-Function returns everything in set s1 that is not in set s2

```
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;
}
```

-Function takes in a TreeNode* pointing to the root of a BST, and a bool which is true if you want to chop the left side, and false if you want to chop the right

```
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
    }
}
```

-std::map and std::set iterator's operator++ use an in-order transversal to move through a tree

-ds_set::find is an example of a depth-first search

-There exists a valid BST if [6,12,19,4,100,-100] is the post-order traversal of the tree

-If a tree is balanced, then the height will be O(logn). If it's unbalanced, the height may be O(n).

-Function takes a TreeNode pointer to the root of a tree and returns the largest value in the tree

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

Time Complexity: O(h)

-Function finds the smallest value in the tree starting from root such that a<ret<b

```
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,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;

}
```

Time Complexity: O(n)

Lab Section 6
TA: Maurício
Mentors: John, Fred, Alec, Matt

-Function returns a vector with the values of the tree in order from smallest to largest

```
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;
}
```

Time Complexity: O(g(n)+n*f(n))

-Program creates a third set of integers containing all of the values from set1 that are also contained in set2

```
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);
  }
}
```

Time Complexity: O(k(log(n)+log(k))

-Function counts how many times the letter 'E' shows up in a string

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

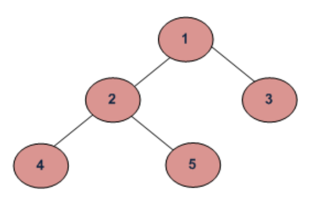
-Recursive function returns false if placing *value* within a subtree of *node* violates the BST property of the whole tree, and true otherwise

```
bool BelongsInSubtree(Node* node, int value) {
  if (node == NULL) return false;
  // check for duplicate
  if (node->value == value) return false;
  // made it to the root!  this value fits on this branch
  if (node->parent == NULL) return true;
  // doesn't belong to the left of the grandparent
  if (node->value < node->parent->value && value > node->parent->value) return false
  // doesn't belong in the right subtree of the grandparent
  if (node->value > node->parent->value && value < node->parent->value) return false
  return BelongsInSubtree(node->parent,value);
}
```

-Function clears all allocated memory associated with the upside-down tree

```
void DestroyTree(std::list<Node*> &leaves) {
  // use an STL set to collect all tree nodes (removes duplicates)
  std::set<Node*> nodes;
  for (std::list<Node*>::iterator itr = leaves.begin(); itr != leaves.end(); itr++)
    Node* tmp = *itr;
    while (tmp != NULL) {
      if (!nodes.insert(tmp).second)
        break;
      tmp = tmp->parent;
    }
  }
  // now delete everything
  for (std::set<Node*>::iterator itr = nodes.begin(); itr != nodes.end(); itr++) {
    delete *itr;
  }
  // set the tree to the empty tree
  leaves.clear();
}
```

Time Complexity: O(nlog(n))



-the keys in the left subtree are less than the key in the parent node; **L < P**
- the keys in the right subtree are greater than the key in the parent node; **R > P**

Depth First Traversals:
(a) Inorder (Left,Root,Right) : 4 2 5 1 3
(b) Preorder (Root,Left,Right) : 1 2 4 5 3
(c) Postorder (Left,Right,Root) : 4 5 2 3 1
(d) Breadth-First/Level Order : 1 2 3 4 5

```cpp
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;
```

Time Complexity: O(n+log(n))=>O(n)

```cpp
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;
    for(it = orders.begin(); it!=orders.end(); it++){
        if(it->first.getID() == order_id){
            break;
        }
    }

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

Time Complexity: O(n*i*log(d))

```cpp
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;
}
```

4: Cheese Pizza 2.25, Soda 1.25	Bob
8: Meat Pizza 3.25, Meat Pizza 3.25	Eve
9: Soda 1.25	Bob
11: Soda 1.25	Alice

Abstract Representation of **orders**

-Function determines if a Node* can be considered a tree

```cpp
template <class T> bool is_tree(Node<T> *n, std::set<Node<T>*> &already_seen) {
  if (n == NULL) return true;
  if (already_seen.find(n) != already_seen.end()) return false;
  already_seen.insert(n);
  for (int i = 0; i < n->children.size(); i++) {
    if (!is_tree(n->children[i],already_seen))
      return false;
  }
  return true;
}


template <class T> bool is_tree(Node<T> *n) {
  std::set<Node<T>*> already_seen;
  return is_tree(n,already_seen);
}
```

-STL maps store pairs of “associated” values
-Map iterators refer to pairs
-Map search,insert,and erase:O(logn)
-Maps are ordered by increasing value of the key
-pairs are a templated struct with just 2 members
-Every leaf has the same depth in an exactly balanced tree

-Hash tables don’t store data in sorted order
-A hash table is implemented with an array at the top level
-A hash table has constant time access
-A hash function takes in 1 argument, and returns an integer index
-Hash function has a fast O(1) computation

-Function determines if a Node* can be considered a binary tree

```cpp
template <class T> bool is_tree(Node<T> *n, std::set<Node<T>*> &already_seen) {
  if (n == NULL) return true;
  if (already_seen.find(n) != already_seen.end()) return false;
  already_seen.insert(n);
  for (int i = 0; i < n->children.size(); i++) {
    if (!is_tree(n->children[i],already_seen))
      return false;
  }
  return true;
}


template <class T> bool is_tree(Node<T> *n) {
  std::set<Node<T>*> already_seen;
  return is_tree(n,already_seen);
}
```

-Function determines if a Node* can be considered a binary tree

```cpp
template <class T> bool is_binary(Node<T> *n) {
  if (n == NULL) return true;
  if (n->children.size() != 2) return false;
  for (int i = 0; i < n->children.size(); i++) {
    if (!is_binary(n->children[i]))
      return false;
  }
  return true;

}
```

-Function determines if Node* can be considered a BST

```cpp
template <class T>
bool is_bst(Node<T> *n, Node<T> *lower=NULL, Node<T> *upper=NULL) {
  if (n == NULL) return true;
  if (lower != NULL && n->value < lower->value) return false;
  if (upper != NULL && n->value > upper->value) return false;
  if (!is_bst(n->children[0],lower,n)) return false;
  if (!is_bst(n->children[1],n,upper)) return false;
  return true;

}
```

-Function takes in a list of strings and creates and returns a pointer to a well-balanced binary tree of Nodes with breadth-first traversal order that matches the input

```cpp
Node* construct_breadth(std::list<std::string> input) {
  // note: input passed by copy since the editing below
  // these changes should not be permanent
  if (input.size() == 0) return NULL;
  Node* answer = new Node(input.front());
  input.pop_front();
  // store the current leaves (nodes with no children)
  std::list<Node*> leaves;
  leaves.push_back(answer);
  while (input.size() > 0) {
    // add a left child to the 'next' leaf
    Node* tmp = leaves.front();
    leaves.pop_front();
    tmp->left = new Node(input.front());
    input.pop_front();
    leaves.push_back(tmp->left);
    if (input.size() > 0) {
      // also add a right child
      tmp->right = new Node(input.front());
      input.pop_front();
      leaves.push_back(tmp->right);
    }
  }
  return answer;
}
```

-Function prints nodes of binary tree in postorder traversal

```cpp
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    cout << node->data << " ";
}
```

-Function prints nodes of binary tree in inorder traversal

```cpp
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    cout << node->data << " ";

    /* now recur on right child */
    printInorder(node->right);
}
```

-Function prints nodes of binary tree in preorder traversal

```cpp
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first print data of node */
    cout << node->data << " ";

    /* then recur on left sutree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}
```