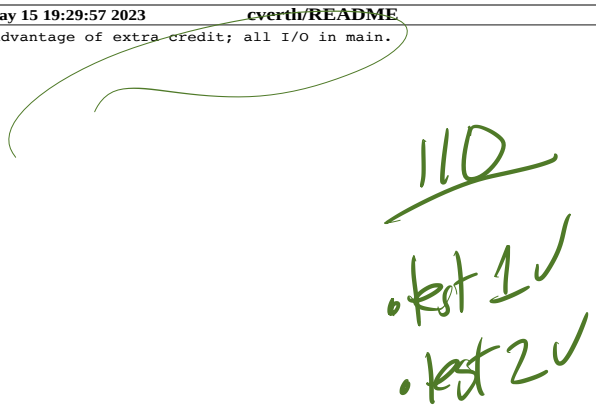
Took advantage of extra credit; all I/O in main.




```
BinarySearchTree::Node* BinarySearchTree::m_min(Node* node) {
   while (node->left) {
      node = node->left;
   }
   return node;
}

BinarySearchTree::Result BinarySearchTree::predecessor(int key) {
   auto keyNode = m_search(m_root, key);
   if (!keyNode) { // 'key' itself not in tree
      return {Status::INVALID_ARGUMENT, 0};
   }
   auto predecessorNode = m_predecessor(keyNode);
   if (!predecessorNode) { // no predecessor
      return {Status::DNE, 0};
   }
   return {Status::FOUND, predecessorNode->key}; // predecessor found
}

BinarySearchTree::Result BinarySearchTree::successor(int key) {
   auto keyNode = m_search(m_root, key);
   if (!keyNode) { // 'key' itself not in tree
      return {Status::INVALID_ARGUMENT, 0};
   }
   auto successorNode = m_successor(keyNode);
   if (!successorNode) { // no successor
      return {Status::DNE, 0};
   }
   return {Status::FOUND, successorNode->key}; // successor found
}

BinarySearchTree::Node* BinarySearchTree::m_predecessor(Node* node) {
   // CASE 1: predecessor is the maximum value "just less" than key of 'node'
   if (node->left) {
      return m_max(node->left);
   }

   // CASE 2: predecessor is the node whose successor is key of 'node'
   auto temp = node->parent;
   while (temp != nullptr && node == temp->left) {
      node = temp;
      temp = temp->parent;
   }
   return temp;
}

BinarySearchTree::Node* BinarySearchTree::m_successor(Node* node) {
   // CASE 1: successor is the minimum value "just more" than key of 'node'
   if (node->right) {
      return m_min(node->right);
   }

   // CASE 2: successor is the node whose predecessor is key of 'node'
   auto temp = node->parent;
   while (temp != nullptr && node == temp->right) {
      node = temp;
      temp = temp->parent;
   }
   return temp;
}

void BinarySearchTree::insert(int key) {
   Node* node = new Node(key);
   Node* temp = m_root; // used to search for position to place 'node'
   Node* prev = nullptr; // trails behind 'temp'
```



```
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file binary_search_tree.cpp
* @assignment 7: Binary Search Tree
*/

#include "binary_search_tree.h"

BinarySearchTree::BinarySearchTree() {}

BinarySearchTree::~BinarySearchTree() {
   m_destruct(m_root);
}

void BinarySearchTree::m_destruct(Node* node) {
   if (node) {
      auto left = node->left;
      auto right = node->right;
      delete node;
      m_destruct(left);
      m_destruct(right);
   }
}

BinarySearchTree::Status BinarySearchTree::search(int key) {
   if (m_search(m_root, key)) {
      return Status::FOUND;
   }
   return Status::DNE;
}

BinarySearchTree::Node* BinarySearchTree::m_search(Node* node, int key) {
   if (!node || node->key == key) {
      return node;
   }
   if (key < node->key) {
      return m_search(node->left, key);
   }
   else {
      return m_search(node->right, key);
   }
}

BinarySearchTree::Result BinarySearchTree::min() {
   if (m_root) { // Min value only exists if tree has nodes
      return {Status::FOUND, m_min(m_root)->key};
   }
   return {Status::DNE, 0};
}

BinarySearchTree::Result BinarySearchTree::max() {
   if (m_root) { // Max value only exists if tree has nodes
      return {Status::FOUND, m_max(m_root)->key};
   }
   return {Status::DNE, 0};
}

BinarySearchTree::Node* BinarySearchTree::m_max(Node* node) {
   while (node->right) {
      node = node->right;
   }
   return node;
}
```



```
   // Search for position to place 'node'
   while (temp) {
      prev = temp;
      if (node->key < temp->key) {
         temp = temp->left;
      }
      else {
         temp = temp->right;
      }
   }

   node->parent = prev; // 'prev' is node above nullptr node

   if (!prev) { // tree empty edge case
      m_root = node;
      return;
   }

   // Sets the children of 'prev'
   if (node->key < prev->key) {
      prev->left = node;
   }
   else {
      prev->right = node;
   }
}

std::vector<int> BinarySearchTree::inOrderTraverse() {
   m_traversalBuffer.clear(); // Clear buffer
   m_inOrderTraverse(m_root);
   return m_traversalBuffer;
}

std::vector<int> BinarySearchTree::postOrderTraverse() {
   m_traversalBuffer.clear(); // Clear buffer
   m_postOrderTraverse(m_root);
   return m_traversalBuffer;
}

std::vector<int> BinarySearchTree::preOrderTraverse() {
   m_traversalBuffer.clear(); // Clear buffer
   m_preOrderTraverse(m_root);
   return m_traversalBuffer;
}

void BinarySearchTree::m_inOrderTraverse(Node* node) {
   if (node) {
      m_inOrderTraverse(node->left);
      m_traversalBuffer.push_back(node->key);
      m_inOrderTraverse(node->right);
   }
}

void BinarySearchTree::m_postOrderTraverse(Node* node) {
   if (node) {
      m_postOrderTraverse(node->left);
      m_postOrderTraverse(node->right);
      m_traversalBuffer.push_back(node->key);
   }
}

void BinarySearchTree::m_preOrderTraverse(Node* node) {
   if (node) {
      m_traversalBuffer.push_back(node->key);
      m_preOrderTraverse(node->left);
      m_preOrderTraverse(node->right);
```

```
    }
}

BinarySearchTree::Status BinarySearchTree::remove(int key) {
    auto deleteNode = m_search(m_root, key);
    if (!deleteNode) { // 'key' to delete does not exist
        return Status::INVALID_ARGUMENT;
    }
    m_remove(deleteNode);
    return Status::FOUND;
}

void BinarySearchTree::m_remove(Node* node) {
    // CASE 1/2: 'node' has zero or one child
    if (!node->left) {
        m_transplant(node, node->right);
    }
    else if (!node->right) {
        m_transplant(node, node->left);
    }

    // CASE 3: 'node' has two children
    else {
        auto successor = m_successor(node);
        if (successor != node->right) { // 'node' has a right subtree > 1 child
            // transplant successor with right child
            m_transplant(successor, successor->right);

            // update successor's new inherited right tree
            successor->right = node->right;
            successor->right->parent = successor;
        }
        // transplate successor to 'node' itself
        m_transplant(node, successor);

        // update successor's new inherited left tree
        successor->left = node->left;
        successor->left->parent = successor;
    }
    delete node;
}

void BinarySearchTree::m_transplant(Node* remove, Node* merge) {
    if (!remove->parent) { // edge case where 'remove' is the root
        m_root = merge;
    }
    else if (remove->parent->left == remove) { // 'remove' is a left child
        remove->parent->left = merge;
    }
    else { // 'remove' is a right child
        remove->parent->right = merge;
    }
    if (merge) { // 'merge' exists
        merge->parent = remove->parent;
    }
};
```



```
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file binary_search_tree.h
* @assignment 7: Binary Search Tree
*/

#pragma once

#include <utility>
#include <vector>

class BinarySearchTree {

    struct Node; // Forward declaration

public:

    /**
    * 'Status' enumerators are returned by public methods and convey information
    * about the result of an operation.
    */
    enum Status {
        DNE = 0, // The result does not exist
        FOUND, // The result exists
        INVALID_ARGUMENT // Operation to calculate result could not execute
    };

    /**
    * 'Result' objects are returned by public methods and contain an integer
    * output whose validity depends on its associated 'Result'.
    */
    typedef std::pair<Status, int> Result;

    /**
    * 'BinarySearchTree' default constructor.
    */
    BinarySearchTree();

    /**
    * 'BinarySearchTree' destructor.
    */
    ~BinarySearchTree();

    /**
    * Finds the smallest key in the tree.
    * @return 'Result' containing min key in tree and found status, else DNE
    * 'Result' if tree is empty
    */
    Result min();

    /**
    * Finds the largest key in the tree.
    * @return 'Result' containing max key in tree and found status, else DNE
    * 'Result' if tree is empty
    */
    Result max();

    /**
    * Finds the successor.
    * @param 'key' key to find successor of
    * @return 'Result' indicating if 'key' did not exist, if successor to 'key'
    * did not exist, or if successor to 'key' did exist with the successor key
    */
    Result successor(int key);
```



```
    /**
    * Finds the predeccessor.
    * @param 'key' key to find predecessor of
    * @return 'Result' indicating if 'key' did not exist, if predecessor to
    * 'key' did not exist, or if predecessor to 'key' did exist with the
    * predecessor key
    */
    Result predecessor(int key);

    /**
    * Searchs for a key.
    * @param 'key' key to search for
    * @return 'Status' object indicating if 'key' was found
    */
    Status search(int key);

    /**
    * Removes a key from the tree.
    * @param 'key' key to remove
    * @return 'Status' objecting indicating if 'key' did not exist or if it was
    * removed
    */
    Status remove(int key);

    /**
    * Inserts a key into the tree.
    * @param 'key' key to insert
    */
    void insert(int key);

    /**
    * Invokes an in order traversal from 'm_root'.
    * @return 'std::vector' containing the keys of the tree in order
    */
    std::vector<int> inOrderTraverse();

    /**
    * Invokes a pre order traversal from 'm_root'.
    * @return 'std::vector' containing the keys of the tree pre order
    */
    std::vector<int> preOrderTraverse();

    /**
    * Invokes a post order traversal from 'm_root'.
    * @return 'std::vector' containing the keys of the tree post order
    */
    std::vector<int> postOrderTraverse();

private:

    /**
    * Searchs for a 'Node'.
    * @param 'node' node of root of valid binary search tree
    * @param 'key' key of node to find
    * @return 'Node*' of node with 'key' in 'node', else nullptr if it does not
    * exist
    */
    Node* m_search(Node* node, int key);

    /**
    * Finds a 'Node' with the smallest key.
    * @param 'node' node of root of valid binary search tree
    * @return 'Node*' of node with smallest key in 'node'
    */
    Node* m_min(Node* node);
```



```
    /**
    * Finds a 'Node' with the largest key.
    * @param 'node' node of root of valid binary search tree
    * @return 'Node*' of node with smallest key in 'node'
    */
    Node* m_max(Node* node);

    /**
    * Finds the successor of a 'Node'.
    * @param 'node' node to find successor of
    * @return 'Node*' successor to 'node'
    */
    Node* m_successor(Node* node);

    /**
    * Finds the predecessor of a 'Node'.
    * @param 'node' node to find predecessor of
    * @return 'Node*' predecessor to 'node'
    */
    Node* m_predecessor(Node* node);

    /**
    * Removes all 'Node' objects including and under a provided 'Node';
    * invoked by 'Binary Search Tree' destructor.
    * @param 'node' base node to destruct from
    */
    void m_destruct(Node* node);

    /**
    * Connects the parent of a 'Node' to another 'Node'.
    * @param 'remove' pointer to Node to connect its parent to 'merge'
    * @param 'merge' pointer to Node to connect itself to the parent of
    * 'upper'
    */
    void m_transplant(Node* remove, Node* merge);

    /**
    * Deletes a node.
    * @param 'node' node to delete
    */
    void m_remove(Node* node);

    /**
    * Traverses the tree in order, filling 'm_traversalBuffer'.
    * @param 'node' node to begin traversal at
    */
    void m_inOrderTraverse(Node* node);

    /**
    * Traverses the tree pre order, filling 'm_traversalBuffer'.
    * @param 'node' node to begin traversal at
    */
    void m_preOrderTraverse(Node* node);

    /**
    * Traverses the tree post order, filling 'm_traversalBuffer'.
    * @param 'node' node to begin traversal at
    */
    void m_postOrderTraverse(Node* node);

    std::vector<int> m_traversalBuffer; // Temp array storage for traversals

    Node* m_root = nullptr; // Root of 'Binary Search Tree'

    /**
```

```
   * 'Node' objects are units that compose a 'Binary Search Tree'; they contain
   * keys and can connect to their parent and children 'Node' objects.
   */
   struct Node {
      Node() = default;
      Node(int key) : key(key) {}
      Node* left = nullptr;
      Node* right = nullptr;
      Node* parent = nullptr;
      int key;
   };
};
```



```
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file main.cpp
* @assignment 7: Binary Search Tree
*/

/**
* This program implements a Binary Search Tree (BST). A "session trace" is read
* from STDIN in 'main' that allows the user to interact with the BST; the
* results of these operations are outputted to STDOUT. The program was tested by
* hand and with valgrind for memory leaks and appears to be working to all
* specifications.
*/

#include <iostream>
#include <utility>
#include <string>
#include <sstream>

#include "binary_search_tree.h"

typedef std::pair<std::string, int> Command; // From parsed session trace

typedef BinarySearchTree BST; // For brevity

/**
* Processes a trace into a command.
* @param 'trace' trace of command
* @return 'Command' parsed from trace
*/
Command processTrace(std::string trace);

/**
* Executes search on BST and prints results.
* @param 'tree' BST
* @param 'key' to search for
*/
void search(BST& tree, int key);

/**
* Executes insert on BST and prints results.
* @param 'tree' BST
* @param 'key' key to insert
*/
void insert(BST& tree, int key);

/**
* Executes remove on BST and prints results.
* @param 'tree' BST
* @param 'key' key to remove
*/
void remove(BST& tree, int key);

/**
* Executes min on BST and prints the min.
* @param 'tree' BST
*/
void min(BST& tree);

/**
* Executes max on BST and prints the max.
* @param 'tree' BST
*/
void max(BST& tree);
```



```

/**
* Executes predecessor and prints results.
* @param 'tree' BST
* @param 'key' key to predecessor of
*/
void predecessor(BST& tree, int key);

/**
* Executes successor and prints results.
* @param 'tree' BST
* @param 'key' key to find successor of
*/
void successor(BST& tree, int key);

/**
* Traverses and prints BST in order.
* @param 'tree' BST
*/
void inorder(BST& tree);

/**
* Traverses and prints BST post order.
* @param 'tree' BST
*/
void postorder(BST& tree);

/**
* Traverses and prints BST pre order.
* @param 'tree' BST
*/
void preorder(BST& tree);

/**
* Prints a vector containing traversal elements to STDOUT.
* @param 'vec' vector containing traversal elements
*/
void printTraversal(std::vector<int> vec);

void insert(BST& tree, int key) {
   tree.insert(key);
   std::cout << "inserted " << key << "." << std::endl;
}

void search(BST& tree, int key) {
   auto status = tree.search(key);
   if (status == BST::Status::FOUND) {
      std::cout << key << " found." << std::endl;
      return;
   }
   std::cout << key << " not found." << std::endl;
}

void min(BST& tree) {
   auto result = tree.min();
   if (result.first == BST::Status::FOUND) {
      std::cout << "min is " << result.second << "." << std::endl;
      return;
   }
   std::cout << "tree empty." << std::endl;
}

void max(BST& tree) {
   auto result = tree.max();
   if (result.first == BST::Status::FOUND) {
      std::cout << "max is " << result.second << "." << std::endl;
```



```
      return;
   }
   std::cout << "tree empty." << std::endl;
}

void successor(BST& tree, int key) {
   auto result = tree.successor(key);
   if (result.first == BST::Status::FOUND) {
      std::cout << key << " successor is " << result.second << "."
      << std::endl;
   }
   else if (result.first == BST::Status::DNE) {
      std::cout << "no successor for " << key << "." << std::endl;
   }
   else {
      std::cout << key << " not in tree." << std::endl;
   }
}

void predecessor(BST& tree, int key) {
   auto result = tree.predecessor(key);
   if (result.first == BST::Status::FOUND) {
      std::cout << key << " predecessor is " << result.second << "."
      << std::endl;
   }
   else if (result.first == BST::Status::DNE) {
      std::cout << "no predecessor for " << key << "." << std::endl;
   }
   else {
      std::cout << key << " not in tree." << std::endl;
   }
}

void preorder(BST& tree) {
   std::cout << "preorder traversal:";
   printTraversal(tree.preOrderTraverse());
}

void postorder(BST& tree) {
   std::cout << "postorder traversal:";
   printTraversal(tree.postOrderTraverse());
}

void inorder(BST& tree) {
   std::cout << "inorder traversal:";
   printTraversal(tree.inOrderTraverse());
}

void printTraversal(std::vector<int> vec) {
   std::cout << std::endl;
   for (size_t i = 0; i < vec.size(); i++) {
      if (i != 0) {
         std::cout << " " << vec[i];
      }
      else {
         std::cout << vec[i];
      }
   }
   std::cout << std::endl;
}

void remove(BST& tree, int key) {
   auto status = tree.remove(key);
   if (status == BST::Status::FOUND) {
      std::cout << "deleted " << key << "." << std::endl;
      return;
```

```
    }
    std::cout << "delete " << key << " - not found." << std::endl;
}

Command processTrace(std::string trace) {
    // Remove after comment if there is one
    size_t commentIndex = trace.find("#");
    if (commentIndex != std::string::npos) {
        trace.erase(commentIndex);
    }

    // Parses command type and argument
    std::stringstream ss(trace);
    std::string commandType;
    int argument;
    ss >> commandType;
    ss >> argument;

    return {commandType, argument};
}

int main() {
    BST tree;
    std::string trace;
    while (std::getline(std::cin, trace)) {
        Command command = processTrace(trace);
        auto commandType = command.first;
        auto commandArgument = command.second;
        if (commandType == "search") {
            search(tree, commandArgument);
        }
        else if (commandType == "insert") {
            insert(tree, commandArgument);
        }
        else if (commandType == "delete") {
            remove(tree, commandArgument);
        }
        else if (commandType == "min") {
            min(tree);
        }
        else if (commandType == "max") {
            max(tree);
        }
        else if (commandType == "predecessor") {
            predecessor(tree, commandArgument);
        }
        else if (commandType == "successor") {
            successor(tree, commandArgument);
        }
        else if (commandType == "inorder") {
            inorder(tree);
        }
        else if (commandType == "postorder") {
            postorder(tree);
        }
        else if (commandType == "preorder") {
            preorder(tree);
        }
    }
} // End of main
```



```
p7: main.o binary_search_tree.o
	g++ -o p7 main.o binary_search_tree.o

main.o: main.cpp
	g++ -c main.cpp

binary_search_tree.o: binary_search_tree.cpp binary_search_tree.h
	g++ -c binary_search_tree.cpp

clean:
	rm -f p7 *.o *~
```