza("Meat Pizza",3.25);
Item soda("Soda",1.25);

Diner d1("Bob");
Diner d2("Eve");
Diner d3("Alice");

AddToOrder(orders,4,cheese_pizza,d1);
AddToOrder(orders,8,meat_pizza,d2);
AddToOrder(orders,8,meat_pizza,d2);
AddToOrder(orders,4,soda,d1);
AddToOrder(orders,9,soda,d1);
AddToOrder(orders,11,soda,d3);

BILL_TYPE bills = CalculateBills(orders);
PrintBills(bills); //Assume this is already written.
```

And the output:

```
Alice owes $1.25
Bob owes $4.75
Eve owes $6.50
```

7.1 Visualizing *ORDERS_TYPE* [/ 8]

First, draw the abstract representation of `orders` the way it would look after the example code above has run:



7.2 *AddToOrder* [/ 18]

Next, implement the *AddToOrder* function, which should add a given item to an order. You can assume that we will only use one diner per order ID. An order can contain the same item several times.

```
void AddToOrder(ORDERS_TYPE& orders, int order_id, const Item& i, const Diner& d){
    //Search for the order ID to see if it exists
    ORDERS_TYPE::iterator it;

//Slow linear search to avoid constructing Order object
    for(it = orders.begin(); it!=orders.end(); it++){
//If we need to assume order IDs are not unique
        //if(it->first.getID() == order_id && it->second.getName() == d.getName()){
        if(it->first.getID() == order_id){
            break;
        }
    }

//Alternate method that removes the O(o) search above
    //it = orders.find(Order(order_id));

    Order o;
    if(it == orders.end()){
        o = Order(order_id);
    }
    else{
        o = it->first;
        orders.erase(it);
    }
    o.AddItem(i);
    orders[o] = d;
}
```

Using the first method, $O(o)$ to find if the order exists, otherwise $O(log\ o)$. $O(log\ o)$ to erase an existing order (because the key is constant, and is the Order). $O(log\ o)$ to insert a new Order (which always happens). Adding an item is $O(1)$, copying an Order is $O(i)$. Dropping constants, overall we have $O(o + i + log\ o) => O(o + i)$ using the first method, or $O(i)$ using the alternate method.

7.3 *CalculateBills* [/ 9]

Next, implement the *CalculateBills* function, which should figure out how much each diner owes the restaurant. This function should not do any printing.

```
BILL_TYPE CalculateBills(const ORDERS_TYPE& orders){
    BILL_TYPE ret;
    for(ORDERS_TYPE::const_iterator it = orders.begin(); it!=orders.end(); it++){
        for(std::size_t i = 0; i<it->first.getItems().size(); i++){
            ret[it->second] += it->first.getItems()[i].getPrice();
        }
    }
    return ret;
}
```

We traverse o orders, and for each one we have to get the price of all items. In total we look at i items spending per item to $O(1)$ looking up the cost. If every order contains 1 item, but every diner has a lot of orders, we will be doing a lot of map lookups for insertion, overall we will do $O(o*i*log\ d)$ insertions. So this gives $O(o*i*log\ d + o*i) => O(o*i*log\ d)$.

7.4 *Performance Discussion* [/ 5]

The owner of the restauarant is not happy with how slow *AddToOrder()* is. They insist that since the program is using a `map`, this should be fast. Using 2-4 complete sentences, briefly explain how you could change `ORDERS_TYPE` to speed up the program, and why it would improve the speed.

`std::map<int, std::pair<Order,Diner> >` is a good example. Really anything that lets us look up by ID, since that's easy to pass in and doesn't change once it's set. Looking up by Order is a problem because the Order object is complicated, we can't just call .find() without a lot of work, or without designing the class the way it was here, where `operator<` is non-intuitive and only depends on ID, not diner name or anything else. Even if we do leverage the fact we only need ID to compare Order objects, Order is a bad primary key because we can't change a key, so we have to keep erasing and inserting, which is very annoying and a lot of extra operations.

Note: If you used find(), then you probably didn't see a huge slow-down, but historically most students do not use .find() because they assume you need `Order::operator==`. If you have `Order::operator<` and `a<b` is false, and `b<a` is false, then inherently `a==b`, and this is how the STL `map` handles it. You still should be thinking about how to eliminate the extra erase/insert/copying of Order - if we didn't have to keep copying the object, we could get this down to $O(log\ o)$.

Looking up by Diner is a problem because a diner may have many orders, but we could do something like `std::map<Diner, std::vector<Order> >`. If one diner has a lot of orders though, this will be less efficient than just using the order_id as a key. Unique keys that don't change are where maps make the most sense.

8 Short Answers [/ 12]

a) depth-first search
b) breadth-first search
c) pre-order traversal
d) in-order traversal
e) post-order traversal
f) narrower and taller
g) wider and shorter

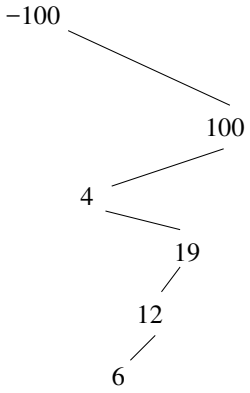
For each of the statements below, write a single letter corresponding to the phrase that fills in the bold **???** to make the statement true. No letter is used more than once. Each incorrect answer will decrease your score by 4 points (but cannot make your score on this problem negative).

d `std::map` and `std::set` iterator's `operator++` use a(n) **???** to move through the tree.

a `ds_set::find` is an example of a(n) **???**.

e There exists a valid BST if [6, 12, 19, 4, 100, -100] is the **???** of the tree.

g B+ trees are usually **???** than balanced binary search trees of similar size.

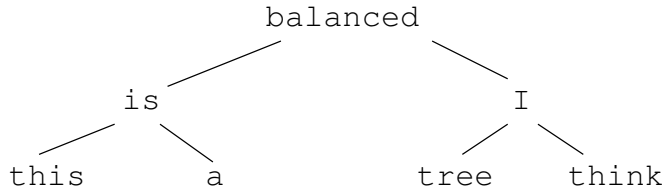


9 Short Answers - Trees [/12]

9.1 In-Order Traversal [/3]

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

Solution:

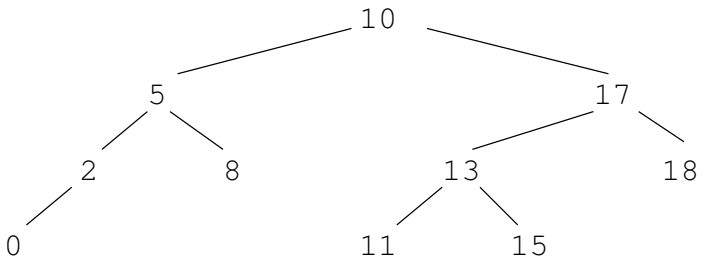


9.2 Balanced Trees in Structures [/2]

Explain in 1-2 complete sentences: Why do data structures like STL `set`, STL `map`, and `ds_set` need to have their internal tree representation be balanced to have fast performance?

Solution: Several functions such as `find` and `iterator operator++` may have to traverse the entire height of the internal tree structures. If the tree is balanced, then the height will be O($logn$), while if it's unbalanced the height may be O(n).

For the next two questions, consider the following binary search tree:



9.3 Pre-Order Traversal [/3]

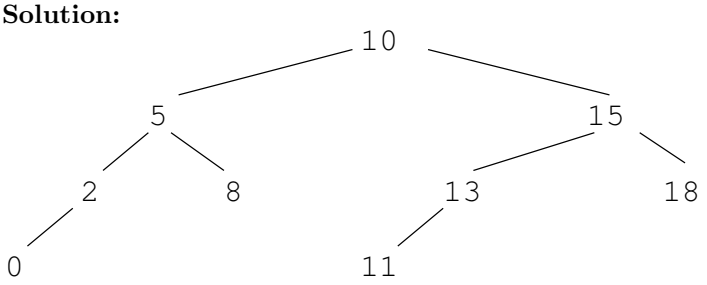
First write the pre-order traversal of this tree:

Solution: 10 5 2 0 8 17 13 11 15 18

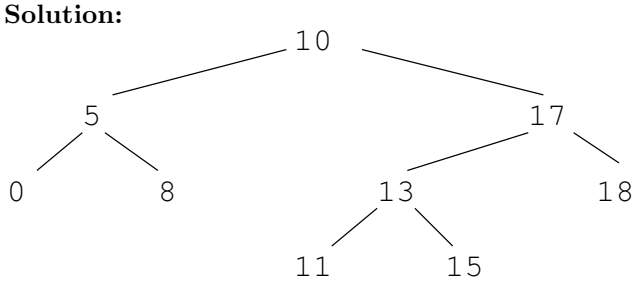
9.4 Erasing From Trees [/4]

Draw the resulting trees from the following `erase` calls. Assume this is two independent calls to the tree shown above, and not `erase(17); erase(2);`. If you need to choose between the left and right subtree, choose the left subtree.

`erase(17)`
Solution:



`erase(2)`
Solution:



10 Reciprocal Best Hits [/25]

Biologists often search large databases of genomic sequences looking for partial matches of a sequence of interest against the database sequences. One technique that is used to find related gene sequences is called *reciprocal best hit*. You don't have to know anything about genomics to understand the idea. A query sequence is a string of letters, such as "ATGGGCGA..." The database consists of many such strings. Each query sequence is matched against every other sequence in the data base. The matches to other sequences are likely to be partial rather than exact matches. Each match is given a score. If two sequences have each other as their best scores (called hits) they are considered to be possibly related. For example, if SequenceA's best score is with SequenceC and SequenceC's best score is with SequenceA, they are consider as possibly related.

10.1 `RBH` Best Hit Map [/5]

For our purposes, the data is stored in a map,
std::map <std::string, std::map<std::string, float> > called *sequences*. The first string in the map is called the query. Each string is a sequence ID, such as "SequenceA" and the float is the match score of the query sequence against the second sequence. The values of the floats range from 0 (no match) to 100 (perfect match). Scores below 10 are not displayed

Suppose for some small set of queries SequenceA has scores 75.1 when matched with SequenceC and 12.4 with SequenceD. SequenceB has a score of 11.7 with SequenceE, and SequenceC has scores of 81.5 with SequenceA and 21.7 with SequenceF.

First draw the structure of the *sequences* map where SequenceA, SequenceB, and SequenceC are the query sequences.

Solution:

SequenceA	SequenceC 75.1 SequenceD 12.4
SequenceB	SequenceA 11.7
SequenceC	SequenceA 81.5 SequenceF 21.7

10.2 RBH Best Hits [/10]

Next, write code to search the map of maps and find the sequence ID with the best score for each query sequence. The name of the sequence and its best scoring partner are to be stored in a new map called *best_hits*, **std::map, <std::string, std::string >**. Don't store the score of a sequence against itself, if found. That is, your map should not contain entries where the key and the value are the same sequence ID. For example, your map should no contain key-value pairs such as SequenceA as key and SequenceA as a value. Use const where appropriate.

You don't need *include* statements. You can assume *using namespace std;*

Solution:

```
  std::map<std::string, std::string> best_hits;
  std::map<std::string, std::map<std::string, float> >::const_iterator it;
  for(it = sequences.begin(); it != sequences.end(); ++it) {
    std::map<std::string, float> hits = it->second;
    std::map<std::string, float>::const_iterator it2;;
    std::string best_hit_name;
    float max_hit = -1;
    for(it2 = hits.begin(); it2 != hits.end(); ++it2) {
      if (it->first == it2->first)
continue;
      if (it2->second > max_hit) {
max_hit = it2->second;
best_hit_name = it2->first;
      }
    }
    best_hits[it->first] = best_hit_name;
  }
```

10.3 RBH Reciprocal Hits [/10]

Now, write some code to search the map of sequence ID pairs to find and print the reciprocal best hit pairs; that is, cases such as where SequenceA's best score is with SequenceC, and SequenceC's best score is with SequenceA. Don't print duplicate messages.

You don't need *include* statements. You can assume *using namespace std;*

Here is the typical output

```
SequenceA has a reciprocal best hit with SequenceC
SequenceD has a reciprocal best hit with SequenceK
```

Solution:

```
  std::map<std::string, std::string>::const_iterator it3;
  for(it3 = best_hits.begin(); it3 != best_hits.end(); ++it3) {
    if (it3->first > it3->second)
      continue;
    std::map<std::string, std::string>::iterator it2;
    it2 = best_hits.find(it3->second);
    if (it2 != best_hits.end() && it2->second == it3->first) {
      std::cout << it3->first << " has a reciprocal best hit with " << it3->second << std::endl;
    }
  }
```

11 Strange BST Tree Sort [/35]

For this problem, you will be helping Ben Bitdiddle implement some functions that he claims will lead to a faster sort for binary search trees. *All trees in this problem are binary search trees.* To start with, you are given the code for the TreeNode class:

```
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;
};
```

11.1 FindLargestInTree [/11]

Write a function called `FindLargestInTree` that takes a *TreeNode* pointer to the root of a tree and returns the largest *value* in the tree. Remember to use const and & where appropriate. This function should work for any type T that has `operator<` defined.

Solution:

```
template <class T>
const T& FindLargestInTree(TreeNode<T>* root){
  while(root->right){
    root = root->right;
  }
  return root->value;
}
```

If n is the number of nodes in the tree, and h is the height of the tree, what is the worst-case running time of `FindLargestTree`? *Hint: use the tightest (smallest) bound that is still accurate. All functions are technically O(∞) but that won't earn you any points.*

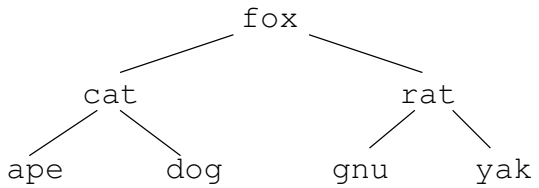
Solution: O(h)

11.2 FindSmallestInRange [/11]

The next thing Ben wants you to do is write a function that will find the smallest value *ret* in the tree starting from *root* such that $a < ret < b$. You can assume there are no duplicates inside the tree and that a is a value in the tree. If no value of *ret* can satisfy the constraints, the return value should be NULL. To help you get started, he has written the following function:

```
template <class T>
TreeNode<T>* FindSmallestInRange(const T& a, const T& b, TreeNode<T>* root){
  if(!root){
    return NULL;
  }

  T best_value = FindLargestInTree(root);
  TreeNode<T>* ret = FindSmallestInRange(a, b, root, best_value);
  if(best_value >= b){
    return NULL;
  }
  return ret;
}
```

Below is an example tree, and some sample calls and the corresponding output.




```
cout << FindSmallestInRange("fox","zebra",root)->value << endl;
cout << FindSmallestInRange("ape","dog",root)->value << endl;
cout << FindSmallestInRange("ape","cat",root) << endl;

gnu
cat
0x0
```

Your task is to fill in the helper function in the box on the next page.

```
template <class T>
TreeNode<T>* FindSmallestInRange(const T& a, const T& b, TreeNode<T>* root, T& best_value){
```

Solution:

```
  if(!root){
    return NULL;
  }

  TreeNode<T>* left_subtree = FindSmallestInRange(a,b,root->left,best_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;

}
```

If n is the number of nodes in the tree, what is the worst-case running time of `FindSmallestInRange`?

Solution: O(n)

11.3 Writing `TreeSort` [/13]

Finally Ben says you are ready to write his sort function, `TreeSort`. The function should take in a pointer to a `TreeNode` which is the root of the tree. Your function should return a `vector` with the values in order from smallest to largest. You should not use `std::sort` or STL `map`, `set`, or `list`. Furthermore, Ben insists that you don't use the *left*, *right*, or *parent* pointers in your implementation, though it's okay to call `FindLargestInTree` and `FindSmallestInRange`. It's also okay to call `FindSmallestInTree`, even though you didn't implement it.

A sample code fragment using `TreeSort`:

```
vector<string> sorted = TreeSort(root);
for(unsigned int i=0; i<sorted.size(); i++){
  cout << sorted[i] << " ";
}

ape cat dog fox gnu rat yak
```

Solution:

```
template <class T>
std::vector<T> TreeSort(TreeNode<T>* root){
  std::vector<T> ret;
  const T& smallest = FindSmallestInTree(root);
  const T& largest = FindLargestInTree(root);

  ret.push_back(smallest);
  TreeNode<T>* find = FindSmallestInRange(ret[ret.size()-1],largest,root);
  while(find){
    ret.push_back(find->value);
    find = FindSmallestInRange(ret[ret.size()-1],largest,root);
  }
  ret.push_back(largest);
  return ret;
}
```

If `FindSmallestInRange` runs in O($f(n)$) time, and both `FindLargestInTree` and `FindSmallestInTree` run in O($g(n)$) time, and there are n nodes, what's the worst-case run time for `TreeSort()`? Use $f(n)$ and $g(n)$ where appropriate, even if you expressed your run times in the earlier parts of the problem using n. Is this run time better, the same, or worse than doing an in-order traversal instead of Ben's method?

Solution: O($g(n) + n * f(n)$). An in-order traversal will be O(n) and each node will only be visited a maximum of 3 times (once on the way down to the left, once coming back up from the left, and once coming back up from the right). In Ben's approach we have to do n calls to `FindSmallestInRange` which must be worse. Plugging in the O(n) from 3.2's solution, this TreeSort becomes O(n^2).

12 Set Overlap [/15]

Given two sets of integers, `set1` and `set2`, write an *efficient* code fragment that creates a third set of integers, `set3`, containing all of the values from `set1` that are also contained in `set2`. When an element is found, it should be removed from `set2`.

Solution:

```
std::set<int> s3;
for (std::set<int>::iterator it = s1.begin(); it != s1.end(); ++it) {
  std::set<int>::iterator it2 = s2.find(*it);
  if (it2 != s2.end()) {
    s3.insert(*it);
    s2.erase(it2);
  }
}
```

If set 1 has k elements and set 2 has n elements, what is the order of your method?

Solution: O(k (log n + log k))

13 Recursion Redux [/10]

For the following two problems, do not use any `for` or `while` loops.

13.1 Recursive Letter Counting [/6]

Fill in the missing code for this pair of functions that will count how many times the letter E shows up in *str*.

```
int EInString(const std::string& str, int index){
```

Solution:

```
  if(index>str.length()){
    return 0;
  }
  if(str[index]=='E'){
    return 1 + EInString(str,index+1);
  }
  return EInString(str,index+1);

}

int EInString(const std::string& str){
  return EInString(str,0);
}
```

13.2 Recursive `for` Loop [/4]

Write a recursive function `RecursiveFor()` which replaces the following for loop with `RecursiveFor(0,35);`

```
for(int i=0; i<35; i++){
  x();
}
```

Solution:

```
void RecursiveFor(int i,int max){
  if(i<max){
    x();
    RecursiveFor(i+1,max);
  }
}
```